

Oracle8 ConText[®] Cartridge

Application Developer's Guide

Release 2.3

October 1997

Part No. A58164-01

Oracle8 ConText Cartridge Application Developer's Guide

Part No. A58164-01

Release 2.3

Copyright © 1996, 1997, Oracle Corporation. All rights reserved.

Primary Author: Colin McGregor

Contributing Author: D. Yitzik Brenman

Contributors: Peter Bell, Chandu Bhavsar, Anny Chan, Chung-Ho Chen, Yun Cheng, Roy Clarke, Paul Dixon, Garret Kaminaga, Kim Kepchar, Jackie Kud, Kavi Mahesh, Yasuhiro Matsuda, Mohammad Faisal, Josh Powers, Gerda Shank, Dipti Sonak, and Steve Yang.

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be licensee's responsibility to take all appropriate fail-safe, back up, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle disclaims liability for any damages caused by such use of the Programs.

This Program contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright patent and other intellectual property law. Reverse engineering of the software is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free.

If this Program is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Programs delivered subject to the DOD FAR Supplement are 'commercial computer software' and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are 'restricted computer software' and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle, SQL*Net, SQL*Plus, and ConText are registered trademarks of Oracle Corporation. Oracle8, Oracle Forms, Oracle Server, PL/SQL and Gist are trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Contents

Send Us Your Comments	xiii
Preface.....	xv
1 Query Concepts	
Text Queries	1-2
Case-Sensitivity	1-2
Section Searching.....	1-3
Theme Queries	1-4
Query Methods.....	1-5
Two-step Queries.....	1-5
One-step Queries	1-5
In-memory Queries	1-5
Counting Query Hits.....	1-6
Query Expressions	1-7
Stored Query Expressions	1-7
Query Expression Feedback.....	1-7
Scoring	1-8
Score Range	1-8
Scoring Algorithm for Text Queries	1-8
Result Tables.....	1-11
Document Presentation	1-12
Presenting Highlighted Documents	1-12
Presenting Linguistic Output (English Only).....	1-12

2 Query Methods

Selecting a Query Method	2-2
Using Two-Step Queries	2-3
Two-Step Query Example	2-3
Scoring.....	2-4
Hitlist Result Tables.....	2-4
SELECT from a Pre-defined View	2-6
Composite Textkey Queries	2-6
Structured Queries.....	2-7
Querying Columns in Remote Databases	2-8
Two-Step Queries in Parallel.....	2-9
Using One-Step Queries	2-10
One-Step Query Processing.....	2-10
One-Step Query Example	2-11
Multiple CONTAINS	2-11
Scoring.....	2-12
Restrictions.....	2-13
Multiple Policies.....	2-13
Composite Textkey Queries	2-13
Querying Columns in Remote Databases	2-13
Using In-Memory Queries	2-15
In-Memory Query Example	2-15
In-Memory Queries and Composite Textkeys	2-16
In-Memory Query Limitations.....	2-17
Querying Columns in Remote Databases	2-17
Counting Query Hits	2-18
Using COUNT_HITS Before the Query	2-18
Using COUNT_LAST After the Query	2-18

3 Understanding Query Expressions

About Query Expressions	3-2
Query Terms.....	3-2
Case-Sensitive Queries.....	3-3
Composite Word Queries for German.....	3-4
Base-Letter Queries.....	3-5

Query Expression Examples	3-5
Logical Operators	3-7
AND Operator	3-7
OR Operator	3-7
NOT Operator	3-8
Equivalence Operator	3-8
WITHIN Operator	3-10
Score-Changing Operators	3-12
Accumulate Operator.....	3-12
MINUS Operator	3-13
Near Operator	3-13
Weight Operator.....	3-14
Result-Set Operators	3-16
Threshold Operator.....	3-16
Max Operator	3-17
First/Next Operator	3-17
Combined First/Next and Max Queries	3-18
Expansion Operators	3-19
Stem Expansions.....	3-19
Soundex Expansions	3-20
Fuzzy Expansions.....	3-20
Penetration in Expansion Operators.....	3-21
Examining Query Expansions	3-22
Base-letter Support	3-22
Thesaurus Operators	3-23
Thesaurus Arguments.....	3-24
Synonym Operator	3-25
Preferred Term Operator.....	3-26
Related Term Operator	3-26
Narrower Term Operators	3-26
Broader Term Operators.....	3-27
Broader and Narrower Term Operator on Homographs	3-27
Top Term Operator	3-28
Thesaural Expansions and Case-Sensitivity	3-28
Base-letter Support for Thesaural Queries.....	3-29

Wildcard Characters	3-30
Grouping Characters	3-31
Stored Query Expressions	3-32
Using Stored Query Expressions.....	3-32
Session and System SQEs	3-33
Re-evaluation of Stored Query Expressions	3-33
Iterative Queries.....	3-34
SQE Tables	3-35
Using Operators in Stored Query Expressions.....	3-36
PL/SQL in Query Expressions	3-37
Example.....	3-37
Operator Precedence.....	3-38
Group 1.....	3-38
Group 2.....	3-39
Procedural Operators	3-39
Precedence Examples	3-39
Altering Precedence.....	3-40
Escaping Reserved Words and Characters	3-41
Example.....	3-41
Reserved Words	3-42
Querying Escape Characters	3-43
Querying with Stopwords.....	3-44
Stopwords by Themselves.....	3-44
Stopwords with Non-stopwords.....	3-44
Stopwords with Operators	3-44
Querying with Special Characters.....	3-46
Querying with Punctuation and Continuation Characters	3-47
Querying with Printjoins and Skipjoins	3-47
Querying with Numjoins and Numgroups.....	3-48
Querying with Startjoin and Endjoin Characters.....	3-49

4 Theme Queries

Understanding Theme Queries.....	4-2
The Knowledge Catalog	4-2
Theme Indexing	4-5

Theme Querying.....	4-7
Constructing Theme Queries	4-9
Using Operators.....	4-9
Phrasing Theme Queries	4-10
Refining Theme Queries	4-12
Restricting a Query.....	4-12
Expanding a Query	4-13
Theme Query Examples.....	4-15
Two-Step Query.....	4-15
One-Step Query	4-15

5 Query Expression Feedback

The Feedback Process	5-2
Understanding ConText Parse Trees	5-4
Operator Precedence.....	5-5
Query Expansions.....	5-6
Theme Query Normalization.....	5-8
Query Optimization	5-9
Stopword Rewrite.....	5-10
Understanding the Feedback Table.....	5-11
Table Structure	5-11
Example.....	5-14
Obtaining Query Expression Feedback.....	5-15
Creating the Feedback Table.....	5-15
Executing CTX_QUERY.FEEDBACK	5-15
Retrieving Data from Feedback Table	5-15
Constructing the Parse Tree	5-16

6 Document Presentation

Overview of Document Presentation.....	6-2
Using CTX_QUERY.HIGHLIGHT	6-3
Creating Highlighted Text.....	6-6
Allocating Result Tables	6-6
Issuing a Query	6-7
Calling CTX_QUERY.HIGHLIGHT.....	6-7

Presenting HIGHLIGHT Output.....	6-8
Release Highlight Result Tables	6-9
Document Presentation in Windows	6-10
Viewing Without the ConText Viewer Control.....	6-11

7 Linguistic Concepts

Overview of ConText Linguistics	7-2
Requirements.....	7-2
Application Program Interface (API)	7-3
Linguistic Personality.....	7-4
Services Queue	7-4
Creating Linguistic Output	7-5
Linguistic Core	7-6
Lexicon	7-6
Knowledge Catalog	7-6
Parsing Engine	7-7
Linguistic Output	7-8
List of Themes	7-8
Theme Summaries	7-9
Gists	7-10
Theme Indexes	7-11
Linguistic Settings	7-12

8 Using ConText Linguistics

Specifying Linguistic Settings	8-2
Generating Linguistic Output	8-3
Creating Output Tables.....	8-3
Generating Themes and Gists	8-4
Monitoring the Services Queue	8-7
Monitoring the Status of Requests	8-7
Removing Pending Requests	8-8
Clearing Requests with Errors	8-8
Specifying Completion and Error Procedures	8-10
Logging Parse Information	8-12
Combining Theme/Text Queries with Linguistic Output	8-13

Implementation.....	8-13
---------------------	------

9 SQL Functions

Query Functions.....	9-2
Prerequisites	9-2
CONTAINS	9-3
SCORE	9-5
SELECT Statement.....	9-6

10 PL/SQL Packages

Developing with ConText PL/SQL Packages	10-2
CTX_QUERY: Query and Highlighting.....	10-3
CLOSE_CON	10-4
CONTAINS.....	10-5
COUNT_HITS	10-8
COUNT_LAST	10-10
FEEDBACK.....	10-12
FETCH_HIT.....	10-14
GETTAB	10-16
HIGHLIGHT	10-18
OPEN_CON	10-22
PKDECODE.....	10-24
PKENCODE	10-25
PURGE_SQE.....	10-27
REFRESH_SQE.....	10-28
RELTAB	10-29
REMOVE_SQE.....	10-30
STORE_SQE.....	10-31
CTX_LING:Linguistics	10-33
CANCEL	10-34
GET_COMPLETION_CALLBACK.....	10-35
GET_ERROR_CALLBACK	10-36
GET_FULL_THEMES	10-37
GET_LOG_PARSE	10-38
GET_SETTINGS_LABEL.....	10-39

REQUEST_GIST	10-40
REQUEST_THEMES	10-43
SET_COMPLETION_CALLBACK	10-45
SET_ERROR_CALLBACK	10-46
SET_FULL_THEMES	10-47
SET_LOG_PARSE	10-48
SET_SETTINGS_LABEL	10-49
SUBMIT	10-51
CTX_SVC: Services Queue Administration	10-53
CANCEL	10-54
CANCEL_ALL	10-55
CANCEL_USER	10-56
CLEAR_ALL_ERRORS	10-57
CLEAR_ERROR	10-58
CLEAR_INDEX_ERRORS	10-59
CLEAR_LING_ERRORS	10-60
REQUEST_STATUS	10-61

A Result Tables

Hitlist Table Structure	A-2
Composite Textkey Hitlist Tables	A-2
Highlight Table Structures	A-3
HIGHTAB Highlight Table	A-3
MUTAB Highlight Table	A-3
ICFTAB Highlight Table	A-4
Display Table Structures	A-5
NOFILTAB Display Table	A-5
PLAINTAB Display Table	A-5
Linguistic Output Table Structures	A-6
Theme Table	A-6
Gist Table	A-7

B SQL*Plus Sample Code

Setting Up the ConText Sample Applications	B-2
Overview of CTXPLUS	B-3

Concepts.....	B-3
Using CTXPLUS.....	B-4
CTXPLUS Examples.....	B-5
Overview of CTXLING.....	B-7
Concepts.....	B-7
Using CTXLING	B-7
CTXLING Examples.....	B-9

C Stopword Transformations

Understanding Stopword Transformations.....	C-2
Word Transformations.....	C-3
AND Transformations	C-4
OR Transformations	C-4
Accumulate Transformations	C-5
MINUS Transformations	C-5
NOT Transformations.....	C-6
Equivalence Transformations	C-6
NEAR Transformations	C-7
Weight Transformations.....	C-7
Threshold Transformations.....	C-7
Max Transformations.....	C-7
First/Next Transformations.....	C-8
WITHIN Transformations	C-8

Index

Send Us Your Comments

Oracle8 ConText Cartridge Application Developer's Guide, Release 2.3

Part No. A58164-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- FAX - (650) 506-7200 Attn: ConText Documentation Manager
- postal service:
Oracle Corporation
Attn: ConText Documentation Manager
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A

If you would like a reply, please give your name, address, and telephone number below.

Preface

This manual explains the SQL*Plus and PL/SQL tools you use to issue text and theme queries with Oracle8 ConText Cartridge and how to enable users to view queried documents. It also explains how to generate document summaries using the linguistic capabilities of Oracle8 ConText Cartridge.

Audience

This document is intended for an application designer, application programmer, or systems analyst responsible for designing and developing text query applications using the facilities provided by ConText.

It is also applicable to the user responsible for managing text in a ConText application. Such users could also include DBAs or system administrators.

Prerequisites

This document assumes that you have experience with the Oracle relational database management system, SQL, SQL*Plus, and PL/SQL. See the documentation provided with your hardware and software for additional information.

If you are unfamiliar with the Oracle RDBMS and related tools, read Chapter 1, “A Technical Introduction to the Oracle Server”, in the *Oracle8 Server Concepts Manual*. The chapter is a comprehensive introduction to the concepts and terminology used throughout Oracle documentation.

Related Publications

For more information about ConText, see:

- *Oracle8 ConText Cartridge QuickStart*
- *Oracle8 ConText Cartridge Administrator's Guide.*
- *Oracle8 Error Messages.*
- *Oracle8 ConText Cartridge Workbench User's Guide.*

For more information about the Oracle8 server, see:

- *Oracle8 Server Concepts.*
- *Oracle8 Server Administrator's Guide.*
- *Oracle8 Server Utilities*
- *Oracle8 Server Tuning*
- *Oracle8 Server SQL Reference.*
- *Oracle8 Server Application Developer's Guide.*

For more information about PL/SQL, see:

- *PL/SQL User's Guide and Reference.*

How To Use This Manual

This manual is designed to be used by application developers to produce text retrieval applications for end users.

Specific tasks in the application design process depend on the type and complexity of the application being developed, but in general, the development process consists of six tasks:

- Analyzing user requirements
- Designing the application
- Developing a ConText application
- Estimating data storage requirements for the application
- Creating the ConText system environment with the database administrator
- Tuning the application's performance

This book only deals with developing a ConText application and tuning the application's performance. All the information necessary to develop and maintain ConText applications is covered in the following chapters.

The *Oracle8 ConText Cartridge Administrator's Guide* contains information about creating and maintaining the system environment to support ConText applications. The administrator's guide and the application developer's guide are designed to be used together.

How This Manual Is Organized

Chapter 1: Query Concepts

This chapter defines the concepts of querying text and themes with ConText.

Chapter 2: Query Methods

This chapter describes and compares the different query methods.

Chapter 3: Understanding Query Expressions

This chapter describes the various operators you can use to build query expressions.

Chapter 4: Theme Queries

This chapter describes how to issue theme queries.

Chapter 5: Query Expression Feedback

This chapter describes query expression feedback.

Chapter 6: Document Presentation

This chapter describes how to create highlighted output from a text or theme query and how to present highlighted documents to users.

Chapter 7: Linguistic Concepts

This chapter describes what is linguistic output and what are the requirements for producing this output.

Chapter 8: Using ConText Linguistics

This chapter describes how to create linguistic output, including managing the service queue and combining theme/text queries with linguistic output.

Chapter 9: SQL Functions

This reference chapter describes the SQL functions you can use with ConText.

Chapter 10: PL/SQL Packages

This reference chapter describes the procedures and functions included in the PL/SQL packages shipped with ConText.

Appendix A: Result Tables

This appendix describes the schema for the result tables used for issuing text and theme queries, highlighting text, and creating linguistic output.

Appendix B: SQL*Plus Sample Code

This appendix contains explanations of the demonstration applications distributed with ConText.

Appendix C: Stopword Transformations

This appendix lists all ConText stopwords transformations.

Type Conventions

This book adheres to the following type conventions:

Type	Meaning
Uppercase	Uppercase letters indicate Oracle commands, standard database objects and constants, and standard Oracle PL/SQL procedures.
Lowercase Italics	Italics indicate variable names, PL/SQL parameter names, table names, view names and the names of example PL/SQL procedures.
Monospace	Monospace type indicate example SQL*Plus commands and example PL/SQL code. Type in the command or code exactly as it appears.

Customer Support

You can reach Oracle Worldwide Customer Support 24 hours a day.

In the USA: **1.415.506.1500**

In Europe: + **44.344.860.160**

Please be prepared to supply the following information:

- your CSI number
This helps Oracle Corporation track problems for each customer.
- the release numbers of the Oracle Server and associated products
- the operating system name and version number
- details of error numbers and descriptions
Write down the exact errors.
- a description of the problem

- a description of the changes made to the system

Your Comments Are Welcome

Please use the Reader's Comment Form to convey your comments to us. You can also contact us at:

Documentation Manager,
ConText Group
Server Technologies
Oracle Corporation
500 Oracle Parkway
Redwood Shores, California 94065
Phone: 1.650.506.7000 FAX: 1.650.506.7360
E-mail: infotext@us.oracle.com

Query Concepts

This chapter explains the fundamental concepts that underlie ConText text and theme processing. The following topics are covered in this chapter:

- Text Queries
- Theme Queries
- Query Methods
- Query Expressions
- Scoring
- Result Tables
- Document Presentation

Text Queries

In ConText, a text query is a search for a word or phrase in a document set. A document set is usually stored in a base-table containing a text-column, where every row in the column contains a document. To issue a text query, the text column must be indexed.

See Also: For more information about creating text indexes for columns, see *Oracle8 ConText Cartridge Administrator's Guide*.

You can issue text queries (and theme queries) using one of the following query methods:

- one-step
- two-step
- in-memory

All three methods produce the same query results, returning a hitlist of the documents (rows in the base-table) that satisfy your query. Each returned row contains information such as the primary key and relevance score of a document that matches the query. You choose a method depending on the application.

In an application, you can present the hitlist to the user ordered by score and let the user select a document. You can then present the selected documents to the user with query terms highlighted or by summarizing the documents thematically using linguistic analysis.

Case-Sensitivity

By default, ConText creates text indexes without being sensitive to the case of tokens in the documents. Because of this, text queries are case-insensitive. That is, a query on *United* returns documents that contain *United* and *UNITED* and *united*.

However, you can make text queries case-sensitive by using a case-sensitive lexer when you or your ConText administrator indexes the document set. When you create a case-sensitive index, a query on *United* is different from *united*, which is different from *UNITED*.

See Also: For more information about issuing case-sensitive text queries, see Case-Sensitive Queries in Chapter 3.

For more information about creating case-sensitive text indexes for columns, see *Oracle8 ConText Cartridge Administrator's Guide*.

Section Searching

In addition to searching for words within documents, ConText enables you to narrow text searches down to pre-defined sections within documents. To do section searching, you or your ConText administrator must define sections by specifying what tags delimit the section.

For example in an HTML document set, you can define all appropriately tagged headings as a document section, and then use the WITHIN operator to query for a term within all headings across all documents.

See Also: For more information about how to issue section searches, see “WITHIN Operator” in Chapter 3.

For more information about defining sections, see the *Oracle8 Con-Text Cartridge Administrator's Guide*.

Theme Queries

In addition to querying English-language documents by words or phrases (text query), you can query these documents by theme, or by their main concepts.

Theme queries work similarly to text querying in that you must create an index (theme) for the documents before you can query. Theme queries differ from text queries in that you need not provide the word patterns for the search. ConText interprets your query conceptually according to its view of the world and returns an appropriate document hitlist based on theme, along with a measure of how relevant each document is to the query.

You can use the standard query methods to perform theme queries, namely one-step, two-step, and in-memory. In a theme query, you can use most of the operators you use in regular text queries.

See Also: For more information about theme queries, see Chapter 4, “Theme Queries”.

Query Methods

ConText supports three different methods for performing queries:

- two-step
- one-step
- in-memory

In addition, ConText provides a method for counting query hits without performing an actual query.

Two-step Queries

Two-step queries use a PL/SQL procedure in the first step to store the results in a specified result table.

The second step uses a SELECT statement to select the results from the result table. In addition, the hitlist table can be joined with the original table to return more detailed document information. In the two-step method, the physical hitlist table is available to the application program.

See Also: For more information about using two-step queries, see “Using Two-Step Queries” in Chapter 2.

One-step Queries

In a one-step query, you create a single SQL statement to search for relevant documents. ConText returns directly to you the rows and columns of the text table that satisfy the query.

ConText creates the hitlist using internal result tables. As a result, you do not have to create result tables before running a one-step query; however, the internal result tables are not available to the application program.

See Also: For more information about using one-step queries, see “Using One-Step Queries” in Chapter 2.

In-memory Queries

In-memory queries use a buffer and a CONTAINS cursor to the buffer to return query results, rather than the result tables used in two-step and one-step queries. As a result, in-memory queries are generally faster than two-step and one-step queries for shorter hitlists.

In an in-memory query, you open a cursor to the query buffer and run a query. ConText writes the results of the query to the buffer. You fetch the results, then close the cursor.

Results can be returned in order of their textkeys or sorted by score.

See Also: For more information about using in-memory queries, see “Using In-Memory Queries” in Chapter 2.

Counting Query Hits

In addition to fully executing two-step, one-step, and in-memory queries, you can count the number of hits in a two-step or in-memory query before or after you issue the query. The documents can be stored in a local or remote database. Counting query hits helps to audit queries to ensure large and unmanageable hitlists are not returned.

See Also: For more information about counting query hits, see “Counting Query Hits” in Chapter 2.

Query Expressions

Query expressions are made up of words and phrases (query terms) combined with operators and other special characters to produce search criteria. Operators specify the relative importance of the query terms, define relationships between those terms, control how the search is performed, and determine how much output is returned.

The most basic kind of query expression is single words or phrases that return documents with a score based on the number of occurrences of the words or phrases. More complex expressions allow the user to weight certain terms, search for words that sound like each other, and find all of the words based on a particular root.

See Also: For more information about query expressions, see Chapter 3, “Understanding Query Expressions”.

Stored Query Expressions

A stored query expression (SQE) is a named query expression that has been stored in database tables along with the results of the query.

You can combine queries by referencing an SQE within the query expression of another query. Using an SQE in a query results in faster execution of the query because the results are already stored in the database.

Stored query expressions can also be used to perform interactive queries, in which an initial query is refined using one or more additional queries.

See Also: For more information about using stored query expressions, see “Stored Query Expressions” in Chapter 3.

Query Expression Feedback

Query expression feedback is a feature that enables you to know how ConText parses a text or theme query expression *before* you execute the query. Knowing how ConText evaluates a text or theme query expression is useful for refining and debugging queries.

You can also design a query application so that it uses the feedback information to help users write better queries.

See Also: Chapter 5, “Query Expression Feedback”

Scoring

When you issue either a text query or theme query, ConText returns the hitlist of documents that satisfy your query. Each document has a score that indicates how relevant the document is to the query you entered; the higher the score, the more relevant the document. You can use scores to order the hitlist to show the most relevant documents first.

In two-step queries, ConText calculates the score when you issue CTX_QUERY.CONTAINS procedure and stores the score in a result table called the hitlist table.

In one-step queries, ConText calculates scores when you use the CONTAINS function. You obtain scores using the SCORE function.

In in-memory queries, ConText returns the score for a hit as an out parameter with the CTX_QUERY.FETCH_HIT function.

Score Range

The score of a document in the result set is an integer within the range 1 to 100 inclusive. The highest score for a given query is not necessarily 100; it can be any integer in the range $1 \leq n \leq 100$.

Scoring Algorithm for Text Queries

Note: This section discusses how ConText calculates score for text queries, which is different from the way it calculates score for theme queries.

For more information about scoring for theme queries, see “Theme Querying” in Chapter 4.

To calculate a relevance score for a returned document in a text query, ConText uses an inverse frequency algorithm. Inverse frequency scoring assumes that frequently occurring terms in a document set are "noise" terms, and so these terms are scored lower. For a document to score high, the query term must occur frequently in the document but infrequently in the document set as a whole.

The following table illustrates ConText’s inverse frequency scoring. The first column shows the number of documents in the document set, and the second column shows the number of terms in the document necessary to score 100.

This table assumes that only one document in the set contains the query term.

Number of Documents in Document Set	Frequency of Term in Document
1	34
5	20
10	17
50	13
100	12
500	10
1,000	9
10,000	7
100,000	5
1,000,000	4

The table illustrates that if only one document contained the query term and there were five documents in the set, the term would have to occur 20 times in the document to score 100. Whereas, if there were 1,000,000 documents in the set, the term would have to occur only 4 times in the document to score 100.

Example

You have 5000 documents dealing with chemistry in which the term *chemical* occurs at least once in every document. The term *chemical* thus occurs frequently in the document set.

You have a document that contains 5 occurrences of *chemical* and 5 occurrences of the term *hydrogen*. No other document contains the term *hydrogen*.

Because *chemical* occurs so frequently in the document set, its score for the document is lower with respect to *hydrogen*, which is infrequent in the document set as a whole. This is so even though both terms occur 5 times in the document.

Note: Even if the relatively infrequent term *hydrogen* occurred 4 times in the document, and *chemical* occurred 5 times in the document, the score for *hydrogen* might still be higher, because *chemical* occurs so frequently in the document set (at least 5000 times).

Inverse frequency scoring also means that adding documents that contain *hydrogen* lowers the score for that term in the document, and adding more documents that do not contain *hydrogen* raises the score.

DML and Scoring

Because the scoring algorithm is based on the number of documents in the document set, inserting, updating or deleting documents in the document set is likely change the score for any given term before and after the DML.

If DML is heavy, you or your ConText administrator must optimize the index. Perfect relevance ranking is obtained by executing a query right after optimizing the index.

If DML is light, ConText still gives fairly accurate relevance ranking.

In either case, you or your ConText administrator must synchronize the index with CTX_DML.SYNC whenever DML is performed on the index.

See Also: For more information about optimizing and synchronizing an index, see *Oracle8 ConText Cartridge Administrator's Guide*.

Result Tables

Result tables are storage areas used by ConText to store output from user queries. These tables are allocated by the application program or procedure and exist until they are released by the application.

Result tables store the following:

- output of a two-step query.
- query expression feedback information
- highlighting output for viewing query terms in documents.
- linguistic output.

Result tables are also used in one-step queries; however, the tables used in one-step queries are internal tables that are allocated by ConText and cannot be accessed from application program.

You can create result tables using the SQL command `CREATE` or using the `CTX_QUERY.GETTAB` function.

See Also: For more information about creating and using result tables, see “Hitlist Result Tables” in Chapter 2.

For more information about the structure of result tables, see Appendix A, “Result Tables”.

For more information about the feedback result table, see “Understanding the Feedback Table” in Chapter 5.

For more information about generating linguistic output, see “Generating Linguistic Output” in Chapter 8.

Document Presentation

When your application obtains the results of a query, it can let the user select a document from the hitlist and then present the following:

- the document with or without query terms highlighted (text queries)
- the document with or without paragraphs highlighted (theme queries)
- linguistic output of the document (English only)

Presenting Highlighted Documents

Context enables you to present documents to the user with query terms highlighted for text queries, or with relevant paragraphs highlighted for theme queries.

With PL/SQL, you create the viewable output by calling a highlighting procedure after you issue the query. This procedure outputs the highlighted information to a result table, which you use to present the document.

Context also has a OCX control that you can embed programmatically in Windows client-side applications. This control allows users to query documents and then view them in their native formats, such as Microsoft Word, with query terms or paragraphs highlighted.

See Also: For more information about presenting highlighted documents, see Chapter 6, “Document Presentation”.

Presenting Linguistic Output (English Only)

For English-language documents, ConText linguistics enables you to create different views of the contents of documents that allow the user to quickly review the essential content of documents.

Because these services are separate and distinct from text and theme indexing, you can incorporate linguistic analysis and functionality in a text application, independent of the text/theme indexing process.

ConText linguistics can generate the following forms of linguistic output for documents:

Output Type	Description
Themes	The main concepts of a document.
Gist	Paragraph or paragraphs in a document that best represent what the document is about as a whole.

Output Type	Description
Theme Summary	Paragraph or paragraphs in a document that best represent a given theme in the document.
Sentence-Level Gist	Sentence or sentences in a document that best represent the themes in the document as a whole.
Sentence-Level Theme Summary	Sentence or sentences in a document that best match a single theme in the document.

You obtain linguistic output by submitting a linguistic request using the CTX_LING PL/SQL package.

See Also: For more information about ConText linguistics, see Chapter 7, “Linguistic Concepts”.

Query Methods

This chapter describes the different query methods you can use in your ConText application. You can use these methods with text queries and theme queries. The following topics are covered:

- Selecting a Query Method
- Using Two-Step Queries
- Using One-Step Queries
- Using In-Memory Queries
- Counting Query Hits

Selecting a Query Method

Each of the query methods (two-step, one-step, and in-memory) provide advantages and disadvantages that you must consider when developing an application. The following table briefly describes each method and illustrates the various advantages and disadvantages to using each:

Query Method	Use	Advantage	Disadvantage
One-step	Used in SQL*Plus. Best suited for interactive queries.	<ul style="list-style-type: none">■ No pre-allocation of result tables■ Uses standard SQL statements■ Uses table and column names■ Query results returned in a single step■ Can retrieve all hits at once	<ul style="list-style-type: none">■ Generally slower than two-step or in-memory queries■ No access to result tables
Two-step	Two-step queries are best suited for PL/SQL-based applications that require all the results to a query.	<ul style="list-style-type: none">■ Result tables can be manipulated■ Generally faster than one-step queries, especially for mixed queries■ Can retrieve all hits at once■ Structured data can be queried as part of the CONTAINS (first step)	<ul style="list-style-type: none">■ Requires pre-allocation of result tables■ Uses policy names■ Requires two steps to complete■ Requires join to base text table to return document details
In-memory	In-memory queries are best suited for PL/SQL-based applications that might generate large hitlists, but where only a small portion of the hits are required at a time, such as World Wide Web applications.	<ul style="list-style-type: none">■ No result tables■ Faster response time than two-step, since you need not retrieve all hits in the hitlist.■ Large hitlists generally faster than one-step and two-step queries■ Can specify the number of hits returned	<ul style="list-style-type: none">■ Uses policy names■ Cannot retrieve all hits at once■ With small hitlists, performance improvement over two-step is negligible■ Requires three steps, including a loop, to complete■ Queries for structured data must be performed separately and joined with in-memory results■ Max and first/next operators are not supported

Using Two-Step Queries

To perform a two-step query, do the following:

1. Execute CTX_QUERY.CONTAINS. The procedure selects all documents that match the specified search criteria (query expression) and generates a score for each document.

The document textkeys and scores are stored in the specified result table.

Note: You must create the result table *before* you execute the CONTAINS procedure.

2. Use a SELECT statement on the result table (and the base text table, if desired) to return the specified columns as a hitlist for the rows (documents) that satisfy the query expression.

Two-Step Query Example

The following example shows a simple two-step query. The query uses a policy named ARTICLES_POL to search the text column in a table named TEXTTAB for any articles that contain the word *petroleum*. The CONTAINS procedure populates the CTX_TEMP results table with the document primary keys that satisfy the query.

The select statement then joins the results in CTX_TEMP with TEXTAB to create a list of document titles ordered by score.

Note that before the two-step query example is executed, the result table, CTX_TEMP, is created:

```
create table CTX_TEMP(
    textkey varchar2(64),
    score number,
    conid number);

execute ctx_query.contains('ARTICLE_POLICY','petroleum','CTX_TEMP')

SELECT SCORE, title
FROM CTX_TEMP, TEXTTAB
WHERE texttab.PK=ctx_temp.textkey
ORDER BY SCORE DESC;
```

In this example, the articles with the highest scores appear first in the hitlist because the results are sorted by score in descending order.

Scoring

In a two-step query, the score results generated by the CONTAINS procedure are physically stored in a result table that has been allocated (either by the application developer or dynamically within the application).

If you want to include scores in the hitlist returned by a two-step query, select the from the result table in the second step of the query.

Note: The way in which ConText calculates a relevance score for text queries is different than the way it calculates scores for theme queries.

To learn more about how ConText calculates relevance score for text queries, see “Scoring” in Chapter 1.

To learn more about how ConText calculates relevance scores for theme queries, see “Understanding Theme Queries” in Chapter 4.

Hitlist Result Tables

In two-step queries, ConText uses result tables called hitlist tables to store intermediate results. Intermediate results can be merged into the standard SQL query through a join operation or a sub-query operation. The result tables must be created before the query is performed. A hitlist table can be created manually or allocated through the CTX_QUERY.GETTAB procedure.

Hitlist tables can be named anything; however, they must have the following structure:

Column Name	Column Datatype	Purpose
TEXTKEY	VARCHAR2(64)	Stores textkeys of the rows satisfying the query
SCORE	NUMBER	Stores the score for each row (document)
CONID	NUMBER	Stores the CONTAINS ID when multiple CONTAINS procedures utilize the same result table

See Also: For more information about the structure of the hitlist result tables, see “Hitlist Table Structure” in Appendix A.

Sharing a Hitlist Result Table

For applications that support multiple concurrent users, ConText allows for sharing a single result table among all the users rather than allocating a separate table for each user.

You control sharing of result tables with the *sharelevel* and the *query_id* parameters of the CTX_QUERY.CONTAINS procedure. If the result table is shared, the CONTAINS procedure must specify that *sharelevel* is equal to one and include a unique *query_id* so that each result can be distinguished from others in the result table.

When *sharelevel* is equal to 0:

- the hitlist result table is intended for exclusive use
- ConText truncates the hitlist result table at the start of each query
- after the query is completed, CONID values are NULL

When *sharelevel* is equal to 1 then:

- the hitlist result table is intended for shared use
- specify a unique number for *query_id* in the CONTAINS procedure to identify which entries belong to you in the hitlist result table. This number will be assigned to the CONID for each row in the result table generated by the query.
- before the query is run, you must delete existing rows in the result table with the same *query_id* as that specified in the CONTAINS procedure
- after the query is complete, the CONID column for all rows returned by the query contains the *query_id* specified in the CONTAINS procedure
- select your rows by specifying the appropriate CONID in the WHERE clause of the SELECT statement

Attention: ConText does not verify that these rules are observed. You must control multiple concurrent usage by passing a different *query_id* to the requestor if the result table is shared.

Composite Textkey Result Tables

When you execute a two-step query on a table with a composite textkey, the number of textkey columns in the result table must match the composite keys count in the document table. For example, if you want to execute a query on a document table that has a two-column textkey, create a result table with the following schema: TEXTKEY, TEXTKEY2, SCORE, CONID.

The following SQL*Plus examples show two different ways in which to create a result table with a two-column composite textkey:

```
/* create composite textkey result table manually */
create table ctx_temp(
    textkey varchar2(64),
    textkey2 varchar2(64),
    score number,
    conid number);

/* allocate composite textkey result table with CTX_QUERY.GETTAB() */
exec ctx_query.gettab(ctx_query.HITTAB, :hit_tab, 2)
```

See Also: For more information on the structure of composite textkey result tables, see “Composite Textkey Hitlist Tables” in Appendix A.

SELECT from a Pre-defined View

There is an alternative to the second step of a two-step query. Rather than joining the result table and text table in a SELECT statement, you can create a view to perform the join. Then use a SELECT statement to select the appropriate rows from that view. Use this approach when the development tool does not allow tables to be joined in a SELECT statement (e.g. Oracle Forms).

For example:

```
CREATE VIEW SURVEY AS SELECT * FROM TEXTTAB, CTX_TEMP
WHERE PK = TEXTKEY;

SELECT SCORE, AUTHOR FROM SURVEY
ORDER BY SCORE DESC;
```

In this example:

- The CREATE VIEW statement joins the table of articles (TEXTTAB) and the result table (CTX_TEMP). The PK column holds the primary key of the documents.
- The SELECT statement retrieves the scores from the view.

Composite Textkey Queries

To execute a two-step query on a table with a composite textkey, you first specify the multiple textkey columns when you create the policy for the text column.

See Also: For more information about creating policies for composite textkey tables, see *Oracle8 ConText Cartridge Administrator's Guide*.

In addition, before the two-step query, create a result table in which the number of TEXTKEY columns match the number of columns in the composite textkey in the document table. You can create the result table manually or using the CTX_QUERY.GETTAB procedure.

See Also: For more information on the structure of composite textkey result tables, see “Composite Textkey Hitlist Tables” in Appendix A.

For example, to create a result table manually with a composite textkey consisting of two columns, issue the following SQL statement:

```
create table CTX_TEMP2(
    textkey varchar2(64),
    textkey2 varchar2(64),
    score number,
    conid number);
```

In the two-step query, use the AND operator in the WHERE condition when you join the result and text tables. For example:

```
exec ctx_query.contains('ARTICLE2_POLICY','petroleum','CTX_TEMP2')
SELECT SCORE, title
FROM CTX_TEMP2, TEXTTAB2
WHERE texttab2.PK=ctx_temp2.textkey AND
      texttab2.PK2=ctx_temp2.textkey2
ORDER BY SCORE DESC;
```

Structured Queries

A structured query is a query based on a text column and a structured data column. The structured data column is usually in the same table as the text column. For example, you might use a structured query to retrieve documents on a certain subject that were written after a certain date, where the document content is in a text column and date information is in a structured data column.

