Introduction

Welcome

The goal of this course is to be able to use SQL effectively within database environments.

This course covers how databases are organised and how relational databases work.

CRUD stands for: * Create * Read * Update * Delete

02. SQL Overview

About the overview

This chapter will be a quick intro to SQL, it will not be exhaustive.

SQL is a language designed to interact with relational databases.

An example of a statement:

```
SELECT * FROM Countries WHERE Continent = 'Europe';
```

SQL statements are terminated with a semi colon `;`

The 4 fundamental functions of a database are: * Create * Read * Update * Delete

The SELECT statement is how you retrive data from the database.

The INSERT statement is used to add a row to a table.

```
INSERT INTO Customer (name, city, state)
VALUES ('Jimmy Hendrix', 'Renton', 'WA');
```

The UPDATE statement is used to change data that is already present in the database. This is often used with a WHERE clause to identify the rows that need changing.

```
UPDATE Customer
    SET
        Address = '123 Music Avenue',
        ZIP = '90210'
    WHERE id = 5;
```

The DELETE statement is used to remove rows from a table in a database.

```
DELETE FROM Customer WHERE id = 4;
```

Database Organisation

The aim of a database is to organise data in a way that can be accessed.

A database is made up of one or more tables.

A single table is made up of rows and columns. A single table has 2 dimensions.

A row is a single line in a table. Also may be reffered to as a record.

A column is like a field.

All tables must have a unique key. The unique key of a table may or may not be hidden.

When a column in the table is used as the unique key, this column is referred to as the primary key.

Keys are used to create relationships between tables.

A foreign key is when there is a column in the table that is the unique key in another table. For example the 'Customer ID' field in the Sales table is a foriegn key because the 'Customer ID' field is the primary key in the Customer table.

Foreign keys make it possible to use joined queries.

The Select Statement

The SELECT statement retrieves data from a database.

```
SELECT 'Hello, World';
```

The AS keyword creates an alias for the column

```
SELECT 'Hello, World' AS Result;
```

An asterix in the SELECT statement is a wildcard match. In this usage it is saying to return all of the columns (ans all of the rows) from the Country table

```
SELECT * FROM Country;
```

To sort the results we can add the ORDER BY keywords to the statement.

```
SELECT * FROM Country ORDER BY Name;
```

Instead of selecting all of the columns we can just list out the ones we want to just pull those.

```
SELECT Name, LifeExpectancy FROM Country ORDER BY Name;
```

We can also change the name of the columns in the results table. Making the LifeExpectancy field more readable

```
SELECT Name, LifeExpectancy AS "Life Expectancy" FROM Country ORDER BY Name;
```

In the above example many database systems will allow the use of single quotes instead of double quotes for the alias identifier.

Selecting Rows

Use SELECT with a WHERE clause to select specific rows from a table.

The example below looks for rows where the Continent column matches the literal string 'Europe'.

```
SELECT Name, Continent, Region FROM Country WHERE Continent = 'Europe';
```

Sorting is still possible when using the WHERE clause

```
SELECT Name, Continent, Region FROM Country WHERE Continent = 'Europe' ORDER BY Name;
```

ORDER BY always comes **after** the WHERE clause.

To just return the first n rows of the results the LIMIT clause can be used

```
SELECT Name, Continent, Region FROM Country WHERE Continent = 'Europe' ORDER BY Name LIMIT 5;
```

To get the second 5 rows we can use the OFFSET clause

```
SELECT Name, Continent, Region FROM Country WHERE Continent = 'Europe' ORDER BY Name LIMIT 5 OFFSET 5;
```

LIMIT and OFFSET are not present in all SQL implementations.

Selecting Columns

All columns can be returned by using the *. The following will select all columns from the Country table

```
SELECT * from Country;
```

It is possible to have only certain columns returned by listing out those columns in the SELECT statement.

```
SELECT Name, Continent, Region from Country;
```

It is possible to have the columns name changed in the table that is returned to you. This is achieved by using the AS clause.

```
SELECT Name AS Country, Continent, Region from Country;
```

Columns can appear in any desired order. Simply list the columns in the SELECT statement as you would like them to appear in the table

```
SELECT Continent, Region, Name AS Country from Country;
```

Counting Rows

Counting rows can be done in SQL by using the COUNT function.

```
SELECT COUNT(*) FROM Country;
```

The above SQL returns a single row with the number of rows that the Country table has.

