Verity™

Verity
Collection Reference

Version 6.1

November 30, 2005
Part Number DM0708

Verity, Incorporated
894 Ross Drive
Sunnyvale, California 94089
(408) 541-1500

Verity Benelux BV
Coltbaan 31
3439 NG Nieuwegein
The Netherlands

Notice to Government End Users

If this product is acquired under the terms of a **DoD contract:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of 252.227-7013. **Civilian agency contract:** Use, reproduction or disclosure is subject to 52.227-19 (a) through (d) and restrictions set forth in the accompanying end user agreement. Unpublished-rights reserved under the copyright laws of the United States. Verity, Inc., 894 Ross Drive Sunnyvale, California 94089.

11/29/05

Contents

PART I STYLE-FILE REFERENCE

PART II COLLECTION TOOLS REFERENCE

APPENDIXES

Figures, Tables, and Listings

Preface

The *Verity K2 Collection Reference Guide* describes the architecture and design of Verity® collections. Collections represent groups of documents that can be searched by one or more Verity applications.

This book is for Verity administrators. It is intended for readers who have a Verity installation and who are familiar with the basic concepts of search applications

This preface contains the following sections:

- Using This Book
- Related Documentation
- Verity Technical Support

Using This Book

This section briefly describes the organization of this book and the stylistic conventions it uses.

Version

The information in this manual is current as of K2 Enterprise version 5.5. The content of the manual was last modified November 30, 2005. Corrections or updates to this information may be available through the Verity Customer Support site; see "Verity Technical Support" on page 23.

Organization of This Book

This book includes the following chapters and appendixes:

Part I: Style-File Reference

- Chapter 1, "Configuring and Managing Collections," gives overview information about Verity® collections and how to build them for your Verity application.
- Chapter 2, "Configuring Gateways," describes gateway features and the File System gateway and HTTP gateway in detail.
- Chapter 3, "Setting Indexing and Search Policies," describes how you can customize the Verity engine's behavior by assigning an indexing mode or by defining what is included in the document hit count.
- Chapter 4, "Filtering and Formatting Documents," describes how to configure document filters to affect search engine operations, like indexing and displaying documents stored in a variety of native formats.
- Chapter 5, "Defining Collection Fields," describes field definitions and the schema for a collection's document table.
- Chapter 6, "Populating Collection Fields," describes methods for populating fields.
- Chapter 7, "Defining Document Zones" on page 179 describes how to enable zone searching through the use of the zone filter for: SGML, HTML, internet-style email messages, internet-style Usenet news articles.
- Chapter 8, "Tuning Collections," describes how to customize collection indexes using optional style files.

Part II: Collection-Tool Reference

- Chapter 9, "Command-Line Tool Summary," lists the command-line tools provided with K2 and links to their descriptions.
- Chapter 10, "Using mkvdk," describes how to use the principal command-line tool for creating and maintaining collections.

- Chapter 11, "Using Bulk Insert Files," documents BIF format and explains how to use BIFs to modify collection content.
- Chapter 12, "Using Other Collection Tools," describes the command-line tools (other than `mkvdk`) available for your use.

Appendixes:

- Appendix A, "Supported Document Formats," describes the document types recognized by the universal filter and what the file name extensions are, if any.
- Appendix B, "Supported Date Formats," describes the variety of date formats that can be recognized during indexing and searching.
- Appendix C, "Supported Regular Expressions," describes the syntax of regular expressions.
- Appendix D, "Collection Limits," describes certain ranges and limits for collections.

Stylistic Conventions

The following stylistic conventions are used in this book.

Convention	Usage
Plain	Narrative text.
Bold	User-interface elements in narrative text: ■ Click **Cancel** to halt the operation.
Italics	Book titles and new terms: ■ For more information, see the *Verity K2 Getting Started Guide*. ■ An *index* is a Verity collection, parametric index, or recommendation index.
`Monospace`	File names, paths, and code: ■ The `name.ext` file is installed in: `C:\Verity\Data\`
`Monospace italic`	Replaceable strings in file paths and code: ■ `user` *`username`*
`Monospace bold`	Data types and required user input: ■ **`SrvConnect`** A connection handle. ■ In the **User Interface** text box, type **`user1`**.

The following conventions are used in this book to describe command-line tool syntax.

Convention	Usage
`[ optional ]`	Brackets describe optional syntax, as in `[ -create ]` to specify a non-required option.
`|`	Bars indicate "either | or" choices, as in `[ option1 ] | [ option2 ]` In this example, you must choose between option1 or option2.
`{ required }`	Braces describe required syntax in which you have a choice and that at least one choice is required, as in `{ [ option1 ] [ option2 ] }` In this example, you must choose either `option1`, `option2`, or both options.
`required`	Absence of braces or brackets indicates required syntax in which there is no choice; you must enter the required syntax without modification, as in `mkre`.
`variable`	Italics specify variables to be replaced by actual values, as in `-merge` *`filename1`*
`...`	Ellipses indicate repetition of the same pattern, as in `-merge` *`filename1`*`,` *`filename2`* `[,` *`filename3`* `... ]` where the ellipses specify `,` *`filename4`*, and so on.

Use of punctuation, such as single and double quotes, commas, periods, and such, indicate actual syntax; they are not part of the syntax definition.

Related Documentation

The collection administrator may need to use some or all of the following books in conjunction with this manual to create and configure collections:

- *Verity K2 Dashboard Administrator Guide*
- *Verity Command-Line Indexing Reference*
- *Verity Gateway Guides* (Notes, ODBC, Exchange, Documentum)
- *Verity Knowledge Console Guide*
- *Verity Intelligent Classification Guide*

Verity Technical Support

Verity Technical Support exists to provide you with prompt and accurate resolutions to difficulties relating to using Verity software products. You can contact Technical Support using any of the following methods:

Telephone: (403) 294-1107

Fax: (403) 750-4100

Email: tech-support@verity.com

Web: `http://www.verity.com`

Product documentation, release notes, and document updates are available on the Verity Customer Support Site, at

`https://customers.verity.com`

It is recommended that you periodically check the Customer Support site for the existence of updates to this and other Verity product documents.

Access to the contents of the Customer Support site requires a user name and password. To obtain a user name and password, follow the signup instructions on the Customer Support site home page. You will need to supply your Verity entity ID and Verity license key.

PART I

Style-File Reference

1

Configuring and Managing Collections

This chapter provides overview information about Verity® collections and how to configure them to fit the needs of your Verity application.

This chapter introduces these basic concepts about Verity collections:

- About Collections
- Internal Collection Structure
- About Style Files
- About Style Sets
- Collection-Management Tools

About Collections

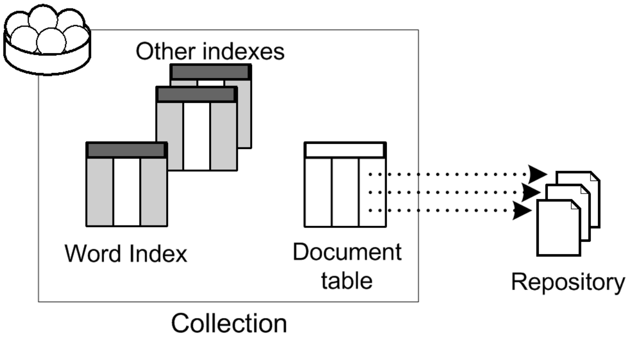
A Verity *collection* is the fundamental structure underlying the powerful and sophisticated search capabilities demonstrated by Verity applications. A collection consists of information about a particular set of documents that are available for searching and viewing. You can implement a range of application features to support searching, viewing and navigating your collection and its documents.

A collection represents *metadata*, or data about data, applied to a a set of documents. The metadata in a collection includes indexes to aid searching plus tables defining document locations and document-field content.

Collection Content

A collection includes metadata that references a *repository* (a logical group of documents) plus metadata that describes document content. The specific information stored for a collection includes a word index and various other indexes, plus a document table containing document field information and logical pointers to the document files (Figure 1-1).

Figure 1-1 Collection indexes and pointers to repository documents

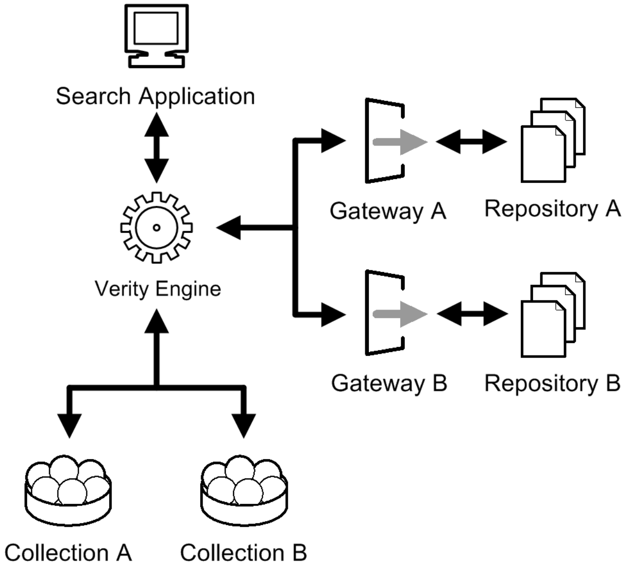


The additional indexes optimize searching or offer advanced search features such as category search, parametric search, adaptive ranking, and topic search. See “Internal Collection Structure” on page 41 for more details.

Gateway Access to Repositories

A Verity application can control numerous collections, with the number of documents associated with each collection optimized for searching. As shown in Figure 1-2, the same application can use different collections to search documents in repositories of different types, such as a Web site versus a database.

Figure 1-2 Search access to multiple types of repositories



The Verity engine uses *gateways* to access repositories. A gateway is a software module that provides access to a repository of a given type. Verity provides gateways for accessing the following types of repositories:

- File system (Windows or UNIX)
- Web (HTTP)
- Lotus Notes
- ODBC-compliant databases
- Documentum
- Microsoft Exchange

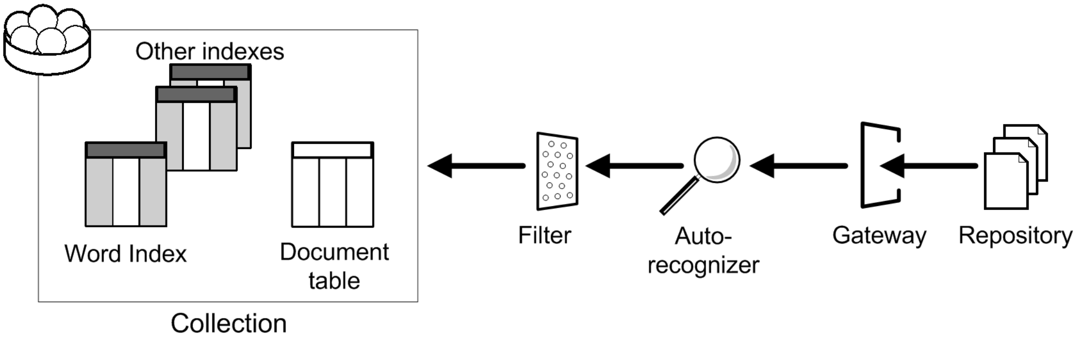
Your Verity license agreement determines which of these gateways are available for your use. See"Configuring Gateways" on page 61 for more information on using and configuring Verity gateways.

Note Verity customers can design and build custom gateways to provide access to still other types of repositories. See *Verity Gateway Developer's Kit Programming Reference* for more information.

Universal Document Support

Verity applications make use of *document filters* (Figure 1-3) to support indexing and viewing of documents in many different formats. Many filter types are available.

Figure 1-3 Using filters to access document content in multiple formats



During indexing, a Verity module called the *autorecognizer* first identifies the format of the document, then the appropriate document filter for that format converts document content from its native format into a plain-text format that can be processed to create the word index and other indexes.

During document viewing, the filter performs the same task so that the Verity application can display the document to the user. For documents in standard word-processing formats such as Microsoft Word and Lotus 1-2-3, the filters can generate high-fidelity WYSIWYG document representations for viewing.

The available document filters include these:

- The Verity universal filter, which calls other filters to convert a wide variety of documents.

  You can configure this filter to control its behavior.
  (See "The Universal Filter" on page 104.)

- Verity KeyView filters for numerous native formats.
  (See "The KeyView Filters" on page 123 and "Supported Document Formats" on page 357.)

- The Verity XML filter for XML documents.
  (See "The XML Filter" on page 132.)

- The Verity zone filter for HTML and tagged ASCII formats.
  (See "Defining Document Zones" on page 179.)

- Verity PDF filters for Adobe Acrobat documents. There are two PDF filters available; one using Verity's viewing technology, and one based on Adobe libraries. (See "The KeyView PDF Filter" on page 124.)

The universal filter uses any of the other filters as "helper" subfilters to actually perform the conversion. For example, If you use the universal filter, Microsoft Word documents are automatically converted with a KeyView filter, HTML documents are automatically converted with the zone filter, and PDF documents are automatically converted with the PDF filter. You do not need to restrict a collection to a single type of document.

Note Using the VDK API, you can develop your own custom filters.

How to Build a Collection

You typically build Verity collections using one of these approaches:

- Using the graphical interface of the K2 Dashboard.
- Running the command-line tool `mkvdk`.
- Running either of the command-line spidering tools Verity Spider and K2 Spider.
- Writing and executing a program that uses the Verity C or Java APIs.

The detailed steps required to create and index a collection vary with the tools that you use, but the overall process is the same.

1. **Specify a style set.**

   Establish a set of *style files*, configuration files that will control indexing behavior and collection characteristics.

   - With the K2 Dashboard, you can set up the style files from within the collection-creation dialogs.
   - With the command-line tools, you set up the style files separately, before creating the collection.

   You configure style files either through manual editing or by using the StyleSet Editor, a graphical tool for manipulating style files. (For a simple collection using files accessed through the file system and without using spidering, you can use `mkvdk` and no style-file setup is necessary.)

   For more information, see "About Style Files" on page 47 and "About Style Sets" on page 53.)

2. **Create the collection structure.**

   - With the K2 Dashboard, you create the collection using the Collection Wizard or though the Create Collection dialog.
   - With the `mkvdk` command-line tool, you use the `-create` option. With the spider tools, you use the `-collection` command option.

     (With the command-line tools, you also specify other options along with `-create`, such as the `-style` option to specify the set of style files to use for configuration.)
   - Your Verity application can create collections programmatically, using the VDK or VAdministration APIs.

   For more information on creating a collection structure, see these documents:

   - K2 Dashboard: the creating collections chapter of the *Verity K2 Dashboard Administrator Guide*
   - `mkvdk`: "Creating and Indexing Collections" on page 281.
   - K2 Spider: the running K2 Spider Server chapter of *Verity Command-Line Indexing Reference*.
   - Verity Spider: the Verity Spider reference chapter of *Verity Command-Line Indexing Reference*.
   - C APIs: Collection Maintenance Suite chapter of *Verity Developer's Kit Programming Reference*.
   - Java APIs: VAdministration Javadocs.

3. **Index documents into the collection.**

   - With the K2 Dashboard, you perform indexing through an *indexing job*, a task that you can set up using the Collection Wizard or through the Create Job dialog.
   - With `mkvdk`, you can either specify a *bulk insert file* (BIF), which lists the specific documents that are to be indexed, or list the documents explicitly on the command line itself.
   - With Verity Spider and K2 Spider, you use command-line options to set up crawling and indexing tasks across multiple directories, or alternatively you specify the location of a bulk insert file.
   - Your Verity application can index documents into a collection programmatically, using the collection-indexing API.

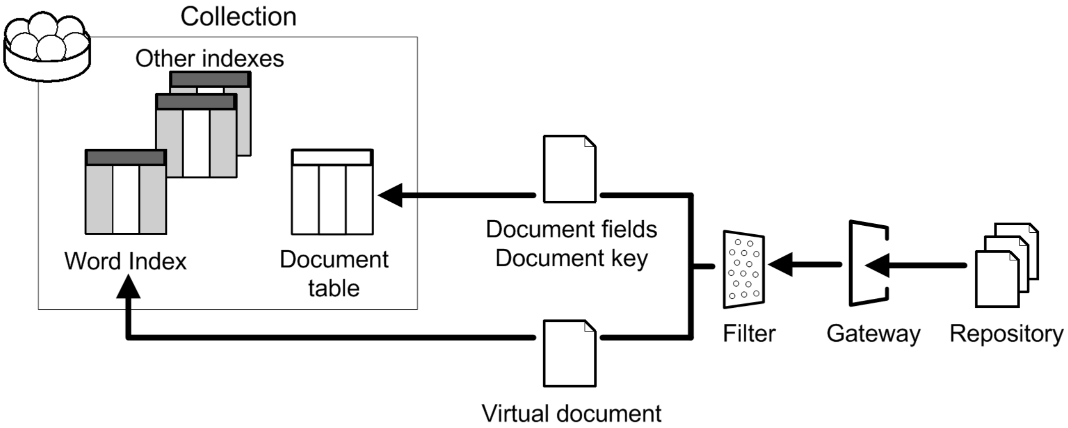
For more information on indexing, see these documents:

- K2 Dashboard: the creating collection-indexing jobs chapter of the *Verity K2 Dashboard Administrator Guide*
- mkvdk: "Creating and Indexing Collections" on page 281.
- K2 Spider: the running K2 Spider Server chapter of *Verity Command-Line Indexing Reference*.
- Verity Spider: the Verity Spider reference chapter of *Verity Command-Line Indexing Reference*.
- APIs: The collection-indexing suite chapter of the *Verity K2 Collection Indexing Programmer Guide*. See also the *Verity K2 Spider Programming Guide* and the *Verity Developer's Kit Programming Reference* (for VdkCollectionSubmit).

The Indexing Process

Figure 1-4 summarizes what happens when a document is indexed into a collection. The steps are the same whether a collection is being indexed for the first time or whether an individual document is being added (inserted) into an existing collection.

Figure 1-4 Indexing a collection



1. The indexing job, spidering tool, or mkvdk goes through the appropriate gateway to retrieve a document from the repository.

2. The proper document filter (or filters) converts the retrieved document into a *virtual document*, a plain-text representation of the document's content. (You can configure the virtual document, as noted in "Configuring the Virtual Document" on page 35.)

   The filter also extracts field information, if any, from the document, and in addition assigns it a *document key* (`VdkVgwKey`), a gateway-specific pointer to the document that serves as a unique document identifier.

3. The document filter inserts the document's `VdkVgwKey` into the collection's document table. It also inserts field information associated with the document into the collection fields in the document table. (You can configure which fields are created, and how they are populated, as noted in "Defining and Populating Collection Fields" on page 35).

4. A portion of the Verity engine called the *tokenizer* processes the virtual document, producing index information for its entire content. That information is inserted into the collection's word index.

5. Depending on what other indexes have been specified for the collection, the Verity engine may extract information from the virtual document or the document fields to create or update those indexes.

Note that a document's text content does not itself become part of the collection; only the detailed index information about it goes into the collection. Selected document field content, however, is written into the collection.

The Verity engine may process the information differently, or create different indexes, depending on what indexing mode has been selected; see "Setting the Indexing Mode" (next).

The indexing process is highly dependent on the language and character set of the documents being indexed. Indexing always occurs in the context of a particular Verity *locale*, or language definition. Verity applications can index, search, and display documents in collections of many different locales. See the*Verity Locale Configuration Guide* for more information.

Setting the Indexing Mode

As part of setting up the style set before indexing, you can specify an indexing mode. The indexing mode assigned to a collection determines performance and scheduling of indexing tasks performed by the Verity engine. For example, special indexing modes are available for indexing from bulk insert files, for indexing newsfeeds, and for creating indexes optimized for fast searching.

Before you index the collection, you assign the indexing mode in the collection's style file (`style.plc`). If no mode is assigned, a default generic indexing mode is used. For more information about using indexing modes, see "Setting Indexing and Search Policies" on page 83.

Configuring the Virtual Document

The definition of a virtual document includes:

- A document layout definition for the document body (the textual content)
- A document filter specification that identifies the filters to be used (universal filter plus format filters)

The virtual document's specification is defined in the style file `style.dft`. By default, the document layout consists of a single field that holds nothing but the document's complete text content, formatted as if written into a table, one character per cell, beginning at row 1, column 1.

Before you index the collection, you can customize `style.dft` to specify which parts of the collections's documents will be available for searching, for viewing, or for summarization (creating document summaries). For example, you can

- allow for the inclusion of various document-related fields (such as title or author) in the document content.
- define multiple document layout fields, which means that you can combine multiple files into a single virtual document.
- define certain document fields as zones (see "Defining Document Zones" on page 179), so that they will be indexed and available for zone searching.

For more information about configuring the virtual document, see "The Virtual Document" on page 96.

Defining and Populating Collection Fields

Besides the indexes that map to document content, a collection can include any number of fields. Verity allows for the inclusion of several types of fields, in order to support various information management needs.

- **Internal fields.** These fields are defined and populated by the Verity engine. Internal fields include the document key (`VdkVgwKey`) and other fields used to control search and viewing operations. Internal fields are defined in the style file `style.ddd`, which should not be edited. See "Internal Fields (style.ddd)" on page 149.

- **Standard fields.** These fields are defined in the `style.sfl` file and populated automatically by the Verity engine and document filters. Standard fields include `Title`, `Author`, `Chariest`, and `Date`.

  You can customize the set of standard fields in your collection by commenting or uncommenting individual lines in `style.sfl`. See "Standard Fields (style.sfl)" on page 152.

- **Custom fields.** These are application-defined fields. Typically, these fields are populated at indexing time by one of these methods:

  - Importing their content in a bulk insert file (see "Using Bulk Insert Files" on page 313).

  - Extracting field data from document content, using `mkvdk` and the style file `style.tde` (see "Extracting Field Values" on page 168).

  - Parsing metatags, using the zone filter or XML filter (see "Defining Zones as Collection Fields" on page 210 and "The XML Filter" on page 132).

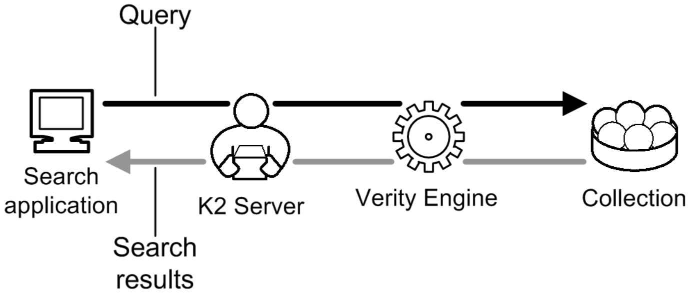
  You define your collection's set of custom fields in the style file `style.ufl`. See "User-Defined Fields (style.ufl)" on page 156.

- **External fields.** These fields exist as entities in another application and their contents are copied to Verity fields through the use of a Verity gateway. External fields are defined in the gateway style files. See "Gateway Style Files" on page 47.

Searching a Collection

A Verity application uses collections to provide users with fast search and browse access to large numbers of documents. At its simplest (Figure 1-5), the search application's function is to forward user queries to the collection and present results back to the user.

Figure 1-5 Searching a collection



Query Handling

Verity applications can use the Verity Query Language (VQL) to search collections. VQL is a powerful search language that is described in the *Verity Query Language and Topic Guide*.

The application's query can be sent through a K2 Server or K2 Broker, which may distribute the search to multiple collections. Setting up K2 Servers and K2 Brokers is discussed in the *Verity K2 Dashboard Administrator Guide*.

The Verity engine interprets the query, locates documents that are relevant to the query, and returns a list of the most relevant documents as search results. The search results include document field information (such as title and document summary) and the document key (written into a link that allows the application to use the Verity engine to retrieve the document from the repository).

A search can be applied to document content or to any of the collection fields. Note that the actual document repository is not accessed during a typical search; only the collection itself is processed. (For secure collections, the repository might be accessed to determine which search results can be displayed to the user.)

Queries can be stored as *topics*. Topics are a search convenience, in that a user can effectively invoke a long, complex query simply by entering the name of a topic into the search application. A number of topics can be grouped into a *topic set* and attached to a collection, and even indexed for faster searching (see "Collection Indexes" on page 43). Topic sets can also be combined into *knowledge bases*.

Topics and topic sets are described in the *Verity Query Language and Topic Guide*. Knowledge bases are described in the *Verity Developer's Kit Programming Reference*.

Specifying Search Characteristics

You can tune the search characteristics of your collection in several ways by making changes to the following style files before indexing:

- `style.stp`. Use this file to keep extraneous words out of the collection's word index, in order to speed search. See "Defining Indexing Stop Words (style.stp)" on page 257.
- `style.go`. Use this file to specify the full set of words that are allowed into the collection's word index. See "Defining Indexing Go Words (style.go)" on page 259.
- `style.prm`. Use this file to specify that additional indexes (such as SOUNDEX) and structures (such as feature vectors) be built into the collection. These features support additional types of search.

  See "Collection Indexes" on page 43 for a discussion of these additional features. See "Adding Extra Collection Capabilities (style.prm)" on page 233 for more information about `style.prm`.
- `style.ufl`. Besides defining the custom fields for the collection, this style file allows you to specify that the content of any of the custom fields should itself be indexed, to make field searches much faster. See "Indexed Field Type" on page 231.

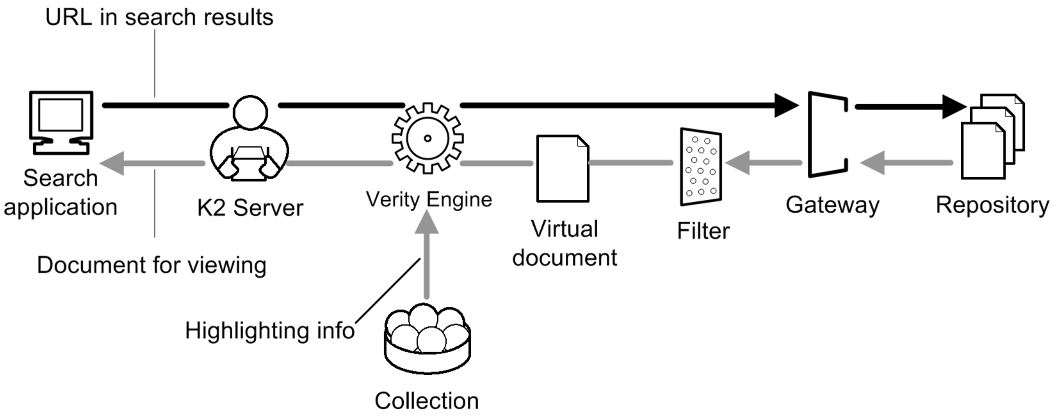
Viewing Collection Documents

After a search, if the user of a Verity application wants to view one of the returned documents, the user typically clicks a link on the search-results page. The application then displays the document in one of two ways:

- The application retrieves the document directly and displays it through its own display capabilities or by launching another application, such as a Web browser.
- The application uses Verity to retrieve and display the document.

If the application uses Verity, it follows the search-results link (which is based on the document's `VdkVgwKey`) and uses the gateway to retrieve the document from the repository (Figure 1-6).

Figure 1-6 Using Verity to view a document from search results



The appropriate document filters are applied to the document to reproduce the virtual document, then highlighting information is extracted from the collection and applied. The document is then returned to the application for display to the user (possibly in a Web browser), with the user's search terms highlighted.

Note Highlighting of search terms is not available if the application itself retrieves and displays the document.

Note that, apart from providing information for highlighting, the collection itself is not accessed for document viewing.

To use the Verity viewing service, your application calls the Viewing Service API, described in the *Verity K2 Viewing Service Programming Guide* and in the VView Javadocs. The document formats that can be displayed by the Verity Viewing Service are listed in "Supported Document Formats" on page 357.

Your application can use separate gateways for indexing and viewing. For example, if indexing with the Lotus Notes gateway, it might be more convenient to use the HTTP gateway for viewing. This requires using two separate gateway style files; see "Using Separate Gateways for Indexing and Viewing" on page 63 for details.

Note You can customize the appearance of the document summaries that appear in search results. Using the style file `style.fxs`, you can specify words that are to be excluded during feature extraction, so that those words do not appear in document summaries and clusters. See "Defining Feature-Extraction Stop Words (style.fxs)" on page 259.

Optimizing Collections

The Verity collection-optimization features allow you to configure a collection for the best possible search performance. There are two instances in which you would want to do this:

1. When a collection becomes static (will never need re-indexing) and you want to publish it for general use. Also, whenever you are distributing a collection on a CD-ROM disc.
2. When a periodically changing collection need regular optimizations, either to improve search performance or to recover space still occupied by deleted document indexes.

There are a few different ways to optimize collections. Information about the available options is provided in "Optimizing Collections" on page 289.

Concurrent Access and Updating

The Verity collection architecture includes the following features that support high availability and efficient optimization:

- Collections can be updated continuously while Verity applications are searching them.

  Collections have many features that support consistent and continual access (even while updates are occurring) from multiple applications. Any Verity collection-building application can read and add documents to any collection to which it has valid access. Concurrent access to collections by multiple Verity sessions is enabled through file sharing, and is synchronized through file locking and collection servicing.

- Document indexing can occur continuously and in concert with client operations, since a collection maintains constantly updated data about documents.

  The Verity engine controls an application's access to collections by updating collection metadata on an ongoing basis. When a collection-building application submits documents to be indexed, the Verity engine processes the index request and updates the collection by creating a new set of metadata in order to not disrupt searches. When the update is complete, the new set of metadata is used, and the old set is cleaned up by the general housekeeping and background servicing functions.

- The Verity engine regularly performs housekeeping services on collections to ensure efficient search performance.

  General housekeeping is a service that cleans up the collection files, deletes older disk files that are no longer needed, monitors the collection for problems, and prevents the optional system log from exceeding a certain size. Other background services preserve the integrity of indexed documents while making those documents accessible for searching at all times.

Your application can programmatically invoke housekeeping operations as well. See the auto-administration chapter of the *Verity Developer's Kit Programming Reference* for more information.

- Collections are repaired automatically when certain error conditions occur.

To perform some of these housekeeping operations manually, you can use the `mkvdk` command-line tool (see "Optimizing Collections" on page 289), the `VDKAdminOptimize` function (see *Verity Developer's Kit Programming Reference*), or the `optimize` method of `VCollection` (see the VParametric Javadocs).

Internal Collection Structure

A collection consists of indexes, tables, and optional structures used for specialized functions. Large collections can be subdivided into smaller units. Each collection is stored in a specific, accessible directory structure.

Collection Partitions

When indexing documents, the Verity engine stores document metadata in collection units called *partitions*. Each partition contains metadata for up to 64K documents. The metadata includes the document table and the word index.

Partitions have a scalable architecture that supports incremental searching and display of results. If a collection has multiple partitions, the Verity engine can search one partition at a time and provide search results after each partition search, rather than waiting until the entire collection has been searched. In this way, search performance (time to first results) can be uniform regardless of whether a collection is 1 megabyte or 1 gigabyte in size.

The Document Table

The document table is a collection structure that

- maintains document keys, the pointers to all documents that have been indexed onto the partition.

- defines and holds the contents of the collection fields, a set of metadata fields that apply to each document in the collection.

There is one document table for each partition in the collection.

Document Keys

`VdkVgwKey` is a special field in the document table that is used as the *document key*, a persistent document pointer. When accessing documents through the file-system gateway, the Verity engine by default assigns the document file pathname to `VdkVgwKey`. When accessing documents through other gateways, the Verity engine assigns other identifying information to `VdkVgwKey`.

Note Verity K2 APIs define a document key called `k2DocKey`, which is of the form *`VdkVgwKey@collection`*, where *`collection`* is the name (alias) of the collection to which the document specified in *`VdkVgwKey`* belongs.

Collection Fields

The fields of the document table are persistent or transitory.

- **Persistent Fields.** Persistent fields are persistent between sessions, and they can be internal, standard, custom, or external.
    - □ Internal fields are nonsearchable, non-visible fields internally managed by the Verity engine. Internal fields are defined in the style file `style.ddd`.
    - □ Standard fields are the default visible, searchable fields (such as `author`, `title`, and `summary`) that are defined and populated by Verity for each collection. Standard fields are defined in the style file `style.sfl`.
    - □ Custom fields are visible, searchable fields that are defined and populated by the application for an individual collection. Custom fields are defined in the style file `style.ufl`.
    - □ External fields hold information that is stored in repositories, such as mail applications or relational databases. Examples of external fields might be `userID` or `mailbox` (in situations where that information is kept in the repository but not in the documents themselves). External fields are accessed through Verity gateways. External fields are defined in the style file `style.ufl`.
- **Transitory Fields.** Transitory fields are stored during a session, and they go away when the session is over. As a logical construct, the transitory field is added to the table format. An example of a transitory field is a document's score. Other transitory

fields can be defined using the Verity Developer's Kit, as described in the *Verity Developer's Kit Programming Reference*.

Collection Indexes

At the core of each Verity collection is the *word index* (also called *full word index* or *full inverted index*), a data structure that maps the location of essentially every word (or, optionally, every sentence or paragraph) in every document that has been indexed into the collection. The word index is what makes fast searching possible.

A Verity collection has a separate word index for each partition.

A collection optionally includes the following other indexes and structures. Some are optimization features that can speed searching; some support specialized types of search:

- **zone index.** This index is a word index for portions of the document that are in zones. Zones are created at indexing time from tagged documents in HTML, XML, or Internet Message format, or from other kinds of tagged document for which you have created a custom style file (`style.zon` or `style.xml`). Users can search for terms within specific document zones in a collection.

  You can also manually add zones to a document by changing settings in the style file `style.dft`.

  For more information on zones and their style files, see "Defining Document Zones" on page 179, "The XML Filter" on page 132, and "Using the style.dft File" on page 98.

- **spanning word list**. This optimization feature is an extension of the collection's word index. It is a word index that crosses all partitions, saving the Verity engine the time of having to look up a search term separately in several word indexes. It is especially important for large, multi-partition collections.

  To create a spanning word list for a collection, you can use the K2 Dashboard, `mkvdk`, or an index-optimization API call. See, for example, "Creating a Spanning Word List" on page 292.

- **ngram index**. This optimization feature is a search accelerator for fuzzy searches (queries that use operators such as `<TYPO>` and `<WILDCARD>`). It indexes partial words. Creating an ngram index requires also creating a spanning word list.

  To create an ngram index for a collection, you can use the K2 Dashboard, `mkvdk`, or an index-optimization API call. See, for example, "Creating an ngram Index" on page 292.

- **Stem index**. This index supports stemmed search, in which results that share the same word stem as the search term are returned. For example, a stemmed search for the term "runs" might return results containing "runs", "run", or "running".

A stem index is created by default. To exclude a stem index during collection indexing, you change a setting in the style file `style.prm`. See “style.prm File Syntax” on page 245.

- **Case index**. This index supports case-sensitive search. All case variants of a word are indexed separately so that, for example, a case-sensitive search for “NeXT” doesn’t return instances of “next” or “Next”.

  A case index is created by default. To exclude a case index during collection indexing, you change a setting to the style file `style.prm`. See “style.prm File Syntax” on page 245.

- **Soundex index**. This optional index is required to support “sounds-like” searches, which can return results that have similar spellings to the submitted term. For example, searching for the word “Smith” can return values such as “Smithe”, “Smyth”, or “Smythe”.

  To create a Soundex index for a collection, you make a setting in the style file `style.prm`. See “Soundex Search” on page 242.

- **Topic-set index**. An application can use a topic set to aid searching, so that when users submit a search term that matches the name of a topic in the topic set, that topic is used as the query instead of the search term itself. To speed searching that uses topics, a topic set can be indexed into a collection. When a topic set is indexed, all its queries are run against the collection and the results are stored in the index.

  You create a topic-set index for a collection by using the `mktopics` command-line tool. For more information, see the chapter on building topic sets in the *Verity Query Language and Topic Guide*.

- **Field indexes**. For faster searching of field data, you can specify that certain fields in a collection be themselves indexed. (If a field is not indexed, it is searched with a simple text matching algorithm, which can be very slow in large collections.)

  Field indexes are of two types: indexed (for general text fields) and minmax (for fields that can contain a specific range of values). Setting up either type of field index involves making settings in the style file `style.ufl`. See “Indexing Collection Fields (style.ufl)” on page 231.

- **Feature vectors**. These optional structures are stored in a field in the document table. A feature vector contains key words and phrases (typically nouns or noun phrases) that summarize or characterize the content of the document they are extracted from.

  Feature vectors are prerequisite to supporting certain other search capabilities, such as document summaries, clustering of search results, fast query-by-example search, and recommendation indexes. To enable feature extraction during the indexing of a a collection, you make a setting in the style file `style.prm`. See “Enabling Document Features” on page 236.

Some advanced Verity search capabilities use the following indexes and structures. These indexes are not part of a collection itself, but they can be based in part on information in the collection:

- **Parametric indexes**. These indexes are used for parametric search. A parametric index can be based either on a collection or on XML data. A *parametric tree* is a structure within a parametric index that implements a hierarchical taxonomy that can be browsed or searched. Parametric trees are associated with collections. For more information on parametric indexes and trees, see the *Verity K2 Parametric Developer Guide*.
- **Recommendation indexes**. The Recommendation Engine uses these indexes to make recommendations to a user, based on the user's role and past history. Recommendation indexes are based on entity profiles, some of which rely on collection data. For more information, see the *Verity K2 Recommendation Engine Guide*.
- **Profile nets**. The Verity Profiler Service and the Verity K2 Profiler Service use these structures to evaluate documents for purposes such as document routing. The documents to be evaluated may be obtained through a collection, although documents not i n collections can also be evaluated. For more information, see the *Verity Profiler Programming Guide* or the *Verity K2 Profiler Programming Guide*.
- **Knowledge trees**. Knowledge trees are searchable and browsable hierarchical taxonomies associated with collections. For more information, see the *Verity Intelligent Classification Guide*.

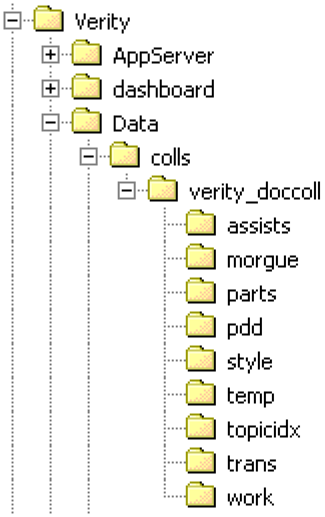
Collection Directory Structure

A collection is a directory structure that stores the indexes, tables, and other information that make up the collection. The directory structure is portable across multiple platforms.

Note Secure collections are not portable across platforms.

Figure 1-7 shows the directory structure of the collection `verity_doccoll`, included with a basic installation of Verity K2. The collection's directory and its contents are created entirely by the collection-building application. Other than the contents of the `style` subdirectory, you should not modify anything in the collection directory.

Figure 1-7 The collection `verity_doccoll` and its subdirectories



Each collection includes the following subdirectories:

- `assists`. Contains files that give general collection information and assist in optimizing searches, such as spanning word lists (`*.wld`), the collection “about” file (`*.abt`), and ngram indexes (`*.ngm`).
- `morgue`. Contains collection files scheduled for deletion.
- `parts`. Contains the internal fields table (`*.ddd)` and the word index (`*.did`) for each of the partitions in the collection.
- `pdd`. Contains the partition map file (`*.pdd`) for the collection.
- `style`. The style set that configures the collection. Contains both gateway style files and collection style files.
- `temp`. Temporary storage used by Verity Spider and K2 Spider.
- `topicidx`. Contains indexed topic sets, if they exist for this collection.
- `trans`. Contains files (`*.trn`) that store information on pending indexing transactions.
- `work`. Temporary storage for files being processed.

About Style Files

A set of style files within each collection determines its configuration options. All documents that are inserted into a collection must be indexed using the same set of style files.

Style files are human-readable text files that contain configuration settings. Verity collections are highly configurable; therefore, a large number of style files, each containing multiple configuration settings, are associated with each collection.

This section summarizes the style files that Verity provides, with links to more detailed information.

Gateway Style Files

Each collection includes one set of gateway style files, which configure the gateway associated with the collection. These are the gateway style files provided for each of the standard Verity gateways:

File-system gateway

`vgwfsys.cfg`

`vgwfsys.gfl`

`vgwfsys.prm`

`vgwfsys.vgw`

HTTP gateway

`vgwhttp.cfg`

`vgwhttp.gfl`

`vgwhttp.prm`

`vgwhttp.vgw`

Lotus Notes gateway

`vgwnotes.cfg`

`vgwnotes.gfl`

`vgwnotes.prm`

`vgwnotes.vgw`

ODBC gateway

`vgwodbc.cfg`

`vgwodbc.prm`

`vgwodbc.vgw`

Documentum gateway

`vgwdctm.cfg`

`vgwdctm.gfl`

`vgwdctm.prm`

`vgwdctm.vgw`

`vgwdctm.cpy`

Exchange gateway

`vgwmsxch.cfg`

`vgwmsxch.gfl`

`vgwmsxch.prm`

`vgwmsxch.vgw`

Fore each gateway, the files have the following purposes:

- `vgw*.cfg`: Sets the indexing parameters for the collection.
- `vgw*.gfl`: Defines internal gateway-related collection fields.

- `vgw*.prm`: Sets various parameters, including settings for viewing documents.
- `vgw*.vgw`: Defines the gateway driver definition file and security module.
- `vgw*.cpy`: Defines collection fields to copy from document metadata in the repository. (Used only by Documentum gateway.)

In general, you should not directly edit any of these style files. For additional overview information on gateways and gateway configuration, see "Configuring Gateways" on page 61. For detailed information on specific gateways and their style files, see these documents:

- HTTP gateway: "Using the HTTP Gateway" on page 66
- File-system gateway: "Using the File System Gateway" on page 79
- Lotus Notes gateway: *Verity K2 Lotus Notes Gateway Guide*
- ODBC gateway: *Verity K2 ODBC Gateway Guide*
- Documentum gateway: *Verity K2 Documentum Gateway Guide*
- Exchange gateway: *Verity K2 Exchange Gateway Guide*

Collection Style Files

Table 1-1 lists the standard style files provided by Verity. Most of these style files are mentioned earlier in this chapter, in conjunction with the collection features they affect. All editable style files are described in more detail later in this book

IMPORTANT Not all style files should be edited or customized—only those identified as editable in Table 1-1. Non-editable style files should remain in the style-set directory without modification.

Table 1-1 Standard style files

File	Explanation
`style.ddd`	**Purpose:** Defines internal collection fields. **Editable?** No. **Description:** The internal field definition file. It defines the collection's internal fields. **See:** "Field Definition Files" on page 148.
`style.dft`	**Purpose:** Document format and filter specification. **Editable?** Yes. **Description:** The document format file. It overrides the default virtual document definition. This file defines the content of the document, which determines how the document will be filtered and viewed. **See:** "The Virtual Document" on page 96 "Using the style.dft File" on page 98.
`style.did`	**Purpose:** Word-index structure. **Editable?** No. **Description:** The internal word index definition file. It is used to generate the word index.
`style.fxs`	**Purpose:** Feature extraction tuning **Editable?** Yes. **Description:** The feature extraction word-exclusion file. In this file, you can specify the words that are not to be considered when creating feature vectors, used for scoring sentences for best-sentence summaries and for forming clusters. **See:** "Defining Feature-Extraction Stop Words (style.fxs)" on page 259.
`style.go`	**Purpose:** Word index tuning **Editable?** Yes. **Description:** The word-inclusion file. It contains a list of the specific words that can be included in a collection's full-word index. Only the words listed can appear in the index. **See:** "Defining Indexing Go Words (style.go)" on page 259.

Table 1-1 Standard style files (continued)

File	Explanation
`style.lex`	**Purpose:** Word index tuning **Editable?** Yes. **Description:** The lexical definition file. You can use it to define non-alphanumeric characters to be interpreted as searchable characters. **Note:** Applies only to older 7-bit locales (such as `english`). Not used by or recommended for current 8-bit locales. **See:** "Defining Indexing Stop Words (style.stp)" on page 257
`style.ngm`	**Purpose:** ngram index specification **Editable?** No. **Description:** The ngram definition file. Contains parameters controlling the ngram indexes. The ngram index is an optional index that helps to accelerate searching.
`style.pdd`	**Purpose:** Collection partitioning **Editable?** No. **Description:** The partition definition file. It contains parameters for configuring and managing the collection's partitions.
`style.plc`	**Purpose:** Indexing mode specification **Editable?** Yes. **Description:** The indexing policy file. You use it to specify the indexing mode to be used for a collection. **See:** "Using style.plc" on page 93
`style.prm`	**Purpose:** Collection tuning and indexing behavior **Editable?** Yes. **Description:** The parameter file. You use it to specify features that you want included in or excluded from the collection indexes. Features include clustering, query-by-example support, and the SOUNDEX operator. The `style.prm` file can be used to specify case-insensitive word indexes to support case-insensitive searching. **See:** "Adding Extra Collection Capabilities (style.prm)" on page 233

Table 1-1 Standard style files (continued)

File	Explanation
`style.pw`	**Purpose:** Indexing access to password-protected documents **Editable?** Yes. **Description:** The password file. Lists passwords required to access particular documents. Generated automatically by the StyleSet Editor, which typically encrypts the passwords. Manual editing or creation usually not required. **See:** "Providing Passwords for Document Access (style.pw)" on page 255
`style.sfl`	**Purpose:** Standard collection fields **Editable?** Yes. **Description:** The standard collection-field list. It defines the collection's document-table schema for the Verity standard fields. By default, these fields are populated in the collection by the universal filter. **See:** "Defining Collection Fields" on page 145 "Contents of style.sfl" on page 153
`style.sid`	**Purpose:** Indexing of topics **Editable?** No. **Description:** The topic index definition file contains parameters for topic index configuration. Topic indexes are optional indexes that allow fast searching of topics.
`style.stp`	**Purpose:** Word-index tuning **Editable?** Yes. **Description:** The stop-word list (word exclusion file). It contains a list of words that are to be excluded from a collection's full-word index. **See:** "Defining Indexing Stop Words (style.stp)" on page 257
`style.tde`	**Purpose:** Document-field extraction **Editable?** Yes. **Description:** This is the field extraction rule file in which you specify the rules to be used when extracting fields with the `-extract` option of `mkvdk`. **See:** "Populating Collection Fields" on page 167

Table 1-1 Standard style files (continued)

File	Explanation
style.tkm	**Purpose:** Token mapping **Editable?** Yes. **Description:** Specifies document zones and fields to be created based on document formatting and location information. Can be used to improve relevance ranking or for other purposes. **See:** "Using Custom Zones to Improve Relevance (style.tkm)" on page 248
`style.ufl`	**Purpose:** User-defined collection fields **Editable?** Yes. **Edit with StyleSet Editor?** Yes. **Description:** The user-defined field list. It defines the collection's document-table schema for custom fields that you define. **See:** "Defining Collection Fields" on page 145 "Contents of style.ufl" on page 157
`style.uni`	**Purpose:** Document-filtering configuration **Editable?** Yes. **Description:** The universal filter file. For every supported document type, it specifies which helper filters the universal filter is to load, in what order, and what characteristics they should have. **See:** "Using the style.uni File" on page 108
`style.ve`	**Purpose:** Entity-extraction configuration **Editable?** Yes. **Description:** The Verity Extractor filter configuration file. You use it to set extraction engines and to specify mappings form entities to fields or zones. **See:** *Verity Extractor Programming Guide.*
`style.vgw`	**Purpose:** Gateway specification **Editable?** No. **Description:** The gateway definition file. It identifies the gateway to be used to access the collection's document repository. This file should normally not be edited if you are using a gateway supplied with Verity K2, unless you are indexing container files; see "Supporting Container Files (ZIP and PST)" on page 118. **See:** *Verity Gateway Developer's Kit Programming Reference* (For custom gateways)

Table 1-1 Standard style files (continued)

File	Explanation
`style.wld`	**Purpose:** Spanning word list specification **Editable?** No. **Description:** The spanning word list file. It configures a word index that includes all partitions of the collection. The spanning word list is an optional index that aids fast searching.
`style.xfl`	**Purpose:** Extra collection fields **Editable?** No. **Description:** The extra fields definition file. It contains includes for the `style.sfl`, `style.ufl` and `style.gfl` files.
`style.xml`	**Purpose:** XML filter configuration **Editable?** yes. **Description:** The XML-filter configuration file. It supports indexing, metadata extraction, and viewing of XML files. "The XML Filter" on page 132.
`style.zon`	**Purpose:** Zone filter configuration **Editable?** Yes. **Description:** The zone-filter configuration file. It is used to define how tagged (SGML and HTML) documents are filtered. Does not apply to XML documents. **See:** "Defining Document Zones" on page 179

About Style Sets

A style set is a directory of style files that contain specific configuration options used to create a given collection. Each collection has one style set, and you must define that style set before you create the collection.

This section only summarizes style-set creation. For details on how to use the K2 Dashboard and the Verity command-line tools to create and modify style sets, see

- *Verity K2 Dashboard Administrator Guide*
- *Verity Command-Line Indexing Reference*
- Gateway Guides:
    - *Verity K2 Lotus Notes Gateway Guide*
    - *Verity K2 ODBC Gateway Guide*
    - *Verity K2 Documentum Gateway Guide*
    - *Verity K2 Exchange Gateway Guide*

Standard and Default Style Sets

Verity supplies standard style sets and style-set templates for your use. Some of them (the standard style sets) you can use as they are, and others (the templates) you must customize before use. In most cases, regardless of whether you are starting with a standard style file or a template, you are likely to need to modify various style-file entries or create new style files to customize the behavior of your collection.

Apart from configuring a specific collection, you can also modify a standard style set or template, in order to globally customize the behavior of all future collections. If you do so, first make a backup copy of the original style set in a safe location, preferably outside of the Verity product installation directory. That way, you can always recover the original style files.

Note None of the standard style sets configures collections for document-level security. However, you can use the style-set templates (see "Style Sets Used With the StyleSet Editor" on page 56) to create style sets in which document-level security is enabled.

Table 1-2 summarizes the locations the standard style sets and templates available on a K2 system, and notes how the style sets are accessed during collection creation.

Table 1-2 Locations of the standard style sets and templates

Location (description)	Accessed through...[a]
dataDir\stylesets\stylesetName\\ (Standard style sets for HTTP and file-system gateways)	K2 Dashboard rcadmin
productDir\_nti40\bin\templates\vgw*GatewayName*\\ (Style-sets templates for Notes, ODBC, Documentum, and Exchange gateways)	StyleSet Editor
productDir\common\styles\stylesetName\\ (Standard style sets for HTTP and file-system gateways —specified with -style option)	mkvdk vspider k2spider
productDir\common\style\\ (Standard style sets for HTTP and file-system gateways —used when no style set is specified)	mkvdk vspider k2spider

[a] The style sets are also accessed programmatically, for example, through the VAdministration Java (see the VAdministration Javadocs) or the Administration C API (see the *Verity K2 Administration Programming Guide*).

Style Sets Used With the K2 Dashboard

When you create a collection through the K2 Dashboard, you can either assign a pre-existing style set to it or you can create a new style set for it. The pre-existing style sets available for you to choose from are those that have been imported into the K2 Dashboard; they are stored in the directory *dataDir*\stylesets (where *dataDir* is the data directory—for example, C:\Program Files\Verity\data on Windows) on the Master Administration Server machine.

These are the standard style sets initially available in this directory:

- Def_Filesystem. Default style set for non-secure access through the file-system gateway.
- Def_Filesystem_Secure. Default style set for secure access through the file-system gateway.
- Def_Filesystem_PushAPI. Default style set for access through the collection-indexing API.
- Def_HTTP. Default style set for non-secure access through the HTTP gateway.
- Def_HTTP_Secure. Default style set for secure access through the HTTP gateway.

If one of the available style sets is exactly perfect for the collection you are creating, you can select it. But if your collection requires a style set that is different (even only slightly different) from the existing ones, you need to create a new style set.

If you choose to create a new style set, you start with a cloned copy of one of the above standard style sets (based on your gateway and security requirements), or you can choose to clone a copy of any other registered style set (for your gateway). You can then customize the copy to fit your needs.

Every new style set that you create through the K2 Dashboard is registered with the Dashboard, is stored in `dataDir\stylesets`, and is available for use or cloning for future collections.

Note The K2 Dashboard allows you to edit any of its registered style sets. Doing so will affect the behavior of all future collections created with that style set, but will have no effect on any existing collections.

Style Sets Used With the StyleSet Editor

If you create a new style set through the K2 Dashboard, the StyleSet Editor (SSE) is invoked to create the style set.

- For a collection that is to access documents through the File System, HTTP, or ODBC gateways, the Java version of the StyleSet Editor is used.
- For a collection that is to access documents through the Notes, Documentum, or Exchange gateways, the Microsoft Management Console (MMC) version of the StyleSet Editor is used. (You can also use the MMC StyleSet Editor outside of the K2 Dashboard.)

  Note The MMC StyleSet Editor runs only on Windows platforms.

When you use SSE to create a style set, you start with a style set cloned from a template specific to your gateway. The style-set templates are stored in the directory `productDir\_nt40\bin\templates` (where *productDir* is the product-specific directory in your Verity installation—such as `C:\Program Files\Verity\k2`) on any machine on which SSE has been installed.

These are the style-set templates available in this directory:

- `vgwdctm`. Style set for the Documentum gateway.
- `vgwmsxch`. Style set the Exchange gateway.
- `vgwnotes`. Style set for the Lotus Notes gateway.

You then use SSE to customize your style set to fit your needs. Note that none of these style-set templates can be used as-is; each needs customization for the particular collection its applies to.

A style set created by the StyleSet Editor is saved as follows:

- If you invoke SSE through the K2 Dashboard, the style set is registered with the Dashboard, stored in `dataDir\stylesets`, and made available for cloning for use by future collections.
- If you call the StyleSet Editor from a command-line tool, you specify where the new style set should be saved. The style set is not registered with the K2 Dashboard.

Note If you manually edit a style set in `productDir\_nt40\bin\templates`, the changes will affect the behavior of all future collections created with that style set, but will have no effect on any existing collections.

Other Standard Style Sets

If you create a collection using any of the Verity command-line indexing tools such as `vspider`, `k2spider_srv`, or `mkvdk`, you use the `-style` option to specify the style set to use for creating the collection's style set.

For the HTTP and file-system gateways, Verity provides sample standard style sets that you can use for this purpose. (For the Notes, ODBC, Documentum, or Exchange gateways, you need to create a customized style set using the SSE, as mentioned in the previous section.)

You can use the standard style sets mentioned under "Style Sets Used With the K2 Dashboard" on page 55 with the `-style` option. You can also use another group of standard style sets, in the directory `productDir\common\styles` on any machine on which K2 or VDK has been installed.

These are the usable standard style sets available in this directory:

- `fspush`
- `fssec`
- `fsusec`
- `httpsec`
- `httpusec`

(Other style sets in this directory have names that begin with `vgw`; they apply to gateways other than HTTP or file system and are not directly usable, although you can edit them manually if desired.)

These style sets are named somewhat differently from, but are identical to, the equivalent standard style sets in `dataDir\stylesets`, as described in "Style Sets Used With the K2 Dashboard" on page 55.

Note If you manually edit a style set in `productDir\common\styles`, the changes will affect the behavior of all future collections created with that style set, but will have no effect on any existing collections.

The Default Style Set

If you create a collection using a command-line tool and you do not use the `-style` option to specify a style set, the tool automatically applies the default Verity style set to your collection. The default style set is in the directory `productDir\common\style` on any machine on which K2 or VDK has been installed.

The default style set can be used only for non-secure repository access through the file-system or HTTP gateways.

Note If you manually edit the default style set, the changes will affect the behavior of all future collections created with that style set, but will have no effect on any existing collections.

A Collection's Internal Style Set

When you create a collection, the style set that you choose (or create) for it is copied into the collection itself, in the directory `collectionName\style`. All actions taken on that collection use the collection's copy of that style set, not the original style set that you specified at creation.

Therefore, if you wish to use style-file modifications to change an existing collection's characteristics or behavior, you must make those modifications on the collection's copy of the style set. Those changes will of course not alter the characteristics or behavior of any other existing or future collection.

