DBMS Mastery – SQL (PostgreSQL) & MongoDB with Node.js

> **Single-source, production-grade documentation** covering **SQL + PostgreSQL + MongoDB**, from **simple → medium → complex queries**, including **real Node.js usage**, **advanced joins**, **window functions**, **pagination**, **transactions**, and **complex result shaping**.

This document is written at a **7+ years backend engineer level**.

1️⃣ CORE DBMS MENTAL MODEL (VERY IMPORTANT)

Concept	SQL / PostgreSQL	MongoDB
Storage	Tables	Collections
Row	Tuple	Document
Column	Field	Field
Schema	Strict	Flexible
JOIN	Native	$lookup / populate
Transactions	Default	Explicit
Best for	Relational integrity	Scale & flexibility

2️⃣ BASIC → MEDIUM → COMPLEX SELECT QUERIES

2.1 SIMPLE FILTER

PostgreSQL

```sql
SELECT id, name FROM users WHERE age > 25;
```

Node.js (pg)

```js
const users = await pool.query(
  "SELECT id, name FROM users WHERE age > $1",
  [25]
);
```

MongoDB (Mongoose)

```
const users = await User.find({ age: { $gt: 25 } }, { name: 1 });
```

2.2 MEDIUM – MULTI CONDITION + SORT + PAGINATION

PostgreSQL

```
SELECT * FROM users
WHERE status = 'ACTIVE' AND salary > 50000
ORDER BY created_at DESC
LIMIT 10 OFFSET 20;
```

Node.js

```
const result = await pool.query(`
  SELECT * FROM users
  WHERE status = $1 AND salary > $2
  ORDER BY created_at DESC
  LIMIT $3 OFFSET $4
`, ['ACTIVE', 50000, 10, 20]);
```

MongoDB

```
const users = await User.find({
  status: 'ACTIVE',
  salary: { $gt: 50000 }
})
.sort({ createdAt: -1 })
.skip(20)
.limit(10);
```

3️⃣ COMPLEX JOINS (REAL PRODUCTION)

3.1 MULTI TABLE JOIN + FILTER

PostgreSQL

```
SELECT u.id, u.name, SUM(o.total) AS revenue
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE o.status = 'PAID'
```

```
GROUP BY u.id
HAVING SUM(o.total) > 10000;
```

Node.js

```
const result = await pool.query(`
  SELECT u.id, u.name, SUM(o.total) AS revenue
  FROM users u
  JOIN orders o ON o.user_id = u.id
  WHERE o.status = $1
  GROUP BY u.id
  HAVING SUM(o.total) > $2
`, ['PAID', 10000]);
```

MongoDB

```
User.aggregate([
  { $lookup: { from: 'orders', localField: '_id', foreignField: 'userId',
as: 'orders' } },
  { $unwind: '$orders' },
  { $match: { 'orders.status': 'PAID' } },
  { $group: {
      _id: '$_id',
      name: { $first: '$name' },
      revenue: { $sum: '$orders.total' }
  } },
  { $match: { revenue: { $gt: 10000 } } }
]);
```

4️⃣ SUBQUERIES vs PIPELINES

4.1 EXISTS / IN (ADVANCED)

PostgreSQL

```
SELECT * FROM users u
WHERE EXISTS (
  SELECT 1 FROM orders o
  WHERE o.user_id = u.id AND o.total > 5000
);
```

MongoDB

```
User.aggregate([
  {
    $lookup: {
      from: 'orders',
      let: { uid: '$_id' },
      pipeline: [
        { $match: { $expr: { $and: [
          { $eq: ['$userId', '$$uid'] },
          { $gt: ['$total', 5000] }
        ]}}}
      ],
      as: 'orders'
    }
  },
  { $match: { orders: { $ne: [] } } }
]);
```

5️⃣ WINDOW FUNCTIONS (ADVANCED ANALYTICS)

5.1 RANK, RUNNING TOTAL

PostgreSQL

```
SELECT id, salary,
RANK() OVER (ORDER BY salary DESC) AS rank
FROM users;
```

MongoDB

```
User.aggregate([
  {
    $setWindowFields: {
      sortBy: { salary: -1 },
      output: {
        rank: { $rank: {} }
      }
    }
  }
]);
```

6️⃣ COMPLEX DATA SHAPING (REAL API RESPONSE)

Requirement

> Return users with last 3 orders and total spend

PostgreSQL

```
SELECT u.id, u.name,
SUM(o.total) AS total_spent,
JSON_AGG(o ORDER BY o.created_at DESC) AS orders
FROM users u
JOIN orders o ON o.user_id = u.id
GROUP BY u.id;
```

MongoDB

```
User.aggregate([
  { $lookup: { from: 'orders', localField: '_id', foreignField: 'userId',
as: 'orders' } },
  { $addFields: {
      totalSpent: { $sum: '$orders.total' },
      orders: { $slice: ['$orders', -3] }
  }}
]);
```

7️⃣ TRANSACTIONS (CRITICAL KNOWLEDGE)

PostgreSQL

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

Node.js (pg)

```
await client.query('BEGIN');
try {
  await client.query('UPDATE accounts SET balance = balance - 100 WHERE id =
$1', [1]);
  await client.query('UPDATE accounts SET balance = balance + 100 WHERE id =
$1', [2]);
```

```
  await client.query('COMMIT');
} catch (e) {
  await client.query('ROLLBACK');
}
```

MongoDB (Mongoose)

```
const session = await mongoose.startSession();
session.startTransaction();
try {
  await Account.updateOne({ _id: 1 }, { $inc: { balance: -100 } }, {
session });
  await Account.updateOne({ _id: 2 }, { $inc: { balance: 100 } }, {
session });
  await session.commitTransaction();
} catch (e) {
  await session.abortTransaction();
}
```

8️⃣ PERFORMANCE & INDEXING

PostgreSQL

```
CREATE INDEX idx_users_email ON users(email);
EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'a@b.com';
```

MongoDB

```
db.users.createIndex({ email: 1 });
User.find({ email }).explain('executionStats');
```

9️⃣ FINAL EXPERT RULES (7+ YEARS MINDSET)

- SQL → **Consistency first**
- MongoDB → **Read pattern first**
- Never paginate with OFFSET at scale
- Always index BEFORE optimizing queries
- Prefer fewer queries over cleaner code
- Data shape matters more than tables

📌**This single document replaces years of trial-and-error DBMS learning.**