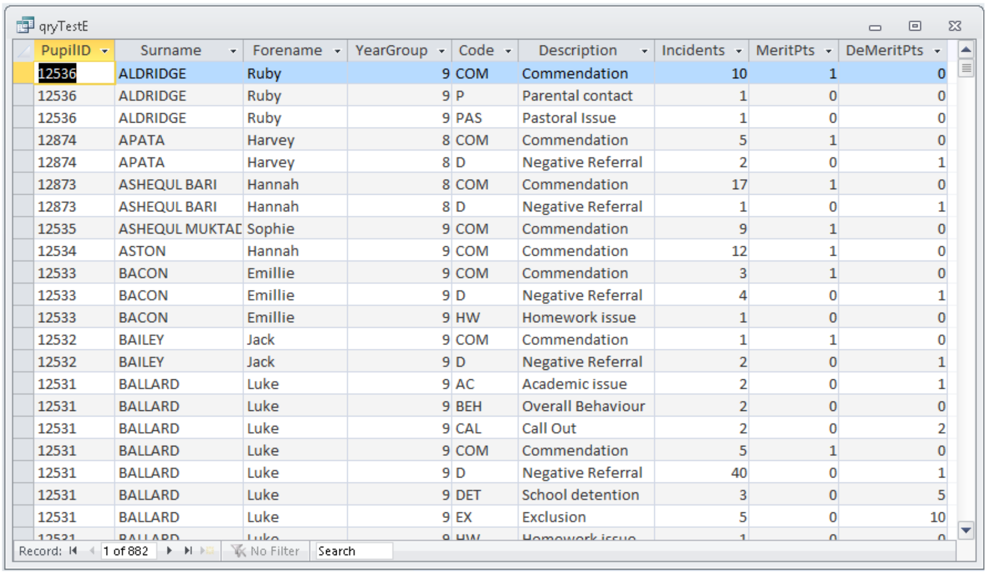
Speed Tests - Optimise Queries

Allen Browne has an excellent web page devoted to various methods of improving query performance:
http://allenbrowne.com/QueryPerfIssue.html

I thought it would be helpful to others to illustrate his suggestions by performing a series of speed tests showing the effect of each suggested change.

To reduce file size, the queries are based on cut down versions of 3 tables from the DEMO version of my **School Data Analyser** application. All data is for fictitious students in a fictitious school.

The aim of the query is to get the count of each type of pastoral incident recorded for each student in the year 2018. The query is also filtered to those students whose date of birth was in the year 2005.


qryTestE

PupilID	Surname	Forename	YearGroup	Code	Description	Incidents	MeritPts	DeMeritPts
12536	ALDRIDGE	Ruby	9	COM	Commendation	10	1	0
12536	ALDRIDGE	Ruby	9	P	Parental contact	1	0	0
12536	ALDRIDGE	Ruby	9	PAS	Pastoral Issue	1	0	0
12874	APATA	Harvey	8	COM	Commendation	5	1	0
12874	APATA	Harvey	8	D	Negative Referral	2	0	1
12873	ASHEQUL BARI	Hannah	8	COM	Commendation	17	1	0
12873	ASHEQUL BARI	Hannah	8	D	Negative Referral	1	0	1
12535	ASHEQUL MUKTAD	Sophie	9	COM	Commendation	9	1	0
12534	ASTON	Hannah	9	COM	Commendation	12	1	0
12533	BACON	Emillie	9	COM	Commendation	3	1	0
12533	BACON	Emillie	9	D	Negative Referral	4	0	1
12533	BACON	Emillie	9	HW	Homework issue	1	0	0
12532	BAILEY	Jack	9	COM	Commendation	1	1	0
12532	BAILEY	Jack	9	D	Negative Referral	2	0	1
12531	BALLARD	Luke	9	AC	Academic issue	2	0	1
12531	BALLARD	Luke	9	BEH	Overall Behaviour	2	0	0
12531	BALLARD	Luke	9	CAL	Call Out	2	0	2
12531	BALLARD	Luke	9	COM	Commendation	5	1	0
12531	BALLARD	Luke	9	D	Negative Referral	40	0	1
12531	BALLARD	Luke	9	DET	School detention	3	0	5
12531	BALLARD	Luke	9	EX	Exclusion	5	0	10
12531	BALLARD	Luke	9	HW	Homework issue	1	0	0

