Structured Query Language

Aggregates

Grouping and Aggregates

- Aggregates are operator which has set operand
- Group By it is just a clause which forms sets which are passed on to aggregates operator / functions

Grouping and Aggregates

- sometimes you will want to find trends in your data that will require database server to cook data a bit before you can generate results
- SELECT EmployeeID FROM Orders ORDER BY EmployeeID
- With only 24 rows in the account table, it is relatively easy to see that four different
- with many rows difficult to see the number of orders made by employees

Grouping and Aggregates

- can ask database server to group data for you by using group by clause
- SELECT EmployeeID FROM Orders
- GROUP BY EmployeeID
- EmployeeID
- 1
- 2
- 3
- …
- 8
- 9

Grouping and Aggregates

- to see how many Orders each Employee made you can use *aggregate function* in select clause to count number of rows in each group
- SELECT EmployeeID, COUNT(EmployeeID) AS HowMany FROM Orders GROUP BY EmployeeID
- EmployeeID HowMany
- 1 123
- 2 96
- 3 127
- …
- 8 104
- 9 43

count()

- aggregate function counts number of rows in each group, and asterisk tells server to count everything in group
- SELECT EmployeeID, COUNT(*) AS HowMany
- FROM Orders
- GROUP BY EmployeeID

Having clause

- since group by clause runs *after* where clause has been evaluated, you cannot add filter conditions to your where clause
- SELECT EmployeeID, COUNT(*) AS HowMany FROM Orders
- GROUP BY EmployeeID HAVING COUNT(*) > 100
- EmployeeID HowMany
- 1 123
- 3 127
- 4 156
- 8 104

Having clause

- groups containing fewer than hundred orders have been filtered out via having clause, result set now contains only those employees who have made more then hundred orders

SELECT * FROM TestAggregation

Results | Messages

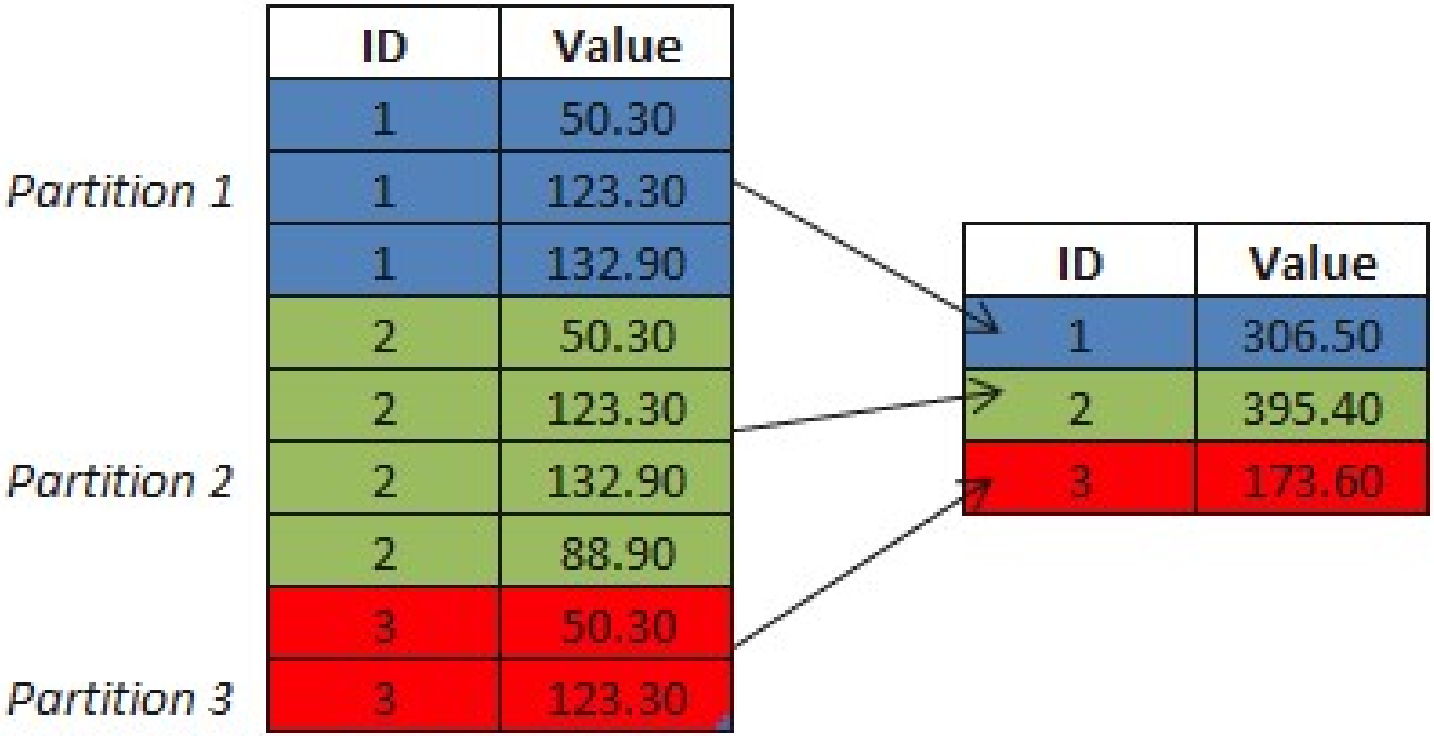
	ID	Value
1	1	50.30
2	1	123.30
3	1	132.90
4	2	50.30
5	2	123.30
6	2	132.90
7	2	88.90
8	3	50.30
9	3	123.30

