Interstage Shunsaku Data Manager
Application Development Guide

Trademarks

Trademarks of other companies are used in this user guide only to identify particular products or systems.

Product	Trademark/Registered Trademark
Microsoft, Windows, and Windows Server	Registered trademarks of Microsoft Corporation in the United States and other countries.
Java	Java and other trademarks relating to Java are trademarks of Sun Microsystems, Inc in the United States and other countries.
Linux	Trademark or registered trademark of Linus Torvalds in the United States and other countries.
Red Hat, RPM, and all trademarks and logos based on Red Hat	Trademarks or registered trademarks of Red Hat, Inc., in the USA and other countries.

Fujitsu documentation may contain specific technologies that apply to foreign exchange and foreign trade control laws. When such specific technology is described in the document, and that document is either exported or provided to a non-resident, permission based on these laws is required.

Fujitsu Limited

First Edition (August 2004)

Preface

Purpose of this Manual

This manual describes how to create applications that use the APIs provided by Interstage Shunsaku Data Manager (hereafter abbreviated as Shunsaku).

Target Audience

This manual is aimed at the following readers:

- Persons using Shunsaku.
- Persons developing applications that use the Shunsaku APIs.

Required Knowledge

This manual assumes that the reader has an understanding of the following topics:

- XML
- C language
- Java
- Internet basics

- Windows

- Linux

Organization of this Manual

This manual is organized as follows:

- *Chapter 1 - Overview*

  This chapter provides an overview of the functions of Shunsaku.

- *Chapter 2 - Environment Setup*

  This chapter explains the environment settings needed for creating applications that use the Shunsaku APIs.

- *Chapter 3 - Data Search Methods*

  This chapter explains the data search methods that use search expressions, return expressions and sort expressions specified as arguments for the APIs provided by Shunsaku.

- *Chapter 4 - Data Updating Methods*

  This chapter explains how to update data using the Shunsaku APIs.

- *Chapter 5 - Developing Java Applications*

  This chapter explains how to develop applications using Shunsaku's Java APIs.

- *Chapter 6 - Developing C Applications*

  This chapter explains how to develop applications using Shunsaku's C APIs.

- *Appendix A - Formats of Search Expressions, Return Expressions and Sort Expressions*

  This appendix explains the formats of the search expressions, return expressions and sort expressions that are specified as arguments of the Shunsaku APIs.

- *Appendix B - Sample Java Programs*

  This appendix provides sample programs that use the Java APIs.

- *Appendix C - Sample C Programs*

  This appendix provides sample programs that use the C APIs.

- *Appendix D - Allowable Values*

  This appendix explains the allowable values that are involved during the development of Shunsaku applications.

- *Appendix E – Estimating Resources*

  This appendix explains how to estimate resources when using applications to add, delete or search for data.

- *Appendix F - Notes on XML documents*

  This appendix provides notes on the XML documents that are stored in Shunsaku.

- *Glossary*

  This defines the terms used in this manual.

Related Manuals

Relevant manuals are provided in the form of online manuals.

Title used in this manual	Full manual title
User's Guide	Interstage Shunsaku Data Manager User's Guide
Application Development Guide (This manual)	Interstage Shunsaku Data Manager Application Development Guide
Java API Reference	Interstage Shunsaku Data Manager Java API Reference
C API Reference	Interstage Shunsaku Data Manager C API Reference

Positioning of this Manual

The following table shows how Shunsaku manuals are organized.

Manual title	Content
User's Guide	Explains the functions provided by Shunsaku, and describes how to set up the environment and operate Shunsaku.
Application Development Guide (This manual)	Explains how to create applications that use the Shunsaku APIs.
Java API Reference	Explains the syntax of the Shunsaku Java APIs.
C API Reference	Explains the syntax of the Shunsaku C APIs.

Supplementary Information

Platform-specific Information

This manual provides information for all platforms that are supported by Shunsaku.

Where information differs for each platform, the following icons are displayed in front of the information relating to each platform.

In this case, only refer to the information relating to the platform being used.

Windows : Indicates Windows-specific information.

Linux : Indicates Linux-specific information.

Applicable Products

Interstage Shunsaku Data Manager

Windows

- Windows Interstage® Shunsaku Data Manager Enterprise Edition V6.0L30

Linux

- Linux Interstage® Shunsaku Data Manager Enterprise Edition V6.0L30

Terminology used in Shunsaku manuals

The following table shows the correspondence of terms used in Shunsaku manuals.

Windows	Linux	Term used in Shunsaku manual
Folder	**Directory**	The term ‘directory’ appears in some parts of Shunsaku manuals.

Abbreviations

This document uses abbreviations for product names as shown in the following table:

Abbreviation	Product name
Shunsaku	Windows Interstage® Shunsaku Data Manager Enterprise Edition Linux Interstage® Shunsaku Data Manager Enterprise Edition
Windows	Microsoft® Windows® 2000 Server operating system Microsoft® Windows® 2000 Advanced Server operating system Microsoft® Windows Server™ 2003, Standard Edition Microsoft® Windows Server™ 2003, Enterprise Edition
Linux	Red Hat Enterprise Linux AS Red Hat Enterprise Linux ES Red Hat Linux

Table of Contents

Chapter 1

Overview

This chapter provides an overview of Shunsaku functions.

- Performing Data Searches from Applications
- Performing Data Updates from Applications

Performing Data Searches from Applications

The Shunsaku APIs enable data searches to be initiated from applications.

Shunsaku provides Java and C APIs for search operations.

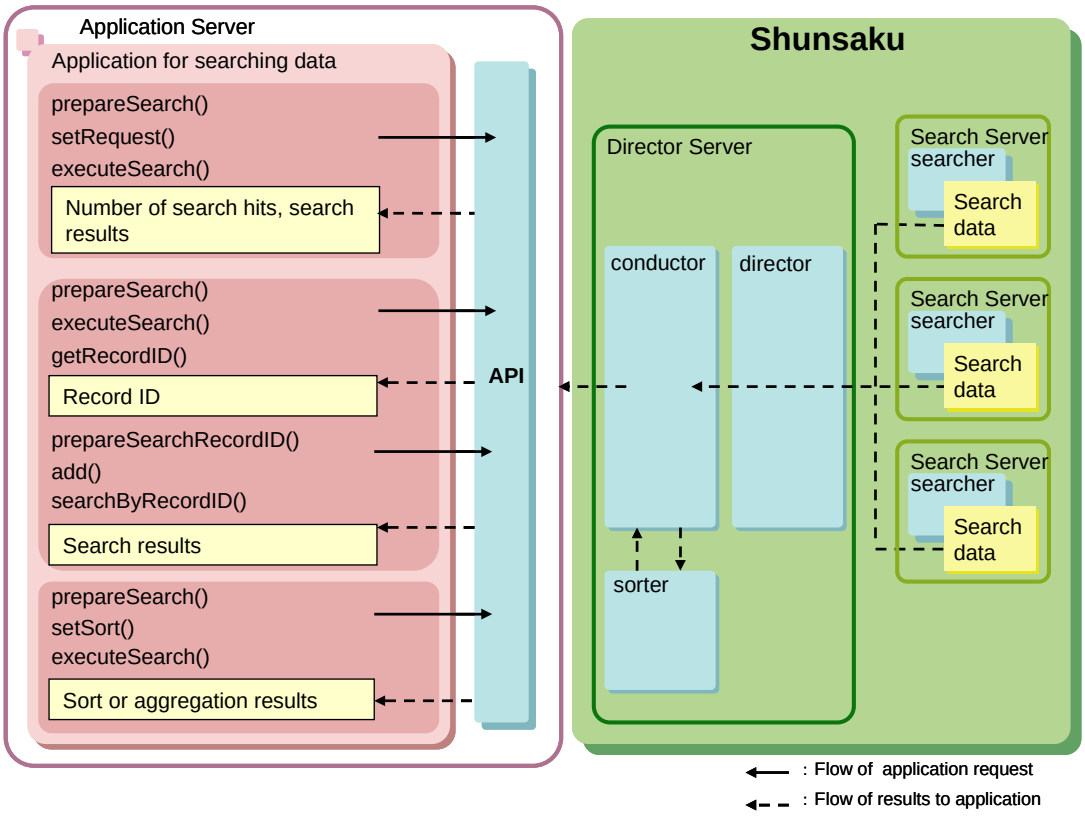


Figure 1-1 Searching Data from Applications

Performing Data Updates from Applications

The Shunsaku APIs enable data updates to be initiated from applications.

Shunsaku provides Java and C APIs for update operations.

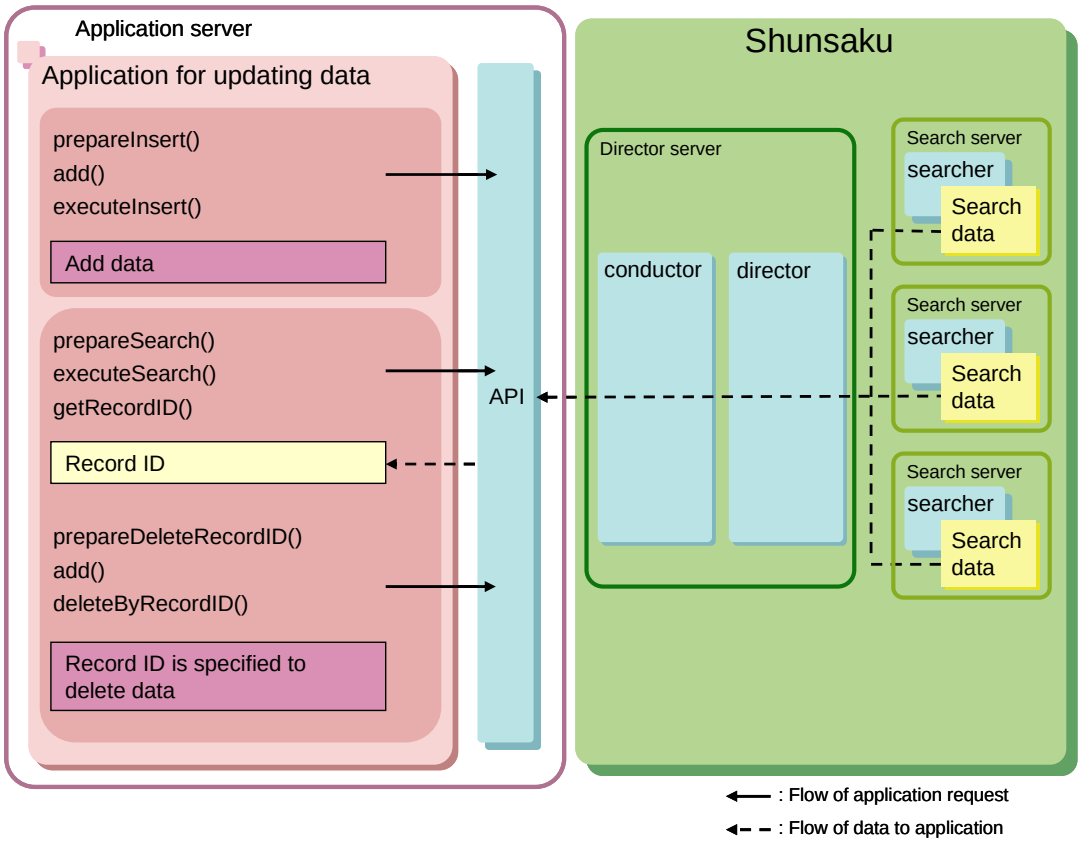


Figure 1-2 Updating Data from Applications

Chapter 2

Environment Setup

This chapter explains the environment settings that are required to create applications that can use Shunsaku.

- API Configuration
- Setup
- Setting Environment Variables

API Configuration

This section explains the configuration of Shunsaku APIs.

Java APIs

This section explains the configuration of Java APIs.

File Configuration for Java APIs

When Shunsaku APIs are installed, the following jar format file is created.

Windows

Shunsaku installation folder\Shunsaku\lib\shunapi.jar

Linux

/opt/FJSVshnsk/lib/shunapi.jar

API Package (shunapi.jar)

This package contains Java classes for creating applications.

Refer to the Java API Reference for more information on the functions that can be implemented by using the methods provided by the Shunsaku APIs.

C APIs

This section explains the configuration of C APIs.

Directory Configuration for C APIs

When Shunsaku APIs are installed, the following directories are created.

Windows

Shunsaku installation folder\Shunsaku\include:

This is the include file provided by the Shunsaku APIs.

Shunsaku installation folder\Shunsaku\lib:

This is the library provided by the Shunsaku APIs.

Linux

/opt/FJSVshnsk/include:

This is the include file provided by the Shunsaku APIs.

/opt/FJSVshnsk/lib:

This is the library provided by the Shunsaku APIs.

Include File

This include file is referenced by each function in the Shunsaku APIs. The include file is shown below.

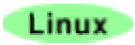
Table 2-1 Include File

Include file name	Usage
libshun.h	This file is referenced by each function in Shunsaku APIs.

Library

The following file contains the C function library for creating applications:

f3hyshun.lib



libshun.so

Refer to the C API Reference for more information on the functions that can be implemented by using the libraries provided by the Shunsaku APIs.

Setup

Refer to the User’s Guide for more information on how to install Shunsaku APIs on an application server.

Setting Environment Variables

This section explains the environment variables that are needed to use the Shunsaku APIs.

Setting Environment Variables (Java APIs)

Set the environment variables required for using the Shunsaku Java APIs.

Add ‘shunapi.jar’ to the CLASSPATH environment variable.

Examples of how to set these environment variables are shown below.

Windows

Example

When Shunsaku is installed using the Typical installation option:

```
SET CLASSPATH=C:\Program Files\Interstage
Shunsaku\Shunsaku\lib\shunapi.jar;%CLASSPATH%
```

Linux

Example 1

For bash, Bourne and Korn shell:

```
CLASSPATH=/opt/FJSVshnsk/lib/shunapi.jar:$CLASSPATH; export CLASSPATH
```

Example 2

For C shell:

```
setenv CLASSPATH /opt/FJSVshnsk/lib/shunapi.jar:$CLASSPATH
```

Setting Environment Variables (C APIs) Linux

Set the environment variables required for using the Shunsaku C APIs.

Add '/opt/FJSVshnsk/lib' to the LD_LIBRARY_PATH environment variable.

Examples of how to set these environment variables are shown below.

Example 1

For bash, Bourne and Korn shell:

```
LD_LIBRARY_PATH=/opt/FJSVshnsk/lib:$LD_LIBRARY_PATH ; export
LD_LIBRARY_PATH
```

Example 2

For C shell:

```
setenv LD_LIBRARY_PATH /opt/FJSVshnsk/lib:$LD_LIBRARY_PATH
```

Chapter 3

Data Search Methods

This chapter explains how the search expressions, return expressions and sort expressions specified as arguments to Shunsaku API functions, are used to search for data.

- Data Search Overview
- Specifying the Search Expression
- Sorting Search Results
- Extracting Search Results
- Extracting Aggregated Results

Data Search Overview

This section explains the basic approach to searching for XML documents stored in Shunsaku.

- A search expression is used to search for particular XML documents stored in Shunsaku.
- A sort expression is used to change the sequence in which XML documents are returned.
- A return expression is used to decide which elements will be aggregated or extracted from the XML documents that are found by a search.
- A group key is selected when aggregation is to be performed. The group key is specified in the sort expression.

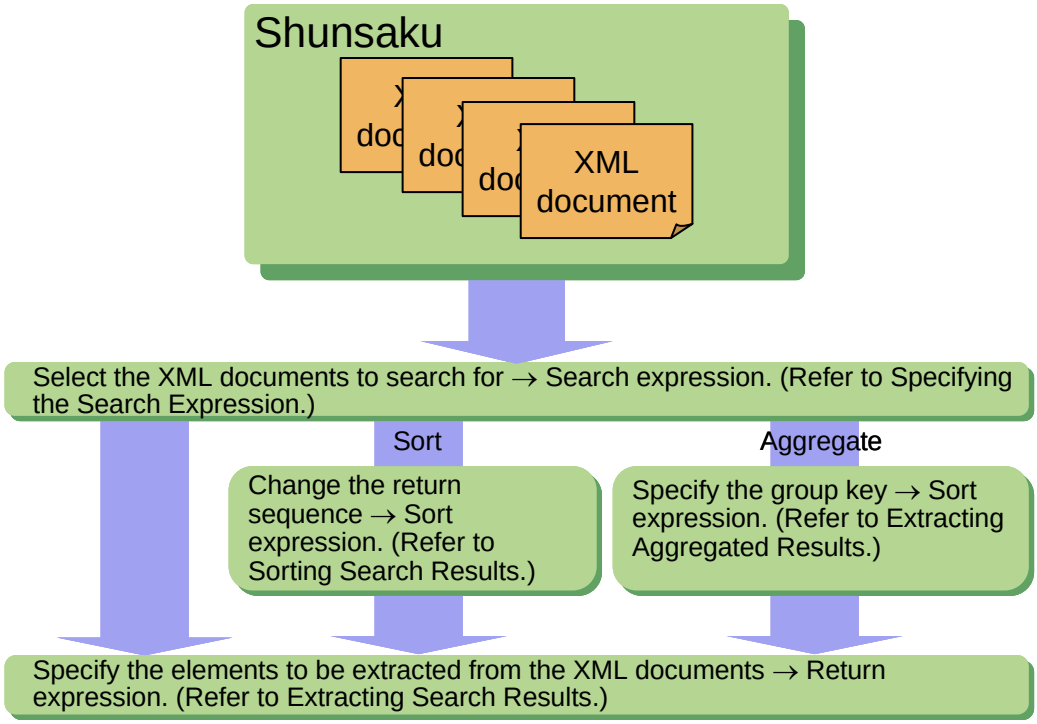


Figure 3-1 Searching for XML Documents Stored in Shunsaku

Refer to Appendix A, Formats of Search Expressions, Return Expressions and Sort Expressions for more information on the search expressions, sort expressions and return expressions.

The explanations in the following sections use a 'Business Trip Report' as an example. Assume that the following XML document exists:

Sample Document

```
<doc>
  <employee>                                    Employee information
    <eno>Employee number</eno>
    <name>Employee name</name>
    <sno>Section number</sno>
    <phone>Extension</phone>
    <email>address</email>
```

```
  </employee>
  <basic>                                          Basic information
    <date>Trip date</date>
    <expense>Trip expenses</expense>
  </basic>
  <detail>                                         Details
    <destination>Trip destination</destination>
    <area>Trip area</area>
    <purpose>Trip purpose</purpose>
    <train>Traveling expenses</train>
    <taxi>Taxi fares</taxi>
    <hotel>Accommodation costs</hotel>
    <comment>Comments</comment>
  </detail>
  <report>Trip report</report>                     Report
</doc>
<doc>
   :
   :
```

Specifying the Search Expression

A search expression is used to retrieve XML documents that satisfy specified conditions from the XML documents stored in Shunsaku. The search expression is specified in the APIs provided by Shunsaku.

Refer to Search Expressions in Appendix A for more information.

Searching with Character Strings

It is possible to search for XML documents by specifying conditions that apply to character strings contained in any given element node of an XML document. The following three methods can be used to specify the conditions that apply to the character string:

- Searching for Documents that Contain a Search Keyword
- Searching for Documents that Match a Search Keyword
- Performing a Size Comparison with a Search Keyword

The examples that follow assume that the following documents exist:

Document A

```
<doc>
    :
  <basic>
    <date>30/01/2004</date>
    :
  </basic>
  <detail>
    <destination>Tokyo Office</destination>
    <area>Chiyoda-ku, Tokyo</area>
    <purpose>Attendance at Shunsaku Sales Conference</purpose>
    <train>7200 yen</train>
    <taxi></taxi>
    <hotel>8500 yen</hotel>
    <comment>Hotel charges were incurred due to attendance at an informal
meeting</comment>
  </detail>
  <report>The Systems Manager of the IT Department of Company A was also
at the conference</report>
</doc>
```

Document B

```
<doc>
    :
  <basic>
    <date>31/01/2004</date>
    :
  </basic>
  <detail>
    <destination>Osaka branch</destination>
    <area>Abeno-ku, Osaka-shi, Osaka</area>
    <purpose>V6.0L30 installation report</purpose>
    <train>14200 yen</train>
```

```
    <taxi>1820 yen</taxi>
    <hotel></hotel>
    <comment>Plan to return home directly</comment>
  </detail>
  <report>Because the installation only took a short time</report>
</doc>
```

Document C

```
<doc>
    :
  <basic>
    <date>02/03/2004</date>
    :
  </basic>
  <detail>
    <destination>Head office</destination>
    <area>Kohoku-ku, Yokohama-shi, Kanagawa</area>
    <purpose>Regular Interstage meeting</purpose>
    <train>2400 yen</train>
    <taxi></taxi>
    <hotel></hotel>
    <comment>Sales conference materials were verified in advance</comment>
  </detail>
  <report>I received permission from the Sales Coordination Department to
verify the materials in advance</report>
</doc>
```

Searching for Documents that Contain a Search Keyword

It is possible to search for XML documents with an element node that has a character string that contains a search keyword. This type of search is called a partial match search. Partial match searches are specified by connecting the path expression and the search keyword with the relational operator '='.

Refer to Path Expression in Appendix A for more information.

Example 1

The following example shows how to search for documents that use the keyword 'Shunsaku' in the trip purpose (*purpose*):

```
/doc/detail/purpose = 'Shunsaku'
```

Result

Document A is returned.

Example 2

If '//' is specified in the path expression, a partial match search can be performed on all levels of element nodes under any given element node. By specifying '*' in the path expression, a partial match can be performed for all element nodes under any given element node.

Note

The '//' specification is a useful way to find documents that contain the search keyword in any element node of an entire XML document, and not just in a specific element node.

The following example shows how to search for documents that contain the keyword ‘Sales Conference’ somewhere in the detailed information (*detail*):

```
/doc/detail/* = ‘Sales Conference’
```

Result

Documents A and C are returned.

Example 3

If ‘//’ is specified in the path expression, a partial match search can be performed on all levels of element nodes under any given element node. By specifying ‘*’ in the path expression,a partial match can be performed for all element nodes under any given element node.

Note

The ‘//’ specification is a useful way to find documents that contain the search keyword in any element node of an entire XML document, and not just in a specific element node.

The following example shows how to search for documents that contain the string ‘V6.0L30’:

```
/doc// = 'V6\.0L30'
```

Result

Document B is returned.

Note

The character \’ is an escape character for a dot ‘.’.

Refer to Escape Characters for more information on escape characters.

Example 4

It is also possible to specify a wildcard within keywords to allow searches for words that begin with or end with certain characters. Wildcards are specified as ellipses ‘…’. This kind of search is called a ‘ellipse search’. Refer to Ellipse Searches in Appendix A for more information.

The following example shows how to search for documents that contain keywords beginning with ‘Systems Manager’ and ending with ‘Company A’:

```
/doc/report = 'Systems Manager...Company A'
```

Result

Document A is returned.

Searching for Documents that Match a Search Keyword Exactly

It is possible to search for documents with an element node that contain a character string that exactly matches a given keyword. This type of search is known as a complete match search. The complete match search is specified by connecting the path expression and the search keyword with the relational operator ‘==’.

Refer to Path Expression in Appendix A for more information.

Example 1

The following example shows how to search for documents in which the trip destination (*destination*) is specified as 'Head Office':

```
/doc/detail/destination == 'Head Office'
```

Result

Document C is returned.

Example 2

If '//' is specified in the path expression, a complete match search can be performed on all levels of element nodes under any given element node. By specifying '*' in the path expression, a complete match search can be performed for all element nodes under any given element node

The following example shows how to search for documents that have '31/01/2004' as the value of any element node in basic information (*basic*):

```
/doc/basic/* == '31/01/2004'
```

Result

Document B is returned.

Performing a Size Comparison with a Search Keyword

It is possible to search for documents that meet conditions by performing a size comparison between a character string in any element node of an XML document and any keyword. (The size relationship between character strings refers to the size relationship between the character code values of those strings.) This type of search is known as a size comparison search. The size comparison search is specified by connecting the path expression and the search keyword with the relational operator '<', '<=', '>' or '>='.

Refer to Relational Operators in Character Searches in Appendix A for more information on relational operators.

Example

The following example shows how to search for documents in which the trip date (*date*) is later than 01/02/2004:

```
/doc/basic/date > '01/02/2004'
```

Result

Document C is returned.

Note

To perform a size comparison between character codes, the character strings need to be in the same format. For example, if the date is specified as '1/2/2004', the intended documents will not be found.

Searching by Numeric Value

It is possible to search for documents that meet conditions by performing a size comparison between the data in any element node of an XML document and the numeric value of a keyword. For document searches using a numeric value, the character string in an element node in the XML document is treated as a numeric value. Therefore, unlike the size comparison search, the format of the character string need not be considered.

The data in the element node specified by the path expression is treated as a numeric value and is compared to the numeric value specified by the keyword. The path expression and the search keyword are specified using a relational operator.

The following relational operators can be used: '=', '!=', '<', '<=', '>' and '>='.

The keyword is specified as a numeral not enclosed in quotes.

Refer to Path Expressions in Appendix A for more information.

Refer to Numeric Value Searches in Appendix A for more information on numeric values and relational operators.

The explanation that follows assumes that the following documents exist:

Document A

```
<doc>
  <employee>
    <eno>19980120</eno>
    <name>Taro Suzuki</name>
    <sno>1001</sno>
    <phone>2201-1101</phone>
    <email>suzuki.taro@shunsaku.fujitsu.com</email>
  </employee>
  <basic>
    <date>16/02/2004</date>
    <expense>15700 yen</expense>
  </basic>
    :
</doc>
```

Document B

```
<doc>
  <employee>
    <eno>20012111</eno>
    <name>Hanako Sato</name>
    <sno>2002</sno>
    <phone>2201-1204</phone>
    <email>sato.hanako@shunsaku.fujitsu.com</email>
  </employee>
  <basic>
    <date>18/02/2004</date>
    <expense>8500 yen</expense>
  </basic>
    :
</doc>
```

Example 1

The following example shows how to search for documents in which the section number (*sno*) is specified as 1001:

```
/doc/employee/sno = 1001
```

Result

Document A is returned.

Example 2

If '//' is specified in the path expression, a search can be performed on all levels of element nodes under any given element node. By specifying '*' in the path expression, all element nodes under any given element node can be searched.

Note

The '//' specification is useful when the target element nodes can be uniquely determined without having to express a hierarchical structure.

The following example shows how to search for documents in which the trip expenses (*expense*) are not more than ¥10,000:

```
//expense <= 10000
```

Result

Document B is returned.

Searching by Joining Multiple Conditions with Logical Operators

Searches can be conducted using multiple conditions joined by logical operators. Logical operators make it possible to search for XML documents that satisfy two conditions, or that satisfy only one of two conditions.

The logical operators AND and OR can be specified. When both AND and OR are specified in a search expression, AND takes precedence over OR.

Refer to Logical Operators in Appendix A for more information.

The explanation that follows assumes that the following documents exist:

Document A

```
<doc>
    :
  <basic>
    <date>30/01/2004</date>
    <expense>15700 yen</expense>
  </basic>
  <detail>
    <destination>Tokyo Office </destination>
    <area>Chiyoda-ku, Tokyo</area>
```

```
    <purpose>Attendance at Shunsaku Sales Conference</purpose>
    :
  </detail>
    :
</doc>
```

Document B

```
<doc>
    :
  <basic>
    <date>16/02/2004</date>
    <expense>16020 yen</expense>
  </basic>
  <detail>
    <destination>Osaka branch</destination>
    <area>Abeno-ku, Osaka-shi, Osaka</area>
    <purpose>V6.0L30 installation report</purpose>
    :
  </detail>
    :
</doc>
```

Document C

```
<doc>
    :
  <basic>
    <date>02/03/2004</date>
    <expense>2400 yen</expense>
  </basic>
  <detail>
    <destination>Head office</destination>
    <area>Kohoku-ku, Yokohama-shi, Kanagawa</area>
    <purpose>Regular Interstage meeting</purpose>
    :
  </detail>
    :
</doc>
```

Example 1

The following example shows how to search for documents that contain the keyword ‘Interstage’ or ‘Shunsaku’ in the trip purpose (*purpose*):

```
/doc/detail/purpose = 'Interstage' OR
/doc/detail/purpose = 'Shunsaku'
```

Result

Documents A and C are returned.

Example 2

The following example shows how to search for documents in which the trip expenses (*expense*) are not less than ¥10,000 and the trip area (*area*) includes ‘Osaka’:

```
/doc/basic/expense >= 10000 AND
/doc/detail/area = 'Osaka'
```

Result

Document B is returned.

Example 3

To force OR to have precedence over AND, the OR conditional expression must be enclosed in parentheses.

The following example shows how to search for documents in which the trip destination (*destination*) is ‘Head Office’ or ‘Osaka Branch’ and the trip date (*date*) includes ‘02/2004’:

```
(/doc/detail/destination == 'Head Office' OR
 /doc/detail/destination == 'Osaka Branch') AND
/doc/basic/date = 02/2004'
```

Result

Document B is returned.

Sorting Search Results

XML documents found using a search expression can be sorted according to a specified key before being returned. To do so, a sort expression is specified in the Shunsaku APIs. The sort expression is specified as a sort key text expression or as a single-line function specification.

Refer to Sort Expressions in Appendix A for more information on sort expressions.

Sorting by Character String

It is possible to sort search results using a character string in any element nodes of an XML document. The size relationship between character strings refers to the size relationship between the character code values of those strings. To use a character string to sort search results, a key specification in the sort expression must be defined using either a text expression or the *rlen* single-line function. Refer to Sort Expression Formats in Appendix A for more information on key specifications.

The explanation that follows assumes that the following documents exist:

Document A

```
<doc>
    :
  <basic>
    <date>2201/2004</date>
    :
  </basic>
  <detail>
    :
    <area>Yaesu, Chiyoda-ku, Tokyo</area>
    :
  </detail>
    :
</doc>
```

Document B

```
<doc>
    :
  <basic>
    <date>03/02/2004</date>

  </basic>
  <detail>
    :
    <area>Abenomotomachi, Abeno-ku, Osaka-shi, Osaka</area>
    :
  </detail>
    :
</doc>
```

Document C

```
<doc>
    :
  <basic>
    <date>13/01/2004</date>
    :
  </basic>
  <detail>
    :
    <area>Chuo-ku, Sapporo-shi, Hokkaido</area>
    :
  </detail>
    :
</doc>
```

Document D

```
<doc>
    :
  <basic>
    <date></date>
    :
  </basic>
  <detail>
    :
    <area>Kakuozan, Chikusa-ku, Nagoya-shi, Aichi</area>
    :
  </detail>
    :
</doc>
```

Example 1

The following example shows how to sort documents in ascending order according to the trip date (*date*):

```
/doc/basic/date/text()
```

Result

Documents are returned in the order C, A, B, D.

Example 2

To sort character codes in descending order, specify ‘DESC’ after the key specification.

The following example shows how to sort documents in descending order according to the trip date (*date*):

```
/doc/basic/date/text() DESC
```

Result

Documents are returned in the order B, A, C, D.

Note

If there are no text nodes in an XML document specified by a text expression, that XML document will be returned last, regardless of whether a 'DESC' specification is in place. In the above example, Document D would be returned last in both an ascending sort and a descending sort.

Example 3

When a sort is performed using a character string, the first 20 bytes of that string are used as the sort key. Therefore, if the length of the character string specified as the key is greater than 20 bytes, bytes 21 onwards will not become part of the key, and the XML documents will not be sorted correctly. In such cases, the rlen function can be used in the key specification to determine how many characters from the start of the string will be used as the sort key. Refer to Single-line Function Specifications in Appendix A for more information on the *rlen* function.

The following example shows how to sort documents according to the trip area (area):

```
rlen(/doc/detail/area/text(),30)
```

When the above is specified, the first 30 characters of the trip area will be used as the sort key.

Sorting by Numeric Value

It is possible to sort character strings in element nodes of XML documents as if they were numeric values. To do so, specify the single-line *val* function in the key specification in the sort expression. A text expression is specified as the argument of the *val* function. Refer to Single-line Function Specifications in Appendix A for more information on single-line function specifications.

The explanation that follows assumes that the following documents exist:

Document A

```
<doc>
    :
  <basic>
    :
    <expense>7650 yen</expense>
  </basic>
</doc>
```

Document B

```
<doc>
    :
  <basic>
    :
    <expense>12980 yen</expense>
  </basic>
</doc>
```

Document C

```
<doc>
  <basic>
    :
```

```
    <expense>No trip expenses required</expense>
  </basic>
</doc>
```

Document D

```
<doc>
  <basic>
    :
    <expense>480 yen</expense>
  </basic>
</doc>
```

Example 1

The following example shows how to sort documents in ascending order according to the trip expenses (*expense*):

```
val(/doc/basic/expense/text())
```

Result

Documents are returned in the order C, D, A, B.

Notes

- If the character string in the text node in the XML document specified by the text expression does not contain a numeric value, the *val* function will treat the value of the node as 0. In this example, the trip expenses in Document C will be treated as 0.
- If the text node in the XML document specified by the text expression does not exist, that XML document will be returned last.

Example 2

Specifying ‘DESC’ after the key specification will sort numeric values in descending order.

The following example shows how to sort documents in descending order according to the trip expenses (*expense*):

```
val(/doc/basic/expense/text()) DESC
```

Result

Documents are returned in the order B, A, D, C.

Sorting with Multiple Keys

It is possible to perform a sort using the values of more than one element node in an XML document. Commas are used to separate key specifications in the sort expression. The keys can be either a numeric or character string.

Multiple keys make it possible to perform more sophisticated sorting operations. For example, XML documents can be sorted in ascending order using one element node and, if that element node is the same in more than one document, the documents can be further sorted in ascending order using a second element node.

Up to eight keys can be specified.

