

SQL Server for Developers Advanced T-SQL and Troubleshooting

Module 3: Summarization and Analysis

Bob Beauchemin
BobB@SQLskills.com



Outline

- **Aggregate functions**
 - User-Defined Aggregates
 - Grouping Sets
 - ROLLUP and CUBE
 - Data Warehouse Queries & Columnstore Indexes
- **OVER clause and windowing functions**
 - In SQL Server 2005-2008
 - In SQL Server 2012
 - With analytic and ranking functions
- **TOP**
- **Paging**
- **PIVOT and UNPIVOT**



Aggregates

- **SQL Server supports a superset of ANSI standard aggregate functions**
 - ANSI standard aggregates include COUNT, AVG, MIN, MAX, SUM
 - User-defined aggregate functions using .NET
 - When using most aggregate functions, rows with NULL values are thrown away
 - Count of a column is the count of not NULL values and count(*) is a row count. If the column does not allow null values then count(column) will be as efficient as count(*).
 - COALESCE, and ISNULL can provide defaults for NULL values
 - GROUP BY ALL retains groups that have no matching rows - returned as NULL

Using ISNULL and COALESCE

```
-- substitutes $10.00 if price is NULL
SELECT AVG(ISNULL(price, $10.00)) FROM titles

-- returns first non-null expression
-- NULL if all are NULL
SELECT
COALESCE((zip + 44), (state + 'OR'), county)
  AS LOCATION

-- returns '99999' if zip is
-- anything other than '99999'
-- NOT related to ISNULL()
SELECT state, NULLIF('99999', zip) FROM clients
```

User-Defined Aggregates

- **In SQL Server 2005 and above**
 - .NET class with 4 special methods
 - These are recognized by the query engine
 - Not enabled with window functions
- **Properties can be specified for optimization**
 - On `SqlUserDefinedAggregate` attribute
 - `IsInvariantToNulls` - call with a NULL row
 - Most built in aggregates - true
 - `IsInvariantToDuplicates` - call with dup values
 - MIN, MAX - true
 - `IsNullIfEmpty`
 - Most built in aggregates - true
 - `IsInvariantToOrder`
 - Cannot be used yet, no way for engine to guarantee

ROLLUP Operator

- **Rollup operator adds rows with summaries**
 - Last row is "grand total"
 - Adds summary rows at each group level
 - NULL values in the grouped columns
 - Can be overridden by defining GROUPING columns
 - Or can sort summaries to top

Using the GROUP BY Clause with the ROLLUP Operator

```
USE northwind
SELECT productid, orderid
       , SUM(quantity) AS total_quantity
FROM [order details]
GROUP BY ROLLUP (productid, orderid)
ORDER BY productid, orderid
```

productid	orderid	total_quantity	Description
NULL	NULL	95	Grand total
1	NULL	15	Summarizes only rows for productid 1
1	1	5	Detail value for productid 1, orderid 1
1	2	10	Detail value for productid 1, orderid 2
2	NULL	35	Summarizes only rows for productid 2
2	1	10	Detail value for productid 2, orderid 1
2	2	25	Detail value for productid 2, orderid 2
3	NULL	45	Summarizes only rows for productid 3
3	1	15	Detail value for productid 3, orderid 1
3	2	30	Detail value for productid 3, orderid 2

Cube Operator

- CUBE operator is a cross tabulation
 - Groups summary column with every value of other columns
 - Adds special "ALL" columns

Using the GROUP BY Clause with the CUBE Operator

```
USE northwind
SELECT productid, orderid, SUM(quantity) AS
total_quantity
FROM [order details]
GROUP BY CUBE(productid, orderid)
ORDER BY productid, orderid
```

The CUBE operator produces two more summary values than the ROLLUP operator

productid	orderid	total_quantity	Description
NULL	NULL	95	Grand total
NULL	1	30	Summarizes all rows for orderid 1
NULL	2	65	Summarizes all rows for orderid 2
1	NULL	15	Summarizes only rows for productid 1
1	1	5	Detail value for productid 1, orderid 1
1	2	10	Detail value for productid 1, orderid 2
2	NULL	35	Summarizes only rows for productid 2
2	1	10	Detail value for productid 2, orderid 1
2	2	25	Detail value for productid 2, orderid 2
3	NULL	45	Summarizes only rows for productid 3
3	1	15	Detail value for productid 3, orderid 1
3	2	30	Detail value for productid 3, orderid 2