The CTX_QUERY.CONTAINS procedure provides an additional parameter, *struct_query*, for specifying the WHERE condition in a structured query. For example, to select all news articles that contain the word *Oracle* that were written on or after October 1st, 1996, you might use:

```
exec ctx_query.contains('news_text','Oracle','res_tab',  
struct_query => 'issue_date >= (''1-OCT-1996'')')
```

Note: Because the *struct_query* parameter expects a WHERE condition, you can specify a subquery. This is useful when the structured data column is in another table.

Executing a structured query with the *struct_query* parameter improves performance over processing a query on a text column and then refining the hitlist by applying a where condition against a structured column. This is especially so when the selectivity of the WHERE condition is high, because when you use the structured query parameter, the ConText server executes the entire query without first writing out a potentially large hitlist to be refined later by the Oracle server.

Note: If the user who includes a structured query in a two-step query is not the owner of the table containing the structured and text columns, the user must have SELECT privilege with GRANT OPTION on the table. In addition, if the object being queried is a view, the user must have SELECT privilege with GRANT OPTION on the base table for the view. SELECT privilege with GRANT OPTION can be granted to a user using the GRANT command in SQL.

For more information, see *Oracle8 Server SQL Reference*.

Querying Columns in Remote Databases

If a database link has been created for a remote database, two-step queries support querying text columns in the remote database.

Note: Database links are created using the CREATE DATABASE LINK command in SQL.

For more information about creating database links, see *Oracle8 Server SQL Reference*.

To perform a two-step query for a text column in a remote database, specify the database link for the remote database in the CONTAINS procedure as part of the policy for the column in the remote database.

In addition, the result table specified in CONTAINS must exist in the remote database, and you, the user performing the query, must have the appropriate privileges on the result table.

For example:

```
exec ctx_query.contains('MY_POL@DB1', 'petroleum', 'CTX_TEMP')
```

In this example, MY_POL exists in a remote database identified by the database link DB1. The CTX_TEMP result table exists in the same remote database.

See Also: For more information about remote queries and distributed databases, see *Oracle8 Server Concepts*.

Two-Step Queries in Parallel

The CONTAINS procedure provides an argument for processing two-step queries in parallel. Processing queries in parallel helps balance the load between ConText servers and might improve query performance.

When the CONTAINS procedure is called in a two-step query, the PARALLEL argument can be used to specify the number of ConText servers, up to the total number of ConText servers running with the Query personality, that are used to process two-step queries and write the results to the result table.

For example:

```
exec ctx_query.contains('ARTICLE_POLICY', 'petroleum', 'CTX_TEMP', parallel=>2)
```

In this example, the text column in the ARTICLE_POLICY policy is queried for documents that contain the term *petroleum*. The query is processed in parallel by any two available ConText servers with the Query personality and the results are written to CTX_TEMP.

Using One-Step Queries

The one-step query uses the CONTAINS and SCORE functions in a SQL statement to execute a user's request for documents. Rows and columns containing the text and structured data for relevant documents are returned to the application program as a record set like any other query in SQL.

Note: Before one-step queries can be executed, the database in which the text resides must be text enabled by setting the ConText initialization parameter TEXT_ENABLE = TRUE. This can be done by either setting it in the `initsid.ora` system initialization file, or by using the ALTER SESSION command.

For more information about initialization parameters and the `initsid.ora` file, see *Oracle8 Server Administrator's Guide*.

For more information about using the ALTER SESSION command, see *Oracle8 Server SQL Reference*.

One-Step Query Processing

After a user has submitted a one-step query, ConText performs the following tasks to return the results to the user:

1. The query is placed on the text queue (query pipe). The Oracle server intercepts the query and passes the text portion (CONTAINS) to ConText.
2. A ConText server with the Query personality picks up the text portion of the query, processes the CONTAINS function(s) and stores the results in an internal table created automatically for the user who submitted the query. This table (and the corresponding intermediate results) are not available to the application.
3. The ConText server rewrites the query as a standard SQL statement and passes it back to Oracle.
4. The rewritten query is executed by an Oracle server and the results are returned to the user.
5. The internal result table is truncated.

One-Step Query Example

The following SELECT statement shows a simple one-step query. This query searches a text table called TEXTTAB for any articles that contain the word *petroleum*.

```
SELECT *  
FROM texttab  
WHERE CONTAINS (text, 'petroleum') > 0;
```

Because ConText functions execute within normal SQL statements, all of the capabilities for selecting and querying normal structured data fields, as well as text, are available. For instance, in the example, if the text table had a column listing the date the article was published, the user could select articles based on that date as well as the content of the text column.

Note: The asterisk wildcard character (***) specifies that the record set returned by the query includes all the columns of the text table for the selected documents, as well as the scores generated for each document. If a query has more than one CONTAINS function, the asterisk wildcard does not return scores for the multiple CONTAINS and the SCORE function must be called explicitly. See “Scoring” in this chapter for an example.

Multiple CONTAINS

One-step queries support calling more than one CONTAINS functions in the WHERE clause of a SELEC statement. Multiple CONTAINS can be used in a one-step query to perform queries on multiple text columns located either in the same table or in separate tables.

If multiple ConText servers with the Query personality are running and a one-step query with multiple CONTAINS is executed, the query is processed in parallel. Each CONTAINS function is evaluated by one of the available ConText servers and the results from the servers are combined before they are returned to the user.

Suggestion: If your application makes use of multiple CONTAINS in one-step queries, ensure that multiple ConText servers with the Query personality are running to optimize query performance. The number of ConText servers should be at least equal to the number of CONTAINS you support in one-step queries for the application.

Scoring

In a one-step query, the document scores are generated by the CONTAINS function and returned by the SCORE function.

Each CONTAINS function in a query produces a separate score. When there are multiple CONTAINS functions, each CONTAINS function must have a label (a number) so the SCORE value can be identified in other clauses of the SELECT statement.

The SCORE function may be used in a SELECT list, an ORDER BY clause or a GROUP BY clause.

For example:

```
SELECT SCORE (10), SCORE(20), title FROM DOCUMENTS
WHERE CONTAINS (TEXT, 'holmes', 10) > 0
      OR CONTAINS (TEXT, 'moriarty', 20) > 0
      OR CONTAINS (TEXT, 'baker street', 30) > 0
ORDER BY SCORE(10)
GROUP BY SCORE(30)
```

Note: The way in which ConText calculates a relevance score for text queries is different than the way it calculates scores for theme queries.

To learn more about how ConText calculates relevance score for text queries, see “Scoring” in Chapter 1.

To learn more about how ConText calculates relevance scores for theme queries, see “Understanding Theme Queries” in Chapter 4.

Restrictions

The CONTAINS function can only appear in the WHERE clause of a SELECT statement.

You cannot issue the CONTAINS function in the WHERE clause of an UPDATE, INSERT or DELETE statement.

Multiple Policies

For a text column that has more than one policy associated with it, you must specify which policy to use in the CONTAINS clause using the *pol_hint* parameter.

You might create two policies for a column when you want to perform both theme and text queries on the column, or in any application where you build two separate indexes for a text column.

See Also: For more information on issuing one-step queries with multiple policies see “Theme Query Examples” in Chapter 4.

To learn more about using the *pol_hint* parameter, see the specification for the SELECT Statement in Chapter 9.

Composite Textkey Queries

You can perform one-step queries on text tables with composite textkeys. The syntax for the query is the same as the syntax for a query on a table with a single-column textkey.

Querying Columns in Remote Databases

If a database link has been created for a remote database, one-step queries support querying text columns in the remote database.

To perform a one-step query for a text column in a remote database, the database link for the remote database is specified as part of the table name in the SELECT clause.

For example:

```
SELECT *  
FROM texttab@db1  
WHERE CONTAINS (text, 'petroleum') > 0;
```

In this example, *texttab* exists in a remote database identified by the database link DB1

Note: One-step queries do not support querying LONG and LONG RAW columns in remote database tables.

For more information about creating database links, see *Oracle8 Server SQL Reference*.

For more information about remote queries and distributed databases, see *Oracle8 Server Concepts*.

Using In-Memory Queries

In-memory queries use a buffer and a cursor to return query results. Returning query results to a buffer in memory improves performance over writing and reading query results to and from database result tables, which is typical of one- and two-step queries.

To perform an in-memory query, do the following:

1. Call the `CTX_QUERY.OPEN_CON` function. `OPEN_CON` performs the following operations:
 - opens a cursor to the query buffer
 - queries a text column using the specified policy and query expression
 - stores in the query buffer the document textkeys and scores for all the documents that meet the search criteria. Hits are stored in order that they are returned or ranked by score, depending on the argument specified for `OPEN_CON`

In addition, you can specify that `OPEN_CON` return additional columns (up to five) for the selected documents from the text table.

2. Call the `CTX_QUERY.FETCH_HIT` function for each textkey in the buffer to fetch the desired query results, one hit at a time, until the desired number of hits has been returned or no hits remain in the buffer.
3. Call the `CTX_QUERY.CLOSE_CON` procedure to release the cursor opened by `OPEN_CON`.

In-Memory Query Example

The following example shows a simple in-memory query. This query uses a policy named `ARTICLES_POL` to search the text column in a table named `TEXTTAB` for any articles that contain the word *petroleum*.

```
declare
  score  char(5);
  pk     char(5);
  curid  number;
  title  char(256);

begin
  dbms_output.enable(100000);
  curid := ctx_query.open_con(
    policy_name => 'ARTICLES_POL',
    text_query  => 'petroleum',
    score_sorted => true,
    other_cols  => 'title');
  while (ctx_query.fetch_hit(curid, pk, score, title)>0)
  loop
    dbms_output.put_line(score||pk||substr(title,1,50));
  end loop;
  ctx_query.close_con(curid);
end;
```

In this example, the TITLE column from the table is also returned by OPEN_CON, so a variable must be declared for TITLE.

DBMS_OUTPUT.ENABLE sets the buffer size to the maximum of 100000 bytes (1 Mb) to ensure that the buffer is large enough to hold the results of the query.

The SCORE_SORTED argument in OPEN_CON is set to *true* which causes OPEN_CON to store the hits in the query buffer in descending order by score.

FETCH_HIT is called in a loop to fetch SCORE, PK, and TITLE for each hit until a value less than zero is returned, indicating that the buffer is empty.

DBMS_OUTPUT.PUT_LINE prints the results to the standard output.

See Also: For more information about the DBMS_OUTPUT PL/SQL package, see *Oracle8 Server Application Developer's Guide*.

In-Memory Queries and Composite Textkeys

You can perform in-memory queries on text tables that have multiple column textkeys. When you use CTX_QUERY.FETCH_HIT to retrieve each hit from the buffer, the PK argument is returned as an encoded string. To access an individual textkey, you must use CTX_QUERY.PKDECODE.

In-Memory Query Limitations

In-memory queries have the following limitation:

Max and First/Next Operators

You *cannot* use the max and first/next operators with in-memory queries.

Querying Columns in Remote Databases

If a database link has been created for a remote database, in-memory queries support querying text columns in the remote database.

Note: Database links are created using the CREATE DATABASE LINK command in SQL.

For more information about creating database links, see *Oracle8 Server SQL Reference*.

To perform an in-memory query for a text column in a remote database, the database link for the remote database is specified in the CTX_QUERY.OPEN_CON procedure as part of the policy for the column in the remote database.

In addition, the result table specified in CTX_QUERY.CONTAINS must exist in the remote database and the user performing the query must have the appropriate privileges on the result table.

See Also: For more information about remote queries and distributed databases, see *Oracle8 Server Concepts*.

Counting Query Hits

In addition to two-step, one-step, and in-memory queries, you can count the number of hits in a two-step or in-memory query. The documents can be stored in a local or remote database. Counting query hits helps to audit queries to ensure large and unmanageable hitlists are not returned.

You can count the number of hits before or after you issue the query using one of the following functions:

- `CTX_QUERY.COUNT_HITS`
- `CTX_QUERY.COUNT_LAST`

Using `COUNT_HITS` Before the Query

Before you issue a two-step or in-memory query, you can use the `CTX_QUERY.COUNT_HITS` function to return the number of hits for the query without generating scores for the hits or returning the textkeys for the documents.

`COUNT_HITS` can be called in two modes, estimate and exact. The results in estimate mode may be inaccurate; however, the results are generally returned faster than in exact mode

See Also: `CTX_QUERY.COUNT_HITS` in Chapter 10.

Using `COUNT_LAST` After the Query

You can use the `CTX_QUERY.COUNT_LAST` function to obtain the number of hits in a two-step query and in-memory query *after* issuing `CONTAINS` or `OPEN_CON`.

`COUNT_LAST` returns the number of hits obtained from the last call to `CTX_QUERY.CONTAINS` or `CTX_QUERY.OPEN_CON`.

For two-step queries, the time it takes to issue the query with `CONTAINS` and then to call `COUNT_LAST` is not as fast as calling `COUNT_HITS` before the query. However, in the case where you need to process all hits in a two-step query, issuing the query with `CONTAINS` and then calling `COUNT_LAST` is more efficient than calling `COUNT_HITS` and then calling `CONTAINS`.

With in-memory queries, issuing `OPEN_CON` and then calling `COUNT_LAST` is always a more efficient way to obtain an estimate of the query hits over calling `COUNT_HITS` and then calling `OPEN_CON`, since `COUNT_LAST` returns a number faster than `COUNT_HITS`.

See Also: `CTX_QUERY.COUNT_LAST` in Chapter 10.

Understanding Query Expressions

This chapter explains how to use ConText to create query expressions to find relevant text in documents. The topics covered in this chapter are:

- About Query Expressions
- Logical Operators
- WITHIN Operator
- Score-Changing Operators
- Result-Set Operators
- Expansion Operators
- Thesaurus Operators
- Wildcard Characters
- Grouping Characters
- Stored Query Expressions
- PL/SQL in Query Expressions
- Operator Precedence
- Escaping Reserved Words and Characters
- Querying with Stopwords
- Querying with Special Characters

About Query Expressions

A query expression defines the search criteria for retrieving documents using ConText. A query expression consists of query terms (words and phrases) and other components such as operators and special characters which allow users to specify exactly which documents are retrieved by ConText.

A query expression can also call stored query expressions (SQEs) to return stored query results or call PL/SQL functions to return values used in the query.

When a query is executed using any of the methods supported by ConText, one of the arguments included in the query is a query expression. ConText then returns a list of all the documents that satisfy the search criteria, as well as scores that measure the relevance of the document to the search criteria.

Query Terms

Query terms can consist of words and phrases. Query terms can also contain stop-words.

Words and Phrases

The words in a query expression are the individual tokens on which the query expression operators perform an action. If multiple words are contained in a query expression, separated only by blank spaces (no operators), the string of words is considered a phrase and the entire string is searched for during a query.

Stopwords

Stopwords are common words, such as *and*, *the*, *of*, and *to*, that are not considered significant query terms by themselves because they occur so often in text. However, stopwords can provide useful search information when combined with more significant terms.

For example, a query for documents containing the phrase *peanut butter and jelly* returns different results than a query for documents containing the terms *peanut butter* and *jelly*.

When you define a policy for a column, ConText lets you identify a list of stopwords. When stopwords are encountered in the documents in the column, they are not included as indexed terms in the text index; however, they are recorded.

As a result, stopwords cannot be searched for explicitly in text queries, but can be included as part of a phrase in a query expression.

See Also: For more information about querying with stopwords, see “Querying with Stopwords” in this chapter.

Stoplists can be created in any language supported by ConText. ConText provides a default stoplist in English.

Note: Stopwords do not have an affect on the theme indexes generated by ConText for your English-language documents.

Query Expression Components

In addition to query terms, a query expression may contain any or all of the following components:

Component	Purpose
Operators	Define the relationships between the terms in a query expression and specify the output returned by the query. The different types of operators are: logical, ranking, result set, proximity, expansion, and thesaurus.
Wildcard Characters	Expand query terms using pattern matching
Grouping Characters	Group terms and operators in a query expression
Stored Query Expressions (SQEs)	Return the results of a query that has been executed and the results stored in an SQE table
PL/SQL Functions	Execute a function and use the results in a query expression

Case-Sensitive Queries

With text queries, you can issue case-sensitive and case-insensitive queries. The ability to query in a case-sensitive way depends on the lexer preference used to index the document set.

By default, ConText uses a lexer preference that is not case-sensitive when indexing documents. Therefore, with a policy containing the default lexer preference, queries are not case-sensitive. When queries are not case-sensitive, a query on *United* returns the same hits as a query on *united*.

To issue case-sensitive text queries, you or your ConText administrator must first index your document set using a policy with a case-sensitive lexer preference. Using the same policy, you can issue case-sensitive queries. With case-sensitive queries, a query on *United* is different from a query on *united*.

Case-sensitive querying helps to identify words that have different meaning when capitalized. For example, to query on the proper noun *Church* (as someone's name) without getting the hits for the common noun *church*, you issue *Church* as your query. ConText returns all appearances of *Church*.

Note: Because a case-sensitive query on a term such as *Church* returns all appearances of *Church*, the hitlist includes occurrences of *Church* at the beginning of a sentence, whether it is the common or proper noun.

Stopwords and Case-Sensitivity

When you have case-sensitivity enabled, searches on stopwords are also case-sensitive. Thus when you issue a case-sensitive query on a phrase containing stopwords and non-stopwords, ConText searches for the phrase containing the stopwords with the specified case.

For example, assuming the word *on* is a stopword and case-sensitivity is enabled, a search on the phrase *on the waterfront* does not return hits for documents containing the phrase *On the waterfront*.

Composite Word Queries for German

German-language text contains composite words. With ConText, you can create a composite index and subsequently issue queries to search for composite words using a subcomposite word as your query term.

For example when using a German composite index, a query on the term *Bahnhof* (train station) returns documents that contain *Bahnhof* or any word containing *Bahnhof* as a sub-composite, such as *Hauptbahnhof*, *Nordbahnhof*, or *Ostbahnhof*.

However, a query on *Bahnhof* does not return documents that contain the single words *Bahn* or *Hof*.

To query against a composite index, you specify the policy associated with the composite index with two-step, or in-memory queries. For one-step queries, you must specify the policy if the text column has more than one index attached to it.

See Also: For more information about creating a composite index for German, see *Oracle8 Context Cartridge Administrator's Guide*.

Highlighting Composite Terms

You can use text highlighting with composite word queries in German. When you do so, ConText highlights the entire composite word, not just the sub-composite you entered as your query.

For example, when you issue *Bahnhof* as your query, context highlights the words *Hauptbahnhof*, *Nordbahnhof*, and *Ostbahnhof* entirely.

See Also: For more information on highlighting text queries, see Chapter 6, “Document Presentation”.

Base-Letter Queries

For languages that use an 8-bit character set, such as French and Spanish, Context gives you the option of converting characters to their base-letter representation before text indexing. This means that words with accents, umlauts, and so on are converted to their base-letter representation before their tokens are placed in the text index.

When you specify a text index that has used base-letter conversion in a query, ConText converts the term in the query expression to match the base-letter representation before the query is processed. In addition, all expansion and stopwords checking for the query is performed on the base-letter terms.

Note: The terms in a thesaural query are *not* converted to base-letter representation before look-up in the thesaurus. The base-letter conversion takes place after the thesaurus look-up and is performed on all the terms returned by the thesaurus.

For more information about creating an index that supports base-letter conversion, see *Oracle8 Context Cartridge Administrator's Guide*.

Query Expression Examples

The following example of a one-step query returns all articles that contain the word *wine* in the TEXTTAB.TEXT_COLUMN column. The query expression consists only of the query term *wine*, surrounded by single quotes.

```
SELECT articles FROM texttab
WHERE CONTAINS(textcol, 'wine') > 0;
```

The following example of a one-step query returns all articles that contain the phrase *wine and roses* in the TEXTTAB.TEXT_COLUMN column. The query expression consists of the query phrase *wine and roses*, surrounded by single quotes.

```
SELECT articles FROM texttab  
WHERE CONTAINS(textcol, '{wine and roses}') > 0;
```

See Also: For more information about the CONTAINS function used in one-step queries, see CONTAINS in Chapter 9.

Logical Operators

Logical operators combine the terms in a query expression. All single words and phrases may be combined with logical operators. When query terms are combined, the number of spaces around the logical operator is not significant.

Logical operators link query terms together to produce scores that are based on the relationship of the terms to each other. The logical operators combine the scores of their operands up to a maximum value of 100. Operands can be any query terms, as well as other operators.

Operator	Syntax	Description
AND	term1&term2 term1 and term2	Returns documents that contain <i>term1</i> and <i>term2</i> . Returns the minimum score of its operands. All query terms must occur; lower score taken.
OR	term1 term2 term1 or term2	Returns documents that contain <i>term1</i> or <i>term2</i> . Returns the maximum score of its operands. At least one term must exist; higher score taken.
NOT	term1~term2 term1 not term2	Returns documents that contain <i>term1</i> and not <i>term2</i> .
EQUIVALENCE	term1=term2 term1 equiv term2	Specifies that <i>term2</i> is an acceptable substitution for <i>term1</i> .

AND Operator

Use the AND operator to search for documents that contain at least one occurrence of *each* of the query terms. For example, to obtain all the documents that contain the terms *batman* and *robin* and *penguin*, issue the following query:

```
'batman & robin & penguin'
```

In an AND query, the score returned is the score of the lowest query term. In the example above, if the three individual scores for the terms *batman*, *robin*, and *penguin* is 10, 20 and 30 within a document, the document scores 10.

OR Operator

Use the OR operator to search for documents that contain at least one occurrence of *any* of the query terms. For example, to obtain the documents that contain the term *cats* or the term *dogs*, use one of the following:

```
'cats | dogs'
```

`'cats OR dogs'`

In an OR query, the score returned is the score for the highest query term. In the example above, if the scores for *cats* and *dogs* is 30 and 40 within a document, the document scores 40.

NOT Operator

Use the NOT operator to search for documents that contain one query term and not another.

For example, to obtain the documents that contain the term *animals* but not *dogs*, use the following expression:

`'animals ~ dogs'`

Similarly, to obtain the documents that contain the term *transportation* but not *automobiles* or *trains*, use the following expression:

`'transportation not (automobiles or trains)'`

Note: The NOT operator does not affect the scoring produced by the other logical operators.

Equivalence Operator

Use the equivalence operator to specify an acceptable substitution for a word in a search. For example, if you want all the documents that contain the phrase *alsatians are big dogs* or *labradors are big dogs*, you can write:

`'labradors=alsatians are big dogs'`

ConText processes the above query faster and more efficiently than the same query written with the accumulate operator. For example, you could write the above query less efficiently and less concisely as follows:

`'labradors are big dogs, alsatians are big dogs'`

The savings you gain in using the equivalence operator over the accumulate operator is most significant when you have more than one equivalence operator in the query expression. For example, the following query

`'labradors=alsatians are big canines=dogs'`

is a more efficient, more concise form of:

```
'labradors are big dogs,  
alsatians are big dogs,  
alsatians are big canines,  
labradors are big canines'
```

Precedence of Equivalence Operator

The equivalence operator has higher precedence than all other operators except the unary operators (fuzzy, soundex, stem, and PL/SQL function calls).

WITHIN Operator

Use the WITHIN operator to narrow down a query into pre-defined document sections.

For example in an HTML document set, you or your ConText administrator can define a section for all headings delimited with `<HEAD>` and `<\HEAD>` and subsequently issue a query for a term in a heading across all documents.

See Also: For more information about defining sections, see the *Oracle8 Context Cartridge Administrator's Guide*.

The syntax for the WITHIN operator is as follows:

Syntax	Description
<i>term</i> WITHIN <i>section</i>	Searches for <i>term</i> within the pre-defined <i>section</i> . The WITHIN operator has no effect on score.

Note: The WITHIN operator requires you to know the name of the section you wish to search. A list of defined sections can be obtained using the CTX_ALL_SECTIONS or CTX_USER_SECTIONS views.

Examples

To find all the documents that contain the term *San Francisco* within the pre-defined section *Headings*, write your query as follows:

```
'San Francisco within Headings'
```

To find all the documents that contain the term *sailing* and contain the term *San Francisco* within the pre-defined section *Headings*, write your query as follows:

```
'(San Francisco within Headings) and sailing'
```

To find all documents that contain the terms *dog* and *cat* within the pre-defined section *Headings*, write your query as follows:

```
'dog and cat within Headings'
```

Note that the above query is logically different from:

```
'dog within Headings and cat within Headings'
```

which finds all documents that contain *dog* and *cat* where the terms *dog* and *cat* are in different *Headings* sections.

To find all documents in which *dog* is near *cat* within the section *Headings*, write your query as follows:

```
'dog near cat within Headings'
```

Limitations

The WITHIN operator has the following limitations:

- The theme lexer does not support the WITHIN operator
- You cannot embed the WITHIN clause in a phrase. For example, you cannot write: *term1 WITHIN section term2*
- You cannot combine WITHIN with expansion operators
- You cannot combine WITHIN with the near operator as follows: *term1 WITHIN section near term2*
- Subqueries passed to WITHIN cannot use the Max or First/Next operators.
- You cannot nest the WITHIN operator For example, you cannot write: *dog WITHIN body WITHIN heading*.
- Since WITHIN is a reserved word, you must escape the word with braces to search on it.

Score-Changing Operators

Score changing operators behave like logical operators in that they return documents given the terms you specify. However, these operators affect document scores differently and, as such, can be used to change a document’s rank in a hitlist with respect to a query term. The following table describes these operators:

Operator	Syntax	Description
ACCUMULATE	term1,term2 term1 accum term2	Returns documents that contain <i>term1</i> or <i>term2</i> . Calculates score by adding the score of each operand. Similar to OR, except that the returned score is the <i>sum</i> of all scores.
MINUS	term1-term2 term1 minus term2	Returns documents that contain <i>term1</i> . Calculates score by subtracting occurrences of <i>term2</i> from occurrences of <i>term1</i> .
NEAR	term1;term2 term1 near term2	Returns documents that contain <i>term1</i> and <i>term2</i> . Calculates score based on how close <i>term1</i> is to <i>term2</i> ; a score of 100 means terms are adjacent to one another.
WEIGHT	term*n	Returns documents that contain <i>term</i> . Calculates score by multiplying the raw score of <i>term</i> by <i>n</i> , where <i>n</i> is a number from 0.1 to 10.

Accumulate Operator

Use the accumulate operator to search for documents that contain at least one occurrence of *any* of the query terms, where the documents that contain the most frequent occurrences of the query terms are given the highest score.

For example, to search for documents that contain either term *Brazil* or *soccer* and to have the highest scores attached to the documents that contain the most occurrences of these words, you can issue:

```
'soccer,Brazil'
```

Accumulate is similar to OR, in the sense that a document satisfies the query expression if any of the terms occur in the document; however, the scoring is different. OR returns a score based *only* on the query term that occurs most frequently in a document. Accumulate combines the scores for all the query terms that occur in a document, topping out at 100 when the sum exceeds 100. Thus documents that contain the most query terms are ranked the highest.

MINUS Operator

Use the MINUS operator to search for documents that contain a query term, and when you want the presence of a second query term to cause the document to be ranked lower.

The minus operator is useful for lowering the score of documents that contain "noise". For example, suppose a query on the term *cars* always returned high scoring documents about *Ford cars*. You can lower the scoring of the Ford documents by using the expression:

```
'cars - Ford'
```

In essence, this expression returns the documents that contain the term *cars*. However, the score returned for a document is the number of occurrences of *cars* minus the number of occurrences of *Ford*. When a returned document does not contain *Ford*, the occurrence of the term *Ford* is counted as zero.

Near Operator

Words or phrases that occur close together are considered to be more closely associated than those that are farther apart. The near operator calculates a score based on how close words are to each other rather than on how often the word or phrase appears in the document.

The score for a document is the highest score out of all the query terms that occur in proximity to each other. A score of 100 is returned when the query terms are adjacent. When the terms are not adjacent, ConText returns a score based on the following formula:

$$\text{score} = 100 - (\text{number of words between the two query terms})$$

When there are more than 100 words separating the terms, ConText scores the document as 1.

For example, if the query expression is *ice;cream*, the phrase *I love ice cream* would score 100, while the phrase *ice is colder than cream* would score 97. If both phrases occurred in a document, ConText retrieves the document and scores it as 100.

Near Operator with Multiple Words

You can use the near operator with more than two search terms. For example, you can issue a query such as:

```
fish;whales;sea
```

This query asks for all documents that have the three terms *fish*, *whales*, and *sea* close to one another. The score is calculated by the following formula:

score = 100 - size of the smallest block containing all query terms + number of query terms

Thus if a document contained the phrase: *Fish, lobsters and whales live in the sea*, and this phrase was the smallest block containing all three terms, the document scores $(100 - 8 + 3) = 95$.

Near Operator with Threshold

You can use the threshold operator with the near operator to restrict your result set. For example, to request all documents in which the terms fish and whales are at most three words apart or less, you can write:

```
fish:whales > 97
```

Weight Operator

The *weight* operator multiplies the score by the given factor, topping out at 100 when the product exceeds 100. For example, the query *cat, dog*2* sums the score of *cat* with twice the score of *dog*, topping out at 100 when the score is greater than 100.

In expressions that contain more than one query term, use the weight operator to adjust the relative scoring of the query terms. You can reduce the score of a query term by using the weight operator with a number less than 1; you can increase the score of a query term by using the weight operator with a number greater than 1 and less than 10.

The weight operator is useful in accumulate, OR, or AND queries when the expression has more than one query term. With no weighting on individual terms, the score cannot tell you which of the query terms occurs the most. If you are interested in documents that contain a particular query term more than another term, the overall ranking tells you nothing about which documents pertain to the term that you are most interested in.

Example

You have a collection of sports articles. You are interested in the articles about soccer, in particular Brazilian soccer. It turns out that a regular query on *soccer, Brazil* returns many high ranking articles on US soccer. To raise the ranking of the articles on Brazilian soccer, you can issue the following query:

```
'soccer, Brazil*3'
```

Table 3–1 illustrates how the weight operator can change the ranking of three hypothetical documents A, B, and C, which all contain information about soccer. The columns in the table show the total score of four different query expressions on the three documents.

	<i>soccer</i>	<i>Brazil</i>	<i>soccer,Brazil</i>	<i>soccer,Brazil*3</i>
A	20	10	30	50
B	10	30	40	100
C	50	10	60	70

Table 3–1

The score in the third column containing the query *soccer, Brazil* is the sum of the scores in the first two columns. The score in the fourth column containing the query *soccer,Brazil*3* is the sum of the score of the first column *soccer* plus three times the score of the second, *Brazil*.

With the initial query of *soccer,Brazil*, the documents are ranked in the order C B A. With the query of *soccer,Brazil*3*, the documents are ranked B C A, which is the preferred ranking.

Result-Set Operators

Use the result-set operators to control what documents are returned from a query result set. The operands for these operators are expressions, which can be an individual query term or a logical combination of query terms that use other operators.

Note: Because these operators manipulate a result set, they cannot be embedded within each other; they must be placed at the outermost level of the query expression.

These operators also have no effect on highlighting with CTX_QUERY.HIGHLIGHT.

Result set operators are typically used to exclude noise from the hitlist (irrelevant documents) and to retrieve documents out of a hitlist more efficiently. There are three result set operators:

Operator	Syntax	Description
THRESHOLD	<i>expression</i> > <i>n</i>	Returns only those documents in the result set that score above the threshold <i>n</i> .
	<i>term</i> > <i>n</i>	Within an expression, selects documents that contain the query term with score of at least <i>n</i> .
MAX	<i>expression</i> : <i>n</i>	Returns the first <i>n</i> highest scoring documents. For example, :20 means to return the top 20 documents in the hitlist. The value <i>n</i> must be an integer between 1 and 65535.
FIRST/NEXT	<i>expression</i> # <i>m</i> - <i>n</i>	Returns the specified number of documents as ordered in the hitlist range <i>m</i> to <i>n</i> .

Threshold Operator

You can use the threshold operator in two ways:

- at the expression level
- at the query term level

Expression level

Use the expression level threshold operator to eliminate documents in the result set that score below a threshold number. For example, to search for documents that

contain *relational databases* and to return only documents that score greater than 75, use the following expression:

```
'relational databases > 75'
```

Query Term Level

Use the query term threshold operator in a query expression to select a document based on how a term scores in the document. For example, to select documents that have at least a score of 30 for *lion* and contain *tiger*, use:

```
'(lion > 30) and tiger'
```

Max Operator

Use the max operator to retrieve a given number of the highest scoring documents. For example, to obtain the twenty highest scoring documents that contain the word *dance*, you can write:

```
'dance:20'
```

The max operator is particularly useful to prevent writing a large number of records to the hitlist table, which could result in performance degradation.

Note: The max operator cannot be used with the CTX_QUERY.COUNT_HITS function or with in-memory queries.

First/Next Operator

Use the first/next operator to return a specified range of documents from the hitlist.

Note: In a first/next query, the order of the returned documents is *not* based on score or textkey. ConText returns the documents based on the order in which it encounters the documents in the queried text column

For example, to return the first 10 documents encountered by ConText that contain the term *dog*, use the following expression:

```
'dog#1-10'
```

You could then return the next 10 documents using the following expression:

```
'dog#11-20'
```

The first/next operator can be used to create an application interface in which query results (rows in the hitlist) are returned incrementally. Because the query results are returned incrementally, query response is generally faster. The application can display the hitlists in a more manageable size, and control can be returned to the user faster.

Note: The first/next operator cannot be used with the CTX_QUERY.COUNT_HITS function or with in-memory queries.

Combined First/Next and Max Queries

You can use the first/next operator extract chunks of a sorted hitlist returned by the max operator. For example, if you use the max operator to return only the highest scoring 50 documents that contain the term *cat*, you can extract the first 10 documents from the 50 as follows:

```
'cat:50#1-10'
```

Note: Placing the max operator inside the first/next operator as such is the only instance in which you can embed the max operator in a query expression.

Expansion Operators

The expansion operators expand a query expression to include variants of the query term supplied by the user. There are three kinds of expansion operators:

Operator	Syntax	Description
STEM	\$term	Expands a query to include all terms having the same stem or root word as the specified term.
SOUNDEX	!term	Expands a query to include all terms that sound the same as the specified term (English-language text only).
FUZZY	?term	Expands a query to include all terms with similar spellings as the specified term (English-language text only).

The expansion operators are unary operators. They may be used in combination with each other and with any other operators described in this chapter. In addition, searches can be broadened by performing an expansion on an expansion.

The methods used by the expansion operators to perform stemming, fuzzy matching, and soundex matching for a text column are determined by the Wordlist preference in the policy for the column.

See Also: For more information about setting up preferences and policies, see *Oracle8 Context Cartridge Administrator's Guide*.

Stem Expansions

Use the STEM (\$) operator to search for terms that have the same linguistic root as the query term. For example:

Input	Expands To
\$scream	scream screaming screamed
\$distinguish	distinguish distinguished distinguishes
\$guitars	guitars guitar
\$commit	commit committed
\$cat	cat cats
\$sing	sang sung sing

The ConText stemmer, licensed from Xerox Corporation's XSoft Division, supports the following languages: English, French, Spanish, Italian, German, and Dutch.

Note: If STEM returns a stopwords, the stopwords is not included in the query or highlighted by CTX_QUERY.HIGHLIGHT.

Soundex Expansions

The soundex (!) operator enables searches on words that have similar sounds; that is, words that sound like other words. This function allows comparison of words that are spelled differently, but sound alike in English.

Soundex in ConText uses the same logic as the soundex function in SQL to search for words that have a similar sound. It returns all words in a text column that have the same soundex value.

The following example illustrates the results that could be returned for a one-step query that uses SOUNDEX:

```
SELECT ID, COMMENT FROM EMP_RESUME
WHERE CONTAINS (COMMENT, '!SMYTHE') > 0
```

```
ID COMMENT
--
23 Smith is a hard worker who..
```

Note: SOUNDEX works best for languages that use a 7-bit character set, such as English. It can be used, with lesser effectiveness, for languages that use an 8-bit character set, such as many Western European languages.

For more information about the SOUNDEX function in SQL, see *Oracle8 Server SQL Reference*.

Fuzzy Expansions

Fuzzy (?) expansions generate words that are spelled similarly. This type of expansion is helpful for finding more accurate results when there are frequent misspellings in the documents in the database.

Unlike the stem expansion, the number of words generated by a fuzzy search depends on what is in the text index; results can vary significantly according to the contents of the database index.

For example:

Input	Expands To
?cat	cat cats calc case
?feline	feline defined filtering
?apply	apply apple applied April
?read	lead real

Note: Fuzzy works best for languages that use a 7-bit character set, such as English. It can be used, with lesser effectiveness, for languages that use an 8-bit character set, such as many Western European languages. Also, the Japanese lexer provides limited fuzzy matching.

In addition, if fuzzy returns a stopword, the stopword is not included in the query or highlighted by CTX_QUERY.HIGHLIGHT.

