# Transact-SQL (T-SQL): Coding Standards

1. What is Transact-SQL?

Transact-SQL is the Microsoft implementation of the SQL standard.

1. Why do we need to follow Coding Standards / Best Practices?

* Readability
* Reusability
* Consistency
* Maintainability
* Accuracy
* Performance
* Standardization

1. **Formatting and Indentation Guidelines:**
   1. There’s no best SQL formatting style, but what matters is consistency & readability.
   2. Following proper SQL Formatting will help us to read & understand the SQL quickly, when compared to the poorly formatted SQL code.
   3. Add appropriate comments to describe what the SQL does, apart from the developer name and revision/modification history including the date.
   4. Proper indentation and starting each major SQL statement in a separate line will help us to structure the SQL, for better visualization and readability.

Please see below example:

SELECT

FirstName,

LastName

FROM

HumanResources.vEmployee

WHERE

StateProvinceName = 'Washington'

* 1. Even though the SQL keywords are case-insensitive (SELECT, FROM, WHERE, etc.), let us have a practice of having the keywords in uppercase, the data types in lowercase, and the object names in Pascal Case.

Please see below example:

IF OBJECT\_ID('Books', 'U') IS NOT NULL

DROP TABLE Books

GO

CREATE TABLE Books

(

BookID int NOT NULL IDENTITY,

BookTitle nvarchar(50) NOT NULL,

AuthorName nvarchar(400) NOT NULL,

CONSTRAINT PK\_Books\_ID PRIMARY KEY CLUSTERED (BookID ASC)

)

GO

1. **Best Practices:**
   1. Please avoid using wildcard (\*) in the SELECT list, instead list the column names.

SELECT \*

FROM Books

Vs

SELECT

BookTitle, AuthorName

FROM

**Books**

* 1. Please avoid referencing the columns with the corresponding integer value.

SELECT ProductID, Name

FROM [SalesLT].[Product]

ORDER BY ***2*** DESC

Vs

SELECT

ProductID, Name

FROM

[SalesLT].[Product]

ORDER BY

***Name*** DESC

* 1. We can insert data into a table using an INSERT statement, without specifying the columns of the respective table. However, if the table definition changes in future, this INSERT statement would fail. So, it is better to include column names in the INSERT statement.

INSERT [dbo].[Books]

VALUES ('SQL Server Best Practies', 'DCReddy')

Vs

INSERT

[dbo].[Books] ([BookTitle],[AuthorName])

VALUES

('SQL Server Best Practies 2018', 'CDKReddy')

* 1. Please give meaningful names / abbreviations for the column and table *aliases*.

Please see below example:

SELECT

(Con.FirstName + ' ' + Con.LastName) AS FullName,

Emp.LoginID,

Emp.Title

FROM

HumanResources.Employee AS Emp

INNER JOIN Person.Contact AS Con

ON Emp.ContactID = Con.ContactID

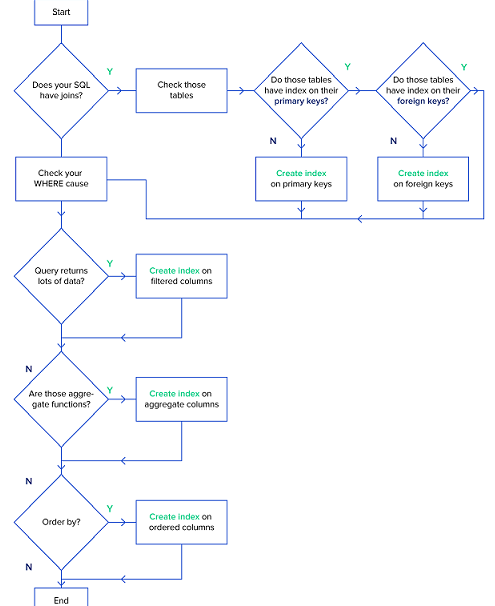
ORDER BY

Con.LastName

1. **Naming Conventions:**
   1. Please see below naming conventions that are suggested for the objects to develop using T-SQL:

| **Object** | **Notation** | **Prefix** | **Abbreviation** | **Example** |
| --- | --- | --- | --- | --- |
| Database | PascalCase | No | Yes | HumanResources |
| Database Trigger | PascalCase | DTR\_ | Yes | DTR\_CheckLogin |
| Schema | lowercase | No | Yes | myschema |
| File Table | PascalCase | FT\_ | Yes | FT\_MyTable |
| Global Temporary Table | PascalCase | No | Yes | ##MyTable |
| Local Temporary Table | PascalCase | No | Yes | #MyTable |
| Table | PascalCase | No | Yes | MyTable |
| Table Column | PascalCase | No | Yes | MyColumn |
| Table Check Column Constraint | PascalCase | CHK\_ | Yes | CHK\_MyTable\_MyColumn |
| Table Check Table Constraint | PascalCase | CTK\_ | Yes | CTK\_MyTable\_MyColumn\_AnotherColumn |
| Table Primary Key | PascalCase | PK\_ | Yes | PK\_MyTableID |
| Table Alternative Key | PascalCase | AK\_ | Yes | AK\_MyTable\_MyColumn\_AnotherColumn |
| Table Foreign Key | PascalCase | FK\_ | Yes | FK\_MyTable\_ForeignTableID |
| Table Clustered Index | PascalCase | CI\_ | Yes | CI\_MyTable\_MyColumn\_AnotherColumn |
| Table Non Clustered Index | PascalCase | NCI\_ | Yes | NCI\_MyTable\_MyColumn\_AnotherColumn |
| Table Trigger | PascalCase | TR\_ | Yes | TR\_MyTable\_LogicalName |
| View | PascalCase | V\_ | No | V\_LogicalName |
| Stored Procedure | PascalCase | USP\_ | No | USP\_LogicalName |
| Scalar User-Defined Function | PascalCase | UDF\_ | No | UDF\_FunctionLogicalName |
| Table-Valued Function | PascalCase | TVF\_ | No | TVF\_FunctionLogicalName |
| Synonym | camelCase | Syn\_ | No | Syn\_logicalName |
| Sequence | PascalCase | Seq\_ | No | Seq\_TableName |
| CLR Assembly | PascalCase | CA | Yes | CALogicalName |
| CLR Stored Procedures | PascalCase | CLR\_SP\_ | Yes | CLR\_SP\_CAName\_LogicalName |
| CLR Scalar User-Defined Function | PascalCase | CLR\_UDF\_ | No | CLR\_UDF\_CAName\_LogicalName |
| CLR Table-Valued Function | PascalCase | CLR\_TVF\_ | No | CLR\_TVF\_CAName\_LogicalName |
| CLR User-Defined Aggregates | PascalCase | CLR\_UDA\_ | No | CLR\_UDA\_CAName\_LogicalName |
| CLR User-Defined Types | PascalCase | CLR\_UDT\_ | No | CLR\_UDT\_CAName\_LogicalName |
| CLR Triggers | PascalCase | CLR\_TR\_ | No | CLR\_TR\_CAName\_LogicalName |

1. **Performance Tuning Tips:**
   1. *Indexing is an effective way to tune any SQL database which is often neglected during development.*



What is an Index?

*Index is 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 B-Tree structure that enables SQL Server, to find the row or rows associated with the key values quickly and efficiently.

Please refer to the adjacent flowchart that can be used as a guideline to create indexes.

Please note that, if you frequently use INSERT, UPDATE, and DELETE operations on the tables then you should be careful while indexing, because you may end up decreasing performance, as all indexes need to be modified after these operations.

