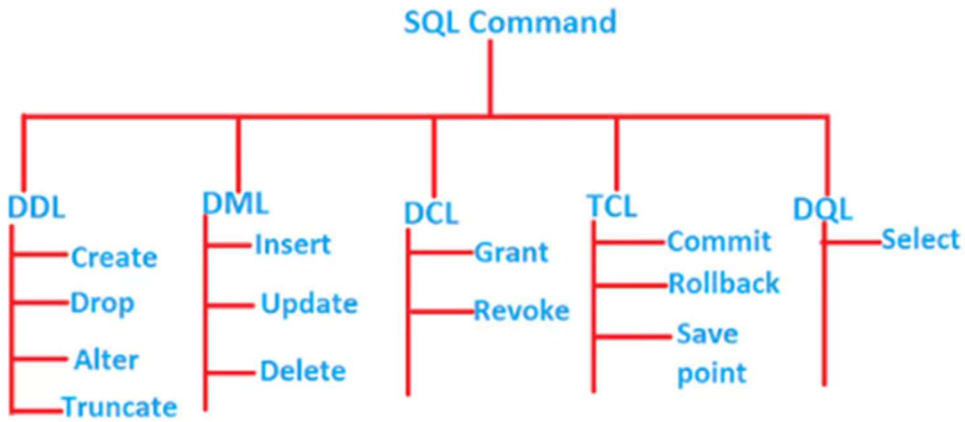
SQL Summary Advanced 2026 – Version 6 for Ms Access.

Structure Query Language (SQL) is the most popular RDBMS (Relational Data Base Management System) language.

A database supports CRUD (Create, Read, Update, Deletion) of tables and data within a database. SQL supports this and is the principal language for **creating, accessing, manipulating and managing databases** (not only querying).

SQL originating in 1970 at IBM by Donald Chamberlin and Raymond Boyce was based on a theoretical database design model proposed by Edgar Codd – he invented the relational algebra behind SQL. Different implementations of Codd's model exist which explains the variations in different SQL versions e.g. MYSQL, Ms Access, SQLite, SQL Server, IBM Db2 and PostgreSQL, Oracle database etc

The 5 sub languages of SQL are **DDL, DML, DQL/DRL, DCL,** and **TCL.**



Source: analyticsvidhya.com

DDL – Data Definition Language e.g. creating a table. You will notice this only if your school teaches MYSQL.

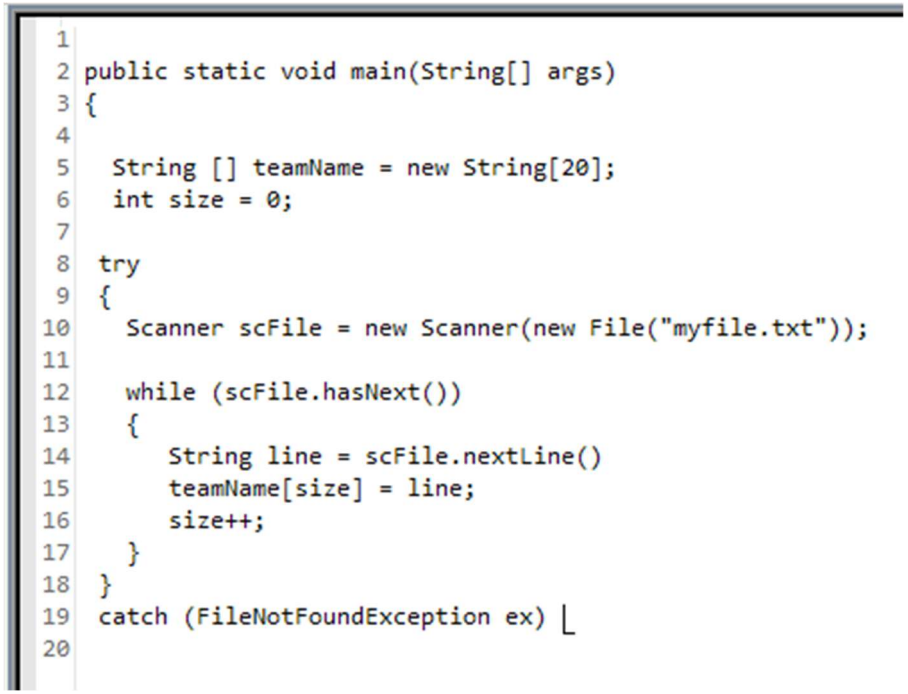
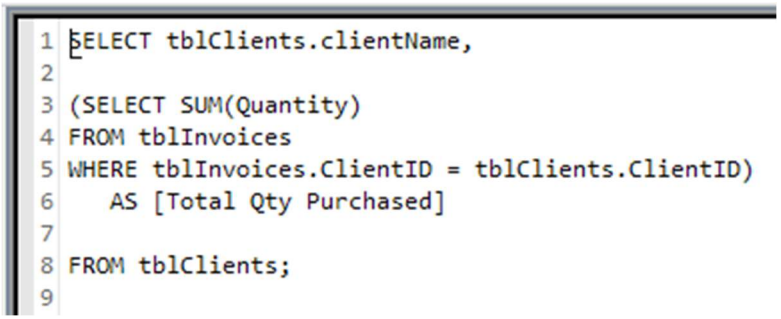
DML – Data Manipulation Language e.g. changing data already in a table. (in the grade 12 curriculum)

DCL – Data Control Language e.g. granting rights to a user to view data in a table.

TCL – Transactional Control Language e.g. controlling and protecting the integrity of the database

DQL – Data Query Language e.g. finding data within the database using the SELECT statement(in the grade 12 SAGS)

Concept – Java: Never a loop, SQL: Always a loop

Java – Never a loop	SQL – Always a loop
	
Java assumes you want one piece of data. You have to code loops by hand. Java never assumes you want ALL the records in a text file (or database). In line 12 you must create a loop and force Java to read each and every line until there are no more lines left in the text file (until it reaches the end-of-file marker). The role of the line counter (size) in critical.	SQL assumes you want ALL the data. The loop is automatic. In this embedded query we are asking for the client's name plus the total amount they purchased. By default, SQL will loop and give us all the names in the table unless we use conditional statements to filter out those records we don't want. Therefore, in our SQL we are interested in **filtering out** data. We are in the business of limiting the data SQL gives us. We do this with the WHERE clause, the HAVING clause and the use of the logical operators AND, OR and NOT. In line 5 we are telling SQL to only give us the names where the clientID in one table is equal to the ClientID in another table.

