- **Scenario (what CSC sees in production)**
- **Bad plan pattern (what DBAs typically observe)**
- **Code to reproduce (bad)**
- **How to read Estimated vs Actual plan (what to look for)**
- **Fix options (stats / rewrite / index / Query Store plan forcing)**
- **Improved "good plan" approach**
- Plus **sys.dm_exec_query_stats** hooks to measure impact

Use Case 1 — Parameter Sniffing (OLTP Proc: "Entity Filings by Status")

Scenario

CSC compliance portal has a stored procedure:

- Sometimes called for a **rare status** (e.g., "Overdue" = 0.1%)
- Sometimes for a **common status** (e.g., "Pending" = 60%)

First execution compiles plan for one value → reused for others → **random slowness**.

BAD code (creates sniffing behavior)

```
CREATE OR ALTER PROCEDURE dbo.usp_GetFilingsByStatus
  @Status VARCHAR(20)
AS
BEGIN
  SET NOCOUNT ON;

  SELECT f.filing_id, f.entity_id, f.due_date, f.filing_status
  FROM dbo.ComplianceFilings f
  WHERE f.filing_status = @Status
  ORDER BY f.due_date DESC;
END;
GO

-- Bad pattern: first execution compiles plan for rare/common value and
reuses it
EXEC dbo.usp_GetFilingsByStatus @Status = 'Overdue';
EXEC dbo.usp_GetFilingsByStatus @Status = 'Pending';
```

BAD plan pattern (what you'll see)

- Plan compiled for **rare value** might choose **Index Seek + Nested Loops**
- When reused for **common value**, it becomes expensive:
  - lots of lookups
  - high CPU/reads
- Or vice versa: compiled for common value → **Scan/Hash** reused for rare value → slow.

What to check in Estimated vs Actual plan

- **Estimated rows vs Actual rows** at the main access operator
- Join choice, key lookups, memory grant changes

Fix options (in order)

Fix A (quick + safe for skew):

OPTION (RECOMPILE)

```
CREATE OR ALTER PROCEDURE dbo.usp_GetFilingsByStatus
  @Status VARCHAR(20)
AS
BEGIN
  SET NOCOUNT ON;

  SELECT f.filing_id, f.entity_id, f.due_date, f.filing_status
  FROM dbo.ComplianceFilings f
  WHERE f.filing_status = @Status
  ORDER BY f.due_date DESC
  OPTION (RECOMPILE);
END;
GO
```

✅ Best when execution frequency is moderate and skew is high.

Fix B (stabilize with Query Store plan forcing)

Use Query Store after identifying the good plan (see Use Case 5).

Fix C (index to support both patterns)

If you frequently filter + sort:

```
CREATE INDEX IX_Filings_Status_DueDate
ON dbo.ComplianceFilings (filing_status, due_date DESC)
```

```
INCLUDE (entity_id);
GO
```

Use Case 2 — Bad Estimates from Stale Statistics (Join flips wrong)

Scenario

After data load/month-end, queries that join **Entities** → **Filings** become slow.

No code change. Root cause: **stale stats** → wrong row estimates → wrong join choice.

BAD query

```
SELECT e.entity_id, e.entity_name, f.filing_id, f.due_date
FROM dbo.Entities e
JOIN dbo.ComplianceFilings f
  ON f.entity_id = e.entity_id
WHERE e.country_code = 'IN'
  AND f.due_date < DATEADD(day, 30, GETDATE());
```

BAD plan pattern

- Huge mismatch between **Estimated vs Actual rows**
- Optimizer may pick:
    - Nested Loops + repeated lookups (bad if result set large)
    - Or Hash Join with huge memory grant/spills (if estimate is wrong)

Fix A — Update stats (fastest “proof” fix)

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

Fix B — Add/adjust index to match predicates

```
CREATE INDEX IX_Entities_Country
ON dbo.Entities (country_code)
INCLUDE (entity_name);
```

```
GO

CREATE INDEX IX_Filings_Entity_DueDate
ON dbo.ComplianceFilings (entity_id, due_date)
INCLUDE (filing_id);
GO
```

What to teach DBAs

- If estimates are wrong → plans are unstable
- Stats freshness is *often* the first lever before rewriting everything

---

Use Case 3 — Scan vs Seek due to Non-SARGable Predicate (CPU + logical reads spike)

Scenario

A support query is written like this:

- It "looks right"
- But causes **table/index scan** and heavy logical reads

BAD query (non-sargable)

```
SELECT f.filing_id, f.entity_id, f.due_date
FROM dbo.ComplianceFilings f
WHERE CONVERT(date, f.due_date) = CONVERT(date, GETDATE());
```