* 1. *Please avoid using functions in the WHERE clause, wherever possible.* This is because, SQL Server has to go for a table scan or index scan to get the correct results using functions; instead of doing an index seek, if there is an index that can be used. The reason for this is that the function value has to be evaluated, for each row of data to determine it matches the given criteria.

Please see below example:

* Index Scan: Uses the **LEFT** function to get the first two characters of the email address, and then each row is evaluated to see if it starts with "os"

SELECT EmailAddress

FROM [Person].[EmailAddress]

WHERE LEFT (EmailAddress, 2) = 'os'

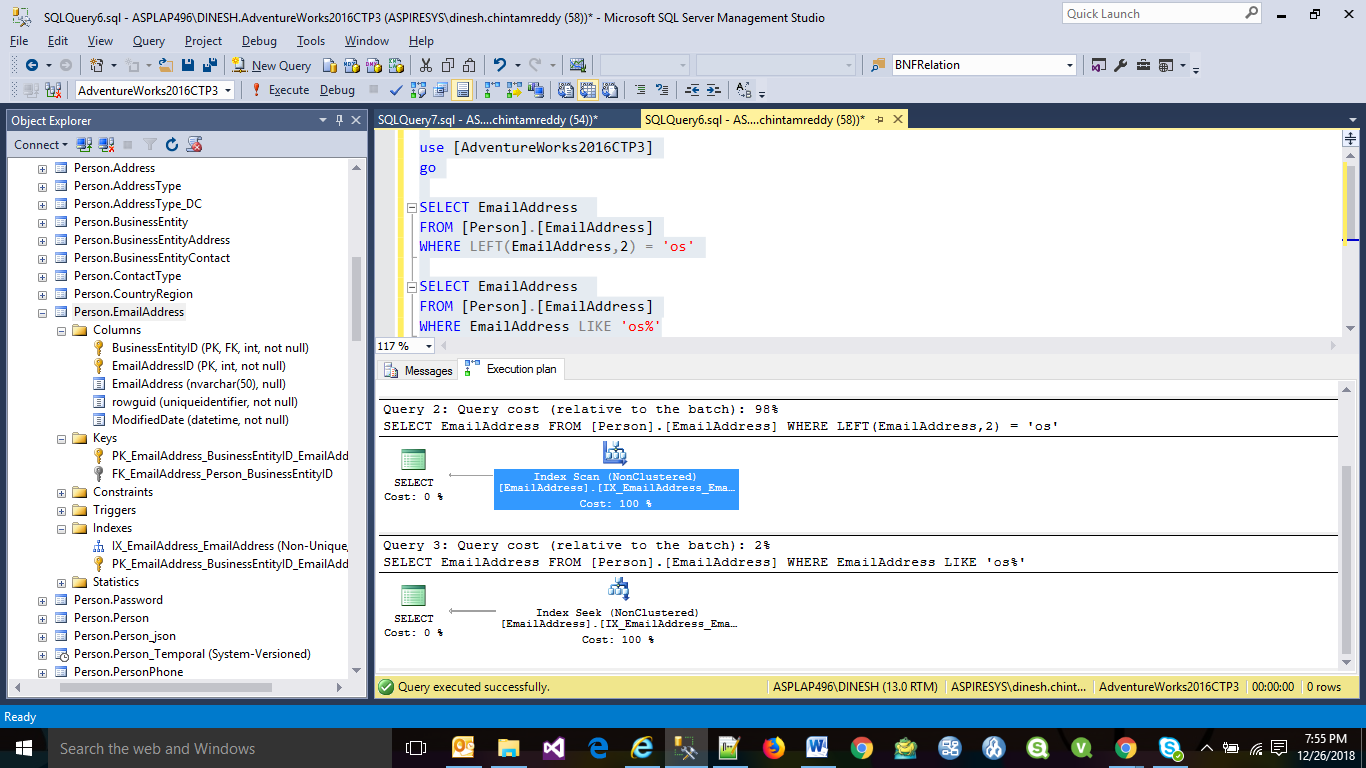
Vs

* Index Seek: Uses the **LIKE** clause to get all data that begins with "os". Since there is an index on EmailAddress column, SQL Server can do an index seek which is much more efficient then an index scan.

SELECT EmailAddress

FROM [Person].[EmailAddress]

WHERE EmailAddress LIKE 'os%'



* 1. *Please avoid writing Correlated SQL Subqueries:*

What is a Correlated Subquery?

Correlated Subquery (also known as a Synchronized Subquery) is a subquery (a query nested inside another query) that uses values from the outer query. Since the subquery may be evaluated once for each row processed by the outer query, it may not be efficient. Correlated subqueries may appear elsewhere besides the WHERE clause as well; for example, the following query uses a correlated subquery in the SELECT clause:

SELECT

C.NAME,

C.CITY,

(

SELECT COMPANYNAME

FROM COMPANY

WHERE COMPANYID = C.COMPANYID

) AS COMPANYNAME

FROM

CUSTOMER C

The main problem in the above SQL is that the inner query (SELECT COMPANYNAME …) runs for each row returned by the outer query (SELECT C.NAME …). So, it would be better to refactor the correlated subquery using a join:

SELECT

Cust.NAME,

Cust.CITY,

Comp.COMPANYNAME

FROM CUSTOMER Cust

LEFT JOIN COMPANY Comp

ON Cust.COMPANYID = Comp.COMPANYID

* 1. If you want to check whether a record exists, use EXISTS () instead of COUNT (), because COUNT () scans the entire table, counting up all entries matching your condition, whereas EXISTS () will exit as soon as it sees the result it needs.

IF (SELECT COUNT(1) FROM [HumanResources].[Employee] WHERE [JobTitle] LIKE 'Chief%Officer') > 0

PRINT 'Record Exists'

Vs

IF EXISTS (SELECT [LoginID] FROM [HumanResources].[Employee] WHERE [JobTitle] LIKE 'Chief%Officer')

PRINT 'Record Exists'

Likewise, please use “NOT EXISTS” instead of using “NOT IN”. For example, in sub query statements such as the following, the NOT IN clause causes an internal sort/ merge:

SELECT \* FROM Student

WHERE StudentId NOT IN (SELECT StudentId FROM Class)

Instead, please use:

SELECT \* FROM Student Stu

WHERE NOT EXISTS

(SELECT 1 FROM Class Cls

WHERE Cls.StudentId = Stu.StudentId)

* 1. Please make sure to add **SET NOCOUNT ON** at the top of each stored procedure and **SET NOCOUNT OFF** at the bottom.
  2. Please avoid temp tables as much as you can, but if you need a temp table, create it explicitly by *Create Table #temp*.

1. **Performance Tuning Tools:**

Please refer to the following tools that can be used to collect performance related data:

1. Dynamic Management Views (DMVs) and System Catalog Views
2. Profiler and Server Side Traces
3. Windows Performance Monitor
4. Built in performance reports in SSMS
5. Query Plans
6. Database Tuning Advisor
   1. **Dynamic Management Views:**

Microsoft introduced Dynamic Management Views (DMVs) in SQL Server 2005, which allow us to get better insight of the SQL Server, in order to troubleshoot performance related issues.

DMVs actually come in two flavors DMVs (dynamic management views) and DMFs (dynamic management functions) and are sometimes classified as DMOs (dynamic management objects). The DMVs act just like any other view where you can select data from them and the DMFs require values to be passed to the function just like any other function.

