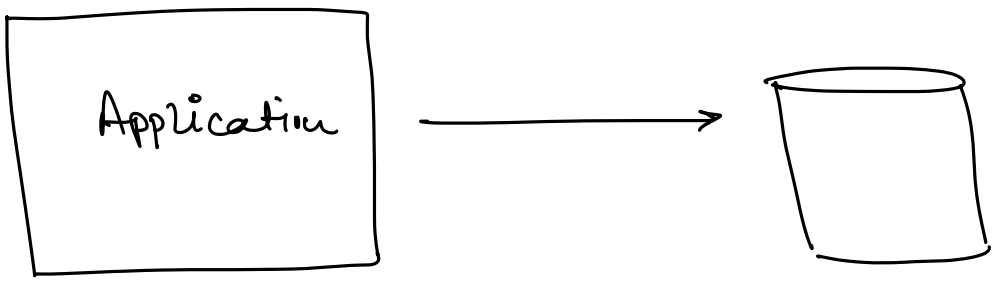
Agenda.

→ Introduction to Spring Data JPA.
  - → ORM
  - → JPA
  - → JDBC

→ Repository Pattern

→ UUID's.



⇒ Work with Databases.
  - → Create tables
  - → CRUD.

```
CreateNewProduct (Product p) {

Connect with DB {  Database db = new MySQL(url, username,
                                           password);

    String q = " insert into product
                 value
                 (p.title, ------ );

    db.execute(q);

}
```

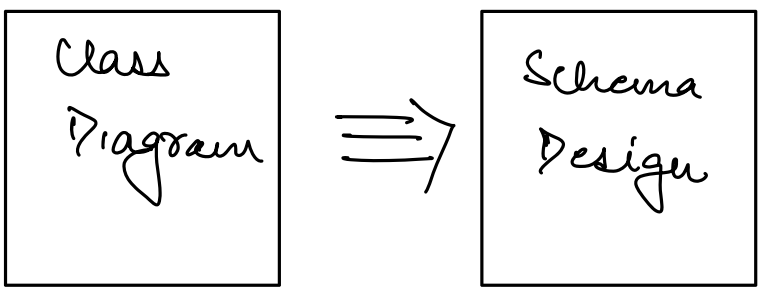
```
Product getProductById(long id){
    Database db = new MySQL(url, username,
                            password);

    String q = "select * from
                Products where id = ";

    Row r = db.execute(q);

    //convert r in product object
}
```

⇒ Create the tables in DB.



@Entity ⇒ JPA

Product
-id -title -desc -qty -image

⇒ table in DB.

Products

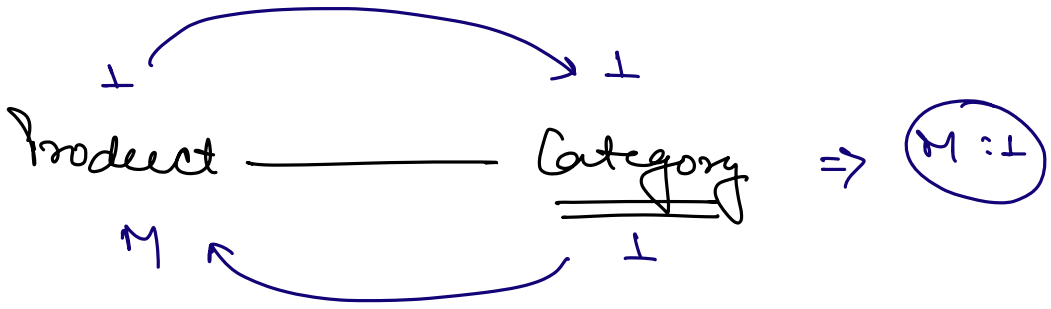
id	title	desc	qty	image.

⇒ ORM Libraries.

↳ Object Relational Mapping.

Provides us an easy way to work with databases based on the models that are there in our codebase.

1. Automatically creates tables corresponding to our model classes.
2. Easy to use methods to perform CRUD operations on DB.



