The Wayback Machine - https://web.archive.org/web/20140311035504/http://www.ibm.com:80/developerworks/ibmi/library/i-omnifind/omn…

Searching Spool Files and IFS Stream Files

Utilizing OmniFind extensions

Nick Lawrence (ntl@us.ibm.com), Advisory Software Engineer, IBM
Jian Li (cdllij@cn.ibm.com), Staff Software Engineer, IBM

Summary: This article will explain how to use OmniFind Text Search Server for DB2 for i to index and search IFS Stream files and Spool Files. This provides search capabilities for IBM i objects; similar to what is commonly available using web search engines.

Date: 01 Jun 2011
Level: Introductory
Also available in: Chinese

Activity: 6318 views
Comments: 0 (View | Add comment - Sign in)

Rate this article

Overview

The IBM OmniFind Text Search Server product for DB2 for i is a no additional charge product that allows IBM i developers to index and search text documents (including rich text documents such as Microsoft Word, PDF, XML, etc) that are stored in a DB2 column. For example, if I have a CLOB column that contains short stories, and I want to retrieve the rows where the column has data that pertains to “big bad wolves”, I can use SQL built in functions to find those rows, even if the actual text contained in the row’s column is “the big bad wolf”. Furthermore, I can order the matching rows so that the most relevant documents are first in the result set.

Searching text data in a DB2 column has been a really great feature, but not all of the interesting text data on the system is stored in DB2. For example you may need to search a set of reports stored in spool files within an Output Queue. Or suppose you want to search IFS stream files that contain PDF data. Fortunately, starting in IBM i 7.1, a solution exists that allows us to index and search the text data associated with these IBM i objects.

For a more complete overview of OmniFind Text Search Server, reference this whitepaper. This article focuses its attention on the new enhancements for searching spool files and stream files in IFS.

Software Requirements

In order to use the functions described in this paper, it is necessary to order and install the OmniFind Text Search Server V1R2 product (5733-OMF) and to apply IBM i 7.1 PTF SI45696. In general, it is best practice to apply the latest IBM Database Group PTF for IBM i 7.1 on the system at the same time you load the PTF. In addition, the OmniFind product requires a few other products to be installed on the system. These required software products are documented in the OmniFind reference manual.

Supported Object Types and Text Attributes

IBM i objects typically have multiple attributes that are text. For example, an output queue has a text description associated with it – and also contains spool files with text in them. In order to allow IBM to extend support to different objects and text information, we choose to identify the text by both an object and an attribute of the object that contains the text. For example a specific spool file’s data in output queue NICK/QUEUE1, or the data in a stream file /home/nick/file1.txt.

Spool Files in an Output Queue

Spool files with SNA Character Stream (SCS) data contain printer output that is formatted as EBCDIC text data. Spool files created with the SCS format are commonly used for storing reports and job logs in an output queue so that they can be processed later by an application.

Although spool files exist within an output queue, they are usually selected using other criteria. For example the work spool file command (`WRKSPLF`) uses file name, job, spool file number, creation system, and creation timestamp to identify spool files.

OmniFind provides several approaches for identifying which SCS spool files to index that are based on the selection criteria used by other system APIs and CL commands.

Spool files that use formats other than SCS are less common and more difficult to extract text from, OmniFind does not support them.

More information on Spooled files can be found in the infocenter.

IFS Stream Files

The Integrated File System (IFS) provides a set of hierarchal file systems similar to what is used in a UNIX® environment. Each file system has its own unique attributes, such as case sensitivity, and allowed types of objects.

OmniFind supports indexing text data in stream file objects, using an IFS path to identify the file. A stream file object (*STMF) is a randomly accessible sequence of bytes, with no further structure imposed by the system. A simpler definition of a stream file is a PC file or a UNIX® file.

Steam files that contain plain text or rich text can be indexed and searched. As mentioned earlier, examples of rich text are PDF, PowerPoint, Lotus WordPro, Microsoft Word, etc.

It is important to understand that a stream file is different than a database file or source physical file; because database and source physical files are record oriented, and stream files are byte oriented. OmniFind does not support using an IFS path to index object types other than stream files at this time.

