

Excalibur Text Search DataBlade Module

User's Guide

Version 1.1
March 1998
Part No. 000-5143

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Copyright © 1981-1998 by Informix Software, Inc., or its subsidiaries, provided that portions may be copyrighted by third parties, as set forth in documentation. All rights reserved.

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by “®,” and in numerous other countries worldwide:

INFORMIX®; Illustra™; DataBlade®

All other marks or symbols are registered trademarks or trademarks of their respective owners.

ACKNOWLEDGMENTS

Writing Team: Juliet Shackell, Tom Demott

Contributors: David Ashkenas, Kimberly Bostrom, Kevin Foster, Mike Frame, Frank Glandorf, Daniel Howard, Steven Leslie, Karin Moore, Bob Nowacki, Kumar Ramaiyer, Hari Rao, Jackie Ryan, Dave Segleau, Scott Stark, Weiming Ye

RESTRICTED RIGHTS/SPECIAL LICENSE RIGHTS

Software and documentation acquired with US Government funds are provided with rights as follows: (1) if for civilian agency use, with Restricted Rights as defined in FAR 52.227-19; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by negotiated vendor license as prescribed in DFAR 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce the legend.

Table of Contents

Introduction

About This Manual	3
Organization of This Manual	3
Types of Users	4
Software Dependencies	5
Documentation Conventions	5
File Naming Conventions	6
Typographical Conventions	6
Icon Conventions	7
Syntax Conventions	7
Additional Documentation	8
Printed Documentation	9
On-Line Documentation	10
Related Reading	11
Informix Welcomes Your Comments	12

Chapter 1

DataBlade Module Overview

Conceptual Overview	1-4
The Excalibur Text Search DataBlade Module	1-5
How the DataBlade Module Works	1-5
Components of the Excalibur Text Search DataBlade Module	1-7
The etx Access Method	1-7
The etx_contains() Operator	1-12
Routines Defined for the DataBlade Module.	1-16
Word Lists	1-16
Synonym Lists	1-16
Stopword Lists	1-19
Character Sets	1-20
Built-In Character Sets	1-21
User-Defined Character Sets	1-21

Highlighting	1-22
The etx_GetHilite() Function.	1-23
The etx_HiliteType Data Type	1-24

Chapter 2

Text Search Concepts

Excalibur Text Search Types	2-3
Keyword Search	2-3
Boolean Search	2-5
Phrase Search	2-6
Proximity Search	2-8
Performing a Fuzzy Search	2-10
Substitution and Transposition	2-10
Ranking the Results of a Fuzzy Search	2-12
Limiting the Number of Rows a Fuzzy Search Returns.	2-14
Pattern Search	2-15
Phrase Searching with Pattern Matching.	2-17

Chapter 3

Tutorial

Step 1: Creating a Table Containing a Text Search Column	3-4
Creating dbspaces and sbspaces for Storage	3-4
Creating the Table	3-5
Step 2: Populating the Table with Text Search Data	3-5
Step 3: Creating Word Lists and a User-Defined Character Set	3-6
Creating a Stopword List	3-6
Creating a Synonym List	3-7
Creating a User-Defined Character Set	3-8
Step 4: Creating an etx Index	3-9
Determining the etx Index Parameters	3-10
Creating an etx Index	3-10
Step 5: Performing Text Search Queries	3-12
Step 6: Highlighting	3-14

Chapter 4

Data Types

etx_ReturnType	4-4
etx_HiliteType.	4-6
IfxDocDesc	4-9
IfxMRData	4-14

Chapter 5

Routines

etx_contains()	5-5
--------------------------	-----

	etx_CloseIndex()	5-15
	etx_CreateCharSet()	5-17
	etx_CreateStopWlst()	5-21
	etx_CreateSynWlst()	5-23
	etx_DropCharSet()	5-26
	etx_DropStopWlst()	5-28
	etx_DropSynWlst()	5-30
	etx_GetHilite()	5-31
	etx_Release()	5-40
	txt_Release()	5-41
Chapter 6	etx Index Parameters	
	Overview of the etx Index Parameters	6-3
	WORD_SUPPORT	6-4
	PHRASE_SUPPORT	6-4
	CHAR_SET	6-5
	STOPWORD_LIST	6-6
	INCLUDE_STOPWORDS	6-7
Chapter 7	DataBlade Module Administration	
	Retrieving DataBlade Module Version Information	7-3
	Sbspaces	7-4
	Default Sbspaces	7-4
	Logging of Sbspaces	7-5
	Parameters of the onspaces Utility	7-6
	DataBlade Dependencies	7-7
	Estimating the Size of an etx Index	7-7
	Internal Structure of an etx Index	7-9
	etx Index Sharing	7-10
	Advantages and Disadvantages of Text Storage Data Types	7-10
	BLOB	7-11
	CLOB	7-11
	LVARCHAR	7-11
	IfxDocDesc	7-12
	IfxMRData	7-12
	Performance Tips	7-13
	Database Server Configuration	7-13
	Loading Data	7-14

Appendix A	Character Sets
	Glossary
	Index

Introduction

About This Manual	3
Organization of This Manual	3
Types of Users	4
Software Dependencies	5
Documentation Conventions	5
File Naming Conventions	6
Typographical Conventions	6
Icon Conventions	7
Syntax Conventions	7
Additional Documentation	8
Printed Documentation	9
On-Line Documentation.	10
Related Reading	11
Informix Welcomes Your Comments.	12

T

his chapter introduces the *Excalibur Text Search DataBlade Module User's Guide*. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout.

About This Manual

This manual describes the Excalibur Text Search DataBlade module and how to access and use its components, which include the **etx** access method, the **etx_contains()** search operator, synonym matching, and highlighting. The manual also includes a tutorial for setting up and performing example text searches.

This section discusses the organization of the manual, the intended audience, and the associated software products that you must have to develop and use the DataBlade module.

Organization of This Manual

This manual includes the following chapters:

- This Introduction provides an overview of the manual, describes the documentation conventions used, and explains the generic style of this documentation.
- [Chapter 1, “DataBlade Module Overview,”](#) provides an overview of the Excalibur Text Search DataBlade module and its principal components.
- [Chapter 2, “Text Search Concepts,”](#) discusses in detail the different types of searches provided by the Excalibur Text Search DataBlade module, text search terminology, and syntax.

- [Chapter 3, “Tutorial,”](#) provides step-by-step instructions that show you how to create tables with text search columns, insert data into the tables, create custom synonym and stopword lists and user-defined character sets, create text search indexes, and perform text searches.
- [Chapter 4, “Data Types,”](#) is a reference chapter that describes the Informix constituent data types `IfxMRData`, `IfxDocDesc`, `etx_ReturnType`, `etx_HiliteType`, and `LLD_Locator`. It describes how to store text data in the `IfxMRData` and `IfxDocDesc` storage data types and how to use the `etx_ReturnType` and `etx_HiliteType` data types to access scoring and highlighting information.
- [Chapter 5, “Routines,”](#) is a reference chapter that describes the **`etx_contains()`** operator that you use to define and perform text searches. It also describes the DataBlade-defined routines that enable you to create and maintain synonym and stopword lists and user-defined character sets.
- [Chapter 6, “etx Index Parameters,”](#) is a reference chapter that describes the parameters you can use to customize **`etx`** indexes.
- [Chapter 7, “DataBlade Module Administration,”](#) discusses system administration topics such as estimating the size of an **`etx`** index, creating and using sbspaces for storing smart large object data, the advantages and disadvantages of storing text data in various different types of data types, and general performance tips.
- [Appendix A, “Character Sets,”](#) describes the three built-in character sets that the Excalibur Text Search DataBlade module supports: ASCII, ISO, and OVERLAP_ISO. It also provides a generic map of the standard and extended characters to aid you in creating your own user-defined character sets.
- A glossary of relevant database and text search terms follows the chapters, and an index directs you to areas of particular interest.

Types of Users

This manual is written for the following groups:

- Users who want to perform SQL searches of text data stored in proprietary format

- Programmers who want to build applications that call on the Excalibur text search engine to perform text searches
- Programmers who want to modify existing applications to take advantage of **etx** indexes

Software Dependencies

You must have the following Informix software to use the Excalibur Text Search DataBlade module:

- Informix Dynamic Server with Universal Data Option
- The Informix LOB Locator DataBlade module
- The Informix Text Descriptor DataBlade module

The following three items are not necessary to install or run the Excalibur Text Search DataBlade module but are listed as available application development tools for using the DataBlade module:

- DB-Access
- INFORMIX-ESQL/C
- DataBlade API

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set.

The following conventions are covered:

- File naming conventions
- Typographical conventions
- Icon conventions
- Syntax conventions

File Naming Conventions

This manual uses the UNIX file separator when specifying a file or directory on the operating system: forward slash (/). If you are using the NT version of the Excalibur Text Search DataBlade module, substitute the UNIX file separator with a backslash (\). For example, the NT version of the UNIX file `/local/excal/dbms.txt` is `\local\excal\dbms.txt`.

This manual also refers to the UNIX environment variable `$INFORMIXDIR`. On NT, this environment variable is `%INFORMIXDR%`.

Typographical Conventions

This manual uses the following standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.




Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
<i>italics</i>	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
monospace	Information that the product displays and information that you enter appear in a monospace typeface.



***Tip:** When you are instructed to “enter” characters or to “execute” a command, immediately press RETURN after the entry. When you are instructed to “type” the text or to “press” other keys, no RETURN is required.*

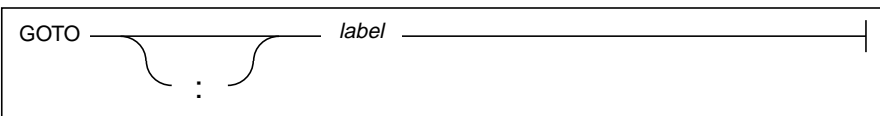
Icon Conventions

Comment icons identify warnings, important notes, or tips. This information is always displayed in *italics*.

Icon	Description
	The <i>warning</i> icon identifies vital instructions, cautions, or critical information.
	The <i>important</i> icon identifies significant information about the feature or operation that is being described.
	The <i>tip</i> icon identifies additional details or shortcuts for the functionality that is being described.

Syntax Conventions

This section describes conventions for syntax diagrams. Each diagram displays the sequences of required and optional keywords, terms, and symbols that are valid in a given statement, command line, or other specification, as in the following diagram of the GOTO statement.



Keep in mind the following rules when you read syntax diagrams in this book:

- For ease of identification, all of the keywords (like GOTO in the preceding diagram) are shown in UPPERCASE characters, even though you can type them in either uppercase or lowercase characters.
- Terms for which you must supply specific values or names are in *italics*. In this example, you must replace *label* with an identifier. Below each diagram that contains an italicized term, a table identifies what you can substitute for the term.
- All of the punctuation and other nonalphabetic characters are literal symbols. In this example, the colon is a literal symbol.
- Each syntax diagram begins at the upper left corner and ends at the upper right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement. (For a few diagrams, notes in the text identify path segments that are mutually exclusive.)

Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. Except for separators in loops, which the path approaches counterclockwise from the right, the path always approaches elements from the left and continues to the right. Unless otherwise noted, at least one blank character separates syntax elements.

Additional Documentation

The Excalibur Text Search DataBlade module documentation set includes printed manuals, on-line manuals, and on-line help.

This section describes the following parts of the documentation set:

- Printed documentation
- On-line documentation
- Related reading

Printed Documentation

The following related Informix documents complement the information in this manual set:

- Whoever installs your Informix products should refer to the *UNIX Products Installation Guide* for your particular release to ensure that your Informix product is properly set up before you begin to work with it. The *UNIX Products Installation Guide* includes a matrix that depicts possible client/server configurations.
- Before you can use the Excalibur Text Search DataBlade module, you must install and configure Informix Dynamic Server with Universal Data Option. The administrator's guide for your database server provides information about how to configure Universal Data Option and also contains information about how it interacts with DataBlade modules.
- Installation instructions for the Excalibur Text Search DataBlade module are provided in the hard copy ReadMe First sheet for DataBlade modules that is packaged with this product. Once you have installed the DataBlade module, you must use BladeManager to register it into the database where the DataBlade module will be used. See the [BladeManager User's Guide](#) for details on registering DataBlade modules.
- The Excalibur Text Search DataBlade module uses data types and routines defined by the LOB Locator DataBlade module. Refer to the [LOB Locator DataBlade Module Programmer's Guide](#) for more information on these data types and functions.
- The Excalibur Text Search DataBlade module uses data types defined by the Text Descriptor DataBlade module. Refer to the Text Descriptor DataBlade module release notes for more information on the Text Descriptor DataBlade module. [Chapter 4, "Data Types,"](#) of this guide also describes the data types in detail.
- If you plan to develop your own DataBlade modules using the Excalibur Text Search DataBlade module as a foundation, read the [DataBlade Developers Kit User's Guide](#). This manual describes how to develop DataBlade modules using BladeSmith, BladePack, and BladeManager.

- If you have never used Structured Query Language (SQL), read the [Informix Guide to SQL: Tutorial](#). It provides a tutorial on SQL as it is implemented by Informix products. It also describes the fundamental ideas and terminology for planning and implementing an object-relational database.
- A companion volume to the tutorial, the [Informix Guide to SQL: Reference](#), includes details of the Informix system catalog tables, describes Informix and common environment variables that you should set, and describes the data types that Informix database servers support.
- The [Guide to GLS Functionality](#) describes how to use nondefault locales with Informix products. A locale contains language- and culture-specific information that Informix products use when they format and interpret data.
- The [DB-Access User Manual](#) describes how to invoke the DB-Access utility to access, modify, and retrieve information from Informix database servers.

Informix Error Messages might also be useful if you do not want to look up your error messages on-line.

On-Line Documentation

In addition to the Informix set of manuals, the following on-line files supplement the information in this manual. They are located in the directory `$INFORMIXDIR/extend/ETX.<version>`, where `<version>` refers to the latest version of the DataBlade module installed on your computer.

On-Line File	Purpose
ETXREL.TXT	Describes known problems and their workarounds, new and changed features, and special actions required to configure and use the Excalibur Text Search DataBlade module on your computer
ETXDOC.TXT	Describes features not covered in the manuals or modified since publication
ETXMAC.TXT	Describes platform-specific information regarding the release.

Please examine these files because they contain vital information about application and performance issues.

Related Reading

For additional technical information on database management, consult the following books. The first book is an introductory text for readers who are new to database management, while the second book is a more complex technical work for SQL programmers and database administrators. The third book discusses object-relational database management systems:

- *Database: A Primer* by C. J. Date (Addison-Wesley Publishing, 1983)
- *An Introduction to Database Systems* by C. J. Date (Addison-Wesley Publishing, 1994).
- *Object-Relational DBMSs: The Next Great Wave* by Michael Stonebraker and Dorothy Moore (Morgan Kaufman Publications, 1996)

To learn more about the SQL language, consider the following books:

- *A Guide to the SQL Standard* by C. J. Date with H. Darwen (Addison-Wesley Publishing, 1993)
- *Understanding the New SQL: A Complete Guide* by J. Melton and A. Simon (Morgan Kaufmann Publishers, 1993)
- *Using SQL* by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

To learn more about indexing and compressing text documents, consider the following book:

- *Managing Gigabytes* by Ian H. Witten, Alistair Moffat, and Timothy C. Bell (Van Nostrand Reinhold, 1994)

To use the Excalibur Text Search DataBlade module, you should be familiar with your computer operating system. If you have limited UNIX system experience, consult your operating system manual or a good introductory text before you read this manual. The following texts provide a good introduction to UNIX systems:

- *Introducing the UNIX System* by H. McGilton and R. Morgan (McGraw-Hill Book Company, 1983)
- *Learning the UNIX Operating System*, by G. Todino, J. Strang, and J. Peek (O'Reilly & Associates, 1993)

- *A Practical Guide to the UNIX System*, by M. Sobell (Benjamin/Cummings Publishing, 1989)
- *UNIX for People* by P. Birns, P. Brown, and J. Muster (Prentice-Hall, 1985)
- *UNIX System V: A Practical Guide* by M. Sobell (Benjamin/Cummings Publishing, 1995)

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Please include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
Technical Publications
300 Lakeside Dr., Suite 2700
Oakland, CA 94612

If you prefer to send electronic mail, our address is:

`doc@informix.com`

Or, send a facsimile to Technical Publications at:

650-926-6571

We appreciate your feedback.

DataBlade Module Overview

Conceptual Overview	1-4
The Excalibur Text Search DataBlade Module	1-5
How the DataBlade Module Works	1-5
Components of the Excalibur Text Search DataBlade Module	1-6
The etx Access Method	1-7
Data Types	1-7
Operator Classes	1-8
The Filtering Capability	1-9
Creating an etx Index	1-9
Specifying etx Index Parameters	1-11
The etx_contains() Operator	1-11
Performing Simple Keyword Searches	1-12
Performing Boolean Searches	1-14
Routines Defined for the DataBlade Module.	1-16
Word Lists	1-16
Synonym Lists	1-16
Stopword Lists	1-19
Character Sets	1-20
Built-In Character Sets	1-21
User-Defined Character Sets	1-21
Highlighting	1-22
The etx_GetHilite() Function	1-23
The etx_HiliteType Data Type.	1-24

T

he Excalibur Text Search DataBlade module is a collection of data types and routines that extends Informix Dynamic Server with Universal Data Option to enable you to search your data in ways that are faster and more sophisticated than the keyword matching that SQL provides. Excalibur text search capabilities include phrase matching, exact and fuzzy searches, compensation for misspelling, and synonym matching.

This chapter provides an overview of the Excalibur Text Search DataBlade module. It explores the concept of a fuzzy text search, the key component of the text DataBlade module. This chapter also defines the terms and explains the concepts you need to set up and perform text searches. The following topics are covered:

- Text search concepts and terminology
- Overview of the **etx** access method
- Overview of creating an **etx** index
- Basics of performing keyword and Boolean searches
- Word list concepts and use
- User-defined character sets concepts and use
- Overview of highlighting the clue in the search text

Although this chapter does contain examples, it does not contain any step-by-step instructions that explain how to use the Excalibur Text Search DataBlade module. Instead, these instructions are contained in [Chapter 3, “Tutorial.”](#) Informix recommends that you learn the concepts that are explained in this chapter before moving on to Chapter 2.

Conceptual Overview

Under traditional relational database systems, you are limited to using LIKE and MATCHES when searching for words or phrases in a column that contains text data. For example, if you want to return all rows of the table **videos** in which the VARCHAR **description** column contains the phrase `multimedia text editor`, you are limited to executing a statement such as the following:

```
SELECT * FROM videos
WHERE description LIKE '%multimedia text editor%';
```

Although this SELECT statement returns a correct list of rows, it is probably very slow, depending on the size of the table and the amount of text data in the **description** column. Since a traditional secondary index on the column **description** does not help in this type of search, the whole table has to be scanned. Rows in the table that contain the word `multimedia` consistently misspelled as `mulitmedia`, for example, are not returned, although they are probably of interest to you.

If you want to also find documents that contain synonyms or alternate spellings of the words you are searching for, you must construct a complicated statement that contains many ORs in the WHERE clause, such as the following:

```
SELECT * FROM video
WHERE description LIKE '%multimedia text editor' OR
description LIKE '%multi-media text editor%' OR
description LIKE '%multimedia document editor%' OR
description LIKE '%multi-media document editor%';
```

This type of SELECT statement is often extremely slow. In sum, traditional relational database systems are not constructed to perform sophisticated and fast searches of this type.

The Excalibur Text Search DataBlade Module

The Excalibur Text Search DataBlade module solves this problem by dynamically linking in the Excalibur class library, or text search engine, to perform the text search section of the SELECT statement, instead of having the database server perform a traditional search. The text search engine is specifically designed to perform sophisticated and fast text searches. It runs in one of the database server-controlled virtual processes.

When you execute searches with the Excalibur Text Search DataBlade module, instead of using LIKE or MATCHES in the SELECT statement, you use an operator called **etx_contains()** that is defined for the DataBlade module and that instructs the database server to call the text search library of functions to perform the text search. This operator takes a variety of parameters to make the search more detailed than one using LIKE.

With the Excalibur Text Search DataBlade module you can create a specialized type of secondary index called **etx** on the column to be searched that will index the text data, resulting in much faster searches.

How the DataBlade Module Works

Unlike the searches performed using LIKE and MATCHES, a search of an **etx** index is not performed by Universal Data Option's standard access method, B-tree. Instead, when the server receives a request to process a query that uses the **etx_contains()** operator, it relays that part of the query to the Excalibur text search engine. One can think of the Excalibur text search engine as a specialized part of Universal Data Option that is executed when the **etx_contains()** operator is used in a SELECT statement.

The text search engine produces a group of rows that satisfy the conditions you set in the **etx_contains()** operator and returns them to the database server. If the query contains criteria other than those specified by the **etx_contains()** operator, Universal Data Option further qualifies the set of rows. Finally, Universal Data Option returns a list of rows that satisfy all the conditions of the WHERE clause. [Figure 1-1](#) illustrates this process.

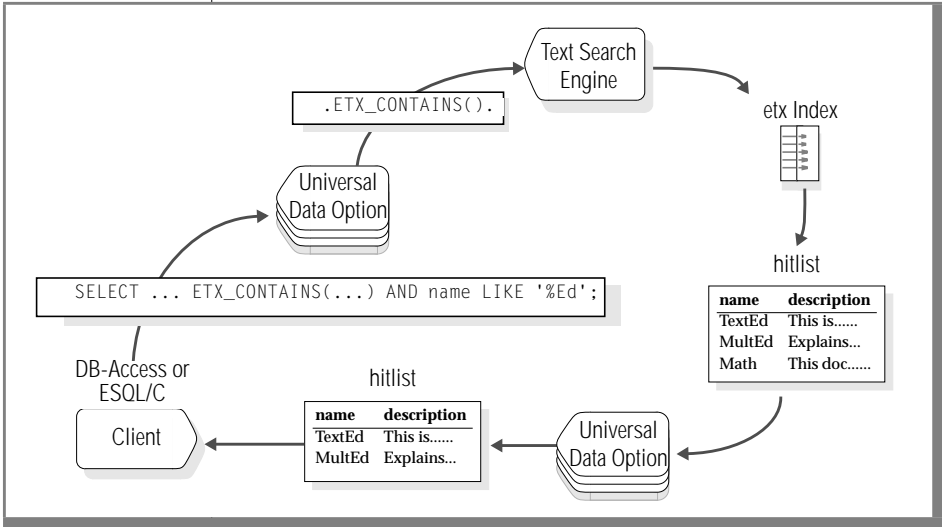


Figure 1-1
Text Search
Process Illustrated

Even though, for illustrative purposes, the preceding figure shows the Excalibur text search engine as a separate process, it is important to remember that it is actually dynamically linked to, and considered part of, Universal Data Option.

The following sections describe the components of the Excalibur Text Search DataBlade module and explain how it works in more detail.

Components of the Excalibur Text Search DataBlade Module

The Excalibur Text Search DataBlade module has three principal components: the **etx** access method, the **etx_contains()** operator, and the supporting routines defined for the DataBlade module. Each of the components is described in detail in the following sections.

The examples in this chapter are based on the **videos** table, a simple table with three columns, as defined in the following example:

```
CREATE TABLE videos
(
    id            INTEGER,
    name          VARCHAR(30),
    description    CLOB
);
```

The etx Access Method

The **etx** access method allows you to call on the Excalibur Text Retrieval Library to create indexes that support sophisticated searches on table columns that contain text. The indexes that you create with the **etx** access method are called **etx** indexes.

Data Types

To take advantage of the **etx** access method, you must store the data you want to search—called search text—in a column of type BLOB, CLOB, LVARCHAR, CHAR, VARCHAR, IfxDocDesc, or IfxMRData.

If most of your documents are large (over 2 KB), Informix recommends you store them in columns of type BLOB or CLOB, both of which are examples of smart large objects. If your documents contain binary data, such as proprietary formatting information from word processing programs such as Microsoft Word, store them in columns of type BLOB. If your documents contain only standard ASCII character data, such as HTML pages, store them in columns of type CLOB.

If all your documents are smaller than 2 KB and they do not contain any binary data, consider storing the documents in columns of type `LVARCHAR`.

Columns of type `CHAR` or `VARCHAR` are limited in the amount of information they can hold and will probably not be adequate for large documents. Columns of these two data types are useful for descriptive information such as titles and names. However, since it is possible to create **etx** indexes on columns of these types, already existing tables that contain legacy information stored in `CHAR` or `VARCHAR` columns can still be indexed without having to migrate the information.

The last two data types in this list, `IfxDocDesc` and `IfxMRData`, are types that Informix designed specifically for use with text access methods, **etx** in this particular case. By using the `IfxDocDesc` data type, you can store documents in either the database or in the operating system file system. `IfxMRData` is a multirepresentational data type, which means that the data type itself decides whether to store your document as an `LVARCHAR` or as a smart large object, depending on the initial size of the document.

For more detailed information on the `IfxDocDesc` and `IfxMRData` data types, refer to [Chapter 4, “Data Types.”](#) For a comparison of the advantages and disadvantages of using each data type, refer to [Chapter 7, “DataBlade Module Administration.”](#)

Operator Classes

When creating an **etx** index, you must specify the *operator class* defined for the data type of the column being indexed. An operator class is a set of functions that Universal Data Option associates with the **etx** access method for query optimization and building of the index. Each of the seven data types that supports an **etx** index has a corresponding operator class that must be specified when creating the index. The following table lists each data type and its corresponding operator class.

Data Type	Operator Class
BLOB	etx_blob_ops
CLOB	etx_clob_ops
LVARCHAR	etx_lvarc_ops



Data Type	Operator Class
CHAR	etx_char_ops
VARCHAR	etx_varc_ops
IfxDocDesc	etx_doc_ops
IfxMRData	etx_mrd_ops

Important: You must always specify an operator class when creating an **etx** index, even though each supported data type has only one operator class defined for it. Be sure to specify the correct operator class for the data type of the column being indexed by consulting the preceding table.

The following section shows an example of specifying an operator class when creating an **etx** index.

The Filtering Capability

Filtering refers to the process of stripping away all the proprietary formatting information from a document so that only its content remains in ASCII format.

For example, Microsoft Word documents usually contain formatting information that describes the fonts, paragraph styles, character styles, and layout of the text. Although this information can be indexed, it is not really useful for users who want to search the content of the document. Its inclusion in an **etx** index can significantly increase the size of the index and affect the performance of text searches. Filtering removes all this information and leaves just standard ASCII text.

Informix supports a number of filters for specific file formats. Since the products that produce each file format, such as Microsoft Word, are constantly being upgraded, the full list of supported filters often changes.

