SQL Server Basic Query Tuning Techniques

SQLMaestros Channels

Web	:	www.SQLMaestros.com
YouTube	:	https://www.youtube.com/user/SQLMaestros/videos
RSS	:	http://feeds.feedburner.com/SQLMaestros_AmitBansal
Twitter	:	www.twitter.com/SQLMaestros
LinkedIn	:	https://www.linkedin.com/showcase/14630987
Telegram	:	https://t.me/sqlmaestros
FB	:	www.facebook.com/SQLMaestros

HOLS Feedback	:	holfeedback@SQLMaestros.com
HOLS Support	:	holsupport@SQLMaestros.com

Terms of Use, Copyright & Intellectual Property

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of eDominer Systems.

The names of manufacturers, products, or URLs are provided for informational purposes only and eDominer makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of eDominer of the manufacturer or product. Links are provided to third party sites. Such sites are not under the control of eDominer and eDominer is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. eDominer is not responsible for webcasting or any other form of transmission received from any linked site. eDominer is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of eDominer of the site or the products contained therein.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property. This lab document (under the HOLs service by SQLMaestros) is just a learning document that enables you to learn and practice a topic/subject step-by-step, to be practiced in your test environment.

Microsoft, Excel, Office, and SQL Server, Azure are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

You hereby confirm and state that you will be the sole recipient of SQLMaestros Hands-On-Labs, depending on whatever you have purchased. You understand that you have purchased the lab for a lifetime use but the IP remains with eDominer Systems (parent co of SQLMaestros). You confirm that you will not share the lab document(s) or any part of it with any other individual and /or group and/or company and/or any entity, be it known to me or unknown to me. You specifically confirm that you will not give the downloaded/purchased labs or even a part of it to your company or your team. Further, you will not reproduce, or make multiple copies or store or introduce into a retrieval system, or transmit in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose. You understand that you are allowed to make one and only one backup for safekeeping/DR purposes. You will not share the backup link with any individual or any entity. Sharing the labs with anyone will be a gross violation of “terms of use”. If you are a trainer/coach/facilitator/teacher/instructor or are involved in teaching in any capacity, you confirm that you will not use the lab documents or share them with your class students or participants. In summary, you understand that the lab documents are only for your self-learning, for your personal use. You also understand and agree that any violation of the above, will cause irreparable harm/damage to the SQLMaestros brand, the Author & eDominer Systems as a company and all its brands. Under such circumstances, eDominer Systems shall be entitled to such injunctive or equitable relief as may be deemed proper by a court of competent jurisdiction against you.

Table of Contents

Before You Begin

Estimated time to complete this lab

40 minutes

Objectives:

After completing this lab, you will learn:

- Different query/table hints
- Ad Hoc query
- Parameter sniffing

Lab Setup Requirements

Before executing this lab:

- You must have SQL Server 2012 Standard/Developer/Enterprise/Evaluation edition or higher. Click here to download SQL Server evaluation edition
- You must have AdventureWorks sample databases. It is recommended that you have AdventureWorks2012 or higher. Click here to download AdventureWorks sample databases

Prerequisites

Before executing this lab:

- It is recommended that you have basic experience with SQL Server
- You have met the Lab Setup Requirements mentioned above

Lab Scenario

Many administrators address performance problems solely by tuning system-level server performance: for example, memory size, type of file system, number and type of processors, and so on. However, many performance problems cannot be resolved this way. They are better addressed by also analyzing the application queries and updates that the application submits to the database, and how these queries and updates interact with the data contained in the

database and the database schema. In the first exercise of this lab we will look into different query and table hints available in SQL Server. In the second exercise we will look into ad hoc query and how to deal with them and in the third and last exercise we will look into parameter sniffing and how to deal with it.

Tips to complete this lab successfully

Following these tips will be helpful in completing the lab successfully in time

- All lab files are located in **SQL Server Basic Query Tuning Techniques** folder
- The script(s) are divided into various sections marked with 'Begin', 'End' and 'Steps'. As per the instructions, execute the statements between particular sections only or for a particular step
- Read the instructions carefully and do not deviate from the flow of the lab
- Practice this lab only in your test machine/environment. Do not run this lab in your production environment

Exercise 1: Query & Table Hints

Overview

Query hints specify that the indicated hints should be used throughout the query. They affect all operators in the statement. If **UNION** is involved in the main query, only the last query involving a **UNION** operation can have the **OPTION** clause. Query hints are specified as part of the **OPTION** clause. If one or more query hints cause the query optimizer not to generate a valid plan, error **8622** is raised.

Table hints override the default behavior of the query optimizer for the duration of the data manipulation language (DML) statement by specifying a locking method, one or more indexes, a query processing operation such as a table scan or index seek, or other options. Table hints are specified in the **FROM** clause of the DML statement and affect only the table or view referenced in that clause.

Scenario

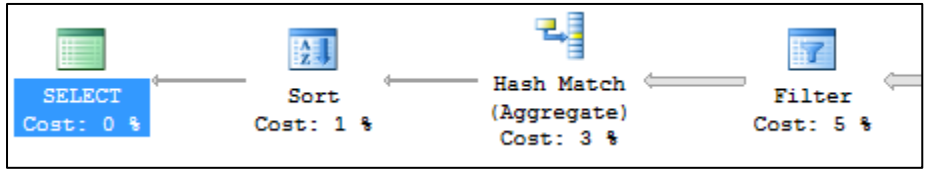
In this exercise, we will look into different query hints available in SQL Server.

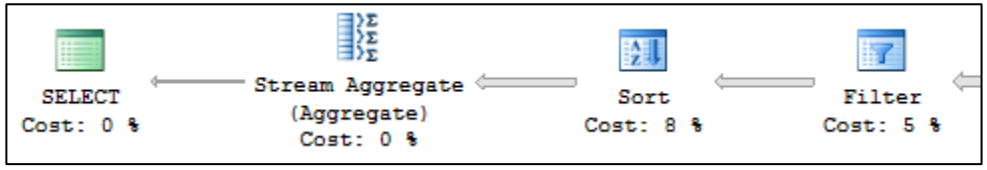
Tasks	Detailed Steps
Launch **SQL Server Management Studio**	1. Click **Start** | **All Programs** | **SQL Server 2012** | **SQL Server Management Studio** 2. In the **Connect to Server** dialog box, click **Connect**
Open **1_QueryHints.sql**	1. Click **File** | **Open** | **File** or press **(Ctrl + O)** 2. In **Open File** dialogue box, navigate to **SQL Server Basic Query Tuning Techniques\Scripts** folder 3. Select **1_QueryHints.sql** and click **Open**
Select **AdventureWorks2012** database	Execute the following statement(s) to select **AdventureWorks2012** database `-- Step 1: Execute the following statements to select AdventureWorks2012 database` `USE AdventureWorks2012;` `SET NOCOUNT ON;` `SET STATISTICS IO ON;` `GO`

Execute **SELECT** statement(s)

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) and observe the execution plan

```
------------------
-- Begin: Step 2
------------------
-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE ProductID = 870
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (HASH GROUP);
GO

-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE ProductID = 870
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (ORDER GROUP);
GO
------------------
-- End: Step 2
------------------
```





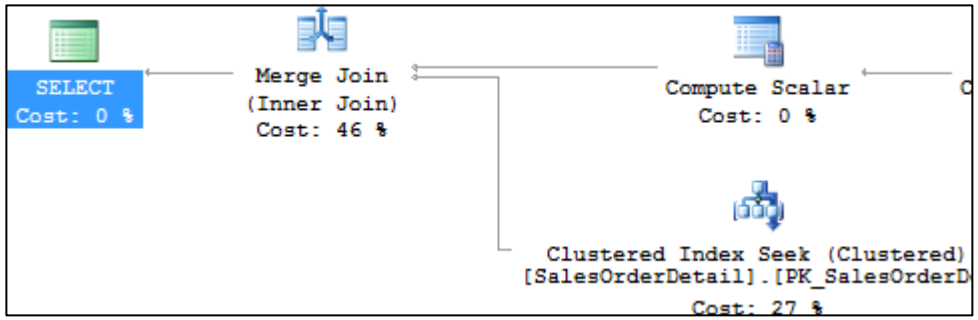
Explanation: {**HASH** | **ORDER**} **GROUP** query hints specifies that aggregations described in the **GROUP BY**, or **DISTINCT** clause of the query should use hashing or ordering. Note in case of hash aggregate sorting is done after aggregation and in the case of stream aggregate, sorting is done prior to aggregation. In the first **SELECT** statement we have used query hint **HASH GROUP**, thus the first query resulted into a hash match aggregate. In the second **SELECT** statement we have used query hint **ORDER GROUP**, thus the second query resulted into a stream aggregate.

