Programmer’s Guide

softbool

SOFTWARE CORPORATION

www.softbool.com

Chapter 1
Application development

This chapter describes the architecture of applications that use Boolware Index server. In Boolware there are several different API:s for developers: the Functional API for C++ and Delphi programmers, the XML API when communicating via a HTTP/XML protocol and finally the COM object when using Microsoft tools such as for example ASP or Visual Basic.

Topics treated are: definition of client and server concepts and how application developers shall use the Boolware server.

Who should read this book

This book is intended for application programmers, who need a deeper understanding how Boolware can be manipulated: through a C++, Delphi application, via a HTTP/XML protocol, via PHP, JAVA or through COM.

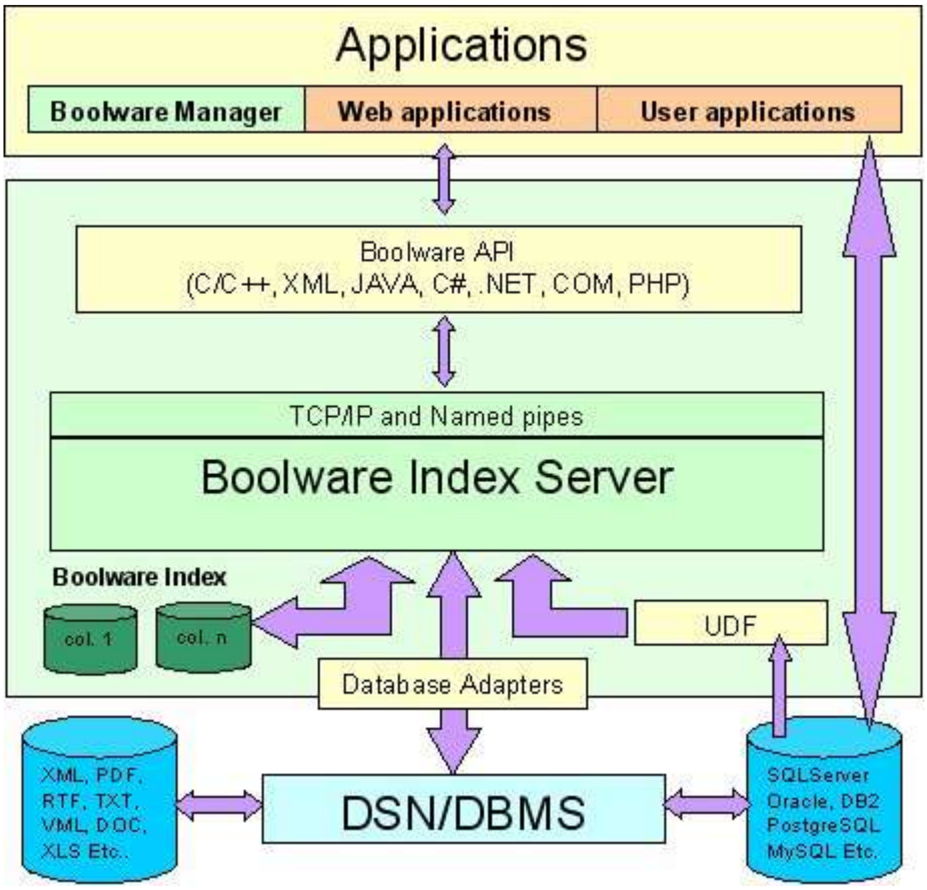
Definitions

Definitions used in this chapter:

Term	**Description**
Boolware server	Software that is installed on a central server computer.
Boolware client	Software used by applications to communicate with Boolware server.
Boolware Index	All index files for a specific data source.
Table	Correlates to a table in the data source
Column	Correlates to a column/field in a table in the data source
Term, indexterm search term .	A searchable word in a Boolware Index
Primary key	One or more columns that uniquely identifies a single row in a table.

Boolware overview

This is a conceptual view of Boolware. The picture intends to illustrate how Boolware – through various APIs – connects with applications, XML scripts and Boolware Manager. The picture also shows how Boolware communicates with different data sources and the Index files.



The data source connections are made through adapter plugins. Each adapter is specially designed for its respective data source. A general interface for many data sources is ODBC. But since ODBC isn’t well supported on all platforms, any adapter can be coded to use a native data link, which can be seen in the picture.

Refer to chapter 11 of the Operations Guide for detailed descriptions of vendor specific adapters and their particular characteristics.

Definition of a client

A Boolware client is any application process that uses the Boolware client library, directly or via a middleware interface, to establish a communication channel to a Boolware server. The connection can be local if the application executes on the same node as Boolware server, or *remote* if the application must use a network to connect to the Boolware server.

The Boolware client library

The Boolware client library is a library that developers of applications use to initiate connections to a server and to programmatically perform database operations. The library uses the TCP/IP protocol to communicate with one or more servers, and implements a special Boolware client/server application protocol on top of a network protocol.

The client library provides a set of high-level functions as an Application Programmers Interface (API) for communication with a Boolware server. Any client application or middleware must use the API to access a Boolware Index.

Definition of a server

The Boolware server is a software process that executes on the node that hosts the storage space for Indexes. The server process is the only process on any node that can perform direct I/O to the Index files.

Clients send to the server process requests to perform several different types of actions on the Inverted files, such as:

- Search the Index based on criteria
- Rank the resulting data
- Return sets of data
- Inspect the list of available databases, tables and columns
- View searchable index terms

The server process is fully network-enabled; it services connection requests that originate from another computer. The server process implements the same Boolware API that the client uses. Many clients can remain connected to the multi-threaded server process simultaneously.

Application development

Once you create and populate Boolware Inverted files, you can access the information through an application. You can design and implement a new application by embedding SQL statements and Boolware API calls in an application written in programming languages such as Visual Basic, C or C++.

Step by step introduction

This is an overview of the Boolware API, what functions that are available and how to use them.

Chapter 2 **API description**, contains a detailed description of each function.

The API enables easy access to powerful searches for applications written for Microsoft Windows.

All functions return an integer status code that should be checked. A negative value (less than **SOFTBOOL_OK**) should be regarded as an **error**, while positive values (greater than **SOFTBOOL_OK**) should be treated as **information**.

These are six easy steps to communicating with Boolware server:

- Step 1 – connect with **Boolware** and establish a session
- Step 2 – list all available **Boolware Indexes**
- Step 3 – attach to one **Boolware Index**
- Step 4 – list all tables and columns within a **Boolware Index**

- Step 5 – query the **Boolware Index**
- Step 6 – end the session.

Note: In the following examples, there are intentionally no tests for return codes to increase readability.

Use the API functions **BCGetErrorCode(), BCGetErrorMsg()** to retrieve error codes and messages.

Please note that all orderly integers used are zero-based, e.g. the first element is at index zero.

All list boxes and combo boxes used in these examples should be defined by the application.

Step 1

The first thing to accomplish is to create a Boolware client and establish a connection with Boolware server.

A Boolware client is an instance of an internal handle that provide thread safety and handle all communication with Boolware Server. This handle should be saved and applied to all Boolware client functions. This call should only be done once and the returned handle should be used throughout the entire session.

In this small tutorial we assumed that there is a global client instance with the name mClient that have been created once and for all. This client instance should be freed at the end of the session with a call to the function **BCFreeClient()** .

```
mClient = BCCreateClient();
```

Boolware server must be running on a computer that is accessible somewhere on the network. Use the server computers network name or IP address to identify it, for example 192.168.0.1.

```
rc = BCConnect(mClient, "Charlie", "");
```
or
```
rc = BCConnect(mClient, "192.168.0.1", "");
```

If a connection was established (rc is zero), the application can now use the Server.

The above example call to **BCConnect()** will create a session named "MySessID". The session name is the unique identification of a session.

Boolware will generate a unique session name if no name is passed. The session name can be retrieved by the app using the **BCGetSessionInfo()** API.

The session name can be used to reconnect to an existing session. This comes in handy when for example coding Web applications, since http 1.0 by default connects and disconnect for each request.

If you pass in your own session name, make sure that it doesn’t conflict with other sessions.

This would most likely not be what you intend.

Example:

```
mClient = BCCreateClient();
rc = BCConnect(mClient, "Charlie ", "MySessID")
```

When disconnecting from the server, you have the option of closing the session (logging out) or not. A session can be reconnected to (using BCConnect()) provided that you do not close it when disconnecting.

Example:

Disconnect but do not close session:

```
BCDisconnect(mClient, false);
```

Reconnect to Boolware Server and the previously established session:

```
BCConnect(mClient, "Charlie", "MySessID");
```

Boolware Server manages all alive sessions. It will automatically close any session that has been idle for too long. The default time that a session can be idle before it is closed is 1200 seconds (20 minutes), but this can be changed using the Manager.

The default TCP/IP port to be used is 7008. If the server is setup to use a different port, this port can be specified in the call to **BCConnect()**.

The port number can be specified as a suffix to the IP address, separated by a colon. For example: "192.168.0.1:2000" will try to connect using port 2000. Please note that Boolware must be configured (using the Manager) to listen to this port.

Step 2

When a successful connection has been made, an application can list which databases are available.

Example:
List all available databases and save for later use.

```
int      rc, num;
char   name[128];
BCDatabaseInfo_t  info;

// Get number of available databases
if (BCGetNumberDatabases(mClient, &num) != SOFTBOOL_OK)
   return HandleError();

// Get information about each database
for(i = 0; i < num; i++)
   {
   // Get info about this database
   BCGetDatabaseInfo(mClient, i, &info);

   // Save database identities in a list
   databaseList->Add(info.dsnName);
   }
```

Step 3

Choose one of the available indexes by attaching to it using its dsn-name.
The database DSN names were fetched using **BCGetDatabaseInfo()** in step 2.

Example:

Attach to the dsn named "Company".

```
rc = BCAttach(mClient, "Company");
```

Step 4

When a database has been successfully attached to, it can be further investigated. For example, we can list the tables and columns from that database.

Column information guides the application how it is indexed, etc.

See the structures **BCTableInfo_t** for information about a table, and **BCColumnInfo_t** for info about columns.

Example:
Get number of tables in the "Company" database. Save table names in a list for later use, and fetch info on all columns from all tables.

```
int numTab, numCol, i, j;
BCTableInfo_t tab;
BCColumnInfo_t col;

// Get number of tables
BCGetNumberTables(mClient, &numTab);

// Get info on each table
for(i = 0; i < numTab; i++)
   {
   // Get info about this table
   BCGetTableInfo(mClient, i, &tab);

   // Save table name in a list
   tabList->Add(tab.tabName);

   // Get number of columns in this table
   BCGetNumberColumns(mClient, tab.tabName, &numCol);

   // Get info on each column
   for(j = 0; j < numCol; j++)
      {
      // Get info about this column
      BCGetColumnInfo(mClient, tab.tabName, j, &col);

      // Save column in a list
      colList->Add(col.colName);
      }
   }
```

Step 5

This step describes how to search and retrieve data from Boolware.

The database used in this example is "Company". This database has a table called "Employee".

That table has columns such as: "LastName", "FirstName, "Address" "City", "Zip" etc. The primary key of this table is "ID" (which we found out using the **BCGetColumnInfo(**) API, examining the ***flags*** field from the **BCColumnInfo_t** record. The content in this column makes it possible to retrieve the tuple from the data source.

Example:

In this example we are looking for persons with a first name beginning with "An". For example "Ann", "Andreas", "Anthony" etc., residing in the City of London.

For more information regarding the query language, see "**Softbool Query Language**".

```
int     result, recNo rankMode, i, j;
float   score;
char   key [128], tabName [128];
BCRowData_t  rowData;

// Do the search
BCQuery(mClient, "Employee", "FIND FNAME:An* AND
            City:LONDON", &result, NULL);
// Fetch primary keys for all matching records
for(i = 0; i < result; i++)
   {
   // Get primary key
   BCFetchKey(mClient, "Employee", "ID", i, key, sizeof(key),
              &score, &recNo, rankMode);
   // Save primary key in list
   pkList->Add(key);
   }
```

You can also let Boolware fetch the rows directly from the data source.

```
// Get the 50 first chars from FNAME and CITY
for(i = 0; i < result; i++)
  {
  // Get one row
  BCFetchRow(mClient, "Employee", "FNAME, CITY",
         i, 50, &rowData);

  // Get each column from the row
  for(j = 0; j < rowData.count; j++)
    {
    // Save name of each found column
    colList->Add(rowData.cols->name);

    // Save column value
    colList->Add(rowData.cols->value);
    }
```

Step 6

Disconnect from Boolware and close the session (log out).
If the session isn't closed by the application, Boolware will automatically close it after a certain time of inactivity.

Example:
Disconnect and logout.

```
BCDisconnect(mClient, true);
BCFreeClient(mClient);
```

Sessions

Overview

When you connect to a Boolware server with one of the available ***connect*** methods, a session object will automatically be created on the server side. The session object contains among other things, information about the current search and the result. All sessions in Boolware must have an ID associated with them. Session ID can be up to max 128 characters long. When you connect to a Boolware server you can optionally specify a session ID. If you don't specify a session ID, the Boolware server will automatically generate one and associate it with the new session.

After the function call to the ***connect*** method the session object and its ID will automatically be used for all subsequent function calls to the Boolware API until the ***disconnect*** method is called.

Because a session object is created immediately when you connect to a Boolware server the XML attributes ***name*** and ***server*** in XML element ***open_session*** in a Boolware XML request, are only relevant when using the Boolware ISAPI or Apache client (where you do not do any explicit function call to a connect function).

Lifetime

A session is available until you explicitly call the ***disconnect*** method with the parameter ***logout=true*** or until the session has reach it's timeout value. The default timeout value for sessions is 300 seconds, but that can be changed with the Boolware manager or programmatically by changing the ***sessiontimeout*** setting. If a session reaches its timeout value, the Boolware server will automatically delete the session object and close the connection (socket) on the server side.

If you call the ***disconnect*** method with the parameter ***logout=false*** the session will not be deleted from the Boolware server, only the connection (socket) will be closed. You can later use the method ***reconnectIfExists*** to try to connect to the same session. Note however that the session may have reached its timeout value and therefore been deleted from the Boolware server.

Management

You can see all active sessions in a Boolware server with the Boolware manager if you click on the ***Sessions*** tab. Here you also can logout sessions (forced logout).

Softbool Query Language

The Softbool Query Language (QL) is a simple version of the Common Command Language (CCL) adjusted for Boolware. The main purpose is to be able to specify both simple and complex queries in a uniform and simple way.

Refer to **Operations Guide** for detailed information about the Softbool query language.

Execute commands in Boolware

Commands can be sent as text to Boolware, using the BCConnectExecute and BCExecute functions, using XML / JSON (execute-element/property) calls. This section describes the commands that are available.

EXPORTRESULT

This command is used to export a result to a file. You can select parts of the result set and which fields you want to export.

NOTE! This command requires a database to be selected.

exportresult **table**="<*table name*>"
field="<*field name1>[,<field name2>...etc.]*"
outfile="<*file name*>"
[**rowsep**="<*row separator*>"]
[**fieldsep**="<*field separator*>"]
[**quote**="<*character surrounding text*>"]
[**sort**="<*sort order*>"]
[**from**="<*from row*>"]
[**count**="<*number of rows*>"]
[**maxchars**=<number of chars>]
[**columnnames**="<*yes/no*>"]
[**replacecolumnnames**="<*field name1>[, <field name2>...etc.]*"]
[**randomfetch**="<yes/no>"]
[**encoding**="<*iso-8859-1/utf-8/xls/xlsx*>"]
[**append**="<*yes/no*>"]
[**excludesubfields**="<*yes/no*>"]

Mandatory parameters:

table	table name
field	name of the fields (comma separated) to be exported
outfile	full name of the file that will contain the result

Optional parameters:

rowsep	row separator. The default value is CRLF
fieldsep	field separator. The default value is TAB
quote	character surrounding text in the fields, allowed characters are quote or apostrophe. Default value is empty value.
sort	which fields (comma separated) to be sorted on and sort order, *asc* or *desc*. If the parameter is omitted or is empty, no sorting will take place.
from	number of the item in the result to start from. Default value is '1'
count	number of items to be exported. The entire result is used if the parameter is omitted.
maxchars	can be set to a value greater than '0' indicating the maximum number of characters to be exported from each field. The default value is '0' which means that all data in the field will be exported.
columnnames	The value '*no*' or omitted parameter means that no field names are added to the empty out file. The value '*yes*' means that field names are added to the first line in the out file.
replacecolumnnames	field names (comma separated) that replaces the data source field names on the first line (if the *columnnames* parameter is enabled). The field names of the data source is used if the parameter is omitted.

randomfetch	The value 'yes' will store rows randomly in the out file regardless of any sort expression. Value 'no' or omitted parameter will not store rows in random order.
encoding	export format. Valid values are '*iso-8859-1*', '*utf-8*', '*xls*' or '*xlsx*'. If the parameter is omitted, the value with which the session is connected is used. '*xls*' is the old Excel format, and '*xlsx*' is the new format (MS Excel Open XML format). The exported file cannot exceed 2 GB and cannot contain more than 65,535 identical strings if "*exportencoding*" is set to 'xls'.
append	The value '*no*' or omittted means start from the beginning of the file. The value '*yes*' means the export continues at the end of the existing file. The parameter is **not** used when encoding is '*xls*' or '*xlsx*'. Default value is '*no*'.
excludesubfields	The value '*yes*' prevents XML subfields from being exported. The value '*no*' allows XML subfields to be exported. If the parameter is omitted, XML subfields are allowed to be exported

Example:

```
exportresult table="tab1"
             field="field1, field2, field3"
             outfile="c:\export.txt"
             sort="Turnover desc"
             quote=""""
```

All records from the resultset will be exported to the file **export.txt** in the **c:** directory. Before the records are exported, they will be sorted descending on **Turnover**. Only the fields: **field1**, **field2** and **field3** from the table **tab1** will be exported and the fields will be separated by a TAB and each record will start on a new line (CRLF). The text in each field will be enclosed in **quotation marks** ("); if the text in any field contains quotation marks, this will be represented by two quotation marks.

```
exportresult table="tab1" field="field1, field2, field3"
             outfile="c:\export.txt"
             sort="Turnover desc"
             quote="'"
             append="yes"
             columnnames="yes"
             replacecolumnnames="Name, Address, Town"
             fieldsep=";"
```

All records from the resultset will be exported to the external file **export.txt** in the c: directory. The records will be appended to the previous result in the export.txt file. If the file is empty the first line will be a header line containing the specified field names: **Name**, **Address** and **Town**. Before the records will be written to the file they will be sorted on *Turnover descending*. The only fields that will be exported are: **field1**, **field2** and **field3** from table **tab1** and the fields will be separated by **semicolon** (;) and each record will start on a new line. The text of each field will be enclosed within **apostrophes** (‘). Note that if the text contains an apostrophe it will be represented by two apostrophes.

FETCHCOORDINATES

With this command you can fetch coordinates for current result.
If having group data configured, the group data will be grouped on coordinate position and in the current result order. You cannot fetch more coordinates than the current search result.

NOTE! This command requires a database to be selected.

fetchcoordinates **table**="<*table name*>"
latitude="<*latitude*>"
longitude="<*longitude*>"
[**from**="<*from row*>"]
[**count**="<*number of rows*>"]

[**rowsep**="<*row separator*>"]
[**fieldsep**="<*field separator*>"]
[**groupdatafieldsep**="<*data field separator*>"]

Mandatory parameters:

table	table name
latitude	the field name of the field that contains latitude data
longitude	the field name of the field that contains longitude data

Optional parameters:

from	start value from where to start the fetch. Default value is 1.
count	how many lines you want to retrieve. Default value is the entire result.
rowsep	row separator. The default value is CRLF
fieldsep	field separator; The default value is TAB
groupdatafieldsep	separator between grouped coordinate data. Default value is |*|

Example:
Fetch 50 coordinates fom table "Person" where latitude field name is "Coord_Y" and longitude field name is "Coord_X".

```
fetchcoordinates table="Person" latitude="Coord_Y" longitude="Coord_X"
                 from="1" count="50"
```

FETCHNOTFOUND

This command is used to fetch the terms that was not found in the Boolware index at the last *orsearchex* sub-command. The terms are valid for the current table and user. The terms will be deleted by the next FIND command. You could browse through the terms by specifying a start value and number of terms you want to fetch. The maximum number of terms that could be fetched by one command is 50.000. Each fetched term is ended by a CR/LF (new line).

NOTE! This command requires a database to be selected.

fetchnotfound table="<*table name*>"
[**from**="<*from term*>"]
[**count**="<*number of terms*>"]

Mandatory parameters:

table	the current table where the search was performed.

Optional parameters:

from	the term you want to start from (used when browsing). The first term has a starting value of 1.
count	specifies how many terms you want to retrieve (maximum number in a command is 50,000). The default value is all terms.

If only the first parameter *table* is specified, the answer contains the total number of terms that were not found in the index at the previous *orsearchex*.

Example:
After a big orsearchex search on 700.000 DUNS numbers in the table Companies there were 87.953 DUNS numbers that were not found in the Boolware index. These DUNS numbers have been stored in a file. By using the **fetchnotfound** you could fetch required DUNS numbers not found in the Boolware index.

```
fetchnotfound table="Companies"
```

The response is:
87953 is the total number of ‘not found’ terms.

```
fetchnotfound table="Companies" from="1" count="500"
```

The response is the 500 first DUNS numbers that was not found in the Boolware index.

FETCHPOLYGONSFOUND

The command is used to fetch extended information on which polygons that contained the given coordinate in the sub-command *geowithinpoygon*.

NOTE! This command requires a database to be selected.

fetchpolygonsfound table="<*table name*>" **field**="<*field name*>"

Mandatory parameters:

table	the current table where the search was performed
field	the name of the field where the search was performed

After a query with the sub-command *geowithinpolygon* that have a result, the command **fetchpolygonsfound** can be executed to obtain more information about which polygons that contains the given coordinate.

Example:
```
fetchpolygonsfound table="mapshapes" field="shape"
```

This could generate the following result of primary keys and which polygon within the field "shape" that contains the given coordinate. The WKT/GeoJSON format permit multiple polygons to be stored in the field and are enumerated from 1.

```
PK="12345" polygons="1"
PK="23451" polygons="2"
PK="34521" polygons="1,3"
```

FLOWINFO

This command lists all flows for one or more specified database names.

flowinfo database="<*database name1>[,<database name2>]*" *[***xml**="<*yes/no*>"*]*

Mandatory parameters:

database	one or more database names (comma separated). Enter the asterisk (*) for all databases that have flows.

Optional parameters:

xml	The value "*yes*" means that the result will be presented as xml. The value "*no*" means that the result will be presented in text format. Default value is '*yes*'.

In the following example there are four databases:DB1, DB2, DB3 and DB4. DB1 has two flows: Search and match. DB2 also has two flows: list and match. DB4 has one flow: lookup. DB3 has no flows at all.

Example1:
List all flows for the data bases DB1, DB2 and DB3:

```
flowinfo database="DB1,DB2,DB3"
```

The result from this command is:

```
<flowinfo>
 <database name="DB1">
  <flow name="Search"/>
  <flow name="match"/>
 </database>
 <database name="DB2">
  <flow name="list"/>
  <flow name="match"/>
 </database>
</flowinfo>
```

Example2:
List all flows in all databases that contain flows:

```
flowinfo database="*"
```

The result from this command is:

```
<flowinfo>
 <database name="DB1">
  <flow name="Search"/>
  <flow name="match"/>
 </database>
<database name="DB2">
  <flow name="list"/>
  <flow name="match"/>
 </database>
<database name="DB4">
  <flow name="lookup"/>
</database>
</flowinfo>
```

FLOWINFOPARAMETERS

This command lists flow variables for one or more specified database names and flows.

flowinfoparameters **database**="*<database name1>[,<database name2>]*"
flow="*<flow1>[,<flow2>]*"
*[***xml**="*<yes/no>*"*]*

Mandatory parameters:

database	one or more database names (comma separated).
flow	the name of the flows to be listed (comma separated)

Optional parameters:

xml	The value "*yes*" means that the result will be presented as xml. The value "*no*" means that the result will be presented in text format. Default value is '*yes*'.

In a flow you could specify a parameter section that could be listed by applications.
These have the element name <parameter> and should appear within the <flow_input> element which should be directly under the <flow> element.

Example:

```
<flow>
 <flow_input>
  <parameter name="Name" type="data" defaultvalue="" description=""/>
  <parameter name="Address" type="data" defaultvalue="" description=""/>
 </flow_input>
</flow>
```

These elements could be fetched by an application to get information on special indata elements from one or more specified flows from one or more specified databases.

In the following example there are four databases:DB1, DB2, DB3 and DB4. DB1 has two flows: Search and match. DB2 also has two flows: list and match. DB4 has one flow: lookup. DB3 has no flows at all.

Example:
Fetch flow parameters from the flows: "Search" and "match" from the databases: "DB1", "DB2" and "DB4".

flowinfoparameters database="*DB1,DB2,DB4*" **flow**="*Search,match*"

```
<flowinfo>
 <database name="DB1">
  <flow name="Search">
   <flow_input>
    <parameter name="Name" type="data" defaultvalue="" description=""/>
    <parameter name="Address" type="data" defaultvalue="" description=""/>
   </flow_input>
  </flow>
  <flow name="match">
   <flow_input>
    <parameter  name="who" type="data" defaultvalue="" description=""/>
    <parameter  name="where" type="data" defaultvalue="" description=""/>
   </flow_input>
  </flow>
 </database>
 <database name="DB2">
  <flow name="match">
   <flow_input>
    <parameter  name="who" type="data" defaultvalue="" description=""/>
    <parameter  name="where" type="data" defaultvalue="" description=""/>
   </flow_input>
  </flow>
 </database>
</flowinfo>
```

GETAUTOTRUNC

Displays whether automatic truncation is enabled or not (*yes/no*) for the session. Default value is '*no*'.

getautotrunc

GETDUPRULES

Displays which duplicate rules are available for a given table.

NOTE! This command requires a database to be selected.

getduprules table="<*table name*>"

The response is a list of duplicate rule names available for the table, one per row (CRLF)

GETENCODING

Displays the current character encoding (iso-8859-1/utf-8) for the session. The default value is iso-8859-1.

getencoding

GETEXITPOINT

Displays the last used exit point in a search flow for the session. Default value is empty string.

getexitpoint

GETHISTORY

Displays whether search history is enabled or not (*yes/no*) for the session. Default value is '*no*'

gethistory

GETINDEXEXIT

Displays whether terms created by custom indexing are used in search or not (*yes/no*) for the session.

getindexexit

If the value is '*no'*, only search terms created by Boolware will be used in the search. If the value is '*yes*', search terms created by the customized indexing will also be used in the search. The default value is '*yes*'.

GETMAXEXECUTIONTIME

Displays the current maximum execution time for the session in seconds.

getmaxexecutiontime

GETQUERYLIMVALUE

Displays the limit value for the number of hits per term for the session when using the querylim subcommand.

getquerylimvalue

If the term will be found in more records than this value it will be handled as stop word. NOTE This value could temporarily be overridden when performing the querylim sub-command.

GETSESSIONTIMEOUT

Displays the current "session timeout" for the session in seconds

getsessiontimeout

GETSETSEARCH

Displays whether set search is enabled or not (*yes/no*) for the session.

getsetsearch

GETSTRICTASIS

Displays whether wildcards are handled by the wordasis and stringasis (*yes/no*) subcommands for the session.

getstrictasis

HITLISTPOSITION

Retrieve the position of a specifically specified record in the current hit list. This command requires that the requested record be identified by a primary key in the specified table. If the primary key consists of several fields, all fields must be specified.

NOTE! This command requires a database to be selected.

```
hitlistposition table="<table name>"
                <pk_field1>="<pk_value1>"
                 [<pk_ field2>="<pk_value 2>"]
                 [<pk_ field3>="<pk_value 3>"]
                ...etc.
```

Mandatory parameters:

table	table name
<*pk_field1*>	First primary key field with its value

Optional parameters:

<*pk_field2*>	The second primary key field with its value if the primary key consists of several fields. *…etc.*

Returns the current position of the specified record in the current hit list.

Example:
Retrieve the position of a particular record in the current hit list. The table is "Companies" and the primary key consists of two fields: "CompanyID" and "CompanyType". The identification of the record is: CompanyID = 12345 and CompanyTyp = AB.
To retrieve the position of this record if it is in the current hit list, enter the following:

```
hitlistposition table="Companies" CompanyID="12345" CompanyType="AB"
```

INDEXEX

This command is used to list index terms from a Boolware Index.

The response from this command is the searchable terms that reside in the specified field in the specified table.

NOTE! This command requires a database to be selected.

indexex **table**="<*table name*>"
field="<*field name*>[<*subzoom expression*>]"
[**max_terms**="<*number of terms*>"]
[**start_position**="<*start position*>"]
[**type**="< *type number*>"]
[**zoom**="<*yes/no*>"]
[**zoomresult**="<*name*>"]
[**tothits**="<*yes/no*>"]
[**continuation**="<*yes/no*>"]
[**allixtypeterms**="<*index type>"*]
[**resultixtypeterms**="<*yes/no*>"]
[**ixtypeterms**="<*yes/no*>"]
[**statistics**="<*sum/max/min/avg*>"]
[**order**="<*asc/desc*>"]
[**termnumber**="<*yes/no*>"]
[**freqtype**="<*index type*>"]
[**freqlimits**="<*limit*>"]
[**sepgroups**="<*yes/no*>"]
[**keepzeroterms**="<*yes/no*>"]
[**skipgeneratedterms**="<*yes/no*>"]
[**totdocs**="<*yes/no*>"]
[**selected**="<*yes/no*>"]
[**reportaction**="<*open/close/save*>"]
[**reporttemplatename**="<*report template name*>"]
[**levelformaxrecords**="<*level*>"]
[**maxrecords**="<*maximum number*>"]
[**commandheader**="<*yes/no*>"]

Mandatory parameters:

table	the current table
field	the current field. After the field name you can enter a subzoom expression to display hits spread over one or more other values. See examples of this in "Operation Manual" Chapter 11.

Optional parameters:

max_terms	maximum number of terms to fetch. The default value is 50. The maximum number of terms to fetch in one call is 30,000.
start_position	start position. If the parameter is omitted, the first position is used.
type	Can be an indexing type or function. Default value is 1 (word). Can be any of the following:

1	Index type: Word
2	Index type: String
3	Index type: Stemmed
4	Index type: Phonetic
5	Index type: Left truncation
6	Index type: Numeric
10	Hierarchic presentation ordered by number of hits (grouped index)
11	Index order, each group will be presented (grouped index)
12	Hierarchic presentation in alphabetical order (grouped index)
13	Search terms from search term statistics
14	Frequency; shows the terms in frequency order
15	Index type: Exactly as is Word

16	Index type: Exactly as is String
17	Index type: Case sensitive
20	Index type: Within words
21	Shows all terms that fulfills the rules for similarity in Fuzzy
22	Use term number as start position for Index type Word
23	Use term number as start position for Index type String
24	Use term number as start position for Index type Phonetic
25	Use term number as start position for Index type Stemmed
26	Use term number as start position for Index type Left truncation
27	Use term number as start position for Index type Case sensitive
28	Use term number as start position for Index type Within words
29	Use term number as start positio§n for Index type Exactly as is Word
30	Use term number as start position for Index type Exactly as is String
31	Index type: Within string
32	Use term number as start position for Index type Within string

zoom	lists only terms from a result. Default value is '*no'*. If *field* contains a subzoom expression, *zoom* is activated automatically.
zoomresult	saved result to be used or named scratch (scratch results). If the parameter is omitted, the current result is used.
tothits	gives the total number of records per term from the entire index. Default value is '*no*'.
continuation	continues from the last listed term. Default value is ’*no*’.
allixtypeterms	the total number of terms for all index types. Default value is ’*no*’.
resultixtypeterms	the total number of terms for given index type in the current result. Default value is ’*no*’.
ixtypeterms	the total number of terms for given index type. Default value is ’*no*’.
statistics	statistics on the lowest level when subzoom; sum, max, min and mean/avg. Default value is no statistics.
order	sort order, *asc* or *desc*. Default value is '*asc*'.
termnumber	gives the current order numbers for this term (relative and absolute). Default value is ’*no*’.
freqtype	the index type for terms to be sorted by frequency (*type*=14). See allowed indexing types in the parameter *type* above.
freqlimits	limits when terms sorted by frequency (*type*=14). Enter as: **n**, >[=]**n** or <[=]**n**, where **n** is the frequency.
sepgroups	the terms should be listed in two groups; first all terms in the result (zoomed) then all the other terms. Default value is ’*no*’.
keepzeroterms	when fetching a zoomed index all terms will be listed; the one in the result will be listed with a hitcount while the terms not belonging to the result will be listed with hitcount zero. Default value is ’*no*’.
skipgeneratedterms	skip all strings generated by plugin functions. Default value is ’*no*’.
totdocs	gives the total number of records in the current table. Default value is ’*no*’.
selected	marks all terms that were marked at the latest reportaction="*close".* Default value is ’*no*’.
reportaction	What handling of current "analysis" is desired.
	'*open'* opens the current "analysis" and notes which terms should be marked.
	'close' closes current "analysis" and saves the terms that are marked.
	'*save*' saves the terms that are marked.
reporttemplatename	gives the name of the current report template to be used in a subzoom expression (pre built).
levelformaxrecords	level where max no. of records should be tested when "report"
maxrecords	max no: of records that should be fetched at *levelformaxrecords* when "report"
commandheader	specifies whether to print a header line for the command. Default value is ’*yes*’

Only terms from the requested index type are listed.

You can specify if you want terms from the complete index or if you only want terms that are included in a previous result.

You can also get the total number of terms for all index types, the total number of terms in the complete index for given index type and the total number of terms included in a previous result for given index type.

For each term you get the number of occurrences. Two values can be obtained: total number of occurrences or number of occurrences in a previous result.

You could order the terms ascending or descending alphabetically or on the number of occurrences.

The result of this command consists of two parts: the first line is an overview of the request (omitted if **commandheader**='*no*' is specified) and the following lines contain the generated terms.

The following lines could look a little bit different depending on what index type is listed.

1. " Common" index types:
Word (1), String (2), Stemmed (3), Phonetic (4), Left truncation (5), Numeric (6), Grouped Index Hierarchic frequency order (10), Grouped Index (11), Grouped Index Hierarchic alphabetical order (12), Exactly as is Word (15), Exactly as is String (16), Case sensitive (17), Within words (20) and Term number for all above types (22-30). Each term for the above index types has the following layout:

(n[/N]) Term [(T)], where

n = number of occurrences
N = number of occurrences in the entire index (if zoom="*no*" n = N)
T = order number of the current term

2. "Fuzzy":
Lists all words that meet the rules for fuzzy (21):

(n[/N]) Term [(T)] [(F)], where

n = number of occurrences
N = number of occurrences in the entire index (if zoom="*no*" n = N)
T = order number of the current term
F = similarity percent that the terms are sorted on (100 is exact)

3. Termer sorterade på indextyp
This index type (type="14") sorts the terms in frequency order (in how many records the term occurs). The sort order is specified in *order*. If *order*="desc" the terms that occur in most records will be sorted first. If *order*="asc" the terms that occur in few records will be sorted first. Default value is descending (*order*="desc").

In the attribute *freqlimits* you could specify a condition. The condition could be: >N, <N or =N, where N is the number of records the term must occur in to be approved.

The listing of some special index types is described in Manual "Operations guide", Chapter 11 "Interactive Query" section " Viewing the contents of a Boolware Index".

Example 1:
```
indexex table="Companies" field="Company name"
```

This is the simplest way to specify this command. The only attributes specified are the two mandatory attributes: ‘table’ and ‘field’. Assume that the column "Company name" is indexed in the following way: Word, String and Phonetic.

As no other attributes than table and field are specified the following default values will be used: index type will be set to Word, max_terms is set to 50, start from the beginning of the index and Words from the entire index will be fetched.

The result will be as follows:

```
Table="Companies" Field="Company name" Number of terms="50" Numeric="no"
(1376)  A
(1)       A.A:S
(1)       A.AHLQVIST
(1)       A.BENGTSSON
(1)       A.BORG
(1)       A.CARLSSON
(1)       A.CONTE
(1)       A.DAHLIN
(3)       A.DMAN
(1)       A.EMNEUS
etc.
```

Example 2:

```
indexex table="Companies" field="Company name" type="2"
        resultixtypeterms="yes"
```

In this request you say that you want strings and also the total number of terms (strings). The result can look like this:

```
Table="Companies" Field="Company name" Number of terms="50" Numeric="no" Result index
type terms="464048"
(1)     "CALL US UP" JAN KILSAND TRANSPORT AB
(1)     "COLD STORES" I ESLÖV AB
(1)     "DA CAPO" RESTAURANG AB
(1)     "K" LINE (SWEDEN) AB
(1)     "LÅSET" HENRIKSSON AB
(1)     "RALLY HARRY" BIL AB
(1)     "STURE, LAILA AXELSSON AB"
(1)     "SÄLJPROFIL CHRISTER BR
(1)     ‘ALLO ‘ALLO AB
(1)     ‘TH BYGG’, TORD HANSSON BYGG AB
(1)     A + B SWEDEN AB
(1)     A + G ENLUND AB
etc.
```

Example 3:

```
indexex table="Companies" field="Company name" type="1"
        resultixtypeterms="yes" zoom="yes" tothits="yes" ixtypeterms="yes"
```

This request was preceded by a question, in which all companies in Stockholm were searched. In this request, they say that they want words. Since you only want terms from the latest result (zoom="*yes*"), it is a good idea to request *resultixtypeterms* and *ixtypeterms*.
The result can look like this:

```
Table="Companies" Field="Company name" Number of terms="50" Numeric="no" Result index
type terms="58045" Index type terms="251725"
(220/1376)  A
(1/1)           A.BORG
(2/3)           A.DMAN
```

```
(1/1)      A.HANSER
(1/1)      A.LEKSELL
(1/1)      A.MQ
(1/1)      A.NASRI
(1/2)      A.O:S
(2/5)      A:S
(1/1)      A:SON
(15/68)    AA
(4/5)      AAA
(1/2)      AAAAA
(1/1)      AABC
(1/1)      AABGRUPPEN
(2/4)      AAC
(1/1)      AACG
(1/1)      AAEW
(1/1)      AAF
(3/5)      AAGAARD
(1/3)      AAGESEN
etc.
```

The first line of the result indicates that the total number of terms (Words) in the result is 58,045, while the total number of Words in the entire index is 251,725.

For each term, two numbers are given: the first is mandatory and indicates the number of occurrences within the current result, while the second indicates how many occurrences of the word are in the entire index.

Example 4:

```
indexex table="Companies" field="Company name" type="1"
        resuleixtypeterms="yes" zoom="yes" tothits="yes" ixtypeterms="yes"
        termnumber="yes"
```

In this case, you also request an order number for each term.

```
Table="Companies" Field="Company name" Number of terms="50" Numeric="no" Result index
type terms="58045" Index type terms="251725"
(220/1376) A (1)
(1/1)      A.BORG (2)
(2/3)      A.DMAN (3)
(1/1)      A.HANSER (4)
(1/1)      A.LEKSELL (5)
(1/1)      A.MQ (6)
(1/1)      A.NASRI (7)
(1/2)      A.O:S (8)
(2/5)      A:S (9)
(1/1)      A:SON (10)
(15/68)    AA (11)
(4/5)      AAA (12)
(1/2)      AAAAA (13)
(1/1)      AABC (14)
(1/1)      AABGRUPPEN (15)
(2/4)      AAC (16)
(1/1)      AACG (17)
(1/1)      AAEW (18)
(1/1)      AAF (19)
(3/5)      AAGAARD (20)
etc.
```

The numbers within parentheses **after** the term is the order number of the current term.

LOG

This command is used for custom logging. Applications can write lines to their very own log files using this command. The log file is located in the directory specified for log files in Boolware Manager. The name of the log file will be "useryyyymmdd.log", where yyyymmdd is today's date. The name of the log file can be affected if the key *name* is used. In that case, the name will be name*yyyymmdd*.log instead of user*yyyymmdd*.

log message="<*message*>" [**name**="<*name*>]

Example: Makes a note in the ”boolware” log file
```
log message="User ’Mike’ has connected" name="boolware"
```

PERFCOUNTERS

This command retrieves all the performance counters, corresponding to the performance tab in Boolware Manager.

perfcounters format="<*raw/rawc/text/xml/json*>"

Mandatory parameters:

format	the format in which the response should be displayed.	
	raw	used to get an unformatted printout. Fields are separated by TAB (\t) and a counter ends with a new line (\n): E.g. XML/JSON request:**\t**0**\t**XML/JSON request: performed**\t**0**\t**0**\t**0**\t**XML/JSON request: thread time (msec)**\t**0**\t**0**\t**0**\t**XML/JSON request: total time (msec)**\t**0**\t**0**\t**0**\n**
	rawc	same as *'raw'* except that average values are added plus clear text in 'measure time' and 'os time' fields E.g. XML/JSON request:**\t**0**\t**XML/JSON request: performed**\t**0**\t**0**\t***0.0***\t**0**\t**XML/JSON request: thread time (msec)**\t**0**\t**0**\t***0.0***\t**0**\t**XML/JSON request: total time (msec)**\t**0**\t**0**\t***0.0***\t**0**\n**
	text	is used for formatted printout corresponding to and formatted as the look in the Boolware Manager performance tab
	xml	is used for printing the performance counters in XML format
	json	is used for printing the performance counters in JSON format

Example for printout in XML format:
```
perfcounters format="xml"
```

RANK

This command are used to set the proper rank mode before presenting the result. The rank mode will be reset after each query.

NOTE! This command requires a database to be selected.

rank table="<*table name*>"
  mode="<*rank type*>"

[**term**="<*term*>"]
[**weights**="<*field1*>=<*weight1*>*[,<field2>=<weight2> etc.]*"]

Mandatory parameters:

table	current table
mode	requested rank type. The following types can be used:

norank	No ranking
occurrency	Rank by occurrence
frequency	Rank by frequency
similiarity	Rank by similarity
ascending	Rank by sort (Ascending)
descending	Rank by sort (Descending)
weightedoccurrency	Rank by weighted occurrence
weightedfrequency	Rank by weighted frequency
customrank	Rank by Custom

Optional parameters:

term	the term to be ranked on
weights	field name and its weight

Example:
In a table, Articles, the following query has been performed: FIND text:yellow cars AND title:volvo OR ford. The records containing most occurrences of the search terms should be presented first.

```
rank table="Articles" mode="occurrency"
```

will present records containing most search terms at the top of the list.

```
rank table="Articles " mode="weightedoccurrency" weights="title=10, text=3"
```

search terms found in column **title** will be multiplied by 10 and search terms found in column **text** will be multiplied by 3 before added to the total for a record.

RANKTERMS

This command will show rank statistics on all search terms used in the current query.

NOTE! This command requires a database to be selected.

rankterms **table**="<*table name*>"
mode="<*rank type*>"
[**row**=*"*<*row(s)*>"]

Mandatory parameters:

table	current table
mode	rank type. The following rank types are available:

occurrency	Rank by occurrence
frequency	Rank by frequency
weightedoccurrency	Rank by weighted occurrence
weightedfrequency	Rank by weighted frequency

Optional parameters:

row	which rows from the result on which statistics are to be calculated. Lines can be comma-separated and/or as a range of rows. If the parameter is omitted, all lines in the result are used.

Example:
```
rankterms table="Articles" mode="occurrency"
```

will get statistics on all search terms used in the current query:

```
    Number of ranked Terms=4
    Term:  text:bilar     count=4.212
    Term:  text:gul       count=313
    Term:  title:ford     count=715
    Term:  title:volvo    count=419
```

RELATE, TABLES

See *Operations guide*, Chapter *11 Interactive Query* section Related search (Join) for a complete explanation of these commands.

SAVEQUERY

Saves the current query for the specified table with the specified name.

NOTE! This command requires a database to be selected.

savequery name="<*name*>" **table**="<*table name*>" [**public**="<*session-/username*>"]

Mandatory parameters:

name	a name of the query to be saved
table	current table

Optional parameters:

public	is an id for session-/username. It is the application's responsibility to ensure that only authorized users have access to the publicly saved Queries/Results. An id can consist of users within, for example, an authority or a department within a company.

Example:
```
savequery name="Query 1" table="Articles"
```

All commands, arguments and operators from the most recent FIND command will be saved to the Articles table under the name **Query 1**, which is then used as an identification term for future use.

Example:
```
savequery name="Public query 1" table="Articles" public="SalesDep"
```

All commands, arguments and operators from the most recent FIND command will be saved to the Articles table under the name **Public query 1**, which is then used as an identification term for future use. To use **Public query 1**, you must belong to the session-/username **SalesDep** and enter this when searching. Only users whose session-/username starts with **SalesDep** can be considered.

REVIEWQUERY

Displays the contents of a saved query.

NOTE! This command requires a database to be selected.

reviewquery **name**="<*name*>"
[**database**="<*database name*>"]
[**table**="<*table name*>"]
[**public**="<*session-/username*>"]
[**ctime**="<*time stamp*>"]
[**order**="<*name/ctime/dbname/tabname*>"> [**dir**="<*asc/desc*>"]]

Mandatory parameters:

name	the name of a query to be displayed. Enter asterisk (*) for all queries

Optional parameters:

database	filtering by database name	
table	filtering by table name	
public	filtering by session-/username or beginning of session-/username	
ctime	filtering on time stamp when a query was created. The timestamp is preceded by >, < or =.	
	Intervals can be specified in the format:	
	yyyymmdd:hh:mm:ss..yyyymmdd:hh:mm:ss.	
	yyyymmdd	is year, month and day
	hh:mm:ss	is hour, minute and second
order	sort order. Valid values are:	
	dbname	sorting by database name
	tabname	sorting by table name
	name	sorting by name
	ctime	sorting by time stamp
dir	ascending (asc) or descending (desc) sort order. Used only if order is specified.	

Example:
```
reviewquery  name="*sport*" ctime=">2002" order="ctime" dir="desc"
```

All saved *Queries* that contain the word **sport** in name and are created after **2002** will be displayed in reverse chronological order; the most recently saved *Query* is presented first.

Example:
```
reviewquery  name="*sport*" public="SalesDep" ctime=">2002" order="ctime"
             dir="desc"
```

All saved *Queries* that contain the word **sport** and were created after **2002** will be displayed in reverse chronological order; the most recently saved *Query* is presented first. In this case, only public saved *Queries* for the session-/username starting with **SalesDep** will be listed.

DELETEQUERY

Deletes a saved query.

NOTE! This command requires a database to be selected.

deletequeryname="<*name*>"
[**database**="<*database name*>"]
[**table**="<*table name*>"]
[**public**="<*session-/username*>"]

[**ctime**="<*time stamp*>"]

Mandatory parameters:

name	the name of a saved query to delete. Enter asterisk (*) for all queries.

Optional parameters:

database	filtering by database name
table	filtering by table name
public	filtering by session-/username or beginning of session-/username
ctime	filtering on time stamp when a query was created. The timestamp is preceded by >, < or =. Intervals can be specified in the format: yyyymmdd:hh:mm:ss..yyyymmdd:hh:mm:ss. yyyymmdd is year, month and day; hh:mm:ss is hour, minute and second

Example:
```
deletequery name="*" ctime="<20030701"
```

In this example, all saved *Queries* created before the **first of July 2003** will be deleted.

Example:
```
deletequery name="Query1" public="SalesDep"
```

In this example, the publicly saved *Query* **Query1** for session-/username starting with "**SalesDep**" will be deleted. Note that only the one who created the saved *Query* can delete it.

SAVERESULT

Saves the current result for the specified table with the specified name.

NOTE! This command requires a database to be selected.

saveresult name="<*name*>" **table**="<*table name*>" [**public**="<*session-/username*>"]

Mandatory parameters:

name	a name of the result to be saved
table	current table

Optional parameters:

public	is an id for session-/username. It is the application's responsibility to ensure that only authorized users have access to the publicly saved Queries/Results. An id can consist of users within, for example, an authority or a department within a company.

Example:
```
saveresult name="Sports results1" table="Articles"
```

All commands, arguments and operators from the latest FIND command and the end result will be saved under the name **Sport results1**, which is then used as an identification term for future use.

Example:
```
saveresult name="Public result 1" table="Articles" public="SalesDep"
```

All commands, arguments and operators from the latest FIND command as well as the end result will be saved under the name **Public result 1**, which is then used as an identification term for future use. To use **Public Result 1**, you must belong to the session-/username **SalesDep** and enter this when searching.

REVIEWRESULT

Displays the contents of a saved result.

NOTE! This command requires a database to be selected.

reviewresult **name**="<*name*>"
[**database**="<*database name*>"]
[**table**="<*table name*>"]
[**public**="<*session-/username*>"]
[**ctime**="<*time stamp*>"]
[**order**="<*name/ctime/dbname/tabname*>"> [**dir**="<*asc/desc*>"]]

Mandatory parameters:

name	the name of a result to be displayed. Enter asterisk (*) for all results

Optional parameters:

database	filtering by database name	
table	filtering by table name	
public	filtering by session-/username or beginning of session-/username	
ctime	filtering on time stamp when a result was created. The timestamp is preceded by >, < or =.	
	Intervals can be specified in the format:	
	yyyymmdd:hh:mm:ss..yyyymmdd:hh:mm:ss.	
	yyyymmdd	is year, month and day
	hh:mm:ss	is hour, minute and second
order	sort order. Valid values are:	
	dbname	sorting by database name
	tabname	sorting by table name
	name	sorting by name
	ctime	sorting by time stamp
dir	ascending (asc) or descending (desc) sort order. Used only if order is specified.	

Example:
```
reviewresult name="result 4"
```

The saved *Result*, **result 4**, will be displayed in its entirety. This means that the complete search string will be displayed. In some cases, when, for example, similarity search is included in the saved *Result*, the search string can consist of many thousands of characters.

Example:
```
reviewresult name="*sport*" public="SalesDep" ctime=">2002"
             order="ctime" dir="desc"
```

All saved *Results* that contain the word **sport** and were created after **2002** will be displayed in reverse chronological order; the last saved result is presented first. In this case, only public saved *Results* for the session-/username starting with **SalesDep** will be listed.

DELETERESULT

Deletes a saved result.

NOTE! This command requires a database to be selected.

deleteresultname="<*name*>"
[**database**="<*database name*>"]
[**table**="<*table name*>"]
[**public**="<*session-/username*>"]
[**ctime**="<*time stamp*>"]

Mandatory parameters:

name	the name of a saved result to be deleted. Enter asterisk (*) for all results

Optional parameters:

database	filtering by database name
table	filtering by table name
public	filtering by session-/username or beginning of session-/username
ctime	filtering on time stamp when a result was created. The timestamp is preceded by >, < or =. Intervals can be specified in the format: yyyymmdd:hh:mm:ss..yyyymmdd:hh:mm:ss. yyyymmdd is year, month and day; hh:mm:ss is hour, minute and second

Example:
```
deleteresult name="Fotboll*" table="sports" ctime=">20020901"
```

In this example, all saved *Results* created after **the first of September 2002** and belonging to the table **sports** will be deleted

Example:
```
deleteresult name="Result1" public="SalesDep"
```

In this example, the publicly saved *Result* **Result1** will be deleted. Note that only the one who created the saved *Result* can delete it.

SAVESCRATCH

Saves the current result in a scratch result for the specified table with the specified name.

NOTE! This command requires a database to be selected.

savescratch **name**="<*name*>"
table="<*table name*>"
[**resultat**="<*intermediate/custom*>"]

Mandatory parameters:

name	a name of the scratch result to be saved
table	current table

Optional parameters:

resultat which result to save. Valid values are:

intermediate	the result of the last query
custom	the result in the ‘Custom list’ when using Flow

If the parameter is omitted, the current result is used

Example1:
```
savescratch name="people" table="People"
```

The current result from the last search will be temporarily saved in the ***People*** table in the named ***'people'*** scratch. This result can be used at a later time.

Example2:
```
savescratch name="flow" table="Companies" result="custom"
```

The current result from the 'Custom list' will be saved temporarily in the named scratch *'flow'*. This result can be used at a later time.

Example3:
```
savescratch name="tmppeople" table="People" result="intermediate"
```

The number of hits from the most recent search command from the ***People*** table will be temporarily saved in the named ***'tmppeople'*** scratch. This result can then be used at a later time.

DELETESCRATCH

Deletes a saved scratch result.

NOTE! This command requires a database to be selected.

deletescratch **name**="<*name*>"
[**database**="<*database name*>"]
[**table**="<*table name*>"]

Mandatory parameters:

name	the name of a scratch result to be deleted. Enter the asterisk (*) for all scratch results.

Optional parameters:

database	filtering by database name
table	filtering by table name

Example1:
```
deletescratch name="address" table="Companies"
```

The saved scratch result with the name ***address*** in the **Companies** table will be deleted.

Example2:
```
deletescratch name="address"
```

The saved scratch result named ***address*** will be deleted.

SETAUTOTRUNC

This command is used to enable or disable automatic truncation.

setautotrunc value="<*yes/no*>"

Mandatory parameters:

value	automatic truncation (*yes/no*). Default value is ‘*no*’.

SETENCODING

This command is used to set character encoding.

setencoding value="<*iso-8859-1/utf-8*>"

Mandatory parameters:

value	character encoding. Default values is *iso-8859-1*. See Chapter 12 "Unicode" in the Operations Guide.

SETHISTORY

This command is used to enable or disable search history.

sethistory value="<*yes/no*>"

Mandatory parameters:

value	search history or not (*yes/no*). Default value is ‘*no*’.

SETINDEXEXIT

This command is used to enable or disable the use of terms from custom indexing.

setindexexit value="<*yes/no*>"

Mandatory parameters:

value	use terms from customized indexing when searching (*yes/no*). Default value is *‘yes’*.

SETMAXEXECUTIONTIMEOUT

This command is used to set the maximum execution time value for the session.

setmaxexecutiontimeout value="<*value*>"

Mandatory parameters:

value	sets new maximum execution time in seconds for the session. If the value exceeds the system's maximum execution time, the value will be set to the system's maximum execution time.

SETQUERYLIMVALUE

This command is used to set the limit value for the number of hits per term.

setquerylimvalue value="<*value*>"

Mandatory parameters:

value	limit value for number of hits per term. Default value is 100.

SETQUERYOPTIONS

This command is used to set values for the subsequent query. The specified values only apply to the next query and will after this query be reset to the default setting.

NOTE! This command requires a database to be selected.

setqueryoptions [**backonzero**="<*yes/no*>"]
[**defaultoperator**="<*operator*>"]
[**querylimvalue**="<*value*>"]
[**resultbitmap**="<*result/customresult*>"]

Optional parameters:

backonzero	indicates whether automatic back at zero results between operators should be applied or not. The value '*yes*' indicates that automatic back should be done, while the value '*no*' (default value) means that no automatic back must be performed.
defaultoperator	indicates which operator is to be used between terms that are separated by a blank. Valid values are: *and*, *or*, *not* and *xor*. If no value is specified *AND* will be used.
querylimvalue	all terms that are in more records than this value will be ignored when searching, when using the querylim subcommand. Default value is 0.
resultbitmap	indicates which result should be active, *result* or *customresult*. The default value is '*result*'.

Example:
Specify that there should be automatic back at zero response, the operator between commands should be AND and all terms should be included when searching:

```
setqueryoptions  backonzero="yes" defaultoperator="and" querylimvalue="0"
```

GETQUERYOPTIONS

This command displays the values that the following query will use. The specified values only apply to the next query and will be reset to the default value after the query.

NOTE! This command requires a database to be selected.

getqueryoptions [**type**="<*backonzero/defaultoperator/querylimvalue/resultbitmap*>"]

Optional parameters:

type	the type of setting to be displayed.	
	backonzero	*'yes'* means that automatic back at zero result applies while '*no*' means that no automatic back will be performed.
	defaultoperator	returns the current operator: AND, OR, NOT or XOR.
	querylimvalue	the limit value when the querylim subcommand is used
	resultbitmap	which result is active, *result* or *customresult*

If the *type* parameter is omitted, the settings for all types are returned.

SETSESSIONTIMEOUT

This command is used to set the session timeout value.

setsessiontimeout value="<*value*>"

Mandatory parameters:

value	sets a new "session timeout" for this session (in seconds).

SETSETSEARCH

This command is used to enable or disable set search.

setsetsearch value="<*yes/no*>"

Mandatory parameters:

value	set search or not (*yes/no*). Default value is '*no*'.

REVIEWSET

Displays the contents of a saved set.

NOTE! This command requires a database to be selected.

reviewset **name**="<*name*>"
[**database**="<*database name*>"]
[**table**="<*table name*>"]
[**ctime**="<*time stamp*>"]
[**order**="<*name/ctime/dbname/tabname*>"> [**dir**="<*asc/desc*>"]]

Mandatory parameters:

name	the name of a set to be displayed. Enter asterisk (*) for all sets

Optional parameters:

database	filtering by database name	
table	filtering by table name	
ctime	filtering on time stamp when a set was created. The timestamp is preceded by >, < or =. Intervals can be specified in the format: yyyymmdd:hh:mm:ss..yyyymmdd:hh:mm:ss.	
	yyyymmdd	is year, month and day
	hh:mm:ss	is hour, minute and second
order	sort order. Valid values are:	
	dbname	sorting by database name
	tabname	sorting by table name
	name	sorting by name
	ctime	sorting by time stamp
dir	ascending (*asc*) or descending (*desc*) sort order. Used only if *order* is specified.	

Example:
```
reviewset name="S*1" database="Magazines" ctime=">20031016:12:00:00"
          order="tabname" dir="desc"
```

All saved Sets, starting with **S** and ending with **1** and are created after **12:00 on October 16, 2003** and belong to the database **Magazines** will be displayed sorted by table name in descending order.

DELETESET

Deletes a saved set result.

NOTE! This command requires a database to be selected.

deleteset **name**="<*name*>"
[**database**="<*database name*>"]
[**table**="<*table name*>"]
[**ctime**="<*time stamp*>"]

Mandatory parameters:

name	the name of a set result to delete. Enter asterisk (*) for all sets.

Optional parameters:

database	filtering by database name
table	filtering by table name
ctime	filtering on time stamp when a set was created. The timestamp is preceded by >, < or =. Intervals can be specified in the format: yyyymmdd:hh:mm:ss..yyyymmdd:hh:mm:ss. yyyymmdd is year, month and day; hh:mm:ss is hour, minute and second

Example:
```
deleteset name="S1*" database="Magazines"
```

In this example, all saved *Sets* that start with and belong to the *Magazines* database will be deleted.

SETSTRICTASIS

This command is used to enable or disable strict character handling for the wordasis() and stringasis() subcommands.

setstrictasis value="<*yes/no*>"

Mandatory parameters:

value	enable/disable (*yes/no*) strict character handling for the wordasis() and stringasis() subcommands. Default value is '*no*'. The value '*yes*' means that the normal wildcards '*', '?' '!' and '#' will be handled as regular characters and not as wildcards when searching these subcommands

STATISTICS

This command is used to retrieve simple statistics from numeric Boolware indexes.

NOTE! This command requires a database to be selected.

statistics **table**="<*table name*>"
field="<*field name*>"
numgroups="<*number of groups*>"
[**useall**="<*yes/no*>"]
[**groupvalues**="<*yes/no*>"]
[**emptyvalues**="<yes/*no*>"]

Mandatory parameters:

table	is the name of the table
field	is the name of the numeric field
numgroups	is the number of groups to divide the result into

Optional parameters:

useall	is set to '*yes*' if you want to retrieve values from the complete table. Default value is '*no*'
groupvalues	is set to '*yes*' if you want all limit values for specified groups. Default value is '*no*'
emptyvalues	is set to 'yes' if empty values are to be included. Default value is '*no*'

Values obtained are: number of records included in the statistics, the value that occurs most often, number of records that contain the most common value, the sum of all values, the arithmetic mean, the smallest value, the largest value, the standard deviation, the variance, the median value, upper value for specified 'group', lower value for specified 'group' and all values for a specified group.

You can specify whether you want to perform statistics for the entire table or only on the searched result.

A 'group' indicates in how many parts you want to divide the result obtained.

By default, you always get the lower and upper value for the specified 'group'. You can specify with a parameter, if you want all limit values for the specified 'group'.

Example: Get statistics for the numeric field "Solidity". Perform the calculation on the current result (all companies in Stockholm) and determine the number of groups to 5 (quintile). You also want all limit values for the specified group.

```
statistics table="Companies" field="Solidity" numgroups="5"
           useall="no" groupvalues="yes"
```

```
No. of values:  38283
Sum:            -779180537.840000

Max:            1326.410000
Average:        -20353.173415
Min:            -425099700.000000

Variance:       5744603486147.931600
Std. deviation: 2396790.246590

Upper:          82.435000
Medium:         38.870000
Lower:          9.500000

Most frequent:  100.000000
Occurrences:    3095

Group values:    9.500000, 27.825000, 51.435000, 82.435000
```

No. of values:	no. of observations (no. of records containing values in Solidity)
Sum:	the sum of all observations

Max:	the highest value
Average:	the arithmetic mean value
Min:	the lowest value
Variance:	calculated variance
Std. deviation:	calculated standard deviation
Upper:	highest limit value for the specified group
Lower:	lowest limit value for the specified group
Most frequent:	the most frequent value
Occurrences:	no. of observations for the most frequent value
Group values:	all limit values for the specified group

Example: Get statistics for the field "Cash liquidity". Start from the total number of entries in the table and determine the number of groups to 10. You also want all limit values for the specified group.

```
statistics table="Companies" field="Cash liquidity" numgroups="10"
           useall="yes" groupvalues="yes"
```

```
No. of values:  217869
Sum:            610811992.529998

Max:            161918100.000000
Average:        2803.574591
Min:            -35215300.000000

Variance:       179916774079.192990
Std. deviation: 424165.974683

Upper:          538.960000
Medium:         123.810000
Lower:          33.890000

Most frequent:  100.000000
Occurrences:    63

Group values:   33.890000, 60.820000, 83.630000, 103.610000, 123.810000,
                149.970000, 191.260000, 272.360000, 538.960000
```

Chapter 2
API description

This chapter describes in detail all functions available in the Boolware API library.

The Functions are described regarding: name, parameters and return codes.