The explanation that follows assumes that the following documents exist:

Document A

```
<doc>
  <basic>
    <date>03/03/2004</date>
    :
    <expense>7650 yen</expense>
  </basic>
    :
</doc>
```

Document B

```
<doc>
  <basic>
    <date>03/03/2004</date>
    :
    <expense>11500 yen</expense>
  </basic>
    :
</doc>
```

Document C

```
<doc>
  <basic>
    <date>10/03/2004</date>
    :
    <expense>7650 yen</expense>
  </basic>
    :
</doc>
```

Example

To sort documents according to the trip date (*date*) and, if multiple documents contain the same trip date, then sort in descending order according to the trip expenses (*expense*):

```
/doc/basic/date/text(),val(/doc/basic/expense/text()) DESC
```

Result

Documents are returned in the order B, A, C.

Extracting Search Results

It is possible to search XML documents stored in Shunsaku and extract data in XML format or in text format. The search results are extracted using the return expression in the APIs provided by Shunsaku.

Refer to Formats Used When Not Performing Aggregation in Appendix A for more information on the return expression.

Extracting Data in XML Format

Extracting the XML documents found by a search in XML format enables XML tools such as DOM and SAX to be used.

To extract data in XML format, specify a path expression in the return parameter. This can be done in two ways, as shown below. Use the method appropriate to the situation.

- Extracting an Entire XML Document

  Extract the original XML document stored in Shunsaku without changing its format. This is the extraction method most often used.

- Extracting Data in XML Format by Specifying an Element Node

  Extract specified element nodes in XML format.

Extracting an Entire XML Document

An entire XML document can be extracted by specifying only '/' in the return expression.

Example

To extract an entire XML document:

```
/
```

Result

```
<doc>
  <employee>
    <eno>19980120</eno>
    <name>Taro Suzuki</name>
    <sno>1001</sno>
    <phone>2201-1101</phone>
    <email>suzuki.taro@shunsaku.fujitsu.com</email>
  </employee>
  <basic>
    <date>16/02/2004</date>
    <expense>7200 yen</expense>
  </basic>
  <detail>
    <destination>Head Office</destination>
    <area>Kohoku-ku, Yokohama-shi, Kanagawa</area>
    <purpose>Regular project meeting</purpose>
    <train>6600 yen</train>
    <taxi>600 yen</taxi>
    <hotel></hotel>
    <comment>None</comment>
  </detail>
```

```
  <report>Must create and report a sales results chart by next
meeting</report>
</doc>
```

Note

When a sort expression is specified, XML documents found using the search expression will be returned in the order determined by the sort expression.

Extracting Data in XML Format by Specifying an Element Node

It is possible to specify a path expression in the return parameter to extract data under any node of an XML document in XML format. The data in the element node indicated by the path expression is expressed in XML format as an element under the root tag of the XML document.

The explanations accompanying the examples that follow assume that the following single XML document has been found using a search expression.

```
<doc>
  <employee>
    <eno>19980120</eno>
    <name>Taro Suzuki</name>
    <sno>1001</sno>
    <phone>2201-1101</phone>
    <email>suzuki.taro@shunsaku.fujitsu.com</email>
  </employee>
  <basic>
    <date>16/02/2004</date>
    <expense>7200 yen</expense>
  </basic>
  <detail>
    <destination>Head office</destination>
    <area>Kohoku-ku, Yokohama-shi, Kanagawa</area>
    <purpose>Regular project meeting</purpose>
    <train>6600 yen</train>
    <taxi>600 yen</taxi>
    <hotel></hotel>
    <comment>None</comment>
  </detail>
  <report>Must create and report a sales results chart by next
meeting</report>
</doc>
```

Example 1

The following example shows how to extract the employee's name (*name*).

```
/doc/employee/name
```

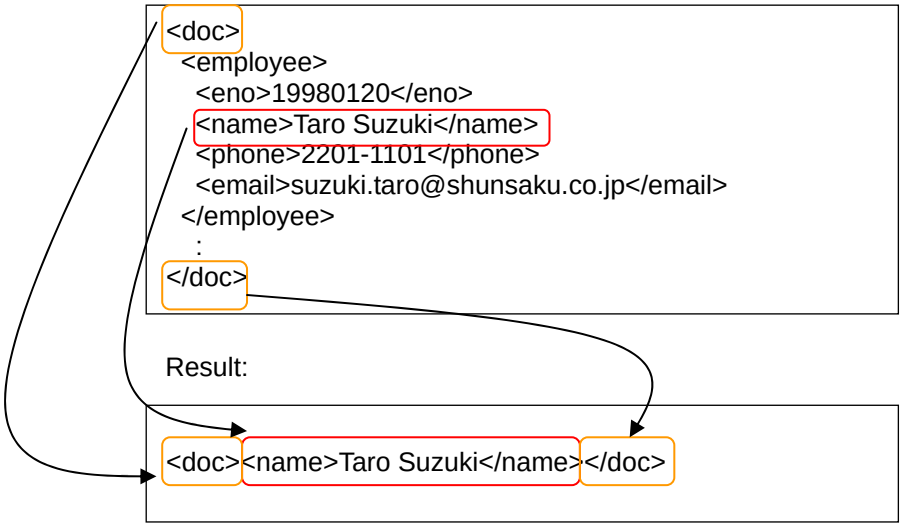


Figure 3-2 Extracting Data in XML Format

More than one return parameter can be specified in the return expression. Each path expression must be separated using a comma.

Example 2

The following example shows how to extract the employee name (*name*), basic information (*basic*) and the trip purpose (*purpose*).

```
/doc/employee/name,/doc/basic,/doc/detail/purpose
```

Result

```
<doc><name>Taro Suzuki</name><basic>
    <date>16/02/2004</date>
    <expense>7200 yen</expense>
  </basic><purpose>Regular project meeting</purpose></doc>
```

Notes

- When a sort expression is specified, XML documents found using the search expression will be returned in the order determined by the sort expression.
- When multiple return parameters are specified, they must all be in the form of a path expression. They cannot be specified together with text expressions or single line function specifications (result is returned in XML format).
- If a return parameter that can match multiple elements is specified (ie. more than one path expression, '//' or '*' is specified), the application will not be able to determine the relationship between the elements that have been extracted.
  - If some elements do not exist, the application will not be able to determine which they are.
  - The application will not be able determine the path to the extracted elements.

  In these situations, either extract the entire XML document, or extract data using a specification where the return parameter isolates a single element.

Extracting Data in Text Format

It is possible to specify a text expression in the return expression to extract data under any node of an XML document in text format.

It is also possible to include a single-line function specification in the return expression. When a single-line function specification is used, the results of the *rlen* function or the *val* function can be extracted in text format.

Refer to Single-line Function Specifications in Appendix A for more information on single-line function specifications.

The explanations accompanying the examples that follow assume that the following single XML document has been found using a search expression.

```
<doc>
  <employee>
    <eno>19980120</eno>
    <name>Taro Suzuki</name>
    <sno>1001</sno>
    <phone>2201-1101</phone>
    <email>suzuki.taro@shunsaku.fujitsu.com</email>
  </employee>
  <basic>
    <date>16/02/2004</date>
    <expense>7200 yen</expense>
  </basic>
  <detail>
    <destination>Head Office</destination>
    <area>Kohoku-ku, Yokohama-shi, Kanagawa</area>
    <purpose>Regular project meeting</purpose>
    <train>6600 yen</train>
    <taxi>600 yen</taxi>
    <hotel></hotel>
    <comment>None</comment>
  </detail>
  <report>Must create and report a sales results chart by next
meeting</report>
</doc>
```

Example 1

The following example shows how to extract a trip report (*report*):

```
/doc/report/text()
```

Result

```
Must create and report a sales results chart by next meeting
```

Example 2

The following example shows how to extract numeric values from trip expenses (*expense*):

```
val(/doc/basic/expense/text())
```

Result

```
7200
```

Example 3

More than one return parameter can be specified in the return expression. Each return parameter must be separated with a comma. If multiple return parameters are specified, the values of the results returned by those parameters are separated with a delimiter.

- When the Java APIs are used, the delimiter is a comma.
- When the C APIs are used, the delimiter is the character represented by character code '\001'.

The following example shows how to extract the employee number (*eno*), the employee name (*name*), the trip date (*date*) and the trip purpose (*purpose*):

```
/doc/employee/eno/text(),/doc/employee/name/text(),/doc/basic/date/text(),
/doc/detail/purpose/text()
```

Result

When the Java APIs are used

```
19980120,Taro Suzuki,16/02/2004,Regular project meeting
```

Notes

- When a sort expression is specified, the search results will be returned in the order determined by the sort expression.
- When multiple return parameters are specified, all return parameters must be specified as either text expressions or single-line function specifications. They cannot be specified together with a path expression (result returns in XML format).

Extracting Aggregated Results

XML documents found using a search expression can be grouped and aggregated according to a given key. To perform aggregation, include the aggregation function specification in the return expression. A group key must also be specified in the sort expression. To produce the results of the aggregation function specification, XML documents with the same group key are treated as a single group. The data indicated by the text expression specified as the argument of the aggregation function specification is treated as a numeric value for each group, and then totals, averages and other values are determined.

Refer to Formats Used When Performing Aggregation in Appendix A for more information on the return expression.

The following figure provides an overview of the aggregation process.

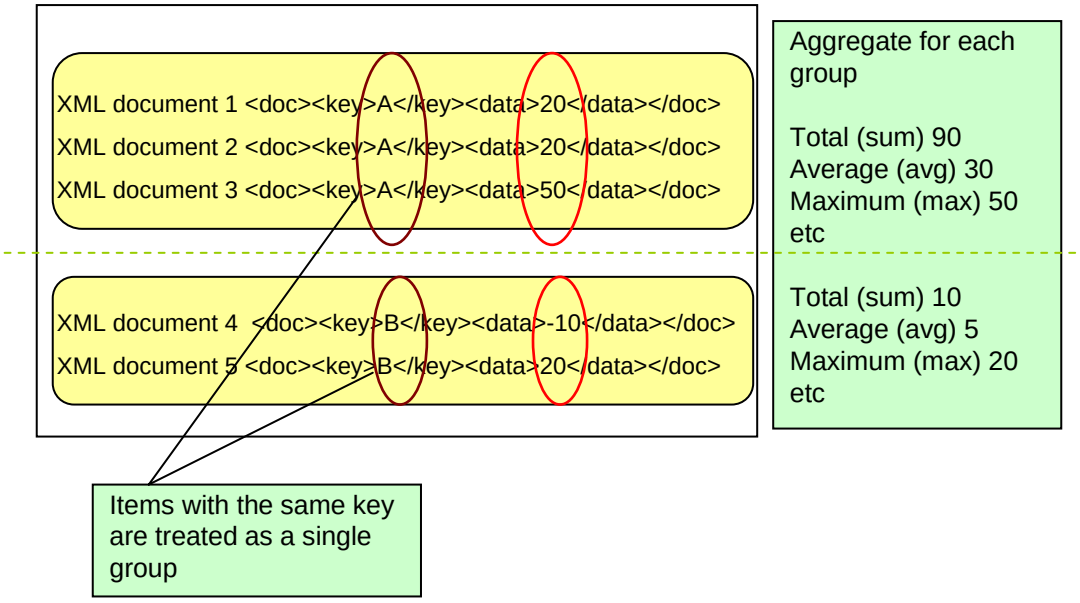


Figure 3-3 Overview of the Aggregation Process

This is the method used to decide how search results will be grouped, and what will be aggregated.

- Grouping Search Results
- Aggregating Search Results

Grouping Search Results

XML documents found using a search expression are grouped according to a given key. The key used to perform the grouping is specified in the sort expression.

Grouping by Numeric Value

Search results are grouped by treating the value of an element node in an XML document as a numeric value. To group by numeric values, specify the single-line *val* function in the key specification of the sort expression. Specify a text expression as the argument of the *val* function.

The explanation that follows assumes that the following documents exist:

Document A

```
<doc>
  <employee>
    :
    <sno>2001</sno>
    :
  </employee>
    :
</doc>
```

Document B

```
<doc>
  <employee>
    :
    <sno>2002</sno>
    :
  </employee>
    :
</doc>
```

Document C

```
<doc>
  <employee>
    :
    <sno>2001</sno>
    :
  </employee>
    :
</doc>
```

Document D

```
<doc>
  <employee>
    :
    <sno>2002</sno>
    :
  </employee>
    :
</doc>
```

Example

The following example shows how to group XML documents that contain the same section number (*sno*):

```
val(/doc/employee/sno/text())
```

Result

Documents A and C are handled as one group, and documents B and D are handled as another group.

Grouping by Character String

Grouping can be performed using the value of an element node in an XML document. To group by a character string, specify a text expression in the key specification of the sort expression.

The explanation that follows assumes that the following documents exist:

Document A

```
<doc>
  <employee>
    :
    <name>Taro Suzuki</name>
    :
  </employee>
    :
  <detail>
    :
    <area>Kakuozan, Chikusa-ku, Nagoya-shi, Aichi </area>
    :
  </detail>
    :
</doc>
```

Document B

```
<doc>
  <employee>
    :
    <name>Taro Suzuki</name>
    :
  </employee>
    :
  <detail>
    :
    <area>Yaesu, Chiyoda-ku, Tokyo</area>
    :
  </detail>
    :
</doc>
```

Document C

```
<doc>
  <employee>
```

```
    :
    <name>Hanako Sato</name>
    :
  </employee>
    :
  <detail>
    :
    <area>Yaesu, Chiyoda-ku, Tokyo</area>
    :
  </detail>
    :
</doc>
```

Document D

```
<doc>
  <employee>
    :
    <name>Hanako Sato</name>
    :
  </employee>
    :
  <detail>
    :
    <area>Imaike, Chikusa-ku, Nagoya-shi, Aichi</area>
    :
  </detail>
    :
</doc>
```

Example 1

The following examples shows how to group according to the employee name (*name*):

```
/doc/employee/name/text()
```

Result

Documents A and B are handled as one group, and documents C and D are handled as another group.

Example 2

The process of grouping by a character string takes place using the first 20 bytes of the string as a key. Therefore, if the value in the XML document that is indicated by the key exceeds 20 bytes, differences in character strings from byte 21 onwards will be ignored, and dissimilar documents may end up being placed in the same group. In such cases, the *rlen* function can be included in the key specification to specify the number of characters from the start of the string that will be used as the key. Refer to Single-line Function Specifications in Appendix A for more information on the *rlen* function.

The following example shows how to group according to the trip area (*area*):

```
rlen(/doc/detail/area/text(),30)
```

When the above is specified, grouping will occur using the first 30 characters of the trip area.

Grouping by Multiple Keys

To group by multiple keys, specify the key specifications in the sort expression by separating them with commas. Each key specification can be specified as either a numeric or character string. When multiple key specifications are specified, XML documents that match all the key values will be handled as a single group. Up to eight key specifications can be specified.

The explanation that follows assumes that the following documents exist:

Document A

```
<doc>
  <basic>
    <date>03/03/2004</date>
    :
  </basic>
  <detail>
    :
    <destination>Head Office</destination>
    :
  </detail>
    :
</doc>
```

Document B

```
<doc>
  <basic>
    <date>03/03/2004</date>
    :
  </basic>
  <detail>
    :
    <destination>Head Office</destination>
    :
  </detail>
    :
</doc>
```

Document C

```
<doc>
  <basic>
    <date>10/03/2004</date>
    :
  </basic>
  <detail>
    :
    <destination>Head Office</destination>
    :
  </detail>
    :
</doc>
```

Example

The following example shows how to group documents according to the trip destination (*destination*) and the trip date (*date*):

```
/doc/detail/destination/text(),/doc/basic/date/text()
```

Result

Documents A and B are handled as one group, and document C is handled as another group.

Aggregating Search Results

When an aggregation is performed, a group key is also extracted at the same time. The aggregation function specification and the key specified by the sort expression can be included in the return expression. Each return parameter is separated with a comma. If multiple return parameters are specified, the values returned by those parameters are separated with a delimiter.

- When the Java APIs are used, the delimiter is a comma.
- When the C APIs are used, the delimiter is the character represented by character code '\001'.

The explanations accompanying the examples that follow assume that the following six XML documents have been found using a search expression.

Document A

```
<doc>
    :
  <detail>
    <destination>Head Office</destination>
    :
    <train>8600 yen</train><taxi></taxi><hotel>6800 yen</hotel>
    :
  </detail>
    :
</doc>
```

Document B

```
<doc>
    :
  <detail>
    <destination>Head Office</destination>
    :
    <train>900 yen</train><taxi></taxi><hotel></hotel>
    :
  </detail>
    :
</doc>
```

Document C

```
<doc>
    :
```

```
  <detail>
    <destination>Head office</destination>
    :
    <train>13000 yen</train><taxi></taxi><hotel>8000 yen</hotel>
    :
  </detail>
    :
</doc>
```

Document D

```
<doc>
    :
  <detail>
    <destination>Tokyo Office</destination>
    :
    <train>1600 yen</train><taxi>600 yen</taxi><hotel></hotel>
    :
  </detail>
    :
</doc>
```

Document E

```
<doc>
    :
  <detail>
    <destination>Tokyo Office</destination>
    :
    <train>400 yen</train><taxi>690 yen</taxi><hotel></hotel>
    :
  </detail>
    :
</doc>
```

Document F

```
<doc>
    :
  <detail>
    <destination>Tokyo Office</destination>
    :
    <train>280 yen</train><taxi>600 yen</taxi><hotel>8200 yen</hotel>
    :
  </detail>
    :
</doc>
```

It is assumed that the following sort expression has been specified in all the examples that follow:

```
Sort expression: /doc/detail/destination/text()
```

Note

The sort expression used for aggregation specifies the grouping key. Sorting is also performed using that key. 'DESC' can also be specified in a sort expression used for aggregation. When 'DESC' is specified, aggregation results can be extracted in descending order according to the size of the grouping key.

Example 1

The following example shows how to find the average of traveling expenses (*train*):

```
Return expression:
/doc/detail/destination/text(),avg(/doc/detail/train/text())
```

Result

When the Java APIs are used

```
Tokyo Office,760
Head Office,7500
```

Example 2

The following example shows how to count the number of people who used taxis:

```
Return expression:
/doc/detail/destination/text(),count(/doc/detail/taxi/text())
```

Result

When the Java APIs are used

```
Tokyo Office,3
Head Office,0
```

Note

If no value is indicated by the text expression used as the argument of the aggregation function, it will not be targeted for aggregation. In the above example, no one used a taxi to travel to the head office, so the count result is 0.

If the value '0' is stored in the taxi element node, it will be targeted for aggregation. To prevent data stored in this way from being aggregated, the data can be excluded from the search process by specifying in the search expression the condition that the taxi fare must be greater than 0.

Example 3

This example shows how to obtain the maximum taxi fare (*taxi*), the total accommodation costs (*hotel*), and the number of trips:

```
Return expression :
/doc/detail/destination/text(),max(/doc/detail/taxi/text()),sum(/doc/detai
l/hotel/text()),count(/doc/detail/destination/text())
```

Result

When the Java APIs are used

```
Tokyo Office,690,8200,3
Head Office,,14800,3
```

Note

If no value is indicated by the text expression used as the argument of the aggregation function, it will not be targeted for aggregation. In the above example, no one used a taxi to visit the head office, so there is no maximum value.

Chapter 4

How to Update Data

This chapter explains how to use the Shunsaku APIs to update data.

- Overview
- Adding Data
- Deleting Data

Overview

This section explains the basic concepts of updating XML documents stored in Shunsaku.

- XML documents can be updated either from an application via the API or using a command.
- XML documents can be added or deleted.

  The process of updating XML documents stored in Shunsaku can be used to add or delete XML documents.

XML document updates are processed for each director.

Shunsaku automatically detects if more than one update process is performed simultaneously on a single director. When Shunsaku detects such a conflict between API-based update processes, it will perform the processes in order. If a conflict occurs between command-based update processes or between a command-based update process and an API-based process, Shunsaku will perform one of the processes and return an error to the other process. Table 4-1 gives details of Shunsaku operations when conflicts occur.

When an error occurs, its cause can be determined by examining the messages output by each director.

Refer to the User's Guide for more information on directors.

Table 4-1 Operations that Occur when Update Processes Conflict

Preceding process	Subsequent process	Operation when update processes conflict
API-based update	API-based update	The subsequent update process is performed after the preceding update process has completed.
API-based update	Command-based update	The command of the subsequent process generates an error.
Command-based update	API-based update	The API of the subsequent process generates an error.
Command-based update	Command-based update	The command of the subsequent process generates an error.

Note: The *shundimport* command is used to perform command-based updates. Refer to the User's Guide for more information on the *shundimport* command.

Adding Data

To add data using the API, specify an XML document. To add more than one XML document, either place the XML documents in a single area, or place the XML documents in multiple areas and specify them together.

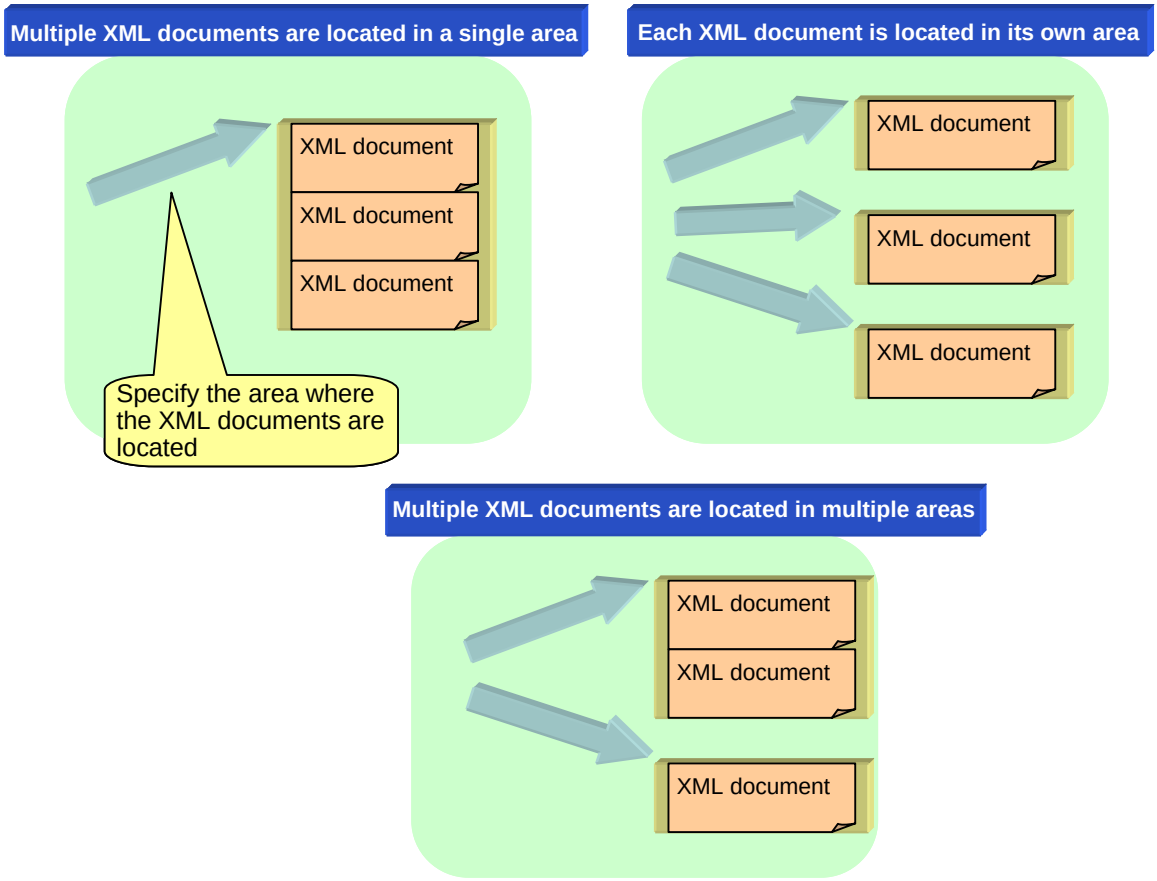


Figure 4-1 How to Specify Multiple XML Documents when Adding Data Using the API

When more than one director is used, the search-initiating director specified by the StartPoint parameter in the conductor environment file is treated as the starting point, and XML documents are added to the director in the last position.

Refer to the User's Guide for more information on the conductor environment file.

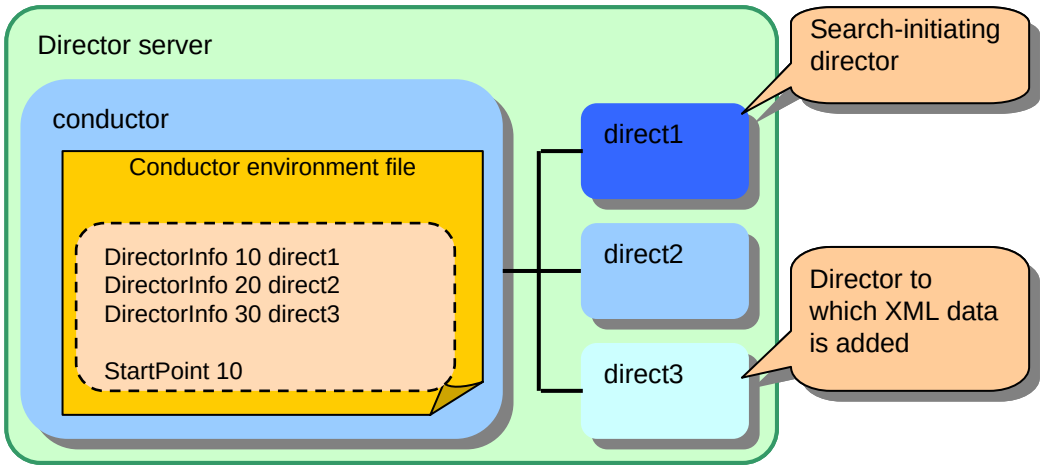


Figure 4-2 When using More than One Director

Deleting Data

The following procedure is used to delete data via the API:

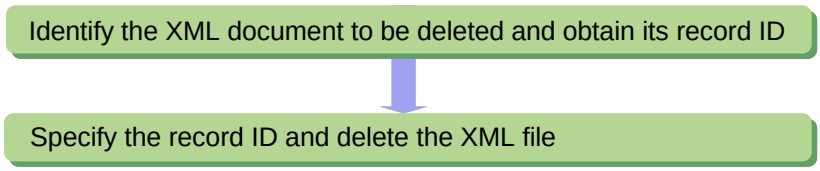


Figure 4-3 Procedure to Delete Data

1. Identify the XML document to be deleted and obtain its record ID.
   Perform a data search to identify the XML document to be deleted, and obtain its record ID.
2. Specify the record ID and delete the XML document.
   Specify the obtained record ID to invoke the API that will perform the deletion, and delete the XML document.

More than one record ID can be specified at once to delete multiple XML documents. If more than one director is in use, the record IDs of the XML documents to be deleted can be specified without making any distinction between directors. Shunsaku will gather the record IDs of the XML documents to be deleted for each director and perform the deletion process on each director. Even if an error is detected in a director, the deletion process will continue until its completion and the API will return the error. At that time, the user should examine the messages output by each director to identify and eliminate the cause of the error, then repeat the deletion process by performing another search for XML documents to be deleted.

Chapter 5

Java Application Development

This chapter explains how to develop applications that use the Java APIs provided by Shunsaku.

- Java API Overview
- How to Use Java APIs
- Character Encoding Used by Java APIs
- Error Codes Output when Java APIs are Used

Java API Overview

The Java APIs are interfaces used to manipulate Shunsaku data from applications written in Java.

The following table lists the Java API classes.

Table 5-1 Java API Classes

Class	Description
ShunConnection	Opens and closes connections.
ShunPreparedRecordID	Performs data searches and data deletions based on record identifiers. Shunsaku data can be uniquely identified using record identifiers. Record identifiers provide valid information so long as the records that they refer to still exist, even after the connection to Shunsaku has been closed.
ShunPreparedStatement	Performs data searches and data additions based on search expressions, return expressions and sort expressions.
ShunResultSet	Looks up the results of the search.

Refer to the Java API Reference for more information on the Java APIs provided by Shunsaku.

The Java APIs are used to create objects. The following table lists the objects created by the Java APIs.

Table 5-2 Objects Created by the Java APIs

Object name	Description
ShunConnection	This object is created when a connection is opened.
ShunPreparedRecordID	This object is used to perform data searches and data deletions based on record identifiers, using the ShunConnection object.
ShunPreparedStatement	This object is used to perform data searches and data additions based on search expressions, return expressions and sort expressions, using the ShunConnection object.
ShunResultSet	This object is used to look up the results of the search, using either the ShunPreparedStatement object or the ShunPreparedRecordID object.

How to Use Java APIs

This section explains how to use the Java APIs.

Opening Connections

To open connections, create a ShunConnection object using either of the following two methods:

- Specify the host name and port number in a Java Properties object
- Specify the host name and port number directly

Specifying the Host Name and Port Number in a Java Properties Object

This section explains how to open a connection by specifying the host name and port number in a Java Properties object.

Entry Format for the Java Properties Object

```
connection.host=host-name or IP-address
connection.port=port-number
```

Entry Items

connection.host

Specify either the host name or the IP address of the connection destination.

connection.port

Specify the port number of the connection destination.

Entry Example of the Java Properties Object

When the host name is 'DServer' and the port number is '33101'

```
connection.host=DServer
connection.port=33101
```

Entry Example

```
// Specify the file name
String sFileName = "Property.txt";

// Load the host name and port number from the file
Properties property = new Properties();
property.load(new FileInputStream(sFileName));

// Create a ShunConnection object
ShunConnection con = new ShunConnection(property);
```

Specifying the Host Name and Port Number Directly

This section explains how to open a connection by specifying the host name and port number directly.

Syntax

```
ShunConnection object-name = new ShunConnection (host-name, port-number);
```

Arguments

host-name

Specify either the host name or the IP address of the connection destination.

port-number

Specify the port number of the connection destination.

Entry Example

When the host name is 'DServer' and the port number is '33101'

```
ShunConnection con = new ShunConnection("DServer", 33101);
```

Searching Data

Java APIs can be used to perform the following operations:

- Finding the number of XML documents that match the search conditions
- Obtaining the XML documents that match the search conditions in a specified format
- Obtaining all of a particular XML document
- Finding XML documents that match the search conditions and obtaining the documents after they are sorted
- Finding XML documents that match the search conditions and obtaining the documents after their contents are aggregated

These operations can be combined to create a wide range of applications. Refer to Searching Data in Appendix B for sample programs used for searching data. The rest of this section will describe how to create applications that search for data.

Obtaining Search Results According to the Number of Data Items

Web-based search applications usually display only a few dozen search results per page, rather than displaying all of the search results in a single window.

In such cases, the number of data items to be obtained can be controlled by passing the reply start number ('position') and the number of items to return per request ('requestCount') as parameters to the setRequest method.

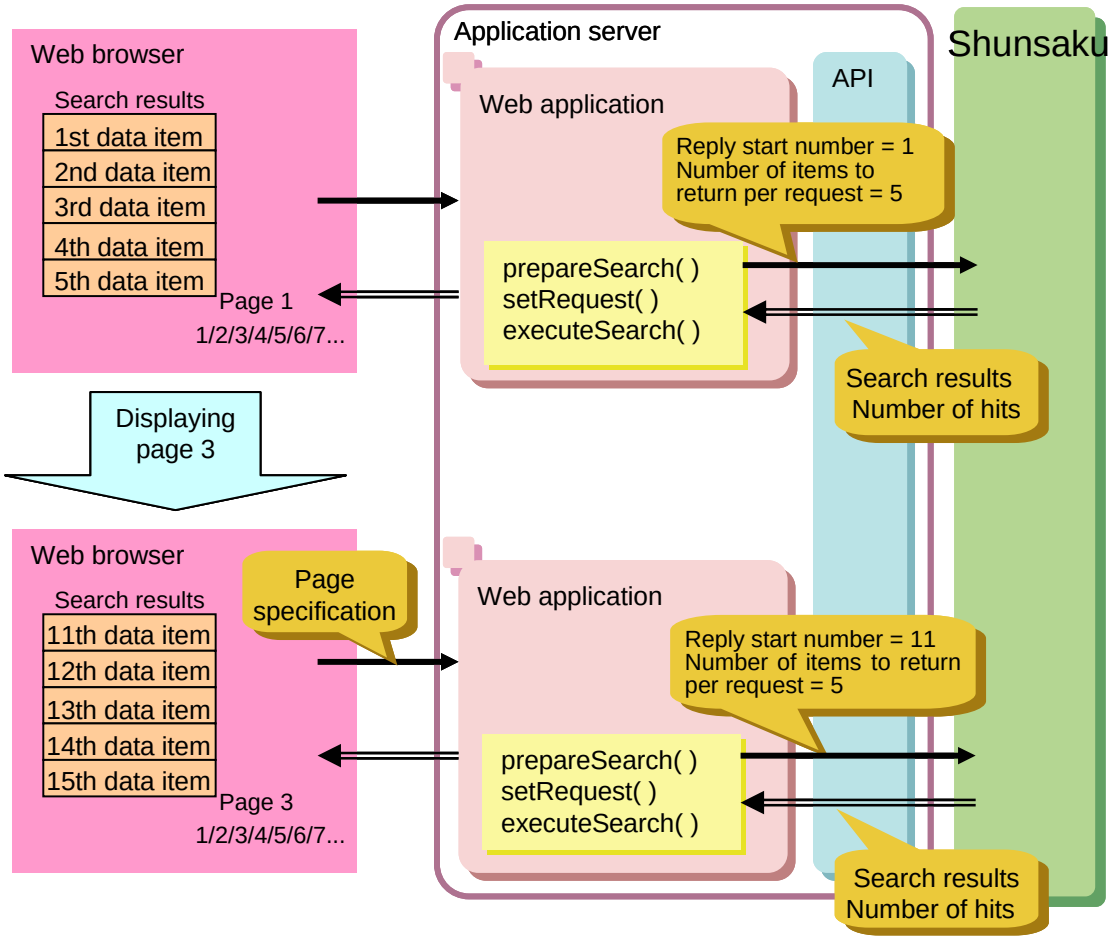


Figure 5-1 Obtaining Search Results According to the Number of Data Items

Entry Example

```
ShunConnection con = new ShunConnection();

String query = "/document/base/prefecture == 'Osaka'";
String returnQuery = "/document/base/name, /document/base/price ";
ShunPreparedStatement pstmt = con.prepareSearch(query, returnQuery);    (1)
pstmt.setRequest(11,5);                                                 (2)
ShunResultSet rs = pstmt.executeSearch();                               (3)
System.out.println("[Number of hits] = " + rs.getHitCount());           (3)
while(rs.next()) {                                                      (4)
  System.out.println("[Search results] = " + rs.getString());           (4)
}
rs.close();                                                             (5)
```

```
pstmt.close();                                                    (5)

con.close();
```

(1) Create a ShunPreparedStatement Object

Create a ShunPreparedStatement object by specifying a search expression and a return expression as parameters of the prepareSearch method. Refer to Appendix A, Format of Search, Return and Sort Expressions for more information about search expressions and return expressions.

