**SQL Server Index Basics**

## Indexes speed up the querying process by providing speedy access to rows in the data tables

## Index Structures

Indexes are created on columns in tables or views. The index provides a fast way to look up data based on the values within those columns. For example, if you create an index on the primary key and then search for a row of data based on one of the primary key values, SQL Server first finds that value in the index, and then uses the index to quickly locate the entire row of data. Without the index, a table scan would have to be performed in order to locate the row, which can have a significant effect on performance.

An index is made up of a set of pages (index nodes) that are organized in a B-tree structure. This structure is hierarchical in nature, with the root node at the top of the hierarchy and the leaf nodes at the bottom. When a query is issued against an indexed column, the query engine starts at the root node and navigates down through the intermediate nodes, with each layer of the intermediate level more granular than the one above. The query engine continues down through the index nodes until it reaches the leaf node

### Clustered Indexes

A clustered index stores the actual data rows at the leaf level of the index. An important characteristic of the clustered index is that the indexed values are sorted in either ascending or descending order. As a result, there can be only one clustered index on a table or view. In addition, data in a table is sorted only if a clustered index has been defined on a table.

Note: A table that has a clustered index is referred to as a clustered table. A table that has no clustered index is referred to as a heap.

### Nonclustered Indexes

Unlike a clustered indexed, the leaf nodes of a nonclustered index contain only the values from the indexed columns and row locators that point to the actual data rows, rather than contain the data rows themselves. This means that the query engine must take an additional step in order to locate the actual data.

A row locator’s structure depends on whether it points to a clustered table or to a heap. If referencing a clustered table, the row locator points to the clustered index, using the value from the clustered index to navigate to the correct data row. If referencing a heap, the row locator points to the actual data row.

Nonclustered indexes cannot be sorted like clustered indexes; however, you can create more than one nonclustered index per table or view. SQL Server 2005 supports up to 249 nonclustered indexes, and SQL Server 2008 support up to 999. This certainly doesn’t mean you should create that many indexes. Indexes can both help and hinder performance

In addition to being able to create multiple nonclustered indexes on a table or view, you can also add included columns to your index. This means that you can store at the leaf level not only the values from the indexed column, but also the values from non-indexed columns. This strategy allows you to get around some of the limitations on indexes. For example, you can include non-indexed columns in order to exceed the size limit of indexed columns (900 bytes in most cases).

Clustered Index

* Only one per table
* Faster to read than non clustered as data is physically stored in index order

Non Clustered Index

* Can be used many times per table
* Quicker for insert and update operations than a clustered index

Both types of index will improve performance when select data with fields that use the index but will slow down update and insert operations.

Because of the slower insert and update clustered indexes should be set on a field that is normally incremental ie Id or Timestamp.

SQL Server will normally only use an index if its selectivity is above 95%.

# [What is the difference between OFFLINE and ONLINE index rebuild in SQL Server?](https://stackoverflow.com/questions/6309614/what-is-the-difference-between-offline-and-online-index-rebuild-in-sql-server)

In ONLINE mode the new index is built while the old index is accessible to reads and writes. any update on the old index will also get applied to the new index. An antimatter column is used to track possible conflicts between the updates and the rebuild (ie. delete of a row which was not yet copied). See [Online Index Operations](http://download.microsoft.com/download/8/5/e/85eea4fa-b3bb-4426-97d0-7f7151b2011c/OnlineIndex.doc). When the process is completed the table is locked for a brief period and the new index replaces the old index. If the index contains LOB columns, ONLINE operations are not supported in SQL Server 2005/2008/R2.

In OFFLINE mode the table is locked upfront for any read or write, and then the new index gets built from the old index, while holding a lock on the table. No read or write operation is permitted on the table while the index is being rebuilt. Only when the operation is done, the lock on the table released and reads and writes are allowed again.

1) OFFLINE index rebuild is faster than ONLINE rebuild.

2) Extra disk space required during SQL Server online index rebuilds.

3) SQL Server locks acquired with SQL Server online index rebuilds.

* This schema modification lock blocks all other concurrent access to the table, but it is only held for a very short period of time while the old index is dropped and the statistics updated.

## [REBUILD vs. REORGANIZE](https://www.sqlskills.com/blogs/paul/sqlskills-sql101-rebuild-vs-reorganize/)

**Rebuild**: An index ‘rebuild’ creates a fresh, sparkling new structure for the index. [If the index is disabled](https://www.brentozar.com/archive/2013/02/disabling-vs-dropping-indexes/), rebuilding brings it back to life. You can [apply a new fillfactor when you rebuild an index](https://www.brentozar.com/archive/2013/04/five-things-about-fillfactor/). If you cancel a rebuild operation midway, it must roll back (and if it’s being done offline, that can take a while).

**Reorganize**: This option is more lightweight. It runs through the leaf level of the index, and as it goes it fixes physical ordering of pages and also compacts pages to apply any previously set fillfactor settings. This operation is always online, and if you cancel it then it’s able to just stop where it is (it doesn’t have a giant operation to rollback).

SELECT \* FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, NULL)

ALTER INDEX indexName ON tableName REBUILD;

ALTER INDEX indexName ON tableName REORGANIZE;

**Space Required**

Rebuilding an index requires building a new index before dropping the old index, regardless of the amount of fragmentation present in the old index. This means you need to have enough free space to accommodate the new index.

Reorganizing an index first squishes the index rows together to try to deallocate some index pages, and then shuffles the remaining pages in place to make their physical (allocation) order the same as the logical (key) order. This only requires a single 8-KB page, as a temporary storage for pages being moved around. So an index reorganize is extremely space efficient, and is one of the reasons I wrote the original *DBCC INDEXDEFRAG* for SQL Server 2000 (the predecessor of *ALTER INDEX … REORGANIZE*).

If you have space constraints, and can’t make use of single-partition rebuild, reorganizing is the way to go.

**Algorithm Speed**

An index rebuild will always build a new index, even if there’s no fragmentation. The length of time the rebuild takes is related to the size of the index, not the amount of fragmentation in it.

Reorganizing an index only deals with the fragmentation that exists, and the more fragmentation there is, the longer a reorganize will take.

This means that for a lightly fragmented index (e.g. less than 30% fragmentation), it’s generally faster to reorganize the index, but for a more heavily fragmented index, it’s generally faster to just rebuild the index. This is why you may have seen thresholds of 0 to 5-10% do nothing, 5-10% to 30% reorganize, 30%+ rebuild.

**Transaction Log Generated**

In the FULL recovery mode, an index rebuild is fully logged, so the transaction log will have to accommodate the full size of the index in a single transaction..

In the SIMPLE and BULK\_LOGGED recovery modes, the amount of transaction log generated by an offline index rebuild will be minimal (online index rebuild is always fully logged) – just the allocations of pages and extents. However, the next log backup performed (either in BULK\_LOGGED or after switching to FULL) will also contain all the extents changed by the rebuild, and so the log backup will be roughly the same size as if the rebuild was done in the FULL recovery mode. The benefits are in time and the fact that the transaction log itself does not have to accommodate the full size of the index during the rebuild in a single transaction.

In all recovery modes, reorganizing an index is fully logged, but is performed as a series of small transactions so should not cause the transaction log to grow inordinately. And of course, transaction log is only generated for the operations performed, which may be less for a reorganize as it only deals with fragmentation that exists.

**Locks Required**

An offline index rebuild of any index holds a schema-modification (i.e. super-exclusive) table lock – no updates or reads of the entire table.

An online index rebuild of any index acquires a short-term shared table lock at the start of the operation, holds an intent-shared table lock throughout the operation (which will only block exclusive and schema-modification table locks), and then acquires a short-term schema-modification  table lock at the end of the operation.

An index reorganize holds an intent-exclusive table lock throughout the operation, which will only block shared, exclusive, and schema-modification table locks. One of the major reasons I wrote DBCC INDEXDEFRAG for SQL Server 2000 was as an online alternative to DBCC DBREINDEX.

**Interruptible or Not**

An index rebuild operation cannot be interrupted without it rolling back everything it’s done so far – it’s atomic – all or nothing. In SQL Server 2017, however, there is a resumable-online index rebuild feature.

An index reorganize can be interrupted and the worst that will happen is that a single page move operation is rolled back.

**Statistics**

An index rebuild will always rebuild the index column statistics with the equivalent of a full scan (or sampled, for an index partition or if the index is partitioned).

An index reorganize does not see a total view of the index and so cannot update statistics, meaning that manual index statistics maintenance is required.

**Summary**

As you can see, there are quite a few major differences between rebuilding and reorganizing, but there’s no right answer as to which one you should use – that’s your choice.