Detailed API description

The functions are presented in alphabetical order.

All functions return an integer code, which is zero (SOFTBOOL_OK) on success, positive when the system has a notice (warning or information) and it is negative if an error occurred. The two exceptions are the functions **BCCreateClient()** and **BCGetErrorCode()**, where the Boolware client instance is created and the current error code is returned.

The examples do not include tests for return codes for the sake of readability. A real application should always test the return code after each call.

It is also assumed that a proper call have been done to the **BCCreateClient()** and the Boolware client is stored in a variable called ***mClient***.

Depending on your language of choice, the Boolware interface is defined in different modules.

C/C++: **sbtypes.h, softbool.h, boolwarelclient.h**.

Records and error codes are documented in Appendix 1 and Appendix 2.
In all communication with Boolware, where an integer number is used to identify objects, the first object is always zero. For example, the first record in a search result is record zero.

BCAddCalcColumn()

BCAddCalcColumn(**BWClient** client, **const char** *table, **const char** *column,
const char *formula)

Adds a computed column to a table.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- the name of the table
const char *column	- name of the new column
const char *formula	- formula for the new column

You can use formulas to derive new (virtual) columns that are calculated from the values of other columns.
When fetching a row from the data source, computed columns can be included. If so, Boolware will perform the arithmetic and return the calculated value.
A calculated column remains in existence until a call to **BCDropCalcColumn(), BCAttach()** or **BCDetach()** is done.
Calculated columns can be explicitly removed using BCDropCalcColumn().

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCDropCalcColumn()

Example:
Simple formula for calculating profit per employee.

```
if((BCAddCalcColumn(mClient, "Companies",
   "Profit per employee", "profit / numemp") != SOFTBOOL_OK)
        BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCAttach()

BCAttach(**BWClient** client, **const char** *dsnName)

Attaches to (makes current) a Boolware Index.

Parameters:

BWClient client	- the Boolware client instance
const char *dsnName	- the name of the Index

The Boolware library operates with the concept of a "current index". This is the index that all subsequent actions are directed at. Using the "**BCAttach()"** function, you select the current index.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:
**BCDetach(), BCGetNumberDatabases(), BCGetDatabaseInfo(**)

Example:
Attach to the Index called "Companies" at the server "Charlie"

```
char msg [256];
BCConnect(mClient, "Charlie", "");
if((BCAttach(mClient, "Companies") != SOFTBOOL_OK)
  BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCConnect()

BCConnect(**BWClient** client, **const char** *server,
char *session)

Connects to a Boolware server.

Parameters:

BWClient client	- the Boolware client instance
const char *server	- A computer name or an IP address, where Boolware server executes.
char *session	- The name of the session to create, or reconnect to.

Boolware Server must be started on a computer somewhere on the network, that can be reached by the application before a connection can be done.
The server shall be the computer network name, or its IP address; for example 192.168.0.1.
Please note that the server must be accessible (have access rights) from the application computer.
If you pass an empty string as session name, Boolware will create a new, unique name for the session. This name can be retrieved using **BCGetSessionInfo().**

The connection will remain until BCDisconnect() is called.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCDisconnect(), BCGetSessionInfo()

Example:
Connects with server "Charlie" and lets Boolware create a new, unique session name.

```
char msg [256];
BCConnect(mClient, "Charlie", "") != SOFTBOOL_OK)
  BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCConnectExecute()

BCConnectExecute(**BWClient** client, **const char** *server, **char** *sessName,
char * encoding, **int** stateless, **const char** *cmd, **char** **response,
int *int1, **int** *int2)

Executes a Boolware command with automatic connect to Boolware server, returning a response string and two integers.

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

If *encoding* is set to the string "utf-8" the session will be set as a unicode session, otherwise it will be a "ISO-8859-1" session

If *stateless* is 1 the session will be disconnected and logged out automatically by Boolware server, else the session will stay alive until **BCDisconnect** has been called.

Parameters:

BWClient client	- The Boolware client instance
*const char *server*	- A computer name or an IP address to Boolware server
*char *sessName*	- The name of the session
*char *encoding*	- Requested session encoding
int stateless	- Should be set to 1 or 0
const char *cmd	- Boolware command string
*char **response*	*- Response string*
int *int1	- First response integer
int *int2	- Second response integer

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in ConnectExecute.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCDisconnect(), BCExecute(), BCGetSessionInfo()

BCConnectXml()

BCConnectXml(**BWClient** client,
const char *server,
char *sessName,
int stateless,
const char *request,
BCXmlReply_t *reply)

Performs an XML request with an automatic connection to Boolware server and return the reply as an XML document.

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

If *stateless* is 1 the session will be disconnected and logged out automatically by Boolware server, else the session will stay alive until **BCDisconnect** has been called.

Parameters:

BWClient client	- The Boolware client instance
*const char *server*	- A computer name or an IP address to Boolware server
*char *sessName*	- The name of the session
int stateless	- Should be set to 1 or 0
*const char *reques t*	- The XML request
*CBXmlReply_t *reply*	- Pointer to the xml response and the length of the reply

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCDisconnect(), BCGetSessionInfo()

BCConnectXmlNoResponse()

BCConnectXml(**BWClient** client,
const char *server,
char *sessName,
const char *request)

Performs an XML request with an automatic connection to Boolware server.

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

To fetch the retrieved data the method **BCFetchRow** should be used.

Parameters:

BWClient client	- The Boolware client instance
*const char *server*	- A computer name or an IP address to Boolware server
*char *sessName*	- The name of the session
*const char *reques t*	- The XML request

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCDisconnect(), BCGetSessionInfo()

BCCreateClient()

BCCreateClient ()

Parameters:
    None

Create a Boolware client instance.

This routine creates a Boolware client instance that should be used throughout all other function calls.
Only one call to this function should be done during a session.

Return:
- BWClient instance
- NULL

See:
    BCFreeClient ()

Example:

Create a Boolware client instance and make a connection to a Boolware server.

```
char msg [256];
mClient = BCCreateClient();
if(mClient != NULL)
      BCConnect(mClient, "127.0.0.1", "");
```

BCDetach()

BCDetach(**BWClient** client)

Detaches from the current Boolware Index.

Parameters:

BWClient client - the Boolware client instance

No searches can be performed until a new Index is made current using BCAttach.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCAttach()

Example:
Detach from the current Boolware index.

```
char msg [256];
if((BCDetach(mClient) != SOFTBOOL_OK)
  BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCDisconnect()

BCDisconnect(**BWClient** client, **const int32** terminate)

Disconnects from Boolware.

Parameters:

BWClient client	- the Boolware client instance
const int32 terminate	-Indicates if the application wants to close the session (log out), or leave it running so that it can be connected to later. 0, do not close the session 1, close the session (log out)

If *logout* is zero (0), the current session will remain in Boolware, and can later be reconnected to using BCConnect() with the appropriate session name.
If *logout* is non-zero, the session will be closed.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCConnect(), BCDetach()

Example:
Disconnect from Boolware, but let the session remain.

```
if((BCDisconnet(mClient, 0) != SOFTBOOL_OK)
  BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

Disconnect from Boolware, and close this session.

```
if((BCDisconnet(mClient, 1) != SOFTBOOL_OK)
  BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCDropCalcColumn()

BCDropCalcColumn(**BWClient** client, **const char** *table, **const char** *column)

Drops a calculated column from a table.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- name of the table
const char *column	- name of the calculated column

The specified *column* will be dropped from the table.
If "*" is used as *column*, all calculated columns will be dropped from the table.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCAddCalcColumn()

Example:
Drop all calculated columns from a table.

```
if((BCDropCalcColumn(mClient, "Company", "*") != SOFTBOOL_OK)
  BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCExecute()

BCExecute(**BWClient** client, **const char** *cmd, **char** **response,
int *int1, **int** *int2)

Executes a Boolware command, returning a response string and two integers.

Parameters:

BWClient client	- the Boolware client instance
const char *cmd	- Boolware command string
char **response	- Response string
int *int1	- First response integer
int *int2	- Second response integer

The response areas depends on the command string. Valid commands are:

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:
**BCQuery(**)

Example:
Activates automatic truncation.

```
char *response;
int  int1;
int  int2;
if((BCExecute(mClient, "setautotrunc value='yes'", &response, &int1, &int2) !=
SOFTBOOL_OK)
  BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCFetchKey()

BCFetchKey(BWClient client, const char *table,
 char *col,
 const int32 hitNo,
 char *buff,
 const int32 buffSz,
 float *score,
 int32 *recNo,
 int32 *rankMode)

Fetches a primary or foreign key.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- Table name
char *col	- Column name
const int32 hitNo	- Hit number (starting from zero)
char *buff	- Returned string
const int32 buffSz	- Size of 'buff'
float *score	- Score for this hit number
int32 recNo	- Record ID for this hit number
int32 rankMode	- Current rank mode

The column to be fetched must be part of a primary key or a foreign key.
The buff argument must be sufficiently large to hold the returned value, or it will be truncated.
The size of the buff parameter is given in the buffSz parameter.
The hitNo parameter tells from which row to fetch the primary or foreign key.
In addition to the above the following information is also returned regarding the tuple:

1 score – the score for the tuple. Scores are assigned when ranking and performing similarity searches, and are used to present results in relevance order. The score is always 1 if neither similarity search nor ranking has been done.

2 recNo is Boolware's internal record ID

3 rankMode tells how the search result is arranged. It can be ordered after occurrences of a specific word, alphabetical order, similarity etc.

The retrieved primary/foreign key can be used to fetch a tuple from the data source (using perhaps ODBC).

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCFetchRow().

Example:
Fetch the first 25 primary keys resulting from a successful search in the "Company" table.

```
int32 i, result, recNo, rankMode, maxPK;
char  key [128];
float score;

// Search
BCQuery(mClient, "Company", "FIND Name:An*", &result, NULL);
maxPK = min(result, 25);

for(i = 0; i < maxPK; i++)
   {
   // Get next primary key
   BCFetchKey(mClient, "Company", "ID", i, key, sizeof(key),
              &score, &recNo, &rankMode);
   // Use the primary key here …
   }
```

BCFetchQueryHistory()

BCFetchQueryHistory(**BWClient** client, **const char** *table,
const int32 rowNo,
BCQHistoryData_t *resultRow)

Fetches a row from the query history.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- Table name. Use this to specify from which table to retrieve Query history.
const int32 rowNo	- Desired row number. Use this to tell Boolware which row you want. The first row is zero.
BCQHistoryData_t resultRow	- Returned row

Use this function to retrieve a single entry from the query history. You supply which table and line number, Boolware returns information about the Query history line in result.
The Query history is an optional feature that can be used to keep track of how you have arrived at the current result.
Query history means that all queries and results are saved from one FIND command to the next FIND. The Query history can also be navigated using the BACK and FORWARD commands.
Boolware maintains a log of the commands used and the number of found records for each Query command (find, and, or, not etc.). This log consists of one "line" per query command.
The rowNo parameter tells Boolware which query history item (row) you desire.
The row argument is where Boolware returns information about the requested search.
You should use the API BCGetQueryHistoryInfo() first, so that you know how many items are available.
Note that Query history only is available after a call to **BCSetSessionInfoXml()**. If Query history isn't available, an error code is returned from this function.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCGetQueryHistoryInfo(), BCSetSessionInfoXml(), QL language BACK and FORWARD.

Example:
```
int32 i;
BCQHistoryInfo_t queryHistory;
BCQHistoryData_t resultRow;

// Get information about Query history
BCGetQueryHistoryInfo(mClient, "Company", &queryHistory);

// Fetch all query history lines
for(i = 0; i < queryHistory.currQHistoryRow; i++)
   {
   // Fetch next Query history line
   BCFetchQueryHistory(mClient, "Company", i, &resultRow);

   // Use the retrieved information
   resultRow.QLString  // NULL terminated string
   resultRow.hitCnt    // Total result
   resultRow.intermediateRes // Result for string
   }

// After the three first lines
FIND Name:John          7.540   7.540
AND  Age:>40              960  12.972
AND  Salary:>1000          37   9.432
```

The first column contains the query string. The second column contains the number of found records, and the third column contains the number of found records for the query string alone. In

other words, there are 9.432 persons with a salary higher than 1000, but only 37 of them are named "John" and are more than 40 years old.
There is another way to achieve the above Query History by sending only one command to Boolware; you use the special operators ANDF, ORF, NOTF and XORF:
FIND Name:John ANDF Age:>40 ANDF Salary:>1000.
See description of the special operators (ANDF, ORF, NOTF and XORF) in the Operations Guide.

BCFetchRow()

BCFetchRow(**BWClient** client, **const char** *table,
  const char *cols,
  const int32 hitNo,
  const int32 maxChars,
  BCRowData_t *rowData)

Fetches a tuple from the data source.

Parameters:

BWClient	client	- the Boolware client instance
const char	*table	- table name
const char	*cols	- comma separated list of columns
const int32	hitNo	- the hit number of the desired tuple. First tuple is zero.
const int32	maxChars	- maximum number of characters for any field
BCRowData_t	rowData	- returned tuple

The table name controls from which table to fetch the row.
The names of the desired columns should be passed in the cols parameter. An asterisk (*) means "all columns".
Use the hitNo parameter to tell Boolware which row to retrieve. The first row in your search result is row number zero.
Use the maxChars parameter to limit the number of retrieved characters from any column. Pass zero (0) to retrieve all data in the field, regardless of size.
Boolware will return the tuple in the rowData parameter.
When fetching many tuples, you are advised to use the BCSetFetchSize() function. This tells the system to fetch more than one row per turn-around to the data source, which in turn helps to improve performance.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCSetFetchSize(), BCRowData_t, BCFetchKey()

Example:
Fetch the first 25 tuples from the current search result. Fetch no more than 20 characters from any field.

```
int32 i, result, recNo, rankMode, maxRows;
char  key [128];
BCRowData_t resultRow;

// Search
BCQuery(mClient, "Company", "FIND Name:Anders", &result, NULL);

// Limit no. of retrieved tuples
maxRows = min(result, 25);

// Set fetch size
BCSetFetchSize(mClient, maxRows);

// Go get the found tuples
for(i = 0; i < maxRows; i++)
   {
   // Get next tuple
   BCFetchRow(mClient, "Company", " Name, Address, Phone",
              i, 20, &resultRow);
   // Use the retrieved tuple…
   }
```

BCFetchSimVector()

BCFetchSimVector(**BWClient** client, **const char** *table,
  const int32 vectorType,
  const int32 contentType,
  const int32 hitNo,
  BCRowData_t *rowData)

Fetches a similarity vector as a tuple from the Boolware Server.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- table name
const int32 vectorType	- type of vector requested
const int32 contentType	- type of vector content requested
const int32 hitNo	- the hit number of the desired tuple; First tuple is zero.
BCRowData_t rowData	- returned tuple

The table name controls from which table to fetch the similarity vector.
Use the *vectorType* parameter to get either the query vector or the result vector. Use either **BSIM_QUERY_VECTOR** for the query vector or **BSIM_RESULT_VECTOR** for the result vector.
Use the *contentType* parameter to get the vector as terms (text) or as numeric codes. Use either **BSIM_TERM_TEXT** for the actual term or **BSIM_TERM_NUMBER** for the numeric code.
Use the *hitNo* parameter to tell Boolware which row to start from within the current result. The first row in your search result is row number zero.
Boolware will return the tuple in the *rowData* parameter with the two columns: "*Primary Key*" and "*Similarity vector*".
When fetching many tuples, you are advised to use the **BCSetFetchSize()** function. This tells the system to fetch more than one row per turn-around to the data source, which in turn helps to improve performance.
NOTE! To fetch the terms as text could take long time.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCSetFetchSize(), BCRowData_t

Example:
Fetch the query vector and then fetch 25 similarity vectors from the current similarity search result starting with the relative record 11 (NOTE, that the first record is 0 (zero)). The terms will appear as numeric codes rather than text in this case.

```
int32 i;
BCRowData_t resultRow, queryRow;

// Get the Query vector as numeric codes
BCFetchSimVector(mClient, "Company", BSIM_QUERY_VECTOR,
      BSIM_TERM_NUMBER, 0, &queryRow);

// Set maximum number of lines to fetch
BCSetFetchSize(mClient, 25);

// Get 25 similarity vectors starting from the 11th
for(i = 10; i < 35; i++)
   {
   // Get next tuple
   BCFetchSimVector(mClient, "Company", BSIM_RESULT_VECTOR,
      BSIM_TERM_NUMBER, i, &resultRow);

   // Use the retrieved tuple…
   // NOTE Score is a member of BCRowData_t
   }
```

A resultrow contains two columns; the first column contains the Primary key as text, while the second column contains all terms for the current record. The terms could be presented as text (words) or numeric codes (digits). The frequency is always specified within slashes(/). In BCRowData_t an element, score (float), could be used when exporting the vectors. The Query vector does not contain any score or Primary key. In this case the Primary key is set to: Query Vector and the score is set to 0.0000.

Column1 (PK)	Column2 (The Similarity Vector)
Query Vector	123/2/, 213/3/, 435/1/, … 3242/1/
143	112/1/, 123/1/, 435/2/, … 6798/2/
22	123/3/, 213/2/, 682/4/, … 5467/1/
7743	11/4/, 100/4/, 123/1/, … 4798/3/
…	
…	
79	72/1/, 104/1/, 435/1/, … 12455/1/

BCFetchTerm()

BCFetchTerm(**BWClient** client, **BCTerm_t** *ixTerm)

Fetches an index word (term) from a column.

Parameters:

BWClient client	- the Boolware client instance
BCTerm_t *ixTerm	- the term

The *ixTerm* parameter is used as follows:

hits	no. of records within your current search result where this term occurs at least once
totalHits	no. of records where this term occurs at least once
termNo	not used
termType	not used
term	the index word (term). A zero terminated string.

The function **BCStartFetchTerm()** must be used before calling this function, to tell Boolware from which table and column to retrieve terms etc.

If subzoom is active, there may occur column names within the fetched terms. These are recognized by their "hits" element is set to -1. This means that "term" contains current column name for the coming terms, up to the next "hits" of -1 is found. The "totalHits" element indicates what level (zero based) and its terms have in the hierarchy.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCStartFetchTerm()

Example:

Fetch the first 25 terms from the "Name" column in the "Employee" table. Start fetching terms alphabetically beginning from term "John" and retrieve only those terms that are found in your current search result.

```
int32    i, result;
bool     zoomed = TRUE;
BCTerm_t term;
// Search
BCQuery(mClient, "Employee", "FIND Text:stock", &result, NULL);

// Initiate fetching index terms
strcpy(term.term, "john");
BCStartFetchTerm(mClient, "Employee", "Name", &term, zoomed);

// Fetch desired terms
for(i = 0; i < 25; i++)
   {
   // Fetch next term
   if(BCFetchTerm(mClient, &term) != SOFTBOOL_OK)
      break;

   // Use this term here…
   }
```

BCFreeClient()

BCFreeClient (BWClient client)

Parameters:

BWClient client - Boolware client instance

Free a Boolware client instance.

This function free the Boolware client instance created by a call to the function **BCCreateClient ()** and have been used throughout the entire session..

Returns:

- SOFTBOOL_OK

See:

BCCreateClient ()

Example:

```
BCFreeClient(mClient);
```

BCGetColumnInfo()

BCGetColumnInfo(**BWClient** client, **const char** *table,
  const int32 num,
  BCColumnInfo_t *info)

Fetches information about a specific column.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- desired table
const int32 num	- column index, first is zero
BCColumnInfo_t *info	- returned column information

The information retrieved is: the column name, indexing flags, column size, data type, if column is part of primary key, decimal count (if numeric) etc.
Before this function can be used, it is wise to use the **BCGetNumberColumns()** function.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCGetNumberTables(), BCGetTableInfo(), BCGetNumberColumns().

Example:
Connect to server "Charlie" and database "Company". Fetch info on all columns in the second table.

```
int32 i, noTables, noCols;
BCTableInfo_t  tabInfo;
BCColumnInfo_t colInfo;

// Connect with "Charlie"
BCConnect(mClient, "Charlie", "");

// Attach to the index "Company"
BCAttach(mClient, "Company");

// Get number of tables in this database
BCGetNumberTables(mClient, &noTables);

// Check that at least two tables exist
if(noTables < 2)
      Give Error Message;

// Get information about the second table
BCGetTableInfo(mClient, 1, &tabInfo);

// Get number of columns in this table
BCGetNumberColumns(mClient, tabInfo.tabName, &noCols)

// Get each column
for(i = 0; i < noCols; i++)
   {
   // Get next column
   BCGetColumnInfo(mClient, tabInfo.tabName, i, &colInfo);

   // Use info about this column
   }
```

BCGetColumnInfoEx()

BCGetColumnInfoEx(**BWClient** client, **const char** *table,
  const int32 num,
  BCColumnInfoEx_t *info)

Fetches information about a specific column.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- desired table
const int32 num	- column index, first is zero
BCColumnInfoEx_t *info	- returned column information

The information retrieved is: the column name, indexing flags, column size, data type, if column is part of primary key, decimal count (if numeric) etc.
Before this function can be used, it is wise to use the **BCGetNumberColumns()** function.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCGetNumberTables(), BCGetTableInfo(), BCGetNumberColumns().

Example:
Connect to server "Charlie" and database "Company". Fetch info on all columns in the second table.

```
int32 i, noTables, noCols;
BCTableInfo_t  tabInfo;
BCColumnInfoEx_t colInfo;

// Connect with "Charlie"
BCConnect(mClient, "Charlie", "");

// Attach to the index "Company"
BCAttach(mClient, "Company");

// Get number of tables in this database
BCGetNumberTables(mClient, &noTables);

// Check that at least two tables exist
if(noTables < 2)
      Give Error Message;

// Get information about the second table
BCGetTableInfo(mClient, 1, &tabInfo);

// Get number of columns in this table
BCGetNumberColumns(mClient, tabInfo.tabName, &noCols)

// Get each column
for(i = 0; i < noCols; i++)
   {
   // Get next column
   BCGetColumnInfoEx(mClient, tabInfo.tabName, i, &colInfo);

   // Use info about this column
   }
```

BCGetDatabaseInfo()

BCGetDatabaseInfo(**BWClient** client, **const int32** num,
BCDatabaseInfo_t *info)

Fetches information about a specific database.

Parameters:

BWClient client	- the Boolware client instance
const int32 num	- database number
BCDatabaseInfo_t *info	- returned database info

The number (index) of the desired database is passed by the application in parameter num. Boolware will return information about the desired database in parameter info.
Information retrieved is: database name, remark, status, data source name (DSN) etc.
NOTE. It is the dsnName part of the **BCDatabaseInfo_t** structure that should be used in the **BCAttach()** function.
It is wise to use **BCGetNumberDatabases(**) before calling this function

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCConnect(), BCGetNumberDatabases(), BCDatabaseInfo_t.

Example:
Connect to server "Charlie" and fetch information about all databases at this server.

```
int32            i, noDatabases;
BCDatabaseInfo_t databaseInfo;

// Connect with "Charlie"
BCConnect(mClient, "Charlie", "");

// Fetch number of databases at this server
BCGetNumberDatabases(mClient, &noDatabases);

// Get info on each database
for(i = 0; i < noDatabases; i++)
   {
   // Get info on next database
   BCGetDatabase(mClient, i, &databaseInfo);

   // Use the retrieved information
   …
   }
```

BCGetErrorCode()

BCGetErrorCode(**BWClient** client)

Fetches the most recent error code

Parameters:

BWClient client - the Boolware client instance

The code for the most recently occurred error (or warning) in Boolware is returned.
The code can be used to retrieve a textual message. Messages exists in two languages: English and Swedish.
Using the error code, its textual message can be retrieved using **BCGetErrorMsg().**

Returns:

- the most recent error code; SOFTBOOL_OK if no error occurred.

See also:

BCGetErrorMsg().

Example:

Shows the last error message

```
char msg [256];
// Fetch last known error to msg
BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCGetErrorMsg()

BCGetErrorMsg(**BWClient** client, **const int32** errCode,
char *msg,
const int32 size)

Get the last Error Message.

Parameters:

BWClient client	- the Boolware client instance
const int32 errCode	- the Error Code of the requested Error Message
char *msg	- buffer to hold Error Message
const int32 size	- size of buffer to hold Error Message

The code for the Error Message that the text should be fetched for is specified in errCode.
The fetched text is stored in msg.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCGetErrorCode().

Example:
Write the requested Error Message.

```
char msg [256];
// Fetch the requested Error Message and put it msg
BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCGetNumberColumns()

BCGetNumberColumns(**BWClient** client, **const char** *table,
    int32 *num)

Fetch number of Columns in the specified Table.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- current Table
int32 *num	- number of Columns

The name of the current Table is specified in table.
Number of Columns for the specified Table will be stored in num.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
    BCGetColumnInfo().

Example:
    Fetch number of Columns in Table "Employees".

```
int32 noColumns;
// Fetch number of Columns in Table "Employees" in the attached database
BCGetNumberColumns(mClient, "Employees", &noColumns);
```

BCGetNumberDatabases()

BCGetNumberDatabases(**BWClient** client, **int32** *num)

Fetch number of Boolware Index.

Parameters:

BWClient client	- the Boolware client instance
int32 *num	- number Databases

Number of Databases on the connected server is stored in the num.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCConnect(), BCGetDatabaseInfo().

Example:
Get number of Databases on the server named "Charlie".

```
int32 noDatabases;

// Connect to "Charlie"
BCConnect(mClient, "Charlie", "");

// Get number databases
BCGetNumberDatabases(mClient, &noDatabases);
```

BCGetNumberTables()

BCGetNumberTables(BWClient client, int32 *num)

Get number of Tables for selected Boolware Index.

Parameters:

BWClient client	- the Boolware client instance
int32 *num	- number Tables

Number tables in the selected Database is stored in num.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCGetTableInfo().

Example:
Get number tables in the Boolware Index "Company".

```
int32 noTables;

// Connect to server "Charlie"
BCConnect(mClient, "Charlie", "");

// Attach to the index "Company"
BCAttach(mClient, "Company");

// Get number of Tables in the Database "Company"
BCGetNumberTables(mClient, &noTables);
```

BCGetPerfCounters()

***Note!** This function is deprecated, please use the execute command **perfcounters** instead.*
Read more in chapter 1 "Execute commands in Boolware".

BCGetPerfCounters(**BWClient** client, **BCPerfCounters** *perf)

Get all performance counters from the Boolware server.
Tips! Check out the Boolware Manager on the "Performance"-tab to see content of all the counters.

Parameters:

BWClient client	- the Boolware client instance
BCPerfCounters *perf	- information storage area

In the *perf* the requested information will be stored about all performance counters.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

Example:
Get performance counters from connected Boolware server.

```
int32 i;
BCPerfCounters perf;

// Get all counters
BCGetPerfCounters(mClient, &perf);
```

BCGetQueryHistoryInfo()

BCGetQueryHistoryInfo(**BWClient** client, **const char** *table,
BCQHistoryInfo_t *resultInfo)

Get information about current query history.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- requested table
BCQHistoryInfo_t *resultInfo	- information storage area

Table name *table* selects from which table the requested query history will be fetched.
In the *resultInfo* the requested information will be stored about current query history.
In the query history all queries and all results will be stored from a FIND command to the next FIND command.
In this query history it is possible to navigate forward and backward with the commands BACK and FORWARD.
After a call to this function a call can be done to the function **BCFetchQueryHistory()** to obtain information about each line in the query history.
The information that is returned about the current query history is: total number of hits for the query and current result after the query.
Normally those results are alike (total and current), but by navigate with the commands BACK and FORWARD they can be different.
In the document **Operations Guide** there are some examples on the commands BACK and FORWARD and how they will affect the query history.
Note, this information is only available if the *queryhistory* is enabled via **BCSetSessionInfoXml().** If *queryhistory* is not enabled an error code is return when calling this function.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:
**BCFetchQueryHistory(), BCSetSessionInfoXml(**), QL-language BACK and FORWARD.

Example:
Get information about current Query History.

```
int32 i;
BCQHistoryInfo_t queryHistory;

// Information about current Query History for table
// "Company"
BCGetQueryHistoryInfo(mClient, "Company", &queryHistory);

// Total number of query history lines and the current
// number of lines might be used to display the Query
// History via function BCFetchQueryHistory().
```

BCGetRankMode()

BCGetRankMode(**BWClient** client, **const char** *table,
    int32 *rankMode)

Get current order of the retrieved lines.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- requested Table
int32 *rankMode	- current order

The name table is the name of the requested Table.
Current order will be stored in the rankMode,
The order of the result depends on the Query, what index method used for the columns involved in the query and if any sort command has been performed.
Following orders are available:

- order by frequency of query terms
- order by frequency of number of query terms
- order by weighted frequency of query terms
- order by weighted frequency of number of query terms
- order by sort of content
- order by similarity

A closer description of all orders are described in **softbool.h**.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCSetRankMode().

BCGetSessionInfo()

BCGetSessionInfo(**BWClient** client, **BCSessionInfo_t** *info)

Get settings for current session.

Parameters:

BWClient client	- the Boolware client instance
BCSessionInfo_t *info	- information about the session

In the info all settings about current session are stored.
Settings returned are:

- a text identifier to identify the session
- current attached Index if any
- current selected Table if any
- TRUE, if automatic right truncation is active
- value for word distance for proximity search
- TRUE, if order is requested for proximity search

To be able to add more types of settings without modify the functions **BCGetSessionInfo/ BCSetSessionInfo** there is an extended version of the function with the suffix **Xml**. The extended version handles language and query history for a session.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:
**BCSetSessionInfo (), BCGetSessionInfoXml(), BCSetSessionInfoXml(**).

Example:
Get session settings.

```
BCSessionInfo_t sessionInfo

// Get settings for current session
BCGetSessionInfo(mClient, &sessionInfo);
```

BCGetSessionInfoXml()

BCGetSessionInfoXml(**BWClient** client, **char** *info,
                         const int32 sz)

Get settings in XML format.

Parameters:

BWClient client	- the Boolware client instance
char *info	- area to store settings in XML format
const int32 sz	- size of info buffer

Xml tagged settings will be stored in the info buffer about the session.
A detailed description of the xml elements is found under the head line, **XML elements for session settings**, in this document.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCGetSessionInfo(), BCSetSessionInfo(), **BCSetSessionInfoXml()**

XML elements for session settings.

Example:

Get current session settings

```
char info[512];
BCGetSessionInfoXml(mClient, &info, sizeof(info));
```

BCGetStatistics()

BCGetStatistics(**BWClient** client, **const char** table, **const char** column,
const int32 value,
BCStatisticsInfo_t *info)

Get statistics for the requested column. The values that are fetched are calculated from the current query result.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- requested Table name
const char *column	- requested column name
const int32 groups	- upper and lower group value requested
BCStatisticsInfo_t *info	- area to store the values

The parameter *table* and *column* requests from what Table
and upon what column to perform the calculation for the statistics.

In the parameter *groups* a value, tell what part of the result that is valid for the upper and lower limit values; e.g. 4, select upper and lower quartile, but 5 selects the upper and lower quintile. Valid values for groups are 3 - 8.

In the *info* all statistic values are stored for the requested column. Only the current query result will be part of the statistics.

Statistic values that are calculated: number records for the statistics, the summary of all values, arithmetic average, min value, max value, standard deviation, variance, median, most frequent value, number of most frequent value, upper and lower limit value for selected group.
For a more detailed description of the different values see the description of the **BCStatisticsInfo_t**.

To get all limit values for a specified group you should use the **BCExecute**. The command is described in Chapter 1 section "Execute commands in Boolware".

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCStatisticsInfo_t()

Example:
Retrieve all companies in the London area and get statistics about the ‘Turnover’. The upper and lower group limit is the quintile values.

```
BCStatisticsInfo_t statInfo;
int32 groups = 5, result;

rc = BCQuery(mClient, "Companies", "FIND City:London", &result, NULL);
BCGetStatistics(mClient, "Companies", "Turnover", groups, &statInfo);
```

BCGetTableInfo()

BCGetTableInfo(**BWClient** client, **const int32** num,
BCTableInfo_t *info)

Get information about a Table connected to a Boolware Index.

Parameters:

BWClient client	- the Boolware client instance
const int32 num	- relative table number
BCTableInfo_t *info	- record to store table information

The parameter *num* is the relative number of the requested Table.
In the record *info* information about the Table will be stored. The information is: the name of the Table, current status and the last query result for this table.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCConnect(), BCAttach(), BCGetNumberTables()

Example:
Connect to the server ‘Charlie’ and attach to the Boolware index ’Company’. Get all tables available in the Boolware Index and write them to the screen.

```
BCTableInfo_t tabInfo;
int32 num, i;

BCConnect(mClient, "Charlie", "");
BCAttach(mClient, "Company");
BCGetNumberTables(mClient, &num);

for(i = 0; i < num; i++)
if(BCGetTableInfo(mClient, i, &tabInfo) == 0)
        cout << tabInfo.tabName << endl;
```

BCGetTableInfoByName()

BCGetTableInfoByName(**BWClient** client, **const char** *tableName,
BCTableInfo_t *info)

Get information about a Table connected to a Boolware Index.

Parameters:

BWClient client	- the Boolware client instance
const char *tableName	- the name of the table
BCTableInfo_t *info	- record to store table information

The parameter *tableName* is the name of the requested table.

In the record *info* information about the Table will be stored. The information is: the name of the Table, current status and the last query result for this table.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCConnect(), BCAttach(), BCGetNumberTables()

Example:
Connect to the server 'Charlie' and attach to the Boolware index 'Company'. Get information about the table 'CompanyInfo'.

```
BCTableInfo_t tabInfo;
int32 num, i;

BCConnect(mClient, "Charlie", "");
BCAttach(mClient, "Company");

for(i = 0; i < num; i++)
if(BCGetTableInfoByName(mClient, "CompanyInfo", &tabInfo) == 0)
      cout << tabInfo.tabName << endl;
```

BCGetVersion()

BCGetVersion(**BWClient** mClient, **char** *msg,
    const int32 buffSz)

Get the current version of the Boolware Client and Boolware Server.

Parameters:

BWClient client	- the Boolware client instance
char *msg	- buffer to store version
const int32 buffSz	- size of buffer

Current version for the Boolware Client and Boolware Server is stored in *msg*.
To be able to get the version of the Boolware Server, the client must be connected to a Boolware Server (see **BCConnect()**).

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCConnect()

Example:
Get version information string

```
char verStr[256];
BCGetVersion(mClient, verStr, sizeof(verStr));
verStr has the following contents:
Boolware client version: major.minor.release.build CR/LF
Boolware server version: major.minor.release.build
where major, minor, release and build are digits.
Example:
Boolware client version: 2.1.0.4 CR/LF
Boolware server version: 2.1.0.12
```

BCQuery()

BCQuery(**BWClient** client, **const char** *table,
  const char *str,
  int32 *result,
  float *qtime)

Perform a Query.

Parameters:

BWClient client	- the Boolware client instance
const char *table	- requested table
const char *str	- the Query, QL-string
int32 *result	- buffer to store the query result
float *qtime	- buffer to store the query time, NULL if not requested

table is the name of the requested table.
The *str* is the actual Query. A detailed description of the query language is described under a special head line in this document.
Number of found records is stored in the parameter *result*.
The actual query time is stored in the parameter *qtime* if supplied
In the description of the **Query Language** – with examples – how to obtain the Boolware query facilities.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCConnect(), BCAttach()
QL "Softbool Query Language"

Example:
Get number of records in the table 'Employee' that contains the name 'Bob' in the 'Name' column and the city 'London' in the 'City' column in the 'Company' database.

```
int32 rc, result;

// Connect to Server
BCConnect(mClient, "192.168.0.1", "");

// Attach to the index 'Company'
BCAttach(mClient, "Company");

// The query
rc = BCQuery(mClient, "Employee", "FIND Name:bob AND City:London", &result, NULL);
```

BCReconnectIfExists()

BCReconnectIfExists(**BWClient** client,
const char *srv,
const char *sessName)

Connect the current session to Boolware server on requested computer.

Parameters:

BWClient	client	- Boolware client instance
const char	*srv	- name of server or IP address, where Boolware server executes
const char	*sessName	- the name of the session to be connected

Boolware Server must be running on a computer in the network, which could be reached by the client before the connection could take place.

The parameter '*srv*' should be the name of the computer in the network or its IP-address; for example 192.168.0.1.

Note that the computer the client is running on must be able to access the server where Boolware is installed.

Return:
- SOFTBOOL_OK upon success
- Otherwise error code

See also:
BCConnect(), BCDisconnect()

Example:
Connect a current session "Bob" to the server "Charlie".

```
char msg [256];

// Connect if exists otherwise create session "Bob" it
if (BCReconnectIfExists(mClient, "Charlie", "Bob") != SOFTBOOL_OK)
   if (BCConnect(mClient, "Charlie", "Bob") != SOFTBOOL_OK)
      BCGetErrorMsg(mClient, BCGetErrorCode(mClient), msg, sizeof(msg));
```

BCSetFetchSize()

BCSetFetchSize(**BWClient** client, **const int32** fetchSize)

Set number of rows to fetch from the data source in each call.

Parameters:

BWClient	client	- the Boolware client instance
const int32	fetchSize	- number of rows to fetch

By fetching more than one row at a time the performance will increase. Available value is within the interval 1 - 500. If the specified value is less than 1 it will be reset to 1. If the specified value is greater than 500 it will be reset to 500.

Be aware of that a high value will increase the download of the network and increase the memory allocation. The number of records will be fetched from the data source before anything is sent back to the client and this can give a slow impression.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCFetchRow()

Example:

```
Set fetch size to 25, a common result table.
BCSetFetchSize(mClient, 25);
```

BCSetRankMode()

BCSetRankMode(**BWClient** client, **const char** *table,
    const int32 mode)

Change the order for the result.

Parameters:

BWClient client		- the Boolware client instance
const char	*table	- requested table
const int32	mode	- requested order

table is the name of the requested table.
The different orders depends on how the query is performed, what index methods that are used for the queried columns and if any sort has previously been done.
Following orders are available:

- order by frequency of query terms
- order by frequency of number of query terms
- order by weighted frequency of query terms
- order by weighted frequency of number of query terms
- order by sort of content
- order by similarity

A closer description of all orders are described in **softbool.h**.
If no query result is available this function has no effect, there is nothing to put in a certain order.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:
    BCQuery()

Example:
Set rank mode to occurrence order for table "Employee". The result is now obtained in the order of records that contains most query terms will come first.

```
BCSetRankMode(mClient, "Employee", BOCCRANK);
```

BCSetSessionInfo()

BCSetSessionInfo(**BWClient** client**, const BCSessionInfo_t** *info)

Change settings for current session.

Parameters:

BWClient client	- the Boolware client instance
const BCSessionInfo_t *info	- record containing settings.

info contains all settings that for the current session.
Following settings are available:

- automatic right truncation ON/OFF
- word distance for proximity search
- word order for proximity search ON/OFF

To be able to add more types of settings without modify the functions **BCGetSessionInfo/ BCSetSessionInfo** there is an extended version of the function with the suffix Xml. The extended version handles language and query history for a session.

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCGetSessionInfo(), BCGetSessionInfoXml(), BCSetSessionInfoXml()

Example:
Set automatic right truncation ON.

```
BCSessionInfo_t info;

// Get current settings
BCGetSessionInfo(mClient, &info);
info.autoTrunc = 1;

// Set new settings for session in Boolware Server
BCSetSessionInfo(mClient, &info);
```

BCSetSessionInfoXml()

BCSetSessionInfoXml(**BWClient** client, **const char** *info)

Change settings for session via XML.

Parameters:

BWClient client	- the Boolware client instance
const char *info	- buffer containing XML elements info contains all settings to change for the session.

A detailed description of the XML elements are described under the head line, **XML elements for session settings,** in this document.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCGetSessionInfo(), BCGetSessionInfoXml(), BCSetSessionInfo(),
XML elements for session settings.

Example:

Set query history active.

```
char info [512];
strcpy(info, "<softbool><session>< queryhistory all="1"/></session></softbool>");
BCSetSessionInfoXml(mClient, &info);
```

BCSort()

BCSort(**BWClient** client, **const char** *table,
  const char *expression)

Sorts the query result by given column(s).

Parameters:

BWClient	client	- the Boolware client instance
const char	*table	- requested table
const char	*expression	- column(s) to sort

expression syntax:

```
<colname> [asc/desc[:nn]] [emptydata='first/last'] [sortalias='col1, col2'] [,]
```

where :

colname	the column name to perform the sort on.
optional:	
asc/desc	ascending or descending; default is ascending
:nn	sort the *nn* first at each sort request
emptydata	*first*/*last* set fictive sort order if no data in column *first* indicates that the empty value will be treated as sort value ascii 0 *last* indicates that the empty value will be treated as sort value ascii 255 default is *last*

sortalias	upon empty data in *colname* use another column to collect data that will be used for sorting.
	Up to 5 sortalias columns, comma separated, can be given i.e. if *col1* is empty try next specified column *col2* etc.
	If column name needs quotation marks make sure to double quote if using the same quotation mark as araound the whole *sortalias* expression. E.g.
	`sortalias='''Col 1'', ''Col 2'''`
	`sortalias="'Col 1', 'Col 2'"`

,	separates multiple sort columns

Perform a sort and the primary keys or rows will be fetched in sorted order.

The format of the parameter expression is: column name followed by order type, 'ASC', ascending, or 'DESC', descending. By the parameter emptydata=first/last you could control where to "sort" records that do not contain any data in the sort column; first or last. Separate columns by a comma sign.

Default order type is 'ASC' and can be omitted. Default for emptydata is last.

If the first Column to sort on is indexed as numeric or string, you could take advantage of the fast Boolware incremental Index Sort. By specifying the number of records to be sorted after the column information you tell Boolware to sort just that number of records; it is not necessary to sort a great number of records if you just want to present the top twenty.

The format of the parameter expression when incremental sort: column *order:n*, where *order* is asc or desc and *n* a number.

Note It is only the Boolware incremental Index Sort that could take advantage of the specified number of records to be sorted.

The type of sort algorithm chosen is determined by Boolware. See Chapter 11 Interactive Query section "Rank by Sort" in the Boolware Operations Guide for detailed information.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

Example 1:
Order current result in the table ‘Employee’ in column sort order: ‘Name’ ascending and ‘Age’ descending. Get primary keys in sorted order; the primary key is the column ‘ID’.

```
int   rc, i, recNo, rankMode;
float score;
char  key [128];

// Sort by name ascending and age descending
   rc = BCSort(mClient, "Employee", "Name, Age DESC emptydata=first");

// Get PK in sorted order
if(rc == SOFTBOOL_OK)
   {
   // Get PK for the 25 first records
   for(i = 0; i < 25; i++)
      {
      BCFetchKey(mClient, "Employee", "ID", i, key, sizeof(key), &score, &recNo,
&rankMode);
      // Store PK in a list for later use pkList->Add(key);
      }
   }
```

Example 2:
A query has generated 2.725.389 hits and you want to get the 25 persons with the highest salary.
Sort the result in the table ‘Employee’ on the column ‘Salary’ descending (the highest salary at top). Fetch the primary keys for the first 25 records.

As the column ‘Salary’ is indexed as numeric the Boolware incremental Index Sort will be used. Specify in the sort command that only 25 records should be sorted.

This split into two parts: in the first part you specify the query and determines the sort attributes, while you in the second part browse in the result set. This is to show that Boolware only "sort" when it is necessary to save system resources.

Part 1:

Perform query and determine sort attributes:

```
int   rc, i, recNo, rankMode;
float score;
char  key [128];

// The query
rc = BCQuery(mClient, "Employee", "FIND City:London", &result, NULL);

// Sort by salary descending; only sort the first 25 records
rc = BCSort(mClient, "Employee", "Salary DESC:25");

// Get PK in sorted order
if(rc == SOFTBOOL_OK)
   {
   // Get PK for the 25 first records
   for(i = 0; i < 25; i++)
      {
      BCFetchKey(mClient, "Employee", "ID", i, key, sizeof(key), &score, &recNo,
&rankMode);
      // Store PK in a list for later use pkList->Add(key);
      }
   }
```

Part 2:

Continue to fetch records 75 to 100:

```
int   rc, i, recNo, rankMode;
float score;
char  key [128];

// Note that you do not need to do any query or sort this time; Boolware will
automatically
// sort the records that need to be sorted.

// Get PK for the records 75 - 100 in sorted order
for(i = 74; i < 100; i++)
   {
   BCFetchKey(mClient, "Employee", "ID", i, key, sizeof(key), &score, &recNo,
&rankMode);
   // Store PK in a list for later use pkList->Add(key);
   }
```

Note that no query or sort is performed; Boolware will automatically sort the records that need to be sorted. In this case Boolware will sort records 26 -100; the records 1 - 25 was already sorted in Part1.

BCStartFetchTerm()

```
BCStartFetchTerm(BWClient client,
                         const char       *table,
                         const char       *col,
                         const BCTerm_t   *startTerm,
                         const bool       zoomed)
```

Prepare Boolware Server to fetch index terms.

Parameters:

BWClient	client	- the Boolware client instance
const char	*table	- requested table
const char	*col	- requested column
const BCTerm_t	*startTerm	- start term
const bool	zoomed	- TRUE if only terms within current result

table contains the requested table name and col contains requested column for the index terms.

The parameter *col* could have some more information: type and order. Type could be *term* or count; if no type is specified *term* will be in effect. *term* means that the index terms should be fetched in alphabetical order while *count* means that they should be fetched in frequency order. Frequency in this context means the occurrence in number of records in the table. The order could be ascending, *asc*, or descending, *desc*. Order has only meaning when the type is *count*; when the type is *term* the terms will always be fetched in ascending alphabetical order. Subzoom is activated by using more than one column in *col*, separated with parenthesis between levels, and comma within the same level. Refer to section about subzoom in "Operations Guide".

Choose start position in the index tree by startTerm. Empty string starts from the beginning of the index tree. You could specify a term, a term number or an index type depending on the index function chosen.

The parameter startTerm->term could hold a sub-command, where you specify the type of index term to fetch; word, string, phonetic etc. These sub-commands are the same as those used in the Boolware Query Language when you want to query a special index type. Valid sub-commands are: string(), reverse(), sound(), stem(), frequency() and searchterms(). If the sub-command searchterms is used there must be a special table designed for statistics on Query terms (see detailed description in "Operations Guide"). In the structure **BCTerm_t** the element termType you could set the index type instead of using the sub-command. See chapter "Execute commands in Boolware" and command "indexex" for a list of valid values.

If *zoomed* is activated, TRUE, only terms within the current searched result will be fetched.

In the manual "Operations Guide" Chapter 11 "Interactive Query" you could read about frequency index in section "View Frequency Index" and about Query term statistics in section "Statistics on Query Terms".

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

See also:

BCFetchTerm()

Example 1:
Get 25 terms and start from the term nearest "Charlie" in the column of ‘Name’ in the table ‘Employee’. Get all terms not just zoomed ones.

```
BCTerm_t term;
int      i = 0;
```

```
// Initiate term to start from
strcpy(term.term, "Charlie");
term.termtype = 1; // Indexing method is word

// Initiate fetch of terms
BCStartFetchTerm(mClient, "Employee", "Name", &term, false);

   // Get next 25 terms
   while(i++ < 25)
      {
      // Get next term; break if no more terms
      if(BCFetchTerm(mClient, &term) != SOFTBOOL_OK)
         break;

      // Print the hit count and the term
      print term.hits;
      print term.term;
      }
```

Example 2:
Get the 25 most used Query terms during the period 11.00 - 13.00 the 16th of June 2005. The Query terms should be presented in descending order; the most common Query terms first. In this case it is word (not stings) that should be fetched. The special table containing the Query terms for statistics is named QueryTerms and the column holding the query terms is called Terms.

```
BCTerm_t term;
int      i = 0;

// Initiate the time interval from which the statistics should be fetched
strcpy(term.term, "searchterms(20050616 11:00..20050616 13:00)";
term.termtype = 1; // Indexing method is word


// Initiate fetch of terms
BCStartFetchTerm(mClient, "QueryTerms", "Terms", &term, false);

   // Get the 25 most common query terms for the specified period
   while(i++ < 25)
      {
      // Get next term; break if no more terms
      if(BCFetchTerm(mClient, &term) != SOFTBOOL_OK)
         break;

      // Print the hit count and the term
      print term.hits;
      print term.term;
      }
```

Example 3:
Get the 25 most common terms and present them in descending order; the most common terms first. Only terms with the indexing method word should be fetched. The table, Articles, from which the terms should be fetched contains 10.000.000 records. The column you are interested in, Text, contains 20.000.000 unique terms. To make the extract of terms much more efficient you should limit the extract to terms that are contained in more than 50.000 records.

```
BCTerm_t term;
int      i = 0;

// Set limit
strcpy(term.term, ">50000";
term.termtype = 1; // Indexing method is word

// Initiate fetch of terms
BCStartFetchTerm(mClient, "Articles", "[Text] count desc", &term, false);
```

```
   // Get the 25 most common terms
   while(i++ < 25)
      {
      // Get next term; break if no more terms
      if(BCFetchTerm(mClient, &term) != SOFTBOOL_OK)
         break;

      // Print the hit count and the term
      print term.hits;
      print term.term;
      }
```

Example 4:
Get number of phonetic terms that are stored in the column Name in the table Companies.

```
BCTerm_t term;

// Initialize to get number of phonetic terms
strcpy(term.term, "4"); // Index type phonetic
term.termtype = 18; // Index function; get number of terms for specified Index
type

BCStartFetchTerm(mClient, "Companies", "Name", &term, false);

   // Get number of phonetic terms in column Name; there is only one entry to
fetch
   if(BCFetchTerm(mClient, &term) != SOFTBOOL_OK)

   print term.hits;
   print term.term; // Specifies the Index type
```

Example 5:
Get 25 strings starting from term number 12.000.

```
BCTerm_t term;
int      i = 0;

// Initialize fetching strings starting from string number 12.000
strcpy(term.term, "12000";
term.termtype = 23; // Index type string

// Initialize fetching strings
BCStartFetchTerm(mClient, "Artiklar", "[Text] count desc", &term, false);

   // Get the 25 requested strings
   while(i++ < 25)
      {
      // Get the next string
      if(BCFetchTerm(mClient, &term) != SOFTBOOL_OK)
         break;

      print term.hits;
      print term.term;
      }
```

Important: After the function SubZoom was implemented all column names containing parentheses must be enclosed within quotation marks. See also Operations Guide chapter 11 "Interactive Query" section "Hits projected hierarchically over one or multiple other values (SubZoom)".

BCXmlRequest()

BCXmlRequest(**BWClient** client, **const char** *request,
BCXmlReply_t *reply)

Request and answer in xml-format.

Parameters:

BWClient client	- the Boolware client instance
const char *request	- the xml request
BCXmlReply_t *reply	- pointer to the xml reply and length of the reply

This function will execute a whole series of commands specified in the xml request.
The request is formed by different xml elements and their attributes, see chapter 3 XML API for a detailed description of the xml elements.

Return:
- SOFTBOOL_OK upon success
- Error code otherwise

See also:
BCConnect ()

Example:
Get number rows containing the name 'Bob' and lives in the city of 'London' in the table 'Employees' in the database 'Company'. Get the column data for 'Name' and 'City' from the first two records found. Maximum 50 characters from each column.

```
int32 xmlReq[512], rc;
BCXmlReply_t reply;

// Connect to Boolware Server with session name 'xml'
BCConnect(mClient, "192.168.0.1", "xml");

// Build the xml request
strcpy(xmlReq,
"<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?> \
<SoftboolXML_requests> \
<SoftboolXML_request type=\"query\">\
<open_session name=\"\" queryhistory=\"0\"/>\
<database name=\"Company\"/>\
<table name=\"Employees\"/>\
<query> FIND Name:Bob AND City:London </query>\
<response type=\"\" href=\"\" queryhistory=\"0\">\
 <records from=\"1\" count=\"2\" maxchars=\"50\">\
  <field name=\"Name\"/>\
  <field name=\"City\"/>\
 </records> </response>\
</SoftboolXML_request> \
</SoftboolXML_requests>");
// Send the request to Boolware Server
rc = BCXmlRequest(mClient, xmlReq, &reply);

// Reply contains now a pointer to the reply and the length of the reply
<?xml version="1.0" encoding="iso-8859-1" ?>
<SoftboolXML_responses>
<SoftboolXML_response type="query" error_code="0">
<session>xml</session>
<records total="19" from="1" to="2" rank="no rank">
<record score="1.000">
 <field name="Name">Smith Bob</field>
 <field name="City">LONDON</field>
</record>
<record score="1.000">
 <field name="Name">Jonson Bob</field>
 <field name="City">LONDON</field>
</record></records></SoftboolXML_response></SoftboolXML_responses>
```

BCXmlRequestNoResponse()

BCXmlRequestNoResponse(**BWClient** client, **char** *request)

Performs an XML request with an automatic connection to Boolware server.

To fetch the retrieved data the method **BCFetchRow** should be used.

Parameters:

BWClient client	- The Boolware client instance
*const char *reques t*	- The XML request

Return:

- SOFTBOOL_OK upon success
- Error code otherwise

XML elements for session settings

Each session can have their own unique settings concerning:
1. Language
2. Automatic right truncation
3. Word distance for proximity search
4. If terms has to occur in specified order for proximity search (0 = order; 1 = ignore order)
5. Threshold for similarity search
6. If user-written plugins should be used when generating search terms
7. Activate or deactivate the Query History
8. Type of encoding; valid values are: ANSI and UTF-8

Language can be selected by the application to select messages in different languages for different users.

Automatic right truncation means that all query terms given will be truncated i.e. all terms that begin with the given term will match the query. E.g.: FIND car will match carport, cars, cart etc.

Word distance for proximity search. This means that the sub command near can specify number of "other" words between the ones in the actual query. E.g. FIND *near*(tiny cottage,3,0) means, two (2) words might exists between tiny and cottage and still get a match. Records like the following are allowed: "*tiny red cottage*", "*cottage that is tiny*", "*tiny cottage*", but "*tiny boy and red cottage*" will not be allowed, there are three (3) "other" terms between the specified ones.

If terms has to come in a specified order means that the terms has to appear in that order in the record to give a match.
E.g. FIND near(tiny cottage,3,**1**) will match the following: "*tiny cottage*", "*tiny red cottage*", "*tiny old red cottage*", but the following will not match: "*cottage is tiny*".

A special case is when search as a string:
FIND near(tiny cottage,1,1) will only match when "*tiny cottage*" appears in the text.

Similarity search means that the query is not done for specific terms but the query is done for the content. After a query the result can be as much as 80 –90 % of the database. The records are sorted by its similarity due to the reference text. The records that will appear at the end of the list are not similar to the reference text at all – maybe just 0.002 -, but will be retrieved anyway. To avoid records that has no similarity with the reference text specify a threshold value between 0.0 and 1.0. All records with lower score than the threshold value will be treated as no match and will not be part of the result.
E.g. Threshold is set to 0.5.
FIND *sim*(reference text...), ignore all records below the score of 0.5000.

By setting the element *indexexit* you could activate a user-written plugin that generates additional search terms. These terms together with the search terms generated by Boolware will be used when searching.
E.g. FIND SL500, if you only use Boolware generated search terms you will only find records containing the word SL500; not SL 500 or 500 SL. A user-written plugin could generate additional search terms: SL and 500, which means that you will find records containing the words: SL500, SL 500 or 500 SL. Moreover you could specify you query in three different ways and get the very same result: FIND SL500, FIND SL 500 or FIND 500 SL. If you only want records containing the word SL500 you just turn this element off.

Query history means that result of all queries between the FIND command will be stored in the server. Within the query history it is possible to navigate with the commands BACK and FORWARD. This option will bind extra resources to the session and therefore it can be turned ON or OFF.

By specifying the encoding element you could tell Boolware how to communicate with the application; if you specify ANSI, Boolware expects that everything coming from the application is in ANSI and everything that leaves Boolware is in ANSI. The same thing is valid when the encoding is UTF-8. If no encoding is specified, ANSI will be used.

In the document **Operations Guide** the functions *sim*, *near*, *BACK* and *FORWARD* are describe with examples.
In the future it is possible that further settings will be introduced. To achieve a flexible way of handling session settings in the system XML element will be used for new settings.

The following XML elements are available:

```
<softbool>
<session name=’’ lang=’’ encoding=’’ sessiontimeout=’’ maxexecutiontime=’’>
 <languages/>
  <search trunc=’’ proxgap=’’ proxorder=’’ vsmthreshold=’’ indexexit=’’ strictasis=’’ />
  <queryhistory all=’’/>
  <database name=’’>
   <table name=’’/>
   </database>
 </session>
</softbool>
```

Attribute for the session element

name	Session name read only
encoding	Requested character encoding
lang	Current language code
sessiontimeout	The current session timeout
maxexecutiontime	The current execution timeout

Data for the language element

All possible language codes separated by blank read only

Attribute for search element

trunc	0 truncation off and 1 for truncation on
indexexit	1 means that the corresponding plugin function will generate additional search terms. 0 means that only search terms that are part of the text should be used.
proxgap	Word distance for proximity search
proxorder	Word order for proximity search. 0 any order 1 given order
vsmthreshold	Lowest threshold value. A value between 0.0 – 1.0
strictasis	1 means that strict asis is active and will treat ordinary wildcard characters, ‘*’, ‘?’, ‘!’ and ‘#’ as ordinary characters in the subcommands wordasis() and stringasis(). Default is 0; not active.

Attribute for the queryhistory element

all	0 query history off or 1 query history on

Attribute for the database element

name	Name of attached Boolware Index read only

Attribute for the table element

name	Name of table(s) for attached Boolware Index read only

Example:

1. Set encoding to UTF-8 and error message in English
2. Set right truncation on, proximity gap three words and words in order
3. Activate query history

```
bw_set_settings_xml($link, "<softbool>
        <session encoding=\"UTF-8\" lang=\"en\">
        <search trunc=\"1\" proxgap=\"3\" proxorder=\"1\"
                strictasis=\"0\"/>
        <queryhistory all=\"1\"/></session>
        </softbool>");
```

Chapter 3
XML/JSON API

This chapter describes in detail how to communicate with Boolware Server via a XML/JSON protocol. Here it is described how to achieve the functionality described in chapter 2.

Introduction

To achieve a more flexible communication with Boolware Server a XML/JSON protocol is available. For a complete description see the *XMLSchema* and *JSONSchema* in the install directory.

This protocol can be seen as a **complement** to the functional API described in chapter 2.

This is a strict script based API based upon XML/JSON elements. For JSON-requests the API function execute should be used.

The protocol is divided into two parts: request and response.

The request is actually built up by two parts: what to do and how to receive the result.
In the first part of the request you tell the system what you want to do: search via Boolware Index server, list index terms etc.

In the second part of the request (element *'response'*) you describe how the result should be obtained: how many records, what record to start from, in what order, query history, statistics etc.

The response is a XML/JSON document were different elements and their attributes contains different parts of the response.

Although the Boolware system is based upon an interaction between the client and the server it is important for a user to be able to do a refinement of a query or listing the result from an earlier query.

By a unique session identifier which is either specified by the user or by the system, a unique session is created within the Boolware Server. This session identifier is always returned in the element '*session'* in the response and can then be used in the attribute '*name'* for the element '*open_session*' in a later request or query refinement.

This is called a session in the Boolware Server and makes it possible for a user to stay alive in the Boolware Server even if the physical connection is closed by the browser.
Depending on the connection to the Boolware Server the first connection can take some time.

The documentation has the following structure:

1. Description of the request; all elements and their attributes are described with short examples. For a complete description of all elements, see the XMLSchema or JSONSchema files.

2. Description of the response; all elements and their attributes are described with short examples.

3. Complete examples how to use this API in different cases.

How to call via "web" (http)

The protocol supports the WEB-server IIS 5.0 and later for the Windows platform and Apache 2.x for Linux.

The request follows the standard HTTP protocol for a POST command, (GET is ignored with an error message), which makes it easy to write effective scripts to take advantage of the search capabilities in the Boolware Server.

The default number of xml clients that could be connected at the same time is 100. This value could be changed via the environment variable **bwxmlclients** to a suitable value.

The web server must be stopped and started before the new value takes effect.

Note:
All elements and their attributes must be written and used as described below. The elements are case sensitive, so be careful with upper and lower case letters.

For XML, some characters are reserved by XML, and must be coded as so called "entities". Those are: amp (**&**), less than (**<**), greater than (**>**), quotation mark (**"**) and apostrophe (’). When these letters occur in the text they have to be coded as follows: **&** -> *&amp*;, **<** -> *&lt*;, **>** -> *&gt*;, **"** -> *&quot*; and **’** -> *&apos*;.

It is not always necessary to specify both the query and the response in a request. Sometimes only a query and the number of hits is of interest to be able to refine the query, there is no need for the response part (element *’response’*). Sometimes just a list of records is of interest. You do not need to supply the query (element ‘*query’*) and the sort (element ‘*sort’*) again to get the next part of the result list.

Mandatory elements - except the XML header for XML request – are: *’SoftboolXML_request'*, ’*database*’ and ’*table*’.

XML Schema definition Softbool_XMLRequests/Softbool_XMLResponses

When installing Boolware you will receive two XML schema definition files which describes all elements that are defined in a Boolware XML request and Boolware XML response. The files Softbool_XMLRequest.xsd and Softbool_XMLResponse.xsd is stored in XMLSchema which is a subdirectory to the directory where the Boolware Clients are stored.

JSON Schema definition SoftboolJSON_requests/SoftboolJSON_responses

When installing Boolware you will receive two JSON schema definition files which describes all properties that are defined in a Boolware JSON request and Boolware JSON response. The files SoftboolJSON_requests.json and SoftboolJSON_responses.json is stored in JSONSchema which is a subdirectory to the directory where the Boolware Clients are stored.

These two schemas are tested against NJSONSchema (for C#) and jsonschema2pojo (for Java) to generate classes from each schema. **Note** that all attributes are prefixed with a '@' in the JSON-request/JSON-response and the content of a element when attributes exists are stored in the element *'#text'*.

