Day2_Data Modeling in NoSQL_ Lab

- Document Design Principles
- Embedded vs. Referenced Documents
- Denormalization vs. Normalization trade-offs
- Schema Design for:
    - Customer Profiles
    - Account Transactions
    - Product Catalogs
- Best Practices for Scalability & Indexing

You will:

1. Create DB & base collections
2. Practice **Document Design Principles**
3. Compare **Embedded vs Referenced**
4. See **Denormalization vs Normalization** in action
5. Implement **schema** for:
    - customers
    - accounts + transactions
    - products (product catalog)
6. Add **indexes for scalability** and test with explain()

Use mongosh or MongoDB Compass → **Shell tab**.

🔰 Step 0 – Start & Select Database

```
mongosh
use dbs_tech_bank;
```

What's happening?

You are working in a database called dbs_tech_bank. MongoDB will create it when we first insert data.

Step 1 – Create Base Collections

```
db.createCollection("customers");
db.createCollection("accounts");
db.createCollection("transactions");
db.createCollection("products");
db.createCollection("customerAddresses");  // for referenced demo
```

Why?

You're defining the core collections needed for Customer360 + product catalog, plus one extra (customerAddresses) to demo **referenced documents**.

Step 2 – Document Design & Embedded vs Referenced

2.1 Insert Customer with

Embedded Address

(good for 1–few)

```
db.customers.insertOne({
  custId: "C1001",
  fullName: "Rohit Sharma",
  email: "rohit.sharma@example.com",
  mobile: "9876543210",
  kycStatus: "VERIFIED",
  riskRating: "LOW",
  addresses: [
    {
      type: "home",
      line1: "Pune Nagar Road",
      city: "Pune",
      state: "MH",
      pin: "411014",
      country: "IN"
    },
    {
      type: "office",
      line1: "BKC",
      city: "Mumbai",
      state: "MH",
      pin: "400051",
      country: "IN"
    }
  ],
  createdAt: ISODate("2024-01-10T10:00:00Z"),
  updatedAt: ISODate("2025-01-10T10:00:00Z")
});
```

Why this step?

Shows **embedded documents**: addresses is an array *inside* the customer → perfect for 1–few, always-read-together data.

2.2 Query Embedded Address

```
db.customers.find(
```

```
  { custId: "C1001" },
  { _id: 0, custId: 1, fullName: 1, "addresses.city": 1 }
).pretty();
```

Why?

You're seeing how nested fields like "addresses.city" are easily queried.

2.3 Create Referenced Addresses (separate collection)

Now you'll model addresses as a **separate collection** linked by custId.

```
db.customerAddresses.insertMany([
  {
    custId: "C1001",
    type: "home",
    line1: "Pune Nagar Road",
    city: "Pune",
    state: "MH",
    pin: "411014",
    country: "IN"
  },
  {
    custId: "C1001",
    type: "office",
    line1: "BKC",
    city: "Mumbai",
    state: "MH",
    pin: "400051",
    country: "IN"
  }
]);
```

Why?

This demonstrates **referenced documents**: data is split for flexibility or very large sets.

2.4 Join-like View with

$lookup

(Embedded vs Referenced Feel)

```
db.customers.aggregate([
  { $match: { custId: "C1001" } },
  {
    $lookup: {
      from: "customerAddresses",
      localField: "custId",
```

```
      foreignField: "custId",
      as: "refAddresses"
    }
  },
  {
    $project: {
      _id: 0,
      custId: 1,
      fullName: 1,
      embeddedAddresses: "$addresses",
      referencedAddresses: "$refAddresses"
    }
  }
]).pretty();
```

Why?

You directly compare **embedded vs referenced** addresses in one output.

3️⃣ Step 3 – Normalization vs Denormalization (Hands-on)

We'll use customers + transactions to show the trade-off.

3.1 Insert Customers (normalized side)

```
db.customers.insertMany([
  {
    custId: "C1002",
    fullName: "Anita Desai",
    email: "anita.desai@example.com",
    mobile: "9998887776",
    kycStatus: "VERIFIED",
    riskRating: "MEDIUM",
    addresses: []
  },
  {
    custId: "C1003",
    fullName: "Vikram Iyer",
    email: "vikram.iyer@example.com",
    mobile: "9898989898",
    kycStatus: "PENDING",
    riskRating: "HIGH",
    addresses: []
  }
]);
```

3.2 Insert

Normalized Transactions

(no duplicated names)

```
db.transactions.insertMany([
  {
    txnId: "T9001",
    accNo: "SAV1001",
    custId: "C1001",
    type: "DEBIT",
    amount: 2500,
    channel: "UPI",
    timestamp: ISODate("2025-01-10T10:15:00Z"),
    status: "POSTED"
  },
  {
    txnId: "T9002",
    accNo: "SAV1002",
    custId: "C1002",
    type: "DEBIT",
    amount: 75000,
    channel: "NETBANKING",
    timestamp: ISODate("2025-01-10T11:15:00Z"),
    status: "POSTED"
  },
  {
    txnId: "T9003",
    accNo: "CURR1003",
    custId: "C1003",
    type: "DEBIT",
    amount: 500000,
    channel: "NETBANKING",
    timestamp: ISODate("2025-01-10T12:30:00Z"),
    status: "POSTED"
  }
]);
```

3.3 Customer Name via

Join

(Normalized)

```
db.transactions.aggregate([
  {
    $match: { amount: { $gt: 20000 } }
  },
  {
    $lookup: {
      from: "customers",
      localField: "custId",
      foreignField: "custId",
      as: "customer"
    }
  },
  { $unwind: "$customer" },
  {
    $project: {
      _id: 0,
      txnId: 1,
      custId: 1,
      customerName: "$customer.fullName",
```

```
      amount: 1,
      channel: 1
    }
  }
]).pretty();
```

What you just did:

You computed a **report** using a **normalized** schema (no duplicate names, but needs $lookup).

