MySQL

The ultimate MySQL Bootcamp: Go from SQL Beginner to Expert

Try-SQL Editor – W3Schools.

→ SELECT * FROM customers;

All columns from customer table.

→ SELECT * FROM orders;

→ SELECT * FROM products
ORDER BY price DESC;

→ SELECT customerName, COUNT(*) AS 'number of orders'
FROM customers
INNER JOIN orders
ON orders.customerID = customers.customerID
GROUP BY customers.customerID.

only common

Structuring - Intentation

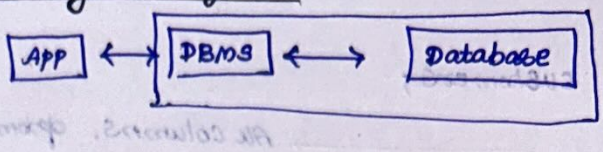
```
SELECT
    username,
    photos.id,
    photos.image_url,
    COUNT(*) AS total
FROM photos
INNER JOIN likes
    ON likes.photo_id = photos.id
INNER JOIN users
    ON photos.user_id = users.id
GROUP BY photos.id
ORDER BY total DESC
LIMIT 1;
```

```
SELECT first_name,
    last_name,
    Count(rating)  AS count,
    Ifnull(min(rating),0)  AS MIN
    Ifnull(MAX(rating,0)  AS MAX
    Round(Ifnull(Avg(rating,0),2)
                AS AVG,
CASE
    WHEN Count(rating) >=10
        THEN 'power USER'
    WHEN Count(rating) >0 THEN
        'ACTIVE'
    ELSE 'INACTIVE'
end                 AS STATUS
FROM reviewers
LEFT JOIN reviews
    ON reviewers.id=
        reviews.reviewerid
GROUP BY reviewers.id;
```

Database: * Collection of data (medical records ...)

* A phone book
* A method for accessing & manipulating the data.

Database (vs) Database Management System:



Postgresql
MySql
oracle database
Sqlite.

'A structured set of computerized data with an accessible interface.

MySql (vs) Sql

* Structured query language. - language to talk to our databases.
* SELECT * FROM Users
  WHERE Age >= 18.
* work with mysql - writing sql
  - Std. Sql (distribution may be)

'slight diffences' But Sql Std should work!

---

* DBMS: unique by operations they offer, not the language!

Install mysql:

* Cloud9 - online interactive dev environment

Why Cloud9:

Good, help, Uniform.

Create databases

* Database server → Databases
  - eg: Soap shop database
    Dog walker database.
* Show databases; → list current existing databases

```
CREATE DATABASE <name>;
```

eg: CREATE DATABASE DogAPP;

'My APP' → Not recommended

'Use Snake case'

DROP DATABASE <name>;

Create and delete databases:

```
CREATE DATABASE hello_world_dB
DROP DATABASE hello_world_dB
```

Use databases:

There may be multiple databases - use a specific one.

```
USE hello_world_dB
USE sql_hr
```
→ Database changed.

SELECT ~~data base dB~~(); → SELECT database();

↓

Shows current database!

Say: we use a database, deleted it - what we are using now? → NULL.

```
USE hello_world_dB;
SELECT database(); → hello_world_dB
DROP hello_world_dB;
SELECT database(); → NULL (∵ deleted).
```

Now switch

```
USE C9;
SELECT database(); → C9.
```

Tables

* Term heard eg SQL.
* Database: bunch of tables. (Related databases)
* Tables: Holds data. (Related data in a structured format).

Cats table

Columns (headers)

Name	Breed	Age
Blue	Scottish Fold	1
Rocket	Persian	3
Monty	Tabby	10
Sam	Munchkin	5

Rows. (Actual data).

Datatypes

Text | Text | Number

Numeric types:	String types	Date types
INT	Char	Date
Small int	Varchar	Datetime
tiny int	binary	Timestamp
medium int	Varbinary	Time
Big int	blob	year
decimal	tinyblob	
numeric	mediumblob	
float	longblob	
double	text	
bit	tinytext	
	mediumtext	
	Longtext	
	Enum	

INT — whole number [max value : 4294967295]

0, 12, -9999, 49, 3145677

Varchar — Text / String (variable length) — B/w 1 to 255 characters

'Coffee', 'aAbbb AKIJSd', '-999'

'The quick brown fox jumps over the lazy brown dog'

Varchar(100) → max length [max: 255]

Tweets table

* username: 15 chars → Varchar(15)
* Tweet content (140 chars) → Varchar(140)
* Number of favourites. (likes) → INT.

username	TweetContent	favourites
'cool_guy'	'my tweet'	1
'guitar'	'music :)'	10
'user123'	'I'm nobody'	0

Implement table

```
CREATE TABLE tablename
(
    column_name datatype,
    column_name datatype
);
```

```
CREATE TABLE CATS
(
    name VARCHAR(100),
    age INT
);
```

```
CREATE DATABASE cat_app;
CREATE TABLE cats (
    name VARCHAR(100);
    age INT
);
```

Is it worked?

SHOW TABLES; ⟶ cats.

SHOW COLUMNS FROM <tablename>; ⟶ shows fields, types, Key.....

DESC <tablename>; ⟶ describe. (Stick with it)

Deleting tables

DROP TABLE <tablename>;

⟶ DROP TABLE cats;

⟶ SHOW TABLES;

⟶ DESC cats; ⟶ cats no longer exist (error)

Activity

create a pastries table ⟶ Name: 50 max char.

* 2 columns — name, quantity
* Inspect table
* Delete table & database.

⟶ CREATE DATABASE tweet_dB;

⟶ SHOW DATABASES;

⟶ USE tweet_dB;

⟶ CREATE TABLE tweets (

```
    username VARCHAR(15),
    content VARCHAR(140),
    favourites INT
);
```

⟶ Table created (But no data).

⟶ SHOW TABLES; ⟶ tweet

⟶ SHOW COLUMNS FROM tweet;

INSERT — Adding data

```
INSERT INTO cats (name, age)
VALUES ('Jetson', 7);
```

⟶ add. (order matters)

name, age

↓ 'Jetson' ↓ 7 !

```
INSERT INTO cats (age, name)
VALUES (12, 'Victoria');
```
] order matters.

How to Know it worked?

```
SELECT * FROM cats; ⟶ Show all Columns.
```

Add values to tweet

```
INSERT INTO tweet (username, content, favourites)
VALUES ('joe', 'hello world', 0),
       ('James', 'I'm James', 1),
       ('Baby', 'Nice day', 2),
       ('bob', 'weather', 11);
```
] multiple values. [Just Comma]

```
SELECT * FROM tweet;
```

warnings

People:

first_name	last_name	age
Tina	Belcher	13
Bob	Belcher	42
Linda	Belcher	45
Philip	Frond	38
Calvin	Fischoeder	70

```
DROP TABLE people;
Show tables; ⟶ NULL.
```

warnings

```
VALUES ('This cat ..... ', 10)
```
↳ gigantic name! (warning)

→ SHOW WARNINGS; ⟶ Data truncated for column 'name' at row 1.
(not stored: truncated)!

→ mixed data type

```
INSERT INTO tweet (username, content, favourites)
VALUES ('Hi', 'Hello', 'How');
```
↳ Supposed to be INT

> SHOW WARNINGS; ⟶ Incorrect Integer value

> SELECT * FROM Cats; ⟶ Lima 0
↳ default value placed!

SHOW WARNINGS; ⟶ only shows for Recent Command!

DESC <tablename>

→ ok to be NULL (default)

Field	Type	NULL	Key	Default	Extra
name	varchar(5)	YES		NULL	
age	int(11)	YES		NULL	

NULL: Not known (not means zero)

```
INSERT INTO Cats (name)
VALUES ('Alabama');
```

SELECT * FROM Cats;

↳ Alabama NULL
↳ Age!

INSERT INTO Cats ()
VALUES ();

⟶ NULL NULL

Prevent NULL

```
CREATE TABLE Cats2
(
  name VARCHAR(100) NOT NULL;
  age  INT          NOT NULL
);
```

DESC Cats2;

NULL	...
No	
No	

```
INSERT INTO Cats2 (name)
VALUES ('Texas');
```

↓
NULL not allowed
↓
warning
↓
age doesn't have a default value
Can't be NULL.
(default 0)

SELECT * FROM Cats

Texas	0

But with warnings!

```
INSERT INTO Cats2 (name)
VALUES ('Texas')
```

default
'warning' : age 0

```
INSERT INTO Cats2 (age)
VALUES (12)
```

WARNINGS
name: (empty space)

name	Age
	13

empty string (Not NULL) → ""

DEFAULT values

```
CREATE TABLE cats3
(
    name VARCHAR(100) DEFAULT 'unnamed',
    age  INT          DEFAULT 99
);
```

DESC cats3; →

DEFAULT
unnamed
99

→ INSERT INTO cats3 (age)
VALUES (13);

→

unnamed	13

```
CREATE TABLE cats4
(
    name VARCHAR(100) NOT NULL DEFAULT 'unnamed',
    age  INT          NOT NULL DEFAULT 99
);
```

→ Is this Redundant?

---

* DEFAULT → Telling Not NULLS?

Note: I can manually set something to NULL!
To prevent that — NOT NULL!

```
INSERT INTO cats3 (name, age)
VALUES ('Montana', NULL);
```

→ ∴ we are giving values — so default not used (NULL allowed)

Montana	NULL

Avoid this — No NULL

```
INSERT INTO cats4 (name, age)
VALUES ('cali', NULL); ⟶ error — age can't be NULL!
```

---

one more thing!

Key?

In previous cases: No key! : multiple similar values allowed

Do 6 times.

```
INSERT INTO Cats (name, age)
VALUES ('Helena', 6);
```

* do six times – 6 similar values!
* why problem? : Retrieve: only when uniquely identifiable!
* Not able to keep track uniquely → Key! (comes).