If you have an index maintenance routine that always rebuilds and never considers reorganizing, you should reconsider. It’s usually better to reorganize a lightly fragmented index and rebuild a more heavily fragmented index – to save time and resources.

**What  is the Meaning of Bookmark lookup , Key lookup , RID lookup in SQL Server ?**

* bookmark lookup or RID lookup showing in the Execution plan when you select Columns not included on your index
* if table has clustered index, it is called bookmark lookup (or key lookup); if the table does not have clustered index, but a non-clustered index, it is called RID lookup.
* You Must remove the lookup from the Execution plan to improve your Performance .

When a small number of rows are requested by a query, the SQL Server optimizer will try to use a non-clustered index on the column or columns contained in the WHERE clause to retrieve the data requested by the query. If the query requests data from columns not present in the non-clustered index, SQL Server must go back to the data pages to get the data in those columns. Even if the table has a clustered index or not, the query will still have to return to the table or clustered index to retrieve the data.

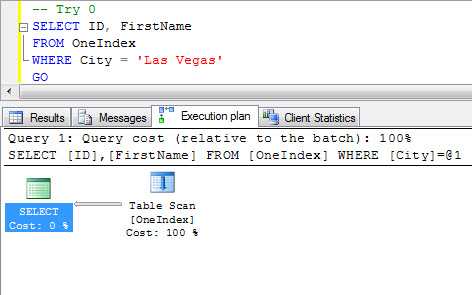
In the above scenario, if table has clustered index, it is called **bookmark lookup** (or key lookup); if the table does not have clustered index, but a non-clustered index, it is called **RID lookup**. This operation is very expensive. To optimize any query containing bookmark lookup or RID lookup, it should be removed from the execution plan to improve performance. There are two different ways to remove bookmark/RID lookup.

Before we understand these two methods, we will create sample table without clustered index and simulate RID lookup. RID Lookup is a bookmark lookup on a heap that uses a supplied row identifier (RID).

USE tempdb  
GO  
-- Create Table OneIndex with few columns  
CREATE TABLE OneIndex (ID INT,  
FirstName VARCHAR(100),  
LastName VARCHAR(100),  
City VARCHAR(100))  
GO  
-- Insert One Hundred Thousand Records  
INSERT INTO OneIndex (ID,FirstName,LastName,City)  
SELECT TOP 100000 ROW\_NUMBER() OVER (ORDER BY a.name) RowID,  
'Bob',  
CASE WHEN ROW\_NUMBER() OVER (ORDER BY a.name)%2 = 1 THEN 'Smith'  
ELSE 'Brown' END,  
CASE  
WHEN ROW\_NUMBER() OVER (ORDER BY a.name)%1000 = 1 THEN 'Las Vegas'  
WHEN ROW\_NUMBER() OVER (ORDER BY a.name)%10 = 1 THEN 'New York'  
WHEN ROW\_NUMBER() OVER (ORDER BY a.name)%10 = 5 THEN 'San Marino'  
WHEN ROW\_NUMBER() OVER (ORDER BY a.name)%10 = 3 THEN 'Los Angeles'  
ELSE 'Houston' END  
FROM sys.all\_objects a  
CROSS JOIN sys.all\_objects b  
GO

Now let us run following select statement and check the execution plan.

SELECT ID, FirstName  
FROM OneIndex  
WHERE City = 'Las Vegas'  
GO

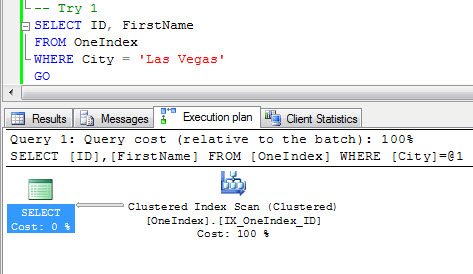


As there is no index on table, scan is performed over the table. We will create a clustered index on the table and check the execution plan once again.

-- Create Clustered Index  
CREATE CLUSTERED INDEX [IX\_OneIndex\_ID] ON [dbo].[OneIndex] (  
[ID] ASC  
) ON [PRIMARY] GO

Now, run following select on the table once again.

SELECT ID, FirstName  
FROM OneIndex  
WHERE City = 'Las Vegas'  
GO



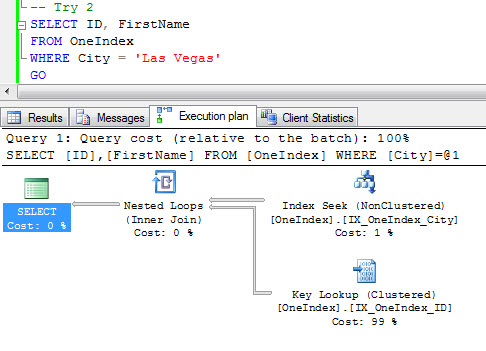
It is clear from execution plan that as a clustered index is created on the table, table scan is now converted to clustered index scan. In either case, base table is completely scanned and there is no seek on the table.

Now, let us see the WHERE clause of our table. From our basic observation, if we create an index on the column that contains the clause, a performance improvement may be obtained. Let us create non-clustered index on the table and then check the execution plan.

-- Create Index on Column City As that is used in where condition  
CREATE NONCLUSTERED INDEX [IX\_OneIndex\_City] ON [dbo].[OneIndex] (  
[City] ASC  
) ON [PRIMARY] GO

After creating the non-clustered index, let us run our select statement again and check the execution plan.

SELECT ID, FirstName  
FROM OneIndex  
WHERE City = 'Las Vegas'  
GO



As we have an index on the WHERE clause, the SQL Server query execution engine uses the non-clustered index to retrieve data from the table. However, the columns used in the SELECT clause are still not part of the index, and to display those columns, the engine will have to go to the base table again and retrieve those columns. This particular behavior is known as bookmark lookup or key lookup.

There are two different methods to resolve this issue. I have demonstrated both the methods together; however, it is recommended that you use any one of these methods for removing key lookup. I prefer Method 2.

**Method 1: Creating non-clustered cover index.**

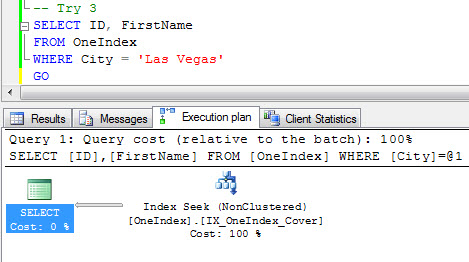
In this method, we will create non-clustered index containing the columns, which are used in the SELECT statement, along with the column which is used in the WHERE clause.

CREATE NONCLUSTERED INDEX [IX\_OneIndex\_Cover] ON [dbo].[OneIndex] (  
City, FirstName, ID  
) ON [PRIMARY] GO

Once the above non-clustered index, which covers all the columns in query, is created, let us run the following SELECT statement and check our execution plan.

SELECT ID, FirstName  
FROM OneIndex  
WHERE City = 'Las Vegas'  
GO

From the execution plan, we can confirm that key lookup is removed the only index seek is happening. As there is no key lookup, the SQL Server query engine does not have to go to retrieve the data from data pages and it obtains all the necessary data from index itself.

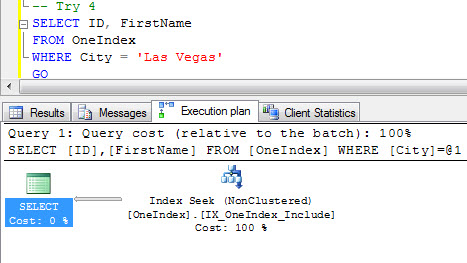


**Method 2: Creating an included column non-clustered index.**

Here, we will create non-clustered index that also includes the columns, which are used in the SELECT statement, along with the column used in the WHERE clause. In this method, we will use new syntax introduced in SQL Server 2005. An index with included nonkey columns can significantly improve query performance when all columns in the query are included in the index.

CREATE NONCLUSTERED INDEX [IX\_OneIndex\_Include] ON [dbo].[OneIndex] (  
City  
) INCLUDE (FirstName,ID) ON [PRIMARY] GO

From the execution plan, it is very clear that this method  also removes the key lookup as well.



In summary, Key lookup, Bookmark lookup or RID lookup reduces the performance of query, and we can improve the performance of query by using included column index or cover index.

**When we drop clustered index on the column, does it automatically drop primary key on the same column (if created)?**

* It is not possible to drop clustered index if there is a primary key on the same table. If your primary key is in a different column, then clustered index, you can for sure manipulate them.
* If you try to drop clustered index on the column, which is also primary key, it will give you an error
* In a simple words, we can’t drop clustered index if it is on Primary Key. We will have to remove the primary key first if we want to drop clustered index.