Structure of DQL. Queries that search and return the data we ask for. (Data and the table structure are not altered)

DQL – The clauses	SQL operators	
SELECT <field(s)> **FROM <table(s)>** **WHERE condition(s)** **GROUP BY expression** **HAVING condition – only works with GROUP BY** **ORDER BY expression**	**Arithmetic** **Comparison** **Logical** **Concatenation** **Special**	**+ - * / MOD ^** **< <= > >= = (equal) <> (not equal)** **NOT, AND, OR** **&** (joining text) **Is NULL** or **is NOT NULL** **LIKE** **BETWEEN val1 and val2** **IN (, , ,)**

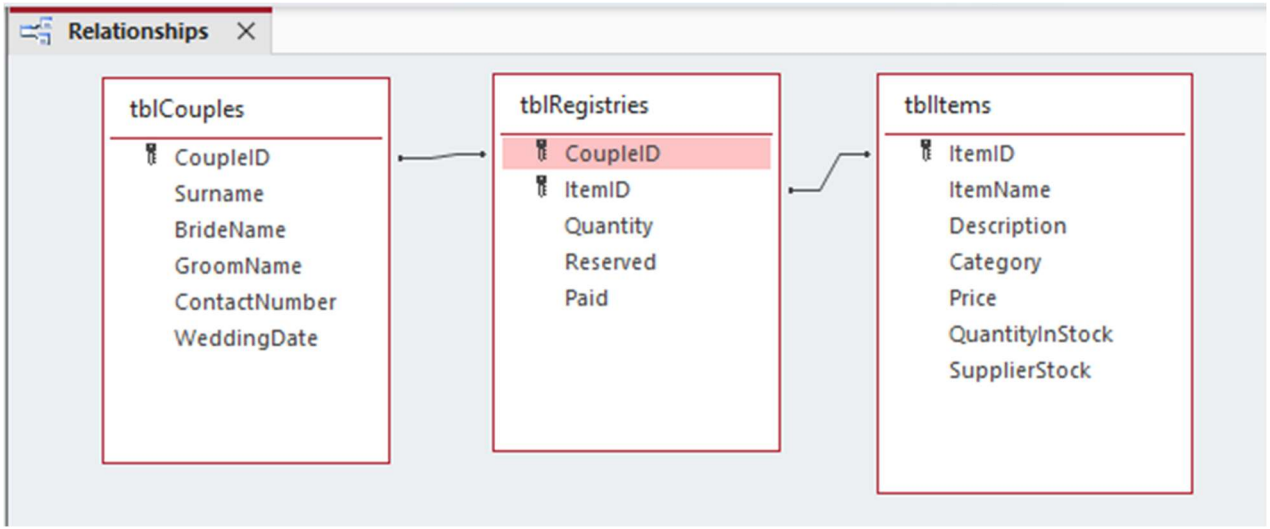
Structure of DML. Queries that alter data in a table (insert, delete or edit existing records) (Table structure is not altered)

INSERT . . . INTO . . . . VALUES

UPDATE . . . . SET

DELETE . . . . FROM

Example: The Wedding Registry.



In Ms Access the primary and foreign keys are indicated by the small key icon.

The relationships between the various tables are indicated by the joining line.

The registry table uses a **composite primary key** – two keys which when used together uniquely identify each record – hence the two key icons in tblRegistries.

The table tblRegistries offers a many-to-many relationship.

Example: A wedding registry database where couples can place items (gifts) into a registry that their own wedding guests can choose from. This gives the guests an idea of what the couples need. Guests can reserve and pay for the items of their choice which also avoids duplicate gifts.

This wedding registry database has three tables. The couples table and the items table are static. Couples can be added or deleted as can items, but other than that there is little movement.

The dynamic part of the database (the transactional part) is the joining/linking table of tblRegistries. This achieves a **many-to-many relationship,** between tblCouples and tblItems i.e. a couple can choose many gifts, and a gift can be chosen by many couples.

Revisiting aggregate functions.

Aggregate functions; MAX(), MIN(), AVG(), SUM(), COUNT()

SELECT aggregate function(field) . . . e.g. SELECT MIN(SizeKB) . . . . SELECT AVG(TotalPages) . . . . SELECT SUM(Cost)

Works with **columns only** – this does NOT work . SELECT AVG(test1 + test2 + test3) AS Average FROM markbook; X

NOTE: Aggregate functions only return a **single value** based on the values in a **column** – no other details. Each aggregate function needs its own SELECT statement. If you need more information, you must use the aggregate function in an **embedded query (subquery)** to get the information you are missing i.e. there are two SELECT statements. Aggregate functions are used a lot with the GROUP BY clause.

Examples:

The SQL **SUM( )** function is an aggregate function that calculates and returns the total sum of values in a numeric column.

SUM Data Types: Works only with numeric data types such as INTEGER, FLOAT, DECIMAL, and REAL.
NULL Handling: By default, it ignores NULL values. If every value in the selected column is NULL or the result set is empty, it returns NULL

1
`SELECT SUM(columnName)`
`FROM tableName;`
	2
	`SELECT SUM(Salary)`
	`FROM Employees;`
		3
		`SELECT SUM(Salary)`
		`FROM Employees`
		`WHERE Department = 'Sales';`

Reminder: The field you GROUP BY must be in the SELECT statement if it is **not** an aggregate function.	**4**
	`SELECT category_id, SUM(quantity)`
	`FROM OrderItems`
	`GROUP BY category_id;`
		5
		`SELECT user_id, SUM(views)`
		`FROM videos`
		`GROUP BY user_id`
		`HAVING SUM(views) > 10000;`

The SQL **COUNT( )** aggregate function is used to return the number of rows that match specified criteria in a query.

The behavior of the COUNT() function changes based on the argument provided:

- COUNT(*): Counts all rows in a table, including those with NULL values and duplicates.
- COUNT(column name): Counts only the non-null values in the specified column i.e. column name

Usage with Clauses.

COUNT() is frequently paired with other SQL clauses to refine results:

- WHERE: Filters individual rows **before** the count is performed.
- GROUP BY: Groups rows with the same values into summary rows (different result sets), allowing you to find counts for specific categories (e.g., number of customers **per city**).
- HAVING: Filters the result set **after** the grouping has occurred, typically based on the aggregate count (e.g. HAVING COUNT(*) > 5).

