# Steps to Identify and Mitigate High CPU SQL Server Usage Queries

## Problem Summary

- SQL Server can consume high CPU, leading to poor performance.
- Symptoms include:
    - Slow simple queries
    - Lagging reports
    - CPU process (sqlservr.exe) near 100%
- High CPU is commonly related to *query patterns*, *missing indexes*, or *out-of-date statistics*.

## Causes of High CPU Usage

Common contributors:

- High logical reads from missing indexes or stale statistics
- Sudden workload spike
- Concurrency
- Parallel query execution
- Long-running queries
- Heavy calculations
- Sorts, Temp Tables
- Other applications competing for CPU
- Compression tasks, backups, etc.

## Simulating CPU Load

To generate CPU load for demonstration (using SQLQueryStress):

```
SELECT TOP (1000)
   [WorkOrderID], [ProductID], [OrderQty], [StockedQty],
   [ScrappedQty], [StartDate], [EndDate], [DueDate],
   [ScrapReasonID], [ModifiedDate]
FROM [AdventureWorks2022].[Production].[WorkOrder]
WHERE OrderQty = 5;
GO

SELECT TOP (1000)
   [BusinessEntityID], [PersonType], [NameStyle], [Title],
   [FirstName], [MiddleName], [LastName], [Suffix],
   [EmailPromotion], [AdditionalContactInfo], [Demographics],
   [rowguid], [ModifiedDate]
FROM [AdventureWorks2022].[Person].[Person]
WHERE FirstName = 'ken';
GO
```

*Used to stress the server CPU.*

## Checking Overall CPU Usage

- Task Manager can show overall and SQL Server CPU utilization.
- If SQL Server engine process is using large percentage of CPU, proceed with deeper analysis.

**Identify Top CPU-Consuming Queries**

Use DMVs to find queries consuming the most CPU:

```
SELECT TOP 10
  COALESCE(DB_NAME(st.dbid),DB_NAME(CONVERT(INT, qp.dbid))) AS [DatabaseName],
  qs.creation_time AS [PlanCreationTime],
  qs.last_execution_time AS [LastExecutedTime],
  qs.execution_count AS [ExecutionCount],
  qs.total_worker_time / 1000 AS [Total_CPU_Time_ms],
  (qs.total_worker_time / qs.execution_count) / 1000 AS [Avg_CPU_Time_ms],
  (qs.total_worker_time / 1000) AS [Cumulative_CPU_Time_All_Executions_ms],
  (qs.total_logical_reads + qs.total_logical_writes) AS [TotalLogicalIO],
  (qs.total_logical_reads + qs.total_logical_writes) / qs.execution_count AS [AvgLogicalIO],
  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 [QueryText],
   qp.query_plan AS [ExecutionPlan]
FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) AS qp
WHERE COALESCE(DB_NAME(st.dbid),DB_NAME(CONVERT(INT, qp.dbid)))
    NOT IN ('master','model','msdb','tempdb','DBADashDB')
ORDER BY qs.total_worker_time DESC;
```

*Finds top CPU-heavy queries.*

**Queries with Missing Indexes**

Missing indexes can cause scans instead of seeks, increasing CPU:

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/showplan' AS p)
SELECT
   COALESCE(DB_NAME(st.dbid), DB_NAME(CONVERT(INT, qp.dbid))) AS DatabaseName,
   qs.creation_time AS PlanCreationTime,
   qs.last_execution_time AS LastExecutedTime,
   qs.execution_count AS ExecutionCount,
   qs.total_worker_time / 1000 AS Total_CPU_Time_ms,
   (qs.total_worker_time / qs.execution_count) / 1000 AS Avg_CPU_Time_ms,
   (qs.total_worker_time / 1000) AS Cumulative_CPU_Time_All_Executions_ms,
   (qs.total_logical_reads + qs.total_logical_writes) AS TotalLogicalIO,
   (qs.total_logical_reads + qs.total_logical_writes) / qs.execution_count AS AvgLogicalIO,
   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 QueryText,
   qp.query_plan AS ExecutionPlan
```

```
FROM (
   SELECT TOP 10 *
   FROM sys.dm_exec_query_stats
   ORDER BY total_worker_time DESC
) AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) AS qp
WHERE qp.query_plan.value('count(//p:MissingIndexGroup)', 'int') > 0;
GO
```

*Lists top CPU queries with missing index hints in execution plans.*

**Script to Generate Missing Index Create Statements**

