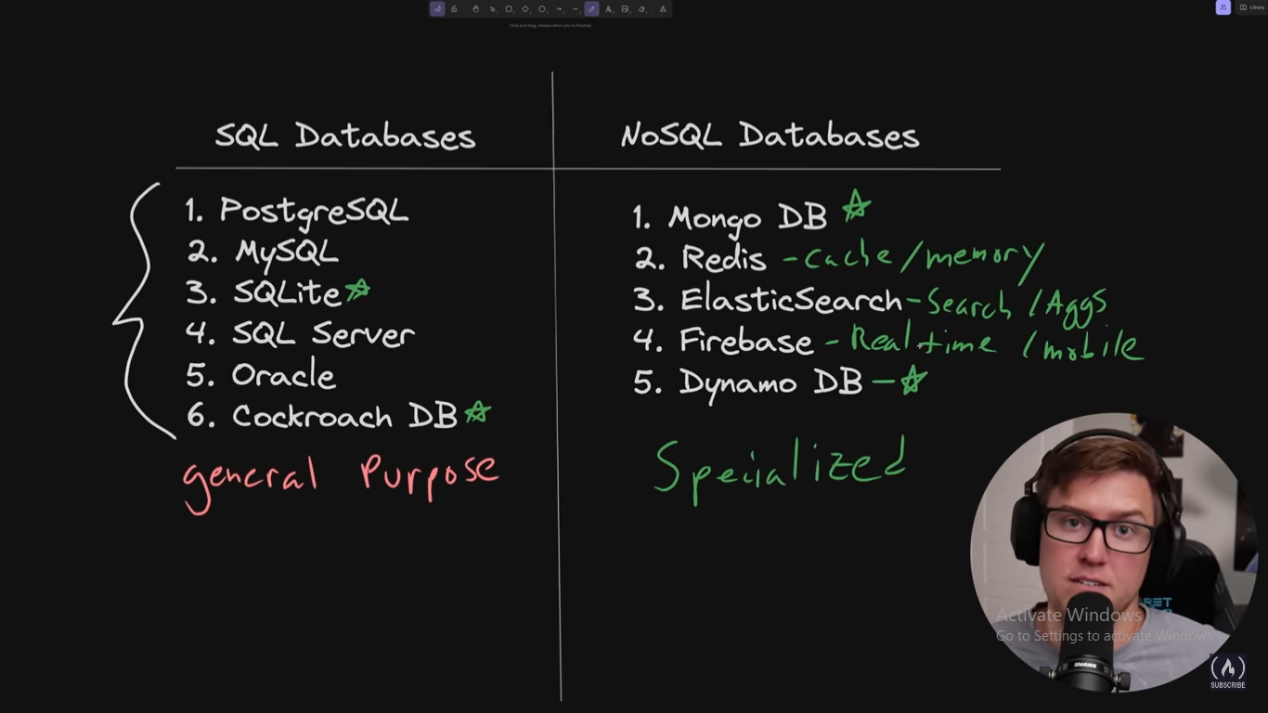
**SQL**

1. Difference between SQL and NoSQL DB is found in the use of it.
2. 
3. SQLite does not enforce type checking unlike PostGreSQL.

CREATE TABLE users (id INTEGER, name TEXT, age INTEGER);

INSERT into users (id, name, age) values (1, 'John Doe', 21);

INSERT into users (id, name, age) values (2, '1', 33);

SELECT \* FROM users;

Inserting ‘1’, an integer, also passes as a value.

1. **Create table**

create table transactions(

id INTEGER,

recipient\_id INTEGER,

sender\_id INTEGER,

note text,

amount INTEGER

);

1. **Alter table**

ALTER TABLE employees

RENAME TO contractors;

ALTER TABLE contractors

RENAME COLUMN salary TO invoice;

ALTER TABLE contractors

ADD COLUMN job\_title TEXT;

ALTER TABLE contractors

DROP COLUMN is\_manager;

1. **SQLite datatypes**

NULL - Null value.

INTEGER - A signed integer stored in 0,1,2,3,4,6, or 8 bytes.