(2) Set the Reply Start Number and the Number of Items to Return per Request

Specify the reply start number and the number of items to return per request in the setRequest method. If the setRequest method is omitted, the number of items to return per request will be set to the value specified in the AnsMax parameter in the conductor or director environment file.

Note

For the number of items to return per request, specify the number of data items to display on each page.

(3) Execute the Search (Create a ShunResultSet Object)

Execute the search using the executeSearch method. A ShunResultSet object will be created to hold the results of the search.

Note

The number of hits (XML documents that match the search conditions) can be obtained using the getHitCount method. This value can be used to find such things as the number of pages of the search results.

(4) Extract the Results of the Search

Always invoke the next method before extracting the results of the search. The next method returns true if there is still more data that can be extracted and false otherwise.

Use one of the getXXX methods to extract the XML documents. The following table shows the methods that can be used.

Method	Function
getString	Extracts XML documents as String objects.
getStringArray	Extracts XML documents as a two-dimensional array of String objects.
getStream	Extracts XML documents as InputStream objects.

Notes

The getStringArray method can be used to extract the search results in text format.

The getRecordID method can be used to obtain record identifiers (that uniquely identify each data item) as well as the results of the search. Record identifiers are used to extract or delete the corresponding entire XML documents.

(5) Close the ShunResultSet Object and the ShunPreparedStatement Object

When they are no longer required, always close the objects using the close methods of the ShunResultSet object and the ShunPreparedStatement object.

Obtaining Search Results While Adding Search Conditions

When a search produces a large number of hits, it is sometimes useful to be able to narrow down the scope of the search by adding more search conditions.

In such cases, perform a new search process by creating a new search expression and adding extra search conditions to the search expression specified with the prepareSearch method. By repeating this operation, the user can narrow down the search results while referring to the search results displayed on the screen.

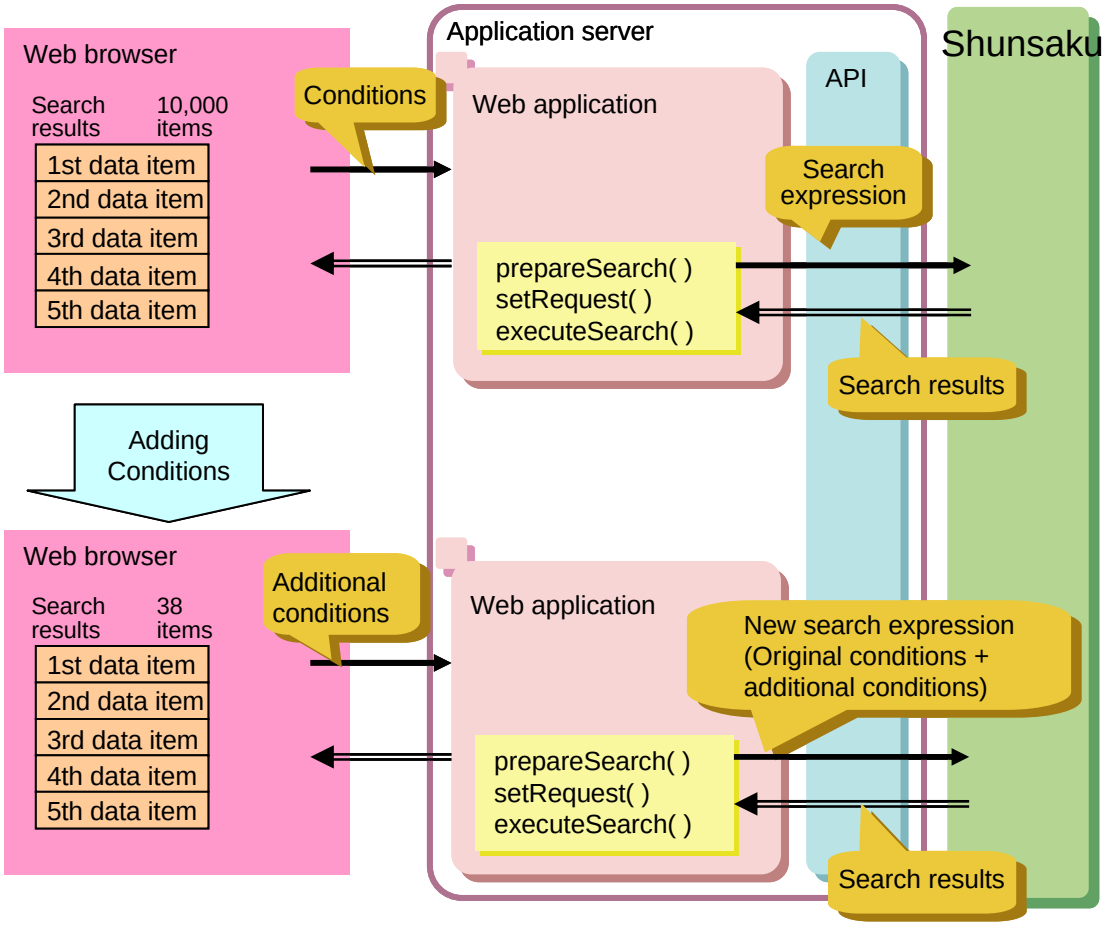


Figure 5-2 Obtaining Search Results While Adding Search Conditions

Obtaining Entire XML Documents

When searching for a particular XML document, do not obtain the entire document straightaway. Instead, start by obtaining the partial information that can effectively identify the document. The user can then use this partial information to pick out the desired XML document and obtain detailed information. To extract an entire XML document, use the record identifier that is returned when the partial information is extracted. The entire target XML document can be extracted by using this record identifier.

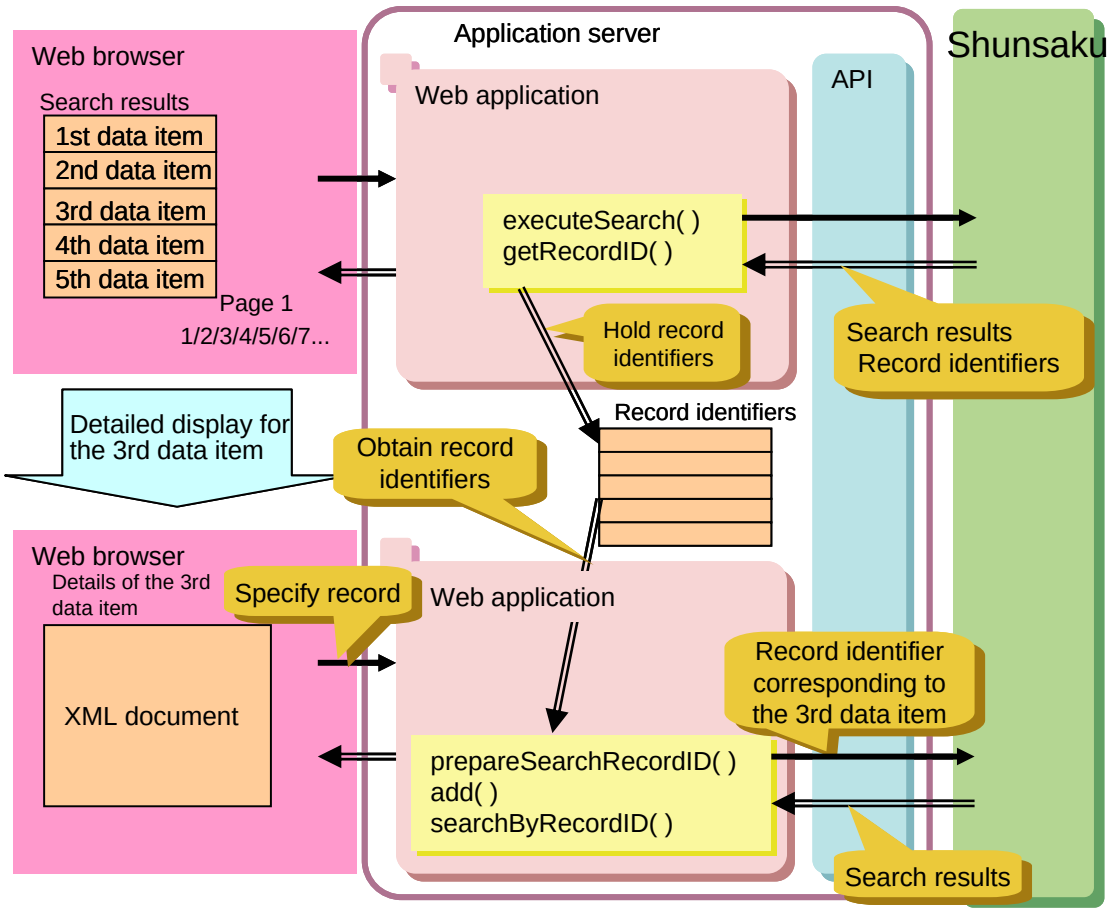


Figure 5-3 Obtaining Entire XML Documents

Entry Example

```
ShunConnection con = new ShunConnection();

ShunPreparedRecordID prid = con.prepareSearchRecordID();              (1)
prid.add(recordID);                                                   (2)
ShunResultSet rs = prid.searchByRecordID();                           (3)
while (rs.next()) {                                                   (4)
  System.out.println("[Result] = " + rs.getString());                 (4)
}
rs.close();                                                           (5)
prid.close();                                                         (5)

con.close();
```

(1) Create a ShunPreparedRecordID Object

Use the prepareSearchRecordID method to create a ShunPreparedRecordID object.

(2) Specify the Record Identifier

Use the add method to specify the record identifier. Record identifiers can be obtained using the getRecordID method.

Multiple record identifiers can be specified with the add method. Any existing record identifiers will be overwritten if they are specified again.

Note

By specifying multiple record identifiers with the add method, multiple XML documents can be obtained at once.

(3) Execute the Search (Create a ShunResultSet Object)

Execute the search using the searchByRecordID method. A ShunResultSet object will be created to hold the results of the search.

(4) Extract the Results of the Search

Always invoke the next method before extracting the results of the search. The next method returns true if there is still more data that can be extracted and false otherwise.

Use one of the getXXX methods to extract the XML documents. The following table shows the methods that can be used.

Method	Function
getString	Extracts XML documents as String objects.
getStream	Extracts XML documents as InputStream objects.

(5) Close the ShunResultSet Object and the ShunPreparedRecordID Object

When they are no longer required, always close the objects using the close methods of the ShunResultSet object and the ShunPreparedRecordID object.

Obtaining Sorted Data

Sometimes it is useful to be able to sort the results of a search, using a particular element as a sort key.

To obtain sorted partial information about the data, use the setSort method.

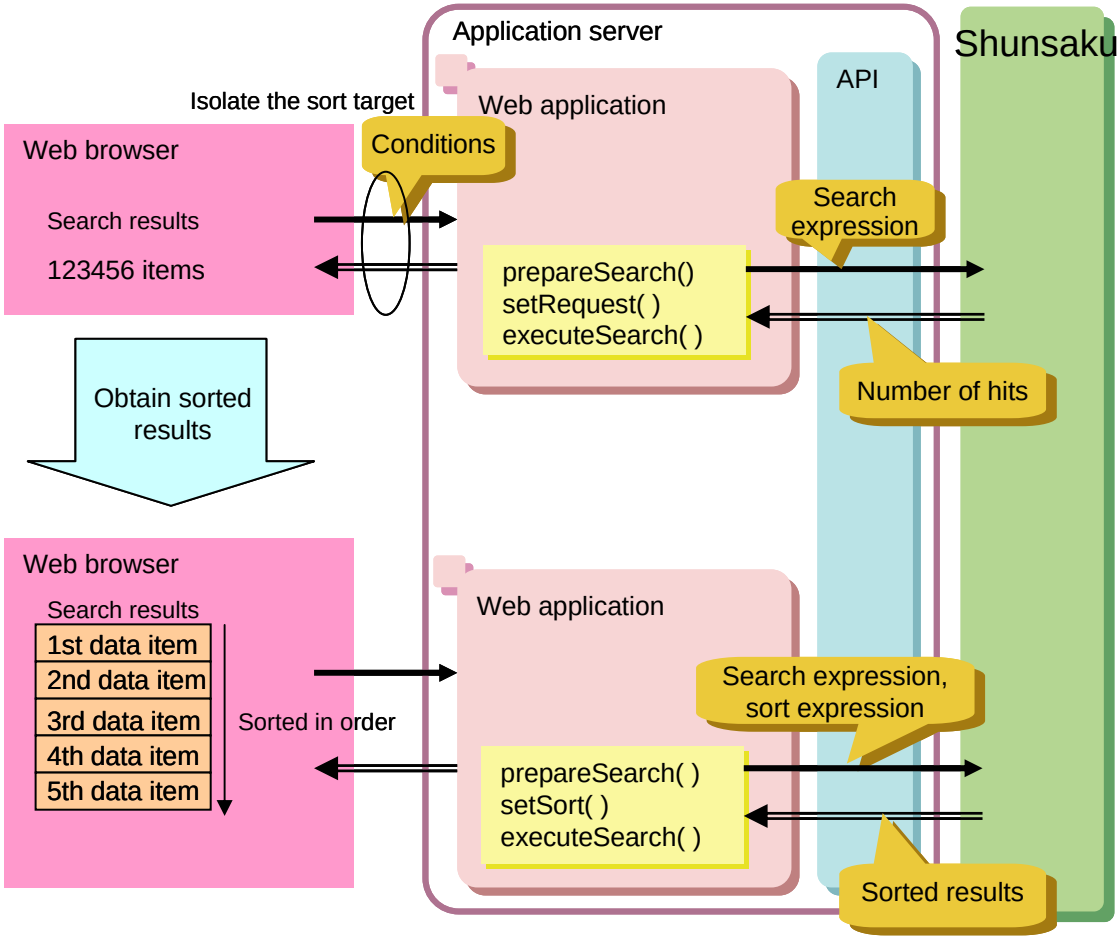


Figure 5-4 Obtaining Sorted Data

Note

When search results are sorted, the sort process looks up all of the XML documents that match the search conditions. To improve response performance, it is important to narrow down the data to be sorted by specifying a search expression that produces an appropriate number of hits. To find out how many XML documents match the search conditions before starting a sort process, specify 0 in the requestCount parameter (the number of items to return per request) of the setRequest method.

Entry Example

```
ShunConnection con = new ShunConnection();

String query = "/document/base/prefecture == 'Osaka'";
String returnQuery = "/document/base/name, /document/base/price ";
String sort = "val(/document/base/price/text()) DESC";              (1)
ShunPreparedStatement pstmt = con.prepareSearch(query, returnQuery);
                                                                    (2)
pstmt.setSort(sort);                                                (3)
```

```
ShunResultSet rs = pstmt.executeSearch();                          (4)
 while(rs.next())
{                                                                  (5)
  System.out.println("[Search results] = " + rs.getString());     (5)
}
rs.close();                                                        (6)
pstmt.close();                                                     (6)

con.close();
```

(1) Create a Sort Expression

Create a sort expression. Refer to Sort Expressions in Appendix A for more information on sort expressions.

(2) Create a ShunPreparedStatement Object

Specify a search expression and a return expression with the prepareSearch method to create a ShunPreparedStatement object. Refer to Appendix A, Format of Search, Return and Sort Expressions for more information about search expressions and return expressions.

(3) Set the Sort Expression

Use the setSort method to set the sort expression.

Note

The total length of the sort keys specified in the sort expression determines the number of data items that can be returned. A maximum of 1,000 items can be returned. No more data can be returned even if a higher value is specified for the reply start number or the number of items to return per request. The getReturnableCount method can be used to extract the maximum number of data items that can be returned. Refer to Appendix D, Allowable Values, for a relationship between the number of data items that can be returned and the total length of sort keys.

(4) Execute the Search (Create a ShunResultSet Object)

Execute the search using the executeSearch method. A ShunResultSet object will be created to hold the results of the search.

(5) Extract the Results of the Search

Always invoke the next method before extracting the results of the search. The next method returns true if there is still more data that can be extracted and false otherwise.

Use one of the getXXX methods to extract the XML documents. The following table shows the methods that can be used.

Method	Function
getString	Extracts XML documents as String objects.
getStringArray	Extracts XML documents as a two-dimensional array of String objects.
getStream	Extracts XML documents as InputStream objects.

Notes

The getStringArray method can be used to extract the search results in text format.

The getRecordID method can be used to obtain record identifiers (that uniquely identify each data item) as well as the results of the search. Record identifiers are used to extract or delete the corresponding entire XML documents.

(6) Close the ShunResultSet Object and the ShunPreparedStatement Object

When they are no longer required, always close the objects using the close methods of the ShunResultSet object and the ShunPreparedStatement object.

Aggregating the Content of the Data that Matches Search Conditions

Sometimes it is useful to be able to aggregate the results of a search, using the values of particular elements. Use the setSort method to aggregate the content of the data. By specifying an aggregation function in the return expression used as a parameter of the prepareSearch method, the results of the search will be aggregated before they are returned. The aggregation process can be used to calculate totals, averages, maximums, minimums, and the number of items.

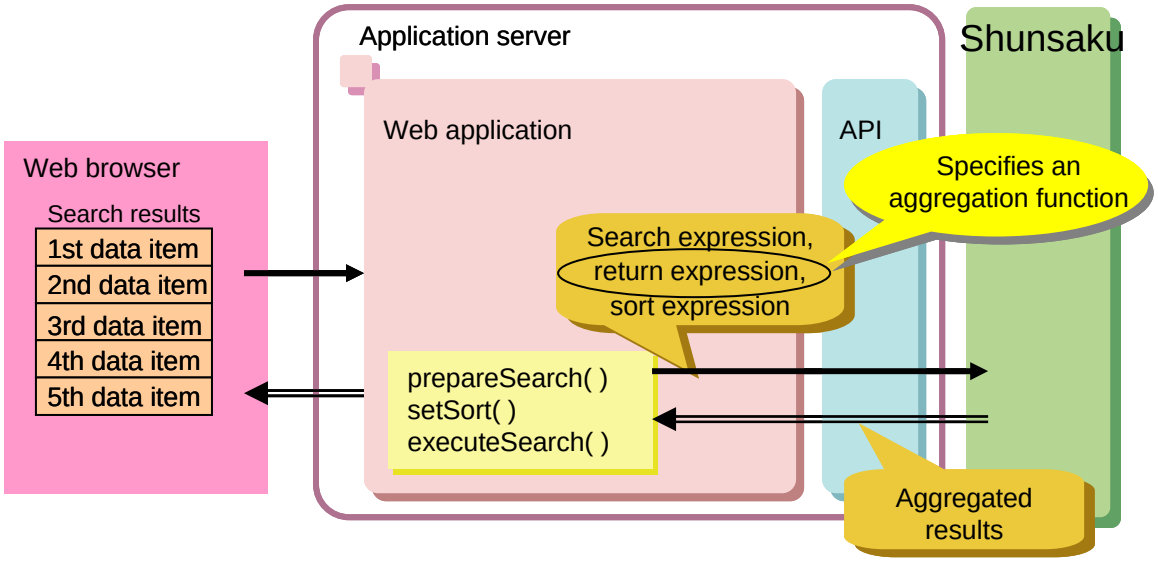


Figure 5-5 Aggregating the Content of the Data that Matches Search Conditions

Entry Example

```
ShunConnection con = new ShunConnection();

String query = "/document/base/prefecture == 'Osaka'";
String returnQuery = "max(/document/base/price/text())";                (1)
String sort = "/document/base/prefecture/text() ";                      (2)
ShunPreparedStatement pstmt = con prepareSearch(query, returnQuery);
                                                                        (3)
pstmt.setSort(sort);                                                    (4)
ShunResultSet rs = pstmt.executeSearch();                               (5)
 while(rs.next()) {                                                     (6)
  System.out.println("[Search results] = " + rs.getString());          (6)
}
rs.close();                                                             (7)
pstmt.close();                                                          (7)

con.close();
```

(1) Create a Return Expression

Create a return expression. To aggregate the search results, specify an aggregation function in the return expression. Refer to Return Expressions in Appendix A for more information on return expressions and specifying aggregation functions.

(2) Create a Sort Expression

Create a sort expression. Refer to Sort Expressions in Appendix A for more information on sort expressions.

(3) Create a ShunPreparedStatement Object

Specify a search expression and a return expression with the prepareSearch method () to create a ShunPreparedStatement object. Refer to Appendix A, Format of Search, Return and Sort Expressions for more information about search expressions and return expressions.

(4) Set the Sort Expression

Use the setSort method to set the sort expression.

Note

The total length of the sort keys specified in the sort expression determines the number of groups that can be returned. A maximum of 1,000 items can be returned. No more data can be returned even if a higher value is specified for the reply start number or the number of items to return per request. Use the getReturnableCount method to obtain the maximum number of data items that can be returned. Refer to Appendix D, Allowable Values, for a relationship between the number of data items that can be returned and the total length of sort keys.

(5) Execute the Search (Create a ShunResultSet Object)

Execute the search using the executeSearch method. A ShunResultSet object will be created to hold the results of the search.

(6) Extract the Results of the Search

Always invoke the next method before extracting the results of the search. The next method returns true if there is still more data that can be extracted and false otherwise.

Use one of the getXXX methods to extract the XML documents. The following table shows the methods that can be used.

Method	Function
getString	Extracts XML documents as String objects.
getStringArray	Extracts XML documents as a two-dimensional array of String objects.
getStream	Extracts XML documents as InputStream objects.

(7) Close the ShunResultSet Object and the ShunPreparedStatement Object

Use the close methods of the ShunResultSet object and the ShunPreparedStatement object to close these objects when they are no longer required.

Updating Data

The Java APIs can be used to add and delete data. Refer to Updating Data in Appendix B for sample programs used for updating data. This section describes how to create applications that perform data updates.

Adding Data

To add data, use the prepareInsert method.

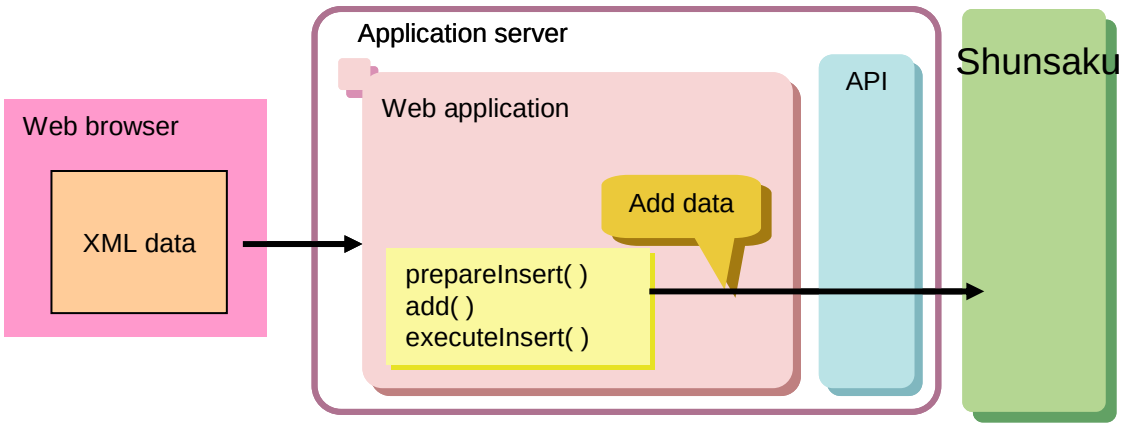


Figure 5-6 Adding Data

Entry Example

```
ShunConnection con = new ShunConnection();

ShunPreparedStatement pstmt = con prepareInsert();                    (1)
// Read data from file
FileInputStream oFIS = new FileInputStream("newData.xml");
pstmt.add(oFIS);                                                      (2)
pstmt.executeInsert();                                                (3)
pstmt.close();                                                        (4)

con.close();
```

(1) Create a ShunPreparedStatement Object

Use the prepareInsert method to create a ShunPreparedStatement object.

(2) Specify the Data to Add

The following table shows the methods that can be used to add data.

Method	Function
add (String data)	Adds String object data.
add (InputStream data)	Adds InputStream object data.

The data specified with the add method can include multiple data items. Also, data that is stored in multiple files can be added by invoking the add method multiple times. However, a single data item cannot be split into multiple segments and then specified in multiple instances of the add method.

(3) Add the Data

Use the executeInsert method to add the data.

(4) Close the ShunPreparedStatement Object

When the object is no longer required, always close it using the close method of the ShunPreparedStatement object.

Deleting Data

To delete data, use the prepareDeleteRecordID method.

The Java APIs use record identifiers to delete data. Before deleting data, use the getRecordID method to obtain the record identifiers of the data items to be deleted.

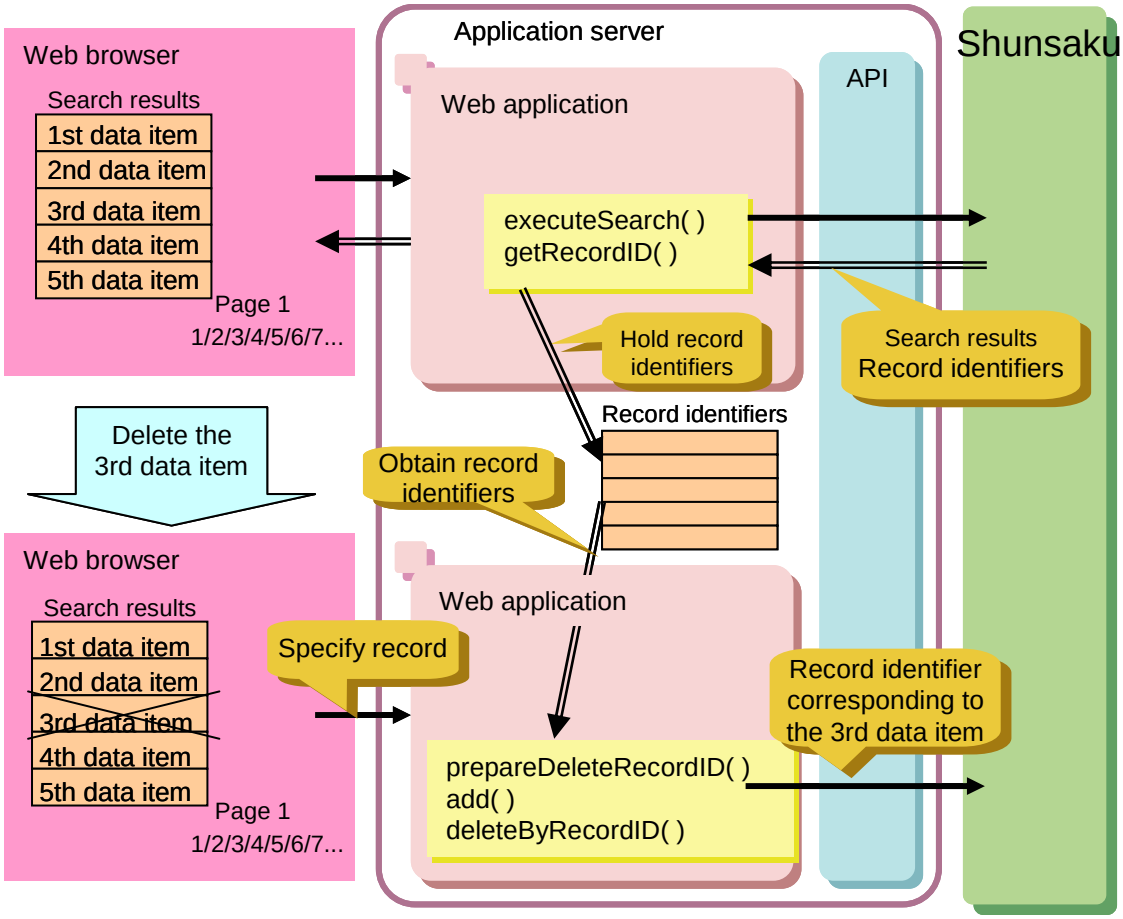


Figure 5-7 Deleting Data

Entry Example

```
ShunConnection con = new ShunConnection();

ShunPreparedRecordID prid = con prepareDeleteRecordID();              (1)
prid.add(recordID);                                                   (2)
prid.deleteByRecordID();                                              (3)
prid.close();                                                         (4)

con.close();
```

(1) Create a ShunPreparedRecordID object

Use the prepareDeleteRecordID method to create a ShunPreparedRecordID object.

(2) Specify the Record Identifiers of the Data Items to be Deleted

Use the add method to specify the record identifier. Record identifiers can be obtained using the getRecordID method.

Multiple record identifiers can be specified with the add method. Any existing record identifiers will be overwritten if they are specified again.

Note

By specifying multiple record identifiers with the add method, multiple XML documents can be deleted at once.

(3) Delete the Data

Execute the delete process using the deleteByRecordID method.

(4) Close the ShunPreparedRecordID Object

When the object is no longer required, always close it using the close method of the ShunPreparedRecordID object.

Closing Connections

After the data searches or data updates have finished, close the connection to Shunsaku. Use the close method to close the connection.

Entry Example

```
ShunConnection con = new ShunConnection();
   :
con.close();
```

Error Handling

If an error occurs with an application that uses the Java APIs, a ShunException is thrown. The methods of the ShunException class can be used to obtain the following information:

- Error code.
- Error level.
- Error message.

Entry Example

```
try {
    :
} catch (ShunException ex) {
     System.out.println("Error code:" + ex.getErrCode());              (1)
     System.out.println("Error level:" + ex.getErrLevel());            (2)
     System.out.println("Error message:" + ex.getMessage());           (3)
}
```

(1) Error Code

The error code can be obtained using the getErrCode method. Refer to Error Codes Output when Java APIs are Used for more information on error codes.

(2) Error level

The error level can be obtained using the getErrLevel method.

When errors occur, the connection to Shunsaku is sometimes closed forcibly. Users can check the error level to find out the state of the application where the error occurred.

There are two error levels, as shown in the following table.

Constant	Meaning
SHUN_ERROR	Indicates that a warning level error (such as a parameter error) has occurred. Retry from the point where the error occurred.
SHUN_ERROR_CONNECTION_TERMINATED	An error has occurred, causing the connection to Shunsaku to be forcibly closed. Close the ShunConnection object where the error occurred and then open a connection again.

(3) Error message

The error message can be obtained using the getMessage method. Refer to Error Codes Output when Java APIs are Used for more information on error messages.

Character Encoding Used by Java APIs

This section explains the character encoding used by the Java APIs.

Specify UNICODE as the character encoding to be used in conditional expressions, return expressions and sort expressions.

Search results are returned using the UNICODE encoding.

The following diagram illustrates an overview of character code conversion in the entire Shunsaku system.

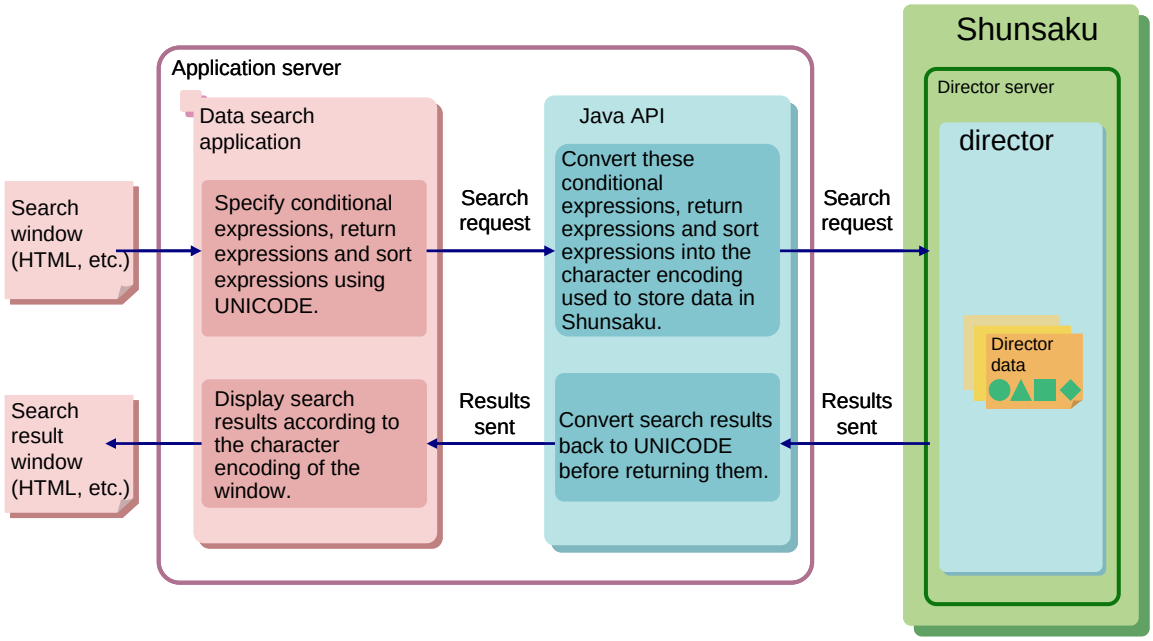


Figure 5-8 Overview of Character Code Conversion

Error Codes Output when Java APIs are Used

The following table shows the error codes output when Java APIs are used.

