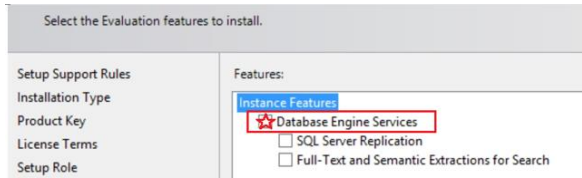
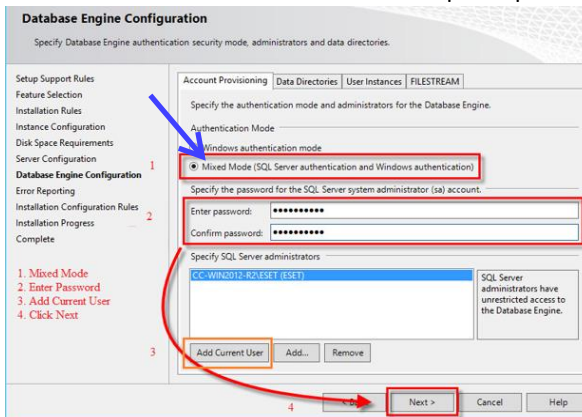


## The "SandBox" Setup: Know your Tools & Environment!

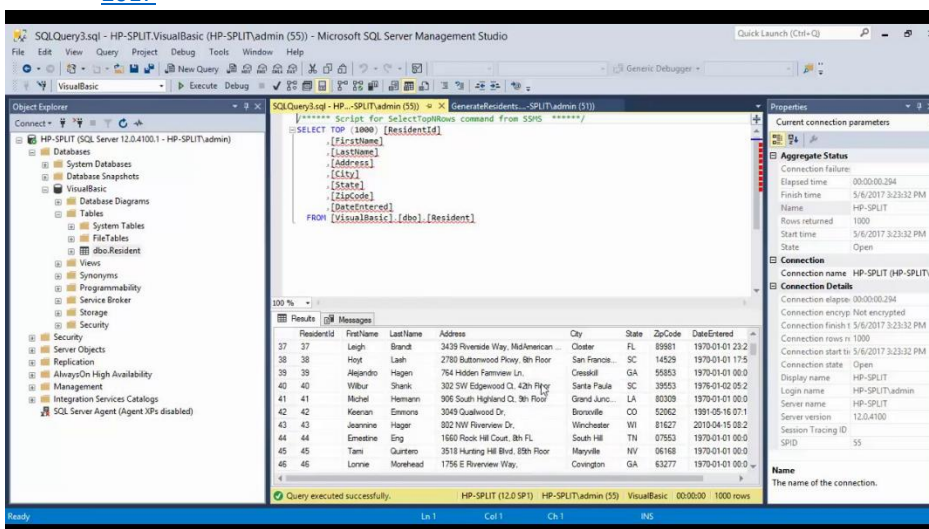
- Download and install SQL Express or Developer Edition (Both are FREE – Developer has more features – Express is perfect for learning)
  - <https://www.microsoft.com/en-us/sql-server/sql-server-downloads>
- Pick Database Engine from the features list.



- Use all the defaults but its quicker if "Check for Updates" is not selected.
- Select Mixed Mode – Enter sa for User – pick a password – for simplicity Password1 or Pa\$\$word will work..



- Finish the wizard...
- Download and install SSMS – SQL Server Management Studio
  - <https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017>



- Basic SSMS settings – good place to learn your tools...
  - <https://docs.microsoft.com/en-us/sql/ssms/tutorials/ssms-configuration?view=sql-server-2017#maximize-query-editor>
- Space

## Containers & Relations in "Theory"

- A relation in the relational model is what SQL represents with a table.
- Relation Model vs. Cursor - p7
- Relation = Table - has a heading and a body.
- Heading = a set of Attributes = columns
- Body = a set of Tuples = Rows
- Attributes must have unique names - Alias with AS
- Note that if an ORDER BY is specified, the result isn't relational.
  - This means that you can't operate on such result with an outer query because an outer query expects a relation as input.

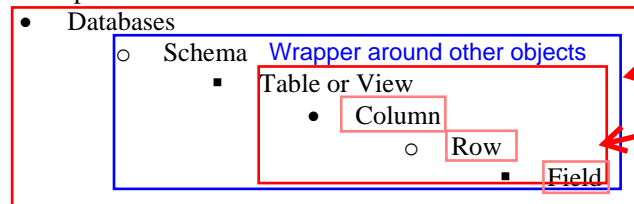
## Container Hierarchy or Parent/Child and what gets inherited.... such as Security – *Know the "Boundaries"*

Domain - International or national in scope

Name if for what it does - naming conventions.. (If you can)

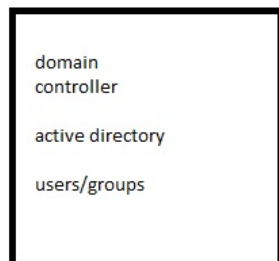
- OS: Windows, Linux etc....
  - SQL Server Instance: Default or named...
    - File Groups

