# WEEK 3

SQL

# TODAY

- More Labs
- More DB Concepts
  - Foreign Keys
  - Indexes
- Views
  - Joins (inner & left)
- Parent Id Concept

# ALTER TABLE

- Command used to modify a database table after created
  - Can add columns
  - Drop columns
  - Modify columns
- Format

  Alter table [tablename]
  [Add] [columnname] OR
  [Drop] COLUMN [columnname] OR
  [alter/modify] COLUMN [columnname] [datatype]
- Example

```
Alter table products
add product_description varchar(500) null;


alter table products
drop column product_description;
```

**Lab**
Add new description field to products table
Note: Pay attention to wording!!

# DESTRUCTIVE DB COMMANDS: DROP & DELETE

## DELETE TABLE DATA

- Can manually delete specific rows or all data in a table
  - delete from [tablename] where [filter]
  - delete from products

## DROP TABLE STRUCTURE

- Manually deletes the entire table
- You must delete data from table before dropping
  - drop [tablename]
  - drop products;

# FK CONSTRAINTS

- The Foreign Key Constraint maintains data and referential integrity
  - Stops users from accidentally deleting data from the parent table
  - Can also enforce updating parent table when cascade delete \ cascade update

- In most standard DBs alter table products

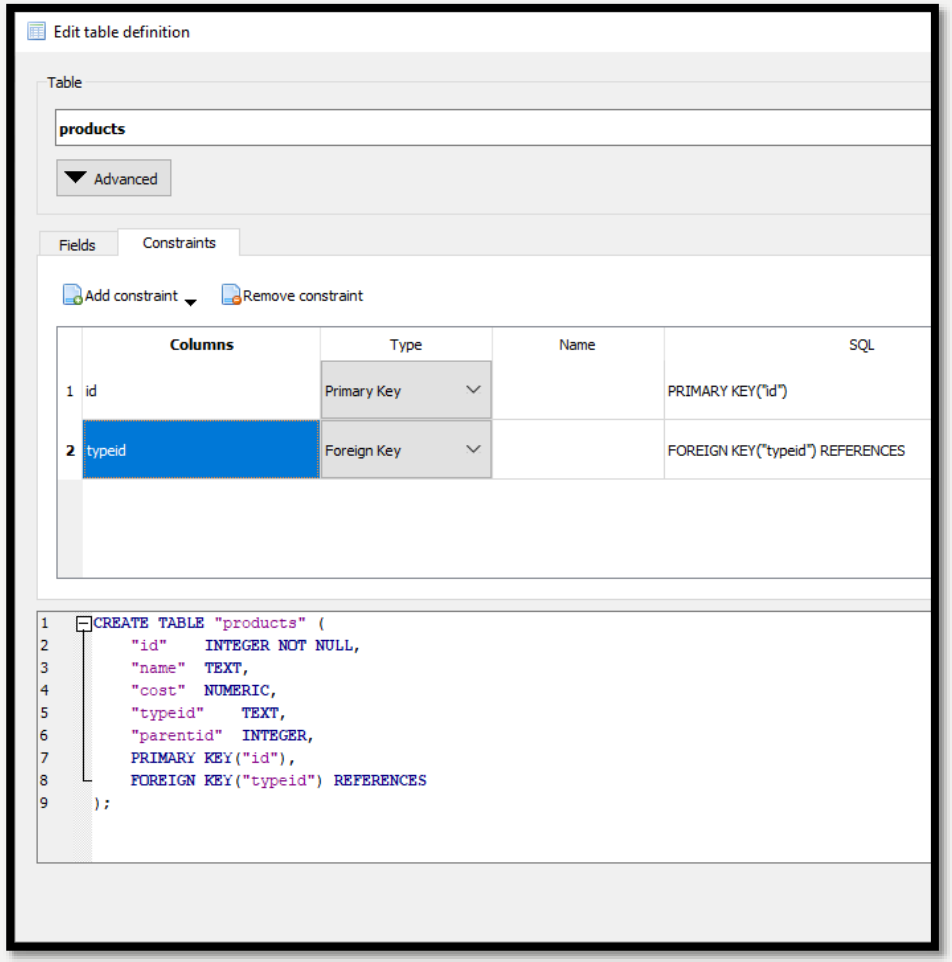
  add constraint  fk_typeid

  Foreign key product(typeid) references type(id);