For NJSONSchema (C#), you need to adjust the name of variables generated according to the example below:

```
// Dervied schema property name generator
public class CMySharpPropertyNameGenerator : CSharpPropertyNameGenerator
{
  public override string Generate(JsonProperty property)
  {
    var str = base.Generate(property);
    return str.Replace("#", "__");  // replace # (e.g. #text) to double
underscores
  }
}

public void RequestSchema2Classes()
{
  // Read schema from file
  var schema = JsonSchema4.FromFileAsync("SoftboolJSON_requests.json").Result;

  // Setup namespace and create my own property name generator
  var settings = new CSharpGeneratorSettings();
  settings.Namespace = "boolwarejsonrequest";
  settings.PropertyNameGenerator = new CMySharpPropertyNameGenerator();

  // Create "schema to class" generator
  var generator = new CSharpGenerator(schema, settings);

  // Generate classes
  var file = generator.GenerateFile();
}
```

All the examples below refers to XML-request and XML-response but the same can be done with JSON-requests and the response will default be JSON-responses. One important issue is the encoding of the request. In the XML-request there is a header that tells the encoding format e.g. `<?xml version="1.0" encoding="iso-8859-1"?>` this indicate that the entire request is encoded in "ISO-8859-1" which also is the default value for a XML-request. But in a JSON-request the default value is "UTF-8". This can be given in an attribute on the first element in the request and is named *'@encoding'* if it differs from "UTF-8".

Request

XML

Every request must start with the standard XML head:
<?xml version="1.0" encoding="iso-8859-1"?>

The only approved encodings are: **UTF-8** and **iso-8859-1**.
The request and the response will always have the same **encoding**.

A request contains different XML elements with their attributes.

```
<SoftboolXML_requests>
  <SoftboolXML_request type=query/execute/index/metadata>
  <dtdref type= name= sysid= pubid= >
    <open_session name=(string)>
    <database name=(string) dsn=(string)>
    <table name=(string)>

    <index field= start_position= max_terms=(int) zoom=(no/yes) type= >
    <query [flow=(name)] [qtime=0/1]>
      <(fieldname)>text</(fieldname)> / (QL-string)
    </query>
    <metadata database= table= field= >
    <execute></execute>

    <response type= href= queryhistory=1/0 [raw= fieldsep= rowsep= quotes=]>
      <sort expression=(string)>
      <records from=(int) count=(int) rank=(int) maxchars=(int)>
        <field name=(string) formula=(string)/>
      </records>
    <simvectors from=(int) count=(int) content= />
    <statistics column= group= onall= getallvalues= >
    </response>

    <flow>
    ...
    </flow>
  </SoftboolXML_request>

  <SoftboolXML_request type=query/execute/index/metadata>

  ...    another request

  </SoftboolXML_request>

</SoftboolXML_requests>
```

All XML-elements in a request are described in the file Softbool_XMLRequest.xsd which is described above.

JSON

The only approved encodings are: **UTF-8** and **iso-8859-1**.
The request and the response will always have the same **encoding**.

A request containing different JSON elements and their attributes:

```
{
    "@encoding": "iso-8859-1",
    "SoftboolJSON_request": [
      {
        "@type": "query/execute/index/metadata",
        "dtdref": {
          "@type": "",
          "@name": "",
          "@sysid": "",
          "@pubid": ""
        },
        "open_session": {
          "@name": "(string)"
        },
        "database": {
          "@name": "(string)"
        },
        "table": {
          "@name": "(string)"
        },
        "index": {
          "@field": "",
          "@start_position": "",
          "@max_terms": "(int)",
          "@zoom": "(no/yes)",
          "@type": ""
        },
        "query": {
          "@flow": "(name)",
          "@qtime": "0/1",
          "fieldname": "text"
        },
        "metadata": {
          "@database": "",
          "@table": "",
          "@field": ""
        },
        "execute": {
          "#text": ""
        },
        "response": {
          "@type": "",
          "@href": "",
          "@queryhistory": "1/0",
          "@fieldsep": "",
          "@rowsep": "",
          "@quotes": "",
          "sort": {
            "@expression": "(string)"
          },
          "records": [{
            "@from": "(int)",
            "@count": "(int)",
            "@rank": "(int)",
            "@maxchars": "(int)",
            "field": [{
              "@name": "(string)",
              "@formula": "(string)"
            }]
          }],
          "simvectors": {
            "@from": "(int)",
            "@count": "(int)",
            "@content": ""
          },
          "statistics": [{
            "@column": "",
            "@group": "",
            "@onall": "",
            "@getallvalues": ""
          }
        }],
        "flow": {
```

```
            ...
          }
        },
        {
          "@type": "query/execute/index/metadata"

           ... another request
        }
      ]
}
```

All JSON-elements in a request are described in the file SoftboolJSON_request.json which is described above.

What to do in the request

In this part of the request you specify what Boolware Server should do. The following elements are available: ’SoftboolXML_request/SoftboolJSON_request’, ’open_session’, ’close_session’, ’database’, ’table’, ’index’ and ’query’.

Several Requests

A ’SoftboolXML_requests' element or an array of 'SoftboolJSON_requests’ could hold several Requests. See example at end of this chapter.

Request type

There are different types of requests, which is specified in the attribute ’type’ of the element *’SoftboolXML_request/SoftboolJSON_request’*.

Allowed types are:

Type	**Explanation**
query	Ordinary query to the Boolware Server specified with the QL syntax (see Softbool Query Language).
execute	Followed by other commands.
index	Gets a list of index terms from Boolware.
metadata	Gets a description of tables and columns from Boolware.

A request always starts with the root element *’SoftboolXML_request*’ with the requested type.

```
<SoftboolXML_request type="query">
</SoftboolXML_request>
```

If the type is *’query’* or *’index’*, the following information must be given:

```
<open_session name=""> </open_session>
<database name=""> </database>
<table name=""> </table>
<query> </query>
...   or   ...
<index=> </index>
```

If the request type is ‘*execute’* the following information must be given:
```
<open_session name=""> </open_session>
<execute></execute>
```

If the request type is *’metadata’*, the following information must be given:
```
<open_session name=""> </open_session>
<metadata database="" table="" field=""/>
```

When the query type is flow (type=*’flow’*), there is a more complete description with examples in Chapter 4 "Flow Queries".

Queries

Here is an example of a fairly simple query. A search for names that sound like "charter" within zip code area "11230".

```
<SoftboolXML_request type="query">
<database name="db1"/>
<table name="tab1"/>
<query>
FIND "name":sound(charter) and "zip":11230
</query>
</SoftboolXML_request>
```

Open a session

A connection to the Boolware Server is called a session.

In the element ’open_session’ you can specify the attributes: *’name’*, *’autotrunc’*, *’server’*, *’terminate’*, *’lang’*, *’queryhistory’* , *’setsearch’*, *’querylimvalue’* and *’indexexit’*.

name:
The value that is specified must be unique so it does not get mixed with another session in further calls to Boolware Server. If no value is specified the Boolware Server will create a unique session identifier that shall be used with further calls to Boolware Server for this session.
This attribute is only valid if the xmlclient is running as an extension to Apache 2.x or IIS. In all other clients the session name must be set using the function connect().

autotrunc:
This attribute should be set to "**1**", if automatic right truncation is required. This means that all terms starting with the given term will match the query. Value "**0**" means no right truncation. Default value is "**0**".

server:
This attribute shall contain the name of the server that host the Boolware Server. Either a computer name or an IP address. If no value given, the environment variable ’SERVER_NAME’ will be used. This attribute is only valid if the xmlclient is running as an extension to Apache 2.x or IIS. In all other clients it is the function connect() which tells the Boolware server to connect to.

terminate:
This attribute tells the Boolware Server to terminate the session or keep alive after this request. If "**1**" is specified, the session will be terminated in the Boolware Server and all old queries will be lost. If not supplied default is "**0**". This attribute is only valid if the xmlclient is running as an extension to Apache 2.x or IIS. In all other clients it is the function disconnect() that shuts down the connection to the client. The connection to Boolware is closed if terminate is set to "**1**".

lang:
This attribute tell the Boolware Server in what language error messages will be produced. The language code follows the internet standard: ‘en’ for English, ‘sv’ for Swedish etc.

There has to be a corresponding message file containing messages at the Boolware Server location. The installation package contains two different message files: one English and one Swedish. Default is ‘en’.

queryhistory:
Query history contains all queries between two FIND commands. In the query history it is possible to navigate with commands BACK and FORWARD. If the value "**1**" is specified the query history is activated. Default is "**0**"

setsearch:
Activate or deactivate set-search. Set-search will automatically save all queries and results during a search session with the names "S1", "S2"… "Sn". These saved results can be used in the Boolwares query language as ordinary query terms. The can be listed with the element ’execute’ see section "Set-search".

Set-search is a requires a lot of resources and should not be activated if not needed.
If "**1**" is specified, set-search is activated, default is deactivated.

querylimvalue:
Sets a global limit used by the sub-command *querylim*. If the number of hits for a term is higher than this value the term will not be part of the query (handles like a stop word). The global limit could temporarily be overridden by a values set in queryoptions or directly in the *querylim* sub-command. After the current query the global limit will be in effect again.

indexexit:
If set to other value than zero it indicates that custom indexing should be used when querying. Boolware contains three built in custom indexing exits: Split, Shrink and LinkWords. In the manual "Boolware Operations Guide" Chapter 14 "Custom indexing" you get a detailed description on how and when to use these exits.

To close down a session use the element ’close_session’ with the attribute *’name’*. Specify value of *’name’* to the session name obtained in the ‘session’ element in the response.

```
<open_session name="usr1" autotrunc="1" server="srv1" terminate="0" lang="sv"
queryhistory="1"> </open_session>
<close_session name="usr1"> </close_session>
```

Automatic right truncation is required, keep session, "usr1" alive in the Boolware Server, error messages in Swedish and activate the query history.

Specify database

The Boolware Index to query is given in the attribute ’name’ in the element *’database’*. Further one attribute is available, *’dsn’*.
```
<database name="db1" > </database>
```

Specify table

Specify the table for the query in the attribute *’name’* in the element *’table’*.
```
<table name="tab1"/>
```

Fetch index terms

The type 'index' in the element 'SoftboolXML_request/SoftboolJSON_request' in the request will fetch index terms.

The response from this request are the searchable terms that are in the specified column. The following attributes are valid in the element '*record*': '*field*', '*total*', '*allixtypeterms*', '*resultixtypeterms*', '*ixtypeterms*', '*zoom*', '*statistics*', '*numeric*' and '*totdocs*'.

The name of the column from which index terms have been fetched is stored in the attribute 'field'. The attribute 'field' could have additional information: presentation type and order. Two different presentation types are valid: *term* (default), when the terms should be presented in alphabetical order and *count* when the terms should be presented in frequency order (no. of occurrences). There are two directions: *asc* and *desc*. If no order is specified the terms will be presented in ascending order.

The total number of index terms fetched are stored in the attribute '*total*'. It is mostly equal to number of requested terms (max_terms) but could be less if not more are available.

If you have requested the total number of index terms for all index types, this is obtained in the attribute '*allixtypeterms*'.

If the total number of index terms has been requested for given index type, it is stored in the attribute '*resultixtypeterms*'. This number tells how many index terms there are totally for this request. If '*zoom*' (see below) has been specified in the request the total number of index terms within the current result will be stored else the total number of index terms for the specified index type will be stored.

If you - when using '*zoom*' in the request - also request the total number of index terms for the specified index type it will be stored in the attribute '*ixtypeterms*'. The attribute '*ixtypeterms*' will always hold the total number of terms for the given index type. If no '*zoom*', the value in '*resultixtypeterms*' and '*ixtypeterms*' will be the same.

If a 'zoom' request the attribute '*zoom*' will be set to "yes".

If a 'statistics' request the attribute '*statistics*' will contain the chosen statistics method (sum, max, min, mean/avg).

If the current column contains numeric data the attribute '*numeric*' will be set to "yes".

If a 'totdocs' request has been performed the attribute '*totdocs*' will hold the total number of records in the current table.

There will be one element 'record' for each fetched index term containing the following attributes: '*term*', '*count*', '*tothits*', '*termnumber*', '*abstermnumber*' and '*selected*'.

In the attribute '*term*' the current index term will always be stored.

The attribute '*count*' will hold the number of records the current term is contained in. If the attribute '*zoom*' is specified in the request only records contained in the current result are counted. This attribute is always filled in.

The attribute '*tothits*' holds the number of records the current term is contained in the complete index. If no 'zoom', '*count*' and '*tothits*' contain the same value. This attribute must be asked for in the request.

The attribute '*termnumber*' holds the sequence number of the current index term in the last result while '*abstermnumber*' always holds the absolute number in the total index for this index type. If no 'zoom' in the request, '*termnumber*' and '*abstermnumber*' will always be the same for the index term until an update or load is performed. If 'zoom' is requested, '*termnumber*' is the current sequence number for the current index term in the current result. '*abstermnumber*' always holds the same number each term regardless of the query.

The following attributes are valid in an index request: 'field', ’max_terms', 'start_position', 'type', 'zoom', 'zoomresult', 'tothits', 'continuation', 'allixtypeterms', 'resultixtypeterms', 'ixtypeterms', ’statistics’ , 'order', 'termnumber', 'freqtype', 'freqlimits', 'sepgroups', ‘keepzeroterms’, 'skipgeneratedterms', 'selected', 'totdocs', 'reportaction', 'reporttemplatename', 'levelformaxrecords', and 'maxrecords'.

Some attributes - 'field', 'type', 'start_position', max_terms', 'zoomresult', ’statistics’, 'order', 'freqtype', 'reporttemplatename', 'levelformaxrecords', 'reportaction' and 'maxrecords' - should contain a value while other attributes - 'zoom', 'tothits', 'allixtypeterms', 'resultixtypeterms', 'ixtypeterms', 'termnumber', 'sepgroups', ‘keepzeroterms’, 'continuation', 'skipgeneratedterms' and 'selected' - are activated by "yes".

The only attribute that has to be specified is ''field', which tells from which column the index terms should be fetched.

The attribute 'type' tells which index type (word, string phonetic etc.) should be fetched. The attribute '*type*' could also indicate another function (a complete description of the approved types could be found in Chapter 1 "Execute commands in Boolware" and section "**indexex**")

In the attribute 'start_position' you could specify where in the index you want to start fetching index terms. If you specify a word or a start of a word the system finds the closes term in the index to start from. You could also start from a sequence number by specifying that in the 'start_position'. To be able to decide if it is a sequence number or a word (a word could also be a number) you have to set the proper 'type'. If the attribute 'start_position' is omitted or set to empty string the fetch of index terms will start from the beginning or end depending on the value of 'order'.

In the attribute 'max_terms' you specify how many index terms you want. If omitted the default number of terms is set to 50. The maximum number of index terms to be fetch in one request is 1.000. If you want more use the 'continuation' attribute.

By specifying the attribute 'zoom' you will just get index terms that belong to records in the current result. All other attributes will be handled as when fetching index terms from the entire index.

The attribute 'zoomresult' is very much connected to the 'zoom' attribute. In this attribute you specify from which result you want to fetch index terms. Three different results are valid: the current result, a result from a saved result and the current result in named scratch result. If omitted the current result is used. The name of the saved result uses that result to get the proper index terms. To get index terms from a named scratch result, e.g. named '*address*', you should specify zoomresult="scratch(address)"

The attribute 'tothits' is most usable when 'zoom' is active. When you have got the number of occurrences for the current term in the current result saved in 'count' you could get the number of occurrences in the entire index for the current term in 'tothits'. If no 'zoom', the value in 'count' and 'tothits' is the same.

Using the attribute ‘allixtypeterms’ indicates that you want the total number of index terms for all index types.

If the attribute 'resultixtypeterms' is active the total number of terms for the current index type (specified in 'type') will be returned. If 'zoom', only terms within the current result will be counted.

The attribute 'ixtypeterms' means that you want the total number of terms in the entire index regardless of any result for the given index type. If no 'zoom', the value in 'resultixtypeterms' and 'ixtypeterms' is the same.

In the attribute ’statistics’ you could get statistics on the lowest level in a subzoom request. The following statistics are valid: sum (sum of all values), max (the highest value), min (the lowest value) and mean/avg (the average value).

In the attribute 'order' you determine the order (ascending or descending) the index terms should be fetched. The value in 'order' should be "asc" for ascending and "desc" for descending. If omitted the order will be ascending.

If the attribute 'termnumber" is active each term will contain sequence numbers. Two values: ’termnumber’ and ’abstermnumber’ will be fetched. The ’abstermnumber’ is always the same and gives the absolute order number of the term within the current index type. The ’termnumber’ is dependent on 'zoom'. If 'zoom' is activated the ’abstermnumber’ will be the order number of the current term within the last result.

In the attribute 'freqtype' you specify the type (word, string, phonetic etc.) you want to fetch when the index terms should be in concurrency order rather than alphabetic order ('type'="14").

In the attribute 'freqlimits' you should specify a condition for the terms to extract when sorting terms on occurrence (type="14"). The condition could be: < (less than), > (greater than) or = (equal to) a specified number of records the terms should appear in. For example freqlimits=&gt;100000 means that only terms that appears in more than 100.000 records will be fetched.

If the attribute 'sepgroups' is activated it means that the index terms within the current result should be separated. Terms within the result should be fetched first and terms not within the result should be fetched after. Compare this to the attribute ‘keepzeroterms’ where terms within the result and the other terms are listed together in alphabetic order. This attribute is only valid if the attribute 'zoom' is activated.

If the attribute ‘keepzeroterms’ is activated it means that all index terns will be fetched. The terms within the result will be listed with the appropriate hitcount, while the terms not in the result will be listed with hitcount set to zero. All terms will be fetched in alphabetic order. Compare this to the attribute ‘sepgroups’ where terms within the result and the other terms are listed in separate groups. This attribute is only valid when the attribute ‘zoom’ is active.

Note:
The attributes ‘sepgroups’ and ‘keepzeroterms’ could never be active at the same time. If so an error message will occur.

The attribute 'continuation' means that you continue from the latest fetched index term.

Read more about the values for the attribute 'type' in the chapter "Execute commands to Boolware" and the command "**indexex**".

Get maximum 10 index terms from col1 start from term AAA. Only strings should be fetched and should be within the current search result.

```
<index field="Ort" start_position="B" max_terms="10" zoom="yes" type="2"
tothits="yes">
</index>
```

Response:
```
<records field="City" total="10" zoom="yes">
<record term="BALTIMORE" count="23" tothits="23192"/>
<record term="BARCELONA" count="12" tothits="14367"/>
<record term="BELGRADE" count="9" tothits="11333"/>
<record term="BERLIN" count="17" tothits="20101"/>
<record term="BERN" count="8" tothits="12876"/>
<record term="BIRMINGHAM" count="44" tothits="65323"/>
<record term="BONN" count="11" tothits="14593"/>
<record term="BOSTON" count="21" tothits="23876"/>
<record term="BRUSSELS" count="12" tothits="13586"/>
```

```
<record term="BUDAPEST" count="9" tothits="11984"/>
</records>
```

Get the maximum number of employees by Cities on the current result.

```
<index field="City(Employees)" type="2" max_terms="12" statistics="max"/>
</index>
```

Response:

```
<records field="ort([Antal anställda])" total="108" statistics="max">
<record term="City" count="-1"/>
<record term="BALTIMORE" count="64363"/>
<record term="Employees" count="-1"/>
<record term="101760" count="1"/>
<record term="City" count="-1"/>
<record term="BOSTON" count="51157"/>
<record term="Employees" count="-1"/>
<record term="87994" count="1"/>
<record term="City" count="-1"/>
<record term="BERLIN" count="48966"/>
<record term="Employees" count="-1"/>
<record term="67855" count="1"/>
</records>
```

Attributes for "Extended statistics between several Boolware indexes".

If the attribute 'skipgeneratedterms' is activated, all strings generated by a plugin function will be skipped.

The attribute 'selected' tells that each term should contain the attribute 'selected'. If the value on a term is "yes" the term was selected when the last 'reportclose' was performed else it should contain "no". If 'selected' is omitted or set to "no" no 'select' attribute will appear on the listed terms.

The attribute 'reportaction' should contain one of three values: 'open', 'close' and 'save'.
The value 'open' opens one index at a time in the named reporttemplate ("report") and reads corresponding files that tells which terms were marked at last 'reportclose'. Requested number of terms are fetched for all indexes that belongs to the current reporttemplate ("report"). If 'selected' is activated all terms for those indexes will get the attribute 'selected'. If the current term was marked at the last 'close' 'selcted' will be set to "yes" else it will be set to "no". There should be one call for each involved index. The value 'close' closes one index at a time for the current reporttemplate ("report") and saves marked terms in memory. There should be one call for each involved index. The value 'save' saves the saved marked terms for all indexes in a file for later use (the next reportaction="'open" for this report).

The attribute 'reporttemplatename' tells which reporttemplate ("report") to be used.

In the attribute 'levelformaxrecords' you should specify the level that should contain a maximum of 'maxrecords' records. Normally that is the lowest level - the level on which the requested 'statistics' is performed - that should only contain e.g. the 5 "best" values.

The attribute 'maxrecords' tells how many records the 'levelformaxrecords' could contain at a maximum.

Below you will find an example of the records that are created when using report templates. The general idea is that the records should be used as input to programs that could present the statistics graphically.

Example 1:

The table Companies contains geographical and economic information on companies. A report template, *CityNameTurnover*, has been created and contains the following information: First dimension: City, Second dimension CompanyName and Column for aggregation value: Turnover. You want statistics on turnover per company within a city. In the below example you have chosen to aggregate against the current result (zoom=yes) and the statistic method is the sum (sum). In order to get only the three "best" companies from the cities that have the highest turnover you specify maxrecords=3. In this example you are just interested in companies in the three cities: Chicago, Washington and Seattle.

The XML request look like this:
```
<?xml version="1.0" encoding="ANSI"?>
<SoftboolXML_requests>
  <SoftboolXML_request type="query">
  <open_session name=""/>
  <database name="Companies"/>
  <table name="Companies"/>
  <query>FIND City:Chicago OR Whasington OR Seattle</query>
  <response/>
  </SoftboolXML_request>

  <SoftboolXML_request type="index">
  <database name="Companies"/>
  <table name="Companies"/>
  <index
    field="City(CompanyName(Turnover))" zoom="yes" type="2"
     max_terms="36" statistics="sum"
     skipgeneratedterms="yes" reporttemplatename="CityNameTurnover"
     levelformaxrecords="3" maxrecords="3">
  </index>
  </SoftboolXML_request>
</SoftboolXML_requests>
```

Gives the following response:
```
<records field="City(CompanyName(Turnover))" total="36" zoom="yes"
statistics="sum">
<record term="City" count="-1" level="0"/>
<record term="CHICAGO" count="1" level="0"/>
<record term="4287216143749000" count="1" level="0"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="THE BOEING COMPANY" count="1" level="1"/>
<record term="52457000000" count="1" level="1"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="CLEAR FUSION INC" count="1" level="1"/>
<record term="42000000000" count="1" level="1"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="BANK ONE CORPORATION" count="1" level="1"/>
<record term="20724000000" count="1" level="1"/>
<record term="City" count="-1" level="0"/>
<record term="WASHINGTON" count="1" level="0"/>
<record term="1247678562124000" count="1" level="0"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="UNITED STATES POSTAL SERVICE" count="1" level="1"/>
<record term="68529000000" count="1" level="1"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="MARRIOT INTERNATIONAL INC" count="1" level="1"/>
<record term="10099000000" count="1" level="1"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="DANAHER CORPORATION" count="1" level="1"/>
<record term="6889301000" count="1" level="1"/>
<record term="City" count="-1" level="0"/>
<record term="SEATTLE" count="1" level="0"/>
<record term="698742198197000" count="1" level="0"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="WASHINGTON MUTUAL INC." count="1" level="1"/>
<record term="15962000000" count="1" level="1"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="NORDSTROM INC" count="1" level="1"/>
```

```
<record term="7131388000" count="1" level="1"/>
<record term="CompanyName" count="-1" level="1"/>
<record term="AMAZON.COM INC" count="1" level="1"/>
<record term="6921124000" count="1" level="1"/>
</records>
```

The response should be interpreted in the following way:

'term'	is the current "term". If 'count' is -1 the current term is the Column name. In the aggregated column - in this case - turnover the aggregated value - in this case - the sum is the value in 'term'.
'count'	if 'count' is -1 it indicates that 'term' contains the Column name, else the number of occurrences of the current 'term'.
'level'	is the current 'level'. The highest level (the first field) is 0 (zero).

Each complete record on each level is built up of a number of records.

This is the records for level 0:
In the first record the contents of 'term' should be regarded as a Column name as the 'count'=-1. In this case the Column name is City.

The second record contains the current 'term' and its count for this level. In this case the 'term' is Chicago and there is only one city Chicago.

The third record contains the aggregated value - in this case the sum - of all turnover in all companies in Chicago.

The following are the records for level 1 under the current level 0 (City is Chicago):
In the first record the contents of 'term' should be regarded as a Column name as the 'count'=-1. In this case the Column name is CompanyName.

The second record contains the current term and its count for this level. In this case the term is a company named "THE BOEING COMPANY" and there is only one ('count'=1) company with that name in Chicago.
The third record contains the aggregated value - in this case the sum - of Turnover for all companies named "THE BOEING COMPANY". In this case there is only one company with that name and the sum is the same as the Turnover for this company.

As you have specified that you only want the three "best" companies (the ones with the highest turnover for each city) only: "THE BOEING COMPANY", "CLEAR FUSION INC" and "BANK ONE CORPORATION" will be presented for Chicago.

After this the three "best" companies in Washington will be presented and last the three "best" companies in Seattle will be presented.

When presenting statistics graphically in the application it could be convenient to do the searching via the indexes instead of via an ordinary search form.

In a search form you usually show the searchable fields and the end user fills in all search arguments in the corresponding search fields. When searching via indexes you present the content of those indexes that appears in the search form. The end user then marks the terms (search arguments) that should be part of the query in the different indexes. Boolware will the use the indexes and "zoom" a result in each of them based on the current search result.

Assume that you have four indexes that you want to use for this purpose: State, City, CmpName and Turnover. At start the first N (e.g. 20) terms in all indexes are shown. When you mark for instance Washington in the index for State only cities, company names and turnover from records in the state Washington will be presented. The presented terms could also contain number of occurrences: number of occurrences in the current result is always returned and the number of occurrences in the total table will be returned if *tothits* is set to 'yes'.

You could continue to mark more terms in the index State to increase the number of terms in all indexes (equivalent to the operator OR) or you could mark terms in another index to reduce the number of terms in all index (equivalent to the operator AND). Instead of only present the terms that are part of the current result you could specify an attribute (*sepgroups*) so that all terms within the current result will be presented first and all other terms after these "found" terms. Another way to see all terms is to use the attribute 'keepzeroterms' where all the terms are listed in alphabetic order; the terms within the result has a proper hitcount while the hitcount for the terms not within the result is set to zero.

Below is an example of the XML request:
Example 2:
In this example Alaska (the 2nd term in this index) is marked in the index State and we want all terms that is part of the current result should be presented before all other terms in all indexes. You could use the sub command when searching for Alaska. This command picks up the proper term - by using the absolute termnumber - from the specified index and performs an ordinary search.

All indexes are "zoomed" against the result and the terms that are part of the result will be presented first in each index followed by the terms that are not part of the current result. This is controlled by the attribute *sepgroup*=yes. The start term (*start_position*) is specified by term number (*type*=23). Only the first 10 terms are fetched (*max_terms*). You want both the relative and the absolute term number (*termnumber*=yes), the terms that are marked (*selected*=yes), the total number of terms (*resultixtypeterms*=yes), the total number of records (*totdocs*=yes) and you want to skip all strings generated by a plugin function (*skipgeneratedterms*=yes).

```
<?xml version="1.0" encoding="ANSI"?>
<SoftboolXML_requests>
  <SoftboolXML_request type="query">
  <open_session name=""/>
  <database name="Companies"/>
  <table name="Companies"/>
  <query>FIND "State":searchviaindexnogen(2)</query>
  <response />
  </SoftboolXML_request>

  <SoftboolXML_request type="index">
  <database name="Companies"/>
  <table name="Companies"/>
  <index field="State" start_position="1" max_terms="10" zoom="yes" type="23"
     sepgroups="yes" termnumber="yes" selected="yes" resultixtypeterms="yes"
     totdocs="yes" skipgeneratedterms="yes">
  </index>
  </SoftboolXML_request>

  <SoftboolXML_request type="index">
  <database name="Companies"/>
  <table name="Companies"/>
  <index field="City" start_position="1" max_terms="10" zoom="yes" type="23"
     sepgroups="yes" termnumber="yes" selected="yes" resultixtypeterms="yes"
     totdocs="yes" skipgeneratedterms="yes">
  </index>
  </SoftboolXML_request>

  <SoftboolXML_request type="index">
  <database name="Companies"/>
  <table name="Companies"/>
  <index field="CmpName" start_position="1" max_terms="10" zoom="yes"
     type="23" sepgroups="yes" termnumber="yes" selected="yes"
     resultixtypeterms="yes" totdocs="yes" skipgeneratedterms="yes">
  </index>
  </SoftboolXML_request>

  <SoftboolXML_request type="index">
  <database name="Companies"/>
  <table name="Companies"/>
  <index field="Turnover" start_position="1" max_terms="10" zoom="yes"
     type="23" sepgroups="yes" termnumber="yes" selected="yes"
     resultixtypeterms="yes" totdocs="yes" skipgeneratedterms="yes">
  </index>
  </SoftboolXML_request>
</SoftboolXML_requests>
```

Gives the following response:

Index State:

```
<records field="State" total="10" resultixtypeterms="54" zoom="yes"
totdocs="100321260">
<record term="ALASKA" termnumber="1" abstermnumber="91" count="108686"
selected="yes"/>
<record term="ALABAMA" termnumber="2" abstermnumber="82" count="0"
selected="no"/>
<record term="ARIZONA" termnumber="3" abstermnumber="369" count="0"
selected="no"/>
<record term="ARKANSAS" termnumber="4" abstermnumber="370" count="0"
selected="no"/>
<record term="CALIFORNIA" termnumber="5" abstermnumber="777" count="0"
selected="no"/>
<record term="COLORADO" termnumber="6" abstermnumber="1132" count="0"
selected="no"/>
<record term="CONNECTICUT" termnumber="7" abstermnumber="1153" count="0"
selected="no"/>
<record term="DELAWARE" termnumber="8" abstermnumber="1247" count="0"
selected="no"/>
<record term="DISTRICT OF COLUMBIA" termnumber="9" abstermnumber="1272"
count="0" selected="no"/>
<record term="FLORIDA" termnumber="10" abstermnumber="1543" count="0"
selected="no"/>
</records>
```

The response should be interpreted in the following way:
The first row tells which index (*field*) the terms belong to, *total* is the total number of fetched terms, *resultixtypeterms* tells how many terms there are in the current result, *zoom* means that terms will be presented according to the current result and finally *totdocs* gives the total number of records in the entire table.

The first term that will be presented is ALASKA, where *termnumber* is the relative order number within the result, *abstermnumber* is the absolute order number within this index, *count* tells how many records there are that contain Alaska and finally *selected* is 1 means that the term is marked.

None of the following terms are part of the result which is indicated by *count* equals 0. Moreover, none of these terms have been marked; *selected* is set to 0. The relative order number, *termnumber*, and the absolute order number, *abstermnumber*, are set on all terms.

Index City:

```
<records field="physical_town_city_name" total="10" resultixtypeterms="241"
zoom="yes" totdocs="100321260">
<record term="AKIACHAK" termnumber="1" abstermnumber="3016" count="22"
selected="no"/>
<record term="AKIAK" termnumber="2" abstermnumber="3017" count="13"
selected="no"/>
<record term="AKUTAN" termnumber="3" abstermnumber="3122" count="14"
selected="no"/>
<record term="ALAKANUK" termnumber="4" abstermnumber="3313" count="42"
selected="no"/>
<record term="ALEKNAGIK" termnumber="5" abstermnumber="4168" count="31"
selected="no"/>
<record term="ALLAKAKET" termnumber="6" abstermnumber="4647" count="18"
selected="no"/>
```

```
<record term="AMBLER" termnumber="7" abstermnumber="6192" count="18"
selected="no"/>
<record term="ANAKTUVUK PASS" termnumber="8" abstermnumber="6693" count="23"
selected="no"/>
<record term="ANCHOR POINT" termnumber="9" abstermnumber="6792" count="392"
selected="no"/>
<record term="ANCHORAGE" termnumber="10" abstermnumber="6793" count="39119"
selected="no"/></records>
```

The response should be interpreted in the following way:
The first row should be interpreted in the same way as the first row for index State above.

All terms that are presented are part of the result since *count* is greater than 0. The relative order number, *termnumber*, and the absolute order number, *abstermnumber*, are set on all terms. No term has been marked since the *selected* is '*no*' on all terms.

The other indexes int the request: CmpName and Turnover will be presented in a similar way.

In the manual "**Operations Guide**" Chapter 11 "Interactive Query" you could read about frequency index in section "View Frequency Index" and about Query term statistics in section "Statistics on Query Terms". There is also a section "Extended statistics between several Boolware indexes" that describes the use of report templates.

Important: After the function SubZoom was implemented all column names containing parentheses must be enclosed within quotation marks. See also Operations Guide chapter 11 "Interactive Query" section "Hits projected hierarchically over one or multiple other values (SubZoom)".

Search in a database/Boolware Index

Select type of query by specifying *'query'* in the attribute 'type' in the element *SoftboolXML_request/SoftboolJSON_request'*.

Type *'query'* means to query against the Boolware Index.

The syntax for the type *'query'*, Softbools Query Language (QL), is described in Chapter 1 "*Softbool Query Language*".

The following attributes could be used in the <*query*> element: '*flow*', '*trace*', '*qtime*' and '*resultbitmap*'.

In the attribute '*flow*' you specify the name of a flow that should be executed. Normally the *query* request contains: commands, search arguments and operators, but when this attribute is specified the given parameters will be run through the named Flow. See Chapter 4 Flow Queries below.

In the <query> element, the attribute 'trace' can be used together with the attribute 'flow' if the <scoring> element has been used in the flow. If the attribute 'trace' is enabled (1 = on and 0 = off, default setting), the <trace> element in the <*SoftboolXML_response*> will describe how the scoring was performed by the <*scoring*> element in the flow.

If the attribute '*time*' has been activated the execution time for the current *query* request will be returned.

When using the type *'query'* you can omit the actual query to avoid a re-search if only the next part of a result list should be displayed.

```
<SoftboolXML_request type="query">
  <database name="database"/>
  <table name="table"/>
  <response>
    <records from="101" count="100">
      <field name="*"/>
    </records>
  </response>
</SoftboolXML_request>
```

The response

The response part of the request describe what kind of information that will be received from the Boolware Server: number of records, what record to start from, what order of the records, query history, statistics etc.

What information that will be obtained is dictated by elements and their attributes.
The following attributes are available: *'response'*, *'dtdref'*, *'sort'*, *'records'*, *'field'* and '*statistics'*.

How to display the result

The element, *'response'*, have the attributes: *'type'*, *'href'*, '*raw'*, *'queryhistory,* '*time''* and '*outputformat'*.

The '*raw'* attribute is a bit special, in that it changes the Boolware response format. This means that fetched tuples, and index terms, will be returned in tab separated format instead of XML. The intention is to return a format which is easier and faster to parse, than standard XML would be.

If both *'type'* and *'href'* have values, there will be a XML "style sheet" element in the response part with following look:

```
<?xml-stylesheet type="type" href="href"?>
```

immediate after the leading XML header:

`<?xml version="1.0" encoding="iso-8859-1"?>`

By using the attribute *'outputformat'* you can change the format of the complete response. Valid values are "xml" or "json". If the request is in XML and you set `outputformat="json",` the response will be delivered in JSON format. If the request is in JSON and you set `"@outputformat":"xml"` , the response will be delivered in XML format.

If *'queryhistory'* is set to "**1**", the query history will be fetched if it is activated. Default is "**0**".

If *'time'* is set to "**1**", a time element will be produced in the response indicating the thread time an overall time for the request. Default is "**0**".

Connect XML-response to an extern DTD

In the element, *'dtdref'*, an extern file can be specified that contains a document definition, DTD, and it will be included in the response.

The element, *'dtdref'*, have four attributes: *'sysid'*, *'type'*, *'name'* and *'pubid'*.

The system identification for the DTD is given in the attribute *'sysid'* and is specified as an URI.

An URI is either a complete filename or an URL. If this attribute is empty or is missing, no DTD-reference element is created in the XML-response.

The attribute *'type'* can have one of two values: PUBLIC or SYSTEM; default is SYSTEM. If PUBLIC is given, a descriptive name can be specified in the 'pubid' for this DTD.

The attribute *'name'* should contain the name of the root element, if no value is given the root element is assumed to be *'SoftboolXML_response'*.

The last attribute, *'pubid'*, is an explaining text for this DTD stored if the attribute *'type'* has the value PUBLIC.

If the element, *'dtdref'*, is present and the attribute *'sysid'* has a value, the Boolware Server will create a 'DOCTYPE' element in the response.

Examples:

```
<dtdref sysid="c:\my.dtd" type="SYSTEM"/>
```

will generate the following response from the Boolware Server:

```
<!DOCTYPE SoftboolXML_response SYSTEM "c:\my.dtd">
<dtdref sysid="c:\my.dtd" type="PUBLIC" pubid="DTD for Company database"/>
```

will generate the following response from the Boolware Server:

```
<!DOCTYPE SoftboolXML_response SYSTEM "c:\my.dtd" pubid="DTD for Company
database">
```

Sort the result

If a special sort order is requested use the element 'sort'. The element has one attribute *'expression'*, were to put the columns to sort upon and what order ascending/descending. The columns will be sorted in specified order. If the column name contains special characters it has to be quoted. The order within each column is given by the suffix "**asc**" for ascending and "**desc**" for descending. Default is ascending if omitted. By the parameter emptydata=first/last you could control where to "sort" records that do not contain any data in the sort column; first or last. Separate columns with a comma sign (,). A special attribute 'coordinatedistance' could be used as a sort element. It means that the retrieved records will be sorted on a distance from the given position.

A maximum of 20 sort elements (column names) could be given in a sort request.

Syntax for 'expression':
<sortelement1>[, < sortelement2> ... < sortelement20>]

```
< sortelement>
    <column name>|<coordinatedistance>  <order>  <emptydata>
       < column name>       the specified column must be part of the current table
       <coordinatedistance="<latitude value>;<longitude value>;<column1>;<column2>;
           < latitude value>   latitude value (proper format) where to start calculation
           < longitude value> longitude value (proper format) where to start calculation
           <column1>          name of the column containing the latitude values
           <column2>          name of the column containing the longitude values
       <order>               'asc' (default) ascending or 'desc' descending
       <emptydata>           'last' (default) tells that all records that has no value will be put last
                             for this column; 'first' tells that they will come first
```

If the first Column to sort on is indexed as numeric or string you could take advantage of the very fast Boolware incremental sort. If you have a large number of records in your result and just want to present the one hundred first after a sort, you could specify that number in the sort command and Boolware will interrupt the sort after 100 records. See description and more examples in Chapter 2 "API description" section BCSort() above.

Example:

```
<sort expression="column1 asc emptydata=first, column2 desc"/>
```

The result will be viewed sorted ascending on column1 and descending on column2. No data in column1 will "sort" these records first.

Incremental sort:

```
<sort expression="column1 asc:100"/>
```

The current result will be presented sorted on the contents of column1 ascending, but only the 100 first records will be sorted. To take advantage of this fast sort the column must be indexed as string or numeric.

Distance sorting:

```
<sort expression="coordinatedistance=&quot;&quot;N 59 20 43.42&quot;; &quot;E 17 57
54.65&quot;; Latitude; Longitude;&quot;"/>
```

The retrieved records will be sorted on the distance from the given position (N 59 20 43.42; E 17 57 54.65); the closest will be sorted first. As the format is WGS84, the values have to be quoted. The name of the columns don’t contain any special characters and thus need not be quoted. Note that the order must be latitude/longitude both when giving the start position and when specifying the column names. Each entity must be ended by a semicolon (;); even the last one.

```
<sort expression="coordinatedistance=&quot;6582497; 1622891; coordinates(long, lat); ;
&quot;"/>
```

In this case both coordinates are contained in the same column (coordinates) and you have to specify in what order they appear (long, lat). If no order is specified the order will be: lat, long. Note that an "extra" semicolon must be specified for the "missing" column name.

Which records should be fetched and in what order

The element ’*records*’, contains the following attributes: ’*from*’, ’*count*’, ’*rank*’, ’*maxchars*’, ’*tablename*’, '*sortexpression*' and '*scorescale*'.

The attribute *’from’* give the relative start of the records to be fetched and the attribute ‘count’ specify number of records to fetch. To improve performance a restriction of maximum records to fetch in one call is set to 500 (see also Chapter 2 API description section BCSetFetchSize()).

If any particular order is requested – other than the data source -, a value can be set in the attribute *’rank’*. The order is based upon how many times the query terms occur in the record. If the value is set to "**1**" or "**occurrency**", the order is calculated by the occurrence of the query terms. The value "**2**" or "**frequency**" order the records by frequency; frequency in this context is the number of occurrences divided by number of terms in the record. The order can be affected by giving the query terms different weights. When calculated, the weight is multiplied with the occurrence of the term. This will affect the order for occurrence and frequency. The attributes for this orders are: "**6**" or "**weightedoccurrency**" for weighted occurrence and "**7**" or "**weightedfrequency**" for weighted frequency. If a fuzzy search have been performed the records can be fetched in fuzzy-order i.e. records with lowest string distance will be sorted first. The attribute for this order is "10" or "fuzzy". See example in the **Softbool Query Language in Operations Guide**. If "**0**" or no value is given, the records will appear in the order they will be fetched from the data source.

In the attribute *’maxchars’* the maximum number of characters that will be fetched from each column. If "**0**" is specified the entire column value will be fetched. Default is "**0**".

When more than one table have been involved in the search - global search or relate - it could be convenient to be able to present the result from several tables in one request. By using the attribute ‘*tablename’* the result from the tables involved in the search could be fetched. Moreover a sort attribute ‘sortexpression’ could be specified for each table.

In the attribute '*scorescale*' you could specify the wanted number of decimals for the score value. Valid values are: 0 - 7. If no value or a non-valid value is specified the number of decimals will be 3.

Example 1:

```
<records from="12" count="40" rank="occurrency" maxchars="20"/>
```

In this example fetch 40 records and start from record number 12 in the current result. Order by occurrence and just the 20 first characters from each column should be fetched.

Example 2:

```
<records from="12" count="40" maxchars="20" tablename="table3"
sortexpression="column4 desc"/>
   <field name="*"/>
</records>

<records from="12" count="40" rank="occurrency" maxchars="20"
tablename="table5" sortexpression=""/>
   <field name="*"/>
</records>
```

In this example fetch 40 records and start from record number 12 in the current result. The records should be sorted in descending order depending on the content in column4 when the result is fetched from table3. The records should be ordered by occurrence when fetched from table5. In both cases only 20 characters for each field should be fetched.

Select columns to fetch

The element *’field’* have three attributes: *’name’*, *’formula’* and ‘*dataenclosedelement’*.

In the attribute *’name’* the name of the requested column is specified and is mandatory.

The attribute *’formula’* is used only to specify a calculating formula for a nonexistent column specified by ‘*name’*. If ’*formula*’ is omitted or empty, field will be treated as an ordinary *’field’* element.

The attribute ‘*dataenclosedelement*’ is used when you want to enclose field data within a specified element name (see example under section Lookup below).

The element ’*field*’ can be repeated within the *’records’* element to obtain data from more than one column at a time.

Example:

```
<records from="12" count="40" rank="frequency" maxchars=20>
   <field name="column1"> </field>
   <field name="column6"> </field>
   <field name="column2"> </field>
   <field name="Sum" formula="column1 + column2"/>
</records>
```

In this case 40 records are requested with start from the relative record number 12 from the current result. Order the records by frequency of the query terms. Fetch the first 20 characters from the columns: column1, column6 and column2. One column with a calculated value of the sum of column1 and column2 as column "Sum".

Calculate and fetch statistics

The element *'statistics'* have four attributes: *'column'*, *'groups'*, '*onall*' and '*getallvalues*'. The attribute *'column'* tells what column to perform the statistic calculation upon and the attribute *'groups'* tells in how many groups the result should be divided.

The attribute *'column'* must contain an existing column in the database and should be numeric. *'groups'* gives the upper and lower group values 3 – 100.

If the attribute '*onall*' is set to 1 the calculation will be performed on all records in the table rather that on the current result.

If the attribute '*getallavlues*' is set to 1 you will get all limit values for the current group.

Example:
Get statistics for the column "*Solidity*" and all the limit values for the specified group (5). The statistics should be calculated on the current result (companies in Stockholm). Get statistics for the column "*Liquidity*" and all the limit values for the specified group (10). In this case the calculation should be performed on all records in the table.

Request:
```
<?xml version="1.0" encoding="ANSI"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <open_session name="" queryhistory="1"/>
  <database name="Companies"/>
  <table name=" Companies "/>
  <query>FIND City:Stockholm</query>
  <response queryhistory="1" type="" href="">
    <statistics column="Solidity" groups="5" getallvalues="1"/>
    <statistics column="Liquidity" groups="10" onall="1" getallvalues="1"/>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

Response:
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<SoftboolXML_responses>
<SoftboolXML_response type="query" error_code="0" databasename="Företag"
flowexit="" >
<session>MySession</session>
<records total="64428" tablename="Company">
</records>
<queryhistory total="1">
<queryitem result="64428" intermediate="64428">FIND City:Stockholm</queryitem>
</queryhistory>
 <statistics column="Solidity" tablename="Company" groups="5" onall="0">
  <statistic name="count" value="38283"/>
  <statistic name="modecount" value="3095"/>
  <statistic name="mode" value="100.000000"/>
  <statistic name="sum" value="-779180537.840000"/>
  <statistic name="avg" value="-20353.173415"/>
  <statistic name="min" value="-425099700.000000"/>
  <statistic name="max" value="1326.410000"/>
  <statistic name="std" value="2396790.246590"/>
  <statistic name="var" value="5744603486147.931600"/>
  <statistic name="median" value="38.870000"/>
  <statistic name="upper" value="82.435000"/>
  <statistic name="lower" value="9.500000"/>
  <statistic name="groups" values="9.500000, 27.825000, 51.435000, 82.435000/>
 </statistics>
<statistics column="Liquidity" tablename="Företag" groups="10"
            onall="1">
  <statistic name="count" value="217869"/>
  <statistic name="modecount" value="63"/>
  <statistic name="mode" value="100.000000"/>
  <statistic name="sum" value="610811992.529998"/>
  <statistic name="avg" value="2803.574591"/>
  <statistic name="min" value="-35215300.000000"/>
  <statistic name="max" value="161918100.000000"/>
  <statistic name="std" value="424165.974683"/>
  <statistic name="var" value="179916774079.192990"/>
  <statistic name="median" value="123.810000"/>
  <statistic name="upper" value="538.960000"/>
  <statistic name="lower" value="33.890000"/>
  <statistic name="groups" values="33.890000, 60.820000, 83.630000,
             103.610000, 123.810000, 149.970000, 191.260000, 272.360000,
             538.960000/>
  </statistics>
</SoftboolXML_response>
</SoftboolXML_responses>
```

Fetch Similarity vectors

The element *’simvectors’* have three attributes: *’from’*, *’count’* and *’content’*.

The attribute *’content’* tells the type of term to fetch from a retrieved record, the term as text or the term as a numeric code.

The value of the attribute can be either *’numeric’* or *’text’*. Default value is *’numeric’*.

The attribute *’from’* gives the starting point of the retrieved records and the attribute *’count’* is the number of records to fetch.

Example:

Fetch similarity vectors for the first 10 records in numeric codes.
```
<simvectors from="1" count="10" content="numeric"/>
```

Execute

The element ‘*execute’* is used to perform commands described in Chapter 1 "Application development" section "Execute commands in Boolware".

The syntax of the ‘*execute’* element is:
```
<execute> command and its parameters </execute>
```

The syntax for each command is described in Chapter 1.

A response part, ‘*execute_response*’ is connected to the ‘*execute’* element in which the response from the current command is saved (the Set Search commands are treated in a different way which is described below).

```
<execute_response>
response from the current command
</execute_response>
```

The saved result could be very different depending on the specified command. Some commands just saves a 0 or 1 while other commands save a lot of data. In Chapter 1 the different commands and their output are described in detail. The response is not formatted at all.

Example:
The test database Northwind contains a lot of tables with relations. The database is described in "Operations Guide" Chapter 11 "Interactive Query" under section "Related search (Join)".

The example below shows how to use an XML script to make an efficient Join.

If you want to find out which customers (Customs) in the US that have ordered the product: "Uncle Bob's Organic Dried Pears" or the product "Alice Mutton". The following query could be specified in Boolware:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
<SoftboolXML_request type="execute">
<open_session name="User1" queryhistory="1"/>
<database name="Northwind"/>
<table name=" "/>
<execute>
  relate(Customers)
  FIND  Customers.Country:usa
  ANDF Products.ProductName:(uncle bob organic dried pears)  OR  (alice
mutton)
</execute>

<response queryhistory="1" type="" href="">
   <sort expression=""/>
   <records from="1" to="25" maxchars="0" tablename="Products"
   sortexpression="Productname desc">
      <field name="ProductID"/>
      <field name="ProductName"/>
   </records>
   <records from="1" count="25" maxchars="0" tablename="Order Details"
sortexpression="">
      <field name="*"/>
   </records>

</response>
</SoftboolXML_request>
</SoftboolXML_requests>
```

The above XML script will retrieve the 6 requested records. Note that the result is fetched from two of the involved tables: Products and Order Details. Order Details is not specified in the query but to relate the search result from table Products to table Customers you have to go via table Order Details.

Set Search

The element '*execute'* is used for set-search, review saved queries, save query and delete saved query. The various commands are described in Chapter 1 under the heading "Execute commands in Boolware".

Fetch metadata

The element *'metadata'* has three attributes: *'database'*, *'table'* and *'field'*.

The attribute *'database'* hold the requested database to fetch meta data from. The name of the requested database or '*' for all databases.

The attribute *'table'* hold the requested table to fetch meta data from. The name of the table or '*' for all tables within requested database.

The attribute *'field'* hold the requested field to fetch meta data from. The name of the requested field or '*' for all fields within the requested table and database.

The attributes *'table'* + *'field'* can be omitted and the attribute *'field'* can be omitted.

Example: Fetch all of databases available and their attributes:
```
<metadata database ="*"/>
```

Example: Fetch all tables from the database 'Company' and their attributes.
```
<metadata database ="Company" table="*"/>
```

Example: Fetch all fields available and their attributes from database 'Company' and table 'Employee'.
```
<metadata database ="Company" table="Employee" field="*"/>
```

Response from Boolware Server

The specified response in the response part of the request will be sent as a XML coded document.

It always starts with a XML Header: **<?xml version="1.0" encoding="iso-8859-1"?>** and is followed by the following start element *'SoftboolXML_responses*'.

A Response will always have the same encoding (*UTF-8* or *iso-8859-1*) as the Request.

A response is formatted as XML/JSON, except for the special case of "´raw" responses. Below all the elements and their attributes are described:

XML element	Attribute	Description
SoftboolXML_responses		Start of complete response
SoftboolXML_response	type, error_code	Start of response
SoftbooJSON_response	type, error_code	Start of response
session		The current session records
records	field, total	Response when type 'index'
record	term, count	Record when type 'index'
records	total, from, to, rank	Response when type 'query'
record	score	Record when type 'query'
records	total	Response when type 'execute'
record	name, ctime, database, result	Record when type 'execute'
field	name	Current Column in record when type 'query'
queryhistory	total	Number of rows in the Query History
queryitem	result, intermediate	Total result and result for the current query
simvectors	from, to, content	Similarity vectors
statistics	column, groups, onall, groupvalues	Statistics for the specified Column on the current result
statistic	name, value	A named statistic method

		and its value
execute_response		Data from an 'execute' command
save, review, delete	type, total, name, database, ctime, result	Elements in the response when type is 'execute'
time	thread, total	Time specification for the request
metadata	See below	Meta data about databases tables and fields

The contents in the response part varies depending on:
1. Whether the request was successful
2. The type of the request ("index" or "query")
3. What is specified in the 'response' element

Was the request successful or not

The element *'SoftboolXML_response/SoftboolJSON_response*' has the following attributes: *'type'* and *'error_code'*.
The attribute *'type'* holds the request for this response. The *'type'* is specified in the element *'SoftboolXML_request/SoftboolJSON_request*' and could contain one of the following values: "*index*" or "*query*".
In the attribute *'error_code*' the outcome of the request is denoted; "**0**" means success, while a negative value means failure. If an error has occurred an element *'error'* is generated, where the current error message is stored.

In some cases the request could generate a warning, which will be notified in the element *'warning'*. This element has an attribute *'warning_code'*, where the warning code (a positive value) is followed by the warning message.

A very important element in the response from Boolware Server is *'session'*. This element holds the unique session identification. The session identification could explicitly be specified in the request or it could be generated by Boolware. Regardless how the unique session identification was created this session identification should be used to communicate with Boolware.

Example:
```
<SoftboolXML_response type="query" error_code="-112">
   <session>User1</session>
   <error>End Interval Term missing</error>
   [<warning warning_code=></warning>]
</SoftboolXML_response>
```

In this example Boolware has found an error and reports it in the response.

Raw responses

In order to simplify and speed up parsing at the application, the response can be formatted as column separated values instead of XML. This can be used when tuples (or index terms) are fetched as a response.

Example:

```
<response raw="1">
<records from="1" count="25" maxchars="0">
  <field name="name"/>
  <field name="zip"/>
  <field name="board"/>
```

```
</records>
</response>

Company no  "58"     12345Mr Smith Mr Jones
```

By default, fields are separated by tab and tuples by carriage return + line feed. This can be changed using the "fieldsep" and "rowsep" attributes. Let’s say that you want to change the field separator to a tab surrounded by hash marks:

```
<response raw="1" fieldsep="#\t#">
```

Note that since the example data contains tabs and new lines, these must be changed to space. Otherwise the response would be impossible to interpret. This is a simple solution, but has the drawback that data is modified. If this is a problem to you, there is a way to overcome this – by quoting data.
Value of the attribute 'quotes' can be set to: 'yes', '1', 'no', '0' or litterals '&quot;' or '&apos;'

```
<response raw="1" fieldsep=";" quotes="1">
<records from="1" count="25" maxchars="0">
  <field name="name"/>
  <field name="zip"/>
  <field name="board"/>
</records>
</response>

"Company no; ""58""";"12345";"Mr Smith
Mr Jones"
```

Note that the quotes around "58" are doubled, and that the semicolon (the field separator) in the name and the embedded line break in the Board field are kept intact in the response.

Export retrieved records to a file

Example:
Retrieve all records in the table "Bromma" using the $pk index and then export field data from the specified columns "CompanyName," "PostalCode," and "City" to the file "f:\temp\file.txt" for all retrieved records. Sort the records by "CompanyName" and enclose the data in quotation marks.

```
<SoftboolXML_request type="query">
<open_session name="" queryhistory="0"/>
<database name="Bromma"/>
 <table name="Bromma"/>
 <query>FIND $pk:*</query>
 <response raw="1" rowsep="\r\n" fieldsep="\t" quotes="1">
  <sort expression="CompanyName asc"/>
  <records exportresult="1" exportfile="f:\temp\file.txt" from="1"
count="all" maxchars="0" columnnames=""
          replacecolumnnames="" randomfetch="" exportencoding=""
exportappend="" exportexcludesubfields="">
   <field name="CompanyName"/>
   <field name="PostalCode"/>
   <field name="City"/>
  </records>
 </response>
</SoftboolXML_request>
```

To export records to a file, certain attribute values need to be specified in different elements, and here is a description of these attributes.

The "*raw*" attribute must be set to '1' in the "response" element, see "Tab-separated response," and the "*exportresult*" and "*exportfile*" attributes must be specified and contain '1' and a valid file name, respectively, in the Boolware server environment for the "records" element.

The "*rowsep*" and "*fieldsep*" attributes have default values of 'CRLF' and 'TAB' but can be changed in the "response" element. If data needs to be enclosed in quotation marks, it should also be specified as an attribute in the "response" element with the "*quotes*" attribute set to '1' (the default value is '0').

In the "sort" element, a sorting criterion can be specified in the "*expression*" attribute, e.g.: <sort expression="CompanyName asc, City desc"/> - to sort the exported result first by "CompanyName" in ascending order and then by "City" in descending order.

The "field" element with the "*name*" attribute specifies which field or fields that should be included in the export and in what order.

The following attributes can be specified in the "records" element:

The "*exportresult*" attribute should be set to '1' for exporting the retrieved result, and the "*exportfile*" attribute should be assigned a valid file name. The file with the specified name will be created by the Boolware Server, and all exported records will be stored in this file.

The "*from*" attribute indicates from which hit the data should be exported, and the "*count*" attribute specifies the number of records to be exported. If "*count*" is set to the value 'all', all retrieved records from the starting record ("*from*") will be exported.

The "*maxchars*" attribute can be set to a value greater than '0' indicating the maximum number of characters to be exported from each field. The default value is '0' which means that all data in the field will be exported.

The "*columnnames*" attribute is set to '1' if field names should be exported to the first row of the exported file. If field names should be exported but replaced with other field names, this can be specified using a comma-separated list of field names in the "*replacecolumnnames*" attribute.

The "*randomfetch*" attribute can be set to '1' if the records in the export should be exported in random order, regardless of what is specified in the "sort" element.

The "*exportencoding*" attribute controls the format of data in the output file. Accepted values are 'iso-8859-1', 'utf8', 'xls' or 'xlsx' with the default value in the exported data being the same as the encoding in the XML request.
'xls' is the old Excel format, and 'xlsx' is the new format (MS Excel Open XML format).
The exported file cannot exceed 2 GB and cannot contain more than 65,535 identical strings if "exportencoding" is set to 'xls'.

If the "*exportappend*" attribute is set to '1' the exported records will be appended to the end of the selected output file. The "exportappend" attribute is not used if either 'xls' or 'xlsx' is selected for the "exportencoding" attribute.

The "*exportexcludesubfields*" attribute should be set to '1' if XML subfields should be excluded from the export.

Response for a request of type Query

If the request type is *'query'*, the following elements could occur depending on the outcome: *'records'*, *'record'* and *'field'*.

The element ’records’ has the following attributes: *’total’*, *’from’*, *’to’* and *’rank’*. The attribute ’total’ holds the total number of found records. This attribute will always contain a value. The attribute *’from’* contains the same value that was set in the request part, while the attribute *’to’* holds the number of the last fetched record (the same as set in the request except when there were no more records to fetch). E.g. if you specify in the request *’from’*="**1**" and *’to’*="**100**" and there only are 37 records that fulfills your query the *’to’* attribute will be changed to "**37**" in the response part. The attribute *’rank’* could be changed by Boolware Server together with a corresponding warning message, if a non-existing rank mode was specified in the request. E.g. you want to rank regarding occurrence (**1**) but the Column is not indexed in this way thus the rank mode will be reset to "**0**" (no ranking) and a warning message will be generated.

The element *’record’* contains every requested record.

The following attributes are valid: *’score’*, where the score for the current record is stored. The score could represent different things depending on what ranking mode has been requested. After a Boolean query and no ranking (**0**) the score is 1.000 for all records. If ranking mode is set to occurrence (**1**) in a Boolean query the number of occurrences of the search terms in each record will be presented as an integer. If the rank mode is frequency (**2**) a value between **0** and 1 will represent the score in the form 0.xxx. If similarity search is requested (see Chapter 1 "*Softbool Query language*"), the score is also a value between 0 and 1.

The contents of the requested record is stored in the element *’field’*. In the attribute *’name’* the name of the current Column is saved followed by the requested information for that Column. An element ’field’ is sent for each requested Column in each record.

Example:
```
<records total="7" from="1" to="2" rank="occurrency">
   <record score="23">
      <field name="Name">John Andersson</field>
      <field name="Address">1124 Oak Road</field>
      <field name="Zipcode">112 45</field>
      <field name="City">Boston</field>
   </record>
   <record score="4">
      <field name="Name">Ralph Johnson</field>
      <field name="Address">2134 South Drive</field>
      <field name=" Zipcode">114 15</field>
      <field name="City"> Boston </field>
   </record>
</records>
```

A request has generated **7** records and in the request the records should be ranked according to occurrence **1** (**occ**) and the first two records should be presented. From each record the information of four Columns are requested: Name, Address, Zipcode and City.

"John Andersson" is ranked before "Ralph Johnson", as the number of occurrences in the first record is **23** compared to **4** for the second record.

Response for a request of type Index

If the type of the request was index the following attributes are valid for the element *’records’*: *’field’* and *’total’*.

The name of the Column from which the index terms were fetched are saved in *’field’* and the attribute *’total’* holds the total number of index terms delivered in this response.

In the element *’record’* the attributes *’term’* and *’count’* contains an extracted index term.

The attribute *’term’* holds the index term and in the attribute *’count’* a value is stored. This value indicates in how many records the current index term occurs in the entire database.

If the attribute *’zoom’* is set to "yes", the value in the attribute *’count’* reflects the number of occurrences in the number of records in the current result rather than in the entire database.

Example:
```
<records field="Name" total="10">
   <record term="JOHNSON" count="339"/>
   <record term="JOLSON" count="97"/>
   <record term="JONSON" count="439"/>
   <record term="JUKE" count="3"/>
   <record term="KALMA" count="2"/>
   <record term="KATZ" count="21"/>
   <record term="KAUFMAN" count="14"/>
   <record term="KEMPF" count="3"/>
   <record term="KULPERT" count="4"/>
   <record term="LARSEN" count="12"/>
</records>
```

From a database you want to fetch **10** index terms contained in the Column ’Name’ from a Boolware Index. In the request the word "**Johnson**" is specified in the attribute *’start_position’*.

Response containing Query History

By specifying "**1**" in the attribute *’queryhistory’* in the element *’open_session’* in the request, the Query History will be activated in Boolware. When Query History is activated all queries and responses from the last FIND command are saved in Boolware.

By specifying "**1**" in the attribute *’queryhistory’* in the element *’response’* Boolware will send the Query History in the response.

The element *’queryhistory’* in the response part contains an attribute *’total’*, where the total number of rows of the Query History to be sent is saved. Each row represents a query and its response.

Each row in the Query History has an element *’queryitem’* containing two attributes: *’result’* and *’intermediate’*. The attribute *’result’* holds the total result, while the attribute *’intermediate’* holds the result of the current query.

At the end of each row the query is saved (only the 128 first characters of the query is sent back).

Example:
```
<queryhistory total="4">
   <queryitem result="1403" intermediate="1403"
   FIND JOHN </queryitem>
   <queryitem result="706" intermediate="29456"
   AND ANDERSON </queryitem>
   <queryitem result="350" intermediate="47987"
   AND BOSTON </queryitem>
   <queryitem result="296" intermediate="12453"
   NOT SALESMAN </queryitem>
</queryhistory>
```

Four (**4**) queries are contained in this search session. In the first query you search for "*John*" and the reply is **1.403** records.

In this case the *’result’* and ’*intermediate*’ are the same (this is the case for all FIND commands).

Next query is a refinement of the last result: AND "*Anderson*", which means that both "*John*" and "*Anderson*" must appear in the records. In this case the result is **706**; **706** records contains both "*John*" and "*Anderson*", but "*Anderson*" appears in **29.456** records, which is indicated in the attribute *'intermediate'*.

The next query indicates that the term "*Boston*" also must be contained in the found records.

The result after this query reduces the number of records to **350** which is stored in *'result'* and the number of records containing the term "**Boston**", **47.987**, is saved in *'intermediate'*.

The last query will skip all records containing a "*salesman*". The final result is **296** records; and "*salesman*" is contained in **12.453** records.

Response for Statistics

By specifying the element *'statistics'* containing the attributes: *'column', 'groups', 'onall'* and '*groupvalues*' you could get statistics on a numeric Column. The statistics is - by default - calculated on the current result. By the attribute '*onall*' you could specify that all records in the table should be contained in the calculation.

Example1:
Get statistics on the numeric column "Solidity". Perform the calculation on the current result (all companies in Stockholm) and set number of groups to 5 (quintile). You also want all the limit values for the current group.

Request:
```
<?xml version="1.0" encoding="ANSI"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <open_session name="" queryhistory="1"/>
  <database name="Companies"/>
  <table name=" Companies"/>
  <query>FIND City:Stockholm</query>
  <response queryhistory="1" type="" href="">
    <statistics column="Solidity" groups="5" getallvalues="1"/>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

Response:
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<SoftboolXML_responses>
<SoftboolXML_response type="query" error_code="0" databasename="Companies"
flowexit="" >
<session>MySession</session>
<records total="64428" tablename="Companies">
</records>
<queryhistory total="1">
<queryitem result="64428" intermediate="64428">FIND City:Stockholm</queryitem>
</queryhistory>
 <statistics column="Solidity" tablename="Companies" groups="5" onall="0">
  <statistic name="count" value="38283"/>
  <statistic name="modecount" value="3095"/>
  <statistic name="mode" value="100.000000"/>
  <statistic name="sum" value="-779180537.840000"/>
  <statistic name="avg" value="-20353.173415"/>
  <statistic name="min" value="-425099700.000000"/>
  <statistic name="max" value="1326.410000"/>
  <statistic name="std" value="2396790.246590"/>
  <statistic name="var" value="5744603486147.931600"/>
  <statistic name="median" value="38.870000"/>
  <statistic name="upper" value="82.435000"/>
```

```
    <statistic name="lower" value="9.500000"/>
    <statistic name="groups" values="9.500000, 27.825000, 51.435000, 82.435000/>
  </statistics>
</SoftboolXML_response>
</SoftboolXML_responses>
```

where:

count	number of records contained in the statistics
modecount	number of records containing mode
mode	the most common value within this statistics
sum	the sum of all values
avg	the arithmetic mean value
min	the lowest value
max	the highest value
std	the standard deviation
var	the variance
median	the median value
upper	upper limit for the specified 'group'
lower	lower limit for the specified 'group'
groups	all limit values for the specified 'group'

Request:

```
<?xml version="1.0" encoding="ANSI"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <open_session name="" queryhistory="1"/>
  <database name="Companies"/>
  <table name=" Companies"/>
  <query>FIND City:Stockholm</query>
  <response queryhistory="1" type="" href="">
    <statistics column="Liquidity" groups="10" onall="1" getallvalues="1"/>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

Response:

```
     <?xml version="1.0" encoding="iso-8859-1"?>
     <SoftboolXML_responses>
     <SoftboolXML_response type="query" error_code="0" databasename="Companies"
     flowexit="">
<session>MySession</session>
     <records total="64428" tablename="Companies">
     </records>
     <queryhistory total="1">
     <queryitem result="64428" intermediate="64428">FIND Ort:Stockholm</queryitem>
     </queryhistory>
      <statistics column="Liquidity" tablename="Företag" groups="10" onall="1">
       <statistic name="count" value="217869"/>
       <statistic name="modecount" value="63"/>
       <statistic name="mode" value="100.000000"/>
       <statistic name="sum" value="610811992.529998"/>
       <statistic name="avg" value="2803.574591"/>
       <statistic name="min" value="-35215300.000000"/>
       <statistic name="max" value="161918100.000000"/>
       <statistic name="std" value="424165.974683"/>
       <statistic name="var" value="179916774079.192990"/>
       <statistic name="median" value="123.810000"/>
       <statistic name="upper" value="538.960000"/>
       <statistic name="lower" value="33.890000"/>
       <statistic name="groups" values="33.890000, 60.820000, 83.630000,
                103.610000, 123.810000, 149.970000, 191.260000, 272.360000,
                538.960000/>
      </statistics>
     </SoftboolXML_response>
     </SoftboolXML_responses>
```

Response for Similarity vectors

Get the similarity vectors from the retrieved result.

