

## Retrieving Data From a Single Table

### The SELECT Statement

```

| SELECT *
| FROM customers
| -- WHERE customer-id = 1
| ORDER BY first-name
  
```

```

| SELECT first-name, last-name
| FROM customers
  
```

### The SELECT clause

```

| SELECT
|   last-name,
|   first-name,
|   points,
|   (points + 10) * 100 AS 'discount factor'
| FROM customers
  
```

```

| SELECT DISTINCT state
| FROM customers
  
```

### The WHERE Clause

```

| SELECT *
| FROM customers
| WHERE points > 3000
  
```

$\geq$   
 $\leq$   
 $=$   
 $!=$   
 $<$   
 $>$

```
SELECT *  
FROM Customers  
WHERE state != 'VA' = WHERE state <> 'VA'
```

```
SELECT *  
FROM Customers  
WHERE birthdate > '1990-01-01'  
↓ year    ↓ Month    ↓ date
```

The AND, OR and NOT operators

```
SELECT *  
FROM Customers  
WHERE birth-date > '1990-01-01' OR  
(points > 1000 AND state = 'VA')
```

```
SELECT *  
FROM Customers  
WHERE NOT (birthdate > '1990-01-01' OR  
points > 1000)
```

equivalent

```
WHERE birth-date <= '1990-01-01' AND  
points <= 1000
```

```
SELECT *  
FROM order-items  
WHERE order-id = 6 AND  
unit-price * quantity > 30
```

## The -IN Operator

**equivalent**

```
SELECT *  
FROM Customers  
WHERE state = 'VA' OR state = 'GA' OR state = 'FL'
```

```
SELECT *  
FROM Customers  
WHERE state IN ('VA', 'FL', 'GA')
```

## The BETWEEN Operator

**equivalent**

```
SELECT *  
FROM Customers  
WHERE points >= 1000 AND points <= 3000
```

**equivalent**

```
SELECT *  
FROM Customers  
WHERE points BETWEEN 1000 AND 3000
```

## The LIKE Operator

```
SELECT *  
FROM Customers  
WHERE last_name LIKE 'b%'
```

WHERE last\_name LIKE 'brush%'  
or b%  
or %b%

b or B has to somewhere in  
last name, doesn't matter  
whether its in starting,  
middle or end

%.Y  
Last name ends with 'Y' or 'y'

Starts with b or B  
followed by anything  
or any number of  
characters

WHERE last-name LIKE 'b---y' → last name is exactly 6 characters ending with y or Y

" " " b---y → last name starts with 'b' or 'B' followed by 4 characters, ending with y only.

## The REGEXP Operator

equivalent → WHERE last-name LIKE 'field'.  
WHERE last-name REGEXP 'field'

SELECT \*

FROM customers

WHERE last-name REGEXP '^field'

→ last name starts with field (Not Case Sensitive)

| WHERE last-name REGEXP '\$field\$'

→ last name ends with field

| WHERE last-name REGEXP 'field|mac|rose'

→ last name which have field, mac or rose

| WHERE last-name REGEXP '^field|mac|rose'

→ last name should either starts with word field or it should have word mac in it or it should have the word rose

| WHERE last-name REGEXP 'field\$|mac|rose'

| WHERE last-name REGEXP '[gim]e'

→ customer who have ge or im or me in their last-name

REGEXP 'e[fmgd]'  
REGEXP '[a-h]e'

# The IS NULL Operator

```
| SELECT *  
| FROM customers  
| WHERE phone IS NULL
```

| Where phone IS NOT NULL

# The ORDER BY Clause

```
| SELECT *  
| FROM customers  
| ORDER BY first-name DESC
```

| ORDER BY state, first-name

Justly, sort the customer by state  
and within state sort by first-name

```
| SELECT first-name, last-name  
| FROM customers  
| ORDER BY birth-date
```

# The LIMIT Clause (will always comes at end)

```
| SELECT *  
| FROM customers  
| LIMIT 3
```

```
| SELECT *  
| FROM customers  
| LIMIT 6,3
```

It tells my  
SQL to skip  
first 6 records

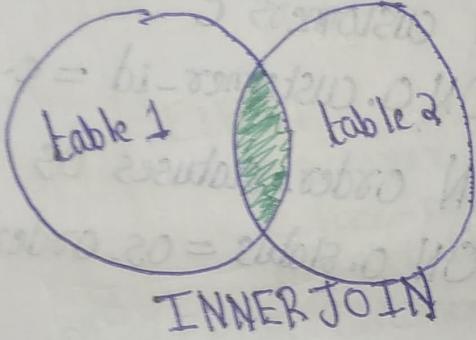
Then pick  
3 records

# Retrieving Data From Multiple Tables

## Inner Joins

JOIN by default means INNER JOIN

`SELECT order_id, o.customer_id, first_name, last_name  
FROM orders o  
JOIN customers c  
ON o.customer_id = c.customer_id`



these are unique column names among both the columns so no need to mention from which column they belong.  
customer\_id is common among both the table which causes MySQL eventually leads to error and so we need to specify the column name.  
only return the records that matches this condition

## Joining Across Databases

`SELECT *  
FROM order_items oi  
JOIN sql-inventory.products p  
ON oi.product_id = p.product_id`

## Self Joins

`SELECT *  
FROM employees e  
JOIN employees m  
ON e.reports_to = m.employee_id`

`SELECT e.employee_id,  
e.first_name,  
m.first_name AS manager,  
FROM employees e  
JOIN employees m  
ON e.reports_to = m.employee_id`

# Joining Multiple Tables

SELECT

```
O. order-id,  
O. order-date,  
C. first-name,  
C. last-name,  
OS.name AS status  
FROM orders O  
JOIN customers C  
ON O.customer-id = C.customer-id  
JOIN order-statuses OS  
ON O.status = OS.order-status-id
```

## Compound Join Conditions

SELECT

```
FROM order-items oi  
JOIN order-items oim  
ON oi.order-id = oim.order-id  
AND oi.product-id = oim.product-id
```

## Implicit Join Syntax

SELECT \*

```
FROM orders O  
JOIN customers C  
ON O.customer-id = C.customer-id
```

equivalent

## Implicit Join Syntax

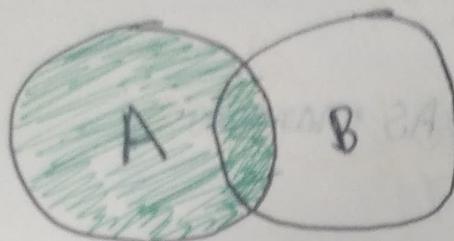
SELECT \*

```
FROM orders O, customers C  
WHERE O.customer-id = C.customer-id
```

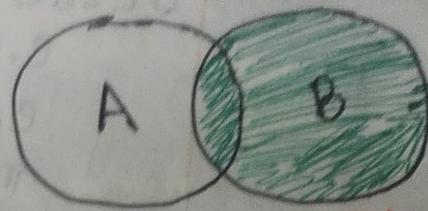
Not recommended

## Outer Joins

↳ LEFT JOIN ✓  
↳ RIGHT JOIN



LEFT JOIN



RIGHT JOIN

Left Outer Join  
Outer is optional  
Left

all the records  
from the customers  
are returned  
whether this  
condition is true  
or not

