Database Designs

Section 4: Subqueries, Constraints & Advanced Concepts

4.1 Subqueries Introductor

So far we have explored how to query from tables and virtual table to generate a results table. SQL Subqueries is a fairly advanced concept which allows us to query from a SQL query i.e. a query within a query.

So far we have seen queries as standalone commands that fetch data from a database; however, in reality, queries are generally plug-and-play - what do we mean by this? Plug-and-play means the ability to use queries in places where you would not expect them to be used, this is because the results of queries are tables.
Tables can be real tables, tables that are generated by joins or tables that are a result of queries. Therefore, queries are plug-and-play into other pieces of SQL.

A query is a command that returns a table (columns and rows). If you imagine the result of a query as a table by itself, then this mentality will open all the possibilities that you can do with the results of a query. For example:

- We could calculate the Union, Intersection or Differences of two queries.
- We could use one query inside another (via Subqueries).
- We could use a subquery to populate a table via an **INSERT**.

To conclude, a subquery is nothing more than a query of which the results are used in another query, otherwise there is no logical differences between a subquery and a normal standalone query.

4.2 Union, Union All, Intersect and Except

We can calculate the Union, Intersection and Difference between two queries provided they have the same columns i.e. the number, order and type of the columns are identical across the two queries.

To demonstrate the **UNION**, **UNION ALL**, **INTERSECT** and **EXCEPT** set operators subqueries, below are two tables QUERY_ONE and QUERY_TWO.
The QUERY_ONE table row data is coloured turquoise while the QUERY_TWO table row data is coloured peach. The common data rows of both tables are coloured in yellow.

This will help clearly illustrate how SQL creates the subquery tables using each Set operators and how we can use subqueries to perform more advanced queries and open our minds to all the possibilities SQL offers using the plug-and-play mentality.

QUERY_ONE			
String	Number	Date	Boolean

QUERY_TWO			
String	Number	Date	Boolean

The **UNION** set operator creates a new query table which combines the two queries together but creates one copy of the common row i.e. removes duplicate rows.

UNION_QUERY			
String	Number	Date	Boolean

The **UNION ALL** set operator creates a new query table which combines the two queries together containing all duplicate rows.

UNION_ALL_QUERY			
String	Number	Date	Boolean

The **INTERSECT** set operator finds the commonality between the two sets that it is intersecting i.e it returns the common row between the two query tables.

INTERSECT_QUERY			
String	Number	Date	Boolean

Finally, the **EXCEPT** set operator creates a new query table which removes everything in the first table (i.e. the left table) that is present in the second table (i.e. the right table). This returns the difference between the two sets of input queries.

EXCEPT_QUERY			
String	Number	Date	Boolean

EXCEPT_QUERY			
String	Number	Date	Boolean

Important Note: The Except query can return either of the above results depending on which table was made as the left table in the subquery. The left table is always the first table mentioned in the query within the **FROM** clause.

The **UNION**, **UNION ALL**, **INTERSECT** and **EXCEPT** are all a kind of Set operations. We know that a query is a command that returns a table and not really a Set. The Entity Relationship theory tells us that a table is a collection of tuples (a tuple relates to one row within the table). A Set cannot contain duplicates but a table can.

By default the **UNION** will eliminate duplicates but SQL has the Union All operator to keep all the duplicate tuples if we seek that operation behaviour. There is another rule which applies to the **UNION** Set operator which is where individual queries that participate in a **UNION** operator cannot have the **ORDER BY** clause. This is because the elements of a Set are not ordered. Below is an example demonstrating this:

Pet_Owners	
AptNumber	Name
123	John
345	Tim
349	Vandana
567	Bilal

Flat_Owners	
FlatNumber	Name
234	Mary
567	Bilal
879	Jane
903	Ellen

In this example we have two tables of Pet_Owners and Flat_Owners both have the same columns i.e. they have in common the same number, order and type of columns. The names of the columns is irrelevant. In the above example, both tables have two columns of integer and string in that specific order. Below is the syntax to perform the **UNION** Set operator on the two tables.

```
(SELECT AptNumber AS FlatNumber, Name FROM Pet_Owners)
UNION
(SELECT FlatNumber, Name FROM Flat_Owners);
```

It is OK for the column names in the physical tables to be different. This is because in a query we can easily alias one column name to another - notice the AptNumber **AS** FlatNumber which simply glosses/hides the fact that the original column name was

something different. This is the reason for why the actual name from the physical table is irrelevant. The final subquery table will return a results table with two columns called FlatNumber and Name as seen below:

UNION_RESULTS_TABLE	
FlatNumber	Name
123	John
345	Tim
349	Vandana
234	Mary
567	Bilal
879	Jane
903	Ellen

As previously mentioned, individual queries that have been ordered by the **ORDER BY** clause cannot be used with the UNION set operator. Therefore, the below example syntax is invalid and will not work returning an error:

```
(SELECT AptNumber AS FlatNumber, Name FROM Pet_Owners ORDER BY Name)
UNION
(SELECT FlatNumber, Name FROM Flat_Owners ORDER BY Name);
```

However, we can write the syntax so that the results of the **UNION** is ordered by the column Name. In this case the **ORDER BY** clause will work on ordering the **UNION** results table:

```
(
  (SELECT AptNumber AS FlatNumber, Name FROM Pet_Owners)
  UNION
  (SELECT FlatNumber, Name FROM Flat_Owners)
) ORDER BY Name;
```

Pay particular attention to the curly brackets which wraps around the whole **UNION** set operator subquery. The **ORDER BY** clause is applied to the results of the **UNION** because it is falls outside the wrapping curly brackets. Therefore, it is only the individually queries participating in the **UNION** that cannot have the **ORDER BY** clause and the reason for why the second syntax works without throwing any errors.

4.3 Query-In-A-Query

Subqueries are very useful because they allow us to write SQL queries entirely free of hardcoded values or very large intermediary tables. Subqueries at first can seem very difficult to get use to at the beginning but once you know how to use them they are

extremely powerful. We will now explore how to use a query within a query (also known as subqueries) using the tables below.