Please see below DMVs that we should be familiarized:

sys.dm\_exec\_cached\_plans - Cached query plans available to SQL Server

sys.dm\_exec\_sessions - Sessions in SQL Server

sys.dm\_exec\_connections - Connections to SQL Server

sys.dm\_db\_index\_usage\_stats - Seeks, scans, lookups per index

sys.dm\_io\_virtual\_file\_stats - IO statistics for databases and log files

sys.dm\_tran\_active\_transactions - Transaction state for an instance of SQL Server

sys.dm\_exec\_sql\_text - Returns TSQL code

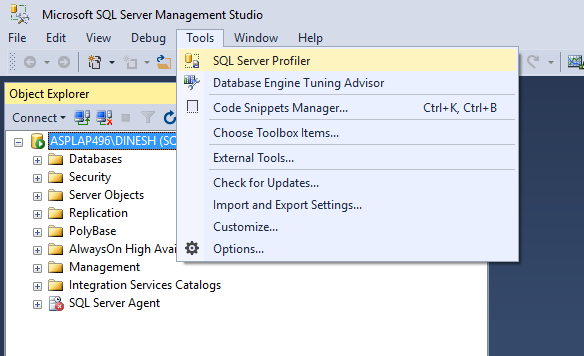
sys.dm\_exec\_query\_plan - Returns query plan

sys.dm\_os\_wait\_stats - Returns information what resources SQL is waiting on

sys.dm\_os\_performance\_counters - Returns performance monitor counters related to SQL Server

* 1. **Profiler and Server Side Traces:**

SQL Server Profiler is a GUI based tool, that allows us to see what statements are running on our SQL Server apart from collecting metrics such as duration, number of reads, number of writes, the machine that ran the query, etc...



* 1. **Windows Performance Monitor** allows us to capture and graph many aspects for the Windows server. There are counters for .NET, Disks, Memory, Processors, Network, etc... as well several counters related to each instance of SQL Server on the box. If we have multiple instances running on one server, the counters are available for each instance so we can see what is occurring at the instance level.

Launching Performance Monitor:

Start > Control Panel > Administrative Tools > Performance Monitor

Or, through the Run command: **PerfMon.exe**

* 1. **Built in Performance Reports in SSMS:**

We can also get performance related information from SQL Server using the built-in performance reports. The reports were first introduced with SQL Server 2005 as an add-on, but are now standard with later versions. The reports provide useful information that can assist us in determining where the performance bottlenecks may be. Please see below list:

Server Dashboard

Scheduler Health

Memory Consumption

Activity - All Blocking Transactions

Activity - Top Sessions

Activity - top Connections

Top Connections by Block Transactions Count

Top Transaction by Locks Count

Performance - Batch Execution Statistics

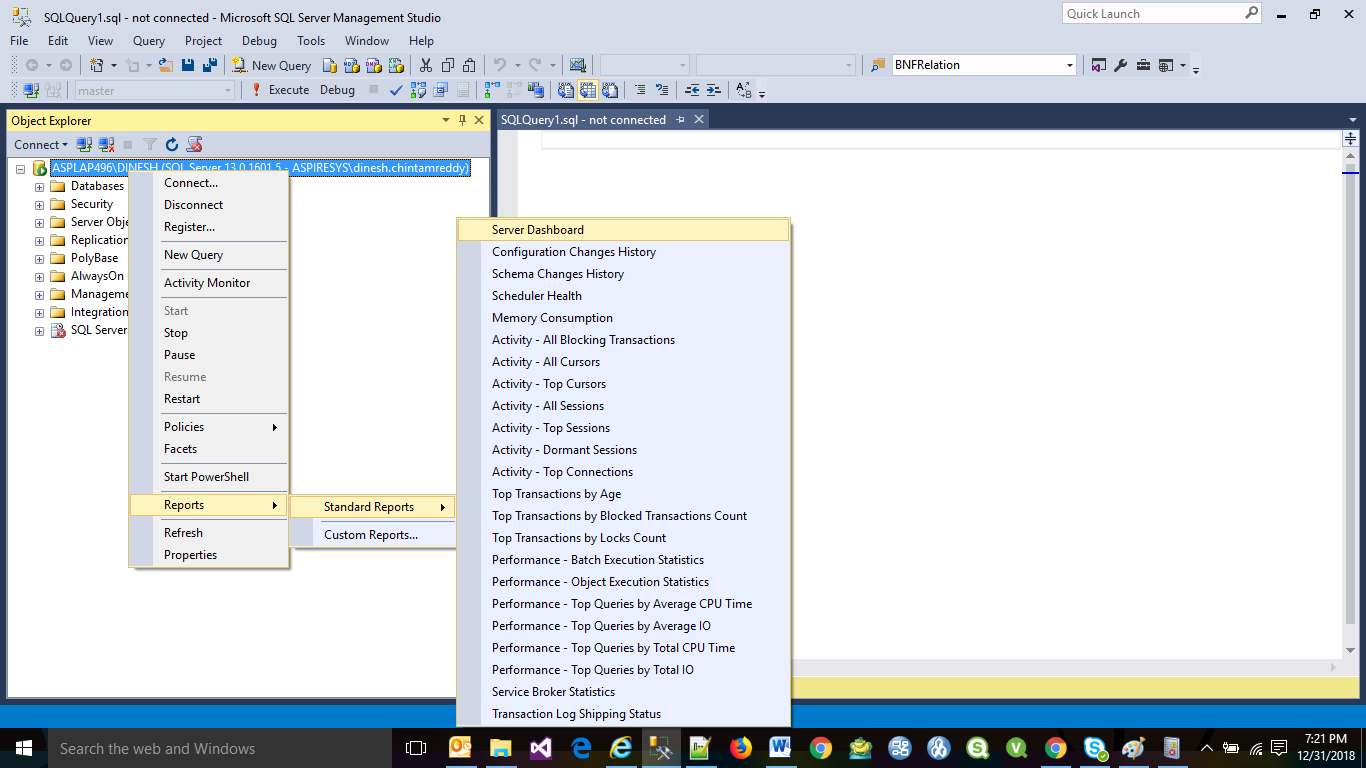
Performance - Object Execution Statistics