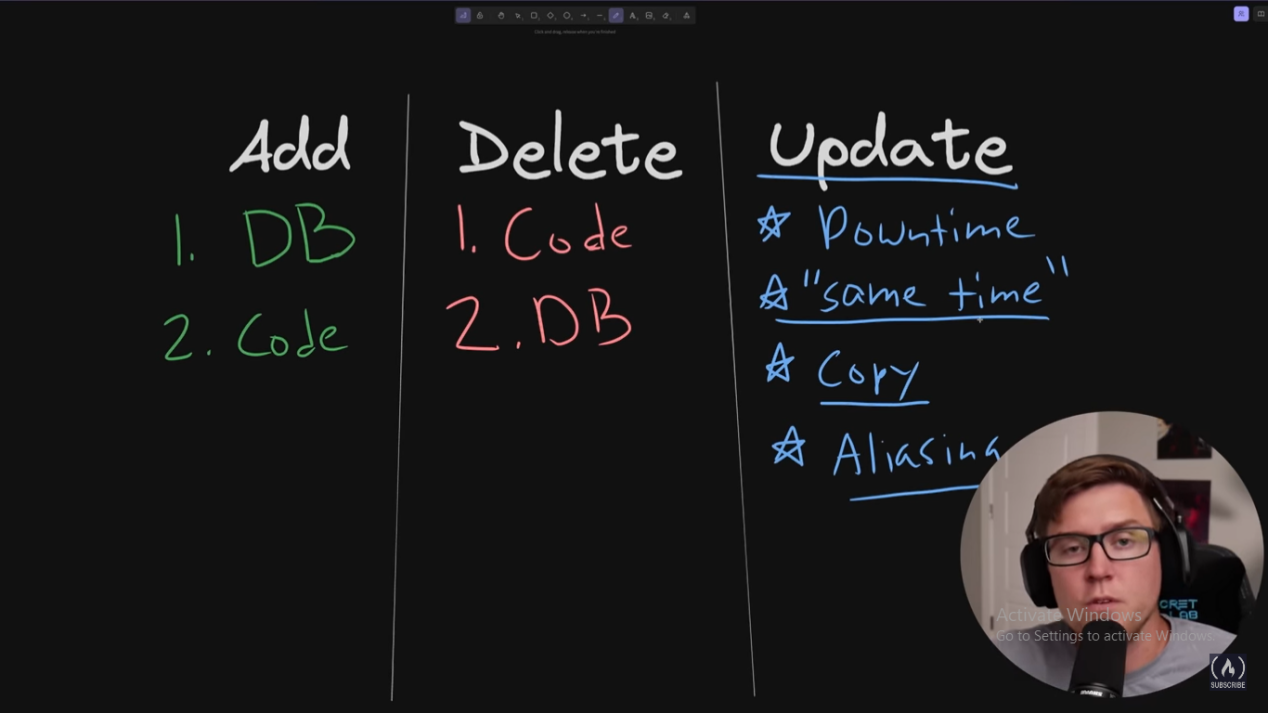
REAL - Floating point value stored as an 64-bit IEEE floating point number.

TEXT - Text string stored using database encoding such as UTF-8

BLOB - Short for Binary large object and typically used for images, audio or other multimedia.

