Phil Factor 15 November 2010

Consuming JSON Strings in SQL Server

It has always seemed strange to Phil that SQL Server has such complete support for XML, yet is completely devoid of any support for JSON. In the end, he was forced, by a website project, into doing something about it. The result is this article, an iconoclastic romp around the representation of hierarchical structures, and some code to get you started.

Updated 2nd May 2013
Updated 8th May 2014

> *"The best thing about XML is what it shares with JSON, being human readable. That turns out to be important, not because people should be reading it, because we shouldn't, but because it avoids interoperability problems caused by fussy binary encoding issues.*
>
> *Beyond that, there is not much to like. It is not very good as a data format. And it is not very good as a document format. If it were a good document format, then wikis would use it."*
>
> **Doug Crockford March 2010**

This article describes a TSQL JSON parser and its evil twin, a JSON outputter, and provides the source. It is also designed to illustrate a number of string manipulation techniques in TSQL. With it you can do things like this to extract the data from a JSON document:

```
Select * from parseJSON('{   "Person":
  {
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "Address":
    {
      "streetAddress":"21 2nd Street",
      "city":"New York",
      "state":"NY",
      "postalCode":"10021"
    },
    "PhoneNumbers":
    {
      "home":"212 555-1234",
      "fax":"646 555-4567"
    }
  }
}
')
```

And get:

	element_id	parent_ID	Object_ID	NAME	StringValue	ValueType
1	1	1	NULL	streetAddress	21 2nd Street	string
2	2	1	NULL	city	New York	string
3	3	1	NULL	state	NY	string
4	4	1	NULL	postalCode	10021	string
5	5	2	NULL	home	212 555-1234	string
6	6	2	NULL	fax	646 555-4567	string
7	7	3	NULL	firstName	John	string
8	8	3	NULL	lastName	Smith	string
9	9	3	NULL	age	25	int
10	10	3	1	Address	1	object
11	11	3	2	PhoneNumbers	2	object
12	12	4	3	Person	3	object
13	13	NULL	4	-		object

...or you can do the round trip:

```
DECLARE @MyHierarchy Hierarchy  INSERT INTO @myHierarchy
select * from parseJSON('{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}')
SELECT dbo.ToJSON(@MyHierarchy)
```

To get:

```
{  "menu" :   {
  "id" : "file",
  "value" : "File",
  "popup" :   {
    "menuitem" :   [
       {
       "value" : "New",
       "onclick" : "CreateNewDoc()"
       },
       {
       "value" : "Open",
       "onclick" : "OpenDoc()"
       },
       {
       "value" : "Close",
       "onclick" : "CloseDoc()"
       }
     ]
   }
  }
}
```

Background

TSQL isn't really designed for doing complex string parsing, particularly where strings represent nested data structures such as XML, JSON, YAML, or XHTML.

You can do it but it is not a pretty sight; but why would you ever want to do it anyway? Surely, if anything was meant for the 'application layer' in C# or VB.net, then this is it. 'Oh yes', will chime in the application thought police, 'this is far better done in the application or with a CLR.' Not necessarily.

Sometimes, you just need to do something inappropriate in TSQL.

There are a whole lot of reasons why this might happen to you. It could be that your DBA doesn't allow a CLR, for example, or you lack the necessary skills with procedural code. Sometimes, there isn't any application, or you want to run code unobtrusively across databases or servers.

I needed to interpret or 'shred' JSON data. JSON is one of the most popular lightweight markup languages, and is probably the best choice for transfer of object data from a web page. It is, in fact, executable JavaScript that is very quick to code in the browser in order to dump the contents of a JavaScript object, and is lightning-fast to populate the browser object from the database since you are passing it executable code (you need to parse it first for security reasons – passing executable code around is potentially very risky). AJAX can use JSON rather than XML so you have an opportunity to have a much simpler route for data between database and browser, with less opportunity for error.

The conventional way of dealing with data like this is to let a separate business layer parse a JSON 'document' into some tree structure and then update the database by making a series of calls to it. This is fine, but can get more complicated if you need to ensure that the updates to the database are wrapped into one transaction so that if anything goes wrong, then the whole operation can be rolled back. This is why a CLR or TSQL approach has advantages.

I wrote the parser as a prototype because it was the quickest way to determine what was involved in the process, so I could then re-write something as a CLR in a .NET language. It takes a JSON string and produces a result in the form of an adjacency list representation of that hierarchy. In the end, the code did what I wanted with adequate performance (It reads a json file of 540 name\value pairs and creates the SQL hierarchy table in 4 seconds) so I didn't bother with the added complexity of maintaining a CLR routine. In order to test more thoroughly what I'd done, I wrote a JSON generator that used the same Adjacency list, so you can now import and export data via JSON!

"Sometimes, you just need to do something inappropriate in TSQL…"

These markup languages such as JSON and XML all represent object data as hierarchies. Although it looks very different to the entity-relational model, it isn't. It is rather more a different perspective on the same model. The first trick is to represent it as a Adjacency list hierarchy in a table, and then use the contents of this table to update the database. This Adjacency list is really the Database equivalent of any of the nested data structures that are used for the interchange of serialized information with the application, and can be used to create XML, OSX Property lists, Python nested structures or YAML as easily as JSON.