Table 5-3 Error Codes Output when Java APIs are Used

Category	Error code	Message
Parameter error	-300	shun: ERROR: -300: The size of parameter @1@ is incorrect.@2@
	-301	shun: ERROR: -301: Null was specified as a parameter.@1@,@2@
	-302	shun: ERROR: -302: A negative number was specified as a parameter.@1@,@2@
	-303	shun: ERROR: -303: A blank space was specified as a parameter.@1@,@2@
	-304	shun: ERROR: -304: Data was not specified for insertion.@1@
	-305	shun: ERROR: -305: The requested host could not be found.
	-306	shun: ERROR: -306: Either the director or the conductor has not been started, or the port number is incorrect.
	-307	shun: ERROR: -307: @1@ is not a supported encoding type.
	-308	shun: ERROR: -308: The properties file is null.
	-309	shun: ERROR: -309: An invalid value was specified in the properties file(@1@).
	-310	shun: ERROR: -310: There is an error in the XML data.@1@
	-311	shun: ERROR: -311: text() was specified in the return expression.@1@
	-312	shun: ERROR: -312: A number less than or equal to zero was specified as a parameter.@1@,@2@
	-313	shun: ERROR: -313: No RecordID has been specified.@1@
	-314	shun: ERROR: -314: An invalid value was specified as a parameter.@1@,@2@
Sequence error	-320	shun: ERROR: -320: ShunConnection is closed.@1@
	-321	shun: ERROR: -321: This ShunPreparedStatement is not a @1@ statement.@2@
	-322	shun: ERROR: -322: ShunResultSet is closed.@1@
	-323	shun: ERROR: -323: The cursor position is invalid.@1@
	-324	shun: ERROR: -324: ShunPreparedStatement is closed.@1@
	-325	shun: ERROR: -325: ShunPreparedRecordID is closed.@1@
	-326	shun: ERROR: -326: This ShunPreparedRecordID is not a @1@ statement.@2@
Environment error	-330	shun: ERROR: -330: An unexpected exception occurred:@1@.@2@
	-331	shun: ERROR: -331: A timeout error occurred.@1@
	-332	shun: ERROR: -332: A communication error occurred.
Syntax error, etc	-340	shun: ERROR: -340: An error occurred in the director or the conductor(errCode=@1@).@2@
Others	-345	shun: ERROR: -345: This method is not implemented.@1@
	-349	shun: ERROR: -349: A system error occurred.@1@,@2@

Detailed information about each error code and message is shown below.

Error Code = -300

Message

shun: ERROR: -300: The size of parameter @1@ is incorrect.@2@

Description

The size of parameter @1@ of method @2@ is invalid.

Parameters

@1@: Java API parameter name

@2@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that the size of the parameter is within acceptable limits and then rerun the application. Refer to Appendix D, Allowable Values, for more information on allowable values.

Error Code = -301

Message

shun: ERROR: -301: Null was specified as a parameter.@1@,@2@

Description

A null value was specified in parameter @2@ of method @1@.

Parameters

@1@: Java API method name

@2@: Java API parameter name

System Action

Stops processing this method.

User Response

Fix the application so that a correct value is set for the parameter and then rerun the application.

Error Code = -302

Message

shun: ERROR: -302: A negative number was specified as a parameter.@1@,@2@

Description

A negative number was specified in parameter @2@ of method @1@.

Parameters

@1@: Java API method name

@2@: Java API parameter name

System Action

Stops processing this method.

User Response

Fix the application so that a correct value is set for the parameter and then rerun the application.

Error Code = -303

Message

shun: ERROR: -303: A blank space was specified as a parameter.@1@,@2@

Description

A blank space was specified in parameter @2@ of method @1@.

Parameters

@1@: Java API method name

@2@: Java API parameter name

System Action

Stops processing this method.

User Response

Fix the application so that a correct value is set for the parameter and then rerun the application.

Error Code = -304

Message

shun: ERROR: -304: Data was not specified for insertion.@1@

Description

No additional data was specified with method @1@.

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that additional data is set correctly and then rerun the application.

Error Code = -305

Message

shun: ERROR: -305: The requested host could not be found.

Description

There is an error with the host name or IP address.

System Action

Stops processing this method.

User Response

Specify the correct host name or IP address with the ShunConnection method and then rerun the application. Refer to Opening Connections for more information on how to specify host names and IP addresses.

Error Code = -306

Message

shun: ERROR: -306: Either the director or the conductor has not been started, or the port number is incorrect.

Description

Either the director or the conductor has not been started, or the port number is incorrect.

System Action

Stops processing this method.

User Response

- If the port number is incorrect, take the following action:

  Specify the correct port number with the ShunConnection method and then rerun the program. Refer to Opening Connections for more information on how to specify port numbers.

- If the director or the conductor has not been started, take the following action:

  Start the director or conductor and then rerun the program. Refer to the User's Guide for more information on how to start a director or conductor.

- If the connection destination server has not been started, take the following action:

  Start the server and then rerun the program.

- In all other cases, take the following action.

  Wait for a few moments and then rerun the command.

Error Code = -307

Message

shun: ERROR: -307: @1@ is not a supported encoding type.

Description

@1@ is not a supported character encoding.

Parameters

@1@ The specified encoding

System Action

Stops processing this method.

User Response

Use a supported character encoding.

Error Code = -308

Message

shun: ERROR: -308: The properties file is null.

Description

The Properties class object specified with the ShunConnection method is null.

System Action

Stops processing this method.

User Response

Fix the application so that the Properties class object specified with the ShunConnection method can be created correctly and then rerun the application. Refer to Opening Connections for more information on how to create Properties class objects.

Error Code = -309

Message

shun: ERROR: -309: An invalid value was specified in the properties file(@1@).

Description

There is an error with a value (@1@) specified in the Java Properties file.

Parameters

@1@: parameter name

System Action

Stops processing this method.

User Response

Specify the correct information in the Java Properties file specified with the ShunConnection method and then rerun the program. Refer to Opening Connections for more information on how to set up Java Properties files.

Error Code = -310

Message

shun: ERROR: -310: There is an error in the XML data.@1@

Description

There is an error with the XML document @1@.

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Check the content of the XML document.

Error Code = -311

Message

shun: ERROR: -311: text() was specified in the return expression.@1@

Description

text() was specified in the return expression @1@.

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that a correct value is set in the return expression and then rerun the application. Refer to Return Expressions in Appendix A for more information on return expressions.

Error Code = -312

Message

shun: ERROR: -312: A number less than or equal to zero was specified as a parameter.@1@,@2@

Description

A value of zero or less was specified in parameter @2@ of method @1@.

Parameters

@1@: Java API method name

@2@: Java API parameter name

System Action

Stops processing this method.

User Response

Fix the application so that a correct value is set in the parameter and then rerun the application.

Error Code = -313

Message

shun: ERROR: -313: No RecordID has been specified.@1@

Description

No record identifier has been specified for method @1@.

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that it sets a correct record ID and then rerun the application.

Error Code = -314

Message

shun: ERROR: -314: An invalid value was specified as a parameter.@1@,@2@

Description

An invalid value was specified in parameter @2@ of method @1@.

Parameters

@1@: Java API method name

@2@: Java API parameter name

System Action

Stops processing this method.

User Response

Fix the application so that a correct value is set in the parameter and then rerun the application.

Error Code = -320

Message

shun: ERROR: -320: ShunConnection is closed.@1@

Description

The ShunConnection object has been closed. @1@

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that this method is not called until the ShunConnection object has been created, and then rerun the application.

Error Code = -321

Message

shun: ERROR: -321: This ShunPreparedStatement is not a @1@ statement.@2@

Description

This ShunPreparedStatement object is not a @1@ statement. @2@

Parameters

@1@: The output is as follows:

- Search
- Insert:
- Delete

@2@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that it creates a ShunPreparedStatement object that can execute the method, and then rerun the application.

Error Code = -322

Message

shun: ERROR: -322: ShunResultSet is closed.@1@

Description

The ShunResultSet object has been closed. @1@

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that this method is not called until the ShunResultSet object has been created, and then rerun the application.

Error Code = -323

Message

shun: ERROR: -323: The cursor position is invalid.@1@

Description

The cursor position is invalid. @1@

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that this method is called while the cursor is in a valid position, and then rerun the application.

Error Code = -324

Message

shun: ERROR: -324: ShunPreparedStatement is closed.@1@

Description

The ShunPreparedStatement object has been closed. @1@

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that this method is not called until the ShunPreparedStatement object has been created, and then rerun the application.

Error Code = -325

Message

shun: ERROR: -325: ShunPreparedRecordID is closed.@1@

Description

The ShunPreparedRecordID object has been closed. @1@

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that this method is not called until the ShunPreparedRecordID object has been created, and then rerun the application.

Error Code = -326

Message

shun: ERROR: -326: This ShunPreparedRecordID is not a @1@ statement.@2@

Description

This ShunPreparedRecordID object is not a @1@ statement. @2@

Parameters

@1@: The output is as follows:

- Search
- Insert
- Delete

@2@: Java API method name

System Action

Stops processing this method.

User Response

Fix the application so that it creates a ShunPreparedRecordID object that can execute the method, and then rerun the application.

Error Code = -330

Message

shun: ERROR: -330: An unexpected exception occurred:@1@.@2@

Description

A @1@ exception has occurred with method @2@.

Parameters

@1@: Exception type

@2@: Java API method name

System Action

Stops processing this method.

User Response

Take action appropriate to the type of exception that has occurred.

Error Code = -331

Message

shun: ERROR: -331: A timeout error occurred.@1@

Description

A timeout error occurred with method @1@.

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

Review the conditions in the search expression or the content of the additional data.

Error Code = -332

Message

shun: ERROR: -332: A communication error occurred.

Description

A communication error has occurred.

System Action

Stops processing this method.

User Response

Review the communication environment.

When updating, sorting, or aggregating data, make sure that the connection destination is a conductor.

Error Code = -340

Message

shun: ERROR: -340: An error occurred in the director or the conductor(errCode=@1@).@2@

Description

An error (errCode = @1@) has been output by the director or the conductor. @2@

Parameters

@1@: The error code output by the director or the conductor

@2@: Java API method name

System Action

Stops processing this method.

User Response

Take the appropriate action as described in Error Codes Output by the Conductor or the Director. If a Shunsaku system message has been output on the connection destination host, take the appropriate action by referring to Messages that are Output to the System Log in the User's Guide.

Error Code = - 345

Message

shun: ERROR: -345: This method is not implemented.@1@

Description

Method @1@ is not implemented.

Parameters

@1@: Java API method name

System Action

Stops processing this method.

User Response

In the application, delete this method or replace with another method.

Error Code = - 349

Message

shun: ERROR: -349: A system error occurred.@1@,@2@

Description

A system error has occurred. @1@, @2@

Parameters

@1@: Java API method name

@2@: System error information

System Action

Stops processing this method.

User Response

Note the information in this message, and contact Fujitsu systems engineer.

Error Codes Output by the Conductor or the Director

The following table shows the error codes output by the conductor or the director when the error code is -340.

Table 5-4 Error Codes Output by the Conductor or the Director

Error code	Meaning	User response
-1	Cannot connect to host.	Check the host name or IP address and the port number for the connection destination. If they are correct, wait a few moments and then rerun the command.
-10	The service is not ready to receive a request.	Wait a few moments and then rerun the command. When updating, sorting or aggregating data, make sure that the connection destination is a conductor. Windows If an error occurs when the command is rerun, determine the cause of the error using the event log, eliminate the error cause, and then rerun the command. Linux If an error occurs when the command is rerun, determine the cause of the error using the system log (syslog), eliminate the error cause, and then rerun the command.
-11	The maximum number of simultaneous requests has been exceeded.	Wait a few moments and then rerun the command.
-12	A timeout occurred while the system was waiting for a response.	Wait a few moments and then rerun the command.
-13	The maximum number of items that can be updated, sorted or aggregated has been exceeded.	Wait a few moments and then rerun the command.
-20	An error was found in the specification of a passed parameter.	Set the correct parameter and rerun the command.
-21	A search expression syntax error occurred.	Check the input parameter search expression and rerun the command.
-22	A return expression syntax error occurred.	Check the input parameter return expression and rerun the command.
-23	A search expression size error occurred.	Check the length of the input parameter search expression and then rerun the command. (The size of the search expression must be from 1 to 65, 535 bytes.)
-24	A return expression size error occurred.	Check the length of the input parameter return expression and then rerun the command. (The size of the return expression must be from 0 to 65, 535 bytes.)
-25	A sort expression syntax error occurred..	Check the sort expression input parameter and rerun the command.
-26	A sort expression size error occurred.	Check the length of the input parameter sort expression and rerun the command. (The size of the sort expression must be from 1 to 65, 535 bytes.)

Error code	Meaning	User response
-27	There is an inconsistency between the content specified in the sort expression and the return expression.	Check the input parameter sort expression and return expression and then rerun the command.
-30	The reply data storage area was insufficient. (*1)	Increase the size of the reply data storage area and perform the search again.
-201	There is a problem with the Shunsaku environment.	Reinstall Shunsaku.
-204	The addition processing has failed.	Check the content of the XML data and rerun the command. Check the status of the director server and rerun the command.
-207	The deletion processing has failed.	Check the content of the conductor control information and the record identifier(s) and rerun the command. Check the status of the director server and rerun the command.
-208	The process that executes either the addition or deletion processing failed to start.	Check the status of the director server and rerun the command.
-209	The addition process or the deletion process was interrupted.	Check the status of the director server and rerun the command.
-210	Failed to open a work file.	Check the value of the WorkFolder parameter in the director environment file and rerun the command.
-211	Failed to write to a work file.	Check the status of the disk specified in the WorkFolder parameter in the director environment file and rerun the command.

*1: If error -30 occurs, reply data is not saved.

Chapter 6

C Application Development

This chapter explains how to develop applications that use the C APIs provided by Shunsaku.

- C API Overview
- How to Use C APIs
- Error Codes Output when C APIs are Used

C API Overview

The C APIs are interfaces used to manipulate Shunsaku data from applications written in C.

The following table lists the C APIs provided by Shunsaku.

Table 6-1 API List

Function category	Function name	Description
Lookup operations	shunsearch1	Finds the number of XML documents that match the search conditions
	shunsearch2	Obtains the XML documents that match the search conditions in a specified format
	shunsearch3	Obtains all of a particular XML document.
	shunsort	Finds data items that match the search conditions and obtains the data items after they are sorted. Also finds data items that match the search conditions and obtains the data items after their values are aggregated.
Update operations	shunadd	Adds data.
	shundeletebyrecid	Deletes data.

Refer to the C API Reference for more information on the C APIs provided by Shunsaku.

How to Use C APIs

This section explains how to use the C APIs.

Searching Data

The following operations can be performed using the data lookup functions provided by the C APIs:

- Finding the number of XML documents that match the search conditions
- Obtaining the XML documents that match the search conditions in a specified format
- Obtaining all of a particular XML document
- Finding XML documents that match the search conditions and obtaining the documents after they are sorted
- Finding XML documents that match the search conditions and obtaining the documents after their contents are aggregated

These operations can be combined to create a wide range of applications. Refer to Searching Data in Appendix C for sample programs used for searching data. The rest of this section will describe how to create applications that use this data to search for functions.

Obtaining Search Results According to the Number of Data Items

Web-based search applications typically display only a few dozen search results per page, rather than displaying all of the search results in a single window. In such cases, the number of data items to be obtained can be controlled by specifying the reply start number and the number of items to return per request as arguments of the shunsearch2 function.

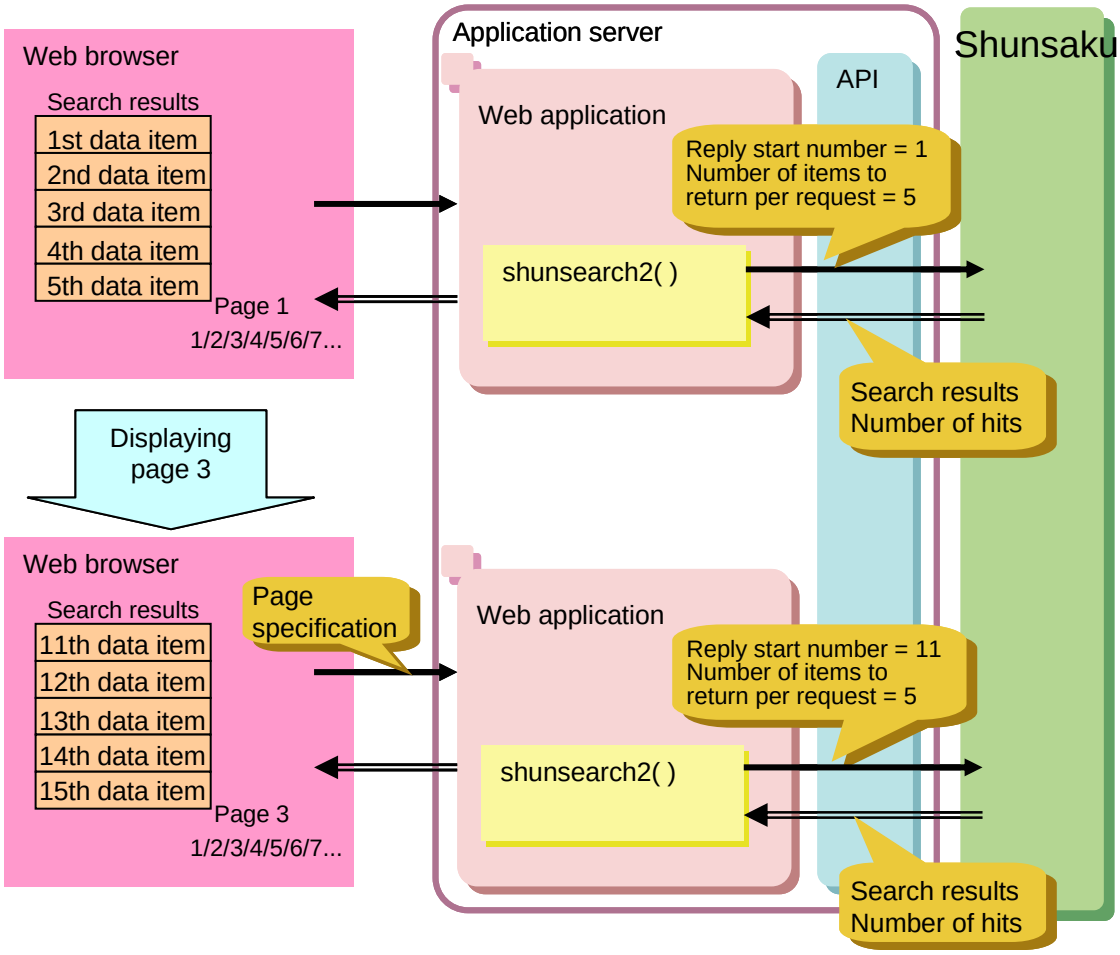


Figure 6-1 Obtaining Search Results According to the Number of Data Items

Notes

- The shunsearch2 function returns record identifiers (conductor control information and reply record identifiers) that uniquely identify the XML data along with the search results. Record identifiers are used to extract or delete corresponding entire XML documents.
- The shunsearch2 function returns the number of hits (XML documents that match the search conditions). This value can be used to find such things as the number of pages of the search results.
- For the number of items to return per request, specify the number of data items to display on each page.

Obtaining Search Results while Adding Search Conditions

When a search produces a large number of hits, it is sometimes useful to be able to narrow down the scope of search by adding more search conditions.

In such cases, perform a search process with the shunsearch2 function again by adding extra search conditions to the search expression specified with the shunsearch2 function argument and creating a new search expression. By repeating this operation, the user can narrow down the search results while referring to the search results displayed on the screen.

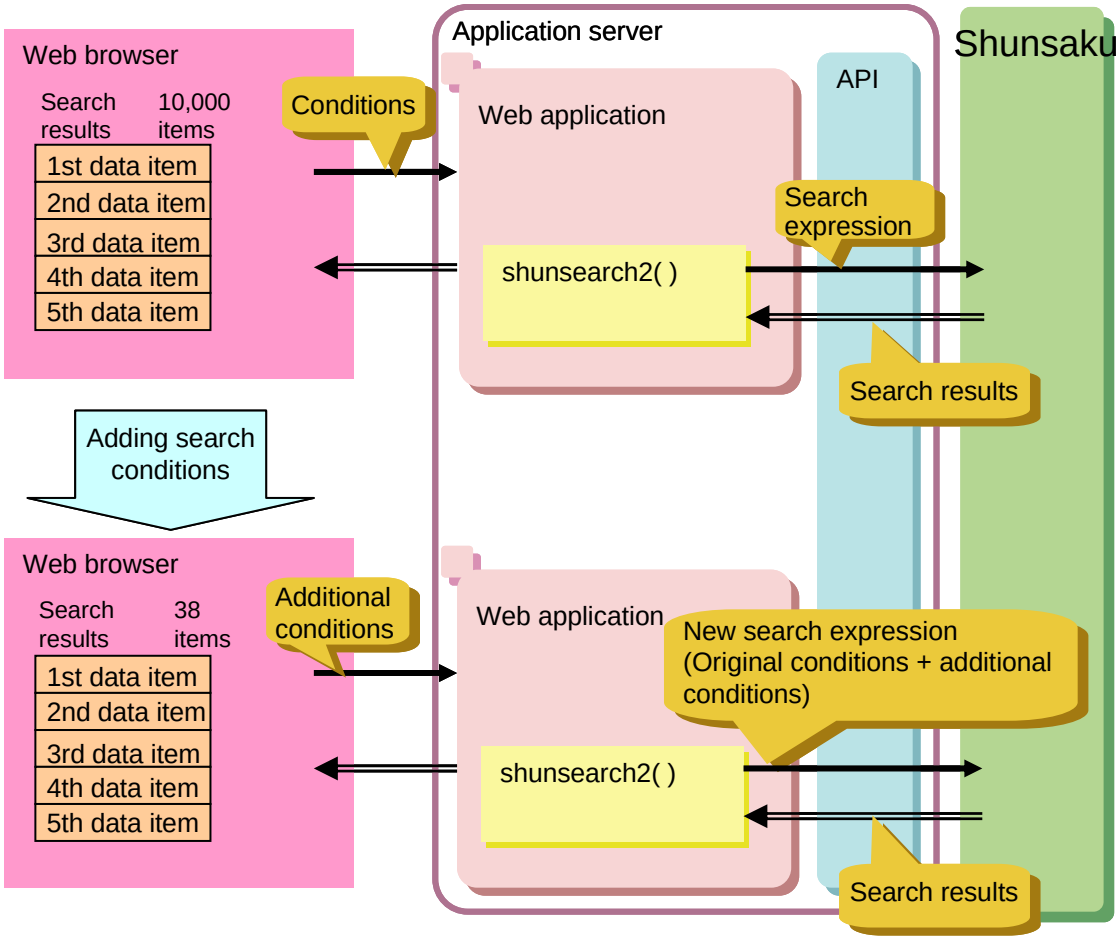


Figure 6-2 Obtaining Search Results while Adding Search Conditions

Notes

The shunsearch2 function returns record identifiers (conductor control information and reply record identifiers) that uniquely identify the XML documents along with the search results.

Record identifiers are used to extract or delete corresponding entire XML documents.

Obtaining Entire XML Documents

When searching for a particular XML document, do not obtain the entire document straightaway. Instead, start by obtaining the partial information that can effectively identify the document. The user can then use this partial information to pick out the desired XML document and obtain detailed information. To extract an entire XML document, use the record identifier (conductor control information and reply record identifiers) that is returned when the partial information is extracted. The entire target XML document can be extracted by specifying this record identifier with the shunsearch3 function argument.

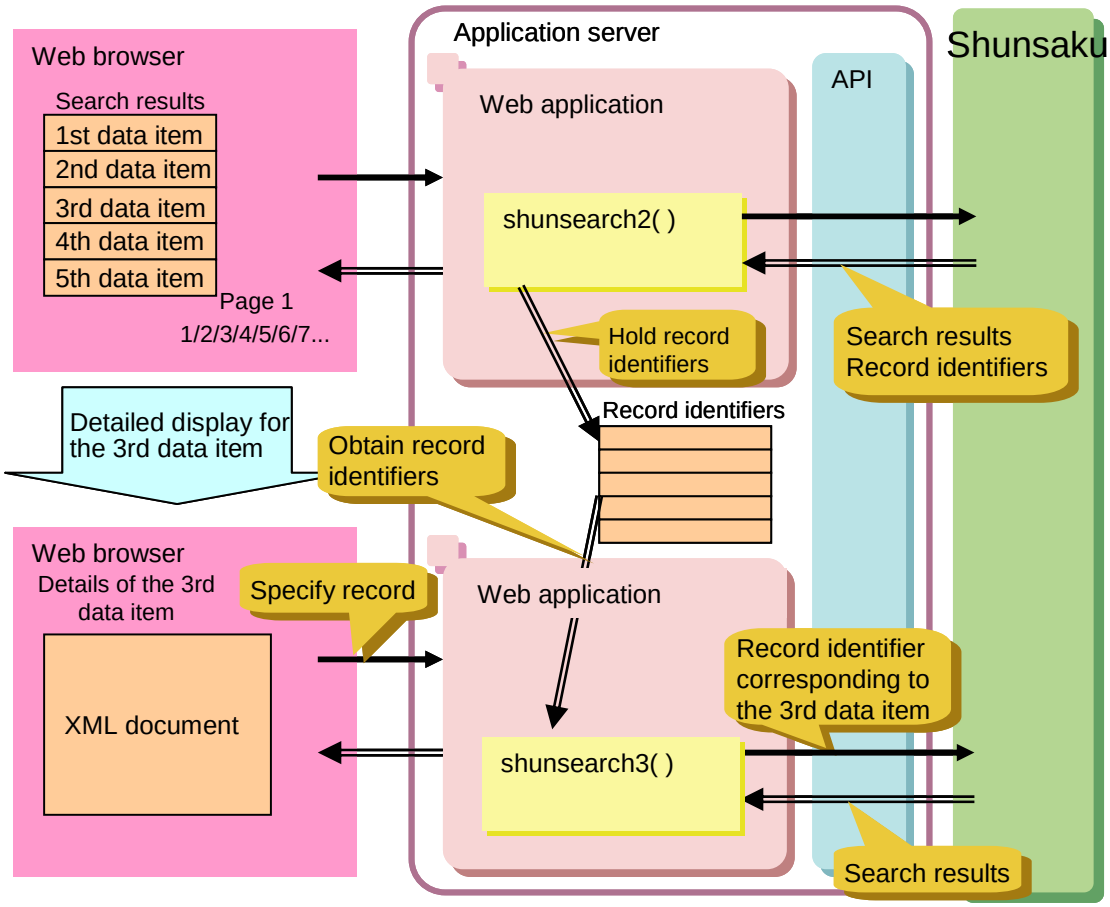


Figure 6-3 Obtaining Entire XML Documents

Note

By specifying multiple record identifiers with the shunsearch3 function, multiple XML documents can be obtained at once.

Obtaining Sorted Data

Sometimes it is useful to be able to sort the results of a search, using a particular element as a sort key.

To obtain sorted partial information about the data, use the shunsort function.

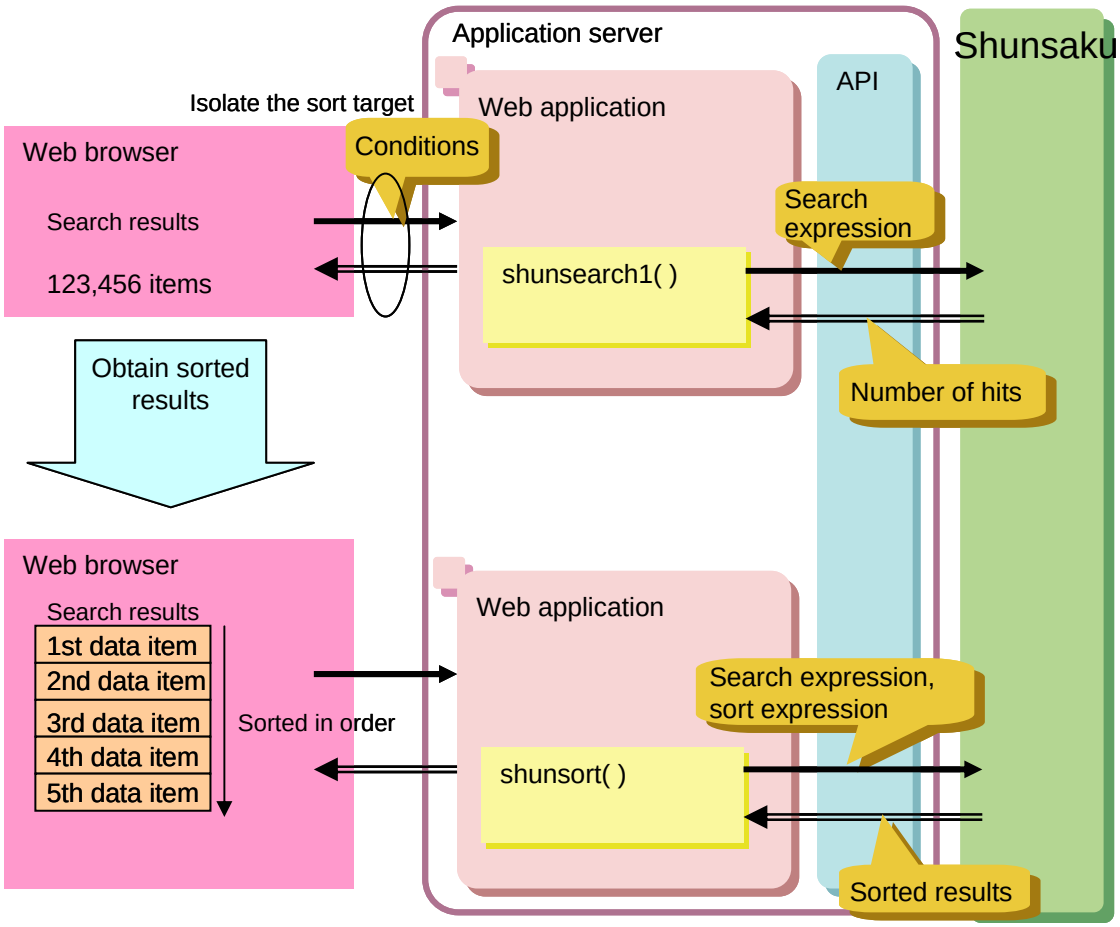


Figure 6-4 Obtaining Sorted Data

Notes

- In case of the shunsort function, the sort process looks up all of the XML documents that match the search conditions. To improve response performance, it is important to narrow down the data to be sorted by specifying a search expression that produces an appropriate number of hits. To find out how many XML documents match the search conditions before starting a sort process, use the shunsearch1 function.
- The shunsort function returns record identifiers (conductor control information and reply record identifiers) that uniquely identify the XML documents along with the search results.. Record identifiers are used to extract or delete corresponding entire XML documents.

Note

In case of the shunsort function, the total length of the sort keys specified in the sort expression determines the number of data items that can be returned. A maximum of 1,000 items can be returned. No more data can be returned even if a higher value is specified for the reply start number or the number of items to return per request. Refer to Appendix D, Allowable Values, for a relationship between the number of data items that can be returned and the total length of sort keys.

Aggregating the Content of the Data that Matches Search Conditions

Sometimes it is useful to be able to aggregate the results of a search, using the values of particular elements. Use the shunsort function to aggregate the content of the data. By specifying an aggregation function in the return expression used as an argument of the shunsort function, the results of the search will be aggregated before they are returned. The aggregation process can be used to calculate totals, averages, maximums, minimums, and the number of items.

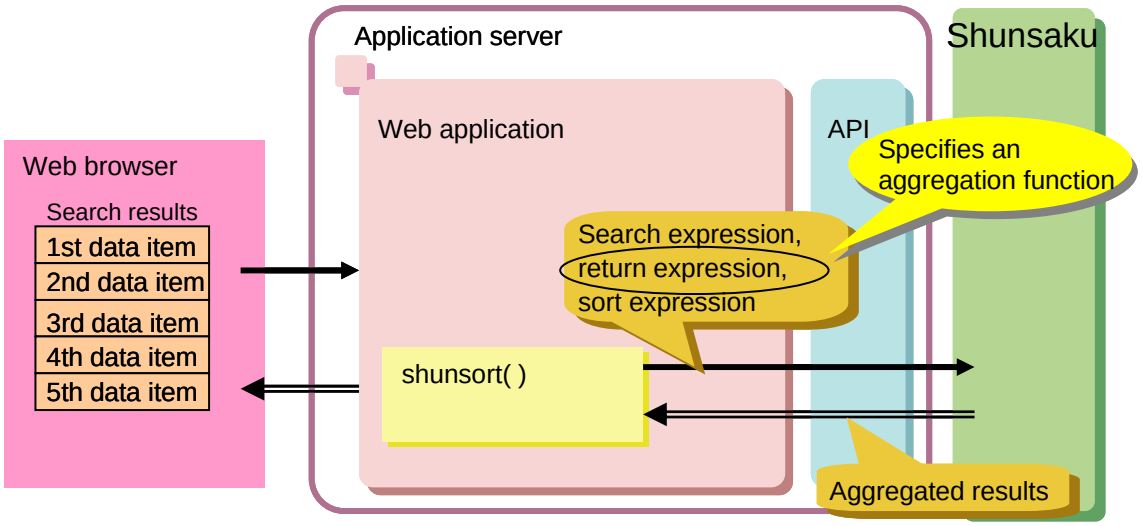


Figure 6-5 Aggregating the Content of the Data that Matches Search Conditions

Note

In case of the shunsort function, the total length of the sort keys specified in the sort expression determines the number of groups that can be returned. A maximum of 1,000 items can be returned. No more data can be returned even if a higher value is specified for the reply start number or the number of items to return per request. Refer to Appendix D, Allowable Values, for a relationship between the number of groups that can be returned and the total length of sort keys.