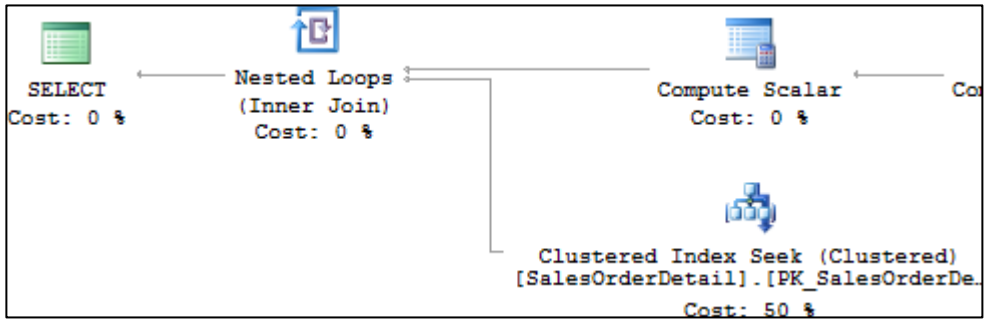
Execute **SELECT** statement(s)

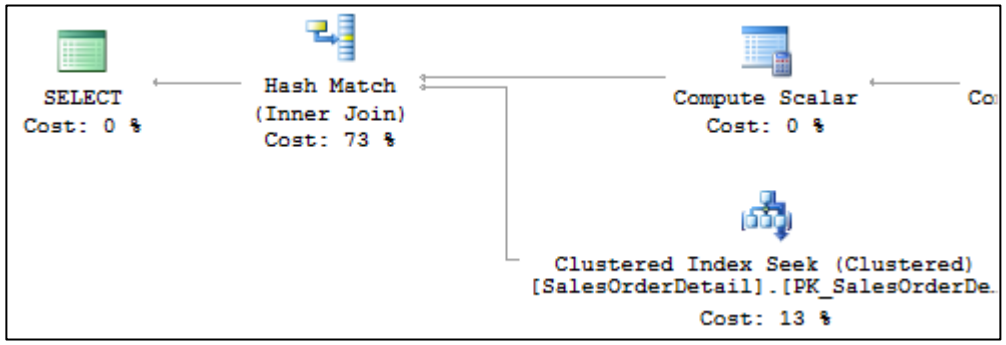
1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ()
2. Execute the following statement(s) and observe the execution plan

```
-------------------
-- Begin: Step 3
-------------------
-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT SOD.SalesOrderID, SOD.OrderQty, SOD.ProductID, SOH.CustomerID, SOH.TotalDue
FROM Sales.SalesOrderDetail AS SOD
INNER JOIN Sales.SalesOrderHeader AS SOH
ON SOD.SalesOrderID = SOH.SalesOrderID
WHERE SOD.SalesOrderID = 49999
OPTION(MERGE JOIN)

-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT SOD.SalesOrderID, SOD.OrderQty, SOD.ProductID, SOH.CustomerID, SOH.TotalDue

FROM Sales.SalesOrderDetail AS SOD
INNER JOIN Sales.SalesOrderHeader AS SOH
ON SOD.SalesOrderID = SOH.SalesOrderID
WHERE SOD.SalesOrderID = 49999
OPTION(LOOP JOIN)
```

```
-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT SOD.SalesOrderID, SOD.OrderQty, SOD.ProductID, SOH.CustomerID, SOH.TotalDue
FROM Sales.SalesOrderDetail AS SOD
INNER JOIN Sales.SalesOrderHeader AS SOH
ON SOD.SalesOrderID = SOH.SalesOrderID
WHERE SOD.SalesOrderID = 49999
OPTION(HASH JOIN)
--------------------
-- End: Step 3
--------------------
```







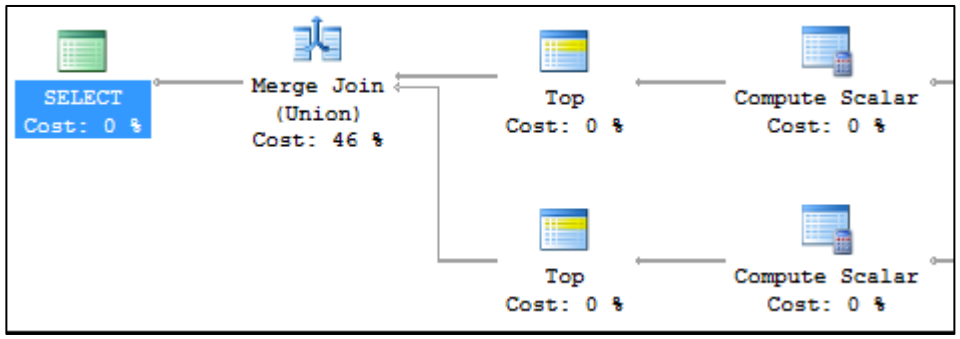
Explanation: { **LOOP** | **MERGE** | **HASH** } **JOIN** query hints specifies that all join operations are performed by **LOOP JOIN**, **MERGE JOIN**, or **HASH JOIN** in the whole query. If more than one join hint is specified, the optimizer selects the least expensive join strategy from the allowed ones. In the first **SELECT** statement we have used query hint **MERGE JOIN**, thus SQL Server performs a merge join to perform the join operation. In the second **SELECT** statement we have used query hint **LOOP JOIN**, thus SQL Server uses nested loop join to perform the join operation. In the third **SELECT** statement we have used query hint **HASH JOIN**, thus SQL Server uses hash join to perform the join operation.

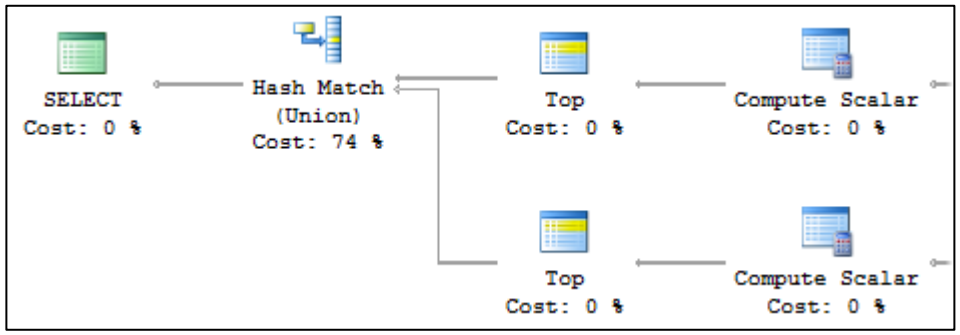
Execute **SELECT** statement(s)

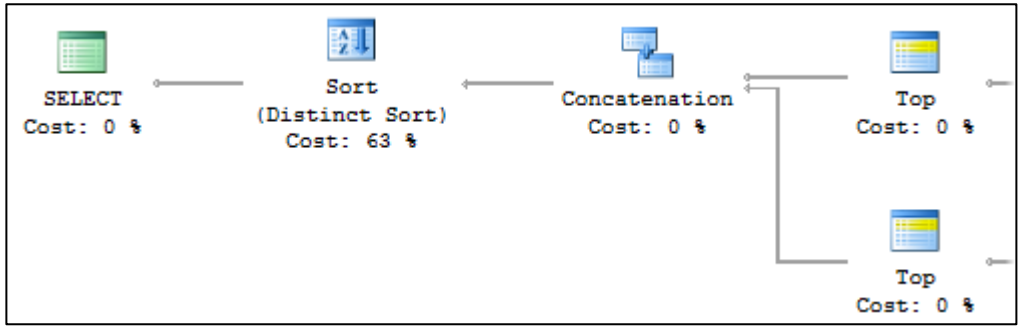
1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) and observe the execution plan

```
------------------
-- Begin: Step 4
------------------
-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT TOP 10 * FROM Sales.SalesOrderDetail
UNION
SELECT TOP 10 * FROM Sales.SalesOrderDetail
OPTION(MERGE UNION)

-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT TOP 10 * FROM Sales.SalesOrderDetail
UNION
```

```
SELECT TOP 10 * FROM Sales.SalesOrderDetail
OPTION(HASH UNION)

-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT TOP 10 * FROM Sales.SalesOrderDetail
UNION
SELECT TOP 10 * FROM Sales.SalesOrderDetail
OPTION(CONCAT UNION)
--------------------
-- End: Step 4
--------------------
```







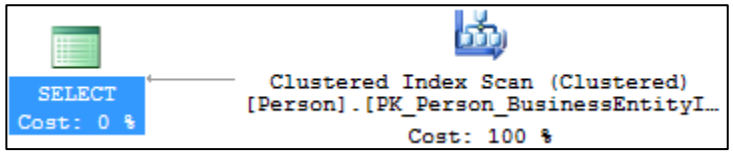
Explanation: {**MERGE** | **HASH** | **CONCAT**} **UNION** query hints specifies that all **UNION** operations are performed by merging, hashing, or concatenating **UNION** sets. If more than one **UNION** hint is specified, the query optimizer selects the least expensive strategy from those hints specified. In the first **SELECT** statement we have used query hint **MERGE UNION**, thus SQL Server uses merge join union to perform the union operation. In the second **SELECT** statement we have used query hint **HASH UNION**, thus SQL Server uses hash join union to perform the union operation. In the third **SELECT** statement we have used query hint **CONCAT JOIN**, thus SQL Server uses concatenation to perform the join operation.

