BI Notes

James Solomon-Rounce

2018-03-12

Contents

Preface

The following notes were taken by me for educational, non-commercial, purposes. If you find the information useful, buy the material/take the course.

Chapter 1

Querying Data with Transact-SQL

Notes taken during/inspired by the edX course 'Querying Data with Transact-SQL - Microsoft: DAT201x' by Graeme Malcolm and Geoff Allix.

Course Handouts

- Course Syllabus
- Getting Started Guide Including Install
- Adventure Works Entity Relationship Diagram
- Adeventure Works db install script
  NOTE: Remember to ensure read only access for everyone to the folder containing the .SQL and other files
- GitHib Repo for course including course materials, slids, labs etc
- A copy of the above materials should they be changed or removed

Other useful links * Transact-SQL Refrence

```
library(DBI)
```

```
## Loading required package: methods
```

```
# creates a connection to the SQL database
# note that "con" will be used later in each connection to the database
con <- DBI::dbConnect(odbc::odbc(),
                      Driver = "SQL Server",
                      Server = "localhost\\SQLEXPRESS",
                      Database = "AdventureWorksLT",
                      Trusted_Connection = "True")

# Sets knitr to use this connection as the default so we don't need to specify it for every chunk
knitr::opts_chunk$set(connection = "con")
```

1.1 Introduction to Transact-SQL

SQL or Structured Query Language was first developed in the 1970s by IBM as a way of interacting with databases. Other vendors have specfic versions of SQL for instance Oracle is PL/SQL, Microsoft's implentation is TSQL or Transact SQL. Both SQL Server (on prem) and Azure SQL Databases (cloud) use the same query language, however Azure is a subset of full TSQL since it some commands relate to local files and data functions within .NET that relate only to SQL Server. However, as new features are added to Azure, some new commands are being added to Azure.

SQL is a declarative language - you express what it is that you want, the results - rather than specifying the steps taken to acheive that - it is not prodecural like other programming languages, it is set theory based. It is possible to write proceedural elements or steps within TSQL, however if this is ocurring a lot, it is perhaps better done in another language, which may also perform or run better.

In databases, we typically talk about entities - one type of thing - which is contained in each table.
* Entities are represented as relations (tables) * And entity attributes as domains (columns)

Most relationships are normalized, with relationships between primary and foreign keys. This helps to reduce duplication, however there are instances where de-normalised data is desired.

Schemas are namespaces for database objects - is shows a logical layout for all or part of a relational database, *"As part of a data dictionary, a database schema indicates how the entities that make up the database relate to one another, including tables, views, stored procedures, and more."*(see Lucid Chart on Database Schemas). The process of creating a database schema is called data modelling.

When referring to objects in a database, we could use a fully qualified name, such as:

- [server_name.][database_name.][schema_name.]object_name

This is only really relevant for SQL Server, since Azure will only work with one database at a time. Most of the time we typically just use

- schema_name.object_name

The schema name somwtimes be discarded, but it is considered best practice to include this, since there is sometimes some ambiguity about tables e.g. if we have two tables - Product.Order and Customer.Order - which order table is being referred to, that in the customer or product schema?

SQL has a number of SQL Statement Types:

- DML or Data Manipulation Language - SELECT, INSERT, UPDATE, DELETE
- DDL or Data Definition Language - CREATE, ALTER, DROP
- DCL or Data Control Language - GRANT, REVOKE, DENY

The course focuses on DML which is typically for working with data.

SELECT statement has a number of possible sub-components:

- FROM [table]
- WHERE [condition for filtering rows]
- GROUP BY [arranges rows by groups]
- HAVING [condition for filtering groups]
- ORDER BY [sorts the output]

Whilst a SQL statement can look like English, it doesn't neccessarily run from top to bottom in terms of the sequence of elements that are run in a query. For instance, the FROM is the first thing that will be run, then the WHERE filter will be run, then we GROUP BY, then SELECT the columns we are interested in and finally ORDER the results. This can be important when running some queries, which will be explored later in the course. When we run a query, it is not an actual table in a database that is return but a set of rows or record set or subset.

1.1.1 Data Types

There are a number of different data types in T-SQL as shown below, which are grouped in to a number of different types.

Exact Numeric	Approximate Numeric	Character
tinyint	float	char
smallint	real	varchar
int		text
bigint		nchar
bit		nvarchar
decimal/numeric		ntext
numeric		
money		
smallmoney		

Figure 1.1: Transact-SQL Data Types

(#fig:Data Types)

This is more relevant when designing a database, however it is useful to know when querying what data type you have in a broad sense - numeric, data, string and so on - as the types will determine what type of combinations can be combined together in expressions e.g. you can concatenate strings or add numbers together, but you can't concatenate a string and a number together.

Sometimes it is neccessary to convert data from one type to another, there are two ways this could happen

- Implicit conversion - compatible data types are automatically converted
- Explicit conversion - requires an explicit function e.g. CAST / TRY_CAST, STR, PARSE ? TRY_PARSE, CONVERT / TRY_CONVERT

The TRY options will attempt a conversion and if it does not work, a NULL will be returned rather than an error in the non-TRY version.

1.1.2 Working with NULLs

There are recognised standards for treating NULL values - ANSI - which says that anythign involving a NULL should return a NULL. There are functions that help us handle NULL values:

- ISNULL(column/variable, value) - Returns *value* (which you can specify) if the column or variable is NULL
- NULLIF(column/variable, value) - Returns NULL if the column or variable is a value - we are almost recoding a non-null to a null
- COALESCE (column/variable1, column/variable2, …) - Returns the value of the first non-NULL column or variable in the list - for instance if contact details, someone might not have an email, so we might want a telephone number, if they don't have that, return an address etc

NULL is used to indicate an unknown or missing value. NULL is **not** equivalent to zero or an empty string.

ISNULL can be used like an IF function in excel, for instance:

> SELECT name, ISNULL(TRY_CAST(size AS Integer), 0) AS NumericSize
> FROM SalesLT.Product;

In this instance, if there is a value that will be returned, if not, the NULL value will be returned as a 0.

We can also use a CASE statement to return a value whilst integrating NULL in to our query, e.g.

```
SELECT name,
    CASE size
      WHEN 'S' THEN 'SMALL'
      WHEN 'M' THEN 'MEDIUM'
      WHEN 'L' THEN 'LARGE'
      WHEN 'XL' THEN 'EXTRA LARGE'
      ELSE ISNULL(Size, 'N/A')
    END AS PRODUCT
  FROM SalesLT.Product;
```

1.1.3 Lab Exercises

AdventureWorks Cycles is a company that sells directly to retailers, who then sell products to consumers. Each retailer that is an AdventureWorks customer has provided a named contact for all communication from AdventureWorks.

The sales manager at AdventureWorks has asked you to generate some reports containing details of the company's customers to support a direct sales campaign. Let's start with some basic exploration.

First we display the sales person, the customer's title, surname and telephone number

Table 1.1: Displaying records 1 - 10

name	PRODUCT
HL Road Frame - Black, 58	58
HL Road Frame - Red, 58	58
Sport-100 Helmet, Red	N/A
Sport-100 Helmet, Black	N/A
Mountain Bike Socks, M	MEDIUM
Mountain Bike Socks, L	LARGE
Sport-100 Helmet, Blue	N/A
AWC Logo Cap	N/A
Long-Sleeve Logo Jersey, S	SMALL
Long-Sleeve Logo Jersey, M	MEDIUM

Table 1.2: Displaying records 1 - 10

SalesPerson	CustomerName	Phone
adventure-works\pamela0	Mr. Gee	245-555-0173
adventure-works\david8	Mr. Harris	170-555-0127
adventure-works\jillian0	Ms. Carreras	279-555-0130
adventure-works\jillian0	Ms. Gates	710-555-0173
adventure-works\shu0	Mr. Harrington	828-555-0186
adventure-works\linda3	Ms. Carroll	244-555-0112
adventure-works\shu0	Mr. Gash	192-555-0173
adventure-works\josé1	Ms. Garza	150-555-0127
adventure-works\josé1	Ms. Harding	926-555-0159
adventure-works\garrett1	Mr. Caprio	112-555-0191

```
SELECT SalesPerson, Title + ' ' + LastName AS CustomerName, Phone
FROM SalesLT.Customer;
```

Next we cast the CustomerID column to a VARCHAR and concatenate with the CompanyName column

```
SELECT CAST(CustomerID AS VARCHAR) + ': ' + CompanyName AS CustomerCompany
FROM SalesLT.Customer;
```

The SalesLT.SalesOrderHeader table contains records of sales orders. You have been asked to retrieve data for a report that shows:

The sales order number and revision number in the format () (e.g. SO71774 (2)). The order date converted to ANSI standard format yyyy.mm.dd (e.g. 2015.01.31).

```
SELECT SalesOrderNumber + ' (' + STR(RevisionNumber, 1) + ')' AS OrderRevision,
       CONVERT(NVARCHAR(30), OrderDate, 102) AS OrderDate
FROM SalesLT.SalesOrderHeader;
```

Next we write a query that returns a list of customer names. We use ISNULL to check for middle names and concatenate with FirstName and LastName.

```
SELECT FirstName + ' ' + ISNULL(MiddleName + ' ', '') + LastName
AS CustomerName
FROM SalesLT.Customer;
```

Next, we will imagine that some data has been deleted - customer email addresses - then we try to find

Table 1.3: Displaying records 1 - 10

CustomerCompany
1: A Bike Store
2: Progressive Sports
3: Advanced Bike Components
4: Modular Cycle Systems
5: Metropolitan Sports Supply
6: Aerobic Exercise Company
7: Associated Bikes
10: Rural Cycle Emporium
11: Sharp Bikes
12: Bikes and Motorbikes

Table 1.4: Displaying records 1 - 10

OrderRevision	OrderDate
SO71774 (2)	2008.06.01
SO71776 (2)	2008.06.01
SO71780 (2)	2008.06.01
SO71782 (2)	2008.06.01
SO71783 (2)	2008.06.01
SO71784 (2)	2008.06.01
SO71796 (2)	2008.06.01
SO71797 (2)	2008.06.01
SO71815 (2)	2008.06.01
SO71816 (2)	2008.06.01

Table 1.5: Displaying records 1 - 10

CustomerName
Orlando N. Gee
Keith Harris
Donna F. Carreras
Janet M. Gates
Lucy Harrington
Rosmarie J. Carroll
Dominic P. Gash
Kathleen M. Garza
Katherine Harding
Johnny A. Caprio

Table 1.6: Displaying records 1 - 10

CustomerID	PrimaryContact
1	245-555-0173
2	keith0@adventure-works.com
3	donna0@adventure-works.com
4	janet1@adventure-works.com
5	lucy0@adventure-works.com
6	rosmarie0@adventure-works.com
7	dominic0@adventure-works.com
10	kathleen0@adventure-works.com
11	katherine0@adventure-works.com
12	johnny0@adventure-works.com

contact details in sequence.

```
UPDATE SalesLT.Customer
SET EmailAddress = NULL
WHERE CustomerID % 7 = 1;
```

Next we write a query that returns a list of customer IDs in one column, and a second column named PrimaryContact that contains the email address if known, and otherwise the phone number.

```
SELECT CustomerID, COALESCE(EmailAddress, Phone) AS PrimaryContact
FROM SalesLT.Customer;
```

You have been asked to create a query that returns a list of sales order IDs and order dates with a column named ShippingStatus that contains the text "Shipped" for orders with a known ship date, and "Awaiting Shipment" for orders with no ship date.

Again, we imagine that some data is missing by deleting some first.

```
UPDATE SalesLT.SalesOrderHeader
SET ShipDate = NULL
WHERE SalesOrderID > 71899;
```

```
SELECT SalesOrderID, OrderDate,
  CASE
    WHEN ShipDate IS NULL THEN 'Awaiting Shipment'
    ELSE 'Shipped'
  END AS ShippingStatus
FROM SalesLT.SalesOrderHeader;
```

1.2 Querying Tables with SELECT

1.2.1 Removing Duplicates

If we wanted to know what colours our products are, we would run something like the following.

```
SELECT Color
FROM SalesLT.Product;
```

Here each product has a row and corresponding colour. However, if we just want colour, we are typically interested in removing duplicates to just show what colours we are actually producing. This is achieve using

Table 1.7: Displaying records 1 - 10

SalesOrderID	OrderDate	ShippingStatus
71774	2008-06-01	Shipped
71776	2008-06-01	Shipped
71780	2008-06-01	Shipped
71782	2008-06-01	Shipped
71783	2008-06-01	Shipped
71784	2008-06-01	Shipped
71796	2008-06-01	Shipped
71797	2008-06-01	Shipped
71815	2008-06-01	Shipped
71816	2008-06-01	Shipped

Table 1.8: Displaying records 1 - 10

Color
Black
Red
Red
Black
White
White
Blue
Multi
Multi
Multi

Table 1.9: Displaying records 1 - 10

Color
NA
Black
Blue
Grey
Multi
Red
Silver
Silver/Black
White
Yellow

Table 1.10: Displaying records 1 - 10

Color	Size
NA	NA
Black	NA
Black	38
Black	40
Black	42
Black	44
Black	46
Black	48
Black	52
Black	58

the DISTINCT keyword

```
SELECT DISTINCT Color
FROM SalesLT.Product;
```

These results are DISTINCT at the row level, so if we have two combinitions of columns - say size and colour - then it would be DISTINCT colour and size combinations that appear in the database.

```
SELECT DISTINCT Color, Size
FROM SalesLT.Product;
```

Here is just the size - here IS NULL will return a 'None' if the Size is missing (NA).

```
SELECT DISTINCT ISNULL(Size, 'None') AS Size
FROM SalesLT.Product;
```

1.2.2 Sorting Results

ORDER By is how we sort the results. Any aliased fields used in the SELECT element are visible by ORDER BY. You can oder the results using columns that are not selected in the SELECT clause. You can also ORDER BY multiple columns, either ascending or descending.

We can also just show the top 10 products, e.g. the top 10 most expensive. This is done using the keywork TOP

Table 1.11: Displaying records 1 - 10

Size
38
40
42
44
46
48
50
52
54
56

Table 1.12: Displaying records 1 - 10

Category	Name	ListPrice
6	Road-150 Red, 62	3578.27
6	Road-150 Red, 44	3578.27
6	Road-150 Red, 48	3578.27
6	Road-150 Red, 52	3578.27
6	Road-150 Red, 56	3578.27
5	Mountain-100 Silver, 38	3399.99
5	Mountain-100 Silver, 42	3399.99
5	Mountain-100 Silver, 44	3399.99
5	Mountain-100 Silver, 48	3399.99
5	Mountain-100 Black, 38	3374.99

```
SELECT TOP (10) ProductCategoryID AS Category, Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice DESC, Category;
```

We can also use TOP (N) Percent or TOP (N) WITH TIES. If we say wanted the bottom 10 items - say those with the lowest price - there is no 'BOTTOM' Keyword, instead we would sort our data so that they are now in the the order we want - with those we are interested at the top - the use TOP again.

1.2.3 Paging through results

This is achieved through using the OFFSET-FETCH which is an extension of ORDER BY. This might be useful if you have a set of web page results and you want to see certain ones.

You first say how many rows you want to skip using e.g. OFFSET 10 ROWS, then use specify how many rows you are interested in retrieving from the database e.g. FETCH NEXT 10 ROWS ONLY.

1.2.4 Filtering and Using Predicates

We can use the WHERE clause with a number of coniditions or predicates. For instance = (equals) <> (not equals), IN, BETWEEN (is an inclusive statement e.g. BETWEEN 100 AND 200 includes 100 and 200), LIKE, AND, OR and NOT. IN can be more efficient in coding terms when testing multiple attributes, as you just say color IN (red, blue) rather than colour = 'red' OR colour = 'blue'. This becomes more useful

Table 1.13: Displaying records 1 - 10

Name	Color	Size	ProductNumber
HL Road Frame - Black, 58	Black	58	FR-R92B-58
HL Road Frame - Red, 58	Red	58	FR-R92R-58
HL Road Frame - Red, 62	Red	62	FR-R92R-62
HL Road Frame - Red, 44	Red	44	FR-R92R-44
HL Road Frame - Red, 48	Red	48	FR-R92R-48
HL Road Frame - Red, 52	Red	52	FR-R92R-52
HL Road Frame - Red, 56	Red	56	FR-R92R-56
LL Road Frame - Black, 58	Black	58	FR-R38B-58
LL Road Frame - Black, 60	Black	60	FR-R38B-60
LL Road Frame - Black, 62	Black	62	FR-R38B-62

Table 1.14: Displaying records 1 - 10

Name	Color	Size	ProductNumber
HL Road Frame - Black, 58	Black	58	FR-R92B-58
HL Road Frame - Red, 58	Red	58	FR-R92R-58
LL Road Frame - Black, 58	Black	58	FR-R38B-58
LL Road Frame - Red, 58	Red	58	FR-R38R-58
ML Road Frame - Red, 58	Red	58	FR-R72R-58
Road-450 Red, 58	Red	58	BK-R68R-58
Road-650 Red, 58	Red	58	BK-R50R-58
Road-650 Black, 58	Black	58	BK-R50B-58
Road-250 Red, 58	Red	58	BK-R89R-58
Road-250 Black, 58	Black	58	BK-R89B-58

when testing multiple conditions e.g. IN (red, blue) AND size = large - this would have more typing with explicit code for each combition.