Adjacency list tables have the same structure whatever the data in them. This means that you can define a single Table-Valued Type and pass data structures around between stored procedures. However, they are best held at arms-length from the data, since they are not relational tables, but something more like the dreaded EAV (Entity-Attribute-Value)

tables. Converting the data from its Hierarchical table form will be different for each application, but is easy with a CTE. You can, alternatively, convert the hierarchical table into XML and interrogate that with XQuery.

JSON format.

JSON is designed to be as lightweight as possible and so it has only two structures. The first, delimited by curly brackets, is a collection of name/value pairs, separated by commas. The name is followed by a colon. This structure is generally implemented in the application-level as an *object*, record, struct, dictionary, hash table, keyed list, or associative array. The other structure is an ordered list of values, separated by commas. This is usually manifested as an *array*, vector, list, or sequence.

The first snag for TSQL is that the curly or square brackets are not 'escaped' within a string, so that there is no way of shredding a JSON 'document' simply. It is difficult to differentiate a bracket used as the delimiter of an array or structure, and one that is within a string. Also, interpreting a string into a SQL String isn't entirely straightforward since hex codes can be embedded anywhere to represent complex Unicode characters, and all the old C-style escaped characters are used. The second complication is that, unlike YAML, the datatypes of values can't be explicitly declared. You have to sniff them out from applying the rules from the JSON Specification.

"Using recursion in TSQL is like Sumo Wrestlers doing Ballet. It is possible but not pretty."

Obviously, structures can be embedded in structures, so recursion is a natural way of making life easy. Using recursion in TSQL is like Sumo Wrestlers doing Ballet. It is possible but not pretty.

The implementation

Although the code for the JSON Parser/Shredder will run in SQL Server 2005, and even in SQL Server 2000 (with some modifications required), I couldn't resist using a TVP (Table Valued Parameter) to pass a hierarchical table to the function, **ToJSON,** that produces a JSON 'document'. Writing a SQL Server 2005 version should not be too hard.

First the function replaces all strings with tokens of the form **@Stringxx**, where **xx** is the foreign key of the table variable where the strings are held. This takes them, and their potentially difficult embedded brackets, out of the way. Names are always strings in JSON as well as string values.

Then, the routine iteratively finds the next structure that has no structure contained within it, (and is, by definition the leaf structure), and parses it, replacing it with an object token of the form '**@Objectxxx**', or '**@arrayxxx**', where **xxx** is the object id assigned to it. The values, or name/value pairs are retrieved from the string table and stored in the hierarchy table. Gradually, the JSON document is eaten until there is just a single root object left.

The JSON outputter is a great deal simpler, since one can be surer of the input, but essentially it does the reverse process, working from the root to the leaves. The only complication is working out the indent of the formatted output string.

