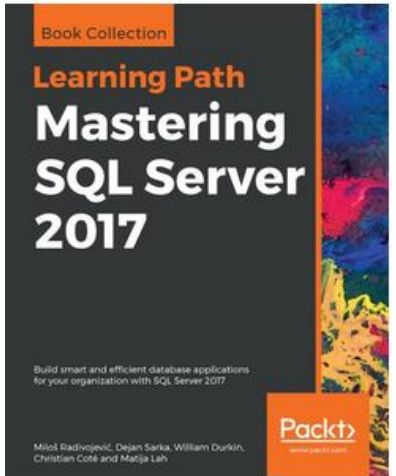
PARAMETER SNIFFING

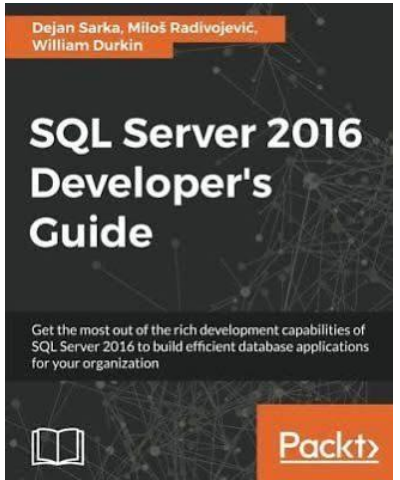
IN SQL SERVER STORED PROCEDURES

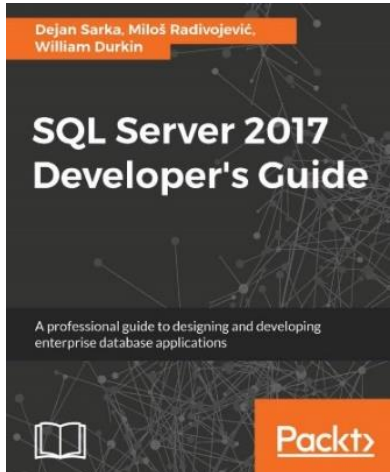
MILOŠ RADIVOJEVIĆ, MICROSOFT DATA PLATFORM MVP

Miloš Radivojević

- Data Platform MVP
- Principal Database Consultant at bwin, Vienna, Austria
- Co-Founder: SQL Pass Austria
- Conference Speaker, Book Author








DEMO

Demo – Sample Table

Orders

	Column Name	Data Type	Allow Nulls
🔑	Id	int	☐
	CustomerId	int	☐
	OrderDate	datetime	☐
	Amount	int	☐
	Other	char(500)	☑
			☐

Results | Messages

	Id	CustomerId	OrderDate	Amount	Other
1	1	47031	2017-09-13 00:00:00.000	2826	other
2	2	32779	2020-02-19 00:00:00.000	3197	other
3	3	43391	2012-11-18 00:00:00.000	7991	other
4	4	10909	2015-04-03 00:00:00.000	5998	other
5	5	33664	2018-09-05 00:00:00.000	1051	other
6	6	20319	2017-01-09 00:00:00.000	8739	other
7	7	36850	2019-05-31 00:00:00.000	2035	other
8	8	41105	2014-09-08 00:00:00.000	757	other
9	9	22042	2016-11-28 00:00:00.000	6258	other
10	10	20641	2016-03-10 00:00:00.000	8647	other

- 5M rows
- Indexes: student_id and exam_date

Demo – Requirements

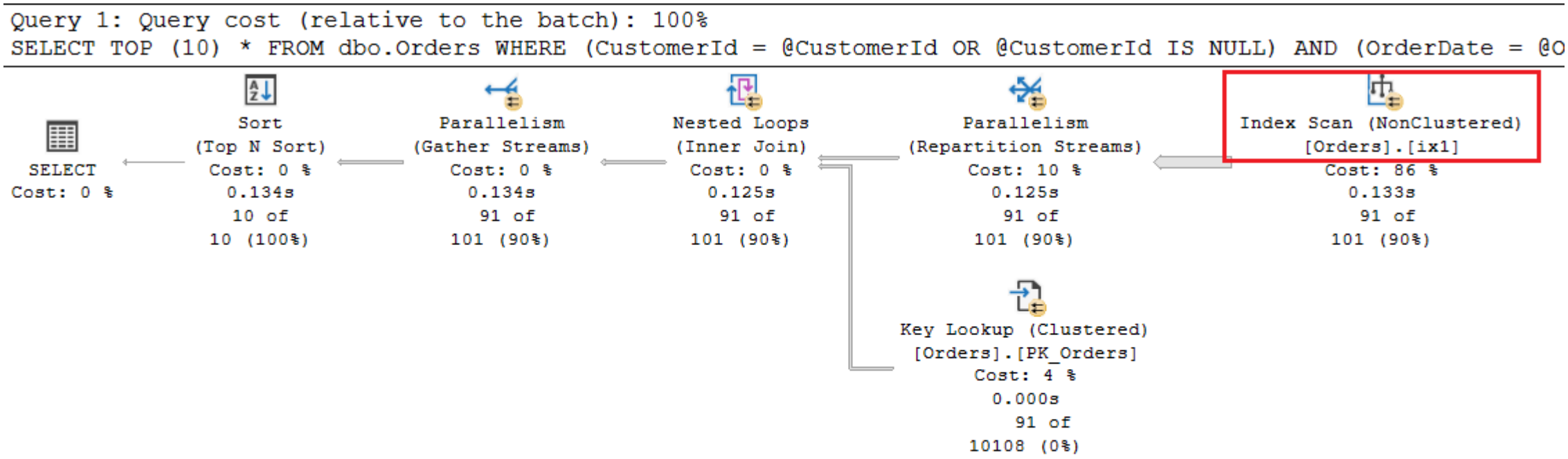
- Two input parameters: Student Id and Order Date
- Both parameters are optional
- The resulted data set should contain up to 10 orders sorted by the amount descending

Common Solution

```
CREATE OR ALTER PROCEDURE dbo.GetOrders
    @CustomerId INT = NULL, @OrderDate DATETIME = NULL
AS
BEGIN
    SELECT TOP (10) *
    FROM dbo.Orders
    WHERE
        (CustomerId = @CustomerId OR @CustomerId IS NULL)
        AND
        (OrderDate = @OrderDate OR @OrderDate IS NULL)
    ORDER BY Amount DESC
END
GO
```

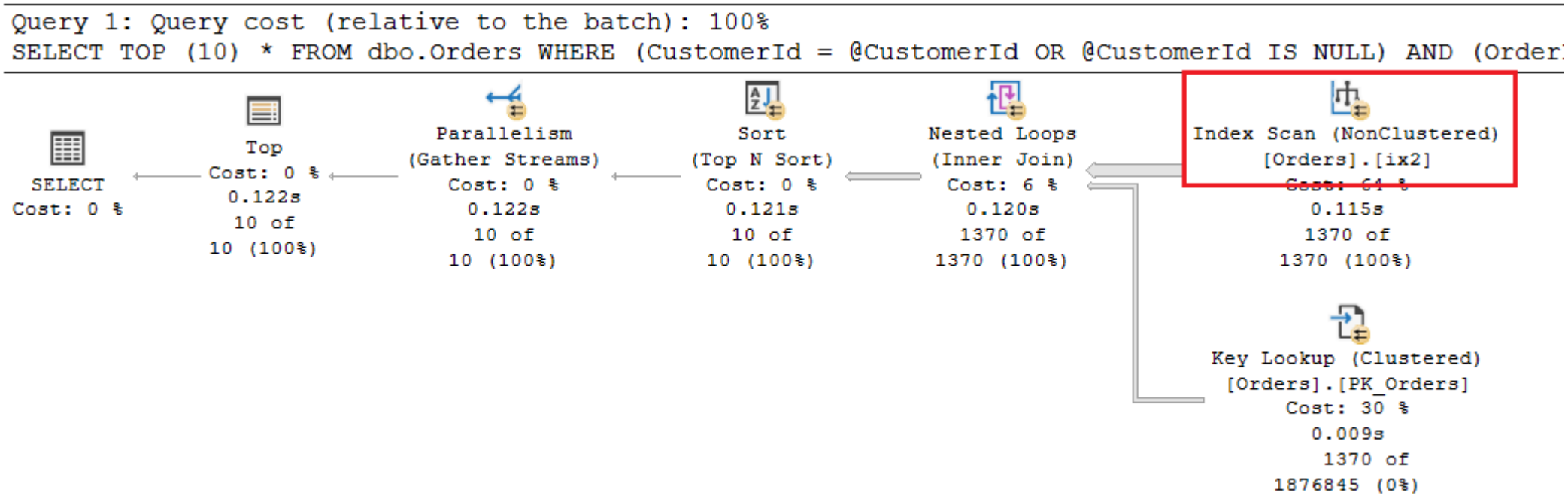
Execution Plans – Plan 1

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
EXEC dbo.GetOrders 567, NULL;
```



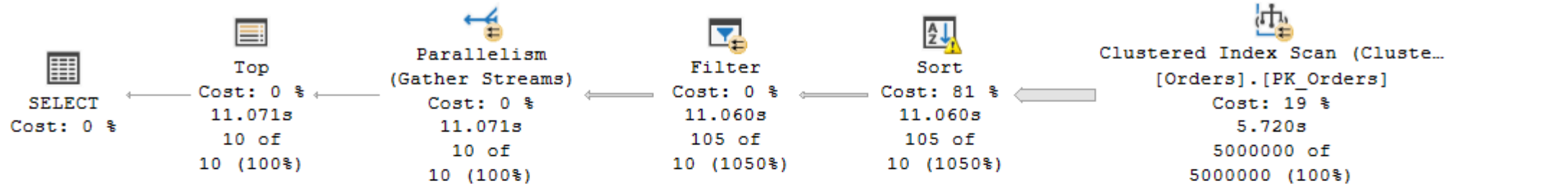
Execution Plans – Plan 2

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
EXEC dbo.GetOrders NULL,'20200401';
```