IFS provides an integrated structure over all information on an IBM i. It's a common misstatement to refer to an "IFS File". Since IFS includes everything, and there are a number of different kinds of files on the system, "IFS File" doesn't have a lot of meaning. Most of the time when the expression is used, what is really meant is a stream file which is being accessed using IFS. Our new procedures and documentation often use both the terms "IFS" and "Stream file" together to avoid confusion.

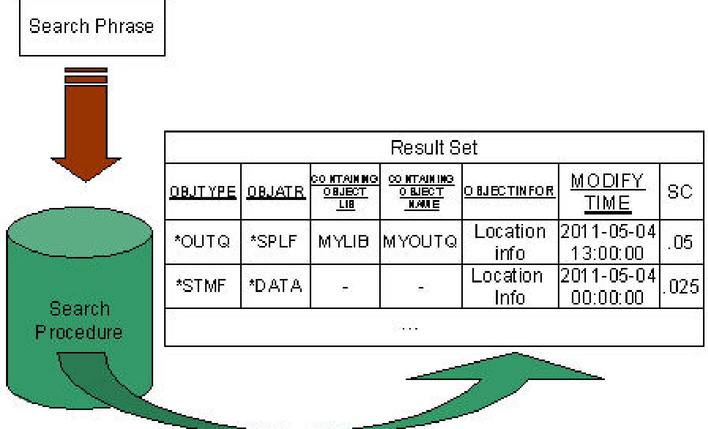
More information on steam files can be found in the infocenter.

Stored Procedures and Result Sets

The new administrative and search capabilities are provided via SQL stored procedures. It was extremely important to design interfaces that are simple to invoke from an application, and are flexible enough to allow IBM to potentially support additional object types in the future. Since many IBM i developers are familiar with stored procedures, SQL procedure calls are a natural solution.

SQL result sets provide a convenient mechanism for working with search results. When a search is performed, an SQL result set containing the results of the search is returned from the search procedure to the application. The result set approach allows DB2 to efficiently manage the storage on behalf of the application, and allows applications to retrieve and work with the results using existing DB2 interfaces such as JDBC, ODBC, or embedded SQL.

Figure 1: Search procedure result set



Creating a text search collection

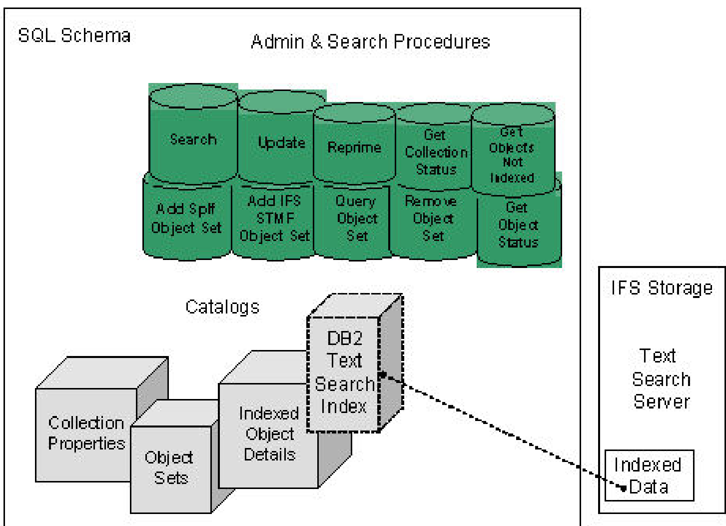
Before we can start indexing and searching, we need to create a text search collection. A text search collection is an SQL Schema that contains tables for tracking the indexed objects, and the SQL procedures for administering and searching the index.

Creating a text search collection is performed by invoking the `SYSPROC.SYSTS_CRTCOL` stored procedure.

```
CALL SYSPROC.SYSTS_CRTCOL('COLLECTION_FOR_NICK');
```

The above call will cause an SQL schema 'COLLECTION_FOR_NICK' to be created on the system. The schema contains all of the DB2 objects associated with the text search collection. This includes the catalogs and administrative procedures. The text search index data is stored outside of DB2 in the integrated file system, much like regular DB2 text search indexes.

