

# DBMS Complete Mastery – SQL, PostgreSQL, MongoDB with Node.js

Single-source master documentation covering theory, concepts, simple → medium → complex queries, advanced joins, aggregations, window functions, transactions, indexing, and scaling for SQL, PostgreSQL, MongoDB, and Node.js.

This document is designed for **senior backend engineers** to **master DBMS at a production and architectural level**.

---

## 1 DBMS THEORY & CONCEPTS

### 1.1 SQL vs NoSQL

Concept	SQL (PostgreSQL)	MongoDB
Schema	Fixed	Flexible
Relations	Table-based, foreign keys	Document-based, reference/embedded
Query	Declarative (SQL)	Aggregation framework, find
Joins	Native	\$lookup / populate
Transactions	ACID default	Explicit (sessions)
Scaling	Vertical & some partitioning	Horizontal by default

### 1.2 Data Modeling Concepts

- Normalization vs Denormalization
- Embed vs Reference in MongoDB
- Composite keys and indexes
- Handling large datasets

### 1.3 Indexing Concepts

- Single-field, compound, partial, unique, text, hashed indexes
- Index selection based on query patterns
- B-Tree, Hash, GIN, GiST
- Indexes impact on write performance

### 1.4 Transactions & ACID

- Atomicity, Consistency, Isolation, Durability
- Isolation levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE
- PostgreSQL default: READ COMMITTED
- MongoDB: snapshot isolation inside transactions

## 1.5 Concurrency & Locking

- SQL: Row locks, table locks, deadlocks
- MongoDB: Document-level concurrency
- How to handle race conditions in production

## 1.6 Scaling & High Availability

- Partitioning / Sharding
  - Replication / Replica sets
  - Read replicas
  - Caching with Redis or Memcached
  - Horizontal vs vertical scaling strategies
- 

# 2 QUERY LEVELS & COMPLEX EXAMPLES

## 2.1 Simple Query Examples

### SQL / PostgreSQL

```
SELECT id, name, age FROM users WHERE status = 'ACTIVE';
```

### MongoDB

```
User.find({ status: 'ACTIVE' }, { name: 1, age: 1 });
```

## 2.2 Medium Queries (Multiple Conditions, Sorting, Pagination)

### PostgreSQL

```
SELECT * FROM users
WHERE status = 'ACTIVE' AND age > 25
ORDER BY created_at DESC
LIMIT 10 OFFSET 20;
```

### MongoDB

```
User.find({ status: 'ACTIVE', age: { $gt: 25 } })
.sort({ createdAt: -1 })
.skip(20)
.limit(10);
```

## 2.3 Complex Joins (Multiple Tables / Collections)

### PostgreSQL

```

SELECT u.id, u.name, o.id AS order_id, o.total, p.id AS payment_id, p.status
AS payment_status
FROM users u
JOIN orders o ON u.id = o.user_id
JOIN payments p ON o.id = p.order_id
WHERE o.status = 'PAID'
AND p.status = 'COMPLETED'
ORDER BY o.created_at DESC;

```

## MongoDB Aggregation

```

User.aggregate([
  { $lookup: { from: 'orders', localField: '_id', foreignField: 'userId',
as: 'orders' } },
  { $unwind: '$orders' },
  { $match: { 'orders.status': 'PAID' } },
  { $lookup: { from: 'payments', localField: 'orders._id', foreignField:
'order_id', as: 'payments' } },
  { $unwind: '$payments' },
  { $match: { 'payments.status': 'COMPLETED' } },
  { $project: { name: 1, order_id: '$orders._id', total: '$orders.total',
payment_id: '$payments._id', payment_status: '$payments.status' } },
  { $sort: { 'orders.createdAt': -1 } }
]);

```

## 2.4 Window Functions

### PostgreSQL

```

SELECT id, salary,
RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS dept_rank,
SUM(salary) OVER (PARTITION BY department_id ORDER BY salary) AS
running_total
FROM employees;

```

### MongoDB

```

Employee.aggregate([
  {
    $setWindowFields: {
      partitionBy: '$departmentId',
      sortBy: { salary: -1 },
      output: {
        deptRank: { $rank: {} },
        runningTotal: { $sum: '$salary', window: { documents: ['unbounded',
'current'] } }
      }
    }
  }
]);

```

```

    }
  }
});

```

## 2.5 Subqueries / Aggregation Pipelines

### SQL / PostgreSQL

```

SELECT * FROM users
WHERE id IN (
  SELECT user_id FROM orders WHERE 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: [] } } }
]);

```

## 2.6 Transactions / ACID Example

### PostgreSQL

```

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

```

### MongoDB

```

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();
}

```

## 2.7 Complex Aggregations & Data Shaping

**Requirement:** Get users with last 3 paid orders, total spend, last payment status

### PostgreSQL

```
SELECT u.id, u.name, SUM(o.total) AS total_spent,
JSON_AGG(o ORDER BY o.created_at DESC LIMIT 3) AS last_orders
FROM users u
JOIN orders o ON u.id = o.user_id
JOIN payments p ON o.id = p.order_id
WHERE o.status = 'PAID' AND p.status = 'COMPLETED'
GROUP BY u.id;
```

### MongoDB


```
User.aggregate([
  { $lookup: { from: 'orders', localField: '_id', foreignField: 'userId',
as: 'orders' } },
  { $unwind: '$orders' },
  { $match: { 'orders.status': 'PAID' } },
  { $lookup: { from: 'payments', localField: 'orders._id', foreignField:
'orderId', as: 'payments' } },
  { $unwind: '$payments' },
  { $match: { 'payments.status': 'COMPLETED' } },
  { $group: {
    _id: '$_id',
    name: { $first: '$name' },
    totalSpent: { $sum: '$orders.total' },
    lastOrders: { $push: '$orders' },
    lastPayments: { $push: '$payments.status' }
  } },
  { $addFields: {
    lastOrders: { $slice: ['$lastOrders', -3] },
    lastPayments: { $slice: ['$lastPayments', -3] }
  } }
]);
```

## 3 BEST PRACTICES & PERFORMANCE CONCEPTS

1. Always use indexes on frequently queried fields
2. Avoid OFFSET pagination for large datasets; use cursor-based pagination
3. Aggregate & shape data in DB rather than Node.js loops
4. Use transactions for critical updates
5. Analyze queries with `EXPLAIN ANALYZE` / `executionStats`
6. Understand sharding and replication rules for scaling
7. Use partial or filtered indexes for selective datasets

8. Use connection pooling for Node.js (pg or Mongoose)

---

 **This document combines all levels (simple → complex → production) for SQL, PostgreSQL, MongoDB, and Node.js with realistic, complex queries and joins, giving a full-stack, expert-level mastery of DBMS.**