Surviving Client/Server: Indexing Freeform Text

by Steve Troxell

Suppose you had a medical database containing a memo field for patient symptoms. How would you find all the cases admitted with convulsions? Or suppose you had a legal database containing depositions from dozens of witnesses for a certain case. How would you find all the witnesses who saw Mr Smith?

When our tables contain discrete fields, it's easy for us to index those fields and select records matching certain field values. But in the case of freeform string fields or memos, database systems generally do not provide a direct way of finding specific text within the field.

One approach is to mimic the Windows help system and store a set of keywords linked to the memo containing those keywords. Then we just search the keyword list to find the corresponding memo records. This task becomes a little more complicated as we add capabilities to do some limited Boolean algebra with operators such as `AND`, `OR` and `NOT`. We really begin to scratch our heads when we consider proximity matches like the `NEAR` operator, which means "x and y when they are within n words of each other." Finally, we must consider how we are going to find a phrase of any length within the text without scanning each memo one by one for each query. Oh, and you have to do all this in SQL.

We'll see how to do all this, and the techniques are not specifically tied to indexing text. They can be used with any master-detail relationship where we need to locate master records containing a certain combination and/or sequence of detail records.

Basic Keyword Indexing

For our first attempt at producing a keyword index, we will consider a "data table" containing a single memo field to be indexed. We'll use the `BIOLIFE` table from the `DBDEMOS` example database. This table contains a freeform text field called `Notes`. We will allow the user to enter an expression such as you might enter in any Internet search engine. For example, `florida and beach`. We will support the conjunctive operators `AND`, `OR` and `NOT` to retrieve records conforming to the query, and we will only consider single word operands. The data requirements for this type of query are simple: for each memo field we wish to index, we need a list of keywords and identifiers for the records containing each keyword. Figure 1 shows the definition of our "index table." The data table can be considered the master table, and the index table can be considered the detail table.

First we will need to populate the index table from the data table. The fundamental process for this is to take a string of text and produce a list of all the words within that text, then save this list to the index table. Generally we would want to omit all the minor words such as `the`, `and`, `a`, `this` and so on, so we would like to be able to exclude certain words to keep down the size of the index table.

Ideally, we would want the keywords updated automatically each time a record with a memo field was added, or the memo was updated. This begs for insert and update triggers on the table, but to use SQL to parse a memo and produce a list of unique keywords would be extremely difficult. We will rely on our Delphi application to detect that a memo requires keyword indexing and perform the task for us. However, many database servers allow us to link DLL procedures or functions into SQL, so we may be able to use the power of Delphi to parse the memo and still link it into a database trigger.

```
Table BiolifeIdx1
FieldName          Datatype   Size
═══════════════    ═══════    ════
Keyword            Varchar    50
RecordID           Integer
Primary Index: Keyword, RecordID
Secondary Index: RecordID
```

➤ *Figure 1*

TMemoScanner

Listing 1 shows the `TMemoScanner` class which parses a given memo field and returns a list of all the unique keywords found in the memo. We can then write these keywords to the index table. `TMemoScanner` derives from Delphi's `TBLOBStream` class, which simplifies streaming data out of a BLOB field in a dataset.

Our memo scanner defines keywords as any text delimited by white space or punctuation marks. However, we make allowances for hyphenated words and words forming the possessive with a trailing `'s`. We compare each keyword against a predefined list of "discard words" (the, and, this, etc), throw out any matches we find, and return the rest in a string list. The scanner is bound to a memo field when it is created, then the Scan method is called to parse the memo. Upon return, the keywords can be retrieved with the `Keywords` property.

So how does the scanner work? Looking at the `Scan` method: we read the memo as a stream one character at a time. We can improve performance considerably by reading the stream in "chunks" into a buffer and then scanning the buffer one character at a time. As long as we fail to find a delimiting character, we accumulate the text as a keyword. Once we do find a delimiting character, we

compare our keyword to the discard list. If a match is not found we add the keyword to our internal list. Duplicate words are automatically eliminated from the list. Note that we are calculating word offsets for the position of the keyword within the memo, but this is not used currently. We'll take advantage of this information later on.

