Fullstack Development

Database Design

Content

- Database ranking

- SQL database

- NoSQL database

- Schema patterns
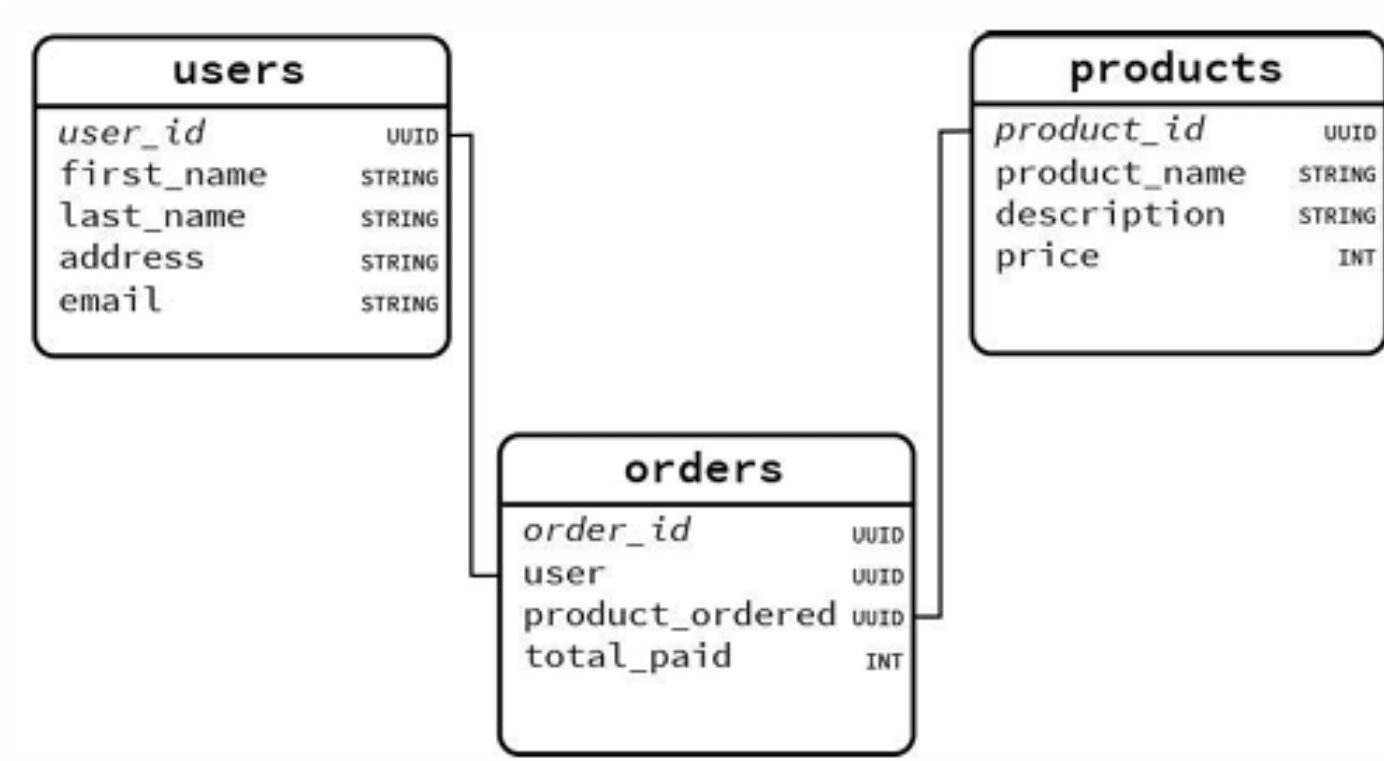
- Some useful information

Database Engine Ranking

- Database engine
  - DBMS (Database Management System)

- A brief history of databases

- DB-Engines Ranking

SQL Database

- Relational database
- Organize data into `tables` of related information
- Utilize `Structured Query Language (SQL)` for managing/manipulating data

SQL Database

Popular RDBMS

- Open source: MySQL, PostgreSQL

- Commercial: Oracle Database, Microsoft SQL Server, IBM DB2

- RDBMS Ranking

SQL

The standard language used to interact with SQL databases

- Data Definition Language (DDL)
  - e.g., `CREATE TABLE` , `ALTER TABLE` , `DROP TABLE`
- Data Manipulation Language (DML)
  - e.g., `INSERT` , `UPDATE` , `DELETE` , `SELECT`
- Data Control Language (DCL)
  - e.g., `GRANT` , `REVOKE`

ACID Properties

- An acronym that stands for ...
  - `Atomicity`
  - `Consistency`
  - `Isolation`
  - `Durability`
- Ensure reliable transaction processing and data integrity
- What does ACID Means?