How many countries have a population greater than 1 million?

```
SELECT COUNT(*) FROM Country WHERE Population > 1000000;
```

Multiple conditions can be used in the WHERE clause by separating them with AND

```
SELECT COUNT(*) FROM Country WHERE Population > 100000000 AND Continent = 'Europe' ;
```

Using a Column name in the COUNT function will return the number of rows where that column has a value (not blank). This does not count rows that are NULL

```
SELECT COUNT(LifeExpectancy) FROM Country;
```

Inserting data

INSERT INTO is used to put data into the database.

```
INSERT INTO customer (name, address, city, state, zip) VALUES ('Fred Flintstone', '123 Cobblestone Way', 'Bedrock', 'CA', '91234');
```

When inserting data it is not necessary to list all of the columns.

```
INSERT INTO customer (name, city, state) VALUES ('Jimi Hendrix', 'Renton', 'WA');
```

Any columns that are not updated will have a NULL placed in them, indicating that no value was entered into that particular row and column.

Updating data

Using the UPDATE statement it is possible to change already existing data in a database table

```
UPDATE customer SET address = '123 Music Avenue', zip = '98056' WHERE id = 5;
```

It is also possible to set existing values to NULL

```
UPDATE customer SET address = NULL, zip = NULL WHERE id = 5;
```

Using the WHERE clause in an UPDATE statement allows you to specify which rows you would like to change.

Deleting data

It is possible to delete rows using the delete statement.

Best preactice is to select the rows you want to delete and inspect them to emsure you are deleting what you mean to.

```
SELECT * FROM customer WHERE id = 4;
```

Once happy with the rows to delete, then perform the delete statement

```
DELETE FROM customer WHERE id = 4;
```

Using the WHERE clause with a delete statement identifies which rows you would like to delete.

03. Fundamental Concepts

Creating a table

The CREATE statement can be used to create tables within a database.

The below SQL creates a table with 2 columns.

```
CREATE TABLE test (
  a INTEGER,
  b TEXT
);
```

Different databases support different datatypes.

Deleting a table

The DROP TABLE statement is used to delete tables from a database.

```
DROP TABLE test;
```

If it is unclear that the table to be deleted is in the database or not then can nest the DROP TABLE function in an IF EXISTS clause. If the table is there it will be deleted. This method is used to help avoid errors.

```
DROP TABLE IF EXISTS test;
```

Inserting data

The INSERT INTO is used to add data to a database table.

```
INSERT INTO test VALUES ( 1, 'This', 'Right here!' );
INSERT INTO test ( b, c ) VALUES ( 'That', 'Over there!' );
```

To add an empty row use the DEFAULT VALUES clause

```
INSERT INTO test DEFAULT VALUES;
```

It is possible to use a select statement to add data into a table. This can insert multiple rows into the table

```
INSERT INTO test ( a, b, c ) SELECT id, name, description from item;
```

The example above performs the SELECT statement, then the results from that are placed in test table in columns a, b, and c.

Deleting rows

It is a good idea to aution the WHERE clause in the delete statement before actually deleting.

```
SELECT * FROM test WHERE a = 1;
```

Then once happy can actually delete the data

```
DELETE FROM test WHERE a = 1;
```

Deleted rows can not be easily recovered.

The NULL value

NULL is a special state for a result with no value

NULL is the lack of a value.

It is not possible to test for NULL values in the traditional way. The follwong will not return any rows.

```
SELECT * FROM test WHERE a = NULL;
```

To test for a NULL value we need to test using a IS NULL clause

```
SELECT * FROM test WHERE a IS NULL;
```

To test the inverse ie rows where there is actually a value can test using the IS NOT NULL clause

```
SELECT * FROM test WHERE a IS NOT NULL;
```

We can specify in the creation of a table if a column is allowed to accept null values.

```
CREATE TABLE test (
  a INTEGER NOT NULL,
  b TEXT NOT NULL,
  c TEXT
);
```

Any attempt to add NULL values into column a or b will result in a failure.