Grouping Sets

- **Grouping Sets allow multiple GROUP BY clauses in a single SQL statement**
 - Multiple, arbitrary, sets of subtotals
 - Single read pass for performance
 - Nested subtotals provide ever better performance
- **Grouping Sets are an ANSI-standard**
 - COMPUTE BY is deprecated
- **ANSI syntax for ROLLUP and CUBE in SQL2008**
 - Pre-2008 non-ANSI syntax is deprecated

Query Plan Enhancements

- **WITH ROLLUP** - $n+1$ different groupings of data
 - Where n is the number of columns in GROUP BY
 - One aggregate and 1 rollup aggregate
- **WITH CUBE** produces $2n$ different groupings
 - Where n is the number of columns in GROUP BY
 - Multiple aggregates and rollup aggregates
- **GROUPING SETS** provide a "halfway measure"
 - More groupings than ROLLUP, but less than CUBE
 - Rollup aggregates dependent on which sets

GROUPING_ID and GROUPING

- **Grouping Sets can produce non-homogeneous sets**
 - Grouping set includes NULL values for group members
 - Need to distinguish by grouping and NULL values
- **GROUPING (column expression) returns 0 or 1**
 - Is this a group based on column expr. or NULL value?
- **GROUPING_ID (a,b,c) is a bitmask**
 - GROUPING_ID bits are set based on column expressions a, b, and c

Columnstore Indexes

- **Data warehouse star or snowflake schema consists of**
 - Fact Tables – large number of rows, wide rows
 - Dimension Tables – Small tables to group by
- **Fact Tables are notoriously difficult to index**
 - Aggregates are pre-calculated offline for speed
- **Data warehouse queries**
 - Fact table with aggregation limited by dimensions
 - Columnstore indexes help here
 - Without columnstore – can use Bitmaps

xVelocity (was Vertipac)

- **Columnstore Indexes based on xVelocity**
 - Introduced in PowerPivot
 - In SQL Server 2012, implemented in SSAS and database engine
- **Combines**
 - Column-based storage
 - Smart compression and encoding
- **In database engine**
 - Batch mode to speed up query even more
- **Profound speedup for data warehouse queries**
 - 10-100 times faster
 - Not all queries benefit

Columnstore is Read Only

- **Columnstore indexes are read-only**
 - Entire table must be read-only
 - Usually the case with data warehouse
- **To create columnstore**
 - Create table
 - Load data
 - Define columnstore index (makes table read-only)

Columnstore and Query Plans

- **New iterator added for columnstore**
 - BatchProcessing (Batch Hash Table Build)
- **Optimizer is still cost-based**
 - Optimizer can choose index
 - Optimizer can choose batch mode
 - Batch mode not always used
- **Can observe in query plan**
 - Columnstore index use
 - Batch execution mode
- **Query hints can be used**
 - Some queries slower with columnstore

Window Functions

- **Applied against a set of rows**
 - Returns one row per input row
 - vs. GROUP BY returns one row per group
- **Specified using OVER clause**
 - In SELECT or ORDER BY clauses
- **Introduced in SQL Server 2005**
 - Useable with ranking functions
 - ROW_NUMBER, RANK, DENSE_RANK, NTILE(n)
 - Partially usable with aggregate functions
 - Only PARTITION BY allowed

SQL 2012 Windowing Extensions

- **Window Frame specification**
- **Order and frame clauses with aggregates**
- **Offset functions**
 - LAG/LEAD
 - FIRST_VALUE/LAST_VALUE
- **Additional Analytical Functions**
 - Distribution functions
 - PERCENT_RANK, CUME_DIST
 - Inverse distribution functions
 - PERCENTILE_CONT, PERCENTILE_DISC

Window Function Uses

- **Window functions can eliminate most remaining usage for cursors, including...**
 - Running totals
 - Moving averages
 - Moving sums
 - Median values
 - Find percentages within a group
- **Performance, ease-of-use advantage over**
 - Grouped queries
 - Subqueries