Figure 2: Text search collection



The create collection procedure has several interesting options available for it that mirror the options used to create a DB2 text search index. The procedure's interface allows us to configure the update frequency, language, text format, and CCSID. The syntax for each of these options is described in detail in the OmniFind Extensions User Guide.

As an example, if I wanted to create my collection so that it updates once every day at midnight, I can create my text search collection this way:

```
CALL SYSPROC.SYSTS_CRTCOL('COLLECTION_FOR_NICK',
'UPDATE FREQUENCEY D(*) H(0) M(0)');
```

All of the procedures for administrating the collection are created in the SQL schema. This allows authority to be granted to other users so that they can search the text search collection, or to update it.

```
SET CURRENT SCHEMA COLLECTION_FOR_NICK;
GRANT EXECUTE ON PROCEDURE SEARCH(VARCHAR) TO DILBERT;
```

Setting the Path to the current collection

Because the administrative procedures are created inside of the text search collection (with the exception of create and drop collection, which are in SYSPROC), it is possible to use the SQL path to specify which text search collection we are working with.

```
SET CURRENT PATH COLLECTION_FOR_NICK;
```

This avoids the need to explicitly qualify each procedure call with a schema. For simplicity, the other examples in this article will assume the path is set to the value above.

Adding object sets

After creating the text search collection, OmniFind needs to know which objects to index. An object set defines a set of objects that will be included in the text index during the update process.

The OmniFind extensions support two types of object sets.

- Spool Files in an output queue
- IFS Stream Files in a specific IFS directory

Adding an object set does not update the index with the text data from that object set. OmniFind will update the index with the text from these objects during the next update, which will happen when the `UPDATE` stored procedure is called, or when a scheduled update runs.

Adding a spool file object set

The `ADD_SPLF_OBJECT_SET` procedure was created to add spool file object sets when the text search collection was created. There are actually a number of different versions of the procedure, and we can use any of them to select which spool files to index.

For example, this procedure will add a spool file object set for all spool files in an output queue NTL/MYOUTQ:

```
CALL ADD_SPLF_OBJECT_SET('NTL', 'MYOUTQ');
```

A slightly different invocation will add an object set for all spool files owned by NTL:

```
CALL ADD_SPLF_OBJECT_SET('', '', 'NTL');
```

In the above example, empty string is used for the output queue library and queue name to indicate that spool files from any output queue are considered for indexing. It is perfectly fine to have multiple object sets in the same collection, and also to have object sets where the same objects exist in multiple sets. More complex examples are possible using the version of the procedure that contains all supported parameters.

This example indexes spool files that were created with user data 'MYAPP' and were created in 2010:

```
CALL ADD_SPLF_OBJECT_SET('',              -- library
                         '',              -- queue name
                         '',              -- user name
                         '',              -- job name
                         '',              -- job user
                        '',               -- job number
                        'MYAPP',          -- user data
                 '2010-01-01T00:00:00',   -- start time
                 '2011-01-01T00:00:00'    -- end time
);
```

The OmniFind Extensions User Guide contains the complete syntax and description of each parameter.

Adding an IFS Stream file object set

Adding an IFS stream file object set is done with the `ADD_IFS_STMF_OBJECT_SET` procedure. Let's assume I want to add an object set for stream files in my home directory '/home/ntl'

```
CALL ADD_IFS_STMF_OBJECT_SET('/home/ntl');
```

If some of these files are rich text (Word, powerpoint, PDF, etc), then I would need to have created the collection with the FORMAT INSO option.

```
CALL SYSPROC.SYSTS_CRTCOL('COLLECTION_FOR_NICK',
                          'FORMAT INSO');
```

Format INSO means that the update process analyzes the document (INSide Out) to determine what kind of document data is being indexed. The extra processing will slow down the indexing process – but will provide more flexibility.

OmniFind will not implicitly include subdirectories in the object set, but it is possible to add any number of directories as unique object sets.

It is not a problem to have both spool file object sets, and IFS stream file object sets in the same collection. The result set from the search will contain the object type information, allowing an application to filter the results based on object type.