Using TMemoScanner to create the index table should be fairly obvious. For each memo, we instantiate a TMemoScanner, call Scan, delete any existing keywords in the index table and write the contents of TMemoScanner.Keywords to the index table.

Retrieving Expression Matches

Now that we have keyword information to draw upon, how do we make use of this information in our applications? For our purposes here we'll restrict ourselves to expressions involving only one term, or two terms joined by AND, OR or NOT. Realistically we would allow any number of terms joined by any combination of operators, but the parsing and expression tree logic would quickly get beyond the scope of this article. For the moment we'll further restrict ourselves to single word terms. Later in the article we'll handle terms of multiple words.

The easiest approach for us to take is to examine the expression entered by the user and dynamically generate a custom SQL statement on the fly to find the matches in the keyword table and join with the data table. Let's take a look at how the SQL needs to be constructed to support our different operators.

In essence we need to produce a list of all the unique record numbers of memos containing keywords that conform to the search expression. Then retrieve the relevant data from the data table for those records. Let's say we wanted all records referring to California. For a simple single word search expression, it's obvious that the SQL shown in Listing 2 will tell us which records have memos containing that word, keeping in mind that the list of keywords for a single memo does not duplicate. This becomes a subquery which we then use to locate and retrieve records from the BIOLIFE table, as shown in Listing 3. All of our queries will have this same "wrapper" query around them, and from this point on, we'll only concern ourselves with the inner query which identifies the record numbers to fetch.

Our expression engine simply needs to alter the subquery in accordance with the operator

➤ *Listing 1*

```
const
  MaxBufferSize = 1024;
type
  TMemoScanner = class(TBLOBStream)
  private
    Buffer: array[1..MaxBufferSize] of Char;
    Punctuation: string;
    WhiteSpace: string;
    DiscardWords: TStringList;
  protected
    FKeywords: TStringList;
    procedure DefineDiscardWords(aList: TStrings); virtual;
    function DefinePunctuation: string; virtual;
    function DefineWhitespace: string; virtual;
    function GetKeyword(aIndex: Integer): string;
    function GetKeywordCount: Integer;
    procedure KeywordFound(aKeyword: string;
      aWordOffset: Integer); virtual;
  public
    constructor Create(aField: TBLOBField);
    destructor Destroy; override;
    procedure Scan;
    property KeywordCount: Integer read GetKeywordCount;
    property Keywords[aIndex: Integer]: string
      read GetKeyword;
  end;
implementation
constructor TMemoScanner.Create(aField: TBLOBField);
begin
  inherited Create(aField, bmRead);
  FKeywords := TStringList.Create;
  FKeywords.Sorted := True;
  FKeywords.Duplicates := dupIgnore;
  DiscardWords := TStringList.Create;
  DiscardWords.Sorted := True;
  DiscardWords.Duplicates := dupIgnore;
  DefineDiscardWords(DiscardWords);
  Punctuation := DefinePunctuation;
  WhiteSpace := DefineWhiteSpace;
end;
destructor TMemoScanner.Destroy;
begin
  DiscardWords.Free;
  inherited Destroy;
end;
procedure TMemoScanner.DefineDiscardWords(aList: TStrings);
begin
  { There are various methods for implementing the lookup
    list. A hash table might be faster. }
  with aList do begin
    Add('A');
    Add('ALL');
    Add('AN');
    Add('AND');
    {...list truncated for brevity...}
  end;
end;
function TMemoScanner.DefinePunctuation: string;
begin
  { we specifically omit the hyphen and apostrophe }
  Result := '~'!@#$%^&*()+={}[]|\:;"<>,.?/';
end;
function TMemoScanner.DefineWhiteSpace: string;
begin
  Result := #32#8#9#13#10;
end;
function TMemoScanner.GetKeyword(aIndex: Integer): string;
begin
  Result := FKeywords[aIndex];
end;
function TMemoScanner.GetKeywordCount: Integer;
begin
  Result := FKeywords.Count;
end;
procedure TMemoScanner.KeywordFound(aKeyword: string;
  aWordOffset: Integer);
begin
  FKeywords.Add(aKeyword);
end;
procedure TMemoScanner.Scan;
var
  Ch: Char;
  Keyword: string;
  I: Integer;
  BufLen: Integer;
  WordOffset: Integer;
begin
  FKeywords.Clear;
  Keyword := '';
  Position := 0;
  WordOffset := -1;
  while Position < Size do begin
    BufLen := Read(Buffer, SizeOf(Buffer));
    for I := 1 to BufLen do begin
      Ch := UpCase(Buffer[I]);
      { Is it a keyword delimiter? }
      if Pos(Ch, WhiteSpace + Punctuation) <> 0 then begin
        if Length(Keyword) <> 0 then begin
          Inc(WordOffset);  { count words in text }
          if DiscardWords.IndexOf(Keyword) = -1 then
            KeywordFound(Keyword, WordOffset);
          Keyword := '';
        end;
      end else
        {accumulate current keyword}
        Keyword := Keyword + Ch;
    end;
  end;
end;
```

```
SELECT RecordID FROM BiolifeIdx1
  WHERE Keyword = "California"
```

