Exploring the IBM OmniFind Text Search Server

Performing high-speed linguistic text searches against DB2 text data and documents stored in rich-text formats

Kent Milligan
ISV Business Strategy and Enablement
August 2009

Table of contents

Abstract

This paper explores the newest IBM DB2 for i indexing technology — the IBM OmniFind Text Search Server. This paper introduces you to the OmniFind text-indexing technology and how it compares with traditional database index technologies. In addition, the paper discusses how applications that are developed in any language can use SQL to perform text searches against simple text data stored in DB2 databases, business data encapsulated in XML documents, and data stored in rich-text document formats such as Adobe PDF or Microsoft Word.

Introduction

Many databases contain long-character columns that have free-form description and memo fields with notes describing a customer or product. Obviously, these free-form character fields might contain data that provide valuable business insights — assuming that a business analyst can quickly find and locate the data. Standard database search predicates and indexing technologies are not optimal when finding text strings in the middle of a paragraph. These standard technologies perform byte-by-byte comparisons to find an exact match of the specified search string. This type of approach does not work or perform well when the specified search string is the next-to-last word in a 1000-byte character column.

Standard database technologies also provide no value when customer or product data is stored in XML or rich-text documents that have been created with tools such as Microsoft® Word or Adobe Acrobat. With the proliferation of computing devices and word-processing software, there is a very good chance that data stored in rich-text documents can help your company.

The challenges associated with searching and analyzing text data are why most database products now offer full text-search solutions — thus, delivering high-speed, advanced text-search capabilities by looking at text columns and documents as a series of words, instead of a sequence of characters.

This paper describes how IBM® OmniFind® Text Search Server can enable your applications to perform advanced, high-speed linguistic searches against text data that is stored in either simple IBM DB2® character columns or rich-text document formats.

IBM OmniFind for DB2 for i

OmniFind Text Search Server for DB2 for i delivers the common DB2 Family text-search technology to the IBM i developer community. It requires a minimum operating-system release level of IBM i 6.1. OmniFind also replaces the DB2 Text Extender product that is available on previous IBM i releases. Unlike its predecessor product, OmniFind is a no-charge product. Currently, the product must be ordered and installed separately. The OmniFind Text Search Server product identification number is 5733-OMF.

OmniFind provides interfaces and indexing technology that let applications perform high-speed linguistic text searches against character data. The OmniFind technology is not limited to text data stored in simple character columns. It also supports text stored in rich-text document formats such as Microsoft Word, Open Office and Adobe Acrobat, as well as data stored in XML and HTML files. Further, these rich-text documents do not have to be stored in a DB2 table. OmniFind can index and search rich-text documents stored in a DB2 LOB (large object) column or outside of DB2 in the IBM i integrated file system (IFS).

In addition, the OmniFind product supports a wide range of languages (26 languages, including double-byte languages such as Chinese, Japanese, and Korean).

Sample search capabilities

Before getting into setup and usage details for OmniFind, here are a few examples to show its powerful search capabilities. The first thing to notice is that both examples use SQL. Interaction with OmniFind Text Search Server is accomplished with the SQL CONTAINS and SCORE scalar functions.

The first example uses the CONTAINS function to search the contents of a DB2 column named *story*. The story column is defined with BLOB column data type and Microsoft Word documents are stored within this BLOB column. The application directs the CONTAINS function to find all the stories that contain the words 'mice chasing cats'. The AND logical operator is the default operator between the search keywords. Thus, this example search specifies that a match should only be returned when the target text contains all three words ('mice' AND 'chasing' AND 'cats'). When a match is found by the CONTAINS function, the function returns a value of **1** — this explains why the selection predicate compares the value of the CONTAINS function to **1**.

```
SELECT author_name, title FROM books
  WHERE CONTAINS(story, 'mice chasing cats') = 1
        AND pubdate >= '1/1/2005'
```

Figure 1. OmniFind search example with CONTAINS

The CONTAINS function call in this example not only returns matches on the three search arguments of 'mice chasing cats', but also returns matches on variations of these search words. For instance, the CONTAINS function returns a match if the story text includes the following sentence:

Melvin the mouse chased the Charlie the cat.

The sentence contains 'mouse', 'chased' and 'cat', which are all variations of words in the original search argument. This is a good example of the OmniFind advanced linguistic-search capabilities that are not possible with standard database technologies. Remember, traditional database technologies are usually designed to only find an exact match. This exact match difference also arises with the case-insensitive searches that OmniFind performs by default. Case-insensitivity means that the sample query in Figure 1 will identity the following phrase, 'MICE CHASING CATS', as a match even though the search argument is entered in all lower-case characters.

The second OmniFind example in Figure 2 performs searches on text data stored in feeddoc, which is defined as a variable-length character column that stores text captured from various Internet feeds.

This example also incorporates the OmniFind, SCORE function. SCORE returns a relevance score value that measures how well the target text matches the search criteria. The more occurrences of the search criteria that are found, the larger the score value is. When a match is not found, a zero score value is returned. Otherwise, the score value is in a range that is greater than zero and less than one.

This example also shows that you can combine OmniFind functions with traditional SQL syntax, such as search predicates (feedDate > '01/01/2008') and ordering criteria (ORDER BY searchScore DESC).

```
SELECT feedSrc, feedDate,
       SCORE (feedDoc, 'California insurance settlement') AS searchScore
 FROM newsfeeds
 WHERE CONTAINS(feedDoc, 'California insurance settlement') = 1
       AND feedDate > '01/01/2008'
 ORDER BY searchScore DESC
```

Figure 2. OmniFind example with SCORE *and* CONTAINS

This second example also provides another demonstration of the advanced linguistic capabilities of the text-search server. The search string on both SCORE and CONTAINS refers to the name of a state, *California*. The OmniFind server recognizes California as a state and also searches for the two-digit state abbreviation ('CA') at the same time that it searches for the 'California' string. As a result, the following sets of news feed text will be identified as a match by this OmniFind search request:

- $100 million insurance settlement to CA firm
- California man wins insurance settlement

These simple examples just scratch the surface of the types of searches that you can perform with OmniFind Text Search Server. Now that you understand the product's capabilities at a high level, it is time to learn about the requirements for OmniFind.

Database requirements

The OmniFind product can be used to index and search text data stored in DB2 tables (physical files). The table being indexed must contain a ROWID column or have a primary key or unique key constraint defined over it. A uniquely-keyed physical file cannot be indexed until either the Add Physical File Constraint (ADDPFCST) command or SQL ALTER TABLE statement have been used to register that DDS-defined unique key as a constraint. The unique key is required so that OmniFind can update and maintain the associated text-search index.

You must define a column with one of the following data types to use the OmniFind text search support:

- BINARY
- VARBINARY
- BLOB
- CHAR
- VARCHAR
- CLOB
- DBCLOB
- GRAPHIC
- VARGRAPHIC

If you use OmniFind to search text in rich-text documents, then you should verify that the document format type is supported; see the following partial list of document types supported by OmniFind:

- XML
- HTML
- Adobe PDF
- Rich Text Format (RTF)
- Microsoft Excel
- Microsoft PowerPoint

- Microsoft Word
- IBM Lotus® 123
- IBM Lotus Freelance®
- IBM Lotus WordPro
- IBM Lotus Symphony
- OpenOffice 1.1 and 2.0
- OpenOffice Calc
- Quattro Pro
- JustSystems Ichitaro
- StarOffice Calc

This partial list of supported formats shows that OmniFind does support a wide variety of formats — including the most popular document formats.

You can only specify the OmniFind CONTAINS and SCORE functions on SQL statements that can be processed by the SQL Query Engine (SQE). That is, if the SQL request or underlying database contain attributes that cause the Classic Query Engine (CQE) to be used, references to the CONTAINS or SCORE functions are not allowed.

Configuring OmniFind

OmniFind Text Search Server runs separately from DB2 for i. DB2 for i interacts with the text search to process any text-search or text-index requests. Figure 3 depicts this server interaction.