In the implementation, you'll see a fairly heavy use of PATINDEX. This uses a poor man's RegEx, a starving man's RegEx. However, it is all we have, and can be pressed into service by chopping the string it is searching (if only it had an optional third parameter like CHARINDEX that specified the index of the start position of the search!). The STUFF function is also a godsend for this sort of string-manipulation work.

```
CREATE FUNCTION dbo.parseJSON( @JSON NVARCHAR(MAX))
RETURNS @hierarchy TABLE
  (
   element_id INT IDENTITY(1, 1) NOT NULL, /* internal surrogate primary key gives the order of parsing and the list order */
   sequenceNo [int] NULL, /* the place in the sequence for the element */
   parent_ID INT,/* if the element has a parent then it is in this column. The document is the ultimate parent, so you can get the stru
cture from recursing from the document */
   Object_ID INT,/* each list or object has an object id. This ties all elements to a parent. Lists are treated as objects here */
   NAME NVARCHAR(2000),/* the name of the object */
   StringValue NVARCHAR(MAX) NOT NULL,/*the string representation of the value of the element. */
```

```
    ValueType VARCHAR(10) NOT null /* the declared type of the value represented as a string in StringValue*/
    )
  AS
  BEGIN
    DECLARE
      @FirstObject INT, --the index of the first open bracket found in the JSON string
      @OpenDelimiter INT,--the index of the next open bracket found in the JSON string
      @NextOpenDelimiter INT,--the index of subsequent open bracket found in the JSON string
      @NextCloseDelimiter INT,--the index of subsequent close bracket found in the JSON string
      @Type NVARCHAR(10),--whether it denotes an object or an array
      @NextCloseDelimiterChar CHAR(1),--either a '}' or a ']'
      @Contents NVARCHAR(MAX), --the unparsed contents of the bracketed expression
      @Start INT, --index of the start of the token that you are parsing
      @end INT,--index of the end of the token that you are parsing
      @param INT,--the parameter at the end of the next Object/Array token
      @EndOfName INT,--the index of the start of the parameter at end of Object/Array token
      @token NVARCHAR(200),--either a string or object
      @value NVARCHAR(MAX), -- the value as a string
      @SequenceNo int, -- the sequence number within a list
      @name NVARCHAR(200), --the name as a string
      @parent_ID INT,--the next parent ID to allocate
      @lenJSON INT,--the current length of the JSON String
      @characters NCHAR(36),--used to convert hex to decimal
      @result BIGINT,--the value of the hex symbol being parsed
      @index SMALLINT,--used for parsing the hex value
      @Escape INT --the index of the next escape character

    DECLARE @Strings TABLE /* in this temporary table we keep all strings, even the names of the elements, since they are 'escaped'
in a different way, and may contain, unescaped, brackets denoting objects or lists. These are replaced in the JSON string by tokens r
epresenting the string */
      (
       String_ID INT IDENTITY(1, 1),
       StringValue NVARCHAR(MAX)
      )
    SELECT--initialise the characters to convert hex to ascii
      @characters='0123456789abcdefghijklmnopqrstuvwxyz',
      @SequenceNo=0, --set the sequence no. to something sensible.
    /* firstly we process all strings. This is done because [{} and ] aren't escaped in strings, which complicates an iterative parse. */
      @parent_ID=0;
    WHILE 1=1 --forever until there is nothing more to do
      BEGIN
        SELECT
          @start=PATINDEX('%[^a-zA-Z]["]%', @json collate SQL_Latin1_General_CP850_Bin);--next delimited string
        IF @start=0 BREAK --no more so drop through the WHILE loop
        IF SUBSTRING(@json, @start+1, 1)='"'
          BEGIN --Delimited Name
            SET @start=@Start+1;
            SET @end=PATINDEX('%[^\]["]%', RIGHT(@json, LEN(@json+'|')-@start) collate SQL_Latin1_General_CP850_Bin);
          END
        IF @end=0 --no end delimiter to last string
          BREAK --no more
        SELECT @token=SUBSTRING(@json, @start+1, @end-1)
        --now put in the escaped control characters
        SELECT @token=REPLACE(@token, FROMString, TOString)
        FROM
          (SELECT
           '\"' AS FromString, '"' AS ToString
          UNION ALL SELECT '\\', '\'
          UNION ALL SELECT '\/', '/'
          UNION ALL SELECT '\b', CHAR(08)
          UNION ALL SELECT '\f', CHAR(12)
          UNION ALL SELECT '\n', CHAR(10)
          UNION ALL SELECT '\r', CHAR(13)
          UNION ALL SELECT '\t', CHAR(09)
          ) substitutions
        SELECT @result=0, @escape=1
    --Begin to take out any hex escape codes
        WHILE @escape>0
          BEGIN
            SELECT @index=0,
            --find the next hex escape sequence
            @escape=PATINDEX('%\x[0-9a-f][0-9a-f][0-9a-f][0-9a-f]%', @token collate SQL_Latin1_General_CP850_Bin)
            IF @escape>0 --if there is one
              BEGIN
                WHILE @index<4 --there are always four digits to a \x sequence
                  BEGIN
                    SELECT --determine its value
                      @result=@result+POWER(16, @index)
                      *(CHARINDEX(SUBSTRING(@token, @escape+2+3-@index, 1),
                              @characters)-1), @index=@index+1 ;

                  END
                  -- and replace the hex sequence by its unicode value
                SELECT @token=STUFF(@token, @escape, 6, NCHAR(@result))
              END
          END
        --now store the string away
        INSERT INTO @Strings (StringValue) SELECT @token
        -- and replace the string with a token
        SELECT @JSON=STUFF(@json, @start, @end+1,
```