```
SELECT TOP 25
  DB_NAME(dm_mid.database_id) AS DatabaseID,
  dm_migs.avg_user_impact*(dm_migs.user_seeks+dm_migs.user_scans) Avg_Estimated_Impact,
  dm_migs.last_user_seek AS Last_User_Seek,
  OBJECT_NAME(dm_mid.OBJECT_ID,dm_mid.database_id) AS [TableName],
  'CREATE INDEX [IX_'
   + OBJECT_NAME(dm_mid.OBJECT_ID,dm_mid.database_id)
   + '_' + REPLACE(REPLACE(REPLACE(ISNULL(dm_mid.equality_columns,''),', ','_'),'[',''),']','')
   + CASE WHEN dm_mid.equality_columns IS NOT NULL
      AND dm_mid.inequality_columns IS NOT NULL THEN '_' ELSE '' END
   + REPLACE(REPLACE(REPLACE(ISNULL(dm_mid.inequality_columns,''),', ','_'),'[',''),']','')
   + '] ON ' + dm_mid.statement
   + ' (' + ISNULL(dm_mid.equality_columns,'')
   + CASE WHEN dm_mid.equality_columns IS NOT NULL
        AND dm_mid.inequality_columns IS NOT NULL THEN ',' ELSE '' END
   + ISNULL(dm_mid.inequality_columns,'')
   + ')'
   + ISNULL(' INCLUDE (' + dm_mid.included_columns + ')', '') AS Create_Statement
FROM sys.dm_db_missing_index_groups dm_mig
JOIN sys.dm_db_missing_index_group_stats dm_migs
   ON dm_migs.group_handle = dm_mig.index_group_handle
JOIN sys.dm_db_missing_index_details dm_mid
   ON dm_mig.index_handle = dm_mid.index_handle
WHERE dm_mid.database_ID = DB_ID()
ORDER BY Avg_Estimated_Impact DESC;
GO
```

*Builds recommended index scripts based on missing index DMVs.*

**Important:** Before creating indexes, evaluate impact — unnecessary or redundant indexes can degrade performance.

### Finding Out-of-Date Statistics

Outdated statistics can lead to poor query plans:

```
SELECT DISTINCT
   OBJECT_NAME(s.[object_id]) AS TableName,
   c.name AS ColumnName,
   s.name AS StatName,
   STATS_DATE(s.[object_id], s.stats_id) AS LastUpdated,
   DATEDIFF(d, STATS_DATE(s.[object_id], s.stats_id), GETDATE()) AS DaysOld,
   dsp.modification_counter,
   s.auto_created,
   s.user_created,
   s.no_recompute,
   s.[object_id],
   s.stats_id,
   sc.stats_column_id,
   sc.column_id
FROM sys.stats s
JOIN sys.stats_columns sc
    ON sc.[object_id] = s.[object_id]
   AND sc.stats_id = s.stats_id
JOIN sys.columns c
    ON c.[object_id] = sc.[object_id]
   AND c.column_id = sc.column_id
JOIN sys.partitions par
    ON par.[object_id] = s.[object_id]
JOIN sys.objects obj
    ON par.[object_id] = obj.[object_id]
CROSS APPLY sys.dm_db_stats_properties(sc.[object_id], s.stats_id) AS dsp
WHERE OBJECTPROPERTY(s.OBJECT_ID,'IsUserTable') = 1
  AND (s.auto_created = 1 OR s.user_created = 1)
ORDER BY DaysOld;
GO
```

*Shows age of statistics per object/column.*

### Update Statistics

If outdated, update:

```
EXEC sp_updatestats;
```

*Refreshes statistics across user tables.*

⚠ This can force recompiles and temporarily increase load.

**Step-by-step operational playbook** we can actually use in production to **diagnose, analyze, and fix High CPU issues in SQL Server**, based on real-world DBA practice.

**SQL Server High CPU – Step-by-Step Troubleshooting Playbook**

**STEP 1: Confirm SQL Server Is the CPU Culprit**

**OS Level**

- Open **Task Manager**
- Check:
  - Overall CPU %
  - sqlservr.exe CPU usage
- If SQL Server is consuming most CPU → proceed

**SQL Level (optional quick check)**

