solarwinds

eBOOK

Understanding the Fundamentals of Database Performance

by Joey D'Antoni

ActualTech Media

Table of Contents

Introduction

Databases can be a performance bottleneck for a lot of applications, and developers and system administrators don't always understand why. SysAdmins often wonder why the database is consuming so much memory, while developers ask why a query is taking so long. This paper discusses how database engines work and what you need to look for and evaluate when reviewing your workloads. It also looks at the internal operations of the database engine, to see how they can affect overall server performance.

It's important to understand database servers aren't like your other servers, and hence need to be managed differently. As part of that overview, we'll delve into some approaches for optimizing performance and see how to identify anti-patterns in your database queries.

Database Engines and How They Work

What happens when you or your application executes and runs a query? This varies a little, depending on the database engine you're using, but is mostly the same across different engines. The initial process is known as query optimization, and it's how the database engine builds its process to retrieve the rows required to answer the query.

QUERY OPTIMIZATION

Let's start by assuming you've logged in to your database and you're running a query against a sample table called Employee. The following is an example of the data definition language (DDL) that defines the table:

```
CREATE TABLE [Employee](
    [EmployeeID] [int] NOT NULL,
    [LoginID] [nvarchar](256) NOT NULL,
    [JobTitle] [nvarchar](50) NOT NULL,
    [BirthDate] [date] NOT NULL,
    [MaritalStatus] [nchar](1) NOT NULL,
    [Gender] [nchar](1) NOT NULL,
    [HireDate][date] NOT NULL,
    [SalariedFlag] [bit] NOT NULL,
    [VacationHours] [smallint] NOT NULL,
    [SickLeaveHours] [smallint] NOT NULL,
    [IsActive] [bit] NOT NULL)
```

As you can see, each column has a data type defined, which represents how the data is stored within the engine. Some columns are defined as `NOTNULL`, which means they have to be populated to insert data into the table. Once the table is populated, you can run queries against it, such as this query:

```
SELECT EmployeeID FROM Employee
WHERE VacationHours > 40
```

When you execute this query, the database engine parses it, first checking to ensure the SQL syntax is valid. For example, if you typed `ELECT` instead of `SELECT`, the database engine would return an error. Next, the engine checks the system catalog to validate the table and columns exist, are correct, and, in the case of this query, validates the data type of the `WHERE` clause (for example, you couldn't take the average of a string value).

After parsing is complete, we move to the next extremely important step—generating an explain or execution plan. This involves query optimization, which is critical in terms of performance. The database engine uses the statistics it has about the data to generate a set of logical operations which represent the process used to retrieve data from memory and/ or disk. This is based on the number of expected values—if the statistics say there are a small number of records, for example, the optimizer will choose one type of data retrieval action. If, on the other hand, the statistics say the expected values represent most of the rows in the table, it would then be more efficient to scan all of the rows.

The execution plan is the most important factor in tuning query performance, and there are two ways an administrator can influence it: by altering the index or updating column statistics, and by adding or removing indexes.

The output of this process is the execution or explain plan (the name depends on the database engine) and shows operations like scans, seeks, and joins. In SQL Server and Oracle, these plans are cached in a specific execution plan cache—subsequent query executions (based on the hash value of the query) will reuse the same plan. MySQL doesn't have an execution plan cache, and PostgreSQL has limited use of plan caching for prepared statements.

The reason some engines cache plans is because plan generation is a CPU-intensive process. The engine analyzes statistics on the distribution of data in the columns and indexes, and then generates several possible plans. The plan with the lowest execution cost—i.e., how long the query is expected to run—is then chosen and ran.

QUERY EXECUTION

When the query is executed, databases use locks to ensure the atomicity of transactions—you wouldn't want two updates to your bank balance to happen at the same time. If another process is currently updating a record and hasn't yet committed (written the transaction to the database's log file), your second update query will be blocked. This means your query will wait until the prior query is completed before being able to be executed.

Typically, locking doesn't affect queries only reading data, but this depends on the configuration of the specific database. While locking problems are more common on busy transactional systems, they can also occur on less busy systems with poor code or transaction handling. Ideally, when coding for database work, you want to keep transactions to as short a duration as possible to reduce the risks of blocking.