For the most current and complete list of the filters supported by the Excalibur Text Search DataBlade module, refer to the on-line release notes file **SINFORMIXDIR/extend/ETX.<version>/ETXREL.TXT**, where **<version>** refers to the latest version of the DataBlade module installed on your computer.

Creating an etx Index

To create an **etx** index, you specify the **etx** access method in the USING clause of the CREATE INDEX statement. For example, suppose your search text is contained in the column **description**, of type CLOB, in the **videos** table. To create an **etx** index named **desc_idx** for this table that is stored in the sbspace **sbsp1**, use the following syntax:

```
CREATE INDEX desc_idx ON videos (description etx_clob_ops)
      USING etx in sbsp1;
```

An *sbspace* is a logical storage area that contains one or more chunks that only store BLOB and CLOB data types. The operator class **etx_clob_ops** is specified directly after **description**, the column to be indexed. Be sure to always specify the correct operator class for the type of the indexed column. Refer to the table in the previous section for valid operator class names and their corresponding data types.

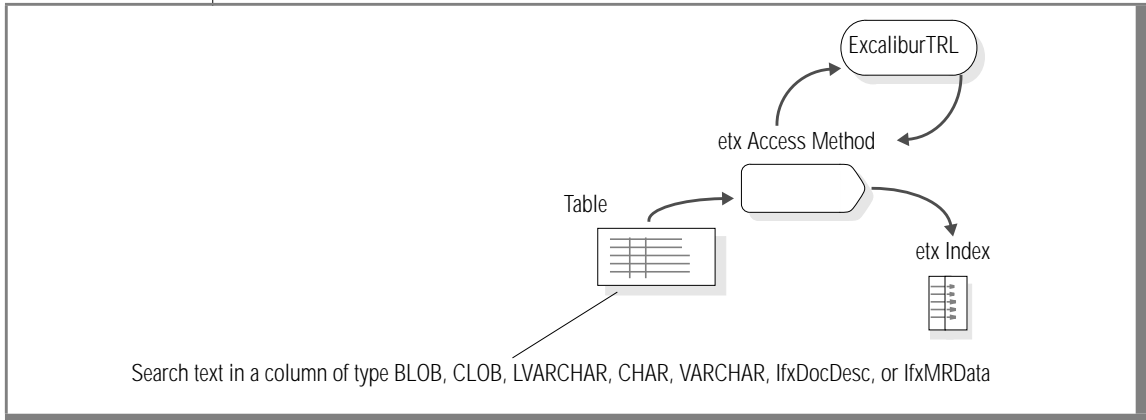
Refer to [Informix Guide to SQL: Syntax](#) for detailed information on the CREATE INDEX statement. Refer to the administrator's guide for your database server for detailed information on sbspaces and smart large objects such as BLOBs and CLOBs.



Warning: The following options to the CREATE INDEX statement are not supported for **etx** indexes: *CLUSTER*, *UNIQUE*, *DISTINCT*, *ASC*, *DESC*, and *FILLFACTOR*.

Figure 1-2 illustrates the process of creating an **etx** index.

Figure 1-2
etx Index Creation



The **etx** access method supports both table and index fragmentation. For more information on fragmentation support, see [Chapter 3, “Tutorial.”](#)

Specifying etx Index Parameters

An *index parameter* is a variable that you use to specify the characteristics of an index based on the searches you plan to perform. You set an index parameter in the **USING** clause of a **CREATE INDEX** statement when you create an **etx** index.

For example, suppose you want to create an index identical to the index created in the previous example, but with one difference: you want to index international characters as well as the standard ASCII characters. To do this, you must create the index specifying the ISO character set via the index parameter **CHAR_SET**.

The following example shows how to create this type of index:

```
CREATE INDEX desc_idx1 ON videos (description etx_clob_ops)
  USING etx (CHAR_SET = 'ISO') IN sbasp1;
```

By creating indexes based on likely search characteristics, you can improve the performance of your searches and reduce the size of your indexes.

You cannot alter the characteristics of an **etx** index after you create it. Instead, you must drop the index and then create it again with the new or altered characteristic.

For more detailed information on index parameters, see [Chapter 6, “etx Index Parameters.”](#)

For more examples on how to create custom indexes, see [Chapter 3, “Tutorial.”](#)

The `etx_contains()` Operator

You use the **etx_contains()** operator to perform searches of **etx** indexes. This section explains how to perform simple searches such as *keyword* and *Boolean searches* using the **etx_contains()** operator. These and other more complex types of searches are described in more detail in [Chapter 2, “Text Search Concepts.”](#)

Performing Simple Keyword Searches

The **etx_contains()** operator is defined for the DataBlade module to allow you to define, qualify, and fine-tune searches of text. The **etx_contains()** operator takes three arguments, the first two required, the third optional. Use the first argument to specify the column that contains your search text. Use the second argument to specify what you are searching for, called a *clue* or a *search string*. The third optional argument is a *statement local variable* (SLV), used to return scoring information from a search.

To perform a search, use the **etx_contains()** operator in the WHERE clause of a SELECT statement. Suppose that you want to search the column **description** for the word `multimedia` using the index created in the previous section. The following statement performs this search:

```
SELECT id, description FROM videos
      WHERE etx_contains(description, Row('multimedia'));
```

This example illustrates what is called a *keyword search*, the default search type. This example is illustrated in [Figure 1-3](#). The top section of the figure shows a table and its contents, and the bottom section shows the results of the search.

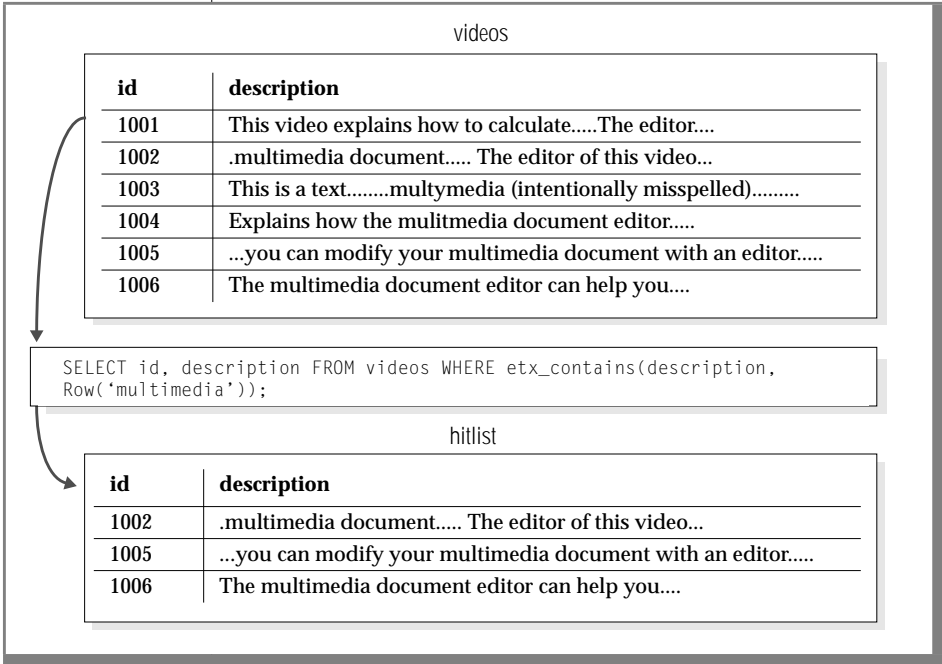


Figure 1-3
Simple Keyword
Search Illustrated

Note that the example created the clue `multimedia` by using the **Row()** constructor. If you do not specify any tuning parameters, the **Row()** constructor in the **etx_contains()** operator is optional. This means that the preceding example can also be specified as:

```
SELECT id, description FROM videos  
WHERE etx_contains(description, 'multimedia');
```

For more information on the **Row()** constructor, see the [Informix Guide to SQL: Syntax](#) manual.

Performing Boolean Searches

You perform a Boolean search when you use any Boolean expression in a text search. For example, to search for text in the **description** column that contains the words **multimedia** and **editor** but not the word **video**, execute the following SQL statement:

```
SELECT id, description FROM videos
WHERE etx_contains (description,
Row('multimedia & editor & !video',
'SEARCH_TYPE = BOOLEAN_SEARCH'));
```

In this example, the **&** symbol is used to represent a logical AND operator. You can also use the **|** symbol to represent a logical OR and the **!** or **^** symbols to represent the logical NOT. [Figure 1-4](#) displays the qualifying rows, also called a *hitlist*, when the query in the example is run on the **videos** table.

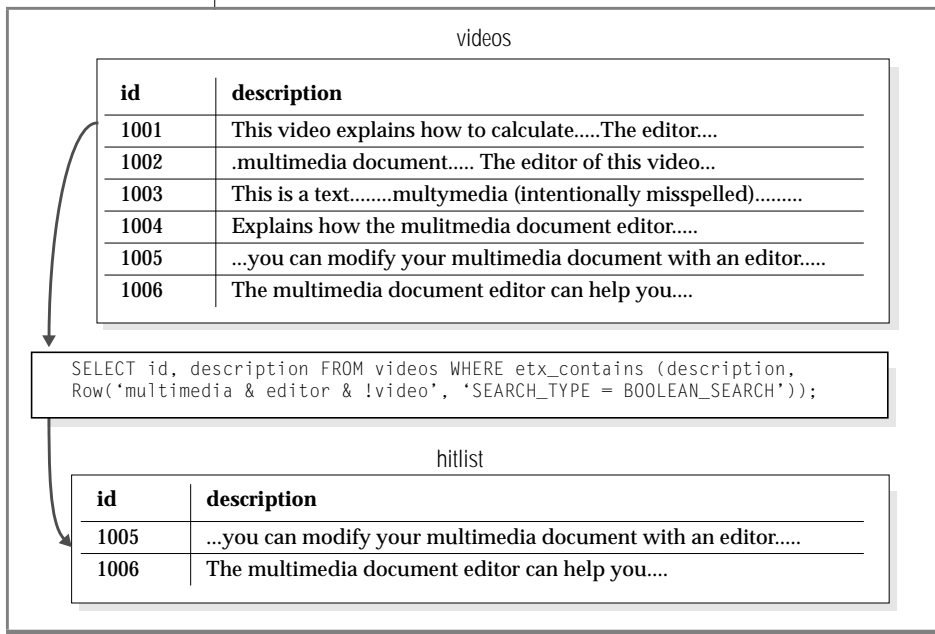


Figure 1-4
Sample Text Search
Query that Uses a
Boolean Expression

You can use the third, optional argument of the **`etx_contains()`** operator to pass an SLV to the search engine. The search engine uses this variable to record the score it assigns to each row in the hitlist and to record highlighting information. The data type of the SLV is `etx_ReturnType`, an Informix-defined row type that consists of two fields that together contain the scoring and highlighting information.

See [“Ranking the Results of a Fuzzy Search” on page 2-12](#) for more information using the scoring information of the SLV. See [“Character Sets” on page 1-20](#) for more information on using the highlighting information of the SLV.

For more information on the `etx_ReturnType` data type, its two fields, and how to use the information contained in the fields, see [Chapter 4, “Data Types.”](#) For more general information on SLVs, see the [Informix Guide to SQL: Syntax](#).

For more detailed information on the types of searches you can execute with the Excalibur Text Search DataBlade module, see [Chapter 2, “Text Search Concepts.”](#)

[Figure 1-5](#) provides a summary of the **`etx_contains()`** syntax used thus far.

Figure 1-5
Summary of `etx_contains` Syntax

```
CREATE INDEX desc_idx ON videos (description etx_clob_ops) USING etx();
```

Column of type BLOB, CLOB, LVARCHAR,
CHAR, VARCHAR, lfxDocDesc, or lfxMRData

```
SELECT * FROM videos WHERE etx_contains(description, Row('multimedia'));
```

Indexed column

Search string

Routines Defined for the DataBlade Module

In addition to the **etx_contains()** operator, Informix supplies several routines defined for the DataBlade module that you can use to perform tasks such as creating and dropping synonym and stopword lists.

An example of one of these routines is **etx_CreateSynWlst()**. This routine allows you to create a list of words that you want the search engine to treat as synonyms. To use this procedure to create a synonym list, you use the EXECUTE PROCEDURE statement as shown in the following example:

```
EXECUTE PROCEDURE etx_CreateSynWlst  
('syn_list', '/local0/excal/synonyms','sbsp1');
```

This statement creates a synonym list named **syn_list** from an operating system file called **/local0/excal/synonyms** and stores it in the sbspace called **sbsp1**.

For a complete list of Excalibur Text Search DataBlade module routines and the functions they perform, see [Chapter 5, “Routines.”](#)

Word Lists

The Excalibur Text Search DataBlade module supports two types of word lists: synonym lists and stopword lists. Although you can perform successful searches without word lists, using them can be beneficial in ways that are explained in the following sections.

Synonym Lists

A *synonym list* consists of a *root word* and one or more words whose meaning is similar to the root word. Synonym lists are useful when you do not know the exact content of the text you are searching.

For example, suppose you want to search the **description** column of the **videos** table. You know that the **description** column contains references to videos that explain how to use multimedia document editors, but you are not sure if the editors are consistently described as multimedia *document* editors or multimedia *text* editors.

You can resolve this problem with a synonym list entry for the words **document** and **text**.

To instruct the search engine to use this word list when you perform a search, you use the `MATCH_SYNONYM` tuning parameter, as shown in the following example:

```
SELECT id, description from videos
WHERE etx_contains(description,
Row('document', 'MATCH_SYNONYM = syn_list'));
```

You can maintain multiple synonym lists within a database. You specify which synonym list you want the search engine to use by setting `MATCH_SYNONYM` to the name of the synonym list. For example, to make **syn_list2** the active synonym list while querying the **videos** table, execute the following statement:

```
SELECT id, description from videos
WHERE etx_contains(description,
Row('document', 'MATCH_SYNONYM = syn_list2'));
```

The Excalibur Text Search DataBlade module includes a default synonym list that contains many common English-language synonyms. This default synonym list is created during the registration of the DataBlade module and is called **etx_thesaurus**.

If you specify the `MATCH_SYNONYM` tuning parameter in the **etx_contains()** operator but do not set it equal to a value, the default synonym list is consulted. If a value for the `MATCH_SYNONYM` parameter is specified, the specified synonym list is used instead of the default list.

For more information on viewing and changing the default synonym list, refer to [“etx_CreateSynWlst\(\)” on page 5-23](#).

To create a custom synonym list, first create an operating system file that contains root words with one or more synonyms, all on one line and separated by blanks. The lines of text must be separated by one blank line. For example, the following is a possible excerpt from an operating system file that will be used to create a synonym list:

```
quick speedy fast

monitor terminal CRT screen
```

Then use the routine **etx_CreateSynWlst()** defined for the DataBlade module to make the synonym list known to the Excalibur Text Search DataBlade module.

In this example, `quick` and `monitor` are the root words. A word must be present as a root word for the synonyms of the word to be used. This means that if you want to search for synonyms of `speedy`, it must itself be listed as a root word; it is not enough to simply exist as a synonym for `quick` in the synonym list.



Important: Be sure to include the extra blank line between the lines of text in the operating system synonym file, as described above. If you omit the blank lines, the DataBlade module will not return an error, but it will never find any synonyms during a synonym matching search.

The default synonym list **etx_thesaurus** is created using the operating system file:

```
$INFORMIXDIR/extend/ETX.<version>/wordlist/etx_thesaurus.txt
```

In the preceding filename, `<version>` refers to the current version of the DataBlade module installed on your computer. This file contains many standard English- language synonyms. You may consult this file when you create your own custom synonym lists.



Important: The Excalibur Text Search DataBlade module finds pattern matches of root words only, not of their synonyms.

For example, assume you execute a text search and specify the `PATTERN_ALL` and `MATCH_SYNONYM` tuning parameters and that the specified synonym list contains the root word `abandon` with one synonym: `surrender`. The search returns documents that contain the word `abanden`, a pattern match of the root word `abandon`, but does not return documents that contain the word `surender`, a pattern match of the synonym `surrender`.

The `PATTERN_ALL` tuning parameter is discussed in [Chapter 2, “Text Search Concepts.”](#)

For more information on how to create and drop synonym lists, see [Chapter 5, “Routines.”](#)

Stopword Lists

A *stopword* is a word that you want excluded from your index and, as a consequence, from your searches. A typical stopwords list includes words like *of*, *the*, and *by*. You will find, however, that stopwords lists depend on the content and type of your data. Any frequently occurring word that you want excluded from your index is a candidate for inclusion in a stopwords list. Stopword lists can reduce the time it takes to perform a search, reduce index size, and help you avoid false hits.

You create and drop stopwords lists using procedures defined for the DataBlade module. For example, to create a stopwords list, first create an operating system file that contains the list of stopwords, one word per line. Then make the stopwords list known to the Excalibur Text Search DataBlade module by executing the procedure **etx_CreateStopWlst()**, as shown in the following example:

```
EXECUTE PROCEDURE etx_CreateStopWlst
('stopwlist', '/local0/excal/stopwlist');
```

This statement creates the stopwords list **stopwlist** from the operating system file **/local0/excal/stopwlist**. An optional third argument can be used to specify the sbpace where the list is to be stored. If you don't specify a specific sbpace to store the list, it is stored in the default sbpace. The default sbpace is specified by the SBSPACENAME parameter in the ONCONFIG file.

The Excalibur Text Search DataBlade module includes an operating system file that contains a list of standard English-language stopwords. This file is called:

```
$INFORMIXDIR/extend/ETX.<version>/wordlist/etx_stopwords.txt
```

In the preceding filename, *<version>* refers to the current version of the DataBlade module installed on your computer. You can consult this file when you create your own custom stopwords list.

You can have at most one stopwords list associated with an **etx** index. The stopwords list is specified when the index is initially created via the index parameter **STOPWORD_LIST**. This means that the stopwords list must already exist when the **etx** index is created.

At times, you might want to include words in a search that currently exist in your stopwords list. For example, suppose that the following words exist in your stopwords list: `to`, `or`, and `be`. Suppose further that you want to search for the exact phrase `to be or not to be`. To occasionally search for stopwords using an **etx** index that has a stopwords list associated with it, specify the `INCLUDE_STOPWORDS` index parameter when you create the index.

Then use the `CONSIDER_STOPWORDS` tuning parameter when executing the search, which forces the search engine to include words that you previously stipulated as stopwords. For example, you can search for the phrase `to be or not to be` as follows:

```
SELECT id, description FROM videos
WHERE etx_contains(description,
  Row('to be or not to be',
    'SEARCH_TYPE = PHRASE_EXACT & CONSIDER_STOPWORDS'));
```



Important: The `CONSIDER_STOPWORDS` tuning parameter of the **etx_contains()** operator works only if the `INCLUDE_STOPWORDS='TRUE'` index parameter was specified at **etx** index creation.

For more information on how to create and drop stopwords lists, see [Chapter 5, “Routines.”](#)

Character Sets

Character sets define the characters that are indexed when an **etx** index is built on a column that contains text documents. Any character in the text document not found in the specified character set is treated as white space in the index. The character is not changed in the text document itself.

Specify a character set by setting the `CHAR_SET` index parameter to the character set's name when you create an **etx** index with the `CREATE INDEX` statement.

The Excalibur Text Search DataBlade module provides three built-in character sets: `ASCII`, `ISO`, and `OVERLAP_ISO`. You can also define your own character set if the ones provided are not adequate for your text documents. The following sections describe the built-in character sets and explain when and how to create your own.

Built-In Character Sets

The ASCII character set includes the numbers 0 through 9 and the uppercase letters A through Z, and the lowercase letters a through z. All lower case letters are mapped internally to uppercase, which means that searches that use the ASCII character set are case-insensitive. ASCII is the default character set that is used if you do not specify the CHAR_SET index parameter in the CREATE INDEX statement.

The ISO and OVERLAP_ISO character sets extend the ASCII character set by including many international characters such as Å and ñ. The OVERLAP_ISO character set maps similar international characters to a single ASCII character.

For more information on the three built-in character sets, refer to [Chapter 6](#), “etx Index Parameters,” and [Appendix A](#), “Character Sets.”

User-Defined Character Sets

At times, the built-in character sets might be inadequate for your text documents or the types of searches you plan to perform. For example, you might want to index the hyphen character to be able to index and search for hyphenated words such as `English-language`. Since the three built-in character sets index only alphanumeric characters, you must create your own character set to index the hyphen character.

You create and drop user-defined character sets with the routines **etx_CreateCharSet()** and **etx_DropCharSet()** provided by the DataBlade module. You must create a user-defined character set before you use it to create an **etx** index.

The **etx_CreateCharSet()** routine takes two parameters: the name of the new user-defined character set and the full pathname of an operating system file that contains a description of the character set. The operating system file contains a 16 X 16 matrix of hexadecimal numbers that represents which characters should be indexed.

The following example shows how to execute the **etx_CreateCharSet()** routine to create a new user-defined character set called **my_charset** from the description contained in the operating system file named **/local0/excal/my_char_set_file**:

```
EXECUTE PROCEDURE etx_CreateCharSet
    ('my_charset', '/local0/excal/my_char_set_file');
```

The following example shows how to create an **etx** index that uses the user-defined character set **my_charset** by specifying it as an option to the **CHAR_SET** index parameter:

```
CREATE INDEX desc_idx2 ON videos (description etx_clob_ops)
    USING etx (WORD_SUPPORT = 'PATTERN', CHAR_SET = 'my_charset')
    IN sbsp1;
```

For more detailed information on the structure of the operating system file and instructions for using the **etx_CreateCharSet()** routine, refer to [Chapter 5, “Routines,”](#) and [Appendix A, “Character Sets.”](#)

For more information on the **CHAR_SET** index parameter, refer to [Chapter 6, “etx Index Parameters.”](#)

Highlighting

In addition to searching for a text document that contains a specified clue, you might also want to know where in the document the clue appears. This process of pinpointing the location of a clue in a search text is called *highlighting*.

Use the **etx_GetHilite()** function, specifically defined for the Excalibur Text Search DataBlade module, to return highlighting information from a text search. This function can be executed only in the **SELECT** list of a query, as shown in the following example:

```
SELECT etx_GetHilite (description, rc) FROM videos
    WHERE etx_contains(description,
        'multimedia', rc # etx_ReturnType);
```


This query returns all documents that contain the keyword `multimedia` in the **description** column of the table **videos**. In addition, for each row returned, the query also returns information about the location of every instance of the word `multimedia` in the corresponding document.

The returned highlighting information consists of location offsets and text documents contained in the two fields of the `etx_HiliteType` row data type, the return value of the **`etx_GetHilite()`** function. The `etx_HiliteType` data type is described in [“The `etx_HiliteType` Data Type” on page 1-24](#).

The `etx_GetHilite()` Function

The **`etx_GetHilite()`** function takes two required parameters. The first parameter is the name of the column that contains the document for which you want highlighting information. In the previous example, the column is called **description**. The second parameter is the name given to the statement local variable (SLV) of the **`etx_contains()`** operator that executes the search. In the example, the SLV is called **rc**.

The string to be highlighted is not specified in the **`etx_GetHilite()`** function. Instead, the clue specified in the **`etx_contains()`** operator is used. In the example, the word `multimedia` is the highlight string.

Since **`etx_GetHilite()`** is a function, it returns a value. The data type of the returned value is `etx_HiliteType`, a row data type defined by the Excalibur Text Search DataBlade module that contains the highlighting information. The next section discusses the `etx_HiliteType` row data type in more detail.

For more detailed information on the **`etx_GetHilite()`** function, refer to [Chapter 5, “Routines.”](#)



Important: If you use the **`etx_GetHilite()`** function in a query that returns more than one row, the function executes once for each row. This means that each row will have its own highlighting information, contained in the `etx_HiliteType` row data type returned by the **`etx_GetHilite()`** function. This highlighting information pertains only to the document contained in the row, not to any other document.

The *etx_HiliteType* Data Type

The *etx_HiliteType* row data type consists of two fields: **vec_offset** and **viewer_doc**. The **vec_offset** field contains pairs of integers that describe the location of every instance of the highlight string in the document. The **viewer_doc** field contains the text document itself.

Use the information returned by the **etx_GetHilite()** function in your applications to actually highlight the clue in the document. For example, your application might use the location integer pairs stored in the **vec_offset** field to insert the HTML tags `<U>` and `</U>` before and after the clue in the text document stored in the **viewer_doc** field. With these tags in place, all instances of the clue will appear underlined when viewed through a browser such as Netscape Navigator.

Since the data types of the **vec_offset** and **viewer_doc** fields are CLOB and BLOB, respectively, your application must use standard Informix ESQL/C large object functions such as **ifx_lo_open()** and **ifx_lo_read()** or DataBlade API large object functions such as **mi_lo_open()** and **mi_lo_read()** to manipulate the contents.

For more information on the ESQL/C functions available to manipulate CLOB and BLOB data types, refer to the [INFORMIX-ESQL/C Programmer's Manual](#). For more information on the DataBlade API functions, refer to the [DataBlade API Programmer's Manual](#).

For more information on the *etx_HiliteType* row data type and the information contained in its fields **vec_offset** and **viewer_doc**, refer to [Chapter 4](#), “Data Types.”

Text Search Concepts

Excalibur Text Search Types	2-3
Keyword Search	2-3
Boolean Search	2-5
Phrase Search	2-6
Exact Phrase Search	2-6
Approximate Phrase Search	2-8
Proximity Search	2-8
Performing a Fuzzy Search	2-10
Substitution and Transposition	2-10
Ranking the Results of a Fuzzy Search	2-12
Limiting the Number of Rows a Fuzzy Search Returns	2-14
Pattern Search	2-15
Phrase Searching with Pattern Matching	2-17

T

his chapter discusses the different types of searches that can be performed with the Excalibur Text Search DataBlade module.

Excalibur Text Search Types

The Excalibur Text Search DataBlade module supports a variety of searches, such as keyword, Boolean, proximity, pattern (or fuzzy), and phrase. Phrase searching can be further classified into two types: exact and approximate.

Use the `SEARCH_TYPE` *tuning parameter* to indicate what kind of search you want to perform. A tuning parameter is a variable that you use to guide the way in which the text search engine performs a search. You pass this parameter to the search engine as the second input parameter of the `Row()` constructor of the `etx_contains()` operator.

The following sections provide examples of using tuning parameters to perform different types of searches. The examples use the same `videos` table described in Chapter 1. Refer to [Chapter 5, “Routines,”](#) for a full list of supported tuning parameters.

