Day 2 – Indexing, Monitoring, Backup & Atlas (8 Hours)

1. MongoDB Indexes (1 hr)

- Why indexes matter
- Single-field indexes
- Query planner introduction
- Lab: Create and test index performance

---

2. MongoDB Indexes in Detail (1 hr 30 min)

- Compound indexes
- Multikey indexes
- Text indexes
- TTL indexes
- Partial & Sparse indexes
- Index performance monitoring
- Lab: Identify slow queries & fix them with indexes

---

3. MongoDB Logging Basics (1 hr)

- Log file structure
- Slow query logs
- Profiling levels
- Lab: Enable profiler and analyze logs

---

4. Getting Started with MongoDB Atlas (1 hr)

- Create free tier cluster
- Cluster deployment options
- Network & security configuration
- Lab: Deploy and connect to Atlas cluster

---

5. MongoDB Administrator Tools (1 hr)

- mongosh
- mongodump, mongorestore
- mongotop, mongostat
- Compass & Atlas Metrics
- Lab: Use admin tools on a dataset

---

6. Database Metrics & Monitoring (1 hr 30 min)

- Understanding MongoDB metrics
- WiredTiger cache
- IOPS, CPU, memory metrics
- Monitoring tools: Atlas, Cloud Manager, Prometheus, Grafana
- Lab: Live monitoring of workload

End of Day Assignment

- Create indexes and measure performance gain
- Generate and analyze profiler logs

---

1. MongoDB Indexes (1 hr)

- Why indexes matter
- Single-field indexes
- Query planner introduction
- Lab: Create and test index performance

Indexes are *the most critical performance feature* in MongoDB. Good indexing can reduce query time from **seconds** → **milliseconds**.

A. Why Indexes Matter

MongoDB stores documents in collections **without any built-in ordering**.

To search efficiently, indexes are required.

---

1. What is an Index?

An index is a data structure stored separately that allows MongoDB to quickly locate documents.

Think of it like the **index of a book** — instead of scanning every page, you jump directly to the needed section.

---

2. Without Index (Collection Scan – COLLSCAN)

MongoDB checks **every document** in the collection → slow.

```
> COLLSCAN
> nScannedDocs = 50,000
```

---

3. With Index (Index Scan – IXSCAN)

MongoDB uses a B-tree structure → fast.

```
> IXSCAN
> nScannedDocs = 50
```

---

4. Why CSC Needs Indexes

Use cases inside Corporation Service Company (CSC):

✔ Compliance dashboards

Query by jurisdiction, entityName, dueDate

✔ Domain monitoring

Search by domain, expiry, DNS type

✔ Trademark queries

Find by class, country, renewal date

✔ Officer lookup

Query by name, role, country

Indexes ensure:

- Faster search
- Reduced CPU
- Lower latency for APIs
- Better customer experience

---

5. Indexes Reduce:

- CPU load
- Response time
- Memory pressure
- Disk I/O

---

6. Indexes Increase:

- Storage usage
- Write cost (because indexes must be updated on every insert/update)

B. Single-Field Indexes (20 minutes)

Basic syntax:

```
db.collection.createIndex({ field: 1 })     // Ascending
db.collection.createIndex({ field: -1 })    // Descending
```

---

1. Ascending Index Example

```
db.employees.createIndex({ salary: 1 });
```

Useful for:

- Range queries

- Sorting

---

2. Descending Index Example

```
db.entities.createIndex({ lastFiled: -1 });
```

Useful when retrieving:

- Latest filing records
- Latest audit trails

---

3. Unique Index

Prevents duplicate values.

```
db.users.createIndex({ email: 1 }, { unique: true });
```

CSC Use Case:

- Unique corporate entity IDs
- Unique trademark numbers
- Unique domain names

---

4. Partial Index

Index only documents matching a filter.

```
db.logs.createIndex(
  { status: 1 },
  { partialFilterExpression: { status: "ERROR" } }
);
```

CSC Use Case:

- Index only compliance errors
- Index only active legal entities

---

5. Index Options

Option	Meaning
unique	Prevent duplicates
sparse	Only index documents where the field exists
expireAfterSeconds	TTL index for auto-expiry

Option	Meaning
name	Custom index name

Example – TTL Index

Delete audit logs after 30 days:

```
db.auditLogs.createIndex({ ts: 1 }, { expireAfterSeconds: 2592000 });
```

C. Query Planner Introduction (10 minutes)

MongoDB must decide *how* to execute a query.

This is called the **Query Planner**.

---

1. explain() Function

Shows how MongoDB runs a query.

```
db.employees.find({ city: "Pune" }).explain("executionStats");
```

Important fields:

Field	Meaning
COLLSCAN	Slow, no index used
IXSCAN	Index used
nReturned	Number of returned documents
nScannedDocuments	How many scanned before match
executionTimeMillis	Query speed

---

2. Example – Without Index

```
db.customers.find({ city: "Delhi" }).explain("executionStats");
```

Output:

```
stage: "COLLSCAN"
nScannedDocuments: 10000
executionTimeMillis: 52
```

---

3. Example – With Index

```
db.customers.createIndex({ city: 1 });

db.customers.find({ city: "Delhi" }).explain("executionStats");
```

Output:

```
stage: "IXSCAN"
nScannedDocuments: 10
executionTimeMillis: 1
```

Interpretation

- The index drastically improves performance
- Only a few documents scanned
- Planner prefers indexed paths

D. LAB — Create & Test Index Performance (15 minutes)

Goal:

Observe difference between **COLLSCAN** and **IXSCAN**.

