12TH ANNUAL

SedonaOffice® USERS CONFERENCE

MARCO ISLAND, FLORIDA
JANUARY 27 - 29, 2014

LEARN

NETWORK

ACHIEVE

SQL Quick Start 1 & 2

Presented By:

Jim Mayes

Table of Contents

PART 1

What is SQL? 4
How is Information Stored in SQL? 5
Is SQL a Foreign Language? 5
How do I Learn the Database and Table Names? 5
How do you ask questions? 6
Vocabulary 6
Grammar 7
How do I limit my results? 8
DATA 8
Types of Data 8
Ways to compare Data 9
Multiple Data Tests 10
Examples 11
Working with Dates and Times 12
An Aside: Functions 12
Date Functions 13
Examples *15*
Totaling results 17
Vocabulary 17
Examples 17
Having vs. Where 20
Where are we now? 21

PART 2

Case statements 22
Vocabulary 22
How Case Statements Work 22
Joins 25
What Are Joins? 26
How Do Joins Work? 26
Vocabulary 26
Using Inner Joins 27
Using Left Outer Joins 29
Using Right Outer Joins 31
Using Full Outer Joins 31
Working with the results from joined tables 33
Joining to more than one table 33
Using Distinct to Reduce Duplication 34

Writing Queries That Make Sense 37
Letting Others Know What You Were Thinking 38
When You Need Help 38
Appendix 39
Query to find table and column names 39
Query to find a specific Column 39
Query to find a Column or Table 39

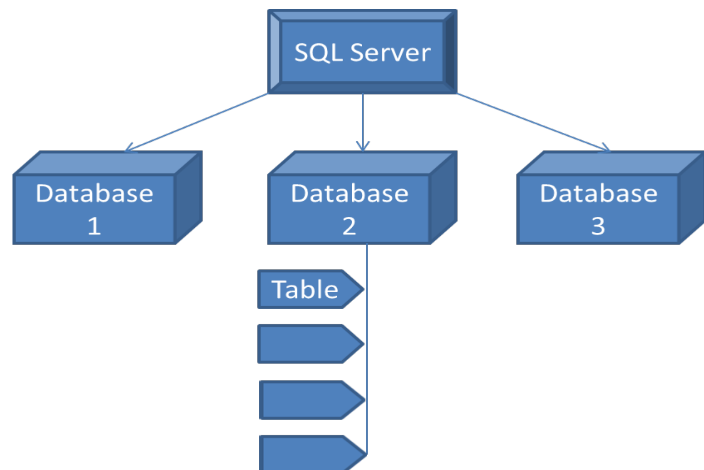
Session Overview

SQL Quick Start is designed for people who have little or no knowledge of SQL and want to gain a working knowledge in a short period of time. We will move fast and the student is not expected to understand everything immediately. The hope is that the student will use this class as a foundation and, in concert with these materials, their notes and the internet continue to develop their SQL skills independently.

Part 1

What is SQL?

SQL, Structured Query Language, is a means (script) to ask questions (queries) and get information (results, data) from a collection of information (database). This is no different than asking someone you know to list all of the colors that have blue in them or asking a baseball junkie to name all the players who have hit more than 500 homeruns! The trick is asking the right question and doing it in a language they understand. That is how we are going to approach SQL. This is nothing more than a foreign language and the database is a friend who knows a lot! I want to stress that this is not programming. Writing programs requires logical analysis, in-depth knowledge of the programming language, testing and hours of hard work. Getting information from an SQL (pronounced sea-quall) database can be simple. By the end of this session you will be able to create queries to ask the database lots of useful questions. You may not become a virtuoso, but you will be able to speak the right language!



How is Information Stored in SQL?

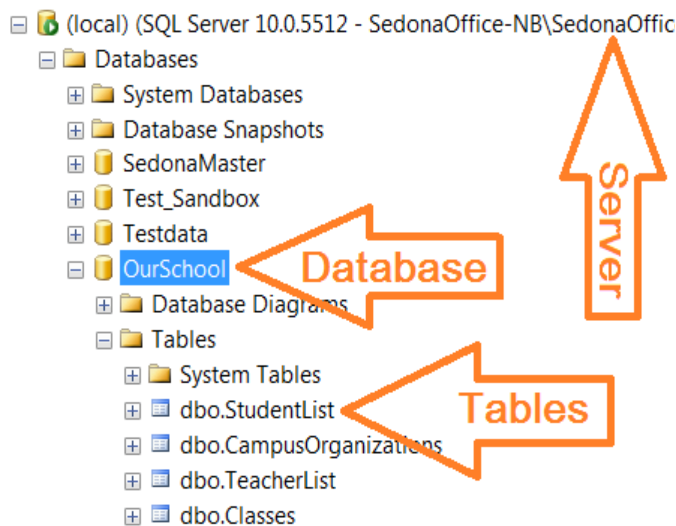
The server is a large repository of information divided into databases. Each database could represent a company, a school, etc. Within each database is a collection of tables, stored procedures, functions and views. For the purposes of this class, we will focus on the tables. Each table is further divided into columns and rows. The columns are the types of information being stored while the rows are the values of that information.

Is SQL a Foreign Language?

Every language consists of vocabulary and grammar. The same is true of SQL, once you learn the vocabulary and grammar, you will be able to query (question) the database (your know-it-all friend) as needed. The above descriptions are too "techie" for this course. We want terms and ideas that fit our learning model. So instead of Table, think filing cabinet and instead of Row, think File Folder.

How do I Learn the Database and Table Names?

This all depends on what method you will be using to question your databases. In the Appendix there are formatted questions (called scripts) that will ask the database to give you the database name, the tables in the database and even the columns in each table. You can copy these and paste them into whatever query window you are using. If you are using the SQL Management Studio you have the option of the following Hierarchical display:

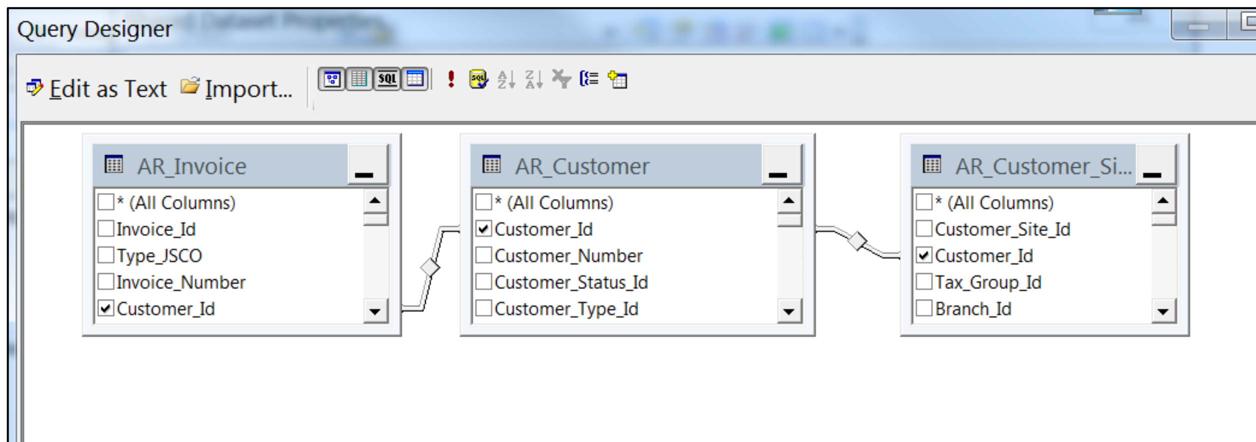


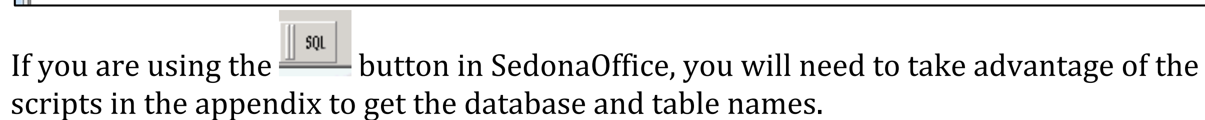
Columns

Rows

First Name	Last Name	Age	Sex
John	Smith	34	M
Nancy	Jones	28	F

If you are using an application like Reporting Services or Access you will have the option of a query designer (pictured below).



If you are using the  button in SedonaOffice, you will need to take advantage of the scripts in the appendix to get the database and table names.

How do you ask questions?

Let's start your vocabulary lesson with these words/symbols/terms:

Vocabulary

Select — *show me, give me, tell me*. Use **select** to ask the database for information. In any question you pose, **select** will be the first word of the question.

***** — {pronounced *star*} is the equivalent of saying all or everything.

From — I think this one translates directly. **From** means from!

Where — Many questions will require conditions to be met and those conditions will be stated using **where**. The best translation I can think of is "matching".

TableName — Inside of every database are the data tables. When you want to ask for information from a table, you need to use the table name as a part of the question.

ColumnName — Within each table are the columns that define what information is stored. If you want to ask for a specific piece of information from the table, you need to use the column name.

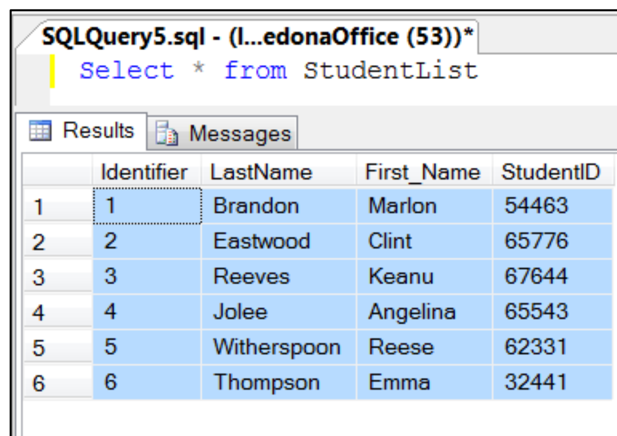
Grammar

Every question you ask will take the form:

Select *What you want to know* **From** *Table Name*

Example: **Select** * **from** StudentList

Translation: Give me everything from the Student List



	Identifier	LastName	First_Name	StudentID
1	1	Brandon	Marlon	54463
2	2	Eastwood	Clint	65776
3	3	Reeves	Keanu	67644
4	4	Jolee	Angelina	65543
5	5	Witherspoon	Reese	62331
6	6	Thompson	Emma	32441

The above results illustrate a neat trick. If you ever need to know the column names in a table, asking for everything provides the list quickly.

If you want to explicitly ask for one or more columns, you only need to replace ***** with the column name. In the case of more than one column you have to put a '**,**'(comma) between the column names. If the column name contains a space, you need to wrap the column name in **[]** to let SQL know that it is a single column name with spaces (see third example below).

