MySQL - RDBMS

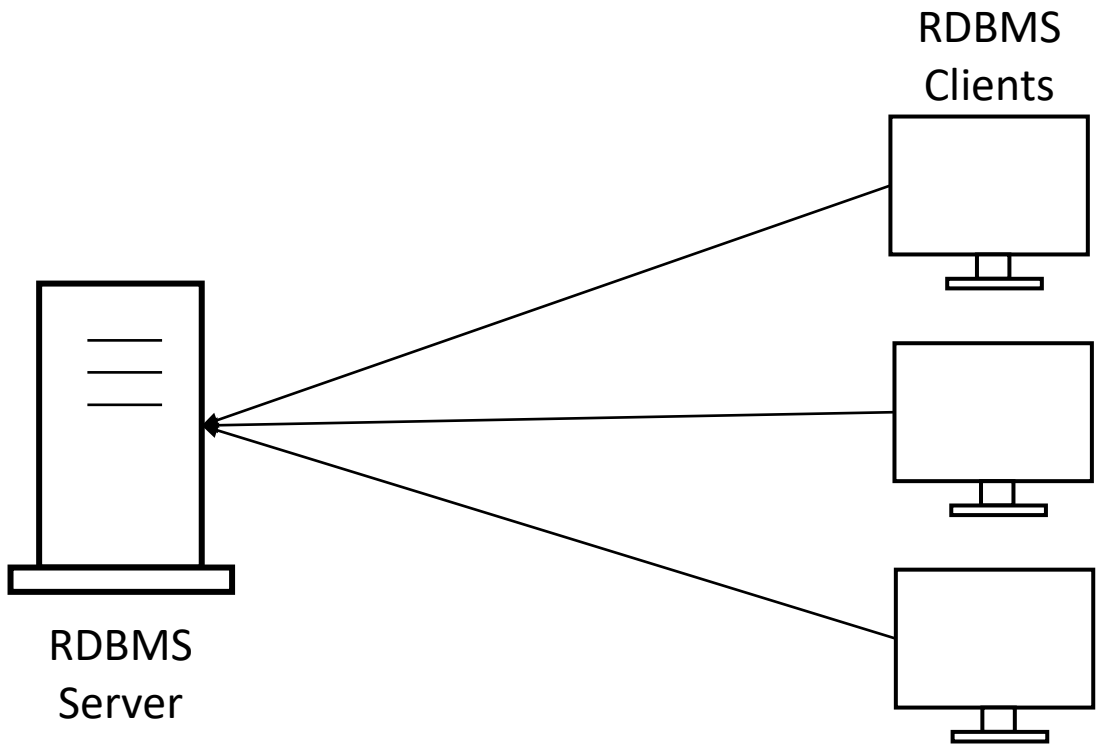
Trainer: Mr. Rohan Paramane

DBMS

- Any enterprise application need to manage data.
- In early days of software development, programmers store data into files and does operation on it. However data is highly application specific.
- Even today many software manage their data in custom formats e.g. Tally, Address book, etc.
- As data management became more common, DBMS systems were developed to handle the data. This enabled developers to focus on the business logic e.g. FoxPro, DBase, Excel, etc.
- At least CRUD (Create, Retrieve, Update and Delete) operations are supported by all databases.
- Traditional databases are file based, less secure, single-user, non-distributed, manage less amount of data (MB), complicated relation management, file-locking and need number of lines of code to use in applications.

RDBMS

- RDBMS is relational DBMS.
- It organizes data into Tables, rows and columns. The tables are related to each other.
- RDBMS follow table structure, more secure, multi-user, server-client architecture, server side processing, clustering support, manage huge data (TB), built-in relational capabilities, table-locking or row-locking and can be easily integrated with applications.
- e.g. DB2, Oracle, MS-SQL, MySQL, MS-Access, SQLite, …
- RDBMS design is based on Codd's rules developed at IBM (in 1970).



SQL

- Clients send SQL queries to RDBMS server and operations are performed accordingly.
- Originally it was named as RQBE (Relational Query By Example).
- SQL is ANSI standardised in 1987 and then revised multiple times adding new features.
- SQL is case insensitive.
- There are five major categories:
  - DDL: Data Definition Language e.g. CREATE, ALTER, DROP, RENAME. Truncate (faster than delete), no where clause
  - DML: Data Manipulation Language e.g. INSERT, UPDATE, DELETE.
  - DQL: Data Query Language e.g. SELECT.
  - DCL: Data Control Language e.g. CREATE USER, GRANT, REVOKE.
  - TCL: Transaction Control Language e.g. SAVEPOINT, COMMIT, ROLLBACK.
- Table & column names allows alphabets, digits & few special symbols.
- If name contains special symbols then it should be back-quotes.
- e.g. Tbl1, `T1#`, `T2$` etc. Names can be max 30 chars long. 64 characters in latest version

MySQL

- Developed by Michael Widenius in 1995. It is named after his daughter name Myia.
- Sun Microsystems acquired MySQL in 2008.
- Oracle acquired Sun Microsystem in 2010.
- MySQL is free and open-source database under GPL. However some enterprise modules are close sourced and available only under commercial version of MySQL.
- MariaDB is completely open-source clone of MySQL.
- MySQL support multiple database storage and processing engines.
- MySQL versions:
  - < 5.5: MyISAM storage engine
  - 5.5: InnoDb storage engine
  - 5.6: SQL Query optimizer improved, memcached style NoSQL
  - 5.7: Windowing functions, JSON data type added for flexible schema
  - 8.0: CTE, NoSQL document store.
- MySQL is database of year 2019 (in database engine ranking).

Getting started

- root login can be used to perform CRUD as well as admin operations.
- terminal> mysql –u root –pmanager mydb
  - mysql> SHOW DATABASES;
  - mysql> SELECT DATABASE();
  - mysql> USE mydb;
  - mysql> SHOW TABLES;
  - mysql> CREATE TABLE student(id INT, name VARCHAR(20), marks DOUBLE);
  - mysql> INSERT INTO student VALUES(1, 'Abc', 89.5);
  - mysql> SELECT * FROM student;

Database logical layout

columns=attributes, rows= tuples

- Database/schema is like a namespace/container that stores all db objects related to a project.
- It contains tables, constraints, relations, stored procedures, functions, triggers, ...
- There are some system databases e.g. mysql, performance_schema, information_schema, sys, ... They contains db internal/system information.
  - e.g. SELECT user, host FROM mysql.user;
- A database contains one or more tables.
- Tables have multiple columns.
- Each column is associated with a data-type.
- Columns may have zero or more constraints.
- The data in table is in multiple rows.
- Each row have multiple values (as per columns).

Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

Agenda

```
DBMS VS RDBMS
SQL
SQL Categories
MySQL
Getting Started with MySQL
Database- Logical & Physical Layout
Data Types
Char vs Varchar vs TEXT
SQL Scripts
```

Documentation Link

- https://dev.mysql.com/doc/refman/8.0/en/introduction.html

DBMS

```
Excel
```

Client - Server Architecture

RDBMS - SQL -> Structured Query Language

RQBE (Relational Query By Example)

MySQL

```
- It was developed by Micheal Widenius in 1995.
- It was named after her daughter Myia
- Sun Microsystems acquired this MySQL in 2008.
- Oracle acquired sun microsystems in 2010.
- Maria DB is a clone of Mysql
```

Getting Started

```
- cmd>mysql -u root -p
- u - Username
- p - password
- h - hostname
```

Program Files

```
C:\Program Files\MySQL\MySQL Server 8.0\bin
mysql.exe
mysqld.exe
```

Program data

```
C:\Program Files\MySQL\MySQL Server 8.0\data
```

Steps to Follow

```
-- Login to your mysql Server using below command
cmd> mysql -u root -p

-- See which database you are currently using
SELECT DATABASE(); --null

--If you wish to see all existing databases
SHOW DATABASES;

-- Create your first database
CREATE DATABASE mydb;
SHOW DATABASES;
USE mydb;
SHOW TABLES; -- Empty set

CREATE TABLE stud(
    rollno INT,
    name CHAR(20),
    marks DOUBLE,std ENUM("Std-1","Std-2")
);

SHOW TABLES;
```

SQL Categories

1. DDL -> Data Defination Language
    - Create,Alter,Drop,Truncate Rename
2. DQL -> Data query Language
    - Select
3. DML -> Data Manipulation Language
    - Insert,Update,Delete
4. DCL -> Data Control Language
    - Create user,grant,revoke

5. TCL -> Transaction Contol Language
    - Savepoint,commit,rollback

```
--To add the data into the table use below DML query
INSERT INTO stud VALUES(1,"Pradnya",80);
INSERT INTO stud VALUES(2,"Girish",70);

INSERT INTO stud VALUES("Suvrunda",80,3); -- error cannot do

INSERT INTO stud(name,marks,rollno) VALUES("Suvrunda",80,3);

INSERT INTO stud VALUES(4,"Ganesh",50),(5,"Manjusha",60);

-- To view all the data from the table use below DQL Query
SELECT * FROM stud; --* means all

-- To check the table structure use below query
DESCRIBE stud;
```

Datatypes

- Numeric Datatypes
    - tinyint(1 byte)
    - smallint(2 bytes)
    - medium int (3 bytes)
    - int (4 bytes)
    - bigint(8 bytes)
    - Float(4 bytes)
    - Double (8 bytes)
    - Decimal (m,n)
        - m-> no of digits
        - n-> no of digits after decimal eg - DECIMAL(4,2) -> 12.34
- String Datatypes
    - Char(n)
    - Varchar(n)
    - tinytext(255)
    - text(64K)
    - medium text(16 MB)
    - longText(4 GB)
- Binary DataTypes
    - TinyBlob, MediumBlob,Blob
- Date Datatypes

    - Date - (yyyy-mm-dd) (1000-01-01 to 9999-12-31)
    - Time - (hr:min:sec) (839:59:32)
    - DateTime - (yyyy-mm-dd :: hr:min:sec)
    - Year - 1901- 2155

- Misc Datatypes

    - ENUM
    - SET

Char vs Varchar vs TEXT

```
CREATE TABLE temp(c1 char(4),c2 varchar(4),c3 text(4));

INSERT INTO temp VALUES('ab','ab','ab');
INSERT INTO temp VALUES('pqr','pqr','pqr');
```

To delete the database (Think Twice before deleting)

```
DROP DATABASE mydb;
```

Sql Scripts

```
CREATE DATABASE classwork;
USE classwork;
SOURCE <Path of .sql file to be imported>
```

MySQL - RDBMS

Trainer: Mr. Rohan Paramane

CHAR vs VARCHAR vs TEXT

- CHAR
  - Fixed inline storage.
  - If smaller data is given, rest of space is unused.
  - Very fast access.
- VARCHAR
  - Variable inline storage.
  - Stores length and characters.
  - Slower access than CHAR.
- TEXT
  - Variable external storage.
  - Very slow access.
  - Not ideal for indexing.
- CREATE TABLE temp(c1 CHAR(4), c2 VARCHAR(4), c3 TEXT(4));
- DESC temp;
- INSERT INTO temp VALUES('abcd', 'abcd', 'abcdef');

INSERT – DML

- Insert a new row (all columns, fixed order).
  - INSERT INTO table VALUES (v1, v2, v3);
- Insert a new row (specific columns, arbitrary order).
  - INSERT INTO table(c3, c1, c2) VALUES (v3, v1, v2);
  - INSERT INTO table(c1, c2) VALUES (v1, v2);
  - Missing columns data is NULL.
  - NULL is special value and it is not stored in database.
- Insert multiple rows.
  - INSERT INTO table VALUES (av1, av2, av3), (bv1, bv2, bv3), (cv1, cv2, cv3).
- Insert rows from another table.
  - INSERT INTO table SELECT c1, c2, c3 FROM another-table;
  - INSERT INTO table (c1,c2) SELECT c1, c2 FROM another-table;

SQL scripts

- SQL script is multiple SQL queries written into a .sql file.
- SQL scripts are mainly used while database backup and restore operations.
- SQL scripts can be executed from terminal as:
  - terminal> mysql –u user –ppassword db < /path/to/sqlfile
- SQL scripts can be executed from command line as:
  - mysql> SOURCE /path/to/sqlfile
- Note that SOURCE is MySQL CLI client command.
- It reads commands one by one from the script and execute them on server.

SELECT – DQL

- Select all columns (in fixed order).
  - SELECT * FROM table;
- Select specific columns / in arbitrary order.
  - SELECT c1, c2, c3 FROM table;
- Column alias
  - SELECT c1 AS col1, c2 col2 FROM table;
- Computed columns.
  - SELECT c1, c2, c3, expr1, expr2 FROM table;

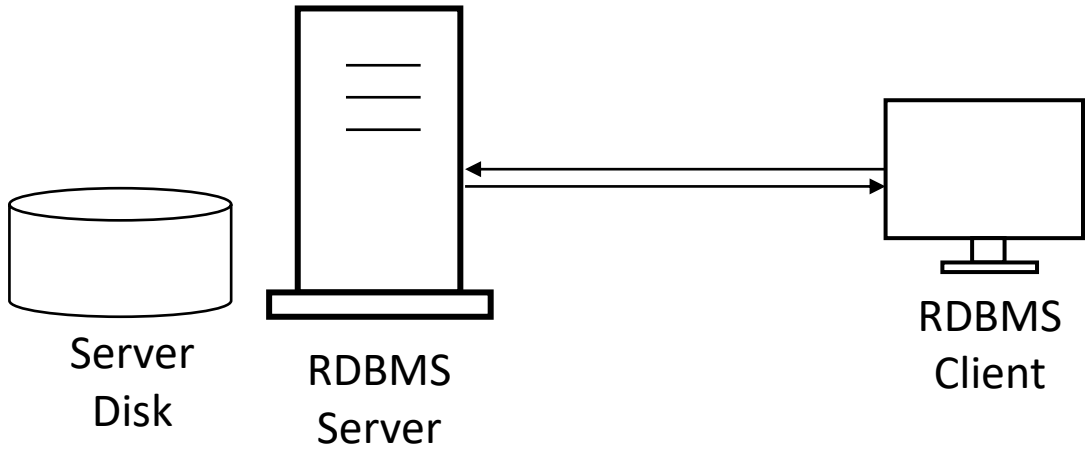
```
SELECT c1,
CASE WHEN condition1 THEN value1,
WHEN condition2 THEN value2,
...
ELSE valuen
END
FROM table;
```

SELECT – DQL

- Distinct values in column.
  - SELECT DISTINCT c1 FROM table;
  - SELECT DISTINCT c1, c2 FROM table;
- Select limited rows.
  - SELECT * FROM table LIMIT n;
  - SELECT * FROM table LIMIT m, n;

SELECT – DQL – ORDER BY

- In db rows are scattered on disk. Hence may not be fetched in a fixed order.
- Select rows in asc order.
  - SELECT * FROM table ORDER BY c1;
  - SELECT * FROM table ORDER BY c2 ASC;
- Select rows in desc order.
  - SELECT * FROM table ORDER BY c3 DESC;
- Select rows sorted on multiple columns.
  - SELECT * FROM table ORDER BY c1, c2;
  - SELECT * FROM table ORDER BY c1 ASC, c2 DESC;
  - SELECT * FROM table ORDER BY c1 DESC, c2 DESC;
- Select top or bottom n rows.
  - SELECT * FROM table ORDER BY c1 ASC LIMIT n;
  - SELECT * FROM table ORDER BY c1 DESC LIMIT n;
  - SELECT * FROM table ORDER BY c1 ASC LIMIT m, n;



SELECT – DQL – WHERE

- It is always good idea to fetch only required rows (to reduce network traffic).
- The WHERE clause is used to specify the condition, which records to be fetched.
- Relational operators
  - <, >, <=, >=, =, != or <>
- NULL related operators
  - NULL is special value and cannot be compared using relational operators.
  - IS NULL or <=>, IS NOT NULL.
- Logical operators
  - AND, OR, NOT

SELECT – DQL – WHERE

- BETWEEN operator (include both ends)
  - c1 BETWEEN val1 AND val2
- IN operator (equality check with multiple values)
  - c1 IN (val1, val2, val3)
- LIKE operator (similar strings)
  - c1 LIKE ‘pattern’.
  - % represent any number of any characters.
  - _ represent any single character.

UPDATE – DML

- To change one or more rows in a table.
- Update row(s) single column.
  - UPDATE table SET c2=new-value WHERE c1=some-value;
- Update multiple columns.
  - UPDATE table SET c2=new-value, c3=new-value WHERE c1=some-value;
- Update all rows single column.
  - UPDATE table SET c2=new-value;

Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

Agenda - Day 02

```
- DQL
- Computed Columns
- Distinct
- LIMIT
- Order By
- WHERE clause
- Relational Operators
- IN,BETWEEN Operator
- LIKE
- DML-UPDATE
```

DQL

```sql
SELECT DATABASE();
USE classwork;

--Display all data from emp table
SELECT * FROM emp;

--To dsplay specific columns from table
SELECT empno,ename,deptno FROM emp;
SELECT deptno,ename,empno FROM emp;

--To Calculate DA and Gross salary for emps
SELECT ename,sal FROM emp;
SELECT ename,sal,sal*0.5 FROM emp;
SELECT ename,sal,sal*0.5 AS DA,sal+sal*0.5 AS GS FROM emp;
SELECT ename,sal,sal*0.5 DA,sal+sal*0.5 GS FROM emp;
SELECT ename,sal,sal*0.5 AS DA,sal + sal*0.5 AS `Gross salary` FROM emp;

-- dispaly ename,deptno,dname from emp table.
SELECT ename,deptno, CASE
WHEN deptno=10 THEN 'ACCOUNTING'
WHEN deptno=20 THEN 'RESEARCH'
WHEN deptno=30 THEN 'SALES'
ELSE 'UNKNOWN'
END AS dname
FROM emp;
```

Distinct

```sql
--display all job from emp
SELECT job FROM emp;
SELECT DISTINCT job FROM emp;
```

```sql
--display unique deptno from emp
SELECT DISTINCT deptno FROM emp;

--display unique jobs from each dept
SELECT DISTINCT deptno,job FROM emp;
```

Limit

```sql
-- disply 5 employees from emp table.
SELECT * FROM emp LIMIT 5;

--skip first 3 records and then give next 5 records
SELECT * FROM emp LIMIT 3,5;
```

OrderBy

```sql
--display emps sorted on their sal
SELECT * FROM emp ORDER BY sal ASC;
SELECT * FROM emp ORDER BY sal;
SELECT * FROM emp ORDER BY sal DESC;

--display emps sorted based on their ename
SELECT * FROM emp ORDER BY ename DESC;

--dispaly deptno and job in sorted manner.
SELECT deptno,job FROM emp ORDER BY deptno,job;

--display top 3 emps as per sal
SELECT * FROM emp ORDER BY sal DESC;
SELECT * FROM emp ORDER BY sal DESC LIMIT 3;

--display emp whose name is last in alphabetical order.
SELECT * FROM emp ORDER BY ename DESC LIMIT 1;

-- dispaly emp with lowest salary
SELECT * FROM emp ORDER BY sal ASC LIMIT 1;

-- dispaly emp with third lowest salary
SELECT * FROM emp ORDER BY sal ASC LIMIT 2,1;

-- dispaly emp with second highest salary
SELECT * FROM emp ORDER BY sal DESC LIMIT 1,1;

-- display emp sorted on their DA
SELECT ename,sal,sal*0.5 FROM emp ORDER BY sal*0.5;
SELECT ename,sal,sal*0.5 AS DA FROM emp ORDER BY DA;
SELECT ename,sal,sal*0.5 AS DA FROM emp ORDER BY 3;
```

Where Clause

```
--display all the employees from dept 30
SELECT * FROM emp WHERE deptno=30;

-- dispaly emps with job as Clerk
SELECT * FROM emp WHERE job='clerk';

-- display all emps with sal greater than 2000
SELECT * FROM emp WHERE sal > 2000;

-- dispaly all the employess who are not in dept 30
SELECT * FROM emp WHERE deptno != 30;
SELECT * FROM emp WHERE deptno <> 30;

--display all emps who are not salesman
SELECT * FROM emp WHERE job != 'salesman';
SELECT * FROM emp WHERE job <> 'salesman';
SELECT * FROM emp WHERE NOT job = 'salesman';

--display emps with job as Analyst and As Manager
SELECT * FROM emp WHERE job = 'Analyst' or job = 'Manager';
SELECT * FROM emp WHERE job IN ('Analyst','Manager');

-- dispaly emps with salaray range 2000 to 3000
SELECT ename,sal FROM emp WHERE sal>=2000 AND sal<=3000;
SELECT ename,sal FROM emp WHERE sal BETWEEN 2000 AND 3000;

-- display all the emps that were hired in 1982.
SELECT * FROM emp WHERE hire>='1982-01-01' and hire<='1982-12-31';
SELECT * FROM emp WHERE hire BETWEEN '1982-01-01' and '1982-12-31';
```

```
-- display all emps with comm as null.
SELECT * FROM emp WHERE comm IS NULL;
SELECT * FROM emp WHERE comm <=> NULL;

-- display all emps with comm as not null.
SELECT * FROM emp WHERE comm IS NOT NULL;
SELECT * FROM emp WHERE NOT comm IS NULL;
```

```
--Practice Examples
INSERT INTO emp(ename) VALUES('B'),('J'),('K');

--dispaly all the enames that ranges from B to J;
SELECT ename FROM emp;
SELECT ename FROM emp WHERE ename>='B';
SELECT ename FROM emp WHERE ename<='J';
```

```
SELECT ename FROM emp WHERE ename>='B' AND ename <'K';

--display all the emps whose salaray is not in range of 1000 and 2000
SELECT * FROM emp WHERE NOT sal BETWEEN 1000 AND 2000;

--dispaly all the enames between B to K and T to Z;
SELECT ename FROM emp WHERE ename>='B' AND ename <'L';
SELECT ename FROM emp WHERE ename>='T';
SELECT ename FROM emp WHERE ename>='B' AND ename <'L' or ename>='T';
```

LIKE OPERATOR

```
% -> Any no of characters
_ -> Single occurance of character
NOT LIKE -> reverse of like
```

```
--display enames with name starting with 'M'
SELECT ename FROM emp WHERE ename>='M' AND ename <'N';
SELECT ename FROM emp WHERE ename LIKE 'M%';

--display enames with name not starting with 'M'
SELECT ename FROM emp WHERE ename NOT LIKE 'M%';

--display enames with name starting with 'H'
SELECT ename FROM emp WHERE ename LIKE 'H%';

--display enames with name ending with 'N'
SELECT ename FROM emp WHERE ename LIKE '%N';

--display enames with name having letter 'O'
SELECT ename FROM emp WHERE ename LIKE '%O%';

--display enames with name having letter 'A' two times
SELECT ename FROM emp WHERE ename LIKE '%A%A%';

--dispaly enames having 4 letter name
SELECT ename FROM emp WHERE ename LIKE '____';

--dispaly enames having 3rd letter as 'R'
SELECT ename FROM emp WHERE ename LIKE '__R%';

--dispaly enames having 4 letter name and having 3rd letter as 'R'
SELECT ename FROM emp WHERE ename LIKE '__R_';
```

```
--Examples for Practice
-- display single emp with highest salary in range 1000 to 2000;
SELECT * FROM emp;
SELECT * FROM emp WHERE sal BETWEEN 1000 AND 2000;
SELECT * FROM emp WHERE sal BETWEEN 1000 AND 2000 ORDER BY sal DESC;
SELECT * FROM emp WHERE sal BETWEEN 1000 AND 2000 ORDER BY sal DESC LIMIT 1;

--display clerk with min salary.
SELECT job,sal FROM emp;
SELECT job,sal FROM emp WHERE job='clerk';
SELECT job,sal FROM emp WHERE job='clerk' ORDER BY sal;
SELECT job,sal FROM emp WHERE job='clerk' ORDER BY sal LIMIT 1;

--display fifth lowest salary from dept 20 and 30
SELECT deptno,sal FROM emp;
SELECT DISTICNT sal FROM emp WHERE deptno IN(20,30);
SELECT DISTINCT sal FROM emp WHERE deptno IN(20,30) ORDER BY sal LIMIT 4,1;
```

DML- UPDATE

```
UPDATE emp SET empno=1234 WHERE ename='B';

UPDATE emp SET deptno=40 WHERE empno=1234;

UPDATE emp SET sal=3000,comm=600 WHERE empno=1234;
```

MySQL - RDBMS

Trainer: Mr. Rohan Paramane

DELETE – DML vs TRUNCATE – DDL vs DROP – DDL

- DELETE
  - To delete one or more rows in a table.
  - Delete row(s)
    - DELETE FROM table WHERE c1=value;
  - Delete all rows
    - DELETE FROM table
- TRUNCATE
  - Delete all rows.
    - TRUNCATE TABLE table;
  - Truncate is faster than DELETE.
- DROP
  - Delete all rows as well as table structure.
    - DROP TABLE table;
    - DROP TABLE table IF EXISTS;
  - Delete database/schema.
    - DROP DATABASE db;

Seeking HELP

- HELP is client command to seek help on commands/functions.
  - HELP SELECT;
  - HELP Functions;
  - HELP SIGN;

DUAL table

single row and single table

- A dummy/in-memory a table having single row & single column.
- It is used for arbitrary calculations, testing functions, etc.
  - SELECT 2 + 3 * 4 FROM DUAL;
  - SELECT NOW() FROM DUAL;
  - SELECT USER(), DATABASE() FROM DUAL;
- In MySQL, DUAL keyword is optional.
  - SELECT 2 + 3 * 4;
  - SELECT NOW();
  - SELECT USER(), DATABASE();

SQL functions

- RDBMS provides many built-in functions to process the data.
- These functions can be classified as:
  - Single row functions
    - One row input produce one row output.
    - e.g. ABS(), CONCAT(), IFNULL(), ...
  - Multi-row or Group functions
    - Values from multiple rows are aggregated to single value.
    - e.g. SUM(), MIN(), MAX(), ...
- These functions can also be categorized based on data types or usage.
  - Numeric functions
  - String functions
  - Date and Time functions
  - Control flow functions
  - Information functions
  - Miscellaneous functions

Numeric & String functions

- ABS()
- POWER()
- ROUND(), FLOOR(), CEIL()

- ASCII(), CHAR()
- CONCAT()
- SUBSTRING()
- LOWER(), UPPER()
- TRIM(), LTRIM(), RTRIM()
- LPAD(), RPAD()
- REGEXP_LIKE()

Date-Time and Information functions

- VERSION()
- USER(), DATABASE()

- MySQL supports multiple date time related data types
  - DATE (3), TIME (3), DATETIME (5), TIMESTAMP (4), YEAR (1)
- SYSDATE(), NOW()
- DATE(), TIME()
- DAYOFMONTH(), MONTH(), YEAR(), HOUR(), MINUTE(), SECOND(), …
- DATEDIFF(), DATE_ADD(), TIMEDIFF()
- MAKEDATE(), MAKETIME()

Control and NULL and List functions

- NULL is special value in RDBMS that represents absence of value in that column.
- NULL values do not work with relational operators and need to use special operators.
- Most of functions return NULL if NULL value is passed as one of its argument.
- ISNULL()
- IFNULL()
- NULLIF()
- COALESCE()

- GREATEST(), LEAST()

- IF(condition, true-value, false-value)

Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

Agenda

```
DML-DELETE
DDL- DROP,TRUNCATE
DUAL
SQL FUNCTIONS
    String Functions
    Numeric Functions
    Date and Time Functions
    Flow Control Functions
    Group Functions
```

DELETE-

```
DELETE FROM emp WHERE ename='B';
--Whenever you delete compulsary give the condition.
--Dont delete without giving condition

DELETE FROM emp WHERE ename='J';
--If the condition fetches multiple records then all those records will be deleted

DELETE FROM emp;
--It will delete all the records from that table
```

DDL- DROP,TRUNCATE

```
TRUNCATE table books;
--It is going to delete all the records in tha table

DROP TABLE emp;
--this will remove the entire table along with its structure from the database

DROP DATABASE classwork;
--this will remove the entire database
```

DELETE,DROP,TRUNCATE

```
- DELETE
    - IT is a DML Query
    - It deletes data based on where clause
    - Deleted data can be rolled back.
- TRUNCATE
    - It is a DDL Query
```

```
    - It deletes the entire data
    - truncated data cannot be rolled back
- DROP
    - It is a DDL Query
    - It will remove the entire table/database structure.
    - It cannot be rolled back
```

DUAL

```
- It is in a memory with sigle row and single column
- It is a virtual Table.
```

```sql
SELECT 5*18 FROM DUAL;
SELECT DATABASE() FROM DUAL;
```

HELP

```
It will help to get the documentation part for any SQL category/Functions etc.
```

FUNCTIONS

```
HELP FUNCTIONS
HELP String Functions
```

String Functions

```sql
SELECT UPPER('sunbeam');
SELECT LOWER('SUNBEAM');
SELECT LOWER(ename) AS ename,sal FROM emp;

SELECT LEFT('SUNBEAM',3); -- SUN
SELECT RIGHT('Sunbeam',3);-- eam

--dispaly first 2 chars and last 2 chars of ename
SELECT LEFT(ename,2), RIGHT(ename,2) FROM emp;

--dispaly emps witing range from B to J
SELECT ename FROM emp WHERE ename BETWEEN 'B' AND 'J';
SELECT ename FROM emp WHERE LEFT(ename,1) BETWEEN 'B' AND 'J';

SELECT SUBSTRING('sunbeam',2);
```

```
--It will go to postition from left
SELECT SUBSTRING('sunbeam',2,3);

SELECT SUBSTRING('sunbeam',-2);
--It will go to postition from right
SELECT SUBSTRING('sunbeam',-5,3);

--dispaly 2nd to 4th position chars of ename.
SELECT SUBSTRING(ename,2,3) FROM emp;

SELECT CONCAT('Sunbeam'," ", "Infotech");

--dispaly emp-job from emp table
SELECT CONCAT(ename,"-",job) FROM emp;

--dispaly output from emp table as below
-- SMITH is working in dept 20 as CLERK
SELECT CONCAT(ename," ","is working in ",deptno," as ",job) AS Details FROM emp;

--dispaly ename firstname as capital and rest all in lower case.
SELECT LEFT(ename,1), SUBSTRING(ename,2) FROM emp;
SELECT UPPER(LEFT(ename,1)), LOWER(SUBSTRING(ename,2)) FROM emp;
SELECT CONCAT(UPPER(LEFT(ename,1)), LOWER(SUBSTRING(ename,2))) as ename FROM emp;

SELECT LENGTH('sunbeam');
SELECT TRIM("Sunbeam   ");

--Homework
--Diplay 16 digit credit card number as 1234 XXXX XXXX 5678
```