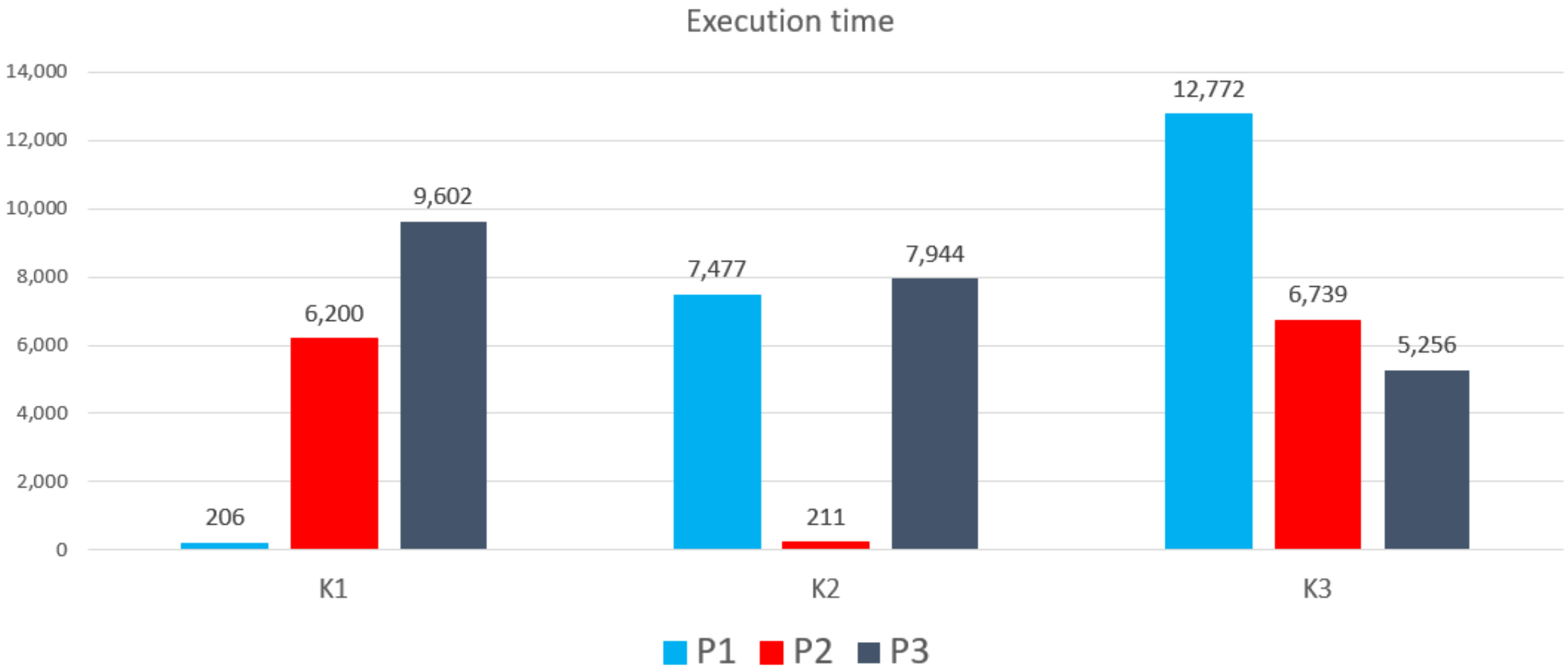


Execution Plans – Plan 3

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
EXEC dbo.GetOrders NULL, NULL;
```



Results


Execution time
14,000
12,000
10,000
8,000
6,000
4,000
2,000
0
206
6,200
9,602
7,477
211
7,944
12,772
6,739
5,256
K1
K2
K3
P1
P2
P3

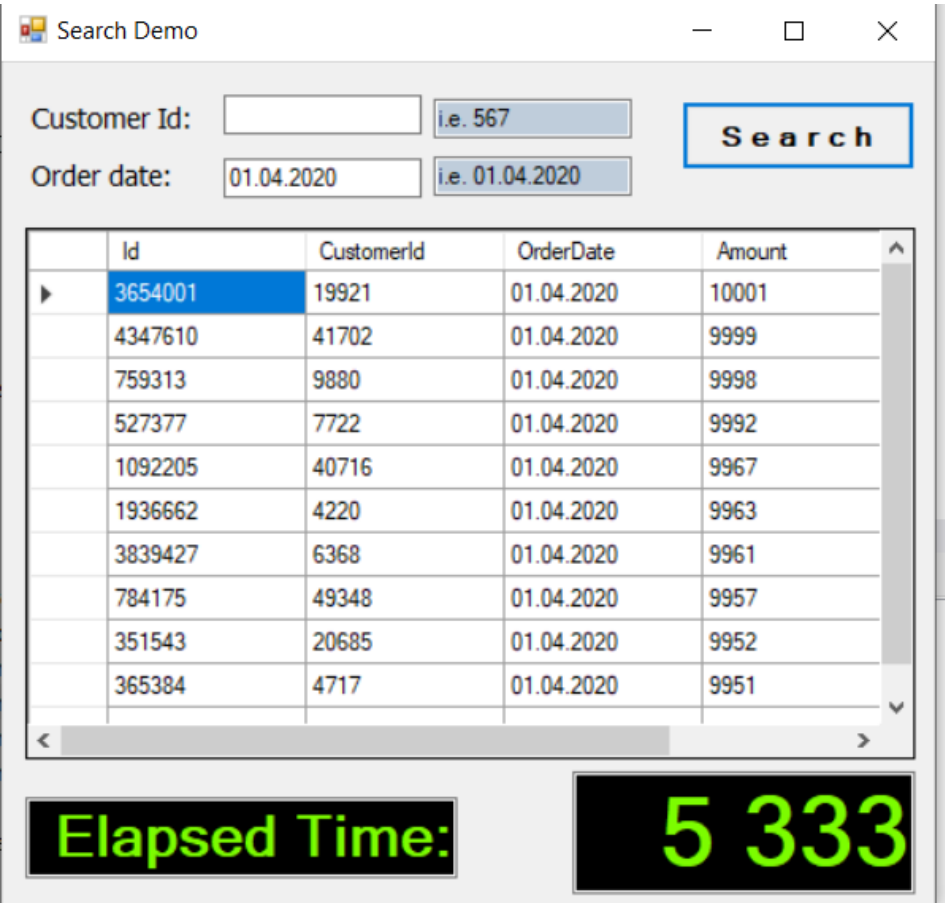
What is Parameter Sniffing?

- During the plan compilation, the values from input parameters are evaluated (sniffed) and used for cardinality estimations
- Future executions will re-use this plan
- This is a behavior, not a bug
  - It is good for invocations with similar parameters
  - It can significantly degrade the performance for very different parameters

What is Parameter Sniffing?

- When several execution plans are possible
- Stored procedures prone to parameter sniffing
  - with parameters participating in range operators
  - with optional parameters
- Not only limited to stored procedures
  - Static parameterized queries
  - Dynamic queries executed with sp_executesql

Application vs. SSMS



Id	CustomerId	OrderDate	Amount
3654001	19921	01.04.2020	10001
4347610	41702	01.04.2020	9999
759313	9880	01.04.2020	9998
527377	7722	01.04.2020	9992
1092205	40716	01.04.2020	9967
1936662	4220	01.04.2020	9963
3839427	6368	01.04.2020	9961
784175	49348	01.04.2020	9957
351543	20685	01.04.2020	9952
365384	4717	01.04.2020	9951