BOOLEAN- 1 for true, 0 for false

1. **DB Operation**



1. **Constraints**

create table transactions(

id INTEGER PRIMARY KEY,

recipient\_id INTEGER NOT NULL,

sender\_id INTEGER NOT NULL,

note text UNIQUE,

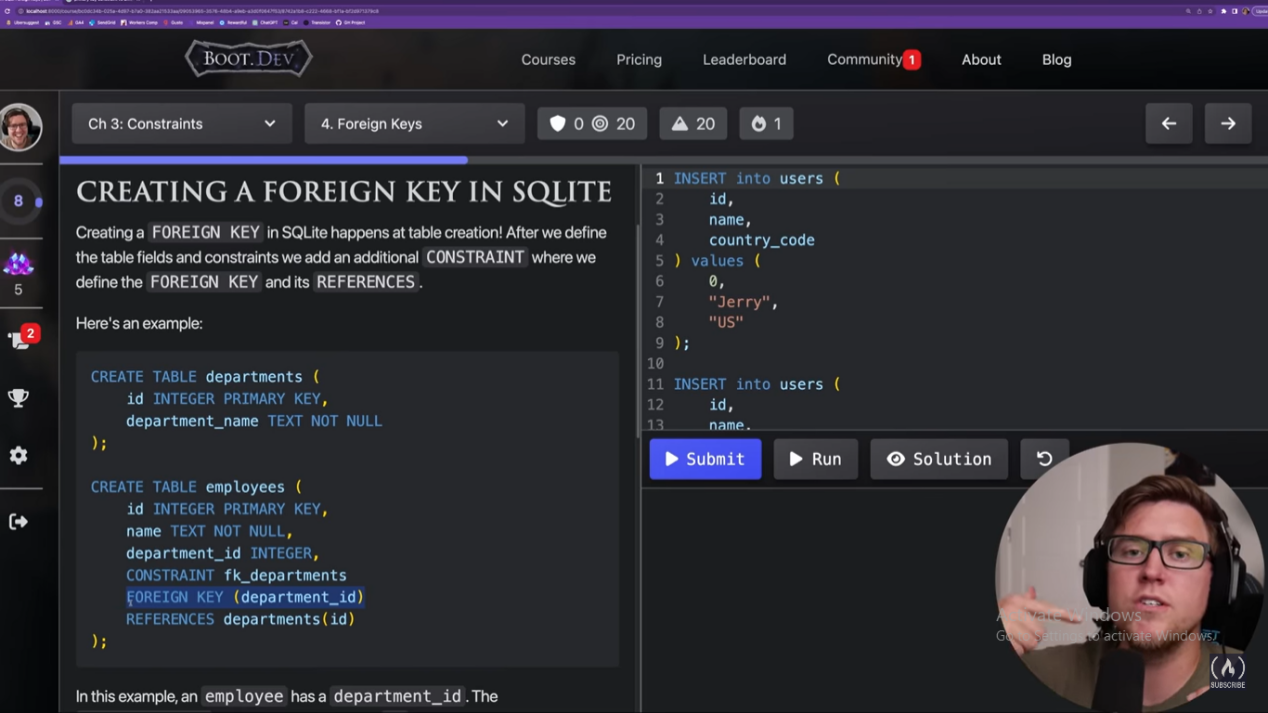
amount INTEGER

);

UNIQUE can be multiple in a table and can also be NULL, unlike PRIMARY KEY.

We can have primary key like ‘id’ starting with 0;

1. **Foreign key**



1. Balance, amount are stored in INTEGER and not in REAL because floating numbers have rounding off issues. So, instead of 1 rupees 45 paisa, we store 145 paisa in INTEGER.
2. Non-Relational DB store nested data. They store records within other records like a giant JSON Blob.
3. **INSERT** INTO users

(id, name, age, country\_code, username, password, is\_admin)

VALUES

(1, “David”, 23, “US”, “David Malan”, “dummy Pass”, false);

1. SELECT **count**(\*) FROM users;
2. SELECT emp\_id, emp\_name FROM users **WHERE** salary>9000;
3. SELECT emp\_id, emp\_name FROM users **WHERE** emp\_name **IS NULL**; // **IS NOT NULL**
4. **DELETE** from users where salary<20000;
5. DB backups using daily snapshots or hourly snapshots. Relatively cheap to operate. Second method is ‘Append only log’. More granular and robust but little expensive to operate.
6. Soft delete is marking the row as deleted instead of actually deleting the data.
7. **UPDATE** users **SET** emp\_name=”sachit”, emp\_salary= 20000 **WHERE** emp\_id=3;
8. **ORM** is used to map database records to in-memory databases. Less control over your DB, not easy to debug at a low-level.
9. SELECT emp\_id **AS** id, emp\_name **AS** name from employees;
10. **SQL functions: IIF is more like a ternary operator.**

SELECT \* ,

**IIF**( was\_successful, “No audit needed”, “Perform audit”) AS audit

FROM employees;

1. SELECT emp\_name, emp\_salary FROM employees **WHERE** salary BETWEEN 5000 and 10000;

SELECT product\_name, qty FROM products **WHERE** quantity **NOT BETWEEN** 20 and 100;

1. SELECT **DISTINCT** company\_name FROM employees;
2. SELECT product\_name, qty FROM products **WHERE** shipment\_status=’pending’ **AND** qty BETWEEN 0 and 10; //**OR**
3. SELECT count(\*) FROM users WHERE (country\_code =”US” **OR** county\_code=”CAN”) **AND** age>18;
4. SELECT product\_name, qty FROM products WHERE shipment\_status **IN** (‘shipped’, ‘pending’, ‘out of stock’);
5. %--> 0 or more matches whereas \_ ---> exactly one match

SELECT \* FROM users WHERE emp\_name **LIKE** ‘Al%’;

**LIKE** ‘\_oot’;

SELECT \* FROM users WHERE name **LIKE** ‘Al\_\_\_’; //name having exactly 5 chars starting with Al.