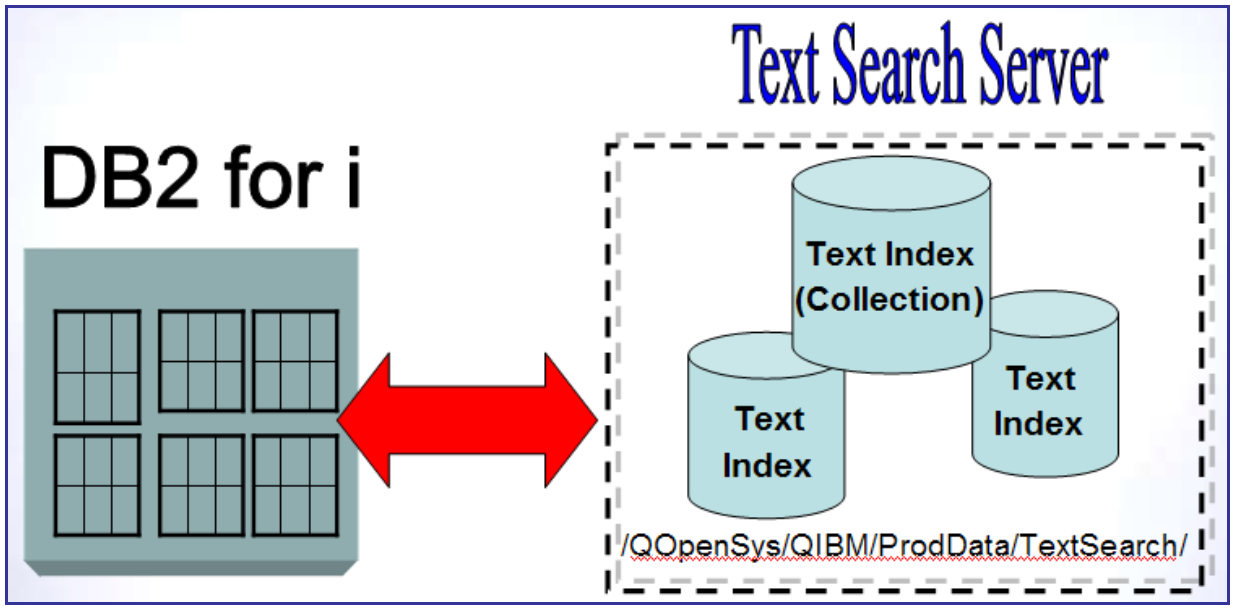

Figure 3. OmniFind Text Search Server

Also notice that the text indexes created and used by OmniFind are also stored separately from normal DB2 objects. The IFS directory (/QOpenSys/QIBM/ProdData/TextSearch/) used by OmniFind for text index storage is created during installation of the OmniFind product.

You can manually start and stop OmniFind Text Search Server by using the following configuration stored procedures, which are found in the SYSPROC schema (library):

- SYSTS_START
- SYSTS_STOP

When the OmniFind Text Search Server has been started manually at least one time, DB2 for i will automatically try to restart the server when it's not active — including after a system IPL. Whenever DB2 for i receives an SQL statement that references the CONTAINS or SCORE functions, it checks to see if a text-search server is running and automatically starts the OmniFind Text Search Server when it is not active. The only time the automatic start process is not performed by DB2 is when the server has been manually stopped with the SYSTS_STOP stored procedure. The status of the text-search server can be verified programmatically by interrogating the SERVER_STATUS column in the QSYS2.SYSTEXTSERVERS catalog.

Understanding text indexes

OmniFind uses text index objects to deliver high-performance searches. A text index object is implemented as an IFS object instead of a formal IBM i operating system object. Be aware that some OmniFind product documentation uses the term *collection* to refer to a text index. An OmniFind collection is the IFS directory used to store the contents of a text index.

Text indexes differ from traditional database index technologies in several ways. A key difference is that a tree-based structure is not used for the index structure. Instead, a text index usually just contains a list of the significant words found in the underlying text. Various metrics are stored along with the words, such as the number of occurrences. This unique structural difference is the primary reason text indexes can outperform traditional index technologies in the area of text searches.

Before learning how to create a text index, it is useful to understand the other differences between DB2 indexes and the text indexes used by OmniFind. The storage location difference is reflected in Figure 3. Text indexes are stored in IFS, as opposed to being stored within a schema (library) like a traditional DB2 index. This is an important fact to know, because it means that your backup process must be changed to include the IFS directory used by OmniFind. Backup and recovery considerations are fully documented in the *IBM OmniFind Reference Manual* (http://publib.boulder.ibm.com/systems).

By default, DB2 indexes are maintained immediately (that is, as the underlying data in a table is changed through inserts, updates and deletes). This is not the case with an OmniFind text index. Text indexes are maintained asynchronously on a scheduled basis. The frequency of the scheduled updates is customizable by the user. The scheduling options are covered in the next section.

To understand the differences between these two indexing technologies in real-world terms, a text index and SQL index were both created over the same variable-length column in a table. This variable-length column had a length of 32 000 bytes and the table contained 175 000 rows of data. Table 1 contains metrics that capture some of the differences between the two indexing technologies. (Your performance results will vary based on the data and system configuration.)

The first thing that jumps out from this table is how much longer it took to create the text index. That is not too surprising when you consider that OmniFind has to parse and process each individual word stored in the column. Contrast that process with the SQL index approach, which essentially treats all of the text in the column as a very long, single word. It is easy to see why the SQL index creation runs so much faster.

Although the long build time for text indexes can be troubling, realize that the SQL index is useless when it comes to speeding up text searches against the variable-length column. When an application needs to determine if the string 'IBM' is contained anywhere within the column, the SQL index cannot be used to search the text within the column. Using traditional SQL, the application must perform a byte-by-byte search of the entire column. The text index can figure out instantly whether the search string 'IBM' exists anywhere within the column. Text indexes trade longer build times for high-speed search performance at run time. Tips for improving the build times for text indexes are covered in a later session.

Metrics	SQL index	Text index
Creation time	3 minutes	3 hours
Object size	1.7 GB	0.1 GB

Table 1. Comparison of SQL and Text index

This extra processing that occurs during the creation of a text index also explains the large difference in object sizes. The key values in the SQL index contain white spaces and multiple occurrences of the same word because the variable-length column value is treated as one long word. In contrast, the text index build process eliminates white space as it parses each individual word within the column. The text index also only stores a single occurrence of each word. Thus, it is no surprise that text indexes are much smaller in size than traditional SQL indexes.

Creating a text index

Similar to the text search server, you create and manage OmniFind text indexes by using system stored procedure calls. The SYSTS_CREATE stored procedure in SYSPROC supplies the text index creation interface. Figure 4 shows a sample call to the SYSTS_CREATE procedure.

```
CALL SYSPROC.SYSTS_CREATE (
  ‘myschema’,
  ‘resumes_index’,
  ‘myschema.resumes(applicant_resume)’,
  ‘FORMAT INSO  UPDATE FREQUENCY D(*) H(*) M(0) INDEX
CONFIGURATION(UPDATEAUTOCOMMIT 500000)’
 )
```

Figure 4. Example of text index creation

The first two parameters are used to supply the text index schema and object names, respectively. For this example, the text index name and schema are *resumes_index* and *myschema*. The third parameter identifies the column that contains the text or document that needs to be indexed. This example creates an OmniFind text index on the applicant_resume column in the resumes table found in myschema. The applicant_resume column is a 250 MB BLOB column that is used to store rich-text documents containing resumes. The fourth and final parameter is an options parameter. The options parameter in this example specifies the format of the text being indexed and controls the frequency of updates to the text index.

The FORMAT option describes the format of the text in the column that is being indexed. In this case, the INSO format is specified because the resume text is stored in rich-document formats. The stored resume documents do not have to use the same document format. The INSO format type allows the resume documents that are stored in this column to use any of the document types supported by OmniFind (Adobe PDF, Microsoft Excel, RTF, and more). While the INSO format type does support all of the document format types, it should only be used when there is a mix of document formats. For instance, specifying the INSO format for a column containing plain text would unnecessarily slow down the text index build time.

The other supported FORMAT values are: TEXT, XML and HTML. With these format values, all of the documents in the indexed column should be of the same format. Again, a careful review of the underlying text data should be performed to ensure that the FORMAT value correctly describes the format of the text data. Unexpected results and performance slowdowns may occur when the FORMAT option value does not fully match the data in the indexed column. For example, if a column contains XML documents and TEXT is specified for FORMAT value is TEXT, then the user will not be able to use the XML search functionality. This example is a scenario that OmniFind is unable to recognize as having an incompatible FORMAT value — in some situations, OmniFind will signal a parser error when the FORMAT option value is not compatible with the underlying text data.

