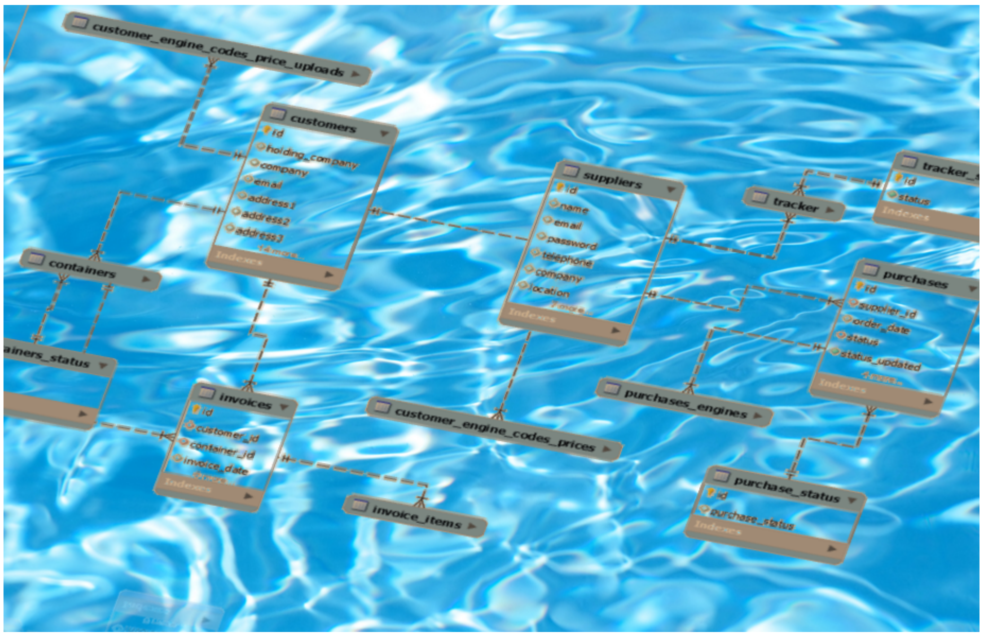
Data Analytics - Databases and SQL

Introduction

Welcome to this database and SQL tutorial.



The Structured Query Language is somtimes seen as the most important invention; well, right after the washing machine;).

It is custom to begin with a “Hello World” example:

```
select "Hello World";
```

1 records

“Hello World”
Hello World

Now try it yourself and press `Run Code`. You can also `Start Over` or have a look at the proposed `Solution`.

SQL Code Start Over Solution Run Code

```

select "hello"

```

1 records

“hello”
hello

Now try to display (with the keyword `select`) some basic calculations (1+2+3)*2.

SQL Code Start Over Solution Run Code

```

select (1+2+3)*2

```

1 records

(1+2+3)*2
12

You can concatenate strings and numbers by using `||`

SQL Code Start Over Solution Run Code

```
select "hello" || " world: 1+2+3 = " || (1+2+3)


```

1 records

"hello" || " world: 1+2+3 = " || (1+2+3)
hello world: 1+2+3 = 6

Now, after this little "ice-breaker," let us start learning SQL via an example.

The aim is to build a car sales system, and gain business insights.

First, we will need customers to buy cars. Of course, we must have cars that can be sold to the customers. The sales transactions need to be recorded as well. Hence, we need three tables: `customer` , `car` and `sale` .

We will use the SQLite database system and assume that an empty database exists. Note, that this tutorial will work for almost all relational databases such as PostgreSQL and MySQL. I have also provided a similar tutorial for `Access` . If you are interested in the popularity of database engines go to db-engines.com.

Table operations

We begin by creating a `customer` table.

SQL Code Start Over Hint Run Code

```
create table customer (info text)


```

Executing the above SQL command does not show anything. However, we can see the table using the `sqlite_master` table. The following *select query* displays the customer table name.

SQL Code Start Over Run Code

```sql
select name from sqlite_master


```

1 records

name
customer

We have not quite thought through this. Let us delete the table again.

