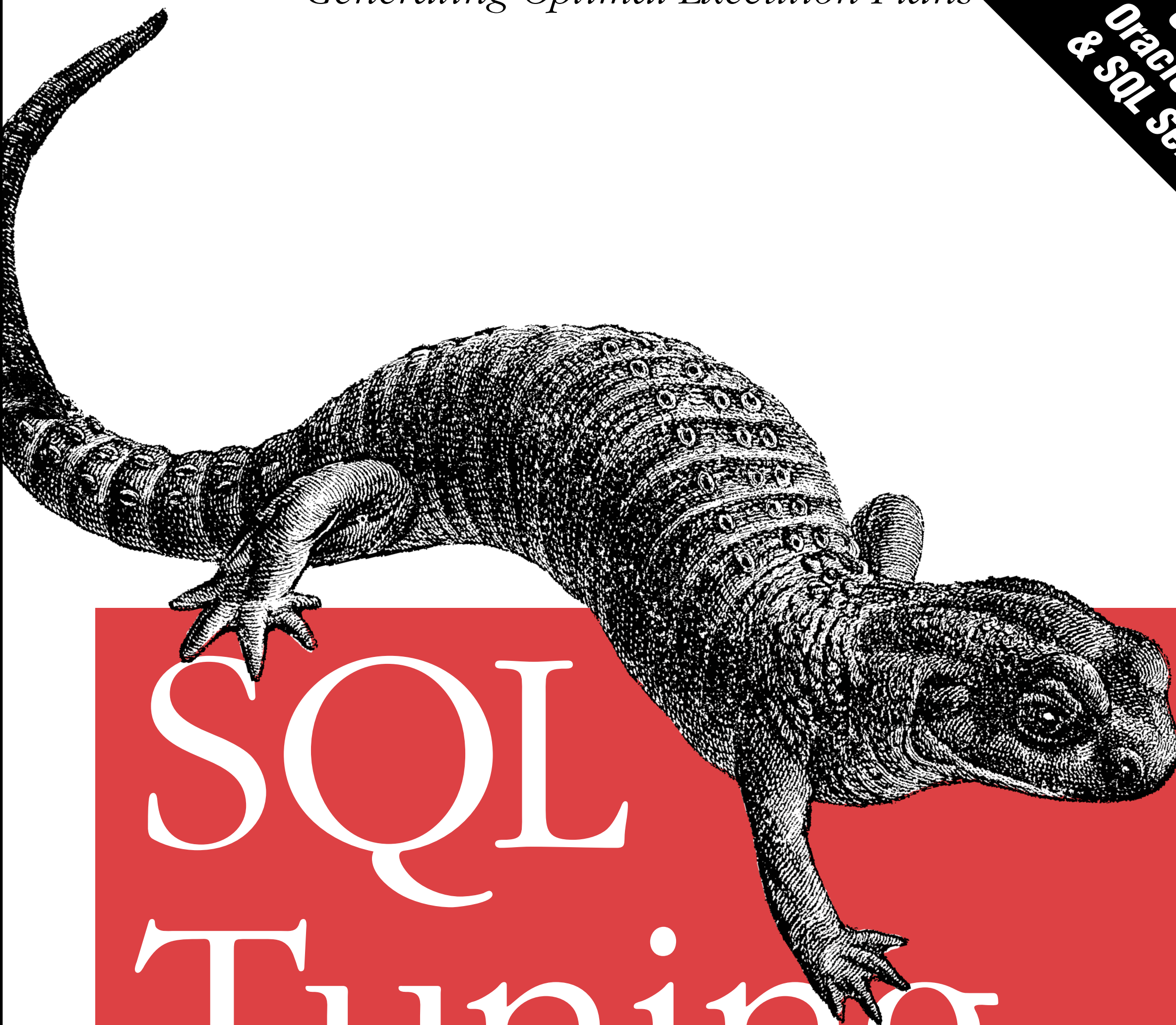
Generating Optimal Execution Plans

Covers Oracle, DB2 & SQL Server



SQL Tuning

O'REILLY®

Dan Tow

CHAPTER 1
Introduction

Well begun is half done.
—Aristotle
Politics, Bk. V, Ch. 4

This book is for readers who already know SQL and have an opportunity to tune SQL or the database where the SQL executes. It includes specific techniques for tuning on Oracle, Microsoft SQL Server, and IBM DB2. However, the main problem of SQL tuning is finding the optimum path to the data. (The path to the data is known as the *execution plan*.) This optimum path is virtually independent of the database vendor, and most of this book covers a vendor-independent solution to that problem.

