SQL Database Performance Tuning for Developers

Database tuning can be an incredibly difficult task, particularly when working with large-scale data where even the most minor change can have a dramatic (positive or negative) impact on performance.

In mid-sized and large companies, most SQL database tuning will be handled by a Database Administrator (DBA). But believe me, there are [plenty of developers](https://www.toptal.com/sql) out there who have to perform DBA-like tasks. Further, in many of the companies I’ve seen that *do* have DBAs, they often struggle to work well with developers—the positions simply require different modes of problem solving, which can lead to disagreement among coworkers.

When working with large-scale data, even the most minor change can have a dramatic impact on performance.

On top of that, corporate structure can also play a role. Say the DBA team is placed on the 10th floor with all of their databases, while the devs are on the 15th floor, or even in a different building under a completely separate reporting structure—it’s certainly hard to work together smoothly under these conditions. In this article, I’d like to accomplish two things:

1. Provide developers with some developer-side database tuning techniques.
2. Explain how developers and DBAs can work together effectively.

**Database Optimization (in the Codebase)**

**Indexes**

If you’re a complete newcomer to databases and even asking yourself “What is SQL tuning?”, you should know that indexing is an effective way to tune your SQL database that is often neglected during development. In basic terms, an [index](https://en.wikipedia.org/wiki/Database_index) is a data structure that improves the speed of data retrieval operations on a database table by providing rapid random lookups and efficient access of ordered records. This means that once you’ve created an index, you can select or sort your rows faster than before.

Indexes are also used to define a primary-key or unique index which will guarantee that no other columns have the same values. Of course, indexing is a vast an interesting topic to which I can’t do justice with this brief description ([here’s](http://stackoverflow.com/questions/1108/how-does-database-indexing-work/1130#1130) a more detailed write-up).

If you’re new to indexes, I recommend using this diagram when structuring your queries:



Basically, the goal is to index the major searching and ordering columns.

Note that if your tables are constantly hammered by INSERT, UPDATE, and DELETE, you should be careful when indexing—you could end up [decreasing performance](http://stackoverflow.com/questions/141232/how-many-database-indexes-is-too-many/141243#141243) as all indexes need to be modified after these operations.

Further, DBAs often drop their indexes before performing million+ row batch inserts to [speed up the insertion process](http://stackoverflow.com/questions/13700575/is-a-good-practice-to-disable-indexes-before-insertion-of-many-records-on-sql-se). After the batch is inserted, they then recreate the indexes. Remember, however, that dropping indexes will affect every query running in that table; so this approach is only recommended when working with a single, large insertion.

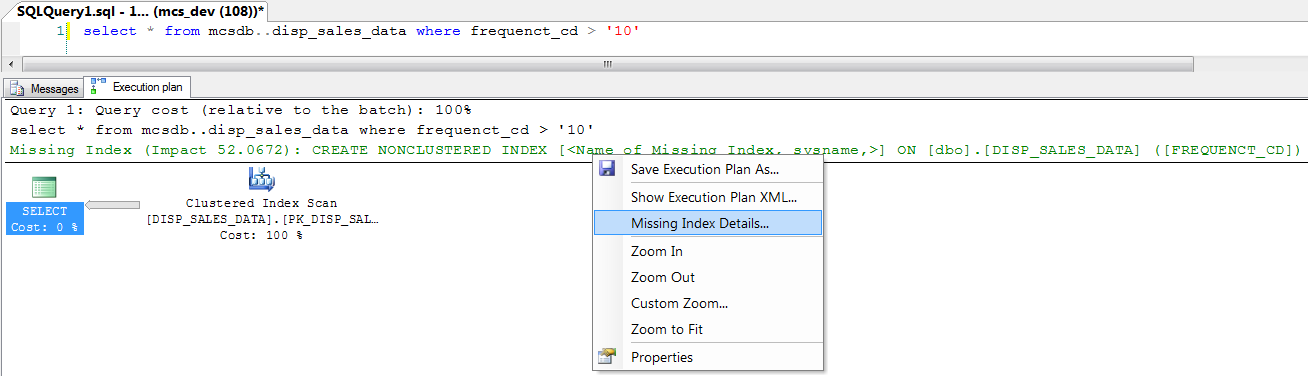
**SQL Server Performance Tuning: Execution Plans**

By the way: the Execution Plan tool in [SQL Server](https://www.toptal.com/sql-server/how-to-tune-microsoft-sql-server-for-performance) can be useful for creating indexes.

Its main function is to graphically display the data retrieval methods chosen by the SQL Server query optimizer. If you’ve never seen them before, [here’s](http://www.brentozar.com/archive/2013/05/the-basics-of-sql-server-execution-plans-video/) a detailed walkthrough.