SQL Code Start Over Run Code

```sql
drop table customer


```

What information do we need from a customer? Name, phone number, email and an address seem to be a reasonable starting point. How can we identify a customer? We could use the name as identifier. However, if there are two people with exactly the same name then there is an issue. A number to identify these individuals could resolve this. More generally using a unique integer number to identify a record is common practice. A unique identifier for a record is known as *primary key*. Note, it could be an integer number, but it could be any field (e.g. the customer's name assuming its uniqueness). The following SQL statement puts this all together:

SQL Code Start Over Solution Run Code

```sql
create table customer
(
  id int primary key,
  name text,
  phone_number text,
  email text,
  address_building text,
  address_street text,
  postcode text
)
```

Here, the field names are id, name, phone_number, email, etcetera. These are also known as column names. On the right of each field name is a *data type* specifier. Here, we used int (representing whole numbers, integers) and text (representing characters). Right next to the field `id` we wrote `primary key` . This is a *constraint* on the field, which means when inserting a record a unique value needs to be provided. To summarise a *field* contains a field name, its data type and a constraint.

Again, we can see the table in the master overview.

SQL Code Start Over Run Code

```
select * from sqlite_master


```

2 records

type	name	tbl_name	rootpage	sql
table	customer	customer	2	CREATE TABLE customer

(id int primary key, name text, phone_number text, email text, address_building text, address_street text, postcode text) | |index |sqlite_autoindex_customer_1 |customer | 3|NA |

Insert data

Now, let us insert a bit of test data.

SQL Code Start Over Run Code

```
insert into customer values
(1, 'Wolfgang','0779...','w.garn@surrey.ac.uk',
 '20MS02','UoS','GU2 7XH')
```

The above represents an entire *record*. A *row* in a table is known as *record* or *tuple*.

To verify that we have really inserted the above values we can execute the `select` query.

SQL Code Start Over Solution Run Code

```

select * from customer

```

1 records

id	name	phone_number	email	address_building	address_street	postcode
1	Wolfgang	0779…	w.garn@surrey.ac.uk	20MS02	UoS	GU2 7XH

Let us insert the customers with names Dominic, Michael and spook. When inserting incomplete records (i.e. some field values are missing) the column names (especially *required* fields, such as primary key) have to be specified.

SQL Code Start Over Run Code

```
insert into customer (id, name)
values (2,'Dominic'),(3,'Michael'),(4,'spook')

```

View the previously inserted records.

SQL Code Start Over Solution Run Code

```

select * from customer

```

4 records

id	name	phone_number	email	address_building	address_street	postcode
1	Wolfgang	0779…	w.garn@surrey.ac.uk	20MS02	UoS	GU2 7XH
2	Dominic	NA	NA	NA	NA	NA
3	Michael	NA	NA	NA	NA	NA
4	spook	NA	NA	NA	NA	NA

Changing data

Okay, we really don't want to have a "spook" customer. So, let us remove that customer.

SQL Code Start Over Run Code

```
delete from customer where name = "spook";


```

In order to display only the column `id` and `name` we replace the asterisk.

SQL Code Start Over Solution Run Code

```

select id, name from customer

```

3 records

id	name
1	Wolfgang
2	Dominic
3	Michael

Here, we see the "beauty" of SQL - it is intuitive and natural: "delete from my customer table where the name is spook" (at least somewhat natural). Important is that we used the `where` clause, otherwise all records would disappear.

Let us assume we need to "correct" (update) the customer with identifier 1 and add the surname "Garn."

SQL Code Start Over Run Code

```
update customer
set name = "Wolfgang Garn"
where id = 1
```

Observe the result of the update query.

SQL Code Start Over Run Code

```sql
select * from customer;
```

1 records

id	name	phone_number	email	address_building	address_street	postcode
1	Wolfgang	0779…	w.garn@surrey.ac.uk	20MS02	UoS	GU2 7XH

Now, assume we want a new field in the customer table, which identifies the `status` of being a current (someone in the middle of purchase), potential (showed some interest) or no-longer (moved on). This means we will alter (change) the table.