EXECUTE AS Admin

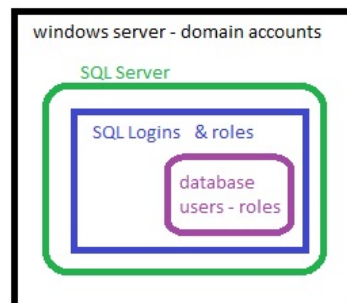


.mdb or .ndb - Log Files - .Ldb - Snapshot

Least Privilege



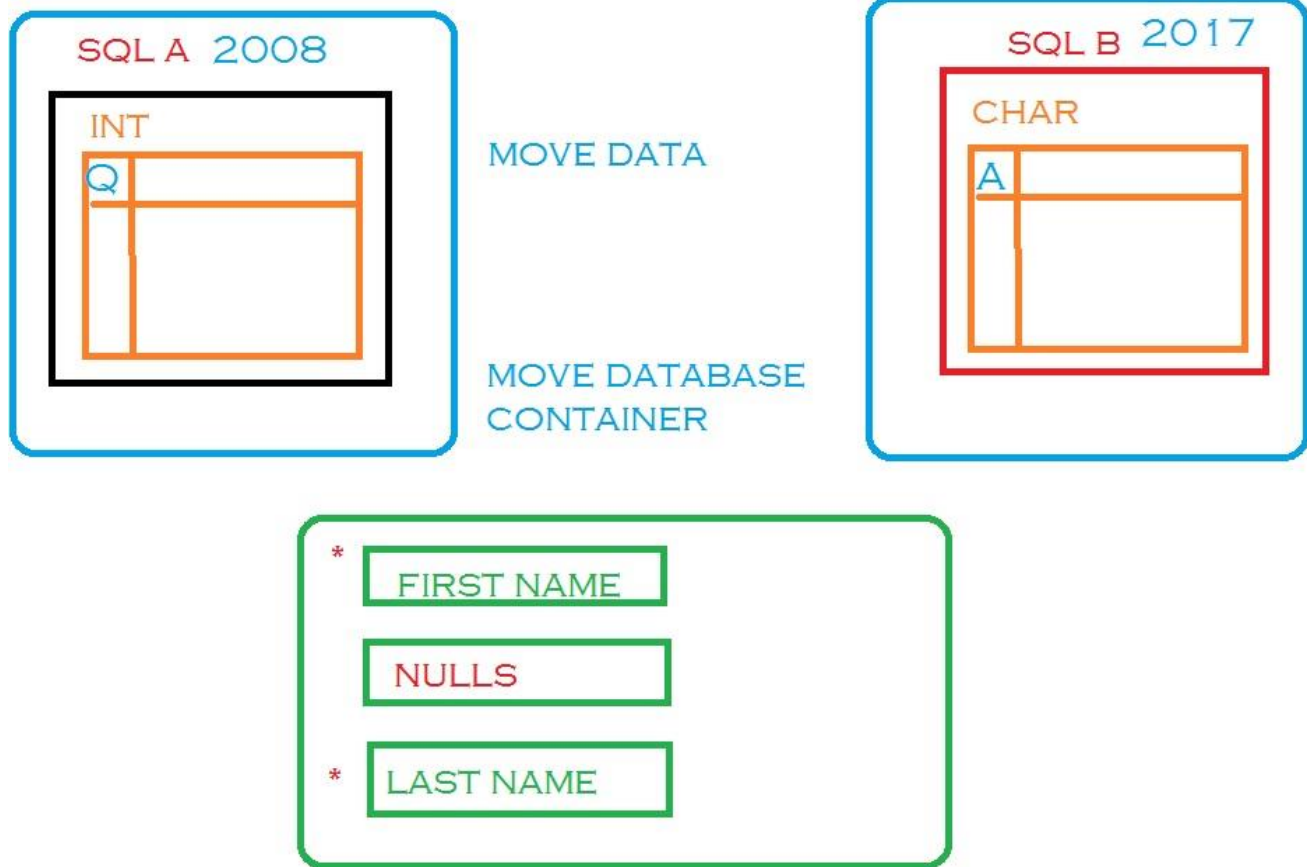
scope of boundary



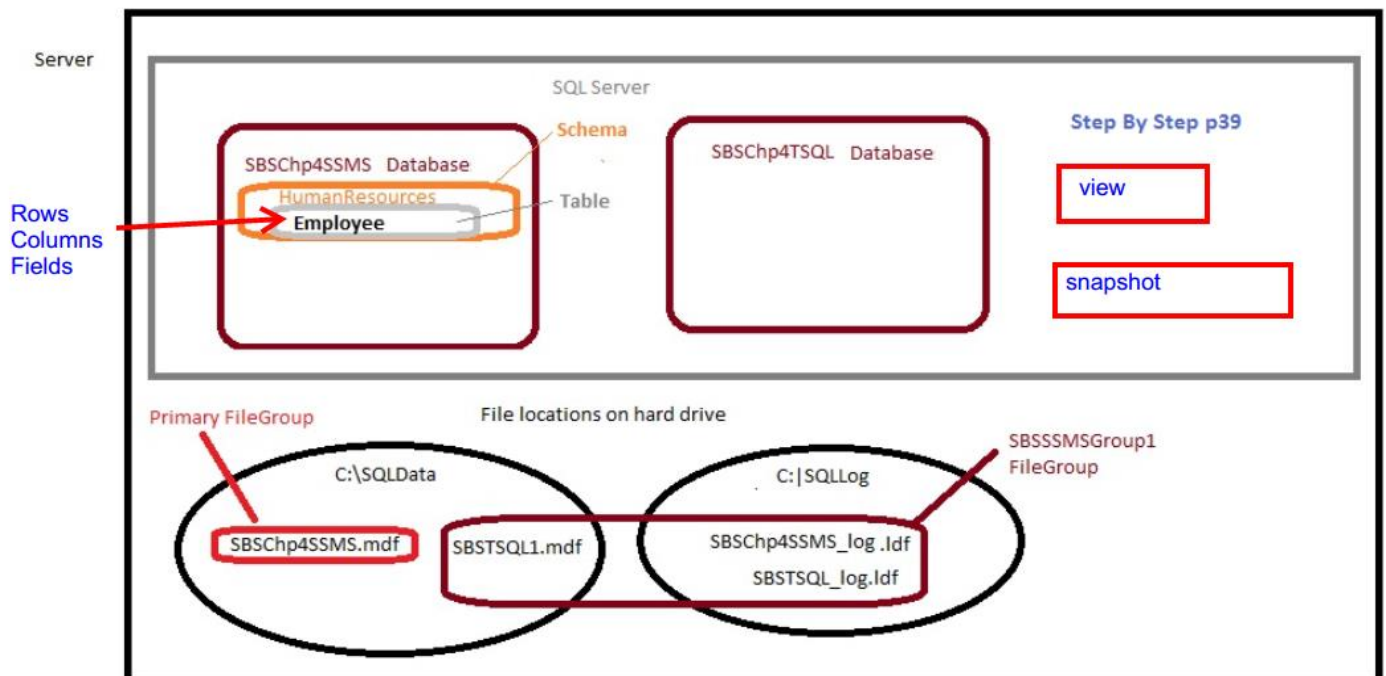
domain account user

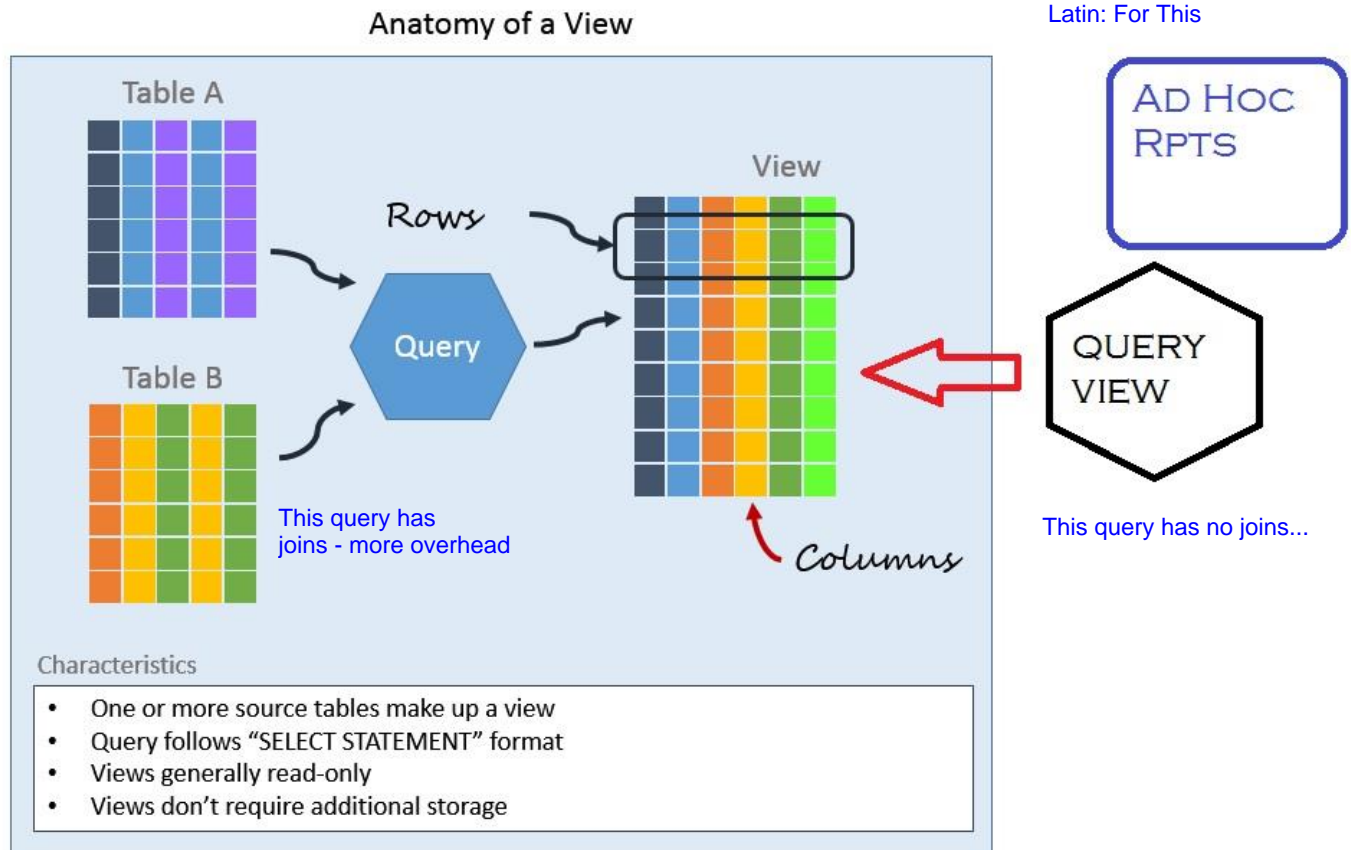


## Move Data or Move a Container



## Windows/SQL Server Database Engine Instance/Database/Schema/Table/Column/Row – View/Snapshot



**The View****Programmatically created "Containers"****CTEs... Common Table Expression**

- Are unindexable (but can use existing indexes on referenced objects)
- Cannot have constraints
- Are essentially disposable VIEWS
- Persist only until the next query is run
- Can be recursive
- Do not have dedicated stats (rely on stats on the underlying objects)

**#Temp Tables...**

- Are real materialized tables that exist in tempdb
- Can be indexed
- Can have constraints
- Persist for the life of the current CONNECTION
- Can be referenced by other queries or sub-procedures
- Have dedicated stats generated by the engine

## CTE Anatomy

CTE definition begins with an **WITH** statement. If CTE is not the first statement in the batch, then the last statement prior to CTE should terminate with symbol ;

**CTE expression\_name**

**CTE column\_names**, specifying column names is optional. If specified it should match to the number columns returned by the CTE query\_definition. If column names are not specified it will return the column names as-is returned by the CTE query\_definition

**WITH** **BasicCTE** (**Id, Name**)

**AS** (**SELECT 1, 'Basavaraj Biradar'**)

**SELECT \* FROM BasicCTE;**

**CTE query\_definition**, **SELECT** statement which populates the CTE expression. If there are more than one **SELECT** statement then they should be combined by one of the operators: **UNION ALL**, **UNION**, **EXCEPT**, or **INTERSECT**

**SELECT** statement consuming the CTE **BasicCTE**. Only the immediate single **SELECT**, **UPDATE**, **INSERT** or **DELETE** Statement can consume the CTE

### CTE RESULT:

```
WITH BasicCTE ( Id, Name ) AS (SELECT 1, 'Basavaraj Biradar')
SELECT * FROM BasicCTE;
```

Results	
Messages	
Id	Name
1	Basavaraj Biradar

```
SQLQuery6.sql - SU...URESH\Suresh (52)* - X
-- Creating Global Temp Table in SQL Server
CREATE TABLE ##GlobalTemp
(
    [ID] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](255) NULL,
    [LastName] [nvarchar](255) NULL,
    [Education] [nvarchar](255) NULL,
    [Occupation] [nvarchar](255) NULL,
    [YearlyIncome] [float] NULL,
    [Sales] [float] NULL
)

-- Inserting Values into Global SQL Temp Table
INSERT INTO ##GlobalTemp (
    [FirstName], [LastName], [Education], [Occupation], [YearlyIncome], [Sales]
) VALUES (
    'Tutorial', 'Gateway', 'Masters Degree', 'Teaching', 12000, 200
), ('Imran', 'Khan', 'Bachelors', 'Skilled Professional', 13900, 100)
, ('Doe', 'Lara', 'Degree', 'Management', 25000, 60)
, ('Ramesh', 'Kuman', 'Bachelors', 'Professional', 35400, 630)
```

100 % < @tutorialgateway.org

Messages

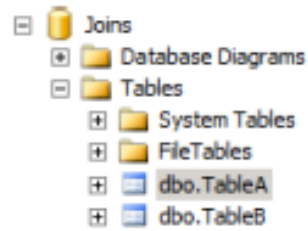
(4 row(s) affected)

### JoinCerSize: Primary Key, Foreign Key, One to One, One to Many, Many to Many & NULLS

1. Create "Joins" database
2. Add 2 tables: TableA & TableB – Columns: ID & Name – **No PK or IDENTITY**
3. Add 4 records to each

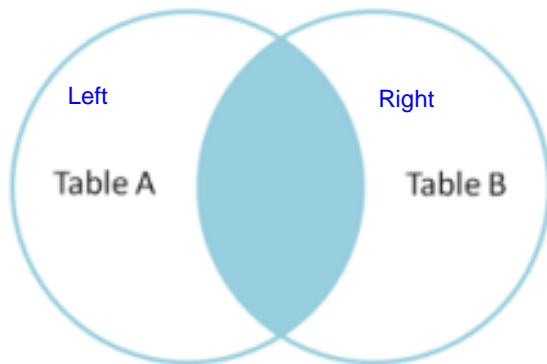
SELECT \* FROM TableA **On the Left side**  
 INNER JOIN TableB **On the Right side**  
 ON TableA.name = TableB.name

Table A		Table B	
ID	Name	ID	Name
1	Pirate	1	Rutabaga
2	Monkey	2	Pirate
3	Ninja	3	Darth Vader
4	Spaghetti	4	Ninja



MIA-SQL1\MKTG.Joins - dbo.TableA			
	Column Name	Data Type	Allow Nulls
	ID	int	<input type="checkbox"/>
	Name	varchar(50)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

MIA-SQL1\MKTG.Joins - dbo.TableA			
	Column Name	Data Type	Allow Nulls
	ID	int	<input type="checkbox"/>
	Name	varchar(50)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>



WHERE is for filtering

ON - LIKE etc - matching

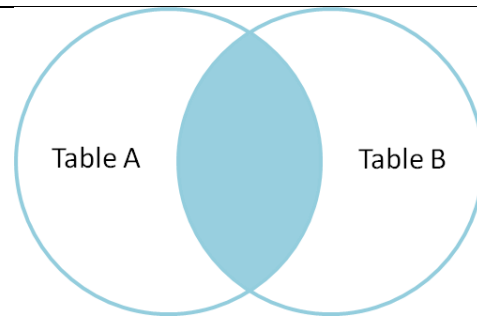
Join\_Cer\_Size Companion

```
SELECT * FROM TableA
INNER JOIN TableB
ON TableA.name = TableB.name
```

Inner join produces only the set of records that match in both Table A and Table B.