Penetration in Expansion Operators

Penetration allows complex query expansions to be expressed in short concise notation. Penetration is a system of notation for query expressions and does not affect the meaning of the expansion operators or the order in which operations are performed; it is a tool to help you generate non-ambiguous queries using the expansion operators.

Penetration applies the expansion operators to each term within an explicit expression (i.e., an expression delimited by parentheses or braces). Any expansion operators outside an expression delimited by parentheses () or braces { } is applied to each word or phrase inside the expression.

For example:

Query Before Penetration	Query After Penetration
?(dog, cat, mouse)	?dog, ?cat, ?mouse

Query Before Penetration	Query After Penetration
?(dog,! (cat & mouse))	?dog, (!?cat & !?mouse)
?((cat=feline) meows)	(?cat =?feline)?meows

In the first example, a fuzzy expansion is performed on each term.

In the second example, a fuzzy expansion is performed on each term and a soundex expansion is performed only on the terms cat and mouse because *cat* and *mouse* are enclosed in a separate set of parentheses

In the third example, a fuzzy expansion is performed on each term, including both equivalence terms.

Note: Expansion operators do not penetrate expressions delimited by brackets [].

Examining Query Expansions

You can use query expression feedback to examine how ConText expands query expressions containing fuzzy, stem and soundex operators.

See Also: Chapter 5, “Query Expression Feedback”.

Base-letter Support

If you have base-letter conversion specified for a text column and the query expression contains a SOUNDEX or FUZZY operator, ConText operates on the base-letter form of the query.

The STEM operator does not support base-letter conversion.

Thesaurus Operators

The thesaurus operators expand a query for a single term (word or phrase) using a thesaurus that defines relationships between the user-specified term and other semantically related terms.

There are ten kinds of thesaurus operators, corresponding to the ten types of relationships that can be defined in an ISO2788 standard thesaurus.

Operator	Syntax	Description
SYNONYM	SYN(term[,thes])	Expands a query to include all the terms defined in the thesaurus as synonyms for <i>term</i> .
PREFERRED	PT(term[,thes])	Replaces the specified word in a query with the preferred term for <i>term</i> .
RELATED	RT(term[,thes])	Expands a query to include all the terms defined in the thesaurus as a related term for <i>term</i> .
TOP	TT(term[,thes])	Replaces the specified word in a query with the top term in the standard hierarchy (BT, NT) for <i>term</i> .
NARROWER	NT(term[,level[,thes]])	Expands a query to include all the lower level terms defined in the thesaurus as narrower terms for <i>term</i> .
NARROWER GENERIC	NTG(term[,level[,thes]])	Expands a query to include all the lower level terms defined in the thesaurus as narrower generic terms for <i>term</i> .
NARROWER PARTITIVE	NTP(term[,level[,thes]])	Expands a query to include all the lower level terms defined in the thesaurus as narrower partitive term for <i>term</i> .
NARROWER INSTANCE	NTI(term[,level[,thes]])	Expands a query to include all the lower level terms defined in the thesaurus as narrower instance term for <i>term</i> .
BROADER	BT(term[,level[,thes]])	Expands a query to include the term defined in the thesaurus as a broader term for <i>term</i> .
BROADER GENERIC	BTG(term[,level[,thes]])	Expands a query to include all terms defined in the thesaurus as a broader generic terms for <i>term</i> .

Operator	Syntax	Description
BROADER PARTITIVE	BTP(term[,level[,thes]])	Expands a query to include all the terms defined in the thesaurus as broader partitive terms for <i>term</i> .
BROADER INSTANCE	BTI(term[,level[,thes]])	Expands a query to include all the terms defined in the thesaurus as broader instance terms for <i>term</i> .

Internally, ConText processes the expansion by bracketing each individual term returned by the expansion, then the terms are accumulated together using the ACCUMULATE operator.

For example, if *bird*, *birdy*, and *avian* are all synonyms:

SYN(bird) is expanded to *{bird},{avian},{birdy}*.

If a term in a thesaural query does not have corresponding entries in the specified thesaurus, no expansion is produced and the term itself is used in the query.

See Also: For more information about viewing thesaural expansions, see Chapter 5, “Query Expression Feedback”.

For more information about thesaural relationships and creating thesauri, see *Oracle8 Context Cartridge Administrator's Guide*.

Limitations

The thesaurus operators can be used in conjunction with all the other query expression operators and special characters supported by ConText, with the *exception* of the near operator.

The maximum length of the expanded query is 32000 characters.

Thesaural operations cannot be nested. For example, the following query is *not* allowed.

```
'SYN(BT(bird))'
```

Thesaurus Arguments

The thesaurus operators are implemented in ConText as PL/SQL functions, and, as such, have arguments that must be specified with the operator. All of the notational conventions and usage rules for PL/SQL apply to the thesaurus operators.

The thesaurus operators have the following arguments:

term

Specify the operand for the thesaurus operator. You *must* specify a term when using the NT operator. For preferred term (PT) and top term (TT) queries, *term* is replaced by the preferred term/top term defined for the term in the specified thesaurus; however, if no PT or TT entries are defined for the term, the term is not replaced and is used in the query.

For all other thesaural queries, *term* is expanded to include the synonymous, related, broader, or narrower terms defined for the term in the specified thesaurus.

level

Specify the number of levels traversed in the thesaurus hierarchy to return the broader (BT, BTG, BTP) or narrower (NT, NTG, NTP) term for the specified term. For example, a level of 1 in a BT query returns only the broader term, if one exists, for the specified term. A level of 2 returns the broader term for the specified term, as well as the broader term, if one exists, for the broader term.

The level argument is optional and has a default value of one (1). Zero or negative values for the level argument return only the original query term.

thes

Specify the name of the thesaurus used to return the expansions for the specified term. The *thes* argument is optional and has a default value of DEFAULT. As a result, a thesaurus named DEFAULT *must* exist in the thesaurus tables before using any of the thesaurus operators.

Synonym Operator

Use the synonym operator (SYN) to expand a query to include all the terms that have been defined in a thesaurus as synonyms for a specified term.

The following query returns all documents that contain the term *tutorial* or any of the synonyms defined for *tutorial* in the DEFAULT thesaurus:

```
'SYN(tutorial)'
```

Compound Phrases in Synonym Operator

Expansion of compound phrases for a term in a synonym query are returned as AND conjunctives.

For example, the compound phrase *temperature + measurement + instruments* is defined in a thesaurus as a synonym for the term *thermometer*. In a synonym query for *thermometer*, the query is expanded to:

```
{thermometer},{({temperature}&{measurement}&{instruments})}
```

Note: In a thesaurus, compound phrases can only be defined in synonym relationships for a term.

Preferred Term Operator

Use the preferred term operator (PT) to replace a term in a query with the preferred term that has been defined in a thesaurus for the term.

For example, the term *building* has a preferred term of *construction* in a thesaurus. A PT query for *building* returns all documents that contain the word *construction*. Documents that contain the word *building* are not returned.

Related Term Operator

Use the related term operator (RT) to expand a query to include all terms with the related term that has been defined in a thesaurus for the term.

For example, the term *dinosaur* has a related term of *paleontology*. A RT query for *dinosaur* returns all documents that contain the word *paleontology*. Documents that contain the word *dinosaur* are not returned.

Narrower Term Operators

Use the narrower term operators (NT, NTG, NTP, NTI) to expand a query to include all the terms that have been defined in a thesaurus as the narrower or lower level terms for a specified term. They can also expand the query to include all of the narrower terms for each narrower term, and so on down through the thesaurus hierarchy.

Note: The hierarchy can contain four separate branches, represented by the four narrower term operators. During a narrower term query, the specified operator only searches down the designated branch of the hierarchy.

The following query returns all documents that contain either the term *tutorial* or any of the NT terms defined for *tutorial* in the DEFAULT thesaurus:

```
'NT(tutorial)'
```

The following query returns all documents that contain either *fairy tale* or any of the narrower instance terms for *fairy tale* as defined in the DEFAULT thesaurus:

```
'NTI(fairy tale)'
```

That is, if the terms *cinderella* and *snow white* are defined as narrower term instances for *fairy tale*, ConText returns documents that contain *fairy tale*, *cinderella*, or *snow white*.

Broader Term Operators

Use the broader term operators (BT, BTG, BTP, BTI) to expand a query to include the term that has been defined in a thesaurus as the broader or higher level term for a specified term. They can also expand the query to include the broader term for the broader term and the broader term for that broader term, and so on up through the thesaurus hierarchy.

Note: The hierarchy can contain four separate branches, represented by the four broader term operators. In a broader term query, the specified operator only searches up the designated branch of the hierarchy.

The following query returns all documents that contain the term *tutorial* or the BT term defined for *tutorial* in the DEFAULT thesaurus:

```
'BT(tutorial)'
```

Broader and Narrower Term Operator on Homographs

If a homograph (a word or phrase with multiple meanings, but the same spelling) appears in two or more nodes in the same hierarchy branch of a thesaurus, a qualifier is required for each occurrence of the term in the branch.

If the qualifier is not specified for a homograph in a broader or narrower term query, the query expands to include all of the broader/narrower terms for the homograph.

For example, if *machine* is a broader term for *crane (building equipment)* and *bird* is a broader term for *crane (waterfowl)*:

BT(crane) expands to *{crane},{machine},{bird}*

If the qualifier for a homograph is specified in a broader or narrower term query, only the broader/narrower terms for the qualified homograph are returned.

Using the previous example:

BT(crane{(waterfoul)}) expands to *{crane},{bird}*

Note: When specifying a qualifier in a broader or narrower term query, the qualifier and its notation (parentheses) must be escaped, as is shown in this example.

Top Term Operator

Use the TOP TERM operator (TT) to replace a term in a query with the top term that has been defined for the term in the standard hierarchy (BT, NT) in a thesaurus. Top terms in the generic (BTG, NTG) and partitive (BTP, NTP) hierarchies are not returned.

For example, the term *tutorial* has a top term of *learning systems* in the standard hierarchy of a thesaurus. A TT query for *tutorial* returns all documents that contain the phrase *learning systems*. Documents that contain the word *tutorial* are not returned.

Thesaural Expansions and Case-Sensitivity

Thesaural expansions in text queries can differentiate between terms based on case.

For example, a case-sensitive thesaurus named *thes1* is created and *Mercury* is defined as a narrower term for *planets*, while *mercury* is defined as a narrower term for *metals*.

During a query, the following expansions occur:

BT(mercury,1,thes1) expands to *{MERCURY}, {METALS}*

BT(Mercury,1,thes1) expands to *{MERCURY}, {PLANETS}*

Note: There is no way to enable or disable case-sensitivity. ConText preserves the case of all entries entered in a thesaurus based on whether the thesaurus was specified during creation to be case-sensitive. Similarly, text queries use the cases of terms to perform the thesaural look-up based on the thesaurus specified for the term(s).

Limitations

Because text queries are case-insensitive, case-sensitive thesauri only affect the expansion of a term and not the terms actually used in the query.

For example:

BT(Mercury,1,thes1) expands to {MERCURY}, {PLANETS}

However, the query returns all documents in which the two terms occur, regardless of case. In other words, documents that contain *mercury*, *Mercury*, *planets*, *Planets*, or any other combinations of case for the two terms are all returned by the query.

Base-letter Support for Thesaural Queries

When ConText processes a query on a base-letter index and the expression contains a thesaurus operator, ConText looks up the query term in the thesaurus without converting the query to base-letter. The expansions obtained from the thesaurus are converted to base-letter and looked up subsequently within the index according to query rules.

This sequence of look-up enables base-letter queries to work independent of whether the thesaurus is in base-letter form. However, if the keys in the thesaurus are in base letter form, these keys will not match the corresponding non-base letter form query terms. When you have a base-letter thesaurus, you must specify the base-letter form in the query.

Wildcard Characters

Wildcard characters can be used in query expressions to expand word searches into pattern searches. The wildcard characters are:

Wildcard Character	Description
%	The percent wildcard specifies that any characters can appear in multiple positions represented by the wildcard.
_	The underscore wildcard specifies a single position in which any character can occur.

For example, the following abbreviated one-step query finds all terms beginning with the pattern *scal* in a column named *text*:

```
...contains(TEXT, 'scal%') > 0
```

Note: To expand the wildcard query, ConText uses the word list for the text column and rewrites the query with these terms. When your wildcard query expands to a number of terms greater than the maximum allowed in a query, ConText returns an error.

In addition, if a wildcard expression translates to a stopword, the stopword is not included in the query or highlighted by CTX_QUERY.HIGHLIGHT.

Grouping Characters

The grouping characters control operator precedence by grouping query terms and operators in a query expression. The grouping characters are:

- parentheses ()
- brackets []

The beginning of a group of terms and operators is indicated by an open character from one of the sets of grouping characters. The ending of a group is indicated by the occurrence of the appropriate close character for the open character that started the group. Between the two characters, other groups may occur.

For example, the open parenthesis indicates the beginning of a group. The first close parenthesis encountered is the end of the group. Any open parentheses encountered before the close parenthesis indicate nested groups.

Brackets perform the same function as the parentheses, but prevent penetration for the expansion operators.

Stored Query Expressions

You can store the results of a query expression and then call the SQE later in a query expression to return the stored results. To call a stored query expression, use the SQE operator.

Operator	Syntax	Description
Stored Query Expression	SQE(SQE_name)	Returns the stored result of <i>SQE_name</i> .

The advantage of calling an SQE in a query expression, rather than specifying query terms, is that the results are typically returned faster, since ConText does not have to query the text table directly.

In addition, SQEs can be used to perform iterative queries, in which an initial query is refined using one or more additional queries.

Using Stored Query Expressions

The process for using stored query expressions is:

1. Call CTX_QUERY.STORE_SQE to store the results for the text column or policy. With STORE_SQE, you specify a name for the SQE, a policy (which identifies the text column for the SQE), a query expression, and whether the SQE is a session or system SQE
2. Call the stored query expression in the query expression of a text (or theme) query. ConText returns the results of the SQE in the same way it returns the results of a regular query. If the results of the SQE are out-of-date, ConText automatically re-evaluates the SQE before returning the results.

Note: Because ConText must first determine if the results are out-of-date with respect to the document index, many changes to the index though inserting, deleting, and updating documents will slow down the retrieval of the stored query expression results.

Administration of stored query expressions can be performed using the REFRESH_SQE, REMOVE_SQE, and PURGE_SQE procedures in the CTX_QUERY PL/SQL package.

Example

To create a session SQE named PROG_LANG, use CTX_QUERY.STORE_SQE as follows:

```
exec ctx_query.store_sqe('emp_resumes', 'prog_lang', 'cobol', 'session');
```

This SQE queries the text column for the EMP_RESUMES policy (in this case, EMP.RESUMES) and returns all documents that contain the term *cobol*. It stores the results in the SQE table for the policy.

PROG_LANG can then be called within a query expression as follows:

```
select score, docid from emp
where contains(resume, 'sqe(prog_lang)')>0
order by score;
```

Session and System SQEs

When you initially create an SQE using CTX_QUERY.STORE_SQE, you can specify whether the SQE is for the current session or for all sessions (system SQE).

You can use session SQEs only in the current session. These SQEs are stored only for the duration of the session. When a session is terminated, all session SQEs created during the session are deleted from the SQE tables. If you want to use a session SQE in another session, you must recreate the SQE.

System SQEs can be used in all sessions, including concurrent sessions. When a session is terminated, system SQEs created during the session are *not* deleted from the SQE tables and can be used in future sessions.

Re-evaluation of Stored Query Expressions

If the text column referenced by an stored query expression has been modified since the stored query expression was created, the stored query expression results may be out-of-date. Before returning the results of an stored query expression in a query expression, ConText verifies that the results are current. If they are not current, ConText automatically evaluates the differences and updates the results.

ConText also verifies that any stored query expressions nested within an stored query expression have up-to-date results

Note: ConText does not verify whether PL/SQL functions in stored query expressions have been updated. If a PL/SQL function in an stored query expression has been updated, the stored query expression must be manually re-evaluated.

Result lists in stored query expression tables may get fragmented by consecutive re-evaluations. You can resolve fragmentation by calling `CTX_QUERY.REFRESH_SQE`.

Iterative Queries

Iterative queries are queries built on other queries to refine or add to the result set of the original query. Once you define a stored query expression, you can add additional search criteria in two ways:

- extending the expression in the `CONTAINS` procedure
- nesting SQEs

Extending the Expression in the `CONTAINS` Procedure

Sometimes you might want to add a condition to a stored query expression to re-define your search criteria. You can do so by extending the query with additional operators when you call `CTX_QUERY.CONTAINS`. When you extend stored queries in this way, the response time is usually faster than an equivalent query without the `SQE` operator.

For example, you find that wildcard queries take a long time to process. You therefore define a wildcard query as a stored query expression, `Q1`, to return all documents indexed under policy *pol* that have words beginning with the letter *z*:

```
ctx_query.store_sqe('pol', 'Q1', 'z%', 'session');
```

You then extend the query by adding an `OR` condition: You ask for all documents indexed under policy *pol* that contain words beginning with the letter *z* or contains the word *cat*:

```
ctx_query.contains('pol', 'SQE(Q1) | cat', 'ctx_temp');
```

Internally, ConText must still use the text index to find those documents that might have the word *cat* but not *z*%; however, the response time is generally much faster than the following equivalent query:

```
ctx_query.contains('pol', 'z% | cats', 'ctx_temp');
```

Nesting Stored Query Expressions

You can use stored query expressions to define other stored query expressions. This is useful when you want to refine the result set returned from a stored query expression.

For example, you define the stored query expression, Q1 as follows:

```
ctx_query.store_sqe('pol', 'Q1', 'lions | tigers', 'session');
```

You then want to reduce this hitlist by adding another condition, so you define Q2 as follows:

```
ctx_query.store_sqe('pol', 'Q2', 'SQE(Q1) and zoos', 'session');
```

You then execute Q2 as follows:

```
ctx_query.contains('pol', 'SQE(Q2)', 'ctx_temp');
```

This query searches for all documents that contain the terms *lions* or *tigers* and *zoos*. It is generally faster than the following equivalent query:

```
ctx_query.contains('pol', 'lions | tigers and zoos', 'ctx_temp');
```

SQE Tables

Each stored query expression is stored in two tables: a central or system table owned by CTXSYS and an text index table attached to the policy for which the stored query expression was created.

The table owned by CTXSYS is an internal table which stores the stored query expression definitions for all the stored query expressions that have been created for all existing policies. It cannot be accessed directly, but can be viewed through two views, CTX_SQES (users with CTXADMIN role) and CTX_USER_SQES (users with CTXAPP and CTXADMIN roles).

The table used to store the results of an stored query expression for a text column is one of the tables created automatically when the column is indexed; however, the SQR table is only populated when an stored query expression is created and updated when an stored query expression is re-evaluated.

The tablespace, storage clause, and other parameters used to create the SQR table are specified by the Engine preference in the policy for the text column of the stored query expression.

Note: Similar to the other ConText index tables, the SQR table is an internal table that is accessed only by ConText when an stored query expression is processed in a query.

For more information about policies, preferences, text indexing, and the structure of the stored query expression tables and views, see *Oracle8 Context Cartridge Administrator's Guide*.

Using Operators in Stored Query Expressions

You can use all query expression operators in stored query expressions, with the following exceptions:

- Max
- First/Next

Stored query expressions also support all of the special characters and other components that can be used in a query expression, including PL/SQL functions and other stored query expressions.

PL/SQL in Query Expressions

In a query expression, you can call a PL/SQL function that returns a value. The syntax for the PL/SQL operator is as follows:

Syntax	Description
@owner_name.fname(arg1, arg2,...,argn)	Executes <i>fname()</i> where <i>fname()</i> returns a value. Return values that are not of type VARCHAR2 are cast into strings when possible. If <i>fname()</i> does not return a value, an exception is raised.
execute owner_name.fname()	
exec owner_name.fname()	

Example

Calling a PL/SQL function within a query is useful for converting words to alternate forms. For example, you can call a function that takes acronyms and returns the expanded string.

Suppose you, as user *ctxuser*, create a function named CONVERT that takes an acronym as input and returns the fully-expanded version of the acronym. Then, to obtain all documents that contain either *IBM* or *International Business Machine*, you issue the following query:

```
'execute ctxuser.convert(IBM), IBM'
```

Likewise, you can call a PL/SQL function that translates words. For example, you can call a function *french* that converts an English word to its French equivalent. You can then search on the French word for *cat* by issuing the following query:

```
'@ctxuser.french(cat)'
```

Operator Precedence

Operator precedence is the order in which the components of a query expression are evaluated. ConText query operators can be divided into two sets of operators that have their own order of evaluation. These two groups are described below as Group 1 and Group 2.

In all cases, query expressions are evaluated in order from left to right according to the precedence of their operators. Operators with higher precedence are applied first. Operators of equal precedence are applied in order of their appearance in the expression from left to right.

Group 1

Within query expressions, the Group 1 operators have the following order of evaluation from highest precedence to lowest:

EQUIV	=
NEAR	;
Weight, Threshold	* >
MINUS	-
NOT	~
WITHIN	
AND	&
OR	
ACCUM	,
Max	:
First/Next	#

Group 2

Within query expression, the Group 2 operators have the following order of evaluation from highest to lowest:

Wildcard	% _
Stem	\$
Fuzzy	?
Soundex	!

Procedural Operators

Other operators not listed under Group 1 or Group 2 are procedural. These operators have no sense of precedence attached to them. They include the SQE, PL/SQL, and thesaurus operators.

Precedence Examples

Query Expression	Order of Evaluation
w1 w2 & w3	(w1) (w2 & w3)
w1 & w2 w3	(w1 & w2) w3
?w1, w2 w3 & w4	(?w1), (w2 (w3 & w4))
abc = def ghi & jkl = mno	((abc = def) ghi) & (jkl=mno)
dog and cat WITHIN body	(dog and cat) WITHIN body

In the first example, because AND has a higher precedence than OR, the query returns all documents that contain *w1* and all documents that contain both *w2* and *w3*.

In the second example, the query returns all documents that contain both *w1* and *w2* and all documents that contain *w3*.

In the third example, the fuzzy operator is first applied to *w1*, then the AND operator is applied to arguments *w3* and *w4*, then the OR operator is applied to term *w2* and the results of the AND operation, and finally, the score from the fuzzy operation on *w1* is added to the score from the OR operation.

The fourth example shows that the equivalence operator has higher precedence than the AND operator.

The fifth example shows that the AND operator has higher precedence than the WITHIN operator.

Altering Precedence

Precedence is altered by grouping characters as follows:

- expansion or execution of operations within parentheses is resolved before other expansions regardless of operator precedence

Precedence of operators is maintained during evaluation of expressions inside of the parentheses.

- expansion operators are not applied to expressions within brackets unless the operators are also within the brackets

Escaping Reserved Words and Characters

To query on words or symbols that have special meaning to query expressions such as *and* & *or* / *accum*, *execute*, you must escape them. There are two ways to escape characters in a query expression:

Escape Symbol	Meaning
{ }	Use braces to escape a string of characters or symbols. Everything within a set of braces is considered part of the escape sequence.
\	Use the backslash character to escape an individual character or symbol. Only the character immediately following the backslash is escaped.

Example

In the following examples, an escape sequence is necessary because each expression contains a ConText operator or reserved symbol:

```
'AT\&T'  
'{AT&T}'  
  
'high\-voltage'  
'{high-voltage}'
```

Note: If you use braces to escape an individual character within a word, the character is escaped, but the word is broken into three tokens.

For example, a query written as *high{-}voltage* searches for *high - voltage*, with the space on either side of the hyphen.

Reserved Words

The following is a list of ConText reserved words and characters that must be escaped to be searched on:

Operator	Reserved Word	Equivalent Reserved Character
And	AND	&
Or	OR	
Accumulate	ACCUM	,
Minus	MINUS	-
Near	(none)	;
Stem	(none)	\$
Soundex	(none)	!
Fuzzy	(none)	?
Threshold	(none)	>
Weight	(none)	*
First/Next	(none)	#
Max	(none)	:
Wildcard (multiple)	(none)	%
Wildcard (single)	(none)	_
Grouping (parentheses)	(none)	()
Grouping (brackets)	(none)	[]
Escape (multiple characters)	(none)	{ }
Escape (single character)	(none)	\
PL/SQL call	EXECUTE EXEC	@
Stored Query Expression	SQE	(none)
Synonym	SYN	(none)
Preferred	PT	(none)
Related	RT	(none)
Top	TT	(none)

Operator	Reserved Word	Equivalent Reserved Character
Broader	BT	(none)
Narrower	NT	(none)
Broader Generic	BTG	(none)
Narrower Generic	NTG	(none)
Broader Partitive	BTP	(none)
Narrower Partitive	NTP	(none)

Querying Escape Characters

The open brace { signals the beginning of the escape sequence, and the closed brace} indicates the end. Everything between the opening brace and the closing brace is part of the query expression (including any open brace characters). To include the close brace character in a query expression, use}}.

To escape the backslash escape character, use \\.

Querying with Stopwords

Stopwords are words for which ConText does not create an index entry. They are usually common words that are unlikely to be searched on by themselves.

ConText is shipped with a default list of stopwords in English containing common words such as *this* and *that*. However, you or ConText administrator can define stopwords.

See Also: For more information about defining stopwords, see *Oracle8 Context Cartridge Administrator's Guide*.

Stopwords by Themselves

You cannot query on a stopword by itself or a phrase of only stopwords; whenever you attempt to query on a stopword by itself or a stopword-only phrase, the result is always no hits.

For example, you cannot issue a query to retrieve all documents that contain *this* if *this* is defined as a stopword, nor can you issue a query on a phrase of stopwords such as *the who*, if the words *the* and *who* are defined as stopwords.

Stopwords with Non-stopwords

You can query on phrases that contain stopwords as well as non-stopwords, such as *this boy talks to that girl*, where *this* and *that* are the only stopwords. This is possible because Context records the position of stopwords even though it does not create an index entry for them.

Case-Sensitivity

If you have case-sensitivity enabled for text queries and you issue a query on a phrase containing stopwords and non-stopwords, you must specify the correct case for the stopwords. For example, a query on *this boy talks to that girl* does not return documents that containing the phrase *This boy talks to that girl*, assuming *this* is a stopword.

See Also: For more information about issuing case-sensitive text queries, see “Case-Sensitive Queries” in this chapter.

Stopwords with Operators

When you use a stopword or a stopword-only phrase as an operand of a query operator, ConText rewrites the expression to eliminate the stopword or stopword-only phrase and then executes the query.

The following table describes *some* common stopwords transformations. The *Stopword Expression* column describes the query expression or component of a query expression you enter, while the right-hand column describes the way ConText rewrites the query.

In these examples, a value of *no_token* for the rewritten expression means no hits are returned for the query.

Stopword Expression	Rewritten Expression
<i>non_stopword</i> AND <i>stopword</i>	<i>non_stopword</i>
<i>stopword</i> AND <i>non_stopword</i>	<i>non_stopword</i>
<i>stopword</i> AND <i>stopword</i>	<i>no_token</i>
<i>non_stopword</i> NOT <i>stopword</i>	<i>non_stopword</i>
<i>stopword</i> NOT <i>non_stopword</i>	<i>no_token</i>
<i>stopword</i> NOT <i>stopword</i>	<i>no_token</i>

For example, assuming that the word *this* is a stopwords and that the word *dog* is a non-stopword, the query *dog and that* is rewritten to *dog*, applying the first transformation is the list.

See Also: For a complete list of stopwords transformations, see Appendix C, “Stopword Transformations”.

To learn about how to examine stopwords transformations, see Chapter 5, “Query Expression Feedback”.

Querying with Special Characters

Context indexes text by identifying tokens (words). For English and most European languages it assumes that blank spaces delimit tokens. At index time, ConText must also know how to interpret punctuation characters and characters that occur within words and numbers. Such special characters must be defined in the BASIC LEXER preference. They are described as follows:

Type of Character	Description
Punctuations	Characters that delimit the end of sentences such as the period '.' and question mark '?' and those that occur next to words and numbers, such as the comma ',' and the dollar sign '\$'. These characters are not indexed.
Continuation	Characters that indicate a word continues on the next line. An example is the hyphen '-'. These characters are not indexed.
Printjoins	Characters that join words together such as hyphen '-'. These characters are indexed.
Skipjoins	Characters that join words together such as hyphen '-'. These characters are not indexed.
Numjoin	Characters that occur in numbers such as the decimal point '.'. These characters are indexed.
Numgroup	Characters that group digits within a number such as the comma ','. These characters are indexed.
Startjoin	Non-alphanumeric characters that occur at the beginning of a token. For example, you can define < as a startjoin character for HTML tagged text. These characters are indexed.
Enjoin	Non-alphanumeric characters that occur at the end of a token. For example, you can define > as an endjoin character for HTML tagged text. These characters are indexed.

In the BASIC LEXER preference, ConText defines a default set of characters for each group.

The way you query on tokens that contain these characters depends on how ConText indexes the tokens containing these characters. This is because ConText tokenizes words at query time the same way it tokenizes words at index time. To query on words or numbers that contain special characters, you must know how these words are represented in the index.

See Also: For more information about defining special characters for the BASIC LEXER preference, see *Oracle8 Context Cartridge Administrator's Guide*.

Querying with Punctuation and Continuation Characters

Punctuation and continuation characters are not indexed with the words they occur next to or with, and thus are ignored by ConText at query time. The following table shows how ConText strips punctuation characters at query time:

Query	Equivalent Query
'John swims fast. Sharks eat.'	'John swims fast sharks eat'
'John swims. Fast sharks eat.'	'John swims fast sharks eat'
'{John swims, fast sharks eat}'	'John swims fast sharks eat'
'{SHAZAM!}'	'SHAZAM'
'{\$250}'	'250'
'{#101}'	'101'
'{phone#}'	'phone'

Suggestion: Because ConText strips punctuation characters at query time, leaving them out of the query expression and using the equivalent query might be a better approach, especially when the characters are reserved as in the last five examples.

Querying with Printjoins and Skipjoins

Printjoins and skipjoins are characters such as hyphens that join words together.

When you define a character as a printjoin, such as a hyphen, you specify that the words on either side of the hyphen are to be indexed with the hyphen. For example, *sister-in-law* is indexed as the token *sister-in-law*.

When you define a character as a skipjoin, such as a hyphen, you specify that the two words on either side of the hyphen are to be indexed as one token without the hyphen. For example, *sister-in-law* is indexed as *sisterinlaw*.

To query on words that contain a join character, you must know if the character is defined as a skipjoin or printjoin in the BASIC LEXER preference.

For example, if the hyphen character is defined as a printjoin, you must write your query with the hyphen, since the indexed token contains the hyphen. Thus, to query on all the documents that contain the term *sister-in-law*, you must write your query as follows with the hyphen:

```
'{sister-in-law}'
```

Note: The '-' character must be escaped, or else ConText interprets it as the MINUS operator.

However, if the hyphen character is defined as a skipjoin, you must write your query without the hyphen. Thus, to query on all documents that contain *sister-in-law*, you must write your query as:

```
'sisterinlaw'
```

This query really returns all documents that contain *sisterinlaw* and *sister-in-law*, provided the hyphen is defined as a skipjoin.

Querying with Numjoins and Numgroups

Numjoin and numgroup characters are characters that can appear in numbers, such as the decimal point and the comma.

Numjoin

A numjoin is a character that occurs once in a string of digits, such as a decimal point, and gets indexed with the number. (ConText defines the decimal as a default numjoin character for the BASIC LEXER preference.) For example, the number *3.14* is indexed as *3.14*. Thus to query on *3.14* with the decimal point defined as a numjoin character, you write:

```
'3.14'
```

When you define the numjoin character to be NULL, Context indexes *3.14* as the two separate numbers 3 and 14.

Note: When a period follows a number such as at the end of a sentence, ConText knows to index the number without the decimal point. For example, the number fourteen in the following sentence gets indexed as *14* without the period:

The score was San Francisco 21, Dallas 14.

Numgroup

A numgroup is a character such as a comma that groups digits together in a number. Numgroup characters get indexed with the number. (ConText defines the comma as a default numgroup character for the BASIC LEXER preference.) For example, the number *6,344,555* gets indexed as *6,344,555*.

To query on a number that contains numgroup characters, you must write the query with the numgroup character. For example, to query on *6,344,555*, you write:

```
'{6,344,555}'
```

Note that the comma must be escaped

Note: When you have the comma defined as a numgroup character, you must query on numbers using the comma. That is, a query on *{1,000}* does not return documents that contain *1000* without the comma. A better query is with the equivalence operator:

```
'{1,000}=1000'
```

When you define the numgroup character as NULL, numbers such as 1,000 get indexed as *1* and *000*.

Querying with Startjoin and Endjoin Characters

Startjoin and endjoin characters are non-alphanumeric characters that start and end tokens. These characters are indexed with the token they occur with.

You or your ConText administrator typically define startjoin and endjoin characters when you index tagged text such as HTML. This makes it easy to define sections for section searching as well as to query on the tags themselves.

For example, to query on the tag *<HEAD>* with *<* defined as a startjoin and *>* defined as an endjoin, write your query as follows:

```
'{<HEAD>}'
```

In the query above, an escape sequence is necessary, since *>* is an operator.

See Also: For more information about section searching, see “WITHIN Operator” in this chapter.

Theme Queries

This chapter describes how to perform theme queries. The following topics are covered:

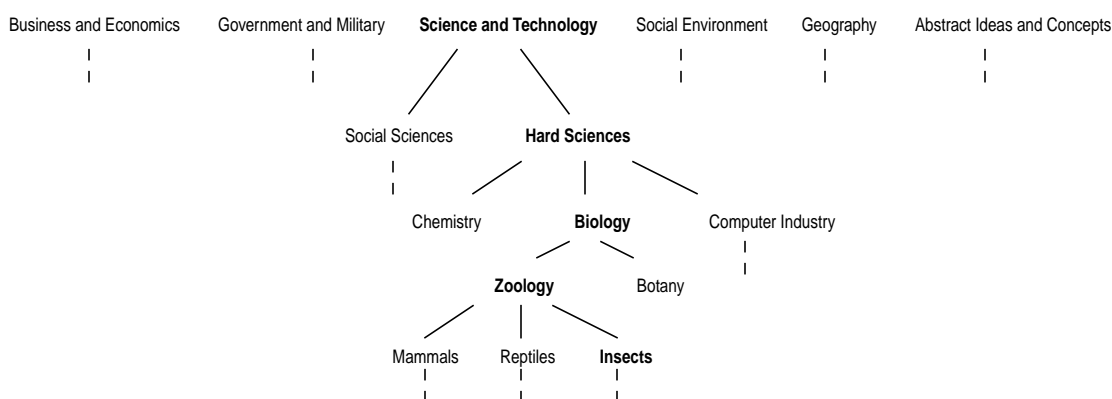
- Understanding Theme Queries
- Constructing Theme Queries
- Refining Theme Queries
- Theme Query Examples

Understanding Theme Queries

Theme queries enable you to search for documents by their major concepts. The following sections illustrate the theme indexing and querying processes and how they use the knowledge catalog.

The Knowledge Catalog

Figure 4–1



A theme query is usually a word or phrase that captures the concept for which you are searching. To better understand how to select the word or phrase that represents your idea, you must have a sense of how concepts and categories are organized in the knowledge catalog, the information store Context uses to derive themes during indexing and querying.

Tree-Structure and Categories

The knowledge catalog is a tree-like structure whose branches break down various realms of discourse. The knowledge catalog is divided into the following six main categories as shown in Figure 4–1:

- Business and Economics
- Government and Military

- Science and Technology
- Social Environment
- Geography
- Abstract Ideas and Concepts

These categories are divided further into more specific categories and concepts. Categories are defined as classifications of related nouns and ideas that can be subdivided into further categories and concepts.

Children categories are related to parent categories by an "is-associated-with" relationship, loosely defined as such to cover other standard child-parent type relationships such as "is-a-part-of", "belongs-to", or "is-a".

Figure 4-1 illustrates the basic structure of the knowledge catalog, showing a break down of an example branch within the top-level category of *science and technology*. In the example branch, the concept of *insects* belongs to the category of *zoology*, which is a part of the more general category of *biology*, which is part of the even more general category of *science and technology*.

The organization of the knowledge catalog has the following implications for theme indexing and querying:

- During indexing, the branches of the knowledge are the categories to which theme vectors attach; that is, a theme vector is essentially a branch of the knowledge catalog.
- Since concepts and categories are organized from general to specific, specifying broad themes in a query usually returns more documents than specifying more specific themes.

Concrete and Abstract Concepts

Concepts are leaf nodes in the knowledge catalog and can be associated with any level of category. Concepts can be either concrete or abstract.