Update the Collection

The object sets included in the collection are not indexed until an update is performed. This can either be a scheduled update that was configured when calling `SYSPROC.SYSTS_CRTCOL`, or a manual update by calling the UPDATE stored procedure.

```
CALL UPDATE;
```

The update processing will determine which objects are new or changed on the system and index the text data for those objects. Updates after the initial update are incremental, unchanged objects that have already been indexed will not be indexed again. Some processing time is spent at the beginning of each update process determining which objects have been created, deleted, or changed.

After `UPDATE` has completed, the collection is now ready to be searched.

Search the collection

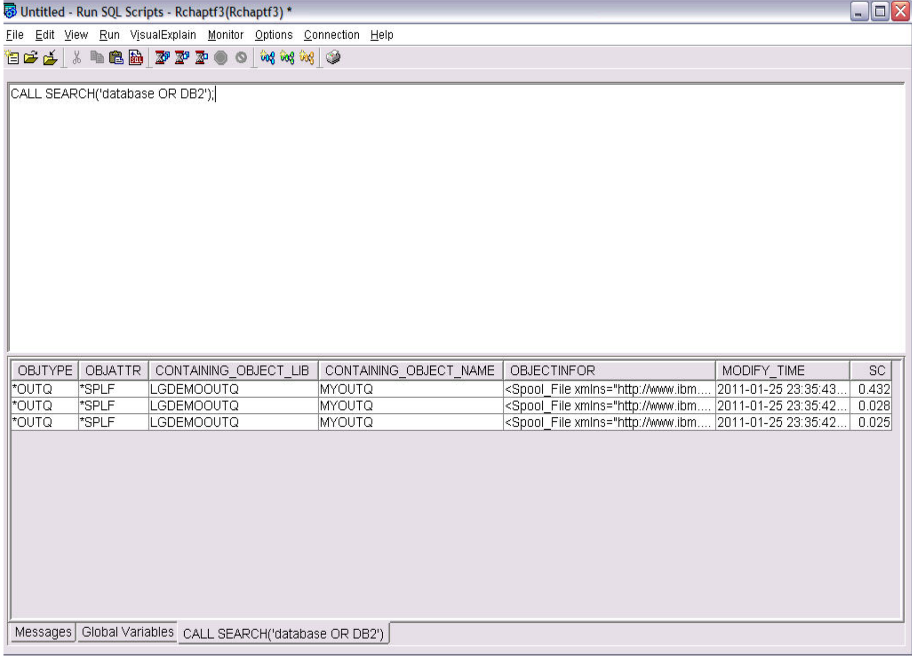
Searching the collection is done by calling the SEARCH stored procedure. The search expression works a lot like the web search syntax that most of us are familiar with. There are some advanced features that are documented in the IBM InfoCenter for those programmers that require more advanced capabilities – but the basic syntax is intuitive.

```
CALL SEARCH(‘database OR DB2’);
```

A result set is returned to the client application that contains the results, with the most relevant results being ordered first.

It’s easy to view the result set by calling SEARCH from IBM System i Navigator’s Run SQL Scripts. The result set will appear as a tab on the bottom of the window.

Figure 3: Navigator output



Applications can access the result set via any SQL interface that supports result sets (embedded SQL, JDBC, etc). We’ve provided an example that demonstrates how to connect to the data base using JDBC, execute a search, and use the retrieved data to locate the object.

Object Information

The most interesting column in the result set is the OBJINFOR column, which contains the location of the indexed object. XML was chosen as the data type for this column, both because of its flexible structure and also because of the wide variety of tools and parsers available for working with XML data.

XML is human readable, for example a matching spool file location could look like listing 1 shown below.

Listing 1: Spool file object information

```
<Spool_File
   xmlns=”http://www.ibm.com/xmlns/prod/db2textsearch/obj1”>
<job_name>QPADEV000C</job_name>
<job_user_name>USERA</job_user_name>
```

```
<job_number>009907</job_number>
<spool_file_name>DSXSVRALS</spool_file_name>
<spool_file_number>1</spool_file_number>
<job_system_name>ZD21BP1</job_system_name>
<create_date>1081027</create_date>
<create_time>035554</create_time>
</Spool_File>
```

A typical IFS stream file location might look like listing 2 shown below.

