SQL vs ORM: When to Use Each

The Uncomfortable Truth

ORMs are not bad. But they're not magic.

They're a **trade-off**:

- ✅ **Convenience:** Type-safe queries, migrations, easier to write
- ❌ **Abstraction leaks:** Hidden performance issues, limited expressiveness

The goal: Use ORMs for 90% of cases, drop to SQL for the other 10%.

What ORMs Do Well

1. Basic CRUD

ORM (Prisma):

```
const user = await prisma.user.findUnique({ where: { id: 123 } });
await prisma.user.update({ where: { id: 123 }, data: { name: 'Alice' } });
await prisma.user.delete({ where: { id: 123 } });
```

SQL:

```
SELECT * FROM users WHERE id = 123;
UPDATE users SET name = 'Alice' WHERE id = 123;
DELETE FROM users WHERE id = 123;
```

Winner: ORM. Cleaner, type-safe, less boilerplate.

2. Migrations

ORM (Prisma Migrate):

```
// schema.prisma
model User {
  id    Int    @id @default(autoincrement())
  email String @unique
}
```

Run: `npx prisma migrate dev`

SQL (manual):

```
CREATE TABLE users (id SERIAL PRIMARY KEY, email TEXT UNIQUE NOT NULL);
-- Track migrations manually, write rollback scripts, etc.
```

Winner: ORM. Declarative schema, auto-generated migrations, version control.

3. Simple Joins

ORM:

```
const users = await prisma.user.findMany({
  include: { orders: true }
});
```

SQL:

```
SELECT u.*, o.* FROM users u LEFT JOIN orders o ON u.id = o.user_id;
```

Winner: Tie. ORM is slightly cleaner, SQL is more explicit.

4. Type Safety (TypeScript + Prisma)

ORM:

```
const user = await prisma.user.findUnique({ where: { id: 123 } });
// `user` is User | null, with full type info
console.log(user?.email);  // TypeScript knows email exists
```

SQL:

```
const result = await db.query('SELECT * FROM users WHERE id = $1', [123]);
const user = result.rows[0];
// `user` is `any`, no type safety
```

Winner: ORM. Type safety is a huge win.

What ORMs Struggle With

1. N+1 Queries

The Problem:

```
const users = await prisma.user.findMany();
for (const user of users) {
  const orders = await prisma.order.findMany({ where: { userId: user.id } });
  console.log(`${user.name}: ${orders.length}`);
}
```

SQL generated:

```
SELECT * FROM users;
SELECT * FROM orders WHERE user_id = 1;
SELECT * FROM orders WHERE user_id = 2;
-- ... N queries
```

Problem: N+1 database round-trips.

Solution (ORM):

```
const users = await prisma.user.findMany({
  include: { orders: true }
});
users.forEach(user => console.log(`${user.name}: ${user.orders.length}`));
```

SQL generated:

```
SELECT * FROM users u LEFT JOIN orders o ON u.id = o.user_id;
```

One query, but...

Problem: Join explosion. If a user has 10 orders, they appear 10 times in the result set, and Prisma de-duplicates in memory.

Better SQL:

```
SELECT u.name, COUNT(o.id) AS order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id, u.name;
```

One query, aggregated in the database.

ORM can't generate this. You need raw SQL or a complex workaround.

2. Complex Aggregations

Goal: "Users with >10 orders in the last 30 days."

ORM (Prisma):

```
const users = await prisma.user.findMany({
  where: {
    orders: {
      some: {
        createdAt: { gte: new Date(Date.now() - 30 * 86400000) }
      }
    }
  }
});
// Wait, this returns users with at least 1 order, not >10
```

Prisma can't express "count > 10" in a filter.

Workaround (fetch all, filter in JS):

```
const users = await prisma.user.findMany({
  include: {
```

```
    orders: {
      where: { createdAt: { gte: new Date(Date.now() - 30 * 86400000) } }
    }
  }
});
const filtered = users.filter(u => u.orders.length > 10);
```

Problem: Fetches all users and orders, filters in memory. Terrible performance.

SQL:

```
SELECT u.*
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE o.created_at > NOW() - INTERVAL '30 days'
GROUP BY u.id
HAVING COUNT(*) > 10;
```

One query, filtered in the database.

Winner: SQL.

3. Window Functions

Goal: "Each order and the user's total order count."

ORM: Can't generate window functions.

SQL:

```
SELECT
  o.id,
  o.total,
  COUNT(*) OVER (PARTITION BY o.user_id) AS user_order_count
FROM orders o;
```

Workaround (ORM):

```
const orders = await prisma.order.findMany({ include: { user: true } });
const userCounts = await prisma.order.groupBy({
  by: ['userId'],
  _count: true
});
// Merge in memory...
```

Ugly, slow, verbose.

Winner: SQL.