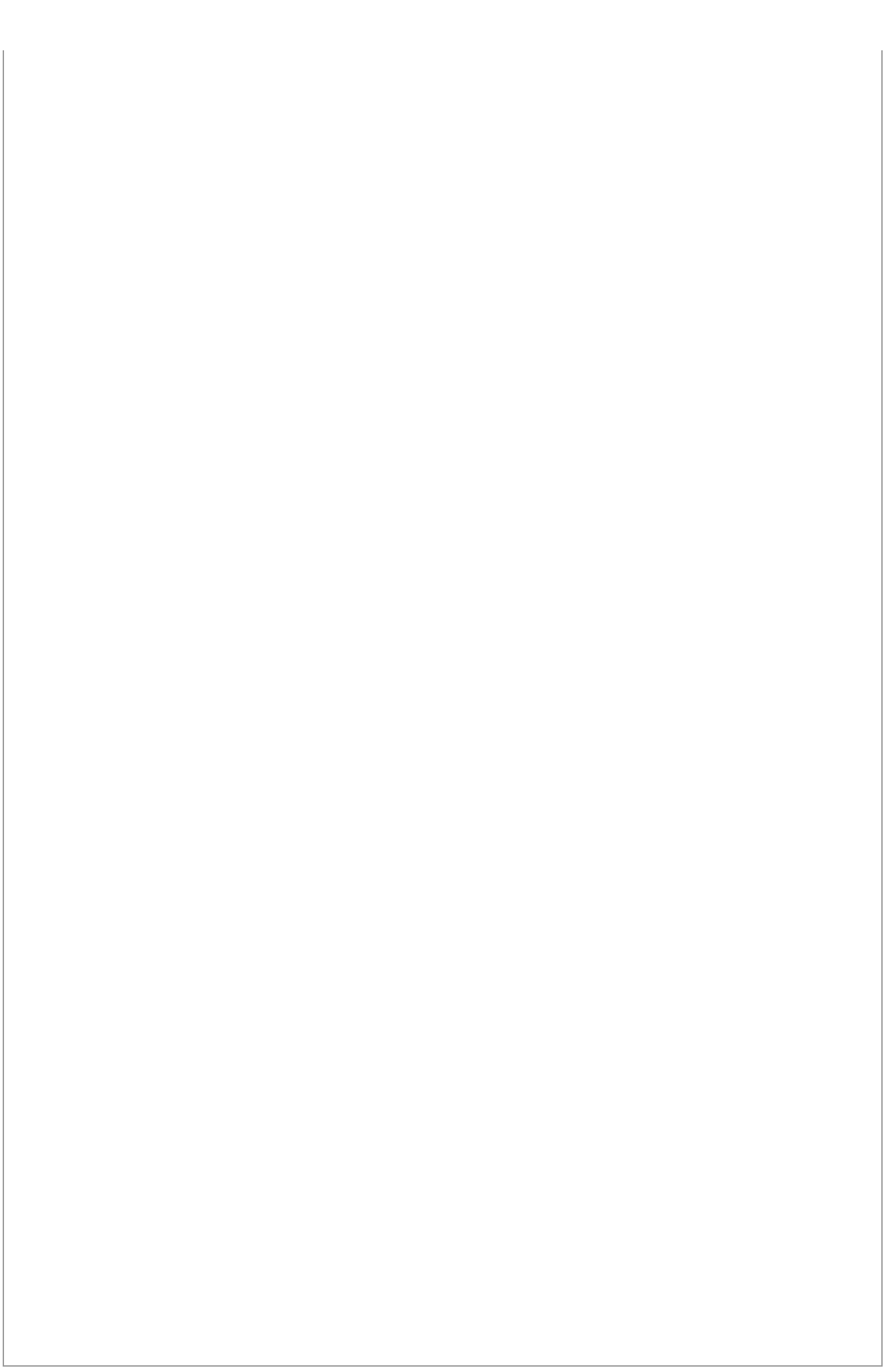
```
            '@string'+CONVERT(NVARCHAR(5), @@identity))
  END
-- all strings are now removed. Now we find the first leaf.
WHILE 1=1  --forever until there is nothing more to do
BEGIN

SELECT @parent_ID=@parent_ID+1
--find the first object or list by looking for the open bracket
SELECT @FirstObject=PATINDEX('%[{[[]%', @json collate SQL_Latin1_General_CP850_Bin)--object or array
IF @FirstObject = 0 BREAK
IF (SUBSTRING(@json, @FirstObject, 1)='{')
  SELECT @NextCloseDelimiterChar='}', @type='object'
ELSE
  SELECT @NextCloseDelimiterChar=']', @type='array'
SELECT @OpenDelimiter=@firstObject
WHILE 1=1 --find the innermost object or list...
  BEGIN
    SELECT
      @lenJSON=LEN(@JSON+'|')-1
--find the matching close-delimiter proceeding after the open-delimiter
    SELECT
      @NextCloseDelimiter=CHARINDEX(@NextCloseDelimiterChar, @json,
                                    @OpenDelimiter+1)
--is there an intervening open-delimiter of either type
    SELECT @NextOpenDelimiter=PATINDEX('%[{[[]%',
          RIGHT(@json, @lenJSON-@OpenDelimiter)collate SQL_Latin1_General_CP850_Bin)--object
    IF @NextOpenDelimiter=0
      BREAK
    SELECT @NextOpenDelimiter=@NextOpenDelimiter+@OpenDelimiter
    IF @NextCloseDelimiter<@NextOpenDelimiter
      BREAK
    IF SUBSTRING(@json, @NextOpenDelimiter, 1)='{'
      SELECT @NextCloseDelimiterChar='}', @type='object'
    ELSE
      SELECT @NextCloseDelimiterChar=']', @type='array'
    SELECT @OpenDelimiter=@NextOpenDelimiter
  END
---and parse out the list or name/value pairs
SELECT
  @contents=SUBSTRING(@json, @OpenDelimiter+1,
                      @NextCloseDelimiter-@OpenDelimiter-1)
SELECT
  @JSON=STUFF(@json, @OpenDelimiter,
              @NextCloseDelimiter-@OpenDelimiter+1,
              '@'+@type+CONVERT(NVARCHAR(5), @parent_ID))
WHILE (PATINDEX('%[A-Za-z0-9@+.e]%', @contents collate SQL_Latin1_General_CP850_Bin))<>0
  BEGIN
    IF @Type='Object' --it will be a 0-n list containing a string followed by a string, number,boolean, or null
      BEGIN
        SELECT
          @SequenceNo=0,@end=CHARINDEX(':', ' '+@contents)--if there is anything, it will be a string-based name.
        SELECT  @start=PATINDEX('%[^A-Za-z@][@]%', ' '+@contents collate SQL_Latin1_General_CP850_Bin)--AAAAAAAA
        SELECT @token=SUBSTRING(' '+@contents, @start+1, @End-@Start-1),
          @endofname=PATINDEX('%[0-9]%', @token collate SQL_Latin1_General_CP850_Bin),
          @param=RIGHT(@token, LEN(@token)-@endofname+1)
        SELECT
          @token=LEFT(@token, @endofname-1),
          @Contents=RIGHT(' '+@contents, LEN(' '+@contents+'|')-@end-1)
        SELECT  @name=stringvalue FROM @strings
          WHERE string_id=@param --fetch the name
      END
    ELSE
      SELECT @Name=null,@SequenceNo=@SequenceNo+1
    SELECT
      @end=CHARINDEX(',', @contents)-- a string-token, object-token, list-token, number,boolean, or null
    IF @end=0
      SELECT  @end=PATINDEX('%[A-Za-z0-9@+.e][^A-Za-z0-9@+.e]%', @Contents+' ' collate SQL_Latin1_General_CP850_Bin)
        +1
     SELECT
      @start=PATINDEX('%[^A-Za-z0-9@+.e][A-Za-z0-9@+.e]%', ' '+@contents collate SQL_Latin1_General_CP850_Bin)
    --select @start,@end, LEN(@contents+'|'), @contents
    SELECT
      @Value=RTRIM(SUBSTRING(@contents, @start, @End-@Start)),
      @Contents=RIGHT(@contents+' ', LEN(@contents+'|')-@end)
    IF SUBSTRING(@value, 1, 7)='@object'
      INSERT INTO @hierarchy
        (NAME, SequenceNo, parent_ID, StringValue, Object_ID, ValueType)
        SELECT @name, @SequenceNo, @parent_ID, SUBSTRING(@value, 8, 5),
          SUBSTRING(@value, 8, 5), 'object'
    ELSE
      IF SUBSTRING(@value, 1, 6)='@array'
        INSERT INTO @hierarchy
          (NAME, SequenceNo, parent_ID, StringValue, Object_ID, ValueType)
          SELECT @name, @SequenceNo, @parent_ID, SUBSTRING(@value, 7, 5),
            SUBSTRING(@value, 7, 5), 'array'
      ELSE
        IF SUBSTRING(@value, 1, 7)='@string'
          INSERT INTO @hierarchy
            (NAME, SequenceNo, parent_ID, StringValue, ValueType)
            SELECT @name, @SequenceNo, @parent_ID, stringvalue, 'string'
```

```
            FROM @strings
            WHERE string_id=SUBSTRING(@value, 8, 5)
          ELSE
            IF @value IN ('true', 'false')
              INSERT INTO @hierarchy
                (NAME, SequenceNo, parent_ID, StringValue, ValueType)
                SELECT @name, @SequenceNo, @parent_ID, @value, 'boolean'
            ELSE
              IF @value='null'
                INSERT INTO @hierarchy
                  (NAME, SequenceNo, parent_ID, StringValue, ValueType)
                  SELECT @name, @SequenceNo, @parent_ID, @value, 'null'
              ELSE
                IF PATINDEX('%[^0-9]%', @value collate SQL_Latin1_General_CP850_Bin)>0
                  INSERT INTO @hierarchy
                    (NAME, SequenceNo, parent_ID, StringValue, ValueType)
                    SELECT @name, @SequenceNo, @parent_ID, @value, 'real'
                ELSE
                  INSERT INTO @hierarchy
                    (NAME, SequenceNo, parent_ID, StringValue, ValueType)
                    SELECT @name, @SequenceNo, @parent_ID, @value, 'int'
      if @Contents=' ' Select @SequenceNo=0
    END
  END
 INSERT INTO @hierarchy (NAME, SequenceNo, parent_ID, StringValue, Object_ID, ValueType)
  SELECT '-',1, NULL, '', @parent_id-1, @type
 --
   RETURN
 END
GO
```