Note If you modify a style file for an existing collection, you must then re-index the collection. (The exception to this is `style.plc`, which affects indexing behavior but not the contents of the collection. See "Setting Indexing and Search Policies" on page 83 for information on `style.plc`.)

Collection-Management Tools

Verity provides a variety of tools for creating, indexing, diagnosing, and maintaining collections.

GUI tools:

- **K2 Dashboard**. The principal application for collection management in Verity K2. Using the Dashboard graphical interface, you can create and index collections, attach them to K2 Servers and Brokers, schedule regular updates, generate usage reports, and perform other maintenance tasks.

  The K2 Dashboard is described in the *Verity K2 Dashboard Administrator Guide*.

- **Verity Collaborative Classifier (VCC)**. The principal application for managing classification structures (taxonomies, parametric indexes, parametric trees, and topic sets). The VCC graphical interface helps you to build information-classification capabilities on top of Verity collections.

  VCC is described in the *Verity Collaborative Classifier Guide*.

- **Verity Intelligent Classifier (VIC)**. An application with a graphical interface for managing knowledge trees and topic sets. VIC also supports generation of taxonomies by applying thematic mapping and logistic-regression classification to collections.

  VIC is described in the *Verity Intelligent Classification Guide*.

Command-line tools:

The following are some of the command-line tools most commonly used on collections:

- `mkvdk`. The basic command-line tool for collection maintenance in Verity VDK. You can use `mkvdk` to create a collection, index documents into it, insert or delete documents, control indexing behavior and performance, perform simple maintenance tasks like purging, and delete the collection.

  `mkvdk` is described in "Using mkvdk" on page 279.

Note rmkvdk ("remote mkvdk") is a sample program that provides mkvdk-like functionality in a server environment. It makes use of the K2 Index Server and the collection-indexing API.

- didump. Displays the word list for a collection, one partition at a time. Also displays a list of zones, if zones are used. Described in "didump" on page 327.
- browse. Lists the collection fields and their values, as stored in a collection's document table, one partition at a time. Described in "browse" on page 331.
- rcvdk. A simple command-line search client that allows you to search over a collection and list the collection fields. Described in "rcvdk" on page 336.
- merge. Allows you to split a collection or merge collections that have the same schema (the same set of style files). Described in "merge" on page 335.

For a more complete list of Verity command-line tools, see "Command-Line Tool Summary" on page 267.

2

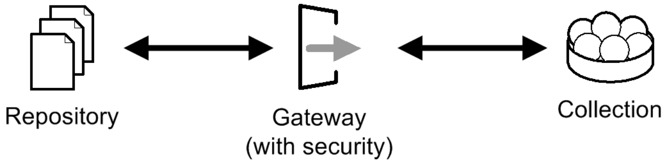
Configuring Gateways

Verity gateways are used to access data anywhere across the network, including data on Web servers. The role of the gateway is to access the data for index, search, and view functions. This chapter describes gateway features in general, and the File System gateway and HTTP gateway in detail.

For detailed information on other Verity gateways, see the *Verity K2 Lotus Notes Gateway Guide*, *Verity K2 ODBC Gateway Guide*, *Verity K2 Documentum Gateway Guide*, and *Verity K2 Exchange Gateway Guide*.

Gateway Configuration Overview

A single gateway can be configured for one repository type, like HTTP or Lotus Notes, one Verity collection, and a specific security method. In the description for each gateway in this chapter, the security features are discussed.



Gateway configuration:

- One repository type
- One security method
- One collection

Primary Document Key Format

The format for the `VdkVgwKey` field uses a URL-style syntax. There is a format for simple keys (non-compound documents) and compound keys (compound documents).

Simple Keys

The format for simple keys follows standard URL conventions, except query strings and anchors are not supported as part of the URL schema. The format for a simple key is:

```
protocol://schema-info
```

Gateway	Protocol Name	Description
File System `vgwfsys`	`file`	Solaris file system, NTFS
HTTP `vgwhttp`	`http`	HTTP 1.0, HTTP 1.1
Exchange `vgwmsxch`	`msx`	Microsoft Exchange

Gateway	Protocol Name	Description
Notes `vgwnotes`	`notes`	Lotus Notes
ODBC `vgwodbc`	`odbc`	ODBC
Documentum `vgwdctm`	`dctm`	Documentum

Gateway Field Types

There are three basic field types used by gateways:

Repository Fields. Database fields that contain the metadata in a repository. Typically, repository fields are the actual database column names, DBMS attributes, etc. Repository fields cannot be read by Verity. Repository fields can be mapped to external fields.

External Fields. Database fields that can be read by Verity to perform some functions. External fields can be displayed in a results list or used in a document view composition via the `style.dft` file.

Internal Fields. Database fields that can be read by Verity without accessing the gateway to perform all field-related functions. Internal fields can be displayed in a results list or used in a document view composition via the `style.dft` file. Unlike external fields, internal fields can be used to perform field searches and results list sorting.

Security Method

All gateways implement the *Results list filtering* security method. Using this method the results list includes only those documents that match the query term and which the user has access to.

Using Separate Gateways for Indexing and Viewing

When you use either the Notes gateway or the Exchange gateway to index, you can implement the HTTP gateway for viewing.

The configuration files for both the indexing gateway and the HTTP gateway for viewing must be in the style directory used. The sample style directories for the Notes and the Exchange gateway each have these configuration files defined. All gateways defined for a particular gateway implementation are defined in DataSource sections of the `style.vgw` file. If a gateway happens to be defined at the top of a `style.vgw` file, it is treated as if it were defined in a distinct DataSource section.

Gateway-Related Style Files

Style directories include both gateway-related style files and standard style files. This means that a single style directory, and therefore a single collection, is tied to a single gateway type, such as file system, HTTP, Lotus Notes, ODBC, or Microsoft Exchange.

The gateway-related style files determine the configuration and behavior of a particular gateway type. The default style files for the File System and HTTP gateways are located here:

productDir`/common/style`

where *productDir* represents the directory containing the installed verity product (such as `usr/verity/k2`). You can enable one gateway through one set of style files.

Verity uses a naming convention to identify style-file types. Standard style files are named `style.*`, where `*` is an extension that describes the purpose of the style file. Gateway-related style files are named `vgw*.*`, where `vgw*` reflects the gateway the style file applies to. For example, the configuration style file for the HTTP gateway is called `vgwhttp.cfg`.

File	Name	Description	Purpose
`vgw*.cfg`	Gateway-type configuration file	Specifies the repositories to index and gateway-specific configuration options to apply.	Repository access

File	Name	Description	Purpose
vgw*.gfl	Gateway field-definition file	Defines fields that are required for proper gateway functioning.	Document-table schema definition
vgw*.prm	Gateway-type parameter file	Specifies the Verity security module to load with the gateway type. If the security module syntax is commented out ($define VGW*_SECURITY 1, where VGW* represents a gateway type like VGWHTTP for the HTTP gateway), security is not implemented. By default, security is implemented.	Gateway performance
vgw*.vgw	Gateway-type definition file	Governs repository access.	Repository access

Gateway-related style files have been set up by Verity and do not generally require configuration. However, to disable security, the line in the vgw*.prm file noted in the above table must be commented out.

Using Different Gateways for Indexing and Viewing

For the Notes and Exchange gateways, there are two sets of gateway-related style files:

- vgw*.* files are configured for repository access for indexing
- vgwhttp.* files are configured for repository access for viewing

For each gateway, both sets of gateway-related style files must be present in order for the gateway to function.

Using the HTTP Gateway

The HTTP gateway enables access to documents on CGI-compliant web servers. Using this gateway, documents managed by web servers can be indexed, searched, and viewed.

Overview

The HTTP gateway has these features:

- Supports indexing local and remote web sites.
- Supports all of the features in Verity Spider, including mime-type mapping and performance enhancements.
- Frame sets are indexed.
- Uses the web server's authentication for access to documents.
- The document returned for `VdkVgwKey` is only the page; scripts are not interpreted; images and frames are not downloaded.

The HTTP gateway accesses all repositories by default in this release. The administrator can limit which repositories, secure directories, and URLs to be accessed by editing the default HTTP gateway configuration file, `vgwhttp.cfg`.

Sample style files appropriate for the HTTP gateway are available in the default style directory, at

`productDir``/common/style/`

where *`productDir`* is the directory for the Verity product you have installed. (For example, for K2 it might be `usr/verity/k2_6/k2`.)

It is recommended that you backup the default style directory to another location before making changes.

Gateway Configuration File Syntax

The HTTP gateway configuration file, called `vgwhttp.cfg`, must be present in the style directory for proper functioning of the gateway. Listing 2-1 shows the content of the default `vgwhttp.cfg` file in `common/style`.

Listing 2-1 HTTP gateway configuration file (vgwhttp.cfg)

```
#
# vgwhttp.cfg - vgwhttp configuration store
#
# This is the default HTTP Gateway config file.
#

$control:1

## the global settings for cookie configuration (optional)
## and will take precendence if no local section is specified
## for settings of cookies for each repository
# cookies:
# {
### you can specify the cookie file (optional), which allows gateway to fetch
### cookie at anthentication time for each user. The file must be in style
### directory.
#   cookieurlfile: "-local meme.txt"
### you can turn on/off cookie support in gateway, the default is on
#   useCookies: true
# }

## the global settings for security cache configuration (optional)
## and will take precendence if no local section is specified
## for settings of security for each repository
# security:
# {
### you can specify the cache level in gateway (optional), the default is document
#   mode: none|webserver|directory|document
#   cachetimeout: seconds # can only be global setting, the default is 3600 seconds
#   usePreauthentication: true # the default is false
# }

## Repository settings.  Zero or multiple repository entries can be listed.
# repository: name
# {
#   securityModuleId: 0x0 or 0xdff4
#   url: http://.*
#
### loginURL (optional). Enable gateway to validate if the user can access
### specified URL, and use the credentials in logon page as necessary
#   loginURL: http://hostname:port/docpath/doc

### the local settings for cookie configuration (optional)
#   cookies:
#   {
##### you can specify the cookie file (optional), which allows gateway to fetch
##### cookie at anthentication time for each user. The file must be in style
##### directory.
#     cookieurlfile: "-local meme.txt"
##### you can turn on/off cookie support in gateway, the default is on
#     useCookies: true
#   }

### the global settings for security cache configuration (optional)
#   security:
#   {
##### you can specify the cache level in gateway (optional), the default is document
#     mode: none|webserver|directory|document
#     usePreauthentication: true # the default is false
#   }
# }

## Proxy settings (optional)
# proxy: hostname portnum
# {
#   proxyAuth: username password (optional)
#   noproxy: (optional)
#   {
#     server: hostname or IP-address (one or more "server" lines ok, up to 255)
#   }
# }

## User-Agent (optional).  Sent as User-Agent in http requests
# userAgent: string

## Timeout (optional).  Count in seconds before timing out a connection.
## Gateway will wait 2 x count for data once connection is established.
# timeout: count

## autoLogin (optional).  This option if set to TRUE indicates the HTTP gateway
## should always send credentials (userid:password) on each GET request if they
## are available for the HTTP target repository.  This is as opposed to the
## default behaviour of only sending credentials if the remote system
## returns a authentication (401) error and they are available.
# autoLogin: TRUE

## User-defined Header (optional).  Sent as a header in http requests
## The string could contain multi headers, but user must have "\r\n"
## for each header. For example:
## header: "attribute: value\r\n"
# header: string

## Ignore the "Charset" defined in the "Content-Type" parameter of the HTTP header.
## The default value is False.
## The HTTP 'Charset' parameter, if defined, will be used as the stream charset
## (instead of detecting the charset from the content-type metadata).
## To ignore the HTTP header charset setting, set the value to True.
# ignoreHeaderCharset: False

$$
```

Note that all statements in Listing 2-1 are commented out, with the exception of `$control:1` and `$$`. All elements in the configuration file, therefore, are optional.

The `vgwhttp.cfg` file may contain any of the following elements.

Table 2-1 vgwhttp.cfg file syntax elements

Element	Description
`cookies:`	Specifies a cookie file to use globally. (See "Cookies Section Syntax" on page 72.) Subelements are: - `cookieurlfile: "-local` *`filename`*`"` - `usecookies: True` or `False`. Default = `True`.
`security:`	Specifies the security level to use globally. (See "Security Levels" on page 74.) Subelements are: - `mode:` Can be `none`, `webserver`, `directory`, or `document`. Default = `document`. - `usePreauthentication: True` or `False`. Default = `False`.

Table 2-1 vgwhttp.cfg file syntax elements (continued)

Element	Description
`repository:` *`name`*	Repository settings. *`name`* is an arbitrary string that identifies a single, logical repository. Subelements are: ▪ `securityModuleID:` *`value`* where *`value`* = `0x0` for nonsecure access, `0xDFF4` for secure access. ▪ `url:` *`url`* where *`url`* is the URL for accessing the repository. *`url`* can have regular-expression syntax. The default `vgwhttp.cfg` file uses an expression for *`url`* that gives access all HTTP repositories, without limit. You can supply multiple `url` entries for each repository. ▪ `loginURL: http://`*`host`*`:`*`port`*`/`*`docpath`*`/`*`document`* An optional URL to a login page, to authenticate the user for repository access. Optional subelements are: - `cookies:` Specifies a cookie file for this repository. (See “Cookies Section Syntax” on page 72.) `cookieurlfile: "-local` *`filename`*`"` `usecookies:` `True` or `False`. Default = `True`. - `security:` Specifies the security level for this repository. (See “Security Levels” on page 74.) `mode:` Can be `none`, `webserver`, `directory`, or `document`. Default = `document`. `usePreauthentication:` `True` or `False`. Default = `False`. Multiple `repository` sections can be used to define multiple repositories. **Note:** Additional subelements of `repository` are required to support forms-based authentication. See “Configuring Forms-Based Authentication” on page 77.

Table 2-1 vgwhttp.cfg file syntax elements (continued)

Element	Description
`proxy:` *`host port`*	Proxy settings. *`host`* and *`port`* specify the proxy server to use. Optional subelements are: - `proxyAuth:` *`userID password`* Credentials for authentication to proxy server. - `noproxy:` Introduces a list of servers (up to 255) that are to be accessed directly, rather than through the proxy server. Each server is identified like this: `server:` *`hostname`* or `server:` *`IPAddress`*
`userAgent:` *`string`*	A string to send as the value for `User-Agent` in HTTP requests.
`timeout:` *`count`*	*`count`* is the time in seconds to wait before timing out a connection attempt. Once a connection is established, the gateway will wait twice this value to receive data.
`autoLogin:`	`True` or `False`. (Default = `False`.) If `True`, the gateway sends credentials (*`userID`*`:`*`password`*), if they are available, with each `GET` request. If `False`, the gateway sends credentials only if the remote system returns a 401 error.
`header:` *`string`*	A string to be sent as an HTTP header with all HTTP requests. Multiple header lines, each terminated with `\r\n`, are permitted. For example: `header: "`*`attrib`*`: `*`value`*`\r\n`*`attrib`*`: `*`value`*`\r\n"`
`ignoreheaderCharset:`	`True` or `False`. (Default = `False`.) If `True`, the character set defined in the `Content-Type` parameter of the HTTP header is ignored, and the document's character set is determined by the Verity auto-detection filter.

Gateway Configuration File Sample

A sample HTTP gateway configuration file is shown here. This configuration file defines three separate repositories. Each repository is defined for a secure directory.

```
$control:1

  repository: A
```

```
    {
      /securityModuleId = 0x0
      /url = http://www.verity.org/dirA/*
    }

   repository: B
    {
      /securityModuleId = 0x0
      /url = http://www.verity.org/dirB/*
    }

   repository: C
    {
      /securityModuleId = 0x0
      /url = http://www.verity.org/dirC/*
    }
  $$
```

Using the above `vgwhttp.cfg` file, Verity prompts the user for credentials for each repository, as defined.

Using Dynamic Cookies

The HTTP gateway supports dynamic cookies. A cookie is a `name = value` pair that is passed from one URL request to another based on the user and the path or site. Cookies are a general mechanism that server side connections (such as CGI scripts) can use to both store and retrieve information on the client side of the connection. Dynamic cookies means multiple cookies can be passed per site or path. The HTTP gateway supports dynamic cookies and gathers all known cookies in the HTTP header. You can specify the use of dynamic cookies globally, or for a particular repository.

By default, the HTTP gateway supports dynamic cookies.

Cookies Section Syntax

To configure the HTTP gateway to support dynamic cookies, add a `cookies` section to your `vgwhttp.cfg` file.

To add cookie support for a particular repository, add a `cookies` section within that repository's `repository` section.

To add cookie support globally, add a `cookies` section to the main body of the `vgwhttp.cfg` file, outside of all repository sections. A repository's `cookies` setting takes precedence over the global `cookies` setting.

The `cookies` section contains two entries, `usecookies` and `cookieurlfile`:

- `usecookies`. Tells the HTTP gateway to support dynamic cookies. Default value is `True`.
- `cookieurlfile`. Optional. Specifies the file that contains URLs to web servers from which cookies will be received to be used at authentication time. The URL file does not itself contain any cookie information. This file must be located in the style directory. Specify `-local` before the file name to force the HTTP gateway to look for the file in the current style file directory. Otherwise, if the full path is not provided, the HTTP gateway looks in the current directory for the file.

Sample vgwhttp.cfg with Cookies Sections

The following sample from a `vgwhttp.cfg` file shows two *`cookies`* sections, one within a repository and one outside.

```
repository: SalesRepository
{
# repository cookie setting

  cookies:
  {
    usecookies: True
    cookieurlfile: "-local url.txt"
  }
}
...
# global cookie setting
cookies:
{
  usecookies: True
}
```

URL Redirection

The HTTP gateway supports URL redirection in the HTTP header section. Redirection is not supported for JavaScript, HTTP META tags, or in the body of the document. If the redirection becomes circular, the HTTP gateway detects the infinite loop and processes the last URL requested.

Security Levels

Search performance is affected by the level of security specified for the HTTP gateway. HTTP gateway has four levels of security configurable on a per-site basis. The four security levels in decreasing order are:

- Document
- Directory
- Web Server
- No Security

Each level of security takes advantage of an in-memory cache with configurable time-out, significantly increasing the performance of results-list filtering. The cache is created per user, is destroyed when the user logs off and is purged when the time-out is reached. The cache minimizes trips to the web server for known URLs. Existing collections created with previous versions of K2 can also be used to increase performance for all security levels except document.

To configure the HTTP gateway for the different levels of security, modify your `vgwhttp.cfg` file.

Document Level

Document is the highest level of security. The HTTP gateway determines access on a per document basis. Each document URL is checked against the web server for authentication. This level of security has the slowest performance.

The HTTP gateway considers each and every URL unique and contacts the web server to ask for access.

Directory Level

The HTTP gateway determines access based on every unique path to a document.

For example, with the following URLs,

```
www.abcd.com/a/b/c/x.html
www.abcd.com/a/b/z.html
www.abcd.com/a/y.html
www.abcd.com/d/e/f.html
www.abcd.com/d/e/g.html
```

the first three URLs have unique paths, so the HTTP gateway contacts the web server for access for each URL. Since the last two URLs have the same path above the file, after the HTTP gateway determines access to the second last URL, it does not contact the web server again for the last URL.

Note If the destination page uses forms-based authentication, the HTTP gateway cannot fill in the user name and password to authenticate or create a cookie.

Web Server Level

The HTTP gateway determines access based on the URL the site resides in. Only URLs from different sites are contacted.

No Security

Results list checking is skipped for all documents. This provides the fastest performance.

Security Section Syntax

To configure the HTTP gateway for the different levels of security, you modify your `vgwhttp.cfg` file. To add or modify security for a particular repository, add or modify a `security` section for that repository's `repository` section. To configure security for all repositories, create or modify a `security` section outside all `repository` sections.

Use the following options in the `security` sections of the `vgwhttp.cfg` file to configure security levels.

- `mode`. Specifies the security level. This can be one of the following values:
  - `none`. Specifies that no security is to be provided
  - `webserver`. Specifies web server security level
  - `directory`. Specifies directory level security
  - `document`. Default. Specifies document level security.

- `cachetimeout`. Specifies (in seconds) when the search will time out. This option can only be applied globally (not in a repository section). Default value is 3600.
- `usePreauthentication`. Pass only the user name, without the password. This option supports the K2 single sign-on feature. This setting can only be applied globally (not in a repository section). Default value is `False`.
- `cachecgiurl`. Specifies that the HTTP gateway cache CGI URLs. By default, CGI URLs are not cached. This setting can only be applied globally, outside of any repository section. The default value is `False`. An example CGI URL would be:

  ```
  http://www.verity.com/search.exe? query=hello &user=test
  ```

Sample vgwhttp.cfg with Security Sections

The following example shows part of a `vgwhttp.cfg` file with one security section inside of a repository, and one outside.

```
repository: name
{
  # respository redirect setting
  security:
  {
    mode: directory
    cachecgiurl: True
  }
}

...
# global security setting
security:
{
  mode: webserver
  cachetimeout: 5
  usePreauthentication: True
  cachecgiurl: True
}
```

Configuring Forms-Based Authentication

The HTTP gateway supports forms-based authentication. For the HTTP gateway to access a form, the following conditions must be met:

- A `[FormsAuth]` section must exist in a K2 Spider *`job`*`.ini` file.

  For more information, see the FormsAuth section in the Job Initialization File Reference chapter of the *Verity Command-Line Indexing Reference*.

- A repository section must exist in the `vgwhttp.cfg` file of the target collection.

  Configuring a repository section in `vgwhttp.cfg` file is explained later in this section.

- The value for `form_loginurl` in `vgwhttp.cfg` must return a status code of 302 and redirect to a URL that contains a form. If the login URL is a form itself, and returns a status code of 200 OK, the HTTP gateway cannot process it.

The K2 Spider job.ini File

In the `[FormsAuth]` section of the *`job`*`.ini` file, you specify the form elements required for authentication of the form. One of these elements may be a password. As described in the *Verity Command-Line Indexing Reference*, you use the K2 Spider Client to create an encrypted password that is stored in the *`job`*`.ini` file and provided at indexing time.

The vgwhttp.cfg File

You must create a `repository` section in the `vgwhttp.cfg` file for each form you want the HTTP gateway to access. Listing 2-2 is an example `repository` section.

Listing 2-2 repository section of vgwhttp.cfg for forms-baed authentication

```
repository: secure_by_form_A
{
  # it must be http security module (0xdff4) for this feature
  securityModuleId: 0xdff4
  # the repository or realm definition
  url: http://www.verity.com/finance/.*
  url: http://www.verity.com/hr/exec/.*

  # info for form based authentication
  # the form login url
  form_loginurl: http://www.verity.com/finance/login.asp
  for_errorurl: http://www.verity.com/finance/error.asp

  # the form method
```

```
form_method: GET
# the form action url
form_action: http://www.verity.com/finance/submit.asp
# the form character set
form_charset = <character_set>

# the credentials required by the form
form_credential: username
form_credential: password
form_credential: mmname
```

Table 2-2 describes the elements that are required in the `repository` section to support forms-based authentication.

Table 2-2 Elements of vgwhttp.cfg file required for forms-based authentication

Element	Description
`repository`	Specifies a unique identifier for this form repository.
`url`	Specifies URLs to documents that are protected by the login page. For example, you could type `http://www.verity.com/finance/*` to include all of the pages beneath `/finance` that are protected by the `login.asp` page from the example.
`form_loginurl`	Specifies the URL of the login page containing the authentication form.
`form_errorurl`	Specifies the URL of the page to which the K2 Spider is redirected when authentication fails. This entry is optional.
`form_action`	Specifies the URL of the page used to process the form. This entry is optional.
`form_charset`	Specifies the character set of the login page. This entry is optional.
`form_method`	Specifies either a GET or POST method is used to submit credentials to the form.
`form_credential`	Specifies a form field name expected by the form. For example, `mmname` for mother's maiden name. Ensure you have defined equivalent entries, with their values, in the *`job`*`.ini` file.

Using the File System Gateway

The File System gateway enables access to documents in the local file system.

Features

The File System gateway has these features:

- Supports indexing files available from the network
- Uses NTFS and Unix permissions for authorization and access rights to documents

Style Directory

The default style directory contains the style files for the File System gateway. The default style directory is located at `productDir/common/style/`, where *`productDir`* is the directory (such as `D:\verity\k2_6\k2`) holding the Verity product that has been installed.

It is recommended that you backup the sample directory to another location before making changes.

It is recommended that you use the File System gateway configuration file, called `vgwfsys.cfg`, as is. The sample File System gateway configuration file has been designed to work on Verity supported platforms and does not require edits or configuration.

Pre-Authentication Support

The File System gateway supports pre-authentication in conjunction with the K2 Ticket Server. To configure a collection to support pre-authentication, edit the `vgwfsys.cfg` configuration file for the desired collection as follows:

```
preAuth = Yes
```

By default, this entry is disabled.

Once you have configured a collection to support pre-authentication, you must use the K2 Dashboard or the `rcadmin` command-line tool to configure your K2 Ticket Server.

For information on corresponding configuration with the K2 Ticket Server, see the *Verity K2 rcadmin Guide*.

Configuration File Syntax

The gateway configuration file, called `vgwfsys.cfg`, must be present for proper functioning of the gateway. The is the default `vgwfsys.cfg` file, from *`productDir`*`/common/style`.

Listing 2-3 File System gateway configuration file (vgwhttp.cfg)

```
#
# Verity File System Gateway Configuration File

$include vgwfsys.prm

$control: 1
fsgw:
{
# Field mapping:  RepositoryFieldName   ExternalFieldName
  map:  "VgwDocKey"             "VgwDocKey"
  map:  "VgwViewURL"            "VgwViewURL"
  map:  "VgwFileCreateDate"     "VgwFileCreateDate"
  map:  "VgwFileModifyDate"     "VgwFileModifyDate"
  map:  "VgwFileAccessDate"     "VgwFileAccessDate"
  map:  "VgwFileSize"           "VgwFileSize"
  map:  "VgwFilePath"           "VgwFilePath"
  map:  "VgwFileOwner"          "VgwFileOwner"

$ifdef  VGWFSYS_SECURITY
  map:  "_VgwAccessType"        "_VgwAccessType"
  map:  "_VgwFileAccessValue"   "_VgwFileAccessValue"
$endif

$ifdefVGWFSYS_FSKEYS
  fskeys: yes
$endif

#repositoryName needs to be specified if it needs to be machine
#or host independent
#repositoryName: MY_REPOSITORY_NAME

#To enable directory level access cache and check.
#For windows, it's off by default because of performance
$ifdef  TP_NTOS
  enableFolderSecurity: NO

## add local server entries to allow gw to fetch local group info,
## for example:
##  localServer: server_name1
##  localServer: server_name2

$else
  enableFolderSecurity: YES
$endif

#To enable the Signle-Sign-On feature in gateway
#preAuth: Yes

# By default, gateway uses process owner to access document without
# authentication. Uncomment it to disable the default bahavior
#defaultUserAccess: 0

# Default data path
  path: "data_default" "."
#  viewURLFormat: http://%s
}
$$
```

Element	Description
`fsgw:`	Identifies the file as a File System gateway configuration file.
`map: "`*`field`*`" "`*`field`*`"`	The mapping from repository field to collection field in the document table. The first value is the repository field name; the second value is the collection field name. Mapped fields are treated as “external” fields—these fields can be displayed in a results list or document view. For information about external fields, see “Gateway Field Types” on page 63. For a secure collection (`VGWFSYS_SECURITY` is defined), two additional field mappings are defined for controlling user access.
`fskeys:`	`Yes` if document keys are to be in native file-system format instead of URI format.
`repositoryName:` *`name`*	By default, the repository name is the machine name of its host. To allow the repository to be moved among hosts, give it an explicit name.
`enableFolderSecurity:`	If `Yes`, folder-level security is enabled. For Windows (`TP_NTOS` is defined), it is off by default; for other platforms, it is on by default.
`localServer:` *`name`*	For Windows, you can name one or more local servers from which the gateway can fetch group information that controls file security.
`preAuth:`	If `Yes`, pre-authentication to support single sign-on is enabled. Commented out by default.
`defaultUserAccess: 0`	If uncommented, the gateway cannot use the process owner to access documents without authentication. Commented out by default.
`path: "`*`base`*`" "."`	A path to an additional data directory holding indexable documents.
`viewURLFormat:` *`URL`*	A partial URL to be combined with a filename or `VdkVgwKey` to form a URL for viewing each document in a Web browser.

3

Setting Indexing and Search Policies

This chapter discusses how you can customize certain behaviors such as indexing mode and what types of documents are returned. The following information is covered:

- About Indexing Modes
- Using Indexing Modes
- Built-in Indexing Modes
- Custom Indexing Modes
- Returning Document Counts
- Skipping Results Set Filtering

About Indexing Modes

An indexing mode is a collection of settings, possibly stored in a policy style file (`style.plc`), that affects Verity engine's indexing behavior. An indexing mode affects the engine's performance during indexing plus the layout of the index data in a collection.

What Indexing Modes Do

Selecting an indexing mode sets a number of metaparameters that are used to build optimized collection components called VDBs. A VDB refers to a proprietary data structure that Verity uses to store different kinds of collection data, including the full-word index, document table, and optional spanning word list. When a VDB is optimized its contents are organized in the best possible layout so that the engine's search performance over the collection is most efficient.

Dynamically Changing Modes

Modes can be changed and updated dynamically for an indexer. This means you can change modes even after some indexing has taken place.

Background vs. Administrative Optimizations

Indexing modes are intended to optimize the way the Verity engine operates during indexing time on a continuous basis. Some optimizations such as squeezing deleted documents can be done only by administrative tools, like `VdkAdminOptimize` and `mkvdk -optimize`. For more information on `VdkAdminOptimize`, see the Verity Developer's Kit documentation. For information on `mkvdk` and its options, see the *Verity K2 Indexers Guide*.

The administrative functions are intended to do all the work necessary to optimize a collection for fast retrieval all in one call. After the function is performed, the VDBs of a collection are optimally packed. You can then make changes to the collection, such as submitting new documents, and make the collection nonoptimal again.

Using Indexing Modes

The Verity engine has many built-in modes to support different indexing behaviors. To use one of the built-in modes, you specify the mode name as an input argument to an indexer. Mode names are case-insensitive.

The method for implementing an indexing mode varies depending on the application:

- Using the Verity VDK API. (No policy style file required.)

- Using the `mkvdk -mode` command-line option. (No policy style file required.)
- Defining a custom mode called `default` in a policy style file.

  The policy style file (`style.plc`) must be stored in a custom style directory that is specified at indexing time, using `mkvdk -style` or the `style` member of the `VdkCollectionOpenArgRec` structure of the VDK API.

  You can create a policy style file with any text editor. It should contain only plain ASCII text. As shipped, Verity K2 does not include a `style.plc` file.

  For more information on defining custom indexing modes, see "Custom Indexing Modes" on page 89.

Using the Verity API

When developing an indexing application using the VDK API, your application can specify the indexing mode when it calls `VdkCollectionOpen`. The `mode` field in the `VdkCollectionOpenArgRec` structure should point to a string that contains the name of the mode to use while indexing.

For example, this sample code shows how to use the `bulkload` mode:

```
{
    VdkCollectionOpenArgRec openRec;
    VdkStructInit(&openRec);
    ...
    openRec.path = "mycollection";
    openRec.mode = "bulkload";
    ...
    VdkCollectionOpen(session, &collection, &openRec);
}
```

You can change an indexing mode after the collection has been opened, using `VdkCollectionSetInfo`.

Using mkvdk

Use the `-mode` option with `mkvdk` to set a policy mode, as follows:

```
mkvdk -collection coll -mode BulkLoad -insert -bulk bulkfile
```

where *`coll`* is a collection name and *`bulk`* file is the bulk load file.

Built-in Indexing Modes

Several built-in modes are predefined by the Verity engine, each of which is designed to support a different indexing behavior. For any one collection, the application can implement one or more built-in or custom indexing modes.

The indexing mode names are described here.

Table 3-1 Predefined Indexing Mode Names

Mode Name	Description
`generic`	The `generic` mode is the base mode from which all other modes inherit their behaviors. It is optimized to give the average overall performance.
`fastsearch`	The `fastsearch` mode can be used to optimize indexes for fastest possible searching.
`bulkload`	The `bulkload` mode can be used to index large numbers of documents using the bulk modify/bulk update feature.
`newsfeedidx`	The `newsfeedidx` mode can be used to index documents arriving from a live feed, quickly and efficiently.
`newsfeedopt`	The `newsfeedopt` mode can be used to optimize collections that were indexed using the `newsfeedidx` mode.
`readonly`	The `readonly` mode can be used to disable modifications to indexes.

You can also define your own indexing mode. For more information, see "Custom Indexing Modes" on page 89.

Generic Mode

The `generic` mode is the base mode from which all other modes inherit their behaviors. It is optimized to give the average overall performance without assuming anything about the desired indexing rates of documents, how many searches are occurring simultaneously, and so on.

The `generic` mode is not very efficient at performing any particular optimization in a short amount of time. It does not perform advanced search optimizations such as creating spanning word lists or squeezing deleted documents.

The Verity engine builds optimized VDBs for the generic mode. The generic indexing mode is equivalent to setting the following metaparameters:

typical_document_size	2000
document_throughput	60
document_latency	200

The `generic` mode (named "generic") is the default mode if the `style.plc` file does not exist at all, or if `style.plc` does not specify a default mode for all applications, or if `VdkCollectionOpenArgRec` does not specify a default mode for custom applications.

Fast Search Mode

The `fastsearch` mode is optimized to index documents so that retrievals happen as quickly as possible. This mode causes the Verity engine to do more work at indexing time.

The Verity engine performs the following optimizations for the `fastsearch` mode:

- It builds optimized VDBs
- It builds spanning word lists, and search accelerators like the ngram index
- It configures collection data to minimize search overhead

The `fastsearch` mode is equivalent to setting the following metaparameters:

typical_document_size	2000
document_throughput	60
document_latency	200

Bulk Load Mode

The `bulkload` mode is for indexing large numbers of documents in large batches with bulk modify/bulk update mechanism. It is primarily intended to create new collections from a large amount of pre-existing documents. The `bulkload` mode inherits most of its settings from the `fastsearch` mode.

The Verity engine performs the following optimizations for the `bulkload` mode:

- It builds optimized VDBs
- It builds spanning word lists, and search accelerators like the ngram index
- It configures collection data to minimize search overhead

The `bulkload` mode is equivalent to setting the following metaparameter:

typical_document_size	2000

News Feed Indexer Mode

The `newsfeedidx` mode is optimized to accept a moderate number of documents in a short amount of time where the documents arrive in frequent small batches. It is designed keep up with the high arrival rates of news feeds without falling behind in the indexing.

Designed to index incoming documents and perform small merges for partitions of up to 100 documents each. These small partitions are not optimized VDBs, since optimization of such small partitions would incur significant overhead.

The `newsfeedidx` mode sets the following metaparameters:

typical_document_size	2000
document_throughput	1000
document_latency	60

If you are developing an indexing application using the VDK API, the following service levels must be set for the session with the `newsfeedidx` mode: `VdkServiceLevel_Index`, `VdkServiceLevel_Optimize`.

News Feed Optimizer Mode

The `newsfeedopt` mode is designed to perform background work that the `newsfeedidx` mode does not. Both modes are designed to be used together.

What the `newsfeedopt` mode does is merge partitions (components of collections) that the `newsfeedidx` mode creates into large and optimized partitions. This mode ensures fast search performance by:

- Performing merges that would result in partitions that exceed 100 documents

- Creating optimized VDBs to allow faster access and larger VDB size at the expense of merging

The Verity engine performs the following optimizations for the `newsfeedopt` mode:

- It builds optimized VDBs
- It builds spanning word lists
- It configures collection data to minimize search overhead

If you are developing an indexing application using the Verity Developer's Kit, the following service levels must be set for the session with the `newsfeedopt` mode: `VdkServiceLevel_Optimize`, `VdkServiceLevel_DBA`, `VdkServiceLevel_Delete`.

Read Only Mode

The `readonly` mode is not an indexing mode in the sense that it does not affect how indexing occurs. It disables data writes to the collection. This mode is useful for accessing a collection on a read-only medium such as CD-ROM.

Custom Indexing Modes

You can define a custom indexing mode by specifying metaparameter modifiers in a policy style file.

Metaparameter modifiers in style.plc

The `style.plc` file has a number of metaparameter modifiers which are used to define indexing modes. These modifiers are a convenient way of setting a number of low-level parameters all at once. Given one metaparameter, the Verity engine calculates appropriate values for a number of low-level parameters. Using metaparameters, system performance can be tuned easily.

The metaparameter modifiers are provided here.

Metaparameter modifier	Description
`/inherit`	Inherits settings from another mode, either one of the built-in modes, or another user-defined mode. Typically, a new named mode inherits a majority of its settings from another mode, then modifies one or two. Default is to inherit the setting from the generic mode.
`/typical_document_size`	A best-guess expression of the typical or average number of indexable text words in the documents you will be submitting. Default is 2000 words.
`/document_throughput`	A best-guess expression of how many documents per hour will be added or updated in the collection. Default is 60 documents per hour.
`/document_latency`	The acceptable latency, in seconds, between the time that the document submitted to the collection and the time that it can be retrieved with a search. The minimum latency is 15 seconds. Default is 200 seconds.

Defining a Custom Mode

Because you cannot modify any of the predefined indexing modes, you must define your own indexing mode in `style.plc`. A custom mode is defined in `style.plc` by specifying a mode name and metaparameter modifiers. The name you specify must be unique.

As a shortcut, you can use the `/inherit` modifier to inherit the metaparameters from one of the predefined modes. Then, you can optionally override individual metaparameters.

Defining a Default Indexing Mode

The Verity engine has a mode called default that it uses if an application does not set the indexing mode using `mkvdk` or the VDK API. The following example defines the default mode as the `bulkload` mode.

```
$control: 1
policy:
{
  mode: default
```

```
    /inherit=bulkload
}
$$
```

Note You must use the mode name of default. However, you can specify any predefined mode as the basis, using the /inherit modifier, and you can also override any of the inherited mode's metaparameters by specifying your own values.

Inheriting From a Predefined Indexing Mode

In this example, you want to define a new mode, called mymode, that inherits all metaparameters of bulkload with an override of typical_document_size=1000.

```
$control: 1
policy:
{
  mode: mymode
    /inherit=bulkload
    /typical_document_size=1000
}
$$
```

Defining Multiple Custom Indexing Modes

You can define multiple modes by repeating the mode: *name* entry with any relevant metaparameter modifiers. For example:

```
$control: 1
policy:
{
  mode: mymode
    /inherit=bulkload
  mode: myothermode
    /inherit=bulkload
    /typical_document_size=1000
}
$$
```

Forcing Serialization of Bulk Transactions

In multithreaded execution, VDK normally does not serialize bulk updates—that is, it does not check to make sure that there are no duplicate keys in the update transactions performed by each thread. If duplicate keys exist, they can be added as redundant entries in the collection's document table.

To avoid the possibility of adding duplicate keys to a collection during bulk updates, you can force the threads to execute serially. `style.plc` provides the `/serialize` modifier for this purpose. Use it like this:

```
$control: 1
policy:
{
  mode: default
    /inherit=Generic
    /serialize=1
}
$$
```

Serializing bulk transactions can cause a significant degradation in indexing performance, because it disallows simultaneous execution of different update threads. If the likelihood of duplicate keys for your bulk update transactions is low, Verity recommends that you do not use this option.

Returning Document Counts

To return the total number of unsecured hits in a search, you must do the following:

- Include the entry `unfiltered_count:yes` in the `style.plc` file before or after indexing.

  For more information, see "Using style.plc" (next).

  For K2 users, you must additionally use the following command-line tool with this option:

  ```
  rck2 -checkmaxdocs
  ```

- For Verity Developer Kit (VDK) users, you must enable the `VdkSearchParam_CountAllHitsOpt` flag in `VdkSearchNew`

  For more information, see the *Verity Developer's Kit API Reference Guide V4.5.*

Using style.plc

You must specify the `unfiltered_count:yes` line in a `style.plc` file, and place it in the style file directory for the relevant collections for which you want to enable this feature.

Note The StyleSet Editor includes a form (**Collection Parameters**) that will create or edit `style.plc` to enable unfiltered counts.

Listing 3-1 is a sample `style.plc` that enables only the unfiltered count feature.

Listing 3-1 Example style.plc file

```
$control: 1
policy:
{
unfiltered_count:yes
}
$$
```

Note If you are also specifying an indexing mode, you should include the line, `unfiltered_count=yes`, at the bottom of your `style.plc` file before the last curly bracket.

For more information on indexing modes, see the earlier sections in this chapter.

Skipping Results Set Filtering

To skip results-set filtering on a secure collection, you can modify a policy style file before or after indexing.

Note The StyleSet Editor includes a form (**Collection Parameters**) that will create or edit `style.plc` to skip results-set filtering.

Include the entry `skip_results_set_filtering:yes` in the `style.plc` file and place it in the style set for the collection for which you do not want to perform filtering of search results.

The following is an example `style.plc` that does nothing but enable the `skip_results_set_filtering` feature.

```
$control: 1
policy:
{
skip_results_set_filtering:yes
}
$$
```

Note If your policy file also specifies an indexing mode, place the `skip_results_set_filtering:yes` line at the bottom of the file, just before the last curly bracket.

4

Filtering and Formatting Documents

This chapter discusses how to configure document filters to affect search engine operations, like indexing and displaying documents stored in a variety of native formats.

Note The zone filter, used for indexing HTML and SGML documents, is covered in detail in Chapter 7. The Verity Extractor filter (`flt_ve`), used to extract entities from documents, is described in the *Verity Extractor Programming Guide*.

This chapter covers these topics:

- The Virtual Document
- Using the style.dft File
- The Universal Filter
- Using the style.uni File
- Universal Filter Document Types
- The KeyView Filters
- The KeyView PDF Filter
- The XML Filter
- Troubleshooting Filters

The Virtual Document

A virtual document represents the document text to be indexed and viewed by the application. There is one virtual document definition for each collection. The definition of a virtual document includes:

- A document layout definition for the document body (that is, the textual content)
- A document filter specification that identifies the filters to be used (that is, the universal filter, plus helper filters for WYSIWYG, Acrobat PDF, HTML)

Document Layout Definition

By default, the Verity engine assumes that the document layout consists of the entire contents of each document's file, beginning at row 1, column 1. You can redefine what the virtual document looks like using the `style.dft` file. Document layout options affecting the placement of the document on the screen for display, with or without field information, can be implemented in the `style.dft` file. The document layout definition indicates the document body text to be indexed and viewed.

Document Filter Specification

Document filters are available to stream documents for indexing and viewing functions. Filters support most major word processing and desktop publishing formats. The default configuration of the universal filter (specified in the `style.uni` file) implements helper filters for all supported document types, including WYSIWYG, PDF, and HTML documents. The default `style.dft` file specifies that the universal filter is to be used.

The universal filter and its configuration are described in "The Universal Filter" on page 104. For information about the document types recognized by the universal filter, see "Supported Document Formats" on page 357.

Default style.dft File

Verity provides a default document format file, called the `style.dft` file that is used to override the virtual document definition. The default `style.dft` file is shown in Listing 4-1.

Note The StyleSet Editor includes a form (**DFT Fields Definition**) for editing `style.dft`.

Listing 4-1 Default style.dft file

```
#
# Document Format
#
$control: 1
dft:
{
  field: DOC
    /filter="universal"
  zone-begin: NOEXTRACT
    /hidden=yes
  field:Title
    /zone=Title
    /hidden=yes
  zone-end: NOEXTRACT
  zone-begin: NOEXTRACT
    /hidden=yes
  field:Keywords
    /zone=Keywords
    /hidden=yes
  zone-end: NOEXTRACT
}
```

In this listing, the main content of the document is defined by the `DOC` field. Two special (`NOEXTRACT`) zones contain information that is not to be used to extract document features for clustering or summarization. Furthermore, fields within those zones specified as `/hidden` are not shown when the virtual document is displayed.

The following section describes in more detail how `style.dft` works.

Using the style.dft File

A `style.dft` file is referred to as a document format file, since this file contains specifications that override the default virtual document definition. The dispatch field consists of the text of the document which begins in row one, column one of the display.

To override the default virtual document definition, you must include a `style.dft` file in the style directory used for creating your collection.

Note If you create a `style.dft` file that contains any fields other than a single dispatch field, and the dispatch field is filtered, your application will be unable to get the raw binary stream from the Verity engine.

style.dft File Syntax

A sample `style.dft` file called `wsjstyle.dft`, is shown here. The sample file illustrates how to add Verity collection fields to the document layout. In this case, the document layout includes these elements:

- The Verity collection field “Source” in row 1, column 5
- The Verity collection field “Title” in row 2, column 5
- The document text starting in row 3, column 1

```
# wsjstyle.dft
#
$control: 1
dft:
  /fill = no
{
    field: Source        # displays the Source field
        /row = 1         # 1st row
        /col = 5         # 5th column
    field: Title         # displays the Title field
        /delta-row = 1   # start on 2nd row
        /col = 5         # start in 5th column
    field: DOC           # start display of doc text
        /delta-row = 1   # start on 3rd row
        /col = 1         # start in 1st column
}
```

```
$$
```

Using the `style.dft` file above, the Verity engine invokes the ASCII filter.

style.dft File Statements

The description for the `style.dft` file syntax for statements is provided here.

Element	Description
$control: 1	The `$control` statement is the first noncomment line in the `style.dft` file. This statement identifies the file as a Verity control file.
dft:	The `dft` statement identifies the control file as a `style.dft` file and it must appear on the second noncomment line in a `style.dft` file. There are three optional modifiers for the `dft` statement. Modifiers assigned to the `dft` statement apply to all values specified in the keyword statements.
/fill = yes | no	This optional modifier to the `dft` statement identifies whether a newline is created if a newline character appears in the field value or constant. By default, newlines are retained (`/fill=no`). If you enter `/fill=yes`, a single newline character in the field value or constant is absent in a document window, and two newline characters in a row are displayed as one.
/right-margin = *`margin_num`*	This optional modifier to the `dft` statement identifies the right margin of the field value or constant to be displayed in a document window. The right margin is expressed as an integer, and the default right margin is 0.
/tabsize = *`tab_chars`*	This optional modifier to the `dft` statement identifies the indent created in a document window when a tab character appears in the field value or constant. The indent created is expressed as a number of characters, and by default a tab character is translated into an 8-character indent.

style.dft File Keywords

Table 4-1 describes the syntax for keywords in `style.dft`.

Table 4-1 style.dft keywords

Element	Description
field: *`fieldname`*	This keyword specifies the name of a field as defined in the document table that you want displayed with each document. These optional modifiers can be used with the field keyword: - The `/filter` modifier specifies which filter to use. - The `/charmap` modifier specifies which character map to use to map the textual output of the filters or gateways into the internal character set. - The `/filter` and `/charmap` modifiers are described in detail in the next section, "style.dft Keyword Modifiers." (See also note about gateway fields following this table.)
constant: "*`string`*"	This keyword specifies a string that you want displayed with each document. The string to be displayed can contain a maximum of 132 characters, and if the string contains white space, the entire string must be enclosed in quotation marks.
system: "*`syscall`*"	This keyword specifies a system call that you want the Verity engine to execute to produce text that you want redirected to the virtual document. To specify a parameter for a field, precede the field name with a dollar sign ($). For example, for a field named title, you could enter `$title` in a system call. The $$ special parameter represents the name of a temporary file to hold the output of the system call; text in the temporary file is redirected to the virtual document definition for each document. For example, this system keyword specifies a script named `myscript` taking the title field as a parameter. `system: "myscript $title > $$"` The output of `myscript` is redirected to the virtual document.

Table 4-1 style.dft keywords

Element	Description
zone-begin: *zone_name*	This keyword specifies a zone name that identifies the beginning of the zone to include in the virtual document. A special zone named noextract specifies fields whose content is not to be used for feature extraction. You can also use this zone to specify hidden elements (text) in the virtual document. Hidden elements get indexed but cannot be viewed. To implement hidden elements, see "Hidden Elements in Zones" on page 214. For complete information about zones, refer to "Defining Zones for Virtual Documents" on page 213. **Note:** If the Verity K2 Viewing Service (described in the *Verity K2 Client Programming Guide*) is to be used to display document content in this collection, you cannot use the zone-begin/ zone-end keywords except in the hidden parts of no-extract zones (see the description of the /hidden modifier in Table 4-2). You can, however, use the /zone modifier as described in "Shorthand Notation for zone-begin and zone-end"(next).
zone-end: *zone_name*	This keyword identifies the end of a zone (defined with zone-begin) to include in the virtual document.

Note Some gateways and filters emit certain fields only when streaming the dispatch field (DOC). When streaming documents from a source other than a collection, any instances of such fields will be empty (unpopulated) unless they are listed in style.dft after the DOC field.

Shorthand Notation for zone-begin and zone-end

A shorthand notation exists for the zone-begin and zone-end combination. You can use instead the /zone construct. For example, as an alternative to the following:

```
zone-begin:   zname
  field: fname
zone-end:     zname
```

you could substitute the following:

```
field:  fname
  /zone = zname
```

style.dft Keyword Modifiers

The `style.dft` file keywords can include one or more modifiers, as described in Table 4-2. The `dft` statement can have a maximum of three modifiers, and there are several more modifiers available for the keywords.

The modifiers available for the `dft` statement are also available for the keywords. The modifiers for the `dft` statement are global variables for the keyword elements. If a modifier for a keyword exists also as a modifier for the `dft` statement, the keyword modifier takes precedence.

Table 4-2 style.dft modifiers

Modifier	Description
/filter="*value*"	This modifier specifies which filter will be used. If not specified, the internal ASCII filter will be used. Valid values are: ■ `universal` for the universal filter (the default; see "Hidden Elements in Zones" on page 214). ■ `flt_pdf` for the PDF filter (see "The KeyView PDF Filter" on page 124). ■ `flt_xml` for the XML filter ■ `zone [-mode]` for the zone filter (see "Defining Document Zones" on page 179). It is recommended that you use the universal filter for all filtering due to its superior performance and handling of character sets.
/charmap	This modifier is used to specify the character set that the document is written in. The search engine will automatically character map the text of the document into the internal character set if necessary before it is indexed or viewed. This modifier is required to properly map any document containing non-ASCII characters. These character map codes can be entered as follows for the Western European languages: ■ `1252` for code page 1252; ■ `850` for IBM code page 850; ■ `8859` for ISO-8859; ■ For Asian localizations, you can enter a character map defined for the locale.

Table 4-2 style.dft modifiers

Modifier	Description
/fill	This optional modifier specifies whether a newline is created if a newline character appears in the field value or constant. By default, newlines are retained (`/fill=no`). If you enter `/fill=yes`, a single newline character in the field value or constant is absent in a document window, and two newline characters in a row are displayed as one. If specified, the fill option given overrides the fill option selected in the same modifier in the `dft` statement.
/right-margin	This optional modifier specifies the right margin of the field value or constant to be displayed in a document window. The right margin is expressed as an integer, and the default right margin is zero. If specified, the right margin given overrides the right margin specified in the same modifier in the `dft` statement.
/tabsize	This optional modifier specifies the indent created in a document window when a tab character appears in the field value or constant. The indent created is expressed as a number of characters, and by default a tab character is translated into an 8-character indent. If specified, the tab given overrides the tab specified in the same modifier in the `dft` statement.
/row	This optional modifier specifies the row number in which the field value or string will be displayed. The first row of a virtual document display is row one.
/col	This optional modifier specifies the row number in which the first character of the field value or string will be displayed. The left-most column of a virtual document display is column one.
/delta-row	The optional modifier specifies a row number relative to the text above it where you want the field value or string displayed. For example, if field is defined to appear in row four, and you specify `/delta-row=2` for a second field, the second field appears two rows ahead, in row six.
/delta-col	This optional modifier specifies a column relative to the right-most character in a row where you want the first character of a field value or string displayed. For example, if a field is defined to appear in row three from columns five to 15, and you specify `/delta-col=5`, the second field will appear five columns ahead, beginning in column 20.

Table 4-2 style.dft modifiers

Modifier	Description
/hidden	This optional modifier specifies that one or more fields are defined as hidden elements. Valid values are: - `YES` to treat the zone's fields as hidden elements. - `NO` (default) to not treat the zone's fields as hidden elements. A special zone named `noextract` is used to specify hidden elements (text) in the virtual document; hidden elements get indexed but cannot be viewed. To implement hidden elements, see "Hidden Elements in Zones" on page 214. For complete information about zones, refer to "Defining Zones for Virtual Documents" on page 213.

Date Formats in the style.dft File

If one of the fields in your `style.dft` file is a date field, you must use the same date output format for both indexing and viewing. If you do not, you may have incorrect highlights (unless your retrieval client uses dynamic highlighting).

Late Binding for Field Elements

The Verity engine uses late binding for field elements in the virtual document, meaning that the value of a field specified in the `style.dft` is not read until the field element is actually inserted into the stream. This enables field values populated by gateways and filters, such as HTML META tags and Microsoft Office properties, to be added to the document stream following the text of the main document.

It is not possible to capture the values of the VdkSummary and VdkFeatures field in the virtual document because these fields are generated after the entire virtual document has already been streamed.

The Universal Filter

This section provides an overview of the universal filter and its implementation.

The universal filter is a document filter that produces indexable (or viewable) text like any other filter. The difference is that it dynamically filters documents according to the type of those documents using a number of “helper” subfilters. For example, Microsoft Word documents are filtered with a certain set of filters (using the KeyView Filter Kit), and HTML documents are filtered in a different way with a different set of filters (the current zone filter).

The advantage of the universal filter is that it removes the need to specify the document type and character set of documents before creating the collection, and it allows multiple document types written in multiple character sets to be indexed into the same collection.

The universal filter is configurable. It has a configuration file that tells it how to filter each type of document that it sees. It also allows multiple filters on each document, so that you are not limited to a single type of filter. The goal of the universal filter design, however, was to be able to filter all important document types “out of the box.” That is, the default configuration file that ships with the search engine should be sufficient for almost all documents that you might want to index. Configuration is offered in case you have special needs that are not addressed in the standard configuration file.

Invoking the Universal Filter

The universal filter is invoked by default, unless you override the default `style.dft` file in the styleset for your collection. When you index your collection or view documents in your collection, the universal filter will filter each document appropriately.

```
mkvdk -create -collection mycoll -insert *
```

How the Universal Filter Works

The following sections describe how the universal filter components work together to filter documents during indexing and viewing operations.

Components

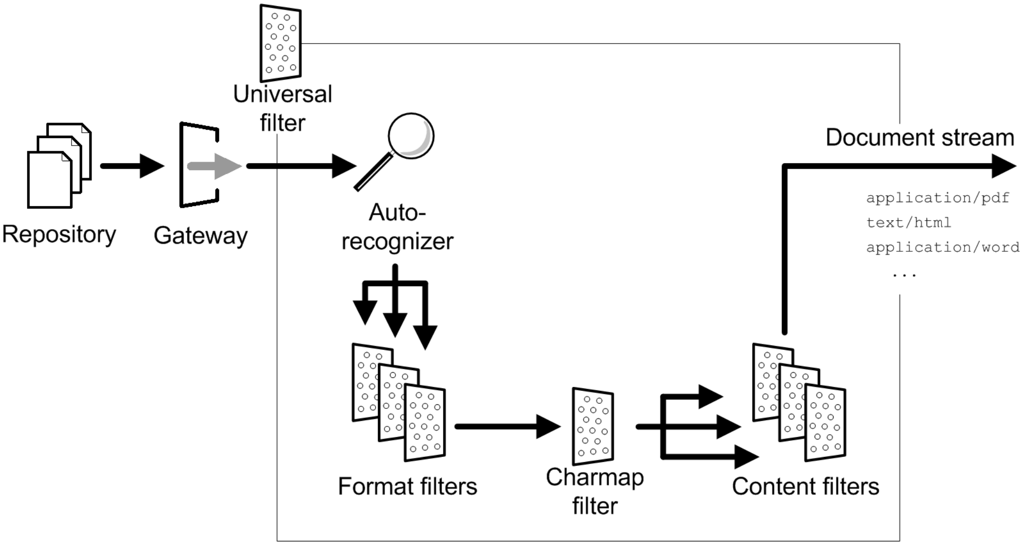
The universal filter has a number of different components:

1. The universal filter itself: This segment installs and synchronizes all the other stream segments.
2. The autorecognizer segments: These segments recognize the type of the document.
3. The format filters: The job of the format filters is to extract indexable text from a binary file.

