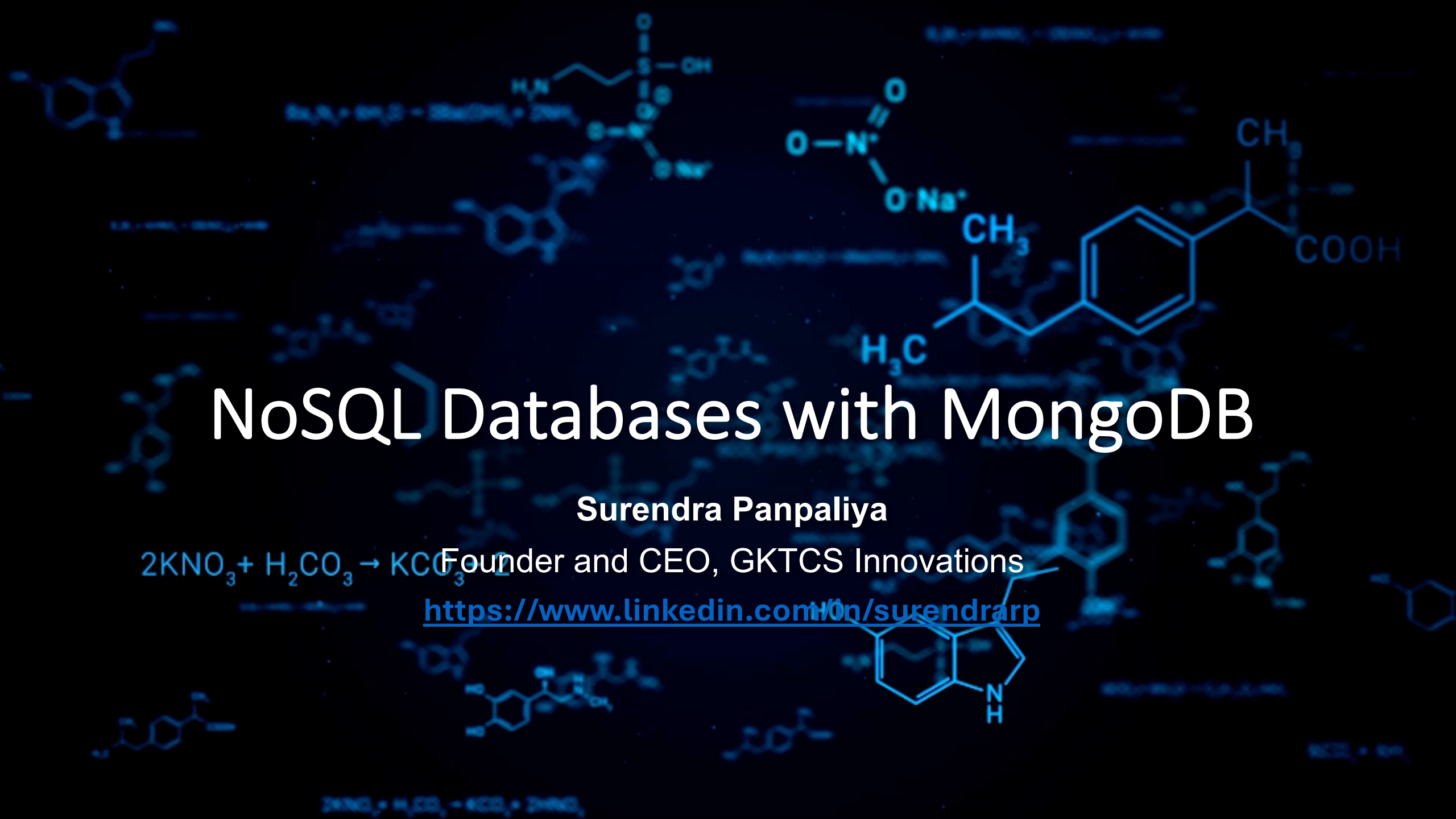


NoSQL Databases with MongoDB

Surendra Panpaliya

Founder and CEO, GKTCS Innovations

https://www.linkedin.com/in/surendrarp

Surendra Panpaliya, DTM

Founder & CEO, GKTCS Innovations – building future-ready enterprises.