Updating Data

The C APIs can be used to add and delete data. Refer to Updating Data in Appendix C for sample programs used for updating data. This section describes how to create applications that perform data updates.

Adding Data

To add data, use the shunadd function.

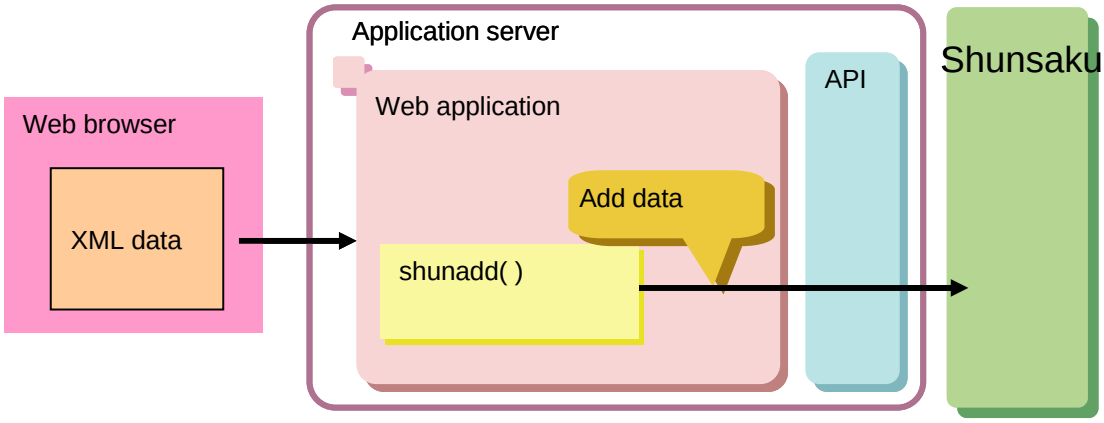


Figure 6-6 Adding Data

Note

The shunadd function can simultaneously add multiple XML documents using either of the following ways:

- By sequentially listing multiple XML documents in a single data addition area
- By specifying multiple data addition areas

Deleting Data

To delete data, use the shundeletebyrecid function.

The shundeletebyrecid function uses record identifiers (conductor control information and reply record identifiers) to delete data. Before executing the shundeletebyrecid function, use the shunsearch2 function or the shunsort function to obtain the record identifiers of the data to be deleted.

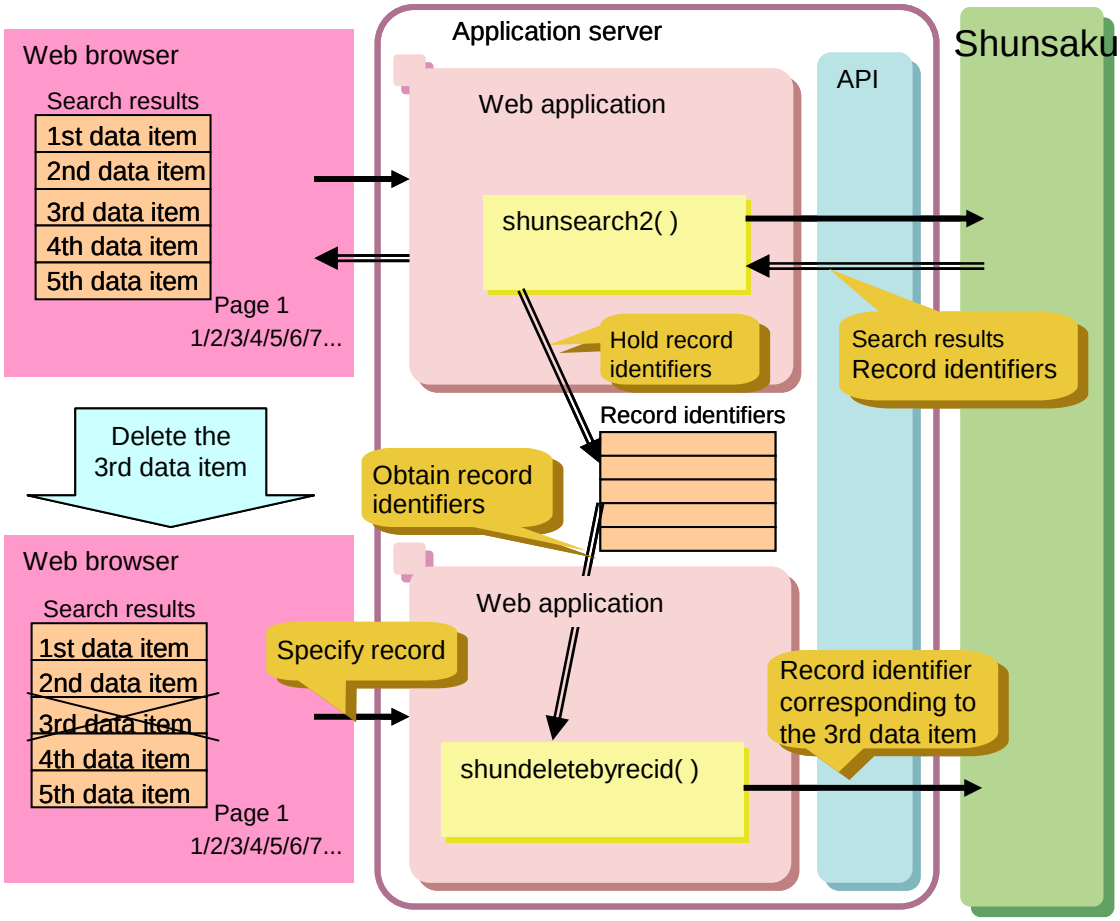


Figure 6-7 Deleting Data

Note

By specifying multiple record identifiers with the shundeletebyrecid function, multiple XML documents can be deleted at once.

Character Encoding Used by the C APIs

For the C API search expressions, return expressions and sort expressions, use the same character encoding as is used to store the XML documents in Shunsaku.

Error Codes Output when C APIs are Used

The following table shows the error codes output when C APIs are used.

Table 6-2 Error Codes Output when C APIs are Used

Error code	Meaning	User response
-1	Cannot connect to host.	Check the host name or IP address and the port number for the connection destination. If they are correct, wait a few moments and then rerun the command.
-10	The service is not ready to receive a request.	Wait a few moments and then rerun the command. When updating, sorting or aggregating data, make sure that the connection destination is a conductor. Windows If an error occurs when the command is rerun, determine the cause of the error using the event log, eliminate the error cause, and then rerun the command. Linux If an error occurs when the command is rerun, determine the cause of the error using the system log (syslog), eliminate the error cause, and then rerun the command.
-11	The maximum number of simultaneous requests has been exceeded.	Wait a few moments and then rerun the command.
-12	A timeout occurred while the system was waiting for a response.	Wait a few moments and then rerun the command.
-13	The maximum number of items that can be updated, sorted or aggregated has been exceeded.	Wait a few moments and then rerun the command.
-20	An error was found in the specification of a passed parameter.	Set the correct parameter and rerun the command.
-21	A search expression syntax error occurred.	Check the input parameter search expression and rerun the command.
-22	A return expression syntax error occurred.	Check the input parameter return expression and rerun the command.
-23	A search expression size error occurred.	Check the length of the input parameter search expression and then rerun the command. (The size of the search expression must be from 1 to 65, 535 bytes.)
-24	A return expression size error occurred.	Check the length of the input parameter return expression and then rerun the command. (The size of the return expression must be from 0 to 65, 535 bytes.)
-25	A sort expression syntax error occurred.	Check the sort expression input parameter and rerun the command.
-26	A sort expression size error occurred.	Check the length of the input parameter sort expression and rerun the command. (The size of the sort expression must be from 1 to 65, 535 bytes.)

Error code	Meaning	User response
-27	There is an inconsistency between the content specified in the sort expression and the return expression.	Check the input parameter sort expression and return expression and then rerun the command.
-30	The reply data storage area was insufficient. (*1)	Increase the size of the reply data storage area and perform the search again.
-201	There is a problem with the Shunsaku environment.	Reinstall Shunsaku.
-204	The addition processing has failed.	Check the content of the XML data and rerun the command. Check the status of the director server and rerun the command.
-207	The deletion processing has failed.	Check the content of the conductor control information and the record identifier(s) and rerun the command. Check the status of the director server and rerun the command.
-208	The process that executes either the addition or deletion processing failed to start.	Check the status of the director server and rerun the command.
-209	The addition process or the deletion process was interrupted.	Check the status of the director server and rerun the command.
-210	Failed to open a work file.	Check the value of the WorkFolder parameter in the director environment file and rerun the command.
-211	Failed to write to a work file.	Check the status of the disk specified in the WorkFolder parameter in the director environment file and rerun the command.

*1: If error -30 occurs, reply data is not saved.

Appendix A

Format of Search, Return and Sort Expressions

This appendix explains the format of search expressions, return expressions and sort expressions specified as arguments to Shunsaku API functions.

- Common Format
- Search Expressions
- Return Expressions
- Sort Expressions

Common Format

This section details formatting common to the three types of expression (search expressions, return expressions and sort expressions).

Path Expressions

The XML document structure is represented as a tree.

Path expressions are used to describe the position of nodes within the XML tree structure.

The format of path expressions is shown below.

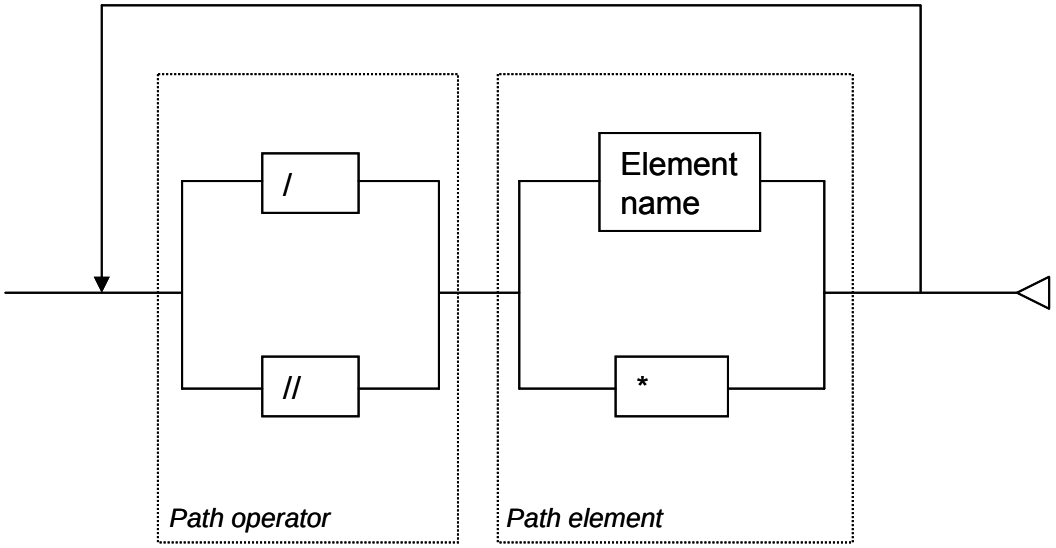


Figure A-1 Path Expressions

Path Element

Path elements are used to identify individual element nodes in an XML document.

Table A-1 Path Elements

Path element	Description
Element name	Indicates the name of an element node.
*	Indicates all of the element nodes below the upper node.

Path Operator

Path operators express the relationship between path elements. Path operators are described in the following table.

Table A-2 Path Operators

Path operator	Description
/	The target will be the node below the upper node.
//	The target will be all descendent nodes below the upper node.

Note

Symbols '//' and '*' cannot be entered consecutively in path expressions.

Example

An example path expression is shown below.

Document

```
<company>
  <name>fujitsu</name>
  <employee>
    <name>smith</name>
    <id>2000</id>
  </employee>
</company>
```

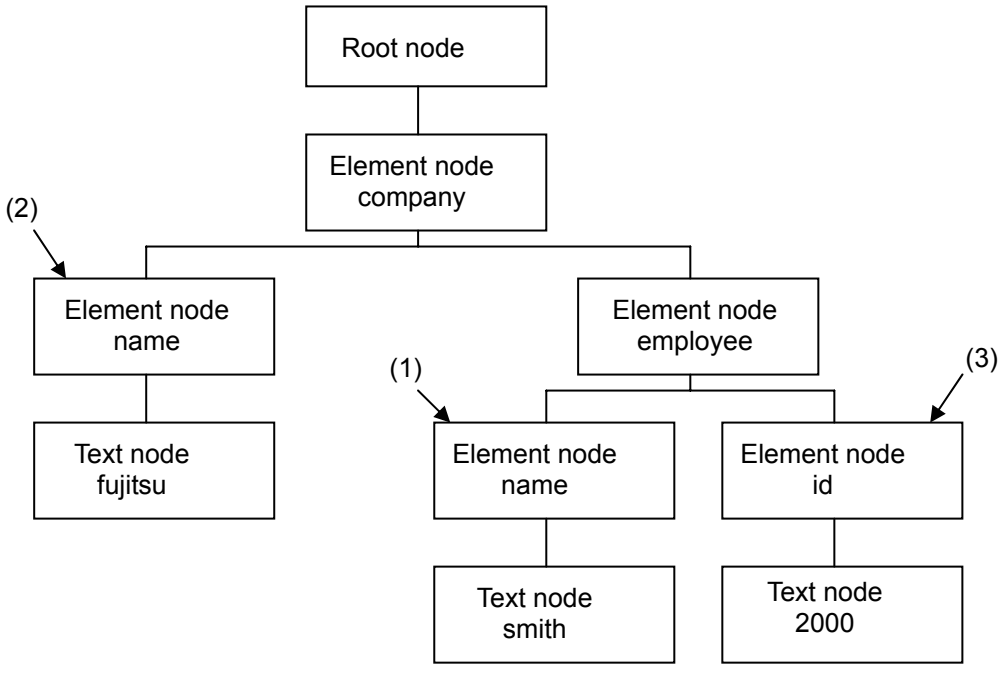


Figure A-2 Tree Representation of Data

```
/company/employee/name
```

This path expression indicates the 'name' element node below the 'employee' element node below the 'company' element node below the root node. This node is indicated by (1) in the figure above.

```
//name
```

This path expression indicates all 'name' element nodes below the root node. These nodes are indicated by (1) and (2) in the figure above.

```
/company/*/id
```

This path expression indicates an 'id' element node below any element node (‘name’ or ‘employee’ in the example of the figure above) below the 'company' element node below the root node. This node is indicated by (3) in the figure above.

Text Expressions

A text expression specifies the (string) value of the text node below element nodes of an XML document specified using path expressions.

The definition format for text expressions is shown below.

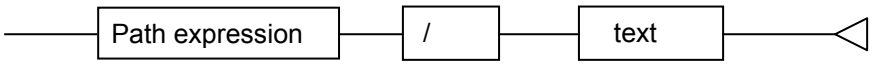


Figure A-3 Text Expressions

Path Expressions

The path expression is used to identify the positions of particular nodes within the XML tree structure.

Refer to Path Expressions for more information on path expressions.

Note

- When path expressions are specified as part of text expressions, the final element of the path expression cannot be '*'.

text()

This function specifies the value of the text node below the element nodes specified by the path expression. Always specify text() in text expressions.

Example

An example text expression is shown below.

Document

```
<company>
    <name>fujitsu</name>
    <employee>
      <name>smith</name>
      <id>2000</id>
    </employee>
</company>
```

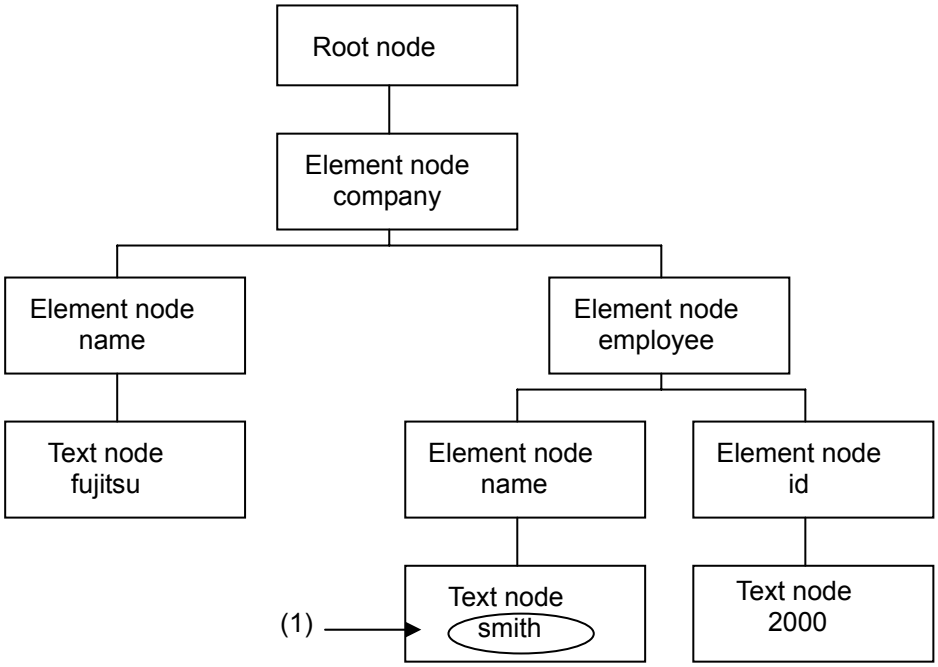


Figure A-4 Tree Representation of Data

```
/company/employee/name/text()
```

This text expression indicates the value of the text node below the 'name' element node, below the 'employee' element node, below the 'company' element node, below the root node. This value is ‘smith’ indicated by (1) in the above figure.

Single-Line Function Specification

If single-line functions are specified, the value specified by the text expression is converted by the function and the results are returned.

The definition format for single-line function specification is shown below.

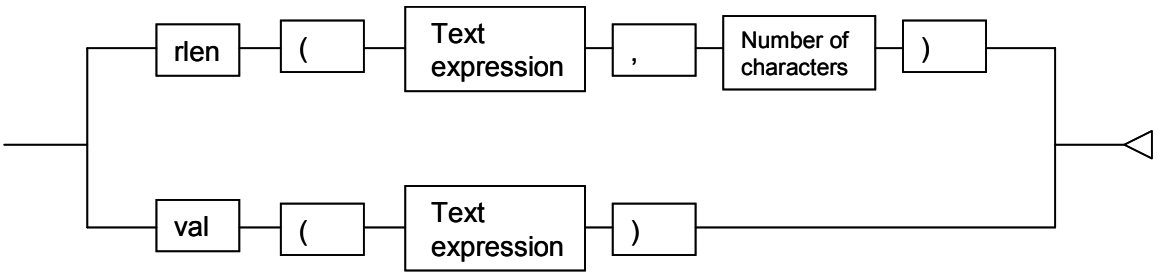


Figure A-5 Single-line Functions

The *rlen* Function

The *rlen* function returns the specified number of characters starting with the first character of the string expressed by the text expression.

Notes

- A number of characters between 1 and 2147483647 inclusive can be specified.
- The string expressed by the text expression will be returned unchanged if it is shorter than the number of characters specified.
- The '//' path operator cannot be specified in the text expression.
- The '*' path element cannot be specified in the text expression.

Example

```
rlen(/company/name/text(),16)
```

The *val* Function

The *val* function extracts only numeric values from the string expressed by the text expression.

Note

- The first numeric value that matches the format below is extracted from the string specified by the text expression.

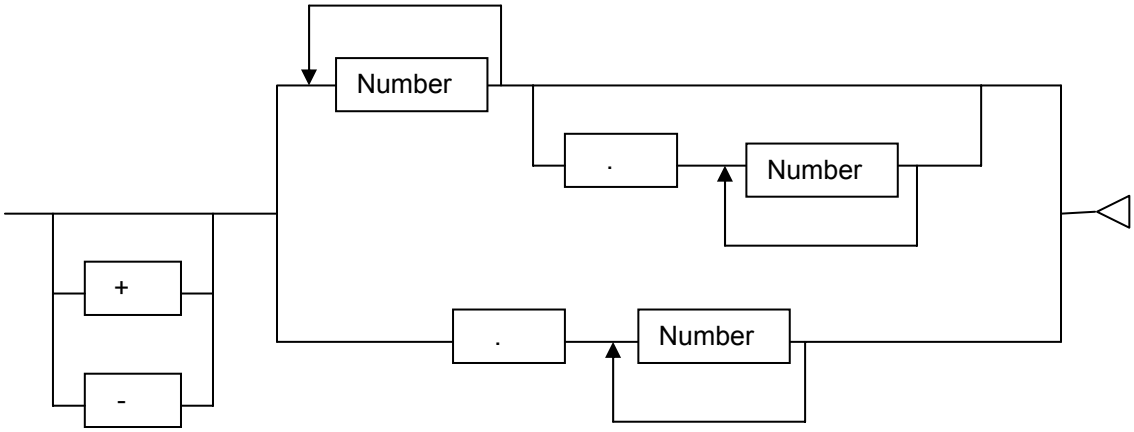


Figure A-6 *val* Function

Notes

- Commas in the integer part are ignored. If a decimal point is specified, all digits immediately following the decimal point up to the first non-numeric character are used as the fractional part.
- Strings specified by the text expression that do not contain numbers are handled as 0.
- An error will occur if the integer part except for leading zeros has more than 18 digits.
- The first 18 digits of the decimal part are significant. Any subsequent digits are truncated.
- The '//' path operator cannot be specified in the text expression.
- The '*' path element cannot be specified in the text expression.

Example

```
val(/company/employee/id/text())
```

Search Expressions

Search expressions are used to specify search conditions that that apply to XML documents to be retrieved

A search expression consists of one or more conditional expressions and filter expressions. Refer to Conditional Expressions and Filter Expressions later in this section for details on conditional expressions and filter expressions.

Use logical operators to specify more than one conditional expression or filter expression. Refer to Logical Operators later in this section for details.

The format for search expressions is shown below.

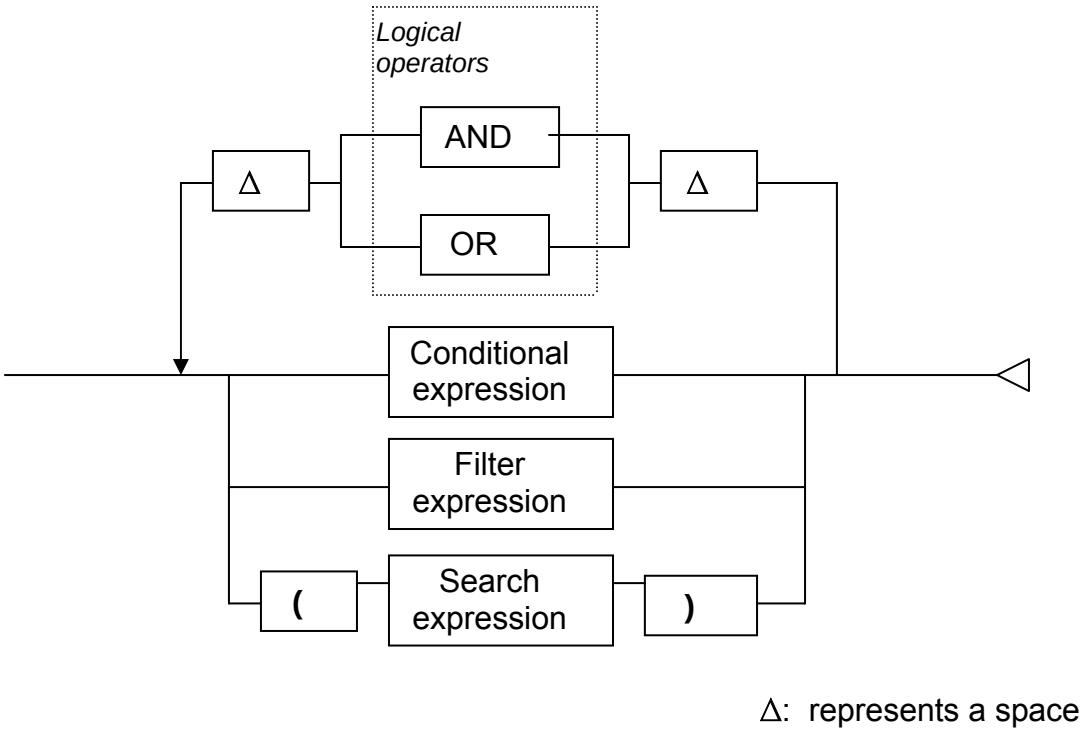


Figure A-7 Search Expressions

Note

- Search expressions must have at least one conditional expression or filter expression.

Logical Operators

Logical operators indicate the logical relationship between two adjacent conditional expressions and filter expressions when multiple expressions are specified.

Logical operators are briefly explained in the following table.

Table A-3 Logical Operators

Logical operator	Type of logical operation	Description
AND	AND operation	Combines two adjacent conditional expressions by logical operator AND. Evaluates to TRUE only if both conditional expressions are TRUE. Evaluates to FALSE if either or both conditional expressions are FALSE.
OR	OR operation	Combines two adjacent conditional expressions by logical operator OR. Evaluates to TRUE if either or both conditional expressions are TRUE. Evaluates to FALSE only if both conditional expressions are FALSE.

Notes

- The AND operator is evaluated first in search expressions that contain both AND and OR operators.
- Use parentheses '()' to change the order in which the logical operators are evaluated.

In the following example, ([conditional expression 2] OR [conditional expression 3]) is evaluated first.

```
[conditional expression 1] ΔANDΔ ([conditional expression 2] Δ OR Δ
[conditional expression 3])…
```

Δ: represents a space

Conditional Expressions

Conditional expressions are used to make comparisons between keywords and the value of element nodes (expressed by path expressions) in XML documents.

The definition format for conditional expressions is shown below.

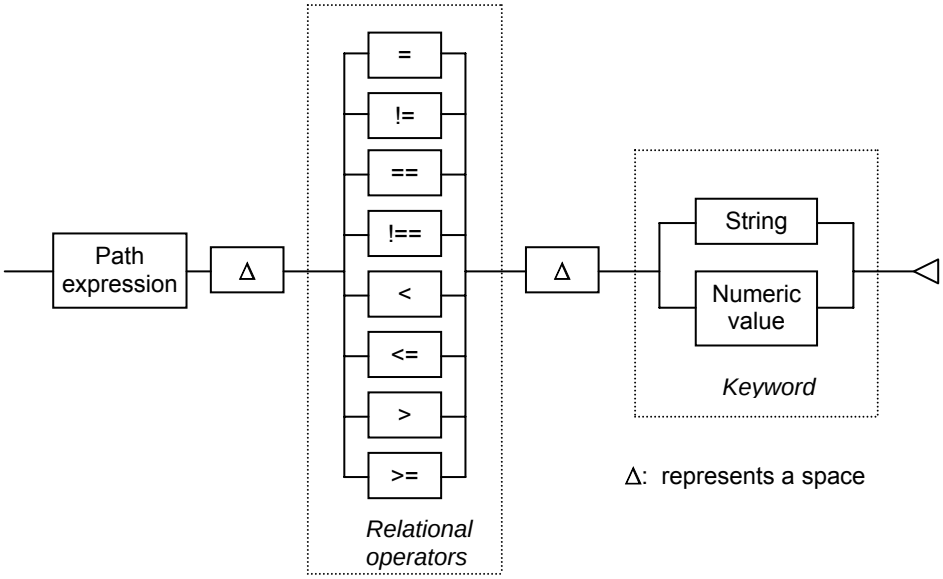


Figure A-8 Conditional Expressions

Conditional expressions consist of a path expression, a relational operator and a keyword.

Path Expressions

The path expression is used to identify the position of a node in the XML tree structure.

Specify an element node in the XML document to be compared.

Refer to Path Expressions for more information on path expressions.

Notes

- When performing partial match string searches, the user can specify the '//' path operator at the end of the path expression. This represents all element nodes below the element node specified by the path expression.
- When performing partial match string searches, the user can specify the '*' path element at the end of the path expression.

Keywords

The keyword is the string or numeric value that is compared to the value of the element node in the XML document that is specified with the path expression.

Character String

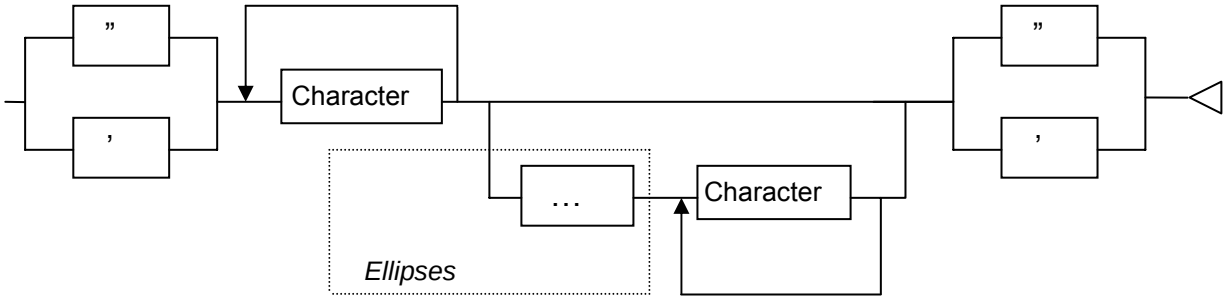


Figure A-9 Character String

If a string is specified for the keyword, the search will be a string search.

Enclose strings in either double- or single-quotes. Double- and single-quotes cannot be used together.

Refer to Character String Searches for more information on string searches.

Ellipses

Ellipses ‘...’ are specified in order to search for strings that contain an arbitrary number of unknown characters.

Escape Characters

The characters in the following table must be preceded by the escape character '\' when they are specified in strings.

For example, if 'abc\\' is specified, the search target string will be 'abc\'. The following characters must be preceded by '\'.

Table A-4 Escape Sequence

Character	Specification method
.	\.
\	\\
"	\"
'	\'

Entity References

Characters such as '<' and '>' have special meaning in XML documents. These characters can be expressed in XML documents using entity references.

Notes

- If a symbol (such as '<') is specified in the keyword of a partial match string search, the search will look in the XML document for both the symbol ('<') and entity references to the symbol ('<').
- If a symbol (such as '<') is specified in the keyword of a complete match string search, the search will only look for the symbol ('<') and not for entity references.
- If an entity reference (such as '<') is specified in the keyword, the search will only look for entity references in the XML document.

Table A-5 Mapping Symbols to Entity References

Entity references	Symbol represented
&lt;	<
&gt;	>
&amp;	&
&apos;	'
&quot;	"

Numeric Values

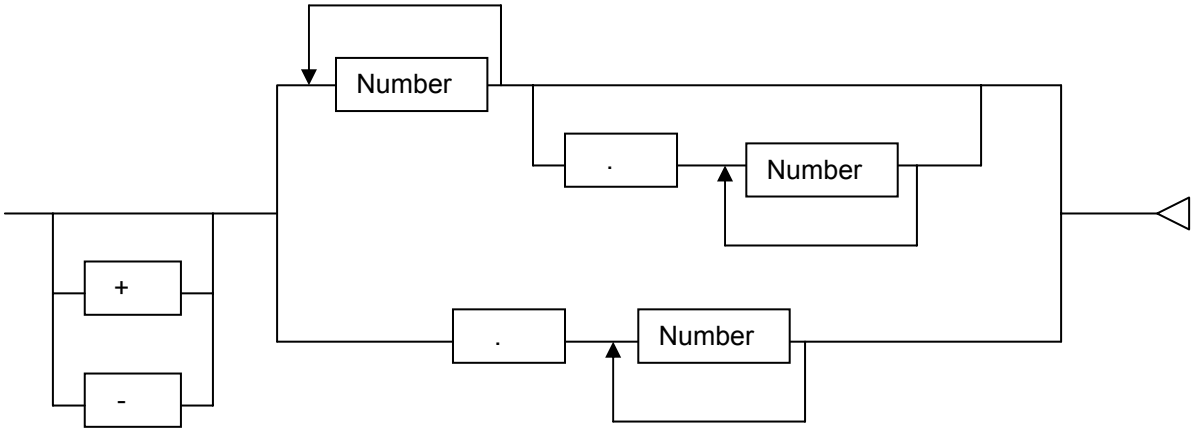


Figure A-10 Numeric Values

Numbers

Numbers between 0 and 9 inclusive can be specified.

If a numeric value is specified in the keyword, a numeric value search will be performed.

Refer to Numeric Value Searches for more information on numeric value searches.

Character String Searches

Three types of string search can be specified:

- Partial matches.
- Complete matches.
- Size comparison searches.

Relational Operators for Character String Searches

Relational operators for string searches specify how the string values of the element nodes specified by the path expression will be compared to the string specified by the keyword.

The relational operators are shown below.

Table A-6 Relational Operators for Character String Searches

Relational operators	Search type	Description
=	Partial match	TRUE if the string specified by the keyword is contained in the value of element node.
!=		TRUE if the string specified by the keyword is not contained in the value of element node.
==	Complete match	TRUE if the keyword matches the value of the element node exactly.
!==		TRUE if the keyword and the value of the element node are even slightly different.
<, <=, >, >=	Size comparison	Compares the size of the value of the element node to the keyword using the character encoding.