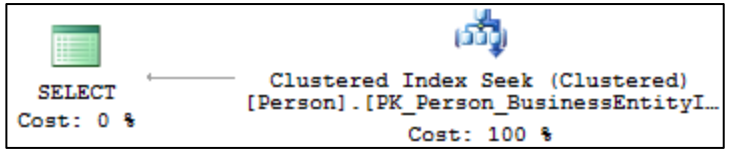
Execute **SELECT** statement(s)

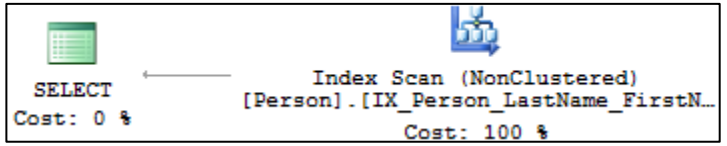
1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ()
2. Execute the following statement(s) and observe the execution plan

```
-------------------
-- Begin: Step 5
-------------------
-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT BusinessEntityID, FirstName, LastName
FROM Person.Person WITH(INDEX(0)) WHERE BusinessEntityID = 100

-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT BusinessEntityID, FirstName, LastName
FROM Person.Person WITH(INDEX(1)) WHERE BusinessEntityID = 100
```

```
-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT BusinessEntityID, FirstName, LastName
FROM Person.Person WITH(INDEX(2)) WHERE BusinessEntityID = 100
-------------------
-- End: Step 5
-------------------
```







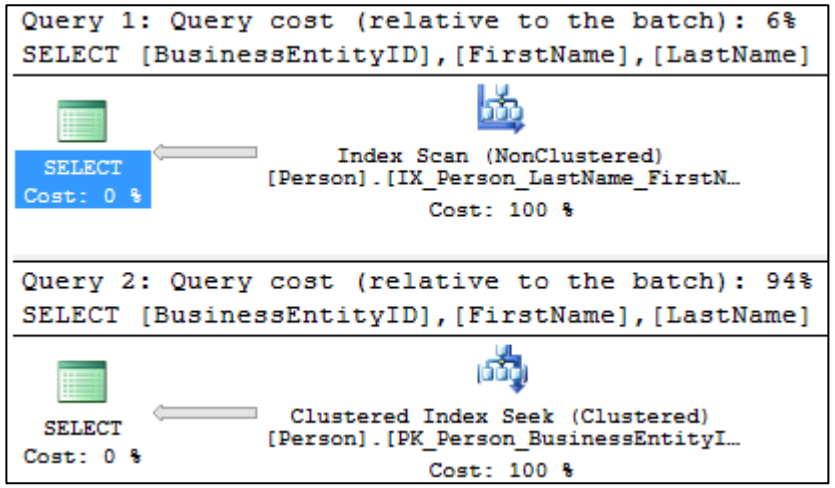

```
Table 'Person'. Scan count 1, logical reads 3834, physical reads 0,
Table 'Person'. Scan count 0, logical reads 3, physical reads 0, re
Table 'Person'. Scan count 1, logical reads 196, physical reads 0,
```

Explanation: {**INDEX** (**index_value | index_name**)} table hint specifies the names or IDs of one or more indexes to be used by the query optimizer when it processes the statement. The alternative **INDEX** = syntax specifies a single index value. Only one index hint per table can be specified. If a clustered index exists, **INDEX (0)** forces a clustered index scan and **INDEX (1)** forces a clustered index scan or seek. If no clustered index exists, **INDEX (0)** forces a table scan and **INDEX (1)** is interpreted as an error. In the first **SELECT** statement we have used table hint **INDEX (0)**, thus SQL Server uses clustered index scan to perform the operation. In the second **SELECT** statement we have used table hint **INDEX (1)**, thus SQL Server uses clustered index seek to perform the operation. In the third **SELECT** statement we have used table hint **INDEX (2)**, thus SQL Server uses non-clustered index scan to perform the operation.

Execute SELECT statement(s)

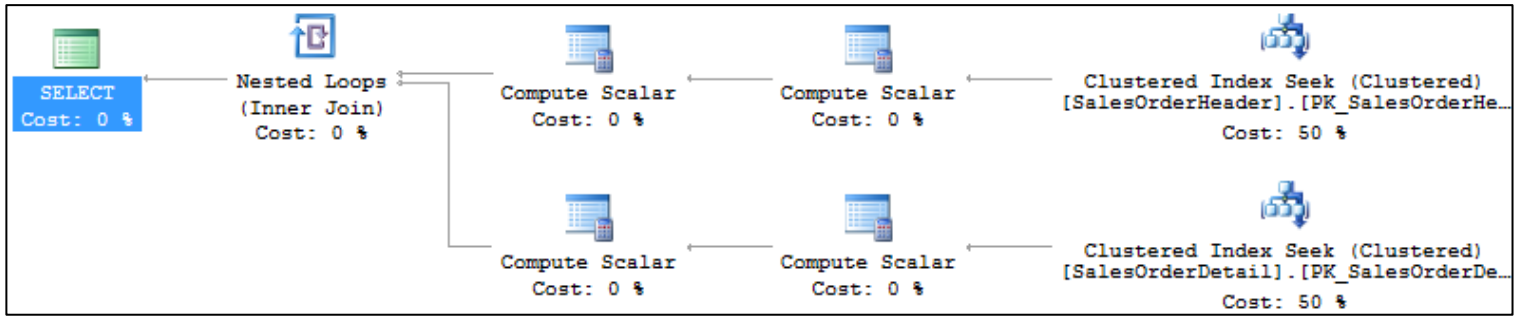
1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) and observe the execution plan

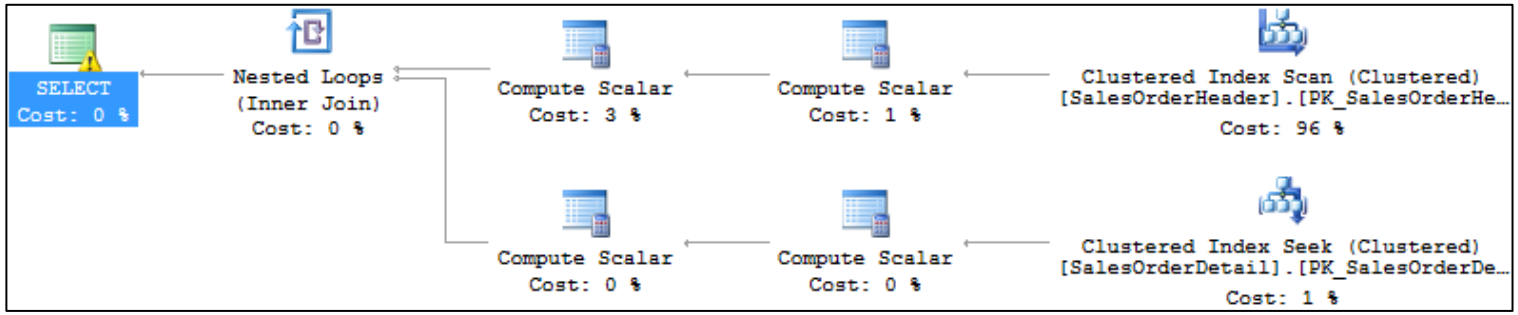
```
------------------
-- Begin: Step 6
------------------
-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT BusinessEntityID, FirstName, LastName
FROM Person.Person  WHERE BusinessEntityID > 100

-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT BusinessEntityID, FirstName, LastName
FROM Person.Person WITH(FORCESEEK) WHERE BusinessEntityID > 100
------------------
-- End: Step 6
------------------
```




```
Table 'Person'. Scan count 1, logical reads 196, physical reads 0,
Table 'Person'. Scan count 1, logical reads 3827, physical reads 0,
```

	Explanation: **FORCESEEK [(index_value(index_column_name [,... n]))]** table hint specifies that the query optimizer use only an index seek operation as the access path to the data in the table or view. Starting with SQL Server 2008 R2 SP1, index parameters can also be specified. In that case, the query optimizer considers only index seek operations through the specified index using at least the specified index columns. In the first **SELECT** statement we have not used any table hints, thus SQL Server uses non-clustered index scan to retrieve the result set. In the second **SELECT** statement we have used table hint **FORCESEEK,** thus SQL Server uses clustered index seek to retrieve the result set.
Execute **SELECT** statement(s)	1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar () 2. Execute the following statement(s) and observe the execution plan `------------------` `-- Begin: Step 7` `------------------` `-- Execute below select statement with actual execution plan (Ctrl + M)` `SELECT * FROM Sales.SalesOrderDetail AS SOD` `INNER JOIN Sales.SalesOrderHeader AS SOH` `ON SOD.SalesOrderID = SOH.SalesOrderID` `WHERE SOH.SalesOrderID = 50000` `-- Execute below select statement with actual execution plan (Ctrl + M)` `SELECT * FROM Sales.SalesOrderDetail AS SOD` `INNER JOIN Sales.SalesOrderHeader AS SOH WITH (FORCESCAN)` `ON SOD.SalesOrderID = SOH.SalesOrderID` `WHERE SOH.SalesOrderID = 50000` `------------------` `-- End: Step 7` `------------------`