Concrete concepts are ideas founded in the real world, usually described by nouns or noun phrases. Examples of concrete concepts are *jazz music* and *football*.

Abstract concepts are ideas such as *happiness* or *success*, usually described by abstract nouns. For example, if a news article was about the success of a famous football player and the article used words and phrases that described success, Context might attach a theme of *success* to the document. Likewise, a news article describing a Christmas celebration might have a theme of *happiness* attached to it.

Note: Do not confuse with literary interpretation the way in which ConText attaches abstract concepts to documents. ConText does not do literary interpretation. For example, do not expect ConText to attach a theme of *happiness* to a short story that had a happy mood or tone in the literary sense.

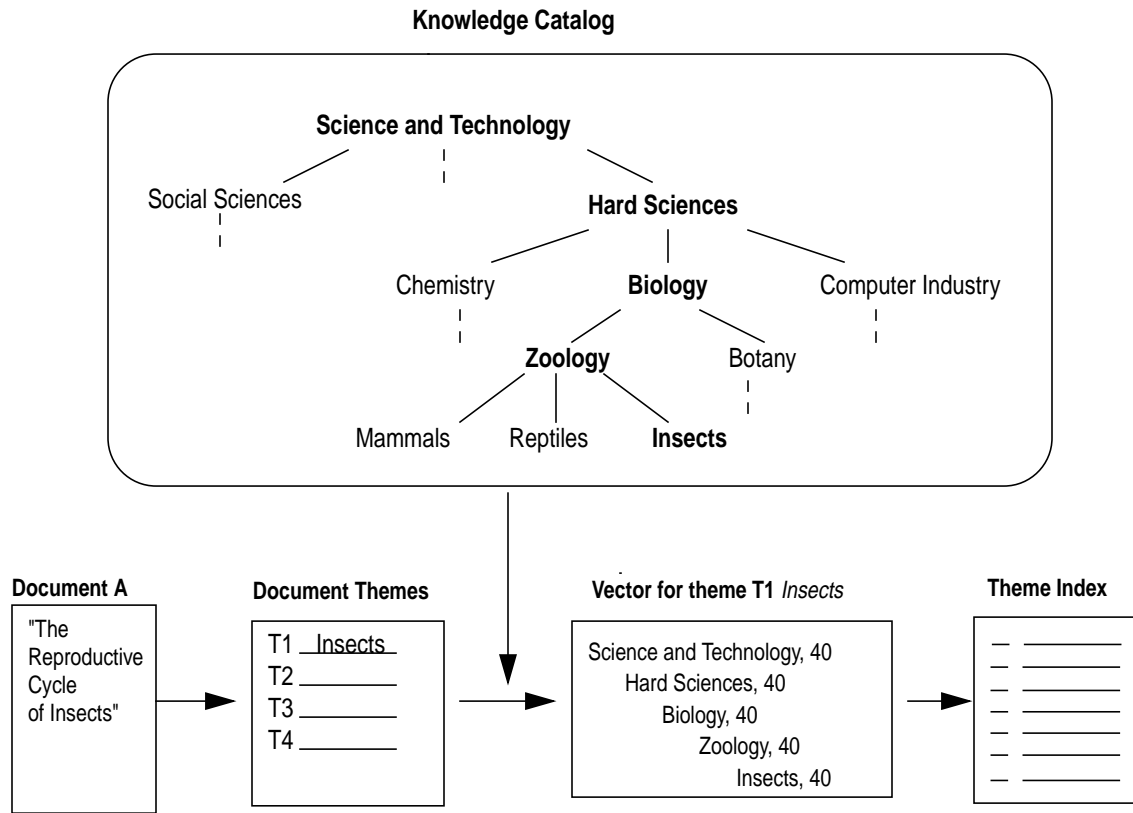
Normal Forms

When analyzing documents for theme querying and theme indexing, ConText must convert words and phrases you enter to their normal forms so they can attach into the knowledge hierarchy. To make this conversion, the knowledge catalog keeps the following lists:

Type of List	Description
Standard Noun Forms	A list of mappings from inflected variations of words to their standard noun forms as stored in the knowledge catalog's hierarchy of concepts. For example, the words <i>notify</i> and <i>notifies</i> are mapped to the normal form <i>notification</i> ; likewise, the words <i>summarize</i> and <i>summarizes</i> are mapped to the normal form <i>summaries</i> .
Alternate Forms	A list of mappings from acronyms, abbreviations, and alternate spellings to their standard forms. For example, <i>IBM</i> is a acronym for the standard form <i>IBM - International Business Machines Corporation</i>

Theme Indexing

Figure 4–2



Before you can issue a theme query, your set of documents must be indexed by theme. During theme indexing, ConText extracts up to sixteen main concepts or themes of a document. A theme can be a concrete concept, such as *insects*, or abstract concept, such as *success*, sufficiently developed in the document.

When indexing a document by theme, ConText attempts to classify document concepts using the knowledge catalog. Some concepts in a document might not have representation in the knowledge catalog; other themes might be inherently ambigu-

ous terms that ConText cannot place in the knowledge catalog. Hence, ConText recognizes the following types of themes:

- document themes that can attach to a branch of the knowledge catalog.
- document themes that have no representation in knowledge catalog. These can be themes unknown to the knowledge catalog or themes that are inherently ambiguous.

See Also: For more information about how to create a theme index, see *Oracle8 Context Cartridge Administrator's Guide*.

Known Themes

ConText creates theme vectors for every theme that can attach into the knowledge catalog. A theme vector is the branch of the knowledge catalog to which the concept attaches. Every level in the theme vector is weighted equally in the index. Refer to Figure 4–2.

In the example in Figure 4–2, the hypothetical document A entitled "The Reproductive Cycle of Insects" contains information about *insects*. The document theme vector T1 has five levels corresponding to the branch of the knowledge catalog, *science and technology*, *hard sciences*, *biology*, *zoology*, and *insects*. Every level of the branch gets entered as a searchable row in the theme index.

Unknown Themes

Themes that cannot attach into the knowledge catalog are indexed as a single row. For example, if ConText determined that a document was about a person *John Smith*, and *John Smith* was not known in the knowledge catalog, ConText indexes this name as a single row theme vector.

Ambiguous document themes such as the term *cricket* or the term *table* also have no representation in the knowledge catalog and hence are indexed as a single row. To query on such document themes, you would rely on other supporting themes such as *sports* or *insects* being indexed with an ambiguous theme like *cricket*.

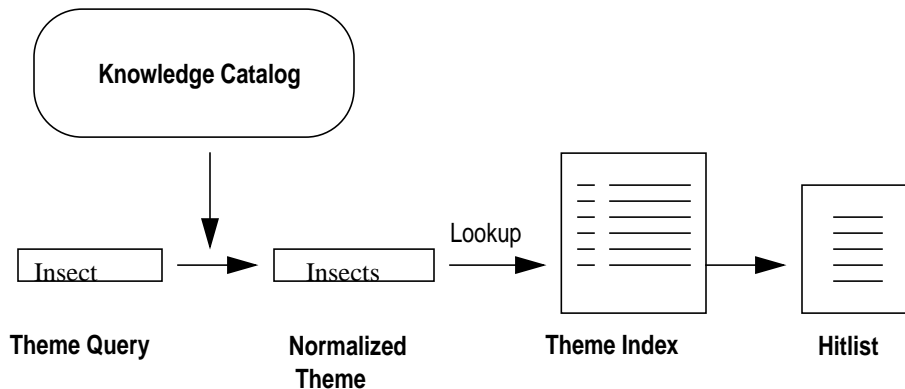
See Also: For more information about querying ambiguous themes, see "Refining Theme Queries" in this chapter.

Theme Weight

The theme weight is a measure of the strength of a theme relative to the other themes in a document. Weights are associated with theme vectors, and thus every level within a theme vector has the same weight. For example in Figure 4–2, every level in theme vector T1 has a weight of 40.

Theme Querying

Figure 4–3



To execute a theme query, you specify a query string, which can be a sentence or a phrase with or without operators. ConText uses the knowledge catalog to normalize the word or phrase you enter into a standard form. It then looks up the normalized theme in the index and returns the documents that were indexed with the given theme. See Figure 4–3. Scores are calculated based on the weights associated with each theme in the index.

In the example above, a theme query on either *science and technology*, *hard sciences*, *biology*, *zoology*, or *insects* will retrieve the document indexed in Figure 4–2 entitled, "The Reproductive Cycle of Insects".

Note: When you issue a theme query, you are asking ConText to return to you all the documents that ConText indexed with that theme. For ConText to attach a theme to a document, the idea or concept must be developed sufficiently in the document. If a concept is not developed sufficiently in a document, ConText does not index it as a document theme, and consequently the document is not returned in a query for that theme.

Scoring

ConText returns a relevance score for each document it returns in a theme query; the higher the score, the more relevant the returned document. This relevance score is out of 100 and is based on the weight of the indexed theme.

Generally, specifying broader themes or concepts in a theme query will return higher scoring documents.

When using operators in theme queries, the scoring behavior is the same as for regular text queries. For example, the OR operator returns the higher score of its operand, and the AND operator returns the lower score of its operands.

Constructing Theme Queries

Using Operators

With theme queries, the following operators have the same semantics as with regular text queries:

Operator	Symbol
Accumulate	,
Or	
And	&
Minus	-
Not	~
Weight	*
Threshold	>
Max	:

Examples

Some valid theme query strings using operators are as follows:

```
contains(text, 'cricket ~ insects') > 0;
contains(text, 'cricket & sports') > 0;
contains(text, 'music, reggae*5') > 0;
contains(text, 'chemistry > 30') > 0;
contains(text, 'soccer | basketball') > 0;
contains(text, 'computer software - Microsoft') > 0;
contains(text, 'music:20') > 0;
```

See Also: For more information about how to use operators in theme queries, see “Refining Theme Queries” in this chapter.

For more information about the semantics of query operators, see Chapter 3, “Understanding Query Expressions”.

Thesaurus Operators

In a theme query, the thesaurus operators (synonym, broader term, narrower term etc.) work the same way as in a regular text query, provided a thesaurus has been created/loaded.

Grouping Characters

In theme query expressions, the grouping characters () [] have the same semantics as with a regular text query.

Wildcard Characters

In theme query expressions, the wildcard characters% _ work the same way as in regular text queries.

Note: There is a risk of ambiguity when using the wildcard character. For example, doing a theme query on %court% might return documents that have a theme of *court of law* or *tennis court*.

Unsupported Operators

ConText does not support the following query expression operators with theme queries:

Operator	Symbol
Near	;
Fuzzy	?
Soundex	!
Stem	\$

Phrasing Theme Queries

Use Noun Forms

When you enter your theme query, ConText normalizes the word or phrase representing your theme into a form that it can use to compare with document themes in the index. This normal form is nouns and noun phrases, such as *chemistry* or *personal computer*. It is therefore better to use nouns and noun phrases when constructing theme queries. Avoid using sentences or long phrases.

For example, to search for documents about *computer programming*, use the noun form *computer programming* not *programming my computer*.

Avoid Splitting Phrases

Avoid splitting phrases that describe your idea as a whole. For example, use the phrase *physical chemistry*, not *physical and chemistry*.

Understand Case-sensitivity

Unlike regular text queries, theme queries are case-sensitive. For example, doing a query on the common noun *turkey*, which describes a type of bird, will not produce a hit on the proper noun *Turkey*, which describes a country.

Refining Theme Queries

Depending on how you write your theme query, ConText usually returns documents that are relevant to your query as well as documents that might be irrelevant to your query. Before you issue the query, you do not know what combination of document themes your query will return.

For example, a query on *cricket* might return documents on *sports* and *insects* depending on your document set. The best way to know the possible outcome is to run the query and examine the set of returned documents. Then you run the query again, using logical operators to eliminate unwanted documents.

You can approach the trial and error method in one of two ways:

- **Restrict query.** You select a broad category/concept, examine results, and then issue the query again using the AND or NOT operator to further restrict the query hitlist.
- **Expand query.** You select a specific category, examine the results, then expand query to include more documents in the hitlist.

Restricting a Query

Starting with broad theme queries might generate noise or unwanted documents. This is because of the following:

- the word or phrase in your query can represent more than one concept
- a document can have more than one theme attached to it

You can use the AND or NOT operator to eliminate unwanted documents. However, use these operators with caution, because in both cases you run the risk of eliminating documents that you might be interested in. For this reason, it is always better to have some noise than none at all.

Using AND

You can use the AND operator with a qualifying theme to restrict your theme query and hence eliminate noise.

For example, if a theme query on cricket always returned documents about the sport *cricket* and the insect *cricket*, and you were interested only in those documents about cricket the sport, you can restrict your query by qualifying cricket with the more general category *sports* as follows:

```
'cricket and sports'
```

The disadvantage of using AND with a restricting theme is that a successful query depends on both themes developed sufficiently in the document for ConText to index them as such. For example, a hypothetical news article about the personal affairs of cricket player might not have the theme of sports developed substantially for ConText to index it as a theme, and therefore such a document would not be returned in the above query.

Suggestion: When choosing the restricting condition to use with the AND operator, we recommend choosing a broad category; choosing a very specific category as the restricting condition might inadvertently eliminate relevant documents.

Using NOT

You can use the NOT operator to exclude unwanted themes. For example, suppose you have a collection of news articles. You find that a theme query on cricket returns documents about *cricket* the sport as well as *cricket* the insect.

In such a scenario, you can use the not operator to exclude the unwanted theme. Thus if you are interested in those documents only about the sport cricket, you exclude documents about insects as follows:

```
'cricket not insects'
```

One disadvantage of using the *not* operator is that you run the risk of excluding documents that are coincidentally about the desired theme and the unwanted theme. For example, the above query does not return a hypothetical document about a cricket game that was swarmed by locusts, assuming that the theme of *insects* is developed sufficiently for ConText to index *insects* as a document theme.

Another disadvantage of using NOT is that you usually have a better idea of the themes you want, not of the themes you don't want. Predicting unwanted themes depends on knowing your document corpus. For this reason, using NOT is best suited for eliminating irrelevant high-ranking documents you specifically know about.

Expanding a Query

Sometimes it is better to start with specific categories and then expand these queries into more general ones, especially when your query covers a topic that is categorized specifically in the world. For example, if you are searching for documents that are about *bees*, you issue a query on *bees*, which is a specific category of insects.

If you find that the result set is not returning the documents you need, you can expand the query by issuing a theme of *insects*, which is slightly broader.

After expanding a query, you can use the NOT or AND operators to scale back the query.

Theme Query Examples

To execute a theme query, you specify a query string, which can be a sentence or a phrase with or without operators. ConText interprets your query, creating a normalized form of your query that it can use to match against document themes in the index. Context returns a list of documents that satisfy the query, based on certain rules, along with a score of how relevant each document is to the query.

You can issue themes queries using either the two-step or one-step method. The way in which ConText matches themes and scores hits is the same for both methods.

Two-Step Query

To execute a theme query with the CTX_QUERY.CONTAINS procedure, you must specify a policy that has a theme lexer associated with it.

For example, you specify a theme query on *computer software* as follows:

```
execute ctx_query.contains('THEME_POL', 'computer software', 'CTX_TEMP');
```

In the above example, ConText normalizes *computer software*, and then attempts to match the normal form with document themes in the index.

When a match is found, ConText uses the weight of the matched theme to compute a score that reflects how relevant the match is to the query; the higher the score, the more relevant the hit. ConText returns the matched document as part of the hitlist.

One-Step Query

You can execute theme queries in SQL*Plus using the one-step method. To do so, the text column must have a theme policy attached to it. The way in which ConText matches themes and scores hits is the same as in a two-step query.

For example, to execute a theme query on *computer software*:

```
SELECT * FROM TEXTAB  
WHERE CONTAINS (text, 'computer software') > 0
```

Multiple Policies

For a text column that has more than one policy associated with it, you must specify which policy to use in the CONTAINS clause using the *pol_hint* parameter. You might create two policies for a column when you want to perform both theme and text queries on the column.

For example, if the column *text* had a regular text policy and a theme policy *THEME_POL* associated with it, you issue a theme query as follows:

```
SELECT ID, SCORE(0) FROM TEXTAB  
WHERE CONTAINS (text, 'computer software', 0, 'THEME_POL') > 0
```

Since the *pol_hint* parameter is last, when you need to specify a policy in the *CONTAINS* function as in this example, you must also specify a placeholder, in this case 0, for the *LABEL* parameter.

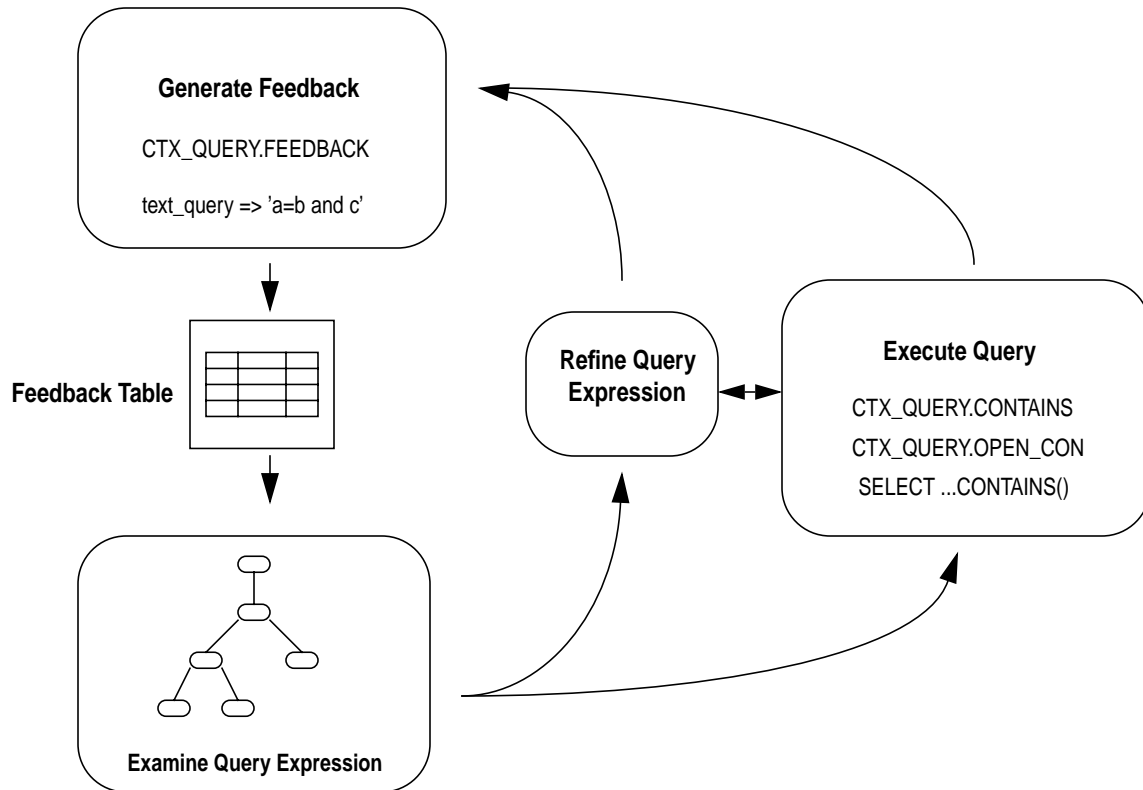
See Also: For more information about using the *pol_hint* parameter in the *CONTAINS* function, see the specification for *CONTAINS* in Chapter 9.

Query Expression Feedback

This chapter describes query expression feedback. The following topics are covered:

- The Feedback Process
- Understanding ConText Parse Trees
- Understanding the Feedback Table
- Obtaining Query Expression Feedback

The Feedback Process



Query expression feedback is a feature that enables you to know how ConText parses a text or theme query expression *before* you execute the query. Knowing how ConText evaluates a text or theme query expression is useful for refining and debugging queries. You can also design your application so that it uses the feedback information to help users write better queries.

The diagram above shows how you use query expression feedback. You execute the PL/SQL procedure `CTX_QUERY.FEEDBACK`, which generates and stores feedback information to a table. From the data in this feedback table, you can visualize the ConText parse tree to examine how the expression was expanded and parsed.

You can then refine the query and re-execute FEEDBACK, or you can execute the real query with CONTAINS for two-step queries, OPEN_CON for in-memory queries, or SELECT for one-step queries.

In text queries, query expression feedback is especially useful for knowing how context expands expressions that contain stem, wildcard, thesaurus, fuzzy, soundex, PL/SQL, or SQE operators before you execute the query. This is because such queries can potentially expand into many tokens or result in very large hitlists, causing much overhead.

In theme queries, query expression feedback is useful for knowing how ConText uses the knowledge catalog to normalize query expressions.

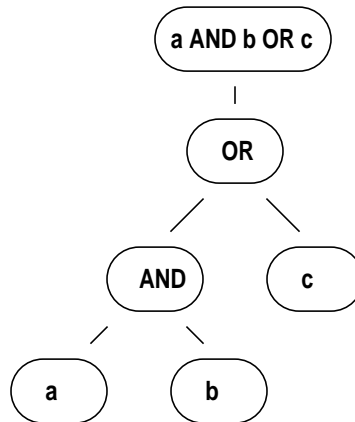
Understanding ConText Parse Trees

Before ConText executes a query, it parses the expression. The resulting expression can be represented as a parse tree. A ConText parse tree can show:

- order of execution (precedence of operators)
- stem, fuzzy, thesaurus, soundex, PL/SQL, SQE, and wildcard expansions
- theme query normalization
- query optimization
- stop-word transformations

The output table of the FEEDBACK procedure is graphical representation of a ConText parse tree.

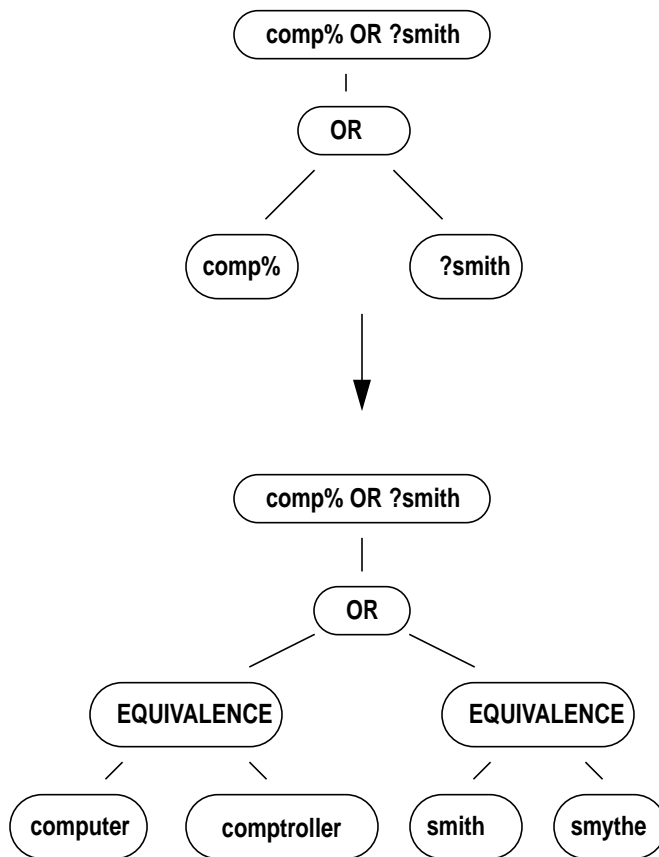
Operator Precedence



Parse trees are read in a depth-first manner and from left to right. This means the first operation is always furthest to the left and at the bottom of the branch. In this way, parse trees illustrate operator precedence.

The example above shows the parse tree for the evaluation of *a AND b OR c*, where *a*, *b* and *c* stand for three arbitrary words. Since the *and* operation *a AND b* is the left-most operation and at the bottom of the tree, it is executed first. In this way, the parse tree above indicates correctly that the *and* operator has higher precedence over the *or* operator. The resulting query is hence *(a AND b) OR c* rather than *a AND (b OR c)*.

Query Expansions

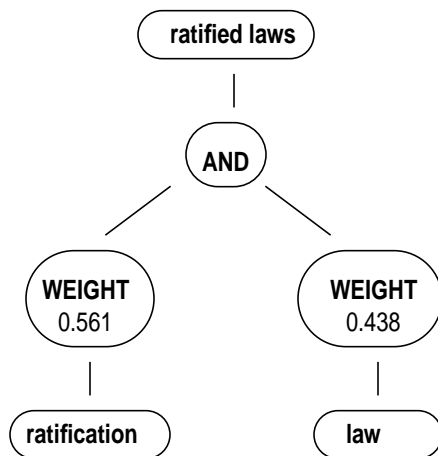


The above example shows how ConText expands the query `comp% OR ?smith`. The parse tree shows that before ConText executes the query, the token `comp%` is expanded to `computer` and `comptroller`, while `?smith` is expanded to `smith` and `smythe`.

ConText parse trees show similar expansions with thesaurus, wildcard, soundex, stem, SQE, and PL/SQL operators. In the case of the wildcard, soundex, and fuzzy operators, ConText obtains the correct word expansions from the index.

Note: When you include the SQE operator in the feedback expression, the feedback (expansion of the stored query expression) is based on the current state of the index and will take into account any inserts, updates, or deletes made to the base table; however, unlike a call to CONTAINS, the stored query expression is *not* updated or refreshed as a result of the call to FEEDBACK.

Theme Query Normalization



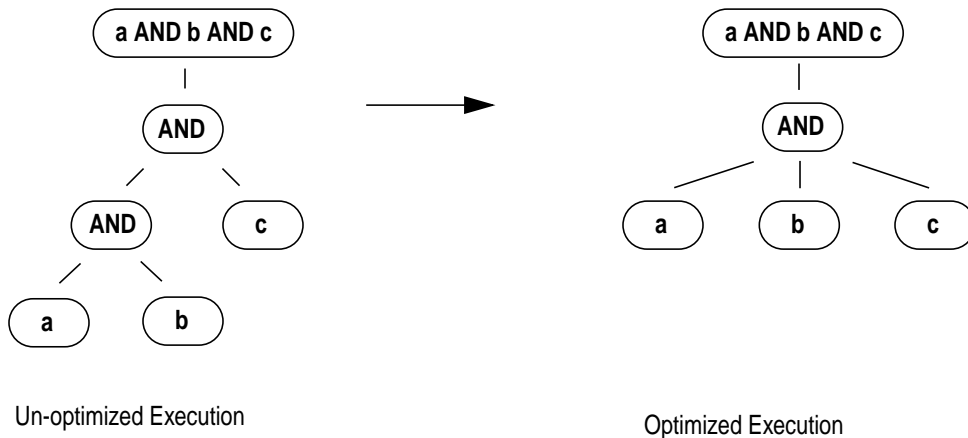
You can use query expression feedback to know how ConText interprets theme queries. The feedback information provides the normalized version of the query as obtained from the knowledge catalog.

The example above shows how ConText normalizes the theme query *ratified laws* to the themes *ratification* and *law*. The resulting expression is an AND operation with weights attached to the normal forms: *ratification*0.561 AND law*0.438*.

Note: Because numbers are rounded off when displayed, weights might not always add up to 1.000 exactly.

See Also: For more information about theme queries, see Chapter 4, “Theme Queries”.

Query Optimization

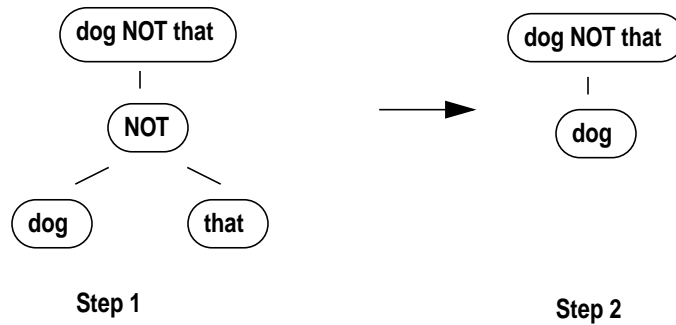


The example above shows how ConText optimizes the expression `a AND b AND c`, where `a` and `b` and `c` stand for three different words.

In the first step of the parse, ConText evaluates `a AND b`, then ANDs the result with `c`. With such a parse tree, ConText must search for all documents that contain `a` and `b`, then search for all documents that contain `c`, and then intersect the two result sets.

The ConText optimizer realizes this query is more efficiently executed by simultaneously searching for all the documents that contain `a` and `b` and `c`, which is illustrated in the second step of the optimizing process.

Stopword Rewrite



The example above shows the parse sequence for the stopwords transformation:

non_stopword NOT stopwords => non_stopword

Assuming *that* is a stopwords, ConText reduces the query *dog NOT that* to *dog*.

See Also: To learn more about querying with stopwords, see “Querying with Stopwords” in Chapter 3.

For a list of all possible stopwords transformations, see Appendix C, “Stopword Transformations”.

Understanding the Feedback Table

Before you issue a query, you can obtain the parse tree information for the query expression. The procedure `CTX_QUERY.FEEDBACK` creates a graphical representation of the parse tree and stores this information in a feedback table, which you create before executing `CTX_QUERY.FEEDBACK`. To reconstruct ConText parse trees, you must understand the structure of this table.

Table Structure

The feedback table has the following structure:

Table 5–1

Column Name	Datatype	Description
FEEDBACK_ID	VARCHAR2(30)	The value of the <i>feedback_id</i> argument specified in the FEEDBACK call.
ID	NUMBER	A number assigned to each node in the query execution tree. The root operation node has ID =1. The nodes are numbered in a top-down, left-first manner as they appear in the parse tree.
PARENT_ID	NUMBER	The ID of the execution step that operates on the output of the ID step. Graphically, this is the parent node in the query execution tree. The root operation node (ID =1) has PARENT_ID = 0.
OPERATION	VARCHAR2(30)	Name of the internal operation performed. Refer to Table 5–2 for possible values.
OPTIONS	VARCHAR2(30)	Characters that describe a variation on the operation described in the OPERATION column. When an OPERATION has more than one OPTIONS associated with it, OPTIONS values are concatenated in the order of processing. See Table 5–3 for possible values.
OBJECT_NAME	VARCHAR2(64)	Section name, or wildcard term, or term to lookup in the index.
POSITION	NUMBER	The order of processing for nodes that all have the same PARENT_ID. The positions are numbered in ascending order starting at 1.
CARDINALITY	NUMBER	Reserved for future use. You should create this column for forward compatibility.

OPERATION Column

Table 5–2 lists the possible values for the OPERATION column in the feedback table:

Table 5–2

Operation Value	Query Operator	Equivalent Symbol
ACCUMULATE	ACCUM	,
AND	AND	&
EQUIVALENCE	EQUIV	=
FIRST_NEXT_DOC	#	#
MAX_DOC	:	:
MINUS	MINUS	-
NEAR	NEAR	;
NOT	NOT	~
NO_HITS	(no hits will result from this query)	
OR	OR	
PHRASE	(a phrase term)	
SECTION	(section)	
THRESHOLD	>	>
WEIGHT	*	*
WITHIN	within	(none)
WORD	(a single term)	

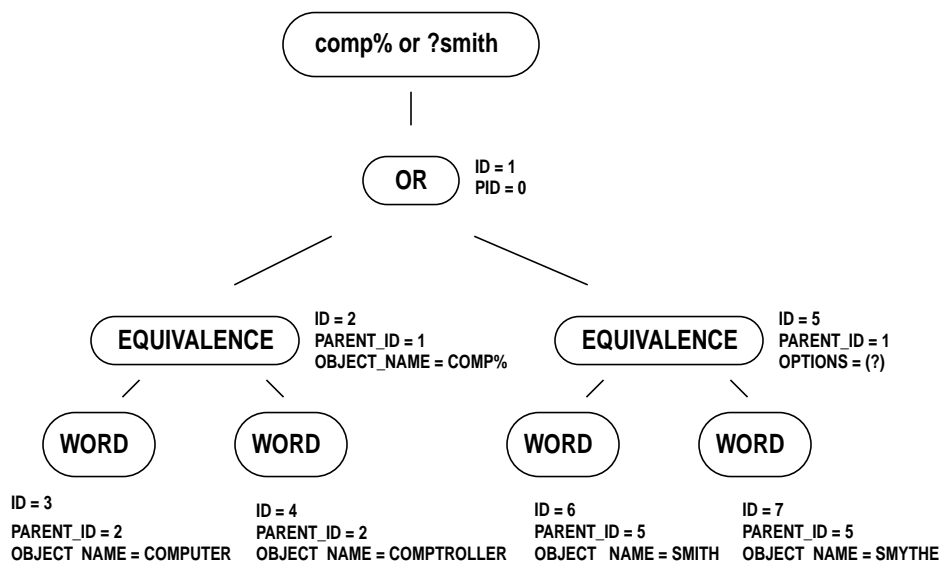
OPTIONS Column

Table 5–3 shows the values for the OPTIONS column in the feedback table. When an OPERATION has more than one OPTIONS associated with it, the OPTIONS values are concatenated in the order of processing.

Table 5–3

Options Value	Description
(\$)	Stem
(?)	Fuzzy
(!)	Soundex
(n)	A number associated with threshold, weight, or max
(m-n)	First next range (m and n are integers)

Example



The figure above shows how ConText encodes the parse tree for the query *comp% OR \$smith*, which is asking for all documents that contain words beginning with *comp* or contain words that are spelled like *smith*.

Each node is labeled with a value that corresponds to the OPERATION column in the feedback table. The tree above contains one OR node, two EQUIVALENCE nodes, and four WORD nodes.

The ID and PARENT_ID values are listed beside each node. For example, the OR node has an ID of 1 and PARENT_ID of 0, since it is the root node.

The EQUIVALENCE node with ID = 2, PARENT_ID = 1, has an OBJECT_NAME value of *COMP%*, because this equivalence operation is a result of wildcard term *comp%*.

The WORD node with *id* = 3 has an OBJECT_NAME value of *computer*, because in this instance, *computer* is one of the words that satisfy *comp%*.

Obtaining Query Expression Feedback

To obtain query expression feedback information, you must do the following:

1. Create the feedback table.
2. Execute CTX_QUERY.FEEDBACK.
3. Retrieve data from feedback table.
4. Optionally, construct expansion tree from table information.

Creating the Feedback Table

To create a feedback table called *test_feedback* for example, use the following SQL statement:

```
create table test_feedback(
    feedback_id varchar2(30)
    id number,
    parent_id number,
    operation varchar2(30),
    options varchar2(30),
    object_name varchar2(64),
    position number,
    cardinality number);
```

Executing CTX_QUERY.FEEDBACK

To obtain the expansion of a query expression such as *comp% OR ?smith*, use CTX_QUERY.FEEDBACK as follows:

```
ctx_query.feedback(
    policy_name => 'scott.test_policy',
    text_query => 'comp% OR ?smith',
    feedback_table => 'test_feedback',
    sharelevel => 0,
    feedback_id => 'Test');
```

Retrieving Data from Feedback Table

To read the feedback table, you can select the columns as follows:

```
select feedback_id, id, parent_id, operation, options, object_name, position
from test_feedback
order by id;
```

The output is ordered by ID to simulate a hierarchical query:

FEEDBACK_ID	ID	PARENT_ID	OPERATION	OPTIONS	OBJECT_NAME	POSITION
Test	1	0	OR	NULL	NULL	1
Test	2	1	EQUIVALENCE	NULL	COMP%	1
Test	3	2	WORD	NULL	COMPTROLLER	1
Test	4	2	WORD	NULL	COMPUTER	2
Test	5	1	EQUIVALENCE	(?)	SMITH	2
Test	6	5	WORD	NULL	SMITH	1
Test	7	5	WORD	NULL	SMYTHE	2

Constructing the Parse Tree

You can optionally construct an approximate graphical representation of the parse tree using a hierarchical query. This type of query outputs rows in a hierarchical manner, where children nodes are indented under parent nodes.

The following statement selects from a populated feedback table, indenting the output according to level:

```
select lpad(' ',2*(level-1)) || operation operation, options, object_name,
position
from test_feedback
start with id = 1
connect by prior id = parent_id;
```

This statement produces hierarchical output for the query *comp% OR ?smith* as follows:

OPERATION	OPTIONS	OBJECT_NAME	POSITION
OR	NULL	NULL	1
EQUIVALENCE	NULL	COMP%	1
WORD	NULL	COMPTROLLER	1
WORD	NULL	COMPUTER	2
EQUIVALENCE	(?)	SMITH	2
WORD	NULL	SMITH	1
WORD	NULL	SMYTHE	2

Document Presentation

This chapter describes how ConText query applications can present documents with highlighted information.

The following topics are covered in this chapter:

- Overview of Document Presentation
- Using CTX_QUERY.HIGHLIGHT
- Creating Highlighted Text
- Document Presentation in Windows

Overview of Document Presentation

In a typical query application, users can issue text or theme queries. The application executes the query and returns to the user a hitlist, allowing the user to select one or more documents.