Example: Give me the last names from the student list
Select LastName **from** StudentList

Give me the student ID's and last names from the student list
Select StudentID, LastName **from** StudentList

Give me a student listing with first and last name
Select [First Name], LastName **from** StudentList

SQLQuery5.sql - (l...edonaOffice (53))*

```
Select LastName from StudentList
```

Results | Messages

	LastName
1	Brandon
2	Eastwood
3	Reeves
4	Jolee
5	Witherspoon
6	Thompson

SQLQuery5.sql - (l...edonaOffice (53))*

```
Select StudentID, LastName from StudentList
```

Results | Messages

	StudentID	LastName
1	54463	Brandon
2	65776	Eastwood
3	67644	Reeves
4	65543	Jolee
5	62331	Witherspoon
6	32441	Thompson

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select [First Name], LastName from StudentList
```

Results | Messages

	First Name	LastName
1	Marlon	Brandon
2	Clint	Eastwood
3	Keanu	Reeves
4	Angelina	Jolee
5	Reese	Witherspoon
6	Emma	Thompson
7	Waylon	Jennings
8	Billy	King

How do I limit my results?

You limit your result set, by using the **where** clause to force conditions upon the returned data. Before we can discuss the **where** clause we must discuss the types of data and the ways in which you can compare and filter.

DATA

Types of Data

While there are plenty of sub-types of data, there are only a few basic types: Integer, Real, String, DateTime and Money.

Integers are whole numbers like 1,2,3,4,8,9

Reals are fractional numbers like 4.5, 3.2, 7.0, 8.9 {notice that 7.0 is the same as 7}

Strings are alpha-numeric expressions like *Smith, Green, D75GHY*. In SQL when you type a character string you must begin and end with the apostrophe character '. So you would refer to *this is a test string* by typing it as 'this is a test string'. (The size of the apostrophes has been increased to bring attention to them.)

Note: SQL is not case sensitive so 'SQl' is the same as 'sQl' is the same as 'sqL'

Beware the apostrophe trap! ' was copied directly from SQL Management Studio and ‘ was typed into MS Word. Notice that, even though they are both apostrophes, they look very different. The first example will work in SQL and the second one does not! Be careful when copying from other applications into SQL that all of your apostrophes match the first example. If you have problems running a copied query, check the apostrophes!
You can find out the length of a string by **Len** (stringname).

DateTime entries describe the date and/or time of an event and, like strings, must be encased in apostrophes. '09/24/2006', '09-24-2006', '09-24-2006 12:30:45'

Money is a special type of real number that is limited to two decimal places (like dollars and cents). This is normally a subtype of the real data type, but it is so widely used that it deserves a special mention: $13.45, $5.92, $1856.01.

Ways to compare Data

As we proceed through the material, you will be introduced to many of SQL's comparison operators. In the appendix there is a chart of all the operators and a brief description of how they work. Here are a few you may find useful.

= Equal compares two items of data and determines if they are identical. Note: comparing two string data types is not case sensitive so 'ThiS iS a StrInG' is the same as 'this is a string'!
> Greater than
< Less Than
<> Not Equal to
>= Greater than or equal
<= Less than or equal
>= Greater than or equal

Between fits the gap? checks to see if the value is greater than or equal to the first test value and less than or equal to the second value. Example: 10 **Between** 5 **and** 15

In one of these? you provide a list of possible values and if the test value matches one of these, the condition is true.
Example: 'Red' **in** ('Blue', 'Green', 'Red') is true and 5 **in** (1, 2, 3, 4) is false.

Like — **like** tries to compare two strings and determines if the test string matches the prospect string. There are wildcards that help the test process:

- **%** — joker, you can start and/or end the test string with **%** and the system will accept any matches with characters before/after the test string.
- **_** — (underline character) single character wild card.

Examples:

```
SQLQuery2.sql - (l...edonaOffice (56))*
Select [First Name], LastName
from StudentList
Where [First Name] Like '%on%'
```

	First Name	LastName
1	Marlon	Brandon
2	Waylon	Jennings
3	Leonard	DaVinci

```
SQLQuery2.sql - (l...edonaOffice (56))*   SQLQuery1.sql - (l...edonaOffice (53))*
Select [First Name], LastName
from StudentList
Where [First Name] Like '%on'
```

	First Name	LastName
1	Marlon	Brandon
2	Waylon	Jennings

```
SQLQuery2.sql - (l...edonaOffice (56))*   SQLQuery1.sql - (l...edonaOffice (53))*
Select [First Name], LastName
from StudentList
Where [First Name] Like 'Cl_nt'
```

	First Name	LastName
1	Clint	Eastwood

Multiple Data Tests

When we need to apply more than one test at the same time, we use the words:

And — When **And** is used, the conditions on either side must be true for the entire expression to be true.

5 > 3 **And** -3 < 0 is true
6 <> 12 **And** 'A' = 'B' is not true

Or — When **Or** is used, if either of the conditions is true then the entire expression is true.

6 <> 12 **Or** 'A' = 'B' is true
6 > 12 **Or** 'Smith' = 'Jones'

() — Parentheses are used to group the conditions for **And** and **Or**. This will force one condition to be applied before all others!

(6 + 5) * 3 = 11 * 3 = 33 6 + 5 * 3 = 6 + 15 = 21
((4 * 3) + 7) * 2 = (12 + 7) * 2 = 19 * 2 = 38 4 * 3 + 7 * 2 = 12 + 14 = 26

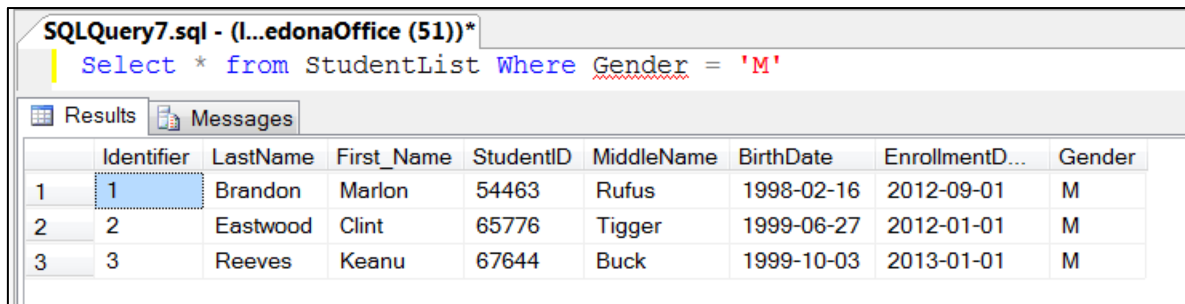
Examples

Now that we understand the types of data and ways to compare them let's try some!

Give me all students that are males.

Select * **from** StudentList **Where** Gender = 'M'

As before, the question starts with **Select** and includes * to get all of the columns, **from** and the table name to identify where to look for the information. What's new is the use of the **Where** clause to only bring in the men. The result set looks like this:


SQLQuery7.sql - (l...edonaOffice (51))*

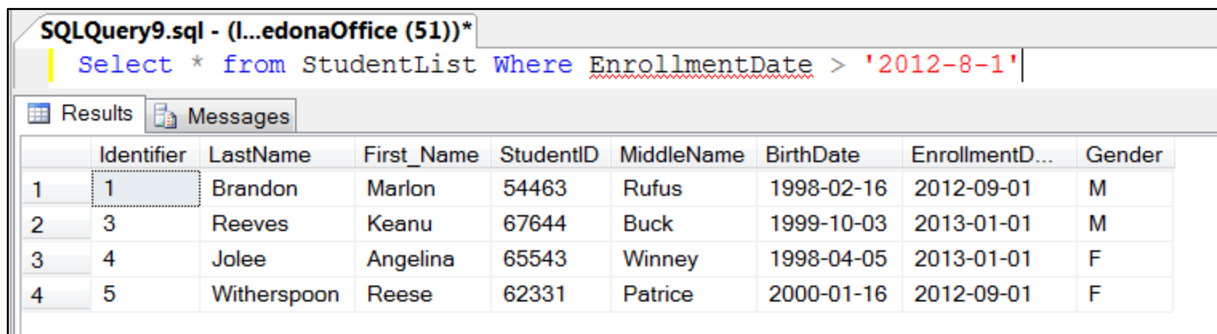
```
Select * from StudentList Where Gender = 'M'
```

Results | Messages

	Identifier	LastName	First_Name	StudentID	MiddleName	BirthDate	EnrollmentD...	Gender
1	1	Brandon	Marlon	54463	Rufus	1998-02-16	2012-09-01	M
2	2	Eastwood	Clint	65776	Tigger	1999-06-27	2012-01-01	M
3	3	Reeves	Keanu	67644	Buck	1999-10-03	2013-01-01	M

Give me all of the students who enrolled after August 1, 2012

Select * from StudentList **Where** EnrollmentDate > '2012-8-1'


SQLQuery9.sql - (l...edonaOffice (51))*

```
Select * from StudentList Where EnrollmentDate > '2012-8-1'
```

Results | Messages

	Identifier	LastName	First_Name	StudentID	MiddleName	BirthDate	EnrollmentD...	Gender
1	1	Brandon	Marlon	54463	Rufus	1998-02-16	2012-09-01	M
2	3	Reeves	Keanu	67644	Buck	1999-10-03	2013-01-01	M
3	4	Jolee	Angelina	65543	Winney	1998-04-05	2013-01-01	F
4	5	Witherspoon	Reese	62331	Patrice	2000-01-16	2012-09-01	F

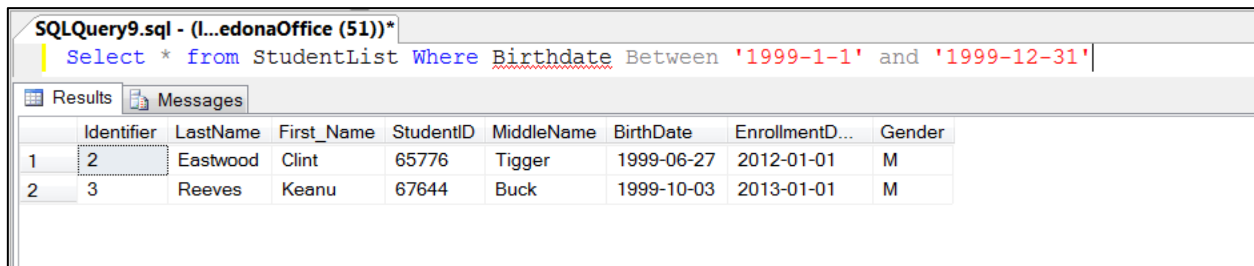
Show me all of the students who have a birthdate in the year 1999

Select * from StudentList **Where** Birthdate **Between** '1999-1-1' **and** '1999-12-31'

Here we will use **between** to cover the entire span of 1999. We can also use the expression:

Select * from StudentList **Where** Birthdate >= '1999-1-1' and Birthdate <= '1999-12-31'

to serve the same purpose. You can use either method although the first involves less typing!


SQLQuery9.sql - (l...edonaOffice (51))*

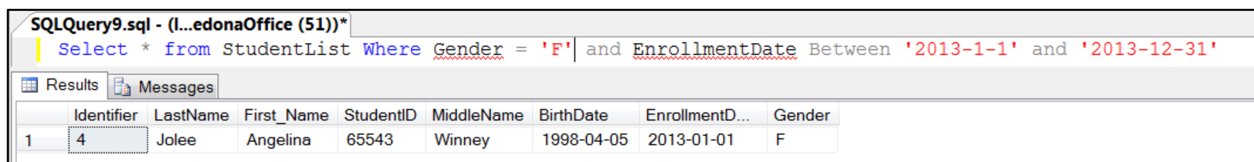
```
Select * from StudentList Where Birthdate Between '1999-1-1' and '1999-12-31'
```

Results | Messages

	Identifier	LastName	First_Name	StudentID	MiddleName	BirthDate	EnrollmentD...	Gender
1	2	Eastwood	Clint	65776	Tigger	1999-06-27	2012-01-01	M
2	3	Reeves	Keanu	67644	Buck	1999-10-03	2013-01-01	M

Show me all of the women students that enrolled in 2013

Select * from StudentList **Where** Gender = 'F' and EnrollmentDate **Between** '2013-1-1' **and** '2013-12-31'


SQLQuery9.sql - (l...edonaOffice (51))*