SQL Code Start Over Run Code

```sql
alter table customer
add status text
```

SQL Code Start Over Run Code

```sql
select * from customer;
```

3 records

id	name	phone_number	email	address_building	address_street	postcode
1	Wolfgang	0779…	w.garn@surrey.ac.uk	20MS02	UoS	GU2 7XH
2	Dominic	NA	NA	NA	NA	NA
3	Michael	NA	NA	NA	NA	NA

If we decide adding the field was not a good idea, it can be undone with `alter table customer drop status` .

Practice

Add the surname "Garn" to the names Michael and Dominic by using the `update` function twice and show your result. Note you can comment SQL code by using `-- ` (two hyphens and a space).

SQL Code Start Over Solution Run Code

```sql

```

Only display the custoner name.

SQL Code Start Over Solution Run Code

Update (not insert) the status to current, no-longer and potential for customers with identifier 1, 2 and 3 respectively.

$$Note : updatethreetimes, usecomments$$

SQL Code Start Over Solution Run Code

Display the id, name and status.

SQL Code Start Over Solution Run Code

Car Table

Now, that we have the customer table. Let us create a car table. So, that we can sell these type of cars to customers. What fields should we add? What car do you currently drive? How much was it? What fuel does it need? This gives us the field names: manufacturer, model, price and fuel type. We need to decide about the data types - `text` seems to be fine for all but the `price` column, where we will use `float` . Again, let us introduce the column `id` as primary key.

SQL Code Start Over Run Code

```sql
create table car (
  id int primary key,
  manufacturer text, -- Mazda
  model text, -- CX-5
  price float, -- £27,000
  fuel_type text -- diesel
)
```

Show the car table using the `sqlite_master` .

SQL Code Start Over Solution Run Code

Insert a few test-records.

SQL Code Start Over Run Code

```
insert into car values
(1,'Mazda'     ,'CX-5'  ,27000, 'diesel'),
(2,'Mazda'     ,'MX-30' ,35000, 'electric'),
(3,'BMW'       ,'i3'    ,35000, 'electric'),
(4,'BMW'       ,'2'     ,27000, 'petrol'),
(5,'Volkswagen','e-golf',27000, 'electric')
```

SQL Code Start Over Run Code

```
select * from car


```

5 records

id	manufacturer	model	price	fuel_type
1	Mazda	CX-5	27000	diesel
2	Mazda	MX-30	35000	electric
3	BMW	i3	35000	electric
4	BMW	2	27000	petrol
5	Volkswagen	e-golf	27000	electric

Insert two more records a Volkswagen - GTI with petrol, which costs £33k; and Ford - Fiesta Van with LPG, which costs £35k.

SQL Code Start Over Solution Run Code

```

insert into car values
(1,'Mazda'     ,'CX-5'  ,27000, 'diesel'),
(2,'Mazda'     ,'MX-30' ,35000, 'electric'),
(3,'BMW'       ,'i3'    ,35000, 'electric'),
(4,'BMW'       ,'2'     ,27000, 'petrol'),
(5,'Volkswagen','e-golf',27000, 'electric')

```

Display all car entries:

SQL Code Start Over Solution Run Code

```
select * from car
```

5 records

id	manufacturer	model	price	fuel_type
1	Mazda	CX-5	27000	diesel
2	Mazda	MX-30	35000	electric
3	BMW	i3	35000	electric
4	BMW	2	27000	petrol
5	Volkswagen	e-golf	27000	electric

Lookup Tables

In the previous table we use `fuel type` with several repetition. In order to avoid inconsistent spelling (e.g. electric, electrical or Electric). We could introduce a *lookup table*.

```
  create table fuel (type text primary key)
```

A lookup table only has one field, which is a primary key. Let us insert the lookup values:

```
insert into fuel
values ('diesel'),('electric'),('petrol'),('LPG')
```

Display the table:

SQL Code Start Over Solution Run Code

```

select * from fuel

```

4 records

type
diesel
electric
petrol
LPG

Foreign keys

Now the interesting part is, how do we "link" the `fuel` table with the `car` table. Obviously, it needs to be done using the common fields `car.fuel_type` and `fuel.type`.