4. The charmap filter: The job of the charmap filter is to guarantee that all text is written in the internal character set.

5. The content filters: The job of the content filters is to extract meta-information such as fields or zones from the text of the document and send that meta-information up the stream.

Figure 4-1 Universal filter components



How Filtering Occurs

The following steps demonstrate the functioning of each of the parts shown in Figure 4-1.

1. If the first document to be processed was generated by an application in PDF format, the autorecognizer recognizes the document as type `application/pdf` and invokes the PDF format filter. The charmap filter determines its character set and converts it if necessary to the collection’s character set. The appropriate content filter extracts PDF metadata. The document content and metadata are then passed along the document stream to be tokenized and indexed.

2. If the next document to be processed is an HTML document, the PDF filters remain in memory for future use. The autorecognizer recognizes the document as type `text/html`, invokes the appropriate format filter and content filter, and the universal filter sends the content and metadata along the document stream.

3. If the next document to be processed is a Microsoft Word document, the PDF and text filters remain in memory for future use. The autorecognizer recognizes the document as type `application/pdf`, invokes the appropriate format filter and content filter for Word documents, and the universal filter sends the content and metadata along the document stream.

4. Each subsequent document to be processed is handled according to its format. If another PDF document is indexed, for example, the universal filter would reuse the PDF filters it had previously set up.

Character Set Recognition and Mapping

The charmap segment, which is inserted between the format filters and the content filters, guarantees that all text it produces is written in the internal character set. Because different file types are written in different character sets, the charmap segment must sometimes dynamically determine the character set of the text of the document for each document. If the `/charset=guess` modifier is given for any type in the `style.uni` file, the charmap segment will automatically determine the character set of each document and install the correct character set mapping.

The Verity internationalization infrastructure includes the ability to determine the character set of a piece of text with very high precision. For Western European languages, the recognition can be more than 99% correct.

The charmap segment can recognize the character sets listed in Appendix A of the *Verity Locale Configuration Guide*. For Western European languages and the multilanguage locale, it includes these character sets:

- 1252
- 850
- 8859
- Mac1
- UTF-8

Checking File Types

You can determine information about how the Verity engine evaluates the document type and character set for a particular document by looking at the “info” messages that the engine produces. You can see these “info” messages using `mkvdk` with the `-verbose` flag, as shown in the command-line syntax here:

```
mkvdk -verbose -create -collection mycoll -insert mydocs/*
```

The above example illustrates how to index a set of documents into a collection called `mycoll` in verbose mode. For complete information about using `mkvdk`, see the "Using mkvdk" on page 279.

You may want to check the document type recognized by the engine if errors occur. For example, if a web page is interpreted by the engine as a plain ASCII document, then zone searching will not work; if the autorecognizer thinks that the document is written in an incorrect character set, extended characters will not be displayed.

Document types recognized by default are listed in detail in "Supported Document Formats" on page 357.

Using the style.uni File

The universal filter is controlled with a style file called `style.uni`. This style file tells the universal filter which helper filters to load in what order for every possible document type.

Listing 4-2 is an example of a short `style.uni` file that can filter Microsoft Word documents, PDF documents, and email documents.

Listing 4-2 Example style.uni file

```
$control: 1
 types:
 {
   autorec: "flt_kv -recognize"
   postformat: "flt_lang "

   type: "application/msword"
     /format-filter  = flt_kv
     /charset        = guess
     /def-charset    = 1252

   type: "application/pdf"
     /format-filter  = "flt_pdf -charmapto 1252"
     /charset        = guess
     /def-charset    = 1252

   # this is the MIME Content Type for email messages
```

```
      type: "message/rfc822"
        /charset         = guess
        /def-charset     = 1252
        /content-filter = "zone -email -nocharmap"

      # if we get anything else, just skip it.
      default:
        /action = skip
    }
    $$
```

Syntax of style.uni File Statements

The description for the `style.uni` file syntax for statements is provided in Table 4-3.

Table 4-3 style.uni statements

Element	Description
`$control: 1`	Must be the first noncomment line in the `style.dft` file. This statement identifies the file as a Verity control file.
`types:`	Identifies the control file as a `style.uni` file, and it must appear on the second noncomment line in a `style.uni` file.

Syntax of style.uni File Keywords

The description for the `style.uni` file syntax for keywords is provided in Table 4-4.

Table 4-4 style.uni keyword syntax

Element	Description
`autorec: "`*`filter`*`"`	Specifies the name of the filter segment to use as an `autorec` segment. Valid values are: ■ `flt_rec`. The generic autorecognizer; it determines which filter type is appropriate for each document (required) ■ `flt_kv`. The WYSIWYG autorecognizer. Use it with one of these sets of options: `-recognize`. Interpret binary file types for the WYSIWYG filters (required if you have binary WYSIWYG documents). `-trust -recognize`. Enforce the document-type assignment made by the universal filter, regardless of the document's file extension. `-bifmime`. Ignore the document-type assignment made by the universal filter, and instead use a document-type assignment specified in the `MIME-type` field of a BIF file (or in a call to the Collection Indexing API, described in the *Verity K2 Collection-Indexing Programmer Guide*). **Note:** This option affects only the first dispatch field (as specified by `style.dft`) of the virtual document generated for each file. This option is enabled by default only in the `fspush` sample style set. `-unzip`. Index the contents of archive files. When it encounters an archive file, the recognizer extracts its files into a disk cache and creates a queue of file-name tokens to be used for streaming the extracted documents into a single virtual document. (Compare this with separate indexing of archived files, described in "Supporting Container Files (ZIP and PST)" on page 118.) **Note**: This is no longer the recommended method for handling zip files. See"Adding/Removing Container-File Support" on page 119. There may be multiple `autorec` statements in the `style.uni` file. When multiple statements are used, they are installed in the order that they are specified, with the first one being attached to the gateway and the last one being attached on the other end to the universal filter. This argument can be a document data access (DDA) specification for external DDA filters written by a Verity developer.
`default:`	Specifies what the universal filter should do with any document type that is not explicitly listed with a `type` keyword. There can be only one default keyword in the `style.uni` file.

Table 4-4 style.uni keyword syntax (continued)

Element	Description

`postformat:`
`"`*`filter -options`*`"`
	Specifies a format filter (plus appropriate options, if applicable) to apply to all document types after the standard format filter for that type.
	Each `postformat` statement specifies one filter; multiple instances of this keyword are allowed if you want to chain multiple `postformat` filters.
	Specify `flt_lang` for *`filter`* to perform language identification on all documents and write the output into the collection's `VLANG` and `CHARSET` fields. If multiple post-format filters are specified, `flt_lang` must be the last filter in the chain.
	For information on configuring `flt_lang`, see "Configuring the Language-Identification Filter" on page 115.

`postcontent:`
`"`*`filter -options`*`"`
	Specifies a content filter (plus appropriate options, if applicable) to apply to all document types after the standard content filter for that type.
	Each `postcontent` statement specifies one filter; multiple instances of this keyword are allowed if you want to chain multiple `postcontent` filters.

`prerec:`
`"`*`filter -options`*`"`
	Specifies a filter (plus appropriate options, if applicable) to apply to all document types before the auto-recognition filter (`autorec`).
	Each `prerec` statement specifies one filter; multiple instances of this keyword are allowed if you want to chain multiple `prerec` filters.

`type: "`*`type`*`"`	Specifies what the universal filter should do with a particular document type. There may be many type keywords in the `style.uni` file, one for each content type. This argument specifies the name of the content type token as it is emitted from the `autorec` segment. It is usually in the form of class/subtype. For a complete list of the file types defined in the default `style.uni` file, see "Supported Document Formats" on page 357.

Syntax of style.uni Keyword Modifiers

For a `style.uni` file, the `type` and `default` keywords can include modifiers.

Table 4-5 style.uni keyword-modifier syntax

Modifier	Description

`/format-filter="`*`value`*`"`	This modifier specifies that a filter will be used to extract text from a binary file. Valid values are:
	■ `flt_kv` for the KeyView filters;
	■ `flt_pdf` for the PDF filter;
	■ `flt_xml` for the XML filter;
	■ *`DDA spec`* for any DDA-based filter;
	There can be multiple format-filter modifiers, and the binary information will be filtered through each of the specified filters in the order that they are specified in the `style.uni` file. The default is to install no filters.
	■ The `flt_xml` filter can be run without converting META tags to text elements by using the `-nometa` flag:
	`/format-filter=”flt_xml -nometa”`
	■ The `flt_kv` filter can be run in-process for a given MIME type by using the `-noprot` flag:
	`/format-filter=”flt_kv -noprot”`
	■ The `flt_kv` filter can be configured to generate text-formatting zones for a given MIME type by using the `-zoned` flag:
	`/format-filter=”flt_kv -zoned”`
	(See“Generating Text-Formatting Zones” on page 118.)
	■ The `flt_kv` filter can be configured to extract header and footer information from a word-processing file by using the `-headfoot` flag:
	`/format-filter=”flt_kv -headfoot”`
	(See “Extracting Page Headers and Footers” on page 120.)
	■ The `flt_kv` filter can be configured stop filtering a document after a specified number of seconds, by using the `-timeout` flag:
	`/format-filter=”flt_kv -timeout 300”`
	Use this option to ensure that very slow filtering operations time out instead of appearing to hang the system.

Table 4-5 style.uni keyword-modifier syntax

Modifier	Description
`/content-filter="`*`value`*`"`	This modifier specifies that a filter will be used for extracting meta-information from the text. Valid values are: `zone` for the zone filter; `flt_meta` for HTML documents with META tags (for more information, see "Extracting META Tags as Fields" on page 185.); *`DDA spec`* for any DDA-based filter. There may be multiple content-filter modifiers, and the text will be filtered through each of the specified filters in the order that they are specified in the `style.uni` file. The default is to install no filters.
`/charset=`*`name`*	This modifier is used to specify the character set used to represent characters in the document after it has been format-filtered. The text will be automatically character mapped into the internal character set. The valid settings are: - `guess` causes the charmap segment to guess what the character set of the text is (it currently has about 99% accuracy on files larger than 512 bytes written in Western European languages). - `none` causes the charmap segment to pass the text through without any character set mapping. - `1252` for code page 1252. - `850` for IBM code page 850. - `8859` for Latin1 (ISO-8859-1) encoding. - `mac1` for Macintosh Roman1 encoding. Other character sets can be specified, depending on the locale under which the search engine is currently running. The default is to perform no character set mapping.
`/def-charset=`*`name`*	If the `/charset` modifier is given the argument `guess`, the guessing might fail for various reasons. For example, the file might not have been long enough to guess properly. In this case, the `/def-charset` specifies the default character set to use for character set mapping when the guess fails. The valid values for the name are the same as for the `/charset` modifier in the preceding, without the `guess` argument. The default setting for the default character set is `none`.

Table 4-5 style.uni keyword-modifier syntax

Modifier	Description
`/action=`*`action-name`*	This optional modifier specifies the action to perform with documents of this type. Valid values are: - `index` to index a document that should be streamed as normal. - `skip` to skip this type of document, so that it is not indexed or viewed. - `fields-only` to stream this type of document for the purposes of extracting field information only to put in the document table. The text of the document will not be indexed or viewed.
`/protocol=`*`protocol-name`*	This optional modifier specifies a Verity gateway-supported protocol for accessing documents. Valid values are: - `vzip` for accessing the contents of zip files as separate files. - `vpst` for accessing the contents of PST files as separate files. **Note:** The `/action` modifier takes precedence over the `/protocol` modifier, if both are applied to the same type of document. **IMPORTANT:** Any protocol specified in `style.uni` must exactly match in spelling one of the protocols specified in `style.vgw`.

Here is an example of what the initial part of a `style.uni` file that uses `prerec`, `postformat`, and `postcontent` statements might look like:

```
$control: 1
  types:
  {
    prerec: "filter1 -options"
    prerec: "filter2 -options"
    autorec: "flt_kv -recognize"
    postformat: "filter3 -options"
    postformat: "flt_lang "
    postcontent: "filter5 -options"

  type: "application/pdf"
    /format-filter   = "flt_pdf -charmapto 1252"
    /charset         = guess
    /def-charset     = 1252

  type: "message/rfc822"
```

```
    /charset         = guess
    /def-charset     = 1252
    /content-filter = "zone -email -nocharmap"
...
```

Configuring the Language-Identification Filter

You can specify `flt_lang` in a `postformat` statement in `style.uni` to perform language identification on all documents indexed into a collection. The language and character-set information is written to the collection's `VLANG` and `CHARSET` fields.

This is the syntax of the `flt_lang` assignment:

`postformat: "flt_lang [-buffer_size` *`buffer_size`*`] [-config` *`config_path`*`] [-n` *`num_languages`*`] [-lang` *`language`*`] [-nocontent]"`

where

- `-buffer_size` *`buffer_size`* (optional) specifies how much of the document, in bytes, to analyze for language detection. Default = 2048.

  Note In some situations, especially when `flt_lang` needs to distinguish between similar languages, language detection can improve significantly if the buffer size is increased (for example, from 2048 to 4096 bytes).

- `-config` *`config_path`* is the path to the language-list file (`langlist.cfg`). Default = *`verity_product`*`/common/langlist.cfg`.

  The language-list file specifies the languages to test the incoming documents for. See the using locales chapter of the *Verity Locale Configuration Guide* for more information about this file.

  Note If any file names in *`config_path`* contain spaces, enclose the entire path in quotes. Also, on Windows, you may need to use double-backslashes as path separators.

- `-n` *`num_languages`* is the maximum number of languages to assign to a single document. Default = 1.

  If `n` > 1 and more than one language is detected, language codes written into `VLANG` are ordered by score and separated by commas. (`CHARSET` has only one value, the value of the first language encountered.) Language codes are documented in Appendix A of the *Verity Locale Configuration Guide*.

- `-lang` *`language`* is the language (if known) of all incoming documents. If all documents in all repositories for this collections are in the same language, using this option allows immediate assignment of language, bypassing the detection process.

  Note The filter does not verify that the language you supply here is supported. The string you provide is written unchanged into `VLANG`.

- `-nocontent` instruct s `flt_lang` to write only the `VLANG` field and the `CHARSET` field when it processes a document, ignoring document content.

  This option can be used with Profiler to improve performance if only these fields are needed.)

To enable the writing of debugging information to a log file, set the environment variable `VERITY_LANG=DEBUG`. On UNIX, the logging information is written to `/var/tmp/lang`*`xxx`*`.log`. On Windows, it is written to `C:\temp\lang`*`xxxx`*`.log` (or to the location defined by the environment variable `temp` on Windows 2000).

Note these issues with `flt_lang`:

- For archive documents, The values of `VLANG` and `CHARSET` are determined by the first indexable document in the archive.
- For compound documents, `flt_lang` assigns language and character set to each sub-document.
- For markup documents (HTML, XML, SGML), language assignment is based on content, not the tags themselves.
- If multiple post-format filters are specified in `style.uni`, `flt_lang` must be the last one.

The language-identification filter does not assign a language or character set if it has already been assigned by a higher-priority method. These are the priorities for assignment:

VLANG priority	CHARSET priority
1. From BIF file	1. From BIF file

VLANG priority	CHARSET priority
2. From `flt_lang -lang` *`language`*	2. From `/charset` modifier in `style.uni`
3. From gateway or other format filter	3. From gateway or other format filter
4. From `flt_lang` language detection	4. From `flt_lang` language detection

For additional configuration of language detection, including modification of `langlist.cfg` to control the set of languages considered for detection, see the using locales chapter of the *Verity Locale Configuration Guide*.

Conditionally Loading Filters

The VDK API supports the concept of stream mode, based on `VdkDocStreamType`, which differentiates the processing of documents for indexing, profiling, or viewing. Applications can specify the stream mode when initializing a stream to process a document.

Different modes might require the use of different document filters. Therefore, the filter specifications in `style.uni` need to be sensitive to the current stream mode. `style.uni` implements this flexibility by supporting the conditional loading of filters based on the existence of these three defines:

- `VDKSTREAMMODE_INDEX` (for indexing)
- `VDKSTREAMMODE_PROFILE` (for profiling)
- `VDKSTREAMMODE_VIEW` (for any other mode, including viewing)

For example, to use the language-identification filter during profiling only, add lines like this to the initial part of `style.uni`:

```
$control: 1
  types:
  {
    prerec: "filter -options"
    autorec: "flt_kv -recognize"
$ifdef VDKSTREAMMODE_PROFILE
    postformat: "flt_lang "
$endif
    postformat: "filter -options"
...
```

As another example, to use the Adobe PDF filter for indexing or profiling PDF documents, while using the KeyView filter for viewing them, add lines like this to the `type` statement for PDFs:

```
  type: "application/pdf"
$ifdef VDKSTREAMMODE_VIEW
  /format-filter="flt_kv"
$else
  /format-filter="flt_pdf"
$endif
  /charset=utf8
```

For more information on stream mode, see the chapter on writing custom filters in the *Verity Developer's Kit API Reference Guide*.

Generating Text-Formatting Zones

You can use the `flt_kv` filter to generate zones and attributes from the formatting information in a document. For documents whose formatting information can be read by the KeyView filters, `flt_kv` can generate zones for headings, footers, links, font assignments, tables, image tags, and so on. This information can then be used by the token-map segment (see "Using Custom Zones to Improve Relevance (style.tkm)" on page 248) to aid relevance ranking or for any customer-defined purpose.

The set of zones that is generated varies with the individual document format (MIME type) processed.

You enable this capability in `style.uni`, by applying the `-zoned` flag to the `flt_kv` value for the `/format-filter` modifier to a given MIME type:

```
/format-filter="flt_kv -zoned"
```

Supporting Container Files (ZIP and PST)

You can use the `vgwkvcn` built-in gateway to support the indexing and viewing of documents archived within a container file in ZIP or PST format. The contained documents are indexed as separate child documents of the container document.

Adding/Removing Container-File Support

You use the /protocol modifier (see Table 4-5 on page 112) to set up this support. The default version of style.uni uses these lines:

```
#uncomment the next 2 lines to index zip files
  #type: "application/zip"
  #   /protocol = vzip

  #uncomment the next 2 lines to index pst files
  #type: "application/x-ms-pst"
  #   /protocol = vpst
```

Note that to include support for indexing these archive file types, you must uncomment the lines. Furthermore, if you have configured style.uni to support the legacy method for indexing zip files (see the discussion of flt_kv -unzip in Table 4-4 on page 110), you will also need to remove that configuration.

Specifying Cache Characteristics

VDK stores an archive's uncompressed documents in a cache that by default is within the collection directory. You can change the location of that cache, and you can also limit its time of validity, by making changes to the collection's style.vgw file. This is the vgw: statement in an example version of style.vgw:

```
vgw:
  {
    dda: vgwkvcn
    protocols: vzip vpst
    #can be an absolute file path
    config: cache-dir ../kvcache
    #in secs, -1, means no expiration
    #          0, means cache is always recreated
    config: cache-timeout -1
  }
```

Change the config: cache-dir line to specify the desired cache location. Change the config: cache-timeout line to specify the number of seconds after indexing at which the cache should be considered expired and would need to be re-created.

To turn caching off completely, you can set cache-timeout to 0.

Note Turning caching off may cause indexing to take up to 4 times longer, and searching and viewing to take up to 20 times longer, than would be required with caching enabled.

Disabling Filtering

You can disable filtering for a particular mime-type in the `style.uni` file. To disable the use of a KeyView filter for the mime type entry, place a pound sign (#) at the beginning of each line in the entry as shown here:

```
# type: "application/x-lotus-amipro"
#    /format-filter  = flt_kv
#    /charset        = guess
#    /def-charset    = 1252
```

As a result, the Verity engine will not index or display documents for the mime-type shown.

Extracting Page Headers and Footers

The universal filter allows extraction of page headers and footers from word-processing documents during indexing, so that they can be searched and displayed.

When indexed, the header and footer information becomes part of the virtual document. Any full text search will include header and footer text; display of the document will include its headers and footers.

By default, headers and footers are not extracted. To enable this feature, edit the filter type definition in `style.uni` for the type of document whose headers you want displayed. Add the `-headfoot` flag to the `/format-filter` modifier under the `type` statement. For example, to extract headers and footers in Microsoft Word documents, change the Microsoft Word `type` statement like this:

```
type: "application/msword"
  /format-filter = "flt_kv -headfoot"
  /charset = guess
  /def-charset = 8859
```

Note that the `/format-filter` modifier value must be enclosed in quotation marks, as shown.

Consequences of Changing style.uni

If you change the `style.uni` file, you must re-index an existing collection to update it later.

Note If you change the `style.uni` file at all after you have created a collection, you run the risk of changing the way a document is filtered. The main consequence is that your highlights may become considerably inaccurate. If you must change the `style.uni` file, you should use dynamic highlighting from that point onward to guarantee more accurate highlights.

Universal Filter Document Types

This section lists the document types recognized by the universal filter and what the file name extensions are, if any.

Recognized Document Types

When you invoke the universal filter, the following document types are recognized by default. These document types are defined in the `style.uni` file.

Type	Possible File Name Extensions
application/msword application/x-ms-wordpc	doc
application/wordperfect5.1 x-corel-wordperfect	
application/x-ms-excel	xls
application/x-ms-powerpoint	ppt
application/x-ms-works	
application/x-ms-write	
application/postscript	ps, ai, eps, prn
application/rtf	rtf

Type	Possible File Name Extensions
application/x-lotus-amipro	
application/x-lotus-123	123
application/pdf	pdf
application/x-executable	exe, com, dll, 386, ovl, so, sl
application/zip	zip
message/news	
message/rfc822	
text/html	htm, html, shtml, htmls
text/sgml	sgm, sgml
text/ascii	
text/enriched	
text/richtext	rtx
text/container	
text/tab-separated-values	tsv
text/plain	txt, c, h, cpp, y, cc, hh, m, f90, java, csh, ksh, sh, tcl, pl, sed, awk, ini, bat
image/gif	gif

Recognized Categories of Document Types

When the universal filter is invoked, some document types may be recognized even though they do not have MIME types defined yet. Also, some document types may be recognized even though the KeyView Filter Kit does not support them yet. The following universal filter type entries define generic categories of document types.

Type	Category Description
application/x-wordprocessor	Word processor formats that are supported by KeyView Filter Kit, but do not have unique MIME types defined yet
application/x-wordprocessor-nokv	Word processor formats that are unsupported by the KeyView Filter Kit
application/x-spreadsheet	Spreadsheet formats that are supported by the KeyView Filter Kit, but do not have unique MIME types defined yet

Type	Category Description
application/x-spreadsheet-nokv	Spreadsheet formats that are unsupported by the KeyView Filter Kit
application/x-presentation	Presentation graphics formats that are supported by the KeyView Filter Kit, but do not have unique MIME types defined yet
application/x-presentation-nokv	Presentation graphics formats that are unsupported by the KeyView Filter Kit
application/x-graphics	Graphics formats that are supported by the KeyView Filter Kit, but do not have unique MIME types defined yet

The KeyView Filters

The KeyView Filter Kit includes document filters that support indexing and viewing of numerous native document formats. Features of the KeyView filters are:

- Threadsafe filtering of multiple documents simultaneously
- Fast, reliable performance
- Automatic detection of source document format
- Filters for popular formats, including word processing, desktop publishing, spreadsheets and presentations
- Multiplatform support

Numerous popular document suites and formats are supported, including: Microsoft Office 95, 97, and 2000, Corel WordPerfect, Microsoft Word, Microsoft Excel, Lotus AMI Pro, Lotus 1-2-3.

The KeyView PDF Filter

PDF indexing is supported through a dynamically loadable PDF filter (`flt_pdf.so` or `flt_xml.sl` on UNIX, `flt_pdf.dll` on Windows). By default, the Verity engine invokes the universal filter with the PDF filter as a helper filter.

The following is an example of how to construct and add documents to a PDF collection that does not use the universal filter. It uses the PDF filter on its own. Given a Verity installation directory of `/usr/verity`, the PDF collection can be created as follows:

```
% mkvdk -collection pdfcoll -create -style /usr/verity/data/
stylesets/def_filesystem
```

To index the file `REPORT.PDF` into the collection, use the following command:

```
% mkvdk -collection pdfcoll -insert REPORT.PDF
```

The PDF filter offers great versatility, since the PDF documents can reside in any repository as long as it is supported with a valid Verity gateway. For example, the PDF filter can be used in conjunction with the HTTP gateway to index PDF documents on the World Wide Web.

Note The PDF filter tokenizes documents independently of the current locale's tokenizer. Therefore, certain characters in non-English words might not get tokenized. A possible alternative is to use the KeyView filter (`flt_kv`) instead of `flt_pdf` to index PDF documents.

Custom Lexing Rules Not Supported

The PDF filter streams PDF documents and performs the task of lexing. The output of the PDF filter is a series of word tokens and punctuation tokens. These tokens are ignored by the Verity default lexer or any custom lexer that might be defined in the `style.lex` file. There is no way to specify alternative lexing rules.

Specifying the PDF Filter

The PDF filter, named `flt_pdf` can be invoked together with the universal filter or as a single filter. By default, the PDF filter is invoked with the universal filter.

The PDF filter can be invoked in two ways. To invoke the PDF filter with the universal filter, it must be specified in the `style.uni` file with the `type` keyword and the `/format-filter` modifier, as shown in the sample `style.uni` syntax here:

```
type: "application/pdf"
      /format-filter  = "flt_pdf"
      /charset        = 1252 #1252 is the default
```

To invoke the PDF filter as a single filter for a collection using the 850 character set, you must specify the filter in the `style.dft` file using the `field` keyword and the `/filter` modifier, as shown in the sample `style.dft` file syntax here:

```
field: DOC
  /filter   = "flt_pdf -charmapto 850"
  /charmap  = 850
```

If the PDF filter is invoked as a single filter, the engine will index PDF documents only, so the collection will be limited to PDF documents.

Using the -fieldoverride Option

The PDF filter specification located in the `style.uni` file can include a field override option, `-fieldoverride`, that specifies that the field values generated by the PDF filter override those generated by a Verity gateway.

To use the `-fieldoverride` option, include it as part of the `/format-filter` specification as follows:

```
type: "application/pdf"
      /format-filter  = "flt_pdf -fieldoverride"
      /charset        = 1252 #1252 is the default
```

Using the -charmapto Option

The PDF filter specification located in the `style.dft` can include a character mapping option, `-charmapto`, to control the character set output by the filter. This option is specified in the `style.dft` file and is used only when the PDF filter is invoked as a single filter. Valid values for the `-charmapto` option are:

-charmapto Value	Description
1252	For code page 1252
850	For IBM code page 850
8859	For ISO-8859
mac1	For Macintosh systems

The default character set used is platform-dependent. When the `-charmapto` option is not specified, the PDF filter uses the platform's default character encoding. On Unix and Windows systems, the default character encoding is 8859; on Macintosh systems it is mac1.

PDF Fields

While processing each document, the PDF filter generates a series of field tokens containing information extracted and derived from the PDF document. When these fields are defined in the `style.sfl` file, they are populated in the collection's document table. PDF fields can be populated by the PDF filter if they exist in the information dictionary for the PDF document.

Standard PDF Fields

The following PDF fields are predefined as standard fields in the default `style.sfl` file. These fields are populated unless changes are made to the `style.sfl` file. For the predefined fields, the Adobe PDF field names are mapped to Verity collection names as described here.

PDF Field Name	(Verity Collection Field Name) Description

PageMap	PageMap This field represents a vector of integers, one for each page, describing the number of word instances for each page. This field is required. In the default `style.sfl` file, the PageMap field is defined as:
	`varwidth:   PageMap   xya`
	`   /_hexdata=yes`

FTS_Author	(Author) The author of the PDF document obtained by reading the value for the `Author` key in the PDF document's information dictionary. Definition is:
	`varwidth:   Author   ddh`
	`   /alias=FTS_Author`

FTS_Keywords	(Keywords) This field contains the keywords key for the PDF document obtained by reading the value for the `Producer` key in the PDF document's information dictionary. Definition is:
	`varwidth:   Keywords   ddh`
	`   /alias = FTS_Keywords`

FTS_ModificationDate	(Date) The last modification date of the PDF document obtained by reading the value for the `ModDate` key in the PDF document's information dictionary. Definition is:
	`fixwidth:    Date    4    date`
	`   /alias = FTS_ModifidationDate`

FTS_Title	(Title) The title of the PDF document obtained by reading the value for the `Title` key in the PDF document's information dictionary. Definition is:
	`varwidth:    Title    ddh`
	`   /alias= FTS_Title`

Optional PDF Fields

There are several optional PDF fields that can be defined as standard fields. These fields exist in the `style.sfl` file, but are commented out and therefore are not populated by the PDF filter. For information on defining these fields, see "Defining Optional PDF Fields" following this table.

PDF Field Name	Description
DirID	The Adobe path specification for the directory where the PDF file exists. If the PDF document is being pulled from a repository other than the file system, this directory will be the `temp` directory. Definition is: `varwidth:    DirID   ddc`
FileName	The Adobe filename specification for the PDF document. Definition is: `varwidth:    FileName    xya`
FTS_CreationDate	The creation date of the PDF document obtained by reading the value for the `CreationDate` key in the PDF. Definition is: `fixwidth:    FTS_CreationDate    4 date`
FTS_Creator	The creator of the PDF document obtained by reading the value for the `Creator` key in the PDF document's information dictionary. Definition is: `varwidth:   FTS_Creator   xya`
FTS_Producer	The producer of the PDF document obtained by reading the value for the `Producer` key in the PDF document's information dictionary. Definition is: `varwidth:   FTS_Producer   xya`
FTS_Subject	The subject of the PDF document obtained by reading the value for the `Subject` key in the PDF document's information dictionary. Definition is: `varwidth:   FTS_Subject   xyd`
InstanceID	The changing ID found in `/ID` array (position 1) in the trailer of the PDF document. If it does not exist, one is generated using the last modification time. Definition is: `fixwidth:    InstanceID    32   text`
NumPages	The number of pages in the PDF document. Definition is: `fixwidth:   NumPages   4   unsigned-integer`
PermanentID	The changing ID that is found in `/ID` array (position 0) in the trailer of the PDF document. If it does not exist, one is generated using the last modification time. Definition is: `fixwidth:   PermanentID   32   text`
WXEVersion	The version of the Adobe Word Finder used to extract the text from the PDF document. Definition is: `fixwidth:    WXEVersion    1   unsigned-integer`

Defining Optional PDF Fields

In oder to define these option PDF fields, you must do the following:

- Add the `-fieldoverride` option to the PDF filter specification in the `style.uni` file
- Define a new field in the `style.sfl` file

Editing the style.uni File

In a text editor, open the `style.uni` file and add the `-fieldoverride` option to the PDF filter specification as follows:

```
type: "application/pdf"
  /format-filter = "flt_pdf -fieldoverride"
```

Editing the style.sfl File

In order to use one of the optional PDF fields, you must define your own field, using the optional PDF field's definition, that aliases the optional PDF field. In a text editor, open the `style.sfl` file and do the following:

1. Define your new field.

   When you define the new field, add a comment line prior to where you insert the definition so you can easily review what you have added. For example:

   ```
   #My new field to define FTS_CreationDate
   ```

2. Define your field by using the appropriate definition from the table of optional PDF fields provided previously, but replace the field name with your own name. For example:

   ```
   #My new field to define FTS_CreationDate
   fixwidth: PdfCreatedDate 4 date
   ```

3. Add an alias specification that refers to the optional PDF field. For example:

   ```
   #My new field to define FTS_CreationDate
   fixwidth: PdfCreatedDate 4 date
     /alias = FTS_CreationDate
   ```

Note When defining a field for FTS_CreationDate, you also need to add an alias to the field Created as follows:

```
#My new field to define FTS_CreationDate
fixwidth: PdfCreatedDate 4 date
  /alias = FTS_CreationDate
  /alias = Created
```

When defining your fields for the other optional fields, you can just alias the optional field itself.

Paragraph Ordering

The PDF format is primarily designed for presentation and printing of brochures, magazines, forms, reports, and other materials with complex visual designs. Most PDF files do not contain the *logical structure* of the original document—the correct reading order, for example, and the presence and meaning of significant elements such as headers, footers, columns, tables, and so on.

KeyView can filter a PDF file by either using the file's internal unstructured paragraph flow, or by applying a structure to the paragraphs to reproduce the logical reading order of the visual page. Logical reading order enables KeyView to output PDF files containing languages that read from right-to-left (Hebrew, and Arabic) in the correct reading direction.

Note The algorithm used to reproduce the reading order of a PDF page is based on common page layouts. The paragraph flow generated for PDFs with unique or complex page designs may not emulate the original reading order exactly.

For example, page design elements such as drop caps, callouts that cross column boundaries, and significant changes in font size, may disrupt the logical flow of the output text.

Paragraph Direction Options

By default, the KeyView filter produces an *unstructured* text stream for PDF files. This means PDF paragraphs are extracted in the order in which they are stored in the file, not the order in which they appear on the visual page. For example, a three-column article could be output with the headers and the title at the end of the output file, and the second column extracted before the first column. Although this output does not represent a logical reading order, it accurately reflects the internal structure of the PDF.

You can configure KeyView to produce a *structured* text stream that flows in a specified direction. This means PDF paragraphs are extracted in the order (logical reading order) and direction (left-to-right or right-to-left) in which they appear on the page.

The following paragraph direction options are available:

Paragraph Direction Option	Description
Left-to-right	Paragraphs flow logically and read from left to right. This option should be specified when most of your documents are in a language using a left-to-right reading order, such as English or German.
Right-to-left	Paragraphs flow logically and read from right to left. This option should be specified when most of your documents are in a language using a right-to-left reading order, such as Hebrew or Arabic.
Dynamic	Paragraphs flow logically. The PDF filter determines the paragraph direction for each PDF page, and then sets the direction accordingly. When a paragraph direction is not specified, this option is used.

Note Filtering may be slower when logical reading order is enabled. For optimal speed, use an unstructured paragraph flow.

The paragraph direction options control the direction of paragraphs on a page; they do not control the text direction in a paragraph. For example, let us say a PDF file contains English paragraphs in three columns that read from left to right, but 80% of the second paragraph contains Hebrew characters. If the left-to-right logical reading order is enabled, the paragraphs are ordered logically in the output—title paragraph, then paragraph 1, 2, 3, and so on—and flow from the top left of the first column to the bottom right of the third column. However, the *text* direction of the second paragraph is determined independently of the page by the PDF filter, and is output from right to left.

Enabling Logical Reading Order

To enable logical reading order, modify the KeyView `formats.ini` file as follows. (The `formats.ini` file is in the directory `*install*\k2\_nti40\filters\`, where `*install*` is the pathname of the K2 installation directory.):

1. Change the PDF reader entry in the `[Formats]` section of the `formats.ini` file as follows:

```
[Formats]
200=lpdf
```

2. Optionally, specify the paragraph direction. Add the following section to the end of the `formats.ini` file:

```
[pdf_flags]
pdf_direction=paragraph_direction
```

where *`paragraph_direction`* is one of the following:

Flag	Description
`LPDF_LTR`	Left-to-right paragraph direction
`LPDF_RTL`	Right-to-left paragraph direction
`LPDF_AUTO`	The PDF filter determines the paragraph direction for each PDF page, and then sets the direction accordingly. When a paragraph direction is not specified in the `formats.ini`, this option is used.
`LPDF_RAW`	Unstructured paragraph flow. This is the default when logical order is not enabled. If logical reading order is enabled, and you want to return to an unstructured paragraph flow, set this flag.

The XML Filter

The XML filter supports indexing and viewing well-formed XML documents. Metadata extraction is also supported.

Requirements for Indexing XML Documents

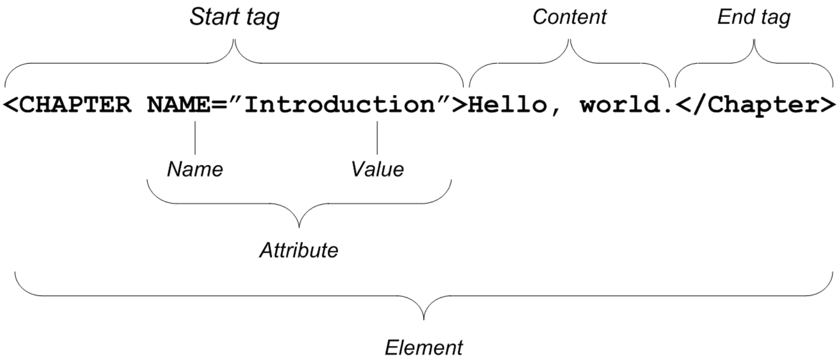
To prepare for indexing XML documents:

1. Make sure that the XML filter (`flt_xml.dll`, `flt_xml.sl`, `flt_xml.so`) resides in the `bin` directory for the installed platform.
2. Make sure that `style.uni` contains the directive for invoking the XML filter.
3. If custom fields or zones are required, define them in `style.ufl`.
4. Specify custom fields to be populated in `style.xml`.

Requirements for Data Files

To be properly indexed, XML data files must be well-formed XML documents as specified in the Extensible Markup Language Recommendation (`http://www.w3.org/TR/REC-xml`).

A well-formed XML document contains elements that begin with a start tag and terminate with an end tag. One element, which is called the root or document element, cannot appear in the content of another element. For all other elements, if the start tag is in the content of another element, the end tag is also in the content of the same element.



The XML data files must have an extension of `.xml` if the universal filter is used; the universal filter is specified in the `style.dft` file. XML documents without the `.xml` extension can be indexed into a collection that contains only XML documents if the `style.dft` file specifies the XML filter instead of the universal filter. For more information, see “style.dft File” on page 141.

Implementation Summary

Verity support for XML documents is implemented by the XML filter and controlled using a number of style files.

The XML filter (`flt_xml.dll`, `flt_xml.sl`, `flt_xml.so`) resides in the platform-specific `bin` directory for the Verity installation.

The following style files are required to enable and configure the indexing of XML files.

Style File	Description
`style.uni`	Invokes the XML filter for indexing XML documents.
`style.xml`	Modifies the default behavior of the XML filter.
`style.ufl`	Defines custom fields in XML documents. The fields must also be defined in the `style.xml` file.
`style.dft`	Specifies whether the universal filter or the XML filter will be used to index the collection. If the XML filter is specified, XML documents can be indexed into their own collection and the `.xml` file extension for data files is not required.

Note As delivered, Verity products are configured to support indexing of XML documents; no style-file customization is required. You can, however, change the XML-filtering characteristics by making appropriate changes to the style files.

Style File Configuration

style.uni File

To index XML documents, `style.uni` must include the following lines:

```
type: "text/xml"
  /format-filter = "flt_xml"
  /charset       = guess
  /def-charset   = 8859
```

Note Older versions of `style.uni` may specify that `text/xml` content is to be handled by `flt-zone`. This specification should be replaced with the above construct.

style.xml File

By default, the XML filter indexes regions of the document delimited by XML tags as *zones*, with the zones given the same name as the XML tag. META tags are automatically indexed as *fields* unless they are in a suppressed region (see the `suppress` command in Table 4-6 on page 139).

The `style.xml` file allows you to change the default indexing behavior for XML documents. You can specify field and zone indexing for regions of the document delimited by XML tags and skip regions of the document delimited by XML tags.

Note The StyleSet Editor includes a form (**XML Styles Definition**) for editing `style.xml`.

The default `style.xml` file in Listing 4-3 contains example commands that are commented out.

Listing 4-3 Default style.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?note: this is a sample comment line?>
<style.xml version="3.0.0">
  <?note:
   ? this following line dictates all xmltags be ignored
   ? <ignore xmltag="*" />
   ?>

  <?note:
   ? "ignore" will skip indexing xmltag, yet index contents
   ? between the beginning and end of this pair of xmltags
   ?>
  <?next 2 sample lines commented out:
  <ignore xmltag="section_1" />
  <ignore xmltag="section_2" />
   ?>

  <?note:
   ? "preserve" indexes xmltag as zone with the presence of
   ? <ignore xmltag="*" />
   ?>
  <?next 1 sample line commented out:
  <preserve xmltag="section_3" />
   ?>
```

```
<?note:
 ? "suppress" will suppress every xmltag embedded within
 ?>
<?next 2 sample lines commented out:
<suppress xmltag="region_1" />
<suppress xmltag="region_3" />
 ?>

<?note:
 ? "zone" will use the value of attribute zonename instead of
 ? the value of xmltag as zone name
 ?>
<?next 1 sample line commented out:
 ? <zone xmltag="section_3" zonename="vdk_zone_name"/>
 ?>

<?note:
 ? "field" will further index content between the beginning
 ? and end of this pair of xmltags as field values
 ?>
<?next 1 sample line commented out:
<field xmltag="column_1" />
 ?>

<?note:
 ? if attribute "xmlattribute" is present, "field" will index
 ? the value of the attribute as its field value
 ?>
<?next 1 sample line commented out:
<field xmltag="column_1" xmlattribute="attribute_1"/>
 ?>

<?note:
 ? if attribute "fieldname" is present, above content will
 ? be indexed into VDK field under the value of fieldname
 ? instead of the field under the name of xmltag/xmlattribute
 ?>
<?next 1 sample line commented out:
<field xmltag="column_2" fieldname="vdk_field_2" />
 ?>

<?note:
 ? if attribute "index" is set to "override", above content
 ? will be indexed into VDK field overriding values read in
```

```
   ? from bulk insert file, if any
   ?>
  <?next 1 sample line commented out:
  <field xmltag="column_3" index="override" />
   ?>

  <?note:
   ? fieldname & index attributes could both exist
   ?>

  <?note:
   ? noindex will skip indexing xmltag and contents
   ? between the beginning and end of this pair of xmltags
   ?>
  <?next 1 sample line commented out:
   ? <noindex xmltag="section_1" />
   ?>

  <?note:
   ? A paragraph token will be generated when xmltag occurs
   ?>
  <?next 2 sample lines commented out:
   ? <paragraph xmltag="section_1" />
   ? <paragraph xmltag="/section_1" />
   ?>

  <?note:
   ? A sentence token will be generated when xmltag occurs
   ?>
  <?next 2 sample lines commented out:
   ? <sentence xmltag="section_1" />
   ? <sentence xmltag="/section_1" />
   ?>

</style.xml>
```

style.xml Commands

Each command in `style.xml` is a single tag that follows this syntax:

```
<command attribute="value"/>
```

To uncomment a command in `style.xml`, you need to remove it from within its enclosing tag and remove its comment mark(`?`). For example, to map the contents of the XML tag `<author>` to a collection field, do this:

- *Before:*

```
<?note:
 ? "field" will further index content between the beginning
 ? and end of this pair of xmltags as field values
 ?>
<?next 1 sample line commented out:
<field xmltag="column_1" />
 ?>
```

- *After:*

```
<?note:
 ? "field" will further index content between the beginning
 ? and end of this pair of xmltags as field values
 ?>
<field xmltag="author" />
```

Table 4-6 describes the available `style.xml` commands and their attributes.

Table 4-6 style.xml commands

Command	Description
`field`	Map the content of the specified XML tag, or the value of the specified attribute, as field values. By default, the field name is the same as the `xmltag` or `xmlattribute` value, unless otherwise specified by the `fieldname` attribute. **Attributes:** ■ `xmltag`. The name of the XML tag whose contents are to be mapped to a collection field. ■ `fieldname`. The name to assign to the collection field. ■ `index="override"`. If present, specifies that this content should override the content of any BIF-created collection field with the same name. ■ `xmlattribute`. The name of the attribute (within this XML tag) whose values are to be put into the field. If `xmlattribute` is present, the specified attribute's value is assigned to the field; otherwise, the specified XML tag's content is used. **Note:** If multiple instances of the specified tag or attribute occur in the document, the field is populated with all of the values (separated by <>).
`ignore`	Do not index (as a zone) the content of the specified XML tag. (The tag's content is still indexed as regular collection content.) **Note:** Subtags within this tag are not ignored; their content is indexed as zones. **Attributes**: ■ `xmltag`. The name of the XML tag to ignore. (To ignore all XML tags, assign the value `"*" to xmltag`.)
`preserve`	Index (as a zone) the content of the specified XML tag, or the value of the specified attribute. Use this command to override an existing `ignore xmltag="*"` command. **Note:** Subtags within this tag are not preserved through this command. You must preserve them explicitly or they will be ignored. **Attributes**: ■ `xmltag`. The name of the XML tag to preserve.
`noindex`	Do not index the content of the specified XML tag—either as a zone or as regular collection content. **Note:** Subtags within this tag are still indexed, as zones. Subtag content, however, is not indexed as regular collection content. **Attributes**: ■ `xmltag`. The name of the XML tag that should not be indexed.

Table 4-6 style.xml commands (continued)

Command	Description
`suppress`	Do not index (as zones) the content of this tag or of any of its subtags—either as zones or as regular collection content. **Attributes**: ▪ `xmltag`. The name of the XML tag to suppress.
`zone`	Index the contents of the specified XML tag into a collection zone with the specified zone name (instead of using the XML tag name as the zone name). **Attributes**: ▪ `xmltag`. The name of an XML tag that is to be indexed. ▪ `zonename`. The name to give the collection zone holding the tag's contents.
`paragraph`	Treat the content of the specified XML tag as a paragraph (generate a paragraph token for it during indexing). **Attributes**: ▪ `xmltag`. The name of the XML tag to generate a paragraph token for.
`sentence`	Treat the content of the specified XML tag as a sentence (generate a sentence token for it during indexing). **Attributes**: ▪ `xmltag`. The name of the XML tag to generate a sentence token for.

style.xml Command Examples:

The following command ignores all XML tags in the document (creates no zones), indexing all content as regular collection content:

```
<ignore xmltag="*" />
```

The following command skips creating a zone for the tag `<section_1>`, indexing the tag's content only as regular collection content:

```
<ignore xmltag="section_1" />
```

The following command creates a zone for the tag `<section_1>` (used only to override for specific tags the command `ignore xmltag="*"`):

```
<preserve xmltag="section_1" />
```

The following command stops indexing (as zones) the tag `<section_1>` and all of its subtags. Furthermore, the content of the tag and all its subtags is not indexed as regular collection content:

```
<suppress xmltag="section_1" />
```

The following command stops indexing (as a zone) the tag `<section_1>`. Any subtags within the tag are indexed. The content of the tag and all its subtags is not indexed as regular collection content:

```
<noindex xmltag="section_1" />
```

The following command maps the content of the tag `<section_1>` to a collection field, which is given the same name as the tag:

```
<field xmltag="column_1" />
```

The following command maps the content of the tag `<section_2>` to a collection field, which is given the name specified in the `fieldname` attribute:

```
<field xmltag="column_2" fieldname="vdk_field_2" />
```

The following command maps the content of the tag `<section_2>` to a collection field (with the same name as the tag), overriding any existing value for the field:

```
<field xmltag="column_2" index="override" />
```

The following command maps the content of the attribute `id` of the tag `<book>` to a collection field named `id`:

```
<field xmltag="book" xmlattribute="id" />
```

The following command maps the content of the attribute `id` of the tag `<book>` to a collection field named `ISBN`:

```
<field xmltag="book" xmlattribute="id" fieldname="ISBN" />
```

style.ufl File

If you have used `style.xml` to specify the population of custom fields, you must also define those fields in the collection's `style.ufl` file or `style.sfl` file, using standard syntax. See "Defining Collection Fields" on page 145.

style.dft File

To create a collection that contains only XML documents, administrators can modify the `style.dft` file to invoke the XML filter directly. In this case, the XML documents do not need a `.xml` extension.

The `style.dft` must include the following lines:

```
$control: 1
dft:
  {
   field: DOC
      /filter="flt_xml"
  }
```

Indexing From the Command Line

To index XML documents using a command-line indexer, issue commands such as these (in `mkvdk`):

```
mkvdk -create -style styledir -collection collname
mkvdk -collection collname file1.xml file2.xml filen.xml
```

Or, using a file list (in this example, `flist.txt`):

```
mkvdk -create -style styledir -collection collname @flist.txt
```

(The file list is a simple list of pathnames, one per line.)

The specified style directory contains the `style.uni` and `style.xml` files for your collection, optionally customized to provide the XML-document indexing that you need.

Troubleshooting Filters

This section contains information about some actions that you can take when errors appear to have occurred with your documents.

Checking File Types

You can determine information about how the Verity engine evaluates the document type and character set for a particular document by looking at the “info” messages that the engine produces. You can see these “info” messages using `mkvdk` with the `-verbose` flag, as shown in this command line (which indexes the documents in the `mydocs` directory into the collection `mycoll`):

```
mkvdk -verbose -create -collection mycoll -insert mydocs/*
```

You may want to check the document type recognized by the engine if errors occur. For example, if a web page is interpreted by the auto-recognizer engine as a plain ASCII document, then zone searching will not work; if the auto-recognizer thinks that the document is written in an incorrect character set, extended characters will not be displayed.

Disable Document Filters by MIME Type

You can disable filtering for a particular MIME Type in the style file `style.uni`. To disable the use of a KeyView filter for the MIME Type entry, place a pound sign (#) at the beginning of each line in the entry as shown here:

```
# type: "application/x-lotus-amipro"
#    /format-filter  = flt_kv
#    /charset        = guess
#    /def-charset    = 1252
```

As a result, the Verity engine will not index or display documents for the MIME Type shown.

5

Defining Collection Fields

The schema definition for a collection's document table is derived from the field definitions made in the `style.ddd` file (internal fields) and the style files `style.ufl` (custom user fields), and `style.sfl` (standard fields). A schema must be defined for each collection, and a default schema is defined in the default `style.ddd` file for your Verity application.

This chapter covers these topics:

- Data Types
- Field Types
- Field Definition Files
- style.ufl Syntax

Data Types

A variety of data types are supported for the Verity field types. Data types define the type of data you want to store in a Verity field. The Verity field types define the scope of the field, meaning whether the field information is the same across the collection or just document-by-document.

The Verity field types are described in the following section, "Field Types."

The following table lists valid data types.

Data Type	Description
`text`	ASCII characters.
`signed-integer`	A range of integers (from negative 2 billion to positive 2 billion) stored in binary format.
`unsigned-integer`	A range of integers (from 0 to 4 billion) stored in binary format.
`date`	An internal date format that stores dates (and times) for the range of years: 1904 to 2037. If field-level searches are to be conducted over a field containing a date, the date field must be defined as a fixed-width or constant field that is assigned a data type called `date` or `xdate`. More than one date field can be defined for a collection.
`xdate`	An internal date format that stores dates (and times) for an extended range of years than the `date` format: 1000 AD to 3000 AD, with a one minute resolution. Due to a calendar adjustment in the 16th century, the day of the week may be inaccurate for dates before the adjustment.
`float`	Floating point number stored in IEEE 754 format.

Data Tables

Field types and data types and variables are stored in one or more data tables. The number of data tables used for a collection depends upon the field types to be included. In some cases, isolating a field definition in its own data table results in improved system performance. More information about the various field and data types and issues surrounding system performance are described in the sections that follow.

Every data table in a collection must contain at least one physical field. That is, at least one field in every data table must be defined with a field type other than `constant`.

Field Types

At the most basic level, there are two categories of Verity fields, distinguished by whether the information in the field types for each category is collection- or document-wide.

- Constant Fields. Constant fields contain the same information for every document in a collection. Within this category are several field types: constant, autoval and worm.
- Variable Fields. Variable fields contain information that can change document-by-document. Within this category are several field types: fixwidth and varwidth.

Constant Fields

Constant fields are populated with values that remain constant throughout the collection.

Value	Description
`constant`	A constant field, defined by a `constant` keyword, assumes a constant value for every document in the collection. Of the `constant` field types listed, this field type functions most efficiently.
`autoval`	An automatic value field, defined by an `autoval` keyword, assumes a constant value for every document in a collection. **Note:** This field type is used internally by the Verity engine and should only be used at the direction of Verity technical support.
`worm`	A `worm` field, defined in a `worm` keyword, assumes a constant value for every document in a collection. The name `worm` stands for Write Once Read Many, which describes how this field behaves. **Note:** This field type is used internally by the Verity engine and should only be used at the direction of Verity technical support.

Variable Fields

Variable fields are populated with variable values parsed from the original document.

Value	Description
`fixwidth`	A fixed-width field, defined in a style file by the `fixwidth` field type, is populated by values that do not change in length.
`varwidth`	A variable-width field, defined by the `varwidth` field type, is populated by text information of variable lengths (such as author or title).
`dispatch`	A variable-width field, defined by the `dispatch` field type, represents free-form text that will be searched and presented for viewing.

Field Definition Files

The field definition files define the collection schema. These are the style files used for field definition:

- Internal Fields (style.ddd)

  Internal fields are defined in the default `style.ddd` file. This file controls the collection's schema and uses a `$include` statement to include and load the `style.ufl`, `style.sfl` and `style.prm` files.

Note You should never manually change the contents of the `style.ddd` file.

- User-Defined Fields (style.ufl)

  Custom user fields (custom application fields) are defined in the `style.ufl` file. This file is included by the `style.xfl` file.

- Standard Fields (style.sfl)

  Standard fields are defined in the default `style.sfl` file. This file is included by the `style.xfl` file.

Internal Fields (style.ddd)

The `style.ddd` file contains definitions for all internal fields included in the default schema for the collection's document table. This file includes a `$include` statement which refers to the `style.xfl` that in turn includes the `style.sfl` and `style.ufl` files.

Contents of style.ddd

Listing 5-1shows the contents of the default `style.ddd` file for the file-system gateway.

IMPORTANT Do not edit the `style.ddd` file.

Listing 5-1 Default style.ddd file

```
#
# Document Dataset Descriptor
#
# DO NOT add user fields to this file - add them to style.ufl
# which is included at the end of this file.
$control: 1
$include style.prm
$subst: 1
descriptor:
  /collection = yes
{
  # Header information for partition management
  data-table:_df
    /num-records = 1
    /max-records = 1
  {
    worm:_DBVERSIONtext
    fixwidth:_DDDSTAMP4 date
    varwidth:_DOCIDX_dv
    fixwidth:   _DOCIDX_OF    4 unsigned-integer
    fixwidth:   _DOCIDX_SZ    3 unsigned-integer
    fixwidth:_PARTDESC40 text # +8 bytes from _SPARE1

    constant:_FtrCfgtext "${DOC-FEATURES:}"
    constant:_SumCfgtext "${DOC-SUMMARIES:}"
    constant:_PBSumCfgtext "${DOC-PBSUMMARIES:}"

    fixwidth:_SPARE18 text # stole 8 bytes from here
```

```
      fixwidth:_SPARE24 signed-integer
    }

    # Required internal fields per document
    data-table:_df
      /offset = 64
    {
      autoval:_STYLEsirepath
      fixwidth:_DOCID4 unsigned-integer
      fixwidth:_SECURITY4 unsigned-integer
        /minmax = yes
      fixwidth:_INDEX_DATE4 date
        /_minmax-nonzero = yes

      # Required for Version 6 collections that support the enhanced
      # Parent child functionality
      fixwidth:VDK_IS_PARENT1 unsigned-integer
      varwidth:_VDK_PARENT_KEYS_pk

      # VLang is filled in by: flt_lang
      varwidth:VLang_pk
        /alias = vdk:VLang
        /alias = gw:LANGUAGE

      # Charset is filled in by: flt_lang or flt_cmap
      varwidth:Charset_pk
        /alias = vdk:Charset
    }

  $ifdef DOC-FEATURES
    # Optional feature vector per document
    data-table:_dg
    {
      varwidth:VDKFEATURES_dh
        /_implied_size
        /alias = dc:Subject
        /alias = vdk:VDKFEATURES
    }
  $endif

  $ifdef DOC-SUMMARIES
    # Optional generated summary per document
    data-table:_di
    {
```

```
    varwidth:VDKSUMMARY_dj
      /_implied_size
      /alias = dc:Description
      /alias = vdk:VDKSUMMARY
  }
$endif

$ifdef DOC-PBSUMMARIES
  # Optional tokenized and compressed texts per document for
  # passage-based summarization
  data-table:_dm
  {
    varwidth: VDKPBSUMMARYDATA _dn
      /_implied_size
      /alias = dc:PBSummaryData
      /alias = vdk:VDKPBSUMMARYDATA
  }
$endif

  data-table:_dk
  {
    dispatch:DOC
    varwidth:DOC_FN_dl
  }

# ------------------------------------------------------------------
# The VdkVgwKey is the application's primary key to identify
# each document in the Document Data Table.  By default, the
# VdkVgwKey is a text string no more than 32000 bytes (VdkDocKey_MaxSize)
# in length.  It is stored in a separate data-table, indexed and
# minmaxxed to minimize the time required to lookup by VdkVgwKey.

data-table: aaa
{
  varwidth: VdkVgwKey aab
    /indexed = yes
    /minmax = yes
    /alias = vdk:VdkVgwKey
}


# ------------------------------------------------------------------
# All extensions the the DDD schema are included via style.xfl
# This includes TIS Standard fields, User defined fields and
# gateway specific fields.