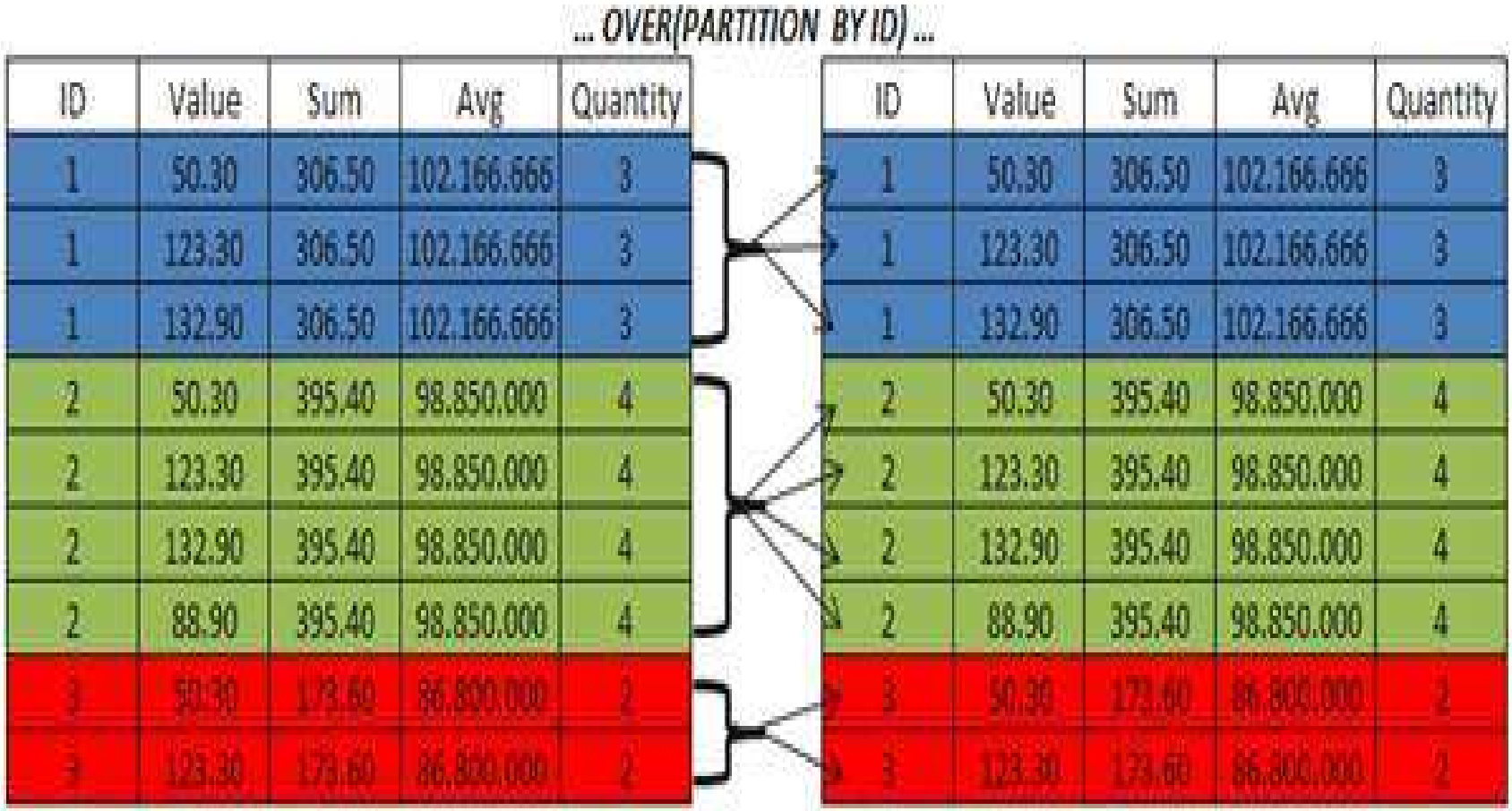
SELECT ID, SUM(Value)
FROM TestAggregation
GROUP BY ID;

