http://www.sqlservercentral.com/articles/Full-Text+Search+(2008)/64248/
Printed 2010/04/22 12:19PM

A Google-like Full Text Search

By Michael Coles, 2008/08/31

Thanks to Internet search engines like Google and Yahoo!, your search application users are more sophisticated and demanding than ever. Your application users have a wealth of search application knowledge that they're probably not even aware of. Ask one of them to find a website dedicated to their favorite band or movie and odds are they'll have their browser pointed to that page in a matter of seconds.

You can tap into the self-education that comes from countless hours spent at home trying to locate information on the Web and turn it to your advantage in your own SQL Server-based search applications. With just a little bit of code you can help reduce your training costs and give your users an easy to use interface that will make them want to use your search applications. In this article I'll explain how to convert Google-style queries to SQL Server's full-text search `CONTAINS` predicate syntax.

Google Style

The key to a successful application is to make it easy to use but powerful. Google has done this with their Web search engine. The syntax for queries is simple and intuitive, but full-featured. Though the basic building blocks of a Google query are simple you can combine them in powerful ways. I'll begin with basic Google query syntax and add some additional operators to take advantage of the power of SQL Server `CONTAINS` predicate syntax. The full Google syntax is defined in the *Google Help:Cheat Sheet* at http://www.google.com/help/cheatsheet.html.

For starters, I'll take as much as I can from the Google syntax, and then add a few SQL Server specific features. The final syntax will include the following features:

Although I tried to remain faithful to the Google syntax, you'll notice some slight variation. Where I've deviated I've done so for good reason (mostly, anyway):

- I added some features to take advantage of SQL Server FTS-specific functionality, like the proximity search operator.
- I had to modify the operation of some features like the * wildcard to fit with SQL Server's FTS query requirements.
- Some features I just added because it seemed like a good idea and made sense, such as the (optional) explicit `AND` operator and parentheses.

The Grammar

The search engine grammar that implements the syntax shown in the Google to FTS Cheat Sheet above can be defined using Extended Backus-Naur Form (EBNF). In EBNF grammatical constructs, known as productions, are written out using the following format:

```
lhs ::= rhs
```

In this format the lhs (left-hand side) is the name of the production. The rhs (right-hand side) consists of the terminals and nonterminals that make up the production. The `::=` symbol itself can be read as meaning "consists of", so that the production `Query ::= OrExpression` means that the `Query` production consists of an `OrExpression` nonterminal. The `|` symbol on the right-hand side indicates a choice of terminals and nonterminals that the production can consist of.

I've taken a few liberties with the EBNF grammar form for readability purposes, but the following holds true: (1) parentheses can be used in EBNF to indicate a group of terminal and nonterminal choices on the right-hand side; (2) braces are used to indicate optional components; (3) single quotes indicate literal characters or strings; (4) curly braces indicate optional terminals or nonterminals; and (5) the + sign following a terminal, nonterminal, or group indicates one or more occurrences.

```
Query ::= OrExpression

OrExpression ::= AndExpression
                 | OrExpression ( 'or' | '|' ) AndExpression

AndExpression ::= PrimaryExpression
                  | AndExpression { 'and' | '&' } PrimaryExpression
```

```
PrimaryExpression ::= Term
                    | '-' Term
                    | Phrase
                    | '(' OrExpression ')'
                    | '<' ( Term | Phrase )+ '>'

ExactExpression ::= '+' Term
                  | '+' Phrase

Term ::= ('A'...'Z'|'0'...'9'|'!'|'@'|'#'|'$'|'%'|'^'|'*'|'_'|'’'|'.'|'?')+

Phrase ::= '"' (string characters)+ '"'
```

Describing the EBNF grammar is an important step to formally define your language. Fortunately for us the EBNF description of this search engine grammar is nice and simple: it consists of 7 rules (5 nonterminals and 2 terminals). By way of contrast the EBNF for the SQL language consists of hundreds of production rules. Now that we have the grammar defined, it's time to turn our attention to the implementation.

Conversion Engine

For the conversion engine I decided to use Roman Ivantsov's Irony .NET Compiler Construction Kit, freely available for download from http://www.codeplex.com/irony. I chose Roman's Irony toolkit over other lexical analyzer/parser solutions for a few reasons:

1. Irony is built in C# and optimized for .NET; you can create a grammar as a .NET class.
2. Irony is much easier to use for simple grammars than some of the other tools available, like Yacc and Lex, Gold Parser, and others.
3. Irony automatically generates an optimized abstract syntax tree (AST), where other parsers require you to build your own. This was a huge timesaver.