The least interesting, easiest parts of the SQL tuning problem are vendor-specific techniques for viewing and controlling execution plans. For completeness, this book covers these parts of SQL tuning as well, for Oracle, Microsoft SQL Server, and IBM DB2. Even on other databases, though (and on the original databases, as new releases bring change), the vendor-independent core of this book will still apply. As such, this book is fairly universal and timeless, as computer science texts go. I have used the method at the core of this book for 10 years, on four different vendors' databases, and I expect it to apply for at least another 10 years. You can always use your own vendor's current documentation (usually available online) to review the comparatively simple, release-dependent, vendor-specific techniques for viewing and controlling execution plans.

Why Tune SQL?

Let's begin with a basic question: should someone tune the SQL in an application, and is that someone you? Since you are reading this book, your answer is at least moderately inclined to the positive side. Since it took me several years to appreciate just how positive my own answer to this question should be, though, this chapter lays my own viewpoint on the table as an example.

Let's describe your application, sight-unseen, from an admittedly datacentric point of view: it exists to allow human beings or possibly another application to see, and possibly to enter and manipulate, in a more or less massaged form, data that your organization stores in a relational database. On the output data, it performs manipulations like addition, multiplication, counting, averaging, sorting, and formatting, operations such as those you would expect to see in a business spreadsheet. It does not solve differential equations or do any other operations in which you might perform billions of calculations even on a compact set of inputs. The work the application must do *after* it gets data out of the database, or *before* it puts data into the database, is modest by modern computing standards, because the data volumes handled outside of the database are modest, and the outside-the-database calculation load per datapoint is modest.

Online applications and applications that produce reports for human consumption should produce data volumes fit for human consumption, which are paltry for a computer to handle. *Middleware*, moving data from one system to another without human intervention, can handle higher data volumes, but even middleware usually performs some sort of aggregation function, reducing data volumes to comparatively modest levels.

Even if the vast number of end users leads to high calculation loads outside the database, you can generally throw hardware at the application load (the load outside the database, that is), hanging as many application servers as necessary off the single central database. (This costs money, but I assume that a system to support, say, 50,000 simultaneous end users is supported by a substantial budget.)

On the other hand the database behind a business application often examines millions of rows in the database just to return the few rows that satisfy an application query, and this inefficiency can completely dominate the overall system load and performance. Furthermore, while you might easily add application servers, it is usually much harder to put multiple database servers to work on the same consistent set of business data for the same application, so throughput limits on the database server are much more critical. It is imperative to make your system fit your business volumes, not the other way around

Apart from these theoretical considerations, my own experience in over 13 years of performance and tuning, is that the database—more specifically, the SQL from the application—is the best place to look for performance and throughput improvements.

Improvements to SQL performance tend to be the safest changes you can make to an application, least likely to break the application somewhere else, and they help both performance and throughput, with no hardware cost or minimal cost at worst (in the case of added indexes, which require disk space). I hope that by the end of this book you will also be persuaded that the labor cost of tuning SQL is minimal, given expertise in the method this book describes. The benefit-to-cost ratio is so high that all significant database-based applications should have their high-load SQL tuned.

Performance Versus Throughput

Performance and throughput are related, but not identical. For example, on a well-configured system with (on average) some idle processors (CPUs), adding CPUs might increase throughput capacity but would have little effect on performance, since most processes cannot use more than a single CPU at a time. Faster CPUs help both throughput and performance of a CPU-intensive application, but you likely already have about the fastest CPUs you can find. Getting faster SQL is much like getting faster CPUs, without additional hardware cost.

Performance problems translate to lost productivity, as end users waste time waiting for the system. You can throw money at poor performance by hiring more end users, making up for each end user's reduced productivity, rather than leave the work undone. Over short periods, end users can, unhappily, work through a performance problem by working longer hours.

You have fewer options to solve a throughput problem. You can eliminate the bottleneck (for example, add CPUs) if you are not already at the system limit, or you can tune the application, including, especially, its SQL. If you cannot do either, then the system will process less load than you want. You cannot solve the problem by throwing more end users at it or by expecting those end users to tolerate the rotten performance that results on load-saturated systems. (CPUs do not negotiate: if your business requires more CPU cycles than the CPUs deliver, they cannot be motivated to work harder.) If you cannot tune the system or eliminate nonessential load, this amounts to cutting your business off at the knees to make it fit the system and is the worst possible result, potentially costing a substantial fraction of your revenue.