$include style.xfl

}
$$
```

Note Some internal fields are not defined in `style.ddd` and are not accessible. Any collection field whose name is *`fieldName_XX`* is considered by VDK to be an internal field, if there is a field named *`fieldName`* of type `Dispatch` or `Varwidth`, or if *`XX`* is `MI`, `MX`, or `IX`.

Standard Fields (style.sfl)

All collections have the following fields defined in the document table by default because they are defined in the default `style.sfl` file (standard fields file). Standard fields are populated by the filters and gateways used. Not all filters and gateways populate all standard fields so that it is possible that some standard fields will be defined but not populated during indexing.

Field Name	Description
`Title`	Title of the document
`Author`	Author of the document
`Keywords`	Keywords in the document
`MIME-Type`	The mime type of the document. This field is populated by the universal filter.
`Charset`	The character set used in the document
`To`	To field in the document, such as the value of “To:” in an email.
`Date`	Last date the document was modified
`Newsgroups`	News groups (populated only by the news zone filter)
`PageMap`	A vector of integers, one for each page, describing the number of word instances for each page (populated only by the PDF filter)

The default `style.sfl` file includes many field definitions which can be defined and populated as standard fields. If all of these fields were uncommented, that would create a very large document index, so they are left commented for now. If you need any of them, you can uncomment them before creating your collection. For example, if you would like to search over a PDF field in PDF documents, you should uncomment the desired fields in the `style.sfl` file.

For complete information about the universal filter and its configuration, see "The Universal Filter" on page 104.

Field Aliases in style.sfl

The `style.sfl` file uses field aliases to alias field names from various filters to a set of common field names. For example, in the default `style.sfl` file the PDF field name "FTS_Title" is an alias for the collection field named "Title." The field aliasing feature simplifies field display and gives users flexibility to use field names they are familiar with in their queries. This allows you to perform field searches using a field name or an aliased field name. For example, by default you can perform field searches using the field name "FTS_Title" or "Title". To override this default mapping, you can create a custom field in the `style.ufl` file. This file is described in "User-Defined Fields (style.ufl)" on page 156.

Contents of style.sfl

The `style.sfl` file is referenced by the `style.xfl` file, which is in turn referenced by the `style.ddd` file. Listing 5-2 shows the contents of the default `style.sfl` file for the file-system gateway.

Note The StyleSet Editor includes a form (**Defining Collection fields**) for editing `style.sfl`.

IMPORTANT Do not edit the `style.xfl` or `style.ddd` files; instead, change the state of fields here in the `style.sfl` file, or add your custom fields to the `style.ufl` file.

Listing 5-2 Default style.sfl file

```
#
# style.sfl - Verity-Defined Standard Fields
#
# These fields are included in the internal documents table.
# They are filled in by various filters and gateways that Verity ships,
# and are the "standard fields" that Verity suggests should exist in all
# Verity collections. They are not required in your collection. Instead,
# they are merely highly recommended to promote the ability to use your
# collection with other products that use Verity's search technology.
# You can comment out the fields below to save space, or uncomment others
# to gain functionality.
#
data-table:   _sf
{
  # Title is filled in by: zone -html, flt_pdf, flt_kv
   varwidth: Title_sv
     /alias = FTS_Title
     /alias = dc:Title
     /alias = vdk:Title

  # Subject is filled in by: flt_pdf, zone -email, zone -news, flt_kv
   varwidth: Subject_sv
     /alias = FTS_Subject
     /alias = vdk:Subject

  # Author is filled in by: flt_pdf, zone -email, zone -news, flt_kv
  varwidth: Author_sv
     /alias = From
     /alias = FTS_Author
     /alias = Source
     /alias = dc:Creator
     /alias = vdk:Author

  # Keywords is filled in by: flt_pdf, zone -news, flt_kv
  varwidth: Keywords_sv
     /alias = FTS_Keywords
     /alias = Keyword
     /alias = vdk:Keywords
     /alias = Reference

  # Snippet is filled in by: universal
  #varwidth: Snippet_sv
  #  /alias = Abstract
  #  /alias = vdk:Snippet

  # MIME-Type is filled in by: universal
  varwidth: MIME-Type_sv
     /alias = dc:Format
     /alias = vdk:MIME-Type

  # To/CC/BCC is filled in by: zone -email
  varwidth: To_sv
     /alias = Destination
     /alias = vdk:To

  varwidth: CC_sv
     /alias = vdk:CC

  varwidth: BCC_sv
     /alias = vdk:BCC

  # NewsGroups is filled in by: zone -news
```

```
varwidth: NewsGroups_sv

# PageMap is filled in by: flt_pdf
# This field is required to do highlighting in pdf documents. Do not
# comment this if you want pdf highlighting!
varwidth: PageMap_sv
  /_hexdata = yes
  /alias = vdk:PageMap

#
# The following are fields that are available to be populated by Verity
# filters, but are commented out by default to save space as they are
# currently not populated by many documents.
# To enable them to be populated, remove the hash (#) character
# before indexing your documents.
#
# The following fields are filled in by "zone -news"
#varwidth:   References_sv

# The following fields are filled in by flt_pdf
# To enable any of these fields, copy one of the lines to a new line
# and replace the field name with a unique name of your choosing.
# Add an alias line beneath your new field that refers to the PDF field.
# For example, an enabled FTS_Producer field, using the name MyProducer,
# would look like the following:
#    #varwidth:  FTS_Producer    _sv
#    varwidth:  MyProducer    _sv
#       /alias = FTS_Producer
#
# These lines are commented here to keep them from being interpreted.
# Note the relationship, however, of the lines you need to add to the
# original line that begins with the hash (#) character.

#varwidth:   FileName    _sv
#fixwidth:   NumPages    4 unsigned-integer
#fixwidth:   PermanentID 32 text
#fixwidth:   InstanceID 32 text
#varwidth:   DirID       _sv
#fixwidth:   WXEVersion  1 unsigned-integer
#varwidth:   FTS_Creator     _sv
#varwidth:   FTS_Producer        _sv
#fixwidth:   FTS_CreationDate        4   date

# The following fields are filled in by spider
varwidth: URL           _sv
  # Field for storing standard document viewing URL
  # Used for prefix map feature
  /alias = vgw:ViewURL
  /alias = vgw:VgwViewURL
varwidth: _Created      _sv
varwidth: _Modified     _sv
fixwidth: Size          4 unsigned-integer
```

```
  fixwidth: Created  4 date
  varwidth: _TOKENMAP_FN  _sv
    # Field for file pointing to external zones

  # The following fields are filled in by flt_kv
  #varwidth: Dictionary_sv
  #varwidth: CodePage_sv
  #varwidth: Comments_sv
  #varwidth: Template_sv
  #  /alias = dc:Source
  #varwidth: LastAuthor_sv
  #varwidth: RevNumber_sv
  #fixwidth: PageCount          4 unsigned-integer
  #fixwidth: WordCount          4 unsigned-integer
  #fixwidth: CharCount          4 unsigned-integer
  #varwidth: AppName            _sv
  #varwidth: ThumbNail          _sv
  #fixwidth: Doc_Security       4 unsigned-integer

}

data-table:   _sg
{
  # Date is filled in by: zone -email, zone -news, flt_pdf, flt_kv
  #
  # This field is the "last modified" date, not the creation date
  fixwidth: Date        4 date
     /alias = Modified
     /alias = Sent
     /alias = FTS_ModificationDate
     /alias = Recorded_Date
     /alias = Version_Date
     /alias = dc:Date
     /alias = vdk:Date
     /indexed = yes
     /minmax = yes
}
```

User-Defined Fields (style.ufl)

The `style.ufl` file should be used to add custom user fields for the application. The default `style.ufl` file for each gateway does not contain field definitions. The syntax for `style.ufl` is identical to the syntax for the `style.ddd` file.

The `style.ufl` file is referenced by the `style.xfl` file, which is in turn referenced by the `style.ddd` file. Do not edit the `style.xfl` or `style.ddd` files; instead, add your custom fields here to the `style.ufl` file, or change the state of fields in the `style.sfl` file.

Note You should not create a custom field with a name that might conflict with an internal VDK field. Any collection field whose name is *`fieldName_XX`* is considered by VDK to be an internal field, if there is a field named *`fieldName`* of type `Dispatch` or `Varwidth`, or if *`XX`* is `MI`, `MX`, or `IX`.

Contents of style.ufl

Listing 5-3 shows the contents of the default `style.ufl` file for the file-system gateway. Note that no user fields are defined by default.

Note The StyleSet Editor includes a form (**Defining Collection fields**) for editing `style.ufl`.

Listing 5-3 Default style.ufl file

```
#
# style.ufl - Application-specific User Fields
#
# These fields are included in the internal documents table. For
# more information about adding fields to the internal documents
# table, see the discussion of style.ufl syntax in the chapter on defining
# collection fields in the Verity Collection Reference.
# -----------------------------------------------------------------
# Specify additional application-specific fields here in their own
# data-table[s].
```

style.ufl Syntax

The `style.ufl` file syntax is also valid for the `style.ddd` file. You should only edit the `style.ufl` to add custom fields, or the `style.sfl` file to comment or uncomment standard fields. Do not edit the `style.ddd` file unless directed by Verity technical support or sales engineering staff.

Note that delimiters (curly braces) surround the field definitions associated with a `data-table` statement.

Mandatory Statements

$control

The `$control` statement appears on the first non-comment line in a `style.ufl` file. This statement identifies the file as a Verity control file, and it always appears as `$control: 1`.

$control Syntax

```
$control: 1
```

descriptor

The `descriptor` statement identifies the file as a document dataset descriptor file. The descriptor statement must be followed by a `/collection = yes` modifier.

description Syntax

The `descriptor` statement appears as the first statement after the `$control` statement in a `style.ddd/style.sfl/style.ufl` file.

```
descriptor:
  /collection = yes
```

`descriptor:`	This is a required statement that does not take an argument. A `descriptor` statement can be followed by one or more `data-table` statements.
`/collection = yes`	This modifier identifies this file as a collection descriptor file to be used by the Verity engine.

data-table

The `data-table` statement identifies a data table of fields to be included in the collection. A collection can include one or more data tables.

Multiple `data-table` statements can be specified in the `style.ufl` file.

data-table Syntax

The `data-table` statement is a child of the `descriptor` statement.

```
descriptor:
{
data-table: name
```

`data-table:` This statement identifies the name of the data table.

`name` This argument is a unique three-character name assigned to a data segment. Segment names including underscores (_) are reserved for use by Verity. Segment names must be unique within a collection.

Note: Segment names should not contain the same three-characters as style file extensions.

Using data-table Statements

When indexing documents for a collection, the Verity engine creates data segments based on the syntax provided in a `style.ddd` file and `style.ufl` file. If the files have one data table defined, only one data segment is produced for each partition in the collection; if two data tables are defined, two data segments are produced. Each collection's document data table contains one or more field definitions depending again on the syntax of the `style.ddd` file and its included `style.ufl` file used to create the collection.

For a fixed-width or variable-width field, the application developer can elect to store a single field in its own data segment. Storing these types of fields in separate data segments improves their accessibility to the Verity search engine, and can result in improved retrieval performance time. Searching will be faster; display of results slower.

To store a field in its own data segment, use a construct similar to the following:

```
data-table: name
{
varwidth: fieldname identifier
}
```

Constant Field Types

constant

The `constant` field type for a `data-table` statement identifies a field that assumes a constant value for every document in the collection. An application reads a constant field from the collection's document table at run time. The field value assigned to a constant field is always the same for all documents in the collection using the same style.

constant Syntax

The `constant` field type is defined in a `data-table` statement, and the syntax of the `constant` field type is shown here. For a description of *name*, see "data-table Syntax" on page 159.

```
data-table: name
{
constant: fieldname data_type value
}
```

Element	Description
fieldname	This required argument is the field name for the constant field. The name can have a maximum of 124 characters, consisting of alphanumeric characters. Field names including underscores (_) are reserved for Verity.
data_type	This required argument identifies the type of data to be stored in the `constant` field type. The table of valid data types can be found in "Data Types" on page 145.
value	This required argument identifies the value assigned to the constant field. If the value specified contains white space, then it must be enclosed in quotation marks.

autoval

The `autoval` field type for the `data-table` statement identifies a field that automatically assumes a constant value for every document in a collection. This field is referred to as an automatic value field because the Verity engine automatically computes the field value from the `autoval` keyword in the `style.ddd` file at run time.

Note This field type is used internally by the Verity engine and should only be used at the direction of Verity technical support.

The constant value assigned to an automatic value field can fluctuate by collection. For example, if the name of one collection is `march02`, and the name of another collection is `april02`, document fields in the first collection and the second collection are assigned different values.

autoval Syntax

The `autoval` field type is defined in a `data-table` statement, and the syntax of the `autoval` field type is shown here. For a description of *name*, see "data-table Syntax" on page 159.

```
data-table: name
{
autoval: fieldname DBNAME|DBPATH|SIRENAME|SIREPATH
}
```

fieldname This argument is the field name for the automatic value field. The field name can have a maximum of 124 characters, consisting of alphanumeric characters. Field names including underscores (_) are reserved for Verity.

DBNAME The name of the collection that stores information to make documents retrievable. When DBNAME is specified, the name of the collection is assigned as the value of the autoval field.

DBPATH The full or relative pathname of the collection. When DBPATH is specified, the pathname of the collection associated with a document is assigned as the autoval field value.

SIRENAME The name of the `style.ddd` file used to create the collection. When SIRENAME is specified, the name of the `style.ddd` file associated with a document is assigned as the autoval field value.

SIREPATH The full or relative pathname of the `style.ddd` file. When SIREPATH is specified, the pathname of the `style.ddd` file associated with a document is assigned as the autoval field value.

worm

The worm keyword for a data-table statement identifies a field that assumes a constant value for every document in a collection. The field is referred to as a *worm* field because functionally the Verity engine *w*rites the value *o*nce to the collection's documents and *r*eads the value *m*any times (Write Once Read Many).

Note This field type is used internally by the Verity engine and should only be used at the direction of Verity technical support.

worm Syntax

The worm field type is defined in a data-table statement, and the syntax of the worm field type is shown here. For a description of *name*, see "data-table Syntax" on page 159.

```
data-table: name
{
worm: fieldname data_type
}
```

fieldname This argument is the name of the worm field. The name can have a maximum of 124 characters, consisting of alphanumeric characters. The name cannot begin with an underscore.

data_type This required argument identifies the type of data to be stored in the worm field type.

The table of valid data types can be found in "Data Types" on page 145.

Variable Field Types

fixwidth

The fixwidth field type for a data-table statement identifies a fixed-width field. The application developer may elect to store fixed-width field definitions in separate data tables to improve retrieval performance. See "Using data-table Statements" on page 159 for information about storing variable-width field definitions in separate data segments.

fixwidth Syntax

The fixwidth field type is defined in a data-table statement, and the syntax of the fixwidth field type is shown here. For a description of *name*, see “data-table Syntax” on page 159.

```
data-table: name
{
fixwidth: fieldname length data_type
  /indexed = yes|no
  /case-sensitive = yes|no
}
```

fieldname This argument is the name of the fixed-width field. The name can have a maximum of 124 characters, consisting of alphanumeric characters. Note that the name cannot begin with an underscore.

length This required argument identifies the length of the fixed-width field expressed as a number of characters.

data_type This required argument identifies the type of data to be stored in the fixwidth field.

The table of valid data types can be found in “Data Types” on page 145.

/indexed This optional modifier identifies whether the Verity engine creates an index for this field. By default, an index is not created. If you enter /indexed=yes, an index is created.

/case-sensitive This optional modifier identifies whether the Verity engine creates a case-sensitive index if an index is created for this field. By default, an index is not case-sensitive. If you enter /case-sensitive=yes, a case-sensitive index is created.

fixwidth Length and Ranges for Integer Data Types

The following table lists the range for each integer type (unsigned and signed) based on field *length*.

Length (bytes)	Unsigned-Integer Range	Signed-Integer Range
1	0 to 255	-128 to 127
2	0 to 65535	-32768 to 32767
4	0 to $(2^{32}-1)$	-2^{31} to $(2^{31}-1)$

varwidth

The `varwidth` field type for a `data-table` statement identifies a variable-width field. The application developer may elect to store variable-width field definitions in separate data tables to improve retrieval performance. See "Using data-table Statements" on page 159 for information about storing variable-width field definitions in separate data segments.

varwidth Syntax

The `varwidth` field type is defined in a `data-table` statement, and the syntax of the `varwidth` field type is shown here. For a description of *name*, see "data-table Syntax" on page 159.

```
data-table: name
{
varwidth: fieldname identifier
  /indexed = yes|no
  /case-sensitive = yes|no
}
```

fieldname — This argument is the name of the variable-width field. The name can have a maximum of 124 characters, consisting of alphanumeric characters. Note that the name cannot begin with an underscore.

identifier — This required argument identifies a three-character name for the data segment that will store all variable-width field definitions. The segment name should begin with the letters `dd`. The specified name cannot begin with an underscore (_) or the letter combination `di`.

`/indexed` This optional modifier identifies whether the Verity engine creates an index for this field. By default, an index is not created. If you enter `/indexed=yes`, an index is created.

`/case-sensitive` This optional modifier identifies whether the Verity engine creates a case-sensitive index for the field if the field will be indexed. By default, an index is not case-sensitive. If you enter `/case-sensitive=yes`, a case-sensitive index is created.

dispatch

A document dispatch field, defined by a `dispatch` field type, represents free-form text that will be searched and presented for viewing. It points to the document text in place, without copying the text into the collection. By default, the document dispatch field is named `DOC`.

The `dispatch` field identifies a special field referred to as the document dispatch field. In most cases, a collection has only one document dispatch field that stores information about a document's location and size. This information is used to dispatch the document when the application issues a command to display it. The dispatch field is defined in the default `style.ddd` file. It is recommended that this field definition remain in the `style.ddd` file, not in one of the include files.

dispatch Syntax

The `dispatch` field is defined in a `data-table` statement in the `style.ddd` file, and the syntax of the `dispatch` field is shown here. For a description of *`name`*, see "data-table Syntax" on page 159.

```
data-table: name
{
dispatch: fieldname
}
```

`dispatch:` This identifies the field type as a document dispatch field.

`fieldname` This argument is the name of the dispatch field. The default `style.ddd` file assigns the name "DOC" to the document dispatch field.

Assigning a Custom Name to a Dispatch Field

If you want to assign a name other than DOC, then you must create a separate data table for a variable-width field having the same root name as the document dispatch field.

Note Place the data-table segment in the style.ufl file, which is accessed from a $include statement in the style.xfl file, which is in turn accessed from a $include statement in the style.ddd file. For example, if you want to name the document dispatch field article, then the syntax of the style.ufl file would need to include the following syntax:

```
data-table: ddf
{
varwidth: article_fn dd1
dispatch: article
}
```

The variable-width field for the document's file name must have the same root name as the dispatch field, and the variable-width field must have the _fn ending.

If a custom dispatch field name is used (only if your are creating a second dispatch field), you must define a style.dft file that minimally includes the field keyword with the custom dispatch field name. For example, for a dispatch field named article, the following statement must appear in the style.dft file:

```
field:  article
```

For complete information about the style.dft file, see "Filtering and Formatting Documents" on page 95.

6

Populating Collection Fields

This chapter describes using field extraction rules with `mkvdk`. Two field-parsing methods are supported by Verity collection-building products. These topics are covered:

- Methods for Populating Fields
- style.tde Syntax
- style.tde Example

Note For best performance, you should populate fields using the bulk modify feature, described in "Using Bulk Insert Files" on page 313.

Methods for Populating Fields

You may wish to set field values related to your documents, so that the documents are searchable by fields. `mkvdk` supports the following methods for populating fields:

- Using the bulk modify feature to insert documents and set field values in the bulk modify file(s)
- Extracting field values from documents

Using the Bulk Modify Feature

To use the bulk modify feature to populate fields, complete the following steps:

1. Define the fields in the `style.ufl` file.

   For more information about the `style.ufl` file, see “Defining Collection Fields” on page 145.

2. Create a bulk modify file specifying the documents to insert and the field values for each document.

3. Run `mkvdk` using the `-bulk` option and specify the bulk modify file(s).

   For more information about bulk modify files, refer to the chapters on `mkvdk` in the *Verity K2 Indexers Guide*.

Extracting Field Values

To populate fields by extracting field values from documents, complete the following steps:

1. Define the fields in the `style.ufl` file.

   For more information about the `style.ufl` file, refer to “Defining Collection Fields” on page 145.

2. Create a `style.tde` file containing the field extraction rules (the datamap).

   For more information about the `style.tde` file, refer to the next section.

3. Run `mkvdk` using the `-extract` option.

For more information about bulk modify files, refer to the chapters on `mkvdk` in the *Verity K2 Indexers Guide*.

style.tde Syntax

This section explains the `style.tde` syntax used to supply field extraction rules.

In this section, an example of the `style.tde` syntax template is followed by descriptions of the `style.tde` statements in alphabetical order. A statement is any word that may appear at the beginning of a line and is always immediately followed by a colon (:).

Syntax Template

Listing 6-1 is a syntax template for `style.tde`. It includes the syntax relevant to `mkvdk` for extracting field values.

Listing 6-1 Syntax of style.tde

```
$control: 1
tde:
{
  pre-process:
  {
    relative-path: yes|no
  }

  {
    datamap:
      /docsep = "pattern"
      /filter = "filter_name"
      /system = "system_call"
      /charmap = charmap_code
    {
      define: pattern_name "pattern"
      ...
      field: field_name FILENAME|TIME|FILETIME|
      FILESIZE|FILEPATH|PATTERN|LINE num|"pattern"
        /required = yes|no
        /which = [1|###|LAST|ALL]
        /string-before = "string"
        /string-between = "string"
        /string-after = "string"
        /default = "field_value"
        /alsowrite = [field_name|"field_names"]
      ...
      dispatch: field_name
        /required = yes|no
        /start-line = "string"
        /start-pattern = "string"
        /end-line = "string"
        /end-pattern = "string"
        /inclusive = yes|no
      ...
    }
  }
  }
  $$
```

$control

The `$control` statement identifies the file as a Verity control file. It must be the first non-comment line in the `style.tde` file.

Syntax

```
$control: 1
```

datamap

The `datamap` section of the `style.tde` file defines the document body text and the rules for populating field values from documents. The document body text is used to create a collection's full-word index. Field population rules populate fields in the collection's document table, as defined by the `style.ddd` and `style.ufl` files.

The following statements may appear as children of `datamap`: `define, dispatch, field`. `define, dispatch,` and `field` are described later in this section.

Syntax

```
datamap:
/docsep = "pattern"
/filter = "filter_name"
/system = "system_call"
/charmap = charmap_code
{
datamapping
}
```

`/docsep`

This optional modifier identifies the characters that separate documents within a file. If a `/docsep` modifier is not specified, the default separator is the end-of-file. If you have multiple documents in a file, you must define a dispatch field to avoid indexing the entire file for each document within the file. This modifier can be used only if your documents are in plain text (ASCII) format.

`/filter="universal”`

This optional modifier specifies that the universal filter will be used to read documents during indexing and to present them for viewing, and is used only when you want the engine to convert the format of documents before parsing them. Note that if this modifier is used, a `style.dft` file with the same `/filter` modifier is also required.

`/system`

This optional modifier allows you to specify your own filter. Two substitution variables are available: `$filename` and `$$`. The variable `$filename` represents the input text file name, and `$$` represents the output temporary file name to be parsed by the Verity engine.

`/charmap`

This modifier specifies the character map used to print characters to the screen. Note that this modifier is required if a language other than English is used. The following character map codes are available:

- 1252 for code page 1252
- 850 for IBM code page 850
- 8859 for ISO-8859

`datamapping`

This represents a set of statements that define the document body text and the rules for extracting field values from documents. The document body text is used to create the full-word index. Extracted field values may be stored in a collection. To store extracted field values in a collection, the field(s) must be defined in the `style.ufl` file.

define

The `define` statement is a child of the `datamap` statement. It assigns a name to a regular expression. Use this name to represent the expression elsewhere in the `datamap` section of the `style.tde` file.

Syntax

```
define: pattern_name "pattern"
```

`pattern_name` The name assigned to a pattern. The name specified can be up to 128 characters long, and can consist of alphanumeric characters, underscores, and hyphens.

`pattern` A regular expression describing a character or set of characters. For more information, see "Supported Regular Expressions" on page 371. The pattern specified must be enclosed in quotes.

dispatch

The `dispatch` statement is a child of the `datamap` statement. It supplies the rules for populating the dispatch field defined in the `style.ddd` or `style.ufl` file. It identifies the document body text to include in the full-word index. If there is no `style.dft` file, the document body text, as specified by the `dispatch` statement, is displayed for viewing.

The start and end of the document body can be identified by a line number or a pattern written as a regular expression.

Syntax

```
dispatch: field_name
/required = yes|no
/start-line = "string"
/start-pattern = "string"
/end-line = "string"
/end-pattern = "string"
/inclusive = yes|no
```

`field_name`
The name of the dispatch field as defined in the collection's `style.ddd` or `style.ufl` file. The document dispatch field may be assigned a field name other than DOC, but the field name must match the field name specified in the `style.dft` file.

`/required`
This optional modifier identifies whether the field is required in order for the Verity engine to include the document in the collection. If you specify `yes`, the Verity engine ignores the document if the field is not found. The default is `no`.

`/start-line`
This optional modifier identifies the line of the document on which the dispatch field begins. If neither `/start-line` nor `/start-pattern` is given, the field begins on line 1.

`/start-pattern`
This optional modifier identifies the pattern, represented by a regular expression, with which the dispatch field begins. If neither `/start-line` nor `/start-pattern` is given, the field begins on line 1.

`/end-line`
This optional modifier identifies the line of the document on which the dispatch field ends. If neither `/end-line` nor `/end-pattern` is given, the dispatch field ends at the end of the file. A dollar sign (`$`) can be used to signify the end of the file.

`/end-pattern`
This optional modifier identifies the pattern, represented by a regular expression, that occurs at the end of every document. If an `/end-line` modifier or an `/end-pattern` modifier is not given, the documents end at the end of the file.

`/inclusive`
This optional modifier identifies whether a specified start pattern and end pattern are to be included in the dispatch field. By default, these patterns are not included. If you specify YES, both patterns will be included in the dispatch field.

field

The `field` statement is a child of the `datamap` statement. It identifies the rules the Verity engine follows to parse data for a specified field name. Field values can be stored in collections. To store extracted field values in a collection, the field(s) must be defined in the `style.ufl` file. A `field` keyword must be present for each field for which you want to store extracted values.

Syntax

```
field: field_name {FILENAME|FILEPATH|FILETIME|FILESIZE|
LINE|PATTERN|TIME} num|"pattern"
   /required = yes|no
   /which = [1|###|LAST|ALL]
   /string-before = "string"
   /string-between = "string"
   /string-after = "string"
   /default = "field_value"
   /alsowrite = [field_name|"field_names"]
```

field_name — The name of a field for which you want to extract values from your documents. If this name corresponds to a name in the `style.ufl`, the Verity engine stores the extracted field values in the collection. Field names are case-insensitive and may contain alphanumeric characters, hyphens (-), and underscores (_). Note that field names cannot contain blank spaces.

`FILENAME` — The name of the source file containing the document, as in this example:

field: DOC_FILENAME `FILENAME`

`FILEPATH` — The fully-qualified pathname where the source file containing the document is located, as in this example:

field: DOC_PATHNAME `FILEPATH`

`FILETIME` — The time when the source file containing the document was last edited, as in this example:

field: DOC_FILETIME `FILETIME`

`FILESIZE`
The size of the source file, in bytes, that contains the document, as in this example:

field: DOC_FILESIZE `FILESIZE`

`LINE num`
Assigns the text at that line in the document to the specified field, as in this example:

field: DOC_LINE LINE 3

`PATTERN "`*`pattern`*`"`
The flag that precedes a regular expression that the Verity engine matches. The specified regular expression must appear in quotes as shown in this example:

`field: TITLE PATTERN "Title  :<.*>"`

`PATTERN "{`*`pattern_name`*`}"`
The flag that precedes a regular expression that the Verity engine matches. The `pattern_name` variable represents a macro which can be substituted for a long regular expression. The macro must surrounded by curly braces and must be specified in a define statement. An example is provided at the end of this chapter.

`TIME`
The time when the document was parsed. For example:

field: DOC_TIME `TIME`

`/required`
This optional modifier identifies whether the field is required in order for the Verity engine to include the document in the collection. If you specify `yes`, the Verity engine ignores the document if the field is not found. The default is `no`.

`/which`
This optional modifier specifies how the Verity engine behaves when multiple instances of a particular field are found in a document.

- `1`. the first instance of the field is used (the default is `1`).
- *`Number`*. The field with the given instance number is used.
- `LAST`. The last instance of the field is used.
- `ALL`. All instances of the field are used.

`/string-before`
This optional modifier identifies a string to be inserted before the field value.

`/string-between`

This optional modifier identifies a string to be inserted between field values when the Verity engine extracts many field values for the field. This option is only valid when the modifier `/which=all` is specified.

`/string-after`

This optional modifier identifies a string to be inserted after the field value.

`/default`

This optional modifier specifies the string that the Verity engine assigns the field if a field value is not found. If this modifier is not specified, the field is empty and appears blank if displayed in a Verity application.

`/alsowrite`

Using this modifier, you can store a parsed value in two fields: the field named by the `field` statement, and the field named by the `/alsowrite` modifier.

`/alsowrite` also allows you to populate more than two fields with the same value. Simply specify a space-separated list of field names delimited by quote marks:

```
/alsowrite = "Us1 Us2 Gx1 Gx2"
```

pre-process

The `pre-process` statement identifies the beginning of the pre-process section of the `style.tde` file. Note that no more than one `pre-process` statement should be included in the `style.tde` file. Each `pre-process` statement can have up to 1000 child statements.

Syntax

```
pre-process:
{
pre-processing
}
```

`pre-processing`	A set of statements that define the work done by the Verity engine when documents are initially parsed. For example, the `relative-path: NO` statement can be specified to build collections with full paths to documents (by default, relative paths are used).

tde

The `tde` statement identifies the control file as a `style.tde` file. It should be the first non-comment line after the `$control` statement.

Syntax

```
tde:
```

style.tde Example

Sample style.tde and style.ufl files for defining the use of a custom macro are shown here.

Listing 6-2 Example style.tde file

```
$control: 1
tde:
```

```
{
  pre-process:
  {
     datamap:
     {
        define: writename "<E.*>"
        field:  Writer PATTERN "{writename}"
          /required = yes
        dispatch: DOC
     }
  }
}
$$
```

The following `style.ufl` field definition would be used with the `style.tde` example in Listing 6-2.

```
data-table:  dad
{
   varwidth:  Writer  dxa
}
```

7

Defining Document Zones

Zone filtering allows an application to search within regions, or zones, of document text. Searching within document regions is an efficient way to find information. This chapter describes how to enable zone searching through the use of the zone filter for: SGML, HTML, internet-style email messages, internet-style Usenet news articles.

This chapter covers the following:

Zone Filter Overview

This section provides an introduction to zones, describes the kinds of documents that use zones, and explains the differences between fields and zones.

Introduction to Zones

Zones are specific regions of a document to which searches can be limited. The Verity engine uses the zone filter to build zone information into a collection's full-word index. The enhanced index permits quick and efficient searches using zones. A zone may be automatically defined by the zone filter, or you may define it in the `style.zon` file.

Zone searching is useful when you believe that limiting your search to a particular zone will produce more accurate search results. Speed of searching is not a factor, since searching a zone for information use the same amount of time as searching the entire document. Note however, that searching a zone is faster than field searching, since zone searching uses the fast search algorithm of the search engine, whereas field searching is a linear process.

Document Types

You can use the zone filter and search over zones for two specific types of documents:

- **Markup Language documents.** This includes both SGML and HTML. The zone filter includes built-in support for HTML. You can use the `style.zon` file to specify zones for any SGML document.
- **Internet Message Format documents.** This includes standard email and Usenet news messages. These documents must conform to the RFC822 standard. The zone filter includes built-in support for both email and Usenet news.

Zones vs. Fields

Fields are extracted from the document and stored in the collection for retrieval and searching, and can be returned on a results list. Zones, on the other hand, are merely the definitions of "regions" of a document for searching purposes, and are not physically extracted from the document in the same way that fields are extracted. The contents of a zone cannot be returned in a search results list.

A region of text in the content of a document must first be defined as a zone in order to be extracted as a field. Therefore, it can be a zone only, or it can be both a field and a zone. Whether you define a region of text as a zone only or as both a field and a zone depends on your particular requirements.

Advantages of Using a Field

- For numeric fields like dates, you can perform relative comparisons using the relational operators (=, >, <, >=, <=).

  For example, you can do a query like the following on a field: `date > may 1, 1993`. Because zones are not parsed for content, such searches cannot be performed on them.

- The value of a field can be returned on a results list.

  This is most useful for those parts of a document that help identify it, like the title and the author. Zones cannot be returned on a results list.

- Fields are stored with a collection, so the source document need not be accessed to get field values.

Advantages of Using a Zone

- For text regions of a document, searching a zone is much faster than searching a field.

  Arbitrary query searches can be restricted to a zone, but not to a field (which can be searched only with `CONTAINS` and relative comparisons).

- Zones can be any length with little impact on the index file size.

  This is because the source text is not stored in the binary collection files. Only a description of where each zone starts and ends is stored, so the zone's size does not matter. Field values are stored in a table in the collection, and therefore tend rapidly to increase the size of the collection.

Processing Order

Field parsing and populating in the document table are performed before full-text indexing. If zones are defined, they are interpreted during full-text indexing.

Zones and Zone Occurrences

When you extract a zone from a document you may be extracting a single zone or you may be extracting multiple occurrences of a zone. For example, a Usenet news or internet email document will only have one `Subject:` field. However, an HTML document may have several <h2> tags. When you create a zone, all of the <h2> tags are extracted, and all are searched when you submit a query on the zone. The specifics of zone searches are discussed in "Searching in Zones" on page 218.

Invoking the Zone Filter

By default, the Verity engine invokes the universal filter with the zone filter as a helper filter. If you open the default `style.uni` file you will see that it includes zone filter with options that invoke the built-in *modes* for processing HTML (`-html`), email (`-email`), and Usenet news (`-news`) files. Besides the built-in modes, you can also define and specify custom user modes, as described later in this chapter.

The zone filter can index and access for display documents in tagged ASCII formats, like HTML, SGML, email and Usenet news. The tagged regions of text are defined by the search engine as zones and these zones can be searched by users.

If, for a particular collection, you know you will only be indexing one of these tagged ASCII formats, and no other format, then you can avoid the overhead of the universal filter and instead specify a zone filter directly in the `style.dft` file as described later.

For more information about the universal filter, the syntax of the `style.uni` and `style.dft` files, see "Filtering and Formatting Documents" on page 95.

Specifying the Zone Filter

The zone filter can be invoked together with the universal filter or as a single filter. By default, the built-in zone modes (for HTML, email, and Usenet news) are all invoked with the universal filter. The zone filter specifications for the built-in zone modes are included in the default `style.uni` file.

To invoke the zone filter with the universal filter, you need to specify the filter in the `style.uni` file using the `type` keyword with the `/content-filter` modifier, as shown in the sample `style.uni` syntax here:

```
type: text/html
      /charset        = guess
      /def-charset    = 1252
      /content-filter = "zone -html -nocharmap"
```

Note The -nocharmap argument specifies that zone filter will not perform character set mapping; instead, the universal filter will use its character set recognizer.

To invoke the zone filter as a single filter, you need to specify the filter in the style.dft file using the field keyword and the /filter modifier, as shown in the sample style.dft file syntax here:

```
field: DOC
  /filter="zone -html"
```

If the zone filter is invoked as a single filter, the engine will index documents in the mode specified, so the collection will be limited to those documents.

Mode Options

The zone filter supports several filtering modes, each appropriate for a specific document type. To specify a particular zone-filter mode when invoking the zone filter, you need to use one of the options shown in Table 7-1 with the zone argument.

Table 7-1 Zone-filter modes

Option	Description
`-html`	Documents in Hypertext Markup Language format for World Wide Web.
`-email`	Internet email conforming to the RFC822 standard.
`-news`	Internet Usenet news conforming to the RFC822 standard.
`-usermode` *`modeName`*	User-defined mode for handling documents of type *`modeName`*. See "Custom Zone Definitions" for more information.
`(No option)`	Documents in SGML format.

Note that, if you specify no mode option when invoking the zone filter, it operates in a mode appropriate for SGML documents.

Specifying one of these modes causes a corresponding mode flag in the style.zon file to be defined. Use of mode flags allows a single style.zon file to be capable of handling all of the above types of files. The mode definitions are listed in Table 7-3 on page 189.

Note Several of the zone-filter modes support entity translation, and by default rely upon built-in tables of entity translations that are appropriate for the current mode and character set. Please see "Built-In Default Entities" on page 196 for more information.

Character Mapping Options

To filter a document with the zone filter, you need to know which character set a text file is written in to be able to parse it properly. The character mapping options are general enough that you can use them for all Web documents, even those in, for example, Korean, Chinese, Russian, and Czech.

When you use the zone filter a helper filter to the universal filter, the universal filter's character set recognizer should be used. In this case, you should include the `-nocharmap` option in your filter specification.

When you use the zone filter as a single filter for a collection, use one of the following options in the filter specification to instruct the zone filter how to perform character mapping.

Option	Description
`-precharmap` *`name`*	Map from the named character set before parsing the document. The name variable is the name of the character set to map from.
`-autocharmap`	Guess the character set of the document, and then map from that character set to the internal character set before parsing the document.
`-nocharmap`	Do not perform any character set mapping before parsing the document. When the `-html` flag and the `-nocharmap` flag are given together, the document will not be mapped from the HTML standard of 8859 before being parsed.

For example, to invoke the zone filter as a single filter and to perform character mapping before parsing the documents, you need to specify the filter in the `style.dft` file using the `field` keyword and the `/filter` modifier, as shown in the sample `style.dft` file syntax:

```
field: DOC
  /filter="zone -html -precharmap 850"
```

A key purpose of the -precharmap and the -autocharmap options is to support Japanese and other Asian languages in which Web pages are commonly written in a multibyte character set rather than the HTML standard of ISO-8859-1. In Japan in particular, there are three common character sets, and a Web page might be written in any of those three.

In the previous example, the -autocharmap option is a convenient way of handling a Web page written in an unknown character set. The same mechanism also works for Web pages written in other locales, as long as that locale supports a character set detection function.

Extracting META Tags as Fields

The universal filter supports filtering <META> tags, which are typically defined in HTML documents, using the special flt_meta content filter with the zone filter. The flt_meta filter can be specified for the text/html document type in the style.uni file.

The default style.uni file automatically invokes the flt_meta content filter with the zone filter. Here is an example of a short style.uni file that can filter HTML documents with <META> tags (the type: statement here also appears in the default style.uni file):

```
$control: 1
 types:
 {
   autorec: "flt_rec"
   autorec: "flt_kv -recognize"
   type: text/html
     /charset        = guess
     /def-charset    = 1252
     /content-filter = "zone -html -nocharmap"
     /content-filter = "flt_meta"
   # if we get anything else, just skip it
   default:
     /action = skip
 }
 $$
```

The flt_meta filter watches markup tokens in the document stream. When the filter encounters a <META> tag, it produces a field token based on the tag and then the field token is stored as a field in the collection. In the collection’s document table, the field name is the name of the <META> tag’s name attribute, and the field value is the value of the content attribute in the <META> tag.

A sample <META> tag in HTML is shown here:

```
<META name="Abstract" content="This is a long document">
```

When filtering the HTML above, the `flt_meta` filter produces a field token of this form:

```
ABSTRACT: This is a long document
```

In the default `style.sfl` file, the field name `Abstract` is populated with the value `This is a long document`. A field definition that corresponds to the `<META>` tag's name attribute must appear in the `style.ufl` or `style.sfl` in order for the field to be populated by the filter. In the example above, the field named `Abstract` is aliased to the `Snippet` field in the default `style.sfl` file so you would not need to add a field definition.

Extracting Zones as Fields

The zone filter supports a method for extracting zones as fields that differs from the method used by the `flt_meta` filter to extract `<META>` tags as fields. The zone filter watches HTML in the document stream and produces a field tokens based on the zone name(s) specified in the `style.zon` file, where a zone name corresponds to a tag name. In the collection's document table, the field is defined as the tag name and the tag value is the field value.

For example, when the zone filter encounters this HTML:

```
<TITLE>This is the title</TITLE>
```

the filter produces the following field token:

```
TITLE: This is the title
```

Since `TITLE` is defined as a standard field by default, the zone filter populates the `TITLE` field with the value "This is the title." For more information about extracting zones as fields, see "Defining Document Zones" on page 179.

style.zon File Syntax

You can use the `style.zon` file to specify the tags you want to create as zones by using the `element` and `attribute` keywords. The examples in the description of the `style.zon` file refer to common entities in SGML.

style.zon File Structure

The structure of the `style.zon` file is as follows:

```
$control: 1
zonespec:
[/ignoreattributes = YES|NO]
[/html] | [/email] | [/news]
{
    .
    .
    .
}
$$
```

The `style.zon` file must reside in the `style` directory of the collection. The first line in the file must be `$control: 1`, and `zonespec:` must be the first uncommented line following it. The file must end with `$$` on a line by itself.

The content of the file depends upon the types of document for which you are creating zones, and how you want the various zones stored in the collection. The syntax for the various `style.zon` keywords and sample `style.zon` sections for the various document types are included in "Zones for Markup Language Documents" on page 198 and "Zones for Internet Message Format Documents" on page 203.

zonespec Modifiers

The `zonespec` keyword is immediately followed by one of the modifiers listed in Table 7-2.

Table 7-2 zonespec modifiers

Modifier	Description
`/ignoreattributes`	Specify `YES` or `NO`. The default is `YES`. Ignores tag attributes unless overridden by a statement beneath it.
`/html`	Specifies that, when the zone filter is invoked with the `-html` option, the zone filter should use this `zonespec` for processing documents. If this modifier is not present, internally coded processing consistent with the HTML standard occurs.

Table 7-2 zonespec modifiers

Modifier	Description
/email	Specifies that, when the zone filter is invoked with the -email option, the zone filter should use this zonespec for processing documents. If this modifier is not present, internally coded processing consistent with the standard for RFC822 email files occurs.
/news	Specifies that, when the zone filter is invoked with the -news option, the zone filter should use this zonespec for processing documents. If this modifier is not present, internally coded processing consistent with the standard for Usenet news files occurs.

If none of the -html, -email, -news, or -user modes is specified when the zone filter is loaded, then the zone filter uses SGML mode. In SGML mode and in user mode, no zonespec modifier is required.

In modes other than SGML and user, the zonespec keyword must have the appropriate mode modifier (/html, /email, or /news) or else style.zon will be ignored and hard-coded settings will be used.

The zonespec keyword appears as follows when using the /ignoreattributes modifier.

```
$control: 1
zonespec:
  /ignoreattributes = no
{
  element: *
}
$$
```

Conditionally Configuring Modes

When it parses style.zon, the zone filter defines one of the flags listed in Table 7-3, based on which zone-filter mode option (see Table 7-1 on page 183) was specified in style.uni. For example, if the filter is invoked with the -HTML option, the ZONEMODE_HTML flag becomes defined in style.zon.

These flags are used in style.zon to conditionally adjust the configuration according to zone-filter mode. By adding conditional expressions based on the following mode definitions to your style.zon file, you can effectively create multiple versions of style.zon, one for each type of zone-filter mode that you will use. Within the section of style.zon covered by each flag, you can then include a zonespec line followed by its appropriate modifier, in turn followed by the elements, attributes, and entities appropriate for processing that type of document.

Table 7-3 Optional mode definition flags in style.zon

Mode	Description
ZONEMODE_SGML	Conditionally activates the section that handles SGML documents. This flag is defined in style.zon when the zone filter is invoked without any of the options -html, -email, -news, or -user.
ZONEMODE_HTML	Conditionally activates the section that handles HTML documents. This flag is defined in style.zon when the zone filter is invoked with the option -html.
ZONEMODE_EMAIL	Conditionally activates the section that handles RFC822 email documents. This flag is defined in style.zon when the zone filter is invoked with the option -email.
ZONEMODE_NEWS	Conditionally activates the section that handles Usenet news documents. This flag is defined in style.zon when the zone filter is invoked with the option -news.
ZONEMODE_USER	Conditionally activates a custom, user-defined section of style.zon that handles other types of documents. This flag is defined in style.zon when the zone filter is invoked with the option -usermode *modeName*. See also "Implementing Multiple User Modes in style.zon," below.

Note that if none of the -html, -email, -news, or -user modes is specified when the zone filter is loaded, the ZONEMODE_SGML definition is activated in style.zon. No zonespec modifier is required for the section of style.zon covered by ZONEMODE_SGML.

See "Default style.zon File" on page 221 for an example of how these zone definitions appear in style.zon.

Implementing Multiple User Modes in style.zon

If you need multiple, differently-configured custom modes, you can modify style.zon so that it identifies the correct mode to use by testing the value of $ZONEMODE, which—if the zone filter is called with the -usermode *modeName* option—will have the value *modeName*.

Here is an example of how this might appear in `style.zon`, for *`modeName`* values of `XML_internal` and `XML_external`:

```
$SUBST: 1
$if $ZONEMODE == XML_internal
... # Configuration for internal docs
$elif $ZONEMODE == XML_external
... # Configuration for external docs
$else
 ...  # Configuration for all other docs
$endif
$SUBST: 0
```

The element Keyword

The `element` keyword specifies extraction or exclusion of elements (tags) in the document being processed. It uses the following syntax:

`element:` *`elementname`*

where *`elementname`* specifies the name of the element (that is, the tag) you want to extract as a zone. Element names are case-insensitive. To extract all tags as zones, use `* for` *`elementname`*. (See "Wildcards" on page 196 for more information about using the asterisk.)

You can use the following optional modifiers with the `element` keyword.

Modifier	Description
`/ignore`	Specify `YES` to ignore the specified element. If you use the wildcard character (`*`) for *`elementname`*, only those elements specified with the `/ignore=yes` modifier are ignored. If you do not use the asterisk, all the elements specified are extracted and those omitted are ignored.
`/field`	Specify `YES` to extract the specified element as a field as well as a zone. See “Defining Zones for Virtual Documents” on page 213. The extracted field value is stored in the *`elementname`* field. To extract attribute names, you must also extract the element name.
`/nonwordbreaking`	Specify `YES` to prevent a zone boundary from occurring in the middle of a word. Otherwise, occurrences of this zone's tag in the middle of a sequence of characters will split those characters and cause them to be indexed as two words. If you specify `/nonwordbreaking`, zone searches for the entire word that would otherwise have been split will succeed, and searches for the word’s fragments will not succeed.
`/noindex`	Specify to YES to cause all instances of this zone to be surrounded by `<noindex>` tags. This will prevent content of this zone from being searched. See “Special Noindex and Noextract Zones” on page 216 for more information. **Note:** this modifier is disregarded if the `/ignore` modifier has the value `YES`. It is also disregarded for any zones named `noindex`.
`/noextract`	Specify to YES to cause all instances of this zone to be surrounded by `<noextract>` tags. This will prevent content of this zone from being used for feature extraction. See “Special Noindex and Noextract Zones” on page 216 for more information. **Note:** this modifier is disregarded if the `/ignore` modifier has the value `YES`. It is also disregarded for any zones named `noextract`.

There are two approaches to specifying the elements to extract as fields. The first is to specify the asterisk, and then use the `/ignore` modifier to list any tags you do not want extracted. The following is an example of the first approach:

```
$control: 1
zonespec:
```

```
{
  element: *

  element: heading3
    /ignore = yes

  element: list-item
    /ignore = yes
}
$$
```

In this case, all elements are extracted as zones except for `heading3` and `list-item`, which are ignored.

The second approach is to explicitly list only those elements you want extracted. The following is an example of the second approach.

```
$control: 1
zonespec:
{
  element: header
  element: body
  element: title
  element: textzone
  element: section
  element: sub-section
  element: footnote
  element: appendix
}
$$
```

In this case, only the eight elements specified are extracted as zones. All the rest are ignored.

The attribute Keyword

The `attribute` keyword specifies extraction or exclusion of attributes within a tag. It is entered in the `style.zon` file as a child of `element` and uses the following syntax:

`attribute:` *`attributename`*

where `attributename` specifies the name of the attribute you want to extract as a zone. Attribute names are case-insensitive. To extract all attribute names as zones, use `*` for `attributename`. (See "Wildcards" on page 196 for more information about using the asterisk.)

You can use the following optional modifiers with the `attribute` keyword.

Modifier	Description
`/ignore`	Specify YES to ignore the specified attribute. To extract the attribute, specify `NO` (default). If you use the asterisk for `attributename`, only those attributes specified with the `/ignore=yes` modifier are ignored. If you do not use the asterisk, all the attributes specified are extracted and those omitted are ignored.
`/field`	Specify `YES` to extract the specified attribute as a field value as well as a zone. See "Defining Zones for Virtual Documents" on page 213. When a /field=YES modifier is assigned to an attribute, the attribute name and value are prepended to the field value named by the element name. **Note:** Using `/field=YES` does not cause the attribute information to be extracted into its own field.
`/default`	Specify the default attribute value you want to use when the attribute name does not occur in the zone tag.
`/values`	Specify values that may appear in a tag without the corresponding attribute name.

The following is an example of the attribute keyword and its modifiers:

```
$control: 1
zonespec:
{
  element: header
  element: body
  element: title
  {
   attribute: company
     /default: "IBM"
  }
  element: textzone
  element: section
  element: sub-section
  element: footnote
  element: appendix
}
```

```
$$
```

The default behavior is to extract all attributes as zones. In some circumstances, you can suppress this behavior:

- With wildcards (See “Wildcards” on page 196), you can set new default behavior like this:

  ```
  element: *
  {
      attribute: *
      /ignore = yes
  }
  ```

- If you specify the `/ignoreattributes=yes` modifier, all attributes are ignored.
- If an element is ignored (for example by means of an `/ignore=yes` modifier), that element’s attributes are ignored as well.

In either case, you can override the ignore behavior and extract an attribute as a zone by specifying `/ignore=no` for that attribute.

The entity Keyword

The `entity` keyword specifies the translation of entities to their equivalents. It uses the following syntax:

`entity: *name* "*value*"`

where *`name`* is the `name` of the entity as it appears in the document, and *`value`* is the way you want the entity to display. You can use the following optional modifiers with the `entity` keyword.

Modifier	Description
`/ignore`	Specify either `YES` or `NO`. The default is `NO`.
`/punct`	If present, specifies that this entity is to be considered punctuation (not a character that can be part of a word).

Entities in SGML are used to specify characters that would otherwise be considered as part of the markup language or that cannot be typed on the normal keyboard.

The entity begins with an ampersand (&) and ends with a semicolon (;) or white space. No space is permitted between the ampersand character and the following entity name. The entity name consists of alphanumeric characters plus any combination of underscores, dashes, and number signs (#). If the entity is terminated with a semicolon, the semicolon is also part of the string that is replaced by the equivalent string. If the entity is terminated by a whitespace character, that whitespace is not considered part of the string that is replaced.

For example, assume the following entities and their translations:

Entity	Translation
`&amp;`	&
`&greaterthan`	>
`&lessthan`	<

The `style.zon` file would then appear as follows:

```
$control: 1
zonespec:
{
  entity: amp "&"
  entity: lessthan "<"
  entity: greaterthan ">"
}
$$
```

The following is a sample of how the actual document would appear in ASCII text form:

```
Here is some text. First an entity delimited by a semicolon:
S&amp;P's stock index. Second, entities delimited by a spaces:
the &greaterthan character and the &lessthan character.
```

Using the above `style.zon` file, the resulting document would then appear as follows

```
Here is some text. First an entity delimited by a semicolon:
S&P's stock index. Second, a entities delimited by spaces:
the > character and the < character.
```

If an entity is encountered and no translation is given for it in the `style.zon` file or in the built-in rules, then the text of that entity is passed through the filter unchanged.

Entity Substitution

When the zone filter does entity substitution, it literally replaces the entity string (the ampersand followed by the symbolic name) with the string (typically just one character) specified in the substitution table. An exception to this behavior has been incorporated into the built-in HTML filter which interprets non-alphabetic entities as punctuation tokens. Using the built-in HTML filter, entities such as &amp, &lt, and &gt are not streamed to the literal characters: &, <, and >. A custom HTML zone filter will perform entity substitution for all entities, and all entities are streamed to literal characters.

Built-In Default Entities

If the zone filter uses `style.zon` but `style.zon` defines no entities, the zone filter uses a set of built-in default entities appropriate for the current zone-filter mode and character set. If `style.zon` defines any entities, none of the built-in entities are used. Therefore, if you define any entities in `style.zon`, you must define all that you need.

To view the built-in set of default entities, you can use the `-dump` argument to the zone filter. See “Dumping style.zon Information” on page 208 for details.

Wildcards

`style.zon` recognizes the asterisk (`*`) as a wildcard character. An asterisk can be used as a wildcard for the name of an element, header or attribute. You use wildcards to establish default values used for subsequent definitions in `style.zon`, and to establish settings to be used for zones or headers that have no definition listed in `style.zon`.

When the zone filter parses `style.zon`, any settings established with a wildcard will remain in effect as default modifiers until overridden by another wildcard. These default modifiers will be inherited by subsequent keywords that do not have explicit modifiers. (The `/ignore` modifier is excepted, and is never inherited.) In the following example, the `subject` and `from` headers are extracted to fields, but the `date` header is not.

