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
```

> >=
< <=
=
! =
<>

```
SELECT *
FROM Customers
WHERE State != 'va'
```
= `WHERE State <> 'va'`

```
SELECT *
FROM Customers
WHERE birthdate > '1990-01-01'
```
(1990 = year, 01 = Month, 01 = date)

The AND, OR and NOT Operator

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

```
SELECT *
FROM Customers
WHERE NOT (birth_date > '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

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

equivalent

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

The BETWEEN Operator

```
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%'
```

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

WHERE Last_name LIKE 'brush%'

" " '%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'

WHERE last_name LIKE '_____Y' → last_name is exactly 6 characters ending with y or Y

" " '_____Y' → last_name starts with 'b' or 'B' followed by 4 characters, ending with y or Y

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[fmq]'

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
```

firstly, Sort the customer by state and within state Sort by first_name

```
SELECT first_name, last_name
FROM customers
ORDER BY birth_date
```

```
SELECT *, quantity * unit_price AS total_price
FROM order_items
WHERE order_id = 2
ORDER BY total_price DESC
```

The LIMIT clause (Will always comes at end)

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

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

6 → offset, it tells my SQL to skip first 6 records

3 → then pick 3 records

Retrieving Data From Multiple Tables

Inner Joins

JOIN by default means INNER JOIN

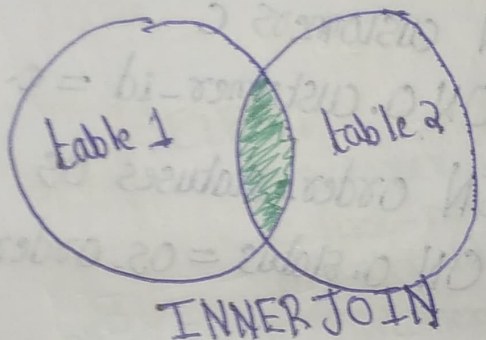
```
SELECT order_id, o.customer_id, first_name, last_name
FROM orders o
JOIN customers c
    ON o.customer_id = c.customer_id
```

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 tables which confuses mysql, eventually leads to error and so we need to specify the column name.

aliases

only return the records that matches this condition



INNER JOIN

Joining Across Databases

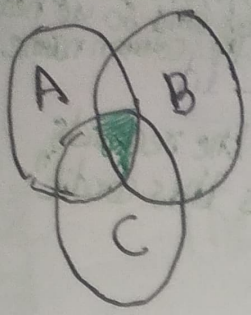
```
SELECT *
FROM order_items oi
JOIN sql_inventory.products p
    ON oi.product_id = p.product_id
```

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
```

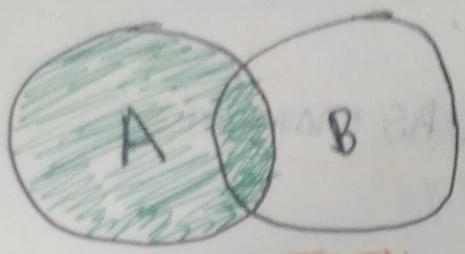
equivalent

```
-- Implicit Join Syntax
SELECT *
FROM orders o, customers c
WHERE o.customer_id = c.customer_id
```

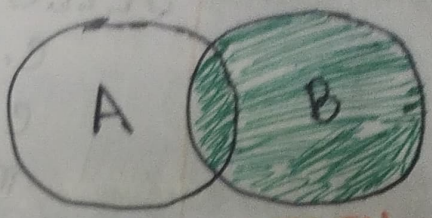
Not recommended

Outer Joins

- LEFT JOIN ✓
- RIGHT JOIN



LEFT JOIN



RIGHT JOIN

Left Outer Join
↓
outer is optional

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, ~~orders~~ 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

Column Attributes

- Primary Key
- Not Null
- Auto Increment
- Default

Inserting a Row

```
INSERT INTO customers
VALUES (
        DEFAULT,
        'John',
        'Smith',
        '1990-01-01',
        NULL,
        'address',
        'city',
        'CA',
        DEFAULT)
```

```
INSERT INTO customers (
        first_name,
        last_name,
        birth_date,
        address,
        city,
        state)
VALUES (
        'Suraj',
        'Singh',
        '2003-01-17',
        'address1',
        'city',
        'Maihar')
```

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);

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

LAST_INSERT_ID() → returns the id that mysql generates when we insert a new row

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

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 value and aggregate them to produce a single value.