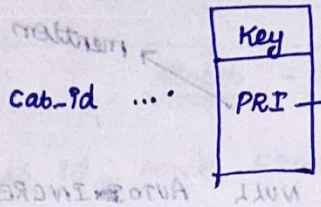
No two same usernames

Assign ID – unique (ID won't repeat)

* Primary key – Identify something unique.

```
CREATE TABLE unique_cats (cat_id  INT NOT NULL,
                          name    VARCHAR(100),
                          age     INT,
                          PRIMARY KEY (cat_id));
```

```
DESC unique_cats;
```



Cat_id ... → Primary key!

```
INSERT INTO unique_cats (id, name, age)
VALUES (1, "fred", 23);
```

→ duplicate (error!)

```
"
VALUES (1, "Lois", 18);
```

Increment – generate

```
CREATE TABLE unique_cats2 (cat_id  INT NOT NULL AUTO_INCREMENT,
                           name    VARCHAR(100),
                           age     INT,
                           PRIMARY KEY (cat_id));
```

* each time by default increment by 1.

→ `DESC unique_cats2;`

```
INSERT INTO unique_cats2 (name, age)
VALUES ('Skippy', 4),
       ('Jiff', 3),
       ('Jiff', 3),
       ('Skippy', 4);
```

1	Skippy	4
2	Jiff	3
3	Jiff	3
4	Skippy	4

My turn

employee table:

Id - number (automatically increments), mandatory, primary key

last_name - text, mandatory

first_name - "

middle name - ", not mandatory

age - number, must

Current_status - text, must, default: 'employed'

2ways:

```
CREATE TABLE Employees (
    ...
    PRIMARY KEY (Id)
);
```

→ Seperately mention

(or)

```
CREATE TABLE Employees (
    Id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ...
);
```

→ mention directly.

CRUD - Create, Read, update & delete

```
INSERT INTO Cats (name, age)
VALUES ('Taco', 4);
```

→ create

Read

```
DROP TABLE cats;
```

→ deleted the table.

```
CREATE TABLE Cats
(
    cat_id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(100),
    breed VARCHAR(100),
    age INT,
    PRIMARY KEY (cat_id)
);
```

Ringo Tabby 4

Cindy Maine Coon 10

→ See Sql.

⋮

(data)

Retrieve and search data

→ SELECT * FROM cats;
  ↳ give me all columns.

→ SELECT name FROM cats;

→ SELECT name, age FROM cats; → order matters

name	age

→ SELECT age, breed, name, cat_id FROM cats;

WHERE

→ SELECT * FROM cats WHERE age = 4; → 2 records with age = 4

→ SELECT * FROM cats WHERE name = 'Egg'; → 1 record.

case insensitive: EGG, Egg are all same! (eggs)

Aliases – Easy to read results.

→ SELECT cat_id AS id, name FROM cats;

→ SELECT name AS cat_name, breed as kitty_breed FROM cats;

- Aliases just gives name as aliases (fields) – return
- Actual fields won't be changed!

UPdate

UPDATE cats SET breed = 'Shorthair'
WHERE breed = 'Tabby';
] update where even breed is Tabby!

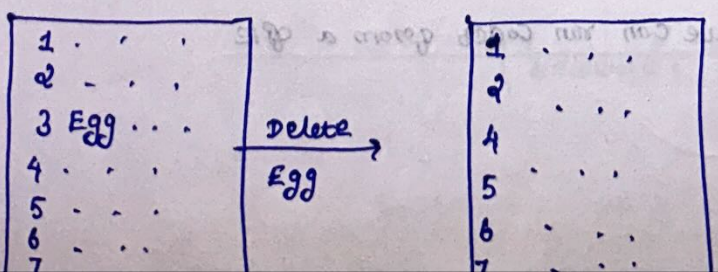
UPDATE cats SET age = 14
WHERE name = 'Misty';

No undo! ← Rule of thumb – Try SELECTING before updating!

DELETE

DELETE FROM cats WHERE name = 'Egg'; → delete row having egg!

↳ Cat_id → stay same



why? unique_ID → multiple table → change in all instances shared

! 'no unique_id' – delete → others won't be updated.

Different (* DELETE FROM cats; → delete all rows! → Null table!
* DROP TABLE cats; → delete entire table

* DELETE all 4 years cats

shirts_db

shirt_id	article	color	shirt_size	last worn
1	tshirt	white	S	10
2	tshirt	green	S	200
3	Polo	black	M	10
4	tank	blue	S	50
5	tshirt	pink	S	0
6	Polo	red	M	5
7	tank	white	S	200
8	tank	blue	M	15

↓ primary key (can't be NULL)

→ purple – poloshirt – M – last worn: 50 days ago!
→ SELECT all shirts → Article & color.
→ shirt_id (except).
→ update all polo shirts – size L
→ update last worn – 15d → 0 days
→ All white shirts → size XS → Color 'off white'
→ All old shirts (200 days) – Delete.
→ Delete tank tops.
→ Delete all shirts.
→ drop shirts table!

multiple: UPDATE shirts SET _____, _____, _____
WHERE _____

String

* Concatenate, substring, replace, upper case, lower case.

we can run codes from a file

→ book_data.sql

Source myfirst_sql.sql

Source testing/insert.sql → path important

CONCAT - Combine data

author_fname, author_lname

→ SELECT author_fname, author_lname FROM books;

Full names

* CONCAT (x, y, z)
* CONCAT (column1, column2) → No space!

we want comma (or) space.

→ CONCAT (author_fname, ' ', author_lname)

→ SELECT CONCAT ('Hello', 'world');

HelloWorld

→ SELECT CONCAT (author_fname, ' ', author_lname) AS name

FROM books;

→ SELECT author_fname AS first, author_lname AS last

CONCAT (first, ' ', last) AS full

FROM books;

CONCAT_WS - Concat with seperator

SELECT

CONCAT_WS (' - ', title, author_fname, author_lname)

FROM books;

SUBSTRING

→ SELECT SUBSTRING ('Hello world', 1, 4);

String, start → end) Indices

→ 1st index

('Helloworld', 7);

→ start until end.

→ SELECT SUBSTRING('Hello World', 3, 8);

→ SELECT SUBSTRING('Hello World', 3);

llo world

→ SELECT SUBSTRING('Hello World', -3);

↳ starts from end

rld

→ use '' '' to use '

· SUBSTR()

Some titles unfinished → add ... at the end.

```
SELECT
    CONCAT(
        SUBSTR(title, 1, 10),
        '...'
    ) AS 'Short title'
FROM books;
```

Replace

→ REPLACE('Hello World', 'Hell', '%$#@');

↳ 'case sensitive'

→ REPLACE('HellO World', 'O', '*');

↳ HellO W*rld

→
```
SELECT
    REPLACE('cheese bread Coffee milk', ' ', ' and ');
```

↳ o/p cheese and bread and coffee and milk

```
SELECT
    SUBSTR(REPLACE(title, 'e', '3'), 1, 10);
```

↳ Replace e by 3

only get 1 to 10 th index!

→ SELECT REVERSE ('Hello world'); ⟶ dlroW olleH

→ SELECT
CONCAT (author_fname, REVERSE (author_fname)) FROM books;

CHAR_LENGTH

* how many characters?

* SELECT CHAR_LENGTH ("Hello world"); ⟶ 11

→ SELECT author_lname, CHAR_LENGTH (author_lname) AS length
FROM books;

→ SELECT CONCAT (author_lname, ' is ', CHAR_LENGTH(
author_lname), ' ', 'characters long')
FROM books;

SQL Formatter ⟶ google & use.

→ UPPER(), LOWER()

SELECT UPPER ('Hello world');
SELECT LOWER ('Hello world');

DISTINCT

* SELECT DISTINCT author_lname FROM books; ⟶ Unique authors

* SELECT DISTINCT author_lname, author_fname
FROM books;
↳ means I want their unique
Combinations
(not just unique author_lname)

ORDER BY – Sort results.

1) SELECT author_fname FROM books
ORDER BY author_lname; ⟶ Default ASCENDING.

2) DESC ⟶ DESCENDING
ASC ⟶ ASCENDING.

3) SELECT released_year FROM books
ORDER BY released_year DESC;

```
SELECT title, author_fname, author_lname
FROM books ORDER BY 2;
```

↳ shortcut for author_fname (2nd column selected).

```
SELECT author_lname, title
FROM books
ORDER BY 2 DESC;
```

→ title descending order!

Sort by more than 1 columns

```
SELECT author_fname, author_lname FROM books
ORDER BY author_lname, author_fname;
```

↳ Sort by this

(any clash: ← Sort by this)

LIMIT

→ Specify - how many results?

→ `SELECT title FROM books LIMIT 3;` ⟶ 3 books (top)

→ 5mask recently released books

```
SELECT title, released_year FROM books
ORDER BY released_year DESC
LIMIT 5;
```

→ `LIMIT 0, 5;`

↳ Top five

→ `LIMIT 10, 1;`

↳ Skask from 10th book, Choose 1 book. (11th book).

10 → excluding

---

for selecting from some row to end

↓

use really large number!

```
SELECT * FROM tb1
LIMIT 5, 10;
```

```
SELECT * FROM tb1
LIMIT 95, 18446744073709551615;
```

→ largest number greater than Pks row limit → print upto limit

LIKE

★ BETTER Searching

Author may be → Dan / Dave

(I remember: Da)

→ WHERE author_fname LIKE '%da%'

↳ wildcards. (something/anything)

da → there → print

% → 0 or more characters.

'da %' → start with da

'% the' → end with the

'the' → exactly 'the'

'%the%' → Something the in it.

'____' → 4 digits / char exactly.

WHERE stock_quantity LIKE '____'; exactly 4 digits

(235) 234 - 0987. → LIKE ('%')

↳ Any thing b/w parenthesis

I want exactly three

LIKE '(___)'

LIKE '(___)___-____'

↳ Parenthesis, 3 dig, 3dig, hyphen, 4digits.

Book with % in it! (or) '_'

Escape character : \

'%.\%.%'

↳ start with anything, % sign, end with anything.

Aggregate functions

COUNT → how many books.

SELECT COUNT(*) FROM books;

→ SELECT COUNT (DISTINCT author_fname) FROM books; → 12

→ SELECT COUNT (title) FROM books
WHERE title LIKE '%the%';

→

GROUP BY

* GROUP BY summarizes or aggregates identical data into single rows

SELECT title, author_lname FROM books;
GROUP BY author_lname;

↓

giving unique author: pick 1st title (Actual order).