Results | Messages

	ID	(No column name)
1	1	306.50
2	2	395.40
3	3	173.60

Partitions


ID
Value
1
50.30
Partition 1
1
123.30
1
132.90
2
50.30
2
123.30
Partition 2
2
132.90
2
88.90
3
50.30
Partition 3
3
123.30
ID
Value
1
306.50
2
395.40
3
173.60


... OVER(PARTITION BY ID) ...
ID Value Sum Avg Quantity
1 50.30 306.50 102.166.666 3
1 123.30 306.50 102.166.666 3
1 132.90 306.50 102.166.666 3
2 50.30 395.40 98.850.000 4
2 123.30 395.40 98.850.000 4
2 132.90 395.40 98.850.000 4
2 88.90 395.40 98.850.000 4
3 50.30 173.60 86.800.000 2
3 123.30 173.60 86.800.000 2
ID Value Sum Avg Quantity
1 50.30 306.50 102.166.666 3
1 123.30 306.50 102.166.666 3
1 132.90 306.50 102.166.666 3
2 50.30 395.40 98.850.000 4
2 123.30 395.40 98.850.000 4
2 132.90 395.40 98.850.000 4
2 88.90 395.40 98.850.000 4
3 50.30 173.60 86.800.000 2
3 123.30 173.60 86.800.000 2

Aggregate Functions

- perform specific operation over all rows in group
- belong to type of function known as ‘set function’, means function that applies to set of rows
- common aggregate functions implemented by all major servers include:

Aggregate Functions

- Max() - returns maximum value within set
- Min() - returns minimum value within set
- Avg() - returns average value across set
- Sum() - returns sum of values across set
- Count() - returns number of values in set

- SELECT
- MAX(UnitPrice) max_price,
- MIN(UnitPrice) min_price,
- AVG(UnitPrice) avg_price,
- SUM(UnitPrice) total_price,
- COUNT(UnitPrice) num_products
- FROM Products JOIN Categories ON Categories.CategoryID = Products.CategoryID

max_price min_price avg_price total_price num_products

- 263.50 0.00 28.4962 2222.71 78

- results from this query tell you that, across all products there is maximum value of UnitPrice $263.50, a minimum values of $0.00, an average value of UnitPrice of $28.49, and total values of $2,222.71 across all 78 products
- every value returned by query is generated by aggregate function, and since there is no group by clause, there is a single, *implicit* group (all rows returned by query)

