**QUERIES**

**Introduction**

Queries are used to retrieve data from the database. They can be very flexible in what columns they select, what rows they select, and what aggregate calculations they return. Being able to write accurate and efficient queries is a critical skill when working with databases.

Queries are written with the SELECT statement. The full syntax available for a select statement is:

SELECT [ALL | DISTINCT] column\_list INTO [new\_table\_name][FROM table\_name [, table\_name2 […, table\_name16]][INNER, LEFT OUTER, RIGHT OUTER JOIN]

[WHERE clause][GROUP BY clause][HAVING clause][ORDER BY clause][COMPUTE clause]

If it looks complex don’t worry. All the “clauses” listed don’t need to be used when creating a query. However, what you do use MUST be in the order shown above (HAVING **MUST** be after GROUP BY for example and not vice versa). In addition, there are other functionality beyond the syntax such as the ability to use string, date, and aggregate functions in your queries to retrieve and calculate date to be returned. We will start by looking at simple queries and then adding in the additional clauses after that.

Al the examples can be executed in the IQSchool database so you can see the results.

**SIMPLE QUERY**

A simple query can select data from one table. The query can select many rows, calculated data, or even hard coded data when needed. Calculated data be mathematical calculations (subtotal \* 1/05) or can be calculated string data (firstname + ‘ ‘ + lastname). It can even be a combination of them both (‘total: ‘ + subtotal \* 1.05). While formatting what you selecting in your query is not a common practice (formatting is best left to the application requesting the data) is can be useful sometimes.

**Example:**

C:\Users\sbell\AppData\Local\Temp\SNAGHTML445063a9.PNG

As you can see this will select the students firstname, lastname and city for all the student records. The From clause indicates the table the columns are in that you are selecting. Queries like this are very simple to create and are quite common.

To select the students name as one column instead of separate first name and last name columns we can concatenate them together as follows.

C:\Users\sbell\AppData\Local\Temp\SNAGHTML4450e1bd.PNG

In SQL the + operator will add strings together as well as numbers. Just remember to add (concatenate) a space between the names otherwise you will get only one long name with firstname and lastname combined together.

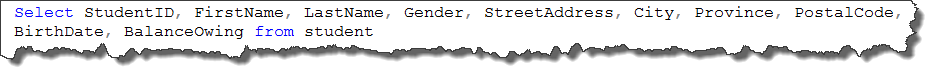
There is another difference with the query that returns the names separately than the one that returned them as one column. The column in the second query does not have a name. This is a problem. We should give all columns a name. Calculated columns do not have a name because they are calculated from many different columns and values. So, we must give the column a name. There are several different ways to do this.

C:\Users\sbell\AppData\Local\Temp\SNAGHTML44516ec0.PNGOR

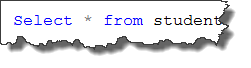
C:\Users\sbell\AppData\Local\Temp\SNAGHTML44521b34.PNG

In this course you will be expected to ALWAYS have column names.

To select all the student information for all the students and creating one custom column for student name could look like this:



Since we are selecting all the columns in the student table we could use the ‘\*’ operator. The asterisk represents all columns when used in the column list of a query. So the same query could look like this:



However, there are some possible problems by taking the shortcut and not listing all the column names explicitly.

By looking at the second query can you tell what columns are being returned? It is much easier to maintain the query if all the columns are listed.

It is important to understand that the data being returned will be displayed on a user friendly screen to most users. This could include a desktop application, a web page, or a mobile device. Consider this scenario. 10 years ago the query was written with an asterisk to return all the student information to be displayed on the user’s desktop computer. The program was written to display 10 columns of data. Last week, someone added another 6 columns of confidential data to the table but those are not supposed to be displayed to the user who is using the original program. Depending on how the program was written it may have these results:

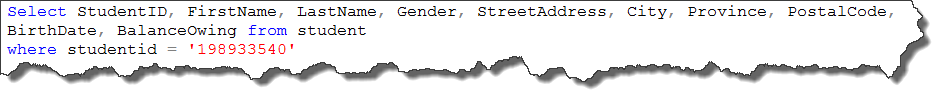
1. It may display all the columns (including the new ones which may contain confidential information) when it only should have displayed the original 10 columns that existed when the program was written and the query created.
2. It may crash the users program because the program was created to display 10 columns of data and the additional columns now being returned cause an error.
3. The program behaves normally but there is 6 columns of data being returned to the program that it does not use. An inefficient situation.

While it can be argued that by using the asterisk instead of the column list the query will always return ALL the columns from the table. Even if the table has columns added at a later date and therefore always keeping the query current. This is something to consider but those situations are fewer.

In this course the expectation will be that you will not use the asterisk in the column list to select all columns. List the columns explicitly.