NoSQL

- `non SQL` or `not only SQL`

- Store data in a format other than relational tables

- Mostly designed for high `scalability` and `availability`

Types of NoSQL Database

- Document-oriented

- Column-oriented

- Graph-based

- Key-Value pair

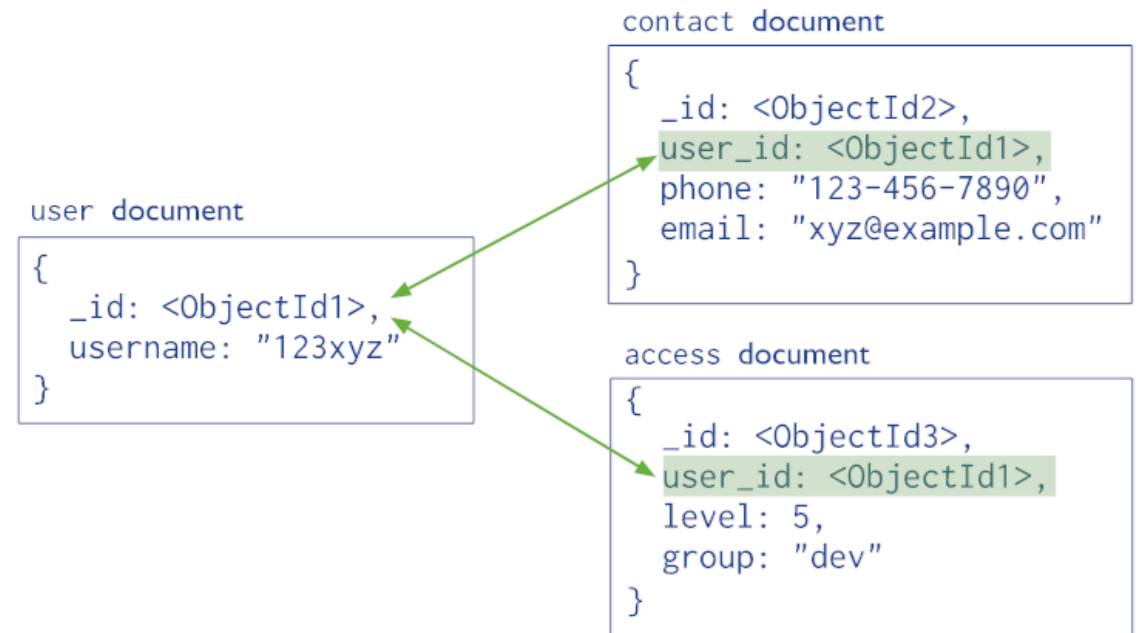- Time series

Document Database

- The data is stored in `document`
- Each `document` is typically a `nested structure` of `keys` and `values`
- **Possible to retrieve only parts of a document**
- The most commonly used data format are `JSON`, `BSON`, and `XML`
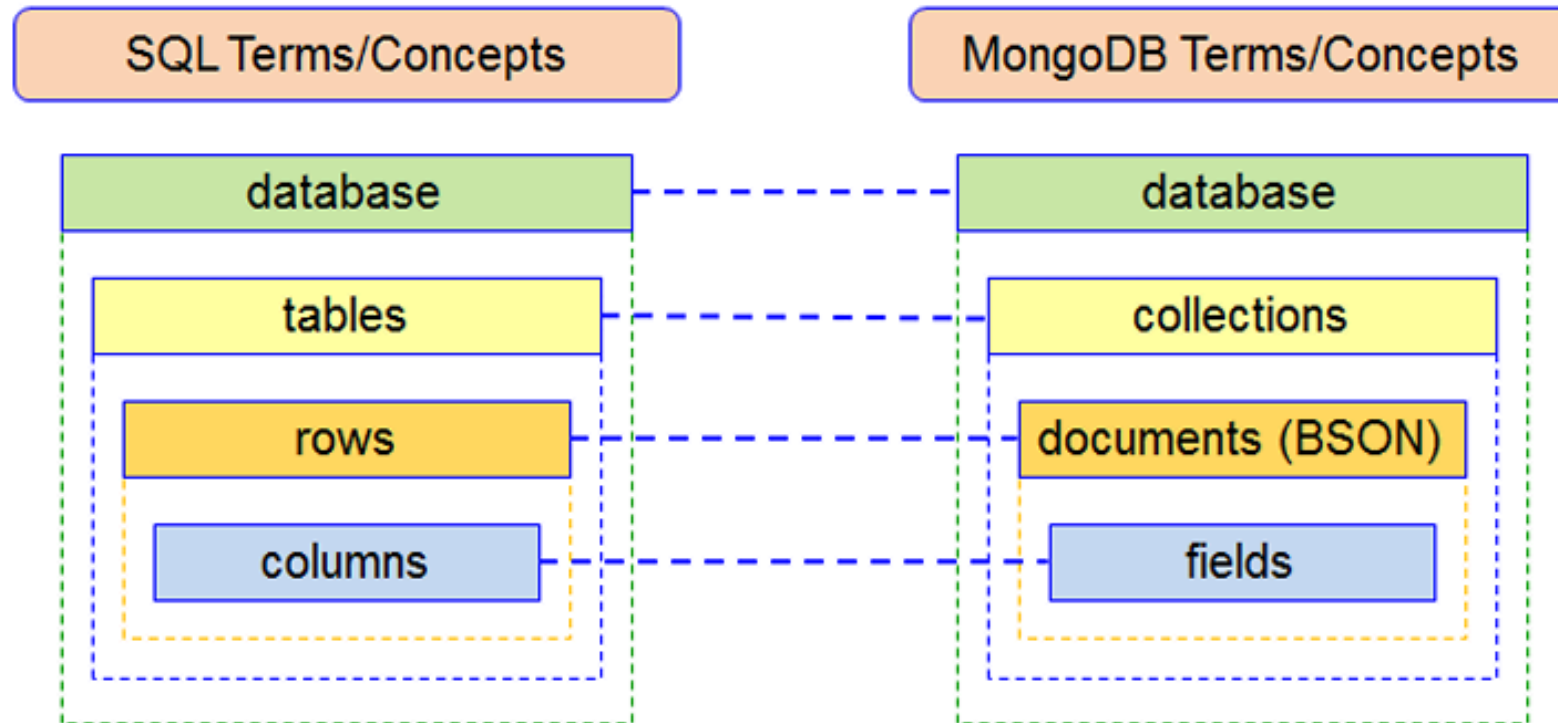- e.g., MongoDB, Apache CouchDB

Document Database

(a) Embedded Data Model

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
           phone: "123-456-7890",
           email: "xyz@example.com"
         },                              >  Embedded sub-
                                            document
  access: {
           level: 5,
           group: "dev"
         }                               >  Embedded sub-
                                            document
}
```

(b) Normalized Data Model

contact document
```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