SELECT

c.customer\_id,  
c.first\_name,  
o.order\_id

FROM customers c

LEFT JOIN orders o

ON c.customer\_id = o.customer\_id

ORDER BY c.customer\_id

SELECT

c.customer\_id,  
c.first\_name,  
o.order\_id

FROM customers c

RIGHT JOIN orders o

ON c.customer\_id = o.customer\_id

ORDER BY c.customer\_id

## SELF OUTER JOIN

USE SQL-hr;

SELECT

e.employee-id,

e.first-name,

m.first-name

AS manager

FROM employee e

LEFT JOIN employee m

ON e.reports-to = m.employee-id

## The USING Clause

JOIN customers c

ON o.customer-id = c.customer-id

equivalent

JOIN customers c  
USING(customer-id)

## NATURAL JOIN

Not recommended

SELECT \*

FROM orders o

NATURAL JOIN customers c

we don't have to explicitly define column names, database will look into the table, and join them based on the common columns

## CROSS JOIN

SELECT \*  
FROM customers c  
CROSS JOIN products p

Every record in customers table will be combined with every record in the products table

equivalent

SELECT \*  
FROM customers c, products p

## UNIONS

SELECT  
order\_id,  
order\_date,  
'Active' AS status  
FROM orders  
WHERE order\_date >= '2019-01-01'

UNION

SELECT  
order\_id,  
order\_date,  
'Archived' AS status  
FROM orders  
WHERE order\_date < '2019-01-01'

# Inserting, Updating, and Deleting Data

SQL

## Column Attributes

Primary Key

Not Null

Auto Increment

Default

SELECT \*

FROM customers

CROSS JOIN

INNER JOIN

LEFT JOIN

RIGHT JOIN

## Inserting a Row

INSERT INTO customers  
VALUES (

DEFAULT,

'John',

'Smith',

'1990-01-01',

NULL,

'address',

'city',

'CA',

DEFAULT)

10- | INSERT INTO customers (

first-name,

last-name,

birth-date,

address,

city,

state)

VALUES (

'Sunny',

'Singh',

'2002-01-17',

'address1',

'city',

'Delhi')

## Inserting Multiple Rows -

```
INSERT INTO shippers (name)  
VALUES ('Shipper1'),  
('Shipper2'),  
('Shipper3')
```

## Inserting Hierarchical Rows -

```
INSERT INTO orders (customer_id, order_date, status)  
VALUES (1, '2019-01-02', 1);
```

LAST\_INSERT\_ID()  
→ Here id that my query generates when we insert new row

```
INSERT INTO order_items  
VALUES (LAST_INSERT_ID(), 1, 1, 2.95),  
(LAST_INSERT_ID(), 2, 1, 3.95)
```

## Creating a Copy of a Table

```
CREATE TABLE orders_archived AS  
SELECT * FROM orders → Subquery
```

## Updating a Single Row

```
Update invoices  
SET payment_total = 10, payment_date =  
'2019-03-01'  
WHERE invoice_id = 1
```

## Updating Multiple Rows

```
UPDATE invoices  
SET  
payment_total = invoice_total * 0.5,  
payment_date = due_date  
WHERE client_id = 3
```

# Using Subqueries in Updates

```
UPDATE invoices
SET
    payment_total = invoice_total * 0.5,
    payment_date = due_date
WHERE client_id =
    (SELECT client_id
     FROM clients
     WHERE name = 'Myworks')
```

If the Subquery returns multiple records we cannot use (=) sign anymore, so we have to replace it with IN operator

```
WHERE client_id IN
    (SELECT client_id
     FROM clients
     WHERE state IN ('CA', 'NY'))
```

## Deleting Rows

```
DELETE FROM invoices
```

```
WHERE invoice_id = 1
```

```
DELETE FROM invoices
WHERE client_id = (
```

```
SELECT *
     FROM clients
```

```
WHERE name = 'Myworks'
```

## Summarizing Data

Aggregate Functions - mysql comes with a bunch of built-in functions, and some of the are called aggregate functions, because they take a series of values and aggregate them to produce a single value.

**MAX()**  
**MIN()**  
**AVG()**  
**SUM()**  
**COUNT()**

Count() function  
returns Count of  
records with non-null  
payment\_dates

```
SELECT MAX(invoice-total) AS highest,  
       MIN(invoice-total) AS lowest,  
       AVG(invoice-total) AS average,  
       SUM(invoice-total) AS total  
       COUNT(invoice-total) AS number-of-invoices  
       COUNT(payment-date) AS count-of-payments  
       COUNT(*) AS total-records  
       COUNT(DISTINCT client-id) AS total-records  
FROM invoices
```

## The GROUP BY clause

Group By is used to group certain records of result set based on specified column. Suppose, if the specified column has duplicate values. Then those values are combined or grouped and according to this grouping other column values are also grouped. Group By is used in conjunction with aggregate functions.

```
SELECT client_id, SUM(invoice_total) AS total_sales
FROM invoices
WHERE invoice_date >= '2019-07-01'
GROUP BY client_id
ORDER BY total_sales DESC
```

```
SELECT state, city, SUM(invoice_total) AS total_sales
```

FROM invoices  
JOIN clients USING (client\_id)  
Group BY state, city

# The HAVING Clause

~~SELECT client\_id, SUM(invoice\_total) AS total\_sales  
FROM invoices  
WHERE total\_sales > 500  
GROUP BY client\_id  
HAVING total\_sales > 500~~

we can't do this because at this point we haven't grouped our data

we use Having clause to filter data after we GROUPBy rows

~~SELECT c.customer\_id, c.first\_name, c.last\_name, SUM(oi.quantity \* oi.unit\_price) AS amount\_spent  
FROM customers c  
JOIN orders o  
USING(customer\_id)  
JOIN order\_items oi  
USING(order\_id)  
WHERE state = 'VA'  
GROUP BY c.customer\_id, c.first\_name, c.last\_name  
HAVING amount\_spent > 100~~

# The ROLLUP Operator

~~SELECT payment\_method, SUM(amount) AS total  
FROM payments p  
JOIN payment\_methods pm  
ON p.payment\_method = pm.payment\_method\_id  
GROUP BY payment\_method WITH ROLLUP~~

~~Waiting Complex Query~~

## Subqueries

-- Find products that are more expensive than Lettuce (id = 3)

```
SELECT *  
FROM products  
WHERE unit-price > (  
    SELECT unit-price  
    FROM products  
    WHERE product-id = 3  
)
```

-- Find employees having salary more than average

```
SELECT *  
FROM employees  
WHERE salary >  
     (SELECT AVG(salary)  
      FROM employees)
```

## The IN Operator

-- Find the products that have never been ordered

```
SELECT *  
FROM products  
WHERE product-id NOT IN (  
    SELECT DISTINCT product-id  
    FROM order-items  
)
```

-- Find clients without invoices

```
SELECT *  
FROM clients  
WHERE client-id NOT IN (  
    SELECT DISTINCT client-id  
    FROM invoices  
)
```

## Subqueries VS Joins

-- Find customers who have ordered lettuce (id = 3)