> Basically: grouping - Super row (with all books by an author) - but prints 1st one alone

Grouped data: → COUNT (each author: how many books?)

SELECT title, author_fname, author_lname FROM books
GROUP BY author_lname;

↳ print first book by authors!

SELECT author_fname, author_lname, COUNT (*)
FROM books
GROUP BY author_lname;

↳

DAN	Hasrouk	1
Dave	Hasrouk	1

Note: we are grouping using author_lname

↳ Same last name but unique in common first name

↳ Also grouped (wrong!).

wrong!

Dan	Hasrouk	2

SELECT author_fname, author_lname, COUNT (*) FROM books
GROUP BY author_lname, author_fname;

↳ Group by both!

(Avoid clash)!

```
SELECT CONCAT ('IN ', released_year, COUNT(*), ' book(s) released') FROM books
GROUP BY released_year;
```

MIN and MAX

```
SELECT MIN(released_year) FROM books;
```
→ 1945

```
SELECT title, MAX(pages)
FROM book_shop.books;
```
→ won't work
prints 1st book & MAX(pages)
↳ 'wrong'

what I need?

Book corresponds to maximum page. (The adv of Amazon...)

Solution: Involves subquery

```
SELECT * FROM books
WHERE pages = (SELECT MIN(pages) FROM Books);
```
— Subquery.

```
SELECT * FROM books
WHERE pages = (SELECT MAX(pages) FROM books);
```

(or)

```
SELECT * FROM books
ORDER BY pages ASC LIMIT 1;
```
→ other way! (No subqueries)

MIN/MAX with GROUP BY

Find the year each author published their first book.

```
SELECT author_fname, author_lname, MIN(released_year)
FROM books
GROUP BY author_lname, author_fname;
```

Longest page count for each author

```
SELECT CONCAT(author_fname, ' ', author_lname) AS author
    MAX(pages)
FROM books
GROUP BY author_lname, author_fname;
```

SUM

* Sum all pages in entire DB → same as → Sum all pages each author has written

```
SELECT CONCAT(author_fname, ' ', author_lname) AS author, SUM(pages)
FROM book_shop.books
GROUP BY author_lname, author_fname;
```

AVG

```
SELECT AVG(released_year) FROM books;
SELECT AVG(pages) FROM books;
```

Average released_year across all books

→ Avg stock quantity for books released in the same year.

```
SELECT released_year, AVG(stock_quantity) FROM books
GROUP BY released_year
ORDER BY released_year;
```

CHAR – fixed length

* 0 to 255 (length).
* when chars are stored – right padded with spaces.
* when retrieved – trailing spaces – removed

UNLESS

PAD_CHAR_TO_FULL_LENGTH

SQL mode is enabled.

* CHAR → faster for fixed length text

eg: CA, NY
Y/N
M/F

CHAR(5) → Everything 5 chars with space padded

Not fixed → USE Varchar!

DECIMAL

DECIMAL(13, 2)
- → After Decimal
- → No. of digits (Both before & after decimal)

999.99 DECIMAL (5,2).

3dig 2dig

5dig

* 7999987654 ⟶ 999.99 (Converted) ⟶ to max allowed!
* Rounded to two decimal places.

DECIMAL ⟶ fixed data type (calc: exact)

FLOAT, DOUBLE ⟶ Floating point types (approximate).

---

* FLOAT, DOUBLE ⟶ store large numbers using less space

'Comes at a cost of precision'

FLOAT ⟶ 4 bytes ⟶ ~7 digits (precision)

DOUBLE ⟶ 8 bytes ⟶ ~15 digits (precision)

---

* Always use decimal – unless precision doesn't matter
* BANK, Financial data – use decimal!

FLOAT

```
CREATE TABLE things (price FLOAT);
INSERT INTO things (price)
VALUES (88.45),      ⟶ 88.45
       (8877.45),    ⟶ 8877.45
       (8877665544.45) ⟶ 8877670000
```

Rounded! (Stored 7 dig alone)

Dates & Times

DATE ⟶ with a date not time

'YYYY-MM-DD'

TIME ⟶ No date only time

'HH:MM:SS'

DATETIME ⟶ Both

'YYYY-MM-DD HH:MM:SS'

```
CREATE TABLE people (
    name VARCHAR(100),
    birthdate DATE,
    birthtime TIME,
    birthdt DATETIME
);
```

```
INSERT INTO people (name, birthdate, birthtime, birthdt)
VALUES ('Padma', '1983-11-11', '10:07:35', '1983-12-28 10:07:35'),
('Laxmy', '1943-12-25', '04:10:42', '1943-12-25 04:10:42');
```

name	birthdate	birthtime	birthdt
Padma	1983-11-11	10:07:35	1983-11-11 10:07:35
Laxmy	1943-12-25	04:10:42	1943-12-25 04:10:42

Some functions:

- CURDATE() — current date
- CURTIME() — current time
- NOW() — current date time.

```
INSERT INTO people (name, birthdate, birthtime, birthdt)
VALUES ('microwave', CURDATE(), CURTIME(), NOW());
```

Formatting Dates

April 21st 2017 → documentation → massive task!

→ `SELECT name, birthdate FROM people;`

- DAY() → get day number (date)
- DAYNAME → returns that day!
- DAYOFWEEK (birthdate) → return which day (1 to 7)
- DAYOFYEAR (birthdate) → return which (1 to 365/366)

need 1983-12-12 (format)

but TIME (may be DATETIME) ✓

↳ NULL

MONTH( )

MONTHNAME( )

Time

HOUR(birthtime)

MINUTE(")

"2017-04-21" ⟶ "April 21st 2017"

→ Extract month name + ' ' + DAY(birthdate), ' ', YEAR(...))

CONCAT(MONTHNAME(birthdate), ' ', DAY(__), ' ', YEAR(__));

Better way → Refer documentation,

DATE_FORMAT(datetime, '%W %M %Y');

- %W → month name
- %M → Name of month
- %Y → year.
- Name of day!

%W - %M - %Y ⟶ Sunday - October - 2009.

%W ⟶ day name

%M ⟶ Name of month

%Y ⟶ year

%m ⟶ month number

DATE_FORMAT(-, '%m/%d/%y')

→ Some datetime.

%h
%m

Ref documentation!

Date Math

→ DATEDIFF(__, __); ⟶ Returns Interval.

↓ Datetime ↓ Datetime

→ DATE_ADD(date, INTERVAL expr unit);

↳ Format (HOURS, YEARS...)

DATE_ADD('2100-12-31 23:59:59', INTERVAL 1 DAY);

1 SECOND.

'Documentation'!

Otherway

```
SELECT birthdt, birthdt + INTERVAL 1 MONTH FROM people;
```

↳ directly

```
birthdt - INTERVAL 5 MONTH (FROM people;
```

TIMESTAMP

* Adding timestamps ⟶ Store info (meta data) - when something is created / updated.
* In MYSQL - TimeStamp is a datatype.

TIMESTAMP : Both date & time parts - Range [1970 - 2038 back]

DATETIME ⟶ 1000 to 9999

TIMESTAMP ⟶ 1970 to 2038

⟶ use: where date within this range! - use TIMESTAMP

why? : Takes less type (4 bytes) double : 8 bytes!

Same as Datetime except range!

```
CREATE TABLE comments (
    content VARCHAR(100),
    created_at TIMESTAMP DEFAULT NOW()
);
INSERT INTO comments (content)
VALUES ('lol funny');
```

⟶ 2017-04-21 20:26:36

"

2017-04-21 20:26:46

```
ORDER BY created_at;
```

```
CREATE TABLE comments2 (
    content VARCHAR(100),
    changed_at TIMESTAMP DEFAULT NOW() ON
        UPDATE CURRENT_TIMESTAMP
);
```

ON UPDATE ⟶ whenever rows changed – update!
(content changed)

skjdkasdja	2017-04-21	20:30:18
lolol	2017-04-21	20:30:25
I like cats and dogs	2017-04-21	20:30:45

```
UPDATE Comments2 SET content = 'changed' WHERE
Content = 'skjdkasdja';
```

Now:

changed	2017-04-21	20:33:02

Say: Auto update!

---

* CHAR → when we know 100% fixed length. (M/F), (Y/N)
* price (may be decimal – cents) ⟶ Decimal! DECIMAL(6,2)
  ∴ < 1 lakh.

DATETIME ⟶) DATETIME ⟶ big range ⟶ 8 bytes
TIMESTAMP ⟶ ⟶ small range ⟶ 4 bytes. ⟶ (meta data use!)

```
NOW();
CURTIME(); CURDATE()
DAYOFWEEK(CURDATE())  (or)  DAYOFWEEK(NOW())
                              ↳ 1 to 7
```

DATE FORMAT (%w)

```
SELECT DATE_FORMAT(NOW(), '%w');
                              ↳ 0 to 6
DAYNAME(NOW())
DATE_FORMAT(NOW(), '%W');
```

---

mm/dd/yyyy

```
SELECT DATE_FORMAT(NOW(), '%m/%d/%Y');
```

↓

CURDATE()

January 2nd at 3:15

April 1st at 10:18

↓ ↓ ↓

%M %D HH:MM

%h:%i

```
SELECT DATE_FORMAT(NOW(), '%M %D %h:%i');
```

```
CREATE TABLE tweets (
    content VARCHAR(100),
    username VARCHAR(20),
    created_at TIMESTAMP DEFAULT NOW()
);
```

Logical operators

AND
OR
NOT

`!=, =`

```
SELECT title FROM books WHERE title LIKE '%W%';
```

```
SELECT title FROM books WHERE title NOT LIKE '%W%';
```

↓

No 'W' in any title!

`>, >=,`

```
SELECT 99 > 1; ⟶ 1 (TRUE)
               0 (FALSE)
```

'a' < 'b' ⟶ yes

'A' > 'a' ⟶ no!