Stores_Data		
storeID	storeLocation	city
1	ASDA Wilmslow	Manchester
2	ASDA Stratford	London

Products_Data	
productID	productName
1	Bread
2	Milk
3	Noodles
4	Nutella

Sales_Data			
storeID	productID	salesDate	totalRevenue
1	1	November 20, 2020	7,233.32
1	3	November 20, 2020	3,234.84
1	2	November 20, 2020	5,865.55
1	2	November 20, 2020	6,849.99
2	3	November 20, 2020	2,110.95
2	2	November 20, 2020	4,558.24
2	4	November 20, 2020	2,284.75

If a database existed like the above it is most likely that the business would like to pull a report to see what are the annual revenue is like for various products and make business decisions around this information. The query to extract this report would look like the following:

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM Sales_Data AS s
INNER JOIN Products_Data AS p
            ON s.productID = r.productID
WHERE (p.productName = 'Bread' or p.productName = 'Milk')
           AND (YEAR(date) = 2020)
GROUP BY p.productName, YEAR(date);
```

In the above syntax we are matching on the productID column and combining the Products_Data table onto the Sales_Data table using an **INNER JOIN**. The **WHERE** clause shows that we are interested in products 'Bread' and 'Milk' which are believed to be the top sellers and we are interested in the current year. Finally, we wan to **GROUP BY** the productName and the year. In conclusion this query will return back using the **SELECT** statement the productName, year and the **SUM** of the totalRevenue for each product within the current year groups.

The above query will only return information for the year 2020 and about 'Bread' and 'Milk' because that is the date and products specified in the **WHERE** clause. This query would get the job done to extract the report from the database for the business. However, there are many issues with the above query which relate to hardcoding the values. Our objective is to re-write the code to avoid any hardcoded values.

First, the query is hardcoded for the year 2020 which means the query would need to be updated every year. To avoid this hardcoding of values we can plug one query into another using subqueries as seen below.

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM Sales_Data AS s
INNER JOIN Products_Data AS p
        ON s.productID = r.productID
WHERE (p.productName = 'Bread' or p.productName = 'Milk')
        AND (YEAR(date) = (SELECT YEAR(MAX(date)) FROM Sales_Data))
GROUP BY p.productName, YEAR(date);
```

Here we are using one query inside another larger query. The inner query (highlighted in yellow) returns the maximum value of the year from the Sales_Data table. This query will look for the last transaction from the Sales_Data table because this would be the maximum (latest) date and return the year value from the date. Therefore, the inner query returns a single value i.e. the year in which the last transaction occurred.
In the **WHERE** clause of the outer query it simply compares the year of the date that is returned from the inner query. The **WHERE** clause functions as it did before.

Note that the the inner queries are always evaluated first.

The next issue relates to the hardcoded products value within the **WHERE** clause. What if we do not want to hardcode these values. Let's imagine that we instead have a TopSellers_Data table which has the productID and productName columns. This table holds the top products we are interested in at any point in time.

TopSellers_Data	
productID	productName
1	Bread
2	Milk

Instead of hardcoding the products in our query we instead want to retrieve the products from the TopSellers_Data table. Again we can do this using a subquery as seen below:

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM Sales_Data AS s
INNER JOIN Products_Data AS p
        ON s.productID = r.productID
WHERE (p.productName IN (SELECT p.productName FROM TopSellers_Data)
        AND (YEAR(date) = (SELECT YEAR(MAX(date)) FROM Sales_Data))
GROUP BY p.productName, YEAR(date);
```

Once again we have plugged one query into another (the inner query is highlighted in yellow). The inner **SELECT** statement simply returns all the products from the TopSellers_Data table. This TopSellers_Data table can be modified/updated and the query will always return the latest top products. The inner query returns a range of values which the outer query can then use these values using the **IN** keyword for its **WHERE** clause check for the current top products.

The final issue with the original query is to do with the size of our table. Let us imagine as time passes the business is operating extremely well. This means more sales will be recorded in the Sales_Data table and the possibility that the Sales_Data table eventually becomes enormous. We now have an issue that the table is too large to perform the **INNER JOIN** on the Sales_Data table which could take too long to generate the report. What we wan to do is reduce the number of rows that we select from the Sales_Data table. We know that at any point in time we are only interested in a few products which are present in the TopSellers_Data table. So How can we reduce the number of rows in the Sales_Data table that should be part of the query? Once again we would use subqueries as seen below:

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM (SELECT * FROM Sales_Data WHERE productID IN (SELECT productID
        FROM TopSellers_Data)) AS s
INNER JOIN Products_Data AS p
        ON s.productID = r.productID
WHERE (p.productName IN (SELECT p.productName FROM TopSellers_Data)
        AND (YEAR(date) = (SELECT YEAR(MAX(date)) FROM Sales_Data))
GROUP BY p.productName, YEAR(date);
```

Instead of only using the Sales_Data table in the **INNER JOIN**, we can use a **SELECT** statement which returns a table to be one of the tables in the **INNER JOIN**. Remember the result from a **SELECT** statement is also a virtual table which can therefore be used as part of any of the joins i.e. wherever we use a table, we can use a query. Once again we have used one query plugged into another query to improve the performance of our query. We are now using a subset of the Sales_Data table which helps improve the performance of our query. The outer query uses the subset table (which is also a table) in the **FROM** clause to do the **INNER JOIN**.

We now have a final subquery that looks like the below:

```
SELECT p.productName, YEAR(date), SUM(totalRevenue)
FROM (SELECT * FROM Sales_Data WHERE productID IN (SELECT productID
        FROM TopSellers_Data)) AS s
INNER JOIN Products_Data AS p
            ON s.productID = r.productID
WHERE (p.productName IN (SELECT p.productName FROM TopSellers_Data)
           AND (YEAR(date) = (SELECT YEAR(MAX(date)) FROM Sales_Data))
GROUP BY p.productName, YEAR(date);
```

We can see that this query is entirely free from any hardcoded values or very large intermediate tables all thanks to the use of subqueries rather than the tables itself.

To conclude, subqueries allows us to use non-hardcoded values by returning values from subqueries as well as optimising the performance of our queries by using subset tables created by subqueries. As we can see subqueries can be difficult to grasp at the beginning but once we understand how to harness them they become very powerful and useful tools when querying databases.

4.4 Inserting via Subqueries

We have been exploring the different ways of how we can use subqueries as plug-and-play. We know that queries are perfect substitutes for tables in SQL. In this section we will explore how we can populate a table using subqueries.

We could use a subquery to populate a table via a **INSERT**. The **INSERT** statement is the mechanism we use to get data into a table. We have already explored the **CREATE TABLE** and **INSERT** statements in Section 2.8 and 2.10. The **CREATE TABLE** is used to create a table in a database while the **INSERT** statement is used to insert new tuples (row data) into a table.