Like mathematics, SQL works on PEMDAS sequencing - parenthesis, exponents, multiplication, division, addition, subtraction.

Some examples.

First look for products that start with an FR:

```
SELECT Name, Color, Size, ProductNumber
FROM SalesLT.Product
WHERE ProductNumber LIKE 'FR%';
```

Or we can look for products that end in a 58:

```
SELECT Name, Color, Size, ProductNumber
FROM SalesLT.Product
WHERE ProductNumber LIKE '%58';
```

Or we can use underscores to specify a number of charecters e.g. one _ is one missing charecter. A wildcard (%) would match any number of chars. The figures in brackets then are like regex, so if we want a numeric value between 0 and 9 we use [0-9]. Equally we could use a similar query to find things like email addresses in a string, or email addresses that end in a .co.uk.

```
SELECT Name, Color, Size, ProductNumber
FROM SalesLT.Product
```

Table 1.15: Displaying records 1 - 10

Name	Color	Size	ProductNumber
Road-150 Red, 62	Red	62	BK-R93R-62
Road-150 Red, 44	Red	44	BK-R93R-44
Road-150 Red, 48	Red	48	BK-R93R-48
Road-150 Red, 52	Red	52	BK-R93R-52
Road-150 Red, 56	Red	56	BK-R93R-56
Road-450 Red, 58	Red	58	BK-R68R-58
Road-450 Red, 60	Red	60	BK-R68R-60
Road-450 Red, 44	Red	44	BK-R68R-44
Road-450 Red, 48	Red	48	BK-R68R-48
Road-450 Red, 52	Red	52	BK-R68R-52

Table 1.16: Displaying records 1 - 10

Name
Mountain Bike Socks, M
Mountain Bike Socks, L
ML Road Frame - Red, 44
ML Road Frame - Red, 48
ML Road Frame - Red, 52
ML Road Frame - Red, 58
ML Road Frame - Red, 60
HL Mountain Frame - Silver, 44
HL Mountain Frame - Silver, 48
HL Mountain Frame - Black, 44

```sql
WHERE ProductNumber LIKE 'BK-_[0-9][0-9]_-[0-9][0-9]';
```

We can use the BETWEEN clause on things like dates to select all products that were removed from sale in 2016:

```sql
SELECT Name
FROM SalesLT.Product
WHERE SellEndDate BETWEEN '2006/1/1' AND '2006/12/31';
```

Note that it is often useful to order results in the order you want, even if it currently appears it the correct order. Sometimes these queries may change as data or the database does, so it is best to be explicit and use an ORDER BY.

1.2.5 Lab Exercises

You are being told that transportation costs are increasing and you need to identify the heaviest products.

```sql
-- select the top 10 percent from the Name column
SELECT TOP (10) Percent Name, Weight
FROM SalesLT.Product
-- order by the weight in descending order
ORDER BY Weight DESC;
```

Next, we want to ignore the first 10 records - to page through using offset

Table 1.17: Displaying records 1 - 10

Name	Weight
Touring-3000 Blue, 62	13607.70
Touring-3000 Yellow, 62	13607.70
Touring-3000 Blue, 58	13562.34
Touring-3000 Yellow, 58	13512.45
Touring-3000 Blue, 54	13462.55
Touring-3000 Yellow, 54	13344.62
Touring-3000 Yellow, 50	13213.08
Touring-3000 Blue, 50	13213.08
Touring-3000 Blue, 44	13049.78
Touring-3000 Yellow, 44	13049.78

Table 1.18: Displaying records 1 - 10

Name
Mountain-500 Silver, 52
Mountain-500 Black, 52
Mountain-500 Black, 48
Mountain-500 Silver, 48
Mountain-500 Silver, 44
Mountain-500 Black, 44
Touring-2000 Blue, 60
Mountain-500 Black, 42
Mountain-500 Silver, 42
Touring-2000 Blue, 54

```sql
SELECT Name
FROM SalesLT.Product
ORDER BY Weight DESC
-- offset 10 rows and get the next 100
OFFSET 10 ROWS FETCH NEXT 100 ROWS ONLY;
```

Next we create a query to find the names, colors, and sizes of the products with a product model ID of 1.

```sql
-- select the Name, Color, and Size columns
SELECT Name, Color, Size
FROM SalesLT.Product
-- check ProductModelID is 1
WHERE ProductModelID = 1;
```

Now we would like more information on products of certain colors and sizes. We retrieve the product number and name of the products that have a Color of ‘Black’, ‘Red’, or ‘White’ and a Size of ‘S’ or ‘M’.

Table 1.19: 3 records

Name	Color	Size
Classic Vest, S	Blue	S
Classic Vest, M	Blue	M
Classic Vest, L	Blue	L

Table 1.20: Displaying records 1 - 10

ProductNumber	Name
SO-B909-M	Mountain Bike Socks, M
SH-M897-S	Men's Sports Shorts, S
SH-M897-M	Men's Sports Shorts, M
TG-W091-S	Women's Tights, S
TG-W091-M	Women's Tights, M
GL-H102-S	Half-Finger Gloves, S
GL-H102-M	Half-Finger Gloves, M
GL-F110-S	Full-Finger Gloves, S
GL-F110-M	Full-Finger Gloves, M
SH-W890-S	Women's Mountain Shorts, S

Table 1.21: Displaying records 1 - 10

ProductNumber	Name	ListPrice
BK-R93R-62	Road-150 Red, 62	3578.27
BK-R93R-44	Road-150 Red, 44	3578.27
BK-R93R-48	Road-150 Red, 48	3578.27
BK-R93R-52	Road-150 Red, 52	3578.27
BK-R93R-56	Road-150 Red, 56	3578.27
BK-R68R-58	Road-450 Red, 58	1457.99
BK-R68R-60	Road-450 Red, 60	1457.99
BK-R68R-44	Road-450 Red, 44	1457.99
BK-R68R-48	Road-450 Red, 48	1457.99
BK-R68R-52	Road-450 Red, 52	1457.99

```sql
-- select the ProductNumber and Name columns
SELECT ProductNumber, Name
FROM SalesLT.Product
-- check that Color is one of 'Black', 'Red' or 'White'
-- check that Size is one of 'S' or 'M'
WHERE Color IN ('Black', 'Red', 'White') AND Size IN ('S', 'M');
```

Next you have been asked to retrieve the product number, name, and list price of products that have a product number beginning with 'BK-'.

```sql
-- select the ProductNumber, Name, and ListPrice columns
SELECT ProductNumber, Name, ListPrice
FROM SalesLT.Product
-- filter for product numbers beginning with BK- using LIKE
WHERE ProductNumber LIKE 'BK-%';
```

Finally, the product manager is interested in a slight variation of the last request regarding product numbers with a particular prefix.

We are interested in products with product number beginning with 'BK-' followed by any character other than 'R', and ending with a '-' followed by any two numerals. Not an R is [^R].

```sql
-- select the ProductNumber, Name, and ListPrice columns
SELECT ProductNumber, Name, ListPrice
FROM SalesLT.Product
```

Table 1.22: Displaying records 1 - 10

ProductNumber	Name	ListPrice
BK-M82S-38	Mountain-100 Silver, 38	3399.99
BK-M82S-42	Mountain-100 Silver, 42	3399.99
BK-M82S-44	Mountain-100 Silver, 44	3399.99
BK-M82S-48	Mountain-100 Silver, 48	3399.99
BK-M82B-38	Mountain-100 Black, 38	3374.99
BK-M82B-42	Mountain-100 Black, 42	3374.99
BK-M82B-44	Mountain-100 Black, 44	3374.99
BK-M82B-48	Mountain-100 Black, 48	3374.99
BK-M68S-38	Mountain-200 Silver, 38	2319.99
BK-M68S-42	Mountain-200 Silver, 42	2319.99

```
-- filter for ProductNumbers
WHERE ProductNumber LIKE 'BK-[^R]%-[0-9][0-9]';
```

1.3 Querying Tables with Joins

We usually join tables based on primary key - foreign key relationships. We don't run two queries then join, but match at the time of the query. When we are talking about joins, we typically represent them as Venn diagrams.

The convention (ANSI SQL-92) is to specify the JOIN operator in the FROM clause:

SELECT … FROM Table1 JOIN Table 2 ON ;

There is an older standard (ANSI SQL-89) where the tables are joined using commas in the FROM clause and using a WHERE operator, but this can lead to accidental cartesian (aka cross) products.

1.3.1 INNER Joins

INNER Joins are typically the most common join type. It involves a join only where a match is found in both input tables. You can add multiple joins after each other.

Some examples - first a basic inner join where the schema, table and column are explicitly stated.

```
SELECT SalesLT.Product.Name AS ProductName, SalesLT.ProductCategory.Name AS Category
FROM SalesLT.Product
INNER JOIN SalesLT.ProductCategory
ON SalesLT.Product.ProductCategoryID = SalesLT.ProductCategory.ProductCategoryID;
```

Next, we can do the same query but make this less cumbersome by using table aliases.

```
SELECT p.Name AS ProductName, c.Name AS Category
FROM SalesLT.Product AS p
INNER JOIN SalesLT.ProductCategory AS c
on p.ProductCategoryID = c.ProductCategoryID;
```

Next, we look at joining multiple tables, where we want the sales, including order level details, the products in the order and the product details. If we don't specify the join type, the assumption is it is a INNER join, as shown below.

Table 1.23: Displaying records 1 - 10

ProductName	Category
Mountain-100 Silver, 38	Mountain Bikes
Mountain-100 Silver, 42	Mountain Bikes
Mountain-100 Silver, 44	Mountain Bikes
Mountain-100 Silver, 48	Mountain Bikes
Mountain-100 Black, 38	Mountain Bikes
Mountain-100 Black, 42	Mountain Bikes
Mountain-100 Black, 44	Mountain Bikes
Mountain-100 Black, 48	Mountain Bikes
Mountain-200 Silver, 38	Mountain Bikes
Mountain-200 Silver, 42	Mountain Bikes

Table 1.24: Displaying records 1 - 10

ProductName	Category
Mountain-100 Silver, 38	Mountain Bikes
Mountain-100 Silver, 42	Mountain Bikes
Mountain-100 Silver, 44	Mountain Bikes
Mountain-100 Silver, 48	Mountain Bikes
Mountain-100 Black, 38	Mountain Bikes
Mountain-100 Black, 42	Mountain Bikes
Mountain-100 Black, 44	Mountain Bikes
Mountain-100 Black, 48	Mountain Bikes
Mountain-200 Silver, 38	Mountain Bikes
Mountain-200 Silver, 42	Mountain Bikes

Table 1.25: Displaying records 1 - 10

OrderDate	SalesOrderNumber	ProductName	OrderQty	UnitPrice	LineTotal
2008-06-01	SO71774	ML Road Frame-W - Yellow, 48	1	356.898	356.8980
2008-06-01	SO71774	ML Road Frame-W - Yellow, 38	1	356.898	356.8980
2008-06-01	SO71776	Rear Brakes	1	63.900	63.9000
2008-06-01	SO71780	ML Mountain Frame-W - Silver, 42	4	218.454	873.8160
2008-06-01	SO71780	Mountain-400-W Silver, 46	2	461.694	923.3880
2008-06-01	SO71780	Mountain-500 Silver, 52	6	112.998	406.7928
2008-06-01	SO71780	HL Mountain Frame - Silver, 38	2	818.700	1637.4000
2008-06-01	SO71780	Mountain-500 Black, 42	1	323.994	323.9940
2008-06-01	SO71780	LL Mountain Frame - Black, 48	1	149.874	149.8740
2008-06-01	SO71780	HL Mountain Frame - Black, 42	1	809.760	809.7600

Table 1.26: Displaying records 1 - 10

OrderDate	SalesOrderNumber	ProductName	OrderQty	UnitPrice	LineTotal
2008-06-01	SO71774	ML Road Frame-W - Yellow, 48	1	356.898	356.8980
2008-06-01	SO71774	ML Road Frame-W - Yellow, 38	1	356.898	356.8980
2008-06-01	SO71776	Rear Brakes	1	63.900	63.9000
2008-06-01	SO71780	ML Mountain Frame-W - Silver, 42	4	218.454	873.8160
2008-06-01	SO71780	Mountain-400-W Silver, 46	2	461.694	923.3880
2008-06-01	SO71780	Mountain-500 Silver, 52	6	112.998	406.7928
2008-06-01	SO71780	HL Mountain Frame - Silver, 38	2	818.700	1637.4000
2008-06-01	SO71780	Mountain-500 Black, 42	1	323.994	323.9940
2008-06-01	SO71780	LL Mountain Frame - Black, 48	1	149.874	149.8740
2008-06-01	SO71780	HL Mountain Frame - Black, 42	1	809.760	809.7600

```sql
SELECT oh.OrderDate, oh.SalesOrderNumber, p.Name As ProductName, od.OrderQty, od.UnitPrice, od.LineTotal
FROM SalesLT.SalesOrderHeader AS oh
JOIN SalesLT.SalesOrderDetail AS od
ON od.SalesOrderID = oh.SalesOrderID
JOIN SalesLT.Product AS p
ON od.ProductID = p.ProductID
ORDER BY oh.OrderDate, oh.SalesOrderID, od.SalesOrderDetailID;
```

It is possible to do joins based on more than one criteria, e.g. we could join based on the productID and where the ListPrice is less than the unitprice i.e. there has been a discount.

```sql
-- Multiple join predicates
SELECT oh.OrderDate, oh.SalesOrderNumber, p.Name As ProductName, od.OrderQty, od.UnitPrice, od.LineTotal
FROM SalesLT.SalesOrderHeader AS oh
JOIN SalesLT.SalesOrderDetail AS od
ON od.SalesOrderID = oh.SalesOrderID
JOIN SalesLT.Product AS p
ON od.ProductID = p.ProductID AND od.UnitPrice < p.ListPrice --Note multiple predicates
ORDER BY oh.OrderDate, oh.SalesOrderID, od.SalesOrderDetailID;
```

Table 1.27: Displaying records 1 - 10

FirstName	LastName	SalesOrderNumber
Orlando	Gee	NA
Keith	Harris	NA
Donna	Carreras	NA
Janet	Gates	NA
Lucy	Harrington	NA
Rosmarie	Carroll	NA
Dominic	Gash	NA
Kathleen	Garza	NA
Katherine	Harding	NA
Johnny	Caprio	NA

1.3.2 OUTER Joins

In an outer join we return all the rows from one table, and any matching rows from the second table. The records in the 'outer' table are preserved, typically we use language such as LEFT, RIGHT and FULL keywords. We are pulling in records from that OUTER table. FULL keeps records from both tables, but are typically not seen in practice. OUTER is OPTIONAL e.g. LEFT JOIN is the same as LEFT OUTER JOIN.

Now some examples. First, we bring up a list of customers, with any matching sales records i.e. we have a list of customers who HAVE bought someting and those WHO HAVE NOT.

```
--Get all customers, with sales orders for those who've bought anything
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
ON c.CustomerID = oh.CustomerID
ORDER BY c.CustomerID;
```

Next we can look just for those customers who have not purchased anything using IS NULL.

```
--Return only customers who haven't purchased anything
SELECT c.FirstName, c.LastName, oh.SalesOrderNumber
FROM SalesLT.Customer AS c
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
ON c.CustomerID = oh.CustomerID
WHERE oh.SalesOrderNumber IS NULL
ORDER BY c.CustomerID;
```

Next we add records from multiple tables. If we add tables on to the chain of tables, having first decleared a left or right join, you have to keep using LEFT joins. You could use an INNER Join but you would loose some records e.g. if you used an INNER join on the second table below, you would loose those records (products) that had never been sold.