The UPDATE FREQUENCY option dictates how often the text index is automatically updated to reflect changes that have been made to the indexed column. Remember that text indexes are not maintained immediately to reflect data changes (as are SQL indexes). Users control the text index maintenance behavior with the UPDATE FREQUENCY and UPDATE MINIMUM options. If an update frequency is not specified, the text index is not updated on a scheduled basis. Instead, an administrator must update the text index manually. The text index in this example is updated every day at the start of every hour because of the '*' character specified on the day and hour attributes. You can find out more information on the update-frequency values in the *IBM OmniFind Reference Manual* (http://publib.boulder.ibm.com/systems).

The remaining low-level text index configuration options must be specified along with the INDEX CONFIGURATION keyword. The example in Figure 4 specifies a value for the UPDATEAUTOCOMMIT configuration option. During the index update process, the UPDATEAUTOCOMMIT configuration option controls how frequently index updates are committed and how often synchronization occurs between the text-search server and the requesting DB2 job. The default value for UPDATEAUTOCOMMIT is 150. Specifying a larger value for this option can significantly improve the performance of the initial index update or build. The performance impact of this index configuration option is covered in more detail in a later section.

Here is a list of the options that you can specify on the *options* parameter of the SYSTS_CREATE stored procedure. Details on each option can be found in the *IBM OmniFind Reference Manual* (http://publib.boulder.ibm.com/systems).

- Text information
  - CCSID
  - LANGUAGE
  - FORMAT

- Index maintenance options
  - UPDATE FREQUENCY
  - UPDATE MINIMUM
- Index configuration options
  - CJKSEGMENTATION
  - COMMENT
  - IGNOREEMPTYDOCS
  - KEY COLUMN
  - SERVER
  - UPDATEAUTOCOMMIT

When a text index is created, five additional DB2 objects are also created. Figure 5 shows these DB2 for i objects that are created in addition to the text index object that resides in the IFS. An application performing a text search never needs to refer to these objects. Instead, the OmniFind Text Server uses the objects for internal processing.

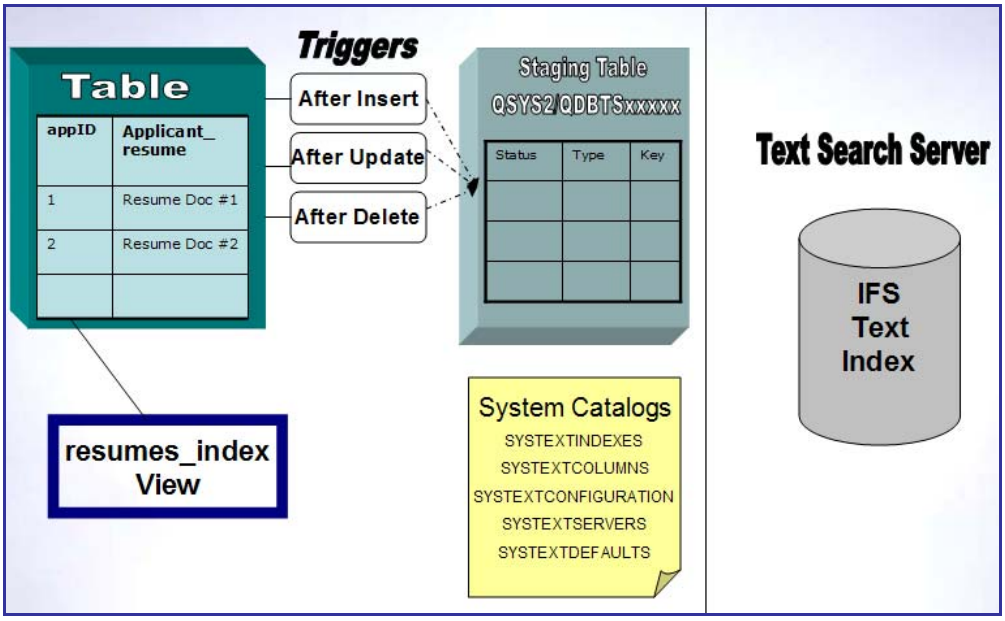

Figure 5. Objects created by SYSTS_CREATE

Awareness of these objects is good knowledge to have because four of the five additional DB2 objects create object dependencies on the table that is being indexed. First, an SQL view is created with the same name as the text index name that references the underlying table. The SQL view contains data about the underlying text index, such as the staging table name, number of pending updates, and a timestamp reflecting the last time that the text index was updated. This type of information can be beneficial to an administrator or programmer looking to check on the current status of a text index. The SQL view also serves as a symbolic link between the table and the text index object that is stored in IFS. When database object relationships are examined with tools, such as the IBM System i™ Navigator

Show Related task or the Display Database Relations (DSPDBR) system command, the SQL view created by OmniFind represents the text index's relationship to the underlying table.

Three SQL triggers are also created on the underlying table. DB2 for i calls these SQL triggers after every update, insert and delete on the underlying table to capture any changes that were made to the indexed text column. The SQL triggers are responsible for tracking changes to the text data, so that the changes can eventually be propagated to the text index object.

The text data changes are tracked by writing rows into a staging table. The staging table is the only DB2 object created by OmniFind that does not directly reference the table being indexed. The staging table has an indirect relationship with the column being indexed. The OmniFind server creates all staging tables in the QSYS2 schema (one staging table for each text index). When the specified update-frequency threshold is reached, OmniFind uses the staging table to determine which changes to propagate to the text index.

The System Catalogs box in Figure 5 highlights the fact that OmniFind adds text index metadata to DB2 system catalogs in QSYS2 each time a text index is created. You can also query these catalog views to determine the status of the text-search search server, a text index, or the relationship with DB2 objects.

A text index is not populated with values during the initial creation. By default, the text index population is deferred until the first update-frequency threshold is reached. Another alternative is manually updating the text index with SYSTS_UPDATE stored procedure or initiating a full index rebuild with the SYSTS_REPRIMEINDEX stored procedure. Both of these system stored procedures are discussed later in this paper.

Populating a text index

One of the ways to manually update a text index is with the SYSTS_REPRIMEINDEX stored procedure. This procedure performs a full rebuild of the text index by clearing the contents of the index and repopulating it with values stored in the indexed column. Often, the SYSTS_REPRIMEINDEX procedure is called immediately after creating a text index instead of waiting for the first scheduled update period to populate the newly created index. Outside of being used immediately after a text index creation, the SYSTS_REPRIMEINDEX should only be called when the data in the existing index needs to be thrown away and repopulated. This situation could occur on a restore operation where a backup copy of the underlying table is restored and the data in the table does not match up with the contents of the text index.

Figure 6 shows an example invocation of this procedure. The text index's schema name and object name are the first two parameters on the SYSTS_REPRIMEINDEX procedure call. The third parameter is reserved for future usage.

```
CALL SYSPROC.SYSTS_REPRIMEINDEX (
  ‘myschema’,
  ‘resumes_index’,
  ''
 )
```

Figure 6. Example of SYSTS_REPRIMEINDEX

Batch execution of the index rebuild (or initial build) can be accomplished by using the Run SQL Statements (RUNSQLSTM) system command to perform the procedure call. In fact, any of the OmniFind stored procedures can be executed in batch mode with the RUNSQLSTM command.

The performance recommendations discussed in the index maintenance section of this paper also apply to invocations of the SYSTS_REPRIMEINDEX stored procedure.

Maintaining a text index

Although updates to a text index are not performed immediately, as it is for an SQL index, text index updates can occur automatically. You can control the automatic updates by supplying update-frequency settings during the initial index creation with the SYSTS_CREATE procedure. You can use the SYSTS_ALTER procedure to add or change the update settings on an existing OmniFind text index.

The update settings consist of two values — UPDATE FREQUENCY and UPDATE MINIMUM. UPDATE FREQUENCY controls the scheduling of the text index update. The settings specified on this option dictate how often changes to the underlying text column are propagated to the associated text index. The maximum frequency you can specify for text index updates is five minutes. The text index update frequency is likely to be different for each application, depending on business requirements. Weekly updates to a text index might work for one application, yet another text-search solution needs hourly updates.