```
Select * from StudentList Where Gender = 'F' and EnrollmentDate Between '2013-1-1' and '2013-12-31'
```

Results | Messages

	Identifier	LastName	First_Name	StudentID	MiddleName	BirthDate	EnrollmentD...	Gender
1	4	Jolee	Angelina	65543	Winney	1998-04-05	2013-01-01	F

Working with Dates and Times

Date information can be the most challenging data to work with. You cannot use the same operators from other date types and there are lots of pitfalls.

3 + 2 = 5 → Easy

December 12, 2012 + 8 = ? → Not so easy!

When you use the datatype **DateTime** you type '12-31-2013' and the system sees this as '12/31/2013' at 00:00 (midnight). If you want to include the time, you must use '12-31-2013 00:00'. This is very important when comparing values! '12-7-2013' is not the same as '12/7/2013 11:30' and in an expression EnrollmentDate **between** '12-1-2013' **and** '12-7-2013', the value '12/7/2013 11:30' is not included!

An Aside: Functions

Functions are preloaded tools you can use to alter or compare data without having to explicitly make the calculation. Some functions come standard with SQL and others are created by users. If you keep learning about SQL you may someday create your own! Using a function is termed "calling" the function. The specific situation may change, but calling the function always takes the form:

CallThisFunction (value1, value2, value3…**)**

Note: *there can be no values*!

Functions will correspond to one of the available data types (int, money, NVarChar, etc.) and can only be used where that data type is compatible. If **CallThisFunction()** is an integer, then you can use it in the following example:

Select * from TestTable **where** CallThisFunction (DataValue) = 5

But you could not use it in the following:

Select * from TestTable **where** CallThisFunction (DataValue) = 'Bob'

The values inside of the parenthesis are called "parameters" and must agree with the data types expected for the specific parameters. If **CallThisFunction()** requires the parameters (name, Birth Date) then name must be a string type and Birth date must be a date type. (we are assuming that the birth date is a datetype and not a string representation of a birth date!)

Select * from TestTable **where** CallThisFunction ('Bob', '12-15-2013')

works, while

Select * from TestTable **where** CallThisFunction (1234, 'XYZ')

generates an error. Now that you understand the basics of how functions work, we can review some important date functions.

Date Functions

The following functions are for use with dates. These are the most important, but not the only ones available.

GetDate() returns the current date and time from the system. You must include the empty parenthesis!

What is the current date?
Select GetDate()

Show me all the future appointments?
Select * from AppointmentList **Where** AppointmentTime > GetDate()

DatePart (Part You Want, Date associated**)**

This returns a value for the portion of the date you are requesting.

Parts you want:
- Day = dd,
- Month = mm
- Year = yy
- Hour = hh
- minute = mi
- day of week = dw

Examples:

Select DatePart (dd, '12-17-2013') → 17
Select DatePart (mm, '12-17-2013') → 12
Select DatePart (dw, '12-17-2013') → 3

Tuesday is day 3 of the week. (Sunday = 1, Saturday = 7)

DateDiff (Part you want, First Date, Second Date**)**

This returns the value of the difference between first date and second date based on the comparison you asked for.

Examples:

Select DateDiff (dd, '12-17-2013', '12-24-2013') → 7
Select DateDiff (mm, '11-12-2013', '12-17-2013') → 1

Note: number returned is complete months, so 35 days = 1 month!

DateAdd (Part you want, number, Date**)**

This function returns a new date which is the value of date modified by the number of part selected. Note: number can be positive or negative!

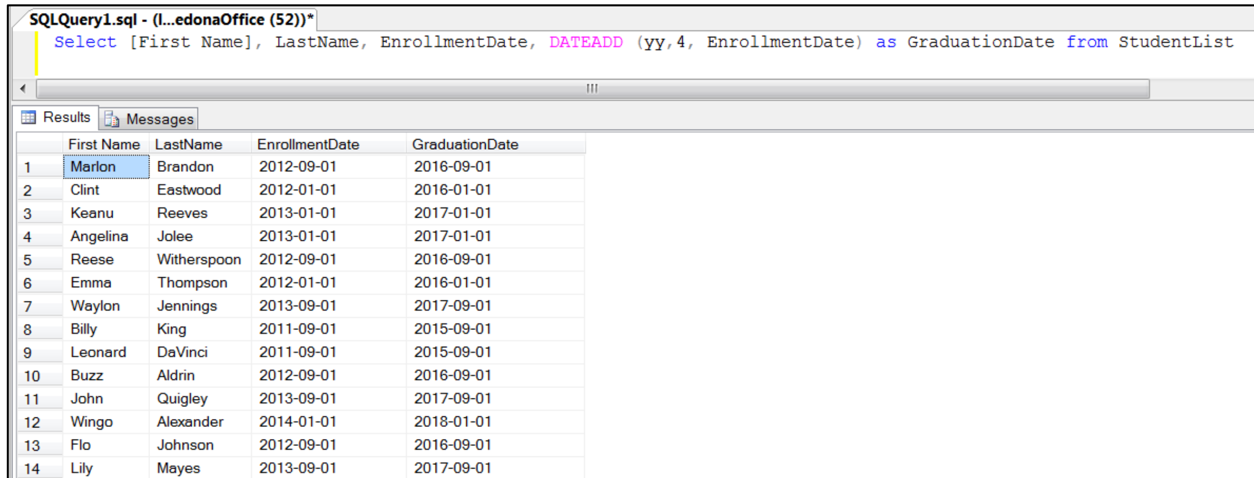
Examples:

Select DateAdd (dd, 5, '12-17-2013') → '12-22-2013'
Select DateAdd (dd, -5, '12-17-2013') → '12-12-2013'

Examples

Now that you know about these date functions, let's ask some questions. The above examples used what are called "explicit" dates. The date is typed in and the value is obvious and unchanging. The functions operate the same if you replace the explicit value with a column name.

Show me the anticipated graduation date for all of the students
(Note: we will assume that each program normally takes four years)


SQLQuery1.sql - (l...edonaOffice (52))*

```
Select [First Name], LastName, EnrollmentDate, DATEADD (yy,4, EnrollmentDate) as GraduationDate from StudentList
```

Results | Messages

	First Name	LastName	EnrollmentDate	GraduationDate
1	Marlon	Brandon	2012-09-01	2016-09-01
2	Clint	Eastwood	2012-01-01	2016-01-01
3	Keanu	Reeves	2013-01-01	2017-01-01
4	Angelina	Jolee	2013-01-01	2017-01-01
5	Reese	Witherspoon	2012-09-01	2016-09-01
6	Emma	Thompson	2012-01-01	2016-01-01
7	Waylon	Jennings	2013-09-01	2017-09-01
8	Billy	King	2011-09-01	2015-09-01
9	Leonard	DaVinci	2011-09-01	2015-09-01
10	Buzz	Aldrin	2012-09-01	2016-09-01
11	John	Quigley	2013-09-01	2017-09-01
12	Wingo	Alexander	2014-01-01	2018-01-01
13	Flo	Johnson	2012-09-01	2016-09-01
14	Lily	Mayes	2013-09-01	2017-09-01

Using the word **AS**. **AS** = You can call me…
You can change the name of a column in a result set on the fly by using the word **AS**. Whatever you follow **AS** with becomes the new name of the column!

If you think the above example isn't quite right, we will revisit this example later and get it 100%!

Show me how old each student is and how old they were when they enrolled in school

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select [First Name], LastName, BirthDate, EnrollmentDate,
DATEDIFF (yy,BirthDate, GetDate()) as CurrentAge,
DATEDIFF (yy,BirthDate, EnrollmentDate) as EnrollmentAge
From StudentList
```

Results | Messages

	First Name	LastName	BirthDate	EnrollmentD...	CurrentAge	EnrollmentAge
1	Marlon	Brandon	1994-04-12	2012-09-01	20	18
2	Clint	Eastwood	1993-01-17	2012-01-01	21	19
3	Keanu	Reeves	1995-07-03	2013-01-01	19	18
4	Angelina	Jolee	1995-08-24	2013-01-01	19	18
5	Reese	Witherspoon	1994-09-16	2012-09-01	20	18
6	Emma	Thompson	1994-08-11	2012-01-01	20	18
7	Waylon	Jennings	1994-10-12	2013-09-01	20	19
8	Billy	King	1993-02-18	2011-09-01	21	18
9	Leonard	DaVinci	1994-07-30	2011-09-01	20	17
10	Buzz	Aldrin	1994-12-06	2012-09-01	20	18
11	John	Quigley	1995-12-08	2013-09-01	19	18
12	Wingo	Alexander	1996-04-23	2014-01-01	18	18
13	Flo	Johnson	1994-07-14	2012-09-01	20	18
14	Lily	Mayes	1996-03-22	2013-09-01	18	17

Show me all of the students who were Fall enrollments

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select [First Name], LastName, EnrollmentDate
From StudentList
Where DATEPART (mm,EnrollmentDate) = 9
```

Results | Messages

	First Name	LastName	EnrollmentDate
1	Marlon	Brandon	2012-09-01
2	Reese	Witherspoon	2012-09-01
3	Waylon	Jennings	2013-09-01
4	Billy	King	2011-09-01
5	Leonard	DaVinci	2011-09-01
6	Buzz	Aldrin	2012-09-01
7	John	Quigley	2013-09-01
8	Flo	Johnson	2012-09-01
9	Lily	Mayes	2013-09-01

Totaling results

So far we have used queries to generate lists of information. What if you want to total a column or count the number of members in a listing? These operations are termed "aggregate" functions.

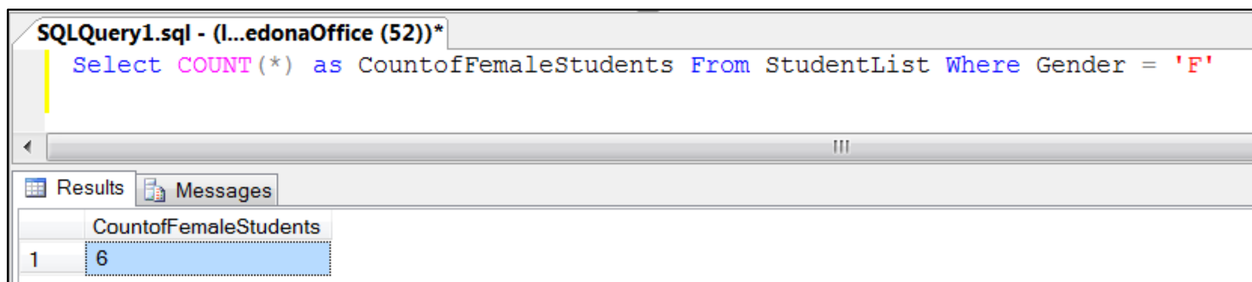
Vocabulary

Sum (*column*) total the column, this will sum all of the values in the result set. The value for *column* must be numeric.
Count (*) how many are there, this will give you the number of results returned.
Avg calculates the average
Group By gather, this chooses which columns you want to aggregate based on
Having matching, this is the equivalent of the where statement for aggregates. If you need to force conditions on an aggregate, you do it with *having*.

Examples

How many students are women?

We could list all of the female students and then count the number of rows or we could do this:

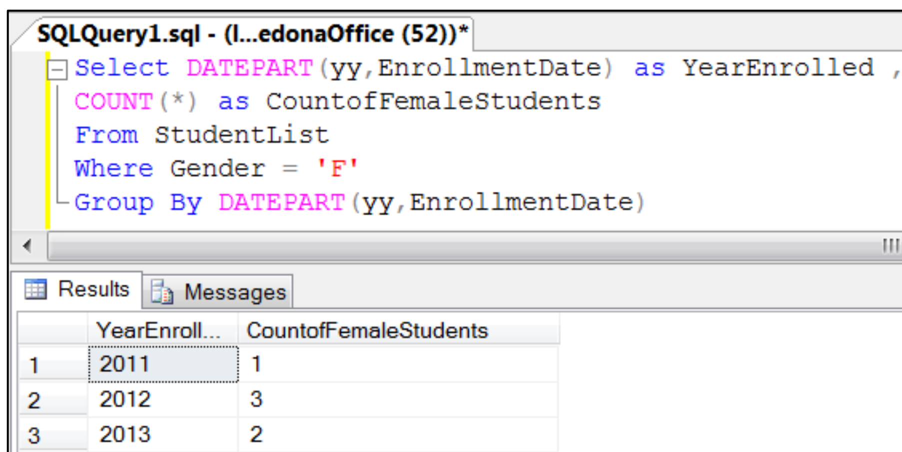

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select COUNT(*) as CountofFemaleStudents From StudentList Where Gender = 'F'
```

Results | Messages

	CountofFemaleStudents
1	6

How many female students enroll each year?


SQLQuery1.sql - (l...edonaOffice (52))*

```
Select DATEPART(yy,EnrollmentDate) as YearEnrolled ,
COUNT(*) as CountofFemaleStudents
From StudentList
Where Gender = 'F'
Group By DATEPART(yy,EnrollmentDate)
```

Results | Messages

	YearEnroll...	CountofFemaleStudents
1	2011	1
2	2012	3
3	2013	2

The **Group By** statement collects the results into each year. You do not have to include the year as a part of the select statement, but you cannot include other information unless it is in the **Group By** statement or is an aggregate function! In the following,

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select EnrollmentDate, COUNT(*) as CountofFemaleStudents
From StudentList
Where Gender = 'F'
Group By DATEPART(yy,EnrollmentDate)
```

Messages

```
Msg 8120, Level 16, State 1, Line 1
Column 'StudentList.EnrollmentDate' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.
```

we have added the EnrollmentDate and receive the error indicated. Even though EnrollmentDate is a part of the ***DatePart*** function, it does not count as being a part of the **Group By**. We can correct this by adding EnrollmentDate to the **Group By** clause, but that changes the results and fails to answer the original question.

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select EnrollmentDate, COUNT(*) as CountofFemaleStudents
From StudentList
Where Gender = 'F'
Group By DATEPART(yy,EnrollmentDate), EnrollmentDate
```

Results | Messages

	EnrollmentDate	CountofFemaleStude...
1	2011-09-01	1
2	2012-01-01	1
3	2012-09-01	2
4	2013-01-01	1
5	2013-09-01	1

What is the total tuition for our school?