Sometimes it is neccessary to go through one table to get to another e.g. to Products -> Orders requires going through order details first. Even if, we are not brining back any tables from the intermediary table.

```
--More than 2 tables
SELECT p.Name As ProductName, oh.SalesOrderNumber
FROM SalesLT.Product AS p
LEFT JOIN SalesLT.SalesOrderDetail AS od
ON p.ProductID = od.ProductID
LEFT JOIN SalesLT.SalesOrderHeader AS oh --Additional tables added to the right must also use a left jo
```

Table 1.28: Displaying records 1 - 10

FirstName	LastName	SalesOrderNumber
Orlando	Gee	NA
Keith	Harris	NA
Donna	Carreras	NA
Janet	Gates	NA
Lucy	Harrington	NA
Rosmarie	Carroll	NA
Dominic	Gash	NA
Kathleen	Garza	NA
Katherine	Harding	NA
Johnny	Caprio	NA

Table 1.29: Displaying records 1 - 10

ProductName	SalesOrderNumber
HL Road Frame - Black, 58	NA
HL Road Frame - Red, 58	NA
Sport-100 Helmet, Red	SO71782
Sport-100 Helmet, Red	SO71783
Sport-100 Helmet, Red	SO71784
Sport-100 Helmet, Red	SO71797
Sport-100 Helmet, Red	SO71902
Sport-100 Helmet, Red	SO71936
Sport-100 Helmet, Red	SO71938
Sport-100 Helmet, Black	SO71782

```
ON od.SalesOrderID = oh.SalesOrderID
ORDER BY p.ProductID;
```

Next another example with multiple tables, but this time the order of the tables is different. We SELECT from the product table, then we LEFT Join the Order details so we can identify products that have never sold, then we LEFT JOIN to the order header table so we can get the order number where we use a left outer join as we are left joining those records to our original table. Then we INNER Join product category, because we are joining product category back to the first table - Products. The final table joins back to our original table, before the outer joins. So whether you need to use an OUTER or INNER join depends on where you wish to place the records based on the current list of tables, however you could reorder the tables to do it differently e.g. do the INNER JOIN first, then then LEFT JOINS.

```
SELECT p.Name As ProductName, c.Name AS Category, oh.SalesOrderNumber
FROM SalesLT.Product AS p
LEFT OUTER JOIN SalesLT.SalesOrderDetail AS od
ON p.ProductID = od.ProductID
LEFT OUTER JOIN SalesLT.SalesOrderHeader AS oh
ON od.SalesOrderID = oh.SalesOrderID
INNER JOIN SalesLT.ProductCategory AS c --Added to the left, so can use inner join
ON p.ProductCategoryID = c.ProductCategoryID
ORDER BY p.ProductID;
```

** Key Points**

- Use a Left Outer Join to include all rows from the first table and values from matched rows in the

Table 1.30: Displaying records 1 - 10

ProductName	Category	SalesOrderNumber
HL Road Frame - Black, 58	Road Frames	NA
HL Road Frame - Red, 58	Road Frames	NA
Sport-100 Helmet, Red	Helmets	SO71782
Sport-100 Helmet, Red	Helmets	SO71783
Sport-100 Helmet, Red	Helmets	SO71784
Sport-100 Helmet, Red	Helmets	SO71797
Sport-100 Helmet, Red	Helmets	SO71902
Sport-100 Helmet, Red	Helmets	SO71936
Sport-100 Helmet, Red	Helmets	SO71938
Sport-100 Helmet, Black	Helmets	SO71782

Table 1.31: Displaying records 1 - 10

Name	FirstName	LastName	Phone
All-Purpose Bike Stand	Orlando	Gee	245-555-0173
All-Purpose Bike Stand	Keith	Harris	170-555-0127
All-Purpose Bike Stand	Donna	Carreras	279-555-0130
All-Purpose Bike Stand	Janet	Gates	710-555-0173
All-Purpose Bike Stand	Lucy	Harrington	828-555-0186
All-Purpose Bike Stand	Rosmarie	Carroll	244-555-0112
All-Purpose Bike Stand	Dominic	Gash	192-555-0173
All-Purpose Bike Stand	Kathleen	Garza	150-555-0127
All-Purpose Bike Stand	Katherine	Harding	926-555-0159
All-Purpose Bike Stand	Johnny	Caprio	112-555-0191

second table. Columns in the second table for which no matching rows exist are populated with NULLs.
- Use a Right Outer Join to include all rows from the second table and values from matched rows in the first table. Columns in the first table for which no matching rows exist are populated with NULLs.
- Use a Full Outer Join to include all rows from the first and second tables. Columns in the either table for which no matching rows exist are populated with NULLs.

1.3.3 Cross Joins

Cross Joins create caretesian products - they combine each row from the first table with each row from the second table - to give all possible combinations of products.

One example of it being used would be if we have a list of staff with their attributes, we might want to compare them to the attributes required for all jobs to see which they match, perhaps as an internal job search. Another example is if we have two list of addresses and we calculate the edit distance between the two addresses from the different sources, to try and match those addresses. We would rank the addresses based on their edit distance, with those scoring highest the closest and most likely to be the same address. It can be used to generate test data also.

An example

```
--Call each customer once per product - perhaps not the most realistic example!
SELECT p.Name, c.FirstName, c.LastName, c.Phone
FROM SalesLT.Product as p
CROSS JOIN SalesLT.Customer as c;
```

Table 1.32: 9 records

EmployeeID	EmployeeName	ManagerID
1	adventure-works\david8	8
2	adventure-works\garrett1	1
3	adventure-works\jae0	NA
4	adventure-works\jillian0	3
5	adventure-works\josé1	1
6	adventure-works\linda3	3
7	adventure-works\michael9	9
8	adventure-works\pamela0	3
9	adventure-works\shu0	3

Table 1.33: 9 records

EmployeeName	ManagerName
adventure-works\jae0	NA
adventure-works\garrett1	adventure-works\david8
adventure-works\josé1	adventure-works\david8
adventure-works\linda3	adventure-works\jae0
adventure-works\pamela0	adventure-works\jae0
adventure-works\shu0	adventure-works\jae0
adventure-works\jillian0	adventure-works\jae0
adventure-works\david8	adventure-works\pamela0
adventure-works\michael9	adventure-works\shu0

1.3.4 Self Joins

You might want to join data on to itself but in a different sequence. For instance, we might want to join a person's manager on to their employee(s), but managers are also employees, so this would be a self join. We would use aliases for the table names as we have the same table twice. So when defining a self-join, you must specify an alias for at least one instance of the table being joined.

So our original table looks like this:

```
SELECT *
FROM SalesLT.Employee;
```

Then the actual self join query - we use a left join as some people i.e. the CEO, will not have a manager.

```
SELECT e.EmployeeName, m.EmployeeName AS ManagerName
FROM SalesLT.Employee AS e
LEFT JOIN SalesLT.Employee AS m
ON e.ManagerID = m.EmployeeID
ORDER BY e.ManagerID;
```

1.3.5 Lab Exercises

Write a query that returns the company name from the Sale.Customer table, the sales order ID and total due from the SalesLT.SalesOrderHeader table.

```
-- select the CompanyName, SalesOrderId, and TotalDue columns from the appropriate tables
SELECT c.CompanyName, oh.SalesOrderId, oh.TotalDue
```

Table 1.34: Displaying records 1 - 10

CompanyName	SalesOrderId	TotalDue
Professional Sales and Service	71782	43962.7901
Remarkable Bike Store	71935	7330.8972
Bulk Discount Store	71938	98138.2131
Coalition Bike Company	71899	2669.3183
Futuristic Bikes	71895	272.6468
Channel Outlet	71885	608.1766
Aerobic Exercise Company	71915	2361.6403
Vigorous Sports Store	71867	1170.5376
Thrilling Bike Tours	71858	15275.1977
Extreme Riding Supplies	71796	63686.2708

Table 1.35: Displaying records 1 - 10

CompanyName	AddressLine1	AddressLine2	City	StateProvince	PostalCod
Good Toys	99700 Bell Road		Auburn	California	95603
West Side Mart	251 The Metro Center		Wokingham	England	RG41 1QV
Nearby Cycle Shop	Burgess Hill	Edward Way	West Sussex	England	RH15 9UD
Professional Sales and Service	57251 Serene Blvd		Van Nuys	California	91411
Eastside Department Store	9992 Whipple Rd		Union City	California	94587
Action Bicycle Specialists	Warrington Ldc Unit 25/2		Woolston	England	WA1 4SY
Extreme Riding Supplies	Riverside		Sherman Oaks	California	91403
Riding Cycles	Galashiels		Liverpool	England	L4 4HB
Thrifty Parts and Sales	Oxnard Outlet		Oxnard	California	93030
Engineered Bike Systems	123 Camelia Avenue		Oxnard	California	93030

```sql
FROM SalesLT.Customer AS c
JOIN SalesLT.SalesOrderHeader AS oh
-- join tables based on CustomerID
ON c.CustomerID = oh.CustomerID;
```

In order to send out invoices to the customers, we need their addresses. Extend your customer orders query to include the main office address for each customer, including the full street address, city, state or province, postal code, and country or region.

```sql
SELECT c.CompanyName, a.AddressLine1, ISNULL(a.AddressLine2, '') AS AddressLine2, a.City, a.StateProvin
FROM SalesLT.Customer AS c
-- join the SalesOrderHeader table
JOIN SalesLT.SalesOrderHeader AS oh
ON oh.CustomerID = c.CustomerID
-- join the CustomerAddress table
JOIN SalesLT.CustomerAddress AS ca
-- filter for where the AddressType is 'Main Office'
ON c.CustomerID = ca.CustomerID AND AddressType = 'Main Office'
JOIN SalesLT.Address AS a
ON ca.AddressID = a.AddressID;
```

The sales manager wants a list of all customer companies and their contacts (first name and last name), showing the sales order ID and total due for each order they have placed. Customers who have not placed any orders should be included at the bottom of the list with NULL values for the order ID and total due.

Table 1.36: Displaying records 1 - 10

CompanyName	FirstName	LastName	SalesOrderID	TotalDue
Central Bicycle Specialists	Janeth	Esteves	71946	43.0437
Bulk Discount Store	Christopher	Beck	71938	98138.2131
Metropolitan Bicycle Supply	Krishna	Sunkammurali	71936	108597.9536
Remarkable Bike Store	Cory	Booth	71935	7330.8972
The Bicycle Accessories Company	Guy	Gilbert	71923	117.7276
Discount Tours	Melissa	Marple	71920	3293.7761
Essential Bike Works	Linda	Mitchell	71917	45.1995
Aerobic Exercise Company	Rosmarie	Carroll	71915	2361.6403
Many Bikes Store	Jeffrey	Kurtz	71902	81834.9826
Coalition Bike Company	Donald	Blanton	71899	2669.3183

Table 1.37: Displaying records 1 - 10

CompanyName	FirstName	LastName	Phone
A Bike Store	Orlando	Gee	245-555-0173
Progressive Sports	Keith	Harris	170-555-0127
Advanced Bike Components	Donna	Carreras	279-555-0130
Modular Cycle Systems	Janet	Gates	710-555-0173
Metropolitan Sports Supply	Lucy	Harrington	828-555-0186
Aerobic Exercise Company	Rosmarie	Carroll	244-555-0112
Associated Bikes	Dominic	Gash	192-555-0173
Rural Cycle Emporium	Kathleen	Garza	150-555-0127
Sharp Bikes	Katherine	Harding	926-555-0159
Bikes and Motorbikes	Johnny	Caprio	112-555-0191

```sql
-- select the CompanyName, FirstName, LastName, SalesOrderID and TotalDue columns
-- from the appropriate tables
SELECT c.CompanyName, c.FirstName, c.LastName, oh.SalesOrderID, oh.TotalDue
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.SalesOrderHeader AS oh
-- join based on CustomerID
ON oh.CustomerID  = c.CustomerID
-- order the SalesOrderIDs from highest to lowest
ORDER BY oh.SalesOrderID DESC;
```

A sales employee has noticed that AdventureWorks does not have address information for all customers. Write a query that returns a list of customer IDs, company names, contact names (first name and last name), and phone numbers for customers with no address stored in the database.

```sql
SELECT c.CompanyName, c.FirstName, c.LastName, c.Phone
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.CustomerAddress AS ca
-- join based on CustomerID
ON c.CustomerID = ca.CustomerID
-- filter for when the AddressID doesn't exist
WHERE ca.AddressID IS NULL;
```

Some customers have never placed orders, and some products have never been ordered.

Write a query that returns a column of customer IDs for customers who have never placed an order, and a

Table 1.38: Displaying records 1 - 10

CustomerID	ProductID
1	NA
2	NA
3	NA
4	NA
5	NA
6	NA
7	NA
10	NA
11	NA
12	NA

column of product IDs for products that have never been ordered.

Each row with a customer ID should have a NULL product ID (because the customer has never ordered a product) and each row with a product ID should have a NULL customer ID (because the product has never been ordered by a customer).

```sql
SELECT c.CustomerID, p.ProductID
FROM SalesLT.Customer AS c
FULL JOIN SalesLT.SalesOrderHeader AS oh
ON c.CustomerID = oh.CustomerID
FULL JOIN SalesLT.SalesOrderDetail AS od
-- join based on the SalesOrderID
ON od.SalesOrderID = oh.SalesOrderID
FULL JOIN SalesLT.Product AS p
-- join based on the ProductID
ON p.ProductID = od.ProductID
-- filter for nonexistent SalesOrderIDs
WHERE oh.SalesOrderID IS NULL
ORDER BY ProductID, CustomerID;
```

1.4 Using Set Operators

A union query is unlike a join, where as a join adds more columns, a union typically adds more rows. You put all records from one query on to the records at the end of another query. NOTE that it is a list of distinct (non duplicate) records - this will be checked every single row in one table and then checks if that record exists across every single row in the other table. Obviously as more tables are added, checking this can become more time consuming and affects performance.

UNION ALL will not undertake this checking, leading to more performance, but some duplicates. John Smith may appear in the employees table but he may also appear in the customers table. Sometimes this is the desired result. In such an instance it makes sense to add a new field using an alias at the time of ther query, such as record type, so we know where this record occurs.

When using Union:

- It is a good idea to use column aliases, so we know which table the column occurs in. However, only aliases in the first query are recognised, so any aliases should be set against the first table
- The number of columns must be the same across tables, you can add an additional column but this must be given a specific value or a NULL

Table 1.39: Displaying records 1 - 10

FirstName	LastName
Catherine	Abel
Kim	Abercrombie
Frances	Adams
Jay	Adams
Samuel	Agcaoili
Robert	Ahlering
Stanley	Alan
Amy	Alberts
Paul	Alcorn
Gregory	Alderson

Table 1.40: Displaying records 1 - 10

FirstName	LastName
Catherine	Abel
Kim	Abercrombie
Frances	Adams
Jay	Adams
Samuel	Agcaoili
Robert	Ahlering
Stanley	Alan
Amy	Alberts
Paul	Alcorn
Gregory	Alderson

- The data types must be approximatley similar i.e. we can do an implicit or explict conversion so they match.

The following example joins the Employees to the customer using Union i.e. removing duplicates.

NOTE: In the following example, views were created (using code provided in the course), before running the queries on the database. Therefore to reproduce these results, it is neccessary to run that same code (Module 4, Union sql file, in the CourseFiles zip) before the following queries will run on the AdventureWorksLT db.

```sql
SELECT FirstName, LastName
FROM SalesLT.Employees
UNION
SELECT FirstName, LastName
FROM SalesLT.Customers
ORDER BY LastName;
```

This gives 440 rows, if however we use UNION ALL

```sql
SELECT FirstName, LastName
FROM SalesLT.Employees
UNION ALL
SELECT FirstName, LastName
FROM SalesLT.Customers
ORDER BY LastName;
```

This results in 441 rows, so there is 1 row which occurs in both - a Donna Carreras appears both in the

Table 1.41: 5 records

FirstName	LastName	Type
Andy	Carothers	Employee
Donna	Carreras	Customer
Donna	Carreras	Employee
Rosmarie	Carroll	Employee
Joseph	Castellucio	Employee

Table 1.42: 1 records

FirstName	LastName
Donna	Carreras

customer and emplyee table. This could be the same person or indeed just someone with the same name.

