45 DATABASE PERFORMANCE TIPS FOR DEVELOPERS

PHIL FACTOR

GRANT FRITCHEY

K. BRIAN KELLEY

MICKEY STUEWE

IKE ELLIS

JONATHAN ALLEN

LOUIS DAVIDSON

Database Performance Tips for Developers

As a developer, you may or may not need to go into the database and write queries, or design tables and indexes, or help determine configuration of your SQL Server systems. But if you do, these tips should help to make that a more pain free process.

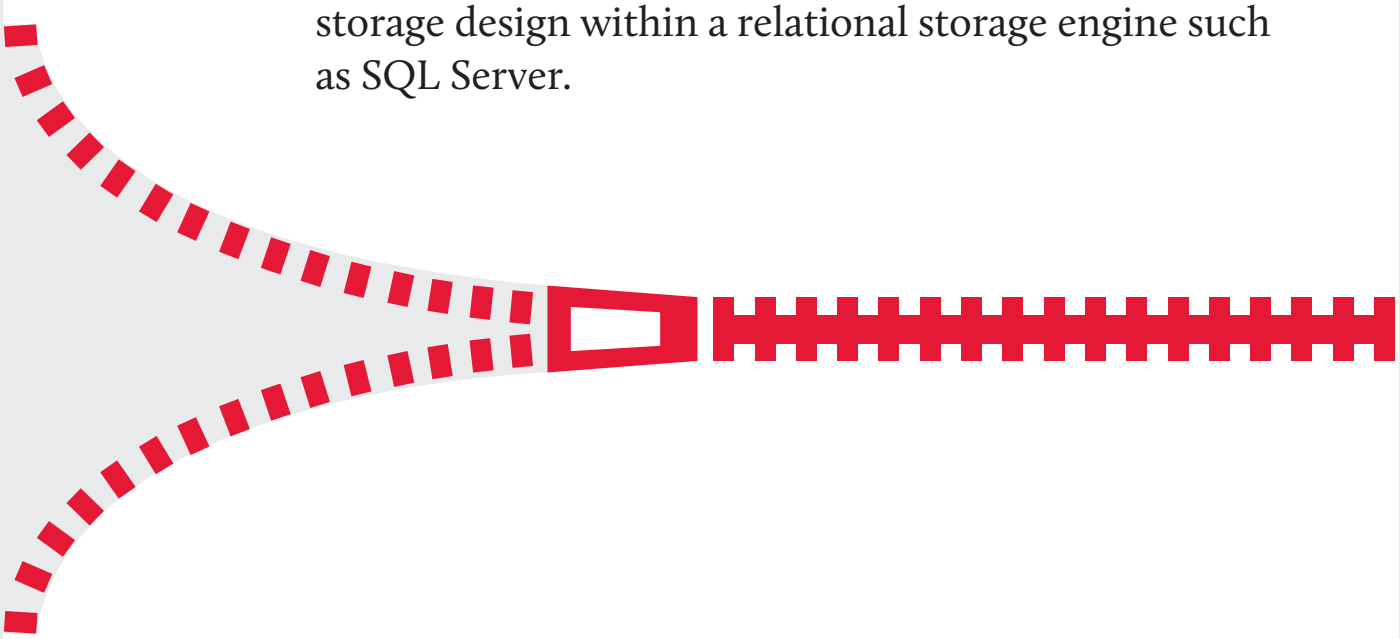
redgate

Contents

ORM Tips

More and more people are using Object to Relational Mapping (ORM) tools to jump the divide between application code that is object oriented and a database that stores information in a relational manner. These tools are excellent and radically improve development speed, but there are a few 'gotchas' to know about.

1. Avoid following the 'Hello World' examples provided with your ORM tool that turns it into an Object to Object Mapping. Database storage is not the same as objects for a reason. You should still have a relational storage design within a relational storage engine such as SQL Server.



2. Parameterized queries are exactly the same as stored procedures in terms of performance and memory management. Since most ORM tools can use either stored procedures or parameterized queries, be sure you're coding to these constructs and not hard-coding values into your T-SQL queries.

3

Create, Read, Update, and Delete (CRUD) queries can all be generated from the ORM tool without the need for intervention. But, the Read queries generated are frequently very inefficient. Consider writing a stored procedure for complex Read queries.

4

Since the code generated from the ORM can frequently be ad hoc, ensure that the SQL Server instance has 'Optimize for Ad Hoc' enabled. This will store a plan stub in memory the first time a query is passed, rather than storing a full plan. This can help with memory management.