3.4 Add Denormalized Field to Transactions

Now add customerName directly into transactions.

```
db.transactions.updateMany(
  { custId: "C1001" },
  { $set: { customerName: "Rohit Sharma" } }
);

db.transactions.updateMany(
  { custId: "C1002" },
  { $set: { customerName: "Anita Desai" } }
);

db.transactions.updateMany(
  { custId: "C1003" },
  { $set: { customerName: "Vikram Iyer" } }
);
```

3.5 Same Report Without $lookup (Denormalized)

```
db.transactions.find(
  { amount: { $gt: 20000 } },
  { _id: 0, txnId: 1, custId: 1, customerName: 1, amount: 1, channel: 1 }
).pretty();
```

Observation (important hands-on point):

- With denormalization, query is **simpler** and often **faster**.

- But if a customer changes their name, you must update **multiple docs**.

4️⃣ Step 4 – Schema Design Hands-on

4.1 Insert Accounts

```
db.accounts.insertMany([
  {
    accNo: "SAV1001",
    custId: "C1001",
    productCode: "SAV_STD",
    type: "SAVINGS",
    currency: "INR",
    branchCode: "BR_PUNE_01",
    status: "ACTIVE",
    balance: 152340.75,
    openedAt: ISODate("2023-10-10T09:00:00Z")
  },
  {
    accNo: "SAV1002",
    custId: "C1002",
    productCode: "SAV_STD",
    type: "SAVINGS",
    currency: "INR",
    branchCode: "BR_MUM_01",
    status: "ACTIVE",
    balance: 50200.00,
    openedAt: ISODate("2024-03-15T09:00:00Z")
  },
  {
    accNo: "CURR1003",
    custId: "C1003",
    productCode: "CURR_BIZ",
    type: "CURRENT",
    currency: "INR",
    branchCode: "BR_BLR_01",
    status: "ACTIVE",
    balance: 983450.00,
    openedAt: ISODate("2022-05-01T09:00:00Z")
  }
]);
```

4.2 Insert Products (Product Catalog)

```
db.products.insertMany([
  {
    productCode: "SAV_STD",
    category: "DEPOSIT",
    type: "SAVINGS",
    name: "DBS Standard Savings Account",
    description: "Regular savings account for retail customers.",
    currency: "INR",
    segment: "RETAIL",
    features: {
```

```
      minBalance: 0,
      interestRate: 3.5,
      atmFreeTxnsPerMonth: 5
    },
    status: "ACTIVE",
    validFrom: ISODate("2024-01-01T00:00:00Z"),
    validTo: null
  },
  {
    productCode: "CURR_BIZ",
    category: "DEPOSIT",
    type: "CURRENT",
    name: "DBS Business Current Account",
    description: "Current account for SMEs.",
    currency: "INR",
    segment: "SME",
    features: {
      minBalance: 50000,
      overdraftAvailable: true
    },
    status: "ACTIVE",
    validFrom: ISODate("2024-01-01T00:00:00Z"),
    validTo: null
  }
]);
```

4.3 Quick Query: Show Accounts + Product Info (Customer View)

```
db.accounts.aggregate([
  { $match: { custId: "C1001" } },
  {
    $lookup: {
      from: "products",
      localField: "productCode",
      foreignField: "productCode",
      as: "product"
    }
  },
  { $unwind: "$product" },
  {
    $project: {
      _id: 0,
      accNo: 1,
      type: 1,
      balance: 1,
      "product.name": 1,
      "product.segment": 1
    }
  }
]).pretty();
```

Why?

You're linking **accounts → product catalog**, giving a richer Customer360.

Step 5 – Scalability & Indexing (Hands-on)

5.1 Create Indexes for Customers

```
db.customers.createIndex({ custId: 1 }, { unique: true });
db.customers.createIndex({ mobile: 1 }, { unique: true });
db.customers.createIndex({ email: 1 }, { unique: true });
db.customers.createIndex({ kycStatus: 1, riskRating: 1 });
```

Try this query:

```
db.customers.find({ mobile: "9876543210" }).explain("executionStats");
```

Observe:

You should see "IXSCAN" (index scan) in the plan if index is used.

5.2 Indexes for Accounts

```
db.accounts.createIndex({ accNo: 1 }, { unique: true });
db.accounts.createIndex({ custId: 1, status: 1 });
```

Try:

```
db.accounts.find({ custId: "C1001", status: "ACTIVE"
}).explain("executionStats");
```

5.3 Indexes for Transactions

```
// Statement queries: by account + time
db.transactions.createIndex({ accNo: 1, timestamp: -1 });

// Customer analytics
db.transactions.createIndex({ custId: 1, timestamp: -1 });

// Channel analytics
db.transactions.createIndex({ channel: 1, timestamp: -1 });
```

Test:

```
db.transactions.find(
  { accNo: "SAV1001" }
).sort({ timestamp: -1 }).limit(5).explain("executionStats");
```

What you're seeing:

- If index is used → faster **sorted** queries.
- This links the **indexing** concept directly to **query patterns**.

5.4 Index for Product Catalog

```
db.products.createIndex({ productCode: 1 }, { unique: true });
db.products.createIndex({ category: 1, segment: 1, status: 1 });
```

Try:

```
db.products.find(
  { category: "DEPOSIT", segment: "RETAIL", status: "ACTIVE" }
).explain("executionStats");
```