Step 1 — Create lab database & insert dataset

```
use csc_index_lab;

for (let i = 1; i <= 30000; i++) {
  db.entities.insertOne({
    entityId: i,
    entityName: "Entity " + i,
    jurisdiction: ["DEL", "NY", "TX", "AMS"][i % 4],
    lastFiled: new Date(2024, (i % 12), (i % 28) + 1)
  });
}
```

Step 2 — Query without index

```
db.entities.find({ jurisdiction: "DEL" }).explain("executionStats");
```

Expected:

```
COLLSCAN
nScannedDocuments: 30000
executionTimeMillis: ~20ms
```

Step 3 — Create Index

```
db.entities.createIndex({ jurisdiction: 1 });
```

Step 4 — Query again

```
db.entities.find({ jurisdiction: "DEL" }).explain("executionStats");
```

Expected:

```
IXSCAN
nScannedDocuments: ~7500
executionTimeMillis: ~2ms
```

Step 5 — Test sort performance

Without index:

```
db.entities.find().sort({ lastFiled: -1 }).explain("executionStats");
```

Create index:

```
db.entities.createIndex({ lastFiled: -1 });
```

Re-run sort:

```
db.entities.find().sort({ lastFiled: -1 }).explain("executionStats");
```

🎯 LAB Outcomes

Students will:

- Understand how indexes reduce scan time
- Compare COLLSCAN vs IXSCAN
- Learn how query planner chooses paths
- Experience index performance effects in real-time

---

2. MongoDB Indexes in Detail (1 hr 30 min)

This module builds on basic indexing and covers:

- Compound Indexes
- Multikey Indexes
- Text Indexes
- TTL Indexes
- Partial & Sparse Indexes
- Index Monitoring & Query Performance

- Hands-on Index Optimization Lab

A. Compound Indexes (20 min)

A **compound index** includes **multiple fields** in a specific order.

✔ Syntax

```
db.collection.createIndex({ field1: 1, field2: -1 })
```

Key Rule: Index ORDER matters

Index: { a: 1, b: 1 }

Can support queries on:

- a
- a + b

Cannot support:

- b alone

⭐ CSC Use Case:

Querying legal entities:

```
db.entities.find({
  jurisdiction: "DEL",
  entityName: /^CSC/
});
```

Optimal index:

```
db.entities.createIndex({ jurisdiction: 1, entityName: 1 });
```

Sort Optimization

If your query uses:

```
sort({ entityName: 1 })
```

AND the filter is:

```
{ jurisdiction: "DEL" }
```

Then compound index:

```
{ jurisdiction: 1, entityName: 1 }
```

removes in-memory sorting → faster.

❗ WRONG INDEX ORDER

Don't create this:

```
{ entityName: 1, jurisdiction: 1 }
```

Because the query filters by jurisdiction first → index won't help.

⭐ Compound Index Prefix Rule

Index { a: 1, b: 1, c: 1 }

Supports prefix queries:

- a
- a + b
- a + b + c

Does NOT support:

- b
- c
- b + c

B. Multikey Indexes (10 min)

Used for **array fields**.

✔ Automatically created when indexing an array

```
db.products.createIndex({ tags: 1 })
```

If document:

```
{ tags: ["electronics", "gaming", "laptop"] }
```

MongoDB indexes **each value**.

⭐ CSC Example

Index officers:

```
db.entities.createIndex({ "officers.role": 1 });
```

Supports:

```
db.entities.find({ "officers.role": "Director" })
```

❗ Limitations:

- Cannot create compound index with more than one array field
- Impacts write performance due to multiple index entries

C. Text Indexes (15 min)

Used for **full-text search**.

✔ Create a Text Index

```
db.products.createIndex({ description: "text" });
```

✔ Search Text

```
db.products.find({ $text: { $search: "gaming laptop" } });
```

⭐ CSC Use Cases

- Trademark description search
- Legal entity notes search
- Domain WHOIS text search

✔ Score Results

Sort by relevance:

```
db.products.find(
```

```
  { $text: { $search: "gaming" } },
  { score: { $meta: "textScore" } }
).sort({ score: -1 });
```

❗ Limitations

- One text index per collection
- Not suitable for large-scale search (use Atlas Search instead)

D. TTL Indexes (10 min)

Automatically deletes documents after a fixed time.

✔ Syntax

```
db.auditLogs.createIndex({ ts: 1 }, { expireAfterSeconds: 86400 });
```

⭐ CSC Use Case:

Delete compliance logs 90 days after creation.

```
db.compliance.createIndex(
  { createdAt: 1 },
  { expireAfterSeconds: 7776000 }
);
```

TTL is perfect for:

- Session expiry
- Temp files
- Cache
- Audit logs

❗ Warnings:

- TTL deletion is not immediate (checked every 60 seconds)
- TTL does NOT work on capped collections
- Field must be a **Date** type

E. Partial & Sparse Indexes (15 min)

---

1. Sparse Index

Indexes only documents where field exists.

```
db.customers.createIndex({ email: 1 }, { sparse: true });
```

Use cases at CSC:

- Entities missing tax IDs
- Optional compliance fields

Sparse index reduces index size dramatically.