6
`SELECT COUNT(ProductName)`
`FROM Products;`
	7
	`SELECT COUNT(price) AS [Unique prices]`
	`FROM (SELECT DISTINCT price`
	`FROM tblProducts);`
	
	`// must be an embedded query`
		8
		`SELECT COUNT(ProductID)`
		`FROM Products`
		`WHERE Price > 20;`

REMINDER: Two examples alongside, 9 and 10, count all rows (records) including null values and duplicates.	**9**
	`SELECT COUNT(*) AS [Number records]`
	`FROM Products;`
		10
		`SELECT COUNT(*) AS [Num records], CategoryID`
		`FROM Products`
		`GROUP BY CategoryID;`

NOTE: Example alongside number 11, does not work as aggregate functions only return a single value and cannot return values from other fields.	**11** `SELECT COUNT(*), itemName` X `FROM tblItems;`	

The concept of a Result Set (rs). Also called a dataset.

The result set (dataset) is the output of a DQL SQL query, but it can be more than that – it can be viewed as a static read-only subset of the database from which we can create a new database table or create an array of objects in Java that we can work with.

11

```
SELECT ItemName, category, price, quantityinStock AS [Quantity]
FROM tblItems
WHERE description LIKE "*wood*";
```

In **Java** we could create a **class** called WoodenGifts. We could create an array of WoodenGift objects based on the result set alongside. We can sort, count, add, subtract etc the wooden items in the array of wooden items using our Java code.

Copy Of Query1

ItemName	category	price	Quantity
Picture Frame	Decor	R250,00	60
Shoe Rack	Furniture	R400,00	15
Dining Table	Furniture	R5 000,00	12
Coffee Table	Furniture	R1 200,00	20
Kitchen Knife Set	Kitchenware	R1 500,00	18
Chopping Board	Kitchenware	R300,00	50
Cooking Spoon Set	Kitchenware	R250,00	40
Cookbook Stand	Kitchenware	R250,00	50
Trolley	Furniture	R700,00	30

12

```
SELECT
    surname,
    weddingdate,
    LEFT(surname, 3) & INT(rnd (coupleID) * 10) + 1 & LEN(groomname)
AS [Reference Number]
FROM
    tblCouples
ORDER BY
    surname;
```

This query creates a unique reference number.

In **Java** we can create a **Surname class** based on the result set (rs) alongside. This array would provide unique functionality based on the generated reference number (a reference number not found in the original database table)

Copy Of D1

surname	weddingdate	Reference Number
Anderson	2026/06/07	And106
Botha	2026/06/07	Bot37
Brown	2026/11/07	Bro84
Du Plessis	2026/06/21	Du 46
Hirst	2026/09/13	Hir35
Jackson	2026/09/24	Jac105
Jansen	2026/04/04	Jan75
Johnson	2026/08/01	Joh74
Jones	2026/05/23	Jon55
King	2026/10/19	Kin16
Lopez	2026/07/19	Lop66
Mahlobo	2026/09/20	Mah75

13 `SELECT * FROM tblItems;` This query seems pointless until you realize you use it to duplicate your **Items table** into your Java code.	This query returns a result set that is the whole table. This is the query you use to copy your database tables into your PAT. By creating a Java class called “Items” you can create an array of Item objects that matches the original table in the database exactly.
14 `SELECT * FROM tblCouples;` This query seems pointless until you realize you use it to duplicate your **Couples table** in your Java code.	This query returns a result set that is the whole Couples table. This is the query you use to copy your database tables into your PAT. By creating a class called “Couples” you can create an array of Couple objects in your Java code.

The process of using the result set to create arrays of objects in your Java code is outlined on the next page.

How to use the "result set". This is only an overview of the process – some detail is not evident here. Not for the practical exam.

15

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

private static Connection con = null;
private static Statement state = null;
private ResultSet rs = null;
private static Couples[ ] couplesArr = new Couples[100];
private static int size = 0;

public RegistryManager()
{
   getConnection();
   rs = null;

   try
   {
       state = con.createStatement( );
       rs = state.executeQuery (" Select * FROM tblCouples");