CREATE TABLE [dbo].[Users]

(

[UserID] [INT] NOT NULL ,

[UserName] [VARCHAR](50) NOT NULL ,

[UserPassword] [VARCHAR](50) NOT NULL ,

PRIMARY KEY CLUSTERED ( [UserID] )

)

GO

CREATE TABLE [dbo].[Users]

(

[UserID] [INT] NOT NULL ,

[UserName] [VARCHAR](50) NOT NULL ,

[UserPassword] [VARCHAR](50) NOT NULL ,

CONSTRAINT PK\_UserID

PRIMARY KEY CLUSTERED ( [UserID] )

)

GO

ALTER TABLE dbo.Users DROP CONSTRAINT PK\_UserID

go

CREATE CLUSTERED INDEX IX\_UserName ON [dbo].[Users] (UserName)

ALTER TABLE dbo.Users ADD CONSTRAINT PK\_UserID

PRIMARY KEY NONCLUSTERED (UserID)

## Index Types

In addition to an index being clustered or nonclustered, it can be configured in other ways:

* **Composite index:** An index that contains more than one column. In both SQL Server 2005 and 2008, you can include up to 16 columns in an index, as long as the index doesn’t exceed the 900-byte limit. Both clustered and nonclustered indexes can be composite indexes.
* **Unique Index:** An index that ensures the uniqueness of each value in the indexed column. If the index is a composite, the uniqueness is enforced across the columns as a whole, not on the individual columns. For example, if you were to create an index on the FirstName and LastName columns in a table, the names together must be unique, but the individual names can be duplicated.

A unique index is automatically created when you define a primary key or unique constraint:

* + **Primary key:** When you define a primary key constraint on one or more columns, SQL Server automatically creates a unique, clustered index if a clustered index does not already exist on the table or view. However, you can override the default behavior and define a unique, nonclustered index on the primary key.
  + **Unique:** When you define a unique constraint, SQL Server automatically creates a unique, nonclustered index. You can specify that a unique clustered index be created if a clustered index does not already exist on the table.
* **Covering index:** A type of index that includes all the columns that are needed to process a particular query. For example, your query might retrieve the FirstName and LastName columns from a table, based on a value in the ContactID column. You can create a covering index that includes all three columns.

# unique indexes vs. unique constraints:

* A unique index ensures that the values in the index key columns are unique.
* A unique constraint also guarantees that no duplicate values can be inserted into the column(s) on which the constraint is created. When a unique constraint is created a corresponding unique index is automatically created on the column(s).

# SQL Server by default creates a unique non-clustered index (if there is no clustered index on the table, we can create a unique index/key as clustered) on the selected column(s).  In this situation, the type of index does not impact the query plan selected by the SQL Server engine and there were no performance differences based on my testing.

# **Q6) Can you tell something about the Primary key in SQL and what is its significance?**

# It is basically an array or a group of fields that generally specify a row. It is considered as one of the unique keys that always have some defined or specific value. Generally, the users need not to worry about anything when it is enabled as it cannot have a null value. It is capable to identify all the records in a database simply and the users are free to get the best possible outcome with minimum efforts. This is exactly what that makes sure of uniqueness.

## Creating Indexes on Persisted Computed Columns

You can create an index on a computed column that is defined with a deterministic, but imprecise, expression if the column is marked PERSISTED in the CREATE TABLE or ALTER TABLE statement. This means that the Database Engine stores the computed values in the table, and updates them when any other columns on which the computed column depends are updated. The Database Engine uses these persisted values when it creates an index on the column, and when the index is referenced in a query. This option enables you to create an index on a computed column when Database Engine cannot prove with accuracy whether a function that returns computed column expressions, particularly a CLR function that is created in the .NET Framework, is both deterministic and precise.

# [How does SQL Server store a composite primary key?](https://stackoverflow.com/questions/3384266/how-does-sql-server-store-a-composite-primary-key)

A composite key is exactly the same as a ordinary, single column, key, but is longer and composed of multiple values. Consider you have a B-Tree on a single column, say A. The non-leaf pages contains slots with values of A column, and pointers to leaf-pages. The leaf pages contain slots with the column A values (the key), followed by all the other columns of the row. A composite key is exactly the same, but the values in the slots will be the composite values, in the order they are declared in the key

**Parallel Index Operations**

# The number of processors used to run a single index statement is determined by the [max degree of parallelism](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-max-degree-of-parallelism-server-configuration-option?view=sql-server-2017) configuration option, the current workload, and the index statistics. The max degree of parallelism option determines the maximum number of processors to use in parallel plan execution. If the SQL Server Database Engine detects that the system is busy, the degree of parallelism of the index operation is automatically reduced before statement execution starts. The Database Engine can also reduce the degree of parallelism if the leading key column of a non-partitioned index has a limited number of distinct values or the frequency of each distinct value varies significantly.

# Table Scan: A table scan is a pretty straightforward process. When your query engine performs a table scan it starts from the physical beginning of the table and goes through every row in the table. If a row matches the criterion then it includes that into the result set. It’s the fastest way to retrieve data especially if your table is quite small. It starts being bad when your table starts growing. In a small table, query engine can load all data in just one shot but in a large table, it's not possible, which means more IO and more time to process those data. Normally, a full table scan is used when your query doesn't have a [WHERE clause](http://javarevisited.blogspot.sg/2017/01/a-better-way-to-write-sql-queries.html#axzz51i34qgwV) i.e. you want more or less every record from a table. Btw, if your query is taking too long in a large table then most likely it using either table scan or index scan.

# Index Scan: If your table has a clustered index and you are firing a query which needs all or most of the rows i.e. query without WHERE or HAVING clause, then it uses an index scan. It works similar to the table scan, during the query optimization process, the query optimizer takes a look at the available index and chooses the best one, based on information provided in your joins and where clause, along with the statistical information database keeps.

# Once the right index is chosen, SQL Query processor or engine navigates the tree structure to the point of data that matches your criteria and again extract only the records it needs.

# The main difference between a full table scan and an index scan is that because data is sorted in index tree, the query engine knows when it has reached the end of the current it is looking for. It can then send the query, or move on to the next range of data as necessary. Index Seek: When your search criterion matches an index well enough that the index can navigate directly to a particular point in your data, that's called an index seek. It is the fastest way to retrieve data in a database. The index seeks are also a great sign that your indexes are being properly used. This happens when you specify a condition in [WHERE](http://javarevisited.blogspot.sg/2013/08/difference-between-where-vs-having-clause-SQL-databse-group-by-comparision.html#axzz51i34qgwV) clause e.g. searching an employee by id or name if you have a respective index.

## Difference between table scan, index scan, and index seek in SQL:

# Based upon our understanding of indexes, you can now deduce following points to summarize the difference between table scan, index scan, and index seek in a database: 1) Table scan and an Index scan is used when you need to retrieve all data e.g. 90% to 100% while index seek is used when you need to retrieve data based upon some conditions e.g. 10% of data. 2) If your query doesn't have WHERE clause and your table doesn't have a clustered index then a full table scan is used, if it does have a [clustered index](http://javarevisited.blogspot.sg/2013/08/difference-between-clustered-index-and-nonclustered-index-sql-server-database.html#axzz51i34qgwV) then index scan is used. 3) Index scan is faster than a table scan because they look at sorted data and query optimizer knows when to stop and look for another range. 4) Index seek is the fastest way to retrieve data and it comes into picture when your search criterion is very specific. Normally, when you have WHERE clause in your query and you are using a column which also has an index, then index seek is used to retrieve data as shown in the following query

# Difference between actual number of rows and estimated number of rows

Row estimates are based on statistics. Inaccurate estimates can come from a number of factors:

* Low cardinality in your data or stats
* Out of date or incomplete stats
* Inefficient execution plan
* Concurrency issues (changes in data from creation of execution plan to execution time)

As a rule, don't worry about it until you have issues.

It's most often based on the cardinality in your stats.

If you are selecting based on two fields, both of which have indexes, the row estimate will be based on a product of the likelihood of the values in the respective indexes, times the total number of rows.

Low cardinality indexes are those **indexes with relatively few unique values**. Columns like account enabled or published are likely to have only two unique values (yes and no). An index on such a column would have a cardinality of 2.

**High cardinality** means that the column contains a large percentage of totally unique values. **Low cardinality** means that the column contains a lot of “repeats” in its data range.