The conversion engine is really pretty simple. It consists of one class called `SearchGrammar`. The `SearchGrammar` class contains the grammar definition in its constructor and exposes a class that walks the AST, converting the Google-style query string to a SQL Server FTS `CONTAINS` query. The `SearchGrammar` constructor defines the terminals and nonterminals that make up the Google-style grammar. As you can see in the listing below, it closely resembles the standard Extended Backus-Naur Form (EBNF) grammar form that it was derived from.

```
public SearchGrammar()
{
        // Terminals
        var Term = new IdentifierTerminal
        (
                "Term",
                "!@#$%^*_'.?",
                "!@#$%^*_'.?0123456789"
        );

        Term.Priority = Terminal.LowestPriority;
        var Phrase = new StringLiteral("Phrase");

        // NonTerminals
        var OrExpression = new NonTerminal("OrExpression");
        var OrOperator = new NonTerminal("OrOperator");
        var AndExpression = new NonTerminal("AndExpression");
        var AndOperator = new NonTerminal("AndOperator");
        var ExcludeOperator = new NonTerminal("ExcludeOperator");
        var PrimaryExpression = new NonTerminal("PrimaryExpression");
        var ThesaurusExpression = new NonTerminal("ThesaurusExpression");
        var ThesaurusOperator = new NonTerminal("ThesaurusOperator");
        var ExactOperator = new NonTerminal("ExactOperator");
        var ExactExpression = new NonTerminal("ExactExpression");
        var ParenthesizedExpression = new NonTerminal("ParenthesizedExpression");
        var ProximityExpression = new NonTerminal("ProximityExpression");
        var ProximityList = new NonTerminal("ProximityList");

        this.Root = OrExpression;
        OrExpression.Rule = AndExpression
                | OrExpression + OrOperator + AndExpression;
        OrOperator.Rule = Symbol("or") | "|";
        AndExpression.Rule = PrimaryExpression
                | AndExpression + AndOperator + PrimaryExpression;
        AndOperator.Rule = Empty
                | "and"
                | "&"
                | ExcludeOperator;
```

```
            ExcludeOperator.Rule = Symbol("-");
            PrimaryExpression.Rule = Term
                    | ThesaurusExpression
                    | ExactExpression
                    | ParenthesizedExpression
                    | Phrase
                    | ProximityExpression;
            ThesaurusExpression.Rule = ThesaurusOperator + Term;
            ThesaurusOperator.Rule = Symbol("~");
            ExactExpression.Rule = ExactOperator + Term
                    | ExactOperator + Phrase;
            ExactOperator.Rule = Symbol("+");
            ParenthesizedExpression.Rule = "(" + OrExpression + ")";
            ProximityExpression.Rule = "<" + ProximityList + ">";
            MakePlusRule(ProximityList, Term);

            RegisterPunctuation("<", ">", "(", ")");

    }
```

The `ConvertQuery` function accepts the AST generated by the grammar as input and recursively walks the AST starting with the root node. This is a very simple method of operation which works well for a simple grammar. The listing below is the AST walker. Notice that some special conditions, such as the wildcard * operator are better handled by the AST walker instead of trying to handle them in the grammar definition itself.

```
    public enum TermType
    {
            Inflectional = 1,
            Thesaurus = 2,
            Exact = 3
    }

    public static string ConvertQuery(AstNode node, TermType term)
    {
            string result = "";
            switch (node.Term.Name)
            {
                    case "OrExpression":
                            result = "(" + ConvertQuery(node.ChildNodes[0], type) + " OR " +
                            ConvertQuery(node.ChildNodes[2]) + ")";
                            break;

                    case "AndExpression":
                            AstNode tmp2 = node.ChildNodes[1];
                            string opName = tmp2.Term.Name;
                            string andop = "";

                            if (opName == "-")
                            {
                                    andop += " AND NOT ";
                            }
                            else
                            {
                                    andop = " AND ";
                                    type = TermType.Inflectional;
                            }
                            result = "(" + ConvertQuery(node.ChildNodes[0], type) + andop +
                            ConvertQuery(node.ChildNodes[2]) + ")";
                            type = TermType.Inflectional;
                            break;

                    case "PrimaryExpression":
                            result = "(" + ConvertQuery(node.ChildNodes[0], type) + ")";
                            break;

                    case "ProximityList":
                            string[] tmp = new string[node.ChildNodes.Count];
                            type = TermType.Exact;
                            for (int i = 0; i < node.ChildNodes.Count; i++)
                            {
                                    tmp[i] = ConvertQuery(node.ChildNodes[i], type);
                            }
                            result = "(" + string.Join(" NEAR ", tmp) + ")";
                            type = TermType.Inflectional;
                            break;

                    case "Phrase":
                            result = '"' + ((Token)node).ValueString + '"';
                            break;
```

```
                case "ThesaurusExpression":
                        result = " FORMSOF (THESAURUS, " +
                                ((Token)node.ChildNodes[1]).ValueString + ") ";
                        break;

                case "ExactExpression":
                        result = " \"" + ((Token)node.ChildNodes[1], type).ValueString + "\" ";
                        break;

                case "Term":
                        switch (type)
                        {
                                case TermType.Inflectional:
                                        result = ((Token)node).ValueString;
                                        if (result.EndsWith("*"))
                                                result = "\"" + result + "\"";
                                        else
                                                result = " FORMSOF (INFLECTIONAL, " +  result + ") ";
                                        break;

                                case TermType.Exact:
                                        result = ((Token)node).ValueString;
                                        break;
                        }
                        break;

                default:
                        throw new ApplicationException("Converter failed: unexpected term: " +
                        node.Term.Name + ". Please investigate.");

        }
        return result;
}
```