When the user chooses a document, ConText enables you to present the selected document with the query terms highlighted for text queries, or with the relevant paragraphs highlighted for theme queries.

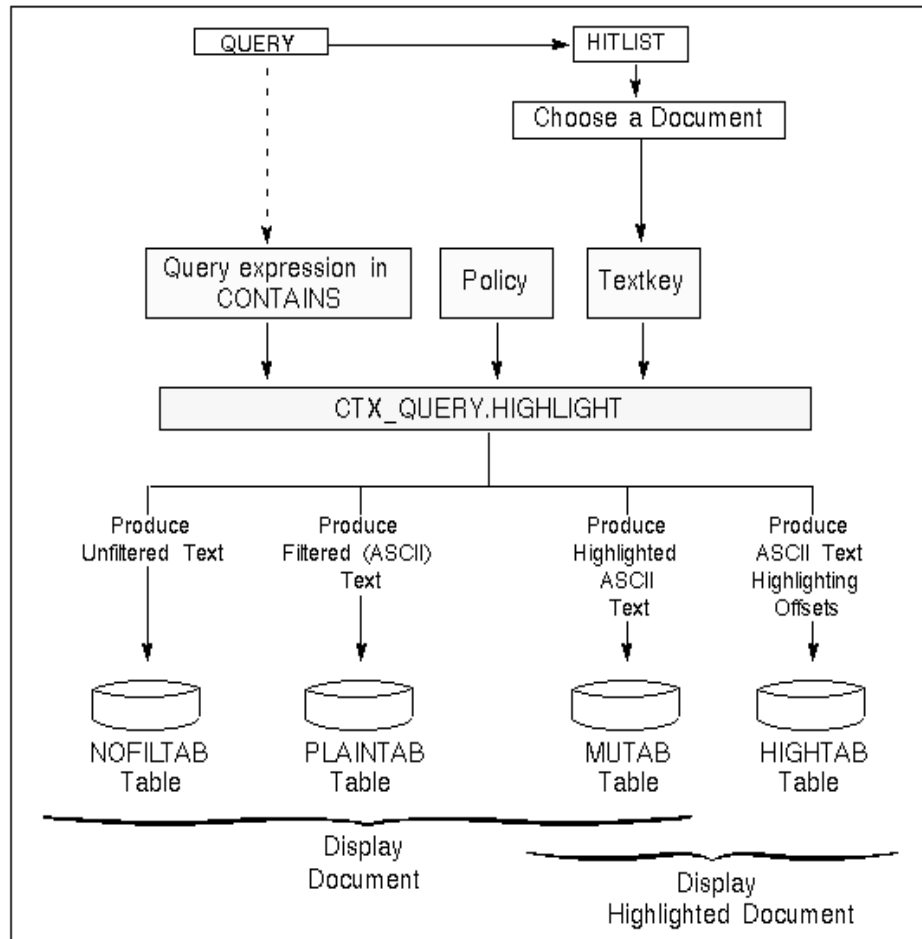
Your application can also present linguistic summaries of the selected documents.

See Also: For more information about linguistic output, see Chapter 7, “Linguistic Concepts”.

With ConText, you use the `CTX_QUERY.HIGHLIGHT` procedure to create various forms of highlighted output that can be presented to users.

This chapter describes how to present highlighted documents for applications built in PL/SQL as well as applications built in a Windows 32-bit client-side environment.

Using CTX_QUERY.HIGHLIGHT



The PL/SQL procedure `CTX_QUERY.HIGHLIGHT` generates filtered text, marked-up highlight text, and highlight information. You typically call `CTX_QUERY.HIGHLIGHT` after executing a text or theme query.

With text queries, `HIGHLIGHT` marks the relevant words or phrases in the document.

With theme queries, `HIGHLIGHT` marks the relevant paragraphs in the document.

Note: ConText does not do sentence-level theme highlighting.

Use CTX_QUERY.HIGHLIGHT to generate the following output for a document:

- an unfiltered version of the original document (stored in a NOFILTAB table)
- a plain (ASCII) text version of the document (stored in a PLAINTAB table)
- a marked-up ASCII version of the document with occurrences of the specified word or paragraph highlighted (stored in a MUTAB table)

Note: If the document is an HTML document filtered through the internal HTML filter, the marked-up ASCII text version generated by HIGHLIGHT and stored in a MUTAB table retains the original HTML tags from the document.

- highlight information that identifies the position and length of the query terms or paragraphs found in the source document (stored in a HIGHTAB table)

The positions and lengths of the query terms are specified as offsets from the beginning of the ASCII text version of the document.

- A fifth type of output, ICF, is generated automatically by HIGHLIGHT when a document in one of the supported formats is viewed in the Windows 32-bit viewer.

See Also: For more information about the structure of the highlight output tables, see “Highlight Table Structures” in Appendix A.

For more information about the Windows 32-bit viewer, see “Document Presentation in Windows” in this chapter.

Highlighting Mark-up

When you call CTX_QUERY.HIGHLIGHT, you can specify the markup used to indicate the start and end of a highlighted word or phrase for text queries, or the start and end of a highlighted paragraph for theme queries.

When you specify no markup, HIGHLIGHT uses default markup. The default highlighting mark-up produced by HIGHLIGHT differs depending on the format of the source document.

If the source document is an ASCII document or a formatted document, the default highlighting markup is three angle brackets immediately to the left (<<<) and right (>>>) of each term.

If the source document is an HTML document filtered through an external filter, the default highlighting markup is the same as the highlighting markup for ASCII or formatted documents (<<< and >>>).

If the source document is an HTML document filtered through the internal HTML filter, the default highlighting markup is the HTML tags used to indicate the start and end of a font change:

- to the immediate left of the term
- to the immediate right of the term

See Also: For more information about internal and external filters, see *Oracle8 ConText Cartridge Administrator's Guide*.

Creating Highlighted Text

To present highlighted documents in an application, do the following:

1. Allocate one or more highlight result tables to store the results.
2. Issue a query to obtain a list of documents.
3. Call the CTX_QUERY.HIGHLIGHT procedure for a document from the hitlist.
4. Display (or otherwise use) the output generated by HIGHLIGHT.
5. Release the result table(s).

Allocating Result Tables

The result tables required by the HIGHLIGHT procedure can be allocated manually using the CREATE TABLE command in SQL or using the CTX_QUERY.GETTAB procedure.

For example, to create a MUTAB table to store highlighted ascii mark-up, issue the following statement:

```
create table mu_ascii
(
  id number,
  document long
);
```

To create a HIGHTAB table to store highlight offset information, issue the following statement:

```
create table highlight_ascii
(
  id number,
  offset number,
  length number,
  strength number
);
```

See Also: For more information about the structure of the highlight output tables, see “Highlight Table Structures” in Appendix A.

Issuing a Query

Issue a one-step, two-step, or in-memory query to return a hitlist of documents. You can issue either a text or theme query. For text queries, you call CONTAINS with a text policy; for theme queries, you call CONTAINS with a theme policy. The hitlist provides the textkeys that are used to generate highlight and display output for specified documents in the hitlist.

Calling CTX_QUERY.HIGHLIGHT

Call CTX_QUERY.HIGHLIGHT with a pointer to a document (generally the textkey obtained from the hitlist) and a text or theme query expression.

CTX_QUERY.HIGHLIGHT returns various forms of the specified document that can be further processed or displayed by the application.

ConText uses the query expression specified in the HIGHLIGHT procedure to generate the highlight offset information and marked-up ASCII text. In addition, the offset information is based on the ASCII text version of the document.

Note: While the query expression is usually the same as the expression used to return documents in the text query, it is not required that the query expressions match. For example, you might allow a user to search for all articles by a particular author and then allow the user to view highlighted references to a specified subject in the returned documents.

Text Query Highlighting

To create highlight mark-up for text queries, you must specify a *text* policy, which is usually the policy you specify with the CONTAINS procedure for the same query.

For example, to highlight all the occurrences of the term *dog* with a document identified by textkey *14*, issue the following statement:

```
ctx_query.highlight
(
  cspec=> 'text_policy',
  textkey => '14',
  query => 'dog',
  id=> 14,
  hightab => 'highlight_ascii',
  mutab => 'mu_ascii'
);
```

Theme Query Highlighting

To create highlight mark-up for a theme query, you must specify a *theme* policy, which is usually the policy you specify with the CONTAINS procedure for the same query. With theme queries, the HIGHLIGHT procedure highlights the relevant paragraphs in the document.

For example, to highlight all the paragraphs that are relevant to the theme query *computers* for document with textkey *12*, issue the following query:

```
ctx_query.highlight
(
  cspec=> 'theme_policy',
  textkey => '12',
  query => 'computers',
  id=> 12,
  hightab => 'highlight_ascii',
  mutab   => 'mu_ascii'
);
```

Presenting HIGHLIGHT Output

You can use the MUTAB table to view highlighted ascii text. For example in SQL*Plus, you can issue the following statement to view a MUTAB table called *mu_ascii*:

```
select * from mu_ascii order by id;
```

You can also use the offset information in the HIGHTAB table to highlight the document in ways that suit your application.

Text Query Highlight Output

With text queries, the word or phrase is highlighted. For example, a text query on *dog* might produce the following type of highlighted ascii output for a document:

```
...
The quick brown <<dog>> jumped over the fox.
...
```

Theme Query Highlight Output

With theme queries, the relevant paragraphs in the document are highlighted. For example, a theme query of *computers* produces the following type of highlighted ascii output for a document:

<<< LAS VEGAS -- International Business Machines Corp. is using the huge computer trade show here this week to try to prove a much disputed marketing claim of the past year and a half: that its PS/2 line of personal computers really does offer unique benefits.>>>

In the battle for the hearts and minds of the 100,000 dealers, corporate customers and other spectators gathered here, IBM has set up a series of demonstrations of the Micro Channel, which is the PS/2's internal data pathway. The demonstrations seek to show that this pathway has extra flexibility that can translate into more speed. One demonstration uses an add-in circuit board that IBM claims allows data to be sent over a network about 60% faster. Another illustrates a quicker way to store the huge amounts of data handled by a so-called file server, the machine that controls a network of personal computers.

<<< While most personal computers contain just one "master" processor -- the chip that tells the various parts of the computer what to do -- the Micro Channel allows for more than one. That means that in Micro Channel machines, the workhorse central processor can dump lots of work onto another processor, freeing itself to go about other tasks.>>>

...

In this three paragraph excerpt of a news article that satisfies the theme query *computers*, ConText highlights (with angle brackets) only the paragraphs that are about computers.

Release Highlight Result Tables

After documents have been processed by the HIGHLIGHT procedure and displayed to the user, drop the highlight result tables.

If the tables were allocated using CTX_QUERY.GETTAB, you use CTX_QUERY.RELTAB to release the tables.

If the tables were created manually, drop the tables using the SQL command DROP TABLE.

Document Presentation in Windows

You can use the Oracle8 ConText Cartridge Viewer Control (CTXV32.OCX) to present highlighted documents to users in a Windows 32-bit environment, such as Windows NT or Windows 95. The viewer enables the user to browse documents in the supported formats with query terms highlighted.

Note: The viewer control is part of the Oracle8 ConText Cartridge Workbench, which is included in each Oracle8 ConText Cartridge distribution.

You embed the control in client-side applications. To operate the viewer, you need not write any PL/SQL code; given the database connection, the document textkey, and the query term, the viewer control displays the document with highlights.

The user can view a Word document, for example, as it would appear in Microsoft Word. The user can also scroll through the document using the Next and Previous buttons to jump to the next or previous occurrence of the search term(s).

Using the ConText Viewer Control

As OCX modules are not stand-alone executables, you need a development environment such as Visual C++ or Visual Basic to use the ConText Viewer Control. Within such an environment, you can add the control to the tool palette, from where you can place instances of the control on a form or canvas.

For example, in Visual Basic 4.0, you add the control to the tool palette by selecting Custom Controls from the Tool menu. Use the browser to select the Oracle8 ConText Viewer Control, CTXV32.OCX, from the *oracle_home*\BIN directory.

Alternatively, you can create instances of the control dynamically, using the identification string *CTXV32.CTXViewer.1*.

If the viewer control is embedded in an HTML page, the browser must support ActiveX components and the client machine must have the viewer installed on it with all required support files. The viewer uses SQL*Net to communicate with the database. Within HTML, you can invoke the methods using Visual Basic scripting, for example, and change properties with the OBJECT tag and parameter settings syntax.

See Also: For more information about the methods and properties associated with the 32-bit viewer control, see the ConText Viewer Control help file, CTXV32.HLP. This file has Visual Basic and HTML examples.

Supported Formats

You can use the ConText Viewer Control to view documents in the following server-side supported formats:

- ASCII
- Microsoft Word for Windows 2, 6.x
- WordPerfect for Windows 5.x, 6.x
- WordPerfect for DOS 5.0, 5.1, 6.0

Viewing Without the ConText Viewer Control

If you are not using the ConText viewer control to present documents in a 32-bit Windows environment, you can use the ConText I/O utility (CTXIO32) to move documents (highlighted or not) from database tables to the client operating system and vice-versa. Documents downloaded to the client operating system can be viewed in their native applications.

See Also: For more information about the 32-bit Windows I/O utility, see the *Oracle8 ConText Cartridge Workbench User's Guide*.

Linguistic Concepts

This chapter describes the approach used by ConText linguistics to provide advanced analysis of English-language text.

The following topics are covered in this chapter:

- Overview of ConText Linguistics
- Application Program Interface (API)
- Linguistic Core
- Linguistic Output
- Linguistic Settings

Overview of ConText Linguistics

ConText linguistics is used to analyze the content of English-language documents. You use ConText linguistics to create different views of the contents of documents that allow the user to quickly review the essential content of documents and determine their relevance.

Because these services are separate and distinct from text and theme indexing, you can incorporate linguistic analysis and functionality in a text application, independent of the text/theme indexing process.

ConText linguistics can generate the following forms of linguistic output for documents:

Output Type	Description
Themes	The main concepts of a document.
Gist	Paragraph or paragraphs in a document that best represent what the document is about as a whole.
Theme Summary	Paragraph or paragraphs in a document that best represent a given theme in the document.
Sentence-Level Gist	Sentence or sentences in a document that best represent the themes in the document as a whole.
Sentence-Level Theme Summary	Sentence or sentences in a document that best match a single theme in the document.

You obtain linguistic output by submitting a linguistic request using the CTX_LING PL/SQL package. Linguistic requests can only be processed by ConText servers running with the Linguistic personality.

Requirements

The requirements for using ConText linguistics are:

- text stored in a column (either directly or indirectly through a pathname to files)
- a policy for the column
- ConText server running with Linguistic personality

Note: The setup requirements of having text in a column and having a policy for the column apply to ConText indexes (text/theme) as well as ConText linguistics. The procedures for storing text and creating policies are not discussed in this manual. For more information about storing text in columns and creating policies for the columns, see *Oracle8 ConText Cartridge Administrator's Guide*.

Application Program Interface (API)

Linguistic and queue management functions are invoked by using PL/SQL procedures called or executed within the programming language in which the application is developed. If the application is developed in PL/SQL, these procedures may be invoked directly as PL/SQL execute statements. If the application is developed in another language, such as C, the PL/SQL procedures for linguistic and queue management functions are accessed through the Oracle Call Interface (OCI).

ConText provides the following PL/SQL packages for generating linguistic output and managing the Services Queue, respectively:

- CTX_LING
- CTX_SVC

CTX_LING Package

The stored procedures in CTX_LING are used to request linguistic output and submit the requests to the Services Queue. CTX_LING also provides procedures for specifying user settings for generating linguistic output and enabling logging of parse information generated during the processing of a request.

The model for submitting requests and querying the linguistic output is similar to the two-step query model (CONTAINS procedure) provided within the ConText framework for content-based text retrieval.

For example, to generate themes for a document, you first create a table to store the results of the theme generation, then call CTX_LING.REQUEST_THEMES procedure followed by the CTX_LING.SUBMIT function. ConText stores the results in a theme table. To view the results, issue a SELECT statement to select the theme from the output table.

See Also: For more information about the procedures in the CTX_LING package, see “CTX_LING:Linguistics” in Chapter 10.

CTX_SVC Package

The stored procedures in CTX_SVC are used to monitor the Services Queue for the status of specific requests. CTX_SVC can be used to check the status of pending requests, and to display errors encountered. You can also cancel the request if it has not been picked up for processing by a ConText server or clear the request if the request encountered an error.

See Also: For more information about procedures in the CTX_SVC package, see “CTX_SVC: Services Queue Administration” in Chapter 10.

Linguistic Personality

To process requests for linguistic output (themes and Gists), a ConText server with the Linguistic personality must be running. A ConText server with the Linguistic personality can also have other personalities in its personality mask.

Starting up ConText servers is the task of the ConText administrator, through the CTXSYS Oracle user.

See Also: For more information about the Linguistic personality and about specifying personality masks for ConText servers, see *Oracle8 ConText Cartridge Administrator's Guide*.

Services Queue

The Services Queue is used for managing ConText linguistic requests. Such a request is cached in memory until the requestor submits the request, at which time the request is added to the Services Queue. If more than one request is cached in memory when the user submits the requests, ConText stores all of the requests as a single batch job.

If a ConText server has the appropriate Linguistic personality, the server monitors the Service Queue for requests and processes the next request in the queue.

Note: If no ConText servers with the 'L' personality are running, the Services Queue still accepts requests and holds the requests for the next available ConText server with the appropriate personality.

The ConText administration tool can be used to perform all administration functions on the Services Queue (e.g., cleaning up entries, etc.). In addition, the

CTX_SVC PL/SQL package can be used to perform ConText administration from the command-line.

Creating Linguistic Output

You can generate linguistic output in batch during the text indexing process or generate it as needed. Because the generation of linguistic output is independent of the text-indexing process, ConText places no restrictions on when you can create themes and Gists.

See Also: For more information about generating linguistic output at indexing time versus generating linguistic output on demand, see “Combining Theme/Text Queries with Linguistic Output” in Chapter 8.

Linguistic Core

The linguistic core is made up of the following components:

- lexicon
- knowledge catalog
- parsing engine

Lexicon

The lexicon is a static knowledge base that provides word and phrase information for the parsing engine. The lexicon recognizes over one million English words and phrases and defines hundreds of lexical characteristics for each word.

Note: The lexicon is specific to the English language, but it recognizes the difference between American and British usage and spelling.

Linguistic information about words in the lexicon is divided into the following types:

Information Type	Description
Syntax	Syntax flags provide surface level assessments of a word or phrase isolated from its grammatical context.
Theme	Theme flags identify the thematic qualities of a word (e.g. weak noun/needs support, strong verb). The parser uses these flags to determine how a word contributes to the thematic construction of the sentence as a whole.

Knowledge Catalog

The knowledge catalog is a language-independent organization of industries, fields of study, special terms and jargon, and abstract concepts. It creates a classification scheme that defines ConText's semantic view of the world.

Context uses the knowledge catalog to generate linguistic output, to classify documents by theme during theme indexing, and to normalize theme queries.

See Also: For more information about the knowledge catalog, see “Understanding Theme Queries” in Chapter 4.

Parsing Engine

ConText uses the linguistic parsing engine whenever you request thematic analysis of text either through `CTX_LING.REQUEST_GIST` or `CTX_LING.REQUEST_THEMES` or through theme indexing and querying.

The parsing engine grammatically analyses text, identifying phrase, sentence and paragraph boundaries. It then interprets meaning, selecting the high-information content to produce theme output. The lexicon and knowledge catalog provide the reference information necessary to do this processing.

If case-conversion is enabled, the parsing engine converts all the text to lowercase and processes the text through the case-sensitivity routines to determine capitalization.

Note: Case conversion does not affect the original text of the documents being processed; only the output of the parsing engine is stored in mixed-case.

Linguistic Output

ConText linguistics produces the following output:

- lists of themes
- theme summaries
- Gists
- theme indexes

List of Themes

You can generate a list of themes or list of main concepts of a document on a per document basis. Because themes present a profile of the main subjects of a document, a list of themes provide a snapshot of what the document is about. To generate a list of themes, use `CTX_LING.REQUEST_THEMES`. You can generate a list of themes in two ways:

- single themes
- theme hierarchies

Single Themes

You can generate up to 16 themes for each document, using the `CTX_LING.REQUEST_THEMES` procedure. This procedure writes a single word or phrase that represents the theme to a row in the theme table. The words or phrases that represent the themes are normalized themes derived from the knowledge catalog.

Theme Hierarchies

You can also generate each theme name accompanied with its parent themes. To enable hierarchical list of themes output, you must use `CTX_LING.SET_FULL_THEMES` before you call `CTX_LING.REQUEST_THEMES`.

Generating theme hierarchical information in the theme table helps to match themes with theme summaries generated with `CTX_LING.REQUEST_GIST`.

Note: ConText linguistics produces only document-level themes; paragraph-level themes cannot be produced.

See Also: For more information about generating themes, see “Generating Themes and Gists” in Chapter 8.

Theme Weight

When you generate document themes, ConText assigns a weight that measures the strength of the theme relative to the other themes in the document.

The cumulative weight of a theme also reflects the overall thematic content of the document. As such, theme weights can be used to compare a document theme to other themes within the same document or to other documents with the same theme.

Theme Summaries

A theme summary for a document provides a short summary of the document from a specific point-of-view. You can generate two types of theme summaries:

- paragraph-level
- sentence-level

A paragraph-level theme summary consists of the paragraph or paragraphs that best represent a single document theme. A sentence-level theme summary consists of the sentence or sentences that best match a single document theme.

To create either paragraph-level or sentence-level theme summaries, use `CTX_LING.REQUEST_GIST`.

Because it provides a concise, focused summary for a particular theme in a document, a theme summary can be used to compare documents with similar themes.

You can control the size of sentence-level and paragraph-level theme summaries with linguistic settings.

Note: The settings for theme summaries can only be modified by creating custom setting configurations in the GUI administration tool.

See Also: For more information about how to generate theme summaries, see “Generating Themes and Gists” in Chapter 8.

For more information on specifying linguistic settings, see “Specifying Linguistic Settings” in Chapter 8.

For a complete list of ConText’s predefined labels, see the specification for CTX_LING.SET_SETTINGS_LABEL in Chapter 10.

Gists

A Gist for a document provides a summary that reflects all of the themes in the document. You can generate two types of Gists:

- paragraph-level
- sentence-level

A paragraph-level Gist consists of the document paragraphs that best represent the themes in a document as a whole. A sentence-level Gist is the sentence or sentences that best represent the themes in a document as a whole.

To generate either a paragraph-level or sentence-level Gist, use CTX_LING.REQUEST_GIST.

Because a Gist is generally longer than a theme summary, it serves better as a document reading tool than a document selection tool. For example, it can be used to quickly scan a document and to extract the most meaningful thematic information.

Note: The settings for Gist can only be modified by creating custom setting configurations in the GUI administration tool.

See Also: For more information about how to generate a Gists, see “Generating Themes and Gists” in Chapter 8.

For more information on specifying linguistic settings, see “Specifying Linguistic Settings” in Chapter 8.

For a complete list of ConText’s predefined labels, see the specification for CTX_LING.SET_SETTINGS_LABEL in Chapter 10.

Theme Indexes

Theme indexes are created as a prerequisite for issuing theme queries. Given a theme policy, you can create a theme index for all documents in an entire text column using CTX_DDL.CREATE_INDEX

Note: A theme index is the only type of linguistic output you can generate *without* running an 'L' server.

See Also: For more information about creating theme indexes, see “Understanding Theme Queries” in Chapter 4 and the *Oracle8 ConText Cartridge Administrator's Guide*.

Linguistic Settings

You can perform linguistic processing of documents to generate themes and Gists only when a ConText server with the Linguistic personality is running. ConText provides two pre-defined linguistic settings, one for mixed case text and one for all upper-case text:

Setting	Description
GENERIC	Default configuration. Parses mixed-case English text. Produces theme output.
Case Sensitive (SA)	<p>Same as GENERIC except that ConText converts all-uppercase or all lower-case text to case-sensitive text before performing theme analysis.</p> <p>When your text is all upper-case or all lower-case and you use this setting to convert the text, Oracle Corporation does not recommend creating theme indexes or issuing theme queries. Creating theme indexes with the SA setting does not produce consistent results.</p>

You can set these options with the CTX_LING.SET_SETTINGS_LABEL procedure. You can also define your own settings with the administration tool and set these settings with CTX_LING.SET_SETTINGS_LABEL. With the administration tool, you can create settings to control the following options:

- size of Gist
- size of theme summary
- Gist generation method

When you use the administration tool to create your own settings, Oracle Corporation recommends using one of the ConText predefined settings as a starting point, depending on whether your text is mixed case, or all upper-case or all lower-case text.

See Also: For more information on how to specify linguistic settings, see “Specifying Linguistic Settings” in Chapter 8.

For more information about the using the administration tool to set your own labels, see the help file for the administration tool.

Using ConText Linguistics

This chapter explains how to use the ConText linguistics to generate linguistic output for English text. It also provides some tips and suggestions for building linguistically-enhanced text applications.

The topics covered in this chapter are:

- Specifying Linguistic Settings
- Generating Linguistic Output
- Monitoring the Services Queue
- Specifying Completion and Error Procedures
- Logging Parse Information
- Combining Theme/Text Queries with Linguistic Output

Specifying Linguistic Settings

When a ConText server with the Linguistic personality is started, ConText automatically loads a default setting configuration (GENERIC) from the database. The default setting configuration is active during your database session unless you explicitly specify a label for a different setting configuration with the `CTX_LING.SET_SETTINGS_LABEL` function.

You can specify one of the two predefined setting configurations (GENERIC or SA) provided with ConText or a custom setting configuration that you create using the administration tool.

To specify a setting configuration for a session, use the `CTX_LING.SET_SETTINGS_LABEL` procedure with a setting label. For example, to process all-uppercase or all-lowercase text for your current session:

```
execute ctx_ling.set_settings_label('SA')
```

When you specify a setting configuration label, ConText checks the label against the setting configuration that is currently active. If the specified setting configuration is not already active, ConText loads the new settings from the database before any documents are processed by ConText servers with the Linguistic personality.

The specified setting configuration is active for your session until `SET_SETTINGS_LABEL` is called with a new setting configuration label.

You can use the `CTX_LING.GET_SETTINGS_LABEL` function to return the label for the active setting configuration for the current session.

Generating Linguistic Output

Before theme and Gist information can be used in an application, you must perform the following tasks:

- create linguistic output tables for the theme and Gist output.
- call the either REQUEST_GIST or REQUEST_THEMES in the CTX_LING package to generate the output.
- call CTX_LING.SUBMIT to submit the request to the services queue.

Note: For ConText to generate linguistic output, at least one server must be running with the Linguistic (L) personality. For more information about ConText Servers, see *Oracle8 ConText Cartridge Administrator's Guide*.

Creating Output Tables

To create a theme table called CTX_THEMES, issue the following SQL statement:

```
create table ctx_themes (
  cid      number,
  pk       varchar2(64),
  theme    varchar2(2000),
  weight   number);
```

To create a Gist table called CTX_GIST, issue the following SQL statement:

```
create table ctx_gist (
  cid      number,
  pk       varchar2(64),
  pov      varchar2(80),
  gist     long);
```

Note: Because the combination of the CID (column ID) and PK (primary key) columns in the output tables uniquely identify each document in a text column, you can use the output tables to store theme and Gist information for multiple text columns. You can also choose to create multiple output tables to store the theme and Gist information separately for each text column.

See Also: For more information about the structure of linguistic output tables, see “Linguistic Output Table Structures” in Appendix A.

Creating Composite Textkey Output Tables

To create a theme table whose textkey has two columns, issue the following SQL statement:

```
create table ctx_themes (  
    cid          number,  
    pk1          varchar2(64),  
    pk2          varchar2(64),  
    theme        varchar2(2000),  
    weight       number);
```

To create a Gist table whose textkey has two columns, issue the following SQL statement:

```
create table ctx_gist (  
    cid          number,  
    pk1          varchar2(64),  
    pk2          varchar2(64),  
    pov          varchar2(80),  
    gist         long);
```

See Also: For more information about the structure of linguistic output tables, see “Linguistic Output Table Structures” in Appendix A.

Generating Themes and Gists

To generate linguistic output for the documents in a text column, you first call CTX_LING.REQUEST_GIST or CTX_LING.REQUEST_THEMES for each document in the column, then call CTX_LING.SUBMIT to enter these requests in the services queue as a single transaction for that particular document.

Note: A policy must be defined for a column before you can generate linguistic output for the documents in the column.

The following example shows how you could use the procedures and functions in CTX_LING package to generate linguistic output:

```

declare handle number;
begin
  ctx_ling.request_themes(
    'CTXSYS.DOC_POLICY',
    '7039',
    'CTXSYS.CTX_THEMES');
  ctx_ling.request_gist(
    'CTXSYS.DOC_POLICY',
    '7039',
    'CTXSYS.CTX_GIST');
  handle := ctx_ling.submit;
end;

```

The first two calls request themes and Gist output for document 7039 in the text column for the DOC_POLICY policy. These procedures store the themes and Gists in the linguistic output tables (*ctx_themes* and *ctx_gists*), which were created in the previous step.

The final API call submits the requests as one batch request to the services queue and returns a handle which can be used to monitor the status of the request. Because the two requests are submitted as one batch request, ConText parses the document only once while still generating both theme and Gist output.

Generating Theme Hierarchical Information

By default, ConText generates single theme names when you request a list of themes with CTX_LING.REQUEST_THEMES. To generate the hierarchical theme information with theme names, you must set the full themes flag to TRUE with CTX_LING.SET_FULL_THEMES.

For example, the following SQL statements generate and output single theme information for a document identified by *pk*:

```

SQL> exec ctx_ling.request_themes('ctx_thidx', pk, 'ctx_themes')
SQL> exec ctx_ling.submit(200)
SQL> select theme from ctx_themes;

```

THEME

```

-----
NASDAQ - National Association of Securities Dealers Automated Quotation System
stocks
indexes
weakness
composites
prices

```

```
franchises
shares
cellularity
declining issues
measures
analysts
OTC
purchases
Wall Street
lows
```

16 rows selected.

However, when you set the full themes flag to TRUE, ConText generates theme hierarchical information:

```
SQL> exec ctx_ling.set_full_themes(TRUE)
SQL> exec ctx_ling.request_themes('ctx_thidx', pk, 'ctx_themes')
SQL> exec ctx_ling.submit(200)
SQL> select theme from ctx_themes
```

THEME

```
-----
:stock market:NASDAQ - National Association of Securities Dealers Automated
Quotation System:
:stock market:stocks:
:catalogs, itemization:indexes:
:weakness, fatigue:weakness:
:combination, mixture:composites:
:retail trade industry:prices:
:business fundamentals:franchises:
:possession, ownership:shares:
:cellularity:
:stock market:declining issues:
:analysis, evaluation:measures:
:analysis, evaluation:analysts:
:OTC:
:general commerce:purchases:
:general investment:Wall Street:
:bottoms, undersides:lows:
```

Generating theme hierarchical information as such helps to match themes with the theme summaries generated with CTX_LING.REQUEST_GIST.

Monitoring the Services Queue

When you submit a request to the services queue with `CTX_LING.SUBMIT`, a handle is returned. With this handle, you can use procedures in the `CTX_SVC` package to perform the following tasks:

- monitor the status of requests in the queue
- remove pending requests (requests that have not yet been picked up by a ConText server)
- clear requests with errors

Monitoring the Status of Requests

To monitor the status of requests in the Services Queue, use the `CTX_SVC.REQUEST_STATUS` function. This function returns one of the following statuses:

Status	Meaning
PENDING	The request has not yet been picked up by a ConText server.
RUNNING	The request is being processed by a ConText server.
ERROR	The request errored.
SUCCESS	The request completed successfully.

For example, the following PL/SQL procedure submits a request to generate themes and gist for a document with an id of 49. It then checks the status of the request.

```
CREATE OR REPLACE PROCEDURE GENERATE_THEMES AS

    v_Handle number;
    v_Status varchar2(10);
    v_Time date;
    v_Errors varchar2(60);

BEGIN
    DBMS_OUTPUT.PUT_LINE('Begin generate_themes procedure' );

    ctx_ling.request_themes('CTXDEMO.DEMO_POLICY', '49', 'CTXDEMO.ctx_themes' );
    ctx_ling.request_gist('CTXDEMO.DEMO_POLICY', '49', 'CTXDEMO.ctx_gist' );
    v_Handle := ctx_ling.submit;
```

```
DBMS_OUTPUT.PUT_LINE( v_Handle );

v_Status := ctx_svc.request_status( v_Handle, v_Time, v_ErrorS );
DBMS_OUTPUT.PUT_LINE( v_Status );
DBMS_OUTPUT.PUT_LINE( v_Time );
DBMS_OUTPUT.PUT_LINE( substr( v_Errors, 1, 20 ) );

EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE( ' Exception handling' );

END GENERATE_THEMES;
/
```

This procedure binds the return value of REQUEST_STATUS to *v_Status* for the linguistic request identified by *v_Handle*. The value for *v_Handle* is returned by the call to CTX_LING.SUBMIT which placed the requests for the themes and gists in the Services Queue.

Removing Pending Requests

To remove requests with a status of PENDING from the Services Queue, use the CTX_SVC.CANCEL procedure.

For example:

```
execute ctx_svc.cancel(3321)
```

In this example, a pending request with handle 3321 is removed from the Services Queue.

If a request has a status of RUNNING, ERROR, or SUCCESS, it cannot be removed from the Services Queue.

Clearing Requests with Errors

To remove requests with a status of ERROR from the Services Queue, use the CTX_SVC.CLEAR_ERROR procedure.

For example:

```
execute ctx_svc.clear_error(3321)
```

In this example, a request with handle 3321 is removed from the Services Queue.

If a value of 0 (zero) is specified for the handle, all requests with a status of ERROR are removed from the queue. If a request has a status of PENDING, RUNNING, or SUCCESS, it cannot be removed from the queue using CLEAR_ERROR.

Specifying Completion and Error Procedures

To specify a procedure to be called when a linguistic request completes or errors, use the `SET_COMPLETION_CALLBACK` and `SET_ERROR_CALLBACK` procedures in `CTX_LING`. ConText invokes the procedure defined by `SET_COMPLETION_CALLBACK` after it processes a linguistic request; ConText invokes the procedure defined by `SET_ERROR_CALLBACK` when it encounters an error.

The following is an example of how to define and use a completion callback procedure. This example is taken from `genling.sql` in the *ctxling* demonstration provided with the ConText distribution package.

For every linguistic request processed, *ling_comp_callback* keeps track of the number of articles processed by decrementing *num_docs*, previously defined as the number of articles in the table. The procedure also keeps track of the any errors by incrementing *num_errors*.

```
create or replace procedure LING_COMP_CALLBACK (  
    p_handle in number,  
    p_status in varchar2,  
    p_errors in varchar2  
) IS  
    l_total number;  
    l_pk      varchar2(64);  
BEGIN  
  
    -- decrement the count in the tracking table  
  
    update ling_tracking set num_docs = num_docs - 1;  
  
    -- if the request errored, mark the errors in the pending table  
  
    IF (p_status = 'ERROR') then  
        update ling_tracking set num_errros = num_errors + 1;  
    end IF;  
    commit;  
  
END;  
/
```

The following code is an anonymous PL/SQL block that sets the linguistic completion callback procedure to *ling_comp_callback* and then generates linguistic output for every document in the *articles* table:

```
declare  
    cursor c1 is select article_id
```



```
                                from articles;
l_handle number;

begin

-- set the completion callback procedure to keep the pending table
-- in sync with the number of documents processed (completed requests)
-- and the number of errored requests.

    ctx_ling.set_completion_callback('LING_COMP_CALLBACK');
end;

-- loop through all articles in the article table, requesting themes
-- and gists
--

for crec in c1 loop

ctx_ling.request_themes('DEMO_POLICY', crec.article_id, 'ARTICLE_THEMES');
ctx_ling.request_gist('DEMO_POLICY', crec.article_id, 'ARTICLE_GISTS');
l_handle := ctx_ling.submit;

end loop;

end;
```

Logging Parse Information

At start-up of a ConText server, the logging of linguistic parse information is disabled by default.

To enable logging of the parse information generated by ConText linguistics during a session, use the CTX_LING.SET_LOG_PARSE procedure.

For example:

```
execute ctx_ling.set_log_parse('TRUE')
```

Once you enable parse logging for a session, it is active until you explicitly disable it during the session. You can use the CTX_LING.GET_LOG_PARSE function to know whether parse logging is enabled or disabled for the session.

Attention: Parse logging is a useful feature if you are having difficulty generating linguistic output and you want to monitor how ConText is parsing your documents; however, parse logging may affect performance *considerably*. As such, you should *only* enable parse logging if you encounter problems with generating linguistic output.