Partial Matches

Checks if the keyword is included in the value of the element node.

Example

```
/root/date = 'March'
```

This expression will be TRUE if the string 'March' is included in the value of the element node indicated by '/root/date'.

Complete Matches

Checks if the string is the same as the value of the element node.

Example

```
/root/date == 'March 09, 2004'
```

This expression will be TRUE if the value of the element node indicated by '/root/date' is equal to the string 'March 09, 2004'.

Size Comparison Searches

Compares the size of the value of the element node to the specified string, going from left to right using the character encoding value (the hexadecimal code value of a character).

Example

```
/root/date > 'March 09, 2004'
```

This expression compares the character encoding value of the element node indicated by '/root/date' to the character encoding value of the string 'March 09, 2004' one character at a time, starting from the leftmost character, and evaluates to TRUE if the character encoding value of the element node indicated by '/root/date' is larger.

Note

- For string comparisons, make sure that the string specified as the keyword is the same length as the value of the element node to which it is being compared.

Example

In the following example, the lookup document does not match the conditional expression.

Conditional Expression

```
/root/date <= 'March 09, 2004'
```

Target Lookup Document

```
<date>March 9, 2004</date>
```

Ellipses Searches

Ellipses searches are specified in order to search for strings that contain an arbitrary number of unknown characters. For example, specifying 'ab…c' for the keyword will search for all strings that contain any number of characters (including no characters at all) between 'ab' and 'c'.

Example

In the following example, both Document A and Document B match the search conditions.

```
/root//name = 'ab...c'
```

Document A

```
<root>
    <company>ABC
        <name>abxxxxc</name>
        <id>2000</id>
    </company>
</root>
```

Document B

```
<root>
    <company>ABC
        <name>abc</name>
        <id>2000</id>
    </company>
</root>
```

Note

- Ellipses can only be used for partial match searches.

Example

In the following example, '==' is specified as the relational operator, so ellipses searches cannot be performed. In this case, only Document A is regarded as matching the search conditions.

```
/root//name == 'ab...c'
```

Document A

```
<root>
    <company>ABC
        <name>ab...c</name>
        <id>2000</id>
    </company>
</root>
```

Document B

```
<root>
    <company>ABC
        <name>abxxxc</name>
        <id>2000</id>
    </company>
</root>
```

Numeric Value Searches

Numeric value searches use relational operators to specify either numeric matches or numeric size comparisons.

Relational Operators for Numeric Value Searches

Relational operators for numeric searches specify how numeric values contained in the element nodes specified by the path expression will be compared to the numeric value specified by the keyword.

Valid relational operators are shown below.

Table A-7 Relational Operators for Numeric Searches

Relational operator	Search type	Description
=	Match	TRUE if the numeric value contained in the element node matches the numeric value specified by the keyword.
!=		TRUE if the numeric value contained in the element node does not match the numeric value specified by the keyword.
<, <=, >, >=	Size comparison	Compares the size of the numeric value contained in the element node to the numeric value specified by the keyword.

Numeric Comparisons

Numeric comparisons extract the numeric value from the element node specified by the path expression and compare this value to the numeric value specified in the keyword.

Notes

- The first string in the value of the element node specified by the path expression that matches the format below is treated as a numeric value. The format used for numeric values is as shown below.

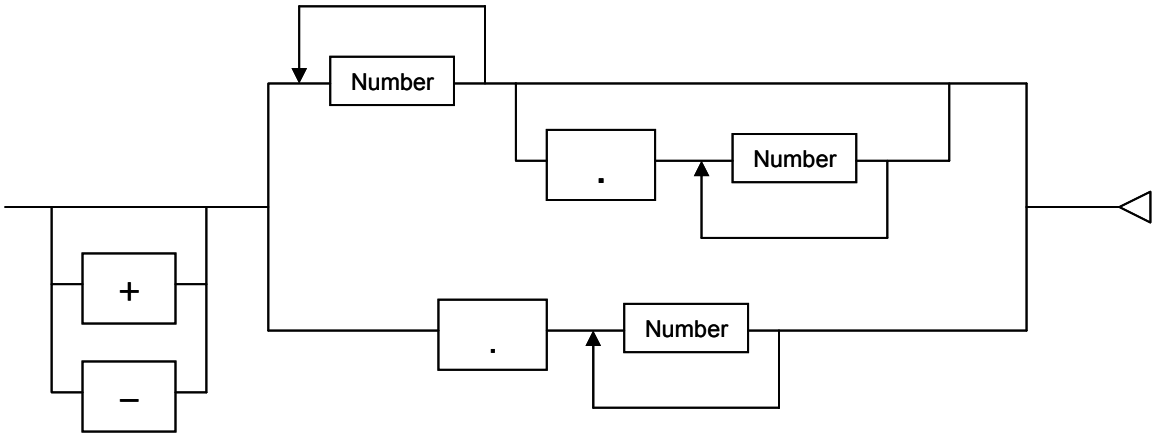


Figure A-11 Format for Numeric Values

- Commas in the integer part are ignored. If a decimal point is specified, the part from the first decimal place to a numeric character immediately before the first non-numeric character is treated as the fractional part.
- An error will occur if the integer part except for leading zeros has more than 18 digits.
- The first 18 digits of the decimal part are significant. Any subsequent digits are truncated.

Example

```
/doc/money = 1000
```

This expression evaluates to TRUE if the numeric value extracted from the text node below the 'money' element node below the 'doc' element node below the root node is equal to 1000.

In the following example, the value of the element node specified by the path expression contains multiple numeric values. In such cases, only the first numeric value is extracted.

Document A

```
<money>ABC123,456@789</money>
```

- 123456 is extracted.

Document B

```
<money>123456 7890123</money>
```

- 123456 is extracted.

Document C

```
<money>1,500 thousand yen</money>
```

- 1500 is extracted.

Note

- If the lookup data does not contain a valid numeric string, the result of the search conditions will be regarded as FALSE.

The following target search string does not contain a valid numeric value string.

```
<money></money>
```

Notes

- Symbols '//' cannot be entered at the end of a path expression when performing numeric comparisons.
- Symbols '*' cannot be entered at the end of a path expression when performing numeric comparisons.
- The number of digits in the numeric value specified as the keyword does not need to match that of the value of the element node specified with the path expression.

Keyword

```
/root/money > 1000
```

In the following example, the element node specified with the path expression will be regarded as matching the search conditions.

```
<money>1000.5</money>
```

Note

- The number of digits used to express the integer parts and the fractional parts of element node value does not need to be consistent across multiple XML documents.

  Document 1

  ```
  <money>1000.1</money>
  ```

  Document 2

  ```
  <money>2000.05</money>
  ```

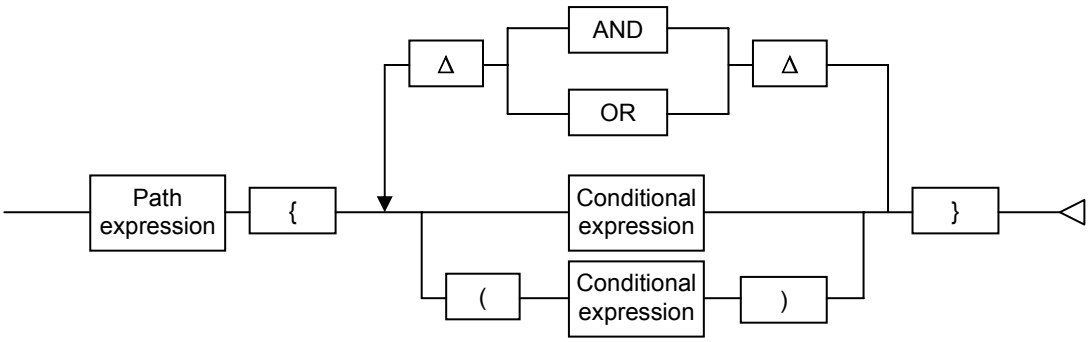
  Document 3

  ```
  <money>10.5</money>
  ```

Filter Expressions

Filter expressions are used to specify search conditions within the closed range of element nodes specified by the path expressions.

The definition format for filter expressions is shown below.



Δ: represents a space

Figure A-12 Filter Expressions

Filter expressions must have two or more conditional expressions.

Example

An example search expression containing filter expressions is shown below.

Search expression

```
/root/company/employee{/name = 'smith' AND /id = '1000'}
```

Document A

```
<root>
  <company>
    <employee>
      <name>smith</name>
      <id>1000</id>
    </employee>
  </company>
  <company>
    <employee>
      <name>jones</name>
      <id>2000</id>
    </employee>
  </company>
</root>
```

Document B

```
<root>
  <company>
    <employee>
      <name>smith</name>
      <id>2000</id>
    </employee>
  </company>
  <company>
    <employee>
      <name>jones</name>
      <id>1000</id>
    </employee>
  </company>
</root>
```

In this example, Document A matches the search expression but Document B does not.

Treat '/root/company/employee' as a single block. Document A contains data that matches the conditions in braces (/name = 'smith' AND /id = '1000'), while Document B does not.

Note

- The '*' path element cannot be specified as the final element of the path expressions used in filter expressions. Always specify an element name.

Example

In the following example, an error will occur because '*' is specified as the final path element of the path expression.

```
/root//company/*{/name = 'smith' AND /id = '1000'}
```

Return Expressions

Return expressions are used to specify data extraction formats for either extracting only particular elements from the XML document that meets specified search conditions, or for aggregating that data.

Return expressions use different formats, depending on whether aggregation is performed.

Format Used when not Aggregating

To return the entire record, specify '/' in the return expression.

The entire record will also be returned if a null character '' is specified in the return expression.

```
/
```

To return only certain element nodes or element values, specify either a path expression or a text expression. Single-line functions can also be specified to convert element values.

As shown in the following definition, multiple return items can be specified, separated by commas ','.

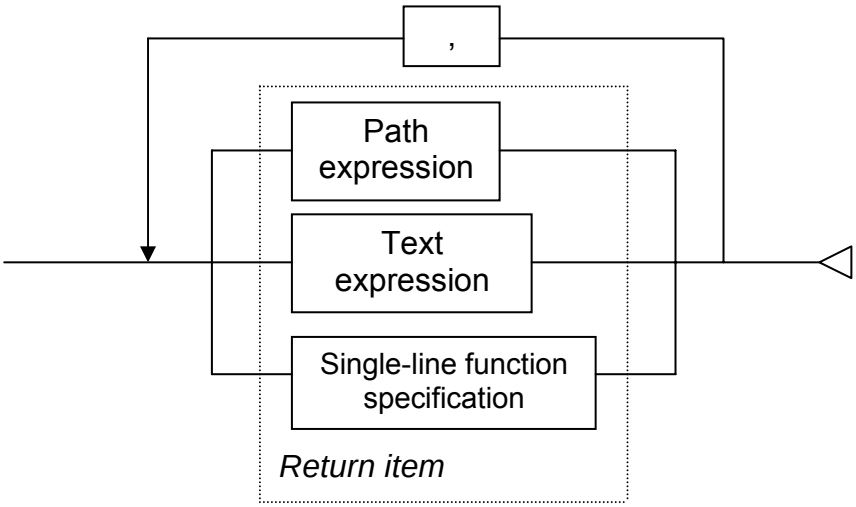


Figure A-13 Return Expression used when Aggregation is not Performed

Path Expressions

The path expression specifies which element nodes to return. The values of element nodes specified by the path expression are returned in XML format.

The '*' path element cannot be specified as the final path element.

Refer to Path Expressions for more information on path expressions.

Example

```
/root/company/name
```

Text Expressions

The text expression is used to specify which text nodes to return. The (string) values of the text nodes specified by text expressions are returned in text format.

Refer to Text Expressions for more information on text expressions.

Example

```
/root/company/name/text()
```

Single-line Function Specification

The single-line function specification uses a function to convert the value indicated by the text expression passed as an argument. The results of single-line function specifications are returned without tags.

Refer to Single-Line Function Specification for more information on single-line function specification.

Example

```
rlen(/root/company/name/text(),10)
```

Example Return Expressions when not Aggregating

Some example return expressions are shown below.

Document

```
<doc>
  <companyname>fujitsu</companyname>
  <employee>
    <name>smith</name>
    <id>2000</id>
    <age>30</age>
  </employee>
</doc>
```

Return Specification in XML Format

If a path expression is specified in the return item, the corresponding element nodes will be returned in XML format.

If multiple path expressions are specified in the return item, multiple elements are returned in the order specified. Results will be returned for each matching document, enclosed by the root tag.

If there are no matching elements, no elements will be returned.

Example 1

Return expression

```
/
```

Result: The entire record is returned.

```
<doc>
  <companyname>fujitsu</companyname>
  <employee>
    <name>smith</name>
    <id>2000</id>
    <age>30</age>
  </employee>
</doc>
```

Example 2

Return expression

```
/doc/employee/name
```

Result

```
<doc><name>smith</name></doc>
```

Example 3

Return expression

```
/doc/employee
```

Result

```
<doc><employee><name>smith</name><id>2000</id><age>30</age></employee>
</doc>
```

Example 4

Return expression

```
/doc/companyname,/doc/employee/id
```

Result

```
<doc><companyname>fujitsu</companyname><id>2000</id></doc>
```

Notes

- If the return expression includes a path expression, all return items must include a path expression. It cannot coexist with text expressions and single-line function specifications.
- If a return parameter that will match multiple elements is specified (that is, if more than one path expression is specified, or if "//" or "*" is specified), the application will not be able to determine the relationship between the elements that have been extracted, as shown below. In these situations, either extract the entire XML document, or extract data using a specification where the return parameter isolates a single element.

Example 1

If some elements do not exist, the application will not be able to determine which elements do not exist. In this example, there is no /doc/president/name element, and so only /doc/employee/name will be returned, but the application will not be able to determine whether the element returned is /doc/president/name or /doc/employee/name.

Return expression

```
/doc/president/name,/doc/employee/name
```

Result

```
<doc><name>smith</name></doc>
```

Example 2

The application will not be able to determine the path to the elements that have been extracted.

In this example, the application cannot determine whether the path to the name element is /doc/president/name or /doc/employee/name.

Document

The explanations that follow assume that the following XML document has been found.

```
<doc>
 <companyname>fujitsu</companyname>
  <president>
    <name>thompson</name>
    <id>1849</id>
    <age>61</age>
  </president>
  <employee>
    <name>smith</name>
    <id>2000</id>
    <age>30</age>
  </employee>
</doc>
```

Return expression

```
//name
```

Result

```
<doc><name>thompson</name><name>smith</name></doc>
```

Text Format Return Specification

If a text expression or a single-line function specification is specified in the return item, the results of the text expression or single-line function specification will be returned as a string.

If multiple path expressions are specified in the return item, multiple strings are returned in the order specified.

If there are no matching elements, an empty element will be indicated by consecutive delimiters.

Java APIs

Extracting the Search Results Obtained by the getString Method or the getStream Method

- The values returned by each return item are separated by commas ','.
- If the specified element occurs more than once in the same document, each occurrence will be separated by a vertical bar '|'.

Extracting the Search Results Obtained by the getStringArray Method

- Results are returned as a two dimensional array for each return item. The number of elements in the high-order array is equal to the number of return expressions. The number of elements in the low-order array is equal to the number of data items in a single document that apply to the specified return expression.

C APIs

The values returned by each return item are separated by the character code '\001'.

If the specified element occurs more than once in the same document, each element will be separated by the character code '\002'.

The character code '\001' is always attached at the end of the results returned.

In the following example, character codes '\001' and '\002' are expressed as '\1' and '\2' respectively.

Example 1: Example return item

Return expression

```
/doc/employee/name/text()
```

Result

- **For the Java APIs**

```
smith
```

- **For the C APIs**

```
smith \1
```

Example 2: Example where multiple return items are specified

Return expression

```
/doc/employee/name/text(),val(/doc/employee/age/text())
```

Result

- **For the Java APIs**

```
Smith,30
```

- **For the C APIs**

```
Smith \1 30 \1
```

Note

If a return parameter that will match multiple elements is specified (that is, if more than one path expression is specified, or if "//" or "*" is specified), the application will not be able to determine the relationship between the elements that have been extracted

Example

Because there is no ‘age element node’ for ‘jones’ in the document below, it is impossible to tell whether the ‘30’ in the third result relates to ‘smith’ or ‘jones’.

Document

```
<doc>
  <companyname>fujitsu</companyname>
  <employee>
    <name>smith</name>
    <id>2000</id>
    <age>30</age>
  </employee>
  <employee>
    <name>jones</name>
    <id>1000</id>
  </employee>
</doc>
```

Return expression

```
/doc/companyname/text(),/doc/employee/name/text(),val(/doc/employee/age/te
xt())
```

Result

- **For the Java APIs**

```
fujitsu,smith|jones,30
```

- **For the C APIs**

```
fujitsu \1 smith \2 jones \1 30 \1
```

Format Used when Aggregating

To aggregate data, specify text expressions, single-line function specifications, or aggregation function specifications in the return expression. Multiple specifications can be entered with each separated with commas. The return expression must include at least one aggregation function specification.

A sort expression must be specified when aggregating. Refer to Sort Expressions in Appendix A for more information on sort expressions.

Aggregation is performed as follows:

- data is first sorted according to the sort key specified by the text expressions or single-line functions specified in the sort expression
- data with the same sort key is treated as a single group
- the result of the aggregation function specification is determined for each group.

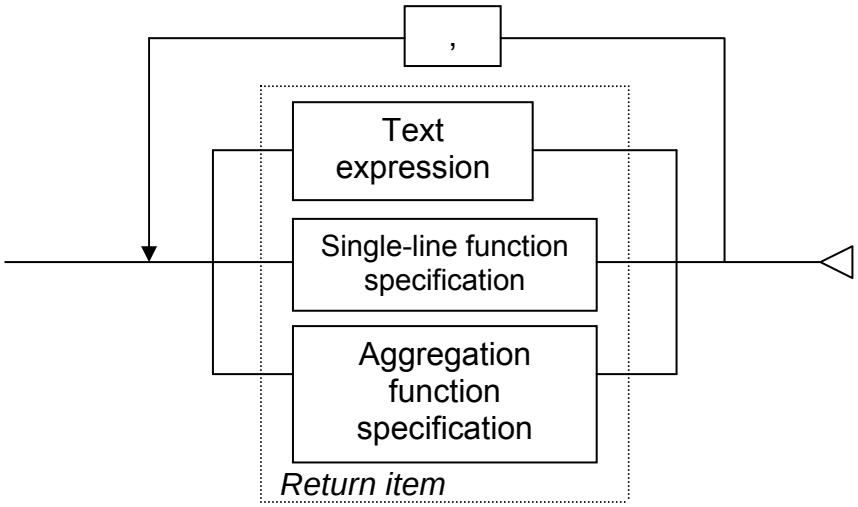


Figure A-14 Return Expressions used when Aggregating

Text Expressions

Specify a text expression when extracting a key for aggregation.

Refer to Text Expressions for more information on text expressions.

Note

- If a text expression is specified, it must be the same as the text expression used in the sort expression. The value used as the key for aggregation will be returned.

Example

```
/doc/company/employee/dept/text()
```

Single-line Function Specification

Specify single-line functions and extract the value of them when using single-line function specifications as the key for aggregation.

Refer to Single-Line Function Specification for more information on single-line function specification.

Note

- If a single-line function is specified, it must be the same as the single-line function specification used in the sort expression. The value used as the key for aggregation will be returned.

Example

```
rlen(/doc/company/employee/name/text(),3)
```

Aggregation Function Specifications

Aggregation function specifications aggregate the values specified by the text expression passed as an argument.

The definition format for aggregation function specifications is shown below.

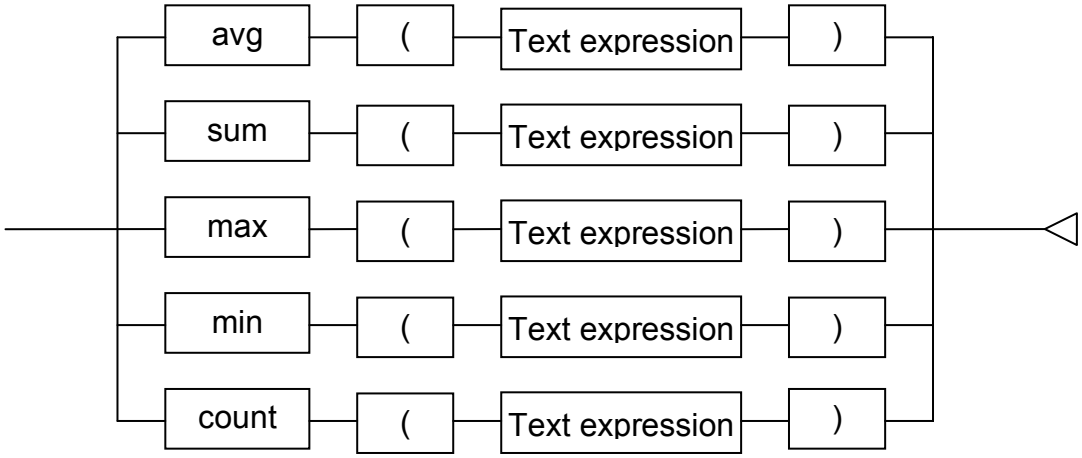


Figure A-15 Aggregation Function Specifications

Notes

- If the element node specified by the text expression is not in the document, it will not be included in the aggregation target.
- The *avg*, *sum*, *max* and *min* functions extract numeric values from the strings specified by the text expression and aggregate these values. The principles for extracting numeric values from text expressions are the same as for the *val* single-line function specification.

The *avg* Function

Extracts only numeric values from the strings expressed by the text expression and determines the average of these values.

The *sum* Function

Extracts only numeric values from the strings expressed by the text expression and determines the sum of these values.

The *max* Function

Extracts only numeric values from the strings expressed by the text expression and determines the maximum of these values.

The *min* Function

Extracts only numeric values from the strings expressed by the text expression and determines the minimum of these values.

The *count* Function

Determines the number of elements specified by the text expression.

Notes

- The '//' path operator cannot be specified in the text expression.
- The '*' path element cannot be specified in the text expression.

Example Return Expressions used when Aggregating

Some example return expressions are shown below.

Document A

```
<doc>
  <companyname>fujitsu</companyname>
  <employee>
    <name>smith</name>
    <id>2000</id>
    <age>30</age>
    <comment></comment>
  </employee>
</doc>
```

Document B

```
<doc>
  <companyname>fujitsu</companyname>
  <employee>
    <name>jones</name>
    <id>1000</id>
    <age>35</age>
    <comment>team leader</comment>
  </employee>
</doc>
```

Example 1: Example of the *avg* function

Return expression

```
avg(/doc/employee/age/text())
```

Sort expression

```
/doc/companyname/text()
```

Result

```
32.5
```

Example 2: Example of the *count* function

Return expression

```
count(/doc/employee/comment/text())
```

Sort expression

```
/doc/companyname/text()
```

Result

```
1
```

Note

- If the element specified by the text expression occurs more than once in the same document, only the first element value is used.

Example

In the following example, the text expression '/student/subject/test/score/text()' is passed as an argument to the *avg* function. In this case, Document A will not be included in the aggregation target, the value of Document B will be treated as 80, and Document C will be 0.

Document A

```
<student>
  <name>smith</name>
  <subject>
    <subjectname>science</subjectname>
  </subject>
  <name>smith</name>
</student>
```

Document B

```
<student>
  <name>jones</name>
  <subject>
    <subjectname>science</subjectname>
    <test>
      <date>January 12, 2004</date>
      <score>80</score>
    </test>
    <test>
      <date>December 08, 2003</date>
      <score>70</score>
    </test>
  </subject>
</student>
```

Document C

```
<student>
  <name>murphy</name>
  <subject>
    <subjectname>science</subjectname>
    <test>
      <date>January 12, 2004</date>
      <score>--</score>
    </test>
  </subject>
</student>
```

Return expression

```
avg(/student/subject/test/score/text())
```

Sort expression

```
/student/subjectname/text()
```

Result

```
40
```

Sort Expressions

Sort expressions are used to specify keys for either sorting or aggregating search results.

Sort Expression Format

Sort expressions can be specified using either text expressions or single-line function specifications. In each case, the sort order (ascending or descending) can be specified. Multiple specifications can be made with each separated by commas.

The definition format for sort expressions is shown below.

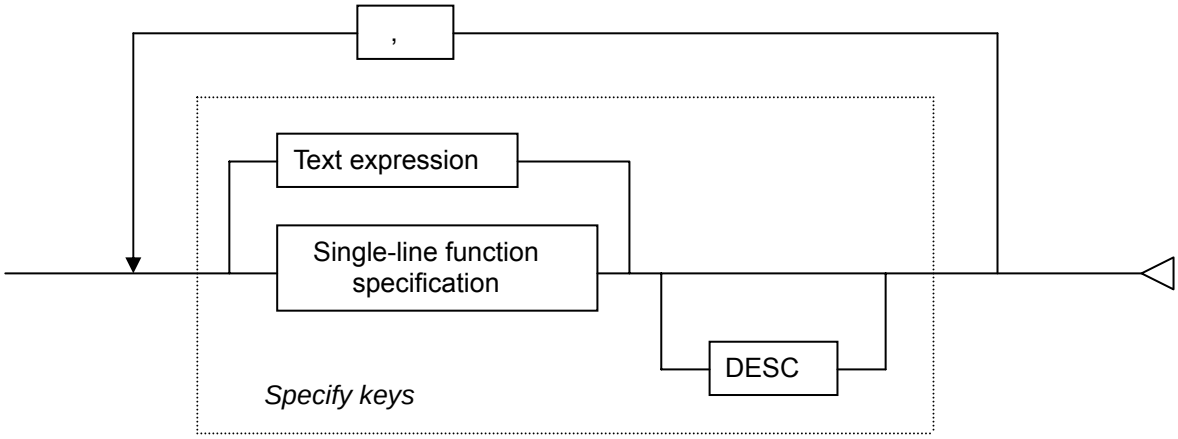


Figure A-16 Sort Expressions

Note

- Sort expressions can include up to eight key specifications.

Text Expressions

For text expressions, specify a text node to use as the key for sorting the data.

Refer to Text Expressions for more information on text expressions.

Notes

- The '//' path operator cannot be specified in the text expression.
- The '*' path element cannot be specified in the text expression.

Example

```
/company/employee/dept/text()
```

Single-line Function Specification

Single-line function specifications can be used to sort the results of single-line functions.

Refer to Single-Line Function Specification for more information on single-line function specifications.

Example

```
rlen(/company/name/text(),16)
```

DESC

Specify DESC to return results in descending order. If omitted, results will be returned in ascending order.

Example

```
/company/employee/dept/text() DESC
```

Sorting

The following rules apply to sorting:

- If a text expression is specified, results will be sorted using the string held by the text node specified by the text expression.
- By default, only the leftmost 20 bytes of the string held by the text node specified with the text expression are valid. The number of valid characters can be changed using the 'rlen' function. The range of values that can be specified is 1 to 128.
- Any spaces, tabs or linefeed characters in the string held by the text node specified with the text expression will be distinguished as valid values.
- If the text node specified by the text expression occurs more than once in the document, only the value of the first text node will be used.
- If the text node specified by the text expression is not in the document, the document will be placed last in the sorted results, regardless of the sort order.
- Specify the *val* function to sort numeric values. If the *val* function is specified, the data will be treated as numeric, and there is no need to line up digits.

Example

If the two document items below are sorted as strings, '20' will be larger than '100'. By specifying 'val(/value/text())' in the sort expression, these values will be treated as the numbers 100 and 20, and so 100 will be larger than 20.

```
<value>100</value>
```

```
<value>20</value>
```

Aggregation

The following rules apply to aggregating:

- If a text expression is specified, all documents where the text node specified by the text expression hold the same string are treated as a single group.
- By default, only the leftmost 20 bytes of the string held by the text node specified with the text expression are valid. The number of valid characters can be changed using the 'rlen' function. The range of values that can be specified is 1 to 128.
- Any spaces, tabs or linefeed characters in the string held by the text node specified with the text expression will be distinguished as valid values.
- If the text node specified by the text expression occurs more than once in the document, only the value of the first text node will be used.
- Documents where the text node specified by the text expression does not exist are treated as a single group where the group key has no value.
- Specify the *val* function to aggregate search results using a numeric key. If the *val* function is specified, the data will be treated as numeric, and so elements with the same numeric value will be treated as the same group even if some strings contain non-numeric values, or if the positioning of the decimal places below the decimal point is different.

Example

If the two documents below are aggregated using '/doc/key/text()' as the key, each will be placed in a separate group. If 'val(/doc/key/text())' is specified in the sort expression, both data items will be treated as the numeric value 1000, and so will be aggregated as the same group.

```
<doc>
  <key>net1000.00g</key>
  <ship>2000</ship>
</doc>
```

```
<doc>
  <key>1,000g</key>
  <ship>1000</ship>
</doc>
```

Example Sort Expressions

Some example sort expressions that either sort or aggregate data are shown below.

Document A

```
<employee>
  <name>smith</name><age>33</age><dept>sales</dept>
</employee>
```

Document B

```
<employee>
  <name>jones</name><age>30</age><dept>general affairs</dept>
</employee>
```

Document C

```
<employee>
  <name>murphy</name><age>22</age><dept>sales</dept>
</employee>
```

Document D

```
<employee>
  <name>fraser</name><age>54</age><dept>general affairs</dept>
</employee>
```

Document E

```
<employee>
  <name>morrison</name><dept>general affairs</dept>
</employee>
```

Document F

```
<employee>
  <name>mcdonald</name><age>42</age><dept>general affairs</dept>
</employee>
```

Entry Example of Data Sorting

Example 1

Return expression

```
/employee/name/text(),/employee/age/text()
```

Sort expression

```
val(/employee/age/text())
```

Result

- **For the Java APIs**

```
murphy,22
jones,30
smith,33
mcdonald,42
fraser,54
morrison,
```

- **For the C APIs**

```
murphy \1 22 \1
jones \1 30 \1
smith \1 33 \1
mcdonald \1 42 \1
fraser \1 54 \1
morrison \1 \1
```

Example 2

Return expression

```
/employee/name,/employee/age
```

Sort expression

```
val(/employee/age/text())
```

Result

```
<employee><name>murphy</name><age>22</age></employee>
<employee><name>jones</name><age>30</age></employee>
<employee><name>smith</name><age>33</age></employee>
<employee><name>mcdonald</name><age>42</age></employee>
<employee><name>fraser</name><age>54</age></employee>
<employee><name>morrison</name><age></age></employee>
```

Entry Example of Data Aggregation

Example

Return expression

```
avg(/employee/age/text()),count(/employee/age/text()),/employee/dept/text(
)
```

Sort expression

```
/employee/dept/text()
```

Result

- **For the Java APIs**

```
42,3,general affairs
27.5,2,sales
```

- **For the C APIs**

```
42 \1 3 \1 general affairs \1
27.5 \1 2 \1 sales \1
```

Appendix B

Sample Java Programs

This appendix provides sample programs that use Java APIs.

- Searching Data
- Updating Data

Searching Data

Use the APIs provided with Shunsaku to obtain search results by specifying search conditions for the data to be retrieved.

Using Shunsaku APIs, the following searches can be performed:

The following operations can be performed using the Java APIs:

- Finding the number of XML documents that match the search conditions
- Obtaining the XML documents that match the search conditions in a specified format
- Obtaining all of a particular XML document
- Finding XML documents that match the search conditions and obtaining the documents after they are sorted
- Finding XML documents that match the search conditions and obtaining the documents after their contents are aggregated

Refer to the Java API Reference for details on Java APIs.

The example below uses 'hotel reservations situation search' to explain these operations. This example assumes that the XML document below exists.

Refer to Appendix F, Notes on XML Documents for details on XML documents.

Sample Document

```
<document>
    <base>
        <name>Hotel 1</name>
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>9000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, two minutes walk to train station
XX</note>
</document>
<document>
    <base>
        <name>Hotel 2</name>
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>6000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, five minutes walk to train station
XX</note>
</document>
<document>
    <base>
        <name>Hotel 3</name>
```

```
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>7500</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, ten minutes walk to train station
XX</note>
</document>
<document>
    <base>
        <name>Hotel 4</name>
        <prefecture>Osaka</prefecture>
        <address>Kita-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>5000</price>
    </base>
    <information>
        <date>2004/07/10</date>
    </information>
    <note>En-suite bathroom and toilet, three minutes walk to train
station ZZ</note>
</document>
<document>
    <base>
        <name>Hotel 5</name>
        <prefecture>Osaka</prefecture>
        <address>Kita-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>6000</price>
    </base>
    <information>
        <date>2004/07/10</date>
    </information>
    <note>En-suite bathroom and toilet, two minutes walk to train station
ZZ</note>
</document>
<document>
    <base>
        <name>Hotel 6</name>
        <prefecture>Kanagawa</prefecture>
        <address>Kohoku-ku Yokohama-shi Kanagawa</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>8000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, two minutes walk to train station
XX</note>
</document>
<document>
    <base>
        <name>Hotel 7</name>
        <prefecture>Kanagawa</prefecture>
        <address>Kohoku-ku Yokohama-shi Kanagawa</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>7000</price>
    </base>
    <information>
```

```
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, three minutes walk to train
station XX</note>
</document>
<document>
    <base>
        <name>Hotel 8</name>
        <prefecture>Kanagawa</prefecture>
        <address>Kanagawa-ku Yokohama-shi Kanagawa</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>6000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, five minutes walk to train station
XX</note>
</document>
```

Find the Number of XML Documents that Match the Search Conditions

If there is a large amount of data to be searched, it is not practical to search all data items that match the conditional expression. In such cases, it is useful to find out how many data items match the specified condition.

The example below shows how to use the Java APIs to find out how many hotels match the date and location specified as the search condition.

Search Conditions

'How many hotels are there in Osaka with vacancies on July 18 2004.'