➤ *Listing 2*

we've chosen. Listing 4 shows how we would handle the OR operator. We use the DISTINCT operator to eliminate any duplicates produced by the same memo having both keywords.

For the AND operator we find all the records containing one search word, then for each of those we check to see if the other search word can be found in the same memo. To do this we can use a correlated subquery. A correlated subquery is two nested queries where values from each row of the outer query are passed into the inner query which is then executed using those values. The result of the inner query may impact the row in the outer query.

What we will do is make our outer query produce a list of memos containing the first search word, then for each one we execute an inner query to see if the other search word is present in that memo. Listing 5 shows how we do this. The inner query obtains the values from the outer query by referencing the outer query's table. It then plugs those values into itself and runs itself. This is repeated for each row found by the outer query. If the inner query fails to find the second search word, then it returns an empty result set and the boolean function EXISTS returns False. Handling the NOT operator is essentially the same and is also shown in Listing 5.

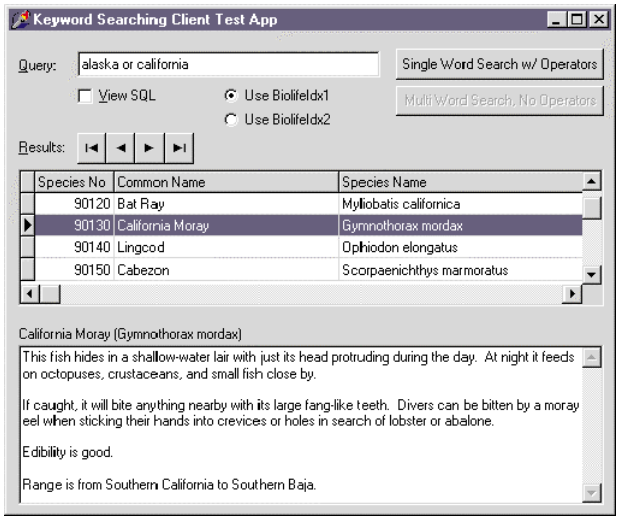
Putting It Into Practice

Figure 2 shows a demo program which puts all this together. This demo can be found on the disk accompanying this issue. You'll have to create the BIOLIFEIDX1 index table (see Figure 1) in DBDEMOS, but you'll find a utility on the disk to populate the table.

To use the demo, enter the search terms and click the Single Word Search button. The BIOLIFE records containing a Notes field that matches the expression are returned in the grid. If we double click on an entry in the grid, the Notes field for that record is displayed in the memo control below. We simply write code to assemble the SQL query as we have discussed and put it into a TQuery control. If View SQL is checked, then clicking the Single Word Search button shows the SQL generated for the given search terms.

The SELECT statement returning the records matching the expression only returns the name fields and the primary key field. We could have returned the memo itself within this result set. However, a more realistic approach in client/server is to return the minimum amount of information to allow the user to select the record they are really interested in, then grab the full details of the record only when they request it. If we linked the memo to the result set grid, there could be noticeable delay and unnecessary load on the network as the user scrolled through records while the memo fields were fetched from the server for each record.