- https://www.w3schools.com/sql/sql_foreignkey.asp

# ADDING FK TO ALL TABLES: SQLITE

SQLite does not allow addition of FKs, so you have to drop your tables and then recreate with FKs




```
CREATE TABLE "products" (
        "id"  INTEGER NOT NULL,
        "name"    TEXT,
        "cost"    NUMERIC,
        "typeid"  TEXT,
        "parentid"      INTEGER,
        PRIMARY KEY("id"),
        FOREIGN KEY("typeid") REFERENCES
type(id)
);
```

# LAB

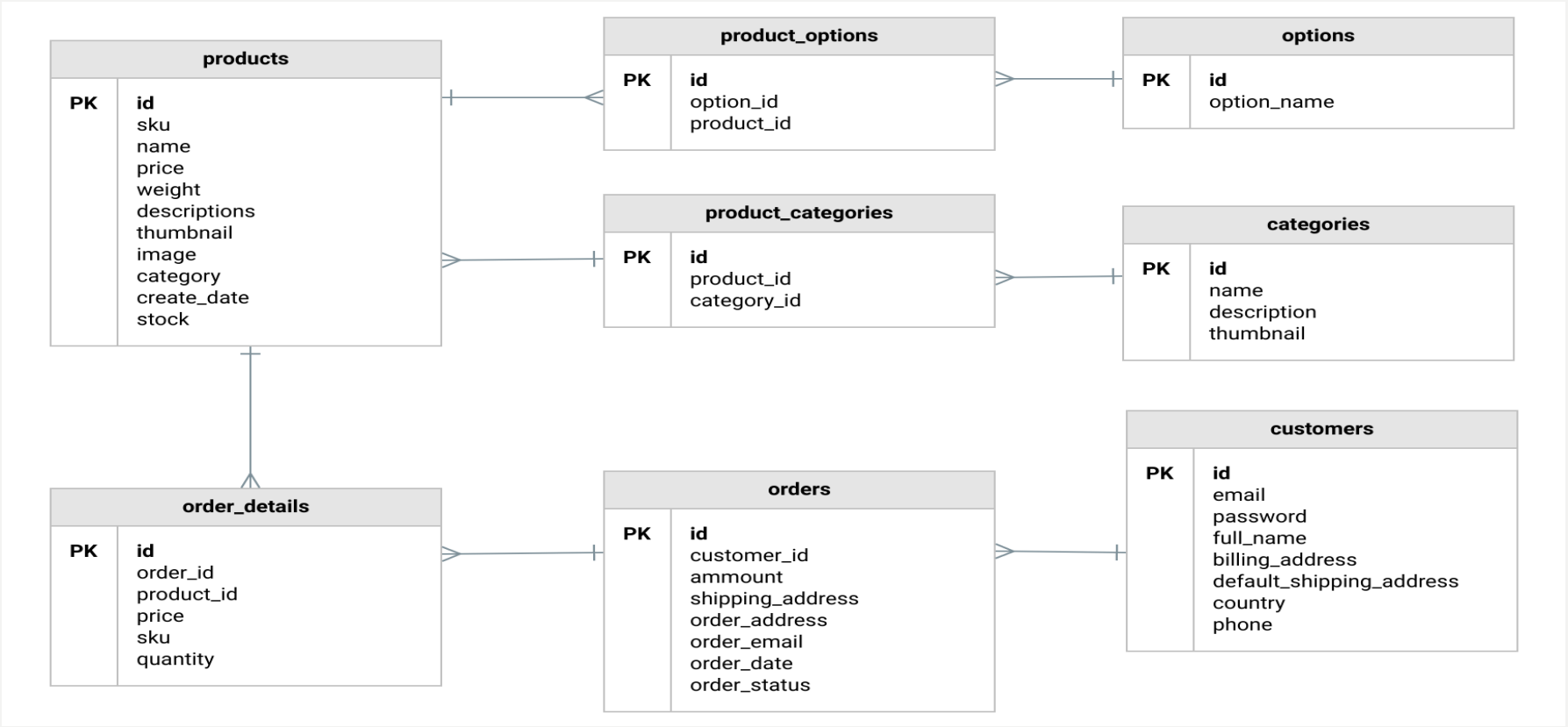
- Export data from the products table
  - Right-click on table, choose export to csv
- Manually Delete data from the products table
  - delete from [tablename]
- Manually Drop the products table
  - drop [tablename]
- Manually recreate products table

```
CREATE TABLE "products" (
        "id"            INTEGER NOT NULL,
        "name"          TEXT,
        "cost"          NUMERIC,
        "typeid"        TEXT,
        "parentid"      INTEGER,
        PRIMARY KEY("id"),
        FOREIGN KEY("typeid") REFERENCES type(id)
);
```