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

```
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
FROM invoices
```

count(*) AS total_records

count(DISTINCT client_id) AS total_records

count() function returns count of records with non-null payment_dates

The GROUP BY clause

Group BY is used to group certain records 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 i
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 GROUP our rows

we can apply the condition on Having clause only if they are in Select clause

```
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
```

Writing 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
)
```

IN

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 = c.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
WHERE total_sales IS NOT NULL
```

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

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')
```
3

```
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
```

start position
length (optional)

1 2 3 ... 12

```
SELECT LOCATE('n', 'Kindergarten')
3
```

returns the first occurence of a character or a sequence of characters

if the character is not present then mysql will return 0

```
SELECT REPLACE('Kindergarten', 'garten', 'garden')
Kindergarden
```

what to replace

by what to replace

```
SELECT CONCAT('first', 'last')
firstlast
```

```
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

SELECT YEAR(NOW())
2021

SELECT HOUR(NOW())
13

SELECT DAYNAME(NOW())
Friday

SELECT EXTRACT(DAY FROM NOW())
23

SELECT *
FROM orders
WHERE YEAR(order_date) = YEAR(NOW())
```

minute, second

MONTHNAME

year, month, second

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(), '%m %D %y')
07 23rd 21

SELECT TIME_FORMAT(NOW(), '%H:%i %p')
14:19 PM
```

Calculating Dates and Times

```
SELECT DATE_ADD(NOW(), INTERVAL 1 DAY)
```
→ YEAR

2021-07-25 09:30:13 → return tomorrows 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')
```
time doesn't taken into consideration

63 → return difference in no. of days

```
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
```

```
SELECT
    order_id
    COALESCE(shipper_id, comments, 'Not Assigned') AS shipper
FROM orders
```

The IF Function

```
SELECT
    order_id,
    order_date,
    IF(
        YEAR(order_date) = YEAR(NOW()),
        'ACTIVE',
        'Archived') AS category
FROM orders
```

if this expression is true → return this; or else this

```
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,
    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
```

View behave like a virtual table, Views don't store data, our data is actually stored in tables, a View Just provide a View to the underlying tables

```
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
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
```

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

Updateable Views

We can use SELECT Statement. (views on views)

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
- UNION

This is an updateable view

```
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
```

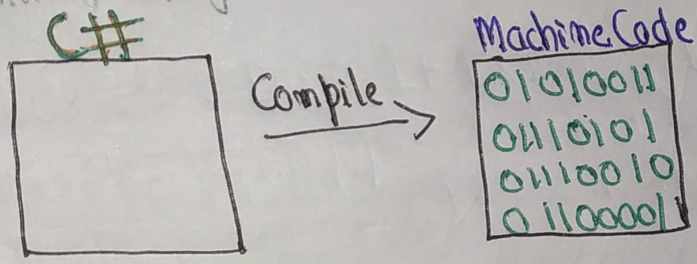
Stored Procedures

What are stored Procedures

Let's say we are building a 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 reason, One reason is that, it will make our application code messy and hard to maintain.

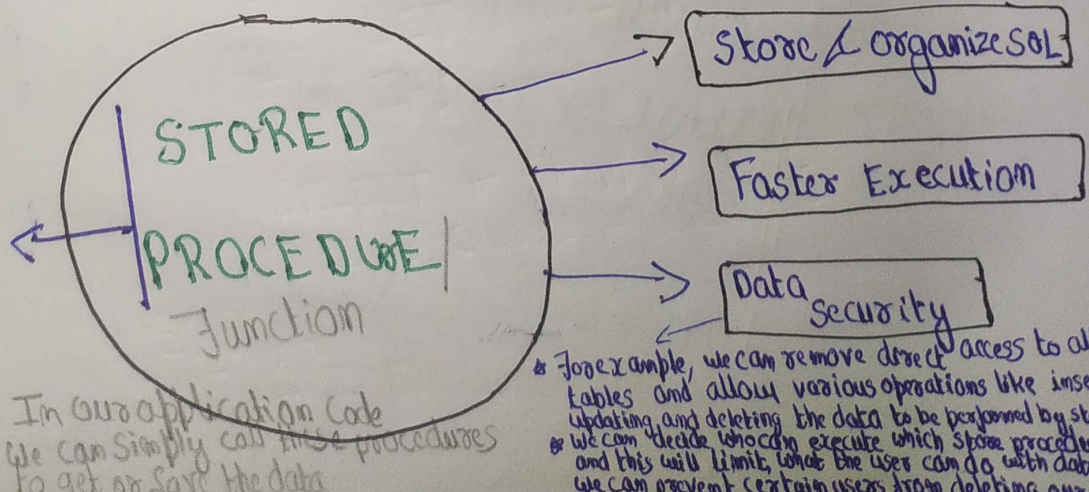
For example, if we are building our application in Java or C# or python, we don't want to mix our java code with Sql because the sql code gets in the way and it makes our application code looks messy and hard to read maintain

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 affect 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 a database object that contains a block of sql code



* 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 proc.
* We can decide who can execute which store procedure and this will limit, what the user can do with data. For ex, we can prevent certain users from deleting our data

Creating a Stored Procedure –

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

Gap