Empowered 35,000+ IT professionals through training, mentoring, and consulting.

Partnered with 300+ multinational corporations to accelerate business growth.

🎯 Learning Objectives

- Understand NoSQL fundamentals and
- how they differ from RDBMS.
- Model data efficiently for NoSQL systems.
- Perform CRUD operations, aggregations, and
- indexing in MongoDB.

🎯 Learning Objectives

Apply NoSQL patterns to real-world banking scenarios

e.g., Customer360, Transaction Logging, Fraud Detection

Build and query NoSQL databases

using MongoDB tools.

Prerequisites

- Basic understanding of **Databases & SQL** concepts
- tables, joins, normalization
- Familiarity with **Python / JavaScript** syntax
- for data access (optional but helpful).
- Prior exposure to **JSON** and **REST APIs** recommended.

Lab Setup Requirements

System Requirements:

Laptop Configuration:

8 GB RAM minimum, 20 GB free disk space

Operating System:

Windows 10+, macOS, or Ubuntu

Lab Setup Requirements

Software Installation:

MongoDB Community Edition (latest version)

MongoDB Compass – GUI for query visualization

VS Code or **Jupyter Notebook** for scripting

Python 3.11+ with pymongo library installed

Agenda

Day1

Introduction to NoSQL & MongoDB Fundamentals

Day2

Data Modeling, Aggregation, and Use Cases

Module 1: NoSQL Overview

Evolution of Databases:

RDBMS → NoSQL

Key Characteristics:

Scalability, Flexibility,

Schema-less Design

Module 1: NoSQL Overview

Types of NoSQL Databases:

Key-Value Stores (Redis)

Document Stores (MongoDB)

Column Stores (Cassandra)

Graph Databases (Neo4j)

Module 1: NoSQL Overview

Relational vs. NoSQL

Comparison table

Architecture diagram

Module 2: MongoDB Essentials

MongoDB Architecture:

Database → Collection → Document

BSON & JSON structure

CRUD Operations:

insertOne(), find(),

updateOne(), deleteOne()

Module 2: MongoDB Essentials

Indexing & Query Optimization

Data Import/Export using

mongoimport, mongoexport

Module 3: Hands-On Lab

Lab 1: Create and Query a MongoDB Customer Database

Create a customers collection

Insert 10 customer profiles (JSON documents)

Query customers using filters (find(), $and, $or)

Update address & contact info using update operators

Day 2: Data Modeling, Aggregation, and Use Cases

Surendra Panpaliya

Global IT Trainer | Consultant | Thought Leader

Founder & CEO, GKTCS Innovations | DTM

Module 4: Data Modeling in NoSQL

- Document Design Principles
- Embedded vs. Referenced Documents
- Denormalization vs. Normalization trade-offs

Module 4: Data Modeling in NoSQL

Schema Design for:

Customer Profiles

Account Transactions

Product Catalogs

Best Practices for
Scalability & Indexing

Module 5: Querying & Aggregations

Aggregation Pipeline Basics

$match, $group, $project, $sort, $limit

Filtering and transforming data

Case Study: Summarize total transactions per customer

Module 6: Banking Use Cases & Labs

Lab 2: Implement Account Transaction Logging using MongoDB

Create a transactions collection

Insert transactions with timestamps, account numbers, and amounts

Query high-value transactions ($gt, $sum)

Build a fraud-detection query pipeline

Module 7: Capstone Assignment & Discussion

Design a Mini Customer360 Database for DBS Tech Bank

Collections: customers, accounts, transactions, alerts

Define key fields, indexing strategy, and relationships

Submit ER diagram (document-based) and sample queries

Module 4: Data Modeling in NoSQL

- Document Design Principles
- Embedded vs. Referenced Documents
- Denormalization vs. Normalization trade-offs

What Is Data Modeling in NoSQL?

Designing how your **data is stored**, **connected**, and **retrieved** using:

Documents (instead of rows)

Collections (instead of tables)

Embedded data (nested JSON)

References (link documents)

What Is Data Modeling in NoSQL?

NoSQL is **schema-flexible**,

So design focuses on:

Performance

Query patterns

Real-world objects (Customer, Account, Transactions)

Core Document Design Principles (MongoDB)

Principle 1: Model Data Around Queries (Not Just Structure)

Design documents

based on

how your app queries data.