**Where**

So far we have only selected data that includes ALL the records on the table. While this is not uncommon it is likely more common that you will not only want to select the columns you want but also only the records you want. The WHERE clause allows you to limit the records that are returned based on criteria you provide.



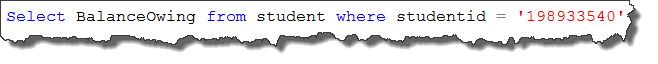
This query would return all the student information for the student with studentid 198933540. The where clause can contain many different expressions to identify the records the query should return.

**Search Criteria Operators and Conditions**

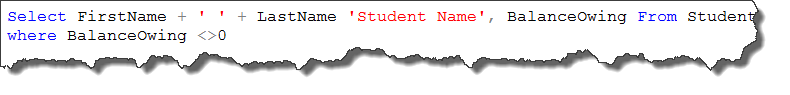
Sometimes the conditions that must be met in the data for certain records to be returned involved many expressions that may be combined using AND/OR operators or several other ones.

Operators

= The equals sign is used when looking for an exact match



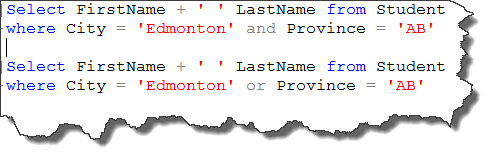
<> OR != The not equal sign is used when looking for something that does not match your criteria



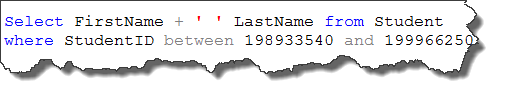
Other common operators include <, <=, >, >=

AND Both expressions must be true for the record to be returned

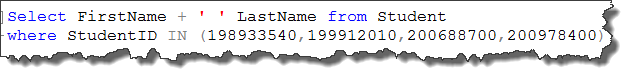
OR If either of the expressions are true then the record is returned



BETWEEN Returns all records where the search criteria is between two values (inclusive). Can use NOT BETWEEN to return records where the search criteria is outside the range.



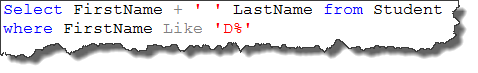
IN allows you to list a number of valid search values for a column. You can always code the same query with a compound expression with multiple OR conditions but that can get quite large with many values. You may also use NOT to return everything that is NOT in the list.



**LIKE**

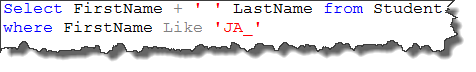
The Like operator is used when you use wildcards in your search or perform pattern matching. Wildcards are characters that when used with the Like operator allows the wildcards to represent any character or range of characters.

For example, If we wanted to return all the students names whose names started with the letter ‘D’ we would use the % wildcard character. The % wildcard means any character and any number of characters.



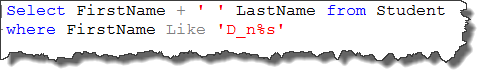
It is important to use the like operator when using wildcards. The % would be interpreted as a literal ‘%’ if we used = instead of Like.

The \_ (underscore) is a single character wildcard. This would be used when you are looking for any one character in your search criteria. If we wanted the students names whose First Name was 3 characters long and started with ‘JA’ we could use this query.

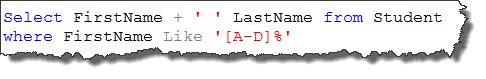


If we had used a % instead of \_ we would also return Students whose First Name was longer than 3 characters.

They can also be used in combination with each other. Let’s return all the Student Names that start with ‘D’, the third character is ‘n’ and end in ‘s’.

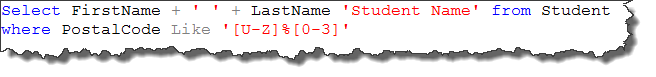


You can also perform a search on a range of values using Like. A query to return all the student names that start with A through D could look like this.



The [A-D] means that the first character must be in that range and can be followed by any number of characters after that (%).

You can also use a ranger of numbers for a particular position. Let’s select all the student names who have a Postal Code that starts with [U-Z] and ends with [0-3].



**UNION**

# **UNION**

The union operation lets you combine the data retrieved by multiple

select statements into a single result table

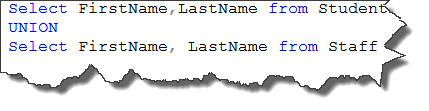
syntax:

SELECT . . .

UNION [ALL]

SELECT . . .

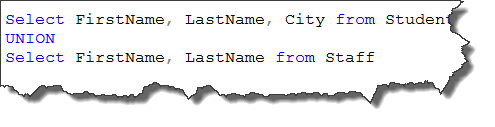
Let’s write a query to return all the student names and combine it (UNION) with all the staff names.