To retrieve the execution plan (in SQL Server Management Studio), just click “Include Actual Execution Plan” (CTRL + M) before running your query.

Afterwards, a third tab named “Execution Plan” will appear. You might see a detected missing index. To create it, just right click in the execution plan and choose the “Missing Index Details…”. It’s as simple as that!



**Tune SQL query performance by avoiding coding loops**

Imagine a scenario in which 1000 queries hammer your database in sequence. Something like:

for (int i = 0; i < 1000; i++)

{

SqlCommand cmd = new SqlCommand("INSERT INTO TBL (A,B,C) VALUES...");

cmd.ExecuteNonQuery();

}

You should [avoid such loops](http://codeutopia.net/blog/2010/10/07/optimizing-sql-removing-queries-inside-loops/) in your code. For example, we could transform the above snippet by using a unique INSERT or UPDATE statement with multiple rows and values:

INSERT INTO TableName (A,B,C) VALUES (1,2,3),(4,5,6),(7,8,9) -- SQL SERVER 2008

INSERT INTO TableName (A,B,C) SELECT 1,2,3 UNION ALL SELECT 4,5,6 -- SQL SERVER 2005

UPDATE TableName SET A = CASE B

WHEN 1 THEN 'NEW VALUE'

WHEN 2 THEN 'NEW VALUE 2'

WHEN 3 THEN 'NEW VALUE 3'

END

WHERE B in (1,2,3)

Make sure that your WHERE clause avoids updating the stored value if it matches the existing value. Such a trivial optimization can dramatically increase SQL query performance by updating only hundreds of rows instead thousands. For example:

UPDATE TableName

SET A = @VALUE

WHERE

B = 'YOUR CONDITION'

AND A <> @VALUE -- VALIDATION

**Avoid correlated subqueries**

A [correlated subquery](https://en.wikipedia.org/wiki/Correlated_subquery) is one which uses values from the parent query. This kind of query tends to run [row-by-row](http://technet.microsoft.com/en-us/library/ms187638(v=sql.105).aspx), once for each row returned by the outer query, and thus decreases SQL query performance. New developers are often caught structuring their queries in this way—because it’s usually the easy route.

Here’s an example of a correlated subquery:

SELECT c.Name,

c.City,

(SELECT CompanyName FROM Company WHERE ID = c.CompanyID) AS CompanyName

FROM Customer c

In particular, the problem is that the inner query (SELECT CompanyName…) is run for *each* row returned by the outer query (SELECT c.Name…). But why go over the Company again and again for every row processed by the outer query?

A more efficient performance tuning technique would be to refactor the correlated subquery as a join:

SELECT c.Name,

c.City,

co.CompanyName

FROM Customer c

LEFT JOIN Company co

ON c.CompanyID = co.CompanyID

In this case, we go over the Company table just once, at the start, and JOIN it with the Customer table. From then on, we can select the values we need (co.CompanyName) more efficiently.

**Select sparingly**

One of my favorite SQL optimization tips is to avoid SELECT \*! Instead, you should individually include the specific columns that you need. Again, this sounds simple, but I see this error all over the place. Consider a table with hundreds of columns and millions of rows—if your application only really needs a few columns, there’s no sense in querying for *all* the data. It’s a massive waste of resources. (*For more issues, see*[*here*](http://stackoverflow.com/questions/3639861/why-is-select-considered-harmful#answer-3639964)*.*)

For example:

SELECT \* FROM Employees

vs.

SELECT FirstName, City, Country FROM Employees

If you really need every column, explicitly list every column. This isn’t so much a rule, but rather, a means of preventing future system errors and additional database performance tuning. For example, if you’re using an INSERT... SELECT... and the source table has changed via the addition of a new column, you might run into issues, even if that column isn’t needed by the destination table, e.g.:

￼￼￼￼￼￼￼INSERT INTO Employees SELECT \* FROM OldEmployees

Msg 213, Level 16, State 1, Line 1

Insert Error: Column name or number of supplied values does not match table definition.

To avoid this kind of error, you should declare each column individually:

INSERT INTO Employees (FirstName, City, Country)

SELECT Name, CityName, CountryName

FROM OldEmployees

Note, however, that there are some situations where the use of SELECT \* could be appropriate. For example, with temp tables—which leads us to our next topic.

Top of Form

Bottom of Form

**The use of temp(orary) tables**

[Temporary tables](http://www.tutorialspoint.com/sql/sql-temporary-tables.htm) usually increase a query’s complexity. If your code can be written in a simple, straightforward manner, I’d suggest avoiding temp tables.