```
zonespec:
/email
/news
{
    header: *
    /field = YES
    header: subject
    header: from
    header: *
    /field = NO
    header: date
}
```

Note also that whatever wildcard values are in effect at the end of parsing of `style.zon` will become the values used for any headers or elements that are not explicitly defined in `style.zon`. The `/ignore` modifier is also applied to those elements or headers.

The following modifiers can be used with header wildcards:

- `/ignore` (affects only undefined zones)
- `/field`

The following modifiers can be used with element wildcards:

- `/ignore` (affects only undefined zones)
- `/field`
- `/nonwordbreaking`
- `/noindex`
- `/noextract`

A wildcard attribute can also be specified for a wildcard element, and will establish default attribute settings for all subsequent elements. Note that, other than a wildcard attribute, any other attributes specified for a wildcard element are ignored.

The following modifiers can be used with attribute wildcards:

- `/ignore` (not inherited by explicitly listed attributes)
- `/field`

Note that the `/ignoreattributes` modifier to `zonespec` is equivalent to:

```
element: *
{
  attribute: *
    /ignore = YES
}
```

style.zon Default Behavior

If no `style.zon` exists, or if a mode is used for which the `zonespec` in `style.zon` does not have the appropriate modifier, then zone filter will revert to default behavior that preserves backward compatibility with previous versions of Verity software. In these cases, the zone filter will generally parse documents in accordance with the appropriate industry standard for the specified mode.

Zones for Markup Language Documents

Markup languages use tags embedded in the text of documents to specify the document's structure and formatting. Viewers and print programs are designed to read the tags and display or print the document appropriately.

The international standard for markup languages is SGML, or Standard Generalized Markup Language. SGML is the basis for HTML, or Hypertext Markup Language, which is the means used to create pages for the World Wide Web. A newer markup language named XML (Extensible Markup Language) uses user-definable tags to extend the capabilities offered by HTML.

There are a number of reasons why you might want to search HTML, SGML or XML tags as zones. For example, if you are looking for information on Ecuador, you may want to search for the word "Ecuador" on the title or first-level heading. Using the title or first-level heading as a zone will help ensure that documents retrieved have Ecuador as their primary focus, rather than simply being briefly mentioned in the body of the text.

Note For indexing XML files, Verity recommends that you use the XML filter, described in "The XML Filter" on page 132.

How the Zone Filter Parses Markup Language Documents

When the zone filter encounters a start zone tag, it opens a new zone. When it encounters an end zone tag, it closes that zone. The indexer makes a zone out of all the text between the two tags. The tags themselves are ignored during filtering.

The syntax for a start zone tag is:

```
<name[attributes...]>
```

The syntax for an end zone tag is:

`</`*`name`*`[`*`attributes...`*`]>`

That is, a start tag begins with a left angle bracket, followed by the element name. The end tag starts with a slash and a left angle bracket, followed by the element name. There can be no space between the left angle bracket and the following characters. The element name is followed by zero or more attributes. Attributes can be arbitrary text, including strings and whitespace characters, but frequently have the form:

`AttributeName=Value`

where *`Value`* can be an identifier, a string literal, a URL, or anything else, so long as it does not contain a right angle bracket.

Exclamation point and question mark metatags are parsed, but ignored. Examples are:

```
<!name[attributes...]>
<?name[attributes...]>
```

Here is an example of a document containing valid SGML text and tags.

```
<HEAD> This text is in the header zone. </HEAD>
<BODY> This text is in the body zone.
  <section>
    This text is in a body zone AND the section zone, which is
    nested inside the body zone.
  </section>
</BODY>
```

Tag names are case-insensitive. They can consist of all the alphabetic characters (upper and lower case), numeric characters, the dash, the underscore character, or the number sign (#).

Implicit Zone Endings

The zone filter implicitly ends zones that have not been explicitly ended with an end tag. For example:

```
<ul>
  <li> This is the li zone.
</ul>
This is outside all zones.
```

The `</ul>` tag ended the `ul` zone, but also implicitly ended the `li` zone. It is equivalent to the following, in which the implicit end tag is underlined:

```
<ul>
  <li> This is the li zone. </li>
</ul>
```

Zone end tags are implicit in only two cases:

- At the end of the file, all still-open zones are automatically closed.
- When an end tag is encountered, its matching start tag is found, and all zones that were opened between those matching start and end tags that are not closed are implicitly closed.

The filter does not perform any implicit end of zones when a start tag is encountered. For example, here is an HTML construct:

```
<ul>
   <li> This is in the li zone, which is nested in the ul zone.
   <li> This is another li zone, which is also nested in the ul
        zone. In HTML, this li ends the previous one. With the
        zone filter, it does not.
</ul>
```

It would be interpreted with the following implicit end tags:

```
<ul>
  <li> This is in the li zone, which is nested in the ul zone.
  <li>This is another li zone, which is also nested in the ul
      zone.</li>
  </li>
</ul>
```

Implicit end tags are not handled when start tags are encountered because it is not very useful to search contiguous zones. Searching for “text in the `li` zone” in the preceding examples has little meaning because all the text is already in the `li` zone.

Zones for HTML Documents

HTML is based on SGML. It is, essentially, one SGML DTD. However, because of its popularity and widespread use, most HTML tags and entities are commonly recognized and read by Web browsers and authoring tools. Therefore, HTML tags and entities are built into the zone filter.

Zone Filter Specification for HTML

The following zone filter specification in the `style.uni` file is appropriate for HTML documents:

```
type: text/html
      /charset        = guess
      /def-charset    = 1252
      /content-filter = "zone -html -nocharmap"
```

The above specification will invoke the built-in HTML filter.

Supported HTML Tags

The zone filter recognizes most standard tags through HTML 4.01 and XHTML 1.0. It automatically extracts certain tags as zones and ignores others.

For a list of the HTML tags that are supported and the tags that are ignored, see the default version of `style.zon` ("Default style.zon File" on page 221).

Supported HTML Entities

HTML uses entities to represent certain characters. An entity is a representation of a character that, when interpreted by the HTML browser, displays the proper character. Entities in HTML are used to specify characters that would otherwise be considered as part of the markup language or that cannot be typed on the normal keyboard. For example the < is used to denote the beginning of a tag. If you want this character to display in an HTML browser, you need to enter the entity `lt` in your HTML document. Likewise, you can display the character Á by using the entity `Aacute`. The zone filter supports all of the ISO8859-1 entities as specified by the HTML 4.01 specification.

Additional HTML Parsing Rules

The zone filter observes the following additional rules for HTML:

1. No header lines are parsed, and the parsing starts off in markup language parsing mode.
2. The title tag is extracted as a zone *and* a field. You must make sure to put the title in your `style.ufl` file to be able to store this field. (For more information, see "Defining Zones as Collection Fields" on page 210.)
3. The character set for HTML pages is ISO 8859. The zone filter automatically translates the characters into the internal character set specified at startup.

Zones for SGML Documents

SGML documents rely on a Document Type Definition (DTD) to define their tags. Unlike HTML, different groups of SGML documents use different DTDs. Therefore, you must define the zones you want to extract for each group of SGML documents that share a common DTD.

There is a limitation to using a Verity zone filter for SGML documents. The length of any SGML attribute name plus its corresponding attribute value is limited to 256 bytes. If the indexing engine meets this limitation for an attribute, the engine truncates the zone attribute value.

Zone Filter Specification for SGML

The following zone filter specification in the `style.uni file` is appropriate for SGML documents:

```
type: text/sgml
      /charset        = guess
      /content-filter = "zone -nocharmap"
```

Using style.zon with SGML Documents

You can use the `style.zon` file to specify the SGML tags you want to create as zones by using the `element` and `attribute` keywords. You can specify the conversion of SGML entities using the `entity` keyword. A sample `style.zon` file for SGML is shown in Listing 7-1.

Listing 7-1 Example style.zon file

```
$control: 1
zonespec:
{
  element: *

  element: heading3
    /ignore = yes

  element: list-item
    /ignore = yes
}
$$
```

For complete information about using the `style.zon` file, see “Custom Zone Definitions” on page 207.

Zones for Internet Message Format Documents

The zone filter recognizes documents in internet message format that conform to the RFC822 standard. This includes most standard email and Usenet news messages.

How the Zone Filter Parses Internet Message Format Documents

The zone filter parses the headers of Internet-style email and Usenet news messages to create zones.

For example, the following email message can be parsed to extract zones from the headers automatically.

```
From johns@verity.com Thu Dec 15 11:38:18 1994
From: John Smith <johns@verity.com>
Received: (from johns@localhost) by grimaldi
 (8.6.6.Beta9/8.6.6.Beta9) id LAA12705 for johns; Thu, 15 Dec
 1994 11:36:35 -0800
Message-Id: <199412151936.LAA12705@grimaldi>
Subject: test message
To: johns (John Smith)
Date: Thu, 15 Dec 1994 11:36:34 -0800 (PST)

This is a test message.
John
```

Here is the same document with implicit start and end zone markers for the above message. The implicit zone starts and ends are surrounded by square brackets and are underlined for easy identification.

```
From johns@verity.com Thu Dec 15 11:38:18 1994
From: [from-beg] John Smith <johns@verity.com>
[from-end] Received: (from johns@localhost) by grimaldi
 (8.6.6.Beta9/8.6.6.Beta9) id LAA12705 for johns; Thu, 15 Dec
 1994 11:36:35 -0800
Message-Id: <199412151936.LAA12705@grimaldi>
```

```
Subject: [subject-beg] test message
[subject-end] To: [to-beg] johns (John Smith)
[to-end]Date: [date-beg] Thu, 15 Dec 1994 11:36:34 -0800 (PST)
[date-end]

This is a test message.
John
```

Header lines should conform to the RFC822 standard for email and news messages. RFC822 specifies the following syntax:

```
Header-line-name: data data data \n
 [<whitespace>more data, more data more data \n] ...
```

The first line of a header line must begin with the header line name, which can consist entirely of alphanumeric characters, underscores, or dashes, followed by a colon. The rest of the line until the return character is the text of the header line. Header line names, like tag names, are case-insensitive. (For example, `to` matches `To`.)

Optionally, the header line can be continued on the next line with a continuation line. Lines whose first character is a whitespace character are continuation lines. The text of the entire continuation line is included as part of the previous header line. For example, the `To` header line in the following email spans multiple lines. Again, zone starts and ends are underlined.

```
From:[from-beg] John Smith <johns@verity.com>
[from-end]Subject:[subject] another test message
[subject-end]To:[to-beg] johns (John Smith),
       toddq@verity.com (Todd Quidnunc),
       mick@verity.com (Mickey O'Donnicker),
       ralphp@verity.com (Ralph Poobah)
[to-end]
```

The header section of a document is ended by the first line that contains only whitespace characters, or that starts with an SGML element tag. After that point, the parser reverts from header line parsing to SGML element parsing. If for some reason you have an internet message format document that contains embedded markup language tags, you can specify those tags in the `style.zon` file and they will be extracted as zones.

Zone Filter Specification for Email

The following zone filter specification in the `style.uni` file is appropriate for email documents:

```
  type: message/rfc822
        /charset         = guess
        /def-charset     = 1252
        /content-filter = "zone -email -nocharmap"
```

The above specification will invoke the built-in email filter. The rules for the built-in email filter are as follows:

- The following header lines are extracted as zones: `Cc`, `Bcc`, `Organization`, `To`, `From`, `Subject`, and `Date`.
- The following headers are extracted as fields as well as zones: `To`, `From`, `Subject`, and `Date`. These fields must be listed in the `style.ufl` file. (For more information, refer to "Defining Zones as Collection Fields" on page 210.)
- No markup language element tags are extracted as zones.
- No markup language entities are translated.

Using style.zon with Email

You can customize the default parsing rules for email documents by altering the default version of `style.zon`, described in "Default style.zon File" on page 221. When setting up the `style.zon` file to process email, you use the `header` keyword.

The `header` keyword specifies extraction or exclusion of header lines. The syntax is as follows:

```
header: headername
```

where *`headername`* specifies the name of the header line you want to extract as a zone. Header names are case insensitive. To extract all header names as zones, use `*` for *`headername`*. You can use the following optional modifiers with the `header` keyword.

Element	Description
`/ignore`	Specify `YES` to ignore the specified header. If you use the asterisk for *`headername`*, only those attributes specified with the `/ignore=yes` modifier are ignored. If you do not use the asterisk, all the headers specified are extracted and those omitted are ignored.
`/field`	Specify `YES` to extract the specified element as a field as well as a zone. See "Defining Zones for Virtual Documents" on page 213.

There are two approaches to specifying the header lines to extract as fields.

1. Specify the asterisk, and then to list any tags you do not want extracted using the `/ignore` modifier.

2. Explicitly list only those elements you want extracted.

The following is an example of the first approach:

```
$control: 1
zonespec:
  /email
  {
    header: *

    header: received
      /ignore = yes

    header: message-id
      /ignore = yes
  }
$$
```

In this example, all headers are extracted as zones, except received and message-id, which are ignored.

The following is an example of the second approach:

```
$control: 1
zonespec:
  /email
  {
    header: received

    header: message-id
  }
$$
```

In this example, only received and message-id are extracted as zones. All others are ignored.

Zone Filter Specification for Usenet News

The following zone filter specification in the `style.uni` file is appropriate for Usenet news documents:

```
type: message/news
      /charset        = guess
      /def-charset    = 1252
```

```
        /content-filter = "zone -news -nocharmap"
```

The above specification will invoke the built-in Usenet news filter. The rules for the built-in Usenet news filter are as follows:

- The following header lines are extracted as zones: `Organization`, `Summary`, `Newsgroups`, `From`, `Subject`, `Date`, and `Keywords`.
- The following headers are extracted as fields as well as zones: `From`, `Subject`, `Date`, and `Keywords`. These fields must be listed in your `style.ufl` file. (For more information, refer to "Defining Zones as Collection Fields" on page 210.)
- No markup language element tags are extracted as zones.
- No markup language entities are translated.

Using style.zon with Usenet News

You can customize the default parsing rules for Usenet news documents by altering the default version of `style.zon`, described in "Default style.zon File" on page 221. As with email, you use the `header` keyword when setting up the `style.zon` file to process news files.

Custom Zone Definitions

You can customize how zones are defined by modifying any of the standard mode definitions (see Table 7-3 on page 189) provided in the default version of `style.zon`, or by creating a custom mode definition to suit your needs.

Note that if a mode is specified for which no `zonespec` section appears in `style.zon`, or if there is no `style.zon`, the zone filter reverts to hard-coded settings that preserve backward compatibility with earlier versions of K2.

For most implementations, you can create custom definitions by modifying the default definitions in `style.zon`. However, some modes make use of hard-coded default settings; for example, HTML and SGML modes typically use hard-coded entity translation tables that are not visible in `style.zon`. In such a case, it can be useful to dump the definitions of the mode, as described next, so you can modify them for use in your custom `style.zon` file.

Dumping style.zon Information

You can dump the contents of the `style.zon` configuration settings used by any zone-filter mode (including any hard-coded definitions that are not actually in `style.zon`) to standard output. This can be useful in various circumstances, including debugging the `style.zon` file and modifying the hard-coded behavior of certain modes.

1. Set the `-dump` flag in `style.uni`. Here is an example zone filter specification with the `-dump` flag for the HTML mode:

```
type: text/html
      /charset        = guess
      /def-charset    = 1252
      /content-filter = "zone -html -dump"
```

2. Run an indexing tool, such as `mkvdk`, on a document and pipe the output to a file.

When you attempt to index a document with the `-dump` flag present in the `style.uni` file, the zone filter prints to the standard output a `style.zon` file with the settings that are in effect at the time of filtering. After the `style.zon` file is printed, the actual indexing does not take place.

The `-dump` option produces output in the character set of the current locale. The output can be mapped to another character set using the `-charmap` option of `mkvdk`.

Modifying Default Behavior

In general, you can modify the zone filter's default behavior for any document type by editing the appropriate section of the default `style.zon` file. In some circumstances, such as when you must alter a built-in setting such as an entity, you may need to apply changes to a complete configuration dump. In these circumstances, do the following:

1. Dump the contents of the built-in mode you want to modify, as described above.
2. Edit the output to reflect the behavior you want.
3. Use the modified configuration in your `style.zon` file.

   For use with the `-html`, `-email`, or `-news` options, be sure to include the `/html`, `/email`, or `/news` modifier for your `zonespec` keyword.

4. If the universal filter is being used, make sure the appropriate zone-filter specification is in the `style.uni` file. Otherwise, make the specification in the `style.dft` file, as described in "Invoking the Zone Filter" on page 182.

Attribute Extraction

Attributes can be extracted into a field named by an `element` keyword. This means the element value and the one or more attribute values are stored together in the same collection field. Consider the following `style.zon` file:

```
$control: 1
zonespec:
  /ignoreattributes = no
{
  element: name
    /field = yes
  {
    attribute: first
      /field = yes
    attribute: last
      /field = yes
  }
}
$$
```

A sample document with attributes to be indexed is shown here:

```
This is <name first="emily" last="shaffer">AAA</name>here.
Another is <name first="al" last="jones">ZZZ</name>here.
```

When the style.zon file above is in effect and the document above is indexed, the following behavior occurs:

- A field named “name” is populated with:

```
"first=al last=jones ZZZ"
```

- The fields named “first” and “last” are not populated.
- The `/field=yes` modifier on the `element` statement indicates that the field is populated (zone contents are extracted and stored in the field).
- Attributes that are children of the element and that have the `/field=YES` modifier defined cause an attribute name and value string to be prepended to the field. The format of the attribute string is:

```
attributename=attributevalue
```

- Multiple occurrences of the `name` element in the document cause the field value to be overwritten. The last occurrence encountered is saved in the field. The sample document contains two occurrences of the name element, so the values in the second instance were saved.

Defining Zones as Collection Fields

Any zone can also be extracted as a collection field. The differences between zones and fields are described under “Zones vs. Fields” on page 180.

The following lines in a style.zon file show how to extract the To, From, and Subject lines from an email message as fields as well as zones:

```
$control: 1
zonespec:
{
  # Extract all header lines of this email message as zones
  header: *

  # also extract these three header lines as fields as well as
  # zones
  header: From
    /field = yes

  header: To
    /field = yes

  header: Subject
    /field = yes
}
$$
```

Fields listed in a style.zon file must also be listed in the collection’s style.ufl or style.sfl file. Otherwise, when the zone filter extracts these fields, the indexer will have no place to store the values and the values will be ignored.

The field definitions for the Verity standard fields are included in the default style.sfl file. In this file, there are field definitions for several fields including “Subject”, which is aliased to the field named “Title”. All of the built-in zone filter modes automatically populate “Title” by default. The email and news zone modes populate the “To” and “From” fields.

While the standard field definitions cause the zone filter to define zones as collection fields, sometimes you need to create custom field definitions. If you are using the HTML zone mode, and you want to define the "To" and "From" zones as custom fields, then you need to provide a field definition corresponding to a zone name in the `style.ufl` file.

Here is the data table of the `style.ufl` file to use with the previous `style.zon` file:

```
data-table:    ddf
  {
    # User fields go here. These fields also listed in
    # the style.zon file
    varwidth:   From            dd1
    varwidth:   To              dd4
  }
```

For information about making a field definition in the `style.ufl` file, see "Defining Collection Fields" on page 145.

Extracting HTML Zones as Fields

The zone filter supports a method for extracting zones as fields that differs from the method used by the "flt_meta" filter to extract META tags as fields (as described in the next section, "Extracting META Tags as Fields" on page 185).

The zone filter watches HTML in the document stream and produces a field tokens based on the zone name(s) specified in the `style.zon` file, where a zone name corresponds to a tag name. In the collection's document table, the field is defined as the tag name and the tag value is the field value.

For example, when the zone filter encounters this HTML:

```
<TITLE>This is the title</TITLE>
```

the filter produces the following field token:

```
TITLE: This is the title
```

Since TITLE is defined as a standard field by default, the zone filter populates the TITLE field with the value "This is the title".

Extracting META Tags as Fields

The universal filter supports filtering <META> tags, which are typically defined in HTML documents, using the special “flt_meta” content filter with the zone filter. The “flt_meta” filter can be specified for the text/html document type in the `style.uni` file.

The default `style.uni` file automatically invokes the “flt_meta” content filter with the zone filter. Here is an example of a short `style.uni` file that can filter HTML documents with META tags (the `type:` statement here also appears in the default `style.uni` file):

```
$control: 1
 types:
 {
   autorec: "flt_rec"
   autorec: "flt_kv -recognize"
   type: text/html
     /charset        = guess
     /def-charset    = 1252
     /content-filter = "zone -html -nocharmap"
     /content-filter = "flt_meta"
   # if we get anything else, just skip it
   default:
     /action = skip
 }
 $$
```

The “flt_meta” filter watches markup tokens in the document stream. When the filter encounters a <META> tag, it produces a field token based on the tag and then the field token is stored as a field in the collection. In the collection’s document table, the field name is the name of the META tag’s `name` attribute, and the field value is the value of the `content` attribute in the META tag.

A sample <META> tag in HTML is shown here:

```
<META name="Abstract" content="This is a long document"
```

When filtering the HTML above, the “flt_meta” filter produces a field token of this form:

```
ABSTRACT: This is a long document
```

In the default `style.sfl` file, the field name “Abstract” is populated with the value “This is a long document”. A field definition that corresponds to the META tag’s name attribute must appear in the `style.ufl` or `style.sfl` in order for the field to be populated by the filter. In the example above, the field named “Abstract” is aliased to the “Snippet” field in the default `style.sfl` file so you would not need to add a field definition.

Defining Zones for Virtual Documents

The `style.dft` file is used to compose a virtual document “on the fly” that is made up of the body of the real document, plus the text of various fields that were previously extracted from the real document. This virtual document does not actually exist on disk, but is composed every time you view or index the real document. The virtual document allows the user to view all the relevant fields without needing to see all the fields.

You can define zones in this virtual document with the `zone-begin` and `zone-end` keywords in the `style.dft` file. Each keyword takes one argument, which is the name of the zone to begin and end.

Here is an example of a `style.dft` file that composes a virtual document containing zones.

```
$control: 1
dft:
{
  # this begins the "headers" zone, which contains all the
  # headers of this virtual document
  zone-begin: headers

  constant: "Title: "
  field: TITLE
  constant: "\nAuthor: "
  field: AUTHOR
  constant: "\nDate: "
  field: DATE

  # This ends the "headers" zone
  zone-end: headers

  # Now comes the actual text of the document as it is read
  # from the gateway. The engine knows to recognize the special
  # "DOC" field and read it from the gateway rather than
  # from the collection.
  #
  # Note that the document is filtered using the zone filter,
  # and the given style.zon instructs the zone filter which
  # zones to extract.
  #
  # Also note that the entire text of the document resides in a
  # zone called "body", separate from the headers zone above.
```

```
    zone-begin:  body

    field: DOC
      /filter="zone"

    zone-end:    body
  }
  $$
```

With this example `style.dft` file and its associated `style.zon` file, you can search for text that matches in the headers zone, or only in the body zone.

A shorthand notation exists for the `zone-begin` and `zone-end` combination. You can use instead the `/zone` construct. For example, as an alternative to the following:

```
  zone-begin:  zname
    field: fname
  zone-end:    zname
```

you could substitute the following:

```
  field:  fname
    /zone = zname
```

Hidden Elements in Zones

Hidden elements allow you to add text to the virtual document that gets indexed but cannot be viewed. This provides a way to add document fields to the full-text index for the document, allowing them to be searched faster than with standard field search, but preventing them from being viewed as part of the document. If the fields are enclosed in “hidden” zones, the fields can be searched using standard zone search syntax.

Hidden elements must be placed after all of the visible elements in the virtual document, as defined in the `style.dft` file.

For complete information about the zone filter and the special “noextract” and “noindex” zones, see “Special Noindex and Noextract Zones” on page 216

Entries in the style.dft File

Hidden elements are defined in the `style.dft` file using the `/hidden` modifier. Here's an example:

```
$control: 1
  dft:
  {
    field: DOC
      /filter="universal"

    zone-begin: NOEXTRACT
      /hidden=yes
    field: Title
      /zone=Title
    field: Keywords
      /zone=Keywords
    zone-end: NOEXTRACT
  }
```

Using the `style.dft` file above, the Verity engine adds a `Title` zone and a `Keywords` zone to the end of the virtual document. The zones will be indexed but not included in the viewing stream. If the `Keywords` field is generated by the META tag filter, it will be inserted and indexed in the Keywords zone.

The zone-begin and zone-end keywords define the boundaries for the hidden zone. The `noextract` label tells the summarizer to disregard the text of these hidden fields for feature extraction and summarization in the indexing stream. A `noextract` zone is a special zone type. In the example above, the noextract type with the /hidden attribute is applied to the `Title` and `Keywords` zones. In general, if the `/hidden` attribute is specified for a zone, all elements contained within the zone are also hidden.

Paragraph breaks are automatically inserted between hidden fields so that `PHRASE` queries don't hit across document-field and field-field boundaries (though `NEAR` queries might still hit).

Searching over Hidden Zones

Searching over hidden zones is like searching over regular zones. When the above `style.dft` file is implemented, you can write queries using the <IN> operator, like this:

`(`*`query`*`, `*`this`*`, `*`zone`*`) <IN> Keywords`

Hidden elements affect some of the Verity search engine's standard behavior in these ways:

- You don't see highlights for hits in the hidden text.
- The search engine considers the hidden text to be a real part of the document, so full text searches may find hits on words in the hidden text, which can't be seen by the user.

For information about searching over zones using the <IN> operator, refer to the *Verity K2 Query Language Guide*.

Special Noindex and Noextract Zones

The special noindex and noextract zones are optional. The noindex zone is used to mark text that will not be indexed. The noextract zone is used to mark text that will not be processed during feature extraction (for clustering and Query-By-Example) and summarization.

Noindex Zones

The `noindex` zone is a special zone whose contents do not get indexed. When the contents are not indexed, the Verity engine won't find hits on the text marked by the `noindex` zone.

To mark a zone as a `noindex` zone, you need to specify the `/zone=noindex` modifier in the `style.dft` file. An example `style.dft` file with `noindex` zones defined is shown here.

```
$control: 1
     dft:
     {
           constant: "Title: "
                   /zone=noindex
           field: TITLE
                   /zone=noindex
           constant: "\n"
                   /zone=noindex
           field: DOC
                   /filter=zone
```

```
          }
```

In the preceding example, the `constant`, `field`, and `constant` keywords together go into a zone called `noindex`. This is a special zone for the indexer. If the indexer sees this zone, it continues counting words inside that zone as it normally does, but it doesn't put those words into the full word index. Also, it doesn't store the boundaries of the `noindex` zone in the zone index either.

Here's another example with a different syntax, but equivalent meaning:

```
$control: 1
     dft:
     {
           zone-start: noindex
           constant: "Title: "
           field: TITLE
           constant: "\n"
           zone-end: noindex
           field: DOC
                   /filter=zone
     }
```

An added feature that falls out of this is that you can specify noindex zones inside your SGML documents if you like:

```
This is normal text. More normal text.
<noindex>
   This text won't be indexed because the zone filter will spit
   out a "zone-start noindex" token when it sees the above
   noindex tag. We can put weird words like "onomatopoeia" here
   and they won't show up in the full-word index.
</noindex>
   These words are outside the noindex zone again, so they will
   show up in the index again, i.e. Ya gotta be careful what
   you write out here!
```

Now, when you do a search for “onomatopoeia”, you will not find it.

Noextract Zones

The `noextract` zone is a special zone whose contents are not processed during feature extraction. Using this zone, you have the ability to selectively exclude sections of a document from being considered for feature extraction (for clustering/

Query-By-Example) and summarization. The summarization/feature extraction component recognizes the special zone token called NOEXTRACT. Anything between the start and end of a noextract zone is ignored by the feature extractor.

The use is analogous to the use of the special noindex zone, which is described in the previous section. Like a noindex zone, a noextract zone can be inserted either with the style.dft mechanism, with NOEXTRACT tags in SGML documents, or manually if you are using a custom gateway.

If you are developing a custom gateway using the Verity Gateway Developer Kit, you simply need to insert a zone token named noextract before and after the text to be ignored (with the start and end flags set appropriately).

The noextract zone is not indexed as a zone by the indexer, though the text within the zone is indexed.

Hidden Elements in NoExtract Zones

Hidden elements in noextract zones allow you to add text to the virtual document that gets indexed but cannot be viewed. This provides a way to add document fields to the full-text index for the document, allowing them to be searched faster than with standard field search, but preventing them from being viewed as part of the document. If the fields are enclosed in “hidden” zones, the fields can be searched using standard zone search syntax.

Hidden elements must be placed after all of the visible elements in the virtual document, as defined in the style.dft. For information implementing hidden elements in zones, see the previous section, “Hidden Elements in Zones,”

Searching in Zones

This section describes how users can search in zones when the zone filter is implemented.

Zones can be searched in two ways:

- Using the IN operator of the query language
- Using a custom query parser

Each of these methods is described in this section.

Using the Query Language IN Operator

Use the `IN` operator to search within zones.

The syntax of the `IN` operator is:

```
(query) <IN> zone
```

or

```
(query) <IN> (zone1, zone2, ...)
```

`query` represents any query expression. To preclude ambiguity, the query expression must be placed within parentheses. The *`zone`* variables represent the zone names. The zone name supplied must match the zone names defined in your collections. If more than one zone is to be searched, they must appear in a comma-separated list with parentheses surrounding them.

IN Operator Examples

The following examples illustrate the proper use of the IN operator.

To search in the zone named `summary` using the topic named `safety`, use the following query expression:

```
(safety) <IN> summary
```

To search in two zones, `summary` and `title`, using the topic named `safety`, use the following query expression syntax:

```
(safety) <IN> (summary, title)
```

If the query expression contains a comma, enclose the query expression in parentheses. Thus, to search in the zone `summary` using the query “safety, environmental regulation,” use the following query expression:

```
(safety, environmental regulation) <IN> summary
```

To search using the previous query in the `summary` and `title` zones, use the following query expression:

```
(safety, environmental regulation) <IN> (summary, title)
```

To specify searching a zone nested within a zone, nest the `IN` operator, as shown in the following example:

```
((happiness <AND> health) <IN> subsection) <IN> section
```

If you specify a query expression with the IN operator, and no zones have been defined for your collections, the search yields no results. No documents will be retrieved when zones have not been defined.

Using a Custom Query Parser

If you are using a custom query parser, you can set a field in the VdkQParserNewArgRec called defaultZone. This has the same effect as applying <IN> zone to the end of the custom query you define. This field should point to a string that is the zone specification for the search. You can leave this field NULL if you do not want to restrict the search to a particular zone.

You might use the defaultZone field when defining a form search, and restrict each field in the form to search within one particular zone:

```
To:        ____________    (search for this in the To zone)
Subject:   ____________    (search for this in the Subject zone)
From:      ____________    (search for this in the From zone)
```

Searching Multiple Zone Occurrences

When you create a zone, it may occur only once in the document (as in the case of the <title> tag extracted as a zone), or it may occur multiple times in a document (as in an <h2> tag extracted as a zone). When a zone occurs more than once in a document, you can search for words that appear together in the same occurrence of the zone, or you can search for words without respect to whether or not they appear in the same occurrence of the zone.

To search for two words appearing together in the same occurrence of the zone, use the standard syntax as described above, for example:

```
(health <AND> safety) <IN> h2
```

To search for the two words without regard to whether they appear in the same occurrence of the zone, use the following syntax:

```
(health <IN> h2) <AND> (safety <IN> h2)
```

Default style.zon File

Listing 7-2 shows the contents of the default `style.zon` file shipped with K2. This style file re-implements the default behavior that K2 5.5 exhibited when no `style.zon` file was used, except that it now honors no-index and no-extract zones for HTML files.

Listing 7-2 Default style.zon file

```
#
# style.zon - zone filter configuration
#
# This file is used to configure the behavior of the zone filter.
# macro definitions similar to those allowed by the C preprocessor.
# Note that style.zon may be removed, causing the zone filter to default
# to legacy behavior.
# Refer to the "Defining Document Zones" chapter in the
# Verity Collection Reference for more information.

#
# Generally, this file contains a zonespec (with possible modifiers) containing
# element, header and entity parameters.
#
# element: <zone name>
# [/modifiers]
#
# element parameters specify which content tags should be considered zones.
# The element zone name may be a wildcard ("*"), in which case any
# applicable modifiers (except /ignore) are inherited as default values for
# subsequently-named element parameters.  Modifiers associated with the
# last-appearing element parameter (including /ignore) are also retained and
# used for any elements enountered that have not been explicitly listed in
# this file.
#
# Note that /noindex and /noextract element modifiers have no effect for
# elements with the /ignore modifier.
#
#
# header: <header name>
# [/<modifiers>]
#
# header parameters specify which headers should be considered zones.
# The header name may be a wildcard ("*"), and header wildcards behave
# as element wildcards do.
#
#
# entity: <entity string> <equivalent string>
# [/<modifiers>]
#
# entity parameters specify the translation of entities.
# If no entity parameters are specified, default tables for the appropriate
# charset and document type will be used.
#

$control: 1

$ifdef ZONEMODE_SGML
# This section is used when the zone filter is launched without
# a -html, -email, -news, -traffic or -usermode arguments.
zonespec:
/ignoreattributes = no
{
    element: *
}

$elifdef ZONEMODE_HTML
# This section is used when the zone filter is launched with
# the -html argument.  Note that if no zonespec /html modifier is used,
# the style.zon file will be ignored and hard-coded default settings
# used.
zonespec:
/html
/ignoreattributes = no
{
    # Ignore anything not explicitly listed
    element: *
        /ignore = yes

    # Allow use of noindex/noextract tags
    # These tags are not HTML tags, but are useful for specifying content
    # that should not be indexed or should not have feature extraction.
    element: noindex
    element: noextract

    element: applet
        /ignore = yes
    element: area
        /ignore = yes
    element: basefont
        /ignore = yes
    element: bdo
        /ignore = yes
    element: br
        /ignore = yes
    element: button
        /ignore = yes
    element: center
        /ignore = yes
    element: col
        /ignore = yes
    element: colgroup
        /ignore = yes
    element: dd
        /ignore = yes
    element: del
        /ignore = yes
    element: dir
        /ignore = yes
    element: div
        /ignore = yes
    element: dl
        /ignore = yes
    element: dt
        /ignore = yes
    element: fieldset
        /ignore = yes
    element: fig
        /ignore = yes
    element: frameset
        /ignore = yes
    element: hr
        /ignore = yes
    element: input
        /ignore = yes
    element: ins
        /ignore = yes
    element: isindex
        /ignore = yes
    element: kbd
        /ignore = yes
    element: label
        /ignore = yes
    element: legend
        /ignore = yes
    element: li
        /ignore = yes
    element: listing
        /ignore = yes
    element: map
        /ignore = yes
    element: math
        /ignore = yes
    element: menu
        /ignore = yes
    element: meta
        /ignore = yes
    element: nextid
        /ignore = yes
    element: noframes
        /ignore = yes
    element: noscript
        /ignore = yes
    element: ol
        /ignore = yes
    element: optgroup
```

```
    /ignore = yes
element: option
    /ignore = yes
element: p
    /ignore = yes
element: param
    /ignore = yes
element: plaintext
    /ignore = yes
element: pre
    /ignore = yes
element: s
    /ignore = yes
element: select
    /ignore = yes
element: small
    /ignore = yes
element: span
    /ignore = yes
element: strike
    /ignore = yes
element: style
    /ignore = yes
element: sub
    /ignore = yes
element: sup
    /ignore = yes
element: tab
    /ignore = yes
element: table
    /ignore = yes
element: tbody
    /ignore = yes
element: tfoot
    /ignore = yes
element: thead
    /ignore = yes
element: td
    /ignore = yes
element: th
    /ignore = yes
element: tr
    /ignore = yes
element: tt
    /ignore = yes
element: ul
    /ignore = yes
element: var
    /ignore = yes
element: xmp
    /ignore = yes
element: a
```

```
    element: abbr
    element: abbrev
    element: acronym
    element: au
    element: address
    element: b
        /nonwordbreaking = yes
    element: banner
    element: base
    element: big
        /nonwordbreaking = yes
    element: blockquote
    element: body
    element: caption
    element: cite
    element: code
    element: dfn
    element: em
        /nonwordbreaking = yes
    element: fn
    element: font
        /nonwordbreaking = yes
    element: form
    element: frame
    element: h1
    element: h2
    element: h3
    element: h4
    element: h5
    element: h6
    element: head
    element: html
    element: i
        /nonwordbreaking = yes
    element: iframe
    element: img
    element: lang
    element: link
    element: note
    element: object
    element: person
    element: q
    element: samp
    element: script
    element: strong
        /nonwordbreaking = yes
    element: textarea
    element: title
        /field = yes
    element: u
        /nonwordbreaking = yes
}
```

```
$elifdef ZONEMODE_EMAIL
# This section is used when the zone filter is launched with
# the -email argument.  Note that if no zonespec /email modifer is used,
# the style.zon file will be ignored and hard-coded default settings
# used.
zonespec:
/email
{
    header: To
        /field = yes
    header: From
        /field = yes
    header: Subject
        /field = yes
    header: Date
        /field = yes
    header: Cc
        /field = yes
    header: Bcc
        /field = yes
    header: Organization
    header: Sent
        /field = yes
    header: Priority
        /field = yes
    header: Importance
        /field = yes
}

$elifdef ZONEMODE_NEWS
# This section is used when the zone filter is launched with
# the -news argument.  Note that if no zonespec /news modifer is used,
# the style.zon file will be ignored and hard-coded default settings
# used.
zonespec:
/news
{
    header: From
        /field = yes
    header: Subject
        /field = yes
    header: Date
        /field = yes
    header: Keywords
        /field = yes
    header: Organization
    header: Summary
    header: Newsgroups
        /field = yes
    header: References
        /field = yes
}

$elifdef ZONEMODE_TRAFFIC
# This section is used when the zone filter is launched with
# the -traffic argument.  Note that if no zonespec /traffic modifer is used,
# the style.zon file will be ignored and hard-coded default settings
# used.
zonespec:
/traffic
{
    header: *
}

$else
# This default section is typically not invoked.
# Note that a "-usermode <ModeName>" argument may be passed to the zone
# filter, and will cause $ZONEMODE_USER and $ZONEMODE="ModeName" to be
# defined.  This can be used to create custom style.zon configurations for
# handling various document types.
zonespec:
/ignoreattributes = no
{
    element: *
}
$endif
$$
```

8

Tuning Collections

This chapter discusses how to customize style files to give your collections greater capabilities, or to refine collection content to better meet your needs.

The following topics are covered:

- Style Files and Index Tuning
- Indexing Collection Fields (style.ufl)
- Adding Extra Collection Capabilities (style.prm)
- Using Custom Zones to Improve Relevance (style.tkm)
- Providing Passwords for Document Access (style.pw)
- Defining Indexing Stop Words (style.stp)
- Defining Indexing Go Words (style.go)
- Defining Feature-Extraction Stop Words (style.fxs)
- Customizing 7-Bit Tokenization (style.lex)

Style Files and Index Tuning

The Verity engine indexes documents using configuration options specified in style files. A directory of default style files is included with each Verity product.

The configuration options specified in the style files listed below affect how word indexes and feature vectors (for clustering and summarization) are generated during the indexing process.

Style File Name	Function
`style.lex`	Applies only to 7-bit character sets such as ASCII. Specifies that nonalphanumeric characters be used as search criteria. The specified characters can be interpreted as legal characters so that words containing the specified nonalphanumeric, like OS/2, will appear as index entries.
`style.stp`	Specifies an excluded word list. This list excludes selected words from the collection word index—freeing the index of unwanted words.
`style.go`	Specifies an included word list. This list includes only selected words in the collection word index—limiting the word index to a narrow vocabulary.
`style.ufl`	Specifies collection fields for which field indexes will be built. By implementing an /indexed or /minmax field index, the Verity engine indexes collection field values so that a search can search the field values more efficiently.
`style.prm`	Specifies additional data to be stored in the index, including: SOUNDEX data, assist data, and highlight data
`style.fxs`	Specifies words to exclude from feature vectors so the words do not appear in document summaries and clusters.
`style.tkm`	Specifies document zones and fields to be created based on document formatting and location information. Can be used to improve relevance ranking or for other purposes.

To tune an existing collection, you edit the style files in the `*collection_name*/style` (see "A Collection's Internal Style Set" on page 58) and then re-index.

To create a new collection with the desired tuning parameters, you create a new style set based on one of the default style sets (see "Standard and Default Style Sets" on page 54), change its parameters, then use it to create a collection.

Note Never edit a collection's `style.ddd` file.

The following sections explain the style-file options available for index tuning.

Indexing Collection Fields (style.ufl)

Using the `style.ufl` file, you can select to index collection field values for certain field types so that information agents can perform field searches more quickly. Field indexes are distinct from word indexes, and they are entirely optional.

Two types of indexed fields can be implemented: indexed and minmax. These field types are discussed in this section. Indexing field values increases the time it takes to index documents in general, but the payoff is in improved search speed.

The `style.ufl` file is described in "style.ufl Syntax" on page 157. The default `style.ufl` file is shown in Listing 5-3 on page 157.

Indexed Field Type

You can opt to index fields defined in the document collection by specifying the `/indexed = yes` modifier for the fields to be indexed in the relevant `style.ufl` file.

Verity recommends that you use indexed fields as follows:

- When you anticipate queries will involve field searches over one or more collection fields
- When you anticipate users will want to sort query results by one or more collection fields

An indexed field index can be case-sensitive. If the `/case-sensitive` entry is specified in addition to the `/indexed` entry for a field keyword in a `style.ufl` file a case-sensitive index of field values is created.

Creating case-sensitive indexes for fields is valuable when a query contains case-sensitive search criteria. For example, if a retrieval contains a CASE modifier, a case-sensitive field index could speed the retrieval.

If case-sensitive queries are not issued, and case-sensitive field indexes are created, then maximum efficiency is not achieved, and the user will not experience improved search speed as the result of indexing fields.

Each indexed field must be in its own data table. A data table containing an indexed field must not contain any other fields. At index time, the engine issues an error message if this rule is violated:

```
Error E)-)448(Vdb): Indexed fields must be in their own
table (NAME)
```

where NAME is the name of the indexed field that is not defined in its own data table.

It is not an error to have more than one indexed varwidth field in the same data segment. You can define two indexed varwidth fields in two different data tables in the same data segment. This configuration can affect search and results display as described in "Defining Collection Fields" on page 145

Minmax Field Type

Defining a field as a minmax field is recommended for fixed-width and variable-width fields in the collection of documents when you anticipate queries will involve field searches over these fields. minmax fields greatly improve retrieval speed for field searches, especially when your document collection is very large. Only fixed-width and variable-width fields may be defined as minmax fields.

Defining minmax field indexes is very worthwhile if there are a specific range to the field's values, and if users want to perform field searches over the field regularly.

You can implement a fixed- or variable-width field as a minmax field in the `style.ufl` files to be used to build the collections. To define one of these types of collection fields as a minmax field, enter the `minmax=yes` modifier to the appropriate keyword, either `fixwidth` or `varwidth`, as shown in the example here.

```
varwidth: author  dd5
    /minmax = yes
```

A minmax field can store a maximum of 256 bytes. If the field value is longer, the Verity engine stores the first 256 characters in the minmax field index.

When minmax fields are defined, the Verity engine creates and maintains `worm` (*write once, read many*) field indexes containing all field values for all documents in a collection. During search processing, the search engine reads information in the field indexes, instead of reading through the actual document records.

If the information provided in the minmax `worm` fields is not sufficient for the search engine either to select all of the documents in the collection or to pass over all the documents, then the search engine must read the individual document records contained in the collection.

Adding Extra Collection Capabilities (style.prm)

Using the `style.prm` file, you can specify additional data that you want included in the collection indexes. Additional data in collection indexes support some search features, like clustering and summarization. The `style.prm` file can be used to specify index features such as stemming, case-insensitive word indexes (to support case-insensitive search), and SOUNDEX search.

See "Default style.prm File" on page 245 for an example of `style.prm`.

Specifying Instance Vector Encodings

Paragraph and sentence boundaries are detected at index time by whatever lexer or tokenizer is use. The boundary determination can be based on punctuation, indentation, or anything else the lexer implements. The built-in lexer supplied with Verity products is punctuation based.

If you modify the `style.prm` file, you can configure the internal, punctuation-based lexer for all document types except PDF. If you use a Verity Locale, the behavior of the tokenizer supplied with the locale cannot be modified.

PSW and WCT are two different instance vector encodings. As words are put into the word index, their positions are encoded using one of the two encodings. PSW stands for Paragraph-Sentence-Word encoding. WCT stands for Word-Count encoding.

WCT Encoding Issues

When WCT encoding is used, explicit sentence and paragraph position information is not stored in the word index. However, word count is incremented by one any time the indexer sees a Paragraph or Sentence token coming from the tokenizer. This behavior prevents phrases from spanning a sentence and/or paragraph boundary.

WCT encoding is implemented by default; the `style.prm` file included with Verity products in the default and sample style directories includes this entry:

```
$define IDX-CONFIG      "WCT"
```

If no sentence and/or paragraph boundaries are detected at indexing time (as with PDF documents), then phrase searches can return phrases that span sentence boundaries. For non-PDF documents, the Verity lexer determines sentence and paragraph boundaries, so phrase searches do not appear to span sentence boundaries.

If you want to allow phrase searches in all documents to return phrases that span sentence and paragraph boundaries, you can force that with the NOEOS option:

```
$define IDX-CONFIG      "WCT NOEOS"
```

PSW Encoding Issues

When PSW is used, the explicit paragraph and sentence position of each word instance is stored in the word index. The paragraph and sentence counters are incremented whenever the indexer receives a Paragraph or Sentence token from the tokenizer.

PSW encoding allows only 255 words in a sentence and only 255 sentences in a paragraph. If you index something that does not have sentence and/or paragraph boundaries, the indexer creates sentence and paragraph boundaries with these limits. The PDF tokenizer, for example, does not detect paragraph or sentence boundaries, meaning that the indexer creates arbitrary boundaries that do not correspond to the apparent punctuation in the documents.

When a boundary is met the indexer produces a message like this:

```
Warn E2-0526 (Document Index): Document 1 (report.pdf):
Sentence 1 in paragraph 0 has more than 255 words - splitting
sentence.
```

PSW encoding is activated by uncommenting the following line in the `style.prm` file:

```
$define IDX-CONFIG      "PSW Many"
```

If PSW encoding is used, `NEAR` and `NEAR/N` queries will not cross sentence or paragraph boundaries. In other words, a `NEAR` query returns documents in which search terms appear within *N* words in the same sentence and paragraph.

Note The `Many` option, used with either WCT or PSW encoding, improves results from VQL searches that use the `Many` modifier. However, it can enlarge the collection and slow search performance.

SENTENCE and PARAGRAPH Search Operators

When used in queries, the semantics of the `SENTENCE` and `PARAGRAPH` operators are the same regardless of whether the collection was built with PSW or WCT encoding.

For PSW collections, the operators use stored position information.

For WCT indexes, the sentence and paragraph boundaries are approximated using 15-word and 100-word rules. These word windows are applied dynamically at search time (for example, as `long` as the children of a `SENTENCE` operator are within 15 words of each other, the `SENTENCE` operator will succeed.) This means that `SENTENCE` and `PARAGRAPH` operators match documents if the search terms occur within a certain distance of each other, whether or not the terms occur in the same actual paragraph or sentence.

Note If you need the SENTENCE operator to be accurate, meaning results contain documents that only have words in the same sentence, then use PSW encoding. Using WCT encoding means you might have results where words are not always in the same sentence.

Enabling Storage of Nouns and Noun Phrases

For thematic mapping, your collection needs to include a list of all the nouns and noun phrases in a collection and their term frequencies in each document. To enable indexing and storage of nouns and noun phrases, do this:

1. Ensure that noun and noun-phrase extraction is not disabled for your locale. See the discussion of the `-nonnp` option in the locale-configuration chapter of the *Verity Locale Configuration Guide*.
2. Un-comment the following `$define` statements in your collection's `style.prm` file:

   ```
   #$define NOUN-IDXOPTS   ""
   #$define NPHR-IDXOPTS   ""
   ```

   The uncommented lines should appear as:

   ```
   $define NOUN-IDXOPTS   ""
   $define NPHR-IDXOPTS   ""
   ```

The statements following the `$define` will ensure that if these options are active and no Casedex is active (the default), nouns and noun phrases will be indexed in upper case. If these options are active and Casedex is active, nouns and noun phrases will be indexed using case-sensitivity.

Enabling Document Features

To use clustering, VDK summarization, recommendation, and fast query-by-example, the generation of document feature vectors at indexing time must be enabled. This requires that an un-commented `$define` directive with the `DOC-FEATURES` parameter must exist in the collection's `style.prm` file. The `DOC-FEATURES` parameter can have any of three values:

`$define DOC-FEATURES "TF"`	Generate feature vectors from all word types.
`$define DOC-FEATURES "NP"`	Generate feature vectors from noun phrases only.
`$define DOC-FEATURES "NNP"`	Generate feature vectors from nouns and noun phrases.

By default, the `"NNP"` definition is un-commented and the others are commented out, meaning that feature vectors based on nouns and noun phrases are enabled. You can change the commenting to change the kinds of features vectors, if any, that are to be generated.

If your collection's `style.prm` specifies either `"NP"` or `"NNP"`, you must also ensure that noun and noun-phrase extraction is not disabled for your locale or language. See the discussion of the `-nonnp` option in the locale-configuration chapter of the *Verity Locale Configuration Guide*.

`TF`, `NP`, and `NNP` can take an optional argument, `MaxFtrs` *n*, that is only rarely used, where *n* specifies the number of features to store per document in the collection. An example of the complete syntax is

```
$define DOC-FEATURES "TF MaxFtrs n"
```

If `DOC-FEATURES` is defined, the `VdkFeatures` field is automatically included in the document-table schema, to hold the generated feature vectors.

Configuring Document Summaries

By presenting a short summary for each document in a results list, an application can help users to quickly assess the relevance of the documents without retrieving the full text of each document. Verity supports the creation of three types of document summaries:

- Static Summaries
- Dynamic Summaries
- Passage-Based Summaries

Dynamic summaries are generated at search time through the K2 client API, as described in the *K2 Client C Programming Guide*, or through the VDK API, as described in the *Verity Developer's Kit Programming Reference*. Static summaries and passage-based summaries are enabled through settings in `style.prm` and other style files, as described here.

Static Summaries

To automatically generate and store document summaries at indexing time, include the `$define` directive with the `DOC-SUMMARIES` parameter in the `style.prm` file, as follows:

```
$define DOC-SUMMARIES "type [Zone]"
```

where *`type`* specifies the type of summary to generate: XS, LS or LB. The type is required and must appear first in the string ahead of any of the optional parameters.

Summary Types

Type	Description
`XS`	Extract the "best" sentences from the document
`LS`	Use the first sentences from the document
`LB`	Use the first bytes of text from the document (with white space compressed)

Summary Type Parameters

Along with the summary type, you can also specify optional parameters as described in the following table.

Parameter	Description
`MaxBytes` *`n`*	*`n`* specifies the maximum size of a summary in bytes. Summaries longer than *`n`* are truncated with an ellipse (...). This parameter is supported by all three summary types The default value is 400.
`MaxSents` *`n`*	*`n`* specifies the maximum number of sentences in a summary. This parameter is supported by the XS and LS summary types. The default value is 2.
`TruncSent` *`n`*	*`n`* specifies the maximum length of any sentence. Sentences longer than *`n`* are truncated with an ellipse (...). This parameter is supported by the XS and LS summary types. The default value is 400.

Zone Argument

The optional Zone argument adds the summary to the end of a document as a zone, allowing you to perform zone searches on the summary.

Examples of Summary Definitions

Generate summaries comprised of the best 3 sentences from each document, or a maximum of 500 bytes:

```
$define DOC-SUMMARIES "XS MaxBytes 500 MaxSents 3"
```

Generate summaries comprised of the first 400 bytes from each document, compressing white space, while storing the summary as a zone:

```
$define DOC-SUMMARIES "LB MaxBytes 400 Zone"
```

If a `$define` directive with the `DOC-SUMMARIES` parameter is defined, the `VdkSummary` field is automatically included in the document table schema to contain the generated summaries.

HTML Code in Summaries

If an indexed document contains HTML code samples, and if code from those samples is included in the document's stored summary, the summary may display incorrectly in an application's HTML search-results page. This problem occurs because the browser attempts to interpret and display the summary as actual HTML instead of sample code.

Stored static summaries can contain only plain-text data. The search application is responsible for any necessary pre-processing on the summary (for example, escaping characters such as "<" in HTML sample code) so that it displays correctly for the user.

Passage-Based Summaries

A passage-based summary consists of one or more passages (sentences or phrases) from the document, each of which contains an instance, optionally highlighted, of the search term that was used to locate the document. This type of summarization is also called *keywords in context*, because the keyword (the search term) is displayed in its context within the document.

A passage-based summary is different from other kinds of summaries in that a single document can yield a different summary for each different search term (query) that is applied to the document.

The following is an example of a passage-based summary for a document returned from a stemmed search for **Verity collection**. This summary includes three passages:

> ...This chapter introduces the basics about **Verity collections**: ...how to build and configure **Verity collections** for your search application. ...Any **Verity**

> **collection**-building application can read and add documents to any collection to which it has valid access. ...

To enable or disable generation of passage-based summaries, you must make the appropriate settings in the `style.prm` and `style.ddd` files in your collection's style set.

Modifying style.prm

In `style.prm`, locate the following section:

```
#----------------------------------------------------------------
# Passage-based summarization is enabled by uncommenting the
# DOC-PBSUMMARIES line below. This stores tokenized and
# compressed text version of documents in the document table.
# The tokenized texts can then be used in the passage-based
# summarization, which delivers the summary with search term
# highlighted.

