UseCase_Module 3:Index & Statistics Tuning

DAY 2 – Indexing, Statistics & Concurrency
Module 3: Index & Statistics Tuning (4 Hours)
Concepts

- Clustered vs Non-clustered indexes
- Covering & filtered indexes
- Columnstore indexes (when to use)
- Index fragmentation myths
- Fill factor strategy
- Statistics:
  - Histograms
  - Auto-update thresholds
  - Multi-column stats

Hands-on Labs

- Create optimal indexes for slow queries
- Analyze missing index DMVs
- Rebuild vs reorganize decisions
- Fix queries using statistics updates

Tools Used

- sys.dm_db_index_usage_stats
- sys.dm_db_missing_index_details
- DBCC SHOW_STATISTICS

Create CSC Global Use Cases for DBA professionals. Also explain each line of code.

Module 3: Index & Statistics Tuning (4 Hours)

I’ve structured it exactly how **senior DBA training** should run:

- Real **enterprise use cases**
- **Hands-on labs**
- **Every line of code explained**
- Clear **why this matters in production**

DAY 2 – Indexing, Statistics & Concurrency

Module 3: Index & Statistics Tuning (4 Hours)

USE CASE 1 – “Overdue Compliance Report is CPU Heavy”

(Clustered vs Nonclustered + Covering Index)

Business Context (CSC Style)

The global compliance dashboard is slow every morning.

DBA suspects **scans + key lookups**.

Lab 1A – Observe the problem

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;

SELECT
   e.country_code,
   e.client_name,
   cf.filing_type,
   COUNT_BIG(*) AS overdue_count
FROM dbo.ComplianceFilings cf
JOIN dbo.Entities e
   ON e.entity_id = cf.entity_id
WHERE
   cf.filing_status = 'OVERDUE'
   AND cf.due_date < GETDATE()
GROUP BY
   e.country_code,
   e.client_name,
   cf.filing_type;
```

Line-by-line explanation

- SET STATISTICS IO ON

  Shows logical reads → **true performance cost**

- SET STATISTICS TIME ON

  Shows CPU & elapsed time

- JOIN dbo.Entities e ON e.entity_id = cf.entity_id

  Join depends on index availability

- COUNT_BIG(*)

Safe aggregation for large datasets

- GROUP BY

  Forces aggregation → may cause hash aggregate

Key DBA Learning

🔑 **Scans + Aggregation = expensive if indexes are wrong**

Lab 1B – Create a covering nonclustered index

```
CREATE INDEX IX_CF_Status_DueDate_Cover
ON dbo.ComplianceFilings (filing_status, due_date)
INCLUDE (entity_id, filing_type);
```

Explanation

- filing_status, due_date → WHERE clause
- INCLUDE (entity_id, filing_type) → JOIN + GROUP BY columns
- This **eliminates key lookups**

Learning

🔑 **Covering indexes reduce IO more than almost any other tuning**

USE CASE 2 – "DBA Added 50 Indexes… Performance Got Worse"

(Index Usage + Fragmentation Myths)

Lab 2A – Find unused indexes

```
SELECT
   OBJECT_NAME(i.object_id) AS table_name,
   i.name AS index_name,
   s.user_seeks,
   s.user_scans,
   s.user_lookups,
   s.user_updates
FROM sys.indexes i
LEFT JOIN sys.dm_db_index_usage_stats s
   ON i.object_id = s.object_id
```

```
  AND i.index_id = s.index_id
WHERE OBJECTPROPERTY(i.object_id,'IsUserTable') = 1
ORDER BY s.user_seeks + s.user_scans + s.user_lookups;
```

Line-by-line explanation

- sys.indexes → all indexes
- sys.dm_db_index_usage_stats → how indexes are used
- user_seeks → good usage
- user_scans → may indicate poor design
- user_updates → maintenance overhead
- LEFT JOIN → includes unused indexes

Key DBA Learning

🔑 **Unused indexes slow INSERT/UPDATE/DELETE & waste memory**

Index Fragmentation Myth (Important Teaching Point)

Fragmentation ≠ performance problem for OLTP seeks
Fragmentation mostly affects **range scans**

Lab 2B – Decide rebuild vs reorganize