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select SUM(Tuition) as TuitionAmount from StudentList
```

Results | Messages

	TuitionAmount
1	67525.00

What is the tuition for our school by gender?

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select Gender, SUM(Tuition) as TuitionAmount
from StudentList
Group By Gender
```

Results | Messages

	Gender	TuitionAmount
1	F	29800.00
2	M	37725.00

What is our tuition based on the year and semester the students were enrolled?

SQLQuery2.sql - (l...SedonaOffice (53)) | SQLQuery1.sql - (l...edonaOffice (52))*

```
Select DATEPART(yy,EnrollmentDate),DATEPART(MM,EnrollmentDate),
SUM(Tuition) as TuitionAmount
from StudentList
Group By DATEPART(yy,EnrollmentDate), DATEPART(MM,EnrollmentDate)
```

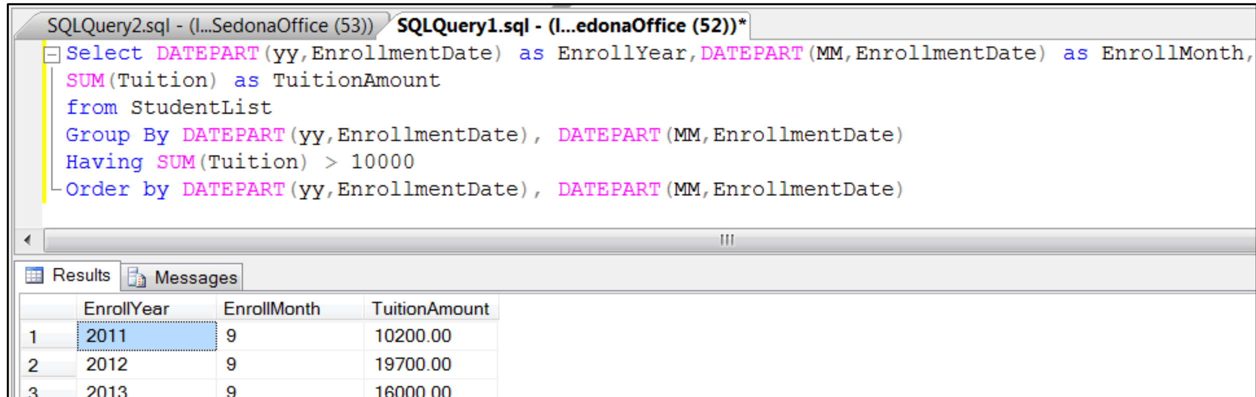
Results | Messages

	(No column nam...	(No column na...	TuitionAmount
1	2011	9	10200.00
2	2012	1	8300.00
3	2012	9	19700.00
4	2013	1	8775.00
5	2013	9	16000.00
6	2014	1	4550.00

Note: When you do not use **AS** to name a column for an aggregate or a function, the system uses “No column name”. If you want to avoid confusion always name the column!

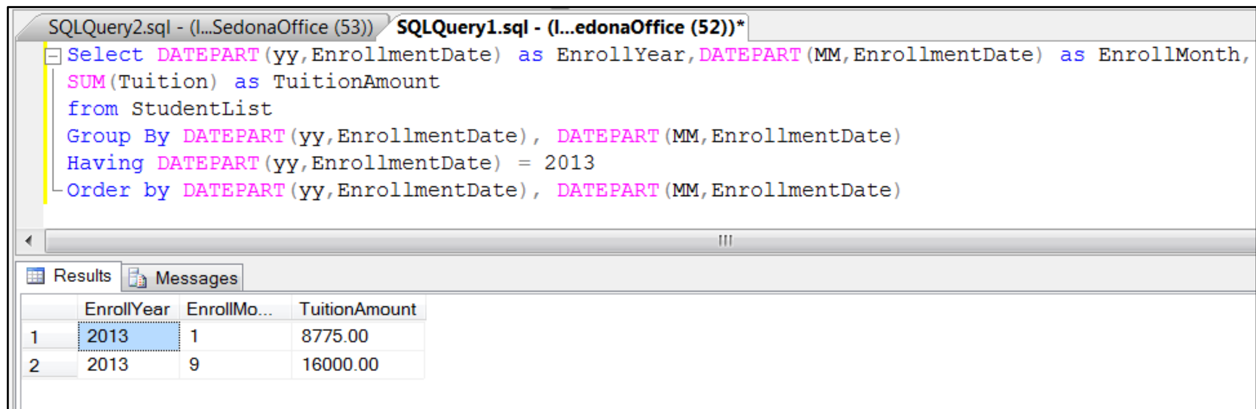
Sometimes you are looking to focus on a particular set of data.

Which semesters and years account for at least $10,000 in tuition?



Having vs. Where

The **Having** clause operates identical to a **Where** clause when applied to aggregates. You can also use **Having** to filter other parts of the query.



In the above statement, the **Having** clause is used to filter the year. You could have used a **where** clause to do exactly the same thing.

```
Select DATEPART(yy,EnrollmentDate) as EnrollYear,DATEPART(MM,EnrollmentDate) as EnrollMonth,
SUM(Tuition) as TuitionAmount
from StudentList
Where DATEPART(yy,EnrollmentDate) = 2013
Group By DATEPART(yy,EnrollmentDate), DATEPART(MM,EnrollmentDate)
Order by DATEPART(yy,EnrollmentDate), DATEPART(MM,EnrollmentDate)
```

	EnrollYear	EnrollMo...	TuitionAmount
1	2013	1	8775.00
2	2013	9	16000.00

Where and **Having** are not completely interchangeable! You cannot include something in the **Having** clause if it is not included in the **Group By** list. You cannot include an aggregate in the **Where** clause. So, while they work similarly and sometimes overlap, you should keep the prior two constraints in mind when deciding which to use.

What is the average tuition of students by enrollment year?

```
Select DATEPART(yy,EnrollmentDate) as EnrollYear,
AVG(Tuition) as AverageTuition
from StudentList
Group By DATEPART(yy,EnrollmentDate) --, DATEPART(MM,EnrollmentDate)
Order by DATEPART(yy,EnrollmentDate) --, DATEPART(MM,EnrollmentDate)
```

	EnrollYear	TuitionAmount
1	2011	5100.00
2	2012	4666.6666
3	2013	4955.00
4	2014	4550.00

Where are we now?

Given a single table, you can list all of the members, sort and filter them as well as aggregate the numeric values to analyze data. This provides a lot of functionality and can fulfill many of your reporting needs! In Part 2 we will examine complex comparisons, merge information from multiple tables and discuss query style and clarity. Yeah!

Part 2

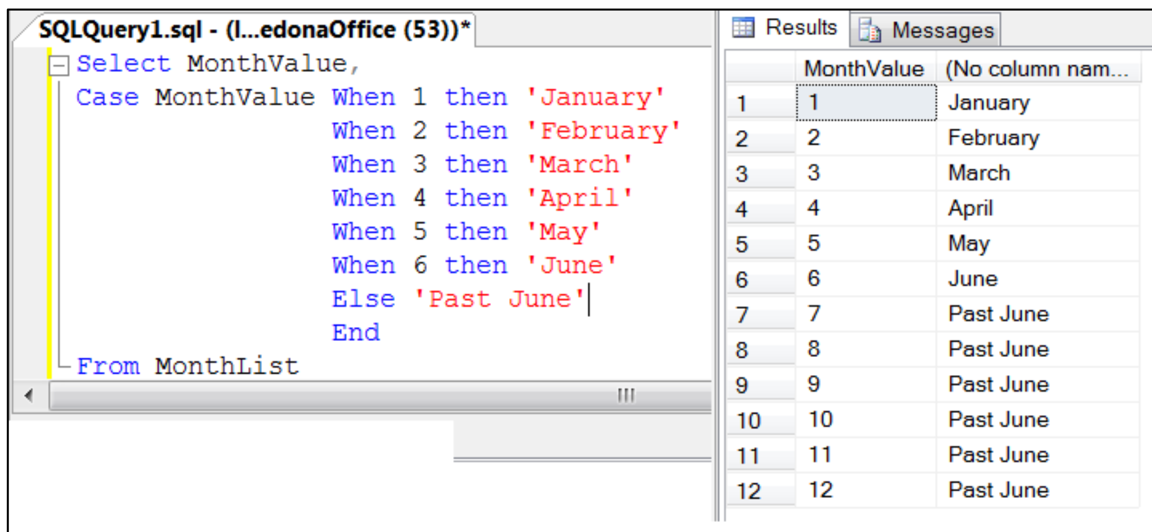
Case statements

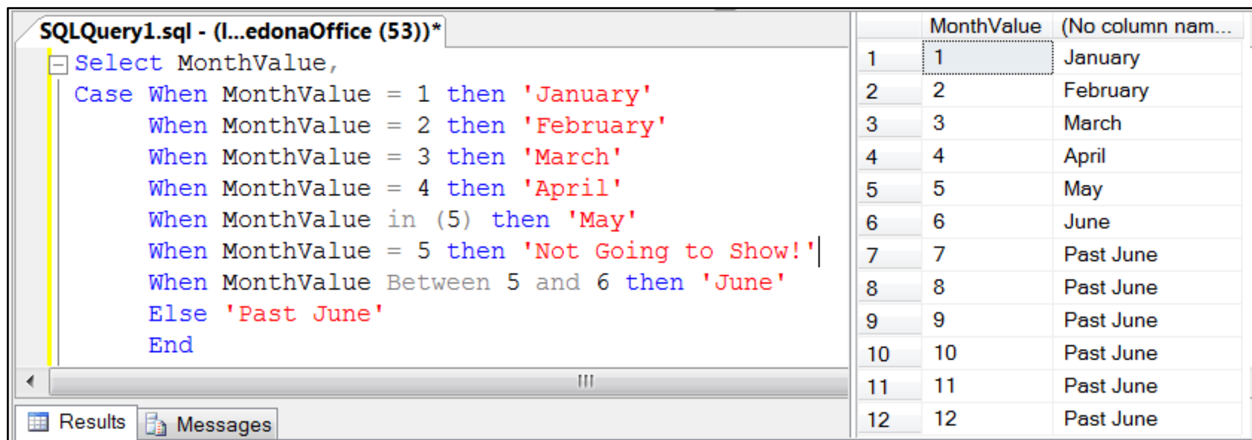
Vocabulary

Case match one of these, case statements provide functionality for comparing a value to other values and returning a result.
When match this one, each condition of a case statement starts with when
Then use this one, then indicates what value the case statement will return if the condition is met
Else catch all, in a case statement if no conditions are met the else is chosen
End we are done, lets SQL know that the case statement is finished

How Case Statements Work

Case lets you test multiple conditions and replace results to suit your needs. As an example, you may have a column that stores an integer value for month, but you don't want to report the month as 1..12. You can use **case** to substitute the months name for the column value.





The grammar for **Case** can take two forms, each illustrated above. In the first form, you start with **Case** followed by the value to test and then the conditions to test on (**When**) and the result of the test (**Then**). Finally you have the option of an **Else** statement and the term **End** to let SQL know that you are done. **Else** is optional but I advise using this just in case unexpected data creeps in. The second style is nearly identical to the first, however you skip the initial value entry and place a specific test after each of the **When** clauses. The first style is easier to type in and the second style is more flexible. You will need to decide for yourself which works best for your situation.

Things to notice:

1) You can use nearly any expression as a test condition
2) The tests are applied sequentially. In the second example 5 is valid for three of the conditions yet only results in the first condition being applied.
3) The results of the tests can be any data type, however must all be the **same** data type throughout a specific case statement.

Nested Case Statements
You can include a **case** statement inside of another **case** statement. This will not be covered here, but you can experiment on your own!

Case statements are valid almost anywhere that you can use a data value!

