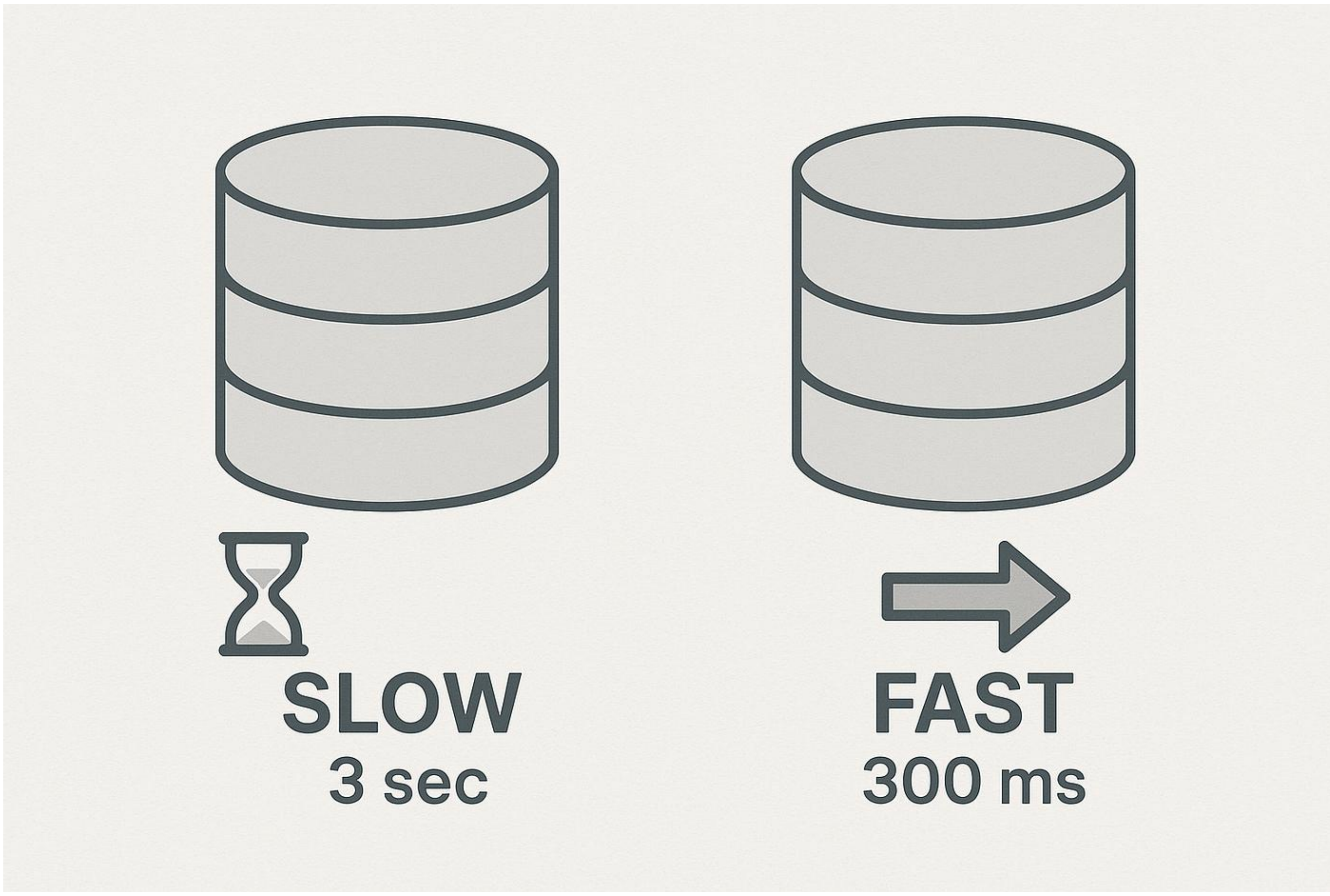
SQL QUERY OPTIMIZATION



Let me share a quick story from our backend team.

We were working on this fairly simple API it fetched a user's transaction history with pagination. Pretty standard stuff. For the first few months, it worked smoothly. But as data kept growing, the same query that used to return in 200ms… started taking 2 to 3 **seconds**.