Using the **INSERT** statement requires us to use the statement many times to insert many data into a table (we also explored bulk loading data to speed up inserted data into a table using an external data source).

It is also possible that the data we want to insert into a table already exists in another table in some other form. This is where subqueries are useful as a plug-and-play solution. Subqueries are far more convenient to populate a table from existing tables in the database.

There are two methods we can do this:

1. Create the table as usual and then insert the data using a subquery, or
2. Create the table and populate it directly using a subquery in one go

Both methods are demonstrated below starting with the first method then followed by the second method:

Method 1:

```
CREATE TABLE EmailAddresses (
        Email VARCHAR(30) NOT NULL,
        Category VATCHAR(10) NOT NULL
);

INSERT INTO EmailAddresses
(SELECT DISTINCT Email, 'Student' AS Category FROM Students)
UNION
(SELECT DISTINCT Email, 'Faculty' AS Category FROM Faculty);
```

In this first example, we create a new table called EmailAddresses (using the normal process) which has two columns of Email and Category. This is then followed by a second command that **INSERT INTO** the EmailAddresses table. However, this is not the normal **INSERT INTO** command we are use to, instead what follows the statement is a query which is unusual.
The query that follows the **INSERT INTO** statement is a **UNION** of two queries . Note that this does not necessarily need to always be a **UNION** but rather it needs to be any query where the result will be inserted into the table. There is one requirement. The query needs to match the table in the number of and type of columns.

In the above the query has to select two columns because the EmailAddresses table has two columns of Email and Category and the type/order of the columns returned from the query must both be the same type/order as specified in the EmailAddresses columns i.e. VARCHAR for both.

Method 2:

```
CREATE TABLE EmailAddresses (
        Email VARCHAR(30) NOT NULL,
        Category VATCHAR(10) NOT NULL
)
AS
(SELECT DISTINCT Email, 'Student' AS Category FROM Students)
UNION
(SELECT DISTINCT Email, 'Faculty' AS Category FROM Faculty);
```

In the second example, the first part before the **AS** keyword is the **CREATE TABLE**'s table definition. The second part after the **AS** keyword is the query. Both of these parts are combined together to form one SQL Statement using **AS** keyword.
The query is exactly the same as the first method example and the same rules applies for the second method i.e. the same number of columns and column types have to match the table specified in the **CREATE TABLE** portion of the statement.
The **AS** keyword is the linking portion of the statement which links the **CREATE TABLE** with the query.
There is nothing special with the second method other than combing two separate statements into one statement.

We can refactor the above statement so that it is even shorter by eliminating some part of statement which we considered as not important:

```
CREATE TABLE EmailAddresses
AS
(SELECT DISTINCT Email, 'Student' AS Category FROM Students)
UNION
(SELECT DISTINCT Email, 'Faculty' AS Category FROM Faculty);
```

The above has removed the table column definitions entirely because we can figure out what columns the EmailAddresses table should have on the basis of **SELECT** query. However, skipping this step is not recommended because any constraints and keys we want to specify in the EmailAddresses table will be missing.

Note that we can also use subqueries in **UPDATE** and **DELETE** statements in a similar fashion to the above.

This concludes the three different ways we can use subqueries in a plug-and-play manner.

4.5 NOT NULL Constraints

Constraints are conditions that data in a database must satisfy. The database administrators/architects specifies the constraints/conditions when creating a database table and the DBMS enforces the constraints. This functionality is a huge part of the appeal of DBMS over say a File System for data storage.

Transactions are logical units of work and the database guarantees that each transaction satisfies the four properties known as the ACID properties. ACID is a famous acronym for Atomicity, Consistency, Isolation and Durability. The underpinning of the ACID properties is the Consistency i.e. it is the most important property because it encapsulates a lot that occurs with the database. The Consistency in a database is explicitly measured in terms of constraints. As long as the constraints in a database are all satisfied the database is considered consistent. Constraints collectively are sometime called integrity constraints because the explicit purpose of the constraint is to make sure the integrity of the data in the database are maintained and not compromised.

There are four common types of constraints that we should understand:

- NOT NULL Constraints
- Primary Key Constraints
- Foreign Key Constraints
- Check Constraints

Important Note: Constraints are usually added when the table is created but can also be added after a table has been created using the **ALTER TABLE** command. If we try to add a constraint to a table and the pre-existing data in the table does not satisfy the constraint, the DBMS will fail to alter the table and the SQL interpreter will provide an error message telling us that the data is not in a state of integrity to begin with. We can go ahead and fix up the data and then re-run **ALTER TABLE** command.

NOT NULL constraints (and default values constraints) help define whether NULL values are allowed/permissible in a particular column of a table.

Remember that NULL implies that a value does not exist. This is a special value and is not the same as a blank string or zero value which are values that do exist. NULL is also neither TRUE or FALSE (boolean values) and is simply NULL (absence of existence)

Any column can contain a value of NULL provided the table has been defined in a way that allows NULL values in that column.

Default Value are a closely related concept to NOT NULL constraints. They are a way of specifying what value a particular column should take when data value for that column is not present. We use the **DEFAULT** keyword followed by a value to specify the default value for that column when setting up the table. Therefore, we can specify a table with a NOT NULL constraint but also provide a default value. If an insert statement skips the value of the column, the default value will be assigned instead of a NULL. This will ensure the NOT NULL constraint is being fulfilled while at the same time not restricting users in situations where a value is not provided.

4.6 Primary Key Constraints

There are two types of Key Constraints: Primary Key and Foreign Key.

A key is a set of columns whose values are unique for each row in a table. A Primary key is a special type of key/candidate key. A key/candidate key is a set of columns whose values are unique in each row of a table. A table can have any number of keys/candidate keys of which only one of the key is marked as being special.

There should always be a Primary Key in every table created and if it is missing there should be a very valid reason - typically because there is a Foreign Key constraint. If a table has neither a Primary Key nor a references to another Primary Key in another table then this might be a sign that the table design is flawed.

The DBMS will often construct an index on the primary key even without being explicitly told to do so. The database is predicting ahead of time that if we mark a column or a set of columns as a primary key then we are going to be doing a lot of queries/lookup on the primary key and therefore it makes sense for the database to be ahead of our requests and make it easy to perform these queries. Smart database optimisations such as these which are based on heuristic signals that database designers and optimiser have picked up over the years of preparing database engines are an important reason for why database systems are getting so much better steadily over time.