```
SELECT
  cpu_count,
  scheduler_count,
  hyperthread_ratio
FROM sys.dm_os_sys_info;
```

**STEP 2: Identify Top CPU-Consuming Queries (MOST IMPORTANT STEP)**

**Run this during high CPU**

```
SELECT TOP 10
  COALESCE(DB_NAME(st.dbid),DB_NAME(CONVERT(INT, qp.dbid))) AS DatabaseName,
  qs.execution_count,
  qs.total_worker_time / 1000 AS Total_CPU_ms,
  (qs.total_worker_time / qs.execution_count) / 1000 AS Avg_CPU_ms,
  qs.total_logical_reads,
  qs.last_execution_time,
  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 QueryText,
  qp.query_plan
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY qs.total_worker_time DESC;
```

**How to interpret**

- **High Total CPU** → long-running / heavy queries
- **High Avg CPU** → inefficient logic
- **High execution_count + medium CPU** → “death by 1000 cuts”

**Action:** Copy top 3 queries and focus only on those

### STEP 3: Check for Missing Indexes in CPU-Heavy Queries

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/showplan' AS p)
SELECT
    qs.total_worker_time / 1000 AS Total_CPU_ms,
    qs.execution_count,
    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 QueryText,
    qp.query_plan
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
WHERE qp.query_plan.value('count(//p:MissingIndexGroup)', 'int') > 0
ORDER BY qs.total_worker_time DESC;
```

### What to look for

- MissingIndexGroup in execution plan
- Table scans instead of index seeks

### STEP 4: Generate Missing Index Scripts (DO NOT AUTO-CREATE)

```
SELECT TOP 25
   DB_NAME(dm_mid.database_id) AS DatabaseName,
   dm_migs.avg_user_impact*(dm_migs.user_seeks+dm_migs.user_scans) AS Impact,
   OBJECT_NAME(dm_mid.OBJECT_ID,dm_mid.database_id) AS TableName,
   'CREATE INDEX IX_'
     + OBJECT_NAME(dm_mid.OBJECT_ID,dm_mid.database_id)
     + ' ON ' + dm_mid.statement
     + ' (' + ISNULL(dm_mid.equality_columns,'')
     + CASE WHEN dm_mid.equality_columns IS NOT NULL
           AND dm_mid.inequality_columns IS NOT NULL THEN ',' ELSE '' END
     + ISNULL(dm_mid.inequality_columns,'')
     + ')'
     + ISNULL(' INCLUDE (' + dm_mid.included_columns + ')', '') AS CreateIndexScript
FROM sys.dm_db_missing_index_groups dm_mig
JOIN sys.dm_db_missing_index_group_stats dm_migs
  ON dm_migs.group_handle = dm_mig.index_group_handle
JOIN sys.dm_db_missing_index_details dm_mid
  ON dm_mig.index_handle = dm_mid.index_handle
WHERE dm_mid.database_id = DB_ID()
ORDER BY Impact DESC;
```

### Before creating index

Check:

- Existing similar indexes
- Write workload (INSERT/UPDATE heavy?)
- Index size & maintenance impact

Never blindly create all suggested indexes

**STEP 5: Check Statistics Age & Modifications**

```
SELECT
  OBJECT_NAME(s.object_id) AS TableName,
  s.name AS StatName,
  STATS_DATE(s.object_id, s.stats_id) AS LastUpdated,
  dsp.modification_counter,
  DATEDIFF(day, STATS_DATE(s.object_id, s.stats_id), GETDATE()) AS DaysOld
FROM sys.stats s
CROSS APPLY sys.dm_db_stats_properties(s.object_id, s.stats_id) dsp
WHERE OBJECTPROPERTY(s.object_id,'IsUserTable') = 1
ORDER BY DaysOld DESC;
```

**Red flags**

- Stats **> 7–30 days old**
- High modification_counter

**STEP 6: Update Statistics (Controlled)**

**Safer (table-specific)**

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

**Broad (use carefully)**

```
EXEC sp_updatestats;
```

Expect recompiles & short CPU spike

**STEP 7: Look for Parallelism Issues**

**Check CXPACKET / CXCONSUMER waits**

```
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'CX%';
```

**Check MAXDOP**

```
SELECT value_in_use
FROM sys.configurations
WHERE name = 'max degree of parallelism';
```

If misconfigured → CPU spikes due to excessive parallelism

**STEP 8: Query Design Red Flags (Very Common)**

Check if top queries have:

- SELECT *
- Functions in WHERE clause
- Implicit conversions
- Scalar UDFs
- Large sorts (ORDER BY)
- Cartesian joins
- Parameter sniffing symptoms

**STEP 9: Use Query Store (If Enabled)**

```
SELECT *
FROM sys.database_query_store_options;
```

If ON:

- Identify **regressed plans**
- Force good plan
- Compare CPU usage historically

**STEP 10: Validate Fix**

After changes:

1. Re-run **Top CPU query DMV**
2. Compare:
   - Total CPU
   - Avg CPU
   - Logical reads
3. Monitor CPU trend for 24 hours

**DBA Rule of Thumb**

| **Symptom** | **Likely Fix** |
|---|---|
| High CPU + scans | Missing index |
| High CPU + low reads | Bad logic |
| Sudden CPU spike | Stats or plan change |
| CPU only during peak | Concurrency / parallelism |
| CPU steady always | Inefficient design |

## SQL Server High CPU – COMPLETE END-TO-END GUIDE

*(Troubleshooting • Execution Plans • Query Tuning • Emergency Actions • Production Checklist)*

### PART 1: IMMEDIATE "CPU 100%" EMERGENCY ACTIONS

**When users are screaming and CPU is pegged**

**1) Confirm SQL Server is the cause**

- Task Manager → sqlservr.exe high CPU
- If not SQL Server → stop here

**2) Identify what is running RIGHT NOW**

```
SELECT
   r.session_id,
   r.status,
   r.cpu_time,
   r.total_elapsed_time,
   r.logical_reads,
   SUBSTRING(t.text, r.statement_start_offset/2,
      (CASE WHEN r.statement_end_offset = -1
          THEN LEN(t.text)
          ELSE r.statement_end_offset END - r.statement_start_offset)/2) AS RunningQuery
FROM sys.dm_exec_requests r
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) t
ORDER BY r.cpu_time DESC;
```

If **one query dominates CPU**, that's your target.

**3) Emergency kill (LAST RESORT)**

```
KILL <session_id>;
```

Use only if business impact is severe.

### PART 2: FIND TOP CPU QUERIES (HISTORICAL)

```
SELECT TOP 10
  DB_NAME(st.dbid) AS DatabaseName,
  qs.execution_count,
  qs.total_worker_time / 1000 AS Total_CPU_ms,
  (qs.total_worker_time / qs.execution_count) / 1000 AS Avg_CPU_ms,
  qs.total_logical_reads,
  qs.last_execution_time,
  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 QueryText,
  qp.query_plan
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY qs.total_worker_time DESC;
```

**How to read results**

| Pattern | Meaning |
|---|---|
| High Total CPU | Heavy query |
| High Avg CPU | Inefficient logic |
| High executions | Chatty app |
| High reads | Missing index |

## PART 3: EXECUTION PLAN – HOW TO READ (VERY IMPORTANT)

**Focus ONLY on these operators**

| Operator | Meaning |
|---|---|
| Table Scan | No useful index |
| Index Scan | Partial index issue |
| Index Seek | GOOD |
| Hash Match | CPU-heavy joins |
| Sort | CPU + memory heavy |
| Key Lookup | Missing INCLUDE |

**Red flags**

- Scan on large table
- Sort + Hash Match together
- Missing Index warnings
- Estimated vs Actual rows mismatch

## PART 4: REAL QUERY TUNING EXAMPLE

**Bad Query**

```
SELECT *
FROM Orders
WHERE YEAR(OrderDate) = 2024;
```

**Why bad?**

- Function on column
- Forces table scan
- High CPU

✅ **Fixed Query**

```
SELECT *
FROM Orders
WHERE OrderDate >= '2024-01-01'
  AND OrderDate < '2025-01-01';
```

✅ Allows index seek

✅ CPU drops massively

## PART 5: MISSING INDEX DETECTION

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/showplan' AS p)
SELECT
```

```
    qs.total_worker_time / 1000 AS Total_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 QueryText,
    qp.query_plan
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
WHERE qp.query_plan.value('count(//p:MissingIndexGroup)', 'int') > 0
ORDER BY qs.total_worker_time DESC;
```

**PART 6: GENERATE INDEX SCRIPTS (REVIEW FIRST)**