- Import your CSV into products table
  - Click on the table
  - Choose import table from file
  - Choose the file
  - Click columns in first row
  - Import

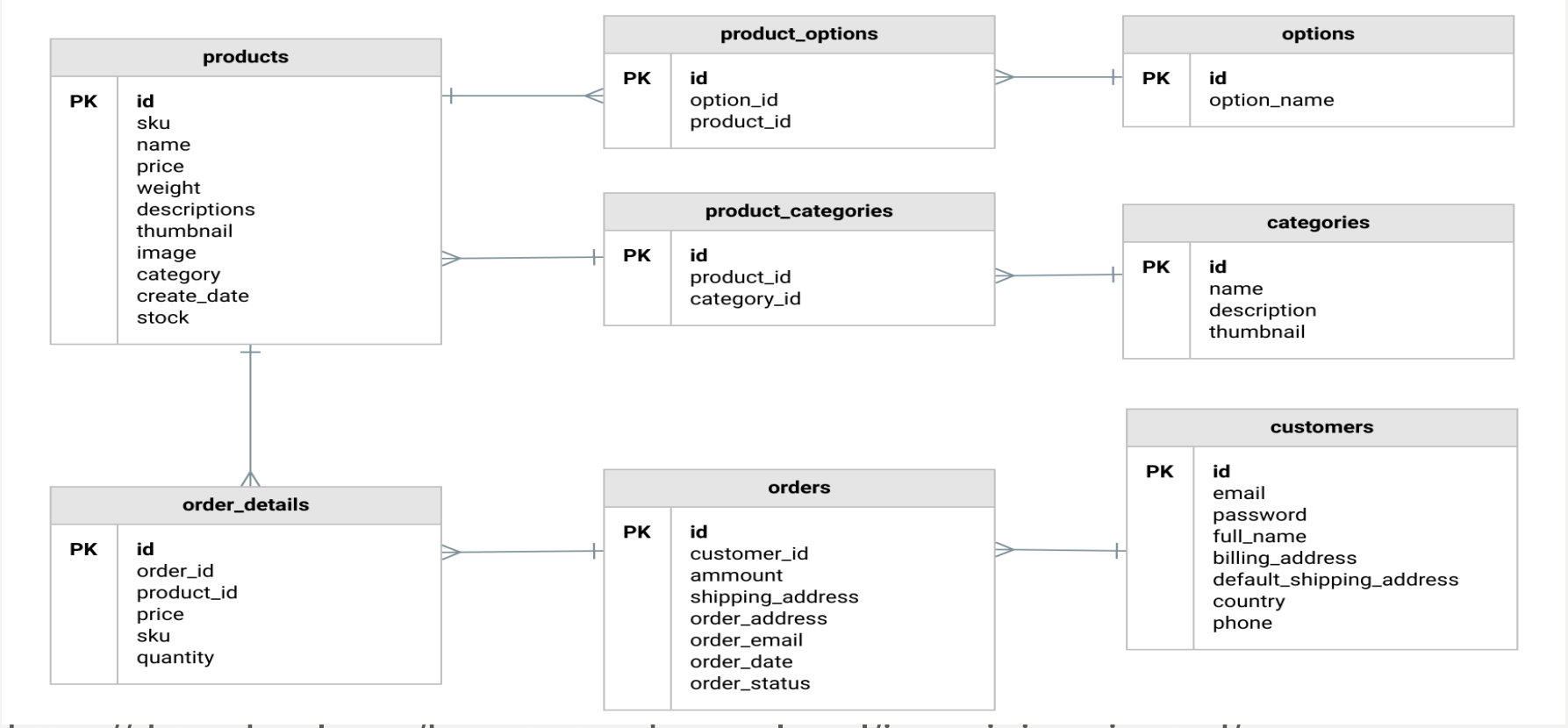