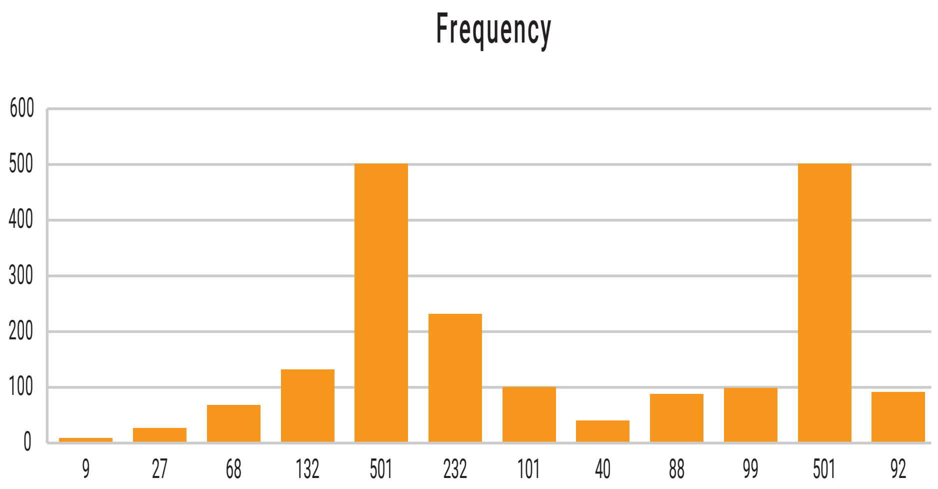


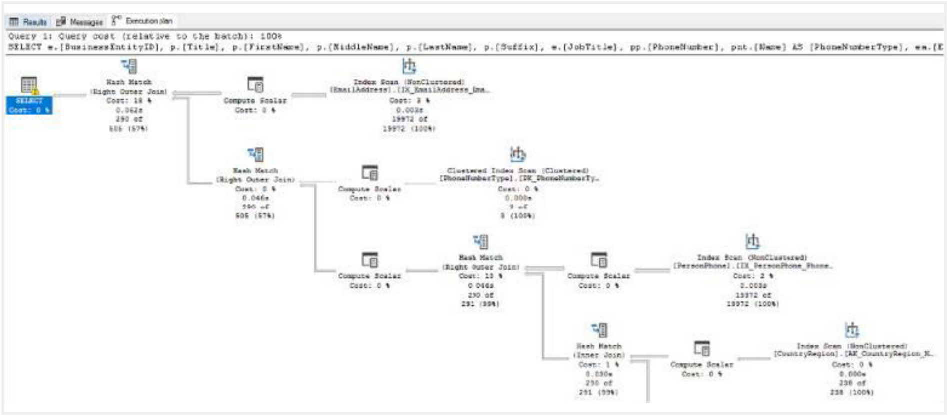
Figure 1: A sample histogram of data points

The execution plan is the most important factor in tuning query performance, and there are two ways an administrator can influence it: by altering the index or updating column statistics, and by adding or removing indexes. Statistics are metadata about the distribution of data in a given column or index to help the optimizer understand how many rows are going to be the result of a given query. Statistics are stored in the database engine as histograms, as shown in **Figure 1.**

Based on these statistics, the optimizer decides which types of joins to choose, how much memory to allocate to perform the joins, and which other operations to perform. Given the effect of those tasks on query performance, you can understand the importance of having good statistics. Some database engines will auto-update statistics as data changes in the tables, but be sure you understand how this process works on your database. Keep in mind, some workloads benefit from updating statistics more frequently than the auto-update process happens, especially with rapidly changing data.