Listing 2: Stream file object information

```
<Stream_File
      xmlns=”http://www.ibm.com/xmlns/prod/db2textsearch/obj1”>
            <file_path>/home/usera/a.xml</file_path>
</Stream_File>
```

Processing the result set

The result set can contain rows with location values for both spool files and stream files. There is no requirement that the search be restricted to a particular object type.

DB2 for i does not provide the SQL XMLTABLE built in table function for converting XML data to relational data using XPath expressions, however tools for parsing and working with XML data are widely available for host languages. Our example shows one approach for dealing with XML data in Java.

The result set contains some other columns with helpful information

- modification time (`MODIFY_TIME`)
- object type (`OBJTYPE`)
- object attribute (`OBJATTR`)
- containing object library (`CONTAINING_OBJECT_LIB`)
- containing object name (`CONTAINING_OBJECT_NAME`)
- score value (`SC`)

These columns provide additional information about the search result and can be used by the application to filter results. They are described in detail in the OmniFind Extensions User Guide.

Additional Procedures

We’ve covered the basic procedure interfaces, but there are several other ones that are worth checking out in the OmniFind Extensions User Guide. Procedures exist to

- query the status of an object that has been indexed,
- query the status of the index,
- query the object sets,
- remove an object set, or
- drop a text search collection from the system.

Sample Application

Our example program performs a search of a text search collection. We choose to work with Java and JDBC because of their popularity and platform independence. There are similar approaches available for other languages and environments – but this one is relatively easy to explain.

Environment setup

Before we can run the program, we have to create the text search collection and update the data. We’ll set up the collection so that it updates every fifteen minutes. For this example, we want to have all stream files in the IFS directory `/home/ntl`, and all spool files in the output queue `NTL/MYOUTQ` included the index.

1. Create a text search collection.
   ```
   CALL SYSPROC.SYSTS_CRTCOL(
   ‘COLLECTION_FOR_NICK’, ‘UPDATE FREQUENCY D(*) H(*) M(0, 15, 30, 45)’);
   ```
2. Set the Path/Schema to current collection.
   ```
   SET CURRENT SCHEMA COLLECTION_FOR_NICK;
   SET CURRENT PATH COLLECTION_FOR_NICK;
   ```

3. Add spool file object set into the collection.
   `CALL ADD_SPLF_OBJECT_SET('NTL', 'MYOUTQ');`
4. Add IFS stream file object set into the collection.
   `CALL ADD_IFS_STMF_OBJECT_SET('/home/ntl');`
5. Update the collection.
   `CALL UPDATE;`

The update in step e is not strictly necessary since the collection will update on a regular basis, but forcing the update here means that we know the update has completed before issuing the search.

After the infrastructure is created, and the update process is complete, we can search keywords from the collection.

Call SEARCH Stored Procedure in Java

The Java statements in listing 3 show how to write java code to invoke SEARCH Stored Procedure. The key is to use the JDBC `java.sql.CallableStatement` class. A Boolean result is used to indicate that a result set has been returned.

Listing 3: Call SEARCH stored procedure in Java

```
// Variable keywords is the words you want to search for.
String searchSQL =
 "CALL COLLECTION_FOR_NICK.SEARCH('" + keywords + "')";

// Create CallableStatement for calling a stored procedure.
// Variable connection is a java.sql.Connection instance.
CallableStatement cstmt = connection.prepareCall(searchSQL);

// The execute method returns a Boolean to indicate that a
// result set has been returned
boolean isResultSetReturned = cstmt.execute();
```

Retrieve result set after calling SEARCH

After calling the `SEARCH` stored procedure, we can retrieve the result set from the `CallableStatement`. For each row, we will use the JDBC `ResultSet.getString()` function to get the value for the `OBJECTINFOR` column as a `String`. In this way, we can iterate over the result set, and get the XML value (as a string) for each object's location information.

Listing 4: Get result set after calling SEARCH

```
// Get the result set from CallableStatement object
 ResultSet rs = cstmt.getResultSet();

 // The ResultSet next method is used to iterate over
 // the rows of a ResultSet.
 // The next method must be called once before the
 // first data is available for viewing. As long as next
 // returns true, there is another row of data that
 // can be used.
 while (rs.next()) {
     String objectinfor = rs.getString("objectinfor");

    /** Process the object information using the XML string
       here **/
  }
```

Extract XML Object Information

The XML Object information contains the location information for the indexed object. In order to work with the object, we need to extract the data.