This results in 25 records being returned. There are 16 students and 10 staff. Why are we missing one? UNION omits duplicates unless you us UNION ALL. You can UNION as many queries together as you need to.

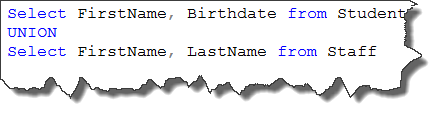
There are some rules to follow however:

1. All the queries being combined with UNION must return the same number of columns



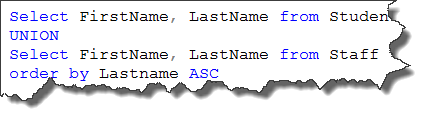
All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists.

1. The columns being combined must share similar datatypes



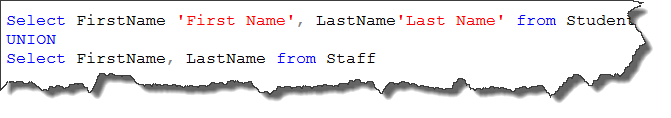
Conversion failed when converting character string to smalldatetime data type.

1. Ordering the results must be done after the last query



**WORKS!**

1. Column names are defined in the first query



**WORKS!**

AGGREGATE FUNCTIONS

Aggregate functions calculate a single value using the data from

many records. If you have used function in Excel such as SUM, AVERAGE, COUNT, etc…. then you have used aggregate functions.

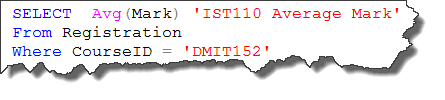
The basic syntax for all the aggregate functions is the same:

aggregate function name([ all | distinct ] expression)

* All and distinct are optional with All being the default if none is used.
* expression can be any appropriate column name

**AVG**

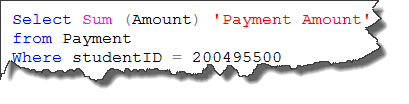
The AVG function returns the average of a column of values. It can only be used on numeric columns and records containing null are ignored and not included in the average calculation.



Returns the average mark from the registration table for Course DMIT152. Note that aggregate columns are not columns in the database and therefore do not have a name. Remember, all column must be given a name if they do not have one.

**SUM**

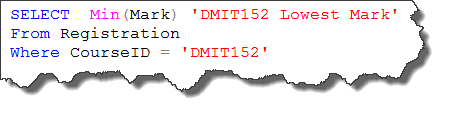
The SUM function returns the sum of a column of values. It can only be used on numeric columns and records containing null are ignored and not included in the sum calculation.



Returns the Sum of all the payments for studentID 200495500.

**MIN**

The MIN function returns the minimum value from a column of values. Can be used with columns that have a a numeric, date or character datatype. Records with a null value are ignored.

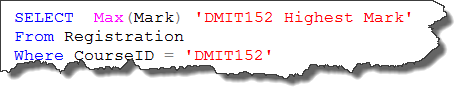


Returns the lowest Mark in DMIT152.

**MAX**

The MAX function returns the maximum value from a column of values.

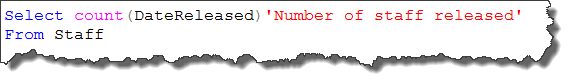
Can be used with columns that have a a numeric, date or character datatype. Records with a null value are ignored.



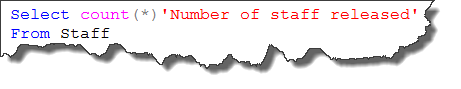
Returns the highest Mark in DMIT152.

**COUNT**

The COUNT function returns the number of non-null values from a column of values OR returns the number of records that match the where criteria.



Returns the number of staff that have been released (have a value in the DateReleased column).



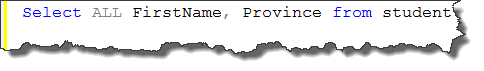
Returns the number if records returns by this query (number of staff RECORDS in the staff table).

NOTE: The \* operator is an overloaded operator. This means that it has more than one meaning depending on how it is used. In a column list in a query it means all columns (which should be avoided unless necessary). In the count function it means ‘records’.

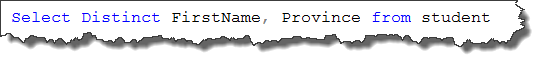
**If you are simply counting records use count(\*). Do not use a column name unless you must only count the number of records where a particular column is not null.**

**ALL AND DISTINCT**

ALL and DISTINCT can be in a query or with a COUNT function. ALL is the default and usually not explicitly declared.