The element '*simvectors*' with the attributes *'from'*, *'to'* and *'content'* within the element *'response'*.

If available the query vector always is fetched as the first record and thereafter the requested number of similarity vectors.

If the query vector is not available a warning element is produced instead. The query vector is available if a similarity query has been executed.

The result contains records containing two fields: *'Primary Ke*y' and *'Similarity Vector'* and the attribute *'score'* within the element *'record'* containing the score of the similarity.

The field *'Primary Key'* contains the tuples primary key and *'Similarity Vector'* contains the similarity vector.

The attribute *'content'* is the way to guide if the content terms shall be in text terms or numeric terms. The similarity vectors can be large, containing hundreds of terms.

Example: Get similarity vectors for the first two records in numeric form.
```
<simvectors from="1" count="2" content="numeric"/>
```

The response will be like:
```
<simvectors from="1" to="2" content="number">
 <record score="0.000">
  <field name="Primary Key">Query Vector</field>
  <field name="Similarity Vector">37544/1/</field>
 </record>
 <record score="1.000">
  <field name="Primary Key">556416-5966</field>
  <field name="Similarity Vector">37544/1/</field>
 </record>
 <record score="0.707">
  <field name="Primary Key">556266-4101</field>
  <field name="Similarity Vector">37544/1/,
    94994/1/</field>
 </record>
</simvectors>
```

Response for Set Search

Set search consists of the following commands, *savequery, saveresult, savescratch, reviewset, reviewquery, reviewresult, deleteset, deletequery, deleteresult* and *deletescratch*. These are called with the type *'execute'* in the request.

Example: Save current result with the name 'MyResult'

```
<SoftboolXML_request type="execute">
 <open_session name="charlie" server="192.168.1.1"/>
 <execute>
  saveresult name="MyResult" table="MyTable"
 </execute>
</SoftboolXML_request>

Response of the request:
<SoftboolXML_response type="execute" error_code="0">
```

```
<save type="result"/>
</SoftboolXML_response>
```

The element *’save’* contains the attribute *’type’* that tells the type of the saved query.

Example: Fetch all saved results in ascending name order

```
<SoftboolXML_request type="execute">
 <open_session name="charlie" server="192.168.1.1"/>
 <execute>
  reviewresult name="*" order="name" dir="asc"
 </execute>
</SoftboolXML_request>
```

Response of the request:

```
<SoftboolXML_response type="execute" error_code="0">
<review type="result"/>
 <records total="1">
  <record name="MyResult" ctime="2003-09-09" database="FTimes"
   table="Financial_Times" result="14">
   Text/DOC/HEADLINE:sport AND Text/DOC/TEXT:golf
  </record>
 </records>
</SoftboolXML_response>
```

The element *’review’* contains the attribute *’type’* that contains the query type.

The element *’records’* contains the attribute *’total’* that is the number of ‘*record’* that will follows.

The element *’record’* contains the attributes ‘*public’* the group identification for a public object, *’name’*, the name of the saved query, *’ctime’* contains the time of the saved query, *’database’* contains the database, ‘*table’* contains the table for the query. If the type is *’result* ’or *’set’* the attribute *’result’* is the number of hits for the saved query.

Example: Fetch all saved global results in ascending name order for the group SalesDep

```
<SoftboolXML_request type="execute">
 <open_session name="charlie" server="192.168.1.1"/>
 <execute>
  reviewresult public="SalesDep" name="*" order="name" dir="asc"
 <execute>
</SoftboolXML_request>

Response of the request:
<SoftboolXML_response type="execute" error_code="0">
<review type="result"/>
 <records total="1">
  <record public="SalesDep" name="MyResult" ctime="2003-09-09"
   database="FTimes" table="Financial_Times" result="14">
   Text/DOC/HEADLINE:sport AND Text/DOC/TEXT:golf
  </record>
 </records>
</SoftboolXML_response>
```

Example: Delete the result with the name ’MyResult

```
<SoftboolXML_request type="execute">
 <open_session name="charlie" server="192.168.1.1"/>
 <execute>
  deleteresult name="MyResult"
 </execute>
</SoftboolXML_request>

Response of the request:
```

```
<SoftboolXML_response type="execute" error_code="0">
<delete type="result" total="1"/>
</SoftboolXML_response>
```

The element *’delete’* contains the attributes *’type’* that contains the query type of the deleted query and *’total’* contains the number of queries deleted.

Response for metadata

Fetched metadata from Boolware Server is hierarchically ordered with element database as the database top element and the element field with its attribute the lowest.
The element ’metadata’ contains all metadata.
The element ’databases’, attribute ’count’ holds number of databases that follows.
The element ’database’ is top for all databases in the reply.
The attribute ’name’ holds the name of the database.
The attribute ’descr’ holds the description of the database.
The attribute ‘provider’ holds the database provider.
The attribute ’status’ holds the database status.

The element ’tables’, attribute ’count’ holds number of tables that follows.
The element ’table’ is the top of each table in the reply.
The attribute ’schema’ holds the schema of the table.
The attribute ’name’ holds the name of the table.
The attribute ’status’ holds the status of the table.
The attribute ’count’ hold number of rows in the table.

The element ’fields’, attribute ’count’ holds the number of fields that follows.
The element ’field’ is the start of each field that follows.
The attribute ’name’ holds the name of the field.
The attribute ’datatype’ holds the type of the field in ODBC constants.
The attribute ’datasize’ holds the size of the field.
The attribute ’precision’ holds the size of the field.
The attribute ’scale’ holds the number of decimals of the field if field is float type.

The element ’attribute’ holds all properties of the field in the Boolware Server system. The value for these attributes is ’0’ for ’no’ and ’1’ for ’yes’.
The attribute ’word’ holds word indexed.
The attribute ’string’ holds string indexed.
The attribute ’proximity’ holds proximity.
The attribute ’phonetic holds phonetic indexed.
The attribute ’similarity’ holds similarity indexed.
The attribute ’lefttrunc’ holds the reversed indexed.
The attribute ’proxline’ holds proximity line.
The attribute ’compress’ holds interpunctuated abbreviations.
The attribute ’permutate’ holds permutated strings.
The attribute ’stemmed’ holds stemmed indexd.
The attribute ’cluster’ holds grouped strings.
The attribute ’rank’ holds rankable.
The attribute ’fieldsearch’ hold search scoop field.
The attribute ’freetext’ hold the search scoop free text.
The attribute ’alias’ holds non indexed alias field.
The attribute ’aliasindex’ holds indexed alias field.
The attribute ’markuptags’ holds ignore HTML tags.
The attribute ’stopwords’ holds active stop words.
The attribute ’foreignkey’ holds foreign key.
The attribute ’dataxml’ holds field containing xml data.
The attribute ’subfield’ holds xml-field.
The attribute ’virtual’ holds virtual field.

The attribute’ category’ used for categorization
The attribute ‘datafield’ data stored in Boolware data file
The attribute ’searchterm’ marked as search term
The attribute ’prefix’ field content used to prefix xml-content words
The attribute ’attribute’ is xml-attribute
The attribute ‘datautf8‘ field contains utf-8 encoded data
The attribute ’case’ indexed case sensitive
The attribute ’memmap’ field is memory mapped
The attribute ’presort’ field is presorted
The attribute ’geopos’ geografic Long or Lat in WGS84 format
The attribute ’geometer’ geografic meter format e.g. RT90
The attribute ’geomult’ geografic field containing both Lat and Long
The attribute ’within’ field index with Within
The attribute ‘wordasis’ word indexed as is
The attribute ‘stringasis’ string indexed as is
The attribute ‘withinstring’ string indexed ‘Within string’

```
<metadata>
 <databases count="">
  <database name="" descr="" status="">
   <tables count ="">
    <table schema="" name ="" status=""
     count="">
     <fields count="">
      <field name="" datatype=""
       datasize="" precision="" scale=""
       pksequence="">
       <attributes word="" string =""
        proximity="" phonetic =""
        similarity ="" lefttrunc =""
        proxline ="" compress =""
        permutate = "" stemmed = ""
        cluster ="" rank =""
        fieldsearch ="" freetext =""
        alias ="" aliasindex ="" markuptags =""
        stopwords ="" foreignkey = ""
        dataxml ="" subfield =""
        virtual ="" category="" datafield="" searchterm="" prefix=""
         attribute="" datautf8="" case="" memmap="" presort=""
        geopos="" geometer="" geomult="" wordasis="" stringasis="" withinstring=""/>
      </field>
     </fields>
    </table>
   </tables>
  </database>
 </databases>
</metatdata>
```

LOOKUP

Is a way of getting data from another table than the one that is currently in use in the xml request part. Of course you could perform a "lookup" within the table currently in use as well.

Request

Lookup have the following grammar and is a sub-element to the element `<field>` in the response part of the xml request. The element `<field>` should be repeated for every field that should be fetched.

```
<records from="" count="">
 <!—- field -->
 <field name="" dataenclosedelement="">
```

```
  <lookup replace="" subrecordlimit="" sort="">
   <lookuptable name=""/>
   <lookupfield src="" target=""/>
   <field name=""/>
   <field name=""/>
   <!—Nested lookup -->
   <field name="">
    <lookup>
    <lookuptable name=""/>
    <lookupfield src="" target=""/>
    <field name=""/>
    <field name=""/>
    </lookup>
  </lookup>
 </field>
 <field name=""/>
 <field name=""/>
</records>
```

Note!
A valid lookup must have: one `<lookuptable name=""/>` element, and at least one `<lookupfield src="" target=""/>` element and at least one `<field name=""/>` element to be performed.

Element:
`<lookup [replace=""] [subrecordlimit=""] [sort=""] [suppressempty=""] />`

Start and description of the requested lookup for the parent `<field>` element

Attribute:

replace	replace the parent field; default is 0
subrecordlimit	number of sub-records that should be printed in the response; default is 1
sort	a sort expression to sort the lookup records that should be fetched in those cases were the lookup generates more than on record; the sort can be on any field in the target table, Softbool AB recommends that the string index type is set on fields that should be sorted; default is no sort
suppressempty	Available if **not** *raw* request. Suppress empty subrecords; default is 0

Element:
`<lookuptable name=""/>`

The target table where the lookup should take place; query and fetch records

Attribute:

name	the name of the target table

Element:
`<lookupfield [src=""] [target=""] [autotrunc=""] [value=""] [method=""] [op=""] [flowvariablevalue=""]/>`

Here is the query part of the lookup described. If more than one `<lookupfield>` element is given the Boolean command AND is done between all lookup fields.

Attribute:

src	the name of the field in the source table where data is fetched for the query
target	the name of field in the target table. Can be empty if field is specified in the value instead
autotrunc	if the query should be right truncated; default is 0
value	replace the column data value from the src data field as written. Max size is 256 bytes. Default method in this case is word.

method	apply search method *word, string, syn, thes, sound, stem, near, fuzzy, stringasis* eller *wordasis*. The query will performed with the proper subcommand: string(), syn(), thes(), sound(), stem(), near(), fuzzy(), stringasis() eller wordasis(); default is *string*.
op	set the default operator *AND* or *OR* between search terms; Default is *AND*
flowvariablevalue	Use a variable that can be set in a flow. If the variable is found and contains a value, this value will be used in the search field specified in the "*target*". This has a higher priority than "value". Default method is in this case word

Element:
```
<field name="" [aliasname=""] [conditionalreplacevalue=""]
[conditionalcomparefield=""] [nohitreplacevalue=""]
[nohitkeepparentvalue=""]/>
```

The field that should be fetched in the response part.

Attribute:

name	name of the datafield in the query table
aliasname	replace the name attribute in the `<field>` or `<subfield>` element in the response part
conditionalreplacevalue	replace content of a field containing a specified phrase or term e.g. *conditionalreplacevalue="'*';'NOT ALLOWED'"*
conditionalcomparefield	name of the field whose contents are matched against given criteria in *conditionalreplacevalue.* Default is the same datafield name that is specified in the *name* attribute.
nohitreplacevalue	if the lookup query does not generate any hit, the given value will be used in the response of the <subfield> element
nohitkeepparentvalue	if the lookup query does not generate any hit, the source field value where you placed the lookup will be used in the response of the <subfield> element. If set to 1, this setting has higher priority than *nohitreplacevalue.* Default is 0.

For more information about <field> attributes, see Softbool XML Request Schema.

Response

Hierarchy if the attribute **replace** is off in the request lookup:
```
<record score="1.000">
 <field name="">
  <subrecord>
   <subfield name=""></subfield>
  </subrecord tablename="">
 </field>
</record>
```

Element:
```
<field name=""/>
```

Starts the data of a requested field

Attribute:

name	If the attribute *aliasname* is given in the request part of the `<field>` element it is the alias name that is given for the *name* attribute; otherwise it is the actual field name from the table

Hierarchy if the attribute replace is given in request lookup:
```
 <record score="1.000">
 <subrecord>
  <subfield name=""></subfield>
 </subrecord>
</record>
```

Element:
`<subrecord tablename=""/>`

Start of one or more `<subfield>` element.

Attribute:

tablename	name of the table for the *subrecord* element

Element:
`<subfield name=""/>`

Start data for given lookup field in the request part.

Attribute:

name	If the attribute *aliasname* is given in the request part of the `<field>` element it is the alias name that is given for the *name* attribute; otherwise it is the actual field name from the table

Lookup example

Search in a table containing company information. This table contains one field with the SIC code for the company. Here we want to add the correlated text for the code in the response part. This corresponding text is stored in another table in the same database.

The table with the company information is named "CompanyInfo" and the table with the sic code text is named "SIC".

The table "CompanyInfo" contains among other fields: "CompanyName", "City", SICCode" and "CompanyLang".

The table SIC have three fields: "Code", "Lang" and "Info"; were the "Code" field contains the SIC code, the "Lang" contains a language code and "Info" contains the actual describing text.

Get all companies from the company info table that have the word "IKEA" in its name.
Sort the result in company name ascending order and the fetch data from the fields: "CompanyName", "SICCode", "City" and "CompanyLang".

We also perform a lookup to get the describing text for the SIC code by using the element `<lookup>`.

Also apply an alias name, "Lookuptext", to replace the field name "Info" in the response part.

Build the xml request:
```
<?xml version="1.0" encoding="UTF-8"?>
<SoftboolXML_requests>
<SoftboolXML_request type="query">
<open_session name="" queryhistory="1"/>
<database name="Companies"/>
<table name="CompanyInfo"/>
<query>FIND CompanyName:IKEA </query>
<response queryhistory="0" type="" href="">
<sort expression="CompanyName asc"/>
<records from="1" count="2" maxchars="0" >
<field name="CompanyName"/>
<field name="SICCode" dataenclosedelement="value">
 <lookup subrecordlimit="3">
  <lookuptable name="SIC"/>
  <lookupfield src="SICCode" target="Code"/>
  <field name="Lang"/>
  <field name="Info" aliasname="Lookuptext"/>
 </lookup>
</field>
<field name="City"/>
<field name="CompanyLang"/>
</records>
</response>
</SoftboolXML_request>
</SoftboolXML_requests>
```

The response could be:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SoftboolXML_responses>
<SoftboolXML_response type="query" error_code="0" databasename="Companies" flowexit="" >
<session>CARLOS</session>
<records total="13935" from="1" to="2" rank="sort asc" tablename="CompanyInfo">
<record score="1.000">
<field name="CompanName">IKEA AB</field>
<field name="SICCode"><value>361</value>
 <subrecord tablename="SIC">
  <subfield name="Lang">SE</subfield>
  <subfield name="Lookuptext">Tillverkning av möbler</subfield>
 </subrecord>
 <subrecord tablename="SIC">
  <subfield name="Lang">EN</subfield>
  <subfield name="Lookuptext">Manufacturing of furniture</subfield>
 </subrecord>
 <subrecord tablename="SIC">
  <subfield name="Lang">EN</subfield>
  <subfield name="Lookuptext">Manufacturing of furniture, tables</subfield>
 </subrecord>
</field>
<field name="City">BAGARMOSSEN</field>
<field name="CompanyLang">SE</field>
</record>
<record score="1.000">
<field name="CompanyName">IKEA Sweden AB</field>
<field name="SICCode">505
 <subrecord tablename="SIC">
  <subfield name="Lang">SE</subfield>
  <subfield name="Lookuptext">Detaljhandel inom drivmedel</subfield>
 </subrecord>
 <subrecord tablename="SIC">
  <subfield name="Lang">EN</subfield>
  <subfield name="Lookuptext">Retaling fuel</subfield>
 </subrecord>
 <subrecord tablename="SIC">
  <subfield name="Lang">EN</subfield>
  <subfield name="Lookuptext">Retaling fuel, oil</subfield>
 </subrecord>
</field>
<field name="City">STOCKHOLM</field>
<field name="CompanyLang">EN</field>
</record>
</records>
</SoftboolXML_response>
</SoftboolXML_responses>
```

Refine the lookup by only fetching the text that correlates with the language code for the company as well, the field "LanguageCode".

To do that we add an extra `<lookupfield src="" target=""/>` element to the lookup to achieve a Boolean AND command to get only the records with the correct SIC code and correct language code from the company record.

```
<?xml version="1.0" encoding="UTF-8"?>
<SoftboolXML_requests>
<SoftboolXML_request type="query">
<open_session name="" queryhistory="1"/>
<database name="Companies"/>
<table name="CompanyInfo"/>
<query>FIND CompanyName:IKEA</query>
```

```
<response queryhistory="0" type="" href="">
<sort expression="CompanyName asc"/>
<records from="1" count="2" maxchars="0" >
<field name="CompanyName"/>
<field name="SICCode">
 <lookup subrecordlimit="3">
  <lookuptable name="SIC"/>
  <lookupfield src="SICCode" target="Code"/>
  <lookupfield src="CompanyLang" target="Lang"/>
  <field name="Lang"/>
  <field name="Info" aliasname="Lookuptext"/>
 </lookup>
</field>
<field name="City"/>
</records>
</response>
</SoftboolXML_request>
</SoftboolXML_requests>
```

The response of the above will be:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SoftboolXML_responses>
<SoftboolXML_response type="query" error_code="0" databasename="Companies" flowexit="">
<session>CARLOS</session>
<records total="13935" from="1" to="2" rank="sort asc" tablename="CompanyInfo">
<record score="1.000">
<field name="CompanName">IKEA AB</field>
<field name="SICCode">361
 <subrecord tablename="SIC">
  <subfield name="Lang">SE</subfield>
  <subfield name="Lookuptext">Tillverkning av möbler</subfield>
 </subrecord>
 <subrecord tablename="SIC">
  <subfield name="Lang">SE</subfield>
  <subfield name="Lookuptext">Detaljhandel inom drivmedel</subfield>
 </subrecord>
 <subrecord tablename="SIC">
  <subfield name="Lang"></subfield>
  <subfield name="Lookuptext"></subfield>
 </subrecord>
</field>
<field name="City">BAGARMOSSEN</field>
<field name="CompanyLang">SE</field>
</record>
<record score="1.000">
<field name="CompanyName">IKEA Sweden AB</field>
<field name="SICCode">505
 <subrecord tablename="SIC">
  <subfield name="Lang">EN</subfield>
  <subfield name="Lookuptext">Retaling fuel</subfield>
 </subrecord>
 <subrecord tablename="SIC">
  <subfield name="Lang">EN</subfield>
  <subfield name="Lookuptext">Retaling fuel, oil</subfield>
 </subrecord>
 <subrecord tablename="SIC">
  <subfield name="Lang"></subfield>
  <subfield name="Lookuptext"></subfield>
 </subrecord>
</field>
<field name="City">STOCKHOLM</field>
<field name="CompanyLang">EN</field>
</record>
</records>
</SoftboolXML_response>
</SoftboolXML_responses>
```

Here it will be one `<subrecord>` element with no data because the attribute *subrecordlimit* is set to 3 in the `<lookup>` element in the request and it was only two lookup records that matched the Boolean AND query with SIC-code and language code from the company record.

Lookup example with conditional replace

In this example we want to lookup every phonenumber to see if we are allowed to display it.

```
<?xml version="1.0" encoding="UTF-8" ?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <database name="Person" />
  <table name="Person" />
  <query>Find PhoneNumber:*</query>
  <response queryhistory="0">
    <records from="1" count="25">
      <field name="PhoneNumber">
        <lookup replace="1" subrecordlimit="1" >
          <lookuptable name="NotAllowedPhoneNumbers" />
          <lookupfield src="PhoneNumber" target="PhoneNumber" method="word" />
          <field name="PhoneNumber" nohitkeepparentvalue="1"
            conditionalreplacevalue="'*';'NOT ALLOWED'"/>
        </lookup>
      </field>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

Example with conditional replace <u>without lookup</u>

In this example we want to check if phonenumber is allowed to be displayed or not.

```
<?xml version="1.0" encoding="UTF-8" ?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <database name="Person" />
  <table name="Person" />
  <query>Find PhoneNumber:*</query>
  <response queryhistory="0">
    <records from="1" count="25">
      <field name="PhoneNumber"
                  conditionalreplacevalue="'1';'NOT ALLOWED'"/>
                  conditionalcomparefield="NIX"/>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

Example XML

Below you will find some complete XML scripts where both the request and the response will be shown.

As mentioned earlier Boolware should be used interactively; one user (session) should keep its logical connection to Boolware during a long time to refine the search result by using the Boolean operators stepwise. Using this technique makes it possible to fetch information from the data source at any time without re-searching; perhaps you want to re-order the records by sorting them between two presentations.

All examples are made within the same session, but of course it is possible to start several sessions in Boolware.

The conditions for the below examples are:
In a huge database, *Newspapers*, which contains millions of articles from different magazines you want to perform different types of queries.

The database contains one Table, *Articles*, and the following Columns exist: *Article no.*, *Magazine*, *Publishing date*, *Author*, *Category*, *Size* and *Text*.

The Column *Article no*. contains a unique identification (Primary key) for the articles stored in the database. The Column *Magazine* contains the name of the magazine where the article was published. Publishing *date* contains the date the current article was published in the form **yyyymmdd**, the name of the author is stored in the Column *Author*. The type of article: sport, domestic politics, foreign politics, culture, finance etc. is saved in the Column *Category*. The Column *Size* contains the size of the article in words. Finally the body of the article is stored in the Column *Text*.

Example 1   A simple query

You want to find all articles that contain the word **car**. Probably it is the within the Column Text you should search to find the wanted articles. The simple query looks like this:

FIND *Text*:**car**. Lets say that there are 12.435 articles containing the word car. This time no articles should be fetched from the data source.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query" queryhistory="1">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <query>FIND Text:car</query>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

As no articles should be fetched from the data source the *'response'* element should be omitted.

The answer from Boolware Server will be:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records total="12435"/>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The only answer in this case is: type of request (query), an indication that the request went OK (error_code = **0**), what session it is (**User1**) and the result of the query **12.435**.
User1 is the session that will be used in all examples.

Despite no Query History is requested it must be specified for the FIND command if you - later on - within this search session (between two FIND commands) want to fetch the Query History.

Example 2   Refined query with response

By specifying more search criteria you could very easily reduce the number of articles:
AND Publishing date:>=**20000101**.

This query will give all articles about cars published this century. To continue from the previous example it is very important to specify the very same session, **User1**, as specified in the previous call. This time you want the three (3) articles containing most search terms to be fetched from the data source. The Columns you want information from are: *Article no*., *Publishing date* and *Text*. A maximum of 20 characters from each Column should be fetched.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query" queryhistory="1">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <query>AND "Publishing date":&gt;=20000101</query>
  <response>
    <records from="1" count="3" rank="occurrency" maxchars=20>
      <field name="Article no."> </field>
      <field name="Publishing date"> </field>
      <field name="Text"> </field>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

After this query the number of articles is reduced to **735**.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records total="735" from="1" to="3" rank="occ"/>
    <record score="14">
      <field name="Article no."> 8432 </field>
      <field name="Publishing date"> 20000912 </field>
      <field name="Text"> Article about Ford...</field>
    </record>
    <record score="12">
      <field name="Article no."> 736 </field>
      <field name="Publishing date"> 20000422 </field>
      <field name="Text"> Article about Saab...</field>
    </record>
    <record score="7">
      <field name="Article no."> 12456 </field>
      <field name="Publishing date"> 20000714 </field>
      <field name="Text"> Article about Volvo...</field>
    </record>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

This time you will get the requested articles along with the result. As the articles are requested in a certain order (number of occurrences), the first article is the one containing most search terms (**car** and >=**20000101**; **14** times).

Example 3   Refine and Query History

Another refinement should be performed; only articles that belong to the category **culture** should be in the result.

This time you want the articles in frequency order and as several queries have been performed the Query History should be contained in the response.

The same Columns should be fetched this time.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <open_session name="User1" queryhistory="1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <query>AND Category:culture</query>
  <response queryhistory="1">
    <records from="1" count="3" rank="frequency" maxchars=20>
      <field name="Article no."> </field>
      <field name="Publishing date"> </field>
      <field name="Text"> </field>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

After this query the number of articles is reduced to **5**.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session> User1</session>
  <records total="5" from="1" to="3" rank="freq"/>
    <record score="0.338">
      <field name="Article no."> 12456 </field>
      <field name="Publishing date"> 20000714 </field>
      <field name="Text"> Article about Volvo...</field>
    </record>
    <record score="0.306">
      <field name="Article no."> 8432 </field>
      <field name="Publishing date"> 20000912 </field>
      <field name="Text"> Article about Ford...</field>
    </record>
    <record score="0.288">
      <field name="Article no."> 736 </field>
      <field name="Publishing date"> 20000422 </field>
      <field name="Text"> Article about Saab...</field>
    </record>
  </records>
  <queryhistory total="3">
    <queryitem result="12435" intermediate="12435"
     FIND Text:car </queryitem>
    <queryitem result="735" intermediate="10726"
     AND "Publishing date":>=20000101 </queryitem>
    <queryitem result="5" intermediate="2489"
     AND Category:culture </queryitem>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

This time you will get - along with the articles - the Query History. The attribute 'result' shows the result after each query, while the attribute *'intermediate'* tells the number of articles for the current query. In this example the query for *Category*:**culture** generated **2.489** hits.

Example 4   Get the remaining articles

In this example the remaining two articles from the earlier search session should be fetched. Totally there are 5 articles and three of them has already been fetched (Example 3). This example will show how to continue a fetch of result rows without re-searching the database.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <open_session name="User1" queryhistory="1"/>
  <database name="Newspapers"/>
  <table name="Artiklar"/>
  <response queryhistory="1">
    <records from="4" count="2" rank="frequency" maxchars=20>
      <field name="Article no."> </field>
      <field name="Publishing date"> </field>
      <field name="Text"> </field>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
```

```
 <SoftboolXML_response type="query" error_code="0">
  <session> User1</session>
  <records total="5" from="4" to="5" rank="freq"/>
    <record score="0.204">
      <field name="Article no."> 645 </field>
      <field name="Publishing date"> 20000304 </field>
      <field name="Text"> Article about Simca...</field>
    </record>
    <record score="0.198">
      <field name="Article no."> 1802 </field>
      <field name="Publishing date"> 20000131 </field>
      <field name="Text"> Article about Citroen.</field>
    </record>
  </records>
  <queryhistory total="3">
    <queryitem result="12435" intermediate="12435"
     FIND Text:car </queryitem>
    <queryitem result="735" intermediate="10726"
     AND "Publishing date":>=20000101 </queryitem>
    <queryitem result="5" intermediate="2489"
     AND Category:culture </queryitem>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

In this example you do not need to specify any query as you should use the result from Example 3; you should just fetch the two remaining articles. The Query History will, of course, be exactly the same and thus could be omitted as well.

Example 5   Sort the result

In this example it is only the presentation order of the found articles that differs. Instead of presenting the articles in frequency order they should appear in chronological order. The article published first should be presented first. If the sort column "Publishing date" has no data it should be "sorted" first. All articles should be fetched. As no query is performed the Query History is omitted.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <response>
    <sort expression="&quot;Publishing date&quot; asc emptydata=first"/>
    <records from="1" count="5" maxchars=20>
      <field name="Article no."> </field>
      <field name="Publishing date"> </field>
      <field name="Text"> </field>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records total="5" from="1" to="5" rank="sort asc"/>
    <record score="1.000">
      <field name="Article no."> 1802 </field>
      <field name="Publishing date"> 20000131 </field>
      <field name="Text"> Article about Citroen.</field>
    </record>
    <record score="1.000">
      <field name="Article no."> 645 </field>
      <field name="Publishing date"> 20000304 </field>
      <field name="Text"> Article about Simca...</field>
    </record>
      <record score="1.000">
      <field name="Article no."> 736 </field>
      <field name="Publishing date"> 20000422 </field>
      <field name="Text"> Article about Saab...</field>
    </record>
  <record score="1.000">
      <field name="Article no."> 12456 </field>
      <field name="Publishing date"> 20000714 </field>
      <field name="Text"> Article about Ford...</field>
    </record>
    <record score="1.000">
      <field name="Article no."> 8432 </field>
      <field name="Publishing date"> 20000912 </field>
      <field name="Text"> Article about Volvo...</field>
    </record>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

As the result from Example 3 should be used no query should be performed. All that should be done is to sort the articles before fetching them from the data source. The Query History is omitted as well.

Example 6   Fetch Index terms

In some cases it is necessary to see what index terms there are to search for. Within the Boolware Server there are two ways of fetching the index terms: fetch index terms that is in any record, or fetch index terms that appear in records contained in the current result (*zoom*).
By specify a *'start_position'* you could determine where in the index to start. The number of index terms to be fetched could also be specified.

Assume that you want to see 10 index terms from the Column *'Text'* starting with **car**. All records should be included.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <index field="Text" type="1" start_position="car" max_terms="10"/>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="index" error_code="0">
  <session>User1</session>
  <records field="Text" total="10"/>
    <record term="CAR" count="12435"/>
    <record term="CARAT" count="72"/>
    <record term="CARAVAN" count="12"/>
    <record term="CARBON" count="43"/>
    <record term="CARD" count="2497"/>
    <record term="CARDS" count="14435"/>
    <record term="CARE" count="10435"/>
    <record term="CAREER" count="4851"/>
    <record term="CARES" count="9439"/>
    <record term="CARESS" count="48"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The value that is found in ’count’ reflects in how many records within the entire database the term appears.

Example 7 Fetch Index terms within result

In this example you want to find index terms still within records within the current result. The result is from the following query: FIND *Text*:**car** which gives **12.435** hits.
Assume that you want to see 10 index terms from the Column *’Text’* starting with **car**.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <index field="Text" type="1" start_position="car" zoom="yes"
   max_terms="10"/>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records field="Text" total="10"/>
    <record term="CAR" count="12435"/>
    <record term="CARAT" count="5"/>
    <record term="CARD" count="546"/>
    <record term="CARDS" count="773"/>
    <record term="CARE" count="213"/>
    <record term="CAREER" count="17"/>
    <record term="CARES" count="7"/>
    <record term="CARL" count="3"/>
    <record term="CARVE" count="1"/>
    <record term="CASE" count="14"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The value that is found in *’count’* reflects in how many records within the current result the term appears.

Note that some of the terms no longer are fetched, as they do not appear in the result (CARAVAN, CARBON, and CARESS).

Example 8 Fetch hierarchic index

In this example you want to fetch index terms from a grouped index in hierarchic order. There are three different ways to present a hierarchic index in Boolware:

1. Hierarchic presentation ordered by number of occurrences (*type* = 10)
2. True index order where each group will be presented separately (*type* = 11)
3. Hierarchic presentation in alphabetical order (*type* = 12)

Given is a column ‘Date’ indexed as grouped index in table ‘Person’. The presentation should be ordered by number of occurrences (type = 10). The dates are stored in the following way: yyyymmdd and are grouped like this: 4, 6, 8. The first 10 records are requested.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Register"/>
  <table name="Person"/>
  <index field="Date" start_position="" type="10" max_terms="10" />
 </SoftboolXML_request>
</SoftboolXML_requests>

<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="index" error_code="0">
  <session>User1</session>
  <records field="Date" total="10"/>
    <record term="1945" count="14837"/>
    <record term="194503" count="1318"/>
    <record term="19450321" count="60"/>
    <record term="19450312" count="52"/>
    <record term="19450310" count="48"/>
    <record term="19450327" count="47"/>
    <record term="19450301" count="32"/>
    <record term="19450331" count="30"/>
    <record term="19450302" count="30"/>
    <record term="19450313" count="25"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The numbers in *’count’* the number of occurrences for the corresponding term.

The same table and column as in the above example but this time you want the hierarchic index presented in alphabetical order (type = 12).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Register"/>
  <table name="Person"/>
  <index field="Date" start_position="" type="12" max_terms="10" />
 </SoftboolXML_request>
</SoftboolXML_requests>

<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="index" error_code="0">
  <session>Användare1</session>
  <records field="Date" total="10"/>
    <record term="1904" count="9837"/>
    <record term="190401" count="11"/>
    <record term="19040101" count="60"/>
    <record term="19040102" count="52"/>
    <record term="19040103" count="96"/>
    <record term="19040104" count="147"/>
    <record term="19040105" count="14"/>
    <record term="19040106" count="86"/>
    <record term="19040107" count="32"/>
    <record term="19040108" count="12"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The numbers in *’count’* the number of occurrences for the corresponding term.
To get the next 10 records you just activate the ‘continuation’ attribute.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_request>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Register"/>
  <table name="Person"/>
  <index field="Date" start_position="" type="12" max_terms="10"
continuation="yes"/>
 </SoftboolXML_request>
</SoftboolXML_requests>

<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="index" error_code="0">
  <session>Användare1</session>
  <records field="Date" total="10"/>
    <record term="19040109" count="12"/>
    <record term="19040110" count="19"/>
    <record term="19040111" count="55"/>
    <record term="19040112" count="77"/>
    <record term="19040113" count="21"/>
    <record term="19040114" count="63"/>
    <record term="19040115" count="89"/>
    <record term="19040116" count="16"/>
    <record term="19040117" count="55"/>
    <record term="19040118" count="72"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

Example 9   Fetch Index terms within result

In this example you want to get index terms in frequency order; number of occurrences. The order should be descending; the most frequent terms will be presented first.

Assume that you want to see the 10 most common index terms from the Column *'Text'*. To limit the selection you are only interested in terms that occur in more than 1.000 records.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <index field="Text" type="14" order="desc" freqtype="1"
     start_position=">1000" max_terms="10"/>
 </SoftboolXML_request>
</SoftboolXML_requests>

<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records field="Text" total="10"/>
    <record term="IN" count="104706"/>
    <record term="AND" count="99723"/>
    <record term="ON" count="95468"/>
    <record term="WHICH" count="93732"/>
    <record term="TO" count="92996"/>
    <record term="A" count="92865"/>
    <record term="OF" count="91604"/>
    <record term="WITH" count="90673"/>
    <record term="AN" count="90567"/>
    <record term="FOR" count="87103"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The value that is found in *'count'* reflects in how many records within the current result the term appears.

Note that only terms appearing in more than 1.000 records will be selected. This means that the extracting and sorting of the terms will be considerably faster compared to extract and sort all index terms. In the next example, Example 9, positioning within this selected group of index terms will take place.

Example 10   Positioning within selected group of terms in frequency order

In this case you want to position to a certain frequency before starting to fetch the index terms selected in Example 8. Note that you could only position within the selected index terms; in this case index terms that are contained in more than 1.000 records.

The order should be ascending and you want to start from a term that is contained in 2.000 (or the closest higher frequency) records.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <index field="Text" type="14" order="asc" freqtype="1" start_position="2000"
max_terms="10"/>
 </SoftboolXML_request>
</SoftboolXML_requests>

<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records field="GREATEST" total="10"/>
    <record term="GIVE" count="2001"/>
    <record term="NO" count="2002"/>
    <record term="ON" count="2002"/>
    <record term="DOWN" count="2003"/>
    <record term="THINK" count="2007"/>
    <record term="GONE" count="2008"/>
    <record term="EVERY" count="2011"/>
    <record term="BOTH" count="2012"/>
    <record term="TIME" count="2014"/>
    <record term="LONGER" count="2015"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The value that is found in *'count'* reflects in how many records within the current result the term appears.

Example 11   Fetch specified index terms

In this case you want only index terms that meet certain criteria. By using the Boolware wild cards **(*, ?, !** and #) you could specify a "mask" as *start_position*. Only those terms in the index that match this "mask" will be fetched. This function could of course also be used for zoomed index; just fetch approved index terms that are contained in the current result (*zoom*).

The number of index terms to fetch is specified in the usual way by *max_terms*.

From the column '*Text'* you want to fetch index terms (car models) from the entire index that match the following "mask": !##* (the terms must start with a letter (!) and be followed by at least two digits (##); the * tells that anything could follow. (You could specify the wild cards together with "normal" letters and digits as you like.) In this example only 10 index terms are fetched.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_request>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <index field="Text" type="1" start_position="!##*" max_terms="10"/>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="index" error_code="0">
  <session>User1</session>
  <records field="Text" total="10"/>
    <record term="C70" count="2435"/>
    <record term="S40" count="7295"/>
    <record term="S60" count="124"/>
    <record term="S80" count="435"/>
    <record term="V50" count="24"/>
    <record term="V70" count="4435"/>
    <record term="W116" count="10435"/>
    <record term="W123" count="485"/>
    <record term="W124" count="9439"/>
    <record term="W210" count="48"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The value that is found in *'count'* reflects in how many records within the current result the term appears.

Example 12 Get statistics of used query terms

In this case you want to fetch the 10 most common terms used when querying between 11.00 and 13.00 the 16th of June 2005. The terms should be presented in descending order; the most common query terms first. It is query words (not strings) that should be fetched. The special table containing query terms is named QueryTerms and the column containing the query terms is called Terms.

The order should be ascending and you want to start from a term that is contained in 2.000 (or the closest higher frequency) records.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="index">
  <open_session name="User1"/>
  <database name="Statistics QueryTerms"/>
  <table name="QueryTerms"/>
  <index field="Terms" type="14" order="desc"
              start_position="searchterms(20050616 11:00..20050616 13:00)"
              max_terms="10"/>
 </SoftboolXML_request>
</SoftboolXML_requests>

<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records field="Terms" total="10"/>
    <record term="TERROR" total="6007"/>
    <record term="WAR" count="5968"/>
    <record term="NASDAQ" count="3897"/>
    <record term="DOLLAR" count="3787"/>
    <record term="GOLD" count="2007"/>
    <record term="LONDON" count="2004"/>
    <record term="EU" count="986"/>
    <record term="BUSH" count="712"/>
    <record term="PRESIDENT" count="710"/>
    <record term="ISLAM" count="687"/>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

The value that is found in *’count’* reflects the number of times the term has been used when querying Boolware during the specified time interval.

Example 13   Get records in specified rank order on specified term

This query will give all articles containing Volvo or Ford. This time you want the three (3) articles to be fetched from the data source. The Columns you want information from are: *Article no*., *Publishing date* and *Text*. A maximum of 20 characters from each Column should be fetched. You want articles where Ford appears most frequently presented first.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query" queryhistory="1">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <query>FIND "Text":Volvo OR Ford</query>
  <response>
    <records from="1" count="3" rankterm="Text:Ford" maxchars=20>
      <field name="Article no."> </field>
      <field name="Publishing date"> </field>
      <field name="Text"> </field>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

After this query the number of articles is **7.354**.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records total="7354" from="1" to="3" rank="occ each term"/>
    <record score="7">
      <field name="Article no."> 8432 </field>
      <field name="Publishing date"> 20000912 </field>
      <field name="Text"> Article about Ford...</field>
    </record>
    <record score="5">
      <field name="Article no."> 736 </field>
      <field name="Publishing date"> 20000422 </field>
      <field name="Text">Article about Volvo where Ford is mentioned..</field>
    </record>
    <record score="2">
      <field name="Article no."> 12456 </field>
      <field name="Publishing date"> 20000714 </field>
      <field name="Text">Article about Saab where Ford is mentioned..</field>
    </record>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

This time you will get the requested articles first where the word Ford appears most frequently.

Example 14   Get records in specified rank order on weighted columns

This query will give all articles containing Volvo or Ford from the year 2003. This time you want the three (3) articles to be fetched from the data source. The Columns you want information from are: *Article no*., *Publishing date* and *Text*. A maximum of 20 characters from each Column should be fetched.

Words appearing in the Text column are more important than words from other columns so you will set a weight on this column (all other columns will have a weight of 1).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query" queryhistory="1">
  <open_session name="User1"/>
  <database name="Newspapers"/>
  <table name="Articles"/>
  <query>FIND (Text:Volvo OR Ford OR Saab) AND "Publishing date":2003*</query>
  <response>
    <records from="1" count="3" rank="weightedoccurrency" rankweights="text=5"
             maxchars=20>
      <field name="Article no."> </field>
      <field name="Publishing date"> </field>
      <field name="Text"> </field>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

After this query the number of articles is **3.547**.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_responses>
 <SoftboolXML_response type="query" error_code="0">
  <session>User1</session>
  <records total="7354" from="1" to="3" rank="weight occ"/>
    <record score="51">
      <field name="Article no."> 4718 </field>
      <field name="Publishing date"> 20030912 </field>
      <field name="Text"> Article about Ford...</field>
    </record>
    <record score="46">
      <field name="Article no."> 1622 </field>
      <field name="Publishing date"> 20030422 </field>
      <field name="Text"> Article about Volvo...</field>
    </record>
    <record score="21">
      <field name="Article no."> 453 </field>
      <field name="Publishing date"> 20030714 </field>
      <field name="Text"> Article about Saab...</field>
    </record>
  </records>
 </SoftboolXML_response>
</SoftboolXML_responses>
```

All occurrences of Ford, Volvo and Saab will be multiplied by 5 (column *Text*), while the date (2003 in column *Publishing date*) will be multiplied by 1.

Example 15   Search in two databases with two requests

The first query finds customers using an interval search on customer id from a customer database. Before fetching address information about the customers they are sorted on title and name.

The second query finds distributers from London in another database . The address information is sorted on name.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<SoftboolXML_requests>
  <SoftboolXML_request type="query">
    <open_session name="" queryhistory="1" autotrunc="1" />
    <database name="CustomerDb" />
    <table name="Export" />
    <query>FIND XML/customer/custNo:01410000..01410980 </query>
    <response queryhistory="0" type="" href="">
      <sort expression="XML/customer/title, XML/custom/custno" />
      <records from="1" count="100" maxchars="0">
        <field name="XML/customer/custName" /field>
        <field name="XML/customer/custAddress" /field>
        <field name="XML/customer/custZip" /field>
        <field name="XML/customer/custCity" /field>
        <field name="XML/customer/custPhone" /field>
      </records>
    </response>
  </SoftboolXML_request>

  <SoftboolXML_request type="query">
    <open_session name="" queryhistory="1" autotrunc="1" />
    <database name="DistrDb" />
    <table name="Import" />
    <query>FIND DistrCity:London </query>
    <response queryhistory="0" type="" href="">
      <sort expression="DistrName" />
      <records from="1" count="100" maxchars="0">
        <field name="DistrName" /field>
        <field name="DistrAddress" /field>
        <field name="DistrZip" /field>
      </records>
    </response>
  </SoftboolXML_request>

</SoftboolXML_requests>
```

Chapter 4 Flows

This chapter describes how to use the Boolware Flow functionality.
See also the help function in Boolware Manager.

Overview

Flow Queries are scripts that are executed in Boolware server and are started via XML/JSON requests i.e. via the XMLRequest API function or JSON-request via API-function execute. All XML/JSON calls to Boolware starts with the root element 'SoftboolXML_requests' or an array of 'SoftboolJSON_request' followed by elements to specify which database and table and what functionality to use - e.g. "query" to perform a search.

To be regarded as a Flow Query the XML/JSON request must contain the following two things:

1. The *query* element has a 'flow' attribute which names the flow and
2. within the *query* element there is one or more named fields

```
<query flow="match">
  <name>Johnson</name>
  <city></city>
</query>

"query": {
 "@flow": "match",
 "name": "Eriksson",
 "city": ""
}
```

The script normally is stored in a file in the Boolware database directory, but could also be specified directly in the call. The fields in in a flow query are regarded as input and a number of functions in Boolware could be used to for example normalize the text, search in different ways, make calculations and rank (set score on) retrieved records.

The flow scripts contain two different parts: one main part in XML and script functions that are called together with variables to in an easy way program sophisticated searches.

Example of a flow:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow>
  <search text="str" dbfield="TelNo" goal="1" methods="word"/>
  <if test="hitcount > 0">
    <exit type="found" />
  </if>
  <exit type="notfound" />
</flow>
```

The XML part is a flow - a logical search strategy - with one entry point and several possible exit points. The exit points are named and are used by the caller to decide the result of the query (which path the query took).

Within the flow variables are used when searching and normalizing and by performing tests (*if*, *while*, *for*) control what path the query should follow. The XML element *if* is for example used to examine a variable and depending on the outcome takes one or the other path.

The *search* element performs a query in Boolware and returns information on the number of found records in the system variable *hitcount*.

The XML elements *while* and *for* are used for loops; to repeat something several times.

The XML element *set* is used to set a value to a variable (could also be done using *call script*).

The XML element *call* and *return* are used to make call to script functions, procedures and other flows.

The XML elements *fetch*, *customrank*, *resultset*, *scoring* and *score* are used to build custom designed resultsets (*Custom List*).

Note that the XML element *customrank* is used to initialize the building of custom designed resultsets. This implies that the elements *resultset*, *scoring* and *score* must be within the element *customrank*.

This is described by the below simple flow "Matching":

```
<?xml version="1.0" encoding="iso-8859-1"?>
<flow>
  <!-- Check if any input at all -->
  <if test="len(compname) <= 0">
    <print text="'No name specified in the flow query.' " type="comment"/>
    <exit type="no input"/>
  </if>

  <!-- Search for the specified name in the field CompanyName -->
  <search text="compname" dbfield="CompanyName" autotrunc="1" methods="word"/>

  <!-- Just exit if no hits -->
  <if test="hitcount <= 0">
    <print text="'No match on specified name ' . compname . '.' " type="comment"/>
    <exit type="no match"/>
  </if>

  <!-- Initialize Custom rank for custom designed resultsets -->
  <customrank>
    <!-- Clear earlier result -->
    <resultset action="clear"/>

    <!-- Add found records to Custom list -->
    <resultset action="add" score="100" limit="200"/>

    <!-- Do final ranking for found records -->
    <scoring minscore="92" maxrows="50" use_existing_score="1">
      <score var="compname" weight="95" norm="1" dbfield="CompanyName"/>
    </scoring>
  </customrank>
</flow>
```

Tests on terms specified in the flow query and queries **could** be outside the element *customrank*, while the elements *resultset*, *scoring* and *score* **must** be within the element *customrank*. An exception for the tag *resultset* with the attribute *clear, action=""* or *action="sort"*, which could be outside the element *customrank*.

The XML elements *log* and *print* are used for writing text, variables, results etc.

Flow Queries

Boolware can handle complex queries that include application logic. These are called "Flow Queries", since they start at a point, and flows its way down through the defined search logic (like a flow chart) until producing a desired result. An application can - in a single call - include several query fields as well as logic how the fields should be processed in the query. These rules can for example describe what to do if a search doesn't generate any results, or if it generates "too many" or "too few" hits.

By using the XML elements: *customrank*, *resultset* (described below) you could rank the retrieved records in another order that you could using the standard Boolware sort and rank functions.

Queries are passed as XML to Boolware. Three parts can be identified:

1. The query (query)
2. The flow description (flow)
3. The response (response)

The XML is such that the query semantics are indisputable. The flow description is also coded as XML.

This is an example, where an application sends an address record for retrieval of similar candidates.

```
<?xml version="1.0" encoding="UTF-8"?>
<SoftboolXML_requests>
 <SoftboolXML_request type="query">
  <open_session name="user"/>
  <database name="ATLAS"/>
  <table name="Companies"/>

  <query flow="match"
    <name>JOHN PARTRIDGE</name>
    <address>10 UPING STREET</address>
    <zip>WE 1212</zip>
    <city>STRATFORD ON-AVON</city>
  </query>

  <response type="" href="" queryhistory="0">
    <records from="1" count="20">
     <field name="*"/>
    </records>
  </response>
 </SoftboolXML_request>
</SoftboolXML_requests>
```

Note that the query consists of an actual record with four fields. The field names are used as XML element names, and there is a reference to a named "flow": match.

The flow is stored as an XML document at the server, in the current database directory. The flow below is an example of matching logic.

First you attempt an exact match using all four fields (search field='*'). If this gets exactly one hit, a <response> is sent back immediately. Otherwise the flow continues by attempting to find the name using gradually decreased precision (first near, then word and finally sound), until the goal of 10 hits is achieved.

If no records were found, a note about this is made in the log. Otherwise the selection continues by querying the other fields one at a time, using AND logic. The result is kept if it doesn't go down to zero, no hits. (zeroback="1")

In the following example the names of the search fields are the same as the corresponding column names in the table. In other words the table contains the columns: *name*, *address*, *zip* and *city*.

The flow ’**match**’ (flow file: **match.flow.xml**):

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```
<flow>
  <!-- Try full match first -->
  <search field="*" goal="1" methods="word"/>
  <if test="hitcount == 1">
    <exit type="found"/>
  </if>

  <!-- Must find at least 1, using company name (stop when 10 found) -->
  <search field="name" goal="10" methods="near,word,stem,sound"/>
  <if test="hitcount == 0">
    <log type="error" msg="not found"/>
    <exit type="error"/>
  </if>

  <!-- Include fields that generate hits, ignore other fields -->
  <search op="and" zeroback="1" field="address" goal="10"
methods="word,sound"/>
  <search op="and" zeroback="1" field="zip" goal="10" methods="string,word"/>
  <search op="and" zeroback="1" field="city" goal="10" methods="string,word"/>
  <log type="choice" msg="alternatives"/>
  <exit type="choice"/>
</flow>
```

Another query that calls the flow named ‘**match**’ (the **'match.flow.xml'** flow file):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
<SoftboolXML_request type="query">
<open_session name="user"/>
<database name="db"/>
<table name="table"/>

<query flow="match">
  <name>BLUE LADY</name>
  <address>12 rue de rien</address>
  <zip>12345</zip>
  <city>ANTIBES</city>
</query>

<response queryhistory="0">
<records from="1" count="20">
  <field name="*"/>
</records>
</response>
</SoftboolXML_request>
</SoftboolXML_requests>
```

Here is an example of an inline flow (passed as part of the query).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SoftboolXML_requests>
<SoftboolXML_request type="query">
<open_session name="user"/>
<database name="Atlas"/>
<table name="Companies"/>

<query flow="test">
  <name>JOHN PARTRIDGE</name>
  <address>10 UPING STREET</address>
  <zip>WE 1212</zip>
  <city>STRATFORD ON-AVON</city>
</query>

<!-- <response queryhistory="1"> -->
<response raw="1" quotes="0" fieldsep=";" rowsep="\r\n" queryhistory="1">
```

```
  <records from="1" count="20">
    <field name="name"/>
    <field name="street"/>
    <field name="zip"/>
    <field name="city"/>
  </records>
</response>

<flow>
  <!-- Company name given -->
  <if test="len(name)== 0">
    <log type="Warning" msg="Company name is empty"/>
    <exit type=" Company name is empty "/>
  </if>

  <!-- A unique match -->
  <search field="name" methods="word"/>
  <if test="hitcount == 1">
    <exit type="Single match found"/>
  </if>

  <!-- No hits. If so, try phonetic search -->
  <if test="hitcount == 0">
    <search field="name" methods="sound"/>

    <if test="hitcount == 0">
      <log type="Warning" msg="No phonetic hits, trying again with two words
                               removed from the right"/>

      <!-- Drop the two rightmost words -->
      <call script="dropterm(name, right, 2)"/>

      <!-- ....and try new phonetic search -->
      <search field="name" methods="sound"/>

      <if test="hitcount == 0">
        <exit type="Still no hits.."/>
      </if>
      <log type="Info" msg="Matches on name, continue with street"/>
    </if>
  </if>

  <!-- AND address..... -->
  <search op="and" field="street" methods="word"/>
  <if test="hitcount" op="eq" value="0">
    <search cmd="back"/>

    <!-- Increase street nos. by 4 both ways and retry... -->
    <call script="numrange(street, 4)"/>

    <!-- and retry -->
    <search op="and" field="street" methods="word"/>

    <if test="hitcount == 0">
      <exit type="No hits after street no range"/>
    </if>
   <log type="Info" msg="Hits in both name and address, press on"/>
  </if>

  <!-- ...continue with zip and city as long as matches remain -->
  <search op="and" zeroback="1" field="zip" methods="string,word"/>
  <search op="and" zeroback="1" field="city" methods="string,word"/>
  <log type="Info" msg="All fine, list of found records"/>
  <exit type="OK"/>
</flow>

</SoftboolXML_request>
</SoftboolXML_requests>
```

Variables

The fields in a flow query are called variables. These variables could be "read" from the flow, but should not be manipulated. If you want to "change" these variables you should move them to own defined variables. A variable name is case sensitive and must not have name such as a function name e.g. a variable name cannot be ‘string’ etc.

A variable has a name and its value could be changed. Own variables could be created and modified in a flow using the function *set* or *call script*.

Examples:

Set the variable *name1* to the input parameter *CompanyName*:
```
<set var="name2" value="CompanyName"/>
```

The following syntax may also be used instead of the above:
```
<call script="name2 = CompanyName"/>
```

Set the variable *name2* to the string constant Johnson:
```
<set var="name2" string="Johnson"/>
```

The same as above but uses the attribute "*value*" instead of attribute "*string*".
Note: In this case we must enclose the string constant in single quotes:
```
<set var="name2" value="'Johnson'"/>
```

The following syntax may also be used to set the variable *name2* to the string constant:
```
<call script="name2 = 'Johnson' "/>
```

Some variables are already defined (reserved words) for the Flows. That means that they already exist and are maintained by Boolware; thus do not use own variables with these names:

bwdocno	– document number of the last fetched tuple
bwhostip	– ip-address of the Boolware server running the current flow
bwhostname	– computer name of the Boolware server running the current flow
bwsessionip	– ip-address of the calling session running the current flow
bwsessionname	– session name of the calling session running the current flow
chewpos	– position in the input string in a CHEW function where a substring was found and was cut out, this is an array of positions if *multichew* The chewpos is 1-based and value 0 indicate that nothing have been cut out from the string
currhitcount	– number of records in the current search result for the currently searched table
currintermediateresult	– sub result for the current query before the command is performed from the currently searched table
errmess	– the latest error message from Boolware when *retcode* wasn't 0
errmessarray	– all error messages during the execution of a flow
executeresponse	- message buffer containing output from the flow element *execute*
hitcount	– number of records in the current search result for the table specified – in the current XML request
intermediateresult	– sub result for the current query before the command is performed
paramcount	– number of parameters sent to a proc
retcode	– the return code from the last called proc or call to Boolware
row	– the currently read record
rowcount	– number of records in a custom (customrank)
termzerobacked	– true if any term in the query caused a zero result and the term was ignored in the search
termzerobackedarray	– contains the terms that didn’t give any result in the current query
trigterm	– if a term is found in the function WASH in the *triggerwords*, the base synonym of this term will be saved in this global system variable
warnmess	– the latest warning message from Boolware when the retcode was > 0

warnmessarray	– all warning messages during the execution of a flow
xmlrequestdatabase	– name of the given database in the XML request
xmlrequestflowname	– name of the main flow in the XML request
xmlrequesttable	– name of the given table in the XML request
zerobacked	– true if the query has been backed caused by a zero result and zero back

Fetch rows

In a flow you can fetch rows from a result set with the XML element "*fetch*". The predefined variable "*row*" will contain the row that was fetched.

You could specify if you want to fetch a row from the current search result or from the current Custom List (see details in description of the function Fetch).

Example: Fetch the content from all columns from the first retrieved record:

<fetch row="1" cols="*"/>

To retrieve column data from a row do you use the following syntax:

```
row[columnName]
row[columnName1]
row[columnName2]...
```

The score for a row can be retrieved using the special column name '@score.'
Use '.' to concatenate column data.

Example:

```
<print text="row['FName'] . ' ' . row['LName'] . ' Score:' . row['@score']" />
```

You could also get records from another table within the current database:

```
<fetch table="table name" row="1" cols="*"/>
```

Strings, numeric and arrays

The variables could be of type string, numeric or array. Type numeric should be used on variables that is part of a calculation; divide, multiply etc. Strings are - in principal - all other texts such as: Name, Address, City etc.

An array is a numbered list of values. For example the string "Mercedes-Benz SL500" could be split into "words" and be put in an array-variable (see section describing split for more information):

```
arry = Split('Mercedes-Benz SL500', '-,; ')
```

The result will be an array containing four "words":

```
arry = array (
0 => Mercedes
1 => Benz
2 => SL
3 => 500
)
```

The variable arry contains four "words"; arry[0] to arry[3]. You could use the function length to test the number of elements in the current array; in this example it is 4.

