Transact-SQL Optimization Tips

- **Use views and stored procedures instead of heavy-duty queries.**
  This can reduce network traffic, because your client will send to server only stored procedure or view name (perhaps with some parameters) instead of large heavy-duty queries text. This can be used to facilitate permission management also, because you can restrict user access to table columns they should not see.

- **Try to use constraints instead of triggers, whenever possible.**
  Constraints are much more efficient than triggers and can boost performance. So, you should use constraints instead of triggers, whenever possible.

- **Use table variables instead of temporary tables.**
  Table variables require less locking and logging resources than temporary tables, so table variables should be used whenever possible. The table variables are available in SQL Server 2000 only.

- **Try to use UNION ALL statement instead of UNION, whenever possible.**
  The UNION ALL statement is much faster than UNION, because UNION ALL statement does not look for duplicate rows, and UNION statement does look for duplicate rows, whether or not they exist.

- **Try to avoid using the DISTINCT clause, whenever possible.**
  Because using the DISTINCT clause will result in some performance degradation, you should use this clause only when it is necessary.

- **Try to avoid using SQL Server cursors, whenever possible.**
  SQL Server cursors can result in some performance degradation in comparison with select statements. Try to use correlated sub-query or derived tables, if you need to perform row-by-row operations.

- **Try to avoid the HAVING clause, whenever possible.**
  The HAVING clause is used to restrict the result set returned by the GROUP BY clause. When you use GROUP BY with the HAVING clause, the GROUP BY clause divides the rows into sets of grouped rows and aggregates their values, and then the HAVING clause eliminates undesired aggregated groups. In many cases, you can write your select statement so, that it will contain only WHERE and GROUP BY clauses without HAVING clause. This can improve the performance of your query.

- **If you need to return the total table's row count, you can use alternative way instead of SELECT COUNT(*) statement.**
  Because SELECT COUNT(*) statement make a full table scan to return the total table's row count, it can take very many time for the large table. There is another way to determine the total row count in a table. You can use sysindexes system table, in this case. There is ROWS column in the sysindexes table. This column contains the total row count for each table in your database. So, you can use the following select statement instead of SELECT COUNT(*): SELECT rows FROM sysindexes WHERE id = OBJECT_ID('table_name') AND indid < 2 So, you can improve the speed of such queries in several times.

- **Include SET NOCOUNT ON statement into your stored procedures to stop the message indicating the number of rows affected by a T-SQL statement.**

This can reduce network traffic, because your client will not receive the message indicating the number of rows affected by a T-SQL statement.

- **Try to restrict the queries result set by <u>using the WHERE</u> clause.**
  This can results in good performance benefits, because SQL Server will return to client only particular rows, not all rows from the table(s). This can reduce network traffic and boost the overall performance of the query.

- **Use the select statements with <u>TOP keyword or the SET ROWCOUNT</u> statement, if you need to return only the first n rows.**
  This can improve performance of your queries, because the smaller result set will be returned. This can also reduce the traffic between the server and the clients.

- **Try to restrict the queries result set by r<u>eturning only the particular columns</u> from the table, not all table's columns.**
  This can results in good performance benefits, because SQL Server will return to client only particular columns, not all table's columns. This can reduce network traffic and boost the overall performance of the query.
    - Indexes
    - avoid more number of triggers on the table
    - unnecessary complicated joins
    - correct use of Group by clause with the select list
    - in worst cases Denormalization

Index Optimization tips

- Every index increases the time it takes to perform INSERTS, UPDATES and DELETES, so the number of indexes should not be very much. Try to use maximum 4-5 indexes on one table, not more. If you have read-only table, then the number of indexes may be increased.

- Keep your indexes as narrow as possible. This reduces the size of the index and reduces the number of reads required to read the index.

- Try to create indexes on columns that have integer values rather than character values.

- If you create a composite (multi-column) index, the order of the columns in the key are very important. Try to order the columns in the key as to enhance selectivity, with the most selective columns to the left most of the key.

- If you want to join several tables, try to create surrogate integer keys for this purpose and create indexes on their columns.

- Create surrogate integer primary key (identity for example) if your table will not have many insert operations.

- Clustered indexes are more preferable than nonclustered, if you need to select by a range of values or you need to sort results set with GROUP BY or ORDER BY.

- If your application will be performing the same query over and over on the same table, consider creating a covering index on the table.

- You can use the SQL Server Profiler Create Trace Wizard with "Identify Scans of Large Tables" trace to determine which tables in your database may need indexes. This trace will show which tables are being scanned by queries instead of using an index.

- You can use sp_MSforeachtable undocumented stored procedure to rebuild all indexes in your database. Try to schedule it to execute during CPU idle time and slow production periods.
  sp_MSforeachtable @command1="print '?' DBCC DBREINDEX ('?')"