**DBCC FREEPROCCACHE**

Use DBCC FREEPROCCACHE to clear the procedure cache. Freeing the procedure cache would cause, for example, an ad-hoc SQL statement to be recompiled rather than reused from the cache. If observing through SQL Profiler, one can watch the Cache Remove events occur as DBCC FREEPROCCACHE goes to work. DBCC FREEPROCCACHE will invalidate all stored procedure plans that the optimizer has cached in memory and force SQL Server to compile new plans the next time those procedures are run.

Use DBCC DROPCLEANBUFFERS to test queries with a cold buffer cache without shutting down and restarting the server. DBCC DROPCLEANBUFFERS serves to empty the data cache. Any data loaded into the buffer cache due to the prior execution of a query is removed.

• **Logical Reads**

This value indicates the total number of page accesses needed to process the query. Every page is read from the data cache, whether or not it was necessary to bring that page from disk into the cache for any given read. This value is always at least as large and usually larger than the value for Physical Reads. The same page can be read many times (such as when a query is driven from an index), so the count of Logical Reads for a table can be greater than the number of pages in a table.

• **Physical Reads**

This value indicates the number of pages that were read from disk; it is always less than or equal to the value of Logical Reads. The value of the Buffer Cache Hit Ratio, as displayed by Performance Monitor, is computed from the Logical Reads and Physical Reads values as follows:

• **Read Ahead Reads**

The Read Ahead Reads value indicates the number of pages that were read into cache using the read ahead mechanism while the query was processed. These pages are not necessarily used by the query. If a page is ultimately needed, a logical read is counted but a physical read is not. A high value means that the value for Physical Reads is probably lower and the cache-hit ratio is probably higher than... [truncated by vgv8]

• **Scan Count**

The Scan Count value indicates the number of times that the corresponding table was accessed. Outer tables of a nested loop join have a Scan Count of 1. For inner tables, the Scan Count might be the number of times "through the loop" that the table was accessed. The number of Logical Reads is determined by the sum of the Scan Count times the number of pages accessed on each scan. However, even for nested loop joins, the Scan Count for the inner table might show up as 1. SQL Server might copy the needed rows from the inner table into a worktable in cache and use this worktable to access the actual data rows. When this step is used in the plan, there is often no indication of it in the STATISTICS IO output. You must use the output from STATISTIC TIME, as well as information about the actual processing plan used, to determine the actual work involved in executing a query. Hash joins and merge joins usually show the Scan Count as 1 for both tables involved in the join, but these types of joins can involve substantially more memory. You can inspect the memusage value in sysprocesses while the query is being executed, but unlike the physical\_io value, this is not a cumulative counter and is valid only for the currently running query. Once a query finishes, there is no way to see how much memory it used."

**Nested loop Join**

Nested loop join is possible for small tables with index (or) either of the big tables have indexed. It works great for small tables like, compares each row from one table to each row from the other table ‘looping’.

**Merge Join**

Merge join is possible for the tables have an index on the join column. The index either clustered or covering non-clustered index. It's best join for this circumstance. Because it needs an index for both the tables. So it’s already sorted and easily match and return the data.

**Hash Join**

Hash joins is possible for tables with no index (or) either of the big tables has indexed. It’s worked great for big tables with no index and run the query parallel (more than one processor) and give the best performance. Most of folk says its heavy lifter join.

**What is Parameter Sniffing?**

Parameter sniffing is the process whereby SQL Server creates an optimal plan for a stored procedure by using the calling parameters that are passed the first time a stored procedure is executed. By “first time”, I really mean whenever SQL Server is forced to compile or recompile a stored procedures because it is not in the procedure cache. Every subsequent call to the same store procedure with the same parameters will also get an optimal plan, whereas calls with different parameter values may not always get an optimal plan.

The SQL Server engine looks at a query and determines the optimal strategy for execution.  It looks at what the query is doing, uses the parameter values to look at the statistics, does some calculations and eventually decides on what steps are required to resolve the query.  This is a simplified explanation of how an execution plan is created.  The important point for us is that those parameters passed are used to determine how SQL Server will process the query

**How to Deal With Parameter Sniffing**

### *Option 1: With Recompile*

The problem with parameter sniffing is the fact that the first set of parameter values are used to determine the execution plan.  To overcome this problem, all you need to do is to recompile the stored procedure every time it is executed.

The drawback of this option is the store procedure is recompiled with every execution.  This means, you are incurring additional system resources to compile this procedure with each execution. Depending on the performance gains you get with different sets of parameter values, will determine if there is value in using this option to overcome the parameter sniff issue.

### *Option 2: Disabling Parameter Sniffing*

This is not done with a switch or database option, but can be done from within the script of  your stored procedure code.

To disable parameter sniffing, all I did was to change the way the parameter values were used within the stored procedure.  By creating two different local variables (**@StartDate** and **@EndDate**) inside my procedure, setting those variables to the passed parameters, and then using the local variables in the**BETWEEN** condition, I was able to disable parameter sniffing.  Parameter sniffing is disabled because the optimizer is not able to identify the parameters’ values in the actual **SELECT** statement.

### *Option 3: Creating Multiple Stored Procedures*

As I said before, there is no ideal solution to resolving the problems with parameter sniffing.  This option uses multiple stored procedures where each stored procedure can be optimize for a specific type of parameters values.

Cached plans that are created based on one set of parameters may not always run optimally for a different set of parameters, due to parameter sniffing.

**What is the difference between covering and included indexes?**

A covering index is an index that contains all columns referenced in the query. A clustered index is a covering index by definition, but the adjetive is mainly used for non-clustered indexes. The columns in an index can be part of the key, or can be included and be part of the leaf nodes. The columns participating in the key are by default in the leaf nodes too (B-Tree).

If an index is not covering, then SQL Server will have to go to the clustered index, or the table if it is a heap, to pull the values for the rest of the columns that are not part of the index.

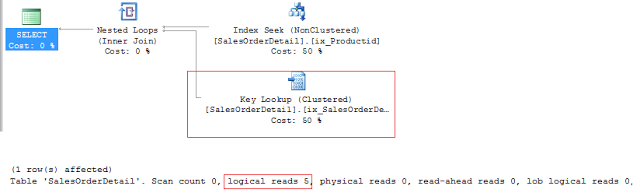
There is no such concept as included index, but an index can use the INCLUDE clause, to incorporate in the leaf nodes, columns that are not part of the key. Before SS 2005, the only way a column can be part of an index was participating in the key.

The columns participating in the key, are referenced, in a query, in the clauses JOIN, WHERE, GROUP BY, HAVING, ORDER BY, etc and the ones that have been included are mainly referenced in the list of columns on the SELECT clause.

### SQL Server : Part 8 : Explaining The Covering Index or Included Columns

In our earlier [discussion](http://www.practicalsqldba.com/2013/03/sql-server-part-4-explaining-non.html)about non clustered index ,we have seen that, the leaf level of a non clustered index contain only the non clustered index key column and clustered index key (if the table is a clustered table). To fetch the remaining column from the clustered index structure or heap structure, SQL server has to do a bookmark/key look up operation.Many time the bookmark or key look up operation might be costly affair. Let us see an example.

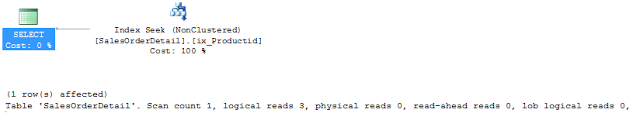
USE mydb  
GO  
DROP TABLE dbo.SalesOrderDetail                                 
GO                                 
SELECT \* INTO dbo.SalesOrderDetail FROM AdventureWorks2008.Sales.SalesOrderDetail  
GO  
CREATE UNIQUE CLUSTERED INDEX ix\_SalesOrderDetail ON dbo.SalesOrderDetail(SalesOrderDetailID)  
GO  
CREATE UNIQUE NONCLUSTERED INDEX ix\_Productid ON dbo.SalesOrderDetail(ProductId,SalesOrderId)  
GO   
SET STATISTICS IO ON  
GO  
SELECT SalesOrderDetailid,productid,salesorderid,orderqty,unitprice FROM SalesOrderDetail WHERE productid=707 AND   
SalesOrderID=43680  
  
The execution plan and the out put of IO statistics of the select statement are given below.

[](http://2.bp.blogspot.com/-88dMeOjZtuo/UVDkt9nm7XI/AAAAAAAADsM/0znHUGdzyaI/s1600/Covering+Index.png)

In the execution plan, you can see that ,50 percent of the query cost is contributed by the Key Lookup operation.In the output of the IO statistics , it clearly says SQL server performed 5 IO operation to fetch the single record.   
  