# DATABASE NORMALIZATION

Databases contain numerous discrete tables  to avoid data redundancy



# DB RELATIONSHIPS

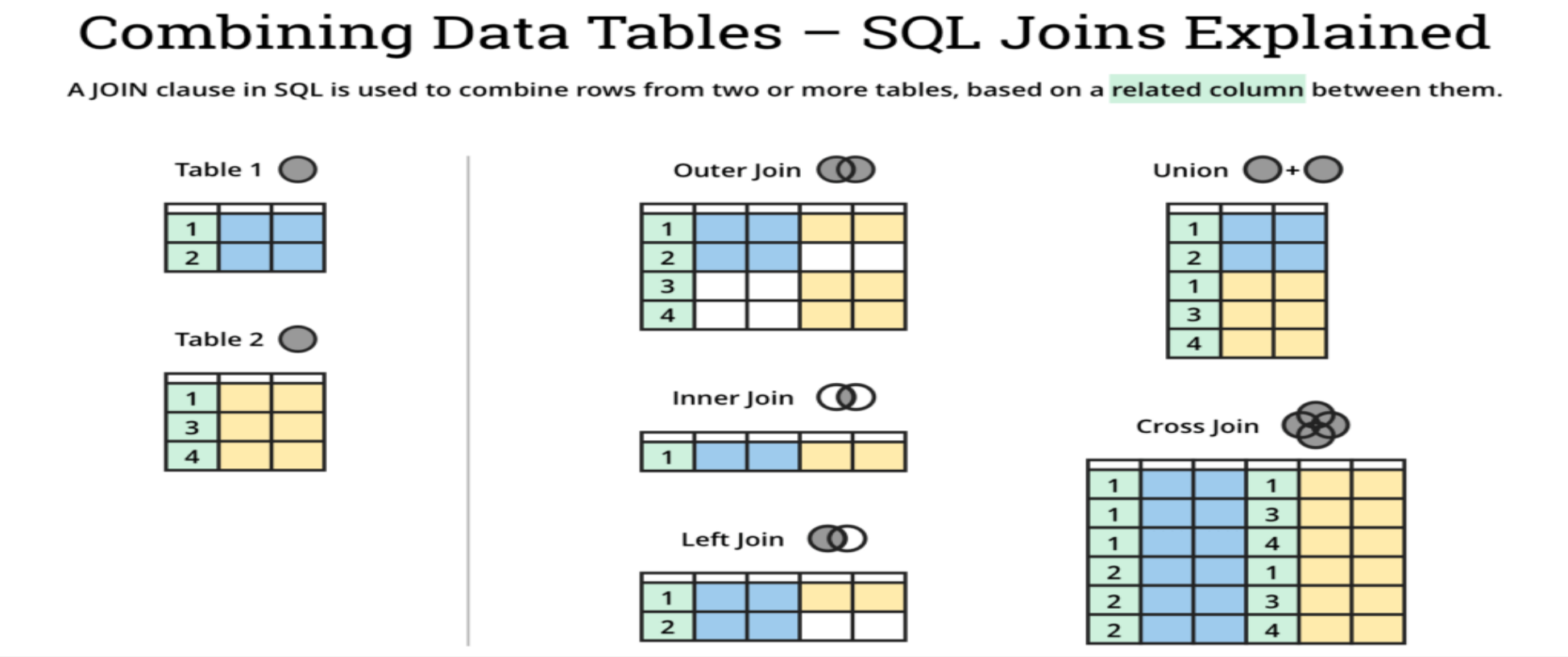
- Tables eventually connect back to each other through foreign keys