Keyword Search

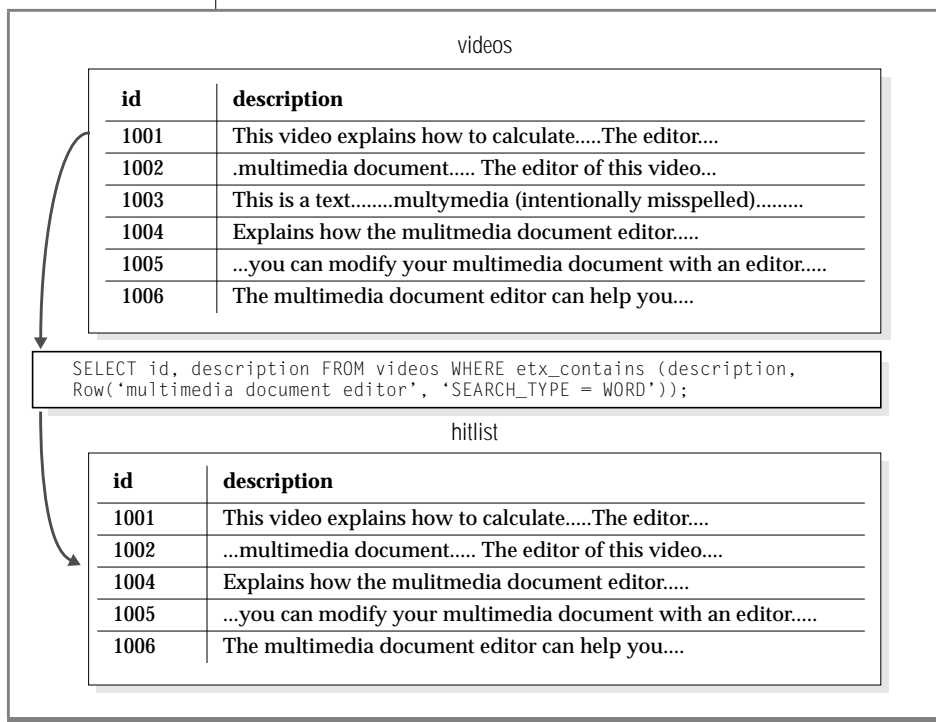
When the clue you specify contains more than one word, you can direct the search engine to treat each word as a separate entity. This type of search is called a *keyword search*. When the text engine performs a keyword search, it returns a row whenever it encounters one or more of the words in your clue.

You specify a keyword search by setting the `SEARCH_TYPE` tuning parameter to `WORD` and passing it to the search engine as the second parameter of the `Row()` constructor. For example, consider the following keyword search for the words `multimedia`, `document`, and `editor`:

```
SELECT id, description FROM videos
WHERE etx_contains (description,
Row('multimedia document editor', 'SEARCH_TYPE = WORD'));
```

This query instructs the search engine to return any row that contains one or more occurrences of `multimedia`, `document`, or `editor` in the `description` column. The [Figure 2-1](#) illustrates this example.

Figure 2-1
Example of
Keyword Search



The search did not return the row with ID 1003 because the word `multimedia` is misspelled and the text does not contain the other two words in the clue. Even though the word `multimedia` is misspelled in the row with ID 1004, the row is still returned because it contains the other two words in the clue, `document` and `editor`.

The score of the results of a keyword search is based on the number of keywords found in the documents. For example, a document that contains two of three keywords would be scored twice as high as a document that contains only one of the keywords.

If you do not specify the `SEARCH_TYPE` tuning parameter in the **`etx_contains()`** operator, the text search engine defaults to keyword search. This means that the following two searches are equivalent and are therefore often used interchangeably in this manual:

```
SELECT id, description FROM videos
WHERE etx_contains (description,
Row('multimedia document editor'));

SELECT id, description FROM videos
WHERE etx_contains (description,
Row('multimedia document editor', 'SEARCH_TYPE = WORD'));
```

Boolean Search

Boolean searches are similar to keyword searches in that both search for words as separate entities. A Boolean search additionally allows you to combine the keywords in a Boolean expression to create more complicated clues.

The preceding example shows how to perform a keyword search for the words `multimedia document editor`. Documents that contain at least one of the keywords are returned. If you want to specify that you want documents that contain both the words `multimedia` and `document` but not the word `video`, you have to execute a Boolean search, as shown in the following example:

```
SELECT id, description FROM videos
WHERE etx_contains (description,
Row('multimedia & editor & !video',
'SEARCH_TYPE = BOOLEAN_SEARCH'));
```

The Boolean operators `&` and `!` are used to build the desired Boolean expression of keywords. You specify a Boolean search by setting the `SEARCH_TYPE` tuning parameter to `BOOLEAN_SEARCH`.

When you specify a Boolean search by using the `SEARCH_TYPE = BOOLEAN_SEARCH` tuning parameter, you must include at least one Boolean operator in the clue, or the Excalibur Text Search DataBlade module will return an error. Use parentheses for complex expressions.

Phrase Search

In the context of the Excalibur Text Search DataBlade module, a *phrase* is defined as any clue that contains more than one word. In a phrase search, the text search engine does not treat the words in the clue as separate entities. Instead, it treats the whole phrase as a single unit.

Excalibur text search supports two types of phrase searches: exact phrase search and approximate phrase search.



Important: *Exact and approximate phrase searches can be performed only on columns whose **etx** indexes were created with the `PHRASE_SUPPORT` index parameter. If you try to perform a phrase search on a column whose index was not created with this index parameter, the search will return unpredictable and possibly incorrect results.*

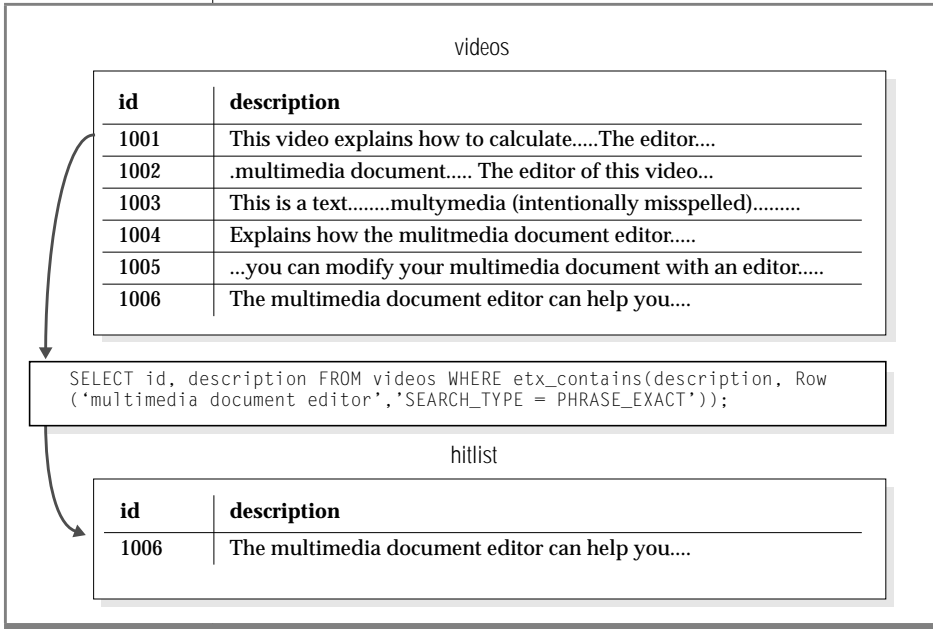
Exact Phrase Search

An exact phrase search is successful when the search engine finds a phrase that contains all the words in the clue in the exact order that you specify. To execute an exact phrase search, you set the `SEARCH_TYPE` tuning parameter to `PHRASE_EXACT`, as shown in the following example:

```
SELECT id, description FROM videos
WHERE etx_contains(description,
  Row ('multimedia document editor', 'SEARCH_TYPE = PHRASE_EXACT'));
```


Figure 2-2 displays the resulting hitlist when the query in the example is run on the **videos** table. Since an exact phrase search with no pattern matching is specified, the document that contains the phrase `multimedia document editor` (`multimedia` intentionally misspelled) is not returned.

Figure 2-2
Example of Exact
Phrase Search



An exact phrase search for a clue that contains a stopwords returns zero rows, even if the clue is contained in a document. This happens only if the **etx** index ignores stopwords, or in other words, if the index was created with the `STOPWORD_LIST` index parameter and the stopwords list contains one or more words in the clue.

For example, assume the **etx** index was created with the index parameter `STOPWORD_LIST = 'my_list'`, and that the stopwords list **my_list** includes the word `the`. In this case, an exact phrase search for the clue `walk the dog` will always return zero rows, even if this exact phrase is contained in a document.

There are two ways to work around this behavior:

- Use an approximate phrase search instead of an exact phrase search. Approximate phrase searches, however, can be slower than exact phrase searches.

- Include stopwords in the **etx** index by specifying the `INCLUDE_STOPWORDS` index parameter when you create the index. Then specify that stopwords be considered when executing an exact phrase search for the phrase `walk the dog` by specifying both the `CONSIDER_STOPWORDS` and `SEARCH = PHRASE_EXACT` tuning parameters in the **etx_contains()** operator.

Approximate Phrase Search

An approximate phrase search is successful when the search engine finds a phrase in the search text that contains words that are approximately the same as the words you specify in the clue. To execute an approximate phrase search, you set the `SEARCH_TYPE` tuning parameter to `PHRASE_APPROX`, as shown in the following example:

```
SELECT * FROM videos
WHERE etx_contains(description,
  Row ('document editor','SEARCH_TYPE = PHRASE_APPROX'));
```

Use an approximate phrase search when you want to search for a phrase, but you do not know or remember all the words in a phrase. Use an exact phrase search with pattern matching if you know all of the words in a phrase, but are not sure how one or more of the words in the phrase is spelled.



***Tip:** Informix recommends that you refrain from setting `SEARCH_TYPE` to `PHRASE_APPROX` and passing the `PATTERN_BASIC` or `PATTERN_ALL` tuning parameters for the same search. Doing so can result a large number of results, and hence, in poor performance for searches of this type. The tuning parameters `PATTERN_BASIC` and `PATTERN_ALL` are discussed in a later section.*

Proximity Search

When you perform a word search for a phrase that contains multiple words, the search engine returns any row that contains one or more of the words in the clue.

For example, suppose you want to search for the phrase multimedia document editor, as shown in the following example:

```
SELECT id, description FROM videos
WHERE etx_contains(description,
  Row('multimedia document editor',
    'SEARCH_TYPE = WORD'));
```

If you execute this search on the following row, the search engine reports a match, even though the sentence is probably not of interest to you (because it has nothing to do with a multimedia document editor):

```
The multimedia application is now in formal beta testing.
```

On the other hand, if you use an exact phrase search, the search engine fails to return rows similar to the following phrase:

```
The editor for multimedia documents .....
```

A proximity search resolves these problems by allowing you to specify the number of nonsearch words that can occur between two or more search words. For example, consider the following word search:

```
SELECT id, description FROM videos
WHERE etx_contains(description,
  Row('multimedia document editor',
    'SEARCH_TYPE = PROX_SEARCH(8)'));
```

The last line uses the PROX_SEARCH option to the SEARCH_TYPE tuning parameter to specify that the search engine is to return a row only when each of the three search words occurs within eight words of the last word matched. Suppose you executed this query on the following sample text:

```
The multimedia application is now in formal beta testing. A
text editor.....
```

In this example, the search engine does not return this row because it has more than eight words between multimedia and editor. However, the search engine does return the following row, because it has fewer than eight words between each of the search words:

```
The editor for multimedia documents .....
```

The following statement always fails to produce results because the value of `PROX_SEARCH` is fewer than the number of words to search for:

```
SELECT id, description FROM videos
WHERE etx_contains(description ,
Row('multimedia document editor',
'SEARCH_TYPE = PROX_SEARCH(2)'));
```

Performing a Fuzzy Search

Although the text search feature allows you to perform keyword searches for both words and phrases, the real power of the text search feature lies in its ability to perform fuzzy searches. A *fuzzy search* is a search for text that matches your clue closely instead of exactly. Fuzzy searches can also be referred to as *pattern searches*.

The following section discusses the simplest example of a fuzzy search: a search that takes into account misspellings in both the clue and the search text.



Important: *Fuzzy or pattern searches can be performed only on columns whose `etx` indexes were created with the `WORD_SUPPORT = PATTERN` index parameter.*

Substitution and Transposition

Misspellings most frequently occur when letters in a word are mistakenly substituted or transposed. Misspelling `editor` as `editer` is an example of substitution; misspelling `multimedia` as `mulitmedia` is an example of transposition. Transposition refers only to switching the order of two *adjacent* characters.

The tuning parameters that instruct the text search engine that you want transpositions and substitutions taken into account are `PATTERN_TRANS` and `PATTERN_SUBS`. Unlike the `SEARCH_TYPE` tuning parameter, `PATTERN_TRANS` and `PATTERN_SUBS` do not have values that you set. Instead, you pass these tuning parameters directly to the search engine using the Boolean operator `&`, as shown in the following example:

```
SELECT * FROM videos
WHERE etx_contains(description,
  Row('multimedia' , 'PATTERN_TRANS & PATTERN_SUBS')) ;
```

This example initiates a fuzzy search for the `multimedia` clue. The search engine returns all words that have exactly one substitution or one transposition, as well as words that match the clue exactly. [Figure 2-3](#) displays the resulting hitlist when the query in the example is run on the `videos` table.

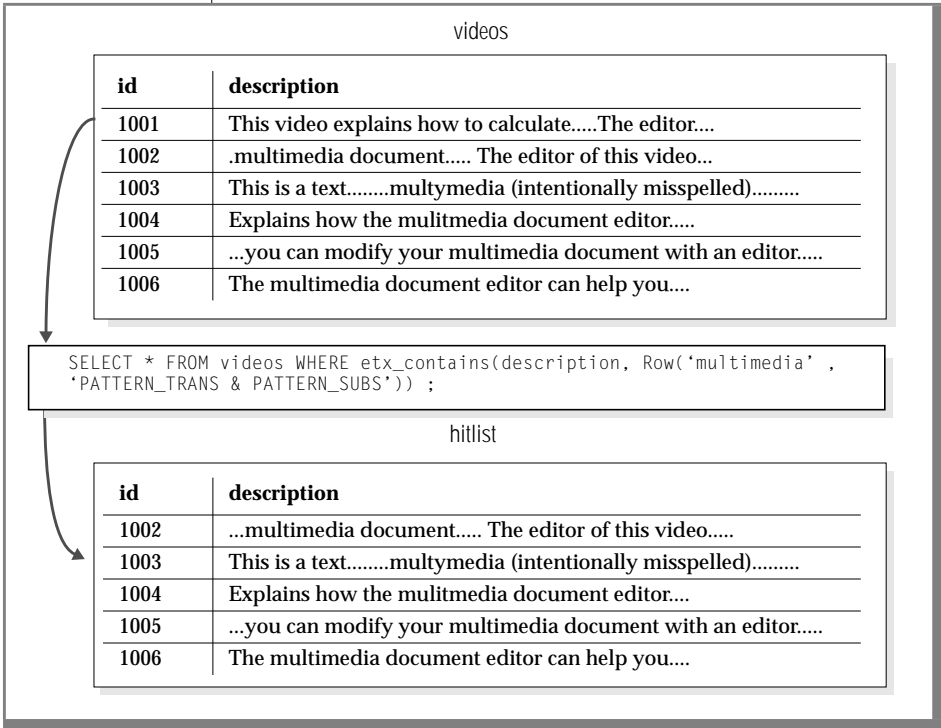


Figure 2-3
Sample Text Search
Query That Uses
Tuning Parameters

The tuning parameters `PATTERN_TRANS` and `PATTERN_SUBS` only take into account the transposition or substitution of a single letter per word.

Ranking the Results of a Fuzzy Search

When you perform a fuzzy search, some of the rows that the text search engine returns might satisfy search criteria better than others. To determine the degree of similarity between your clue and each of the rows returned, you can instruct the search engine to assign a value, called a *score*, to each of the rows.

Scores vary from 0 to 100, with 0 indicating no match and 100 indicating a perfect match. Values between 0 and 100 indicate approximate matches; the higher the value, the closer the match. A null value indicates that the row was not ranked. This could happen if the row was returned because of a non-**etx_contains()** operator in a WHERE clause that contains an OR predicate.

To access score information, use a statement local variable (SLV) as the optional third parameter to the **etx_contains()** operator. The data type of the SLV is **etx_ReturnType**, an Informix-defined row type that consists of two fields. The scoring information is contained in the **score** field.

For example, if **rc** is an SLV, you can use it to obtain score information, as shown in the following example:

```
SELECT rc.score, id, description FROM videos
WHERE etx_contains(description,
  Row('multimedia' ,
    'PATTERN_TRANS & PATTERN_SUBS'), rc # etx_ReturnType)
ORDER BY 1;
```

Figure 2-4 displays the resulting hitlist when the query in the example is run on the **videos** table. The column **score** in the hitlist contains scoring information.

The SLV has a scope that is limited to the statement in which you use it. It is a way for the Excalibur text search engine to send back information about the search it just performed to the **etx_contains()** operator that called it.

Although the data type of the SLV is always **etx_ReturnType**, you must still explicitly specify its type when using it in the **etx_contains()** operator, as shown in the example. The example also shows how you can use the **ORDER BY** clause to instruct the database server to rank the rows it returns by the **score** field of the SLV.

For more information on the `etx_ReturnType` data type, its two fields, and how to use the information contained in the fields, see [Chapter 4, “Data Types.”](#) For more general information on SLVs, see the [Informix Guide to SQL: Syntax](#).

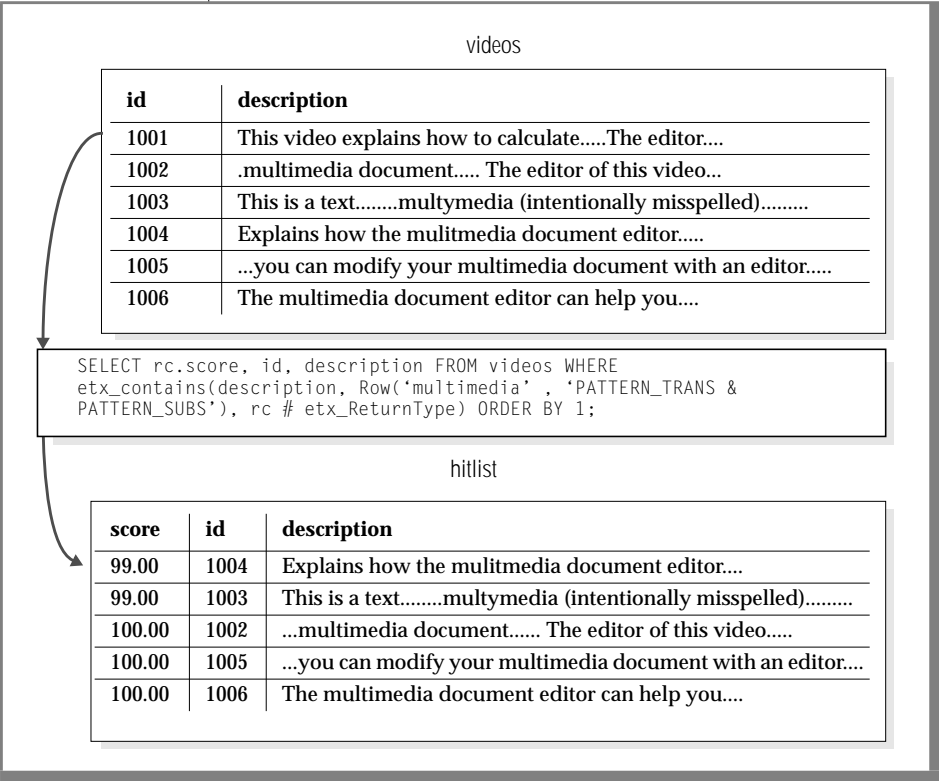


Figure 2-4
Sample Text Search
Query that Uses
SLVs

The Excalibur Text Search DataBlade module uses the following two rules when assigning scores:

- Exact matches are always scored slightly higher than pattern matches.
- All pattern matches are scored equally, even if some matches appear to approximate the clue better than others.

These rules have a significant role in the scoring of pattern searches and exact phrase searches. Phrases that match the clue exactly are scored higher than the phrases that pattern-match the words of the clue.

Limiting the Number of Rows a Fuzzy Search Returns

At times, you might find that the search engine returns more information than you require. The `MAX_MATCHES` and `WORD_SCORE` tuning parameters allow you to limit the number of rows that the search engine returns.

For example, suppose that, when you execute the following query, the text search engine takes a while to return one hundred rows, although you are interested in just the first three:

```
SELECT id, description FROM videos
WHERE etx_contains(description,
  Row('multimedia document editor' ,
    'PATTERN_TRANS & PATTERN_SUBS'));
```

You can specify the maximum number of rows the search engine returns by using the `MAX_MATCHES` tuning parameter, as shown in the following example:

```
SELECT id, description FROM videos
WHERE etx_contains(description,
  Row('multimedia document editor' ,
    'PATTERN_TRANS & PATTERN_SUBS & MAX_MATCHES=3'));
```

It is important to note that if you specify the `MAX_MATCHES` tuning parameter in a query, it is possible that not all rows that satisfy all the search criteria will be returned, since this is the nature of the tuning parameter. This can cause misleading results if a subsequent qualification, different from the `etx_contains()` operator, is applied in the query.

Another way to limit the number of returned rows is with the `WORD_SCORE` tuning parameter. For example, suppose you are interested in having the text search engine return only rows that have a score of 85 or better. You would specify this condition by specifying `WORD_SCORE = 85` in the `etx_contains()` operator, as shown in the following example:

```
SELECT id, description FROM videos
WHERE etx_contains ( description,
  Row('multimedia document editor' ,
    'PATTERN_TRANS & PATTERN_SUBS & WORD_SCORE = 85'));
```

Pattern Search

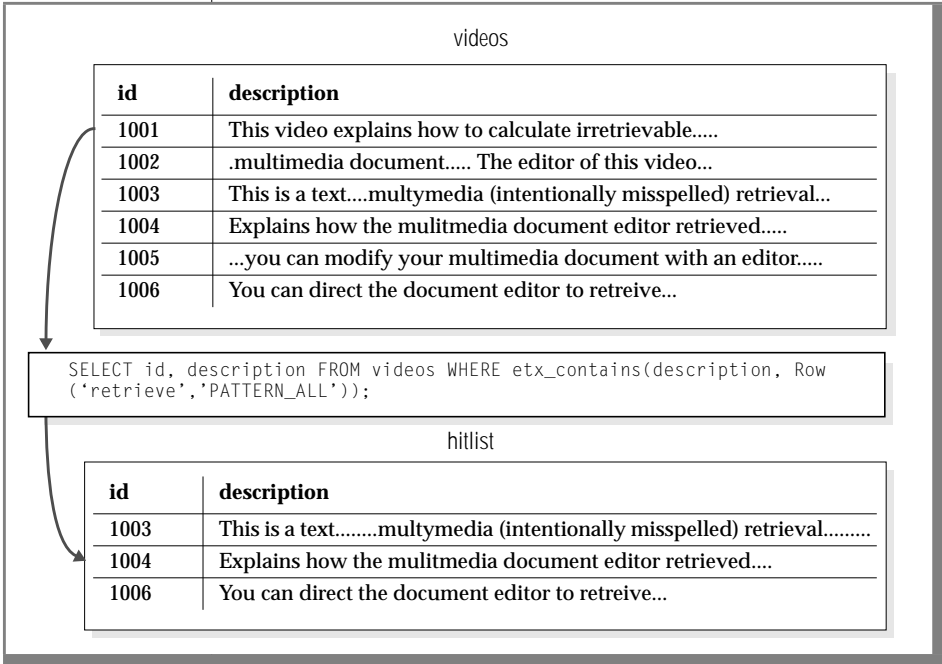
A pattern search is a fuzzy search that takes into account multiple transpositions and substitutions and also returns any substrings or superstrings of the clue. To enable a pattern search, you pass the `PATTERN_ALL` tuning parameter to the search engine.

For example, to perform a pattern search for the word `retrieve`, execute the following SQL statement:

```
SELECT id, description FROM videos
WHERE etx_contains(description,
  Row ('retrieve','PATTERN_ALL'));
```

Given the search text shown in [Figure 2-5](#), rows that the search engine returns are those that contain `retrieved`, `retrieval`, or `retrieve`. Because of the high default word score threshold of 70, the word `irretrievable` is not found. However, if you use the `WORD_SCORE` tuning parameter to set the word score lower, the word `irretrievable` might be returned, but with a lower score.

Figure 2-5
Example of Pattern Search



A pattern search differs from a substitution or a transposition search in that a pattern search allows multiple substitutions and transpositions. A substitution search returns words that differ from the clue by a *single* substitution. For example, a search for the word `travel` with `PATTERNS_SUBS` set might return `gravel`. A transposition search returns words that contain a single transposition. For example, a search for the word `travel` with `PATTERN_TRANS` set might return `travle`.

In contrast, a search for the same clue with `PATTERN_ALL` enabled returns words such as `traveled`, `travelled`, `unraveled`, and `travvel`, in addition to `gravel` and `travle`. The text search engine assigns a higher score to words matched with `PATTERNS_SUBS` or `PATTERN_TRANS` enabled than it does to words that are matched with `PATTERN_ALL` enabled.

If you want to enable basic pattern matching, use the `PATTERN_BASIC` tuning parameter. A search with this parameter will return the best matches that have a score of `WORD_SCORE` or better. The search might return transpositions, substitutions, and superstring and substring pattern matches depending on the value of `WORD_SCORE`, though it is not guaranteed.

The `PATTERN_ALL` tuning parameter is equivalent to specifying the three parameters `PATTERN_BASIC`, `PATTERN_SUBS`, and `PATTERN_TRANS` at once.

Phrase Searching with Pattern Matching

When you perform a search with `SEARCH_TYPE = PHRASE_EXACT`, the search text must contain a phrase that is identical to the clue for a hit to occur. If you specify a pattern search in addition to an exact phrase search (by specifying `PATTERN_ALL`, for example), the individual words in the clue must pattern-match the corresponding words in the search text in the same order in which the words appear in the search text.

For example, in a pattern search combined with an exact phrase search, the text `jill john jones` matches the clue `jyll jonh gones` but does not match the clue `john jill jones`. That is, order always counts in an exact phrase search, regardless of whether you also specify pattern matching.

In an exact phrase search, all words in the clue (or pattern matches thereof) must be found in the search text. Partial matches, where one or more words are missing, do not count as hits.

When you perform a search with `SEARCH_TYPE = PHRASE_APPROX`, the search text must contain either a phrase identical to the clue, one or more words of the clue in the same order, or one or more words of the clue in a different order. The Excalibur Text Search DataBlade module uses the number of words and word order to produce a score for all hits; the more words in closer order a search text has, the higher the score it is assigned.