The Output

The conversion engine accepts a Google-style query as input and outputs a SQL Server FTS-style `CONTAINS` query. Once that's done you can open a database connection and execute the `CONTAINS` query against your database. For this function I've added the `ExecuteQuery` method to my `SearchGrammar` class. This method accepts the FTS query string generated by the `ConvertQuery` function as a parameter, opens a database connection and executes the full-text query. In the listing below I execute the query against the `Production.Document` table in the AdventureWorks sample database. I chose this table because it's already got a full-text index built on it - one less thing to worry about.

```
private static string connectionString = "DATA SOURCE=(local);" +
        "INITIAL CATALOG=AdventureWorks;" +
        "INTEGRATED SECURITY=SSPI;";

public static DataTable ExecuteQuery(string ftsQuery)
{
        SqlDataAdapter da = null;
        DataTable dt = null;
        try
        {
                dt = new DataTable();
                da = new SqlDataAdapter
                (
                        "SELECT ROW_NUMBER() OVER (ORDER BY DocumentId) AS Number, " +
                        "   Title, " +
                        "   DocumentSummary " +
                        "FROM Production.Document " +
                        "WHERE CONTAINS(*, @ftsQuery);",
                        connectionString
                );
                da.SelectCommand.Parameters.Add("@ftsQuery", SqlDbType.NVarChar, 4000).Value = ftsQuery;
                da.Fill(dt);
                da.Dispose();
        }
        catch (Exception ex)
        {
                if (da != null)
                da.Dispose();
                if (dt != null)
                dt.Dispose();
                throw (ex);
        }
        return dt;
}
```

I've chosen to populate a `DataTable` via a `SqlDataAdapter` just for simplicity of code, since I'm going to populate a `DataGridView` with the results. Alternatively you can use a `SqlDataCommand` and `SqlDataReader` to retrieve the results. The important part is the `CONTAINS` clause of the query, which should ultimately resemble this:

```
SELECT ...
FROM ...
WHERE CONTAINS(*, @ftsQuery);
```

The `@ftsQuery` parameter is your actual `CONTAINS` clause FTS query with all of its strange and wondrous keywords and operators. In order to provide transparency to the conversion process, and make it easier to test, the sample application in the download displays the FTS query that your Google-syntax query generates, as shown below.

As you can see, the Google-style sample query in the example:

```
("aluminum" +crank) OR tire
```

Is converted to the FTS `CONTAINS` query:

```
(("aluminum" AND  "crank" ) OR  FORMSOF (INFLECTIONAL, tire) )
```

Performing tests of this form is the best way to determine how the conversion engine performs translates your queries to FTS syntax.

Conclusion

With just a little bit of code you can create a simple and powerful interface for you SQL Server-based full-text search applications. Leveraging the Web search knowledge that your end users already have can cut down training time and costs, make your users more productive, and ensure that users will take full advantage of the search-based applications you design for them.

Download

The source code used in this article is available for download from the link above. This code was written in C# using Visual Studio 2008, and is designed to run against the SQL Server 2005 AdventureWorks sample database. The Irony .NET Compiler Construction Toolkit is available from http://www.codeplex.com/irony.

Thanks

To finish up this article I'd like to thank Roman Ivantsov for creating the Irony .NET Compiler Construction Toolkit, for reviewing my initial grammar, and for taking the initial crack at the AST walker.

About The Author

Michael Coles is a Microsoft MVP and the author of the Pro T-SQL 2008 Programmer's Guide and coauthor of the upcoming book Pro Full Text Search In SQL Server 2008.