Who Should Tune SQL?

So, you are persuaded that SQL tuning is a good idea. Should you be the one to do it, on your system? Chances are that you originated at most a small fraction of the SQL on your system, since good-sized teams develop most systems. You might even—like me, in most of my own history—be looking at an application for which you wrote none of the SQL and were not even responsible for the database design. I assumed for years that the developers of an application, who wrote the SQL, would always understand far better than I how to fix it. Since I was responsible for performance, anyway, I thought the best I could do was identify which SQL statements triggered the most load, making them most deserving of the effort to tune them. Then it was (I thought) my job to nag the developers to tune their own highest-load SQL. I was horribly, embarrassingly, wrong.

As it turns out, developers who tune only their own SQL are at a serious disadvantage, especially if they have not learned a good, systematic approach to tuning (which has been lacking in the literature). It is hard to write a real-world application that works, functionally, even without worrying about performance at all. The time

left over for the average developer to tune SQL is low, and the number of self-built examples that that developer will have to practice on to build tuning expertise is also low.

The method this book teaches is the best I know, a method I designed myself to meet my own needs for tuning SQL from dozens of applications other people wrote. However, if you really want to be a first-rate SQL tuner, the method is not enough. You also need practice—practice on other people's SQL, lots of other people's SQL, whole applications of SQL. But how do you cope with the sheer complexity of entire applications, even entire applications you hardly know? Here is where SQL delivered me, at least, a great surprise: you do not need to understand other people's SQL to tune it!

Treat SQL as a *spec*—a clear and unambiguous declaration of which rows of which tables the application requires at some particular point in a program. (SQL is clear because it was designed for casual use by nonprogrammers. It is necessarily unambiguous; otherwise, the database could not interpret it.) You do not need to know why the application needs those rows, or even what those rows represent. Just treat the rows and tables as abstract, even mathematical, entities. All you need to know or figure out is how to reach those rows faster, and you can learn this by just examining the SQL, tables, and indexes involved, with simple queries to the database that are completely independent of the semantic content of the data. You can then transform the SQL or the database (for example, by adding indexes) in simple ways that guarantee, almost at the level of mathematical proof, that the transformed result will return exactly the same rows in the same order, but following a much better, faster path to the data.

How This Book Can Help

There are three basic steps to SQL tuning:

1. Figure out which execution plan (path to reach the data your SQL statement demands) you are getting.
2. Change SQL or the database to get a chosen execution plan.
3. Figure out which execution plan is best.

I deliberately show these steps out of logical order to reflect the state of most material written on the subject. Almost everything written about SQL tuning focuses almost exclusively on the first two steps, especially the second. Coverage of the third step is usually limited to a short discussion about when indexed access is preferred to full table scans. The implied SQL tuning process (lacking a systematic approach to the third step) is to repeat step 2, repeatedly tweaking the SQL, until you stumble on an execution plan that is fast enough, and, if you do not find such a plan, to keep going until you utterly lose patience.

Here is an analogy that works pretty well. Understanding the first step gives you a clear windshield; you know where you are. Understanding the second step gives you a working steering wheel; you can go somewhere else. Understanding the third step gives you a map, with marks for both where you are and where you want to be. If you can imagine being in a strange city without street signs, without a map, in a hurry to find your hotel, and without knowing the name of that hotel, you begin to appreciate the problem with the average SQL tuning education. That sounds bad enough, but without a systematic approach to step 3, the SQL tuning problem is even worse than our lost traveler's dilemma: given enough time, the traveler could explore the entire two-dimensional grid of a city's streets, but a 20-way join has about 20! (20 factorial, or $1 \times 2 \times 3 \times 4 \times \ldots \times 19 \times 20$) possible execution plans, which comes to 2,432,902,008,176,640,000 possibilities to explore. Even your computer cannot complete a trial-and-error search over that kind of search space. For tuning, you need a method that you can handle manually.

With this insight, we can turn the usual process on its head, and lay out a more enlightened process, now expressed in terms of questions:

1. Which execution plan is best, and how can you find it without trial and error?
2. How does the current execution plan differ from the ideal execution plan, if it does?
3. If the difference between the actual and ideal execution plans is enough to matter, how can you change some combination of the SQL and the database to get close enough to the ideal execution plan for the performance that you need?
4. Does the new execution plan deliver the SQL performance you needed and expected?