4. CTEs (WITH Clauses)

Goal: Multi-stage query.

ORM: Can't generate CTEs.

SQL:

```
WITH high_value_users AS (
  SELECT user_id FROM orders GROUP BY user_id HAVING SUM(total) > 10000
)
SELECT u.* FROM users u
JOIN high_value_users hvu ON u.id = hvu.user_id;
```

ORM workaround: Multiple queries, merge in JS.

Winner: SQL.

5. Raw SQL with Complex JOINs

Goal: "Products sold in last 30 days, with category and total sales."

ORM (Prisma):

```
const products = await prisma.product.findMany({
  include: {
    category: true,
    orderItems: {
      where: {
        order: { createdAt: { gte: new Date(Date.now() - 30 * 86400000) } }
      }
    }
  }
});
// Compute total sales in JS
products.forEach(p => {
  p.totalSales = p.orderItems.reduce((sum, item) => sum + item.quantity *
item.price, 0);
});
```

Problem: Fetches all order items, aggregates in memory.

SQL:

```
SELECT
  p.id,
  p.name,
  c.name AS category_name,
  COALESCE(SUM(oi.quantity * oi.price), 0) AS total_sales
FROM products p
JOIN categories c ON p.category_id = c.id
LEFT JOIN order_items oi ON p.id = oi.product_id
LEFT JOIN orders o ON oi.order_id = o.id AND o.created_at > NOW() - INTERVAL '30
days'
GROUP BY p.id, p.name, c.name;
```

One query, aggregated in the database.

Winner: SQL.

When to Use Raw SQL

✅ **Use raw SQL when:**

1. **Complex aggregations** (GROUP BY with HAVING, multiple aggregates)
2. **Window functions** (ROW_NUMBER, RANK, running totals)
3. **CTEs** (multi-stage queries)
4. **Performance-critical queries** (ORM-generated SQL is suboptimal)
5. **Database-specific features** (JSONB, arrays, full-text search)
6. **Reporting / analytics** (complex joins, subqueries)

✅ **Use ORM when:**

1. **Simple CRUD** (90% of cases)
2. **Straightforward JOINs** (1-2 tables, no aggregation)
3. **Type safety matters** (TypeScript)
4. **Rapid development** (less boilerplate)

How to Drop Down to SQL Safely

Option 1: Raw Queries with Parameter Binding

Prisma:

```
const users = await prisma.$queryRaw<User[]>`
  SELECT * FROM users WHERE email = ${email}
`;
```

Benefits:

- **SQL injection protection** (parameterized)
- **Type hints** (with `<User[]>`)

Drawbacks:

- **No type safety on columns** (you assert the type)

Option 2: Execute Raw SQL

Prisma:

```
await prisma.$executeRaw`
  UPDATE users SET last_login = NOW() WHERE id = ${userId}
`;
```

Use for: INSERT/UPDATE/DELETE.

Option 3: Hybrid Approach

Use ORM for writes, SQL for reads:

```
// Write with ORM (type-safe, migrations)
await prisma.user.create({ data: { email, name } });

// Read with SQL (performance)
const stats = await prisma.$queryRaw`
  SELECT
    DATE_TRUNC('day', created_at) AS day,
    COUNT(*) AS user_count
  FROM users
  WHERE created_at > NOW() - INTERVAL '30 days'
  GROUP BY day
  ORDER BY day;
`;
```

Option 4: Views + ORM

Create a database view:

```
CREATE VIEW user_stats AS
SELECT
  u.id,
  u.name,
  COUNT(o.id) AS order_count,
  COALESCE(SUM(o.total), 0) AS total_spent
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id, u.name;
```

Query with ORM:

```
// Prisma sees it as a table
const stats = await prisma.userStats.findMany();
```

Benefits:

- **Complex SQL in the view**
- **ORM for querying**

Real-World Scenario: E-Commerce Dashboard

Goal: Show sales stats.

ORM Approach (Naive)

```
const orders = await prisma.order.findMany({
  where: { createdAt: { gte: startDate, lte: endDate } },
  include: { items: { include: { product: true } } }
});

// Compute stats in JS
```

```
const totalRevenue = orders.reduce((sum, o) => sum + o.total, 0);
const topProducts = /* complex reduce logic */;
```

Problems:

- Fetches all orders and items (huge payload)
- Aggregates in memory (slow)

SQL Approach

```
const stats = await prisma.$queryRaw`
  SELECT
    COUNT(DISTINCT o.id) AS order_count,
    SUM(o.total) AS total_revenue,
    JSONB_AGG(
      JSONB_BUILD_OBJECT('product', p.name, 'quantity', SUM(oi.quantity))
      ORDER BY SUM(oi.quantity) DESC
    ) FILTER (WHERE p.id IS NOT NULL) AS top_products
  FROM orders o
  LEFT JOIN order_items oi ON o.id = oi.order_id
  LEFT JOIN products p ON oi.product_id = p.id
  WHERE o.created_at BETWEEN ${startDate} AND ${endDate}
`;
```

Benefits:

- One query
- Aggregated in database
- Returns minimal payload

Winner: SQL.