       while(rs.next ( ) )
       {
          int coupleID = rs.getInt("CoupleID");
          String surname = re.getString("Surname");
          String brideName = rs.getString("BrideName");
          String groomName = rs.getString(GroomName");
          String contactNumber = rs.getString("ContactNumber);
          String weddingDate = rs.getString("WeddingDate")
       } // end while loop. The code below can be done in one step.
       Couples c = new Couples (coupleID, surname, bridgname, groomName,
         contactNumber, weddingDate);
       couplesArr[size] = c;
       size++;
   }
} catch etc etc etc
```

Import the SQL libraries for the connection, driver, result set and the SQL statement.

Create your global variables. One for the SQL connection, one for the SQL statement and one for the result set.
Create an array of Couples objects.
Create the counter and initialize to zero.

Constructor method of the manager class.
Connect to the SQL database using the driver.
NB. Reset the result set back to null.

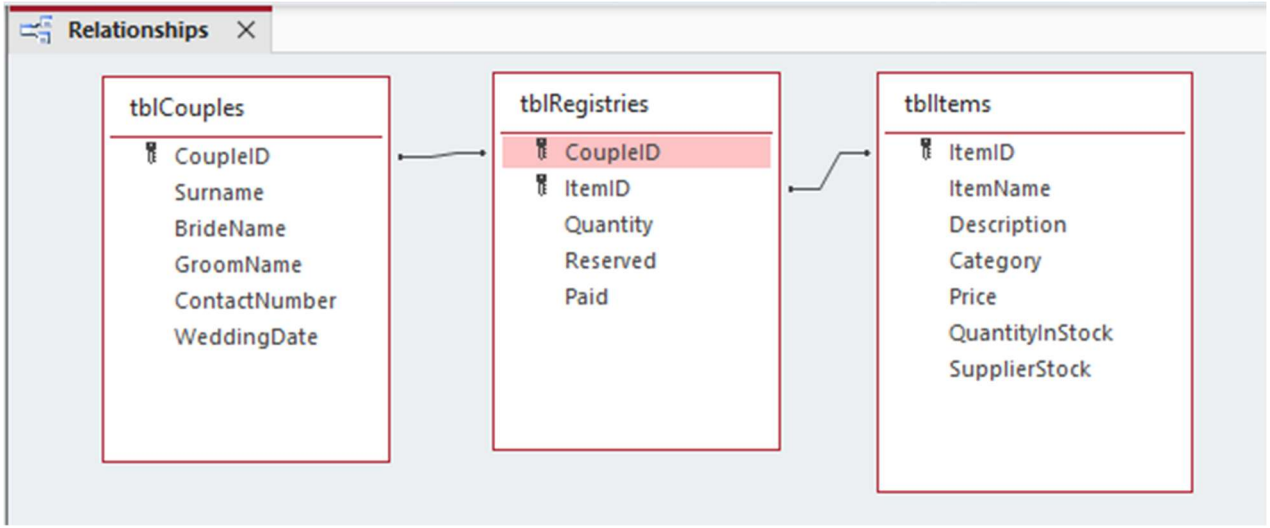
try . . . catch as per text files.
Execute the query and store the result set.

Loop through the result set until finished.

Read in fields from the result set. Most will be String.
Correct order, correct number, correct datatype are mission critical.

Create a standalone Couples object.
Copy the standalone Couples object into the next index in the array.
Increase the counter.

Harvesting information from more than one table.



In Ms Access the primary and foreign keys are indicated by the small key icon.

The relationships between the various tables are indicated by the joining line.

The registry table uses a **composite primary key** – two keys which when used together uniquely identify each record – hence the two key icons in tblRegistries.

The table tblRegistries offers a many-to-many relationship.

Example: A wedding registry database where couples can place items (gifts) into a registry that their own wedding guests can choose from. This gives the guests an idea of what the couples need. Guests can reserve and pay for the items of their choice which also avoids duplicate gifts.

This wedding registry database has three tables. The couples table and the items table are static. Couples can be added or deleted as can items, but other than that there is little movement.

The dynamic part of the database (the transactional part) is the joining/linking table of tblRegistries. This achieves a **many-to-many relationship,** between tblCouples and tblItems i.e. a couple can choose many gifts, and a gift can be chosen by many couples.

You will notice that tblRegistries has a **composite primary key** i.e. two fields used together to **uniquely identify each record**. This prevents a couple placing the same item into their registry more than once because that would violate the primary key (see below).

CoupleID	ItemID	Quantity	Reserved	Paid
1	47	5	☑	☑
1	71	1	☐	☐
1	19	2	☐	☐
1	18	1	☐	☑
1	4	1	☐	☑
1	16	1	☐	☐
1	72	1	☐	☐
2	51	1	☑	☐
2	91	4	☑	☐
2	62	1	☐	☐
2	47	3	☐	☐
3	50	1	☐	☐
3	15	1	☐	☑
3	32	1	☐	☐
3	86	1	☐	☑
3	29	2	☐	☐
4	51	1	☐	☐
4	62	1	☐	☑
4	47	3	☐	☑
4	91	4	☐	☐
5	13	2	☐	☑
5	7	1	☐	☐
5	69	1	☐	☐
5	63	3	☐	☐
5	44	1	☐	☐
5	18	1	☐	☑
5	17	2	☐	☑
6	37	1	☐	☐
6	65	1	☐	☐
6	53	1	☐	☐
6	84	1	☐	☑

To harvest information from different tables **you must explain the join** in the WHERE clause. After the join you can still code the condition(s) that apply (filter out) by using the logical "AND" operator (see queries 13 below)

16
The join from tblCouples to tblRegistries looks like this . . .
 	
`FROM tblCouples, tblRegistries`
`WHERE tblCouples.CoupleID = tblRegistries.CoupleID`
	Primary key to Foreign key relationship

17
The join from tblItems to tblRegistries looks like this . . .
 	
`FROM tblItems, tblRegistries`
`WHERE tblItems.ItemID = tblRegistries.ItemID`
	Primary key to Foreign key relationship

18
The join from tblCouples to tblItems must go through the joining table and therefore looks like this . . .
 	
`FROM tblCouples, tblItems, tblRegistries`
`WHERE tblCouples.CoupleID = tblRegistries.CoupleID`
`AND tblItems.ItemID = tblRegistries.ItemID`
	Primary key to Foreign key relationship x2 joined using AND

Example: Write a query that gives the names of the couple getting married and the item, description and quantities of the items in their gift registry; only show those gifts that have been reserved and sort the list by surname. Here the data is spread over all three tables. You must explain the join from table to table.	**19**
	`SELECT surname, brideName, groomName, itemName, description, quantity, reserved`
	`FROM tblCouples, tblItems, tblRegistries`
	`WHERE`
	`tblCouples.CoupleID = tblRegistries.CoupleID AND`
	`tblItems.ItemID = tblRegistries.ItemID AND`
	`reserved = TRUE`
	`ORDER BY surname ASC;`

GROUP BY

GROUP BY allows the query to break a single result set down into multiple, individual result sets based on the field in the GROUP BY clause. The results sets are presented in alphabetical order automatically. The GROUP BY clause in SQL is used to arrange identical data into different result sets (groups). It collapses multiple rows that share the same values in specified columns into a single "summary" row.

Key Functions

Summarization: It is almost always used with aggregate functions (like COUNT(), SUM(), AVG(), MIN(), or MAX()) to perform calculations on each result set (group) individually.
Categorization: It allows you to segment your data by specific attributes, such as finding total sales **per region**, the number of employees **per department** or perform calculations in **each** group.
Hides Duplicates: When used without aggregate functions, it behaves similarly to SELECT DISTINCT by returning only the unique values in the specified columns.

Core Rules

Selection Constraint: Any column in your SELECT statement that is **not** part of an aggregate function must be included in the GROUP BY clause. It seldom makes sense to GROUP BY using more than one value.
Clause Order: In a standard SQL query, GROUP BY must come after the WHERE clause (which filters individual rows) and before the HAVING clause (which filters the groups themselves).
Handling NULLs: Most database systems treat all NULL values as a single group, combining them into one row in the output.

Markbook table

markbook

ID	LearnerName	test1	test2	test3	grade
1	Aphiwe	79	50	50	10
2	Mark	80	50	50	11
3	Gugu	82	50	50	12
4	Andy	43	86		10
5	Betty	65	87	60	10
6	Charlie	45	45	43	12
7	Dave	72	83	91	11
8	Eric	91	58	45	11
9	Francis	78	77	76	10

Consider this simple markbook table for the GROUP BY examples that follow.

Andy in grade 10 has a NULL value for test 3. This null will be ignored by the AVG aggregate function.

Some Basic Examples

20

```
SELECT grade,
    COUNT(*) AS [Number in Grade]
FROM markbook
GROUP BY grade;
```

Generally, only one field in the GROUP BY clause. Must be in the SELECT statement, if it is not an aggregate.
This query counts the number of learners in each grade.

21

```
SELECT *
FROM markbook
GROUP BY grade;
```

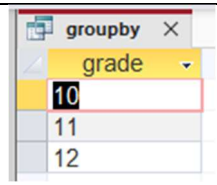
X

This does **not** work as you cannot use GROUP with the wildcard being in the SELECT clause.

22

```
SELECT grade
FROM markbook
GROUP BY grade;
```

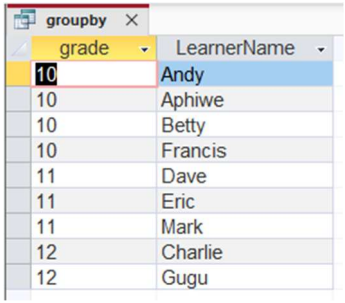
Works the same as
DISTINCT grade . . . ORDER BY.



23