So once we have a hierarchy, we can pass it to a stored procedure. As the output is an adjacency list, it should be easy to access the data. You might find it handy to create a table type if you are using SQL Server 2008. Here is what I use. (Note that if you drop a Table Valued Parameter type, you will have to drop any dependent functions or procedures first, and re-create them afterwards).

```
-- Create the data type  IF EXISTS (SELECT * FROM sys.types WHERE name LIKE 'Hierarchy')
  DROP TYPE dbo.Hierarchy
go
CREATE TYPE dbo.Hierarchy AS TABLE
(
   element_id INT NOT NULL, /* internal surrogate primary key gives the order of parsing and the list order */
   sequenceNo [int] NULL, /* the place in the sequence for the element */
    parent_ID INT,/* if the element has a parent then it is in this column. The document is the ultimate parent, so you can get the struc
ture from recursing from the document */
   [Object_ID] INT,/* each list or object has an object id. This ties all elements to a parent. Lists are treated as objects here */
   NAME NVARCHAR(2000),/* the name of the object, null if it hasn't got one */
   StringValue NVARCHAR(MAX) NOT NULL,/*the string representation of the value of the element. */
   ValueType VARCHAR(10) NOT null /* the declared type of the value represented as a string in StringValue*/
    PRIMARY KEY (element_id)
)
```



