BUILD A WEBSITE OF YOUR OWN ON TRIPOD

Ads by Google

Open Source MDM Solutions
Minimize Implementation Complexity w/ Talend MDM Tools. Download Now!
www.talend.com/open_source_mdm

SQL Server Performance
Monitor DB performance & Server. Easy setup. Download Now!
www.manageengine.com

Passing a list/array to an SQL Server stored procedure

Last updated: October 29th '05 | Best viewed with: All popular browsers | Best viewed at: 1024x768 | Links to external sites will open in a new window

About myself
My technical skills
My favorites
My picture album

Shortcut keys
My code library

VB resources
SQLServer resources
SQLServer books
Replication FAQ
Scripting resources
ASP resources

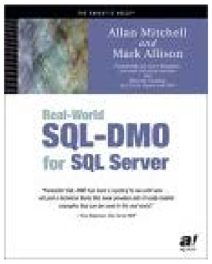
Search my site
Sign my guestbook
Contact information

This month's 2 click survey:
Is .NET important for a database professional?

SQL Server Articles
New

Click here to find out the top 15 SQL Server books purchased by this site's visitors! NEW

Featured Book:



BUILD A WEBSITE OF YOUR OWN ON TRIPOD

NEW!!! Subscribe to my newsletter:
Want to keep in touch with the latest in SQL Server world? Email **vyaskn@hotmail.com** with 'subscribe' in the subject line

How to pass a list of values or array to SQL Server stored procedure?

Note: Information & code samples from this article are tested on SQL Server 2005 RTM (Yukon) and found to be working. Will update the article in case of any compatibility issues.

unfortunately, there is no built-in support for arrays in SQL Server's T-SQL. SQL Server 2000 did add some new datatypes like sql_variant, bigint etc, but no support for the much needed arrays. There are some situations, that require the ability to pass a list of values to a stored procedure. Think about a web page, that lets the user select one or more of his/her previous orders, on submit, retrieves complete information about the selected orders. In this case, passing a list of selected order numbers to the stored procedure, in one go, and getting the results back is more efficient, compared to calling the same stored procedure for each selected order number.

Since, we cannot create arrays of variables or input parameters or columns in T-SQL, we need to look for workarounds and alternatives. Over the years, programmers developed different techniques, some of which are not so efficient, some efficient, but complex. The most popular technique is to pass in a list of values, separated by commas (CSV). With this method, the normal input parameter of the stored procedure receives a list of say, OrderIDs, separated by commas. In this article, I'll present some of these techniques. At the end of the article I will also provide you with links to articles and books, that discussed the implementation of arrays in T-SQL.

The following examples are simplified, just to give you an idea of how things work. You may have to adapt them to suit your needs. Also, the following stored procedures query the Orders table from the Northwind sample database, that ships with SQL Server 7.0 and 2000. You should be able to create and execute these procedures in Northwind.

Method 1: Dynamic SQL
(Works in both SQL Server 7.0 and 2000)

```
CREATE PROC dbo.GetOrderList1
(
	@OrderList varchar(500)
)
AS
BEGIN
	SET NOCOUNT ON

	DECLARE @SQL varchar(600)

	SET @SQL =
	'SELECT OrderID, CustomerID, EmployeeID, OrderDate
	FROM dbo.Orders
	WHERE OrderID IN (' + @OrderList + ')'

	EXEC(@SQL)
END
GO

GRANT EXEC ON dbo.GetOrderList1 TO WebUser
GO

GRANT SELECT ON dbo.Orders TO WebUser
GO
```

The above stored procedure receives a list of OrderIDs separated by commas, as an input parameter. It then dynamically constructs an SQL statement and executes it using EXEC.

Dynamic SQL has its limitations, and is not something I would recommend. For starters, notice the "GRANT EXEC" command in the above script. That statement grants EXECUTE permission to the user WebUser. But that is not enough for WebUser to execute this stored procedure. The user executing dynamic SQL commands needs explicit permissions on the underlying tables, which is not something I would do on a production system. Because of this limitation, I added a "GRANT SELECT" command in the above script, to enable WebUser to run the stored procedure.