'A' = 'a' ⟶ yes! (equal)

`<, <=`

&& ⟶ Logical AND

BETWEEN (upper and lower range)

without between: `>= 2004 AND <= 2015`

Syntax: `BETWEEN x AND y`

↓

NOT BETWEEN

NOT BETWEEN 2004 AND 2015;

Side note: Comparing dates:

CAST() → Convert the values to the desired data type.

eg: compare DATETIME to two DATE values (or) DATE → DATETIME] Convert one type to another.

'2001-1-1' → String → Convert to DATETIME

'Just Cast'

eg: '2017-05-02' → Just a text

SELECT CAST('2017-05-02' AS DATETIME);

O/P: 2017-05-02 00:00:00

SELECT name, birthdt FROM people
WHERE birthdt BETWEEN '1980-01-01' AND '2000-01-01';

Text not a DATE

But do this ← But mySql smart: Converts to DATETIME, do the job.

CAST('1980-01-01' AS DATETIME)

↳ failsafe!

IN

SELECT All books written by Carver or Lahiri or Smith

one way: WHERE author_name = 'Lahiri' or
_____ = _____ or
_____ = _____

other way:

WHERE author_name IN ('Lahiri', 'Smith', 'Carver');

NOT IN

WHERE released_year NOT IN (2002, 2004, ... 2016);

% operator: released_year % 2 != 0

CASE Statements

```
SELECT title, released_year,
CASE
    WHEN released_year >= 2000 THEN 'modern lit'      → if else
    ELSE '20-th century lit'
END AS GENRE
FROM books;
```

Lower than 50 *
100 **
100+ ***

title, stock_quantity, stock,

---

10 != 10 → 0

15 > 14 && 99 - 5 <= 94 → 1

SELECT 1 IN (5,3) || 9 BETWEEN 8 AND 10; → 1

`SUBSTR(___, 1, 1) = 'C'` → or use LIKE 'C%'

Joins

Books → orders, Reviews, customers, versions, Authors

1) one to one Relationship → Not common
2) one to many
3) many to many

one to one: username & password. ...
Details of customer

One customer has one detail row → Not Common!

* One books — many review (one to many)
* many to many → book → many authors; author → many books

1: Many

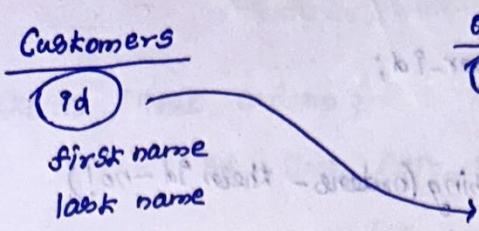
Customers & orders.

Customer first name, last name
email
date of purchase
price of the order

first	last	email	date	amount

Lot of Repitations! (Against rdbms)

Duplication data - Avoid



Primary key - unique (cust_id)

Foreign key: ref to another table within a given table (cust_id)

Constraint: foreign key must be there in Customers (else can't be added in Orders)

Cross join

SELECT * FROM customers, orders; → all cust ⇄ customers

"Permutation"

Joiny every cust with every order.

SELECT * FROM customers, orders → only cust & their corr. orders.
WHERE customers.id = orders.id;

↓

Inner join

only when match.

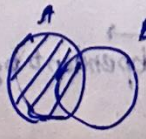
Explicit inner join: → use this!

```
SELECT * FROM Customers
JOIN orders
ON customer.id = orders.customer_id;
ORDER BY amount;
```

(or)

```
GROUP BY customer_id;
```

→ print 1st alone (no sense)!



everything from A (including common ones).

```
SELECT * FROM customers
LEFT JOIN orders
    ON customers.id = orders.customer_id;
```

use: some customers may not bought anything (orders - their id - no!)
↳ NULL.

---

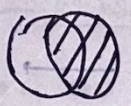
case: we need sum of each cust purchases

SUM (NULL) = NULL [I need 0]

IFNULL (SUM(amount), 0)
↳ replace with!

```
ISNULL (SUM(amount), 0) AS total
```

Right join



```
DELETE FROM customers WHERE first_name = 'Boy';
```

↳ this ∴ Reference (foreign key)
doesn't allow to create faulty / non existent ones
so we can't delete the dependant!

My way: delete customer + all orders associated with it.
(Dependency)

DROP TABLE customers; ⟶ delete customers (not possible) ∴ dependant orde

```
DROP TABLE orders;
DROP TABLE customers
```

```
DELETE FROM customers WHERE email = 'george@gmail.com'
```

↓
error

∴ Customer ⟶ dependant by Orders.

I need - delete customer

↳ orders of them deleted (automatically)

```
CREATE TABLE orders (
        id  INT  AUTO_INCREMENT  PRIMARY KEY,
        ⋮
        FOREIGN KEY (cust_id)
            REFERENCES Customers (id)
            ON DELETE CASCADE  → deletes job!
```

```
DROP TABLE Customers,
WHERE  Customers.id = 2 ;
```
] No error!

Exercises

Students
- id
- first_name

Papers
- title
- grade
- student_id

IFNULL (title, 'Missing')

IFNULL (grade, 0)

```
CASE
    WHEN AVG(grade) IS NULL THEN 'missing'
    WHEN AVG(grade) >= 75 THEN 'passing'
    ELSE 'FAILING'
    END AS STATUS
```

Many to Many Relationship

MANY : MANY

Books ↔ Authors

Blog Post ↔ Tags → [Content – tags → used in many contents] → many tags

Students ↔ classes

each have many photos.

★ Student → multiple classes

★ classes → multiple students.

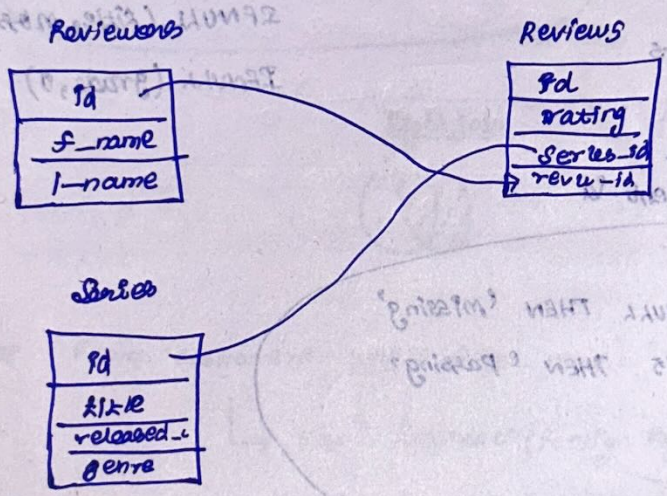
TV show receiving applications

Key idea: Sign up (reviewer) → 3 tables (union table/Join table)

Series data
↕
Reviews data.
↕
Reviewers data

Series → many
Reviewers → many
connected by Reviews.

Reviewers	Reviews	Series
id	rating	title
first_name	series_id	released_year
last_name	reviewer_id	genre.
		id



Reviewers

id	f_name	l_name
1	—	—
2	—	—

→ Reviewers

id	title	released_year	genre
1	—	—	—
2	—		

→ Series

Reviews

id	rating	re_id	Series_id
—	—	—	—
—			

* title, Rating → all title + its ratings.

first_name	last_name	rating

title	rating	reviewer

3 diff tables!

Instagram

USERS PHOTOS HASHTAGS

COMMENTS LIKES FOLLOWERS/FOLLOWEES.

1) USERS → id (int, pri, Auto inc)
   - → username (VARCHAR(255), UNIQUE, NOTNULL)
   - → created_at (TIMESTAMP DEFAULT NOW())
     - → Small & easy to store

we use as primary key
(but: long: slower!)

2) Photos
   - → id (int, inc, prim)
   - → img_url (varc (255), NonNull)
   - → user_id (int, NOT NULL) → user_id → id of users
   - → created_at (Default)

3) Comments → id
   - → Comment_text
   - → user_id → id of users
   - → photo_id → id of photos
   - → Created_at

4) Likes
   - → user_id → users 's id
   - → photo_id → photo's 's id
   - → Created_at

Ensure 1 like per user

1 photo, 1 user → combination (unique)

make them primary!

→ Avoid duplications

Follows:

follower_id → id of USERS

followee_id → id of USERS

created_at

PRIMARY KEY (Combination - only one)!

Tags → (Limited tags - only stored)

Can't store add. info

Have to be careful with search

→ '#pets #cats #dog' → As string [not good]

→ other idea: Seperate tag table.

tag_name	photo_id
#cute	1
#cute	3
#microwave	2
#gross	2

don't need to worry how many ids in string!

slower!

Other Solution

Three tables → photos

→ tag table (tag_name, id)

→ photo_id, tag_id

photos
Tags
photo_Tags

→ save storage

∴ No need to put string multiple times.

* orphans - delete a tag! (Remove it from photo_tags)

* Cost: nice structure, extra work!
  ↳ use this!

material: Tag system! ⟶ Common tags (lot of space)!

```
CREATE TABLE tags(
    id INTEGER AUTO_INCREMENT PRIMARY KEY,
    tag_name VARCHAR(255) UNIQUE
    Created_at TIMESTAMP DEFAULT NOW()
);
```

USERS

```
CREATE TABLE users(
    id INTEGER AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    Created_at TIMESTAMP DEFAULT NOW()
);
```

Photos

```
CREATE TABLE Photos (
    id INTEGER AUTO_INCREMENT PRIMARY KEY,
    image_url VARCHAR(255) NOT NULL,
    user_id INTEGER NOT NULL,
    Created_at TIMESTAMP DEFAULT NOW(),
    FOREIGN KEY (user_id) REFERENCES users (id)
);
```

Comments

```
CREATE TABLE Comments (
    ;
)
```

⟶ Ref github.

→ Reward – based on – who have been around the longest

1) 5 oldest users:

2) what day of the week do most users register [ad campaign]

DAYNAME (_____) ← TIMESTAMP

GROUP BY day;

```
SELECT DAYNAME (created_at) AS day,
       COUNT(*) AS total
FROM users
GROUP BY day
ORDER BY total DESC
LIMIT 1;
```

Inactive users

* never posted a photo.