Example Query in DBS Bank

Fetch Customer360

customer + all accounts + balance + last 5 transactions

Embed account summary inside customer document:

```
{
  "custId": "C1001",
  "name": "Rohit Sharma",
  "accounts": [
    { "accNo": "SAV1001", "type": "Savings", "balance": 150000 },
    { "accNo": "CRD2001", "type": "Credit Card", "limit": 100000 }
  ]
}
```

Why?

Only **1 query** to get full customer view.

Faster response for dashboards.

Principle 2: Embed When Data Has 1–N Relationship & Low Growth

- Use embedded documents when child data is:
- Small
- Frequently read with parent
- Not updated independently

Example: Customer → Address

A customer may have 1–2 addresses.


```
{
  "custId": "C1001",
  "name": "Rohit Sharma",
  "address": {
    "line1": "Pune Nagar Road",
    "city": "Pune",
    "pin": "411014"
  }
}
```

Why?

SIMPLE

FAST

ALL CUSTOMER INFO IN ONE DOCUMENT

❌ When NOT to Embed

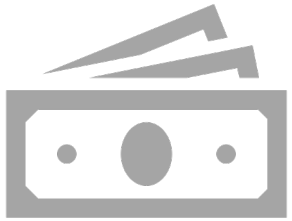
If child grows indefinitely

If many updates

If too large (MongoDB 16MB limit)

✕ When NOT to Embed

Example: **transactions**



A customer may have **millions of transactions**

DO NOT embed.

Principle 3: Reference When Data Grows Large (1–Many, Many–Many)

Example:

Customer → Transactions

Store transactions in a separate collection:

customers collection


```
{
  "custId": "C1001",
  "name": "Rohit Sharma"
}
```

transactions collection


```
{
  "txnId": "T9001",
  "custId": "C1001",
  "type": "DEBIT",
  "amount": 2000,
  "timestamp": "2025-01-01T10:00:00Z"
}
```

How to fetch?

db.transactions.find({ custId: "C1001" })

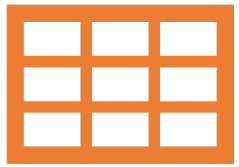
Efficient

No large document issue

Transactions scalable (billions)

Principle 4: Use Denormalization for Speed

MongoDB prefers



duplicate data

to speed up reads.

Example

Store customer name in the transaction document.


```
{
  "txnId": "T9001",
  "custId": "C1001",
  "customerName": "Rohit Sharma",
  "amount": 2000
}
```

Why?

Faster reporting

No join needed

Slight duplication acceptable in NoSQL

Principle 5: Use Aggregation Pipelines Instead of Joins

MongoDB aggregates are powerful.

Example: Customer + last transaction

```
db.transactions.aggregate([
  { $match: { custId: "C1001" } },
  { $sort: { timestamp: -1 } },
  { $limit: 1 }
])
```

Principle 6: Avoid Deep Nesting (> 3 Levels)

Bad (too deep):

```
{
  "customer": {
    "accounts": [
      {
        "transactions": [
          { "notes": { "audit": { "updatedBy": "admin" }}}
        ]
      }
    ]
  }
}
```

Principle 6: Avoid Deep Nesting (> 3 Levels)

Good:

Keep customer, accounts, transactions separate

Link using custId, accNo

Principle 7: Use a Consistent Naming Pattern

Prefer

custId, accNo, txnId

lowercase + camelCase

Principle 7: Use a Consistent Naming Pattern

Avoid:

CUSTID, Cust_ID, CUST-id

This keeps documents clean and consistent.

Recommended MongoDB Model for DBS

customers

```
{
  "custId": "C1001",
  "name": "Rohit Sharma",
  "mobile": "9876543210",
  "addresses": [
    { "type": "home", "city": "Pune", "pin": "411014" }
  ]
}
```

accounts


```
{
  "accNo": "SAV1001",
  "custId": "C1001",
  "type": "Savings",
  "balance": 150000
}
```

transactions


```
{
  "txnId": "T9001",
  "accNo": "SAV1001",
  "custId": "C1001",
  "type": "DEBIT",
  "amount": 2000,
  "timestamp": "2025-01-01T10:00:00Z"
}
```

How These Principles Help DBS Bank?