This distinction between a Primary Key and a general candidate key is something that is often asked in interviews so it is important to remember this constraint.

A Primary Key could include either a single column or multiple columns. A multiple columns key is also known as a Composite Primary Key and the DBMS will check that the combination of values from the selected columns are unique.

Primary Key columns can never contain NULL values. This is a common sense rule to keep in mind because it does not make sense for a Primary Key column to not exist and NULL means a value does not exist.

Primary Keys are a type of constraint because it specifies a condition that the data in the table must satisfy. This is an Integrity constraint and by far the most important type of Integrity constraint. The DBMS will throw an error if this constraint is violated for example inserting two rows with the same values for the Primary Key column(s). This is how DBMS can maintain the ACID properties i.e. Consistency and Integrity of the data.

We specify the Primary Key when creating a new table. This can be specified at the very end of the **CREATE TABLE** command (i.e. after the table columns have been defined) or inline if it is a single column Primary Key (i.e. add **PRIMARY KEY** after the column which is the key attribute) as seen below:

```
CREATE TABLE Students (
        studentID INT NOT NULL AUTO_INCREMENT,
        firstName VARCHAR(30) NOT NULL,
        PRIMARY KEY(StudentId)
);

CREATE TABLE Students (
        studentID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        firstName VARCHAR(30) NOT NULL
);
```

We cannot use the inline method to define multi-column Composite Primary Keys which requires the former explicit method out-of-line definition. We can add Primary Key constraints after a table has been created and pre-existing data exists in the table. We would do this using the **ALTER TABLE ADD INDEX** command. This command will go through all the pre-existing data in the table to ensure that it does not violate this Primary Key constraint to begin with before applying the alteration to the table.

To conclude on Primary Key Constraints, any columns or set of columns could be declared using the **UNIQUE** keyword and the syntax is exactly like that of a **PRIMARY KEY**. A table can have only one **PRIMARY KEY** but any number of **UNIQUE** constraints; however, the **PRIMARY KEY** constraint implies uniqueness but the reverse is not true for **UNIQUE** constraints.

```
CREATE TABLE Students (
        studentID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        firstName VARCHAR(30) NOT NULL,
        emailAddress VARCHAR(30) NOT NULL UNIQUE
);
```

4.7 Foreign Key Constraints

The Foreign Key Constraint is used to define a column with a table that is also a key in another table. Such constraints are called Foreign Key Constraints or Referential Integrity Constraints. It is called a Referential Integrity Constraint because one table refers to another table to check its integrity is maintained and cannot be purely determined internally within the table alone i.e. it is only possible to determine the integrity via the reference to the external table.

```
CREATE TABLE Campus_Housing (
        studentID INT NOT NULL,
        dormitoryName VARCHAR(50),
        apartmentNumber INT,
        CONSTRAINT fk_student_studentid,
        FOREIGN KEY(studentID),
        REFERENCES Students(studentID)
);
```

In the example syntax above, the **CONSTRAINT** keyword is used to setup the referential integrity of which we can provide a name for this Foreign Key Constraint e.g. fk_student_studentid. The **FOREIGN KEY**(studentID) refers to the studentID columns from the internal Campus_Housing table while the **REFERENCES** Students(studentID) refers to the studentID from the external Students table. Again we can add a foreign key constraint at a later time after the table has been created and has pre-existing data but the table must satisfy the constraint before it is applied to the table.

We should always make sure to add our constraints during the creation of the table and that our tables have either a Primary Key or Foreign Key constraint. Foreign Keys are usually added when one table relies on another table to make sense of the data (but not vice versa). In the example above the Campus_Housing table relies on the Students table to make sense of the data held in that table while the Students table does not rely on the Campus_Housing table and can make sense on its own.

4.8 DELETE and UPDATE with Foreign Key Constraints

Delete and Update commands can make Foreign Key Constraints quite complicated. For demonstration purposes let us imagine that we have two tables called Current_Employees and Upcoming_Promotions as demonstrated below:

Current_Employees		
EmployeeID	FirstName	LastName

Upcoming_Promotions		
EmployeeID	CurrentTitle	NewTitle

Clearly only current employees can be promoted i.e. a Foreign Key Constraint is called for the Upcoming_Promotions table. Below is the syntax for creating the above two tables:

```
CREATE TABLE Current_Employees (
        EmployeeID INT NOT NULL PRIMARY KEY,
        FirstName VARCHAR(50) NOT NULL,
        LastName VARCHAR(50) NOT NULL
);

CREATE TABLE Upcoming_Promotions (
        EmployeeID INT NOT NULL,
        CurrentTitle VARCHAR(50),
        NewTitle VARCHAR(50),
        CONSTRAINT fk_upcomingpromo_current,
        FOREIGN KEY(EmployeeID),
        REFERENCES Current_Employees(EmployeeID)
);
```

The Upcoming_Promotions table references the EmployeeID Primary Key column from the Current_Employees as the referential integrity constraint for its Foreign Key column of EmployeeID (as shown by the highlighted code).

What would happen to the Upcoming_Promotions table if an employee leaves a company without waiting for their promotion? It is clear that we would have to delete the employee from the Current_Employees table because they are no longer a current employee. However, this now leaves a decision as to what should happen to the corresponding row data for the employee in the Upcoming_Promotions table. This gets to the heart of the reason for why **UPDATES** and **DELETES** when combined with Foreign Key Constraints can make it pretty complicated to handle. There are a few options to handle this edge case scenario:

Option 1 - Cascade the Deletion
This is known as “on-delete-cascade” whereby the deletion occurs from the reference table to the referencing table(s). This is conceptually the cleanest method in handling this edge case scenario. The syntax would look like the below:

```
CREATE TABLE Upcoming_Promotions (
        EmployeeID INT NOT NULL,
        CurrentTitle VARCHAR(50),
        NewTitle VARCHAR(50),
        CONSTRAINT fk_upcomingpromo_current,
        FOREIGN KEY(EmployeeID),
        REFERENCES Current_Employees(EmployeeID),
        ON DELETE CASCADE
);
```

The **ON DELETE CASCADE** keyword in the table definition will tell the DBMS to perform a cascade deletion. However, cascade deletions are quite complicated and can lead to significant hit in the database performance. There are two reasons for this first, each time a row is deleted from a table the DBMS has to check whether there are any other tables which depends on the column of this row. This would need to be done on every column on the reference table. The second reason is because there could be potential further cascading deletions which can impact on performance.