user document
```
{
   _id: <ObjectId1>,
   username: "123xyz"
}
```

access document
```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```
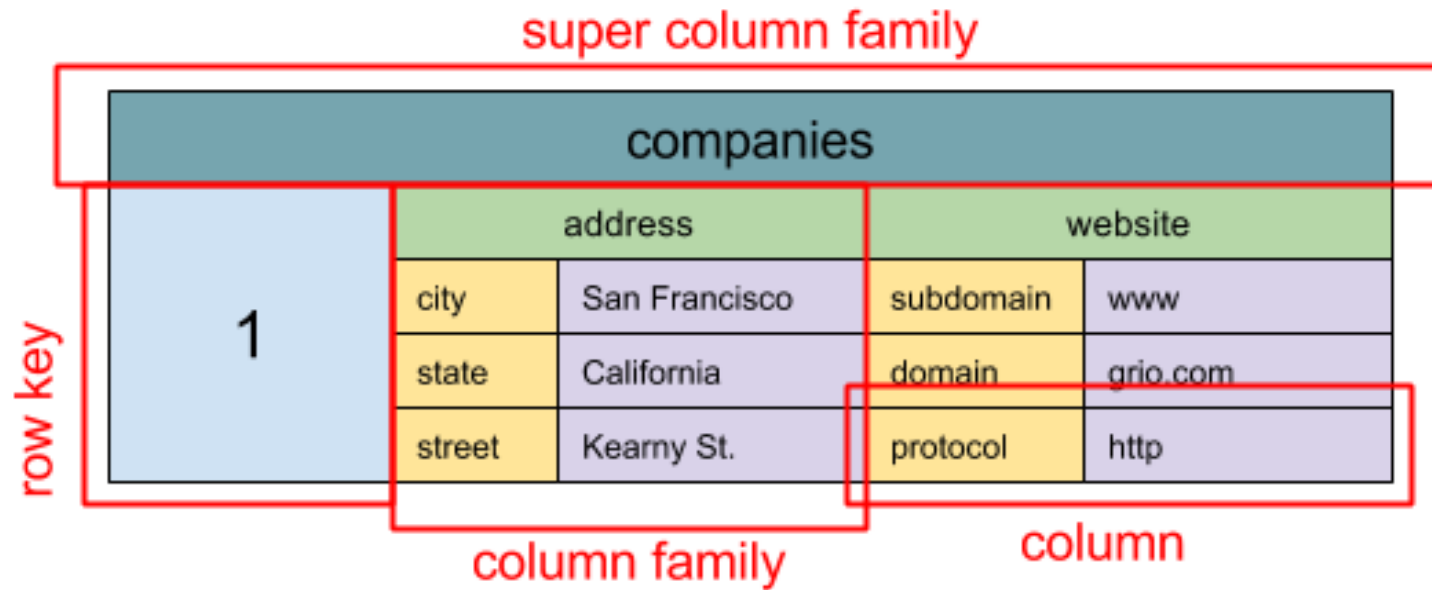
Document Database: Terminology

Wide Column Data Store

- Store data in columns rather than rows

- Able to store large amounts of data in a single column

- Allows to reduce disk resources and the time to retrieve information

- Highly scalable and flexible

- e.g., Apache Cassandra

Wide Column Data Store

Graph Database

- Store and query highly connected data
- Data are modeled in the form of **entities** (`nodes`) and **relationships** (`edges`) between them
- Able to traverse from `nodes` or `edges` along defined relationship types until reaching some `defined condition`
  - Results : `lists` , `maps` , or graph `traversal path`
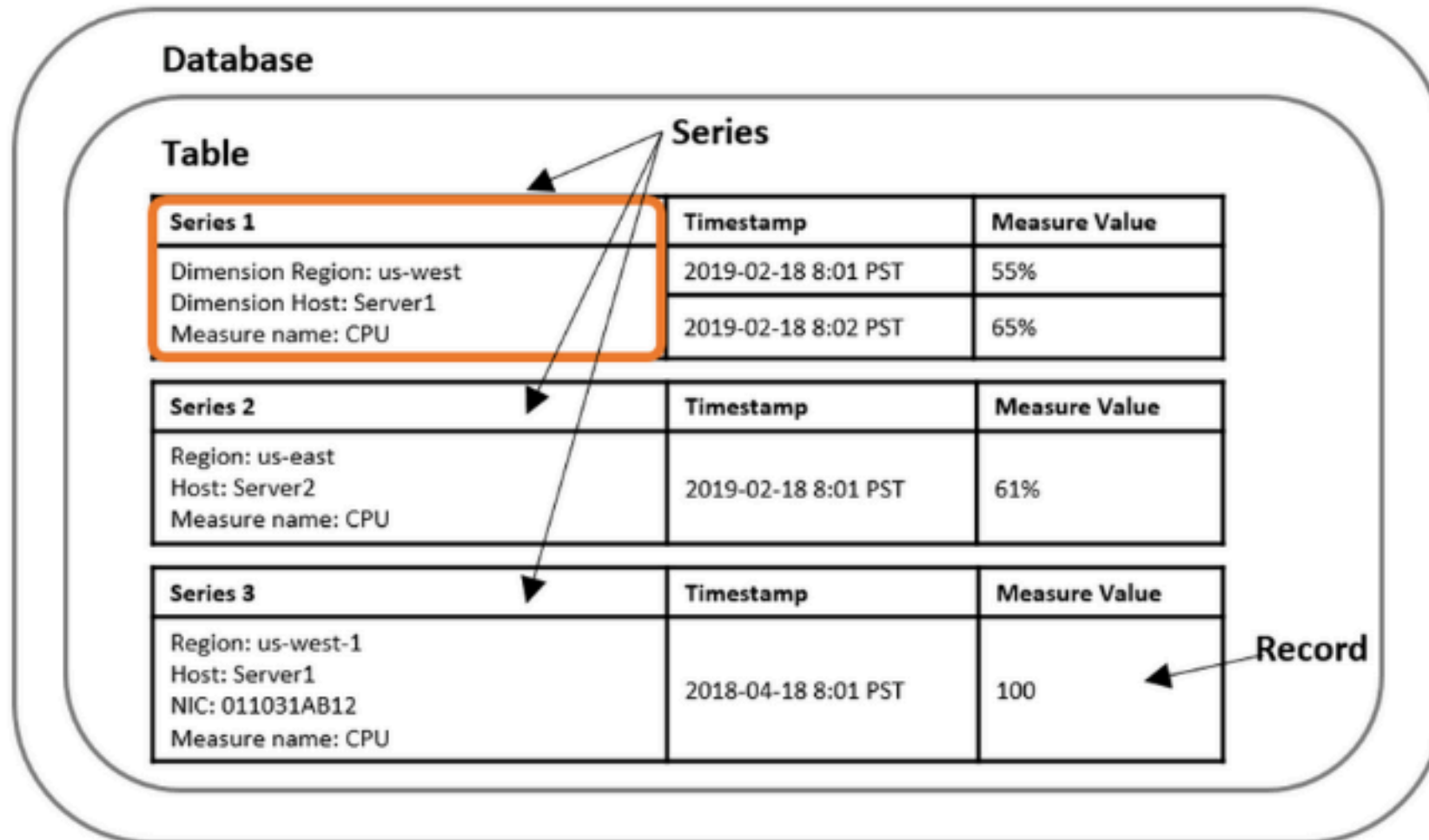- e.g., Neo4j

Graph Database
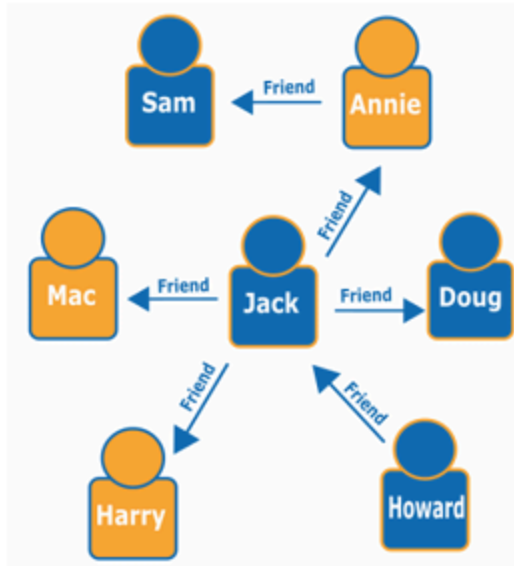
Time Series Database

- Store and retrieve data records that are **sequenced by** `time`
  - Sets of data points associated with `timestamps` and stored in time sequence order
- Easy to measure how data change over time (e.g., IoT application)
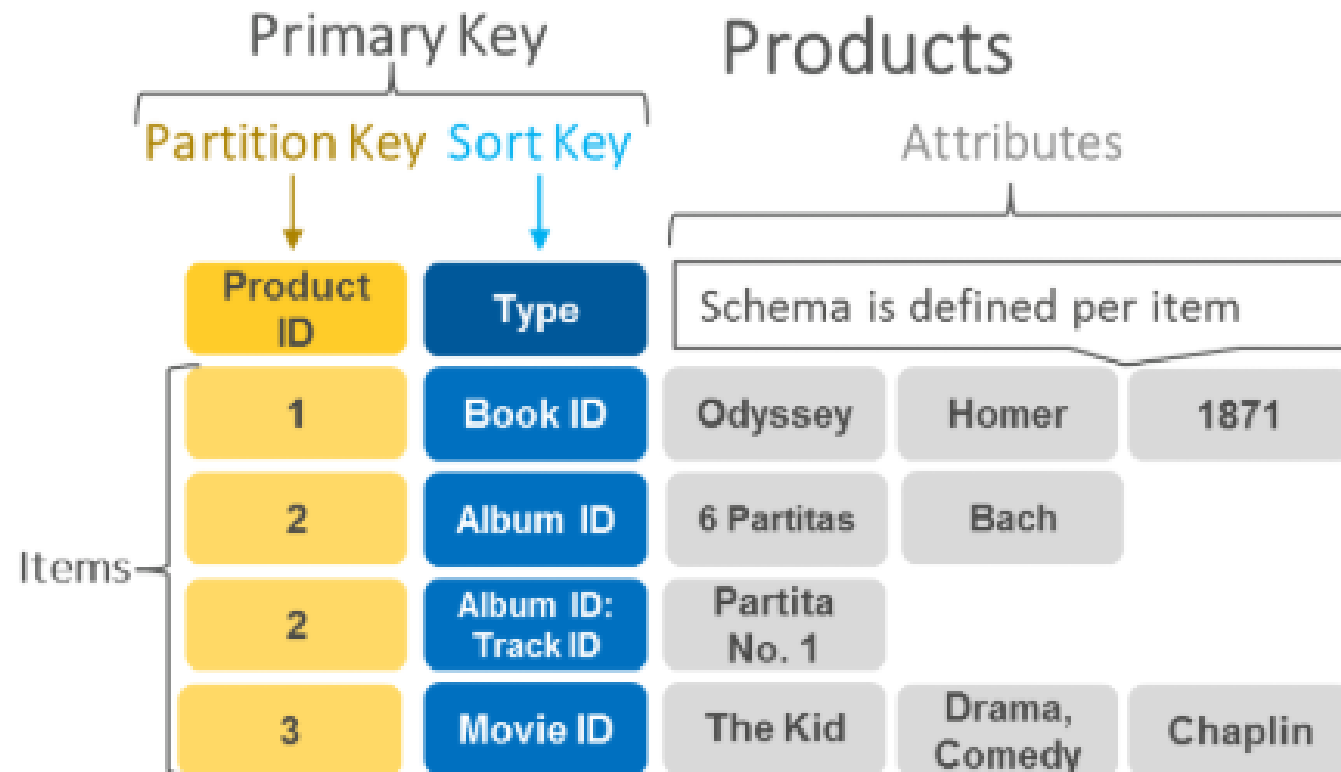- e.g., InfluxDB, Prometheus

Time Series Database

Key-value Data Store

- Stores data as a collection of `key-value pairs`

- Each data item is identified by a `unique key`

- The `value` can be anything (string, number, object, …)