@ManyToOne

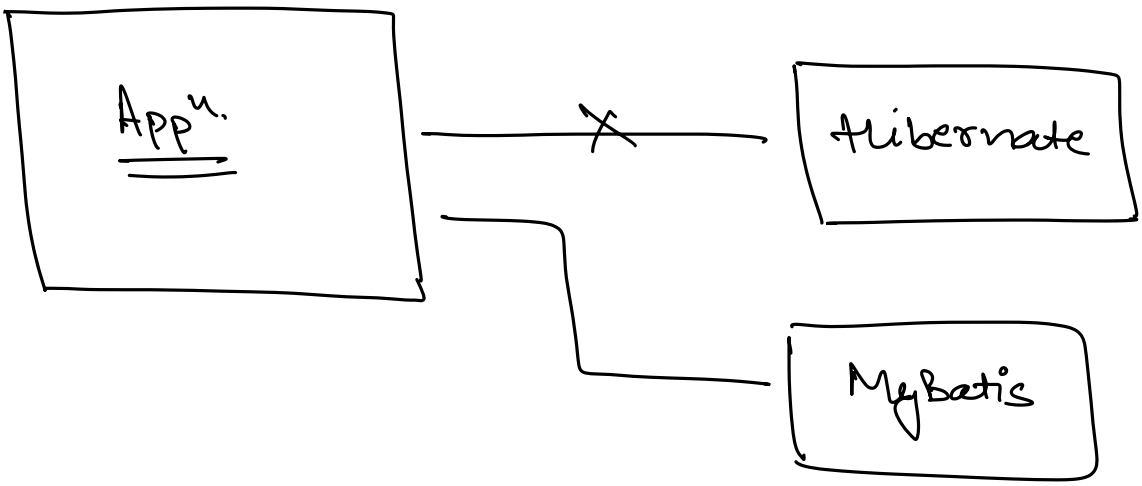
@ManyToMany

⇒ Some famous ORM libraries.

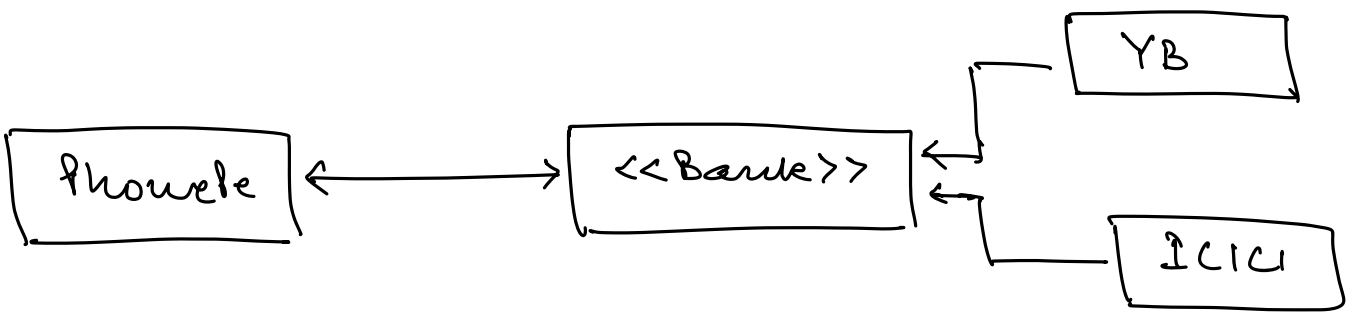
→ Hibernate

→ MyBatis

→ Jooq



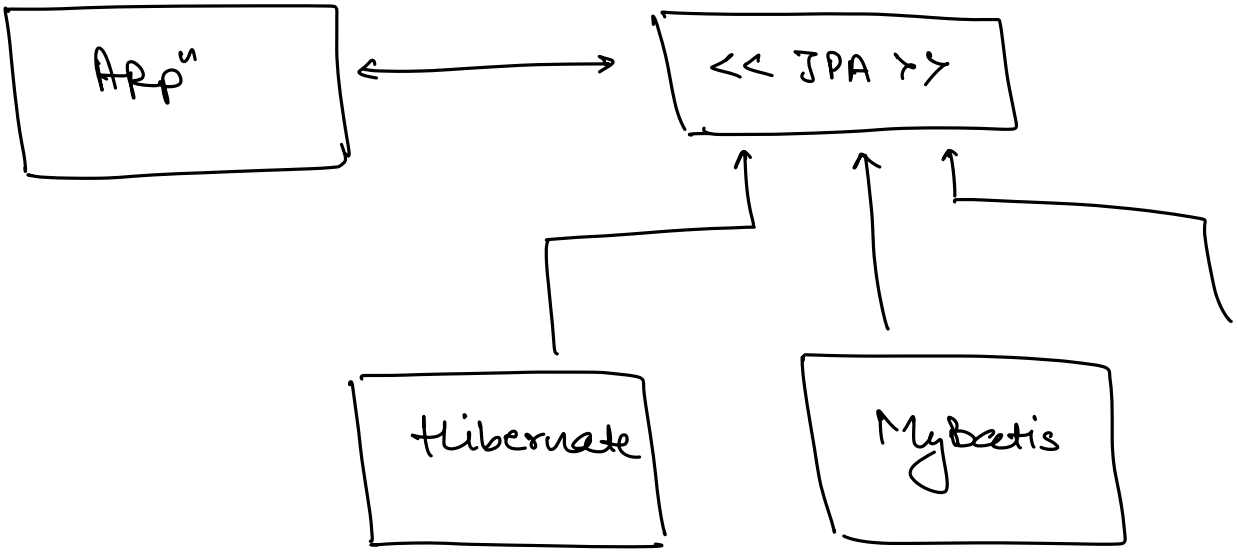
⇒ If our app" is directly depending on an ORM library then it will be very difficult to migrate to some other ORM.



