Troubleshooting Oracle Performance

Copyright © 2008 by Christian Antognini

PART 1

■ ■ ■

Foundations

Chi non fa e' fondamenti prima, gli potrebbe con una grande virtú farli poi, ancora che si faccino con disagio dello architettore e periculo dello edifizio.

He who has not first laid his foundations may be able with great ability to lay them afterwards, but they will be laid with trouble to the architect and danger to the building.[1]

—Niccoló Machiavelli, *Il principe*. **1532.**

---

1. Translated by W. K. Marriott. Available at `http://www.gutenberg.org/files/1232/1232-h/1232-h.htm`.

■ ■ ■

Performance Problems

Too often, tuning begins when an application's development is already finished. This is unfortunate because it implies that performance is not as important as other crucial requirements of the application. Performance is not merely optional, though; it is a key property of an application. Not only does poor performance jeopardize the acceptance of an application, it usually leads to a lower return on investment because of lower productivity. In fact, as shown in several IBM studies from the early 1980s, there is a close relationship between performance and user productivity. The studies showed a one-to-one decrease in user think time and error rates as system transaction rates increased. This was attributed to a user's loss of attention because of longer wait times. In addition, poorly performing applications lead to higher costs for software, hardware, and maintenance. For these reasons, this chapter discusses why it is important to plan performance and how to know when an application is experiencing performance problems. Then the chapter covers how to approach performance problems on a running system.

Do You Need to Plan Performance?

In software engineering, different models are used to manage development projects. Whether the model used is a sequential life cycle like a waterfall model or an iterative life cycle like Rational Unified Process, an application goes through a number of common phases (see Figure 1-1). These phases may occur once (in the waterfall model) or several times (in the iterative model) in development projects.

Figure 1-1. *Essential phases in application development*

If you think carefully about the tasks to carry out for each of these phases, you may notice that performance is inherent to each of them. In spite of this, real development teams quite often forget about performance, at least until performance problems arise. At this point, it may be too late. Therefore, in the following sections, I'll cover what you should not forget, from a performance point of view, the next time you are developing an application.

Requirements Analysis

Simply put, a *requirements analysis* defines the aim of an application and therefore what it is expected to achieve. To do a requirements analysis, it is quite common to interview several stakeholders. This is necessary because it is unlikely that only one person can define all the business and technical requirements. Since requirements come from several sources, they must be carefully analyzed, especially to find out whether they potentially conflict. It is crucial when performing a requirements analysis to not only focus on the functionalities the application has to provide but also to carefully define the utilization of them. For each specific function, it is essential to know how many users[1] are expected to interact with it, how often they are expected to use it, and what the expected response time is for one usage. In other words, you must define the expected performance figures.

RESPONSE TIME

The time interval between the moment a request enters a system or functional unit and the moment it leaves is called *response time.* The response time can be further broken down into the time needed by the system to process the request, which is called *service time*, and the time the request is waiting to be processed, which is called *wait time*.

response time = service time + wait time

If you consider that a request enters a system when a user performs an action, such as clicking a button, and goes out of the system when the user receives an answer in response to the action, you can call that interval *user response time*. In other words, the user response time is the time required to process a request from the user's perspective.

In some situations, like for web applications, it is not common to consider user response time because it is usually not possible to track the requests before they hit the first component of the application (typically a web server). In addition, most of the time it is not possible to guarantee a user response time because the provider of the application is not responsible for the network between the user's application, typically a browser, and the first component of the application. In such situations it is more sensible to measure, and guarantee, the interval between the entry of requests into the first component of the system and when they exit. This elapsed time is called *system response time.*

Table 1-1 shows the performance figures for the actions provided by JPetStore.[2] For each action, the guaranteed system response times for 90 percent and 99.99 percent of the requests entering the system are given. Most of the time, guaranteeing performance for all requests (in other words, 100 percent) is either not possible or too expensive. It is quite common, therefore, to define that a small number of requests may not achieve the requested response time. Since the workload on the system changes during the day, two values are specified for the maximum

---

1. Note that a user is not always a human being. For example, if you are defining requirements for a web service, it is likely that only other applications will use it.
2. JPetStore is a sample application provided, among others, by the Spring Framework. See `http://www.springframework.org` to download it or to simply get additional information.

arrival rate. In this specific case, the highest transaction rate is expected during the day, but in other situations—for example, when batch jobs are scheduled for nights—it could be different.