```
SELECT grade,learnerName
FROM markbook
GROUP BY grade, learnerName;
```

Works the same as ORDER BY.



From example 22 and 23 above you can see the GROUP BY comes into its own when used with aggregate functions

Further GROUP BY examples using the markbook table

24

```
SELECT grade,
    ROUND(AVG(test1), 0) AS [Ave test1],
    ROUND(AVG(test2), 0) AS [Ave test2],
    ROUND(AVG(test3), 0) AS [Ave test3]
FROM markbook
GROUP BY grade;
```

Determines the average per grade and per test

groupby

grade	Ave test1	Ave test2	Ave test3
10	66	75	62
11	81	64	62
12	64	48	46

25

```
SELECT grade,
    COUNT(*) AS [Number in Grade]
FROM markbook
GROUP BY grade;
```

Counts the number of learners in each grade.

groupby

grade	Number in Grade
10	4
11	3
12	2

26

```
SELECT
    grade,
    MAX(test1) AS [Max test1],
    MAX(test2) AS [Max test2],
    MAX(test3) AS [Max test3]
FROM markbook
GROUP BY grade;
```

Determines the highest mark for each test for each grade.

In this example we do not know who got the highest mark. If you want to know who, then you must use TOP1 in conjunction with ORDER BY DESC

groupby 2

grade	Max test1	Max test2	Max test3
10	79	87	76
11	91	83	91
12	82	50	50

GROUP BY using the Wedding Registry database.

tblItems

ItemID	ItemName	Description	Category	Price	QuantityInS	SupplierSto	Click
1	Crystal Glassware	Set of 6 fine crystal wine glasses	Kitchenware	R850,00	25	2430	
2	Kitchen Mixer	High-performance stand mixer with multiple attachm	Appliances	R3 200,00	15	645	
3	Silverware Set	Complete silverware set for 8 people	Kitchenware	R1 600,00	30	3000	
4	Dinner Plates	Set of 12 elegant white porcelain dinner plates	Kitchenware	R1 100,00	50	248	
5	Cookware Set	Non-stick cookware set including frying pans and s	Kitchenware	R2 200,00	20	1520	
6	Blender	Powerful kitchen blender for smoothies and soups	Appliances	R900,00	40	2641	
7	Electric Kettle	Stainless steel electric kettle with rapid boiling	Appliances	R350,00	45	4500	
8	Toaster	4-slice toaster with adjustable settings	Appliances	R600,00	35	2697	
9	Coffee Machine	Espresso coffee machine with milk frother	Appliances	R1 800,00	12	2685	
10	Linen Set	4-piece luxury linen bedding set	Homeware	R950,00	60	822	
11	Towels Set	Set of 6 soft bath towels in various colors	Homeware	R550,00	70	3154	
12	Cutlery Set	24-piece stainless steel cutlery set	Kitchenware	R700,00	40	3813	
13	Throw Blanket	Cozy knitted throw blanket (pastel blue)	Homeware	R450,00	50	268	
14	Candle Set	Set of 6 scented candles with elegant holders	Decor	R300,00	90	845	
15	Wall Clock	Modern wall clock with minimalist design	Decor	R550,00	25	2500	
16	Picture Frame	Wooden photo frame with glass cover	Decor	R250,00	60	952	
17	Serving Platter	Large ceramic serving platter with floral design	Kitchenware	R600,00	15	1500	
18	Tea Set	Traditional tea set including teapot and cups	Kitchenware	R750,00	18	362	

27

```
SELECT category
FROM tblItems
GROUP BY category;

SELECT DISTINCT category
FROM tblItems
ORDER BY category;
```

Without an aggregate function GROUP BY works the same as **DISTINCT . . . ORDER BY**

T4

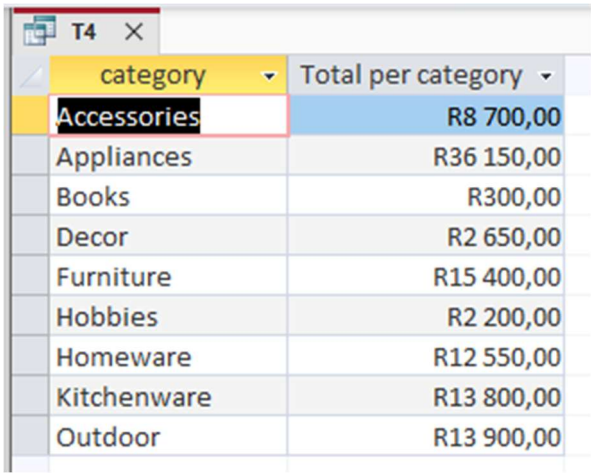
category
Accessories
Appliances
Books
Decor
Furniture
Hobbies
Homeware
Kitchenware
Outdoor

28

```
SELECT category, sum(price) AS [Total per category]
FROM tblItems
GROUP BY category;
```

In this case the single result set has been broken into nine different result sets, and the prices for each result set have been added up.

Reminder: Fields that are not aggregate functions in the SELECT statement need to be included in the GROUP BY. It seldom makes sense to have more than one field in the GROUP BY clause.



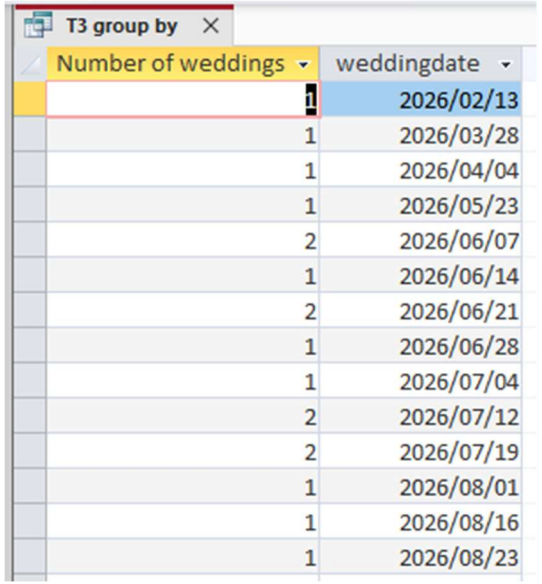
T4

category	Total per category
Accessories	R8 700,00
Appliances	R36 150,00
Books	R300,00
Decor	R2 650,00
Furniture	R15 400,00
Hobbies	R2 200,00
Homeware	R12 550,00
Kitchenware	R13 800,00
Outdoor	R13 900,00

29