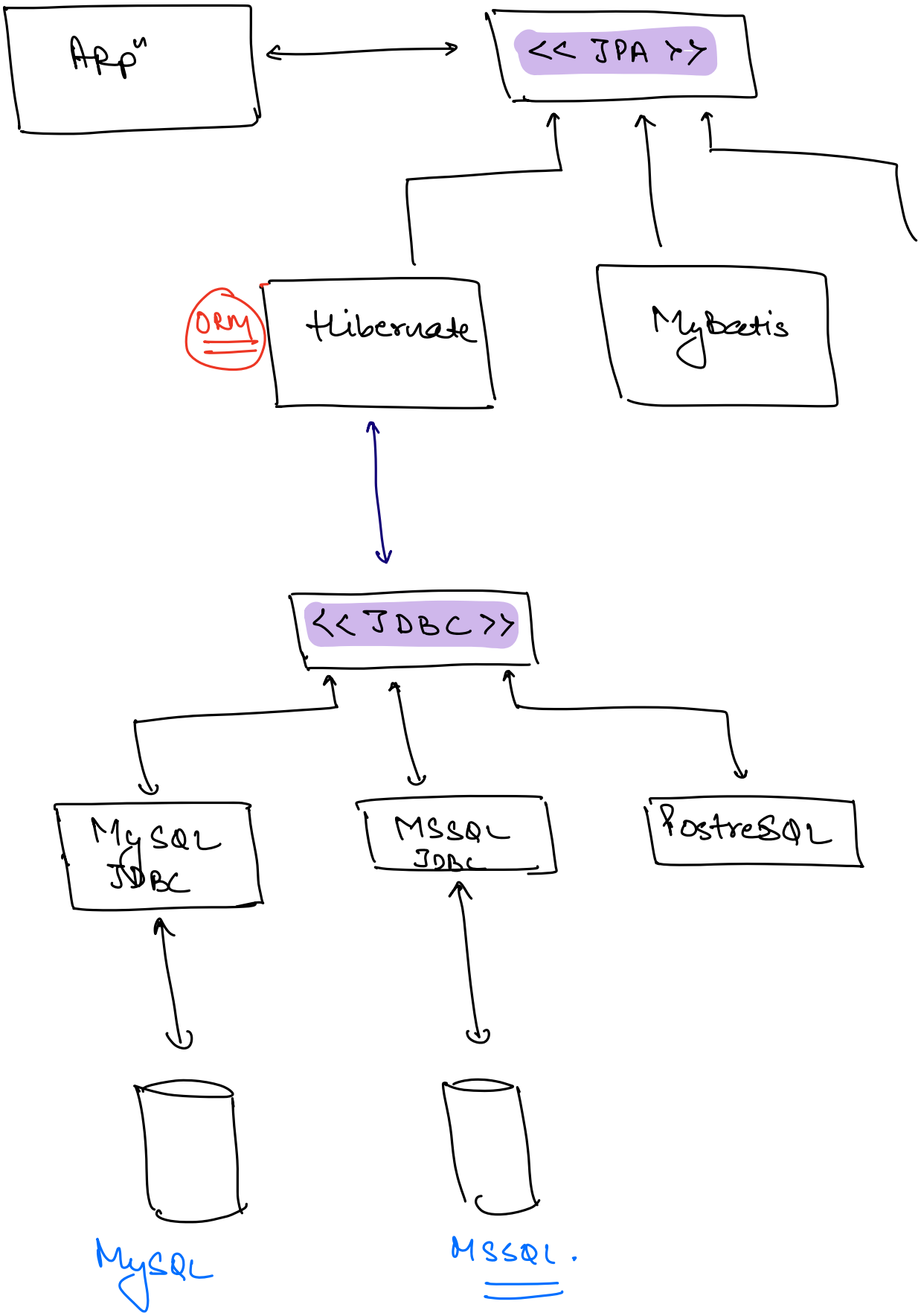
JPA

↳ Java Persistence API.