Returns all the Student First Names and their Province. This returns 16 records because there are 16 students.



Returns all the unique (Distinct) student First Names and their Province. This returns only 15 records because there are 2 students with a First Name of Joe from Alberta.

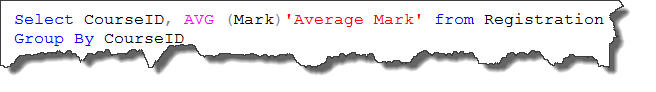
When DISTINCT is used with the COUNT function it would count only the unique(Distinct) values.

**GROUP BY**

The GROUP BY clause is used in conjunction with aggregate functions to provide subtotal calculations of the aggregate function. We could easily return the average Mark from the grade table. What if we wanted the average Mark for each Course? For each student? For each course for each student? To do this we would use GROUP BY.



Returns the average of all the marks in the Registration table.



Returns the average mark for each CourseID (GROUP BY CourseID) in the Registration Table. Another way to think of the syntax GROUP BY is the ‘for each’.

It is important to note that you cannot combine an aggregate column with a non-aggregate column(s) unless you group by all the non-aggregate columns in your query.

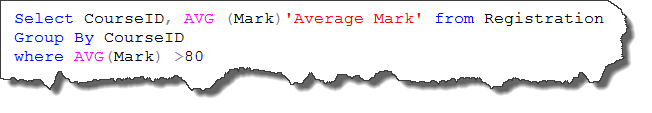


Column 'Registration.CourseId' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

Remember that an aggregate returns one calculated value. How can it return the average mark for the Reigstration table (one value) as well as each course ID that is in the Registration table (many values). It simply does not make sense and cannot be done. So the error you see above reminds you that you must group by all the columns you are using in your query that are not part of the aggregate function. In this case, CourseID.

**HAVING**

Having is like the where clause. Except, it applies its criteria after group by has grouped its aggregate data. With the GROUP BY clause we selected the average mark of CourseID. What if we only wanted to return the CourseID’s and average mark for courses that have averages that are greater than 80?



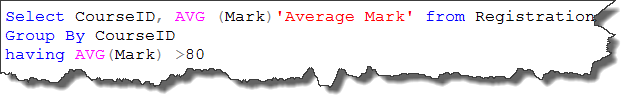
Incorrect syntax near the keyword 'where'.

Fails because the where clause must be before the Group By Clause. Remember, a query does not need to use all the clauses but they must be in the right order!



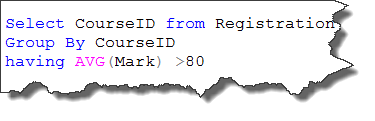
An aggregate may not appear in the WHERE clause unless it is in a subquery contained in a HAVING clause or a select list, and the column being aggregated is an outer reference.

Fails because and aggregate cannot be in a where clause. It also does not really make sense. It is not possible to return only the CourseID’s that have average over 80 until all the Course averages have been calculated. And that is done by the Group By Clause. So, the Group By Clause must execute before the criteria is applied to return only the CourseID’s with averages over 80.



Works!The Select with the group by selects the averages for each course and then the having clause returns only the ones whose average was over 80.

What if we only wanted to return the CourseID’s that have averages over 80 and did not want the actual average returned. Simply do not select the AVG(Mark). Remember, what we select (return to the user) does not have to match columns used in the where or having clauses.



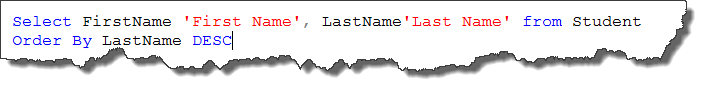
Works! ☺

**Having is used with aggregate functions only. It is used to apply where clause functionality AFTER the aggregate function and group by has occured. Only use having in this situation. For queries without a group by clause you should use the WHERE clause.**

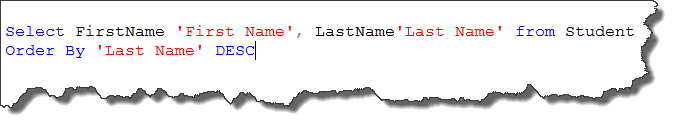
ORDER BY

The ORDER BY clause is used to return the results in a certain order. The results can be sorted by one column or more than one column in either descending (DESC) or ascending (ASC) order. When no order is specified, ASC is the default.

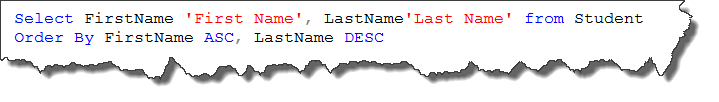
There are numerous ways to use ORDER BY.



Returns the Student Names in Descending order by last name (reverse alphabetical).