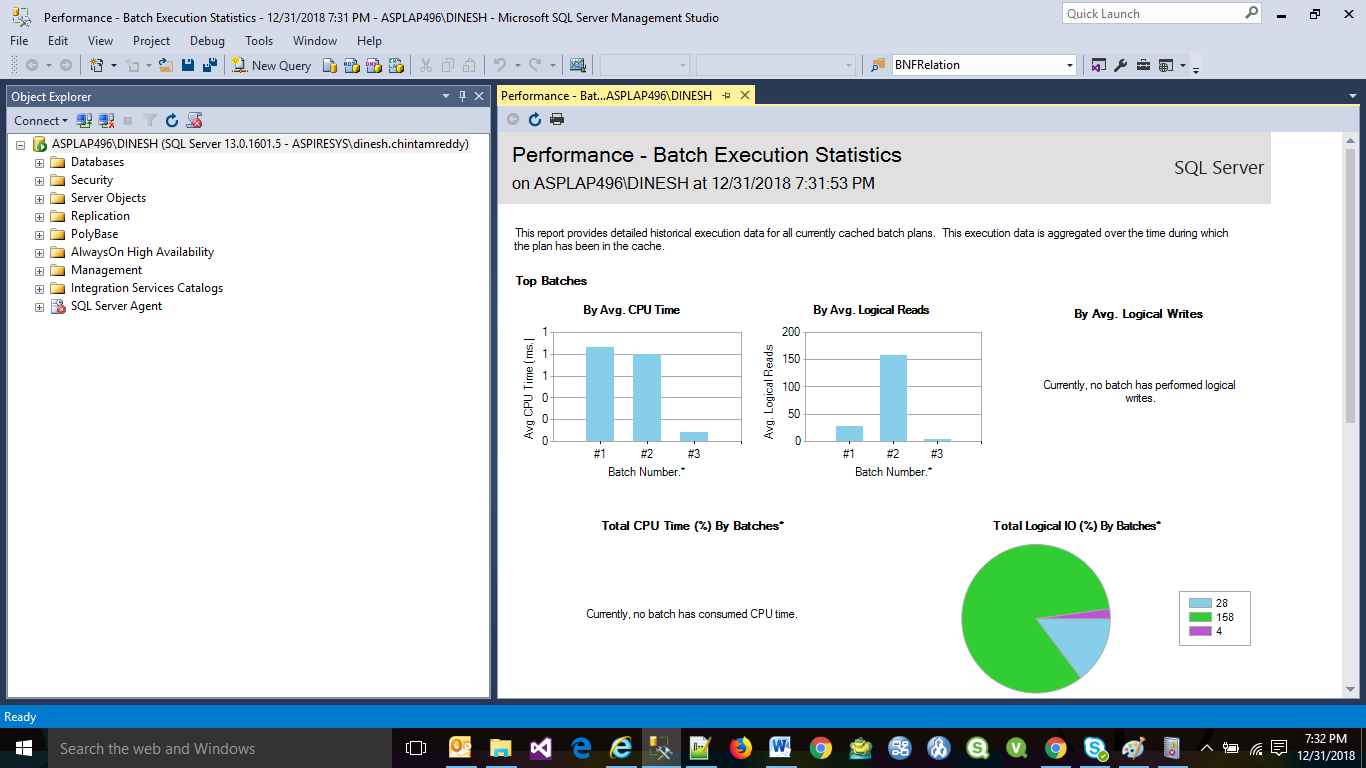
Performance - Top Queries by Average CPU Time

Performance - Top Queries by Average IO

Performance - Top Queries by Total CPU Time

Performance - Top Queries by Total IO





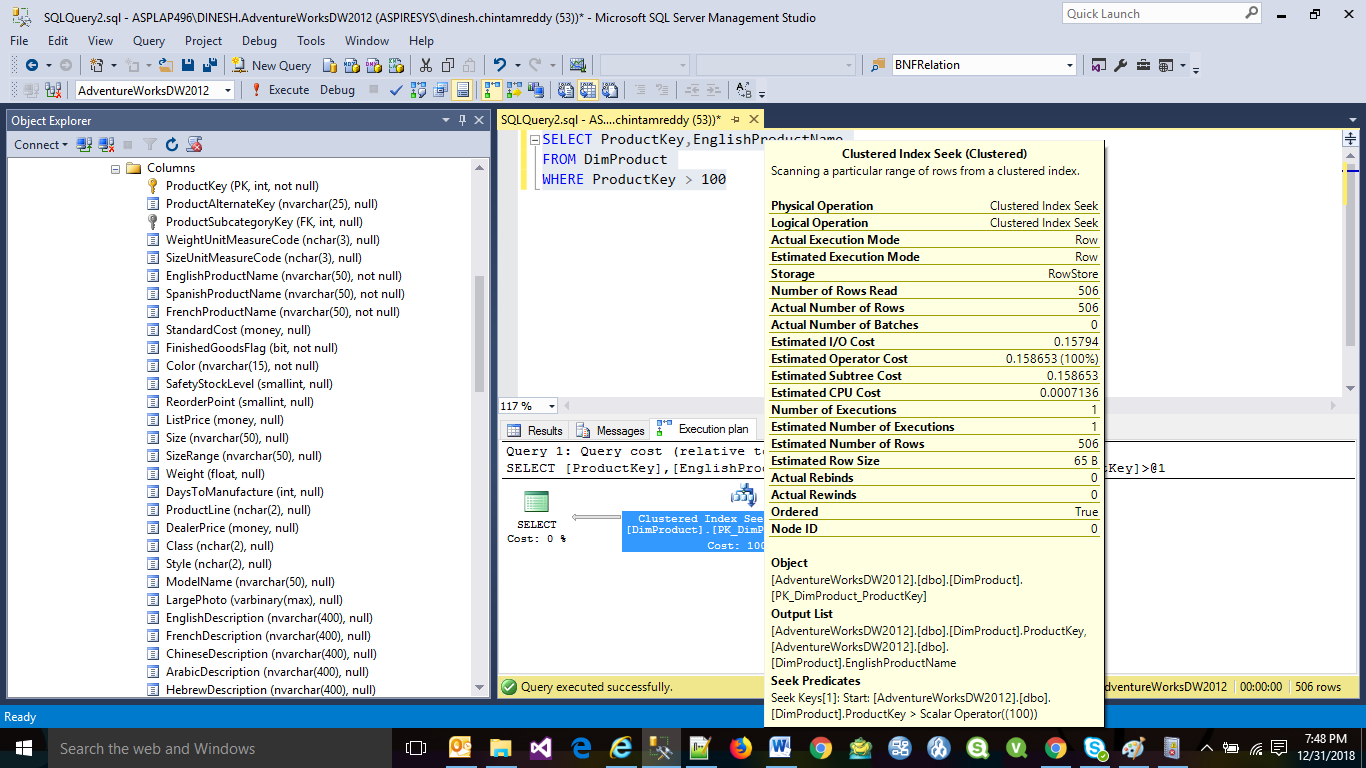
* 1. **Query Execution Plans**

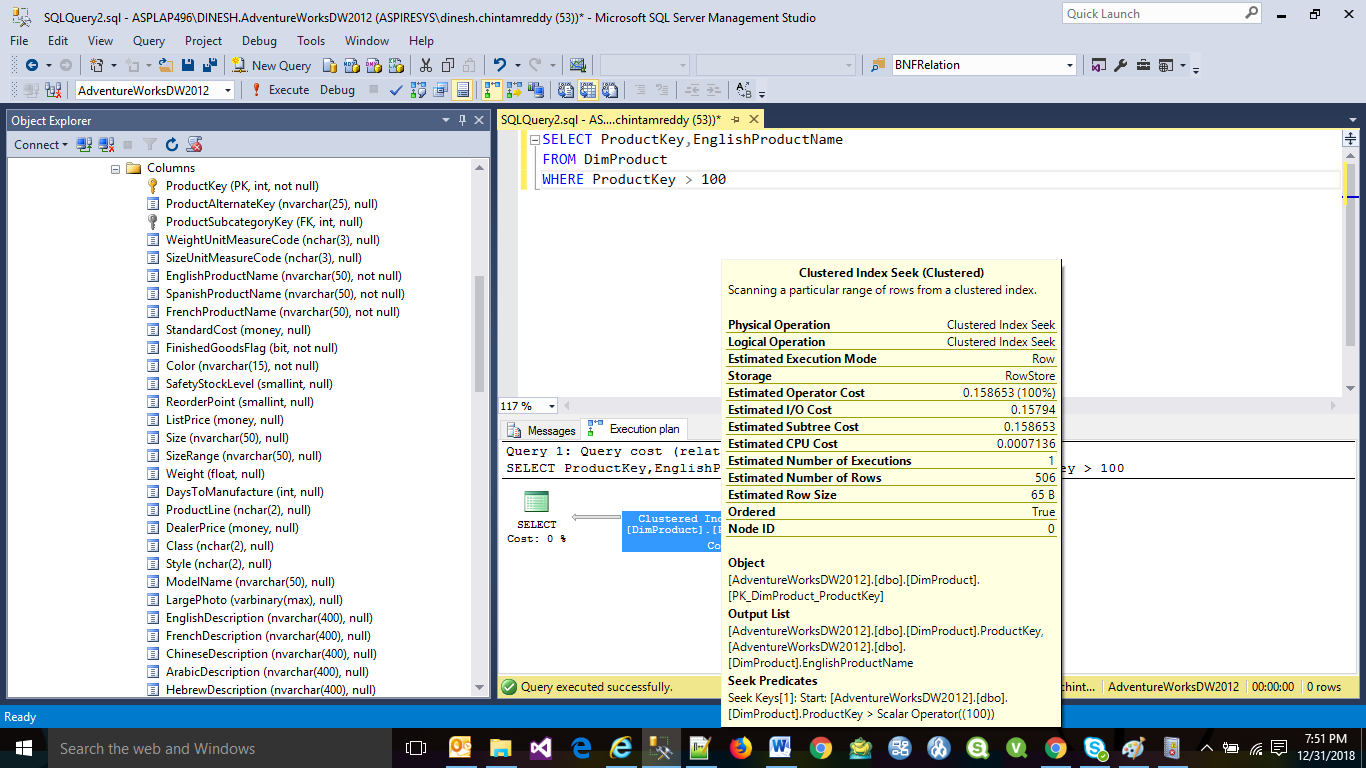
The Query Execution Plans describe the steps and the order used to access or to modify data in the database. SQL Server can create execution plans in two ways:

* Actual Execution Plan is created after execution of the query and contains the steps that were performed
* Estimated Execution Plan is created without executing the query and contains an approximate execution plan

Execution plans can be presented in the following three ways and each option offers benefits over the other:

* Text Plans
* Graphical Plans
* XML Plans

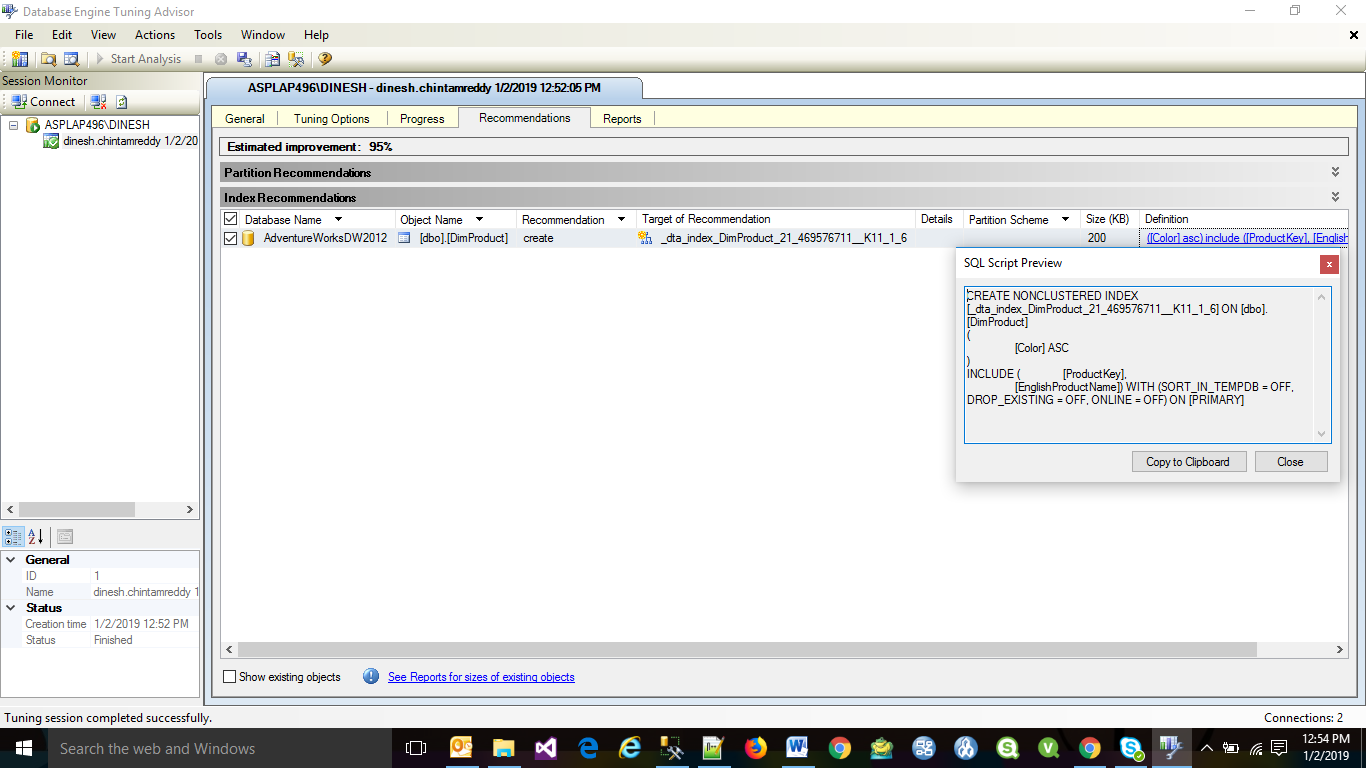




* 1. **Database Tuning Advisor:**

The Database Tuning Advisor helps us to figure out if additional indexes are helpful, whether partitioning improves the performance, etc. Here is a summary of the options:

* Adding indexes (clustered, non-clustered, and indexed views)
* Adding partitioning
* Adding statistics including multi-column statistics which are not created automatically even when we enable the AUTO\_CREATE\_STATISTICS database option.