```
SELECT
   OBJECT_NAME(object_id) AS table_name,
   index_id,
   avg_fragmentation_in_percent,
   page_count
FROM sys.dm_db_index_physical_stats
(
   DB_ID(),
   NULL,
   NULL,
   NULL,
   'SAMPLED'
)
WHERE page_count > 1000;
```

Explanation

- avg_fragmentation_in_percent
  - < 5% → ignore
  - 5–30% → reorganize
  - 30% → rebuild

- page_count > 1000

  Ignore tiny indexes

Learning

🔑 **Fragmentation decisions must be data-driven, not scheduled blindly**

USE CASE 3 – "Client-wise Reports Are Inconsistent"

(Statistics, Histograms & Cardinality)

Lab 3A – Inspect statistics histogram

```
DBCC SHOW_STATISTICS ('dbo.ComplianceFilings', 'IX_CF_Status_DueDate_Cover')
WITH HISTOGRAM;
```

Explanation

- DBCC SHOW_STATISTICS → internal optimizer data
- HISTOGRAM → distribution of column values
- Shows:
  - RANGE_HI_KEY
  - EQ_ROWS
  - RANGE_ROWS

Teaching Point

- SQL Server estimates rows **based on histogram steps**
- Skewed data = wrong estimates

Key DBA Learning

🔑 **Bad statistics = bad plans even with good indexes**

Lab 3B – Show auto-update threshold problem

```
SELECT
  name,
  auto_created,
  STATS_DATE(object_id, stats_id) AS last_updated
FROM sys.stats
```

```
WHERE object_id = OBJECT_ID('dbo.ComplianceFilings');
```

Explanation

- auto_created = stats created automatically
- STATS_DATE = last refresh time

Teaching Point

- Auto-update happens only after ~20% + 500 rows change
- Large tables wait too long

Lab 3C – Fix with manual stats update

```
UPDATE STATISTICS dbo.ComplianceFilings WITH FULLSCAN;
```

Explanation

- FULLSCAN reads all rows
- Best estimates (slower but accurate)

Learning

🔑 **Critical reporting tables need proactive stats maintenance**

USE CASE 4 – "Query Uses Multiple Columns but Estimates Are Wrong"

(Multi-column Statistics)

Lab 4A – Create multi-column statistics

```
CREATE STATISTICS ST_CF_Status_DueDate
ON dbo.ComplianceFilings (filing_status, due_date);
```

Explanation

- Helps optimizer understand **column correlation**
- Single-column stats assume independence (often wrong)

Learning

🔑 **Multi-column stats can fix estimates without adding indexes**

USE CASE 5 – “Missing Index DMV Lies”

(When & How to Use Missing Index DMVs)

Lab 5A – View missing index recommendations

```
SELECT
   migs.avg_total_user_cost,
   migs.avg_user_impact,
   mid.statement,
   mid.equality_columns,
   mid.inequality_columns,
   mid.included_columns
FROM sys.dm_db_missing_index_details mid
JOIN sys.dm_db_missing_index_groups mig
   ON mid.index_handle = mig.index_handle
JOIN sys.dm_db_missing_index_group_stats migs
   ON mig.index_group_handle = migs.group_handle
ORDER BY migs.avg_user_impact DESC;
```

Explanation

- avg_user_impact → estimated improvement %
- equality_columns → WHERE col = ?
- inequality_columns → WHERE col <, >
- included_columns → SELECT list

Teaching Warning

🚨 **Never auto-create missing indexes**

- They:
    - Ignore maintenance cost
    - Ignore existing indexes
    - Often overlap

Learning

🔑 **Missing index DMV is a suggestion engine, not a solution engine**

USE CASE 6 – Columnstore Index (When to Use)

Business Context

Compliance analytics, monthly reporting, read-heavy workload.

Lab 6A – Create Nonclustered Columnstore