* users table, photos table.

```
SELECT username FROM users
LEFT JOIN photos ON users.id = photos.user_id
WHERE photos.id = NULL;
```

who can get most likes (who won)

* most popular photo & user who created it

likes, photos, users

How many times does the average user post

```
SELECT
    (SELECT (COUNT (*) FROM photos) / (SELECT COUNT (*)
    FROM users) AS avg;
```

which tag to use in a post

Top 5 most common!

★ most used tag! (users, tags, photo_tags)

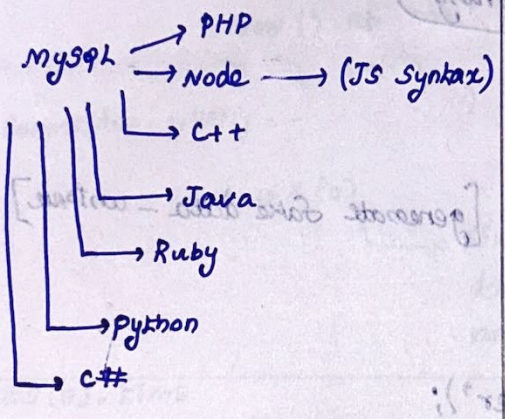
boks

★ liked every single photo on the site!

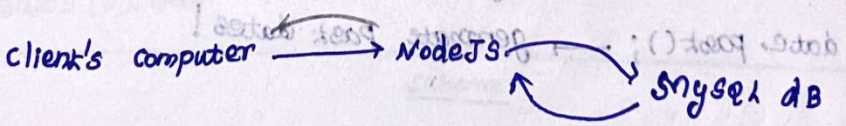
```
SELECT username, COUNT(*) AS total
FROM users
INNER JOIN likes
    ON users.id = likes.user_id
GROUP BY likes.user_id
HAVING total = 257;
```

Node JS

★ Combine ★ with Nodejs [web apps]



Interact with mysql — external code!



Start up mailing list application

collect mail

connect mysql with node!

JOIN US

--- 518 others. 100% free

mail

Join now

USERS → email
→ created_at

---

* use NodeJS to randomly generate & insert 500+ users in a dB

Setting up environment:

```
Console.log("Some Code");
```

```
Node filename.js
```

```
for (var i = 0; i < 500; i++){
    Console.log("Hello world");
}
```

```
for (var i=0; i < 500; i++){
    // Insert new user
}
```

huge library

npm install faker

↳ package [generate fake data - unique]

```
var faker = require('faker');
faker.internet.email(); ⟶ generates email.
faker.date.past(); ⟶ generate past dates!
```

Node package

* Adapter / connector.

NodeJS ⟷ MySQL

```
var Connection = mysql.createConnection({
    host : 'localhost'
    user : 'root'
    database: e ?
});

var q = 'SELECT CURDATE()';
```

```
connection.query ('q', function (error, results, fields){
    if (error) throw error;
    console.log('The solution is: ', results[0].solution);
});
```

results → [{ 'CURDATE()' : Mon May 01 . . . }]

```
connection.query ('SELECT 1 + 6 AS answer', function (error,
                                    results, fields){
    ____
    ____ results[0])  → 2017-05-01
});
```

```
var q = 'SELECT CURTIME() as time, CURDATE() as date,
         NOW() as now';
connection.query . . . . . );
});
```

results [0] → time :
date :
now :

---

results[0].time
results[0].date
results[0].now

Schema

```
CREATE TABLE users (
    email VARCHAR(255) PRIMARY KEY,
    created_at TIMESTAMP DEFAULT NOW()
);
```

Inserting data

```
var q = 'INSERT INTO users (email)
         VALUES ("wyatt@gmail.com")';
```

using faker

```
var q = 'INSERT INTO Users (email) VALUES ("' + faker.internet.email() +
        '")';

connection.query(q, function (error, results, fields){

});
```

Other way

```
var person = {email: '_____'};

connection.query('INSERT INTO users SET ?', person, function(
        error, result){

});
```

means get from 2nd argument.

```
var person = {email: faker.internet.email()};
```

Is way of dates using faker

↓

not compatible! (with mySQL)

Solution:

```
var person = {
        email: faker.internet.email(),
        created_at: faker.date.past()
};
```

→ Now not created_at is

0000-... 00:00:00..

magic of faker

Note: by directly passing date in mySQL – Not working

do by using JS ⟶ working (behind the scenes)

See all

```
var end_result = Connection.query('INSERT INTO users SET ?', ...)
...
});
```

↓ Save in a variable.

```
Console.log(end_result.sql);
```

---

multiple users

```
var data = [
  ['e_____', 'e_____'],
  ['ugh@', '_____'],
  ...
];
```

→ ∴ no key value pairs. So [ ]

```
var q = 'INSERT INTO users (email, created_at) VALUES ?';

Connection.query(q, [data], function(err, result){
  ...
});
```

```
var data = [];

data.push([
  faker.internet.email(),
  faker.date.past()
]);
```

→ create array of arrays!

```
Connection.query(q, [data], function(err, result){
  ...
});
```

Find earliest user joined.

* Express → framework! [Package]

client → / Comp ←

Express app (Node)

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
    res.send('Hello');
});

app.listen(8080, function () {
    console.log('Listening');
});
```

(Incoming request) → home page! – path is nothing only '/' (get i/p)

listen & do job.

---

use

* localhost 8000 port
* res.send("You've reached the home page!");

req → from browser
res → from us.

Adding routes

/joke

```
app.get("/joke", function (req, res) {
    var joke = "What do you call a dog?";
    res.send(joke);
});

app.get("/random_num", function (req, res) {
    var num = Math.floor((Math.random() * 10) + 1);
    res.send("Lucky number: " + num);
});
```

Integrate mysql

HTML

EJS – Embedded JavaScript

```
app.set("view engine", "ejs");
```

res.render('home')

npm install --save ejs

→ must be in a folder named views

Inserting data in html

<%= data %>

res.render("home", {data: count});

<%= 4 * 98 %> → evaluate!

→ give value during rendering as json.

MySQL - database triggers

→ SQL statements that run automatically - when a specific table is changed.
(triggered)

```
CREATE TRIGGER trigger_name
    trigger_time  trigger_event  ON table_name. FOR EACH ROW
BEGIN
....
END;
```

trigger_time → Before | After

trigger_event → INSERT | UPDATE | DELETE

table_name → eg: Photos
users.

```
DELIMITER $$
CREATE TRIGGER must_be_adult
    BEFORE INSERT ON people FOR EACH ROW
    BEGIN
        IF NEW.age < 18
        THEN
            SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'Must be an adult!';
        END IF
    END
$$
DELIMITER ;
```

→ Run the trigger! (don't allow < 18)

DELIMITER →

NEW.age → Incoming one

OLD.age → Inserted one.

SIGNAL → For showing error!

error → numeric error code

→ Sql state (5 char value) – MySQL specific

→ message – textual desc.

(3 parts)

45000 → A generic state prep. unhandled user-defined exception

$$ → make a block

Prevent bells & follows

```
CREATE TRIGGER trigger_name
    trigger_time trigger_event ON table_name FOR EACH ROW
    BEGIN
    END;
$$ DELIMITER ;
```

```
DELIMITER $$
CREATE TRIGGER prevent_self_follows
BEFORE INSERT ON follows FOR EACH ROW
BEGIN
    IF NEW.follower_id = NEW.followee_id
    THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'You can't follow yourself';
    END IF;
END;
$$
```

Logging unfollows

```
DELIMITER $$
CREATE TRIGGER capture_unfollow
    AFTER DELETE ON follows FOR EACH ROW
    BEGIN
        INSERT INTO unfollows (follower_id, followee_id)
        VALUES (OLD.follower_id, OLD.followee_id);
    END
$$
```

other way

```
SET follower_id = OLD.follower_id
SET followee_id = OLD.followee_id
```

Manage triggers

```
SHOW TRIGGERS;
```
→ Some big lines

Remove trigger

```
DROP TRIGGER trigger_name;
```

warning

TRIGGERS make → debugging hard!

Say: Insert → 2 items created! → problem!

Say bug in triggers (can't be identified) – easily.

Bunch of triggers – forgot → difficult mess!

Data breach at Kalo Corp

Keiko Corp:

Database:

* Collection of data – method for accessing & manipulating that data
* 2.5 quintillion bytes of data – every year
* Capture – database (do)
* 44 – zettabytes (our digital world)
* Hardware & Software (Computers) →

Google excel → dB

→ drum memory [disk drives: cylindrical]

Convention: dB – rep by drum memory!

excel: too much data – don't handle

[Integrity, lot of data, Automate, Combine] → dB do!

→ Hadoop, Presto (FB), big query (google), Redshift (Amazon)

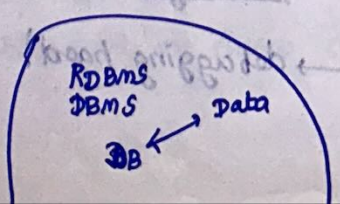
→

1. Put data in dB
2. CRUD
3. Remove

DBMS – database man. system (Software)

* Receive info → do job
* RDBMS – subset of DBMS
* Sql

Sql



PostgreSQL

Customers

CustID Name Addr

orders

orderId custid data ProductId quantity

Products

proId Name Quant.

→ created relations (linked to one another)

Types of DB

- RDBMS
- Object
- Too much there!
- Last in memory

* Relational models [asset transaction]
* Document [mongo, Couche, Firebase] → document (rather than rows & Columns)
  - ↳ Scalable.
* Key value: [DBnamo DB] – Simple way [Key : Value]
* Graph [Neo4j, AWS, Neptune] – more [complex]. [Social networks]
* wide Columnar [new – pioneered by google's big table] → apache, Cassandra.

---

what is a dB

System – hardware + Software → allows → store, organize, use data.

Tools: db-fiddle.com

Instructions → queries [question] → questioning dB [Sql statement]

* → wildcard [everything]