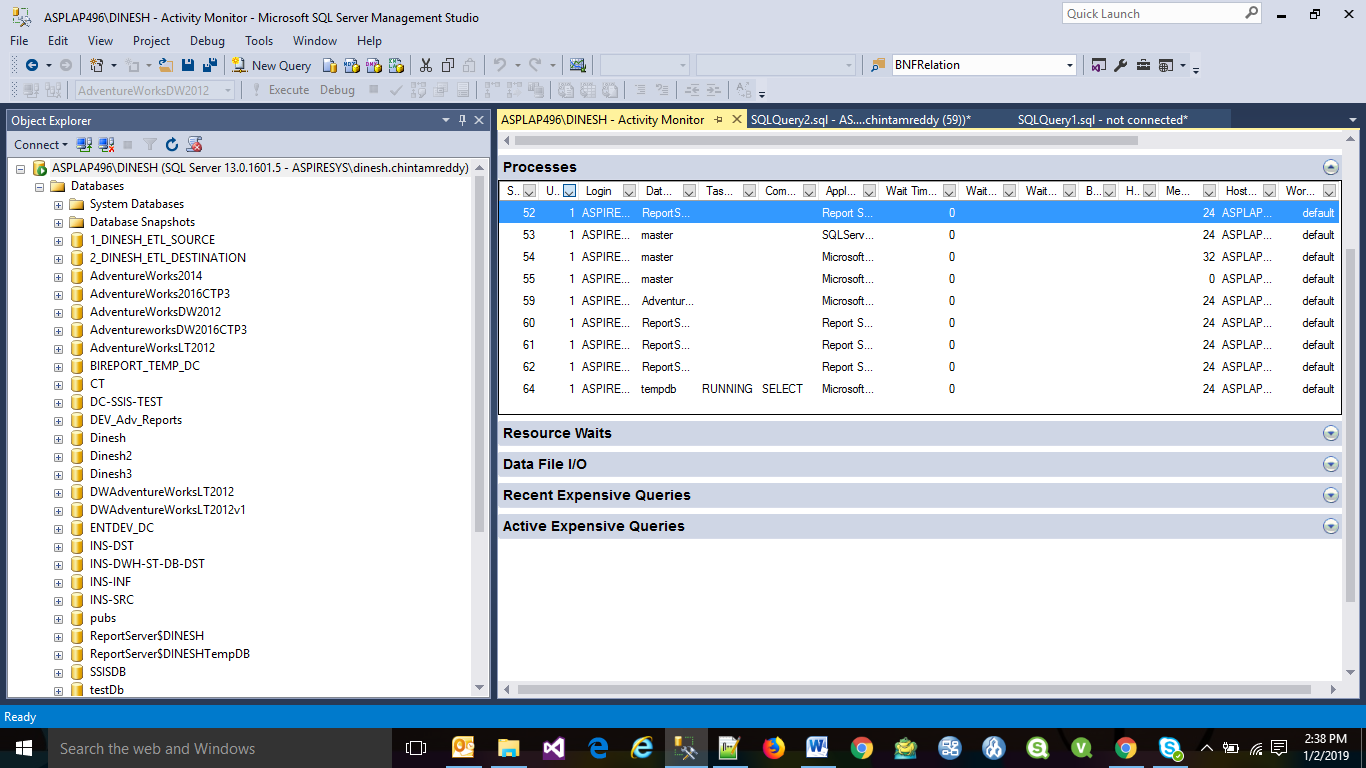
1. **Performance Issues:**

There are several factors that can degrade SQL Server performance and let us go through some of the common areas that can affect performance.

* 1. **Blocking:**

In order for SQL Server to maintain data integrity for both reads and writes it uses locks, so that only one process has control of the data at any one time. There are several types of locks that can be used such as Shared, Update, Exclusive, Intent, etc., and each of these has a different behavior and effect on our data.

When locks are held for a long period of time they cause blocking, which means one process has to wait for the other process to finish with the data and release the lock before the second process can continue.



* 1. **Deadlocks:**

A deadlock occurs when two or more processes are waiting on the same resource and each process is waiting on the other process to complete before moving forward. When this situation occurs and there is no way for these processes to resolve the conflict, SQL Server will choose one of processes as the deadlock victim and rollback that process, so the other process or processes can move forward.

Deadlock information can be captured in the SQL Server Error Log or by using Profiler / Server Side Trace.

Trace Flags: If you want to capture this information in the SQL Server Error Log, you need to enable one or both of the following trace flags.

1204 - This provides information about the nodes involved in the deadlock

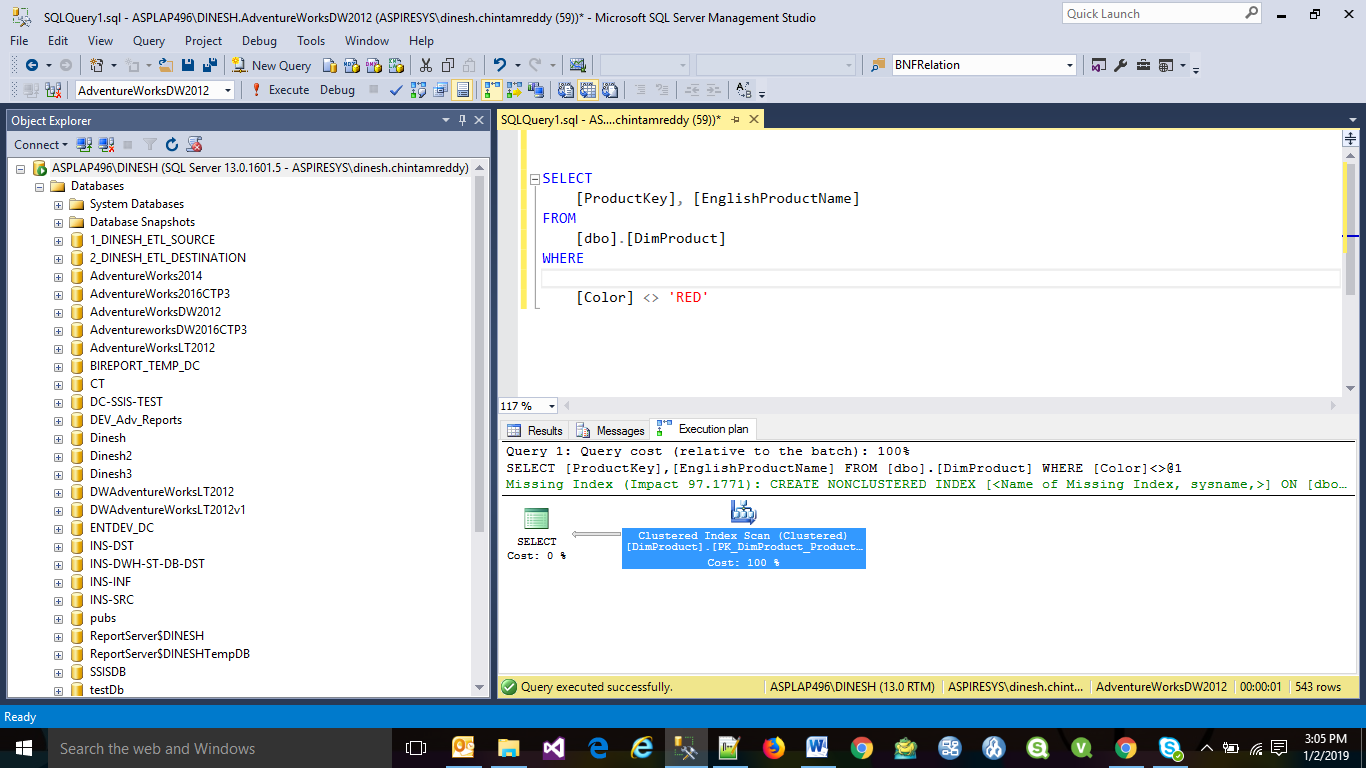
1222 - Returns deadlock information in an XML format