Figure 7 contains an example of using the SYSTS_ALTER stored procedure to change the UPDATE FREQUENCY value for an existing index. In this case, the text index will now be automatically updated twice an hour — at the beginning of the hour and halfway through the hour at the 30-minute mark.

```
CALL SYSPROC.SYSTS_ALTER( 'ixschema', 'ixname',
                          'UPDATE FREQUENCY D(*) H(*) M(0,30)')
```

Figure 7. Example of the SYSTS_ALTER procedure

UPDATE MINIMUM is the other option that directly influences the automatic update behavior of a text index; it sets a threshold for the minimum number of updates that must have taken place on the indexed column to enable the automatic text index update. Consider a text index that is created to be automatically updated nightly at midnight. In addition, the text index is created with an UPDATE MINIMUM value of 100. At midnight, the OmniFind server only updates the text index if at least 100 rows in the indexed column were changed since the last update to the text index. If the minimum number of changes threshold is not reached, the automatic index update is deferred until the next frequency interval is reached.

If the UPDATE MINIMUM value is not specified, the default value is **1**. This means that automatic index processing occurs when at least one change has been made to the indexed column

Figure 8 contains examples of the UPDATE MINIMUM option in action on the SYSTS_ALTER and SYSTS_UPDATE stored procedures. The SYSTS_ALTER stored procedure changes the UPDATE minimum setting to 250 for an existing text index. This means that the automatic index updates are only done when there have been at least 250 changes to the underlying text column. The SYSTS_UPDATE stored procedure is called to initiate manual updates to a text index. The default behavior is for the SYSTS_UPDATE to unconditionally update the text index. When the USING MINIMUM UPDATE parameter is specified, the manual index update only takes place if the indexed column has changes that exceed the index's UPDATE MINIMUM threshold value.

```
CALL SYSPROC.SYSTS_ALTER( 'ixschema', 'ixname', 'UPDATE MINIMUM 250')

CALL SYSPROC.SYSTS_UPDATE('ixschema', 'ixname', 'USING UPDATE MINIMUM')
```

Figure 8. Examples of UPDATE MINIMUM

Behind the scenes, OmniFind use the IBM i job scheduler to schedule the automatic index updates. The OmniFind scheduler entries can be reviewed with the Work with Job Schedule Entries (WRKJOBSCDE) command. The OmniFind entry-scheduler job names start with the prefix of *US.*

The incremental-update process is represented visually in Figure 9. As changes to the indexed table are made, the SQL triggers that were created by OmniFind log a record of the changes into the associated staging table. When the next automatic index update initiates, the OmniFind server interrogates the staging table to determine which changes in the indexed column need to be propagated to the text index. This enables only the changed column values to be processed instead of the OmniFind server having to process all of the column values.

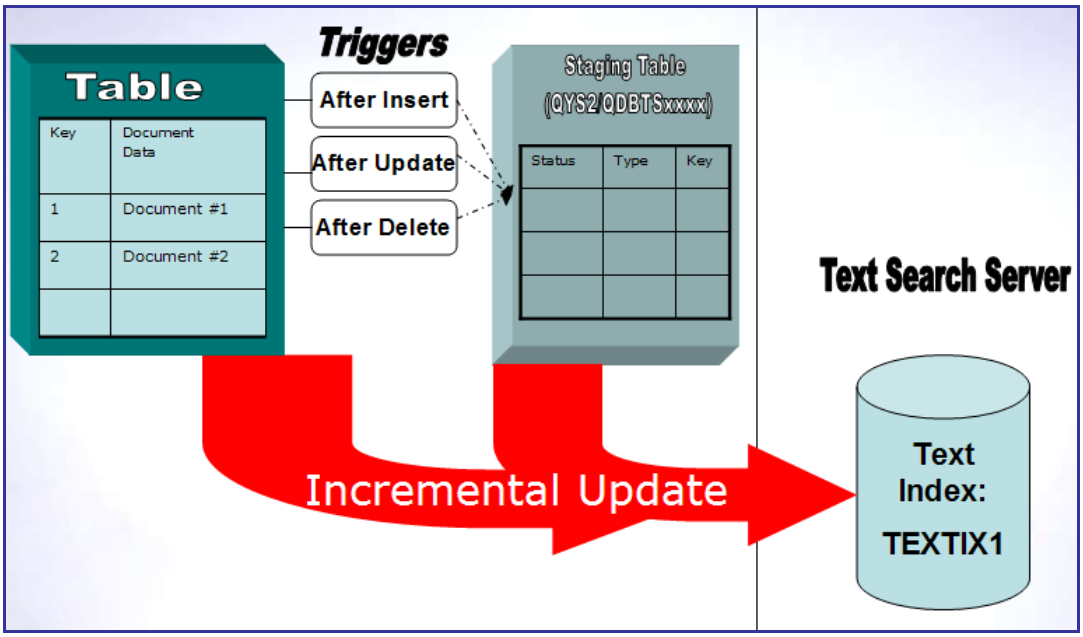

Figure 9. Incremental-update process

A text index does remain searchable during the incremental-update process.

Index maintenance performance

As documented earlier, the UPDATEAUTOCOMMIT index configuration option can be used to significantly improve the performance of index maintenance and index builds. The UPDATEAUTOCOMMIT configuration option controls how frequently index updates are committed and how often synchronization occurs between the text-search server and the invoker's job. By default, synchronization processing and commits are performed after every 100 updates. These extra synchronization operations can slow index maintenance performance significantly.

The commit operations release locks acquired by OmniFind when processing rows in the staging table. During the initial build of an index, the staging table should not contain any rows. Thus, in this initial build scenario the commit operations are providing no value to the index build process. Specifying a larger value for the UPDATEAUTOCOMMIT configuration option enables you to reduce the number of times that OmniFind interrupts the index build and maintenance processes to issue commit operations and synchronization processing. In one performance test, the index build process ran 6 times faster than the default settings when a value of 50000 was used for the UPDATEAUTOCOMMIT option when building an index over a table containing 35000 rows. Obviously, performance results will vary from system to system.

While a large UPDATEAUTOCOMMIT option value can improve performance, it can also produce a negative impact in some situations. A large value results in synchronization and commits being executed less often. As a result, system resources such as locks are held longer which can put a burden on system performance — especially in the following two scenarios:

- An incremental update is being performed on the text index and the associated staging table contains thousands of changes
- An index build or incremental update is being performed over text data that results in a large number of errors and warnings being returned. Errors and warnings can be caused when the text-search server encounters anomalies such as malformed XML documents and corrupt rich-document format files.

If these two scenarios will be common occurrences with your application environment, then a smaller UPDATEAUTOCOMMIT value is recommended.

Another possible solution is use a mix of UPDATEAUTOCOMMIT settings for a text index. A large UPDATEAUTOCOMMIT value is specified for the initial build of a text index to improve build performance. After the index build completes, the UPDATEAUTOCOMMIT option is reduced to a smaller value to minimize the time that system resources are held during incremental index updates.

Figure 10 contains an example of this scenario demonstrating how to vary the UPDATEAUTOCOMMIT option with the SYSTS_ALTER stored procedure. First, the UPDATEAUTOCOMMIT is set to a 100000. A text index can also be created with a large UPDATEAUTOCOMMIT setting as shown in Figure 4. Next, the SYSTS_REPRIMEINDEX procedure initiates the text index build process with the large commit setting active. After the index build completes, the SYSTS_ALTER procedure is called to reduce the UPDATEAUTOCOMMIT option to a smaller value.

```
CALL SYSPROC.SYSTS_ALTER( 'ixschema', 'ixname', 'INDEX
CONFIGURATION(UPDATEAUTOCOMMIT 100000)')
CALL SYSPROC.SYSTS_REPRIMEINDEX( 'ixschema', 'ixname', '')
CALL SYSPROC.SYSTS_ALTER( 'ixschema', 'ixname', 'INDEX
CONFIGURATION(UPDATEAUTOCOMMIT 100)')
```

Figure 10. Example of using multiple UPDATEAUTOCOMMIT values

Dropping a text index

The SYSTS_DROP stored procedure provides the mechanism for dropping (deleting) a text index. This procedure removes the text index object that resides in the IFS and also removes the additional DB2 objects (see Figure 5) that the OmniFind server created for internal processing.

Note that dropping a table may not delete the text index object or any of the DB2 objects associated with the text index. Thus, removal of a text index and its associated objects shouldt be done with the SYSTS_DROP stored procedure.

Calling the SYSTS_DROP procedure is straight-forward since it only requires the name of the text index schema and text index as parameters.

Indexing external data

As mentioned earlier, the OmniFind Text Search server also has the ability to index and search text data that is **not** stored in a DB2 for i table. Assume there is an IFS directory on your server that contains brochures for each of your products in PDF format. With a little extra programming, the OmniFind server can index the text data in those PDF brochures and enable SQL searches against that same text data — even though the data is external and stored outside of DB2 for i.