---

2. Partial Index

Indexes only documents matching a filter.

```
db.customers.createIndex(
  { premiumScore: 1 },
  { partialFilterExpression: { premiumCustomer: true } }
);
```

CSC Examples:

Index only:

- Active legal entities
- High-priority trademarks
- Expiring domains

Example:

```
db.domains.createIndex(
  { expiry: 1 },
  { partialFilterExpression: { status: "ACTIVE" } }
);
```

F. Index Performance Monitoring (10 min)

⭐ 1. List Indexes

```
db.collection.getIndexes()
```

⭐ 2. Validate Index Usage via explain()

```
db.entities.find({ jurisdiction: "TX" })
  .explain("executionStats")
```

Look for:

- IXSCAN
- nReturned
- nScannedDocuments
- executionTimeMillis

⭐ 3. Check Index Size

```
db.collection.stats().indexSizes
```

⭐ 4. Drop Unused Indexes

```
db.collection.dropIndex("index_name")
```

⭐ 5. Query Profiler

Enable profiler:

```
db.setProfilingLevel(1);
```

Find slow queries:

```
db.system.profile.find().sort({ millis: -1 }).limit(5);
```

⭐ 6. Atlas Performance Tools (If using Atlas)

- Index suggestions
- Query profiler
- Slow query alerts

G. LAB — Identify Slow Queries & Fix Them with Indexes (20 min)

🎯 Goal:

Create a realistic scenario with slow queries → analyze → index → optimize.

Step 1 — Insert 50,000 Test Records

```
use csc_index_advanced;

for (let i = 1; i <= 50000; i++) {
  db.logs.insertOne({
    logId: i,
    user: ["admin", "ops", "api", "system"][i % 4],
    status: ["INFO", "WARN", "ERROR"][i % 3],
    ts: new Date(2024, (i % 12), (i % 28)),
    details: "System event " + i
  });
}
```

Step 2 — Run a Slow Query

```
db.logs.find({ user: "admin", status: "ERROR" })
       .sort({ ts: -1 })
       .limit(5)
       .explain("executionStats");
```

Expected:

```
COLLSCAN
nScannedDocuments: 50000
executionTimeMillis: HIGH
```

Step 3 — Create a Compound Index

Optimal index:

```
db.logs.createIndex({ user: 1, status: 1, ts: -1 });
```

Step 4 — Re-run Query

```
db.logs.find({ user: "admin", status: "ERROR" })
       .sort({ ts: -1 })
       .limit(5)
       .explain("executionStats");
```

Expected:

```
IXSCAN
```

```
nScannedDocuments: VERY LOW
executionTimeMillis: LOW
```

Step 5 — Analyze Improvement

Students compare:

- Before vs after index
- nScannedDocuments
- executionTimeMillis
- Query planner stage

🎯 LAB Outcomes:

Learners will learn:

- How to identify slow queries
- How to choose correct index fields
- How index order impacts performance
- How to validate improvements with explain()
- How to fix real-world performance issues

---

3. MongoDB Logging Basics (1 hr)

- Log file structure
- Slow query logs
- Profiling levels
- Lab: Enable profiler and analyze logs

Logging is a core part of MongoDB database administration.

Admins use logs to identify performance issues, security events, slow queries, replication events, and operational failures.

A. Log File Structure

MongoDB writes logs into **mongod.log**, typically located at:

- Linux: /var/log/mongodb/mongod.log
- Windows: C:\Program Files\MongoDB\Server\log\mongod.log
- Atlas: Integrated logging via UI

---

1. Log Format

A typical log line looks like:

```
2025-01-21T12:45:32.123+0530 I COMMAND  [conn82] command test.users
command: find { filter: { city: "Bangalore" }, limit: 1 }
planSummary: COLLSCAN keysExamined:0 docsExamined:50000
executionTimeMillis: 40
```

Key components:

Field	Meaning
Timestamp	When the event occurred
Severity	I (info), W (warning), E (error), F (fatal)
Component	COMMAND, STORAGE, NETWORK, SHARDING
Connection ID	Session identifier
Event	What operation was executed

---

2. Common Log Types

- **Startup logs** → version, storage engine, config
- **Connection logs** → clients connecting/disconnecting
- **Query execution logs** → slow queries
- **Replication logs** → oplog events
- **Crash & shutdown logs**

---

3. CSC Administrator Use Cases

Logging helps CSC DBAs:

- Detect slow dashboard queries
- Identify long-running compliance data scans
- Trace unauthorized access attempts
- Debug Atlas function/apis

- Monitor replication lag in multi-regional setups

B. Slow Query Logs (10 minutes)

A “slow query” is any query that exceeds the threshold defined by:

```
operationProfiling:
  slowOpThresholdMs: 100
```

Default: **100 milliseconds**

MongoDB automatically logs slow queries into mongod.log.