5

Be sure your code is generating a parameter size equivalent to the data type defined within the table in the database. Some ORM tools size the parameter to the size of the value passed. This can lead to serious performance problems.

T-SQL Tips

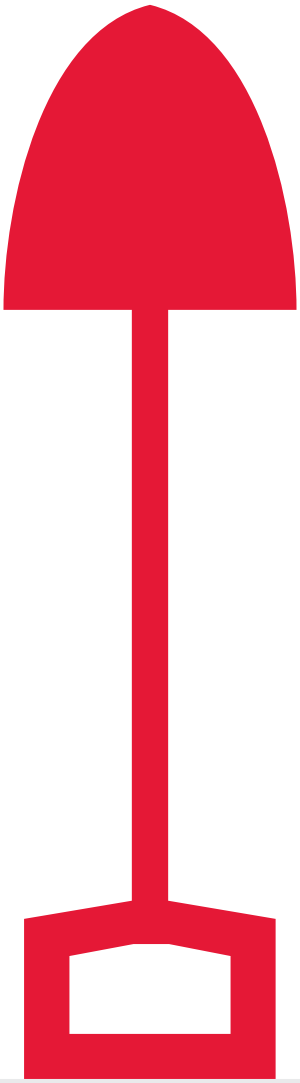
While much of your code may be generated, at least some of it will have to be written by hand. If you are writing some or all of your T-SQL code, these tips will help you avoid problems.

6

SELECT * is not always a bad thing, but it's a good idea to only move the data you really need to move and only when you really need it, in order to avoid network, disk, and memory contention on your server.

7

Keep transactions as short as possible and never use them unnecessarily. The longer a lock is held the more likely it is that another user will be blocked. Never hold a transaction open after control is passed back to the application – use optimistic locking instead.



8

For small sets of data that are infrequently updated such as lookup values, build a method of caching them in memory on your application server rather than constantly querying them in the database.

9

If doing processing within a transaction, leave the updates until last if possible, to minimize the need for exclusive locks.

10

Cursors within SQL Server can cause severe performance bottlenecks. Avoid them. The WHILE loop within SQL Server is just as bad as a cursor.

11

Ensure your variables and parameters are the same data types as the columns. An implicit or explicit conversion can lead to table scans and slow performance.

12

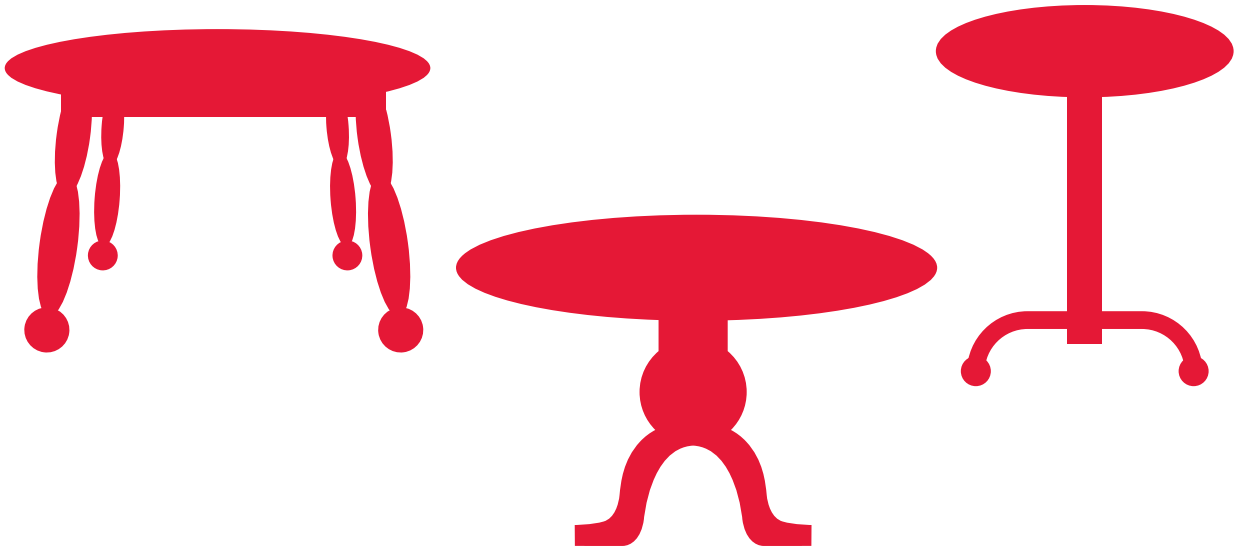
A function on columns in the WHERE clause or JOIN criteria means that SQL Server can't use indexes appropriately and will lead to table scans and slow performance.