Constraining Columns

Constraints are used to define certain rules and behaviours on certain columns within a table.

Constraints are set on table creation.

Not accepting NULL values into column c

```
CREATE TABLE test ( a TEXT, b TEXT, c TEXT NOT NULL );
```

Constraints can be used to set default values if no value is enrtered

```
CREATE TABLE test ( a TEXT, b TEXT, c TEXT DEFAULT 'panda' );
```

If a column needs to contain only unique values this can be set with UNIQUE. NULL values are often exempted from UNIQUE constraints. That is to say if a column has a unique constraint there can be multiple rows with NULL in them.

```
CREATE TABLE test ( a TEXT UNIQUE, b TEXT, c TEXT DEFAULT 'panda' );
```

Multiple contraints can be used on a single column

```
CREATE TABLE test ( a TEXT UNIQUE NOT NULL, b TEXT, c TEXT DEFAULT 'panda' );
```

Changing a schema

The ALTER TABLE statement can be used to modify a table after it has been created.

Adding a column 'd' to an existing test table

```
ALTER TABLE test ADD d TEXT;
```

ID columns

An ID column is a column that holds a unique value for each row within the table.

Typically ID columns are automatically populated.

Creation of ID column varies between each different database system.

The example below is how setting up an ID column works for SQLite.

```
CREATE TABLE test (
  id INTEGER PRIMARY KEY,
  a INTEGER,
  b TEXT
);
```

In SQLite setting the primary key as an integer means the database will handle populating this field and auto incrementing the values in this for you.

It is not necessary to insert values into an integer primary key.

Each table may have only one ID column.

Filtering data

SELECT statements are filtered using the WHERE clause.

The WHERE clause takes a boolean expression (True or False)

```
SELECT Name, Continent, Population FROM Country WHERE Population < 100000 ORDER BY Population DESC;
```

Also include rows that have NULL as the population

```
SELECT Name, Continent, Population FROM Country WHERE Population < 100000 OR Population IS NULL ORDER BY Population DESC;
```

To filter when both or multiple conditions are true use the AND command in the WHERE clause

```
SELECT Name, Continent, Population
FROM Country
WHERE Population < 100000 AND Continent = 'Oceania'
ORDER BY Population DESC;
```

The like operator can be used to perform string matching.

Percent signs work as a wild card and means any 1 or more characters.

```
SELECT Name, Continent, Population FROM Country WHERE Name LIKE '%island%' ORDER BY Name;
```

Underscores can be used in a like, this is used to match a single character

The in operator can be used to identify if row values are in a list of string literals

```
SELECT Name, Continent, Population FROM Country WHERE Continent IN ( 'Europe', 'Asia' ) ORDER BY Name;
```

Removing Duplicates

There are times when you want to know only the range of values contained in a column. SELECT DISTINCT can do this for you.

```
SELECT DISTINCT Continent FROM Country;
```

SELECT DISTINCT can also be used to get the distinct combinations of 2 columns

```
SELECT DISTINCT a, b FROM test;
```

The above query will return all the distinct value pairs that columns a and b have.

Ordering output

The ORDER BY clause is used to sort results from a query

```
SELECT Name FROM Country ORDER BY Name;
```

The order can be specified by passing either ASC or DESC to enforce whether the column being sorted on is in ascending or descending order

```
SELECT Name FROM Country ORDER BY Name ASC;

SELECT Name FROM Country ORDER BY Name DESC;
```

Passing multiple columns to the ORDER BY clause nests the ordering according to the order in which they were passed.

The following example sorts on continent and then within each continent sorts by name.

```
SELECT Name, Continent FROM Country ORDER BY Continent, Name;
```

It is possible to specify whether each column passed to the ORDER BY clause is in ascending or descending order independently.

In the below example Continent is sorted in a descending order while Region and Name are both in ascending order.

```
SELECT Name, Continent, Region FROM Country ORDER BY Continent DESC, Region, Name;
```

Conditional expressions

conditional expressions in sql are cumbersome.