- e.g., Redis, Memcached
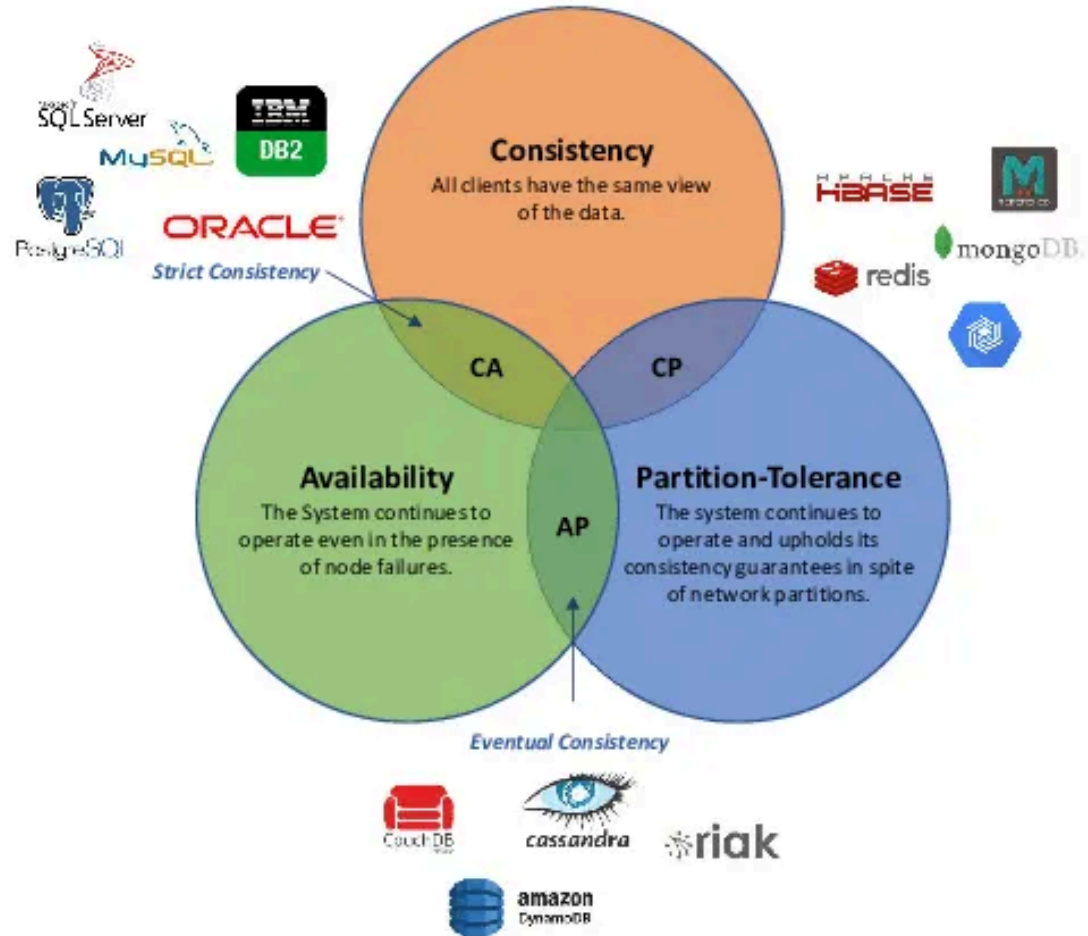
Key-value Data Store

CAP Theorem

In a distributed data system, it is impossible to simultaneously guarantee all of these properties:

- **C** : `Consistency`
- **A** : `Availability`
- **P** : `Partition Tolerance`

Many NoSQL databases are **AP** systems

CAP Theorem

Database Schema

What is Database Schema?

- DB Schema defines how data is organized within the databasae

- Outlining how data is logically stored

- **Key components**:
  - Tables, Columns, Data types, Constraints

  - Primary / Foreign keys

  - Relationships (`one-to-one` , `one-to-many` , `many-to-many`)

Relationship : One-to-Many

e.g., **"Social media status post"**
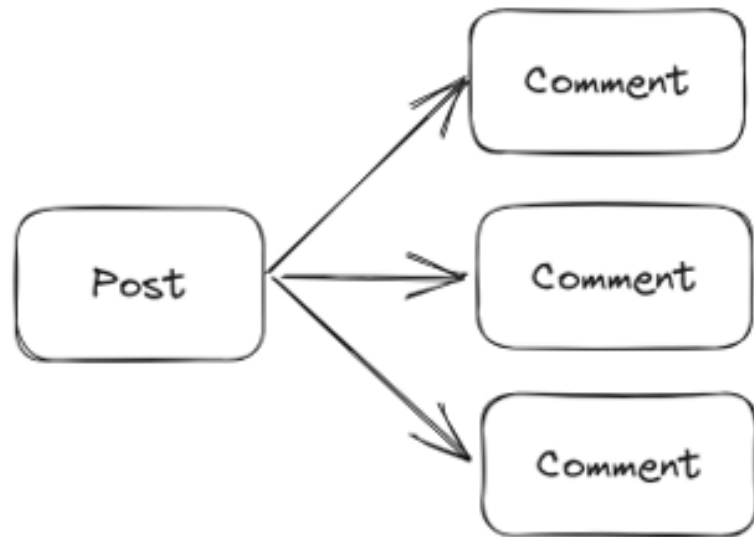
- A `Post` may have many `comments`
- A `comment` belongs to only one `Post`

SQL Schema : One-to-Many

Post Table

#	Name	Datatype
🔑 1	**id**	**INTEGER**
2	ownerId	INTEGER
3	postText	TEXT
4	createdAt	TIMESTAMP
5	updatedAt	TIMESTAMP

Comment Table

#	Name	Datatype
🔑 1	**id**	**INTEGER**
2	postId	INTEGER
3	createdAt	TIMESTAMP
4	updatedAt	TIMESTAMP
5	commentText	TEXT

SQL Query : One-to-Many

e.g., **"Get a** `Post` **together with its** `Comments` **"**

SELECT * FROM `Post` JOIN `Comment` ON `Post.Id` = `Comment.postId` ;

NoSQL Schema #1 : One-to-Many

Option 1 - Embedding `Comments` as array in `Post` document

- Assuming that a `Post` has <u>less than a hundred</u> `Comments`