The ability to index and search external data is accomplished through a user-defined function (UDF), which is required to redirect the OmniFind capabilities to an external location. This is where the extra programming becomes necessary — you are responsible for creating the UDF.

The example in Figure 11 attempts to capture the high-level requirements of the UDF. The input parameter for the function is a file-path value identifying the location of the external file that needs to be processed by the OmniFind server. The output of the external UDF is most likely a large object (LOB) locator value because the data stored in an external file is written in binary format. The locator provides a reference or handle to the stream of text data that is stored in the external file. The output locator value is then used by OmniFind to process and index the text data stored in the IFS file.

```
CREATE FUNCTION getIFSfile(VARCHAR(2000))
      RETURNS BLOB(2G) AS LOCATOR
      LANGUAGE C++
      EXTERNAL NAME 'LIB1/PGM1(getIfsFile)'
      PROGRAM TYPE SUB
      DETERMINISTIC
      PARAMETER STYLE SQL
```

Figure 11. Example of a UDF for external data

In this example, C++ is used to create the service program, LIB1/PGM1(getIfsFile). This service program is registered as an external UDF with the CREATE FUNCTION statement. You can use any high-level programming language (RPG or COBOL, for example) supported by the IBM i operating system in the development of external UDFs. To obtain the source code for the UDF in this example send an email to: rchudb@us.ibm.com.

After the UDF is in place, there are a few other steps. You must store the file path for each external file in a column of a DB2 table. The column definition is similar to the brochurePath column in Figure 12. Next, you must insert a row in the prod_brochure table for each external file that OmniFind needs to process. You can automate the population of the table with the external file-name values by using a shell script that uses the grep command.

```
CREATE TABLE prod_brochure (
    prodID INTEGER PRIMARY KEY,
    brochurePath VARCHAR(2000)
)
```

Figure 12. Example of a file-path column for external data

The final step is creating the text index. You still create a text index for the external data with the SYSTS_CREATE procedure. However, the invocation is slightly different because you must refer to the UDF on the *column* parameter. Remember, the third parameter of the SYSTS_CREATE stored procedure identifies the column that the text index will be created over. Notice in the example shown in Figure 13 that the third parameter contains a reference to the getIfsFile UDF. Because this UDF returns the content of the external file, the OmniFind server processes and indexes the file contents, instead of indexing the file-path value that is stored in the brochurePath column. A text index over the external file names is not useful. That is why the UDF is a critical piece of the solution for data that is stored outside of DB2.

```
CALL SYSPROC.SYSTS_CREATE (
  'myschema',
  'brochure_index',
  'myschema.prod_brochure(getIfsFile(brochurePath))',
  'FORMAT INSO'
 )
```

Figure 13. Creating an index over external data

Maintenance of the text indexes created over external files requires additional planning. When data in the external file is changed, the triggers created by OmniFind to track changes in the indexed column are not called. The triggers are only capable of detecting changes to a DB2 column. This means the triggers are only called when the file-path value in column changes, if the recommended solution for external data is followed. As a result of the database triggers not being able to track changes, no automatic updates are performed on the text indexes.

The simplest solution to address the index maintenance issue is to regularly rebuild the text index with the SYSTS_REPRIMEINDEX stored procedure. The regularity of the text-index rebuilds depends upon the requirements of your application and upon the frequency of changes to the data within the external files.

OmniFind search capabilities

As documented earlier in this paper, the CONTAINS and SCORE function provide the interface that enables an application to perform advanced, high-speed searches using the text index. Figure 14 contains a visual representation of this search process. Each invocation of the SCORE and CONTAINS functions results in the request being passed along to the OmniFind Text Search Server for execution.

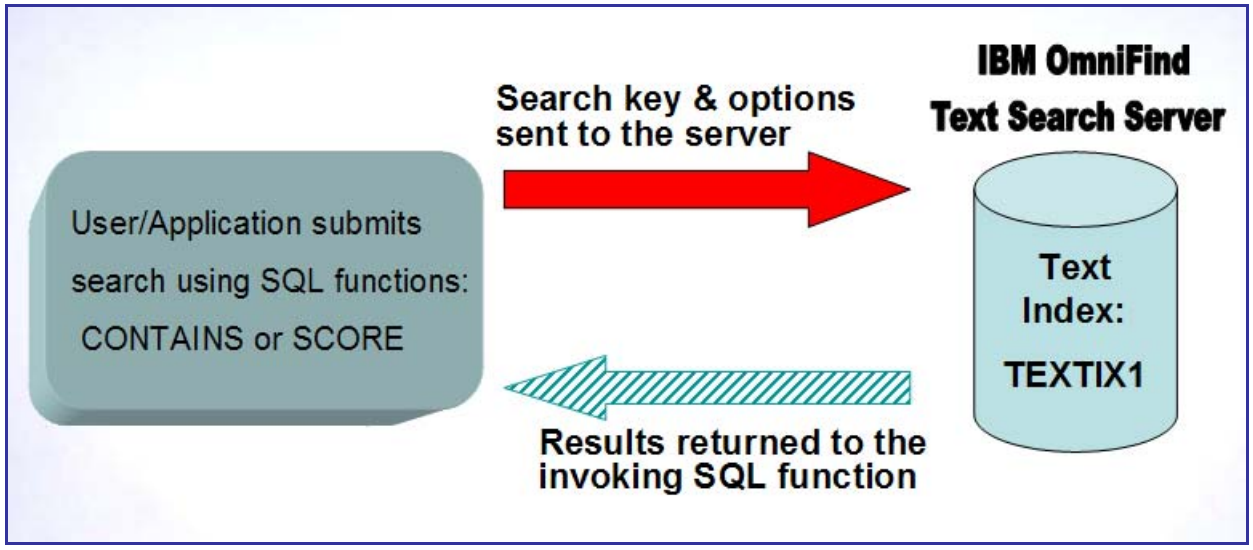

Figure 14. Overview of search processing

Interaction with the text search server is transparent to the developer. The developer simply needs to understand the syntax and semantics of these two functions.

CONTAINS function

As shown in earlier examples, the CONTAINS function provides the interface for searching a text index. The function returns an integer value of **1** if the specified column contains a match for the specified search argument. A value of **0** is returned if no match is found.

The CONTAINS function accepts up to three parameters with the third parameter being optional. The three respective parameters are:

- **Column name:** This parameter identifies the name of the column that has a text index created over it.
- **Search argument:** This parameter contains the search string that will be used when trying to find a match in the text index.
- **Search argument options**: This optional parameter contains options to customize the search behavior. The available options are QUERYLANGUAGE, RESULTLIMIT and SYNONYM. When this parameter is specified, it does not have to contain a reference to all three options. If multiple options are specified on the parameter, they must be separated by at least one blank space.

Figure 15 shows an invocation of the CONTAINS function with all the parameters specified. In this example, the OmniFind server identifies customers who have expressed an interest in product changes. The first thing to notice is that the search-string value is not passed as a literal value. Instead, the search string value is passed indirectly through a host variable. The RESULTLIMIT search option specifies that the result set returned by this query request should contain a maximum of 100 matches. Thus, if more than 100 customers ask for a product change, the result set is limited to 100 customers.

```
char search_arg[100];
 ...

EXEC SQL DECLARE C1 CURSOR FOR
         SELECT custkey FROM customers
         WHERE CONTAINS( comments, :search_arg, 'RESULTLIMIT = 100') = 1
;
 ...
EXEC SQL SET :search_arg = 'product change';

EXEC SQL OPEN C1;
 ...
```

Figure 15. CONTAINS function with all three parameters

Search argument options

Here are more details on the search argument options:

- **RESULTLIMIT:** As shown in Figure 15, you can use this option to identify the maximum result-set size returned by the associated query. If the DB2 query optimizer determines that the best query plan involves calling the OmniFind server for each row of the result, the RESULTSET option is ignored. Therefore, an application must not have dependencies on this option limiting the number of results returned on a query.
- **QUERYLANGUAGE:** This option identifies the language of the search string and does not have to be specified when the search-string language matches the language of the indexed text as well as the language value of the text index. This is reinforced in earlier examples of the CONTAINS functions where the QUERYLANGUAGE option is omitted because the search string, indexed text and text index all use the English language. Figure 16 shows that the language option requires a five-character language code. The list of supported language codes are found in the *IBM OmniFind Reference Manual* (http://publib.boulder.ibm.com/systems). For this example, let's assume that the account_status column contains a combination of English and French text. The CONTAINS invocation in Figure 16 denotes French as the query language because the search string is a French word. Not only does the QUERYLANGUAGE option denote the language of the search string, it also tells the OmniFind server which internal dictionary to use on the search request. OmniFind uses these internal dictionaries to determine the linguistic variations of a word. The search string in this example is *en retard,* which is the French word for *late*. With the language option set to French, the CONTAINS function also returns matches for variations of *en retard*, such as *plus retard (*the French word for *later*). If the QUERYLANGUAGE option is left to its default value of *English*, the text-search server can only match exact occurrences of *en retard,* because the English dictionary has no variations for this French word.

```
… CONTAINS(account_status, 'en retard', 'QUERYLANGUAGE=fr_FR') = 1 …
```