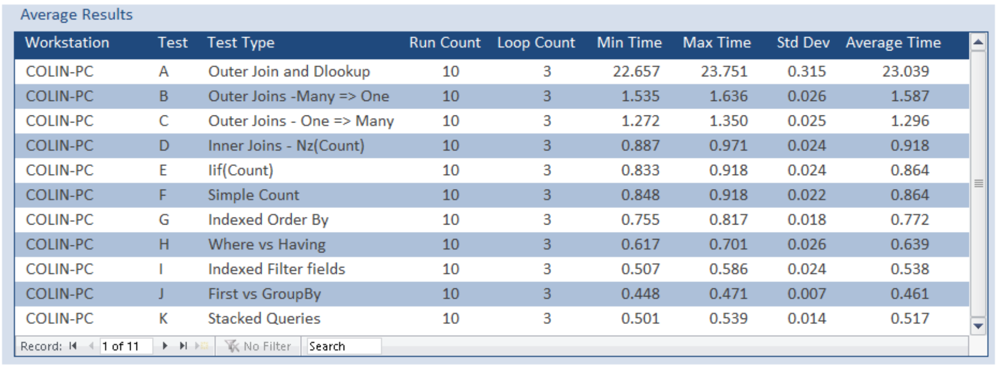
Record: 1 of 882 No Filter Search

There are 11 versions of the query with varying amounts of optimisation starting with a (deliberately) badly designed query and ending with the most optimised. All queries return the same records (total = 882) but the times should get progressively faster each time (except for the final stacked queries test).

Each test is run several times to reduce natural variations caused by other processes that may be running in the background. The total time recorded is for the set number of loops. By default, the number of loops = 3

The fields used in each table to filter and sort the data are indexed to speed up searches:
The indexed fields are Surname, Forename, DateOfBirth, DateOfIncident

The average times recorded after running each set of tests 10 times was as follows:

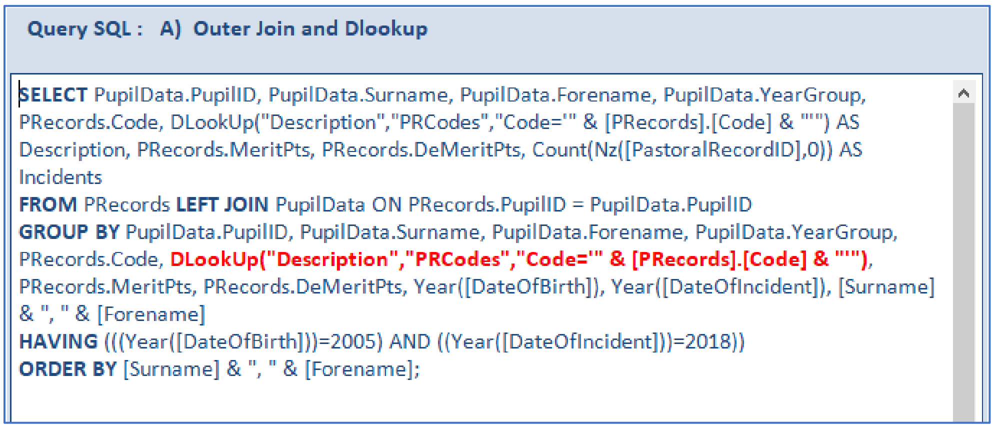

Average Results

Workstation	Test	Test Type	Run Count	Loop Count	Min Time	Max Time	Std Dev	Average Time
COLIN-PC	A	Outer Join and Dlookup	10	3	22.657	23.751	0.315	23.039
COLIN-PC	B	Outer Joins -Many => One	10	3	1.535	1.636	0.026	1.587
COLIN-PC	C	Outer Joins - One => Many	10	3	1.272	1.350	0.025	1.296
COLIN-PC	D	Inner Joins - Nz(Count)	10	3	0.887	0.971	0.024	0.918
COLIN-PC	E	Iif(Count)	10	3	0.833	0.918	0.024	0.864
COLIN-PC	F	Simple Count	10	3	0.848	0.918	0.022	0.864
COLIN-PC	G	Indexed Order By	10	3	0.755	0.817	0.018	0.772
COLIN-PC	H	Where vs Having	10	3	0.617	0.701	0.026	0.639
COLIN-PC	I	Indexed Filter fields	10	3	0.507	0.586	0.024	0.538
COLIN-PC	J	First vs GroupBy	10	3	0.448	0.471	0.007	0.461
COLIN-PC	K	Stacked Queries	10	3	0.501	0.539	0.014	0.517

Record: 1 of 11 No Filter Search

The times taken improved significantly from over 23 s originally down to about 0.46 s – almost 50 times faster

The first query uses an **outer join** between 2 tables (**PupilData / PRecords**) and a **DLookup** value from the third table (**PRCodes**). It took over 23 s to do 3 loops – VERY SLOW