Procedures and calls to other flows

Within a flow you could define procedures. A procedure is a sub-routine that could be called with different parameters.

```
<proc name="WashSplit">
  <call script="$P1 = wash($P1, '@namewash.txt')"/>
  <call script="words = split($P1, '/.,-\b')"/>
</proc>
```

In the example above a callable procedure with the name "WashSplit" is defined. It could be called anywhere in the flow. In the following example it is called with company name as a parameter:

```
<call proc="WashSplit" P1="company_name"/>
```

You could also call another flow. When the called flow terminates the control is returned to the calling flow.

```
<call flow="second"/>
```

Macros and parameters

Parameters to procedures are sent as "P1", "P2" and so on.

As all variables are global - except those which have a leading under score (_) - it is in most cases more efficient to use them directly instead of using parameters.

To be able to refer to the value in the parameter macro expansion is used. By preceding the variable with a dollar sign ($) you will get the value of the variable.

This strategy is also used in the search tag when you want to perform a search via the attribute cmd.

Example:
```
<search cmd="find street:$street and zip:$zip"/>
```

If you "escape code" the dollar sign (using a backslash \$) Boolware will interpret it as a dollar sign and not as a macro.

Example:
```
<search cmd='find "\$freetext":cars'/>
```

In this case Boolware will treat $freetext as a column name when searching and **not** as a macro.

Example:
```
<call script="arr = Match('Make \$50,000.00 fast with this no risk system',
'\\\$\\d{1,3}(,\\d{3})*(\\.\\d{2})?\\b'"/>
```

In this case '... \$50,000.00 …' the '$'-sign will not be interpreted as a macro and the '$'-sign in the regex-expression will end up with the escape sign in the regex-engine.

String compare

To be able to tell the similarity between two strings: "Fred E. Anderson" and "Freddy Anderson" a compare function is used internally which gives the likeness in number of errors.

The function - Compare - is adapted to be tolerant and is available from a flow.

By tolerant means that it accepts spelling errors and different spellings.

Normalizing

Before a compare takes place it is very important to normalize the strings; to make all characters upper case, remove diacritical marks etc.

In Boolware there are a number of functions for normalizing of strings which could be used. For example could phonetic coding be used causing words sounding alike will be treated as alike even if the spelling differs.

You could also take advantage of powerful search and replace functions based on the "RegEx" standard. This makes it possible to make own functions for "washing" and normalizing the data before the compare takes place.

See separate documentation on RegEx for more information.

Weighting

As some fields are more important than other as identification you could set different weights on different columns. For example is social security number and name a stronger identification than city or address as you move more often than changing your name and social security number.

The weight is specified as percent where 100 means "full weight" and 50 as "half weight". In the example: social security number, name, address and city a weighting could be: 100, 90, 50 and 30 respectively.

A weight of 50 makes a difference less significant compared to a weight of 100. The weighting is used within the elements score and scoring.

For more information and examples see section Scoring below.

Description of scoring

Scoring is a result of the comparison between the fields read from the database and what you have been searching on. The result of the comparison is controlled by different weights and parameters and is saved as a score.

The found records could later on be sorted descending on the current score and thus the most similar records will be presented first.

The XML element "scoring" is used to set up rules for how to rank found records. Scoring contains one or more "scores"; one per field.

Example:

```
<scoring maxrows="10" minscore="93" >
  <score var="name2" weight="90" norm="131" dbfield="Last+First"/>
  <score var="street2" weight="50" norm="129" dbfield="Street"/>
  <score var="zip2" weight="100" norm="1" dbfield="Zip"/>
  <score var="city" weight="40" norm="1" dbfield="City"/>
</scoring>
```

In the above example you could see that "scoring" will produce the 10 best hits. However, to be approved no score could be less than 93 (100 - no. of errors).

Each *score* element describes how to treat a field. The content of the field in the query form (mostly "massaged") (var) will be compared to the corresponding field(s) in the database (dbfield) in each found record.

The "massaged" field from the flow query saved in the variable name2 will be compared to the concatenated values from the fields Last and First in the database.

Different degrees of normalization could be performed before the two strings are compared. This normalization is controlled by the attributes: norm=, sortwords=, noice=, namefile=, lth=, leftmost=, synfile=, washchars= and condignore=.

norm	See the string function Normalize().
sortwords	1 = sort the words, 2 = sort the characters, 3 = test all combinations between the words and choose the one that gives the best similarity. 1 should be used in this case Johnson Ben and Ben Johnson. 2 could be used when dealing with words that sometimes are written together and sometimes are written apart: e.g. Ann Marie Johnson, Ann-Marie Johnson, Annmarie Johnson.
noise	Is a comma separated list with words that should be ignored before the comparison takes place. For example "Rue des Herroniers, Rue de la Heronnerie"; noice="des, de, la" will ignore these words when the comparison takes place.
namefile	Could be used to separate common male and female names. By writing a file containing all common female name, one name on each line, you could use this to distinguish between men and women.
lth	Only compare the x first characters.
leftmost	Higher score the earlier in the text the word is found.
synfile	The file that contains synonyms used by *Leftmost*. When using *leftmost* you could specify synonyms for special words. For example you could specify the following synonym: ltd(limited, inc, incorporated, corp). When testing in *leftmost* all synonyms will be used in the comparison.
washchars	Is used to "wash" certain characters from a text. Could be used to get rid of periods (.) commas (,) etc.
condIgnore	A very special attribute. Is used to determine if a compare should be skipped conditionally depending on if the previous field matched the first or second reference field.

See section "score" below to get more information.

Custom scoring

Boolware offers complete flexibility when it comes to ranking, as it allows you to program a custom module in C/C++ and use it instead of Boolware to compare tuples.

Boolware's built-in scoring is based on the *score*-element, used inside scoring, e.g:

```
<scoring maxrows="10" minscore="73">
  <score var="fname2" weight="50" norm="3" dbfield="fnamn"/>
  <score var="lname2" weight="50" norm="3" dbfield="enamn"/>
  <score var="pnr2" weight="100" dbfield="personnr" lth="8"/>
  <score var="street2score" weight="50" norm="259" dbfield="street"/>
  <score var="cityscore" weight="40" maxwords="4" dbfield="postort"/>
</scoring>
```

Custom scoring is activated by naming the custom DLL in the custom attribute of the scoring element. When Boolware sees this it'll send all comparison tuples to this external DLL. You can pass your own parameters to the DLL, as everything that stands within the scoring tag is sent to the external DLL:

```
<!-- Scoring will be performed by a plugin -->
<scoring maxrows="$maxRows" custom="xm" minscore="$lowScore" use_existing_score="1">
  <param>
  <search_group id="Name">
  <search_field id="Company_Name">
  <match_rule id="Same words and same order" reduction="1" weight="40"/>
  <match_rule id="Same words but different order" reduction="2" weight="35"/>
  <match_rule id="Fewer search words but all match" reduction="3" weight="20"/>
  </search_field>
  </search_group>
  </param>
</scoring>
```

The external module shall be programmed as a DLL on Windows, and as an .so file on Linux. It should be placed in the Boolware program directory and must have certain entry points. Further, the name must be on the format e02.modulename.dll or .so, depending on platform. In Chapter 10 "Plugins" description of how to write and registrate the external module is found.

Reduction of score when searching and manual scoring

Automatic scoring is often used by help of the scoring and score tags. This is a way to get an overview to collect variables like weight and retrieved fields in one place.

In some cases it is preferable to make an own scoring controlled by the fields searched on and the score used when the records were saved by *resultset add*. The XML element *search* could in this case be used to reduce the score if the for example searched data was found in the "wrong" field. If for example the name of the company was not found in CompanyName but in an extra information field it should perhaps result in a lower score - reduction of score.

The value to be reduced from the score should be specified in the *dbfield* within parentheses:

Example:

```
<search field="name" dbfield="company_name,extra(7)"/>
<resultset method="add" score="95"/>
```

If name is found in company_name no reduction occurs, but if it will be found in the field extra a reduction of 0.7 will be made and the final score will be 88 (95 - 7).

Summary

- Queries are coded as XML, using field names as XML element names.
- Flows are coded as XML, at the server with the ".flow.xml" extension.
- Flows can be edited in Manager.

Flow elements - reference

Below all available flow elements (tags) will be described.

break

```
<break [id="name"]/>
```

Is used together with *for* and *while* to terminate a loop before the specified condition is met.

If nested loops are use you could name the different loops and use this name to *break* the proper loop.

See examples below in *for* and *while*.

call flow

```
<call flow="name [P1=(value) P2=(value) ... ]/>
```

Calls another flow with a given name and passes parameters if any.

The called flow could use the variable "paramcount" to get the number of parameters.
Each parameter is referred to as "P<number>"; P1 for the first parameter.

If the called flow calls exit both flows will terminate else the control is returned to the calling flow.

The flow could return a value using the element *return*.

Example: call another flow "second.flow.xml" if "zip" is empty.

```
<if test="length(trim(zip))" op="eq" value="0">
  <call flow="second"/>
</if>
```

call proc

```
<call proc=(name) [P1=(value) P2=(value) ...]/>
```

Calls a defined procedure and passes parameters (if any).

The called procedure could use the variable "paramcount" to get the number of parameters.

Each parameter is referred to as "P<number>"; P1 for the first parameter.

The procedure may return a value using the *return*-element.

Another way to communicate with other procedures is to use global variables; see element set.

```
<proc name="log">
  <for loop="j" from="1" to="hitcount">
    <fetch cols="$P1"/>
    <log type="data" script="row[$P1]"/>
  </for>
  <return value="1"/>
</proc>
```

```
<call proc="log" P1="zip"/>
```

call script

```
<call script="name"/>
```

Call another function. See section Functions and variables for approved values.

In many cases this XML element is synonym with <set ..../>.

```
<call script="xxx = 'yyyyyyyyyyy' . 'zzzzzzzzzzzzz' . 'wwwwwwwwwwwwww'"/>
```

Example: call another function to erase a word from a search field.

```
<call script="dropterm(name,right,1)"/>
```

Example: assign a new value to an array element.

```
<call script="names[2] = 'Joe'"/>
```

continue

```
<continue [id="name"]/>
```

Is used together with *for* and *while* to continue the iteration in the loop without going to the end of the loop.

If nested loops are use you could name the different loops and use this name to do *continue* in the proper loop.

The command <next/> is a synonym to <continue/> see below.

See examples below in *for* and *while*.

customrank

```
<customrank [duplicates='0'] [resetresult='1'/>
```

Tells that you want to order (rank) the records in a special (own) way. The attribute duplicates tells whether duplicates should be included, the attribute 'resetresult' tells whether the previous custom result should be cleared or not.

You could save the ranked records in a special list using the *resultset*-element.

After that you could compare each record with the search string and get a score, which tells the likeness (100 = identical). By help of the element scoring you could reset the score of the retrieved records.

How the comparison should be done is determined in the element score.

When all records have got a new score they will be sorted (descending) on this value before presentation.

The newly ranked resultset will be activated when the XML element *customrank* will be closed.

else

```
<else/>
```

Could only be specified together with 'if' and take care of the "alternative" path.

```
<!-- Searches for 'name' and checks for 10 and 50 found records -->
<search field="name" methods="near"/>
<if test="hitcount" op="gt" value="10">
  <log type="Info" msg="More than 10 records found"/>
<else/>
  <if test="hitcount" op="eq" value="0">
    <log type="Info" msg="No records found"/>
  </if>
  <log type="Info" msg="Records found; less than 10"/>
</if>
```

elseif

```
<elseif test="(variable) </<=/==/>/>=/!= (variable)"/>
```

Could only be specified together with 'if'.

```
<!-- Searches for 'name' and checks for 10 and 50 found records -->
<search field="name" methods="near"/>
<if test="hitcount" op="gt" value="10">
  <log type="Info" msg="More than 10 records found"/>
<elseif test="hitcount" op="eq" value="0"/>
  <log type="Info" msg="No records found"/>
<else/>
  <log type="Info" msg="Records found; less than 10"/>
</if>
```

execute

```
<execute cmd=(string)/>
```

Perform an execute command and store the result text buffer in the flow system variable *executeresponse*. Note that not all execute commands will produce a textbuffer.

```
<!-- Perform an orsearchex and fetch the first 10 terms that gave zero result
-->
<flow>
 <search cmd="FIND companyname:orsearchex(ab, ltd gmbh) "/>
 <execute cmd="fetchnotfound table=CompanyTable from=1 count=10"/>
</flow>
```

Examine the system flow variable *executeresponse* to see the outcome of the execute command.

exit

```
<exit type=(string)/>
```

Terminates the flow. The value "type" is returned to the calling routine as an attribute in the response so that the caller knows which exit point has been used.

The application could also get the name of the exit point by the Execute API: Execute("getexitpoint").

fetch

```
<fetch table="tablename" row="num" cols=(col/*) maxrows="num" maxchars="num"
result="1/0" fetch_customlist="1/0" sort="" randomfetch="0/1"/>
```

Fetches a record from the current result or from the Custom List. The first record is 1. If the attribute "row" is omitted the next record will be fetched.

If the rows should be fetched from the current search result or from the Custom List depends on in what status you are (inside or outside the <customrank/>).

By using the attribute "*fetch_customlist*" you tell explicitly that you want to fetch rows from the Custom List (if there exists such a list).

If a table name - other than the main table - is specified the rows will be fetched from the current result of this table.

If rows have been saved in the Custom List and you have left the element <customrank/> the rows will be fetch from the Custom List.

In all other cases the rows will be fetched from the current result in the main table.

The columns to be fetched are controlled by the attribute "cols", can be comma separated. The content of a column is read by using the syntax: "row[column name]". An asterisk (*) instead of column name means all columns.

Only columns that have been specified in the *fetch* could be retrieved using the "row[column name]".

In "*maxrows*" you specify the maximum number of rows to fetch. At the first call to *fetch* a certain number of rows are "cached" in memory to avoid reading at each *fetch*. This attribute is only used to optimize the "cache"; if you just want to fetch 5 rows only 5 rows will be "cached" at the first call.

The attribute "*maxchars*" tells the maximum number of characters to fetch from each column. If this attribute is zero or omitted all characters from all columns will be fetched.

The attribute "sort" might contain a sort expression to receive rows in another order than database source order.

The attribute "randomfetch" can bet set to "1" if random fetch order is required.

If you want to fetch the current result (number of found records) from a table that is not the main table the attribute "*result*" in combination with the attribute "*table*" should be used. The result will be saved in the global system variable *currhitcount*. The current result of the main table could always be found in *hitcount*.

A *fetch* without any attributes will read all columns from the next record.

When trying to fetch records outside the search result an error occurs and the flow will terminate.

```
<!-- Read next row and log column "zip" -->
<fetch cols="zip"/>
<log type="data" script="row[zip]"/>
```

flow

<flow>

Root node of the flow

flow_input

<flow_input>

This element can contain <parameter> elements that can be obtained by applications
The element <flow_input> must only occur once in a flow and must occur first in a flow, before possible <include> elements.

Example:

```
<flow>
  <flow_input>
    <parameter name="Name" type="data" defaultvalue="" description=""/>
    <parameter name="Address" type="data" defaultvalue="" description=""/>
   </flow_input>
</flow>
```

for

```
<for loop=(var) from=(expr) to=(expr) step=(expr) [id="name"]>
```

A for-loop is used to iterate a certain number of times. The number of times is controlled by the variable "*loop*".

The variable "*loop*" will be created if not present. It is regarded as an error to modify the variable "*loop*". If it is necessary the while-loop should be used instead.

By using the attribute "*id*" you could give the for-loop a name, which could be used by the commands <break/>, <next/> and <continue/>. This is useful when using nested for-loops.

If you want to leave a loop before the specified condition is met you could use the command <break/>. This command could also be used to leave a named (outer) loop, when you are using nested loops.

Another useful command is <next/> or <continue/>. This command is used when you want to continue with the next iteration without go to the end of the loop.

```
<!-- Checks passed parameters, exits if any is empty -->
<for loop="i" from="1" to="paramcount" step="1">
  <if test="length($P$i) == 0">
    <exit type="empty"/>
  </if>
</for>
```

An example of how to use the <break/> command.

```
<!-- Checks passed parameters, exits if any is empty -->
<proc name="LoopBreak">
  <call script="words = array('one', 'two', 'three', 'four')"/>

  <!-- Two for-loops and one while-loop demonstrates the break command -->
  <for loop="i" from="0" to="length(words) - 1" id="outer">
    <for loop="j" from="length(words) - 1" to="0" step="-1" id="inner">
      <set var="k" value="0"/>
      <while test="k < 10">
        <set var="k" value="k+1"/>
        <if test="j == 2">
          <break/>
        </if>
        <if test="j == 1">
          <break id="inner"/>
        </if>
        <if test="j == 0">
          <break id="outer"/>
        </if>
      </while>
    </for>
  </for>
</proc>
```

An example of how to use the <next/> and <continue/> commands.

```
<!-- Two for-loops and one while-loop demonstrates the next command-->
<proc name="LoopNext">
  <call script="words = array('one', 'two', 'three', 'four')"/>

  <for loop="i" from="0" to="length(words) - 1" id="outer">
    <for loop="j" from="length(words) - 1" to="0" step="-1" id="inner">
      <set var="k" value="0"/>
      <while test="k < 10">
        <set var="k" value="k+1"/>
        <if test="j == 2">
          <next/>
        </if>
        <if test="j == 1">
          <next id="inner"/>
        </if>
        <if test="j == 0">
          <continue id="outer"/>
        </if>
      </while>
    </for>
  </for>
```

if

```
<if test=(variable) op=lt/le/eq/ge/gt/ne value=(string)>
```

An other - more straight forward way - to specify an if-statement is:

```
<if test=(variable1 </<=/==/>=/>/!= variable2)>
```

Test a variable and follows one of two paths depending on the result of the test. Approved variables - except own defined variables - to test on are the global system variables.

The variable "*hitcount*" is the total result after the specified command has been performed while "*intermediateresult*" is the sub result of the current query.

Example of test of an interval:

```
<if test="len(name) > 0">
  <!-- Searches for 'name' and checks for between 10 and 50 found records -->
  <search field="name" methods="near"/>
  <!-- Test an interval -->
  <if test="(hitcount >= 10) AND (hitcount <= 50)">
    <!-- .. perform something .. -->
  </if>
</if>
```

If the query has been performed in another table than the one specified in the XML request (the main table), the result will be stored in "currhitcount"; in "currhitcount" you could always found the result from the last searched table.

```
<if test="len(zipcode) != 0">
  <!-- Searches for 'zipcode' in table Table2 -->
  <search cmd="Tables(Table2) Find zip:$zipcode" autotrunc="0"/>
  <!-- If the result is between 10 - 50 do something -->
  <if test="(currhitcount >= 10) AND (currhitcount <= 50)">
    <!-- .. perform something .. -->
  </if>
</if>
```

include

```
<include file=(string)/>
```

Include must be defined before the main flow begins.

Include another flow. Useful for common procedures and variables.

The attribute "file" must not include the directory path of the file name. Boolware will supply the path automatically.

```
<!-- Include shared procedures and declarations -->
<include file="global.flow.xml"/>
```

log

```
<log type=(string) msg=(string) script=(expression) vars=(string)
fields=(string)/>
```

Creates a line in a logfile. Fields are separated by semicolon.

The attribute "*type*" is always written first.

If "*script*" is used the result of "*script*" will be written otherwise the "*msg*" will be written.

If "*vars*" or "*fields*" are given the values will be written.

"*fields*" refers to original query fields in the "query" element.
An explicit field could be given or "*" for all given fields.

"*vars*" refers to variables and changed field values. i.e. after a call to "dropterm" on a field the new value can be inspected by using "*vars*".

The logfile is written in the directory specified in Boolware. The name of the logfile is same as the name of the flow with the extension ".log". If the name of the flow is **test** the name of the logfile will be: "dbname.flow.**test**20080120.log".

An example of a line in the logfile:

```
choice;msg;name=higgins;street=;zip=16866;city=lipton
```

next

```
<next [id="name"]/>
```

Is used together with for and while to continue the iteration in the loop without going to the end of the loop.

The command <continue/> is a synonym to <next/> see above.

If nested loops are use you could name the different loops and use this name to do *next* in the proper loop.

See examples below in *for* and *while*.

output

```
<output print="1"/>
```

The attribute *print=0* could be used to block all *print* in a flow.

Example:

```
<output print="0" />
<print text="'A string...'" />
```

parameter

```
<parameter name="paramname" [type="" defaultvalue="" description=""]/>
```

This element must contain the attribute name with a proper value, other attributes are optional.
The element must reside within a <flow_input> element.
These elements can be obtained by an application to get information about a specific flow via *flowinfoparameters* command.

print

```
<print text=(script) type="elementname" stricttype="0/1"/>
```

The *print*-element will write text to the print buffer. The print buffer is a part of the XML response that can be used as an informative text and is enclosed by the tag `<print>` in the response.

The attribute *stricttype* is default "0", which means that if type contains a valid element name, e.g. "*comment*", an element with the name `<comment>`, will be created and will also contain the printed output. If *stricttype* is set to "1" the printing will not be printed in the `<print>` element only in the given *type* element e.g `<comment>`.

Example:

```
<proc name="PrintRecord">
  <print text="row['Company_id']" />
  <print text="row['First_Name'] . ' ' . row['Last_Name'] . ' - Phone:' .
               row['Phone'] . ' - Score:' . row['@score']" />
  <if test="len(row['Member_Last_Name'])" op="gt" value="0" >
    <print text="row['Member_First_Name'] . ' ' .
                 row['Member_Last_Name']" />
  </if>
  <print text="row['Street_Name'] . ' ' .
               row['House_No_or_Entrance_No']" />
  <print text="row['Zip'] . ' ' . row['Postal_Name']" />
  <print text="'-----'" />
</proc>
```

proc

```
<proc name=(name)>
```

Proc must be defined before the main flow begins.

Declaration of a procedure. A procedure is a callable bunch of instruction that can return a value to the caller.

The procedure can use the global system variable "paramcount" to determine the number of input parameters. Each parameter is referred to with the syntax P<number>, e.g. P1 for the first input parameter.

The procedure can return a value through the element return. Values can also be passed through global variables.

Example:

```
<proc name="log">
  <for loop="j" from="1" to="hitcount">
    <fetch cols="zip"/>
    <log type="data" script="row[zip]"/>
  </for>
  <return value="1" />
</proc>
```

The caller could test the return value in the variable *retcode*.

resultset

```
<resultset action=(string) rowstodelete=(value) limit=(value) score=(value)
sort=(expression) sort_customlist=(value) subtractscore=(value) hitno=(value)
rankmode=(value) randomfetch=(value) duplicates=(value)
use_existing_score=(value) customrankexit=(name) customrankselection=(name)
customranksearchcriteria=(arg) customrankautotrunc=(value)
customranklimit=(value) />
```

The resultset is used to produce a ranked searchresult.

The following attributes are applicable in the resultset entity:

action	"**add**" to add records to the Custom list "**clear**" to delete all records from the Custom list "**excludefrom**" remove records from the Custom list "**sort**" or "" to sort the Custom list or the current search result If *action* is omitted "add" will be used.
rowstodelete	only valid in conjunction with *action=excludefrom*. Tells how many records to delete from *hitno*; default is all records from *hitno*
limit	maximum amount of records to be stored. Default 100000
score	the "score" that will be applied on records. Default 100
sort	sort expression applied **before** records are stored in Custom list
sort_customlist	if sorting should be done on the Custom list or the current search result 0 - sort the current search result; default 0. 1 - sort the Custom list
subtractscore	1 - if score reduction occurred in the *search*-element. Default 1 0 - if no reduction should be done
hitno	is used in conjunction with *action*="excludefrom*"* to remove this record and all following records. Default 2
rankmode	re-rank record that should be added; the following rank modes are valid: 0 - No ranking; default 1 - Rank by occurrence 2 - Rank by frequency 6 - Rank by weighted occurrence 7 - Rank by weighted frequency 10 - Rank by string distance from previous fuzzy search
randomfetch	0 - if no random fetch should be done; default 0. 1 - if the records should be stored in random order rather than in the same order as they appear from the data source.
duplicates	0 – only one duplicate record with the highest score is saved 1 – all duplicate records are saved regardless of score value If omitted the duplicates specified in *customrank* will be used.

use_existing_score	0 - Do not use existing scores, instead the given value in the attribute *score* will be used, default. 1 - Use existing scores e.g. sort/rank will be made within existing score groups but position adjusted based on the given value in the attribute *score*
customrankexit	the name of the custom ranking plugin to use
customrankselection	the name of the ranking criteria to be used
customranksearchcriteria	the search arguments that was used for each ranking field. The search arguments for each ranking field are separated by a comma character. In order for the e04.customrank plug-in to be able match the search arguments with the correct ranking field, it is important that the search arguments for respective ranking fields are listed in the same order as they were specified with the configuration parameter "fieldnames", in the plug-in configuration. Example: In the e04.customrank configuration, we have specified the following ranking fields: fieldnames = company_name, address, city We then search for "McDonald's" in the field "company_name" and "Stockholm" in the field "city".Now we need to specify search argument as follows:

	customranksearchcriteria = "McDonald's,,Stockholm" Since we did not search in the "address" field, we do not have any search arguments for this field. But we must always include all ranking fields when we specify the "customranksearchcriteria", so we leave it blank but mark the field's presence with an additional comma character.
customrankautotrunc	0 - Rank without truncated search arguments, default. 1 - Rank with truncated search arguments
customranklimit	maximum number of items to be ranked
customrankgeodistance	here you can specify a centrum coordinate and a radius in meters. All hits within the specified circle will be ranked by how close they are to the centrum coordinate .
customrankuserdata	this attribute can be used to send user-defined data to a custom developed customrank module.

The sorting of records could be done in three ways:

1. When you add records to the Custom list (*action*="add") and a sort expression is specified in the attribute *sort* the records will be sorted in requested order **before** they are stored in the Custom list.
   The sort have the following precedence *randomfetch*, *customrankexit*, *sort* and *rankmode*.

2. When you exclude records from the Custom List (*action*="excludefrom") and a sort expression is specified in the attribute *sort*, sort the records in the Custom list **before** the records are removed from the Custom list.

3. If only sort should take place (*action*="sort") or (*action*="") and a sort expression is specified in the attribute *sort*. With the attribute *sort_customlist*="0/1" the sort expression is applied on either the current search result or the records in the Custom list.

return

```
<return value=(expr)/>
```

Terminates a called procedure or flow and returns a value to the caller. The returned values are numeric. If the attribute "value" is absent the return value will be 0.

Example:

```
<return value="-1"/>
```

score

```
<score var=(string) dbfield=(string) dbvalue=(string) condignore=(string)
namefile=(string) weight=(value) norm=(value) leftmost=(value)
synfile=(string) washchars=(string) spacechars=(string) noise=(string)
sortwords=(string) lth=(value) irregnames=(string) maxwords=(string)
wash=(string) samewords=(string) trywithout=(string)
samewords_phon_punish=(value) streetno_punish=(string) condvar=(string)
condflag=(value) honor_empty_data=(value)/>
```

The score is used to calculate the score value of records that are saved under the element "resultset".

By describing which columns that should be compared and then apply rules for the comparison a score is obtained for each record. If the score value exceeds the value specified in the attribute "minscore" in the element "scoring" the value is applied to the record; if not the record is removed.

The following attributes are used in the score element:

var	search field
dbfield	field to compare
dbvalue	value instead of content of *dbfield* to be compared
condignore	used to determine if a comparison should be skipped conditionally depending on previous field matched the first or secondly referenced field. Any value activates this function.
namefile	input file to specify certain typically female names to distinguish from similar male names to prevent the same "score".
weight	weight on the field; the higher importance the higher weight.
norm	specify what normalize method to use, see function "Normalize" below.
leftmost	higher score the earlier in the text the word is found. If >0 this function will be activated.
synfile	when *leftmost* activated you could specify synonyms. E.g. specify the following synonyms: Ltd(limited, corp, corporation). When using *leftmost* the specified synonyms will be used when performing the compare.
washchars	remove specified characters from the string before comparison take place; e.g. remove space characters in a zip code field
spacechars	as washchars, but replaces certain chars with space.
noise	list of words that should be removed before the comparison takes place; noise="de la le af von" in name fields
sortwords	before the comparison takes place; no sort (0), sort the words (1), sort letters (2) alternatively (3) which tests all combinations between the present words and selects the most similar.
lth	is used to compare the only first N characters.
irregnames	names a synonym file used for irregular spellings.
maxwords	max allowed words in field, if more field is considered containing garbage
wash	names a wash file for the database field
samewords	gives full match if a given number of words are found in both strings, i.e. all words doesn't have to match. You either specify a number or one of the special codes: * compare least number of words % least number of words, least number of chars ! as %, but no reduction for missing initials.
trywithout	Try samewords again, after removing specified chars.
samewords_phon_punish	reduction if samewords gives match on phonetic comparison. default 0.
streetno_punish	reduction if street numbers differ. default 0.
condvar	name of the field that is affected by *condflag*.
condflag .	specifies the flag to be set in normalize (norm) for the field specified in *condvar*.
honor_empty_data	1 = high score although data is missing in the current field (overrides value set in *scoring*). Default = 0.

Suppose that *condvar* is a "social security number" and *condflag* is set to 128. If "social security number" is specified (not empty in the flow query), 128 will be OR:ed to the value set in *norm* for this field. The meaning of the different norm-flags is described below under the section Normailze.

scoring

```
<scoring minscore=(value) maxrows=(value) honor_empty_data=(value)
use_existing_score=(value) custom=(string)/>
```

Is used to calculate the score on records saved by the resultset. The records are sorted automatically when the *scoring*-element is closed. Scoring use the flow function Normalize to normalize terms and the default Boolware character file is used.

To calculate the score the element score is used were rules can be specified as attributes.

The following attributes can be used in the "scoring" element:

minscore	the record must exceed this score to be approved. Default = 1
maxrows	maximum numbers of record to save. Default = 1.000.000.
honor_empty_data	1 = high score although data is missing in one or more fields. Default = 0.
use_existing_score	0 (default) is used to re-score all records and assume that 100 is the highest possible value, 1 use the score applied on the record.
custom	the name of an external module to be used instead of the built in function in Boolware.

Note! You cannot use conditional statements (*if*) within *scoring*-elements, but you can use macro expansion for the attributes. Another way is to make several sections within conditional statements:

```
<if test="pnrFound > 0">
  <scoring maxrows="$maxRows" minscore="$lowCandidateScore" honor_empty_data="0">
    <score var="pnrNo" weight="$pnrWeight" lth="8" dbfield="$pnrFields"/>
    <score var="personName" weight="$nameWeight" trywithout=" " sortwords="1"
           samewords="1" norm="3" spacechars="-" irregnames="@irregnames.txt" noise="DE
           LA VON VAN" dbfield="$nameFields"/>
    <score var="personZip" weight="$zipWeight" dbfield="$zipFields"/>
    <score var="personCity" weight="$cityWeight" norm="3" samewords="1" maxwords="4"
           wash="@citywash.txt" dbfield="$cityFields"/>
    <score var="personStreet1" weight="$streetWeight" norm="$streetNorm" trywithout=" "
           sortwords="1" samewords="2" spacechars=" " wash="@streetwash.txt"
           dbfield="$streetFields"/>
    <score var="personCo" weight="$coWeight" norm="259" trywithout=" " sortwords="1"
           samewords="2" spacechars=" " dbfield="$coFields"/>
  </scoring>
<else/>
  <scoring maxrows="$maxRows" minscore="$lowCandidateScore" honor_empty_data="0">
    <score var="personName" weight="$nameWeight" trywithout="-" sortwords="1"
           samewords="!" norm="3" spacechars="-" irregnames="@irregnames.txt" noise="DE
           LA VON VAN" dbfield="$nameFields"/>
    <score var="personZip" weight="$zipWeight" dbfield="$zipFields"/>
    <score var="personCity" weight="$cityWeight" norm="3" samewords="1" maxwords="4"
           wash="@citywash.txt" dbfield="$cityFields"/>
    <score var="personStreet1" weight="$streetWeight" norm="$streetNorm" trywithout=" "
           sortwords="1" samewords="2" spacechars=" " wash="@streetwash.txt"
           dbfield="$streetFields"/>
    <score var="personCo" weight="$coWeight" norm="259" trywithout=" " sortwords="1"
           samewords="2" spacechars=" " dbfield="$coFields"/>
  </scoring>
</if>
```

search

```
<search cmd=(text) op=(QL op) opterm=(QL op) zeroback=1/0 zerobackterm=1/0
querylim=1/0 querylimvalue=(int) field=(field/*) text=(variable)
dbfield=(field) goal=(int) methods=(string) func=(string) autotrunc=1/0
near=(param) fuzzy=(param) sim=(param) nolinkwords=1/0
maxqueryexecutiontime=(int)resultbitmap=""/>
```

Perform a query. Either a QL-query (via the syntax "*cmd*=") or a query on supplied value (via the syntax "*field*="). If "*cmd*" is used Boolware will ignore all other attributes in the search element except "*zeroback*", "*zerobackterm*", "*autotrunc*", "*opterm*", "*querylimvalue*" and "*resultbitmap*".

If the syntax field=, *dbfield*=, *text*=, *op*=, *goal*=, *methods*= is used you are limited to pure search commands (FIND, AND, OR, NOT och XOR). If you on the other hand build up your search command you can also use: *save*, *delete*, *setsearch*, *back* etc. to handle saved results/sets.

As the flow handles all queries within the same session, it is best to use the local Sysindex. The local Sysindex will be used when *saved Set* and *saved named Set*. *Saved Set* requires that the Setsearch is activated and it will be activates until deactivated. When Setsearch is activated all queries will be saved as *Sets* and will be given a name by Boolware (S1, S2, … Sn). It could be difficult to keep track of the automatically generated names when they should be used in a query and thus it is more convenient to use the *saved named Set*; see description in Manual "**Operations guide**", Chapter 11 "Interactive Query" section "Set Search". SysIndex (both the global and the local) is saved on disk which means writing and reading has to be done. To avoid these write/read as much as possible you could use the named *Scratch* to save results that should be used later on. Named scratch results are "saved" in memory but have the same functionality as *saved named Sets*.

Summary: use named Scratch results as the first option when a result should be saved. As second option use *saved named Set*. You should avoid *saved result*.

cmd contains the actual query command, see " **Operations guide**", Chapter 11 "Interactive Query".

Example:
Find one American president.
<search cmd="FIND president: John Baines Kennedy" />

NOTE!
The following attributes can be used together with *cmd*:
autotrunc, *maxqueryexecutiontime*, *opterm*, *resultbitmap*, *zeroback* and *zerobackterm*.

op contains a valid QL-commands (FIND, AND, OR, NOT, XOR). Default is "FIND".

opterm is the operator between terms in a "search". If for instance the search string is "Ingvar Kamprad Elmtaryd Agunnaryd", the "opterm" is the implicit operator between the terms (words). Can be AND, OR, NOT, XOR. Default value is: 'AND'.

zeroback only valid between operators AND and NOT and is used to avoid zero results. In cases of zero result (the entire command) the query will automatically be revoked to the previous result. Default value is "0" - no automatically revoke of zero result.

zerobackterm is similar to "*zeroback*" but is applied on each term in a search string (containing several words); it not useful when the "*opterm*"=OR.
Default is "0" - no automatic back. The search terms that causes zero result and "back" to the previous intermediate result are saved in the global system variable *termzerobackedarray*. In the global system variable *termzerobacked* you could see how many terms are saved. You have to divide the number by two as each zero backed term generates two entries in the *termzerobackedarray* (the current column name and the current term).

Example:

Find an american president; if a name is not found it should automatically be zero backed.

```
<search cmd="FIND presidents:John Baines Kennedy" zerobackterm="1"/>
```

No president has the specified name, but as *zerobackterm* was activated the name Baines which causes a zero result will be backed. Thus the result is 1: John Fizerald Kennedy. You could check if any name caused a zero result and if so which name(s) it was by checking the global system variables *termzerobacked* and *termzerobackedarray*.

```
<if test="termzerobacked > 0">
  <!-- Each backed term generates two elements in termzerobackedarray, -->
  <!-- column name and search term.                                    -->
  <set var="backedTerms" value="termzerobackedarray[1]"/>
  <for loop="i" from="3" to="length(termzerobackedarray)-1" step="2">
    <set var="backedTerms"
         value="backedTerms . ',' . termzerobackedarray[i]"/>
  </for>
  <print text="'Zero backed terms: ' . backedTerms"/>
</if>
```

querylim if set to 1 each result on a term should be compared with "*querylimvalue*". If no "*querylimvalue*" is given the global "querylimvalue" will be used (default 100). If number of hits for the actual term exceeds the limit value it will be revoked from the search, (similar to stop words). If "querylim" is set to 0 or is absent, no test will be done against any limit value. This option is meaningful only if the search string contains more than one word and the *opterm* is OR.

querylimvalue is a temporary value - for this search only - that will be used as a limit value if "*querylim*" is set to 1.

field is the name of the field in the search (which search field to handle). The name of the field should have a corresponding field in the table. The content of the specified field will be used in the query. As an alternative an asterisk can be used to select all fields. In that case all fields will be match against the table to find an "exact match", where all fields match the search criteria. To use this syntax the fields in the flow query (input) agree with the fields in the table. Moreover, you could not manipulate ("wash") the content of the input. A more flexible way is to use "*text*" for the search words and "*dbfield*" to specify the corresponding field(s) in the table. an even more flexible way is to use the *cmd*= syntax. In this case you build the search command exactly as you want it.

text could contain a variable "massaged" via the built in string functions (or a constant).

dbfield contains the name(s) of in column(s) that should be searched in the current table. The search arguments are probably saved in the "*text*" attribute. By specifying *table.column* the search will be directed to another table within the same database; a relate query will be generated. The result will then automatically be "transformed" to the main table.

Examples:
The database "Persons" contains two tables "table1" and "table2".

```
<?xml version="1.0" encoding="iso-8859-1"?>
<database name="Persons"/>
<table name="table1"/>
...
<name>Higgins</name>
...

<search field="name"/>
```

A FIND will be performed in the column "name" in the table "table1" and the search argument is Higgins.

```
<search field="table2.lastname" text="name"/>
```

In this case a search for Higgins will take place in the column "lastname" in table "table2" The result will be transformed into "table1".

The attributes "*goal*" and "*methods*" is always used together. The combination of these are a powerful way to express variants of different search methods ("*methods*") until a "*goal*" is reached. Specified search methods in "*methods*" will be performed from left to right until the goal is reached or all methods performed. Methods are written as comma separated values. Valid "*methods*" are: *string, word, syn, thes, stem, sound, fuzzy, case, near, stringasis, withinstring, wordasis , nearwordasis, nearsound, nearcase* and *sim*. If "*methods*" is omitted *word* will be performed. If "*goal*" is omitted 1 will be used.

func could contain a call to a script function. See section "Functions and variables" below.

autotrunc could be set to 1 to perform an automatic truncation of all search terms. If no truncation is desired *autotrunc* should be set to 0. If no value is set the current value will be used, which could be unpredictable because you are not sure what has been done before.

near holds the parameters for the *near* method; near="N,O". The parameters are N and O, where N is the max allowed gap between the specified search terms. When O is 1 the search terms must appear in the specified order; 0 means any order. See description in "Operations Guide" Chapter 11 "Interactive Query".

Example:
```
<search text="'john smith'" goal="20" methods="near" near="1,1"/>
```
means that john smith must appear as a phrase to be approved in the near search.

fuzzy holds parameters for the *fuzzy* method; fyzzy="D,L". The parameters are D and L where D is the maximum difference (in number of characters) to generate a hit. L is a limit for the maximal difference in spelling. See description in "Operations Guide" Chapter 11 "Interactive Query".

Example:
```
<search field="name" goal="20" methods="fuzzy" fuzzy="2,97"/>
```
means that all words that have a length that does not differ more than two from the specified search term and the number of misspellings is less than three will generate a hit.

sim holds the parameters for the *sim-method*; sim="T,O". The parameters are T and O where T is the threshold a value between 0.0 and 1.0 and the O is optimize and relevance factor, a value between 0.0 – 1.0.

Example:
```
<search field="name" goal="20" methods="sim" sim="0.2,0.1"/>
```

nolinkwords tells that "linkwords" should not be used in the current search when the value is set to 1. Default is 0 (use linkwords if active on field). "linkwords" are used by the *word* and *sound* methods. See description in "Operations Guide" Chapter 11 "Interactive Query".

In *maxqueryexecutiontime* you could specify a maximum execution time in milliseconds for the current command. When the specified time is reached the execution is terminated with an error code. NOTE the specified time is only valid for the current command, else the maximum execution time for a session set in the Boolware Manager will be used. If you used the *cmd*=(string) to specify the search command the *maxqueryexecutiontime* could be specified as an sub command. This sub command must immediately follow the specified command.

Example:
```
<search dbfield="name" text="'john smith'" maxqueryexecutiontime="20"/>
```
In this case the current search will be terminated if it takes more than 20 milliseconds.

```
<search cod="FIND name:jo* smith" maxqueryexecutiontime="50"/>
```
In this case the current search will be terminated if it takes more than 50 milliseconds.

maxqueryexecutiontime could also be specified as an sub command if *cmd*= is used:

```
<search cmd="FIND maxqueryexecutiontime(50) name:jo* smith"/>
```

resultbitmap can be set to the string "custom" if the operation AND or NOT should be performed directly on the result in the "Custom List" when using the *cmd*.

Some more examples:

Example:

```
<search cmd="FIND name:higgins"/>
```

Performs a query to search for "higgins" in the field 'name' in the current table.

```
<search cmd="FIND name:$name"/>
```

Performs a query to search for the contents in the search field 'name' in the field 'name' in the current table.

```
<search opterm="or" querylimvalue="50" cmd="OR type:querylim(sound($name))"/>
```

Performs a query and uses the sub command querylim to search in the field 'type' in the current table using the contents in the search field 'name'. All terms that appears in more than 50 records will be ignored when searching. The operator between the different terms is OR.

```
<search opterm="or" querylim="1" querylimvalue="50" field="name"
dbfield="type" methods="sound"/>
```

Performs a query and uses the sub command querylim to search in the field 'type' in the current table using the contents in the search field 'name'. All terms that appears in more than 50 records will be ignored when searching. The operator between the different terms is OR.

```
<search field="name"/>
```

Performs a search in the field 'name' in the current table and uses what is specified within the element <name>. The command is "find" and the method is "word" as nothing is specified.

```
<search op="and" field="city"/>
```

Performs AND city:<city> where "city" is fetched from the flow query.

```
<search op="and" field="city" zeroback="1"/>
```

Performs the same as above but twill automatically back if the number of found records is zero.

```
<search cmd="back"/>
```

Performs a BACK command; makes the previous result the current.

```
<search field="name" goal="20" methods="near,word,sound"/>
```

Will first perform a FIND name:near(<name>) (proximity) and then compares the result to "goal". If the result is less than "goal" it will continue with FIND name:<name> (normal word search operator AND between the terms). If the result still is less than "goal" it will continue with FIND name:sound(<name>) (phonetic search). The point is to gradually widen the result. The exact same could be expressed as follows:

```
<search field="name" methods="near"/>
<if test="hitcount < 20">
  <search field="name" methods="word"/>
  <if test="hitcount < 20">
    <search field="name" methods="sound"/>
  </if>
</if>
```

Perform a relate query by searching in table Bromma and then convert the result to the target table Company (specified in the element <table name=>):

```
<?xml version="1.0" encoding="iso-8859-1"?>
...
<database name="companies"/>
<table name="Company"/>

<flow>
  <!-- Must find at least 1, using company name -->
  <search field="Bromma.Companyname" goal="20" methods="word, sound"/>
  <if test="hitcount == 0">
    <log type="error" msg="not found"/>
    <exit type="error"/>
  </if>
</flow>
```

Uses the content in the query element "a" to search for "telno" in the database column. If "dbfield" is not specified Boolware assumes that the name of the column in the database is the same as in the query ("a" in this example).

```
<search field="a" dbfield="telno"/>
```

Perform a search in a table - that is part of the database - and is not specified in the XML request. Pick up the result from a column in that table and use the extracted words to perform a search in the table specified in the XML request. The database Persons contains two tables: Person_data and Address_data.

```
<?xml version="1.0" encoding="iso-8859-1"?>
...
<database name="Persons"/>
<table name="Address_data"/>


<flow>
<!-- Searches for 'name' in table Person_data -->
<if test="len(name) != 0">
  <!-- Searches for ‘name’ in table Person_data -->
  <search cmd="TABLES(Person_data) FIND last_name:$name" autotrunc="0"/>

  <!-- If the result is between 10-50 pick up zipcode in the found records -->
  <if test="(currhitcount > 10) AND (currhitcount < 50)">
    <!-- Pick up zipcodes from the found records -->
    <set var="i" value="1"/>
    <while test="i <= currhitcount">
      <fetch row="i" table="Person_data" cols="zipcode"/>
      <set var="zip" value="row['ZipCode']"/>
      <!-- Build an orString of extracted zipcodes -->
      <set var="orString" value="orString . ' ' . zip"/>
      <set var="i" value="i+1"/>
    </while>

    <!-- Search for found zipcodes in table Address_data -->
    <search cmd="FIND zipcode:orsearch($orString)"/>

    <!-- Check if any result and do something -->
    <if test="hitcount > 0">
      <-- Pick up corresponding Cities from Address_data -->
      <set var="i" value="1"/>
      <set var="cities" value="''"/>
      <while test="i <= hitcount">
        <fetch row="i" cols="City"/>
        <set var="cities" value="cities . ', ' . row['City']"/>
        <set var="i" value="i+1"/>
      </while>

      <-- All found cities in variable 'cities' -->
    </if>
  </if>
</if>
</flow>
```

set

```
<set var=(name) value=(expr) string=(string)/>
```

Declare a variable and set a value. If the variable does not exist it will be created.

The attribute *var* must exist and gives the name of the current variable.

If something is specified in the attribute *string* it will be moved to the variable specified in *var* exactly as is. No variables could be specified in *string*. If *string* is specified any value specified in the attribute *value* will be ignored.

Variables are normally global and could be used outside the procedure.

Local variables must start with a '_' in the name and disappear when the procedure exit.

Variable name could not start with a dollar sign '$' as it is reserved for the macro.

```
<!-- Create variable -->
<set var="counter" string="75"/>

<!-- Increment variable -->
<set var="counter" value="counter+1"/>
```

NOTE! The **set** syntax can not be used to assign a new value to an array element.
To assign a new value to an array element, use the `call script` syntax.

Example:

<call script="myArray[2] = 'newString'/>

while

```
<while test=(name) </<=/==/>/>=/!= (expr) [id="name"]>
```

Tests on a value and performs the loop if the value is true. A while-loop is very much alike a for-loop but is more general when it comes to comparison (see for).

By using the attribute id you could give the while-loop a name, which could be used by the commands <break/>, <next/> and <continue/>.

If you want to leave a loop before the specified condition is met you could use the command <break/>. This command could also be used to leave a named (outer) loop when you are using nested loops.

Another useful command is <next/> or <continue/>. This command is used when you want to continue with the next iteration without go to the end of the loop.

```
<!-- Checks passed parameters, exits if any is empty -->
<set var="i" value="1"/>
<while test="i" op="le" value="paramcount">
  <if test="length($P$i) == 0">
    <exit type="empty"/>
  </if>
  <set var="i" value="i+1"/>
</while>
```

An example of how to use the <break/> command.

```
<!-- Checks passed parameters, exits if any is empty -->
<proc name="LoopBreak">
  <call script="words = array('one', 'two', 'three', 'four')"/>

  <!-- Two for-loops and one while-loop demonstrates the break command-->
  <for loop="i" from="0" to="length(words) - 1" id="outer">
    <for loop="j" from="length(words) - 1" to="0" step="-1" id="inner">
      <set var="k" value="0"/>
      <while test="k < 10">
        <set var="k" value="k+1"/>
        <if test="j == 2">
          <break/>
        </if>
        <if test="j == 1">
          <break id="inner"/>
        </if>
        <if test="j == 0">
          <break id="outer"/>
        </if>
      </while>
    </for>
  </for>
</proc>
```

An example of how to use the <next/> and <continue/> commands.

```
<!-- Two for-loops and one while-loop demonstrates the next command-->
<proc name="LoopNext">
  <call script="words = array('one', 'two', 'three', 'four')" />

  <for loop="i" from="0" to="length(words) - 1" id="outer">
    <for loop="j" from="length(_words) - 1" to="0" step="-1" id="inner">
      <set var="k" value="0"/>
      <while test="k < 10">
        <set var="k" value="k+1"/>
        <if test="j == 2">
          <next/>
        </if>
        <if test="j == 1">
          <next id="inner"/>
        </if>
        <if test="j == 0">
          <continue id="outer"/>
        </if>
      </while>
    </for>
  </for>
```

Functions - overview

A great number of functions are available which makes it possible to take advantage of all indexing logic that is part of the Boolware system. There are for example functions for normalizing of spelling (phonetic coding) and access to synonyms and thesaurus.

Calculation

addition, subtraction, multiplication, division, shl, shr, modulus, and, or, not, xor

Mathematic functions
abs, cos, date, deg, pi, rad, sin, sqrt, tan, time

String functions

array, chew, compare, compress, delete, dropchar, dropterm, env, false, iif, insert, instr, lower (lcase), length (len), lookup, match, mid (substr), normalize, numeric, numrange, only, replace, reverse, sort, space, split, stem, string, syn, thes, tostring, trim, true, upper (ucase), ver, wash

Logical operators

and(&&), or(||).

Functions - reference

Below is an alphabetic list of all available functions.

Abs

`number = abs(number)`

Returns the absolute value of a number.

Example:

```
Abs(-2.3) => 2.3
Abs(-157) => 157
```

Array

`array(item1 [, item2..itemN])`

Creates and initializes an array with the specified elements.

Example:

```
names = array('Donald', 'Duck')

names [0]   Donald
names [1]   Duck
```

Chew

```
resultstring = Chew(inputstring, regex-strings, lookup-words, lookup-index, flags,
wordSep)
```

Identifies and extracts one or more sub-strings in a text, *'inputstring*', and stores the extracted result in '*resultstring*'*.* If '*lookup-words*' is specified and used, the extracted sub-strings from *'inputstring'* are replaced with the found base synonym and stored in the *'resultstring'*.

The parameter '*inputstring*' contains the string that should be examined.

The parameter '*regex-strings*' contains one or more regex strings to search for, separated by new line (Lf or CrLf). If starting with a '@' it will be regarded as a file name which contains regex expressions. If the sub-string is found in any of the given regex expressions it will be removed from *'inputstring'* and stored in the *'resultstring'* and the global variable "*chewpos*" will contain its start position in *'inputstring'.* The "*chewpos*" is 1-based and is 0 if nothing is removed from the '*inputstring*'.

The parameter *'lookup-words'* contains a list of "quick-search" terms; for example the 200 biggest cities in US to identify a City. This parameter can contain either a file name - started with a '@' or the argument *#left*. The argument *#left* means that you, in the field specified in the '*lookup-index*', should also search for the word to the left of the sub-string found in *'regex-strings' e.g. North bondstreet*.

The parameter '*lookup-index*' could contain a field name. If the string could not be identified by '*regex-strings*' nor by '*lookup-words*' Boolware will use the index for the specified field to see if any term in '*inputstring*' is part of that field (for instance address). If the argument *#left* has been specified in '*lookup-words*' and a term has been found in '*regex-strings*' a search on the word to the left of the found sub-string will take place in the corresponding Boolware index.

In the parameter '*flags*' you could specify the following values: *sound=n, alphanumericsplit, replacemode, multilookup, nocleanup, rightlookup, indexbeforelookupfile, onlystring* and *multichew*.

If *sound* is specified the specified *n* (see Normalize) tells which phonetic algorithm to be used when searching in the Boolware index.

If *alphanumericsplit* is specified the words will be split between alpha characters and digits.

If *replacemode* is specified it means that the found word will be replaced using the regex expression '*regex-strings*'.

If *multilookup* is specified search in '*lookup-words*' will continue until all occurrences has been found; the found base synonyms will be separated by comma (,) and are stored in the *'resultstring'*.

If *nocleanup* is specified it means that the input string – '*inputstring*' - will not be cleaned up from interpunctuations and digits before searching in the Boolware index specified in the parameter '*lookup-index*'.
If *nocleanup* is not specified, punctuations and digits will be replaced with space character and consecutive spaces will be reduced to one space character.

If *rightlookup* is specified the search in the input string – '*inputstring*' - starts from the right (from the end).

If *indexbeforelookupfile* is specified it means that search in the given field name - '*lookup-index*' - will take place in the index before looking in the file specified in '*lookup-words*'.

If *onlystring* is specified, the search will occur only in string index.

If *multichew* is specified it means that if there are more than one sub-string matched in given '*regex-strings*', '*lookup-words*' or *'lookup-index*'. All extracted sub-strings from *'inputstring'* will be stored in '*resultstring*' if the match was in the '*regex-strings*' or *'lookup-index*'. If the match was found in the '*lookup-words*' all base synonyms is stored in the *'resultstring'*. The "*chewpos*" variable is now an array with positions.

The *wordSep* parameter specifies word break characters to be used. It could also refer to a Boolware character file. If a file name is used it should be preceded by (@) e.g. (@db.tab.col.chr). If nothing is specified in this parameter the default Boolware character file will be used.

Example:

```
<!-- Check (Swedish) car license number (3 letters followed by 3 digits) -->
<set var="input" string="ayd040 is my car"/>
<call script="regno = chew(input, '\\b[a-zA-Z]{3} ?[0-9]{3}\\b')"/>
regno now contains: ayd040
input now contains: is my car

<!-- Check for city name -->
<set var="input" string="All in NY"/>
<call script="postal = chew(input, '', '@cities.txt', 'Postal_Name')"/>

The file cities.txt:
New York(NY)
Los Angeles(LA)
…etc

postal now contains: New York
input now contains: All in

<!-- Check for cars -->
<set var="input" string="My car is a sl500"/>
<call script="car = chew(input, '', '@cars.txt', '', 'split'"/>)

The file cars.txt:
Mercedes(sl500, clk 320, sl, 500)
Volvo(glt404, pv 444, pv, 444)

car now contains: Mercedes
input now contains: My car is a
```

Compare

`Score = Compare(string1, string2, options, stopWords, percent)`

Compares two strings - *string1* and *string2* - and returns a score depending on the "similarity" between the two strings. 100 means identical after regarding the other parameters ('*options*' and '*stopWords*').

The parameter '*options*' tells how to make the comparison; should case be ignored, should the order of the words be ignored, should phonetic coding be used, see Normalize.

The words separated by a blank that are specified in the parameter '*stopWords*' will **not** take part in the comparison. The '*stopWords*' will also be affected by the parameter '*options*'.

See function *Normalize* to get a better understanding of these parameters ('*options*' and '*stopWords*') they have exactly the same meaning for both functions.

The result ('*Score*') is normally based on the number of "errors" detected by an algorithm by Damerau-Levenshtein. It is 100 when the two strings are identical (0 errors taking in account the parameters '*options*' and '*stopWords*'). 100 is reduced by one by each detected "error". If the parameter '*percent*' is set to 1, the result will be specified in percent.

Example:
```
Compare('Christoffer Kvist', 'Kristoffer Quist', 2) => 100
```

Before the comparison the two strings should be transformed by the Swedish phonetic algorithm which makes them identical.

Example:
```
Compare('Christoffer Kvist', 'Kristoffer Quist') => 96
```

If the two strings are compared as they look - without any phonetic algorithm - the result will be different. Four errors are found: 'Ck'/'K' and 'Kv'/'Qu' and the result will be 100 - 4 = 96.

Example:
```
Compare('Christoffer kale Kvist', 'Kristoffer Quist', 2, 'kalle') => 100
```

Before the comparison the two strings should be transformed by the Swedish phonetic algorithm which makes them identical.

The word 'kale' will be ignored when comparing as it is the same that has been specified in '*stopWords*'. The '*stopWords*' 'kalle' will be transformed by the Swedish phonetic algorithm to 'kale'.

Example:
```
Compare('Christoffer Kvist', 'Kristoffer Quist', 0, '', 1) => 76
```

If the two strings are compared as they look - without any phonetic algorithm - and specifies that the result should be given in percent the result will be: 76. The longest string contains 17 characters. Four errors are found: 'Ck'/'K' and 'Kv'/'Qu' which means that 13 characters are the same (13/17*100 = 76%).

Compress

`text = compress(text)`

Single characters separated by: space, dot (.), slash (/) plus (+) or ampersand (&) will be pulled together into one word.

Note: If any of these punctuation marks are set as *Letter* or *Digit* in the character table "Word forming" at the table, database or system level, then the character is not removed.

Example:

```
I B M
I.B.M
```

will both give the word IBM.

Cos

`number = cos(degrees)`

Calculates cosine for an angle. The angle should be specified in degrees.

Example:

```
cos(deg(pi)) => -1
```

Date

`string = date()`

Returns the current date in the format CCYY-MM-DD.

Example:

```
date() => 2015-07-15
```

Deg

`degrees = deg(radians)`

Converts radians to degrees.

Example:

```
deg(1) => 57.295780
```

Delete

`string = Delete(string, pos, length)`

Removes a specified number of characters from a given position in a string.

The position (*pos*) is specified as an offset starting from 0 (zero).

If '*length*' is omitted then everything from the '*pos*' will be erased.

If string is an array '*pos*' is the start element (starts from 0) and '*length*' is the number of elements to delete.

Example:

String:
```
Delete('The quick brown fox ', 10, 6) => 'The quick fox'
```

Array:
```
names = array('John', 'Fizgerald', 'Kennedy')
Delete(names, 1, 1) => names [0] John, names [1] Kennedy
```

Dropchar

`Dropchar(string [, direction[, count]])`

Erases one or more characters from a specified string from the left or the right.

'*direction*' tells from which direction characters should be removed; valid values are right and left. If no '*direction*' is specified right is assumed.

The parameter '*count*' is optional and the default value is one (1).

Example:
Assume the variable "*str*" contains: "The quick brown fox".

Remove two characters from the right (from the end):
Dropchar(str, right, 2) => 'The quick brown f'

One character will be removed if no '*count*' specified:
Dropchar(str, left) => 'he quick brown fox'

If no parameter but '*str*' is specified, the last character will be removed
Dropchar(str) => 'The quick brown fo'

Dropterm

`Dropterm(string [, direction[, count [, fieldname]]])`

Normalize and erases one or more words from a specified string from the 'left' or 'right'.

'*direction*' tells from which direction characters should be removed; valid values are right and left. If no 'direction' is specified right is assumed.

The parameter '*count*' is optional and the default value is one (1).

The parameter '*fieldname*' can contain a field name from the current specified table and in that case the character set specified for that field will be used. If field name omitted the system default character set will be used.

Example 1:

The variable *str* contains the following words: "Steel Company Boston Mass".

Remove two words from the right (from the end)
```
Dropterm(str, right, 2) => 'STEEL COMPANY'
```

One word will be removed if no '*count*' specified:
```
Dropterm(str, left) => 'COMPANY BOSTON MASS'
```

If no parameter but '*str*' is specified, the last word will be removed
```
Dropterm(str) => 'STEEL COMPANY BOSTON'
```

Example 2:

The variable *str* contains the following words: "Fish & chips,Coffee and Chocolate Cake".

```
Dropterm(str, right, 2) => 'FISH CHIPS COFFEE AND'
```

i.e. all break characters are replaced with space character and words are normalized.

Env

`string = Env(string)`

Reads the specified environment variable.

Example:

```
env('temp') => 'c:\windows\temp'
env('username') => 'Bill'
```

False

The value of Boolean "false" (0).

Iif

`Iif(expr, truepart, falsepart)`

Immediate if returns one of its two parameters based on the evaluation of an expression.

expr	is the expression that is to be evaluated.
truepart	defines what the Iif function returns if the evaluation of *expr* is true.
falsepart	defines what the Iif function returns if the evaluation of *expr* is false.

There are also operators to accomplish the same purpose, generally referred to as ternary operators; ?:, as used in C, C++ and related languages.

expr ? truepart : falsepart

Insert

`string = Insert(substring, string, pos)`

Inserts a string into another, at a specific position. The position is given as an offset, starting from zero. The *pos* value must be a positive value or zero.

Example:

```
Insert('quick ', 'The brown fox.', 4) => 'The quick brown fox.'
```

The Insert function can also be used with arrays to insert a new string at a given position.

```
array = Insert(string, array, pos)
```

The new *string* will be inserted at the given position *pos* and the size of the array will increase by one. If the index position *pos*, refer to a element that does not exist, new empty strings will automatically be inserted into the array up to the newly inserted *string* and the size of the array will grow accordingly to the numbers of element inserted.

Example:

```
names = array
(
[0] => Mike
[1] => Joe
[2] => Eve
[3] => Marie
)
```

Insert 'Tom' after Joe at index position 2:

```
names = Insert('Tom', names, 2)

names = array
(
[0] => Mike
[1] => Joe
[2] => Tom
[3] => Eve
[4] => Marie
)
```

Append a sixth string 'Liz' to the array at index position 5:

```
names = Insert('Liz', names, 5)

names = array
(
[0] => Mike
[1] => Joe
[2] => Tom
[3] => Eve
[4] => Marie
[5] => Liz
)
```

If you want to replace a string element in the array, you use the following syntax:

```
<!-- Replace 'Joe' with 'John' →
<call script="names[1] = 'John'"/>

names = array
(
[0] => Mike
[1] => John
[2] => Tom
[3] => Eve
[4] => Marie
[5] => Liz
)
```

InStr

`i = InStr(string, substring)`

Finds the first occurrence of a substring within a string, and returns its start position.

If the substring is not found zero (0) will be returned.

The function can return the following values:
- If string is "" - InStr returns 0
- If substring is "" - InStr returns 0
- If substring cannot be found - InStr returns 0

- If substring is found within string - InStr returns the offset (start from 1)

LCase, Lower

`string = LCase(string)`

Converts a string to lower case. The computer's "locale" affects how the translation will be done.

Example:

`<set var="str" value="lcase('ABCDE')"/>, will set variable str to "abcde".`

Len, Length

`i = Len(string)`

Returns the length of a string (number of characters) or an array (number of elements).

