Assignment 1

Submit the GIT repo link as a submission on Openverse

1. Given the table:
    1. orders

id	user_id	status	created_at

2. Suggest appropriate indexes
3. Explain why you chose them

Table

orders

Column	Type (assumed)
id	primary key
user_id	foreign key
status	string / enum
created_at	timestamp

Suggested Indexes

1. Primary Key Index on id

```
PRIMARY KEY (id)
```

Why this index is needed

- Every row must be uniquely identifiable.
- Most queries that fetch a single order use the id.
- The database automatically creates this index.

Benefit

- Extremely fast lookup of a single order.
- Ensures no duplicate order IDs exist.

Example

> **This works like a unique roll number for each order, making it easy and fast to find.**

2. Index on `user_id`

```
CREATE INDEX idx_orders_user_id ON orders(user_id);
```

Why this index is needed

- Applications frequently fetch all orders of a user.
- Common query:

```
SELECT * FROM orders WHERE user_id = ?;
```

- Without an index, the database would scan the entire table.

Benefit

- Faster retrieval of user-specific orders.
- Improves performance when joining with the `users` table.

Example: Join Query Using `user_id`

Query

SELECT

u.id,

u.name,

o.id AS order_id,

o.status,

o.created_at

FROM users u

JOIN orders o

ON u.id = o.user_id

WHERE u.id = 123;

Why the `user_id` Index Helps Here

What happens without an index on `orders.user_id`

- Database scans **all rows** in the `orders` table.
- For each order, it checks if `user_id = 123`.
- This is slow when the orders table is large.

👉 This is called a **full table scan**.

What happens with an index on `orders.user_id`

CREATE INDEX idx_orders_user_id ON orders(user_id);

- Database directly jumps to orders with `user_id = 123`.
- No need to scan unrelated orders.
- Join becomes much faster.

👉 This is called an **index lookup**.

Data Size Example

- Users table: 10,000 users
- Orders table: 1,000,000 orders

Without index:

- Database checks all 1,000,000 orders

With index:

- Database checks only the orders belonging to that user (maybe 20–30 rows)

---

3. Composite Index on (`user_id`, `created_at`)

```
CREATE INDEX idx_orders_user_created ON orders(user_id, created_at);
```

Why this index is needed

Most applications show a user's orders sorted by date.

Common query:

```
SELECT *
FROM orders
WHERE user_id = ?
ORDER BY created_at DESC;
```

This query does **two things**:

1. Filters rows using `user_id`
2. Sorts the result using `created_at`

Case 1: No Index

What the database does:

- Scans the entire `orders` table
- Filters rows where `user_id = 101`
- Sorts the filtered result by `created_at`

Problem:

- Full table scan
- Expensive sorting step

Case 2: Separate Indexes

CREATE INDEX idx_user_id ON orders(user_id);
CREATE INDEX idx_created_at ON orders(created_at);

What the database does:

- Uses `idx_user_id` to find orders for user 101
- Still needs to **sort the result** using memory or disk
- Cannot efficiently combine both indexes for this query

Problem:

- Sorting still required
- Extra CPU and memory usage

Case 3: Composite Index (Best Case)

CREATE INDEX idx_user_created ON orders(user_id, created_at);

How the composite index is stored

The data in the index is already ordered like this:

user_id	created_at
101	2024-01-10
101	2024-01-05
101	2024-01-01
102	2024-01-08

What the database does now

- Uses the index to **directly find user_id = 101**
- Rows are **already sorted by created_at**
- No extra sorting step needed

Result:

- Fast filtering

- Fast sorting
- Minimal resource usage

Simple Analogy

- **Separate indexes**:
  Find all pages for “Chapter 3”, then sort them manually.
- **Composite index**:
  Chapter 3 pages are already in order.

Performance Benefit Summary

Index Type	Filtering	Sorting	Performance
No index	❌	❌	Very slow
Separate indexes	✅	❌	Medium
Composite index	✅	✅	Fastest