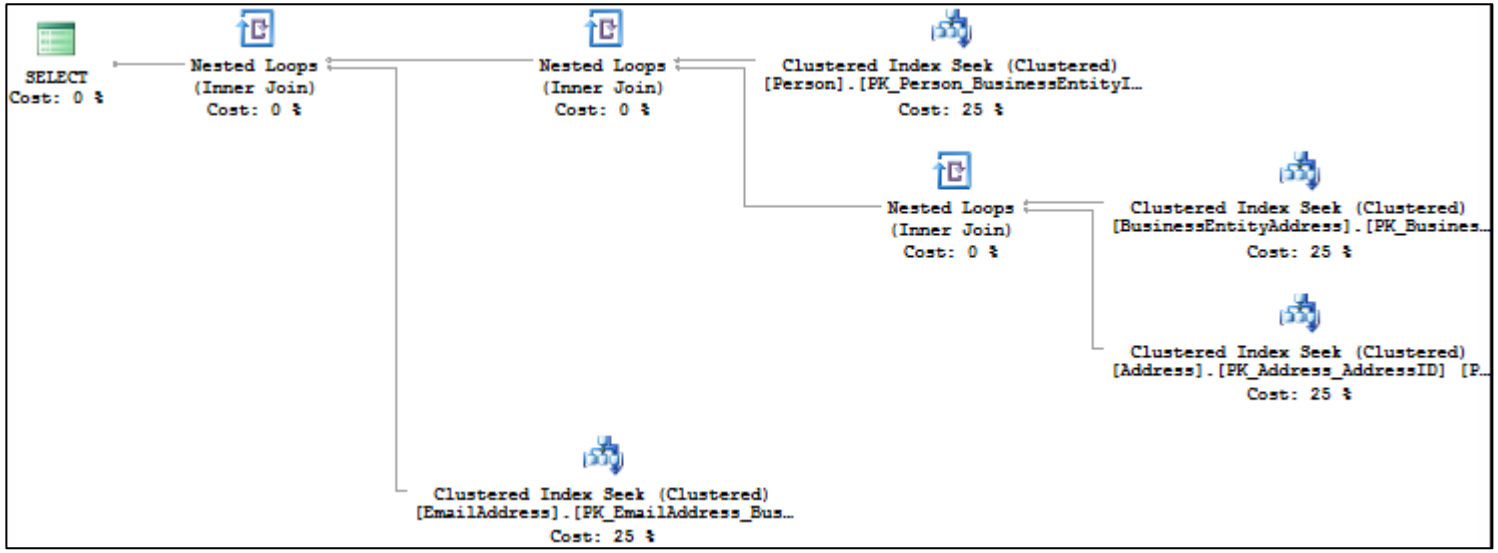


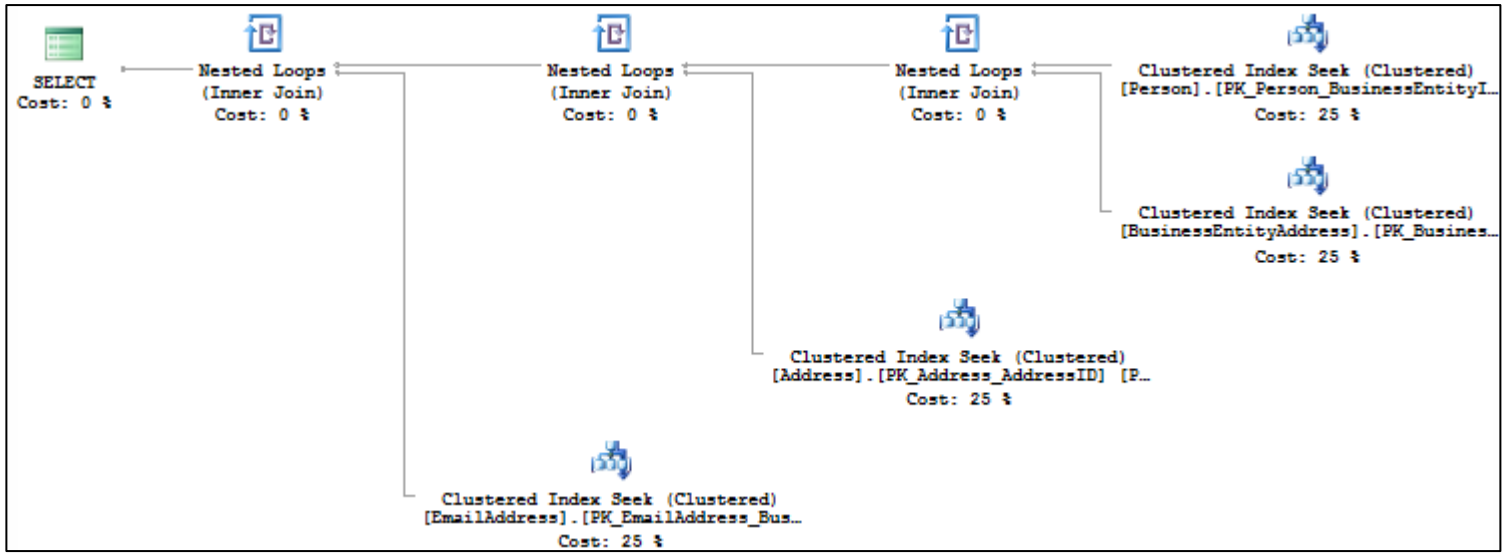

```
Table 'SalesOrderDetail'. Scan count 1, logical reads 3, physical reads 0,
Table 'SalesOrderHeader'. Scan count 0, logical reads 3, physical reads 0,
Table 'SalesOrderDetail'. Scan count 1, logical reads 3, physical reads 0,
Table 'SalesOrderHeader'. Scan count 1, logical reads 689, physical reads 0
```

Explanation: **FORCESCAN** table hint specifies that the query optimizer uses only an index scan operation as the access path to the referenced table or view. The **FORCESCAN** hint can be useful for queries in which the optimizer underestimates the number of affected rows and chooses a seek operation rather than a scan operation. When this occurs, the amount of memory granted for the operation is too small and query performance is impacted. In the first **SELECT** statement, we have not used any table hints, thus SQL Server uses clustered index seek operation to retrieve the result set. In the second **SELECT** statement we have used query hint **FORCESCAN**, thus SQL Server uses clustered index scan to retrieve the result set.

Execute a **SELECT** statement	1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar () 2. Execute the following statement(s) and observe the execution plan

```
-------------------
-- Begin: Step 8
-------------------
-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT
PP.BusinessEntityID,PP.FirstName,PP.LastName,
PBEA.AddressID,PA.AddressLine1,PA.AddressLine2,
PA.City,PA.PostalCode,PEA.EmailAddress
FROM Person.Person AS PP
INNER JOIN Person.BusinessEntityAddress AS PBEA
ON PP.BusinessEntityID = PBEA.BusinessEntityID
INNER JOIN Person.Address AS PA
ON PA.AddressID = PBEA.AddressID
INNER JOIN Person.EmailAddress AS PEA
ON PEA.BusinessEntityID = PP.BusinessEntityID
WHERE PP.BusinessEntityID = 100

-- Execute below select statement with actual execution plan (Ctrl + M)
SELECT
PP.BusinessEntityID,PP.FirstName,PP.LastName,
PBEA.AddressID,PA.AddressLine1,PA.AddressLine2,
PA.City,PA.PostalCode,PEA.EmailAddress
FROM Person.Person AS PP
INNER JOIN Person.BusinessEntityAddress AS PBEA
ON PP.BusinessEntityID = PBEA.BusinessEntityID
INNER JOIN Person.Address AS PA
ON PA.AddressID = PBEA.AddressID
INNER JOIN Person.EmailAddress AS PEA
ON PEA.BusinessEntityID = PP.BusinessEntityID
WHERE PP.BusinessEntityID = 100
OPTION(FORCE ORDER)
-------------------
-- End: Step 8
-------------------
```


SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Clustered Index Seek (Clustered)
[Person].[PK_Person_BusinessEntityI...
Cost: 25 %
Nested Loops
(Inner Join)
Cost: 0 %
Clustered Index Seek (Clustered)
[BusinessEntityAddress].[PK_Busines...
Cost: 25 %
Clustered Index Seek (Clustered)
[Address].[PK_Address_AddressID] [P...
Cost: 25 %
Clustered Index Seek (Clustered)
[EmailAddress].[PK_EmailAddress_Bus...
Cost: 25 %


SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Clustered Index Seek (Clustered)
[Person].[PK_Person_BusinessEntityI...
Cost: 25 %
Clustered Index Seek (Clustered)
[BusinessEntityAddress].[PK_Busines...
Cost: 25 %
Clustered Index Seek (Clustered)
[Address].[PK_Address_AddressID] [P...
Cost: 25 %
Clustered Index Seek (Clustered)
[EmailAddress].[PK_EmailAddress_Bus...
Cost: 25 %

	Observation: **FORCE ORDER** query hint specifies that the join order indicated by the query syntax is preserved during query optimization. Using **FORCE ORDER** does not affect possible role reversal behavior of the query optimizer.
Close all the query windows	Close all the query windows () and if **SSMS** asks to save changes, click **NO**

Summary

In this exercise, you have learned:

- Different query hints available in SQL Server
- Different table hints available in SQL Server

Exercise 2: Ad Hoc Query Optimization

Scenario

In this exercise, we will look at Ad Hoc query optimization techniques.

Tasks	Detailed Steps
Open **2_ADHOCQuery.sql**	1. Click **File** | **Open** | **File** or press (Ctrl + O) 2. In **Open File** dialogue box, navigate to **SQL Server Basic Query Tuning Techniques\Scripts** folder 3. Select **2_ADHOCQuery.sql** and click **Open**
Select **AdventureWorks2012** database	Execute the following statement(s) to select **AdventureWorks2012** database `-- Step 1: Execute the following statements to select AdventureWorks2012 database` `USE AdventureWorks2012;` `SET NOCOUNT ON;` `SET STATISTICS IO ON;` `DBCC FREEPROCCACHE;` `GO` **Note**: **SET STATISTICS IO** will give us the I/O statistics for any query executed in this session. **DBCC FREEPROCCACHE** will free the processes cache. Do not execute this command in your production environment
Execute **SELECT** statement(s)	Execute the following **SELECT** statement(s) in **step 2** **Warning:** Execute the below three queries (**SELECT** Statements) properly or else we will not get any output in **step 3.**

```
--------------------
-- Begin: Step 2
--------------------

-- Execute the following select statement(s)
SELECT P.Name,
       THA.TransactionDate,
       THA.TransactionType,
       THA.Quantity,
       THA.ActualCost
FROM Production.TransactionHistoryArchive AS THA
       JOIN Production.Product AS P
       ON THA.ProductID = P.ProductID
WHERE P.ProductID = 461

SELECT P.Name,
       THA.TransactionDate,
       THA.TransactionType,
       THA.Quantity,
       THA.ActualCost
FROM Production.TransactionHistoryArchive AS THA
       JOIN Production.Product AS P
       ON THA.ProductID = P.ProductID
WHERE P.ProductID = 712

SELECT P.Name,
       THA.TransactionDate,
       THA.TransactionType,
       THA.Quantity,
       THA.ActualCost
FROM Production.TransactionHistoryArchive AS THA
       JOIN Production.Product AS P
       ON THA.ProductID = P.ProductID
WHERE P.ProductID = 888

--------------------
-- End: Step 2
--------------------
```

