5. Maintenance & Performance Optimization

1. Common DBA maintenance tasks
2. Query optimization (with explain)
3. Cache handling (WiredTiger)
4. Index lifecycle management
5. Lab: identify & fix performance bottlenecks

---

0. CSC Scenario – What Are We Optimizing?

CSC runs a **Compliance & Filings** platform with DB: csc_compliance

Collections:

- entities – each legal entity CSC manages
- filings – all statutory filings (annual reports, franchise tax, etc.)
- reminders – upcoming filing reminders

Symptom:

- At **month-end**, Bangalore SRE team sees **slow dashboards & timeouts**.
- Mumbai reporting team runs heavy queries for **DE & CA filings**.

Goal of this session:

- Find **slow queries** and **inefficient indexes**
- Understand **cache behavior**
- Establish a **maintenance routine** for CSC MongoDB clusters (self-managed & Atlas)

For examples, assume:

```
use csc_compliance
```

---

1. Common DBA Maintenance Tasks (Self-Managed + Atlas)

1.1 Check Basic Health & Stats

In mongosh:

```
db.stats()          // per-DB stats
db.filings.stats()  // per-collection stats
```

Key fields:

- count – number of documents
- size – data size
- storageSize – space allocated
- nindexes – number of indexes

✅ **CSC use case**

CSC DBAs in Bangalore run this monthly to see how fast filings is growing and plan storage/indexing strategy.

1.2 Check Indexes

```
db.filings.getIndexes()
```

Look out for:

- Duplicate indexes
- Redundant compound indexes
- Missing expected indexes (e.g. on state, status, due_date)

1.3 Check Logs for Slow Queries (Self-Managed)

Set profiling (lab-level, not prod-level):

```
db.setProfilingLevel(1, { slowms: 50 })  // log queries slower than 50ms
db.getProfilingStatus()
```

Look at system.profile:

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

You'll see slow operations, their filters, and millis.

On **Linux**: slow logs are also in /var/log/mongodb/mongod.log

On **Windows**: in the log path from mongod.cfg

On **Mac**: typically /usr/local/var/log/mongodb/mongo.log or similar

✅ **CSC use case**

SRE finds slow queries for "open DE filings" and "overdue filings" in system.profile. These become the **top candidates** for optimization.

1.4 Atlas Equivalent

In **Atlas**:

- Use **Performance Advisor** (if available): suggests indexes.
- Use **Profiler** in Atlas UI:
  - Choose DB csc_compliance
  - Filter slow operations (>50ms)
- Check **Metrics → Query Targeting & Operation Execution Time**.

1.5 Basic Maintenance Tasks Checklist

Self-managed (Linux / Windows / Mac):

- Monitor slow queries (system.profile, logs)
- Monitor disk usage (OS + db.stats())
- Check index health (getIndexes())
- Schedule compactions / repairs when needed
- Apply minor upgrades / patches regularly

Atlas:

- Monitor Metrics dashboard (CPU, IOPS, WT cache, latency)
- Review Performance Advisor recommendations
- Manage auto-scaling / instance size

---

2. Query Optimization Techniques – Hands-On

We'll use **one slow query** and fix it.

2.1 Create "Realistic" Data (Lab Setup)

Run once on any platform:

```
use csc_compliance

// Create 50k filings
for (let i = 0; i < 50000; i++) {
  db.filings.insertOne({
    entity_id: 1000 + (i % 1000),
    filing_type: (i % 2 === 0) ? "Annual Report" : "Franchise Tax",
    state: (i % 3 === 0) ? "DE" : ((i % 3 === 1) ? "CA" : "NY"),
    due_date: new Date(2025, (i % 12), 1),
    filed_date: (i % 5 === 0) ? new Date(2025, (i % 12), 5) : null,
    status: (i % 5 === 0) ? "FILED" : "OPEN",
    amount: 1000 + (i % 200) * 10
  })
}
```

2.2 Slow Query Example

We want:

"All **OPEN filings in DE**, sorted by due_date soonest first."