- in most cases, however, you will want to retrieve additional columns along with columns generated by aggregate functions
- what if, for example, you wanted to extend previous query to execute the same aggregate functions for *each* product category type, instead of just for checking all products
- you would want to retrieve product's category name column along with aggregate functions

- SELECT CategoryName,
- MAX(UnitPrice) max_price,
- MIN(UnitPrice) min_price,
- AVG(UnitPrice) avg_price,
- SUM(UnitPrice) total_price,
- COUNT(UnitPrice) num_products
- FROM Products JOIN Categories ON Categories.CategoryID = Products.CategoryID
- GROUP BY CategoryName

CategoryName max_price min_price avg_price total_price num_products

- Beverages 263.50 0.00 35.05 455.75 13
- Condiments 43.90 10.00 23.06 276.75 12
- Confections 81.00 9.20 25.16 327.08 13
- Dairy Products 55.00 2.50 28.73 287.30 10
- Grains/Cereals 38.00 7.00 20.25 141.75 7
- Meat/Poultry 123.79 7.45 54.00 324.04 6
- Produce 53.00 10.00 32.37 161.85 5
- Seafood 62.50 6.00 20.68 248.19 12

- with the inclusion of group by clause, server knows to group together rows having same value in CategoryName column first and then to apply aggregate functions to each groups

Counting Distinct Values

- when using count() function to determine number of members in each group, have choice of counting *all* members or counting only *distinct* values for column across all members of group
- sum(), max(), min(), and avg() functions all ignore any null value

- SELECT COUNT(*) FROM Customers
- 92 *rows in table Customers*
- SELECT COUNT(Region) FROM Customers
- 32 *rows in which Region column it is not NULL*
- SELECT COUNT(DISTINCT Region) FROM Customers
- 19 *such distinct values representing Region*

Grouping

- single-column
- multicolumn
- via expressions

- SELECT CategoryName, Products.ProductName,
- MAX(UnitPrice) max_price,
- MIN(UnitPrice) min_price,
- AVG(UnitPrice) avg_price,
- SUM(UnitPrice) total_price,
- COUNT(UnitPrice) num_products
- FROM Products JOIN Categories ON Categories.CategoryID = Products.CategoryID
- GROUP BY CategoryName

Error

- Column 'Products.ProductName' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause
- on grouping and using aggregates you cannot use in SELECT clause expression which are not either aggregate or are used in GROUP BY
- previous error – server don’t know what productName to present

- SELECT CategoryName
- MAX(UnitPrice) max_price,
- MIN(UnitPrice) min_price,
- AVG(UnitPrice) avg_price,
- SUM(UnitPrice) total_price,
- COUNT(UnitPrice) num_products
- FROM Products JOIN Categories ON Categories.CategoryID = Products.CategoryID
- GROUP BY CategoryID

- SELECT CategoryName,
- MAX(UnitPrice) max_price,
- MIN(UnitPrice) min_price,
- AVG(UnitPrice) avg_price,
- SUM(UnitPrice) total_price,
- COUNT(UnitPrice) num_products
- FROM Products JOIN Categories ON Categories.CategoryID = Products.CategoryID
- GROUP BY Products.CategoryID

SubQueries

subquery

- query contained within another SQL statement (which we refer to as *containing statement*
- always enclosed within parentheses
- usually executed prior to containing statement
- returns result set that may consist of:
  - single row with single column
  - multiple rows with single column
  - multiple rows and columns

- when containing statement has finished executing, data returned by any subqueries is discarded, making subquery act like temporary table with *statement scope*
- one of most powerful tools in SQL

- if you use subquery in equality condition, but subquery returns more than one row, you will receive error
- SELECT * FROM Products WHERE UnitPrice = (
- SELECT MAX(UnitPrice) FROM Products)
- *better* - single thing cannot be equated to set
- SELECT * FROM Products WHERE UnitPrice IN (
- SELECT MAX(UnitPrice) FROM Products)