Numeric Functions

```
-- HELP Numeric Functions
SELECT POWER(8,2);
SELECT SQRT(4);
SELECT PI();

SELECT ROUND(1.25);
SELECT ROUND(10.48);--10
SELECT ROUND(10.52);--11
SELECT ROUND(10.1256,2); -- 10.13
SELECT ROUND(178.1256,-2); -- 200

-- dispaly price of books upto 2 decimal places
SELECT name,price FROM BOOKS;
SELECT name,ROUND(price,2) AS Price FROM BOOKS;

SELECT ROUND(12.25), CEIL(12.25), FLOOR(12.25);
SELECT ROUND(12.75), CEIL(12.75), FLOOR(12.75);
```

Date and Time Functions

```
SELECT NOW(); --2022-03-30 11:50:07
SELECT SYSDATE(); -- 2022-03-30 11:52:43
SELECT DATE(NOW());-- 2022-03-30
SELECT TIME(NOW()); -- 11:52:43
SELECT DATE_ADD(NOW(),INTERVAL 1 YEAR);

--If you want the diffence between two dates in terms of Days
SELECT DATEDIFF(NOW(),'2020-03-22');

SELECT TIMESTAMPDIFF(YEAR,NOW(),'2020-03-22'); --Diff in years

--display experiance in no of years for all employees
SELECT ename,hire,TIMESTAMPDIFF(YEAR,hire,NOW()) as Years FROM emp;

--display experiance in no of years and no of months for all employees
SELECT ename,hire,TIMESTAMPDIFF(YEAR,hire,NOW()) as Years,
TIMESTAMPDIFF(MONTH,hire,NOW())%12 as Months
FROM emp;


SELECT DAY(NOW()),MONTH(NOW()),YEAR(NOW());
SELECT HOUR(NOW()),MINUTE(NOW()),SECOND(NOW());

--dispaly Emps hired in 1982.
SELECT * FROM emp WHERE hire>='1982-01-01' AND hire<='1982-12-31';
SELECT * FROM emp WHERE hire BETWEEN '1982-01-01' AND '1982-12-31';
SELECT * FROM emp WHERE YEAR(hire)='1982';
```

Flow Control Functions

```
-- dispaly emp,sal,category
--Category sal>2500 RICH
--Category sal<2500 POOR

SELECT ename,sal,CASE
WHEN sal>=2500 THEN 'RICH'
ELSE 'POOR'
END AS Category
FROM emp;

SELECT ename,sal,IF(sal>=2500,"RICH",'POOR') AS Category FROM emp;
```

```
--dispaly total income of all emps.
SELECT ename,sal,comm FROM emp;
SELECT ename,sal,comm,IFNULL(comm,0) FROM emp;
SELECT ename,sal,comm,IFNULL(comm,0) as newcomm FROM emp;
```

```
SELECT ename,sal,comm,sal+IFNULL(comm,0) AS TOTAL FROM emp;

--display ename,sal,tax=null if sal=1250.
SELECT ename,sal,NULLIF(sal,1250) FROM emp;
```

List Functions

```
SELECT CONCAT ('a','b','c','d');
SELECT CONCAT ('a','b',NULL,'d');
SELECT GREATEST(10,40,30,20);
SELECT GREATEST(10,40,NULL,20);
SELECT LEAST(10,40,30,20);
SELECT LEAST(10,40,NULL,20);
```

Group Functions

```
-- Display count of total emps from the emp table
SELECT COUNT(*) AS Total FROM emp;

SELECT MAX(sal) FROM emp;
SELECT MIN(sal) FROM emp;
SELECT AVG(sal) FROM emp;

--dispaly max and min income from emp
--income -> sal+comm
SELECT MAX(sal+IFNULL(comm,0)) AS max,MIN(sal+IFNULL(comm,0)) AS min FROM emp;
```

Steps to change the sql_mode

```
1. Run notepad with Administrator privileges
2. Go into the path where your my.ini File is present.
    - C:\Program Data\MySQL\MySQL Server 8.0\my.ini
3. Click on open file in notepad go to the above path.
4. Select All Files Option.
5. When the my.ini file is visible select it and click on open
6. Under Server Section Below mysqld find a variable named sql-mode.
7. Change the varibale value with the below given value.
    - "ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"
8. Save the file and close it.
9. Close cmd and then restart the mysql Server from task manager.
10. Go to services tab in task manager search for mysql.
11. rightclick on MYSQL80 and select Restart.
12. When everthinh is done open the mysql once again and check
    - SELECT @@sql_mode;
    - ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION
```

MySQL - RDBMS

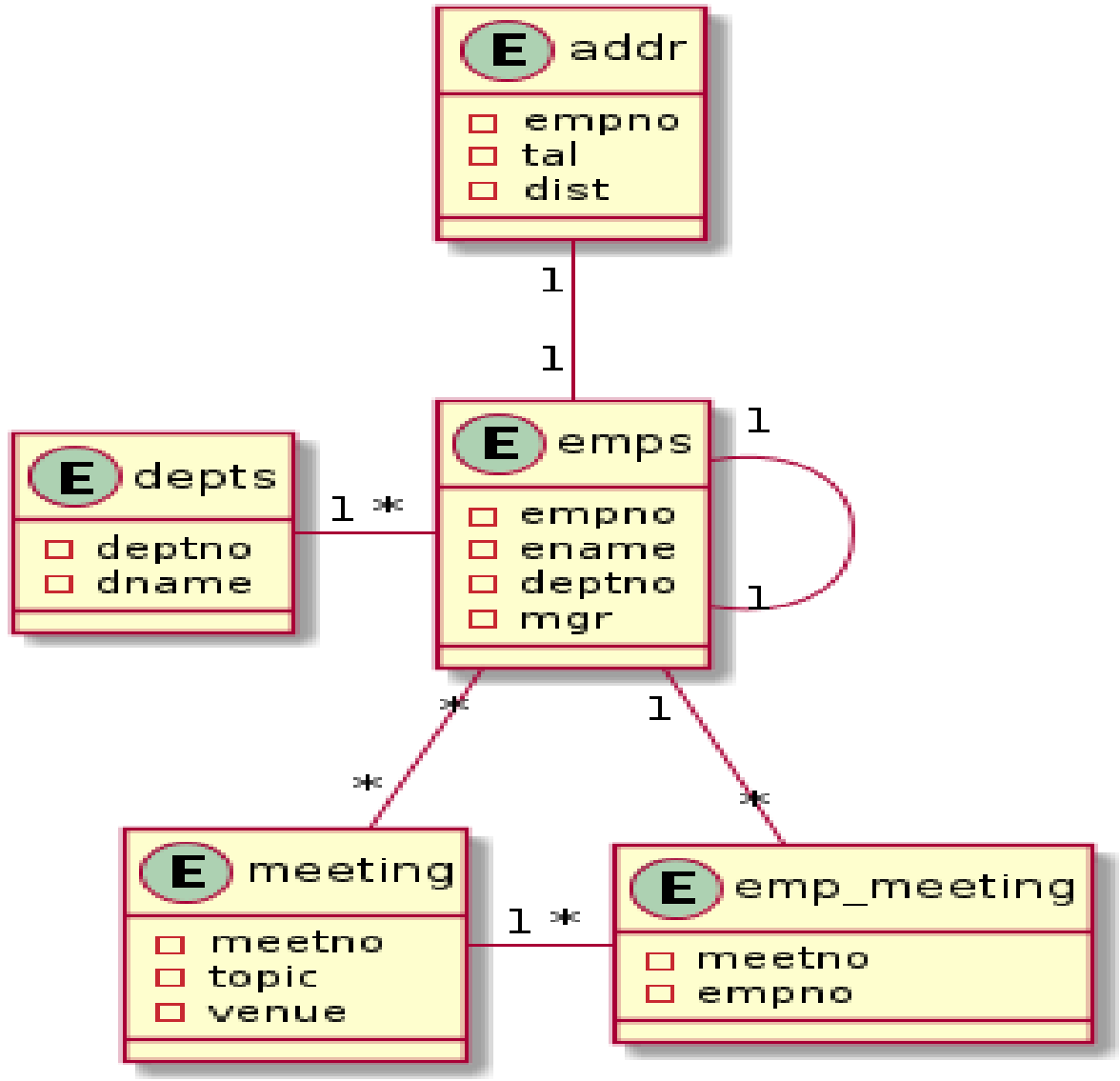
Trainer: Mr. Rohan Paramane

GROUP BY clause

- GROUP BY is used for analysis of data i.e. generating reports & charts.
- When GROUP BY single column, generated output can be used to plot 2-D chart. When GROUP BY two column, generated output can be used to plot 3-D chart and so on.
- GROUP BY queries are also called as Multi-dimensional / Spatial queries.
- Syntactical Characteristics:
  - If a column is used for GROUP BY, then it may or may not be used in SELECT clause.
  - If a column is in SELECT, it must be in GROUP BY.
- When GROUP BY query is fired on database server, it does following:
  - Load data from server disk into server RAM.
  - Sort data on group by columns.
  - Group similar records by group columns.
  - Perform given aggregate ops on each column.
  - Send result to client.

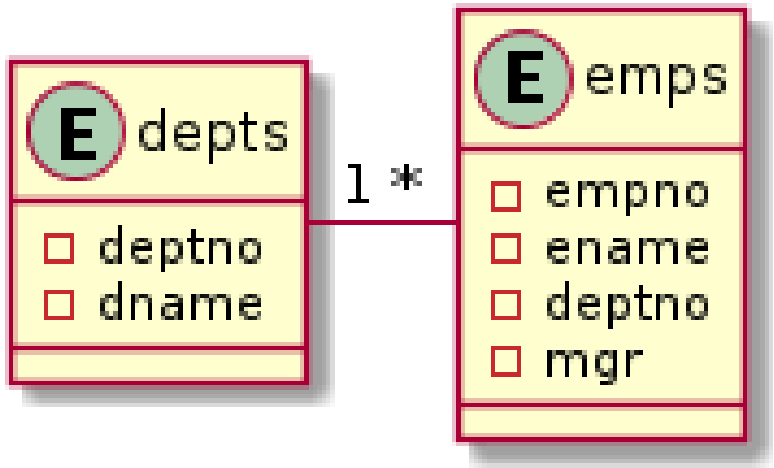
Entity Relations

- To avoid redundancy of the data, data should be organized into multiple tables so that tables are related to each other.
- The relations can be one of the following
  - One to One
  - One to Many
  - Many to One
  - Many to Many
- Entity relations is outcome of Normalization process.



SQL Joins

- Join statements are used to SELECT data from multiple tables using single query.
- Typical RDBMS supports following types of joins:
  - Cross Join
  - Inner Join
  - Left Outer Join
  - Right Outer Join
  - Full Outer Join
  - Self join



Cross Join

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50

- Compares each row of Table1 with every row of Table2.
- Yields all possible combinations of Table1 and Table2.
- In MySQL, The larger table is referred as "Driving Table", while smaller table is referred as "Driven Table". Each row of Driving table is combined with every row of Driven table.
- Cross join is the fastest join, because there is no condition check involved.

Inner Join

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50

natural join = inner equi-join

- The inner JOIN is used to return rows from both tables that satisfy the join condition.
- Non-matching rows from both tables are skipped.
- If join condition contains equality check, it is referred as equi-join; otherwise it is non-equi-join.

Left Outer Join

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50

- Left outer join is used to return matching rows from both tables along with additional rows in left table.
- Corresponding to additional rows in left table, right table values are taken as NULL.
- OUTER keyword is optional.

Right Outer Join

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50

- Right outer join is used to return matching rows from both tables along with additional rows in right table.
- Corresponding to additional rows in right table, left table values are taken as NULL.
- OUTER keyword is optional.

Full Outer Join

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

empno	ename	deptno
1	Amit	10
2	Rahul	10
3	Nilesh	20
4	Nitin	50
5	Sarang	50

- Full join is used to return matching rows from both tables along with additional rows in both tables.
- Corresponding to additional rows in left or right table, opposite table values are taken as NULL.
- Full outer join is not supported in MySQL, but can be simulated using set operators.

Set operators

ename	dname
Amit	DEV
Rahul	DEV
Nilesh	QA
NULL	OPS
NULL	ACC

ename	dname
Amit	DEV
Rahul	DEV
Nilesh	QA
Nitin	NULL
Sarang	NULL

- UNION operator is used to combine results of two queries. The common data is taken only once. It can be used to simulate full outer join.
- UNION ALL operator is used to combine results of two queries. Common data is repeated.

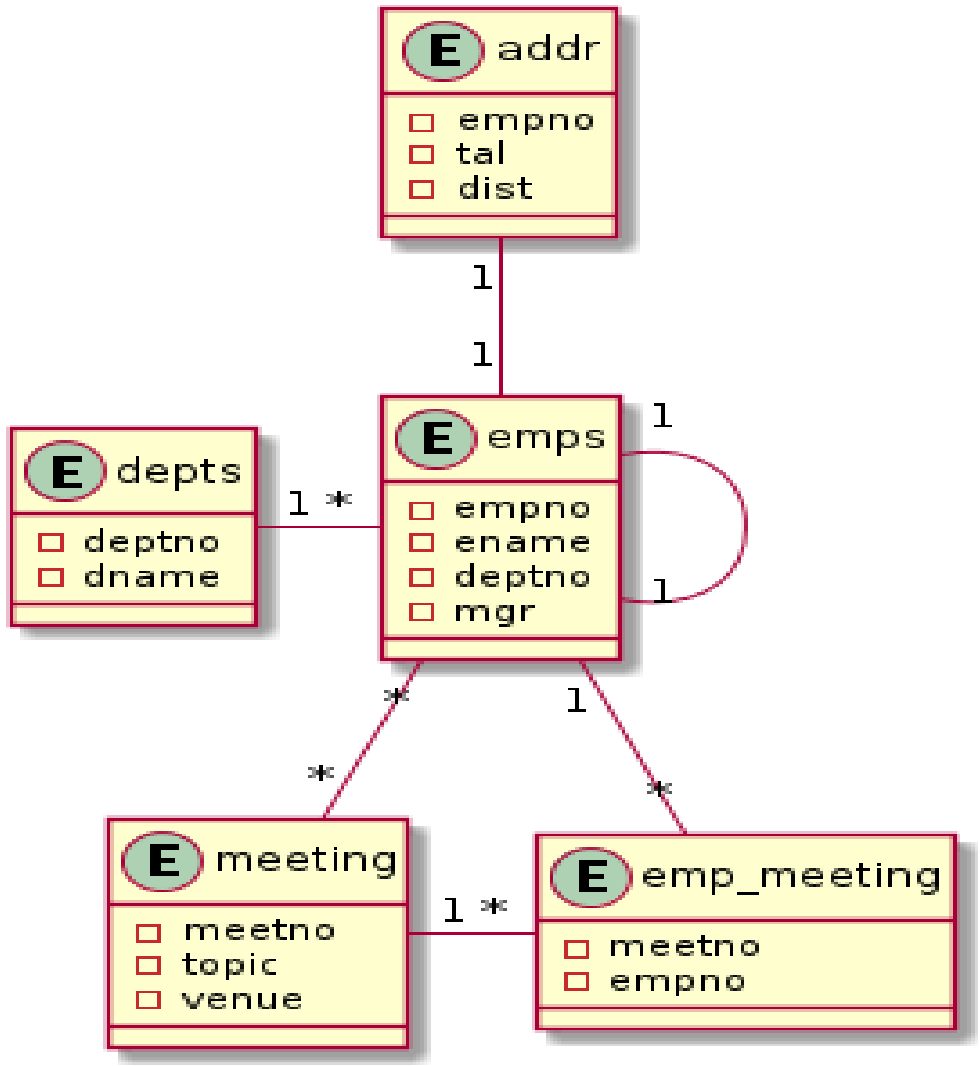
Self Join

- When join is done on same table, then it is known as "Self Join". The both columns in condition belong to the same table.
- Self join may be an inner join or outer join.

empno	ename	deptno	mgr
1	Amit	10	4
2	Rahul	10	3
3	Nilesh	20	4
4	Nitin	50	5
5	Sarang	50	NULL

empno	ename	deptno	mgr
1	Amit	10	4
2	Rahul	10	3
3	Nilesh	20	4
4	Nitin	50	5
5	Sarang	50	NULL

Multi-Table Joins



Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

Agenda

```
Assignment Submission
Group By
Having Clause
Joins
```

Assignment Submission

```
- Change the mysql Prompt name to your name
    - PROMPT B1_12345_Rohan>
- To change the mysql Prompt Permanently follow the below steps
    - Open your Environment Variables
    - Look for User Variables For <user> and click on new
    - Enter Variable name as  MYSQL_PS1
    - Enter Variabe Value as B1_12345_ROHAN>
    - Click Ok.
    - Close all cmd and open once again.

- To start the submission use a new word document for noting down your solution
- Write down the question and solution for it.
- Take a screenshot of the solution that you have got in mysql and paste it below
your solution in word document.
- After completing commit the same on to your git lab account.
- NOTE :- After your assignment is completed on any specific database logically it
should be commited.
- Meaningful message should be given while commiting
    - git commit -m "assignmnet 5 on Sales database is completed"
    - git commit -m "assignmnet 5 q3 deleted"
```

Group Functions Limitation

```sql
SELECT ename,MAX(sal) FROM emp;
-- we cannot select non aggregated column with group functions

SELECT LOWER(ename),MAX(sal) FROM emp;
-- we cannot use single row functions with group functions

SELECT * FROM emp WHERE sal>AVG(sal);
-- we cannot use group function in where clause

SELECT MAX(SUM(sal)) FROM emp;
-- we cannot nest group functions
```

Group By

```
-- dispaly deptno,sum,count,max,min,avg of sal.
SELECT deptno,SUM(sal) FROM emp;
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno;
SELECT deptno,COUNT(sal),SUM(sal),MAX(sal),MIN(sal),AVG(sal) FROM emp GROUP BY
deptno;

-- dispaly job,sum,count,max,min,avg sal for that job.
SELECT job,COUNT(sal),SUM(sal),MAX(sal),MIN(sal),AVG(sal) FROM emp GROUP BY job;

--disply deptno and no of employees working in that dept.
SELECT deptno,COUNT(deptno) FROM emp GROUP BY deptno;

--disply job and no of employees working in that job.
SELECT job,COUNT(*) FROM emp GROUP BY job;

--dispaly deptno,job,countof emps.
SELECT deptno,job FROM emp;
SELECT DISTINCT deptno,job FROM emp;
SELECT deptno,job,COUNT(*) FROM emp GROUP BY deptno,job;
SELECT deptno,job,COUNT(*) FROM emp GROUP BY deptno,job ORDER BY deptno;
```

Having Clause

```
Having clause is used only with group by clause.
If we have any condition based on group function we should be using this clause
```

```
--dispaly dept which have Total spending < 9000
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno;
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno HAVING SUM(sal)<9000;

--dispaly jobs for which AVG sal is > 2000
SELECT job,AVG(sal) FROM emp GROUP BY job HAVING AVG(sal)>2000;

-- dispaly job,max sal for each job from dept 20 and 30
SELECT deptno,job,MAX(sal) FROM emp GROUP BY job,deptno HAVING deptno IN(20,30);
--This is inefficient

SELECT job,MAX(sal) FROM emp WHERE deptno IN(20,30) GROUP BY job;
--This is efficient

--dispaly max sal for each job only if max sal is more than 2500. for emps in
deptno 10 and 20.
SELECT job,MAX(sal) FROm emp WHERE deptno IN(10,20) GROUP BY job;
SELECT job,MAX(sal) FROm emp WHERE deptno IN(10,20) GROUP BY job HAVING
MAX(sal)>2000;
```

```
-- find the one dept that spends max on emp sal's.
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno;
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno ORDER BY SUM(sal) DESC;
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno ORDER BY SUM(sal) DESC LIMIT 1;
```

JOINS

```
-- Import the joins.sql into your classwork Database.
SHOW TABLES;
SELECT * FROM emps;
SELECT * FROM depts;
SELECT * FROM emp_Meeting;
SELECT * FROM meeting;
SELECT * FROM addr;
```

1. CROSS JOIN

```
SELECT e.ename,d.dname FROM emps e CROSS JOIN depts d;

SELECT e.ename,d.dname FROM emps AS e CROSS JOIN depts AS d;

SELECT emps.ename,depts.dname FROM emps CROSS JOIN depts;
```

2. INNER JOIN

```
-- display ename and deptname.
SELECT  e.ename,d.dname FROM emps e
INNER JOIN depts d on e.deptno=d.deptno;
-- equi join

-- dispaly ename and deptname in which he is not working
SELECT  e.ename,d.dname FROM emps e
INNER JOIN depts d on e.deptno!=d.deptno;
-- non-equi join
```

3. LEFT OUTER JOIN

```
LEFT OUTER JOIN = Interection + EXTRA ROWS FROM LEFT TABLE
```

```
-- display ename and deptname.
SELECT e.ename,d.dname FROM emps e
LEFT OUTER JOIN depts d ON e.deptno=d.deptno;

SELECT e.ename,d.dname FROM depts d
LEFT OUTER JOIN emps e ON e.deptno=d.deptno;

SELECT e.ename,d.dname FROM emps e
LEFT JOIN depts d ON e.deptno=d.deptno;
```

3. RIGHT OUTER JOIN

```
RIGHT OUTER JOIN = Interection + EXTRA ROWS FROM RIGHT TABLE
```

```
-- display ename and deptname.
SELECT e.ename,d.dname FROM emps e
RIGHT OUTER JOIN depts d ON e.deptno=d.deptno;

SELECT e.ename,d.dname FROM depts d
RIGHT OUTER JOIN emps e ON e.deptno=d.deptno;

SELECT e.ename,d.dname FROM emps e
RIGHT JOIN depts d ON e.deptno=d.deptno;
```

4. FULL OUTER JOIN

```
FULL OUTER JOIN = Interection + EXTRA ROWS FROM LEFT TABLE + EXTRA ROWS FROM RIGHT
TABLE
FULL OUTER JOIn does not exists in MySQL but can be simulated using set Operators
```

```
SELECT e.ename,d.dname FROM emps e
LEFT OUTER JOIN depts d ON e.deptno=d.deptno
UNION ALL
SELECT e.ename,d.dname FROM emps e
RIGHT OUTER JOIN depts d ON e.deptno=d.deptno;

SELECT e.ename,d.dname FROM emps e
LEFT OUTER JOIN depts d ON e.deptno=d.deptno
UNION
SELECT e.ename,d.dname FROM emps e
RIGHT OUTER JOIN depts d ON e.deptno=d.deptno;
-- Simiulation of Full Outer Join
```

5. SELF JOIN

```
--display ename and mgrname of all emps
-- emps e   emps m
SELECT e.ename,m.ename FROM emps e
INNER JOIN emps m ON e.mgr=m.empno;

SELECT e.ename,m.ename as MGR FROM emps e
INNER JOIN emps m ON e.mgr=m.empno;
```

```
-- dispaly ename,deptname and emps district.
SELECT  e.ename,d.dname FROM emps e
INNER JOIN depts d on e.deptno=d.deptno;

SELECT  e.ename,a.dist FROM emps e
INNER JOIN addr a on e.empno=a.empno;

SELECT  e.ename,d.dname,a.dist FROM emps e
LEFT JOIN depts d on e.deptno=d.deptno
INNER JOIN addr a on e.empno=a.empno;

--disply empname and his meeting topic
SELECT e.ename,em.meetno FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno;

SELECT m.topic,em.empno FROM meeting m
INNER JOIN emp_meeting em ON m.meetno=em.meetno;

SELECT e.ename,m.topic FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno
INNER JOIN meeting m ON m.meetno=em.meetno;
```

Products

pid	Name	cid	Price
1	p1	1	5000
2	p2	2	2500
3	p3	1	
4	p4	2	
5	p5	1	
6	p6	2	

Category

cid	Category
1	Home applinnce
2	Watch

Employee

eno	ename	sal	deptno
1	e1	1000	10
2	e2	2000	10
3	e3	3000	20

Dept

deptno	dname
10	dept1
20	dept2
30	dept3

emps

empno	ename	deptno	mgr
1	Amit	10	4
2	Rahul	10	3
3	Nilesh	20	4
4	Nitin	50	5
5	Sarang	50	NULL

depts

deptno	dname
10	DEV
20	QA
30	OPS
40	ACC

addr

empno	Flatno	Building Name
1	F2	Abc
2	F1	Abc
3	f34	pqr
4	F67	xyz
5	Plot No 2	lmn

meeting

meetno	meetname	Venue
100	m1	v1
200	m2	v2
300	m3	v3

emp_meeting

empno	meetno
1	100
2	100
3	200
4	200
5	200
1	300
2	300
3	300
4	300
5	300

Amit	DEV
Amit	QA
Amit	OPS
Amit	ACC
Rahul	DEV
Rahul	QA
Rahul	OPS
Rahul	ACC
Nilesh	DEV
Nilesh	QA
Nilesh	OPS
Nilesh	ACC
Sarang	

MySQL - RDBMS

Trainer: Mr. Rohan Paramane

Transaction

- Transaction is set of DML queries executed as a single unit.
- Transaction examples
  - accounts table [id, type, balance]
  - UPDATE accounts SET balance=balance-1000 WHERE id = 1;
  - UPDATE accounts SET balance=balance+1000 WHERE id = 2;
- RDBMS transaction have ACID properties.
  - Atomicity
    - All queries are executed as a single unit. If any query is failed, other queries are discarded.
  - Consistency
    - When transaction is completed, all clients see the same data.
  - Isolation
    - Multiple transactions (by same or multiple clients) are processed concurrently.
  - Durable
    - When transaction is completed, all data is saved on disk.

Transaction

- Transaction management
  - START TRANSACTION;
  - …
  - COMMIT WORK;

  - START TRANSACTION;
  - …
  - ROLLBACK WORK;
- In MySQL autocommit variable is by default 1. So each DML command is auto-committed into database.
  - SELECT @@autocommit;
- Changing autocommit to 0, will create new transaction immediately after current transaction is completed. This setting can be made permanent in config file.
  - SET autocommit=0;

Transaction

- Save-point is state of database tables (data) at the moment (within a transaction).
- It is advised to create save-points at end of each logical section of work.
- Database user may choose to rollback to any of the save-point.
- Transaction management with Save-points
  - START TRANSACTION;
  - …
  - SAVEPOINT sa1;
  - …
  - SAVEPOINT sa2;
  - …
  - ROLLBACK TO sa1;
  - …
  - COMMIT; // or ROLLBACK
- Commit always commit the whole transaction.
- ROLLBACK or COMMIT clears all save-points.

Transaction

- Transaction is set of DML statements.
- If any DDL statement is executed, current transaction is automatically committed.
- Any power failure, system or network failure automatically rollback current state.
- Transactions are isolated from each other and are consistent.

Row locking

- When an user update or delete a row (within a transaction), that row is locked and becomes read-only for other users.
- The other users see old row values, until transaction is committed by first user.
- If other users try to modify or delete such locked row, their transaction processing is blocked until row is unlocked.
- Other users can INSERT into that table. Also they can UPDATE or DELETE other rows.
- The locks are automatically released when COMMIT/ROLLBACK is done by the user.
- This whole process is done automatically in MySQL. It is called as "OPTIMISTIC LOCKING".

Row locking

- Manually locking the row in advanced before issuing UPDATE or DELETE is known as "PESSIMISTIC LOCKING".
- This is done by appending FOR UPDATE to the SELECT query.
- It will lock all selected rows, until transaction is committed or rollbacked.
- If these rows are already locked by another users, the SELECT operation is blocked until rows lock is released.

- By default MySQL does table locking. Row locking is possible only when table is indexed on the column.

Data Control Language

- Security is built-in feature of any RDBMS. It is implemented in terms of permissions (a.k.a. privileges).
- There are two types of privileges.
- System privileges
  - Privileges for certain commands i.e. CREATE, ALTER, DROP, ...
  - -Typically these privileges are given to the database administrator or higher authority user.
- Object privileges
  - RDBMS objects are table, view, stored procedure, function, triggers, …
  - -Can perform operations on the objects i.e. INSERT, UPDATE, DELETE, SELECT, CALL, ...
  - Typically these privileges are given to the database users.

User Management

- User management is responsibility of admin (root).
- New user can be created using CREATE USER.
  - CREATE USER user@host IDENTIFIED BY 'password';
  - host can be hostname of server, localhost (current system) or '%' for all client systems.
- Permissions for the user can be listed using SHOW GRANTS command.
  - SHOW GRANTS FOR user@host;
- Users can be deleted using DROP USER.
  - DROP USER user@host;
- Change user password.
  - ALTER USER user@host IDENTIFIED BY 'new_password';
  - FLUSH PRIVILEGES;

Data Control Language

- Permissions are given to user using GRANT command.
  - GRANT CREATE ON db.* TO user@host;
  - GRANT CREATE ON *.* TO user1@host, user2@host;
  - GRANT SELECT ON db.table TO user@host;
  - GRANT SELECT, INSERT, UPDATE ON db.table TO user@host;
  - GRANT ALL ON db.* TO user@host;
- By default one user cannot give permissions to other user. This can be enabled using WITH GRANT OPTION.
  - GRANT ALL ON *.* TO user@host WITH GRANT OPTION;
- Permissions assigned to any user can be withdrawn using REVOKE command.
  - REVOKE SELECT, INSERT ON db.table FROM user@host;
- Permissions can be activated by FLUSH PRIVILEGES.
  - System GRANT tables are reloaded by this command. Auto done after GRANT, REVOKE.
  - Command is necessary is GRANT tables are modified using DML operations.

Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

Agenda

```
Joins Practice
Non Standard Joins
Security
    - DCL
Transactions
    - Locking
```

```
--Display job ID of jobs that were done by more than 1 employees for more than
1500 days.
SELECT job_id FROM job_history GROUP BY job_id HAVING COUNT(employee_id)>3 and
SUM(datediff(end_date,start_date))>100;

SELECT job_id FROM job_history WHERE datediff(end_date,start_date)>100 GROUP BY
job_id HAVING COUNT(employee_id)>3;
```

Joins Practice

```
-- display empname,deptname,dist and meeting topic which he is attendaning.
SELECT e.ename,d.dname FROM emps e
LEFT JOIN depts d ON e.deptno=d.deptno;

SELECT e.ename,a.dist FROM emps e
INNER JOIN addr a on e.empno=a.empno;

SELECT e.ename,em.meetno FROM emps e
INNER JOIN emp_meeting em ON e.empno=em.empno;

SELECT em.empno,m.topic FROM emp_meeting em
INNER JOIN meeting m ON em.meetno=m.meetno;

SELECT e.ename,d.dname,a.dist FROM emps e
LEFT JOIN depts d ON e.deptno=d.deptno
INNER JOIN addr a on e.empno=a.empno;

SELECT e.ename,d.dname,a.dist,m.topic FROM emps e
LEFT JOIN depts d ON e.deptno=d.deptno
INNER JOIN addr a on e.empno=a.empno
INNER JOIN emp_meeting em ON e.empno=em.empno;

SELECT e.ename,d.dname,a.dist,m.topic FROM emps e
LEFT JOIN depts d ON e.deptno=d.deptno
INNER JOIN addr a on e.empno=a.empno
INNER JOIN emp_meeting em ON e.empno=em.empno
INNER JOIN meeting m ON em.meetno=m.meetno;
```

```
-- display dname and count of employees in that dept
SELECT deptno,COUNT(empno) FROM emps GROUP BY deptno;

SELECT d.dname,COUNT(e.empno) FROM emps e
LEFT JOIN depts d ON e.deptno=d.deptno
GROUP BY d.dname;

--dispaly emp name and their meeting count in desc manner of meeting count
SELECT empno,count(meetno) FROM emp_meeting GROUP BY empno;

SELECT e.ename,count(em.meetno) FROM emp_meeting em
INNER JOIN emps e ON em.empno=e.empno
GROUP BY e.ename
ORDER BY count(em.meetno) DESC;

-- display emps from DEV department.
SELECT * FROM emps;
SELECT e.ename,d.dname FROM emps e
INNER JOIN depts d ON e.deptno=d.deptno;

SELECT * FROM emps;

SELECT e.ename,d.dname FROM emps e
INNER JOIN depts d ON e.deptno=d.deptno
WHERE d.dname='DEV';
```

Non Standard Joins

```
--dispaly ename and dname for all employees
SELECT e.ename,d.dname FROM emps e
INNER JOIN depts d ON e.deptno=d.deptno;
-- standard Join

-- Below all are non Standard Joins
SELECT ename,dname FROM emps e
INNER JOIN depts d ON e.deptno=d.deptno;

SELECT ename,dname FROM emps e
JOIN depts d ON e.deptno=d.deptno;

SELECT ename,dname FROM emps e
CROSS JOIN depts d ON e.deptno=d.deptno;

SELECT ename,dname FROM emps e
CROSS JOIN depts d WHERE e.deptno=d.deptno;

SELECT ename,dname FROM emps e,
depts d WHERE e.deptno=d.deptno;
```

```
SELECT ename,dname FROM emps e
INNER JOIN depts d USING (deptno);

SELECT ename,dname FROM emps e
NATURAL JOIN depts d;
```

Security

```
root> CREATE USER 'mgr'@'%' IDENTIFIED BY 'mgr';
root>SELECT user FROM mysql.user;

root> CREATE USER 'teamlead'@'localhost' IDENTIFIED BY 'teamlead';
root> CREATE USER 'dev1'@'localhost' IDENTIFIED BY 'dev1';

mgr>SHOW GRANTS;
teamlead>SHOW GRANTS;

root>GRANT ALL PRIVILEGES ON classwork.* TO 'mgr'@'%' WITH GRANT OPTION;
mgr>SHOW DATABSES;
mgr>USE classowrk;
mgr>SHOW TABLES;
mgr>GRANT SELECT,INSERT,UPDATE,DELETE ON classwork.emp TO 'teamlead'@'localhost';
mgr>GRANT SELECT,INSERT,UPDATE,DELETE ON classwork.dept TO 'teamlead'@'localhost';

mgr>GRANT SELECT,INSERT ON classwork.dept TO 'dev1'@'localhost';
mgr>GRANT SELECT,INSERT ON classwork.dept TO 'dev1'@'localhost';

teamlead>SHOW GRANTS;
teamlead>USE classowrk;
teamlead>SHOW TABLES;
teamlead>SELECT * FROM emp;

dev1>SHOW GRANTS;
dev1>USE classowrk;
dev1>SHOW TABLES;
dev1>DELETE FROM dept WHERE deptno=40; -- cannot do.

mgr>REVOKE INSERT ON classwork.dept FROM 'dev1'@'localhost';

root>DROP USER 'dev1'@'localhost';
mgr>ALTER USER 'mgr'@'%' IDENTIFIED BY 'manager';
```

Transactions

** TCL COMMANDS

- START TRANSACTION
- COMMIT
- ROLLBACK

- SAVEPOINT

```
CREATE TABLE accounts(id INT Primary Key, type Varchar(15),balance Decimal(9,2));
INSERT INTO accounts(id,type,balance)
VALUES(1,"Savings",20000),
(2,"Current",30000),
(3,"Savings",15000),
(4,"Savings",25000),
(5,"Current",18000);

INSERT INTO accounts(id,type,balance) VALUES(6,"CURRENT",28000);

SELECT * FROM accounts;

START TRANSACTION;

INSERT INTO accounts(id,type,balance) VALUES(7,"CURRENT",29000);

SELECT * FROM accounts;

ROLLBACK;

SELECT * FROM accounts;

INSERT INTO accounts(id,type,balance) VALUES(7,"CURRENT",29000);

COMMIT;

SELECT * FROM accounts;

START TRANSACTION;

DELETE FROM accounts WHERE id=7;
SAVEPOINT s1;

DELETE FROM accounts WHERE id IN(5,6);
SAVEPOINT s2;

DELETE FROM accounts WHERE id=4;
SAVEPOINT s3;

DELETE FROM accounts WHERE id=(2,3);
SAVEPOINT s4;

SELECT * FROM accounts;

ROLLBACK TO s4;

SELECT * FROM accounts;

ROLLBACK TO s2;

ROLLBACK;
```

```
SELECT * FROM accounts;
```

Savepoints

```
Savepoint is state of database within a transaction.
User can rollback to any of the savepoint using ROLLBACK TO sa;
In this case all changes after savepoint "sa" are discarded.
ROLLBACK TO "sa" do not ROLLBACK the whole transaction. The same transaction can
be continued further (can make more DML queries).
Transaction is completed only when COMMIT or ROLLBACK is done. All savepoint
memory is released.
COMMIT is not allowed upto a savepoint. We can only commit whole transaction.
```

Transaction properties/characteristics

```
1. Atomicity: All DML queries in transaction will be successful or
failed/discarded. Partial transaction never committed.
2. Consistent: At the end of transaction same state is visible to all the users.
3. Isolation: Each transaction is isolated from each other. All transactions are
added in a transaction queue at server side and process sequentially.
4. Durability: At the end of transaction, final state is always saved (on server
side).

When an user is in a transation, changes done by the user are saved in a temp
table. These changes are visible to that user.
However this temp table is not accessible/visible to other users and hence changes
under progress in a transaction are not visible to other users.
When an user is in a transaction, changes committed by other users are not visible
to him. Because he is dealing with temp data.
```

ROW LOCKING

```
When deleted or updated the row gets locked.

- Pessimistic Locking
SELECT * FROM accounts WHERE id = 2 FOR UPDATE;

- TABLE LOCKING
When deleted or updated the table gets locked if primary key does not exists
```

```
-- TABLE LOCKING

root> SELECT * FROM depts;
root> START TRANSACTION;
root> DELETE FROM depts WHERE deptno = 40;
-- whole table is locked (bcoz no primary key)
mgr> DELETE FROM depts WHERE deptno = 30;
-- blocked
root> COMMIT;
-- root user unblocked
root> SELECT * FROM depts;
mgr> SELECT * FROM depts;
```

```
--ROW LOCKING

mgr> SELECT * FROM accounts;
mgr> START TRANSACTION;
mgr> SELECT * FROM accounts WHERE id = 2 FOR UPDATE;
-- row locked
root> SELECT * FROM accounts;
root> SELECT * FROM accounts WHERE id = 2 FOR UPDATE;
-- blocked
root> DELETE FROM accounts WHERE id = 2;
root> ROLLBACK;
-- root user unblocked
```

MySQL - RDBMS

Trainer: Mr. Rohan Paramane

Sub queries

- Sub-query is query within query. Typically it work with SELECT statements.
- Output of inner query is used as input to outer query.
- If no optimization is enabled, for each row of outer query result, sub-query is executed once. This reduce performance of sub-query.
- Single row sub-query
  - Sub-query returns single row.
  - Usually it is compared in outer query using relational operators.

Sub queries

- Multi-row sub-query
  - Sub-query returns multiple rows.
  - Usually it is compared in outer query using operators like IN, ANY or ALL.
    - IN operator compare for equality with results from sub-queries (at least one result should match)..
    - ANY operator compares with the results from sub-queries (at least one result should match).
    - ALL operator compares with the results from sub-queries (all results should match).

Sub queries

- Correlated sub-query
  - If number of results from sub-query are reduced, query performance will increase.
  - This can be done by adding criteria (WHERE clause) in sub-query based on outer query row.
  - Typically correlated sub-query use IN, ALL, ANY and EXISTS operators.

Sub query

- Sub queries with UPDATE and DELETE are not supported in all RDBMS.
- -In MySQL, Sub--queries in UPDATE/DELETE is allowed, but sub--query should not SELECT from the same table, on which UPDATE/DELETE operation is in progress.

Views

- RDBMS view represents view (projection) of the data.
- View is based on SELECT statement.
- Typically it is restricted view of the data (limited rows or columns) from one or more tables (joins and/or sub-queries) or summary of the data (grouping).
- Data of view is not stored on server hard-disk; but its SELECT statement is stored in compiled form. It speed up execution of view.

Views

- Views are of two types: Simple view and Complex view
- Usually if view contains computed columns, group by, joins or sub-queries, then the views are said to be complex. DML operations are not supported on these views.
- DML operations on view affects underlying table.
- View can be created with CHECK OPTION to ensure that DML operations can be performed only the data visible in that view.

View

- Views can be differentiated with: SHOW FULL TABLES.
- Views can be dropped with DROP VIEW statement.
- View can be based on another view.

- Applications of views
  - Security: Providing limited access to the data.
  - Hide source code of the table.
  - Simplifies complex queries.

Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

Agenda

```
SubQuery
Views
```

SubQuery

```
- It is query inside another query
```

Single Row Subquery.

```
Inner Query return a single row.
```

```sql
-- display emps with maximum/highest salary.
SELECT * FROM emp ORDER BY sal DESC LIMIT 1;
-- It will generate partial output.

SELECT MAX(sal) FROM emp;
SELECT * FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);

-- display emps with second highest salary.
SELECT * FROM emp ORDER BY sal DESC LIMIT 1,1;

SELECT sal FROM emp ORDER BY sal DESC LIMIT 1,1;
SELECT DISTINCT sal FROM emp ORDER BY sal DESC LIMIT 1,1;

SELECT * FROM emp WHERE sal = (SELECT DISTINCT sal FROM emp ORDER BY sal DESC
LIMIT 1,1);

--display emps working in dept of 'KING'
SELECT deptno FROM emp WHERE ename='KING';

SELECT * FROM emp where deptno=(SELECT deptno FROM emp WHERE ename='KING');
```

Multi Row Subquery

```
Inner Query return multiple rows.
They can be compared by using ALL,ANY or IN Operator
```

```
-- display all emps having sal greater than 'all' salesman
SELECT sal FROM emp WHERE job="Salesman";

SELECT * FROM emp WHERE sal > ALL(SELECT sal FROM emp WHERE job="Salesman");
--(sal>1600 AND sal>1250 AND sal>1500)

SELECT * FROM emp WHERE sal >(SELECT MAX(sal) FROM emp WHERE job="Salesman");

-- display all emps having sal less than 'any' emp in dept = 20;
SELECT sal FROM emp WHERE deptno=20;

SELECT * FROM emp WHERE sal < ANY(SELECT sal FROM emp WHERE deptno=20);
--(sal<800 OR sal<2975 OR sal<3000 OR sal<1100)
        AND  OR
0 0 -   0    0
0 1 -   0    1
1 0 -   0    1
1 1 -   1    1

SELECT * FROM emp WHERE sal <(SELECT MAX(sal) FROM emp WHERE deptno=20);

--display dept names which have employees in them
SELECT * FROM dept;

SELECT dname FROM dept WHERE deptno IN(10,20,30);
SELECT dname FROM dept WHERE deptno IN(SELECT DISTINCT deptno FROM emp);
SELECT dname FROM dept WHERE deptno = ANY (SELECT DISTINCT deptno FROM emp);

---display dept names which dont have employees in them
SELECT dname FROM dept WHERE deptno NOT IN(SELECT DISTINCT deptno FROM emp);
SELECT dname FROM dept WHERE deptno != ALL(SELECT DISTINCT deptno FROM emp);
```

ANY vs IN operator

```
- ANY can be used in sub-queries only, while IN can be used with/without sub-
queries.
- ANY can be used for comparision (<, >, <=, >=, =, or !=), while IN can be used
only for equality comparision (=).
- Both operators are logically similar to OR operator.
```

ANY vs ALL operator

```
- Both can be used for comparision (<, >, <=, >=, =, or !=).
- Both are usable only with sub-queries.
- ANY is similar to logical OR, while ALL is similar to logical AND.
```

Co-related Subquery

```
- By default for every row of outer query inner query is executed
- We need some optimization to be done as if nothing done subquery can become
slower than join
- If the condition of inner query depends on current row of outer query then such
type of subquery is called as Co-related subquery.
```

```
SELECT dname FROM dept WHERE deptno = ANY (SELECT deptno FROM emp);
--10, ACCOUNTING --> SELECT deptno FROM emp - 14 rows
--20, RESEARCH --> SELECT deptno FROM emp - 14 rows
--30, SALES --> SELECT deptno FROM emp - 14 rows
--40, OPERATIONS --> SELECT deptno FROM emp - 14 rows

SELECT * FROM dept d WHERE d.deptno IN(SELECT e.deptno FROM emp e WHERE
e.deptno=d.deptno)
--10, ACCOUNTING --> 10,10,10 - 3 rows
--20, RESEARCH --> 20,20,20,20,20 - 5 rows
--30, SALES --> 30,30,30,30,30,30 - 6 rows
--40, OPERATIONS --> 0 rows

SELECT * FROM dept d WHERE d.deptno IN(SELECT DISTINCT e.deptno FROM emp e WHERE
e.deptno=d.deptno);
SELECT * FROM dept d WHERE d.deptno = (SELECT DISTINCT e.deptno FROM emp e WHERE
e.deptno=d.deptno);
--10, ACCOUNTING --> 10 - 1 row
--20, RESEARCH --> 20 - 1 row
--30, SALES --> 30 - 1 row
--40, OPERATIONS --> 0 rows
```

Subquery in projection

```
-- display deptno,count of emp and total no of emps .
--Acc - 3 - 14
--Dev - 5 - 14

SELECT deptno,COUNT(*) as emp_count FROM emp GROUP BY deptno;
SELECT COUNT(*)AS Total_emp FROM emp;
SELECT deptno,COUNT(*) as emp_count,(SELECT COUNT(*) FROM emp) AS Total_emp FROM
emp GROUP BY deptno;
```

Subquery in FROM Clause

```
- Inner-query can be written in FROM clause of SELECT statement. The output of the
inner query is treated as a table (MUST give table alias) and outer query execute
```

```
on that table.
- This is called as "Derived table" or "Inline view".
```

```
--display emp,sal,category(2000>'Rich',Poor)
SELECT ename, sal, IF(sal > 2000, 'RICH', 'POOR') AS category FROM emp;

--display count of employees category wise.
SELECT category,COUNT(*) AS emp_count FROM (SELECT ename, sal, IF(sal > 2000,
'RICH', 'POOR') AS category FROM emp) AS cat_tab  GROUP BY category;
```

Subquery in DML Operations

```
--insert new employee with dept as Operations
SELECT deptno FROM dept WHERE dname='Operations';

INSERT INTO emp(empno,ename,sal,deptno) VALUES(1000,'Jill',2000,(SELECT deptno
FROM dept WHERE dname='Operations'));

--update the comm of the emp to 100 where dept is operations
UPDATE emp SET comm=100 WHERE deptno=(SELECT deptno FROM dept WHERE
dname='Operations');

-- delete all emps from operations dept.
DELETE FROM emp WHERE deptno=(SELECT deptno FROM dept WHERE dname='Operations');

-- delete emps having highest sal.
SELECT MAX(sal) FROM emp;

DELETE FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);
```

SQL Performance

```
- Modern RDBMS implement lot of optimization mechanisms (internally based on data
structures and algorithms) like caching (materialization), semijoins or hashjoins,
etc.
- These optimization logic will differ from RDBMS to RDBMS.
- In MySQL, query Performance is measured in terms of query cost. Lower the cost,
better is performance.
- The cost of query depends on data in the table(s), MySQL version, server machine
config,optimizer settings, etc.
```

```
SELECT @@optimizer_switch;

EXPLAIN FORMAT = JSON SELECT dname FROM dept WHERE deptno = ANY (SELECT deptno
```

```
FROM emp);

EXPLAIN FORMAT = JSON SELECT * FROM dept d WHERE d.deptno IN(SELECT e.deptno FROM
emp e WHERE e.deptno=d.deptno);

EXPLAIN FORMAT = JSON SELECT * FROM dept d WHERE d.deptno = (SELECT DISTINCT
e.deptno FROM emp e WHERE e.deptno=d.deptno);

EXPLAIN FORMAT = JSON SELECT * FROM dept d WHERE d.deptno IN (SELECT DISTINCT
deptno FROM emp);
```

Views

```
View is a Projection of data.
1. Simple View - DQL + DML
2. Complex View - DQL
```

```
SELECT ename, sal, IF(sal > 2000, 'RICH', 'POOR') AS category FROM emp;

CREATE VIEW v_emp_category AS SELECT ename, sal, IF(sal > 2000, 'RICH', 'POOR') AS
category FROM emp;

SHOW TABLES;

SHOW FULL TABLES;

DESC v_emp_category;

SELECT * FROM v_emp_category;

--display count of employees category wise.
SELECT category,COUNT(*) FROM v_emp_category GROUP BY category;

SHOW CREATE VIEW v_emp_category;

ALTER VIEW v_emp_category AS SELECT ename, sal, IF(sal > 2500, 'RICH', 'POOR') AS
category FROM emp;

DROP VIEW v_emp_category;
```

```
--view v_empsal with columns as empno,ename,sal
--view v_richemp with columns as empno,ename,sal>2500
--view v_empincome with columns as empno,ename,sal,comm,total_income
--view v_jobsummary with columns as job,sum(sal),max sal,min sal,avg sal ,count

CREATE VIEW v_empsal AS SELECT empno,ename,sal FROM emp;
```

```
CREATE VIEW v_richemp AS SELECT empno,ename,sal FROM emp WHERE sal>2500;

CREATE VIEW v_empincome AS SELECT empno,ename,sal,comm,sal+IFNULL(comm,0) AS
total_income FROM emp;

CREATE VIEW v_jobsummary AS SELECT job,SUM(sal) sumsal,MAX(sal) maxsal,MIN(sal)
minsal,AVG(sal) avgsal,COUNT(*) emp_count FROM emp GROUP BY job;

INSERT INTO v_jobsummary VALUES('DEVELOPMENT',1000,3000,1500,1222,2);
--cannot do on complex view

INSERT INTO emp(ename,job,sal,deptno) VALUES('JILL','Developemnet',6000,40);

INSERT INTO v_richemp VALUES('1000',"JAMES",2000); --OK

ALTER VIEW v_richemp AS SELECT empno,ename,sal FROM emp WHERE sal>2500 WITH CHECK
OPTION;

INSERT INTO v_richemp VALUES('2000',"JAS",3000);
```

```
SELECT * FROM v_richemp;

CREATE VIEW v_richemp2 AS SELECT empno,ename FROM v_richemp;

SHOW FULL TABLES;

SELECT * FROM v_richemp2;

DROP VIEW v_richemp;

SELECT * FROM v_richemp2; -- error

DROP VIEW v_richemp2;
```

```
--Q1. display empno,ename,deptno,dname.
SELECT e.empno,e.ename,d.deptno,d.dname FROM emp e
INNER JOIN dept d ON e.deptno=d.deptno;

--Q2. dipsplay all employees from Accounting dept
SELECT e.empno,e.ename,d.deptno,d.dname FROM emp e
INNER JOIN dept d ON e.deptno=d.deptno WHERE d.dname = 'ACCOUNTING';

--Q3. Create view for the Q1.
CREATE VIEW v_emp_list AS SELECT e.empno,e.ename,d.deptno,d.dname FROM emp e
INNER JOIN dept d ON e.deptno=d.deptno;

--Q4. dipsplay all employees from Accounting dept using cretaed view
SELECT * FROM v_emp_list WHERE dname='ACCOUNTING';
```

MySQL - RDBMS

Trainer: Mr. Rohan Paramane

Index

- Index enable faster searching in tables by indexed columns.
  - CREATE INDEX idx_name ON table(column);
- One table can have multiple indexes on different columns/order.
- Typically indexes are stored as some data structure (like BTREE or HASH) on disk.
- Indexes are updated during DML operations. So DML operation are slower on indexed tables.

Index

- Index can be ASC or DESC.
  - It cause storage of key values in respective order (MySQL 8.x onwards).
  - ASC/DESC index is used by optimizer on ORDER BY queries.
- There are four types of indexes:
  - Simple index
    - CREATE INDEX idx_name ON table(column [ASC|DESC]);
  - Unique index
    - CREATE UNIQUE INDEX idx_name ON table(column [ASC|DESC]);
    - Doesn't allow duplicate values.
  - Composite index
    - CREATE INDEX idx_name ON table(column1 [ASC|DESC], column2 [ASC|DESC]);
    - Composite index can also be unique. Do not allow duplicate combination of columns.
  - Clustered index
    - PRIMARY index automatically created on Primary key for row lookup.
    - If primary key is not available, hidden index is created on synthetic column.
    - It is maintained in tabular form and its reference is used in other indexes.

Index

- Indexes should be created on shorter (INT, CHAR, …) columns to save disk space.
- Few RDBMS do not allow indexes on external columns i.e. TEXT, BLOB.
- MySQL support indexing on TEXT/BLOB up to n characters.
  - CREATE TABLE test (blob_col BLOB, …, INDEX(blob_col(10)));
- To list all indexes on table:
  - SHOW INDEXES FROM table;
- To drop an index:
  - DROP INDEX idx_name ON table;
- When table is dropped, all indexes are automatically dropped.
- Indexes should not be created on the columns not used frequent search, ordering or grouping operations.
- Columns in join operation should be indexed for better performance.

Query performance

- Few RDBMS features ensure better query performance.
  - Index speed up execution of SELECT queries (search operations).
  - Correlated sub-queries execute faster.
- Query performance can observed using EXPLAIN statement.
  - EXPLAIN FORMAT=JSON SELECT …;
- EXPLAIN statement shows
  - Query cost (Lower is the cost, faster is the query execution).
  - Execution plan (Algorithm used to execute query e.g. loop, semi-join, materialization, etc).
- Optimizations can be enabled or disabled by optimizer_switch system variable.
  - SELECT @@optimizer_switch;
  - SET @@optimizer_switch='materialization=off';

Constraints

- Constraints are restrictions imposed on columns.
- There are five constraints
  - NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK
- Few constraints can be applied at either column level or table level. Few constraints can be applied on both.
- Optionally constraint names can be mentioned while creating the constraint. If not given, it is auto-generated.
- Each DML operation check the constraints before manipulating the values. If any constraint is violated, error is raised.

Constraints

- NOT NULL
  - NULL values are not allowed.
  - Can be applied at column level only.
  - CREATE TABLE table(c1 TYPE NOT NULL, ...);
- UNIQUE
  - Duplicate values are not allowed.
  - NULL values are allowed.
  - Not applicable for TEXT and BLOB.
  - UNIQUE can be applied on one or more columns.
  - Internally creates unique index on the column (fast searching).
  - A table can have one or more unique keys.
  - Can be applied at column level or table level.
    - CREATE TABLE table(c1 TYPE UNIQUE, ...);
    - CREATE TABLE table(c1 TYPE, ..., UNIQUE(c1));
    - CREATE TABLE table(c1 TYPE, ..., CONSTRAINT constraint_name UNIQUE(c1));

Constraints

- PRIMARY KEY
  - Column or set of columns that uniquely identifies a row.
  - Only one primary key is allowed for a table.
  - Primary key column cannot have duplicate or NULL values.
  - Internally index is created on PK column.
  - TEXT/BLOB cannot be primary key.
  - If no obvious choice available for PK, composite or surrogate PK can be created.
  - Creating PK for a table is a good practice.
  - PK can be created at table level or column level.
  - CREATE TABLE table(c1 TYPE PRIMARY KEY, ...);
  - CREATE TABLE table(c1 TYPE, ..., PRIMARY KEY(c1));
  - CREATE TABLE table(c1 TYPE, ..., CONSTRAINT constraint_name PRIMARY KEY(c1));
  - CREATE TABLE table(c1 TYPE, c2 TYPE, ..., PRIMARY KEY(c1, c2));

DDL – ALTER statement

- ALTER statement is used to do modification into table, view, function, procedure, ...
- ALTER TABLE is used to change table structure.
- Add new column(s) into the table.
  - ALTER TABLE table ADD col TYPE;
  - ALTER TABLE table ADD c1 TYPE, c2 TYPE;
- Modify column of the table.
  - ALTER TABLE table MODIFY col NEW_TYPE;
- Rename column.
  - ALTER TABLE CHANGE old_col new_col TYPE;
- Drop a column
  - ALTER TABLE DROP COLUMN col;
- Rename table
  - ALTER TABLE table RENAME TO new_table;

Add the constraint
modify the datatypes
change the name of the column
drop the column
rename the name of the table

Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

Agenda

```
Indexes
Constraints
    Not Null
    Unique
    Primary Key
    Foreign Key
    CHECK
ALTER
```

Indexes

```
It helps for Faster Searching
```

1. Simple Index - If index is applied on Single column

```
EXPLAIN FORMAT = JSON SELECT * FROM books WHERE subject = 'C Programming';
--1.55

CREATE INDEX idx_books_subject ON books(subject);

EXPLAIN FORMAT = JSON SELECT * FROM books WHERE subject = 'C Programming';
--0.90

EXPLAIN FORMAT = JSON SELECT e.ename,d.dname FROM emps e INNER JOIN depts d ON
e.deptno=d.deptno;
--2.50

CREATE INDEX idx_emps_deptno ON emps(deptno);
CREATE INDEX idx_depts_deptno ON depts(deptno);

EXPLAIN FORMAT = JSON SELECT e.ename,d.dname FROM emps e INNER JOIN depts d ON
e.deptno=d.deptno;
--2.05

SHOW INDEXES FROM emps;

DESC emps;
```

2. Unique Index

```
SELECT * FROM emps;
CREATE UNIQUE INDEX idx_emps_ename ON emps(ename);
SELECT * FROM emps WHERE ename='Rahul';

INSERT INTO emps VALUES(6,'Nitin',50,5);
-- error -  Duplicate entry 'Nitin' for key 'emps.idx_emps_ename'

INSERT INTO emps VALUES(6,NULL,50,5);
INSERT INTO emps VALUES(7,NULL,30,4);
-- Multiple NULL values are allowed.

CREATE UNIQUE INDEX idx_emps_mgr ON emps(mgr);
--error
```

3. Composite Index - Index on 2 cols

```
--display emps from dept = 20 and job as analyst
EXPLAIN FORMAT = JSON SELECT * FROM emp WHERE deptno=20 and job='Analyst';
--2.05

CREATE INDEX idx_emp_dj ON emp(deptno ASC,job ASC);

EXPLAIN FORMAT = JSON SELECT * FROM emp WHERE deptno=20 and job='Analyst';
--0.70

EXPLAIN FORMAT = JSON SELECT * FROM emp WHERE sal=5000;
--2.05

EXPLAIN FORMAT = JSON SELECT * FROM emp WHERE deptno=20;
--1.00

EXPLAIN FORMAT = JSON SELECT * FROM emp WHERE job='Analyst';
--2.05
```

```
CREATE TABLE students(roll INT,std INT,name char(20), marks Decimal(5,2));
INSERT INTO students VALUES(1,1,'s1',10);
INSERT INTO students VALUES(2,1,'s2',20);
INSERT INTO students VALUES(3,1,'s3',30);
INSERT INTO students VALUES(1,2,'s4',10);
INSERT INTO students VALUES(2,2,'s5',20);

CREATE UNIQUE INDEX idx1 ON students(roll,std);

INSERT INTO students VALUES(2,2,'s6',30);
--error

INSERT INTO students VALUES(3,2,'s6',30);
```

4. Clustered Index
    - It is Create automatically on primary key.
    - If you dont give primary key then also a hidden column will be added and index will be cretaed on this column
    - These indexes are called as clustered index.