In SQL: * 1 is considered true * 0 is considered false * anything non zero considered true * empty string considered false * any string considered true

The below example is testing if a and b themselves are true or false.

```
SELECT
    CASE WHEN a THEN 'true' ELSE 'false' END as boolA,
    CASE WHEN b THEN 'true' ELSE 'false' END as boolB
    FROM booltest
;
```

The example below is comparing a and b to the integer 1

```
SELECT
  CASE a WHEN 1 THEN 'true' ELSE 'false' END AS boolA,
  CASE b WHEN 1 THEN 'true' ELSE 'false' END AS boolB
  FROM booltest
;
```

Relationships

Understanding join

Some tables contain information relating to other tables.

JOINs can be used to combine data from multiple tables in a simple query.

ID columns are usually used in joins to match rows across tables.

The INNER JOIN is the most common form of a JOIN and is the default.

The result of an INNER JOIN will return results from both tables where the join condition is met.

LEFT OUTER JOIN includes rows from both tables where the JOIN condition is met **plus** the remaining rows from the left table.

RIGHT OUTER JOIN includes rows from both tables where the JOIN condition is met **plus** the remaining rows from the right table.

Many databases don't support right joins.

A right join can be re-written as a left join by changing the order of the tables being joined on.

A FULL OUTER JOIN combines the effects of left and right outer joins.

https://www.codeproject.com/KB/database/Visual_SQL_Joins/Visual_SQL_JOINS_V2.png

Accessing related tables

Example query

```
SELECT l.description AS left, r.description AS right
  FROM left AS l
  JOIN right AS r ON l.id = r.id
  ;
```

'l.description' is the description column from the left table. 'r.description' is the description column from the right table.

the left table is aliases as 'l'. This allows us to use the shorthand 'l' to refer to that table.

Another JOIN example

```
SELECT s.id AS sale, s.date, i.name, i.description, s.price
  FROM sale AS s
  JOIN item AS i ON s.item_id = i.id
  ;
```

Relating multiple tables

In database tables it is common to have many to many relationships.

A single customer may buy multiple products, and a single product may be bought by multiple customers.

This is often handled by making use of a junction table.

In the below example the sale table is the junction table as it has in it the id for both customers and items

```
SELECT i.name AS Item, c.name AS Cust, s.price AS Price
  FROM sale AS s
  JOIN item AS i ON s.item_id = i.id
  JOIN customer AS c ON s.customer_id = c.id
  ORDER BY Cust, Item
```

Strings

About SQL strings

SQL has a number of operators to handle strings.

Strings in SQL are typically enclosed in single quotes.

To represent a single quote within a string is achieved with double single quotes.

Concatenating strings is achieved with double pipes as standard ||.

Different database systems have different concatanation functions.

Finding the length of a string

Most database systems have a function to determine the length of strings.

Typically this is a function called LENGTH.

```
SELECT LENGTH('string');
```

or in an actual query

```
SELECT Name, LENGTH(Name) AS Len FROM City ORDER BY Len DESC;
```

Selecting part of a string

SQL standard does not include a substring function, however all major database systems support a substring operation.

A substring function allows you to select parts of a string depending on the position of the characters.

```
SELECT SUBSTR('this string', 6);
SELECT released,
    SUBSTR(released, 1, 4) AS year,
    SUBSTR(released, 6, 2) AS month,
    SUBSTR(released, 9, 2) AS day
  FROM album
  ORDER BY released
```

The second argument to the substring function is the starting string.

The third argument to the substring function us the number of characters to select.

Removing spaces

Often with strings it may be necessary to remove leading or trailing whitespace.

This can be achieved with the TRIM function.

TRIM will remove spaces from the beginning and the end, not from the middle.

TRIM and its variants example

```
-- remove both leading and trailing whitespace
SELECT TRIM('   string   ');

-- remove leading whitespace
SELECT LTRIM('   string   ');

-- remove trailing whitespace
SELECT RTRIM('   string   ');

-- remove trailing or leading characters that isn't whitespace
SELECT TRIM('...string...', '.');
```

Forcing case

There are cases where it may be necessary to force strings to be either upper case or lower case.

In SQL there are the UPPER and LOWER functions to achieve this.

Examples

```
-- the upper function
SELECT UPPER(Name) FROM City ORDER BY Name;

-- the lower function
SELECT LOWER(Name) FROM City ORDER BY Name;
```

SQLite only forces to uppercase and lowercase for asci characters.