Option 2 - Do Nothing at All
This is known as "on-delete-do-nothing" whereby it leaves an orphan tuple in the child table. This is the less complicated option and can be valid way of handling this edge case scenario.

This approach is not conceptually clean but it does make the job of the DBMS a lot simpler and can lead to more simple performant database queries especially when it comes to deleting. The disadvantage of this approach is that it leaves an orphan tuple (data row) in the table because the tuple in the referenced table will be deleted but not in the referencing table. This orphan tuple will need to be deleted by someone and this deletion may happen in the form of a periodic script which is updated/executed by the database admin. The syntax would look like the below:

```
CREATE TABLE Upcoming_Promotions (
        EmployeeID INT NOT NULL,
        CurrentTitle VARCHAR(50),
        NewTitle VARCHAR(50),
        CONSTRAINT fk_upcomingpromo_current,
        FOREIGN KEY(EmployeeID),
        REFERENCES Current_Employees(EmployeeID),
        ON DELETE NO ACTION
);
```

Option 3 - Set the Corresponding Value to NULL
This is known as "on-delete-set-to-NULL" whereby it leaves a NULL value in a **NOT NULL** column in the child table.

This approach is not conceptually clean or perfect option. The advantage of this approach is that it allows us to indicate that there is something special (or something not OK) with the tuple in the child table. The disadvantage of this approach can leave a NULL value in a column which was explicitly specified having a **NOT NULL** constraint - and probably the main reason for avoiding this approach. The syntax would look like the below:

```
CREATE TABLE Upcoming_Promotions (
        EmployeeID INT NOT NULL,
        CurrentTitle VARCHAR(50),
        NewTitle VARCHAR(50),
        CONSTRAINT fk_upcomingpromo_current,
        FOREIGN KEY(EmployeeID),
```

```
        REFERENCES Current_Employees(EmployeeID),
        ON DELETE SET NULL
);
```

Everything that is mentioned above about deletions are also applicable to updates, for example there is an “on-update-cascade” SQL keywords similar to that of the “on-delete-cascade” rule. If we wanted to add cascading to updates we would use the **ON UPDATE CASCADE** as seen below:

```
CREATE TABLE Upcoming_Promotions (
        EmployeeID INT NOT NULL,
        CurrentTitle VARCHAR(50),
        NewTitle VARCHAR(50),
        CONSTRAINT fk_upcomingpromo_current,
        FOREIGN KEY(EmployeeID),
        REFERENCES Current_Employees(EmployeeID),
        ON DELETE CASCADE,
        ON UPDATE CASCADE
);
```

Cascading is the best and logical option but it makes updates/deletes very slow, hitting the performance of the DBMS. The other options are quicker but also leave the database in an inconsistent state. Therefore, we should always weigh up each approach very carefully before deciding on the best option for the database being designed and the edge case scenario it may run into. To conclude, we can now appreciate the reasons for why updating and deleting on tables with Foreign Key Constraints can be quite complicated.

4.9 Check Constraints

The Check Constraint, as the name would suggest, will check whether the value in a particular column satisfies a condition or not. Below is a classic example using gender:

```
CREATE TABLE Students (
        studentID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        firstName VARCHAR(30) NOT NULL,
        gender CHAR(1),
        CHECK gender IN('M', 'F')
);
```

The Check Constraint allows a way to validate data that are being inserted before they are inserted into the database table. Not all databases support Check Constraints. Check Constraints are very simple and useful but are very underused. If the database supports Check Constraints we should use them as it plays an important role in helping to maintain integrity of a database.

4.10 Indices

An index on a database column is not that different to that of a index in a book. It is a way to very quickly to access information at random from that column. The presence/absence of indices can make a dramatic difference to the the speed of the execution of queries.

It is helpful to understand how an index changes the way in which data is laid out on disk. We can have two tables with the exact dame data and run the exact same query but the query that uses index is much more faster to execute. The tables hold the same data but behind the scenes (on disk) they are laid out very differently.

We will take the example of a Sales_Data table seen below.

Sales_Data			
storeLocation	productName	salesDate	revenue
ASDA Coventry	Milk	November 02, 2020	1,233.32
ASDA Birmingham	Milk	November 01, 2020	6,849.99
ASDA Birmingham	Bread	November 02, 2020	5,434.74
ASDA Coventry	Milk	November 01, 2020	4,558.24
ASDA Coventry	Coffee	November 02, 2020	3,855.96
ASDA Coventry	Coffee	November 02, 2020	2,280.90
ASDA Birmingham	Bread	November 01, 2020	2,543.57
ASDA Birmingham	Coffee	November 02, 2020	2,110.95

If the Database Admin (DBA) indexed the Sales_Data by the salesDate the table will be indexed on disk something like the below:

salesDate
November 01, 2020
November 02, 2020

storeLocation	productName	salesDate	revenue
ASDA Birmingham	Milk	November 01, 2020	6,849.99
ASDA Coventry	Milk	November 01, 2020	4,558.24
ASDA Birmingham	Bread	November 01, 2020	2,543.57

storeLocation	productName	salesDate	revenue
ASDA Coventry	Milk	November 02, 2020	1,233.32
ASDA Birmingham	Bread	November 02, 2020	5,434.74
ASDA Coventry	Coffee	November 02, 2020	3,855.96
ASDA Coventry	Coffee	November 02, 2020	2,280.90
ASDA Birmingham	Coffee	November 02, 2020	2,110.95

The database will create an index (a type of data structure) that looks like above three tables i.e. a unique list of salesDate values and from that list there are pointers that point to all of the rows in the table which have that value of the date.
These bits of data are laid out contiguously on disk which makes input and output (I/O) to and from these disk locations extremely efficient.

Important Note: although the above simplifies conceptually the layout of the indexing, in reality the implementation by the DBMS is more likely to be complicated than this. The above is a stylistic representation of a particular type of index.

Not applying a index, the DBMS will layout the data randomly in the disk. Since the data has not been optimised in its layout this could lead to many disk I/O to retrieve the data which is a performance killer. Having a poorly optimised layout of a commonly run query can add hours and hours of reading especially as tables grow in size.

Setting up indices correctly can save hours and hours of time for the end user. The natural question that now arises is how should I setup indices correctly? The SQL command to setup an index is very simple:

```
CREATE INDEX Index_SalesDate ON Sales_Data(salesDate);
```

This command tells the DBMS to create an index called index_SalesDate (the name of the index helps us to **ALTER** or **DROP** the index later) on a particular column(s) of a particular table.