⭐ Example Slow Query Log

```
I COMMAND  [conn89] query test.entities
filter: { jurisdiction: "TX" }
planSummary: COLLSCAN docsExamined: 50000
keysExamined: 0
executionTimeMillis: 120
```

⭐ What you learn from a slow query log

- Query filter
- Execution plan summary (COLLSCAN or IXSCAN)
- Number of documents scanned
- Execution time
- Index usage

⭐ How to Reduce Slow Queries

- Add appropriate index
- Rewrite inefficient filters
- Avoid regex prefix searches
- Improve projection
- Use compound indexes

C. Profiling Levels (15 minutes)

MongoDB profiler helps capture detailed information about database activity.

⭐ There are 3 profiler levels:

Level	Meaning
0	Off (default). No queries logged.
1	Log **slow** operations only.
2	Log **all** operations (high overhead).

⭐ Enable Profiling Level 1 (Recommended)

Logs operations slower than default slowms (100ms):

```
db.setProfilingLevel(1);
```

⭐ Customize slowms Threshold

```
db.setProfilingLevel(1, { slowms: 50 });
```

⭐ Enable Profiling Level 2 (Debug mode)

```
db.setProfilingLevel(2);
```

⚠️ Warning: High overhead. Use only during debugging.

⭐ Disable Profiling

```
db.setProfilingLevel(0);
```

⭐ View Profiler Output

Profiler stores its logs in:

```
system.profile
```

Use:

```
db.system.profile.find().pretty();
```

⭐ Example Profiling Document

```
{
  "op": "query",
  "ns": "test.entities",
  "command": { "find": "entities", "filter": { "jurisdiction": "NY" }},
  "keysExamined": 0,
  "docsExamined": 32000,
  "millis": 42,
  "planSummary": "COLLSCAN"
}
```

D. LAB — Enable Profiler & Analyze Logs (20 minutes)

Goal: Face a slow query. Enable profiler. Identify issue. Fix using an index.

Step 1 — Insert Test Dataset

```
use logging_lab;

for (let i = 1; i <= 40000; i++) {
  db.logs.insertOne({
    eventId: i,
    user: ["system", "admin", "api"][i % 3],
    status: ["INFO", "WARN", "ERROR"][i % 3],
    ts: new Date(2024, (i % 12), (i % 28)),
    message: "Event log entry " + i
  });
}
```

Step 2 — Enable Profiling

```
db.setProfilingLevel(1, { slowms: 20 });
```

Step 3 — Trigger a Slow Query

```
db.logs.find({ user: "admin", status: "ERROR" })
```

```
        .sort({ ts: -1 })
        .limit(5);
```

Step 4 — Read Profile Logs

```
db.system.profile.find().sort({ millis: -1 }).limit(3).pretty();
```

Look for:

- COLLSCAN
- High millis
- Many docsExamined

Step 5 — Identify Missing Index

Query needs index on:

```
{ user, status, ts }
```

Step 6 — Create Compound Index

```
db.logs.createIndex({ user: 1, status: 1, ts: -1 });
```

Step 7 — Rerun Query & Check Profiling

```
db.logs.find({ user: "admin", status: "ERROR" })
        .sort({ ts: -1 })
        .limit(5)
        .explain("executionStats");
```

Expected:

```
IXSCAN
docsExamined small
executionTimeMillis very low
```

Step 8 — Validate Profiler Again

```
db.system.profile.find().sort({ ts: -1 }).limit(3).pretty();
```

Compare profile before vs after index.

🎯 LAB Outcomes

Students will learn to:

- Enable MongoDB profiler safely
- Analyze slow queries
- Interpret planSummary (COLLSCAN vs IXSCAN)
- Identify missing indexes
- Resolve performance issues
- Validate improvements using profiler

Getting Started with MongoDB Atlas (1 hr)

- Create free tier cluster
- Cluster deployment options
- Network & security configuration
- Lab: Deploy and connect to Atlas cluster

---

12. Getting Started with MongoDB Atlas (1 hr)

MongoDB Atlas is MongoDB's **fully managed cloud database service**, available on AWS, Azure, and GCP.

Admins use Atlas for **production workloads**, scaling, backup, global deployment, and monitoring.

A. Create Free Tier Cluster (15 min)

Atlas provides an **M0 Free Tier cluster** that is ideal for learning and prototyping.

✔ Step-by-Step Instructions

Step 1 — Sign Up

Go to:

```
https://www.mongodb.com/cloud/atlas
```

Sign up using:

- Google account, or
- Email + password

Step 2 — Create New Project

Atlas organizes clusters inside **projects**.

Steps:

1. Click **New Project**
2. Enter name:

   CSC-Mongo-Lab

3. Add members (optional)

Step 3 — Create Cluster

Select:

- **M0 Free Tier**
- Choose cloud provider: **AWS / Azure / GCP**
- Choose region nearest to India:

  AWS ap-south-1 (Mumbai) → recommended

Step 4 — Choose Cluster Name

Default: Cluster0

Recommended name:

csc-training-db

Step 5 — Create Database User

Create a user for client connections:

Username:

```
cscAdmin
```

Password:

```
Choose strong password
```

Role:

readWriteAnyDatabase (for training)

Step 6 — Add IP Access List

To allow connections, add:

```
0.0.0.0/0
```

(Allows access from any IP — OK for lab, not recommended for production.)