It might be useful to know which table the record occurs in, which is where we can introduce a Type column. Not that after the first alias AS Type, subsequent fields e.g. 'Customer' as shown below do not need to be explicitly aliased, however it can make it easier to understand if we do include the AS Type.

```
SELECT FirstName, LastName, 'Employee' AS Type
FROM SalesLT.Employees
UNION
SELECT FirstName, LastName, 'Customer' AS Type
FROM SalesLT.Customers
ORDER BY LastName
OFFSET 100 ROWS FETCH NEXT 5 ROWS ONLY;
```

Note that even though we have used UNION rather than UNION ALL, this would result in the same number of rows as UNION ALL (441), because the presence of the type column means this is no longer the same person - it is not a duplicate record but a unique one. However, it is worth using UNION ALL to limit the number of records that have to be checked for duplicates, so improving peformance.

By default, UNION eliminates duplicate rows. Specify the ALL option to include duplicates (or to avoid the overhead of checking for duplicates when you know in advance that there are none).

1.4.1 INTERSECT and EXCEPT Queries

In INTERSECT we look at only rows that appear in each set - we are looking for the duplicates. Whereas using a JOIN would append columns to the source table, in set theory, we are looking for rows, so in an INTERSECT we are looking for those rows that appear in both datasets.

IN EXCEPT we are looking for records that appear in one source but not the other. In such a scenario, the order of the tables matters.

If we now try and identify the person who appears in both our datasets, we can identify the person directly.

```
SELECT FirstName, LastName
FROM SalesLT.Customers
INTERSECT
SELECT FirstName, LastName
FROM SalesLT.Employees;
```

Table 1.43: Displaying records 1 - 10

FirstName	LastName
Abraham	Swearengin
Ajay	Manchepalli
Alan	Steiner
Alice	Steiner
Andrea	Thomsen
Ben	Miller
Billy	Trent
Brad	Sutton
Caroline	Vicknair
Cecelia	Marshall

Next we want to find the EXCEPT - so let's find customers who are not employees, so we don't try and sel products to employees. There are 104 customers, so we should get 103 rows, as we remove Donna Carreras.

```
SELECT FirstName, LastName
FROM SalesLT.Customers
EXCEPT
SELECT FirstName, LastName
FROM SalesLT.Employees;
```

1.4.2 Lab Exercises

Customers can have two kinds of address: a main office address and a shipping address. The accounts department wants to ensure that the main office address is always used for billing, and have asked you to write a query that clearly identifies the different types of address for each customer.

```
-- select the CompanyName, AddressLine1 columns
-- alias as per the instructions
SELECT CompanyName, AddressLine1, City, 'Billing' AS AddressType
FROM SalesLT.Customer AS c
JOIN SalesLT.CustomerAddress AS ca
-- join based on CustomerID
ON c.CustomerID = ca.CustomerID
-- join another table
JOIN SalesLT.Address AS a
-- join based on AddressID
ON ca.AddressID = a.AddressID
-- filter for where the correct AddressType
WHERE ca.AddressType = 'Main Office';
```

Adapt the query to retrieve the company name, first line of the street address, city, and a column named AddressType with the value 'Shipping' for customers where the address type in the SalesLT.CustomerAddress table is 'Shipping'.

```
SELECT c.CompanyName, a.AddressLine1, a.City, 'Shipping' AS AddressType
FROM SalesLT.Customer AS c
JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
JOIN SalesLT.Address AS a
ON ca.AddressID = a.AddressID
```

Table 1.44: Displaying records 1 - 10

CompanyName	AddressLine1	City	AddressType
Professional Sales and Service	57251 Serene Blvd	Van Nuys	Billing
Riders Company	Tanger Factory	Branch	Billing
Area Bike Accessories	6900 Sisk Road	Modesto	Billing
Bicycle Accessories and Kits	Lewiston Mall	Lewiston	Billing
Valley Bicycle Specialists	Blue Ridge Mall	Kansas City	Billing
Vinyl and Plastic Goods Corporation	No. 25800-130 King Street West	Toronto	Billing
Fun Toys and Bikes	6500 East Grant Road	Tucson	Billing
Great Bikes	Eastridge Mall	Casper	Billing
Valley Toy Store	252851 Rowan Place	Richmond	Billing
Major Sport Suppliers	White Mountain Mall	Rock Springs	Billing

Table 1.45: Displaying records 1 - 10

CompanyName	AddressLine1	City	AddressType
Family's Favorite Bike Shop	26910 Indela Road	Montreal	Shipping
Center Cycle Shop	1318 Lasalle Street	Bothell	Shipping
Safe Cycles Shop	2681 Eagle Peak	Bellevue	Shipping
Modular Cycle Systems	165 North Main	Austin	Shipping
Progressive Sports	7943 Walnut Ave	Renton	Shipping
Hardware Components	99 Front Street	Minneapolis	Shipping
Sample Bike Store	2000 300th Street	Denver	Shipping
Racing Toys	9228 Via Del Sol	Phoenix	Shipping
Elite Bikes	9178 Jumping St.	Dallas	Shipping
All Cycle Shop	8713 Yosemite Ct.	Bothell	Shipping

```
WHERE ca.AddressType = 'Shipping';
```

Next we can union all these records together to create a list of shipping and billing addresses, using UNION ALL.

```
SELECT c.CompanyName, a.AddressLine1, a.City, 'Billing' AS AddressType
FROM SalesLT.Customer AS c
JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
JOIN SalesLT.Address AS a
ON ca.AddressID = a.AddressID
WHERE ca.AddressType = 'Main Office'
-- UNION
UNION ALL
SELECT c.CompanyName, a.AddressLine1, a.City, 'Shipping' AS AddressType
FROM SalesLT.Customer AS c
JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
JOIN SalesLT.Address AS a
ON ca.AddressID = a.AddressID
WHERE ca.AddressType = 'Shipping'
ORDER BY c.CompanyName
OFFSET 10 ROWS;
```

Table 1.46: Displaying records 1 - 10

CompanyName	AddressLine1	City	AddressType
All Cycle Shop	8713 Yosemite Ct.	Bothell	Shipping
All Cycle Shop	25111 228th St Sw	Bothell	Billing
All Seasons Sports Supply	Ohms Road	Houston	Billing
Alpine Ski House	7505 Laguna Boulevard	Elk Grove	Billing
Alternative Vehicles	3307 Evergreen Blvd	Washougal	Billing
Another Bicycle Company	567 Sw Mcloughlin Blvd	Milwaukie	Billing
Area Bike Accessories	6900 Sisk Road	Modesto	Billing
Area Sheet Metal Supply	399 Clearing Green	London	Billing
Associated Bikes	5420 West 22500 South	Salt Lake City	Billing
Authentic Sales and Service	99 Dean Street, Soho	London	Billing

Table 1.47: Displaying records 1 - 10

CompanyName
A Bike Store
A Great Bicycle Company
A Typical Bike Shop
Acceptable Sales & Service
Action Bicycle Specialists
Active Life Toys
Active Systems
Advanced Bike Components
Aerobic Exercise Company
Affordable Sports Equipment

You have created a master list of all customer addresses, but now you have been asked to create filtered lists that show which customers have only a main office address, and which customers have both a main office and a shipping address.

```
SELECT c.CompanyName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
INNER JOIN SalesLT.Address AS a
ON a.AddressID = ca.AddressID
WHERE ca.AddressType = 'Main Office' --Filters out shipping addresses.  This is a table of all Billing
EXCEPT
SELECT c.CompanyName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
INNER JOIN SalesLT.Address AS a
ON a.AddressID = ca.AddressID
WHERE ca.AddressType = 'Shipping'
ORDER BY c.CompanyName;
```

Or the INTERSECT version to identify the company name of each company that appears in a table of customers with a 'Main Office' address, and also in a table of customers with a 'Shipping' address.

Table 1.48: Displaying records 1 - 10

CompanyName
All Cycle Shop
Center Cycle Shop
Elite Bikes
Family's Favorite Bike Shop
Hardware Components
Modular Cycle Systems
Progressive Sports
Racing Toys
Safe Cycles Shop
Sample Bike Store

```
SELECT c.CompanyName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
INNER JOIN SalesLT.Address AS a
ON a.AddressID = ca.AddressID
WHERE ca.AddressType = 'Main Office' --Filters out shipping addresses.  This is a table of all Billing
INTERSECT
SELECT c.CompanyName
FROM SalesLT.Customer AS c
INNER JOIN SalesLT.CustomerAddress AS ca
ON c.CustomerID = ca.CustomerID
INNER JOIN SalesLT.Address AS a
ON a.AddressID = ca.AddressID
WHERE ca.AddressType = 'Shipping'
ORDER BY c.CompanyName;
```

1.5 Using Functions and Aggregating Data

Now we are looking to not just bring back individual rows but perform calculations, such as aggregations, on those results.

The functions we will look at are shown below.

1.5.1 Scalar Functions

A scalar function returns a single value, not a row or column or table. In database design things can be deterministic or non-deterministic

- Deterministic - we know what the result will be, assuming the data hasn't changed. The data going in and coming out will be the same
- Non-deterministic - we cannot gaurantee what the result will be. For instance if we tested if todays date was less than a value in a table, it would depend on the time and day that the query was run. We can't say what the result will be, based on the data going in/from the db.

A scalar function can be either deterministic or non-deterministic. Scalar is a set of functions, and can do multiple things - date and times, text and image, mathematical, system and system statistical, metadata

Function Category	Description
Scalar	Operate on a single row,
Logical	Scalar functions that com single output
Aggregate	Take one or more input v value
Window	Operate on a window (se
Rowset	Return a virtual table tha Transact-SQL statement

Figure 1.2: Transact-SQL Functions

Table 1.49: Displaying records 1 - 10

SellStartYear	ProductID	Name
2002	680	HL Road Frame - Black, 58
2002	706	HL Road Frame - Red, 58
2005	707	Sport-100 Helmet, Red
2005	708	Sport-100 Helmet, Black
2005	709	Mountain Bike Socks, M
2005	710	Mountain Bike Socks, L
2005	711	Sport-100 Helmet, Blue
2005	712	AWC Logo Cap
2005	713	Long-Sleeve Logo Jersey, S
2005	714	Long-Sleeve Logo Jersey, M

Table 1.50: Displaying records 1 - 10

SellStartYear	SellStartMonth	SellStartDay	SellStartWeekday	ProductID	Name
2002	June	1	Saturday	680	HL Road Frame - Black, 58
2002	June	1	Saturday	706	HL Road Frame - Red, 58
2005	July	1	Friday	707	Sport-100 Helmet, Red
2005	July	1	Friday	708	Sport-100 Helmet, Black
2005	July	1	Friday	709	Mountain Bike Socks, M
2005	July	1	Friday	710	Mountain Bike Socks, L
2005	July	1	Friday	711	Sport-100 Helmet, Blue
2005	July	1	Friday	712	AWC Logo Cap
2005	July	1	Friday	713	Long-Sleeve Logo Jersey, S
2005	July	1	Friday	714	Long-Sleeve Logo Jersey, M

and so on.

We might want to extract the year from a date, for instance to determine when a product was first sold.

```sql
SELECT YEAR(SellStartDate) SellStartYear, ProductID, Name
FROM SalesLT.Product
ORDER BY SellStartYear;
```

Other examples might include being able to extract certain parts of the date, like the month or day of the week, this can be achived with the DATENAME function

```sql
SELECT YEAR(SellStartDate) SellStartYear, DATENAME(mm,SellStartDate) SellStartMonth,
       DAY(SellStartDate) SellStartDay, DATENAME(dw, SellStartDate) SellStartWeekday,
       ProductID, Name
FROM SalesLT.Product
ORDER BY SellStartYear;
```

Or we might want to calculate how long a product has been sold, we can use the DATEDIFF function to work this out.

```sql
SELECT DATEDIFF(yy,SellStartDate, GETDATE()) YearsSold,
       DATEDIFF(mm,SellStartDate, GETDATE()) MonthsSold,
       ProductID, Name
FROM SalesLT.Product
ORDER BY ProductID;
```

Another common example might be to convert text to upper case.

Table 1.51: Displaying records 1 - 10

YearsSold	MonthsSold	ProductID	Name
16	189	680	HL Road Frame - Black, 58
16	189	706	HL Road Frame - Red, 58
13	152	707	Sport-100 Helmet, Red
13	152	708	Sport-100 Helmet, Black
13	152	709	Mountain Bike Socks, M
13	152	710	Mountain Bike Socks, L
13	152	711	Sport-100 Helmet, Blue
13	152	712	AWC Logo Cap
13	152	713	Long-Sleeve Logo Jersey, S
13	152	714	Long-Sleeve Logo Jersey, M

Table 1.52: Displaying records 1 - 10

ProductName
ALL-PURPOSE BIKE STAND
AWC LOGO CAP
BIKE WASH - DISSOLVER
CABLE LOCK
CHAIN
CLASSIC VEST, L
CLASSIC VEST, M
CLASSIC VEST, S
FENDER SET - MOUNTAIN
FRONT BRAKES

```
SELECT UPPER(Name) AS ProductName
FROM SalesLT.Product;
```

We might want to add two strings together but with a space or other text, so we could use the CONCAT function to achieve this.

```
SELECT CONCAT(FirstName + ' ', LastName) AS FullName,
       FirstName, LastName
FROM SalesLT.Customer;
```

Or we might want to return just a specific number of charecters from a string, perhaps there is a structure in the sequence - like the first two chars relate to a product type - so we want to just extract these first two.

```
SELECT Name, ProductNumber, LEFT(ProductNumber, 2) AS ProductType
FROM SalesLT.Product;
```

Or we might have something more complex, where we want to find certain elements of text, we can use CHARINDEX to identify where a char occurs, then combine it with SUBSTRING to extract just a portion of the full string which is n chars before or after a particular charecter.

```
SELECT Name, ProductNumber, LEFT(ProductNumber, 2) AS ProductType,
                            SUBSTRING(ProductNumber,CHARINDEX('-', ProductNumber) + 1, 4) AS ModelCode,
                                          SUBSTRING(ProductNumber, LEN(ProductNumber) - CHARINDEX('-', 
FROM SalesLT.Product;
```

Table 1.53: Displaying records 1 - 10

FullName	FirstName	LastName
Orlando Gee	Orlando	Gee
Keith Harris	Keith	Harris
Donna Carreras	Donna	Carreras
Janet Gates	Janet	Gates
Lucy Harrington	Lucy	Harrington
Rosmarie Carroll	Rosmarie	Carroll
Dominic Gash	Dominic	Gash
Kathleen Garza	Kathleen	Garza
Katherine Harding	Katherine	Harding
Johnny Caprio	Johnny	Caprio

Table 1.54: Displaying records 1 - 10

Name	ProductNumber	ProductType
HL Road Frame - Black, 58	FR-R92B-58	FR
HL Road Frame - Red, 58	FR-R92R-58	FR
Sport-100 Helmet, Red	HL-U509-R	HL
Sport-100 Helmet, Black	HL-U509	HL
Mountain Bike Socks, M	SO-B909-M	SO
Mountain Bike Socks, L	SO-B909-L	SO
Sport-100 Helmet, Blue	HL-U509-B	HL
AWC Logo Cap	CA-1098	CA
Long-Sleeve Logo Jersey, S	LJ-0192-S	LJ
Long-Sleeve Logo Jersey, M	LJ-0192-M	LJ

Table 1.55: Displaying records 1 - 10

Name	ProductNumber	ProductType	ModelCode	SizeCode
HL Road Frame - Black, 58	FR-R92B-58	FR	R92B	58
HL Road Frame - Red, 58	FR-R92R-58	FR	R92R	58
Sport-100 Helmet, Red	HL-U509-R	HL	U509	R
Sport-100 Helmet, Black	HL-U509	HL	U509	
Mountain Bike Socks, M	SO-B909-M	SO	B909	M
Mountain Bike Socks, L	SO-B909-L	SO	B909	L
Sport-100 Helmet, Blue	HL-U509-B	HL	U509	B
AWC Logo Cap	CA-1098	CA	1098	
Long-Sleeve Logo Jersey, S	LJ-0192-S	LJ	0192	S
Long-Sleeve Logo Jersey, M	LJ-0192-M	LJ	0192	M

Table 1.56: Displaying records 1 - 10

Name	Size
HL Road Frame - Black, 58	58
HL Road Frame - Red, 58	58
Sport-100 Helmet, Red	NA
Sport-100 Helmet, Black	NA
Mountain Bike Socks, M	M
Mountain Bike Socks, L	L
Sport-100 Helmet, Blue	NA
AWC Logo Cap	NA
Long-Sleeve Logo Jersey, S	S
Long-Sleeve Logo Jersey, M	M

Table 1.57: Displaying records 1 - 10

Name	NumericSize
HL Road Frame - Black, 58	58
HL Road Frame - Red, 58	58
HL Road Frame - Red, 62	62
HL Road Frame - Red, 44	44
HL Road Frame - Red, 48	48
HL Road Frame - Red, 52	52
HL Road Frame - Red, 56	56
LL Road Frame - Black, 58	58
LL Road Frame - Black, 60	60
LL Road Frame - Black, 62	62

1.5.2 Logical Functions

Logical functions test if someting is true or not i.e. traditional boolean. But we can use logical functions as filters by using CHOOSE for example, you could use IIF which is also a logical, or we could use CASE to achieve the same result.