For example, if the clue is drop many balls, the search engine produces the following ranking based on scores:

```
He can drop many balls(exact match)
He has many balls (2 words, in same order as clue)
He let five balls drop(2 words, different order from clue)
He has many children(1 word)
```

A keyword pattern search works like a search with SEARCH_TYPE = PHRASE_APPROX. However, a pattern search adds pattern matching on a word-by-word basis. For example, if the clue is drop many balls, then the search engine produces the following ranking:

```
He can dorp many valls(assuming dorp pattern matches drop, and so on)
He has mani balds
He let five galls frop
He has many children
```

Tip: Informix recommends that you refrain from setting SEARCH_TYPE to PHRASE_APPROX and passing the PATTERN_BASIC or PATTERN_ALL tuning parameters for the same search. Doing so can result in poor performance for searches of this type.



Tutorial

Step 1: Creating a Table Containing a Text Search Column	3-4
Creating dbspaces and sbspaces for Storage	3-4
Creating the Table	3-5
Step 2: Populating the Table with Text Search Data	3-5
Step 3: Creating Word Lists and a User-Defined Character Set	3-6
Creating a Stopword List	3-6
Creating a Synonym List	3-7
Creating a User-Defined Character Set	3-8
Step 4: Creating an etx Index	3-9
Determining the etx Index Parameters	3-10
Creating an etx Index.	3-10
Creating a Nonfragmented Index	3-10
Creating a Fragmented Index	3-12
Step 5: Performing Text Search Queries	3-12
Step 6: Highlighting	3-14

This chapter provides a step-by-step procedure for setting up and performing text searches using the Excalibur Text Search DataBlade module. The following topics are covered:

- How to create and populate a text search table
- How to create and use word lists
- How to create and use a user-defined character set
- How to create an **etx** index
- How to perform text search queries
- How to highlight text

You should have read [Chapter 1, “DataBlade Module Overview,”](#) and be familiar with the concepts and terminology it explains. The following table lists the steps you must take to set up and use the Excalibur Text Search DataBlade module.

Step	What You Must Do	Refer To
1	Create a table that contains a text search column	page 3-4
2	Populate the table with text search data	page 3-5
3	Create word lists and a user-defined character set	page 3-6
4	Create an etx index	page 3-9
5	Perform text search queries	page 3-12
6	Highlight text	page 3-14

Step 1: Creating a Table Containing a Text Search Column

If the data you want to search is already contained in a BLOB, CLOB, LVARCHAR, CHAR, VARCHAR, IfxDocdesc, or IfxMRData column of a table, you can skip this step and proceed to [“Step 3: Creating Word Lists and a User-Defined Character Set”](#) on page 3-6.

Creating dbspaces and sbspaces for Storage

For optimal performance and to take advantage of data management facilities of a DBMS, such as transaction rollback, Informix recommends that you store your text search table and associated indexes in dbspaces and sbspaces, respectively, that you allocate specifically for this purpose.

If you are using a CLOB, BLOB, or IfxMRData data type, you are required to store your data in an sbspace. Synonym and stopword lists are also stored in sbspaces. And since **etx** indexes are also stored in sbspaces, they are always detached, because the table that contains the indexed column is stored in a dbspace.

The following example creates a dbspace named **dbbsp1** whose initial offset is 0 and which has a size of 10 MB:

```
onspaces -c -d dbbsp1 -p /Dbspace/dbbsp1 -o 0 -s 10000
```

The following example creates two sbspaces named **sbsp1** and **sbsp2**, each with an initial offset of 0 and a size of 100 MB, and logging turned on:

```
onspaces -c -S sbsp1 -g 2 -p /SBspace/sbsp1 -o 0 -s 100000 -Df "LOGGING=ON"  
onspaces -c -S sbsp2 -g 2 -p /SBspace/sbsp2 -o 0 -s 100000 -Df "LOGGING=ON"
```

For more information on how to create dbspaces and sbspaces, and the complete syntax of the **onspaces** utility, consult the administrator's guide for your database server.

Creating the Table

The following example shows how to create a table called **reports** using the CREATE TABLE statement. The text data will be stored in the CLOB column called **abstract**.

```
CREATE TABLE reports
(
    doc_no INTEGER,
    author VARCHAR(60),
    title CHAR(255),
    abstract CLOB
);
```

For the full syntax of the CREATE TABLE statement, see [Informix Guide to SQL: Syntax](#).

Step 2: Populating the Table with Text Search Data

This example populates the CLOB column of the **reports** table with data from the operating system file **/local0/excal/dbms.txt** by using the **FileToCLOB()** smart large object function.

```
INSERT INTO reports (doc_no, author, title, abstract)
VALUES(
    1,
    'C.J. Date',
    'Introduction to Database Systems',
    FileToCLOB ('/local0/excal/dbms.txt', 'client')
);
```

You can also populate a CLOB column with data from another CLOB column in the database via the **LOCOPY()** smart large object function.

For more information on the **FileToCLOB()** and **LOCOPY()** functions, refer to [Informix Guide to SQL: Syntax](#).

Step 3: Creating Word Lists and a User-Defined Character Set

Informix recommends that you always associate a stopwords list with an **etx** index to prevent stopwords from being indexed. This section describes how to create a stopwords list based on the operating system file of stopwords provided by the Excalibur Text Search DataBlade module.

Creating a custom synonym list is an optional activity that you might consider delaying until you become more familiar with the Excalibur Text Search DataBlade module. The DataBlade module includes a default synonym list called **etx_thesaurus** that contains many standard synonyms. The list is created when the DataBlade module is registered in your database.

The Excalibur Text Search DataBlade module provides three built-in character sets (ASCII, ISO, and OVERLAP_ISO) that are used to determine which characters in the text should be indexed. You can define your own character set if the ones provided do not index the desired characters. This section describes how to create a user-defined character set that is similar to the ASCII character set but also indexes hyphens.

For an explanation of word lists and character sets, see [Chapter 1, “DataBlade Module Overview.”](#)

Creating a Stopword List

To create a stopwords list, you use the **etx_CreateStopWlst()** procedure to indicate the name of the stopwords list, the location of the operating system file that currently contains the list of stopwords, and the name of the sbpace that will contain the list.

You may use your own operating system file of stopwords or copy the one provided by the Excalibur Text Search DataBlade module and edit it for your own use. The list of standard stopwords provided by the DataBlade module is called:

```
$INFORMIXDIR/extend/ETX.<version>/wordlist/etx_stopwords.txt
```

In the preceding filename, **<version>** refers to the current version of the DataBlade module installed on your computer.

For example, to create a stopword list named **my_stopwordlist** from the operating system file **/local0/excal/stp_word** and to be contained in the sbspace named **sbsp1**, execute the following statement:

```
EXECUTE PROCEDURE etx_CreateStopWlst
('my_stopwordlist', '/local0/excal/stp_word', 'sbsp1');
```

The format of the operating system file is one stopword per line. The operating system file should not contain any proprietary formatting information but should consist of only standard ASCII characters.

To insert a new stopword into an existing stopword list, you must use the **etx_DropStopWlst()** procedure to drop the list, add the stopword to the operating system file, and then recreate the stopword list using the procedure **etx_CreateStopWlst()**.

For the complete syntax and additional examples illustrating the use of **etx_CreateStopWlst()** and **etx_DropStopWlst()**, see [Chapter 5, “Routines.”](#)

Creating a Synonym List

To create a custom synonym list, use the **etx_CreateSynWlst()** procedure to indicate the name of the synonym list, the location of the operating system file that currently contains the list of synonyms, and the name of the sbspace that will contain the list.

For example, to create a synonym list named **my_synonymlist** that is currently stored in an operating system file **/local0/excal/syn_file** and is to be contained in the sbspace named **sbsp2**, execute the following statement.

```
EXECUTE PROCEDURE etx_CreateSynWlst
('my_synonymlist', '/local0/excal/syn_file', 'sbsp2');
```

The format of the operating system file is one root word followed by one or more synonyms, separated by blank spaces. Each line of text must be followed by one blank line. The operating system file should not contain any proprietary formatting information but should consist of only standard ASCII characters. The following is an example of a synonym list's operating system file:

```
clay earth mud loam

clean pure spotless immaculate unspoiled
```

To indicate to the text search engine that you want it to consult this particular synonym list while performing a search, you must set `MATCH_SYNONYM` to **my_synonymlist**.

If you specify the `MATCH_SYNONYM` tuning parameter without specifying the name of a synonym list, the default synonym list **etx_thesaurus** is automatically consulted.

To insert a new set of synonyms into an existing synonym list, you must use the **etx_DropSynWlst()** procedure to drop the list, add the set of synonyms to the operating system file, and recreate the synonym list using the procedure **etx_CreateSynWlst()**.

For the complete syntax and additional examples illustrating the use of **etx_CreateSynWlst()** and **etx_DropSynWlst()**, see [Chapter 5, “Routines.”](#)

Creating a User-Defined Character Set

To create a user-defined character set, use the **etx_CreateCharSet()** procedure to specify the name of the new character set and the location of the operating system file that currently contains the definition of the character set.

For example, to create a user-defined character set named **my_new_charset** that is currently stored in the operating system file **/local0/excal/my_new_char_set_file**, execute the following statement:

```
EXECUTE PROCEDURE etx_CreateCharSet  
('my_new_charset', '/local0/excal/my_new_char_set_file');
```

The operating system file consists of 16 lines of 16 hexadecimal numbers. Each position corresponds to a specific ASCII character. If you want the character in the position to be indexed, enter the character that it should be indexed as. If you do not want the character to be indexed, enter 00.

In the sample operating system file `/local0/excal/my_new_char_set_file`, shown below, all alpha-numeric characters will be indexed, as well as the hyphen (hexadecimal value `0x2D`). In addition, all lowercase letters are mapped to uppercase.

```
# Character set that indexes hyphens and
# alpha-numeric characters. All lower case letters
# are mapped to upper case.
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 2D 00 00
30 31 32 33 34 35 36 37 38 39 00 00 00 00 00 00
00 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
50 51 52 53 54 55 56 57 58 59 5A 00 00 00 00 00
00 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
50 51 52 53 54 55 56 57 58 59 5A 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

To indicate to the text search engine that you want it to consult this user-defined character set while creating an **etx** index, you must set the `CHAR_SET` index parameter to **my_new_charset**. If you do not specify the `CHAR_SET` index parameter in the `CREATE INDEX` statement, the built-in ASCII character set is used by default.

For the complete syntax and additional examples illustrating the use of `etx_CreateCharSet()`, see [Chapter 5, “Routines.”](#)

Step 4: Creating an etx Index

To create an **etx** index, you specify the **etx** access method in the `USING` clause of the `CREATE INDEX` statement. You specify the operator class right after specifying the indexed column. Refer to [Chapter 1, “DataBlade Module Overview,”](#) for a list of the supported data types and their corresponding operator classes.

You must create an **etx** index for each text column you plan to search. You cannot alter the characteristics of an **etx** index once you create it. Instead, you must drop the index and re-create it with the desired characteristics.

Determining the etx Index Parameters

Index parameters are used to customize **etx** indexes based on the type of searches you plan to perform. Although they are not required, index parameters can increase performance in cases where you know the types of searches that will be executed, because the **etx** index can be built to suit a particular type of search.

For example, the `WORD_SUPPORT` index parameter specifies whether you plan on using exact or pattern-matching searches. The `PHRASE_SUPPORT` parameter indicates what level of phrase searching you want, from full to no support. Use the `CHAR_SET` parameter to specify what character set your documents use. The `STOPWORD_LIST` parameter lets you specify a custom stopword list, while the `INCLUDE_STOPWORDS` parameter indicates that you want the stopwords specified by the `STOPWORD_LIST` parameter to be included in the index.

Refer to [Chapter 6, “etx Index Parameters,”](#) for detailed information on these parameters.

Creating an etx Index

This section describes how to create fragmented and nonfragmented indexes. You should have already created an sbspace in which to store your index. If you have not, consult the the administrator’s guide for your database server for instruction on how to do this.

Creating a Nonfragmented Index

Suppose your search text is contained in a column of type CLOB, named **abstract**, and that the column is located in a table named **reports**. To create an **etx** index named **reports_idx1** for this table, use the following syntax:

```
CREATE INDEX reports_idx1 ON reports (abstract etx_clob_ops)
  USING etx
  IN sbsp1;
```

The preceding example creates an **etx** index that by default supports exact word searches but does not support phrase searches. The index is stored in the sbspace **sbsp1** and indexes, by default, only ASCII characters. Since no stopword list is specified, all words in the document will be indexed. The operator class **etx_clob_ops** is specified since the abstract column is of type CLOB.

The following example creates an **etx** index on the **title** column that is of type CHAR. In this case, the operator class is **etx_char_ops** instead of the previously used **etx_clob_ops** for columns of type CLOB. The index does not include the stopwords found in the list **my_stopwordlist**. By default, the index supports exact matches, but not phrase searches. The index uses the built-in ISO character set, specified by the CHAR_SET parameter. The index will be stored in the sbspace **sbsp1**:

```
CREATE INDEX reports_idx2 ON reports (title etx_char_ops)
  USING etx (STOPWORD_LIST = 'my_stopwordlist',
  CHAR_SET = 'ISO') IN sbsp1;
```

The following statement creates an **etx** index that supports a medium level of exact and approximate phrase searches, supports exact word searches, and indexes ASCII characters. The index does not include the stopwords found in the list **my_stopwordlist**. The index is stored in the sbspace **sbsp1**:

```
CREATE INDEX reports_idx3 ON reports (abstract etx_clob_ops)
  USING etx (WORD_SUPPORT = 'EXACT',
  STOPWORD_LIST='my_stopwordlist', PHRASE_SUPPORT = 'MEDIUM')
  IN sbsp1;
```

The following statement creates an **etx** index that supports the most accurate level of exact and approximate phrase searching, supports pattern word searches, and indexes, by default, ASCII characters. The index does not include the stopwords found in the list **my_stopwordlist**. Since the user-defined character set **my_new_charset** is specified, hyphens are indexed, as well as all alpha-numeric characters. The index is stored in the sbspace **sbsp1**:

```
CREATE INDEX reports_idx4 ON reports (abstract etx_clob_ops)
  USING etx (WORD_SUPPORT = 'PATTERN',
  STOPWORD_LIST='my_stopwordlist', PHRASE_SUPPORT = 'MAXIMUM',
  CHAR_SET = 'my_new_charset' )
  IN sbsp1;
```

Creating a Fragmented Index

You can use the **FRAGMENT BY** clause of **CREATE INDEX** to create both round-robin and expression-based fragmentation. You cannot, however, use the **etx_contains()** operator as part of your fragmentation expression.

Suppose that you want to create a fragmented index on the CLOB column **abstract** of the table **reports** with the following requirements: documents with a **doc_no** value less than 1000 are stored in the sbspace **sbsp1**, and documents with a **doc_no** value greater than or equal to 1000 are stored in the sbspace **sbsp2**.

The following **CREATE INDEX** statement creates a fragmented **etx** index that meets the preceding requirements:

```
CREATE INDEX reports_idx5 ON reports (abstract etx_clob_ops)
  USING etx (WORD_SUPPORT = 'PATTERN',
    STOPWORD_LIST = 'my_stopwordlist', INCLUDE_STOPWORDS = 'TRUE',
    PHRASE_SUPPORT = 'MAXIMUM')
  FRAGMENT BY EXPRESSION
    doc_no < 1000 IN sbsp1,
    doc_no >= 1000 IN sbsp2 ;
```

This index supports pattern and word matching and supports maximum phrase support. Although the word list **my_stopwordlist** is specified, all of the stopwords are actually indexed, due to the **INCLUDE_STOPWORDS** index parameter. However, the stopwords are relevant in a search only if the **CONSIDER_STOPWORDS** tuning parameter is specified in the **etx_contains()** operator.

Step 5: Performing Text Search Queries

To perform a text search, you use the **etx_contains()** operator in the **WHERE** clause of a **SELECT** statement. For example, suppose that you store your search text in an CLOB column named **abstracts**. To execute a pattern search on this column for the phrase multimedia document editor, execute the following statement:

```
SELECT title FROM reports
  WHERE etx_contains(abstract,
    Row('multimedia document editor',
      'SEARCH_TYPE = PHRASE_EXACT & PATTERN_ALL'));
```


The search returns the **title** column of the documents that contain the phrases multimedia document editor, multimedia document editor, and even multimillion documentary editorials, although the last hit will have a much lower score than the first two hits. Since the statement specified an exact phrase search, the search will not find documents that contain just the word multimedia, or the phrase editorial of a multimedia event, because of the differing order of the words multimedia and editorial.

Use a keyword proximity search to find documents that contain either of the phrases multimedia editor or editor of a multimedia event. The preceding example shows that phrase searching might not be your best choice, since the order of the words always counts in phrase searches. Order does not count in keyword searches because the words are treated as separate entities. Proximity searching ensures that the keywords are close to each other. The following is an example of a keyword proximity search:

```
SELECT title FROM reports
WHERE etx_contains(abstract,
  Row('multimedia editor',
    'SEARCH_TYPE = PROX_SEARCH(5) '));
```

The search returns the **title** column of documents that contain both the keywords multimedia and editor as long as they are no more than five words apart, inclusive. This means that the search will not return a document that contains the phrase editor of a world class magazine known for its cutting edge articles on multimedia because the keywords multimedia and editor are separated by more than five words.

Sometimes it is necessary to search for stopwords because they are relevant parts of the clue. For example, you may want to search for the exact phrase plug and play where the word and is a stopwords. The text search engine by default does not search for stopwords, so the result of the search might not be exactly what you want. The following example shows how you can force the inclusion of stopwords in a search:

```
SELECT title FROM reports
WHERE etx_contains(abstract,
  Row('plug and play',
    'SEARCH_TYPE = PHRASE_EXACT & CONSIDER_STOPWORDS'));)
```



Important: You must specify the index parameter `INCLUDE_STOPWORDS` when creating the `etx` index if you want to use the tuning parameter `CONSIDER_STOPWORDS`. The index parameter `INCLUDE_STOPWORDS` forces the stopwords specified by the `STOPWORD_LIST` index parameter to be indexed; by default, the stopwords specified by this parameter are not indexed.

For the complete syntax of the `etx_contains()` operator and additional examples that use it, see [Chapter 5, “Routines.”](#)

Step 6: Highlighting

To obtain highlighting information, specify the `etx_GetHilite()` function in the `SELECT` list of the query, as shown in the following example:

```
SELECT etx_GetHilite(abstract, rc) FROM reports
WHERE etx_contains(abstract,
Row('multimedia editor', 'SEARCH_TYPE = PROX_SEARCH(5)'),
rc # etx_ReturnType);
```

The example shows a proximity search, similar to the one in Step 5. In addition to returning the document, the example also shows how to return highlighting information via the function `etx_GetHilite()`. The data type of the return value of the function is `etx_HiliteType`, an Informix-defined row data type that consists of two fields, `vec_offset` and `viewer_doc`, that contain highlighting information.

The `vec_offset` field contains offset information about every instance of the words `multimedia` and `editor` in the returned document, as long as the two words are within five words of each other. The `viewer_doc` field contains the text document itself.

Since the data types of the `vec_offset` and `viewer_doc` fields are CLOB and BLOB, respectively, your application must use standard ESQL/C large object functions such as `ifx_lo_open()` and `ifx_lo_read()` or DataBlade API large object functions `mi_lo_open()` and `mi_lo_read()` to manipulate the contents.

For more information on the ESQL/C functions that are available to manipulate CLOB and BLOB data types, refer to the [INFORMIX-ESQL/C Programmer's Manual](#). For more information on the DataBlade API functions, refer to the [DataBlade API Programmer's Manual](#).

For more information on the **vec_offset** and **viewer_doc** fields of the **etx_HiliteType** row data type, see [Chapter 4, “Data Types.”](#)

For more information on the **etx_GetHilite()** function, see [Chapter 5, “Routines.”](#)

Data Types

etx_ReturnType.	4-4
etx_HiliteType	4-6
IfxDocDesc	4-9
IfxMRData	4-14

This chapter describes the four new data types that come with the Excalibur Text Search DataBlade module.

Data Type	Description
etx_ReturnType	A named row type used to return side-effect data, such as score and internal highlighting information, to the etx_contains() operator. etx_ReturnType is the data type of the statement local variable (SLV) of the etx_contains() operator.
etx_HiliteType	A named row type used to access highlighting information. etx_HiliteType is the data type of the information returned by the highlighting routine etx_GetHilite() .
IfxDocDesc	A named row type that allows you to store your documents in the database or on the operating system file system. IfxDocDesc is a data type defined in the Text Descriptor DataBlade module, one of the DataBlade modules required by the Excalibur Text Search DataBlade module. Although it is used by all text search DataBlade modules, it is described in this guide.
IfxMRData	A multirepresentational opaque data type that dynamically decides, for the purposes of improving I/O performance, whether to store your text documents as LVARCHARs or CLOBs. IfxMRData is a data type defined in the Text Descriptor DataBlade module, one of the DataBlade modules required by the Excalibur Text Search DataBlade module. Although it is used by all text search DataBlade modules, it is described in this guide.

etx_ReturnType

The data type `etx_ReturnType` is a named row type defined by the Excalibur Text Search DataBlade module as follows:

```
CREATE ROW TYPE etx_ReturnType
(
    score          REAL,
    hilite_info    etx_InternalHilite
);
```

`etx_ReturnType` is the data type of the optional statement local variable (SLV) of the **`etx_contains()`** operator. Use the SLV in a `SELECT` statement to obtain scoring information about the returned document or to pass internal highlighting information to the **`etx_GetHilite()`** routine. The following example shows a typical use of SLVs:

```
SELECT rc1.score, title FROM reports
WHERE etx_contains (abstract,
    Row('video'), rc1 # etx_ReturnType)
AND doc_no > 1005
ORDER BY 1;
```

In the example, **`rc1`** is the SLV, and its **`score`** field is used by the `SELECT` statement to order the returned rows by their score.

For more detailed information on and examples of SLVs, see [Chapter 1, “DataBlade Module Overview,”](#) and [Chapter 5, “Routines.”](#)

The `etx_ReturnType` row type consists of two fields: **`score`** and **`hilite_info`**. The following sections describe the fields and how to use the information contained in them.

The score Field

The **`score`** field contains a numeric value that indicates the relevance of a returned document to the search criteria, compared to that of other indexed records. The higher the score value, the more closely the document matches the criteria.

The score is a value between 0 and 100, 0 indicating no match and 100 indicating the best possible match.

You can use **score** to order the returned rows according to how closely they match the search criteria. You can also use the **score** value in **SELECT** statements to limit a search by specifying the minimum score of returned documents, as shown in the following example:

```
SELECT rc.score, id, description FROM videos
WHERE etx_contains ( description,
    Row('multimedia document editor' ,
        'PATTERN_TRANS & PATTERN_SUBS'),
    rc # etx_ReturnType) AND
rc.score > 85 ;
```

The preceding query specifies that only those documents that have a score of 85 or better should be returned.

The **hilite_info** Field

The **hilite_info** field is used internally by the Excalibur Text Search DataBlade module. The information contained in this field is only relevant to the **etx_GetHilite()** function. You should not attempt to access the information contained in this field.

For more information on the **etx_GetHilite()** function, refer to [Chapter 5, “Routines.”](#)

etx_HiliteType

The data type `etx_HiliteType` is a named row type defined by the Excalibur Text Search DataBlade module as follows:

```
CREATE ROW TYPE etx_HiliteType
(
    vec_offset CLOB,
    viewer_doc BLOB
);
```

`etx_HiliteType` is the data type of the information returned by the **`etx_GetHilite()`** function. This function is used in the SELECT list of a query to pinpoint the location of a clue in a document. This is called *highlighting*. The following example shows how to use the **`etx_GetHilite()`** function in a SELECT statement:

```
SELECT etx_GetHilite (description, rc) FROM videos
WHERE etx_contains(description,
    'multimedia', rc # etx_ReturnType);
```

In the example, `rc` is the statement local variable (SLV) passed to the function **`etx_GetHilite()`**. The function **`etx_GetHilite()`** is used in the SELECT list of the query. The function returns highlighting information stored in the `etx_HiliteType` row data type for each row returned by the query. For more information on the **`etx_GetHilite()`** function, refer to [Chapter 5, “Routines.”](#)

Highlighting information is broken up into two parts: offset information about the location of the clue in a document, and the document itself. The offset information is contained in the **`vec_offset`** field of the `etx_HiliteType` data type and the document is contained in the **`viewer_doc`** field.

The following sections describe in more detail the **`vec_offset`** and **`viewer_doc`** fields of `etx_HiliteType`, as well as how to use the information contained in them.

The `vec_offset` Field

The **`vec_offset`** field, of data type CLOB, contains a list of ordered pairs of integers that pinpoint the location of every instance of the clue in the search text.

Each integer is separated from other integers by a blank space. The first integer of the ordered pair is the beginning offset of the clue, and the second integer is the length of the highlighted string. The first character in a document has an offset of 0.

For example, the following is sample output from the **vec_offset** field if you search for the word `be` in the search text `to be or not to be`:

```
3 2 16 2
```

The first instance of the word `be` starts at offset 3, assuming the first character is at offset 0, and the length of the string to be highlighted is 2. The second instance of the word `be` starts at offset 16 and the length of the string to be highlighted is again 2.

The Excalibur text search engine sometimes expands clues to search for similar words. For example, if you search for the word `house` and request highlighting information by using the **etx_GetHilite()** function, the word `housing` may also be highlighted. This explains why the length of the highlighted string specified via the **vec_offset** field may sometimes differ from the length of the original clue.

Refer to the following section for more information on the contents of the **viewer_doc** field.

Since the **vec_offset** field is of type CLOB, your application must use standard ESQL/C large object functions such as **ifx_lo_open()** and **ifx_lo_read()** or DataBlade API large object functions such as **mi_lo_open()** and **mi_lo_read()** to manipulate its contents.

For more information on the ESQL/C functions that are available to manipulate CLOB data types, refer to the [INFORMIX-ESQL/C Programmer's Manual](#). For more information on the DataBlade API functions, refer to the [DataBlade API Programmer's Manual](#).

The viewer_doc Field

The **viewer_doc** field contains the document that has just been searched. The offsets contained in the **vec_offset** field are relative to this document.

Since the **viewer_doc** field is of type BLOB, your application must use standard ESQL/C large object functions such as **ifx_lo_open()** and **ifx_lo_read()** or DataBlade API large object functions such as **mi_lo_open()** and **mi_lo_read()** to manipulate its contents.

For more information on the ESQL/C functions that are available to manipulate BLOB data types, refer to the [INFORMIX-ESQL/C Programmer's Manual](#). For more information on the DataBlade API functions, refer to the [DataBlade API Programmer's Manual](#).

IfxDocDesc

IfxDocDesc is a named row type, defined by Informix in the Text Descriptor DataBlade module for use with the Excalibur Text Search DataBlade module. It is defined as follows:

```
CREATE ROW TYPE IfxDocDesc
(
    format VARCHAR(18),
    version VARCHAR(10),
    location LLD_Locator,
    params LVARCHAR
);
```

The Text Descriptor DataBlade module is one of the DataBlade modules required by the Excalibur Text Search DataBlade module.

The main advantage of the IfxDocDesc data type is that you can choose to store your text documents either on the operating system file system or in the database. The two types of storage are available for different text documents within the same column of the same table.

The following sections describe the IfxDocDesc data type in more detail and how to use it when inserting data into a column.

The Fields of the IfxDocDesc Row Data Type

The IfxDocDesc row data type has four fields. This section describes the fields and recommends settings when appropriate.

The format Field

This field is used to store format information for the text document. The field is not used by the DataBlade module, so you can enter any text you want, including `NULL`. Informix recommends you use this field for your own record keeping.

An example of a value for this field is `MS Word` for a Microsoft Word document.

The version Field

This field is used to store version information for the text document, based on the information in its **format** field. The field is not used by the DataBlade module, so you can enter any text you want, including `NULL`. Informix recommends you use this field for your own record-keeping.

An example of a value for this field is `7.0`, which indicates the version of Microsoft Word used to create the document.

The location Field

`LLD_Locator`, the data type of the **location** field, is a named row data type, defined by Informix as follows:

```
CREATE ROW TYPE LLD_Locator
(
    lo_protocol      CHAR(18),
    lo_pointer       LLD_Lob,
    lo_location      LVARCHAR
);
```

`LLD_Locator` is defined in the LOB Locator DataBlade module, one of the DataBlade modules required by the Excalibur Text Search DataBlade module.

Use the three fields of `LLD_Locator` to specify the location of your text document, either in the database itself or a file on the operating system file system.

The following table summarizes the information about the fields of the `LLD_Locator` data type.

Field	Description
lo_protocol	Identifies the type of the large object.
lo_pointer	Pointer to a smart large object, or <code>NULL</code> if anything other than a smart large object.
lo_location	Pointer to the large object, if not a smart large object. Set to <code>NULL</code> if it is a smart large object.



Important: *LLD_Lob*, the data type of the **lo_pointer** field of the *LLD_Locator* data type, is an Informix-defined complex data type similar to the *CLOB* and *BLOB* types, but in addition to pointing to the location of a smart large object, it specifies whether the object contains binary or character data.

The value of the **lo_protocol** field determines the values of the other *LLD_Locator* fields. The following table lists the currently available protocols and summarizes the values for the other fields for each protocol.

lo_protocol	lo_pointer	lo_location	Description
IFX_BLOB	SMART LARGE OBJECT POINTER (LLD_LOB)	NULL	Smart large object that might contain binary data
IFX_CLOB	SMART LARGE OBJECT POINTER (LLD_LOB)	NULL	Smart large object that only contains character data
IFX_FILE	NULL	Full file pathname	Operating system file on the database server machine

For a complete description of the *LLD_Locator* and *LLD_Lob* data types, see the [LOB Locator DataBlade Module Programmer's Guide](#).

The params Field

Reserved for use by the **etx** access method. When you load data into a table, specify **NULL**.



Important: *If you insert NULL into any of the fields of an IfxDocDesc column, you must explicitly cast it to its data type in the INSERT statement. The same is true for UPDATE statements that update the value to NULL. This is true for all fields of all row types in Informix Dynamic Server with Universal Data Option.*

Populating a Table That Contains an IfxDocDesc Data Type

The IfxDocDesc data type allows you to store your documents in a table as a smart large objects or to reference the documents on the operating system by filename. Suppose that your search text is stored in the following collection of Microsoft Word files on a UNIX file system:

```
/local0/excal/desc1.doc
/local0/excal/desc2.doc
```

Further suppose that you want to load these files into a table called **videos1**, described by the following example:

```
CREATE TABLE videos1
(
    id            INTEGER,
    name          VARCHAR(30),
    description    IfxDocDesc
);
```

To load the first of these files, **/local0/excal/desc1.doc**, into a row of the **videos1** table, execute the following statement:

```
INSERT INTO videos1 (id, name, description)
VALUES ( 1010, 'The Unforgiven',
Row ('MS Word', '7.0',
    Row ('IFX_FILE', NULL::LLD_Lob,
        '/local0/excal/desc1.doc')::LLD_Locator,
    NULL::LVARCHAR)::IfxDocDesc
);
```

Because you specified IFX_FILE as the protocol, the **description** column does not actually contain the search text, but instead has a pointer (of data type LLD_Locator) to the operating system file specified by the INSERT statement.

Note the heavy use of the **Row()** constructor when specifying values for the named row types LLD_Lob and LLD_Locator.

If you want to store the second text file, **/local0/excal/desc2.doc**, in the database itself, use the **IFX_BLOB** or **IFX_CLOB** protocol, as shown in the following similar example:

```
INSERT INTO videos1 (id, name, description)
VALUES ( 1011, 'The Sting',
Row ( 'MS Word', '7.0',
      Row ( 'IFX_CLOB',
            FileToCLOB ( '/local0/excal/desc2.doc', 'client'),
            NULL::LVARCHAR )::LLD_Locator,
            NULL::LVARCHAR )::IfxDocDesc
      );
```

The **FileToCLOB()** routine reads the file from the operating system into the database.

For more information on the **FileToCLOB()** function and the **Row()** constructor, refer to the [Informix Guide to SQL: Syntax](#).

IfxMRData

IfxMRData is a multirepresentational opaque type, defined by Informix in the Text Descriptor DataBlade module for use with the Excalibur Text Search DataBlade module. It provides for fast and efficient storage of ASCII text data.

The main advantage of using the multirepresentational IfxMRData data type is that it dynamically determines whether to store your documents as LVARCHARS or CLOBs, depending on the size of the documents. Documents smaller than 2 KB are stored as LVARCHARS, and documents greater than 2 KB are stored as CLOBs.



Important: The precise cutoff point between storing documents as either LVARCHARS or CLOBs is 2040 bytes, and not 2048 bytes. This difference is due to the 8 bytes used to store the data structure itself.

Inserting data into LVARCHARS is faster than inserting data into CLOBs, so to speed up performance it is preferable to use LVARCHAR whenever possible. LVARCHAR columns, however, have a size limitation of 2 KB. Since it is often difficult to know ahead of time what the maximum size of a document is, it is risky to use a column type of LVARCHAR. Specifying a column type of IfxMRData solves this problem because the data type itself determines whether your document should be stored as an LVARCHAR or a CLOB.

You can use the IfxMRData data type with ASCII text data only, because it uses CLOBs, and not BLOBs, as a possible storage type. If your documents contain binary data, you should not store them in columns of type IfxMRData, but in columns of type BLOB instead.

Updating a column can change where the data is stored. For example, suppose a document is initially stored as a CLOB because its size is over 2 KB. If you update the value so that the size of the document is now smaller than 2 KB, the DataBlade module changes its storage to LVARCHAR. This change is transparent to users.

Populating a Table That Contains an IfxMRData Column

There are two ways to enter data into a column of type IfxMRData: either by specifying a string in the INSERT statement or by reading the data from an operating system file. This section describes both methods.

Suppose that some of your search text is stored in the following collection of files on the UNIX file system:

```
/local0/excal/desc1.txt  
/local0/excal/desc2.txt
```

Further suppose that you want to enter data into a table called **videos2**, described by the following example:

```
CREATE TABLE videos2  
(  
    id            INTEGER,  
    name          VARCHAR(30),  
    description    IfxMRData  
);
```

Specifying a String

Some of the search text you want to insert into the table is not stored in operating system files since it is very small. In this case you can enter it directly into the IfxMRData column, just as you would enter data into a standard SQL character column, as shown in the following example:

```
INSERT INTO videos2 (id, name, description)  
VALUES(  
    1010,  
    'The Unforgiven',  
    'Academy-award winning western directed by Clint Eastwood.' );
```

Since the data entered into the **description** column is smaller than 2 KB, the data is stored as an LVARCHAR.

Using a Row Constructor

The second way to enter data into an IfxMRData column is to use an unnamed **Row()** constructor to reference the full pathname of an operating system file that contains the text data and to indicate whether the file is found on the client or the server machine. For example:

```
INSERT INTO videos2 (id, name, description)
VALUES(
    1011,
    'The Sting',
    Row ('/local0/excal/desc1.txt', 'client')
);
```

The preceding example shows how to insert the contents of the operating system file **/local0/excal/desc1.txt** into an IfxMRData column. The Excalibur Text Search DataBlade module will look for the file on the client machine.

If the size of the file **/local0/excal/desc1.txt** exceeds 2 KB, the data will be stored as a CLOB; otherwise the data will be stored as an LVARCHAR.

Note that the contents of the operating system file are actually copied into either an LVARCHAR or a CLOB; the source file **/local0/excal/desc1.txt** is not referenced again by the database server.

The following example is similar to the preceding one, but instead of looking for the operating system file **/local0/excal/desc2.txt** on the client machine, the Excalibur Text Search DataBlade module looks for the file on the server machine.

```
INSERT INTO videos2 (id, name, description)
VALUES(
    1012,
    'The Sting',
    Row ('/local0/excal/desc2.txt', 'server')
);
```

For more information on the **Row()** constructor, refer to the [Informix Guide to SQL: Syntax](#).

Routines

etx_contains()	5-5
etx_CloseIndex()	5-15
etx_CreateCharSet()	5-17
etx_CreateStopWlst()	5-21
etx_CreateSynWlst()	5-23
etx_DropCharSet()	5-26
etx_DropStopWlst()	5-28
etx_DropSynWlst()	5-30
etx_GetHilite()	5-31
etx_Release()	5-40
txt_Release()	5-41

This chapter provides you with reference information on the **etx_contains()** operator and other routines defined for the DataBlade module. You use the routines to do the following tasks:

- Execute a search
- Create and drop stopword lists
- Create and drop synonym lists
- Create and drop user-defined character sets
- Determine the location of a clue in a document
- Close an **etx** index

The following routines and operators are included in the Excalibur Text Search DataBlade module.

Routine or Operator	Description
etx_contains()	Executes a search
etx_CloseIndex()	Closes an etx index
etx_CreateCharSet()	Creates a user-defined character set
etx_CreateStopWlst()	Creates a list of words that will be eliminated from the search
etx_CreateSynWlst()	Creates a list of synonyms that can be used in place of a root word in a search
etx_DropCharSet()	Drops a user-defined character set
etx_DropStopWlst()	Drops a stopword list

(1 of 2)

Routine or Operator	Description
etx_DropSynWlst()	Drops a synonym list
etx_GetHilite()	Determines the location of a clue in a document
etx_Release()	Returns version information for the Excalibur Text Search DataBlade module

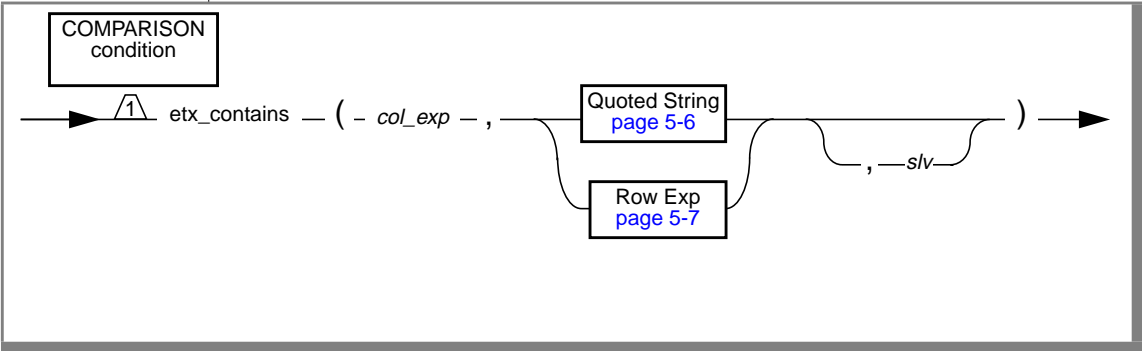
(2 of 2)

The function **txt_Release()**, included in the Text Descriptor DataBlade module, is also described in this chapter. The function returns version information for the Text Descriptor DataBlade module.

etx_contains()

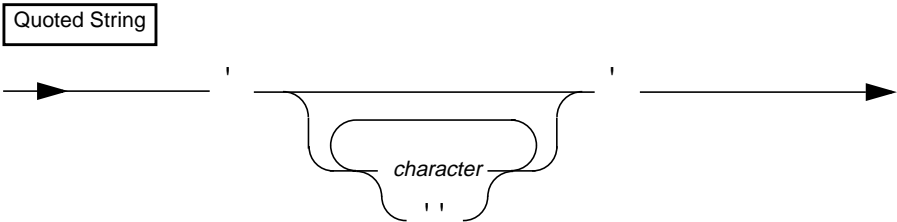
The **etx_contains()** operator executes a search that you define using a clue, tuning parameters, and an optional statement local variable (SLV).

Syntax



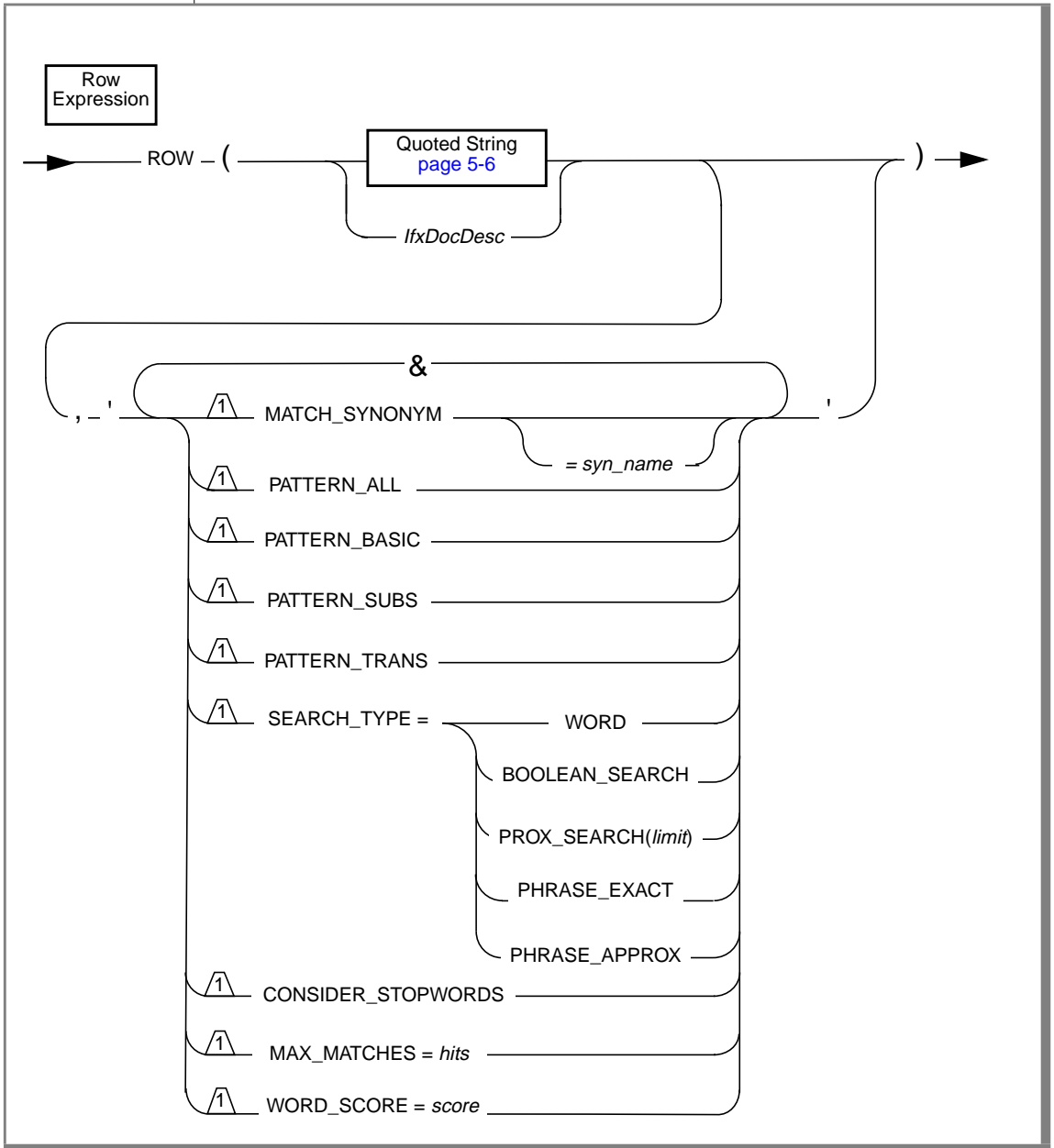
Element	Purpose	Syntax
col_exp	The name of the column you want to search.	Syntax must conform to the column expression syntax in Informix Guide to SQL: Syntax .
slv	A statement local variable (SLV) that the search engine uses to store the score and highlighting information of a particular row.	Syntax must conform to the Identifier segment; see Informix Guide to SQL: Syntax .

Syntax for Quoted String



Element	Purpose	Restrictions	Syntax
<i>character</i>	A character that forms part of the quoted string	<p>When the quoted string is part of a Row() expression (see page 5-7), the following are true:</p> <p>Spaces become delimiters for keywords when performing a keyword, proximity, or Boolean search.</p> <p>The characters & and correspond to the Boolean operators AND and OR, respectively, when performing a Boolean search. The two characters ! and ^ both correspond to the Boolean operator NOT.</p>	Characters are literal values that you enter from the keyboard.

Syntax Row Expression Usage



Element	Purpose	Restrictions	Syntax
<i>hits</i>	Maximum number of hits, per index fragment, returned by a text search.	Data type must be a positive integer.	Syntax must conform to the Literal Number syntax in Informix Guide to SQL: Syntax .
<i>limit</i>	Number of nonsearch words that can occur between two or more search words, inclusive.	Data type must be a nonzero integer.	Syntax must conform to the Literal Number syntax in Informix Guide to SQL: Syntax .
<i>score</i>	Minimum score of a returned hit.	Data type must be a real number.	Syntax must conform to the Literal Number syntax in Informix Guide to SQL: Syntax .
<i>syn_name</i>	Synonym list name.	The synonym list must exist.	Syntax must conform to the Literal Number syntax in Informix Guide to SQL: Syntax .
<i>IfxDocDesc</i>	The clue can be stored in an IfxDocDesc data type.	Only the data in the location field is consulted when a search is executed.	See Chapter 4, “Data Types,” for more information on the IfxDocDesc row data type.

Tuning Parameters

The following table lists the tuning parameters that you can use to guide the search engine when it performs a search.

Tuning Parameter	Description	Restrictions	Default
CONSIDER_STOPWORDS	Indicates to the search engine that stopwords should be included in the search.	The etx index must have been created with the INCLUDE_STOPWORDS AND STOPWORD_LIST index parameters.	Disabled
MATCH_SYNONYM	Enables synonym matching. If no value is specified, the default synonym list etx_thesaurus is consulted. If a value is specified, the specified synonym list is consulted instead of the default list.	<p>If a custom synonym list is specified, it must have already been created via the etx_CreateSynWlst() routine.</p> <p>When used together with pattern matching, such as PATTERN_ALL, only pattern matches of root words are found, not of synonyms.</p>	Disabled
MAX_MATCHES	<p>Allows you to specify the maximum number of hits <i>per index fragment</i> returned by the search engine. Informix recommends that if you want to use this tuning parameter to limit the number of rows returned from a search, you should set it to the larger of 1000 or 10% of the total number of rows in the table.</p> <p><i>Per index fragment</i> means that if, for example, the etx index is fragmented into two parts and this tuning parameter is set to 1000, a possible maximum of 2000 hits might be returned.</p>	None.	All hits.

(1 of 3)

Tuning Parameter	Description	Restrictions	Default
PATTERN_ALL	Enables all the pattern search options: PATTERN_BASIC, PATTERN_TRANS, and PATTERN_SUBS.	The etx index must have been created with the WORD_SUPPORT = PATTERN index parameter.	Disabled
PATTERN_BASIC	Enables the basic search option. The search returns the best pattern matches based on the value of WORD_SCORE. This may include words that are substring or superstring pattern matches of the words in the clue, as well as transpositions and substitutions, though it is not guaranteed.	The etx index must have been created with the WORD_SUPPORT = PATTERN index parameter.	Disabled
PATTERN_SUBS	Indicates to the search engine that you want words returned that match the clue except for one character.	The etx index must have been created with the WORD_SUPPORT = PATTERN index parameter.	Disabled
PATTERN_TRANS	Indicates to the search engine that you want words returned that match the clue except for a single transposition.	The etx index must have been created with the WORD_SUPPORT = PATTERN index parameter.	Disabled
			(2 of 3)

Tuning Parameter	Description	Restrictions	Default
SEARCH_TYPE	Allows you to specify the type of search you want to perform. To execute a keyword search, specify WORD. To execute a Boolean search, specify BOOLEAN_SEARCH. To execute a proximity search, specify PROX_SEARCH. To execute an exact phrase search, specify PHRASE_EXACT. To execute an approximate phrase search, specify PHRASE_APPROX.	One of the following values: WORD, BOOLEAN_SEARCH, PROX_SEARCH(lim), PHRASE_EXACT, or PHRASE_APPROX. If set to PROX_SEARCH, PHRASE_EXACT, or PHRASE_APPROX, the etx index must have been created with the PHRASE_SUPPORT index parameter set to either MEDIUM or MAXIMUM. If set to BOOLEAN, the clue must include at least one Boolean operator.	WORD
WORD_SCORE	Allows you to specify the lowest score of a hit. If a hit has a lower score than the specified WORD_SCORE, the hit is not returned to the user. This tuning parameter only affects pattern searches.	Must be a value from 1 to 100, inclusive. Specifying 0 indicates you want to set the value back to the default, 70.	70

(3 of 3)

Return Type

The **etx_contains()** operator returns BOOLEAN.

Usage

Use **etx_contains()** to execute a search on a document stored in a column of a table. You can use the **etx_contains()** operator only in the WHERE clause of an SQL statement. For example:

```
SELECT title FROM reports
  WHERE etx_contains (abstract, Row('multimedia'))
  AND doc_no > 1005 ;
```

The **etx_contains()** operator has two required parameters: the name of the column containing text data that you want to search, and either a quoted clue or a **Row()** expression that contains the clue and optional tuning parameters. The clue can either be a quoted string or a document stored in an IfxDocDesc data type.



Warning: The column that you want to search must have an **etx** index defined on it if you want to use **etx_contains()** in the WHERE clause.

The optional third parameter of **etx_contains()** is an SLV that returns scoring and internal highlighting information. The contents of the SLV are valid only for the life of the query. The data type of the SLV is **etx_ReturnType**, an Informix-defined row data type. For more information on the **etx_ReturnType** data type, refer to [Chapter 4, “Data Types.”](#)

Although the **etx** access method supports fragmented indexes, you cannot use the **etx_contains()** operator to fragment an index by expression.

If you do not specify any tuning parameters, the **Row()** constructor in the **etx_contains()** operator is optional. This means that the preceding example can also be specified as:

```
SELECT title FROM reports
  WHERE etx_contains (abstract, 'multimedia')
  AND doc_no > 1005 ;
```

Typically, the clue is a quoted string of one or more words, such as the word **multimedia** in the preceding example. Sometimes, however, you might want to use an entire document as the clue. To do this, instead of specifying a quoted string as one of the parameters of the **etx_contains()** operator, specify an IfxDocDesc document.

Due to the flexibility of the `LLD_Locator` data type, the data type of the **location** field of the `IfxDocDesc` data type, you can specify as a clue either a document stored in the database or a document stored as a file on the operating system. Only the **location** field of the `IfxDocDesc` data type is consulted when a document is specified as a clue to the `etx_contains()` operator. The contents of the other fields, such as **format** and **version**, are ignored.

An example of a search that uses an `IfxDocDesc` document as a clue is shown in the next section.

For more information on the `IfxDocDesc` and `LLD_Locator` data types, see [Chapter 4, “Data Types.”](#)

Examples

The following statement searches for the specific word `multimedia` in the column **abstract** and includes other criteria in the `WHERE` clause:

```
SELECT title FROM reports
WHERE etx_contains(abstract, Row('multimedia'))
AND author = 'Joe Smith';
```

The following statement searches for either of the specific words `multimedia` or `video` in the column **abstract**, enables searching for letter transpositions and substitutions, and requests that synonyms from the list named **my_synonymlist** be included in the search:

```
SELECT title FROM reports
WHERE etx_contains(abstract,
Row('multimedia video',
'PATTERN_TRANS & PATTERN_SUBS & MATCH_SYNONYM = my_synonymlist'));
```

The following statement calls for the rank of a text search to be materialized as the statement local variable **rc1** and orders the returned rows by this rank:

```
SELECT rc1.score, title FROM reports
WHERE etx_contains (abstract,
Row('video'), rc1 # etx_ReturnType)
AND doc_no > 1005
ORDER BY 1;
```

The following statement executes a search on the **abstract** column, but instead of specifying a quoted string as the clue, it specifies an IfxDocDesc document stored as a file on the operating system as the clue:

```
SELECT title FROM reports
WHERE etx_contains (abstract,
  Row ( Row ('MS Word', '6.0',
    Row ('IFX_FILE', NULL::LLD_Lob,
      '/local0/excal/clue.txt')::LLD_Locator,
      NULL::LVARCHAR)::IfxDocDesc) );
```

The entire contents of the operating system file **/local0/excal/clue.txt** are automatically converted into the clue. Note that even though no tuning parameters are specified, the IfxDocDesc clue must still be encapsulated within a **Row()** constructor.

For additional examples of the **etx_contains()** operator, see [Chapter 2, “Text Search Concepts,”](#) and [Chapter 3, “Tutorial.”](#)

etx_CloseIndex()

The **etx_CloseIndex()** procedure closes an **etx** index.

Syntax

```
etx_CloseIndex( index_name )
```

Element	Purpose	Data Type
<i>index_name</i>	Name of the index to close.	LVARCHAR

Return Type

None.

Usage

The first time an **etx** index is used in a query (opened), resources such as shared memory are allocated in the database server, and they continue to be allocated even after the query has finished executing. Subsequent user sessions that use the **etx** index share these resources. Since new resources are not allocated for each session while the index is open, query performance is improved.

Once opened, an **etx** index does not automatically free up the shared resources until the database server shuts down. To force an index to be closed, and thus free up the resources, execute the procedure **etx_CloseIndex()**.



***Tip:** Informix recommends that **etx** indexes be left open, since the performance of queries that use the index is improved for subsequent user sessions. You need only use the **etx_CloseIndex()** procedure when database server resources become scarce and shared memory must be freed up.*

Example

The following example closes the **etx** index **reports_idx5**:

```
EXECUTE PROCEDURE etx_CloseIndex ('reports_idx5');
```

etx_CreateCharSet()

The **etx_CreateCharSet()** procedure creates a user-defined character set.

Syntax

```
etx_CreateCharSet (charset_name, file_name)
```

Element	Purpose	Data Type
<i>charset_name</i>	Name of your user-defined character set. If you enter a name longer than 18 characters, the name is silently truncated to 18 characters.	CHAR (18)
<i>file_name</i>	Absolute pathname of the operating system file from which the text search engine loads the character set. The file can be on either the server or the client machine. The client machine is searched first.	LVARCHAR

Return Type

None.

Usage

When you create an **etx** index on a column, you can specify the character set used to index the text data. The character set indicates which letters should be indexed; any characters in the text data that are not listed in the character set are converted to blanks. Use the CHAR_SET index parameter to specify the name of the character set. You must create a user-defined character set before you use it to create an **etx** index.

The Excalibur Text Search DataBlade module provides three built-in character sets: ASCII, ISO, and OVERLAP_ISO. Each of these built-in character sets include only alphanumeric characters and map lowercase letters to uppercase. This is sufficient for most text searches. For a complete description of the three built-in character sets, see [Appendix A](#), “Character Sets.”

There are times, however, when you might want to index nonalphanumeric characters or distinguish between lowercase and uppercase letters. In these cases you must define your own character set.

To define your own character set, first create an operating system file that specifies the characters you want to index. The next section describes in detail the structure of this operating system file.

Then create the character set by executing the `etx_CreateCharSet()` routine. The routine takes two parameters: the name you give the user-defined character set, and the full pathname of the operating system file that contains the characters to be indexed. The new user-defined character set is stored in the default sbspace.



Important: You cannot use the keywords `ASCII`, `ISO`, or `OVERLAP_ISO` (in any combination of uppercase and lowercase letters) as names for your user-defined character set, since these are reserved for the built-in character sets.

To use the user-defined character set, specify its name in the `CHAR_SET` index parameter of the `CREATE INDEX` statement.

Structure of the Operating System Character Set File

The operating system file consists of 16 lines of 16 hexadecimal numbers, plus optional lines that contain comments. Each position corresponds to an ASCII character. If you want the character in the position to be indexed, enter the character’s hexadecimal value. If you do not want the character to be indexed, enter `00`.

The ISO 8859-1 table in [Appendix A](#) lists the ISO 8859-1 character set that can be used as a reference when creating the operating system file.

Comments begin with either a slash, a hyphen, or a pound sign, and can appear anywhere in the file.

For example, if you want to create a user-defined character set that indexes hyphens (hexadecimal value 0x2D), underscores (hexadecimal value 0x5F), backslashes (hexadecimal value 0x5C), and forward slashes (hexadecimal value 0x2F), as well as the alphanumeric characters 0-9, a-z, and A-Z, and maps the lowercase letters a-z to uppercase, the operating system file would look like the following:

```
# Character set that indexes hyphens and
/ alphanumeric characters. All lower case letters
\ are mapped to upper case.
_ Note the different ways of specifying that a
# line is a comment.
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 2D 00 2F
30 31 32 33 34 35 36 37 38 39 00 00 00 00 00 00
00 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
50 51 52 53 54 55 56 57 58 59 5A 00 5C 00 00 5F
00 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
50 51 52 53 54 55 56 57 58 59 5A 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This is very similar to the built-in ASCII character set, except that hyphens, underscores, forward slashes, and backslashes are also indexed instead of being converted to blanks. These four characters are indexed because the position in the matrix for the each character contains its hexadecimal representation: 0x2D, 0x5F, 0x5C, and 0x2F, respectively.