Returns the Student Names in Descending order by last name using the alias (name) we gave to the LastName column.



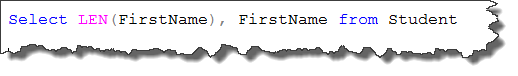
Returns the Student Names in Ascending order of FirstName and when for duplicate first names sorts those by Descending order of LastName.

ASC is the default and is commonly accepted so it is not uncommon to see the Order By clause used for Ascending ordering without ASC. However you must use DESC if you want the results sorted in Descending order.

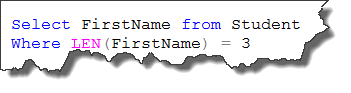
**STRING FUNCTIONS**

There are many string functions that will allow you to work with parts of strings to define search criteria and results.

LEN (column | expression)

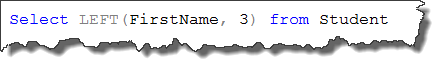


Returns the length of each FirstName and the FirstName of each Student.



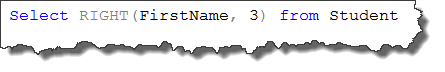
Returns all the First Names that are 3 characters long.

LEFT (column |expression, length)



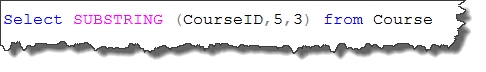
Returns the first 3 characters from the FirstName of each student.

RIGHT (column |expression, length)



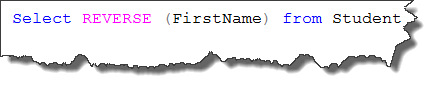
Returns the last 3 characters from the FirstName of each student.

SUBSTRING (column | expression, start, length)



In each courseID returns 3 characters starting at the 5th character.

REVERSE (column | expression)



Returns the Student Firstnames in reverse.

UPPER (column | expression) LOWER (column | expression)

C:\Users\sbell\AppData\Local\Temp\SNAGHTML48e41b2c.PNG

Returns the names in either uppercase or lowercase.

LTRIM (column | expression) RTRIM (column | expression)



Removes the leading (LTRIM) or trailing (RTRIM) spaces from the column data or expression. To remove leading and trailing spaces you must nest one function in another.

**DATE** Functions

GETDATE () returns the system date

DATEADD (**xx**, n, date1) returns a new date adding the number of **xx** to date1

n may be negative

DATEDIFF (**xx**, date1, date2) returns the number of **xx** from date1 (older) to

date2 (newer) – can return negative numbers

DATENAME (**xx**, date1) returns a string representation of the **xx** of date1

DATEPART (**xx**, date1) returns an integer representation of the **xx** of date1

YEAR (date1) returns the year portion of date1,

same functionality as DATEPART (yy, date1)

MONTH (date1) returns the month portion of date1,

same functionality as DATEPART (mm, date1)

DAY (date1) returns the day portion of date1,

same functionality as DATEPART (dd, date1)

**NOTE: xx** represents the datepart from the table below.

**Datepart** **Abbreviation** **Minimum Maximum**

Year yy 1753 9999

Quarter qq 1 4

Month mm 1 12

Week wk 1 53

Day of year dy 1 366

WeekDay dw 1 (Sun.) 7 (Sat.)

Day dd 1 31

Hour hh 0 23

Minute mi 0 59

Second ss 0 59

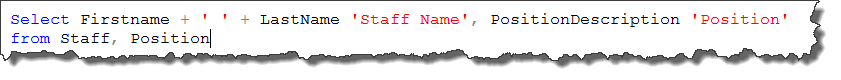
Millisecond ms 0 999

**JOINS**

To work with data that is located in more than one table you need to instruct the server how to relate records from one table with its related record(s) in other tables. To do this you must use the join operator. To join operator lets us control how the server relates rows from different tables.

If you wanted to select all the staff names and their position descriptions you would need to select the staff names from the staff table and the position descriptions from the position table.

According to the syntax of a Select statement you could write a select statement as follows:



When you run this Select statement you return 70 records. This may seem like the correct select statement as there were no errors. However, closer inspection of the data in the two tables involved leads to some questions.

First, the Staff table contains only 10 records. So, we should only be seeing 10 records coming back (select all the staff names and their position descriptions).

Second, the position table has 7 records (there are 7 positions).

So, where did the 70 records come from? If you look at the results of the select closely you will notice that every staff member seems to have every position. 10 Staff X 7 Positions = 70.

What happened was there was nothing in the select statement telling the server how to relate what records on the Position table relate to what records in the Staff table.

Remember, records in tables relate to each other through the foreign key. The value of the primary key in the parent table relates to records in the child table that have the same value. Staff that are Position 2 (Program Chair) will have a value of 2 in the PositionID field which is the foreign key in the Staff Table.