ID	Name	ID	Name
1	Pirate	2	Pirate
3	Ninja	4	Ninja



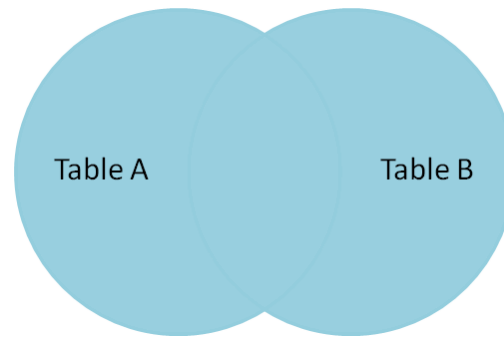
```
SELECT * FROM TableA
```

```
FULL OUTER JOIN TableB
```

```
ON TableA.name = TableB.name
```

ID	Name	ID	Name
1	Pirate	2	Pirate
2	Monkey	NULL	NULL
3	Ninja	4	Ninja
4	Spaghetti	NULL	NULL
NULL	NULL	1	Rutabaga
NULL	NULL	3	Darth Vader

**Full outer join** produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is **no match**, the missing side will contain null.



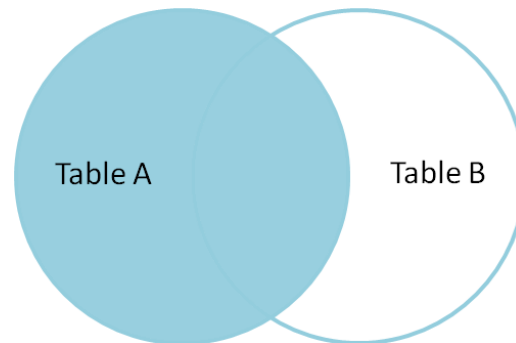
```
SELECT * FROM TableA
```

```
LEFT OUTER JOIN TableB
```

```
ON TableA.name = TableB.name
```

ID	Name	ID	Name
1	Pirate	2	Pirate
2	Monkey	NULL	NULL
3	Ninja	4	Ninja
4	Spaghetti	NULL	NULL

**Left outer join** produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is **no match, the right side will contain null.**



```
SELECT * FROM TableA
```

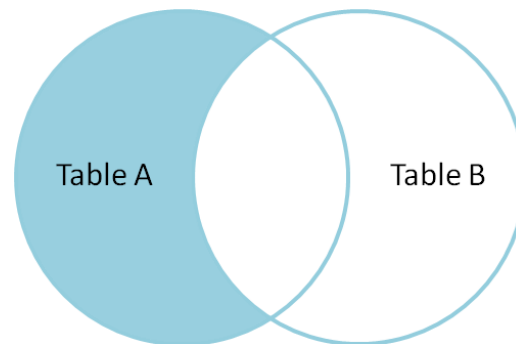
```
LEFT OUTER JOIN TableB
```

```
ON TableA.name = TableB.name
```

```
WHERE TableB.id IS null
```

ID	Name	ID	Name
2	Monkey	NULL	NULL
4	Spaghetti	NULL	NULL

To produce the set of records only in Table A, but not in Table B, we perform the same **left outer join**, then **exclude** the records we don't want **from the right side** via a **where clause**.

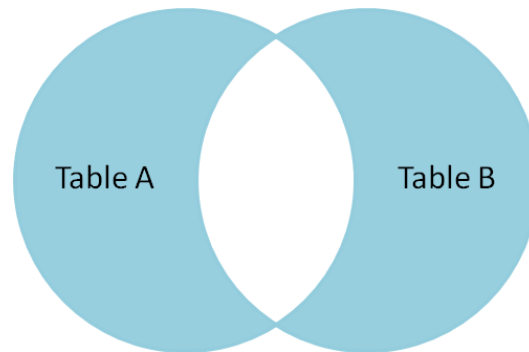




```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
OR TableB.id IS null
```

ID	Name	ID	Name
2	Monkey	NULL	NULL
4	Spaghetti	NULL	NULL
NULL	NULL	1	Rutabaga
NULL	NULL	3	Darth Vader

To produce the set of records unique to Table A and Table B, we perform the same **full outer join**, then **exclude** the records we don't want from **both** sides via a **where clause**.



There's also a cartesian product or cross join, which as far as I can tell, can't be expressed as a Venn diagram:

```
SELECT * FROM TableA
```

```
CROSS JOIN TableB
```

This joins "everything to everything", resulting in  $4 \times 4 = 16$  rows, far more than we had in the original sets. If you do the math, you can see why this is a very dangerous join to run against large tables.

## C.R.U.D - CREATE READ UPDATE DELETE - APPs

## INSERT SELECT UPDATE DELETE - SQL

## Application Form Mapping to SQL Columns

HTML  
Container  
on a web  
Page.

CSS  
Cascade  
Style  
Sheets

Positions  
Styles  
HTML  
containers

Javascript  
AJAX  
JQuery  
AngularJS

based on  
Javascript

HTTP(s)

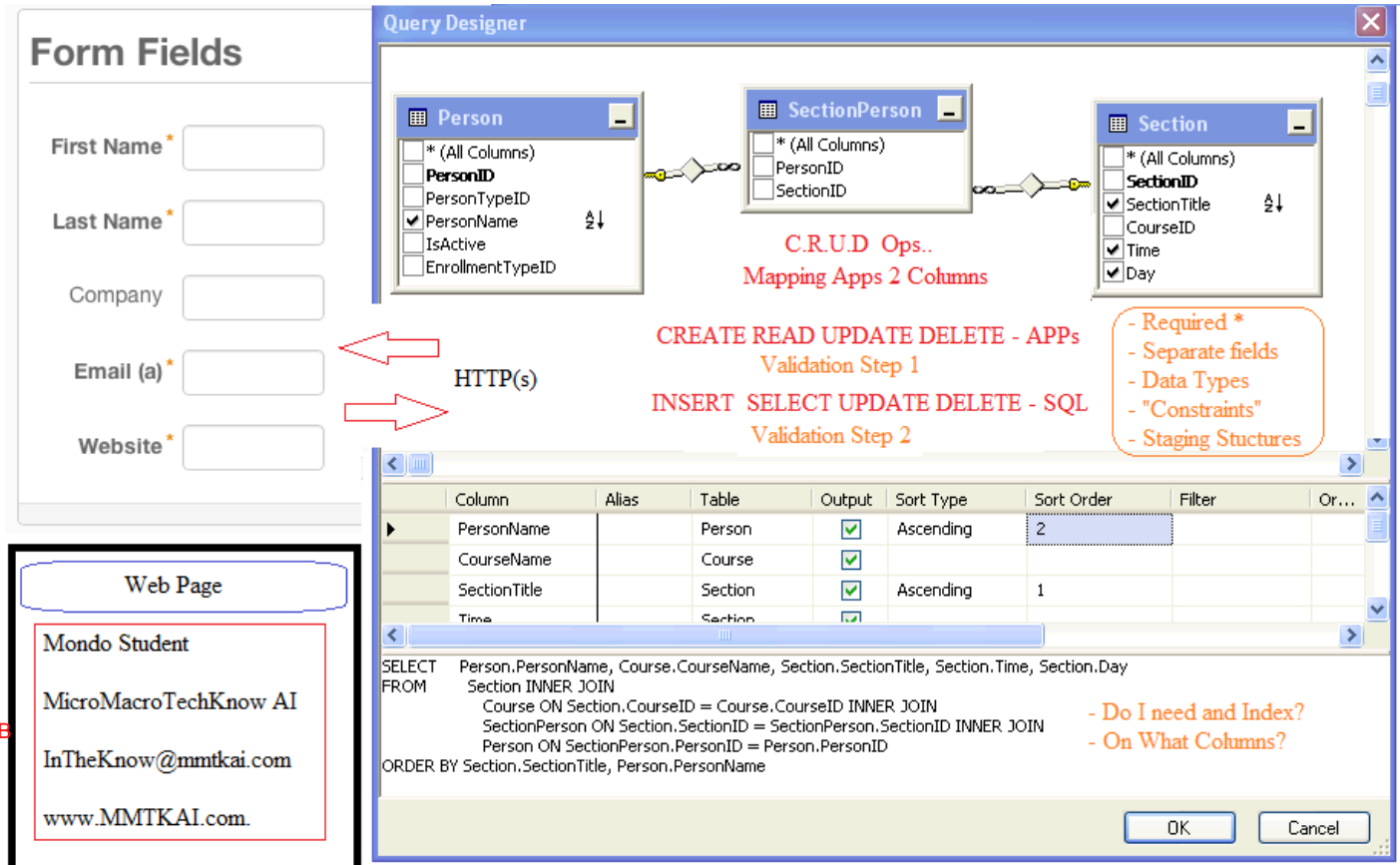
C.R.U.D

server-side

Form  
Handling  
PHP  
ASP - C#/VB

connects to

SQL



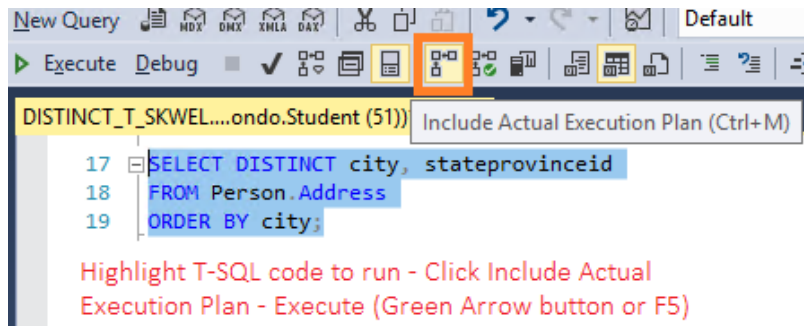
The image shows a web application form on the left and a SQL Query Designer window on the right. The form has fields for First Name, Last Name, Company, Email (a), and Website, each with an asterisk indicating it is required. The SQL Query Designer window shows a query for the Person, SectionPerson, and Section tables. The query is as follows:

```
SELECT Person.PersonName, Course.CourseName, Section.SectionTitle, Section.Time, Section.Day
FROM Section INNER JOIN
      Course ON Section.CourseID = Course.CourseID INNER JOIN
      SectionPerson ON Section.SectionID = SectionPerson.SectionID INNER JOIN
      Person ON SectionPerson.PersonID = Person.PersonID
ORDER BY Section.SectionTitle, Person.PersonName
```