```
SELECT count(weddingDate) AS [Number of weddings],
weddingdate
FROM tblCouples
GROUP BY weddingDate;
```

In this case the single result set of weddings has been broken up into individual dates. The number of weddings **per date** has been counted.



T3 group by

Number of weddings	weddingdate
1	2026/02/13
1	2026/03/28
1	2026/04/04
1	2026/05/23
2	2026/06/07
1	2026/06/14
2	2026/06/21
1	2026/06/28
1	2026/07/04
2	2026/07/12
2	2026/07/19
1	2026/08/01
1	2026/08/16
1	2026/08/23

GROUP BY . . . HAVING

HAVING is a conditional clause, same as WHERE. The condition must be true for the data to feature in the result set (often called "**filtering**"). WHERE is the condition that executes **first.** GROUP BY then breaks the single result set into smaller subsets based on its own field. HAVING is a condition (filter) that is then applied to the multiple result sets **after** the GROUP BY has executed.

30 (compare to above)

```
SELECT category, sum(price) AS [Total per category]
FROM tblItems
GROUP BY category
HAVING sum(price) > 3000;
```

Determines the total cost per category using only those categories that have a total amount larger than R3000-00

Copy Of T4

category	Total per category
Accessories	R8 700,00
Appliances	R36 150,00
Furniture	R15 400,00
Homeware	R12 550,00
Kitchenware	R13 800,00
Outdoor	R13 900,00

31 (compare to and above)

```
SELECT category, sum(price) AS [Total per category]
FROM tblItems
WHERE quantityInStock > 20
GROUP BY category
HAVING sum(price) > 3000;
```

The condition in the WHERE clause executes first.

Only items with more than 20 in stock will feature in the multiple results sets of the GROUP BY clause

Copy Of T4

category	Total per category
Appliances	R6 350,00
Homeware	R10 000,00
Kitchenware	R8 750,00

Embedded queries

In SQL, an embedded query refers to a **subquery**—a **nested** query inside another SQL statement.

A subquery is a SELECT statement written inside the WHERE, FROM, or HAVING clause of a main (outer) query to perform complex data retrieval or filtering. The inner query executes **first** and passes its result to the main query**.**

It allows one query to use the result of another query.

Often used to compare actual values in a table to derived values e.g. aggregate functions like SUM, MIN Max etc
Example: (e.g., WHERE salary > (SELECT AVG(salary) FROM employees)).

The following are examples of embedded queries – a query with two or more SELECT statements.

32 **SELECT** * FROM PrintLogs WHERE TotalPages > (**SELECT** AVG(TotalPages) FROM PrintLogs)	**33** **SELECT** make, model, price FROM tblInventory WHERE price < (**SELECT** AVG(price) FROM tblInventury.)
34 WHERE (**SELECT** AVG(field1) FROM Table1) > WHERE (**SELECT** AVG(field2) FROM Table1).	Each aggregate function must have its own SELECT statement so that it presents its own value for comparison

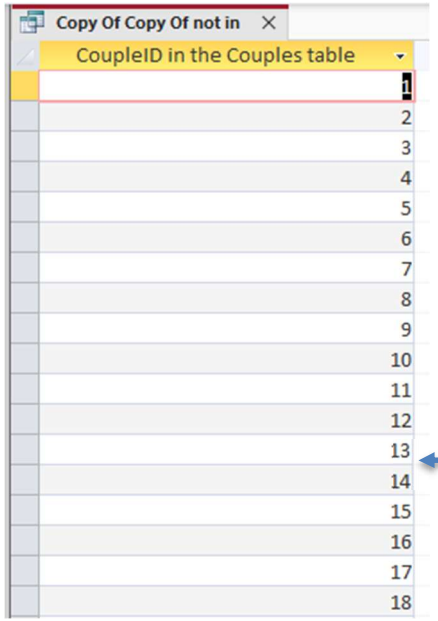
"NOT IN" using the Wedding Registry database. This is an embedded query with two SELECT clauses . . .

In the **many-to-many relationship** created by the tblRegistries, a couple can choose many items for their registry, and an item can be chosen by many different couples. It would be useful to know which couples have not yet selected any items from the Items table . . . they are "NOT IN", the many-to-many relationship. Here we have two results sets. We are looking for couples who are **not in** the second result set.

35

```
SELECT coupleID AS
[CoupleID in the Couples
table]
FROM tblCouples;
```

This is the list of ALL the couples



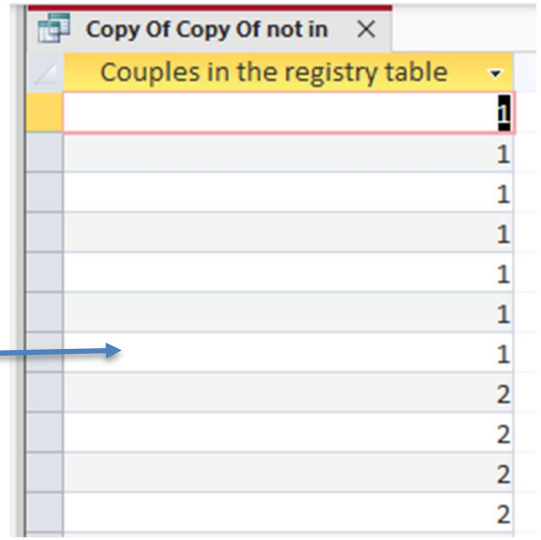
(Not all records are shown)

36

```
SELECT coupleID AS
[Couples in the registry
table]
FROM tblRegistries;
```

This is the list of ALL the couples with an entry in the registry table.

Which of these are not here



(Not all records are shown)

```
SELECT coupleID AS [CoupleID in
the Couples table]
FROM tblCouples;
```

WHERE . . . NOT IN

```
SELECT coupleID AS [Couples in the
Registry table]
FROM tblRegistries;
```

37

```
SELECT coupleID FROM tblCouples WHERE coupleID   NOT IN       (SELECT coupleID FROM tblRegistries);
```

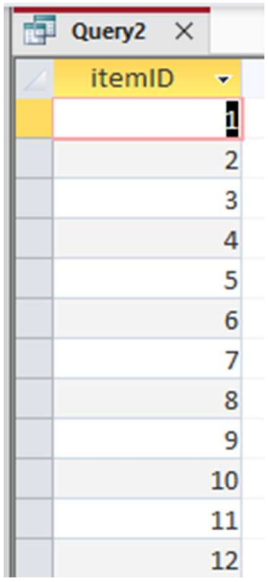
"NOT IN" using the Wedding Registry database. This is an embedded query with two SELECT clauses . . .

In the many-to-many relationship created by the tblRegistries, an item can be chosen many times, and a couple can choose many different items. It would be useful to know if an item has not been chosen by any of the couples . . . the item is "NOT IN", the many-to-many relationship. Here we have two results sets. We are looking for data in the first result set that is not in the second result set.

38