Combining Theme/Text Queries with Linguistic Output

Theme queries allow you to search a set of documents for a given theme. The result is a hitlist containing the IDs of the documents that match the theme.

Generating list of themes is a good way of extending theme queries. For a document in a theme query hitlist, the user can learn more about the document by reading a list-of themes or Gist.

For example, suppose a theme query on *music* returns a hitlist containing 20 documents. If these documents are lengthy, the user might not want to read every single document to find out what each is about. Rather than return to the user the document text, you can return a list of themes or a Gist for each document, which the user could skim.

Implementation

Generally, you can generate linguistic output for a document set at two different times:

- text/theme indexing time
- query time

Generating Linguistic Output at Indexing Time

You can generate linguistic output (creating the list of themes in this case) at indexing time, that is, before the queries are issued against the document set. When you do so, the linguistic output is returned to the user immediately, since the output was already created.

However, while the retrieval time for the linguistic output is good, the drawback to this method is that you have to maintain a permanent theme output table, using your own triggers to keep it updated. A permanent theme table for an entire document set also takes up system disk space.

Generating Linguistic Output at Query-Time

You could also generate a list of themes after executing a query. The advantage of generating themes as needed is that the output table lasts only for the user session; you need not maintain a permanent theme table for all your documents.

However, generating list of themes on the fly takes time depending on the number of documents, the length of the documents, and how your linguistic servers are configured. A user might not want to wait a few minutes to process a large number of documents.

The example below shows how to generate linguistic output after a theme query.

Example

The following PL/SQL code illustrates how to generate a list of themes for every document in a hitlist table. (You can use the same method to loop through any text table, once the text column table has a policy attached to it.)

```
create or replace procedure get_theme IS
handle number;

cursor ctx_cur is
    select textkey from ctx_temp;

BEGIN

ctx_query.contains('DOWTHEME', 'birds', 'ctx_temp');

for ctx_cur_rec in ctx_cur loop
    ctx_ling.request_themes('DOWPOLICY', ctx_cur_rec.textkey, \
        'ctx_themes');
    handle:= ctx_ling.submit;
end loop;

END;
/
```

This routine first declares a cursor that selects the rows from the *ctx_temp* result table, to be populated with a theme query on *birds*.

The cursor FOR loop opens the cursor, executing the select statement that copies all textkeys in the *ctx_temp* table to the cursor. The loop index *ctx_cur_rec* is implicitly defined as a cursor record of type%ROWTYPE.

Every iteration of the loop calls the CTX_LING.REQUEST_THEMES procedure with the document textkey derived from *ctx_cur_rec*. Each request is submitted to the services queue with CTX_LING.SUBMIT, which returns a handle.

The theme output is written to the *ctx_themes* table.

SQL Functions

This chapter contains details for using the ConText SQL functions in SELECT statements to perform one-step queries.

The following topics are covered in this chapter:

- Query Functions
- CONTAINS
- SCORE
- SELECT Statement

Query Functions

In addition to the functions in the PL/SQL packages, ConText provides the following functions for performing one-step queries in SQL*Plus:

Name	Description
CONTAINS	Specifies the query expression and SCORE label for a one-step query.
SCORE	Returns the score generated by CONTAINS.

Prerequisites

Before one-step queries can be executed, the database in which the text resides must be text enabled by setting the ConText initialization parameter `TEXT_ENABLE = TRUE`. This can be done in two ways:

- setting it in the `initSID.ora` system initialization file
- using the `ALTER SESSION` command

See Also: For more information about initialization parameters and the `initSID.ora` file, see *Oracle8 Server Administrator's Guide*.

For more information about using the `ALTER SESSION` command, see *Oracle8 Server SQL Reference*.

CONTAINS

Use the CONTAINS function in the WHERE clause of a SELECT statement to specify the query expression for a one-step query. You can also define a numeric label for the scores generated by the function so that the SCORE function can be used in other clauses of the SELECT statement.

Syntax

```
CONTAINS(  
    column_id      NUMBER,  
    text_query     VARCHAR2,  
    label          NUMBER,  
    pol_hint       VARCHAR2)
```

column_id

Specify the text column to be searched in the table.

text_query

Specify the query expression for the text or theme to search for in *column_id*.

See Also: For more information about how to write query expressions, see Chapter 3, “Understanding Query Expressions”.

label

Specify the label that identifies the score generated by the CONTAINS function (required only if CONTAINS called more than once in a query).

pol_hint

Specify which policy to use for text columns that have multiple policies.

Example

See the SELECT statement syntax in this chapter.

Notes

Each CONTAINS function in a query produces a separate set of score values. When there are multiple CONTAINS functions, each CONTAINS function must have a *label* specified.

If only one CONTAINS function is used in a SELECT statement, the *label* parameter is not required in the CONTAINS function; however, a SCORE label value of zero (0) is automatically generated. When the SCORE function is call (e.g. in a SELECT clause), the function must reference the label value.

The CONTAINS function may only be used in the WHERE clause of a SELECT statement; it may not be issued in the WHERE clause of an UPDATE, INSERT or DELETE statement.

In order to specify *pol_hint*, you must specify *label* as a place holder. *pol_hint* must name a policy that is indexed either by text or theme. Do not specify *user.policy_name* notation for *pol_hint*; specify only policy name, otherwise ConText will raise an error. You cannot specify bind variables for *pol_hint*.

When you do not specify *pol_hint* and *column_id* has more than one indexed policy attached to it, ConText uses the policy whose name is lexicographically first. For example, if a text column had policies named POL1 and POL2 associated with it and you did not specify *pol_hint*, ConText uses POL1.

Suggestion: Oracle Corporation does not recommend relying on ConText to select a policy when you perform queries on columns with multiple policies. In this situation, always specify a policy name in *pol_hint*.

SCORE

The SCORE function returns the score values produced by the CONTAINS function in a one-step query.

Syntax

```
SCORE(label NUMBER)
```

label

Identifies the scores produced by a query.

Notes

The SCORE function may be used in any of these clauses: SELECT, ORDER BY, or GROUP BY.

The value specified for LABEL is the same value defined by the LABEL argument in the CONTAINS function that generated the scores and is referenced by the SCORE function in all other clauses.

If only one CONTAINS function is used in a SELECT statement, the LABEL parameter is not required in the CONTAINS clause, but a SCORE label value of zero (0) will be generated. All other clauses must then refer to SCORE(0) or SCORE(*).

Example

```
SELECT SCORE(10), title FROM documents
WHERE CONTAINS(text, 'dog', 10) > 0
ORDER BY SCORE(10);
```

This example returns the score and title of all articles (documents) in the DOCUMENTS.TEXT column that contain the word dog, sorted by score.

SELECT Statement

You perform one-step queries in SQL*Plus using the SELECT statement. The following syntax illustrates how the CONTAINS and SCORE query functions can be used in a SELECT statement.

Syntax

```
SELECT SCORE(label1), SCORE(label2), ...SCORE(labeln),  
column1, column2, ... columnn  
FROM table[@dblink]  
WHERE CONTAINS (column_id, 'text_query', label1, polhint1) > 0  
CONTAINS (column_id, 'text_query', label2, polhint2) > 0  
CONTAINS (column_id, 'text_query', labeln, polhintn) > 0  
ORDER BY SCORE(labeln)
```

label_x

Specify the numeric label that identifies the specific CONTAINS function that generated the score (required only when CONTAINS is called more than once in a query).

column_n

Specify the columns to be returned by the query. Each CONTAINS clause produces a virtual SCORE column that can be referenced by its numeric label (*label_x*) and included in the query output.

table

Specify the name of the table that contains the text column to be searched.

Note: If a database link has been created for a remote database, the table specified in a one-step query can reside in the remote database. The table name must include the database link (*@dblink*) to access the remote table.

For more information about database links and remote queries, see *Oracle8 Server Concepts*.

column_id

Specify the name of the text column.

text_query

Specify the query expression to be used to return the relevant text.

pol_hint_x

Specify the policy to be used when *column_id* has multiple policies.

Notes

The CONTAINS function must always be followed by the > 0 syntax which specifies that the score value calculated by the CONTAINS function must be greater than zero for the row to be selected.

Note: Other comparison operators and other numeric values can be used to satisfy this requirement and select rows with specific SCORE values; however, this method of refining the selection criteria is significantly less efficient than using the threshold and weight query expression operators.

The following example returns the names of all employees who have listed trumpet in their resume or who have been in an orchestra, sorted by the value of the score for the first CONTAINS (trumpet) and the second CONTAINS (orchestra).

```
SELECT employee_name, SCORE(10), SCORE(20)
FROM employee_database
WHERE CONTAINS (emp.resume, 'trumpet', 10) > 0 OR
CONTAINS (emp.history, 'orchestra', 20) > 0
ORDER BY NVL(SCORE(10),0), NVL(SCORE(20),0);
```

PL/SQL Packages

This chapter describes the ConText Option PL/SQL packages you use to develop applications. The following topics are described in this chapter are:

- Developing with ConText PL/SQL Packages
- CTX_QUERY: Query and Highlighting
- CTX_LING:Linguistics
- CTX_SVC: Services Queue Administration

Developing with ConText PL/SQL Packages

Before you can develop your own PL/SQL stored procedures and triggers that call the procedures in the ConText packages described in this chapter, your ConText administrator must explicitly grant EXECUTE privileges to you for each ConText PL/SQL package you use.

See Also: For more information about granting execute privileges, see *Oracle8 ConText Cartridge Administrator's Guide*.

For more information about creating and invoking PL/SQL packages, see *Oracle8 Server Application Developer's Guide*.

CTX_QUERY: Query and Highlighting

The CTX_QUERY package contains stored procedures and functions that enable processing of two-step queries and highlighting for documents returned by queries.

The package includes the following procedures and functions:

Name	Description
CLOSE_CON	Closes the in-memory query cursor.
CONTAINS	Selects documents in the text column for a policy and writes the results to a specified result table.
COUNT_HITS	Performs a query and returns the number of hits without returning a hitlist.
COUNT_LAST	Returns the number of hits retrieved in the last call to CONTAINS or OPEN_CON.
FEEDBACK	Generates query expression feedback information.
FETCH_HIT	Retrieves hits stored in query buffer by OPEN_CON.
GETTAB	Gets tables from the result table pool.
HIGHLIGHT	Provides filtering and/or highlighting for documents returned by a query.
OPEN_CON	Opens a cursor and executes an in-memory query.
PKDECODE	Decodes a composite textkey string (value).
PKENCODE	Encodes a composite textkey string (value).
PURGE_SQE	Deletes all SQEs from SQE tables.
REFRESH_SQE	Re-executes an SQE and updates the results stored in the SQE tables.
RELTAB	Releases tables allocated by GETTAB.
REMOVE_SQE	Removes a specified SQE from the SQL tables.
STORE_SQE	Executes a query and stores the results in stored query expression tables.

CLOSE_CON

The CTX_QUERY.CLOSE_CON procedure closes a cursor opened by CTX_QUERY.OPEN_CON. It is used in in-memory queries and called after CTX_QUERY.FETCH_HIT, which retrieves the desired number of hits.

Syntax

```
CTX_QUERY.CLOSE_CON(curid NUMBER);
```

curid

Specify the cursor to be closed.

Examples

See CTX_QUERY.FETCH_HIT.

CONTAINS

The CTX_QUERY.CONTAINS procedure selects documents from a text column that match the specified search criteria, generates scores for each document, and writes the results to a specified hitlist result table.

Syntax

```
CTX_QUERY.CONTAINS(  
    policy_name[@dblink] IN VARCHAR2,  
    text_query           IN VARCHAR2,  
    restab              IN VARCHAR2,  
    sharelevel          IN NUMBER DEFAULT 0,  
    query_id            IN NUMBER,  
    cursor_id           IN NUMBER,  
    parallel            IN NUMBER,  
    struct_query        IN VARCHAR2);
```

policy_name

Specify the policy that identifies the text column to be searched.

If a database link to a remote database has been created, the database link can be specified as part of the policy name (using the syntax shown) to reference a policy in the remote database.

text_query

Specify the query expression to be used as criteria for selecting rows.

See Also: For more information about how to write query expressions, see Chapter 3, “Understanding Query Expressions”.

restab

Specify the name of the hitlist table that stores intermediate results returned by CONTAINS.

sharelevel

Specify whether the results table is shared by multiple CONTAINS. Specify 0 for exclusive use and 1 for shared use. This parameter defaults to 0 (single-use).

When you specify 0, the system automatically truncates the result table before the query. In this case, *conid* is set to NULL and *query_id* is ignored.

When you specify 1 for multiple use, you must give a *query_id* to distinguish the results in the shared result table. Because the system does not truncate shared result tables, you must get rid of results from a previous CONTAINS by deleting from the result table where *conid* = *query_id* before you issue the query.

query_id

Specify the ID used to identify query results returned by a CONTAINS procedure when more than one CONTAINS uses the same result table (*sharelevel* = 1).

cursor_id

Not currently used.

parallel

Specify the number of ConText servers (with the Query personality) which execute a query and write the results to *restab*.

struct_query

Specify the structured WHERE condition related to *text_query*. This WHERE condition can include a subquery that selects rows from a structured data column in another table.

Examples

```
exec ctx_query.contains('my_pol', 'cat|dog', 'CTX_TEMP', 1, 10)
```

```
exec ctx_query.contains('my_pol@db1', 'oracle', 'CTX_DB1_TEMP')
```

In the first example, the results of the query for the term *cat* or *dog* are stored in the *ctx_temp* result table. The result table is shared because *sharelevel* is specified as 1. The results in *ctx_temp* are identified by *query_id* of 10.

In the second example, *my_pol* exists in a remote database that has a database link named DB1. The result table, *ctx_db1_temp* exists in the same remote database.

Notes

The *parallel* parameter does *not* support the max (:) and first/next (#) query expression operators. When you specify either operator in the query expression, the query is processed by a single ConText server, regardless of the specified *parallel* level.

sharelevel determines whether the hitlist result table is shared by multiple CONTAINS procedures.

If the result table (*restab*) is used to hold the results of multiple CONTAINS, a *sharelevel* must be specified by each CONTAINS so that the results of previous CONTAINS are not truncated.

If a query is performed on a policy in a remote database, the result table specified by *restab* must exist in the remote database.

In *struct_query*, you can use any predicate, value expression or subquery except USERENV function, CONTAINS function, SCORE function, DISPLAY function and the ROWNUM pseudo column.

If the user who includes a structured query in a two-step query is not the owner of the table containing the structured and text columns, the user must have SELECT privilege with GRANT OPTION on the table. In addition, if the object being queried is a view, the user must have SELECT privilege with GRANT OPTION on the base table for the view.

See Also: For more information about SELECT privilege with GRANT OPTION, see *Oracle8 Server SQL Reference*.

COUNT_HITS

The CTX_QUERY.COUNT_HITS function executes a query for a policy and returns the number of hits for the query. It does *not* populate a result table with query results.

COUNT_HITS can be called in two modes, estimate and exact. The results in estimate mode may be inaccurate; however, the results are generally returned faster than in exact mode.

Syntax

```
CTX_QUERY.COUNT_HITS(  
    policy_name[@dblink]  IN VARCHAR2,  
    text_query             IN VARCHAR2,  
    struct_query           IN VARCHAR2,  
    exact                  IN BOOLEAN DEFAULT FALSE)  
RETURN NUMBER;
```

policy_name[@dblink]

Specify the name of the policy that defines the column to be searched.

If a database link to a remote database has been created, the database link can be specified as part of the policy name (using the syntax shown) to reference a policy in the remote database.

text_query

Specify the query expression to be used as criteria for counting returned hits (rows)

struct_query

Specify the structured where condition related to *text_query*.

exact

Specify TRUE to obtain an exact count of the documents in the hitlist. Specify FALSE to obtain an estimate count. The result returned when you request an estimate count includes hits for documents that have been deleted or updated. The default is FALSE.

Examples

```
declare count number;  
begin
```

```
count := ctx_query.count_hits(my_pol, 'dog|cat', TRUE);  
dbms_output.put_line('No. of Docs with dog or cat:');  
dbms_output.put_line(count);  
end;
```

Returns

NUMBER that represents the number of hits.

Notes

Counting query hits can be performed in two modes: estimate and exact. The modes are based on the method ConText uses to record deleted documents in a text index.

In exact mode, hits are returned *only* for those documents that satisfy the conditions of the query expression and are currently in the text column of the table.

In estimate mode, hits may be included for documents that satisfy the query condition, but have been deleted from the text column or have been updated so that they no longer satisfy the query expression. This can occur when the text index for the column has not been optimized and the internal document IDs are still present in the index.

In general, the inaccuracy of the results returned by COUNT_HITS in estimate mode is proportional to the amount of DML that has been performed on a text column.

Note: If the index being queried has been optimized and no further DML has been performed on the text column, estimate mode will return accurate results.

See Also: For more information about text indexing, DML, and optimization, see *Oracle8 ConText Cartridge Administrator's Guide*.

COUNT_LAST

Use the CTX_QUERY.COUNT_LAST function to obtain the number of hits after executing CONTAINS in a two-step query or OPEN_CON in an in-memory query. The alternative method of obtaining the number of hits is to run the query once to get the row count using CTX_QUERY.COUNT_HITS and then run the query again to get the query results.

Syntax

```
CTX_QUERY.COUNT_LAST          RETURN NUMBER;
```

Returns

The number of hits obtained from the last call to CTX_QUERY.CONTAINS or CTX_QUERY.OPEN_CON.

Examples

In-memory Query

```
declare
    curid number;
    count number;
begin
    curid := ctx_query.open_con('mypol', 'me', score_sorted=>true);
    count := ctx_query.count_last ;
end
```

Two-step Query

```
declare
    count number;
begin
    ctx_query.contains('mypol', 'dog', 'ctx_temp');
    count := ctx_query.count_last ;
end
```

Notes

With two-step queries, COUNT_LAST always returns an exact count.

With in-memory queries, COUNT_LAST returns an exact count except when you include a structured condition, in which case it returns an estimate. This is because COUNT_LAST ignores the structured condition, specified in the *struct_query* parameter of OPEN_CON, when computing number of hits in an in-memory query.

For two-step queries, the COUNT_LAST function is not meant to replace calling COUNT_HITS, which is always faster than running the query. However, in the case where you want to process all hits in a two-step query, issuing the query with CONTAINS and then calling COUNT_LAST is more efficient than calling COUNT_HITS and then calling CONTAINS.

With in-memory queries, issuing OPEN_CON and then calling COUNT_LAST is always a more efficient way to obtain an estimate of the query hits over calling COUNT_HITS and then calling OPEN_CON, since COUNT_LAST returns a number faster than COUNT_HITS.

FEEDBACK

Use CTX_QUERY.FEEDBACK to generate feedback information for query expressions. This procedure creates a graphical representation of the ConText parse tree and stores the information in a feedback table.

Syntax

```
CTX_QUERY.FEEDBACK(  
    policy_name      IN VARCHAR2,  
    text_query       IN VARCHAR2,  
    feedback_table   IN VARCHAR2,  
    sharelevel       IN NUMBER DEFAULT 0,  
    feedback_id      IN VARCHAR2 DEFAULT NULL  
);
```

policy_name

Specify the policy that identifies the text column to be queried.

text_query

Specify the query expression to be used as criteria for selecting rows.

feedback_table

Specify the name of the feedback table to store representation of the ConText parse tree for *text_query*.

sharelevel

Specify whether *feedback_table* is shared by multiple FEEDBACK calls. Specify 0 for exclusive use and 1 for shared use. This parameter defaults to 0 (single-use).

When you specify 0, the system automatically truncates the feedback table before the next call to FEEDBACK.

When you specify 1 for shared use, Context does not truncate the feedback table. Only results with the same *feedback_id* are updated. When no results with the same *feedback_id* exist, then new results are added to the feedback table.

feedback_id

Specify a name that identifies the feedback results returned by a FEEDBACK procedure when more than one FEEDBACK call uses the same shared feedback table. This parameter defaults to NULL.

Notes

The user must have at least INSERT and DELETE privileges on the feedback table. You must have at least CTXUSER role to call FEEDBACK.

When you include a wildcard, fuzzy, or soundex operator in *text_query*, ConText looks at the index tables to determine the expansion.

When you include the SQE operator in *text_query*, the expression feedback (expansion of the SQE expression) is based on the current state of the index and will take into account any inserts, updates, or deletes made to the base table; however, unlike a call to CONTAINS, the SQE is *not* updated or refreshed as a result of the call to FEEDBACK.

Wildcard, fuzzy (?), and soundex (!) expression feedback does not account for lazy deletes.

You cannot use FEEDBACK with remote queries.

To use the FEEDBACK procedure, you must have at least one Q server running.

See Also: For more information on using the FEEDBACK procedure, see Chapter 5, “Query Expression Feedback”.

FETCH_HIT

The `CTX_QUERY.FETCH_HIT` function returns a hit stored in the query buffer created by `CTX_QUERY.OPEN_CON`. You must call `FETCH_HIT` once for each hit in the buffer until the desired number of hits is returned or the buffer is empty.

Syntax

```
CTX_QUERY.FETCH_HIT(  
    curid      IN  NUMBER,  
    pk         OUT VARCHAR2,  
    score      OUT  NUMBER,  
    col1       OUT  VARCHAR2,  
    col2       OUT  VARCHAR2,  
    col3       OUT  VARCHAR2,  
    col4       OUT  VARCHAR2,  
    col5       OUT  VARCHAR2);
```

curid

Specify the cursor opened by `CTX_QUERY.OPEN_CON`.

pk

Returns the primary key of the document. When the primary key is a composite textkey, PK is returned as encoded string. In this situation, use `CTX_QUERY.PKDECODE` to access an individual textkey column.

score

Returns the score of the document.

col1-5

Returns additional columns for the document.

Returns

NUMBER that indicates whether hit was retrieved: 0 if no hits fetched, 1 if hit was fetched.

Example

```
declare
  score  char(5);
  pk     char(5);
  curid  number;
  title  char(256);
begin
  dbms_output.enable(100000);
  curid := ctx_query.open_con(
    policy_name => 'MY_POL',
    text_query  => 'dog',
    score_sorted => true,
    other_cols  => 'title');
  while (ctx_query.fetch_hit(curid, pk, score, title)>0)
  loop
    dbms_output.put_line(score||pk||substr(title,1,50));
  end loop;
  ctx_query.close_con(curid);

end;
```

Notes

If the primary key PK is a composite textkey, use CTX_QUERY.PKDECODE to access the individual columns of the textkey.

GETTAB

CTX_QUERY.GETTAB procedure allocates result tables from the result table pool to be used to store results from CTX_QUERY.HIGHLIGHT or CTX_QUERY.CONTAINS.

If no result table of the specified type exists, GETTAB creates a new table.

Syntax

```
CTX_QUERY.GETTAB(  
    type           IN  VARCHAR2,  
    tab            OUT VARCHAR2,  
    tk_count       IN   NUMBER  DEFAULT 1);
```

type
Specify the type of table to be allocated for text processing. This parameter must be fully qualified with the PL/SQL package name (CTX_QUERY). The type of table you specify can be one of the following:

Table Type	Description	Stores Results For
DOCTAB	Result table which is used to store the marked-up text (MUTAB) or plain ASCII text (PLAINTAB) returned by CTX_QUERY.HIGHLIGHT	MUTAB or PLAINTAB
RDOCTAB	Result table which is used to store the non-filtered documents (NOFILTAB) or ICF output (ICFTAB) returned by CTX_QUERY.HIGHLIGHT	NOFILTAB or ICFTAB
HIGHTAB	Result table which is used to store the textkey, offsets, and lengths of query terms to be highlighted in documents (returned by CTX_QUERY.HIGHLIGHT)	HIGHTAB
HITTAB	Result table which is used to store the hitlist data returned by CTX_QUERY.CONTAINS	Hitlist Result Table.

See Also: For more information about the structure of result tables, see Appendix A, “Result Tables”.
For more information about using HIGHLIGHT, see Chapter 6, “Document Presentation”.

tab

Returns the name of the allocated table.

tk_count

Specify the number of textkeys in the allocated result table. This parameter applies only to HITTAB tables. The *tk_count* parameter defaults to 1.

Examples

```
set serveroutput on
declare
  mytab varchar2(32) ;
begin
  ctx_query.gettab(CTX_QUERY.HITTAB, mytab, 3) ;
  dbms_output.put_line('table : '||mytab) ;
end ;
```

This example returns a HITTAB result table that has a three-column composite text-key. The name of the table is then output.

The schema for the returned table is: TEXTKEY, TEXTKEY2, TEXTKEY3, SCORE, CONID.

Notes

The *tk_count* parameter applies only to HITTAB tables; it has no effect on other table types.

HIGHLIGHT

THE CTX_QUERY.HIGHLIGHT procedure takes a query specification and a document textkey and returns information that you can use to display the document with or without the query terms highlighted. This procedure is usually used after a query, from which you identify the document to be processed.

Syntax

```
CTX_QUERY.HIGHLIGHT(  
    cspec          IN VARCHAR2,  
    textkey        IN VARCHAR2,  
    query          IN VARCHAR2 DEFAULT NULL,  
    id             IN NUMBER   DEFAULT NULL,  
    nofilttab      IN VARCHAR2 DEFAULT NULL,  
    plaintab       IN VARCHAR2 DEFAULT NULL,  
    hightab        IN VARCHAR2 DEFAULT NULL,  
    icftab         IN VARCHAR2 DEFAULT NULL,  
    mutab          IN VARCHAR2 DEFAULT NULL,  
    starttag       IN VARCHAR2 DEFAULT NULL,  
    endtag         IN VARCHAR2 DEFAULT NULL);
```

cspec

Specify the policy name for the column in which the document is stored.

textkey

Specify the unique identifier (usually the primary key) for the document.

The *textkey* parameter can be a single column textkey or an encoded specification for a composite (multiple column) textkey.

query

Specify the original query expression used to retrieve the document. If NULL, no highlights are generated.

id

Specify the identifier to be used in the results tables to identify the rows that were returned by this procedure call. If NULL, the result tables are truncated.

nofilttab

Specify name of the RDOCTAB table where unfiltered document is stored. If NULL, the unfiltered version is not returned.

plaintab

Specify the name of the DOCTAB table where plain text version of document is stored. If NULL, the plain text is not returned.

hightab

Specify the name of the HIGHTAB table where highlight information for the document is stored. If NULL, the highlights are not returned.

icftab

Used internally by the Windows 32-bit viewer to specify where the ICF output required for WYSIWYG viewing of documents is stored. If NULL, the ICF is not returned.

mutab

Specify table where marked-up, ASCII version of document is stored. If NULL, marked-up version is not returned.

starttag

Specify the markup to be inserted by HIGHLIGHT for indicating the start of a highlighted term.

The default for ASCII and formatted documents is '<<<'.

The default for HTML documents filtered using an external filter is '<<<'.

The default for HTML documents filtered using the internal HTML filter is the HTML tag used to indicate the beginning of a font change (i.e.).

endtag

Specify the markup to be inserted by HIGHLIGHT for indicating the end of a highlighted term.

The default for ASCII and formatted documents is '>>>'.

The default for HTML documents filtered using an external filter is '>>>'.

The default for HTML documents filtered using the internal HTML filter is the HTML tag used to indicate the end of a font change (i.e.).

Examples

```
begin
  ctx_query.highlight(cspec => '2354',
    textkey => '23',
    query => 'dog|cat',
    nofiltab => 'FORMATTED_TEXT',
    hightab => 'HIGHLIGHTED_TEXT',
    starttag => '<*<',
    endtag => '>*>');
end;
```

Notes

Before CTX_QUERY.HIGHLIGHT is called, the highlight/display result tables (NOFILTAB, PLAINTAB, HIGHTAB, MUTAB, and ICFTAB) for the desired output must be created, either manually or using the PL/SQL procedure CTX_QUERY.GETTAB.

If the *query* argument is not specified or is set to NULL, highlighting is not generated.

If *query* includes wildcards, stemming, fuzzy matching which result in stopwords being returned, HIGHLIGHT does not highlight the stopwords.

When *textkey* is a composite textkey, you must encode the composite textkey string using the CTX_QUERY.PKENCODER procedure.

If any of the table name parameters are omitted or set to NULL, the respective table is not populated.

If the *id* argument is not specified or if *id* is set to NULL, each specified table has all its rows deleted and the session-id is used as the ID for all inserted rows. If an *id* is specified, all rows with the same *id* are deleted from the respective tables before new rows are generated with that *id* by the HIGHLIGHT procedure.

For HTML documents filtered through the internal HTML filter, the ASCII output generated for MUTAB retains the HTML tags from the original document.

For HTML documents filtered through an external filter, HIGHLIGHT removes all the HTML tags and stores only the plain (ASCII) marked-up text for the document in MUTAB.

See Also: For more information about internal and external filters, see *Oracle8 ConText Cartridge Administrator's Guide*.

For more information about the structure of result tables, see Appendix A, "Result Tables".

OPEN_CON

The `CTX_QUERY.OPEN_CON` function opens a cursor to a query buffer and executes a query using the specified query expression. The results of the query are stored in the buffer and retrieved using `CTX_QUERY.FETCH_HIT`.

Syntax

```
CTX_QUERY.OPEN_CON(  
    policy_name[@dblink]    IN VARCHAR2,  
    text_query               IN VARCHAR2,  
    score_sorted             IN BOOLEAN DEFAULT FALSE,  
    other_cols               IN VARCHAR2,  
    struct_query             IN VARCHAR2)  
RETURN NUMBER;
```

policy_name[@dblink]

Specify the name of the policy that defines the column to be searched.

If a database link to a remote database has been created, the database link can be specified as part of the policy name (using the syntax shown) to reference a policy in the remote database.

text_query

Specify the query expression to be used as criteria for selecting rows.

score_sorted

Specify whether the results are sorted by score.

The default is `FALSE`.

other_cols

Specify a comma separated list of the table columns (up to 5) to be displayed, in addition to document ID and score, in the hitlist.

struct_query

Specify the structured WHERE condition related to *text_query*. This WHERE condition can include a subquery that selects rows from a structured data column in another table.

Returns

Cursor ID.

Examples

```
declare
    cid number;
begin
    cid := ctx_query.open_con('MYPOL', 'dog', score_sorted =>true, struct_query
        => 'id < 900');
end;
```

In this example, the structured condition specifies that ConText must return the documents that contain *dog* and where the document id is greater than 900.

See Also: CTX_QUERY.FETCH_HIT.

PKDECODE

The `CTX_QUERY.PKDECODE` function extracts and returns a composite textkey element from a composite textkey string.

This function is useful for in-memory queries when querying against a composite textkey table. Use `PKDECODE` to extract textkey columns from the primary key returned by `CTX_QUERY.FETCH_HIT`.

Syntax

```
CTX_QUERY.PKDECODE(  
                    encoded_tk  IN VARCHAR2,  
                    which       IN NUMBER)  
RETURN VARCHAR2;
```

encoded_tk

Specify the encoded composite textkey string

which

Specify the ordinal position of which primary key to extract from *encoded_tk*. When *which* is 0 or a number greater than the number of textkeys in *encoded_tk*, *encoded_tk* is returned.

Returns

String that represents the decoded value of the composite textkey.

Examples

```
declare pkey varchar2(64);  
begin  
  pkey := ctx_query.pkdecode('p1,p2,p3', 2)  
  pkey := ctx_query.pkdecode('p1,p2,p3', 0)  
  pkey := ctx_query.pkdecode('p1,p2,p3', 5)  
end;
```

In this example, the value for the textkey is *p1,p2,p3*. The first call to `PKDECODE` returns the value *p2*. The second and third calls to `PKDECODE` specify ordinal positions that don't exist, thus these calls return the same value, which is the concatenated value *p1,p2,p3*.

PKENCODE

The `CTX_QUERY.PKENCODE` function converts a composite textkey list into a single string and returns the string.

The string created by `PKENCODE` can be used as the primary key parameter `PK` in other `ConText` procedures, such as `CTX_LING.REQUEST_GIST`.

Syntax

```
CTX_QUERY.PKENCODE(  
    pk1      IN VARCHAR2,  
    pk2      IN VARCHAR2,  
    pk4      IN VARCHAR2,  
    pk5      IN VARCHAR2,  
    pk6      IN VARCHAR2,  
    pk7      IN VARCHAR2,  
    pk8      IN VARCHAR2,  
    pk9      IN VARCHAR2,  
    pk10     IN VARCHAR2,  
    pk11     IN VARCHAR2,  
    pk12     IN VARCHAR2,  
    pk13     IN VARCHAR2,  
    pk14     IN VARCHAR2,  
    pk15     IN VARCHAR2,  
    pk16     IN VARCHAR2)  
RETURN VARCHAR2;
```

pk1-pk16

Each `PK` argument specifies a column element in the composite textkey list. You can encode at most 16 column elements.

Returns

String that represents the encoded value of the composite textkey.

Examples

```
exec ctx_ling.request_gist('my_policy',\  
    CTX_QUERY.PKENCODE('pk1-date', 'pk2-data'), 'theme table')
```

In this example, *pk1-date* and *pk2-data* constitute the composite textkey value for the document.

PURGE_SQE

The CTX_QUERY.PURGE_SQE procedure removes all session stored query expressions for the current session. Session SQEs in other sessions are not affected by PURGE_SQE.

Syntax

```
CTX_QUERY.PURGE_SQE(policy_name IN VARCHAR2);
```

policy_name

Specify the name of the policy for which the current session SQEs are purged.

Examples

```
exec ctx_query.purge_sqe(my_pol)
```

REFRESH_SQE

The CTX_QUERY.REFRESH_SQE procedure re-executes a stored query expression and stores the results in the SQR table, overwriting existing results.

See Also: For more information about the structure of the SQR table, see *Oracle8 ConText Cartridge Administrator's Guide*.

Syntax

```
CTX_QUERY.REFRESH_SQE(  
    policy_name  IN VARCHAR2,  
    query_name   IN VARCHAR2);
```

policy_name

Specify the policy for the stored query expression.

query_name

Specify the name of the stored query expression to be refreshed.

Examples

```
exec ctx_query.refresh_sqe('my_pol', 'DOG')
```


RELTAB

The `CTX_QUERY.RELTAB` procedure releases a table previously allocated by `CTX_QUERY.GETTAB`.

Syntax

```
CTX_QUERY.RELTAB(tab IN VARCHAR2);
```

tab

Specify the name of table to be released, previously assigned by `CTX_QUERY.GETTAB`.

Examples

```
set serveroutput on
declare
    mytab varchar2(32) ;
begin
    ctx_query.gettab(CTX_QUERY.HITTAB, mytab, 3) ;
    dbms_output.put_line('table : '||mytab) ;

    ....

    ctx_query.reltab(mytab);
end ;
```

This PL/SQL example allocates a HITTAB result table with GETTAB, then releases it with RELTAB.

REMOVE_SQE

The CTX_QUERY.REMOVE_SQE procedure removes a specified stored query expression from the system SQE table and the results of the SQE from the SQR table for the policy.

See Also: For more information about the structure of the SQE and SQR tables, see *Oracle8 ConText Cartridge Administrator's Guide*.

Syntax

```
CTX_QUERY.REMOVE_SQE(  
    policy_name      IN VARCHAR2,  
    query_name       IN VARCHAR2);
```

policy_name

Specify the policy for the stored query expression.

query_name

Specify the name of the stored query expression to be removed.

Examples

```
exec ctx_query.remove_sqe('my_pol', 'DOG')
```

STORE_SQE

The `CTX_QUERY.STORE_SQE` procedure executes a query for a policy and stores the named SQE in the SQE table and results from the SQE in the SQR table for the policy.

See Also: For more information about the structure of the SQE and SQR tables, see *Oracle8 ConText Cartridge Administrator's Guide*.

Syntax

```
CTX_QUERY.STORE_SQE(  
    policy_name      IN VARCHAR2,  
    query_name       IN VARCHAR2,  
    text_query       IN VARCHAR2,  
    scope            IN VARCHAR2);
```

policy_name

Specify the policy for the stored query expression.

query_name

Specify the name of the stored query expression to be created.

text_query

Specify the query expression.

scope

Specify whether the SQE is a *session* or *system*. When you specify *session*, the stored query expression exists only for the current session. When you specify *system*, the stored query expression can be used in all sessions including concurrent sessions. SQEs defined as *system* are not deleted when your session terminates.

Examples

```
exec ctx_query.store_sqe('my_pol', 'DOG', '$(dogs|puppy)', \  
    'session')
```

Notes

SQEs support all of the ConText query expression operators, *except for*:

- `max`

- first/next

SQEs also support all of the special characters and other components that can be used in a query expression, including PL/SQL functions and other SQEs.