All lowercase letters are mapped to uppercase by specifying the uppercase hexadecimal value in the lowercase letter position.

For example, uppercase letter A has a hexadecimal value of 0x41. The position in the matrix of uppercase A contains the hexadecimal value 0x41, thus uppercase A is indexed as uppercase A.

Example

However, the position in the matrix of lowercase a also contains the hexadecimal value 0x41 (which represents uppercase A) instead of the actual hexadecimal representation of lowercase a, 0x61. Thus, lowercase a is mapped to uppercase A, or in other words, lowercase a is indexed as if it were the same as uppercase A. The same is true for all the letters a-z and A-Z.

For more information on the ISO 8859-1 table, refer to [Appendix A](#), “Character Sets.”

Example

The following example creates a user-defined character set named **my_charset**. The search engine stores and loads the contents of **my_charset** from the file called **/local0/excal/my_char_set_file** on the operating system.

```
EXECUTE PROCEDURE etx_CreateCharSet  
('my_charset', '/local0/excal/my_char_set_file');
```


etx_CreateStopWlst()

The **etx_CreateStopWlst()** procedure creates a list of words that the text search engine ignores when it performs a search or builds an index.

Syntax

```
etx_CreateStopWlst (list_name, file_name, sbpace)
```

Element	Purpose	Data Type
<i>list_name</i>	Name of your stopword list. If you enter a name longer than 18 characters, the name is silently truncated to 18 characters.	CHAR (18)
<i>file_name</i>	Absolute pathname of the operating system file from which the text search engine loads the stopwords. The file can be on either the server or the client machine. The client machine is searched first.	LVARCHAR
<i>sbpace</i>	Optional parameter to specify the sbpace in which you want to store the stopword list. If you do not specify an sbpace, the database server stores the stopword list in the default sbpace.	CHAR (18)

Return Type

None.

Usage

A stopword list is a list of words that you want eliminated from both an **etx** index and the text search clue. A typical word list might include the prepositions *of*, *by*, *with*, and *so on*. Eliminating stopwords from the search process can significantly improve the performance of your searches.

When you create a stopwords list, you may use your own operating system file containing a list of stopwords or use the one provided by the DataBlade module:

```
$INFORMIXDIR/extend/ETX.<version>/wordlist/etx_stopwords.txt
```

In the preceding filename, *<version>* refers to the current version of the DataBlade module installed on your computer.

The format of the operating system file that contains the list of stopwords is one stopword per line. The file should consist of only standard ASCII characters and should not contain any proprietary formatting information. Do not include words that contain hyphens, apostrophes, or other non-alphanumeric characters, such as *isn't*, unless you create your index with a user-defined character set that includes these characters.



Important: An *etx* index can be associated with at most one stopwords list, which is specified at index creation via the *STOPWORD_LIST* index parameter. If you want to change the stopwords list associated with an *etx* index, you must first drop the index and then re-create it, specifying the name of the new stopwords list.

Example

The following example creates a stopwords list named **my_stopword**. The search engine stores and loads the contents of **my_stopword** from the operating system file **/local0/excal/stp_word**. The **etx_CreateStopWlst()** procedure stores the stopwords list in an sbspace named **sbsp1**.

```
EXECUTE PROCEDURE etx_CreateStopWlst  
('my_stopword', '/local0/excal/stp_word', 'sbsp1');
```

etx_CreateSynWlst()

The **etx_CreateSynWlst()** procedure creates a synonym list that the search engine uses to identify synonyms during a text search.

Syntax

```
etx_CreateSynWlst (list_name, file_name, sbpace)
```

Element	Purpose	Data Type
<i>list_name</i>	Name of your synonym list. If you enter a name longer than 18 characters, the name is silently truncated to 18 characters.	CHAR (18)
<i>file_name</i>	Absolute pathname of the operating system file from which the text search engine loads the synonym list. The file can be on either the server or the client machine. The client machine is searched first.	LVARCHAR
<i>sbpace</i>	Optional parameter to specify the sbpace in which you want to store the synonym list. If you do not specify an sbpace, the database server stores the synonym list in the default sbpace.	CHAR (18)

Return Type

None.

Usage

A synonym list is a list of words that you want the search engine to treat as equal. For example, suppose you specify `java` as a synonym for the root word `coffee` in your synonym list. The search engine will record a hit when it encounters the word `java` even though you specified the word `coffee` in your clue.

All words in a clue that are candidates for replacement by their synonyms must be listed as root words. For example, `java` is a synonym for the root word `coffee`. A synonym-matching search for the word `coffee` will record a hit if the text search engine finds the word `java`. The reverse situation, in which a search for `java` will record a hit if the search engine finds `coffee`, is only true if `java` is also listed as a root word, with `coffee` as one of its synonyms.

The format of the operating system file that contains the synonyms is one root word per line, followed by one or more synonyms, separated by blanks. The root word and its synonyms must all be on one line. Each line that contains text should be followed by a blank line, as shown by the following example:

```
ABANDON RELINQUISH RESIGN QUIT SURRENDER

ABILITY APTITUDE SKILL CAPABILITY TALENT

SLANT SLOPE INCLINATION TILT LEANING
```

The file should consist of only standard ASCII characters and should not contain any proprietary formatting information.



Important: *Be sure to include the extra blank line between the lines of text in the operating system synonym file, as described above. If you omit the blank lines, the DataBlade module will not return an error, but it will simply never find any synonyms during a synonym-matching search. Also be sure that all synonyms are spelled correctly.*

The Default Synonym List

The Excalibur Text Search DataBlade module creates a default synonym list called **etx_thesaurus** during registration. This default synonym list is consulted when you specify the `MATCH_SYNONYM` tuning parameter without specifying the name of a synonym list.

An error will be returned if you try to create a custom synonym list with the name **etx_thesaurus**.

If you want to change the contents of the default list, you must first drop the list using the procedure **etx_DropSynWlst()**, and then re-create it with the procedure **etx_CreateSynWlst()**.

The Excalibur Text Search DataBlade module uses the following operating system file when it creates the default synonym list **etx_thesaurus**:

```
$INFORMIXDIR/extend/ETX.<version>/wordlist/etx_thesaurus.txt
```

In the preceding filename, *<version>* refers to the current version of the DataBlade module installed on your computer. Informix recommends that you copy this file to a new location if you want to change the contents of the default synonym list instead of updating the original file; then use the name of the new file as a parameter to the **etx_CreateSynWlst()** routine.

Example

The following statement creates a synonym list named **my_synonym**. The search engine stores and loads the contents of **my_synonym** from the operating system file **/local0/excal/syn_file**. The **etx_CreateSynWlst()** procedure stores the synonym list in an sbspace named **sbsp2**.

```
EXECUTE PROCEDURE etx_CreateSynWlst  
  ( 'my_synonym', '/local0/excal/syn_file', 'sbsp2');
```

etx_DropCharSet()

The **etx_DropCharSet()** procedure drops a user-defined character set.

Syntax

```
etx_DropCharSet (charset_name)
```

Element	Purpose	Data Type
charset_name	Name of the user-defined character set you want to drop	CHAR (18)

Return Type

None.

Usage

Use the **etx_DropCharSet()** procedure to drop an existing user-defined character set. The database server drops the user-defined character set as long as no index is currently using it.

***Important:** Use the routine **etx_DropCharSet()** to drop only user-defined character sets that are currently not being used by any **etx** indexes. If you want to change the character set being used by an index, you must first drop the index and then re-create it, specifying the name of the new character set via the **CHAR_SET** index parameter.*

To determine if a user-defined character set is currently being used by an **etx** index, query the system catalog tables, as shown by the following sample query:

```
SELECT idxname, amparam
FROM sysindices
WHERE amparam like '%<charset_name>%';
```



In the preceding query, *<charset_name>* refers to the name of the user-defined character set you want to find information about. The search is case sensitive, so enter the name of the user-defined character set exactly as it was created.

If the query returns no rows, no index is currently using the specified user-defined character set. If the query returns one or more rows, the index or indexes returned in the **idxname** column are currently using the specified user-defined character set. The **amparam** column of the **sysindices** system table stores the index parameters used to create the **etx** index.

For example, the following query returns a row for each **etx** index that is currently using the user-defined character set **my_charset**:

```
SELECT idxname, amparam
FROM sysindices
WHERE amparam LIKE '%my_charset%';
```

Example

The following example drops the user-defined character set **my_charset**:

```
EXECUTE PROCEDURE etx_DropCharSet ('my_charset');
```

etx_DropStopWlst()

The **etx_DropStopWlst()** procedure drops a stopword list.

Syntax

```
etx_DropStopWlst (list_name)
```

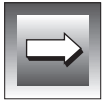
Element	Purpose	Data Type
<i>list_name</i>	Name of the stopword list you want to drop	CHAR (18)

Return Type

None.

Usage

Use the **etx_DropStopWlst()** procedure to drop an existing stopword list. The database server drops the stopword list as long as no index is currently using it.



Important: Use the routine **etx_DropStopWlst()** to drop only stopword lists that are currently not being used by any **etx** indexes. If you want to change the stopword list being used by an index, you must first drop the index and then re-create it, specifying the name of the new stopword list.

To determine if a stopword list is currently being used by an **etx** index, query the system catalog tables, as shown by the following sample query:

```
SELECT idxname, amparam
FROM sysindices
WHERE amparam like '%<stopwordlist_name>%';
```

In the preceding query, *<stopwordlist_name>* refers to the name of the stopword list you want to find information about. The search is case sensitive, so enter the name of the stopword list exactly as it was created.

If the query returns no rows, no index is currently using the specified stopword list. If the query returns one or more rows, the index or indexes returned in the **idxname** column are currently using the specified stopword list. The **amparam** column of the **sysindices** system table stores the index parameters used to create the **etx** index.

For example, the following query returns a row for each **etx** index that is currently using the stopword list **my_stopwordlist**:

```
SELECT idxname, amparam
FROM sysindices
WHERE amparam LIKE '%my_stopwordlist%';
```

Example

The following example drops the stopword list named **my_stopword**:

```
EXECUTE PROCEDURE etx_DropStopWlst ('my_stopword');
```

etx_DropSynWlst()

The **etx_DropSynWlst()** procedure drops a synonym list.

Syntax

```
etx_DropSynWlst ( list_name )
```

Element	Purpose	Data Type
<i>list_name</i>	Name of the synonym list you want to drop	CHAR (18)

Return Type

None.

Usage

Use the **etx_DropSynWlst()** procedure to drop an existing synonym list. The database server drops the synonym list as long as no index is using it.

Example

The following example drops the synonym list named **my_synonym**:

```
EXECUTE PROCEDURE etx_DropSynWlst ('my_synonym');
```

etx_GetHilite()

The **etx_GetHilite()** function returns the location of a clue in a document.

Syntax

```
etx_GetHilite (document, hilite_info)
```

Element	Purpose	Data Type
<i>document</i>	Name of the column in the table that contains the document for which you want highlighting information	One of BLOB, CLOB, LVARCHAR, CHAR, VARCHAR, IfxMRData, or IfxDocDesc
<i>hilite_info</i>	Name given to the statement local variable (SLV) of the etx_contains() operator that returns internal highlighting information	etx_ReturnType

Return Type

The **etx_GetHilite()** function returns `etx_HiliteType`, a named row data type defined by the Excalibur Text Search DataBlade module.

For more information on the `etx_HiliteType` row data type, refer to [Chapter 4, “Data Types.”](#)

Usage

The **etx_GetHilite()** function is used to return the location of a clue in a document or search text. This is referred to as *highlighting*.

You can only use the **etx_GetHilite()** function in the SELECT list of a query. It has two required parameters. The first is the name of the column that contains the document to be highlighted. The second is the name given to the SLV of the **etx_contains()** operator that executes the search.

The **etx_GetHilite()** function does not have as input the clue or search string that is to be highlighted. The function uses the clue specified as the second parameter to the **etx_contains()** operator as the highlight string.

The **etx_GetHilite()** function returns highlighting information via the row data type **etx_HiliteType**. This row data type consists of two fields that contain the two parts of the highlighting information: **vec_offset** and **viewer_doc**.

The **vec_offset** field contains offset information about the location of the clue in the document in the form of ordered pairs of integers. Each integer is separated by a blank. The first integer in the pair describes the offset of an instance of the clue in the document, and the second integer describes the length of the highlight. The first character in a document is assumed to have an offset of 0. The **viewer_doc** field contains the text document itself.

Since the data types of the **vec_offset** and **viewer_doc** fields are CLOB and BLOB, respectively, your application must use standard ESQL/C large object functions such as **ifx_lo_open()** and **ifx_lo_read()** or DataBlade API large object functions such as **mi_lo_open()** and **mi_lo_read()** to manipulate the contents.

For more information on the ESQL/C functions that are available to manipulate CLOB and BLOB data types, refer to the [INFORMIX-ESQL/C Programmer's Manual](#). For more information on the DataBlade API functions, refer to the [DataBlade API Programmer's Manual](#).

For more information on **etx_HiliteType** and its two fields, **vec_offset** and **viewer_doc**, refer to [Chapter 4, "Data Types."](#)



Important: If you use the **etx_GetHilite()** function in a query that returns more than one row, the function executes once for each row. This means that each row will have its own highlighting information, contained in the **etx_HiliteType** row data type returned by the **etx_GetHilite()** function. This highlighting information pertains only to the document contained in the row, not to any other document.



Important: If you use the **etx_GetHilite()** routine to highlight the result of a phrase search, all instances of each word in the clue will be highlighted, not just instances of the entire phrase. You execute a phrase search by setting either of the tuning parameters **SEARCH_TYPE = PHRASE_EXACT** or **SEARCH_TYPE = PHRASE_APPROX**.

*For example, if you execute a phrase search for the clue walk the dog and use the **etx_GetHilite()** routine to highlight the result, all instances of the individual words walk and dog (and the, if stopwords are significant) are highlighted in the resulting document, even if the instances are not close to each other.*

Examples

This section provides both a simple and more complex example of how to use the **etx_GetHilite()** function.

A Simple Example

The following example shows a simple use of the **etx_GetHilite()** function:

```
SELECT etx_GetHilite (description, rc) FROM videos
WHERE etx_contains(description,
    'multimedia', rc # etx_ReturnType);
```

The function **etx_GetHilite()** is executed on the **description** column, the same column being searched with the **etx_contains()** operator. The SLV declared in the **etx_contains()** operator, **rc**, is passed as the second parameter to the **etx_GetHilite()** function. The string to be highlighted is the clue of the **etx_contains()** operator, **multimedia**.

A Complex Example

The following example shows how to manipulate the **vec_offset** and **viewer_doc** data returned by the **etx_GetHilite()** function via a user-defined routine that uses the DataBlade API.

The sample routine, called **my_ViewHilite()**, takes as a parameter the **etx_HiliteType** data returned by the **etx_GetHilite()** routine, converts it to an **LVARCHAR** value, and inserts text before and after every instance of the highlight string.

The **my_ViewHilite()** routine takes three input arguments: the first is the returned data, of data type **etx_HiliteType**, returned by the **etx_GetHilite()** routine, the second is the text you want to appear before the highlight string, and the third is the text you want to appear after the highlight string.



Important: The sample routine **my_ViewHilite()** has a limited use. Since it returns an `LVARCHAR` value, it only works on relatively small documents. It also requires the documents to be in ASCII format.

The C Source Code

This section contains the C source code that is used to create the **my_ViewHilite()** routine. It uses the DataBlade API to manipulate the data passed to it by the **etx_GetHilite()** routine.

For more information on using the DataBlade API, refer to the [DataBlade API Programmer's Manual](#).

```
#include "mi.h"
#include <stdlib.h>

#define MAX_DOC_SIZE 32767

/* This is a sample UDR for viewing highlights */

mi_lvarchar *my_viewhilite(MI_ROW *hilite_type,
    mi_lvarchar *hilite_prefix_lv, mi_lvarchar *hilite_suffix_lv,
    MI_FPARAM *fp)
{
    mi_integer rc;
    MI_CONNECTION *conn = NULL;
    mi_integer    vec_offset_fd = -1,    viewer_doc_fd = -1;
    MI_LO_HANDLE *vec_offset_blob = NULL, *viewer_doc_blob = NULL;
    mi_lvarchar   *hilite_doc_lv = NULL;
    mi_string     *hilite_doc = NULL;
    mi_string     *hilite_doc_cp = NULL;
    mi_integer    len;
    mi_integer    offset, offset_len, offset_last;
    mi_integer    hilite_prefix_len, hilite_suffix_len;
    mi_string     *hilite_prefix, *hilite_suffix;

    /* If the first argument is null then return an error */

    if (mi_fp_argisnull(fp, 0))
        goto error;

    /* If the second or third arguments are null then
     * the prefix/suffix strings default to toggle inverse video for vt100.
     */

    if (mi_fp_argisnull(fp, 1))
        hilite_prefix = "\033[7m";
    else
        hilite_prefix = mi_lvarchar_to_string(hilite_prefix_lv);
    hilite_prefix_len = strlen(hilite_prefix);

    if (mi_fp_argisnull(fp, 2))
        hilite_suffix = "\033[0m";
    else
        hilite_suffix = mi_lvarchar_to_string(hilite_suffix_lv);
    hilite_suffix_len = strlen(hilite_suffix);

    /* establish a connection */

    if (!(conn = mi_open(NULL, NULL, NULL)))
        goto error;

    /* decompose the highlights */
```

```
rc = mi_value(hilite_type, 0, (MI_DATUM)&vec_offset_blob, &len);
rc = mi_value(hilite_type, 1, (MI_DATUM)&viewer_doc_blob, &len);

/* open the blobs */

vec_offset_fd = mi_lo_open(conn, vec_offset_blob, MI_LO_RDONLY);
viewer_doc_fd = mi_lo_open(conn, viewer_doc_blob, MI_LO_RDONLY);

/* parse the highlights */

hilite_doc_cp = hilite_doc = mi_alloc(MAX_DOC_SIZE);
offset_last = 0;
while ((offset = get_offset_num(conn, vec_offset_fd)) >= 0 &&
       (offset_len = get_offset_num(conn, vec_offset_fd)) > 0)
{

/* copy from the end of the last highlight to the beginning of this */

len = offset - offset_last;
if (hilite_doc + MAX_DOC_SIZE < hilite_doc_cp + len)
    len = MAX_DOC_SIZE - (hilite_doc_cp - hilite_doc);
rc = mi_lo_read(conn, viewer_doc_fd, hilite_doc_cp, len);
hilite_doc_cp += rc;
offset_last += len + offset_len;

/* copy the highlighted part into the buffer */

if (hilite_doc + MAX_DOC_SIZE < hilite_doc_cp + hilite_prefix_len)
    hilite_prefix_len = MAX_DOC_SIZE - (hilite_doc_cp - hilite_doc);
memcpy(hilite_doc_cp, hilite_prefix, hilite_prefix_len);
hilite_doc_cp += hilite_prefix_len;

if (hilite_doc + MAX_DOC_SIZE < hilite_doc_cp + offset_len)
    offset_len = MAX_DOC_SIZE - (hilite_doc_cp - hilite_doc);
rc = mi_lo_read(conn, viewer_doc_fd, hilite_doc_cp, offset_len);
hilite_doc_cp += rc;

if (hilite_doc + MAX_DOC_SIZE < hilite_doc_cp + hilite_suffix_len)
    hilite_suffix_len = MAX_DOC_SIZE - (hilite_doc_cp - hilite_doc);
memcpy(hilite_doc_cp, hilite_suffix, hilite_suffix_len);
hilite_doc_cp += hilite_suffix_len;
}

/* copy the rest of the file */

len = MAX_DOC_SIZE - (hilite_doc_cp - hilite_doc);
rc = mi_lo_read(conn, viewer_doc_fd, hilite_doc_cp, len);
hilite_doc_cp += rc;

/* terminate the string */

*hilite_doc_cp = '\0';
hilite_doc_lv = mi_string_to_lvarchar(hilite_doc);
```



```

error:
    if (vec_offset_fd >= 0)
        rc = mi_lo_close(conn, vec_offset_fd);
    if (viewer_doc_fd >= 0)
        rc = mi_lo_close(conn, viewer_doc_fd);
    if (hilite_doc)
        mi_free(hilite_doc);
    if (!hilite_doc_lv)
        mi_fp_setreturnisnull(fp, 0, MI_TRUE);
    if (conn)
        mi_close(conn);

    return hilite_doc_lv;
}

static mi_integer get_offset_num(MI_CONNECTION *conn, mi_integer vec_offset_fd)
{
    mi_integer num = -1;
    mi_string buf[100];
    mi_string *pbuf = buf;

    /* skip spaces */

    while ((num = mi_lo_read(conn, vec_offset_fd, pbuf, 1)) > 0 &&
           isspace(*pbuf))
        ;

    /* save this character if not at eof yet */

    if (num > 0)
        pbuf++;
    else
        return -1;

    /* read the number */

    while ((num = mi_lo_read(conn, vec_offset_fd, pbuf, 1)) > 0 &&
           !isspace(*pbuf))
        pbuf++;
    *pbuf = '\0';

    num = atoi(buf);

    return num;
}

```

Creating the Shared Object

This section provides a sample makefile used to compile and link the C source code on Solaris 2.5 and create a shared object. The makefile uses `my_view.c` as the name of the C source code file and `my_view.so` as the name of the shared object you want to create.

You might need to modify the makefile to conform to your specific compiler and linker.

```
# Sample makefile to create the shared object

SHLIBFLAG = -dy -G
LD_FLAGS = $(SHLIBFLAG)
LD = ld $(LD_FLAGS)

CINCL = -I $(INFORMIXDIR)/incl/dmi -
I$(INFORMIXDIR)/incl/public
CPIC = -KPIC
CFLAGS = -g $(CINCL) $(CPIC)

all: my_view.so

my_view.so: my_view.o
    $(LD) -o $@ my_view.o
    chmod 755 my_view.so

clean:
    rm -rf my_view.so my_view.o
```

Creating the User-Defined Routine

The following SQL statement shows an example of creating the USER-DEFINED ROUTINE **my_ViewHilite()**:

```
-- create the function my_ViewHilite

CREATE FUNCTION my_ViewHilite (hilite etx_hilitetype,
    prefix lvvarchar, suffix lvvarchar)
RETURNS lvvarchar
WITH (HANDLESNULLS)
EXTERNAL NAME
    '/local/excal/my_view.so (my_viewhilite)'
LANGUAGE C NOT VARIANT;
```

Executing the User-Defined Routine

The following SQL statement shows how you can execute the **my_ViewHilite()** routine:

```
-- example of running the function my_ViewHilite

SELECT my_ViewHilite (etx_GetHilite(abstract, rc), '<b>', '</b>')
FROM reports
WHERE etx_contains (abstract, 'concepts', rc # etx_ReturnType);
```

The **SELECT** statement returns all documents that contain the word **concepts** in the **abstract** column. The **my_ViewHilite()** routine returns an **LVARCHAR** value that contains the text of the documents, with every instance of the word **concepts** surrounded by the HTML tags **** and ****. The output might look something like the following sentence:

```
This book explains the main <b>concepts</b> of relational
database management systems.
```



Tip: A copy of the C source code and the sample makefile to create the shared object, called **my_view.c** and **Makefile** respectively, are included in the distribution media in the directory **\$INFORMIXDIR/extend/ETX.<version>/examples**, where **<version>** refers to the version of the DataBlade module installed on your computer.

etx_Release()

The **etx_Release()** function returns version information, such as the installed version and the release date, for the Excalibur Text Search DataBlade module.