Note that, the existing non clustered index have 229 pages and depth is 2( levels in the b tree structure).Seek operation on this index need to perform only 2 IO operation to complete the task.You can verify this using the  DBCC IND  command or refer the earlier [post](http://www.practicalsqldba.com/2013/03/sql-server-part-4-explaining-non.html).

Let us assume that, this query(with different parameters ) is used very frequently from the application and you need to optimize it further.How we can do that ? The only way that we can optimize this query is by avoiding the Key lookup operation. For that we can modify our non clustered index and add the remaining two column (OrderQty and UnitPrice) which are not part of clustered index key or non clustered index key.   
  
DROP INDEX ix\_Productid ON dbo.SalesOrderDetail

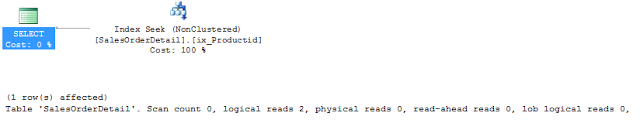
GO  
CREATE UNIQUE NONCLUSTERED INDEX ix\_Productid ON dbo.SalesOrderDetail(ProductId,SalesOrderId,OrderQty ,UnitPrice)  
GO  
SELECT SalesOrderDetailid,productid,salesorderid,orderqty,unitprice FROM SalesOrderDetail WHERE productid=707 ANDSalesOrderID=43680 

[](http://3.bp.blogspot.com/-lm7iG7EbSGQ/UVDs42qxtwI/AAAAAAAADsU/hGpxpPF9o5w/s1600/Covering+Index+2.png)

Now we were  able to get rid of the Key lookup operation from the execution plan and to reduce the IO from 5 to 3.But if we  look into the out put of DBCC IND of the modified non clustered index, we can see that , depth of the b tree is increased by one due to this change. As the index level is increased , the non clustered index has to to perform 3 IO to complete the operation. This will be worst, if we have more column in the select list and we added all those columns into the non clustered index key to avoid the key lookup operation.

Here comes the covering index to help us.Covering index help us to add non key column to leaf level of the non clustered index with very minimal possibility of increasing the depth of the b-tree structure. This can be achieved by adding include column in the CREATE INDEX statement.  
An index that contains all information required to resolve the query is known as a Covering Index.When we create a nonclustered index to cover a query, we can include nonkey columns in the index definition to cover the columns in the query that are not used as primary search columns. Performance gains are achieved because the query optimizer can locate all the required column data within the index; the table or clustered index is not accessed.

DROP INDEX ix\_Productid ON dbo.SalesOrderDetail  
GO  
CREATE UNIQUE NONCLUSTERED INDEX ix\_Productid ON dbo.SalesOrderDetail(ProductId,SalesOrderId)   
include(OrderQty ,UnitPrice)  
GO  
SELECT SalesOrderDetailid,productid,salesorderid,orderqty,unitprice FROM SalesOrderDetail   
WHERE productid=707 AND SalesOrderID=43680 

[](http://2.bp.blogspot.com/--_IQeoUqDWc/UVD3JSi6dSI/AAAAAAAADsk/yDEQv9oOoRU/s1600/Covering+Index+3.png)

With this also, we were able to get rid of the key lookup operation and to reduce the IO operation to 2. The IO operation clearly says the the depth of the clustered index is two.  
Let us see the output of the DBCC IND

SELECT index\_id FROM sys.indexes WHERE name='ix\_Productid' AND OBJECT\_ID= OBJECT\_ID('SalesOrderDetail')  
GO  
DBCC ind('mydb','SalesOrderDetail',2)   
  
This returns 378 records and the root page is 7456 (Value of pagepid column of the record having max value for indexlevel column)

Let us see the root page and one leaf level page

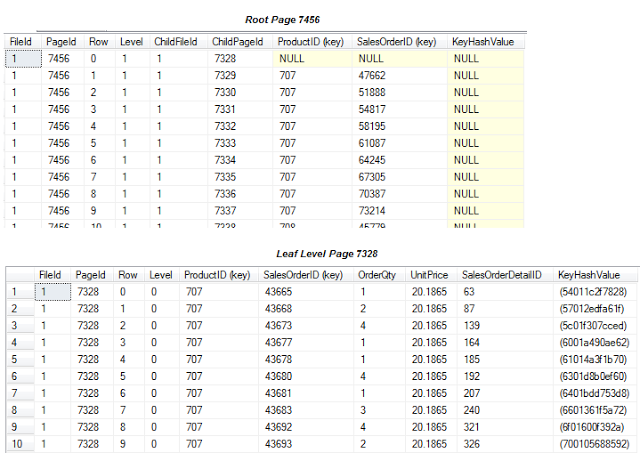
DBCC traceon(3604)

GO

DBCC page ('mydb',1,7456,3)

GO

DBCC page ('mydb',1,7328,3)

[](http://4.bp.blogspot.com/-MZZI0PzTFuQ/UVD8rsItKeI/AAAAAAAADss/dZ7YV51cyLQ/s1600/Covering+Index+4.png)

From the output we can see that, columns mentioned in the include clause are added into the leaf level pages without making any changes in the non leaf level pages.  
  
Include column are useful because we can refer the column that has a data type which can not be used in the index key.More over include columns are not counted in the 900 bytes or 16 key column limitation of index keys. We can include with any data types except text,ntext and image.Included column also support the computed column.

**Find and fix SQL Server 2000 performance issues**

# https://cdn.ttgtmedia.com/digitalguide/images/Misc/siteMap_bullet_section_first.gif Hunt down SQL Server performance problems  https://cdn.ttgtmedia.com/digitalguide/images/Misc/siteMap_vline_spacer_left.gif  https://cdn.ttgtmedia.com/digitalguide/images/Misc/siteMap_bullet_item.gif Home: Introduction  https://cdn.ttgtmedia.com/digitalguide/images/Misc/siteMap_bullet_item.gif Step 1: [CPU usage](https://searchsqlserver.techtarget.com/tutorial/Step-1-CPU-usage)  https://cdn.ttgtmedia.com/digitalguide/images/Misc/siteMap_bullet_item.gif Step 2: [Disk IO queuing](https://searchsqlserver.techtarget.com/tutorial/Step-2-Disk-IO-queuing)  https://cdn.ttgtmedia.com/digitalguide/images/Misc/siteMap_bullet_item.gif Step 3: [Memory consumption](https://searchsqlserver.techtarget.com/tutorial/Step-3-Memory-consumption)  https://cdn.ttgtmedia.com/digitalguide/images/Misc/siteMap_bullet_item.gif Step 4: [Network bandwidth](https://searchsqlserver.techtarget.com/tutorial/Step-4-Network-bandwidth)  https://cdn.ttgtmedia.com/digitalguide/images/Misc/siteMap_bullet_item.gif Step 5: [Transaction-level performance](https://searchsqlserver.techtarget.com/tutorial/Step-5-Transaction-level-performance)

**Step 1: CPU usage**

# Without a doubt, the first metric most IT professionals use to understand SQL Server performance is the overall CPU usage, which is typically determined in one of two ways: First, you may use a remote-desktop technology to review the overall CPU usage in Task Manager.

# Second, you could take the more scientific route and employ these Performance Monitor counters:

* Processor: % Privileged Time
* Processor: % Processor Time
* Processor: % User Time
* System: Context Switches/sec
* System: Processor Queue Length

Unfortunately, this is just the tip of the iceberg. You know that CPU is running at a high percentage or is pegged. Now it is time to dig deeper and determine the status of other key SQL Server metrics.

**Step 2: Disk IO queuing**

# In my opinion, disk activity is the second key general performance metric for SQL Server. To determine general disk activity, once again turn to Performance Monitor, which offers valuable disk drive counters for both physical and logical statistics. These include:

* Physical Disk: % Disk Read Time
* Physical Disk: % Disk Write Time
* Physical Disk: % Idle Time
* Physical Disk: Avg Disk Bytes/Read
* Physical Disk: Avg Disk Bytes/Transfer
* Physical Disk: Avg Disk Bytes/Write
* Physical Disk: Avg Disk Queue Length
* Physical Disk: Current Disk Queue Length

# Another valuable resource that offers specific information related to the activity at a file level in SQL Server is [master.dbo.fn\_virtualfilestats](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_fa-fz_61ih.asp). This function offers the disk IO statistics (reads, writes, etc.) for all database and transaction log files.