Step 7 — Wait for Cluster Deployment

Takes 3–5 minutes.

B. Cluster Deployment Options (10 min)

Atlas supports multiple deployment sizes and architectures.

1. M0/M2/M5 (Shared Tier)

- Free or low-cost
- Shared instances
- No dedicated RAM/CPU
- Good for learning & prototypes

---

2. Dedicated Clusters (M10 and above)

- Full control of compute
- Production-ready
- Supports sharding & backup

---

3. Global & Multi-region Deployment

Used by enterprises like CSC for:

- Low latency across teams
- Disaster recovery
- Data residency compliance

Modes:

- Multi-region write
- Multi-region read
- Custom regional failover

---

4. Advanced Features

- Auto-scaling compute & storage
- Serverless instances
- Atlas Search
- Atlas Device Sync
- Online archive

C. Network & Security Configuration (15 min)

Security is a top priority in Atlas.

---

1. IP Access List (Firewall)

Atlas allows connections *only* from whitelisted IP addresses.

Add:

```
Add My Current IP
```

or for lab:

```
0.0.0.0/0
```

---

2. Database User Authentication

RBAC-based permissions:

- read
- readWrite
- dbAdmin
- clusterAdmin

Creation UI:

Database Access → Add New Database User

---

3. TLS/SSL Encryption

Atlas **forces TLS connections**, ensuring secure communication.

Connection string includes:

```
tls=true
```

---

4. Network Peering (for enterprises)

Connect Atlas to:

- AWS VPC
- Azure VNet
- GCP VPC

Used by CSC for:

- Secure private connection
- Isolation from internet
- Compliance requirements

---

5. Encryption at Rest (Automated)

Atlas automatically encrypts storage using:

- AWS KMS
- Azure Key Vault

- GCP KMS

D. LAB — Deploy & Connect to Atlas Cluster (20 min)

🎯 Goal:

Learners should be able to create a cluster, configure security, and connect using:

- **MongoDB Compass**
- **mongosh**
- **Application driver** (Node.js/Python example)

LAB PART 1: Create Free Tier Cluster

Follow steps from Section A:

1. Create cluster
2. Add IP whitelist
3. Create DB user

LAB PART 2: Connect Using MongoDB Compass

Step 1 — Open Compass

Download from:

https://www.mongodb.com/try/download/compass

Step 2 — Get Connection String

In Atlas UI:

Connect → Compass → Copy connection string

Example:

```
mongodb+srv://cscAdmin:<password>@csc-training-db.abcde.mongodb.net/test
```

Replace <password> with your actual password.

Step 3 — Connect

Paste connection string into Compass and click **Connect**.

Students should be able to:

- View databases
- Add sample data
- Run queries

LAB PART 3: Connect Using mongosh

Step 1 — Install mongosh (if needed)

https://www.mongodb.com/try/download/shell

Step 2 — Connect to Atlas

```
mongosh "mongodb+srv://cscAdmin:<password>@csc-training-
db.abcde.mongodb.net"
```

Step 3 — Create database & collection

```
use atlas_lab;

db.products.insertOne({ name: "Laptop", price: 75000 });
```

LAB PART 4: Connect Using Application Driver

Node.js Example

```
const { MongoClient } = require("mongodb");

const uri = "mongodb+srv://cscAdmin:<password>@csc-training-
db.abcde.mongodb.net/";

const client = new MongoClient(uri);

async function run() {
  await client.connect();
  const db = client.db("atlas_lab");
  console.log(await db.listCollections().toArray());
}

run();
```

🎯 LAB Outcomes

Students will be able to:

- Deploy & configure Atlas cluster
- Understand free vs dedicated vs global clusters
- Configure IP access, DB users, TLS
- Connect via Compass, mongosh, and drivers
- Start using Atlas as their primary MongoDB environment

MongoDB Administrator Tools

- mongosh
- mongodump, mongorestore
- mongotop, mongostat
- Compass & Atlas Metrics
- Lab: Use admin tools on a dataset

MongoDB administrators rely on a combination of **shell tools**, **diagnostic utilities**, and **UI tools** to manage, monitor, and troubleshoot clusters.

This module covers the core admin tools used in **on-prem, self-managed, and Atlas environments**.

A. mongosh – The MongoDB Shell (10 min)

mongosh is the modern JavaScript-based MongoDB shell.

✔ Common Admin Commands

1. Connect to MongoDB

```
mongosh "mongodb://localhost:27017"
```

2. Show databases

```
show dbs
```

3. Show collections

```
use cscdb
show collections
```

4. Add admin user

```
db.createUser({
  user: "cscAdmin",
  pwd: "StrongPass123",
  roles: ["root"]
});
```

5. Server status (monitoring snapshot)

```
db.serverStatus()
```

⭐ CSC Use Cases

- Verify authentication status
- Check replication state
- Inspect server health before deployments
- Run automated scripts for compliance data loads

B. mongodump & mongorestore (15 min)

These tools are essential for **backup**, **migration**, and **offline restore**.