➤ *Figure 2*




```
SELECT Biolife."Species No", Common_Name, Biolife."Species Name"
  FROM Biolife
  WHERE Biolife."Species No" IN
    (SELECT RecordID FROM BiolifeIdx1
     WHERE Keyword = "California")
```

➤ *Listing 3*

```
/* kwd OR kwd */
SELECT DISTINCT RecordID FROM BiolifeIdx1
  WHERE Keyword = "California" OR Keyword = "Alaska"
```

➤ *Listing 4*

```
/* kwd AND kwd */
SELECT RecordID FROM BiolifeIdx1 A
  WHERE Keyword = "California" AND EXISTS(
    SELECT RecordID FROM BiolifeIdx1 B
      WHERE A.RecordID = B.RecordID AND
            B.Keyword = "Alaska")
/* kwd NOT kwd */
SELECT RecordID FROM BiolifeIdx1 A
  WHERE Keyword = "California" AND NOT EXISTS(
    SELECT RecordID FROM BiolifeIdx1 B
      WHERE A.RecordID = B.RecordID AND
            B.Keyword = "Alaska")
```

➤ *Listing 5*

Words NEAR Other Words

The NEAR operator allows you to look for text containing two words that are physically close to each other in the text, for example within 8 words of each other. Obviously our index table format will

have to change to include information about the position of the indexed keywords. Figure 3 shows our new table layout.

Obviously, with this layout, duplicate keywords per indexed record will be introduced and the table will be proportionately larger because of this. All the SQL queries we used to support the AND, OR and NOT operators will still work on this table as long as we make sure to use DISTINCT in the select list to eliminate duplicate record numbers.

To create this index table, we make a new descendant of TMemoScanner which preserves the positional information. Listing 6 shows our extended memo scanner class. Basically all we do is change the keyword list to allow duplicates, override the KeywordFound method to retain the positional information being computed and provide a property to retrieve that information. Note that although we discard words found in the discard list the WordOffset still shows the correct word position within the original memo.

Now all we have to do is write an SQL statement that is essentially the same as our AND query, but also takes into account the "nearness" of the words. We simply make a comparison of the word offsets of the two search terms a condition of the inner query. Listing 7 shows an SQL statement to handle NEAR. The constant 8 in the offset calculation means the words have to be within 8 words of each other. Note that if our SQL dialect had an absolute value function, we should replace the last two terms of the WHERE clause with the single term:

```
ABS(A.WordOffset -
  B.WordOffset) <= 8
```

Phrase Matching

The next technique we will consider is matching an entire phrase of text rather than just single keywords. First we will look at the case where we are asked to find all memos containing a single multi-word phrase, without the complications of the AND, OR, NOT and NEAR operators. Then we will progress to the more complicated topic of using the conjunctive operators with phrases.

When allowing phrase matches, we should not discard "trivial" words as we've defined in the DefineDiscardWords method shown in Listing 1. To create a scanner that does not discard words, we just create a descendant like we've done in Listing 6, but we also make an override of DefineDiscardWords that does nothing.

A phrase match means we are looking for a specific series of words in a specific sequence. Broken down into abstractions, we have nothing more than a master-detail relationship (memos to keywords within the memos) and we are searching for all master records containing a specific combination of detail records in a specific sequence. In principle, our index table tells us everything we need to know to find a match. Once we find the first word of the phrase, we must also find the second word within the same memo and the

```
Table: BiolifeIdx2
FieldName          Datatype  Size
================== ========= ====
Keyword            Varchar   50
RecordID           Integer
WordOffset         Integer
Primary Index: Keyword, RecordID, WordOffset
Secondary Index: RecordID
```

➤ *Figure 3*

```
type
  TMemoScannerExt = class(TMemoScanner)
  protected
    function GetWordOffset(aIndex: Integer): Integer;
    procedure KeywordFound(aKeyword: string;
      aWordOffset: Integer); override;
  public
    constructor Create(aField: TBLOBField);
    property WordOffset[aIndex: Integer]: Integer read GetWordOffset;
  end;
constructor TMemoScannerExt.Create(aField: TBLOBField);
begin
  inherited Create(aField);
  FKeywords.Duplicates := dupAccept;
end;
function TMemoScannerExt.GetWordOffset(aIndex: Integer): Integer;
begin
  Result := Integer(FKeywords.Objects[aIndex]);
end;
procedure TMemoScannerExt.KeywordFound(aKeyword: string; aWordOffset: Integer);
begin
  FKeywords.AddObject(aKeyword, Pointer(aWordOffset));
end;
```