1. SELECT \* FROM employees **WHERE** emp\_name **LIKE** ‘Sa%’ **LIMIT** 50;
2. SELECT \* FROM employees **ORDER BY** salary; // salary **desc**;
3. SELECT \* FROM transactions **WHERE** amount **BETWEEN** 10 and 80

**ORDER BY** amount **DESC LIMIT** 5;

1. **Aggregations**

SELECT **count(\*)** FROM transactions WHERE user\_id=6 AND was\_successful = true;

SELECT **sum(salary)** FROM employees;

SELECT **max(price)** FROM products;

SELECT product\_name, **min(price)** FROM products;

SELECT album\_id, count(song\_id) FROM songs **GROUP BY** album\_id;

SELECT song\_name, **avg**(song\_length) FROM songs;

SELECT album\_id, count(id) AS count FROM songs GROUP BY album\_id **HAVING** count>5;

SELECT sender\_id, sum(amount) AS balance FROM transactions

WHERE note LIKE ‘%lunch%’

GROUP BY sender\_id

HAVING balance>20

ORDER BY balance ASC;

**round( value, precision)**

Select song\_name, **round**(avg(song\_length), 1) FROM songs;

1. **Sub-Queries**

SELECT id, song\_name, artist\_id FROM songs

WHERE artist\_id IN (

**SELECT id**

**FROM artists**

**WHERE artist\_name LIKE ‘Rick%’**);

SELECT \* FROM transactions

WHERE user\_id= (SELECT id FROM users WHERE name= “David”);

1. **Normalization**
2. **Data Integrity**- Instead of saving age, we should save d.o.b. Since, age will become irrelevant next year but we can always calculate age if we have d.o.b.
3. **Joins**

It allows us to query multiple tables at the same time.

Inner join- Join by default is inner join. Table A intersection Table B.

SELECT \* FROM employees

**INNER JOIN** departments

**ON** employees.dept\_id= departments.id;

SELECT users.name, users.age, countries.name AS country\_name

**FROM users INNER JOIN countries**

**ON** users.country\_code= countries.country\_code

ORDER BY country\_name ASC;

Left join- Everything from Table A + (Table A intersection Table B)

SELECT users.name, sum(transactions.amount) as sum,

count(transactions.id) as count

FROM users **LEFT JOIN** transactions

ON user.id= transactions.user\_id

GROUP BY users.id

ORDER BY sum DESC;

SELECT users.id, **users**.name, users.age, users.username, **countries**.name AS country\_name,

sum (**transactions**.amt) AS balance

FROM users

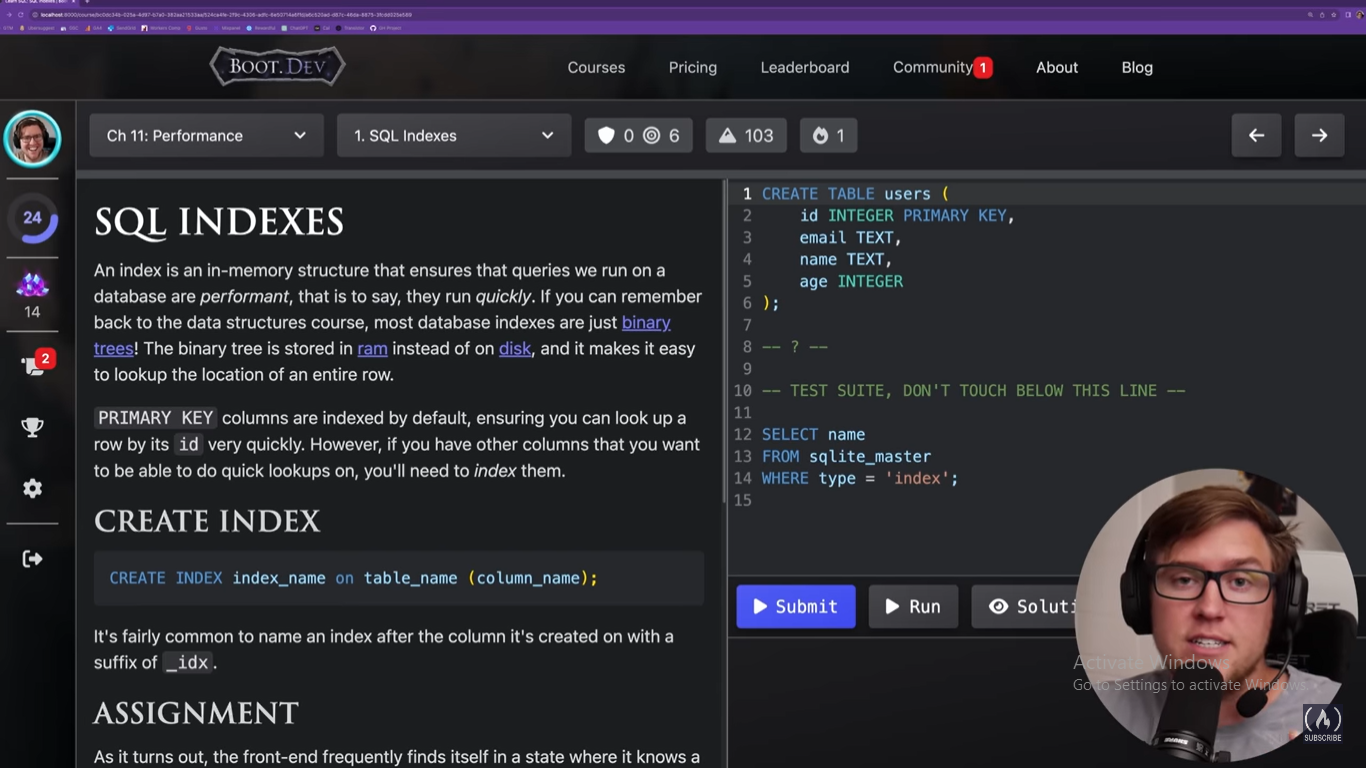
LEFT JOIN countries ON users.country\_code= countries.country\_code