Example:

```
<set var="i" value="length(str)"/>
```

will assign the variable "i" the length of the text in the variable *str*. If *str* contains "Ericsson AB", 11 will be returned.

Lookup

`string = Lookup(string, file [, sep])`

Returns base term for a given term, or an empty string if not found.

Strings are stored in a file in the database directory and named in this call with a preceding @. The file is in standard Boolware synonym file format, and the Lookup is not case sensitive.

If the parameter '*sep*' is specified all "base synonyms" will be returned separated by the specified string.

For example, assume the following content in a file called 'cmd.txt':

HELP(HILFE, ASSIST, ?)

<set var="str" value="lookup('hilfe', '@cmd.txt' "/>

Any of the terms 'hilfe', 'assist', '?' will return 'HELP' as a result. Of course you can have more than one synonym in the file, but Lookup will always return each "base synonym" (the first term) for a found term.

Match

`index = Match(string, pattern, flags)`

Searches for a substring within a text using RegEx. Match is similar to, but more powerful than InStr as wildcards can be used.

Match() returns an array of pairwise integers, where each pair marks the starting and ending offset where a match has been found in the text. If no match can be found Match() returns an empty array. Use Length() to test the size of the returned array.

The simplest form of regular expression is to search for a fixed text.

```
text = 'My name is Jesse James'
<if test="length(match(text, 'Jesse')) > 0">
  ...
</if>
```

The condition is true if "Jesse" can be found somewhere in the string text. By default regular expressions are case sensitive, so for example "My name is jesse james" would not meet the condition in the example. You can use case insensitive matching by calling with flags = 1.

Refer to separate chapter about RegEx - regular expressions.

Example:

```
arry = Match('the red king', '((red|white) (king|queen))') =>

arry = Array
(
[0] => 4
[1] => 12
)
```

Maxvalue

`val = Maxvalue(a, b [,'flags])`

Returns the greatest value of *a* and *b*.
The parameter flags could contain the string 'floattoint' to eliminate decimals from *val*

Example:
```
<call script="val = maxvalue(10, 12, 'floattoint')" />
The variable val will contain 12
```

Mid, Substr

`s = Mid(string, start [, length])`

Extracts a number of characters from a string.

Parameters

string	Text to extract characters from.
start	Start offset, zero-based. If start is negative or >= the number of characters in the string an empty string will be returned ("").
Length	Number of characters to copy. If left out or if there remain fewer characters in the text, all characters from start to the text end will be returned.

Example:
```
<set var="str" value="'Oak road 28'"/>
```

```
<call script="str2 = mid(str, 4, 4)"/>
str2 contains "road"

<call script="str2 = mid(str, 4)"/>
str2 contains "road 28"
```

Minvalue

`val = Minvalue(a, b [,'flags])`

Returns the smallest value of *a* and *b*.
The parameter flags could contain the string 'floattoint' to eliminate decimals from *val*

Example:
```
<call script="val = minvalue(10, 12, 'floattoint')" />
The variable val will contain 10
```

Mod

`remainder = number Mod(divisor)`

Returns the rest from a division.

Parameters

Divisor	A chosen divisor

The function returns the remainder from a division (*number*/*divisor*). If the *remainder* is 0 (zero) the *divisor* is a factor of *number*.

Example:
You need to determine if a number is even or odd. The *number* is set to 101 and the *divisor* is set to 2.

```
remainder = number mod(divisor) and the remainder is 1, which indicates that
the number 101 is odd.
```

Normalize

`string = Normalize(string, method, stopWords, maxleng, synfile, flags, wordSep)`

Returns a normalized string.

The parameter *string* contains the string to be normalized.
The parameter m*ethod* is an integer that controls how normalization is done, and means the following (in increasing "fuzziness" order):

0 - Case insensitive, diacritical insensitive (accents).
1 - Same as 0 (NOTE could be used in the future)
2 - Swedish phonetics
3 - Western European phonetics
4 - English phonetics
5 - Soundex modified

6 - Soundex standard
7 - Custom phonetic, e03 exit

128   - remove digits and punctuation characters
256   - street address, use both street name and street number but discard the rest
512   - street address, just use the street name; skip street number
1024 - compress, compress multiple consecutive letters
2048 - stem, skip endings by using the stemming algorithm
4096 - split between letters and digits; SL500 will be two words SL and 500

The lowest 7 bits (0 - 6 above) should be interpreted as an integer which specifies the fuzziness when comparing. The remaining bits (8 - 32) should be regarded as flags which will be OR:ed to the other values.

Example:

1155 means:
Western European phonetics (3)
Remove digits and punctuation characters (128)
Compress multiple consecutive letters (1024)

If *stopWords* are given, these are removed from the string.

The parameter *maxleng* tells the maximum number of characters the result should have.

In the *synfile* parameter you could specify synonyms which will replace found words in the input string.

If the parameter *flags* contains the value *noalphanumericsplit* words will not be split between letters and digits (the only value that could be specified in *flags*).

The *wordSep* parameter specified the break characters to be used. It could also refer to a Boolware character file. If a file name is used it should be preceded by an at-sign e.g. (@db.tab.col.chr). If nothing specified in this parameter the default Boolware character file will be used.

Example:

```
Normalize('priscilla presley', 4, '') => 'PRISILA PRESLEY'
```

Numeric

`bool = Numeric(string)`

Controls that only digits occur in a string. The string can contain one or more leading spaces followed by a plus or minus sign, followed by digits, decimal point and spaces.

Example:

```
Numeric('The Daily Planet') => 0 (false)
Numeric('3.14') => 1 (true)
```

Numrange

`numrange(field, increase)`

Expands a numeric interval, for example a street number. The value in Increase is used to compute new lower and upper bounds in the interval.

Example:

Assume a search for Street:north road 28.

```
<call script="str = 'Street:north road 28' "/>
<call script="numrange(str, 5) "/>
```

Numrange(str, 5) => modifies 'str' to: "Street:north road (numrange(23..33))".

The value of "increase" can be given as procent instead.
Numrange(str, 10%) => modifies 'str' till: "Street:north road (numrange((25..30))".

Only

```
bool = Only(string, charString)
```

Controls that a string contains only a specific set of characters, for example to test if a string seems to contain a phone number.

Example:

```
Only(telno, ' +-()0123456789') => 1
```

Pi

```
Pi
```

The value of Pi, approximately 3.14159265.

Rad

```
radians = rad(degrees)
```

Converts degrees to radians.

Example:

```
rad(180) => 3.14159265
```

Replace

```
string = Replace(string, old, new)
```

Searches through a text, replacing one sequence with another.

Example:

```
replace('august strindberg', 's', 'sch') => auguscht schtrindberg
```

Reverse

`string = Reverse(string)`

Reverses the letters in a text.

Example:

```
<set var="str" value="Reverse('Impossible error!')" />
```

Result:

```
!rorre elbissopmI
```

Sin

`number = sin(degrees)`

Computes the sine value of an angle. The angle is given in degrees.

Example:

```
sin(deg(pi)) => 0
```

Sort

`Array = Sort(array)`

Sorts the words in an array ascending.

```
arry = Sort(Split('Mercedes-Benz SL500', ' -'))

arry = array (
0 => 500
1 => Benz
2 => Mercedes
3 => SL
)
```

Space

`string = Space(length)`

Returns a text consisting of a number of spaces.

Example:

```
str = Space(8) => '        '
```

Split

`Array = Split(string, splitChars, flags)`

Split string into array.

The first parameter - string - is the text to be parsed.

The second parameter is used to tell which word breaking characters that should be used, either by given characters or by given filename of the Boolware character table that should be used (e.g. @db.tablename.chr). If not given at all, the Boolware system character table will be used (default.chr).

An array is a numbered list of values. For example the text "Mercedes-Benz SL500" can be split into a word array, put into the variable array:

If the parameter *flags* contains the string '*noalphanumericsplit*' words will not be split between letters and digits.
If the parameter *flags* contains the string '*skiphyphen*', the hyphen will be removed if even if it followed by a digit and is given as a break character.
Both '*noalphanumericsplit*' *and* '*skiphyphen*' can be given separated by comma.

```
arry = Split('Mercedes-Benz SL500', ' -')

The result is an array containing four elements:

arry = array (
0 => Mercedes
1 => Benz
2 => SL
3 => 500
)
```

Sqrt

`number = sqrt(number)`

Computes the square root from a value. The same as 1 ^ (1 / 2).

```
sqrt(625) => 25
625 ^ (1 / 2) => 25
```

Stem

`string = stem(language, string)`

Returns a stemmed string.

```
Stem('en', 'TALKED') => 'TALK'
```

String

`string = string(character, length)`

Returns a text consisting of a number of repeated characters.

Example:

```
<set var="s" value="string('=', 18)" /> => "=================="
```

Syn

`string = syn(string [, synfile])`

Returns synonyms for a given word. Boolware's global synonyms are used.

A file, synfile, containing synonyms could be specified; if no file is specified the global synonym file will be used. The file name must be preceded by an at-sign e.g. (@synonyms.txt).

Example:

```
Syn('job') => 'work,employment'
```

Tan

`number = tan(degrees)`

Computes the tangent for an angle. The angle is given in degrees.

Example:

```
tan(45) => 1
```

Thes

`string = thes(string [, synonyms, children, thesfile])`

Returns a thesaurus-string. Boolware's global thesaurus is used. If synonyms are desired set Synonyms to 1. Set Children to the desired number of sublevels.

A file, thesfile, containing a thesaurus could be specified; if no file is specified the global thesaurus file will be used. The file name must be preceded by an at-sign e.g. (@thesaurus.txt).

Example:

```
Thes('VEHICLE', 1, 2) =>

VEHICLE
 ,CAR,SAAB,AUDI,VOLVO
 ,BOAT,SHIP,VESSLE,MAXI,SVAN
 ,PLANE,PROPELLERPLANE,REAPLANE,AEROPLANE
```

Time

`number = time()`

Returns the current time on the format HH-MM-SS.MSEC

Example:

```
time() => 17:52:48.123
```

ToString

`string = ToString(variable [, separator-string [, flags]])`

Converts a variable to a string. If an array is passed, all elements of it will be concatenated using the supplied value as a separator.

The parameter *flags* can be set to the string '*floattoint*'. If *flags* is set and the variable type is float type it will be converted to integer before it is converted to string. If variable is an array all elements within the array that is of float type will be converted to integer before conversion to string.

Example:

```
Array = Split('Mercedes-Benz SL500', ' -')
str = ToString(Array, ' ')
```

The result 'str' is: Mercedes Benz SL 500

Trim

`string = trim(string [, trimchars])`

Removes leading and trailing spaces from a string.

The parameter '*trimchars*' is a string of characters that should be removed from the start and end of the specified string. If '*trimchars*' is omitted the only character that will be removed is blank.
Characters in '*trimchars*' can be specified as XML-encoded characters e.g. space character can be specified as &#32; (decimal) or &#x20; (hexadecimal)

Example:

```
string = trim('  Stockholm  ')
```

If the search field City contains " Stockholm " (with one leading and two trailing spaces), it will return "Stockholm".

```
string = trim('   Home.Made.', ' .')
```

In this example you have specified that both blanks and periods (.) should be removed. The result will be: "Home.Made".

True

The value of boolean "true" (1).

UCase, Upper

```
string = UCase(string)
```

Converts a text to capitals only. (upper case). The computer's "locale" determines how the conversion is done.

Example: Assume that the City column contains "London";
```
<set var="str" value="ucase(row['city'])" />, will set variable str to
"LONDON".
```

Ver

Returns the version.

```
Ver() =>
Boolware flow 2.8.0.60, Copyright (C) 2001-2022 Softbool AB, all rights
reserved
```

Wash

```
string = Wash(string, replaceStrings, triggerWords, wordsep, flags)
```

"Washes" a string by applying repeated search and replace expressions (regular expression). For example "if numeric followed by "fl" is found last in the string, replace it with an empty string".

The 'replaceStrings' parameter may contain a file name if it's prefixed with @. In that case the search and replace list is read from this file.

The 'triggerWords' parameter may contain a file name - which should be prefixed with @ -. This file contains special terms to look for, for example "mobile" or "home", which, if found, are removed and returned in the global variable "trigterm".

In the parameter 'wordsep' you could specify your own break characters that are used to separate words in the input string.

If the parameter *flags* contains the string '*noalphanumericsplit'* words will not be split between letters and digits.

If the parameter *flags* contains the string '*casesensitive'*, a case sensitive search will be performed.

Example:

```
<!-- Wash input criteria -->
<call script="str = wash('+46 [8] 77 (8.8) [9,9]', '[\\]\\[+ .,()]+~')
```

```
The above returns the following in str: 468778899

Another example:
<!-- Wash input criteria -->
<call script="str = wash(input, '@telnowash.txt', '@trigwords.txt')" />

The telnowash.txt file:
; List of search and replace strings
; Search ~ Replace <CrLf>
;
^\+[0-9]{2}~
^Sök~
^Mob~
^Tele~
tel~
till~
[- ]~
```

The trigwords.txt file:

```
mob(mobile,mobileno,mobiletel,mobilenumber)
home(homeno,hometel,tel,telno)
work(work,workno,company)
```

Regex support

A regular expression (or regex, or pattern) is a text string that describes some set of strings.

Using RegEx, you can:
- see if a string matches a specified pattern as a whole, and
- search within a string for a substring matching a specified pattern.

Some regular expressions match only one string, i.e., the set they describe has only one member. For example, the regular expression 'foo' matches the string 'foo' and no others. Other regular expressions match more than one string, i.e., the set they describe has more than one member. For example, the regular expression 'f*' matches the set of strings made up of any number (including zero) of 'f's.

As you can see, some characters in regular expressions match themselves (such as 'f') and some don't (such as '*'); the ones that don't match themselves instead let you specify patterns that describe many different strings.

Boolware use ICU regular expression implementing "POSIX. 1".

array Match(text, regex [, flags%])

Match() will return an array of pair wise integers, where each pair represents a position where a match has been found in the text. Each position is represented as a start offset and an end offset. If no match is found, Match() returns an empty array. You can use the Length() function to test the size of the returned array.

The simplest use for regular expressions is to check if a string matches a given pattern.

```
text = 'My name is Jesse James'
<if test="length(match(text, 'Jesse')) > 0">
 ...
</if>
```

The condition is true if text includes the string Jesse. By default, regular expressions are case sensitive, so "My name is jesse james" wouldn't match the above condition. You can set flags = 1 to ignore case.

Let's now see something more sophisticated.

Special characters

Here are some (but not all) useful special characters you can use in regular expressions:

.	Any single character
*	Zero or more of the last character
+	One or more of the last character
?	Zero or one of the last character

Characters * + and ? are called quantifiers, because they are used to tell how many of something should be matched. - You can use parentheses to group things. Here are some examples:

be.r	matches e.g. bear or beer
bee?	matches both be and bee
(bee)?	matches both bee and an empty string
gr+	matches gr, grr, grrr, grrrr etc.
argh*	matches arg, argh, arghh, arghhh etc.
(argh)+	matches argh, arghargh, argharghargh etc.

For example, you could use

```
text = "The dog said grrrr"
if match(text, "gr+") then RunAwayFast()
```

Square brackets (character classes)

Square brackets are used to match any one of the characters inside them.

[abc]	matches a single a, b, or c.
[abc]+	matches any combination of a, b and c.

A hyphen indicates "between" in ASCII order.

[a-c]	matches a single a, b, or c.
[c-a]	is a syntax error, because c comes after a in ASCII.
[a-zA-Z]	matches any lower- or upper-case character (but not including ASCII 128 - 255)

A caret at the beginning means "not":

[^z]	matches any character except z
[a^z]	matches a, z, or ^, because ^ doesn't start the expression inside the brackets

If you want to include the], [, ^ or - inside [], use the escape character: \], \[, \^ or \-.
Keep in mind that if an regex expression require an escape sequence you must add an extra escape sign because the flow itself also uses escape sequences e.g. the ‘$-’sign.

Example: In a flow you would like to trim square brackets:
<call script="str=wash('+46 [8] 77 (8.8) [9,9]', '[+ .,()\\]\\[]+~')"/>

Some more special expressions

There are a couple of special escape sequences. Here are the most usual ones (see the syntax rules for more):

\w	Any alphanumeric (word) character. By default, the same as [a-zA-Z0-9_€ÿ], that is, underscore, all numbers and letters including ASCII 128 - 255. See the syntax rules for details.
\d	Any digit. The same as [0-9].
\s	Any whitespace character: space, tab, or newline.
\b	The metacharacter \b is an anchor like the caret and the dollar sign. It matches at a position that is called a "word boundary". Simply put: \b allows you to perform a "whole words only" search using a regular expression in the form of \bword\b. A "word character" is a character that can be used to form words. All characters that are not "word characters" are "non-word characters".

Examples:

\w+	Any word
\d+	Any positive integer
x\s+y	"x y", with one or more whitespace between x and y

Practical introduction

In its simplest form, a regular expression is a string of symbols to match "as is".

Regex	Matches
abc	**ab**cabcabc
234	1**234**5

That's not very impressive yet. But you can see that regex match the first case found, once, anywhere in the input string.

Supported expressions

RegEx supports a wide range of regular expression types. Here is a short list.

x*	Zero or more x's
x+	One or more x's
x?	One or zero x's
x{m,n}	At least m and at most n x's
[A-Z]	Any uppercase character A-Z
.	Any single character except a newline
\w	Any alphanumeric character
\d	Any digit (the same as [0-9])

Quantifiers

So what if you want to match several characters? You need to use a quantifier. The most important quantifiers are *?+. They may look familiar to you from, say, the dir statement of DOS, but they're not exactly the same.

* matches any number of what's before it, from zero to infinity.
? matches zero or one.
+ matches one or more.

Regex	**Matches**
23*4	1245, 12345, 123345
23?4	1245, 12345
23+4	12345, 123345

By default, regex are greedy. They take as many characters as possible. In the next example, you can see that the regex matches as many 2's as there are.

Regex	**Matches**
2*	122223

Special characters

A lot of special characters are available for regex building. Here are some of the more usual ones.

.	The dot matches any single character.
\n	Matches a newline character (or CR+LF combination).
\t	Matches a tab (ASCII 9).
\d	Matches a digit [0-9].
\D	Matches a non-digit.
\w	Matches an alphanumeric character.
\W	Matches a non-alphanumeric character.
\s	Matches a whitespace character.
\S	Matches a non-whitespace character.
\	Use \ to escape special characters. For example, \. matches a dot, and \\ matches a backslash.
^	Match at the beginning of the input string.
$	Match at the end of the input string.

Here are some likely uses for the special characters.

Regex	Matches
1.3	123, 1z3, 133
1.*3	13, 123, 1zdfkj3
\d\d	01, 02, 99, ..
\w+@\w+	a@a, email@company.com

^ and $ are important to regex. Without them, regex match anywhere in the input. With ^ and $ you can make sure to match only a full string, the beginning of the input, or the end of the input.

Regex	Matches	Does not match
^1.*3$	**13**, **123**, **1zdfkj3**	x13, 123x, x1zdfkj3x
^\d\d	**01**abc	a01abc
\d\d$	xyz**01**	xyz01a

Character classes

You can group characters by putting them between square brackets. This way, any character in the class will match one character in the input.

[abc]	Match any of a, b, and c.
[a-z]	Match any character between a and z. (ASCII order)
[^abc]	A caret ^ at the beginning indicates "not". In this case, match anything other than a, b, or c.
[+*?.]	Most special characters have no meaning inside the square brackets. This expression matches any of +, *, ? or the dot.

Here are some sample uses.

Regex	Matches	Does not match
[^ab]	**c**, **d**, **z**	ab
^[1-9][0-9]*$	Any positive integer	Zero, negative or decimal numbers
[0-9]*[,.]?[0-9]+	**.1**, **1**, **1.2**, **100,000**	12.

Grouping and alternatives

It's often necessary to group things together with parentheses (and).

Regex	Matches	Does not match
(ab)+	**ab**, **abab**, **ababab**	aabb
(aa|bb)+	**aa**, **bbaa**, **aabbaaaa**	abab

Notice the **|** operator. This is the Or operator that takes any of the alternatives.
With parentheses, you can also define subexpressions to remember after the match has happened. In the below example, the string what is between (.)

Regex	Matches	Stores
a(\d+)a	**a12a**	12
(\d+)\.(\d+)	**1.2**	1 and 2

In these examples, what is matched by (\d+) gets stored. The regex engine will allow you to retrieve the stored value by a successive call. In Boolware, you get all matching values in a single call, returned as an array of offset pairs (start + end).

Regex examples

Here are a few practical examples of regular expressions. They are provided for learning purposes. In real applications, you should carefully design your regexes to match the exact use.

Email matching

It's often necessary to check if a string is an email address or not. Here's one way to do it.

```
^[A-Za-z0-9_\.-]+@[A-Za-z0-9_\.-]+[A-Za-z0-9_][A-Za-z0-9_]$
```

Explanation

^[A-Za-z0-9_\.-]+	Match a positive number of acceptable characters at the start of the string.
@	Match the @ sign.
[A-Za-z0-9_\.-]+	Match any domain name, including a dot.
[A-Za-z0-9_][A-Za-z0-9_]$	Match two acceptable characters but not a dot. This ensures that the email address ends with .xx, .xxx, .xxxx etc.

This example works for most cases but is not written based on any standard. It may accept non-working email addresses and reject working ones. Fine-tuning is required.

Example:

```
Match('Email me at me@my.address.com, it''s my work address',
      '([A-Za-z0-9_\\.-]+)@([A-Za-z0-9_\\.-]+[A-Za-z0-9_][A-Za-z0-9_])')
```

Return:

```
Array
(
[0] => 12
[1] => 29
)
```

Parsing dates

Date strings are difficult to parse because there are so many variations. You can't always trust VB's own date conversion functions as the date may come in an unexpected or unsupported format. Let's assume we have a date string in the following format: 2002-Nov-14.

```
^\d\d\d\d-[A-Z][a-z][a-z]-\d\d$
```

Explanation

^\d\d\d\d	Match four digits that make up the year.
-	Match the separator dash.
[A-Z][a-z][a-z]	Match a 3-letter month name. The first letter is in upper case.
-	Match the separator dash.
\d\d$	Match two digits that make up the day.

If a match is found, you can be sure that the input string is formatted like a date. However, a regex is not able to verify that it's a real date. For example, it could as well be 5400-Qui-32. This doesn't look like an acceptable date to most applications. If you want to prepare yourself for the stranger dates, you'll have to write a more limiting expression:

```
^20\d\d-(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)-(0[1-9]|[1-2][0-
9]|3[01])$
```

Explanation

^20\d\d	Match four digits that make up the year. The year must be between 2000 and 2099. No other dates please!
-	Match the separator dash.
(Jan|Feb|Mar|Apr |May|Jun|Jul |Aug |Sep|Oct|Nov|Dec)	Match the month abbreviation in English. Now you don't accept the date in any other language.
-	Match the separator dash.
(0[1-9]|[1-2][0-9]|3[01])$	Match two digits that make up the day. This accepts numbers from 01 to 09, 10 to 29 and 30 to 31. What if the user gives 2003-Feb-31? There are limitations to what regexes can do. If you want to validate the string further, you need to use other techniques than regexes.

Web logs

Web server logs come in several formats. This is a typical line in a log file.

```
144.18.39.44 - - [01/Sep/2002:00:03:20 -0700] "GET /resources.html HTTP/1.1"
200 3458 "http://www.aivosto.com/" "Mozilla/4.0 (compatible; MSIE 6.0; Windows
NT 5.1)"
```

As you can see, there are several fields on the line. They conform to a complex format. The fields are different from each other. A human-readable way to define the various fields is here:

```
host - - [date] "GET URL HTTP/1.1" status size "ref" "agent"
```

As you can see, there are fields such as host (visitor's Internet address), date and time (enclosed in square brackets), an HTTP request with file to retrieve (enclosed in quotation marks), numeric status code, numeric size of file, referred field (enclosed in quotation marks), and agent (browser) name (enclosed in quotation marks).
To programmatically parse the line, you need to split it into fields, then look at each field. This is a sample regex that will split the fields.

```
^(\S*) - - \[(.*) .....\] \"....? (\S*) .*\" (\d*) ([-0-9]*) (\"([^"]+)\")?
```

Explanation

^(\S*)	Match any number of non-space characters at the start of the line.
- -	Match the two dashes. They are actually empty fields that might have content in another log file.
\[(.*) .....\]	Match the date inside square brackets. The date consists of a datetime string, a space, and a 5-character time zone indication. To actually use the date you'd need to write a more detailed regex to separate the year, month, day, hour, minute, and second.
\"....? (\S*) .*\"	Match the HTTP request inside quotation marks. First there is a 3 to 4-character verb, such as GET, POST or HEAD. (\S*) matches the actual file that is being retrieved. At the end, .* matches HTTP/1.1 or whatever protocol was used to retrieve the file.
(\d*)	Match a numeric status code.
([-0-9]*)	Match a numeric file size, or - if no number is present.
(\"([^"]+)\")?	Match the "ref" field. It's anything enclosed in quotation marks.

In this example, we've left "agent" unmatched. That does no harm because $ is not used to match the end-of-line. We can leave "agent" unmatched if we're not interested in the field.

This example has been taken from a web log file parser script. To use it in your own code, you have to fine-tune it to suit your log file format. The regex assumes that lines come only in the presented format. If, say, a field is missing or the file contains garbage lines, the regex may fail. This results in an unparsed line.

Finding dollars

```
\$\d{1,3}(,\d{3})*(\.\d{2})?\b
```

Example

```
match('Make \$50,000.00 fast with this no risk system!',
      '\\\$\\d{1,3}(,\\d{3})*(\\.\\d{2})?\\b')
```

Returns

```
Array
(
[0] => 5
[1] => 15
)
```

TWo BIg INitials

```
\b[A-Z]{2}[a-z]+([^A-Za-z]+[A-Z]{2}[a-z]+)*\b
```

Example

```
match('I really ENjoy WRiting LIke THIS',
      '\\b[A-Z]{2}[a-z]+([^A-Za-z]+[A-Z]{2}[a-z]+)*\\b')
```

returns

```
Array
(
[0] => 9
[1] => 14
[2] => 15
[3] => 22
[4] => 23
[5] => 27
)
```

cAPS lOCK sYNDROME

```
\b[a-z][A-Z]+([^A-Za-z]+[a-z][A-Z]+)*\b
```

Example

```
Match('There is SOMETHING Wrong with mY cAPS lOCK Button!',
      '\\b[a-z][A-Z]+([^A-Za-z]+[a-z][A-Z]+)*\\b')
```

Returns

```
Array
(
[0] => 30
[1] => 32
[2] => 33
[3] => 37
[4] => 38
[5] => 42
)
```

Extract integers

```
\b\d+(\s|$)
```

Example

```
<call script="address = '3:E VILLAGATAN 7 LGH 1101'"/>
<call script="integers = split(chew(address, '\\b\\d+(\\s|\$)', '', '',
'multichew'))"/>
```

Returns

```
Array
(
[0] => 7
[1] => 1101
)
```

Chapter 5
COM object

This chapter describes how to use the Boolware COM object to create applications. The COM object is a natural first-choice API if using Microsoft tools such as for example ASP or Visual Basic.

General

A COM object is created using a symbolic name, its ’ProgID’. SoftboolCOM has the ProgID "Softbool.Session.1".

SoftboolCOM exposes an object hierarchy that reminds about a relational database, with additions to reach Boolware’s characteristics such as keyword searching and index term listings etc.

At the top of the hierarchy is the "Session" object. It has properties such as "Version" and "ErrorText" as well as methods such as "Connect" and "Disconnect". In addition, it also contains the list of databases registered with Boolware. Each database in the list in turn contains a list of database tables, in turn containing a list of columns.

Error handling model

By default, any error caused when calling Boolware will cause an exception. The principle idea behind this reasoning is that bug free applications will not make bad calls to Boolware, and if they do, these errors should not pass undetected. This model works fine for many projects, and maybe especially well for Visual Basic programs that rely on "on error" statement to handle these errors. In C#, the following syntax should be used:

```
try {
   CallBoolware();
   }
catch (System.Runtime.InteropServices.COMException ce)
   {
   // Handle error here
   }
```

However, it may not fit other languages or programmers equally well. So, the generation of exceptions on error can be inactivated, using the Session.Exceptions property. When inactivated, the application is fully responsible for checking all return codes from Boolware.

Session

This is the top level object of the hierarchy.

Properties

AutoTruncate

Boolean value that controls if searches should be made for any word that "begins" the same (true), or if an exact match is required (false).

ConnectTimeout

Set the client socket connection timeout in msec. Default is 0 indicate no maximum timeout

Database

The currently selected database. Each session can have one open database at a time.

Databases

The list of databases registered with Boolware.

ErrorCode

Integer value that contains the last error code. Zero if no error occurred, negative if an error occurred. Positive if a warning or informational message was generated.

ErrorText

String that contains the error message for the current ErrorCode.

Exceptions

Boolean value that controls if Boolware errors are flagged as exceptions or not.

Int1 and Int2

Two integer values that will be set by the Execute command.

Version

String that holds version on the format:
Boolware client version: 2.8.0.75

Methods

Close

Closes the current database.

Connect(Server, Session)

Connects to a Boolware server. The "Server" parameter should be an IP address or a name that can be looked up using DNS.

Response = ConnectExecute(Server, Session, Encoding, StateLess, CMD)

Connects to a Boolware server and run supplied CMD immediately and return the outcome of the command.

In the section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Parameters:

Server	an IP-address or a name that can be looked up by using DNS

Session	could contain a session name
Encoding	should be set to the string "utf-8" if session should be a unicode session, otherwise it will be a "ISO-8859-1" session
StateLess	must contain 1 if the query should be run stateless, which means the session will be logged out disconnected directly. If 0 is specified the session remains in Boolware and it is still connected.
CMD	string containing the actual command

Response = ConnectXML(Server, Session, StateLess, Request)
Used to reach the Boolware XML API when connecting. This API is more powerful than the standard API, in the sense that more can be done per network call.

Parameters:

Server	an IP-address or a name that can be looked up by using DNS
Session	could contain a session name
StateLess	must contain 1 if the query should be run stateless, which means the session will be logged out disconnected directly. If 0 is specified the session remains in Boolware and it is still connected.
Request	contain the XML request

ConnectXMLNoResponse(Server, Session, Request)
Used to reach the Boolware XML API when connecting. This API is more powerful than the standard API, in the sense that more can be done per network call.

Parameters:

Server	an IP-address or a name that can be looked up by using DNS
Session	could contain a session name
Request	contain the XML request

No XML response will be returned. To get the result the **Table.Recordset.Moveto** object should be used.

Disconnect

Disconnects from Boolware.

Response = ExecuteXML(Request)

Used to reach the Boolware XML API. This API is more powerful than the standard API, in the sense that more can be done per network call.

XMLNoResponse(Request)
Used to reach the Boolware XML API. This API is more powerful than the standard API, in the sense that more can be done per network call.

Parameters:

Request	contain the XML request

No XML response will be returned. To get the result the **Table.Recordset.Moveto** object should be used.

GetPerfCounters

Note! *This function is deprecated, please use the execute command* ***perfcounters*** *instead. Read more in chapter 1 "Execute commands in Boolware".*

Returns all performance counters
Tips! Check out the Boolware Manager on the "Performance"-tab to see content of all the counters.

Open(DataSourceName)

Selects a database to be made current. The "Name" parameter is the database name, as registered with Boolware.

Example

```
Set SB = CreateObject("Softbool.Session.1")
SB.Connect "127.0.0.1", "MySession"
Debug.Print SB.Version
Debug.Print SB.Databases.Count
SB.Open "MyDatabase"
```

Databases

A collection of the databases that have been registered with Boolware.

Properties

Count

Number of databases.

Item (default)

Identifies one Database in the collection.

Example

```
Set SB = CreateObject("Softbool.Session.1")
SB.Connect "127.0.0.1", "MySession"
Debug.Print SB.Version
Debug.Print SB.Databases.Count

For I = 0 To SB.Databases.Count - 1
  Set DB = SB.Databases(I)
  If DB.Status = 0 Then
    Debug.Print DB.Name & " - " & _
      DB.DataSourceName & " - " & _
      DB.Remark & " - " & _
      DB.Status
  End If
Next
```

Database

Corresponds to the database.

Properties

DataSourceName

The DSN (Data source name) of the database. It is used to identify the data source.

Name

The Database name

Remark

A short descriptive text for the database

Status

Can be one of the following:

0	Online (available for querying)
1	Loading (index loading, not available)
2	Offline (not available)
3	Pending (index loading, not available)
4	Readonly (available for querying, but no online updates).

Tables

A collection of the tables that exists in this database.

Methods

Execute

Issues commands to Boolware, that are executed on all the database Tables. Execute returns an error code, where zero means OK.

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Example

```
SB.Database.Execute "relate(Customers) find Products.ProductName:uncle"
Set Table = SB.Database.Tables("Customers")
Set Cursor = Table.Recordset
Cursor.SelectCols = "*"
For I = 1 To Table.HitCount
   Cursor.MoveTo I
   Debug.Print Cursor.Fields(0).Value
Next
```

Table

Corresponds to a database table.

Properties

Fields
A collection of the columns that exists in the table.

HitCount
Number of found tuples, as a result of the last query.

Indexed
Indicates if Table is indexed at all.

Name
This table's name

RecordCount
Total number of tuples in this table.

Recordset
A collection of tuples, current search result.

Status
Table status, one of the following:

0	Online
1	Loading
2	Offline
3	Pending
4	Readonly

Methods

Execute
Used to execute Boolware commands.

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Example

```
Set Table = SB.Database.Tables("Companies")
Set RS = Table.Execute "find [Name]:s*"
If RS.Count = 0 Then
  Exit Sub
End If
RS.FetchSize = 100
RS.SelectCols = "*"
```

```
For R = 1 To 100
  RS.MoveTo R
  If R = 1 Then
    For I = 0 To RS.Fields.Count - 1
      Debug.Print RS.Fields(I).Name
    Next
  End If
  For I = 0 To RS.Fields.Count - 1
    Set Field = RS.Fields(I)
      Debug.Print Field.Name & _
          " = " & Field.Value
  Next
Next
```

Field

Corresponds to a table column.

Properties

ActualSize

A long value. Use the **ActualSize** property to return the actual length of a Field object's value. For all fields, the **ActualSize** property is read-only.

DefinedSize

Returns a Long value that reflects the defined size of a field as a number of bytes.

The **DefinedSize** and **ActualSize** properties are different. For example, consider a **Field** object with a declared type of **adVarChar** and a **DefinedSize** property value of 50, containing a single character. The **ActualSize** property value it returns is the length in bytes of the single character.

Indexed

Indicates how this Field is indexed. See the "Indexed" class.

Name

Returns a String value that is the field name.

NumericScale

Indicates the scale of numeric values in a Field object. Sets or returns a Byte value, indicating the number of decimal places to which numeric values will be resolved.

Precision

Indicates the degree of precision for numeric values in a Field object. Returns a Byte value, indicating the maximum total number of digits used to represent values.

Type

Indicates the type of a Field object.

Value

Indicates the value assigned to a Field object.

Example

```
For I = 0 To RS.Fields.Count - 1
  Debug.Print RS.Fields(I).Name
Next
For I = 0 To RS.Fields.Count - 1
  Set Field = RS.Fields(I)
  Debug.Print Field.Name & _
        " = " & Field.Value
Next
```

Indexed object

Indicates how a field is indexed.

Properties

Word

True if column is Word indexed.

String

True if column is String indexed.

Compress

True if column is "Compressed" before extracting words. Compressed means that white space and punctuation characters is removed around single letters.

Numeric

Indicates if this column is numerically indexed.

Indexed

True if column is indexed at all.

Field

True if column is field indexed.

Freetext

True if column is part of ’free-text’, i.e. searchable regardless of in which column the text appears.

Permutated

True if column is permutated.

Grouped

True if column index is grouped, for enhanced performance.

Near

True if column is proximity searchable.

Nearline

True if column is proximity searchable, within same line.

Rankable

True if column is rank able (see Recordset.RankMode).

Phonetic

True if column is indexed as words "sound" rather than exact spelling.

Stemmed

True if column is stemmed.

Similarity

True if column is part of similarity index.

LeftTruncation

True if column supports high speed left truncated searches (find all records that contains words that end the same).

MarkupCoded

True if column is parsed to avoid indexing markup tags.

Stopwords

True if column should use stop words.

MemoryMapped

True if column index is mapped into core memory.

Alias

True if the field is an non index alias field

GeoWGS

The field contains coordinate data in Long or Lat in WGS84 format

GeoMeter

The field contains coordinate data in meter format e.g. RT90

GeoMultiple

The field contains coordinate data both for Long and Lat in either WGS84 or Meter format

StringAsIs

The field is indexed as exact string

WordAsIs

The field is indexed as exact word

WithinString

The field is indexed with ‘Within string’ property

Within

The field is indexed with Within property

Case

The field is indexed case sensitive

XmlField
The field contains XML data

SubField
The field is a XML-subfield

Presort
The field is presorted

DataField
The field data is stored in a Boolware data file

PerfCounters

Holds all performance counters received from Boolware server.

Properties

NumCounters
Number of counters stored in counters and timerCounters.

SrvVersion
Version of the connected Boolware server.

SrvStarted
Server start in seconds since 1970 01 01 00:00:00

TotSessions
Total number of sessions connected to the Boolware server

NumSessions
Number of session connected just now to the Boolware server

PeakSessions
Highest number of session connected at a time

ExecSessions
Number of executing sessions just now in Boolware server

PeakexecSessions
Number of the highest executing sessions at a time

Errors
Error reported by Boolware server

TimerStart
Time for the last reset of counters in the server in seconds since 1970 01 01 00:00:00

TimerElapsed
Seconds past since timerStart was reset

NumCommands

Number of commands performed by the Boolware server

Counters

Array of Counter classes each holding respectively counter values see class Counter

TimerCounters

Array of TimerCounter class each holding respectively elapsed time value

Counters

A collection of performance counters.

Properties

Count

Number of counters.

Item (default)

Identifies one Counter in the collection.

Example

```
Set SB = CreateObject("Softbool.Session.1")
SB.Connect "127.0.0.1", "MySession"
SB.GetPerfCounters
Debug.Print SB.PerfCounters.NumCounters

For I = 0 To SB.PerfCounters.NumCounters - 1
  Set pc = SB.PerfCounters.Counters(I)
  Debug.Print pc.CounterName
Next
```

Counter

Counter contains the actual values correlate to the counter type

Properties

CounterType

Identifier of the counter, see CounterType

CounterName

Literal name of the counter

Accumulated

Number of times this counter is updated

ItemValue

Accumulated values for the counter, see CounterAttribute

ThreadTimeValue
Accumulated thread time values for the counter, See CounterAttribute

WallTimeValue
Accumulated wall clock time values for the counter, see CounterAttribute

TimerCounters

A collection of TimerCounter counters.

Properties

Count
Number of counters.

Item (default)
Identifies one TimerCounter in the collection.

TimerCounter

TimerCounter contains elapsed server tick counts if connected to a 64-bit Boolware server (in milliseconds).

Properties

CounterType
Identifier of the counter, see CounterType

ElapsedTime
Elapsed time since the Boolware server was started (in milliseconds)

InfoString
Name of the counter

CounterAttribute

CounterAttribute contains the total accumulated value and the highest and lowest values.
It also contains a descriptive counter name

Properties

CounterName
Name of the counter

Accumulated
The accumulated value for the counter
Depending on the counter type this value contains:
counter type is *ct_XMLREQUESTS* - number of xml performed requests

counter type is *ct_BOOLEANQUERIES* - number of hits for boolean queries
counter type is *ct_SIMQUERIES* - number of hits for similarity queries
counter type is *ct_DATAFETCH* - number of tuples fetched
counter type is *ct_INDEXTERMS* - number of index terms fetched
counter type is *ct_SORTINGS* - number of tuples sorted

MaxValue

The maximum value for the counter

MinValue

The minimum value of the counter

Recordset

A Recordset object represents the entire set of records from a result of an executed query.

Properties

Count

Number of found tuples.

FetchSize

Number of records to fetch per network trip. Can improve performance if set higher than 1.

Fields

A collection of the fields part of the result.

Order

The current record order being used.

SelectCols

The columns that should be part of a result row.

Methods

MoveTo(HitNo)

Reads a specific tuple. The "HitNo" parameter can be between 1 and ’Count’; otherwise an error will occur. Note that this is a scroll cursor – you can fetch any resultrows in any order.

GetValue(FieldName / FieldIndex)

Reads a specific column value from the current tuple. You may use either a name, or an index number to designate the desired column. This is faster than using Fields.Field("Name").Value, since COM doesn’t have to create temporary objects.

Words

A list of searchable words from the database (indexed terms). The desired column shall be specified.

Properties

Method

The desired indexing method to list from

1	Word indexing
2	String indexing
4	Phonetic words
8	Reversed indexing
16	Stemmed words

Zoomed

If true, lists only words part of the current result.

Item

The current term.

Methods

IndexOf(Term)

Returns the ordinal index for an index word.

Example:

```
Set Words = Table.Words(Table.Fields(0).Name)
For W = 0 To 9
  Set Word = Words(W)
  If Word.HitCount = -1 Then
    Exit For
  End If
  Debug.Print Word.String; Word.HitCount
Next
```

Word

One searchable word from a Boolware index.

Properties

HitCount

Number of records that contains the current word.

String

The current word.

Example:

```
Set Words = Table.Words(Table.Fields(0).Name)
For W = 0 To 9
  Set Word = Words(W)
```

```
    If Word.HitCount = -1 Then
      Exit For
    End If
    Debug.Print Word.String; Word.HitCount
  Next
```

Chapter 6
Java client

This chapter describes how to use the Boolware Java client to create applications.

General

The Boolware Java client is distributed as a "Java package". Create an instance of the Boolware.Client class, and then use the members of this class to query the database, fetch tuples etc.

The client communicates with the Boolware server using the character encoding UTF-8, which means that all XML/JSON calls made must have the "encoding" attribute set to UTF-8.

Client

extends java.lang.Object
This is the Boolware client class. Create an instance of this class, connect it with a Boolware server and then use the methods of this class to query the database, fetch tuples etc.

Fields

connectTimeout

public int **connectTimeout**

Connect time out set to milliseconds for socket connection timeout.

Set the **connectTimeout** to a proper value, e.g. 5000; wait for 5 seconds, before the call to *connect* or *executeXML* with the connect is done.

The default value is 0 which indicates that it will wait until connected or until a system error occurs.

Methods

connect

public int **connect**(java.lang.String server,
                              java.lang.String sessName)
    throws java.io.IOException, java.net.SocketTimeoutException

Connects this client with Boolware server.

Parameters:

server	– the name or IP address. of the server to connect with.
sessName	– the session ID to use. If empty, Boolware will create a unique session ID which can be read using getSettings().

Returns:
zero on success; otherwise a negative error or positive warning.
See Also:
connectTimeout

disconnect

public int **disconnect**(boolean logout)
    throws java.io.IOException

Disconnects from Boolware server.

Parameters:
logout – logs out the session if true; otherwise just disconnects.

Returns:
zero on success; otherwise a negative error or positive warning.

attach

public int **attach**(java.lang.String dsn)
    throws java.io.IOException

Attaches to a database (makes it current).

Parameters:
dsn - the Data Source Name of the desired database.

Returns:
zero on success; otherwise a negative error or positive warning.

detach

public int **detach**()
    throws java.io.IOException

Detaches from a database.

Returns:
zero on success; otherwise a negative error or positive warning.

execute

public int **execute**(String table, String cmd)
    throws java.io.IOException

Executes a query at Boolware.

This method could only be used for the boolean commands: FIND, AND, OR, NOT and XOR and the browse commands BACK and FORWARD. The method is used when you query one table.

Use getHitCount() to inspect number of matching records when this method returns zero (success).

Use getQueryTime() to get time of the current query.

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Returns:
zero on success; otherwise a negative error or positive warning.

execute

public Response **execute**(String cmd)
    throws java.io.IOException

Executes a command at Boolware.

public Response **execute**(String server, String sessName, boolean stateLess, String cmd)
    throws java.io.IOException

Executes a command at Boolware with an automatic connection to Boolware server and returns a reply depending on the command

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

If *stateless* is set to true then Boolware server will automatically disconnect the connection. If *stateless* is set to false then the connection will **not** be disconnected but the session will continue to exist on the Boolware server until the function **disconnect(true)** is called or until the session has timed out due to inactivity.

Parameters:

server	A computer name or an IP-address to Boolware server
sessName	The name of the session
stateless	Should be set to true or false
cmd	The actual command

.

These are examples of commands that can be executed using this method: TABLE, RELATE, FIND, AND, OR, NOT and XOR; BACK and FORWARD; commands that start with SET and GET, commands that handle the search set/search queries/search results (eg SAVEQUERY, DELETERESULT, REVIEWSET and the RELATE command (join).

If you want to query more than one table: TABLES(table1, table2, table3...) FIND "column name":query terms.

If you want to perform a related search: RELATE("target table") FIND "search table"."column name":query terms

Use getHitCount("table name") to inspect number of matching records for the specified table, when this method returns zero (success).

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Returns:
A Response object. See description on Response below in this chapter.
NULL on error.

executeXML

```
public java.lang.String  executeXML(java.lang.String server,
                                    java.lang.String sessName,
                                    boolean stateless,
                                    java.lang.String request)
   throws java.io.IOException,  java.net.SocketTimeoutException
```

Performs an XML request with an automatic connection to Boolware server and returns the reply as an XML document.

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

If *stateless* is set to true then Boolware server will automatically disconnect the connection. If *stateless* is set to false then the connection will **not** be disconnected but the session will continue to exist on the Boolware server until the function **disconnect(true)** is called or until the session has timed out due to inactivity.

Parameters:

server	A computer name or an IP address to Boolware server
sessName	The name of the session
stateless	Should be set to true or false
request	The XML request

Returns:
A response formatted as an XML document

```
public int  executeXML(java.lang.String server,
                       java.lang.String sessName,
                       java.lang.String request)
   throws java.io.IOException,  java.net.SocketTimeoutException
```

Performs an XML request with an automatic connection to Boolware server.

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

To fetch the retrieved data the method **fetchTuples** should be used.

Parameters:

server	A computer name or an IP address to Boolware server
sessName	The name of the session
stateless	Should be set to true or false
request	The XML request

Returns:

zero on success; otherwise a negative error or positive warning.

public java.lang.String **executeXML**(java.lang.String request)
throws java.io.IOException

Executes an XML request at Boolware and returns its response.

Parameters:

request	– the XML request.

Returns:
A response, as an XML document.

executeXMLByteResponse

public java.lang.Byte[] **executeXMLByteResponse**(java.lang.String request)
throws java.io.IOException

Executes an XML request at Boolware and returns its response in a byte array.

Parameters:

request	– the XML request.

Returns:
A response, as an XML document in a byte array.

executeXMLNoResponse

public int **executeXMLNoResponse**(java.lang.String request)
throws java.io.IOException

Performs an XML request with an automatic connection to Boolware server without formatting or returning an XML document.

To fetch the retrieved data the method **fetchTuples** should be used.

Parameters:

request	The XML request

Returns:
zero on success; otherwise a negative error or positive warning.

getHitCount

public int **getHitCount**()

Returns current hit count (number of found records).

getHitCount

public int **getHitCount**(String tableName)

Returns current hit count (number of found records) for the specified Table.

getErrorCode

public int **getErrorCode**()

Returns the last error code.

getErrorMessage

public java.lang.String **getErrorMessage**()

Returns the last error message.

getNumberDatabases

```
public int getNumberDatabases()
                throws java.io.IOException
```

Returns number of databases.

Returns:
number of databases.

getDatabaseInfo

```
public DBInfo getDatabaseInfo(int dbNo)
                throws java.io.IOException
```

Returns meta-data information about a database.

Parameters:
dbNo - database index, starting from zero.

Returns:
DBInfo object.
NULL on error.

getNumberTables

```
public int getNumberTables()
              throws java.io.IOException
```

Returns number of tables.

getTableInfo

public TableInfo **getTableInfo**(int tableNo) throws java.io.IOException
public TableInfo **getTableInfo**(jave.lang.String tableName) throws java.io.IOException

Returns information about a table.

Parameters:

tableNo	- table index, starting from zero.
tableName	- name of the requested table

Returns:
TableInfo object.
NULL on error.

getNumberColumns

public int **getNumberColumns**(java.lang.String table)
    throws java.io.IOException

Returns number of columns.

getColumnInfo

public ColumnInfo **getColumnInfo**(java.lang.String table, int columnNo)
    throws java.io.IOException

Returns info about a column.

Parameters:

table	- the table name in the currently attached database.
columnNo	- column index.

Returns:
ColumnInfo object.
NULL on error.

getColumnInfoEx

public ColumnInfoEx **getColumnInfoEx**(java.lang.String table, int columnNo)
    throws java.io.IOException

Returns extended info about a column.

Parameters:

table	- the table name in the currently attached database.
columnNo	- column index.

Returns:
ColumnInfoEx object.
NULL on error.

getKeyValue

```
public java.lang.String getKeyValue(java.lang.String table,
                                    java.lang.String column,
                                    int hitNo)
                             throws java.io.IOException
```

Reads a primary or foreign key value.

Parameters:

table	- the desired table.
column	- the desired column (must be primary or foreign key).
hitNo	- the hit number, starting from zero.

Returns:
the requested value.

getIndexWord

```
public IndexWord[] getIndexWord(java.lang.String table,
                                java.lang.String column,
                                int zoomed,
                                int type,
                                int count,
                                java.lang.String seed)
                         throws java.io.IOException
```

Returns an array of sorted indexed terms (searchable keywords).

Parameters:

table	- the desired table.
column	- the desired column. Special columns $pk, $vsm, $freetext and $category may be used. NOTE As these columns contain a special character ($), they have to be enclosed within quotes (") or brackets ([ ]).
zoomed	- true if only to include words that are part of the current result.
type	- the desired type of terms. A list of all available types can be found in chapter "Execute commands in Boolware" and command "indexex".
count	- desired number of words to read.
seed	- a string from where Boolware should start listing terms.

The parameter 'column' could have more information: type of presentation and order. There are two different types of presentation: *term* (default) when the terms should be presented in alphabetical order and *count* when the terms should be presented in frequency order (number of occurrences). There are to sort orders: *asc* and *desc*. If no sort order is specified ascending will be used.

The parameter seed could have a special sub-command searchterms which indicates that you want to get statistics on query terms used when query Boolware durting a certain time interval.

In the manual "Operations Guide" Chapter 11 "Interactive Query" you could read about frequency index in section "View Frequency Index" and about Query term statistics in section "Statistics on Query Terms".

Returns:
an array of IndexWord objects.
NULL on error.

Important: After the function SubZoom was implemented all column names containing parentheses must be enclosed within quotation marks. See also Operations Guide chapter 11 "Interactive Query" section "Hits projected hierarchically over one or multiple other values (SubZoom)".

getIndexWord

```
public IndexWord[] getIndexWord(java.lang.String table,
                                java.lang.String column,
                                int zoomed,
                                int type,
                                int count,
                                java.lang.String seed,
                                int continuation)
                         throws java.io.IOException
```

Returns an array of sorted indexed terms (searchable keywords).

Parameters:

table	- the desired table.
column	- the desired column. Special columns $pk, $vsm, $freetext and $category may be used. NOTE As these columns contain a special character ($), they have to be enclosed within quotes (") or brackets ([ ]).
zoomed	- true if only to include words that are part of the current result.
type	- the desired type of terms (see getIndexWord above).
count	- desired number of words to read.
seed	- where Boolware should start listing terms (term, term no. or index type).
continuation	- if false the seed parameter will be used else continuation from last term.

The parameter 'column' could have more information: type of presentation and order. There are two different types of presentation: *term* (default) when the terms should be presented in alphabetical order and *count* when the terms should be presented in frequency order (number of occurrences). There are to sort orders: *asc* and *desc*. If no sort order is specified ascending will be used.

The parameter seed could have a special sub-command searchterms which indicates that you want to get statistics on query terms used when query Boolware during a certain time interval.

In the manual "Operations Guide" Chapter 11 "Interactive Query" you could read about frequency index in section "View Frequency Index" and about Query term statistics in section "Statistics on Query Terms".

Example:
This is an example how to get a grouped index hierarchical presented ordered by occurrences (type = 10). In table ‘Person’ there is a column ‘Date’ which is grouped indexed. The date is stored in the following notation: yyyymmdd and grouped: 4, 6, 8.

The terms should be fetched from the entire table (zoom = 0). The type is 10 and the requested number of terms 10. The empty string means that the fetch should start from the beginning of the index.

The command is as follows:

getIndexWord("Person", "Date", 0, 10, 10, "" , 0)

The fetched terms are return as an array of object type IndexWord:

hitCountZoomed	*hitCount*	*termNo*	*text*
14837	14837	0	1945
1318	1318	0	194503
60	60	0	19450321
52	52	0	19450310
48	48	0	19450327
47	47	0	19450314
32	32	0	19450331
32	32	0	19450319
25	25	0	19450307
12	12	0	19450303

When the parameter ‘zoom’ is inactive the hitCountZoomed and hitCount will be the same. The element termNo returns always 0 in this version of Boolware.

If you want to continue and fetch the next 10 terms you just have to activate the parameter ‘continuation’.

getIndexWord("Person", "Date", 0, 10, 10, "" , 1)

The fetched terms are return as an array of object type IndexWord:

hitCountZoomed	*hitCount*	*termNo*	*text*
11	11	0	19450301
11	11	0	19450311
10	10	0	19450317
9	9	0	19450313
9	9	0	19450308
9	9	0	19450316
8	8	0	19450302
8	8	0	19450322
7	7	0	19450324
7	7	0	19450304

Returns:
an array of IndexWord objects.
NULL on error.

Important: After the function SubZoom was implemented all column names containing parentheses must be enclosed within quotation marks. See also Operations Guide chapter 11 "Interactive Query" section "Hits projected hierarchically over one or multiple other values (SubZoom)".

getPerfCounters

***Note!** This function is deprecated, please use the execute command **perfcounters** instead. Read more in chapter 1 "Execute commands in Boolware".*

public PerfCounters **getPerfCounters**()
    throws java.io.IOException

Returns all performance counters from connected Boolware server.
Tips! Check out the Boolware Manager on the "Performance"-tab to see content of all the counters.

getQueryTime

public float **getQueryTime**()

Returns time taken for last query.

Note that a call to this method is only meaningful if the execute(String table, String cmd) has been performed (not execute(String cmd)).

getRankMode

public int **getRankMode**(java.lang.String table)
    throws java.io.IOException

Returns the current rank mode.

Parameters:
table - the desired table.

Returns:
rankmode

•	BNORANK	= 0; // No ranking
•	BOCCRANK	= 1; // Rank by occurrence
•	BFREQRANK	= 2; // Rank by frequency
•	BSIMRANK	= 3; // Rank by similarity
•	BSORTRANK	= 4; // Rank by sort (Ascending)
•	BSORTDRANK	= 5; // Rank by sort (Descending)
•	BWEIGHTOCCRANK	= 6; // Rank by weighted occurence
•	BWEIGHTFREQRANK	= 7; // Rank by weighted frequency
•	BEACHTERMOCCRANK	= 8; // Rank on specified Term
•	BCUSTOMRANK	= 9; // Rank by Custom
•	BFUZZYRANK	= 10; // Rank by fuzzy

setRankMode

public int **setRankMode**(java.lang.String table,
    int rankMode)
    throws java.io.IOException

Sets the new rank mode, returning the previous or a return code.

NOTE a Query will reset the rank mode to BNORANK except when it is a similarity query (BSIMRANK) or a query on weight search terms (BWEIGHTFREQRANK).

Parameters:
table - the desired table.
rankmode - the new desired rank mode.

•	BNORANK	= 0; // No ranking
•	BOCCRANK	= 1; // Rank by occurence
•	BFREQRANK	= 2; // Rank by frequency
•	BSIMRANK	= 3; // Rank by similarity
•	BSORTRANK	= 4; // Rank by sort (Ascending)
•	BSORTDRANK	= 5; // Rank by sort (Descending)

- BWEIGHTOCCRANK = 6; // Rank by weighted occurrence
- BWEIGHTFREQRANK = 7; // Rank by weighted frequency
- BEACHTERMOCCRANK = 8; // Rank on specified Term
- BCUSTOMRANK = 9; // Rank by Custom
- BFUZZYRANK = 10; // Rank by fuzzy

Returns:
the previous rank mode or a return code.

getVersion

public java.lang.String **getVersion**()
throws java.io.IOException

Returns the Boolware Server version followed by the Boolware Java Client version separated by CR/LF.

Returns:
a string containing the Boolware version.

getClientVersion

public java.lang.String **getClientVersion**()
throws java.io.IOException

Returns the version of the Boolware Java Client.

Returns:
A string containing the version of the Boolware Java Client.

reconnectIfExists

```
public int reconnectIfExists(java.lang.String  server,
                             java.lang.String  sessName)
   throws java.io.IOException,  java.net.SocketTimeoutException
```

Connect the current session to Boolware server on requested computer.

Parameters:

const char *server	-	name of server or IP-address, where Boolware server executes
const char *sessName	-	the name of the session to be connected

Boolware Server must be running on a computer in the network, which could be reached by the client before the connection could take place.

The parameter ‘server’ should be the name of the computer in the network or its IP-address; for example 192.168.0.1.

Note that the computer the client is running on must be able to access the server where Boolware is installed.

Returns:
zero if everything went ok, else a negative error code or positive warning

sortResult

```
public int sortResult(java.lang.String table,
                      java.lang.String expression)
                     throws java.io.IOException
```

Sorts the current search result. The sort will be performed on the contents of the specified columns. You could also set the sort order; ascending or descending. By the parameter emptydata=first/last you could control where to "sort" records that do not contain any data in the sort column; first or last. If the first sort column is indexed as string or numeric you could specify the number of records to sort. The number is specified after order (column1 desc:100).

Parameters:

table	- the desired table.
expression	- the desired column(s) separated by comma (,)

expression syntax:

```
<colname> [asc/desc[:nn]] [emptydata='first/last'] [sortalias='col1, col2'] [,]
```

where :

colname	the column name to perform the sort on.
optional:	
asc/desc	ascending or descending; default is ascending
:nn	sort the *nn* first at each sort request
emptydata	*first*/*last* set fictive sort order if no data in column *first* indicates that the empty value will be treated as sort value ascii 0 *last* indicates that the empty value will be treated as sort value ascii 255 default is *last*

sortalias	upon empty data in *colname* use another column to collect data that will be used for sorting.
	Up to 5 sortalias columns, comma separated, can be given i.e. if *col1* is empty try next specified column *col2* etc.
	If column name needs quotation marks make sure to double quote if using the same quotation mark as araound the whole *sortalias* expression. E.g.
	`sortalias='''Col 1'', ''Col 2'''`
	`sortalias="'Col 1', 'Col 2'"`

,	separates multiple sort columns

Example1: The result after a Query is 60.000 articles and you want to sort on the type of publication in ascending order and the publication date in descending order (the newest article first). If no data in the sort column PubDate these records should be "sorted" first.

sortResult("Articles", "Publication asc, PubDate desc emptydata=first")

For a deeper description and more examples see Chapter 2 "API description" section BCSort() above.

Example2: The result after a Query is 60.000 articles and you want to sort on the publication date in descending order (the newest article first). You only want to present the first 25 articles. Publication date must be indexed as string.

```
sortResult("Articles", "PubDate desc:25")
```

For a deeper description and more examples see Chapter 2 "API description" section BCSort() above.

fetchTuples

```
public Tuple[] fetchTuples(java.lang.String table,
                           java.lang.String columns,
                           int hitNo,
                           int blockSize,
                           int maxSize)
                    throws java.io.IOException
```

Fetches tuples from the database. This method is capable of fetching XML element content as well as standard columns.

Parameters:

table	- the desired table.
columns	- the columns/elements to fetch.
hitNo	- the requested record index, starting from zero.
blockSize	- number of records to fetch.
maxSize	- maximum size of any column (zero means no limit).

Returns:
an array of Tuple objects.
NULL on error.

fetchVectors

```
public Tuple[] fetchVectors(java.lang.String table,
                            int vector,
                            int content,
                            int hitNo,
                            int blockSize)
                    throws java.io.IOException
```

Reads exported record vectors, in binary or text format.

Parameters:

table	- the desired table.
vector -	
content -	
hitNo	- the requested record index, starting from zero.
blockSize	- number of records to fetch.

Returns:
an array of Tuple objects.
NULL on error.

createCalcColumn

```
public int createCalcColumn(java.lang.String table,
                            java.lang.String column,
                            java.lang.String formula)
                     throws java.io.IOException
```

Creates a calculated column.

Parameters:

table	- the table, in which to add a new column.
column	- the name of the new column.
formula	- the mathematical expression that forms the column value.

Returns:
zero on success; otherwise a negative error or positive warning.

removeCalcColumn

```
public int removeCalcColumn(java.lang.String table,
                            java.lang.String column)
                     throws java.io.IOException
```

Removes a calculated column.

Parameters:

table	- the table, from where to remove the column.
column	- the name of the column to remove.

Returns:
zero on success; otherwise a negative error or positive warning.

computeStatistics

```
public Statistics computeStatistics(java.lang.String table,
                                    java.lang.String column,
                                    int tileCount)
                             throws java.io.IOException
```

Computes statics for a given column, based on the current search result. The parameter tileCount determines the number of percentiles you want. This value must be between 3 and 8.

To get all limit values for a specified group you should use the **execute** or **executeXML**. The command statistics for **execute** is described in Chapter 1 section "Execute commands in Boolware ".

Parameters:

table	- the table.
column	- the column to compute statistics for.
tileCount	- 3 for tertials, 4 for quartiles etc.

Returns:
Statistics() object.
NULL on error.

getHistory

```
public History[] getHistory(java.lang.String table)
                 throws java.io.IOException
```

Gets query history items.

Parameters:

table	- the table that search history is retrieved for.

Returns:
an array of History() items.
NULL on error.

getSettings

```
public Settings getSettings()
                 throws java.io.IOException
```

Gets session settings.

Returns:
a Settings() object.
NULL on error.

setSettings

```
public int setSettings(Settings settings)
            throws java.io.IOException
```

Sets session settings.

Parameters:

settings	- the new settings.