Characters with an accent for example are not converted.

Different databases may implement in a different way to achive transitioning non asci characters.

Numbers

Numeric types

Data types are almost never the same across different database systems.

There are 2 basic categories of numeric types.

Fundamental numeric types are: * Integer * Real

Integer types will fall into the following types * Integer(precision) * Decimal(precision, scale) * Money(precision, scale)

Decimals are normally integer values scaled up.

Real types fall in the following types: * Real (precision) * Float (precision)

Real types sacrifice accuracy for scale. This is to say they are capable of representing very large or very small numbers but only to a limited number of significant figures.

The following SQL query will determine A as not equal to B.

```
SELECT A, B, A = B FROM
  ( SELECT
```

```
    ( ( .1 + .2 ) * 10 ) as A,
    ( 1.0 + 2.0 ) as B
  );
```

What type is that value

The TYPEOF function is used to find the type of an expression.

TYPEOF function usage example:

```
SELECT TYPEOF( 1 + 1 );
```

Not all database systems support the TYPEOF function.

Integer division

When dividing one integer by another the result will always be another integer.

```
-- integer division
SELECT 1 / 2;
```

The above will give the answer 0.

To get a decimal answer one of the numbers needs to be converted into a real number

```
SELECT 1.0 / 2;

-- or
SELECT CAST(1 AS REAL) / 2;
```

Rounding numbers

Many database systems support the ROUND function.

With a single argument the ROUND function will round to the nearest whole number

```
SELECT ROUND(2.55555);
```

The second argument to the ROUND function specifies the decimal places.

```
SELECT ROUND(2.55555, 3);
```

Dates and times

Dates and times

Dates are typically represented in databases with most significant components of the dates first ie Year.

'2018-03-28 15:32:43'

This allows for faster sorting within the database.

Within databases dates and times are normally represented in UTC.

UTC is easily converted to local timezones for reporting and display purposes.

Common data types for dates and times within database systems include: * DATE * TIME * DATETIME * YEAR * INTERVAL

Date and time related functions

Date and time functions that work with SQLite

```
SELECT DATETIME('now');
SELECT DATE('now');
SELECT TIME('now');
SELECT DATETIME('now', '+1 day');
SELECT DATETIME('now', '+3 days');
SELECT DATETIME('now', '-1 month');
SELECT DATETIME('now', '+1 year');
SELECT DATETIME('now', '+3 hours', '+27 minutes', '-1 day', '+3 years');
```

Aggregates

What are aggregates

Aggregate data is derived from more than one row at a time.

SQL has powerful features for dealing with aggregate data.

the COUNT function is an example of aggregation within SQL

```
SELECT COUNT(*) FROM Country;
```

Aggregate functions in combination with groupby allow for aggregation statistics over subsets of the data.

```
SELECT Region, COUNT(*)
  FROM Country
  GROUP BY Region
;
```

The GROUP BY clause groups the results before applying the aggregate function.

This way the aggregate function is applied to groups of data rather than the entire table.

This kind of queries can be combined with ORDER BY to sort the resulting data.

```
SELECT Region, COUNT(*) AS Count
  FROM Country
  GROUP BY Region
  ORDER BY Count DESC, Region
;
```

The aggregate function results can be used to order the resulting data as in the above query.

The below query makes use of a HAVING clause. The HAVING clause is way of filtering the results **after** the grouping and aggregations have been performed.

```
SELECT a.title AS Album, COUNT(t.track_number) as Tracks
  FROM track AS t
  JOIN album AS a
    ON a.id = t.album_id
  GROUP BY a.id
  HAVING Tracks >= 10
  ORDER BY Tracks DESC, Album
;
```

And we can use a WHERE clause as well as a HAVING clause. The WHERE clause is used to filter data **before** the groupings and aggregations are applied.

```
SELECT a.title AS Album, COUNT(t.track_number) as Tracks
  FROM track AS t
  JOIN album AS a
    ON a.id = t.album_id
  WHERE a.artist = "The Beatles"
  GROUP BY a.id
  HAVING Tracks >= 10
  ORDER BY Tracks DESC, Album
;
```

The WHERE clause comes **before** the GROUP BY clause.

The HAVING clause comes **after** the GROUP BY clause.