```
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI_CF
ON dbo.ComplianceFilings
(
  filing_status,
  due_date,
  entity_id,
  filing_type
);
```

Explanation

- Columnstore compresses data
- Batch-mode execution
- Best for:
  - Large scans
  - Aggregations
  - Reporting

When NOT to use

- High-frequency OLTP updates
- Small tables

Learning

🔑 **Columnstore is a reporting accelerator, not an OLTP index**

FINAL DAY-2 DBA TAKEAWAYS (CSC Level)

After this module, DBAs must know:

1️⃣ Indexes are **workload-specific**, not generic

2️⃣ Covering indexes reduce IO more than CPU tuning

3️⃣ Statistics quality matters as much as indexes

4️⃣ Fragmentation is often over-tuned

5️⃣ Missing index DMVs must be interpreted, not trusted blindly

6️⃣ Columnstore is powerful—but only in the right workload

Interview-Ready One-Liners (Use These)

- “Indexes shape plans; statistics shape estimates.”
- “Fragmentation hurts scans, not seeks.”
- “Missing index DMVs show symptoms, not design.”
- “Bad stats can break even perfect indexes.”

DAY-2 DEMO PACK

(Bad Statistics → Bad Plan → Fix → Verify)

Goal for participants:

Understand how bad or stale statistics alone (even with indexes present) can produce wrong plans, high memory grants, scans, and slow performance—and how a DBA fixes it correctly.

DEMO STORY (CSC Global Context)

“Nightly compliance load added ~80,000 new filings.
Indexes exist, but the morning report is suddenly slow.”

Root cause: **Statistics did not refresh → optimizer underestimates rows → wrong plan.**

STEP 0️⃣ – Instructor Setup

```
-- Always enable these for demo clarity
SET NOCOUNT ON;
SET STATISTICS IO ON;
SET STATISTICS TIME ON;

-- Ask participants to enable:
-- Actual Execution Plan (Ctrl + M in SSMS)
```

🎯 **Tell class:**

"Today, we are NOT fixing indexes first. We'll prove stats alone can break plans."

STEP 1️⃣ – Confirm Index Exists (Important!)

```
SELECT
   i.name AS index_name,
   i.type_desc
FROM sys.indexes i
WHERE i.object_id = OBJECT_ID('dbo.ComplianceFilings')
  AND i.name = 'IX_Filings_Status_DueDate';
```

Output Interpretation

- Index **exists**
- So if performance is bad → **indexes are NOT the problem**

🔑 **Key Learning:**

If an index exists and query is still slow, look at statistics next.

STEP 2️⃣ – SIMULATE BAD STATISTICS

Force statistics to be stale

```
-- Simulate stale stats by disabling auto update (DEMO ONLY)

ALTER DATABASE CURRENT SET AUTO_UPDATE_STATISTICS OFF;
GO
```

⚠️ **Instructor note:**

Emphasize this is **for demo only**, not recommended blindly in prod.

STEP 3️⃣ – RUN QUERY WITH BAD STATS (BAD PLAN)

```
DECLARE @AsOfDate DATE = CAST(GETDATE() AS DATE);

SELECT
   e.country_code,
   e.client_name,
   cf.filing_type,
   COUNT_BIG(*) AS overdue_count
FROM dbo.ComplianceFilings cf
JOIN dbo.Entities e
   ON e.entity_id = cf.entity_id
WHERE
   cf.filing_status = 'OVERDUE'
   AND cf.due_date < @AsOfDate
GROUP BY
   e.country_code,
   e.client_name,
   cf.filing_type
ORDER BY
   overdue_count DESC;
GO
```

STEP 4️⃣ – WHAT DBAs SEE (BAD PLAN OUTPUT)

In Actual Execution Plan:

- ❌ **Index Scan** instead of Seek
- ❌ **Hash Match (Aggregate)** with large input
- ❌ **Sort** operator
- ⚠️ Possible **spill warning**
- ⚠️ Huge **memory grant**

In Messages tab:

- High **logical reads**
- Higher **CPU & elapsed time**

🔑 **Key DBA Learning:**

Bad statistics cause bad estimates → bad join + aggregate choices.

STEP 5️⃣ – PROVE STATS ARE WRONG