-- SELECT customer\_id, first\_name, last\_name

SELECT \* DISTINCT customer\_id, first\_name, last\_name  
FROM customers  
WHERE customer\_id IN (  
 SELECT o.customer\_id  
 FROM order\_items oi  
 JOIN orders o USING (order\_id)  
 WHERE product\_id = 3

equivalent

SELECT DISTINCT customer\_id, first\_name, last\_name  
FROM customers c  
JOIN orders o USING (customer\_id)  
JOIN order\_items oi USING (order\_id)  
WHERE oi.product\_id = 3

## The ALL Keyword

-- SELECT invoices larger than all invoices of client 3

SELECT \*  
FROM invoices  
WHERE invoice\_total > (  
 SELECT MAX(invoice\_total)  
 FROM invoices  
 WHERE client\_id = 3

equivalent

SELECT \*  
FROM invoices  
WHERE invoice\_total > ALL (  
 SELECT invoice\_total  
 FROM invoices  
 WHERE client\_id = 3

The ANY Keyword

-- Select clients with atleast two invoices

```
SELECT *  
FROM clients  
WHERE client-id = ANY (  
    SELECT client-id  
    FROM invoices  
    GROUP BY client-id  
    HAVING COUNT(*) >= 2  
)
```

## Correlated Subqueries

-- Select employees whose salary is  
-- above the average in their office

```
SELECT *  
FROM employees e  
WHERE salary > (  
    SELECT AVG(salary)  
    FROM employees  
    WHERE office-id = e.office-id)
```

-- Get invoices that are larger than the  
-- client's average invoice amount

```
SELECT *  
FROM invoices i  
WHERE invoice-total > (  
    SELECT AVG(invoice-total)  
    FROM invoices  
    WHERE client-id = i.client-id)
```

The EXISTS Operator

-- SELECT clients that have an invoice

```
SELECT *  
FROM clients  
WHERE client_id IN (  
    SELECT DISTINCT client_id  
    FROM invoices  
)
```

equivalent

```
SELECT *  
FROM clients c  
WHERE EXISTS (  
    SELECT client_id  
    FROM invoices  
    WHERE client_id = c.client_id  
)
```

Subqueries in the SELECT clause

```
SELECT  
    invoice_id,  
    invoice_total,  
    (SELECT AVG(invoice_total)  
     FROM invoices) AS invoice_average  
    invoice_total - (SELECT invoice_average) AS difference  
FROM invoices
```

## Subqueries in the FROM clause -

```
SELECT *  
FROM (
```

```
    SELECT
```

```
        client_id,
```

```
        name,
```

```
(SELECT SUM(invoice_total)
```

```
    FROM invoices
```

```
    WHERE client_id = client_id) AS total_sales
```

```
(SELECT AVG(invoice_total) FROM invoices) AS average
```

```
(SELECT total_sales - average) AS difference
```

```
    FROM clients C
```

```
) AS Sales_Summary
```

Whenever we use Subquery in the From clause we need to give the Subquery an Alias, it is required.

doesn't matter whether we use it or not

```
WHERE total_sales IS NOT NULL
```

Writing a Subquery in the From clause of the Select statement can make our main query more complex, there is a better way to solve this problem using Views.

# Essential MySQL Functions

## Numeric Functions

| SELECT ROUND(5.73)

6

| SELECT ROUND(5.73, 1)

5.7

| SELECT TRUNCATE(5.7582, 2)

5.75

| SELECT CEILING(5.2)

6

| SELECT FLOOR(5.7)

5

| SELECT ABS(-5.2)

5.2

| SELECT RAND() → Generate random floating point number b/w 0 & 1

## STRING Functions

| SELECT LENGTH('SKY')

| SELECT UPPER('SKY')

SKY

| SELECT LOWER('SKY')

sky

| SELECT LTRIM(' Sky')  
| Sky

| SELECT RTRIM ('SKY')  
| SKY

| SELECT TRIM (' sky ')  
| Sky

| SELECT LEFT ('Kindergarten', 4)  
| Kind

| SELECT RIGHT ('Kindergarten', 6)  
| garten

| SELECT SUBSTRING ('Kindergarten', 3, 5)

nderg      actions like first occurrence  
of characters or a sequence of characters

| SELECT LOCATE ('n', 'Kindergarten')  
|      if the character is not present then mysql will return 0  
|      what to replace by when to replace

| SELECT REPLACE ('Kindergarten', 'garten', 'garden')

Kindergarten

| SELECT CONCAT ('first', 'last')  
| first last

| USE sql-store;  
| SELECT  
|      CONCAT(first-name, ' ', last-name) AS  
|      full-name  
| FROM customers

## Date Functions

| SELECT NOW()

2021-07-23 13:53:49

| SELECT CURDATE()

2021-07-23

| SELECT CURTIME

13:55:55

| SELECT MONTH(NOW())

7

minute, second

| SELECT HOUR(NOW())

13

MONTHNAME

| SELECT DAYNAME(NOW())

Friday

| SELECT EXTRACT(DAY FROM NOW())

23

| SELECT \*

FROM orders

WHERE YEAR(order\_date) = YEAR(NOW())

## Formatting Dates and Times

| SELECT DATE\_FORMAT(NOW(), '%M %d %Y')

July 23rd 2021

| SELECT DATE\_FORMAT(NOW(), '%m %d %Y')

07 23 21

| SELECT DATE\_FORMAT(NOW(), '%d %m %Y')

07 23 21

| SELECT TIME\_FORMAT(NOW(), '%H:%i:%s')

14:19 PM

14:19 PM

# Calculating Dates and Times

| SELECT DATE\_ADD(NOW(), INTERVAL 1 DAY) → YEAR  
| 2020-07-25 09:30:13 → return tomorrow's date and time

| SELECT DATE\_ADD(NOW(), INTERVAL -1 YEAR) → returns previous year

OR

| SELECT DATE\_SUB(NOW(), INTERVAL 1 YEAR)

2020-07-24 09:36:15

| SELECT DATEDIFF('2021-09-08 09:00', '2021-07-07 17:00') → return difference in no. of days  
| 63 → time doesn't taken into consideration

| SELECT TIME\_TO\_SEC('09:00') - TIME\_TO\_SEC('09:02')

| -120

## The IFNULL and COALESCE Functions

| SELECT  
| order\_id,  
| IFNULL(shipper\_id, 'NOT assigned') AS shipper  
| FROM orders  
| WHEN

| SELECT order\_id  
| WHEN  
| COALESCE(shipper\_id, comments, 'Not Assigned') AS shipper  
| FROM orders  
| ELSE  
| END AS

## The IF Function

SELECT

order\_id,  
order\_date,

IF(

YEAR(order\_date) = YEAR(NOW()),

'ACTIVE', → return this

'Archived') AS category; → else this

FROM orders

SELECT

product\_id,

name,

COUNT(\*) AS orders

IF(COUNT(\*) > 1, 'Many times', 'Once') AS frequency

FROM products

JOIN order\_items USING(product\_id)

GROUP BY product\_id, name

## The CASE Operator

SELECT

order\_id, order\_date

CASE

WHEN YEAR(order\_date) = YEAR(NOW()) THEN 'Active'