Obviously every staff member is not in every position at the school so we need to instruct the server which field in the parent table relates to what field in the child table. This is usually the Primary Key in one table and the Foreign Key in another. While this is not the only fields that are used in joins they are on practically 100% of the cases. This is the Parent field in the parent table and the foreign key in the child table. In this case, it is the PositionID which joins the Position and Staff table and forms the relationship. It also identifies what records in the parent relate to what records in the child.

To ensure we get only the Parent records associated with their correct child records we must use syntax which identifies what the common (join) field is between the tables.

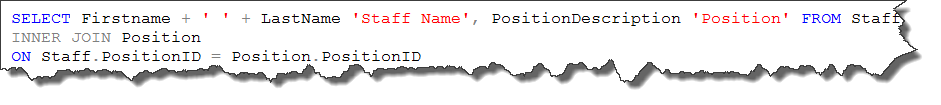
Syntax:

SELECT field1, field2,… FROM table1

INNER JOIN table2

ON table1.joinfield = table2.joinfield

The Staff and Position problem using the correct syntax:

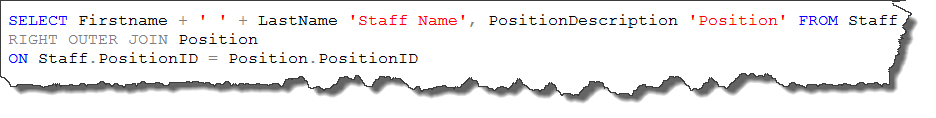


The INNER JOIN syntax tells the server that we are performing an inner join and identifies that PositionID is the field in the 2 tables that relate them together.

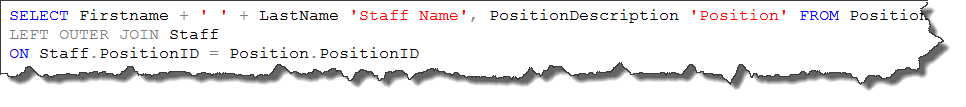
More specifically it performs an Inner Join which is different than an Outer Join. An Inner Join only returns records from the parent table that have related records in the child table. In this example Position is the Parent table (Primary Key in the relationship) and Staff is the child table (has the foreign key in the relationship). Looking back at the results you will notice that one Position Description is not returned in the results. Position 7 (Assistant Dean) was not returned because there are no Staff with that position. With an Inner Join parent records that do not have a matching child record are not returned.

An Outer Join is different in that it returns all the parent records, including the ones that do not have a matching record in the related child table. To return ALL the Postion Descriptions and the staff names that are in those positions you would use an Outer Join.

We will discuss Left Outer Joins and Right Outer Joins. These both have the same purpose and you choose which one to use based on the order you list your tables in your select statement.



OR



Notice the difference between the Left and Right Outer Join?

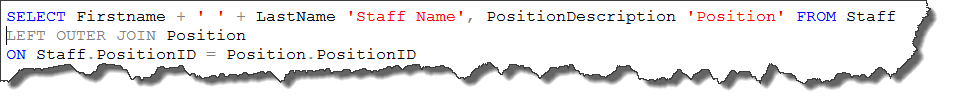
The first example states to select ALL the records from the table that is on the RIGHT side of the OUTER JOIN statement (Position) and the related records from Staff.

The second example states to select ALL the records from the table that is on the LEFT side of the OUTER JOIN statement (Position) and the related records from Staff.

Outer Joins enable us to select parent records that may not have related child records in the other table(s) in the join.

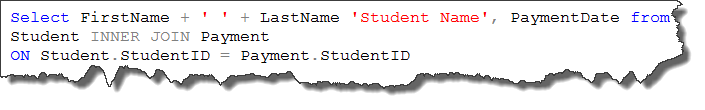
Although not necessary, joins are often coded with the parent table on the left side so you tend to see more Left Outer Join statements than Right Outer Join.

Why would you likely never see this Select statement and why does it not make sense?

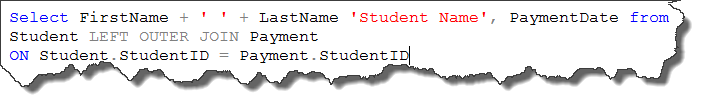


This select says select ALL the staff records and the related records from Position. Since the Staff table is the child table, all the staff records must have a matching parent record in the Position table. Remember, all child records must have a matching parent record.

The difference between INNER and OUTER JOIN:



(returns 33 records)



(returns 42 records)

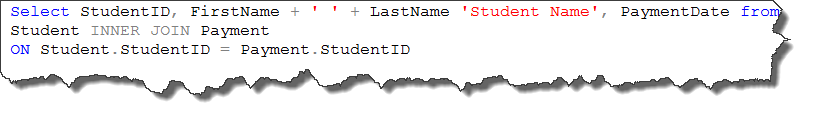
Can you explain the difference?