```
DBCC SHOW_STATISTICS
```

```
(
   'dbo.ComplianceFilings',
   'IX_Filings_Status_DueDate'
)
WITH HISTOGRAM;
```

Explain Output (Important Columns)

Column	Meaning
RANGE_HI_KEY	Upper bound of value range
EQ_ROWS	Exact matches
RANGE_ROWS	Rows between ranges

🎯 **Instructor talking point:**

“If the histogram says ~100 rows but actual is 50,000 → optimizer is blind.”

STEP 6️⃣ – FIX: UPDATE STATISTICS

```
UPDATE STATISTICS dbo.ComplianceFilings WITH FULLSCAN;

UPDATE STATISTICS dbo.Entities WITH FULLSCAN;
GO
```

Why FULLSCAN?

- Reads all rows
- Best histogram accuracy
- Perfect for reporting tables

🔑 **Key DBA Learning:**

Statistics quality is often more important than index quantity.

STEP 7️⃣ – VERIFY (SAME QUERY, BETTER PLAN)

```
DECLARE @AsOfDate2 DATE = CAST(GETDATE() AS DATE);

SELECT
   e.country_code,
   e.client_name,
   cf.filing_type,
   COUNT_BIG(*) AS overdue_count
FROM dbo.ComplianceFilings cf
JOIN dbo.Entities e
   ON e.entity_id = cf.entity_id
WHERE
   cf.filing_status = 'OVERDUE'
   AND cf.due_date < @AsOfDate2
GROUP BY
   e.country_code,
   e.client_name,
   cf.filing_type
ORDER BY
   overdue_count DESC;
GO
```

What improves

- ✅ Index **Seek**
- ✅ Smaller hash/sort
- ✅ Reduced memory grant
- ✅ Lower reads
- ✅ Faster runtime

STEP 8️⃣ – CLEANUP (IMPORTANT)

```
ALTER DATABASE CURRENT SET AUTO_UPDATE_STATISTICS ON;
GO

SET STATISTICS IO OFF;
SET STATISTICS TIME OFF;
```

🎯 DAY-2 CORE DBA TAKEAWAY

"Indexes decide access paths.
Statistics decide how many rows SQL Server thinks will flow.
Bad stats break good indexes."

🧭 INDEX MAINTENANCE DECISION FLOWCHART

(Interview + Production Ready)

```
START
 |
 v
Is the query slow?
 |
 +-- NO --> Do nothing
 |
 v
Is index being used?
 |
 +-- NO --> Check missing index DMV
 |
 v
Is index heavily updated?
 |
 +-- YES --> Check index usage stats
 |        |
 |        +-- user_updates high & seeks low?
 |             |
 |             +-- YES --> Drop or redesign index
 |
 v
Check fragmentation (page_count > 1000)
 |
 +-- < 5%  --> Ignore
 |
 +-- 5–30% --> REORGANIZE
 |
 +-- >30%  --> REBUILD
 |
 v
Is query still slow?
 |
 +-- YES --> Check statistics freshness
 |        |
 |        +-- STATS_DATE old?
 |             |
 |             +-- YES --> UPDATE STATISTICS
 |
 v
Are estimates still wrong?
 |
 +-- YES --> Create multi-column stats
 |
 v
Is workload analytical (large scans)?
 |
 +-- YES --> Consider COLUMNSTORE
 |
 v
END (Query stable)
```

Interview-Ready DBA One-Liners

- “Fragmentation hurts scans, not seeks.”
- “Never rebuild indexes blindly—check page count.”
- “Missing index DMV is a suggestion, not a command.”
- “Bad statistics break good indexes.”
- “Columnstore is for analytics, not OLTP.”

---

4 CSC Global–style DBA use cases for **Module 3 (Index & Statistics Tuning)**.

Each use case includes:

- **Scenario (what CSC DBAs face)**
- **Problem symptoms**
- **Bad query / missing support**
- **Step-by-step code** with **line-by-line explanation**
- **What “good plan” should look like**
- **How to validate via DMVs / SHOW_STATISTICS**

Use Case 1 — Covering Index to Remove Key Lookups (High logical reads)

Scenario

CSC portal shows “Open invoices for an entity”. Query is fast sometimes, slow for entities with many invoices because of **Key Lookups** (bookmark lookups) on a large table.

✅ Bad Query (works but inefficient)