Figure 16. Example of QUERYLANGUAGE option

- **SYNONYM:** This option enables a synonym dictionary to be applied when searching the text index. Being able to provide a user-defined list of synonyms allows you to further enhance the advanced search capabilities of the OmniFind text-search engine. Remember that the base support in the OmniFind server is able to recognize linguistic variations of the same word as equivalent. For example, a search string of 'man' returns matches for text that contains the word *man* or *men*. The synonym option allows you to customize the OmniFind search engine to recognize matching terms in your specific industry. For instance, 'program' and 'code' can be recognized as matching terms when searching data related to the software industry. You supply the synonym dictionary by creating an XML file, and then you must use a tool to associate the synonym file with a text index. Synonym dictionaries are covered in detail later. Even though the synonym dictionary file is associated with a text index, the synonyms are only used by the OmniFind server when the string 'SYNONYM=ON' is specified as a search-argument option.

SCORE function

The SCORE function returns a relevance score denoting how well the target text matches the specified search argument. The result of the function is a double-precision floating-point number. The result value is greater than **0** and less than **1** when a match is found for the specified search criteria. The result value is larger when the search value has been matched repeated times in the indexed text. A value of **0** is returned when no matches are found.

As you might expect, the SCORE function uses the same three parameters as the CONTAINS function.

The example in Figure 17 shows how the SCORE and CONTAINS functions can be used together. The SCORE function is frequently used in conjunction with the CONTAINS function — using the identical parameter values (as shown in this example) to provide a search rating for those text values identified as a match. Normalizing the floating-point value returned from the SCORE function by multiplying it by 100 and converting it to an integer is another common practice to make the rating value easier to read and understand.

```
SELECT projID, projAuthor,
       INTEGER(SCORE(thesis,
                '(parallel OR scalable) AND programming') * 100)
         AS relevance
FROM projects
WHERE CONTAINS(thesis, '(parallel OR scalable) AND programming')=1
ORDER BY relevance DESC
```

Figure 17. Example with SCORE and CONTAINS

Search argument syntax

The search argument in Figure 17 shows some of the advanced syntax provided by the OmniFind product.

Logical operator syntax

The search argument in the prior example includes the logical operators of AND and OR. Logical operators provide developers with the capacity to construct complex search requests to meet their business requirements. The logical operators and grouping of search terms with parentheses in this example enable OmniFind to identify those thesis documents that contain a combination of 'parallel programming' or 'scalable programming'. The ability to include logical operators within the search argument enables this query to be performed with a single request, instead of having to code multiple queries.

When multiple words are contained in the search string, the default logical operator is AND. Thus, a search string of 'computer hardware' is equivalent to the search string of 'computer AND hardware'. This search string only returns a match when the target text contains both computer and hardware in the search string. You can also specify the AND logical operator by using the plus sign (+).

In addition, the text-search server includes support for the NOT logical operator, which you can also specify with the minus sign (-). The NOT operator provides the ability to exclude terms from the search results. The CONTAINS example in Figure 18 finds those papers that discuss secure systems, but excludes any results that reference the Windows operating system.

```
… CONTAINS(papers, 'secure system - windows') = 1 …
```

Figure 18. Example of the minus operator

You must enter the logical operators values of AND, OR and NOT in all uppercase to be recognized as logical operators.

Search-argument syntax

A set of double quotes ("..") transforms an OmniFind search request from a fuzzy search to an exact search. Specifying a search argument of '"late"' on CONTAINS or SCORE only returns a match when the target text contains the word *late*. With an exact match specified, the linguistic variations of a word are not considered, so text values containing the word *later* or *latest* are not identified as matches. Even when an exact match is specified, OmniFind performs the search in a case-insensitive manner.

OmniFind search processing can become fuzzier with the usage of the wildcard character (*). A wildcard character can be useful when the exact spelling of a word or name is not known. A search argument of 'Sh*n Green*', for instance, could return matches for the following spellings of a name: *Shonn Greene*, *Shawn Green* and *Shaun Greene*. You can also use the wildcard character when parts of a name or title are unknown. The search argument of '"John * Kennedy"' identifies *John F Kennedy* and *John Fitzgerald Kennedy* as matches, yet ignores *John Kennedy* as a match. Because an exact match was specified with double quotes, the '"John * Kennedy"' search argument does not identify *John Smith and Jane Kennedy* as a match because there is more than one word in between the search terms of *John* and *Kennedy*. Adding a wildcard character to the beginning of a search argument ('*ter) can slow down the search processing, so use this type of search argument sparingly.

You can customize and weight the relevance score of a text match by using the boost-factor option (^). When a search argument contains multiple words, you can use the boost factor to give a heavier matching weight to one of the words. The search argument in Figure 19 matches any text value that contains *Iowa* or *Minnesota*. However, the boost factor results in a higher relevance score for those text values that contain *Iowa*.

```
SELECT SCORE( textcolumn, 'Iowa^5 OR Minnesota')
```

Figure 19. Example of a boost factor

Searching through XML

OmniFind Text Search Server includes support for indexing and searching XML documents. OmniFind server support allows text searches to be scoped to a specific XML element, attribute or combination thereof. Scoped searches are possible because the OmniFind XML search syntax is based on the XPath standard. Scoped searches of XML documents can improve performance because they reduce the amount of text in the XML document that must be scanned.

To use the OmniFind XML support, a text index must first be created with a format option that supports XML. That format value not surprisingly is XML. Figure 20 contains an example of a text index that is created to support the indexing of XML documents. With this example, the XML documents are stored in a CLOB column (storeBooks) within a DB2 table. Notice that the invocation of the SYSTS_CREATE stored procedure specifies a format type of XML to create a text index for the XML documents that are stored in the DB2 table. The XML format type enables the text-search index to classify and process each XML tag and attribute in addition to the real business values contained in the XML document.

The INSO format value can process and index XML documents. The INSO format, however, does not retain any of the XML attribute or element tags in the text index. As a result, XML-specific searches cannot be performed when the format value is INSO.

```
CREATE TABLE mylib.books (
storeID INTEGER,
storezip CHAR(5) ,
storeBooks CLOB(500K),
PRIMARY KEY( storeID ) ) ;

CALL SYSPROC.SYSTS_CREATE (
  'MYLIB',
  'BOOKS_IX',
  'MYLIB.BOOKS(storeBooks)',
  'FORMAT XML UPDATE FREQUENCY D(*) H(*) M(0)'
 );
```

Figure 20.Example of a XML text index

You can perform XML-specific searches with the same CONTAINS and SCORE functions that have already been discussed. When a CONTAINS or SCORE search argument contains an XML search expression, special syntax is required to direct the OmniFind server to use the XPath parser instead of the normal parser. Activation of the XPath-based parser is accomplished by specifying the '@xpath' opaque term at the beginning of the search argument.

All XML search arguments that you specify on the CONTAINS function must have the format of '@xpath:"XML search string" '. You must also always wrap an XML search expression in single quote characters ('XML search string'). An extra set of single quotes is required on the CONTAINS function

because the XML search string is embedded within the search argument string. To include a single quote character(') within an SQL character string, you must repeat the single quote value two consecutive times. This is why the XML search string expression is delimited by two sets of single-quote characters.