Syntax

```
etx_Release()
```

The **etx_Release()** function does not take any parameters.

Return Type

The **etx_Release()** function returns LVARCHAR.

Usage

The information returned by the **etx_Release()** function includes:

- the version of the installed Excalibur Text Search DataBlade module.
- the date the installed Excalibur Text Search DataBlade module was compiled.
- the full version of the database server with which the Excalibur Text Search DataBlade module was compiled.
- the version of the GLS library with which the Excalibur Text Search DataBlade module was compiled.
- the version of the Excalibur text retrieval library (TRL) with which the Excalibur Text Search DataBlade module was compiled.

Example

The following example shows how to execute the **etx_Release()** function:

```
EXECUTE FUNCTION etx_Release();
```

txt_Release()

The **txt_Release()** function returns version information, such as the installed version and the release date, for the Text Descriptor DataBlade module.

Syntax

```
txt_Release()
```

The **txt_Release()** function does not take any parameters.

Return Type

The **txt_Release()** function returns LVARCHAR.

Usage

The information returned by the **txt_Release()** function includes:

- the version of the installed Text Descriptor DataBlade module.
- the date the installed Text Descriptor DataBlade module was compiled.
- the full version of the database server with which the Text Descriptor DataBlade module was compiled.
- the version of the GLS library with which the Text Descriptor DataBlade module was compiled.

Example

The following example shows how to execute the **txt_Release()** function:

```
EXECUTE FUNCTION txt_Release();
```

etx Index Parameters

Overview of the etx Index Parameters	6-3
WORD_SUPPORT	6-4
PHRASE_SUPPORT	6-4
CHAR_SET	6-5
STOPWORD_LIST	6-6
INCLUDE_STOPWORDS	6-7

T

his chapter describes the index parameters that can be used to customize an **etx** index. They are specified in the USING clause of the CREATE INDEX statement. Use these parameters if you have an idea of the types of searches you plan to perform and want to customize your **etx** index to reflect these types of searches.

Overview of the etx Index Parameters

The following table summarizes the index parameters you can use to customize the **etx** index. Each parameter is discussed in greater detail later in the chapter.

Index Parameter	Brief Description	Rules	Default
WORD_SUPPORT	Specifies the type of keyword searches the index will support	One of EXACT or PATTERN	EXACT
PHRASE_SUPPORT	Specifies the accuracy level of phrase searches	One of NONE, MEDIUM, or MAXIMUM	NONE
CHAR_SET	Specifies the character set translation table that the index will use	One of the built-in character sets ASCII, ISO, or OVERLAP_ISO, or the name of an existing user-defined character set	ASCII
STOPWORD_LIST	Specifies the list of stopwords that will not be indexed	Value must be an existing stopwords list	No default
INCLUDE_STOPWORDS	Specifies that the stopwords in the list specified by the STOPWORD_LIST index parameter should be indexed	One of TRUE or FALSE Must be used together with the STOPWORD_LIST index parameter	FALSE



Index parameters are specified in the USING clause of the CREATE INDEX statement when you create an **etx** index. The parameters must always take the form **<parameter>='<value>'**, such as **CHAR_SET='ISO'**, as shown in the following example:

```
CREATE INDEX reports_idx2 ON reports (title etx_char_ops)
  USING etx (STOPWORD_LIST = 'my_stopwordlist',
  CHAR_SET = 'ISO') IN sbsp1;
```

Important: *You cannot change the characteristics of an **etx** index once you have created it. The only way to change a customized **etx** index is to drop and re-create it.*

For more examples of creating **etx** indexes with these index parameters, refer to [Chapter 3, “Tutorial.”](#)

WORD_SUPPORT

The WORD_SUPPORT index parameter allows you to specify the type of word search the index supports. Specify EXACT if you plan to perform exact word searches only. Specify PATTERN if you want the index to support both exact and pattern searches. The default for this parameter is EXACT.

For an explanation of what exact and pattern searches are, see [Chapter 1, “DataBlade Module Overview.”](#)

PHRASE_SUPPORT

Set this parameter only if you plan to perform phrase searches or proximity searches. If you are unsure what phrase and proximity searches are, see [Chapter 1.](#)

The PHRASE_SUPPORT index parameter allows you to specify the type of phrase searches the index supports. You can set PHRASE_SUPPORT to one of the following settings:

- ★ NONE
- ★ MEDIUM

★ MAXIMUM

The NONE setting means that the index does not support phrase or proximity searches.

Setting PHRASE_SUPPORT to lower levels of phrase and proximity support can cause false matches. However, lower levels of phrase support have the following two advantages:

- Improved performance for searches that use the index
- Reduced disk space consumption by the index

A setting of MEDIUM results in a low accuracy of matches, while MAXIMUM results in the highest accuracy. The default setting is NONE.

Given the nature of the Excalibur search engine, a false hit from a phrase search is more likely with the lower settings of PHRASE_SUPPORT and with smaller phrases. As the size of the phrase grows, the probability of a false hit decreases.

Consider using a MEDIUM setting if one or more of the following circumstances is true:

- Disk space is a factor.
- You can tolerate a small number of false hits.
- The search phrases you plan to use are lengthy.

Consider using a MAXIMUM setting if one or more of the following circumstances is true:

- Disk space is not a factor.
- Accuracy is important.
- The search phrases you plan to use are brief.

CHAR_SET

The CHAR_SET parameter allows you to specify which characters in your text data will be indexed. Characters that are not indexed are treated as white space.

You can set `CHAR_SET` to one of the built-in character sets (ASCII, ISO, or `OVERLAP_ISO`) or to the name of an existing user-defined character set.

The ASCII setting means that only the alphanumeric characters 0 through 9, A through Z, and a through z will be indexed. If the `CHAR_SET` index parameter is not specified when you create an `etx` index, the ASCII table will be used by default.

When using the ASCII setting, words that contain international characters may actually be indexed as multiple words. For example, the word `cañon`, if indexed using the ASCII character set, is indexed as two words, `ca` and `on`, because the nonindexed character `ñ` is treated as white space.

The ISO setting means that both alphanumeric characters and ISO Latin-1 alphabetic characters are indexed. This 68-character set should be used to index data that might contain international characters from the ISO Latin-1 set.

The `OVERLAP_ISO` setting means that the same character set as the ISO setting is indexed, but similar-looking characters are grouped together so that a word will match as long as corresponding letters are from the same group.

Refer to [Appendix A](#), “Character Sets,” for a full description of the characters that make up the ASCII, ISO, and `OVERLAP_ISO` character sets.

If the built-in character sets are inadequate for your text documents, you can define your own character set that defines the characters you want to index. You must create the user-defined character set prior to specifying it as an option to the `CHAR_SET` index parameter. Use the `etx_CreateCharSet()` routine to create your own character set.

Refer to [Chapter 5](#), “Routines” for more information on using the `etx_CreateCharSet()` routine to create your own character set.

STOPWORD_LIST

The `STOPWORD_LIST` parameter allows you to specify a list of words that you do not want indexed. These *stopwords* are words that are usually irrelevant in a text search, such as `and`, `by`, or `the`. All words on this list are automatically eliminated from text searches, significantly improving performance.

If no stopwords list is specified when you create an **etx** index, all words in a document are indexed. In this case, words such as *the* and *an* become as relevant in a search as words such as *video* and *multimedia*, and searches will return more rows, many of which might not be useful. The **etx** index will need to be significantly larger to include every word in a document, and searches would thus be slower.

The stopwords list must already exist before you specify it as an index parameter. To create a stopwords list, use the routine **etx_CreateStopWlst()**, and pass it the name of an operating system file that contains a list of stopwords.

When you create a stopwords list, you can use your own file containing a list of stopwords, or you can use the one provided by the Excalibur Text Search DataBlade module:

```
$INFORMIXDIR/extend/ETX.<version>/wordlist/etx_stopwords.txt
```

In the preceding filename, *<version>* refers to the current version of the DataBlade module installed on your computer.

Refer to [Chapter 5, “Routines,”](#) for more detailed information on creating stopwords lists with the **etx_CreateStopWlst()** routine.

INCLUDE_STOPWORDS

If you specify the index parameter **STOPWORD_LIST** when you create an **etx** index, the words in the list are excluded from the index and are ignored in the clue when you execute subsequent text searches. This behavior is desirable for most searches. Occasionally, however, you might want to execute searches in which stopwords *are* relevant. For example, you might want to search for the exact phrase *to be or not to be* and do not want the stopwords to be excluded.

To force the stopwords specified by the **STOPWORD_LIST** parameter to be indexed, specify the **INCLUDE_STOPWORDS='TRUE'** parameter when you create the **etx** index.



In this case, stopwords will be indexed, similar to the default behavior, but they will be considered part of the clue only if the tuning parameter `CONSIDER_STOPWORDS` is specified in the **etx_contains()** operator. If you do not specify `CONSIDER_STOPWORDS` during a search, the stopwords will be ignored.

Important: *The `CONSIDER_STOPWORDS` tuning parameter of the **etx_contains()** operator works only if `INCLUDE_STOPWORDS` was specified at **etx** index creation.*

The index parameter `INCLUDE_STOPWORDS` should always be used together with `STOPWORD_LIST`. The only way the DataBlade module can recognize stopwords is by consulting the stopword list associated with the index, specified via the `STOPWORD_LIST` index parameter. If you specify `INCLUDE_STOPWORDS` alone, the parameter is essentially ignored.

Refer to [Chapter 5, “Routines,”](#) for more information on the **etx_contains()** operator and its tuning parameters.

DataBlade Module Administration

Retrieving DataBlade Module Version Information	7-3
Sbspaces	7-4
Default Sbspace	7-4
Logging of Sbspaces	7-5
Parameters of the onspaces Utility	7-6
DataBlade Dependencies	7-7
Estimating the Size of an etx Index	7-7
Internal Structure of an etx Index	7-9
etx Index Sharing	7-10
Advantages and Disadvantages of Text Storage Data Types	7-10
BLOB	7-11
CLOB	7-11
LVARCHAR	7-11
IfxDocDesc	7-12
IfxMRData	7-12
Performance Tips	7-13
Database Server Configuration	7-13
Loading Data	7-14

T

his chapter discusses administrative issues of the Excalibur Text Search DataBlade module. It includes such topics as the configuration and use of sbspaces, the size and internal structure of an **etx** index, benefits and disadvantages of using the various supported data types, and tips on improving performance.

Retrieving DataBlade Module Version Information

Use the **etx_Release()** function to return version information for the Excalibur Text Search DataBlade module. The information returned by this function includes:

- the version of the installed DataBlade module.
- the date the installed DataBlade module was compiled.
- the full version of the database server with which the DataBlade module was compiled.
- the version of the GLS library with which the DataBlade module was compiled.
- the version of the Excalibur text retrieval library (TRL) with which the DataBlade module was compiled.

Use the EXECUTE FUNCTION command to execute the **etx_Release()** function, as shown in the following example:

```
EXECUTE FUNCTION etx_Release();
```

The Text Descriptor DataBlade module has an equivalent function: **txt_Release()**. For more information on the **etx_Release()** and **txt_Release()** functions, refer to [Chapter 5, “Routines.”](#)

Sbspaces

The Excalibur Text Search DataBlade module uses sbspaces in a variety of ways. This section covers the minimum you must know about sbspaces to correctly register and use the DataBlade module. For more detailed information about sbspaces in general, refer to the administrator's guide for your database server.

Default Sbspace

During registration, the Excalibur Text Search DataBlade module creates a default synonym list and stores it in the default sbspace. It also sets up internal directories in the default sbspace for stopword lists and user-defined character sets. Therefore, you must create the default sbspace before you register the DataBlade module into any database, or the registration will fail.

To create the default sbspace

1. Set the ONCONFIG parameter SBSPACENAME to the name of your default sbspace.

For example, to name the default sbspace **sbsp1**:

```
SBSPACENAME      sbsp1 # Default sbspace name
```

You must update the ONCONFIG file before you start the database server.

2. Use the **onspaces** utility to create the sbspace.

The following example shows how to create an sbspace called **sbsp1** in the partition **/dev/sbspace**. The sbspace has an initial offset of 0 and a size of 100 MB:

```
% onspaces -c -S sbsp1 -g 2 -p /dev/sbspace -o 0 -s 100000
```

The DataBlade module uses the default sbspace for storing other objects as well. For example, if you do not specify the name of an sbspace when you create an **etx** index, synonym list, or stopword list, all are stored in the default sbspace. User-defined character sets are always stored in the default sbspace. Be sure the default sbspace is large enough to hold all of these objects.

You can use the **FileToBLOB()** function to test whether the default sbspace has been created and configured correctly, as shown in the following example executed in DB-Access:

```
execute function FileToBLOB ('/tmp/some.txt', 'server');
```

The file **/tmp/some.txt** can contain any type of text, but it should be fully accessible to the user who started the database server.

If the function returns without an error, the default sbspace has been created and configured correctly.

For more information on the **FileToBLOB()** function, refer to the [Informix Guide to SQL: Syntax](#).

Logging of Sbspaces

By default, logging is turned OFF when you create an sbspace with the **onspaces** utility. Since logging of sbspaces creates a significant resource overhead, many users decide to turn it off.

This means that if, for example, there is a power outage while an **etx** index is being updated, the index might become corrupt. If the sbspace that holds the index has logging turned off, the changes up to the point of corruption can not be backed out. In this case, to ensure index integrity, the index must be dropped and recreated. If logging is turned on, however, the **etx** index can be recovered normally.

To turn on logging for an sbspace, use the **-Df "LOGGING=ON"** option of the **onspaces** utility. The following example is similar to the example in the preceding section, except that logging is turned on for the **sbsp1** sbspace:

```
% onspaces -c -S sbsp1 -g 2 -p /dev/sbpace -o 0 -s 100000 -Df "LOGGING=ON"
```

Important: *The database server incurs an overhead when logging is turned on for sbspaces. This overhead can degrade the performance of queries that use **etx** indexes since they are stored in sbspaces.*



Parameters of the onspaces Utility

One of the options to the **onspaces** utility is **-Df**, which is used to list the default specifications for the smart large objects stored in the newly created sbospace. Informix recommends that you change the following two specifications for sbospaces that will contain **etx** indexes:

- ★ **AVG_LO_SIZE**
- ★ **MIN_EXT_SIZE**

As an **etx** index grows, some of the smart large objects that contain the index also grow. The smart large object extensions that are created as a result can be costly in both access time and consumption of metadata space. If you specify a relatively large **MIN_EXT_SIZE**, you minimize the number of these extensions. The default values for **MIN_EXT_SIZE** and **AVG_LO_SIZE** are 4 and 64, respectively. Informix recommends that you increase these values substantially, to 1000 and 2000, for example.

The following example shows how to specify new values for **AVG_LO_SIZE** and **MIN_EXT_SIZE** with the **onspaces** utility:

```
% onspaces -c -S sbbsp1 -g 2 -p /dev/sbospace -o 0 \  
-s 1000000 -Df "AVG_LO_SIZE=1000,MIN_EXT_SIZE=2000"
```

The following output from the utility **oncheck -cS sbbsp1** shows the effect of setting these options. The table whose **etx** index is stored in the sbospace **sbbsp1** contains 100,000 documents, averaging 2 KB in size. The number of extensions (column 7, marked **Extns**) is very low.

Large Objects

ID	Ref		Size	Alloced		Creat		Last	
Sbs#	Chk#	Page#	Cnt	(Bytes)	Pages	Extns	Flags	Modified	
1	3	1	1	25	1000	2	----H	Tue Jan 20 12:22:56 1998	
2	3	2	1	868864	1000	2	----H	Tue Jan 20 12:22:56 1998	
3	3	3	1	4147160	3000	3	----H	Tue Jan 20 12:22:56 1998	
4	3	4	1	512	1000	2	----H	Tue Jan 20 12:22:56 1998	
5	3	5	1	42995156	22000	3	----H	Tue Jan 20 12:22:56 1998	
6	3	6	1	806912	1000	2	----H	Tue Jan 20 12:22:56 1998	

For more information on the **onspaces** and **oncheck** utilities and the **ONCONFIG** parameter **SBSPACENAME**, refer to the administrator's guide for your database server.

DataBlade Dependencies

The Excalibur Text Search DataBlade module uses routines and data types defined by the following two products:

- Text Descriptor DataBlade module
- LOB Locator DataBlade module

The LOB Locator DataBlade module is shipped with your database server. It defines the data type LLD_Locator used by the named row type IfxDocDesc.

Refer to the [LOB Locator DataBlade Module Programmer's Guide](#) for more information on this DataBlade module.

The Text Descriptor DataBlade module is packaged together with the Excalibur Text Search DataBlade module on the same distribution media. It defines the data types IfxDocDesc and IfxMRData.

Refer to [Chapter 4, "Data Types,"](#) for more information on these two data types.

Estimating the Size of an etx Index

The size of an **etx** index can vary widely and depends on a number of factors. These factors include:

- The nature of the data to be indexed.
If your documents contain a lot of numerical data, such as the data found in financial reports, and you specify a character set that indexes numeric characters, each string of numbers will be indexed as if it were a word. This can increase the number of unique words in the index.
- The index parameters that you specify.
If you specify either of the index parameters PHRASE_SUPPORT=MEDIUM or PHRASE_SUPPORT=MAXIMUM, the index will be two to three times larger than if you specify PHRASE_SUPPORT=NONE.

- Stopword lists.
An **etx** index built with a stopwords list (specified by the index parameter `STOPWORD_LIST='my_stopwordlist'`) will generally be slightly smaller than an index that contains all the words in a document. However, if you also specify the index parameter `INCLUDE_STOPWORDS='TRUE'`, the index will be approximately 50% larger.
- The total number of words to be indexed.
Predicting the size of an **etx** index is made more complicated by the fact that it is not directly proportional to the number of documents or words in a document. The following table shows **etx** index sizes for various combinations of index parameters.

Index Parameter				Total Index Size, in Disk Pages, for an etx Index Containing 100,000 Documents
Word Support	Phrase Support	Stopword List	Include Stopwords	
exact	none	no	false	29 KB
exact	none	yes	false	28 KB
exact	medium	no	false	69 KB
exact	maximum	no	false	87 KB
exact	maximum	yes	false	62 KB
exact	maximum	yes	true	87 KB
pattern	none	no	false	34 KB
pattern	none	yes	true	34 KB
pattern	medium	no	false	73 KB
pattern	maximum	no	false	92 KB
pattern	maximum	yes	false	67 KB
pattern	maximum	yes	true	92 KB

The average size of the documents in the preceding table is 2 KB.

Internal Structure of an etx Index

An **etx** index is composed of 6 to 13 smart large objects. The number of smart large objects depends on the combination of `WORD_SUPPORT` and `PHRASE_SUPPORT` index parameters used in the `CREATE INDEX` statement. The following table shows how many smart large objects make up an **etx** index for each combination of index parameters.

Number of Smart Large Objects	Index Parameter Combination
6	<code>WORD_SUPPORT=EXACT, PHRASE_SUPPORT=NONE</code>
9	<code>WORD_SUPPORT=EXACT, PHRASE_SUPPORT=MEDIUM</code>
10	<code>WORD_SUPPORT=EXACT, PHRASE_SUPPORT=MAXIMUM</code>
9	<code>WORD_SUPPORT=PATTERN, PHRASE_SUPPORT=NONE</code>
12	<code>WORD_SUPPORT=PATTERN, PHRASE_SUPPORT=MEDIUM</code>
13	<code>WORD_SUPPORT=PATTERN, PHRASE_SUPPORT=MAXIMUM</code>



Important: The default value for the `WORD_SUPPORT` and `PHRASE_SUPPORT` index parameters are `EXACT` and `NONE`, respectively.

The contents of these smart large objects are beyond the scope of this discussion. However, it is important to understand that these smart large objects exist and that, as the **etx** index grows in size (during initial creation and as more documents are inserted into the table), some of these smart large objects will also grow.

This knowledge is useful for tuning the parameters of the subspace in which the **etx** index resides, which is discussed in the section [“Parameters of the onspaces Utility” on page 7-6](#).

etx Index Sharing

The first time an **etx** index is used in a query (opened), resources such as shared memory are allocated in the database server and they continue to be allocated even after the query has finished executing. Subsequent user sessions that use the **etx** index share these resources. Because new resources are not allocated for each user session while the index is open, query performance is improved.

Although query performance is improved for users who use the cached index information while the **etx** index is open, the first user who opens the index incurs the initial cost of storing index information in memory. This means that the first query that opens the **etx** index might run a little slower than subsequent queries.

Once opened, an **etx** index does not automatically free up the shared resources until the database server shuts down. The one exception is during a rollback of a transaction that uses the index. You can force an **etx** index to close, and thus free up the resources, by executing the procedure **etx_CloseIndex()**. The next user who reopens the **etx** index will again incur the initial cost of storing index information in memory.

For more information on using the **etx_CloseIndex()** procedure, refer to [Chapter 5, “Routines.”](#)

Advantages and Disadvantages of Text Storage Data Types

This section describes the advantages and disadvantages of storing text documents in the following supported data types: BLOB, CLOB, LVARCHAR, IfxDocDesc, and IfxMRData.

Although you can also build an **etx** index on columns of type CHAR and VARCHAR, their small storage capacity make them unlikely choices for storing text documents and they are not discussed in this section.

BLOB

BLOB columns work well for large documents that contain binary data, such as Microsoft Word or Acrobat PDF. A column of type BLOB can contain documents of essentially unlimited size, since the data is stored in sbspaces.

There is, however, a significant amount of resource overhead when you store documents in sbspaces. If all your documents are smaller than 2 KB, you might consider storing them in a column of type LVARCHAR.

CLOB

The CLOB data type should be used for large documents that contain only standard ASCII text, such as HTML and SGML documents. A column of type CLOB can contain documents of essentially unlimited size, since the data is stored in sbspaces.

There is, however, a significant amount of resource overhead when you store documents in sbspaces. If your documents are smaller than 2 KB, you might consider storing them in a column of type LVARCHAR.



***Tip:** If your documents contain only standard ASCII text, but are a mixture of small (under 2 KB) and large (over 2 KB) sizes, consider using the IfxMRData data type, described later in this section and in [Chapter 4, “Data Types.”](#)*

LVARCHAR

Text documents inserted into columns of type LVARCHAR are stored in dbspaces, just like data inserted into the standard SQL character data types such as CHAR and VARCHAR. This incurs less resource overhead than storing documents in sbspaces, which is where data in BLOB and CLOB columns are stored.

The disadvantage of storing documents in columns of type LVARCHAR is that the size of the documents is limited to 2 KB.

IfxDocDesc

The main advantage of using a column of type IfxDocDesc is that you can choose to store your text documents either on the operating system file system or in the database itself. A single column of type IfxDocDesc can have rows of each storage type. You specify how you want to store the documents via the **lo_protocol** field of the **location** field of IfxDocDesc by entering one of IFX_FILE, IFX_CLOB, or IFX_BLOB.

One of the disadvantages of using the IfxDocDesc data type, however, is its complexity. IfxDocDesc is a row type whose **location** field is also a row type, LLD_Locator. Inserting into a column of type IfxDocDesc requires two uses of the **Row()** constructor, which though not impossible, is more complex than inserting into the other supported data types.

A disadvantage to storing your document in a file on the operating system file system is that it is up to you to maintain the document and be sure that it is not moved or renamed. Only a pointer to the file is stored in the database, not the data itself. If you edit the operating system file outside the context of the database server, the **etx** index will not be automatically updated. In this case you must manually update the index by either issuing an UPDATE statement to update the row that contains the pointer to the file or dropping and re-creating the **etx** index.

For more detailed information on the IfxDocDesc data type, see [Chapter 4, “Data Types.”](#)

IfxMRData

The main advantage of using the multirepresentational data type IfxMRData is that it dynamically determines whether to store your document in an LVARCHAR data type or a CLOB data type, depending on the size of the document.

This feature is useful if the documents you want to store in a single column of a table vary greatly in size, where some fit in an LVARCHAR data type and some do not. It is preferable to store documents in LVARCHARs if at all possible, since it is faster to insert documents into and retrieve documents from this data type than into smart large object types such as CLOB. IfxMRData lets the data type itself decide where it should store the data so that you do not have to make this decision at the time you create the table.

IfxMRData is also designed to improve I/O performance of documents that contain only standard ASCII text. Therefore, if your documents do not contain binary data, storing them in a column of this data type might improve performance.

For more detailed information on the IfxMRData data type, see [Chapter 4, “Data Types.”](#)

Performance Tips

This section discusses tips on making the Excalibur Text Search DataBlade module perform optimally. It includes topics on configuring Informix Dynamic Server with Universal Data Option, how to optimally load text data into tables and indexes, and how to improve the performance of text queries.

The performance guide for your database server covers other performance issues that are also relevant to the Excalibur Text Search DataBlade module. Refer to that guide for information that is not covered in this section.

Database Server Configuration

When configuring Informix Dynamic Server with Universal Data Option, you can use the following ONCONFIG tuning parameters to optimize the performance of text searches that use **etx** indexes:

- ★ **BUFFERS**

The **BUFFERS** parameter specifies the maximum number of shared memory buffers that Universal Data Option user threads have available for disk I/O on behalf of client applications. The default number is 200. Increasing the number of buffers has been found to improve the performance of text searches.

- ★ **RA_PAGES**

The **RA_PAGES** parameter specifies the number of disk pages that the database server should attempt to read ahead during sequential scans of data or index records. Try setting this parameter to 64.

- ★ **RA_THRESHOLD**

The **RA_THRESHOLD** parameter specifies the number of unprocessed pages in memory that signals the database server to perform the next read ahead. If **RA_PAGES** is set to 64, setting **RA_THRESHOLD** to 33 could improve the performance of your text searches.

- ★ **RESIDENT**

The **RESIDENT** parameter specifies whether the resident portion of shared memory remains resident in operating system physical memory. If your operating system supports forced residency, specifying that the resident portion of shared memory not be swapped to disk by setting this parameter to 1 (ON) has been found to improve the performance of text searches.