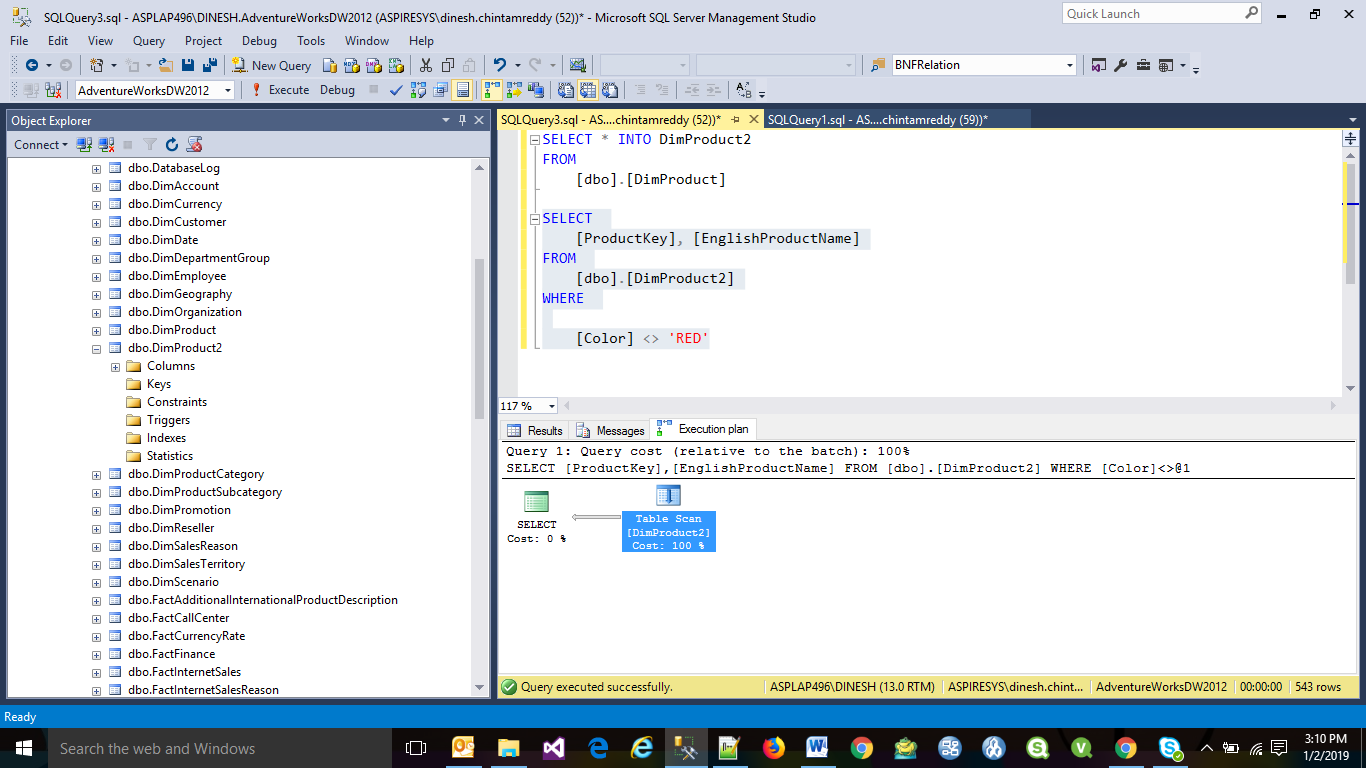
* 1. **Index Scans and Table Scans**

An index scan or table scan happens when SQL Server has to scan the data or index pages to find the appropriate records. A scan is the opposite of a seek, where a seek uses the index to pinpoint the records that are needed to satisfy the query.

The reason you would want to find and fix your scans is because they generally require more I/O and also take longer to process. This is something you will notice with an application that grows over time. When it is first released performance is great, but over time as more data is added the index scans take longer and longer to complete.

Please see below two examples, where Index Scan and Table Scan being used: when a table has a Clustered Index it will do a Clustered Index Scan and when the table does not have a clustered index it will do a Table Scan.





* 1. **Discovering Unused Indexes**

To ensure that data access can be as fast as possible, SQL Server like other relational database systems utilizes indexing to find data quickly. SQL Server has different types of indexes that can be created such as clustered indexes, non-clustered indexes, XML indexes and Full Text indexes.

The benefit of having more indexes is that SQL Server can access the data quickly if an appropriate index exists. The downside to having too many indexes is that SQL Server has to maintain all of these indexes which can slow things down and indexes also require additional storage. So as you can see indexing can both help and hurt performance.

Please see below DMV query that gives the info related to the indexes:

SELECT OBJECT\_NAME (S.[OBJECT\_ID]) AS [OBJECT NAME],

I.[NAME] AS [INDEX NAME],

USER\_SEEKS,

USER\_SCANS,

USER\_LOOKUPS,

USER\_UPDATES

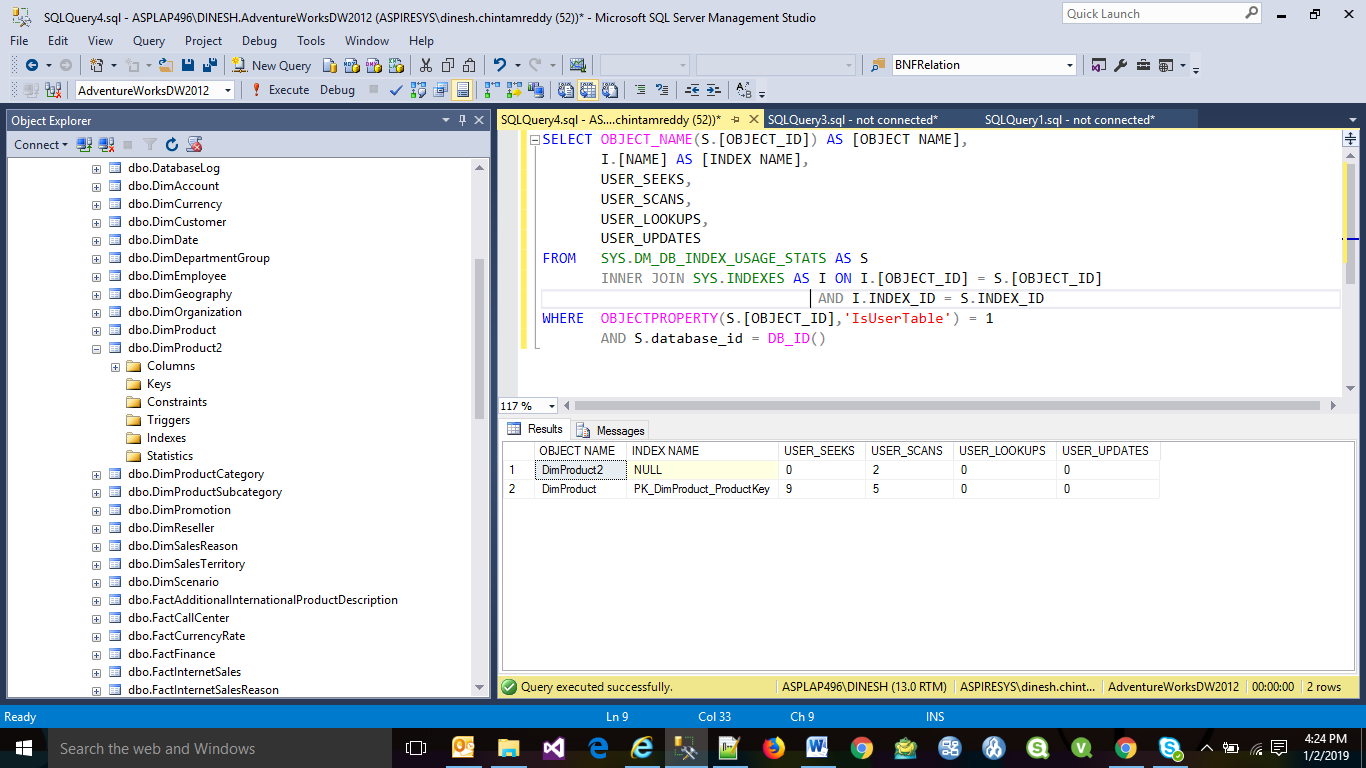
FROM SYS.DM\_DB\_INDEX\_USAGE\_STATS AS S

INNER JOIN SYS.INDEXES AS I ON I.[OBJECT\_ID] = S.[OBJECT\_ID] AND I.INDEX\_ID = S.INDEX\_ID

WHERE OBJECTPROPERTY(S.[OBJECT\_ID],'IsUserTable') = 1

AND S.database\_id = DB\_ID()

If you see indexes where there are no seeks, scans or lookups, but there are updates this means that SQL Server has not used the index to satisfy a query but still needs to maintain the index.



* 1. **I/O bottlenecks**

SQL Server is usually a high I/O activity process and in most cases the database is larger than the amount of memory installed on a computer and therefore SQL Server has to pull data from disk to satisfy queries. In addition, since the data in databases is constantly changing these changes need to be written to disk.