13

Don't use DISTINCT, ORDER BY, or UNION unnecessarily.

14

Table variables have no statistics within SQL Server. This makes them useful for working in situations where a statement level recompile can slow performance. But, that lack of statistics makes them very inefficient where you need to do searches or joins. Use table variables only where appropriate.



15

Multi-statement user-defined functions work through table variables. Because of this, they also don't work well in situations where statistics are required. If a join or filtering is required, avoid using them, especially if you are working with more than approximately 50 rows in the data set.

16 Query hints are actually commands to the query optimizer to control how a query plan is generated. These should be used very sparingly and only where appropriate.

17 One of the most abused query hints is NO_LOCK. This can lead to extra or missing rows in data sets. Instead of using NO_LOCK consider using a snapshot isolation level such as READ_COMMITTED_SNAPSHOT.

18 Avoid creating stored procedures that have a wide range of data supplied to them as parameters. These are compiled to use just one query plan.

19 Try not to interleave data definition language with your data manipulation language queries within SQL Server. This can lead to recompiles which hurts performance.

20 Temporary tables have statistics which get updated as data is inserted into them. As these updates occur, you can get recompiles. Where possible, substitute table variables to avoid this issue.

21 If possible, avoid NULL values in your database. If not, use the appropriate IS NULL and IS NOT NULL code.

22 A view is meant to mask or modify how tables are presented to the end user. These are fine constructs. But when you start joining one view to another or nesting views within views, performance will suffer. Refer only to tables within a view.

23 If you need to insert many rows at once into a table, use, where possible, the multi-row VALUES clause in INSERT statements.

24 Use extended events to monitor the queries in your system in order to identify slow running queries. If you're on a 2005 or earlier system you'll need to use a server-side trace.

Index Tips

Indexing tables is not an exact science. It requires some trial and error combined with lots of testing to get things just right. Even then, the performance metrics will change over time as you add more and more data.

25

You get exactly one clustered index on a table. Ensure you have it in the right place. First choice is the most frequently accessed column, which may or may not be the primary key. Second choice is a column that structures the storage in a way that helps performance. This is a must for partitioning data.

26

Clustered indexes work well on columns that are used a lot for 'range' WHERE clauses such as BETWEEN and LIKE, where it is frequently used in ORDER BY clauses or in GROUP BY clauses.

27

If clustered indexes are narrow (involve few columns) then this will mean that less storage is needed for non-clustered indexes for that table.

28

You do not have to make the primary key the clustered index. This is default behavior but can be directly controlled.

29

You should have a clustered index on every table in the database. There are exceptions, but the exceptions should be exceptional.

30

Avoid using a column in a clustered index that has values that are frequently updated.

31 Only create non-clustered indexes on tables when you know they'll be used through testing. You can seriously hurt performance by creating too many indexes on a table.

32 Keep your indexes as narrow as possible. This means reducing the number and size of the columns used in the index key. This helps make the index more efficient.

33 Always index your foreign key columns if you are likely to delete rows from the referenced table. This avoids a table scan.

34 A clustered index on a GUID can lead to serious fragmentation of the index due to the random nature of the GUID. You can use the function NEWSEQUENTIALID() to generate a GUID that will not lead to as much fragmentation.

35 Performance is enhanced when indexes are placed on columns used in WHERE, JOIN, ORDER BY, GROUP, and TOP. Always test to ensure that the index does help performance.