Using aggregate functions

COUNT(*) is a way of counting the rows in a table.

```
SELECT COUNT(*) FROM Country;
```

COUNT(column_name) is a way of counting the non-null values in that column.

```
SELECT COUNT(Population) FROM Country;
```

Other aggregation functions include: * AVG * MIN * MAX * SUM

```
-- average population per region
SELECT Region, AVG(Population) FROM Country GROUP BY Region;

-- what are the min and max populations for countries in each region
SELECT Region, MIN(Population), MAX(Population) FROM Country GROUP BY Region;

-- what is the total population for each region
SELECT Region, SUM(Population) FROM Country GROUP BY Region;
```

Aggregating distinct functions

COUNT DISTINCT is a way of counting unique values in a column in SQL

```
SELECT COUNT(DISTINCT HeadOfState) FROM Country;
```

Transactions

What are Transactions

A transaction is a group of operations that are handled in one unit of work.

If any operation in a transaction fails then the whole transaction fails.

The database is then restored to the state it was in before the transaction began.

This is most applicable in finance where updating multiple tables makes sense to debit and credit the relevant accounts.

Transactions can sometimes improve performance.

For larger or more complex operations transactions can improve speed and reliability of the database.

Data integrity

Performing a transaction

```
BEGIN TRANSACTION;
INSERT INTO widgetSales ( inv_id, quan, price ) VALUES ( 1, 5, 500 );
UPDATE widgetInventory SET onhand = ( onhand - 5 ) WHERE id = 1;
END TRANSACTION;
```

If it is necessary to roll back to the database status before the transaction began, can use the ROLLBACK statement.

```
BEGIN TRANSACTION;
INSERT INTO widgetInventory ( description, onhand ) VALUES ( 'toy', 25 );
ROLLBACK;
```

Performance

A long list of inserts or updates are easily sped up by using transactions.

Triggers

Automating data with triggers

A trigger is an operation that is automatically performed when a specified database event occurs.

Each database system will have their own trigger syntax.

Example trigger creation

```
CREATE TRIGGER newWidgetSale AFTER INSERT ON widgetSale
    BEGIN
        UPDATE widgetCustomer SET last_order_id = NEW.id WHERE widgetCustomer.id = NEW.customer_id;
    END
;
```

A trigger can be an excellent way to enforce business rules that require a table to be updated when another table is updated.

Preventing updates

Triggers can be used to prevent data being updated.

The below query prevents rows in the widgetSale table being updated if the value in the reconciled column is 1.

```
CREATE TRIGGER updateWidgetSale BEFORE UPDATE ON widgetSale
    BEGIN
        SELECT RAISE(ROLLBACK, 'cannot update table "widgetSale"') FROM widgetSale
            WHERE id = NEW.id AND reconciled = 1;
    END
;
```

Triggers can be used to create timestamps

```
CREATE TRIGGER stampSale AFTER INSERT ON widgetSale
    BEGIN
        UPDATE widgetSale SET stamp = DATETIME('now') WHERE id = NEW.id;
        UPDATE widgetCustomer SET last_order_id = NEW.id, stamp = DATETIME('now')
            WHERE widgetCustomer.id = NEW.customer_id;
        INSERT INTO widgetLog (stamp, event, username, tablename, table_id)
            VALUES (DATETIME('now'), 'INSERT', 'TRIGGER', 'widgetSale', NEW.id);
    END
;
```

Views and Subselects

Creating a subselect

A subselect can also be described as a nested query.

The result of a SELECT statement is a table. This table can then be used as a data source in another SELECT statement. Using like this is called a subselect.

A regular SQL query

```
SELECT SUBSTR(a, 1, 2) AS State, SUBSTR(a, 3) AS SCode,
  SUBSTR(b, 1, 2) AS Country, SUBSTR(b, 3) AS CCode FROM t;
```

The same SQL query in a subselect

```
SELECT co.Name, ss.CCode FROM (
    SELECT SUBSTR(a, 1, 2) AS State, SUBSTR(a, 3) AS SCode,
      SUBSTR(b, 1, 2) AS Country, SUBSTR(b, 3) AS CCode FROM t
  ) AS ss
  JOIN Country AS co
    ON co.Code2 = ss.Country
;
```

Searching within a result set

Using a subselect to provide a list of rows to search in.