This command will pause the DBMS in order to go through the table row by row and calculate a data structure similar to that we saw above. This data structure (*which most likely will be in the form of a b+ tree*) will be added additionally, residing outside of the original table, stored on disk. Therefore, creating an index is a lot of work and takes a lot of time to create the data structure.

Important Note: you need not understand what a b+ tree because it is a complex data structure, but you should keep in mind that this is a balance tree which mean every time you insert a value to a table the index tree also needs to be rebalanced i.e. updating any table which has an index on it is very time consuming.

If you believe there is a column in a table that will be queried a lot, you should create an index on it at the same time as you create the table which will make it easier for the index to be created.

If we want to create an index on multiple columns from a table we simply list all the columns separated by commas within the round brackets as seen in the example below:

```
CREATE INDEX Index_Store_Product ON Sales_Data(storeLocation, productName);
```

Multi-column indices takes more time to create compared to single column indices but they also speed up the execution of more complicated queries.

Indices makes Read queries faster but they cause Update, Delete and Insertions to become really slow. This is a classic database tradeoff between speed of a query and the speed of updating the database. The underlying tradeoff reason is behind how indices are implemented using either a b+ tree or hash tables. Either of these data structures are very quick at lookup/reading data but are slow when updating the data (e.g. the B+ tree has to rebalance the tree when a new data is inserted or an existing data us updated/deleted).

Most DBMS by default will create indices on Primary Keys.

To conclude indices makes sense to be created on commonly queried columns of a table which can help the productivity of the end user querying the database.

Indices on a small table column would not make sense because indexes are created as an additional data structure that resides outside the table and therefore the overhead of setting uptake data structure and updating/maintaining it is probably less than that is required to scan through all of the contents of the small table. Furthermore, small tables are more likely to fit into contiguous disk locations making the benefit of having the speed of an index smaller.

It also does not make sense to create an index on a column containing a lot of NULL values. This is because an index creates a bucket of unique values within a column. If a column stores many NULLS then this will therefore defeat the purpose of an index.

Finally, it does not make sense to create an index on a table column which is written more often than it is read. This is because indexing is useful for querying/reading data but are very slow when updating/deleting/inserting data which will kill the performance of your database.

In summary indices are tied to the implementation of a specific DBMS and so it takes a lot of experience to create indices very well. Indexing a good database is more of an art than a science and the reason for why a skilled DBA is worth their weight in gold. You should keep this all in mind when working with more and more real world databases.

4.11 Stored Procedures

Stored Procedures are to Databases what Functions are to regular programming Code. This is an important analogy to remember. In other words a stored procedure is a way to group a bunch of SQL commands into a Unit, give it a name, specify parameters that goes into and comes out of the Unit and we can call on this Unit exactly like calling a function in coding.

Stored procedures are a little different to most of the other SQL commands we have explored in the previous sections. This is because SQL is by-and-large a declarative language i.e. we tell SQL what we want doing and not how it should be done. Stored procedures are procedural i.e. we tell SQL exactly what we want doing and how it should be done. Stored procedures have all the advantages of functions and are obviously an extension to the SQL programming language.

The syntax of stored procedures varies a lot from one DBMS implementation to another DBMS implementation. Therefore, you should ensure to know the intricacies of the stored procedure calls on the particular DBMS you happen to be using i.e. PostgreSQL, MySQL, SQL Server, Oracle, etc.

Having said the above none of the different implementation of stored procedures should differ from each other conceptually in the important elements. We will now explore an example syntax of declaring and calling a stored procedure.

```
CREATE PROCEDURE GetAnnualRevenue(
        @YEAR INT IN,
        @REVENUE DEC(10, 2) OUT
)
AS
BEGIN
        SELECT YEAR(salesDate), @REVENUE = SUM(revenue)
        FROM Sales_Data
        WHERE YEAR(salesDate) = @YEAR
        GROUP BY YEAR(salesDate)
END;
```

The above is a simple stored procedure that calculates the sum of the revenue for a particular year from the Sales_Data table. Note that the TIME and DATE functions/ operators varies from one DBMS to another and are system dependant.

The **CREATE PROCEDURE** call specifies the name and the parameters of the stored procedure just like we would in a function. The name of the stored procedure is used to call the stored procedure passing in the parameters as a comma separated list.
Each parameters are named with a preceding **@** sign to indicate a variable parameter. After naming the parameter we need to specify the type for the parameter followed by either the **IN** or **OUT** keyword. The **IN** keyword tells the SQL interpreter that the parameter value will be passed into the function as an input and should not be passed

out while the **OUT** keyword tells the SQL interpreter that this parameter value is an output value which is set in the body of the procedure. Some DBMS have **INOUT** parameters which are parameters that are passed into the procedure, modified by the body of the procedure and then output the value.

The body of the stored procedure is written between the **BEGIN** and **END;** keywords. Noticed that we also use the **AS** keyword between the signature and body of the stored procedure.

The body of the procedure is the usual except for the way the parameters are used. The **OUT** parameters must be assigned in the body of the stored procedure. In the above the @REVENUE parameter is assigned the sum of the revenue column. The code which invoked the stored procedure can use this value to do whatever it would like.
On the other hand, **IN** parameters can be used like constants. In the above we check whether the year of a particular date is equal to what has been passed in as the **IN** parameter for @YEAR.

The similarities of stored procedures and functions are unmistakable and if you have used older programming language than C you might be familiar with the concept of procedures which explicitly takes in parameters even for the values the procedure will pass back out.

To call the above procedure the syntax would look something like the below:

```
DECLARE Revenue DEC(10, 2);
EXEC GetAnnualRevenue @YEAR 2015, @REVENUE Revenue;
```

The first statement is a variable declaration. We use this syntax to declare any variable and it's type that we want create. The second statement uses the **EXEC** command to execute the stored procedure by its name. We then pass in a comma separated list of parameters that the stored procedure expects. It is good practice to explicitly specify what value corresponds to which parameters. This is done by tagging each value with the parameter name i.e. @YEAR 2015 explicitly assigns the value of 2015 to the @YEAR parameter. The output value of @REVENUE from the stored procedure body will then be stored in the variable Revenue we declared in the first statement.

Stored Procedures are very powerful and allows us to have all the advantages of functions i.e. reusability and dynamic code using SQL commands and parameters and encapsulation of logic into appropriately named stored procedures.