WHEN YEAR(order\_date) = YEAR(NOW()) - 1 THEN 'Last Year'

WHEN YEAR(order\_date) < YEAR(NOW()) - 1 THEN 'Archived'

ELSE 'Future'

END AS category

FROM orders

# Views

## Creating Views

This is our view and we can use it like table

CREATE VIEW sales-by-client AS

SELECT

c. client-id

c. name

SUM(invoice-total) AS total-sales

FROM clients c

JOIN invoices i USING(client-id)

GROUP BY client-id, name

SELECT \*

FROM sales-by-client

ORDER BY total-sales DESC

SELECT \*

FROM sales-by-client

WHERE total-sales > 500

SELECT \*

FROM sales-by-client

JOIN clients USING(client-id)

Altering or Dropping Views  
1-way → Once we create a view and realized that our query has a problem, so we need to go back and change our view

DROP VIEW sales-by-client

and execute the statement one more time

Other way →

CREATE OR REPLACE VIEW sales-by-client

GROUP BY client-id, name

This is not the ideal way to update views. As a best practice we should save our views in sql files and put them under source control

## Updatable Views

We can use <sup>views in</sup> SELECT Statement.

But we can also use INSERT, UPDATE and DELETE statement  
only under certain circumstances

If the view doesn't have →

→ DISTINCT

→ Aggregate Functions

→ GROUP BY / HAVING

(b) → UNION

CREATE OR REPLACE VIEW invoices-with-balance AS

SELECT

invoice\_id,  
number,  
client\_id,  
invoice\_total,  
payment\_total,  
invoice\_total - payment\_total AS balance,  
invoice\_date,  
due\_date,  
payment\_date

FROM invoices

WHERE (invoice\_total - payment\_total) > 0

DELETE FROM invoices-with-balance

WHERE invoice\_id = 1

UPDATE invoices-with-balance

SET due\_date = DATE\_ADD(due\_date, INTERVAL 2 DAY)

WHERE invoice\_id = 2

This is an  
updatable  
view

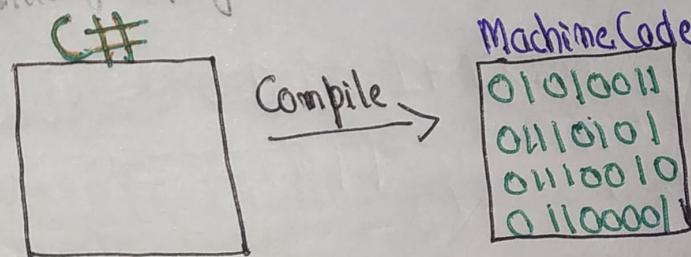
## Stored Procedures

What are stored procedures

Let's say we are building an application that has a database, where are we going to write these SQL statements. We are not going to write it in our application code, for couple of reasons. One reason is that, it will make our application code messy and hard to maintain.

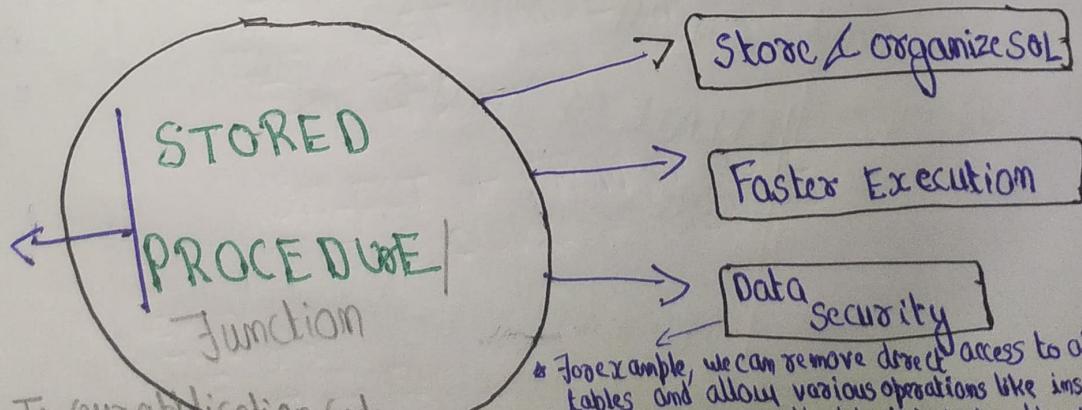
For example, if we are building our application in Java or C# or Python, we don't want to mix our Java code with SQL because the SQL code gets in the way and it makes our application code looks messy and hard to maintain.

Also some languages like C#, Java requires a compilation step, so if we write SQL query in our application code then find out you need to change one of your queries, you will have to recompile your application code for the changes to take effect potentially redeploy.



For all these reasons, we should take our SQL code out of our application code, and store it in database where it belongs, **But Where?**

Stored procedure is  
object that  
contains a block of SQL code  
in database  
and contains a block of SQL code



In our application code  
we can simply call these procedures  
to get or save the data

- \* For example, we can remove direct access to all the tables and allow various operations like inserting, updating and deleting the data to be performed by stored procedure.
- \* we can decide who can execute which stored procedure and this will limit what the user can do with data.
- \* we can prevent certain users from deleting our data.

## Creating a Stored Procedure -

```
| DELIMITER $$
| CREATE PROCEDURE get_clients()
| BEGIN
|   SELECT * FROM clients;
| END $$

DELIMITER ;
```

```
| DELIMITER $$
| CREATE PROCEDURE get_invoices_with_balance()
| BEGIN
|   SELECT *
|   FROM invoice-with-balance
|   WHERE balance > 0;
| END $$

DELIMITER ;
```

CALL sql-invoicing.get-invoices-with-balance();

## Dropping Stored Procedures -

Equivalent

```
| DROP PROCEDURE get_clients
| DROP PROCEDURE IF EXISTS get_clients
```

```
| DROP PROCEDURE IF EXISTS get_clients;
```

```
| DELIMITER $$
```

```
| CREATE PROCEDURE get_clients()
| BEGIN
```

```
|   SELECT * FROM clients;
```

```
| END $$
```

```
| DELIMITER ;
```

## Parameters

```
DROP PROCEDURE IF EXISTS get_clients_by_state;  
DELIMITER $$  
CREATE PROCEDURE get_clients_by_state  
(  
    state CHAR(2)  
)  
BEGIN  
    SELECT * FROM clients C  
    WHERE C.state = state;  
END  
DELIMITER ;
```

## Parameters with Default Values

```
DROP PROCEDURE IF EXISTS get_clients_by_state;  
DELIMITER $$  
CREATE PROCEDURE get_clients_by_state  
(  
    state CHAR(2)  
)  
BEGIN  
    IF state IS NULL THEN  
        SET state = 'CA';  
    END IF;  
    SELECT * FROM clients C  
    WHERE C.state = state;  
END $$  
DELIMITER ;
```

What if instead of returning the clients in CA, we want to return all clients...  
CREATE PROCEDURE

```
BEGIN  
IF state IS NULL THEN  
    SELECT * FROM clients;  
ELSE  
    SELECT * FROM clients c  
    WHERE c.state = state;  
END IF;  
END $$
```

equivalent

```
BEGIN  
SELECT * FROM clients c  
WHERE state = IFNULL(state, c.state);  
END $$
```

CREATE PROCEDURE get-payments

```
(  
    client_id INT,  
    payment_method_id TINYINT  
)
```

```
BEGIN  
SELECT *  
FROM payments p  
WHERE p.client_id = IFNULL(client_id, p.client_id) AND  
    p.payment_method =  
        IFNULL(payment_method_id, p.payment_method);
```

```
END
```

CALL get-payments (NULL, NULL)

CALL get-payments (5, NULL)

CALL get-payments (5, 2)

CALL get-payments (NULL, 2)

# Parameter Validation

CREATE PROCEDURE make\_payment

(  
    invoice\_id INT,  
    payment\_amount DECIMAL(9, 2),  
    payment\_date DATE  
)

BEGIN

    IF payment\_amount <= 0 THEN

        SIGNAL SQLSTATE '2203'

        SET MESSAGE\_TEXT = 'Invalid payment amount.'