Annotations on the image include:

- Form Fields:** First Name \*, Last Name \*, Company, Email (a) \*, Website \*
- Query Designer:**
  - Person Table:** \* (All Columns), PersonID, PersonName, IsActive, EnrollmentTypeID
  - SectionPerson Table:** \* (All Columns), PersonID, SectionID
  - Section Table:** \* (All Columns), SectionID, SectionTitle, CourseID, Time, Day
  - Relationships:** Person to SectionPerson (1:M), SectionPerson to Section (1:M)
  - Red Text:** C.R.U.D Ops.. Mapping Apps 2 Columns
  - Red Text:** CREATE READ UPDATE DELETE - APPs Validation Step 1
  - Red Text:** INSERT SELECT UPDATE DELETE - SQL Validation Step 2
  - Red Text:** - Required \*
  - Red Text:** - Separate fields
  - Red Text:** - Data Types
  - Red Text:** - "Constraints"
  - Red Text:** - Staging Structures
  - Red Text:** - Do I need and Index?
  - Red Text:** - On What Columns?
- Web Page:** Mondo Student, MicroMacroTechKnow AI, InTheKnow@mmtkai.com, www.MMTKAI.com
- HTTP(s):** Indicated by red arrows pointing from the form to the query designer.

## Using Execution Plans to Optimize Queries



## General "Query" Theory...

1. Reading a query like a sentence. (This is a goal!)
  - a. What is the "Question" being asked by the query?
2. How many rows are in the tables being queried?
  - a. `SELECT * FROM tableName` = total number rows.
3. What is the expected output/result – the "Question" – including number of possible rows.
4. Determining what part of the query executes at what point – this is tricky.
  - a. What is the "Precedence" of any operator or keyword.
  - b.  $A + b * c = D$  – is not always the same as:  $a + B * c = D$  or  $(A + b) * c = D$
  - c. A vs. a = when is something case sensitive and therefore not "the same"?
5. JOINS and Predicates – ON and WHERE specifically behave differently depending on the join type.
6. How are NULL values being "handled" – discarded, filtered, returned....
7. DISTINCT / UNION – returning no duplicates vs. with duplicates (UNION ALL)
8. Sort order (ORDER BY) *my relational set is turned into a "cursor" - because its now an ordered list...*
9. Filtered on... WHERE
10. Matched – ON
11. Do I have to index any additional columns to improve SQL performance?
  - a. Add non-clustered index to columns used in queries – Foreign Key, WHERE, ON, ORDER BY (GROUP BY)... typically columns where SQL works the hardest.
  - b. Discover that with Execution Plans.
12. What are the alias column names in the syntax?
  - a. To be a "Relational" query – must have column names.
13. What are the alias column names for the "Presentation"? *DateTimes or Money data types*
  - a. Do I need spaces in existing column names or new names?
14. Are dates, times, money formatted for the "Presentation"?
15. Does a temporary table need to be used?
  - a. CTE or Common Table Expression for example.
16. Does a View need to be created and then create "simpler" queries for reporting, applications – ad hoc users?
17. What use or combination of functions, aggregates, error trapping might there be?
18. Are there **variables & parameters** anywhere?
  - a. How are variables passed to a parameter (becoming an "argument" or "local variable" to another piece of T-SQL code)?
19. Store Procedures/User Defined Functions (UDFs) – What, When & Why...
20. Clean up any baggage – DROP useless items.... **DELETE AND REPEAT** as often as necessary for things to sink in....
  - a. Where did my hair go?

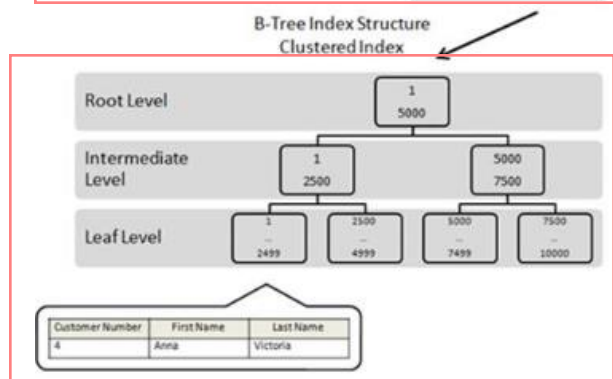
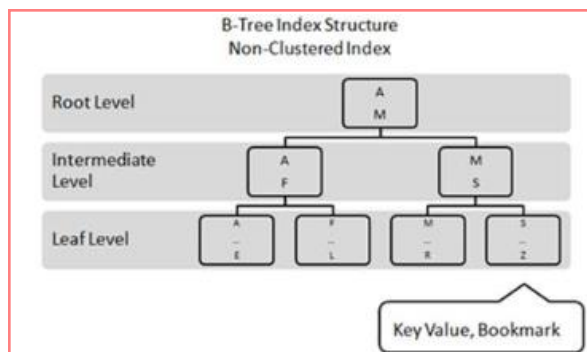
*Variable - a container : Parameter is a private to a function variable*

*When I pass a value it is called an Argument*

## Queries, Indexes, Statistics & Optimizing

When executing a SELECT statement against a table without indexes, SQL Server has to read all records, one by one, to find the records requested by the statement.

An index is an on-disk structure associated with a table or view that speeds retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view. These keys are stored in a structure (B-tree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently.



- **Clustered indexes** sort and store the data rows in the table or view based on their key values. These are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be sorted in only one order.
- **Nonclustered indexes** have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value.
- The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.

Indexes can be helpful for a variety of queries that contain SELECT, UPDATE, DELETE, or MERGE statements.

- Consider the query `SELECT Title, HireDate FROM`

`HumanResources.Employee WHERE EmployeeID = 250` in the AdventureWorks2012 database.

- When this query is executed, the query optimizer evaluates each available method for retrieving the data and selects the most efficient method.
- The method may be a table scan, or may be scanning one or more indexes if they exist.

**Table and index scans should be avoided** as they are resource expensive and require more time. Having a seek operator in the query execution plan is preferred over scan. As the scan operator reads every row, its cost is proportional to the number of table rows.

The seek operator does not read all records, just the specific ones that correspond to the query condition. Whenever possible, use a list of columns in the SELECT statement and a WHERE clause to narrow down the rows and columns returned by the query.

The execution plan describes the sequence of operations, physical and logical, that SQL Server will perform in order to fulfill the query and produce the desired result set.

**There are a number of ways to retrieve an execution plan for a query:**

- **Management Studio provides Display Actual Execution Plan and Display Estimated Execution Plan features, which present the plan in a graphical way.** These features offer the most suitable solution for direct examination and are by far the most often-used approach to display and analyze execution plans. (In this article, I will use graphical plans generated in this way to illustrate my examples.)
- Various SET options, such as SHOWPLAN\_XML and SHOWPLAN\_ALL, return the execution plan as either an XML document describing the plan using a special schema or a rowset with textual description of each of the operations in the execution plan.
- SQL Server Profiler event classes, such as Showplan XML, allow you to gather execution plans of statements collected by a trace.

SQL Server uses a cost-based query optimizer—that is, it tries to generate an execution plan with the lowest estimated cost.

- The estimate is based on the data distribution statistics that are available to the optimizer when it evaluates each table involved in the query.
- If those statistics are missing or outdated, the query optimizer will lack vital information that it needs for the query optimization process and therefore its estimates will likely be off the mark.
- In such cases, the optimizer will choose a less than optimal plan by either overestimating or underestimating the execution costs of different plans.

The query optimizer already updates statistics as necessary to improve the query plan

- **In some cases,** you can improve query performance by using UPDATE STATISTICS or the stored procedure sp\_updatestats to update statistics more frequently than the default updates.
- Updating statistics causes queries to recompile.
- It is recommended to not update statistics too frequently because there is a performance tradeoff between improving query plans and the time it takes to recompile queries.

**One special type of execution plan is called a parallel plan**

- If you are running your query on a server with more than one CPU and your query is eligible for parallelization, a parallel plan may be chosen.
- Control which of your queries can produce parallel plans and how many CPUs each can utilize. You do this by setting the max degree of parallelism option on the server level and overriding it on individual query level with OPTION (MAXDOP n) as needed.

## Query Exercises & Discussions

These examples use the Person.Address table in the AdventureWorks database. (Or any table with the proper indexes on columns used in the queries... you may have to create them – a good exercise.)