The XML search syntax is documented in detail in the *IBM OmniFind Reference Manual* (http://publib.boulder.ibm.com/systems). A good understanding of XPath is required to understand and effectively use the XML search capabilities provided with the OmniFind product since it's based on the XPath standard. If you lack XPath knowledge, it is possible to find many XPath tutorials online. Several XML search examples are included to highlight key aspects of the OmniFind XML search syntax and expressions. All search examples assume that the sample XML file in Figure 21 is stored in the books table (discussed earlier).

```
<bookstore>
    <book genre="autobiography" publicationdate="2008-05-01"
          ISBN="1-414318-02-2" price="14.99">
        <title>Quiet Strength</title>
        <author>
            <first-name>Tony</first-name>
            <last-name>Dungy</last-name>
        </author>
        <publisher>Tyndale</publisher>
    </book>
    <book genre="inspiration" publicationdate="2000-02-01"
          ISBN="0-310234-93-X" price="7.99">
        <title>The Man in the Mirror</title>
        <author>
            <first-name>Patrick</first-name>
            <last-name>Morley</last-name>
        </author>
        <publisher>Zondervan</publisher>
    </book>
    <book genre="religion" publicationdate="1996-08-01"
          ISBN="0-842329-12-9" price="14.99">
        <title>Left Behind: A Novel of the Earth's Last Days</title>
        <author>
            <first-name>Tim</first-name>
            <last-name>LaHaye</last-name>
        </author>
        <publisher>Tyndale</publisher>
    </book>
   <book genre="sports" publicationdate="2003-08-01"
         ISBN="1-590521-550-2" price="19.99">
        <title>Racing to Win</title>
        <author>
            <first-name>Joe</first-name>
            <last-name>Gibbs</last-name>
        </author>
        <publisher>Doubleday</publisher>
    </book>
</bookstore>
```

Figure 21. Example of an XML file for searching

At a high level, the example XML search in Figure 22 returns store information for those stores that have the term 'winning' in any of their book titles. Let's examine each part of the XML search string to make sure it is fully understood. The XML search string is valid because it begins with the @xpath opaque term and the embedded XML search expression (//title[. contains("winning")) is delimited with two pairs of single quotation marks.

```
SELECT storeid, storezip FROM books WHERE
 CONTAINS(storebooks, '@xpath:''//title[. contains("winning")]'' ')=1
```

Figure 22. XML element search for "winning" book titles

As you might expect, the reference to the title element on the XML search expression constrains the search to only the values of the title elements within the XML document. The two consecutive slash values (//) denote that title elements at any level should be searched. If a single slash value had been specified with title (/title), the search is limited to only those XML documents that have a top-level element tag named *title*. This XML document in Figure 21 does not meet the top-level tag requirement because the title element is a nested child tag.

Note that the wildcard character (*) cannot be specified on any XML search path (for example, /root/partialtag*). All XML tag values must be specified in their entirety.

The next part of the search request in Figure 22 is the search predicate, which must always be wrapped in bracket characters ([…]). Search predicates are used to find a specific node or a node containing a particular value. Based on the XML search syntax in Figure 22, the requestor looks for a node element that contains the term *winning*. The period character (.) indicates that the value of the current XML node (the title node in this example) should be compared with the specified search argument. Thus, in written form, the complete search request states that OmniFind must find all title elements at any level and determine if any title element values contain the inputted search-string value (winning).

Also notice that the XPath-based search expressions use the *contains* operator instead of an = comparison operator. This enables the linguistic capabilities of the OmniFind search engine to be applied to the specified search string. The linguistic search engine enables the book titled "Racing to Win" in Figure 21 to be identified as a match for this XML search request, because the word *win* is a linguistic variation of winning. You can only specify the = comparison operator for XML attributes, and it behaves differently than expected — more on this later.

The XML *contains* operator supports almost the same search argument syntax as the CONTAINS function. For instance, an XML search request can find *winning* or *success* in the title by using the following search expression: //title[. contains("winning OR success")]. Along with logical operators, the XML *contains* operator can also include parentheses to control the precedence of the search strings. When you specify search-argument options (for example, SYNONYM and QUERYLANGUAGE) on the third parameter of the SQL CONTAINS function, those search options apply to the XML search request.

There are some subtle differences between the CONTAINS function and the XML *contains* operator. Two key differences are highlighted here. First, the XML *contains* operator uses a different delimiter for the search string. The delimiter is a pair of double-quote characters ("winning") instead of the single-quote character that is used on the CONTAINS function. If it is necessary for the XML search request to only return exact matches for the winning search argument, double-quote characters must be placed around the search argument, just as the SQL CONTAINS function requires. A double set of quotation marks (contains(""""winning"""")) is needed to differentiate these double-quotation marks from the single pair of double-quotation marks required by the XML *contains* operator.

Second, the XML search syntax provides an *exclude* operator, which you can view as the opposite of the XML *contains* operator (NOT contains). An XML search expression of '//title[. excludes("last")] returns a match for any XML document that has at least one title element that does not contain the search term 'last'. This search expression returns a match for the sample XML document in Figure 21 because three of the four title values do not contain the word *last*.

Figure 23 contains an example of how to search for a specific XML attribute. Unlike the previous XML search example, this request specifies that you conduct the attribute search at a specific level within the XML document. By specifying a path of '/bookstore/book', the search is limited to those XML documents that have the bookstore element at the top-level and a direct child-element of *book*. In addition, the search request looks for *book* elements with specific attribute values. The search predicate looks for a price attribute that is less than $15.00 or a *genre* attribute that is equal to autobiography. The search criteria for the two attributes are combined with the *or* logical operator. You must specify logical operators, such as *or* and *and*, in lower case when they are referenced on an XML search expression.

```
SELECT storeid, storezip FROM books
WHERE CONTAINS(storebooks,
 '@xpath:''/bookstore/book[@price<15.00 or @genre="autobiography"]'' ')=1
```

Figure 23.Example of an XML attribute search

This XML attribute search predicate uses the comparison operators (=,>,<,>=,<=,!=). You can only use comparison operators with XML attributes and they are not available for XML element comparisons. The comparison operators are available for comparisons with numeric values. The = operator does support string comparisons as the example in Figure 23 depicts, but you must use it carefully because this string-comparison operation might behave differently than expected.

When the = operator is applied to attribute strings, the comparison is case-insensitive and the order of terms is not enforced. For example, a search predicate value of @myattribute="first second" matches with @myattribute="FIRST SECOND", @myattribute="second first" and @myattribute="second middle first". Further, the = operator only looks for exact matches of the search words and does not consider linguistic variations. As noted earlier, linguistic variations are only referenced with the XML *contains* operator. Figure 24 shows the evaluation of the genre attribute implemented with the *contains* operator.

The XML search in Figure 24 also shows how to specify search criteria for both XML elements and attributes on a single request. This request looks for an XML document with a top tag of *bookstore* with a direct child element of *book*. The *book* element must contain the same attribute values specified in Figure 23 — a price attribute less than *$15.00* or a genre attribute value of *autobiography*. When a *book* element is found that meets the required attribute criteria, then the search request checks that *book* element for a direct child title element that contains *quiet* and *strength* in the book title. This XML search request would return a match on the book authored by Tony Dungy from Figure 21.

```
SELECT storeid, storezip FROM books
WHERE CONTAINS (storebooks,
'@xpath:''/bookstore/book[@price < 15.00 or
@genre contains("autobiography")]/title[. contains("quiet AND
strength")]'' ') = 1
```

Figure 24. Combined search of XML attributes and elements

You can also implement this combination search for specific XML attributes and elements with the search request found in Figure 25 that uses a single XML search predicate. The attribute-search criterion is slightly altered with the addition of parentheses to ensure that one of the attributes meets the specified criteria before checking the title element value. The reference to the title element on the book element search predicate dictates that the book element must have a direct child title element that contains *quiet* and *strength* in the specified search criteria.

```
SELECT storeid, storezip FROM books
WHERE CONTAINS (storebooks,
'@xpath:''/bookstore/book[(@price < 15.00 or
@genre contains("autobiography")) and title contains("quiet AND
strength")]'' ') = 1
```

Figure 25. Combined search of XML attributes and elements with a single predicate

When specifying XML tag and attributes on search requests, you need to know how OmniFind Text Search Server processes XML tags and attributes during the indexing process. First, the OmniFind server does not normalize the case of the XML tags during the indexing process. Thus, the case of the XML tags is significant when coding a search request. Secondly, the text-search server does not recognize name space references in XML documents when building text indexes. If a text index was built over the XML document in Figure 26, the text index entries would recognize the xml tag values as <book>,<title>, and <number>. There is no connection at all with the namespace prefix values of *bk* and *isbn*.

```
<bk:book xmlns:bk='urn:loc.gov:books'
         xmlns:isbn='urn:ISBN:0-395-36341-6'>
    <bk:title>Cheaper by the Dozen</bk:title>
    <isbn:number>1568491379</isbn:number>
</bk:book>
```

Figure 26. XML file with namespace references

For the best performance, use search arguments that are the most unique to the business data that the application is searching for within the indexed XML documents.