CTX_LING:Linguistics

CTX_LING is the package of PL/SQL procedures used to request linguistic output and to control how requests are submitted and processed by ConText servers with the Linguistics personality.

CTX_LING contains the following stored procedures and functions:

Name	Description
CANCEL	Cancels all cached theme and gist requests.
GET_COMPLETION_CALLBACK	Returns the completion callback procedure specified for the current session.
GET_ERROR_CALLBACK	Returns the error callback procedure specified for the current session.
GET_FULL_THEMES	Returns TRUE when theme hierarchy generation is enabled for the current session.
GET_LOG_PARSE	Returns TRUE when parse logging is enabled for current session.
GET_SETTINGS_LABEL	Returns the currently active setting configuration.
REQUEST_GIST	Requests gists for a document.
REQUEST_THEMES	Requests themes for a document.
SET_COMPLETION_CALLBACK	Specifies a procedure to be called when a request completes.
SET_ERROR_CALLBACK	Specifies a procedure to be called if an error is encountered by a request.
SET_FULL_THEMES	Enables/disables the writing of theme hierarchy information.
SET_LOG_PARSE	Enables/disables logging of parse information for the current session.
SET_SETTINGS_LABEL	Specifies a setting configuration for the current session.
SUBMIT	Submits all cached theme and gist requests to Services Queue.

CANCEL

The `CTX_LING.CANCEL` procedure cancels all pending linguistic requests cached in memory.

Syntax

```
CTX_LING.CANCEL ;
```

Examples

```
exec ctx_ling.cancel
```

Notes

Requests for themes and gists are cached in memory until `CTX_LING.SUBMIT` is called. `CTX_LING.CANCEL` only cancels these cached requests. After these requests have been submitted and placed in the Service Queue, `CTX_LING.CANCEL` has no effect.

To cancel requests that have already been submitted to the Services Queue, use `CTX_SVC.CANCEL`.

GET_COMPLETION_CALLBACK

The `CTX_LING.GET_COMPLETION_CALLBACK` function returns the name of the completion callback procedure for the current session (specified in `CTX_LING.SET_COMPLETION_CALLBACK`).

Syntax

```
CTX_LING.GET_COMPLETION_CALLBACK          RETURN VARCHAR2;
```

Returns

Completion callback procedure.

Examples

```
declare callback varchar2(60);
begin
    callback := get_completion_callback;
    dbms_output.put_line('Completion callback:');
    dbms_output.put_line(callback);
end;
```

Notes

To call procedures for both completed task processing as well as error processing, you must also identify the error completion processing routine with `CTX_LING.SET_COMPLETION_CALLBACK`.

If both completion and error callback procedures are defined, the completion callback routine is performed first, then the error callback routine.

The value assigned to `VARCHAR2` in the declarative part of the PL/SQL block depends on the length of the name for the specified completion callback.

GET_ERROR_CALLBACK

The CTX_LING.GET_ERROR_CALLBACK function returns the name of the error callback procedure for the current session (specified in CTX_LING.SET_ERROR_CALLBACK).

Syntax

```
CTX_LING.GET_ERROR_CALLBACK      RETURN VARCHAR2;
```

Returns

Error callback procedure.

Examples

```
declare e_callback varchar2(60);
begin
    e_callback := ctx_ling.get_error_callback;
    dbms_output.put_line('Error callback:');
    dbms_output.put_line(e_callback);
end;
```

Notes

If both completion and error callback are set, the completion callback is performed first, then the error callback.

The value assigned to VARCHAR2 in the declarative part of the PL/SQL block depends on the length of the name for the specified completion callback.

GET_FULL_THEMES

This function returns TRUE if the generation of theme hierarchy information is enabled for the current session; otherwise it returns FALSE.

You enable the generation of theme hierarchy information with SET_FULL_THEMES. ConText writes theme hierarchy information to the THEME column of the theme table when you call REQUEST_THEMES.

Syntax

```
CTX_LING.GET_FULL_THEMES RETURN BOOLEAN;
```

Returns

Returns TRUE if the generation of theme hierarchy information is enabled; otherwise returns FALSE.

GET_LOG_PARSE

The CTX_LING.GET_LOG_PARSE function returns a FALSE or TRUE string to indicate whether parse logging is enabled for the current database session (specified in CTX_LING.SET_LOG_PARSE).

Syntax

```
CTX_LING.GET_LOG_PARSE      RETURN BOOLEAN;
```

Returns

TRUE if parse logging is enabled, FALSE if parse logging is not enabled.

Examples

```
declare parse_logging boolean;
begin
    parse_logging := get_log_parse;
end;
```

GET_SETTINGS_LABEL

The CTX_LING.GET_SETTINGS_LABEL function returns the label for the setting configuration that is active for the current session (specified in CTX_LING.SET_SETTINGS_LABEL).

Syntax

```
CTX_LING.GET_SETTINGS_LABEL RETURN VARCHAR2;
```

Returns

Current settings configuration label.

Examples

```
declare settings varchar2(60);
begin
    settings := get_settings_label;
    dbms_output.put_line('Current setting configuration:');
    dbms_output.put_line(settings);
end;
```

Notes

The value assigned to VARCHAR2 in the declarative part of the PL/SQL block depends on the character length of the label for the specified setting configuration. The maximum length of a setting configuration label is 80 characters.

REQUEST_GIST

Use the `CTX_LING.REQUEST_GIST` procedure to generate theme summaries and a Gist for a document. You can generate paragraph-level or sentence-level Gists and theme summaries.

By default, this procedure generates theme summaries for all the themes in a document (up to 16); however, you can specify a single theme for which a theme summary is to be generated.

Syntax

```
CTX_LING.REQUEST_GIST(  
    policy    IN VARCHAR2,  
    pk        IN VARCHAR2,  
    table     IN VARCHAR2,  
    glevel    IN VARCHAR2 DEFAULT 'PARAGRAPH',  
    pov       IN VARCHAR2 DEFAULT NULL);
```

policy

Specify the name of the ConText policy on the column.

pk

Specify the primary key (textkey) of the document (row) to be processed. The parameter *pk* can be a single column textkey or an encoded specification for a multiple column textkey.

table

Specify the table used to store the gist output.

glevel

Specify the type of Gist/theme summary to produce. The possible values are:

- *paragraph*
- *sentence*

The default is *paragraph*.

pov

Specify the theme for which a single Gist or theme summary is generated. The type of Gist/theme summary generated (sentence-level or paragraph-level) depends on the value specified for *glevel*.

To generate a Gist for the document, specify a theme of 'GENERIC' for *pov*. To generate a theme summary for the document, specify the theme from the document for which the matching paragraphs/sentences are selected.

If you specify a NULL value for *pov*, ConText generates a Gist for the document and a theme summary for each of the document themes (up to 16).

Note: The *pov* parameter is case sensitive. To return a Gist for a document, specify 'GENERIC' in all uppercase. To return a theme summary, specify the theme *exactly* as it is generated for the document.

The themes generated by CTX_LING.REQUEST_THEMES can be used as input for *pov*.

Examples

```
exec ctx_ling.request_gist('my_pol', '34', 'ctx_gist')

begin
ctx_ling.request_gist('doc_pol',
    CTX_QUERY.PKENCOD('Jones', 'Naval Inst Pr', '10-1-1970'),
    'CTX_GIST');
end;
```

Theme Summary Generation for a Single Theme

In the following example, a single, paragraph-level theme summary is generated for a document with a *pk* of 1442 stored in the text column for policy *my_pol*. The theme (*pov*) for which the theme summary is generated is *Oracle Corporation*:

```
exec ctx_ling.request_gist('my_pol', '1442', 'ctx_gist', pov=>'Oracle Corporation')
```

Sentence-level Gist

In the following example, a sentence-level Gist is generated for document with a *pk* of 1442 stored in the text column for policy *my_pol*:

```
exec ctx_ling.request_gist('my_pol', '1442', 'ctx_gist', 'sentence', 'GENERIC')
```

Notes

You must call the CTX_LING.REQUEST_GIST procedure once for each document for which you want to generate gists.

By default, ConText linguistics generates up to 16 themes for a document. If the user settings specify that gists are to be created for only the top 10 themes of the document, the REQUEST_GIST procedure creates a total of 11 gists: one gist for the specified number of themes and one generic gist for the entire document.

The REQUEST_GIST procedure only creates gists if the setting configuration for the session in which REQUEST_GIST is called supports gist generation.

The parameter *pk* can be either a single column textkey or a multiple column (composite) textkey. When *pk* is a composite textkey, you must encode the composite textkey string using the CTX_QUERY.PKENCODER procedure as in the second example above.

Requests are not automatically entered into the Services Queue; each request is cached in memory until the application calls the CTX_LING.SUBMIT procedure.

CTX_LING.SUBMIT explicitly enters all of the cached requests into the Services Queue as a single batch.

All of the linguistic settings that can be specified for Gist-generation also apply to sentence-level Gists/theme summaries when requested. The settings simply act on sentences rather than paragraphs.

For example, the *size* setting for Gists, which determines the maximum number of paragraphs in a paragraph-level Gist, determines the maximum number of sentences in a sentence-level Gist, when a sentence-level Gist is requested.

See Also: For more information about the *size* setting, as well as the other settings that can be specified for Gists and theme summaries, see the help system provided with the ConText System Administration tool.

REQUEST_THEMES

The CTX_LING.REQUEST_THEMES procedure generates a list of up to sixteen themes for a document.

Syntax

```
CTX_LING.REQUEST_THEMES(  
    policy      IN VARCHAR2,  
    pk          IN VARCHAR2,  
    table       IN VARCHAR2);
```

policy

Specify the name of the ConText policy for the column.

pk

Specify the primary key (textkey) of the document (row) to be processed. The parameter *pk* can be a single column textkey or an encoded specification for a multiple column textkey.

table

Specify the table used to store the theme output.

Examples

```
exec ctx_ling.request_themes('my_pol', 34, 'CTX_THEMES')  
  
begin  
    ctx_ling.request_themes('doc_pol',  
        CTX_QUERY.PKENCODE('Jones','Naval Inst Pr','10-1-1970'),  
        'CTX_THEMES');  
end;
```

Notes

You must call CTX_LING.REQUEST_THEMES procedure once for each document for which you want to generate themes.

The parameter *pk* can be either a single column textkey or a multiple column textkey. When *pk* is a composite key, you must encode the composite textkey string using the CTX_QUERY.PKENCODE procedure as in the second example above.

Requests for themes are not automatically entered into the Services Queue; each request is cached in memory pending submission by `CTX_LING.SUBMIT`.

`CTX_LING.SUBMIT` explicitly enters all of the cached requests into the Services Queue as a single batch.

SET_COMPLETION_CALLBACK

The `CTX_LING.SET_COMPLETION_CALLBACK` procedure specifies the user-defined PL/SQL processing routine (usually a procedure) to be called when a Con-Text server finishes processing a request in the Services Queue.

Syntax

```
CTX_LING.SET_COMPLETION_CALLBACK(callback_name IN VARCHAR2);
```

callback_name

Specify the name of the callback procedure. See below for a description of the arguments to the *callback_name* procedure.

Examples

```
exec ctx_ling.set_completion_callback('COMP_PROCEDURE')
```

Notes

A completion callback procedure must be defined before `SET_COMPLETION_CALLBACK` can be called. The completion callback procedure must accept the following arguments:

Argument	Type	Purpose
HANDLE	NUMBER	Specify the internal identifier for the request, as returned by <code>SUBMIT</code> .
STATUS	VARCHAR2	Specify the status of the request: <code>SUCCESS</code> or <code>ERROR</code> .
ERRCODE	VARCHAR2	Specify the code for the error (<code>NULL</code> if request processed successfully).

Control is passed to the `SET_COMPLETION_CALLBACK` procedure at the completion of a linguistic request. It can log errors or otherwise notify the application when a request has finished processing. This can be particularly useful for a large job that is run asynchronously in batch mode.

To call a procedure specifically for requests that terminate with errors, use `CTX_LING.SET_ERROR_CALLBACK`.

SET_ERROR_CALLBACK

The CTX_LING.SET_ERROR_CALLBACK procedure specifies the user-defined PL/SQL processing routine (usually a procedure) to be called when a ConText server encounters an error while processing a linguistic request.

Syntax

```
CTX_LING.SET_ERROR_CALLBACK(callback_name IN VARCHAR2);
```

callback_name
Specify the name of the callback procedure to be used when an error occurs.

Examples

```
exec ctx_ling.set_error_callback('ERROR_PROCEDURE')
```

Notes

An error callback procedure must be defined before SET_ERROR_CALLBACK can be called. The error callback procedure must accept the following arguments:

Argument	Type	Purpose
HANDLE	NUMBER	Specify the internal identifier for the request, as returned by SUBMIT
ERRCODE	VARCHAR2	Specify the code for the error.

Control is passed to the SET_ERROR_CALLBACK procedure at the completion of a linguistic request. The procedure can be used to log errors or otherwise notify the application when a request has finished processing. This can be particularly useful for a large job that is run asynchronously in batch mode.

To call a procedures for both completed task processing and error processing, use SET_COMPLETION_CALLBACK.

SET_FULL_THEMES

Use this procedure to enable the writing of theme hierarchy information to the theme table. ConText writes the theme hierarchy information when you call REQUEST_THEMES. (By default, ConText writes only theme phrase information to the theme table when you call REQUEST_THEMES.)

Syntax

```
CTX_LING.SET_FULL_THEMES (theme_mode IN BOOLEAN DEFAULT TRUE);
```

theme_mode

Specify TRUE for ConText to write theme hierarchy information to the THEME column of the theme table.

Specify FALSE to disable the writing of theme hierarchy information to the THEME column of the theme table.

Notes

At the start of a session, the *theme_mode* flag is FALSE.

Calling SET_FULL_THEMES without an argument is the same as calling this procedure with *theme_mode* set to TRUE.

You can check whether the writing of theme hierarchy information is turned on using GET_FULL_THEMES.

SET_LOG_PARSE

The CTX_LING.SET_LOG_PARSE procedure enables/disables logging of linguistic parsing information for a session.

Syntax

```
CTX_LING.SET_LOG_PARSE(log_mode BOOLEAN DEFAULT TRUE);
```

log_mode

Specify whether to write parse information to a log file during linguistic processing in a session. The default is TRUE.

Examples

```
exec ctx_ling.set_log_parse(TRUE)
```

Notes

At start-up of a ConText server, parse information logging is disabled.

Once logging is enabled, it stays enabled for the session until it is explicitly disabled.

When logging is enabled, the text of the document being parsed and the paragraph offset information used by ConText to separate the document into its constituent paragraphs is written to the log file specified when the ConText server is started.

The log provides information about the input text used to generate linguistic output and can be used for debugging the system. The parse information is especially useful for debugging linguistic output for formatted documents from which the text is extracted before it is processed.

However, due to the large amount of information generated by ConText and written to the log file, parse logging may affect performance considerably. For this reason, you should only enable parse logging if you encounter problems with linguistics.

SET_SETTINGS_LABEL

Use the `CTX_LING.SET_SETTINGS_LABEL` procedure to change the linguistic settings for a database session.

Syntax

```
CTX_LING.SET_SETTINGS_LABEL(settings_label IN VARCHAR2);
```

settings_label

Specify the label for the setting configuration used for the session. You can use one of the following predefined settings or one that you create with the administration tool:

Label	Description
GENERIC	Use this configuration to analyze mixed-case English text to produce theme and Gist output. This configuration is the default.
SA	This configuration is identical to GENERIC, except it converts all-upper-case or all lower-case text to mixed case before processing text to produce theme or Gist output. This setting should be used only when text is all-uppercase or all-lower-case, or where you are not sure of the accuracy of the case.

Examples

```
exec ctx_ling.set_settings_label('SA')
```

Notes

At start-up of a ConText server, the GENERIC default setting configuration is active.

The settingspecified by `SET_SETTINGS_LABEL` is active for the entire session or until you call `SET_SETTINGS_LABEL` with a new setting configuration. In addition, the specified setting is active only for your current session; settings specified for your session have no effect on the server setting.

You can specify any predefined ConText setting configuration or any custom setting configuration. Define custom setting configurations with the Administration Tool provided with ConText Workbench.

When your text is all upper-case or all lower-case and you use the SA setting to convert the text to mixed-case, Oracle Corporation does not recommend creating theme indexes or issuing theme queries. Creating theme indexes with the SA linguistic setting does not produce consistent results.

SUBMIT

The CTX_LING.SUBMIT procedure creates a single request (row) in the Services Queue for all linguistic requests cached in memory for a single row (identified by PK) and returns a handle for the request.

Syntax

```
CTX_LING.SUBMIT(  
    wait          IN NUMBER  DEFAULT 0,  
    do_commit     IN BOOLEAN DEFAULT TRUE,  
    priority      IN NUMBER  DEFAULT 0)  
RETURN NUMBER;
```

wait

Specify maximum time in seconds to block subsequent requests while ConText server processes request. The default is 0.

do_commit

Specify whether the job request should be committed to the database. The default is TRUE.

priority

Specify the priority for the request. Requests are processed in order of priority from lowest priority to highest priority. The default is 0.

Returns

Handle that identifies the request.

Examples

```
declare handle number;  
begin  
    handle := ctx_ling.submit(500);  
end;
```

In this example, procedures to create one or more gists and/or themes have already been executed and the requests cached in memory. The SUBMIT procedure enters the request(s) into the Services Queue and returns a handle. In this case, it also pre-

vents the queue from accepting other submissions from the same requestor for 500 seconds.

Notes

SUBMIT does not cache requests for multiple documents nor for documents in different columns. Only requests for a single document at a time can be submitted.

If more than one request is queued in memory, SUBMIT processes all of the requests as a single batch job. If the request is a batch job, the ConText server processes each request in the batch in order.

All of the individual requests in the batch must be processed successfully or the ConText server returns an ERROR status for the entire batch. The error message stack returned by the ConText server identifies the request that caused the batch to fail.

If SUBMIT is called from a database trigger, the DO_COMMIT argument must be set to FALSE.

CTX_SVC: Services Queue Administration

The CTX_SVC package contains PL/SQL procedures used to query requests in the Services Queue and to perform administrative tasks on the Queue.

CTX_SVC contains the following stored procedures and functions:

Name	Description
CANCEL	Removes a pending request from the Services Queue.
CANCEL_ALL	Removes all pending requests from the Services Queue.
CANCEL_USER	Removes a pending request from the Services Queue for the current user.
CLEAR_ALL_ERRORS	Removes all requests with an error status from the Services Queue.
CLEAR_ERROR	Removes a request that produced an error from the Services Queue.
CLEAR_INDEX_ERRORS	Removes errored indexing requests from the Services Queue.
CLEAR_LING_ERRORS	Removes errored linguistic requests from the Services Queue.
REQUEST_STATUS	Returns the status of a request in the Services Queue.

CANCEL

The `CTX_SVC.CANCEL` procedure removes a request from the Services Queue, if the request has a status of `PENDING`.

Syntax

```
CTX_SVC.CANCEL(request_handle NUMBER);
```

request_handle

Specify the handle, returned by `CTX_LING.SUBMIT`, of the service request to remove.

Examples

```
exec ctx_svc.cancel(3321)
```

Notes

To cancel requests that have not been entered in the Services Queue, use the `CTX_LING.CANCEL` procedure.

CANCEL_ALL

The CTX_SVC.CANCEL_ALL procedure removes all requests with a status of PENDING from the Services Queue.

Syntax

```
CTX_SVC.CANCEL_ALL ;
```

Examples

```
execute ctx_svc.cancel_all
```

CANCEL_USER

The `CTX_SVC.CANCEL_USER` procedure removes all requests with a status of `PENDING` for the current user.

Syntax

```
CTX_SVC.CANCEL_USER ;
```

Examples

```
execute ctx_svc.cancel_user
```

CLEAR_ALL_ERRORS

The CTX_SVC.CLEAR_ALL_ERRORS procedure removes all requests (text indexing, theme indexing, and linguistics) that have a status of ERROR in the Services Queue.

Syntax

```
CTX_SVC.CLEAR_ALL_ERRORS ;
```

Examples

```
execute ctx_svc.clear_all_errors
```

CLEAR_ERROR

The `CTX_SVC.CLEAR_ERROR` procedure removes a request with a status of `ERROR` from the Services Queue.

Syntax

```
CTX_SVC.CLEAR_ERROR(request_handle IN NUMBER);
```

request_handle

Specify the handle, returned by `CTX_LING.SUBMIT`, of the errored service request that is to be removed.

Examples

```
exec ctx_svc.clear_error(3321)
```

Notes

When you call `CTX_SVC.CLEAR_ERROR` with a 0 for the `REQUEST_HANDLE`, ConText removes all requests in the Services Queue that have an `ERROR` status.

You can use `CTX_SVC.REQUEST_STATUS` to return the status of a request in the Services Queue.

CLEAR_INDEX_ERRORS

The CTX_SVC.CLEAR_INDEX_ERRORS procedure removes all indexing requests (text and theme) that have a status of ERROR in the Services Queue.

Syntax

```
CTX_SVC.CLEAR_INDEX_ERROR ;
```

Examples

```
execute ctx_svc.clear_index_errors
```

CLEAR_LING_ERRORS

The `CTX_SVC.CLEAR_LING_ERRORS` procedure removes all linguistic requests that have a status of `ERROR` in the Services Queue.

Syntax

```
CTX_SVC.CLEAR_LING_ERROR ;
```

Examples

```
execute ctx_svc.clear_ling_errors
```


REQUEST_STATUS

The `CTX_SVC.REQUEST_STATUS` function returns the status of a request in the Services Queue.

Syntax

```
CTX_SVC.REQUEST_STATUS(  
    request_handle    IN    NUMBER,  
    timestamp         OUT   DATE,  
    errors            OUT   VARCHAR2)  
RETURN VARCHAR2;
```

request_handle

Specify the handle of the service request, as returned by `CTX_LING.SUBMIT`.

timestamp

Returns the time at which request was submitted.

errors

Returns the error message stack for the request; message stack is returned only if the status of the request is `ERROR`.

Returns

Status of the request, which is one of the following:

PENDING

The request has not yet been picked up by a ConText server.

RUNNING

The request is being processed by a ConText server.

ERROR

The request encountered an error (see `ERRORS` argument).

SUCCESS

The request completed successfully.

Examples

```
declare status varchar2(10);
declare time date;
declare errors varchar2(60)
begin
  status := ctx_svc.request_status(3461,timestamp,errors);
  dbms_output.put_line(status,timestamp,substr(errors,1,20));
end;
```

Notes

Specifying an invalid request handle in REQUEST_HANDLE causes CTX_SVC.REQUEST_STATUS to return a status of SUCCESS.

Result Tables

This appendix describes the database schema of the result tables utilized by Context. Result tables are database tables that store results from the `CTX_QUERY.CONTAINS` and `CTX_QUERY.HIGHLIGHT` procedures as well as the output from linguistic procedures, `CTX_LING.REQUEST_THEMES` and `CTX_LING.REQUEST_GIST`.

The topics described in this chapter are:

- Hitlist Table Structure
- Highlight Table Structures
- Display Table Structures
- Linguistic Output Table Structures

Hitlist Table Structure

The hitlist result table stores the results returned by the CTX_QUERY.CONTAINS procedure in the first step of a two-step query. The results can be queried directly to produce a hitlist for the query or combined with the base table to produce more detailed hitlists.

A hitlist result table must be created before executing a two-step query. It can be created manually or using CTX_QUERY.GETTAB.

If the hitlist table is created manually, it can be given any name; however, the table must have the following columns (with names and datatypes as specified).

Column Name	Type	Description
TEXTKEY	VARCHAR2(64)	Unique identifier (usually the primary key for the table) for documents that satisfy the two-step query.
SCORE	NUMBER	Score generated by CONTAINS function for each document.
CONID	NUMBER	ID for results returned by CONTAINS function when multiple CONTAINS use the same hitlist result table.

Composite Textkey Hitlist Tables

When you perform a two-step query on a text table that has a composite textkey, the schema of the resulting hitlist table is the same as for when you issue a query on a table with a single column textkey, except that a composite textkey result table has additional TEXTKEY columns.

The number of TEXTKEY columns in the hitlist table match the number of columns in the textkey for the original text table. The TEXTKEY columns in the hitlist table are named *TEXTKEY*, *TEXTKEY2*, *TEXTKEY3*,..., *TEXTKEYN*, where *N* is the number of columns in the textkey in the original text table. *N* is always less than or equal to 16.

For example, if you do a query on a text table that has a four-column composite textkey, the schema of the resulting hitlist table is: TEXTKEY, TEXTKEY2, TEXTKEY3, TEXTKEY4, SCORE, CONID.

The resulting TEXTKEY columns in the hitlist table are populated in the same order as they were registered in the column policy.

Highlight Table Structures

The highlight result tables store the highlighting results returned by the CTX_QUERY.HIGHLIGHT procedure.

Highlight tables must be created before calling HIGHLIGHT to generate highlighting results. They can be created manually or using CTX_QUERY.GETTAB.

If a highlight table is created manually, it can be assigned any name; however, the table must have the columns (with names and datatypes) as specified.

HIGHTAB Highlight Table

The HIGHTAB highlight table stores query term offset and length information for query terms in documents.

If a document is formatted, the text is filtered by CTX_QUERY.HIGHLIGHT into plain text and the offset information is generated for the filtered text. The offset information can be used to highlight query terms in a document.

The table must have the following columns:

Column Name	Type	Description
ID	NUMBER	The identifier for the results generated by a particular call to CTX_QUERY.HIGHLIGHT. Only used when table is used to store results from multiple HIGHLIGHTS.
OFFSET	NUMBER	The position of the query terms in the document, relative to the rest of the terms in the documents. Measured from a base of 1.
LENGTH	NUMBER	The length of the query term.
STRENGTH	NUMBER	The strength of the highlight table.

MUTAB Highlight Table

The MUTAB display table stores documents in plain text (ASCII) format with the query terms in the documents highlighted by mark-up tags generated by CTX_QUERY.HIGHLIGHT. This mark-up can be used to provide an ASCII version of the document with query terms highlighted.

The highlighting mark-up tags can be specified when HIGHLIGHT is called or the default mark-up tags can be used.

Note: For HTML documents filtered through the internal HTML filter, the MUTAB stores the document with the original HTML tags.

The table must have the following columns:

Column Name	Type	Description
ID	NUMBER	The identifier for the results generated by a particular call to CTX_QUERY.HIGHLIGHT (only used when table is used to store results from multiple HIGHLIGHTS)
DOCUMENT	LONG	Marked-up text of the document, stored in ASCII format

ICFTAB Highlight Table

The ICFTAB highlight table stores the ICF output generated by CTX_QUERY.HIGHLIGHT.

Note: ICF output is used primarily by the Windows viewer control to provide WYSIWIG viewing of documents in the supported formats. As such, it is stored as binary data in a LONG RAW column and is generally inaccessible to users.

The table must have the following columns:

Column Name	Type	Description
ID	NUMBER	The identifier for the results generated by a particular call to CTX_QUERY.HIGHLIGHT (only used when table is used to store results from multiple HIGHLIGHTS)
DOCUMENT	LONG RAW	Text of the document, stored in ICF format

Display Table Structures

The display result tables store the display results returned by the CTX_QUERY.HIGHLIGHT procedure. The display results can be either the document in its original format or the document filtered to plain (ASCII) text.

Display result tables must be created before calling HIGHLIGHT to generate display output. They can be created manually or using CTX_QUERY.GETTAB.

If a display table is created manually, it can be assigned any name; however, the table must have the columns (with names and datatypes) as specified.

NOFILTAB Display Table

The NOFILTAB display table stores formatted documents in their native format (i.e. WordPerfect, Microsoft Word, HTML, ASCII). No highlighting or filtering is performed on the text of the document.

The NOFILTAB table must have the following columns:

Column Name	Type	Description
ID	NUMBER	The identifier for the results generated by a particular call to CTX_QUERY.HIGHLIGHT (only used when table is used to store results from multiple HIGHLIGHTS)
DOCUMENT	LONG RAW	Text of the document, stored in the original format

PLAINTAB Display Table

The PLAINTAB display table stores documents in plain text (ASCII) format. The documents are processed through the filter defined for the text column and the results are stored in the PLAINTAB table.

The PLAINTAB table must have the following columns:

Column Name	Type	Description
ID	NUMBER	The identifier for the results generated by a particular call to CTX_QUERY.HIGHLIGHT (only used when table is used to store results from multiple HIGHLIGHTS)
DOCUMENT	LONG	Text of the document, stored in ASCII format

Linguistic Output Table Structures

The output tables store the results returned by the ConText linguistics. The output tables serve only as temporary holding areas for the linguistic output. You modify, augment, or truncate the output into a form best suited for your application.

See Also: For more information about generating linguistic output, see “Generating Linguistic Output” in Chapter 8.

Theme Table

The theme results table stores one row for each theme generated by CTX_LING.REQUEST_THEMES. The value stored in the THEME column is either a theme phrase or a colon separated list of parent themes.

The table can be named anything, but must include the following columns with names and datatypes as specified:

Column Name	Type	Description
CID	NUMBER	Policy ID.
PK	VARCHAR2(64)	Primary key (textkey) for the text table.
THEME	VARCHAR2(2000)	Theme phrase or hierarchical list of parent themes separated by colons (:).
WEIGHT	NUMBER	Weight of theme phrase, relative to other theme phrases for the document.

Composite Textkey Theme Tables

You can use CTX_LING.REQUEST_THEMES to generate themes for a document contained in a composite textkey table. When you do so, the schema of the resulting theme table is the same as for when you request a theme on a single column textkey table, except that the composite textkey result table has additional PK columns.

The number of textkey columns in the theme table match the number of textkey columns in the original text table. The textkey columns in the theme table are named *PK1, PK2, PK3,..., PKN*, where *N* is the number of textkeys in the original text table. *N* is always less than or equal to 16.

For example, if you request a theme on a text table that has four textkeys, the schema of the output table would be (CID, PK1, PK2, PK3, PK4, THEME, WEIGHT).

The resulting textkey columns in the theme table are populated in the same order as they were registered.

Gist Table

The Gist result table stores one row for each Gist generated by CTX_LING.REQUEST_GIST.

The table can be named anything, but must include the following columns (with names and datatypes as specified):

Column Name	Type	Description
CID	NUMBER	Policy ID.
PK	VARCHAR2(64)	Primary key (textkey) for the text table.
POV	VARCHAR2(80)	Document theme.
GIST	LONG	ASCII text of Gist or theme summary.

The value in the POV column for a theme summary is a string which identifies the theme in the document.

The value in the POV column for a Gist is the term GENERIC.

Note: GENERIC is the only value that is consistently in all-upper-case. For all other themes in the POV column, the case depends on how the themes were used in the document.

Composite Textkey Gist Tables

You can use CTX_LING.REQUEST_GIST to generate Gists for a document contained in a composite textkey table. When you do so, the schema of the resulting Gist table is the same as for when you request a Gist on a single column textkey table, except that the composite textkey result table has additional PK columns.

The number of textkey columns in the Gist table match the number of textkey columns in the original text table. The textkey columns in the Gist table are named *PK1*, *PK2*, *PK3*, ..., *PKN*, where *N* is the number of textkeys in the original text table. *N* is always less than or equal to 16.

For example, if you request a Gist on a text table that has four textkeys, the schema of the resulting hitlist table is (CID, PK1, PK2, PK3, PK4, POV, GIST).

The resulting textkey columns in the Gist table are populated in the same order as they were registered.

SQL*Plus Sample Code

This appendix describes the sample SQL*Plus scripts provided by ConText. The scripts illustrate how to use SQL*Plus to build simple queries and generate linguistic output using ConText linguistics.

The scripts are divided into two functional areas: CTXPLUS (performing ad-hoc queries) and CTXLING (generating linguistic output).

The following topics are covered in this chapter:

- Setting Up the ConText Sample Applications
- Overview of CTXPLUS
- Overview of CTXLING

Setting Up the ConText Sample Applications

Before you can use either the CTXPLUS or CTXLING, as well as the Oracle Forms sample application, you must create the required demonstration objects by performing the following setup tasks.

Note: The files required for performing the setup tasks are located in the demo directory for ConText. For example, in a UNIX environment, the files are named *demo.dmp* and *demo_setup.sql* and are located in *\$ORACLE_HOME/ctx/demo/install*.

For the exact location and name of the setup files, see the Oracle8 installation documentation specific to your operating system.

1. Import the export file into the predefined ConText user CTXDEMO's schema. For example:

```
IMP ctxdemo/ctxdemo FILE=demo.dmp TABLES=articles
```

Importing the export file creates an ARTICLES table for CTXDEMO and populates ARTICLES.TEXT with the text of the articles used in the samples.

2. Start one or more ConText Server with the DDL (D) personality.
3. Log in to SQL*Plus as the demo user and run the setup script. For example:

```
@demo_setup
```

The script creates the policies, preferences, views, and results tables used by the samples and creates a text index for the ARTICLES table.

This script illustrates the tasks you must perform to set up your tables for processing queries using ConText.

Overview of CTXPLUS

The CTXPLUS sample code consists of the following SQL scripts:

Script	Description
query1.sql	Performs a one-step query using the input query expression and returns a hitlist, sorted by score, to the standard output.
query2.sql	Performs a two-step query using the input query expression and returns a hitlist, sorted by score, to the standard output.
queryc.sql	Performs an in-memory query using the input query expression and returns an <i>unsorted</i> hitlist to standard output
querys.sql	Performs an in-memory query using the input query expression and returns a hitlist, sorted by score, to the standard output.
storeqry.sql	Performs a query and stores the results as a system SQE. The results of the SQE can then be used in a query (one-step, two-step, or in-memory).
showsqe.sql	Returns a list of all the system SQEs that have been stored for a policy. Note that this script is <i>not</i> currently implemented.
view.sql	Selects a document based on the input textkey and returns the text of the document to the standard output.

See Also: For more information about the location of the scripts, see the Oracle8 Server installation documentation specific to your operating system.

Concepts

The ConText concepts illustrated in this sample code are:

- query expression syntax
- one-step queries
- two-step queries
- two-step queries (sorted and unsorted)
- stored query expressions

Using CTXPLUS

To use the CTXPLUS sample SQL scripts:

1. Ensure that one or more ConText servers are running with the Query (Q) personality.
2. Log in to SQL*Plus as the owner of the demonstration objects (usually CTX-DEMO).
3. To initiate a query, run one of the query scripts (query1, query2, queryc, or querys). The scripts prompt you to enter a query expression.

For example:

```
@query1
Enter value for query_terms: coffee|tea
```

The script then returns a hitlist of the documents in the ARTICLES table that satisfy the query expression you enter. The hitlist consists of a score, ID, author, and title.

4. To view an article, run the view.sql script and give it an article ID. The article ID is the value displayed in the ID column in the hitlist generated by the query scripts.

For example:

```
@view 14
```

The script then returns the text for the document with the article ID you specified.

5. To create a stored query expression (SQE), run the storeqry.sql script. The scripts prompt you to enter a name for the SQE and a query expression.

For example:

```
@storeqry
Enter query name: test_sqe
Enter value for query_terms: coffee|tea
```

Note: The script does not return the results of the query to the standard output.

To view the SQEs for the demonstration user, use the CTX_USER_SQES view.

For example:

```
select pol_name, query_name, query_name
from ctx_user_sqes;
```

CTXPLUS Examples

The following examples execute the query1.sql, query2.sql, and queries.sql scripts using the query terms *California* and *politics* and various logical operators (OR, ACCUMULATE, and AND).