The table definition has a clustered index created on the AddressID column and a nonclustered key on the StateProvinceID column.

```
CREATE TABLE [Person].[Address] (
    [AddressID] [int] IDENTITY(1,1) NOT FOR REPLICATION NOT NULL,
    [AddressLine1] [nvarchar](60) NOT NULL,
    [AddressLine2] [nvarchar](60) NULL,
    [City] [nvarchar](30) NOT NULL,
    [StateProvinceID] [int] NOT NULL,
    [PostalCode] [nvarchar](15) NOT NULL,
    [SpatialLocation] [geography] NULL,
    [rowguid] [uniqueidentifier] ROWGUIDCOL NOT NULL,
    [ModifiedDate] [datetime] NOT NULL,
    CONSTRAINT [PK_Address_AddressID] PRIMARY KEY CLUSTERED
    (
        [AddressID] ASC
    ) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

CREATE NONCLUSTERED INDEX [IX_Address_StateProvinceID] ON [Person].[Address]
(
    [StateProvinceID] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,
DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON
[PRIMARY]
GO
```

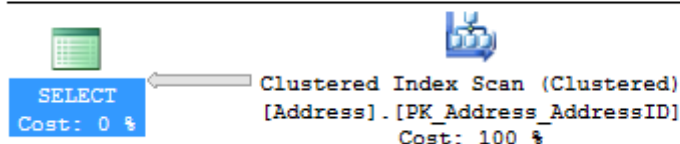
### SELECT \* using the Clustered Index also known as the Primary Key column

```
SELECT * FROM Person.Address
```

The estimated and actual SQL Server query execution plans are identical. As the query is simple, there's no doubt how it will be executed

Query 1: Query cost (relative to the batch): 100%

```
SELECT * FROM Person.Address
```



As the query execution plans are read from right to left, we'll start with the **Clustered Index Scan** and it takes 100% of the whole execution plan time. The Select icon represents execution of the SELECT statement, and as shown in the plan, its cost is 0%

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	19614
Actual Number of Batches	0
Estimated I/O Cost	0.257199
Estimated Operator Cost	0.278931 (100%)
Estimated CPU Cost	0.0217324
Estimated Subtree Cost	0.278931
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	19614
Estimated Row Size	4241 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0
<b>Object</b>	
[AdventureWorks2012].[Person].[Address].	
[PK_Address_AddressID]	
<b>Output List</b>	
[AdventureWorks2012].[Person].[Address].AddressID,	
[AdventureWorks2012].[Person].[Address].AddressLine1,	
[AdventureWorks2012].[Person].[Address].AddressLine2,	
[AdventureWorks2012].[Person].[Address].City,	
[AdventureWorks2012].[Person].[Address].StateProvinceID,	
[AdventureWorks2012].[Person].[Address].PostalCode,	
[AdventureWorks2012].[Person].[Address].SpatialLocation,	
[AdventureWorks2012].[Person].[Address].rowguid,	
[AdventureWorks2012].[Person].[Address].ModifiedDate	

Although the estimated cost values have no units, it's the estimated time in seconds, needed to execute the analyzed query.

The cost is calculated by Query Optimizer. These values are compared to the costs calculated for alternative plans, and the lowest cost plan is shown as the optimal one

The Person.Address table has the PK\_Address\_AddressID clustered index created on the primary key.

When SELECT is executed on a table with a clustered index, the table is scanned in order to find the needed records.

The Object in the clustered index scan tooltip is PK\_Address\_AddressID, which means that it's used to scan the table rows.

As all columns are returned, and the WHERE clause is not used, the database engine must scan all index values



**SELECT statement executed on a table without a clustered index.**

Create a copy of the Person.Address table, and removed all indexes except the non-clustered index on the StateProvinceID column. The records in the Address and Address1 tables are identical, so we can use the estimated costs for comparison.

```
SELECT * FROM Person.Address1
```

The actual and estimated query execution plans are again identical

Query 1: Query cost (relative to the batch): 100%  
SELECT \* FROM Person.Address1

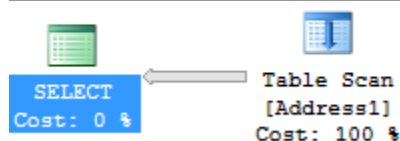


Table Scan	
Scan rows from a table.	
Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	19614
Actual Number of Batches	0
Estimated I/O Cost	0.26831
Estimated Operator Cost	0.290043 (100%)
Estimated CPU Cost	0.0217324
Estimated Subtree Cost	0.290043
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	19614
Estimated Row Size	4241 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0
<b>Object</b>	
[AdventureWorks2012].[Person].[Address1]	
<b>Output List</b>	
[AdventureWorks2012].[Person].[Address1].AddressID,	
[AdventureWorks2012].[Person].[Address1].AddressLine1, [AdventureWorks2012].	
[Person].[Address1].AddressLine2,	
[AdventureWorks2012].[Person].[Address1].City,	
[AdventureWorks2012].[Person].[Address1].StateProvinceID, [AdventureWorks2012].	
[Person].[Address1].PostalCode, [AdventureWorks2012].	
[Person].[Address1].SpatialLocation,	
[AdventureWorks2012].[Person].[Address1].rowguid,	
[AdventureWorks2012].[Person].[Address1].ModifiedDate	

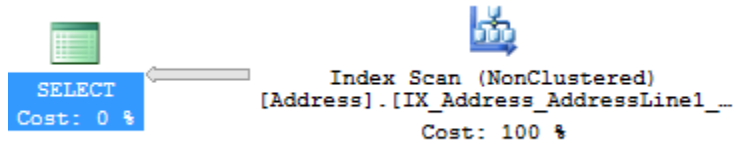
Unlike in the previous example, there is the **Table Scan** icon now shown instead of the **Clustered Index Scan**.

The difference in the query execution plan operators is caused by having a non-clustered index instead of a clustered one.

When all rows from a table are returned (a SELECT statement without a WHERE clause) and no clustered index is created on the table, the engine has to go through the whole table, scanning row after row.

### Use a list of columns in the SELECT statement, instead of SELECT \*

```
SELECT AddressID, AddressLine1, AddressLine2
FROM Person.Address
```



Index Scan (NonClustered)	
Scan a nonclustered index, entirely or only a range.	
Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	19614
Actual Number of Batches	0
Estimated I/O Cost	0.158681
Estimated Operator Cost	0.180413 (100%)
Estimated Subtree Cost	0.180413
Estimated CPU Cost	0.0217324
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	19614
Estimated Row Size	137 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0
<b>Object</b>	
[AdventureWorks2012].[Person].[Address].	
[IX_Address_AddressLine1_AddressLine2_City_StateProv	
incedID_PostalCode]	
<b>Output List</b>	
[AdventureWorks2012].[Person].[Address].AddressID,	
[AdventureWorks2012].[Person].	
[Address].AddressLine1, [AdventureWorks2012].	
[Person].[Address].AddressLine2	

The SQL Server query execution plan is similar to the one in the previous example. The difference is that now, a **NonClustered Index Scan** is used

Note the shorter **Output List** and up to 50% lower estimated cost values

## SELECT with WHERE on a clustered index

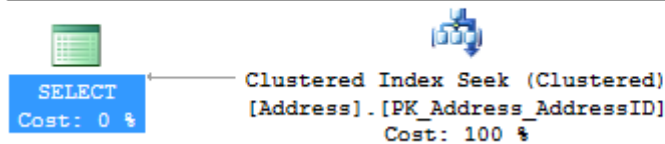
To narrow down the number of rows returned by the query, we will add the WHERE clause with a condition for the table clustered index.

```
SELECT *
FROM Person.Address
where AddressID = 7
```

The SQL Server query execution plan now contains the Clustered index seek.

Query 1: Query cost (relative to the batch): 100%

```
SELECT * FROM [Person].[Address] WHERE [AddressID]=@1
```



Clustered Index Seek (Clustered)	
Scanning a particular range of rows from a clustered index.	
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	1
Actual Number of Batches	0
Estimated Operator Cost	0.0032831 (100%)
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Subtree Cost	0.0032831
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	1
Estimated Row Size	4241 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0
<b>Object</b>	
[AdventureWorks2012].[Person].[Address].	
[PK_Address_AddressID]	
<b>Output List</b>	
[AdventureWorks2012].[Person].[Address].AddressID,	
[AdventureWorks2012].[Person].[Address].AddressLine1,	
[AdventureWorks2012].[Person].[Address].AddressLine2,	
[AdventureWorks2012].[Person].[Address].City,	
[AdventureWorks2012].[Person].[Address].StateProvinceID,	
[AdventureWorks2012].[Person].[Address].PostalCode,	
[AdventureWorks2012].[Person].[Address].SpatialLocation,	
[AdventureWorks2012].[Person].[Address].rowguid,	
[AdventureWorks2012].[Person].[Address].ModifiedDate	
<b>Seek Predicates</b>	
Seek Keys[1]: Prefix: [AdventureWorks2012].[Person].	
[Address].AddressID = Scalar Operator(CONVERT_IMPLICIT	
(int,[@1],0))	

As its tooltip explains, the database engine scans a particular range of rows from the clustered index.

It seeks for the clustered index, not scans for it, therefore we can expect lower query cost.

The engine searches for the specific key values and can quickly find them, as the clustered index also sorts the data so the search is more efficient.

Finding the right record in such ordered tables is quick and resource inexpensive, which is clearly shown by the cost numbers.

If you now compare the estimated operator, I/O, and CPU costs with the costs for the same query without WHERE, you can notice that the values when the WHERE clause is used are smaller by two orders of magnitude

## SELECT with WHERE on a nonclustered index

A similar example is to use the WHERE clause with the condition on the column other than the clustered index, for example on a unique nonclustered index.

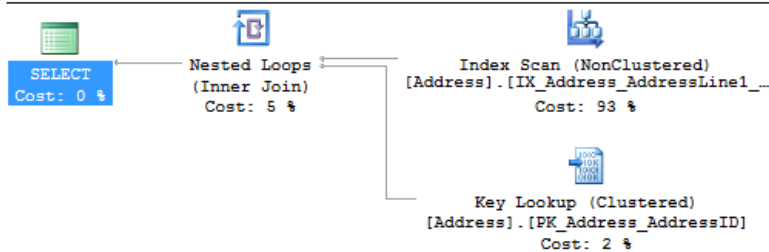
```
CREATE UNIQUE NONCLUSTERED INDEX
[IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode] ON
[Person].[Address]
(
    [AddressLine1] ASC,
    [AddressLine2] ASC,
    [City] ASC,
    [StateProvinceID] ASC,
    [PostalCode] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,
IGNORE_DUP_KEY = OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
GO
```

```
SELECT *
FROM Person.Address
where City = 'New York'
```

Query 1: Query cost (relative to the batch): 100%

SELECT \* FROM [Person].[Address] WHERE [City]=@1

Missing Index (Impact 89.952): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Person].[A...