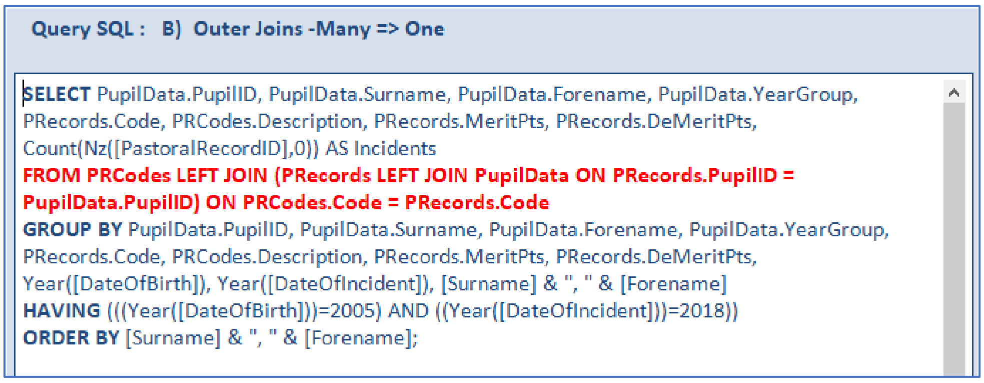

Query SQL : A) Outer Join and Dlookup

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, DLookUp("Description","PRCodes","Code='" & [PRecords].[Code] & "'") AS
Description, PRecords.MeritPts, PRecords.DeMeritPts, Count(Nz([PastoralRecordID],0)) AS
Incidents
FROM PRecords LEFT JOIN PupilData ON PRecords.PupilID = PupilData.PupilID
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, DLookUp("Description","PRCodes","Code='" & [PRecords].[Code] & "'"),
PRecords.MeritPts, PRecords.DeMeritPts, Year([DateOfBirth]), Year([DateOfIncident]), [Surname]
& ", " & [Forename]
HAVING (((Year([DateOfBirth]))=2005) AND ((Year([DateOfIncident]))=2018))
ORDER BY [Surname] & ", " & [Forename];
```

Running domain functions such as **DLookup** in a query is VERY slow as the operation must be performed in turn on each row in the query. It also wastes resources as additional connections have to be made to the data file.

The **query execution plan** involves a huge number of steps as each record is checked in turn

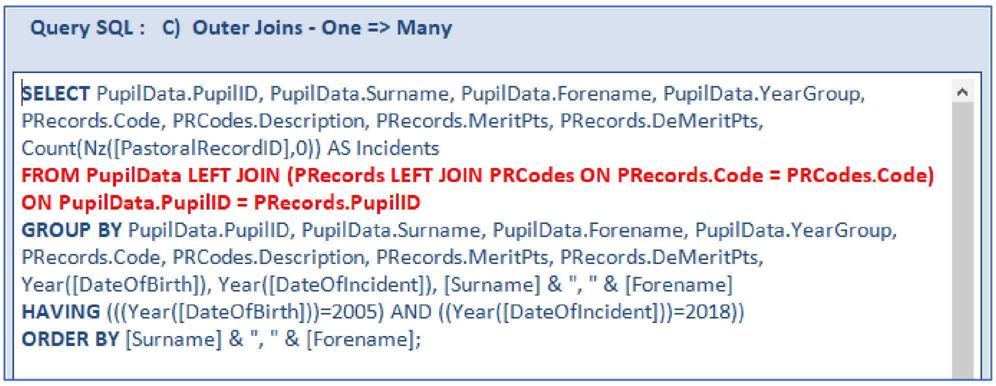
In this case, the **domain function** is totally unnecessary as the same result can be obtained using a **second join**


Query SQL : B) Outer Joins -Many => One

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Count(Nz([PastoralRecordID],0)) AS Incidents
FROM PRCodes LEFT JOIN (PRecords LEFT JOIN PupilData ON PRecords.PupilID =
PupilData.PupilID) ON PRCodes.Code = PRecords.Code
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Year([DateOfBirth]), Year([DateOfIncident]), [Surname] & ", " & [Forename]
HAVING (((Year([DateOfBirth]))=2005) AND ((Year([DateOfIncident]))=2018))
ORDER BY [Surname] & ", " & [Forename];
```

In this case, the joins go from the many side of the main PRecords table : **PRCodes -> PRecords -> PupilData**
Although the join direction is not the best choice, the time taken is dramatically reduced to about 1.6 s.