    ENDIF

    UPDATE invoices i

        SET

        i.payment\_total = payment\_amount,  
        i.payment\_date = payment\_date

        WHERE i.invoice\_id = invoice\_id;

END

    SQLSTATE  
    error code

# Output Parameters

CREATE PROCEDURE get\_unpaid\_invoices\_for\_client

(  
    client\_id INT,  
    OUT invoices\_count INT,  
    OUT invoices\_total DECIMAL(9, 2)  
)

BEGIN

    SELECT COUNT(\*), SUM(invoice\_total)

        INTO invoices\_count, invoices\_total

        FROM invoices i

        WHERE i.client\_id = client\_id

        AND payment\_total = 0;

END

## Variables

-- User or session variables

SET @invoices\_count = 0

-- Local variable (are meaningful only in a stored procedure)  
(can be accessed as local variables and the input parameters inside)

CREATE PROCEDURE get-risk-factor()

BEGIN

DECLARE risk\_factor DECIMAL(9,2) DEFAULT 0;  
DECLARE invoices\_total DECIMAL(9,2);  
DECLARE invoices\_count INT;

SELECT COUNT(\*), SUM(invoice-total)  
INTO invoices\_count, invoices\_total  
FROM invoices;

SET risk\_factor = invoices\_total / invoices\_count \* 5;

SELECT risk\_factor;

END

Function (How to create your own function)

CREATE FUNCTION get-risk-factor-for-client

(client\_id INT)

) RETURNS INTEGER  
READS SQL DATA

BEGIN

SELECT COUNT(\*), SUM(invoice-total)  
INTO invoices\_count, invoices\_total  
FROM invoices  
WHERE i.client\_id = client\_id;

SET risk\_factor =

RETURN risk\_factor;

END

IF NULL(risk\_factor, 0)

DROP FUNCTION IF EXISTS get-risk-factor-for-client;

every mysql function  
should have at least 1 attribute

SELECT  
client\_id,  
name,  
get-risk-factor-for-client(client\_id)  
FROM clients

# Triggers and Events

## Triggers

- A block of SQL code that automatically gets executed before or after an insert, update or delete statement

DELIMITER \$\$

CREATE TRIGGER payments\_after\_insert  
AFTER INSERT ON payments  
FOR EACH ROW

BEGIN

UPDATE invoices

SET payment\_total = payment\_total + NEW.amount

WHERE invoice\_id = NEW.invoice\_id;

END \$\$

DELIMITER ;

INSERT INTO payments  
VALUES(DEFAULT, 5, 3, '2019-01-01', 10, 1)

DELIMITER \$\$

CREATE TRIGGER payments\_after\_delete  
AFTER DELETE ON payments  
FOR EACH ROW

BEGIN

UPDATE invoices

SET payment\_total = payment\_total - OLD.amount

WHERE invoice\_id = OLD.invoice\_id;

END \$\$

DELIMITER ;

DELETE FROM payments

WHERE payment\_id = 10

END

DELIMITER ;

## Viewing Triggers

SHOW TRIGGERS

SHOW TRIGGERS LIKE 'payments'.

## Dropping Triggers

DELIMITER ;

DROP TRIGGER IF EXISTS payments\_after\_insert;

## USING Triggers for Auditing

CREATE TABLE payments\_audit

client_id	INT	NOT NULL,
date	DATE	NOT NULL,
amount	DECIMAL(5,2)	NOT NULL,
action_type	VARCHAR(50)	NOT NULL,
action_date	DATETIME	NOT NULL

DELIMITER \$\$

DROP TRIGGER IF EXISTS payments\_after\_insert;

CREATE TRIGGER payments\_after\_insert

AFTER INSERT ON payments

FOR EACH ROW

BEGIN

UPDATE invoices

SET payment\_total = payment\_total + NEW.amount

WHERE invoice\_id = NEW.invoice\_id;

INSERT INTO payments\_audit

VALUES (NEW.client\_id, NEW.date, NEW.amount,  
'Insert', NOW());

END \$\$

DELIMITER ;

DELIMITER \$\$  
DROP TRIGGER IF EXISTS payments\_after\_delete;  
CREATE TRIGGER payments\_after\_delete  
AFTER DELETE ON payments  
FOR EACH ROW

BEGIN  
UPDATE invoices  
SET payment\_total = payment\_total - OLD.amount  
WHERE invoice\_id = OLD.invoice\_id;  
INSERT INTO payments\_audit  
VALUES (OLD.client\_id, OLD.date, OLD.amount, 'Delete',  
Now());  
END \$\$  
DELIMITER;

INSERT INTO payments  
VALUES (DEFAULT, 5, 3, '2019-01-01', 10, 1)

DELETE FROM payments  
WHERE payment\_id = 11

Events *(we use events to automate database maintenance task.)*

- A task (or block of SQL code) that gets executed according to a schedule

SHOW VARIABLES LIKE 'event%';

SET GLOBAL event\_scheduler = ON

CREATE EVENT *yearly\_delete\_stale\_audit\_rows*  
ON SCHEDULE  
-- AT '2019-05-01'  
EVERY 1 YEAR STARTS '2019-01-01' ENDS '2019-01-01'  
DO BEGIN  
DELETE FROM payments\_audit  
WHERE action\_date < Now() - INTERVAL 1 YEAR;  
END \$\$

If we want to execute on regular basis

In some organizations event schedules might be turned off to save system resources

Viewing, Dropping and Altering

DELIMITER ;  
SHOW EVENTS

DROP TRIGGER IF EXISTS  
CREATE TRIGGER IF NOT EXISTS

CREATE EVENT yearly;  
ALTER EVENT yearly

DROP EVENT IF EXISTS yearly;

ALTER EVENT yearly DISABLE;

SET @year = 19 - 10 = 19;

WHERE

INSERT INTO backup - audit

VALUES (@year);

END //

DELIMITER ;

INSERT INTO backup  
VALUES (YEAR());

DELETE FROM backup  
WHERE backup - 19 = 11

Events of year 11

A task of work is set to a schedule  
according to a sequence between

SHOW VARIABLES LIKE 'innodb\_flush\_log\_at\_trx\_commit';

NO = 0

SET GLOBAL innodb\_flush\_log\_at\_trx\_commit = 0;

CREATE EVENT daily

ON SCHEDULE

AT 10:00:00

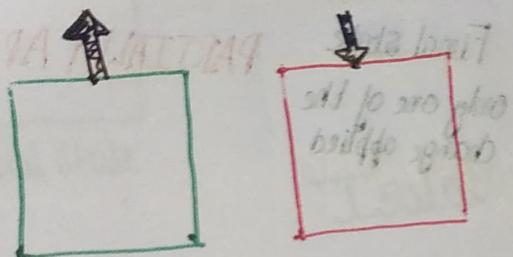
EVERY 1 DAY

# Transactions and Concurrency

Transactions: A group of SQL statements that represent a single unit of work.