```
SELECT itemID
FROM tblItems
```

This is the list of ALL the gift items.



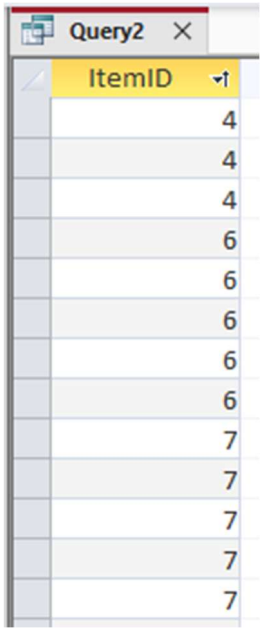
(Not all records are shown)

39

```
SELECT ItemID
FROM tblRegistries
```

This is the list of ALL the items that have been selected and placed into the registry table.

Which of these are not here



(Not all records are shown)

`SELECT itemID FROM tblItems`	WHERE . . . NOT IN	`SELECT ItemID FROM tblRegistries`

40

```
SELECT itemID FROM tblItems WHERE itemID NOT IN (SELECT ItemID FROM tblRegistries);
```

“NOT IN” with GROUP BY and DISTINCT. Wedding Registry Database - continued. Embedded queries.

It would be useful to know if a whole category of gift items is not being selected by the wedding couples.

Each query must have two result sets. We are looking for data in the first result set that is **not in** the second result set.

Below are two versions of the query, one with redundancy and the other without. THE SQL clause DISTINCT gets rid of duplicates.

<table>
<tr><th colspan="2">The query with redundancy.</th><th colspan="2">Using DISTINCT. The same query, improved version, with no redundancy.</th></tr>
<tr>
<td>

41

```
SELECT category
FROM tblItems
WHERE category
NOT IN
(SELECT category
FROM tblRegistries,
tblItems
WHERE
tblRegistries.ItemID =
tblItems.ItemID
GROUP BY category);
```

This will list the categories not selected, but the duplicates are redundant.

</td>
<td>

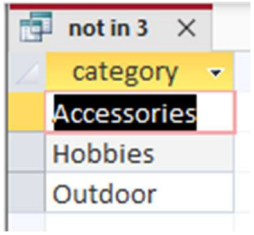
</td>
<td>

42

```
SELECT DISTINCT category
FROM tblItems
WHERE category
NOT IN
(SELECT category
FROM tblRegistries,
tblItems
WHERE
tblRegistries.ItemID =
tblItems.ItemID
GROUP BY category);
```

The addition of **DISTINCT** gets rid of duplicates and avoids redundancy.

</td>
<td>



</td>
</tr>
</table>

DISTINCT – Further examples

In Ms Access you cannot combine an aggregate functions with DISTINCT in a single statement.	
43 `SELECT COUNT(DISTINCT country) FROM customers;`	Counts all the countries without counting the duplicates. X
44 `SELECT SUM(DISTINCT Price) FROM tblProducts;`	Totals only those prices that are unique. X
To combine an aggregate function with DISTINCT in Ms Access you must use an embedded query. Aggregate function on the outside and DISTINCT on the inside.	

45
`SELECT COUNT(country)AS [Unique Countries]`
`FROM (SELECT DISTINCT country FROM tblCountries)`
	Counts all the countries without counting the duplicates. ✓

46
`SELECT SUM(Price) AS [Unique prices total]`
`FROM (SELECT DISTINCT price FROM tblProducts`
	Totals only those prices that are unique. ✓

Embedded queries: Continued.

Example: When we need to **compare** the data in one field to the **result** of an aggregate function from another field (that could also be in another table.

Aggregate functions need their own SELECT statement and can only yield one result. Therefore, the embedded query must be made up of **two parts** – two SELECT statements.

Embedded SQL statement	**Commentary**

47
 	
`SELECT Name`
`FROM tblMarks`
`WHERE Science >`
`(SELECT AVG (Science) FROM tblMarks );`
	Finds those pupils whose science marks exceed the average mark of the class.
	Note that the second part has a SELECT and a FROM (at the very least)

48
 	
`SELECT TrailID, Distance`
`FROM tblTrails`
`WHERE Distance <  (SELECT Avg(Distance)`
`FROM tblTrails);`
	Finds trails whose distance is less than the average distance of all the trails.

49
 	
`SELECT HikerID, Round(((now( ) -`
`DateOfBirth) / 365.25),1) AS Age`
`FROM tblHikers`
`WHERE Now( ) - DateOfBirth >`
`(SELECT Avg((Now( ) - DateOfBirth)) FROM`
`tblHikers);`
	Finds hikers whose exact age is more than the average age of the other hikers

Aggregate functions return the value only. We can use an embedded query to get **additional information** to go with the value. See below.

50

```
SELECT tblClients.clientName,

(SELECT SUM(Quantity)
FROM tblInvoices
WHERE tblInvoices.ClientID =
tblClients.ClientID) AS
TotalQtyPurchased)

FROM tblClients;
```

The **inner query** finds the SUM of the items bought (WHERE the client's ID is found on a **matching** invoice). BUT we don't know the name because aggregate functions don't give us any other detail – they only return one value.

Therefore, we must use the **outer query** to get the name.

Structure of DQL – The clauses	**Embedded queries can be place in the SELECT, FROM, WHERE or HAVING clauses.**
SELECT <field(s)> **FROM <table(s)>** **WHERE condition(s)** **GROUP BY expression** **HAVING condition – works with GROUP BY** **ORDER BY expression**	**SELECT**: Returns a single value for each row in the main result, effectively creating a new calculated column. **FROM:** Acts as a "derived table" or virtual table. The main query then selects data from this temporary result set. **WHERE:** Used **most frequently** to filter records based on dynamic criteria. It often appears with operators like IN, EXISTS, or comparison symbols such as =, >, or <. **HAVING**: Used to filter grouped or aggregated results based on the output of another query.

Queries that alter data in a table (insert records, delete records or edit existing records) (table structure is not altered)

Structure of DML	
INSERT . . . INTO . . . . VALUES	INSERT: Defines a new record to be added to a target table. Here the values are given in the query hence the keyword “VALUE”
INSERT . . . INTO . . . . SELECT FROM WHERE	INSERT: Defines a new record to be added to a target table where the new record based on an existing record. Here VALUE falls away and is replaced by an embedded query (also called a nested query)
UPDATE . . . . SET	UPDATE: Used within the SET clause to assign new values to columns or in the WHERE clause to filter which rows to modify.
DELETE . . . . FROM	DELETE: Found in the WHERE clause to identify specific records that should be removed.

A background comment. Real data from the table can be replaced by other data in the SQL query.

51