The difference is there are only 33 students that have made payments (only 33 parent records that have matching child records) so only 33 records returned.

An Inner Join only returns Parent records that have matching child records.

An Outer Join returns ALL the Parent records (students) even if they do not have any matching child records (payments). Therefore we now know that 9 students have no payment records. Since there were no payment dates for those students the PaymentDate field for those records contains the value of NULL.

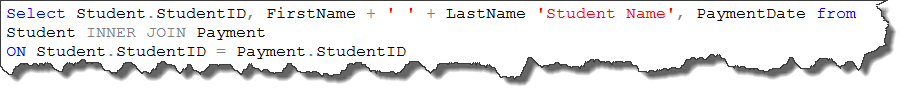
Consider the following Select statement which is now selecting the StudentID field as well:



This will result in an error:

Ambiguous column name 'StudentID'.

This error is due to the fact that we must state the table that we are selecting the fields from. Since StudentID is in both the Student and the Payment table, the server does not know which one to select. To solve this we need to adjust our syntax to identify which table the server should select the ambiguous column from.



While it does not usually matter which table you select the ambiguous columns from it is almost always the parent table.

You can also encounter this error when you have an ambiguous column in other parts of a select statement (like where).

**Selecting From 3 or More Tables**

Selecting from 3 or more table is the same as selecting data that is in 2 tables. Remember the syntax for a basic Inner Join is:

SELECT field1, field2,… FROM table1

INNER JOIN table2

ON table1.joinfield = table2.joinfield

To select from more than 2 tables the syntax would look like:

SELECT field1, field2,… FROM table1

INNER JOIN table2

ON table1.joinfield = table2.joinfield

INNER JOIN table3

ON table.joinfield = table.joinfield

INNER JOIN table4

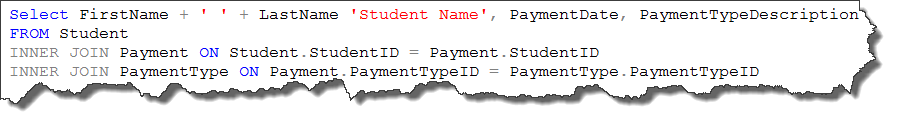
ON table.joinfield = table.joinfield

…..

For every table need to add to the select you use INNER JOIN to add it to the select and indicate with ON how that table relates to another table you have already in your select.

Example:

Select the Student Names, Payment Dates, and Payment Type Descriptions for all the students that have payments:



Going back to non clustered indexes, do you see why we use them on foreign key fields? Everytime we need data that is in different tables we use joins and joins use foreign keys to relate the records in the parent table with their correct child records. Joins are very common in selecting data from a database and indexes help to spead up those join operations.

**SUMMARY**

* Joins enable us to select data that is from different, related tables
* We use join expressions (syntax) to identify what the field is that joins each table in the relationship. This allows the select statement to return information in one table and its related information in another
* Inner Joins only return parent records that have at least one matching child record
* Outer Joins return all the parent records as well as the related child records
* Left and Right Outer Joins perform the same task and you can use either one. You must use the correct one for the order you have the tables in the select statement
* If you are selecting a field that exists in more than one table that you are selecting from you must specify which table to select from. Ambiguous column names errors result otherwise.
* All the other clauses of a Select statement can be used with joins.
* A join can be between 2 or more tables.
* You do not need to select a field from each table you have in your join..

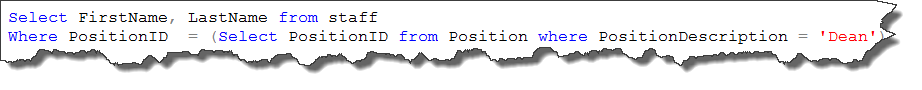
**SUBQUERIES**

A subquery is a SELECT statement that is nested inside another statement. It is a full and complete select statement and can execute on its own. We will first look at subqueries as a Select statement inside another select statement. Subqueries are used in the where or having clause to identify the records that need to be returned.

We often refer to the subquery portion (nested inside the other select) as the inner query and the Select statement around it as the Outer Query.

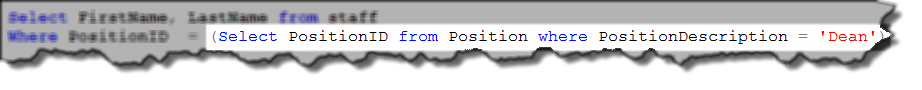
There are two types of subqueries. They are Nested and Correlated. In this course we focus on Nested subqueries only.