An exmaple might be if we want to return some numeric sizes, for instance if we have a table of data with different size formats:

```sql
SELECT Name, Size
FROM SalesLT.Product;
```

We might only be interested in those that have a numeric type, even though the data is in a char data string, 1 in this instance means true:

```sql
SELECT Name, Size AS NumericSize
FROM SalesLT.Product
WHERE ISNUMERIC(Size) = 1;
```

Or we might want to assign a value to something, like a product type of bike to certain values and other to everything else.

```sql
SELECT Name, IIF(ProductCategoryID IN (5,6,7), 'Bike', 'Other') ProductType
FROM SalesLT.Product;
```

Or a more complicated query where we assign mutiple product types.

Table 1.58: Displaying records 1 - 10

Name	ProductType
HL Road Frame - Black, 58	Other
HL Road Frame - Red, 58	Other
Sport-100 Helmet, Red	Other
Sport-100 Helmet, Black	Other
Mountain Bike Socks, M	Other
Mountain Bike Socks, L	Other
Sport-100 Helmet, Blue	Other
AWC Logo Cap	Other
Long-Sleeve Logo Jersey, S	Other
Long-Sleeve Logo Jersey, M	Other

Table 1.59: Displaying records 1 - 10

ProductName	Category	ProductType
Mountain-100 Silver, 38	Mountain Bikes	Bikes
Mountain-100 Silver, 42	Mountain Bikes	Bikes
Mountain-100 Silver, 44	Mountain Bikes	Bikes
Mountain-100 Silver, 48	Mountain Bikes	Bikes
Mountain-100 Black, 38	Mountain Bikes	Bikes
Mountain-100 Black, 42	Mountain Bikes	Bikes
Mountain-100 Black, 44	Mountain Bikes	Bikes
Mountain-100 Black, 48	Mountain Bikes	Bikes
Mountain-200 Silver, 38	Mountain Bikes	Bikes
Mountain-200 Silver, 42	Mountain Bikes	Bikes

```sql
SELECT prd.Name AS ProductName, cat.Name AS Category,
      CHOOSE (cat.ParentProductCategoryID, 'Bikes','Components','Clothing','Accessories') AS ProductType
FROM SalesLT.Product AS prd
JOIN SalesLT.ProductCategory AS cat
ON prd.ProductCategoryID = cat.ProductCategoryID;
```

1.5.3 Window Functions

A window in this context is a set of rows - a window in to the database or table. For instance, we might want to RANK a set (or window) of data. The query below will pull out the top 100 more expensive products (ordered by list price) then creates a ranking based on this list price, then orders the results by this ranking.

```sql
SELECT TOP(100) ProductID, Name, ListPrice,
    RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice
FROM SalesLT.Product AS p
ORDER BY RankByPrice;
```

Or we might want to partition the results by product category, similar to grouping by product category. This will result in a ranking for each product category. The results might look a little odd at first glance.

```sql
SELECT c.Name AS Category, p.Name AS Product, ListPrice,
    RANK() OVER(PARTITION BY c.Name ORDER BY ListPrice DESC) AS RankByPrice
FROM SalesLT.Product AS p
```

Table 1.60: Displaying records 1 - 10

ProductID	Name	ListPrice	RankByPrice
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
752	Road-150 Red, 52	3578.27	1
753	Road-150 Red, 56	3578.27	1
771	Mountain-100 Silver, 38	3399.99	6
772	Mountain-100 Silver, 42	3399.99	6
773	Mountain-100 Silver, 44	3399.99	6
774	Mountain-100 Silver, 48	3399.99	6
775	Mountain-100 Black, 38	3374.99	10

Table 1.61: Displaying records 1 - 10

Category	Product	ListPrice	RankByPrice
Bib-Shorts	Men's Bib-Shorts, S	89.99	1
Bib-Shorts	Men's Bib-Shorts, M	89.99	1
Bib-Shorts	Men's Bib-Shorts, L	89.99	1
Bike Racks	Hitch Rack - 4-Bike	120.00	1
Bike Stands	All-Purpose Bike Stand	159.00	1
Bottles and Cages	Mountain Bottle Cage	9.99	1
Bottles and Cages	Road Bottle Cage	8.99	2
Bottles and Cages	Water Bottle - 30 oz.	4.99	3
Bottom Brackets	HL Bottom Bracket	121.49	1
Bottom Brackets	ML Bottom Bracket	101.24	2

```
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductcategoryID
ORDER BY Category, RankByPrice;
```

1.5.4 Aggregate Functions

There are a lot of aggregate functions including some statistical functions like s.d. as well as some more standard things like MIN, MAX, SUM etc over a set of data. We can use GROUP BY.

First we can get some headline stats of what is in the table.

```
SELECT COUNT(*) AS Products, COUNT(DISTINCT ProductCategoryID) AS Categories, AVG(ListPrice) AS Average
FROM SalesLT.Product;
```

Or we might be interested in summary figures for just one product type, like bikes. Note that we use COUNT then the specific item, since using COUNT(*) would return the number of rows, which may include duplicates.

Table 1.62: 1 records

Products	Categories	AveragePrice
295	37	744.5952

Table 1.63: 1 records

BikeModels	AveragePrice
97	1586.737

Table 1.64: 3 records

Salesperson	SalesRevenue
adventure-works\jae0	518096.4
adventure-works\linda3	209219.8
adventure-works\shu0	138116.9

```
SELECT COUNT(p.ProductID) BikeModels, AVG(p.ListPrice) AveragePrice
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductCategoryID
WHERE c.Name LIKE '%Bikes';
```

However, our aggregate functions are currently returnign totals. In practice, we may want to return figures by groups, so we need to use GROUP BY in our queries. When we use a GROUP BY which groups up the results, we have to group up everything in the select row that isn't being aggregated. You can't have something in the SELECT which is neither being aggregated nor being grouped. As we are grouping the results, we don't see individual rows of data anymore.

The following example calulates the sales revenue for each sales person.

```
SELECT c.Salesperson, SUM(oh.SubTotal) SalesRevenue
FROM SalesLT.Customer c
JOIN SalesLT.SalesOrderHeader oh
ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson
ORDER BY SalesRevenue DESC;
```

Could it be that some sales people have no sales? If so, we need to return values where the total is NULL, by assigning NULL a value of zero. WE also use the LEFT JOIN rather than the INNER JOIN previously, we get sales people included who haven't sold anything.

```
SELECT c.Salesperson, ISNULL(SUM(oh.SubTotal), 0.00) SalesRevenue
FROM SalesLT.Customer c
LEFT JOIN SalesLT.SalesOrderHeader oh
ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson
ORDER BY SalesRevenue DESC;
```

Or we might want to know the number of customers each sales person has.

```
SELECT Salesperson, COUNT(CustomerID) Customers
FROM SalesLT.Customer
GROUP BY Salesperson
ORDER BY Salesperson;
```

We might also want to know the sales Revenue by sales person and customer. Note that we now includer the customer details, we cannot use the CUSTOMER alias from the SELECT query in the GROUP BY, as the SELECT query is run after the GROUP BY.

Table 1.65: 9 records

Salesperson	SalesRevenue
adventure-works\jae0	518096.4
adventure-works\linda3	209219.8
adventure-works\shu0	138116.9
adventure-works\michael9	0.0
adventure-works\pamela0	0.0
adventure-works\jillian0	0.0
adventure-works\josé1	0.0
adventure-works\david8	0.0
adventure-works\garrett1	0.0

Table 1.66: 9 records

Salesperson	Customers
adventure-works\david8	73
adventure-works\garrett1	78
adventure-works\jae0	78
adventure-works\jillian0	148
adventure-works\josé1	142
adventure-works\linda3	71
adventure-works\michael9	32
adventure-works\pamela0	74
adventure-works\shu0	151

```sql
SELECT c.Salesperson, CONCAT(c.FirstName +' ', c.LastName) AS Customer, ISNULL(SUM(oh.SubTotal), 0.00) S
FROM SalesLT.Customer c
LEFT JOIN SalesLT.SalesOrderHeader oh
ON c.CustomerID = oh.CustomerID
GROUP BY c.Salesperson, CONCAT(c.FirstName +' ', c.LastName)
ORDER BY SalesRevenue DESC, Customer;
```

Or the number of products in each category.

```sql
SELECT c.Name AS Category, COUNT(p.ProductID) AS Products
FROM SalesLT.Product AS p
JOIN SalesLT.ProductCategory AS c
ON p.ProductCategoryID = c.ProductCategoryID
GROUP BY c.Name
ORDER BY Category;
```

1.5.5 Filtering Groups

If we want to filter, we can use the WHERE clause as we have seen in some previous examples. However, most of the time we filter with the HAVING caluse. **HAVING will filter the resutls RATHER than the input**.

Having clause provides a search condition that each group must satisfy, for instance where the number of orders by a customer is greater than 10.

In the example below, we are interested in finding out which sales people have more than 100 customers.

Table 1.67: Displaying records 1 - 10

Salesperson	Customer	SalesRevenue
adventure-works\jae0	Terry Eminhizer	108561.83
adventure-works\jae0	Krishna Sunkammurali	98278.69
adventure-works\jae0	Christopher Beck	88812.86
adventure-works\linda3	Kevin Liu	83858.43
adventure-works\jae0	Jon Grande	78029.69
adventure-works\shu0	Jeffrey Kurtz	74058.81
adventure-works\jae0	Rebecca Laszlo	63980.99
adventure-works\linda3	Anthony Chor	57634.63
adventure-works\shu0	Frank Campbell	41622.05
adventure-works\linda3	Catherine Abel	39785.33

Table 1.68: Displaying records 1 - 10

Category	Products
Bib-Shorts	3
Bike Racks	1
Bike Stands	1
Bottles and Cages	3
Bottom Brackets	3
Brakes	2
Caps	1
Chains	1
Cleaners	1
Cranksets	3

Table 1.69: 3 records

Salesperson	Customers
adventure-works\jillian0	148
adventure-works\josé1	142
adventure-works\shu0	151

Table 1.70: Displaying records 1 - 10

ProductID	ProductName	ApproxWeight
680	HL ROAD FRAME - BLACK, 58	1016
706	HL ROAD FRAME - RED, 58	1016
707	SPORT-100 HELMET, RED	NA
708	SPORT-100 HELMET, BLACK	NA
709	MOUNTAIN BIKE SOCKS, M	NA
710	MOUNTAIN BIKE SOCKS, L	NA
711	SPORT-100 HELMET, BLUE	NA
712	AWC LOGO CAP	NA
713	LONG-SLEEVE LOGO JERSEY, S	NA
714	LONG-SLEEVE LOGO JERSEY, M	NA

```sql
SELECT Salesperson, COUNT(CustomerID) Customers
FROM SalesLT.Customer
GROUP BY Salesperson
HAVING COUNT(CustomerID) > 100
ORDER BY Salesperson;
```

Key Points:

- You can use GROUP BY with aggregate functions to return aggregations grouped by one or more columns or expressions.
- All columns in the SELECT clause that are not aggregate function expressions must be included in a GROUP BY clause.
- The order in which columns or expressions are listed in the GROUP BY clause determines the grouping hierarchy.
- You can filter the groups that are included in the query results by specifying a HAVING clause.

1.5.6 Lab Exercises

Write a query to return the product ID of each product, together with the product name formatted as upper case and a column named ApproxWeight with the weight of each product rounded to the nearest whole unit.

```sql
SELECT ProductID, UPPER(Name) AS ProductName, ROUND(WEIGHT, 0) AS ApproxWeight
FROM SalesLT.Product;
```

Extend your query to include columns named SellStartYear and SellStartMonth containing the year and month in which AdventureWorks started selling each product. The month should be displayed as the month name (e.g. ‘January’).

```sql
SELECT ProductID, UPPER(Name) AS ProductName, ROUND(Weight, 0) AS ApproxWeight,
       YEAR(SellStartDate) as SellStartYear,
       DATENAME(m, SellStartDate) as SellStartMonth
FROM SalesLT.Product;
```

Table 1.71: Displaying records 1 - 10

ProductID	ProductName	ApproxWeight	SellStartYear	SellStartMonth
680	HL ROAD FRAME - BLACK, 58	1016	2002	June
706	HL ROAD FRAME - RED, 58	1016	2002	June
707	SPORT-100 HELMET, RED	NA	2005	July
708	SPORT-100 HELMET, BLACK	NA	2005	July
709	MOUNTAIN BIKE SOCKS, M	NA	2005	July
710	MOUNTAIN BIKE SOCKS, L	NA	2005	July
711	SPORT-100 HELMET, BLUE	NA	2005	July
712	AWC LOGO CAP	NA	2005	July
713	LONG-SLEEVE LOGO JERSEY, S	NA	2005	July
714	LONG-SLEEVE LOGO JERSEY, M	NA	2005	July

Table 1.72: Displaying records 1 - 10

ProductID	ProductName	ApproxWeight	SellStartYear	SellStartMonth	ProductType
680	HL ROAD FRAME - BLACK, 58	1016	2002	June	FR
706	HL ROAD FRAME - RED, 58	1016	2002	June	FR
707	SPORT-100 HELMET, RED	NA	2005	July	HL
708	SPORT-100 HELMET, BLACK	NA	2005	July	HL
709	MOUNTAIN BIKE SOCKS, M	NA	2005	July	SO
710	MOUNTAIN BIKE SOCKS, L	NA	2005	July	SO
711	SPORT-100 HELMET, BLUE	NA	2005	July	HL
712	AWC LOGO CAP	NA	2005	July	CA
713	LONG-SLEEVE LOGO JERSEY, S	NA	2005	July	LJ
714	LONG-SLEEVE LOGO JERSEY, M	NA	2005	July	LJ

Extend your query to include a column named ProductType that contains the leftmost two characters from the product number.

```sql
SELECT ProductID, UPPER(Name) AS ProductName, ROUND(Weight, 0) AS ApproxWeight,
       YEAR(SellStartDate) as SellStartYear,
       DATENAME(m, SellStartDate) as SellStartMonth,
       LEFT(ProductNumber, 2) AS ProductType
FROM SalesLT.Product;
```

Extend your query to filter the product returned so that only products with a numeric size are included.

```sql
SELECT ProductID, UPPER(Name) AS ProductName, ROUND(Weight, 0) AS ApproxWeight,
       YEAR(SellStartDate) as SellStartYear,
       DATENAME(m, SellStartDate) as SellStartMonth,
       LEFT(ProductNumber, 2) AS ProductType
FROM SalesLT.Product
WHERE ISNUMERIC(SIZE) = 1;
```

Write a query that returns a list of company names with a ranking of their place in a list of highest TotalDue values from the SalesOrderHeader table.

```sql
SELECT CompanyName, TotalDue AS Revenue,
       RANK() OVER (ORDER BY TotalDue DESC) AS RankByRevenue
FROM SalesLT.SalesOrderHeader AS SOH
```

Table 1.73: Displaying records 1 - 10

ProductID	ProductName	ApproxWeight	SellStartYear	SellStartMonth	ProductType
680	HL ROAD FRAME - BLACK, 58	1016	2002	June	FR
706	HL ROAD FRAME - RED, 58	1016	2002	June	FR
717	HL ROAD FRAME - RED, 62	1043	2005	July	FR
718	HL ROAD FRAME - RED, 44	962	2005	July	FR
719	HL ROAD FRAME - RED, 48	980	2005	July	FR
720	HL ROAD FRAME - RED, 52	998	2005	July	FR
721	HL ROAD FRAME - RED, 56	1016	2005	July	FR
722	LL ROAD FRAME - BLACK, 58	1116	2005	July	FR
723	LL ROAD FRAME - BLACK, 60	1125	2005	July	FR
724	LL ROAD FRAME - BLACK, 62	1134	2005	July	FR

Table 1.74: Displaying records 1 - 10

CompanyName	Revenue	RankByRevenue
Action Bicycle Specialists	119960.82	1
Metropolitan Bicycle Supply	108597.95	2
Bulk Discount Store	98138.21	3
Eastside Department Store	92663.56	4
Riding Cycles	86222.81	5
Many Bikes Store	81834.98	6
Instruments and Parts Company	70698.99	7
Extreme Riding Supplies	63686.27	8
Trailblazing Sports	45992.37	9
Professional Sales and Service	43962.79	10

```
LEFT JOIN SalesLT.Customer AS C
ON SOH.CustomerID = C.CustomerID;
```

Write a query to retrieve a list of the product names and the total revenue calculated as the sum of the LineTotal from the SalesLT.SalesOrderDetail table, with the results sorted in descending order of total revenue.

```
SELECT Name, SUM(LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS SOD
LEFT JOIN SalesLT.Product AS P
ON SOD.ProductID = p.ProductID
GROUP BY P.Name
ORDER BY TotalRevenue DESC;
```

Modify the previous query to include sales totals for products that have a list price of more than 1000.

```
SELECT Name, SUM(LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS SOD
JOIN SalesLT.Product AS P
ON SOD.ProductID = P.ProductID
WHERE ListPrice > 1000
GROUP BY P.Name
ORDER BY TotalRevenue DESC;
```

Modify the previous query to only include products with total sales greater than 20000.