- along with differences regarding type of result set subquery returns (single/multiple rows and columns)
- subqueries are completely selfcontained (called *noncorrelated subqueries*), while others reference columns from containing statement (called *correlated subqueries*)

Correlated Subqueries

- is *dependent* on its containing statement from which it references one or more columns
- executed once for each candidate row (rows that might be included in final results

- noncorrelated subquery is executed once prior to execution of containing statement

- SELECT DISTINCT Customers.CompanyName FROM Orders JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID
- JOIN Products ON Products.ProductID = OrderDetails.ProductID
- JOIN Categories ON Categories.CategoryID = Products.CategoryID
- JOIN Customers ON Orders.CustomerID = Customers.CustomerID
- WHERE Categories.CategoryName = 'Beverages'
- ORDER BY CompanyName

- SELECT DISTINCT Customers.CompanyName FROM Customers
- WHERE Customers.CustomerID IN *(*
- *SELECT DISTINCT Orders.CustomerID FROM Orders JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID*
- *JOIN Products ON Products.ProductID = OrderDetails.ProductID*
- *JOIN Categories ON Categories.CategoryID = Products.CategoryID*
- *WHERE Categories.CategoryName = 'Beverages')*
- ORDER BY CompanyName

- Customers who Ordered Beverages
- Customers who do not Ordered Beverages

- SELECT DISTINCT Customers.CompanyName FROM Orders JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID
- JOIN Products ON Products.ProductID = OrderDetails.ProductID
- JOIN Categories ON Categories.CategoryID = Products.CategoryID
- JOIN Customers ON Orders.CustomerID = Customers.CustomerID
- WHERE Categories.CategoryName != 'Beverages'
- ORDER BY CompanyName

- SELECT DISTINCT Customers.CompanyName FROM Customers
- WHERE Customers.CustomerID NOT IN (
- SELECT DISTINCT Orders.CustomerID FROM Orders JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID
- JOIN Products ON Products.ProductID = OrderDetails.ProductID
- JOIN Categories ON Categories.CategoryID = Products.CategoryID
- WHERE Categories.CategoryName = 'Beverages')
- ORDER BY CompanyName

all operator

- in (membership) operator is used to see whether an element, expression can be found within set
- all operator allows you to make comparisons between single value and every value in set
- to build such condition, you will need to use one of comparison operators (=, <>, <, >, etc.) in conjunction with all operator

any operator

- any operator allows value to be compared to members of set of values
- unlike all, however, condition using any operator evaluates to true as soon as single comparison is favorable
- all operator evaluates to true only if comparisons against *all* members of set are favorable

Set-Comparison Operators

- SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators {<,<=,=,<>,>=,>}
- SOME is also available, but it is just a synonym for ANY

Definition of Some (ANY) Clause

- F <comp> **some** $r \Leftrightarrow \exists\ t \in r$ *s.t. (*F <comp> t)
  Where <comp> can be: $<, \leq, >, =, \neq$

(5< **some** | 0 / 5 / 6 |) = true
(read: 5 < some tuple in the relation)

(5< **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 $\neq$ **some** | 0 / 5 |) = true (since 0 $\neq$ 5)

(= **some**) $\equiv$ **in**
However, ($\neq$ **some**) $\equiv$ **not in**

Definition of All (ALL) Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

(5< **all** | 0, 5, 6 |) = false

(5< **all** | 6, 10 |) = true

(5 = **all** | 4, 5 |) = false

(5 ≠ **all** | 4, 6 |) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≡ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$
- except Operator

- Most expensive Beverage
- Most expensive Product from each Category

- SELECT * FROM Products
- WHERE UnitPrice = (
- SELECT MAX(UnitPrice) FROM Products JOIN Categories ON Products.CategoryID = Categories.CategoryID
- WHERE Categories.CategoryName = 'Beverages')

- SELECT * FROM Products P1 JOIN Categories C1 ON P1.CategoryID = C1.CategoryID
- WHERE C1.CategoryName = 'Beverages' AND UnitPrice = (SELECT MAX(UnitPrice) FROM Products P2 JOIN Categories C2 ON P2.CategoryID = C2.CategoryID
- WHERE C2.CategoryName = 'Beverages')