```
CALL get_clients()
```

```
DELIMITER $$
CREATE PROCEDURE get_invoices_with_balance()
BEGIN
    SELECT *
    FROM invoice_with_balance
    WHERE balance > 0;
END $$
DELIMITER ;
```

we have create the view earlier

```
CALL sql_invoicing.get_invoices_with_balance();
```

Dropping Stored Procedures –

```
DROP PROCEDURE get_clients
```

equivalent

```
DROP PROCEDURE IF EXISTS get_clients
```

```
DROP PROCEDURE IF EXISTS get_clients;
DELIMITER $$
CREATE PROCEDURE get_clients()
    SELECT * FROM clients;
BEGIN
END $$
DELIMITER ;
```

Parameters

```
DROP PROCEDURE IF EXISTS get_clients_by_state;

DELIMITTER $$
CREATE PROCEDURE get_clients_by_state
(
    state CHAR(2)
)
BEGIN
    SELECT * FROM clients c
    WHERE c.state = state;
END
DELIMITER ;
```

```
CALL get_clients_by_state ('CA')
```

Parameters with Default Values

```
DROP PROCEDURE IF EXISTS get_clients_by_state;
DELIMITTER $$
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...

```
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 c.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 '22003'
            SET MESSAGE_TEXT = 'Invalid payment amount';
    END IF;
    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 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)

- Can return only single value

```
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 i
    WHERE i.client_id = client_id;

    SET risk_factor =
    RETURN risk_factor;

END
```

every mysql function should have atleast 1 attribute

Attribute
- DETERMINISTIC
- READS SQL DATA
- MODIFIES SQL DATA

IFNULL(risk_factor, 0)

```
DROP FUNCTION IF EXISTS get_risk_factor_for_client;
```

```
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

```
DELIMITTER $$
CREATE TRIGGER payments_after_insert
    AFTER INSERT ON payments
    FOR EACH ROW
BEGIN
    UPDATE invoices
    SET payment_total = payment_total + NEW.amount
    WHERE invoice_id = NEW.invoice_id;
END $$
DELIMITER ;
```

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

```
DELIMITTER $$
CREATE TRIGGER payments_after_delete
    AFTER DELETE ON payments
    FOR EACH ROW
BEGIN
    UPDATE invoices
    SET payment_total = payment_total - OLD.amount
    WHERE invoice_id = OLD.invoice_id;
END $$
DELIMITTER ;
```

```
DELETE
FROM payments
WHERE payment_id = 10
```

Viewing Triggers

```
SHOW TRIGGERS
SHOW TRIGGERS LIKE 'payments%'
```

DROPping Triggers

```
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(9,2) 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 $$
DELIMITTER;
```

```
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%';
```

all the system variables

```
SET GLOBAL event_scheduler = ON
```

In some organization event-Schedulers might be turned off to save system resources

```
CREATE EVENT yearly_delete_stale_audit_rows
ON SCHEDULE
    -- AT '2019-05-01'
    EVERY 1 YEAR STARTS '2019-01-01' ENDS '2029-01-01'
DO BEGIN
    DELETE FROM payments_audit
    WHERE action_date < NOW() - INTERVAL 1 YEAR;
END $$
```

yearly → contraction

If we want to execute on regular basis

If we want to execute is only once, we will use AT keyword followed by dateTime value

optionally

Viewing, Dropping and Altering

```
SHOW EVENTS
```

```
SHOW EVENTS 'yearly%';
```

```
DROP EVENT IF EXISTS yearly_delete_stale_audit_rows;
```

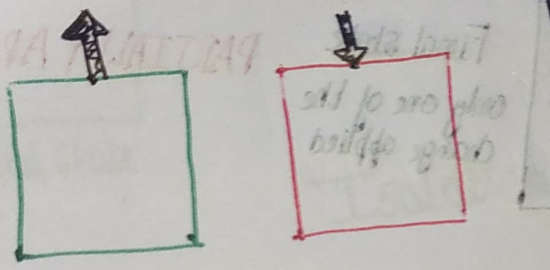
```
ALTER EVENT yearly_delete_stale_audit_rows DISABLE;
```