By default, foreign keys are disabled in SQLite. hence, we need to enable them.

```
PRAGMA foreign_keys = ON;
```

We can check that the foreign keys functionality is on:

```
PRAGMA foreign_keys;
```

1 records

foreign_keys
1

The easiest way is to drop the table `car` and and create it again with a foreign key. Note, in other database systems (e.g. postgreSQL) you would do: `alter table car  add constraint constraint_fuel_type foreign key (fuel_type) references fuel(type)  on update cascade;`

```
dbCT_car(con)
```

```
## [1] 0
```

```
drop table car
```

```
create table car (
  id int primary key,
  manufacturer text, model text, price float,
  fuel_type text REFERENCES fuel(type) ON UPDATE CASCADE
  -- foreign key (fuel_type) references fuel(type)
)
```

We need to insert the data again.

```
insert into car values
(1,'Mazda'     ,'CX-5'  ,27000, 'diesel'),
(2,'Mazda'     ,'MX-30' ,35000, 'electric'),
(3,'BMW'       ,'i3'    ,35000, 'electric'),
(4,'BMW'       ,'2'     ,27000, 'petrol'),
(5,'Volkswagen','e-golf',27000, 'electric'),
(6,'Volkswagen','GTI'   ,33000, 'petrol'),
(7,'Fiesta Van','Ford'  ,35000, 'LPG')
```

Benefiting from a lookup table

Let us change the fuel type `diesel` to `Diesel` in the lookup table.

```
update fuel set type='Diesel' where type = 'diesel'
```

Observe the updated value in the `fuel` table.

```
select * from fuel
```

4 records

type
Diesel
electric
petrol
LPG

Did it update in the `car` table automatically?

SQL Code Start Over Solution Run Code

```

```

However, note that you cannot update the fuel_type in the `car` table:
`update car set fuel_type='Electric' where fuel_type = 'electric'` will through an error message.

Practice

Create a lookup table for the customer status.

```
create table customer_status
 (status text primary key)
```

Insert the values: current, potential and no-longer.

```
insert into customer_status
values ('current'), ('potential'), ('no-longer')
```

Dropping (e.g. `alter table customer drop column status`) or adding a constraint the current status column in the `customer` table, works for most database systems. However, SQLite does not support this. Hence, we drop the entire table and recreate it.

SQL Code Start Over Run Code

```
drop table customer
```

```
create table customer
(
  id int primary key,
  name text,
  phone_number text,
  email text,
  address_building text,
  address_street text,
  postcode text,
  status text REFERENCES customer_status(status) ON UPDATE CASCADE
)
```

Now we insert the test data one more time.

```
insert into customer values
(1, 'Wolfgang','0779...','w.garn@surrey.ac.uk','20MS02','UoS','GU2 7XH','current'),
(2, 'Dominic','','','','','','no-longer'),
(3, 'Michael','','','','','','potential')
```

```
select id, name, status from customer
```

3 records

id	name	status
1	Wolfgang	current
2	Dominic	no-longer
3	Michael	potential

Now, we have the customer and car table. Both, are linked to a lookup table via the foreign keys status and fuel_type respectively.

Sale Table

Next, we'd like to create a table for the sales. This table will need a foreign key to the customer table and another one to link to the car table. What other sale's information do we need. The date of the sale would be definitely good. A quantity field would be good assuming a car record represents "unlimited" supply of this type of car. Let us assume - for simplicity - that the price in the car table is a fixed retail price.

```
create table sale (
  id int, -- sale id
  customer_id int REFERENCES customer(id) ON UPDATE CASCADE,
  car_id int REFERENCES car(id) ON UPDATE CASCADE,
  sale_date text,
  quantity int
)
```

Note, SQLite does not have a specific data type for date/time, but `text` , `real` and `int` can be used (see sqlitetutorial.net/sqlite-date for examples).