⇒ Interface that helps us to connect with different ORM libraries.



JPA jpa = new ~~Hibernate()~~;
Mybatis()


App
<< JPA >>
ORM
Hibernate
Mybatis
<< JDBC >>
MySQL JDBC
MSSQL JDBC
PostreSQL
MySQL
MSSQL.

Appⁿ ⟷ JPA ⟷ ORM ⟷ JDBC ⟷ MySQLJDBC.

Repository Pattern.

Code to interact with persistence layer (DB) should be separate from our business logic.

There should be separate layer to interact with DB.

↓

Repository

ProductRepository {

// Code to interact with Products table.

save(Product p) {

db.execute(query);

}

findProductById(id) {

db.execute(_____);

}

_____

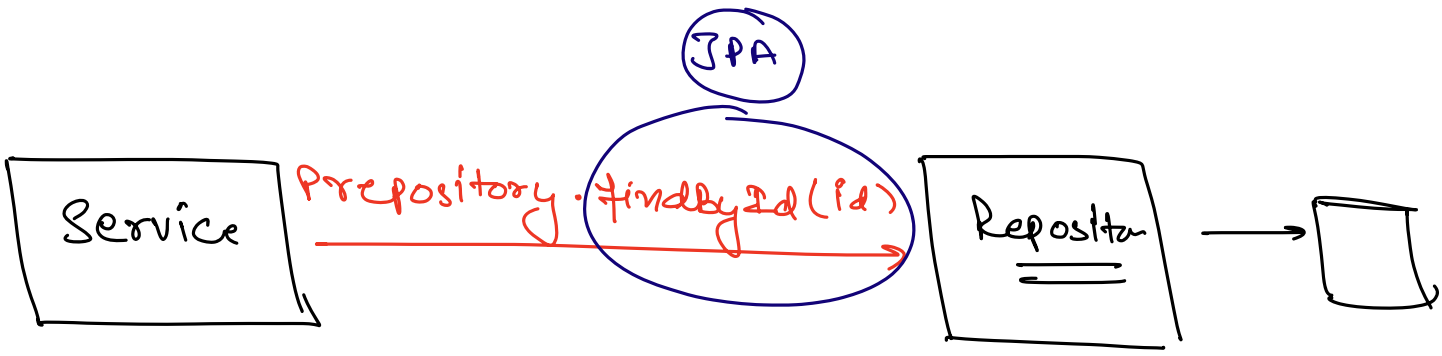
_____

_____

}

UserRepository {

}



interface ProductRepository extends JPARepository<Product, Long> {

Product findById();

}

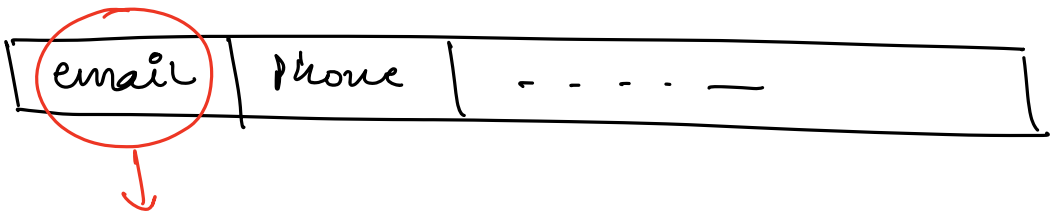
⇒ UUID

↳ Universal Unique Identifier.

⇒ Every table in the Database should have a PK.

Unique.

Users.

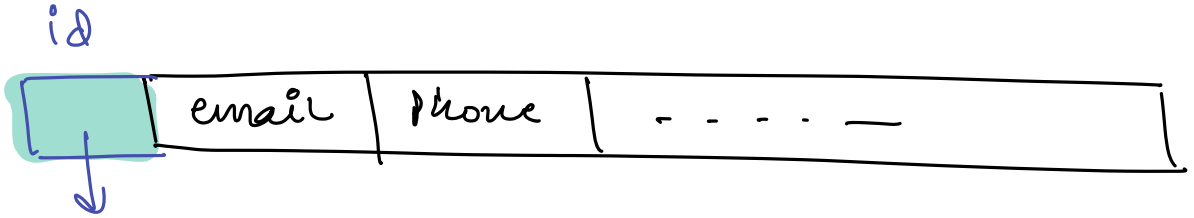


PK ?