Table 1-1. *Performance Figures for Typical Actions Provided by a Web Shop*

Action	Max. Response Time (s)		Max. Arrival Rate (trx/min)	
	90%	99.99%	0–7	8–23
Register/change profile	2	5	1	2
Sign in/sign out	0.5	1	5	20
Search products	1	2	60	240
Display product overview	1	2	30	120
Display product details	1.5	3	10	36
Add/update/remove product in/from cart	1	2	4	12
Show cart	1	3	8	32
Submit/confirm order	1	2	2	8
Show orders	2	5	4	16

These performance requirements are not only essential throughout the next phases of application development (as you will see in the following sections), but later you can also use them as the basis for defining service level agreements and for capacity-planning purposes.

SERVICE LEVEL AGREEMENTS

A *service level agreement* (SLA) is a contract defining a clear relationship between a service provider and a service consumer. It describes, among others things, the provided service, its level of availability regarding uptime and downtime, the response time, the level of customer support, and what happens if the provider is not able to fulfill the agreement.

Defining service level agreements with regard to response time makes sense only if it is possible to verify their fulfillment. They require the definition of clear and measurable performance figures and their associated targets. These performance figures are commonly called *key performance indicators*. Ideally a monitoring tool is used to gather, store, and evaluate them. In fact, the idea is not only to flag when a target is not fulfilled but also to keep a log for reporting and capacity-planning purposes. To gather these performance figures, you can use two main techniques. The first takes advantage of the output of instrumentation code (see Chapter 3 for more information). The second one is to use a monitoring tool that checks the application by applying synthetic transactions (see the section "Response-Time Monitoring" later in this chapter).

Analysis and Design

Based on the requirements, the architects are able to design a solution. At the beginning, for the purpose of defining the architecture, it is essential to consider all requirements. In fact, an application that has to handle a high workload must be designed from the beginning to achieve this requirement. This is especially the case if techniques such as parallelization, distributed computing, or reutilization of results are implemented. For example, designing a client/server application aimed at supporting a few users performing a dozen transactions per minute is quite different from designing a distributed application aimed at supporting thousands of users performing hundreds of transactions per second.

Sometimes requirements also impact the architecture by imposing limits on the utilization of a specific resource. For example, the architecture of an application to be used by mobile devices connected to the server through a very slow network must absolutely be conceived to support a long latency and a low throughput. As a general rule, the architects have to foresee not only where the bottlenecks of a solution might be but also whether these bottlenecks might jeopardize the fulfillment of the requirements. If the architects do not possess enough information to perform such a critical estimation *a priori*, one or even several prototypes should be developed. In this respect, without the performance figures gathered in the previous phase, it is difficult to make sensible decisions. By sensible decisions, I mean those leading to an architecture/design that supports the expected workload with a minimal investment—simple solutions for simple problems, elegant solutions for complex problems.

Coding and Unit Testing

A developer should write code that has the following characteristics:

Robustness: The ability to cope with unexpected situations is a characteristic any software should have, based on the quality of the code. To achieve this, it is essential to perform unit testing on a regular basis. This is even more important if you choose an iterative life cycle. In fact, the ability to quickly refactor existing code is essential in such models. For example, when a routine is called with a parameter value that is not part of the list of allowed values, it must nevertheless be able to handle it without crashing. If necessary, a meaningful error message should be generated as well.

Clarity: Long-term readable and documented code is much simpler (and cheaper) to maintain than code that is poorly written. For example, a developer who packs several operations in a single line of cryptic code has chosen the wrong way to demonstrate his intelligence.

Speed: Code should be optimized to run as fast as possible, especially if a high workload is expected. For example, you should avoid unnecessary operations as well as inefficient or unsuitable algorithms.

Shrewd resource utilization: The code should make the best possible use of the available resources. Note that this does not always mean using the least resources. For example, an application using parallelization requires many more resources than one where all operations are serialized, but in some situations parallelization may be the only way to handle high workloads.

Instrumented: The aim of instrumentation is twofold. First, it allows for the easier analysis of both functional and performance problems when they arise—and they will arise to be sure. Second, it is the right place to add strategic code that will provide information about an application's performance. For example, it is usually quite simple to add code that provides information about the time taken to perform a specific operation. This is a simple yet effective way to verify whether the application is capable of fulfilling the necessary performance requirements.

Not only do some of these characteristics conflict with each other, but budgets are usually limited (and sometimes are *very* limited). It seems reasonable then that more often than not it is necessary to prioritize these characteristics and find a good balance between achieving the desired requirements within the available budget.