# The example below stores up to 8K text for each document.
$define DOC-PBSUMMARIES "MaxBytes 8192"
```

If the `$define DOC-PBSUMMARIES` line is uncommented, the first `MaxBytes` bytes of each indexed document are stored in the internal collection field `VDKPBSUMMARYDATA`, for use in construction of passage-based summaries. By default, `MaxBytes` is 8192, but you can change the value to extract either more or less text from documents during indexing. (The maximum value for `MaxBytes` is 32K.)

Note The information stored in `VDKPBSUMMARYDATA` is tokenized and compressed. It is not directly readable.

Note that the `$define DOC-PBSUMMARIES` line is uncommented by default. To disable passage-based summaries, comment out the line.

Verifying style.ddd

In `style.ddd`, make sure the following section exists:

```
$ifdef DOC-PBSUMMARIES
  # Optional tokenized and compressed texts per document for
  # passage-based summarization
  data-table:_dm
  {
    varwidth: VDKPBSUMMARYDATA _dn
      /_implied_size
      /alias = dc:PBSummaryData
     /alias = vdk:VDKPBSUMMARYDATA
  }
$endif
```

This section adds the field VDKPBSUMMARYDATA to the collection (if DOC-PBSUMMARIES is defined in style.prm).

Creating the Summaries

If you have made the appropriate settings in style.prm and style.ddd to enable the gathering of the data, your search application can then create the passage-based summaries. At search time, the application makes function calls as described in the document-access chapter of the *Verity Client C Programming Guide* or the clustering and summarization chapter of the *Verity Developer's Kit Programming Reference*.

Setting Index Options

In your collection's style.prm file, you can set options that control the existence and characteristics of the following indexes:

index	Description	style.prm parameter	Available options
Word index	Lists all significant words and their positions in each collection document.	WORD-IDXOPTS	Stemdex Casedex Soundex Location*N* Qualify*N*
Zone index	Lists names of all defined document zones in the collection, indirectly indexes the zones' content.	ZONE-IDXOPTS	Dewey
Zone-attribute index	Lists the values of all zone attributes in the collection.	ATTR-IDXOPTS	Casedex Numdex Datedex Xdatedex

index	Description	style.prm parameter	Available options

Zone-content index	Lists the content of all defined document zones in the collection (to support range search on numbers, dates, and text).	`ZONE-CONTENT-IDXOPTS`	`Casedex`
			`Numdex`
			`Datedex`
			`Xdatedex`

Noun index	Indexes the locations of all nouns in each collection document.	`NOUN-IDXOPTS`	(none)
Noun-phrase index	Indexes the locations of all noun phrases in each collection document.	`NPHR-IDXOPTS`	(none)

The options have the following meanings for an index:

- `Stemdex`. The stem of each indexed word is also indexed.
- `Casedex`. Each case variant of a word is stored separately in the index.
- `Soundex`. A phonetic representation of each word is stored in the index.
- `Dewey`. XML structured search is supported on the index.
- `Numdex`. Numeric-range search is enabled on the index.
- `Datedex`. Date-range search (in Verity date format) is enabled on the index.
- `Xdatedex`. Date-range search (in Verity xdate format) is enabled on the index.
- `Location`*`N`*. Store *N* bytes of highlight-location data with each word instance.
- `Qualify`*`N`*. Store *N* bytes of qualifying data with each word instance.

The following sections describe how to use these options to set index characteristics.

Case-Insensitive Search

Case-sensitive word indexes are built by default, so case-sensitive searching occurs automatically. With case-sensitive word indexes, if a user enters a mixed-case query, the engine finds case-sensitive matches only.

If you want your collection to support only case-insensitive searches, you need to build case-insensitive word indexes by making an edit in the default `style.prm` file. To do this, remove the `Casedex` option from this `$define` directive:

```
$define WORD-IDXOPTS    "Stemdex Casedex"
```

so that the edited directive looks like this:

```
$define WORD-IDXOPTS    "Stemdex"
```

The zone-attribute index and the zone-content index also support case-sensitive search by default. To change them to case-insensitive, make a similar change to these `$define` directives:

```
$define ATTR-IDXOPTS            "Casedex Numdex Datedex"

$define ZONE-CONTENT-IDXOPTS    "Casedex Numdex Datedex"
```

so that the edited directives look like this:

```
$define ATTR-IDXOPTS            "Numdex Datedex"

$define ZONE-CONTENT-IDXOPTS    "Numdex Datedex"
```

Stemming

Word stems are by default indexed, so that a search for a specific term (such as “stems”) will find all instances of all variations of its stem (“stem”, stemmed”, “stemming” and so on).

If you do not want your collection to support stemmed searches, you can ensure that word stems are not indexed by editing the default `style.prm` file. To do this, remove the `Stemdex` option from this `$define` directive:

```
$define WORD-IDXOPTS    "Stemdex Casedex"
```

so that the edited directive looks like this:

```
$define WORD-IDXOPTS    "Casedex"
```

Soundex Search

By default, Soundex information is not indexed for a collection, and thus the VQL operator `SOUNDEX` will not work in a search. To specify a Soundex index to be built, you need to include the `Soundex` option with the `WORD-IDXOPTS` parameter. The following directive adds Soundex indexing to the default word index:

```
$define WORD-IDXOPTS    "Stemdex Casedex Soundex"
```

XML Structure Search

By default, VQL searches of XML structures (tag hierarchies) are supported for zones. The search syntax supports a subset of XPath syntax. For example, the following query returns (bibliographic) documents in which the first listed book is about UNIX:

```
VQL: unix <in> //book[1]
```

To remove the XML structure search capability, remove the `Dewey` option from this `$define` directive:

```
$define ZONE-IDXOPTS    "Dewey"
```

so that the edited directive looks like this:

```
$define ZONE-IDXOPTS    ""
```

XML Range Search

By default, the zone-content index holds dates and numeric values in a manner that supports VQL searches of XML documents for value ranges. For example, the following query returns books about UNIX whose price (the value of the `price` attribute) is less than $60.00:

```
VQL: unix <in> book <when> <zone><numeric>price < 60
```

The following query returns books about UNIX that were published since the beginning of 1999:

```
unix <in> book <when> <date>published > "1/1/1999"
```

To remove the ability to search for numeric or date ranges (or both), remove the Numdex or `Datedex` option from this `$define` directive:

```
$define ZONE-CONTENT-IDXOPTS    "Casedex Numdex Datedex"
```

so that the edited directive looks like this (if you remove both):

```
$define ZONE-CONTENT-IDXOPTS    "Casedex"
```

By default, dates are stored and evaluated as standard Verity dates, which range from 1904 to 2037 AD (with 1-second precision). You can specify that dates be stored instead as Verity xdates, which range from 1000 to 3000 AD (with 1-minute precision). To switch to xdate format, replace the `Datedex` option with `Xdatedex` in the default directive, so the edited directive looks like this:

```
$define ZONE-CONTENT-IDXOPTS    "Casedex Numdex Xdatedex"
```

Note The zone-content index is not designed to hold large amounts of content. Any zone content string greater than 250 bytes long is truncated to 250 bytes. Therefore, XML range searches will not succeed on content beyond the initial 250 bytes in a zone.

Highlight Location Data

Application developers may want to build auxiliary highlight data to store information used to highlight words in retrieved documents. For example, you can store auxiliary highlight data to store information like a page number or a byte offset into the original file. To build the auxiliary highlight data into the collection index, use the `WORD-IDXOPTS` parameter.

To build auxiliary highlight data into the collection index, use the `Location`*`N`* option with the `WORD-IDXOPTS` parameter:

```
$define WORD-IDXOPTS    Location4
```

This statement reserves 4 bytes of auxiliary data for each word instance in the index. (You can reserve 1 to 4 bytes.)

Building auxiliary highlight data is considered an advanced feature for application developers using the Verity Developer Kit.

Note The default value stored for the highlight location data (when it is enabled) is the byte offset into the indexing stream to the document.

Qualify Instance Data

Application developers may want to build qualify instance data into the word index, to store auxiliary information about words in the collection. This data is used to support VQL *qualify instance queries*, in which only those instances of a word that match not only the query word itself but also the specified auxiliary data are returned. See the discussion of qualify instance queries in the *Verity Query Language and Topic Guide* for more information.

Using qualify instance data can allow application developers to implement specialized zone-like searches if, for example, words in a document's abstract are given different instance values than words in the body. It can also allow for faster searches of large fields, if words in such a field are identified with a specific instance value.

Implementing qualify instance data is a four-step process, described in the appendix on word qualification in the *Verity Developer's Kit Programming Reference*. Modifying `style.prm` as described here is the first step.

To build qualify instance data into the collection index, use the `Qualify`*`N`* option with the `WORD-IDXOPTS` parameter:

```
$define WORD-IDXOPTS    Qualify4
```

This statement reserves 4 bytes of auxiliary data for each word instance in the index. (You can reserve 1 to 4 bytes.)

Note Do not specify more bytes than you need. Reserving space for qualify instance data can significantly affect collection size.

style.prm File Syntax

The `style.prm` file consists of `$define` directives, several of which are commented out. You may uncomment `$define` directives to control collection index content. Also, you can edit or remove `$define` directives, as appropriate.

Note If you are changing the `style.prm` file for an existing collection, be sure to re-index the collection.

In the `style.prm` file, the `$define` directives may be entered in any order. Blank lines are ignored, and comments are introduced with the # character. The `$define` directive has this structure:

`$define` *`parameter`* `"`*`option string`*`"`

When specifying more than one option for a parameter, put the set of options within double quotes. For example:

```
"Stemdex Casedex"
```

If there is only a single option, it need not be quoted.

Note The StyleSet Editor includes a form (**Collection Parameters**) for editing parts of `style.prm`.

Default style.prm File

The default `style.prm` file for the File System gateway is shown in Listing 8-1.

Listing 8-1 Default style.prm file for file-system gateway

```
#
# style.prm - collection schema parameters
#
# This file is used to enable/disable index schema features through
# macro definitions similar to those allowed by the C preprocessor.
# This file is included in other style files using $include so
# that the selected features are propagated to the schemas of all
# tables in the index.  Refer to the style.prm discussion
# in the chapter "Tuning Collections" in the Verity Collection Reference
# for more information.

# ----------------------------------------------------------------
# The IDX-CONFIG parameter defines the storage format used to
# encode the word positions in the index.  WCT (Word Count) format
# is a compact format, storing the ordinal counting position of the
# word from the beginning of the document.  PSW (Paragraph, Sentence,
# Word) format takes approximately 15-20% more disk space, but
# stores semantically accurate paragraph and sentence boundaries.
# Optionally, Many may be specified with either WCT or PSW to
# improve the accuracy of the <MANY> operator at the expense of
# disk space and search performance.

# This example enables Word Count word position format (the default).
$defineIDX-CONFIG"WCT"



#This example enable Word Count word position format but ignore
#sentence tagging. The word position is bumped upon sentence tokens.
#However, the sentence breaks maybe incorrect, causing phrase op to fail
#to yield a hit. This option ignores sentence tagging during
#the indexing time for word position counting(i.e. word positions will not
#be bumped upon sentence breaks).
#$defineIDX-CONFIG"WCT NOEOS"


# This example turns on Paragraph/Sentence/Word word position format.
# It also enables the <MANY> operator accuracy improvement.
#$define IDX-CONFIG"PSW Many"

# ----------------------------------------------------------------
# The IDXOPTS parameters define which index options are applied to
# the various index token tables.  The following index options are
# supported for each: Stemdex enables an index by the stem of each
# word. Casedex stores all case variants of a word separately, so
# one can search for case sensitive terms such as "Jobs", "Apple",
# and "NeXT" more easily. Soundex stores phonetic representations
# of the word, using AT&T's standard soundex algorithm. Numdex
# enables numeric search on attributes. Datedex enables date search
# on attributes.  Xdatedex does the same for xdates; Datedex and
# Xdatedex are mutually exclusive.  Dewey enables structured
# search for zones.  The application may also store 1-4 bytes of
# application-specific data with each word instance, in the form of
# Location data and/or Qualify Instance data. These options are
# specified separately for each token table: word, zone,
# zone attribute and zone content.
```

```
$defineWORD-IDXOPTS"Stemdex Casedex"
$defineZONE-IDXOPTS"Dewey"
$defineATTR-IDXOPTS"Casedex Numdex Datedex"
$define ZONE-CONTENT-IDXOPTS "Casedex Numdex Datedex"
#$defineNOUN-IDXOPTS""
#$defineNPHR-IDXOPTS""
$ifdef NOUN-IDXOPTS
  $ifdef NPHR-IDXOPTS
    $define NNP-IDXOPTS
  $endif
$endif

# The following example shows how to associate 4 bytes of Location
# and Qualify data with each word instance.
#$defineWORD-IDXOPTS"Location4 Qualify4"

# The DEWEY-IDXOPTS setting below limits the maximum Dewey Ordered
# path length to 128 levels.
$defineDEWEY-IDXOPTS"MaxLevels 128"

# -----------------------------------------------------------------
# Clustering is enabled by uncommenting one of the DOC-FEATURES
# lines below. DOC-FEATURES stores a feature vector for each
# document in the Documents table. These features are used for
# Clustering results and fast Query-by-Example.  See the discussions
# on clustering in Verity Developer's Kit Programming Reference for
# more information.
# The maximum number of features can be controlled by appending
# "MaxFtrs n" to the DOC-FEATURES string. The default is 25.

# The example below creates the DOC-FEATURES from any of the words
# in the document.
#$define DOC-FEATURES    "TF"

# The example below creates the DOC-FEATURES entirely from Nouns
# and Noun Phrases.
$define DOC-FEATURES    "NNP"

# The example below creates the DOC-FEATURES entirely from Noun Phrases.
# $define DOC-FEATURES    "NP"

# -----------------------------------------------------------------
# Document Summarization is enabled by uncommenting one of
# the DOC-SUMMARIES lines below.  The summarization data is
# stored in the documents table so that it might easily be
# shown when displaying the results of a search.
# See the discussions on document summaries in the style.prm section
# of the chapter "Tuning Collections" in the Verity Collection Reference
# for more information.

# The example below stores the best three sentences of
# the document, but not more than 500 bytes.
```

```
$define DOC-SUMMARIES   "XS MaxSents 3 MaxBytes 500 Zone"

# The example below stores the first four sentences of
# the document, but not more than 500 bytes.
#$define DOC-SUMMARIES   "LS MaxSents 4 MaxBytes 500"

# The example below stores the first 150 bytes of
# the document, with white space compressed.
#$define DOC-SUMMARIES   "LB MaxBytes 150"

#----------------------------------------------------------------
# Passage-based summarization is enabled by uncommenting the
# DOC-PBSUMMARIES line below. This stores tokenized and
# compressed text version of documents in the document table.
# The tokenized texts can then be used in the passage-based
# summarization, which delivers the summary with search term
# highlighted.

# The example below stores up to 8K text for each document.
$define DOC-PBSUMMARIES "MaxBytes 8192"
```

Using Custom Zones to Improve Relevance (style.tkm)

The token-map segment is a part of the sequence of filters that processes incoming documents during indexing. The token-map segment performs three tasks that can be useful in assigning relevance or in classifying document information:

- It generates custom zones based on a document's text-formatting zones.
- It tokenizes the contents of the document's `VdkVgwKey` field or any custom field

You use the `style.tkm` file to configure this functionality.

Creating Custom Zones

The token-map segment's purpose is to allow applications to make use of the formatting information contained in a document's text-formatting zones (see "Generating Text-Formatting Zones" on page 118). The token-map segment operates on the text-formatting zones (and their attributes) in the document's content, producing a different set of custom zones as output. (The token-map segment does not change or remove any of the text-formatting zones or attributes.)

For example, the token-map segment could place all headings into a single zone that is considered important for searching purposes. As another example, all bold or italic text, or perhaps all table text, might be placed in a zone of particular importance for certain kinds of queries.

Here are some examples of the kinds of configuration options you can use to create custom zones based on document format:

- **Example 1**. Map any of a set of text-formatting zones to a custom zone:

  Criterion: Include any of the text-formatting zones `h1`, `h2`, or `h3` in custom zone `headings`.

  Document content:
  `<h1>ABC</h1>...<h2>DEF</h2>...<h4>GHI</h4>...`

  Output:
  `ABC` and `DEF` are mapped to zone `headings`.

- **Example 2**. All specified text-formatting zones must apply for the mapping to occur:

  Criterion: Include text that is both `bold` and `italics` in custom zone `emphasis`.

  Document content:
  `<bold>ABC<italics>DE</italics>FG</bold>HI`

  Output:
  `DE` is mapped to zone `emphasis`.

- **Example 3**. Certain attribute values must apply for the mapping to occur:

  Criterion: Include text where `font=Arial` and `size>24` in custom zone `headlines`.

  Document content:
  ```
  <font face="Arial" size=25>ABC</font>
  <font face="Arial" size=24>DEF</font>
  <font face="Palatino" size=25>GHI</font>
  ```

  Output:
  `ABC` is mapped to zone `headlines`.

You configure this capability with the `mapto` keyword in `style.tkm`.

Tokenizing Custom Fields

The token-map segment tokenizes a document's `VdkVgwKey` field (or other custom field) to extract words in it that also relate to the document's content. The token-map segment then writes those words into a zone at the end of the document. The result of this process is a zone that contains terms that are relevant to both the document's location and its content.

For example, the URL key

```
http>//www.acme.com/corporate/financials/2002_AnnualReport.pdf
```

might yield, when tokenized and compared to the contents of the PDF document, a zone containing the following terms:

```
corporate financials 2002 annual report
```

Note in particular that this tokenization extracts the separate terms `annual` and `report`, which would not be possible with just a tokenization of the URL.

You configure this capability with the `tokenizeto` keyword in `style.tkm`.

Note The default version of `style.tkm` is already configured to use the `tokenizeto` keyword to tokenize the `vdkvgwkey` field in a collection.

style.tkm File Syntax

Like all style files, `style.tkm` starts with a `control` directive. That directive is followed immediately by a `definitions` directive:

```
$control: 1
definitions:
{
```

The rest of the file consists of alias definitions followed by mapping rules followed by tokenization definitions.

Alias Definitions

Aliases are specifications of input-zone attribute values that can be used like input zone names in subsequent mapping rules. An `aliasto` definition has this form:

```
aliasto: aliasname
```

```
{
  inputzonename
    /attribute: type expression
    [/attribute: type expression...]
}
```

where

- *aliasname* is the name of the alias (such as `largeFont`).
- *inputzonename* is the name of the input text-formatting zone (such as FONT) to which the attribute that follows is to apply.
- *type* is is the type (`integer` or `string`) of the attribute value. The expression must evaluate to this type.
- *expression* is a specification of the attribute value (such as `size>24`). The available operators to use in *expression* are

  `>` (greater than)
  `<` (less than)
  `=` (equals)
  `!=` (not equals).

In a subsequent mapping rule that includes this alias, when the value of the attribute for this input zone matches all of the expressions, the contents of the input zone are added to the output zone specified in the rule.

All aliases must be defined before any mapping rules.

Mapping Rules

A mapping rule defines a custom (output) zone and specifies which format (input) zones and aliases are to be included in it. The rule can also specify which types of documents the rule applies to. A `mapto` definition has this form:

```
mapto: outputzonename
  [/mimetypeinclude mimetype [mimetype...]]
  [/mimetypeexclude mimetype [mimetype...]]
{
  any | all: inputzonename [inputzonename...]
  [any | all: inputzonename [inputzonename...]...]
}
```

where

- *`outputzonename`* is the name (for example, `relevance1`) of the custom zone being defined.
- *`mimetype`* is a MIME type definition (such as `application/x-ms-powerpoint`) that specifies a type of document to be either included or excluded when creating this zone. Multiple MIME types can appear in the line, separated by spaces.

  Both `mimetypeinclude` and `mimetypeexclude` are optional parameters. If only `mimetypeinclude` appears, all other document types are excluded. If only `mimetypeexclude` appears, all other document types are included. If neither parameter appears, all document types are included.

  The maximum number of MIME-type definitions you can list in a single `mimetypeinclude` or `mimetypeexclude` statement is12.
- *`inputzonename`* is the name of a text-formatting zone (or an alias defined earlier in the file) to include in this custom zone. Use spaces to sparate multiple zone names or alias names.

Note You cannot specify an output zone defined elsewhere as an input zone for this mapping rule.

This line is the mapping rule. The custom zone*`outputzonename`* is created only if this rule is satisfied.

- Use `any` before the list of zone names if document content is to be written to the custom zone wherever any of the listed input zones or aliases apply to the document.
- Use `all` if document content is to be written to the custom zone only where all of the listed input zones and aliases apply to the document.

A single `mapto` definition can include more than one mapping rule.

Tokenization Definitions

A tokenization definition specifies a collection field to tokenize and place in a custom zone. There can be more than one tokenization definition in the file. A `tokenizeto` definition has this form:

```
tokenizeto: outputzonename
{
  field: fieldname [wordcount]
}
```

where

- *`outputzonename`* is the name (for example, `urlkeyterms`) of the custom zone being defined to hold this field information.
- *`fieldname`* is the name of the collection field to tokenize. Most typically `VdkVgwKey` is the field used for this purpose, but you can specify any text-based collection field, including a custom field.
- *`wordcount`* is an optional numeric parameter that specifies how many words in the document (starting from the first word) to use in tokenizing *`fieldname`*.

  Only terms that appear in *`fieldname`* and also within the first *`wordcount`* words in the document are written into *`outputzonename`*. If *`wordcount`* does not appear or is zero, all words in the document are used.

End of File

The `style.tkm` file ends with the close of the `definitions` zone:

```
}
$$
```

Default style.tkm File

The default `style.tkm` file for collections is shown in Listing 8-2.

Listing 8-2 Default style.tkm file

```
$control: 1
definitions:
{
  aliasto: font1
  {
    zone: font
      /attribute: integer size > 19
  }

  aliasto: font2
  {
    zone: font
      /attribute: integer size > 13
      /attribute: integer size < 20
  }

  aliasto: ppt-font1
  {
    zone: font
      /attribute: integer size > 35
  }

  aliasto: ppt-font2
  {
    zone: font
      /attribute: integer size > 23
      /attribute: integer size < 36
  }

  mapto: vdk1
    /mimetypeexclude: application/x-ms-powerpoint text/xml
  {
    any: h1 h2 font1
  }

  mapto: vdk1
    /mimetypeinclude: application/x-ms-powerpoint
  {
    any: h1 h2 ppt-font1
  }

  mapto: vdk2
    /mimetypeexclude: application/x-ms-powerpoint text/xml
  {
    any: big b bold i italic u underline em strong h3 font2
  }

  mapto: vdk2
    /mimetypeinclude: application/x-ms-powerpoint
  {
    any: big b bold i italic u underline em strong h3 ppt-font2
  }

  tokenizeto: vdkvgwkeywords
  {
    field: vdkvgwkey 2500
  }
}
$$
```

Providing Passwords for Document Access (style.pw)

Verity K2 supports indexing of password-protected documents, which may include Zip files, PST files, and PDF documents. Passwords and password expressions for a collection are typically entered by an administrator into the **Defining Password Protected Files Settings** screen of the StyleSet Editor when constructing a style set for a collection that uses the ODBC, File System, or HTTP gateway. The passwords are stored in the style file `style.pw`, from which the collection-indexing tool retrieves them as needed during indexing.

There is no default version of `style.pw` provided with K2; the StyleSet Editor creates the file when an administrator needs to provides passwords for a particular collection.

You normally will not create or edit `style.pw` manually, except in cases where you need to index password-protected documents through a gateway for which the StyleSet Editor does not provide a password-editing interface. In such a situation, you will have to store the passwords in plain-text (unencrypted) form.

Listing 8-3 illustrates the syntax of `style.pw`:

Listing 8-3 Example style.pw file

```
$control: 1
# specify whether passwords in the file are plaintext or encrypted.
default is "no".
plaintext: yes
wildcard:
{
exp: *test1/xml/*.xml
```

```
  /passwd = H@w@1150
# specify whether the above wildcard exp is case sensitive. default is
"no"
  /ignorecase = yes
...
}
regex:
{
exp: test2/xml/.*\.xml$
  /password = H@w@1150
  /ignorecase = no
...
}
```

Each entry in the file is an expression that represents a document, followed by a password assignment and optionally an ignore-case flag.

Note that document specifications can be either wildcard expressions or regular expressions. (See "Supported Regular Expressions" on page 371 for Verity's regular-expression syntax.) The expressions in Listing 8-3 state that any file with an extension of `.xml` in the directory `test1/xml` or `test2/xml` is to be accessed using the specified password. In the case of the wildcard directive, case is ignored, so it would apply to both directories `test1/xml` and `test1/XML`, for example.

Passwords stored in `style.pw` are AES-encrypted and base-64 encoded, unless the value of the `plaintext` directive is `yes`.

Note these usage issues:

- The expression refers to a complete document path. Either use the complete path in the expression, or apply wildcards or regular-expression elements appropriately.
- If you use a regular expression and need to use special characters as literals, escape them by preceding them with a backslash(\):

  Write the expression `test[0].xml` as `test\[0\]\.xml`.
- In expressions (such as Windows pathnames) that include a literal backslash, double the backslash:

  Write the expression `test\xml\*.xml` as `test\\xml\\*.xml`.
- Expressions or passwords that contain " "(space) or "#" (poundsign) characters must be escaped by enclosing them in double quotes(`""`):

  Write the expression `Program Files` as "`Program Files`".

  Write the expression `*#2.xml` as "`*#2.xml`".

- Escaped characters within an expression that is in quotes must be double-escaped; that is, two backslashes precede a regular-expression special character, or two sets of double-backslashes represent a backslash.

  Write the expression `test\xml\#2.xml` as "`test\\\\xml\\\\#2.xml`".

- Accented characters in expressions or in passwords might not be handled correctly. Any character that is not a valid character in the locale of the style set's collection cannot be used in an expression or in a password. (Plain ASCII characters are valid in all locales, and therefore are the safest to use.)

- If the access passwords to any of the documents in a collection change, those documents cannot be re-indexed until the updated passwords have been added to `style.pw`.

Defining Indexing Stop Words (style.stp)

Excluding words from a word index can improve search performance. You can use the optional `style.stp` file to do that. You place into it character strings that would not be useful to search for, and those strings are excluded from the collection.

The `style.stp` file is rarely used. Its main purpose is for excluding rare constructs that look like words in documents (such as the 70-character "words" starting with *M* found in encoded files).

style.stp Syntax

A `style.stp` file is a flat ASCII file containing an excluded word list. The words in the list can appear in any order. The excluded word list should be left justified, and a separate word should appear on each line. A sample `style.stp` file is shown in Listing 8-4.

Listing 8-4 Example style.stp file

```
[0-9a-zA-z]
..........+
an
and
the
of
```

```
or
but
```

style.stp Features

When creating a `style.stp` file, you can control its case-sensitivity and you can also make use of regular expressions.

Case Sensitivity

By default, the collection word index is case-sensitive. If your collection word indexes are case-sensitive, the `style.stp` file must include every case combination for words you want to stop. For example, if you want to stop both “and” and “And”, you must include both entries in your `style.stp` file.

Regular Expressions

You can specify a regular expression as a word in the excluded word list. For example, the following regular expression could be entered:

```
[0-9a-zA-Z]
```

This regular expression excludes every one-letter word appearing in a collection’s documents from appearing in the collection word indexes.

You can also use regular expressions to exclude long words from a word indexes. Long words generally occur infrequently. Thus, users are less likely to search for them, and it is usually not crucial for them to be indexed.

To exclude words that are *n* characters or more in length, enter a regular expression consisting of *n* dots (.) followed by a plus sign (+) in your `style.stp` file. For example, to exclude words of 10 or more characters, enter 10 dots followed by a plus sign in your `style.stp` file, as follows:

```
    ..........+
```

Defining Indexing Go Words (style.go)

In a `style.go` file, you specify the words to be included in a collection's word indexes. A `style.go` file is optional. Words that comprise an industry-specific taxonomy can be specified for an included word list in a `style.go` file. If a `style.go` file is included in a style directory only those words included in this file are included in the word indexes.

The `style.go` file is rarely used.

style.go Syntax

A `style.go` file is a flat ASCII file containing an included word list. The words in the list can be entered in upper or lower case, and they can be listed in any order. The included word list should be left-justified, and a separate word should appear on each line. An example `style.go` file would look just like the sample `style.stp` file in the previous section.

You can specify a regular expression as a word in the included word list to identify a range of values to be included.

Defining Feature-Extraction Stop Words (style.fxs)

During indexing, a special feature extraction process evaluates documents and stores special feature vectors for all documents. The feature vectors are used to generate summaries and clusters for display in the application. The `style.fxs` file can be used to exclude words from appearing in document summaries and clusters.

For information about configuring an application to display clusters and/or summaries, see the documentation for your Verity application.

Words you might want to exclude are proper names, like the names of newspapers. In the `style.fxs` file, you list each word to exclude—one word per line. The stop word file is case-insensitive and accepts only literal words, not regular expressions.

It is possible for sentences containing some terms in the `style.fxs` to score high enough to be included in best sentence type summaries. The Verity engine uses the feature vectors to determine which sentences are the best for presentation in the summaries. The engine does not filter the document text to remove `style.fxs` terms when presenting summaries for display.

Customizing 7-Bit Tokenization (style.lex)

Note The `style.lex` file is used only by the older 8-bit table-driven locales, such as `english`. It is not used by or recommended for any current locales.

The character definitions for a collection affect how words are tokenized and stored in the word index. The character definitions used by the Verity engine are located in the CTYPE table in the older Verity locales. Many accented alphabetic characters are defined for each locale, so a `style.lex` file may not be required to index and search words with these characters. For nonalphanumerics not specified in the locale, use the `style.lex` file so that these characters are interpreted by the engine and words containing these characters will appear in the word index.

For example, if users want to enter nonalphanumeric characters (such as &, /, and ") as search criteria and these characters and not defined in the collection configuration, you can specify these characters in a `style.lex` file. If a `style.lex` file is present in the `style` directory, the word index is built based on the specifications made in the `style.lex` file. For example, if the character “/” is specified as a valid character, the word index will include that character and users can search for such words as “OS/2”.

style.lex File Syntax

Entries in the `style.lex` file identify the patterns that the Verity engine interprets as valid characters in words, punctuation such as newlines and white space, and characters used to perform retrievals such as end-of-sentence and end-of-paragraph delimiters. The application developer creates a `style.lex` file only when it is necessary to override the system defaults.

A sample `style.lex` file is shown in Listing 8-5. This file represents the closest approximation to the default `style.lex` file used by the Verity engine for processing 7-bit ASCII text. The internal implementation is platform dependent, which affects the accuracy of the extended ASCII characters.

Note `style.lex` file handles 7-bit characters only.

Listing 8-5 Example style.lex file

```
# style.lex -- 7-bit only version of internal hardwired lexer
$control: 1
lex:
{
    define: ALNUM "[A-Za-z0-9]"
    define: W "[ \t\f\r\v]"
    token: WORD "{ALNUM}+(\\.{ALNUM}+)*"
    token: EOS "[.?!][.?! \t]*"
    token: EOP "{W}*\n({W}*\n)+"
    token: NEWLINE "{W}*\n"
    token: WHITE "{W}+"
    token: PUNCT "[^A-Za-z0-9 \t\f\r\v.?!]+"
}
$$
```

General Information

The first noncomment lines in a `style.lex` file must be the following:

```
$control:1
lex:
```

After the `lex` statement, two types of keyword statements can be specified: `define` statements and `token` statements. The `define` statements are used to specify macros used in the `style.lex` file. The `token` statements are used to define words, paragraphs, white space, and so on. In the sample `style.lex` file above, the `define` statements are used to define allowed letters and numbers and valid white space characters. The token statements are used to define words, end of sentences, paragraphs and so on that occur in the documents contained in the collection.

In the `style.lex` file, the following symbols are used to create the token definitions.

Symbol Type	Symbol	Description
Quotes	`" "`	Specifies the elements that make up the `define` statement macro or `token` statement definition.
Brackets	`[]`	Defines a character class.
Braces	`{ }`	Specifies a macro that was created in a `define` statement.
Plus	+	Specifies one or more occurrences of a combination of characters and/or numbers.
Asterisk	`*`	Specifies zero or more occurrences of a combination of characters and/or numbers.
Two Backslashes	`\\`	Specifies an escape sequence. When two backslashes are used, it is to escape the second backslash. For instance, (`\\.`) is used to specify a floating decimal.
Pound Sign	#	Specifies that the characters following are a comment.

For additional information regarding regular expressions, see "Supported Regular Expressions" on page 371.

define Statements

The `define` statements used in the `style.lex` file specify macros to be used within the following `token` statements. When `define` statement macros are used in `token` statements, the macro is enclosed in braces {}. Use of `define` statements is optional.

token Statements

Each `token` statement contains a flag identifying tokens such as end-of-sentence, end-of-paragraph, and white space. The default patterns used to match these tokens appear in the various `token` statements. Typical tokens are listed here.

Token	Pattern
`WORD`	A word represented as any string comprised of alphanumeric characters (both uppercase and lowercase) or a floating decimal.
`EOS` `SENT`	An end-of-sentence character represented as either a period (.), question mark (?), or exclamation point (!). EOS and SENT are identical in meaning and are interchangeable.
`NEWLINE`	A single end-of-line represented as a newline.

Token	Pattern
EOP PARA	An end-of-paragraph represented as two or more newlines. EOP and PARA are identical in meaning and are interchangeable.
WHITE	A blank space represented by one or more white spaces.
PUNCT	Any character except a newline.

Statement Interpretation

Two statements of the same type in the `style.lex` file are ORed. For example, if you had the following two statements in your `style.lex`:

```
token:    WORD        "[A-Za-z]+"
token:    WORD        "[0-9]+"
```

then a word would be defined as any string of alphabetical characters or any string of numeric characters.

The order of the `token` statements in the `style.lex` file determines which token the lexical analyzer ("lexer") returns. The lexer returns the longest string that matches any pattern specified in the `style.lex` file. The token associated with that pattern is returned as well. If that string matches more than one pattern, the token that appears earliest in the `style.lex` is returned.

For example, if the following statements appeared in the order here:

```
token:   PUNCT       "."
token:   WORD        "[A-Z]+"
```

and the text looked like this:

```
"XY  Z"
```

then the letters "XY" would be returned as a `WORD` token, the white space would be returned as a `PUNCT` token, and the "Z" would be returned as a `PUNCT` token. The "Z" is not returned as a `WORD` token because it matches the patterns in both `TOKEN` statements, so the Verity engine selects the first matching pattern, in this case `PUNCT`.

As shown, a `token: WORD` statement typically contains a regular expression. If you specify a regular expression that contains a backslash (\), then you must enter two backslashes so that the Verity engine will interpret the additional backslash as a literal. Note that the double-backslash entry is not needed when specifying a predefined character. The backslash usage is consistent with all Verity control files.

A `style.lex` file must specify `token` statements for all the tokens you want the Verity engine to match. Note that default values for individual `token` statements are not provided.

Default Handling of the Dot Character

Using the default version of `style.lex`, the lexical analyzer considers the dot (.) to be a punctuation character. However, the lexer accepts a dot as part of a word if the dot is surrounded by alphabetical characters.

In this situation, the lexer performs the following tokenizations:

- **Agent.Server** becomes one word: **Agent.Server**.
- **P.Blumson** becomes one word: **P.Blumson**.
- **P. Blumson** becomes two words: **P** and **Blumson** (because space follows the dot).
- **P.Q. Blumson** becomes two words: **P.Q** and **Blumson**.

Character Mapping

`style.lex` does not index 8-bit characters (characters outside the standard ASCII range), even though they are valid in English documents. In addition, the character set for the `style.lex` file is the internal character set even if you set everything else in the application to a different code page (8859, for example), unless you add the `$charmap` option to the `style.lex` file, as shown here:

```
$control: 1
$charmap: 8859
lex:
{
  [...]
}
$$
```

The `$charmap` construct specifies that the contents should be mapped to the internal character set before being used for lexing.

PART II

Collection Tools Reference

9

Command-Line Tool Summary

This chapter lists the Verity K2 command-line tools that you can use to manage collections and perform other tasks. Some of these tools are described in this book, some are described in other books, and some are unofficial, untested tools that are undocumented.

Locations and tool names in Table 9-1 are for Windows platforms. For UNIX platforms, substitute forward slashes for the separators and remove the `.exe` extension, if any.

The metavariable *`platformDir`* refers to the full path to the platform-specific directory for the K2 or VDK installation. For example, on Windows:

```
C:\Program Files\Verity\k2\_nti40
```

The most important collection-management tools listed in this chapter are described in this book, in Chapter 10, "Using mkvdk," and Chapter 12, "Using Other Collection Tools." Other tools listed here are described in the referenced Verity books. Tools that are marked undocumented here are not described anywhere outside of this chapter.

This chapter contains two tables:

- Table 9-1 on page 268 lists the executable command-line tools delivered with Verity K2 or Verity VDK.
- Table 9-2 on page 276 lists sample programs that you can build and execute as command-line tools.

In both tables, tools are listed in alphabetic order.

Table 9-1 Verity K2 command-line tools

Tool name	Description
`adminconfigimport`	Copies Administration Server configuration settings during migration. **Location:** `platformDir\bin\adminconfigimport.exe` **See:** The chapter on `adminconfigimport` in the *Verity K2 Migration Guide*
`browse`	Lists the field names and values stored in a collection's document table, one partition at a time. **Location:** `platformDir\bin\browse.exe` **See:** "browse" on page 331
`chkvlkey`	Displays a list of the installed Verity components for which you are licensed. **Location:** `platformDir\bin\chkvlkey.exe` **See:** The discussion on updating license keys in the *Verity K2 Installation and Setup Guide*
`codeconv`	Converts text files from one character set to another. **Location:** `platformDir\bin\codeconv.exe` **See:** Appendix A of the *Verity Locale Configuration Guide*

`delbyqry`	Deletes, from the specified collection, documents that match the submitted query. Use the `-user` option to access secure collections.
	Syntax:
	`delbyqry [-locale locale] query collection [-user username[:password][:domain][:mailbox]]`
	`query` must be standard VQL; enclose `collection` path in quotes if it contains spaces.
	Note: If collection was built with vspider or K2 Spider, using `delbyqry` puts the vsdb out of sync with the collection. Using `vsdb` and `k2spider_cli vsdb` commands is the preferred way to delete records.
	Location:
	`platformDir\bin\delbyqry.exe`
	See:
	(undocumented)

Table 9-1 Verity K2 command-line tools (continued)

Tool name	Description
`didump`	Produces a word list for a collection, one partition at a time. Also produces a list of zones, if zones are used. **Location:** *platformDir*`\bin\didump.exe` **See:** "didump" on page 327
`extract`	Extracts the documents assigned to knowledge tree created by `ktmgr`, for use in converting it to a parametric tree. **Location:** *platformDir*`\bin\didump.exe` **See:** The discussion on converting knowledge trees in the *Verity Intelligent Classification Guide*
`fscrawl`	Sample application showing how to crawl a file system. It creates a BIF for a given file-system directory. **Location:** *platformDir*`\bin\fscrawl.exe` (source code in *installDir*`\k2\samples\vdk)` **See:** `samples` directory discussion in the *Verity Developer's Kit Programming Reference* Discussion on building and testing gateways in the *Verity Gateway Developer's Kit Programming Reference*
`genvlvdk`	Updates a Verity license key. For use by Verity only. **Location:** *platformDir*`\bin\fscrawl.exe`
`getlogs`	Retrieves query logs from K2 Servers and Brokers in preparation for populating or updating the K2 report index. **Location:** *platformDir*`\bin\getlogs.exe` **See:** The discussion on exporting consolidated log data in the reporting chapter of the *Verity K2 Administration Programming Guide.*
`k2admin`	The executable for the K2 Administration Server; can be executed as a command-line tool. **Location:** *platformDir*`\bin\k2admin.exe` **See:** The K2 Spider examples chapter of the *Verity Command-Line Indexing Reference*

Table 9-1 Verity K2 command-line tools (continued)

Tool name	Description
`k2collswap`	Brings a collection online by swapping the online and offline versions of it. **Location:** *`platformDir`*`\bin\k2collswap.exe` **See:** The discussion of custom user-defined jobs in the *Verity K2 Dashboard Administrator Guide*
`k2spider_cli`	Client process that uses `k2spider_srv` to index documents into a collection. **Location:** *`platformDir`*`\bin\k2spider_cli.exe` **See:** The K2 Spider Client chapter of the *Verity Command-Line Indexing Reference*
`k2spider_srv`	Server process that acts as controller, crawler and indexer to index documents into a collection. **Location:** *`platformDir`*`\bin\k2spider_srv.exe` **See:** The K2 Spider Server chapter of the *Verity Command-Line Indexing Reference*
`ktmgr`	Creates and maintains knowledge trees from scripts or from the command line. **Location:** *`platformDir`*`\bin\ktmgr.exe` **See:** The chapter on building knowledge trees from the command line in the *Verity Intelligent Classification Guide*
`ktsrch`	Runs a scoped search against a Knowledge Tree and groups results by category. **Location:** *`platformDir`*`\bin\ktsrch.exe` **See:** The chapter on testing knowledge trees in the *Verity Intelligent Classification Guide*
`langid`	Detects the language and character encoding of supplied documents. **Location:** *`platformDir`*`\bin\langid.exe` **See:** The language ID command tool appendix in the *Verity Locale Configuration Guide*

Table 9-1 Verity K2 command-line tools (continued)

Tool name	Description
`merge`	Allows you to merge or split collections. **Location:** *`platformDir`*`\bin\merge.exe` **See:** "merge" on page 335
`mkenc`	Creates an encryption file for a topic set. **Location:** *`platformDir`*`\bin\mkenc.exe` **See:** The chapter on building topic sets from the command line in the *Verity Intelligent Classification Guide*
`mklrc`	A command-line version of the Logistic Regression Classifier. It creates a topic from a set of positive and negative exemplary documents or from a taxonomy in a parametric index. **Location:** *`platformDir`*`\bin\mklrc.exe` **See:** The chapter on the Logistic Regression Classifier in the *Verity Intelligent Classification Guide*
`mkpi`	Creates and updates parametric indexes. **Location:** *`platformDir`*`\bin\mkpi.exe` **See:** The `mkpi` command-line tool reference in the *Verity K2 Parametric Developer Guide*
`mkprf`	Create, updates, or deletes a profile net. **Location:** *`platformDir`*`\bin\mkprf.exe` **See:** The chapter on K2 Profiler command-line tools in the *Verity K2 Profiler Programming Guide*
`mkre`	Manages recommendation indexes, transaction files, and K2 Server-Broker processes related to recommendation. **Location:** *`platformDir`*`\bin\mkre.exe` **See:** The chapter on administering the Recommendation Engine in the *Verity K2 Recommendation Engine Guide*

Table 9-1 Verity K2 command-line tools (continued)

Tool name	Description
`mkreport`	Populates the report index and allows export of log data. **Location:** `platformDir\lib\mkreport.jar` **See:** *Verity K2 Reporting API Developer Note* (Certain uses of this tool remain undocumented)
`mksyd`	Compiles a thesaurus control file into a thesaurus. **Location:** `platformDir\bin\mksyd.exe` **See:** The appendix on managing thesauruses in the *Verity Query Language and Topic Guide*.
`mktm`	A command-line version of the Thematic Mapper. It automatically extracts key concepts from a set of documents and organizes them into a hierarchy. **Location:** `platformDir\bin\mktm.exe` **See:** The chapter on thematic mapping in the *Verity Intelligent Classification Guide*
`mktopics`	Builds a topic set from an OTL file. **Location:** `platformDir\bin\mktopics.exe` **See:** The chapter on building topic sets from the command line in the *Verity Query Language and Topic Guide*
`mkvdk`	Creates, manages, and optimizes collections. **Location:** `platformDir\bin\mkvdk.exe` **See:** "Using mkvdk" on page 279
`qsrch`	a command-line tool for searching an optimized knowledge tree. **Location:** `platformDir\bin\qsrch.exe` **See:** The chapter on testing knowledge trees in the *Verity Intelligent Classification Guide*.

Table 9-1 Verity K2 command-line tools (continued)

Tool name	Description
`rcadmin`	A command-line alternative to the K2 Dashboard. You can use it to view and update configuration settings for the components of a K2 system. **Location:** *platformDir*`\bin\rcadmin.exe` **See:** The *Verity K2 rcadmin Guide*
`rcidx`	Writes detailed diagnostic information on all open collections and parametric indexes to a log file at *dataDir*`/services/`*index_alias*`/log/status.log`. For use by Verity Technical Support only. **Location:** *platformDir*`\bin\rcidx.exe` **See:** (undocumented)
`rck2`	A general-purpose tool that allows you to interact with a K2 system to search, retrieve from, and view the contents of collections as well as other types of Verity indexes. **Location:** *platformDir*`\bin\rck2.exe` **See:** *Verity K2 Dashboard Administrator Guide* (for collections) *Verity K2 Parametric Developer Guide* (for parametric indexes) *Verity K2 Profiler Programming Guide* (for profile nets) *Verity K2 Recommendation Engine Guide* (for recommendation indexes)
`rcodk`	A command-line tool for browsing a taxonomy or searching a parametric index. **Location:** *platformDir*`\bin\rcodk.exe` **See:** The chapter on testing parametric indexes in the *Verity Intelligent Classification Guide*
`rctk`	Sets an administrative user of the K2 Ticket Server. **Location:** *platformDir*`\bin\rctk.exe` **See:** The chapter on managing security in the *Verity K2 rcadmin Guide*

Table 9-1 Verity K2 command-line tools (continued)

Tool name	Description
`rcvdk`	A simple search client that allows you to search over a collection, list the collection fields, and display documents. **Location:** *`platformDir`*`\bin\rcvdk.exe` **See:** "rcvdk" on page 336
`regsvr32`	Registers or de-registers DLLs. **Location:** *`platformDir`*`\bin\regsvr32.exe` **See:** (undocumented)
`savecred`	Creates a user credentials database. **Location:** *`platformDir`*`\bin\savecred.exe` **See:** The credentials-database appendix of the *Verity K2 Documentum Gateway Guide* The introductory chapter of the *Verity K2 Lotus Notes Gateway Guide*
`standalone`	Launches the Verity StyleSet Editor, an MMC-based tool for managing the contents of style files. **Location:** *`productDir`*`\_nti40\standalone.cmd` (available on Windows only) **See:** Verity Spider examples chapter and K2 Spider Examples chapter of the *Verity Command-Line Indexing Reference*
`taxmgr`	Creates and maintains taxonomy databases from the command line or in scripts. **Location:** *`platformDir`*`\bin\taxmgr.exe` **See:** The chapter on building knowledge trees from the command line in the *Verity Intelligent Classification Guide*
`testqp`	Tests the validity of query syntax. **Location:** *`platformDir`*`\bin\testqp.exe` **See:** (undocumented)

Table 9-1 Verity K2 command-line tools (continued)

Tool name	Description
`top2tax`	Converts a topic set to a taxonomy (`.tax`) file. **Location:** *`platformDir`*`\bin\top2tax.exe` **See:** The chapter on building knowledge trees from the command line in the *Verity Intelligent Classification Guide*
`vconfig`	Configures an OEM installation. For OEM developers only. **Location:** *`platformDir`*`\bin\top2tax.exe` **See:** The chapter on configuring an installation in the *Verity OEM Deployment Guide*.
`vsdb`	Provides access to the Verity Spider persistent store. Useful for diagnosing `vspider` problems. **Location:** *`platformDir`*`\bin\vsdb.exe` **See:** vspider reference chapter of the *Verity Command-Line Indexing Reference*.
`vspider`	The Verity Spider. Indexes documents into a collection. **Location:** *`platformDir`*`\bin\vspider.exe` **See:** vspider reference chapter of the *Verity Command-Line Indexing Reference*.

Note The following programs are sample code only. They have not been tested and are not guaranteed to work correctly. They may need to be built (compiled and linked) before they can be used.

Table 9-2 Verity command-line sample programs

Program name	Description
`ezclust`	Groups collection documents that match a given query into groups, based on their feature vectors. **Location:** *productDir*`\samples\client\c\ezclust\` **See:** The clustering suite discussion in the *Verity K2 Client Programming Guide*.
`ezk2admin`	Illustrates use of the Administration C API to administer a K2 system. **Location:** *productDir*`\samples\client\c\ezk2admin\` **See:** The overview chapter of the *Verity K2 Administration Programming Guide*.
`ezk2prf`	Illustrates use of the K2 Profiler C API to administer profiles nets and evaluate documents against them. **Location:** *productDir*`\samples\client\c\ezk2prf\` **See:** The example applications chapter of the *Verity K2 Profiler Programming Guide*.
`ezk2srch`	Illustrates use of the K2 Client C API to search a collection and retrieve results. **Location:** *productDir*`\samples\client\c\ezk2srch\` **See:** The collection-search chapter of the *Verity K2 Client Programming Guide*.
`ezk2strm`	Illustrates use of the K2 Client C API to retrieve documents from a search-results list. **Location:** *productDir*`\samples\client\c\ezk2strm\` **See:** The document access chapter of the *Verity K2 Client Programming Guide*.
`ezstream`	Illustrates use of the VDK C API to retrieve documents from a search-results list. **Location:** *productDir*`\samples\client\c\ezk2strm\` **See:** The ezstream chapter of the *Verity Developer's Kit Programming Reference*.

Table 9-2 Verity command-line sample programs (continued)

Program name	Description
`ezwatch`	Illustrates use of the K2 Client C API to connect to a remote host and watch for events on that host. **Location:** *productDir*`\samples\client\c\ezwatch\` **See:** The maintenance-suite chapter of the *Verity K2 Client Programming Guide*.
`rmklrc`	A remote version of `mklrc` that works in a K2 environment. It illustrates use of the ODK API to execute the Logistic Regression Classifier. It creates a topic from a set of positive and negative exemplary documents or from a taxonomy in a parametric index. **Location:** *productDir*`\samples\client\java\rmklrc\` **See:** (undocumented)
`rmkpi`	A remote version of `mkpi` that works in a K2 environment. It illustrates use of the Parametric Java API to create and update parametric indexes. **Location:** *productDir*`\samples\client\java\rmkpi\` **See:** (undocumented)

Table 9-2 Verity command-line sample programs (continued)

Program name	Description
`rmktm`	A remote version of `mklrc` that works in a K2 environment. It illustrates use of the ODK Java API to automatically extract key concepts from a set of documents and organize them into a hierarchy. **Location:** `productDir\samples\client\java\rmktm\` **See:** (undocumented)
`rmktopics`	A remote version of `mktopics` that works in a K2 environment. It illustrates use of the ODK Java API to build a topic set from an OTL file. **.Location:** `productDir\samples\client\java\rmktopics\` **See:** (undocumented)
`rmkvdk`	A remote version of `mkvdk` that works in a K2 environment. It illustrates use of the Collection Indexing Java API to create, manage, and optimize collections. **Location:** `productDir\samples\client\java\rmkvdk\` **See:** (undocumented)

10

Using mkvdk

This chapter describes `mkvdk`, a command-line tool that you can use to build and maintain collections.

This chapter includes the following sections:

mkvdk Overview

`mkvdk` is a command-line tool for manipulating collections. For many tasks, alternative tools are available as well, such as the K2 Dashboard, K2 Spider, or Verity Spider. For some optimization tasks, however, there is no alternative to `mkvdk`.

This chapter explains `mkvdk` usage under the following headings:

- "mkvdk Overview" describes basic command syntax.
- "Creating and Indexing Collections" describes the fundamental commands for building collections.
- "Managing Collections" describes commands for modifying and manipulating collections.
- "Optimizing Collections" describes commands for improving search performance and minimizing collection size.
- "Servicing Collections" describes commands for controlling housekeeping operations on collections.
- "Controlling mkvdk Settings" describes commands for setting the characteristics of `mkvdk` or the collections it creates, such as language, logging level, and memory usage.
- "mkvdk Reference" gives a summary of all `mkvdk` command options and keywords.

Basic mkvdk Syntax

The following syntax is valid for `mkvdk`:

```
mkvdk -collection path [option] [...] [docSpec] [...]
```

where brackets (`[]`) indicate optional items and an ellipsis (`. . .`) indicates repetition of the previous item. Thus, the `mkvdk` command line can include multiple options and it can optionally include a series of *`docSpec`* arguments.

The `-collection` *`path`* argument is required for any manipulation on a collection. The options (none is required) are described in this chapter and tabulated in "mkvdk Reference" on page 303. All options must precede the first *`docSpec`* argument.

An optional *docSpec* argument can (1) specify a document to apply to the collection, or (2) specify the path to a file containing a list of documents (in the form of a simple list of document pathnames, one per line). If *docSpec* points to a file containing a list of documents, it should consist of an at-sign (@) followed by the file path, as in `@doclist`.

The default behavior of `mkvdk` (without any specified options) is to create or update the collection specified by the `-collection` argument, by indexing the documents specified by the *docSpec* arguments.

Indexing is one example of *servicing* a collection (see "Servicing Collections" on page 300); whenever `mkvdk` acts on a collection, it by default performs whatever servicing it has been configured to do.

`mkvdk` by default performs its work in the background, as resources become available. You can force indexing to happen immediately, in the foreground, by using the `-synch` option.

Accessing a List of Command-Line Options

To display a usage list of `mkvdk` command-line options, type:

```
mkvdk -help
```

Creating and Indexing Collections

To build a collection with `mkvdk`, you perform two separate tasks:

1. Create the collection structure.
2. Index documents into the collection.

You can put these tasks in two separate `mkvdk` commands, or you can combine them on a single command line.

Creating a Collection

The simplest `mkvdk` command for creating a collection is this:

```
mkvdk -create -collection collpath
```

where *collpath* is the pathname of the collection directory you wish to create. With this command, the Verity engine creates a collection directory including a default set of style files (see "Standard and Default Style Sets" on page 54).

If you want to specify a non-default style set—for example if you are using a gateway other than the HTTP or file-system gateways, or if your collection configuration needs to be customized in anyway— use the `-style` option on the `mkvdk` command line to specify the location of a style set to use with the collection.

The following command creates a collection in *path_1* using the style set specified in *path_2*:

```
mkvdk -create -collection path_1 -style path_2
```

Optionally, you can assign a short description to the collection, using the `-description` option. That description is stored with the collection and can be displayed by a search application, or by using the `-about` option (see Table 10-1 on page 304) of `mkvdk`.

When you create a collection, you can also apply various collection-optimization options to change the indexing mode or to create additional index features. See "Optimizing Collections" on page 289 for details.

Indexing Documents Into a Collection

Once the collection structure is created, you can submit documents to it for indexing.

IMPORTANT The `-create` option can be used only once, to create the collection directory structure. After a collection directory structure has been created, do not use the `-create` option to update the collection.

Specifying Documents on the Command Line

The following command indexes the documents *docSpec1* and *docSpec2* into the collection whose pathname is *path*:

```
mkvdk -collection path docSpec1 docSpec2
```

where *docSpec1* and *docSpec2* are document keys appropriate to the current gateway—for example, pathnames if you are using the file-system gateway.

You can combine collection creation and indexing in a single command. The following command creates a collection in *path_1*, using the style set specified in *path_2*, and it indexes the documents listed in the file *docList*:

```
mkvdk -create -collection path_1 -style path_2 @docList
```

where *docList* contains a simple list of document keys, one per line.

Indexing With a BIF

You can add a large number of documents to a collection, and you can add field information as well as document content, by using one or more bulk insert files (BIFs) and the mkvdk options -bulk and -insert.

The following command inserts the documents specified in the bulk insert file specified by *BIFpath* into the existing collection at *collPath*:

```
mkvdk -collection collPath -bulk -insert BIFpath
```

Because insertion is the default action associated with -bulk, the following command has the same effect:

```
mkvdk -collection collPath -bulk BIFpath
```

Note In your BIF, do not mix new documents with documents that may already exist in the collection. If you specify -insert and mkvdk attempts to index a document that is already in the collection, a duplicate key for that document will be created.

You can use the -bulk option with other options (-update and -delete) to update or delete existing files in a collection, or to modify collection fields. See "Managing Collections" on page 286.)

You can combine collection creation and indexing. The following command both creates the collection at the location *collpath* and inserts the documents specified in the two bulk insert files at *BIFpath1* and *BIFpath2*:

```
mkvdk -create -collection collpath -bulk -insert BIFpath1 BIFpath2
```

You can use the mkvdk options -offset and -numdocs to control where in a BIF to start indexing from, and how many document s to index. These options are especially useful for repeated indexing from a single BIF, as described in "Supporting Continuous Feeds" on page 321.

If you include the -autodel option along with -bulk on the command line, the specified BIF will be automatically deleted after indexing.

For more information on bulk insert file format and usage, see "Using Bulk Insert Files" on page 313.

Specifying a Base for Relative Pathnames

As noted in "Specifying Absolute or Relative Collection Paths" on page 295, if your collection uses the file-system gateway, you can provide either full or partial pathnames when specifying document locations to `mkvdk`.

By default, partial pathnames are relative to the collection's directory. However, you can specify a different base to use, by including the `-datapath` option on the `mkvdk` command line. Each supplied document pathname (whether on the command line or in a specified BIF) is appended to the path specified in `-datapath`, and the resulting pathname is stored in the collection.

Populating Collection Fields

You may wish to include field values related to your documents into your collection, so that field data can be displayed and the collection can be searched by field value. `mkvdk` supports populating collection fields by use of a bulk insert file, and also by use of the `mkvdk` field-extraction capability.

Putting Field Data in a BIF

To use a bulk insert file to populate fields while indexing a collection, take the following steps:

1. Define the fields in the `style.sfl` and/or `style.ufl` file, as appropriate.

   For more information about `style.sfl` and `style.ufl`, see "Defining Collection Fields" on page 145.

2. Create a bulk insert file that specifies the documents to insert and the field values for each document.

   For more information on BIF format and usage, see "Using Bulk Insert Files" on page 313.

3. Run `mkvdk` using the `-bulk` option and the `-insert` flag, specifying the bulk insert file or files.

Using Field Extraction

The Verity field-parsing and field-extraction features allow you to populate collection fields by specifying regular expressions that the Verity engine applies to the collection's documents to extract field data during indexing. Take the following steps to populate fields by extracting field values:

1. Define the fields in the `style.sfl` and/or `style.ufl` file.

   For more information about `style.sfl` and `style.ufl`, see "Defining Collection Fields" on page 145.

2. Create a `style.tde` file containing the field extraction rules.

3. Run `mkvdk` using the `-extract` option, like this:

   ```
   mkvdk -collection path -extract -bulk -insert BIFpath
   ```

   where *`BIFpath`* is the pathname of a BIF specifying the documents to index (and to extract field data from).

For a full discussion of the field extraction process and the rules that you can write into `style.tde`, see "Populating Collection Fields" on page 167.

Using a Field-Extraction Work List

By default, when `mkvdk` processes documents for field extraction, it writes a *work list* (a BIF file named `worklist` that contains the extracted fields) to the collection directory. Use of the work list allows you to separate the field extraction from the indexing process—if, for example, you want to edit the extracted fields before indexing or you want to batch the indexing in a particular way.

If you want that separation to occur, you can run `mkvdk` with the `-nosubmit`, `-noservice`, or `-noindex` option, like this:

```
mkvdk -collection path -extract -nosubmit -bulk -insert BIFpath
```

In this case, the extracted fields are written to the work list in the collection directory, and an internal BIF is created to specify the documents. In a separate, later action—perhaps after manually editing the work list—you can run `mkvdk` again to add the documents and revised field data to the collection.

