



Pivot and Un-Pivot data in SQL

White Paper

© Copyright Decipher Information Systems, 2005. All rights reserved.

The information in this publication is furnished for information use only, does not constitute a commitment from Decipher Information Systems of any features or functions discussed and is subject to change without notice. Decipher Information Systems assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication.

Last revised: June 2006

Table of Contents

| | |
|---|---|
| Table of Contents | 3 |
| Abstract | 4 |
| Sample data to illustrate pivoting..... | 4 |
| Oracle | 5 |
| SQL Server..... | 6 |
| Limitations of the Pivot/Unpivot operators..... | 8 |
| Summary | 9 |

Pivot and Un-Pivot data in SQL

Abstract

It is very common scenario in application development when developers need to transform row-level data into column-level data and vice versa. Some common examples are providing aging information of pending payments, providing sales of all products by years etc.. Even though to display information in tabular or crosstab format is not new and applications like MS Excel has had these features since time immemorial, writing queries in SQL to represent data in such format becomes difficult.

In this whitepaper, we will show you some examples to perform static pivot/unpivot in Oracle, MS SQL Server and DB2 LUW. But we will concentrate more on two new relational operators PIVOT and UNPIVOT introduced in SS2K5 release. We will go through detail example of pivoting data using these new functions, its limitations and its performance implications.

Sample data to illustrate pivoting

```
/*
Create new table: Invoice and populate it with sample data. This is MS SQL Server
syntax. In order to work it for Oracle and DB2, you need to change it accordingly. This
data set is for use in pivoting example.
*/
```

```
CREATE TABLE dbo.Invoice
(
    INVOICE_NUMBER    INT IDENTITY(1,1) NOT NULL,
    INVOICE_DATE      DATETIME NOT NULL,
    CLIENT_ID         INT NOT NULL,
    INVOICE_AMT       NUMERIC(9,2) DEFAULT 0 NOT NULL,
    PAID_FLAG         TINYINT DEFAULT 0 NOT NULL, -- 0 Not paid/ 1 paid
    CONSTRAINT PK_INVOICE PRIMARY KEY (INVOICE_NUMBER)
)
--Filegroup clause
GO

INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-50,101,1100.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-49,102,1100.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-50,103,1100.00);

INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-40,101,1100.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-38,101,1500.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-41,102,1100.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-41,103,1100.00);

INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-22,101,1100.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-20,102,1100.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-28,103,1100.00);

INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-11,101,1100.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-12,102,1100.00);
INSERT INTO INVOICE (Invoice_date, client_ID, Invoice_Amt) VALUES (getDate()-13,103,1100.00);
```

```
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(getDate()-1,101,1100.00);
INSERT INTO INVOICE(Invoice_date, client_ID, Invoice_Amt) VALUES(getDate()-2,102,1100.00);
```

```
/******
```

Create following table and populate it for use in unpivot example. Again it is MS SQL Server syntax so for Oracle and DB2 LUW, you need to change create table script accordingly.

```
*****/
```

```
CREATE TABLE dbo.Product
(
    PRODUCT_ID      INT IDENTITY(1,1) NOT NULL,
    PRODUCT_NAME    VARCHAR(100) NOT NULL,
    STYLE           VARCHAR(10),
    Color           VARCHAR(10),
    Flammable       CHAR(1),
    CONSTRAINT PK_PRODUCT PRIMARY KEY (PRODUCT_ID)
)
```

```
--Filegroup Clause
```

```
GO
```

```
INSERT INTO PRODUCT (PRODUCT_NAME,STYLE,COLOR,FLAMMABLE) VALUES ('CANDLES','REGULAR','BLUE','Y');
INSERT INTO PRODUCT (PRODUCT_NAME,STYLE,COLOR,FLAMMABLE) VALUES ('FRAMES','GOTHIC','SILVER','N');
INSERT INTO PRODUCT (PRODUCT_NAME,STYLE,COLOR,FLAMMABLE) VALUES ('GLASSES','MODERN','RED','N');
INSERT INTO PRODUCT (PRODUCT_NAME,STYLE,COLOR,FLAMMABLE) VALUES ('DISHES','PLAIN','WHITE','N');
```

Oracle

Here is the Oracle query to pivot data. We are aging it in four different categories. No. of Invoice pending <= 15 days, <=30 days, <=45days and more than 45 days.

```
SELECT
Client_ID,
SUM (CASE WHEN trunc(sysdate) - Invoice_date > 1 AND trunc(sysdate) - Invoice_date <= 15
      THEN 1
      ELSE 0
      END) as Days15,
SUM (CASE WHEN trunc(sysdate) - Invoice_date > 15 AND trunc(sysdate) - Invoice_date <= 30
      THEN 1
      ELSE 0
      END) as Days30,
SUM (CASE WHEN trunc(sysdate) - Invoice_date > 30 AND trunc(sysdate) - Invoice_date <= 45
      THEN 1
      ELSE 0
      END) as Days45,
SUM (CASE WHEN trunc(sysdate) - Invoice_date > 45
      THEN 1
      ELSE 0
      END) as Morethan45
FROM Invoice
WHERE paid_flag = 0
GROUP BY Client_ID
ORDER BY Client_ID;
```