Table 1.75: Displaying records 1 - 10

Name	TotalRevenue
Touring-1000 Blue, 60	37191.49
Mountain-200 Black, 42	37178.84
Road-350-W Yellow, 48	36486.24
Mountain-200 Black, 38	35801.84
Touring-1000 Yellow, 60	23413.47
Touring-1000 Blue, 50	22887.07
Mountain-200 Silver, 42	20879.91
Road-350-W Yellow, 40	20411.88
Mountain-200 Black, 46	19277.92
Road-350-W Yellow, 42	18692.52

Table 1.76: Displaying records 1 - 10

Name	TotalRevenue
Touring-1000 Blue, 60	37191.49
Mountain-200 Black, 42	37178.84
Road-350-W Yellow, 48	36486.24
Mountain-200 Black, 38	35801.84
Touring-1000 Yellow, 60	23413.47
Touring-1000 Blue, 50	22887.07
Mountain-200 Silver, 42	20879.91
Road-350-W Yellow, 40	20411.88
Mountain-200 Black, 46	19277.92
Road-350-W Yellow, 42	18692.52

Table 1.77: 8 records

Name	TotalRevenue
Touring-1000 Blue, 60	37191.49
Mountain-200 Black, 42	37178.84
Road-350-W Yellow, 48	36486.24
Mountain-200 Black, 38	35801.84
Touring-1000 Yellow, 60	23413.47
Touring-1000 Blue, 50	22887.07
Mountain-200 Silver, 42	20879.91
Road-350-W Yellow, 40	20411.88

Table 1.78: 1 records

salesorderid	productid	unitprice	orderqty
71946	916	31.584	1

```
SELECT Name, SUM(LineTotal) AS TotalRevenue
FROM SalesLT.SalesOrderDetail AS SOD
JOIN SalesLT.Product AS P
ON SOD.ProductID = P.ProductID
WHERE P.ListPrice > 1000
GROUP BY P.Name
-- add having clause as per instructions
HAVING SUM(LineTotal) > 20000
ORDER BY TotalRevenue DESC;
```

1.6 Sub-queries and Apply

Subqueries are queries within queries (nested). The results of the inner or nested query are pasted to the outer query and behave much like an expression.

Our subquery might be scalr, for instance we might have a single value which is passed to the outer query. So we might be interested in the last order (the maxium id) and the details from that order, where the information is held in two different tables.

```
SELECT salesorderid, productid, unitprice, orderqty
From SalesLT.SalesOrderDetail
WHERE salesorderid =
    (SELECT MAX(salesorderid) AS lastorder
    FROM SalesLT.SalesOrderHeader);
```

The subquery is in brackets, and the system will treat whatever is in brackets as an individual unit.

In a multi-valued query, mutiple values are returned but as a single column set to the outer query. So we might be interested in every customer in Mexico. One will get returned from the subquery is a one dimensional array - a vector.

Table 1.79: Displaying records 1 - 10

CustomerID	SalesOrderID	OrderDate
29485	71782	2008-06-01
29531	71935	2008-06-01
29546	71938	2008-06-01
29568	71899	2008-06-01
29584	71895	2008-06-01
29612	71885	2008-06-01
29638	71915	2008-06-01
29644	71867	2008-06-01
29653	71858	2008-06-01
29660	71796	2008-06-01

1.6.1 Self Contained or Correlated Query

Queries we have looked at so far are self contained queries. The sub-query parts works, then it supplies its value to the outer query. Correlated sub-queries refer to elements of tables used in the outer query. Because the sub-query is searching for values in the outer query, this can be resource intensive. We might, for instance, make a reference to an employee number in the subquery, but that number is dependent on the result of the outer query, so the subqueriy works with the outer query.

As with sub-queries generally, it is best to build the query up iteratively, so you can a) check the query is working and b) check your final sub-query results with those of the individual components to make sure it is doing what it should be. However, this is more difficult when using a correlated query, since there is a dependency.

So our goal is for each customer list all sales on the last day that they made a sale.

So first, lets get a list of customers with their orders and dates.

```sql
SELECT CustomerID, SalesOrderID, OrderDate
FROM SalesLT.SalesOrderHeader AS SO1
ORDER BY CustomerID,OrderDate
```

Then let's say we want just the latest (max) order date in the db, for each customer ID we have, so we have their individual last order details.

```sql
SELECT CustomerID, SalesOrderID, OrderDate
FROM SalesLT.SalesOrderHeader AS SO1
WHERE orderdate =
    (SELECT MAX(orderdate)
FROM SalesLT.SalesOrderHeader AS SO2
WHERE SO2.CustomerID = SO1.CustomerID)
ORDER BY CustomerID;
```

1.6.2 The Apply Operator

Using an apply operator and a table function can achieve something similar to a correlated query, but in less complicated code. A CROSS APPLY applies the right table expression to each row in the left table, a bit like the correlated sub-query, with CROSS APPLY being similar to a CROSS JOIN.

Similarily OUTER APPLY adds rows for those with NULL in columns for the right table, similar to LEFT OUTER JOIN. In these instances, the right table is the table returned by the APPLY function.

Table 1.80: Displaying records 1 - 10

CustomerID	SalesOrderID	OrderDate
29485	71782	2008-06-01
29531	71935	2008-06-01
29546	71938	2008-06-01
29568	71899	2008-06-01
29584	71895	2008-06-01
29612	71885	2008-06-01
29638	71915	2008-06-01
29644	71867	2008-06-01
29653	71858	2008-06-01
29660	71796	2008-06-01

Table 1.81: 1 records

Text
– Setup CREATE FUNCTION SalesLT.udfMaxUnitPrice (@SalesOrderID int) RETURNS TABLE AS RETURN SELEC

The first thing we need to do is to create a function. Notice we are passing the function the SalesOrderID, and the most expensive item in the order gets passed back.

```
-- Setup
CREATE FUNCTION SalesLT.udfMaxUnitPrice (@SalesOrderID int)
RETURNS TABLE
AS
RETURN
SELECT SalesOrderID,Max(UnitPrice) as MaxUnitPrice FROM
SalesLT.SalesOrderDetail
WHERE SalesOrderID=@SalesOrderID
GROUP BY SalesOrderID;
```

To see what code created a function, to help understand what it does at a later time or if created by someone else, we can use the sp_helptext command.

```
sp_helptext 'saleslt.udfmaxunitprice'
```

Then we can use that query to get the results. Note this could have been achieved with a correlated sub-query, but this is perahps a little easier to interpret.

```
SELECT SOH.SalesOrderID, MUP.MaxUnitPrice
FROM SalesLT.SalesOrderDetail AS SOH
CROSS APPLY SalesLT.udfMaxUnitPrice(SOH.SalesOrderID) AS MUP
ORDER BY SOH.SalesOrderID;
```

Key Points:

- The APPLY operator enables you to execute a table-valued function for each row in a rowset returned by a SELECT statement. Conceptually, this approach is similar to a correlated subquery.
- CROSS APPLY returns matching rows, similar to an inner join. OUTER APPLY returns all rows in the original SELECT query results with NULL values for rows where no match was found.

Table 1.82: Displaying records 1 - 10

SalesOrderID	MaxUnitPrice
71774	356.898
71774	356.898
71776	63.900
71780	1391.994
71780	1391.994
71780	1391.994
71780	1391.994
71780	1391.994
71780	1391.994
71780	1391.994

Table 1.83: Displaying records 1 - 10

ProductID	Name	ListPrice
680	HL Road Frame - Black, 58	1431.50
706	HL Road Frame - Red, 58	1431.50
717	HL Road Frame - Red, 62	1431.50
718	HL Road Frame - Red, 44	1431.50
719	HL Road Frame - Red, 48	1431.50
720	HL Road Frame - Red, 52	1431.50
721	HL Road Frame - Red, 56	1431.50
731	ML Road Frame - Red, 44	594.83
732	ML Road Frame - Red, 48	594.83
733	ML Road Frame - Red, 52	594.83

1.6.3 Lab Exercises

AdventureWorks products each have a standard cost that indicates the cost of manufacturing the product, and a list price that indicates the recommended selling price for the product. This data is stored in the SalesLT.Product table.

Whenever a product is ordered, the actual unit price at which it was sold is also recorded in the SalesLT.SalesOrderDetail table.

Use subqueries to compare the cost and list prices for each product with the unit prices charged in each sale.

```
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
WHERE ListPrice >
    (SELECT AVG(UnitPRice) FROM SalesLT.SalesOrderDetail)
ORDER BY ProductID;
```

AdventureWorks is interested in finding out which products are being sold at a loss. Retrieve the product ID, name, and list price for each product where the list price is 100 or more, and the product has been sold for (strictly) less than 100.

```
SELECT ProductID, Name, ListPrice
FROM SalesLT.Product
WHERE ProductID IN
  (SELECT ProductID FROM SalesLT.SalesOrderDetail
   WHERE UnitPrice < 100)
```

Table 1.84: 7 records

ProductID	Name	ListPrice
810	HL Mountain Handlebars	120.27
813	HL Road Handlebars	120.27
876	Hitch Rack - 4-Bike	120.00
894	Rear Derailleur	121.46
907	Rear Brakes	106.50
948	Front Brakes	106.50
996	HL Bottom Bracket	121.49

Table 1.85: Displaying records 1 - 10

ProductID	Name	StandardCost	ListPrice	AvgSellingPrice
680	HL Road Frame - Black, 58	1059.3100	1431.50	NA
706	HL Road Frame - Red, 58	1059.3100	1431.50	NA
707	Sport-100 Helmet, Red	13.0863	34.99	20.9940
708	Sport-100 Helmet, Black	13.0863	34.99	20.6441
709	Mountain Bike Socks, M	3.3963	9.50	NA
710	Mountain Bike Socks, L	3.3963	9.50	NA
711	Sport-100 Helmet, Blue	13.0863	34.99	20.7440
712	AWC Logo Cap	6.9223	8.99	5.3740
713	Long-Sleeve Logo Jersey, S	38.4923	49.99	NA
714	Long-Sleeve Logo Jersey, M	38.4923	49.99	29.9940

```
AND ListPrice >= 100
ORDER BY ProductID;
```

In order to get an idea of how many products are selling above or below list price, you want to gather some aggregate product data. Retrieve the product ID, name, cost, and list price for each product along with the average unit price for which that product has been sold.

```
SELECT ProductID, Name, StandardCost, ListPrice,
    (SELECT AVG(UnitPrice)
    FROM SalesLT.SalesOrderDetail AS SOD
    WHERE P.ProductID = SOD.ProductID) AS AvgSellingPrice
FROM SalesLT.Product AS P
ORDER BY P.ProductID;
```

AdventureWorks is interested in finding out which products are costing more than they're being sold for, on average. Filter the query for the previous exercise to include only products where the cost is higher than the average selling price.

```
SELECT ProductID, Name, StandardCost, ListPrice,
    (SELECT AVG(UnitPrice)
     FROM SalesLT.SalesOrderDetail AS SOD
     WHERE P.ProductID = SOD.ProductID) AS AvgSellingPrice
FROM SalesLT.Product AS P
WHERE StandardCost >
    (SELECT AVG(UnitPrice)
     FROM SalesLT.SalesOrderDetail AS SOD
     WHERE P.ProductID = SOD.ProductID)
ORDER BY P.ProductID;
```

Table 1.86: Displaying records 1 - 10

ProductID	Name	StandardCost	ListPrice	AvgSellingPrice
712	AWC Logo Cap	6.9223	8.99	5.374
714	Long-Sleeve Logo Jersey, M	38.4923	49.99	29.994
715	Long-Sleeve Logo Jersey, L	38.4923	49.99	29.744
716	Long-Sleeve Logo Jersey, XL	38.4923	49.99	29.994
717	HL Road Frame - Red, 62	868.6342	1431.50	858.900
718	HL Road Frame - Red, 44	868.6342	1431.50	858.900
722	LL Road Frame - Black, 58	204.6251	337.22	202.332
738	LL Road Frame - Black, 52	204.6251	337.22	202.332
792	Road-250 Red, 58	1554.9479	2443.35	1466.010
793	Road-250 Black, 44	1554.9479	2443.35	1466.010

Table 1.87: Displaying records 1 - 10

SalesOrderID	CustomerID	FirstName	LastName	TotalDue
71774	29847	David	Hodgson	972.7850
71776	30072	Andrea	Thomsen	87.0851
71780	30113	Raja	Venugopal	42452.6519
71782	29485	Catherine	Abel	43962.7901
71783	29957	Kevin	Liu	92663.5609
71784	29736	Terry	Eminhizer	119960.8240
71796	29660	Anthony	Chor	63686.2708
71797	29796	Jon	Grande	86222.8072
71815	30089	Michael John	Troyer	1261.4440
71816	30027	Joseph	Mitzner	3754.9733

The AdventureWorksLT database includes a table-valued user-defined function named dbo.ufnGetCustomerInformation. Use this function to retrieve details of customers based on customer ID values retrieved from tables in the database.

Retrieve the sales order ID, customer ID, first name, last name, and total due for all sales orders from the SalesLT.SalesOrderHeader table and the dbo.ufnGetCustomerInformation function.

```sql
SELECT SOH.SalesOrderID, SOH.CustomerID, CI.FirstName, CI.LastName, SOH.TotalDue
FROM SalesLT.SalesOrderHeader AS SOH
CROSS APPLY dbo.ufnGetCustomerInformation(SOH.CustomerID) AS CI
ORDER BY SOH.SalesOrderID;
```

Use the table-valued user-defined function dbo.ufnGetCustomerInformation again to retrieve details of customers based on customer ID values retrieved from tables in the database. Retrieve the customer ID, first name, last name, address line 1 and city for all customers from the SalesLT.Address and SalesLT.CustomerAddress tables, using the dbo.ufnGetCustomerInformation function.

```sql
SELECT CA.CustomerID, CI.FirstName, CI.LastName, A.AddressLine1, A.City
FROM SalesLT.Address AS A
JOIN SalesLT.CustomerAddress AS CA
ON A.AddressID = CA.AddressID
CROSS APPLY dbo.ufnGetCustomerInformation(CA.CustomerID) AS CI
ORDER BY CA.CustomerID;
```

Table 1.88: Displaying records 1 - 10

CustomerID	FirstName	LastName	AddressLine1	City
29485	Catherine	Abel	57251 Serene Blvd	Van Nuys
29486	Kim	Abercrombie	Tanger Factory	Branch
29489	Frances	Adams	6900 Sisk Road	Modesto
29490	Margaret	Smith	Lewiston Mall	Lewiston
29492	Jay	Adams	Blue Ridge Mall	Kansas City
29494	Samuel	Agcaoili	No. 25800-130 King Street West	Toronto
29496	Robert	Ahlering	6500 East Grant Road	Tucson
29497	François	Ferrier	Eastridge Mall	Casper
29499	Amy	Alberts	252851 Rowan Place	Richmond
29502	Paul	Alcorn	White Mountain Mall	Rock Springs

1.7 Using Table Expressions

We can create tables in a number of ways that can then be re-usd, rather than being a one-off query.

1.7.1 Views

One way we can do this is using a view. Rather than repeatadly joining and selected fields from two tables, we can create a view that contains the join syntax then just select the view. This view can be used and queried as if it were a table. The data is still in the underlying tables, but it is a view or presentation on top of the tables. It's like a named query, which can make this simpler, plus we can add data at the view level, so they can view rights for the view table, but not the underlying data. Note that whilst you are able to use insert to add new rows, you can only do this to one of the underlying tables.