SQLQuery1.sql - (l...edonaOffice (53))*

```
Select *
from StudentList
Where DormAssignment = Case Gender When 'F' then 2 else 3 end
```

Results | Messages

	Identifier	LastName	First Name	StudentID	MiddleName	BirthDate	EnrollmentD...	Gend...	Tuition	DormAssignment
1	3	Reeves	Keanu	67644	Buck	1995-07-03	2013-01-01	M	4025.00	3
2	6	Thompson	Emma	32441	Gina	1994-08-11	2012-01-01	F	3800.00	2
3	7	Jennings	Waylon	67347	Bob	1994-10-12	2013-09-01	M	5500.00	3
4	11	Quigley	John	56778	Jay	1995-12-08	2013-09-01	M	4300.00	3
5	14	Mayes	Lily	66543	CD	1996-03-22	2013-09-01	F	6200.00	2

We now revisit the query we wrote to list the student's anticipated graduation dates.

SQLQuery1.sql - (l...edonaOffice (53))*

```
Select [First Name], LastName, EnrollmentDate,
DATEADD(yy,4,EnrollmentDate) as OldGraduationDate,
Case DATEPART(mm,EnrollmentDate)
     When 9 then DATEADD (dd,19, DATEADD(mm,-4, DATEADD(YY,4,EnrollmentDate)))
     When 1 then DATEADD (dd,19, DATEADD(mm,-1, DATEADD(YY,4,EnrollmentDate)))
     Else DATEADD(YY,4,EnrollmentDate)
     End as BetterGraduationDate
from StudentList
```

Results | Messages

	First Name	LastName	Enrollment...	OldGraduationDate	BetterGraduationDate
1	Marlon	Brandon	2012-09-01	2016-09-01	2016-05-20
2	Clint	Eastwood	2012-01-01	2016-01-01	2015-12-20
3	Keanu	Reeves	2013-01-01	2017-01-01	2016-12-20
4	Angelina	Jolee	2013-01-01	2017-01-01	2016-12-20
5	Reese	Witherspoon	2012-09-01	2016-09-01	2016-05-20
6	Emma	Thompson	2012-01-01	2016-01-01	2015-12-20
7	Waylon	Jennings	2013-09-01	2017-09-01	2017-05-20
8	Billy	King	2011-09-01	2015-09-01	2015-05-20
9	Leonard	DaVinci	2011-09-01	2015-09-01	2015-05-20
10	Buzz	Aldrin	2012-09-01	2016-09-01	2016-05-20
11	John	Quigley	2013-09-01	2017-09-01	2017-05-20
12	Wingo	Alexander	2014-01-01	2018-01-01	2017-12-20
13	Flo	Johnson	2012-09-01	2016-09-01	2016-05-20
14	Lily	Mayes	2013-09-01	2017-09-01	2017-05-20

Joins

Table Design

Well-designed databases use multiple tables to store information. This is done for both clarity and efficiency! If you were going to create a database of students, one of the pieces of data you might track is the dorm that they live in, including the address of that dorm. For each student entry you could explicitly list the dorm name, the street address, city, state and zip code OR you could create a separate table for the dorm information and then link the two tables!

Consider the following:

SQLQuery1.sql - (l...edonaOffice (52))*

```
select *
from StudentListWasteful
```

Results | Messages

	Identifier	LastName	First Name	StudentID	MiddleName	BirthDate	EnrollmentD...	Gend...	Tuition	DormName	DormStreetAddress	DormCity	DormState	DormZip
1	1	Brandon	Marlon	54463	Rufus	1994-04-12	2012-09-01	M	5600.00	East	201 East Street	Our City	MI	48197
2	2	Eastwood	Clint	65776	Tigger	1993-01-17	2012-01-01	M	4500.00	West	101 West Street	Our City	MI	48197
3	3	Reeves	Keanu	67644	Buck	1995-07-03	2013-01-01	M	4025.00	North	301 North Street	Our City	MI	48197
4	4	Jolee	Angelina	65543	Winney	1995-08-24	2013-01-01	F	4750.00	South	401 South Street	Our City	MI	48197
5	5	Witherspoon	Reese	62331	Patrice	1994-09-16	2012-09-01	F	4750.00	East	201 East Street	Our City	MI	48197
6	6	Thompson	Emma	32441	Gina	1994-08-11	2012-01-01	F	3800.00	West	101 West Street	Our City	MI	48197
7	7	Jennings	Waylon	67347	Bob	1994-10-12	2013-09-01	M	5500.00	North	301 North Street	Our City	MI	48197
8	8	King	Billy	86554	Jean	1993-02-18	2011-09-01	F	5200.00	South	401 South Street	Our City	MI	48197
9	9	DaVinci	Leonard	78655	Xavier	1994-07-30	2011-09-01	M	5000.00	East	201 East Street	Our City	MI	48197
10	10	Aldrin	Buzz	44565	Thomas	1994-12-06	2012-09-01	M	4250.00	West	101 West Street	Our City	MI	48197
11	11	Quigley	John	56778	Jay	1995-12-08	2013-09-01	M	4300.00	North	301 North Street	Our City	MI	48197
12	12	Alexander	Wingo	58103	Wally	1996-04-23	2014-01-01	M	4550.00	South	401 South Street	Our City	MI	48197
13	13	Johnson	Flo	51244	Jo	1994-07-14	2012-09-01	F	5100.00	East	201 East Street	Our City	MI	48197
14	14	Mayes	Lily	66543	CD	1996-03-22	2013-09-01	F	6200.00	West	101 West Street	Our City	MI	48197

Compared to

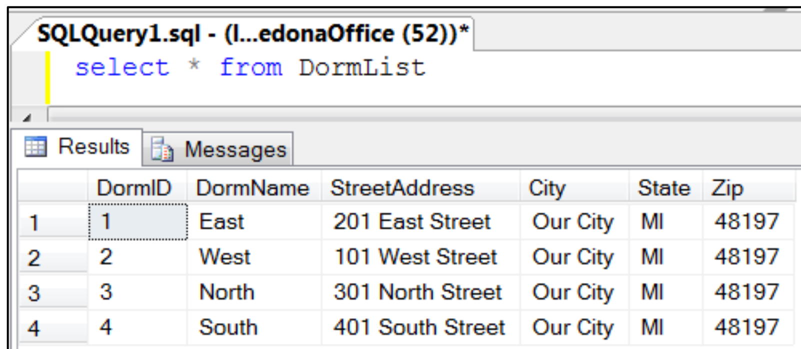
SQLQuery1.sql - (l...edonaOffice (52))*

```
select * from StudentList
```

Results | Messages

	Identifier	LastName	First Name	StudentID	MiddleName	BirthDate	EnrollmentD...	Gend...	Tuition	DormAssignment
1	1	Brandon	Marlon	54463	Rufus	1994-04-12	2012-09-01	M	5600.00	1
2	2	Eastwood	Clint	65776	Tigger	1993-01-17	2012-01-01	M	4500.00	2
3	3	Reeves	Keanu	67644	Buck	1995-07-03	2013-01-01	M	4025.00	3
4	4	Jolee	Angelina	65543	Winney	1995-08-24	2013-01-01	F	4750.00	4
5	5	Witherspoon	Reese	62331	Patrice	1994-09-16	2012-09-01	F	4750.00	1
6	6	Thompson	Emma	32441	Gina	1994-08-11	2012-01-01	F	3800.00	2
7	7	Jennings	Waylon	67347	Bob	1994-10-12	2013-09-01	M	5500.00	3
8	8	King	Billy	86554	Jean	1993-02-18	2011-09-01	F	5200.00	4
9	9	DaVinci	Leonard	78655	Xavier	1994-07-30	2011-09-01	M	5000.00	1
10	10	Aldrin	Buzz	44565	Thomas	1994-12-06	2012-09-01	M	4250.00	2
11	11	Quigley	John	56778	Jay	1995-12-08	2013-09-01	M	4300.00	3
12	12	Alexander	Wingo	58103	Wally	1996-04-23	2014-01-01	M	4550.00	4
13	13	Johnson	Flo	51244	Jo	1994-07-14	2012-09-01	F	5100.00	1
14	14	Mayes	Lily	66543	CD	1996-03-22	2013-09-01	F	6200.00	2

Combined with



SQLQuery1.sql - (l...edonaOffice (52))*

```
select * from DormList
```

Results | Messages

	DormID	DormName	StreetAddress	City	State	Zip
1	1	East	201 East Street	Our City	MI	48197
2	2	West	101 West Street	Our City	MI	48197
3	3	North	301 North Street	Our City	MI	48197
4	4	South	401 South Street	Our City	MI	48197

The second method is more efficient in terms of space savings and performance. The database designer could take this one step further by creating a new table to store the City/State/Zip and linking that back to the DormList. Now need tools to generate queries based on multiple tables: **Joins**.

What Are Joins?

Joins allow you to merge the information in one or more tables based on linkage that you define. In the previous example, the linkage would be the DormAssignment connecting to the DormID value. Database designers try to use column names that will indicate this linkage (DormAssignment = DormID) but there is no hard and fast rule so always confirm your linking!

How Do Joins Work?

There are four types of **joins**: **Inner**, **Left Outer**, **Right Outer** and **Full Outer**. These are not intuitive names or ideas, so we need to illustrate these relationships. Don't worry if you don't understand right away, the **join** concepts are probably the most difficult in SQL!

Vocabulary

Inner Join pair up, results will only return when the members of both tables match the criteria. There is no restriction that the matches are one to one, so you can see results that include the same elements of either table matched to multiples of the other table.

Outer Join pair up when you can, these can occur as Left, Right or Full. There is a convention that the table that is written first is on the Left side of the screen and the table written second is on the Right side of the screen. The result set will include the same entries as the inner join would PLUS the members of the Left or Right table (as indicated) will always return once. On a Full join the results are as if you returned the results of the Inner AND the extra entries from the Left AND Right joins.

ON matching, this is similar to the where clause and the having clause. Whatever condition is listed after **ON** will be used to match the two tables involved in the join.

Using Inner Joins

An **inner join** compares two tables and only reports back elements that match on the criteria chosen. Consider the two tables

SQLQuery1.sql - (l...edonaOffice (52))*

```
Select * from Children
```

Results | Messages

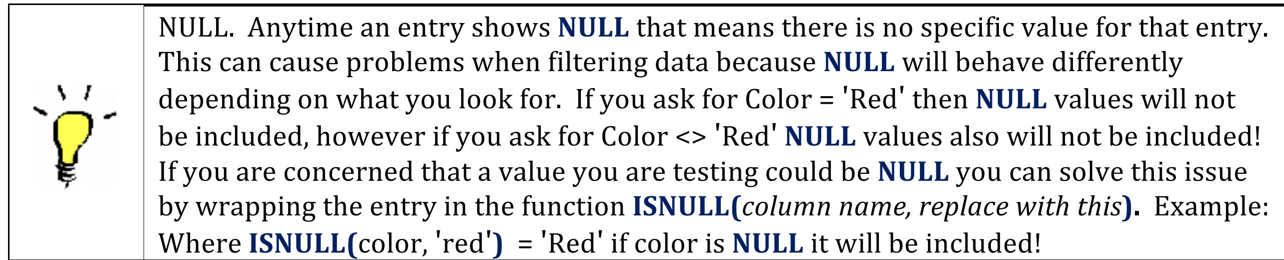
	Child	FavoriteColor	FavoriteShape	LikesGames	Age	LikesLearning
1	Bobby	Red	Square	Y	10	N
2	Cindy	Blue	Round	Y	8	Y
3	Lily	Yellow	Flat	N	6	N
4	Tammy	Green	Square	N	7	N
5	Robby	Blue	Round	Y	12	N
6	Mikey	Orange	Square	Y	10	Y
7	Dennis	Blue	Flat	N	8	Y
8	Wendy	Red	Square	N	9	N
9	Oscar	Yellow	Round	Y	8	N
10	Beth	Green	Round	N	4	Y

SQLQuery2.sql - (l...edonaOffice (53))* SQLQuery1.sql - (l...edonaOffice (52))*

```
Select * from Toys
```