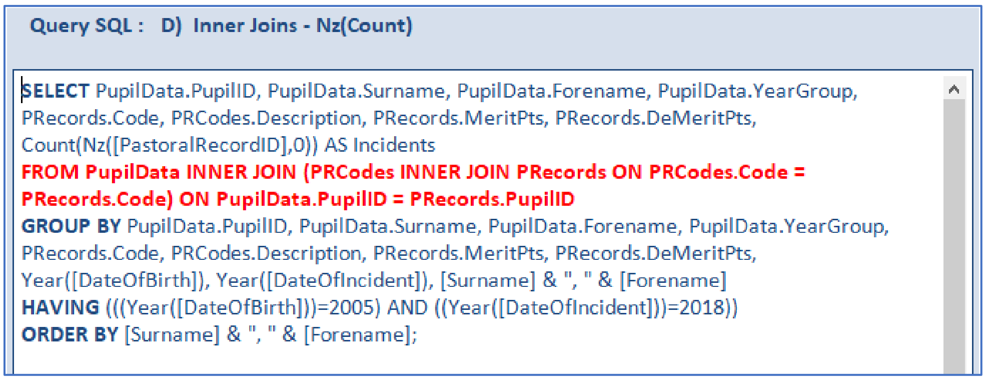
In the third query, the direction of the joins is reversed (one to many): **PupilData -> PRecords -> PRCodes**.


Query SQL : C) Outer Joins - One => Many

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Count(Nz([PastoralRecordID],0)) AS Incidents
FROM PupilData LEFT JOIN (PRecords LEFT JOIN PRCodes ON PRecords.Code = PRCodes.Code)
ON PupilData.PupilID = PRecords.PupilID
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Year([DateOfBirth]), Year([DateOfIncident]), [Surname] & ", " & [Forename]
HAVING (((Year([DateOfBirth]))=2005) AND ((Year([DateOfIncident]))=2018))
ORDER BY [Surname] & ", " & [Forename];
```

This is a more efficient process for Access to manage and the time drops again to about 1.3 s.

This example has been deliberately designed so that using **inner joins** will get exactly the same records.
It always makes sense to use **inner joins** wherever possible as the constraints limit the searching required



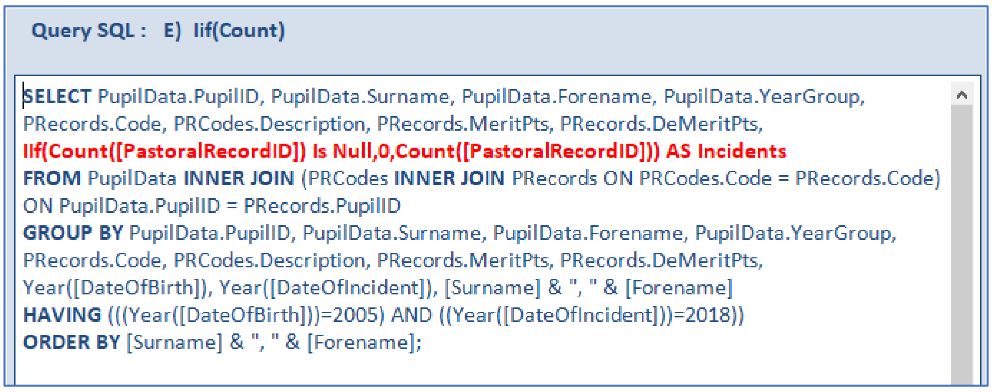
Query SQL : D) Inner Joins - Nz(Count)

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Count(Nz([PastoralRecordID],0)) AS Incidents
FROM PupilData INNER JOIN (PRCodes INNER JOIN PRecords ON PRCodes.Code =
PRecords.Code) ON PupilData.PupilID = PRecords.PupilID
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Year([DateOfBirth]), Year([DateOfIncident]), [Surname] & ", " & [Forename]
HAVING (((Year([DateOfBirth]))=2005) AND ((Year([DateOfIncident]))=2018))
ORDER BY [Surname] & ", " & [Forename];
```

Doing so further reduces the work required from the database engine and the time drops to just over 0.9 s.
All the remaining queries are based on **inner joins**.

Until now, all the aggregate totals have been based on the **VBA Nz** function: **Count(Nz([PastoralRecordID],0))**.
The Nz() function replaces Null with another value (usually a zero for numbers, or a zero-length string for text).
The new value is a **Variant** data type, and VBA tags it with a subtype: String, Long, Double, Date, or whatever.
This will affect the sort order and can lead to incorrect results in some situations

The fifth query replaces the **VBA Nz** function with the use of the **JET IIf** function:
IIf(Count([PastoralRecordID]) Is Null,0,Count([PastoralRecordID]))