- SELECT C1.CategoryName, P1.ProductName, P1.UnitPrice FROM Products P1 JOIN Categories C1 ON P1.CategoryID = C1.CategoryID
- WHERE UnitPrice >= ALL (
- SELECT UnitPrice FROM Products P2 JOIN Categories C2 ON P2.CategoryID = C2.CategoryID
- WHERE C2.CategoryName = C1.CategoryName)

CategoryName ProductName UnitPrice

- Seafood Carnarvon Tigers 62.50
- Confections Sir Rodney's Marmalade 81.00
- Meat/Poultry Thüringer Rostbratwurst 123.79
- Beverages Côte de Blaye 263.50
- Produce Manjimup Dried Apples 53.00
- Grains/Cereals Gnocchi di nonna Alice 38.00
- Dairy Products Raclette Courdavault 55.00
- Condiments Vegie-spread 43.90

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$
- except Operator

- Find Customers who Ordered Products from all Categories

CREATE VIEW CustomCateg AS

- SELECT Customers.CompanyName, Categories.CategoryName
- FROM Customers JOIN Orders ON Orders.CustomerID = Customers.CustomerID
- JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
- JOIN Products ON Products.ProductID = OrderDetails.ProductID
- JOIN Categories ON Categories.CategoryID = Products.CategoryID

SELECT * FROM CustomCateg

• CompanyName	CategoryName
1. Alfreds Futterkiste	Produce
2. Alfreds Futterkiste	Beverages
3. Alfreds Futterkiste	Seafood
4. Alfreds Futterkiste	Condiments
5. Alfreds Futterkiste	Dairy Products
6. Ana Trujillo Emparedados y helados	Dairy Products
7. Ana Trujillo Emparedados y helados	Beverages
8. Ana Trujillo Emparedados y helados	Produce
9. Ana Trujillo Emparedados y helados	Grains/Cereals
10. Ana Trujillo Emparedados y helados	Seafood
11. …	

- SELECT CategoryName FROM CustomCateg WHERE CompanyName = 'Alfreds Futterkiste'
- CategoryName

1. Beverages
2. Condiments
3. Dairy Products
4. Produce
5. Seafood

- SELECT CategoryName FROM CustomCateg WHERE CompanyName = 'QUICK-Stop'
- CategoryName

1. Beverages
2. Condiments
3. Confections
4. Dairy Products
5. Grains/Cereals
6. Meat/Poultry
7. Produce
8. Seafood

- SELECT Categories.CategoryName FROM Categories EXCEPT
- SELECT CategoryName FROM CustomCateg WHERE CompanyName = 'Alfreds Futterkiste'
- CategoryName

1. Confections
2. Grains/Cereals
3. Meat/Poultry

- SELECT Categories.CategoryName FROM Categories EXCEPT
- SELECT CategoryName FROM CustomCateg WHERE CompanyName = 'QUICK-Stop'
- CategoryName

- result it is the Ø empty set

- SELECT CompanyName, COUNT(*) FROM CustomCateg
- GROUP BY CompanyName
- ORDER BY 2 DESC

- SELECT DISTINCT CompanyName FROM CustomCateg C1
- WHERE NOT EXISTS (
- SELECT Categories.CategoryName FROM Categories
- EXCEPT
- (SELECT C2.CategoryName FROM CustomCateg C2
- WHERE C2.CompanyName = C1.CompanyName))

exists Operator

- most common operator used to build conditions that utilize correlated subqueries is the exists operator
- to identify that relationship exists without regard for the quantity
- subquery can return zero, one, or many rows, and condition simply checks whether the subquery returned any rows

Data Manipulation Using Correlated Subqueries

- UPDATE table t1
- SET t1.column =
- (SELECT expression
- FROM table t2
- WHERE t1.some_column = t2.some_other_column);

Data Manipulation Using Correlated Subqueries

- SELECT Products.ProductID, Products.UnitPrice
- FROM Products
- ORDER BY ProductID