At first, we thought it was just load or maybe a missing index. But after digging deeper and checking the DB execution plan, we found something super basic and also super expensive.

It was the OFFSET.

OFFSET Pagination: The Silent Killer

Here's the query we were using:

```sql
SELECT * FROM transactions
WHERE user_id = 42
ORDER BY created_at DESC
LIMIT 20 OFFSET 10000;
```

Looks clean, right? But here's what really happens behind the scenes:

The database fetches **10,020 rows**, then throws away the first 10,000, and only returns the last 20. So even though we're only showing 20 records to the user, the DB still has to do a whole lot of work.

And with every scroll or pagination, OFFSET keeps growing… and performance keeps dropping. The bigger the data, the worse it gets.

The Fix: Keyset Pagination

After trying a few optimizations that didn't move the needle much, we rewrote the query using keyset pagination and that changed everything.

Here's how the new version looked:

```sql
SELECT * FROM transactions
WHERE user_id = 42
  AND created_at < '2024-05-01 10:00:00'
ORDER BY created_at DESC
LIMIT 20;
```

So instead of saying "skip the first 10,000 rows," we just tell the database: "Give me the next 20 rows after this timestamp."

This is called **seek pagination** or **keyset pagination**.

It's much more efficient because the DB doesn't have to scan and throw away anything. It just uses the index to jump directly to the right place.

After making this change, our API response time dropped from **2.6 seconds to under 200ms** no infra changes, no caching, just smarter SQL.

How We Handled Duplicate Timestamps

One small issue we faced: sometimes multiple transactions had the exact same `created_at` timestamp (especially when users bulk-uploaded data).

This caused pagination glitches like repeated or skipped rows when we paged based only on `created_at`.

So, we fixed it like this:

```
WHERE (created_at, id) < ('2024-05-01 10:00:00', 98765)
ORDER BY created_at DESC, id DESC
```

By adding a **tie-breaker (`id`)** in both the `ORDER BY` and the `WHERE` clause, the pagination became stable and predictable.

Some Other Approaches We Considered

Although keyset pagination solved most of our problems, we explored a few other ideas too each one useful depending on the use case.

1. Cursor-Based Pagination

This is essentially keyset pagination, but instead of passing raw timestamps or IDs in the API, you wrap them in a cursor token like:

```
"next_cursor": "2024-05-01T10:00:00Z_98765"
```

This is exactly how Instagram, Twitter, and most modern APIs handle scrolling. It keeps things clean, stateless, and efficient.

2. Index-Only OFFSET

NOTE: When You Have To Use OFFSET

In a few internal tools (like admin dashboards), we *had* to allow jumping to any page like page 7 or 10. In such cases, keyset doesn't work.

So what we did was:

- **Add a covering index** on the fields used in the query
- Only select columns from the index itself

```
CREATE INDEX idx_user_created_id_amount
ON transactions(user_id, created_at DESC, id, amount);
```

This doesn't make OFFSET fast but it does make it faster than it was. If you **must** use OFFSET, this is the least painful way.

3. Materialized Views

NOTE: For Static Dashboards

One of our reporting dashboards kept hitting the same slow query again and again summarizing user transactions daily.

We solved it with a materialized view:

```
CREATE MATERIALIZED VIEW user_summary AS
SELECT user_id, DATE(created_at), SUM(amount)
FROM transactions
GROUP BY user_id, DATE(created_at);
```

We refreshed the view every few minutes using a cron. This made the reports snappy and reduced load on the live tables.

Actual Results After the Fix

Here's how our query time changed after each improvement:

Query TypeAvg Response TimeOFFSET 10000~2600 msOFFSET + index~1300 msKeyset/Seek method**~180 ms**Keyset + cursor token**~190 ms**Materialized View~**50–100 ms**

We didn't expect such a massive difference from just changing the SQL style but it honestly made a huge impact.

Final Thoughts

This experience reminded us that performance doesn't always require big changes or expensive upgrades.

Sometimes, **a simple rewrite of your SQL query** can save you more time, cost, and frustration than throwing resources at the problem.

So if your app is using OFFSET-based pagination and you're seeing slowness at scale I'd highly recommend switching to keyset pagination.

It's simple, elegant, and highly efficient.

Hope this helps someone avoid the rabbit hole we fell into