**Step 3: Memory consumption**

# The actual amount of memory that SQL Server uses may be deceiving because, by default, SQL Server is configured to capture a high percentage of the total memory on the server. That feature is combined with SQL Server's ability to ensure Windows and other applications installed on the server have sufficient memory. If you are familiar with your applications, you can limit the amount of memory that SQL Server dynamically consumes based on the Memory tab of the Server Properties interface in Enterprise Manager.

A good measure of the macro-level memory usage can be achieved by reviewing the following Performance Monitor counters:

* Memory: Available Bytes
* Memory: Pages/sec
* Process: Working Set
* SQL Server: Buffer Manager: Buffer Cache Hit Ratio
* SQL Server: Buffer Manager: Total Pages
* SQL Server: Memory Manager: Total Server Memory (KB)
* SQLServer:Cache Manager - Cache Hit Ratio - \_Total
* SQLServer:Cache Manager - Cache Pages - \_Total
* Memory: Page Reads/sec
* Memory: Page Writes/sec
* Memory: Page Input/sec
* Memory: Page Output/sec
* Paging File: % Usage

**Step 4: Network bandwidth**

# To determine if a network interface card (NIC) limitation is causing performance issues, use the Task Manager results.

To fine tune the overall network usage, the following Performance Monitor counters should be reviewed:

* Network Interface: Bytes Received/sec
* Network Interface: Bytes Sent/sec
* Network Interface: Bytes Total/sec
* Network Interface: Current Bandwidth
* Network Interface: Output Queue Length

**Step 5: Transaction-level performance**

# The primary tool to capture detailed level information about individual transactions is SQL Profiler. Profiler has can be configured to capture all of the transactions on a server or only a subset based on filtering by application name, host, etc. The information captured by Profiler can be stored in a SQL Server table or a flat file for future reference. This image shows a snippet of Profiler's results.

Process-related information can also be captured by the following commands:

* [Sp\_monitor](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_sp_ma-mz_5lte.asp): Snapshot of SQL Server statistics
* [Sp\_who](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_sp_wa-wz_3v8v.asp): Snapshot of SQL Server processes
* [Master.dbo.sysprocesses](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_sys-p_3kmr.asp): System table storing the SQL Server process-related information
* [DBCC OPENTRAN](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_dbcc_5fhq.asp):Open transactions internal to SQL Server
* [DBCC INPUTBUFFER](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_dbcc_8v8y.asp): Last set of statements sent by the client machine to SQL Server

Below is my list of the top 15 things I believe developers should do as a matter of course to tune performance when coding. These are the low hanging fruit of SQL Server performance – they are easy to do and often have a substantial impact. Doing these won’t guarantee lightening fast performance, but it won’t be slow either.

1. Create a primary key on each table you create and unless you are really knowledgeable enough to figure out a better plan, make it the clustered index (note that if you set the primary key in Enterprise Manager it will cluster it by default).
2. Create an index on any column that is a foreign key. If you know it will be unique, set the flag to force the index to be unique.
3. Don’t index anything else (yet).
4. Unless you need a different behaviour, always owner qualify your objects when you reference them in TSQL. Use dbo.sysdatabases instead of just sysdatabases.
5. Use set nocount on at the top of each stored procedure (and set nocount off) at the bottom.
6. Think hard about locking. If you’re not writing banking software, would it matter that you take a chance on a dirty read? You can use the NOLOCK hint, but it’s often easier to use SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED at the top of the procedure, then reset to READ COMMITTED at the bottom.
7. I know you’ve heard it a million times, but only return the columns and the rows you need.
8. Use transactions when appropriate, but allow zero user interaction while the transaction is in progress. I try to do all my transactions inside a stored procedure.
9. Avoid temp tables as much as you can, but if you need a temp table, create it explicitly using Create Table #temp.
10. Avoid NOT IN, instead use a left outer join – even though it’s often easier to visualize the NOT IN.
11. If you insist on using dynamic sql (executing a concatenated string), use named parameters and sp\_executesql (rather than EXEC) so you have a chance of reusing the query plan. While it’s simplistic to say that stored procedures are always the right answer, it’s also close enough that you won’t go wrong using them.
12. Get in the habit of profiling your code before and after each change. While you should keep in mind the depth of the change, if you see more than a 10-15% increase in CPU, Reads, or Writes it probably needs to be reviewed.
13. Look for every possible way to reduce the number of round trips to the server. Returning multiple resultsets is one way to do this.
14. Avoid index and join hints.
15. When you’re done coding, set Profiler to monitor statements from your machine only, then run through the application from start to finish once. Take a look at the number of reads and writes, and the number of calls to the server. See anything that looks unusual? It’s not uncommon to see calls to procedures that are no longer used, or to see duplicate calls. Impress your DBA by asking him to review those results with you.

# If you take these 15 steps, you’ve made a really good first pass.

# Top 10 SQL Server Performance Tuning Tips

We're going to get back to the basics of performance tuning by learning 10 tips that will help you get off on the right foot when you try to identify existing performance issues and prevent future ones. Before we start, take a look at the ["SQL Server 2005 Waits and Queues" white paper on the SQL Server Customer Advisory Team (CAT) website](http://sqlcat.com/whitepapers/archive/2007/11/19/sql-server-2005-waits-and-queues.aspx). It contains example code and much more detail than I can provide here. I highly recommend that you read this white paper and use it as a reference for this article.

**YOUR FREE GUIDE TO SQL SERVER 2016 REPORTING SERVICES**

[Learn everything you need to know about SQL Server 2016 Reporting Services with our exclusive free guide.](http://sqlmag.com/resources/sql-server-2016-reporting-services?code=ED_SQLWB_N16_performance)

### Tip 1: Stop Waiting Around

Every time that SQL Server tries to do something but gets held up for any reason, it tracks the incident in the form of something known as wait statistics. (See the [CAT white paper](http://sqlcat.com/whitepapers/archive/2007/11/19/sql-server-2005-waits-and-queues.aspx) for more information about wait statistics.) This is one of the areas of SQL Server that you must begin to understand to correctly determine the resources that SQL Server is waiting on at any given moment. For example, if you find that most of your waits are related to a page\_IO\_latch issue, you can be pretty sure that your bottleneck has to do with I/O. And, if you see many LCK\_XX type waits occur, you're seeing a blocking issue. In this case, you would spend your time more wisely by investigating the cause of the blockage instead of by looking at the I/O issues.

The [CAT white paper](http://sqlcat.com/whitepapers/archive/2007/11/19/sql-server-2005-waits-and-queues.aspx) contains plenty of detail about the different wait types, and also code examples for capturing and deciphering the results. After you've read the white paper, examine your waits from time to time so that you can stop chasing your tail and start narrowing down where the real bottleneck lies. This is, without a doubt, where you should start your performance tuning day.

### Tip 2: Locate I/O Bottlenecks

[I/O bottlenecks are one of the key reasons why performance suffers](http://sqlmag.com/article/monitoring-and-analysis/i-o-i-o-it-s-why-my-server-s-slow-103396) in SQL Server. You have three relatively easy methods at your disposal to determine whether you have I/O issues:

* Check whether you see high page\_IO\_latch waits or log\_write waits in your wait statistics.
* Use the DMF sys.dm\_io\_virtual\_file\_stats() to locate any areas in which you have excessive physical I/O or excessive stalls on that I/O. These issues can occur at the database level or even at the file level.
* Use the trusty PerfMon counters. At a minimum, use the Avg. Disk sec/Read and Avg. Disk sec/Write counters to see the latency of the reads and writes. On an OLTP system, you would, ideally, want to see log file latency to be just a few ms and data file latency to be less than 10ms. Remember that these are ideal values. Your system might tolerate larger latency and still be fine. Also keep in mind that many times when you find that the storage subsystem can't keep up with the current demand, the cause might not be an I/O bottleneck at all. It might, instead, be pooled.

When you find that you have many physical I/O bottlenecks occurring, your first instinct should be to find the queries that are causing all the physical I/O, and then try to tune them before you add more hardware. One performance aspect that you should never ignore is high latency for log writes. If you start to hold up the writing to the log file, all further DML operations can quickly become impeded, and they’ll remain so until you alleviate the bottleneck. High latency in log writes is a sure way to hinder performance in the database. For a better understanding of storage issues and I/O subsystems in a SQL Server environment, I recommend that you read the Microsoft articles "[Storage Top 10 Practices](http://technet.microsoft.com/en-us/library/cc966534.aspx)" and "[SQL Server Best Practices Article](http://technet.microsoft.com/en-us/library/cc966412.aspx)."