```
SELECT NAME   FROM   USERS        -- Comments
WHERE  ROLE = "MANAGER"
```

- SELECT NAME FROM USERS → Identifiers
- WHERE ROLE = "MANAGER" → Cond → expression

SQL – declarative language.

[what will happen – How it happen]

- what will happen → Simply stating!
- How it happen → No idea (Imperative prog – how it will happen)

SQL → declarative → more flexibility.

Java → Imperative (Python - Both) → using libraries.

---

* SEQUEL → org. name → Structured English query Language.
* 70s [Edgar Codd → prog of IBM → wrote paper → Stds for dB.] & SQL
* paper: A relational model of data for large shared data banks.

↓

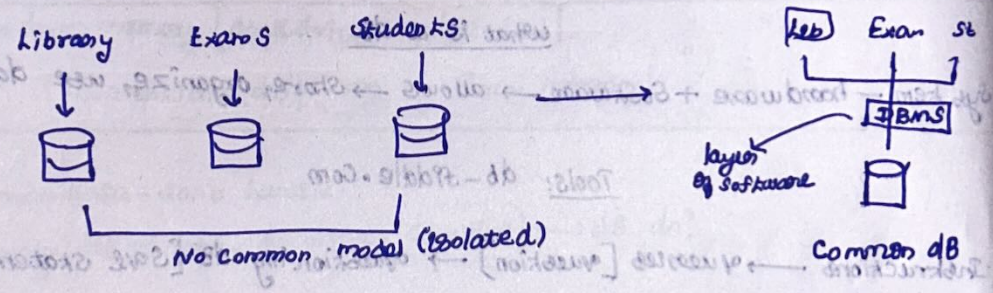
Inspired (IBM) → Donald Chamberland & Raymond Boyce

Implemented.

* Both language & Software to create & manage dB.
* SQL - Standardized

File processing Systems

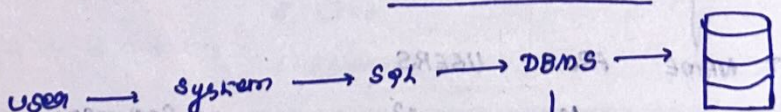
* Huge leap ahead - with flaws
  * Indiv files → No relationships b/w files → too much work
  * edit CRUD → difficult
  * File processing systems [Java, C, C++, Python..]



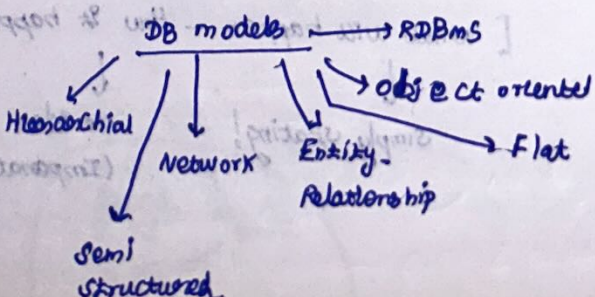
→ Library - Lending (take info from students)

→

Database approach

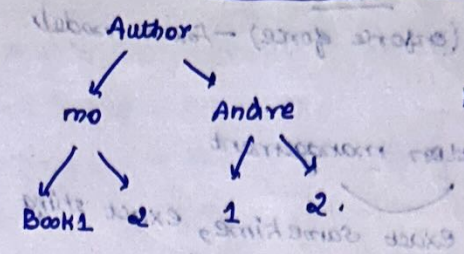


No care
(Linux/windows/anything)
(O/p: SQL) - Unified.



Hierarchical model – old (IBM 60's 70's)

* Not efficient



Parent, child

child → single root

↳ No linking (Coauthoring)

(outdated) – XML

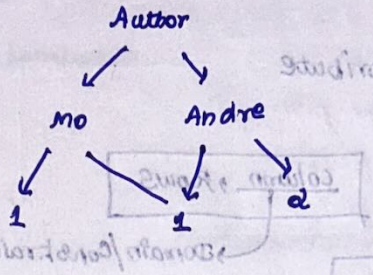
```
<MO>
    <Name> MO Binni </Name>
    :
</MO>
```

Parent → deleted → child deleted } tightly coupled.

Disadv: No many to many (or) many to one

Networking model (exp: hierarchical)

* allow Coauthoring!



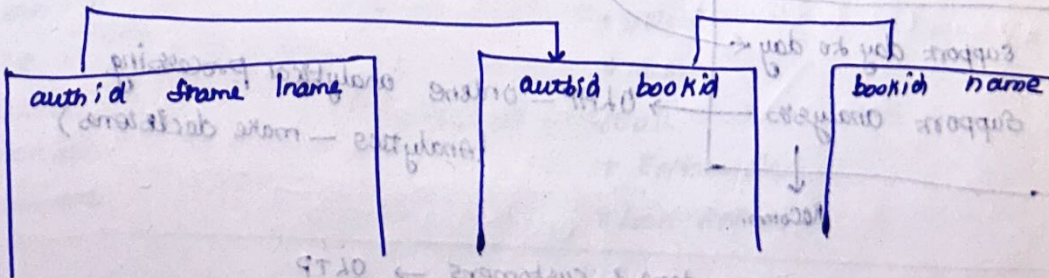