```
SELECT TOP 25
   DB_NAME(dm_mid.database_id) AS DatabaseName,
   dm_migs.avg_user_impact*(dm_migs.user_seeks+dm_migs.user_scans) AS Impact,
   OBJECT_NAME(dm_mid.object_id, dm_mid.database_id) AS TableName,
   'CREATE INDEX IX_' + OBJECT_NAME(dm_mid.object_id, dm_mid.database_id)
   + ' ON ' + dm_mid.statement
   + ' (' + ISNULL(dm_mid.equality_columns,'')
   + CASE WHEN dm_mid.equality_columns IS NOT NULL
        AND dm_mid.inequality_columns IS NOT NULL THEN ',' ELSE '' END
   + ISNULL(dm_mid.inequality_columns,'') + ')'
   + ISNULL(' INCLUDE (' + dm_mid.included_columns + ')','') AS CreateIndexScript
FROM sys.dm_db_missing_index_details dm_mid
JOIN sys.dm_db_missing_index_groups dm_mig
  ON dm_mid.index_handle = dm_mig.index_handle
JOIN sys.dm_db_missing_index_group_stats dm_migs
  ON dm_mig.index_group_handle = dm_migs.group_handle
ORDER BY Impact DESC;
```

Never auto-create all indexes

**PART 7: STATISTICS ANALYSIS**

```
SELECT
   OBJECT_NAME(s.object_id) AS TableName,
   s.name AS StatName,
   STATS_DATE(s.object_id, s.stats_id) AS LastUpdated,
   dsp.modification_counter,
   DATEDIFF(day, STATS_DATE(s.object_id, s.stats_id), GETDATE()) AS DaysOld
FROM sys.stats s
CROSS APPLY sys.dm_db_stats_properties(s.object_id, s.stats_id) dsp
WHERE OBJECTPROPERTY(s.object_id,'IsUserTable') = 1
ORDER BY DaysOld DESC;
```

**Update stats**

```
EXEC sp_updatestats;
or
UPDATE STATISTICS dbo.TableName WITH FULLSCAN;
```

**PART 8: PARALLELISM & MAXDOP**

```
SELECT *
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'CX%';
SELECT value_in_use
FROM sys.configurations
WHERE name = 'max degree of parallelism';
```

Bad MAXDOP = CPU chaos

**PART 9: COMMON CPU KILLERS (MEMORIZE)**

- ❌ SELECT *
- ❌ Scalar UDFs
- ❌ Implicit conversions
- ❌ Cursors
- ❌ RBAR logic
- ❌ Large sorts
- ❌ Parameter sniffing

**PART 10: PRODUCTION CPU CHECKLIST (PRINT THIS)**

- ✅ Identify top CPU queries
- ✅ Check execution plan
- ✅ Fix scans → seeks
- ✅ Add **correct** indexes
- ✅ Update statistics
- ✅ Validate MAXDOP
- ✅ Retest under load
- ✅ Monitor 24 hours

**GOLDEN DBA RULE**

**90% of high CPU issues are fixed by:**

- Indexes
- Statistics
- Query rewrites

## SQL SERVER CPU & PERFORMANCE — RUNBOOK

### 1. PRODUCTION QUERY TUNING (NEED QUERY)

**What is needed:**

1. Query text
2. Actual execution plan (XML or screenshot summary)
3. Table sizes (approx rows)
4. Indexes on involved tables
5. Parameter values used during slow run

**What we have to do:**

- Identify **CPU driver operator**
- Check **row estimate vs actual**
- Rewrite predicates (SARGability)
- Remove unnecessary operators
- Design **minimal correct index**
- Validate with logical reads + CPU delta

### 2. EXECUTION PLAN DECISION TREES (DBA BRAIN MAP)

**ROOT QUESTION**

**What operator consumes the most % cost or CPU?**

### A. TABLE / INDEX SCAN

```
Scan?
 ├─ Small table (<10k rows)? → OK
 └─ Large table?
     ├─ Predicate non-SARGable? → Rewrite WHERE clause
     ├─ Missing index? → Create index
     ├─ Wrong index chosen?
         ├─ Stats outdated → Update stats
         └─ Parameter sniffing → Fix sniffing
```

### B. HASH MATCH (JOIN / AGGREGATE)