Now let us check how we can unpivot the data in Oracle. As indicated earlier, let us say we have PRODUCT table defined with product_id, name and its attributes. One drawback to this table approach is whenever we have some new product line coming up, with some different attributes, we need to add an attribute as a new column. This requires

database change and code change as well. In order to avoid this we decided that we would like to store all attributes as name, value pair combination. So now we need to unpivot the data. Following is the example to unpivot the data.

```
SELECT PRODUCT_ID, 'STYLE', STYLE
   FROM Product
UNION ALL
SELECT PRODUCT_ID, 'COLOR', COLOR
   FROM Product
UNION ALL
SELECT PRODUCT_ID, 'FLAMMABLE', FLAMMABLE
   FROM Product
```

Note: One thing to keep in mind is UNION ALL works only when data types of columns are same otherwise, an error will be thrown when SQL is executed. So make sure that all non-character data types are converted to VARCHAR data types if you have columns with different data types.

SQL Server

In MS SQL Server 2000, pivoting is achieved using CASE/GROUP BY statements, i.e. the same way as we did for Oracle. Syntax for this is as below. This query will work in SS2K5 as well.

```
SELECT Client_ID,
       SUM (CASE WHEN Datediff(day,Invoice_date,getDate()) > 1
                AND Datediff(day,Invoice_date,getDate()) <= 15 THEN 1 ELSE 0 END)
       as Days15,
       SUM (CASE WHEN Datediff(day,Invoice_date,getDate()) > 15
                AND Datediff(day,Invoice_date,getDate()) <= 30 THEN 1 ELSE 0 END)
       as Days30,
       SUM (CASE WHEN Datediff(day,Invoice_date,getDate()) > 30
                AND Datediff(day,Invoice_date,getDate()) <= 45 THEN 1 ELSE 0 END)
       as Days45,
       SUM (CASE WHEN Datediff(day,Invoice_date,getDate()) > 45 THEN 1 ELSE 0 END)
       as Morethan45
   FROM dbo.Invoice
  WHERE paid_flag = 0
  GROUP BY Client_ID
 ORDER BY Client_ID
GO
```

Results are as shown under in tabular format.

| ClientID | Days15 | Days30 | Days45 | Morethan45 |
|----------|--------|--------|--------|------------|
| 101 | 2 | 1 | 2 | 2 |
| 102 | 2 | 1 | 1 | 1 |
| 103 | 3 | 1 | 1 | 1 |

Execution plan for above statement (abridged)

```
|--Sort (ORDER BY: ([DECIPHER].[dbo].[Invoice].[CLIENT_ID] ASC))
|--Hash Match (Aggregate, HASH: ([DECIPHER].[dbo].[Invoice].[CLIENT_ID])
DEFINE: ([Expr1003]=SUM(CASE WHEN datediff(day,[DECIPHER].[dbo].[Invoice].[INVOICE_DATE],getdate())>(1)
AND datediff(day,[DECIPHER].[dbo].[Invoice].[INVOICE_DATE],getdate())<=(15) THEN (1) ELSE (0) END)))
|--Clustered Index Scan (OBJECT: ([DECIPHER].[dbo].[Invoice].[PK_INVOICE]),
WHERE: ([DECIPHER].[dbo].[Invoice].[Paid_Flag]=0))
```

As you can see from above, the execution plan shows that the optimizer performed a clustered index scan and then a hash match (for aggregation) and finally sorting.

Let us run the query using PIVOT operator introduced in SS2K5. We are assuming that you are aware with general syntax of PIVOT. For detailed information you can refer BOL.

```
SELECT Client_Id, [1] as Days15, [2] days30, [3] days45, [4] morethan45
FROM
(
SELECT Client_ID,
(CASE
WHEN Datediff(day,Invoice_Date,getDate()) > 1
AND Datediff(day,Invoice_Date,getDate()) <= 15 THEN 1
WHEN Datediff(day,Invoice_Date,getDate()) > 15
AND Datediff(day,Invoice_Date,getDate()) <= 30 THEN 2
WHEN Datediff(day,Invoice_Date,getDate()) > 30
AND Datediff(day,Invoice_Date,getDate()) <= 45 THEN 3
WHEN Datediff(day,Invoice_Date,getDate()) > 45 THEN 4
END
) as days
FROM DBO.Invoice
WHERE paid_flag = 0
) p
pivot
(COUNT(days) for Days IN ([1],[2],[3],[4])
)
as pvt
GO
```

Result will be the same as returned by CASE/GROUP BY query.

| ClientID | Days15 | Days30 | Days45 | Morethan45 |
|----------|--------|--------|--------|------------|
| 101 | 2 | 1 | 2 | 2 |
| 102 | 2 | 1 | 1 | 1 |
| 103 | 3 | 1 | 1 | 1 |