ORM Anti-Patterns

Anti-Pattern 1: Fetching Everything, Filtering in JS

Bad:

```
const allUsers = await prisma.user.findMany();
const activeUsers = allUsers.filter(u => u.active);
```

Good:

```
const activeUsers = await prisma.user.findMany({ where: { active: true } });
```

Lesson: Push filters to the database.

Anti-Pattern 2: Sequential Queries in Loops

Bad:

```
for (const user of users) {
  const orders = await prisma.order.findMany({ where: { userId: user.id } });
  // ...
}
```

Good:

```
const users = await prisma.user.findMany({
  include: { orders: true }
});
```

Lesson: Use eager loading.

Anti-Pattern 3: Over-Including

Bad:

```
const users = await prisma.user.findMany({
  include: { orders: true, addresses: true, profile: true }
});
```

Problem: Join explosion (Cartesian product).

Good:

```
// Fetch separately or aggregate
const users = await prisma.user.findMany();
const orderCounts = await prisma.order.groupBy({
  by: ['userId'],
  _count: true
});
// Merge in JS
```

Or just use SQL.

Anti-Pattern 4: Not Using Indexes

Bad:

```
const users = await prisma.user.findMany({
  where: { email: { contains: 'example' } }  // Full table scan!
});
```

Good:

```
// Add index: @@index([email])
const users = await prisma.user.findMany({
  where: { email: { startsWith: 'example' } }  // Can use index prefix scan
});
```

Lesson: Understand how your queries use indexes.

Tools for Mixing ORM + SQL

Prisma

Strengths:

- Excellent TypeScript integration
- Schema-first (declarative)
- Good for CRUD

Weaknesses:

- Limited query expressiveness
- Can't generate CTEs, window functions

When to drop to SQL: Complex queries, aggregations, analytics.

Drizzle

Strengths:

- SQL-like syntax (closer to raw SQL)
- More expressive than Prisma
- Type-safe

Weaknesses:

- Smaller ecosystem
- Less mature

When to drop to SQL: Complex CTEs, window functions.

TypeORM / Sequelize

Strengths:

- Active Record / Data Mapper patterns
- Established ecosystems

Weaknesses:

- Verbose
- Less type-safe than Prisma/Drizzle

When to drop to SQL: Same as Prisma.

Kysely

Strengths:

- SQL query builder (not full ORM)
- Very flexible, type-safe

Weaknesses:

- No migrations, schema management (DIY)

When to use: When you want **SQL expressiveness + type safety** without ORM magic.

The Pragmatic Approach

1. Use ORM for Writes

Writes are usually simple:

- INSERT new records
- UPDATE by ID
- DELETE by ID

ORM handles these well:

```
await prisma.user.create({ data: { email, name } });
await prisma.user.update({ where: { id }, data: { name } });
```

2. Use SQL for Reads (When Complex)

Reads are where complexity lives:

- Aggregations
- Multi-table JOINs
- Filtering on aggregate results

SQL handles these better:

```
const stats = await prisma.$queryRaw`...`;
```

3. Profile Everything

Don't guess. Measure.

Check the generated SQL:

```
// Prisma
const users = await prisma.user.findMany({ where: { active: true } });
// Log: SELECT * FROM users WHERE active = true
```

Run EXPLAIN ANALYZE:

```
EXPLAIN ANALYZE <generated-query>;
```

If slow: Rewrite in raw SQL.

4. Create Abstractions

Wrap complex SQL in functions:

```
async function getUserStats(startDate: Date, endDate: Date) {
  return prisma.$queryRaw`
    SELECT ...
```

```
  `;
}
```

Benefits:

- Reusable
- Testable
- Hides complexity

Key Takeaways

1. **ORMs excel at CRUD, migrations, and type safety.**
2. **ORMs struggle with aggregations, window functions, and complex JOINs.**
3. **N+1 queries are the #1 ORM performance problem.** Use eager loading.
4. **Drop to raw SQL for:**
    - Complex aggregations (GROUP BY + HAVING)
    - Window functions
    - CTEs
    - Performance-critical queries
5. **Profile generated SQL with EXPLAIN ANALYZE.** Don't trust the abstraction.
6. **Hybrid approach:** ORM for writes, SQL for reads.
7. **Always parameterize raw SQL** to prevent SQL injection.
8. **Use database views** to bridge ORM and complex SQL.
9. **Don't over-include.** Fetch only what you need.
10. **The goal isn't to avoid ORMs—it's to use them consciously.**

Next (final): Interview questions and real-world query challenges.