But if you have a stored procedure with some data manipulation that *cannot* be handled with a single query, you can use temp tables as intermediaries to help you to generate a final result.

When you have to join a large table and there are conditions on said table, you can increase database performance by transferring your data in a temp table, and then making a join on *that*. Your temp table will have fewer rows than the original (large) table, so the join will finish faster!

The decision [isn’t always straightforward](http://stackoverflow.com/questions/11169550/is-there-a-speed-difference-between-cte-subquery-and-temp-tables#answer-11169910), but this example will give you a sense for situations in which you might want to use temp tables:

Imagine a customer table with millions of records. You have to make a join on a specific region. You can achieve this by using a SELECT INTO statement and then joining with the temp table:

SELECT \* INTO #Temp FROM Customer WHERE RegionID = 5

SELECT r.RegionName, t.Name FROM Region r JOIN #Temp t ON t.RegionID = r.RegionID

(*Note: some SQL developers also avoid using SELECT INTO to create temp tables, saying that this command locks the tempdb database, disallowing other users from creating temp tables. Fortunately, this is*[*fixed in 7.0 and later*](http://stackoverflow.com/questions/1302670/sql-server-select-into-and-blocking-with-temp-tables#answer-1302787).)

As an alternative to temp tables, you might consider using a subquery as a table:

SELECT r.RegionName, t.Name FROM Region r

JOIN (SELECT \* FROM Customer WHERE RegionID = 5) AS t

ON t.RegionID = r.RegionID

But wait! There’s a problem with this second query. As described above, we should only be including the columns we need in our subquery (i.e., not using SELECT \*). Taking that into account:

SELECT r.RegionName, t.Name FROM Region r

JOIN (SELECT Name, RegionID FROM Customer WHERE RegionID = 5) AS t

ON t.RegionID = r.RegionID

All of these snippets will return the same data. But with temp tables, we could, for example, create an index in the temp table to improve performance. There’s some good discussion [here](http://stackoverflow.com/questions/11169550/is-there-a-speed-difference-between-cte-subquery-and-temp-tables/11169910#11169910) on the differences between temporary tables and subqueries.

Finally, when you’re done with your temp table, delete it to clear tempdb resources, rather than just wait for it to be automatically deleted (as it will be when your connection to the database is terminated):

DROP TABLE #temp

**Does my record exist?**

This SQL opimization technique concerns the use of EXISTS(). If you want to check if a record exists, use EXISTS() instead of COUNT(). While COUNT() scans the entire table, counting up all entries matching your condition, EXISTS() will exit as soon as it sees the result it needs. This will give you [better performance and clearer code](http://sqlblog.com/blogs/andrew_kelly/archive/2007/12/15/exists-vs-count-the-battle-never-ends.aspx).

IF (SELECT COUNT(1) FROM EMPLOYEES WHERE FIRSTNAME LIKE '%JOHN%') > 0

PRINT 'YES'

vs.

IF EXISTS(SELECT FIRSTNAME FROM EMPLOYEES WHERE FIRSTNAME LIKE '%JOHN%')

PRINT 'YES'

**Database Optimization (in the Office)**

SQL database administrators (DBAs) and developers often clash over data- and non-data-related issues. Drawn from my experience, here are some tips (for both parties) on how to get along and work together effectively.



**For developers:**

1. If your application stops working suddenly, it may not be a database issue. For example, maybe you have a network problem. Investigate a bit before you accuse a DBA!
2. Even if you’re a ninja SQL data modeler, ask a DBA to help you with your relational diagram. They have a lot to share and offer.
3. DBAs don’t like rapid changes. This is natural: they need to analyze the database as a whole and examine the impact of any changes from all angles. A simple change in a column can take a week to be implemented—but that’s because an error could materialize as huge losses for the company. Be patient!
4. Do not ask SQL DBAs to make data changes in a production environment. If you want access to the production database, you have to be responsible for all your own changes.

**For SQL Database Administrators:**

1. If you don’t like people asking you about the database, give them a real-time status panel. Developers are always suspicious of a database’s status, and such a panel could save everyone time and energy.
2. Help developers in a test/quality assurance environment. Make it easy to simulate a production server with simple tests on real-data. This will be a significant time-saver for others as well as yourself.
3. Developers spend all day on systems with frequently-changed business logic. Try to understand this world being more flexible, and be able to break some rules in a critical moment.
4. SQL databases evolve. The day will come when you have to migrate your data to a new version. Developers count on significant new functionality with each new version. Instead of refusing to accept their changes, plan ahead and be ready for the migration.