DROP INDEX

```
SHOW INDEXES FROM emps;
DROP INDEX idx_emps_ename ON emps;
DROP INDEX idx_emps_deptno ON emps;
DESC emps;
```

Constraints

```
- Constraints are used to check/verify the data during DML Operations.
- It makes the DML Operations a bit slow.
- It ensures that correct data is entered into the database
- Five Constraints
    1. NOT NULL
    2. Primary Key
    3. UNIQUE
    4. Foreign Key
    5. CHECK
- Unique,Primary and Foreign Key creates indexes automatically
```

Level Of Constraint

```
a. Column Level
    All the above constraints you can apply on column level
b. Table level
    Except Not Null All the constraints you can apply on table level
```

```
CREATE TABLE customers(
id INT PRIMARY KEY,
name CHAR(20) NOT NULL,
email CHAR(50) UNIQUE,
mobile CHAR(12) UNIQUE,
addr CHAR(30)
);

CREATE TABLE customers(
id INT,
name CHAR(20) NOT NULL,
```

```
email CHAR(50),
mobile CHAR(12),
addr CHAR(30),
PRIMARY KEY(id),
UNIQUE (email),
UNIQUE(mobile)
);
```

1. NOT NULL NULL Values are not allowed in that column Not Null can be applied on column level only

```
CREATE TABLE temp1(c1 INT, c2 INT ,c3 INT NOT NULL);
INSERT INTO temp1 VALUES(1,1,1);
INSERT INTO temp1(c1,c2) VALUES(2,2); -- error
INSERT INTO temp1 VALUES(2,2,NULL); -- error

INSERT INTO temp1 VALUES(NULL,3,3);
INSERT INTO temp1 VALUES(4,NULL,4);
```

2. UNIQUE
    - It is going to keep only unique values in that column
    - Multiple NULL Values are allowed.
    - Unique Constraint creates Unique indexes automatically
    - If you create UNIQUE Constraint on two columns thne it will create a composite index.

```
    CREATE TABLE temp2(c1 INT,C2 INT,C3 INT UNIQUE);
    DESC temp2;
    SHOW INDEXES FROM temp2;

    INSERT INTO temp2 VALUES(1,1,1);
    INSERT INTO temp2 VALUES(2,2,2);
    INSERT INTO temp2 VALUES(3,3,2);
    INSERT INTO temp2 VALUES(2,3,NULL);
    INSERT INTO temp2 VALUES(3,2,NULL);

    DESC students;
    SHOW INDEXES FROM students;
    DROP TABLE students;

    CREATE TABLE students(roll INT,std INT,name char(20), marks
Decimal(5,2),UNIQUE(roll,std));
    DESC students;
    SHOW INDEXES FROM students;

    INSERT INTO students VALUES(1,1,'s1',10);
    INSERT INTO students VALUES(2,1,'s2',20);
    INSERT INTO students VALUES(3,1,'s3',30);
    INSERT INTO students VALUES(1,2,'s4',10);
    INSERT INTO students VALUES(2,2,'s5',20);
```

```
INSERT INTO students VALUES(2,2,'s6',30);
--error

INSERT INTO students VALUES(3,2,'s6',30);
```

3. Primary Key
    - This constraint is similar to that of Unique(Duplicate values are not allowed) + NOT NULL(Null values are not allowed).
    -

```
CREATE TABLE temp3(c1 INT PRIMARY KEY,C2 INT,C3 INT);
DESC temp3;
SHOW INDEXES FROM temp3;

INSERT INTO temp3 VALUES(1,1,1);
INSERT INTO temp3 VALUES(2,2,2);
INSERT INTO temp3 VALUES(1,3,3); -- Duplicate entries are not allowed
INSERT INTO temp3 VALUES(NULL,2,2); -- NUll values are not allowed.

CREATE TABLE temp4(c1 INT PRIMARY KEY,C2 INT PRIMARY KEY,C3 INT);
-- Multiple primary keys are not allowed.
CREATE TABLE temp4(c1 INT UNIQUE,C2 INT UNIQUE,C3 INT);
-- Multiple uniques are allowed

DESC students;
SHOW INDEXES FROM students;
DROP TABLE students;


-- COMPOSITE PRIMARY KEY

CREATE TABLE students(roll INT,std INT,name char(20), marks Decimal(5,2),
PRIMARY KEY(roll,std));
DESC students;
SHOW INDEXES FROM students;

INSERT INTO students VALUES(1,1,'s1',10);
INSERT INTO students VALUES(2,1,'s2',20);
INSERT INTO students VALUES(3,1,'s3',30);
INSERT INTO students VALUES(1,2,'s4',10);
INSERT INTO students VALUES(2,2,'s5',20);

INSERT INTO students VALUES(2,2,'s6',30);
--error

INSERT INTO students VALUES(3,2,'s6',30);
```

Surrogate Primary Key

```
- ORACLE -> sequence
- MYSQL -> AUTO_INCREMENT
```

```
    CREATE TABLE temp5(C1 INT PRIMARY KEY AUTO_INCREMENT ,C2 INT);
    INSERT INTO temp5(C2) VALUES(10);
    INSERT INTO temp5(C2) VALUES(20);
    INSERT INTO temp5(C2) VALUES(50);

    ALTER TABLE temp5 AUTO_INCREMENT = 100;

    INSERT INTO temp5(C2) VALUES(70);
    INSERT INTO temp5(C2) VALUES(60);

    SELECT * FROM temp5;
```

4. Foreign Key

```
DROP TABLE emps;
DROP TABLE depts;

CREATE TABLE depts(deptno INT PRIMARY KEY,dname CHAR(10));
INSERT INTO depts VALUES(10,'DEV');
INSERT INTO depts VALUES(20,'QA');
INSERT INTO depts VALUES(30,'OPS');
INSERT INTO depts VALUES(40,'ACC');

CREATE TABLE emps(empno INT ,ename CHAR(15),deptno INT,mgr INT,
FOREIGN KEY (deptno) REFERENCES depts(deptno));

DESC emps;
SHOW INDEXES FROM emps;

INSERT INTO emps VALUES(1,'Amit',10,4);
INSERT INTO emps VALUES(2,'Rahul',10,3);
INSERT INTO emps VALUES(3,'Nilesh',20,4);
INSERT INTO emps VALUES(4,'Nitin',50,3);
--error

INSERT INTO emps VALUES(4,'Nitin',40,3);
INSERT INTO emps VALUES(5,'Sarang',NULL,NULL);

DROP TABLE depts;
--error Parent table cannot be dropped before child table

DROP TABLE emps;
DROP TABLE depts;
```

```
CREATE TABLE depts(deptno INT PRIMARY KEY,dname CHAR(10));
INSERT INTO depts VALUES(10,'DEV');
INSERT INTO depts VALUES(20,'QA');
INSERT INTO depts VALUES(30,'OPS');
INSERT INTO depts VALUES(40,'ACC');

CREATE TABLE emps(empno INT ,ename CHAR(15),deptno INT,mgr INT,
FOREIGN KEY (deptno) REFERENCES depts(deptno) ON DELETE CASCADE ON UPDATE
CASCADE);
INSERT INTO emps VALUES(1,'Amit',10,4);
INSERT INTO emps VALUES(2,'Rahul',10,3);
INSERT INTO emps VALUES(3,'Nilesh',20,4);
INSERT INTO emps VALUES(4,'Nitin',40,3);
INSERT INTO emps VALUES(5,'Sarang',NULL,NULL);

UPDATE depts set deptno=50 WHERE deptno=40;
SELECT * FROM depts;
SELECT * FROM emps;

DELETE FROM depts WHERE deptno=10;
SELECT * FROM depts;
SELECT * FROM emps;

DROP TABLE depts;
--error Parent table cannot be dropped before child table
```

Homework

```
- Create table students(roll,std,name); (roll and std will be composite primary
key)
- Create table subject(rollno,std,marks);(roll and std will be composite Foreign
key)
```

5. CHECK

```
CREATE TABLE employees(
empno INT PRIMARY KEY,
ename CHAR(25) UNIQUE,
age INT NOT NULL CHECK(age>18),
sal DOUBLE NOT NULL CHECK (sal>1500)
);

INSERT INTO employees VALUES(1,'rohan',16,1600);
--error
INSERT INTO employees VALUES(1,'rohan',20,1400);
--error
INSERT INTO employees VALUES(1,'rohan',20,1600);
--OK
INSERT INTO employees VALUES(2,'Girish',NULL,1600);
```

```
--error

SHOW CREATE TABLE employees;
```

ALTER

```
CREATE TABLE dept_backup(
    deptno INT PRIMARY KEY,
    dname CHAR(30),
    loc CHAR(20)
);

CREATE TABLE emp_backup(
    empno INT,
    ename CHAR(20),
    deptno INT,
    sal DOUBLE
);

-- HOMEWORK
--Add all the data of dept table in dept_backup
--Add all the data of emp table in emp_backup

SELECT * FROM emp_backup;
DESC emp_backup;

ALTER TABLE emp_backup ADD COLUMN job CHAR(20);

UPDATE emp_backup e set e.job = (SELECT job FROM emp WHERE empno = e.empno);

ALTER TABLE emp_backup MODIFY job VARCHAR(30);

ALTER TABLE emp_backup CHANGE ename name CHAR(20);

ALTER TABLE emp_backup DROP COLUMN sal;

ALTER TABLE emp_backup ADD PRIMARY KEY(empno);

ALTER TABLE emp_backup ADD UNIQUE(name);

ALTER TABLE emp_backup ADD FOREIGN KEY(deptno) REFERENCES dept_backup(deptno);

SHOW CREATE TABLE emp_backup;

ALTER TABLE emp_backup DROP CONSTRAINT emp_backup_ibfk_1;

SHOW CREATE TABLE emp_backup;
```

Extra

```
CREATE TABLE customers(
id INT,
name CHAR(20) NOT NULL,
email CHAR(50),
mobile CHAR(12),
addr CHAR(30),
CONSTRAINT pk_id PRIMARY KEY(id),
CONSTRAINT un_email UNIQUE (email),
UNIQUE(mobile)
);


- Create table students(roll,std,name); (roll and std will be composite primary
key)
- Create table subject_marks(rollno,std,marks);(roll and std will be composite
Foreign key)

CREATE TABLE students(
    roll INT,
    std INT,
    name char(20),
    PRIMARY KEY(roll,std)
    );

CREATE TABLE subject_marks(
    roll INT,
    std INT,
    marks DECIMAL(5,2),
    FOREIGN KEY(roll,std) REFERENCES students(roll,std)
    );
```

MySQL - RDBMS

Trainer: Mr. Rohan Paramane

MySQL Programming

- RDBMS Programming is an ISO standard – part of SQL standard – since 1992.
- SQL/PSM stands for Persistent Stored Module.
- Inspired from PL/SQL - Programming language of Oracle.
- PSM allows writing programs for RDBMS. The program contains set of SQL statements along with programming constructs e.g. variables, if-else, loops, case, ...
- PSM is a block language. Blocks can be nested into another block.
- MySQL program can be a stored procedure, function or trigger.

MySQL Programming

- MySQL PSM program is written by db user (programmers).
- It is submitted from client, server check syntax & store them into db in compiled form.
- The program can be executed by db user when needed.
- Since programs are stored on server in compiled form, their execution is very fast.
- All these programs will run in server memory.

Stored Procedure

- Stored Procedure is a routine. It contains multiple SQL statements along with programming constructs.
- Procedure doesn't return any value (like void fns in C).
- Procedures can take zero or more parameters.
- Procedures are created using CREATE PROCEDURE and deleted using DROP PROCEDURE.
- Procedures are invoked/called using CALL statement.
- Result of stored procedure can be
  - returned via OUT parameter.
  - inserted into another table.
  - produced using SELECT statement (at end of SP).
- Delimiter should be set before writing procedure.

Stored Procedure

```
CREATE TABLE result(v1 DOUBLE, v2 VARCHAR(50));

DELIMITER $$

CREATE PROCEDURE sp_hello()
BEGIN
    INSERT INTO result VALUES(1, 'Hello World');
END;
$$

DELIMITER ;

CALL sp_hello();

SELECT * FROM result;
```

```
-- 01_hello.sql (using editor)
DROP PROCEDURE IF EXISTS sp_hello;
DELIMITER $$
CREATE PROCEDURE sp_hello()
BEGIN
    SELECT 1 AS v1, 'Hello World' AS v2;
END;
$$
DELIMITER ;


SOURCE /path/to/01_hello.sql

CALL sp_hello();
```

Stored Procedure – PSM Syntax

VARIABLES

```
DECLARE varname DATATYPE;
DECLARE varname DATATYPE DEFAULT init_value;
SET varname = new_value;
SELECT new_value INTO varname;
SELECT expr_or_col INTO varname FROM table_name;
```

PARAMETERS

```
CREATE PROCEDURE sp_name(PARAMTYPE p1 DATATYPE)
BEGIN
    …
END;

-- IN param: Initialized by calling program.
-- OUT param: Initialized by called procedure.
-- INOUT param: Initialized by calling program and
modified by called procedure
-- OUT & INOUT param declared as session variables.

CREATE PROCEDURE sp_name(OUT p1 INT)
BEGIN
    SELECT 1 INTO p1;
END;

SET @res = 0;
CALL sp_name(@res);
SELECT @res;
```

IF-ELSE

```
IF condition THEN
     body;
END IF;
-----------------------
IF condition THEN
     if-body;
ELSE
     else-body;
END IF;
-----------------------
IF condition THEN
     if1-body;
ELSE
     IF condition THEN
            if2-body;
     ELSE
            else2-body;
     END IF;
END IF;
-----------------------
IF condition THEN
     if1-body;
ELSEIF condition THEN
     if2-body;
ELSE
     else-body;
END IF;
```

LOOPS

```
WHILE condition DO
    body;
END WHILE;
------------------
REPEAT
    body;
UNTIL condition
END REPEAT;
------------------
label: LOOP
IF condition THEN
        ...
        LEAVE label;
    END IF;
    ...
END LOOP;
```

CASE-WHEN

```
CASE
WHEN condition THEN
      body;
WHEN condition THEN
      body;
ELSE
       body;
END CASE;
```

SHOW PROCEDURE

```
SHOW PROCEDURE STATUS
LIKE ‘sp_name’;

SHOW CREATE PROCEDURE sp_name;
```

DROP PROCEDURE

```
DROP PROCEDURE
IF EXISTS sp_name;
```

MySQL Stored Functions

- -Stored Functions are MySQL programs like stored procedures.
- -Functions can be having zero or more parameters. MySQL allows only IN params.
- -Functions must return some value using RETURN statement.
- Function entire code is stored in system table.
- Like procedures, functions allows statements like local variable declarations, if--else, case, loops, etc. One function can invoke another function/procedure and vice--versa. The functions can also be recursive.
- There are two types of functions: DETERMINISTIC and NOT DETERMINISTIC.

CREATE FUNCTION

```
CREATE FUNCTION fn_name(p1 TYPE)
RETURNS TYPE
[NOT] DETERMINISTIC
BEGIN
   body;
   RETURN value;
END;
```

SHOW FUNCTION

```
SHOW FUNCTION STATUS LIKE 'fn_name';

SHOW CREATE FUNCTION fn_name;
```

DROP FUNCTION

```
DROP FUNCTION IF EXISTS fn_name;
```

MySQL Triggers

- -Triggers are supported by all standard RDBMS like Oracle, MySQL, etc.
- Triggers are not supported by WEAK RDBMS like MS--Access.
- Triggers are not called by client's directly, so they don't have args & return value.
- Trigger execution is caused by DML operations on database.
  - BEFORE/AFTER INSERT, BEFORE/AFTER UPDATE, BEFORE/AFTER DELETE.
- Like SP/FN, Triggers may contain SQL statements with programming constructs. They may also call other SP or FN.
- However COMMIT/ROLLBACK is not allowed in triggers. They are executed in same transaction in which DML query is executed.

CREATE TRIGGER

```
CREATE TRIGGER trig_name
AFTER|BEFORE dml_op ON table
FOR EACH ROW
BEGIN
   body;
   -- use OLD & NEW keywords
   -- to access old/new rows.
   -- INSERT triggers – NEW rows.
   -- DELETE triggers – OLD rows.
END;
```

SHOW TRIGGERS

```
SHOW TRIGGERS FROM db_name;
```

DROP TRIGGER

```
DROP TRIGGER trig_name;
```

Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

Agenda

```
PL/SQL OR PSM
Triggers
```

PL/SQL OR PSM

```
- PL/SQL - Procedural Language
- PSM - Persistant Stored Modules
```

Stored Procedure

```
- In Mysql when ; is found it sends the complete statement to the server for
processing
- this ; is called as Delimiter
- For writing stored procedures you should change the delimiter temporarily
- You can change it by using DELIMITER keyword.
```

Steps For Writing the Stored Procedure.

```
1. Create a .sql file (PSM01.sql)
2. use SOURCE command to import that file into our db.
3. Call the Procedure (call <procedure_name>())
```

SHOW PROCEDURE STATUS WHERE db='classwork';

```
--PSM01 write a procedure to display hello world

--PSM02 write a procedure to insert above msg in result table
CREATE TABLE result(id INT,msg CHAR(30));

--PSM03 write a procedure to calculate area of Circle (Radius-7)

--PSM04 Write a procedure to calculate area of rectangle and insert the area in
result table. Pass the length and breadth from user.
--SET v_area = p_len * p_bre;

--PSM05 Write a procedure to calculate square of a number using IN and OUT
parameter
void square(int num1,int *res){
    *res = num1*num1;
```

```
}

int main(){
    int res;
    square(5,&res);
    printf("square = %d",res);
    return 0;
}

--PSM06 Write a procedure to calculate square of a number using INOUT parameter

--PSM07 Write a procedure to identify wether a given number is even or odd.
```

Triggers

```
DML- I can fire a trigger
Operations on a table
BEFORE INSERT
AFTER INSERT
BEFORE UPDATE
AFTER UPDATE
BEFORE DELETE
AFTER DELETE

If you do DML operations on Multiple rows then Trigger will be fired for once for
each row.

INSERT->NEW
DELETE->OLD
UPDATE->NEW,OLD

Triggers are not called by user. They are automatically fired on DML operations
```

```
CREATE TABLE accounts(
    acc_no INT PRIMARY KEY,
    type CHAR(20),
    balance DECIMAL(9,2)
);

CREATE TABLE transactions(
    tno INT PRIMARY KEY AUTO_INCREMENT,
    acc_no INT,
    type CHAR(20),
    amt DECIMAL(9,2)
);

INSERT INTO accounts VALUES(1,'SAVINGS',10000);
INSERT INTO accounts VALUES(2,'CURRENT',5000);
```

```
INSERT INTO accounts VALUES(3,'SAVINGS',20000);

INSERT INTO transactions(acc_no,type,amt) VALUES(1,'DEPOSIT',5000);
INSERT INTO transactions(acc_no,type,amt) VALUES(3,'Withdraw',5000);
```

Extra - Functions

1. Deterministic
2. NOT Deterministic

```
    --SMITH
    --CONCAT(UPPER(LEFT(ename,1)),LOWER(SUBSTR(ename,2)))
    --Smith
    --Write your function as TITLE('SMITH') -> Smith

    int add(int num1,int num2){
    int res = num1+num2;
    return res;
}
```

MySQL - RDBMS

Trainer: Mr. Rohan Paramane

Codd's rules

- Proposed by Edgar F. Codd – pioneer of the RDBMS – in 1980.
- If any DBMS follow these rules, it can be considered as RDBMS.

- The 0^{th} rule is the main rule known as "The foundation rule".
  - For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.

- The rest of rules can be considered as elaboration of this foundation rule.

Codd's rules

- Rule 1: The information rule:
  - All information in a relational data base is represented explicitly at the logical level and in exactly one way – by values in tables.

- Rule 2: The guaranteed access rule:
  - Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.

- Rule 3: Systematic treatment of null values:
  - Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.

Codd’s rules

- Rule 4: Dynamic online catalog based on the relational model:
  - The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

- Rule 5: The comprehensive data sublanguage rule:
  - A relational system may support several languages. However, there must be at least one language that supports all functionalities of a RDBMS i.e. data definition, data manipulation, integrity constraints, transaction management, authorization.

Codd's rules

- Rule 6: The view updating rule:
  - All views that are theoretically updatable are also updatable by the system.
- Rule 7: Possible for high-level insert, update, and delete:
  - The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data. Also it Should supports Union, Intersection Operation
- Rule 8: Physical data independence:
  - Application programs and terminal activities remain logically unbroken whenever any changes are made in either storage representations or access methods.
- Rule 9: Logical data independence:
  - Application programs & terminal activities remain logically unbroken when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables.

Codd's rules

- Rule 10: Integrity independence:
  - Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.

- Rule 11: Distribution independence:
  - The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only.

- Rule 12: The non-subversion rule:
  - If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time).

Normalization

- Concept of table design: Table, Structure, Data Types, Width, Constraints, Relations.
- Goals:
  - Efficient table structure.
  - Avoid data redundancy i.e. unnecessary duplication of data (to save disk space).
  - Reduce problems of insert, update & delete.
- Done from input perspective.
- Based on user requirements.
- Part of software design phase.
- View entire appln on per transaction basis & then normalize each transaction separately.
- Transaction Examples:
  - Banking, Rail Reservation, Online Shopping.

Normalization

- For given transaction make list of all the fields.
- Strive for atomicity.
- Get general description of all field properties.
- For all practical purposes we can have a single table with all the columns. Give meaningful names to the table.
- Assign datatypes and widths to all columns on the basis of general desc of fields properties.
- Remove computed columns.
- Assign primary key to the table.
- At this stage data is in un-normalized form.
- UNF is starting point of normalization.

Normalization

- UNF suffers from
  - Insert anomaly
  - Update anomaly
  - Delete anomaly

Normalization

- 1. Remove repeating group into a new table.
- 2. Key elements will be PK of new table.
- 3. (Optional) Add PK of original table to new table to give us Composite PK.
  - Repeat steps 1-3 infinitely -- to remove all repeating groups into new tables.
  - This is **1-NF**. No repeating groups present here. One to Many relationship between two tables.

Normalization

- 4. Only table with composite PK to be examined.
- 5. Those columns that are not dependent on the entire composite PK, they are to be removed into a new table.
- 6. The key elements on which the non-key elements were originally dependent, it is to be added to the new table, and it will be the PK of new table.
  - Repeat steps 4-6 infinitely -- to separate all non-key elements from all tables with composite primary key.
  - This is **2-NF**. Many-to-Many relationship.

Normalization

- 7. Only non-key elements are examined for inter-dependencies.
- 8. Inter-dependent cols that are not directly related to PK, they are to be removed into a new table.
- 9. (a) Key ele will be PK of new table.
- 9. (b) The PK of new table is to be retained in original table for relationship purposes.
  - Repeat steps 7-9 infinitely to examine all non-key eles from all tables and separate them into new table if not dependent on PK.
  - This is **3-NF**.

Normalization

- To ensure data consistency (no wrong data entered by end user).
- Separate table to be created of well-known data. So that min data will be entered by the end user.
- This is BCNF(Boyce-Codd Normal Form) or 4-NF.

De-normalization

- Normalization will yield a structure that is non-redundant.
- Having too many inter-related tables will lead to complex and inefficient queries.
- To ensure better performance of analytical queries, few rules of normalization can be compromised.
- This process is de-normalization.

Thank you!

Rohan Paramane<rohan.paramane@sunbeaminfo.com>

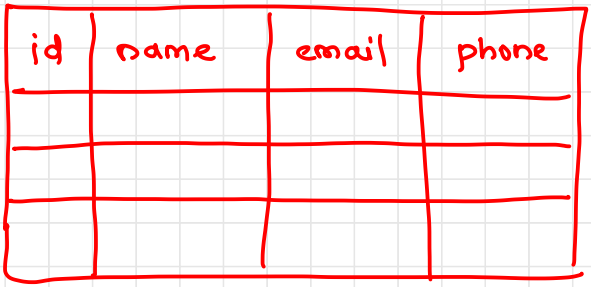

mongoDB.

What are we going to cover?

- What is NoSQL and how is it different from RDBMS?
- What is JSON?
- Introduction to MongoDB
- JSON vs BSON
- MongoDB Fundamentals
- Basic CRUD operations
- Performance optimization → indexing
- Data Modeling → json structure
- Aggregation Pipeline
- Introduction to Geospatial Queries : Geolocation

What is Database ?

- A database is an organized collection of data, generally stored and accessed electronically from a computer system
- A database refers to a set of related data and the way it is organized
- Database Management System
  - Software that allows users to interact with one or more databases and provides access to all of the data contained in the database
- Types
  - RDBMS (SQL)
  - NoSQL



structure (table)

↳ collection of row & col

RDMBS

- The idea of RDBMS was borne in 1970 by E. F. Codd
- The language used to query RDBMS systems is SQL (Sequel Query Language)
- RDBMS systems are well suited for structured data held in columns and rows, which can be queried using SQL
- Supports: DML, DQL, DDL, DTL, DCL
- The RDBMS systems are based on the concept of ACID transactions
  - **Atomic**: implies either all changes of a transaction are applied completely or not applied at all
  - **Consistent**: data is in a consistent state after the transaction is applied
  - **Isolated**: transactions that are applied to the same set of data are independent of each other
  - **Durable**: the changes are permanent in the system and will not be lost in case of any failures

Standard Query Language

- MySQL, Oracle, MS. SQL Server, PostgreSQL etc.

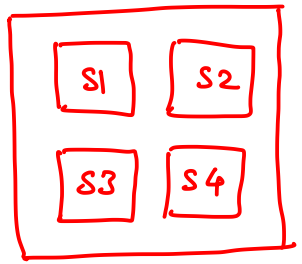
Scaling

Database clustering ⇒ NoSQL

- Scalability is the ability of a system to expand to meet your business needs
- E.g. scaling a web app is to allow more people to use your application
- Types
  - Vertical scaling
    - Add resources within the same logical unit to increase capacity
    - E.g. add CPUs to an existing server, increase memory in the system or expanding storage by adding hard drives
  - Horizontal scaling
    - Add more nodes to a system
    - E.g. adding a new computer to a distributed software application
    - Based on principle of distributed computing
- NoSQL databases are designed for Horizontal scaling *
- So they are reliable, fault tolerant, better performance (at lower cost)

upgrading / downgrading the configuration

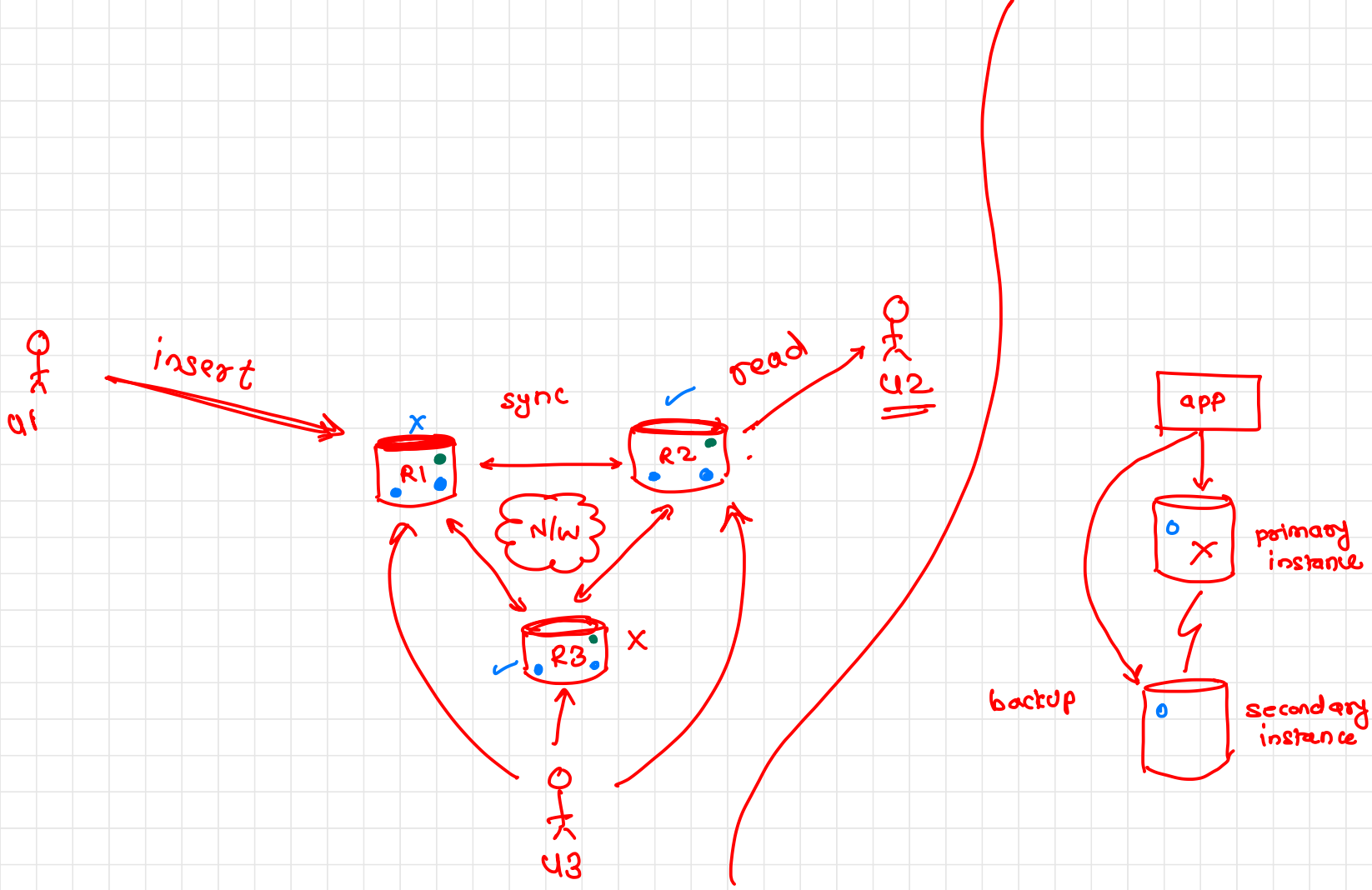
⇒ Single point of failure



Scaling

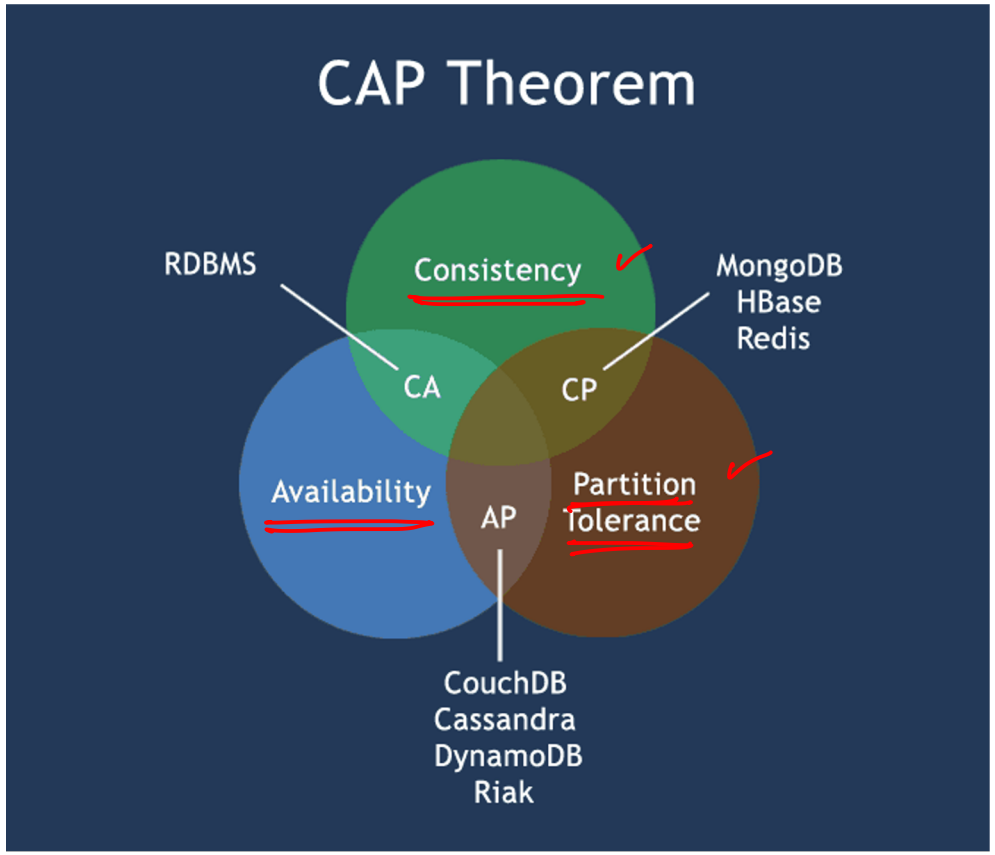
NoSQL

- Stands for Not Only SQL
- NoSQL is a term used to refer to non-relational databases
- The term NoSQL was coined by Carlo Strozzi in 1998 to name his lightweight Strozzi NoSQL open-source relational database
- Does not use any declarative query language
- No predefined schema → no restriction over schema
- Unstructured and unpredictable data
- Supports eventual consistency rather than ACID properties
- Prioritizes high performance, high availability and scalability
- In contrary to the ACID approach of traditional RDBMS systems, NoSQL solves the problem using an approach popularly called as BASE


U1
insert
x
R1
sync
R2
read
U2
N/W
R3
x
U3
app
primary instance
backup
secondary instance

CAP Theorem

- Also known as Brewer's theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees
  - Consistency
    - Data is consistent after operation
    - After an update operation, all clients see the same data
  - Availability
    - System is always on (i.e. service guarantee), no downtime
  - Partition Tolerance
    - System will continue to function even if it is partitioned into groups of servers that are not able to communicate with one another

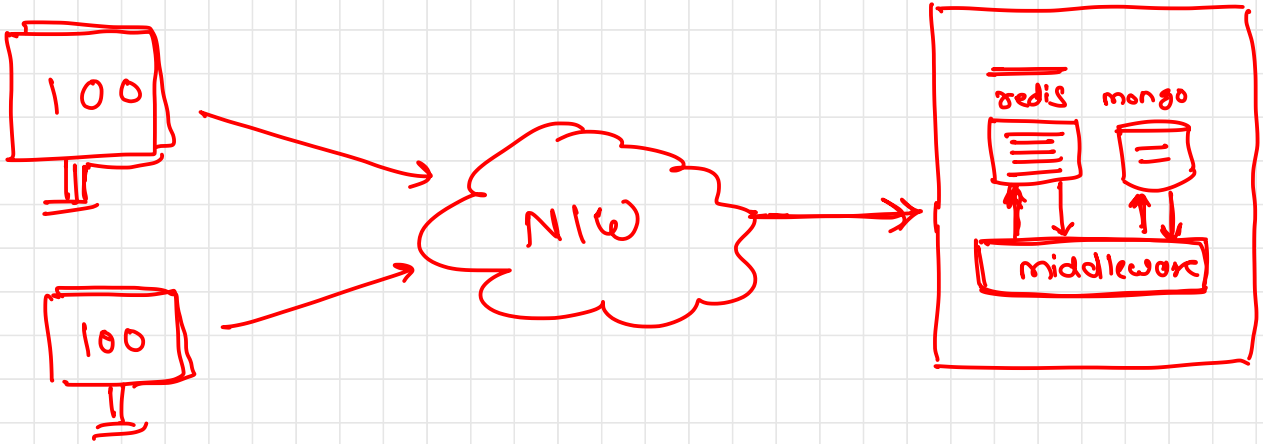


BASE Theorm

- Eric Brewer coined the BASE acronym
- BASE means
  - **Basically Available**: means the system will be available in terms of the CAP theorem.
  - **Soft state** indicates that even if no input is provided to the system, the state will change over time. This is in accordance to eventual consistency.
  - **Eventual consistency**: means the system will attain consistency in the long run, provided no input is sent to the system during that time.