### Tip 3: Root Out Problem Queries

In any given SQL Server instance, there are probably 8 to 10 queries or stored procedures that are responsible for 80 to 90 percent of the poorly tuned activity that you see throughout the day. Naturally, this isn't true for everyone, but it’s true often enough to matter. If you can identify these problem queries, and if you can prioritize tuning them, you can make a significant impact on the overall performance of your server.

The [CAT white paper](http://sqlcat.com/whitepapers/archive/2007/11/19/sql-server-2005-waits-and-queues.aspx) contains information about this subject. But one way that you can easily identify expensive statements is by using the code in Listing 1.

SELECT COALESCE(DB\_NAME(t.[dbid]),'Unknown') AS [DB Name], ecp.objtype AS [Object Type], t.[text] AS [Adhoc Batch or Object Call], SUBSTRING(t.[text], (qs.[statement\_start\_offset]/2) + 1, ((CASE qs.[statement\_end\_offset] WHEN -1 THEN DATALENGTH(t.[text]) ELSE qs.[statement\_end\_offset] END - qs.[statement\_start\_offset])/2) + 1) AS [Executed Statement] , qs.[execution\_count] AS [Counts] , qs.[total\_worker\_time] AS [Total Worker Time], (qs.[total\_worker\_time] / qs.[execution\_count]) AS [Avg Worker Time] , qs.[total\_physical\_reads] AS [Total Physical Reads], (qs.[total\_physical\_reads] / qs.[execution\_count]) AS [Avg Physical Reads] , qs.[total\_logical\_writes] AS [Total Logical Writes], (qs.[total\_logical\_writes] / qs.[execution\_count]) AS [Avg Logical Writes] , qs.[total\_logical\_reads] AS [Total Logical Reads], (qs.[total\_logical\_reads] / qs.[execution\_count]) AS [Avg Logical Reads] , qs.[total\_clr\_time] AS [Total CLR Time], (qs.[total\_clr\_time] / qs.[execution\_count]) AS [Avg CLR Time] , qs.[total\_elapsed\_time] AS [Total Elapsed Time], (qs.[total\_elapsed\_time] / qs.[execution\_count]) AS [Avg Elapsed Time] , qs.[last\_execution\_time] AS [Last Exec Time], qs.[creation\_time] AS [Creation Time] FROM sys.dm\_exec\_query\_stats AS qs JOIN sys.dm\_exec\_cached\_plans ecp ON qs.plan\_handle = ecp.plan\_handle CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) AS t -- ORDER BY [Total Worker Time] DESC -- ORDER BY [Total Physical Reads] DESC -- ORDER BY [Total Logical Writes] DESC -- ORDER BY [Total Logical Reads] DESC -- ORDER BY [Total CLR Time] DESC -- ORDER BY [Total Elapsed Time] DESC ORDER BY [Counts] DESC

The sys.dm\_exec\_query\_stats DMV query contains aggregate performance statistics that are associated with each of the cached query plans in the SQL Server instance. This query easily enables you to order the results in several ways, depending on what type of resource usage you want to concentrate on. The key is to weigh the number of times that the statement was executed versus the average resource usage to better determine which statements would have the largest impact on performance if they were optimized better.

I will caution you not to put too much weight on the total elapsed time or on the overall duration of the statement because other factors, such as blocking, can influence the overall duration. But by using this query, you should be able to quickly identify the top offending statements in your system, and then prioritize the statements so that you can tune them as efficiently as possible.

### Tip 4: Plan To Reuse

Today's database applications deal with an ever-increasing transaction rate as hardware gets more powerful and less expensive. For this reason, you are often trying to pack more and more into a single server, thus increasing the activity. But one of the most expensive operations in terms of memory and CPU usage in the SQL Server instance is the compiling of query plans. We have also seen an increase in applications that can speed development but that often result in calls to the database that make it nearly impossible to reuse the query plan. I'm talking about applications that don't use stored procedures or that don't correctly parameterize their queries. This can result in extremely poor performance, especially as the number of transactions increases. I highly recommend that you also read the ["Plan Caching in SQL Server 2008" white paper](http://msdn.microsoft.com/en-us/library/ee343986.aspx) as an excellent source for describing how SQL Server handles query plans and reuse.

The [CAT white paper](http://sqlcat.com/whitepapers/archive/2007/11/19/sql-server-2005-waits-and-queues.aspx) also goes into some detail about this subject. But the following code example is a quick and easy way to determine which statements aren't reusing query plans:

SELECT b.[cacheobjtype], b.[objtype], b.[usecounts], a.[dbid], a.[objectid], b.[size\_in\_bytes], a.[text]  
FROM sys.dm\_exec\_cached\_plans as b  
CROSS APPLY sys.dm\_exec\_sql\_text (b.[plan\_handle]) AS a  
ORDER BY [usecounts] DESC

This query sorts all of the plans in the procedure cache in descending order of use counts. The use counts column is incremented every time that a plan is reused, and it lets us easily identify which plans have reuse. You can also order the plans by the text column to determine which statements have many similar entries that have a use count of one. This value indicates statements that you call often but that don't reuse the existing plan. After you've identified these statements, you can prioritize which parts of the application you must work on first to get the biggest bang for your buck in terms of plan reuse. Don’t underestimate how seriously a lack of plan reuse can affect performance as the transaction rate increases.

### Tip 5: Monitor Index Usage

The sys.dm\_db\_index\_operational\_stats() DMF is a widely underutilized source of information. It can provide you valuable information about your index usage. By using this DMF, you can decipher all kinds of information to determine not only which indexes are used but also how they're used. For example, are you scanning the index or are you using seeks? This DMF will tell you. It will even tell you things such as the time elapsed for processes, such as latching and locking. Have you ever asked yourself, "Is this index being used?" or, "Which indexes am I using for a given table?" We've all asked these questions at one time or another. So you can use this DMF to get a handle on your index usage.

### Tip 6: Separate Data and Log Files

One of the most basic but often disregarded rules for good performance is to separate the data and the log files onto separate physical drive arrays whenever possible. This is especially true when you use DAS, but it also applies to a SAN. The key principle here is to separate the mostly random access of the data files from the sequential access that occurs by writing to the transaction logs. One aspect familiar to a SAN environment is that even though you're presented with different drive letters or LUNs, you can't be sure that these represent different physical drive arrays.  Often, these apparent drives are carved from the same larger disk array, and this will defeat the intent of separating them in the first place. So make sure that you know what you're really getting when you ask for your storage on a SAN. You'll be amazed at how much difference this can make as the volume of your transactions increases.

### Tip 7: Use Separate Staging Databases

I see far too many production databases that are also used for scratch space when it comes to importing or exporting data. People frequently import data into a table that is real but temporary in nature. Then, they proceed to manipulate the data by performing extensive updates to prepare it for one of the final production tables. The problem with this is that most production databases are in full recovery mode. This means that practically all activity is fully logged in the transaction log, and these operations can be expensive.

By using a separate staging database within the SQL Server instance that's in simple recovery mode, you can achieve several performance benefits. One is that you can often get a minimally logged load instead of a fully logged load. This can dramatically speed up the import process to begin with. With a minimally logged load, the amount of data that's logged to the transaction log is very small in relation to what it would be for a fully logged load, so you have a much lighter burden on the server as a whole.

If you were to perform these operations in the production database, all that activity would have to use resources such as I/O, CPU, and memory. All the preparation would be fully logged in either database; but at least in the staging database, you wouldn't have to back up the log. If this were done in the production database instead, that activity would increase the time required to back up the log. Therefore, the resulting backup files would be larger than necessary. Create a staging database, and perform your initial importing and manipulation in that database to reduce the overall burden on the server and to minimize the impact to the production databases.

### Tip 8: Pay Attention to Log Files

Too many people underestimate the importance of the transaction log file in relation to performance. The most common mistake people make is not leaving enough free space in the transaction log file for your normal operations to occur without forcing an auto-grow operation. Growing the log file can be extremely time-consuming. It can force any DML operations to wait until the growth is complete before the operation can proceed.

By making sure that you always have plenty of free space in the log file, you can avoid these performance hits altogether. The second most common mistake is having far too many virtual log files (VLFs) in the transaction log. For an excellent explanation of this practice, see Kimberly L.Tripp's blog post, "[Transaction Log VLFs - too many or too few?](http://www.sqlskills.com/BLOGS/KIMBERLY/post/Transaction-Log-VLFs-too-many-or-too-few.aspx)" Personally, I prefer to have the VLFs set at 512MB each. You can obtain this value by growing the log file in 8GB increments.