All the statements should be completed successfully or the transaction will fail.

## BANK TRANSACTION



## UNIT OF WORK

DEBIT Operation  
CREDIT Operation

Either both these operations complete successfully or if the first operation succeeds, the second operation failed, we need to rollback or reverse the changes by the first operation.

The database transaction is exactly the same. We use transaction in situation where, we want to make multiple changes to the database, and we want all these transactions to succeed or fail together as a single unit.

INSERT INTO orders

...

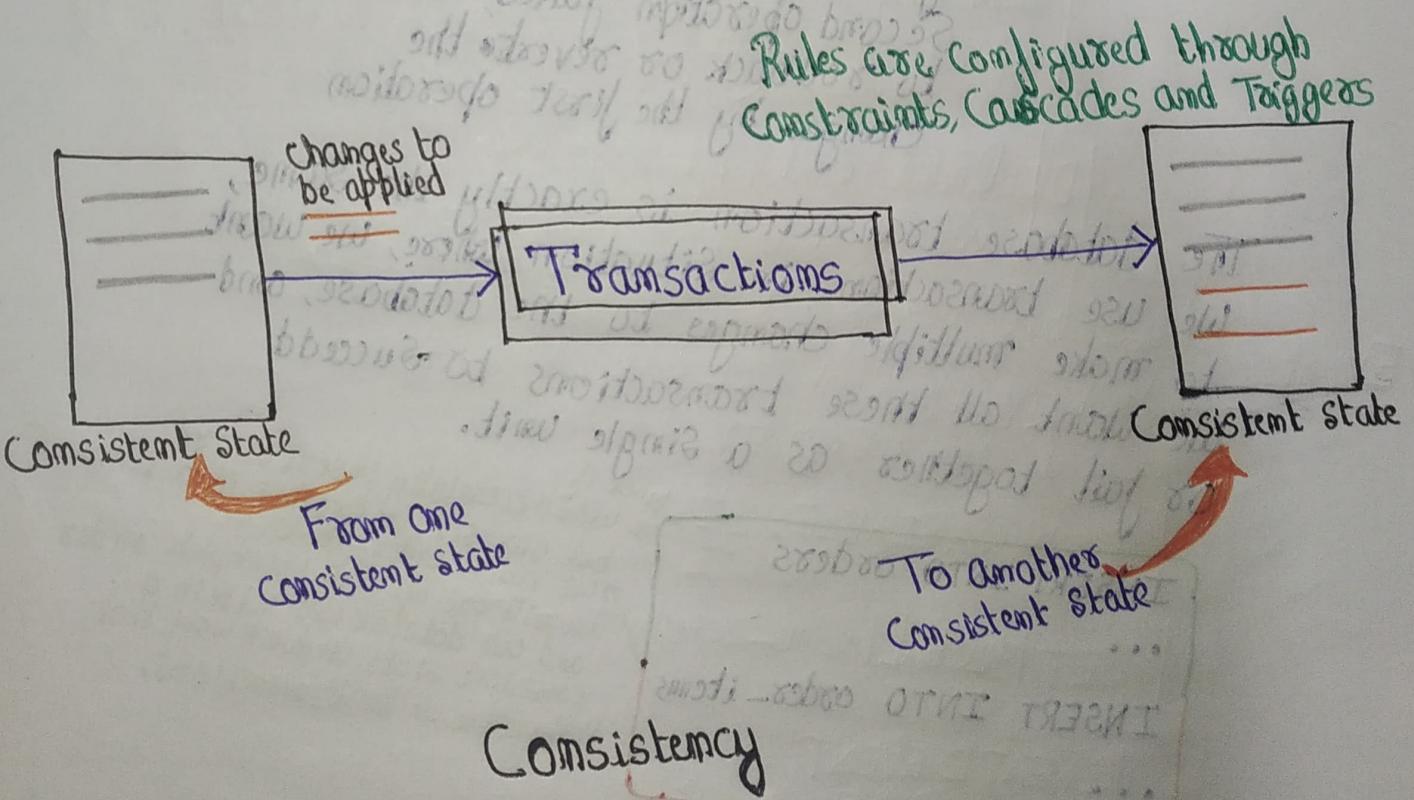
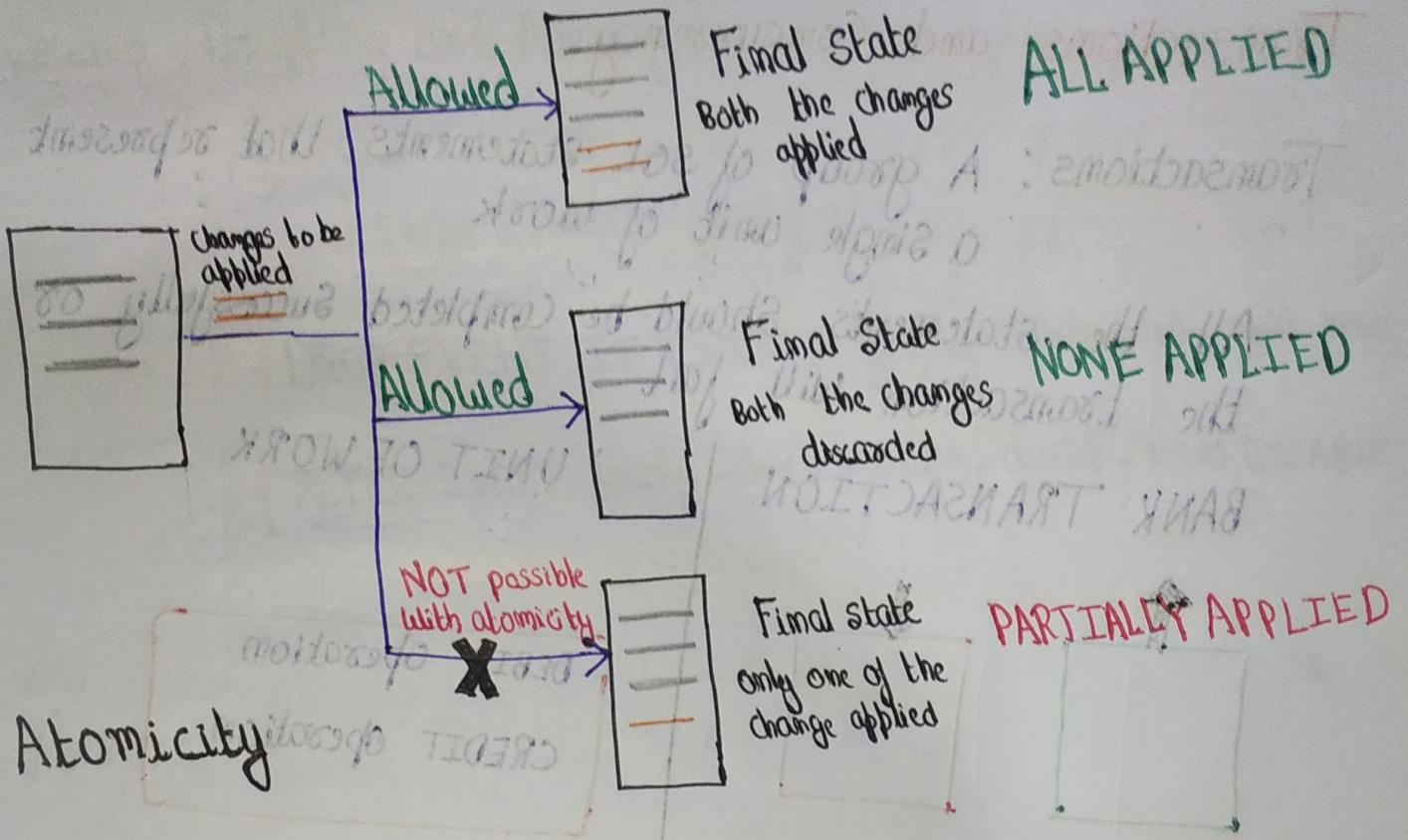
INSERT INTO order-items