ToJSON. A function that creates JSON Documents

Firstly, we need a simple utility function:

```
IF OBJECT_ID (N'dbo.parseJSON') IS NOT NULL     DROP FUNCTION dbo.JSONEscaped
GO

CREATE FUNCTION JSONEscaped ( /* this is a simple utility function that takes a SQL String with all its clobber and outputs it as a stin
g with all the JSON escape sequences in it.*/
  @Unescaped NVARCHAR(MAX) --a string with maybe characters that will break json
  )
RETURNS NVARCHAR(MAX)
AS
BEGIN
  SELECT  @Unescaped = REPLACE(@Unescaped, FROMString, TOString)
  FROM    (SELECT
           '\"' AS FromString, '"' AS ToString
          UNION ALL SELECT '\', '\\'
          UNION ALL SELECT '/', '\/'
          UNION ALL SELECT  CHAR(08),'\b'
          UNION ALL SELECT  CHAR(12),'\f'
          UNION ALL SELECT  CHAR(10),'\n'
          UNION ALL SELECT  CHAR(13),'\r'
          UNION ALL SELECT  CHAR(09),'\t'
         ) substitutions
RETURN @Unescaped
END
```

And now, the function that takes a JSON Hierarchy table and converts it to a JSON string.

```
CREATE FUNCTION ToJSON
(
    @Hierarchy Hierarchy READONLY
)

/*
the function that takes a Hierarchy table and converts it to a JSON string

Author: Phil Factor
Revision: 1.5
date: 1 May 2014
why: Added a fix to add a name for a list.
example:

Declare @XMLSample XML
Select @XMLSample='
  <glossary><title>example glossary</title>
  <GlossDiv><title>S</title>
   <GlossList>
    <GlossEntry id="SGML"" SortAs="SGML">
     <GlossTerm>Standard Generalized Markup Language</GlossTerm>
     <Acronym>SGML</Acronym>
     <Abbrev>ISO 8879:1986</Abbrev>
     <GlossDef>
      <para>A meta-markup language, used to create markup languages such as DocBook.</para>
      <GlossSeeAlso OtherTerm="GML" />
      <GlossSeeAlso OtherTerm="XML" />
     </GlossDef>
     <GlossSee OtherTerm="markup" />
    </GlossEntry>
   </GlossList>
  </GlossDiv>
 </glossary>'

DECLARE @MyHierarchy Hierarchy -- to pass the hierarchy table around
insert into @MyHierarchy select * from dbo.ParseXML(@XMLSample)
SELECT dbo.ToJSON(@MyHierarchy)

      */
RETURNS NVARCHAR(MAX)--JSON documents are always unicode.
AS
BEGIN
  DECLARE
    @JSON NVARCHAR(MAX),
    @NewJSON NVARCHAR(MAX),
    @Where INT,
    @ANumber INT,
    @notNumber INT,
    @indent INT,
    @ii int,
    @CrLf CHAR(2)--just a simple utility to save typing!

  --firstly get the root token into place
  SELECT @CrLf=CHAR(13)+CHAR(10),--just CHAR(10) in UNIX
        @JSON = CASE ValueType WHEN 'array' THEN
        +COALESCE('{'+@CrLf+'  "'+NAME+'" : ','')+'['
        ELSE '{' END
          +@CrLf
          + case when ValueType='array' and NAME is not null then '  ' else '' end
          + '@Object'+CONVERT(VARCHAR(5),OBJECT_ID)
          +@CrLf+CASE ValueType WHEN 'array' THEN
          case when NAME is null then ']' else '  ]'+@CrLf+'}'+@CrLf end
             ELSE '}' END
  FROM @Hierarchy
    WHERE parent_id IS NULL AND valueType IN ('object','document','array') --get the root element
 /* now we simply iterat from the root token growing each branch and leaf in each iteration. This won't be enormously quick, but it is
simple to do. All values, or name/value pairs withing a structure can be created in one SQL Statement*/
  Select @ii=1000
  WHILE @ii>0
    begin
    SELECT @where= PATINDEX('%[^[a-zA-Z0-9]@Object%',@json)--find NEXT token
    if @where=0 BREAK
    /* this is slightly painful. we get the indent of the object we've found by looking backwards up the string */
    SET @indent=CHARINDEX(char(10)+char(13),Reverse(LEFT(@json,@where))+char(10)+char(13))-1
    SET @NotNumber= PATINDEX('%[^0-9]%', RIGHT(@json,LEN(@JSON+'|')-@Where-8)+' ')--find NEXT token
    SET @NewJSON=NULL --this contains the structure in its JSON form
    SELECT
```

```
      @NewJSON=COALESCE(@NewJSON+','+@CrLf+SPACE(@indent),'')
      +case when parent.ValueType='array' then '' else COALESCE('"'+TheRow.NAME+'" : ','') end
      +CASE TheRow.valuetype
      WHEN 'array' THEN '  ['+@CrLf+SPACE(@indent+2)
        +'@Object'+CONVERT(VARCHAR(5),TheRow.[OBJECT_ID])+@CrLf+SPACE(@indent+2)+']'
      WHEN 'object' then '  {'+@CrLf+SPACE(@indent+2)
        +'@Object'+CONVERT(VARCHAR(5),TheRow.[OBJECT_ID])+@CrLf+SPACE(@indent+2)+'}'
      WHEN 'string' THEN '"'+dbo.JSONEscaped(TheRow.StringValue)+'"'
      ELSE TheRow.StringValue
      END
    FROM @Hierarchy TheRow
    inner join @hierarchy Parent
    on parent.element_ID=TheRow.parent_ID
     WHERE TheRow.parent_id= SUBSTRING(@JSON,@where+8, @Notnumber-1)
    /* basically, we just lookup the structure based on the ID that is appended to the @Object token. Simple eh? */
   --now we replace the token with the structure, maybe with more tokens in it.
   Select @JSON=STUFF (@JSON, @where+1, 8+@NotNumber-1, @NewJSON),@ii=@ii-1
   end
  return @JSON
end
go
```