Window Function Specifications

- **Used with OVER clause**
- **Partition By clause**
 - Divides rows into groups
 - Restricts set to rows with same partition value
- **Order By clause**
 - Specifies ordering within a partition
- **Frame clause**
 - Restrict function to a subset of rows
 - Based on ordering in Order By

OVER Clause and Functions

	OVER Clause	PARTITION BY	ORDER BY	Window Frame Clause
Aggregate	Optional	Optional	Optional	Optional w/Default
Ranking PERCENT_RANK CUME_DIST	Required	Optional	Required	No
PERCENTILE_CONT PERCENTILE_DISC	Required	Optional	WITHIN GROUP clause	No
LAG/LEAD	Required	Optional	Required	No
FIRST_VALUE LAST_VALUE	Required	Optional	Required	Optional w/Default
NEXT VALUE FOR	Optional	No	Required	No

Window Frame Specification

- Framing enables running aggregates
- Specify ROWS or RANGE
 - RANGE can be
 - UNBOUNDED PRECEDING / UNBOUNDED FOLLOWING
 - CURRENT ROW
 - ROWS can be
 - All of the above plus
 - Number of rows (Unsigned integer literal)
- RANGE UNBOUNDED PRECEDING AND CURRENT ROW is default
 - If ROWS/RANGE not specified but ORDER BY is

Offset Functions

- **Used with OVER() clause**
 - Order by is required, partition by optional
- **LAG and LEAD**
 - Return rows at offset from current row
 - Three parameters
 - Scalar Expression
 - Offset – defaults to 1
 - Default – Value to return if NULL
- **FIRST_VALUE and LAST_VALUE**
 - Return first or last value of the set
 - Be careful with default window frame spec

ROWS vs. RANGE

- **ROWS is a physical aggregation group**
 - Data must be dense
 - Answer may be non-deterministic
 - Can have multiple sort keys
- **RANGE is a logical aggregation group**
 - Data can be sparse
 - Answer is deterministic
 - Single sort key

Distribution Functions

- **PERCENT_RANK and CUME_DIST**
 - Over clause requirements like ranking functions
 - **PERCENT_RANK**
 - $(\text{Rank of Row} - 1) / (\text{NumRows} - 1)$
 - Return value between 0 and 1 inclusive
 - **CUME_DIST**
 - "NumRows <= current row" / NumRows
 - Return value > 0 and <= 1

Inverse Distribution Functions

- **Special "Within Group" specification**
 - PERCENTILE_CONT/PERCENTILE_DISC
(numeric_literal)
 - WITHIN GROUP
(ORDER BY order_by_expression [ASC | DESC])
 - OVER ([<partition_by_clause>])
- **PERCENTILE_DISC – Discrete Distribution**
 - Value must be an existing row value
- **PERCENTILE_CONT – Continuous Distribution**
 - Value is an interpolated value
- **Numeric literal of 50 is median**

Windowing Performance

- **Best indexing for window would be**
 - Column(s) in PARTITION BY clause
 - Column(s) in ORDER BY clause
 - Additional columns for covering (if desired)
- **Uses WindowSpool plan iterator**
 - Two optimizations of WindowSpool based on window spec.
- **Prefer ROWS to RANGE**
 - Range always uses worktable

Listing the TOP n Values

- **Lists Only the First n Rows of a Result Set**
- **Specifies the range of values in the ORDER BY clause**
 - non-deterministic without ORDER BY
- **Returns "Ties" if WITH TIES Is Used**

```
USE northwind
SELECT TOP 5 orderid, productid, quantity
FROM [order details]
ORDER BY quantity DESC
```

```
USE northwind
SELECT TOP 5 WITH TIES orderid, productid, quantity
FROM [order details]
ORDER BY quantity DESC
```

TOP Enhancement

- **TOP queries can be based on**
 - Number of rows
 - Percentage of rows
- **In SQL Server 2005/2008**
 - An integral variable or expression
 - Can use TOP with INSERT, UPDATE, DELETE
 - more on this later

```
CREATE FUNCTION get_first_page (  
    @rows_per_page int  
)  
AS  
SELECT TOP (@rows_per_page)  
   orderid, productid, quantity  
FROM [order details]  
ORDER BY quantity DESC
```