TUNING QUERY PERFORMANCE WITH EXECUTION PLANS

When tuning query performance, the first place to look is typically the execution plan. It can highlight inconsistencies in statistics and show where an index might be missing or other suboptimal patterns that might be taking place when a query is run. A sample execution plan from SQL Server is shown in **Figure 2.**


Results
Messages
Execution plan
SELECT
Cost: 0 %
Hash Match
(Right Outer Join)
Cost: 18 %
Compute Scalar
Cost: 0 %
Index Scan (NonClustered)
[EmailAddress].[IX_EmailAddress_Ema...
Cost: 3 %
Clustered Index Scan (Clustered)
[PhoneNumberType].[PK_PhoneNumberTy...
Hash Match
(Inner Join)
Index Scan (NonClustered)
[PersonPhone].[IX_PersonPhone_Phone...
Index Scan (NonClustered)
[CountryRegion].[AK_CountryRegion_N...

```
Query 1: Query cost (relative to the batch): 100%
SELECT e.[BusinessEntityID], p.[Title], p.[FirstName], p.[MiddleName], p.[LastName], p.[Suffix], e.[JobTitle], pp.[PhoneNumber], pnt.[Name] AS [PhoneNumberType], ea.[E
```

Figure 2: Sample execution plan from a SQL Server database

In the execution plan, a percentage cost is assigned to each operation, and each operation has metadata associated with it, like the number of rows to be retrieved from a given operation. As noted earlier, you can influence an execution plan without changing a query by updating statistics or by creating, changing, or dropping indexes. If you can't improve the performance of a query using these two approaches, you'll need to consider other options to resolve your performance issues.

STATISTICS

Query optimizers use statistics to estimate the *cardinality*, or number of rows, in a given query result. Based on the number of rows in the cardinality estimate, the query optimizer will choose certain types of operations. Statistics can be on a single column, multiple columns, or indexes (which themselves can be on single or multiple columns).

HOW DATABASES CONSUME SERVER RESOURCES

Server or VM administrators often wonder why the database server is using all the memory. The reason is database engines actively manage their own memory and view the memory utilization from the perspective of the database engine's internal metrics.

Database engines manage their own memory to reduce I/O utilization, using caches to reduce the number of round trips to storage. The most common type of cache is the buffer cache, which stores recently used data blocks (also known as pages) so subsequent queries can be served from memory.

Different database engines have different algorithms to determine how they fill this cache. Some engines are more aggressive about retrieving data beyond what's needed to serve the query. In any case, the buffer cache will often be the largest component of database server memory.

Other caches commonly used in database engines include execution plan caches (as mentioned above), which aim to reduce I/O consumption; a buffer for writing to the transaction log; and possibly a results cache for frequently run queries.

Database engines also use memory for query processing. When an execution plan is generated, the engine allocates memory for sorting rows for joint operations and for sorting result sets. This memory is typically allocated using a calculation involving the number of rows per operator and the available memory on the server.

Query processing is an area where poor statistics can impact server performance, as overestimates of rows can lead to overly large memory grants, and this in turn can impact the number of queries that can be run at the same time. Row underestimates (too small memory grants) may force the query-processing engine to spill to disk to complete the sort or join operation. Different database engines use different nomenclature for this temporary space, but the overall architecture is similar for most engines. Additionally, each connection to a database server consumes a small amount of memory, which can consume a significant amount of memory, especially for applications which don't use connection pooling.

All of these caches and grants can add up to a lot of memory—it's not uncommon to set the maximum memory consumption for a database engine to 80% or more of the memory available on a given server or VM. This is normal and lets the database operate most efficiently.

ACID, in the database world, means Atomicity, Consistency, Isolation, and Durability. It's a standard set of properties that guarantee database transactions are processed reliably. ACID is especially concerned with how a database recovers from a failure during transaction processing as well as data quality.

STORAGE UTILIZATION

Database engines are also heavy users of I/O and storage. Though the memory needs of the database engine are pretty high, without memory, your database would be hitting your storage even harder than they do now. The best two hardware components databases benefit most from are memory and fast storage. Having high throughput, low latency storage reduces the latency of queries not served directly from memory.

The other important storage feature is the write speed of the database's transaction log. For a transaction to be considered complete (for the durability aspect of ACID), it has to be persisted to the database's transaction log. This means the ultimate limiter of throughput is for an OLTP workload with a large amount of write is how fast you can write to the transaction log. To improve those writes, the usual recommendation is to put the transaction log on a dedicated storage volume to allow maximum throughput, which also provides additional recovery options.