```
<MO>
    < Book1 author = "Andrei" relation = "coauthor" />
```

↳ make relation!

Relational model:

* Many to many → delete [every other parent - delete] → that node! (too much work)

Table structure [Relations]



3 tables.

DBMS

* manager → knows everything (enforce force) — follow model
* CRUD
* security, permission, transaction management

exact same time, exact thing

update → undo the job!

Corrupt data → backup.

Microsoft
IBM
oracle
MySQL
Postgresql

CORE → SQL — Compete using features.

One single model → RDBMS

13 rules: [Edgar Codd] → RDBMS

Relational model:

Column → specific attribute

Column, Rows

→ Domain/Constraints (attributes)

Collection of Columns → degree (AU)

Tuples/Rows [data]

[Collection of Rows: Cardinality] → how many rows.

Primary & Foreign Key.

Primary key: uniquely identifies data.

Foreign key: Reference from somewhere else [tables]

OLTP (vs) online transaction processing.

(Just display) — User can select, view

Support day to day

Support analysis → OLAP — online analytical processing.

(Analytics — make decisions)

Recomm.

---

log orders & customers → OLTP

what new products offer → OLAP

derive statistics → OLAP

} not day to day.

keep track of logged in users → OLTP & OLAP

URL - LOCALHOST (Localmachine)

PORT - door (address) - open door Comm.

USERNAME - POSTGRES

PASSWORD - ROOT

CONNECTION URL - LOCALHOST.

Login as user

PSql -U postgres

#postgres \conninfo → Show dB connected info.

Create tables

CREATE TABLE test_table ();

- ★ \dt → relations
- ★ \l → list of databases
- ★ \q → exit.

Load databases

→ PSql -U Postgres -d Employees < employees.sql

password:

(or) → dump in menu → Just stick with mySQL

Sql Commands → DCL → GRANT
→ REVOKE

DDL
- ★Create
- ★Alter
- ★Drop
- ★Rename
- ★Truncate
- ★Comment

DQL
- ★Select

DML
- ★Insert
- ★update
- ★Delete
- ★Merge
- ★Call
- ★ Explain plan
- ★Lock table.

DCL - Data Controlling language (give, Revoke access)
DDL - Data Definition language
DQL - Data query Language
DML - Data modification Language

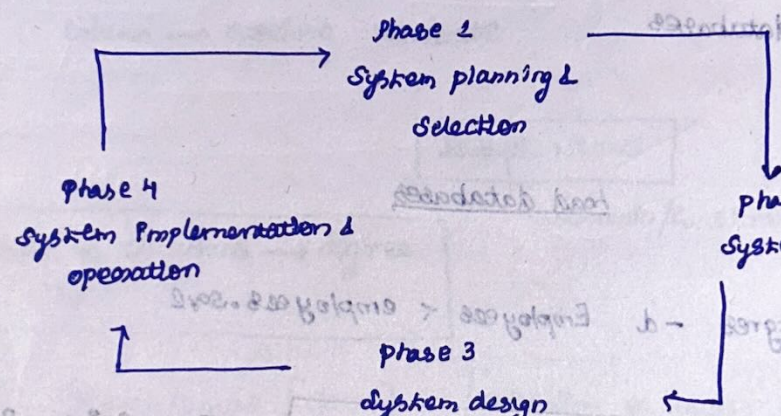
Partition, Rollover, Views

DB
- Regular (From base)
- Template → Never change it (Blueprint)

Roles in mySQL

System design

SDLC - Software Dev life cycle.



Robust → updatable

Implementation ways

Agile, Waterfall, V-model, ...

* Agile: move rapidly (adjust as you go) [ship, test frequently]
* waterfall: process needs to be followed in a regimented way (cycle)
  healthcare.
  ↳ well tested - no screen change. (heavily regulated)
* V-model: (V shaped)
  → In hardware (Slow life cycles)

SDLC phases

Phase 1: Requirements Analysis
Scope? (whether can be done to be done)

Phase : 2 : Whether on time, on budget → Analysis

phase : 3 : Design System architecture → dB, Apps etc.

phase : 4 : Build Software.

---

* Testing, maintenance. ... (other phases possible)

* Test often, early, frequently
* Data techniques to design dB.

Top-down design

* Top down — Starts from zero [from base] — optimal choice (Scratch)
* Gathered upfront (all that need - ∴ there is nothing (fresh))

Bottom up

* Existing - System/data [doctors - existing - match with persons symptoms] - Shape a new system (migration)

*

Driving academy

Phase 1: No online presence - Across US - Employee (pay roll) and Inventory of cars, trucks & motorcycles.

CEO, CTO, BA's, Customers → need dB.

understand business — design!

Now: outdated website — word of mouth (customers)

Want: Strong online presence.

Core requirements:

* well thought - out document with Sections eg functionality.
* Vehicle Inventory for Students to rent.
* Employees at each Branch
* Maintenance for the vehicles.
* optional exam - (twice) end → 80 % → must take lessons.

model a dB

Top down design

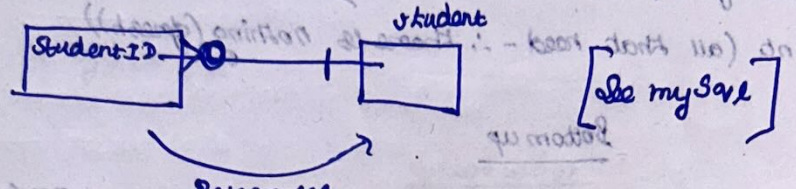
- High level requirements
- User interviews
- Data collection
- Deep understanding

ER-modeling

Top down → ER modeling

Bottom up → Normalization

* Key methods of top down design
* Entity-Relationship model / diagram
  (Structure high-level requirements)



Entities - determine

* Person / place
* Singular name
* has identifier
* Should contain more than 1 instance of Data

Tooling

* UMLet 14.3 - Free UML Tool

Drive me School

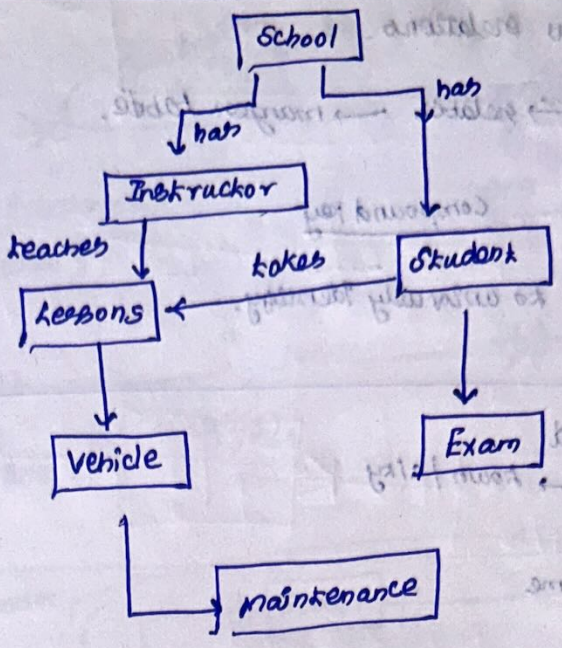
Entities:

Student — School/Branches — Teachers

Vehicle — Maintenance — Exam

Lesson

'More eyes
Better results'



Attributes (give entities the info they store)

* property of entity (belong)
* Atomic - single value [mask small amount of info]
* Single / multivalued (phone numbers)
* Keys

Schema & Instance

Relational Schema → table. & its attributes.

(table)

USER

Relational schema

Relation Instance

Id				DoB

Date Time (Constraint)

Super & Candidate Key

Relation key

Id

Superkey (Id, firstname → Superkey (unlovin))

But what's simplest? → primary key.

→ Candidate Key: minimal one (enough to uniquely identify)

Primary Key: Relation key (Sole simplest)!

Foreign key → draw relations.

m1 (manager id) → relates → manager table.

Compound key

* multiple columns to uniquely identify.

School → School id
- town / city
- address
- Street name
- Street number
- Postal code
- State

Teacher → id
- last name
- first name → dob
- joining date → School id

Student → id
- fname
- lname
- Enrollment_date

doesn't Student enroll in lessons?

Exam → Student_id
- teacher_id
- taken_date
- passed (true/false)
- lesson_id

Lesson → id
- date_of_enrollment
- package
- Student_id

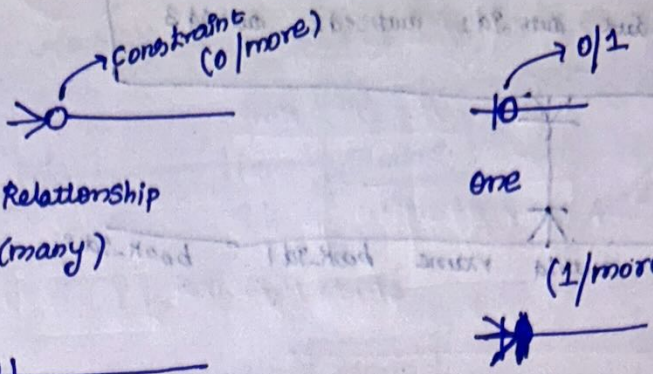
Relationships

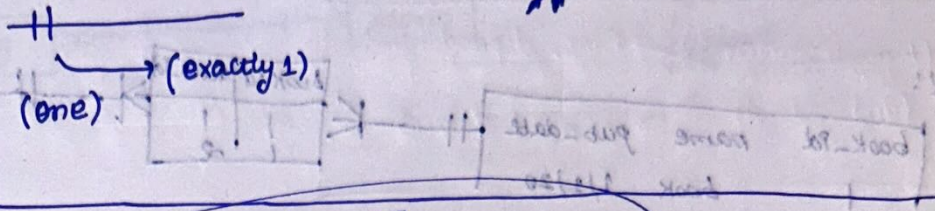
1 - 1

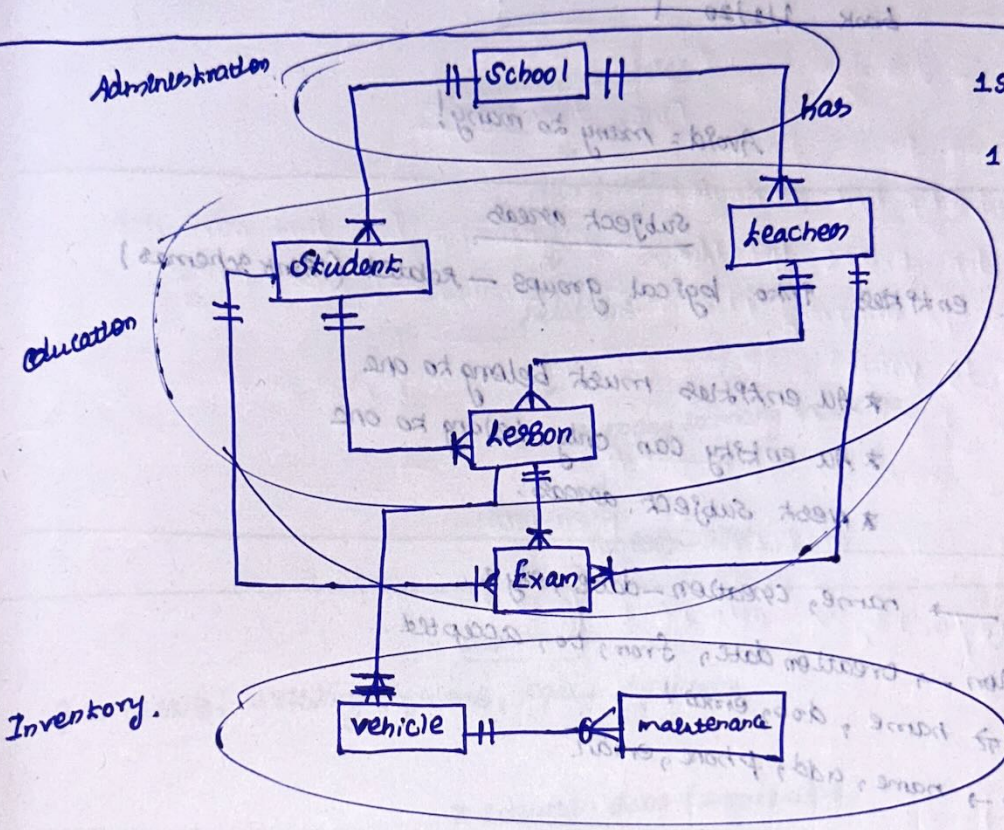
1 - many