Perform a search using the date (2004/07/18) and the location (Osaka) as the search conditions.

Example Using the Java APIs

The following is a sample program using the Java APIs.

```
import com.fujitsu.shun.ShunConnection;
import com.fujitsu.shun.ShunPreparedStatement;
import com.fujitsu.shun.ShunResultSet;
import com.fujitsu.shun.common.ShunException;

/***     Obtain the number of records that meet specified search
conditions     ***/
public class JavaAPISample1 {
    public static void main(String[] args) {

        ShunConnection con = null;
        ShunPreparedStatement pstmt = null;
        ShunResultSet rs = null;

        try {

            // Search conditional expression
            String sQuery = "/document/base/prefecture == 'Osaka' AND
/document/information/date == '2004/07/18'";
            // Return expressions
            String sReturn = "/";
            // Number of hits
            int iHitNum = 0;

            // Create the ShunConnection object
            con = new ShunConnection("DServer", 33101);

            // Specify a search expression and create the
ShunPreparedStatement object
            pstmt = con.prepareSearch(sQuery, sReturn);

            //Set the number of items to return per request
            pstmt.setRequest(1,0);

            //Execute the search and create the ShunResultSet object
            rs = pstmt.executeSearch();

            // Get the number of hits
            iHitNum = rs.getHitCount();
            System.out.println("Number of hits = " + iHitNum);
```

```
            rs.close();
            pstmt.close();
            con.close();
        }
        catch (ShunException ex) {
            System.out.println("Error code:" + ex.getErrCode());
            System.out.println("Error level:" + ex.getErrLevel());
            System.out.println("Error message:" + ex.getMessage());
            ex.printStackTrace();
        }
        catch (Exception ex) {
            System.out.println("Error message:" + ex.getMessage());
            ex.printStackTrace();
        }
        // Recovery process in the event of an error
        finally {
            try {
                if ( rs!=null ) rs.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( pstmt!=null ) pstmt.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( con!=null ) con.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }
}
```

Execution Results

```
Number of hits = 3
```

Obtain the XML Documents that Match the Search Conditions in a Specified Format

Part of the data obtained as search results is sometimes used as additional conditions to perform the next search operation. In this case, it is common to return only part of the data as search results instead of the entire data.

The example below shows how to use the Java APIs to find out how many hotels match the date and location specified as the search condition, as well as to get partial information.

Search Conditions

'I would like to know the names and accommodation rates of up to 30 hotels in Osaka that are available on 18 July 2004.'

In this case, specify the date (2004/07/18) and the location (Osaka) as search conditions; specify the hotel name and its accommodation rate as search results; and then execute the search.

Example Using the Java APIs

The following is a sample program using the Java APIs.

```
import com.fujitsu.shun.ShunConnection;
import com.fujitsu.shun.ShunPreparedStatement;
import com.fujitsu.shun.ShunResultSet;
import com.fujitsu.shun.common.ShunException;

/***     Obtain the number of records that match the specified search
conditions and the data     ***/
public class JavaAPISample2 {
    public static void main(String[] args) {

        ShunConnection con = null;
        ShunPreparedStatement pstmt = null;
        ShunResultSet rs = null;

        try {
            // Search conditional expression
            String sQuery = "/document/base/prefecture == 'Osaka' AND
/document/information/date == '2004/07/18'";
            // Return expression
            String sReturn = "/document/base/name, /document/base/price";
            // Number of hits
            int iHitNum = 0;
            // Data counter
            int iDataCounter = 1;

            // Create the ShunConnection object
            con = new ShunConnection("DServer", 33101);

            // Specify a search expression and create the
ShunPreparedStatement object
            pstmt = con.prepareSearch(sQuery,sReturn);

            //Set the number of items to return per request
            pstmt.setRequest(1,30);

            //Execute the search and create the ShunResultSet object
            rs = pstmt.executeSearch();

            // Get the number of hits
```

```
            iHitNum = rs.getHitCount();
            System.out.println("Number of hits       = " + iHitNum);

            // Obtain data that matches the search conditions one item at
a time
            while(rs.next()) {
               System.out.println("[Result " + iDataCounter + "] = " +
rs.getString());
               iDataCounter++;
        }


            rs.close();
            pstmt.close();
            con.close();

        } catch (ShunException ex) {
            System.out.println("Error code:" + ex.getErrCode());
            System.out.println("Error level:" + ex.getErrLevel());
            System.out.println("Error message:" + ex.getMessage());
            ex.printStackTrace();
        }
        finally {
            try {
                if ( rs!=null ) rs.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( pstmt!=null ) pstmt.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( con!=null ) con.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }
}
```

Execution Results

```
Number of hits       = 3
[Result 1] = <document><name>Hotel 1</name><price>9000</price></document>
[Result 2] = <document><name>Hotel 2</name><price>6000</price></document>
[Result 3] = <document><name>Hotel 3</name><price>7500</price></document>
```

Obtain All of a Particular XML Document

After completing the Find the Number of XML Documents that Match the Search Conditions and Obtain the XML Documents that Match the Search Conditions in a Specified Format search operations, the next step is to refine the search and obtain all the data for a result item based on a more precise set of conditions.

The following example uses the Java APIs to return all the data for one of the hotels whose name was returned using Obtain the XML Documents that Match the Search Conditions in a Specified Format search operation.

Search Conditions

'I would like to obtain detailed information about one of the hotels in Osaka available on 18 July 2004.'

Specify the date (2004/07/18) and the location (Osaka) as search conditions and execute the search. Also use the record ID corresponding to the hotel to obtain detailed information.

Example Using the Java APIs

The following is a sample program using the Java APIs.

```
import com.fujitsu.shun.ShunConnection;
import com.fujitsu.shun.ShunPreparedRecordID;
import com.fujitsu.shun.ShunPreparedStatement;
import com.fujitsu.shun.ShunResultSet;
import com.fujitsu.shun.common.ShunException;

/***     Obtain all record information corresponding to the specified
record ID     ***/
public class JavaAPISample3 {
    public static void main(String[] args) {

        ShunConnection con = null;
        ShunPreparedStatement pstmt = null;
        ShunPreparedRecordID rid = null;
        ShunResultSet rs = null;

        try {

            // Search conditional expression
            String sQuery = "/document/base/prefecture == 'Osaka' AND
/document/information/date == '2004/07/18'";
            // Return expression
            String sReturn = "/document/base/name/text()";
            // Record information
            String sRecordID = "";
            // Number of hits
            int iHitNum = 0;

            // Create the ShunConnection object
            con = new ShunConnection("DServer", 33101);

            // Specify a search expression and create the
ShunPreparedStatement object
            pstmt = con.prepareSearch(sQuery,sReturn);

            //Set the number of items to return per request
            pstmt.setRequest(1,30);
```

```
            //Execute the search and create the ShunResultSet object
            rs = pstmt.executeSearch();

            // Get the number of hits
            iHitNum = rs.getHitCount();
            System.out.println("Number of hits       = " + iHitNum);

            // Create the ShunPreparedRecordID object
            rid = con.prepareSearchRecordID();

            // Obtain one data item that matches the search conditions
            while(rs.next()) {

                // Set record information for Hotel 1
                if(rs.getString().equals("Hotel 1")) {
                    rid.add(rs.getRecordID());
                }
            }

            rs.close();
            pstmt.close();

            // If acquisition of the relevant record ID is successful,
refer to the detailed data
            if(0 < rid.getCount()) {

                // Use the specified record information to create the
ShunResultSet object
                rs = rid.searchByRecordID();

                while(rs.next()) {
                    // Obtain data using the record information
specification
                    System.out.println("[details] = " + rs.getString());
                }

                rs.close();
            }

            rid.close();
            con.close();

        } catch (ShunException ex) {
            System.out.println("Error code:" + ex.getErrCode());
            System.out.println("Error level:" + ex.getErrLevel());
            System.out.println("Error message:" + ex.getMessage());
            ex.printStackTrace();
        }
        finally {
            try {
                if ( rs!=null ) rs.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( pstmt!=null ) pstmt.close();
            }
```

```
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( rid!=null ) rid.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( con!=null ) con.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }
}
```

Execution Results

```
Number of hits       = 3
[detail] = <document>
    <base>
        <name>Hotel 1</name>
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>9000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, two minutes walk to train station
XX</note>
</document>
```

Find XML Documents that Match the Search Conditions and Obtain the Documents after they are Sorted

When performing a search, it is sometimes desirable to obtain the search results after they have been sorted according to a specific element.

The following example specifies the date and location as search conditions and obtains the number of hotels that match the conditions, as well as partial information. It shows how the Java APIs are used to sort data in descending order according to the accommodation rate as it is obtained.

Search Conditions

'I would like to know the hotels in Osaka that are available on 18 July 2004, and I want to sort the results in a descending order according to the hotel rate.'

Specify the date (2004/07/18) and the location (Osaka) as search conditions and specify the accommodation rate as a sort condition, then execute the search.

Example Using the Java APIs

The following is a sample program using the Java APIs.

```
import com.fujitsu.shun.ShunConnection;
import com.fujitsu.shun.ShunPreparedStatement;
import com.fujitsu.shun.ShunResultSet;
import com.fujitsu.shun.common.ShunException;

/***     Obtain the sorted data that matches the specified search
conditions     ***/
public class JavaAPISample4 {
    public static void main(String[] args) {

        ShunConnection con = null;
        ShunPreparedStatement pstmt = null;
        ShunResultSet rs = null;

        try {
            // Search conditional expression
            String sQuery = "/document/base/prefecture == 'Osaka' AND
/document/information/date == '2004/07/18'";
            // Return expression
            String sReturn = "/document/base/name, /document/base/price";
            // Sort expression
            String sSort = "val(/document/base/price/text()) DESC";
            // Number of hits
            int iHitNum = 0;
            // Data counter
            int iDataCounter = 1;

            // Create the ShunConnection object
            con = new ShunConnection("DServer", 33101);

            // Specify a search expression and create the
ShunPreparedStatement object
            pstmt = con.prepareSearch(sQuery,sReturn);

            // Specify sort expression
            pstmt.setSort(sSort);

            //Set the number of items to return per request
            pstmt.setRequest(1,30);
```

```
            //Execute the search and create the ShunResultSet object
            rs = pstmt.executeSearch();

            // Obtain data that matches the search conditions one item at
a time
            while(rs.next()) {
               System.out.println("[Result " + iDataCounter + "] = " +
rs.getString());
               iDataCounter++;
      }

            rs.close();
            pstmt.close();
            con.close();

        } catch (ShunException ex) {
            System.out.println("Error code:" + ex.getErrCode());
            System.out.println("Error level:" + ex.getErrLevel());
            System.out.println("Error message:" + ex.getMessage());
            ex.printStackTrace();
        }
        finally {
            try {
                if ( rs!=null ) rs.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( pstmt!=null ) pstmt.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( con!=null ) con.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }
}
```

Execution Results

```
[Result 1] = <document><name>Hotel 1</name><price>9000</price></document>
[Result 2] = <document><name>Hotel 3</name><price>7500</price></document>
[Result 3] = <document><name>Hotel 2</name><price>6000</price></document>
```

Find XML Documents that Match the Search Conditions and Obtain the Documents after their Contents are Aggregated

When performing a search, it is sometimes desirable to obtain the search results after the value of a specific element has been aggregated.

The following example shows how to use the Java APIs when obtaining the cheapest hotel, the most expensive hotel, and the average hotel rate by specifying the location as a search condition.

Search Conditions

'Of the hotels that are available on 18 July 2004, I would like to know the cheapest hotel, the most expensive hotel, and the average hotel rate for each area.'

Specify the date (2004/07/18) as a search condition, specify an aggregation expression (min, max, avg) for the results to be obtained and then aggregate the results.

Example Using the Java APIs

The following is a sample program using the Java APIs.

```
import com.fujitsu.shun.ShunConnection;
import com.fujitsu.shun.ShunPreparedStatement;
import com.fujitsu.shun.ShunResultSet;
import com.fujitsu.shun.common.ShunException;

/***     Aggregate the data matching the specified search condition
***/
public class JavaAPISample5 {
    public static void main(String[] args) {

        ShunConnection con = null;
        ShunPreparedStatement pstmt = null;
        ShunResultSet rs = null;

        try{

            // Search conditional expression
            String sQuery = "/document/information/date == '2004/07/18'";
            // Return expression
            String sReturn = "/document/base/prefecture/text(),
min(/document/base/price/text()), max(/document/base/price/text()),
avg(/document/base/price/text())";
            // Sort expression
            String sSort   = "/document/base/prefecture/text()";

            // Create the ShunConnection object
            con = new ShunConnection("DServer", 33101);

            // Specify a search expression and create the
ShunPreparedStatement object
            pstmt = con.prepareSearch(sQuery,sReturn);

            // Specify sort expression
            pstmt.setSort(sSort);

            //Set the number of items to return per request
            pstmt.setRequest(1,30);

            //Execute the search and create the ShunResultSet object
            rs = pstmt.executeSearch();
```

```
            // Obtain data that matches the search conditions
            int counter = 0;
            String[][] result;
            while (rs.next()) {
                counter++;
                System.out.println("[Result " + counter + "]");

                result = rs.getStringArray();
                System.out.println("  area                    :" +
result[0][0]);
                System.out.println("  cheapest hotel rate     :" +
result[1][0]);
                System.out.println("  most expensive hotel rate:" +
result[2][0]);
                System.out.println("  average hotel rate      :" +
result[3][0]);
            }

            rs.close();
            pstmt.close();
            con.close();
        }
        // Processing performed when an error occurs during execution of
the application
        catch (ShunException ex)
        {
            System.out.println("Error code:" + ex.getErrCode());
            System.out.println("Error level:" + ex.getErrLevel());
            System.out.println("Error message:" + ex.getMessage());
            ex.printStackTrace();
        }
        finally {
            try {
                if ( rs!=null ) rs.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( pstmt!=null ) pstmt.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( con!=null ) con.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }
}
```

Execution Results

```
[Result 1]
  area                      :Kanagawa
  cheapest hotel rate       :6000
  most expensive hotel rate:8000
  average hotel rate        :7000
[Result 2]
  area                      :Osaka
  cheapest hotel rate       :6000
  most expensive hotel rate:9000
  average hotel rate        :7500
```

Updating Data

The Java APIs provided by Shunsaku are used to update data.

The following can be done using Java APIs:

- Adding Data
- Deleting Data

Refer to the Java API Reference for details on Java APIs.

In this section, the 'Hotel reservation status search' sample document provided in Searching Data will be used to explain the update procedure.

Refer to Appendix F, Notes on XML Documents, for details on XML documents.

Adding Data

The following example shows how the Java APIs are used to add data.

Data to be Added

'I would like to add an item of information about a Kanagawa hotel (Hotel 9 information).'

Assemble the data to be added and then add the data.

Example Using the Java APIs

The following is a sample program using the Java APIs.

```
import com.fujitsu.shun.ShunConnection;
import com.fujitsu.shun.ShunPreparedStatement;
import com.fujitsu.shun.common.ShunException;

/***     Add the specified data     ***/
public class JavaAPISample6 {
    public static void main(String[] args) {

        ShunConnection con = null;
        ShunPreparedStatement pstmt = null;

        try{

            // Create the ShunConnection object
            con = new ShunConnection("DServer", 33101);

            // Create a ShunPreparedStatement object to add the data
            pstmt = con.prepareInsert();

            // Create the data to be added
            String addData = "<document>"
                        + "    <base>"
                        + "        <name>Hotel 9</name>"
                        + "        <prefecture>Kanagawa</prefecture>"
                        + "        <address>Knanagawa-ku Yokohama-shi
Kanagawa</address>"
                        + "        <detail>http://xxxxx.co.jp</detail>"
                        + "        <price>6000</price>"
```

```
                        + "    </base>"
                        + "    <information>"
                        + "        <date>2004/07/18</date>"
                        + "    </information>"
                        + "<note>En-suite bathroom and toilet, five
minutes walk to train station XX</note>"
                        + "</document>";

            // Add data
            pstmt.add(addData);

            // Perform data addition
            pstmt.executeInsert();

            System.out.println("Addition complete");

            pstmt.close();
            con.close();
        }
        // Processing performed when an error occurs during execution of
the application
        catch (ShunException ex)
        {
            System.out.println("Error code:" + ex.getErrCode());
            System.out.println("Error level:" + ex.getErrLevel());
            System.out.println("Error message:" + ex.getMessage());
            ex.printStackTrace();
        }
        catch (Exception e)
        {
            System.out.println("ERROR MESSAGE : " + e.getMessage());
            e.printStackTrace();
        }
        finally {
            try {
                if ( pstmt!=null ) pstmt.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( con!=null ) con.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }
}
```

Execution Results

```
Addition complete
```

Deleting Data

The following example shows how the Java APIs are used to delete data.

Search Conditions

'I would like to delete the Hotel 9 data from the data for hotels in Kanagawa that are available on 18 July 2004.'

Perform a search using the date (2004/07/18) and the location (Kanagawa) as search conditions, and then delete the data that matches the hotel name 'Hotel 9'.

Example Using the Java APIs

The following is a sample program using the Java APIs.

```
import com.fujitsu.shun.ShunConnection;
import com.fujitsu.shun.ShunPreparedRecordID;
import com.fujitsu.shun.ShunPreparedStatement;
import com.fujitsu.shun.ShunResultSet;
import com.fujitsu.shun.common.ShunException;

/***     Delete the specified data     ***/
public class JavaAPISample7 {
    public static void main(String[] args) {

        ShunConnection con = null;
        ShunPreparedStatement pstmt = null;
        ShunPreparedRecordID rid = null;
        ShunResultSet rs = null;

        try{

            // Search expression
            String sQuery = "/document/base/prefecture == 'Kanagawa' AND
/document/information/date == '2004/07/18'";

            // Return expression
            String sReturn = "/document/base/name/text()";

            // Create the ShunConnection object
            con = new ShunConnection("DServer", 33101);

            // Specify a search expression and create the
ShunPreparedStatement object
            pstmt = con.prepareSearch(sQuery,sReturn);

            //Set the number of items to return per request
            pstmt.setRequest(1,30);

            // Create the Shun PreparedRecordIDCreate object
            rid = con.prepareDeleteRecordID();

            // Execute the search and create the ShunResultSet object
            rs = pstmt.executeSearch();

            // Obtain the record ID of Hotel 9
            while(rs.next())
            {
                if(rs.getString().equals("Hotel 9"))
```

```
                {
                    rid.add(rs.getRecordID());
                }
            }


            rs.close();
            pstmt.close();

            // If acquisition of the record ID is successful, delete the
corresponding data.
            if(0 < rid.getCount()) {

                rid.deleteByRecordID();
                System.out.println("Deletion complete");
            }

            rid.close();
            con.close();
        }
        // Processing performed when an error occurs during execution of
the application
        catch (ShunException ex)
        {
            System.out.println("Error code:" + ex.getErrCode());
            System.out.println("Error level:" + ex.getErrLevel());
            System.out.println("Error message:" + ex.getMessage());
            ex.printStackTrace();
        }
        finally {
            try {
                if ( rs!=null ) rs.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( pstmt!=null ) pstmt.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( rid!=null ) rid.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
            try {
                if ( con!=null ) con.close();
            }
            catch (ShunException ex){
                System.out.println("Error code:" + ex.getErrCode());
```

```
                System.out.println("Error level:" + ex.getErrLevel());
                System.out.println("Error message:" + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }
}
```

Execution Results

```
Deletion complete
```

Appendix C

Sample C Programs

This appendix provides examples of programs that use C APIs.

- Searching Data
- Updating Data

Searching Data

Use the APIs provided by Shunsaku to obtain search results by specifying search conditions for the data to be retrieved.

The following operations can be performed using the C APIs:

- Finding the number of XML documents that match the search conditions
- Obtaining the XML documents that match the search conditions in a specified format
- Obtaining all of a particular XML document
- Finding XML documents that match the search conditions and obtaining the documents after they are sorted
- Finding XML documents that match the search conditions and obtaining the documents after their contents are aggregated

Refer to the C API Reference for details on C APIs.

The example below uses 'hotel reservation status search' to explain these operations. This example assumes that the XML document below exists.

Refer to Appendix F, Notes on XML Documents for details on XML documents.

Sample Document

```
<document>
    <base>
        <name>Hotel 1</name>
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>9000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, two minutes walk to train station
XX</note>
</document>
<document>
    <base>
        <name>Hotel 2</name>
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>6000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, five minutes walk to train station
XX</note>
</document>
<document>
    <base>
        <name>Hotel 3</name>
        <prefecture>Osaka</prefecture>
```

```
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>7500</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, ten minutes walk to train station
XX</note>
</document>
<document>
    <base>
        <name>Hotel 4</name>
        <prefecture>Osaka</prefecture>
        <address>Kita-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>5000</price>
    </base>
    <information>
        <date>2004/07/10</date>
    </information>
    <note>En-suite bathroom and toilet, three minutes walk to train
station ZZ</note>
</document>
<document>
    <base>
        <name>Hotel 5</name>
        <prefecture>Osaka</prefecture>
        <address>Kita-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>6000</price>
    </base>
    <information>
        <date>2004/07/10</date>
    </information>
    <note>En-suite bathroom and toilet, two minutes walk to train station
ZZ</note>
</document>
<document>
    <base>
        <name>Hotel 6</name>
        <prefecture>Kanagawa</prefecture>
        <address>Kohoku-ku Yokohama-shi Kanagawa</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>8000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, two minutes walk to train station
XX</note>
</document>
<document>
    <base>
        <name>Hotel 7</name>
        <prefecture>Kanagawa</prefecture>
        <address>Kohoku-ku Yokohama-shi Kanagawa</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>7000</price>
    </base>
    <information>
        <date>2004/07/18</date>
```

```
    </information>
    <note>En-suite bathroom and toilet, three minutes walk to train
station XX</note>
</document>
<document>
    <base>
        <name>Hotel 8</name>
        <prefecture>Kanagawa</prefecture>
        <address>Kanagawa-ku Yokohama-shi Kanagawa</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>6000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, five minutes walk to train station
XX</note>
</document>
```

Find the Number of XML Documents that Match the Search Conditions

If there is a large amount of data to be searched, it is not practical to search all data items that match the conditional expression. In such cases, it is useful to find out how many data items match the specified condition.

The example below shows how to use the C APIs to find out how many hotels match the date and location specified as the search condition.

Search Conditions

'How many hotels in Osaka have vacancies on July 18 2004?'

Perform a search using the date (2004/07/18) and the location (Osaka) as the search condition.

Example Using the C APIs

The following is a sample program using the C APIs.

```
#include <stdio.h>
#include <string.h>
#include "libshun.h"

int main() {

    /*** Declaration of work variables ***/
    int     sts;

    /*** Variable declaration for input parameters ***/
    char    hostname[24];

    /*** Variable declaration for output parameters ***/
    int     Hit_Cnt;

    /*** Initialize input parameters***/
    /* Clear input parameters */
    memset(hostname, 0, sizeof hostname);
    strcpy(hostname, "DServer");

    /*** Initialize output parameters ***/
    /* Clear output parameters */
    sts = 0;
    Hit_Cnt = 0;

    /*** Call the API (the shunsearch1 function) ***/
    sts = shunsearch1(hostname,
                      33101,
                      "/document {/base/prefecture == 'Osaka' "
                      "AND /information/date == '2004/07/18'}",
                      NULL,
                      &Hit_Cnt);

    /*** Export output parameters ***/
    if ( sts == 0 ) {
        printf("Number of hits              = %d\n", Hit_Cnt);
        return 0;
    } else {
        printf("Error code    :%d\n", sts);
        return 1;
```

```
    }
}
```

Execution Results

```
Number of hits = 3
```

Obtain the XML Documents that Match the Search Conditions in a Specified Format

Part of the data obtained as search results is sometimes used as additional conditions to perform the next search operation. In this case, it is common to return only part of the data as search results instead of the entire data.

The example below shows how to use the C APIs to find out how many hotels match the date and location specified as the search condition, as well as to get partial information.

Search Conditions

'I would like to know the names and accommodation rates of up to 30 hotels in Osaka that are available on 18 July 2004.'

Perform a search using the date (July 18, 2004) and the location (Osaka) as the search conditions, and then obtain the names and accommodation rates of 30 hotels from the results of this search.

Example Using the C APIs

The following is a sample program using the C APIs.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "libshun.h"

#define START_NO        1
#define RECORD_CNT      30
#define SECURE_SIZE     4096

int main() {

    /*** Declaration of work variables ***/
    int     i;
    int     sts;
    char    *wkData;

    /*** Variable declaration for input parameters ***/
    char    hostname[24];
    int     StartNo     = START_NO;
    int     RecordCnt   = RECORD_CNT;
    int     Secure_Size = SECURE_SIZE;

    /*** Variable declaration for output parameters ***/
    int     Hit_Cnt;
    int     Return_Cnt;
    int     Stored_Size;
    Sdsma   *Dsma;
    char    *Data;

    /*** Initialize input parameters ***/
    /* Clear input parameters */
    memset(hostname, 0, sizeof hostname);
    strcpy(hostname, "DServer");

    /*** Initialize output parameters ***/
    /* Clear output parameters */
    sts = 0;
    Hit_Cnt = 0;
```

```
    Return_Cnt = 0;
    Stored_Size = 0;

    /*** Allocate memory for output parameter response data ***/
    /* Allocate the area for response data */
    Dsma = (Sdsma*)malloc(sizeof(Sdsma) * RecordCnt);
    memset(Dsma,0,sizeof(Sdsma) * RecordCnt);
    for (i = 0; i < RecordCnt; i++) {
        Dsma[i].Rec_Ctl = (char*)malloc(COND_CTL_LEN);
        Dsma[i].Rec_ID = (char*)malloc(ROW_ID_LEN);
        memset(Dsma[i].Rec_Ctl, 0, COND_CTL_LEN);
        memset(Dsma[i].Rec_ID, 0, ROW_ID_LEN);
        Dsma[i].Rec_Ptr = 0;
        Dsma[i].Rtn_Len = 0;
    }
    Data = (char*)malloc(sizeof(char) * Secure_Size);
    memset(Data, 0, sizeof(char) * Secure_Size);

    /*** Call the API (the shunsearch2 function) ***/
    sts = shunsearch2(hostname,
                      33101,
                      StartNo,
                      RecordCnt,
                      "/document/base/prefecture == 'Osaka' "
                      "AND /document/information/date == '2004/07/18'",
                      "/document/base/name, /document/base/price",
                      NULL,
                      NULL,
                      Secure_Size,
                      &Hit_Cnt,
                      &Return_Cnt,
                      &Stored_Size,
                      Dsma,
                      Data);

    /*** Export output parameters ***/
    if ( sts == 0 ) {
        printf("Number of hits              = %d\n", Hit_Cnt);

        if ( Hit_Cnt != 0 ) {
            for (i = 0; i < Return_Cnt; i++) {
                if ( Dsma[i].Rtn_Len != 0 ) {
                    wkData = (char*)malloc(Dsma[i].Rtn_Len + 1);
                    memset(wkData, 0, Dsma[i].Rtn_Len + 1);
                    memcpy(wkData, Dsma[i].Rec_Ptr, Dsma[i].Rtn_Len);
                    printf("[Result]No. %d = %s\n ", StartNo+i, wkData);
                    free(wkData);
                } else {
                    printf("[Result]No. %d = No data", StartNo+i);
                }
            }
        }
    } else {
        printf("Error code    :%d\n", sts);
        printf("Size of the area for returned data :%d\n", Stored_Size);
    }

    /*** Release memory for the output parameters ***/
    for (i = 0; i < RecordCnt; i++) {
        free(Dsma[i].Rec_Ctl);
        free(Dsma[i].Rec_ID);
    }
```

```
    free(Dsma);
    free(Data);

    if ( sts == 0 ) return 0;
    else            return 1;
}
```

Execution Results

```
Number of hits       = 3
[Result 1] = <document><name>Hotel 1</name><price>9000</price></document>
[Result 2] = <document><name>Hotel 2</name><price>6000</price></document>
[Result 3] = <document><name>Hotel 3</name><price>7500</price></document>
```

Obtain All of a Particular XML Document

After completing the 'Find the Number of XML Documents that Match the Search Conditions' and 'Searching Data' search operations, the next step is to refine the search and obtain all the data based on a more precise set of conditions.

The following example uses the C APIs to return all the data for one of the hotels whose name was returned using the 'Obtain the XML Documents that Match the Search Conditions in a Specified Format' search operation.

Search Conditions

Find the names of 30 hotels in Osaka with vacancies on July 18 2004

Perform a search using the date (2004/07/18) and the location (Osaka) as the search conditions, and then obtain detailed information on a particular hotel using the record identifier corresponding to that hotel name.

Example Using the C APIs

The following is a sample program using the C APIs.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "libshun.h"

#define START_NO        1
#define RECORD_CNT      30
#define SECURE_SIZE     4096

typedef struct record_id{
    char    cond_id[COND_CTL_LEN];
    char    row_id[ROW_ID_LEN];
} record_id_t;

int  main() {

    /*** Declaration of work variables ***/
    int     i;
    int     sts;
    int     sts2;
    char    *wkData;

    /*** Variable declaration for input parameters ***/
    char    hostname[24];
    int     StartNo     = START_NO;
    int     RecordCnt   = RECORD_CNT;
    int     Secure_Size = SECURE_SIZE;

    /*** Variable declaration for output parameters (shunsearch2) ***/
    int     Hit_Cnt;
    int     Return_Cnt;
    int     Stored_Size;
    Sdsma   *Dsma;
    char    *Data;
    record_id_t *record_area = NULL;

    /*** Variable declaration for output parameters (shunsearch3) ***/
    int     Hit_Cnt2;
    int     Return_Cnt2;
```

```
    int     Stored_Size2;
    Sdsma   *Dsma3;
    char    *Data3;
    record_id_t *record_area3 = NULL;
    char    **row_id_area = NULL;
    char    **cond_id_area = NULL;

    /*** Initialize input parameters ***/
    /* Clear input parameters */
    memset(hostname, 0, sizeof hostname);
    strcpy(hostname, "DServer");

    /*** Initialize output parameters ***/
    /* Clear output parameters */
    sts = 0;
    Hit_Cnt = 0;
    Return_Cnt = 0;
    Stored_Size = 0;

    /*** Allocate memory for output parameter response data ***/
    /* Allocate the area for response data */
    record_area = (record_id_t *)malloc(RecordCnt * sizeof(record_id_t));
    memset(record_area, 0x00, RecordCnt * sizeof(record_id_t));

    Dsma = (Sdsma *)malloc(RecordCnt * sizeof(Sdsma));
    memset(Dsma, 0x00, RecordCnt * sizeof(Sdsma));

    for (i = 0; i < RecordCnt; i++) {
        Dsma[i].Rec_Ctl = (char *)&(record_area[i].cond_id);
        Dsma[i].Rec_ID = (char *)&(record_area[i].row_id);
    }

    Data = (char *)malloc(Secure_Size);
    memset(Data, 0x00, Secure_Size);

    /*** Call the API (the shunsearch2 function)***/
    sts = shunsearch2(hostname,
                      33101,
                      StartNo,
                      RecordCnt,
                      "/document/base/prefecture == 'Osaka' "
                      "AND /document/information/date == '2004/07/18'",
                      "/document/base/name, /document/base/price",
                      NULL,
                      NULL,
                      Secure_Size,
                      &Hit_Cnt,
                      &Return_Cnt,
                      &Stored_Size,
                      Dsma,
                      Data);

    /*** Export output parameters ***/
    if ( sts != 0 ) {
        printf("Error code(shunsearch2)   :%d\n", sts);
        printf("Size of the area for returned data(shunsearch2) :%d\n",
Stored_Size);
        goto skip_search3;
    }

    printf("Number of hits(shunsearch2) = %d\n", Hit_Cnt);
    printf("Number of responses(shunsearch2)   = %d\n", Return_Cnt);
```

```
    if ( Return_Cnt == 0 ) {
        printf("There were no responses\n");
        goto skip_search3;
    }

    /*** Initialize output parameters ***/
    /* Clear output parameters */
    sts2 = 0;
    Hit_Cnt2 = 0;
    Return_Cnt2 = 0;
    Stored_Size2 = 0;

    row_id_area = (char **)malloc(Return_Cnt * sizeof(char *));

    memset(row_id_area, 0x00, Return_Cnt * sizeof(char *));
    for (i = 0; i < Return_Cnt; i++) {
        row_id_area[i] = (char *)&(record_area[i].row_id);
    }

    cond_id_area = (char **)malloc(Return_Cnt * sizeof(char *));

    memset(cond_id_area, 0x00, Return_Cnt * sizeof(char *));
    for (i = 0; i < Return_Cnt; i++) {
        cond_id_area[i] = (char *)&(record_area[i].cond_id);
    }

    /* Allocate a return area for the record identifier (ROW_ID) for the
reply data
 */
    /* Allocate an area equal to (the number of items to return per
request) x sizeof(record_id_t) */
    record_area3 = (record_id_t *)malloc(Return_Cnt *
sizeof(record_id_t));
    memset(record_area3, 0x00, Return_Cnt * sizeof(record_id_t));

    /* Allocate an area for the management array that stores reply data */
    /* Allocate an area equal to (the number of items to return per
request) x sizeof(Dsma) */
    Dsma3 = (Sdsma *)malloc(Return_Cnt * sizeof(Sdsma));
    memset(Dsma3, 0x00, Return_Cnt * sizeof(Sdsma));

    for (i = 0; i < Return_Cnt; i++) {
        Dsma3[i].Rec_Ctl = (char *)&(record_area3[i].cond_id);
        Dsma3[i].Rec_ID = (char *)&(record_area3[i].row_id);
    }

    Data3 = (char *)malloc(Secure_Size);
    memset(Data3, 0x00, Secure_Size);

    /*** Call the API (the shunsearch3 function) ***/
    sts2 = shunsearch3(hostname,
                       33101,
                       Return_Cnt,
                       cond_id_area,
                       row_id_area,
                       NULL,
                       Secure_Size,
                       &Hit_Cnt2,
                       &Return_Cnt2,
                       &Stored_Size2,
                       Dsma3,
```

```
                        Data3);

    /*** Export output parameters ***/
    if ( sts2 == 0 ) {
        printf("Number of hits(shunsearch3) = %d\n", Hit_Cnt2);
        printf("Number of responses(shunsearch3)   = %d\n", Return_Cnt2);

        if ( Return_Cnt2 > 0 ) {
            for (i = 0; i < Return_Cnt2; i++){
                wkData = (char *)malloc(Dsma3[i].Rtn_Len + 1);
                memset(wkData, 0x00, Dsma3[i].Rtn_Len + 1);
                if ( Dsma3[i].Rtn_Len != 0 ) {
                    memcpy(wkData,
                           Dsma3[i].Rec_Ptr,
                           Dsma3[i].Rtn_Len);
                }
                printf("[Details]No. %d = %s\n", i+1, wkData);
                free(wkData);
            }
        }
    } else {
        printf("Error code(shunsearch3)   :%d\n", sts2);
        printf("Size of the area for returned data(shunsearch3) :%d\n",
Stored_Size2);
    }

    /*** Release memory for the output parameters ***/

    /* Release the area for response data for shunsearch3*/
    free(cond_id_area);
    free(row_id_area);
    free(record_area3);
    free(Dsma3);
    free(Data3);
    sts = sts2;

  skip_search3:
    /* Release the area for response data for shunsearch2*/
    free(record_area);
    free(Dsma);
    free(Data);

    if ( sts == 0 ) return 0;
    else            return 1;
}
```

Execution Results

```
Number of hits(shunsearch2) = 3
Number of responses(shunsearch2)   = 3
Number of hits(shunsearch3) = 3
Number of responses(shunsearch3)   = 3
[Details]No. 1 = <document>
    <base>
        <name>Hotel 1</name>
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>9000</price>
    </base>
```

```
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, two minutes walk to train station
XX</note>
</document>
[Detail]No. 2 = <document>
    <base>
        <name>Hotel 2</name>
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>6000</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, five minutes walk to train station
XX</note>
</document>
[Detail]No. 3 = <document>
    <base>
        <name>Hotel 3</name>
        <prefecture>Osaka</prefecture>
        <address>Chuo-ku Osaka</address>
        <detail>http://xxxxx.co.jp</detail>
        <price>7500</price>
    </base>
    <information>
        <date>2004/07/18</date>
    </information>
    <note>En-suite bathroom and toilet, ten minutes walk to train station
XX</note>
</document>
```

Find XML Documents that Match the Search Conditions and Obtain the Documents after They are Sorted

When performing a search, it is sometimes desirable to obtain the search results after they have been sorted according to a specific element.