- ★ **NOAGE**

The **NOAGE** parameter controls whether the UNIX operating system lowers the priority of database server processes as the processes run over a period of time. Setting this parameter to 1 (ON, which disables priority aging) has been found to improve the performance of text searches.

This section provides only a minimum of information about the **ONCONFIG** tuning parameters. Before you make any changes to your **ONCONFIG** file, refer to the administrator's guide for your database server for more detailed information on each parameter.

Loading Data

As with all databases and indexing methods, spreading the text data and the **etx** index over multiple disk drives is often preferable. If enough disk drives are available, it is also beneficial to spread the table data into multiple dbspaces or sbspaces with either expression-based or round-robin fragmentation.

Although it is often beneficial to fragment the **etx** index with expression-based fragmentation, round-robin fragmentation can result in worse performance than no fragmentation at all.

Indexing is fastest when you load the data into a nonindexed table and then create the **etx** index. However, inserting before indexing is not always possible. For applications that must insert large numbers of rows into a preindexed table, Informix suggests that multiple rows be collected in a staging or temporary table that is not indexed, and then inserted into the main table as a group.

For example, suppose the preindexed table is called **maintab**. Create a table called **stagingtab** that looks exactly like the **maintab** table, but with no **etx** index built on it. Insert a batch of documents into the **stagingtab** table; then insert these documents into the **maintab** table using an INSERT statement similar the following example:

```
BEGIN WORK;
INSERT INTO maintab ... SELECT * FROM stagingtab;
COMMIT WORK;
```

Encapsulating the INSERT statement in a transaction via the BEGIN WORK and COMMIT WORK statements can also improve performance because a table lock can be taken immediately on the **maintab** table and costly lock escalation avoided.

If you are inserting into a fragmented **etx** index, you can further optimize the insertion throughput if you can segment your incoming data to match the index fragmentation scheme, if you have multiple CPUs available, and if you can run concurrent data loading processes.

The recommended methodology is to create a temporary table and data loading process for each index fragment. Each data loading process would first insert the data into its own copy of a staging table and then into the target table. It would affect only one index fragment and thereby lock only that fragment. Additional loaders can load data into separate staging tables and index fragments and take advantage of the parallelism of the multiple CPU system.

The number of rows staged must be balanced against concurrency requirements, because the affected index fragment or fragments are locked against all other users until the insert is completed and the transaction is committed.

The size of document you want to index is limited by the amount of virtual memory on your machine. For example, if you have 32 MB of virtual memory on your machine, you can only index documents that are no larger than 32 MB. Informix also recommends that you not index documents larger than 268 MB.

Character Sets

When you create an **etx** index, you specify the character set the search engine uses by setting the `CHAR_SET` index parameter. The value you specify determines which characters in your text data are indexed and which are treated as white space.

Informix provides three built-in character sets: ASCII, ISO, and `OVERLAP_ISO`. You can also define your own character sets if the ones provided are inadequate for your particular text.

You cannot change the setting of the `CHAR_SET` parameter once you create the index unless you first drop and then re-create the index. If you do not specify a setting for `CHAR_SET`, the text engine uses the ASCII character set by default.

This appendix contains a complete description of the following three built-in character sets:

- ★ ASCII
- ★ ISO
- ★ `OVERLAP_ISO`

The last section of this appendix, “[ISO 8859-1](#),” contains a 16 x 16 mapping of all the characters in the ISO 8859-1 character set. You might want to use this mapping as a reference when you define your own character set.

ASCII

The text search engine uses this character set by default. The text engine uses this table to translate just the alphanumeric characters in the ASCII set (0 through 9, A through Z, and a through z). The lowercase characters are translated to uppercase, so this table results in 36 possible unique character values.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3	0	1	2	3	4	5	6	7	8	9						
4		A a	B b	C c	D d	E e	F f	G g	H h	I i	J j	K k	L l	M m	N n	O o
5	P p	Q q	R r	S s	T t	U u	V v	W w	X x	Y y	Z z					
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

ISO

The ISO character set is a superset of the ASCII character set displayed on the previous page. In addition to the standard ASCII characters, this character set contains the ISO Latin-1 characters. Each ISO character has its own value, except that lowercase characters are translated to uppercase. Two characters that have no uppercase equivalent are:

- German small sharp s (ß - 0xDF)
- Lowercase y diaeresis/umlaut (ÿ - 0xFF)

This table results in 68 possible unique characters. The ISO character set is shown on the following page.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3	0	1	2	3	4	5	6	7	8	9						
4		A a	B b	C c	D d	E e	F f	G g	H h	I i	J j	K k	L l	M m	N n	O o
5	P p	Q q	R r	S s	T t	U u	V v	W w	X x	Y y	Z z					
6																
7																
8																
9																
A																
B																
C	À à	Á á	Â â	Ã ã	Ä ä	Å å	Æ æ	Ç ç	È è	É é	Ê ê	Ë ë	Ì ì	Í í	Î î	Ï ï
D		Ñ ñ	Ò ò	Ó ó	Ô ô	Õ õ	Ö ö		Ø ø	Ù ù	Ú ú	Û û	Ü ü			ß
E																
F																ÿ

OVERLAP_ISO

The ISO Latin-1 character set is a superset of the ASCII character set shown on the previous page. In addition to the standard ASCII characters, this character set contains the ISO Latin-1 alphabetic characters, arranging the ISO characters into groups of similar-looking characters so that they will share the same numeric values. Uppercase and lowercase characters are grouped together.

Four exceptions that defy convenient categorization are assigned their own unique values and are listed in the following table.

Character Name	Lowercase Symbol	Lowercase Value	Uppercase Symbol	Uppercase Value
AE diphthong	æ	0xE6	Æ	0xC6
Icelandic Eth		0xF0		0xD0
Icelandic Thorn		0xFE		0xDE
German small sharp s	ß	0xDF		

The OVERLAP_ISO character set is shown on the following page.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3	0	1	2	3	4	5	6	7	8	9						
4		Aa ĂĂĂ ĂĂ àáâã äå	Bb Çç	Cc Çç	Dd d	Ee ÊÊÊÊ èéêë	Ff f	Gg g	Hh h	Ii ÌÍÎÏ ìíîï	Jj j	Kk k	Ll l	Mm m	Nn Ññ	Oo ÒÓÔ ÕÖØ òóô õöø
5	P p	Q q	R r	S s	T t	Uu ÛÚŮÛ ùúûü	V v	W w	X x	Yy ÿ	Z z					
6																
7																
8																
9																
A																
B																
C							Æ æ									
D																β
E																
F																

ISO 8859-1

The following table shows the ISO 8859-1 character set. You can reference this table when you create the file that contains the 16 x 16 matrix of characters you want indexed when you create your own character set. Use the routine **etx_CreateCharSet()** to make the user-defined character set known to the Excalibur Text Search DataBlade module.

For more information on using **etx_CreateCharSet()**, refer to [Chapter 5, "Routines."](#)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	NL	VT	NP	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[/]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
8																
9																
A	NBS	ı	ç	£	¤	¥		§	¨	©	ª	«	¬	-	®	¯
B	°				´		¶	·	,		º	»				¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D		Ñ	Ò	Ó	Ô	Õ	Ö		Ø	Ù	Ú	Û	Ü			ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F		ñ	ò	ó	ô	õ	ö		ø	ù	ú	û	ü			ÿ

Glossary

access method	A set of server functions that Informix Dynamic Server with Universal Data Option uses to access and manipulate a table (primary access method) or an index (secondary access method). etx is an example of a secondary access method.
approximate phrase search	A search in which the <i>search text</i> must contain a phrase identical to the <i>clue</i> , or one or more words of the clue. The order of the words in the clue is not important. For example, if the clue is <i>buy three dolls</i> , the search engine will return documents that contain the exact phrase as well as those that contain the phrases <i>three dolls</i> , <i>buy dolls</i> , or <i>dolls buy</i> .
BLOB	<p>A smart large object data type that stores any kind of binary data, including images. The database server performs no interpretation of the contents of a BLOB column.</p> <p>See also <i>smart large object</i>.</p>
Boolean search	A search that uses the Boolean expressions (logical operators) & (AND), (OR), or ! and ^ (NOT). Use the & Boolean operator when you want to search for documents that contain all the words in a keyword list, use when you want to search for documents that contain at least one word in the list, and use ! OR ^ when you want to search for documents that do not contain a specified word. The three Boolean operators can be combined to make more complicated expressions. This type of search is activated by setting the SEARCH_TYPE tuning parameter to BOOLEAN_SEARCH.
clue	The data that you are searching for, specified as the second argument to the etx_contains() operator.

filtering capability	A component of the <i>text search engine</i> that automatically filters out all proprietary formatting information from a formatted document and converts it into <i>ASCII</i> form.
exact phrase search	A search for text that matches your <i>clue</i> exactly. An exact phrase search is successful when the <i>text search engine</i> finds a phrase that contains all the words in the <i>clue</i> in the exact order that you specify.
fuzzy search	A search for text that matches your <i>clue</i> approximately instead of exactly. A fuzzy search takes into account <i>substitutions</i> , <i>transpositions</i> , and basic pattern matching. A search that returns a document that contains the word <code>editor</code> when searching for <code>edito r</code> is an example of a fuzzy search.
hit	The result (a row) of a text search.
hitlist	A list of <i>hits</i> (rows).
highlighting	The process of retrieving the location of every instance of a <i>clue</i> in the <i>search text</i> . The Excalibur Text Search DataBlade module returns highlighting information via pairs of beginning and ending offsets relative to <i>filtered</i> documents.
index parameter	A variable that you use to specify the characteristics of an etx index to accommodate the searches you plan to perform. An example of an index parameter is <code>WORD_SUPPORT='EXACT'</code> .
keyword	Any contiguous group of characters found in the <i>search text</i> or <i>clue</i> , delimited by nonindexable characters such as spaces or tabs.
keyword search	A search in which the words in the <i>clue</i> are treated as separate entities (keywords) instead of a single unit (phrase). When the <i>text search engine</i> performs a keyword search, it returns a row whenever it encounters one or more of the words in your <i>clue</i> .
pattern search	See <i>fuzzy search</i> .
phrase search	A search in which the words in the <i>clue</i> are treated as a single unit (phrase) instead of separate entities (keywords). The two types of phrase searches are <i>exact</i> and <i>approximate</i> .

proximity search	A search in which you specify the number of nonsearch words that can occur between two or more of the search words. You use a proximity search if, for example, you are searching for a phrase that contains the words <code>editor</code> and <code>multimedia</code> but do not want the two keywords separated by more than four nonsearch words. This type of search is activated by setting the tuning parameter <code>SEARCH_TYPE</code> equal to <code>PROX_SEARCH</code> .
rank	The order given to a <i>hitlist</i> based on the <i>score</i> of each of the returned rows.
root word	The word in a <i>synonym</i> list that has one or more synonyms defined for it. It is the leftmost word of a single row of the synonym operating system file. When synonym matching is activated, the keyword being searched for must be a root word for its synonym to be returned instead.
score	A value that the <i>text search engine</i> assigns to each of the returned rows of a <i>fuzzy search</i> that specifies the degree of similarity between your <i>clue</i> and each of the returned rows. Scores vary between 0 and 100, with 0 indicating no match and 100 indicating a perfect match. You access scoring information via the third parameter to the <code>etx_contains()</code> operator, a <i>statement local variable</i> (SLV). The data type of the SLV is <code>etx_ReturnType</code> , an Informix-defined row type that consists of three fields. The scoring information is contained in the score field.
search string	See <i>clue</i> .
search text	The data that is to be searched, stored in a column of a table. The column can be of type <code>IfxDocDesc</code> , <code>BLOB</code> , <code>CLOB</code> , <code>CHAR</code> , <code>VARCHAR</code> , or <code>LVARCHAR</code> . Abbreviation for <i>statement local variable</i> .
stopword	A keyword that you want excluded from your index or your search. Stopwords are typically common words such as <code>and</code> , <code>or</code> , <code>the</code> , and <code>to</code> , or any word that appears frequently in your document that you want to exclude.
substitution	A misspelling of a word, in which one letter has been substituted by another, incorrect one. Misspelling <code>searchk</code> for <code>search</code> is an example of a substitution.
synonym	One of two or more words or expressions that have the same or nearly the same meaning in some or all senses. The word <code>java</code> is a synonym of the word <code>coffee</code> .

text search engine	The component of the Excalibur Text Search DataBlade module that calls the Text Retrieval Library (TRL) of Excalibur Technologies to perform a search. The TRL is a library of C-language object modules designed to perform fast retrieval and automatic indexing of text data. The text search engine is dynamically linked into Informix Dynamic Server with Universal Data Option whenever a text search is performed or text data is indexed.
transposition	A misspelling of a word in which two adjacent letters switch positions. Misspelling <code>saerch</code> for <code>search</code> is an example of a transposition.
tuning parameter	A variable used to guide the way the <i>text search engine</i> conducts a search. Tuning parameters are passed to the text search engine via the second parameter of the Row() constructor of the etx_contains() operator. An example of a tuning parameter is <code>SEARCH_TYPE = WORD</code> .

Index

A

Access method, etx 1-7, 1-10, 2-3, 3-9
 Approximate phrase search 2-8, 5-11
 ASCII value for CHAR_SET index parameter 1-21, A-2
 ASC, option to CREATE INDEX 1-10

B

BLOB data type 1-7, 3-4, 7-11
 Boolean search 1-14, 2-5, 5-11
 BOOLEAN_SEARCH value for SEARCH_TYPE tuning parameter 2-5, 5-11
 BUFFERS, ONCONFIG parameter 7-13

C

Character sets 1-20, 3-10, 5-17, 5-26, 6-3, 6-5, A-1
 CHAR_SET index parameter 1-11, 1-20, 3-10, 5-17, 5-26, 6-3, 6-5, A-1
 CLOB data type 1-7, 3-4, 3-5, 4-14, 7-11
 Closing an index 5-15, 7-10
 CLUSTER, option to CREATE INDEX 1-10
 Conceptual overview 1-4
 CONSIDER_STOPWORDS tuning parameter 1-20, 3-13, 3-14, 5-9, 6-8

Creating stopword lists 1-19, 3-6, 5-21, 5-22, 6-7
 Creating synonym lists 1-16, 3-7, 5-23, 5-25
 Creating user-defined character sets 1-20, 3-8, 5-17, A-9

D

Data type
 advantages and disadvantages of each supported 7-10
 BLOB 1-7, 3-4, 7-11
 CLOB 1-7, 3-4, 3-5, 4-14
 etx_HiliteType 1-24, 3-14, 4-6 to 4-8, 5-31
 etx_ReturnType 1-15, 2-12, 4-4 to 4-7, 5-12
 IfxDocDesc 1-8, 4-9 to 4-13, 5-12, 7-12
 IfxMRData 1-8, 4-14, 7-12
 list of supported 1-7
 LLD_Locator 4-10
 LVARCHAR 4-14, 7-11
 Dbospace, usage with DataBlade module 3-4, 7-14
 Default synonym list 3-8, 5-9, 5-24
 Dependencies, of DataBlade module 7-7
 DESC, option to CREATE INDEX 1-10
 Determining character set usage 5-26
 Determining stopword list usage 5-28

DISTINCT, option to CREATE INDEX 1-10

Dropping stopword lists 3-7, 5-28

Dropping synonym lists 3-8, 5-30

Dropping user-defined character sets 1-21, 5-26

E

etx access method 1-7, 1-10, 2-3, 3-9

etx index

closing 5-15, 7-10

creating 1-10, 3-10 to 3-12

creating fragmented 3-12

estimating size of 7-7

internal structure of 7-9

sharing 5-15, 7-10

etx index parameter

CHAR_SET 1-11, 1-20, 3-10, 5-17, 5-26, 6-3, 6-5, A-1

definition of 1-11

INCLUDE_STOPWORDS 1-20, 3-10, 5-9, 6-3, 6-7, 7-8

PHRASE_SUPPORT 3-10, 3-11, 5-11, 6-3, 6-4, 7-7

specifying 1-11, 3-10, 6-3

STOPWORD_LIST 1-19, 3-10, 3-11, 5-9, 5-22, 6-3, 6-6, 7-8

WORD_SUPPORT 3-10, 3-11, 5-10, 6-3, 6-4

etx_blob_ops, operator class 1-8

etx_char_ops, operator class 1-9

etx_clob_ops, operator class 1-8

etx_CloseIndex() routine 5-15, 7-10

etx_contains() operator

description of 1-5, 1-6, 3-12, 5-5

return type of 5-11

syntax for 5-5

tuning parameters for. *See* Tuning parameter.

usage for 5-7, 5-12

used in a Boolean search 1-14

used in a keyword search 2-4

used in a pattern search 2-11, 2-15, 5-13

used in a phrase search with pattern matching 3-12

used in a proximity search 2-9, 2-10, 3-13

used in an approximate phrase search 2-8

used in an exact phrase search 2-6

used with a statement local variable (SLV) 2-12, 3-14, 5-13

used with stopwords

ignored 1-20, 3-13

used with synonym

matching 1-17, 5-13

etx_CreateCharSet() routine 1-21, 3-8, 5-17, A-9

etx_CreateStopWlst() routine 1-19, 3-6, 5-21, 5-22, 6-7

etx_CreateSynWlst() routine 1-16, 3-7, 5-23, 5-25

etx_doc_ops, operator class 1-9

etx_DropCharSet() routine 1-21, 5-26

etx_DropStopWlst() routine 3-7, 5-28

etx_DropSynWlst() routine 3-8, 5-30

etx_GetHilite() routine 1-23, 3-14, 4-5, 5-31

etx_HiliteType data type
description of 1-24, 4-6 to 4-8
usage of 3-14, 5-31

etx_lvarc_ops, operator class 1-8

etx_mrd_ops, operator class 1-9

etx_Release routine 5-40

etx_Release() routine 7-3

etx_ReturnType data type
definition of 4-4

usage of 1-15, 2-12, 5-12

etx_stopwords.txt, provided

stopword list 1-19, 3-6, 5-22, 6-7

etx_thesaurus, default synonym list 3-8, 5-9, 5-24

etx_varc_ops, operator class 1-9

Exact phrase search 2-6, 3-12, 3-13, 5-11

EXACT, value for
WORD_SUPPORT index
parameter 6-4

Example

of closing an etx index 5-16

of creating a fragmented etx index 3-12

of creating a table 1-7

of creating a user-defined

character set 1-22, 3-8, 5-20

of creating an etx index 1-10, 1-11, 3-10 to 3-12

of creating dbspaces 3-4

of creating sbspaces 3-4, 7-4, 7-5, 7-6

of creating stopword lists 1-19, 3-7, 5-22

of creating synonym lists 1-16, 3-7, 5-25

of determining character set usage 5-26

of determining stopword list usage 5-28

of dropping a user-defined character set 5-27

of highlighting 3-14, 5-33

of ignoring stopwords 1-20, 3-13

of inserting into a table 4-12

of inserting into an IfxDocdesc column 4-12

of inserting into an IfxMRData column 4-15

of limiting a search 2-15

of matching synonyms 1-17

of performing a Boolean search 1-14

of performing a keyword search 2-4, 2-5

of performing a pattern search 2-11, 2-15, 5-13

of performing a proximity search 2-9, 2-10, 3-13

of performing a synonym search 1-17

of performing an approximate phrase search 2-8

of performing an exact phrase search 2-6, 3-12

of ranking the results of a search 2-12, 4-5

of returning version information for the Excalibur Text Search DataBlade module 5-40

of returning version information
for the Text Descriptor
DataBlade module 5-41
of using statement local variables
(SLVs) 2-12, 4-4, 4-5

F

FileToBLOB() routine 7-5
FileToCLOB() routine 3-5, 4-13
FILLFACTOR, option to CREATE
INDEX 1-10
Filtering 1-9
format, field of IfxDocDesc 4-9
Fragmented index 3-12
Fuzzy search 2-10, 2-15, 2-17, 3-12,
5-13

G

Generic mapping for user-defined
character sets A-9

H

Highlighting
description of 1-20, 4-6, 5-31
example of 1-20, 3-14
hilitte_info, field of
etx_ReturnType 4-5
hitlist, definition of 1-14

I

IfxDocDesc data type
definition of 1-8, 4-9
fields of 4-9
usage of 4-12, 7-12
used as a clue 5-12
IfxMRData data type
definition of 1-8, 4-14
usage of 4-14, 7-12
INCLUDE_STOPWORDS index
parameter 1-20, 3-10, 5-9, 6-3,
6-7, 7-8
Index parameters. *See* etx index
parameter.

Index. *See* etx index.
ISO, value for CHAR_SET index
parameter 1-21, A-4

K

Keyword search 1-13, 2-3, 5-11

L

LLD_Locator data type
4-10
LOB Locator DataBlade
module 7-7
location, field of IfxDocDesc 4-10
LOCopy() routine 3-5
Logging of sbspaces 7-5
LVARCHAR data type 4-14, 7-11

M

MATCH_SYNONYM tuning
parameter 1-17, 5-9, 5-24
MAXIMUM value for
PHRASE_SUPPORT index
parameter 5-11, 6-5
MEDIUM value for
PHRASE_SUPPORT index
parameter 5-11, 6-4
Multirepresentational data type. *See*
IfxMRData data type.

N

NOAGE, ONCONFIG
parameter 7-14
NONE value for
PHRASE_SUPPORT index
parameter 6-4

O

oncheck utility 7-6
ONCONFIG parameter 7-4, 7-13,
7-14
onspaces utility 3-4, 7-4, 7-6

Operator classes 1-8, 1-10
OVERLAP_ISO value for
CHAR_SET index
parameter 1-21, A-6

P

params, field of IfxDocDesc 4-11
Pattern search 2-10, 2-15, 3-12, 5-13
Pattern search with phrase
matching 2-17
PATTERN value for
WORD_SUPPORT index
parameter 2-10, 5-10, 6-4
PATTERN_ALL tuning
parameter 1-18, 2-15, 2-17, 3-12,
5-10
PATTERN_BASIC tuning
parameter 2-17, 5-10
PATTERN_SUBS tuning
parameter 2-11, 2-16, 5-10
PATTERN_TRANS tuning
parameter 2-11, 2-16, 5-10
Performance 7-13, 7-14
Phrase search
approximate 2-8, 5-11
exact 2-6, 3-12, 3-13, 5-11
PHRASE_APPROX value for
SEARCH_TYPE tuning
parameter 2-8, 2-17, 5-11
PHRASE_EXACT value for
SEARCH_TYPE tuning
parameter 2-6, 2-17, 5-11
PHRASE_SUPPORT index
parameter 3-10, 3-11, 5-11, 6-3,
6-4, 7-7
Proximity search 2-8, 3-13, 5-11
PROX_SEARCH value for
SEARCH_TYPE tuning
parameter 2-9, 5-11

R

Ranking the results of a search 2-5,
2-12, 2-18, 5-13
RA_PAGES, ONCONFIG
parameter 7-13

RA_THRESHOLD, ONCONFIG
parameter 7-14
RESIDENT, ONCONFIG
parameter 7-14
Root word 1-16, 1-18, 5-24
Routine
etx_CloseIndex() 5-15, 7-10
etx_contains(). See etx_contains()
operator.
etx_CreateCharSet() 1-21, 3-8,
5-17, A-9
etx_CreateStopWlst() 1-19, 3-6,
5-21, 5-22, 6-7
etx_CreateSynWlst() 1-16, 3-7,
5-23, 5-25
etx_DropCharSet() 1-21, 5-26
etx_DropStopWlst() 3-7, 5-28
etx_DropSynWlst() 3-8, 5-30
etx_GetHilite() 1-23, 3-14, 4-5,
5-31
etx_Release() 5-40, 7-3
FileToCLOB() 3-5
LOCopy() 3-5
txt_Release() 5-41
Row() constructor 1-13, 4-12, 4-16

S

SBS spacename, ONCONFIG
parameter 7-4
Sb space, usage with DataBlade
module 1-10, 1-16, 1-19, 3-4,
5-21, 5-23, 7-4
Score 2-5, 2-12, 2-18, 5-13
score, field of etx_ReturnType 2-12,
4-4
Search
approximate phrase 2-8, 5-11
Boolean 1-14, 2-5, 5-11
exact phrase 2-6, 3-12, 3-13, 5-11
keyword 1-13, 2-3, 5-11
pattern 2-10, 2-15, 3-12, 5-13
phrase matching with
pattern 2-17
proximity 2-8, 3-13, 5-11
simple 1-12, 5-13

SEARCH_TYPE tuning
parameter 2-4, 2-6, 2-8, 2-17,
3-13, 5-11
Statement local variable (SLV) 1-15,
2-12, 4-4, 5-13
Stopword lists
creating 3-6, 5-21
discussion of 1-19
dropping 3-7, 5-28
STOPWORD_LIST index
parameter 1-19, 3-10, 3-11, 5-9,
5-22, 6-3, 6-6, 7-8
Substitution 2-10, 2-15, 2-16, 5-10
Synonym lists
creating 3-7, 5-23
default 3-8, 5-9, 5-24
discussion of 1-16, 5-23
dropping 3-8, 5-30

T

Text Descriptor DataBlade
module 4-3, 4-9, 4-14, 5-41, 7-3,
7-7
Transposition 2-10, 2-15, 2-16, 5-10
Tuning parameter
CONSIDER_STOPWORDS 1-20,
3-13, 3-14, 5-9, 6-8
description of 2-3
MATCH_SYNONYM 1-17, 5-9,
5-24
PATTERN_ALL 1-18, 2-15, 2-17,
3-12, 5-10
PATTERN_BASIC 2-17, 5-10
PATTERN_SUBS 2-11, 2-16, 5-10
PATTERN_TRANS 2-11, 2-16,
5-10
SEARCH_TYPE 2-4, 2-6, 2-8,
2-17, 3-13, 5-11
txt_Release routine 5-41

U

UNIQUE, option to CREATE
INDEX 1-10
User-defined character set
creating 1-20, 3-8, 5-17, A-9

creating operating system
file 5-18
dropping 5-26
generic mapping for A-9
using 1-20, 3-11, 6-6
Utility
oncheck 7-6
onspaces 3-4, 7-4, 7-6

V

vec_offset, field of
etx_HiliteType 3-14, 4-6
version, field of IfxDocDesc 4-10
Version, of the Excalibur Text
Search DataBlade module 5-40,
7-3
Version, of the Text Descriptor
DataBlade module 5-41
viewer_doc, field of
etx_HiliteType 3-14, 4-7

W

Word lists
stopword 1-19, 3-6, 5-21, 5-28
synonym 1-16, 3-7, 5-23, 5-30
WORD, value for SEARCH_TYPE
tuning parameter 2-4, 5-11
WORD_SUPPORT index
parameter 3-10, 3-11, 5-10, 6-3,
6-4