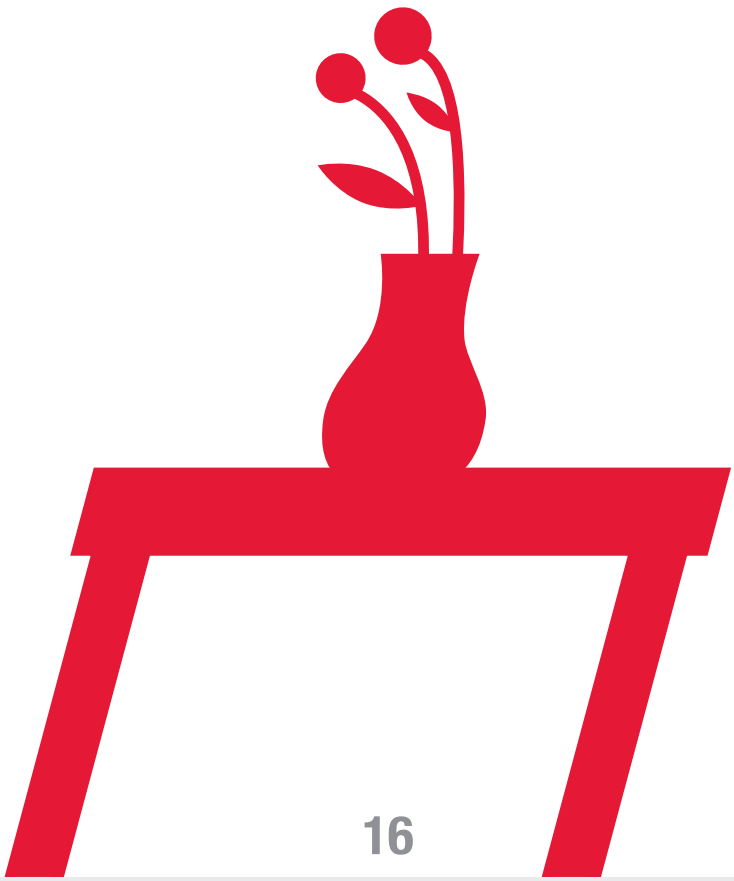
36 If a non-clustered index is useful to your queries, but doesn't have all the columns needed by the query, you can consider using the INCLUDE option to store the extra columns at the leaf level of the index.

37 If temporary tables are in use, you can add indexes to those tables to enhance their performance.

38 Where possible, make the indexes unique. This is especially true of the clustered index (one of the reasons that the primary key is by default clustered). A unique index absolutely performs faster than a non-unique index, even with the same values.

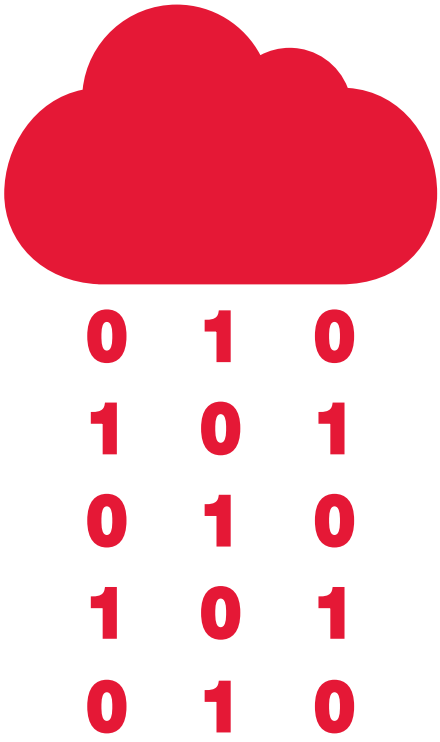
39 Ensure that the selectivity of the data in the indexes is high. Very few unique values makes an index much less likely to be used well by the query optimizer.

40 It is almost always better to let SQL Server update statistics automatically.



Database Design Tips

Again, you may not be delving much into this, but there are a few tips to keep in mind.



41 While it is possible to over-normalize a database, under-normalization is much more prevalent. This leads to repetition of data, inefficient storage, and poor performance. Data normalization is a performance tuning technique as well as a storage mechanism.