### Tip 9: Minimize tempdb Contention

If your application makes heavy use of tempdb, it's possible that you could run into some contention regarding internal structures that are associated with the tempdb files. There are ways to minimize the contention, outlined in the TechNet topic "[Working with tempdb in SQL Server 2005](http://technet.microsoft.com/en-us/library/cc966545.aspx)."

This topic goes into great detail about many aspects of tempdb. However, I want to point out that the article recommends that you create one tempdb data file for each processor core. In many cases, this is excessive and can actually degrade performance. What I recommend is that you keep an eye on the page\_latch\_UP waits for tempdb and that you increase the number of files until the wait either ends or is close to zero. Just remember to make all the files the exact same size; otherwise, you don't experience any benefit because of the allocation algorithm that's based on the amount of free space in each file. Another thing to keep in mind is not to explicitly drop temporary tables in your code if you can help it. Let SQL Server do that on its own. In this way, you can maximize the effect of the caching enhancements that are available in recent versions of SQL Server.

### Tip 10: Change the MAX Memory Limit

There have been improvements in the 64-bit versions of SQL Server regarding memory allocation and sharing with the OS and other applications, but I've yet to see where leaving the MAX Memory setting at the default is ideal in real life. Even though your host server might be dedicated to SQL Server, there are always other applications or parts of the OS that require memory from time to time or even all the time. Do yourself a favor and set the MAX memory setting to at least 1 to 2GB less than the total amount of memory on the server if this is a single instance. If you have multiple instances, or if you're in a multi-instance cluster, you also have to account for that. How much memory you leave depends on what else you have running and how much memory it requires to operate efficiently. But you can adjust this value up or down over time as you see fit.

### Bonus Tip: Just Say No to Shrinking Data Files

OK, this makes 11 tips. But shrinking data files has been a general bad practice for a long time, and it can really impact performance in one of three ways. The shrinking can be very painful to begin with. But because it can cause a lot of fragmentation, your subsequent queries might suffer as a result. And if you don’t have Instant File Initialization turned on, the resultant growth later can also hinder performance and potentially cause timeouts. Although there are times when shrinking a file might be necessary, make sure that you know the impact before you try it. For more details, see Paul Randal's blog post "[Why you should not shrink your data files](http://www.sqlskills.com/BLOGS/PAUL/post/Why-you-should-not-shrink-your-data-files.aspx)."

### Make These Tips Your Own

By following these guidelines and keeping this information in mind, you should be able to identify the most common performance issues in SQL Server—and prevent or minimize future ones, as well. By now, you will have noticed that some of these topics will require more reading and some actual experience for you to fully grasp the concepts and techniques. But none of these topics are out-of-reach for the average DBA. And even a beginner has to start somewhere. So why not here?

**Identify Bottlenecks:**

Simultaneous access to shared resources causes bottlenecks. However, excessive demands on shared resources cause poor response time and must be identified and tuned.

Causes of bottlenecks include:

* Insufficient resources, requiring additional or upgraded components.
* Resources of the same type among which workloads are not distributed evenly; for example, one disk is being monopolized.
* Malfunctioning resources.
* Incorrectly configured resources.

| Possible bottleneck area | Effects on the server |
| --- | --- |
| Memory usage | Insufficient memory allocated or available to Microsoft SQL Server degrades performance. Data must be read from the disk rather than directly from the data cache. Microsoft Windows operating systems perform excessive paging by swapping data to and from the disk as the pages are needed. |
| CPU utilization | A chronically high CPU utilization rate may indicate that Transact-SQL queries need to be tuned or that a CPU upgrade is needed. |
| Disk input/output (I/O) | Transact-SQL queries can be tuned to reduce unnecessary I/O; for example, by employing indexes. |
| User connections | Too many users may be accessing the server simultaneously causing performance degradation. |
| Blocking locks | Incorrectly designed applications can cause locks and hamper concurrency, thus causing longer response times and lower transaction throughput rates. |

Following are the SQL Server monitoring and tuning tools:

| Tool | Description |
| --- | --- |
| [sp\_trace\_setfilter (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-trace-setfilter-transact-sql) | SQL Server Profiler tracks engine process events, such as the start of a batch or a transaction, enabling you to monitor server and database activity (for example, deadlocks, fatal errors, or login activity). You can capture SQL Server Profiler data to a SQL Server table or a file for later analysis, and you can also replay the events captured on SQL Server step by step, to see exactly what happened. |
| [SQL Server Distributed Replay](https://docs.microsoft.com/en-us/sql/tools/distributed-replay/sql-server-distributed-replay) | Microsoft SQL Server Distributed Replay can use multiple computers to replay trace data, simulating a mission-critical workload. |
| [Monitor Resource Usage (System Monitor)](https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/monitor-resource-usage-system-monitor) | System Monitor primarily tracks resource usage, such as the number of buffer manager page requests in use, enabling you to monitor server performance and activity using predefined objects and counters or user-defined counters to monitor events. System Monitor (Performance Monitor in Microsoft Windows NT 4.0) collects counts and rates rather than data about the events (for example, memory usage, number of active transactions, number of blocked locks, or CPU activity). You can set thresholds on specific counters to generate alerts that notify operators.  System Monitor works on Microsoft Windows Server and Windows operating systems. It can monitor (remotely or locally) an instance of SQL Server on Windows NT 4.0 or later.  The key difference between SQL Server Profiler and System Monitor is that SQL Server Profiler monitors Database Engine events, whereas System Monitor monitors resource usage associated with server processes. |
| [Open Activity Monitor (SQL Server Management Studio)](https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/open-activity-monitor-sql-server-management-studio) | The Activity Monitor in SQL Server Management Studio is useful for ad hoc views of current activity and graphically displays information about:  Processes running on an instance of SQL Server.  Blocked processes.  Locks.  User activity. |
| [Live Query Statistics](https://docs.microsoft.com/en-us/sql/relational-databases/performance/live-query-statistics) | Displays real-time statistics about query execution steps. Because this data is available while the query is executing, these execution statistics are extremely useful for debugging query performance issues. |
| [SQL Trace](https://docs.microsoft.com/en-us/sql/relational-databases/sql-trace/sql-trace) | Transact-SQL stored procedures that create, filter, and define tracing:  [sp\_trace\_create (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-trace-create-transact-sql)  [sp\_trace\_generateevent (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-trace-generateevent-transact-sql)  [sp\_trace\_setevent (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-trace-setevent-transact-sql)  [sp\_trace\_setfilter (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-trace-setfilter-transact-sql)  [sp\_trace\_setstatus (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-trace-setstatus-transact-sql) |
| Error Logs | The Windows application event log provides an overall picture of events occurring on the Windows Server and Windows operating systems as a whole, as well as events in SQL Server, SQL Server Agent, and full-text search. It contains information about events in SQL Server that is not available elsewhere. You can use the information in the error log to troubleshoot SQL Server-related problems. |
| [System Stored Procedures (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/system-stored-procedures-transact-sql) | The following SQL Server system stored procedures provide a powerful alternative for many monitoring tasks:  [sp\_who (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-who-transact-sql): Reports snapshot information about current SQL Server users and processes, including the currently executing statement and whether the statement is blocked.  [sp\_lock (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-lock-transact-sql): Reports snapshot information about locks, including the object ID, index ID, type of lock, and type or resource to which the lock applies.  [sp\_spaceused (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-spaceused-transact-sql):  Displays an estimate of the current amount of disk space used by a table (or a whole database).  [sp\_monitor (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-monitor-transact-sql): Displays statistics, including CPU usage, I/O usage, and the amount of time idle since **sp\_monitor** was last executed. |
| [DBCC (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-transact-sql) | DBCC (Database Console Command) statements enable you to check performance statistics and the logical and physical consistency of a database. |
| [Built-in Functions (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/functions) | Built-in functions display snapshot statistics about SQL Server activity since the server was started; these statistics are stored in predefined SQL Server counters. For example, **@@CPU\_BUSY** contains the amount of time the CPU has been executing SQL Server code; **@@CONNECTIONS** contains the number of SQL Server connections or attempted connections; and **@@PACKET\_ERRORS** contains the number of network packets occurring on SQL Server connections. |
| [Trace Flags (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-traceon-trace-flags-transact-sql) | Trace flags display information about a specific activity within the server and are used to diagnose problems or performance issues (for example, deadlock chains). |
| [Database Engine Tuning Advisor](https://docs.microsoft.com/en-us/sql/relational-databases/performance/database-engine-tuning-advisor) | Database Engine Tuning Advisor analyzes the performance effects of Transact-SQL statements executed against databases you want to tune. Database Engine Tuning Advisor provides recommendations to add, remove, or modify indexes, indexed views, and partitioning. |