```
▼ {
▶    "_id": {…},
     "postId": 2,
     "postText": "Good evening!",
▼    "comments": [
▼      {
         "commentText": "Good evening Too!",
         "username": "bob"
       },
▼      {
         "commentText": "Good evening Bro!",
         "username": "Alice"
       }
     ]
   }
```

NoSQL Schema #2 : One-to-Many

Option 2 - Reference to other collections, avoiding massive array

Post Collection

```
_id: ObjectId('649bd52d16b194f723bae06a')
postId: 2
postText: "Good evening!"
```

Comment Collection

```
_id: ObjectId('649bde5f16b194f723bae06e')
commentText: "Good evening Too!"
username: "bob"
postId:  2
```

```
_id: ObjectId('649bde9716b194f723bae06f')
commentText: "Good evening Bro!"
username: "Alice"
postId: 2
```

- Reference each `Comment` to a single `Post`
- What if a `Post` may have thousands of `Comments`

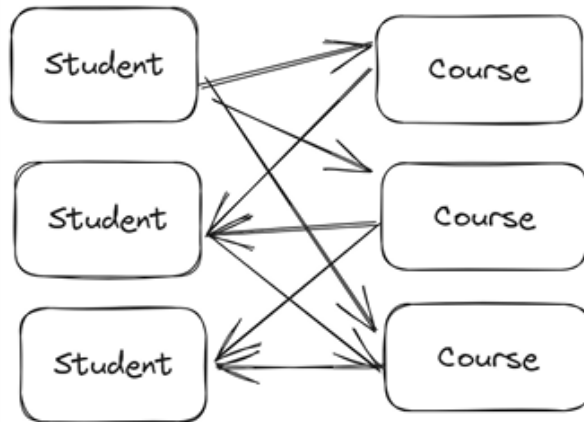# Summary : One-to-Many

SQL

- Create two tables with a `foreign key` (representing a relationship)

NoSQL

- Embedding an `array of objects` in `another type of object`
- References multiple `objects` to `another type of object`

Relationship : Many-to-Many

e.g., **"Students Enrollment"**

- A `Student` may enroll in multiple `Courses`
- A `Course` is enrolled by many `Students`

SQL Schema : Many-to-Many

SQL Query : Many-to-Many

e.g., **"Get all** `Courses` **title enrolled by a** `Student` **with specified** `studentId` **"**

student id 6106331100

Chayanin Suatap

address ....

course no	course title
261207	BASIC COMP LAB
261497	FULL STACK DEV

SQL Query : Many-to-Many

e.g., **"Get all** `Courses` **title enrolled by a** `Student` **with specified** `studentId` **"**
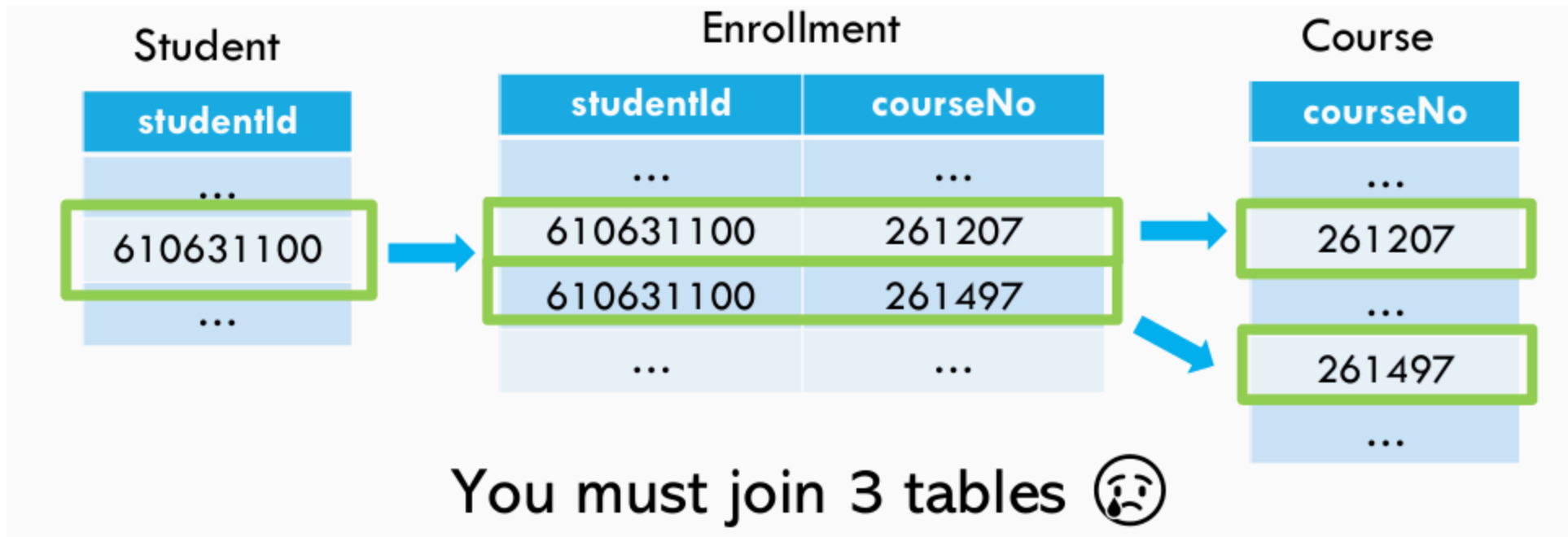
NoSQL Schema #1 : Many-to-Many

Option 1 - Embedding a list of `Courses` in a `Student` document

NoSQL Query #1 : Many-to-Many

e.g., **"Which** `Students` **enroll in my** `Course` **"**

Instructor
Master Shifu

261207 - BASIC COMP LAB
_____
610631100 Chayanin Suatap
610631101 Po

261497 - FULL STACK DEV
_____
610631101 Po
610631102 Mei Mei

NoSQL Schema #2 : Many-to-Many

Option 2 - Embedding a list of `Students` in a `Course` document

What if we want both?

NoSQL Schema #3 : Many-to-Many

Option 3 - Embedding a list of `References` in both documents

Student

```
_id: ObjectId('649c5446b325ea1ba80bc21a')
studentId: "610631100"
firstName: "Chayanin"
lastName: "Suatap"
address: "Some place on earth"
▼ coursesEnrolled: Array
    0: "261207"
    1: "261497"
```

Course