Let us examine now execution plan for the query. Following is the abridged version of plan.

```
|--Compute Scalar (DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[globalagg1010],0),
[Expr1005]=CONVERT_IMPLICIT(int,[globalagg1012],0)))
|--Stream Aggregate(GROUP BY:([DECIPHER].[dbo].[Invoice].[CLIENT_ID])
DEFINE:([globalagg1010]=SUM([partialagg1009]))
|--Sort (ORDER BY:([DECIPHER].[dbo].[Invoice].[CLIENT_ID] ASC))
|--Hash Match(Aggregate, HASH:([DECIPHER].[dbo].[Invoice].[CLIENT_ID])
DEFINE:([partialagg1009]=COUNT(CASE WHEN [Expr1003]=(1) THEN [Expr1003] ELSE NULL END)))
|--Compute Scalar (DEFINE:([Expr1003]=CASE WHEN
datediff(day,[DECIPHER].[dbo].[Invoice].[INVOICE_DATE],getdate())>(1)
|--Clustered Index Scan
(OBJECT:([DECIPHER].[dbo].[Invoice].[PK_INVOICE]))
```

From above execution plan we can see that some extra steps have been performed when we used query with PIVOT operator.

We ran the test with 400806 rows on machine with 4G RAM and 2 CPU (Pentium 4) OS Windows 2003 Service Pack 1. Both queries came back with results in about 1 second. In fact query with PIVOT operator took a little more time.

Limitations of the Pivot/Unpivot operators

Even though the pivot operator has several benefits and is very useful, it suffers from some limitations which we are listing below:

- Value of pivoting columns can be defined by only “IN” expression.
- Also all the values should be known at the time of development of the query. So it is very much static. If column values are not known then we need to resort back to stored procedure approach where we have to build it dynamically. That means, that one will need to write up a stored procedure that takes in a query, the row-column list and dynamically do the pivoting. It can become a performance issue and not only that, one will be exposing their code to SQL injection issues if one does that (though there are ways to mitigate that – we will cover dynamic pivoting in a future whitepaper).

Now let us use convert row data into column data. Make sure that PRODUCT table is created and populated with data. Traditional SQL to convert row data into column data is as under using UNION ALL operator. This will work in both SS2K and SS2K5. As we mentioned earlier, we need to make sure that data type of all column should match.

```
SELECT PRODUCT_ID, 'STYLE' AS ATTRIBUTE, STYLE AS ATTRIBUTE_VALUE
  FROM DBO.PRODUCT WHERE PRODUCT_ID IN (1,2)
UNION ALL
SELECT PRODUCT_ID, 'COLOR', COLOR
  FROM DBO.PRODUCT WHERE PRODUCT_ID IN (1,2)
UNION ALL
SELECT PRODUCT_ID, 'FLAMMABLE', FLAMMABLE
  FROM DBO.PRODUCT WHERE PRODUCT_ID IN (1,2)
```

Result set is shown as under.

| PRODUCT_ID | ATTRIBUTE | ATTRIBUTE_VALUE |
|------------|-----------|-----------------|
| 1 | COLOR | BLUE |
| 1 | FLAMMABLE | Y |
| 1 | STYLE | REGULAR |
| 2 | COLOR | SILVER |
| 2 | FLAMMABLE | N |
| 2 | STYLE | GOTHIC |

Let us try to run query using UNPIVOT operator.

```
SELECT Product_ID,
       Attribute,
       Attribute_Value
  FROM (SELECT PRODUCT_ID, STYLE, COLOR, FLAMMABLE
        FROM DBO.PRODUCT) P
UNPIVOT
 ( ATTRIBUTE_VALUE FOR ATTRIBUTE IN ([STYLE],[COLOR],[FLAMMABLE])
 ) as up
```

Upon execution, query will return following runtime error.

Msg 8167, Level 16, State 1, Line 1
The type of column "FLAMMABLE" conflicts with the type of other columns specified in the UNPIVOT list.

Now change the query as shown and re-run it. This time it will display correct results.

```
SELECT Product_ID,  
       Attribute,  
       Attribute_Value  
FROM (SELECT PRODUCT_ID, STYLE, COLOR, CAST(FLAMMABLE AS VARCHAR(10)) as FLAMMABLE  
      FROM dbo.PRODUCT) P  
UNPIVOT  
( ATTRIBUTE_VALUE FOR ATTRIBUTE IN ([STYLE],[COLOR],[FLAMMABLE])  
 ) as up
```

Difference between two is marked in blue. In case of UNPIVOT relational operator, all the column should exactly be the same. i.e data type and length must match. Data Type and length of all the columns involved in UNPIVOT should match with the column having maximum length. This is true in case of data type bigint and int as well.

Another major difference is, UNPIVOT does not return NULL data from the pivoted data while query using UNION ALL will return null data unless it is excluded using IS NOT NULL condition.

Summary

In this paper, we have explained how we can perform pivoting and unpivoting in Oracle and SQLServer. There are various ways of doing pivoting from which we have shown only 1 way of doing static pivoting for Oracle and DB2 LUW. For SQL Server, we have shown two different approaches: one using the traditional CASE/ GROUP BY syntax and another one using PIVOT/UNPIVOT relational operator introduced in SQLServer 2005. We hope that this will help you in your routine day to day work.