Requirement	Solution
Customer360	Embed account summary → fast queries
Millions of transactions	Reference in separate collection
Reporting dashboards	Denormalized fields
Audit & compliance	Aggregation pipelines
Scalability	No joins, simple relationships

1) Document Design Principles

Work backwards from access patterns

List top **reads**, **writes**, **filters**, **sorts**, and **aggregations**.

Co-locate fields that are **read together** → same document.

Use **compound indexes** that match your query prefixes.

Model by cardinality & growth

One-to-few:

embed (e.g., addresses in customer).

One-to-many (bounded):

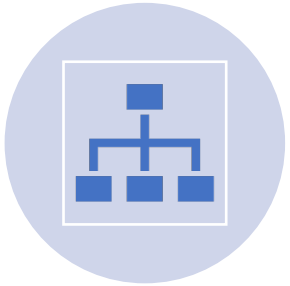
embed or reference with subset embedding.

Model by cardinality & growth

One-to-many (unbounded/high-churn):

reference or **bucket**.

Watch **unbounded arrays**



(can cause bloated docs & update contention).

Operational constraints

16 MB **document size** limit.

Prefer **immutable event docs** (append-only transactions).

Use **Decimal128** for money.

Add **createdAt / updatedAt / version** for audit & forward-compat.

Embedded vs. Referenced Documents

MONGODB GIVES TWO WAYS TO CONNECT DATA:

A. EMBEDDED DOCUMENTS

B. REFERENCED DOCUMENTS

A. Embedded Documents

Store data INSIDE parent document

Use when:

Relationship is **1-to-few**

Data is always needed together

Data size is small

Data rarely changes independently

Example (DBS Bank)

Customer → Address


```
{
  "custId": "C1001",
  "name": "Rohit Sharma",
  "addresses": [
    {
      "type": "home",
      "city": "Pune",
      "pin": "411014"
    },
```

Example (DBS Bank)


```
 "pin": "411014"
    },
    {
      "type": "work",
      "city": "Mumbai",
      "pin": "400093"
    }
  ]
}
```

Why embed?

- Fetching customer profile
- requires **only one query**
- Faster reads for
- Customer360 screen

❌ When NOT to Embed

Do **not** embed when:

Child data grows without limit

Child data is frequently updated

Many users update same sub-document

Document may cross **16MB limit**

B. Referenced Documents

Use a separate collection + reference using ID

Use when:

1-to-many with large children

Data is updated independently

B. Referenced Documents

Use when:

You need scalability

Child collection is huge (millions)

Example (DBS Bank)
Customer → Transactions

customers


```
{
  "custId": "C1001",
  "name": "Rohit Sharma"
}
```

Example (DBS Bank)
Customer → Transactions

transactions


```
{
  "txnId": "T9001",
  "custId": "C1001",
  "amount": 2000,
  "type": "DEBIT"
}
```

Example (DBS Bank)
Customer → Transactions

To fetch transactions:

```
db.transactions.find({ custId: "C1001" })
```

Why reference?

- Transactions grow very fast
- Updating one transaction should not
- rewrite entire customer document
- Better for analytics and compliance

Quick Comparison Table

Feature	Embedded	Referenced
Relationship	1-to-few	1-to-many, many-to-many
Read performance	⭐ Fast	⭐ Medium
Write performance	Medium	Fast for large data
Consistency	Strong	Medium
Document size risk	Yes	No
Recommended for	Customer profile, addresses	Accounts, transactions

Embedded vs. Referenced

Factor	Embed (same doc)	Reference (separate doc + id)	Banking example
Read locality	✅ Excellent	❌ Join at app/2nd query	Customer + top 3 addresses
Write frequency	❌ Hot if subdocs change often	✅ Isolate churn	KYC updates separate from profile
Cardinality	✅ One-to-few	✅ One-to-many/unbounded	Customer → many transactions

Embedded vs. Referenced

Factor	Embed (same doc)	Reference (separate doc + id)	Banking example
Reuse across parents	❌ Hard	✅ Easy	Shared branch data across customers
Size growth	❌ Risk of 16MB	✅ Bounded parent size	Large statements per month
Consistency need	✅ Single-doc atomicity	❌ Multi-doc consistency to manage	Profile + prefs together

2. Denormalization vs. Normalization Trade-offs

MongoDB encourages

denormalization

(duplicate data) but carefully.