Backup storage is also critical. While not nearly as performance sensitive, backup storage can consume a lot of volume, particularly in regulated industries where backups have to be retained for an extended period.

Backups are typically stored on a lower tier of storage with higher density—a backup typically doesn't need fast solid-state storage. It's not uncommon to have at least three to four times the amount of raw storage allocated for backups. Note your backups should be stored on a separate storage device than your data and transaction log files, so when a catastrophe occurs, you can still access your backups.

Improving Performance in Database Systems

Now that you have an idea of how database engines work, you can start to think about how to improve the performance of your database systems. While it's possible to scale up by purchasing bigger and faster hardware, there's a limit to how much hardware you can buy, especially if you're using a cloud database offering. In addition, upgrading your hardware will never be a substitute for optimizing your database code.

IDENTIFYING ANTI-PATTERNS IN DATABASE QUERIES

Anti-patterns are ineffective approaches to problem solving. One of the more common anti-patterns seen in database queries is doing row-by-row processing of large datasets. Though this can be a good pattern when processing small arrays of data in application code, in database tables with thousands or millions of rows it dramatically limits the throughput for a set of operations. Here's a SQL code example for updating an account in a row-by-row fashion:

```
SET NOCOUNT ON;
DECLARE @Result AS TABLE (
accountid INT,
mxvalue MONEY );
```

In the above code block, you're declaring a result set as table variable—this can be used like any other table. The table variable has two columns, for AccountID and MXValue.

```
DECLARE
@acccountid AS INT,
@value AS MONEY,
@prevaccountid AS INT,
@prevvalue AS MONEY;

INSERT INTO #TempTable
SELECT accountid, mxvalue
    FROM dbo.Transactions
    ORDER BY accountid, mxvalue;
```

In this section, the code is inserting data into a newly created temporary table called #TempTable, the AccountID and MXValue records from the transaction table. It's also declaring several variables for AccountID, Value, and Previous AccountID and Previous Value.

```
SELECT TOP(1) @prevaccountid = accountid, @
```

```
prevvalue = mxvalue
FROM #TempTable;
```

In the `SELECT TOP(1)` query, the code is populating the Previous AccountID and Previous Value variables.

```
WHILE @prevaccountid NOT IS NULL
BEGIN
IF @accountid @prevaccountid
    INSERT INTO @Result(accountid, mxvalue) VALUES(@
prevaccountid , @prevvalue);

    DELETE #TempTable
    WHERE accountid = @accountid;

    SELECT TOP(1) @prevaccountid = @accountid, @
prevvalue = @value
    FROM #TempTable;
END
```

In this `WHILE` loop, the code will go through each row of the table, and populate the @Result table variable, and then deletes the record from the temporary table and repopulates the variables.

```
    IF @prevaccountid  IS NOT NULL
    INSERT INTO @Result(accountid, mxvalue) VALUES(@
    prevaccountid , @prevvalue);
    SELECT accountid, mxvalue
    FROM @Result;
    GO
```

Finally, the query retrieves the result from the table variable.

This is T-SQL (SQL Server) code, but you may encounter similar patterns in other databases. The code is loading data into a temporary table and then processing the data row by row from the temp table.

To see a more efficient example of how to accomplish the same goal, execute the following:

```
SELECT
T1.AccountID, T1.TransactionID, T1.Value,
SUM(T2.Value) AS balance
FROM dbo.Transactions AS T1
JOIN dbo.Transactions AS T2 ON T2.AccountID =
T1.AccountID AND T2.TransactionID <= T1.TransactionID
GROUP BY T1.AccountID, T1.TransactionID, T1.Value
```

This code executes a self-join of the Transactions table to itself and calculates the updated balance for accounts in a single transaction.

Set-based operations highlight the real power of a relational database engine. This code will execute orders of magnitude faster than the row-based approach. You should aim to eliminate the use of any row-by-row operations in your code, as set-based operations are far superior. Another anti-pattern is having inconsistent data types across your code, which leads to data type conversions capable of impacting query performance. This can creep in if your application language doesn't use strong data types, or you aren't careful with mapping data types from the application to the database.