Yes.

Cons of having email as PK.

1. String Comparison is costly.
2. Size of index table will be more.



PK.

⇒ Having a separate **unique id** column is a better idea.

Data type of PK column

1) Int
   - 4B
   - $2^{32} \approx 2B$
   - $\approx 2 \times 10^9$

2) BigInt | long
   - 8B
   - 2^{64}
   - 10^{18}

Most Common.

⇒ **Auto increment.**

id
1
2
3
4
⋮

Twitter API.

↳ fetch stream of tweets

↳ Lastly

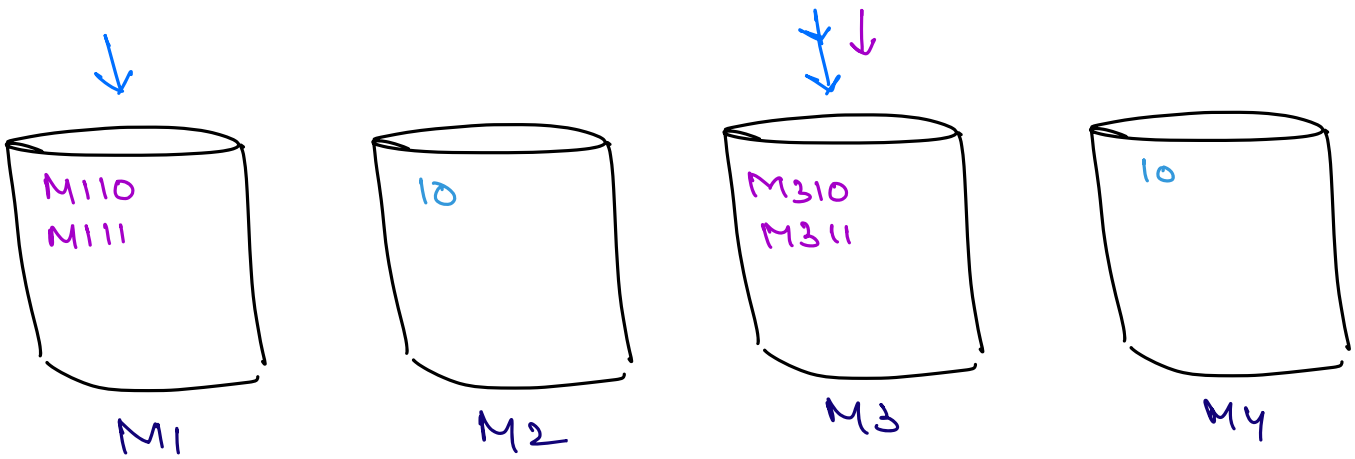
twitter.com/tweets/ ——

youtube.com/videos/id

scaler.com/users/ —— ✗

⇒ If the id's are visible publicly then auto increment is not a good option.

⇒ SHARDING.



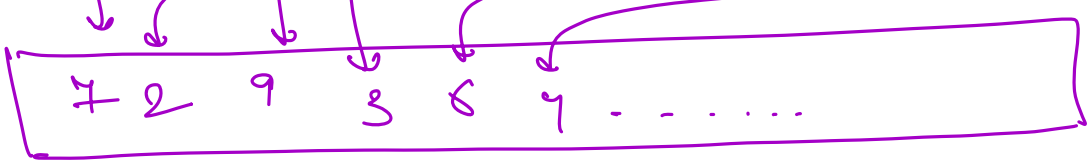
⇒ If data is distributed across multiple DB m/c then auto increment won't work.

Id's ⇒ fⁿ (Machine id + Timestamp + userId + IP + ----)

UUID.

Completely Random.

fn (12798523 + 987654321 + 1234 + 192687384 + ...)



UUID : 128 Bit Number.

↳ Not String

UUID : 10010101001111010101 - - - - -

128 Bits.

↓

HEXADECIMAL. (16 Base)

0000 ⇒ 0
0001 ⇒ 1
0010 ⇒ 2
0011 ⇒ 3
0100 4
0101 5
0110 6
0111 7
1000 8
1001 9
1010 10 → a
1011 11 → b
1100 12 → c
1101 13 → d
1110 14 → e
1111 15 → f

(d) ⇒ (13)

3c09de26-0209-4b4b-834f-057abc43c532 ⇒ HEXA DECIMAL

128 Bit ⇒ $\frac{128}{4} = 32$

———— * ————

Next Class

⇒ Work on Databases.

⇒ —
—
—