```
SELECT CoupleID,"A new name",BrideName,
"the wrong number",WeddingDate
FROM tblCouples;
```

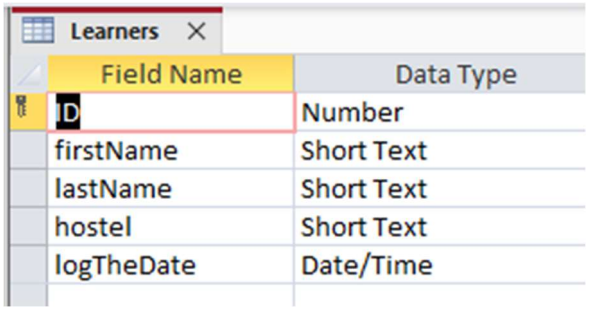
a new name

CoupleID	Expr1001	BrideName	Expr1003	WeddingDate
1	A new name	Emily	the wrong number	2026/06/14
2	A new name	Thandi	the wrong number	2026/07/12
3	A new name	Sophia	the wrong number	2026/08/01
4	A new name	Amina	the wrong number	2026/09/20
5	A new name	Olivia	the wrong number	2026/10/03
6	A new name	Priya	the wrong number	2026/06/21
7	A new name	Boitumelo	the wrong number	2026/09/24

INSERT with VALUES	
52 **INSERT INTO** tblCouples (surname, bridename, groomname, contactnumber, weddingdate) **VALUES**("alpha","beta","charlie","0821234567",#10/10/2010#)	A straightforward INSERT statement. The fields match the data exactly. Note the hash characters (#) to indicate date-data to follow. No SELECT clause therefore VALUES is used.
INSERT with SELECT	
53 **INSERT INTO** tblItems (ItemName, Description, Category, Price, QuantityInStock, SupplierStock) **SELECT** itemName, Description, Category, 1500, *** 50, 100 FROM tblItems WHERE itemID = 19;	**An INSERT with SELECT statement that uses an existing record – therefore the WHERE clause is compulsory and "VALUES" falls away.** Note that only the first statement is surrounded by round brackets. The new record has the same ItemName, Description and Category but the price, QuantityInStock and SupplierStock are different. This is an embedded query because a SELECT statement is needed to get the data from itemID number 19. The second statement runs first (the SELECT). Once the values are prepared, the new record is inserted into the table where itemID (not shown here) is the primary key with a datatype of **AutoNumber.** *** Data from record 19 is replaced by other data in the SELECT statement – see query 51 above.

INSERT with SELECT (cont)

Consider the following database table where we must calculate the next unique primary key value.
Also the use of NOW ( )



Here the primary key (no duplicates allowed) is a number, but the data type is not Autonumber. How will a new record be added to this table?

Also the use of NOW ( ) which logs the date and the time of when the data was captured.

54

```
INSERT INTO
    Learners (ID, firstName, lastName, hostel,
logTheDate)
SELECT
    (
        SELECT
            MAX(id) + 1
        FROM
            Learners
    ),
    "Dave",
    "Delta",
    "Eagle",
    Now ()
FROM
    Learners
WHERE
    firstname = "Andy";
```

Every SELECT is paired with its own FROM.

Aggregate functions must have their own SELECT.

The number of field names must match the number of data items.

When INSERT is used with SELECT to insert new records that match "existing" records the WHERE clause is compulsory **even if every value of the existing record is not used.**

The second SELECT does not have its own round brackets.

The keyword "VALUES" is not used as it is superseded by the SELECT statement.

Now ( ) logs the date and the time that the record was added using the INSERT query.

Revisiting INSERT, UPDATE and DELETE.

51 INSERT – adds a new record to a table and populates all the fields (when autonumber is not the primary key)

INSERT INTO tablename VALUES (field1Data, field2Data, field2Data) – no field names.

NOTE: The VALUES, the order of the values, and the datatypes match the table structure exactly

52 INSERT - adds a new record, specified fields . . . (when autonumber is the primary key)

INSERT INTO tablename (fieldTitle1, fieldTitle2, fieldTitle3) VALUES (field1Data, field2Data, field2Data)

E.g. INSERT INTO tblname (name, DOB, gender, grade, boarder) VALUES ('Lynn', #02 Feb 2000#, 'F', 10, false)

53 UPDATE – all . . . (the whole table, and all its records are given a new value e.g. the school gets a new name – everybody is affected.

UPDATE tablename SET field1 = value1, field2 = value2, fieldN = valueN

54 UPDATE – updates a record that matches a condition.

UPDATE tablename SET field1 = value1, field2 = value2, fieldN = valueN WHERE condition.

E.g. UPDATE PrintLogs SET FirstName = "Henrietta" WHERE Surname = "Bates" AND FirstName = "Henry".

55 DELETE – all . . . Deletes all the records in the table and cannot be undone in Ms Access. The table structure is not affected.

DELETE * FROM tablename

56 DELETE – those that match a condition . . . (NOTE: This delete SQL command cannot be undone Ms Access)

DELETE FROM tablename WHERE fieldname = value

E.g. DELETE FROM tblStudents WHERE studentID = 38 (Use the primary key value, not the person's name)

57 Arithmetic function: INT( ), ROUND( ). Formats the single parameter within the brackets. ROUND rounds up or down. INT does not round.

ROUND(((Now()-invoicedate)/365.25),3) . . . this rounds to 3 decimal places. Take care with the round brackets.

58 Random numbers with predictable starting and ending value ie 1 to 10, 1 to 100, 1 to 1000 etc

RND(seed goes here) – Generating a random number requires a few steps.

A) Using a random number seed within the brackets generates a unique number between zero and 1 for each and every record.

B) Then you need to multiple it by 10, 100 or 1000 to get a real value bigger than one.

C) Then you need to add one to avoid generating a zero.

D) Then you need INT to truncate the real to an integer

Example: **0.87678902** becomes **87.678902 becomes 88.678902** becomes **88**

E.g. SELECT INT (RND(riderID) * 100 + 1) AS [Random number] FROM tblRiders. Here the "rider id" is used as the random number seed.

59 Random numbers: From one arbitrary value to another arbitrary value e.g. 32 to 78

INT((Upper_Bound - Lower_Bound + 1) * RND(ID) + Lower_Bound)

To generate numbers between interval e.g. between 32 and 78 – see below

SELECT INT((78 - 32 + 1) * RND([riderID]) + 32) AS [Random number] FROM tblRiders;