```
EXEC dbo.GetOrders NULL,'20200401';
```

165 %

Results | Messages

	Id	CustomerId	OrderDate	Amount	Other
1	3654001	19921	2020-04-01 00:00:00.000	10001	other
2	4347610	41702	2020-04-01 00:00:00.000	9999	other
3	759313	9880	2020-04-01 00:00:00.000	9998	other
4	527377	7722	2020-04-01 00:00:00.000	9992	other
5	1092205	40716	2020-04-01 00:00:00.000	9967	other
6	1936662	4220	2020-04-01 00:00:00.000	9963	other
7	3839427	6368	2020-04-01 00:00:00.000	9961	other
8	784175	49348	2020-04-01 00:00:00.000	9957	other
9	351543	20685	2020-04-01 00:00:00.000	9952	other
10	365384	4717	2020-04-01 00:00:00.000	9951	other

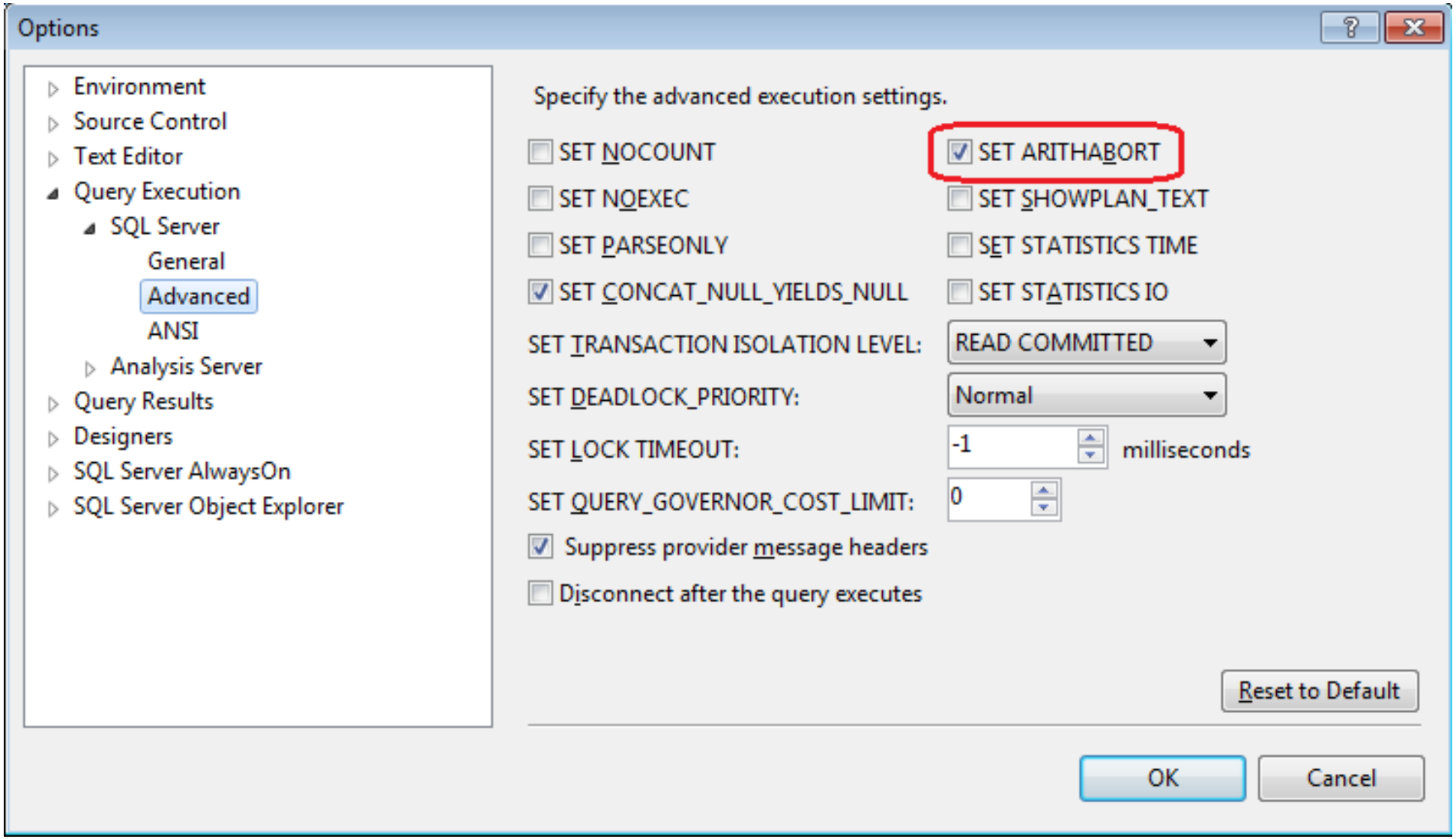
```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 596 ms,  elapsed time = 105 ms.
```

SSMS Mistery

- It works instantly in the SSMS, but it takes 5 seconds in the application
- A new execution plan is created for the stored procedure invocation within SSMS!

- Factors that affect plan-reuse
- ANSI_NULLS
- ANSI_WARNINGS
- QUOTED_IDENTIFIER
- ANSI_PADDING
- ARITHABORT
- CONCAT_NULL_YIELDS_NULL
- DATEFORMAT

SSMS Mistery



SOLUTIONS

Solution 1 – Disable Parameter Sniffing

- **Goal:** to eliminate spikes with an average execution plan
  - SQL Server ignores parameter values when it generates the execution plan

- How to implement?
  - Using the **OPTIMIZE FOR UKNOWN** query hint
  - Wrapping parameters in local variables
  - Disabling PS at the database level