First we create a view

```
-- Create a view
CREATE VIEW SalesLT.vCustomerAddress
AS
SELECT C.CustomerID, FirstName, LastName, AddressLine1, City, StateProvince
FROM
SalesLT.Customer C JOIN SalesLT.CustomerAddress CA
ON C.CustomerID=CA.CustomerID
JOIN SalesLT.Address A
ON CA.AddressID=A.AddressID
```

Then we can query the view.

```
SELECT CustomerID, City
FROM SalesLT.vCustomerAddress
```

And we can join data to that view.

```
SELECT c.StateProvince, c.City, ISNULL(SUM(s.TotalDue), 0.00) AS Revenue
FROM SalesLT.vCustomerAddress AS c
LEFT JOIN SalesLT.SalesOrderHeader AS s
ON s.CustomerID = c.CustomerID
GROUP BY c.StateProvince, c.City
ORDER BY c.StateProvince, Revenue DESC;
```

Table 1.89: Displaying records 1 - 10

CustomerID	City
29698	Burnaby
29997	Seattle
29854	Joliet
30027	Oxnard
30023	Peoria
29637	Irving
29545	Bothell
29890	Port Orchard
29772	Austin
29883	Denver

Table 1.90: Displaying records 1 - 10

StateProvince	City	Revenue
Alberta	Calgary	0
Alberta	Edmonton	0
Arizona	Chandler	0
Arizona	Gilbert	0
Arizona	Mesa	0
Arizona	Phoenix	0
Arizona	Scottsdale	0
Arizona	Surprise	0
Arizona	Tucson	0
British Columbia	Vancouver	0

1.7.2 Using Temporary Tables and Table Variables

Views are persistent database objects, however we might want a temporary table. We prefix the object with a # symbol and they are created in a seperate temporary database called tempdb, rather than the db you are working in. Typically using a single # will mean the table exists for the current user session. If you wanted it to persistent over multiple sessions, use the ## prefix.

An alternative approach is to use table variables, which were introduced to cause performance problems if used again, as the table has to be re-created each user session, which can become a problem if there are many tables. If we want to use one we prefix it with an @ sign to signify it is a variable e.g. **?** table so we define the variable as a table. This is connected to the batch rather than the session, so as long as we are in the block of code. It only works well on small databases or queries. They are for temporary situations or a temporary holding space.

In the next block of code we show how we create a temporary table then update and query it.

```
CREATE TABLE #Colors
(Color varchar(15));

INSERT INTO #Colors
SELECT DISTINCT Color FROM SalesLT.Product;

SELECT * FROM #Colors;
```

A table variable is similar, with some syntax differences, but this works on the code set (batch of commans ran at the same time) rather than being for the entire session.

```
DECLARE @Colors AS TABLE (Color varchar(15));

INSERT INTO @Colors
SELECT DISTINCT Color FROM SalesLT.Product;

SELECT * FROM @Colors;
```

Key Points

- Temporary tables are prefixed with a # symbol (You can also create global temporary tables that can be accessed by other processes by prefixing the name with ##)
- Local temporary tables are automatically deleted when the session in which they were created ends. Global temporary tables are deleted when the last user sessions referencing them is closed.
- Table variables are prefixed with a @ symbol.
- Table variables are scoped to the batch in which they are created.

1.7.3 Table Value Functions (TVF)

This is a specific type of function which returns a table. It is a permanently define database object. Unlike views, we can pass values in to a TVF.

So first we create a TVF.

```
CREATE FUNCTION SalesLT.udfCustomersByCity
(@City AS VARCHAR(20))
RETURNS TABLE
AS
RETURN
    (SELECT C.CustomerID, FirstName, LastName, AddressLine1, City, StateProvince
     FROM SalesLT.Customer C JOIN SalesLT.CustomerAddress CA
```

Table 1.91: 2 records

CustomerID	FirstName	LastName	AddressLine1	City	StateProvince
29559	Robert	Bernacchi	2681 Eagle Peak	Bellevue	Washington
29559	Robert	Bernacchi	25915 140th Ave Ne	Bellevue	Washington

Table 1.92: Displaying records 1 - 10

Category	Products
Bib-Shorts	3
Bike Racks	1
Bike Stands	1
Bottles and Cages	3
Bottom Brackets	3
Brakes	2
Caps	1
Chains	1
Cleaners	1
Cranksets	3

```
     ON C.CustomerID=CA.CustomerID
     JOIN SalesLT.Address A ON CA.AddressID=A.AddressID
     WHERE City=@City);
```

Then we can call the TVF passing a parameter - in this instance we prvoide a city name of Bellevue - which will then return the results of the SELECT statement which is a list of customers in that city.

```
SELECT * FROM SalesLT.udfCustomersByCity('Bellevue')
```

1.7.4 Derived Tables

These are derived tables that return a multi column table. They are a virtual table to simplify a query. The table only exists within that SELECT query. It is a programming construct within a SELECT statement, it is used for programming purposes. They can use internal (aka inline) within the derived table element, or external aliases for columns - outside the () of the derived table.

In the example derived table below, the alias ProdCats is provided external to the query (outside the parentheses). Other elements such as Category as defined as internal aliases.

```
SELECT Category, COUNT(ProductID) AS Products
FROM
    (SELECT p.ProductID, p.Name AS Product, c.Name AS Category
     FROM SalesLT.Product AS p
     JOIN SalesLT.ProductCategory AS c
     ON p.ProductCategoryID = c.ProductCategoryID) AS ProdCats
GROUP BY Category
ORDER BY Category;
```

Table 1.93: Displaying records 1 - 10

Category	Products
Bib-Shorts	3
Bike Racks	1
Bike Stands	1
Bottles and Cages	3
Bottom Brackets	3
Brakes	2
Caps	1
Chains	1
Cleaners	1
Cranksets	3

1.7.5 Using Common Table Expressions (CTEs)

Similar to a TVF before we are defining a temporary table object which is used within the scope of the query. But this time aorund we define the CTE first, then make calls to that CTE afterwards. Unlike a derived query however, you can refer to the CTE mutiple times within the same query/code batch, but it won't live on like a view. We can also use a CTE for recursive elements like loops, so you might want to get everyone within a management tier, then say you want to go through that process three times for the top 3 levels, using the OPTION(MAXRECURSION 3).

So in the following example, we first define the CTE, then we make reference to it. The result here is the same as our derived query previously, but perhaps a little easiter to understand.

```
WITH ProductsByCategory (ProductID, ProductName, Category)
AS
(
    SELECT p.ProductID, p.Name, c.Name AS Category
     FROM SalesLT.Product AS p
     JOIN SalesLT.ProductCategory AS c
     ON p.ProductCategoryID = c.ProductCategoryID
)

SELECT Category, COUNT(ProductID) AS Products
FROM ProductsByCategory
GROUP BY Category
ORDER BY Category;
```

Next we will look at the our organisation chart example as previously mentioned. First let's look at our employee table.

```
SELECT * FROM SalesLT.Employee
```

We can see that each employee has a manager ID apart from Employee 3 who is the CEO/MD. So we have a hierarchy of employees. We will go 3 levels deep this time (three recursions).

```
WITH OrgReport (ManagerID, EmployeeID, EmployeeName, Level)
AS
(
    -- Anchor query
    SELECT e.ManagerID, e.EmployeeID, EmployeeName, 0
    FROM SalesLT.Employee AS e
    WHERE ManagerID IS NULL
```

Table 1.94: 9 records

EmployeeID	EmployeeName	ManagerID
1	adventure-works\david8	8
2	adventure-works\garrett1	1
3	adventure-works\jae0	NA
4	adventure-works\jillian0	3
5	adventure-works\josé1	1
6	adventure-works\linda3	3
7	adventure-works\michael9	9
8	adventure-works\pamela0	3
9	adventure-works\shu0	3

Table 1.95: 9 records

ManagerID	EmployeeID	EmployeeName	Level
NA	3	adventure-works\jae0	0
3	4	adventure-works\jillian0	1
3	6	adventure-works\linda3	1
3	8	adventure-works\pamela0	1
3	9	adventure-works\shu0	1
9	7	adventure-works\michael9	2
8	1	adventure-works\david8	2
1	2	adventure-works\garrett1	3
1	5	adventure-works\josé1	3

```
    UNION ALL

    -- Recursive query
    SELECT e.ManagerID, e.EmployeeID, e.EmployeeName, Level + 1
    FROM SalesLT.Employee AS e
    INNER JOIN OrgReport AS o ON e.ManagerID = o.EmployeeID
)

SELECT * FROM OrgReport
OPTION (MAXRECURSION 3);
```

Key Points

- A derived table is a subquery that generates a multicolumn rowset. You must use the AS clause to define an alias for a derived query.
- Common Table Expressions (CTEs) provide a more intuitive syntax or defining rowsets than derived tables, and can be used mulitple times in the same query.
- You can use CTEs to define recursive queries.

1.7.6 Lab Exercises

AdventureWorks sells many products that are variants of the same product model. You must write queries that retrieve information about these products. Retrieve the product ID, product name, product model name, and product model summary for each product from the SalesLT.Product table and the SalesLT.vProductModelCatalogDescription view.

Table 1.96: Displaying records 1 - 10

ProductID	ProductName	ProductModel	Summary
749	Road-150 Red, 62	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
750	Road-150 Red, 44	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
751	Road-150 Red, 48	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
752	Road-150 Red, 52	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
753	Road-150 Red, 56	Road-150	This bike is ridden by race winners. Developed with the Adventure Wo
754	Road-450 Red, 58	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona
755	Road-450 Red, 60	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona
756	Road-450 Red, 44	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona
757	Road-450 Red, 48	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona
758	Road-450 Red, 52	Road-450	A true multi-sport bike that offers streamlined riding and a revolutiona

```sql
SELECT P.ProductID, P.Name AS ProductName, PM.Name AS ProductModel, PM.Summary
FROM SalesLT.Product AS P
JOIN SalesLT.vProductModelCatalogDescription AS PM
ON P.ProductModelID = PM.ProductModelID
ORDER BY ProductID;
```

You are only interested in products which have a listed color in the database. Create a table variable and populate it with a list of distinct colors from the SalesLT.Product table. Then use the table variable to filter a query that returns the product ID, name, and color from the SalesLT.Product table so that only products with a color listed in the table variable are returned.

```sql
DECLARE @Colors AS TABLE (Color NVARCHAR(15));

INSERT INTO @Colors
SELECT DISTINCT Color FROM SalesLT.Product;

SELECT ProductID, Name, Color
FROM SalesLT.Product
WHERE Color IN (SELECT Color FROM @Colors);
```

The AdventureWorksLT database includes a table-valued function named dbo.ufnGetAllCategories, which returns a table of product categories (e.g. ‘Road Bikes’) and parent categories (for example ‘Bikes’). Write a query that uses this function to return a list of all products including their parent category and their own category.

```sql
SELECT C.ParentProductCategoryName AS ParentCategory,
       C.ProductCategoryName AS Category,
       P.ProductID, P.Name AS ProductName
FROM SalesLT.Product AS P
JOIN dbo.ufnGetAllCategories() AS C
ON P.ProductCategoryID = C.ProductCategoryID
ORDER BY ParentCategory, Category, ProductName;
```

Each AdventureWorks customer is a retail company with a named contact. You must create queries that return the total revenue for each customer, including the company and customer contact names. Retrieve a list of customers in the format Company (Contact Name) together with the total revenue for each customer. Use a derived table or a common table expression to retrieve the details for each sales order, and then query the derived table or CTE to aggregate and group the data.

Table 1.97: Displaying records 1 - 10

ParentCategory	Category	ProductID	ProductName
Accessories	Bike Racks	876	Hitch Rack - 4-Bike
Accessories	Bike Stands	879	All-Purpose Bike Stand
Accessories	Bottles and Cages	871	Mountain Bottle Cage
Accessories	Bottles and Cages	872	Road Bottle Cage
Accessories	Bottles and Cages	870	Water Bottle - 30 oz.
Accessories	Cleaners	877	Bike Wash - Dissolver
Accessories	Fenders	878	Fender Set - Mountain
Accessories	Helmets	708	Sport-100 Helmet, Black
Accessories	Helmets	711	Sport-100 Helmet, Blue
Accessories	Helmets	707	Sport-100 Helmet, Red

Table 1.98: Displaying records 1 - 10

CompanyContact	Revenue
Action Bicycle Specialists (Terry Eminhizer)	119960.8240
Aerobic Exercise Company (Rosmarie Carroll)	2361.6403
Bulk Discount Store (Christopher Beck)	98138.2131
Central Bicycle Specialists (Janeth Esteves)	43.0437
Channel Outlet (Richard Byham)	608.1766
Closest Bicycle Store (Pamala Kotc)	39531.6085
Coalition Bike Company (Donald Blanton)	2669.3183
Discount Tours (Melissa Marple)	3293.7761
Eastside Department Store (Kevin Liu)	92663.5609
Engineered Bike Systems (Joseph Mitzner)	3754.9733

```sql
SELECT CompanyContact, SUM(SalesAmount) AS Revenue
FROM
    (SELECT CONCAT(c.CompanyName, CONCAT(' (' + c.FirstName + ' ', c.LastName + ')')), SOH.TotalDue
     FROM SalesLT.SalesOrderHeader AS SOH
     JOIN SalesLT.Customer AS c
     ON SOH.CustomerID = c.CustomerID) AS CustomerSales(CompanyContact, SalesAmount)
GROUP BY CompanyContact
ORDER BY CompanyContact;
```

1.8 Grouping Sets and Pivoting Data

These are both ways of summarising data - grouping data includes cubing the data.

Grouping sets allows you to define multiple groupings within the same query. So you may haved a number of subtotals for different product types, then a total of all the subtotals combined.

So you would have something like

> SELECT FROM
>
> GROUP BY GROUPING SETS (
>
> ,

Table 1.99: Displaying records 1 - 10

ParentProductCategoryName	ProductCategoryName	Products
Accessories	Bike Racks	1
Accessories	Bike Stands	1
Accessories	Bottles and Cages	3
Accessories	Cleaners	1
Accessories	Fenders	1
Accessories	Helmets	3
Accessories	Hydration Packs	1
Accessories	Lights	3
Accessories	Locks	1
Accessories	Panniers	1

() – empty parentheses if aggregating all rows)

A result line will appear with NULLs in all column(s) which is the grand total.

There are other ways to aggregate the data such as ROLLUP and CUBE. ROLLUP is useful if you haave hierarchy - people in households, towns in regions. Cube will show every possible aggregation, rather than specifying each aggregation.

Both are specified in the GROUP BY X () where x is either ROLLUP or CUBE. You can add in GROUPING_ID

AS fiels to make the interpretation easier when using multiple grouping columns. You can then see the grand total more easily, since it will have a 1 in each groupid column.

So a usual group by might look something like this - with a high level Parent Category followed by a Product Category and a count of items.

```
SELECT cat.ParentProductCategoryName, cat.ProductCategoryName, count(prd.ProductID) AS Products
FROM SalesLT.vGetAllCategories as cat
LEFT JOIN SalesLT.Product AS prd
ON prd.ProductCategoryID = cat.ProductcategoryID
GROUP BY cat.ParentProductCategoryName, cat.ProductCategoryName
ORDER BY cat.ParentProductCategoryName, cat.ProductCategoryName;
```

Here there are no grand total by parent categories. Also, there might be certain items appear in more than one category without totals e.g. bike racks might be in mutiple parent categories.

Alternatively, we might create GROUPING SETS which also allows us to create totals.

```
SELECT cat.ParentProductCategoryName, cat.ProductCategoryName, count(prd.ProductID) AS Products
FROM SalesLT.vGetAllCategories as cat
LEFT JOIN SalesLT.Product AS prd
ON prd.ProductCategoryID = cat.ProductcategoryID
GROUP BY GROUPING SETS(cat.ParentProductCategoryName, cat.ProductCategoryName, ())
ORDER BY cat.ParentProductCategoryName, cat.ProductCategoryName;
```

We now start getting totals, we can see there is a total of 295 products, followed by 3 bib-shorts (irrespective of parent), and later in the table we get the totals by parent categories. But we have no loss some of the relationships with parent categories with this query. It is possible to write a query where these things might appear, GROUPING SETS is the most flexible of the grouping commands, but it might be easier to use one of the other grouping options as required, next shows the the ROLLUP command.