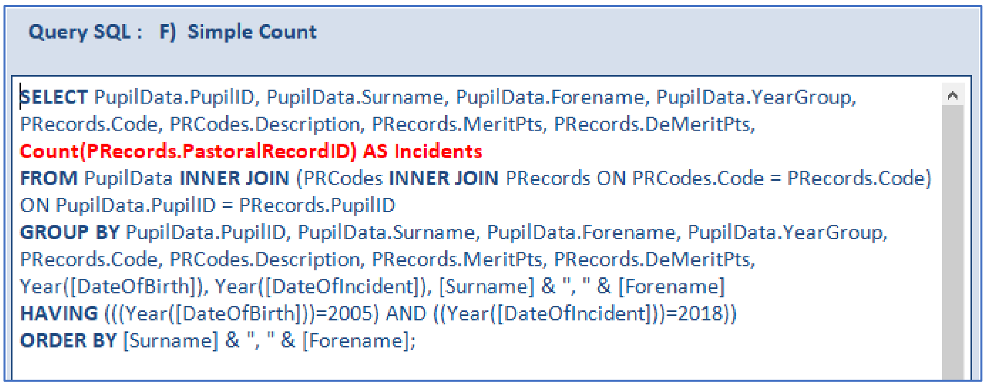


Query SQL : E) Iif(Count)

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
IIf(Count([PastoralRecordID]) Is Null,0,Count([PastoralRecordID])) AS Incidents
FROM PupilData INNER JOIN (PRCodes INNER JOIN PRecords ON PRCodes.Code = PRecords.Code)
ON PupilData.PupilID = PRecords.PupilID
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Year([DateOfBirth]), Year([DateOfIncident]), [Surname] & ", " & [Forename]
HAVING (((Year([DateOfBirth]))=2005) AND ((Year([DateOfIncident]))=2018))
ORDER BY [Surname] & ", " & [Forename];
```

This has several advantages including avoiding an unnecessary **VBA function** call.
In addition, the correct data type is retained (in this case, integer) so the column sorts correctly.
This shaves another 0.05 s off the time which has now become about 0.86 s.

However, by using **inner joins** as in this example, a **simple count** will achieve the same results



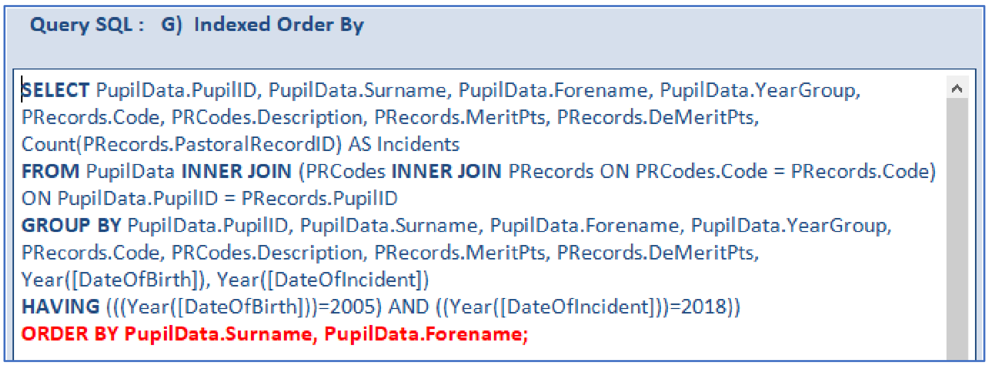
Query SQL : F) Simple Count

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Count(PRecords.PastoralRecordID) AS Incidents
FROM PupilData INNER JOIN (PRCodes INNER JOIN PRecords ON PRCodes.Code = PRecords.Code)
ON PupilData.PupilID = PRecords.PupilID
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Year([DateOfBirth]), Year([DateOfIncident]), [Surname] & ", " & [Forename]
HAVING (((Year([DateOfBirth]))=2005) AND ((Year([DateOfIncident]))=2018))
ORDER BY [Surname] & ", " & [Forename];
```

Although, the expression is simpler, the overall time is the same as before – approximately 0.86 s.

All the above queries were sorted by a **concatenated** expression: **[Surname] & ", " & [Forename]**
Doing so, prevents the database engine making use of the indexes to perform the sort.

The next query fixes that, sorting by the two **indexed fields**: **Surname** and **Forename**.