The following example specifies the date and location as search conditions and obtains the number of hotels that match the conditions, as well as partial information.

It shows how the C APIs are used to obtain the results after they have been sorted according to the accommodation rate.

Search Conditions

'I would like to know 30 hotels in Osaka that are available on 18 July 2004, and I want to sort the results in a descending order according to the accommodation rate.'

Specify the date (2004/07/18) and the location (Osaka) as search conditions and execute the search. Then, obtain the top 30 hotel names after they have been sorted in descending order according to the accommodation rate.

Example Using the C APIs

The following is a sample program using the C APIs.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "libshun.h"

#define START_NO        1
#define RECORD_CNT      30
#define SECURE_SIZE     4096

int main() {

    /*** Declaration of work variables ***/
    int     i;
    int     sts;
    char    *wkData;

    /*** Variable declaration for input parameters ***/
    char    hostname[24];
    int     StartNo     = START_NO;
    int     RecordCnt   = RECORD_CNT;
    int     Secure_Size = SECURE_SIZE;

    /*** Variable declaration for output parameters ***/
    int     Hit_Cnt;
    int     Return_Cnt;
    int     Available_Cnt;
    int     Stored_Size;
    Sdsma   *Dsma;
    char    *Data;

    /*** Initialize input parameters ***/
    /* Clear input parameters */
    memset(hostname, 0, sizeof hostname);
    strcpy(hostname, "DServer");
```

```
    /*** Initialize output parameters ***/
    /* Clear output parameters */
    Hit_Cnt = 0;
    Return_Cnt = 0;
    Available_Cnt = 0;
    Stored_Size = 0;

    /*** Allocate memory for output parameter response data ***/
    /* Allocate the area for response data */
    Dsma = (Sdsma*)malloc(sizeof(Sdsma) * RecordCnt);
    memset(Dsma,0,sizeof(Sdsma) * RecordCnt);
    for (i = 0; i < RecordCnt; i++) {
        Dsma[i].Rec_Ctl = (char*)malloc(COND_CTL_LEN);
        Dsma[i].Rec_ID = (char*)malloc(ROW_ID_LEN);
        memset(Dsma[i].Rec_Ctl, 0, COND_CTL_LEN);
        memset(Dsma[i].Rec_ID, 0, ROW_ID_LEN);
        Dsma[i].Rec_Ptr = 0;
        Dsma[i].Rtn_Len = 0;
    }
    Data = (char*)malloc(sizeof(char) * Secure_Size);
    memset(Data, 0, sizeof(char) * Secure_Size);

    /*** Call the API (the shunsort function) ***/
    sts = shunsort(hostname,
                   33101,
                   StartNo,
                   RecordCnt,
                   "/document/base/prefecture == 'Osaka' "
                   "AND /document/information/date == '2004/07/18'",
                   "/document/base/name, /document/base/price",
                   "/document/base/price/text() DESC",
                   Secure_Size,
                   &Hit_Cnt,
                   &Return_Cnt,
                   &Available_Cnt,

                   &Stored_Size,
                   Dsma,
                   Data);

    /*** Export output parameters ***/
    if ( sts == 0 ) {
        printf("Number of hits              = %d\n", Hit_Cnt);

        if ( Hit_Cnt != 0 ) {
            for (i = 0; i < Return_Cnt; i++) {
                if ( Dsma[i].Rtn_Len != 0 ) {
                    wkData = (char*)malloc(Dsma[i].Rtn_Len + 1);
                    memset(wkData, 0, Dsma[i].Rtn_Len + 1);
                    memcpy(wkData, Dsma[i].Rec_Ptr, Dsma[i].Rtn_Len);
                    printf("[Result]No. %d = %s\n", StartNo+i, wkData);
                    free(wkData);
                } else {
                    printf("[Result]No. %d = No data", StartNo+i);
                }
            }
        }
    } else {
        printf("Error code    :%d\n", sts);
        printf("Size of the area for returned data :%d\n", Stored_Size);
    }
```

```
    /*** Release memory for the output parameters ***/
    for (i = 0; i < RecordCnt; i++) {
        free(Dsma[i].Rec_Ctl);
        free(Dsma[i].Rec_ID);
    }
    free(Dsma);
    free(Data);

    if ( sts == 0 ) return 0;
    else            return 1;
}
```

Execution Results

```
Number of hits              = 3
[Result 1] = <document><name>Hotel 1</name><price>9000</price></document>
[Result 2] = <document><name>Hotel 3</name><price>7500</price></document>
[Result 3] = <document><name>Hotel 2</name><price>6000</price></document>
```

Find XML Documents that Match the Search Conditions and Obtain the Documents after Their Contents are Aggregated

When performing a search, it is sometimes desirable to obtain the search results after the value of a specific element has been aggregated.

The following example shows how to use the C APIs when obtaining the number of hotels that meet the date and location specified in the search conditional expression, as well as the cheapest hotel, the most expensive hotel, and the average hotel rate.

Search Conditions

'Of the hotels that are available on 18 July 2004, I would like to know the cheapest hotel, the most expensive hotel, and the average hotel rate for each area.'

Specify the date (2004/07/18) as a search condition and perform a search.

Example Using the C APIs

The following is a sample program using the C APIs.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "libshun.h"

#define START_NO        1
#define RECORD_CNT      30
#define SECURE_SIZE     4096

int main() {

    /*** Declaration of work variables ***/
    int     i, j, k;
    int     sts;
    char    *wkData;

    /*** Variable declaration for input parameters ***/
    char    hostname[24];
    int     StartNo     = START_NO;
    int     RecordCnt   = RECORD_CNT;
    int     Secure_Size = SECURE_SIZE;

    /*** Variable declaration for output parameters ***/
    int     Hit_Cnt;
    int     Return_Cnt;
    int     Available_Cnt;
    int     Stored_Size;
    Sdsma   *Dsma;
    char    *Data;

    /*** Initialize input parameters ***/
    /* Clear input parameters */
    memset(hostname, 0, sizeof hostname);
    strcpy(hostname, "DServer");

    /*** Initialize output parameters ***/
    /* Clear output parameters */
    Hit_Cnt = 0;
    Return_Cnt = 0;
```

```
    Available_Cnt = 0;
    Stored_Size = 0;

    /*** Allocate memory for output parameter response data ***/
    /* Allocate the area for response data */
    Dsma = (Sdsma*)malloc(sizeof(Sdsma) * RecordCnt);
    memset(Dsma,0,sizeof(Sdsma) * RecordCnt);
    for (i = 0; i < RecordCnt; i++) {
        Dsma[i].Rec_Ctl = (char*)malloc(COND_CTL_LEN);
        Dsma[i].Rec_ID = (char*)malloc(ROW_ID_LEN);
        memset(Dsma[i].Rec_Ctl, 0, COND_CTL_LEN);
        memset(Dsma[i].Rec_ID, 0, ROW_ID_LEN);
        Dsma[i].Rec_Ptr = 0;
        Dsma[i].Rtn_Len = 0;
    }
    Data = (char*)malloc(sizeof(char) * Secure_Size);
    memset(Data, 0, sizeof(char) * Secure_Size);

    /*** Call the API (the shunsort function) ***/
    sts = shunsort(hostname,
                   33101,
                   StartNo,
                   RecordCnt,
                   "/document/information/date == '2004/07/18'",
                   "/document/base/prefecture/text(),"
                   "count(/document/base/prefecture/text()),"
                   "max(/document/base/price/text()),"
                   "min(/document/base/price/text()),"
                   "avg(/document/base/price/text())",
                   "/document/base/prefecture/text()",
                   Secure_Size,
                   &Hit_Cnt,
                   &Return_Cnt,
                   &Available_Cnt,
                   &Stored_Size,
                   Dsma,
                   Data);

    /*** Export output parameters ***/
    if ( sts == 0 ) {
        printf("Number of hits              = %d\n", Hit_Cnt);

        if ( Hit_Cnt != 0 ) {
            for (i = 0; i < Return_Cnt; i++) {
                if ( Dsma[i].Rtn_Len != 0 ) {
                    wkData = (char*)malloc(Dsma[i].Rtn_Len + 1);
                    memset(wkData, 0, Dsma[i].Rtn_Len + 1);
                    printf("[Result]No. %d =", StartNo+i);
                    for (j = 0, k = 0; j < Dsma[i].Rtn_Len; j++) {
                        if ( Dsma[i].Rec_Ptr[j] != '\001' &&
                             Dsma[i].Rec_Ptr[j] != '\002' ) {
                            wkData[k] = Dsma[i].Rec_Ptr[j];
                            k++;
                        } else {
                            printf(" %s", wkData);
                            memset(wkData, 0, Dsma[i].Rtn_Len + 1);
                            k = 0;
                        }
                    }
                    if ( k > 0 ) {
                        printf(" %s", wkData);
                    }
```

```
                    printf("\n");
                    free(wkData);

                } else {
                    printf("[Result]No. %d = No data", StartNo+i);
                }
            }
        }
    } else {
        printf("Error code    :%d\n", sts);
        printf("Size of the area for returned data :%d\n", Stored_Size);
    }

    /*** Release memory for the output parameters ***/
    for (i = 0; i < RecordCnt; i++) {
        free(Dsma[i].Rec_Ctl);
        free(Dsma[i].Rec_ID);
    }
    free(Dsma);
    free(Data);

    if ( sts == 0 ) return 0;
    else            return 1;
}
```

Execution Results

```
Number of hits              = 6
[Result 1] = Kanagawa 3 8000 6000 7000
[Result 2] = Osaka 3 9000 6000 7500
```

Updating Data

The C APIs provided by Shunsaku are used to update data.

The C APIs enable the following operations to be performed:

- Adding Data
- Deleting Data

Refer to C API Reference for details on C APIs.

In this section, the 'hotel reservation status search' sample document provided in Searching Data will be used to explain the update procedure.

Refer to Notes on XML Documents in Appendix F for details on XML documents.

Adding Data

The following example shows how the C APIs are used to add data.

Example Using the C APIs

The following is a sample program using the C APIs.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "libshun.h"

#define ADDDATA_CNT     1
#define ADDDATA_LEN     1024

int  main() {

    /*** Declaration of work variables ***/
    int     i;
    int     sts;

    /*** Variable declaration for input parameters ***/
    char    hostname[24];
    int     DataArray_Cnt = ADDDATA_CNT;
    char    **DataArray;

    /*** Initialize input parameters ***/
    /* Clear input parameters */
    memset(hostname, 0, sizeof hostname);
    strcpy(hostname, "DServer");

    /*** Allocate memory for input parameters and additional record data
***/
    /* Allocate the area for additional record data */
    DataArray = (char**)malloc(sizeof(char *) * DataArray_Cnt);
    memset(DataArray,0,sizeof(char *) * DataArray_Cnt);
    for (i = 0; i < DataArray_Cnt; i++) {
        DataArray[i] = (char *)malloc(ADDDATA_LEN);
        memset(DataArray[i], 0, ADDDATA_LEN);
        strcpy(DataArray[i],
```

```
                "<document>"
                "    <base>"
                "        <name>Hotel 9</name>"
                "        <prefecture>Kanagawa</prefecture>"
                "        <address>Kanagawa-ku Yokohama-shi
Kanagawa</address>"
                "        <detail>http://xxxxx.co.jp</detail>"
                "        <price>6000</price>"
                "    </base>"
                "    <information>"
                "        <date>2004/07/18</date>"
                "    </information>"
                "<note>En-suite bathroom and toilet, five minutes walk to
train station XX</note>"
                "</document>");
    }

    /*** Call the API (the shunadd function) ***/
    sts = shunadd(hostname,
                  33101,
                  DataArray_Cnt,
                  DataArray);

    if ( sts != 0 ) {
        printf("Error code    :%d\n", sts);
    } else {
        printf("Addition successful\n");
    }

    for (i = 0; i < DataArray_Cnt; i++) {
        free(DataArray[i]);
    }
    free(DataArray);

    if ( sts == 0 ) return 0;
    else            return 1;
}
```

Execution Results

```
Addition successful
```

Deleting Data

The following example shows how the C APIs are used to delete data.

Search Conditions for Data to Delete

'I would like to delete the data for hotels in Osaka that are available on 18 July 2004.'

Perform a search using the date (2004/07/18) and the location (Osaka) as search conditions, and then delete the data of up to 30 of the hotels that are found.

Example Using the C APIs

The following is a sample program using the C APIs.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "libshun.h"

#define START_NO        1
#define RECORD_CNT      30
#define SECURE_SIZE     4096

typedef struct record_id{
    char    cond_id[COND_CTL_LEN];
    char    row_id[ROW_ID_LEN];
} record_id_t;

int  main() {

    /*** Declaration of work variables ***/
    int     i;
    int     sts;
    int     sts2;

    /*** Variable declaration for input parameters ***/
    char    hostname[24];
    int     StartNo     = START_NO;
    int     RecordCnt   = RECORD_CNT;
    int     Secure_Size = SECURE_SIZE;

    /*** Variable declaration for output parameters (shunsearch2) ***/
    int     Hit_Cnt;
    int     Return_Cnt;
    int     Stored_Size;
    Sdsma   *Dsma;
    char    *Data;
    record_id_t *record_area = NULL;

    /*** Variable declaration for output parameters (shundeletebyrecid)
***/
    char    **row_id_area = NULL;
    char    **cond_id_area = NULL;

    /*** Initialize input parameters ***/
    /* Clear input parameters */
    memset(hostname, 0, sizeof hostname);
    strcpy(hostname, "DServer");

    /*** Initialize output parameters ***/
```

```
    /* Clear output parameters */
    sts = 0;
    Hit_Cnt = 0;
    Return_Cnt = 0;
    Stored_Size = 0;

    /*** Allocate memory for output parameter response data***/
    /* Allocate the area for response data */
    record_area = (record_id_t *)malloc(RecordCnt * sizeof(record_id_t));
    memset(record_area, 0x00, RecordCnt * sizeof(record_id_t));

    Dsma = (Sdsma *)malloc(RecordCnt * sizeof(Sdsma));
    memset(Dsma, 0x00, RecordCnt * sizeof(Sdsma));

    for (i = 0; i < RecordCnt; i++) {
        Dsma[i].Rec_Ctl = record_area[i].cond_id;
        Dsma[i].Rec_ID = record_area[i].row_id;
    }

    Data = (char *)malloc(Secure_Size);
    memset(Data, 0x00, Secure_Size);

    /*** Call the API (the shunsearch2 function)***/
    sts = shunsearch2(hostname,
                      33101,
                      StartNo,
                      RecordCnt,
                      "/document/base/prefecture == 'Kanagawa' "
                      "AND /document/information/date == '2004/07/18' "
                      "AND /document/base/name == 'Hotel 9'",
                      "/document/base/name, /document/base/price",
                      NULL,
                      NULL,
                      Secure_Size,
                      &Hit_Cnt,
                      &Return_Cnt,
                      &Stored_Size,
                      Dsma,
                      Data);

    /*** Export output parameters ***/
    if ( sts != 0 ) {
        printf("Error code(shunsearch2)   :%d\n", sts);
        printf("Size of the area for returned data(shunsearch2) :%d\n",
Stored_Size);
        goto skip_delete;
    }

    printf("Number of hits(shunsearch2) = %d\n", Hit_Cnt);
    printf("Number of responses(shunsearch2)   = %d\n", Return_Cnt);

    if ( Return_Cnt == 0 ) {
        printf("There were no responses\n");
        goto skip_delete;
    }

    /*** Initialize output parameters ***/
    /* Clear output parameters */
    sts2 = 0;

    row_id_area = (char **)malloc(Return_Cnt * sizeof(char *));
```

```
    memset(row_id_area, 0x00, Return_Cnt * sizeof(char *));
    for (i = 0; i < Return_Cnt; i++) {
        row_id_area[i] = record_area[i].row_id;
    }

    cond_id_area = (char **)malloc(Return_Cnt * sizeof(char *));

    memset(cond_id_area, 0x00, Return_Cnt * sizeof(char *));
    for (i = 0; i < Return_Cnt; i++) {
        cond_id_area[i] = record_area[i].cond_id;
    }

    /*** Call the API (the shundeletebyrecid function) ***/
    sts2 = shundeletebyrecid(hostname,
                             33101,
                             Return_Cnt,
                             cond_id_area,
                             row_id_area);

    if ( sts2 != 0 ) {
        printf("Error code(shundeletebyrecid) :%d\n", sts2);
    } else {
        printf("Deletion successful\n");
    }

    /*** Release memory for the output parameters ***/
    /* Release the area for response data for shunsearch2*/
    free(cond_id_area);
    free(row_id_area);
    sts = sts2;

  skip_delete:
    free(Data);
    free(Dsma);
    free(record_area);

    if ( sts == 0 ) return 0;
    else            return 1;
}
```

Execution Results

```
Number of hits(shunsearch2) = 1
Number of responses(shunsearch2)   = 1
Deletion successful
```

Appendix D

Allowable Values

This appendix explains the allowable values associated with developing Shunsaku applications.

- Search Expressions and Return Expressions
- Sort Requests
- Aggregation Requests

Search Expressions and Return Expressions

The following table lists the allowable values associated with search expressions and return expressions.

Table D-1 Allowable Values for Search Expressions and Return Expressions

Item	Allowable value
Length of search expression	From 1 to 65, 535 bytes (excluding \0)
Length of return expression	From 1 to 65, 535 bytes (excluding \0)

Sort Requests

The following table lists the allowable values associated with sort requests.

Table D-2 Allowable Values for Sort Requests

Item	Allowable value	Remarks
Maximum number of keys that can be specified	8	
Sort expression	From 1 to 65, 535 bytes (excluding \0)	
Total sort key length	1 to 2,000 bytes	The total sort key length is the combined length of all the sort keys in key specifications. Refer to Relationship between the Total Sort Key Length and the Maximum Number of Items that Can be Returned for more information on how the total sort key length affects the number of items that can be returned.
Number of characters when the *rlen* function is specified in key specifications	1 to 128	Refer to Table D-4, Character Encoding and the Byte Length of Each Character of the Character String Produced by the *rlen* Function, for the length of the character string produced by the *rlen* function.
Number of items that can be returned	100 to 1,000 records	When the total length of sort keys is 200 bytes or less, the records corresponding to the top 1,000 items can be returned. When the total length of sort keys exceeds 200 bytes, the number of records that can be returned is indicated in Relationship between the Total Sort Key Length and the Maximum Number of Items that Can be Returned.

Relationship between the Total Sort Key Length and the Maximum Number of Items that Can be Returned

The total sort key length is equal to the total combined length of all sort keys used in key specifications. The sort key length differs according to the key specification format, as shown below.

Table D-3 Relationship between the Total Sort Key Length and the Maximum Number of Items that Can be Returned

Key specification format	Sort key length	Remarks
val function	16 bytes	
Text expression	20 bytes	Data is sorted by using the first 20 bytes of data as a key. If a character string exceeding 20 bytes is used to sort data, specify the *rlen* function in key specifications in the sort expression. Doing so enables a string of up to 128 characters to be used as a sort key.
rlen function	Refer to Table D-4, Character Encoding and the Byte Length of Each Character of the Character String Produced by the *rlen* Function, for the length of the character string produced by the *rlen* function.	

The character encoding of the director data determines the byte length of each character of the character string produced by the *rlen* function, as shown below.

Table D-4 Character Encoding and the Byte Length of Each Character of the Character String Produced by the *rlen* Function

Character encoding	Bytes per character	Example
UTF-8	4 bytes	rlen(/root/name/text(),50) would be treated as 200 bytes.
Shift-JIS	2 bytes	rlen(/root/name/text(),50) would be treated as 100 bytes.
EUC	3 bytes	rlen(/root/name/text(),50) would be treated as 150 bytes.

The following table shows how the total sort key length affects the number of records that Shunsaku can return.

Table D-5 Relationship between the Total Sort Key Length and the Maximum Items Returned

Total Sort Key Length (bytes)	Maximum Items Returned
1 to 200	1,000
201 to 300	700
301 to 400	500
401 to 500	400
501 to 700	300
701 to 1,000	200
1,001 to 2,000	100

Aggregation Requests

The following table lists the allowable values associated with aggregation requests.

Table D-6 Allowable Values for Aggregation Requests

Item	Allowable value	Remarks
Maximum number of keys that can be specified	8	
Sort expression	From 1 to 65, 535 bytes (excluding \0)	
Total group key length	1 to 2,000 bytes	The total group key length is the total combined length of all the group keys in key specifications. Refer to Table D-9, Relationship between the Group Key Length and the Maximum Number of Items that Can be Returned, for more information on the total length of all the group keys in key specifications.
Number of characters when the *rlen* function is specified in key specifications	1 to 128	Refer to Table D-4, Character Encoding and the Byte Length of Each Character of the Character String Produced by the *rlen* Function, for the length of the character string produced by the *rlen* function.
Number of items that can be returned	100 to 1,000 records	When the total length of group keys is 200 bytes or less, the records corresponding to the top 1,000 items can be returned. When the total length of group keys exceeds 200 bytes, the number of groups that can be returned is indicated in Table D-9, Relationship between the Group Key Length and the Maximum Number of Items that Can be Returned.

Relationship between the Total Group Key Length and the Maximum Number of Items that Can be Returned

The total group key length is equal to the total combined length of all group keys used in key specifications. The group key length differs according to the key specification format, as shown below.

Table D-7 Relationship between the Total Group Key Length and the Maximum Number of Items that Can be Returned

Key specification format	Group key length	Remarks
val function	16 bytes	
Text expression	20 bytes	Data is grouped by using the first 20 bytes of data as a key. If a character string exceeding 20 bytes is used to group data, specify the *rlen* function in key specifications in the sort expression. Doing so enables a string of up to 128 characters to be used as a group key.
rlen function	Refer to Table D-4, Character Encoding and the Byte Length of Each Character of the Character String Produced by the *rlen* Function, for the length of the character string produced by the *rlen* function.	

The character encoding of the director data determines the byte length of each character of the character string produced by the *rlen* function, as shown below.

Table D-8 Encoding and the Byte Length of Each Character of the Character String Produced by the *rlen* Function

Character Encoding	Bytes per character	Example
UTF-8	4 bytes	rlen(/root/name/text(),50) would be treated as 200 bytes.
Shift-JIS	2 bytes	rlen(/root/name/text(),50) would be treated as 100 bytes.
EUC	3 bytes	rlen(/root/name/text(),50) would be treated as 150 bytes.

The following table shows how the total group key length affects the number of groups that Shunsaku can return.

Table D-9 Relationship between the Group Key Length and the Maximum Number of Items that Can be Returned

Total Group Key Length (bytes)	Maximum Groups Returned
1 to 200	1,000
201 to 300	700
301 to 400	500
401 to 500	400
501 to 700	300
701 to 1,000	200
1,001 to 2,000	100

Appendix E

Estimating Resources

This appendix explains how to estimate resources when using an application to add, delete or search for data. Use to obtain an approximate estimate of the amount of memory used by the Shunsaku APIs when the application runs

- Local Memory Requirements for Java APIs
- Local Memory Requirements for C APIs

Local Memory Requirements for Java APIs

Java APIs use a memory area known as the heap, which is managed entirely by the JavaVM. The default size of the heap is 64 MB. To change the size of the heap, execute *java* with the '-Xmx' option specified as below.

Example

To set the maximum size of the heap area to 128 MB:

```
java –Xmx128m name of class to be executed
```

Notes

- Specify the '-verbose:gc' option with the *java* command.

  Refer to SDK Tool in the Java 2 SDK, Standard Edition Document for more information on the *java* command.

- Use the *jheap* command accompanied with Interstage Application Server V6.0L10 or later.

  Refer to the Software Release Note - Java heap monitoring tool (jheap) accompanied with Interstage Application Server V6.0L10 or later for more information on *jheap*.

Local Memory Requirements for C APIs

The formula for estimating local memory is shown below.

Definition unit	Variable elements	Size (bytes)
Send/Receive area	Standard record size The number of APIs executed concurrently	Standard record size × the number of APIs executed concurrently

Example

An example calculation for local memory used by APIs is shown below.

If the variable elements are as follows:

- The standard record size: 4,000 bytes
- The number of APIs executed concurrently: 100

```
4,000 × 100 = 400,000(approx.  400 KB)
```

Note

- Local memory allocated by applications is required in addition to the above.

Appendix F

Notes on XML Documents

This appendix provides notes on the following for XML documents in Shunsaku:

- XML Document Format
- XML Documents in Text Files
- Notes on XML Format

XML Document Format

XML documents in Shunsaku must be well-formed.

It is also possible to store only the XML document body text, without the XML declarations, DTDs, etc.

XML Documents in Text Files

Multiple XML documents can be imported directly to Shunsaku from text files. This can be done in a single operation if all of the XML documents are stored sequentially in a single file.

Shunsaku stores multiple XML documents in a single file with each separated as one record (or one document unit).

Note

- Any character strings or symbols following the root end tag for a document are treated as the next XML document.

Document A

```
<!-- Starting A -->
<A>
    <B>aaam</B>
</A>
<!-- End A -->
```

Document B

```
<!-- Starting B -->
<A>
    <C>aaam</C>
</A>
<!-- End B -->
```

If the two documents above are stored as a single file and then imported, the root end tag '</A>' will be treated as the end of document A. When the entire document B is retrieved, it will contain the text '<!-- End A -->' from document A.

Retrieved Document A

```
<!-- Starting A -->
<A>
    <B>aaam</B>
</A>
```

Retrieved Document B

```
<!-- End A -->
<!-- Starting B -->
<A>
    <C>aaam</C>
</A>
```

Notes on XML Format

The following notes apply to the XML format of XML documents stored in Shunsaku:

- Preliminary sections such as XML declarations and DTDs cannot be specified as search targets.
- The values of attributes or namespaces cannot be specified as search targets.
- Shunsaku does not check the syntax of XML documents, so it is necessary to ensure that syntactically correct XML documents are specified.

Glossary

Application server

A server where the applications used to operate Shunsaku are located.

When the Shunsaku APIs are installed, applications can use the APIs to utilize the search and update functions of Shunsaku.

attributeRule

A tag that defines attributes in the data in XML format. This is defined in a mapping rule file.

columnRule

A tag that defines items in the data in XML format. This is defined in a mapping rule file.

Conductor

A module that receives and responds to search requests.

Conductor control information

Information that is used together with record identifiers to uniquely identify records stored in Shunsaku.

Conductor environment file

A file that defines the application environment used to operate a conductor.

Conductor identifier

A name used to identify the application environment for each conductor.

Conductor log file

A file used to maintain the application log of a conductor.

Connection

Connection between the application and Shunsaku.

Default mapping rule

Mapping rules that are used when no mapping rules are specified.

Degradation

A function that enables search and update processing to continue by redistributing search data from searchers when abnormalities occur to the remaining searchers.

Director

A module that receives and responds to search requests.

Director data

Data that has been converted from data in XML format to a format that can be managed within Shunsaku.

Director data file

A general term used to refer collectively to both director data management files and director data storage files.

Director data management file

A general term for files used to manage director data storage files.

Director data management files and director data storage files are referred to collectively as “director data files”.

Director data storage file

A general term for files used to store director data.

Director data storage files and director data management files are referred to collectively as “director data files”.

Director environment file

A file that defines the application environment used to operate a director.

Director identifier

A name used to identify the application environment for each director.

Director log file

A file used to maintain the application log of a director.

Director server

A server where a conductor, directors and a sorter are located. It functions as a window for receiving search and update requests from applications.

documentFountain

An interface for obtaining data in XML format.

documentRule

A tag that defines an XML declaration in the data in XML format. this is defined in a mapping rule file.

DOM (Document Object Model)

A specification that defines an object model and a platform-independent and language-independent interface for data in XML format. This converts data in XML format into a tree structure.

Escape character

The escape character '\' is placed in front of the following characters in order to specify them as part of the search string:

Charater	Description
.	full stop
\	forward slash
“	double quote
‘	single quote

extensionRule

A tag that defines additional information in the data in XML format. This is defined in a mapping rule file.

Filter

A portion of a search expression that specifies certain conditions to retrieve the desired XML data elements.

Fragmentation

Refers to file fragmentation.

Fragmentation refers to the state in which data that should exist as a single block exists as separate fragments.

Shunsaku eliminates fragmentation by optimizing director data files.

Fragmentation rate

Refers to the degree of file fragmentation.

In Shunsaku, the fragmentation rate is determined by the number of stored records, the number of deleted records and the number of updated records in director data files.

A high fragmentation rate means a large amount of wasted disk space.

Grouping

The priority of search processes can be determined by using parentheses “()” in the search expression.

JDBC (Java Database Connectivity)

APIs for accessing relational databases from Java programs.

JDBC driver

A driver for accessing databases via JDBC.

Local memory

The memory area used by a process.

Mapping rule

A rule that defines the correspondence between the input data and the output data in XML format.

Mapping rule file

A file that stores mapping rules. This file is used as input to the XMLGenerator.

MappingRule

A tag that defines the highest order tag in the data in XML format. This is defined in a mapping rule file.

Node

Each junction on the tree structure data in XML format. It represents an element or its content.

parentRule

A tag that defines parent elements in the data in XML format. This is defined in a mapping rule file.

Path

A specification used to indicate nodes targeted for retrieval or nodes to which filtering conditions are to be applied.

Path expression

A specification used to indicate specific nodes or groups of nodes within an XML node tree. It specifies nodes to which filtering conditions are to be applied.

Record identifier

A unique identifier that can be used to identify each Shunsaku record.

ResultSet

An object for storing the results of an SQL query.

Return expression

An expression that represents the data extraction format for extracting particular elements from data that meets certain search criteria or for aggregating these elements.

SAX (Simple API for XML)

APIs used when applications use an XML processor to verify and analyze XML documents. Each line in the XML document is read sequentially starting from the beginning of the document, and the appropriate processing procedures are called as each element is processed.

Search data

Lookup data distributed from a director to a searcher.

Search data is loaded into the memory of a search server.

Search expression

An expression that represents criteria used to search for XML documents that satisfy specified conditions.

Search server

A server where a searcher is located.

Searcher

A module that executes searches.

Searcher environment file

A file that defines the application environment used to operate a searcher.

Searcher identifier

A name used to identify the application environments for each searcher.

Searcher log file

A file used to maintain the application log of a searcher.

Shared memory

An area of memory that can be referenced by different processes.

Sort expression

An expression that represents keys for sorting or aggregating search results.

Sorter

A module that sorts or aggregates search results.

Sorter environment file

A file that defines the application environment used to operate a sorter.

Sorter identifier

A name used to identify the application environments for each sorter.

Sorter log file

A file used to maintain the application log of a sorter.

SQL (Structured Query Language)

An international standard structured query language that stipulates the syntax used to access relational databases.

Well-formed XML document

A document that satisfies the minimum standards needed to conform to the specifications of an XML document, namely:

- It has only one root element.
- If it has a starting tag, it also has an ending tag.
- Tag nesting is correctly encoded.

XML processor

A program that inputs an XML file and parses it so that it can be processed by an application.