The `DocumentBuilder` class provides a way to build an XML document tree from a stream of bytes. Once the document tree is built from the object information, the value of specific elements can be extracted.

This example shows how to extract the "job_name" element's value from a spool file's object information.

The complete listing has a procedure `parseObjectInfo` that shows how code could be written to handle every object type's information, and dump the information to standard output.

Listing 5: Extract XML information

```
DocumentBuilderFactory factory =
           DocumentBuilderFactory.newInstance();
DocumentBuilder builder;
Document doc = null;

factory = DocumentBuilderFactory.newInstance();
builder = factory.newDocumentBuilder();
InputStream is = new

ByteArrayInputStream(objectinfor.getBytes());
doc = builder.parse(is);

String job_name =
          doc.getElementsByTagName("job_name").
item(0).getTextContent();
```

Complete program to do search

This listing contains the complete program. Of special interest is the `parseObjectInfo` function that we have created to handle the different supported object types. This function will dump the object's information to the standard output stream. With a few adjustments, the code here could perform other functions such as retrieving the text data from the object.

Listing 6: Complete listing for search program

```
////////////////////////////////////////////////////////////////////////////////
//
// ISVSearch example. This program uses the native JDBC driver for the
// Developer Kit for Java to call Omnifind stored procedure SEARCH to query
// specific key words
//
// Command syntax:
//     ISVSearch <collection name> <key words>
//
// Before calling this program, user should create collection, add
// object set to collection, update collecton first. These steps are
// used to make the collection searchable. Reference to user documentation
// for detail statements.
//
// This source is an example of how to invoke stored procedure SEARCH
// in java code and how to analyze the result.
//
////////////////////////////////////////////////////////////////////////////////

// Include any Java classes that are to be used. In this application,
// many classes from the java.sql package are used and the
// java.util.Properties class is also used as part of obtaining
// a connection to the database.
// java.io and javax.xml package are used
// to parse the XML column of returned result set of SEARCH stored procedure
import java.io.*;
import java.sql.*;
import java.util.Properties;
import javax.xml.parsers.*;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
//Create a public class to encapsulate the program.
public class ISVSearch {
      // The connection is a private variable of the object.
       private Connection connection = null;

       public static void main(String args[]) {
          // Create an object of type ISVSearch. This
          // is fundamental to object-oriented programming. Once
          // an object is created, call various methods on
          // that object to accomplish work.
          // In this case, calling the constructor for the object
          // creates a database connection that the other
          // methods use to do work against the database.
          ISVSearch isvSearch = new ISVSearch();

          // The search method is called next. This method
          // processes an Omnifind search statement against the collection
          // created before. The output of that query is output to standard
          // out for you to view.
```

```
            isvSearch.search(args[0], args[1]);

            // Finally, the cleanup method is called. This method
            // ensures that the database connection that the object has
            // been hanging on to is closed.
            isvSearch.cleanup();
        }

        public ISVSearch() {
            // Following statements were used to create a connection to
            // DB2 for i
            Properties properties = new Properties();
            properties.put("user", "omnifind");
            properties.put("password", "textsearch");

        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local",
                       properties);
        } catch (Exception e) {
             System.out.println("Caught exception: " + e.getMessage());
        }
   }

   /**
    * Search key words from specific collection.
    * The result will be printed to standard output.
    *
    * @param collection The collection name user created
    * @param keywords The key words user want to search
    */
   public void search(String collection, String keywords) {
        try {
            // Constructed the SQL statement to do search
            // The SQL statement should be like this
            // CALL <COLLECTIONNAME>.SEARCH('keywords')
            String searchSQL = "CALL " + collection + ".SEARCH('" + keywords
                       + "')";

            // Create CallableStatement for calling a stored procedure.
            CallableStatement cstmt = connection.prepareCall(searchSQL);

            // The execute method returns a boolean to indicate the form
            // of the first result
            boolean isResultSetReturned = cstmt.execute();

            // Check if there is ResultSet returned
            if (isResultSetReturned) {
                // GEt the result set from CallableStatement object
                ResultSet rs = cstmt.getResultSet();

               // The ResultSet next method is used to process the rows of a
               // ResultSet. The next method must be called once before the
               // first data is available for viewing. As long as next returns
               // true, there is another row of data that can be used.
               while (rs.next()) {
                     // The result set returned from SEARCH has columns
                     // OBJTYPE, OBJATTR, CONTAINING_OBJECT_LIB,
                     // CONTAINING_OBJECT_NAME
                     // OBJECTINFOR, MODIFY_TIME, SC
                     String objtype = rs.getString("objtype");
                     String objattr = rs.getString("objattr");
                     String containing_object_lib = rs
                              .getString("containing_object_lib");
                     String containing_object_name = rs
                              .getString("containing_object_name");
                     // OBJECTINFOR is an XML column which contains all the