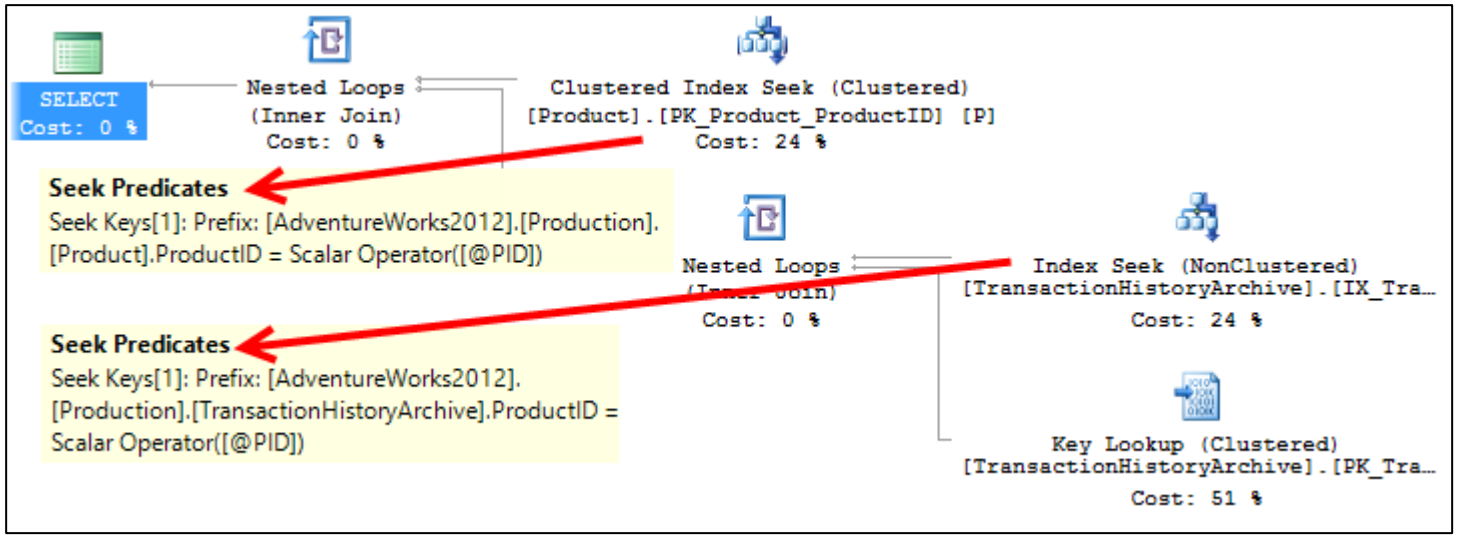
	Explanation: In **step 2** we have executed three **SELECT** statements with different predicate values. Since the predicates are different, SQL Server considers these three queries as separate ones and builds three separate plans. This type of query is better known as ad hoc query and is a major performance bottleneck. In the next step, we will observe this.
View query plan details	Execute the following statement(s) to view query plan details for the above three query `-- Step 3: View plan cache details for the above three query` `SELECT DEQS.execution_count,` `DEQS.query_hash,` `DEQS.query_plan_hash,` `DEST.text,` `DEQP.query_plan` `FROM sys.dm_exec_query_stats AS DEQS` `CROSS APPLY sys.dm_exec_sql_text(DEQS.plan_handle) AS DEST` `CROSS APPLY sys.dm_exec_query_plan(DEQS.plan_handle) AS DEQP` `WHERE DEST.text LIKE 'SELECT P.Name%'` execution_count / query_hash / query_plan_hash / text / query_plan 1 — 1 — 0x6A723E3D31FDC275 — 0xAE3EAFC4E56C7548 — SELECT P.Name, ... — <ShowPlanXML xml 2 — 1 — 0x6A723E3D31FDC275 — 0xBCA26E0843B17935 — SELECT P.Name, ... — <ShowPlanXML xml 3 — 1 — 0x6A723E3D31FDC275 — 0xF7645521D8C5472D — SELECT P.Name, ... — <ShowPlanXML xml **Observation:** In **step 2** we have executed three **SELECT** statements with different predicate values. Since the predicates are different, SQL Server considers these three queries as separate ones and builds three separate plans. We can tell that the three plan origin are same by looking at the **query_hash** (Observe same **query_hash** for the three query). Also, observe the **execution_count** for each query is 1 as we have only executed one instance of the query.

View plan cache details	Execute the following statement(s) to view plan cache details `-- Step 4: View plan cache details` `SELECT objtype AS [CacheType]` `, COUNT_BIG(*) AS [Total Plans]` `, SUM(CAST(size_in_bytes as decimal(18,2)))/1024/1024 AS [Total MBs]` `, AVG(usecounts) AS [Avg Use Count]` `, SUM(CAST((CASE WHEN usecounts = 1 THEN size_in_bytes ELSE 0 END) as decimal(18,2)))/1024/1024 AS [Total MBs - USE Count 1]` `, SUM(CASE WHEN usecounts = 1 THEN 1 ELSE 0 END) AS [Total Plans - USE Count 1]` `FROM sys.dm_exec_cached_plans` `GROUP BY objtype` `ORDER BY [Total MBs - USE Count 1] DESC` `GO` (1)
Clear plan cache	Execute the following statement(s) to clear plan cache `-- Step 5: Clear processes cache` `DBCC FREEPROCCACHE;` `GO` **Warning:** Do not execute this command in a production environment. **DBCC FREEPROCCACHE** clears all the plan cache from memory.

(1)

	CacheType	Total Plans	Total MBs	Avg Use Count	Total MBs - USE Count 1	Total Plans - USE Count 1
1	Adhoc	4	0.546875	1	0.492187	3
2	Check	1	0.031250	6	0.000000	0
3	Prepared	18	0.562500	16	0.000000	0
4	View	4	0.132812	2	0.000000	0

CREATE a stored procedure	Execute the following statement(s) to **CREATE** a stored procedure named **usp_GETPRODETAILS** `-- Step 6: Create a stored procedure usp_GETPRODETAILS` `CREATE PROCEDURE usp_GETPRODETAILS(@PID INT)` `AS` `SELECT P.Name,` `THA.TransactionDate,` `THA.TransactionType,` `THA.Quantity,` `THA.ActualCost` `FROM Production.TransactionHistoryArchive AS THA` `JOIN Production.Product AS P` `ON THA.ProductID = P.ProductID` `WHERE P.ProductID = @PID` **Explanation:** In the case of stored procedure, if we pass predicates as a parameter then same query plan can be used by SQL Server to execute the stored procedure (Shown in next few steps). This is known as parameter sniffing.
Execute stored procedure with different parameter values	Execute the following statement(s) to execute stored procedure **usp_GETPRODETAILS** with different parameter values `------------------` `-- Begin: Step 7` `------------------` `-- Execute stored procedure usp_GETPRODETAILS with different parameter list` `EXEC usp_GETPRODETAILS @PID = 461;` `GO` `EXEC usp_GETPRODETAILS @PID = 712;` `GO` `EXEC usp_GETPRODETAILS @PID = 888;` `GO` `------------------` `-- End: Step 7` `------------------`



Observation: Query plan is built depending upon the first parameter we have passed, and rest will use the same query plan. For the above execution the first parameter we have passed is 461, thus SQL Server uses this value to create the plan and this plan will be reused.

```
CREATE PROCEDURE usp_GETPRODETAILS(@PID INT) AS SELECT P.Name,          THA.TransactionDate,
Production.Product AS P    ON THA.ProductID = P.ProductID WHERE P.ProductID = @PID
```

Parameter List	@PID
Column	@PID
Parameter Compiled Value	(461)

View query plan details

Execute the following statement(s) to view query plan details for the above three query

```
-- Step 8: View query deatils using DMV
SELECT DEQS.execution_count,
       DEQS.query_hash,
       DEQS.query_plan_hash,
       DEST.text,
       DEQP.query_plan
FROM sys.dm_exec_query_stats AS DEQS
       CROSS APPLY sys.dm_exec_sql_text(DEQS.plan_handle) AS DEST
       CROSS APPLY sys.dm_exec_query_plan(DEQS.plan_handle) AS DEQP
WHERE DEST.text LIKE 'CREATE PROCEDURE usp_GETPRODETAILS%'
```

	execution_count	query_hash	query_plan_hash	text	query_plan
1	3	0x6A723E3D31FDC275	0xAE3EAFC4E56C7548	CREATE PROC...	<ShowPlanXML xmlns='

Observation: Since we have used a stored procedure to execute the **SELECT** statement and passed the predicate through a parameter, SQL Server uses parameter sniffing and creates only one execution plan.

Clear plan cache

Execute the following statement(s) to clear plan cache

```
-- Step 9: Clear processes cache
DBCC FREEPROCCACHE;
GO
```

Warning: Do not execute this command in a production environment. **DBCC FREEPROCCACHE** clears all the plan cache from memory.