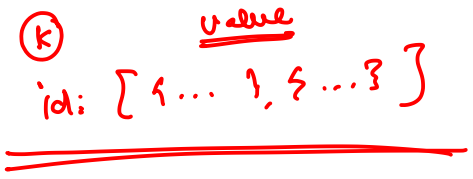
NoSQL Types

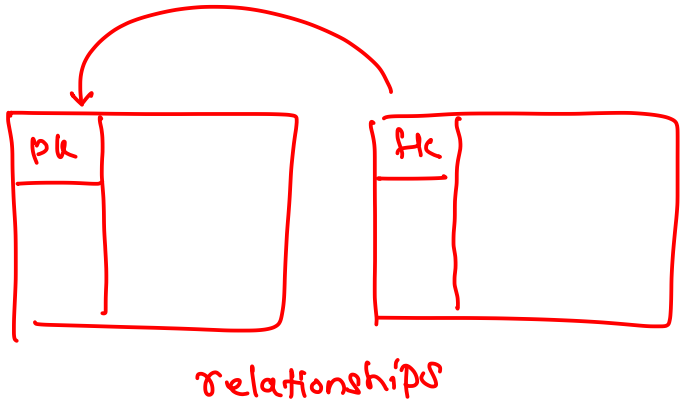
- Following are the types of NoSQL database based on how it stores the data
  - Key Value
  - Document DB
  - Column Based DB
  - Graph DB

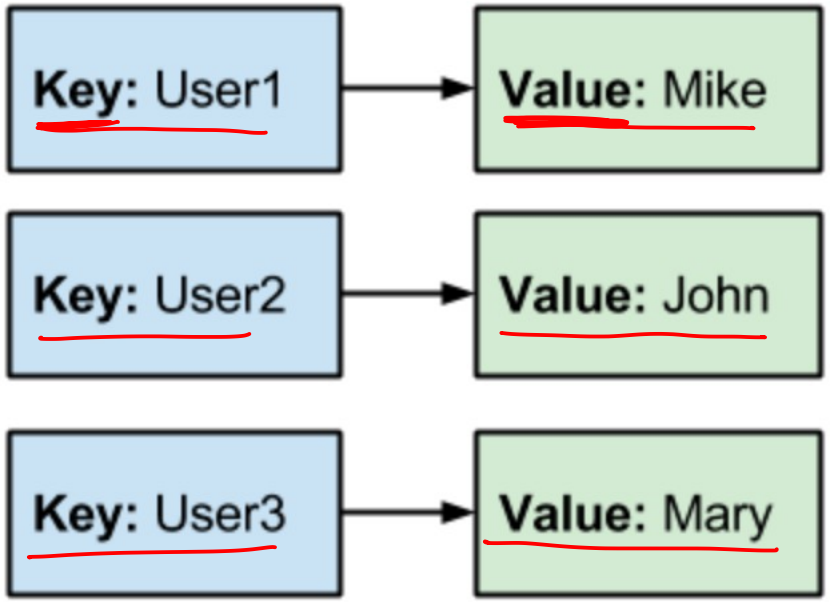

100
100
NIO
redis
mongo
middleware

Key Value Database

- Data is stored as keys and values
- Provides super fast queries and ability to scale almost infinite data and performance level
- No relationships and weak/no schema
- E.g.
  - DynamoDB : AWS - serverless
  - Redis : cache ≈ memory
  - Couchbase
  - ZooKeeper

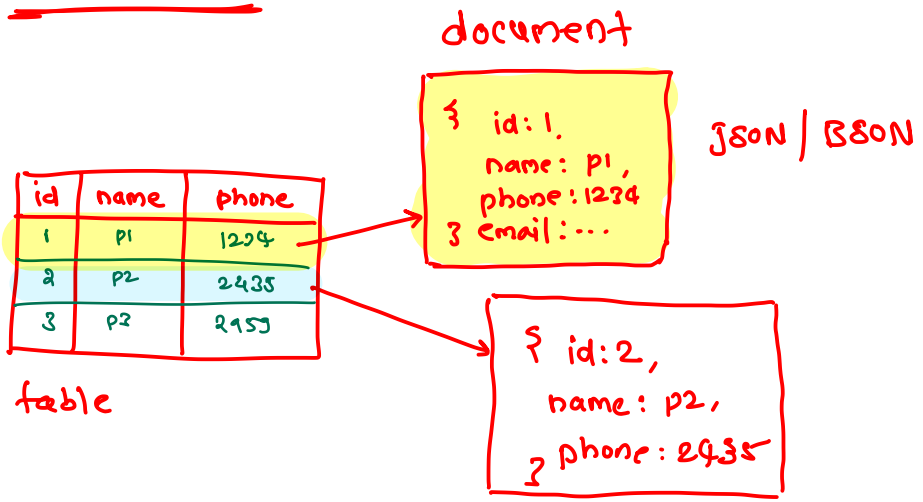


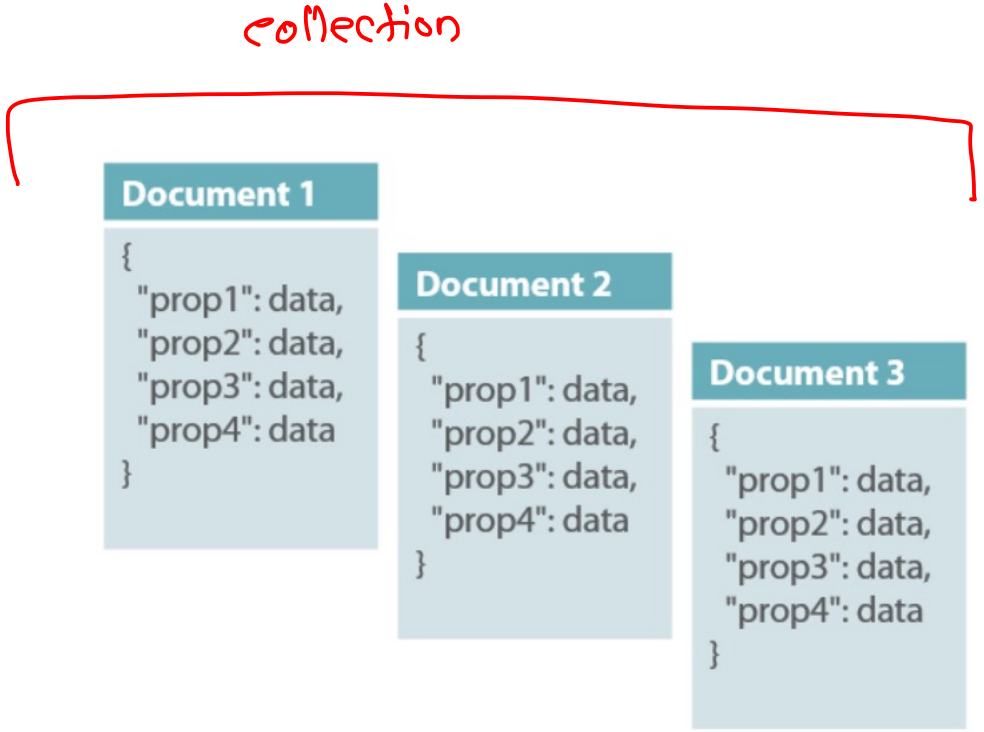




Document DB

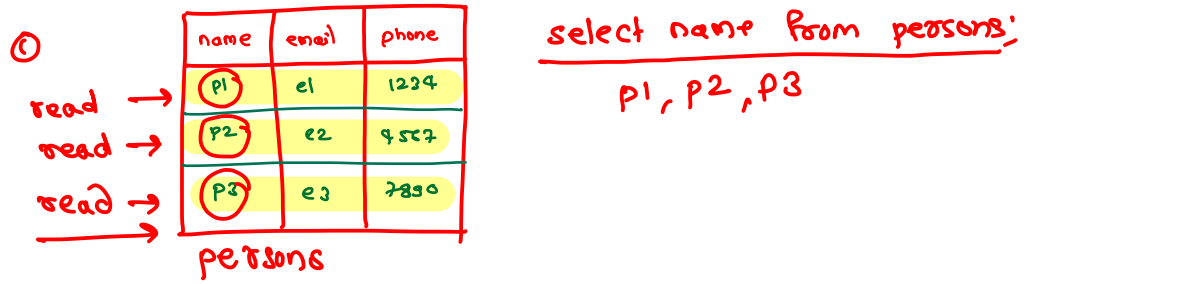
- Data is stored as Documents
- Document: collection of key-value pairs with structure
- Great performance when interacting with documents
- E.g.
  - MongoDB
  - CouchDB
  - RethinkDB

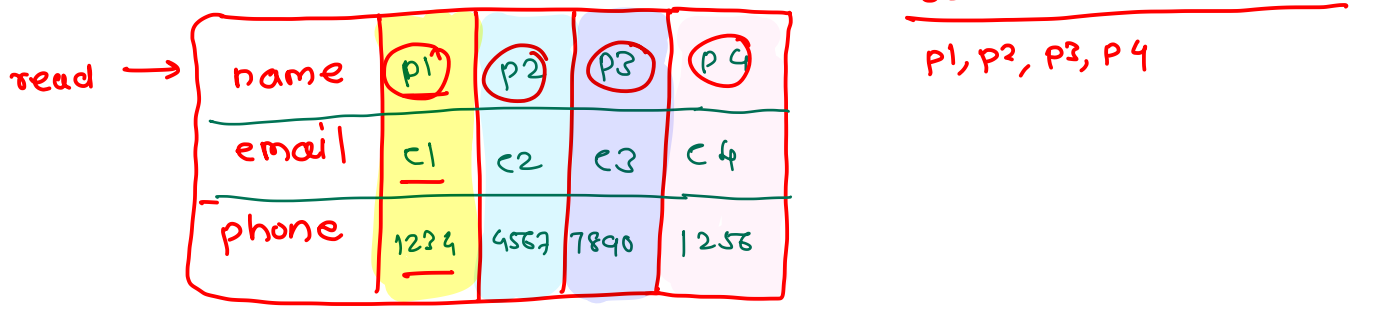




Column Based DB

- Data is stored as columns rather than rows
- Fast queries for datasets
- Slower when looking at individual records
- Great for warehousing and analytics
- E.g.
  - Redshift
  - Cassandra
  - HBase
  - Vertica

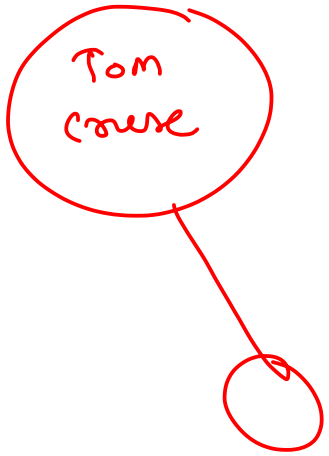




ID	Name
0001	Roffle
0002	Penny
0003	Winkie

Graph DB

- Designed for dynamic relationship
- Stores data as nodes and relationships between those nodes
- Ideal for human related data like social media
- Often as custom query language
- E.g.
  - Neo4J
  - Giraph
  - OrientDB



Advantages of NoSQL

- High scalability
  - This scaling up approach fails when the transaction rates and fast response requirements increase. In contrast to this, the new generation of NoSQL databases is designed to scale out (i.e. to expand horizontally using low-end commodity servers).
- Manageability and administration
  - NoSQL databases are designed to mostly work with automated repairs, distributed data, and simpler data models, leading to low manageability and administration.
- Low cost
  - NoSQL databases are typically designed to work with a cluster of cheap commodity servers, enabling the users to store and process more data at a low cost.
- Flexible data models
  - NoSQL databases have a very flexible data model, enabling them to work with any type of data; they don't comply with the rigid RDBMS data models. As a result, any application changes that involve updating the database schema can be easily implemented.

Disadvantages of NoSQL

- Maturity
  - Most NoSQL databases are pre-production versions with key features that are still to be implemented. Thus, when deciding on a NoSQL database, you should analyze the product properly to ensure the features are fully implemented and not still on the To-do list.
- Support
  - Support is one limitation that you need to consider. Most NoSQL databases are from start-ups which were open sourced. As a result, support is very minimal as compared to the enterprise software companies and may not have global reach or support resources.
- Limited Query Capabilities
  - Since NoSQL databases are generally developed to meet the scaling requirement of the web-scale applications, they provide limited querying capabilities. A simple querying requirement may involve significant programming expertise.
- Administration
  - Although NoSQL is designed to provide a no-admin solution, it still requires skill and effort for installing and maintaining the solution.
- Expertise
  - Since NoSQL is an evolving area, expertise on the technology is limited in the developer and administrator community.

NoSQL Scenarios

- When to use NoSQL ?
  - Large amount of data (TBs)
  - Many Read/Write operations
  - Economical scaling
  - Flexible Schema
- Examples
  - Social Media
  - Recordings
  - Geospatial analysis
  - Information processing

- When NOT to use NoSQL ?
  - Need ACID transactions
  - Fixed multiple relations
  - Need joins
  - Need high consistency
- Examples
  - Financial transactions
  - Business operations

SQL vs NoSQL

	SQL	NoSQL
Types	All types support SQL standard	Multiple types exists, such as document stores, key value stores, column databases, etc
History	Developed in 1970	Developed in 2000s
Examples	SQL Server, Oracle, MySQL	MongoDB, HBase, Cassandra
Data Storage Model	Data is stored in rows and columns in a table, where each column is of a specific type	The data model depends on the database type. It could be Key-value pairs, documents etc
Schemas	Fixed structure and schema	Dynamic schema. Structures can be accommodated
Scalability	Scale up approach is used	Scale out approach is used
Transactions	Supports ACID and transactions	Supports partitioning and availability
Consistency	Strong consistency	Dependent on the product [Eventual Consistency]
Support	High level of enterprise support	Open source model
Maturity	Have been around for a long time	Some of them are mature; others are evolving

Humongous

↳ Huge

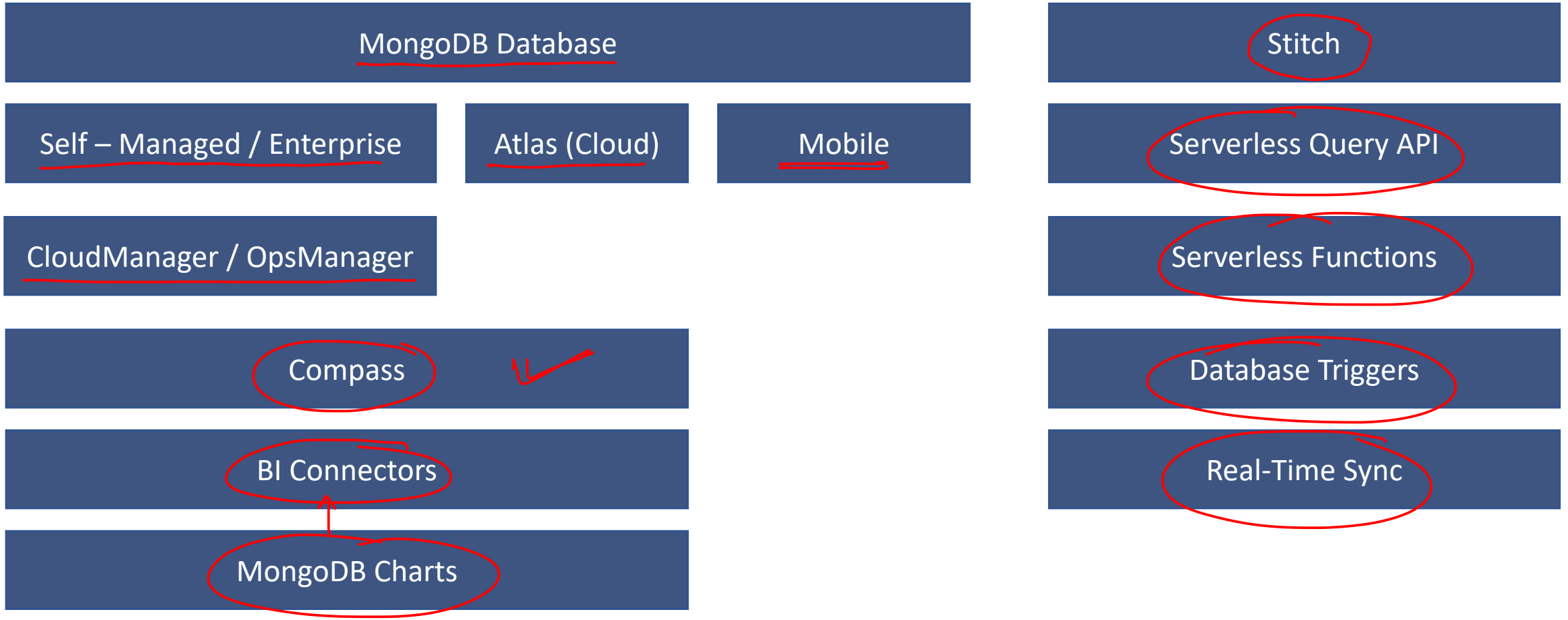
MongoDB Overview

- Developed by 10gen in 2007
- Publicly available in 2009
- Open-source database which is controlled by 10gen
- Document oriented database → stores JSON documents
- Stores data in binary JSON
- Design Philosophy
  - MongoDB wasn't designed in a lab and is instead built from the experiences of building large scale, high availability, and robust systems

Features

- **Indexing** → hashing / bucketing
  - MongoDB supports generic secondary indexes and provides unique, compound, geospatial, and full-text indexing capabilities as well
  - Secondary indexes on hierarchical structures such as nested documents and arrays are also supported and enable developers to take full advantage of the ability to model in ways that best suit their applications
- **Aggregation** *
  - MongoDB provides an aggregation framework based on the concept of data processing pipelines
  - Aggregation pipelines allow you to build complex analytics engines by processing data through a series of relatively simple stages on the server side, taking full advantage of database optimizations
- **Special collection and index types**
  - MongoDB supports time-to-live (TTL) collections for data that should expire at a certain time, such as sessions and fixed-size (capped) collections, for holding recent data, such as logs
- **File storage**
  - MongoDB supports an easy-to-use protocol for storing large files and file metadata

MongoDB Ecosystem



Install MongoDB

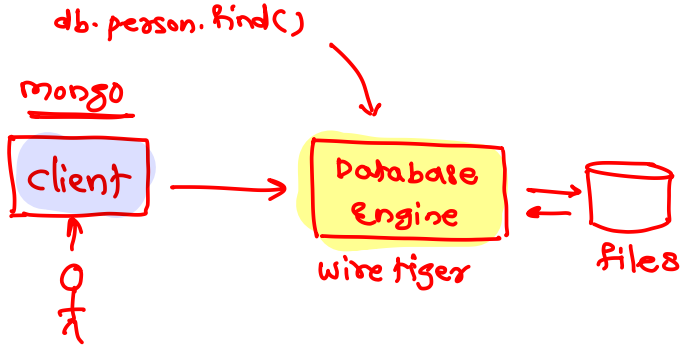
- Install MongoDB by downloading community edition (https://www.mongodb.com/download-center/community)
- Linux and Mac Users
  - Extract the downloaded file somewhere in the disk
  - Set the environment path to use the tools without going to the bin directory

    > export PATH=<path>/bin:$PATH
  - You can also include the above line in the ~/.bash_profile or ~/.basrc file
- Windows Users
  - Install the MongoDB by following all the steps in the installation wizard
  - Set the the environment path to include the <path>/bin

- Create a directory somewhere in the disk to store the data [generally on Linux or Mac, create a directory named data on the root (/): /data/]

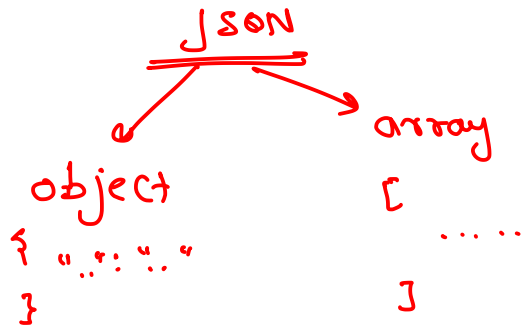
What have you downloaded?

- **mongo**: The Command Line Interface to interact with the db
- **mongod**
  - This is the database server
  - It is written in C, C++ and JS
- **mongodump**: It dumps out the Binary of the Database(BSON)
- **mongoexport**: Exports the document to Json, CSV format
- **mongoimport**: To import some data into the DB
- **mongorestore**: to restore anything that you've exported
- **mongostat**: Statistics of databases



JSON → JavaScript Object Notation

- Defined as part of the JavaScript language in the early 2000s by JavaScript creator Douglas Crockford
- It wasn't until 2013 that the format was officially specified
- JavaScript objects are simple associative containers, wherein a string key is mapped to a value
- JSON shows up in many different cases:
  - APIs
  - Configuration files
  - Log messages
  - Database storage



- However, there are several issues that make JSON less than ideal for usage inside of a database
  - JSON is a text-based format, and text parsing is very slow
  - JSON's readable format is far from space-efficient, another database concern
  - JSON only supports a limited number of basic data types

JSON

```
{
    "_id": 1,
    "name" : { "first" : "John", "last" : "Backus" },
    "contribs" : [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ],
    "awards" : [
        { "award" : "W.W. McDowell Award", "year" : 1967, "by" : "IEEE Computer Society" },
        { "award" : "Draper Prize", "year" : 1993, "by" : "National Academy of Engineering" }
    ]
}
```

Key = string

value = string / boolean / number / object / array

BSON - faster than JSON

- BSON simply stands for “Binary JSON”
- Binary structure encodes type and length information, which allows it to be parsed much more quickly
- It has been extended to add some optional non-JSON-native data types
- It allows for comparisons and calculations to happen directly on data
- MongoDB stores data in BSON format both internally, and over the network
- Anything you can represent in JSON can be natively stored in MongoDB

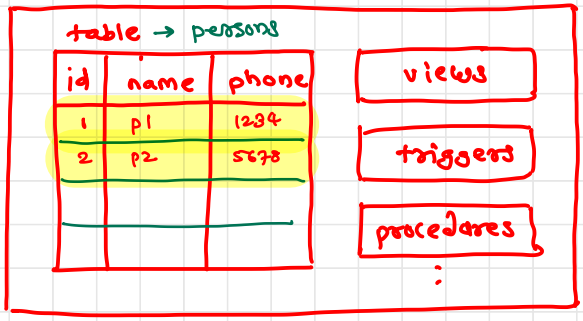
	JSON	BSON
Encoding	UTF-8 String	Binary
Data Support	String, Boolean, Number, Array	String, Boolean, Number (Integer, Float, Long, Decimal128...), Array, Date, Raw Binary
Readability	Human and Machine	Machine Only

Data Types

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server
- **Boolean** – This type is used to store a boolean (true/ false) value
- **Double** – This type is used to store floating point values
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements
- **Arrays** – This type is used to store arrays or list or multiple values into one key
- **Timestamp** – ctimestamp. This can be handy for recording when a document has been modified or added
- **Object** – This datatype is used for embedded documents
- **Null** – This type is used to store a Null value
- **Symbol** – it's generally reserved for languages that use a specific symbol type
- **Date** – This datatype is used to store the current date or time in UNIX time format
- **Object ID** – This datatype is used to store the document's ID → primary key
- **Binary data** – This datatype is used to store binary data
- **Code** – This datatype is used to store JavaScript code into the document

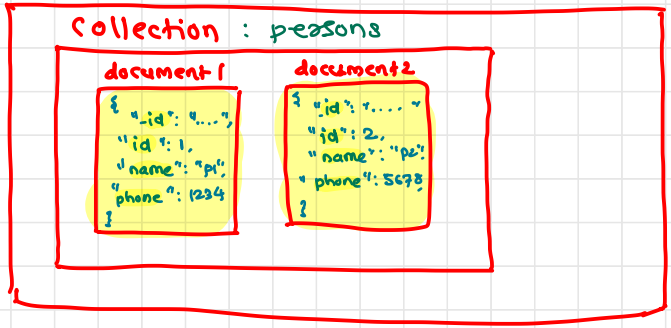
MySQL

Database : sampledb



MongoDB

Database : sampledb



Database	=	Database
table	=	Collection
column	=	attribute / field
row	=	document

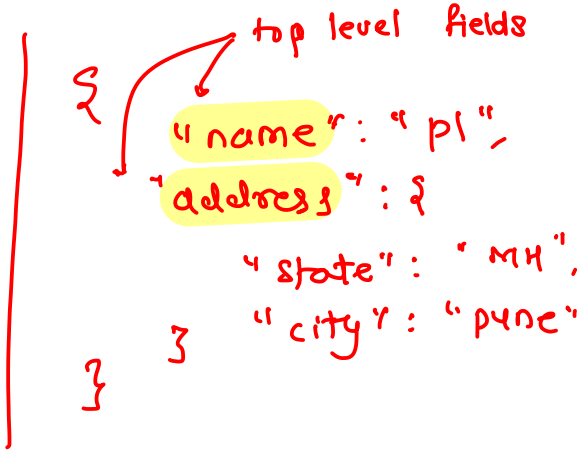
MongoDB Terminology

- **Database**
  - This is a container for collections like in RDMS wherein it is a container for tables
  - Each database gets its own set of files on the file system
  - A MongoDB server can store multiple databases
- **Collection** (table)
  - This is a grouping of MongoDB documents
  - A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL
  - Collections don't enforce any sort of structure
- **Document** (row)
  - A record in a MongoDB collection is basically called a document
  - The document, in turn, will consist of field name and values
- **Field**
  - Field names are strings
  - A name-value pair in a document
  - A document has zero or more fields
  - Fields are analogous to columns in relational databases

Document

[Row]

- MongoDB stores data records as BSON documents
- Maximum size of document is 16MB (1024 x 1024 x 16)
- Restrictions (Every document has _id to identify it uniquely.)
  - The field name _id is reserved for use as a primary key
  - Field names **cannot** contain the null character
  - Top-level field names **cannot** start with the dollar sign ($) character



_id field

- Each document requires a unique _id field that acts as a primary key
- If an inserted document omits the _id field, the MongoDB driver automatically generates an ObjectId for the _id field
- Behaviors
  - By default, MongoDB creates a unique index on the _id field during the creation of a collection
  - The _id field is always the first field in the documents. If the server receives a document that does not have the _id field first, then the server will move the field to the beginning.
  - The _id field may contain values of any BSON data type, other than an array
- Autogenerated _id (of type ObjectId) will be of 12 bytes which contains
  - Timestamp: 4 bytes
  - Machine Id: 3 bytes
  - Process Id: 2 bytes
  - Counter: 3 bytes

CRUD operations

Database Operations

- List existing databases
  - **> show dbs**
  - **> show databases**
- Create and use database
  - **> use <db name>**
- Get the selected database name
  - **> db**
- Show the database statistics
  - **> db.stats()**

Collection operations

- Get the list of collections
  > show collections

- Create Collection
  > db.createCollection('contacts')

- Drop Collection
  > db.contacts.drop()

Create Document (Insert data)

- Create one document

  > db.contacts.insert({ name: 'amit', mobile: '7709859986' })

- Create many documents

  > db.contacts.insertMany([
  { name: 'contact 1', address: 'pune' },
  { name: 'contact 2', address: 'mumbai' }
  ])

db.persons.insert({name: "person1", address: "pune"})

- **Note: if you are passing the _id field, make sure that it is unique. If it is not unique, the document will not get inserted**

Read/Find Documents (Query data)

- Find documents

  > db.contacts.find()