---

1. mongodump (Backup)

Creates a BSON dump of the database.

Dump entire database

```
mongodump --db cscdb --out /backup/cscdb-backup
```

Dump a collection

```
mongodump --db cscdb --collection users --out /backup/users-backup
```

Dump Atlas cluster

```
mongodump --uri "mongodb+srv://<user>:<pw>@cluster.mongodb.net"
```

---

2. mongorestore (Restore)

Restores BSON backup to a MongoDB instance.

Restore a database

```
mongorestore --db cscdb /backup/cscdb-backup/cscdb
```

Restore a collection

```
mongorestore --collection users /backup/users-backup/cscdb/users.bson
```

⭐ CSC Use Cases:

- Migrating client legal entity data
- Creating training/staging environments
- Restoring archived compliance datasets
- Backing up before major schema changes

C. mongotop & mongostat (15 min)

These tools help admins monitor **real-time activity**.

---

1. mongotop

Shows how much time the database spends reading/writing per collection.

Run:

```
mongotop 3
```

Output refreshed every **3 seconds**.

Interpretation

Column	Meaning
ns	Namespace (db.collection)
command	Query time
update	Update time
read/write	Time spent in disk ops

⭐ Use Cases at CSC:

- Detect hot collections (e.g., compliance events)
- Identify collections causing disk pressure
- Debug indexing spikes

⭐ 2. mongostat

Shows key server metrics.

Run:

```
mongostat 2
```

Columns include:

- inserts / updates / deletes
- qr | qw (queued reads/writes)
- ar | aw (active reads/writes)
- netIn / netOut
- connections
- % dirty / % used (WiredTiger cache)

⭐ What DBAs look for

Symptom	Meaning
High queued writes (qw)	Disk bottleneck
High connections	App connection leak
Cache dirty > 40%	Slow checkpoints
inserts spikes	Batch jobs or migrations

D. Compass & Atlas Metrics (10 min)

MongoDB Compass and Atlas UI provide graphical admin insights.

MongoDB Compass

Useful for:

- Browsing databases
- Running queries
- Schema visualization
- Index analysis
- Real-time performance charts

Atlas Metrics Dashboard

Atlas provides:

- CPU usage
- Memory usage
- IOPS
- Connections
- Replication lag
- Slow query panel
- Performance advisor (index recommendations)

⭐ CSC-Relevant Scenarios:

- Monitor ingestion workloads
- Validate health post-deployment
- Track performance before/after schema or index changes
- Investigate spikes in client-facing apps

E. LAB – Use Admin Tools on a Dataset (10 min)

Lab Goal:

Learner should be able to:

- Import dataset
- Inspect with mongostat/mongotop
- Perform backup/restore
- View metrics in Compass or Atlas

Step 1 — Load Sample Dataset

Create a test collection:

```
use admintools;
for (let i = 1; i <= 20000; i++) {
  db.events.insertOne({
    eventId: i,
    type: ["INFO", "WARN", "ERROR"][i % 3],
    user: ["admin", "system", "api"][i % 3],
    ts: new Date(),
    msg: "Event log " + i
  });
}
```

Step 2 — Run mongostat

```
mongostat 2
```

Observe:

- inserts increasing
- connections
- opcounters

Step 3 — Run mongotop

```
mongotop 3
```

Expected:

admintools.events shows high read/write activity.

Step 4 — Run mongodump / mongorestore

Backup:

```
mongodump --db admintools --out /tmp/backup
```

Drop collection:

```
db.events.drop();
```

Restore:

```
mongorestore --db admintools /tmp/backup/admintools
```

Check:

```
db.events.countDocuments()
```

Step 5 — Open in Compass

- Connect
- Validate restored data
- Check schema tab

- Inspect index usage

Step 6 — Atlas Metrics (If using Atlas)

- Navigate to Metrics
- View CPU spikes
- Check connections
- Open Profiler

🎯 LAB Outcomes:

Learners will understand:

- Backup & restore
- Real-time server monitoring
- Performance diagnostics
- Using both command-line + GUI tools

Following Topics should be included with Lab Example

Database Metrics & Monitoring

- Understanding MongoDB metrics
- WiredTiger cache
- IOPS, CPU, memory metrics
- Monitoring tools: Atlas, Cloud Manager, Prometheus, Grafana
- Lab: Live monitoring of workload

Below is your **full instructor-ready Module 14: Database Metrics & Monitoring (1 hr 30 min)** for the MongoDB Administrator Course.

Database Metrics & Monitoring (1 hr 30 min)

Database monitoring is the backbone of MongoDB administration.

As workloads scale, admins must track:

- Latency

- Throughput
- Cache health
- Disk I/O
- Replication lag
- Query efficiency

A. Understanding MongoDB Metrics (20 min)

MongoDB exposes multiple categories of metrics through **serverStatus**, **collStats**, and monitoring tools.