```
SELECT invoice_id, due_date, amount, currency_code
FROM dbo.Invoices
WHERE entity_id = @EntityId
  AND invoice_status = 'Open'
```

```
ORDER BY due_date DESC;
```

Why it becomes slow

- Index may exist on (entity_id, invoice_status) but selected columns aren't covered → SQL Server does **Key Lookup** for each matching row.
- Many rows = many lookups = high reads.

✅ Step-by-step Fix (Covering index)

Code

```
CREATE INDEX IX_Invoices_EntityStatus_DueDate_Cover
ON dbo.Invoices (entity_id, invoice_status, due_date DESC)
INCLUDE (amount, currency_code);
GO
```

Line-by-line explanation

- CREATE INDEX IX_Invoices_EntityStatus_DueDate_Cover

  Creates a new **nonclustered index** with a meaningful name.

- ON dbo.Invoices (entity_id, invoice_status, due_date DESC)

  Puts the most selective filters first (entity_id, invoice_status).

  Adds due_date DESC to support the ORDER BY without extra sort.

- INCLUDE (amount, currency_code)

  Stores these columns in the index leaf level so SQL Server can return them **without key lookups**.

- GO

  Ends the batch.

Expected plan improvement

- **Index Seek** on this new index
- **No Key Lookup**
- Often **no Sort operator**

Validate improvement (usage tracking)

```
SELECT *
FROM sys.dm_db_index_usage_stats
WHERE database_id = DB_ID()
  AND object_id = OBJECT_ID('dbo.Invoices');
```

Explanation

- Shows how often each index is used:
    - user_seeks (good)
    - user_scans
    - user_lookups
    - user_updates (write cost)

Use Case 2 — Filtered Index for Rare Status (Overdue Filings)

Scenario

CSC compliance team frequently checks **Overdue filings** (rare data).

A full index on status may still be heavy because the table is huge.

✅ Query

```
SELECT TOP (2000) filing_id, entity_id, due_date, penalty_amount
FROM dbo.ComplianceFilings
WHERE filing_status = 'Overdue'
ORDER BY due_date DESC;
```

✅ Fix: Filtered Index (targeted & small)

Code

```
CREATE INDEX IX_Filings_Overdue_DueDate
ON dbo.ComplianceFilings (due_date DESC)
INCLUDE (entity_id, penalty_amount)
WHERE filing_status = 'Overdue';
GO
```

Line-by-line explanation

- CREATE INDEX IX_Filings_Overdue_DueDate

  Creates a dedicated index specifically for “Overdue” status queries.

- ON dbo.ComplianceFilings (due_date DESC)

  Sort key supports ORDER BY due_date DESC.

- INCLUDE (entity_id, penalty_amount)

  Covers returned columns; avoids lookups.

- WHERE filing_status = 'Overdue'

  This is the **filtered condition**.

  Only “Overdue” rows are stored in this index → far smaller → faster seeks.

Expected plan improvement

- Seek on filtered index
- No scan on main status index
- Very low logical reads

Validate filtered index suggestion impact (Missing index DMV)