Enable 'optimize for ad hoc workloads' server setting	Execute the following statement(s) to enable 'optimize for ad hoc workloads' server level setting `-- Step 10: Enable optimize for ad hoc workloads server setting` `EXEC sys.sp_configure N'show advanced options', N'1'` `GO` `RECONFIGURE WITH OVERRIDE` `GO` `EXEC sys.sp_configure N'optimize for ad hoc workloads', N'1'` `GO` `RECONFIGURE WITH OVERRIDE` `GO` **Explanation:** The 'optimize for ad hoc workloads' option is used to improve the efficiency of the plan cache for workloads that contain many single use ad hoc batches. When this option is set to 1, the Database Engine stores a small compiled plan stub in the plan cache when a batch is compiled for the first time, instead of the full compiled plan. This helps to relieve memory pressure by not allowing the plan cache to become filled with compiled plans that are not reused. Setting the 'optimize for ad hoc workloads' to 1 affects only new plans; plans that are already in the plan cache are unaffected.
Execute **SELECT** statement(s)	Execute the following **SELECT** statement(s) in **step 11** `--------------------` `-- Begin: Step 11` `--------------------` `-- Execute the following select statement(s)` `SELECT P.Name,` `THA.TransactionDate,` `THA.TransactionType,` `THA.Quantity,` `THA.ActualCost` `FROM Production.TransactionHistoryArchive AS THA` `JOIN Production.Product AS P` `ON THA.ProductID = P.ProductID` `WHERE P.ProductID = 461`

```
SELECT P.Name,
       THA.TransactionDate,
       THA.TransactionType,
       THA.Quantity,
       THA.ActualCost
FROM Production.TransactionHistoryArchive AS THA
       JOIN Production.Product AS P
       ON THA.ProductID = P.ProductID
WHERE P.ProductID = 712

SELECT P.Name,
       THA.TransactionDate,
       THA.TransactionType,
       THA.Quantity,
       THA.ActualCost
FROM Production.TransactionHistoryArchive AS THA
       JOIN Production.Product AS P
       ON THA.ProductID = P.ProductID
WHERE P.ProductID = 888

--------------------
-- End: Step 11
--------------------
```

Note: In **step 11** we have executed three **SELECT** statements with different predicate values. Since the predicates are different, SQL Server considers these three queries as separate ones and builds three separate plans.

Warning: Select the above three queries (**SELECT** statements)properly or else we will not get any output in **step 12**

View query plan details

Execute the following statement(s) to view query plan details for the above three query

```
-- Step 12: View plan cache details for the above three query
SELECT DEQS.execution_count,
       DEQS.query_hash,
       DEQS.query_plan_hash,
       DEST.text,
```

```
    DEQP.query_plan
FROM sys.dm_exec_query_stats AS DEQS
      CROSS APPLY sys.dm_exec_sql_text(DEQS.plan_handle) AS DEST
      CROSS APPLY sys.dm_exec_query_plan(DEQS.plan_handle) AS DEQP
WHERE DEST.text LIKE 'SELECT P.Name%'
```

	execution_count	query_hash	query_plan_hash	text	query_plan
1	1	0x6A723E3D31FDC275	0xAE3EAFC4E56C7548	SELECT P.Na...	NULL
2	1	0x6A723E3D31FDC275	0xBCA26E0843B17935	SELECT P.Na...	NULL
3	1	0x6A723E3D31FDC275	0xF7645521D8C5472D	SELECT P.Na...	NULL

Observation: In **step 11** we have executed three **SELECT** statements with different predicate values. Since the predicates are different, SQL Server considers these three queries as separate ones and builds three separate plans. We can tell that the three plan origin are same by looking at the **query_hash** (Observe same **query_hash** for the three query). Also, observe the **execution_count** for each query is 1 as we have only executed one instance of the query. But no query plan is generated as we have enabled 'optimize for ad hoc workload' server level setting.

Execute a SELECT statement

Execute the following statement(s) two times

```
-- Step 13: Execute the following select statement
SELECT P.Name,
       THA.TransactionDate,
       THA.TransactionType,
       THA.Quantity,
       THA.ActualCost
FROM Production.TransactionHistoryArchive AS THA
       JOIN Production.Product AS P
       ON THA.ProductID = P.ProductID
WHERE P.ProductID = 461
```

Note: Execute the above SELECT statement two times, or else we will not get the query plan in **step 14** output.

View query plan details	Execute the following statement(s) to view query plan details for the above three query `-- Step 14: View plan cache details for the above three query` `SELECT DEQS.execution_count,` `DEQS.query_hash,` `DEQS.query_plan_hash,` `DEST.text,` `DEQP.query_plan` `FROM sys.dm_exec_query_stats AS DEQS` `CROSS APPLY sys.dm_exec_sql_text(DEQS.plan_handle) AS DEST` `CROSS APPLY sys.dm_exec_query_plan(DEQS.plan_handle) AS DEQP` `WHERE DEST.text LIKE 'SELECT P.Name%'` execution_count / query_hash / query_plan_hash / text / query_plan 1 / 1 / 0x6A723E3D31FDC275 / 0xAE3EAFC4E56C7548 / SELECT P.Name,... / <ShowPlanXML xmlns="h 2 / 1 / 0x6A723E3D31FDC275 / 0xAE3EAFC4E56C7548 / SELECT P.Name,... / NULL 3 / 1 / 0x6A723E3D31FDC275 / 0xBCA26E0843B17935 / SELECT P.Name,... / NULL 4 / 1 / 0x6A723E3D31FDC275 / 0xF7645521D8C5472D / SELECT P.Name,... / NULL **Note:** If 'optimize for ad hoc workloads' server setting is enabled, then query plans are generated for the second execution of the same query. For the first execution, SQL Server saves a small compiled plan stub.
Cleanup	Execute the following script in Cleanup section `------------------` `-- Begin: Cleanup` `------------------` `-- Execute the following statement to drop usp_GETPRODETAILS stored procedure` `DROP PROCEDURE usp_GETPRODETAILS` `-- Disable optimize for ad hoc workloads server setting` `EXEC sys.sp_configure N'show advanced options', N'1'` `GO`

	`` RECONFIGURE WITH OVERRIDE GO EXEC sys.sp_configure N'optimize for ad hoc workloads', N'0' GO RECONFIGURE WITH OVERRIDE GO ------------------ -- End: Cleanup ------------------ ```
Close all the query windows	Close all the query windows () and if **SSMS** asks to save changes, click **NO**

Summary

In this exercise, you have learned:

- Concept of ad hoc query
- How to reduce ad hoc query
- Concept of 'optimize for ad hoc workload' server setting

Exercise 3: Parameter Sniffing Optimization

Scenario

In this exercise, we will look at index fragmentation and query performance.

Tasks	Detailed Steps
Open **3_ParameterSniffing.sql**	1. Click **File** | **Open** | **File** or press (Ctrl + O) 2. In **Open File** dialogue box, navigate to **SQL Server Basic Query Tuning Techniques\Scripts** folder 3. Select **3_ParameterSniffing.sql** and click **Open**
Select **AdventureWorks2012** database	Execute the following statement(s) to select **AdventureWorks2012** database `-- Step 1: Execute the following statements to select AdventureWorks2012 database` `USE AdventureWorks2012;` `SET NOCOUNT ON;` `SET STATISTICS IO ON;` `DBCC FREEPROCCACHE;` `GO` **Note**: **SET STATISTICS IO** will give us the I/O statistics for any query executed in this session.
CREATE a stored procedure	Execute the following statement(s) to **CREATE** a stored procedure named **uspAddressByCity** `-- Step 2: Create a stored procedure named uspAddressByCity` `IF(SELECT OBJECT_ID('uspAddressByCity'))` `IS NOT NULL` `DROP PROCEDURE dbo.uspAddressByCity;` `GO` `CREATE PROCEDURE dbo.uspAddressByCity @City NVARCHAR(30)`

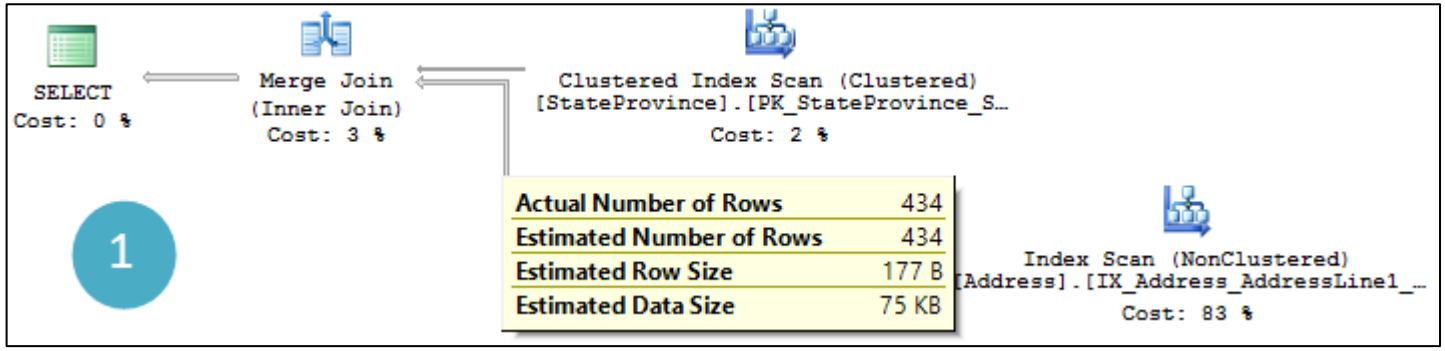
```
AS
SELECT A.AddressID,
       A.AddressLine1,
       A.AddressLine2,
       A.City,
       SP.Name AS StateProvinceName,
       A.PostalCode
FROM Person.Address AS A
       JOIN Person.StateProvince AS SP
       ON A.StateProvinceID = SP.StateProvinceID
WHERE A.City = @City
```