Table 1.100: Displaying records 1 - 10

ParentProductCategoryName	ProductCategoryName	Products
NA	NA	295
NA	Bib-Shorts	3
NA	Bike Racks	1
NA	Bike Stands	1
NA	Bottles and Cages	3
NA	Bottom Brackets	3
NA	Brakes	2
NA	Caps	1
NA	Chains	1
NA	Cleaners	1

Table 1.101: Displaying records 1 - 10

ParentProductCategoryName	ProductCategoryName	Products
NA	NA	295
Accessories	NA	29
Accessories	Bike Racks	1
Accessories	Bike Stands	1
Accessories	Bottles and Cages	3
Accessories	Cleaners	1
Accessories	Fenders	1
Accessories	Helmets	3
Accessories	Hydration Packs	1
Accessories	Lights	3

```sql
SELECT cat.ParentProductCategoryName, cat.ProductCategoryName, count(prd.ProductID) AS Products
FROM SalesLT.vGetAllCategories as cat
LEFT JOIN SalesLT.Product AS prd
ON prd.ProductCategoryID = cat.ProductcategoryID
GROUP BY ROLLUP (cat.ParentProductCategoryName, cat.ProductCategoryName)
ORDER BY cat.ParentProductCategoryName, cat.ProductCategoryName;
```

It is now giving us totals for parents and products. However, we are now missing situations where items might appear in more than one parent category e.g. bottles and cages that are in a category other than accessories. So we can use CUBE to get these other aggregation options, which will give us all possible aggregations.

```sql
SELECT cat.ParentProductCategoryName, cat.ProductCategoryName, count(prd.ProductID) AS Products
FROM SalesLT.vGetAllCategories as cat
LEFT JOIN SalesLT.Product AS prd
ON prd.ProductCategoryID = cat.ProductcategoryID
GROUP BY CUBE (cat.ParentProductCategoryName, cat.ProductCategoryName)
ORDER BY cat.ParentProductCategoryName, cat.ProductCategoryName;
```

Key Points * Use GROUPING SETS to define custom groupings. * Use ROLLUP to include subtotals and a grand total for hierarchical groupings. * Use CUBE to include all possible groupings.

Table 1.102: Displaying records 1 - 10

ParentProductCategoryName	ProductCategoryName	Products
NA	NA	295
NA	Bib-Shorts	3
NA	Bike Racks	1
NA	Bike Stands	1
NA	Bottles and Cages	3
NA	Bottom Brackets	3
NA	Brakes	2
NA	Caps	1
NA	Chains	1
NA	Cleaners	1

Table 1.103: Displaying records 1 - 10

Name	Red	Blue	Black	Silver	Yellow	Grey	Multi	Uncolored
Bib-Shorts	0	0	0	0	0	0	3	0
Bike Racks	0	0	0	0	0	0	0	1
Bike Stands	0	0	0	0	0	0	0	1
Bottles and Cages	0	0	0	0	0	0	0	3
Bottom Brackets	0	0	0	0	0	0	0	3
Brakes	0	0	0	2	0	0	0	0
Caps	0	0	0	0	0	0	1	0
Chains	0	0	0	1	0	0	0	0
Cleaners	0	0	0	0	0	0	0	1
Cranksets	0	0	3	0	0	0	0	0

1.8.1 Pivoting Data

Pivoting is another form of summarising data. We take row based items and pivot them in to single, summarised, columns in a new table. We simply use the PIVOT command. It is possible to undue this using UNPIVOT however this may loose some of the detail that existed in the original data. For instance line level data, such as which particular clothing items were purchased and at which value, is now lost as we just get a clothing total, but re-orientated to match the original data format.

We might want to be able to see the totals by colour in our data and by cateory. You have to know which columns you wish to create using the PIVOT table, for instance in the code below we are listing the actual colours our in the PIVOT command. If one colour was missed, the query would work, but you obviously wouldn't get that colour returned or inlcuded in totals.

```
SELECT * FROM
    (SELECT P.ProductID, PC.Name,ISNULL(P.Color, 'Uncolored') AS Color
     FROM saleslt.productcategory AS PC
     JOIN SalesLT.Product AS P
     ON PC.ProductCategoryID=P.ProductCategoryID
     ) AS PPC
PIVOT(COUNT(ProductID) FOR Color IN([Red],[Blue],[Black],[Silver],[Yellow],[Grey], [Multi], [Uncolored]
ORDER BY Name;
```

If we wanted to, we could copy/archive this in to a new table, which can be re-used again or compared and retrieved in the future.

```
CREATE TABLE #ProductColorPivot
(Name varchar(50), Red int, Blue int, Black int, Silver int, Yellow int, Grey int , multi int, uncolore

INSERT INTO #ProductColorPivot
    SELECT * FROM
    (SELECT P.ProductID, PC.Name,ISNULL(P.Color, 'Uncolored') AS Color
     FROM saleslt.productcategory AS PC
     JOIN SalesLT.Product AS P
     ON PC.ProductCategoryID=P.ProductCategoryID
     ) AS PPC
PIVOT(COUNT(ProductID) FOR Color IN([Red],[Blue],[Black],[Silver],[Yellow],[Grey], [Multi], [Uncolored]
ORDER BY Name;
```

If we then wanted to unpvivot the data we would do so as follows.

```
SELECT Name, Color, ProductCount
FROM
    (SELECT Name,
    [Red],[Blue],[Black],[Silver],[Yellow],[Grey], [Multi], [Uncolored]
    FROM #ProductColorPivot) pcp
UNPIVOT
(ProductCount FOR Color IN ([Red],[Blue],[Black],[Silver],[Yellow],[Grey], [Multi], [Uncolored])
) AS ProductCounts
```

Key Points * Use PIVOT to re-orient a rowset by generating mulitple columns from values in a single column. * Use UNPIVOT to re-orient mulitple columns in a an existing rowset into a single column.

1.8.2 Lab Exercises

AdventureWorks sells products to customers in multiple country/regions around the world.

An existing report uses the query provided in the editor to return total sales revenue grouped by country/region and state/province. Modify the query so that the results include a grand total for all sales revenue and a subtotal for each country/region in addition to the state/province subtotals that are already returned.

```
SELECT a.CountryRegion, a.StateProvince, SUM(soh.TotalDue) AS Revenue
FROM SalesLT.Address AS a
JOIN SalesLT.CustomerAddress AS ca
ON a.AddressID = ca.AddressID
JOIN SalesLT.Customer AS c
ON ca.CustomerID = c.CustomerID
JOIN SalesLT.SalesOrderHeader as soh
ON c.CustomerID = soh.CustomerID
GROUP BY ROLLUP (a.CountryRegion, a.StateProvince)
ORDER BY a.CountryRegion, a.StateProvince;
```

Modify your query to include a column named Level that indicates at which level in the total, country/region, and state/province hierarchy the revenue figure in the row is aggregated.

```
SELECT a.CountryRegion, a.StateProvince,
    IIF(GROUPING_ID(a.CountryRegion) = 1 AND GROUPING_ID(a.StateProvince) = 1, 'Total', IIF(GROUPING_ID
SUM(soh.TotalDue) AS Revenue
FROM SalesLT.Address AS a
JOIN SalesLT.CustomerAddress AS ca
ON a.AddressID = ca.AddressID
```

Table 1.104: 9 records

CountryRegion	StateProvince	Revenue
NA	NA	956303.5949
United Kingdom	NA	572496.5594
United Kingdom	England	572496.5594
United States	NA	383807.0355
United States	California	346517.6072
United States	Colorado	14017.9083
United States	Nevada	7330.8972
United States	New Mexico	15275.1977
United States	Utah	665.4251

Table 1.105: 9 records

CountryRegion	StateProvince	Level	Revenue
NA	NA	Total	956303.5949
United Kingdom	NA	United Kingdom Subtotal	572496.5594
United Kingdom	England	England Subtotal	572496.5594
United States	NA	United States Subtotal	383807.0355
United States	California	California Subtotal	346517.6072
United States	Colorado	Colorado Subtotal	14017.9083
United States	Nevada	Nevada Subtotal	7330.8972
United States	New Mexico	New Mexico Subtotal	15275.1977
United States	Utah	Utah Subtotal	665.4251

```
JOIN SalesLT.Customer AS c
ON ca.CustomerID = c.CustomerID
JOIN SalesLT.SalesOrderHeader as soh
ON c.CustomerID = soh.CustomerID
GROUP BY ROLLUP(a.CountryRegion, a.StateProvince)
ORDER BY a.CountryRegion, a.StateProvince;
```

Or to extend your query to include a grouping for individual cities.

```
SELECT a.CountryRegion, a.StateProvince, a.City,
CHOOSE (1 + GROUPING_ID(a.CountryRegion) + GROUPING_ID(a.StateProvince) + GROUPING_ID(a.City),
        a.City + ' Subtotal', a.StateProvince + ' Subtotal',
        a.CountryRegion + ' Subtotal', 'Total') AS Level,
SUM(soh.TotalDue) AS Revenue
FROM SalesLT.Address AS a
JOIN SalesLT.CustomerAddress AS ca
ON a.AddressID = ca.AddressID
JOIN SalesLT.Customer AS c
ON ca.CustomerID = c.CustomerID
JOIN SalesLT.SalesOrderHeader as soh
ON c.CustomerID = soh.CustomerID
GROUP BY ROLLUP(a.CountryRegion, a.StateProvince, a.City)
ORDER BY a.CountryRegion, a.StateProvince, a.City;
```

AdventureWorks products are grouped into categories, which in turn have parent categories (defined in the SalesLT.vGetAllCategories view).

Table 1.106: Displaying records 1 - 10

CountryRegion	StateProvince	City	Level	Revenue
NA	NA	NA	Total	956303.5949
United Kingdom	NA	NA	United Kingdom Subtotal	572496.5594
United Kingdom	England	NA	England Subtotal	572496.5594
United Kingdom	England	Abingdon	Abingdon Subtotal	45.1995
United Kingdom	England	Cambridge	Cambridge Subtotal	2711.4098
United Kingdom	England	Gloucestershire	Gloucestershire Subtotal	70698.9922
United Kingdom	England	High Wycombe	High Wycombe Subtotal	608.1766
United Kingdom	England	Liverpool	Liverpool Subtotal	86222.8072
United Kingdom	England	London	London Subtotal	206736.1667
United Kingdom	England	Maidenhead	Maidenhead Subtotal	43.0437

Table 1.107: Displaying records 1 - 10

CompanyName	Accessories	Bikes	Clothing	Components
Action Bicycle Specialists	1299.885	76613.6518	2461.6573	9494.082
Aerobic Exercise Company	NA	NA	NA	1732.890
Bulk Discount Store	730.464	70597.2840	851.5620	1980.918
Central Bicycle Specialists	NA	NA	NA	31.584
Channel Outlet	216.000	NA	308.6640	NA
Closest Bicycle Store	NA	20389.6680	559.1641	8001.846
Coalition Bike Company	NA	529.4928	124.7760	1201.938
Discount Tours	72.000	2041.1880	341.0580	72.882
Eastside Department Store	1220.236	51096.0548	2772.2296	10594.848
Engineered Bike Systems	NA	2604.7620	178.7460	63.900

AdventureWorks customers are retail companies, and they may place orders for products of any category. The revenue for each product in an order is recorded as the LineTotal value in the SalesLT.SalesOrderDetail table.

Retrieve a list of customer company names together with their total revenue for each parent category in Accessories, Bikes, Clothing, and Components. Make sure to use the aliases provided, and default column names elsewhere.

```
SELECT * FROM
(SELECT cat.ParentProductCategoryName, cust.CompanyName, sod.LineTotal
 FROM SalesLT.SalesOrderDetail AS sod
 JOIN SalesLT.SalesOrderHeader AS soh ON sod.SalesOrderID = soh.SalesOrderID
 JOIN SalesLT.Customer AS cust ON soh.CustomerID = cust.CustomerID
 JOIN SalesLT.Product AS prod ON sod.ProductID = prod.ProductID
 JOIN SalesLT.vGetAllCategories AS cat ON prod.ProductcategoryID = cat.ProductCategoryID) AS catsales
PIVOT (SUM(LineTotal) FOR ParentProductCategoryName
IN ([Accessories], [Bikes], [Clothing], [Components])) AS pivotedsales
ORDER BY CompanyName;
```

1.9 Modifying Data

This section isn't about querying data, but adding, deleting and updating data in our database.

To add data we use the INSERT ... VALUES command, where you can leave blank columns that allow NULLs and cols with default constraints. You can also leave identify/primary key fields blank since they will be generated automatically, depending on what seed and increment values were specified when the table was set up, however it is possible to overide these values if you wish. You can add a NULL value to a column if you wish.

You can also insert the results of another query in to an existing table using INSERT ... SELECT or INSERT ... EXEC where EXEC is the execution of a stored query.

Another option is SELECT ... INTO will explicity create a new table, inserting data based on query. However this newly created table will not have contraints, defaults, indexes or primary keys, just a set of columns with the results of the query. This is currently not supported in Azure, however you could create the table first with the items needed such as columns and pk, then use the INSERT INTO for Azure.

Our Identity property of a column generates sequential numbers for insertations of new data into a database and we can set the parameters of this identity column. If you insert a new item (row) in to a table, you might want to then get back this identity column for this new item e.g. you automatically generate a new sales ID but a new order, however you might want to then present this back to the customer, this is acheived using @**?**, but this returns the last identity referenced, so if you are adding the sales order first then the individual items in a seperate table, it will retrieve the idenity column from the sales order details rather than sales order header. This command therefore returns the last identity from the current connected session.

An alternative is to use SELECT SCOPE_IDENTITY() AS ORDERID which wil bring back the last identity column for the specified table in the current session, but again this can be problematic. The alternative is to use IDENT_CURRENT(') which passes the last identity inserted into a particular table, however this is across all sessions, so if others are inserting the table this can be problematic. So generally the best option is to use SCOPE_IDENTITY and present that, before then going on to do other things.

If you perhaps wanted to have two different tables for two different order types - say in store purchases and online purchases as two dinstinct tables - but to both use the same sequential list for order ID, you could do this using sequences. They exist independently of tables so can be referenced by mutiple tables, the query will ask the server what the next id is in the sequence and be provided with that. It is like a central store that issues new numbers on request.

So to create a new table for customer call logs we do the following:

```
CREATE TABLE SalesLT.CallLog
(
    CallID int IDENTITY PRIMARY KEY NOT NULL, -- no seed or increment set, so will start at 1 then add
    CallTime datetime NOT NULL DEFAULT GETDATE(), -- if not date and time set, it will get the current
    SalesPerson nvarchar(256) NOT NULL,
    CustomerID int NOT NULL REFERENCES SalesLT.Customer(CustomerID), -- references a foreign key from t
    PhoneNumber nvarchar(25) NOT NULL,
    Notes nvarchar(max) NULL -- the only column which is allowed to be NULL, since perhaps the customer
);
GO
```

If we then wanted to explicitly add or insert a row in to the datbase we do the following - note the CallID does not need to be given since this is automatically created :

– INT is optional, we state values as we are explicitly stating the values

```
INSERT INTO SalesLT.CallLog
VALUES
('2015-01-01T12:30:00', 'adventure-works\pamela0', 1, '245-555-0173', 'Returning call re: enquiry about
```

If we wantedc to accept the default options, we can do that by using DEFAULT (e.g. for data/time) and if there are no notes we can use NULL.

```
INSERT INTO SalesLT.CallLog
VALUES
(DEFAULT, 'adventure-works\david8', 2, '170-555-0127', NULL);
```

Perhaps we need to re-order our fields rather than accepting the default sequence, perhaps it is a modified table or a table form another source. To do this, we explicitly state the columns.

```
INSERT INTO SalesLT.CallLog (SalesPerson, CustomerID, PhoneNumber)
VALUES
('adventure-works\jillian0', 3, '279-555-0130');
```

If we wanted to add two records, we can do this at the same time, and even use some different options for the different rows.

```
INSERT INTO SalesLT.CallLog
VALUES
(DATEADD(mi,-2, GETDATE()), 'adventure-works\jillian0', 4, '710-555-0173', NULL),
(DEFAULT, 'adventure-works\shu0', 5, '828-555-0186', 'Called to arrange deliver of order 10987');
```

There are more examples with adding query results into tables and getting identity values in the demo files.

Key Points * Use the INSERT statement to insert one or more rows into a table. * When inserting explicit values, you can omit identity columns, columns that allow NULLs, and columns on which a default constraint is defined. * Identity columns generate a unique integer identifier for each row. You can also use a sequence to generate unique values that can be used in multiple tables.

Bibliography