Returns:
zero on success; otherwise a negative error or positive warning.

getSettingsXML

```
public java.lang.String getSettingsXML()
                         throws java.io.IOException
```

Gets session settings as XML

Returns:
settings as an XML document.

setSettingsXML

public int **setSettingsXML**(java.lang.String XML)
                throws java.io.IOException

Sets session settings as XML.

Parameters:
XML - the new settings.

Returns:
zero on success; otherwise a negative error or positive warning.

Column

extends java.lang.Object

Contains one column value, as read from the underlying data source. Part of Tuple.

Fields

name

`public java.lang.String` **`name`**
The name of this column

type

`public int` **`type`**
The data type of this column, ODBC coded

fetchSize

`public int` **`fetchSize`**
Fetch size

width

`public int` **`width`**
The maximum width of this column

length

`public int` **`length`**
The actual length of this value

value

`public java.lang.String` **`value`**
The data value, may be null and may also include null

bytes

`public java.lang.String` **`bytes`**
The data value (if it’s a BLOB column). May be null.

ColumnInfo

extends java.lang.Object

Holds meta-data information about one column. Returned from Client.getColumnInfo().

Fields

name

`public java.lang.String` **`name`**

Column name

flags

`public int` **`flags`**

Boolware indexing flags. Values are:

•	0x1	- String indexing
•	0x2	- Word indexing
•	0x4	- Phonetic words
•	0x8	- Proximity search
•	0x10	- Similarity search
•	0x20	- Left truncated search
•	0x40	- Proximity within line
•	0x80	- Normalize string (compress)
•	0x100	- Word permutations
•	0x200	- Stemmed words
•	0x400	- Grouped keys (fast interval)
•	0x800	- Rank able column
•	0x1000	- Categorization field
•	0x10000	- Index with field ID
•	0x20000	- Table global free text index
•	0x40000	- Non indexed alias field
•	0x100000	- Field contains markup elements
•	0x200000	- Field uses stop words
•	0x400000	- Field indexed with Within
•	0x1000000	- Foreign key
•	0x2000000	- XML content column
•	0x4000000	- XML subfield (part of XML content column)
•	0x8000000	- Computed (virtual) column
•	0x10000000	- Case sensitive
•	0x40000000	- Memory mapped

size

`public int` **`size`**

Maximum size of this column

type

`public int` **`type`**

ODBC coded data type.

primaryKeySequence

`public int` **`primaryKeySequence`**

If part of primary key, sequence number.

decimalCount

`public int decimalCount`
Number of decimals

ColumnInfoEx

extends java.lang.Object

Holds meta-data information about one column. Returned from Client.getColumnInfoEx().

Fields

name

`public java.lang.String name`
Column name

flags

`public int flags`
Boolware indexing flags. Values are:

0x1	- String indexing
0x2	- Word indexing
0x4	- Phonetic words
0x8	- Proximity search
0x10	- Similarity search
0x20	- Left truncated search
0x40	- Proximity within line
0x80	- Normalize string (compress)
0x100	- Word permutations
0x200	- Stemmed words
0x400	- Grouped keys (fast interval)
0x800	- Rank able column
0x1000	- Categorization field
0x10000	- Index with field ID
0x20000	- Table global free text index
0x40000	- Non indexed alias field
0x100000	- Field contains markup elements
0x200000	- Field uses stop words
0x400000	- Field indexed with Within
0x1000000	- Foreign key
0x2000000	- XML content column
0x4000000	- XML subfield (part of XML content column)
0x8000000	- Computed (virtual) column
0x10000000	- Case sensitive
0x40000000	- Memory mapped

flags2

`public int flags2`
Boolware extending indexing flags. Values are:

0x1	- Geoposition Long or Lat in WGS84 format
0x2	- Geoposition in metric format e.g. RT90
0x4	- Geoposition containing both Long and Lat
0x10	- Field is exact Word
0x20	- Field is exact String
0x80	- Field is Within string
0x100	- Mixed Alias
0x200	- Polygon

•	0x400	- Primary key set by Boolware Manager
•	0x800	- Foreign key set by Boolware Manager
•	0x1000	- Indexed alias field

NOTE!
The variable **flags2** is always **0** if called Boolware server version is less than 2.6.0.49.

size

`public int` **`size`**
Maximum size of this column

type

`public int` **`type`**
ODBC coded data type.

primaryKeySequence

`public int` **`primaryKeySequence`**
If part of primary key, sequence number.

decimalCount

`public int` **`decimalCount`**
Number of decimals

Counter

extends java.lang.Object

Counter contains the actual values correlate to the counter type

Fields

counterType

public int **counterType**;
Identifier of the counter, see CounterType

counterName

public String **counterName**;
Literal name of the counter

accumulated

public double **accumulated**;
Number of times this counter is updated

itemValue

public CounterAttribute **itemValue**;
Accumulated values for the counter, see CounterAttribute

threadTimeValue

public CounterAttribute **threadTimeValue**;
Accumulated thread time values for the counter, See CounterAttribute

wallTimeValue

public CounterAttribute **wallTimeValue**;

Accumulated wall clock time values for the counter, see CounterAttribute

CounterAttribute

extends java.lang.Object

CounterAttribute contains the total accumulated value and the highest and lowest values.
It also contains a descriptive counter name

Fields

counterName

public String **counterName**;
Name of the counter

accumulated

public double **accumulated**;
The accumulated value for the counter
Depending on the counter type this value contains:
counter type is *ct_XMLREQUESTS* - number of xml performed requests
counter type is *ct_BOOLEANQUERIES* - number of hits for boolean queries
counter type is *ct_SIMQUERIES* - number of hits for similarity queries
counter type is *ct_DATAFETCH* - number of tuples fetched
counter type is *ct_INDEXTERMS* - number of index terms fetched
counter type is *ct_SORTINGS* - number of tuples sorted

maxValue

public double **maxValue**;
The maximum value for the counter

minValue

public double **minValue**;
The minimum value of the counter

CounterTypes

extends java.lang.Object

Performance counter types is the content of the *Counter.counterType* field.

Fields

ct_UNKNOWN

public final static int **ct_UNKNOWN** = -1;
Counter is not initiated

ct_XMLREQUESTS

public final static int **ct_XMLREQUESTS** = 0;
Counter is XML request counter containing number of handled SofboolXML_request

ct_BOOLEANQUERIES

public final static int **ct_BOOLEANQUERIES** = 1;
Counter is the boolean query counter containing number of boolean queries

ct_SIMQUERIES

public final static int **ct_SIMQUERIES** = 2;
Counter is the similarity counter containing number of similarity queries

ct_DATAFETCH

public final static int **ct_DATAFETCH** = 3;
counter is the data fetch counter containing number of data fetches

ct_INDEXTERMS

public final static int **ct_INDEXTERMS** = 4;
Counter is the index term counter containing number of index term fetches

ct_SORTINGS

public final static int **ct_SORTINGS** = 5;
Counter is the sort counter containing number of sortings performed

ct_MAXTYPES

public final static int **ct_MAXTYPES** = 6;
Maximum counter type value

DBInfo

extends java.lang.Object

Holds meta-data information about one database. Returned from Client.getDatabaseInfo().

Fields

name
`public java.lang.String name`
Database name

remark
`public java.lang.String remark`
Boolware remark

dsn
`public java.lang.String dsn`
Database DSN, Data Source Name

History

extends java.lang.Object

Contains one query history item. Returned from Client.getHistory().

Fields

text

`public java.lang.String` **`text`**

This is the query expression used

count

`public int` **`count`**

Number of records found, when combined with previous search

intermediateCount

`public int` **`intermediateCount`**

Number of records found, just for this subquery

IndexWord

extends java.lang.Object

Holds information about one index term. Returned from Client.getIndexWord().

Fields

hitCountZoomed

`public int` **`hitCountZoomed`**

Frequency, if zoomed (just terms that exist in current search result)

hitCount

`public int` **`hitCount`**

Number of records that contains this term in the whole database

termNo

`public int` **`termNo`**

Internal term no.

text

`public java.lang.String` **`text`**

The keyword string

PerfCounters

extends java.lang.Object

Holds all performance counters received from Boolware server

Fields

numCounters

public int **numCounters**;
Number of counters stored in counters and timerCounters

srvVersion

public String **srvVersion**;
Version of the connected Boolware server

srvStarted

public int **srvStarted**;
Server start in seconds since 1970 01 01 00:00:00

totSessions

public int **totSessions**;
Total number of sessions connected to the Boolware server

numSessions

public int **numSessions**;
Number of session connected just now to the Boolware server

peakSessions

public int **peakSessions**;
Highest number of session connected at a time

execSessions

public int **execSessions**;
Number of executing sessions just now in Boolware server

peakexecSessions

public int **peakexecSessions**;
Number of the highest executing sessions at a time

errors

public int **errors**;
Error reported by Boolware server

timerStart

public int **timerStart**;
time for the last reset of counters in the server in seconds since 1970 01 01 00:00:00

timerElapsed

public int **timerElapsed**;
Seconds past since timerStart was reset

numCommands

public long **numCommands**;

Number of commands performed by the Boolware server

counters

public Counter [] **counters**;
Array of Counter classes each holding respectively counter values see class Counter

timerCounters

public ElapsedTimeCounter [] **timerCounters**;
Array of ElapsedTimeCounter class each holding respectively elapsed time value

Response

extends java.lang.Object

Contains a reply from a Boolware command. Is returned from Client.execute(String cmd).

Fields

string

`public int` **`string`**
Response string. Should be interpreted differently depending on the current command.

int1

`public int` **`int1`**
First integer respons. Should be interpreted differently depending on the current command.

int2

`public int` **`int2`**
Second integer response. Should be interpreted differently depending on the current command.

Statistics

extends java.lang.Object

Contains statistics computed for one column. Returned from Client.computeStatistics().

Fields

count

`public int` **`count`**
Number of records with values

modeCount

`public int` **`modeCount`**
Frequency of the most common value

mode

`public double` **`mode`**

The most common value

sum

`public double` **`sum`**
Sum of all values

avg

`public double` **`avg`**
Arithmetic average

min

`public double` **`min`**
Lowest value found

max

`public double` **`max`**
Highest value found

stddev

`public double` **`stddev`**
Standard deviation

variance

`public double` **`variance`**
Variance

median

`public double` **`median`**
Median value

upper

`public double` **`upper`**
Upper tile (tertial, quartile, quintile etc)

lower

`public double` **`lower`**
Lower tile (see also upper)

Settings

extends java.lang.Object

Holds session settings. Returned from Client.getSettings().

Fields

sessionID

`public java.lang.String` **`sessionID`**
This session's unique ID

database

`public java.lang.String` **`database`**

The currently attached database

table

`public java.lang.String` **`table`**
The last used Table.

autoTruncation

`public boolean` **`autoTruncation`**
True if automatic query term right truncation is performed by Boolware

proxGap

`public int` **`proxGap`**
Maximum distance between words, for proximity (Near) searches

proxOrder

`public boolean` **`proxOrder`**
True if order of the terms are important (proximity search)

vsmThreshold

`public double` **`vsmThreshold`**
Lowest acceptable score for similarity searches

hitCount

`public int` **`hitCount`**
Number of records from the current query.

TableInfo

extends java.lang.Object

Holds meta-data information about one table. Returned from Client.getTableInfo().

Fields

name

`public java.lang.String` **`name`**
Table name

flags

`public int` **`flags`**
Boolware Table flags. Bits are:

- 0x1 - Table is indexed by Boolware
- 0x70000000 - Table status bits

hitCount

`public int` **`hitCount`**
Number of found tuples in current search result

recordCount

`public int` **`recordCount`**
Total number of tuples in this table (may be -1 if unknown).

Methods

Indexed

`public boolean Indexed()`

Indicates if this table is indexed by Boolware.

Status

`public int Status()`

Returns this table's status. Possible states are:

- 0 - online
- 1 - loading
- 2 - offline
- 3 - pending
- 4 - read only

Tuple

extends java.lang.Object

Holds a single tuple, as read from the data source. Returned from Client.fetchTuples().

Fields

docNo

`public int docNo`

Boolware record ID

score

`public float score`

Score found, when using any rank mode

columns

`public Column[] columns`

The columns

Chapter 7
.NET client

This chapter describes how to use the Boolware .NET client to create applications.

General

The Boolware .NET client is an implementation of the Boolware client library for C# and other .NET languages and distributed as a "Managed Assembly". There are different versions of the Boolware .NET client that targets different versions of .NET.

The following versions of the Boolware .NET client are currently available:

- Boolware .NET Framework 2.0
- Boolware .NET Framework 4.5
- Boolware .NET Standard 2.0

The Boolware clients developed with .NET Framework 4.5 and .NET Standard 2.0 are functionally equivalent and contain the same APIs. These Boolware clients take advantage of more recent additions in .NET languages such as the `async` and `await` keywords, to support asynchronous I/O.

The Boolware .NET clients are contained in the namespace "Boolware". Create an instance of the Boolware.Client class, and then use the members in this class to search, retrieve rows and configure etc.

The client communicates with the Boolware server using the character encoding UTF-8, which means that all XML/JSON calls made must have the "encoding" attribute set to UTF-8.

NOTE: Methods marked with * (asterisk) are not available in Boolware .NET Framework 2.0

Client

This is the Boolware client class. Create an instance of this class, connect it with a Boolware server and then use the methods of this class to query the database, fetch tuples etc.

Methods

Attach

```
public int Attach(string dsn)
```

`public async Task<int> AttachAsync(string dsn)` *****

Attaches to a database (makes it current).

Parameters:
dsn – the DSN (Data Source Name) of the desired database.

Returns:
zero on success; otherwise a negative error code or a positive warning code.

ComputeStatistics

```
public Statistics ComputeStatistics( string table,
                                     string column,
                                     int tileCount )

public async Task<Statistics> ComputeStatisticsAsync( string table,
                                                      string column,
                                                      int tileCount ) *
```

Computes statics for a given column, based on the current search result. The parameter tileCount determines the number of percentiles you want. This value must be between 3 and 8.

To get all limit values for a specified group you should use the **Execute** or **ExecuteXML**. The command statistics for **Execute** is described in Chapter 1 section "Execute commands in Boolware".

Parameters:
table – the desired table.
column – the column to compute statistics for.
tileCount – 3 for tertials, 4 for quartiles etc.

Returns:
An object of type *Statistics* or null on error.

Connect

```
public int Connect( string server,
                    string sessionId,
                    string reserved )

public async Task<int> ConnectAsync( string hostName,
                                     string sessionId,
                                     string reserved ) *
```

Connects this client with a Boolware server.

Parameters:

server	–	the name or IP address of the server.
sessionId	–	the session ID to use. If empty, Boolware will create a unique session ID which can be read using the method GetSettings().
reserved	–	for future use; pass empty string

Returns:
zero on success; otherwise a negative error code or a positive warning code.

CreateCalcColumn

```
public int CreateCalcColumn( string table,
                             string column,
                             string formula )

public async Task<int> CreateCalcColumnAsync( string table,
                                              string column,
                                              string formula ) *
```

Creates a calculated column.

Parameters:

table	– the name of the table in which column should be added.
Column	– the name of the new column.
formula	– the mathematical expression that forms the column value.

Returns:
zero on success; otherwise a negative error code or a positive warning code.

Detach

```
public int Detach()
```

```
public async Task<int> DetachAsync() *
```

Detaches from a database.

Returns:
zero on success; otherwise a negative error code or a positive warning code.

Disconnect

```
public int Disconnect(bool logout)
```

```
public async Task<int> DisconnectAsync(bool logout) *
```

Disconnects the network connection with the Boolware server.

Parameters:
logout – logs out the session if true; otherwise just disconnects.

Returns:
zero on success; otherwise a negative error code or a positive warning code.

Execute

```
public Response Execute(string cmd)
```

```
public async Task<Response> ExecuteAsync(string cmd) *
```

Executes a command at Boolware.

```
public Response Execute( string server,
                         string sessionId,
                         bool stateless,
                         string cmd )

public async Task<Response> ExecuteAsynv( string server,
                                          string sessionId,
                                          bool stateless,
                                          string cmd) *
```

Automatically connects to a Boolware server server and execute a command.

Server is the name or IP-address to the Boolware server.

If *sessionId* is an empty string Boolware will automatically generate a session id, else the specified *sessionId* will be used.

If *stateless* is set to true then Boolware server will automatically disconnect the connection. If *stateless* is set to false then the connection will **not** be disconnected but the session will continue to exist on the Boolware server until the function **disconnect(true)** is called or until the session has timed out due to inactivity.

Cmd is the command to execute.

This is examples of commands that could be used in this method: TABLES, RELATE, FIND, AND, OR, NOT and XOR; BACK and FORWARD; SET and GET session settings, SAVE, REVIEW and DELETE saved sets/saved queries/saved result and the RELATE command (join).

If you want to query more than one table: TABLES(table1, table2, table3...) FIND "column name":query terms.

If you want to perform a related search: RELATE("target table") FIND "search table"."column name":query terms

Use GetHitCount("table name") to inspect number of matching records for the specified table, when this method returns zero (success).

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Returns:
A Response object. See description on Response below in this chapter.
null on error.

ExecuteStream *

```
public Response ExecuteStream(Stream stream) *
```

```
public async Task<Response> ExecuteStreamAsync(Stream stream) *
```

Executes a command at Boolware.

```
public Response ExecuteStream(string server,
                              string sessionId,
                              bool stateless,
                              Stream stream) *
```

```
public async Task<Response> ExecuteStreamAsync(string server,
                                               string sessionId,
                                               bool stateless,
                                               Stream stream) *
```

Makes a connection to the specified Boolware Server and execute the specified command against the Boolware server.

Server is the name or IP-address to the Boolware server.

If *sessionId* is an empty string Boolware will automatically generate a session id, else the specified *sessionId* will be used.

If *stateless* is set to true then Boolware server will automatically disconnect the connection. If stateless is set to false then the connection will not be disconnected but the session will continue to exist on the Boolware server until the function disconnect(true) is called or until the session has timed out due to inactivity.

Stream is the stream that contains the command to execute. Data in the stream must be in UTF-8 format.

This is examples of commands that could be used in this method: TABLES, RELATE, FIND, AND, OR, NOT and XOR; BACK and FORWARD; SET and GET session settings, SAVE, REVIEW and DELETE saved sets/saved queries/saved result and the RELATE command (join).

If you want to query more than one table: TABLES(table1, table2, table3...) FIND "column name":query terms.

If you want to perform a related search: RELATE("target table") FIND "search table"."column name":query terms

Use GetHitCount("table name") to inspect number of matching records for the specified table, when this method returns zero (success).

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Returns:
A Response object. See description on Response below in this chapter.
null on error.

Note:
There is only one response stream for each instance of the Boolware.Client. The response stream is therefor only valid until the next call to the Boolware server. The data in the response stream is UTF-8 encoded.

ExecuteXML

```
public string ExecuteXML(string server,
                         string sessionId,
                         bool stateless,
                         string request)

public async Task<string> ExecuteXMLAsync(string server,
                                          string sessionId,
                                          bool stateless,
                                          string request) *
```

Connects to a Boolware server and sends a request in XML format and returns a response in XML format.

If *sessionId* is an empty string Boolware will automatically generate a session id, else the specified *sessionId* will be used.

If *stateless* is set to true then Boolware server will automatically disconnect the connection. If *stateless* is set to false then the connection will **not** be disconnected but the session will continue to exist on the Boolware server until the function **disconnect(true)** is called or until the session has timed out due to inactivity.

Parameters:

server	– a computer name or an IP address to Boolware server.
sessionId	– the session id.
stateless	– should be set to true or false.
request	– the XML request.

Return:
A response formatted as an XML document

```
public int ExecuteXML( string server,
                       string sessionId,
                       string request )

public async Task<int> ExecuteXMLAsync( string server,
                                        string sessionId,
                                        string request) *
```

Connects to a Boolware server and sends a request in XML format.
To fetch the retrieved data the method **FetchTuples** should be used.

If *sessionId* is an empty string Boolware will automatically generate a session id, else the specified *sessionId* will be used.

Parameters:

server	– a computer name or an IP address to Boolware server.
sessionId	– the session id.
request	– the XML request.

Returns:
zero on success; otherwise a negative error code or a positive warning code.

```
public string ExecuteXML(string request)

public async Task<string> ExecuteXMLAsync(string request) *
```

Executes an XML request at Boolware and returns its an XML response.

Parameters:
request – the XML request.

Returns:
A response, as an XML document.

ExecuteXMLByteResponse

```
public byte[] ExecuteXMLByteResponse(string request)

public async Task<byte[]> ExecuteXMLByteResponseAsync(string request) *
```

Executes an XML request at Boolware and returns its response in a byte array.

Parameters:
request – the XML request.

Returns:
A response, as an XML document in a byte array.

ExecuteXMLNoResponse

```
public int ExecuteXMLNoResponse(string request)

public async Task<int> ExecuteXMLNoResponseAsync(string request) *
```

Perform an XML request against a connected Boolware server without formatting and returning an XML document

To fetch the retrieved data the method **FetchTuples** should be used.

Parameters:
request – the XML request

Returns:
zero on success; otherwise a negative error code or a positive warning code.

FetchTuples

```
public Tuple[] FetchTuples( string table,
                            string columns,
                            int hitNo,
                            int blockSize,
                            int maxSize )

public async Task<Tuple[]> FetchTuplesAsync( string table,
                                             string columns,
                                             int hitNo,
                                             int blockSize,
                                             int maxSize) *
```

Reads rows from database. This method can read both common standard columns and XML fields.

Parameters:

table	– the desired table.
columns	– the columns/elements to fetch.
hitNo	– the requested record index, starting from zero.
blockSize	– number of records to fetch.
maxSize	– maximum size of any column (zero means no limit).

Returns:
an array of Tuple objects.
null on error.

FetchVectors

```
public Tuple[] FetchVectors( string table,
                             int vector,
                             int content,
                             int hitNo,
                             int blockSize )

public async Task<Tuple[]> FetchVectorsAsync( string table,
                                              int vector,
                                              int content,
                                              int hitNo,
                                              int blockSize ) *
```

Reads exported record vectors, in binary or text format.

Parameters:

table	– the desired table.
vector	– controls which vector is fetched, Query(1) or Record (2).
content	– controls if terms are plain test (1) or coded (2).
hitNo	– the requested record index, starting from zero.
blockSize	– number of records to fetch.

Returns:
an array of Tuple objects.
null on error.

GetColumnInfo

```
public ColumnInfo GetColumnInfo(string table, int columnNo)

public async Task<ColumnInfo> GetColumnInfoAsync(string table, int columnNo) *
```

Returns info about a column.

Parameters:

table	– the table name in the currently attached database.
columnNo	– column index.

Returns:
ColumnInfo object.
null on error.

GetColumnInfoEx

```
public ColumnInfoEx GetColumnInfoEx(string table, int columnNo)

public async Task<ColumnInfoEx> GetColumnInfoExAsync( string table,
                                                      int columnNo) *
```

Returns extended info about a column.

Parameters:

table	– the table name in the currently attached database.
columnNo	– column index.

Returns:
ColumnInfoEx object.
null on error.

GetDatabaseInfo

```
public DBInfo GetDatabaseInfo(int dbNo)

public async Task<DBInfo> GetDatabaseInfoAsync(int dbNo) *
```

Returns meta-data information about a database.

Parameters:
dbNo – database index, starting from zero.

Returns:
DBInfo object.
null on error.

GetErrorCode

```
public int GetErrorCode()
```

Returns:
last error code.

GetErrorMessage

```
public string GetErrorMessage()
```

Returns:
last error message.

GetHistory

```
public History[] GetHistory(string table)

public async Task<History[]> GetHistoryAsync(string table) *
```

Gets query history items.

Parameters:
table – the table that search history is retrieved for.

Returns:
an array of History() items.
null on error.

GetHitCount

```
public int GetHitCount(string table)
```

Returns current hit count (number of found records) for the specified Table.

Parameters:
table – the table name.

GetIndexWord

```
public IndexWord[] GetIndexWord( string table,
                                 string column,
                                 int zoomed,
                                 int type,
                                 int count,
                                 string seed )

public async Task<IndexWord[]> GetIndexWordAsync( string table,
                                                  string column,
                                                  int zoomed,
                                                  int type,
                                                  int count,
                                                  string seed ) *
```

Returns an array of sorted indexed terms (searchable keywords).

Parameters:

table	–	the desired table.
column	–	the desired column. Special columns $pk, $vsm, $freetext and $category may be used. NOTE As these columns contain a special character ($), they have to be enclosed within quotes (") or brackets ([ ]).
zoomed	–	true if only to include words that are part of the current result.
type	–	the desired type of terms. A list of type can be found in chapter "Execute commands in Boolware" and command "indexex".
count	–	desired number of words to read.
seed	–	where Boolware should start listing terms (term, term number or index type).

The parameter 'column' could have more information: type of presentation and order. There are two different types of presentation: *term* (default) when the terms should be presented in alphabetical order and *count* when the terms should be presented in frequency order (number of occurrences). There are to sort orders: *asc* and *desc*. If no sort order is specified ascending will be used.

The parameter seed could have a special sub-command searchterms which indicates that you want to get statistics on query terms used when query Boolware during a certain time interval.

In the manual "Operations Guide" Chapter 11 "Interactive Query" you could read about frequency index in section "View Frequency Index" and about Query term statistics in section "Statistics on Query Terms".

Returns:
an array of IndexWord objects.
null on error.

Important: After the function SubZoom was implemented all column names containing parentheses must be enclosed within quotation marks. See also Operations Guide chapter 11 "Interactive Query" section "Hits projected hierarchically over one or multiple other values (SubZoom)".

GetIndexWord2

```
public IndexWord[] GetIndexWord2( string table,
                                  string column,
                                  int zoomed,
                                  int type,
                                  int count,
                                  string seed,
                                  int continuation )

public async Task<IndexWord[]> GetIndexWord2Async( string table,
                                                   string column,
                                                   int zoomed,
                                                   int type,
int count,
string seed,
int continuation ) *
```

Returns an array of sorted indexed terms (searchable keywords).

Parameters:

table	–	the desired table.
column	–	the desired column. Special columns $pk, $vsm, $freetext and $category may be used. NOTE As these columns contain a special character ($), they have to be enclosed within quotes (") or brackets ([ ]).
zoomed	–	true if only to include words that are part of the current result.
type	–	the desired type of terms (same as GetIndexWord)
count	–	desired number of words to read.
seed	–	where Boolware should start listing terms (term, term number or index type).
continuation	–	if false the seed parameter will be used else continuation from last term.

The parameter 'column' could have more information: type of presentation and order. There are two different types of presentation: *term* (default) when the terms should be presented in alphabetical order and *count* when the terms should be presented in frequency order (number of occurrences). There are to sort orders: *asc* and *desc*. If no sort order is specified ascending will be used.

The parameter seed could have a special sub-command searchterms which indicates that you want to get statistics on query terms used when query Boolware during a certain time interval.

In the manual "Operations Guide" Chapter 11 "Interactive Query" you could read about frequency index in section "View Frequency Index" and about Query term statistics in section "Statistics on Query Terms".

Example:
This is an example how to get a grouped index hierarchical presented ordered by occurrences (type = 10). In table ‘Person’ there is a column ‘Date’ which is grouped indexed. The date is stored in the following notation: yyyymmdd and grouped: 4, 6, 8.

The terms should be fetched from the entire table (zoom = 0). The type is 10 and the requested number of terms 10. The empty string means that the fetch should start from the beginning of the index.

The command is as follows:

```
GetIndexWord2("Person", "Date", 0, 10, 10, "", 0)
```

The fetched terms are return as an array of object type IndexWord:

hitCountZoomed	*hitCount*	*termNo*	*text*
14837	14837	0	1945
1318	1318	0	194503
60	60	0	19450321
52	52	0	19450310
48	48	0	19450327
47	47	0	19450314
32	32	0	19450331
32	32	0	19450319
25	25	0	19450307
12	12	0	19450303

When the parameter ‘zoom’ is inactive the hitCountZoomed and hitCount will be the same. The element termNo always returns 0 in this version of Boolware.

If you want to continue and fetch the next 10 terms you just have to activate the parameter 'continuation'.

```
GetIndexWord2("Person", "Date", 0, 10, 10, "", 1)
```

The fetched terms are return as an array of object type IndexWord:

hitCountZoomed	*hitCount*	*termNo*	*text*
11	11	0	19450301
11	11	0	19450311
10	10	0	19450317
9	9	0	19450313
9	9	0	19450308
9	9	0	19450316
8	8	0	19450302
8	8	0	19450322
7	7	0	19450324
7	7	0	19450304

Returns:
an array of IndexWord objects.
null on error.

Important: After the function SubZoom was implemented all column names containing parentheses must be enclosed within quotation marks. See also Operations Guide chapter 11 "Interactive Query" section "Hits projected hierarchically over one or multiple other values (SubZoom)".

GetKeyValue

```
public string GetKeyValue(string table,
                          string column,
                          int hitNo )

public async Task<string> GetKeyValueAsync( string table,
                                            string column,
                                            int hitNo ) *
```

Reads a primary or foreign key value.

Parameters:

table	– the desired table.
column	– the desired column (must be primary or foreign key).
hitNo	– the hit number, starting from zero.

Returns:
the requested value.

GetNumberColumns

```
public int GetNumberColumns(string table)

public async Task<int> GetNumberColumnsAsync(string table) *
```

Returns number of columns.

Parameters:
table – the desired table.

GetNumberDatabases

```
public int GetNumberDatabases()
```

```
public async Task<int> GetNumberDatabasesAsync() *
```

Returns number of databases.

Returns:
number of databases.

GetNumberTables

```
public int GetNumberTables()
```

```
public async Task<int> GetNumberTablesAsync() *
```

Returns number of tables.

GetPerfCounters

Note! *This function is deprecated, please use the execute command* ***perfcounters*** *instead. Read more in chapter 1 "Execute commands in Boolware"*.

```
public PerfCounters GetPerfCounters()
```

```
public async Task<PerfCounters> GetPerfCountersAsync() *
```

Returns all performance counters
Tips! Check out the Boolware Manager on the "Performance"-tab to see content of all the counters.

GetQueryTime

```
public float GetQueryTime()
```

Returns time taken for last query.

GetRankMode

```
public int GetRankMode(string table)
```

```
public async Task<int> GetRankModeAsync(string table) *
```

Returns the current rank mode.

Parameters:
table – the desired table.

Returns:
current rankmode

• BNORANK	0	No ranking
• BOCCRANK	1	Rank by occurrence
• BFREQRANK	2	Rank by frequency
• BSIMRANK	3	Rank by similarity
• BSORTRANK	4	Rank by sort (Ascending)
• BSORTDRANK	5	Rank by sort (Descending)
• BWEIGHTOCCRANK	6	Rank by weighted occurrence
• BWEIGHTFREQRANK	7	Rank by weighted frequency
• BEACHTERMOCCRANK	8	Rank on specified Term
• BCUSTOMRANK	9	Rank by Custom
• BFUZZYRANK	10	Rank by fuzzy

GetSettings

```
public Settings GetSettings()
```

```
public async Task<Settings> GetSettingsAsync() *
```

Gets session settings.

Returns:
a Settings() object.
null on error.

GetSettingsXML

```
public string GetSettingsXML()
```

```
public async Task<string> GetSettingsXMLAsync() *
```

Gets session settings as XML.

Returns:
settings as an XML document.

GetTableInfo

```
public TableInfo GetTableInfo(int tableNo)
```

```
public TableInfo GetTableInfo(string tableName)
```

```
public async Task<TableInfo> GetTableInfoAsync(int tableNo) *
```

```
public async Task<TableInfo> GetTableInfoAsync(string tableName) *
```

Returns information about a table.

Parameters:
tableNo – table index, starting from zero.

tablename – name of a table

Returns:
TableInfo object.
null on error.

GetVersion

```
public string GetVersion()
```

```
public async Task<string> GetVersionAsync() *
```

Returns the Boolware Server version followed by the Boolware .NET Client version separated by a CR/LF.

Returns:
A string containing the Boolware Server version followed by the Boolware .NET Client version separated by a CR/LF.

GetClientVersion

```
public string GetClientVersion()
```

Returns the version of the Boolware .NET Client.

Returns:
A string containing the version of the Boolware .NET Client.

QueryTable

```
public int QueryTable(string table, string cmd)
```

```
public async Task<int> QueryTableAsync(string table, string cmd) *
```

Executes a query at Boolware.

This method could only be used for the boolean commands: FIND, AND, OR, NOT and XOR and the browse commands BACK and FORWARD. The method is used when you query one table.

Use GetHitCount() to inspect number of matching records when this method returns zero (success).

Returns:
zero on success; otherwise a negative error code or a positive warning code.

ReconnectIfExists

```
public int ReconnectIfExists( string server,
                              string sessionId,
                              string reserved )
```

```
public async Task<int> ReconnectIfExistsAsync( string server,
                                               string sessionId,
                                               string reserved ) *
```

Connect the current session to Boolware server on requested computer.

Parameters:
server – server name or IP address, to the Boolware server.
sessionId – the session id.
reserved – for futer use apply empty string.

Boolware Server must be running on a computer in the network, which can be reached by the Boolware client before the connection can take place.

The parameter 'server' should be the name of the computer in the network or its IP-address; for example 192.168.0.1.

Returns:
zero if everything went ok, else a negative error code or a positive warning code.

RemoveCalcColumn

```
public int RemoveCalcColumn(string table, string column)
```

```
public async Task<int> RemoveCalcColumnAsync(string table, string column) *
```

Removes a calculated column.

Parameters:
table – the table that contains the column to be removed.
column – the name of the column to remove.

Returns:
zero on success; otherwise a negative error code or a positive warning code.

SetRankMode

```
public int SetRankMode(string table, int rankMode)
```

```
public async Task<int> SetRankModeAsync(string table, int rankMode) *
```

Sets the new rank mode, returning the previous or a return code.

NOTE a Query will reset the rank mode to BNORANK except when it is a similarity query (BSIMRANK) or a query on weight search terms (BWEIGHTFREQRANK).

Parameters:
table – the desired table.
rankmode – the new desired rank mode.

• BNORANK	0	No ranking
• BOCCRANK	1	Rank by occurrence
• BFREQRANK	2	Rank by frequency
• BSIMRANK	3	Rank by similarity
• BSORTRANK	4	Rank by sort (Ascending)
• BSORTDRANK	5	Rank by sort (Descending)
• BWEIGHTOCCRANK	6	Rank by weighted occurrence

• BWEIGHTFREQRANK	7	Rank by weighted frequency
• BEACHTERMOCCRANK	8	Rank on specified Term
• BCUSTOMRANK	9	Rank by Custom
• BFUZZYRANK	10	Rank by fuzzy

Returns:
the previous rank mode or an error code.

SetSettings

```
public int SetSettings(Settings settings)
```

```
public async Task<int> SetSettingsAsync(Settings settings) *
```

Sets session settings.

Parameters:
settings – the new settings.

Returns:
zero on success; otherwise a negative error code or a positive warning code.

SetSettingsXML

```
public int SetSettingsXML(string xml)
```

```
public async Task<int> SetSettingsXMLAsync(string xml) *
```

Sets session settings as XML.

Parameters:
xml – the new settings.

Returns:
zero on success; otherwise a negative error code or a positive warning code.

SortResult

```
public int SortResult(string table, string expression)
```

```
public async Task< int> SortResultAsync(string table, string expression) *
```

Sorts the current search result. The sort will be performed on the contents of the specified columns. You could also set the sort order; ascending (ASC) or descending (DESC). If no data in a sort column you could control if it should be "sorted" first or last by the parameter emptydata=first/last. If the first sort column it is indexed as string or numeric you could specify the number of records to sort. The number is specified after order (column1 desc:100).

Parameters:
table – the desired table.
expression – the desired column(s) separated by comma (,)

expression syntax:

```
<colname> [asc/desc[:nn]] [emptydata='first/last'] [sortalias='col1, col2'] [,]
```

where:

colname	the column name to perform the sort on.
optional:	
asc/desc	ascending or descending; default is ascending
:nn	sort the *nn* first at each sort request
emptydata	*first*/*last* set fictive sort order if no data in column *first* indicates that the empty value will be treated as sort value ascii 0 *last* indicates that the empty value will be treated as sort value ascii 255 default is *last*

sortalias	upon empty data in *colname* use another column to collect data that will be used for sorting.
	Up to 5 sortalias columns, comma separated, can be given i.e. if *col1* is empty try next specified column *col2* etc.
	If column name needs quotation marks make sure to double quote if using the same quotation mark as around the whole *sortalias* expression. E.g.
	`sortalias=""Col 1", "Col 2""`
	`sortalias="'Col 1', 'Col 2'"`

,	separates multiple sort columns

Example 1: The result after a Query is 60.000 articles and you want to sort on the type of publication in ascending order and the publication date in descending order (the newest article first). If no data in the column PubDate these records will be "sorted" first.

```
SortResult("Articles", "Publication asc, PubDate desc emptydata=first")
```

Example 2: The result after a Query is 60.000 articles and you want to sort on the publication date in descending order (the newest article first). You only want to present the first 25 articles. Publication date must be indexed as string.

```
SortResult("Articles", "PubDate desc:25")
```

For a deeper description and more examples see Chapter 2 "API description" section BCSort() above.

Column

Contains one column value, as read from the underlying data source. Part of Tuple.

Fields

name

`public string name`
The name of this column

type

`public int type`
The data type of this column, ODBC coded

fetchSize

`public int fetchSize`
Fetch size

width

`public int width`
The maximum width of this column

length

```
public int length
```

The actual length of this value

value

```
public string value
```

The data value, may be null and may also include null

bytes

```
public byte[] bytes
```

The column value (if it’s a BLOB column). May be null.

ColumnInfo

Holds meta-data information about one column. Returned from Client.GetColumnInfo().

Fields

name

```
public string name
```

Column name

flags

```
public int flags
```

Boolware indexing flags. Values are:

0x1	- String indexing
0x2	- Word indexing
0x4	- Phonetic words
0x8	- Proximity search
0x10	- Similarity search
0x20	- Left truncated search
0x40	- Proximity within line
0x80	- Normalize string (compress)
0x100	- Word permutations
0x200	- Stemmed words
0x400	- Grouped keys (fast interval)
0x800	- Rankable column
0x1000	- Categorization field
0x10000	- Index with field ID
0x20000	- Table global free text index
0x40000	- Non indexed alias field
0x100000	- Field contains markup elements
0x200000	- Field uses stop words
0x400000	- Field indexed wit Within
0x1000000	- Foreign key
0x2000000	- XML content column
0x4000000	- XML subfield (part of XML content column)
0x8000000	- Computed (virtual) column
0x10000000	- Case sensitive
0x40000000	- Memory mapped

size

```
public int size
```

Maximum size of this column

type

```
public int type
```

ODBC coded data type.

primaryKeySequence

```
public int primaryKeySequence
```

If part of primary key, sequence number. The first sequence number is one (1).

decimalCount

```
public int decimalCount
```

Number of decimals

ColumnInfoEx

Holds meta-data information about one column. Returned from Client.GetColumnInfoEx().

Fields

name

```
public string name
```

Column name

flags

```
public int flags
```

Boolware indexing flags. Values are:

0x1	- String indexing
0x2	- Word indexing
0x4	- Phonetic words
0x8	- Proximity search
0x10	- Similarity search
0x20	- Left truncated search
0x40	- Proximity within line
0x80	- Normalize string (compress)
0x100	- Word permutations
0x200	- Stemmed words
0x400	- Grouped keys (fast interval)
0x800	- Rank able column
0x1000	- Categorization field
0x10000	- Index with field ID
0x20000	- Table global free text index
0x40000	- Non indexed alias field
0x100000	- Field contains markup elements
0x200000	- Field uses stop words
0x400000	- Field indexed wit Within
0x1000000	- Foreign key
0x2000000	- XML content column
0x4000000	- XML subfield (part of XML content column)
0x8000000	- Computed (virtual) column
0x10000000	- Case sensitive
0x40000000	- Memory mapped

flags2

```
public int flags2
```

Boolware indexing flags. Values are:

0x1	- Geoposition Lat or Long in WGS84 forma
0x2	- Geoposition in metric e.g. RT90
0x4	- Geoposition field containing both lat and long
0x10	- Field is exact Word
0x20	- Field is exact String
0x80	- Field is Within string
0x100	- Mixed Alias
0x200	- Polygon

•	0x400	- Primary key set by Boolware Manager
•	0x800	- Foreign key set by Boolware Manager
•	0x1000	- Indexed alias field

NOTE!
The variable **flags2** is always **0** if called Boolware server version is less than 2.6.0.49

size

```
public int size
```
Maximum size of this column

type

```
public int type
```
ODBC coded data type.

primaryKeySequence

```
public int primaryKeySequence
```
If part of primary key, sequence number.

decimalCount

```
public int decimalCount
```
Number of decimals

Counter

Counter contains the actual values correlate to the counter type

Fields

counterType

```
public int counterType;
```
Identifier of the counter, see CounterType

counterName

```
public String counterName;
```
Literal name of the counter

accumulated

```
public double accumulated;
```
Number of times this counter is updated

itemValue

```
public CounterAttribute itemValue;
```
Accumulated values for the counter, see CounterAttribute

threadTimeValue

```
public CounterAttribute threadTimeValue;
```
Accumulated thread time values for the counter, See CounterAttribute

wallTimeValue

```
public CounterAttribute wallTimeValue;
```
Accumulated wall clock time values for the counter, see CounterAttribute

CounterAttribute

CounterAttribute contains the total accumulated value and the highest and lowest values.

It also contains a descriptive counter name

Fields

counterName

```
public String counterName;
```
Name of the counter

accumulated

```
public double accumulated;
```
The accumulated value for the counter
Depending on the counter type this value contains:
counter type is *ct_XMLREQUESTS* - number of xml performed requests
counter type is *ct_BOOLEANQUERIES* - number of hits for boolean queries
counter type is *ct_SIMQUERIES* - number of hits for similarity queries
counter type is *ct_DATAFETCH* - number of tuples fetched
counter type is *ct_INDEXTERMS* - number of index terms fetched
counter type is *ct_SORTINGS* - number of tuples sorted

maxValue

```
public double maxValue;
```
The maximum value for the counter

minValue

```
public double minValue;
```
The minimum value of the counter

CounterTypes

Performance counter types is the content of the *Counter.counterType* field.

Fields

ct_UNKNOWN

```
public final static int ct_UNKNOWN = -1;
```
Counter is not initiated

ct_XMLREQUESTS

```
public final static int ct_XMLREQUESTS = 0;
```
Counter is XML request counter containing number of handled SofboolXML_request

ct_BOOLEANQUERIES

```
public final static int ct_BOOLEANQUERIES = 1;
```
Counter is the boolean query counter containing number of boolean queries

ct_SIMQUERIES

```
public final static int ct_SIMQUERIES = 2;
```
Counter is the similarity counter containing number of similarity queries

ct_DATAFETCH

```
public final static int ct_DATAFETCH = 3;
```
counter is the data fetch counter containing number of data fetches

ct_INDEXTERMS

```
public final static int ct_INDEXTERMS = 4;
```
Counter is the index term counter containing number of index term fetches

ct_SORTINGS

```
public final static int ct_SORTINGS = 5;
```
Counter is the sort counter containing number of sortings performed

ct_MAXTYPES

```
public final static int ct_MAXTYPES = 6;
```
Maximum counter type value

DBInfo

Holds meta-data information about one database. Returned from Client.GetDatabaseInfo().

Fields

name

```
public string name
```
Database name

remark

```
public string remark
```
Boolware remark

dsn

```
public string dsn
```
Database DSN, Data Source Name

History

Contains one query history item. Returned from Client.GetHistory().

Fields

text

```
public string text
```
This is the query expression used

count

```
public int count
```
Number of records found, when combined with previous search

intermediateCount

```
public int intermediateCount
```
Number of records found, just for this sub query

IndexWord

Holds information about one index term. Returned from Client.GetIndexWord().

Fields

hitCountZoomed

```
public int hitCountZoomed
```
Frequency, if zoomed (just terms that exist in current search result)

hitCount

`public int hitCount`

Number of records that contains this term in the whole database

termNo

`public int termNo`

Internal term no.

text

`public string text`

The keyword string

PerfCounters

Holds all performance counters received from Boolware server

Fields

numCounters

`public int numCounters;`

Number of counters stored in counters and timerCounters

srvVersion

`public string srvVersion;`

Version of the connected Boolware server

srvStarted

`public int srvStarted;`

Server start in seconds since 1970 01 01 00:00:00

totSessions

`public int totSessions;`

Total number of sessions connected to the Boolware server

numSessions

`public int numSessions;`

Number of session connected just now to the Boolware server

peakSessions

`public int peakSessions;`

Highest number of session connected at a time

execSessions

`public int execSessions;`

Number of executing sessions just now in Boolware server

peakexecSessions

`public int peakexecSessions;`

Number of the highest executing sessions at a time

errors

`public int errors;`

Error reported by Boolware server

timerStart

`public int timerStart;`

time for the last reset of counters in the server in seconds since 1970 01 01 00:00:00

timerElapsed

```
public int timerElapsed;
```
Seconds past since timerStart was reset

numCommands

```
public long numCommands;
```
Number of commands performed by the Boolware server

counters

```
public  Counter [] counters;
```
Array of Counter classes each holding respectively counter values see class Counter

timerCounters

```
public ElapsedTimeCounter [] timerCounters;
```
Array of ElapsedTimeCounter class each holding respectively elapsed time value

Response

Holds a Boolware command response. Returned from Client.Execute(string cmd).

Fields

String

```
public int String
```
Response string. Should be interpreted depending on the command used.

Int1

```
public int Int1
```
First integer response. Should be interpreted depending on the command used.

Int2

```
public int Int2
```
Second integer response. Should be interpreted depending on the command used.

Settings

Holds session settings. Returned from Client.GetSettings().

Fields

sessionID

```
public string sessionID
```
This session’s unique ID

database

```
public string database
```
The currently attached database

table

```
public string table
```
The last used Table.

autoTruncation

`public bool autoTruncation`
True if automatic query term right truncation is performed by Boolware

proxGap

`public int proxGap`
Maximum distance between words, for proximity (Near) searches

proxOrder

`public bool proxOrder`
True if order of the terms are important (proximity search)

vsmThreshold

`public double vsmThreshold`
Lowest acceptable score for similarity searches

hitCount

`public int hitCount`
Number of records from the last query.

Statistics

Contains statistics computed for one column. Returned from Client.ComputeStatistics().

Fields

count

`public int count`
Number of records with values

modeCount

`public int modeCount`
Frequency of the most common value

mode

`public double mode`
The most common value

sum

`public double sum`
Sum of all values

avg

`public double avg`
Arithmetic average

min

`public double min`
Lowest value found

max

`public double max`
Highest value found

stddev

`public double stddev`

Standard deviation

variance

`public double variance`

Variance

median

`public double median`

Median value

upper

`public double upper`

Upper tile (tertial, quartile, quintile etc)

lower

`public double lower`

Lower tile (see also upper)

TableInfo

Holds meta-data information about one table. Returned from Client.GetTableInfo().

Fields

name

`public string name`

Table name

flags

`public int flags`

Boolware Table flags. Bits are:

- 0x1 - Table is indexed by Boolware
- 0x70000000 - Table status bits

hitCount

`public int hitCount`

Number of found tuples in current search result

recordCount

`public int recordCount`

Total number of tuples in this table (may be -1 if unknown).

Methods

Indexed

```
public bool Indexed()
```

Indicates if this table is indexed by Boolware.

Status

```
public int Status()
```

Returns this table’s status. Possible states are:

- 0 - online

- 1 - loading
- 2 - offline
- 3 - pending
- 4 - read only

Tuple

Holds a single tuple, as read from the data source. Returned from Client.FetchTuples().

Fields

docNo

```
public int docNo
```
Boolware record ID

score

```
public float score
```
Score found, when using any rank mode

columns

```
public Column[] columns
```
The columns

Chapter 8
PHP extension

This chapter describes how to use the Boolware extension module for PHP to create applications.

General

The Boolware extension module for PHP is distributed as php_boolware.dll (for Windows platforms) and php_boolware.so (for Linux).

Regardless of platform, the extension module should be installed into the PHP extensions directory (see ‘extension_dir’ in php.ini).

To have the Boolware extension loaded when PHP starts, specify extension=php_boolware.dll (or .so) in php.ini. If you do not do this, your scripts must begin with dl("php_boolware.dll") to make the Boolware extension available to PHP.

Boolware’s support for php is delivered as a dynamically linked library which means that php must be authorized to load this library. SELinux, Security Enhanced Linux, must be configured to permit this.

This is done in the configuration file for SELinux: /etc/selinux/config

The switch SELINUX should be set to "permissive", se example below.

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#       enforcing - SELinux security policy is enforced.
#       permissive - SELinux prints warnings instead of enforcing.
#       disabled - SELinux is fully disabled.
SELINUX=permissive
# SELINUXTYPE= type of policy in use. Possible values are:
#       targeted - Only targeted network daemons are protected.
#       strict - Full SELinux protection.
SELINUXTYPE=targeted
```

During development, you are encouraged to set ‘display_errors = On’ in php.ini. This will show any error messages from Boolware in your HTML files. You should also use log_errors = On and error_log = c:\temp\phperr.log to have errors logged.

php_boolware

Before anything else, connect with Boolware using $link = bw_connect(). The bw_connect function takes two or three arguments, the server name and session name and a connect timeout. It returns a connection link, which is expected back in all other Boolware functions.

Functions

bw_add_calc_column

integer **bw_add_calc_column**(resource connection, string table, string formula)

Adds a computed column to a table.
You can use formulas to derive new (virtual) columns that are calculated from the values of other columns.

When fetching a row from the data source, computed columns can be included. If so, Boolware will perform the arithmetic and return the calculated value.

Calculated columns can be explicitly removed using **bw_drop_calc_column().**

Parameters:

connection	– the connection link, returned from bw_connect().
table	– the desired table.
new_col	– the name of the new, calculated column.
formula	– the sort mode expression

Returns:
zero on success; otherwise a negative error code.

bw_compute_statistics

array **bw_compute_statistics**(resource connection, string table, string column, integer tiles)

Computes statistical info for a numeric column. The current search result determines which records are part of the statistical set. Info about quartiles, quintiles etc. are part of the result (as ‘upper’ and ‘lower’.) The parameter ‘tiles’ control what will be returned, for example 4 for quartiles, 5 for quintiles etc. In case of error, NULL is returned.

To get all limit values for a specified group you should use the **bw_execute** or **bw_executeXML**. The command statistics for **bw_execute** is described in Chapter 1 section "Execute commands in Boolware".

The result is returned as an array, containing the following entries:

avg	- Arithmetic average value (sum / cnt)
count	- Number of records with non-null values
lower	- Lower tile
max	- Highest found value
median	- Median value
min	- Lowest found value
mode	- The most frequent value found
modeCount	- Mode frequency
stddev	- Standard deviance
sum	- Sum of all values
upper	- Upper tile
variance	- Variance

bw_connect

resource **bw_connect**(string server, string session [,connecttimeout])

Connects with a Boolware server, using a specific session ID.

Parameters:

server	- the name, or IP-address, of the computer that hosts Boolware.
session	– desired session ID.
connecttimeout	if supplied, set the max socket connect timeout in msec.

Returns:
A connection link on success; otherwise NULL.

Example:
```
<?php
// Connect to server "192.168.1.100" using session ID "My session"
$link = bw_connect("192.168.1.100", "My session");
?>
```

bw_connectexecute

resource **bw_connectexecute**(string server, string session, string encoding, bool stateless, string cmd [,connecttimeout])

Executes a Boolware command with automatic connect to Boolware server, returning a response string.

Parameters:

server	– the name, or IP-address, of the computer that hosts Boolware.
session	– desired session ID.
Encoding	– desired session encoding, "utf-8" or "iso-8859-1"
Stateless	– should be set to 1 or 0
Cmd	– actual command
connecttimeout	– if supplied, set the max socket connect timeout in msec.

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

If *encoding* is set to the string "utf-8" the session will be set as a unicode session, otherwise it will be a "ISO-8859-1" session

If *stateless* is 1 the session will be disconnected and logged out automatically by Boolware server, else the session will stay alive until **BCDisconnect** has been called.

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Returns:
A response string depending on the actual command

bw_connectxml

string **bw_ connectxml**(string server,
string sessName,
bool stateless,
string request [, connecttimeout])

Performs an XML request with an automatic connection to Boolware server and return the reply as an XML document.

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

If *stateless* is 1 the session will be disconnected and logged out automatically by Boolware server, else the session will stay alive until **BCDisconnect** has been called.

Parameters:

server	A computer name or an IP-address to Boolware server
sessName	The name of the session
stateless	Should be set to 1 or 0
request	The XML request
connecttimeout	If supplied, set the max socket connect timeout in msec.

Returns:
A response formatted as an XML document

bw_ connectxmlnoresponse

```
string bw_ connectxmlnoresponse(string server,
                                string sessName,
                                string request [, connecttimeout])
```

Performs an XML request with an automatic connection to Boolware server.

If *sessName* is an empty string Boolware will automatically generate a session name, else the specified *sessName* will be used.

To fetch the retrieved data the method **bw_moveto** should be used.

Parameters:

server	A computer name or an IP-address to Boolware server
sessName	The name of the session
request	The XML request
connecttimeout	If supplied, set the max socket connect timeout in msec.

Returns:
zero on success; otherwise a negative error code.

bw_databases

array **bw_databases**(resource connection, integer index)

Returns information about a database.

Parameters:

connection	– the connection link, returned from bw_connect().
index	– database index, starting from zero.

Returns:
The result is returned as an array, containing the following entries:

name	– database name.
remark	– descriptive remark
dsn	– data source name, always unique.
status	– database status, integer. See appendix 1.

bw_databases_count

integer **bw_databases_count**(resource connection)

Returns the number of databases registered with Boolware.

Parameters:
connection – the connection link, returned from bw_connect().

Returns:
Number of databases, or –1 in case of error.

bw_disconnect

integer **bw_disconnect**(resource connection, bool logout)

Disconnects from Boolware, optionally also logging out. Creating new sessions is expensive, so if possible, it is recommended to not log out.

Parameters:
connection – the connection link, returned from bw_connect().
logout – FALSE to leave the session so it can be reconnected later.

Returns:
zero on success; otherwise a negative error or positive warning.

Example:
```
<?php
// Disconnect current connection
bw_disconnect($link, FALSE);
?>
```

bw_drop_calc_column

integer **bw_drop_calc_column**(resource connection, string table, string column)

Removes a calculated column.

Parameters:
connection – the connection link, returned from bw_connect().
table – desired session ID.
column – the name of the calculated column to drop.

Returns:
zero on success; otherwise a negative error code.

bw_error

string **bw_error**(resource connection)

Returns the most recent error message from Boolware.

Parameters:
connection – the connection link, returned from bw_connect().

Returns:
The most recent error message.

Example:
```
<?php
// Open database, print error on failure
if ($bw_open($link, "db") != 0)
   echo bw_error($link);
?>
```

bw_execute

string **bw_execute**(resource connection, string cmd)

Executes a command string at Boolware, returning a response string.

Parameters:
connection – the connection link, returned from bw_connect().
cmd – the Boolware command string. See chapter 11 of the Operations guide for detailed information about Boolware command strings.

Under section "Execute commands in Boolware" in chapter 1 above you will find a detailed description of the commands that could be used in Execute.

Returns:
a response string.

Example:
```
<?php
// Save current search query
printf(bw_execute($link, "savequery name='MyQuery' table='MyTable'"));
?>
```

bw_execute_xml

string bw_execute_xml(resource connection, string request)

Executes an XML request at Boolware, returning an XML response string.

Parameters:
connection – the connection link, returned from bw_connect().
request – the Boolware XML request. See chapter 3 of this book for detailed information about Boolware XML request strings.

Returns:
an XML response string.

bw_execute_xml_noresponse

int **bw_execute_xml_noresponse**(resource connection, string request)

Performs an XML request with an automatic connection to Boolware server without formatting or returning an XML document.

To fetch the retrieved data the method **bw_moveto** should be used.

Parameters:
connection – the connection link, returned from bw_connect().
request – The XML request

Returns:
zero on success; otherwise a negative error code.

bw_fetch_indexword

array **bw_fetch_indexword**(resource connection [, string table, string column, string seed [, int zoom, int type]])

Lists index terms. The "seed" parameter determines from where to start the listing of terms.

Parameters:

Connection	- the connection link, returned from bw_connect().
table	- the desired table.
column	- the column to list.
seed	- from where to start (term, term number or index type).
zoom	- zoom against current result.
type	- index term type. A list of type can be found in chapter "Execute commands in Boolware" and command "indexex".

The parameter 'column' could have more information: type of presentation and order. There are two different types of presentation: *term* (default) when the terms should be presented in alphabetical order and *count* when the terms should be presented in frequency order (number of occurrences). There are to sort orders: *asc* and *desc*. If no sort order is specified ascending will be used.

The parameter seed could have a special sub-command searchterms which indicates that you want to get statistics on query terms used when query Boolware during a certain time interval.

In the manual "Operations Guide" Chapter 11 "Interactive Query" you could read about frequency index in section "View Frequency Index" and about Query term statistics in section "Statistics on Query Terms".

Returns:
read term on success; otherwise NULL. The array contains the elements "count" and "value".

Example:
```
<?php
$term = bw_fetch_indexword($link, table, column, "seed")
while ($term)
   {
   printf("%d %s\n", $term["count"], $term["value"]);
   $term = bw_fetch_indexword($link);
   }
?>
```

Important: After the function SubZoom was implemented all column names containing parentheses must be enclosed within quotation marks. See also Operations Guide chapter 11 "Interactive Query" section "Hits projected hierarchically over one or multiple other values (SubZoom)".

bw_fetch_keycol

string **bw_fetch_keycol**(resource connection, string table, string column, integer hitno)

Reads the value of one key column from the current tuple. "Key column" means either primary key or foreign key column.

Parameters:
connection – the connection link, returned from bw_connect().
table – the desired table.
column – the column to read. This must be a primary or foreign key column.
hitno – the tuple to read. The first tuple is number zero.

Returns:
read value on success; otherwise NULL.

Example:
```
<?php
// Read all primary keys in current search result
for ($i = 0; $i < $bw_hitcount($link, "mytable"); $i++)
   {
   // Read next primary key (the ID column)
   $pk = bw_fetch_keycol($link, "mytable", "ID", $i);
   }
?>
```

bw_fields

array **bw_fields**(resource connection, string table, integer index)

Returns information about a field.

Parameters:
connection – the connection link, returned from bw_connect().
table – the name of the desired table. The special name "@" can be used to get info about fetched tuples (rather than a complete table).
index – field index, starting from zero.

Returns:
The result is returned as an array, containing the following entries:

name	**-**	database name.
type	**-**	data type
actualsize	**-**	actual size of the current field. (tuple only)
value	**-**	field value, as read from the database. (tuple only)
definedsize	**-**	defined size of the current field.
decimalcount	**-**	number of decimals for numeric fields.
flags	**-**	indexing flags, see appendix 1 – constants and records for description.
primarykey	**-**	zero if not part of primary key, otherwise a sequence number.
flags2	-	more index settings see Appendix 1

NOTE!
The **flags2** element is always **0** if called Boolware server version is less than 2.6.0.49.

bw_fields_count

integer **bw_fields_count**(resource connection, string table)

Returns the number of fields in fetched rows, or in a table.

Parameters:

connection	– the connection link, returned from bw_connect().
table	– the table, or '@' to target fetched tuples.

Returns:
Number of fields, or –1 in case of error.

bw_get_perf_counters

array **bw_get_perf_counters**(resource connection)

Read all performance counters from Boolware server.
Tips! Check out the Boolware Manager on the "Performance"-tab to see content of all the counters.

Parameters:

connection	– the connection link, returned from bw_connect().

Returns:
The return value is an array, with the following content:

numCounters	– number of counters.
srvVersion	– Boolware server version.
srvStarted	– start time in seconds since 1970-01-01.
totSessions	– total number of sessions connected to Boolware server.
numSessions	– number sessions connected right now to Boolware server.
peakSessions	– peak value of number sessions connected to Boolware server.
execSessions	– number of executing sessions
peakexecSessions	– peak number of executing sessions.
errors	– number of errors reported.
timerStart	– measure time started in seconds since 1970-01-01.
timerElapsed	– number of seconds since timerStart.
numCommands	– number of commands handled by Boolware server

Counter "**0**" – "**5**" array containing following elements:

counterType	– counter type
counterName	– descriptive counter name
accumulated	– number of times this counter is updated
itemValue	– array containing "counterName", "accumulated", "maxValue" and "minValue" for current counter
threadTimeValue	– array containing "counterName", "accumulated", "maxValue" and "minValue" for current counter for the thread time
wallTimeValue	– array containing "counterName", "accumulated", "maxValue" and "minValue" for current counter for the wall clock time

"**timerCounter0**" – "**timerCounter5**" array containing "countertype", "elapsedTime" and "infoString" where "elapsedTime" is a value in milliseconds since start of server.

bw_get_query

array **bw_get_query**(resource connection, string table, integer querynumber)

This function fetches the requested query from the QueryHistory. The QueryHistory reflects what has happened since the last FIND command. Each query will be saved as a "line" in the QueryHistory. The purpose of the QueryHistory is to give a possibility to "browse" through the search session and continue the refinement from any query within the QueryHistory. For a detailed description see Operations Guide.

The function returns an array.

Parameters:

connection	– the connection link, returned from bw_connect().
table	– the name of the current table.
querynumber	– the number of the wanted query.

Returns:
The result is returned as an array, containing the following entries:

retcode	– zero on success; otherwise a negative error code.
total	– total number of queries in the QueryHistory.
intermediate	– the number of the current Query
query	– the current query (see Boolware query language in Operations Guide).

bw_get_query_lines

array bw_**get_query_lines**(resource connection, string table)

This function fetches the QueryHistory. The QueryHistory reflects what has happened since the last FIND command. Each query will be saved as a "line" in the QueryHistory. The purpose of the QueryHistory is to give a possibility to "browse" through the search session and continue the refinement from any query within the QueryHistory. For a detailed description see Operations Guide

The function returns an array.

Parameters:

connection	– the connection link, returned from bw_connect().
table	– the name of the current table.

Returns:
The result is returned as an array, containing the following entries:

retcode	– zero on success; otherwise a negative error code.
total	– total number of queries in the QueryHistory.
current	– the number of the current Query

bw_get_query_time

float **bw_get_query_time** (resource connection)

This function returns the Boolware internal query time. CPU time (elapsed time minus disc time) used for the last bw_query() method call, in seconds.

Parameters:

connection	– the connection link, returned from bw_connect().

Returns:
Elapsed CPU time in seconds.

Example:
```
<?php
// Get connection, open database and perform a query
$link = bw_connect("192.168.1.100", "My session");
bw_open($link, "database");
bw_query($link, "table", "find column:a* or s*");