                     // detail info about the indexed object.
                     String objectinfor = rs.getString("objectinfor");
                     Timestamp modify_time = rs.getTimestamp("modify_time");
                     // Score can help user do better ordering
                     double sc = rs.getDouble("sc");

                     // parseObjectInfo is used to parse XML column and output
                     parseObjectInfo(objtype, objattr, objectinfor);
              }
        }
   } catch (SQLException e) {
                     // Display more information about any SQL exceptions that are
                     // generated as output.
                     System.out.println("SQLException exception: ");
                     System.out.println("Message:....." + e.getMessage());
```

```
                  System.out.println("SQLState:...." + e.getSQLState());
                  System.out.println("Vendor Code:." + e.getErrorCode());
                  e.printStackTrace();
            }

    }

    /**
     * Parse object infor XML content to text format and print it to standart
     * output Based on different object type and attribute, there are different
     * way to do parsing.
     *
     * @param objtype
     * @param objattr
     * @param objectinfor
     */
    public static void parseObjectInfo(String objtype, String objattr,
                String objectinfor) {

         // DocumentBuilderFactory creates a factory instance.
         // It will be used to create document builder then
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
         // DocumentBuilder class provides function to parse XML document
         DocumentBuilder builder;
         // Initialize doc to null. This instance will be initialized then
         Document doc = null;
         try {
              // Initialize DocumentBuilder instance
              builder = factory.newDocumentBuilder();
              // Constructed InputStream instance, which will be used as a
              // parameter while calling DocumentBuilder.parse function
              // ByteArrayInputStream is a class implements InputStream.
              // Since objectinfor variable is string, so use
              // ByteArrayInputStream class to constructed a InputStream
              // instance with bytes of objectinfor variable.
              InputStream is = new ByteArrayInputStream(objectinfor.getBytes());
              // parse function is used to parse InputStream to XML Document
              // object
              doc = builder.parse(is);
          } catch (Exception e) {
              System.out.println("Caught exception: " + e.getMessage());
          }

          if (objtype.equals("*OUTQ ") &&
                  objattr.equals("*SPLF ")) {
               // For the object whose object type is "*OUTQ " and object
               // attribute is "*SPLF ",

               // user following way to get the detail value
               System.out.println("=======================================");
               // Get text content for element whose tag name is "job_name" or the
               // other tag.
               System.out.println("Job name:"
                       + doc.getElementsByTagName("job_name").item(0)
                                .getTextContent());
               System.out.println("Job user name:"
                       + doc.getElementsByTagName("job_user_name").item(0)
                                .getTextContent());
               System.out.println("Job number:"
                       + doc.getElementsByTagName("job_number").item(0)
                                .getTextContent());
               System.out.println("spool file name:"
                       + doc.getElementsByTagName("spool_file_name").item(0)
                                .getTextContent());
               System.out.println("spool file number:"
                       + doc.getElementsByTagName("spool_file_number").item(0)
                                .getTextContent());
               System.out.println("Job system name:"
                       + doc.getElementsByTagName("job_system_name").item(0)
                                .getTextContent());
               // The date format CYYMMDD is defined as follows:
               // C Century, where 0 indicates years 19xx and 1 indicates years
               // 20xx.
               // YY Year
               // MM Month
               // DD Day
               System.out.println("create date:"
                       + doc.getElementsByTagName("create_date").item(0)
                                .getTextContent());
               // The time format HHMMSS is defined as follows:
               // HH Hour
               // MM Minutes
```

```
            // SS Seconds
            System.out.println("create time:"
                    + doc.getElementsByTagName("create_time").item(0)
                             .getTextContent());

            /* The output should like below

            ========================================
            Job name:QPRTJOB
            Job user name:NTL
            Job number:066537
            spool file name:QPJOBLOG
            spool file number:3526
            Job system name:RCHASRA5
            create date:1110430
            create time:152003
            */
        } else if (objtype.equals("*STMF ") &&
                objattr.equals("*DATA ")) {
            // For the object whose object type is "*STMF " and object
            // attribute is "*DATA ",
            // user following way to get the detail value
            System.out.println("========================================");
            // Get text content for element whose tag name is "file_path"
            System.out.println("File path:"
                    + doc.getElementsByTagName("file_path").item(0)
                             .getTextContent());
            /* The output should like below

            ========================================
            File path:/home/user/test.txt
            */
       }
 }

 /**
  * The following method ensures that any JDBC resources that are still
  * allocated are freed.
  */
 public void cleanup() {
            try {
                if (connection != null)
                     connection.close();
            } catch (Exception e) {
                System.out.println("Caught exception: ");
                e.printStackTrace();
            }
      }
}
```