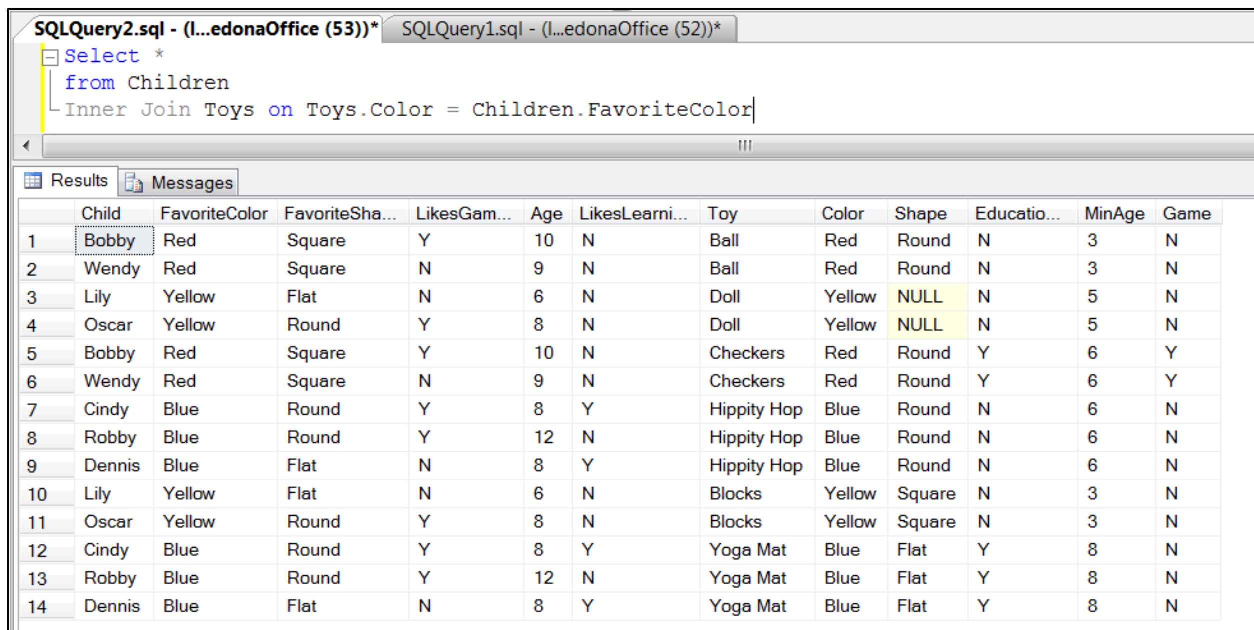
Results | Messages

	Toy	Color	Shape	Educatio...	MinAge	Game
1	Ball	Red	Round	N	3	N
2	Jacks	NULL	Round	N	5	Y
3	Uno	NULL	NULL	N	6	Y
4	Doll	Yellow	NULL	N	5	N
5	Checkers	Red	Round	Y	6	Y
6	Chess	NULL	NULL	Y	9	y
7	Hippity Hop	Blue	Round	N	6	N
8	Blocks	Yellow	Square	N	3	N
9	Legos	NULL	Square	N	6	N
10	Chutes and Ladders	NULL	Flat	N	4	Y
11	Yoga Mat	Blue	Flat	Y	8	N
12	Barney Doll	Purple	Dinosaur	Y	6	N


NULL. Anytime an entry shows **NULL** that means there is no specific value for that entry. This can cause problems when filtering data because **NULL** will behave differently depending on what you look for. If you ask for Color = 'Red' then **NULL** values will not be included, however if you ask for Color <> 'Red' **NULL** values also will not be included! If you are concerned that a value you are testing could be **NULL** you can solve this issue by wrapping the entry in the function **ISNULL(***column name, replace with this***)**. Example: Where **ISNULL(**color, 'red'**)** = 'Red' if color is **NULL** it will be included!

The grammar for an **inner join** is:

Give me the children and toys that match favorite colors to toy colors


SQLQuery2.sql - (l...edonaOffice (53))* | SQLQuery1.sql - (l...edonaOffice (52))*

```
Select *
from Children
Inner Join Toys on Toys.Color = Children.FavoriteColor
```

Results | Messages

	Child	FavoriteColor	FavoriteSha...	LikesGam...	Age	LikesLearni...	Toy	Color	Shape	Educatio...	MinAge	Game
1	Bobby	Red	Square	Y	10	N	Ball	Red	Round	N	3	N
2	Wendy	Red	Square	N	9	N	Ball	Red	Round	N	3	N
3	Lily	Yellow	Flat	N	6	N	Doll	Yellow	NULL	N	5	N
4	Oscar	Yellow	Round	Y	8	N	Doll	Yellow	NULL	N	5	N
5	Bobby	Red	Square	Y	10	N	Checkers	Red	Round	Y	6	Y
6	Wendy	Red	Square	N	9	N	Checkers	Red	Round	Y	6	Y
7	Cindy	Blue	Round	Y	8	Y	Hippity Hop	Blue	Round	N	6	N
8	Robby	Blue	Round	Y	12	N	Hippity Hop	Blue	Round	N	6	N
9	Dennis	Blue	Flat	N	8	Y	Hippity Hop	Blue	Round	N	6	N
10	Lily	Yellow	Flat	N	6	N	Blocks	Yellow	Square	N	3	N
11	Oscar	Yellow	Round	Y	8	N	Blocks	Yellow	Square	N	3	N
12	Cindy	Blue	Round	Y	8	Y	Yoga Mat	Blue	Flat	Y	8	N
13	Robby	Blue	Round	Y	12	N	Yoga Mat	Blue	Flat	Y	8	N
14	Dennis	Blue	Flat	N	8	Y	Yoga Mat	Blue	Flat	Y	8	N

Things to notice:

1) The columns for both tables are included with the columns from Children first and Toys second
2) Some kids are not included (Tammy, Mikey and Beth) because none of the toys match their favorite colors
3) Some of the games are not included because they do not match a child's favorite color.
4) We differentiate between the elements of one table from the other by using TableName.ColumnName.
5) Some children are in the result set more than once because more than one toy matches their favorite color.
6) Some toys are in the result set more than once because more than one child likes the color they are.

You can enforce multiple conditions after the **ON** and these conditions are not required to involve both tables.

SQLQuery2.sql - (l...edonaOffice (53))*

```
Select *
from Children
Inner Join Toys on Toys.Color = Children.FavoriteColor and Educational = 'N'
Order by Children.Child
```

Results | Messages

	Child	FavoriteColor	FavoriteShape	LikesGames	Age	LikesLearning	Toy	Color	Shape	Educational	MinAge	Game
1	Bobby	Red	Square	Y	10	N	Ball	Red	Round	N	3	N
2	Cindy	Blue	Round	Y	8	Y	Hippity Hop	Blue	Round	N	6	N
3	Dennis	Blue	Flat	N	8	Y	Hippity Hop	Blue	Round	N	6	N
4	Lily	Yellow	Flat	N	6	N	Doll	Yellow	NULL	N	5	N
5	Lily	Yellow	Flat	N	6	N	Blocks	Yellow	Square	N	3	N
6	Oscar	Yellow	Round	Y	8	N	Doll	Yellow	NULL	N	5	N
7	Oscar	Yellow	Round	Y	8	N	Blocks	Yellow	Square	N	3	N
8	Robby	Blue	Round	Y	12	N	Hippity Hop	Blue	Round	N	6	N
9	Wendy	Red	Square	N	9	N	Ball	Red	Round	N	3	N

Using Left Outer Joins

A **left outer join** will report all of the members from the Left table (first table in the expression) matched to the Right Table (second table in the expression) when possible. Again, these are not one to one matches so the entries in each table can appear in the results multiple times.

SQLQuery2.sql - (l...edonaOffice (53))*

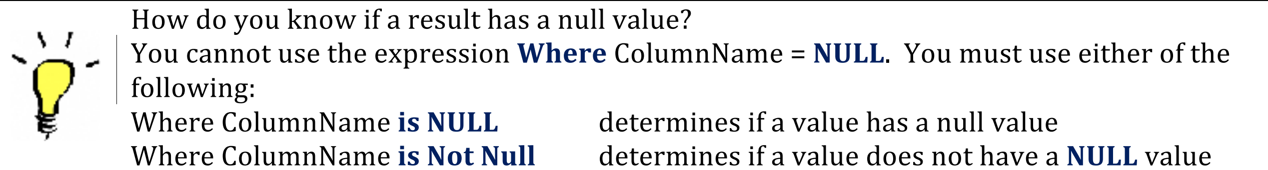
```
Select *
from Children
Left Outer Join Toys on Toys.Color = Children.FavoriteColor
Order by Children.Child
```

Results | Messages

	Child	FavoriteColor	FavoriteShape	LikesGames	Age	LikesLearning	Toy	Color	Shape	Educational	MinAge	Game
1	Beth	Green	Round	N	4	Y	NULL	NULL	NULL	NULL	NULL	NULL
2	Bobby	Red	Square	Y	10	N	Ball	Red	Round	N	3	N
3	Bobby	Red	Square	Y	10	N	Checkers	Red	Round	Y	6	Y
4	Cindy	Blue	Round	Y	8	Y	Hippity Hop	Blue	Round	N	6	N
5	Cindy	Blue	Round	Y	8	Y	Yoga Mat	Blue	Flat	Y	8	N
6	Dennis	Blue	Flat	N	8	Y	Hippity Hop	Blue	Round	N	6	N
7	Dennis	Blue	Flat	N	8	Y	Yoga Mat	Blue	Flat	Y	8	N
8	Lily	Yellow	Flat	N	6	N	Doll	Yellow	NULL	N	5	N
9	Lily	Yellow	Flat	N	6	N	Blocks	Yellow	Square	N	3	N
10	Mikey	Orange	Square	Y	10	Y	NULL	NULL	NULL	NULL	NULL	NULL
11	Oscar	Yellow	Round	Y	8	N	Doll	Yellow	NULL	N	5	N
12	Oscar	Yellow	Round	Y	8	N	Blocks	Yellow	Square	N	3	N
13	Robby	Blue	Round	Y	12	N	Hippity Hop	Blue	Round	N	6	N
14	Robby	Blue	Round	Y	12	N	Yoga Mat	Blue	Flat	Y	8	N
15	Tammy	Green	Square	N	7	N	NULL	NULL	NULL	NULL	NULL	NULL
16	Wendy	Red	Square	N	9	N	Ball	Red	Round	N	3	N
17	Wendy	Red	Square	N	9	N	Checkers	Red	Round	Y	6	Y

Things to notice

1) The entries where all of the values under the columns for Toys indicates that there are no matches for those children. Those children appear once with no suitable match.
2) If you exclude the three non-matching children, this results set is identical to the one from the **Inner Join**

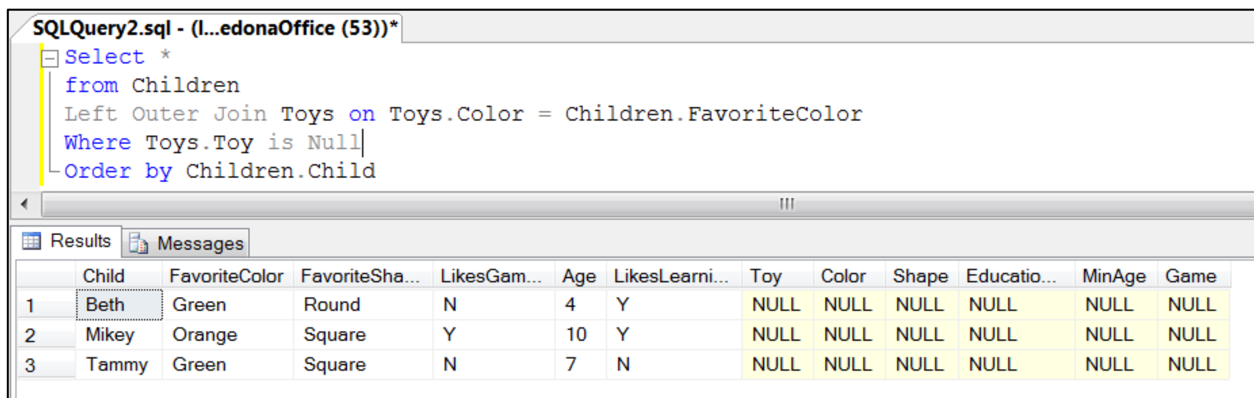


How do you know if a result has a null value?
You cannot use the expression **Where** ColumnName = **NULL**. You must use either of the following:

Where ColumnName **is NULL** determines if a value has a null value
Where ColumnName **is Not Null** determines if a value does not have a **NULL** value

You can use the above relations to identify the results that were not matched in a join.