Now, there are both the **Nonclustered Index Scan** and **Key Lookup**.

- Keep in mind that the query execution plans are read right to left, top to bottom.

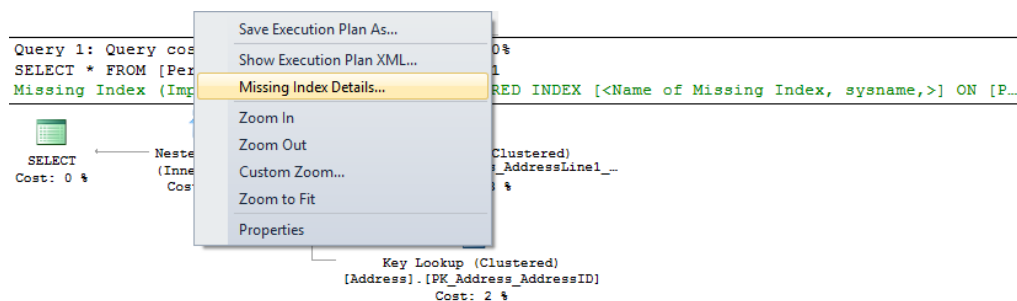
The first operator here is the **Non-Clustered Index Scan**.

Index Scan (NonClustered)	
Scan a nonclustered index, entirely or only a range.	
Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	2
Actual Number of Batches	0
Estimated I/O Cost	0.158681
Estimated Operator Cost	0.180413 (93%)
Estimated Subtree Cost	0.180413
Estimated CPU Cost	0.0217324
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	1.16667
Estimated Row Size	177 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1
<b>Predicate</b>	
[AdventureWorks2012].[Person].[Address].[City]	
=N'New York'	
<b>Object</b>	
[AdventureWorks2012].[Person].[Address].	
[IX_Address_AddressLine1_AddressLine2_City_StatePro	
vinceID_PostalCode]	
<b>Output List</b>	
[AdventureWorks2012].[Person].[Address].AddressID,	
[AdventureWorks2012].[Person].	
[Address].AddressLine1, [AdventureWorks2012].	
[Person].[Address].AddressLine2,	
[AdventureWorks2012].[Person].[Address].City,	
[AdventureWorks2012].[Person].	
[Address].StateProvinceID, [AdventureWorks2012].	
[Person].[Address].PostalCode	

Key Lookup (Clustered)	
Uses a supplied clustering key to lookup on a table that has a clustered index.	
Physical Operation	Key Lookup
Logical Operation	Key Lookup
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	2
Actual Number of Batches	0
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.0034535 (2%)
Estimated CPU Cost	0.0001581
Estimated Subtree Cost	0.0034535
Number of Executions	2
Estimated Number of Executions	1.16667
Estimated Number of Rows	1
Estimated Row Size	4059 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3
<b>Object</b>	
[AdventureWorks2012].[Person].[Address].	
[PK_Address_AddressID]	
<b>Output List</b>	
[AdventureWorks2012].[Person].	
[Address].SpatialLocation, [AdventureWorks2012].	
[Person].[Address].rowguid, [AdventureWorks2012].	
[Person].[Address].ModifiedDate	
<b>Seek Predicates</b>	
Seek Keys[1]: Prefix: [AdventureWorks2012].[Person].	
[Address].AddressID = Scalar Operator	
([AdventureWorks2012].[Person].[Address].	
[AddressID])	

The query execution plans displays the “Missing index (Impact 89.952): CREATE NONCLUSTERED INDEX [<Name Of Missing Index, sysname,>] ON [Person].Address([City])” message

- This is not an error, or warning, you should consider this message as an advice what you can do to improve the query execution performance by almost 90%.
- To see the recommended index, right-click the plan and select the **Missing Index details** option.



The option returns the index details along with T-SQL for creating it.

```

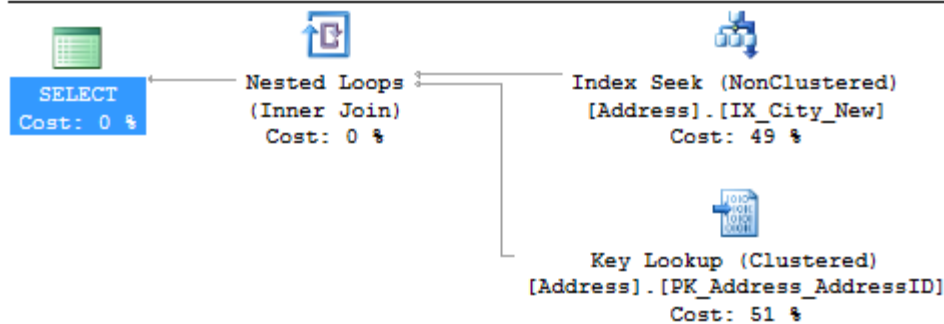
/*
Missing Index Details from SELECT Address4.sql - FUJITSU\SQL2012.AdventureWorks2012 (sa
(56))
The Query Processor estimates that implementing the following index could improve the
query cost by 89.952%.
*/
/*
USE [AdventureWorks2012]
GO
CREATE NONCLUSTERED INDEX [<name of missing index, sysname,>]
ON [Person].[Address] ([City])
GO
*/

```

All you have to do is 1) Enter the index name 2) Remove the comments 3) Execute code...

Query 1: Query cost (relative to the batch): 100%

SELECT \* FROM [Person].[Address] WHERE [City]=@1



- As you can see, the operators are the same, but there's no warning anymore and the cost is distributed almost equally between the **Index Seek** and **Key Lookup**.
- While the **Key Lookup** cost stayed the same, the **Index Seek** cost is significantly reduced.

Index Seek (NonClustered)		Key Lookup (Clustered)	
Scan a particular range of rows from a nonclustered index.		Uses a supplied clustering key to lookup on a table that has a clustered index.	
Physical Operation	Index Seek	Physical Operation	Key Lookup
Logical Operation	Index Seek	Logical Operation	Key Lookup
Actual Execution Mode	Row	Actual Execution Mode	Row
Estimated Execution Mode	Row	Estimated Execution Mode	Row
Storage	RowStore	Storage	RowStore
Actual Number of Rows	2	Actual Number of Rows	2
Actual Number of Batches	0	Actual Number of Batches	0
Estimated Operator Cost	0.0032833 (49%)	Estimated I/O Cost	0.003125
Estimated I/O Cost	0.003125	Estimated Operator Cost	0.0034535 (51%)
Estimated CPU Cost	0.0001583	Estimated CPU Cost	0.0001581
Estimated Subtree Cost	0.0032833	Estimated Subtree Cost	0.0034535

It is highly recommended **not** to create all indexes that are reported as missing by Query Optimizer in the query execution plans. Keep in mind indexing has benefits as well as downsides so it is necessary to determine whether this index will really improve overall performance.

### General Discussion on Execution Plan activity

1. The **Index Scan** output list shows only some of the table columns.
2. These are the columns specified in the index:
  - a. **IX\_Address\_AddressLine1\_AddressLine2\_City\_StateProvinceID\_PostalCode**
  - b. With columns - AddressLine1, AddressLine2, City, StateProvinceID, and PostalCode.
3. The clustered index and primary key column **AddressID** is always returned.
4. The SELECT statement executed must return all table columns.
5. The rest of the columns are read by the **Key Lookup** operator.
6. The key value **AddressID** is used in the **Key Lookup**.
7. The **Key Lookup** returns the rest of the **Person.Address** columns.

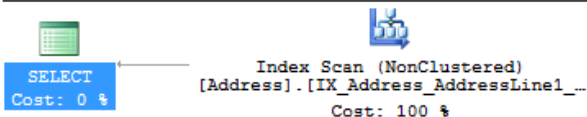
A **Key Lookup** indicates that obtaining all records is done in two steps.

To return all records in a single step, all needed columns should be returned by the **Index Scan**.

- The solution is to specify the list of columns returned by the **Index Scan**.

```
SELECT AddressLine1, AddressLine2, City, StateProvinceID, PostalCode
FROM Person.Address
WHERE City = 'New York'
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [AddressLine1],[AddressLine2],[City],[StateProvinceID],[PostalCode] FROM [Person].[Address]...
Missing Index (Impact 97.9636): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [...
```



Whenever there is a **Key Lookup** operator, it's followed by the **Nested Loops** which is actually a JOIN operator which combines the data returned by the **Index Seek** and **Key Lookup** operators.

- As shown in the tooltips, the **Index Seek** is performed only once (**Number of executions** = 1) and it returns 16 rows (**Actual number of rows**).
- The **Key Lookup** operator uses the 16 rows returned by the **Index Seek** and looks up for the records in the table. A lookup is performed for every row returned by the **Index Seek**, in this example 16 times, which is presented by the **Key Lookup Number of executions**.
- For a small number of rows returned by **Index Seek**, this means a small number of executions by the **Key Lookup**.
- When the **Key Lookup** output is indexed, the time needed to perform a lookup is shorter, so this is a scenario with a small cost.
- In case the **Index Seek** provides all output columns, the **Key Lookup** is not needed as no additional columns should be retrieved, and the **Nested Loops** is not needed as there are no data sets to combine.