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

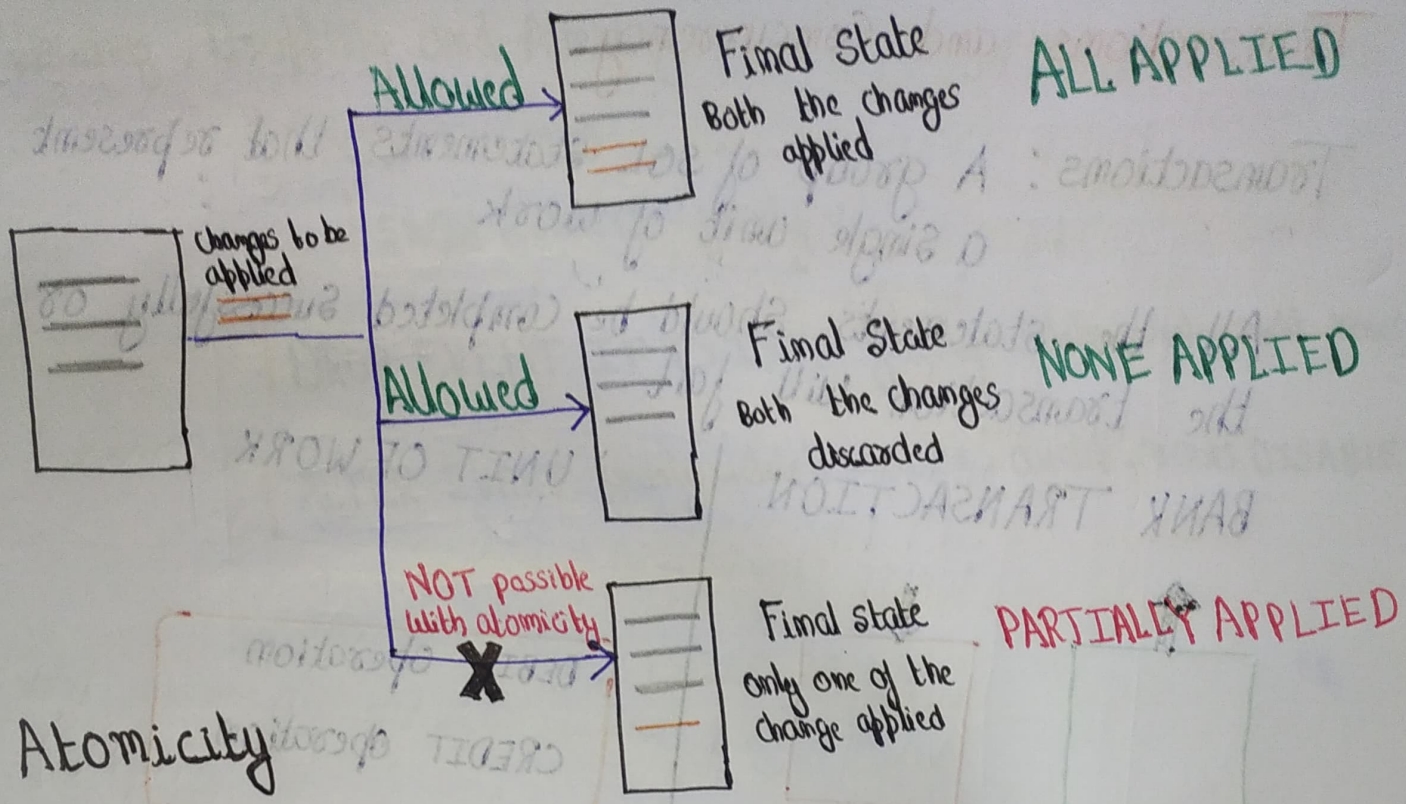
Either both these operations complete successfully or if the first operation succedd, the second operation failed, we need to rollback or reverse the changes by the first operation