```
ALTER DATABASE SCOPED CONFIGURATION SET PARAMETER_SNIFFING = OFF;
```

  - Using TF 4136

Solution 1 – OPTIMIZE FOR UNKNOWN

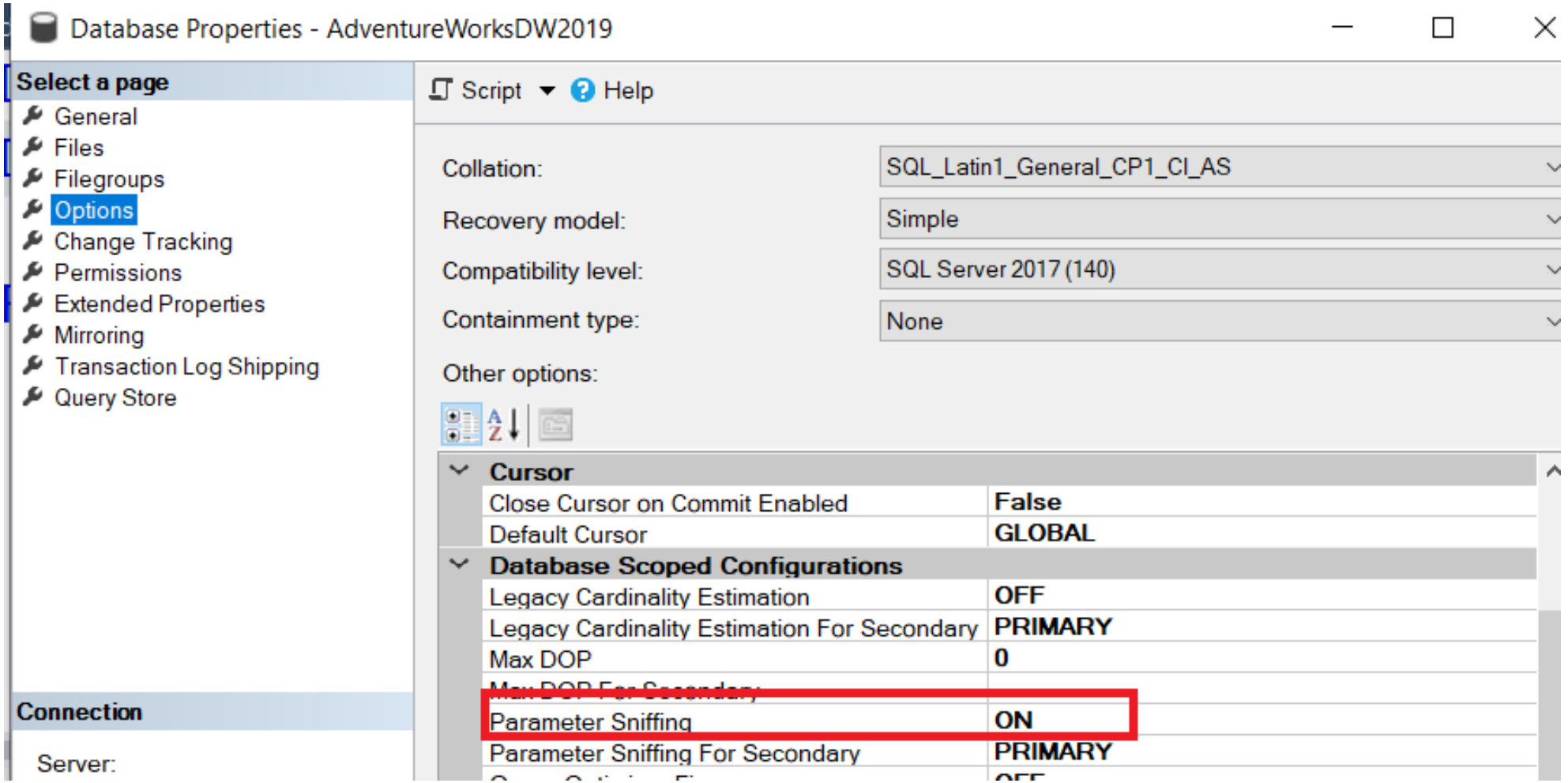
```
CREATE OR ALTER PROCEDURE dbo.GetOrders
    @CustomerId INT = NULL, @OrderDate DATETIME = NULL
AS
BEGIN
    SELECT TOP (10) * FROM dbo.Orders
    WHERE
    (CustomerId = @CustomerId OR @CustomerId IS NULL)
    AND
    (OrderDate = @OrderDate OR @OrderDate IS NULL)
    ORDER BY Amount DESC
    OPTION (OPTIMIZE FOR UNKNOWN)
END
```

Solution 1 – Local Variables

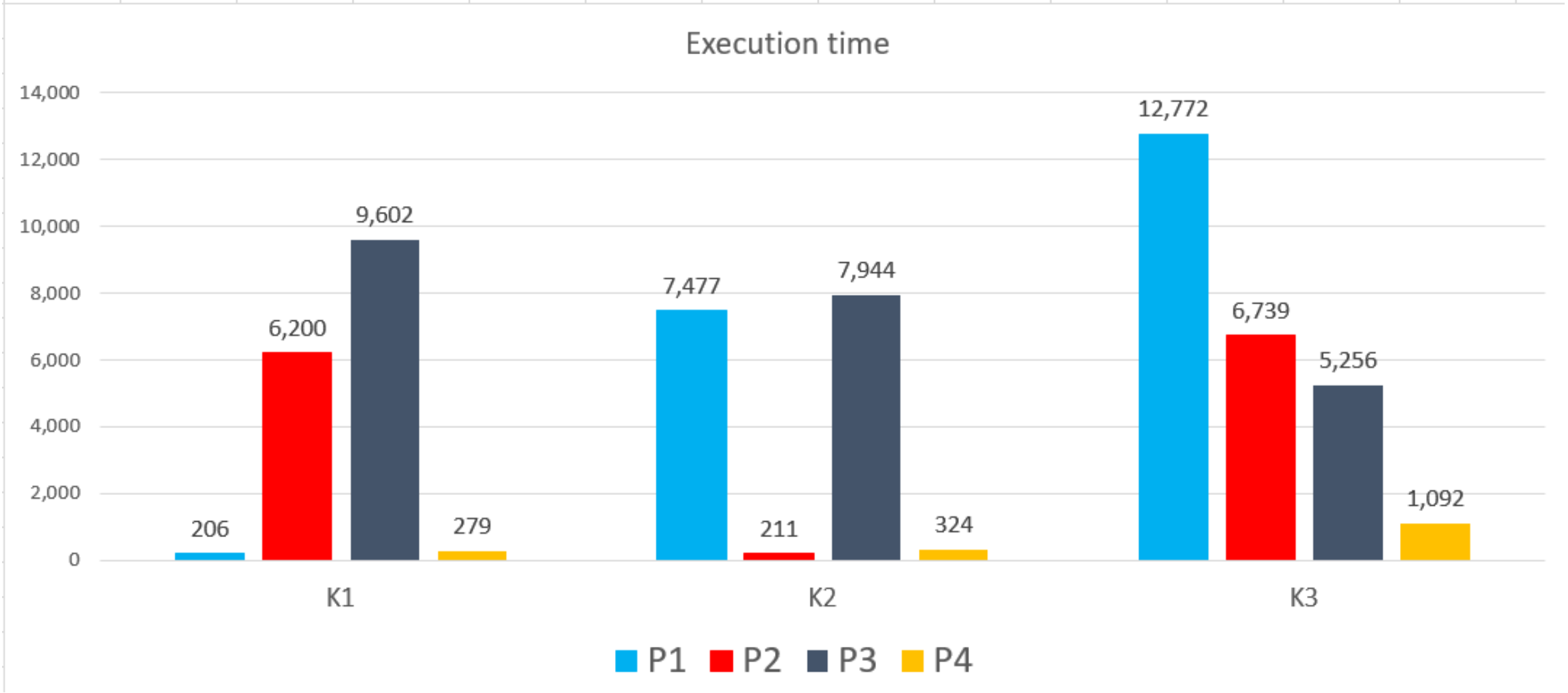
```
CREATE OR ALTER PROCEDURE dbo.GetOrders
    @CustomerId INT = NULL, @OrderDate DATETIME = NULL
AS
    DECLARE @cid INT = @CustomerId;
    DECLARE @od DATETIME = @OrderDate;
    SELECT TOP (10) * FROM dbo.Orders
    WHERE
        (CustomerId = @cid OR @cid IS NULL)
        AND
        (OrderDate = @od OR @od IS NULL)
    ORDER BY Amount DESC
```

Solution 1 – Disable Parameter Sniffing

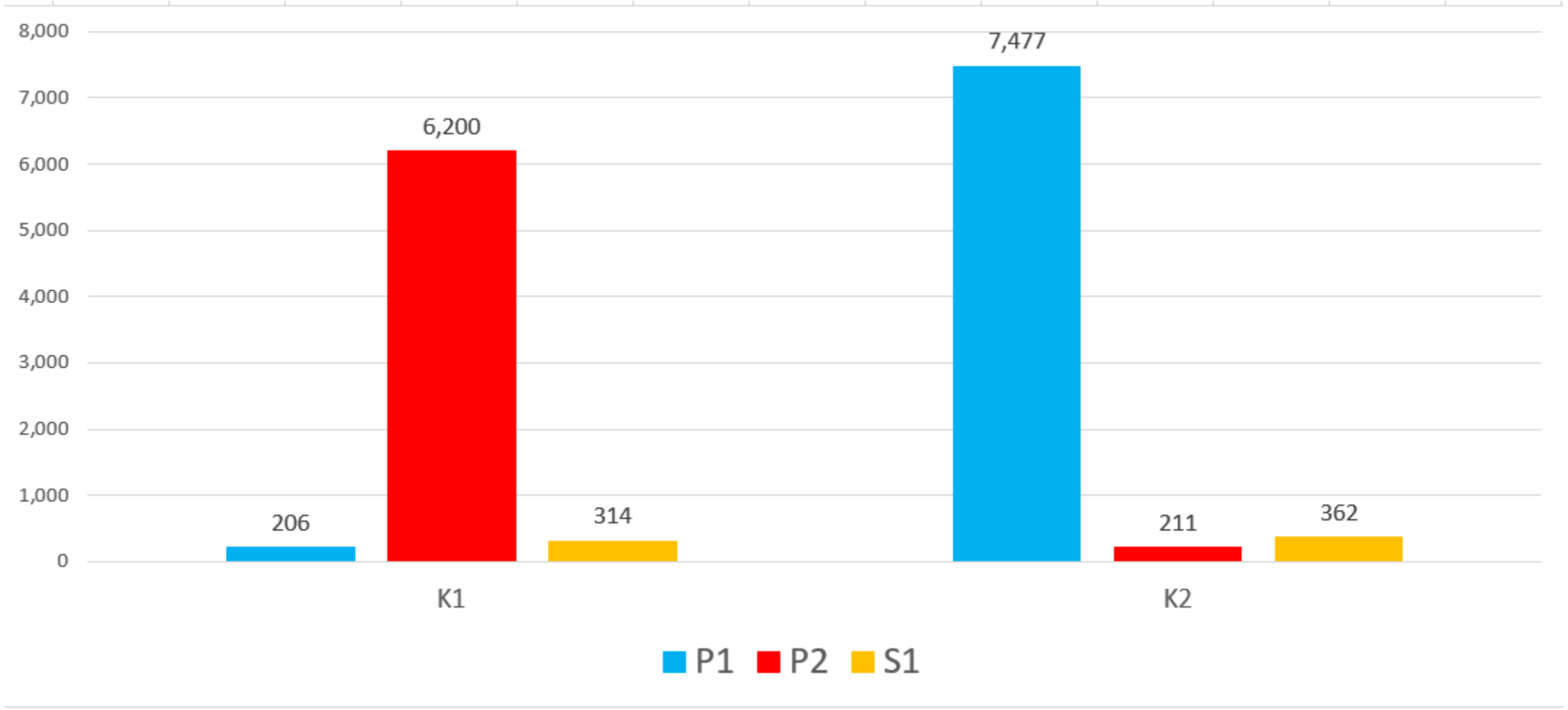
```
ALTER DATABASE SCOPED CONFIGURATION SET PARAMETER_SNIFFING = OFF;
```



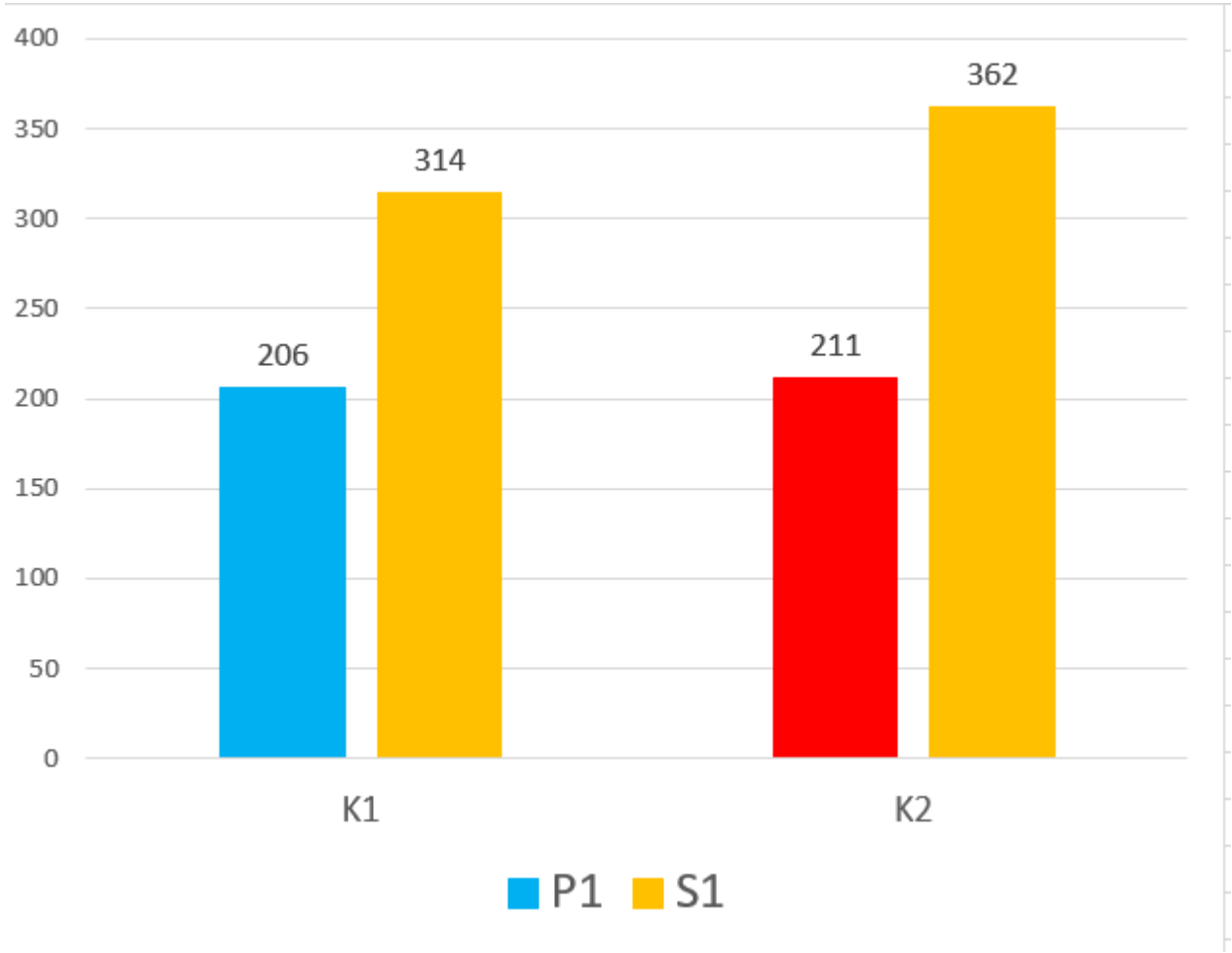
Results – Disable Parameter Sniffing



Results – Disable Parameter Sniffing



Results – Disable Parameter Sniffing



Solution 2 – Favorite Combination

- Goal: to work perfect for the most common or important combination(s)
  - Need to contact business people

- How to implement?
  - Using the OPTIMIZE FOR query hint
  - Query Decomposition (IF)