A. Normalization (Like RDBMS)

Characteristics:

No duplicate data

Relations through references

Write operations easy

Read operations may require multiple queries

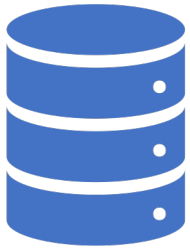
Mongo-style normalized data:

customers

{ "custId": "C1001", "name": "Rohit Sharma" }

- **transactions**
- { "txnId": "T9001", "custId": "C1001", "amount": 2000 }

When to use?



Very large relational datasets

Frequently changing fields

Transaction-heavy systems

B. Denormalization (MongoDB Preferred)

Duplicate some data

to avoid joins and

speed reads.

Example (DBS Bank)

Store **customerName** inside each transaction:

```
{
  "txnId": "T9001",
  "custId": "C1001",
  "customerName": "Rohit Sharma",
  "amount": 2000
}
```

Why denormalize?

- Faster dashboard queries
- No JOIN required
- Simpler and scalable
- Good for analytics

❌ Downsides of Denormalization

Problem	Example
Duplicate data	customerName stored in 10,000 transactions
Update overhead	If customer name changes → update all documents
More storage	Duplicate copies of same fields

Quick Trade-off Summary

Topic	Normalization	Denormalization
Read performance	❌ Slower	⭐ Fastest
Write performance	⭐ Fast	❌ Slower
Data consistency	⭐ Strong	❌ Weaker
Storage	⭐ Efficient	❌ Larger
Use-case	OLTP, updates	Analytics, dashboards

Best Practices for DBS Bank Domain

Use Embedding for

- Addresses
- KYC details
- Preferences
- Branch details (small dataset)

Best Practices for DBS Bank Domain

Use Referencing for:

Transactions

Accounts

Cards

Loans

Audit logs

Alerts/Notifications

Best Practices for DBS Bank Domain

Use Denormalization for:

Storing customerName, branchName, or category inside transactions

Faster Customer360

Faster reporting dashboards

Denormalize (duplicate small, stable fields)

✅ Faster reads (fewer round trips); simpler queries.

❌ Update fan-out (must update duplicates); risk of staleness.

Use when: read-heavy, fields change rarely (e.g., branchName).

Normalize (references)

✅ Single source of truth; smaller docs.

❌ Additional lookups/joins at app side.

Use when: write-heavy or shared/updating entities.

Safe denormalization pattern

Keep an **authoritative id**

branchId and **cache** display fields

(branchName) in the child.

Safe denormalization pattern

Periodically **refresh**

denormalized fields

via batch job or

change streams.

Schema Design Hands On

Customer Profiles

Account Transactions

Product Catalogs

Best Practices for
Scalability & Indexing

Module 5 — Querying & Aggregations

Aggregation Pipeline Basics

What is the Aggregation Pipeline?

- Like a **data processing assembly line**:
- Data flows through **stages**
- Each stage transforms the data
- Output of one stage → input to the next