Listing 7: Example output

Note: This assumes that one spool file and one IFS stream file match the search keywords.

```
========================================
             Job name:QPRTJOB
             Job user name:NTL
             Job number:066537
             spool file name:QPJOBLOG
             spool file number:3526
             Job system name:RCHASRA5
             create date:1110430
             create time:152003

========================================
             File path:/home/ntl/test.txt
```

Conclusion

You should now understand how to use the new OmniFind Extension stored procedures to perform the following operations:

- Create a Text Search Collection

- Add one or more object sets of spool files or stream files in IFS
- Update the index
- Perform a search

These new IBM i search capabilities offer a solution that makes it easy to boost the functionality of your applications; plus an uncomplicated way to write utilities that can quickly find different kinds of objects on the system. OmniFind V1R2 is available for IBM i 7.1 at no additional charge.

Resources

V1R2 OmniFind Reference Manual

OmniFind Extensions User Guide

OmniFind White Paper

DB2 for i Forum

DB2 for i Stored Procedures Redbook

DB2 for i Tech Updates Wiki

About the authors

Nick Lawrence has worked on DB2 for i for twelve years. His responsibilities include full text search for DB2 and SQL/XML.

Jian Li is a Staff Software Engineer in CSTL. He has been working for DB2 for i for about 5 years. He has been working on OmniFind Text Search Server for DB2 for i in last few years. Now he is also working on MySQL storage engine for IBM i.

Close [x]

developerWorks: Sign in

IBM ID:
Need an IBM ID?
Forgot your IBM ID?

Password:
Forgot your password?
Change your password

☐ Keep me signed in.

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

The first time you sign into developerWorks, a profile is created for you. **Select information in your profile (name, country/region, and company) is displayed to the public and will accompany any content you post**. You may update your IBM account at any time.

All information submitted is secure.

Close [x]

Choose your display name

The first time you sign in to developerWorks, a profile is created for you, so you need to choose a display name. Your display name accompanies the content you post on developerWorks.

Please choose a display name between 3-31 characters. Your display name must be unique in the developerWorks community and should not be your email address for privacy reasons.

Display name: (Must be between 3 – 31 characters.)

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

All information submitted is secure.

- 1 star 1 star1 star
- 2 stars 2 stars2 stars
- 3 stars 3 stars3 stars
- 4 stars 4 stars4 stars
- 5 stars 5 stars5 stars

Submit

Add comment:

Sign in or register to leave a comment.

Note: HTML elements are not supported within comments.

Notify me when a comment is added1000 characters left

Post

There is a problem in retrieving the comments. Please refresh the page later.

Print this page Share this page Follow developerWorks

About
Help
Contact us
Submit content
Feeds
Report abuse
Terms of use
Third party notice
IBM privacy
IBM accessibility
Faculty
Students
Business Partners

IBM®