ToXML. A function that creates XML

The function that converts a hierarchy table to XML gives us a JSON to XML converter. It is surprisingly similar to the previous function

```
IF OBJECT_ID (N'dbo.ToXML') IS NOT NULL
   DROP FUNCTION dbo.ToXML
GO
CREATE FUNCTION ToXML
(
/*this function converts a Hierarchy table into an XML document. This uses the same technique as the toJSON function, and uses the '
entities' form of XML syntax to give a compact rendering of the structure */
      @Hierarchy Hierarchy READONLY
)
RETURNS NVARCHAR(MAX)--use unicode.
AS
BEGIN
  DECLARE
    @XMLAsString NVARCHAR(MAX),
    @NewXML NVARCHAR(MAX),
    @Entities NVARCHAR(MAX),
    @Objects NVARCHAR(MAX),
    @Name NVARCHAR(200),
    @Where INT,
    @ANumber INT,
    @notNumber INT,
    @indent INT,
    @CrLf CHAR(2)--just a simple utility to save typing!

  --firstly get the root token into place
  --firstly get the root token into place
  SELECT @CrLf=CHAR(13)+CHAR(10),--just CHAR(10) in UNIX
        @XMLAsString ='<?xml version="1.0" ?>
@Object'+CONVERT(VARCHAR(5),OBJECT_ID)+'
'
```

```
    FROM @hierarchy
    WHERE parent_id IS NULL AND valueType IN ('object','array') --get the root element
/* now we simply iterate from the root token growing each branch and leaf in each iteration. This won't be enormously quick, but it is
simple to do. All values, or name/value pairs within a structure can be created in one SQL Statement*/
  WHILE 1=1
    begin
    SELECT @where= PATINDEX('%[^a-zA-Z0-9]@Object%',@XMLAsString)--find NEXT token
    if @where=0 BREAK
    /* this is slightly painful. we get the indent of the object we've found by looking backwards up the string */
    SET @indent=CHARINDEX(char(10)+char(13),Reverse(LEFT(@XMLasString,@where))+char(10)+char(13))-1
    SET @NotNumber= PATINDEX('%[^0-9]%', RIGHT(@XMLasString,LEN(@XMLAsString+'|')-@Where-8)+' ')--find NEXT token
    SET @Entities=NULL --this contains the structure in its XML form
    SELECT @Entities=COALESCE(@Entities+' ','')+NAME+'="'
     +REPLACE(REPLACE(REPLACE(StringValue, '<', '&lt;'), '&', '&amp;'),'>', '&gt;')
     + '"'
      FROM @hierarchy
      WHERE parent_id= SUBSTRING(@XMLasString,@where+8, @Notnumber-1)
        AND ValueType NOT IN ('array', 'object')
    SELECT @Entities=COALESCE(@entities,''),@Objects='',@name=CASE WHEN Name='-' THEN 'root' ELSE NAME end
      FROM @hierarchy
      WHERE [Object_id]= SUBSTRING(@XMLasString,@where+8, @Notnumber-1)

    SELECT  @Objects=@Objects+@CrLf+SPACE(@indent+2)
          +'@Object'+CONVERT(VARCHAR(5),OBJECT_ID)
          --+@CrLf+SPACE(@indent+2)+''
      FROM @hierarchy
      WHERE parent_id= SUBSTRING(@XMLasString,@where+8, @Notnumber-1)
      AND ValueType IN ('array', 'object')
    IF @Objects='' --if it is a lef, we can do a more compact rendering
        SELECT @NewXML='<'+COALESCE(@name,'item')+@entities+' />'
    ELSE
       SELECT @NewXML='<'+COALESCE(@name,'item')+@entities+'>'
          +@Objects+@CrLf++SPACE(@indent)+'</'+COALESCE(@name,'item')+'>'
     /* basically, we just lookup the structure based on the ID that is appended to the @Object token. Simple eh? */
    --now we replace the token with the structure, maybe with more tokens in it.
    Select @XMLasString=STUFF (@XMLasString, @where+1, 8+@NotNumber-1, @NewXML)
    end
  return @XMLasString
  end
```