➤ *Listing 6*

```
/* kwd NEAR kwd */
SELECT DISTINCT RecordID FROM BiolifeIdx2 A
  WHERE Keyword = "California" AND EXISTS(
    SELECT RecordID FROM BiolifeIdx2 B
      WHERE A.RecordID = B.RecordID AND
            B.Keyword = "Alaska" AND
            B.WordOffset - A.WordOffset <= 8 AND
            B.WordOffset - A.WordOffset >= -8)
```

➤ *Listing 7*

```
SELECT RecordID FROM BiolifeIdx2 A
  WHERE Keyword = "edibility" AND EXISTS
    (SELECT RecordID FROM BiolifeIdx2 B
       WHERE A.RecordID = B.RecordID AND
             ((B.Keyword = "is" AND
               B.WordOffset = A.WordOffset + 1) OR
              (B.Keyword = "excellent" AND
               B.WordOffset = A.WordOffset + 2))
       GROUP BY RecordID
       HAVING COUNT(*) = 2)
```

➤ *Listing 8*

```
Expression: "edibility is excellent" AND grunt-like

/* find records containing "edibility is excellent" */
SELECT RecordID INTO #Temp FROM BiolifeIdx2 A
  WHERE Keyword = "EDIBILITY" AND EXISTS
    (SELECT RecordID FROM BiolifeIdx2 B
       WHERE A.RecordID = B.RecordID AND
             ((B.Keyword = "IS" AND
               B.WordOffset = A.WordOffset + 1) OR
              (B.Keyword = "EXCELLENT" AND
               B.WordOffset = A.WordOffset + 2))
       GROUP BY RecordID
       HAVING COUNT(*) = 2)
/* find records containing "grunt-like" */
INSERT INTO #Temp
  SELECT RecordID FROM BiolifeIdx2 WHERE Keyword = "GRUNT-LIKE"
/* find duplicates */
SELECT RecordID FROM #Temp
  GROUP BY RecordID
  HAVING COUNT(*) > 1
DROP TABLE #Temp
```

➤ *Listing 9*

second word's offset must be 1 greater than that of the first word. Then we must find the third word in the same memo, with an offset 2 greater than that of the first word, and so on through all the words in the phrase.

Seems like a fairly daunting task for SQL doesn't it?

In the BIOLIFE table, the Notes field frequently makes mention of each species' edibility, such as "edibility is good" or "edibility is poor." Suppose we were searching for the phrase "edibility is excellent"; we would find two matches in the BIOLIFE table for French Grunt (Species No 90220) and Yellow Jack (Species No 90260).

To find these matches we're obviously going to have to break our phrase into its constituent words. We need to isolate the first word since the offsets of all other words are compared relative to the first word's offset (the offset of the Nth word must equal the first word's offset + N - 1).

Let's refer to the first phrase word as the anchor word. If we had a set of all the keyword records matching the anchor word, we could take this list and check for the existence of the remaining phrase words in the same memo with the correct offsets from the anchor word.

Kind of sounds like a correlated subquery doesn't it? Basically we have a very specialized AND expression: find all the memos containing the anchor word as well as all the following keywords in sequence. Starting with the SQL statement we wrote for the AND operator (Listing 5), we extend the inner query to look for matches on all the remaining phrase words with the added condition that their offsets must align properly with the anchor word. Listing 8 shows the SQL statement we'll used to find the phrase "edibility is excellent."

The inner WHERE clause alone is not sufficient to find phrase matches accurately. Note carefully the OR operator between the filters for the keyword "is" and the keyword "excellent". Each keyword is in a separate detail record, so we must find the records that are one keyword or the other. This logic will also pick up partial phrase matches such as "edibility is rotten" and "edibility ain't good." In each case at least one of the inner keywords is matched by name and position, but we have not guaranteed that all the phrase words are present and in the right position.