Paging – OFFSET and FETCH

- **ANSI SQL Standard Paging**
 - ORDER BY with OFFSET and FETCH NEXT
- **Can use**
 - Hardcoded number
 - Variables
 - Results from scalar subquery
- **Page data consistency requires**
 - Transactions and snapshot isolation

PIVOT and UNPIVOT

- **PIVOT**
 - Converts row values into column headers
- **UNPIVOT**
 - Converts column headers into row values
- **Pre-SQL 2005**
 - CASE statement – lengthy code
- **PIVOT and UNPIVOT are table operators**

Pivot Using SELECT CASE

- **Each row in ProductInventory contains a ProductID**
 - You want to summarize quantity by product
 - With ProductID as column heading
 - You can accomplish this using SELECT statement with a CASE clause for every value of ProductID

```
SELECT
SUM(CASE WHEN ProductID = 316 THEN Quantity END) AS Blade,
SUM(CASE WHEN ProductID = 331 THEN Quantity END) AS [Fork End],
SUM(CASE WHEN ProductID = 332 THEN Quantity END) AS Freewheel,
SUM(CASE WHEN ProductID = 527 THEN Quantity END) AS Spokes,
SUM(CASE WHEN ProductID = 529 THEN Quantity END) AS Stem
FROM Production.ProductInventory
```

PIVOT

- **PIVOT converts row values into columns**
 - Allows aggregate calculations
 - Rotates a table-valued expression
 - Turns a unique value into multiple columns
 - Performs aggregations on remaining columns

```
SELECT [316] AS Blade, [331] AS [Fork End],  
       [332] AS Freewheel, [527] AS Spokes, [529] AS Stem  
FROM  
    (SELECT ProductID, Quantity  
     FROM Production.ProductInventory) AS pinv  
PIVOT  
    (  
        SUM (Quantity)  
        FOR ProductID IN ([316], [331], [332], [527], [529])  
    ) AS pvt;  
GO
```

PIVOT - how it works

- **Same place as JOIN in logical execution order**
- **Three "steps" in execution**
 - Creates groups for all columns not in IN clause or aggregate clause
 - Adds one column for each member of IN clause
 - Evaluates using the aggregate function

UNPIVOT

- UNPIVOT
- Reverses the PIVOT operation
- Normalizes the pre-pivoted data
- Rotates columns into rows
- Eliminates NULLs in the results value column
- Not an exact reversal of the PIVOT operator
 - If PIVOT produced aggregates

PIVOT and UNPIVOT

```
CREATE TABLE CarSales (Make varchar(6), Year int, Sales int)
go
INSERT INTO CarSales VALUES ('Honda', 1990, 2000)
INSERT INTO CarSales VALUES ('Honda', 1990, 1000)
INSERT INTO CarSales VALUES ('Acura', 1990, 500)
INSERT INTO CarSales VALUES ('Honda', 1991, 3000)
INSERT INTO CarSales VALUES ('Acura', 1991, 300)
INSERT INTO CarSales VALUES ('Acura', 1991, 600)
INSERT INTO CarSales VALUES ('Acura', 1992, 800)
go
```

```
-- One row for Honda, One row for Acura, years are columns
SELECT * INTO CarSalesPivot
from CarSales
PIVOT (SUM(Sales) FOR Year IN ([1990], [1991])) t
GO
-- This returns 4 rows vs 7 in original table
select *
from CarSalesPivot
UNPIVOT (Sales for Year in ([1990], [1991])) t
```

UNPIVOT - how it works

- Same place as JOIN in logical execution order
- Three steps in execution
 - Generates duplicate rows
 - Number of members in IN clause * Input Rows
 - Isolate target column values
 - Filters rows with NULLs

Summary

- Using Aggregates
 - GROUP BY and HAVING
 - ROLLUP and CUBE
 - GROUPING SETS
 - User-defined aggregates
- Beyond Traditional Aggregation
 - Window Functions
 - Ranking and Analytic Functions
 - TOP
 - OFFSET and FETCH
 - PIVOT

References

- A Developer's Guide to SQL Server 2005 - Ch 8, Addison-Wesley, *Bob Beauchemin and Dan Sullivan*
- Microsoft SQL Server 2012 High-Performance T-SQL Using Window Functions, *Itzik Ben-Gan*

Questions?