To be thorough, I cover all of these questions in this book, but by far the most important, and longest, parts of the book are dedicated to answering the first question, finding the best execution plan without trial and error. Furthermore, the range of answers to the first question heavily color my coverage of the third question. For example, since I have never seen a case, and cannot even think of a theoretical case, where the ideal execution plan on Oracle is a sort-merge join, I do not document Oracle's hint for how to force a sort-merge join. (I do explain, though, why you should always prefer a hash join on Oracle anywhere a sort-merge join looks good.)

When we look at the problem of SQL tuning in this new way, we get a surprise benefit: the only really significant part of the problem, deciding which execution plan is best, is virtually independent of our choice of relational database. The best execution plan is still the best execution plan, whether we are executing the statement on Oracle, Microsoft SQL Server, or DB2, so this knowledge is far more useful than anything we learn that is specific to a database vendor. (I even predict that the best execution plan is unlikely to change much in near-future versions of these databases.)

A Bonus

The method this book describes reduces a query to an abstract representation that contains only the information relevant to tuning.

I often substitute *query* for *SQL statement*. Most tuning problems, by far, are queries (SELECT statements, that is). Even for the rest, the problem usually lies in a subquery nested inside the problem update or insert.

This is akin to reducing an elaborate word problem in high-school mathematics to a simple, abstract equation, where the solution of the equation is generally almost automatic once you know the necessary math. The abstract representation of a SQL tuning problem, the query diagram, normally takes the form of an upside-down tree, with some numbers attached, as shown in Figure 1-1.

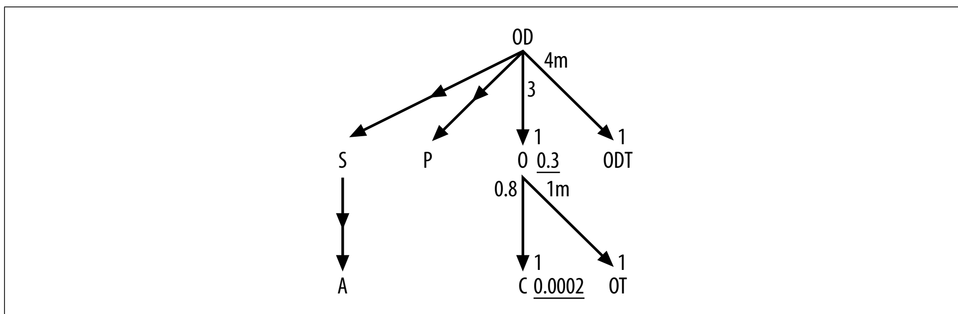


Figure 1-1. An example of a query diagram

As it turns out, SQL is such a flexible language that it is capable of producing queries that do not map to the usual tree form, but it turns out that such queries almost never make sense from a business perspective. This delivers an unplanned-for side benefit: in the course of tuning SQL and producing the abstract query representations that aid you in that process, certain problems with the logic of the queries become obvious, even if you have no prior knowledge of the application. Developers usually catch these problems before you see the SQL, unless the problems lie in the sort of corner cases that they might not test thoroughly, as these problems often do. These corner-case problems can be the worst kind for an application—for example, throwing accounts out of balance long after the application goes live and is assumed to be fine, in subtle ways that are hard to detect and hard to fix.

The worst of these problems will never be found. The business will simply operate based on wrong results, under-billing, over-billing, under-paying, over-paying, or otherwise just doing the wrong thing without anyone tying these problems to a correctable application bug.

Sometimes, fixing a performance problem also requires you to fix a logic problem. Even when the problems are independent (when you can fix performance without fixing the logic flaw), you can perform a major added service by identifying these logic flaws and seeing that they are fixed. This book covers these logic flaws at length, including detailed descriptions of how to find each one and what to do about it. I go so far as to recommend you take any significantly complex, manually written SQL through the SQL diagramming exercise just to find these subtle logic errors, even if you already know that it performs well. Depending on the tool, some products that autogenerate SQL avoid most of these problems.

Outside-the-Box Solutions

Finally, this book discusses outside-the-box solutions: what to do about cases in which you cannot make an individual query fast enough, when treating the query as a spec for what the application requires at that point, just tuning that single query, does not solve the problem. This brings up a class of problems where you really do need to pay some attention to what the application does, when you cannot just treat it as an abstract black box that needs a specified set of rows from some specified tables. Even so, there are some reliable rules of thumb for the kinds of application-level changes that are likely to solve these types of problems. You will likely need to work with developers who know the application details (assuming you do not) to solve these problems, but by understanding the rules you can still offer valuable suggestions without application-specific knowledge.