Searching with synonyms

As mentioned earlier in this paper, the OmniFind search server lets users supply synonyms to augment the linguistic-search capabilities of the text-search engine. Synonym-dictionary files can be useful when you want OmniFind Text Search Server to recognize synonyms that are unique to your industry or business.

As an example, consider the name changes to the computing platform that was originally known as an IBM AS/400 system. If you developed and sold application software for this computing platform, there is a good chance that you have product brochures, web pages and fact sheets that still reference the older platform names instead of the latest product name. By supplying a synonym file like the one found in Figure 27, all of your product searches can be performed with a single search string of 'IBM i'. OmniFind Text Search Server returns matches when it discovers any of the registered synonyms for IBM i in the target search text. For instance, a match is returned for the text string, "Distribution software for AS/400, iSeries, and System i", even though it contains no references to the latest name of IBM i.

```
<?xml version="1.0" encoding="UTF-8"?>
<synonymgroups version="1.0">
<synonymgroup>
<synonym>AS/400</synonym>
<synonym>AS 400</synonym>
<synonym>iSeries</synonym>
<synonym>System i</synonym>
<synonym>IBM i</synonym>
</synonymgroup>
</synonymgroups>
```

Figure 27. Example synonym file, mysyns.xml

Creating a synonym file is the first step in the process of enabling the OmniFind search engine to refer to user-defined synonyms. As Figure 27 shows, user synonyms are supplied in an XML file. This sample file contains a single group of synonyms, but multiple synonym groups can be included in a synonym file. The synonym value of 'AS 400' must be included for this example to work correctly because punctuation characters (/) are currently not supported in the synonym-dictionary files.

After a synonym file is created, you must associate the file with a text index by using a synonym tool. You can associate a synonym file with more than one text index. The synonym tool itself is actually a shell script named synonymTool.sh. As a shell script, the synonym tool will need to be invoked from the Qshell command line (STRQSH) or with the QSH CL command.

The synonym file association step must be performed after the initial update or build of the text index. When the synonym file association occurs before the initial population of the text index, the synonym file association is lost during the build processing. This also means that the synonym file association will be lost each time the text index is completely rebuilt using the SYSTS_REPRIMEINDEX stored procedure. The synonym file itself is not deleted when the association is lost. Thus, the synonym file association step will have to be re-executed each time the association is lost.

Before using the synonym tool, you must collect several pieces of information that are required as parameters. The tool requires the name of the text index collection and the file path of the text-search server. The easiest way to retrieve this information is by querying the text index database catalogs in QSYS2.

The file path for the text-search server is stored in the SERVERPATH column in the SYSTEXTSERVERS catalog table. Assume that the path value stored in the SERVERPATH column is '/QOpenSys/QIBM/ProdData/TextSearch/server1/bin/'. The collection name can be found in the SYSTEXTINDEXES catalog table. The collection is the IFS directory created by the OmniFind server to store the contents of the text index. The SELECT statement in Figure 28 returns the name of the collection (IFS directory) for the resumes_index text index created in Figure 4. The collection name for the example RESUMES_INDEX is 0_2_15_2009_06_17_13_59_25_386602.

```
SELECT collectionname FROM qsys2.systextindexes
WHERE indexschema='MYSCHEMA' AND indexname = 'RESUMES_INDEX'
```

Figure 28. Catalog query for collection name

With the two pieces of required information in hand, you can build the command string for the synonym tool. Figure 29 contains an example invocation of the synonym tool to associate the mysyns.xml synonym file with the resumes_index text index from Figure 4. Right away, you can see that the server file-path value is used at the start of the command string to identify the location of the synonym tool file (synonymTool.sh). The importSynonym argument denotes that the invoker wants to associate or add the synonym file to the text index. The synonymFile command argument identifies the location of the synonym xml file. The synonym file can reside in any IFS directory. The collectionName parameter identifies the name of the IFS directory (0_2_15_2009_06_17_13_59_25_386602) that contains the content of the text-index structure. The *replace* parameter supports two values. The true value used in this example replaces the text index's synonym dictionary with the contents of the specified synonym file. If you specify a value of '-replace false', the contents of the synonym file are appended to the existing synonym dictionary. The *append* option enables you to add parameters to an existing text index and synonym dictionary over time. The final parameter is the absolute file path for the text server's configuration directory. You can construct this value by taking the server path value and replacing the /bin subdirectory with a reference to the /config subdirectory.

```
/QOpenSys/QIBM/ProdData/TextSearch/server1/bin/synonymTool.sh
 importSynonym
  -synonymFile /QOpenSys/QIBM/ProdData/TextSearch/mysyns.xml
  -collectionName 0_2_15_2009_06_17_13_59_25_386602
  -replace true
  -configPath /QOpenSys/QIBM/ProdData/TextSearch/server1/config
```

Figure 29. Sample synonym tool command

You must run the synonym tool command string in Figure 29 from the IBM i Qshell command interface. If it is a requirement to perform setup from a CL program, you can run the synonym tool command string with the QSH CL command, as Figure 30 shows.

```
QSH
CMD('/QOpenSys/QIBM/ProdData/TextSearch/server1/bin/synonymTool.sh
 importSynonym
  -synonymFile /QOpenSys/QIBM/ProdData/TextSearch/mysyns.xml
  -collectionName 0_2_15_2009_06_17_13_59_25_386602
  -replace true
  -configPath /QOpenSys/QIBM/ProdData/TextSearch/server1/config')
```

Figure 30. CL command invocation of synonym tool

After the synonym import request completes, the only remaining step is to specify the SYNONYM search option when coding up search requests with the CONTAINS and SCORE function. Assume that the

human resources department needs to search the resumes that are indexed with a text index in Figure 4 to find applicants with IBM i systems experience. Instead of coding a separate query or a separate search predicate for each legacy name of IBM i, you can simply code the single query shown in Figure 31 with a single search predicate. This query finds any resume that contains the term IBM i or its synonyms (AS/400, iSeries, System i) because the SYNONYM search option is specified and a synonym dictionary has been added to the text index. If the 'SYNONYM=ON' option is not supplied, the query only matches those resumes containing IBM i references.

```
SELECT applicant_name, applicant_number FROM myschema.resumes
WHERE CONTAINS(applicant_resume,'IBM i','SYNONYM=ON') = 1
```

Figure 31. Search example with synonyms

Summary

Hopefully, you now have a good understanding of how the advanced, high-speed linguistic features of the OmniFind product can be leveraged by your application software to deliver additional value to your users. The extreme flexibility of OmniFind to index and search data in multiple formats whether it is stored inside or outside of DB2 for i provides many paths for extracting business value out of text data. In addition, the SQL-based search interfaces enable you to integrate the OmniFind search capabilities into a wide array of languages and interfaces from RPG to Java to DB2 Web Query. The possibilities are almost endless.

You can find additional references regarding the OmniFind product in the *Resources* section.

Resources

These Web sites provide useful references to supplement the information contained in this document:

- IBM OmniFind Reference Manual (found in IBM System i and IBM i Information Center, search for OmniFind)
  http://publib.boulder.ibm.com/systems
- Article: OmniFind Can Help Improve Your Search Technique
  http://www.ibmsystemsmag.com/ibmi/october08/technicalcorner/22018p1.aspx
- Webcast: Exploring the IBM OmniFind Text Search Server
  http://www.centerfieldtechnology.com/files/KM2/KM2.htm
- Article: Integrating OmniFind with DB2 Web Query
  http://mcpressonline.com/database/db2/omnifind-part-ii-integrating-omnifind-text-search-server-with-db2-web-query.html
- IBM System i and IBM i Information Center
  http://publib.boulder.ibm.com/systems
- IBM Systems on PartnerWorld
  ibm.com/partnerworld/systems
- Virtual Loaner Program
  ibm.com/systems/vlp
- IBM Redbooks®
  ibm.com/redbooks

About the author

Kent Milligan is a Senior Certified IT specialist in IBM Systems and Technology Group ISV Enablement, working with IBM i software vendors. Prior to joining the ISV Enablement team in 1997, Kent spent the first eight years of his IBM career as a member of the DB2 development team in Rochester, Minnesota. He speaks and writes regularly on DB2 and other relational-database topics. You can reach Kent at kmill@us.ibm.com.

Trademarks and special notices

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.