Show me the Children that do not like a toy based on the color



SQLQuery2.sql - (l...edonaOffice (53))*

```
Select *
from Children
Left Outer Join Toys on Toys.Color = Children.FavoriteColor
Where Toys.Toy is Null
Order by Children.Child
```

Results | Messages

	Child	FavoriteColor	FavoriteSha...	LikesGam...	Age	LikesLearni...	Toy	Color	Shape	Educatio...	MinAge	Game
1	Beth	Green	Round	N	4	Y	NULL	NULL	NULL	NULL	NULL	NULL
2	Mikey	Orange	Square	Y	10	Y	NULL	NULL	NULL	NULL	NULL	NULL
3	Tammy	Green	Square	N	7	N	NULL	NULL	NULL	NULL	NULL	NULL

Notice that if you applied the criteria **Where** `Toys.Toy` **is Not Null** you would get results identical to the **inner join**!

Using Right Outer Joins

Right Outer Join works the same as the **Left Outer Join** with the roles reversed. Situations where you would use Right vs. Left Outer Join are mostly stylistic, although there could be specific situations where the **Right Outer Join** is the only choice.

Show me the children who could match any of the toys based on colors

SQLQuery2.sql - (l...edonaOffice (53))*

```
Select *
from Children
Right Outer Join Toys on Toys.Color = Children.FavoriteColor
Order by Children.Child
```

Results | Messages

	Child	FavoriteColor	FavoriteSha...	LikesGam...	Age	LikesLearni...	Toy	Color	Shape	Educatio...	MinAge	Game
1	NULL	NULL	NULL	NULL	NULL	NULL	Jacks	NULL	Round	N	5	Y
2	NULL	NULL	NULL	NULL	NULL	NULL	Uno	NULL	NULL	N	6	Y
3	NULL	NULL	NULL	NULL	NULL	NULL	Chess	NULL	NULL	Y	9	y
4	NULL	NULL	NULL	NULL	NULL	NULL	Legos	NULL	Square	N	6	N
5	NULL	NULL	NULL	NULL	NULL	NULL	Chutes and Ladders	NULL	Flat	N	4	Y
6	NULL	NULL	NULL	NULL	NULL	NULL	Barney Doll	Purple	Dinosaur	Y	6	N
7	Bobby	Red	Square	Y	10	N	Ball	Red	Round	N	3	N
8	Bobby	Red	Square	Y	10	N	Checkers	Red	Round	Y	6	Y
9	Cindy	Blue	Round	Y	8	Y	Yoga Mat	Blue	Flat	Y	8	N
10	Cindy	Blue	Round	Y	8	Y	Hippity Hop	Blue	Round	N	6	N
11	Dennis	Blue	Flat	N	8	Y	Hippity Hop	Blue	Round	N	6	N
12	Dennis	Blue	Flat	N	8	Y	Yoga Mat	Blue	Flat	Y	8	N
13	Lily	Yellow	Flat	N	6	N	Blocks	Yellow	Square	N	3	N
14	Lily	Yellow	Flat	N	6	N	Doll	Yellow	NULL	N	5	N
15	Oscar	Yellow	Round	Y	8	N	Doll	Yellow	NULL	N	5	N
16	Oscar	Yellow	Round	Y	8	N	Blocks	Yellow	Square	N	3	N
17	Robby	Blue	Round	Y	12	N	Hippity Hop	Blue	Round	N	6	N
18	Robby	Blue	Round	Y	12	N	Yoga Mat	Blue	Flat	Y	8	N
19	Wendy	Red	Square	N	9	N	Checkers	Red	Round	Y	6	Y
20	Wendy	Red	Square	N	9	N	Ball	Red	Round	N	3	N

Notice that the Order by statement puts all of the **NULL** children first!

Using Full Outer Joins

The **full outer join** matches the two tables as it can and reports at least one row for each member in both tables even if they do not match. As with the Left and Right joins, the unmatched values return **NULL**.

Show me all of the children and toys and match them by color preference if you can

SQLQuery2.sql - (l...edonaOffice (53))*

```
Select *
from Children
Full Outer Join Toys on Toys.Color = Children.FavoriteColor
Order by Children.Child
```

Results | Messages

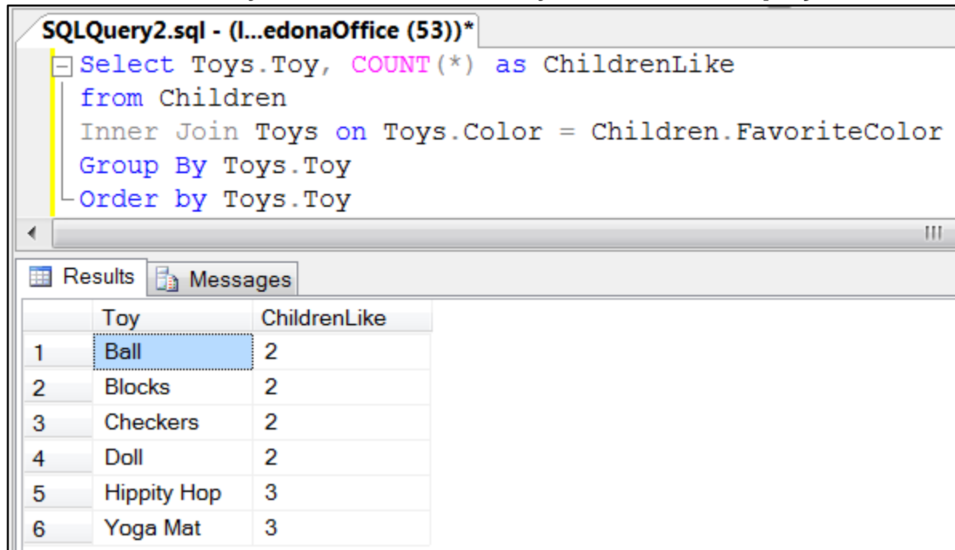
	Child	FavoriteColor	FavoriteSha...	LikesGam...	Age	LikesLearni...	Toy	Color	Shape	Educatio...	MinAge	Game
1	NULL	NULL	NULL	NULL	NULL	NULL	Jacks	NULL	Round	N	5	Y
2	NULL	NULL	NULL	NULL	NULL	NULL	Uno	NULL	NULL	N	6	Y
3	NULL	NULL	NULL	NULL	NULL	NULL	Chess	NULL	NULL	Y	9	y
4	NULL	NULL	NULL	NULL	NULL	NULL	Legos	NULL	Square	N	6	N
5	NULL	NULL	NULL	NULL	NULL	NULL	Chutes and Ladders	NULL	Flat	N	4	Y
6	NULL	NULL	NULL	NULL	NULL	NULL	Barney Doll	Purple	Dinosaur	Y	6	N
7	Beth	Green	Round	N	4	Y	NULL	NULL	NULL	NULL	NULL	NULL
8	Bobby	Red	Square	Y	10	N	Ball	Red	Round	N	3	N
9	Bobby	Red	Square	Y	10	N	Checkers	Red	Round	Y	6	Y
10	Cindy	Blue	Round	Y	8	Y	Hippity Hop	Blue	Round	N	6	N
11	Cindy	Blue	Round	Y	8	Y	Yoga Mat	Blue	Flat	Y	8	N
12	Dennis	Blue	Flat	N	8	Y	Hippity Hop	Blue	Round	N	6	N
13	Dennis	Blue	Flat	N	8	Y	Yoga Mat	Blue	Flat	Y	8	N
14	Lily	Yellow	Flat	N	6	N	Doll	Yellow	NULL	N	5	N
15	Lily	Yellow	Flat	N	6	N	Blocks	Yellow	Square	N	3	N
16	Mikey	Orange	Square	Y	10	Y	NULL	NULL	NULL	NULL	NULL	NULL
17	Oscar	Yellow	Round	Y	8	N	Doll	Yellow	NULL	N	5	N
18	Oscar	Yellow	Round	Y	8	N	Blocks	Yellow	Square	N	3	N
19	Robby	Blue	Round	Y	12	N	Hippity Hop	Blue	Round	N	6	N
20	Robby	Blue	Round	Y	12	N	Yoga Mat	Blue	Flat	Y	8	N
21	Tammy	Green	Square	N	7	N	NULL	NULL	NULL	NULL	NULL	NULL
22	Wendy	Red	Square	N	9	N	Ball	Red	Round	N	3	N
23	Wendy	Red	Square	N	9	N	Checkers	Red	Round	Y	6	Y

The result set is the same as that from the **Inner Join** combined with the unmatched items from the **Left** and **Right Outer joins**.

Working with the results from joined tables

The result sets associated with joined tables work just like those of a single table. You can use the **where** clause and group them for aggregates. When you are working with Outer joins it is very important to pay attention to possible **NULL** values and write your queries to account for them. If you do not do this you can get incorrect results thanks to the strange behavior of **NULL** when filtered.

Show me how many children like each toy based on color preference



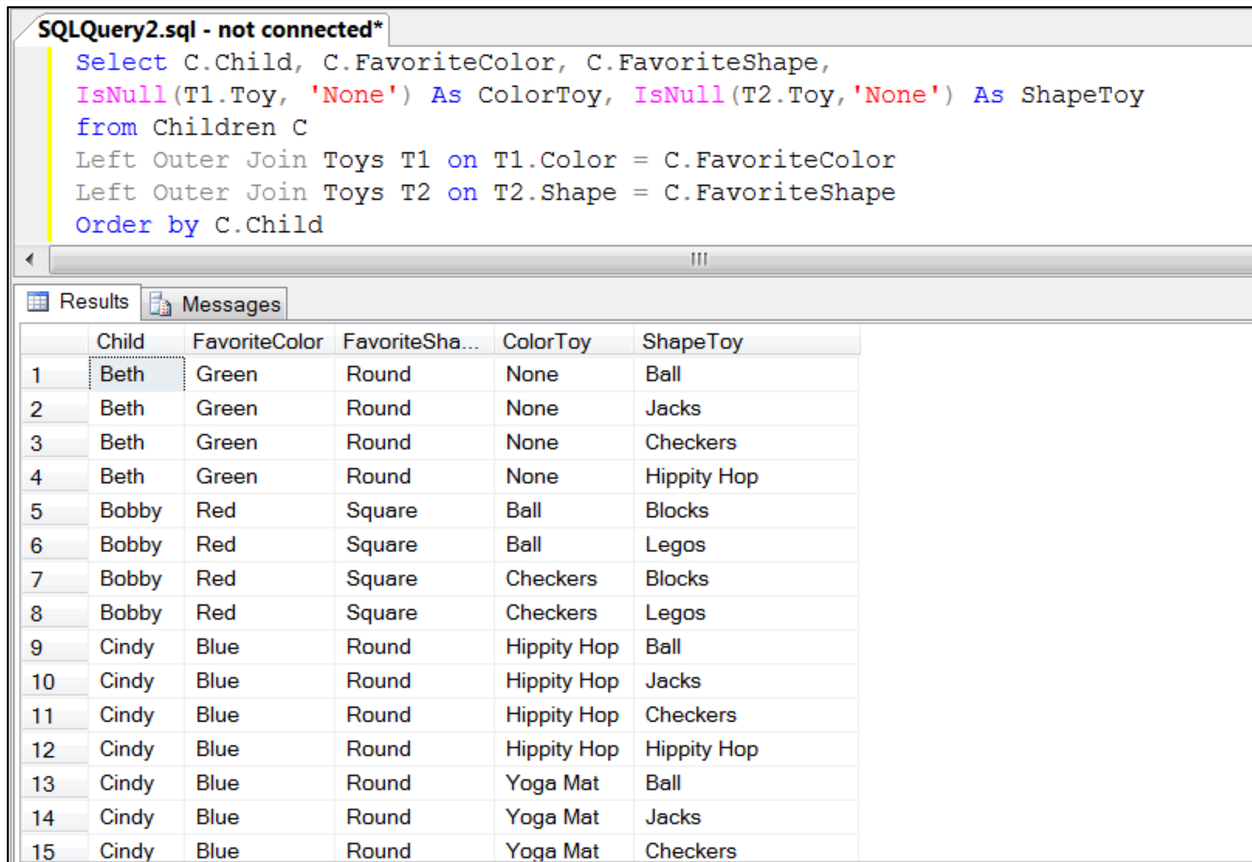
	Toy	ChildrenLike
1	Ball	2
2	Blocks	2
3	Checkers	2
4	Doll	2
5	Hippity Hop	3
6	Yoga Mat	3

Joining to more than one table

There is no limit to how many tables can be involved in a set of joins. Simply follow each join with another join and the table relationship. You can even join to the same table more than once, however you must use an alias when referring to that table.