```
OPTION (OPTIMIZE FOR (@CustomerId = 750))
```

Solution 3 – Procedure Recompile

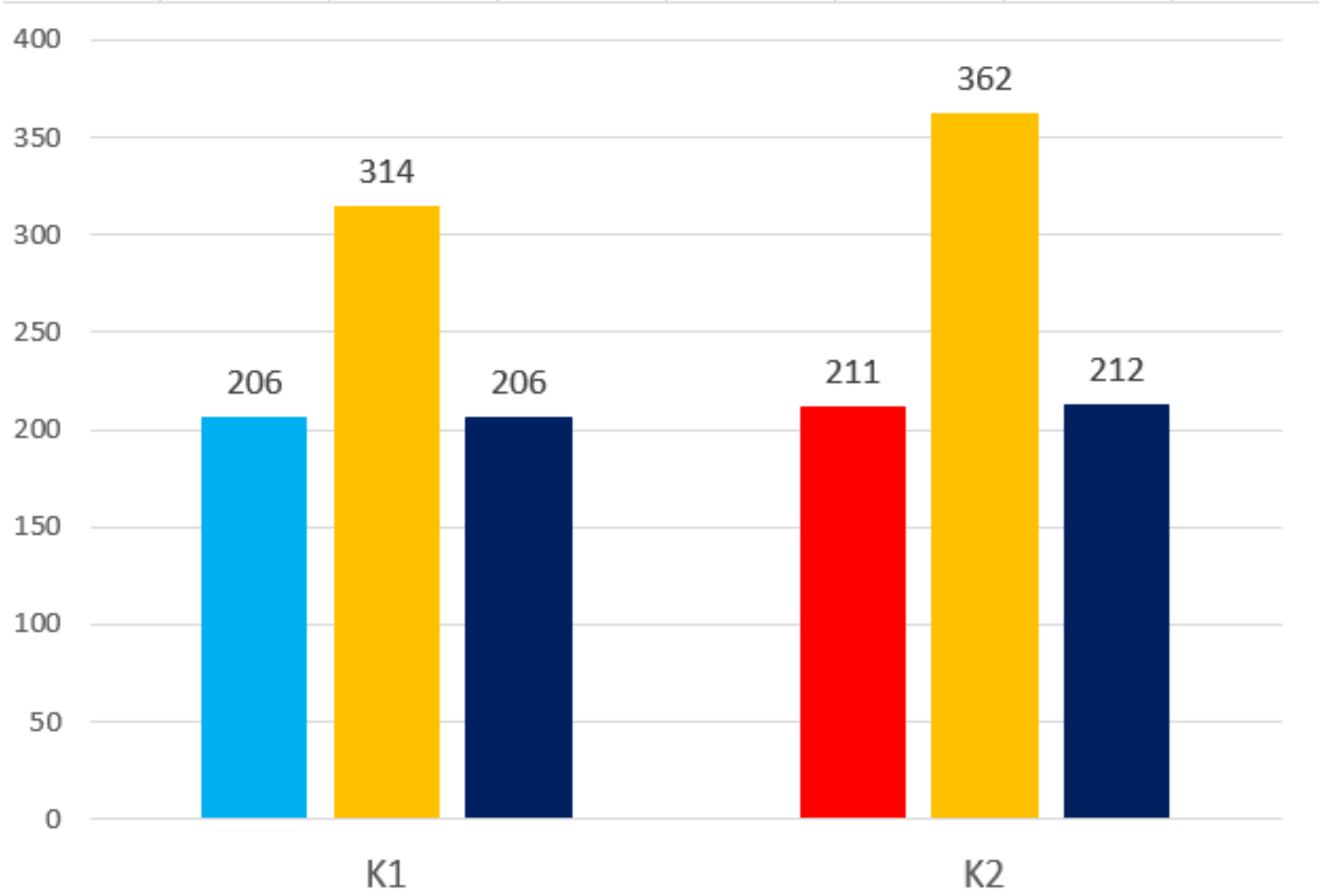
- Goal: to have the same plan as the initial one, for each parameter combination
  - Recompile the plan for each execution
- Recompile at the call level

```
EXEC dbo.GetOrders 567,NULL WITH RECOMPILE;
EXEC dbo.GetOrders NULL,'20200401' WITH RECOMPILE;
```

- Recompile at the procedure level