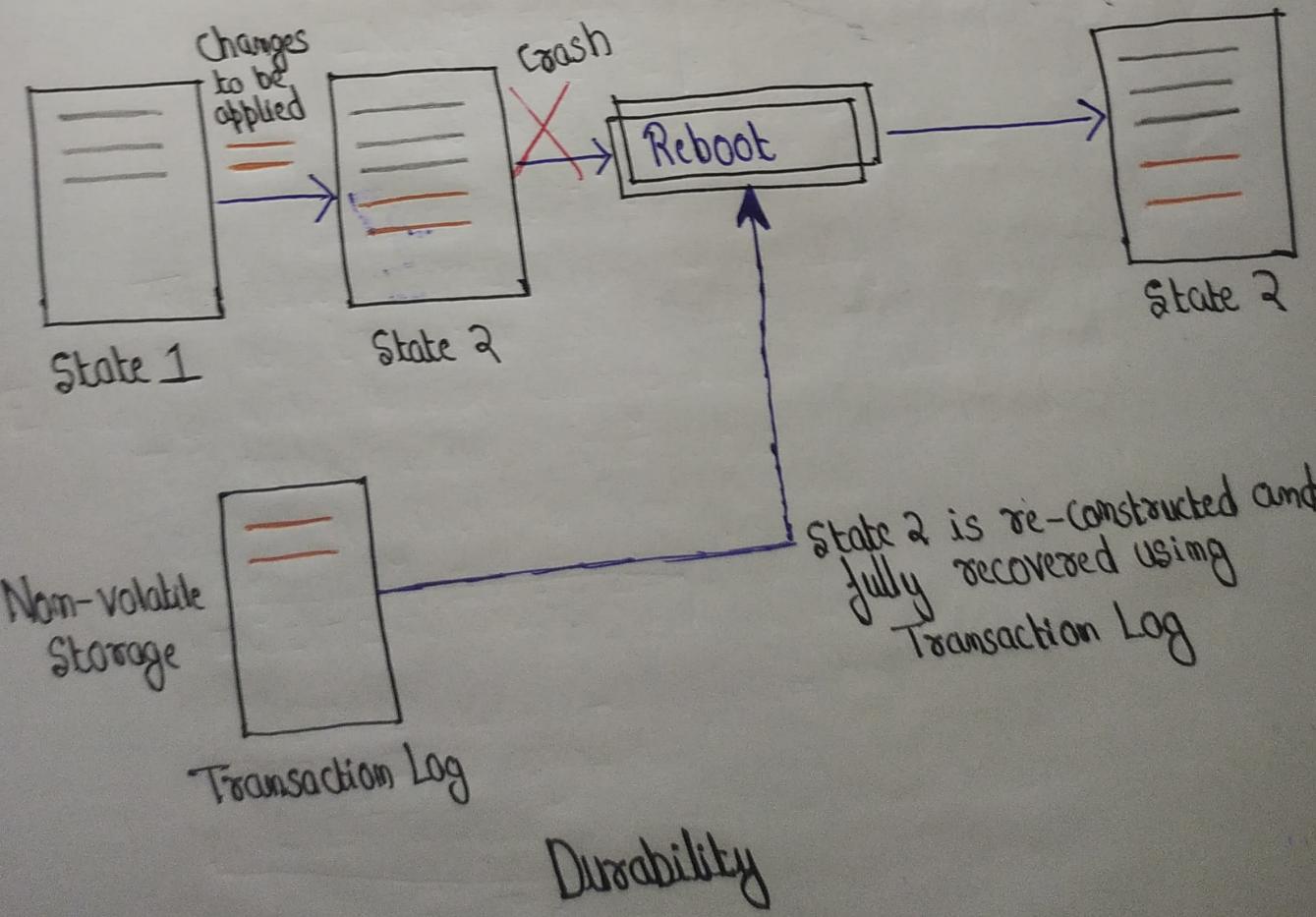
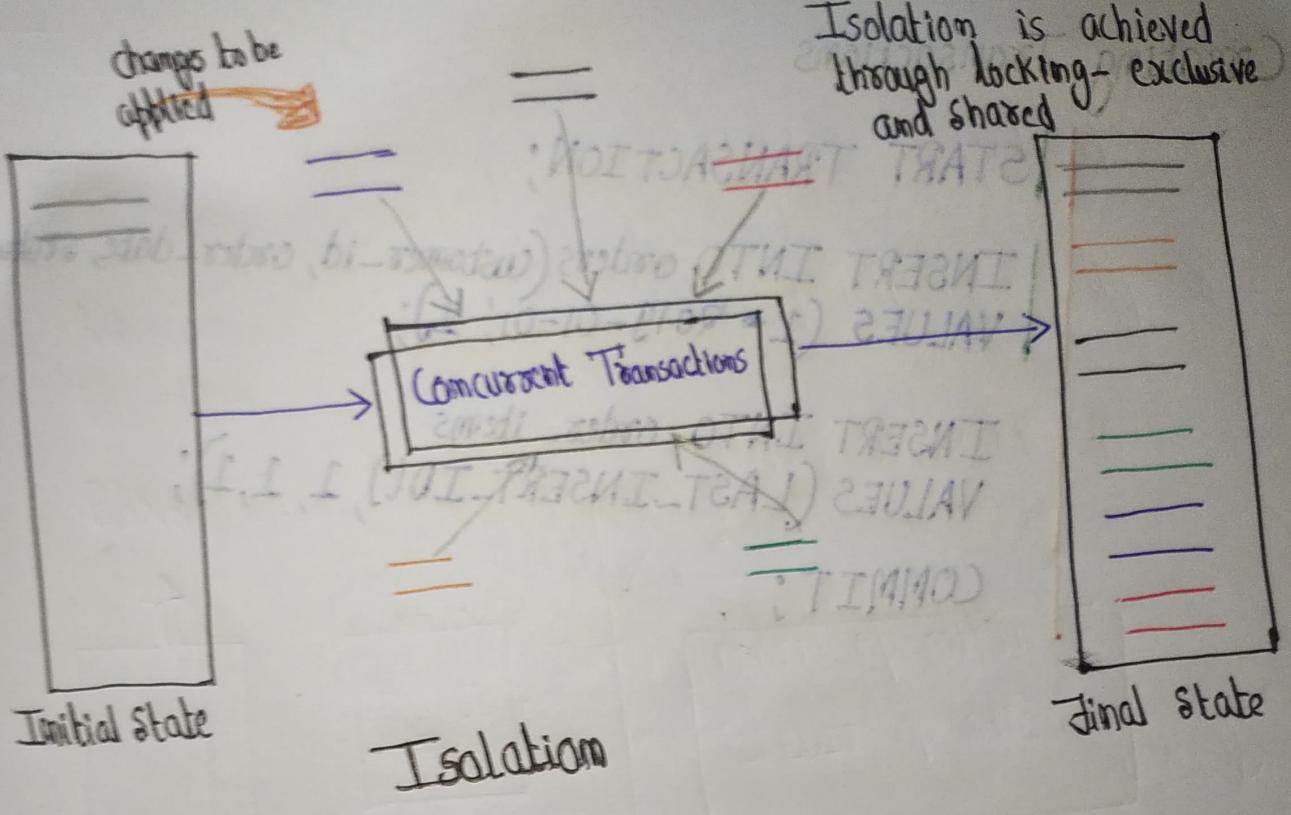
...