- https://dataschool.com/how-to-teach-people-sql/inner-join-animated/

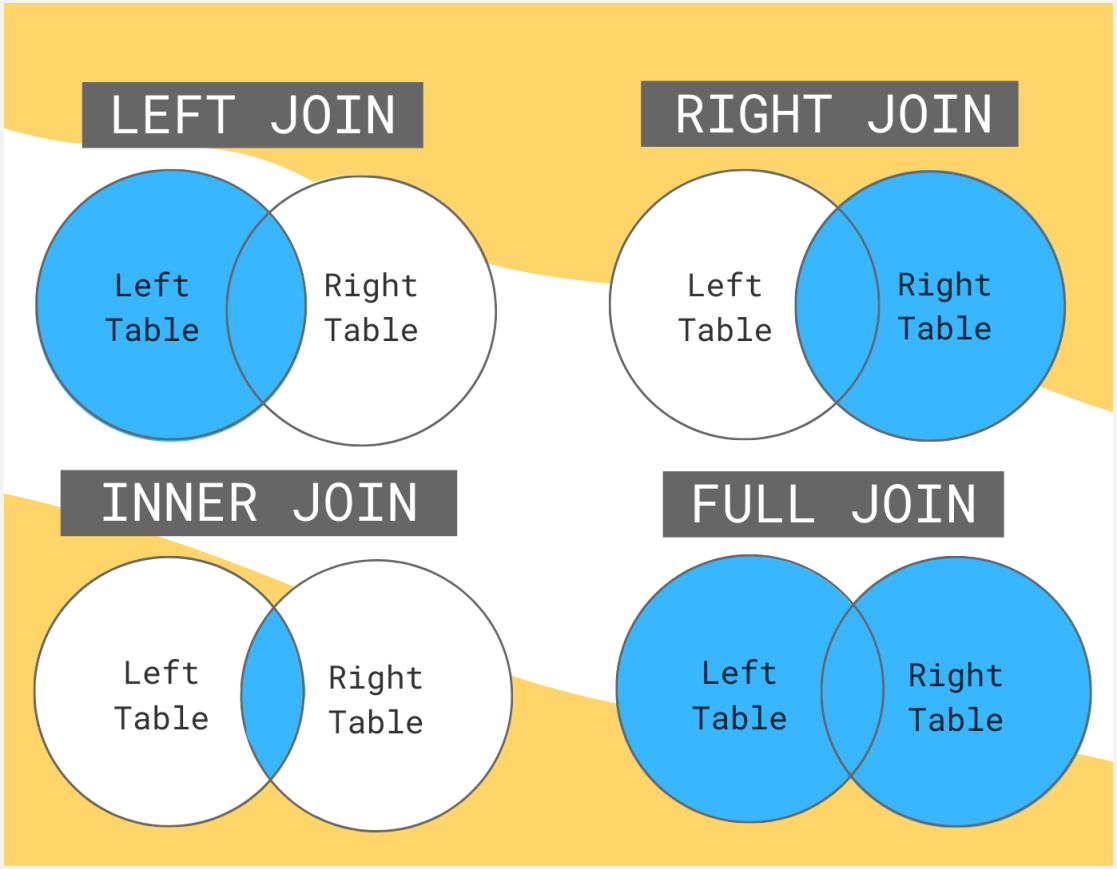
# DB RELATIONSHIPS

- We use SQL Queries to reflect the relationships between the different tables
- Joins are queries that connect tables together through ids
- And different joins bring out different data connections
- Joins are what are used to generate reports




- https://dataschool.com/how-to-teach-people-sql/inner-join-animated/

# JOIN TYPES




https://learnsql.com/blog/learn-and-practice-sql-joins/

```
SELECT product.name, type.name
FROM product
INNER JOIN type
ON
product.typeid = type. id
```

```
SELECT
product.name, type.name
FROM product
RIGHT JOIN type
ON
product.typeid = type. id
```

```
SELECT
product.name, type.name
FROM product
LEFT JOIN type
ON
product.typeid = type. id
```

# LAB: EXPAND OUR DB MODEL

- Create a users table
  - Fields: id, username, email, firstname, lastname
- Create an orders table
  - Fields: id, userid, productid, amount

# VIEWS

- A stored query
- Reusable
- Great for reporting
- Only retrieves data
- Great for complex queries
- Sometimes faster than a unique query*
- Naming standard: Either vw_ or view_ or v_ followed by viewname

**To Create a View**

- CREATE VIEW [viewname] as
  [Standard SQL i.e. select * from dogs]

# JOIN TYPES (INNER)

**OLD SCHOOL INNER**

```
SELECT product.name,
type.name
FROM
product, type
WHERE
product.typeid=type.id
```

**STANDARD INNER**

```
SELECT product.name,
type.name
FROM product
JOIN type
ON
product.typeid = type. id
```

**Standard Inner**

```
SELECT product.name,
type.name
FROM product
INNER JOIN type
ON
product.typeid = type. id
```

# LAB: CREATE VIEW

Create a view, vw_prod, that will join the product and type table (either using old school or standard join)
After creating a view, you can query it like a normal SQL statement
select * from vw_prod