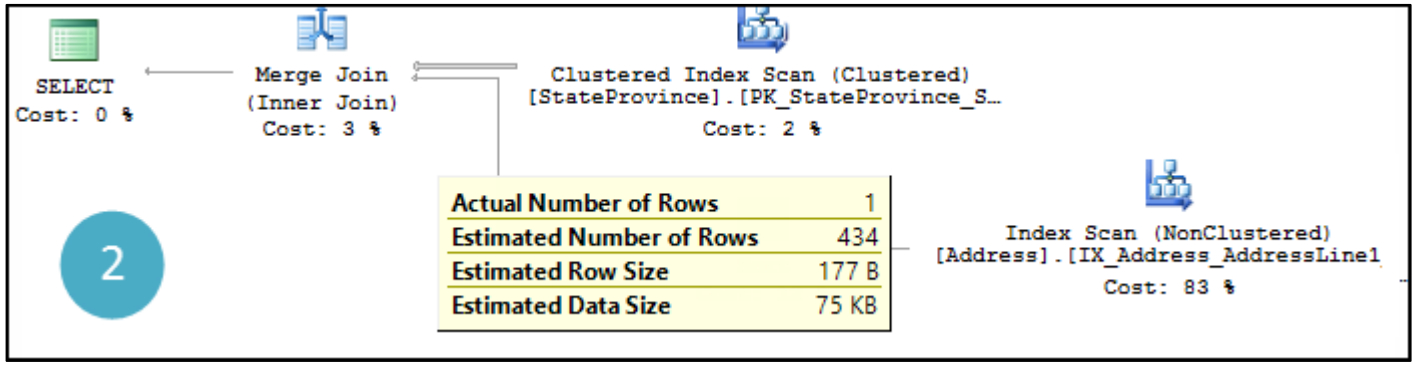
Execute stored procedure with different parameter list

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ()
2. Execute the following statement(s) and observe the execution plan

```
---------------------
-- Begin: Step 3
---------------------
-- Execute the stored procedure with actual execution plan (Ctrl + M)
EXEC uspAddressByCity @City = N'London';

-- Execute the stored procedure with actual execution plan (Ctrl + M)
EXEC uspAddressByCity @City = N'Mentor';
---------------------
-- End: Step 3
---------------------
```

Explanation: When a stored procedure is compiled or recompiled, the parameter values passed for that invocation are "sniffed" and used for cardinality estimation. The net effect is that the plan is optimized as if those specific parameter values were used as literals in the query. In the first stored procedure execution, no. of records returned is 434, thus SQL Server chooses a non-clustered index scan to retrieve the result set and saves that execution plan. In the second stored procedure execution, SQL Server reuses the same execution plan by sniffing the parameter list, though the no. of records is only one, SQL Server uses a non-clustered index scan.





Observation: The second stored procedure execution returns only one row, but if we look at the execution plan of the second stored procedure execution, SQL Server performs a clustered index scan instead of seeking for only one record.

Clear plan cache

Execute the following statement(s) to clear plan cache

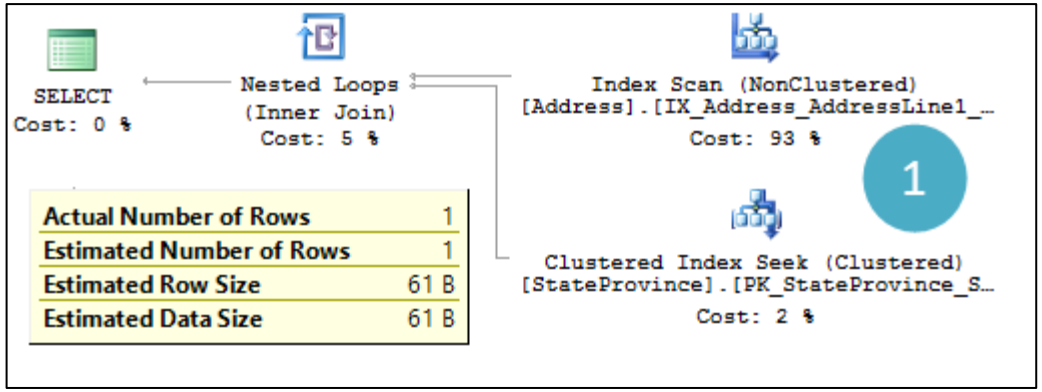
```
-- Step 4: Clear processes cache
DBCC FREEPROCCACHE;
GO
```

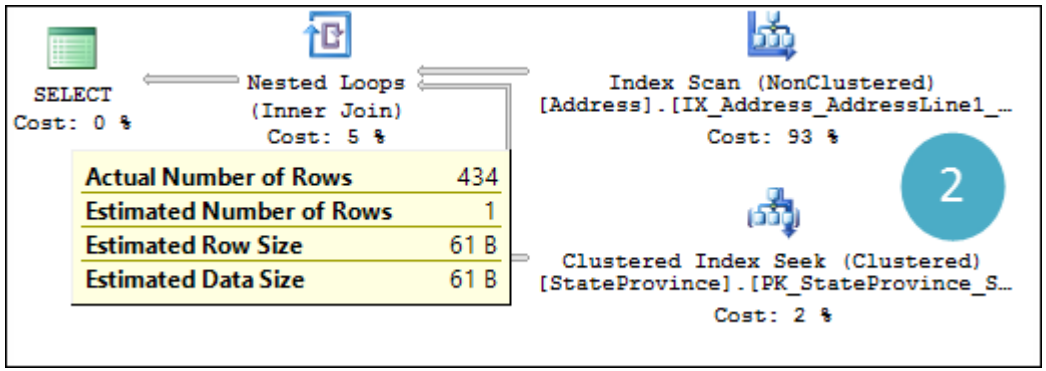
Warning: Do not execute this command in a production environment. **DBCC FREEPROCCACHE** clears all the plan cache from memory.

Execute stored procedure with different parameter list

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) and observe the execution plan

```
---------------------
-- Begin: Step 5
---------------------
-- Execute the stored procedure with actual execution plan (Ctrl + M)
EXEC uspAddressByCity @City = N'Mentor';


-- Execute the stored procedure with actual execution plan (Ctrl + M)
EXEC uspAddressByCity @City = N'London';
---------------------
-- End: Step 5
---------------------
```





Observation: The second stored procedure execution returns 434 rows, but if we look at the execution plan of the second stored procedure execution, SQL Server performs a clustered index seek, previously it was doing a scan.

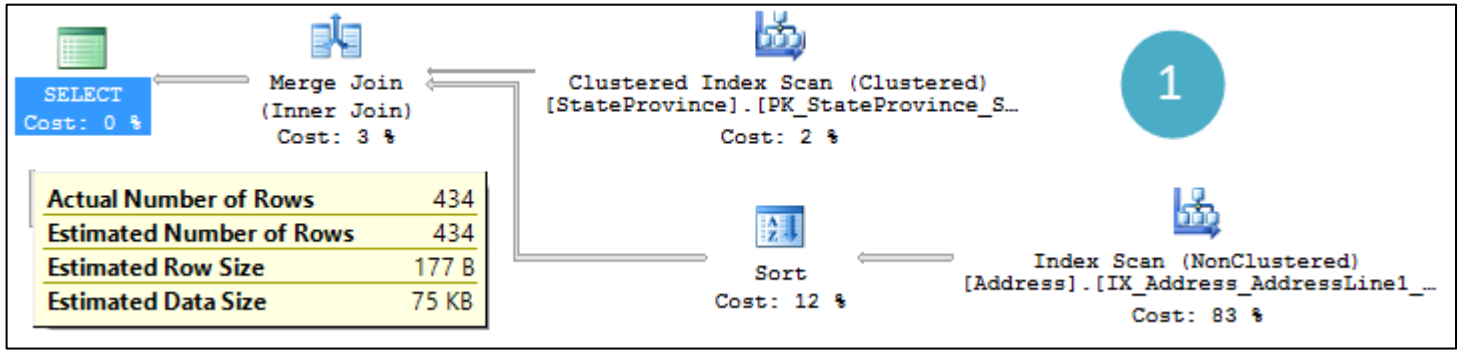
Explanation: In the first stored procedure execution, no. of the record returned is 1, thus SQL Server chooses a non-clustered index seek to retrieve the result set and saves that execution plan. In the second stored procedure execution, SQL Server reuses the same execution plan by sniffing the parameter list, though the no. of records is 434, SQL Server uses a non-clustered index seek.