LEFT JOIN transactions ON transactions.user\_id= users.id

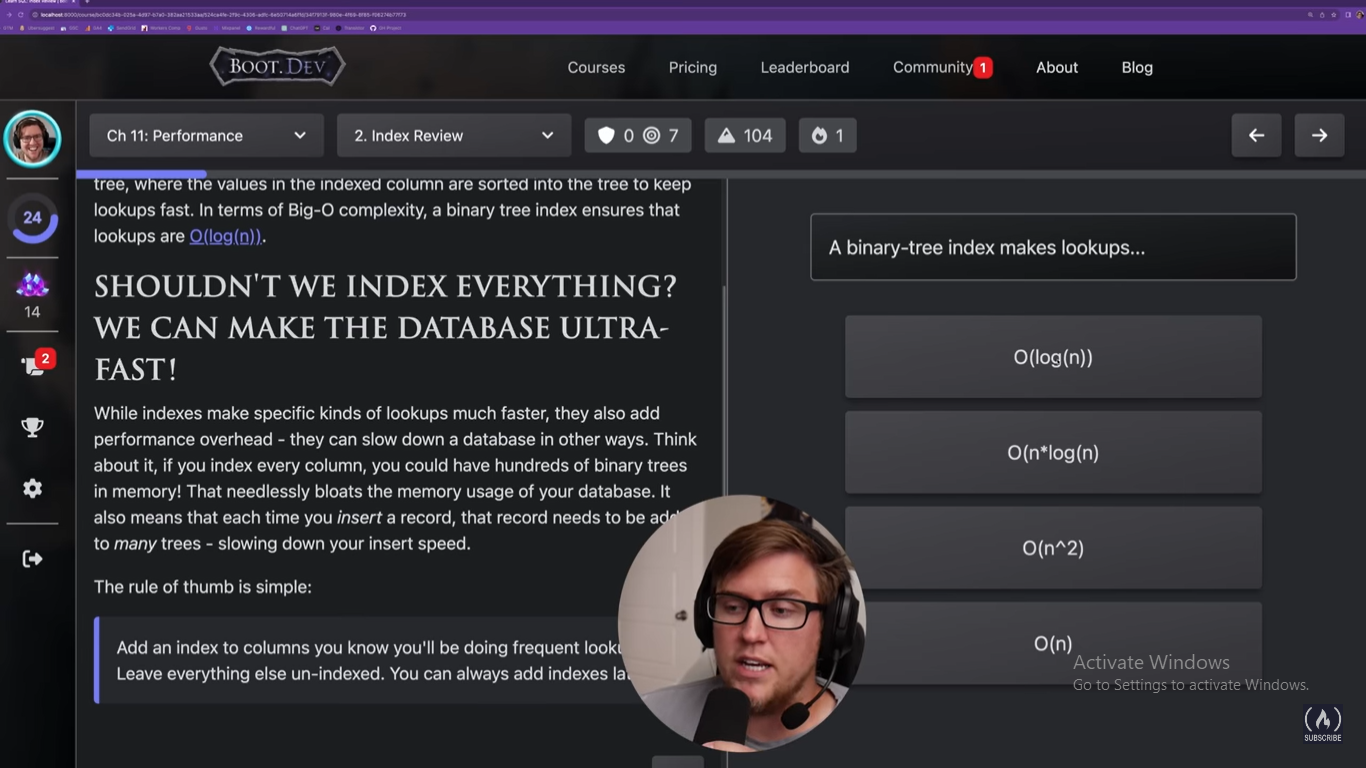
WHERE user.id=6

GROUP BY users.id;

1. **Performance**-



**CREATE INDEX** email\_idx on users (email);



Indexes should be made for the columns which are frequently looked up (the one in which we use most where clauses).

Indexes slow down write speed because we need to update the in-memory binary tree.