Since stored procedures are inherently a procedurally programming language construct, it is also often used for use-cases such as scheduling a bunch of database commands at various periodic intervals/schedules for example generating a business reports at the start/end of day outside core business hours.

4.12 Triggers

Triggers are a way to trigger an action when a condition is met. We could say that a trigger is a system executed action which is a side effect of a modification to the database. A trigger has two components, the action of the trigger i.e. what needs to be executed and the condition of the trigger i.e. what needs to be satisfied before the action is executed. Not all databases supports triggers for example MySQL does not support them. Triggers are considered more advanced features of a DBMS.

Trigger actions could be anything i.e. stored procedure or SQL command. For example, we could have a trigger to write a warning to a log table anytime a product price exceeds a certain number. The example syntax below is generic and the implementation will vary between DBMS to DBMS.

```
CREATE TRIGGER Cap_Price
AFTER INSERT OR UPDATE OF Price ON Products
FOR EACH ROW
WHEN (NEW.Price > 10000)
BEGIN
        INSERT INTO TABLE PriceExeededWarnings
        (WarningMessage, ProductID, NewPrice, OldPrice)
        VALUES
        ('Warning - Price Limit Trigger', NEW.productID, NEW.Price, OLD.price)
END;
```

The **CREATE TRIGGER** keyword is used to create the trigger of the name specified. We can use the trigger name to **ALTER** or **DROP** the trigger at a later date.

The second line defines the condition when the trigger should be executed. Typically speaking triggers will be executed after modifications i.e. **INSERT**, **UPDATE** and **DELETE** on a particular table. It is not usually the case to specify a trigger on a query and is not how triggers work. In the above example we are specifying the trigger should be evaluated **AFTER** an **INSERT** or an **UPDATE** of the column price in the table Products. We could have used the **BEFORE** keywords to check before an **INSERT** or an **UPDATE** of the column price in the table Products.
When we ask for a trigger to be checked **AFTER** a particular action the database will make both the old and the new values of the data available to us in temporary variables. If it is a **BEFORE** trigger there are no need for temporary variables. The **AFTER** triggers are therefore more resource intensive because they require data to be copied.

It is really important to recognise the trigger action has in a sense become part of the Insert/Update/Delete command. For instance, if the trigger condition succeeds and the trigger body is executed but throws an error, that error will cascade down to make the Insert/Update/Delete command to become unsuccessful. Therefore, you should be very careful when using triggers.

The third line of code signifies that the trigger is a row-level trigger. A trigger can be either a row-level or table-level trigger depending on whether the condition affects a single row or the table as a whole. Instead of using **FOR EACH ROW** we would use **FOR EACH STATEMENT** instead.
Depending whether the trigger is a row-level or table-level trigger, the temporary variable that are created and available inside the trigger vary. For instance, in a row-level trigger the body of the trigger can reference old and the new values of the specific row while in a table-level trigger the temporary variable would have been for the entire table. This therefore means that triggers can be quite onerous and resource intensive because it requires a lot of data to be stored in temporary variables.

The forth line defines the condition to evaluate on. We can call on a particular column using the dot operator to check the **NEW**/**OLD** value of a particular row. In the example above the condition is whether the new price of the product is greater than 10,000.

Finally, if the condition is satisfied, the command between the **BEGIN** and **END;** keywords will be executed. In the above example, the trigger action is to write the warning to the log table which is called PriceExeededWarnings.

This above is a very common use case for triggers i.e. to flag users that something has happened. This is really easy to do using Triggers and not so much using Check Constraints. For example, we could have prevented the price ever exceeding 10,000 and that insert/update would have failed using a Check Constraint. However, using a trigger we can allow the change to go through but also flag that change for someone else to review and decide whether to keep the change or update back to the original value.

Another use-case for triggers is to keep track of inventory within a database for example if an inventory drops below a certain threshold the trigger will write an order in some other re-ordering table to replenish the stock.

Despite the flexibility and the power of triggers, triggers are actually not commonly used because that power also makes triggers very complicated. Triggers seem really useful but not all DBMS support them and they are actually rarely used.

There are three downsides to Triggers:

- First, the failure of the **INERT**, **UPDATE** or the **DELETE**, if an error is thrown in the body of the trigger this can lead to an action the user performing the action does not understand because they are unaware of the underlying trigger. This can lead to very mysterious bugs which are hard to debug.
- Secondly, when tables are bulk-loaded/bulk-updated this can potentially cause the execution of the trigger on every row which would kill the database performance. In order to carry out a bulk load/update the DBA would have to first turn off the triggers, bulk load/update and then turn back on the triggers. This process is complicated and error prone.
- Finally, Triggers can launch the executions of even more triggers causing a chaining effect which could potentially lead to an infinite loop. The DBMS needs to be robust enough to detect an infinite loop and to terminate it somehow. This is why the complexity of DBMS increases when it supports triggers and also the reason for why some DBMS simply do not support this feature.

In summary Triggers are a powerful tool at our disposal but we should always be careful when using them. Triggers should only be used when it is absolutely necessary/required and we understand the implications of the trigger action that is being created. In most cases there is usually not much more needed to do beyond what a Constraint and Stored Procedures already provide to us.

4.13 Transactions

A Transaction is a unit of commands or set of activities that comprise a logical unit. The DBMS groups actions (SQL commands) into units called transactions and then ensures each transactions satisfies four ACID properties.

ACID stands for atomicity, consistency, isolation and durability. These four properties when taken together ensures that the database does not do anything weird or unpredictable. Maintaining the ACID principals in databases are extremely important in database and computer science theory in general. The focus in this section is more on how the properties impact the lives of users.

Atomicity:
A transaction is all-or-nothing unit approach. Either all are executed nor none at all — it cannot be partially by executed.

Consistency:
The data must stay consistent i.e. no constraints are violated, data is not corrupted or randomly lost or modified. Carrying out the transaction by itself cannot leave the database in an inconsistent state. For example, if we were modelling a bank system, it is not possible for a transaction to magically create money in an account, it can only transfer money from one persons bank account to another persons account.

Isolation:
Every transaction during its lifetime is isolated from the effects of all other transactions even if they are being executed concurrently on the database. Isolation is a huge topic on its own and it is implemented via concurrency which is similar to concurrency on multi-threaded programming. Databases are the ultimate concurrent application whereby millions of individual users are hitting the database simultaneously reading/writing on the same values. The database is still able to ensure that concurrency works flawlessly.