- SELECT OrderDetails.ProductID, SUM(OrderDetails.Quantity*OrderDetails.UnitPrice) / SUM(OrderDetails.Quantity) AS AvgPrice
- FROM OrderDetails
- GROUP BY OrderDetails.ProductID
- ORDER BY ProductID

ProductID	UnitPrice
1	18.00
2	19.00
3	10.00
4	22.00
5	21.35
6	25.00
7	30.00
8	40.00
9	97.00
10	31.00
11	21.00
12	38.00

ProductID	AvgPrice
1	17.2434
2	17.5583
3	9.3902
4	20.8052
5	19.4669
6	24.4019
7	29.4416
8	36.9892
9	92.9157
10	29.8385
11	19.6912
12	37.4034

- UPDATE Products
- SET Products.UnitPrice =
- (SELECT POD.AvgPrice) FROM (
- SELECT OrderDetails.ProductID, SUM(OrderDetails.Quantity*OrderDetails.UnitPrice) / SUM(OrderDetails.Quantity) AS AvgPrice
- FROM OrderDetails
- GROUP BY OrderDetails.ProductID) POD
- JOIN Products ON POD.ProductID = Products.ProductID

Temp. table

- SELECT POD.ProductID, POD.AvgPrice FROM (
- SELECT OrderDetails.ProductID, SUM(OrderDetails.Quantity*OrderDetails.UnitPrice) / SUM(OrderDetails.Quantity) AS AvgPrice
- FROM OrderDetails
- GROUP BY OrderDetails.ProductID) POD
- JOIN Products ON POD.ProductID = Products.ProductID

- Correlated subqueries are also common in delete statements
- run data maintenance script at end of period that removes unnecessary data.
- removes data from department table that has no child rows in employee table:
- DELETE FROM department
- WHERE NOT EXISTS (SELECT 1
- FROM employee
- WHERE employee.dept_id = department.dept_id);

Outer JOIN

- in all examples thus far that have included multiple tables, we haven't been concerned that join conditions might fail to find matches for all rows in tables
- for example, when joining Categories table to Products table, we did not mention possibility that
- value in CategoryID column of Products table might not match value in CategoryID column of Categories table
- if that were the case, then some of rows in one table (from left side) or other (from right side) would be left out of result set

- SELECT * FROM Categories INNER JOIN Products ON Categories.CategoryID = Products.CategoryID

- SELECT * FROM Categories LEFT OUTER JOIN Products ON Categories.CategoryID = Products.CategoryID
- SELECT * FROM Categories RIGHT OUTER JOIN Products ON Categories.CategoryID = Products.CategoryID
- SELECT * FROM Categories FULL OUTER JOIN Products ON Categories.CategoryID = Products.CategoryID

- if you want your query to return *all* Products, *all* Categories need an *outer join* (FULL OUTER JOIN
- to include Categories without Products
- to include Products without Categories

- columns are Null for all rows except for the matching values Categories.CategoryID = Products.CategoryID

Self Outer Joins

- SELECT s.EmployeeID, s.FirstName, s.LastName, s.Title, s.ReportsTo, m.EmployeeID, m.FirstName, m.LastName, m.Title
- FROM Employees s INNER JOIN Employees m ON s.ReportsTo = m.EmployeeID

Self Outer Joins

- SELECT s.EmployeeID, s.FirstName, s.LastName, s.Title, s.ReportsTo, m.EmployeeID, m.FirstName, m.LastName, m.Title
- FROM Employees s LEFT OUTER JOIN Employees m ON s.ReportsTo = m.EmployeeID

- EmployeeID FirstName LastName Title ReportsTo EmployeeID FirstName LastName Title
- 1 Nancy Davolio Sales Representative 2 2 Andrew Fuller Vice President, Sales
- *2 Andrew Fuller Vice President, Sales NULL NULL NULL NULL NULL*
- 3 Janet Leverling Sales Representative 2 2 Andrew Fuller Vice President, Sales
- 4 Margaret Peacock Sales Representative 2 2 Andrew Fuller Vice President, Sales
- 5 Steven Buchanan Sales Manager 2 2 Andrew Fuller Vice President, Sales
- …