```
_id: ObjectId('649c02ad16b194f723bae081')
courseNo: "261207"
title: "BASIC COMPUTER ENGINEERING LAB"
detail: "Teaching web development using React and JavaScript"
▼ students: Array
    0: "610631100"
    1: "610631102"
```

- `Pros` : query efficiently from both sides
- `Cons` : duplicate data, need to update on both side

Summary : Many-to-Many

SQL

- Create **three** tables with `foreign keys` and `JOIN` them together

NoSQL

- Choose which side of document to be embedded by <u>determining which side has more queries</u>

- Embedded on both sides and apply `data mutation` very carefully

Data Integrity

SQL Data Integrity

Foreign Key

- Keeps data consistent across related tables
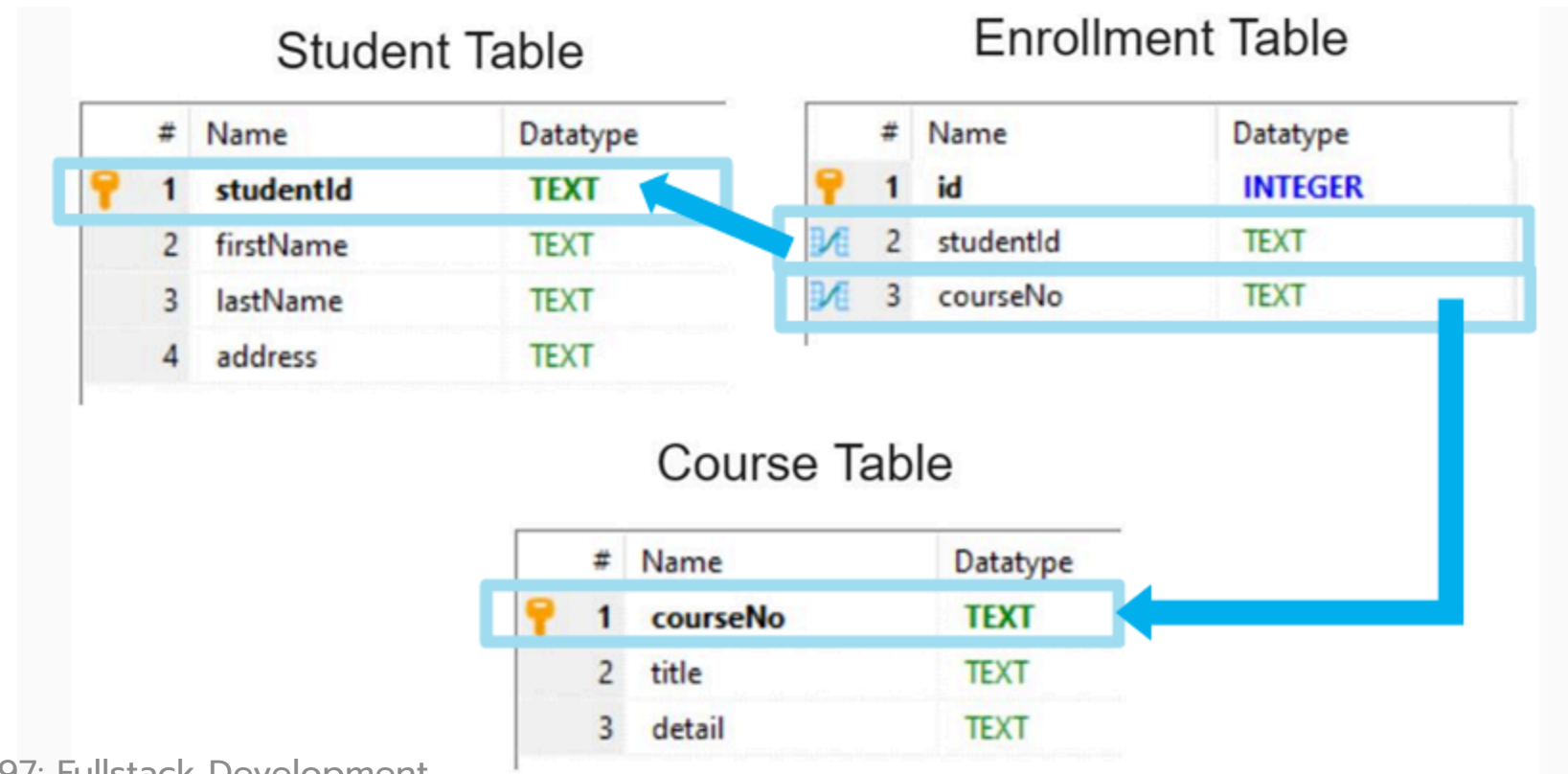
SQL Data Integrity

Data Normalization

- The process of restructuring a relational database
- Helps reducing `data redundancy` (multiple copies of the same data)
- Improves `data integrity` by <u>avoiding updating data in multiple locations</u>

Data Normalization

EmpID	Employee	Age	Dept
1001	ABC	30	Sales,Finance
1002	CDE	30	Sales,Finance,DevOps

DeptID	DeptName
1	Sales
2	Finance
3	DevOps

EmpId	Employee	Age	DepID
1001	ABC	30	1
1001	ABC	30	2
1002	CDE	40	1
1002	CDE	40	2
1002	CDE	40	3

Data Normalization

Compsite Keys

Name	Date	Title	...
AWS_101	9/17/2018	Amazon Web Services	
Azure_101	9/18/2018	SQL Azure Essentials	
DynamoDB_102	9/20/2018	DyanmoDB Advanced Concepts	
SQL_101	11/26/2018	T-SQL Essentials	
SQL_102	11/26/2018	SQL Server for DBA	
AWS_101	11/26/2018	Amazon Web Services	

The column Title is functionally dependent on Name column

Name	Date	...
AWS_101	9/17/2018	
Azure_101	9/18/2018	
DynamoDB_102	9/20/2018	
SQL_101	11/26/2018	
SQL_102	11/26/2018	
AWS_101	11/26/2018	

CourseID	Title
AWS_101	Amazon Web Services
Azure_101	SQL Azure Essentials
DynamoDB_102	DyanmoDB Advanced Concepts
SQL_101	T-SQL Essentials
SQL_102	SQL Server for DBA
AWS_101	Amazon Web Services

How far to take normalization in SQL database?

- This question is **opinion-based**
- Query *more tables* is typically slower (due to more `JOIN` operations)
- Normalize as far as necessary to **remove data integrity issues**
    - Potential data duplication or missing data

Data normalization in NoSQL database

- NoSQL prefers `denormalization`
  - Accept data duplication to improve querying speed
  - `Insert` / `Update` / `Delete` must be performed carefully
- No `foreign key` mechanism built-in

Schema and Data Type Safety

Schema validation

- **SQL** has built-in schema and data type validation (duhh!!!)
- **NoSQL** database usually allows you to annotate and validate JSON documents
  - `MongoDB` uses `JSON schema` to specify validation rules when creating a collection
- Some `ORMs` can be used to defines a schema for **NoSQL**
  - This ORM helps validating data during coding
  - `Prisma ORM` defines a schema for `MongoDB` , providing type safety
  - Unfortunately, `Drizzle` does not natively support MongoDB

MongoDB JSON Schema

```javascript
db.createCollection("students", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            title: "Student Object Validation",
            required: [ "address", "major", "name", "year" ],
            properties: {
                name: {
                    bsonType: "string",
                    description: "'name' must be a string and is required"
                },
                year: {
                    bsonType: "int",
                    minimum: 2017,
                    maximum: 3017,
                    description: "'year' must be an integer in [ 2017, 3017 ] and is required"
                },
                gpa: {
                    bsonType: [ "double" ],
                    description: "'gpa' must be a double if the field exists"
                }
            }
        }
    }
} )
```

Prisma ORM Schema

```
model Post {
  id        String    @id @default(auto()) @map("_id") @db.ObjectId
  slug      String    @unique
  title     String
  body      String
  author    User      @relation(fields: [authorId], references: [id])
  authorId  String    @db.ObjectId
  comments  Comment[]
}

model User {
  id        String    @id @default(auto()) @map("_id") @db.ObjectId
  email     String    @unique
  name      String?
  address   Address?
  posts     Post[]
}
```

```
model Comment {
  id        String @id @default(auto()) @map("_id") @db.ObjectId
  comment   String
  post      Post   @relation(fields: [postId], references: [id])
  postId    String @db.ObjectId
}