---

1. Key Database Metrics

Latency Metrics

- Query latency (ms)
- Write latency
- Lock wait time

Throughput Metrics

- ops/sec → inserts, queries, updates
- network in/out
- queued operations

Storage Engine Metrics

- WiredTiger cache usage
- Checkpoint activity
- Disk I/O latency
- Dirty vs clean pages

Replication Metrics

- Oplog application time
- Replication lag
- Elections

2. How to check metrics manually

Query serverStatus:

```
db.serverStatus()
```

Collection-level metrics:

```
db.collection.stats()
```

WiredTiger metrics:

```
db.serverStatus().wiredTiger
```

---

3. CSC Enterprise Use Cases

Monitoring helps CSC administrators:

- Identify slow legal search queries
- Detect spikes during client onboarding
- Track compliance event ingestion load
- Diagnose Atlas cluster cost drivers
- Prevent replication lag across global offices

B. WiredTiger Cache (20 min)

WiredTiger is the default storage engine in modern MongoDB versions.

⭐ 1. What is WiredTiger Cache?

It is the in-memory cache used for:

- Document reads
- Index reads
- Write buffering
- Page management

Default size:

50% of RAM for standalone

25% of RAM for sharded clusters

⭐ 2. Key cache metrics

Check cache metrics:

```
db.serverStatus().wiredTiger.cache
```

Important fields:

Metric	Meaning
bytes currently in cache	Amount of data in RAM
maximum bytes configured	Cache size limit
pages read into cache	Read load
pages written from cache	Write load
% cache dirty	Dirty pages waiting flush
checkpoint blocked page eviction	Checkpoint causing stalls

⭐ 3. Indicators of a healthy cache

✔ Cache usage < 80%

✔ Dirty pages < 20–25%

✔ Page read/write ratio stable

✔ No checkpoint stalls

⭐ 4. Symptoms of cache issues

Symptom	Cause	Fix
Dirty pages > 40%	Heavy writes	Add index, optimize writes, scaling
Many page evictions	Cache thrashing	Increase instance size
Read stalls	Poor indexes	Create appropriate indexes
Checkpoint slow	Slow disk	Use SSD, adjust cache

⭐ 5. CSC Example

During quarterly client data ingestion, CSC team notices:

- Cache dirty pages 45%
- Slow dashboard queries

Action:

Add index on { clientId, createdAt } → queries become 8× faster.

C. IOPS, CPU, Memory Metrics (15 min)

⭐ 1. CPU Metrics

- % CPU usage
- CPU spikes → caused by sorting, aggregations, or full scans
- High "user" CPU → queries
- High "system" CPU → OS-level overhead

⭐ 2. Memory Metrics

- WiredTiger cache usage
- Resident memory
- Queued read/write operations
- Page faults → insufficient memory

⭐ 3. Disk Metrics (IOPS)

IOPS = Operations Per Second

Measure:

- read IOPS
- write IOPS
- fsync IOPS during checkpoints

High IOPS usage indicates:

- Index building
- Large aggregations
- Update-heavy workloads

⭐ 4. How to identify IOPS problems

Look for:

```
mongostat
```

Indicators:

- High “flushes”
- High “dirty” percentage
- High queue length

D. Monitoring Tools (Atlas, Cloud Manager, Prometheus, Grafana) (20 min)

⭐ 1. MongoDB Atlas Monitoring

Atlas provides built-in dashboards:

Key charts in Atlas:

- CPU
- Memory
- IOPS
- Connections
- Opcounters
- Replication lag
- Slow queries
- Index suggestions

Atlas Profiler

Shows:

- Slowest queries
- Query shapes
- Execution plans
- Index suggestions

⭐ 2. Cloud Manager (Self-managed deployments)

Cloud Manager provides:

- Automation
- Backups
- Monitoring
- Alerts

Used for:

- On-prem MongoDB clusters
- Hybrid environments

⭐ 3. Prometheus + Grafana

These tools are used by DevOps/SRE teams.

Prometheus:

- Pulls metrics from MongoDB Exporter
- Stores time series metrics

Grafana:

- Visualizes dashboard panels:
  - CPU
  - Memory
  - Op counters
  - IOPS
  - Replication lag
  - Cache usage

⭐ Recommended Grafana Dashboards

- MongoDB Overview
- WiredTiger Cache
- Replication Performance
- Cluster Health

E. LAB — Live Monitoring of Workload (15 min)

🎯 Goal:

Generate load → monitor metrics in real time → identify anomalies.

LAB PART 1 — Insert workload generator

Create 50k documents:

```
use monitorlab;

for (let i = 1; i <= 50000; i++) {
  db.logs.insertOne({
    logId: i,
    type: ["INFO", "WARN", "ERROR"][i % 3],
    user: ["system", "client", "admin"][i % 3],
    ts: new Date(),
    details: "Log event " + i
  });
}
```

LAB PART 2 — Monitor via mongostat

In terminal:

```
mongostat 2
```

Observe:

- inserts
- queries
- updates
- connections
- queue depth

LAB PART 3 — Monitor via mongotop

```
mongotop 3
```

Observe:

monitorlab.logs gaining heavy read/write time.