Cross Joins

- Cartesian product - result of joining multiple tables without specifying any join conditions
- are not so common … if, however, you *do* intend to generate Cartesian product of two tables specify *cross join*:
- SELECT * FROM table t1 CROSS JOIN table t2;

CASE

- in certain situations, you may want your SQL logic to branch in one direction or another depending on values of certain expressions.
- statements that can behave differently depending on data encountered during execution

SELECT TitleOfCourtesy, FirstName, LastName FROM Employees

- TitleOfCourtesy FirstName LastName
- Ms. Nancy Davolio
- Dr. Andrew Fuller
- Ms. Janet Leverling
- Mrs. Margaret Peacock
- Mr. Steven Buchanan
- Mr. Michael Suyama
- Mr. Robert King
- Ms. Laura Callahan
- Ms. Anne Dodsworth

- SELECT
- CASE
  - WHEN TitleOfCourtesy = 'Mr.’
    - THEN 'Mister’
  - WHEN TitleOfCourtesy = 'Mrs.’
    - THEN 'Missus’
  - WHEN TitleOfCourtesy = 'Ms.’
    - THEN 'Miss’
  - WHEN TitleOfCourtesy = 'Dr.’
    - THEN 'Doctor’
  - ELSE ' ' END Tit,
- FirstName, LastName FROM Employees

- CASE
  - WHEN TitleOfCourtesy = 'Mr.’
    - THEN 'Mister’
  - WHEN TitleOfCourtesy = 'Mrs.’
    - THEN 'Missus’
  - WHEN TitleOfCourtesy = 'Ms.’
    - THEN 'Miss’
  - WHEN TitleOfCourtesy = 'Dr.’
    - THEN 'Doctor’
  - ELSE ' ' END Tit,

- you could use conditional logic via *case expression*
- CASE
  - WHEN C_1 THEN E_1
  - WHEN C_2 THEN E_2
- …
  - WHEN C_n THEN E_n
  - [ELSE ED]
- END

- symbols C_1, C_2,..., C_n represent conditions
- symbols E_1, E_2,..., E_n represent expressions to be returned by case expression
- if condition in when clause evaluates to true, then case expression returns corresponding expression
- ED symbol represents default expression, which case expression returns if *none* of conditions evaluate to true
  - else clause is optional

- although previous example returns string expressions, keep in mind that case expressions may return any type of expression, including subqueries – but with errors if subquery returned more than 1 value

```
SELECT E.Title, E.FirstName, E.LastName,
CASE
  WHEN E.ReportsTo IS NOT NULL THEN
  (SELECT CONCAT(i.FirstName, ' ', i.LastName)
FROM Employees i
   WHERE i.EmployeeID = E.ReportsTo)
  ELSE 'Unknown'
  END Boss
FROM Employees E;
```

• Title FirstName	LastName	Boss
• Sales Representative	Nancy Davolio	Andrew Fuller
• Vice President, Sales	Andrew Fuller	Unknown
• Sales Representative	Janet Leverling	Andrew Fuller
• Sales Representative	Margaret Peacock	Andrew Fuller
• Sales Manager	Steven Buchanan	Andrew Fuller
• Sales Representative	Michael Suyama	Steven Buchanan
• Sales Representative	Robert King	Steven Buchanan
• Inside Sales Coordinator	Laura Callahan	Andrew Fuller
• Sales Representative	Anne Dodsworth	Steven Buchanan

many other aspects related to SQL

- Transactions, Indexes, Constraints, Views
- Metadata
- Window function
- pre defined function (character strings, numbers, calendar data)
  - standard SQL or implemented in particular DB engine like MySQL or SQL Server

- as always it is not possible to close whiteout **thank you for your kindly attention!**
- *If you get half as much pleasure - the guilty variety, to be sure - from reading this slides as I get from writing it, we're all doing pretty well.*