Call this stored procedure as shown below, and it will retrieve OrderID, CustomerID, EmployeeID and OrderDate columns for the given order numbers:

```
EXEC dbo.GetOrderList1 '10248,10252,10256,10261,10262,10263,10264,10265,10300,10311'
GO
```

Note that, dynamic SQL is vulnerable to SQL Injection, a technique using which a malicious user could inject his own code into your dynamic SQL string and get it executed. Try this example, and see what happens:

```
EXEC dbo.GetOrderList1 '0); SELECT * FROM sysobjects --'
GO
```

There are other limitations that apply to dynamic SQL. Check out the link at the end of this article, for additional information on using dynamic SQL.

Method 2: Parsing the comma separated values into a temporary table and joining the temp table to main table

(Works in both SQL Server 7.0 and 2000)

```
CREATE PROC dbo.GetOrderList2
(
	@OrderList varchar(500)
)
AS
BEGIN
	SET NOCOUNT ON

	CREATE TABLE #TempList
	(
		OrderID int
	)

	DECLARE @OrderID varchar(10), @Pos int

	SET @OrderList = LTRIM(RTRIM(@OrderList))+ ','
	SET @Pos = CHARINDEX(',', @OrderList, 1)

	IF REPLACE(@OrderList, ',', '') <> ''
	BEGIN
		WHILE @Pos > 0
		BEGIN
			SET @OrderID = LTRIM(RTRIM(LEFT(@OrderList, @Pos - 1)))
			IF @OrderID <> ''
			BEGIN
				INSERT INTO #TempList (OrderID) VALUES (CAST(@OrderID AS int)) --Use Appropriate conversion
			END
			SET @OrderList = RIGHT(@OrderList, LEN(@OrderList) - @Pos)
			SET @Pos = CHARINDEX(',', @OrderList, 1)

		END
	END

	SELECT o.OrderID, CustomerID, EmployeeID, OrderDate
	FROM	dbo.Orders AS o
		JOIN
		#TempList t
		ON o.OrderID = t.OrderID

END
GO

GRANT EXEC ON dbo.GetOrderList2 TO WebUser
GO
```

The above stored procedure receives a list of OrderIDs separated by commas, as an input parameter. It then parses the parameter, extracts individual OrderIDs from the comma separated list, inserts the OrderIDs into a temporary table, and then joins the temporary table with the main Orders table, to get the requested results.

Call this stored procedure as shown below, and it will retrieve OrderID, CustomerID, EmployeeID and OrderDate columns for the given order numbers:

```
EXEC dbo.GetOrderList2 '10248,10252,10256,10261,10262,10263,10264,10265,10300,10311'
GO
```

The creation of temporary tables inside a stored procedure, sometimes results in stored procedure recompilations. You can find a link at the end of this article, that has more information on this topic. You could verify this using Profiler. Also, T-SQL string functions are not very efficient, so the parsing could take more CPU cycles with large lists.

Method 3: Parsing the comma separated values into a table variable and joining the table variable to main table

(Works in SQL Server 2000 only)

```
CREATE PROC dbo.GetOrderList3
(
	@OrderList varchar(500)
)
AS
BEGIN
	SET NOCOUNT ON

	DECLARE @TempList table
	(
		OrderID int
	)

	DECLARE @OrderID varchar(10), @Pos int

	SET @OrderList = LTRIM(RTRIM(@OrderList))+ ','
	SET @Pos = CHARINDEX(',', @OrderList, 1)

	IF REPLACE(@OrderList, ',', '') <> ''
	BEGIN
		WHILE @Pos > 0
		BEGIN
			SET @OrderID = LTRIM(RTRIM(LEFT(@OrderList, @Pos - 1)))
			IF @OrderID <> ''
			BEGIN
				INSERT INTO @TempList (OrderID) VALUES (CAST(@OrderID AS int)) --Use Appropriate conversion
			END
			SET @OrderList = RIGHT(@OrderList, LEN(@OrderList) - @Pos)
			SET @Pos = CHARINDEX(',', @OrderList, 1)

		END
	END

	SELECT o.OrderID, CustomerID, EmployeeID, OrderDate
	FROM   dbo.Orders AS o
		JOIN
		@TempList t
		ON o.OrderID = t.OrderID

END
GO

GRANT EXEC ON dbo.GetOrderList3 TO WebUser
GO
```