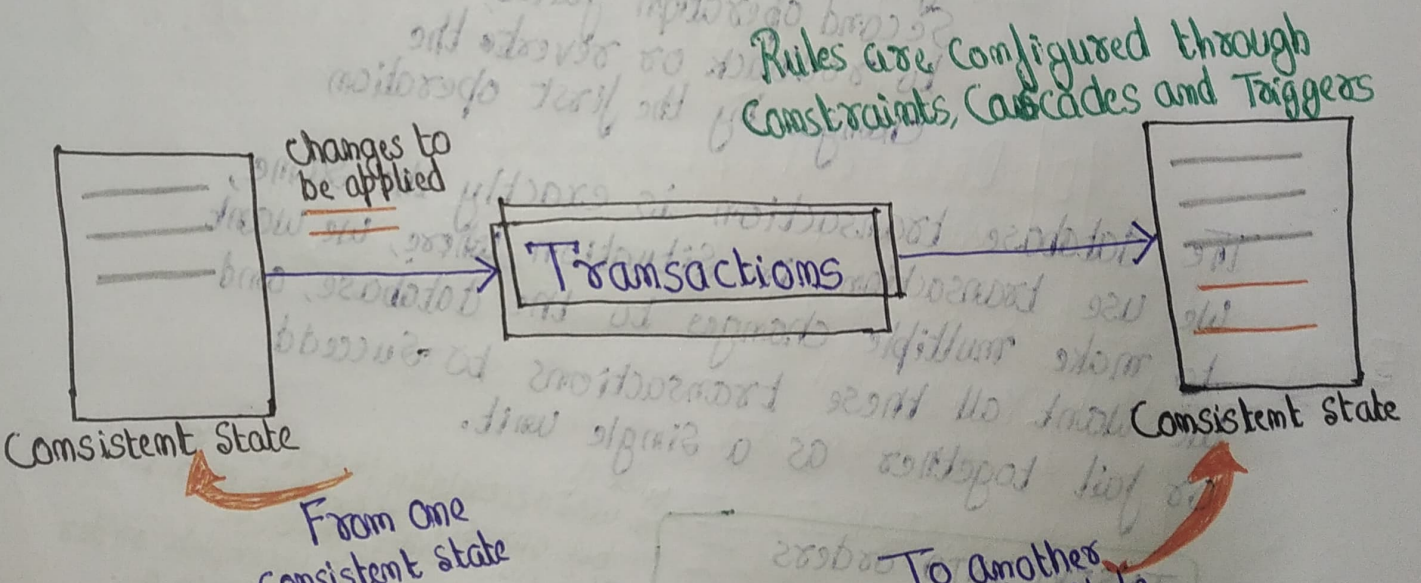
The database transaction is exactly the same, we use transaction in situation where, we want to make multiple changes to the database, and we want all these transactions to succedd or fail together as a single unit.

```
INSERT INTO orders
...
INSERT INTO order_items
...
```

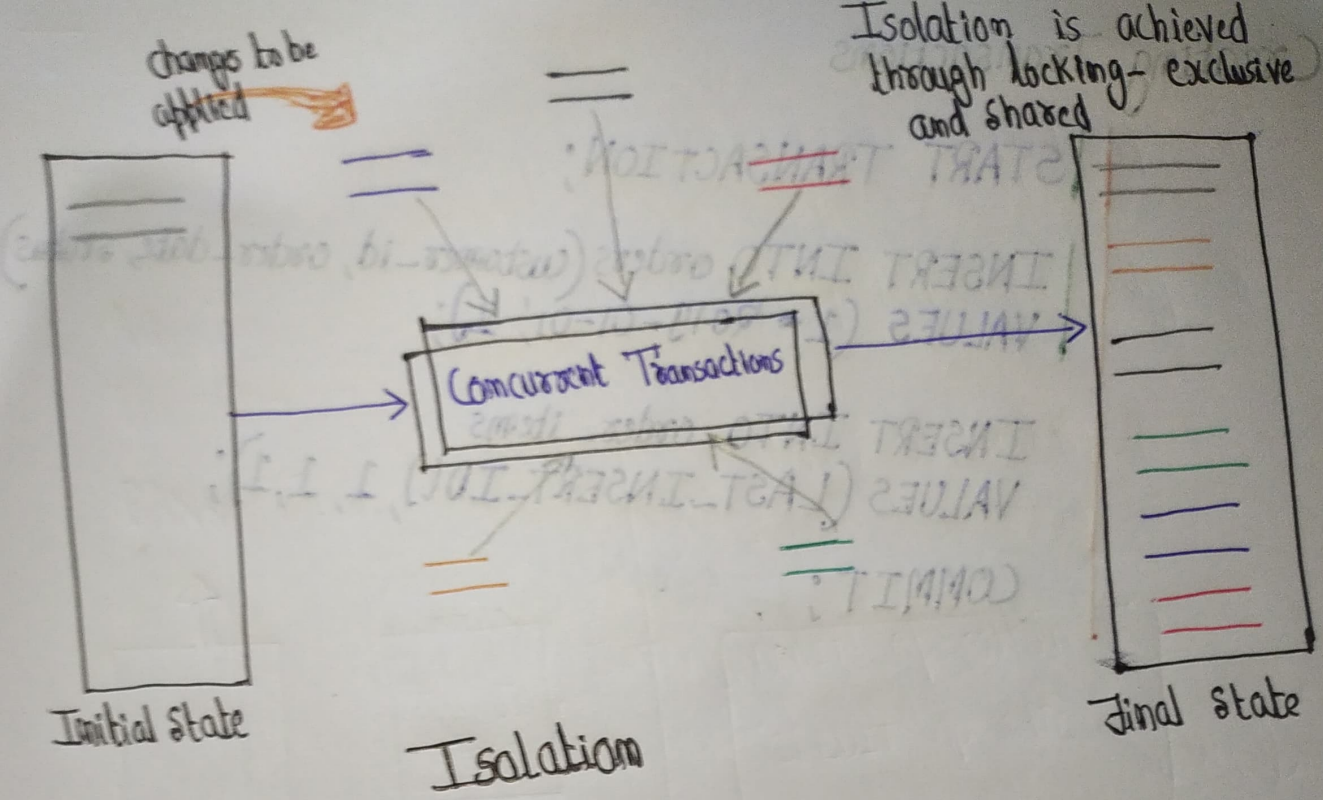
what if we successfully insert order record but our server crashes at a time we insert order_items. we will end up with incomplete order and our database will not be in a consistent state, we don't want that, that's where we use transactions.