Let’s take a look at the following Select statement that uses a subquery:



The subquery (inner query) executes one and returns on or more values and then the outer query executes using the value(s) returned from the subquery.

Execution example:



Executes First

Inner query (subquery) executes first and returns value(s)

Select StaffFirstName, StaffLastName from staff

Where PositionID = (1)

\*\*in this case the subquery returns a value of 1. PositionID for ‘Dean’ is 1.

The outer query then executes using the results of the subquery as criteria for the where clause. The results of the Select executing are:

StaffFirstName StaffLastName

Hugh Guy

This was a two step solution where first the subquery returned the PositionID for the PositionDescription of ‘Dean’ and then Selected all the Staff Names that had that PositionID.

It is important to note that this example used an = in front of the subquery. If our subquery had returned more than one value it would have cause an error when the statement was executed. You can only compare one value to another one value when using an = operator.

**What if the subquery returns multiple values?**

One of the most common uses of a subquery is to retrieve a list of values that is used as a list of acceptable or non-acceptable criteria by the outer query.

To see all the student Names that have made payments

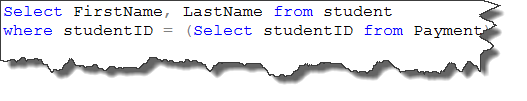
Select StudentFirstName, StudentLastName from student where

studentID IN (Select studentID from Payment)

Breaking it down…

1. Subquery executes first selecting all the studentID’s in the payment table.
2. Those The outer query executes using those StudentID’s as part of the criteria in the where clause.

What happens when you run this Select statement?

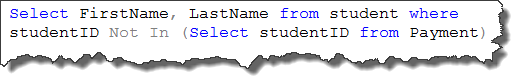


**ERROR**

Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.

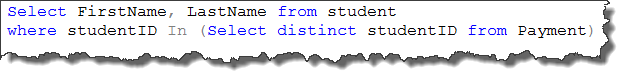
Remember, when using = there must be only one value on each side. If the subquery might return more than one value then you must use IN

What if I wanted to see all students that have never made a payment?



**Extra Geek point!**

In the previous example when the subquery selects all the StudentID’s from the payment table you end up with a lot of duplicate StudentID values. This may seem inefficient when you could write the subquery like this:



Remember, distinct only returns unique values and not duplicate. Actual testing reveals that using distinct in the subquery to return unique values is actually slower. Keep in mind, that there is some overhead in the selecting of only distinct values VS simply selecting all of them. ☺

**Any, Some and All Operators**

So far our subqueries have returned values to be used as criteria for an exact match by the outer query.

Where StudentID IN (Select StudentID …..)

What if we want something other than an exact match?

The ANY or SOME (for SQL they are interchangeable) operators lets you use even more operators (>, <, >=, <=, <>).

SOME compares against any of the values retrieved by the subquery

Where Grade > SOME (Select Grade.....)

If the Grade being selected by the outer query is greater than ***any*** of the grades returned by the subquery then the record is returned.

The ALL operator compares against all of the values retrieved by the subquery

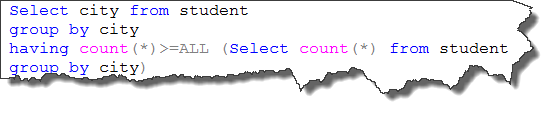
Where Grade > ALL (Select Grade.....)

If the Grade being selected by the outer query is greater than ***all*** of the grades returned by the subquery then the record is returned.

We can use these operators to perform some more complicate selects.

Subqueries can be used in many places where you need to retrieve a set of values for criteria.

Select the City that has the highest number of students in it.



In this example we used the subquery as criteria in the having clause.

Breaking it down:

1. Subquery executes first and returns values.

Subquery returns these values

Select city from student

group by city

having count(\*)>=ALL (1,2,6,2,1,1,1,1,1)

1. Outer query executes comparing the count of students in each city to see which count(s) are greater/equal than or equal to all the ones in the subquery. The record from the outer query that is equal to **the highest one in the subquery is the city with the most students.**

**Correlated Subqueries**

We do not cover correlated subqueries in this course. However, here is a brief introduction ☺

Correlated subqueries are a bit different than the queries we have looked at so far. These were called nested subqueries.

Nested subqueries had the subquery query execute once and then the outer query executed once.

Correleated subqueries work like this:

1. The outer query retrieves a record
2. The subquery executes using data passed to it from the record retrieved by the outer query.
3. The subquery passes databack to the outer query. This data is used by the outer query to finish processing the record.

The process repeats for every record processed by the outerquery.

The hallmark of the correlated subquery is that the subquery references (uses data from) a table specified in the outer query’s from clause.