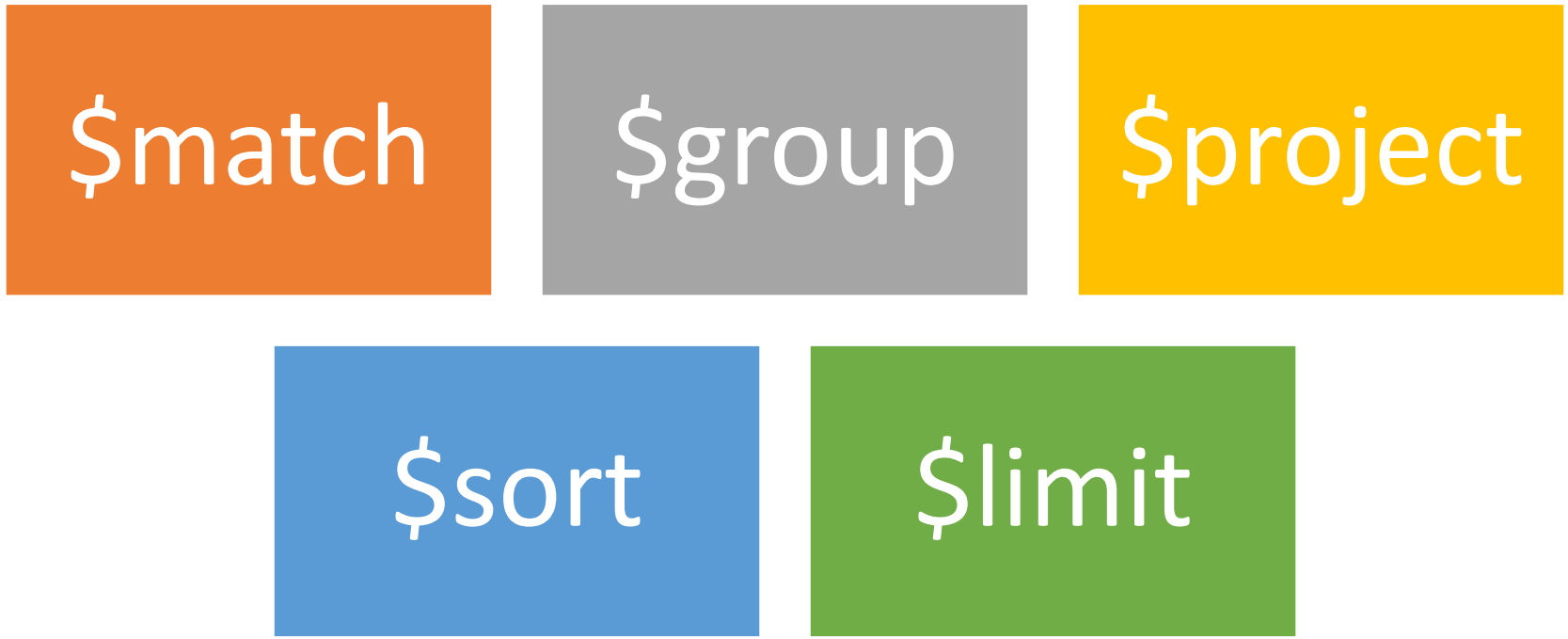
Basic Syntax

```
db.collection.aggregate([
  { /* stage 1 */ },
  { /* stage 2 */ },
  { /* stage 3 */ }
])
```

For DBS Bank, suppose we have a transactions collection:

```
{
  "txnId": "T9001",
  "custId": "C1001",
  "accNo": "SAV1001",
  "type": "DEBIT",
  "amount": 2000,
  "channel": "UPI",
  "txnDate": ISODate("2025-01-02T14:45:00Z")
}
```

Find **total DEBIT amount per customer** and show **top 3** spenders.



Quick Cheat Sheet Table

Stage	Role	Typical Use in DBS Bank
$match	Filter docs	Filter by year, type, channel
$group	Aggregate/group	Total amount per custId/accNo
$project	Shape fields, compute new fields	Rename, hide fields, compute averages
$sort	Order results	Top spenders, latest txns
$limit	Restrict number of rows	Top N customers / txns

Aggregation Pipeline Basics

Mental model

A pipeline is an **ordered list of stages**.

Each stage takes input documents and

outputs transformed documents

to the next stage.

Aggregation Pipeline Basics

Stage	Purpose (bank example)
$match	Filter txns (e.g., last 30 days, DEBIT only)
$project	Shape fields (rename, compute, hide)
$group	Rollups (sum per account/customer)
$sort	Order results (e.g., top spenders)

Aggregation Pipeline Basics

Stage	Purpose (bank example)
$limit	Top-N
$addFields/$set	New fields (e.g., signed amount)
$lookup	Join (e.g., txn → account → customer)
$unwind	Flatten arrays (e.g., multiple addresses)
$facet	Multi-result dashboards in one pass

Let’s Connect

surendra@gktcs.com

https://www.linkedin.com/in/surendrarp

https://www.gktcs.com

Happy Learning !!
Thanks for Your Patience ☺

Surendra Panpaliya

GKTCS Innovations