This provides you the means of converting a JSON string into XML

```
DECLARE @MyHierarchy Hierarchy,@xml XML
INSERT INTO @myHierarchy
select * from parseJSON('{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}')
SELECT dbo.ToXML(@MyHierarchy)
SELECT @XML=dbo.ToXML(@MyHierarchy)
SELECT @XML
```

This gives the result…

```
<?xml version="1.0" ?>
<root>
  <menu id="file""" value="File">
    <popup>
      <menuitem>
        <item value="New" onclick="CreateNewDoc()" />
        <item value="Open" onclick="OpenDoc()" />
        <item value="Close" onclick="CloseDoc()" />
      </menuitem>
    </popup>
  </menu>
</root>


(1 row(s) affected)


<root><menu id="file""" value="File"><popup><menuitem><item value="New" onclick="CreateNewDoc()" /><item value="Ope
n" onclick="OpenDoc()" /><item value="Close" onclick="CloseDoc()" /></menuitem></popup></menu></root>

(1 row(s) affected)
```

Wrap-up

The so-called 'impedence-mismatch' between applications and databases is, I reckon, an illusion. The object-oriented nested data-structures that we receive from applications are, if the developer has understood the data correctly, merely a perspective from a particular entity of the relationships it is involved with. Whereas it is easy to shred XML documents to get the data from it to update the database, it has been trickier with other formats such as JSON. By using techniques like this, it should be possible to liberate the application, or website, programmer from having to do the mapping from the object model to the relational, and spraying the database with ad-hoc TSQL that uses the base tables or updateable views. If the database can be provided with the JSON, or the Table-Valued parameter, then there is a better chance of maintaining full transactional integrity for the more complex updates.

The database developer already has the tools to do the work with XML, but why not the simpler, and more practical JSON? I hope these two routines get you started with experimenting with this.

Interesting JSON-related articles and sites

- YAML: YAML Ain't Markup Language
- XMLHttpRequest
- Introducing JSON
- JSON: The Fat-Free Alternative to XML
- Doug Crockford: Geek of the Week
- JSON and other data serialization languages
- WDDX
- OData: JavaScript Object Notation (JSON) Format
- The Atom Publishing Protocol

> Since writing this article, Phil has also developed a CSV parser and output and an XML parser (Producing JSON Documents from SQL Server queries via TSQL)

JSON.sql (52 KB)

Phil Factor

Phil Factor (real name withheld to protect the guilty), aka Database Mole, has 30 years of experience with database-intensive applications. Despite having once been shouted at by a furious Bill Gates at an exhibition in the early 1980s, he has remained resolutely anonymous throughout his career. See also :

- The Phrenetic Phoughts of Phil Factor
- Phil on Twitter
- Phil on SQL Server Central
- Robyn and Phil's Workbenches
- Phil's Editorials on SQL Server Central

Follow on

View all articles by Phil Factor

Related articles

A L S JO S ON

Microsoft Azure DocumentDB

DocumentDB is a late-entrant in the Document-oriented database field. However, it benefits from being designed from the start as a cloud service with a SQL-like language. It is intended for mobile and web applications. Its JSON document-notation is compatible with the integrated JavaScript language that drives its multi-document transaction processing via stored procedures, triggers and UDFs.... Read more

A L S RO H I N F A C T O R

Writing Efficient SQL: Set-Based Speed Phreakery

Phil Factor's SQL Speed Phreak challenge is an event where coders battle to produce the fastest code to solve a common reporting problem on large data sets. It isn't that easy on the spectators, since the programmers don't score extra points for commenting their code. Mercifully, Kathi is on hand to explain some of the TSQL coding secrets that go to producing blistering performance. ... Read more

A L S SO Q L N

Relational Algebra and its implications for NoSQL databases

With the rise of NoSQL databases that are exploiting aspects of SQL for querying, and are embracing full transactionality, is there a danger of the data-document model's hierarchical nature causing a fundamental conflict with relational theory? We

asked our relational expert, Hugh Bin-Haad to expound a difficult area for database theorists.... Read more

Tags

JSON, Phil Factor, SQL, SQL Server, T-SQL, T-SQL Programming, XML