While `CAST` and `CONVERT` can be used within the database, those operations have a cost, which may be significant in some cases. The use of `SELECT *` instead of specifying a column list in the table(s) is also something better avoided in your code—if the underlying table changes (for example, adds a column), your application could experience breaking errors. Retrieving unneeded additional columns in your queries causes more I/O and can potentially produce errors in your application tier if the columns in the table change. Optimizing your queries is a good start, but the real driver of application performance is having the right indexes for your workload.

To summarize the best practices for building efficient SQL queries:

- Avoid row by row operations; use set-based operations where possible
- Only query for the columns you need
- Ensure your column and index statistics are up to date
- Minimize data type conversions in your code
- Learn how to read an execution plan in your database engine

Understanding Indexes

While tuning your queries is important, without the proper indexes, your database will struggle to scale its performance as the data grows in your database. If you're managing applications where you can't control the source code or queries, which is likely the case with many independent software vendor (ISV) applications, adding indexes can be a very powerful technique for boosting your database performance. You should check with your software vendor about their support of your adding or removing indexes from the application.

Before we dive deeper into indexes, it can be helpful to break down applications into a couple of categories designed to affect the way you think about indexes. Applications tend to be thought of as either of two types—online transaction processing (OLTP) systems and online analytical processing (OLAP) systems.

Some examples of OLTP systems include order entry, retail sales, and financial transaction systems. These systems are characterized by frequent small inserts and updates, and less frequent reporting, with high levels of concurrent usage.

In contrast, the classic example of an OLAP system is a data warehouse, which supports virtually any type of reporting, including for sales and management. OLAP systems are characterized by large volumes of data and, typically, a smaller set of users. To support OLAP systems, ETL processes perform large updates and inserts, along with large sequential reads for report execution.

To an extent, the type of system will affect the approach you use to index, as you can be more aggressive indexing OLAP systems. However, even the most write-intensive OLTP systems might spend 70% of their disk activity reading data.

The reason a database's read/write ratio matters is the cost of indexes comes from the fact that as data is added, the data in the indexes have to be updated or removed from the base table. This means an over-indexed table can delay write operations to the table, which an OLTP system might be more sensitive to.

When considering how to index a database, you should first try to identify the queries you run most often.

With information about the types of database workloads in hand, let's consider how to approach indexing a database.

THE BASICS OF DESIGNING INDEXES

Consider a book with an index at the back. The index allows you to easily find information within the book. That index is a sorted listed of pointers to indicate specific parts of the book.

To simplify the difference between scan and seek operations, let's suppose you want to cook a chicken. If you look up the term "chicken" in a cookbook's index, you can quickly navigate to all of recipes with chicken. This is equivalent to a seek. Without the index, you'd have to examine all of the pages of the book to retrieve those recipes. In effect, you'd be doing a scan. This basic example illustrates the impact a typical database index can make.

A book index is also analogous to a database index in that the pages making up the index are considered to be part of the book. Just like in a database, the pages or blocks making up an index require space on disk to be stored.

Databases store most index data on disk in a tree structure called a b-tree (though there are other types of indexes that use other structures). **Figure 3** illustrates this b-tree structure.

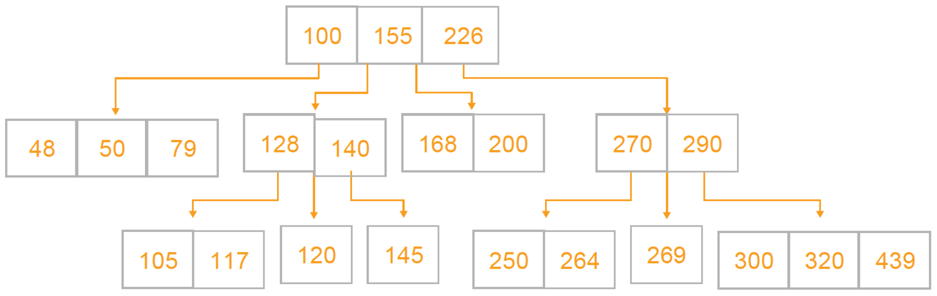


Figure 3: Sample B-Tree Structure

B-trees store point values allowing the database engine to identify rows without having to review the entire table. This tried-and-true approach works well for most database workloads.