BAD plan pattern

- **Index Scan** even if you have an index on due_date
- High logical_reads

GOOD rewrite (sargable range predicate)

```
DECLARE @d date = CONVERT(date, GETDATE());

SELECT f.filing_id, f.entity_id, f.due_date
FROM dbo.ComplianceFilings f
WHERE f.due_date >= @d
  AND f.due_date < DATEADD(day, 1, @d);
```

Supporting index (if needed)

```
CREATE INDEX IX_Filings_DueDate
ON dbo.ComplianceFilings (due_date)
INCLUDE (entity_id, filing_id);
GO
```

What to check in plans

- BAD: Scan + residual predicate
- GOOD: Seek + predicate pushdown + lower logical reads

Use Case 4 — Join Order Problem (filter applied too late → huge work)

Scenario

A report joins big tables then filters afterwards.

Result: optimizer may process far more rows than needed.

BAD query (filter applied late / shape not helping optimizer)

```
SELECT e.entity_name, f.filing_id, f.due_date
FROM dbo.Entities e
JOIN dbo.ComplianceFilings f
  ON f.entity_id = e.entity_id
WHERE f.filing_status IN ('Pending','Overdue')
  AND e.entity_name LIKE '%TECH%';
```

BAD plan pattern

- Large intermediate row sets
- Hash join + large memory grant or spills
- High elapsed time

GOOD rewrite (reduce rows early)

```
;WITH FilteredEntities AS (
  SELECT entity_id, entity_name
  FROM dbo.Entities
  WHERE entity_name LIKE '%TECH%'
),
FilteredFilings AS (
  SELECT filing_id, entity_id, due_date, filing_status
  FROM dbo.ComplianceFilings
  WHERE filing_status IN ('Pending','Overdue')
)
SELECT e.entity_name, f.filing_id, f.due_date
FROM FilteredEntities e
JOIN FilteredFilings f
  ON f.entity_id = e.entity_id;
```

Index support (usually required)

```
CREATE INDEX IX_Entities_Name
ON dbo.Entities (entity_name)
INCLUDE (entity_id);
GO

CREATE INDEX IX_Filings_Status_Entity
ON dbo.ComplianceFilings (filing_status, entity_id)
INCLUDE (due_date, filing_id);
GO
```

Teaching point

- “SQL is declarative” but **query shape influences cardinality + join order**
- Help optimizer by filtering early + indexing filter columns

---

Use Case 5 — Plan Regression after Deployment (Use Query Store to force “good plan”)

Scenario

After a deployment:

- Query becomes slow
- Plan changed
- DBAs need **fast rollback without code change**

Step 1 — Ensure Query Store is ON

```
ALTER DATABASE CURRENT SET QUERY_STORE = ON;
ALTER DATABASE CURRENT SET QUERY_STORE (OPERATION_MODE = READ_WRITE);
GO
```

Step 2 — Identify top resource queries (Query Store view)

```
SELECT TOP (20)
  qsqt.query_sql_text,
  qsp.plan_id,
  rs.avg_duration,
  rs.avg_cpu_time,
  rs.count_executions
FROM sys.query_store_runtime_stats rs
JOIN sys.query_store_plan qsp
  ON rs.plan_id = qsp.plan_id
JOIN sys.query_store_query qsq
  ON qsp.query_id = qsq.query_id
JOIN sys.query_store_query_text qsqt
  ON qsq.query_text_id = qsqt.query_text_id
ORDER BY rs.avg_duration DESC;
```

Step 3 — Force the known good plan

```
EXEC sys.sp_query_store_force_plan @query_id = 123, @plan_id = 456;
```

Step 4 — Validate improvement

- Compare runtime stats for forced plan
- Verify no new side effects
- If needed:

```
EXEC sys.sp_query_store_unforce_plan @query_id = 123, @plan_id = 456;
```

- **Hints** are “hard force” and often permanent debt
- **Query Store forcing** is safer, reversible, and auditable

---

Measuring “Bad vs Good” with sys.dm_exec_query_stats (training-friendly)

Use this to show impact before/after (CPU, reads, elapsed):

```
SELECT TOP (20)
  qs.total_worker_time / 1000 AS total_cpu_ms,
  qs.total_elapsed_time / 1000 AS total_elapsed_ms,
  qs.total_logical_reads,
  qs.execution_count,
  (qs.total_worker_time / NULLIF(qs.execution_count,0)) / 1000 AS
avg_cpu_ms,
  SUBSTRING(st.text, (qs.statement_start_offset/2)+1,
    ((CASE qs.statement_end_offset WHEN -1 THEN DATALENGTH(st.text)
      ELSE qs.statement_end_offset END - qs.statement_start_offset)/2)+1)
AS statement_text
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) st
ORDER BY qs.total_worker_time DESC;
```