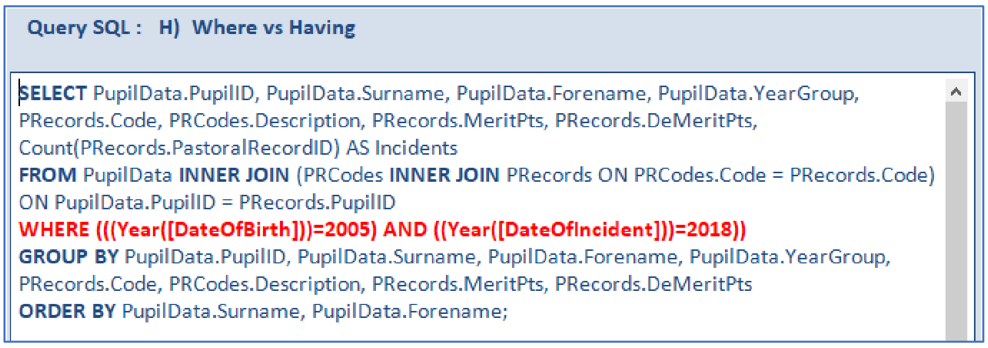

Query SQL : G) Indexed Order By

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Count(PRecords.PastoralRecordID) AS Incidents
FROM PupilData INNER JOIN (PRCodes INNER JOIN PRecords ON PRCodes.Code = PRecords.Code)
ON PupilData.PupilID = PRecords.PupilID
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Year([DateOfBirth]), Year([DateOfIncident])
HAVING (((Year([DateOfBirth]))=2005) AND ((Year([DateOfIncident]))=2018))
ORDER BY PupilData.Surname, PupilData.Forename;
```

Doing so, further reduces the time required to about 0.77 s – approximately 0.1 s faster
Whilst the query is now running well, further improvements can still be made.

Aggregate queries (those with a **GROUP BY** clause) can have both a **WHERE** clause and a **HAVING** clause.
The **WHERE** is executed first - **before aggregation**; the **HAVING** is executed **afterwards** - when the counts have been calculated. Therefore, in some cases (though not always), it can be faster to use **WHERE**

The next query changes the **HAVING** clause to **WHERE** and the time drops to 0.64 s (another 0.13 s faster)

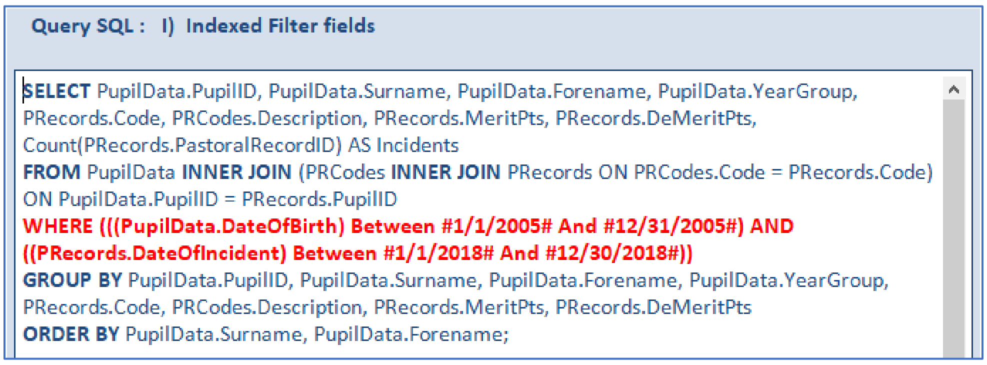

Query SQL : H) Where vs Having

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Count(PRecords.PastoralRecordID) AS Incidents
FROM PupilData INNER JOIN (PRCodes INNER JOIN PRecords ON PRCodes.Code = PRecords.Code)
ON PupilData.PupilID = PRecords.PupilID
WHERE (((Year([DateOfBirth]))=2005) AND ((Year([DateOfIncident]))=2018))
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts
ORDER BY PupilData.Surname, PupilData.Forename;
```

See this separate website article for detailed speed tests based on HAVING vs WHERE

However, although the **WHERE** clause looks simple to run, it is not using the **indexing** of the two date fields.
A better result is obtained using the indexes by indicating a range of values for each of the date fields:

> **WHERE (((PupilData.DateOfBirth) Between #1/1/2005# And #12/31/2005#) AND ((PRecords.DateOfIncident) Between #1/1/2018# And #12/30/2018#))**