The above stored procedure receives a list of OrderIDs separated by commas, as an input parameter. It then parses the parameter, extracts individual OrderIDs from the comma separated list, inserts the OrderIDs into a table variable, and then joins the table variable with the main Orders table, to get the requested results.

Call this stored procedure as shown below, and it will retrieve OrderID, CustomerID, EmployeeID and OrderDate columns for the given order numbers:

```
EXEC dbo.GetOrderList3 '10248,10252,10256,10261,10262,10263,10264,10265,10300,10311'
GO
```

Table variables could be quicker compared to temporary tables. Table variables don't have the limitation of recompilations, unlike method 2. But the parsing could consume more CPU cycles, if the list is huge. At the end of this article you'll find a link to an FAQ on table variables.

Method 4: Using XML

(Works in SQL Server 2000 only)

```
CREATE PROC dbo.GetOrderList4
(
	@OrderList varchar(1000)
)
AS
BEGIN
	SET NOCOUNT ON

	DECLARE @DocHandle int

	EXEC sp_xml_preparedocument @DocHandle OUTPUT, @OrderList

	SELECT o.OrderID, CustomerID, EmployeeID, OrderDate
	FROM    dbo.Orders AS o
		JOIN
		OPENXML (@DocHandle, '/ROOT/Ord',1) WITH (OrderID  int) AS x
		ON o.OrderID = x.OrderID

	EXEC sp_xml_removedocument @DocHandle
END
GO

GRANT EXEC ON dbo.GetOrderList4 TO WebUser
GO
```

The above stored procedure receives a list of OrderIDs, in the form of an XML document, as an input parameter. It then parses the XML document using sp_xml_preparedocument and OPENXML rowset provider, joins the output to the main Orders table to retrieve the order information.

Call this stored procedure as shown below, and it will retrieve OrderID, CustomerID, EmployeeID and OrderDate columns for the given order numbers:

```
EXEC dbo.GetOrderList4 '
<ROOT>
<Ord OrderID = "10248"/> <Ord OrderID = "10252"/>
<Ord OrderID = "10256"/> <Ord OrderID = "10261"/>
<Ord OrderID = "10262"/> <Ord OrderID = "10263"/>
<Ord OrderID = "10264"/> <Ord OrderID = "10265"/>
<Ord OrderID = "10300"/> <Ord OrderID = "10311"/>
<Ord OrderID = "11068"/> <Ord OrderID = "11069"/>
<Ord OrderID = "11070"/> <Ord OrderID = "11071"/>
<Ord OrderID = "11072"/> <Ord OrderID = "11073"/>
<Ord OrderID = "11074"/> <Ord OrderID = "11075"/>
<Ord OrderID = "11076"/> <Ord OrderID = "11077"/>
</ROOT>'
GO
```

Try to keep the element/attribute names in your XML tags as short as possible. This will help keep the document size small, and could improve parsing time. Also, the smaller the XML document, the lesser time it takes to travel over the network (from your application to your database server). Bear in mind that XML is case sensitive.

This SQLXML functionality available in SQL Server 2000 is very powerful and the above stored procedure is just a simple example. Be sure to check your query execution plan when joining tables with OPENXML output. If you see index scans, and the table is large, then you might want to dump the OPENXML output into a temporary table, and join that to your main table. This would more likely result in an index seek, provided you have the right index.