What if we successfully insert record but our server crashes at a time no we insert order items. we will end up with incomplete orders and our database will not be in a consistent state. we don't want that, that's where we use transactions.

## ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability





# Creating Transactions

START TRANSACTION;

INSERT INTO orders (customer\_id, order\_date, status)  
VALUES (1, '2019-01-01', 1);

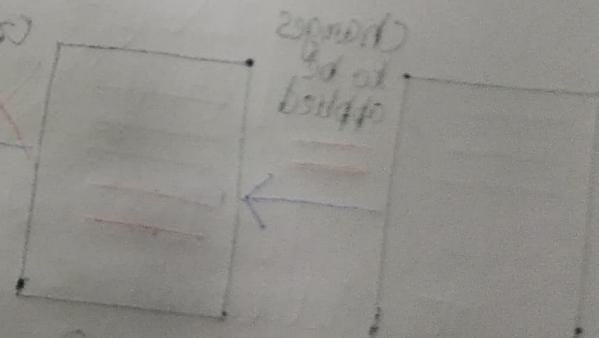
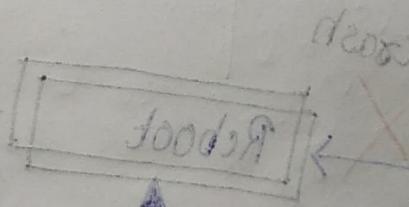
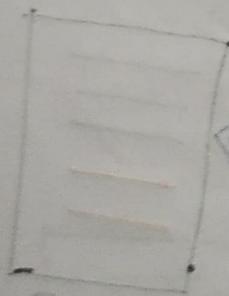
INSERT INTO order\_items  
VALUES (LAST\_INSERT\_ID(), 1, 1, 1);

COMMIT;

start transaction

insert into

start transaction



F\_start

L\_start

time between - 30 s to 2 s  
time between 10 min to 1 h  
for no transaction

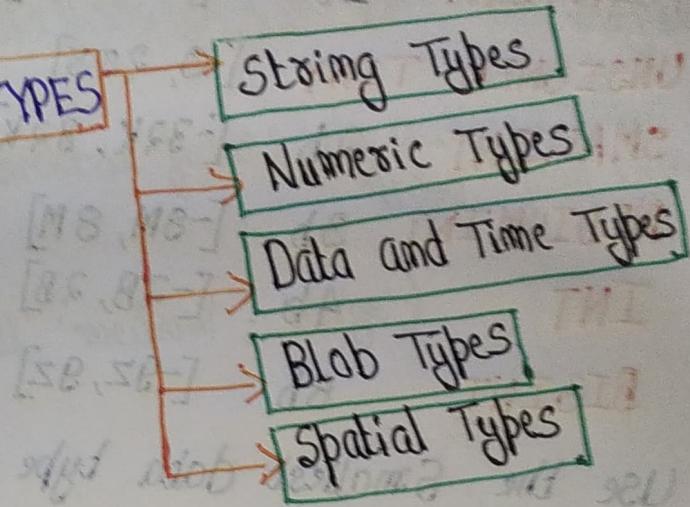
transaction

# Data Types

DATA TYPES

## Introduction

### MYSQL DATA TYPES



## String Types

CHAR(X)

VARCHAR(X)

fixed Length

max: 65,535 characters (v 64KB)

BE Consistent

VARCHAR(50)

for short strings

VARCHAR(355)

for medium-lengths

MEDIUMTEXT

max: 16MB

LONGTEXT

max: 4 GB

TINYTEXT

max: 255 bytes

TEXT

max: 64 KB

All these type supports international characters,

1 bytes

ENGLISH

2 bytes

European  
middle-eastern

3 bytes

Asian

# INTEGERS

TINYINT	1b	[-128, 127]
UNSIGNED TINY		[0, 255]
SMALLINT	2b	[-32K, 32K]
MEDIUMINT	3b	[-8M, 8M]
INT	4b	[-2B, 2B]
BIGINT	8b	[-9Z, 9Z]

Use the smallest data type that suits your needs.

## Fixed-point and Floating-point Types

DECIMAL(p, s)

DEC  
NUMERIC  
FIXED

FLOAT 4b

DOUBLE 8b

## BOOLEANS

BOOL  
BOOLEAN

UPDATE

SET is-published = TRUE # or FALSE

UPDATE

SET is-published = 1 # or 0

Same

moi/bubor/ln

edit pointe

1234567.89

TEXT MEDIUM

TEXT LONG

TEXT

## Enum and Set Types

NOTE

analogy

enum = radio buttons (only accepted values are those listed  
may only choose one)

SET = checkbox fields (only accepted values are those listed, may choose multiple)

- | ENUM ('small', 'medium', 'large')
- | SET (...)

## DATE and Time Types

DATE

TIME

DATETIME

TIMESTAMP

YEAR

8b

4b (up to 2038)

## BLOB Types

We use blob type to store large amount of binary data like images, videos, pdf's, wordfiles, pretty much any binary data.

TINYBLOB

255 b

BLOB

65 KB

MEDIUMBLOB

16 MB

LONGBLOB

4 GB

Generally speaking its better to keep your files out of your databases because relational databases are designed for working with structured relational data not binary data

Problems with  
storing Files in  
a Database

→ Increased database size

→ Slower backups

→ Performance problems

→ More code to read/write images