```
Hash Match?
 ├─ Large input? → Reduce rows earlier
 ├─ Missing join index? → Add index on join column
 ├─ Memory spill? → Increase memory / rewrite
 └─ Should be Nested Loop? → Fix estimates
```

### C. SORT

```
Sort?
 ├─ ORDER BY? → Index with correct key order
 ├─ DISTINCT? → Remove if unnecessary
 ├─ GROUP BY? → Covering index
 └─ Spill to tempdb? → Memory pressure
```

**D. KEY LOOKUP**

```
Key Lookup?
 ├─ High executions? → INCLUDE columns
 └─ Few executions? → Ignore
```

**E. NESTED LOOP (HIGH CPU)**

```
Nested Loop?
 ├─ Inner input large? → Missing index
 ├─ Loop count high? → Join order problem
 └─ Parameter sniffing? → Fix sniffing
```

**3. PARAMETER SNIFFING — DEEP DIVE (VERY IMPORTANT)**

**What it is**

SQL Server:

1. Compiles plan using **first parameter values**
2. Reuses that plan for **all future executions**
3. Disaster when data is skewed

**Classic Example**

```
CREATE PROC GetOrders @CustomerID INT
AS
SELECT *
FROM Orders
WHERE CustomerID = @CustomerID;
```

- CustomerID = 1 → 5 rows
- CustomerID = 999 → 5 million rows

  ⚠ Same plan used for both

**How to DETECT sniffing**

- Query fast sometimes, slow other times
- Execution plan flips between Scan/Seek
- Estimated vs Actual rows wildly different
- Clearing cache temporarily "fixes" it

```
DBCC FREEPROCCACHE; -- NEVER in prod without approval
```

**FIX OPTIONS (IN ORDER OF SAFETY)**

**Option 1: OPTIMIZE FOR UNKNOWN (BEST DEFAULT)**

```
WHERE CustomerID = @CustomerID
OPTION (OPTIMIZE FOR UNKNOWN);
```

**Option 2: RECOMPILE (CPU COST)**

```
OPTION (RECOMPILE);
```

✓ Fixes sniffing

❌ Increases compile CPU

**Option 3: Local Variable Trick**

```
DECLARE @CID INT = @CustomerID;

WHERE CustomerID = @CID;
```

**Option 4: Dynamic SQL (ADVANCED)**

Use when query shape must change based on parameter

**What NOT to do**

- ❌ Blanket WITH RECOMPILE everywhere
- ❌ Clearing plan cache regularly
- ❌ Ignoring skewed data distributions

## 4. QUERY STORE & FORCED PLANS — DEEP EXPLANATION

**What Query Store Does**

- Captures **query text**
- Stores **multiple execution plans**
- Tracks **runtime stats over time**
- Lets you **force a known-good plan**

**When to use Forced Plans**

Plan regression after:

- Index change
- Stats update
- SQL upgrade
- Parameter sniffing fix failed

Do NOT use to hide bad queries

**How to Identify Regression**

- Same query
- New plan
- Higher CPU / duration

```
SELECT *
FROM sys.query_store_runtime_stats
ORDER BY avg_cpu_time DESC;
```

**Force a Plan**

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

**What Happens When Forced**

- SQL Server **must use that plan**
- If plan becomes invalid → force fails
- SQL Server logs force failures

**Forced Plan Risks**

⚠ Schema change breaks plan

⚠ Index dropped → failure

⚠ Data distribution shifts → bad performance

Forced plans are **stabilizers**, not fixes.

**Best Practice**

1. Tune query first
2. Fix indexes & stats
3. Use forced plan **only if regression persists**

### 5. CPU INCIDENT RUNBOOK (COPY / PASTE)

**INCIDENT START**

- Confirm sqlservr.exe CPU
- Capture top CPU sessions
- Save execution plans
- Identify top operator

**ANALYSIS**

- Scan vs Seek
- Row estimate accuracy
- Missing index?
- Parameter sniffing?

**REMEDIATION**

- Rewrite query
- Add index (minimal)
- Update stats
- Adjust MAXDOP if needed

**STABILIZATION**

- Query Store verify
- Force plan if regression
- Monitor 24 hours