ACID Properties
- Atomicity
- Consistency
- Isolation
- Durability

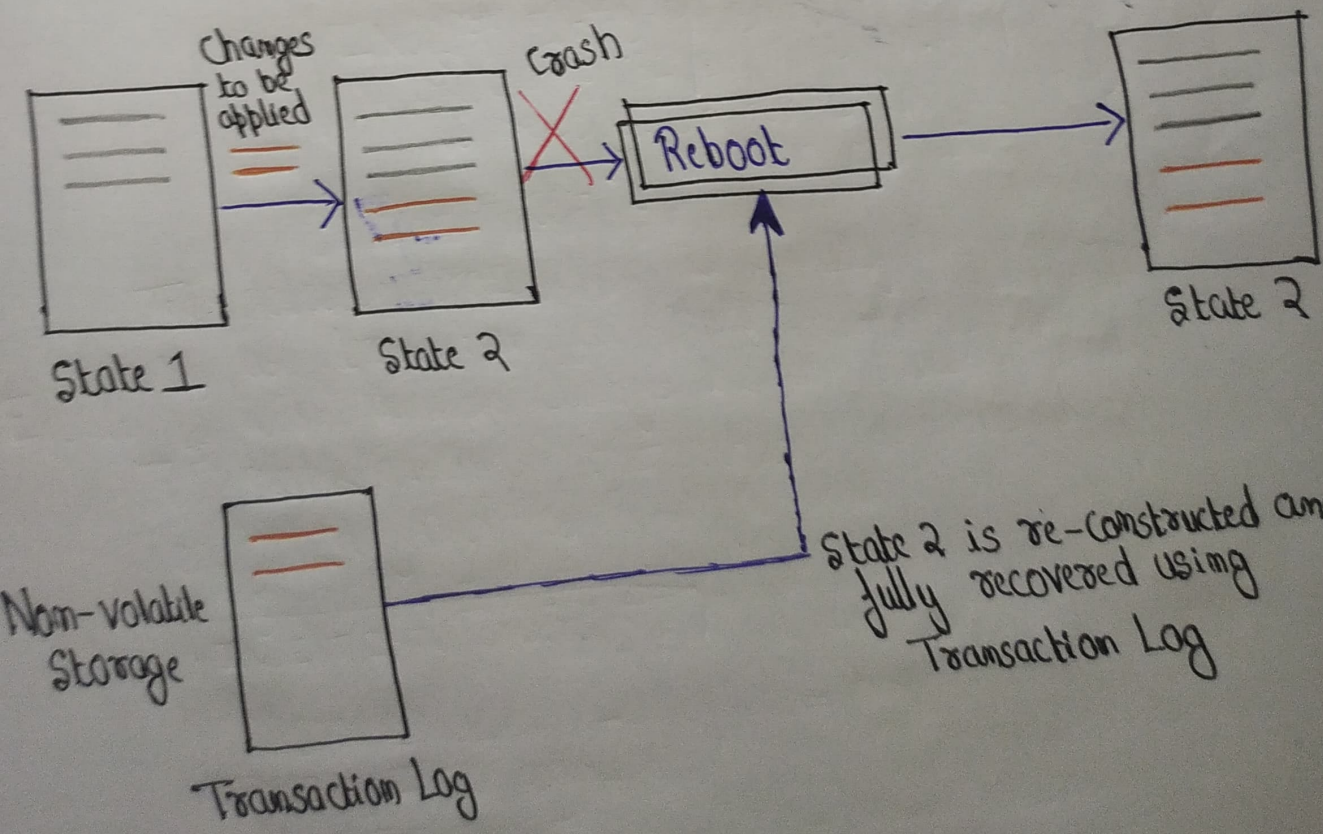




Consistency

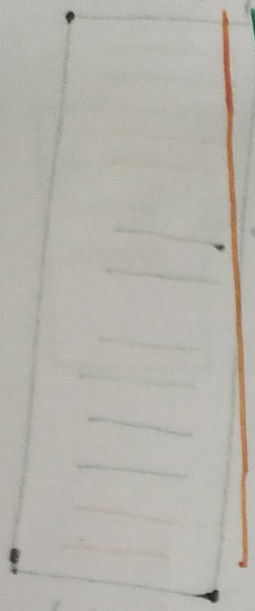


Isolation

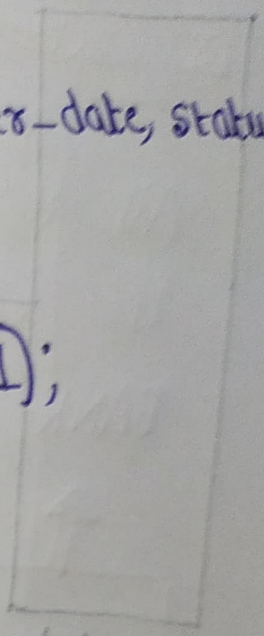


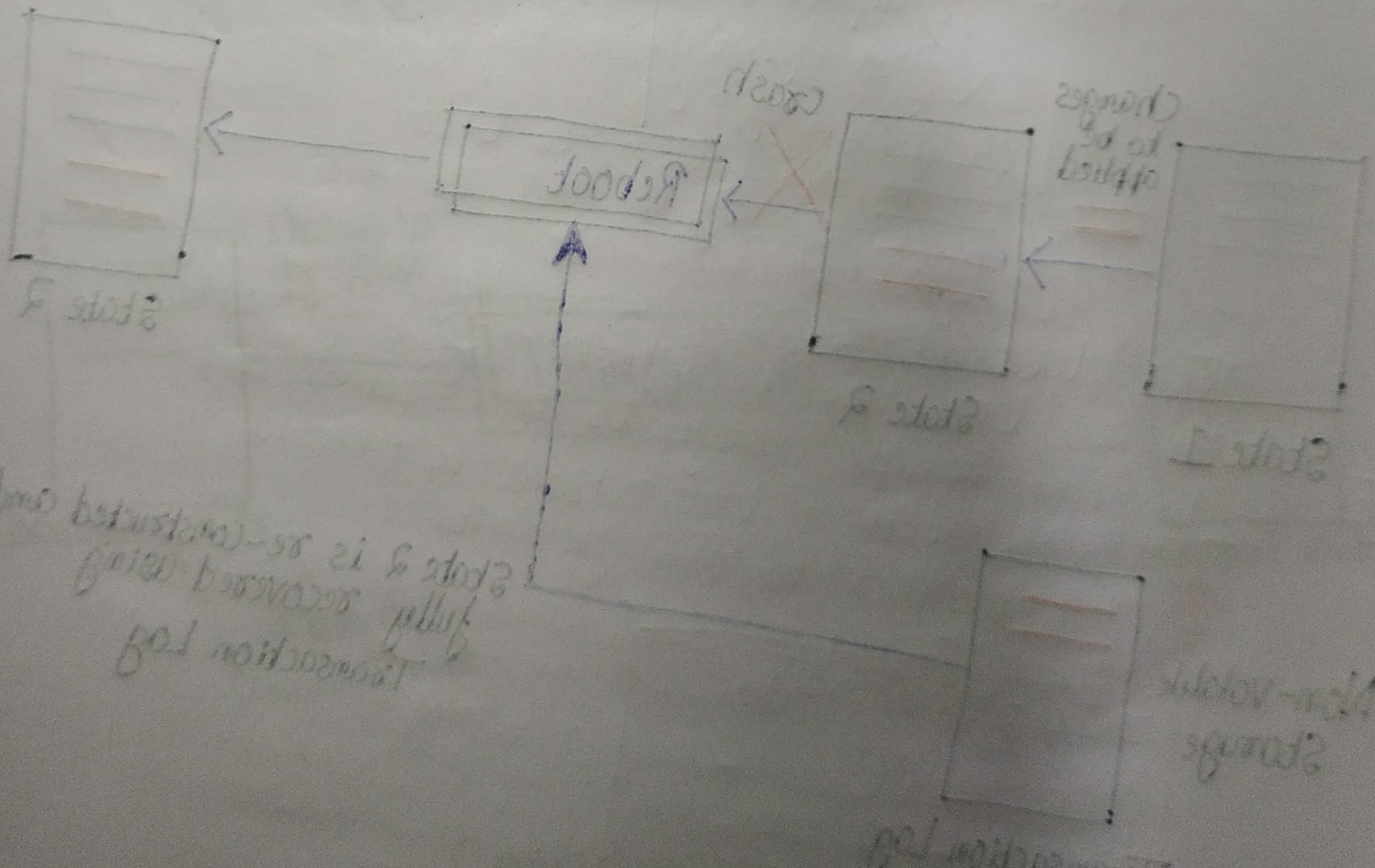
Durability

Creating Transactions



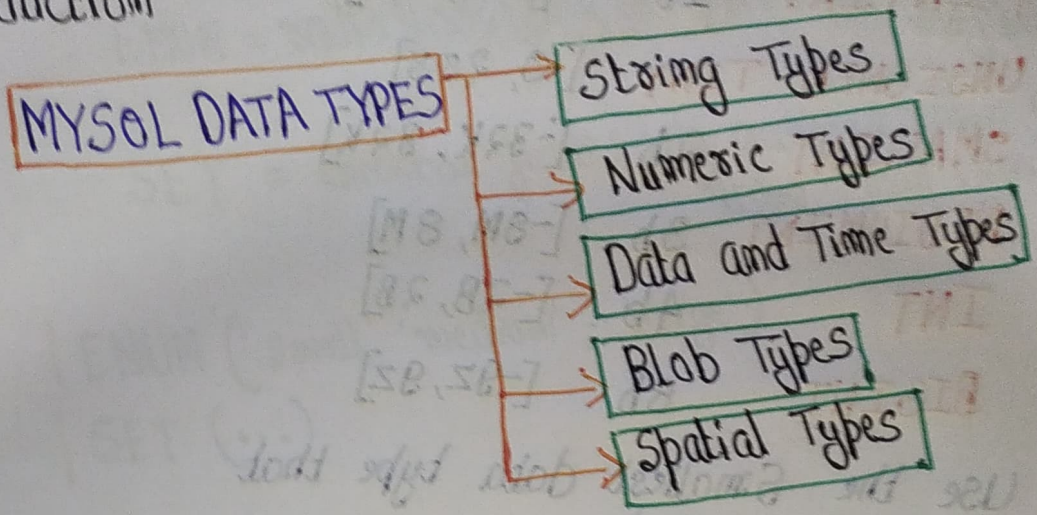
```
START TRANSACTION;

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;
```





Data Types

Introduction



String Types

CHAR(X) fixed Length
VARCHAR(X) max: 65,585 characters (≈64KB)

BE Consistent.

VARCHAR(50) for short strings
VARCHAR(255) for medium-length

MEDIUMTEXT max: 16MB
LONGTEXT max: 4GB

TINYTEXT max: 255 bytes
TEXT max: 64KB

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) => DECIMAL(9,2) => 1234567.89

Same:
- DEC
- NUMERIC
- FIXED

FLOAT	4b
DOUBLE	8b

BOOLEANS

BOOL
BOOLEAN

Internally it is represented as 1 and 0

```
UPDATE
SET is_published = TRUE # or FALSE
```

Same

```
UPDATE
SET is_published = 1 # or 0
```

Enum and Set Types

analogy

ENUM = radio fields (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	8b
TIMESTAMP	4b (up to 2038)
YEAR	

BLOB Types

We use blob type to store large amount of binary data like images, videos, pdf's, word files, 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