many - many.

Crows foot :

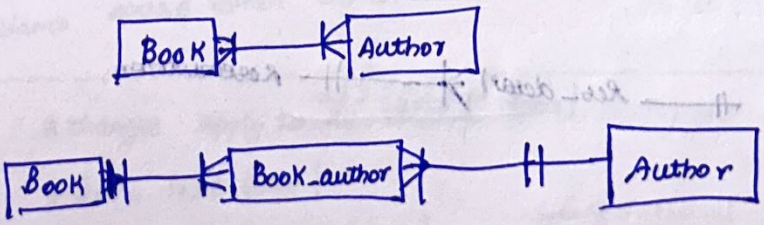




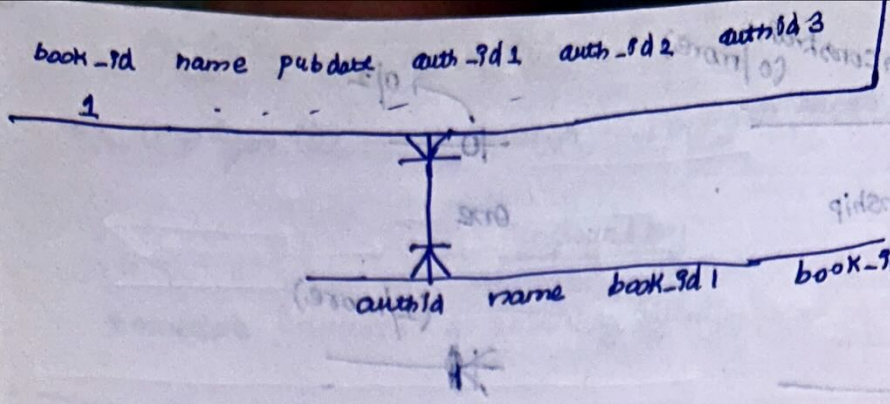


Many to many

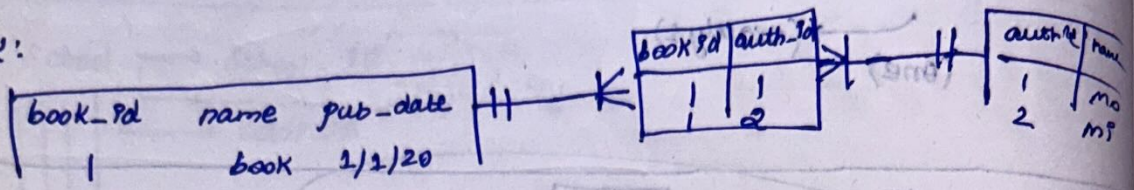
* Can't store in a single field.
* Try to avoid → lot of overhead
  * Insert overhead
  * Update overhead
  * Delete "
  * Potential redundancy.



3 tables — Now not many to many.



Now:



Avoid: many to many!

Subject areas

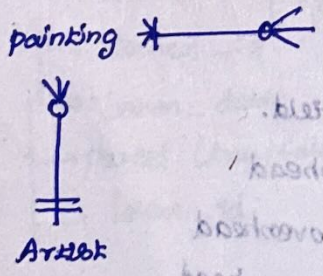
* Divide entities into logical groups — related (think schemas)
  * All entities must belong to one
  * All entity can only belong to one
  * Nest subject areas.

Painting → name, creation_date, style

Reservation → creation date, from, to, accepted

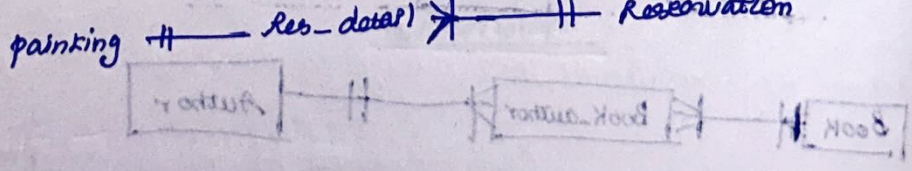
Artist → name, dob, email

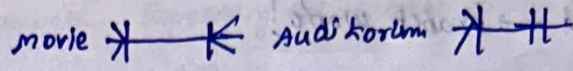
Museum → name, add, phone, email.

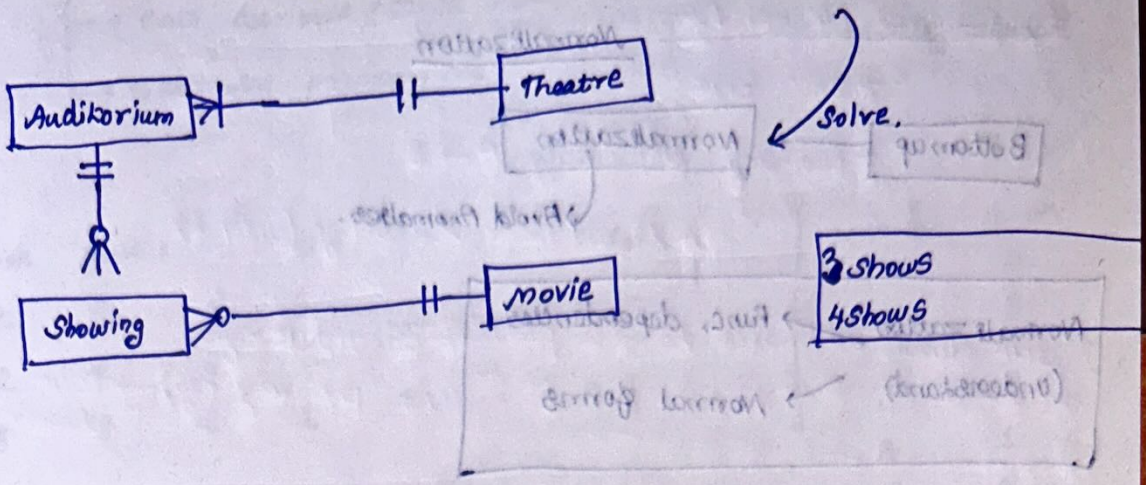


Artist → 0/more paintings
painting → 1 artist

Paint → 0/more reservation
Reservation → book multiple paintings.







Determine entities

↓

Attributes

↓

Relations

↓

Solve many to many relation

↓

Subject area

Bottom-up

Spec details, existing systems, legacy systems.

* Identify data (Attributes)
* Group them (entities)

→ No Anomalies / Redundancy

Anomalies

- Update
- Insert
- Delete

* problems occur when dB Structured Incorrectly : modify Anomalies

---

* changes apply to all related data!
* don't lose data!
* check: Inconsistent data!

Customers

Branch!

So even cust → deleted → branch won't!

Normalization

Bottom up → Normalization → Avoid Anomalies.

Normalization → Func. dependencies
(understand) → Normal forms

Func. dependencies

* Relationship b/w attributes.
* Func dep exists → a relationship b/w two attributes allows you to uniquely identifies (determine) — corresponding attribute's value.

B ⟶ A

Student_Id ⟶ Birthdate

* delete / modify A affects B (good dBMS won't allow)

0NF
1NF
2NF
3NF
BCNF
4 NF
5 NF
6 NF

each normal form — aims to seperate Relationships into smaller instance to create less Redundancy & Anomalies.

0NF to BCNF → Common

6NF → Not yet standardized.

0NF to 1NF

* 0 NF → Unnormalized
* Repeating groups of fields
* positional dependence eg. date
* Non atomic. [In postgres — not possible]

1NF ⟶ eliminate Repeating columns of same data → closely related.

- → each attribute (Single value)
- → determine primary key.

book	auth 1	2	3
1	1	2	3
2	2	2	3
3	3	2	1

⟶

book
1
2
3

auth id	book
1	1
1	3
2	2
2	3
3	1
3	2
3	3

1NF to 2NF

* It is in 1NF
* All non-key attributes - fully functional dependent on primary key.

2NF to 3NF

* It is in 2NF
* No transitive dependencies

A → func. dep on B

B → fun - on C

C → fun dep on A via B

B → A

C → B

A ⇝ C

branch no → province

province → Country

branch no ⇝ country

Solve

Country table with province → no!

province table with Country ✓

bank no → province ✓

pd
Country
province name

BC NF

Boyce – Codd Normal Form

* In 3NF
* For any Dependancy $A \rightarrow B$

A should be a Super key. (unique)

3NF allows attributes to be a part of Candidate key (Not primary key) — BCNF doesn't.

why? → Foreign key: may be repeated

while → primary/candidate key: not.

BCNF

* The primary key is Composite
* more than 1 candidate key
* Some attributes in keys are common

} Not in BCNF

stu_id	tut_id	tutor_sln

Candidate keys: stu_id, tu_id
stu_id, tu_sln
} make it 1.

stu_id	tut_id

tut_id	tut_sln

why 4NF, 5NF – not useful

* Not generally used
* over normality: functional loss (we need freedom).
*

Scalability

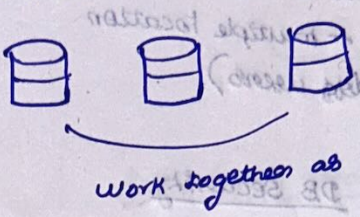
* capability - handle growing data

vertical scaling → more capacity (disk space, power) - physical hardware

Problem: expensive, limitation (no more add - make it powerful)

Horizontal Scaling:

* Add more machine



work together as single.

MongodB, NoSQL → able to do Horizontal scaling!

challenges: Adding machines - comm. with each other
(Avoid inconsistency)

update data in dB → update in other as well

Time latency!

Complexity.

Sharding

* sharding : FB (ton of data) → split (data) → A-J, K-T, U-Z

So direct to dB according to A-Z

* logically
* How shard?] space/ layer

Replication

Loss of data! → Replicate data in multiple machines.
(every once in a while) → one in India, US, UK

Replication → Synchronous (do all in one) → Bank balance] problem
→ Asynchronous (In a while) → Insta like count - no prob

Backups

Replication — happens in real times

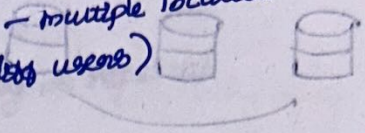
Backup: expensive (done often) — in a usable [reg backup of entire DB]

Avoid disasters [dump]

Revert back!

Distributed (vs) Centralized

* Single location — Centralized — easy data integrity
* Distributed — multiple dB — multiple location (more resistant, data users)

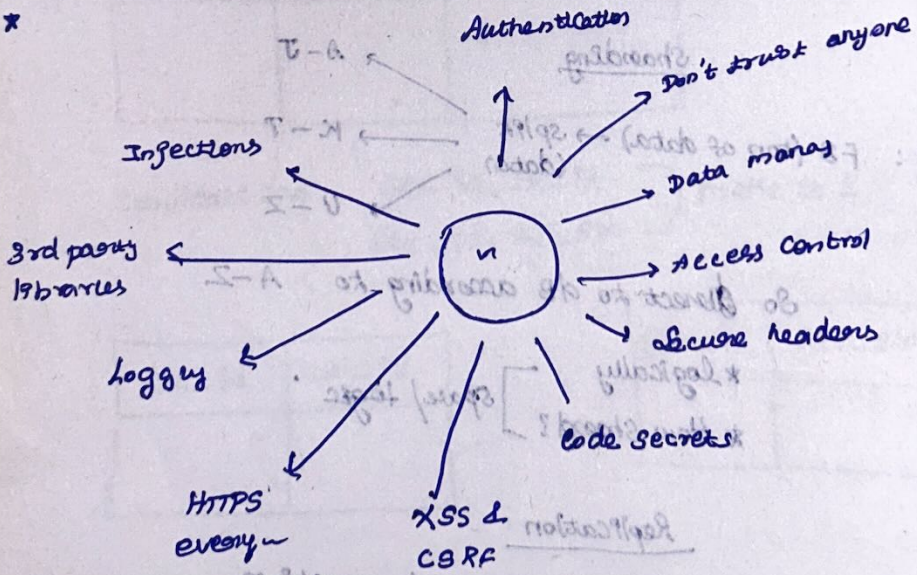


DB Security

* prevent data Corruption
* Tech glitches
* Hackers
* Malware
* Illegal View
*

Sql Injection

*



' (or) 1 = 1 — —

' ; DROP TABLE users; — —

1) Sanitize I/p

2) parametrize queries

3) Knex.js (or) other ORMs.

Store passwords

* Encrypt [bcrypt, scrypt, Argon2]

pgcrypto — encrypt a few columns

↳ library!

Relation vs NoSql

mongodB (vs) postgreSql

↳ doc model [JSON spec]

LinkedIn ⟶ mongodB (All related in one location)

Future of RDBMS

* RDBMS has now horizontal scaling

(Spanner, Vitez, CockroachDB, Spanner)

Elastic Search

* Common dB ⟶ doc model (good for searching)
* RDBMS – not good at text searches
* less reliable! (Slower write)

↓

trade-offs

Amazon S3

* object storage! [doc mode: Sheller]
* dump & retrieve.

Top dB

MySQL, postgres, mongodB.

Amazon document DB

Firebase, ElasticSearch