- Returns cursor on which following operations allowed
  - **hasNext():** returns if cursor can iterate further
  - **next():** returns the next document
  - **skip(n):** skips first n documents
  - **limit(n):** limit the result to n
  - **count():** returns the count of result
  - **toArray():** returns an array of document
  - **forEach(fn):** Iterates the cursor to apply a JavaScript function to each document from the cursor
  - **pretty():** Configures the cursor to display results in an easy-to-read format
  - **sort():** sorts documents
- Shell by default returns 20 records. Press ”it” for more results

Filtering results

> db.contacts.find({ name: 'amit' }) : select * from contacts where name = 'amit';

> db.contacts.find({ name: /amit/ }) : select * from contacts where name like '%amit%';

- Relational operators
  - $eq, $ne, $gt, $lt, $gte, $lte, $in, $nin
- Logical opeators
  - $and, $or, $nor, $not
- Element operators
  - $exists, $type
- Evaluation operators
  - $regex, $where, $mod
- Array operators
  - $size, $elemMatch, $all, $slice

Query Projection

- Projection: selecting required fields while finding the documents
- E.g.
  - db.contacts.find({}, {name: 1})
- Required fields can be added the projection list with value 1
- Non-required fields can be added the projection list with value 0
- Can not mix both required and non-required in the projection

Update Document

- Syntax
  - db.<collection>.update(criteria, newObject)
- E.g.
  - > db.contacts.update({ name: ‘amit’ }, { $set: { address: ’Pune’} })
- Update operators
  - $set, $inc, $push, $each, $pull
- In place updates are faster (e.g. $inc) than setting new object

Delete document

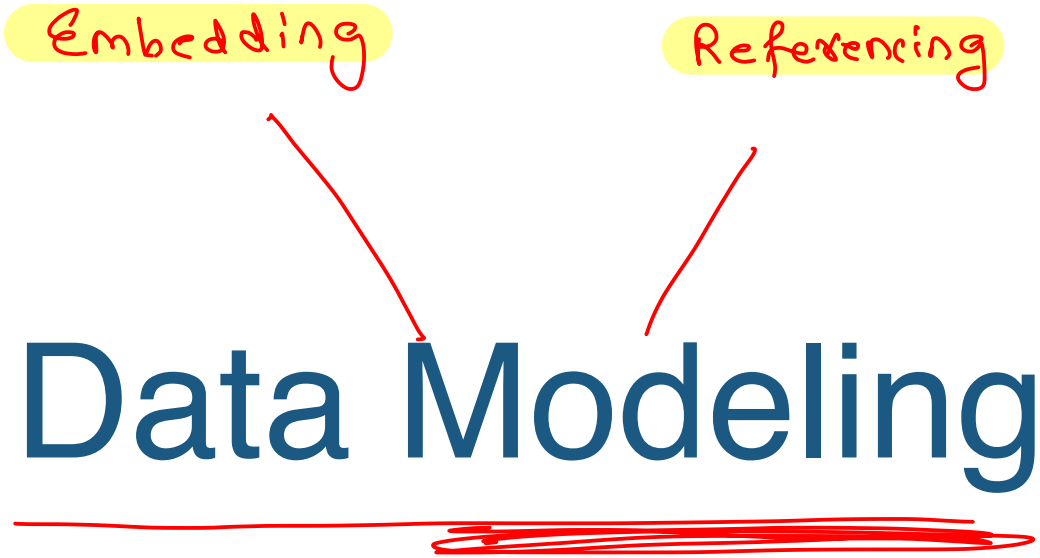
> db.contacts.remove(criteria)

> db.contacts.deleteOne(criteria);

> db.contacts.deleteMany(criteria);

> db.contacts.deleteMany({}); → delete all docs, but not collection

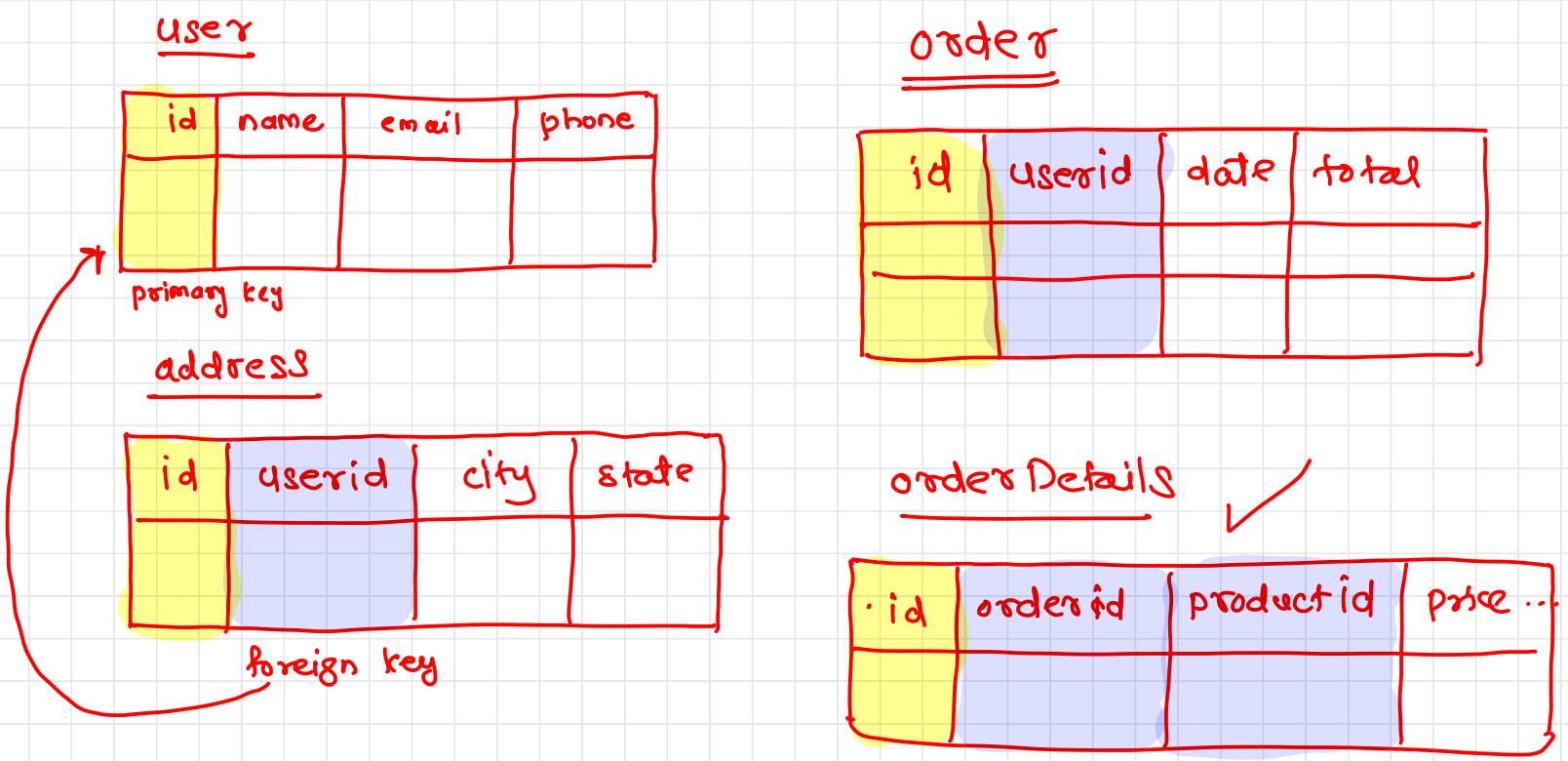
> db.contacts.drop(); → delete all docs & collection as well : efficient


Embedding
Referencing
Data Modeling

Embedded data model

- Suitable for one-to-one or one-to-many relationship
- Faster read operation
- Related data fetch in single db operation
- Atomic update of document
- Document growth reduce write performance and may lead to fragmentation

1) one user may have multiple address
2) one user may place multiple order
3) one order may contain multiple products



User may have multiple addresses

User schema

```
{
  _id: "user1",
  name: " ",
  email: " ... ",
  addresses: [
    { title: " ... ",
      city: " ... ",
      state: " .. ",
    },
    {
    }
  ]
}
```

upper-limit

Embedded data model

User may place multiple orders

user

```
{
  _id: "user1",
  name: " ... ",
  :
}
```

order

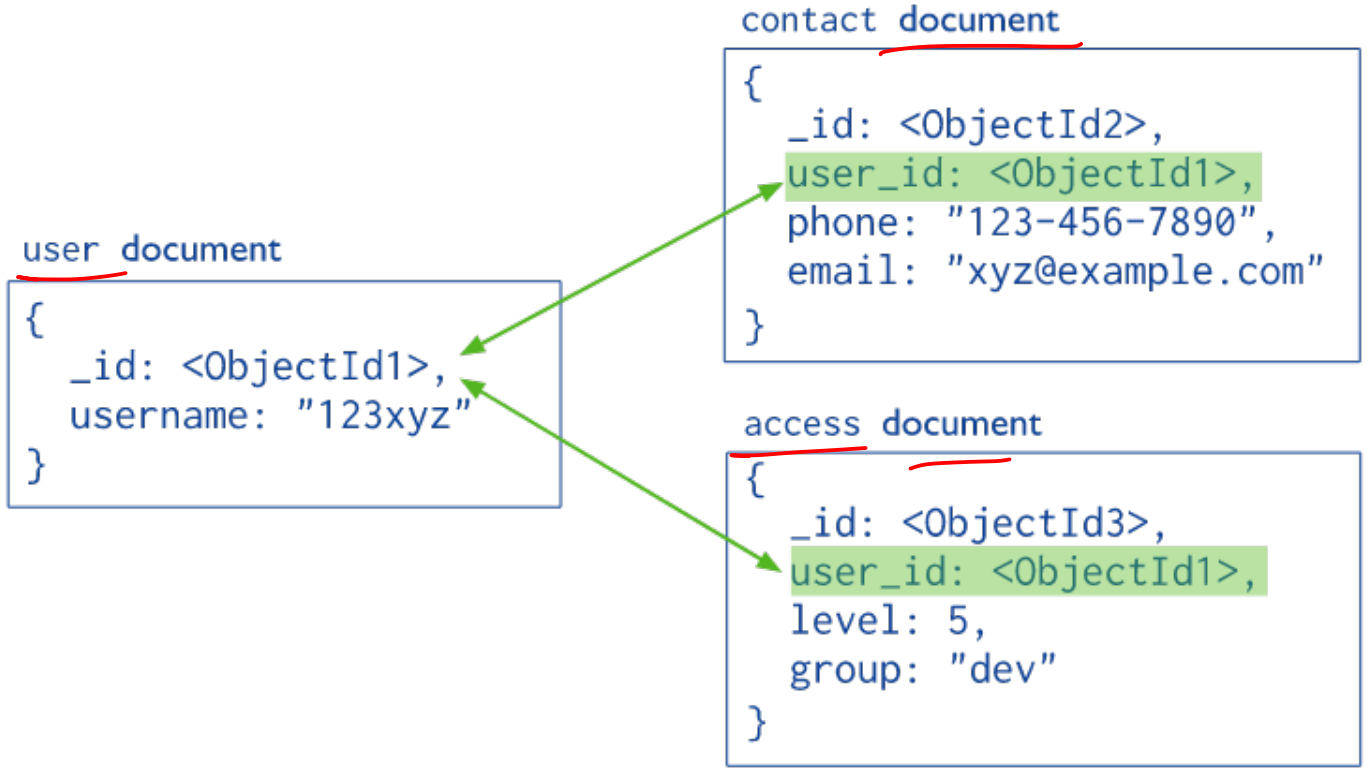
```
{ _id: "01",
  date: " ... ",
  userid: "user1",
  products: {
    {
      productid: "p1",
      quantity: 2,
      price: ...
    },
    {
      ...
    }
  }
}
```

products

```
{
  _id: "p1",
  :
}
```

Normalized data model

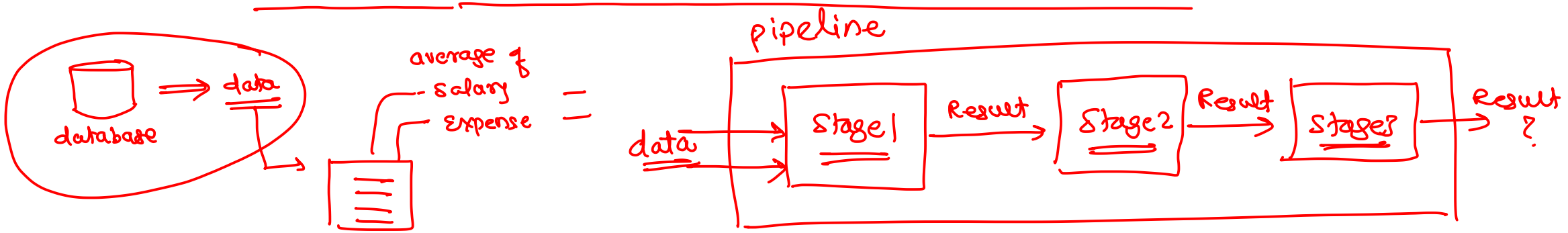
- Use normalized data model
  - to represent more complex many-to-many relationships
  - to model large hierarchical data sets
- Reduce data duplication



Aggregation Pipeline

Overview

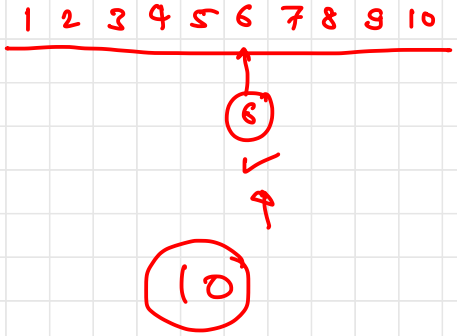
- Aggregation operations process data records and return computed results
- Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result
- Documents enter a multi-stage pipeline that transforms the documents into aggregated results
- Pipeline
  - The MongoDB aggregation pipeline consists of stages
  - Each stage transforms the documents as they pass through the pipeline
  - Pipeline stages do not need to produce one output document for every input document
    - e.g., some stages may generate new documents or filter out documents

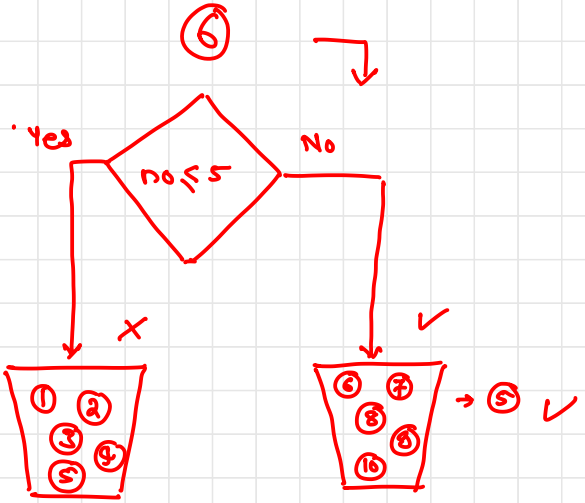


Operators

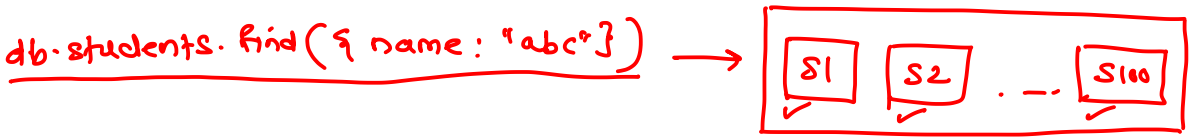
- $project: select columns (existing or computed)
- $match: where clause (criteria)
- $group: group by
  - { $group: { _id: <expr>, <field1>: { <accum1> : <expr1> }, ... } }
  - The possible accumulators are: $sum, $avg, etc.
- $unwind: extract array elements from array field
- $lookup: left outer join
- $out: put result of pipeline in another collection (last operation)

Optimization


1 2 3 4 5 6 7 8 9 10
6
10


6
Yes
no<=5
No
X
✓
1 2 3 4 5
6 7 8 9 10
5
✓

Indexes



- Indexes support the efficient execution of queries in MongoDB
- Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement
- If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect
- MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection
- Create Index

  > db.collection.createIndex(<key and index type specification>, <options>)
- Drop Index

  > db.collection.dropIndex(<key and index type>)
- Get Indexes

  > db.collection.get_indexes()
- The default name for an index is the concatenation of the indexed keys and each key's direction in the index (i.e. 1 or -1) using underscores as a separator
  - For example, an index created on { item : 1, quantity: -1 } has the name item_1_quantity_-1

Indexes - Types

- Regular index (Single Key)
- Composite index (multiple)
- Unique index
- TTL index →
- Geospatial indexes

Capped Collections

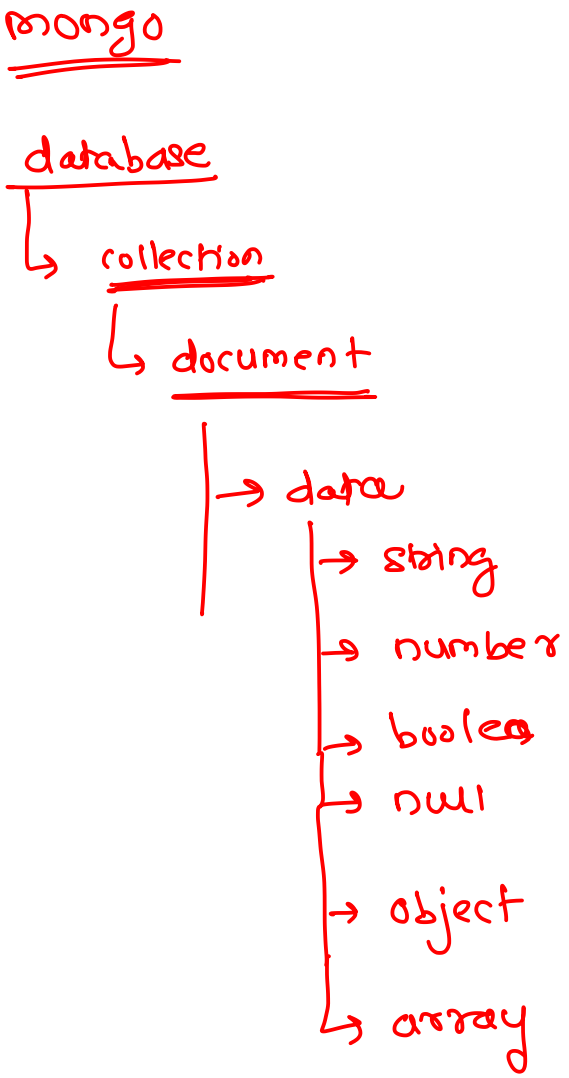
- Capped collections are fixed sized collections for high-throughput insert and retrieve operations
- They maintain the order of insertion without any indexing overhead
- The oldest documents are auto-removed to make a room for new records. The size of collection should be specified while creation
- The update operations should be done with index for better performance. If update operation change size, then operation fails
- Cannot delete records from capped collections. Can drop collection
- E.g.

```
> db.createCollection("logs", { capped: true, size: 4096 });
```

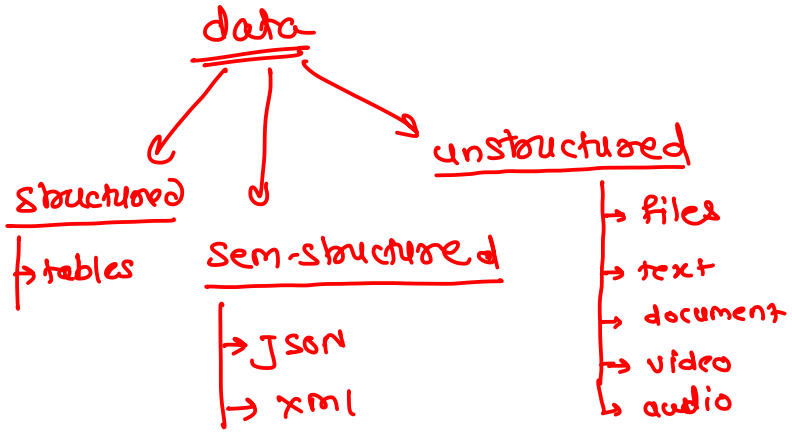
  if size is below 4096, 4096 is considered. Higher sizes are roundup by 256

- Benefits
  - Incredible read/write speed (>10000 operations per seconds)
  - Availability of cursor


mongoDB


mongo
database
collection
document
data
string
number
boolean
null
object
array

GridFS

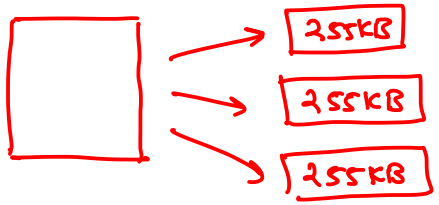

data
structured
tables
semi-structured
JSON
xml
unstructured
files
text
document
video
audio

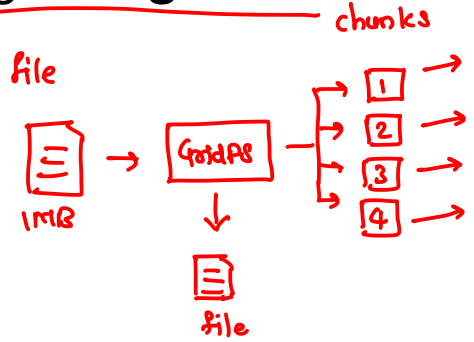
GridFS

document → 16 MB

block size → hadoop → 128MB
mongo → 255KB

- GridFS is a specification for storing/retrieving files exceeding 16 MB
- GridFS stores a file by dividing into chunks of 255 kb
- When queried back, driver collect the chunks as requested
- Query can be range query. Due to chunks file can be accessed without loading whole file in memory.
- It uses two collections for storing files i.e. fs.chunks, fs.files
- It is also useful to keep files and metadata synced and deployed automatically across geographically distributed replica set
- GridFS should not be used when there is need to update contents of entire file atomically
- It can be accessed using **mongofiles** tool or compliant client driver

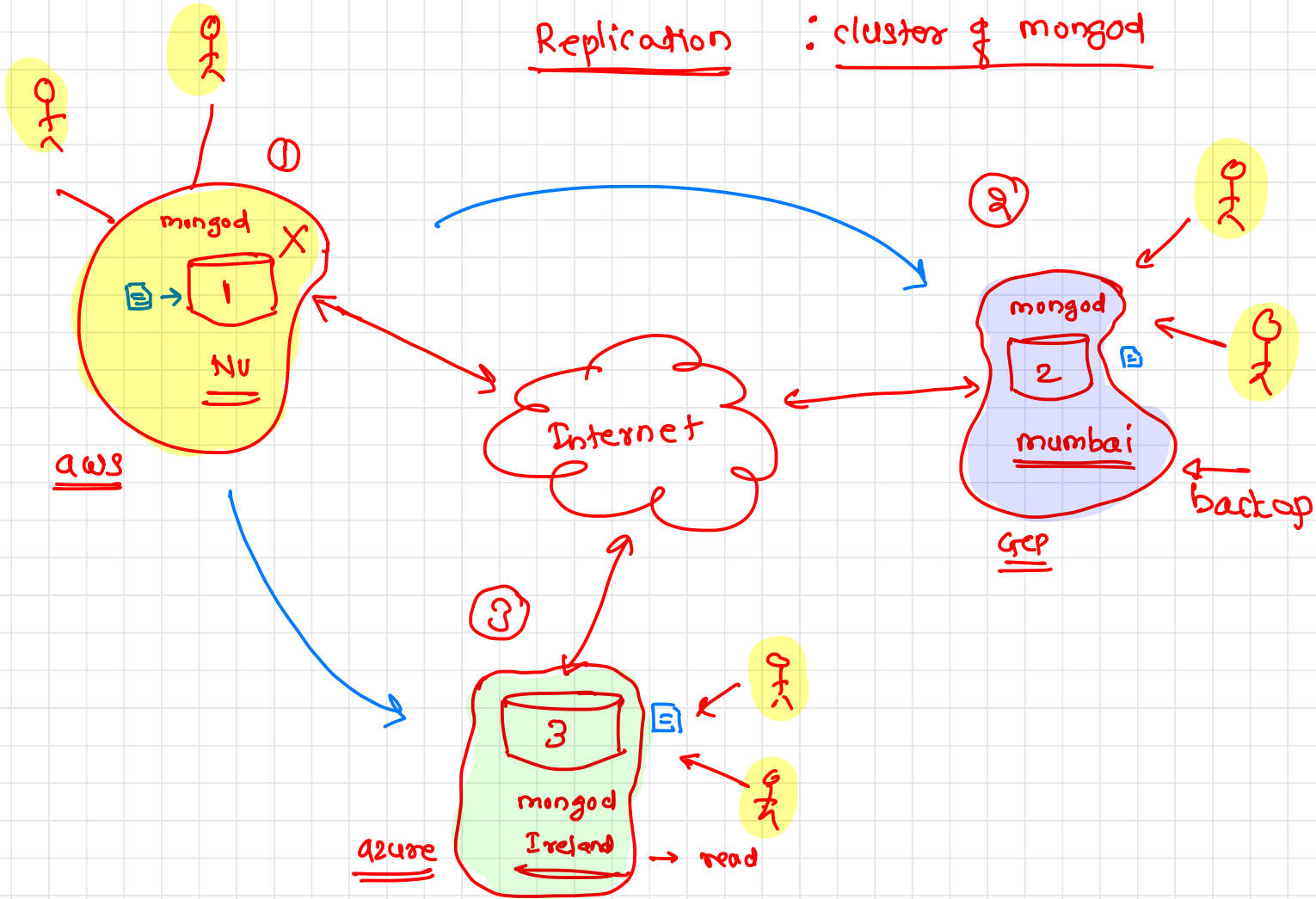




GridFS

- fs.chunks
  - _id, files_id, n, data
- fs.files
  - _id, length, chunkSize, updateDate, md5, filename, contentType
- Files can be searched using
  - db.fs.files.find({ filename: myFileName });
  - db.fs.chunks.find({ files_id: myFileID }).sort({ n: 1 })
  - GridFS automatically create indexes for faster search

Replica Set


Replication : cluster & mongod
mongod
1
NU
aws
Internet
mongod
2
mumbai
backup
GCP
3
mongod
Ireland
read
azure

Replica Sets

same data is copied on to multiple locations

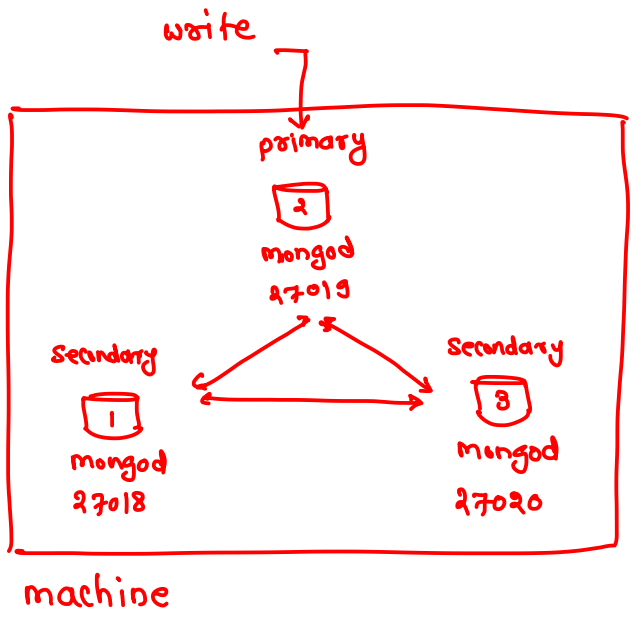
if one of the servers is not working, still the data can be retrieved from another instance

- Replica sets provide redundancy and high availability
- These are the basis for all production deployments
- With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server
- In some cases, replication can provide increased read capacity as clients can send read operations to different servers
- Maintaining copies of data in different data centers can increase data locality and availability for distributed application
- You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.ns

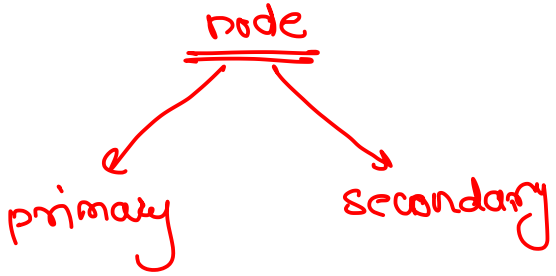
Replica Set in MongoDB

mongod → mongo daemon (database server)

- A replica set in MongoDB is a group of mongod processes that maintain the same data set
- A replica set contains several data bearing nodes and optionally one arbiter node
- Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes

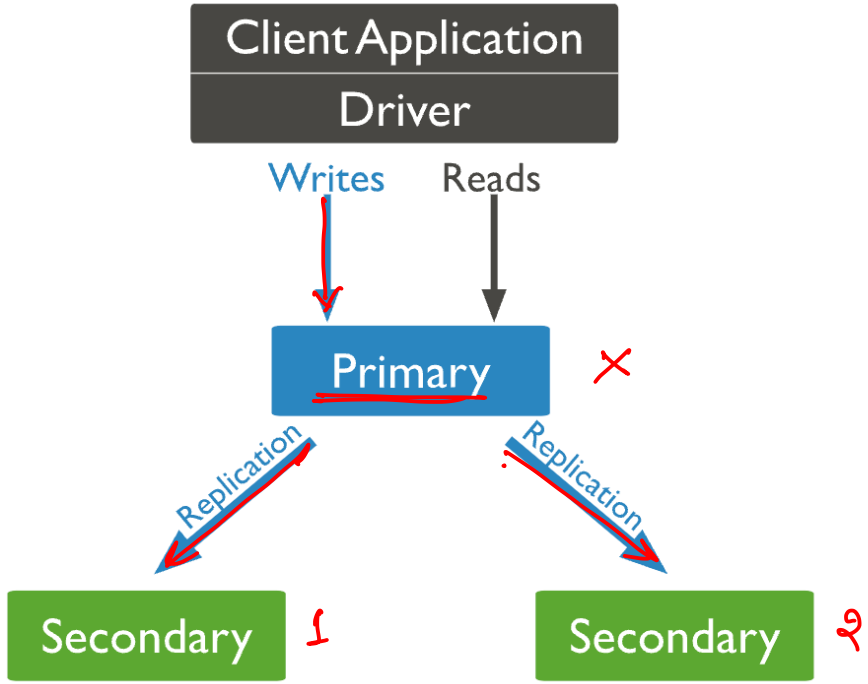


node = machine



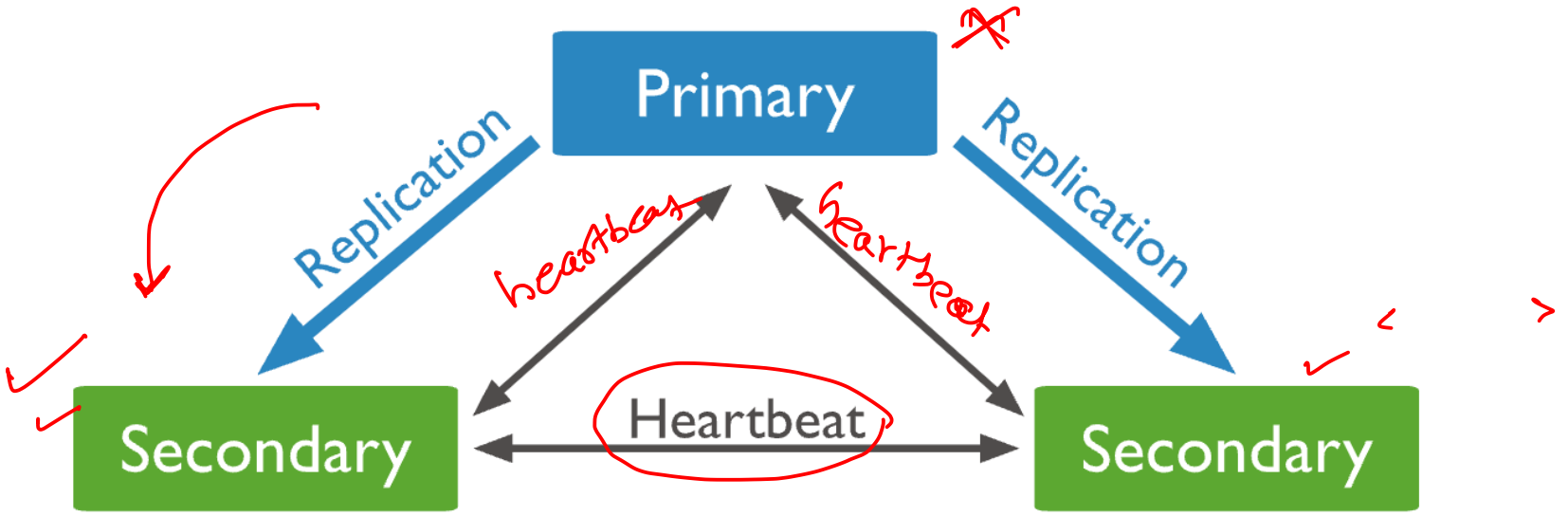
Primary Node

- The primary node receives all write operations
- A replica set can have only one primary capable of confirming writes with { w: "majority" } write concern
- Although in some circumstances, another mongod instance may transiently believe itself to also be primary.