### OLD SCHOOL JOIN ON PRODUCT & TYPE TABLE

```
SELECT table1.fieldname,
table2.fieldname
FROM
table1, table2
WHERE
table1.foreignkey =
table2.primarykey
```

### STANDARD JOIN

```
SELECT table1.fieldname,
table2.fieldname
FROM table1
JOIN table2
ON
table1.foreignkey =
table2.primarykey
```

# INDEXES

- Speed up database queries
- Create on any field that might be used frequently
- naming standard: typically idx_ [actual index name]

Create index [index_name] ON
[table name] [col names]

```
CREATE UNIQUE INDEX "idx_productid" ON "products" (
        "id"
)
CREATE INDEX "idx_products" ON "products" (
        "id",
        "name"
)
```

# LAB: CREATE INDEXES

- Create index on the products table
- Create index on the type table

```
CREATE UNIQUE INDEX "idx_productid" ON "products" (
        "id"
)
```

# MORE SQL FUNCTIONS\CLAUSES

## SUM

```
Select
product.name,
sum(product.amount) as sum,
count(product.amount) as totalproducts,
FROM
product left join type
group by type.name
order by type.name
```

https://www.sqlshack.com

## AVG

```
Select avg(orders.amount)
from orders
```

## DISTINCT

```
Select distinct names from product;
```

## LIMT

```
Select  names from product
LIMT 5;
```

## UPPER

```
CREATE VIEW vw_invoice as
 select \products.id,
upper(products.name),
 type.name,
orders.productamount,
users.username ,
 orders.productamount * products.cost as totalcost
FROM
 products
  join type on products.typeid=type.id
 join orders   on products.id = orders.productid
  join  users   on   orders.userid = users.userid
```

# STORED PROCEDURES

Stored Procedures

Stored Code

Can programmatically manipulate SQL results

Can use parameters

```
CREATE PROCEDURE [procname]   AS
[sql code]
Go;

CREATE PROCEDURE getProduct @id integer AS
AS
Select product.name from products where id = @id;
GO;
EXEC [procname];
```

# TRIGGERS

- Automates a database task
- Typically fires after a database statement (insert/update/delete)
- Can run:
  Before Update \ Insert \ Delete
  After Update  \ Insert \ Delete

- Can be:
  Used to auto update a PK field
  Used to back up a table

```
CREATE TRIGGER trigger_name
[BEFORE/AFTER]  [SQL COMMAND i.e. INSERT\UPDATE\DELETE]
ON
[tablename]
BEGIN
[SQL STATEMENTS]
END;
```

```
create trigger t_insert_products
  BEFORE INSERT ON products
  BEGIN
    select max(id) as oldid from products;
                  insert into products (id, name) values (oldid+1, name);
  END
```

```
CREATE TRIGGER t_insert_products
 After INSERT ON products
 BEGIN
   select "Great job";
 END
```

# AUTOINCREMENT

```
CREATE TABLE history_products (
historyid INTEGER PRIMARY KEY AUTOINCREMENT,
productid INTEGER,
productname TEXT,
datechanged TEXT);
```

https://www.sqlitetutorial.net/sqlite-autoincrement/

# DATE TIME FUNCTION

```
SELECT date('now');
```

# LAB: CREATE HISTORY_PRODUCT TABLE

- Create a history_product table (same fields as product table)
- However also add a history_product_id using autoincrement and add date_added field with date stamp
- Create a trigger, any insert into product will insert into product_history
- SQL (copy data from product into history)
- Now insert new row into product table
- check the history table  see new row there
- now update the product table

https://www.sqlitetutorial.net/sqlite-trigger/

```
CREATE TABLE history_products (
historyid INTEGER PRIMARY KEY AUTOINCREMENT,
productid INTEGER,
productname TEXT,
datechanged TEXT);
```

```
CREATE TRIGGER t_insert_products
 After INSERT ON products
 BEGIN
   insert into history_products (productid, productname,
datechanged) values (NEW.id, NEW.name, date('now'));
  END
```

# FUTURE

- Parent IDs
- SQL Transactions: https://www.sqlitetutorial.net/sqlite-transaction/