These examples illustrate how one-step, two-step, and (sorted) in-memory queries produce the same results and how the operators in a query expression affect the rows and scores returned by a query:

Single Term Queries

@query2

Enter value for query_terms: California

SCR	ID	AUTHOR	TITLE
100	17	Nolo Richards	REVIEW & OUTLOOK (Editorial): California Smashup
50	18	Nolo Richards	State Farm and California
30	25	David Shribman	In the Wilderness: Democrats' Troubles In Winning
20	49	Nolo Richards	California High Court Is Asked to Lift Block Of In
10	16	Heidi Waleson	LEISURE & ARTS: Cynthia Phelps: Violist in Vogue

@query1

Enter value for query_terms: politics

SCR	ID	AUTHOR	TITLE
20	25	David Shribman	In the Wilderness: Democrats' Troubles In Winning
10	13	Frederick C. Kl	LEISURE & ARTS -- Sports: Mediocrity's the Word Ar

Multiple Term Query Using OR

@query5

Enter value for query_terms: politics|California

SCR	ID	AUTHOR	TITLE
100	17	Nolo Richards	REVIEW & OUTLOOK (Editorial): California Smashup
50	18	Nolo Richards	State Farm and California
30	25	David Shribman	In the Wilderness: Democrats' Troubles In Winning
20	49	Nolo Richards	California High Court Is Asked to Lift Block Of In
10	13	Frederick C.	Kl LEISURE & ARTS -- Sports: Mediocrity's the Word Ar
10	16	Heidi Waleson	LEISURE & ARTS: Cynthia Phelps: Violist in Vogue

Multiple Term Query Using ACCUMULATE

@query1

Enter value for query_terms: politics,California

SCR	ID	AUTHOR	TITLE
100	17	Nolo Richards	REVIEW & OUTLOOK (Editorial): California Smashup
50	18	Nolo Richards	State Farm and California
50	25	David Shribman	In the Wilderness: Democrats' Troubles In Winning
20	49	Nolo Richards	California High Court Is Asked to Lift Block Of In
10	13	Frederick C.	Kl LEISURE & ARTS -- Sports: Mediocrity's the Word Ar
10	16	Heidi Waleson	LEISURE & ARTS: Cynthia Phelps: Violist in Vogue

Multiple Term Queries Using AND

@query2

Enter value for query_terms: politics&California

SCR	ID	AUTHOR	TITLE
20	25	David Shribman	In the Wilderness: Democrats' Troubles In Winning

Overview of CTXLING

The CTXLING demo is a set of simple, related SQL*Plus scripts. Two of the scripts automate and track linguistic extraction on the demonstration documents. The remaining scripts can be used to query this linguistic output.

The CTXLING sample code consists of the following SQL scripts:

Script	Description
genling.sql	Requests theme and Gist generation for each of the documents in the ARTICLES table.
status.sql	Shows the status of the theme and Gist generation initiated by genling.sql.
gist.sql	Displays the Gists for a document.
themes.sql	Displays the themes for a document.
similar.sql	Displays documents with similar themes for the input document

See Also: For more information about the location of the scripts, see the Oracle8 Server installation documentation specific to your operating system.

Concepts

The ConText concepts illustrated in this sample code are:

- generating linguistic output using the Linguistic Services
- document theme viewing
- document Gist viewing

Using CTXLING

To use the CTXLING sample SQL scripts:

1. Ensure that one or more ConText servers with the Linguistic (L) personality are running.
2. Log in to SQL*Plus as the owner of the demonstration objects (usually CTX-DEMO).
3. To generate linguistic output, run genling.sql:

```
@genling
Clearing theme table...
Clearing article table...
Initializing ling_tracking table
Creating ling. callback function LING_COMP_CALLBACK...
Submitting all articles for linguistic extraction...
All articles submitted.
```

4. The linguistic generation runs in the background. While this is happening, you can use `status.sql` to check on the progress:

For example:

```
@status
Linguistic Requests left: 36
Request Errors....
```

The extraction is complete when there are 0 Linguistic Requests left.

5. To view the themes or Gists of an article, run the appropriate script and give it an article ID.

For example:

```
@gist 40
Points of View
01 GENERIC ..
15 production
16 purchases
which point of view gist to print: 15
```

The script then returns the themes or Gists for the document with the article ID you specified.

6. To select articles with the same themes as an article, run the `similar.sql` script and give it an article ID.

For example:

```
@similar 14
```

The script then returns a list of the articles with the same themes as the article ID you specified.

CTXLING Examples

The following examples illustrate using `themes.sql`, `gist.sql`, and `similar.sql` to view the linguistic output generated by `genling.sql`.

Theme Viewing

@themes 40

Commodities: Coffee Futures Prices Decline on News That
U.S. Might Not Participate in New International Pact
by John Valentine

T#	THEME	WEIGHT
01	United States	11
02	commerce and trade	10
03	coffee	10
...		

Gist Viewing

@gist 40

Points of View

01 GENERIC ...
15 production
16 purchases

Which point of view gist to print: 15

Commodities: Coffee Futures Prices Decline on News That
U.S. Might Not Participate in New International Pact
by John Valentine

Consuming and producing nations appear to be poles apart
in their positions. Producing countries proposed a quota
that would incorporate the sales of
...

Theme Comparison Viewing

@similar 40

Commodities: Coffee Futures Prices Decline on News That
U.S. Might Not Participate in New International Pact
by John Valentine

Article Themes

- 01 United States
- 02 commerce and trade
- 03 coffee
- ..
- 14 production
- 15 purchases

Which theme to query: 15

Other articles with this theme

ID	WT	AUTHOR	TITLE
1	8	William Power	OTC Focus: Composite Index Falls
33	7	Alex Kaufmann	Your Money Matters: How to Take
5	7	George Anders	Shades of U.S. Steel: J.P.
30	6	Michael Siconol	Mutual Funds: ...And Find Out if
47	6	Nolo Richards	Ponce Federal Bank Is in Talks
45	5	Nolo Richards	Farley Wins Round In His Bid to
35	2	Alix M.	Freedma Supermarkets Push Private-Label

Stopword Transformations

This appendix describes stopwords transformations. The following topic is covered:

- Understanding Stopword Transformations

Understanding Stopword Transformations

When you use a stopwords or stopwords-only phrase as an operand for a query operator, ConText rewrites the expression to eliminate the stopwords or stopwords-only phrase and then executes the query.

The following section describes the stopwords rewrites or transformations for each operator. In all tables, the *Stopword Expression* column describes the query expression or component of a query expression, while the right-hand column describes the way ConText rewrites the query.

The token *stopword* stands for a single stopwords or a stopwords-only phrase.

The token *non_stopword* stands for either a single non-stopwords, a phrase of all non-stopwords, or a phrase of non-stopwords and stopwords.

The token *no_lex* stands for a single character or a string of characters that is neither a stopwords nor a word that is indexed. For example, the + character by itself is an example of a *no_lex* token.

When the *Stopword Expression* column completely describes the query expression, a rewritten expression of *no_token* means that no hits are returned when you enter such a query.

When the *Stopword Expression* column describes a component of a query expression with more than one operator, a rewritten expression of *no_token* means that a *no_token* value is passed to the next step of the rewrite.

Transformations that contain a *no_token* as an operand in the *Stopword Expression* column describe intermediate transformations in which the *no_token* is a result of a previous transformation. These intermediate transformations apply when the original query expression has at least one stopwords and more than one operator.

For example, consider the following compound query expression:

```
'(this NOT dog) AND cat'
```

Assuming that *this* is the only stopwords in this expression, ConText applies the following transformations in the following order:

stopword NOT non-stopword => no_token

no_token AND non_stopword => non_stopword

The resulting expression is:

```
'cat'
```

See Also: To learn more about how to examine stopwords transformations, see Chapter 5, “Query Expression Feedback”.

For more information about defining stopwords, see *Oracle8 Con-Text Cartridge Administrator's Guide*.

Word Transformations

Stopword Expression	Rewritten Expression
<i>stopword</i>	<i>no_token</i>
<i>no_lex</i>	<i>no_token</i>

The first transformation mean that a stopwords or stopwords-only phrase by itself in a query expression results in no hits.

The second transformation says that a term that is not lexed such as + results in no hits.

AND Transformations

Stopword Expression	Rewritten Expression
<i>non_stopword AND stopwords</i>	<i>non_stopword</i>
<i>non_stopword AND no_token</i>	<i>non_stopword</i>
<i>stopword AND non_stopword</i>	<i>non_stopword</i>
<i>no_token AND non_stopword</i>	<i>non_stopword</i>
<i>stopword AND stopwords</i>	<i>no_token</i>
<i>no_token AND stopwords</i>	<i>no_token</i>
<i>stopword AND no_token</i>	<i>no_token</i>
<i>no_token AND no_token</i>	<i>no_token</i>

OR Transformations

Stopword Expression	Rewritten Expression
<i>non_stopword OR stopwords</i>	<i>non_stopword</i>
<i>non_stopword OR no_token</i>	<i>non_stopword</i>
<i>stopword OR non_stopword</i>	<i>non_stopword</i>
<i>no_token OR non_stopword</i>	<i>non_stopword</i>
<i>stopword OR stopwords</i>	<i>no_token</i>
<i>no_token OR stopwords</i>	<i>no_token</i>
<i>stopword OR no_token</i>	<i>no_token</i>
<i>no_token OR no_token</i>	<i>no_token</i>

Accumulate Transformations

Stopword Expression	Rewritten Expression
<i>non_stopword</i> ACCUM <i>stopword</i>	<i>non_stopword</i>
<i>non_stopword</i> ACCUM <i>no_token</i>	<i>non_stopword</i>
<i>stopword</i> ACCUM <i>non_stopword</i>	<i>non_stopword</i>
<i>no_token</i> ACCUM <i>non_stopword</i>	<i>non_stopword</i>
<i>stopword</i> ACCUM <i>stopword</i>	<i>no_token</i>
<i>no_token</i> ACCUM <i>stopword</i>	<i>no_token</i>
<i>stopword</i> ACCUM <i>no_token</i>	<i>no_token</i>
<i>no_token</i> ACCUM <i>no_token</i>	<i>no_token</i>

MINUS Transformations

Stopword Expression	Rewritten Expression
<i>non_stopword</i> MINUS <i>stopword</i>	<i>non_stopword</i>
<i>non_stopword</i> MINUS <i>no_token</i>	<i>non_stopword</i>
<i>stopword</i> MINUS <i>non_stopword</i>	<i>no_token</i>
<i>no_token</i> MINUS <i>non_stopword</i>	<i>no_token</i>
<i>stopword</i> MINUS <i>stopword</i>	<i>no_token</i>
<i>no_token</i> MINUS <i>stopword</i>	<i>no_token</i>
<i>stopword</i> MINUS <i>no_token</i>	<i>no_token</i>
<i>no_token</i> MINUS <i>no_token</i>	<i>no_token</i>

NOT Transformations

Stopword Expression	Rewritten Expression
<i>non_stopword</i> NOT <i>stopword</i>	<i>non_stopword</i>
<i>non_stopword</i> NOT <i>no_token</i>	<i>non_stopword</i>
<i>stopword</i> NOT <i>non_stopword</i>	<i>no_token</i>
<i>no_token</i> NOT <i>non_stopword</i>	<i>no_token</i>
<i>stopword</i> NOT <i>stopword</i>	<i>no_token</i>
<i>no_token</i> NOT <i>stopword</i>	<i>no_token</i>
<i>stopword</i> NOT <i>no_token</i>	<i>no_token</i>
<i>no_token</i> NOT <i>no_token</i>	<i>no_token</i>

Equivalence Transformations

Stopword Expression	Rewritten Expression
<i>non_stopword</i> EQUIV <i>stopword</i>	<i>non_stopword</i>
<i>non_stopword</i> EQUIV <i>no_token</i>	<i>non_stopword</i>
<i>stopword</i> EQUIV <i>non_stopword</i>	<i>non_stopword</i>
<i>no_token</i> EQUIV <i>non_stopword</i>	<i>non_stopword</i>
<i>stopword</i> EQUIV <i>stopword</i>	<i>no_token</i>
<i>no_token</i> EQUIV <i>stopword</i>	<i>no_token</i>
<i>stopword</i> EQUIV <i>no_token</i>	<i>no_token</i>
<i>no_token</i> EQUIV <i>no_token</i>	<i>no_token</i>

Note: When you use query expression feedback, not all of the equivalence transformations are represented in the feedback table.

NEAR Transformations

Stopword Expression	Rewritten Expression
<i>non_stopword</i> NEAR <i>stopword</i>	<i>non_stopword</i>
<i>non_stopword</i> NEAR <i>no_token</i>	<i>non_stopword</i>
<i>stopword</i> NEAR <i>non_stopword</i>	<i>non_stopword</i>
<i>no_token</i> NEAR <i>non_stopword</i>	<i>non_stopword</i>
<i>stopword</i> NEAR <i>stopword</i>	<i>no_token</i>
<i>no_token</i> NEAR <i>stopword</i>	<i>no_token</i>
<i>stopword</i> NEAR <i>no_token</i>	<i>no_token</i>
<i>no_token</i> NEAR <i>no_token</i>	<i>no_token</i>

Weight Transformations

Stopword Expression	Rewritten Expression
<i>stopword</i> * n	<i>no_token</i>
<i>no_token</i> * n	<i>no_token</i>

Threshold Transformations

Stopword Expression	Rewritten Expression
<i>stopword</i> > n	<i>no_token</i>
<i>no_token</i> > n	<i>no_token</i>

Max Transformations

Stopword Expression	Rewritten Expression
<i>stopword</i> : n	<i>no_token</i>
<i>no_token</i> : n	<i>no_token</i>

First/Next Transformations

Stopword Expression	Rewritten Expression
<i>stopword # m-n</i>	<i>no_token</i>
<i>no_token # m-n</i>	<i>no_token</i>

WITHIN Transformations

Stopword Expression	Rewritten Expression
<i>stopword WITHIN section</i>	<i>no_token</i>
<i>no_token WITHIN section</i>	<i>no_token</i>

Index

Symbols

! as punctuation, 3-47
! operator, 3-19, 3-20
 escape character, 3-41
as punctuation, 3-47
operator, 3-16, 3-17
\$ as punctuation, 3-47
\$ operator, 3-19
% wildcard, 3-30
 theme queries, 4-10
& operator, 3-7
* operator, 3-12, 3-14
- (hyphen) as skipjoin or printjoin, 3-47
- operator, 3-12, 3-13
, (comma)
 as a numgroup character, 3-49
 as accumulate operator, 3-12
 as punctuation, 3-47
. (decimal point) numjoin, 3-48
. (period) as punctuation, 3-47
: operator, 3-16, 3-17
; operator, 3-12
= operator, 3-7, 3-8
> operator, 3-16
? operator, 3-19, 3-20
@ operator, 3-37
_ wildcard, 3-30
 theme query, 4-10
{ escape character, 3-41
| operator, 3-7
~ operator, 3-7

Numerics

32-bit environment
 viewing in, 6-10
32-bit I/O utility, 6-11

A

accumulate operator, 3-12
 example, 3-12
 in thesaurus queries, 3-24
 stopword transformations, C-5
administration tool
 linguistic settings, 7-4
algorithm for scoring, 1-8
altering precedence, 3-40
AND operator, 3-7
 example, 3-7
 stopword transformations, C-4
 with theme queries, 4-12
API
 linguistics, 7-3

B

backslash escape character, 3-41
base-letter conversion, 3-5
base-letter support
 expansion operators, 3-22
 thesuarus operator, 3-29
brace escape character, 3-41
brackets
 altering precedence, 3-31, 3-40
 grouping character, 3-31

- broader term generic operator, 3-23
- broader term instance operator, 3-24
- broader term operator, 3-23
- broader term operators
 - example, 3-27
- broader term partitive operator, 3-24
- BT operator, 3-23, 3-27
- BTG operator, 3-23, 3-27
- BTI operator, 3-24, 3-27
- BTP operator, 3-24, 3-27

C

- CANCEL procedure
 - CTX_LING, 10-34
 - CTX_SVC, 8-8, 10-54
- CANCEL_ALL procedure, 10-55
- CANCEL_USER procedure, 10-56
- cancelling linguistic request, 8-8
- case-sensitivity
 - stopwords, 3-4, 3-44
 - text queries, 1-2, 3-3
 - theme queries, 4-11
 - thesaural queries, 3-28
- categories in knowledge catalog, 4-2
- CLEAR_ALL_ERRORS procedure, 10-57
- CLEAR_ERROR procedure, 8-8, 10-58
- CLEAR_INDEX_ERRORS procedure, 10-59
- CLEAR_LING_ERRORS procedure, 10-60
- clearing linguistic requests with errors, 8-8
- CLOSE_CON procedure, 2-15, 10-4
- combined queries
 - first/next and max, 3-18
- comma
 - accumulate operator, 3-12
 - as a numgroup, 3-49
- completion processing, 8-10
- composite textkey table
 - creating theme and Gist, 8-4
 - gist structure, A-7
 - hitlist structure, A-2
 - theme structure, A-6
- composite textkeys
 - in-memory queries, 2-16
 - one-step queries, 2-13

- two-step queries, 2-6
 - using FETCH_HIT, 10-14
 - using PKDECODE, 10-24
 - using PKENCODE, 10-25
- composite word queries, 3-4
 - highlighting, 3-5
- compound phrases
 - in synonym queries, 3-25
- concepts in knowledge catalog, 4-3
- CONTAINS function, 2-10, 2-11, 9-3
 - restrictions, 2-11
 - using multiple, 2-11
- CONTAINS procedure, 2-3, 10-5
 - tables created, A-2
 - using SQEs, 3-34
- continuation character, 3-46
 - querying, 3-47
- control
 - 32-bit viewer, 6-10
- COUNT_HITS function, 2-18, 10-8
- COUNT_LAST function, 2-18, 10-10
- counting hits, 1-6, 2-18
- CREATE TABLE command, 6-6
- CREATE VIEW statement, 2-6
- CTX_LING
 - CANCEL, 10-34
 - GET_COMPLETION_CALLBACK, 10-35
 - GET_ERROR_CALLBACK, 10-36
 - GET_FULL_THEMES, 10-37
 - GET_LOG_PARSE, 8-12, 10-38
 - GET_SETTINGS_LABEL, 10-39
 - package, 7-3, 10-33
 - REQUEST_GIST, 8-4, 10-40
 - REQUEST_THEMES, 8-4, 10-43
 - SET_COMPLETION_CALLBACK, 8-10, 10-45
 - SET_ERROR_CALLBACK, 8-10, 10-46
 - SET_FULL_THEMES, 8-5, 10-47
 - SET_LOG_PARSE, 8-12, 10-48
 - SET_SETTINGS_LABEL, 8-2, 10-49
 - SUBMIT, 8-4, 10-51
- CTX_QUERY
 - CLOSE_CON, 2-15, 10-4
 - CONTAINS, 2-3, 3-34, 10-5
 - COUNT_HITS, 10-8
 - COUNT_LAST, 10-10

- FEEDBACK, 5-2, 5-15, 10-12
- FETCH_HIT, 2-15, 10-14
- GETTAB, 6-6, 6-9, 10-16, A-2
- HIGHLIGHT, 6-7, 10-18
- OPEN_CON, 2-15, 10-22
- package, 10-3
- PKDECODE, 10-24
- PKENCODE, 10-25
- PURGE_SQE, 3-32, 10-27
- REFRESH_SQE, 3-32, 3-34, 10-28
- RELTAB, 6-9, 10-29
- REMOVE_SQE, 3-32, 10-30
- STORE_SQE, 3-32, 10-31
- CTX_SQES view, 3-35
- CTX_SVC
 - CANCEL, 8-8, 10-54
 - CANCEL_ALL, 10-55
 - CANCEL_USER, 10-56
 - CLEAR_ALL_ERRORS, 10-57
 - CLEAR_ERROR, 8-8, 10-58
 - CLEAR_INDEX_ERRORS, 10-59
 - CLEAR_LING_ERRORS, 10-60
 - package, 7-4, 10-53
 - REQUEST_STATUS, 10-61
- CTX_USER_SQES view, 3-35
- CTXIO32, 6-11
- CTXLING sample application, B-7
- CTXPLUS sample application, B-3
- CTXSYS user, 7-4
- CTXV32.HLP, 6-11
- CTXV32.OCX, 6-10

D

database links

- creating, 2-8, 2-17
- in CONTAINS PL/SQL procedure, 10-5
- in COUNT_HITS PL/SQL function, 10-8
- in OPEN_CON PL/SQL function, 10-22
- using in in-memory queries, 2-17
- using in one-step queries, 2-13, 9-6
- using in two-step queries, 2-8
- DBMS_OUTPUT.ENABLE, 2-16
- decimal point
 - as a numjoin, 3-48

- DEFAULT thesaurus, 3-25
- demo application
 - CTXLING, B-7
 - CTXPLUS, B-3
 - setting up, B-2
- DML
 - affect on scoring, 1-10
- document presentation, 1-12, 6-2
 - highlighting German composites, 3-5
 - in Windows, 6-10
 - structure of tables, A-3
- document viewing, 6-2
- DROP TABLE command, 6-9

E

- endjoin character, 3-46
 - querying, 3-49
- equivalence operator, 3-7, 3-8
 - stopword transformations, C-7
- error clearing from services queue, 8-8
- error processing for linguistics, 8-10
- escaping special characters, 3-41
- EXECUTE command, 3-37
- expansion operator
 - fuzzy, 3-19, 3-20
 - penetration, 3-21
 - soundex, 3-19, 3-20
 - stem, 3-19
- expansions
 - viewing query, 5-6
- expression feedback
 - viewing, 5-16
- extending stored query expressions, 3-34

F

- feedback
 - query expansion, 5-6
 - query expression, 5-2
 - query optimization, 5-9
 - stopword rewrite, 5-10
 - theme query normalization, 5-8
- FEEDBACK procedure, 5-2, 5-15, 10-12
- feedback table

- creating, 5-15
- understanding, 5-11

FETCH_HIT function, 2-15, 10-14

first/next operator, 3-16, 3-17

- stopword transformations, C-8
- with max, 3-18

formats

- supported for Windows 32-bit viewing, 6-11

fuzzy expansions

- viewing with expression feedback, 5-6

fuzzy operator, 3-19

- example, 3-20

G

generalizing theme queries, 4-13

German composite queries, 3-4

GET_COMPLETION_CALLBACK function, 10-35

GET_ERROR_CALLBACK function, 10-36

GET_FULL_THEMES, 10-37

GET_LOG_PARSE function, 8-12, 10-38

GET_SETTINGS_LABEL function, 10-39

GETTAB procedure, 6-6, 10-16, A-2

Gist

- about, 7-10
- generating, 8-4, 10-40

Gist table

- composite textkey, 8-4
- composite textkey stucture, A-7
- creating, 8-3
- structure, A-7

granting execute privileges, 10-2

GROUP BY clause, 2-12

grouping characters, 3-31

- with theme queries, 4-10

H

hierarchical query

- query expression feedback, 5-16

hierarchies

- theme, 8-5

highlight output

- text and theme queries, 6-8

HIGHLIGHT procedure, 6-7, 10-18

- output, 6-3
- result table, A-3
- using, 6-3

highlight result tables

- creating, 6-6
- releasing, 6-9
- structure, A-3

highlighting

- composite word queries, 3-5
- mark-up, 6-4
- text query, 6-7
- theme query, 6-8

HIGHTAB table, 6-4, 6-8

- structure, A-3

hitlist

- result tables, 2-4
- sharing, 2-5

hitlist table

- composite textkey, A-2
- structure, A-2

hits counting, 2-18

homographs

- in broader term queries, 3-27
- in narrower term queries, 3-27

HTML

- embedding viewer control in, 6-10

hyphenated words

- querying, 3-47

I

I/O utility, 6-11

ICF highlight output, 6-4

ICFTAB table

- structure, A-4

indexing

- special characters, 3-46
- theme, 4-5

in-memory query, 1-5

- example, 2-15, B-6
- limitations, 2-17
- using, 2-15
- with composite textkeys, 2-16

inverse frequency scoring, 1-8

iterative queries, 3-32, 3-34

K

knowledge catalog, 4-2, 7-6
normal forms, 4-4

L

lexicon, 7-6
linguistic output, 7-8
 combining with theme/text queries, 8-13
 generating, 8-3, 8-13
 table structure, A-6
 tables, 8-3
linguistic personality, 7-4
linguistic request
 about, 1-12, 7-2, 7-4
 clearing, 8-8
 monitoring status, 8-7
 removing, 8-8
 submitting, 8-5
linguistic settings
 about, 7-12
 specifying, 8-2
linguistics
 about, 1-12, 7-2
 Application Program Interface, 7-3
 requirements, 7-2
list of themes, 7-8
 generating, 8-4, 10-43
 generating hierarchies, 8-5
 hierarchies, 7-8
logging linguistic parse information, 8-12
logical operators, 3-7

M

mark-up
 highlighting, 6-4
max operator, 3-16, 3-17
 stopword transformations, C-7
 with first/next, 3-18
MINUS operator, 3-12
 example, 3-13
 stopword transformations, C-5
monitoring services queue, 8-7
multiple CONTAINS, 2-11

multiple policies
 with one-step queries, 2-13, 4-15
MUTAB table, 6-4, 6-8, A-3

N

narrower term generic operator, 3-23
narrower term instance operator, 3-23
narrower term operator, 3-23
narrower term operators
 example, 3-26
narrower term partitive operator, 3-23
near operator, 3-12, 3-13
 stopword transformations, C-7
 with threshold, 3-14
nesting stored query expressions, 3-35
NOFILTAB table, 6-4
 structure, A-5
normal forms in knowledge catalog, 4-4
NOT operator, 3-7
 example, 3-8
 stopword transformations, C-6
 with theme queries, 4-13
NT operator, 3-23, 3-26
NTG operator, 3-23, 3-26
NTI operator, 3-23, 3-26
NTP operator, 3-23, 3-26
numbers
 querying, 3-48
numgroup character, 3-46
 querying, 3-48
numjoin character, 3-46
 querying, 3-48

O

OCX
 32-bit viewer, 6-10
one-step query, 1-5, 2-10
 example, 2-11, B-5
 multiple policies, 2-13, 4-15
 processing, 2-10
 SELECT statement, 9-6
 theme query, 4-15
OPEN_CON function, 2-15, 10-22

OPERATION column of feedback table, 5-12

operator

- accumulate, 3-12

- AND, 3-7

- broader term, 3-27

- equivalence, 3-7, 3-8

- first/next, 3-16, 3-17

- fuzzy, 3-20

- max, 3-16, 3-17

- MINUS, 3-12, 3-13

- narrower term, 3-26

- near, 3-12, 3-13

- NOT, 3-7, 3-8

- OR, 3-7

- penetration, 3-21

- preferred term, 3-26

- related term, 3-26

- soundex, 3-20

- SQE, 3-32

- stem, 3-19

- synonym, 3-23, 3-24, 3-25

- thesaurus, 3-23

- threshold, 3-16

- top term, 3-28

- weight, 3-12, 3-14

- WITHIN, 3-10

operator precedence, 3-38

- examples, 3-39

- viewing with parse trees, 5-5

operators

- expansion, 3-19

- logical, 3-7

- result-set, 3-16

- score-changing, 3-12

- stem, 3-19

- theme query examples, 4-9

- thesaurus, 3-23

operators in SQEs, 3-36

optimization of queries

- expression feedback, 5-9

OPTIONS column of feedback table, 5-13

OR operator, 3-7

- example, 3-7

- stopword transformations, C-4

Oracle Worldwide Technical Support

- how to contact, xix

- how to contact in Europe, xix

- how to contact in U.S.A., xix

output

- generating linguistic, 8-13

- linguistic, 7-8

P

package

- CTX_LING, 7-3, 10-33

- CTX_QUERY, 10-3

- CTX_SVC, 7-4, 10-53

paragraph-level Gist, 7-10

- generating, 10-40

paragraph-level theme summary, 7-9

- generating, 10-40

parallel processing

- two-step queries, 2-9

parent themes

- in list of themes, 7-8

parentheses

- altering precedence, 3-31, 3-40

- grouping character, 3-31

parse logging, 8-12

parse trees

- query expansion, 5-6

- query optimization, 5-9

- stopword transformation, 5-10

- theme query normalization, 5-8

- understanding, 5-4

parsing engine, 7-7

PENDING requests

- removing, 8-8

penetration of operators, 3-21

personality

- linguistic, 7-4

PKDECODE function, 10-24

PKENCODE function, 10-25

PL/SQL

- in text queries, 3-37

PL/SQL packages

- granting privileges, 10-2

PLAINTAB table, 6-4

- structure, A-5

- pol_hint parameter in SELECT statement, 2-13, 4-15, 9-7
- policies
 - multiple, 2-13, 4-15
- precedence of operators, 3-38
 - altering, 3-31, 3-40
 - equivalence operator, 3-9
 - example, 3-39
- preferred term operator, 3-23
 - example, 3-26
- presentation
 - document, 1-12
- printjoin character, 3-46
 - querying with, 3-47
- privileges
 - granting, 10-2
- procedures
 - in queries, 3-37
- proximity operator, see near operator
- PT operator, 3-23, 3-26
- punctuation character, 3-46
 - querying, 3-47
- PURGE_SQE procedure, 3-32, 10-27

Q

- qualifiers
 - using in thesaural queries, 3-27
- query
 - accumulate, 3-12
 - AND, 3-7
 - base-letter, 3-5
 - broader term, 3-27
 - combined first/next and max, 3-18
 - composite textkey, 2-6
 - counting hits, 1-6, 2-18
 - equivalence, 3-8
 - executing PL/SQL function in, 3-37
 - first/next, 3-17
 - in-memory, 1-5, 2-15
 - in-memory example, B-6
 - iterative, 3-32, 3-34
 - max, 3-17
 - MINUS, 3-13
 - narrower term, 3-26

- NOT, 3-8
- one-step, 1-5
- one-step example, B-5
- OR, 3-7
- preferred term, 3-26
- related term, 3-26
- remote, 2-8, 2-13, 2-17, 9-6, 10-5, 10-8, 10-22
- structured, 2-7
- synonym, 3-25
- theme, 1-4
- threshold, 3-16
- top term, 3-28
- two-step, 1-5
- two-step example, B-5
- weight, 3-14
- query expansions
 - viewing with expression feedback, 5-6
- query expression
 - about, 1-7, 3-2
 - components, 3-3
 - examples, 3-5
- query expression feedback
 - about, 5-2
 - obtaining, 5-15
- query optimization
 - viewing with expression feedback, 5-9
- query terms, 3-2
- query_id parameter in CONTAINS, 2-5, 10-6
- querying
 - continuation characters, 3-47
 - numbers, 3-48
 - punctuation characters, 3-47
 - theme, 4-7
- querying with stopwords, 3-44
- queue
 - services, see services queue

R

- re-evaluation of SQEs, 3-33
- refining theme queries, 4-12
- REFRESH_SQE procedure, 3-32, 3-34, 10-28
- related term operator, 3-23
 - example, 3-26
- relevance ranking

- text queries, 1-8
- RELTAB procedure, 10-29
- remote databases
 - counting query hits in, 1-6, 2-18, 10-8
 - in-memory queries in, 2-17, 10-22
 - one-step queries in, 2-13, 9-6
 - two-step queries in, 2-8, 10-5
- remote queries
 - in-memory, 2-17, 10-22
 - one-step, 2-13, 9-6
 - query hits counting, 10-8
 - two-step, 2-8, 10-5
- REMOVE_SQE procedure, 3-32, 10-30
- removing pending requests, 8-8
- request
 - clearing errors, 8-8
 - monitoring linguistic, 8-7
 - removing linguistic, 8-8
- REQUEST_GIST procedure, 8-4, 10-40
- REQUEST_STATUS function, 10-61
- REQUEST_THEMES procedure, 8-4, 10-43
- reserved words and characters, 3-41
 - escaping, 3-41
- restricting theme query, 4-12
- result table
 - allocating, 6-6
 - composite textkey, A-2
 - definition, 1-11
 - highlight, A-3
 - hitlist, 2-4, A-2
 - shared, 2-5
- result-set operators, 3-16
- rewrite
 - stopword, 3-44, C-2
- RT operator, 3-23, 3-26

S

- sample application
 - CTXLING, B-7
 - CTXPLUS, B-3
 - setting up, B-2
- SCORE function, 2-10, 2-12, 9-5
- score-changing operators, 3-12
- scoring, 1-8

- DML, 1-10
- theme queries, 4-8
- two-step queries, 2-4
- scoring algorithm
 - text queries, 1-8
- section searching, 3-10
- SELECT statement, 2-11, 2-12, 9-6
 - in one-step queries, 2-11
 - in two-step queries, 2-3
- sentence-level Gist, 7-10
- sentence-level Gist and theme summary
 - generating, 10-40
- sentence-level theme summary, 7-9
- server personality, 7-4
- service request
 - cancelling, 10-54, 10-55, 10-56
 - removing errors, 10-57, 10-59, 10-60
- services queue
 - about, 7-4
 - monitoring, 8-7
- session and system SQEs, 3-33
- session configuration
 - setting, 8-2
- SET_COMPLETION_CALLBACK procedure, 8-10, 10-45
- SET_ERROR_CALLBACK procedure, 8-10, 10-46
- SET_FULL_THEMES procedure, 8-5, 10-47
- SET_LOG_PARSE procedure, 8-12, 10-48
- SET_SETTINGS_LABEL procedure, 8-2, 10-49
- settings
 - linguistic, 7-12, 8-2
- sharelevel parameter in CONTAINS, 2-5, 10-5
- sharing result table, 2-5
- skipjoin character, 3-46
 - querying with, 3-47
- soundex expansions
 - view with expression feedback, 5-6
- soundex operator, 3-19
 - example, 3-20
- special characters
 - indexing, 3-46
 - querying, 3-46
- SQE operator, 3-32
- SQE tables, 3-35
- SQEs, see stored query expressions

- SQL functions
 - CONTAINS, 9-3
 - SCORE, 9-5
- startjoin character, 3-46
 - querying, 3-49
- stem expansions
 - viewing with expression feedback, 5-6
- stem operator, 3-19
- stoplist, 3-2
- stopword transformation, C-2
 - viewing with expression feedback, 5-10
- stopwords, 3-2
 - case-sensitivity, 3-4, 3-44
 - querying, 3-44
- STORE_SQE procedure, 3-32, 10-31
- stored query expressions
 - behavior with FEEDBACK, 10-13
 - extending, 3-34
 - iterative queries, 3-34
 - nesting, 3-35
 - re-evaluation, 3-33
 - session and system, 3-33
 - support of operators, 3-36
 - using, 3-32
- stripping punctuation characters, 3-47
- structured query, 2-7
- SUBMIT function, 8-4, 10-51
- submitting linguistic requests, 8-5
- supported document formats
 - for Windows 32-bit viewing, 6-11
- SYN operator, 3-23, 3-25
- synonym operator, 3-23
 - example, 3-25

T

- table structure
 - Gist, A-7
 - HIGHLIGHT, A-3
 - hitlist, A-2
 - ICFTAB, A-4
 - MUTAB, A-3
 - NOFILTAB, A-5
 - PLAINTAB, A-5
 - theme, A-6

- tables
 - result, 1-11
 - SQE, 3-35
 - SQR, 3-35, 10-28
- tagged text
 - querying, 3-49
 - searching, 3-10
- text query
 - about, 1-2
 - case-sensitivity, 1-2, 3-3
 - combining with linguistic output, 8-13
 - highlighting, 6-7
 - scoring, 1-8
 - selecting method, 2-2
- theme hierarchy generation
 - setting, 10-47
- theme indexing, 4-5
- theme query, 1-4
 - about, 4-7
 - combining with linguistic output, 8-13
 - constructing, 4-9
 - generalizing, 4-13
 - highlighting, 6-8
 - one-step, 4-15
 - operators not supported, 4-10
 - phrasing hints, 4-10
 - refining, 4-12
 - restricting, 4-12
 - scoring, 1-8, 4-8
 - two-step, 4-15
 - using operators, 4-9
- theme query normalization
 - viewing with expression feedback, 5-8
- theme summaries
 - about, 7-9
 - generating, 10-40
- theme table
 - composite textkey, 8-4, A-6
 - creating, 8-3
 - structure, A-6
- theme weight, 4-6
 - in theme indexes, 4-6
 - list of themes, 7-9
- themes
 - generating, 10-43

- generating hierarchies, 8-5
- generating list of, 8-4
- list of, 7-8
- thesaurus
 - calling in queries, 3-25
 - DEFAULT, 3-25
 - hierarchy levels, 3-25
- thesaurus expansions
 - viewing with expression feedback, 5-6
- thesaurus operators, 3-23
 - arguments, 3-24
 - limitations, 3-24
 - with theme queries, 4-9
- threshold operator, 3-16
 - stopword transformations, C-7
 - with near operator, 3-14
- top term operator, 3-23
 - example, 3-28
- transformation
 - stopword, 3-44, 5-10, C-2
- tree-structure of knowledge catalog, 4-2
- TT operator, 3-23, 3-28
- two-step query, 1-5, 2-3
 - alternative, 2-6
 - example, 2-3, B-5
 - parallel processing, 2-9
 - result table, 2-4
 - scoring, 2-4
 - tables used in, A-2
 - theme query, 4-15

V

- view
 - using in two-step query, 2-6
- viewer control (32-bit), 6-10
- viewing documents, see document presentation
- views
 - CTX_SQES, 3-35
 - CTX_USER_SQES, 3-35
- Visual Basic
 - using the viewer control with, 6-10

W

- weight
 - in list of themes, 7-9
 - theme, 4-6
- weight operator, 3-12, 3-14
 - stopword transformations, C-7
- wildcard characters, 3-30
 - with theme queries, 4-10
- wildcard expansions
 - viewing with expression feedback, 5-6
- Windows viewer control, 6-10
 - table used, A-4
- WITHIN operator, 3-10
 - precedence, 3-39
 - stopword transformations, C-8