```
CREATE OR ALTER PROCEDURE dbo.GetOrders
@CustomerId INT = NULL, @OrderDate DATETIME = NULL
WITH RECOMPILE
AS …
```

Results – Procedure Recompile



Solution 3 – Procedure Recompile

- Pros:
  - The plan equivalent to the initial one for each execution
- Cons:
  - Compiled by each execution

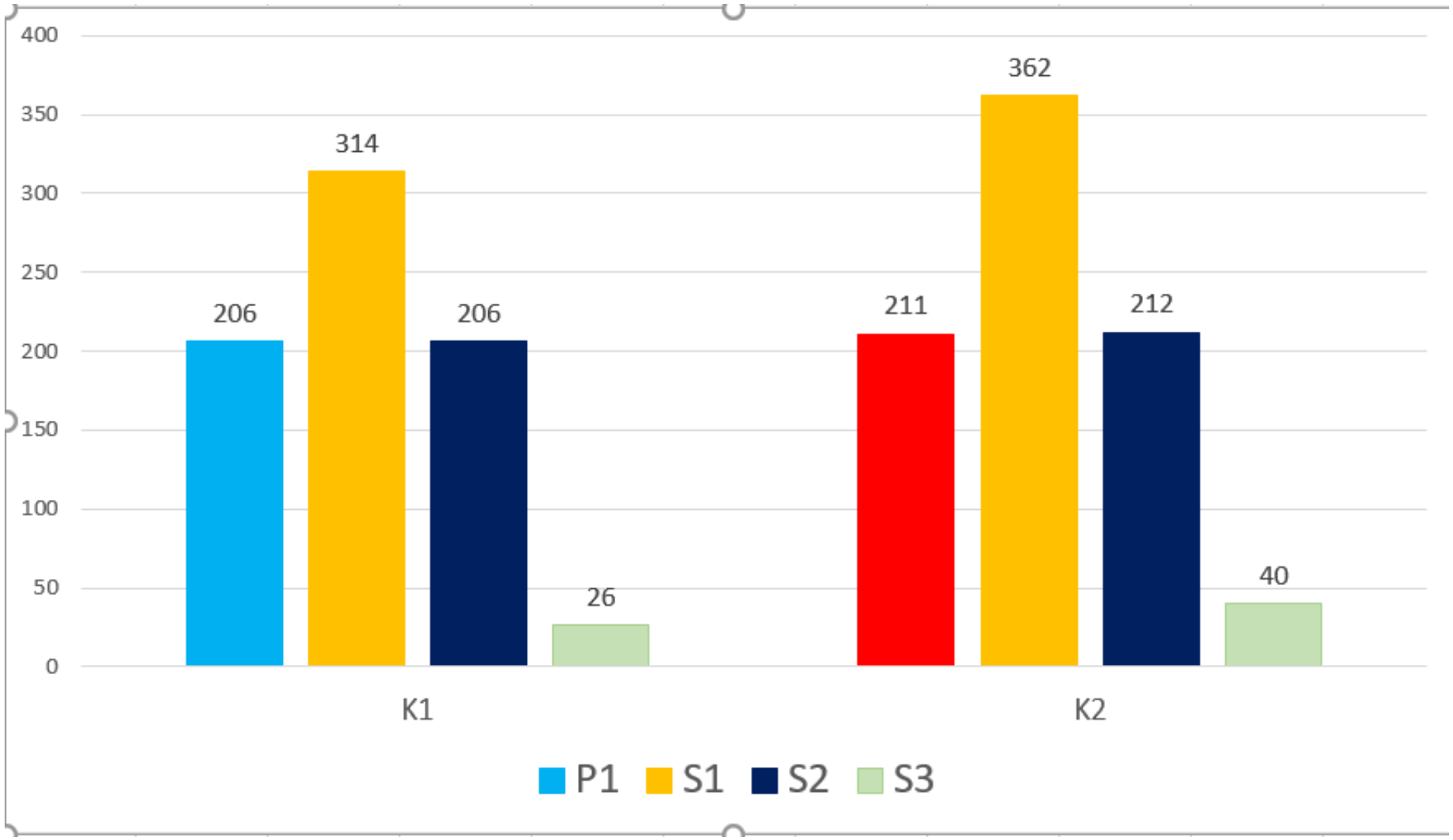
- Do not use the option WITH RECOMPILE at the sp definition

- Use EXEC dbo.GetOrders 567,NULL WITH RECOMPILE ONLY when you do not have access to the stored procedure

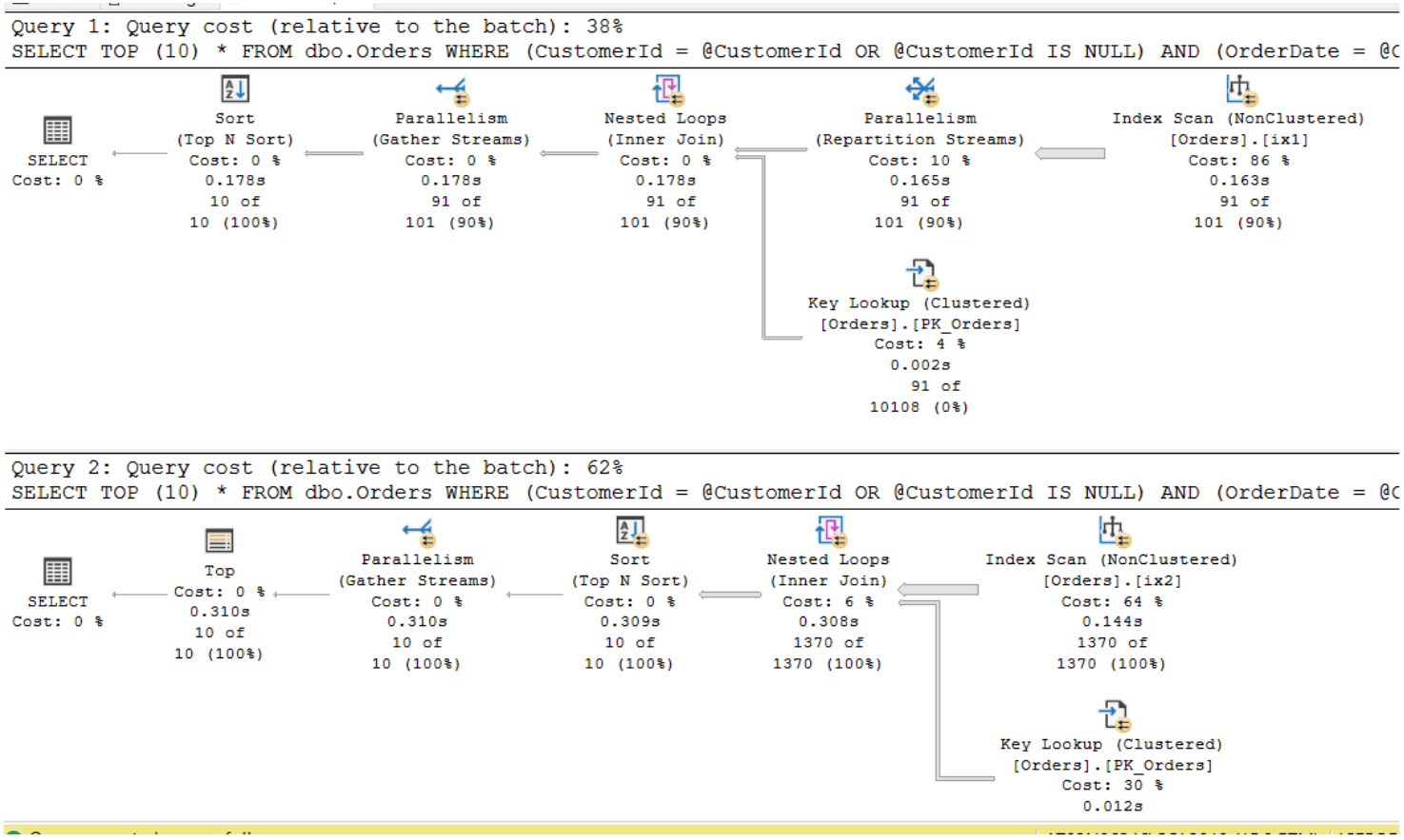
Solution 4 – OPTION (RECOMPILE)