Query:

```
db.filings.find({
  state: "DE",
  status: "OPEN"
}).sort({ due_date: 1 }).limit(50)
```

2.3 Analyze with explain("executionStats")

```
db.filings.explain("executionStats").find({
  state: "DE",
  status: "OPEN"
}).sort({ due_date: 1 }).limit(50)
```

Look at:

- executionTimeMillis
- totalDocsExamined
- totalKeysExamined
- winningPlan.stage (COLLSCAN?)

If it shows COLLSCAN and many docsExamined, it's a good candidate for an index.

✅ **CSC use case**

This is exactly what slows down CSC's **Open Filings dashboard** for Delaware clients.

2.4 Add a Supporting Index

We filter on state + status and sort by due_date:

```
db.filings.createIndex({ state: 1, status: 1, due_date: 1 })
```

Re-run explain:

```
db.filings.explain("executionStats").find({
  state: "DE",
  status: "OPEN"
}).sort({ due_date: 1 }).limit(50)
```

Now you should see:

- winningPlan.stage → IXSCAN / index-based plan
- totalDocsExamined and executionTimeMillis **reduced**

✅ **CSC use case**

After deploying this index, CSC dashboards for "Open DE Filings" become noticeably faster under load.

2.5 Rewrite Bad Queries

If you had something like:

```
db.filings.find({
  state: { $in: ["DE", "CA"] },
  status: { $ne: "FILED" }
})
```

Consider:

- Avoid $ne where possible (hard for index engine).
- Use more selective conditions first (in index order).
- Use equality/specific ranges instead of very broad conditions.

---

3. Cache Handling (WiredTiger Cache)

WiredTiger cache is **where hot documents & indexes live**. If it's too small or overloaded, MongoDB hits disk more often → slower.

3.1 Inspect Cache via

serverStatus

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

Important fields (names may vary slightly by version):

- bytes currently in the cache
- maximum bytes configured
- tracked dirty bytes in the cache
- pages read into cache / pages written from cache

Compute usage %:

```
const wt = db.serverStatus().wiredTiger.cache;
(wt['bytes currently in the cache'] / wt['maximum bytes configured']) * 100
```

3.2 Interpreting CSC Behavior

If:

- Cache usage consistently ~90–95%
- High pages read from disk
- High IOPS and latency

Then:

- Working set doesn't fit into RAM.
- Consider:
  - **Scaling up** instance (Atlas)
  - **Moving cold data** (archive old filings)
  - **Reviewing indexes** (too many / too large)

3.3 Adjust Cache Size (Self-Managed)

You can set cacheSizeGB (usually only if you know what you're doing):

In /etc/mongod.conf (Linux) or mongod.cfg (Windows) or Mac config:

```
storage:
  wiredTiger:
```

```
    engineConfig:
      cacheSizeGB: 4
```

Then restart service.

✅ **CSC use case**

On a Linux server with 32GB RAM, CSC tunes WiredTiger cache to ~16–20GB while leaving RAM for OS + other processes, based on monitoring.

3.4 Atlas Cache

In Atlas:

- Cache size is tied to the **instance size** (M10, M20, …).
- You handle cache issues primarily by **scaling the cluster** or optimizing queries/indexes.

---

4. Index Lifecycle Management

Indexes are not "set-and-forget". They must be managed over time.

4.1 Create, List, Drop

Create:

```
db.filings.createIndex({ state: 1, status: 1, due_date: 1 }, { name:
"idx_state_status_due" })
```

List:

```
db.filings.getIndexes()
```

Drop:

```
db.filings.dropIndex("idx_state_status_due")
```

4.2 Hidden Indexes (Test Before Drop) – MongoDB 4.4+

You can **hide** an index to see if it's still needed.

```
db.filings.hideIndex("idx_state_status_due")
```

Monitor performance and logs. If everything still runs fast, index might be unused.

Unhide if needed:

```
db.filings.unhideIndex("idx_state_status_due")
```

Then decide to drop or keep.

✅ **CSC use case**

CSC had a heavy, old compound index that slowed writes.
They **hid** it for a week, observed no impact, then safely dropped it to make inserts faster.

4.3 Rebuilding / Defragmenting Indexes

On large collections, indices can get fragmented.

```
db.filings.reIndex()
```

Use sparingly and only in low-traffic windows.