// Address is an embedded document
type Address {
  street String
  city   String
  state  String
  zip    String
}
```

Common Database Patterns

Soft DELETE

To delete a row, marks the `status` field as `false`.

Course Table

#	Name	Datatype
🔑 1	**courseNo**	TEXT
2	title	TEXT
3	detail	TEXT
4	status	BOOLEAN

Soft DELETE

- Able to view history of data

- Sensitive data remains in the database

- Undeletion is possible

Cons

- Every query must has `where status = true` condition

- `Size` of **table / collection** is larger

Created-At and Updated-At

"Tracking `when` a row was **created** or **updated**"

Course Table

#	Name	Datatype
🔑 1	**courseNo**	TEXT
2	title	TEXT
3	detail	TEXT
4	status	BOOLEAN
5	createdAt	TIMESTAMP
6	updatedAt	TIMESTAMP

Created-By and Updated-By

"Tracking `who` has interacted with the data"

#	Name	Datatype
🔑 1	**courseNo**	TEXT
2	title	TEXT
3	detail	TEXT
4	status	BOOLEAN
5	createdAt	TIMESTAMP
6	updatedAt	TIMESTAMP
7	createdBy	INTEGER
8	updatedBy	INTEGER

NoSQL Pattern

Example: Social media latest [#] posts

"Get the latest [#] Posts of specified User in one query"
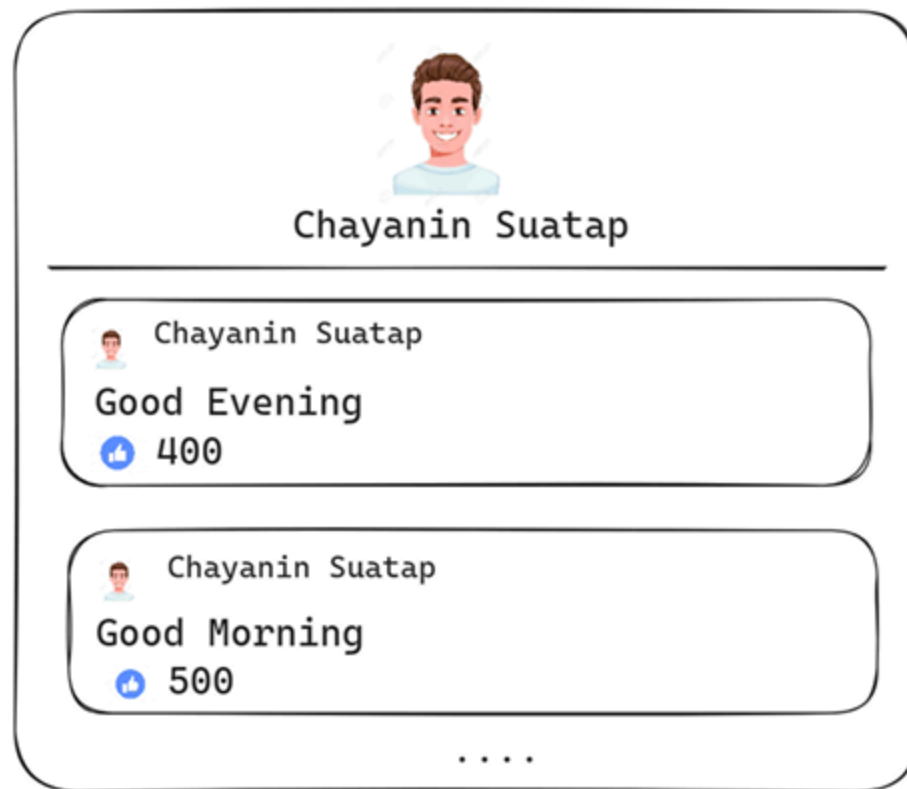
Example: Social media latest [#] posts

User Collection

```
username: "Arm",
password: "xxxx",
birthDate: "xxx",
latest5Posts : [{
        postText:" Good Evening",
        likeNum: 400
    },
    {