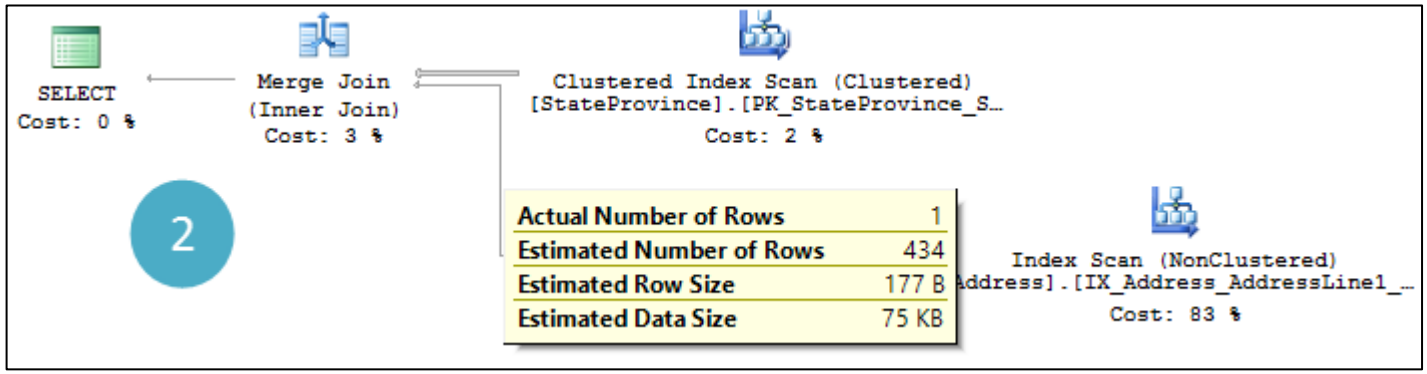
ALTER uspAddressByCity stored procedure

Execute the following statement(s) to **ALTER uspAddressByCity** stored procedure

```
-- Step 6: Alter stored procedure uspAddressByCity
ALTER PROCEDURE dbo.uspAddressByCity @City NVARCHAR(30)
AS
SELECT A.AddressID,
       A.AddressLine1,
        A.AddressLine2,
        A.City,
       SP.Name AS StateProvinceName,
       A.PostalCode
FROM Person.Address AS A
       JOIN Person.StateProvince AS SP
ON A.StateProvinceID = SP.StateProvinceID
```

	`` WHERE A.City = @City OPTION (OPTIMIZE FOR (@City = 'London')); ``` **Explanation**: In the above **ALTER** statement, we have used a query hint **OPTIMIZE FOR** to optimize the query **WHERE City = 'London'**.
Clear plan cache	Execute the following statement(s) to clear plan cache ``` -- Step 7: Execute the following statement to clear processes cache DBCC FREEPROCCACHE; GO ``` **Warning:** Do not execute this command in a production environment. **DBCC FREEPROCCACHE** clears all the plan cache from memory.
Execute stored procedure with different parameter list	1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar () 2. Execute the following statement(s) and observe the execution plan ``` --------------------- -- Begin: Step 8 --------------------- -- Execute the stored procedure with actual execution plan (Ctrl + M) EXEC uspAddressByCity @City = N'London'; -- Execute the stored procedure with actual execution plan (Ctrl + M) EXEC uspAddressByCity @City = N'Mentor'; --------------------- -- End: Step 8 --------------------- ```





Observation: The second stored procedure execution returns 1 row, but if we look at the execution plan of the second stored procedure execution, SQL Server performs a clustered index scan, previously it was doing a seek.

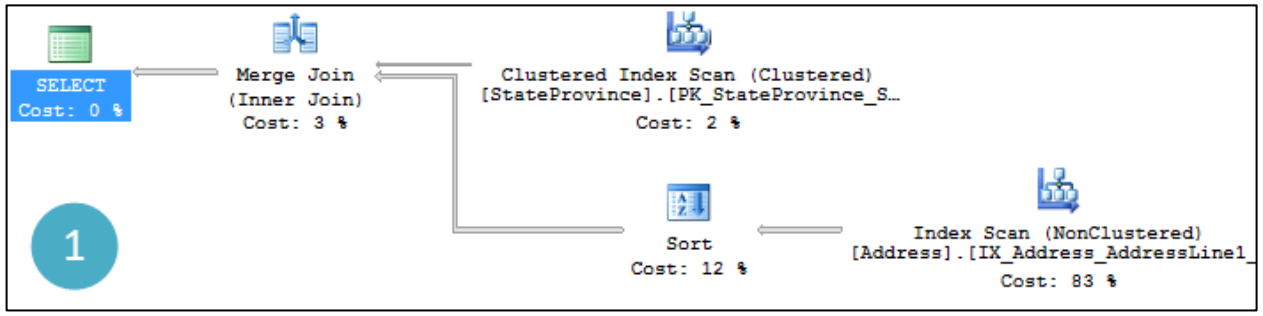
Explanation: In the first stored procedure execution, no. of the record returned is 434, thus SQL Server chooses a non-clustered index scan to retrieve the result set and we also have optimized the query for the first execution. Thus in the second stored procedure execution, SQL Server reuses the same execution plan by sniffing the parameter list, though the no. of records is 1, SQL Server uses a non-clustered index scan.

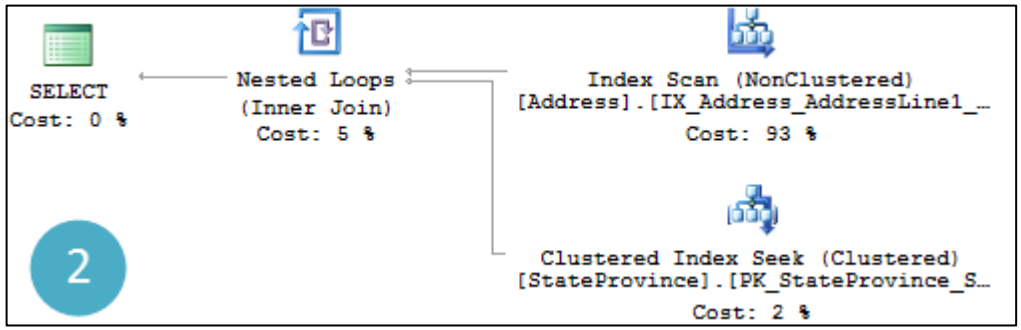
ALTER uspAddressByCity stored procedure	Execute the following statement(s) to **ALTER uspAddressByCity** stored procedure `-- Step 9: Alter stored procedure uspAddressByCity` `ALTER PROCEDURE dbo.uspAddressByCity @City NVARCHAR(30)` `AS` `SELECT A.AddressID,` `A.AddressLine1,` `A.AddressLine2,` `A.City,` `SP.Name AS StateProvinceName,` `A.PostalCode` `FROM Person.Address AS A` `JOIN Person.StateProvince AS SP` `ON A.StateProvinceID = SP.StateProvinceID` `WHERE A.City = @City` `OPTION (RECOMPILE);` **Explanation**: In the above **ALTER** statement, we have used a query hint **RECOMPILE**, which means SQL Server will generate new query plan whenever the above stored procedure gets executed.
Clear plan cache	Execute the following statement(s) to clear plan cache `-- Step 10: Execute the following statement to clear processes cache` `DBCC FREEPROCCACHE;` `GO` **Warning:** Do not execute this command in a production environment. **DBCC FREEPROCCACHE** clears all the plan cache from memory.

Execute stored procedure with different parameter list

1. Turn on actual execution plan. Either press **Ctrl + M** or use the **SQL Editor** toolbar ( )
2. Execute the following statement(s) and observe the execution plan

```
---------------------
-- Begin: Step 11
---------------------
-- Execute the stored procedure with actual execution plan (Ctrl + M)
EXEC uspAddressByCity @City = N'London';

-- Execute the stored procedure with actual execution plan (Ctrl + M)
EXEC uspAddressByCity @City = N'Mentor';
---------------------
-- End: Step 11
---------------------
```





	Observation: The first stored procedure execution results non-clustered index scan as the no. of records returned is 434, on the other hand, the second stored procedure execution a non-clustered index seek as the no. of records returned is 1. **Explanation**: In the previous step (step 9) we have **ALTER** the stored procedure and added query hint **RECOMPILE,** thus SQL Server recompiles the execution plan every time we execute the stored procedure.
DROP stored procedure **uspAddressByCity**	Execute the following statement(s) to **DROP uspAddressByCity** stored procedure `-- Step 12: Execute the following statement to drop uspAddressByCity stored procedure` `DROP PROCEDURE dbo.uspAddressByCity;` `GO`
Close all the query windows	Close all the query windows ( ) and if **SSMS** asks to save changes, click **NO**

Summary

In this exercise, you have learned:

- Concept of parameter sniffing
- How to deal with parameter sniffing

Other learning opportunities from us that might interest you.

Brand	Description	Important Links
DataPlatformGeeks	DataPlatformGeeks is a community of Data, Analytics & AI professionals. It is a community initiative and all activities including webinars, events, etc., are free and community driven. You can join for free and access all learning resources for free. Join here: https://www.dataplatformgeeks.com/registration/ Follow on Twitter. Follow on LinkedIn.	www.DataPlatformGeeks.com YouTube: www.YouTube.com/SQLServerGeeks Telegram: https://t.me/dataplatformgeeks More Social Channels: https://www.dataplatformgeeks.com/our-social-channels/
Data Platform Summit	Data Platform Summit is Asia's largest Data, Analytics & AI learning event. Whole week of deep-dive training takes place in Bangalore each year in the month of August/September. World's best speakers and delegates join from more than 20 countries.	www.DPS10.com www.DataPlatformSummit.in
PEOPLEWARE INDIA TECHNOLOGY TRAINING SPECIALISTS	Peopleware India offers full length video courses on latest technologies.	www.PeoplewareIndia.com
SQLServerGeeks	Another community initiative, SQLServerGeeks is one of the most popular websites on SQL Server with thousands of blogs, articles, videos and learning resources on Microsoft SQL Server. SQLServerGeeks also organizes frequent webinars and events to impart free education on SQL Server.	www.SQLServerGeeks.com www.facebook.com/SQLServerGeeks YouTube: www.YouTube.com/SQLServerGeeks www.twitter.com/SQLServerGeeks Join SSG on LinkedIn
SQLशीघ्र SQLSHIGHRA	At SQLShighra.com you get quick SQL Server Tips and know-how related to SQL Server Internals, Troubleshooting and Performance Tuning. These are short & casual videos, mostly Amit Bansal's SQL brain dump and he records them anywhere, anytime (not literally).	www.SQLShighra.com
SQLMaestros	A team of specialists on SQL Server & Microsoft Data Platform, SQLMaestros offers affordable learning solutions and also offers SQL Server health check & baselining service. Video Courses. Hands-On-Labs. Learning Kits.	Follow/Join SQL Server Discussions: RSS Subscription, Telegram, YouTube, Twitter, LinkedIn, Facebook

*The above information is constantly updated here: www.eDominer.com/learning