You'll find a link to an SQL XML book, at the end of this article.

Method 5: Using a table of numbers or pivot table, to parse the comma separated list

(Works in SQL Server 7.0 and 2000)

```
--Create a table called Numbers
CREATE TABLE dbo.Numbers
(
	Number int PRIMARY KEY CLUSTERED
)
GO

--Insert 8000 numbers into this table (from 1 to 8000)
SET NOCOUNT ON
GO

DECLARE @CTR int
SET @CTR = 1
WHILE @CTR < 8001
BEGIN
	INSERT INTO dbo.Numbers (Number) VALUES (@CTR)
	SET @CTR = @CTR + 1
END
GO
--The above two steps are to be run only once. The following stored procedure uses the number table.

CREATE PROC dbo.GetOrderList5
(
	@OrderList varchar(1000)
)
AS
BEGIN
	SET NOCOUNT ON

	SELECT o.OrderID, CustomerID, EmployeeID, OrderDate
	FROM dbo.Orders AS o
	JOIN
	(

		SELECT LTRIM(RTRIM(SUBSTRING(OrderID, number+1, CHARINDEX(',', OrderID, number+1)-number - 1))) AS OrderID
		FROM
		(
			SELECT ',' + @OrderList + ',' AS OrderID
		) AS InnerQuery
		JOIN
		Numbers n
		ON
		n.Number < LEN(InnerQuery.OrderID)
		WHERE SUBSTRING(OrderID, number, 1) = ','
	) as Derived
	ON o.OrderID = Derived.OrderID

END
GO

GRANT EXEC ON dbo.GetOrderList5 TO WebUser
GO
```

The above stored procedure receives a list of OrderIDs separated by commas, as an input parameter. It then parses the parameter, in just one query, using the number table (unlike WHILE loop in previous examples) and joins the individual OrderIDs to the OrderIDs from Orders table to retrieve the results.

Call this stored procedure as shown below, and it will retrieve OrderID, CustomerID, EmployeeID and OrderDate columns for the given order numbers:

```
EXEC dbo.GetOrderList5 '10248,10252,10256,10261,10262,10263,10264,10265,10300,10311'
GO
```

Method 6: Using a general purpose User Defined Function (UDF) to parse the comma separated OrderIDs

(Works in SQL Server 2000 only)

```
--The following is a general purpose UDF to split comma separated lists into individual items.
--Consider an additional input parameter for the delimiter, so that you can use any delimiter you like.
CREATE FUNCTION dbo.SplitOrderIDs
(
	@OrderList varchar(500)
)
RETURNS
@ParsedList table
(
	OrderID int
)
AS
BEGIN
```

```
	DECLARE @OrderID varchar(10), @Pos int

	SET @OrderList = LTRIM(RTRIM(@OrderList))+ ','
	SET @Pos = CHARINDEX(',', @OrderList, 1)

	IF REPLACE(@OrderList, ',', '') <> ''
	BEGIN
		WHILE @Pos > 0
		BEGIN
			SET @OrderID = LTRIM(RTRIM(LEFT(@OrderList, @Pos - 1)))
			IF @OrderID <> ''
			BEGIN
				INSERT INTO @ParsedList (OrderID)
				VALUES (CAST(@OrderID AS int)) --Use Appropriate conversion
			END
			SET @OrderList = RIGHT(@OrderList, LEN(@OrderList) - @Pos)
			SET @Pos = CHARINDEX(',', @OrderList, 1)

		END
	END
	RETURN
END
GO

CREATE PROC dbo.GetOrderList6
(
	@OrderList varchar(500)
)
AS
BEGIN
	SET NOCOUNT ON

	SELECT  o.OrderID, CustomerID, EmployeeID, OrderDate
	FROM    dbo.Orders AS o
		JOIN
		dbo.SplitOrderIDs(@OrderList) AS s
		ON
		o.OrderID = s.OrderID
END
GO

GRANT EXEC ON dbo.GetOrderList6 TO WebUser
GO
```

The above script creates a Multi-statement table-valued user defined function, that accepts comma separated lists, and splits the list into individual items and returns them in tabular format.