        postText:" Hi bro",
        likeNum: 500

    }
    ...
]
```

Comment Collection

```
{

    username: "Arm"
    postText:"Good Evening",
    likeNum: 400

}
```

```
{

    username: "Arm"
    postText:"Good Morning",
    likeNum: 500

}
```

...

Database Index

What is Index?

- A **data structure** that <u>improves the speed of data retrieval operations</u> on `database table`

- `Index` costs **additional writes** and **storage space** to maintain the index data structure

Index Example

"Get `instructors` with `Finance` discipline"

Index Example

"Get `instructors` ordered by `Salary` "

Composite Index

"Get `Customers` ordered by `Name` and `Town` "
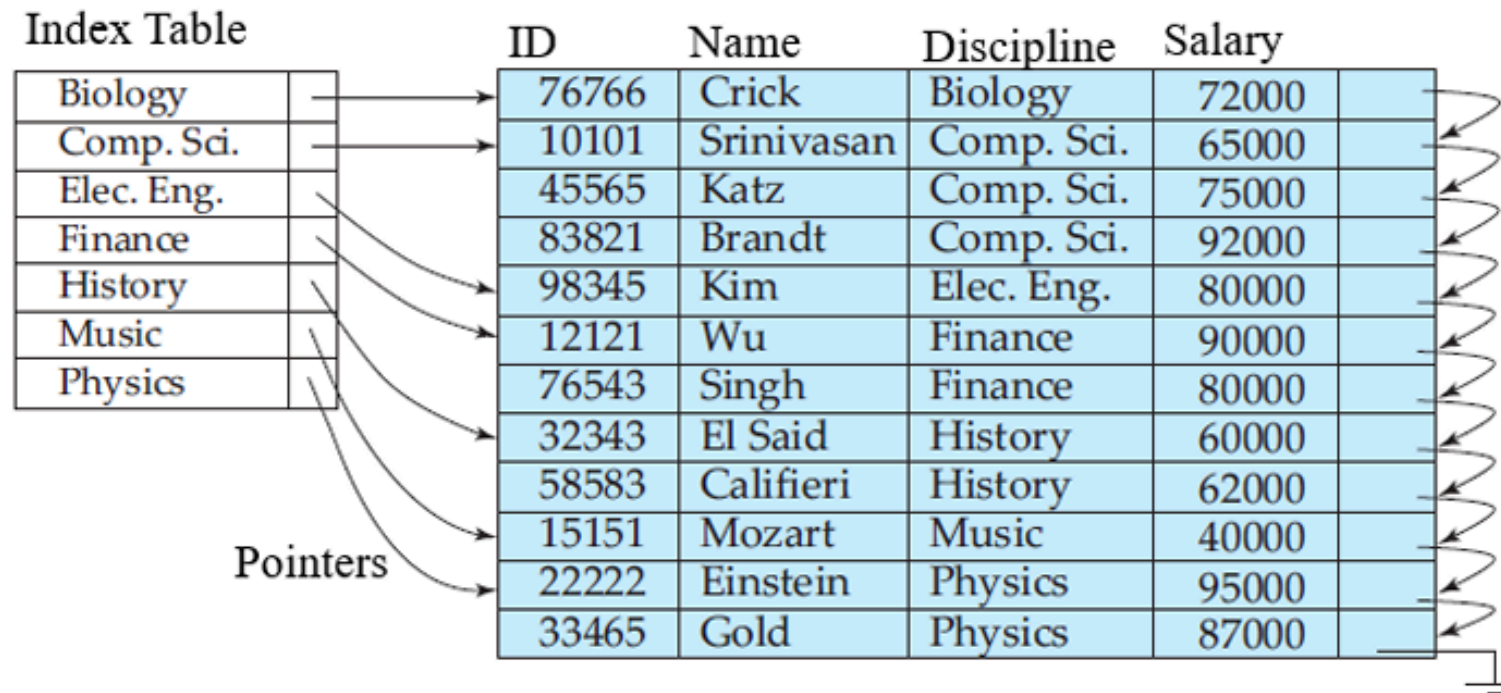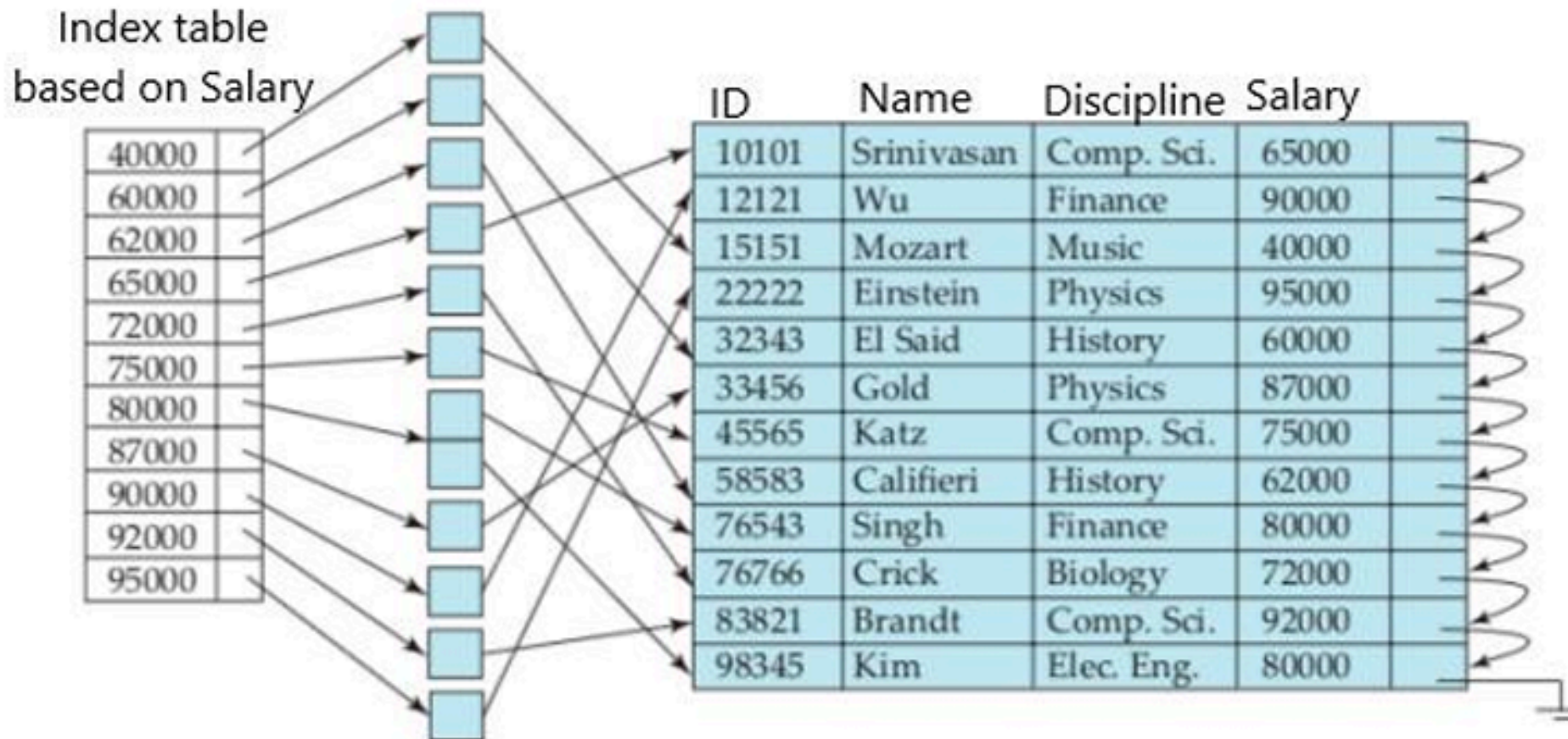

Fact Table

Primary Key (Customer ID)	Customer Data
1	LastName: Smith, Town: Redmond, ...
2	LastName: Jones, Town: Seattle, ...
3	LastName: Robinson, Town: Portland, ...
4	LastName: Brown, Town: Redmond, ...
5	LastName: Smith, Town: Chicago, ...
6	LastName: Green, Town: Redmond, ...
7	LastName: Clarke, Town: Portland, ...
8	LastName: Smith, Town: Redmond, ...
9	LastName: Jones, Town: Chicago, ...
...	...
1000	LastName: Clarke, Town: Chicago, ...
...	...

Index Table

Secondary Key (Town)	Customer Reference (ID)
Chicago	ID: 5
Chicago	ID: 9
Chicago	ID: 1000
...	...
Portland	ID: 3
Portland	ID: 7
Redmond	ID: 1
Redmond	ID: 4
Redmond	ID: 6
Redmond	ID: 8
Seattle	ID: 2
...	...

Index Table

Secondary Key (LastName)	Customer Reference (ID)
Brown	ID: 4
Clarke	ID: 7
Clarke	ID: 1000
Green	ID: 6
Jones	ID: 2
Jones	ID: 9
...	...
Robinson	ID: 3
Smith	ID: 1
Smith	ID: 5
Smith	ID: 8
...	...

Unique Index

- Unique index is used to <u>ensure data uniqueness</u>
- e.g., To ensure that a `Student` cannot enroll the same `Course` twice
  - Create a `compound unique index` containing `studentId` and `courseNo`

Enrollment Table

#	Name	Datatype
🔑 1	**id**	**INTEGER**
2	studentId	TEXT
3	courseNo	TEXT

Index Summary

Pros

- Query data more efficiently, speed up `SELECT` operation

Cons

- <u>Index must be rebuilt</u> when `INSERT` / `UPDATE` / `DELETE`, hence slower
- Index requires <u>addtional storage space on database</u>

Unique Identifier

- A value that <u>distinquishes a **specific record** (row, document) from others within the table</u>

- Prevent ambiguity and enabling efficient data retrieval and management

Auto Increment ID

SQL database only

- MySQL uses the `AUTO_INCREMENT` keyword

```sql
CREATE TABLE table_name (
    id INT AUTO_INCREMENT PRIMARY KEY,
    column2 VARCHAR(255),
    column3 INT,
    -- Add other columns as needed
);
```

Auto Increment ID

- PostgreSQL uses the `SERIAL` or `SEQUENCE` keywords

```
CREATE TABLE my_table (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255)
);
```

```
CREATE SEQUENCE my_sequence START 100 INCREMENT 5;

CREATE TABLE another_table (
    id INTEGER NOT NULL DEFAULT nextval('my_sequence'),
    description TEXT
);

ALTER SEQUENCE my_sequence OWNED BY another_table.id;
```

UUID

Universally Unique ID

- Not able to find the same `UUID` in the same Universe
- The term **Globally Unique Identifier** (`GUID`) is also used
  - Used mostly in Microsoft systems
- `UUID 4` is widely used

UUID v4

- The most commonly used and recommended for general-purpose applications
  - Due to its **simplicity** and reliance on **pseudo-random number generator**
- 128-bit value (32-Hexadecimal)
- Formated as 5-group of characters (8-4-4-4-12)

c9b135f6-f163-40d6-8483-0a57780e3f17

What to use as Primary Keys?

Auto-increment IDs vs. UUIDs ?

```
ID  Value              ID                                      Value
--  -----              ------------------------------------    -----
1   Apple              C87FC84A-EE47-47EE-842C-29E969AC5131    Apple
2   Orange      vs.    2A734AE4-E0EF-4D77-9F84-51A8365AC5A0    Orange
3   Pear               70E2E8DE-500E-4630-B3CB-166131D35C21    Pear
4   Mango              15ED815C-921C-4011-8667-7158982951EA    Mango
```

UUID Pros and Cons

Pros

- Unique across every table, database, every server
- Easy merging of records form different databases
- Easy distribution of databases across multiple servers, aka. **sharding**

Cons

- Larger than tradition typically ID
- This can have serious performance and storage implications
- May be difficult to debug

```
... where userid = '{BAE7DF4-DDF-3RG-5TY3E3RF456AS10}'
```

MongoDB Object ID

- Insert automatically in every document
- Embedded `timestamp` inside the ID
- Can be used to sort as `create date`