1 0 1
0 1 0

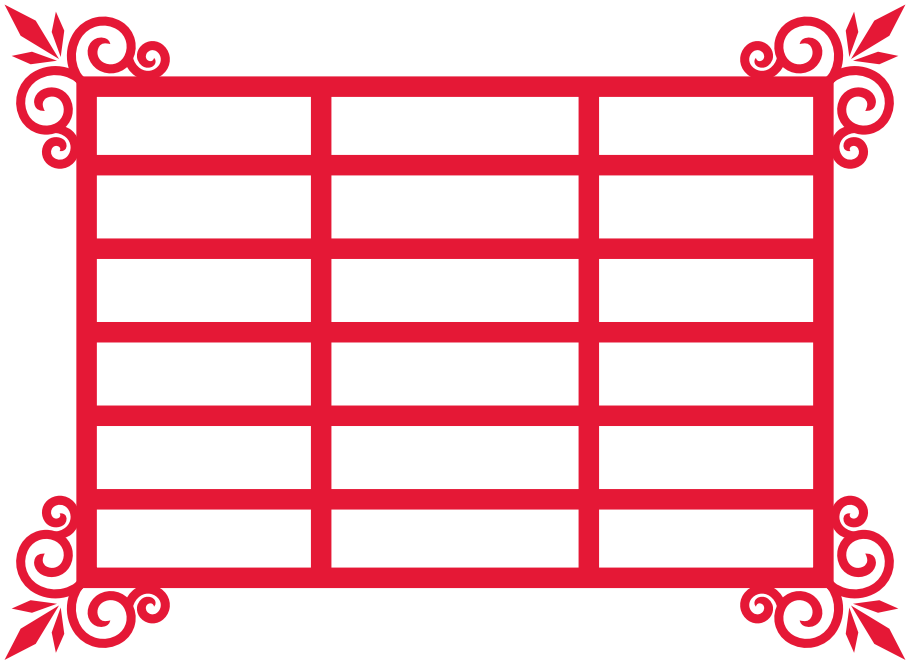
42 Referential integrity constraints such as foreign keys actually help performance, because the optimizer can recognize these enforced constraints and make better choices for joins and other data access.

1 0 1
0 0

43 Unique constraints also help performance because the optimizer can use these in different ways to enhance its query tuning.

44 Make sure your database doesn't hold 'historic' data that is no longer used. Archive it out, either into a special 'archive' database, a reporting OLAP database, or on file. Large tables mean longer table scans and deeper indexes. This in turn can mean that locks are held for longer. Admin tasks such as Statistics updates, DBCC checks, and index builds take longer, as do backups.

45 Separate out the reporting functions from the OLTP production functions. OLTP databases usually have short transactions with a lot of updates whereas reporting databases, such as OLAP and data warehouse systems, have longer data-heavy queries. If possible, put them on different servers.



Tools

SQL Prompt

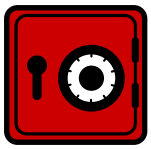
Write, refactor, and explore SQL effortlessly with this SQL Server Management Studio plug-in.

SQL Compare

Compare database schemas and deploy database schema changes.

SQL Source Control

Connect your databases to your version control system within SQL Server Management Studio.



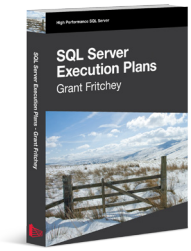
SQL Backup Pro

Schedule backup jobs, verify with DBCC CHECKDB, and compress backups by up to 95%.

SQL Index Manager

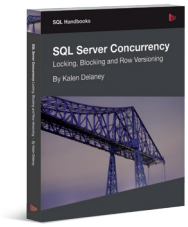
Analyze, manage, and fix database index fragmentation.

Books



SQL Server Execution Plans
by Grant Fritchey

Learn the basics of capturing plans, how to interrupt them in their various forms, graphical or XML, and then how to use the information you find. Diagnose the most common causes of poor query performance so you can optimize your SQL queries and improve your indexing strategy.

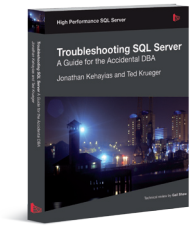


SQL Server Concurrency: Locking, Blocking and Row Versioning
by Kalen Delaney

Your application can have world-class indexes and queries, but they won't help you if you can't get your data because another application has it locked. That's why every DBA and developer must understand SQL Server concurrency, and how to troubleshoot any issues.

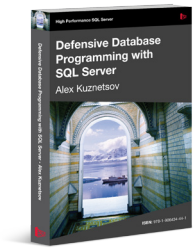
SQL Server Transaction Log Management
by Tony Davis and Gail Shaw

Tony Davis and Gail Shaw strive to offer just the right level of detail so that every DBA can perform all of the most important aspects of transaction log management.



Troubleshooting SQL Server: A Guide for the Accidental DBA
by Jonathan Kehayias and Ted Krueger

Three SQL Server MVPs provide fascinating insight into the most common SQL Server problems, why they occur, and how they can be diagnosed using tools such as Performance Monitor, Dynamic Management Views, and server-side tracing performance, so you can optimize your SQL queries and improve your indexing strategy.



Defensive Database Programming
by Alex Kuznetsov

The goal of defensive database programming is to help you to produce resilient T-SQL code that robustly and gracefully handles cases of unintended use, and is resilient to common changes to the database environment.