```
SELECT TOP (20)
   mid.statement,
   mid.equality_columns,
   mid.inequality_columns,
   mid.included_columns
FROM sys.dm_db_missing_index_details AS mid
ORDER BY mid.index_handle DESC;
```

Explanation

- sys.dm_db_missing_index_details shows **potential** missing indexes.
- Treat as “suggestions”, not truth.
- You still validate with workload and plan.

Use Case 3 — Index Fragmentation "Myth" (Rebuild vs Reorganize decision)

Scenario

A DBA schedules weekly rebuild for *all indexes*.

This increases:

- log generation
- maintenance time
- blocking (if not online)

 ...but performance doesn't improve.

Goal

Teach DBAs to take **data-driven decision**.

Step A: Check fragmentation

```
SELECT
   ips.object_id,
   OBJECT_NAME(ips.object_id) AS table_name,
   ips.index_id,
   ips.avg_fragmentation_in_percent,
   ips.page_count
FROM sys.dm_db_index_physical_stats(DB_ID(), NULL, NULL, NULL, 'SAMPLED') ips
WHERE ips.page_count > 1000
ORDER BY ips.avg_fragmentation_in_percent DESC;
```

Line-by-line explanation

- sys.dm_db_index_physical_stats(...)

  Returns fragmentation info for indexes.

- DB_ID()

  Current database ID.

- NULL parameters

  Means "all tables, all indexes".

- 'SAMPLED'

  Faster than FULL for large DBs; good for daily checks.

- WHERE ips.page_count > 1000

  Ignore tiny indexes—fragmentation there is meaningless.

- ORDER BY avg_fragmentation_in_percent DESC

  Highest fragmentation shown first.

Step B: Decision logic (industry common)

- **< 5%** → do nothing
- **5% to 30%** → REORGANIZE
- **> 30%** → REBUILD

Example code snippets

```
ALTER INDEX IX_Invoices_Status_Due ON dbo.Invoices REORGANIZE;
GO
ALTER INDEX IX_Invoices_Status_Due ON dbo.Invoices REBUILD WITH (FILLFACTOR = 90);
GO
```

Explanation

- REORGANIZE

  Light maintenance, online, less log, slower than rebuild.

- REBUILD

  Full recreation; heavier but resets fragmentation strongly.

- FILLFACTOR = 90

  Leaves free space to reduce page splits for update-heavy indexes.

Use Case 4 — Bad Plans due to Statistics (Histogram + Multi-column Stats)

Scenario

A query filters on **correlated columns** (like country_code + risk_tier).

Without multi-column stats, optimizer assumes independence → wrong estimates → wrong join type.

Query

```
SELECT e.entity_id, e.entity_name
FROM dbo.Entities e
WHERE e.country_code = 'IN'
  AND e.risk_tier = 5;
```

Why estimates go wrong

- country_code and risk_tier may be correlated.
- Optimizer may under/overestimate rows.

Step A: View existing stats histogram

```
DBCC SHOW_STATISTICS ('dbo.Entities', 'IX_Entities_Country') WITH HISTOGRAM;
GO
```

Explanation

- DBCC SHOW_STATISTICS

  Displays statistics details.

- ('dbo.Entities', 'IX_Entities_Country')

  Uses the stats associated with the index.

- WITH HISTOGRAM

  Shows distribution buckets for leading column.

Step B: Create multi-column statistics

```
CREATE STATISTICS ST_Entities_Country_Risk
ON dbo.Entities (country_code, risk_tier);
GO
```

Explanation

- Creates stats on both columns together.
- Helps optimizer estimate combined selectivity better.

Step C: Update stats after major changes

```
UPDATE STATISTICS dbo.Entities ST_Entities_Country_Risk WITH FULLSCAN;
GO
```

Explanation

- FULLSCAN gives most accurate histogram (heavier).
- Great for demo and for critical tables after load.

Expected plan improvement

- Better estimated rows
- Better join choices (in larger queries)
- More stable performance

Final CSC DBA Takeaways (What you want participants to learn)

✅ Covering indexes reduce **Key Lookups**

✅ Filtered indexes are perfect for **rare but critical predicates**

✅ Fragmentation is not a “weekly rebuild for everything” activity

✅ Statistics = optimizer intelligence (histograms + multi-column stats)