LAB PART 4 — Observe Cache Metrics

```
db.serverStatus().wiredTiger.cache
```

Look for:

- bytes in cache
- dirty bytes
- eviction rates

LAB PART 5 — Atlas Metrics (if using Atlas)

Open Metrics dashboard and observe:

- CPU %
- IOPS
- Network usage
- Slow query panel

LAB PART 6 — Identify a Slow Query

Run:

```
db.logs.find({ type: "ERROR" }).sort({ ts: -1 }).limit(5);
```

View profiler:

```
db.system.profile.find().sort({ millis: -1 }).limit(1).pretty();
```

LAB PART 7 — Fix with Index

Create:

```
db.logs.createIndex({ type: 1, ts: -1 });
```

Re-run workload and compare metrics in:

- mongostat

- mongotop
- serverStatus
- Atlas

🎯 LAB Outcomes:

Students will:

- Understand how metrics react to workload
- Interpret cache, CPU, IOPS indicators
- Detect slow queries in real time
- Use multiple monitoring tools together
- Optimize using indexes

⭐ End of Day Assignment (Indexes + Profiler & Logs)

Objective

By the end of this assignment, the learner should be able to:

- Identify slow queries and **speed them up using indexes**
- Use explain() to **measure performance gain**
- Configure the **profiler**, generate logs, and analyze **slow queries**

🧩 PART A – Create Indexes & Measure Performance Gain (45–60 min)

1️⃣ Dataset Setup

Open mongosh and run:

```
use day2_perf_lab;

for (let i = 1; i <= 100000; i++) {
  db.orders.insertOne({
    orderId: i,
    customerId: "CUST" + (i % 5000),
    status: ["NEW", "SHIPPED", "DELIVERED", "CANCELLED"][i % 4],
    amount: Math.floor(Math.random() * 9000) + 1000,
    country: ["IN", "US", "UK", "DE", "SG"][i % 5],
    createdAt: new Date(2024, (i % 12), (i % 28) + 1)
  });
```

```
}
```

2️⃣ Task 1 – Baseline Query (Without Index)

Run this query and capture the execution stats:

```
db.orders.find(
  { country: "IN", status: "DELIVERED" }
).sort({ createdAt: -1 }).limit(10).explain("executionStats");
```

Record in your notes:

- executionTimeMillis
- executionStats.totalDocsExamined
- executionStats.totalKeysExamined
- executionStages.stage (COLLSCAN or IXSCAN?)

3️⃣ Task 2 – Design & Create Index

Based on the query pattern, design a **compound index**.

Suggested:

```
db.orders.createIndex(
  { country: 1, status: 1, createdAt: -1 }
);
```

4️⃣ Task 3 – Rerun Query (With Index)

Run the **same query** again with explain:

```
db.orders.find(
  { country: "IN", status: "DELIVERED" }
).sort({ createdAt: -1 }).limit(10).explain("executionStats");
```

Now record again:

- executionTimeMillis
- totalDocsExamined
- totalKeysExamined
- executionStages.stage

👉 Compare Before vs After:

Write a short note:

1. How much did executionTimeMillis change?

2. Did totalDocsExamined go down significantly?
3. Did the plan change from COLLSCAN to IXSCAN?
4. Is this index suitable for this query? Why?

Task 4 – Extra: Test Wrong Index Order (Optional)

Create another index:

```
db.orders.createIndex({ status: 1, country: 1, createdAt: -1 });
```

Now run the **same query** again with explain().

- Does MongoDB use the "better" index or the "wrong-order" index?
- Check the winningPlan in explain() and identify **which index** was used.

🧩 PART B – Generate & Analyze Profiler Logs (30–45 min)

Task 5 – Enable Profiler

Set the profiler to log operations slower than **20 ms**:

```
use day2_perf_lab;
db.setProfilingLevel(1, { slowms: 20 });
```

Verify:

```
db.getProfilingStatus();
```

Task 6 – Trigger Slow Queries

Run **two different queries**:

Query A – Less selective filter

```
db.orders.find({ status: "DELIVERED" }).sort({ createdAt: -1 }).limit(50);
```

Query B – More selective filter (indexed)

```
db.orders.find(
  { country: "IN", status: "DELIVERED" }
).sort({ createdAt: -1 }).limit(20);
```

Run each 2–3 times.

8️⃣ Task 7 – Read Profiler Output

Check the slowest operations:

```
db.system.profile.find().sort({ millis: -1 }).limit(5).pretty();
```

For at least **two entries**, note down:

- ns (namespace)
- command (filter + sort used)
- millis
- planSummary (COLLSCAN / IXSCAN)
- docsExamined / keysExamined (if present)

Answer:

1. Which query (A or B) appears slower in profiler?
2. Does planSummary show COLLSCAN anywhere?
3. Are any queries NOT using the index you created? Why might that be?

9️⃣ Task 8 – Fix a Slow Query Using Profiler Insight

1. From system.profile, pick **one slow query** that is not using an index optimally.
2. Design a new index for that query's filter/sort.
3. Create the index using createIndex().
4. Rerun the query & check profiler again.

Write a short summary:

- What was the original query?
- What index did you create?
- How did millis and docsExamined change?

Reflection Questions (Short Written Answers)

1. Why can a **compound index** be better than multiple single-field indexes?
2. What is the difference between **COLLSCAN** and **IXSCAN** in explain()?
3. Why should you be careful with setting profiler level to 2 in production?
4. How can **profiler + indexes** together form a performance tuning workflow?