There are a whole bunch of concurrency control mechanism, the most common are: locking (lock protocols), timestamp ordering schemes, validation techniques and multi-versions schemes.
Databases typically implements concurrency using some form of locking. In other words a transaction can lock specific data and make sure that the other transaction are not able to interfere with those data items whilst the transaction is working on them. There are read and write locks and concurrency locking is a huge topic in itself, for example the most common locking protocol is called the two-phase locking. Two-phase locking refers to a way of carrying out a transaction of which a transaction first acquires a whole bunch of locks i.e. it secure exclusive access to a bunch of data, it then executes whatever it is that it needs to execute and then it releases those locks. Doing so ensures that other

transactions are not able to interfere with this transaction. However, locking protocols can lead to problems most notably that of a deadlock i.e. two transactions each hold locks that the other transaction requires and consequently neither transactions are able to complete. In such a circumstance the only option is to rollback i.e. to cancel/give up one or both of these transactions and try again.

Durability:
Once a transaction is complete its effects are permanently reflected in the database. This step of completing a transaction is called a transaction commit. Once the transaction has been committed the updates carried out by the transaction cannot be lost no matter what happens even if there are a hardware or system failures.

These four ACID properties are guaranteed to be satisfied for every transactions individually. This guarantee simplifies what a database has to do in order to maintain these properties for the database as a whole.
If a database can guarantee the ACID for one transaction then given a whole series of transactions, the DBMS nearly needs to decompose those steps into a schedule which corresponds to a serial execution schedule of those transactions one after another. In other words the DBMS then has to deal with the far simpler problem of decomposing a bunch of transactions into schedule which corresponds to a serial execution of those operations in the transactions. Once a schedule has been drawn up the DBMS is then able to make sure that nothing weird happens while executing the schedule.
The above intentionally glosses over the fine details of concurrency and serialisation of transactions, but should hopefully provide a sense that the ACID properties are really important and how a DBMS maintains these properties is really complicated.

As a user of the database, all we need to remember are three simple SQL commands:

START TRANSACTION;

This command tells the database or the engine that a transaction is starting i.e. a new unit of work which we would like to place into a logical unit. This lets the database know that all the operations/commands that follow this **START TRANSACTION** command will go inside of the serialisation schedule i.e. this command kicks off a new transaction. Hopefully, the database is able to serialise all of the transactions successfully and in which case by the end we can commit.

COMMIT;

This SQL command signals a transaction should be saved to the DBMS i.e. written in stone. Committing a transaction signals that the schedule (i.e. set of operations in the particular unit) has been performed successfully. The effects of the transaction should now be saved to the DBMS. All other transactions after this transaction will see the effects of the completed transaction.

ROLLBACK;

There also exists the possibility that things can go wrong. It is entirely possible that a bunch of transactions are being run concurrently and the DBMS is simply unable to find the serial schedule — it might also be that deadlock has resulted i.e. transactions have locks that other transactions need and so one of these transactions needs to be rolled

back. The **ROLLBACK** SQL command is like an Undo. It signals a transaction did not work out and it effects should be undone like it never happened. Rolling back a transaction is really complicated as it requires aborting the transaction i.e. undoing all of its actions but it does not end there. It also requires every other transaction which happen to see any of the effects of the aborted transaction also need to be aborted. Therefore, the serialisation schedule also needs to be recoverable for example: If transaction A sees the of transaction B but that transaction is aborted then Transaction A will also need to be automatically rolled back by the DBMS.

The user of a database do not need to bother themselves with this details because all of this should be done by the DBMS for them. This is the reason for why we take the Transactions functionality for granted. In general, DBMS do an excellent job of satisfying the ACID properties. They make sure that transactions can be interviewed seamlessly, ensure concurrency control can work out of the box, ensure that millions of transactions can happen concurrently and rollbacks are extremely rare. DBMS serialises these transactions and ensure aborts are not cascading. This is done by only allowing transactions to read committed data. Every now and then deadlock or some such issue occurs and that is when we realise how really important the ACID properties are.

4.14 Update and Delete

The **UPDATE** and **DELETE** commands allow us to update and delete data that is sitting in a pre-existing table. The **UPDATE** and **DELETE** commands act on the the rows of an existing table but not the columns. Changing the columns of a table is a more structural and permanent change than changing the rows. The syntax and semantics for updating and deleting data are similar to one another.

```
DELETE
FROM Sales_Data
WHERE storeLocation = 'ASDA Binley';
```

The **DELETE** keyword is used to tell the DBMS to delete data from the database. The **FROM** clause tells the DBMS which table to delete rows from and finally the **WHERE** clause tells the condition that needs to satisfied by the table rows in order to delete it from that table.

The syntax looks exactly the same as the **SELECT** statement but using the **DELETE** keyword instead. It is advisable to always run a select statement to select the data you are about to delete. This allows you to know exactly what data you are selecting before running the **DELETE** command. Deleting data from the database is permanent and cannot be undone. You could save the outputs of the **SELECT** statements in a temporary table incase you need to undo the results of the database operation.

```
DELETE
FROM Sales_Data;
```

Omitting the **WHERE** clause will delete all data rows from a table i.e. clear the table.

This is because there are no conditions to meet and so we are telling SQL to delete every table rows that exists in the specified table. This is a very simple syntax for deleting all data from the table leaving the table entirely empty.

```
UPDATE Sales_Data
SET storeLocation = 'ASDA Keresley'
WHERE storeLocation = 'ASDA Binley';
```

The syntax for updating a table uses the **UPDATE** keyword followed by the table name to tell SQL which table to update. The **SET** keyword is followed by the column name and the new constant value we wish to update to. Finally the **WHERE** clause is used to provide a condition for the u **UPDATE** command. Only those rows which satisfy the condition will be updated to the new value within the **SET** clause.
The above example will update all rows that have a storeLocation of ASDA Biley to the new updated value of ASDA Keresley.

```
UPDATE Sales_Data
SET revenue = revenue * 1.2;
```

Again, omitting the **WHERE** clause will update all row data column value to the new value specific in the **SET** clause. In the above example, we can use expression values (and not only constant values) in the **SET** clause when updating the data — i.e. we can calculate the new updated value rather than hardcoding a value.

To conclude, never run an **UPDATE** or **DELETE** command at work without first:

- Running the corresponding SELECT statement first, and
- Preferably saving the SELECT statements results to a temporary table as a backup

This is not something that is asked in interviews but forgetting these steps can lose business it's data and cost you your job.