Find albums that have a track on it that is less than 90 seconds.

```
SELECT * FROM album
  WHERE id IN (SELECT DISTINCT album_id FROM track WHERE duration <= 90)
;
```

The above query finds the distinct album ids that have a track less than 90 seconds. This list is then matched with the album table to find the associated album name.

This method can also be used in JOINs. In the below example it is used to filter the album data before it is joined.

```
SELECT a.title AS album, a.artist, t.track_number AS seq, t.title, t.duration AS secs
  FROM album AS a
  JOIN track AS t
    ON t.album_id = a.id
  WHERE a.id IN (SELECT DISTINCT album_id FROM track WHERE duration <= 90)
  ORDER BY a.title, t.track_number
;
```

Using it on the left side of a join is also possible

```
SELECT a.title AS album, a.artist, t.track_number AS seq, t.title, t.duration AS secs
  FROM album AS a
  JOIN (
    SELECT album_id, track_number, duration, title
      FROM track
      WHERE duration <= 90
  ) AS t
    ON t.album_id = a.id
  ORDER BY a.title, t.track_number
;
```

Subselects can be used anywhere you would use a table.

Creating a view

In SQL it is possible to save a query as a view and use that view as if it were a table.

```
-- a query to extract track minutes and seconds
SELECT id, album_id, title, track_number, duration / 60 AS m, duration % 60 AS s FROM track;
```

Creating a view from the above query.

```
CREATE VIEW trackView AS
  SELECT id, album_id, title, track_number,
    duration / 60 AS m, duration % 60 AS s FROM track;
SELECT * FROM trackView;
```

This view can then be queried as per any regular table

```
SELECT * FROM trackView;
```

Queries that are often ran can be saved as views for easy access.

The view can then be accessed in all the ways a regular table is. They can be used in JOINs or in subselects themselves.

Views can be deleted similar to tables.

```
DROP VIEW IF EXISTS trackView;
```

Creating a joined view.

Views can be created from joined queries

```
-- a regular join

SELECT a.artist AS artist,
    a.title AS album,
    t.title AS track,
    t.track_number AS trackno,
    t.duration / 60 AS m,
    t.duration % 60 AS s
  FROM track AS t
    JOIN album AS a
      ON a.id = t.album_id
;
```

Making the query into a view

```
CREATE VIEW joinedAlbum AS
  SELECT a.artist AS artist,
      a.title AS album,
      t.title AS track,
      t.track_number AS trackno,
      t.duration / 60 AS m,
      t.duration % 60 AS s
    FROM track AS t
    JOIN album AS a
      ON a.id = t.album_id
;
```

The view can then be queried as per any regular table

```
SELECT * FROM joinedAlbum WHERE artist = 'Jimi Hendrix';
```

And also deleted as per a regular table

```
DROP VIEW IF EXISTS joinedAlbum;
```

Deleting a view does not affect any underlying tables.

Defined functions

Overview

There may be times when you want to use a function that doesn't exist in your database system.

Many database systems provide a way for users to define their own functions.

These are known as UDFs or User Defined Functions.

Different database systems my require UDFs to be written in the following languages: * C or C++ * Proprietary language * Scripting language (Python, Perl, PHP)

UDFs are typically used for the following: * Unit conversions (lbs to kgs) * format conversions (seconds into hours or vice versa) * Aggregations (rolling average)

SQLite accepts UDFs written in PHP.

Defining functions in PHP

Example function written in PHP

```
// SEC_TO_TIME( seconds INTEGER )
function sec_to_time( $sec )
{
    if(is_null($sec)) return NULL;
    $sec = intval($sec);    // make sure it's an integer
    $s = $sec % 60;
    $m = $sec / 60;
    return sprintf('%d:%02d', $m, $s);
}
```

Once defined this can then be used in any queries

```
-- basic usage
SELECT SEC_TO_TIME(320);

-- in an actual query
SELECT title, duration, SEC_TO_TIME(duration) FROM track;\
```

Web page showing how to create custom functions in sqlite using python: https://pynative.com/python-sqlite-create-or-redefine-sqlite-functions/