// Get number of hits
$lim = bw_hitcount($link, "table");

// Get and print result and Boolware internal query time for the last query
$elapsed = bw_get_querytime($link);
printf("<br>%d records found in %f seconds.<br>\r\n", $lim, $elapsed);
?>
```

bw_get_rankmode

integer **bw_get_rankmode**(resource connection, string table)

Returns the current rank mode. The order of the found tuples depend on the rank mode.

Parameters:
connection – the connection link, returned from bw_connect().
table – the desired table.

Returns:
current rank mode, one of the following:

•	BNORANK	0	No ranking
•	BOCCRANK	1	Rank by occurrence
•	BFREQRANK	2	Rank by frequency
•	BSIMRANK	3	Rank by similarity
•	BSORTRANK	4	Rank by sort (Ascending)
•	BSORTDRANK	5	Rank by sort (Descending)
•	BWEIGHTOCCRANK	6	Rank by weighted occurrence
•	BWEIGHTFREQRANK	7	Rank by weighted frequency
•	BEACHTERMOCCRANK	8	Rank on specified Term
•	BCUSTOMRANK	9	Rank by Custom
•	BFUZZYRANK	10	Rank by fuzzy

bw_get_setting

array **bw_get_setting**(resource connection)

Returns session settings. Use execute_xml() or bw_set_settings_xml() to change settings.

Parameters:
connection – the connection link, returned from bw_connect().

Returns:
The result is returned as an array, containing the following entries:

session_id	– session ID, see bw_connect().
database	– name of selected database, see bw_open()

auto_trunc	– Boolean value for automatic right truncation.
prox_gap	– max distance between terms (proximity search)
prox_order	– true if terms should be in specified order (proximity search)
vsm_threshold	– lowest similarity score for similarity search.

bw_get_settings_xml

array **bw_get_settings_xml**(resource connection)

This function returns current session settings.

Use **execute_xml()** or **bw_get_settings_xml()** to change the session settings.

Parameters:
connection – the connection link, returned from bw_connect().

Returns:
The result is returned as an array, containing the following entries:

retcode	– zero on success; otherwise a negative error code.
settings	– the session settings as an xml-string

A detailed description of the xml-string under the head line, XML elements for session settings, in this document.

bw_hitcount

integer **bw_hitcount**(resource connection, string table)

Returns number of found records from the last query.

Parameters:

connection	– the connection link, returned from bw_connect().
table	– the database table.

Returns:
number of found records, or –1 on error.

Example:
```
<?php
// Read all tuples in the current search result
for ($i = 0; $i < $bw_hitcount($link, "mytable"); $i++)
   {
   // Move to next record
   bw_moveto($link, "mytable", $i);
   }
?>
```

bw_moveto

integer **bw_moveto**(resource connection, string table, integer hitno)

Fetches one tuple from the data source. After a successful call to bw_moveto(), the fields are available via bw_fields().

Parameters:

connection	– the connection link, returned from bw_connect().
table	– the desired table.
hitno	– the tuple to read. The first tuple is number zero.

Returns:
zero on success; otherwise a negative error or positive warning.

Example:
```
<?php
bw_set_select_cols($link, "ID, Name, Phone");
// Read all tuples in the current search result
for ($i = 0; $i < $bw_hitcount($link, "mytable"); $i++)
   {
   // Move to next record
   bw_moveto($link, "mytable", $i);
   }
?>
```

bw_movetoex

array **bw_movetoex**(resource connection, string table, integer hitno)

Fetches one tuple from the data source. After a successful call to bw_movetoex(), the fields are available via bw_fields().

Parameters:

connection	– the connection link, returned from bw_connect().
table	– the desired table.
hitno	– the tuple to read. The first tuple is number zero.

Returns:
The result is returned as an array, containing the following entries:

retcode	– zero on success; otherwise a negative error or positive warning.
score	– search score result value

Example:
```
<?php
bw_set_select_cols($link, "ID, Name, Phone");
// Read all tuples in the current search result
for ($i = 0; $i < $bw_hitcount($link, "mytable"); $i++)
   {
   // Move to next record
   $res = bw_movetoex($link, "mytable", $i);
   if($res['retcode'] < 0)
     error;
   }
?>
```

bw_open

integer **bw_open**(resource connection, string DSN)

Opens a specific database, makes it active.

Parameters:

connection – the connection link, returned from bw_connect().
DSN – the Boolware DSN of the desired database.

Returns:
zero on success; otherwise a negative error or positive warning.

Example:
```
<?php
// Open database "Companies", print error
if (bw_open($link, "Companies") != 0)
   echo bw_error($link);
?>
```

bw_query

integer **bw_query**(resource connection, string table, string cmd)

Executes a query at Boolware, returning number of found records.

Parameters:
connection – the connection link, returned from bw_connect().
table – desired session ID.

Returns:
number of found records, or –1 on error.

Example:
```
<?php
// Find persons in zip code 12345 with a salary above 100
$cmd = 'find zip:"12345" and salary:>100';
$count = bw_query($link, 'mytable', $cmd);
?>
```

bw_reconnectifexists

resource **bw_reconnectifexists**(string server, string session [, connecttimeout])

Connect the current session to Boolware server on requested computer.

Parameters:

server	– server name or IP-address to Boolware server
session	– name of the sessions
connecttimeout	– if supplied, set the max socket connect timeout in msec.

Boolware Server must be running on a computer in the network, which could be reached by the client before the connection could take place.

The parameter '*server*' should be the name of the computer in the network or its IP-address; for example 192.168.0.1.

Note that the computer the client is running on must be able to access the server where Boolware is installed.

Returns:
a connection link if everything went OK, else NULL

Example:

```
<?php
// Reconnect to "192.168.1.100" wuth sessions ID "My session"
$link = bw_reconnectifexists("192.168.1.100", "My session");
?>
```

bw_set_error_mode

integer **bw_set_error_mode**(integer errormode)

Controls how this extension should report errors. By default, any serious error such as for example bw_connect() failing because there is no server will terminate the script. This behavior can be changed, so that the application takes full responsibility for checking return codes.

Parameters:

errormode	– the PHP error mode to use(). One of the following:
1	– print error message and terminate the script (default). E_ERROR
2	– print error message, but do not terminate script. E_WARNING
8	– do not print error message, do not terminate script. E_NOTICE.

Returns:
The new errormode if successful; otherwise NULL.

bw_set_fetch_size

integer **bw_set_fetch_size**(resource connection, integer fetch_size[, maxchars])

When fetching tuples sequentially from Boolware, performance will increase if the client fetches more than one tuple per network call to Boolware. Please note that this does not change the way bw_moveto() behaves. It still fetches a single row at a time.

Be aware that a high value may be counterproductive, may slow things down. A good value is usually something around 50.

To improve the performance further you could specify a third optional parameter, maxchars, which will limit the output of each column to the specified number of characters. This is usable when you just want to show a small number of characters from a huge text column. If the parameter is omitted the total number of characters for each column will be fetched.

Parameters:

connection	– the connection link, returned from bw_connect().
fetch_size	– number of tuples to fetch per network call.
maxchars	– max number of characters for each column (optional)

Returns:
zero on success; otherwise a negative error or positive warning.

Example:
```
<?php
bw_set_fetch_size($link, 50);
// Read all tuples in the current search result
for ($i = 0; $i < $bw_hitcount($link, "mytable"); $i++)
   {
   // Move to next record
   bw_moveto($link, "mytable", $i);
   }
bw_set_fetch_size(1);
```

```
?>
```

bw_set_rankmode

integer bw_set_rankmode(resource connection, string table, integer rankmode)

Sets the current rank mode. The order of the search result tuples depend on the rank mode.

Parameters:
connection – the connection link, returned from bw_connect().
table – the desired table.
rankmode – see bw_get_rankmode() on the previous page.

Returns:
zero on success; otherwise a negative error code.

bw_set_select_cols

integer **bw_set_select_cols**(resource connection, string columns)

Controls which columns that Boolware should fetch from the data source, for each call to bw_moveto().

Parameters:
connection – the connection link, returned from bw_connect().
columns – the columns that should be fetched, a comma separated string. Used in SQL context, so an asterisk (*) means "all columns".

Returns:
zero on success; otherwise a negative error or positive warning.

Example:
```
<?php
bw_set_select_cols($link, "ID, Name, Phone");
// Read all tuples in the current search result
for ($i = 0; $i < $bw_hitcount($link, "mytable"); $i++)
   {
   // Move to next record
   bw_moveto($link, "mytable", $i);
   }
?>
```

bw_set_settings_xml

integer **bw_set_settings_xml**(resource connection, string settings)

This function sets the new session settings.

Use **bw_get_setting()** or **bw_get_settings_xml()** to view the current session settings.

The parametern settings is an xml-string which describes the new session settings. A detailed description of the XML elements is found under the head line, XML elements for session settings, in this document.

Parameters:

connection – the connection link, returned from bw_connect().
settings – the new session settings.

Returns:
zero on success; otherwise a negative error code.

bw_sort_result

integer **bw_sort_result**(resource connection, string table, string expression)

The format of the parameter expression is: column name followed by order type, 'ASC', ascending, or 'DESC', descending. Sort columns containing no data could be "sorted" first or last by using the parameter emptydata=first/last. Separate columns by a comma sign. If the first sort column is indexed as string or numeric you could specify the number of records to sort. The number is specified after order (column desc:100).

Default order type is 'ASC' and can be omitted. Default order for emptydata=last and could be omitted.

For a deeper description and examples see Chapter 2 "API description" section BCSort() above.

Parameters:

connection	– the connection link, returned from bw_connect().
table	– the desired table.
expression	– the sort mode expression

expression syntax:

```
<colname> [asc/desc[:nn]] [emptydata='first/last'] [sortalias='col1, col2'] [,]
```

where :

colname	the column name to perform the sort on.
optional:	
asc/desc	ascending or descending; default is ascending
:nn	sort the *nn* first at each sort request
emptydata	*first*/*last* set fictive sort order if no data in column *first* indicates that the empty value will be treated as sort value ascii 0 *last* indicates that the empty value will be treated as sort value ascii 255 default is *last*

sortalias	upon empty data in *colname* use another column to collect data that will be used for sorting.
	Up to 5 sortalias columns, comma separated, can be given i.e. if *col1* is empty try next specified column *col2* etc.
	If column name needs quotation marks make sure to double quote if using the same quotation mark as around the whole *sortalias* expression. E.g.
	`sortalias='''Col 1'', ''Col 2'''`
	`sortalias="'Col 1', 'Col 2'"`

,	separates multiple sort columns

Returns:
zero on success; otherwise a negative error code.

bw_tables

array **bw_tables**(resource connection, integer index)
or
array **bw_tables**(resource connection, 0, string tableName)

Returns information about a table via table index or via table name.

Parameters:

connection	-	the connection link, returned from bw_connect().
index	-	table index, starting from zero.
or		
connection	-	the connection link, returned from bw_connect().
0	-	integer zero
tableName	-	name of the table

Returns:
The result is returned as an array, containing the following entries:

name	– table name.
recordcount	– number of tuples in the table. –1 if database doesn't support this.
hitcount	– number of tuples in current search result.
indexed	– non-zero if table is indexed by Boolware.
status	– table status, online/offline/readonly etc.
fieldcount	– number of fields in the table.

bw_tables_count

integer **bw_tables_count**(resource connection)

Returns the number of tables within the currently selected database.

Parameters:
connection – the connection link, returned from bw_connect().

Returns:
Number of tables, or –1 in case of error.

bw_version

string **bw_version**(resource connection)

Returns a string that holds the Boolware version, for example:
Boolware server version : 2.3.0.0.

Parameters:
connection – the connection link, returned from bw_connect().

Returns:
The Boolware version, or NULL in case of error.

Example:
```
<?php
// Show Boolware version
echo bw_version($link);
?>
```

Example

A very small example how to use the PHP extension when interacting with Boolware.

The example shows how to query a Column in a Table in a Database and fetch the result.

The following steps should be performed:

1. Connect with Boolware; show version info
2. List all Databases, Tables and Columns
3. Open the first Database
4. Query the first Column in the first Table in the opened Database
5. Fetch the first 20 tuples of the result
6. Tuple loop; Print Column Headers and print Column data
7. Disconnect from Boolware

The code could be as follows:

```
<html>
<head>
<title>Boolware PHP test page</title>
</head>

<body>
  <font name=Arial size=2>

<?php

  phpinfo();

  // Connect with Boolware, show version info
  $link = bw_connect("127.0.0.1", "SessionID");

  printf("<b>Boolware version:</b> %s<br>\r\n", bw_version($link));
  printf("<b>Databases:</b> %d<br><br>\r\n", bw_databases_count($link));

  //
  // List all databases, tables and columns
  //
  for ($i = 0; $i < bw_databases_count($link); $i++)
    {
    $DB = bw_databases($link, $i);
    if ($DB["status"] == 0)
      {
      bw_open($link, $DB["dsn"]);

      printf("<b>Name:</b> %s - <b>DSN:</b> %s - <b>Remark:</b> %s -
              <b>Status:</b> %d",
              $DB["name"], $DB["dsn"], $DB["remark"], $DB["status"]);

      printf(" - <b>No. tables:</b> %d<br>", bw_tables_count($link));
      for($j = 0; $j < bw_tables_count($link); $j++)
        {
        $Table = bw_tables($link, $j);
        printf("\r\n<br><b>Table:</b> %s - <b>Records:</b> %d",
               $Table["name"], $Table["recordcount"]);
        printf("<br><b>Columns:</b> (name, type, size)");
        for ($k = 0; $k < $Table["fieldcount"]; $k++)
          {
          $Field = bw_fields($link, $Table["name"], $k);
          printf("<br>%s - %d - %d", $Field["name"], $Field["type"],
          $Field["definedsize"]);
          }
        echo "<br><br>";
        }
      }
    }

  //
  // Open the first database
  //
  $DB = bw_databases($link, 0);
  bw_open($link, $DB["dsn"]);

  //
  // Query the first column in the first table
  //
  $Table = bw_tables($link, 0);
  $Field = bw_fields($link, $Table["name"], 0);
  $cmd = sprintf("find [%s]:*", $Field["name"]);

  bw_query($link, $Table["name"], $cmd);  //
  $lim = bw_hitcount($link, $Table["name"]);
  printf("<br>Records found: %d<br>\r\n", $lim);

  //
  // Fetch the first 20 tuples of the result
  //
  if ($lim  > 20)
    $lim = 20;
  bw_set_fetch_size($link, 20);
  bw_set_select_cols($link, "*");

  echo "<table border=1 font=Arial size=2>\r\n";

  //
  // Tuple loop
  //
  for ($r = 0; $r < $lim; $r++)
    {
    bw_moveto($link, $Table["name"], $r);
    if ($r == 0)
      {
      //
      // Print column headers
      //
      echo "<tr>\r\n";
      for ($i = 0; $i < bw_fields_count($link, "@"); $i++)
        {
        $field = bw_fields($link, "@", $i);
          printf("<td><b>%s</b></td>\r\n", $field["name"]);
        }
      echo "</tr>\r\n";
      }

    //
    // Print column data
    //
    echo "<tr>\r\n";
    for ($i = 0; $i < bw_fields_count($link, "@"); $i++)
      {
      $field = bw_fields($link, "@", $i);
      printf("<td>%s</td>\r\n", $field["value"]);
      }
    echo "</tr>\r\n";
    }

  echo "</table>";
```

```
  echo "Calling disconnect";
  bw_disconnect($link, 1);

?>

  </font>
</body>
</html>
```

Chapter 9
Boolware xml client

This client contains five different functions; *BCXmlOpen*, *BCXmlVersion*, *BCXmlRequest*, *BCXmlGetErrorMsg* and *BCXmlClose*, defined in the file **xmlclient.h**

It is important to be familiar with the Boolware xml-request and xml-response elements on how to build XML requests and interpretate the responses.

See Chapter 3 "XML API" for all details about XML-Requests and Responses.

BCXmlOpen()

BWHANDLE **BCXmlOpen**(**char** *srv, **char** *sessName)

Creates and open a connection to Boolware resided on the server specified with the parameter srv and a session name.

Parameters:

char *srv	the socket address to Boolware [IN]
char *sessName	a session name, optional [IN]

This function establish a connection to a Boolware server resided on the server with specified IP-address. The supplied session name will be the name of the session in Boolware, if empty an unique session name is generated by Boolware server. The session name can be obtained from all XML-responses later on, enclosed by element tag <session>session name</session>.

This function returns, upon success, a handle to an internal memory object that shall be supplied in all other routines associated with this particular connect.

Return:

!= 0	a handle to an internal object that should be supplied to other functions for this session. Referred in this document as the connection handle and the variable bwHandle
0	upon error (NOTE; return value of Zero indicate an error)

Example:

Connect to a Boolware on a server located at IP-address 192.168.1.100, and no session name supplied.

```
BWHANDLE bwHandle = 0;

if((bwHandle = BCXmlOpen("192.168.1.100","")) == 0)
     …handle error
else
     printf("A connection to Boolware is established OK\n");
```

BCXmlReopen()

BWHANDLE **BCXmlReopen**(char *srv, int sessName)

Creates a connection to Boolware on the requested server, srv, with a specified session name, sessName.

Parameters:

char *srv	IP-address to Boolware [IN]
char *sessName	name of session to be re-connected [IN]

This function re-establishes a connection to Boolware on a specified IP-address with a specified session name.

The function returns a handle to an internal memory block which should be used by the session when calling all other functions.

Return:

!= 0	handle to an internal memory block to be used by the session when calling other functions. This handle will be referred to as the session handle with the variable name bwHandle.
0	when error (NOTE; return value zero is an error)

Example:

Re-establish the session "Bob" to Boolware at IP-address 192.168.1.100.

```
BWHANDLE bwHandle;

if((bwHandle = BCXmlReopen("192.168.1.100", "Bob")) == 0)
    ... handle error
else
    printf("A connection to Boolware is established OK\n");
```

BCXmlVersion()

Int **BCXmlVersion**(char *buff, int buffsz)

Obtain a printable version string of the BCXmlClient.

Parameters:

char *buff	output buffer to store version info
int buffsz	size of output buffer in bytes

Example:
Get the version of the BCXmlClient in printable format.

```
char buffer[256];  /* Response buffer */
int rc;
rc = BCXmlVersion(buffer, sizeof(buffer));
printf("Version: %s\n", buffer);
```

BCXmlRequest()

int **BCXmlRequest**(BWHANDLE bwHandle, char *req, char *resp, int respsz, int httpHeader)

Send a xml-request to Boolware and receive a xml-response from Boolware

Parameters:

BWHANDLE bwHandle	client connection handle [IN]
char *req	buffer containing the xml-request [IN] if set to NULL no request will be sent
char *resp	buffer to store the response from Boolware server [OUT] if set to NULL no response will be copied
int respsz	size of the response buffer in bytes [IN]
int httpHeader	if set a http header will be produced [IN]

This function shall be used to send xml-requests and receive xml-responses from Boolware.

Returns:

>= 0	the total size of the response
< 0	communication error to Boolware or memory allocation error if return value eq to BEBUFFERTOOSMALL response do not fit in supplied buffer

The option to set the out buffer, resp, to NULL is done to be able to obtain the size of the response. The response can be quite large, many KB of data and a buffer large enough must be allocated by the caller before obtaining the response. The response is stored in the client after a request is sent to the Boolware server and can be fetched at a second call to BCXmlRequest by setting the parameter req to NULL. The client will not call Boolware server one more time with the request if set to NULL. If calling this function twice to obtain the response part make sure to keep the value of httpHeader flag.
If the size of the response is known just call the BCXmlRequest with proper buffer settings.

If httpHeader is set and the req starts with a http header the xmlclient will save the header and prefix the XML response data with supplied header with a modified "Content-Length" message. If no header found a simple http header will be created containing "Content-Length" and "Content-Type".

Example 1:
Here you know that the response of our xml-request fits in buffer.

```
char buffer[4096];  /* Response buffer */
int rc;
if((rc = BCXmlRequest(bwHandle, req, buffer, 4096, 1)) > 0)
     printf("Respons: %s\n", buffer);
else
     …handle error;
```

Example 2:
Here you do not know the size of the response of our xml-request.

```
char *buffer;  /* Response buffer */
int rc, buffsz;
if((rc = BCXmlRequest(bwHandle, req, NULL, 0, 1)) > 0)
     {
     buffsz = rc;
     buffer = (char *)malloc(buffsz);
     if((rc = BCXmlRequest(bwHandle, NULL, buffer, buffersz, 1)) < 0)
          …handle error
     else
          printf("Response: %s\n",buffer);
     free(buffer); /* Free allocated buffer */
     }
else
     …handle error;
```

BCXmlGetErrorMsg()

int **BWXmlGetErrorMsg**(BWHANDLE bwHandle, char *buff, int buffsz)

On error get information about the error.

Parameters:

BWHANDLE bwHandle	sessions handle [IN]
char *buff	buffer to store the error message [OUT]
int buffsz	size of the buffer in bytes [IN]

Example:
Get error message.

```
char buffer[256];  /* Response buffer */
int rc;
rc = BCXmlGetErrorMsg(bwHandle, buffer, sizeof(buffer));
printf("Error: %s\n", buffer);
```

BCXmlClose()

int BCXmlClose (BWHANDLE bwHandle, int logout)

Close the connection associated with handle from the function BCXmlOpen.

Parameters:

BWHANDLE bwHandle	sessions handle [IN]
int logout	terminate the session in Boolware [IN]

This function close the connection to Boolware server and free allocated memory for the session handler.

After call to this function the session handler is no longer valid.

Returns:

0	upon success
-1	upon error

Example 1:

Close the connection and free the memory hold by the session handle. The parameter logout is set to **0**; indicate that the session remains in the Boolware server and will be logged out by time.

```
BCXmlClose(bwHandle, 0);
bwHandle = 0;
```

Example 2:

Close the connection and free the memory hold by the session handle. The parameter logout is set to **f**; indicate that the session will be logged out from the Boolware server immediately.

```
BCXmlClose(bwHandle, 1);
bwHandle = 0;
```

Program example

Here is a program snippet. This snippet we assume that there is a database in the Boolware server containing company information. The name of the database is ‘Company’ and the name of the table is ‘CompanyInfo’.

Locate all companies resided in London and have more than 20 employees. We will fetch the first 10 companies and column data from columns ‘CompanyName’, ‘CompanyID’, ‘City’ and ‘ZipCode’. Maximum 50 bytes from each column.
We also want to prefix the replay with a http header.

```
int main(int argc, char **argv)
{
BWHANDLE bwHandle = 0;
int buffersz, rc;
char *buffer, msg[256];
char *req = " <?xml version=\"1.0\" encoding=\"iso-8859-1\"?> \
     <SoftboolXML_requests> \
     <SoftboolXML_request type=\"query\"> \
     <open_session name=\"\" queryhistory=\"0\"/> \
      <database name=\"Company\"/> \
      <table name=\"CompanyInfo\"/> \
      <query> FIND  City:london and \"Employees\":>20</query> \
      <response type=\"\" href=\"\" queryhistory=\"1\"> \
      <sort expression=\"\"/> \
      <records from=\"1\" count=\"10\" maxchars=\"50\"> \
      <field name=\"CompanyName\"/> \
      <field name=\"CompanyID\"/> \
      <field name=\"City\"/> \
      <field name=\"ZipCode\"/> \
      </records> \
      </response> \
     </SoftboolXML_request> \
     </SoftboolXML_requests>";

if((bwHandle = BCXmlOpen("192.168.1.103", "")) != 0)
     {
     if((rc = BCXmlRequest(bwHandle, req, NULL, 0, 1)) > 0)
           {
           buffersz = rc;
           buffer = (char*)malloc(buffersz);
           if((rc = BCXmlRequest(bwHandle, NULL, buffer, buffersz, 1)) > 0)
                 printf("%s", buffer);
           }
     }
else
     {
     // Get error message
     BCXmlGetErrorMsg(bwHandle, msg, sizeof(msg));
     printf("Error:%s\n", msg);
     }

// Close connection
BCXmlClose(bwHandle, 1);
Return 0;
}
```

The replay from this command is:

```
Content-Length: 2374
Content-Type: text/html

<?xml version="1.0" encoding="iso-8859-1" ?>
<SoftboolXML_responses>
<SoftboolXML_response type="query" error_code="0" flowexit="">
<session>0000006767</session>
<records total="28" from="1" to="10" rank="no rank">
<record score="1.000">
<field name="CompanyName">Company Alpha Ltd</field>
<field name="CompanyID">111111</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
<record score="1.000">
<field name="CompanyName">Company Bravo Ltd</field>
<field name="CompanyID">222222</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
<record score="1.000">
<field name="CompanyName">Company Charlie Ltd</field>
<field name="CompanyID">333333</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
<record score="1.000">
<field name="CompanyName">Company Delta Ltd</field>
<field name="CompanyID">4444444</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
```

```
<record score="1.000">
<field name="CompanyName">Company Echo Ltd</field>
<field name="CompanyID">5555555</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
<record score="1.000">
<field name="CompanyName">Company Foxtrot Ltd</field>
<field name="CompanyID">6666666</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
<record score="1.000">
<field name="CompanyName">Company Golf Ltd</field>
<field name="CompanyID">77777777</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
<record score="1.000">
<field name="CompanyName">Company Hotel Ltd</field>
<field name="CompanyID">88888888</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
<record score="1.000">
<field name="CompanyName">Company India Ltd</field>
<field name="CompanyID">9999999</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
<record score="1.000">
<field name="CompanyName">Company Juliette Ltd</field>
<field name="CompanyID">00000000</field>
<field name="City">London</field>
<field name="ZipCode">21228</field>
</record>
</records>
<queryhistory total="0">
</queryhistory>
</SoftboolXML_response>
</SoftboolXML_responses>
```

Error handling

The xmlclient can return different types of errors and all of these are defined in the file **xmlclient.h**. All errors returned are critical except the **BEBUFFERTOOSMALL** error.
This error can be received from the function BCXmlRequest() and means that the reply buffer provided is not large enough to hold the entire response.

Other errors and warnings that belongs to the xml request is invoked in the xml response. The element attribute '*error_code'* is positive for warnings and negative for errors:

```
<SoftboolXML_response type="query" error_code="0" flowexit="">
```

and there is a message provided within the element <error> or <warning> depending on the error_code. Warning and error codes are declared in the file **softbool.h**.

Example:
Query syntax error in the query; a right parenthesis is missing.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<SoftboolXML_responses>
<SoftboolXML_response type="query" error_code="-103" flowexit="" >
<session>0000000002</session>
```

```
<error>Right parenthesis expected</error>
</SoftboolXML_response>
</SoftboolXML_responses>
```

Example:
A warning; extra synonyms are used during the query when search for: London

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<SoftboolXML_responses>
<SoftboolXML_response type="query" error_code="103" flowexit="">
<session>0000000005</session>
<warning warning_code="103">The following Synonyms have been searched for:
LONDON, CHELSEA, WIMBLEDON</warning>
…..
</SoftboolXML_response>
</SoftboolXML_responses>
```

Chapter 10
Plugins

This chapter describes how to implement a plugin for Boolware.

General

Plugins are external components that can be used to customize certain functions in Boolware.

A plugin for Boolware should be written as a DLL (or .so on Linux) and must support callbacks.

Boolware is a multi-threaded, thread safe program. This means that several independent execution paths are active at the same time, for example fifty users may post queries at the same time. This poses some simple rules that the plugin must comply with to be thread safe. The most basic rule is that a plugin **may not use global variables**, at least not without taking care in synchronizing access to these using locks. A plugin may use the 'usr1' pointer supplied by Boolware, in order to store thread data.

NOTE: programming errors in a plugin can make Boolware instable. This is unavoidable due to the fact that a plugin is implemented as a shared library, and hence executes in the same address space as the main process Boolware itself. This means that high demands are posed on each plugin. Code with care, and there will be no problems.

Softbool guarantees the quality of plugins developed by Softbool in-house, but cannot take responsibility for problems caused by plugins developed outside Softbool.

Currently there are four different types of plugins in Boolware:

e01 – custom indexing
e02 – custom scoring
e03 – custom phonetic
e04 – dynamic ranking of search results

Registering plugins

All plugins (.dll / .so) must be in the same directory as Boolware server. NOTE that Boolware must be restarted when a plugin is added to or removed from this directory.

Custom indexing (e01)

The name of a custom indexing function should be: e01.name.dll/so.
Example: e01.ownterms.dll for Windows.

In the Boolware Manager you can choose among existing functions. You could use them on Database-, Table- and Column level. See also the Help section in Boolware Manager.

There is a configuration file for each type of plugin (E01, E02, E03 and E04). When a plugin is selected, a section with the name of the plugin is created in the configuration file corresponding to its type.

Example: if the plugin "e01.split.dll" has been chosen for the column Name in the table Company, a conf-file with the name "db.Company.Name.e01.conf" will be created. The content of this file will be:

```
[hooks]
1=Split

[Split]
OnlySearch=0
```

More than one plugin can be selected for a single column:

Example: if the plugin "e01.shrink.dll" has been chosen for the same column Name in the table Company. A new section in the same conf-file "shrink" will be created:

```
[hooks]
1=Split
2=Shrink

[Split]
OnlySearch=0

[Shrink]
Seps=& and og och et
Before=1
After=1
OnlySearch=0
```

Parameters for plugins

If a plugin needs parameters (i.e. file names, paths etc.), there is support for editing the Boolware .conf file using Boolware Manager.

Parameters are stored in .conf file associated with a column. The names of the parameter files are controlled automatically by Boolware. Parameters for a certain plugin is stored in a section with the same name as the plugin itself. See above (Split and Shrink).

Custom scoring (e02)

The name of a custom scoring function should be: e02.name.dll/so.
Example: e02.ownscoring.dll for Windows.

The custom scoring will be called if the element <scoring> contains the attribute custom="name of the custom scoring module".

Custom phonetic (e03)

The name of a custom phonetic function should be: e03.name.dll/so.
Example: e03.ownphonetic.dll for Windows.

Using Boolware Manager, you select which fields that should use the custom phonetic function.

The custom phonetic function is called when the subcommand *sound* is used on a field that has been configured with Boolware Manager to use the custom phonetic plugin.

Custom ranking (e04)

The name of a custom ranking plugin should be: e04.name.dll/so.
Example: e04.ownranking.dll for Windows.

The custom rank plugin is called when the attribute customrankexit="ownranking" is specified for a <resultset> or <sort> element.

Plugin API

Depending on the which type of plugin you are going to develop, different APIs are used. The e04 plugins use a newer API that is described at the end of this chapter.

Together with the Boolware distribution, complete source code examples are provided for two indexing plugins (e01), "Split" and "Shrink", and also one source code examples for a custom rank plugin (e04).

GetInfo()

This function is used for e01, e02 and e03 plugin modules. It is called by Boolware server when initializing the plugin. It supplies Boolware with a name and a description about what the plugin does, as well as which version of the plugin API it is coded for.

The verbal description is used by Boolware Manager when it is showing available plugins

Example:

```
/**
 * Returns info about this plugin back to Boolware.
 *
 * @param hostVer - I - Boolware version in MM.II.RR.CC format,
 *            where MM is Major version, II is Minor version,
 *            RR is Release number and CC is Correction number.
 * @param name – O – name of this plugin, for example "split"
 * @param nameSz – I – size of name area, in bytes
 * @param version – O – version of this plugin, in MM.II format
 * @param versionSz – I – size of version area, in bytes
 * @param info – O – A line of descriptive text
 * @param infoSz – I – size of info area, in bytes
 * @param param – Parameter description strings
 *
 * @return zero if OK, else module will not be used by Boolware.
 */

static char *paramDef[] = {NULL};

int GetInfo(char *hostVer, char *name, int nameSz,
                           char *version, int versionSz,
                           char *info, int infoSz,
                           char *param[])
{
   // Save Boolware version, return name, version and info
   safecpy(bwVersion, hostVer, sizeof(hostVersion));
   safecpy(name, "Split", nameSz);
   safecpy(version, "01.01", versionSz);
   safecpy(info, "Splits letters and digits, for example SL500 " \
                 "becomes two words: SL 500", infoSz);
   *param = (char*) paramDef;
   return 0;
}
```

Execute()

This function is used by plugins that implements custom indexing (e01). The function is called by Boolware server to select terms from a text, to associate with the current tuple.

Parameters:

self	An internal instance handle that must be passed back to Boolware at callbacks. See further info about callbacks in the 'cb' parameter.
field	A pointer to all information regarding this field such as name, data type, selected indexing options and so forth. See further down for detailed information.
docNo	Boolware's internal ID for this tuple. Must be passed back to Boolware by the plugin. when calling back to OutputTerm (see parameter 'cb').
text	Text to be indexed; in UNICODE UCS2 format.
length	Length of text, in characters, not bytes.
cb	Pointer to a control block which contains callback functions that can be invoked to normalize the buffer and deliver terms back to the Boolware server. The control block also includes variables such as *cb->mode* that specifies whether the function is called from indexing or searching.

The Boolware server can specify which types of index terms it wants the function to generate. It does this by setting the requested index types in the parameter *cb->fFlags*. If this parameter is not zero, then this function should generate index terms for the requested index type or no terms at all.

If the parameter *cb->fFlags* is zero, then the parameter *field->fFlags* contains the control governing the type of index terms that the function can generate.

To test the index types that are set in *fFlags* use constants that are defined in the file *customindex.h* subsection "Field Flags", such kIndexWord and kIndexString.

The following briefly describes the callback functions and variables that are available through the control block.

```
enum enumMode { INDEX, SEARCH } mode;
```

Reveals if the plugin is called from indexing or search mode.

```
int fHookNo;
```

This plugins sequence number, > 0.

```
unsigned fFlags;
```

Contains zero - if the flags from the descriptor (word, string etc.) should be used else the flags set by the calling module should be used

```
char fParamFile[256];
```

The parameter file name.

```
int (*CharType)(void *self, UCS2 ch);
```

Classifies the char ‘ch’ – see constants SB_CT… further down.

```
int (*GetChar)(void *self, int type, UCS2 ch);
```

Neutralize the character ‘ch’ via Boolware character tables. The parameter ’type’ indicates which character table to use: standard neutralize or phonetic neutralize. If ‘type’ has another value than the above specified, no neutralizing will be performed and the specified character ‘ch’ will be returned.

```
int (*Normalize)(void *self, int mode, UCS2 *dest, int destSz, UCS2 *src, int
srcSz);
```

Normalizes a text.

Normalizing normally means that all lower case letters will be translated to upper case and all diacrits will be removed. The normalizing is controlled by character tables specified in the Boolware Manager.

‘mode’
0 – Standard neutralizing,
2 – Swedish phonetics,
3 – English phonetics,
4 – soundex,
5 – modified soundex,
6 – west European phonetics

If other value on ‘mode’ than specified above, -1 will be returned.

Note that the size of ‘dest’ could be greater than the size of ‘src’ since some characters generate two characters when doing phonetics.

```
void (*OutputTerm)(void *self, BW_FIELD *field,
            UCS2 *term, int charCnt,
            int docNo, int wordNo, int method);
```

Sends a term to Boolware. The task for the plugin is to produce search terms from a text. Each selected search term is sent back to Boolware using this callback.

Parameters:

self	Same ’self’ as Execute is called with (passed on)
field	Same ’field’ as Execute is called with (passed on)
term	Selected term, in UCS2 UNICODE format
charCnt	Length of ’term’, excluding any terminating NULL
docNo	Same ’docNo’ as Execute was called with (passed on)
wordNo	Not used in this version, always use 0 (zero)
method	This parameter consists of a bit pattern with the following meaning:

Bit 0 – 7	Selected index method for the word generated by the plugin, normally METHOD_WORD
Bit 8 – 15	Current hook number, i.e. callback->fHookNo
Bit 16 – 23	Boolean command between words: ’A’ = AND ’O’ = OR ’N’ = NOT ’X’ = XOR
Bit 24 – 31	Not in use but should be set to 0

Example: method = (((‘A’ << 8) | cb->fHookNo) << 8) | METHOD_WORD;

Indexing method of term (see METHOD_* constants) in the lower eight bits, logical
operator and exit sequence number.
The operator in bits 16-23, the hook number in bits 8-15 and the indexing
method in bits 0-7.
E.g
method = ((('A' << 8) | cb->fHookNo) << 8) | METHOD_WORD;

int (*GetField)(void *self, UCS2 *name, UCS2 *result, int charCnt);

Reads a column from the current tuple.

Parameters:

self	Same 'self' as Execute is called with (passed on)
name	Name of desired column value
result	Buffer where Boolware will place the result, as a zero string
charCnt	Size of 'result', in characters

ExecCMD()

This function is used by plugins that implements custom scoring (e02) and is called by Boolware server to score tuples (custom scoring).

Parameters are:

cmd	Message for the plugin, see following description of BHC_ codes.
p1	Parameter 1, a character pointer. (may be NULL).
p2	Parameter 2, a character pointer. (may be NULL).
cb	Supplied functions for example character normalization.
ExecCMD	returns an integer return code interpreted differently for each cmd.
BHC_INIT	called by Boolware to initialize the plugin. Boolware always sends a BHC_DONE when processing is done, to give the plugin an opportunity to release resources. Returns 0 if everything went fine, < 0 on error. Processing is ended on errors.
BHC_DONE	called by Boolware to give the plugin an opportunity to release resources. Boolware always sends a BHC_DONE when processing is done, regardless if BHC_INIT was successful or not.
BHC_XML	score a database tuple. The call contains two XML documents, p1 contains the query and any parameters, and p2 contains the database tuple.

P1:
```
<?xml version="1.0" encoding="UTF-8"?>
<qp>

<query flow="E02 Company Match">
<CompName>ibm schweiz</CompName>
</query>

<param>
  <field id="Company_Name">
```

```
    <rule id="Same tokens and same order" reduction ="0" weight="100"/>
  </field>
</param>

</qp>
```

P2:

```
<?xml version="1.0" encoding="UTF-8"?>
<tuple>
  <ID>448044383</ID>
  <Name>Contisa Sammelstiftung der Allianz</Name>
  <Street>Kirchstrasse 6</Street>
  <Zip>12345</Street>
  <City>Bern</City>
</tuple>

The callback structure for ExecCMD contains the following:

  int fVersion;      // 4
  char *fDSN,        // DSN name
       *fTableName, // Table name
       *fMsgPtr,     // Error messages
       fParamFile[256];  // Name of parameter file
  void *fContext,    // READ: plugin can use as ID for instance data
       *fTable,      // READ: SysTable pointer, if any
       *fField,      // READ: Field info pointer, if any
       *fTuple,      // READ: Tuple pointer if any
       *fData1,      // READ/WRITE
       *fData2;      // READ/WRITE
  unsigned fFlags;  // READ/WRITE
  double fDouble;   // READ/WRITE (old score)

CharType(cb, field, ch);
```

Classifies a char. See constants SB_CT further back in this text. A field name can be used (e.g. "Name"), since different fields can have different character definitions. Field name should be given as Latin-1 (not UTF-8).

```
Normalize(cb, field, method, dest, destSz, src, srcSz)
```

Normalizes a text buffer. Normalization means converting characters from lower to upper, and to eliminate most diacritics (accents). Normalization is controlled by char tables defined in Boolware Manager and accessed depending on which field is given.

method 0 - normalize (caps etc.), 2 - Swedish sound, 3 - European sound, 4 - English sound, 5 - Soundex, 6 - Modified Soundex.

```
GetFieldInfo(cb, name);
```

Returns info about a certain field. The info is returned as a pointer to a BW_FIELD structure, see descriptions of structures further back. If a field cannot be found, NULL is returned,

Custom ranking (e04)

Plugins of this type implement the following functions:

```
int Cleanup(void *userdata);
const char *Description();
const char *DefaultConfiguration();
int Execute(ExecuteParameters *parameters);
int Initialize(InitParameters *parameters);
int InterfaceVersion();
int Validate(ValidateParameters *parameters);
const char *Version();
```

For more information about these functions and their parameters, see the file: e04.customrank.h that defines the plugin interface for custom ranking.

Questions and answers

Q: How do I determine if the plugin is called from search or indexing?

A: Use the "mode" field in the "cb" block:

```
  // Called from indexing?
  if (cb->mode == SWordextractCB::INDEX)
    return;
```

Q: What is a "term type", and what is its use?

A: Boolware supports several different indexing methods, and has the ability to differentiate between these. This is so that an application can be able to say "now I want to search phonetically" or "now I just want to search stemmed words". If Boolware didn’t differentiate the terms in its index, but mixed standard terms with phonetic and stemmed terms, it would no longer be possible to make an exact search. You would receive unexpected results.

A plugin programmer uses the term type when calling back to Boolware using OutputTerm. The most common is "word" (METHOD_WORD), but other of course exist.

When Boolware calls Execute, it says in the "field.flags" which term types are desired. "field.flags" is directly connected with the selected check boxes in Boolware Manager for this field, and is not the same as the term type. If "field.flags" contains kIndexWord, it means that the caller wants Execute to generate "words", if it is kIndexString then Execute should generate strings (METHOD_STRING). Refer to the code example.

Q: What does "normalization" mean, in Boolware lingo?

A: Often (but not always) you want to equal small and capitalized letter when searching (case insensitivity), and also equal characters with and without accents (é and e, ñ and n etc.). This is done in Boolware using a translate table controlled from Boolware Manager.

It is up to the plugin itself to decide if it wants to normalize the text or not.

Appendix 1
Constants and Structures

In this chapter the records used when transmitting information between the Boolware Client and the Application.

Introduction

In most cases the information between Softbool Client and the Application and vice versa are sent using strings and other standard data types such as integers. In some cases, however, it is more practical to gather the information in a record. Below is a detailed description on the information sent via records.

Some information consists of flags and constants and these are also described in this chapter.

Constants

Below all constants are described. This is a general description on how and when to use the different constants.

As the constants could be slightly modified during the development it is highly recommended to consult the following include files - that are part of the delivery - to get the latest version of the constants: sbTypes.h, softbool.h, boolwareclient.h and xmlclient.h.

Constants to describe the presentation order

BNORANK	0	Same order as in the data source
BOCCRANK	1	Order occurrence
BFREQRANK	2	Order frequency
BSIMRANK	3	Order similarity
BSORTRANK	4	Sort (ascending)
BSORTDRANK	5	Sort (descending)
BWEIGHTOCCRANK	6	Weighted order occurrence
BWEIGHTFREQRANK	7	Weighted order frequency
BEACHTERMOCCRANK	8	Rank on specified Term
BCUSTOMRANK	9	Rank by Custom
BFUZZYRANK	10	Rank by fuzzy

When presenting the result you could get the records in any of the above order.
If no order specified (BNORANK) the records will be presented in the same order they have in the data source.

When order is occurrence (BOCCRANK) or frequency (BFREQRANK) it reflects the occurrences of the search terms within the records. Search terms from all queries since the last FIND command are included in the calculation. Frequency means occurrences divided by total number of words in the corresponding record.

Similarity (BSIMRANK) means the score that is calculated when comparing the records against the text used when performing the similarity search. The score will be a statistical value between 0 and 1 in the form 0.xxx.

Sort ascending (BSORTRANK) or descending (BSORTDRANK) ranks the records in specified order. If more than one Column is part of the sort, you could set individual sortorder on each Column.
Weighted order is a special case of occurrence and frequency. When searching you could specify a weight on the search term which will be multiplied by the occurrence before the score is calculated. In this way you could order records depending on the importance of the search terms.

Constants for index methods used when searching and presenting index terms

BDEFAULTTYPE	0	Use default value
BWORDTYPE	1	Word
BSTRINGTYPE	2	String
BSTEMTYPE	3	Stemming
BPHONETICTYPE	4	Phonetic
BREVERSETYPE	5	Reversed order

As you could specify more than one index method on each Column it is important to tell the system which type of term to search for or fetch from the index.

If the default value (BDEFAULTTYPE) is specified Boolware Server will choose the most suitable type depending on how the Column is indexed. The system tries to find a suitable type in the following order: word, string, phonetic, stemmed and reversed order.

Word (BWORDTYPE) means that the system will search for words when a query is performed. Only words will be fetched when the index is examined.

If string (BSTRINGTYPE) is specified only strings will be searched for and presented when fetching index terms.

If a Column is indexed as stemmed (BSTEMTYPE) all words will be converted to their stems. The words write, wrote, written, writing etc. will all be converted to write.

Phonetic coded words are searched and fetched by help of the phonetic index method (BPHONETICTYPE).

To be able to search for words that are left hand truncated (*ball) as fast as for normal truncated words (car*) you should use this indexing method (BREVERSETYPE).

Constants used for Column attributes and types in variable flags

BCOL_STRING	0x00000001	String
BCOL_WORD	0x00000002	Word
BCOL_PHONETIC	0x00000004	Phonetic
BCOL_PROXIMITY	0x00000008	Proximity
BCOL_SIMILARITY	0x00000010	Similarity
BCOL_LEFTTRUNC	0x00000020	Left truncation
BCOL_PROXLINE	0x00000040	Proximity line
BCOL_COMPRESS	0x00000080	Compressed
BCOL_PERMUTATE	0x00000100	Permutated
BCOL_STEMMED	0x00000200	Stemming
BCOL_CLUSTERED	0x00000400	Clustered
BCOL_RANKING	0x00000800	Ranking

BCOL_FIELDSEARCH	0x00010000	Field search
BCOL_FREETEXT	0x00020000	Free-text search
BCOL_ALIAS	0x00040000	Non indexed alias field
BCOL_MARKUPTAGS	0x00100000	XML element
BCOL_STOPWORD	0x00200000	Stop word
BCOL_FOREIGNKEY	0x01000000	Foreign key
BCOL_DATAXML	0x02000000	XML field
BCOL_SUBFIELD	0x04000000	Sub field (XML)
BCOL_VIRTUAL	0x08000000	Virtual field
BCOL_CASE	0x10000000	Case sensitive
BCOL_AUTOINCR	0x20000000	Automatic ID
BCOL_MEMMAPPED	0x40000000	Memory mapped
BCOL_FLDCHANGED	0x80000000	Presorted (Old description see BCOL_PRESORTED)
BCOL_PRESORTED	0x80000000	Presorted

Constants used for Column attributes and types in variable flags2

BCOL2_GEOPOSITION	0x00000001	Field is Geoposition; Lat or Long, WGS84 format
BCOL2_GEOMETERFORMAT	0x00000002	Field is Meter format like RT90
BCOL2_GEOMULTIPLE	0x00000004	Field contains both Long/Lat in same field
BCOL2_ASISWORD	0x00000010	Field is exact Word
BCOL2_ASISSTRING	0x00000020	Field is exact String
BCOL2_WITHINSTRING	0x00000080	Field is Within string
BCOL2_MIXEDALIAS	0x00000100	Field is Mixed Alias
BCOL2_POLYGONINDEXED	0x00000200	Field is Polygon indexed

The above constants gives the possible attributes and types for a Column. As a Column could contain more than one attribute the different attributes will be OR:ed together.
The constants could be divided into the following groups: indexing methods, field type, field contents and field status.

Indexing methods

String indexing (BCOL_STRING) means that the contents in the Column is indexed as a string. This means that the first 126 bytes of each line will be indexed as a string. Only leading and trailing spaces will be erased from the string. When searching only exact matches will be found.

When word indexing (BCOL_WORD) all words in the Column will be extracted and indexed. The maximum size of a word is 126 bytes.

Phonetic indexing (BCOL_PHONETIC) means that the extracted word will be coded reflecting the pronunciation rather than the spelling . This method is very useful when dealing with names.

Proximity indexing (BCOL_PROXIMITY) is only useful when the Column contains a lot of text where it is important to find combination of words when they appear close together.

Similarity search (BCOL_SIMILARITY) is used when you want to search for the entire contents rather than single words. It is most useful when the Column contains a lot of text.

Left hand truncation (BCOL_LEFTTRUNC) means that the Column is indexed for extremely fast response times when searching for words ended in the same way; FIND *ball.

If you want to search for words within the same line you should use this indexing method (BCOL_PROXLINE). It is much alike a string indexing but in this case you could specify the words in any order.

If a Column is marked for normalizing (BCOL_COMPRESS), all single characters separated by a special character (blank, dot etc.) will be handled in the same way. E.g. IBM could be specified in different ways: I&B&M, I.B.M etc., but using this indexing method all combinations will be coded as: IBM.

Another way to search for words in the same line in any order is to use the permutate indexing method (BCOL_PERMUTATE). This means that all words within the same line are permutated in all different combinations. This method requires a lot of disk space and BCOL_PROXLINE is recommended.

Sometimes you could get better relevance by only using the stems of the words (BCOL_STEMMED). This means that all verbs, adjectives, nouns etc. are converted to their basic form: written, wrote, writing etc. will be coded as write.

In certain columns where you have grouped information the search performance could be very much improved by using the clustered indexing method (BCOL_CLUSTERED).

If you want to rank the result depending on the occurrence of the search terms the columns should be marked with this constant (BCOL_RANKING).

Field types

Field search (BCOL_FIELDSEARCH) means, means that all words in this Column will be saved in a separate Index file. When searching only words within this Column (Field) will be searched.

Free text search (BCOL_FREETEXT) means that all words from Columns marked with this attribute will be saved in a special Index file. When searching free text all Columns with this attribute will be searched.

A Column could be index both for Field and Free text search and it is in the query you determine if you want to search only within this Column or all Columns indexed for free text.

There is another way to search several columns at the same time; you could specify an non indexed alias column (BCOL_ALIAS) that holds several columns. This column does not contain any data but only tells what columns to search.

Field contents

If a Column contains XML elements you could specify this constant (BCOL_MARKUPTAGS) to avoid XML elements to be stored as index terms.

If you do not want noise words to be included into your index you could mark the Column to ignore stop words (BCOL_STOPWORD). This means that all words that are specified in the stop word file will be ignored. For detailed information see Chapter "Stop words" in the manual

Operations Guide.

If the contents of the Column are foreign keys it will be marked by this constant (BCOL_FOREIGNKEY).
A text column that contains XML coded information is marked (BCOL_DATAXML). A XML Column could contain sub fields, which Boolware could handle as a normal Column.

A sub field which is contained within a XML marked Column
(BCOL_DATAXML) is marked by (BCOL_SUBFIELD). This sub field could have attributes as a Column and will be treated as a Column when searching. A sub field is not visible in the data source but is a part of the XML text Column (BCOL_DATAXML).

If the field type is virtual (BCOL_VIRTUAL), the field could contain different types of information. Independent of the contents the virtual fields are not part of the data source but is created temporarily in Boolware. An example of a virtual field is the calculate field that contains a formula and where the value from the calculation is saved

Field status

When the value of a Column is automatically incremented with a unique value by the data source it could not be affected by Boolware. Usually it is a Column that contains an identification (primary key) for the table. The Column is marked (BCOL_AUTOINCR).

Another way to make the searches even faster is to load parts of the index into memory. The indexes that are loaded into memory are marked (BCOL_MEMMAPPED).

Constants that describes the status of a database

During a normal use of Boolware the different databases could have different status. Below the different status are listed:

```
ONLINE     The database could be used "on-line"
LOADING    The database is being loaded
OFFLINE    The database is "off-line"
PENDING    The database is going to be loaded
READONLY   The database could not be updated
```

By using the *BCGetDatabaseInfo* you could get the status for the current database.
When the database is ONLINE all operations could be performed against the database. This is the normal status when using Boolware.

When a database should be loaded or re-loaded it is not possible to use the database for searching. The status of the database will be set to LOADING and the only operation that could be performed is to load the database.

If you do not want anybody to use the database it should be set to OFFLINE. This status is also set by the system if the database is corrupt.

When a database should be loaded a certain amount of resources are required from the system: a minimum amount of memory must be available. If there is not enough resources when the loading of a database should start the status of the database is set to PENDING. The loading of the database will wait until sufficient resources are available and will then start the loading and change the status to LOADING.

During certain operations you do not want updates to affect the database but it is still usable for searching; when reorganizing the database for example. In this case the status of the database is set to READONLY. As soon as the current operation (reorganization) is finished the status is reset to ONLINE and all updates that have been made will automatically be updating the Boolware indexes.

Constants that describes the status of a Table

```
AVAILABLE       The Table is available
NOTAVAILABLE    The Table is not available
```

Constants for the database

The data types that are sent from Boolware to the application are mapped as follows:
Standard data types:

```
BSQL_UNKNOWN_TYPE    0
BSQL_CHAR            1
BSQL_NUMERIC         2
BSQL_DECIMAL         3
BSQL_INTEGER         4
BSQL_SMALLINT        5
BSQL_FLOAT           6
BSQL_REAL            7
BSQL_DOUBLE          8
BSQL_DATETIME        9
BSQL_INTERVAL       10
BSQL_TIMESTAMP      11
BSQL_VARCHAR        12
```

Different data types for date and time:

```
BSQL_TYPE_DATE      91
BSQL_TYPE_TIME      92
BSQL_TYPE_TIMESTAMP 93
```

Extended data types:

```
BSQL_LONGVARCHAR    (-1)
BSQL_BINARY         (-2)
BSQL_VARBINARY      (-3)
BSQL_LONGVARBINARY  (-4)
BSQL_BIGINT         (-5)
BSQL_TINYINT        (-6)
BSQL_BIT            (-7)
BSQL_GUID           (-11)
```

Structures

In this part the used structures (records) are described. This is a general description on how and when to use the different structures.

As the structures could be slightly modified during the development it is highly recommended to consult the following include files - that are part of the delivery - to get the latest version of the structures: sbTypes.h, softbool.h, boolwareclient.h and xmlclient.h.

Information on a Database

```
typedef struct
   {
   char  dbName[128];      Name of the database
   char  descrName[256];   Description
   char  dsnName[128];     Name in data source
   int32 status;           Status
   } BCDatabaseInfo_t;
```

This record is used to pass information on the current database. The name of the database specified in *dbName* is used in Boolware. A descriptive text could be specified in the field *descrName*. The name of the database in the data source is stored in *dsnName*, while *status* contains one of the following values: ONLINE, LOADING, OFFLINE, PENDING or READONLY. These values are described above in the part "Constants that describes the status of the database".

Note. The database identification field that should be used by the function Attach() is the *dsnName*.

Information on a Table

```
typedef struct
   {
   char  tabName[128];     Name of the Table
   int32 flags;            Flags
   int32 hitCnt;           Current result
   int32 recordCnt;        Totalt no. of rows
   } BCTableInfo_t;
```

Information on a Table is passed using the above record. In *tabName* the name of the current Table is stored. The element *flags* could for the moment only have two values: AVAILABLE, NOTAVAILABLE (see part "Constants that describes the status of a Table" above). In the variable *hitCnt* the latest search result is stored and *recordCnt* contains the total number of rows in the current Table.

Information on a Column

```
typedef struct
   {
   char  colName[128];     Name of the Column
   int32 flags;            Attributes
   int32 dataSize;         Size of the field
   int16 dataType;         Data type ODBC
   int16 PKeySeq;          PK sequence no.
   int16 decPart;          No. of decimals
   } BCColumnInfo_t;
```

In this record is the information on the Column passed from function *BCGetColumnInfo*.
The name of the Column is stored in the variable *colName*. In the variable *flags* all attributes for this Column is stored (see part above "Constants used for Column attributes and types in variable flags"). In the variable *dataSize* the allocated size of the Column is stored. The variable *dataType* contains the data type the Column has in the data source. If the current Column is part of the Primary Key the sequence number will be store in the *PKeySeq* variable; 0 (zero) means that the Column is not part of the Primary Key. If values in this Column could contain decimals the number of decimals is stored in the variable *decPart*.

```
typedef struct
   {
   char  colName[256];     Name of the Column
   int32 flags;            Field attributes
   int32 flags2;           More field attributes
   int32 dataSize;         Size of the field
   int16 dataType;         Data type ODBC
   int16 PKeySeq;          PK sequence no.
   int16 decPart;          No. of decimals
   } BCColumnInfoEx_t;
```

In this record is the information on the Column passed from function *BCGetColumnInfoEx*.

The name of the Column is stored in the variable *colName*. In the variable *flags* all attributes for this Column is stored (see part above "Constants used for Column attributes and types in variable flags"). Int the variable *flags2* extended field attributes are stored (see part above "Constants used for Column attributes and types in variable flags2"). In the variable *dataSize* the allocated size of the Column is stored. The variable *dataType* contains the data type the Column has in the data source. If the current Column is part of the Primary Key the sequence number will be store in the *PKeySeq* variable; 0 (zero) means that the Column is not part of the Primary Key. If values in this Column could contain decimals the number of decimals is stored in the variable *decPart*.

Information on data in a Column

```
typedef struct
   {
   char  *name;           The name of the Column
   char  *value;          Data
   int32 length;          Length of data
   int32 dataType;        Type of data
   } BCColData_t;
```

When data is fetched for a Column this record is used. The name of the Column is stored in *name* and identifies which Column the passed data belongs to. In the variable *value*, the current information is stored. The *length* that is specified in length does not include the ending null character. The data type from the data source is saved in *dataType* (see part above "Constants for the database").

Information on a row of data (result row)

```
typedef struct
   {
   float score;           Score
   int32 recordID;        Record ID
   int32 count;           Number of Columns
   BCColData_t *cols;     Data
   } BCRowData_t;
```

Usually you will fetch data from several Columns to build up a result row or to display the complete record. To do this use the above record *BCRowData_t*. The information from each Column is stored in the variable *cols*, which is described above ("Information on data in a Column"). After certain requests a row could contain a score which is used to order the rows. The score could mean different things: similarity, number of occurrences, frequency etc. The score is stored in the variable *score*. In the variable *rankMode* described in the part "Constants to describe the presentation order" above you could see the meaning of the score. A unique identification for Boolware is saved in *recordID* and it could be used when communicating with Boolware later on. The number of Columns that are sent will be stored in the variable *count*.

Information on an index term

```
typedef struct
   {
   int32 hits;            No. within result
   int32 totalHits;       Total number
   int32 termNo;          For the future
   int32 termType;        Indexing method
   char  term[128];       Index term
   } BCTerm_t;
```

When an application should present the index terms that are contained in a Column this record is used to pass the information. Two counters are available: *hits* and *totalHits*. In these counters the number of rows the term appears in. In *totalHits* the occurrence in the entire database is stored, while *hits* holds the occurrence in the current result. When a Column has been index with several attributes - e.g. word and string - the different terms are separated by specifying the indexing method in the variable *termType*. A detailed description on the different index methods could be found in the part "Constants for index methods used when searching and presenting index terms" above. The term will be stored in the variable *term*.

Information on Query History

```
typedef struct
   {
   int32 qHistoryRows;     Total number of queries
   int32 currQHistoryRow;  Current query
   } BCQHistoryInfo_t;
```

Sometimes you need a lot of queries to get the wanted result. In some cases you would like to "back" to a previous query and continue from that point. To be able to do this you should use the Query History. The information should be fetched in two steps: 1. Get number of queries that is saved in the Query History and 2. Fetch the queries.

The record *BCQHistoryInfo_t* is used to get the number of queries in the Query History and which is the current Query (could be another than the last if you have used the "back" command). The variable *qHistoryRows* specifies the total number of queries, while *currQHistoryRow* holds the current query.

Information on statistics

```
typedef struct
   {
   int32  cnt;       No. of items
   int32  modeCnt;   No. of mode
   double mode;      Most common value
   double sum;       The sum of all values
   double avg;       Arithmetic mean value
   double min;       Minimum value
   double max;       Maximum value
   double std;       Standard deviation
   double var;       Variance
   double median;    Median value
   double upper;     Upper limit for group
   double lower;     Lower limit for group
   } BCStatisticsInfo_t;
```

Statistics could be calculated for one numeric Column at a time (via XML you could get statistics on more than one numeric Column in one request). The statistics is calculated on the current result or on all records in the table, and only records that contain values are part of the calculation. The following values are calculated:

Number of records that are part of the calculation is saved in *cnt*. The most common value is stored in *mode* and the number of records containing the most common value is saved in *modeCnt*. The sum of all values could be fetched from *sum*, while the lowest and the highest values are stored in *min* and *max* respectively. The arithmetic mean value is stored in *avg*.

Variance and standard deviation are saved in *var* and *std* respectively. You could choose to get the upper and lower limits for the tertial, quartal, quintil etc. and these values are stored in *upper* and *lower* respectively. Finally the median value is saved in *median*.
If you use execute or XML you could get all the limit values from the specified group.

Appendix 2
Error messages

In this appendix all error codes and their corresponding error messages are specified. The messages could be fetch by using the BCGetErrorMsg API.

Introduction

To have a flexible error handling the error codes and the error messages are stored in external files which could be modified.

The files must be in the same directory as SoftboolSrv.exe and the name of the file must be SoftboolMsg. The file extension should be a language code (max 3 characters) e.g. en for English and sv for Swedish.

Two files are shipped along with the system: the English (SoftboolMsg.en) and the Swedish (SoftboolMsg.sv).

This make it possible to translate the message files to any language and change the extension to the proper language code.

Each line in the files is a complete message built up by a code and the message. The code is of two types: error code (negative) and warning code (positive).

The code and the message must be separated by at least one "white space". It is very important not to change the code as it is used by the Boolware system to identify the error.

In the listing below all messages do not fit in one line but in the file they must.

If there is no message file or if an error code has been removed, the error message will be printed in English.

If nothing is specified in the configuration file SoftboolSrv.ini the SoftboolMsg.en file will be used.

This means that you in the configuration file could change message file by specifying the proper language code in the element ’language’.

Example: Boolware should be used by a Norwegian company and all messages has been translated into Norwegian. The message file is called SoftboolMsg.no and is on the same directory as SoftboolSrv.exe. SoftboolSrv.ini should now be changed: language=no.

All Error and Warning messages are in the softbool.h include file which is part of the Boolware delivery.