	Alias. An alias gives SQL a second "nickname" for a table. This saves typing and may be required in certain cases. The Alias is designated by following the tablename with the chosen abbreviation. Example: **Select * From** ThisIsAReallyLongTableName t → ***t*** is the Alias!

Show me all of the Toys that the Children like based on either color preference or shape preference



SQLQuery2.sql - not connected*

```
Select C.Child, C.FavoriteColor, C.FavoriteShape,
IsNull(T1.Toy, 'None') As ColorToy, IsNull(T2.Toy,'None') As ShapeToy
from Children C
Left Outer Join Toys T1 on T1.Color = C.FavoriteColor
Left Outer Join Toys T2 on T2.Shape = C.FavoriteShape
Order by C.Child
```

Results | Messages

	Child	FavoriteColor	FavoriteSha...	ColorToy	ShapeToy
1	Beth	Green	Round	None	Ball
2	Beth	Green	Round	None	Jacks
3	Beth	Green	Round	None	Checkers
4	Beth	Green	Round	None	Hippity Hop
5	Bobby	Red	Square	Ball	Blocks
6	Bobby	Red	Square	Ball	Legos
7	Bobby	Red	Square	Checkers	Blocks
8	Bobby	Red	Square	Checkers	Legos
9	Cindy	Blue	Round	Hippity Hop	Ball
10	Cindy	Blue	Round	Hippity Hop	Jacks
11	Cindy	Blue	Round	Hippity Hop	Checkers
12	Cindy	Blue	Round	Hippity Hop	Hippity Hop
13	Cindy	Blue	Round	Yoga Mat	Ball
14	Cindy	Blue	Round	Yoga Mat	Jacks
15	Cindy	Blue	Round	Yoga Mat	Checkers

Things to notice:

1) This query uses aliases for the three tables involved. Children is referred to as C and Toys is joined to twice as T1 and T2.
2) **ISNULL** is used to replace **NULL** values with a people friendly 'None'.
3) You have to be careful when interpreting the results using joins (especially outer joins). Cindy lists Hippity Hop four times because it matches her color preference and there are four toys that match her shape preference. If you were to count toys for T1.Toys, the number would be skewed!

Using Distinct to Reduce Duplication

In the above example as in many joins there is the chance you will duplicate lines of data. SQL provides the operation **Distinct** to only report unique combinations of the data. You add **Distinct** after the word **Select** and any replicated lines are excluded. If we change the query to:

SQLQuery2.sql - (l...edonaOffice (54))*

```
Select C.Child, C.FavoriteColor, C.FavoriteShape,
IsNull(T1.Toy, 'None') As ColorToy --, IsNull(T2.Toy,'None') As ShapeToy
from Children C
Left Outer Join Toys T1 on T1.Color = C.FavoriteColor
Left Outer Join Toys T2 on T2.Shape = C.FavoriteShape
Order by C.Child
```

Results | Messages

	Child	FavoriteColor	FavoriteSha...	ColorToy
1	Beth	Green	Round	None
2	Beth	Green	Round	None
3	Beth	Green	Round	None
4	Beth	Green	Round	None
5	Bobby	Red	Square	Ball
6	Bobby	Red	Square	Ball
7	Bobby	Red	Square	Checkers
8	Bobby	Red	Square	Checkers
9	Cindy	Blue	Round	Hippity Hop
10	Cindy	Blue	Round	Hippity Hop
11	Cindy	Blue	Round	Hippity Hop
12	Cindy	Blue	Round	Hippity Hop
13	Cindy	Blue	Round	Yoga Mat
14	Cindy	Blue	Round	Yoga Mat
15	Cindy	Blue	Round	Yoga Mat

Double dash XX In SQL any line preceded by a double dash is treated as a comment and is not included in your query. This is useful to add comments to queries and to temporarily exclude parts of a query when you don't want them to run this time but do not want to remove them.

In the above result set there are lots of duplicated line items but if we add **Distinct**!

SQLQuery2.sql - (l...edonaOffice (54))*

```
Select Distinct C.Child, C.FavoriteColor, C.FavoriteShape,
IsNull(T1.Toy, 'None') As ColorToy --, IsNull(T2.Toy,'None') As ShapeToy
from Children C
Left Outer Join Toys T1 on T1.Color = C.FavoriteColor
Left Outer Join Toys T2 on T2.Shape = C.FavoriteShape
Order by C.Child
```

Results | Messages

	Child	FavoriteColor	FavoriteSha...	ColorToy
1	Beth	Green	Round	None
2	Bobby	Red	Square	Ball
3	Bobby	Red	Square	Checkers
4	Cindy	Blue	Round	Hippity Hop
5	Cindy	Blue	Round	Yoga Mat
6	Dennis	Blue	Flat	Hippity Hop
7	Dennis	Blue	Flat	Yoga Mat
8	Lily	Yellow	Flat	Blocks
9	Lily	Yellow	Flat	Doll
10	Mikey	Orange	Square	None
11	Oscar	Yellow	Round	Blocks
12	Oscar	Yellow	Round	Doll
13	Robby	Blue	Round	Hippity Hop
14	Robby	Blue	Round	Yoga Mat
15	Tammy	Green	Square	None
16	Wendy	Red	Square	Ball
17	Wendy	Red	Square	Checkers

Everything is clean! You can use **distinct** to quickly test if you are getting unexpected duplicate results by running your query without **distinct**, counting the lines and then running it with **distinct** and looking at the difference.

Crossed Join When a join unexpectedly duplicates result lines, this is termed a *crossed join*. You can eliminate crossed joins by correcting the logic of your query or by using **Distinct** when applicable.

Writing Queries That Make Sense

As you produce queries that become longer and more complex you have to be disciplined in how you format your text. Consider the following query:

```
Select S.LastName, S.[First Name], C.ClassName, T.LastName,Convert
(NVARChar(50), DATEADD (hh,CSD.StartTime , DateAdd (dd,CSD.Weekday,'2014-1-
5')),0) as 'Start Time', Convert (NVARChar(50), DateAdd
(mi,CSD.Duration,DATEADD (hh,CSD.StartTime , DateAdd (dd,CSD.Weekday,'2014-1-
5'))),0) as 'End Time' from ClassSchedule Cs Inner Join Classes C On
C.ClassID = Cs.ClassID Inner Join ClassScheduleDays CSD on CSD.ClassID =
C.ClassID Inner Join StudentList S on S.StudentID = Cs.StudentID Inner Join
TeacherList T on T.TeacherID = C.Professor Order by S.LastName, S.[First
Name], CSD.WeekDay, CSD.StartTime
```

You can run this query and it will work fine, but imagine if you have to change something!

The same query can be written like this:

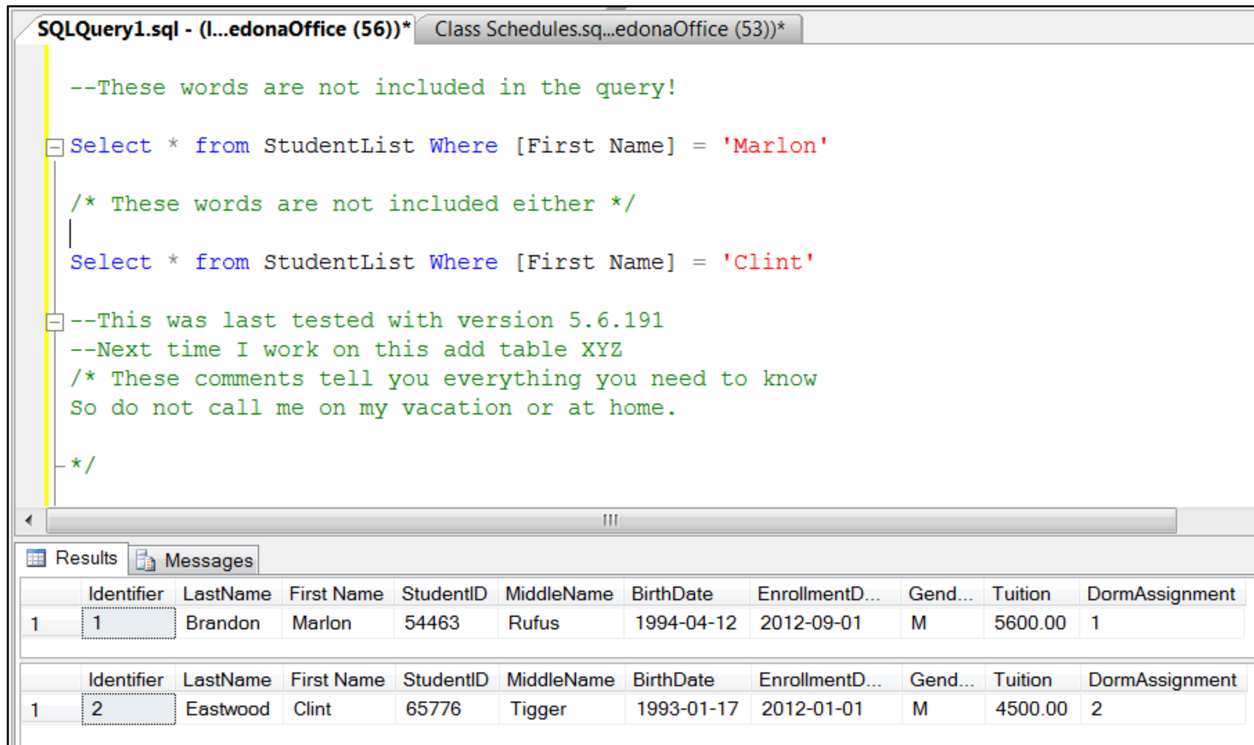
```
Select S.LastName, S.[First Name], C.ClassName, T.LastName,
Convert (NVARChar(50), DATEADD (hh,CSD.StartTime , DateAdd
(dd,CSD.Weekday,'2014-1-5')),0) as 'Start Time',
Convert (NVARChar(50), DateAdd (mi,CSD.Duration,DATEADD (hh,CSD.StartTime ,
DateAdd (dd,CSD.Weekday,'2014-1-5'))),0) as 'End Time'
from ClassSchedule Cs
Inner Join Classes C On C.ClassID = Cs.ClassID
Inner Join ClassScheduleDays CSD on CSD.ClassID = C.ClassID
Inner Join StudentList S on S.StudentID = Cs.StudentID
Inner Join TeacherList T on T.TeacherID = C.Professor
Order by S.LastName, S.[First Name], CSD.WeekDay, CSD.StartTime
```

Try to observe the following rules:

1) Label results columns to avoid confusion
2) Limit the number of results columns on each line.
3) Use a separate line for each join
4) Replace NULL values with people friendly terms ISNULL (????, 'N/A')
5) Use sensible aliases
6) If it looks confusing, then it probably is

Letting Others Know What You Were Thinking

SQL provides two styles for commenting your queries. “Comments” are text added in and around your query that inform the user how it works, why it operates the way it does, what version it is, to do listing, etc.



Simple queries need little or no commenting, while complicated queries should have plenty of comments. It adds time to your process, but saves lots of time later. Do not fall into the “I will comment it later trap” because you won’t. Write comments to tell future you why you did that and you will thank you! `--Rethink the “future” you thing…`

When You Need Help

Don’t buy a book. I have lots of books on SQL and they mostly hold down my bookshelf. The internet should be your #1 source for information and help. Every last feature in SQL is explained clearly with references and suggestions to other things you might want to read about. If you spend time on the internet reading about SQL, next year you will be teaching this class!

Appendix

Appendix

Query to find table and column names

```
Select IST.Table_Name, ISC.COLUMN_NAME
From Information_Schema.Tables IST
Inner Join Information_Schema.Columns ISC On IST.Table_Name = ISC.Table_Name
order by IST.table_name, ISC.COLUMN_NAME
```

Query to find a specific Column

```
Select IST.Table_Name, ISC.COLUMN_NAME
From Information_Schema.Tables IST
Inner Join Information_Schema.Columns ISC On IST.Table_Name = ISC.Table_Name
where column_name like '%?????%'
order by IST.table_name, ISC.COLUMN_NAME
```

Query to find a Column or Table

```
Select IST.Table_Name, ISC.COLUMN_NAME
From Information_Schema.Tables IST
Inner Join Information_Schema.Columns ISC On IST.Table_Name = ISC.Table_Name
where column_name like '%?????%'  or IST.TABLE_NAME like '%?????%'
order by IST.table_name, ISC.COLUMN_NAME
```