4.4 TTL Indexes (For reminders)

For reminders that expire:

```
db.reminders.createIndex(
  { expiry_date: 1 },
  { expireAfterSeconds: 0 }
)
```

Documents automatically deleted when expiry_date has passed.

✅ **CSC use case**

TTL keeps reminders small and hot, avoiding bloat and reducing cache pressure.

---

5. LAB – Identify & Fix Performance Bottlenecks

Here's a **step-by-step lab** you can run on **Windows, Mac, Linux, and Atlas**.

5.1 Step 0 – Setup (All Platforms)

1. Make sure csc_compliance.filings has ~50k docs (from above loop).
2. Ensure **profiling** is on for slow ops:

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

5.2 Step 1 – Create a Bad Query Pattern

Run this query multiple times:

```
for (let i = 0; i < 20; i++) {
  db.filings.find({
    state: "DE",
    status: "OPEN",
    amount: { $gt: 1500 }
  }).sort({ amount: -1 }).limit(100).toArray();
}
```

5.3 Step 2 – Find the Slow Query (Profiler / Logs)

Check profiler:

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

Note:

- ns (should be csc_compliance.filings)
- millis
- docsExamined
- Query shape (command.filter & command.sort)

✅ CSC use case

You identify heavy queries on state, status, amount. These are likely from the “High Value Open Filings” dashboard.

5.4 Step 3 – Analyze with explain

Take the offending query and run:

```
db.filings.explain("executionStats").find({
  state: "DE",
  status: "OPEN",
  amount: { $gt: 1500 }
}).sort({ amount: -1 }).limit(100)
```

Check:

- executionTimeMillis
- totalDocsExamined

- totalKeysExamined
- winningPlan.inputStage or winningPlan.stage

If it's COLLSCAN or scans far more docs than it returns, optimization is needed.

5.5 Step 4 – Design and Apply Index

We filter on { state, status, amount } and sort by amount:

```
db.filings.createIndex({ state: 1, status: 1, amount: -1 }, { name:
"idx_state_status_amount_desc" })
```

Re-run explain:

```
db.filings.explain("executionStats").find({
  state: "DE",
  status: "OPEN",
  amount: { $gt: 1500 }
}).sort({ amount: -1 }).limit(100)
```

Expect:

- index-based plan
- fewer docsExamined
- lower executionTimeMillis

5.6 Step 5 – Observe Impact (Self-Managed vs Atlas)

Self-Managed:

- Watch system.profile entries again – are millis lower?
- Optionally, watch OS metrics:
  - Linux: top, iostat, vmstat
  - Windows: Task Manager, Performance Monitor
  - Mac: Activity Monitor

Atlas:

- Check **Metrics → Operation Execution Time** (zoom into last 5 min).
- Check **Query Targeting** chart (fewer "scanned / returned" ratio).

5.7 Step 6 – Clean Up & Lifecycle

1. Check all indexes:

```
db.filings.getIndexes()
```

1. Hide an index that looks redundant and observe for a day (in real life):

```
db.filings.hideIndex("some_old_index_name")
```

2. If safe – drop it:

```
db.filings.dropIndex("some_old_index_name")
```

✅ **CSC use case**

Over months, CSC uses this **hide → observe → drop** pattern to keep only the essential indexes, improving write throughput and cache efficiency.

---

6. OS & Atlas Specific Notes (Quick Reference)

Linux

- Service control: sudo systemctl status|restart mongod
- Logs: /var/log/mongodb/mongod.log
- Disk/CPU tools: top, htop, iostat, vmstat, df, du

Windows

- Service: net stop MongoDB / net start MongoDB
- Logs: from mongod.cfg path
- Monitoring: Task Manager, PerfMon counters

Mac

- Homebrew service: brew services restart mongodb-community
- Default db/log dirs under /usr/local/var or /opt/homebrew/var
- Monitoring: Activity Monitor, top, vm_stat

Atlas

- No file access; everything via UI/API.
- Use:
    - **Metrics** (CPU, IOPS, cache, latency)
    - **Profiler**
    - **Performance Advisor**