We solve this with the HAVING clause (and by necessity the GROUP BY clause). We find that a complete phrase match results in the inner query finding exactly N - 1 rows where N is the number of words in the search phrase.

We can count the number of rows returned by the inner query by grouping them with GROUP BY and HAVING. HAVING throws out all groups of records not meeting the given condition. Since all the matching keyword records will have the same RecordID by definition of the join operation, we will always have exactly one group in the inner query result set. Therefore, the HAVING clause forces the inner query result set to be empty if we don't have the right number of rows. This eliminates false positive hits when there is only a partial match of the phrase.

What if more than one partial phrase match existed in the memo that coincidentally resulted in the correct number of rows being returned? What if "edibility is rotten" and "edibility ain't good" both appeared in the same memo: wouldn't this make the inner query return two rows and produce a false positive? No, it wouldn't, because each phrase will have an anchor word with a different word offset value. This means they will be separate rows in the outer query, each processed individually by the inner query. Therefore, it is impossible for the inner query to find more than one partial phrase match.

The Mother Of All Keyword Searches

OK, so far we have located memos based on the presence of a single multiword phrase. And we've located memos based on algebraic operators between single word operands. The next logical extension is to allow operators between phrases.

This becomes a bit tougher to tackle within the limitations of Local SQL in Delphi and we must turn to temporary tables on the server. In essence, when a phrase is present in the expression, we run a query to find the phrase matches, run a query to find the other term of the expression, and then combine the two result sets in a fashion consistent with the operator between the terms. Note that "the other term" could also be a phrase.

For example, the expression shown in Listing 9 is a phrase ANDed with a single word. We take our phrase-finding query from Listing 8 and put its result set in a temporary table. Then we take our single-word-finding query from Listing 2 and put its result set in the

```
Expression: "edibility is excellent" NOT grunt-like

/* find records containing "edibility is excellent" */
SELECT RecordID INTO #Temp FROM BiolifeIdx2 A
  WHERE Keyword = "EDIBILITY" AND EXISTS
    (SELECT RecordID FROM BiolifeIdx2 B
       WHERE A.RecordID = B.RecordID AND
             ((B.Keyword = "IS" AND
               B.WordOffset = A.WordOffset + 1) OR
              (B.Keyword = "EXCELLENT" AND
               B.WordOffset = A.WordOffset + 2))
       GROUP BY RecordID
       HAVING COUNT(*) = 2)
/* delete records containing "grunt-like" */
DELETE FROM #Temp WHERE RecordID IN
  (SELECT RecordID FROM BiolifeIdx2 WHERE Keyword = "GRUNT-LIKE")
SELECT RecordID FROM #Temp
DROP TABLE #Temp
```

➤ *Listing 10*

same temporary table. Then all we have to do is find the records that duplicate in the temporary table. Again, we can use `GROUP BY` and `HAVING` in order to isolate duplicate records.

Listing 9 shows a Microsoft SQL Server query to do just this, copying data into the temporary table `#TEMP`. The `SELECT INTO` construct creates a temporary table with a structure matching the columns returned from the `SELECT` statement and copies its result set into the temporary table.

The other operators are handled similarly. For `OR`, you would simply return all the distinct values in the temporary table. The `NOT` operator is a little trickier. We select all the records for the first term, but then we remove all those records matching the second term. What's left is our answer. We use the same approach as Listing 9, but instead of adding the second term's records to the temporary table, we remove any matches to those records from the temporary table. Listing 10 shows the `NOT` query.

Conclusion

We've seen a relatively simple way to query freeform text within a database. But more importantly, we've seen how a task, which at first glance seems impossible for SQL, actually can be handled quite nicely with a little study and SQL know-how. But that's what we're here for right?

I'd like to thank Dave Jewell for his help in initially brainstorming the indexing approach. Since Dave dislikes database programming *[that's the understatement of the year! Editor]*, I had to trick him into talking about indexing text "like Windows Help does." No hard feelings Dave?

Next month we'll begin looking at multi-tiered database applications. What are they all about, and how can Delphi help us build them?

Steve Troxell is a software engineer with Ultimate Software Group in the USA. He can be contacted via email at Steve_Troxell@USGroup.com