```
CREATE OR ALTER PROCEDURE dbo.GetOrders
   @CustomerId INT = NULL, @OrderDate DATETIME = NULL
AS
BEGIN
   SELECT TOP (10) * FROM dbo.Orders
   WHERE
      (CustomerId = @CustomerId OR @CustomerId IS NULL)
      AND
      (OrderDate = @OrderDate OR @OrderDate IS NULL)
   ORDER BY Amount DESC
   OPTION (RECOMPILE)
END
```

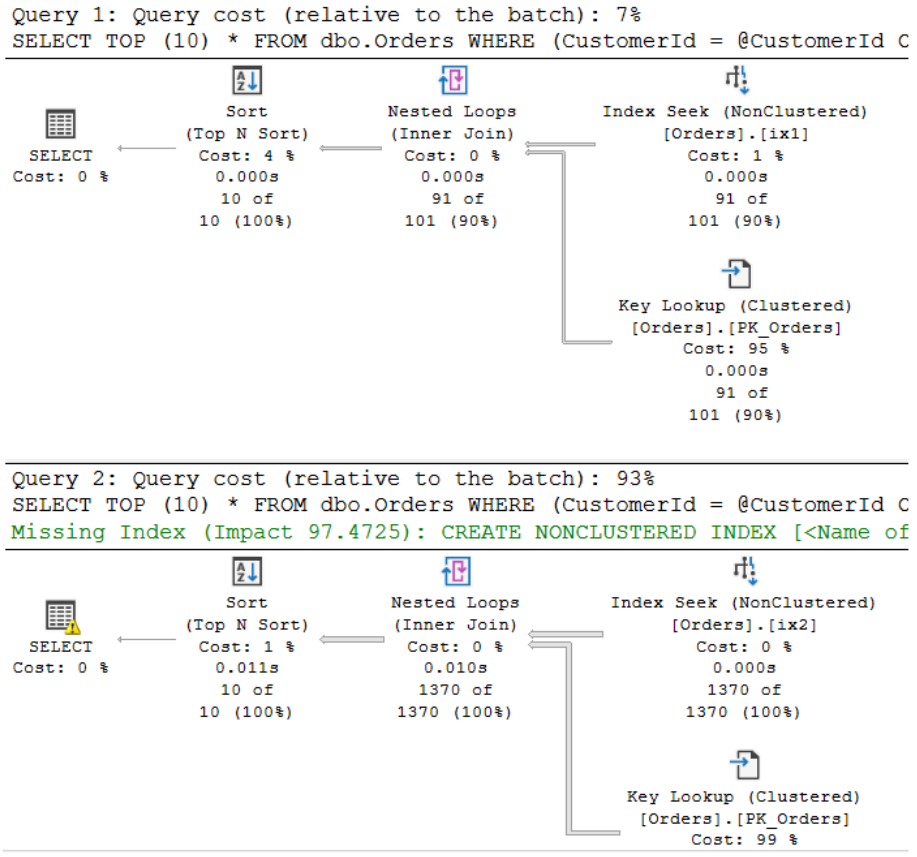
Results – OPTION (RECOMPILE)



Solution 4 – OPTION (RECOMPILE)



original



OPTION (RECOMPILE)

Solution 4 – OPTION (RECOMPILE)

- Pros:
  - The optimal plan for each execution
  - The plan can be better than the best plan in the initial solution!!!
- Cons:
  - Compiled by each execution

- In most of the cases, the best solution for the PS problem!
- BUT…. do not use it when you invoke the procedure a lot of times per second or when a query is complex (and compilation is expensive)!!!

Solution 5 – Decomposition (Decision Tree)

- **Goal:** Always get the optimal execution plan and avoid recompilation
- Pros:
  - Optimal plan for each execution
  - **Reusing the plan**
  - The plan can be better than the best plan in the initial solution!!!
- Cons:
  - Maintenance problems, SQL Injection…
- How to implement?
  - Static SQL (Decision Tree Implementation)
  - Dynamic SQL

Solution 5 – Decomposition (static)

```sql
CREATE OR ALTER PROCEDURE dbo.GetOrders1
@CustomerId INT
AS
BEGIN
        SELECT TOP (10) * FROM dbo.Orders
        WHERE CustomerId = @CustomerId
        ORDER BY Amount DESC;
END
```

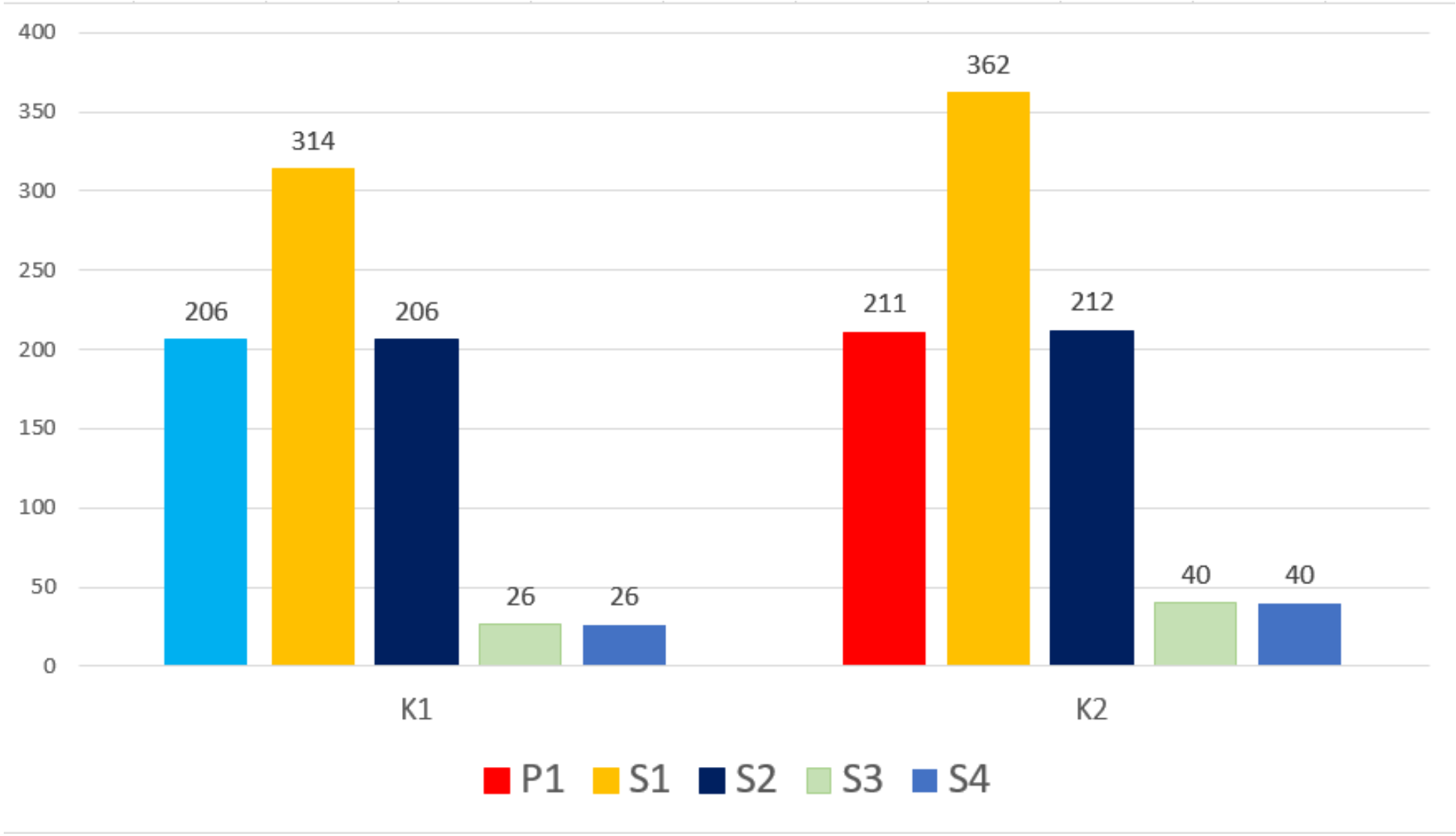
```sql
CREATE OR ALTER PROCEDURE dbo.GetOrders2
@OrderDate DATETIME
AS
BEGIN
        SELECT TOP (10) * FROM dbo.Orders
        WHERE OrderDate = @OrderDate
        ORDER BY Amount DESC;
END
```

```sql
CREATE OR ALTER PROCEDURE dbo.GetOrders3
AS
BEGIN
        SELECT TOP (10) * FROM dbo.Orders
        ORDER BY Amount DESC;
END
```

Solution 5 – Decomposition (static)