```
mkvdk -collection path
```

If you do not want this separation to occur and you do not even want the work list to be saved, you can suppress its creation by using the `-nosave` option (in place of `-nosubmit`). The use of `-nosave` is not common.

Managing Collections

The `mkvdk` tool offers a broad range of collection-management capabilities.

Updating Document Content and Fields

You can use a bulk insert file to update a collection, that is, to re-index changed documents or insert updated values into existing fields. To perform the update, you submit a BIF to `mkvdk` and use the `-bulk` option along with the `-update` option.

The following command updates the collection at *`collpath`* by replacing all current information related to the documents specified in the bulk insert file at *`BIFpath`* with new, re-indexed information:

```
mkvdk -collection collpath -bulk -update BIFpath
```

Note You can mix new documents with updated documents in your BIF. For a document that is not already in the collection, specifying `-update` is equivalent to specifying `-insert`.

Deleting Documents

You can use mkvdk `-delete` option to delete documents from a collection. The following command deletes the two specified documents from a collection:

```
mkvdk -collection collPath -delete fileSpec1 fileSpec2
```

are document keys appropriate to the current gateway. You can alternatively use the *`@docList`* argument to specify a file containing a list of document keys.

The following command specifies bulk deletion of a set of documents:

```
mkvdk -collection collPath -bulk -delete BIFpath
```

where *BIFpath* specifies a bulk insert file that identifies the files to delete. It can be the same BIF used to insert documents; the only difference is that the `-delete` flag is specified instead of `-insert` (or instead of no flag).

Note The `-delete` option actually only marks documents for deletion. After deleting documents from a collection in this way, you should perform a squeeze on the collection to remove the document keys. For instructions, see "Squeezing" on page 290. See also the discussion of squeezing in the chapter on managing collections in the *Verity K2 Dashboard Administrator Guide*.

Updating Fixed-Width Collection Fields Without Re-Indexing

You can use a bulk insert file to update fixed-width fields in a collection without re-indexing the documents. To do this, you create a BIF containing the document field information to be updated, then run `mkvdk` with the `-modify` flag.

By default, the Verity engine takes the collection off line, then reads the bulk insert file, updates values for the requested fields, writes information to collection files (without rewriting the partition), and rebuilds the indexes for any indexed fields. After the updated collection information is committed to disk, the engine brings the collection back on line, so it is once again available for searching.

If you need to leave the collection in an online state while updating it, you can use the `-online` option:

```
mkvdk -collection collPath -bulk -modify -online BIFpath
```

Note Each document record in your BIF should contain only the `VdkVgwKey` field plus collection fields of type `fixwidth`. Collection field types are described in "Field Types" on page 147.

Backing Up a Collection

You can use the `-backup` option to create a backup of a collection. The following command backs up the collection at *collPath* into the directory *backupPath*.

```
mkvdk -collection collPath -backup backupPath
```

`mkvdk` deletes the contents, if any, of the backup directory before copying the new collection into it. If the backup directory does not already exist, `mkvdk` creates it.

Purging a Collection

Purging a collection means deleting all document information from it, but not deleting the collection itself. Purging leaves the collection directory structure intact.

Purging might be necessary if you have changed the collection's style set or its indexing starting points, or otherwise need to remove all currently indexed content before re-indexing.

With `mkvdk`, you use the `-purge` option to purge a collection. By default, purging occurs as a foreground process; to purge in the background, add the `-purgeback` option.

The following command purges the collection specified by *`path`*:

```
mkvdk -collection path -purge
```

The following command purges the collection in the background.

```
mkvdk -collection path -purge -purgeback
```

IMPORTANT A collection must be offline when you purge it.

To wait a specified amount of time before starting the purge, add the `-purgewait` option to the command line. With `-purgewait`, you specify the number of seconds that `mkvdk` is to delay before starting the purge. If you use `-purgewait` but specify no value for it, the wait is 600 seconds.

The following command purges the collection specified by *`path`*, after waiting 300 seconds:

```
mkvdk -collection path -purge -purgewait 300
```

Repairing a Collection

A collection needs *repair* if it has corrupted partitions. You will know that repair is required if `mkvdk` reports the state of a collection as

```
State = Collection needs repair
```

In such a situation, use the `-repair` option of `mkvdk`, as in this example:

```
mkvdk -collection path -repair
```

If you manually repair a collection instead of having `mkvdk` do it, run the `mkvdk -repair` command after the repair, to enable the collection. (For manual repair assistance, contact Verity Technical Support.)

Optimizing Collections

There are several ways that you can use `mkvdk` to create collections that are optimized for various purposes.

Using Optimized Indexing Modes

By using the `-mode` option of `mkvdk`, you can optimize a collection for various broad modes of usage. The `-mode` option controls the way the collection is built, depending on the value of *`mode`*:

```
-mode mode
```

Indexing with these modes is described in more detail in Chapter 3, "Setting Indexing and Search Policies." These are the available values for *`mode`*:

- `generic`. The standard indexing mode. Gives the best overall average performance. For more information, see "Generic Mode" on page 86.
- `fastsearch`. Optimized for fastest return of results at search time. For more information, see "Fast Search Mode" on page 87.
- `bulkload`. Optimized for indexing large numbers of documents through BIFs. For more information, see "Bulk Load Mode" on page 87.
- `newsfeedidx`. Optimized for indexing frequent small batches of documents without falling behind. For more information, see "News Feed Indexer Mode" on page 88.
- `newsfeedopt`. Merges partitions created by the `newsfeedidx` mode. For more information, see "News Feed Optimizer Mode" on page 88.
- `rdonly`. Disables writing to the collection after indexing. For more information, see "Read Only Mode" on page 89.

If you do not include the `-mode` option when indexing with `mkvdk`, the default mode (`generic`) is used.

Using the -optimize Option

By using the `-optimize` option of `mkvdk`, you can perform various specific optimizations on a collection. Most of these optimizations are more narrowly focused than the optimizations performed by the indexing modes described in the previous section. (In fact, the indexing modes are mostly made up of combinations of these specific optimizations.)

You can perform one or several optimizations in a single command, by following the `-optimize` option with one or more optimization keywords:

```
-optimize keyword[-keyword...]
```

If you specify multiple keywords, separate them with hyphens, as in

```
-optimize maxmerge-squeeze-readonly
```

These are the available optimization keywords:

```
maxclean    maxmerge    ngramindex

publish     readonly    spanword

squeeze     tuneup      vdbopt
```

The following sections discuss the optimizations you can perform with these keywords, as well as with other options.

Squeezing

When a document is deleted from a collection, the space that it occupied is not recovered. The document is merely marked as deleted, and it is not available for subsequent searches. The Verity technique *squeezing* actually removes deleted documents from the collection's document table and word indexes, creating a smaller collection and reducing the collection's disk space. The smaller collection has a more efficient structure that uses somewhat less memory and can be searched somewhat faster.

It is safe to squeeze a collection at any time. You do not need to temporarily disable a collection for squeezing, because the `mkvdk` self-administration capability assigns a new revision code to the collection and ensures that it remains available for searching and servicing. After a squeeze has occurred, the Verity engine points the application to the new collection data.

Squeezing is a significant update to a collection. If users are reviewing search results at the time when squeezing occurs, the search results may be invalidated after the squeeze completes.

To perform a squeeze, you use the `squeeze` optimization keyword with the `-optimize` option, as in this example:

```
mkvdk -collection path -optimize squeeze
```

Incremental Squeeze

Incremental squeeze is a collection optimization feature that allows the application administrator to save on the disk space required for squeezing a collection. Incremental squeeze uses significantly less disk space to squeeze a collection than does a normal squeeze.

When normal squeezing is performed on a collection, the disk space required can be up to twice the size of the collection. During a squeeze, all collection partitions that have deleted documents are recreated without the deleted documents. After the squeeze, the original partitions are removed. During the time of squeezing, both the old and new partitions exist, allowing for continuous search access to the collection but occupying perhaps nearly double the space.

With incremental squeeze, the Verity engine squeezes the partitions in the collection one by one. After each partition is squeezed into a new partition, the corresponding old partition is immediately removed. The behavior of incremental squeeze ensures that the extra disk space required to squeeze a collection is no more than the size of the largest partition.

To implement incremental squeeze, you run `mkvdk` with a set of style files that include a `style.plc` file with a special `/incremental_squeeze=YES` entry. The entry is specified as an attribute to the indexing mode used. here is a sample `style.plc` file that implements incremental squeeze with the `default` indexing mode:

```
$control: 1
    policy:
    {
      mode: default
        /inherit=generic
        /incremental_squeeze=yes
    }
```

With this `style.plc` file, the Verity engine performs an incremental squeeze when an `mkvdk` call includes the `-optimize squeeze` option.

Creating a Spanning Word List

A collection consists of one or more partitions, each of which includes a word index. During search processing, the Verity engine needs to separately load in and search the word index from each partition. In a large collection with numerous partitions, the presence of multiple word indexes can slow searching significantly.

You can improve search speed in large collections by creating a *spanning word list*, a word index that encompasses all the partitions. A spanning word list gives a search application quicker access to documents by allowing it to search more efficiently across the entire collection.

Note For spelling suggestion to function in a search application, a spanning word list is required for each collection searched.

Potential disadvantages of creating a spanning word list include the extra indexing time required to construct it, and the somewhat larger collection size that results.

You can use `mkvdk` to create a spanning word list in two ways. One method is to use the `spanword` optimization keyword with the `-optimize` option:

```
mkvdk -collection path -optimize spanword
```

An alternative method is to use the `-words` option:

```
mkvdk -collection path -words
```

Creating an ngram Index

An ngram index is a collection structure that improves the search performance for queries that need to match partial words. VQL queries that use the `<TYPO>` or `<WILDCARD>` operators, for example, benefit greatly from having an ngram index.

The disk space required for the ngram index is not significant; it represents a very small percentage of the total size of the collection. Building an ngram index does take some time, however, so applications that are not dependent on fast wildcard searches may not need it.

Note If you create an ngram index, you must also create a spanning word list.

Note For spelling suggestion to function optimally in a search application, an ngram index is recommended for each collection that is searched.

You can use mkvdk to create an ngram index in two ways. One method is to use the ngramindex optimization keyword with the -optimize option. The following example creates both a spanning word list and an ngram index:

```
mkvdk -collection path -optimize spanword-ngramindex
```

An alternative method is to use the -wordindex option. The following example also creates both a spanning word list and an ngram index:

```
mkvdk -collection path -words -wordindex
```

Creating a Topic-Set Index

If a topic set is attached to your collection, you can index it to shorten search and retrieval time. To create the topic-set index, the Verity engine searches the collection using each of the topics in the topic set and stores the results. By holding pre-computed results of all topic searches, the topic index allows for much faster searching.

Potential disadvantages of creating a topic-set index are that it increases the time required to index a collection and that it increases the collection size.

To create the topic-set index, you use the -topicset option of mkvdk and provide a path to the topic set. For example:

```
mkvdk -collection collPath -topicset topicsetPath
```

The topic-set index is stored in the topicset subdirectory of the collection directory.

Note When a topic set is first created with the mktopics command-line tool, it has an *index type* (“normal” or “named”), which determines what kind of topic-set index will be created if the topic set is indexed. For more information, see the chapter on building topic sets from the command line in the *Verity Query Language and Topic Guide*.

Optimizing Partitions

You can decrease the number of partitions in a collection to a minimum number by using `mkvdk` to perform a merge. The merge operation makes each partition as large as possible (containing up to 64,000 documents), so that the least number of partitions is used to hold all the collection's documents. Having fewer partitions can improve search performance.

To perform the merge, use the `maxmerge` optimization keyword with the `-optimize` option:

```
mkvdk -collection path -optimize maxmerge
```

Cleaning Up and Publishing

Before making a collection available to search-application users, it is typical to perform cleanup procedures on it. It might also be desirable to make it read-only, to prevent accidental or malicious alteration of any of its data. A number of `mkvdk` housekeeping options are available for these purposes.

- **Cleanup**. To perform the most comprehensive cleanup possible and to remove out-of-date collection files, use the `maxclean` optimization keyword with the `-optimize` option:

  ```
  mkvdk -collection path -optimize maxclean
  ```

  Performing this cleanup is recommended only when you are preparing an isolated (never updated or changed) collection for publication.

- **Tuneup**. A tuneup consists specifically of a merge and the creation of a spanning word list. Use the `tuneup` optimization keyword with the `-optimize` option:

  ```
  mkvdk -collection path -optimize tuneup
  ```

  Performing a tuneup is equivalent to specifying `-optimize maxmerge-spanword`.

- **Packing**. Each collection includes a proprietary structure (VDB) that supports fast access to document information in the collection. Optimizing the VDB packs its information into a dense format that allows for extra-fast access. VDB optimization happens during normal indexing operations, but you can also perform it manually, using the `vdbopt` optimization keyword with the `-optimize` option.

  ```
  mkvdk -collection path -optimize vdbopt
  ```

- **Read-only**. Use the `readonly` optimization keyword with the `-optimize` option to prevent a collection from being altered in any way.

```
mkvdk -collection path -optimize readonly
```

Naturally, you would not want to perform this optimization on a collection that might be updated in the future. It is appropriate for collections that you are publishing as static collections, perhaps on a CD-ROM.

- **Publishing**. Once your collection is finalized, and if you never expect to update it, you can use the `publish` optimization keyword with the `-optimize` option to prepare it for final publication, such as to a network server or CD-ROM:

  ```
  mkvdk -collection path -optimize publish
  ```

  This optimization includes all the other optimizations available through the `-optimize` option, except for `ngramindex`. In other word, it is equivalent to

```
-optimize maxclean-maxmerge-readonly-spanword-squeeze-tuneup-vdbopt
```

Controlling mkvdk Settings

This section describes how to use `mkvdk` options and parameters to change global collection characteristics or to affect the functioning and out put of `mkvdk` itself.

Accessing Secure Repositories

If `mkvdk` requires access to a secure repository, you can use the `-credentials` option to pass a user name and password to the gateway. The option takes this form:

```
-credentials username[:password][:domain][:mailbox]
```

where `-credentials` is followed by the specific login credentials that are required by the gateway involved.

Specifying Absolute or Relative Collection Paths

A Verity collection stores the paths to its files in one of two ways:

- A relative path shows the location of the file relative to the collection’s directory.
- An absolute path shows the global location of the file, independent of the location of the collection’s directory.

An individual collection can contain all relative, all absolute, or a mixture of relative and absolute pathnames. When indexing a document, the Verity engine simply uses the pathnames as provided in the *`dockey`* arguments on the `mkvdk` command line. Partial paths are taken as relative to the collection directory; the concept of "current working directory" has no significance to the engine.

In general, relative paths are the most versatile and portable. A convenient way to manage a set of documents and its associated collection is to set up a parent directory that contains both the collection and the repository. However, there are two situations in which absolute paths are preferable:

- Collections on Windows require absolute paths unless the search is being conducted from the same drive that contains the collection and the documents. This is because these systems use drive letters. It is not possible to create a relative path that crosses from one drive letter to another. Thus, relative paths cannot be used if a document exists on a different drive from the collection.
- Absolute paths are also required when the collection and the repository will not exist together as a unit. This happens, for example, when the two are stored in directories far removed from each other. This would be the case when the data is owned by another application.

In general, UNIX systems are the most flexible. Their symbolic linking facilities can be used to work around difficult situations. For example, if a collection that was built on a Windows platform using absolute paths (starting with drive letters) is moved to UNIX, symbolic links with names like `E:` can be created in order to use the collection.

Working With Locales

Every Verity collection is created within the context of a single Verity locale. The collection's documents are presumed to be in the language (or languages) of that locale and are processed according to that locale's rules.

Whenever `mkvdk` executes, it establishes a session locale. You can explicitly specify the session locale using the `-locale` option on the command line. If you do not specify a locale, the default session locale (usually `uni`) is used.

When you create a collection, use the `-locale` option if you want the collection to be in a locale other than the default.

Note If you specify uni for the session locale, you can further specify a default session language, by adding a language designation, like this:

```
-locale uni/fr
```

If `uni` is the default session locale, specifying no locale or specifying `uni` without a language designation is equivalent to specifying `uni/en`.

The locales and language codes supported by Verity are listed in Appendix A of the *Verity Locale Configuration Guide*.

Working With Character Sets

Every collection uses a specific character set, the default (or internal) character set of the collection's locale. When you index a collection and the command line specifies the documents to be indexed, those documents are assumed to be in the collection's internal character set unless you specify their character set using the `-charmap` option.

If you do specify a character set, the documents you submit must be in that character set. They are then converted to the collection's character set by the Verity engine before indexing.

IMPORTANT If you specify a character set using `-charmap`, that character set must be one of the character sets supported by the locale of the collection you are accessing. Supported character sets for each Verity locales are listed in Appendix A of the *Verity Locale Configuration Guide*.

If you submit documents in a BIF (bulk insert file; see "Indexing With a BIF" on page 283), each document can be in a different character set, because the BIF can specify a character set for each document. (The BIF itself must be in the collections's character set.)

Specifying Date Formats

During indexing, if you use the mkvdk options `-extract` (for extraction of field data) or `-bulk` (for submission of field data through a BIF), the field data may include dates. By default, the Verity engine interprets ambiguous numerical dates (such as 04/04/04) as being in month-day-year (MDY) format, where M represents a two-digit month, D represents a two-digit day, and Y represents a two-digit year.

If the data you are submitting uses a different ordering of date components, comparison operations on those date fields will not work correctly. To correct this problem, you can use the `-datefmt` option to specify that the date information you are submitting is in a different format from MDY. The following format values are supported:

```
MDY
DMY
YMD
YDM
USA
EUR
```

See“mkvdk Reference” on page 303 for explanations of each of the formats.

Managing Memory Usage

You can use options to set certain characteristics of the VDK session that `mkvdk` establishes when it executes.

- **Maximum memory**. You can specify the maximum amount of memory that `mkvdk` is permitted to use when processing a command. Use the `-maxmemory` option to set this value (in KB). For example:

  ```
  mkvdk -collection path -maxmemory 20000 -bulk -insert BIFpath
  ```

  If you do not specify `-maxmemory`, `mkvdk` will use as much memory as it can. You might want to set an explicit value to minimize interference with other applications.

- **Maximum open files**. You can specify the maximum number of files that `mkvdk` is permitted to keep open at one time. Use the `-maxfiles` option to set this value. For example:

  ```
  mkvdk -collection path -maxfiles 100 -bulk -insert BIFpath
  ```

  If you do not specify `-maxfiles`, the default value (50) applies.

- **Disk cache size**. You can specify the size of the VDK disk cache that is set up when `mkvdk` executes. Use the `-diskcache` option to set this value (in KB). For example:

  ```
  mkvdk -collection path -diskcache 1000 -bulk -insert BIFpath
  ```

  If you do not specify `-diskcache`, the default value applies.

  In general, a large cache size is most useful for searching, and a small cache size for indexing.

Managing System Messages

You can use command options to control the level of detail that `mkvdk` uses in handling errors and system messages, and to specify how it saves or displays them.

To control how much information `mkvdk` writes to the console screen, use the `-outlevel` option and specify a number between 1 and 127.

To control how much information mkvdk writes to a log file, use the `-loglevel` option and specify a number between 1 and 127. If you use `-loglevel`, you must also use the `-logfile` option to specify the file that `mkvdk` should write the log information to. (No log file is written if you do not specify these options.)

You determine the value to supply for `-outlevel` or `-loglevel` by adding up the number values for any of the following message types:

- `Fatal=1`
- `Error=2`
- `Warning=4`
- `Status=8`
- `Info=16`
- `Verbose=32`
- `Debug=64`

For example, to specify that only fatal, error, and warning messages should be logged, but that all messages up through the verbose level should be displayed onscreen, the command line could include these options:

```
-outlevel 63 -loglevel 7 -logfile filePath
```

The default value for both `-outlevel` and `-loglevel` is 15 (includes fatal, error, warning, and status messages).

These other options also affect the output of system messages:

- You can use the `-verbose` option—which is equivalent to specifying `-outlevel 63`—when you want `mkvdk` to write all system messages up through the verbose level to the screen.
- You can use the `-debug` option—which is equivalent to specifying `-outlevel 127`—when you want `mkvdk` to write all system messages plus debugging information to the screen.

- Regardless of the current value for `-outlevel`, you can force `mkvdk` to write no more than fatal and error messages to the screen by including the `-quiet` option on the command line.
- If you want mkvdk to return error codes when it encounters bad document keys during indexing, use the `-errorcodes` option.

Servicing Collections

By default, when you invoke `mkvdk` to index a collection, a two-step process occurs:

1. `mkvdk` submits the documents to the collection for indexing.
2. `mkvdk` retrieves the documents and indexes them into the collection.

The indexing step is an example of *servicing* a collection, and it is separate from the submission step. The two steps need not occur in immediate sequence, and they need not even be performed in the same session (that is, by the same invocation of `mkvdk`.)

You can use multiple executions of `mkvdk`, using various service-related options, to divide and share the servicing of collections. For example, you could set up a persistent `mkvdk` session that periodically looks in a collection for documents to index, and you could then—as needed—invoke `mkvdk` non-persistently to submit (but not index) documents to that collection. The persistent session would then pick up and index the submitted documents.

This section describes how to specify different kinds of servicing capability for different `mkvdk` sessions.

Setting the Service Level

There are three general classes of service that can be performed on a collection:

- Indexing-related tasks such as inserting, deleting, and updating documents.
- Optimization tasks such as creating special indexes and merging partitions.
- Housekeeping tasks such as removing corrupted or unneeded files.

`mkvdk` recognizes several *service levels* that cover these classes. You can use the `-servlev` option, along with one or more keywords, to set the level of servicing that `mkvdk` is permitted to perform on a given collection. For example:

```
mkvdk -collection path -servlev optimize
```

In this example, optimization is the only form of servicing that this session of `mkvdk` can perform.

The keywords that you can specify with `-servlev` are the following:

- `search`. Enable search and retrieval.
- `insert`. Enable adding and updating documents.
- `optimize`. Enable collection optimization. See "Using the -optimize Option" on page 290.
- `assist`. Enable building of a spanning word list or ngram index.
- `housekeep`. Enable housekeeping of unneeded files and partitions.
- `delete`. Enable document deletion.
- `backup`. Enable backup of the collection.
- `purge`. Enable purging of the collection.
- `repair`. Enable collection repair. See "Repairing a Collection" on page 288.
- `dataprep`. Same as `search-index-optimize-assist-housekeep`.
- `index`. Same as insert-delete.

You can allow multiple levels of servicing by combining keywords on the command line, separated by hyphens:

```
mkvdk -collection path -servlev search-optimize-assist
```

If you specify no service level, it is equivalent to specifying

```
-index -optimize -housekeep
```

Prohibiting Specific Service Levels

You can also use the following options to deny particular levels of service to `mkvdk`:

- **No service**. Use the `-noservice` option to prevent any servicing from occurring.
- **No housekeeping**. Use the `-nohousekeep` option to disallow housekeeping.
- **No indexing**. Use the `-noindex` option to disallow indexing.
- **No optimization**. Use the `-nooptimize` option to disallow optimization.

Persistent Servicing

You can use the `-persist` option to run `mkvdk` as a persistent process. If you do that, you can then use the `-sleeptime` option to perform servicing repeatedly, at specified intervals.

The following command runs `mkvdk` as a persistent process and repeatedly services the collection at *`path`*, with idle waits of *`num`* seconds between servicings.

```
mkvdk -collection path -persist -sleeptime num
```

If you specify `-persist` but do not give a value for `-sleeptime`, `mkvdk` uses the default idle wait time of 30 seconds.

Servicing Examples

This section gives a few simple examples of configuring servicing with `mkvdk`.

Default Servicing

Whenever `mkvdk` is invoked against a collection, it services the collection. If no service-related options are specified on the command line, `mkvdk` performs the default levels of service: it indexes, optimizes the collection, and performs housekeeping.

The following command performs default servicing only:

```
mkvdk -collection path
```

This command might or might not cause indexing, optimization, or housekeeping to occur, depending on whether previous submissions are pending.

Periodic Indexing

The following command executes `mkvdk` as a persistent process that, once an hour, checks the collection at *`path`* for newly submitted documents and indexes them into the collection:

```
mkvdk -collection path -persist -sleeptime 3600 -servlev index
```

Periodic Optimization

The following command executes `mkvdk` as a persistent process that, once a day, optimizes the collection at *`path`*:

```
mkvdk -collection path -persist -sleeptime 86400 -servlev optimize
```

The kind of optimization performed depends in the current indexing mode, which you can specify with the `-mode` option.

Document Submission (No Indexing)

The following command submits the documents specified in *`BIFpath`* for indexing into the collection at *`path`*, but does not perform the indexing or any other service:

```
mkvdk -collection path -noservice -bulk -update BIFpath
```

Using this command in conjunction with the periodic-indexing and periodic-optimization commands listed in the previous sections allows you to spread the collection-servicing burden among several processes.

mkvdk Reference

This section summarizes the mkvdk command syntax and lists all command-line options.

Command Syntax

```
mkvdk -collection path [option] [...] [docSpec] [...]
```

`-collection` *`path`* is required to create or open a collection. All command options must precede the first *`docSpec`* parameter

If *`docSpec`* refers to a file containing a list of files, it should consist of an at-sign (@) followed by the file name containing the list, as in `@filelist`.

Command Options

Table 10-1 describes the `mkvdk` command-line options. The options are listed in alphabetic order.

Table 10-1 mkvdk command-line options

Option	Description
`-about`	This option shows information about the collection, such as its description and the date when it was last modified.
`-autodel`	This option deletes the bulk insert file or files when the bulk submission work is finished.
`-backup` *`path`*	This option backs up the collection into the directory in the specified path. **Note:** To ensure that the backup does not include any old data from previous backups, the contents in the specified directory are deleted before the new backup is created.
`-bulk`	This option tells `mkvdk` to interpret *`docSpec`* as a bulk insert file. The option can be used with `-insert`, `-update`, `-delete`, and `-modify`.
`-charmap` *`charset_name`*	Use the character set specified in *`charset_name`* to display the contents of the collection. *`charset_name`* must be the name of one of the supported character sets for the collection's locale. This option is not required if you want to display the collection data using its locale's internal character set. Appendix A of the *Verity Locale Configuration Guide* lists the supported character sets for each Verity locale and indicates which one is the internal character set.
`-collection` *`path`*	Specifies the path of the collection to create or open. This is required to execute `mkvdk`.
`-common`	Specifies the path of the Verity `common` directory. If you do not use this option, the Verity engine looks for the `common` directory in the vicinity of the `mkvdk` executable, and then along the executable search path, determined by your OS path settings. Using this option is not recommended in most situations.
`-create`	Creates a collection in the specified `-collection` directory. It creates the directory structure, determines the index contents and sets up the document-table schema according to the style files used. If the specified collection already exists, `mkvdk` exits rather than overwriting the existing collection.
`-credentials` *`user`*	When accessing a secure repository, pass the login credentials specified in *`user`* to the gateway. Depending on the login requirements, *`user`* can be *`username`*`[:`*`password`*`] [:`*`domain`*`] [:`*`mailbox`*`]`
`-datapath` *`path`*	This option is not supported.

Table 10-1 mkvdk command-line options (continued)

Option	Description
`-datefmt` *format*	This option is used to convert a date field value into Verity's internal data representation, and can be used in conjunction with the `mkvdk` options `-extract` (for the field extraction feature) and `-bulk` (for the bulk submit feature). The named format string identifies to the date parsing routines as to what order dates are written in when the date string only consists of a sequence of numbers (for example, 03/03/96). The K2 engine interprets the numbers in MDY format, where M represents a two-digit month, D represents a two-digit day, and Y represents a two-digit year. The default is `MDY`. ■ `MDY:` month-day- year (US format, the default). ■ `DMY:` day-month-year (European formats). ■ `YMD:` year-month-day (ISO international format). ■ `YDM:` year-day-month (Swedish format). ■ `USA:` US format (the same as `MDY`). ■ `EUR:` European format (the same as `DMY`).
`-debug`	Runs `mkvdk` in debugging mode. Equivalent to specifying `-outlevel 127`.
`-delete`	Marks the specified documents as deleted and makes them unavailable for searches. To actually remove deleted documents from the collection's document table and word indexes, use the `squeeze` keyword.
`-description` *desc*	Sets the collection's description. Enter any alphanumeric text you like, surrounded by quotes (such as "This collection contains electronic mail from ABC Company.")
`-diskcache` *num*	Sets the size of the VDK cache (in KB).
`-errorcodes`	Outputs error messages when `mkvdk` encounters bad document keys.
`-extract`	Extracts field values from documents, using the field extraction rules specified in the `style.tde` file.
`-help`	Displays `mkvdk` syntax options.
`-insert`	This option adds documents to the collection. This is the default option for `mkvdk`. NOTE: Collections created by the Verity Gateway Developer's Kit (GDK) are not supported by `mkvdk`.

Table 10-1 mkvdk command-line options (continued)

Option	Description
`-locale` *`locale_name`*	The name of the locale in which the collection you are accessing was created. *`locale_name`* must be the name of a locale for which you are licensed, and must be one of the Verity locales listed in Appendix A of the *Verity Locale Configuration Guide*. This option is not required if the collection uses the default session locale (usually `uni`).
`-logfile` *`file_name`*	Saves messages in the specified file.
`-loglevel` *`num`*	Indicates which message types to route to the optional log file. Valid values are determined by adding numbers together that correspond to the desired message types. ▪ `Fatal=1` ▪ `Error=2` ▪ `Warning=4` ▪ `Status=8` ▪ `Info=16` ▪ `Verbose=32` ▪ `Debug=64.` Default value = 15 (`Fatal` + `Error` + `Warning` + `Status`).
`-maxfiles` *`num`*	Sets the maximum number of files that `mkvdk` can have open at once. Default = 50.
`-maxmemory`	Sets the maximum amount of memory that `mkvdk` is permitted to use.
`-mode` *`mode`*	Optimizes the collection for one of the following specific modes of usage: ▪ `generic`. (Default) The standard indexing mode. ▪ `fastsearch`. Optimized for fastest return of results at search time. ▪ `bulkload`. Optimized for indexing large numbers of documents through BIFs. ▪ `newsfeedidx`. Optimized for indexing frequent small batches of documents without falling behind. ▪ `newsfeedopt`. Merges partitions created by the `newsfeedidx` mode. ▪ `rdonly`. Disables writing to the collection after indexing. For more information, see "Built-in Indexing Modes" on page 86

Table 10-1 mkvdk command-line options (continued)

Option	Description
`-modify`	Use this option along with the `-bulk` option to update fixed-width fields in an existing collection. Field name/value pairs are specified in a bulk insert file. **Note:** The Verity engine takes the collection off-line to perform the operation, then brings it back up, unless you also specify `-online`.
`-nohousekeep`	This option prevents housekeeping by this instance of `mkvdk`. Housekeeping includes deleting files that are no longer needed. Using this option turns off the service level `VdkServiceType`
`-noindex`	This option prevents indexing by this instance of `mkvdk`. Documents will not be inserted or deleted. Using this option turns off the service level `VdkServiceType_Index`.
`-nolock`	This option turns off file locking. Locking is on by default.
`-nooptimize`	This option prevents optimization by this instance of `mkvdk`. Using this option turns off the service level `VdkServiceType_Optimize`. The service types determine what type of work the Verity engine and its self-administration features will execute on a collection.
`-nosave`	Specifies that a work list, which is generated by `mkvdk` automatically when the `-extract` option is used, will not be saved in the collection directory in a file called `worklist` (in the Verity bulk insert file format). By default, `mkvdk` saves the work list in the `worklist` file.
`-noservice`	This option prevents collection servicing (servicing includes indexing) by this instance of `mkvdk`.
`-nosubmit`	Specifies that a work list, which is generated by `mkvdk` automatically when the `-extract` option is used, will not be submitted to the indexing engine and will be saved in the collection directory in a file called `worklist` (in the Verity bulk insert file format). This option enables `mkvdk` to process field extraction separately from other indexing tasks.
`-numdocs` *num*	This option specifies the number of documents to insert or delete from the bulk insert file or files. Note that if you specify multiple bulk insert files and use the `-numdocs` option, the `-numdocs` setting is applied to all of the files.
`-offset` *num*	This option specifies the offset into a bulk insert file or files. Note that if you specify multiple bulk insert files and use the `-offset` option, the offset is applied to all of the bulk insert files.

Table 10-1 mkvdk command-line options (continued)

Option	Description
`-online`	Use this option, along with `-bulk` and `-modify`, to update fixed-width fields in an existing collection without first bringing the collection offline. Field name/value pairs are specified in a bulk insert file.
`-optimize` *`keyword`*	Performs specific optimizations on the collection, based on the following values for *`keyword`*: ■ `maxclean`. Perform the most comprehensive housekeeping possible and remove out-of-date collection files. Recommended only when you are preparing an isolated collection for publication. ■ `maxmerge`. Perform maximal merging to create partitions that are as large as possible (up to 64000 documents each). ■ `spanword.` Create a spanning word list across all the collection's partitions. ■ `squeeze.` Remove information related to deleted documents from the collection. Squeezing recovers space in the collection and improves search performance. ■ `tuneup.` A convenience keyword; includes `maxmerge` and `spanword`. ■ `readonly.` Make this collection read-only. Appropriate for CD-ROM collections. ■ `publish.` A convenience keyword; includes all of the optimization types except `ngramindex`. Use this keyword to optimize the collection for the best possible retrieval performance, such as for publication to a network on a server or on a CD-ROM. ■ `vdbopt.` Build optimized VDBs (Verity internal databases that contain data from the indexed documents). ■ `ngramindex.` Build an ngram index for the collection. If you build an ngram index, you must also build a spanning word list. **Note:** You can specify more than one optimization keywords by constructing a string separated by hyphens, such as `maxmerge-spanword-squeeze`.

Table 10-1 mkvdk command-line options (continued)

Option	Description
`-outlevel` *num*	Indicates which message types to display to the console. Valid values are determined by adding numbers together that correspond to the desired message types: ■ `Fatal=1` ■ `Error=2` ■ `Warning=4` ■ `Status=8` ■ `Info=16` ■ `Verbose=32` ■ `Debug=64`. Default value = 15 (`Fatal` + `Error` + `Warning` + `Status`).
`-persist`	This option services the collection repeatedly, at default intervals of 30 seconds. Use the `-sleeptime` option to set a different interval.
`-purge`	This option waits the amount specified by the `purgewait` option and then deletes all documents in the collection, but not the collection itself; it leaves the collection directory structure intact. To specify a different wait period, use the `-purgewait` option instead of `-purge`. If you do not use `purgewait`, the default is 600 seconds. Note that `-purge` deletes all documents in a collection, but does not delete the collection itself. To delete a collection, use operating system commands such as the `rm` command on UNIX to remove the collection directory structure and control files.
`-purgeback`	This option, used with the `-purge` option, performs a purge in the background.
`-purgewait` *seconds*	This option specifies to the `-purge` option how many seconds to wait. If you do not specify *sec*, the default is 600.
`-quiet`	This option displays only fatal and error messages to the console. It overrides the `-outlevel` setting.
`-repair`	This option attempts to repair corrupted partitions in a collection. Use this option when `mkvdk` reports the state of a collection as `State = Collection needs repair`.

Table 10-1 mkvdk command-line options (continued)

Option	Description
`-servlev` *`level`*	Service level. The argument *`level`* is a string consisting of keywords separated by hyphens, such as `search-index-optimize`. Values for *`level`*: ▪ `search`. Enable search and retrieval. ▪ `insert`. Enable adding and updating documents. ▪ `optimize`. Enable collection optimization. ▪ `assist`. Enable building of word list. ▪ `housekeep`. Enable housekeeping of unneeded files. ▪ `delete`. Enable document deletion. ▪ `backup`. Enable backup. ▪ `purge`. Enable background purging. ▪ `repair`. Enable collection repair. ▪ `dataprep`. Same as `search-index-optimize-assist-housekeep`. ▪ `index`. Same as insert-delete.
`-sleeptime` *`sec`*	This option specifies the interval between service calls when `mkvdk` is run with the `-persist` option.
`-style` *`dir`*	This option specifies the style directory that contains the style files to use in creating a collection. This option can only be used with the `-create` option. If you do not specify this option when you use `mkvdk` to create a collection, `mkvdk` uses the style files in the `common/style` directory.
`-synch`	This option performs work immediately. If this option is not used, indexing work is done in the background, as time permits.
`-topicset` *`path`*	This option creates a topic index for the collection based on the specified topic set and stores it in the collection directory. This facilitates efficient searches over the collection when using topics.
`-update`	This option adds documents to the collection, replacing all previous information about the specified documents. NOTE: Collections created by the Verity Gateway Developer's Kit (GDK) are not supported by `mkvdk`.
`-vdkhome`	This option specifies the path of the Verity `VDK home` directory. If you do not use this option, the Verity engine looks for the VDK home directory in the directory containing `mkvdk`, and then along the executable search path, determined by your OS path settings.

Table 10-1 mkvdk command-line options (continued)

Option	Description
`-verbose`	Output system messages to the screen, including all message levels up through `verbose`. Equivalent to specifying `-outlevel 63`.
`-wordindex`	This option builds an ngram index for the collection. This has the same effect as using the option combination `-optimize ngramindex`.
`-words`	This option builds a spanning word list that covers all partitions in the collection. This has the same effect as using the option combination `-optimize spanword`. By default, the collection is optimized before the spanning word list is generated. To prevent optimization from occurring, use `-words` in conjunction with `-nooptimize`.

11

Using Bulk Insert Files

This chapter describes the format of bulk insert files and their usage to support collection indexing.

- About Bulk Insert Files
- BIF Format
- Inserting Documents into a Collection
- Deleting Documents from a Collection
- Supporting Continuous Feeds
- Other Uses for the BIF Format

About Bulk Insert Files

A *bulk insert file* (BIF) is a text file that contains name-value pairs of data. The BIF format is used by various Verity processes to import data from external sources or to export data for later use.

The command-line tools `mkvdk`, `vspider`, and `k2spider_cli` can read BIFs to obtain document data to submit to a collection for indexing and for populating collection fields. Also, `vspider` and `k2spider_srv` can output BIFs containing document data for later submission.

These tools can use BIFs to insert (add), update (replace), or delete documents from collections.

This chapter discusses the use of bulk insert files only for manipulating documents in collections. Other Verity components and products use the BIF format for other purposes, as summarized in "Other Uses for the BIF Format" on page 322 and as fully documented in other books.

BIF Format

A bulk insert file is a text file containing a list of records. Each record is a list of *name*:*value* pairs. The first element in each record is the record key. Each record is terminated with <<EOD>>. For example:

```
VdkVgwKey: ../docs/html/agenda.html
VLANG: en
CHARSET:  utf8
TITLE: Conference Agenda
AUTHOR: Nancy Wilshire
<<EOD>>

VdkVgwKey: ../docs/html/sessions.html
VLANG: en
CHARSET:  utf8
TITLE: List of Sessions
AUTHOR: Nancy Wilshire
<<EOD>>
...
```

This example uses the VdkVgwKey field as the record key. It specifies the document to be inserted, update, or deleted from the collection that this BIF will be applied to. The other fields in each record correspond to collection fields that will be populated with the data for each document.

Statements

Every line in a BIF is a statement. Statements can be field definitions, record terminators, or comments.

Comments

Two types of comment statements are allowed: blank lines and lines that begin with a pound sign (#).

Record Terminator

Each record (set fields for a given key) is terminated by the character sequence `<<EOD>>` (end of document).

Field Definitions

Field definitions are the principal content of a BIF. These rules apply:

- Each field definition is a single name-value pair.
- Each definition begins with the field name, which must begin in the first column of the line. (No leading spaces allowed.)
- Field names are case-insensitive.
- Field names can contain any alphanumeric character, as well as the hyphen (`-`) and underscore (_).
- Field names must be terminated with a colon (`:`).
- Field values begin after the colon and continue to the end of the line.
- Leading and trailing blank spaces are stripped from the field value when the bulk insert file is processed.
- Single and double quotes surrounding the field value are stripped when the bulk insert file is read. Blank spaces that occur between the quotes are preserved.
- C-style escape sequences (such as `\t` representing a tab) are allowed in field names. See "Escape Sequences and Special Characters," following this list.
- One field definition in each record must be the record key. For BIFs used to manipulate documents in collections, that field must be `VdkVgwKey`.

Escape Sequences and Special Characters

If you need to use special characters in your bulk insert file, follow these guidelines.

Using Escape Sequences

An escape sequence is a set of characters that, together, represent one character that may have a special significance. Escape sequences can be used in field values. The lead-in character for an escape sequence is the backslash.

An escape sequence may be embedded within a character constant or string literal. In a string literal, the apostrophe may be represented by itself or by its escape sequence. In a character constant, the quotation mark may be represented by itself or by its escape sequence.

You can use escape sequences in BIF fields to specify nonprintable ASCII characters or characters special to the BIF parser, such as backslash and quotes, as indicated Table 11-1.

Table 11-1 Nonprintable ASCII characters

This string:	Is interpreted as:
`\b`	backspace
`\e`	escape
`\f`	formfeed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
\xnn	a character represented by the hexadecimal value nn
\\	backslash
\"	double quote
\'	single quote

Escaping Pathname Separators

If your application is running on a Windows platform and any of your BIF fields contain pathnames with Windows-style separators (backslashes), the backslashes must be escaped (doubled). For example, a Windows pathname such as this:

```
C:\markets\data
```

must be specified in your BIF as:

```
C:\\markets\\data
```

Alternatively, you can use UNIX-style pathname syntax. The Verity engine correctly interprets UNIX-style pathname syntax on all platforms.

Using the Backslash as a Literal Character

If you want a backslash to be interpreted as a literal character, the control file syntax must include two backslashes instead of one backslash. When the Verity engine's control file reader encounters two backslashes in a row, it strips the first one and interprets the second as a literal backslash.

BIF Character Set

The bulk insert file itself should always be in the internal character set for the current locale (the locale of the collection to which it is to be applied).

BIF Size

There is no hard limit on the number of records that can be included in a BIF. However, to avoid runtime memory-usage problems, Verity recommends that you keep individual BIF files to 1K or fewer keys.

BIF Examples

Listing 11-1 file illustrates the BIF syntax, including the use of comments.

Listing 11-1 Example bulk insert file

```
# Every document must have a key
VdkVgwKey:   538592765
# Construct the pieces of the dispatch field that will
# point at the text. The "document" starts at byte offset 1400
# in text/d940116.txt and extends for 945 bytes.
DOC_FN:       text/d940116.txt
DOC_OF:       1400
DOC_SZ:       945
# Embed a hex character into a field value
Company:       F.SNM \x1a Ti Group Plc (U.TI)
# Use quotes to preserve leading or trailing white space
Hot:          " "
# End the document, then start a new one
<<EOD>>
# The second document
VdkVgwKey:    538592766
DOC_FN:       text/d940116.txt
DOC_OF:       2345
DOC_SZ:       690
Ddate:        16-Jan-94 06:05 am
Headline:     *Zurich Noon Gold At 1100 GMT
Hot:          "H"
<<EOD>>
```

Listing 11-2 shows a bulk insert file that contains four documents. The custom fields `TITLE` and `ACTOR` fields will be populated in the collection if they are defined in the collection's `style.ufl` file.

Listing 11-2 Bulk insert file with custom fields

```
TITLE:          Dances with Wolves
ACTOR:          Kevin Costner
VDKVGWKEY:      docs/articl26
<<EOD>>

TITLE:          Ghost
ACTOR:          Patrick Swayze, Demi Moore, Whoopi Goldberg
VDKVGWKEY:      docs/articl28
<<EOD>>

TITLE:          Nightmare Before Christmas
VDKVGWKEY:      docs/articl29
<<EOD>>
```

```
TITLE:          Gone with the Wind
ACTOR:          Vivien Leigh, Clark Gable
VDKVGWKEY:      docs/articl27
<<EOD>>
```

Inserting Documents into a Collection

Using a bulk insert file, you can associate field values with a set of documents and insert those documents into a collection using `mkvdk`. You can either insert new documents or update existing documents in the collection. Take these steps:

1. **Define the document fields in the collection**

   Any field data included in your BIF must apply to fields that are already defined in the collection. You define collection fields in the style files `style.ufl` and `style.sfl`.

   For instructions on how to define collection field using these style files, see "Defining Collection Fields" on page 145.

2. **Create the BIF**

   □ Be sure to create the BIF in the same character set as the collection it will be used to modify.

   □ For each document to be inserted, include its location (file path, URL, etc.) in the BIF's `VdkVgwKey` field.

   (You don't need to define the VdkVgwKey field in the collection; it is already defined in the collection's `style.ddd` file.)

Note `VdkVgwKey` values must be unique across all collections that will be searched in any one application. Non-unique document keys cause application errors.

   □ Add a field definition line for each of the fields whose data you are importing with the BIF. Not all document records in the BIF have to include all fields being imported; in each document record, add field-definition lines only for fields that have data for that document.

   □ Be sure to end each document record with `<<EOD>>` on its own line.

3. Use mkvdk to submit the BIF

- □ To submit the contents of the BIF to a collection, use the `-bulk` option of `mkvdk`, as in either of these commands:

```
mkvdk -collection collname -bulk -insert filespec
mkvdk -collection collname -bulk -update filespec
```

  where *collname* is the pathname (full or relative) of the collection to be updated, and *filespec* is the pathname of the BIF.

  Use the `-insert` option if the BIF specifies only new documents to insert in the collection. Use the `-update` option if the BIF specifies updates to documents already in the collection.

  Note: It is safest to always use the `-update` option. If a document in the BIF is new to the collection, it is properly inserted even if the `-update` option is specified. But if you specify `-insert` and a document in the BIF is already in the collection, an error can occur.

- □ If you need to start processing the BIF from a position other than the beginning of the file, use the `-offset` option of `mkvdk`, as follows:

```
mkvdk -collection collname -offset num -bulk filespec
```

  where *num* is the offset (in bytes from the beginning of the file) at which to start.

- □ If you want the BIF to be automatically deleted when processing is complete, use the `-autodel` option of `mkvdk`, as follows:

```
mkvdk -collection collname -autodel -bulk filespec
```

- □ If you need to limit the number of documents processed, use the `-numdocs` option of `mkvdk`, as illustrated in the following example:

```
mkvdk -collection collname -numdocs num -bulk filespec
```

  where *num* is the maximum number of documents to process from the BIF.

Deleting Documents from a Collection

You can also use a bulk insert file to delete documents from a collection. To delete all references to a set of documents in a collection, use `mkvdk` as follows:

```
mkvdk -collection collname -delete -bulk filespec
```

where *collname* is the full or relative pathname of the collection to be accessed, and *filespec* is the pathname of the BIF specifying the documents to delete. The BIF can be either a standard bulk insert file (see "BIF Format" on page 314) that, for example, may have been used earlier to insert information into the collection, or it can have the simpler format of one document key per line, like this:

```
VdkVgwKey: trn_log.c
VdkVgwKey: trn_misc.c
VdkVgwKey: trn_pars.c
VdkVgwKey: trn_work.c
VdkVgwKey: trn_wrte.c
```

Deleting files from a collection in this way does not by itself recover disk space; it simply marks the specified documents as deleted and makes them unavailable for searching.

To physically remove information related to deleted documents from the collection's tables and indexes, thereby recovering disk space, you must use `mkvdk` with the `squeeze` keyword. For more information, see "Using mkvdk" on page 279.

Supporting Continuous Feeds

If you are writing an application that reads data from a continuous feed such as the Dow Jones or Reuters news service, you may want your application to continuously append information to a single bulk insert file, rather than starting a new BIF every time the collection is updated. For example, the application could append received documents to a single BIF for an entire day, but use that BIF to add documents to the collection much more frequently, perhaps every 30 seconds or 1 minute.

If you append continuously to a single BIF under these circumstances, your collection-building application needs to know the offset into the BIF at which to start each time it updates the collection, and how many documents it can add to the collection. With `mkvdk`, you specify these values using the `-offset` and `-numdocs` command-line options as mentioned in Step3, "Use mkvdk to submit the BIF" on page 320.

Using Verity Developer's Kit, these options are implemented using the `offset` and `numdocs` numbers in the `VdkCollectionSubmitArgRec` structure. When used, these options permit the collection-building application to append and process documents at the same time.

Other Uses for the BIF Format

See the referenced documents for more information on how bulk insert files are used by other Verity tools, products, and APIs.

Language Identification

The language-identification command-line tool optionally outputs bulk insert files that include a language ID field for each analyzed document. The BIFs can then be used to add documents to the appropriate collections. For example:

```
VDKVGWKEY:/data/french/francais.txt
VLANG: fr
CHARSET:  1252
<<EOD>>

VDKVGWKEY:/data/swiss/francais_deutsch.txt
VLANG: fr
CHARSET: 1252
<<EOD>>
```

This BIF shows that two documents were identified as predominantly French.

For more information on `langID` and its use of BIFs, see the *Verity Locale Configuration Guide.*

Categories

The Verity Intelligent Classifier uses bulk insert files to add or remove documents from categories in a taxonomy. For example:

```
  VdkVgwKey:        docs/demo_templates/searchresults.html
  RemoveCategory:   reports
  <<EOD>>

  VdkVgwKey:        docs/products/keyview/index.html
  AssignCategory:   annual 10001 reports 10002
  <<EOD>>
```

This BIF removes the document `searchresults.html` from the `reports` category, and adds the document `index.html` to two categories: `annual` and `reports`. (The numeric values represent the ranking score to assign to the document in each of the two categories.)

For more information on Intelligent Classifier's use of BIF files, see the *Verity Intelligent Classification Guide.*

Profile Nets

The `mkprf` command-line tool uses bulk insert files to add or remove queries from profile nets. The `queryText` field is the only required field. For example:

```
queryText: Water
<<EOD>>
```

In this case, any document containing the word *Water* generates a query hit when evaluated against the profile net created with this BIF.

For more information on `mkprf` and profile nets and their use of BIFs, see the *Verity K2 Profiler Programming Guide.*

Parametric Indexes

With the Parametric API, you can use bulk insert files to add or remove documents from categories in a parametric index. You specify the documents to add or remove using the document key, the document ID, or a query. For example:

```
QueryQuestion:   installation
AssignCategory:  field
LowerThreshold:  4000
UpperThreshold:  9000
<<EOD>>

QueryQuestion:   News
RemoveCategory:  news
<<EOD>>
```

This BIF assigns any document that matches the query "installation", with a relevance score between 4000 and 9000, to the `field` category. It also removes any document that matches the query "News" from the `news` category.

For more information on using BIFs with parametric indexes, see the *Verity K2 Profiler Programming Guide.*

12

Using Other Collection Tools

This appendix describes useful tools for troubleshooting and maintaining collections. This appendix includes these topics:

- About the Collection Tools
- didump
- browse
- merge
- rcvdk

About the Collection Tools

The collection-management tools described here let you view some of the inner structure of collections, search them from a command line, and perform certain optimizations on them.

Note The collection-management tool `mkvdk` is described in "Using mkvdk" on page 279.

Location

These command-line tools are all in the directory `platformDir\bin`, where *`platformDir`* refers to the full path to the platform-specific directory for the K2 or VDK installation (for example, `usr/verity/k2/_ssol26` on Solaris).

Specifying Locale and Character Set

The `-charmap` and `-locale` options are available for `rcvdk`, `didump`, and `browse` to allow you to access and display information in a variety of languages and character sets. Here is how they are used:

Option	Description
`-charmap` *`name`*	Specifies the character set used by information (such as a BIF) that the tool passes to the Verity engine, and the character set that the Verity engine should use when passing information (such as index contents) back to the tool. *`name`* should be a character set that your system can display properly. It should also be a character set that is supported by the locale, if any, specified in the `-locale` option. For information on supported character sets, see Appendix A of the *Verity Locale Configuration Guide*. **Note:** This value is *not* related to the character set of the collection or of documents being indexed. It only specifies the character set used for communication between the tool and the Verity engine.
`-locale` *`name`*	Specifies the session locale (or language). When creating a collection, the session locale is the locale that will be assigned to the collection being created. See Appendix A of the *Verity Locale Configuration Guide* for a list of the supported locales. If you do not specify a locale, the default session locale is used. The initial value of the default is `uni`. For information on changing the default session locale, see the using locales chapter of the *Verity Locale Configuration Guide*.

didump

The `didump` command-line tool produces a word index for a collection, one partition at a time. It also produces a list of zones, when zones are used.

Using `didump`, you can view the word index components by partition. The word index consists of a list of all words indexed by the Verity engine. The zone index is a list of all zones found by the engine. The zone attribute index is a list of the zone attributes found by the engine.

`didump` can be found in the Verity bin directory. In a typical installation, the path is:

```
platformDir/bin/didump[.exe]
```

where *`platformDir`* is the pathname of the platform-specific directory beneath the Verity installation directory (for example, `usr/verity/k2_ssol26` for Solaris), and `.exe` is the tool's file extension (Windows only).

didump Syntax

This is the syntax for `didump`:

```
didump [-verbose] [-words] [-zones] [-attributes] [-auxdata]
[-nouns] [-nounphrases] [-zonecontent] [-stemdex] [-soundex]
[-rangedex] [-pattern word_pattern] partition_name
```

`-verbose`	Display debugging information.
`-words`	Display the collection's word index.
`-zones`	Display the collection's zone index.
`-attributes`	Display the collection's zone attributes index.
`-auxdata`	Add auxiliary data (highlight location data and qualify instance data) to the word index display
`-nouns`	Display the collection's extracted nouns.
`-nounphrases`	Display the collection's noun phrases.
`-zonecontent`	Display the collection's zone-content index.
`-stemdex`	Display the collection's stem index.

`-soundex` Display the collection's Soundex index.

`-rangedex` Display the collection's numeric/date indexes (`Numdex`, `Datedex`, and `XDatedex`).

`-pattern` Display occurrences of *`word_pattern`* in the word index. *`word_pattern`* can be a word or a regular expression.

`partition_name` The path to the collection partition file containing the indexes to display.

Viewing the Word Index

You can view the contents of the word list for a partition by using the `didump` command-line tool with the `-words` flag. The command-line syntax must include the `-words` flag and a pathname to a partition file, like this:

```
didump -words /z/collref/html/parts/00000003.did
```

The display provides an alphabetical listing of the words in the word index, as shown here.

```
didump - Verity, Inc. Version 4.0.1 (_nti40, Jun 7 2001)

Text                           Size    Doc     Word
A                              10      3       4
a                              34      5       24
abbreviations                  4       1       1
about                          4       1       1
acronym                        5       1       2
acronyms                       4       1       1
actual                         4       1       1
administrator                  3       1       1
advance                        3       1       1
all                            8       2       3
also                           9       2       4
Always                         4       1       1
always                         9       2       3
ampersand                      4       1       1
...
```

The columns in the display indicate:

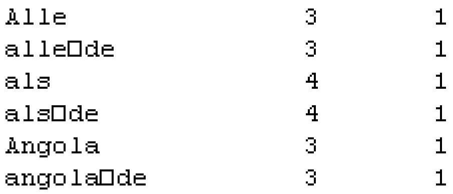
- `Size`. the number of bytes used by the Verity engine to store information about the word
- `Doc`. the number of unique documents in which the word appears
- `Word`. the total number of occurrences of a word for the partition

To view the occurrences of a specific word or pattern, enter a command using the `-pattern` option, as in the following example:

```
didump -pattern acronym 00000003.did
```

The `didump` command-line tool will display information about the number of occurrences of the word “acronym.” You can display the individual occurrences of a word using the verbose (`-verbose`) option.

Note In viewing the word list for a collection created in the multilanguage locale, you may see language-specific word stems formatted as the stem followed by an unprintable (or unintelligible) character, followed in turn by the two-character language code, as in this example for a collection containing German documents:



```
Alle              3       1
alle□de           3       1
als               4       1
als□de            4       1
Angola            3       1
angola□de         3       1
```

Viewing the Zone List

The zone list contains a list of the zones identified by the zone filter. The zones listed can be searched using the Verity `IN` operator in a query. To view the contents of zone list, use `didump` with the `-zones` flag plus the pathname to a partition, like this:

```
didump -zones /z/collref/html/parts/00000003.did
```

The partition above is for a collection containing a document in HTML format. The Verity universal filter invoked the HTML filter by default and indexed the documents using these zones.