The stored procedure is almost similar to the one in method 2, except that it uses the UDF in the JOIN.

Call this stored procedure as shown below, and it will retrieve OrderID, CustomerID, EmployeeID and OrderDate columns for the given order numbers:

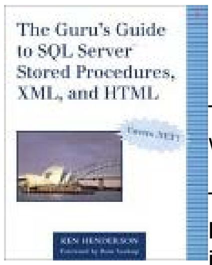
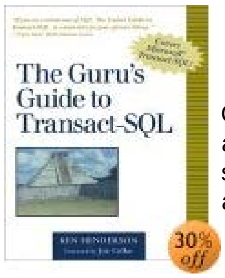
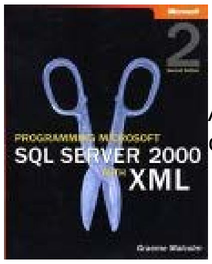
```
EXEC dbo.GetOrderList6 '10248,10252,10256,10261,10262,10263,10264,10265,10300,10311'
GO
```

Top

Here are the links to additional information, followed by links to related books:

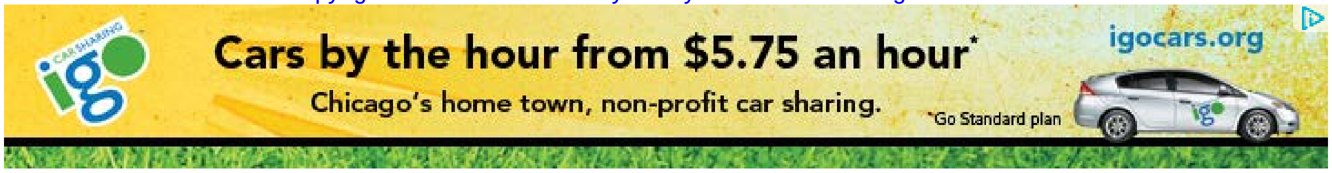
The curse and blessings of dynamic SQL, by Erland Sommarskog (SQL Server MVP)
A more complex example of OPENXML usage, by Linda (SQL Server MVP)
INF: Troubleshooting Stored Procedure Recompilation (Q243586)
INF: Frequently Asked Questions - SQL Server 2000 - Table Variables (Q305977)

Book Title	Review available?
The Guru's Guide to Stored Procedures, XML and HTML Buy from Amazon.com	

 This is an excellent Transact-SQL programming book, covers the basic as well as advanced information and provides loads of tips and tricks. The reason why I'm mentioning this book in this article is: The author, Ken Henderson, provided an extended stored procedure based solution to implement arrays in T-SQL. The C source code is available in the book and on the accompanying CD. Using these extended stored procedure we can easily implement single and multi dimensional arrays in T-SQL. The following extended stored procedures are included in the xp_array.dll : xp_createarray, xp_setarray, xp_getarray, xp_destroyarray, xp_listarray.	No
 The Guru's Guide to Transact-SQL Buy from Amazon.com Chapter 10 of this book is dedicated to Arrays. This chapter covers the alternatives like arrays as big strings, arrays as tables and discusses sorting, transposing dimensions, ensuring array integrity, reshaping the array, comparing arrays etc.	Yes. Click here to read
 SQL Server 2000 With XML Buy from Amazon.com A very useful book, that teaches how to work with the XML functionality of SQL Server 2000.	No

Disclaimer, terms of use and privacy policy

Copyright © 1997 - 2006 Narayana Vyas Kondreddi. All rights reserved.


igo CARSHARING
Cars by the hour from $5.75 an hour*
Chicago's home town, non-profit car sharing.
*Go Standard plan
igocars.org