Query Optimization Strategy

To ensure **scalability** and **high performance** for subscription data access, several targeted optimizations were implemented:

1. Raw SQL for Performance-Critical Paths

- I used **SQLAlchemy Core with raw SQL** instead of ORM `.query.filter()` for subscription listing and history retrieval.
- This avoids SQLAlchemy's overhead and gives full control over the query structure.

```
SELECT id, plan_id, start_date, end_date
FROM user_subscription
WHERE user_id = :user_id
ORDER BY start_date DESC
```

2. Indexing Strategy

Indexes were carefully added to **match query filters and sorting** conditions:

Index Name	Columns	Purpose
`idx_user_active`	(`user_id`, `is_active`)	Speeds up lookup for active subscriptions
`idx_user_start_date`	(`user_id`, `start_date`)	Optimizes order-by for listing history
`idx_user_plan`	(`user_id`, `plan_id`)	Supports filtering or joining by plan
Recommended	`end_date`	Needed for fast access to historical data

3. Pagination with offset-limit

- Endpoints that return lists (e.g., active subscriptions, historical subscriptions) use LIMIT and OFFSET to reduce result set size and improve latency:

  LIMIT :page_size OFFSET :offset

- Separate total count queries: Count queries are isolated from data-fetching queries to improve clarity and avoid unnecessary joins:

  SELECT COUNT(*) FROM user_subscription WHERE user_id = :user_id AND is_active = TRUE

4. Bulk Insert/Seed Optimization

- Data seeding is done using **batched inserts** to avoid per-row overhead.
- Created foreign key references for `user` and `plan` before seeding subscriptions to maintain integrity.

5. Query Performance Testing

- Included benchmark tests using `pytest` to simulate querying against a database with **50,000+ rows**.
- Visual comparisons (see `images/`) demonstrate the performance gains when indexes are in place.

Why `end_date IS NOT NULL`?

- Used to separate **historical subscriptions** from active ones.
- This condition is selective; indexing `end_date` improves speed significantly when querying history.

Pagination Example in Active Subscriptions

- Total count and paginated results are queried **separately** for flexibility and performance.
- This approach supports scalable frontend pagination, load more, or infinite scroll.

```
# Repository layer
get_active_subscriptions_paginated(user_id, page, page_size)
count_active_subscriptions(user_id)

# Service layer
{
  "subscriptions": [...],
  "total": 128,
  "page": 2,
  "page_size": 10
}
```

ORM Overhead in Subscription Handling

Overhead in ORM Queries

- **Session Management**: ORM maintains a session and tracks object states (e.g., dirty-checking), which adds memory and CPU usage.
- **Object Construction**: Each database row is converted into a Python object, even if I only need raw data.
- **Lazy Relationships**: ORM loads relationships lazily by default, which can result in **N+1 query problems** unless explicitly optimized.
- **Join Complexity**: ORM-generated SQL for relationships can be verbose and suboptimal for large datasets.

Example (inefficient with ORM):

```
subscriptions =
UserSubscription.query.filter_by(user_id=1).order_by(UserSubscription.start_date
.desc()).all()
```

- Loads full ORM models into memory.
- Triggers unnecessary relationship loading unless optimized.
- Harder to control `SELECT` columns and query plan.

Optimization Decision

In this project:

- We used **SQLAlchemy Core with raw SQL** for:
  - `list_all_subscriptions`
  - `get_subscription_history`
- This bypasses the ORM session and constructs only what we need—**rows, not objects**.

Result:

- Lower latency for read-heavy endpoints.
- Faster data access and reduced memory usage, especially when handling **50K+ subscriptions**.