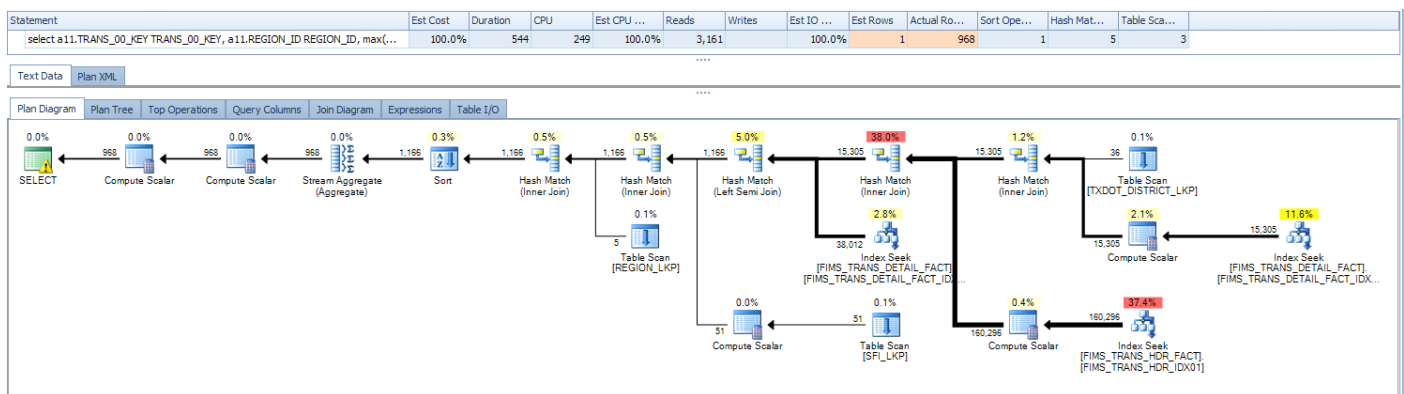
## Analyzing an Execution Plan

Execute this query using Management Studio with the Include Actual Execution Plan option turned on in using Adventure Works database:

```
SELECT c.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od
JOIN Sales.SalesOrderHeader oh
ON od.SalesOrderID=oh.SalesOrderID
JOIN Sales.Customer c ON oh.CustomerID=c.CustomerID
GROUP BY c.CustomerID
```

As a result, I see the execution plan depicted in **Figure 1**.

1. This simple query calculates the total amount of orders placed by each customer of Adventure Works.
2. Graphical execution plans should be read from top to bottom and from right to left.
3. Each icon represents a logical and physical operation performed, and arrows show data flow between operations.
4. The thickness of the arrows represents the number of rows being passed between operations—the thicker the arrow, the more rows involved. If you place your pointer over one of the operator icons, a yellow ToolTip (like the one shown in **Figure 2**) will display details of that particular operation.



## Sample execution plan

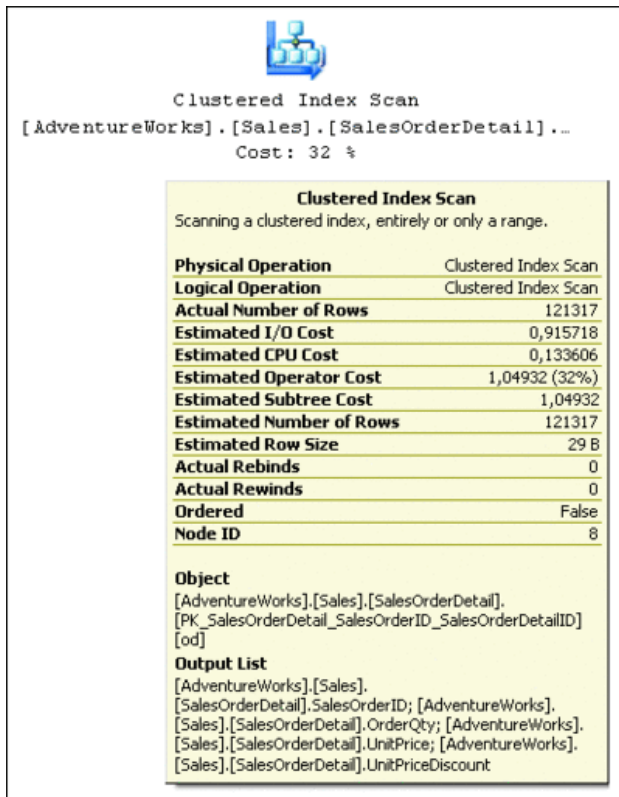


Figure 2 Details about an operation

Look at each of the operators to analyze the sequence of steps performed:

1. The database engine performs a Clustered Index Scan operation on the Sales.Customer table and returns the CustomerID column for all the rows in that table.
2. It then performs an Index Scan (meaning a non-clustered index scan) on one of the indexes in the Sales.SalesOrderHeader table. This is an index on the CustomerID column, but it also implicitly includes the SalesOrderID column (the table clustering key). The values from both of those columns are returned by the scan.
3. Output from both scans is joined on the CustomerID column using the Merge Join physical operator. (This is one of three possible physical ways of performing a logical join operation. It's fast but requires both inputs to be sorted on a joined column. In this case, both scan operations have already returned the rows sorted on CustomerID so there is no need to perform the additional sort operation).
4. Next, the database engine performs a scan of the clustered index on the table Sales.SalesOrderDetail, retrieving the values of four columns (SalesOrderID, OrderQty, UnitPrice, and UnitPriceDiscount) from all rows in this table. (There were 123,317 rows estimated to be returned by this operation and that number was actually returned, as you can see from the Estimated Number of Rows and Actual Number of Rows properties in **Figure 2**—so the estimate was very accurate.)
5. Rows produced by the clustered index scan are passed to the first Compute Scalar operator so that the value of the computed column LineTotal can be calculated for each row, based on the OrderQty, UnitPrice, and UnitPriceDiscount columns involved in the formula.
6. The second Compute Scalar operator applies the ISNULL function to the result of the previous calculation, as required by the computed column formula. This completes the calculation of the LineTotal column and returns it, along with column SalesOrderID, to the next operator.
7. The output of the Merge Join operator in Step 3 is joined with the output of the Compute Scalar operator from Step 6, using the Hash Match physical operator.

- Another Hash Match operator is then applied to group rows returned from Merge Join by the CustomerID column value and calculated SUM aggregate of the LineTotal column.
- The last node, SELECT, is not a physical or logical operator but rather a place holder that represents the overall query results and cost.

This execution plan had an estimated cost of 3,31365 (as shown in **Figure 3**).

- When executed with STATISTICS I/O ON, the query reported performing a total of 1,388 logical read operations across the three tables involved.
- The percentages displayed under each of the operators represent the cost of each individual operator relative to the overall estimated cost of the whole execution plan.
- Looking at the plan in **Figure 1**, you can see that most of the total cost of the entire execution plan is associated with the following three operators: The Clustered Index Scan of the Sales.SalesOrderDetail table and the two Hash Match operators.

Change the query that will eliminate two of the operators altogether.

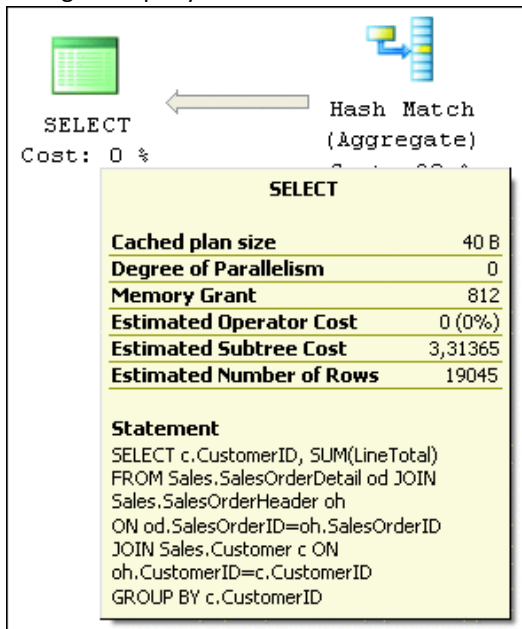


Figure 3 **Total estimated execution cost of the query**

- The only column returned from the Sales.Customer table is CustomerID.
- This column is also included as a foreign key in the Sales.SalesOrderHeaderTable.
- We can completely eliminate the Customer table from the query without changing the logical meaning or the result produced by our query by using this code:

```
SELECT oh.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od JOIN Sales.SalesOrderHeader oh
ON od.SalesOrderID=oh.SalesOrderID
GROUP BY oh.CustomerID
```

This results in a different execution plan, which is shown in **Figure 4**.

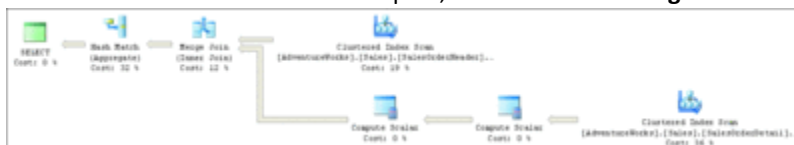


Figure 4 **The execution plan after eliminating Customer table from the query** (Click the image for a larger view)

- **Two operations were eliminated entirely**—the Clustered Index Scan on Customer table and the Merge Join between Customer and SalesOrderHeader—and the Hash Match join was replaced by the much more efficient Merge Join.

In order to use Merge Join between the SalesOrderHeader and SalesOrderDetail tables, the rows from both tables had to be returned sorted by the join column SalesOrderID.

To achieve this, the query optimizer decided to perform a Clustered Index Scan on the SalesOrderHeader table, instead of using the Non-Clustered Index Scan, which would have been cheaper in terms of the I/O involved.

This is a good example of how the query optimizer works in practice—since the cost savings from changing the physical way of performing the join operation were more than the additional I/O cost generated by the Clustered Index Scan, the query optimizer chose the resulting combination of operators because it gave the lowest total estimated execution cost possible.

Even though the number of logical reads went up (to 1,941), the CPU time being consumed was actually smaller and the estimated execution cost of this query was down about 13 percent (2,89548).

#### To further improve the performance of this query.

1. Examine the Clustered Index Scan of the SalesOrderHeader table, which has become the most expensive operator in this execution plan.
2. Since only two columns are needed from this table to fulfill the query.
3. Create a non-clustered index that contains just those two columns
4. Replacing the scan of the whole table with a scan of the much smaller non-clustered index.

The index definition might look something like this:

```
CREATE INDEX IDX_OrderDetail_OrderID_TotalLine
ON Sales.SalesOrderDetail (SalesOrderID) INCLUDE (LineTotal)
```

Notice that the index created includes a computed column. This is not always possible, depending of the definition of the computed column.

After creating this index and executing the same query, I get the new execution plan shown in **Figure 5**.

















Figure 5 **Optimized execution plan**

1. The clustered index scan on the SalesOrderDetail table has been replaced by a non-clustered index scan with a significantly lower I/O cost.
2. Also eliminated is one of the Compute Scalar operators.
3. As the index includes an already calculated value of the LineTotal column.
4. The estimated execution plan cost is now 2,28112 and the query performs 1,125 logical reads when executed.