```
didump - Verity, Inc. Version 4.0.1 (_ssol26, Jun 07 2001)
```

```
ZoneName                        Fmt     Size    Doc     Regions
A                               Wct     10239   85      5016
ADDRESS                         Array   34      1       1
BODY                            Array   197     85      85
CAPTION                         Wct     298     31      85
CODE                            Wct     3868    66      1829
H1                              Array   80      83      83
H2                              Wct     646     53      212
H3                              Wct     517     49      171
H4                              Wct     128     8       47
HEAD                            Array   70      85      85
HTML                            Array   165     85      85
TITLE                           Array   70      85      85
```

The columns in the display indicate:

- `Fmt`. the internal data format used to store the zone information
- `Size`. the number of bytes used by the Verity engine to store information about the zone
- `Doc`. the number of unique documents in which the zone appears
- `Region`. the total number of instances of a zone for the partition

For complete information about the how zones are defined, see "Defining Document Zones" on page 179.

Viewing the Zone Attribute List

The zone attribute list contains a list of the HTML attributes for the zones identified by the HTML zone filter. The zone attributes listed can be searched using the Verity `IN` operator together with the `WHEN` operator in a query. To view the contents of the zone attributes list, use `didump` with the `-attributes` flag plus the pathname to a partition, like this:

```
didump -attributes /z/collbldg/html/parts/00000003.did
```

The partition above is for a collection containing the <book_title>Verity Collection Reference Guide in HTML format.

```
didump - Verity, Inc. Version 4.0.1 (_ssol26, Jun 7 2001)

Text                            Size    Doc     Word
href 01_cbg.htm                 10      2       4
href 01_cbg.htm#282870          3       1       1
href 01_cbg.htm#282872          6       2       2
href 01_cbg1.htm                8       2       3
href 01_cbg1.htm#286513         7       2       2
href 01_cbg1.htm#286520         3       1       1
...
```

The columns in the display indicate:

- `Size`. the number of bytes used by the Verity engine to store information about the zone attribute
- `Doc`. the number of unique documents in which the zone attribute appears
- **`Word`**. the total number of occurrences of a zone attribute for the partition

browse

The `browse` command-line tool lists the field names and values stored in a collection's document table, one partition at a time. `browse` can be used to browse the document table (*`filename`*`.ddd`) in the `/parts` directory of a collection. Field names and values are displayed.

A document table is built for each partition in a collection. The document table is used for field searching and for sorting search results. The fields within the document table are defined by the following collection style files:

- `style.ddd`, defines fields used internally by the Verity engine, identified by an initial underscore character (_)
- `style.sfl`, defines standard fields (many of which are commented out to limit the size of the document table)
- `style.ufl`, defines custom fields that are not included in `style.sfl`

The value of each field can be filled in from source documents or can be provided explicitly. If a field is blank, it has not been populated.

`browse` can be found in the Verity bin directory. In a typical installation, the path is:

/*verity*/*prdname*/k2/*_platform*/bin/browse where *verity/prdname* represents the user-definable portion of the Verity installation directory name, and *_platform* represents the platform name (like `_nti40` for Windows NT v4.0).

Displaying Fields in a Document Table

You should be in the `/parts` directory of a collection when you run the `browse` tool. This saves you the effort of having to type the full path to the document table (*filename*`.ddd`).

To display all the document fields in a document table, follow these steps.

1. Use the following command to start the `browse` tool and display the set of menu options:

```
D:\VERITY\colltest\parts\>browse filename.ddd
```

   where *filename* is an incrementing number, such as `00000002`. The path provided above is for illustrative purposes. The paths to your collections will be different.

   The system displays the following menu of options available for the `browse` tool.

```
D:\VERITY\colltest\parts>browse 00000003.ddd
BROWSE OPTIONS
  ?) help
  q) quit
  c) Number of entries in field
  _) Toggle viewing fields beginning with '_'
  v) Toggle viewing selected fields
 ##) Display all fields in specified record number
Dispatch/Compound field options:
  n) No dispatch
  d) Dispatch
  s) Dispatch as stream
Action (? for help):
```

2. Optionally, type the underscore character (_), and then press Enter to suppress the display of the internal fields.

```
Action (? for help):_
View non ‘_’ fields
Action (? for help):
```

Note At any time, you can type the underscore character (_) and press Enter to toggle between displaying all fields, only internal fields, or only non-internal fields. After you type the underscore character (_) and press Enter, the `browse` tool will display a status message regarding the types of fields that are to be displayed. Then you will see the Action prompt.

3. At the Action prompt, press Enter to display the fields for the first document record.

```
Action (? for help): <Enter>
```

The following partial display of the results of the `browse` command includes internal fields, used by the Verity search engine. An internal field name starts with an underscore (_) character.

```
Record number: 0

50 Created           FIX-date (  4) = 12-Jan-2001 01:52:27 pm
51 Modified          FIX-date (  4) = 24-Apr-2001 02:40:26 pm
52 Size              FIX-unsg (  4) = 5381
53 DOC_OF            FIX-unsg (  4) = 0
54 DOC_SZ            FIX-unsg (  4) = 4294967295
55 DOC_FN_OF         FIX-unsg (  4) = 436

Action (? for help):
```

Note Depending on the size of your command-prompt window (on Windows), the list of fields may scroll past the visible portion of the command prompt. You may want to increase the size of the window so that you can see more of the fields at one time. Or you may want to increase the display buffer size used by the command prompt, so that you can scroll back to view some of the fields.

4. At the Action prompt, press Enter again to display the fields for the next document record.

```
Action (? for help): <Enter>
Record number: 1

50 Created          FIX-date (  4) = 12-Jan-2001 01:52:27 pm
51 Modified         FIX-date (  4) = 24-Apr-2001 02:40:26 pm
52 Size             FIX-unsg (  4) = 5381
53 DOC_OF           FIX-unsg (  4) = 0
54 DOC_SZ           FIX-unsg (  4) = 4294967295
55 DOC_FN_OF        FIX-unsg (  4) = 436

Action (? for help):
```

Note If the collection was created in a non-default (non-`uni`) locale, or if any of the fields have non-ASCII characters, use the `-locale` option before viewing the fields. For example:

```
Action (? for help): -locale uni
Action (? for help): <Enter>
Record number: 1

50 Created          FIX-date (  4) = 12-Jan-2001
01:52:27 pm
51 Modified         FIX-date (  4) = 24-Apr-2001
02:40:26 pm
52 Size             FIX-unsg (  4) = 5381
53 DOC_OF           FIX-unsg (  4) = 0
54 DOC_SZ           FIX-unsg (  4) = 4294967295
55 DOC_FN_OF        FIX-unsg (  4) = 436

Action (? for help):
```

merge

The `merge` command-line tool allows you to merge and split collections. `merge` lets you combine multiple collections that have the same schema (that is, the same set of style files). This is useful for merging smaller collections built from different sources into one, large collection. Also, you can use the `merge` command-line tool to break up the collection into smaller collections of a roughly uniform size.

It is important to note that collections can be merged only if they have identical schemas. Collections can be merged if they have exactly the same set of style files (and style file entries).

Breaking up a large collection helps to optimize search performance, because it allows many applications to perform multiple concurrent search requests over the different collections. After breaking up a large collection, you can also discard older collections to reclaim limited disk storage space.

`merge` can be found in the Verity `bin` directory. In a typical installation, the path is:

/*verity*/*prdname*/k2/*_platform*/bin/merge

where *verity/prdname* represents the user-definable portion of the Verity installation directory name, and *_platform* represents the platform name (like `_nti40` for Windows NT v4.0).

To obtain help for the `merge` command-line tool, enter the following command:

```
merge -help
```

Note After running the `merge` command-line tool, you must optimize the collection, using the `mkvdk -optimize` option.

Merging Collections

The following is the syntax for using the `merge` command-line tool to merge multiple collections into a single collection:

merge *newCollection srcCollection1 srcCollection2*
[*srcCollectionN*]

The command-line tool reads *srcCollection1*, *srcCollection2* and so on and merges them into a single collection with the directory name given for *newCollection* If the directory name given for *newCollection* doesn't exist, then it is created.

Splitting Collections

The following is the syntax for using the `merge` command-line tool to split a single large collection into smaller collections:

```
merge -split srcCollection newCollection1 newCollection2
[-number]
```

The command-line tool reads *`srcCollection`* and splits it in roughly equal-sized pieces, using the file names given for *`newCollection1`* and so on.

If you want to split a very large collection into a large number of new collections, you can use the following option instead of explicitly naming each new collection:

```
merge -split -number N srcCollection newCollection
```

The command-line tool reads the collection identified by *`srcCollection`* and splits it into the number of segments (*`N`*) specified by the `-number` option. The name of the first new collection is generated by appending the first two letters in the alphabet (`aa)` to the directory name given for *`newCollection`*. Each subsequent file name is generated by incrementing one of the appended letters (up to `zz)` for a maximum of 676 partitions. For example, if the value of `-number` is `3`, and the value of *`newCollection`* is *`Collection1`*, the collections are named, *`Collection1aa`*, *`Collection1ab`*, and *`Collection1ac`*.

Note The maximum length of the directory name given for *`newCollection`* is 2 characters less than the length allowed by the file system.

rcvdk

The `rcvdk` command-line tool is a simple command-line search client that allows you to search over a collection and list the collection fields. `rcvdk` is used for searching Verity collections and displaying documents. The `rcvdk` name is an acronym for Retrieval Client Verity Developer Kit.

Using `rcvdk`, you can check the contents of a collection from the command line. `rcvdk` allows you to write a variety of queries, using words and phrases separated by commas and/or Verity query language. A viewing option allows you to see document contents and highlights in a simple text display.

`rcvdk` can be found in the Verity `bin` directory. In a typical installation, the path is:

```
/verity/prdname/k2/_platform/bin/rcvdk
```

where *verity/prdname* represents the user-definable portion of the Verity installation directory name, and *_platform* represents the platform name (like `_nti40` for Windows).

Note `rcvdk` includes the following software limitation. If the first docID that `rcvdk` encounters when trying to read documents is invalid, the gateway returns "no access" to all documents in the collection.

Starting rcvdk

To start `rcvdk` on most systems, type the path and executable name. On a Windows machine, use a DOS command window or the Run dialog. The examples shown here assume you have set your PATH variable set, so you just need to enter "`rcvdk`" to run it.

Specifying a Default Session Language

If you are using `rcvdk` to access a collection created with the multilanguage (`uni`) locale, you can specify a default language to use for the session. The default session language is the language whose rules are to be used for searching when the search query does not include an explicit language specification (using the VQL `<lang/`*`id`*`>` modifier).

To specify a language, use the `-locale` option on the command line when you launch rcvdk, with a locale specification in the form of `uni/`*`id`*. For example:

```
rcvdk -locale -uni/ja
```

(Available language specifications are listed in Appendix A of the *Verity Locale Configuration Guide*.)

Attaching to a Collection on Launch

You make a collection accessible to `rcvdk` by attaching it. You can do that after launch with an `rcvdk` command (see "Starting rcvdk" on page 337), or you can specify the path to the collection (relative to the current working directory) on the command line when you launch rcvdk. For example:

```
rcvdk verity\k2_61\data\colls\verity_doccoll
```

Viewing Available Commands

When you start `rcvdk` with no arguments, you get this message followed by a `RC>` prompt:

```
Type 'help' for a list of commands.
RC>
```

The `help` (or `?`) command produces the following list of available commands:

```
RC> help
Available commands (note that some have 1-character shortcuts, like "s" for "search"):
search|s  <query>         Search collection for <query>
search|s                  Perform null search (returns all documents in collection)
results|r  <number>       Display search results starting at Nth result
results|r                 Display search results starting at current result (default=1)
clusters|c                Display clustered search results
view|v  <number>          View Nth document in results list
view|v                    View document at current point in results list (default=1)
summarize|z  <number>     Get dynamic summary of Nth doc in results list
summarize|z               Get dynamic summary of current doc in list (default=1)
attach|a  <coll_path>...  Attach one or more collections
attach|a                  List currently attached (enabled and disabled) collections
detach|d  <coll_path>...  Detach one or more collections
detach|d                  List currently attached (enabled and disabled) collections
quit|q                    Quit rcvdk or leave document display
about                     Display VDK 'About' info
help|?                    Display help page
expert|x                  Toggle (on/off) expert mode
user|u                    Set user: username|:password |:domain |:mailbox
RC>
```

Additional commands are available in expert mode (type **`expert`** or **`x`**).

At any time, you can enter **`q`** at the `RC>` prompt to quit the application.

Attaching Collections

To search a collection, you first must attach it using the `attach` (or `a`) command. This command must include the pathname to a collection directory as an argument. After you press return, `rcvdk` reports whether the attach command was successful.

```
RC>a /z/colls/mycoll1
Attaching to collection: /z/colls/mycoll1
Successfully attached to 1 collection.
RC>
```

Attaching Multiple Collections

rcvdk allows you to attach more than one collection. After attaching mycoll1, do the following to attach collection mycoll2.

```
RC>a /z/colls/mycoll2
Attaching to collection: /z/colls/mycoll2
Successfully attached to 1 collection.
RC>
```

To view a list of the attached collections, use the attach command without arguments:

```
RC> a
Attached collections [2]:
 Enabled: /z/colls/mycoll1
 Enabled: /z/colls/mycoll2
RC>
```

You can attach multiple collections at once by following the a command with multiple collection paths.

Disabling and Enabling Attached Collections

You can choose to search only a subset of the currently attached collections by disabling one or more of them. You use the disable command to change the enabled/disabled state of one or more attached collections.

The disable command is available only in expert mode, activated by executing the expert (or x) command. For example, to disable both mycoll1 and mycoll2, type these commands:

```
RC> x
Expert mode enabled
RC> disable mycoll1 mycoll2
/z/colls/mycoll1 is now disabled for searching.
/z/colls/mycoll2 is now disabled for searching.
RC>
```

To re-enable one or more disabled collections, again use the disable command:

```
RC> disable /z/colls/mycoll1
/z/colls/mycoll1 is now enabled for searching.
RC>
```

At any time, you can view the enabled/disabled states of all attached collections by executing the attach or detach command without arguments:

```
RC> attach
Attached collections [2]:
 Enabled: /z/colls/mycoll1
Disabled: /z/colls/mycoll2
RC>
```

Detaching From Collections

Attached collections remain attached until you detach from one or more of them using the `detach` (or `d`) command:

```
RC> detach /z/colls/mycoll2
Detaching from collection: /z/colls/mycoll2
Successfully detached from 1 collection.
RC>
```

Basic Searching

To retrieve all documents in the attached collection(s), perform a null search: just enter `s` or `search`. After you press return, a search update message is produced, like the one shown here.

```
RC>s
Search update: finished (100%).  Retrieved: 85(85)/85.
RC>
```

In the message returned for the null search, the notation indicates that 85 of the total 85 documents in the collection were retrieved. If you specify a query, like “universal filter”, a subset of the total documents in the collection will be retrieved.

```
RC>s universal filter
Search update: finished (100%).  Retrieved: 18(18)/85.
RC>
```

In the message returned for the search above, `rcvdk` indicates that 18 documents matched the query.

More elaborate queries using Verity query language can be used, like this:

```
RC>s universal filter <OR> filter
```

You can use the expert-mode `time` (or `t`) command to turn on or off a display of the amount of time a search takes:

```
RC> time
Display of elapsed time after Search enabled
RC> s universal filter
Search update: finished (100%).  Retrieved: 18(18)/85.
Elapsed time is 16 milliseconds
RC>
```

Viewing the Results List

After you have attached to a collection and issued a search command successfully, you can view the results list and look at any of the retrieved documents. Use the `results` (or `r`) command with the following options to view results:

Option	Description
`r`	Displays the results list, starting with the first document. A maximum of 24 documents is displayed.
`r` *`N`*	Displays the results list, starting with the *N*th document. A maximum of 24 documents is displayed.

The results list for the search **`s universal filter`** is shown below. For each document, these fields are displayed by default: `Number`, `Score`, and `VdkVgwKey`.

```
RC> r
Retrieved: 15(15)/85
Number  SCORE VdkVgwKey
1:    1.00  d:\verity\k2\docs\doc\fundmntl\08_k23.htm
2:    0.97  d:\verity\k2\docs\doc\fundmntl\11_k22.htm
3:    0.97  d:\verity\k2\docs\doc\fundmntl\08_k27.htm
4:    0.97  d:\verity\k2\docs\doc\fundmntl\08_k21.htm
5:    0.95  d:\verity\k2\docs\doc\fundmntl\k2toc.htm
6:    0.95  d:\verity\k2\docs\doc\fundmntl\08_k24.htm
7:    0.93  d:\verity\k2\docs\doc\fundmntl\k2ix.htm
8:    0.92  d:\verity\k2\docs\doc\fundmntl\08_k26.htm
9:    0.90  d:\verity\k2\docs\doc\fundmntl\08_k2.htm
10:   0.90  d:\verity\k2\docs\doc\fundmntl\04_k21.htm
11:   0.90  d:\verity\k2\docs\doc\fundmntl\01_k21.htm
12:   0.87  d:\verity\k2\docs\doc\fundmntl\f_k2.htm
13:   0.87  d:\verity\k2\docs\doc\fundmntl\08_k22.htm
```

```
14:    0.84  d:\verity\k2\docs\doc\fundmntl\06_k21.htm
15:    0.80  d:\verity\k2\docs\doc\fundmntl\part4.htm
RC>
```

The default fields have these properties:

Field Name	Description
Number	The rank of the document in the results list. The document with the highest score is ranked number 1.
Score	The score assigned to each retrieved document, based on its relevance to the query. For a NULL query, no scores are assigned, so the Score column in the results list is blank.
VdkVgwKey	The document key used by the Verity engine to refer to the document.

Specifying Fields to Display

You can tell `rcvdk` to display certain fields other than the defaults in the results list. To do so, use the `fields` command, which is available in the expert mode. To activate expert mode, type **`x`** or **`expert`** at the `RC>` prompt, then press Enter.

The parameters to the `fields` command are the field name and display length (in characters) for each field to be displayed. When used, the `fields` command overrides the default fields `Score` and `VdkVgwKey`. (`Number`, the first column in the display, shows the rank order of the results and is not overridden.)

Fields for the results list are returned by the search engine, so if you have done a search, then go to expert mode to use the `fields` command, you must run the search again in order to see the results list with the fields you requested.

All fields in a column are blank if the field is not defined for the collection in the document table (in `style.ddd`, `style.sfl`, or `style.ufl`). A field in an individual document's row will be blank if the field was not populated for that document.

In this example, only the `title` field is specified for display:

```
RC> expert
Expert mode enabled
RC> fields title 40
RC> s universal filter
Search update: finished (100%).  Retrieved: 15(15)/85.
RC> r
Retrieved: 15(15)/85
Number  title
1:      Verity Portal Fundamentals
```

```
2:       Document Types
3:       Indexing File Systems
4:       Indexing Web Servers
5:       Table of Contents
6:       Indexing Exchange Servers
7:       Index
8:       Verifying Collections
9:       Maintaining Collections
10:      Tuning Unstructured Search
11:      Field and Collection Schema Definition
12:      Starting Servers from the Command Line
13:      Using the K2 System Console
14:      Parametric Search
15:      Personalization Engine
RC>
```

Multiple fields can be specified with the `fields` command. The order of fields in the results display corresponds to the order you specify them on the `fields` command line.

After specifying new fields, remember to re-run the search before you display the results.

```
RC> fields score 5 title 40
RC> s universal filter
Search update: finished (100%).  Retrieved: 15(15)/85.
RC>
```

Sorting the Results

You can re-order the displayed results of a search by using the `sort` command, available in expert mode.

```
RC> sort title asc
RC> s universal filter
RC> r
Retrieved: 15(15)/85
Number  title
1:       Document Types
2:       Field and Collection Schema Definition
3:       Index
4:       Indexing Exchange Servers
5:       Indexing File Systems
6:       Indexing Web Servers
7:       Maintaining Collections
8:       Parametric Search
```

```
9:      Personalization Engine
10:     Starting Servers from the Command Line
11:     Table of Contents
12:     Tuning Unstructured Search
13:     Using the K2 System Console
14:     Verifying Collections
15:     Verity Portal Fundamentals
RC>
```

Clustering the Results

You can cluster the results into groups of similar documents by using the command `clusters` (or `c`). The number of groups and the criteria for membership are determined by `rcvdk`, based on feature analysis.

Changing Score Precision

You can change the precision of the displayed relevance score (`SCORE` field) of the results by using the command `precision`, available in expert mode. To change the precision, you include an argument specifying the number of bits of precision: 1, 4, 8, 16, or 32.

If precision is 8, results might look like this:

```
1:     1.00  d:\verity\k2\docs\doc\fundmntl\08_k23.htm
2:     0.97  d:\verity\k2\docs\doc\fundmntl\11_k22.htm
3:     0.97  d:\verity\k2\docs\doc\fundmntl\08_k27.htm
...
```

If precision is 16, the same results might look like this:

```
1:     0.9998  d:\verity\k2\docs\doc\fundmntl\08_k23.htm
2:     0.9732  d:\verity\k2\docs\doc\fundmntl\11_k22.htm
3:     0.9683  d:\verity\k2\docs\doc\fundmntl\08_k27.htm
...
```

Displaying Passage-Based Document Summaries

The static summary field (`VDKSUMMARY`) is one of the fields that you can always specify for display by using the `fields` command. In addition, you can also use the `pbs` command (available in expert mode) to specify that passage-based summaries be generated for documents in the results list.

You configure passage-based summaries by following the `pbs` command with two arguments: the first specifies the number of passages to include in a summary, and the second specifies the size (in bytes) of each passage. If you execute `pbs` without any arguments, `rcvdk` displays the current `pbs` settings.

For example, to specify that each summary should consist of 4 passages of 100 bytes each, do this:

```
RC> x
Expert mode enabled
RC> pbs 4 100
RC> pbs
Number of passages: 4
Passage size: 100
RC>
```

Then, to actually view the passage-based summaries, you must use the `fields` command to make the `VDKPBSUMMARY` field visible in the search results.

```
RC> fields score 10 VdkVgwKey 50 VDKPBSUMMARY 400
RC> fields
Results fields:
 10  score
 50  VdkVgwKey
400  VDKPBSUMMARY
RC> s universal filter
Search update: finished (100%).  Retrieved: 45(45)/4409.
RC> r
Retrieved: 45(45)/4409
Number  score      VdkVgwKey                                  VDKPBSUMMARY


1:      0.9352     ../docs/html/VDKProgramming/filters6.html         ... Filters >   Using
the Universal Filter   The universal filter is a document filter that ... filters (the
current zone filter). The universal filter and its helper filters are used to ... universal
filter, see the Verity Collection Reference.   The advantage of the universal filter ...
into the same collection.   The universal filter is can be configured. It has a
configuration ...

2:      0.9203     ../docs/html/CollectionReference/filters14.html    ... The Universal
Filter   This section provides an overview of the universal filter and its ...
implementation.   The universal filter is a document filter that produces indexable (or ...
zone filter).   The advantage of the universal filter is that it removes the need to
specify ... Subtopics: Invoking the Universal Filter How the Universal Filter Works
Character Set ...

3:      0.9115     ../docs/html/CollectionReference/filters6.html     ... Document >
Universal Filter Functionality   The universal filter can automatically detect ... The
Verity engine implements the universal filter by default based on configuration settings
```

```
... The universal filter and its configuration are described in The Universal Filter. For
... the document types recognized by the universal filter, see Supported Document Formats.
See ...

...
```

Generating Dynamic Document Summaries

You can also use `rcvdk` to generate dynamic summaries, which are similar to the VDK static summaries stored in the collection field `VDKSUMMARY`, except that they are generated on the fly.

Unlike for static or passage-based summaries, you use the `summarize` (or `z`) command to specify (by number in the search results) the individual document you want a dynamic summary of. For example, to view the results list and then generate and display a dynamic summary of the 4th document in the list, do this:

```
RC> r
Retrieved: 15(15)/85
Number  SCORE VdkVgwKey
1:    1.00  d:\verity\k2\docs\doc\fundmntl\08_k23.htm
2:    0.97  d:\verity\k2\docs\doc\fundmntl\11_k22.htm
3:    0.97  d:\verity\k2\docs\doc\fundmntl\08_k27.htm
4:    0.97  d:\verity\k2\docs\doc\fundmntl\08_k21.htm
5:    0.95  d:\verity\k2\docs\doc\fundmntl\k2toc.htm
6:    0.95  d:\verity\k2\docs\doc\fundmntl\08_k24.htm
7:    0.93  d:\verity\k2\docs\doc\fundmntl\k2ix.htm
8:    0.92  d:\verity\k2\docs\doc\fundmntl\08_k26.htm
9:    0.90  d:\verity\k2\docs\doc\fundmntl\08_k2.htm
10:   0.90  d:\verity\k2\docs\doc\fundmntl\04_k21.htm
11:   0.90  d:\verity\k2\docs\doc\fundmntl\01_k21.htm
12:   0.87  d:\verity\k2\docs\doc\fundmntl\f_k2.htm
13:   0.87  d:\verity\k2\docs\doc\fundmntl\08_k22.htm
14:   0.84  d:\verity\k2\docs\doc\fundmntl\06_k21.htm
15:   0.80  d:\verity\k2\docs\doc\fundmntl\part4.htm
RC> z 4
4:    0.97  d:\verity\k2\docs\doc\fundmntl\08_k21.htm
The zone filter can be invoked together with the universal filter
or as a single filter. When the zone filter is invoked with the
universal filter as a helper filter, the universal filter's
character set recognizer should be used. When the zone filter is u
```

Unlike with the other two types of summaries, generating a dynamic summary requires access to the document in the repository, not just the collection.

Displaying Documents

rcvdk can display the content of text-based or HTML/XML/SGML-based documents. To view a document that is listed in the search results, use the view (or v) command:

Option	Description
v	Displays the first or next document in the results list. Highlights, or matched query terms, are indicated using reverse video, if possible. If not, double angle brackets are by default used to delimit the highlighted terms. For example, if you search for the term "universal filter," you see the following in the document display: `>>universal<< >>filter<<` You can change the highlighting delimiters, if desired; see "Highlighting of Search Terms" on page 347. To exit the document display, type **q**.
v *N*	Displays the Nth document in the results list. To exit the document display, type **q**.

Highlighting of Search Terms

By default, static highlighting is used, meaning that the positions of highlighted terms are determined by looking them up in the collection's word table. You can use the hlmode command to toggle to dynamic highlighting, in which highlighting is applied by finding the terms in the document content as it is streamed for display. In some situations, dynamic highlighting is more accurate.

To change how highlighted terms are marked in displayed documents, you can use the highlight command, followed by a paired set of characters (separate the sets by a space), representing the opening and closing highlight delimiters, respectively.

For example, the default highlight delimiters—paired double angle brackets—are equivalent to giving the following command:

```
RC> highlight >> <<
```

You can set any other characters you wish as the opening and closing delimiters. And you can view what the current delimiters are by executing highlight without any arguments. For example:

```
RC> highlight *** ####
RC> highlight
Current ***highlight#### mode.
RC>
```

Furthermore, you can specify different delimiters for different collection fields. For example, to specify double plus-signs as the delimiters for highlighted terms in the document-summary field only, you could give this command:

```
RC> highlight ++ ++ vdksummary
```

Then, if you perform a search and if your results display is configured to include the `vdksummary` field, you might get a result like this:

```
RC> s verity
Search update: finished (100%).  Retrieved: 1(1)/1.
RC> r
Retrieved: 1(1)/1
Number  vdksummary
1:      ++Verity++ provides software that enables organiza
RC>
```

Displaying XML Subdocuments

An XML *subdocument* is a portion of an XML document defined by one or more of the document's tag elements. For example, for a bibliography in XML format, it is possible to extract a subdocument consisting of only book titles.

If you have used `rcvdk` to search a collection containing XML documents, you can then use the `subdocs` command to specify a subdocument that should be displayed for each retrieved XML document.

For example, consider searching a collection of XML catalogs for an author whose name (specified by the `<author>` tag) is "Wilson":

```
RC> s Wilson <in> author
Search update: finished (100%).  Retrieved: 1(1)/30.
RC> r
Retrieved: 1(1)/30
Number  SCORE   VdkVgwKey
1:      0.7967  test4.xml
RC>
```

Viewing a result document in the normal manner causes the entire document to be displayed:

```
RC> highlight * *
RC> v
1:      0.7967  test4.xml
<?xml version="1.0"?>
<catalog>
  <book id="1" year="1998">
    <title>TCP/IP Reference</title>
    <author><last>*Wilson*</last><first>H.</first></author>
    <publisher>Bramburg Press</publisher>
    <price>59.95</price>
  </book>

  <book id="2" year="2004">
    <title>Advanced TCP/IP Programming</title>
    <author><last>*Wilson*</last><first>H.</first></author>
    <publisher>Western Books</publisher>
    <price discount="15%">45.00</price>
  </book>

  <book id="3" year="2000">
    <title>History of Network Protocols</title>
    <author><last>Anderson</last><first>Aaron</first></author>
    <author><last>Burke</last><first>Barry</first></author>
    <author><last>Chase</last><first>Chester</first></author>
    <publisher>Elite Publishers</publisher>
    <price>29.50</price>
  </book>
</catalog>
```

(This three-book catalog file is used in the rest of the examples shown here.)

Using the subdocs command, you can specify, for example, that you want to see only the book titles in the document:

```
RC> subdocs / //title
RC> v
1:      0.7967  test4.xml
<title>TCP/IP Reference<title>Advanced TCP/IP Programming</title>
<title>History of Network Protocols</title>
RC>
```

You can request display of more than one subelement—for example, title and price:

```
RC> subdocs / //title //price
RC> v
1:      0.7967  test4.xml
<title>TCP/IP Reference<price>59.95<title>Advanced TCP/IP
Programming<price>45.00<title>History of Network Protocols<price>
29.50</price>
RC>
```

Note that the XML elements are displayed as encountered in the document, without newlines or any other formatting inserted.

Command Syntax

The subdocs command uses this syntax:

```
subdocs context path1 [path2... ]
```

where *context* and *path*N are XPath fragments that describe the elements to be included in the subdocument. *context* is a prefix that specifies where in the document to start looking for the elements, and each *path*N denotes the path to a particular element, relative to *context*.

The context and path fragments consist of element or attribute names, optionally with certain XPath symbols:

- The context must start with the document root (represented by a single forward slash) or with the name of the outermost element.
- The path fragments consist of element names and certain XPath symbols. For example, a single forward slash (/) means "child of" and a double forward-slash (//) means "any descendent of".

Using the earlier example as an illustration, note that the command

```
RC> subdocs / //title
```

means "Display all `<title>` elements that are descendents of the document root"—in other words, anywhere in the document.

Given the specific XML file used for these examples, the same result could be achieved with this command:

```
RC>subdocs catalog book/title
```

which means "Display all `<title>` elements that are direct children of the element `<book>` that is a direct child of `<catalog>`, which is the element directly beneath the document root." (If any `<title>` elements were to occur outside of that path, they would not be displayed.)

Displaying Elements Based on Attributes

You can use the subdocs command along with an XPath attribute expression (`@`*`attribute`*) to restrict the display to elements that include a given attribute.

In the example file shown earlier, the `<price>` element for one of the books includes an attribute `discount`. The `<price>` element for other books does not include that attribute.

To display all prices, you would use a command like this:

```
RC> subdocs / //price
RC> v
1:      0.8351  test4.xml
<price>59.95</price><price discount="15%">45.00</price><price>
29.50</price>
RC>
```

To display only those prices that include a discount, use a command like this:

```
RC> subdocs / //price[@discount]
RC> v
1:      0.8351  test4.xml
<price discount="15%">RC>
```

Displaying Individual Elements of a Set

If a given XML element has a number of instances of the same subelement—for example, if a single book has multiple authors listed—you can use XPath subscript notation with `subdocs` to specify which of the subelements to display.

For example, to display the title and (first) author for each book in this example, you could use this command (results reformatted for clarity):

```
RC> subdocs /catalog book/title book/author[1]
RC> v
1:      0.7967  test4.xml
<title>TCP/IP Reference
<author><last>*Wilson*</last><first>H.</first>

<title>Advanced TCP/IP Programming
<author><last>*Wilson*</last><first>H.</first>

<title>History of Network Protocols
<author><last>Anderson</last><first>Aaron</first></author>
RC>
```

To display the title and only the third author for each book, change the subscript (results reformatted for clarity):

```
RC> subdocs /catalog book/title book/author[3]
RC> v
1:      0.7967  test4.xml
<title>TCP/IP Reference

<title>Advanced TCP/IP Programming

<title>History of Network Protocols
<author><last>Chase</last><first>Chester</first></author>
RC>
```

In this example, only one of the books has three authors, so no author is displayed for the other books.

Authenticating in rcvdk

For secure collections, you cannot view documents without supplying the appropriate user credentials. The `user` (or `u`) command allows you to set user credentials for an `rcvdk` session, by entering appropriate values for user ID, password, domain, and/or mailbox.

```
RC> user gwashington:cherry:POTOMAC
```

The credentials you supply must match the credentials in the repository where documents are stored.

Checking Document Access

You can also check ahead of time to see whether you have access to an individual document listed in the search results.

- Use the expert-mode `checkid` command to see if you have access to the collection document with a given VDK document ID (an integer value assigned at indexing time).
- Use the `checkkey` command to see if you have access to the collection document with a given document key.

Both `checkid` and `checkkey` allow you to check more than one document at a time. For example:

```
RC> checkkey
Please enter VdkDocKeys, one per line; finish by a blank line:
 ../docs/html/CollRef/filters14.html
../docs/html/CollectionReference/filters6.html

docKey:  ../docs/html/CollRef/filters14.html, access: yes
docKey: ../docs/html/CollRef/filters6.html, access: yes
2 documents checked in total
RC>
```

APPENDIXES

A

Supported Document Formats

This appendix lists the document formats supported both for indexing (using the Verity universal document filter and the XML filter) and for viewing (using the Verity Viewing Service). This information is based on the formats supported by Verity KeyView 9.1.

In the tables in this appendix, **Y** = supported; **N** = not supported; **T** = only text is extracted; and **M** = only metadata is extracted.

Additional information on support for indexing and viewing Microsoft Personal Folder (PST) files is presented in the final section.

This appendix includes the following sections:

Archive Formats

Format	Version(s)	Extension	Indexing	Viewing
PKZIP	through 2.04g	`ZIP`	**Y**	**Y**

Computer-Aided Design

Format	Version(s)	Extension	Filter	Export
AutoCAD Drawing	R13, R14, 2000, 2004	`DWG`	**Y**	**Y**
AutoCAD Drawing Exchange	R13, R14, 2000, 2004	`DLL`	**Y**	**Y**
Microsoft Project	98, 2000, 2002	`MPP`	**M**	**M**
Microsoft Visio	5, 6, 2000, 2002, 2003	`VSD`	**Y**	**T**

Display Formats

Format	Version(s)	Extension	Filter	Export
Adobe Portable Document Format	1.1 to 1.6	`PDF`	**Y**	**Y**

Graphic Formats

Format	Version(s)	Extension	Filter	Export
Computer Graphics Metafile	n/a	CGM	**N**	**Y**
CorelDRAW (TIFF header)	through to 9.0	CDR	**Y**	**Y**
DCX Fax System	TIFF/CCITT/DCX	DCX	**Y**	**Y**
Encapsulated PostScript (raster)	TIFF header	EPS	**N**	**Y**
Enhanced Metafile	n/a	EMF	**Y**	**Y**
Graphic Interchange Format	87, 89	GIF	**N**	**Y**
JPEG	n/a	JPEG	**N**	**Y**
Lotus AMIDraw Graphics	n/a	SDW	**N**	**Y**
Lotus Pic	n/a	PIC	**N**	**Y**
Macintosh Raster	2	PIC PCT	**N**	**Y**
MacPaint	n/a	PNTG	**N**	**Y**
Microsoft Office Drawing	n/a	MSO	**N**	**Y**
PC PaintBrush	3	PCX	**N**	**Y**
Portable Network Graphics	n/a	PNG	**N**	**Y**
SGI RGB Image	n/a	RGB	**N**	**Y**
Sun Raster Image	n/a	RS	**N**	**Y**
Tagged Image File	3.0 to 6.0	TIFF	**M**	**Y**
Truevision Targa	2	TGA	**N**	**Y**
Windows Animated Cursor	n/a	ANI	**N**	**Y**
Windows Bitmap	n/a	BMP	**N**	**Y**
Windows Icon Cursor	n/a	ICO	**N**	**Y**
Windows Metafile	3	WMF	**Y**	**Y**
WordPerfect Graphics 1	1	WPG	**N**	**Y**
WordPerfect Graphics 2	2, 7	WPG	**N**	**Y**

Mail Formats

Format	Version(s)	Extension	Filter	Export
Microsoft Outlook	97, 2000, 2002, 2003	`MSG`	**Y**	**Y**
Microsoft Outlook Express	n/a	`EML`	**Y**	**N**
Microsoft Outlook Personal Folder (Windows only)	97, 2000, 2002, 2003	`PST`	**Y**	**Y**

Multimedia Formats

The multimedia formats supported by Viewing SDK (except MP3) use Microsoft ActiveMovie. ActiveMovie is not included in Viewing SDK and must be licensed from Microsoft.

Format	Version(s)	Extension	Filter	Export
MPEG-1 Audio layer 3	ID3 versions 1 and 2	`MP3`	**M**	**M**

Presentation Formats

Format	Version(s)	Extension	Filter	Export
Applix Presents ()	4.0, 4.2, 4.3, 4.4	`AG`	**Y**	**Y**
Corel Presentations	7, 9, 10, 11, 2000	`SHW`	**Y**	**Y**
Lotus Freelance Graphics	96, 97, 98, R9, 9.8	`PRZ`	**Y**	**Y**
Lotus Freelance Graphics 2	2	`PRE`	**Y**	**Y**
Microsoft PowerPoint Windows	97, 2000, 2002, 2003	`PPT`	**Y**	**Y**

Format	Version(s)	Extension	Filter	Export
Microsoft PowerPoint Windows	95	`PPT`	Y	Y
Microsoft PowerPoint PC	4	`PPT`	Y	Y
Microsoft PowerPoint Macintosh	98	`PPT`	Y	Y

Spreadsheet Formats

Format	Version(s)	Extension	Filter	Export
Applix Spreadsheets	4.2, 4.3, 4.4	`AS`	Y	Y
Comma Separated Values ()	n/a	`CSV`	Y	Y
Corel Quattro Pro	5, 6, 7, 8	`QPW WB3`	Y	Y
Lotus 1-2-3	96, 97, R9, 9.8	`123`	Y	Y
Lotus 1-2-3	2, 3, 4, 5	`WK4`	Y	Y
Lotus 1-2-3 Charts	2, 3, 4, 5	`123`	Y	Y
Microsoft Excel Windows	2.2, through 2003	`XLS`	Y	Y
Microsoft Excel Charts	2, 3, 4, 5, 6, 7	`XLS`	Y	Y
Microsoft Excel Macintosh	98	`XLS`	Y	Y
Microsoft Works Spreadsheet	1, 2, 3, 4	`S30` `S40`	Y	Y

Text-Processing Formats

Text and Markup

Format	Version(s)	Extension	Filter	Export
ANSI	n/a	TXT	Y	Y
ASCII	n/a	TXT	Y	Y
HTML	3, 4.0	HTM	Y	Y
MIME	n/a	EML	Y	Y
IBM DCA/RFT (Revisable Form Text)	SC23-0758-1	DC	Y	Y
Microsoft Excel Windows XML	2003	XML	Y	T
Microsoft Word Windows XML	2003	XML	Y	T
Microsoft Visio XML	2003	VDX	Y	T
OpenOffice	1, 1.1	SXI SXP SXC SXW	Y	T
Rich Text Format	1 through 1.7	RTF	Y	Y
StarOffice	6, 7	SXI SXP SXC SXW	Y	T
Unicode Text	3, 4	TXT	Y	Y
XHTML	1.0	HTM	Y	Y
XML (generic)	1.0	XML	Y	T

Word Processors

Format	Version(s)	Extension	Filter	Export
Adobe Maker Interchange Format	5, 5.5, 6, 7	MIF	Y	Y
Applix Words	3.11, 4, 4.1, 4.2, 4.3, 4.4	AW	Y	Y
DisplayWrite	4	IP	Y	Y
Folio Flat File	3.1	FFF	Y	Y

Format	Version(s)	Extension	Filter	Export
Fujitsu Oasys	7	OA2	Y	Y
JustSystems Ichitaro	8, 9, 10, 12	JTD	Y	Y
Lotus AMI Pro	2, 3	SAM	Y	Y
Lotus AMI Professional Write Plus	2.1	AMI	Y	Y
Lotus Word Pro (Windows x86 only)	96, 97, R9	LWP	Y	Y
Lotus SmartMaster (Windows x86 only)	96, 97	MWP	Y	Y
Microsoft Word PC	4, 5, 5.5, 6	DOC	Y	Y
Microsoft Word Windows	1.0 and 2.0	DOC	Y	Y
Microsoft Word Windows	6, 7, 8, 95	DOC	Y	Y
Microsoft Word Windows	97, 2000, 2002, 2003	DOC	Y	Y
Microsoft Word Macintosh	4, 5, 6, 98	DOC	Y	Y
Microsoft Works	1, 2, 3, 4	WPS	Y	Y
Microsoft Works	6, 2000	WPS	Y	Y
Microsoft Windows Write	1, 2, 3	WRI	Y	Y
WordPad	through 2003	RTF	Y	Y
WordPerfect Windows	5, 5.1	WO	Y	Y
WordPerfect Windows	6, 7, 8, 9, 10, 11, 2000	WPD	Y	Y
WordPerfect Linux	6, 8	WPS	Y	Y
WordPerfect Macintosh	1.02, 2, 2.1, 2.2, 3, 3.1	WPS	Y	Y
XyWrite	4.12	XY4	Y	Y

Notes on K2 Support for PST Files

K2 supports indexing and viewing the contents of Microsoft Personal Folder (PST) files. This section lists some of the prerequisites for successful indexing and viewing of these files.

- Microsoft Outlook must be installed and must be the default email client on the host machine containing the PST file(s) to be indexed. Moreover, the version of the installed Microsoft Outlook installed must match the version of the PST files to be indexed.
- Indexing a PST file requires having write access to it. PST files that are read-only cannot be indexed.
- PST files cannot be accessed for indexing purposes on a mapped drive. Windows does not support opening PST files over a network share.
- Password-protected PST files can be indexed (see "Providing Passwords for Document Access (style.pw)" on page 255). However, note that password characters—if they are not to be encrypted—must be plain ASCII or other valid characters in the encoding of the locale of the collection into which the PST is being indexed.
- Note that, since a PST file is commonly in a compressed format, the collection resulting from indexing it can be up to several times larger than the file itself.

B

Supported Date Formats

The Verity engine can recognize a wide variety of date formats during indexing and searching. Numerous date formats can be parsed according to parsing rules you supply in the form of regular expressions in the `style.tde` file. For information about defining date fields, see "Defining Collection Fields" on page 145.

This appendix contains the following information:

- Date Import Formats
- Date Import Format Strings
- Numeric Date Formats

Date Import Formats

The Verity engine can parse a variety of date formats. A single date format can include a calendar format plus an optional time format.

Table B-1 on page 367 presents date formats supported by the Verity parser. These formats can be read, translated, and stored in a collection's document table.

If you are using `mkvdk`, use the `-datefmt` option. If you are developing an application using a Verity API, then you must set the `dateInputFormat` member in `VdkSessionNewArgRec`.

Date Import Format Strings

A variety of constructs can be used to define the import date format string.

6-digit and 8-digit numbers are interpreted as having a year, month, and day according to the prevailing datefmt specification, as described in "Numeric Date Formats" on page 369. If datefmt is not specified, then the engine first tries to interpret 6- and 8-digit numbers in year-month-day order. If that fails, the engine tries to interpret the numbers in month-day-year order.

Table Conventions

Within the Description column of the Date Formats table, shown here, the following constructs describe a date format element. These are representational constructs only. That is, elements such as MM DD YYYY are not actually ever typed in anywhere.

Date Format Element	Description
MM	Represents a one or two-digit numeric month, as 3 or 12.
Mon	Represents an alphabetic month, 3 or more characters in length, as Feb or February.
DD	Represents a one or two-digit numeric day of the month, as 1, 01, or 29.
YYYY	Represents a four-digit numeric year, as 1997.
TIME	Represents a time format, as described under "Time Formats" on page 368.
DDD	Represents a three-digit numeric Julian day of the year, as 129.

Table B-1 Import Date Formats

Description	Examples
Month-day-year, numeric, American date format (`MM DD YY, MM-DD-YY, MM/DD/YY, or MM.DD.YY`).	2 17 97 02-17-97 02/17/1997 2.17.1997
Day-month-year, numeric, English format (*`DD`*` MM YY, DD-MM-YY, DD/MM/YY, or DD.MM.YY`).	17 02 97 17-2-97 17/02/1997 17.2.97
Year-month-day, numeric, European date format (`YY MM DD, YY-MM-DD, YY/MM/DD`, or `YY.MM.DD`).	97 02 17 97-2-17 1997/2/17 97.2.17
Year-day-month, numeric, Swedish date format (`YY DD MM, YY-DD-MM, YY/DD/MM, YY.DD.MM`).	97 17 02 97-17-2 97/17/2 97.17.2
Day (numeric), month (alphabetic), year (numeric) (`DD Mon YY`).	17 Feb 97 17 February 1997
Month (alphabetic), day (numeric), year (numeric) (`Mon DD YY).`	Feb 17 97 February 17 1997
Month (alphabetic), year (numeric) (`Mon YY`).	Mar 97 January 95
Month (numeric), year (numeric) (`MM YY`).	02 97 12 97
Julian Date format (`YYDDD`).	97364 20001
Any of the preceding, followed with an optional time expression (`DD Mon YY TIME`); the time can be expressed using any of the time formats described under "Time Formats" on page 368.	17 Feb 97 23:59 17 February 97 01:50
Dow Jones date format (`hh mm DD MM YY`)	23 59 25 12 91 00 00 01 01 32
Zulu date format (`DDhhmmZ Mon YY`); see "Zulu Date Format" (next)	252312Z JAN 94 310101Z JAN 94

Zulu Date Format

The Verity engine assumes that the time in Zulu date format is in Greenwich Mean Time (GMT). If you use a different time format when you enter search criteria, local time is assumed. Local time depends on the time and time zone settings of your operating system.

Thus, if you enter the following date as the DATE field value for a document:

```
252312Z JAN 94
```

and your computer is set to Pacific Standard Time (PST), a Verity client finds this document if you query the following `DATE` field value:

```
Jan 25 94 15:12
```

This is because PST is 8 hours behind GMT.

Time Formats

Any of the constructs can include an optional time format, as, for example, `02/17/1997 08:55`.

The Verity engine understands time formats that have one of the following structures. As in Table B-1 on page 367, these are representational constructs only. That is, elements such as `hh:mm:ss` are not actually ever typed in anywhere. You would actually see instead `12:34:56`.

```
hh:mm
```

```
hh:mm:ss
```

```
hh:mm:ss TIMEOFDAY
```

```
hh:mm:ss TIMEOFDAY TIMEZONE
```

The time format elements are described here.

Time Format Element	Description
`hh`	This element represents the hours, as 01,1, 11, or 23.
`mm`	This element represents the minutes, as 01 or 55.

Time Format Element	Description
ss	This optional element represents the seconds, as 01 or 55.
TIMEOFDAY	This optional element specifies the 12-hour representation of the time, one of either AM or PM.
TTIMEZONE	This optional element represents a time zone, as PST or EDT.

For example, the Verity engine can import the following combined date and time:

```
Oct 15 1997 01:33:12 AM PST
```

Field searches can be performed on dates and times using any of the formats shown in the preceding table.

Numeric Date Formats

The date format constructs described in this section resolve ambiguities in numerical date representations.

By default, dates input into the document table are assumed to be in American numeric date format, that is, the month-day-year format (MM-DD-YY). This means that if a user enters a date for a field search query in the same format, the Verity engine can interpret the date and perform the appropriate retrieval. Numeric date formats can be delimited by spaces or slashes in addition to dashes.

If users want to enter date field search criteria in a different format, such as English or European numeric date format, that is, day-month-year or year-month-day, then you must specify to the application which date format to use. If you are using mkvdk, use the -datefmt option. If you are developing your own application, then you must set the dateInputFormat member in VdkSessionNewArgRec.

The datefmt syntax options are listed here.

datefmt syntax	Description
MDY	Represents the month-day-year (MM-DD-YY) numeric format (American format, the default).

DMY	Represents the day-month-year (DD-MM-YY) numeric format (English format).
YMD	Represents the year-month-day (YY-MM-DD) numeric format (European format).
YDM	Represents the year-day-month (YY-DD-MM) numeric format (Swedish format).

Example:

```
mkvdk -collection collname -bulk -insert filespec -datefmt DMY
```

The preceding `mkvdk` command would interpret numeric dates in the format XX-YY-ZZ as DD-MM-YY (day-month-year).

C

Supported Regular Expressions

This appendix describes the syntax of regular expressions that an application can use to identify text patterns for valid words. For example, an application developer can use regular expressions to identify text patterns in the `style.lex` file.

The Verity engine's parser interprets regular expression syntax nearly identical to the UNIX regular expression syntax. The Verity engine's regular expression syntax also includes some extensions for matching substrings.

This appendix covers these topics:

- Symbols
- Substrings

Operators for Regular Expressions

The following table lists the regular expression operators available in the Verity engine and the pattern the operator matches.

Operator	Matched Pattern
`x`	The character “x”.
`\x`	The character “x”, even if x is an operator. You would use this, for example, to search for the $ character, which is an operator. (\n and \t are exceptions; see the following explanations.)
`\b`	A backspace.

`\f`	A form-feed.
`\n`	A newline.
`\r`	A carriage-return.
`\t`	A tab.
`\v`	A vertical tab.
`[xy]`	The character “x” or “y”.
`[x-z]`	The characters “x”, “y”, or “z”; this regular expression searches for a range of characters. For example, the expression [a-z] is used to search for any character in the lowercase alphabet; [0-9] is used to search for any digit.
`.`	Any character but newline.
`[^z]`	Any character but “z”.
`^x`	An “x” at the beginning of a line.
`x$`	An “x” at the end of the line.
`x?`	0 or 1 occurrence of “x”.
`x*`	0 or more occurrences of “x”.
`x+`	1 or more occurrences of “x”.
`x|y`	An “x” or a “y”.
`(x|y)z`	“xz” or “yz”; the parentheses are used for grouping.
`{symbol}`	The translation of a symbol defined earlier in the file.

Symbols

You can define symbols to avoid redefining expressions to search for common patterns. Symbol definitions should appear at the top of a file, before any regular expressions that use them. To define symbols, use the define statement with the following syntax.

```
define: symbol "regular expression"
```

Element	Description
`symbol`	The word replaced by the quoted pattern; the symbol name can contain any alphanumeric characters.
`"reg_exp"`	A regular expression that the Verity engine uses for matching when it encounters the defined symbol. Double quotes are used in the event that the matched pattern contains white space.

Symbol Examples

The `define` statement is used to codify an expression so that user-defined symbols can be included in regular expressions. For example, you might use the following definitions at the beginning of a `style`.lex file:

```
define: D "[0-9]"
```

The preceding statement defines the symbol D, which represents any digit.

```
define: SPACE "[ \t]"
```

The preceding statement defines a symbol SPACE that represents either a space, or a tab.

You can also use previously defined symbols in other symbol definitions. For example, you might first define a symbol for any digits as follows:

```
define: D  "[0-9]"
```

You could then use the symbol {D} in other symbol definitions, as in this definition of a YEAR symbol as follows:

```
define: YEAR "{D}{D}{D}{D}"
```

Symbols in regular expressions must be enclosed in braces.

Substrings

Normally, the text returned is the entire string that matches the pattern in the regular expression. The Verity engine includes an extension to regular expression syntax that allows you to identify a string and then select a substring of that string. To define a substring and retrieve only that substring, enclose the substring in angle brackets.

```
"TITLE:<.*>"
```

This expression returns any characters after the string “TITLE:”, but not including the string “TITLE:”.

```
"Volume{SPACE}+<{DIGIT}+>"
```

This expression returns any number of digits following the string “Volume” and one or more spaces.

Regular Expression Examples

Some simple examples of regular expressions are presented here.

Example 1

```
^[0-9]
```

This expression matches any digit at the beginning of a line.

Example 2

```
^[0-9]+
```

This expression matches one or more digits at the beginning of a line.

Example 3

```
[^0-9]
```

This expression matches any single character except a digit.

Example 4

```
"TITLE:.*$"
```

This expression matches a string beginning with “TITLE:” and followed by any characters until the end of the line.

Example 5

```
"^Sub(j|ject):.*$"
```

This expression matches the string “Subj” or “Subject” that occurs at the beginning of a line and is followed by a colon and any other characters until the end of the line.

Example 6

```
"FIELD:\t"
```

This expression matches the string “FIELD:” followed by a tab character.

Collection Limits

This appendix lists the acceptable limits and ranges for collections. For information on limits in topics, see the *Verity Query Language and Topic Guide*.

Table D-1 Collection limits

Limit	Description
Wildcards	Wildcard auto-expansion is limited to 16,000 matches.
Number of collections	The maximum number of physical collections that each K2 Server can search at one time is 128.
Documents per collection	The maximum number of documents per collection is 16 million.
Documents per partition	The maximum number of documents per partition is 64,000.
Fields per collection	The maximum number of fields per collection is 250.
Field length	The maximum length of any field is 32,000 bytes. The number of actual characters that translates to will depend on the character set used.
Field length in a BIFs	The maximum length of a field value in a bulk insert files is 32 KB. The number of actual characters that translates to will depend on the character set used.
Zones per document	Unlimited.
Characters in path	The maximum path size is 256 characters on Windows and 1024 characters on UNIX.
Maxdocs with sort spec	The maximum number of documents returned when a sort specification is applied is 16,000.

Table D-1 Collection limits (continued)

Limit	Description
Sort fields per search	The maximum number of fields that can be included in a sort specification is 16.
Maximum repository nodes	The maximum number of repository nodes (secured file systems, unsecured file systems, HTTP servers, Exchange servers, and so on) permitted for a given collection is 1024. Defined by the constant `MAX_K2REPOSITORY_NODES`.
Spanning word list size	Maximum size = 4 GB (if file system is 64-bit; otherwise, maximum size is 2 GB). By default, Windows, Solaris, and Linux support 64-bit file systems. To support larger spanning word lists on HP-UX and AIX, you must specifically enable 64-bit filesystem support at the operating-system level; consult your operating-system documentation for instructions. Also for HP-UX and AIX, set the environment variable `VDK_LARGEFILE_SUPPORT` to 1. **Note:** If you need to turn off Verity support for 64-bit file systems (for example, if you are using FAT32 or CD-ROM file systems), set the environment variable `VDK_LARGEfILE_SUPPORT` to 0.

Index