When considering how to index a database, you should first try to identify the queries you run most often. If you manage the source code for your application, this should be pretty easy to figure out.

However, if you're supporting third-party applications, you may need to profile the queries running against your database using a tool to capture those queries. Some databases engines, like SQL Server®, Oracle®, PostgreSQL®, and MySQL® running on Microsoft Azure®, automatically capture query runtime and execution data, allowing you to easily identify frequently run queries.

After you've identified your most frequently run (or most expensive in terms of runtime) queries, you can start to look at what columns are indexed. A good guideline is to ensure the frequently used columns in your WHERE clauses (also known as predicates) and columns you're joining to other tables should be indexed. But try to limit the number of columns in your indexes to only the necessary columns. A narrower index is faster to read and consumes less disk space.

Keep in mind, primary and foreign key columns should also be indexed. A primary key represents one or more columns in a table uniquely identifying a row in that table, while a foreign key references another column or columns in another table. If you're working with an ISV system, your primary key columns are probably already indexed. Your foreign keys should also be indexed to reduce lookup time, but this happens far less frequently, so it's important to verify the columns making up the foreign key relationship are indexed.

There are some specialty indexes you should be aware of for different types of workloads. In recent years, columnstore indexes have become very popular in data warehouse systems. These indexes effectively treat each column in a table as its own index, which allows a great deal of compression to take place, helping reduce I/O. Columnstore indexes only require the columns included in a query to be read, which makes them good for wide reporting tables. But they are far less efficient at handling small inserts and updates, so you wouldn't use columnstore indexes for these.

Other special types of indexes are those for JSON and XML data, which can greatly benefit from indexing due to the nested nature of that type of data. Finally, in some database engines, you can create an index with a WHERE clause, which can be useful for tables with lots of archival records. This is called a filtered index and can greatly increase the efficiency of certain workloads.

GET THE RIGHT DATABASE PERFORMANCE TOOLS

This short introduction to database performance has hopefully given you some food for thought about your own operations. If you're like most organizations, there are plenty of opportunities available to make your databases perform more efficiently.

The result is you can get more done, and in less time. Your web applications will work better and faster, keeping your customers happy and coming back. Happy customers keep buying, which keeps you happy, and your business thriving.

Because of this, it's well worth your time to explore ways to monitor, analyze, and improve your database performance.

It can be a daunting task, to be sure. But the benefits can be game-changing for your bottom line, making it a no-brainer to dig deep into this crucial topic.

When you start to dive into database performance, it's important to have tools to help you monitor your databases, to ensure you're getting the most out of them. When that time comes, remember SolarWinds has a comprehensive set of database management tools to do just that.

Get free trials of both their Database Performance Analyzer and Database Performance Monitor products. They offer powerful insight into your database operations, providing actionable information that can help you spot and solve issues before they become problems.

ABOUT SOLARWINDS

SolarWinds (NYSE:SWI) is a leading provider of powerful and affordable IT management software. Our products give organizations worldwide—regardless of type, size, or complexity—the power to monitor and manage their IT services, infrastructures, and applications; whether on-premises, in the cloud, or via hybrid models. We continuously engage with technology professionals—IT service and operations professionals, DevOps professionals, and managed services providers (MSPs)—to understand the challenges they face in maintaining high-performing and highly available IT infrastructures and applications. The insights we gain from them, in places like our THWACK community, allow us to solve well-understood IT management challenges in the ways technology professionals want them solved. Our focus on the user and commitment to excellence in end-to-end hybrid IT management has established SolarWinds as a worldwide leader in solutions for network and IT service management, application performance, and managed services. Learn more today at www.solarwinds.com.

For additional information, please contact SolarWinds at 866.530.8100 or email sales@solarwinds.com.
To locate an international reseller near you, visit http://www.solarwinds.com/partners/reseller_locator.aspx