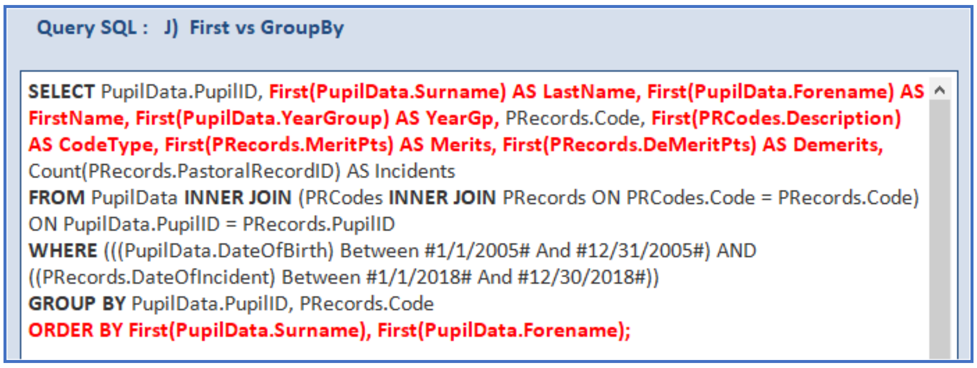

Query SQL : I) Indexed Filter fields

```
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts,
Count(PRecords.PastoralRecordID) AS Incidents
FROM PupilData INNER JOIN (PRCodes INNER JOIN PRecords ON PRCodes.Code = PRecords.Code)
ON PupilData.PupilID = PRecords.PupilID
WHERE (((PupilData.DateOfBirth) Between #1/1/2005# And #12/31/2005#) AND
((PRecords.DateOfIncident) Between #1/1/2018# And #12/30/2018#))
GROUP BY PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PRecords.Code, PRCodes.Description, PRecords.MeritPts, PRecords.DeMeritPts
ORDER BY PupilData.Surname, PupilData.Forename;
```

This further reduces the time by another 0.1 s down to about 0.54 s. Every little helps!

All the above queries have used the default arrangement, grouping all 4 fields from the **PupilData** table.

However the **PupilID** field is the unique primary key field. There is no need to group by other fields in that table. Instead optimise the query by choosing **First** instead of **Group By** in the **Total** row under the other fields. Similarly for the other fields not required for the grouping in the other 2 tables.



Query SQL : J) First vs GroupBy

```
SELECT PupilData.PupilID, First(PupilData.Surname) AS LastName, First(PupilData.Forename) AS
FirstName, First(PupilData.YearGroup) AS YearGp, PRecords.Code, First(PRCodes.Description)
AS CodeType, First(PRecords.MeritPts) AS Merits, First(PRecords.DeMeritPts) AS Demerits,
Count(PRecords.PastoralRecordID) AS Incidents
FROM PupilData INNER JOIN (PRCodes INNER JOIN PRecords ON PRCodes.Code = PRecords.Code)
ON PupilData.PupilID = PRecords.PupilID
WHERE (((PupilData.DateOfBirth) Between #1/1/2005# And #12/31/2005#) AND
((PRecords.DateOfIncident) Between #1/1/2018# And #12/30/2018#))
GROUP BY PupilData.PupilID, PRecords.Code
ORDER BY First(PupilData.Surname), First(PupilData.Forename);
```

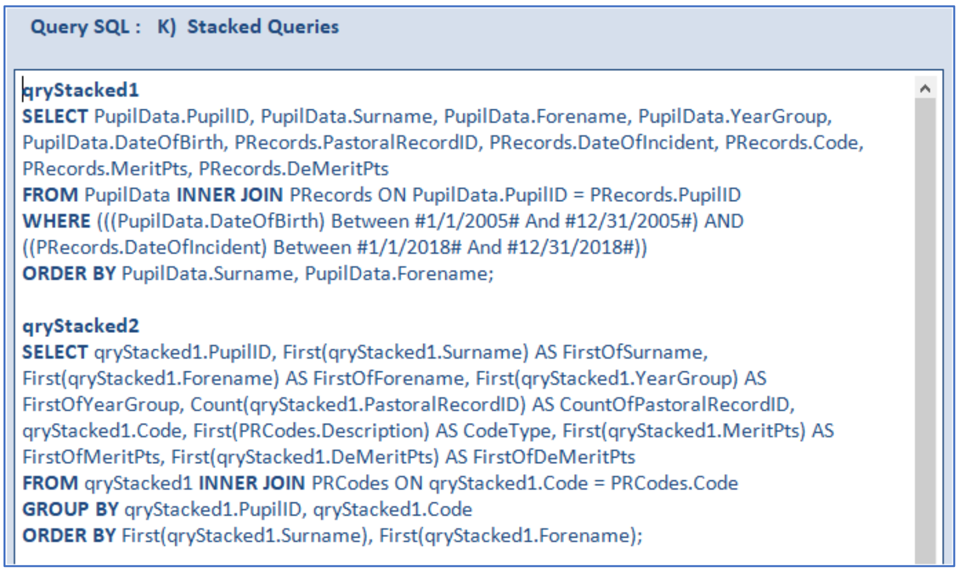
This results in a further **significant reduction** in time to **0.46 s**.
The end result is now **50 times faster** than the original **23 s**!

Using **First** allows the database engine to return the value from the first matching record, without needing to group by the field. In the query above I have used aliases for the fields now based on **First**.