Let us add three records to the sale-table. Let us say say Wolfgang purchased a BMW i3 on the 2nd of March 2021, and bought a Mazda CX-5 on 17th November 2014. Dominic bought a GTI on the 15th of August 2020.

```
 insert into sale values
 (1, 1, 1, '2014-11-17 10:00:00.0',1),
 (2, 1, 3, '2021-03-02 11:00:00.0',1),
 (3, 2, 6, '2020-08-15 17:00:00.0',1)
```

```
 select * from sale
```

3 records

id	customer_id	car_id	sale_date	quantity
1	1	1	2014-11-17 10:00:00.0	1
2	1	3	2021-03-02 11:00:00.0	1

id	customer_id	car_id	sale_date	quantity
3	2	6	2020-08-15 17:00:00.0	1

Queries

Now, it would be great to display the following information: all the sale records but with customer name, manufacturer, model and sale's date. This means we have to collect the information from the customer, car and sale table.

```
select name, manufacturer, model, sale_date
from customer, car, sale
where sale.customer_id = customer.id and
      sale.car_id = car.id
```

3 records

name	manufacturer	model	sale_date
Wolfgang	Mazda	CX-5	2014-11-17 10:00:00.0
Wolfgang	BMW	i3	2021-03-02 11:00:00.0
Dominic	Volkswagen	GTI	2020-08-15 17:00:00.0

The function `strftime('%Y-%m-%d %H:%M:%S', ...)` can be used to extract any specific date/time information. `date` and `time` are two more useful functions.

Now, that we have built a basic car-sales-system it is time to operate it. Okay, some time has passed and the tables have been filled with some data.

Operational Queries

What operational queries would be good to know? How to find a customer record (phone number ,email) by knowing "roughly" the name? How to insert a new type of car? We have done that before. How to add a new customer or sale record. Again we have done that.

So, let us find the customer which contains the characters "Wolf."

SQL Code Start Over Run Code

```
select name, email, phone_number from customer
where name like '%Wolf%'

```

1 records

name	email	phone_number
Wolfgang	w.garn@surrey.ac.uk	0779…

Business Insight Queries

What business insights would be interesting? How much revenue have we generated? Which cars are the top sellers?

For the revenue we need the car retail price and the quantity sold.

```
select price, quantity
from car, sale
where sale.car_id = car.id
```

3 records

price	quantity
27000	1
35000	1
33000	1

The total revenue is:

```
select sum (price* quantity) as revenue
from car, sale
where sale.car_id = car.id
```

1 records

revenue
95000

Which cars are the top sellers?

```
select manufacturer, model, sum(quantity) as nb
from car, sale
where sale.car_id = car.id
group by manufacturer, model
order by nb desc
```

3 records

manufacturer	model	nb
BMW	i3	1
Mazda	CX-5	1
Volkswagen	GTI	1

Summary

We introduced fundamental SQL statements by building a *car-sales-system* and showing how to use it. Concepts such as creating a table, inserting and updating its data were introduced. Simple queries and aggregates were mentioned.

Resources

- w3schools.com is a great systematic introduction into SQL.
- Shah (2020) is a hands-on introduction to data science (Chapter 7 introduces MySQL).

Acknowledgment

This tutorial was created using RStudio, R, rmarkdown, and many other tools and libraries. The packages `learnr` and `gradethis` were particularly useful. I'm very grateful to Prof. Andy Field for sharing his disovr package, which allowed me to improve the style of this tutorial and get more familiar with `learnr` . Allison Horst wrote a very instructive blog "Teach R with learnr: a powerful tool for remote teaching", which encouraged me to continue with `learnr` . By the way, I find her statistic illustrations amazing. Irene Steves' tutorial is great for learning how to use SQL in RStudio.

References

Shah, Chirag. 2020. *A Hands-on Introduction to Data Science*. Cambridge University Press.