## Execution Plan Icons

- What columns get indexed or not for best query “Performance Cost”?
- Is the SQL server using the best “Path” to the data being manipulated?

cache - storage or staging area  
that we dont see...

Icon	Operator	Description
	Table Scan	Scan of a whole table stored as a heap. A table can be organized as a heap or as a clustered index.
	Clustered Index Scan	Scan of a whole table stored as a clustered index. Indexes are stored as balanced trees.
	Clustered Index Seek	SQL Server seeks for the first value in the seek argument (for example, a column value in the WHERE clause) in a clustered index and then performs a partial scan.
	Index Scan	Scan of a whole nonclustered index.
	Index Seek	SQL Server seeks for the first value in the seek argument (for example, a column value in the WHERE clause) in a nonclustered index and then performs a partial scan.
	RID Lookup	Lookup for a single row in a table stored as a heap by using its row identifier (RID).
	Key Lookup	Lookup for a single row in a table stored as a clustered index by using the key of the index.
	Hash Match Join	Joins that use the Hash algorithm.
	Merge Join	Joins that use the Merge algorithm.
	Nested Loops	Joins that use the Nested Loops algorithm.
	Stream Aggregate	Aggregation of ordered rows.
	Hash Match Aggregate	Hash algorithm used for aggregating. Note that the icon is the same as the icon for the Hash Match Join; however, in an execution plan, text below the icons gives you information about whether the operator performed a join or an aggregate.
	Filter	Filters rows based on a predicate (for example, a predicate of the WHERE clause).
	Sort	Sort of incoming rows.

You can rely on statistics that are kept for all queries in the execution plan cache and query them using dynamic management views.

This **example** contains a query that **shows you both the text and the execution plan of the 20 queries in your cache with the highest accumulated number of logical reads**. This *will only show the queries that have their plans cached at the time you run the query*. If something is not cached, you will not see it.

### Identifying top 20 most expensive queries in terms of read I/O

```
SELECT TOP 20 SUBSTRING(qt.text, (qs.statement_start_offset/2)+1,
    ((CASE qs.statement_end_offset
        WHEN -1 THEN DATALength(qt.text)
        ELSE qs.statement_end_offset
        END - qs.statement_start_offset)/2)+1),
    qs.execution_count,
    qs.total_logical_reads, qs.last_logical_reads,
    qs.min_logical_reads, qs.max_logical_reads,
    qs.total_elapsed_time, qs.last_elapsed_time,
    qs.min_elapsed_time, qs.max_elapsed_time,
    qs.last_execution_time,
    qp.query_plan
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
WHERE qt.encrypted=0
ORDER BY qs.total_logical_reads DESC
```

**Once you have identified poor performers**, take a look at their query plans and search for ways to improve their performance by using some of the indexing techniques

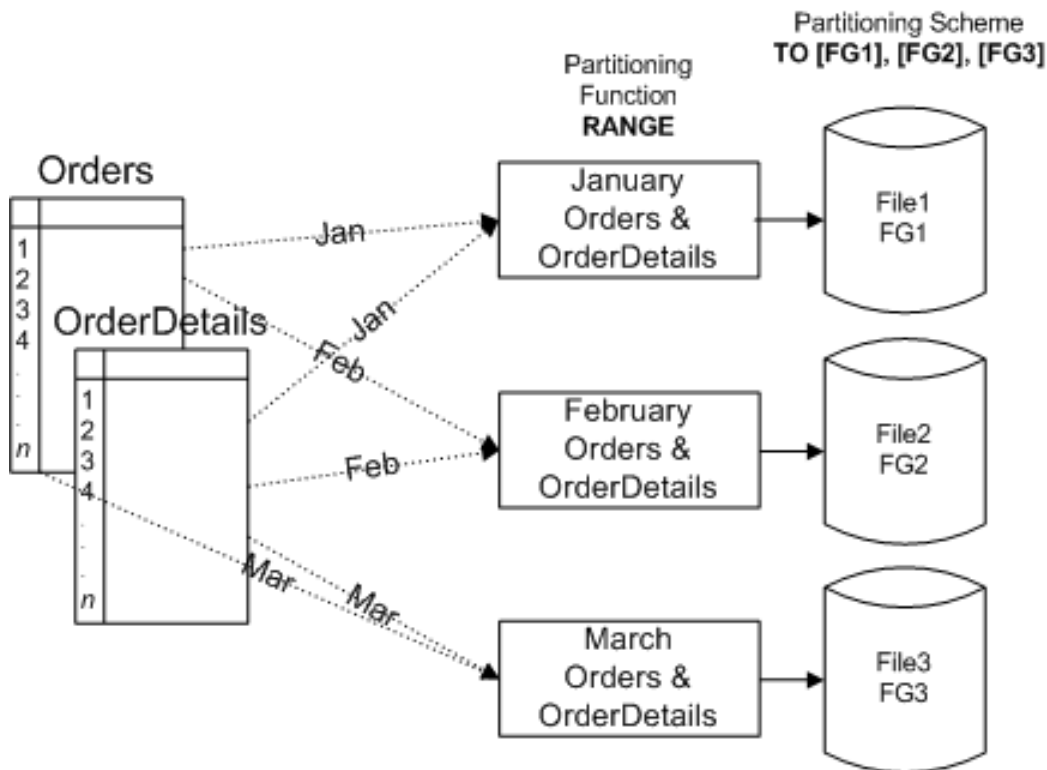
### Rules for Queries

1. Avoid Multiple Joins in a Single Query
  - a. Try to avoid writing a SQL query using multiple joins that includes outer joins, cross apply, outer apply and other complex sub queries. It reduces the choices for Optimizer to decide the join order and join type.
2. Creation and Use of Indexes
  - a. We are aware of the fact that Index can magically reduce the data retrieval time but have a reverse effect on DML operations, which may degrade query performance.
3. Eliminate Cursors from the Query
  - a. Try to remove cursors from the query and use set-based query; set-based query is more efficient than cursor-based. If there is a need to use cursor than avoid dynamic cursors as it tends to limit the choice of plans available to the query optimizer. For example, dynamic cursor limits the optimizer to using nested loop joins.
4. Avoid Use of Non-correlated Scalar Sub Query
  - a. You can re-write your query to remove non-correlated scalar sub query as a separate query instead of part of the main query and store the output in a variable, which can be referred to in the main query or later part of the batch.
5. Avoid Multi-Statement Table Valued Functions (TVFs)
  - a. Multi-statement TVFs are more costly than inline TVFs. SQL Server expands inline TVFs into the main query like it expands views but evaluates multi-statement TVFs in a separate context from the main query and materializes the results of multi-statement into temporary work tables.
6. Creation and Use of Indexes
  - a. We are aware of the fact that Index can magically reduce the data retrieval time but have a reverse effect on DML operations, which may degrade query performance.

7. Understand the Data
  - a. Understand the data, its type and how queries are being performed to retrieve the data before making any decision to create an index. If you understand the behavior of data thoroughly, it will help you to decide which column should have either a clustered index or non-clustered index. If a clustered index is not on a unique column then SQL Server will maintain uniqueness by adding a unique identifier to every duplicate key, which leads to overhead. To avoid this type of overhead, choose the column correctly or make the appropriate changes.
8. Create a Highly Selective Index
  - a. Selectivity define the percentage of qualifying rows in the table (qualifying number of rows/total number of rows). If the ratio of the qualifying number of rows to the total number of rows is low, the index is highly selective and is most useful. A non-clustered index is most useful if the ratio is around 5% or less, which means if the index can eliminate 95% of the rows from consideration. If index is returning more than 5% of the rows in a table, it probably will not be used; either a different index will be chosen or created or the table will be scanned.
9. Position a Column in an Index
  - a. Order or position of a column in an index also plays a vital role to improve SQL query performance. An index can help to improve the SQL query performance if the criteria of the query matches the columns that are left most in the index key. As a best practice, most selective columns should be placed leftmost in the key of a non-clustered index.
10. Drop Unused Indexes
  - a. Dropping unused indexes can help to speed up data modifications without affecting data retrieval. Also, you need to define a strategy for batch processes that run infrequently and use certain indexes. In such cases, creating indexes in advance of batch processes and then dropping them when the batch processes are done helps to reduce the overhead on the database.
11. Statistic Creation and Updates
  - a. You need to take care of statistic creation and regular updates for computed columns and multi-columns referred in the query; the query optimizer uses information about the distribution of values in one or more columns of a table statistics to estimate the cardinality, or number of rows, in the query result. These cardinality estimates enable the query optimizer to create a high-quality query plan.
12. Revisit Your Schema Definitions
  - a. Last but not least, revisit your schema definitions; keep on eye out that appropriate FOREIGN KEY, NOT NULL and CHECK constraints are in place or not. Availability of the right constraint on the right place always helps to improve the query performance, like FOREIGN KEY constraint helps to simplify joins by converting some outer or semi-joins to inner joins and CHECK constraint also helps a bit by removing unnecessary or redundant predicates.



Where do I find it? Do I know my “Containers” & “Boundaries”?



## References

1. Performance Monitoring and Tuning Tools:
  - a. <https://docs.microsoft.com/en-us/sql/relational-databases/performance/performance-monitoring-and-tuning-tools?view=sql-server-2017>
2. Clustered and Nonclustered Indexes Described:
  - a. <https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described>
3. Live Query Statistics:
  - a. <https://docs.microsoft.com/en-us/sql/relational-databases/performance/live-query-statistics?view=sql-server-2017>
4. UPDATE STATISTICS (Transact-SQL):
  - a. <https://docs.microsoft.com/en-us/sql/t-sql/statements/update-statistics-transact-sql>

## Reference (pre-2016)

1. Optimizing SQL Server Query Performance (2007 but a good discussion):
  - a. <https://technet.microsoft.com/en-us/library/2007.11.sqlquery.aspx>
2. Execution Plan Basics:
  - a. <https://www.red-gate.com/simple-talk/sql/performance/execution-plan-basics/>
3. Examples with the SELECT statement:
  - a. <https://www.sqlshack.com/sql-server-query-execution-plans-examples-select-statement/>
4. Examples with the WHERE clause:
  - a. <https://www.sqlshack.com/sql-server-query-execution-plans-where-clause/>
5. space