Allen Browne also points out another benefit if you are grouping by **Memo / Long Text** fields

*If you GROUP BY a memo (Notes in the example), Access compares only the first 255 characters, and the rest are **truncated**! By choosing First instead of Group By, JET is free to return the entire memo field from the first match. So not only is it more efficient; it actually solves the problem of memo fields being chopped off.*

Both **stacked queries** and **subqueries** are often useful in Access though both are normally slower than using a single query where that is achievable. As a test, I also created a **stacked query** version of test J.
The first query **qryStacked1** filters the records in PupilData & PRecords for the required date ranges
The second query **qryStacked2** is an aggregate query based on that



Query SQL : K) Stacked Queries

```
qryStacked1
SELECT PupilData.PupilID, PupilData.Surname, PupilData.Forename, PupilData.YearGroup,
PupilData.DateOfBirth, PRecords.PastoralRecordID, PRecords.DateOfIncident, PRecords.Code,
PRecords.MeritPts, PRecords.DeMeritPts
FROM PupilData INNER JOIN PRecords ON PupilData.PupilID = PRecords.PupilID
WHERE (((PupilData.DateOfBirth) Between #1/1/2005# And #12/31/2005#) AND
((PRecords.DateOfIncident) Between #1/1/2018# And #12/31/2018#))
ORDER BY PupilData.Surname, PupilData.Forename;

qryStacked2
SELECT qryStacked1.PupilID, First(qryStacked1.Surname) AS FirstOfSurname,
First(qryStacked1.Forename) AS FirstOfForename, First(qryStacked1.YearGroup) AS
FirstOfYearGroup, Count(qryStacked1.PastoralRecordID) AS CountOfPastoralRecordID,
qryStacked1.Code, First(PRCodes.Description) AS CodeType, First(qryStacked1.MeritPts) AS
FirstOfMeritPts, First(qryStacked1.DeMeritPts) AS FirstOfDeMeritPts
FROM qryStacked1 INNER JOIN PRCodes ON qryStacked1.Code = PRCodes.Code
GROUP BY qryStacked1.PupilID, qryStacked1.Code
ORDER BY First(qryStacked1.Surname), First(qryStacked1.Forename);
```

The average time for 10 tests was 0.517 s – about 0.5 s SLOWER than the single query equivalent in test J
As expected, **running stacked queries is noticeably slower**

NOTE:
If anyone can see ways in which the above query can be further optimised, please do let me know!

View Query Execution Plans

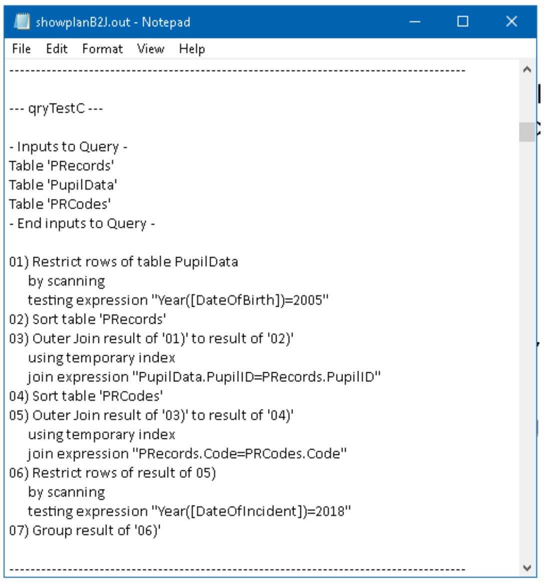
You can use the **JET ShowPlan** feature to view the query execution plans for your queries.
By doing so, you can often obtain useful information to assist with the process of optimising your queries
Using this feature creates a text file **ShowPlan.out** which can be viewed in Notepad

For further information, see this article **ShowPlan – Go Faster**

I have attached three **ShowPlan** files for the above tests

- **ShowPlanA.out** – this lengthy file just covers **Test A** which uses a DLookup.
  This should help explain why using domain functions in a query will ALWAYS be SLOW
- **ShowPlanB2J.out** – this covers all the other main tests: **Tests B => Test J**
- **ShowPlanStacked.out** – this just covers the **stacked query** version used in **Test K**

For example, this is the query execution plan for test C. It is the shortest execution plan by a long way



```
--------------------------------------------------------------------------------

--- qryTestC ---

- Inputs to Query -
Table 'PRecords'
Table 'PupilData'
Table 'PRCodes'
- End inputs to Query -

01) Restrict rows of table PupilData
    by scanning
    testing expression "Year([DateOfBirth])=2005"
02) Sort table 'PRecords'
03) Outer Join result of '01)' to result of '02)'
    using temporary index
    join expression "PupilData.PupilID=PRecords.PupilID"
04) Sort table 'PRCodes'
05) Outer Join result of '03)' to result of '04)'
    using temporary index
    join expression "PRecords.Code=PRCodes.Code"
06) Restrict rows of result of 05)
    by scanning
    testing expression "Year([DateOfIncident])=2018"
07) Group result of '06)'

--------------------------------------------------------------------------------
```

Attached Items

OptimiseQueries.accdb
ShowPlan.out (x3)
Optimise Queries.pdf

Colin Riddington *www.mendipdatasystems.co.uk*