```
CREATE OR ALTER PROCEDURE dbo.GetOrders
@CustomerId INT = NULL, @OrderDate DATETIME = NULL
AS
BEGIN
   IF @CustomerId IS NOT NULL
        EXEC dbo.GetOrders1 @CustomerId;
   ELSE
   IF @OrderDate IS NOT NULL
       EXEC dbo.GetOrders2 @OrderDate;
   ;
   ELSE
       EXEC dbo.GetOrders3;
END
```

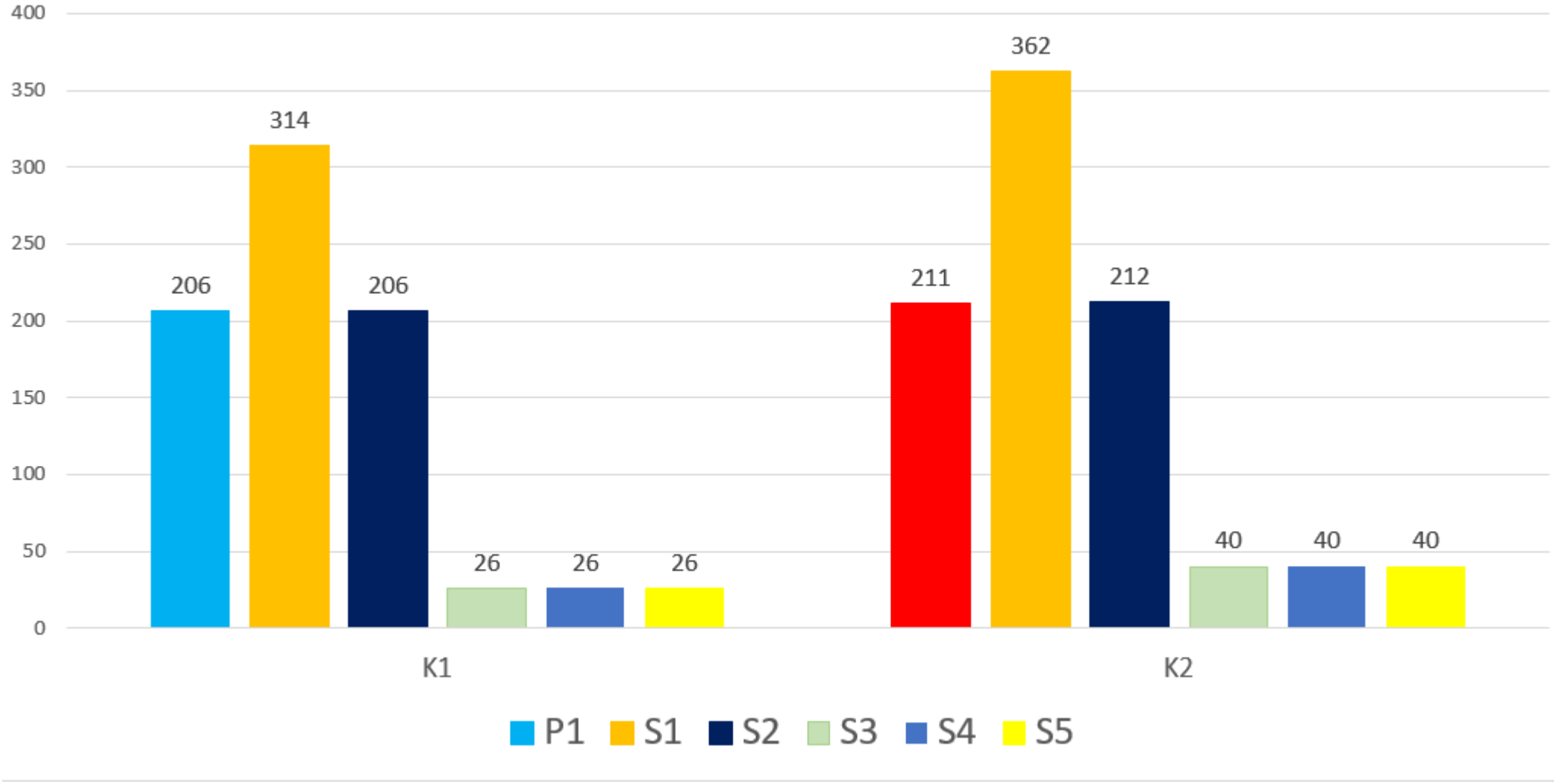
Results – Decomposition



Solution 6 – Decomposition (dynamic)

```
CREATE OR ALTER PROCEDURE dbo.GetOrders
@CustomerId INT = NULL, @OrderDate DATETIME = NULL
AS
BEGIN
    DECLARE @sql NVARCHAR(800) = N'SELECT TOP (10) * FROM dbo.Orders WHERE 1 = 1  ';
    IF @CustomerId IS NOT NULL
        SET @sql+= ' AND CustomerId = @cid ';
    IF @OrderDate IS NOT NULL
        SET @sql+= ' AND OrderDate = @od ';
        SET @sql+= ' ORDER BY Amount DESC ';

        EXEC sp_executesql @sql,  N'@cid INT, @od DATETIME',
             @cid = @CustomerId, @od = @OrderDate;
END
```

Results – Dynamic SQL



Decomposition vs. OPTION (RECOMPILE)

100 parallel sessions execute a query invoking the SP 50 times

Static decomposition

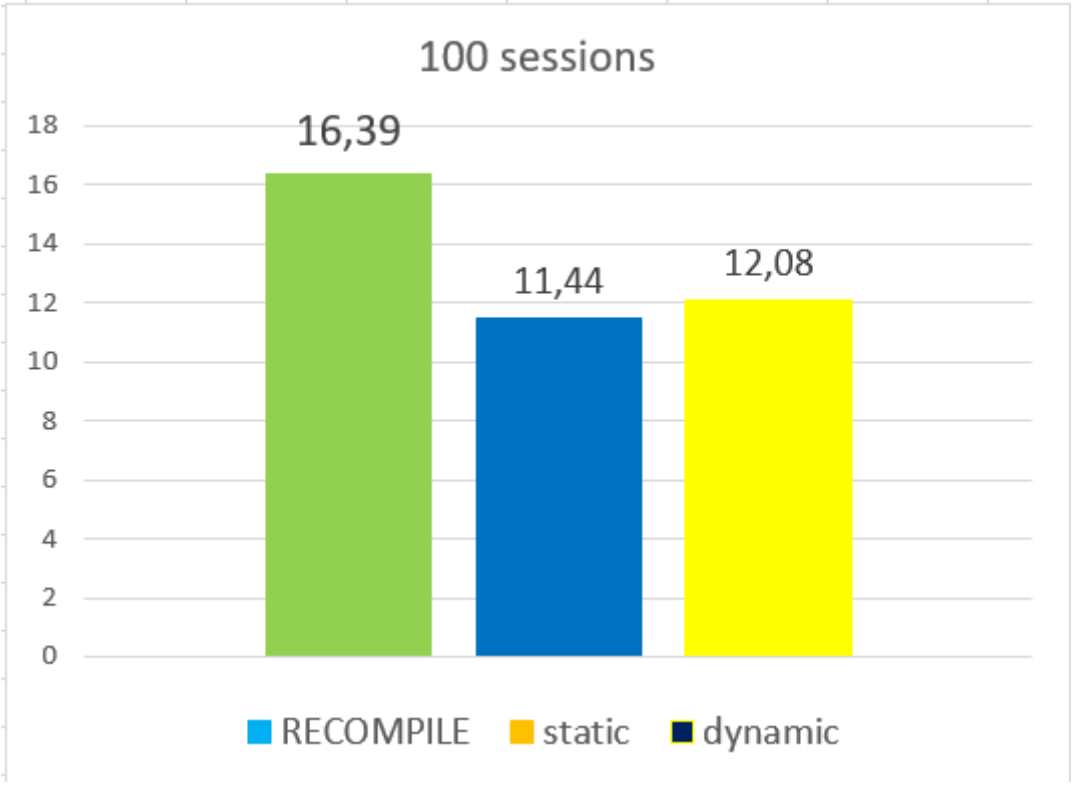
```
04/25/20 22:16:04.815 [0x00011954] Starting query execution...
04/25/20 22:16:04.824 [0x00011954] RsFx I/O completion thread starting
04/25/20 22:16:04.825 [0x00011954] File Stream support enabled.
04/25/20 22:16:04.827 [0x00011954]  BETA: Custom CLR Expression support enabled.
04/25/20 22:16:04.828 [0x00011954] Creating 100 thread(s) to process queries
04/25/20 22:16:04.872 [0x00011954] Worker threads created, beginning execution...
04/25/20 22:16:16.848 [0x00011954] Total IO waits: 0, Total IO wait time: 0 (ms)
04/25/20 22:16:16.848 [0x00011954] OSTRESS exiting normally, elapsed time: 00:00:12.294
04/25/20 22:16:16.849 [0x00011954] RsFx I/O completion thread ended.
```

OPTION (RECOMPILE)

```
04/25/20 22:12:27.812 [0x00010F34] Starting query execution...
04/25/20 22:12:27.825 [0x00010F34] RsFx I/O completion thread starting
04/25/20 22:12:27.825 [0x00010F34] File Stream support enabled.
04/25/20 22:12:27.827 [0x00010F34]  BETA: Custom CLR Expression support enabled.
04/25/20 22:12:27.827 [0x00010F34] Creating 100 thread(s) to process queries
04/25/20 22:12:27.881 [0x00010F34] Worker threads created, beginning execution...
04/25/20 22:12:44.099 [0x00010F34] Total IO waits: 0, Total IO wait time: 0 (ms)
04/25/20 22:12:44.099 [0x00010F34] OSTRESS exiting normally, elapsed time: 00:00:16.555
04/25/20 22:12:44.101 [0x00010F34] RsFx I/O completion thread ended.
```

Decomposition vs. OPTION (RECOMPILE)

- Static and dynamic decomposition perform better than the one that use the OPTION (RECOMPILE)
  - in case of many parallel sessions
  - in case of an expensive query compilation



Solution 7 – A Combined Solution

- Goal: Always get the optimal execution plan and reuse it for most common parameters

- Pros:
  - An optimal plan for the most important executions
  - Plan reusing

- How to implement?
  - Static SQL Decision Tree implementation combined with the OPTION (RECOMPILE)

Conclusion

- If you are OK with an average execution plan, you can disable parameter sniffing
- To get the best possible plan use the OPTION (RECOMPILE), but…
- If the compilation is to expensive, use query decomposition
- If you can use neither RECOMPILE, nor static decomposition, you can use dynamic SQL, but you have to prevent security issues
- You can also make a compromise and optimize just a small set of parameter values