Secondary Node

- The secondaries replicate the primary's oplog and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set
- If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary



Advantages

- In case if one of the nodes is failed, there will be always a backup with you
- There wont be any downtime
- Along with the backup you also will have an option of load balancing

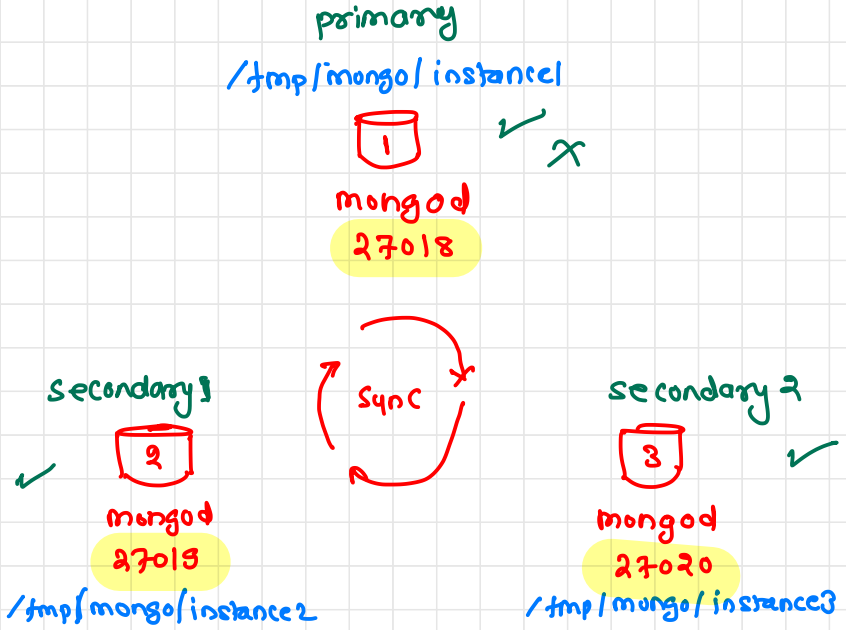
mongod

- daemon
- database server
- default port 27017

steps:

1) start mongod processes (on same / different machines)
2) add secondary node in the cluster

cluster name
- myCluster

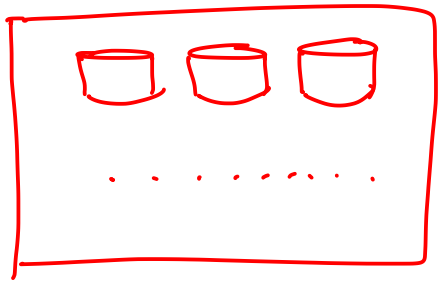


partitioning data

Sharding

Sharding

- Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth
- As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput
- Sharding solves the problem with horizontal scaling
- With sharding, you add more machines to support data growth and the demands of read and write operations

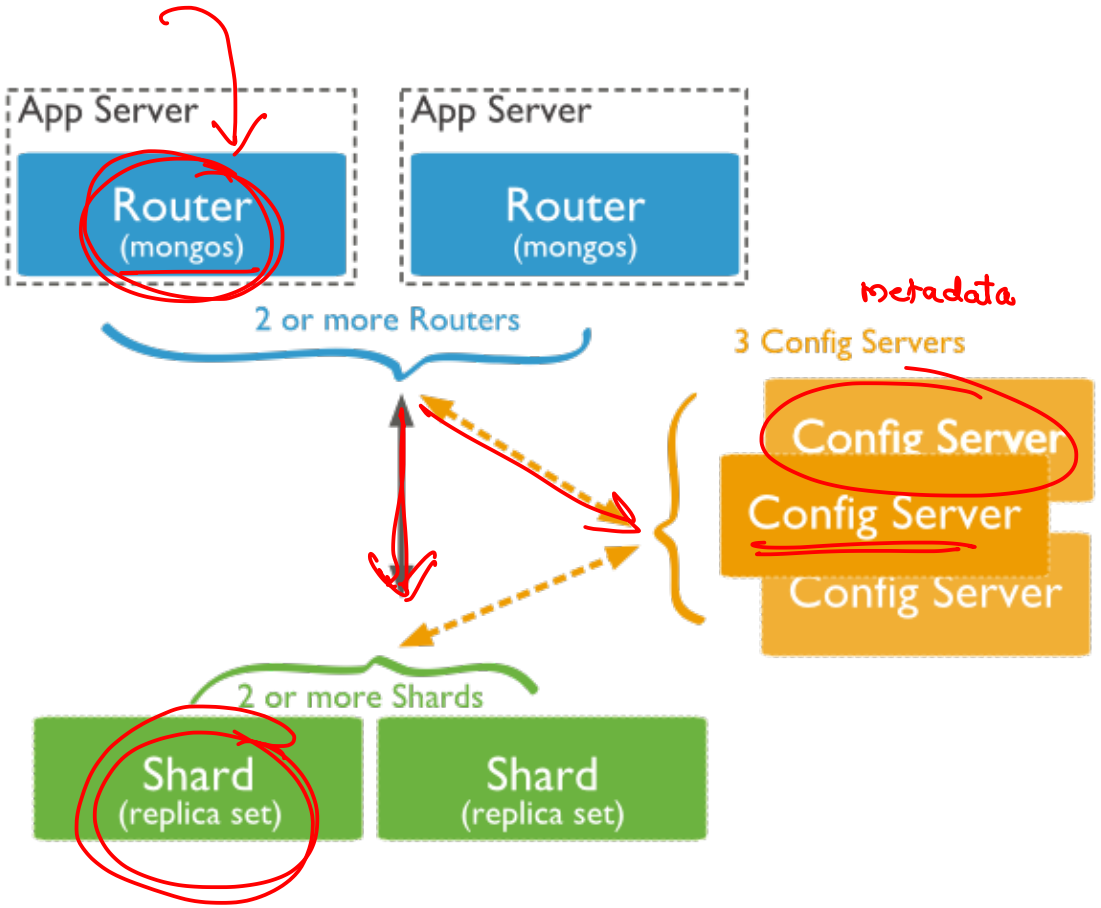


Why Sharding ?

- In replication, all writes go to master node
- Latency sensitive queries still go to master
- Single replica set has limitation of 12 nodes
- Memory can't be large enough when active dataset is big
- Local disk is not big enough
- Vertical scaling is too expensive

Sharding in MongoDB

- **Shards** [part of database]
  - Shards are used to store data
  - They provide high availability and data consistency
  - In production environment, each shard is a separate replica set
- **Config Servers**
  - Config servers store the cluster's metadata
  - This data contains a mapping of the cluster's data set to the shards
  - The query router uses this metadata to target operations to specific shards
  - In production environment, sharded clusters have exactly 3 config servers



Sharding in MongoDB

- **Query Routers**
  - Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard
  - The query router processes and targets the operations to shards and then returns results to the clients
  - A sharded cluster can contain more than one query router to divide the client request load
  - A client sends requests to one query router
  - Generally, a sharded cluster have many query routers

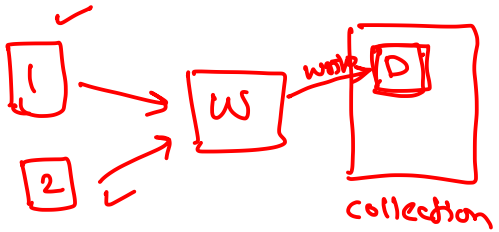
mySQL → InnoDB

mongo → WiredTiger

Storage Engine

WiredTiger

- Storage engine is managing data in memory and on disk
- MongoDB 3.2 onwards default storage engine is WiredTiger; while earlier version it was MMAPv1
- Uses document level optimistic locking for better performance
- Per operation a snapshot is created from consistent data in memory
- The snapshot is written on disk, known as checkpoint > for recovery
- Checkpoints are created per 60 secs or 2GB of journal data
- Old checkpoint is released, when new checkpoint is written on disk and updated in system tables
- To recover changes after checkpoint, enable journaling
- WT uses write-ahead transaction in journal log to ensure durability
- It creates one journal record for each client initiated write operation
- Journal persists all data modifications between checkpoints
- Journals are in-memory buffers that are synced on disk per 50 ms



oplog

WiredTiger

- Wired Tiger stores all collections & journals in compressed form
- Recovery process with journaling
  - Get last checkpoint id from data files
  - Search in journal file for records matching last checkpoint
  - Apply operations in journal since last checkpoint
- WiredTiger use internal cache with size max of 256 MB and (50% RAM 1GB) along with file system cache

mongo

→ NoSQL database vs RDBMS

→ fundamentals: database, collection, document, field

→ CRUD: insert / insertMany, find / findOne, update / updateMany, delete / deleteMany

→ aggregation pipeline

→ ReplicaSets

→ Sharding

→ Optimization → Index

→ WiredTiger

python

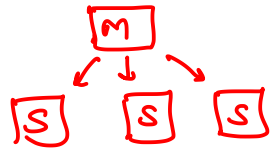
→ pymongo

CQL

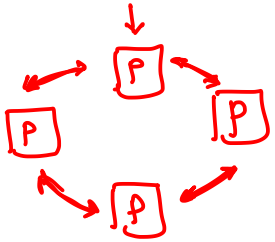
Introduction

managed service → managed by cloud provider

- **Google BigTable** ✓ Google File System – Serverless
  - High performance data storage system built on GFS and other Google technologies
  - Master-slave architecture
  - One key, multiple values
  - Columnar, SSTable (Sorted String Table) Storage, Append-only, Memtable, Compaction



- **Amazon DynamoDb** ✓ – serverless
  - Highly available and scalable key-value storage system
  - Decentralized peer to peer architecture
  - Compompromise on consistency for better availability - Eventual consistency
  - Consistent hashing, Gossip protocol, Replication, Read repair



- **Cassandra**
  - Inherited from BigTable and DynamoDb
  - BigTable: Column families, Memtable, SSTable
  - DynamoDb: Consistent hashing, Partitioning, Replication

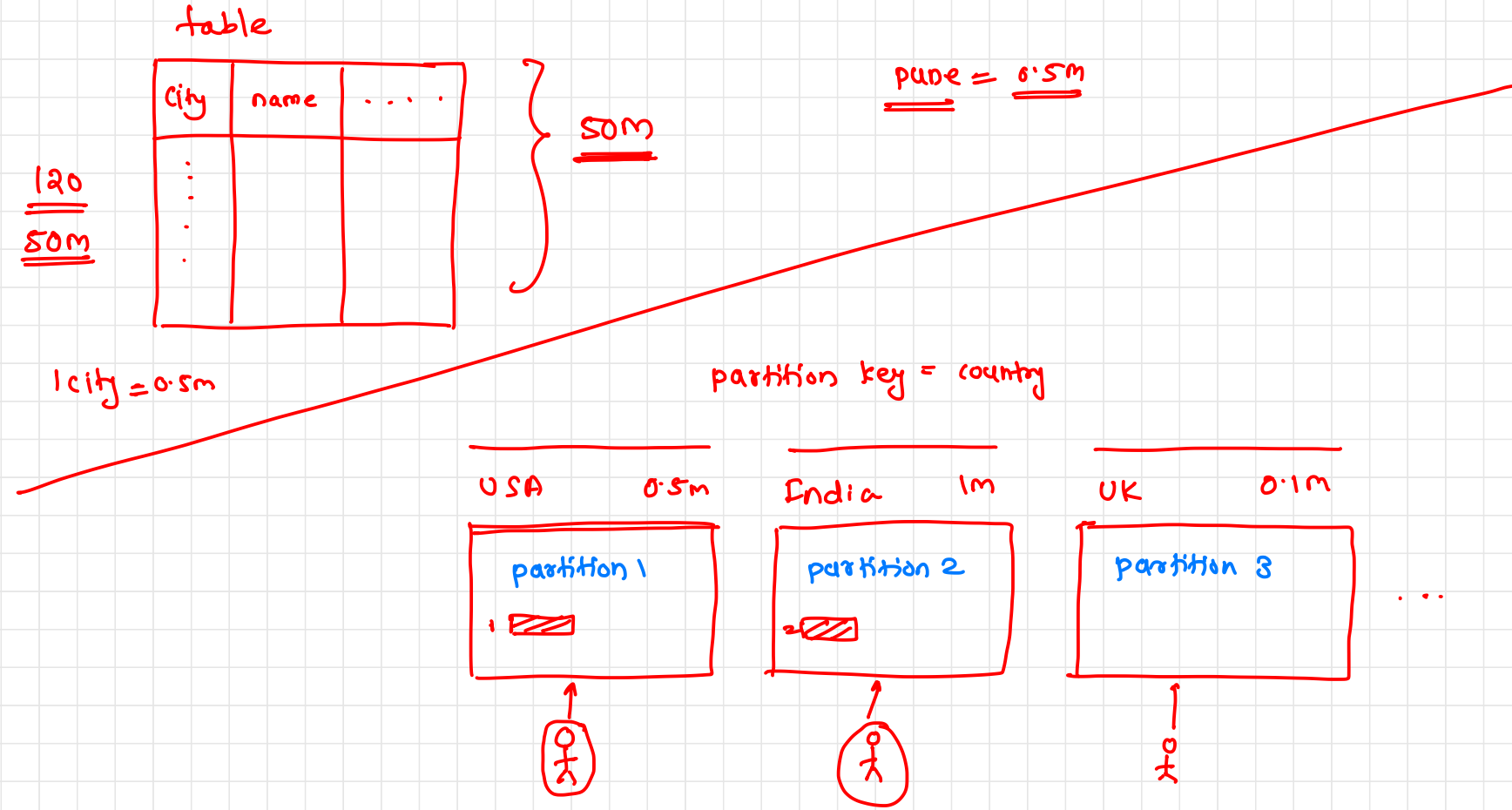
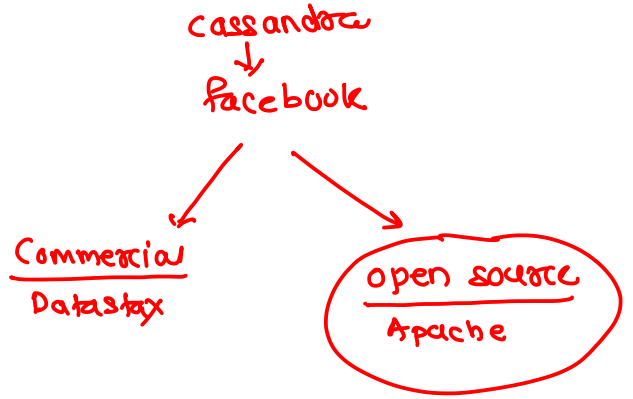


table
city
name
. . . .
50M
120
50M
1 city = 0.5m
pune = 0.5m
partition key = country
USA
0.5m
India
1m
UK
0.1m
partition 1
partition 2
partition 3
. . .

Introduction

- Developed by
  - Avinash Laxman (Co-inventor Amazon DynamoDb)
  - Prashant Malik (Technical Leader at Facebook)
- Goals:
  - Distributed NoSQL database (on commodity hardware)
  - Large amount of structured data
  - High availability
  - No single point of failure
- Basic data model is rows & columns
- Column-oriented, Decentralized peer to peer & follow Eventual consistency
- Datastax company develop and support commercial edition of Cassandra



like MySQL

table
rows
columns

Cassandra Development

- Developed in Java
- 2007-2008 - Developed at Facebook
- July 2008 - Open sourced by Facebook
- March 2009 - Apache Incubator project
- February 2010 - Apache Top-level project
- 2011 - version 0.8 - Added CQL – Cassandra Query Language
- 2013 - version 2.0 - Added light-weight transactions, Triggers
- 2015 - version 3.0 - Storage engine improved, Materialized views
- 2020 - version 3.11 - Latest release

Installation

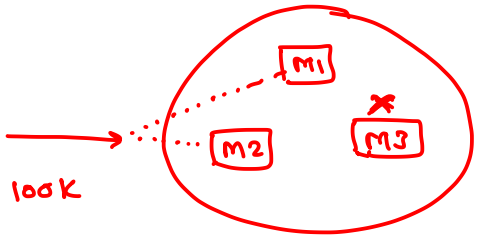
- Prerequisite
  - Java 8 (Java 11 experimental)
- Can be installed through apt or yum tool (Ubuntu/CentOS)
- Manual installation
  - Download Cassandra 3.11.x (.tar.gz) and extract it
  - set CASSANDRA_HOME to Cassandra directory
  - set JAVA_HOME to JDK 8 directory
  - Install python 2.7 (for cqlsh)
  - set CASSANDRA_HOME/bin into PATH variable
  - Start Cassandra
    - terminal1> cassandra → starts cassandra server
    - terminal2> cqlsh → starts shell

macOS
↳ brew

1) MySQL : 3306
2) MongoDB : 27017
3) Redis : 6379
4) Cassandra : 9042

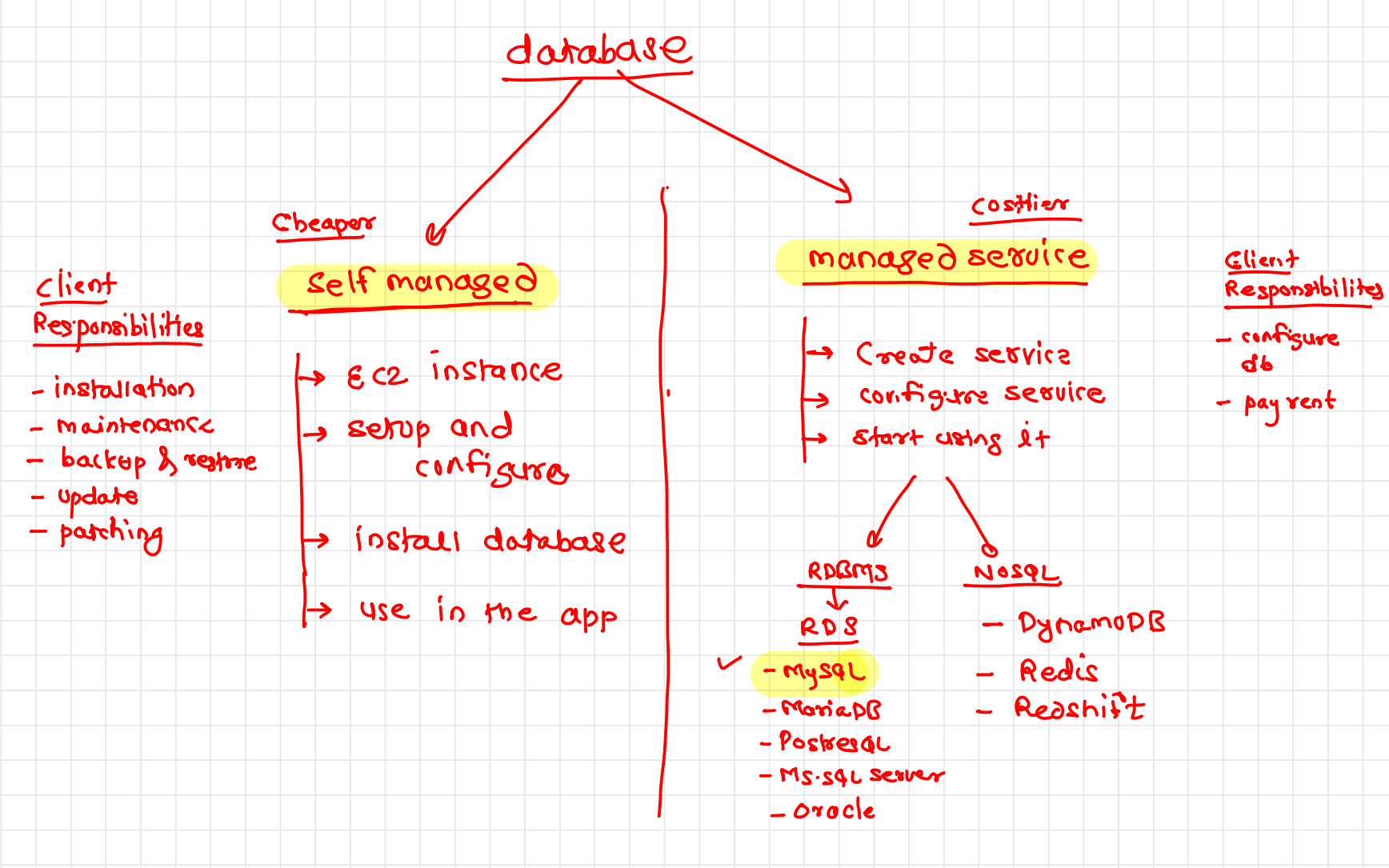
Features

- Peer to peer architecture
- Linear scale performance
- High Performance
- Simplified deployment and maintenance : managed service on cloud provider
- Less expensive
- Supports multiple programming languages : Java, C#, Python, C++ ....
- Operational and Development simplicity :
- Cloud Availability
- Ability to deploy across data centers
- Fault tolerant :
- Configurable consistency (tight or eventual)
- Flexible data model : NoSQL
- Column family store



Limitations

- Aggregation operations are not supported
- Range queries on partition key are not supported
- Not good for too many joins
- Not suitable for transactional data
- During compaction performance / throughput slows down
- Not designed for update-delete


database
Cheaper
Self managed
Client Responsibilities
- installation
- maintenance
- backup & restore
- update
- patching
→ EC2 instance
→ setup and configure
→ install database
→ use in the app
costlier
managed service
Client Responsibilities
- configure db
- payment
→ Create service
→ configure service
→ start using it
RDBMS
RDS
- MySQL
- MariaDB
- PostgreSQL
- Ms-sql server
- Oracle
NoSQL
- DynamoDB
- Redis
- Redshift

Performance

- Performance measures
  - Throughput (operations per second)
  - Latency
- Cassandra vs MySQL
  - MySQL (more than 50GB data)
    - Write speed: 300 ms
    - Read speed: 350 ms
  - Cassandra (more than 50GB data)
    - Write speed: 0.12 ms
    - Read speed: 15 ms

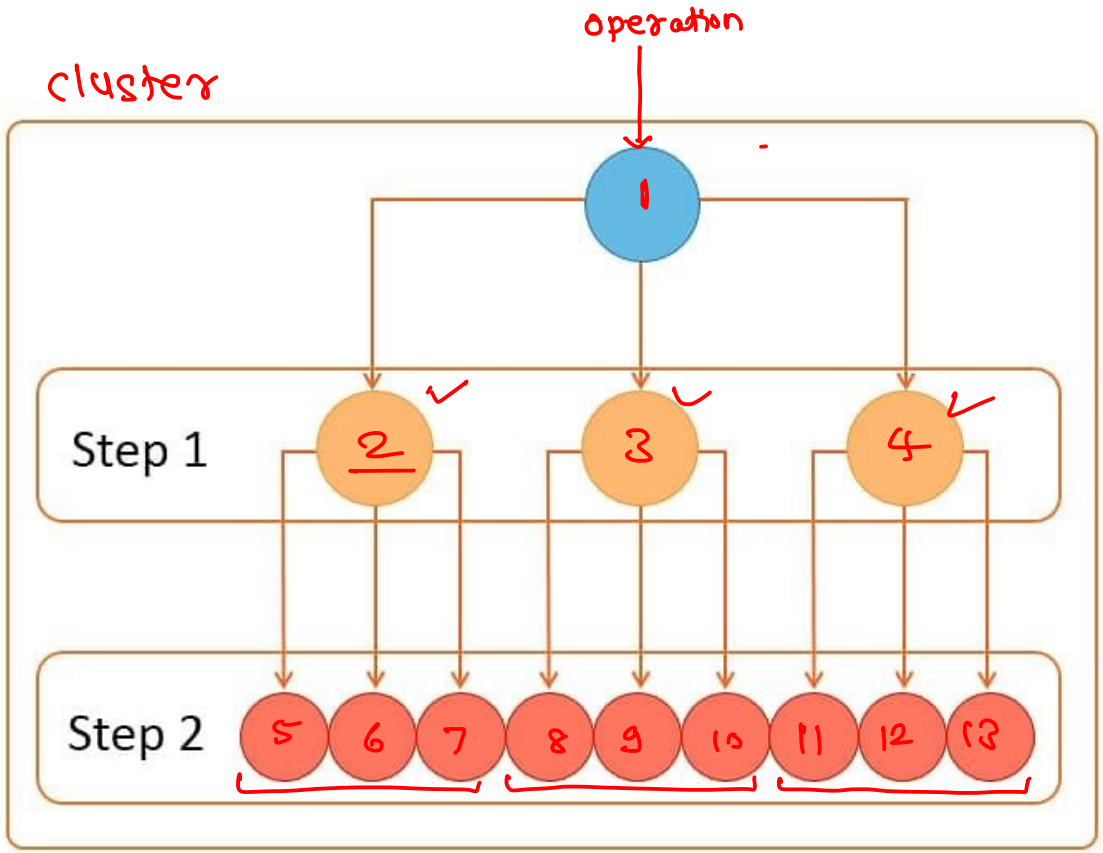
Applications

- Applications
  - Product catalog/Playlist
  - Recommendation/Personalization engine
  - Sensor/IoT data
  - Messaging/Time-series data
  - Fraud detection
- Customers
  - Facebook, Netflix, eBay, Apple, Walmart, GoDaddy
- Application requirements
  - Store and handle time-series data
  - Store and handle large volume of data
  - Scale predictably (Linear Scaling)
  - High availability

Data processing

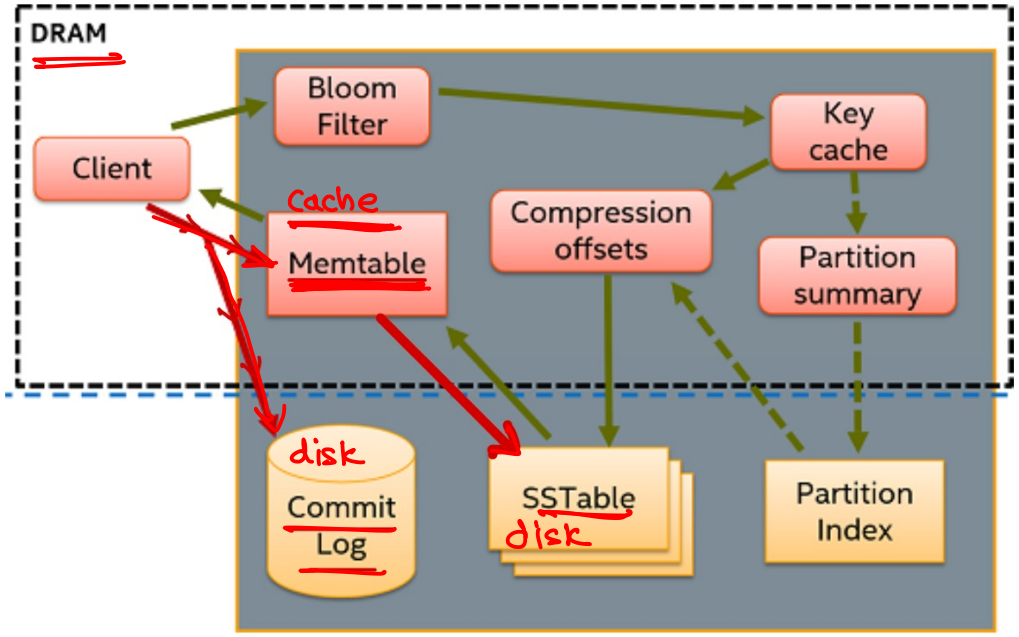
Gossip Protocol

- Cassandra uses a gossip protocol to communicate with nodes in a cluster
- It is an inter-node communication mechanism similar to the heartbeat protocol in Hadoop
- Cassandra uses the gossip protocol to discover the location of other nodes in the cluster and get state information of other nodes in the cluster
- The gossip process runs periodically on each node and exchanges state information with three other nodes in the cluster
- Eventually, information is propagated to all cluster nodes. Even if there are 1000s of nodes, information is propagated to all the nodes within a few seconds