Another process that can consume a lot of I/O is the TempDB database. The TempDB database is a temporary working area for SQL Server to do such things as sorting and grouping. The TempDB database also resides on disk and therefore depending on how many temporary objects are created this database could be busier than your user databases.

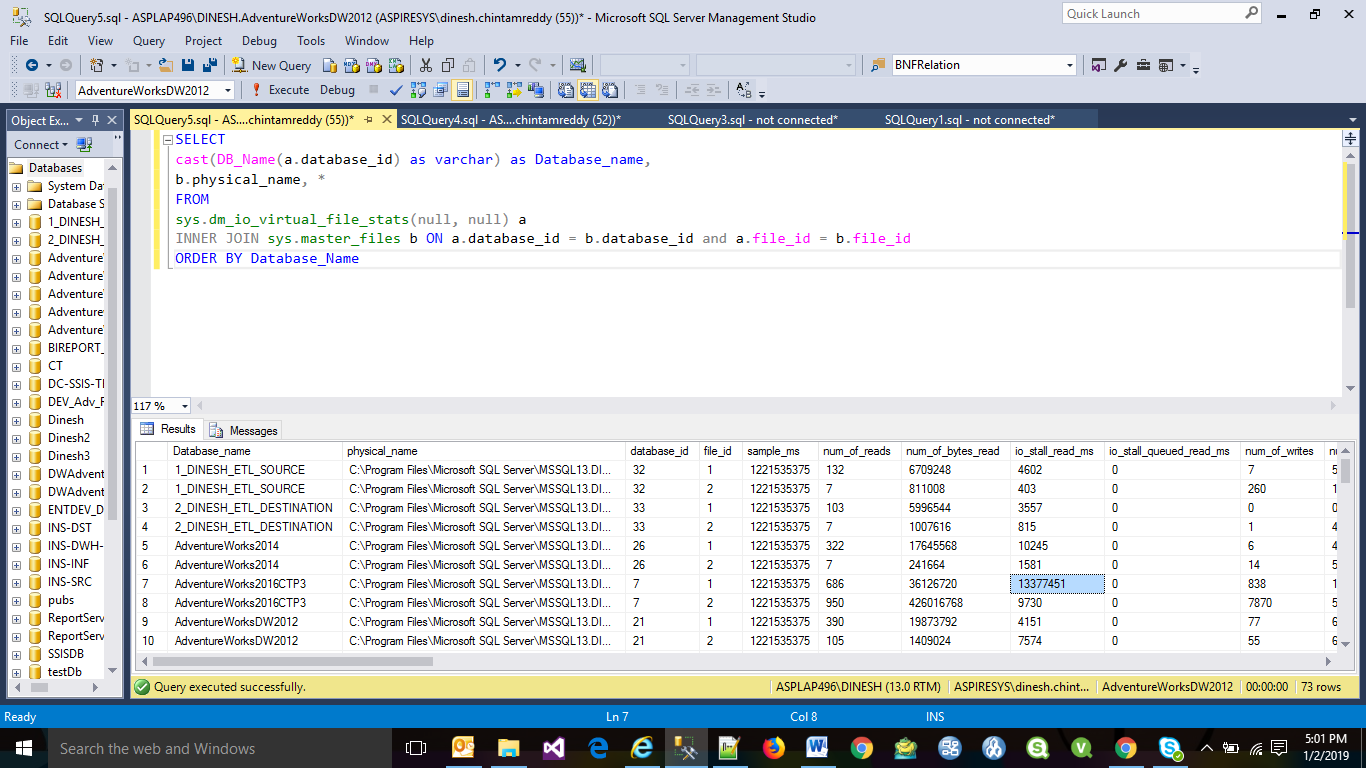
There are two options that will exist in every SQL Server environment to monitor I/O bottlenecks: SQL Server DMVs and Performance Monitor.

DMV: SYS.DM\_IO\_VIRTUAL\_FILE\_STATS

This DMV will give us cumulative file stats for each database and each database file including both the data and log files. Based on this data we can determine which file is the busiest from a read and/or write perspective.

The output also includes I/O stall information for reads, writes and total. The I/O stall is the total time, in milliseconds, that users waited for I/O to be completed on the file. By looking at the I/O stall information we can see how much time was waiting for I/O to complete and therefore the users were waiting.

The data that is returned from this DMV is cumulative data, which means that each time we restart SQL Server the counters are reset. Since the data is cumulative we can run this once and then run the query again in the future and compare the deltas for the two time periods. If the I/O stalls are high compared to the length of the time period then we may have an I/O bottleneck.



**Performance Monitor** is a Windows tool that lets us to capture statistics about SQL Server, memory usage, I/O usage, etc... This tool can be run interactively using the GUI or we can set it up to collect information behind the scenes which can be reviewed at a later time. This tool is found in the Control Panel under Administrative tools.

There are several counters related to I/O and they are located under Physical Disk and Logical Disk. The Physical Disk performance object consists of counters that monitor hard or fixed disk drive on a computer. The Logical Disk performance object consists of counters that monitor logical partitions of a hard or fixed disk drives

1. **Summary on Indexing:**

An index is an on-disk structure associated with a table or view, which 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.

A table or view can contain the following types of indexes:

**Clustered Indexes:** 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 stored in only one order. The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap. If a table is a heap and does not have any Non Clustered Indexes, then the entire table must be examined (a table scan) to find any row.

When a table is stored as a heap, individual rows are identified by reference to a row identifier (RID) consisting of the file number, data page number, and slot on the page. The row id is a small and efficient structure. Sometimes data architects use heaps when data is always accessed through non clustered indexes and the RID is smaller than a clustered index key.

* Do not use a heap when the data is frequently returned in a sorted order. A clustered index on the sorting column could avoid the sorting operation.
* Do not use a heap when the data is frequently grouped together. Data must be sorted before it is grouped, and a clustered index on the sorting column could avoid the sorting operation.
* Do not use a heap when ranges of data are frequently queried from the table. A clustered index on the range column will avoid sorting the entire heap.
* Do not use a heap when there are no non clustered indexes and the table is large. In a heap, all rows of the heap must be read to find any row.

**Non Clustered Indexes:** Non Clustered Indexes have a structure separate from the data rows. A Non Clustered Index contains the Non Clustered 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 Non Clustered 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.

Both Clustered and Non Clustered Indexes can be unique. This means no two rows can have the same value for the index key. Otherwise, the index is not unique and multiple rows can share the same key value. Indexes are automatically maintained for a table or view whenever the table data is modified.

Indexes are automatically created when PRIMARY KEY and UNIQUE constraints are defined on table columns. Well-designed indexes can reduce disk I/O operations and consume fewer system resources therefore improving query performance.

When performing a table scan, the query optimizer reads all the rows in the table, and extracts the rows that meet the criteria of the query. A table scan generates many disk I/O operations and can be resource intensive. However, a table scan could be the most efficient method if, for example, the result set of the query is a high percentage of rows from the table.

When the query optimizer uses an index, it searches the index key columns, finds the storage location of the rows needed by the query and extracts the matching rows from that location. Generally, searching the index is much faster than searching the table because unlike a table, an index frequently contains very few columns per row and the rows are in sorted order.

1. **Implementation references:**
   1. **Physical Database Storage Design:** Please refer to the following article:

<http://technet.microsoft.com/en-us/library/cc966414.aspx>

* 1. **Disk Partition Alignment Best Practices for SQL Server:** Please refer to the following article:

<http://msdn.microsoft.com/en-us/library/dd758814(v=sql.100).aspx>