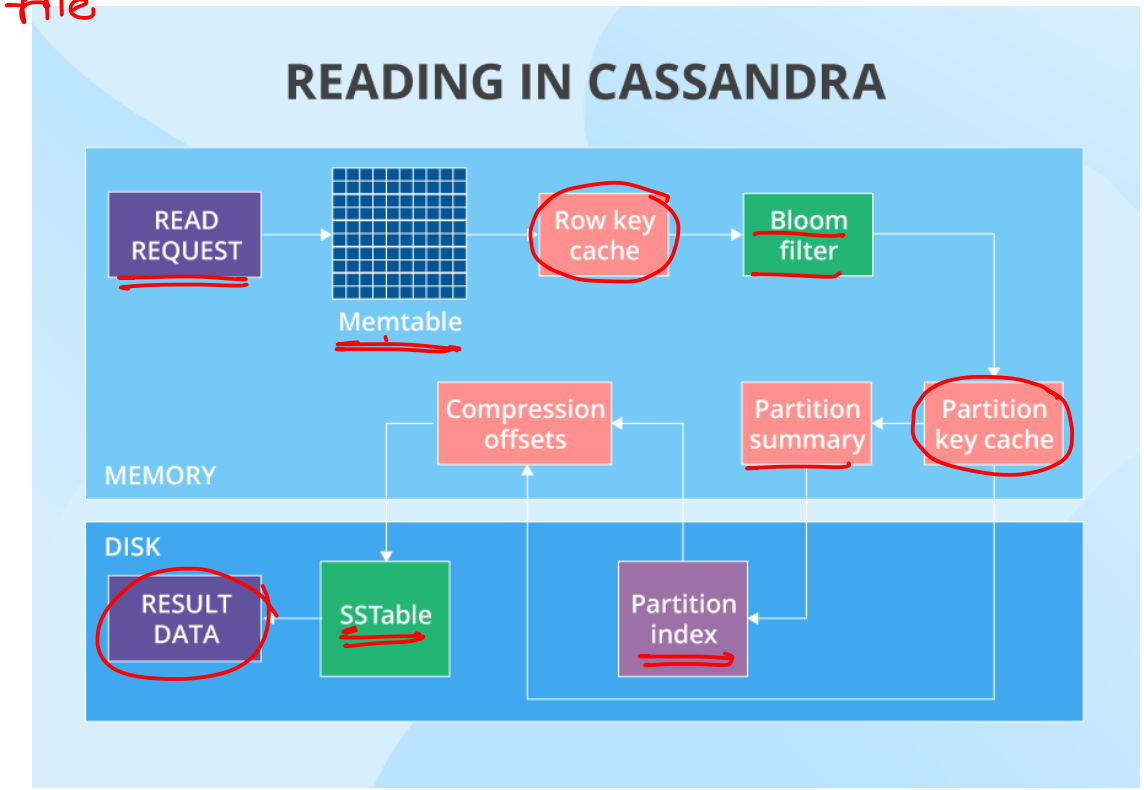
Architecture

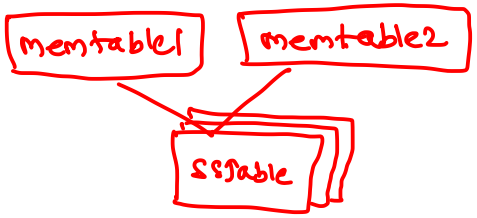
- **Commit Log**
  - Append only log of all mutations local to a node.
  - Client data commit log > memtable.
  - Durability in the case of unexpected shutdown
  - On startup, any changes in log will be applied to tables
- **Memtable**
  - In-memory structures to write Cassandra buffers.
  - One active memtable per table.
- **Sorted String Table**
  - Immutable data files for persisting data on disk.
  - Multiple memtables merged into single SSTable.
- **LSM Tree**
  - Disk based data structure to provide low-cost
  - indexing for a file, in which records are to be inserted at very high rate



Architecture

- **Bloom filter** → helps cassandra finding right sstable file
  - In the read path, Cassandra merges data on disk (in STables) with data in RAM (in memtables).
  - To avoid checking every SSTable data file for the partition being requested, Cassandra employs a data structure known as a bloom filter.
  - Bloom filters are a probabilistic data structure that allows Cassandra to determine one of two possible states
    - The data definitely does not exist in the given file
    - The data probably exists in the given file





Components : cluster

1. **Node** → machine having cassandra installed (peer)
   - A Cassandra node is a place where data is stored
2. **Data center** → cluster
   - Data center is a collection of related nodes
3. **Cluster** → # data centers
   - A cluster is a component which contains one or more data centers
4. **Commit log**
   - In Cassandra, the commit log is a crash-recovery mechanism
   - Every write operation is written to the commit log [create / update / delete]
5. **Mem-table**
   - A mem-table is a memory-resident data structure
   - After commit log, the data will be written to the mem-table
   - Sometimes, for a single-column family, there will be multiple mem-tables

Components

- **SSTable**
  - It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value
- **Bloom filter**
  - These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set
  - It is a special kind of cache
  - Bloom filters are accessed after every query

Data Model

- Cassandra provides the Cassandra Query Language (CQL), an SQL-like language, to create and update database schema and access data
- CQL allows users to organize data within a cluster of Cassandra nodes using:
  - **Keyspace** → Similar to database
    - Defines how a dataset is replicated, per datacenter
    - Replication is the number of copies saved per cluster. Keyspaces contain tables
  - **Table**
    - Defines the typed schema for a collection of partitions. Tables contain partitions, which contain rows, which contain columns. Cassandra tables can flexibly add new columns to tables with zero downtime.
  - **Partition**
    - Defines the mandatory part of the primary key all rows in Cassandra must have to identify the node in a cluster where the row is stored
    - All performant queries supply the partition key in the query.
  - **Row**
    - Contains a collection of columns identified by a unique primary key made up of the partition key and optionally additional clustering keys
  - **Column**
    - A single datum with a type which belongs to a row

CQL

- Users can access Cassandra through its nodes using Cassandra Query Language (CQL)
- CQL treats the database **(Keyspace)** as a container of tables
- Programmers use **cqlsh:** a prompt to work with CQL or separate application language drivers

Data Types

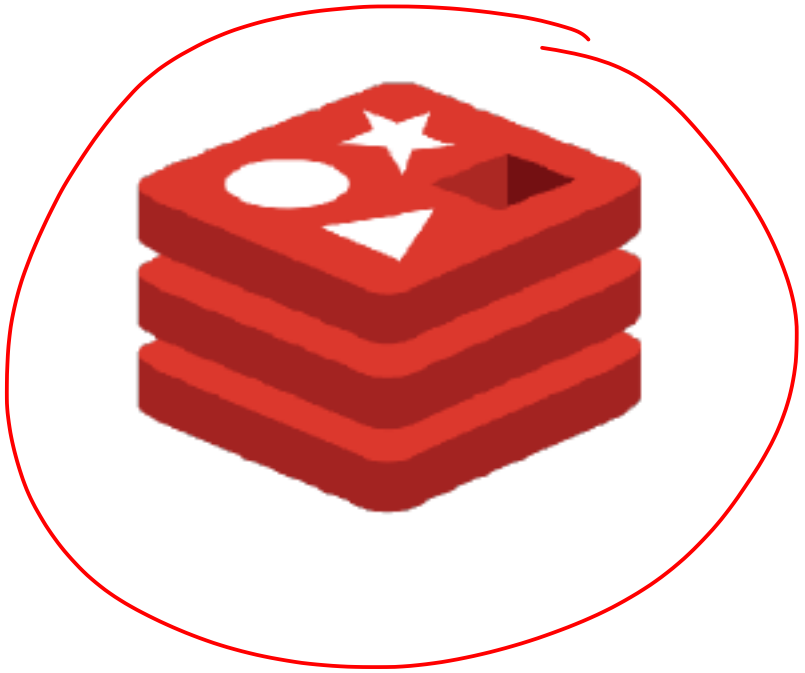
- **ascii**: US-ascii character string
- **bigint**: 64-bit signed long ints
- **blob**: Arbitrary bytes in hexadecimal
- **boolean**: True or False
- **counter**: Distributed counter values 64 bit
- **decimal**: Variable precision decimal
- **double**: 64-bit floating point
- **float**: 32-bit floating point
- **frozen**: Tuples, collections, UDT containing CQL types
- **inet** - IP address in ipv4 or ipv6 string format
- **int**: 32 bit signed integer
- **list**: Collection 01 elements
- **map**: JSON style collection of elements
- **set**: Sorted collection of elements
- **text**: UTF-8 encoded strings
- **timestamp**: ID generated with
- **date+time**: as int/string
- **timeuuid**: Type 1 uuid
- **tuple**: A group of 2,3 fields
- **uuid**: Standard uuid (128-bit)
- **varchar**: UTF-8 encoded string
- **varint**: Arbitrary precision integer

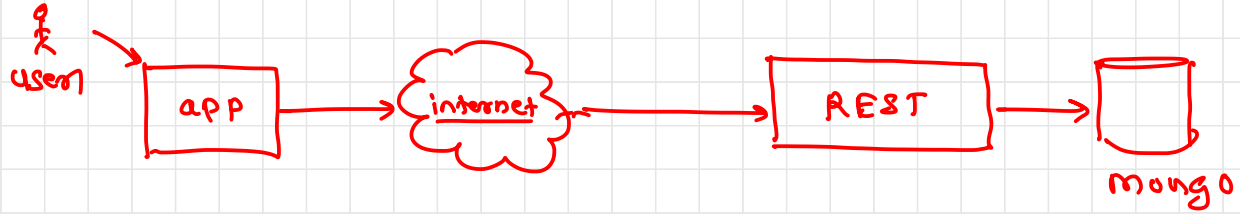
Universally Unique Identifier

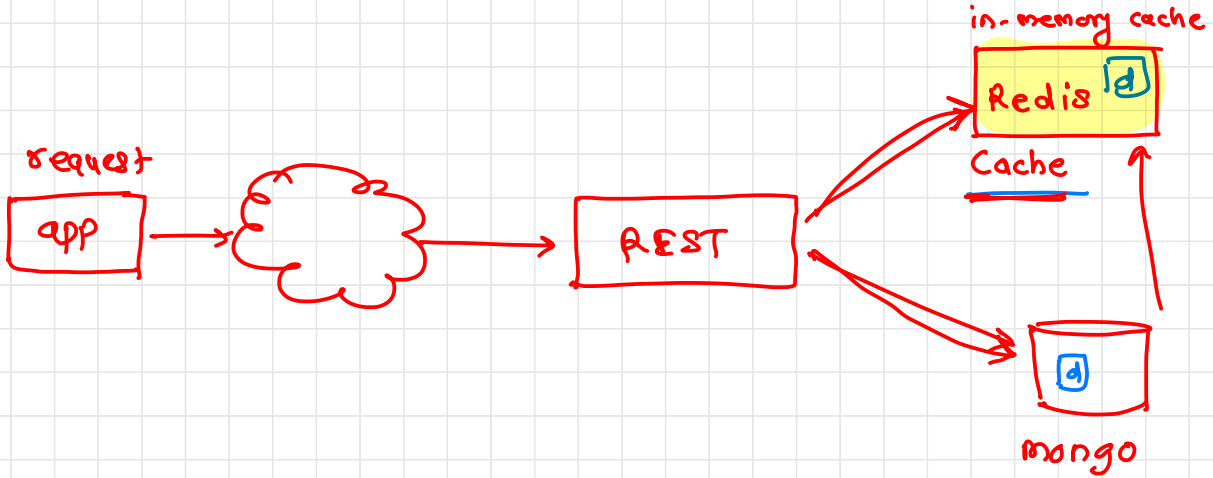
RDBMS vs Cassandra

RDBMS	Cassandra
RDBMS deals with structured data.	Cassandra deals with unstructured data.
It has a fixed schema.	Cassandra has a flexible schema.
In RDBMS, a table is an array of arrays. (ROW x COLUMN)	In Cassandra, a table is a list of “nested key-value pairs”. (ROW x COLUMN key x COLUMN value)
Database is the outermost container that contains data corresponding to an application.	Keyspace is the outermost container that contains data corresponding to an application.
Tables are the entities of a database.	Tables or column families are the entity of a keyspace.
Row is an individual record in RDBMS.	Row is a unit of replication in Cassandra.
Column represents the attributes of a relation.	Column is a unit of storage in Cassandra.
RDBMS supports the concepts of foreign keys, joins.	Relationships are represented using collections.

Redis



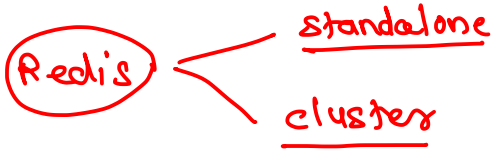

user
app
internet
REST
mongo


in-memory cache
Redis
Cache
request
app
REST
mongo

Introduction

Data gets stored in terms of key-value pairs

- Redis is an open source, advanced key-value store and an apt solution for building high performance, scalable web applications developed in 2009
- REmote DIctionary Server → Dictionary → collection of key-value pairs
- Redis is maintained and developed by Salvatore Sanfilippo
- Based on data structures: strings, hashes, sets, lists, sorted sets, geospatial, indexes, hyperloglogs.
- Redis has three main peculiarities that sets it apart
  - Redis holds its database entirely in the memory, using the disk only for persistence ✓
  - Redis has a relatively rich set of data types when compared to many key-value data stores
  - Redis can replicate data to any number of slaves



Features

- Speed: 110,000 SET/s and 81000 GET/s on entry-level Linux system
- Pipeline: Multiple commands execution for faster execution
- Persistence: Whole data accessed from memory, asynchronously persisted on disk with flexible policies
- Data Structure: Based on data structures like Strings, Hashes, Sets
- Atomic operations: Data is manipulated atomically by multiple clients
- Supported Languages: Drivers available for C/C++, Java, Python, R, PHP
- Master/Slave replication: Easy config and fast execution
- Sharding: Distributing across cluster. Based on client driver capability
- Portable: Developed in C. Work on all UNIX variants. Not supported on Windows

Highlights

- Key-value DB, where values can store complex data types with atomic ops
- Value types are basic data structures made available to programmers without layers of abstraction
- It is in-memory but persistent store i.e. whole database is maintained in server RAM, only changes are updated on disk for backup
- The data storage in disk is in append-only data files
- Maximum data size is limited to the RAM size
- On modern systems if Redis is going out of memory, it will start swapping and slow down the system
- Max memory limit can be configured to raise error on write or evict keys

Redis Versus Other Key-value Stores

- Redis is a different evolution path in the key-value DBs, where values can contain more complex data types, with atomic operations defined on those data types
- Redis is an in-memory database but persistent on disk database, hence it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than the memory
- Another advantage of in-memory databases is that the memory representation of complex data structures is much simpler to manipulate compared to the same data structure on disk. Thus, Redis can do a lot with little internal complexity

Installation

- To install Redis on Ubuntu, go to the terminal and type the following commands –
  - > sudo apt-get update
  - > sudo apt-get install redis-server
- Start Redis server
  - > redis-server
- Run client
  - > redis-cli

Data Types

Strings → collection of characters / sequence

- Redis string is a sequence of bytes
- Strings in Redis are binary safe, meaning they have a known length not determined by any special terminating characters
- Thus, you can store anything up to 512 megabytes in one string
- E.g.
  - redis-cli> set name “amit”
  - redis-cli> get name

here
key: name
value: amit

Hashes

- A Redis hash is a collection of key value pairs
- Redis Hashes are maps between string fields and string values.
- Hence, they are used to represent objects
- Every hash can store up to $2^{32} - 1$ field-value pairs (more than 4 billion)
- E.g.
  - redis-cli> hset user:amit username amit password sunbeam address pune
  - redis-cli> hegetall user:amit
  - redis-cli> heget user:amit username
  - redis-cli> heget user:amit address

if a key is missing hget will return the value as nil

nil → similar to null
→ value is not available

Lists

& collection

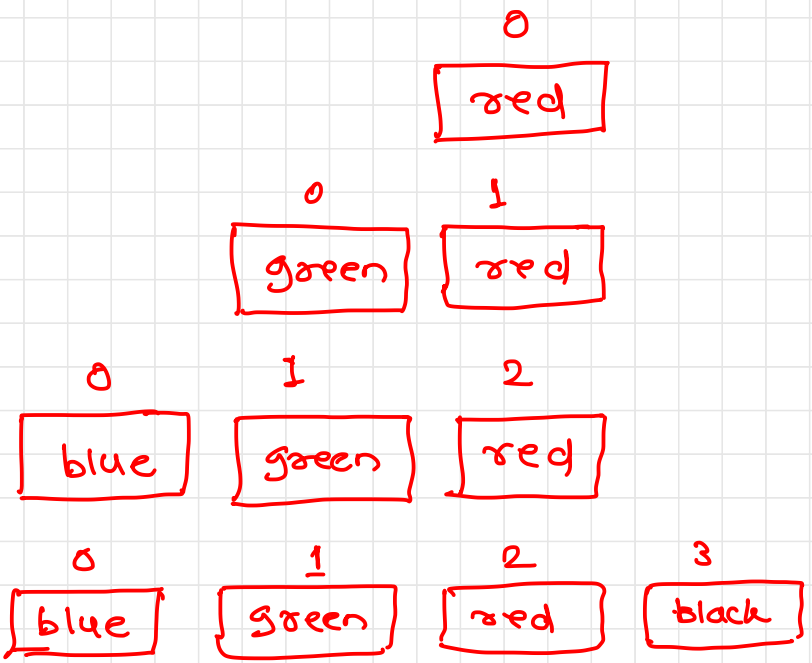
- Redis Lists are simply lists of strings, sorted by insertion order
- You can add elements to a Redis List on the head or on the tail
- The max length of a list is 2^{32} - 1 elements (4294967295, more than 4 billion of elements per list).
- E.g.
  - redis-cli> lpush colors red
  - redis-cli> lpush colors green
  - redis-cli> lpush colors blue
  - redis-cli> lrange colors 0 2

> lpush colors red

> lpush colors green

> lpush colors blue

> rpush colors black



Sets : collection of unique values

- Redis Sets are an unordered collection of strings
- In Redis, you can add, remove, and test for the existence of members in O(1) time complexity
- The max number of members in a set is 2^{32} - 1 (4294967295, more than 4 billion of members per set)
- E.g.
  - redis-cli> sadd colors red
  - redis-cli> sadd colors green
  - redis-cli> sadd colors blue
  - redis-cli> smembers colors 0 2

Ordered Set

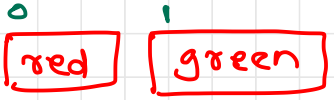
- Redis Sorted Sets are similar to Redis Sets, non-repeating collections of Strings
- The difference is, every member of a Sorted Set is associated with a score, that is used in order to take the sorted set ordered, from the smallest to the greatest score
- While members are unique, the scores may be repeated
- E.g.
  - redis-cli> zadd colors 0 red
  - redis-cli> zadd colors 1 green
  - redis-cli> zadd colors 2 blue
  - redis-cli> zrangebyscore colors 0 2

score

Sorted set

> zadd colors 0 red

> zadd colors 1 green



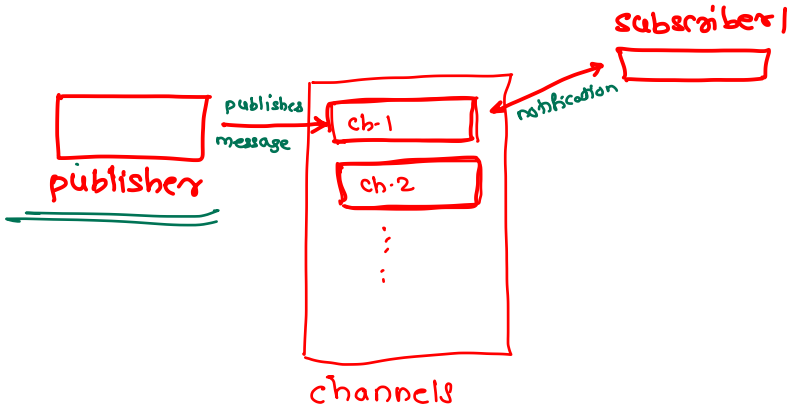
> zadd colors 0 blue

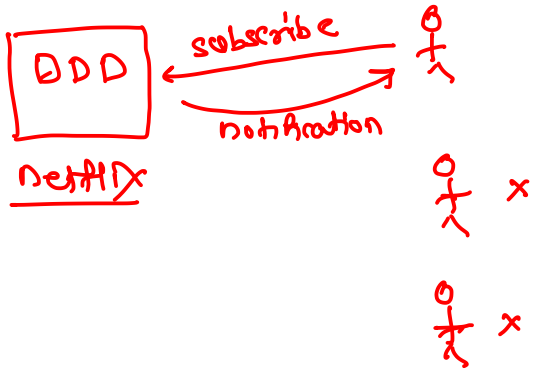
> zadd colors 1 black

Pub-Sub

Publish - Subscribe

- Redis Pub/Sub implements the messaging system where the senders (in redis terminology called publishers) sends the messages while the receivers (subscribers) receive them
- The link by which the messages are transferred is called **channel**
- In Redis, a client can subscribe any number of channels





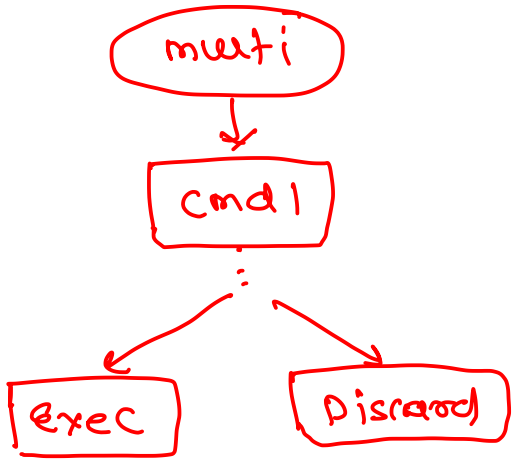
Publish / Subscribe

- SUBSCRIBE channel-pattern
  - receive notifications from given channels. e.g. b?g, b*g, b[ailg
- PUBLISH channel "message"
  - send message to channel
- UNSUBSCRIBE channel-pattern
  - stop receiving notifications from given channels.
- UNSUBSCRIBE channel
  - stop receiving notifications from given channel.
- PUBSUB command
  - monitor pub-sub subsystem
  - e.g. PUBSUB channels

Advanced Redis

Transactions

- Redis transactions allow the execution of a group of commands in a single step
- All commands in a transaction are sequentially executed as a single isolated operation
- It is not possible that a request issued by another client is served in the middle of the execution of a Redis transaction
- Redis transaction is also atomic. Atomic means either all of the commands or none are processed
- E.g.
  - redis-cli> multi
  - redis-cli> set username amit
  - redis-cli> set address pune
  - redis-cli> exec



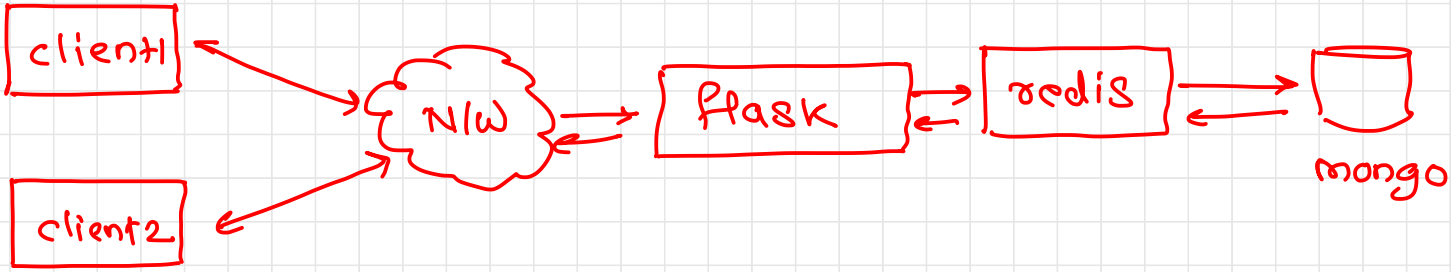
Pipeline

cat "." | grep -e " "

- Redis is a TCP server and supports request/response protocol
- In Redis, a request is accomplished with the following steps
  - The client sends a query to the server, and reads from the socket, usually in a blocking way, for the server response
  - The server processes the command and sends the response back to the client
- The basic meaning of pipelining is, the client can send multiple requests to the server without waiting for the replies at all, and finally reads the replies in a single step
- The benefit of this technique is a drastically improved protocol performance
- The speedup gained by pipelining ranges from a factor of five for connections to localhost up to a factor of at least one hundred over slower internet connections

Backup

- Redis **SAVE** command is used to create a backup of the current Redis database
- Backup will be taken in the redis directory set in the config object
- e.g.
  - redis-cli> save
- Alternatively bgsave command can also be used to take a backup, which runs in background


client1
client2
N/W
flask
redis
mongo

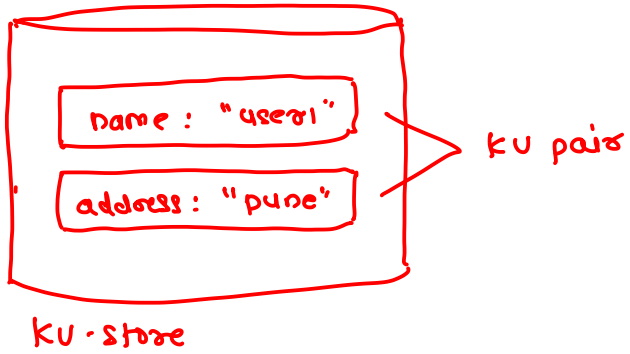
KV store

Similar to redis

Introduction

- Multi-terabyte distributed key-value pair storage - KV Store
- High performance, scalable, Eventual consistency, Durable
- User defined read/write performance levels
- Terminologies:
  - KV Pair - Key (Major & Minor keys), Value : byte array
  - KV Store - Container of KV pairs
  - Partition - Hashed Set of Records (on major keys)
  - Shard - Set of partitions. Group of machines for replication.
    - Shard is chosen transparently i.e. auto selected by oracle nosql db
  - Replication factor - Number of replicas. Default is 3.
  - Storage node Physical machine for storing data (CPU+RAM+Disk)

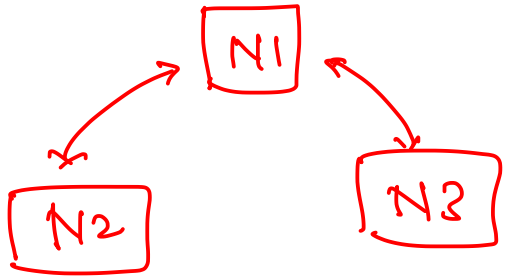
virtual



Introduction

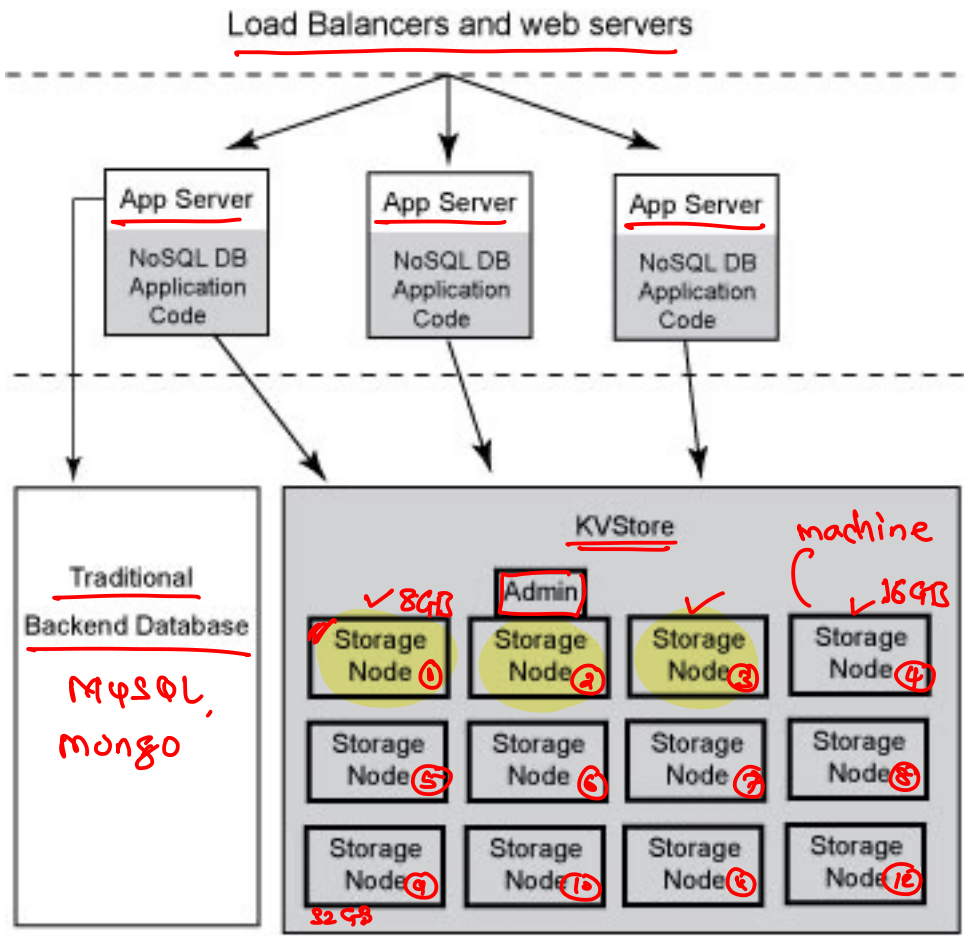
- The KVStore is a collection of Storage Nodes which host a set of Replication Nodes
- Data is spread across the Replication Nodes
- Given a traditional three-tier web architecture, the KVStore either takes the place of your back-end database, or runs alongside it [like a cache]

Similar to redis



Storage Node

- The store contains multiple Storage Nodes
- It is a physical (or virtual) machine with its own local storage
- The machine is intended to be commodity hardware
- It should be, but is not required to be, identical to all other Storage Nodes within the store
- Every Storage Node hosts one or more Replication Nodes as determined by its *capacity*
- The capacity of a Storage Node serves as a rough measure of the hardware resources associated with it
- A store can consist of Storage Nodes of different capacities



Replication Nodes and Shards

- At a very high level, a Replication Node can be thought of as a single database which contains key-value pairs
- Replication Nodes are organized into shards
- A shard contains a single Replication Node which is responsible for performing database writes, and which copies those writes to the other Replication Nodes in the shard
- This is called the master node
- All other Replication Nodes in the shard are used to service read-only operations. These are called the replicas.
- Although there can be only one master node at any given time, any of the members of the shard are capable of becoming a master node. In other words, each shard uses a single master/multiple replica strategy to improve read throughput and availability.