Integration and Acceptance Testing

The purpose of integration and acceptance testing is to verify functional and performance requirements as well as the stability of an application. It can never be stressed enough that performance tests have the same importance as function tests. For all intents and purposes, an application experiencing poor performance is no worse than an application failing to fulfill its functional requirements. In both situations, the application is useless. Still, it is possible to verify the performance requirements only once they have been clearly defined.

The lack of formal performance requirements leads to two major problems. First, the chances are quite high that no serious and methodical stress tests will be performed during integration and acceptance testing. The application will then go to production without knowing whether it will support the expected workload. Second, it will not always be obvious to determine what is acceptable and what is not in terms of performance. Usually only the extreme cases (in other words, when the performance is very good or very poor) are judged in the same way by different people. And if an agreement is not found, long, bothersome, and unproductive meetings follow.

In practice, designing, implementing, and performing good integration and acceptance testing to validate the performance of an application are not trivial tasks. You have to deal with three major challenges to be successful:

- Stress tests should be designed to generate a representative workload. To do so, two main approaches exist. The first is to get real users to do real work. The second is to use a tool that simulates the users. Both approaches have pros and cons, and their use should be evaluated on a case-by-case basis. In some situations, both can be used to stress different parts of the application or in a complementary way.

- To generate a representative workload, representative test data is needed. Not only should the number of rows and the size of the rows match the expected quantity, but also the data distribution and the content should match real data. For example, if an attribute should contain the name of a city, it is much better to use real city names than to use character strings like *Aaaacccc* or *Abcdefghij*. This is important because in both the application and the database there are certainly many situations where different data could lead to different behavior (for example, with indexes or when a hash function is applied to data).

- The test infrastructure should be as close as possible, and ideally the same, as the production infrastructure. This is especially difficult for both highly distributed systems and systems that cooperate with a large number of other systems.

In a sequential life cycle model, the integration and acceptance testing phase occurs close to the end of the project, which might be a problem if a major flaw in the architecture leading to performance problems is detected too late. To avoid such a problem, stress tests should be performed during the coding and unit testing phases as well. Note that an iterative life cycle model does not have this problem. In fact, in an iterative life cycle model, a stress test should be performed for every iteration.

Do You Have Performance Problems?

There is probably a good chance that sooner or later the performance of an application will be questioned. If, as described in the previous sections, you have carefully defined the performance requirements, it should be quite simple to determine whether the application in question is in fact experiencing performance problems. If you have not carefully defined them, the response will largely depend on who answers the question.

Interestingly enough, in practice the most common scenarios leading to questions regarding the performance of an application fall into very few categories. They are short-listed here:

- Users are unsatisfied with the current performance of the application.

- A system-monitoring tool alerts you that a component of the infrastructure is experiencing timeouts or an unusual load.

- A response-time monitoring tool informs you that a service level agreement is not being fulfilled.

The difference between the second point and the third point is particularly important. For this reason, in the next two sections I will briefly describe these monitoring solutions. After that, I will present some situations where tuning appears to be necessary but in fact is not necessary at all.

System Monitoring

System-monitoring tools perform health checks based on general system statistics. Their purpose is to recognize irregular load patterns that pop up as well as failures. Even though these tools can monitor the whole infrastructure at once, it is important to emphasize that they monitor only individual components (for example, hosts, application servers, databases, or storage subsystems) without considering the interplay between them. As a result, it is difficult, and for complex infrastructures virtually impossible, to determine the impact on the system response time when a single component of the infrastructure supporting it experiences an anomaly. An example of this is the high usage of a particular resource. In other words, an alert coming from a system-monitoring tool is just a warning that something could be wrong with the application or the infrastructure, but the users may not experience any performance problems at all (called a *false positive*). In contrast, there may be situations where users are experiencing performance problems but the system-monitoring tool does not recognize them (called a *false negative*). The most common, and simple, cases of false positive and false negative are seen while monitoring the CPU load of SMP systems with a lot of CPUs. Let's say you have a 16-CPU system (or a system

with four quad-core CPUs). Whenever you see a utilization of about 75 percent, you may think that it is too high; the system is CPU-bounded. However, this load could be very healthy if the number of running tasks is much greater than the number of CPUs. This is a false positive. Conversely, whenever you see a utilization of about 8 percent of the CPU, you may think that everything is fine. But if the system is running a single task that is not parallelized, it is possible that the bottleneck for this task is the CPU. In fact, 1/16th of 100 percent is only 6.25 percent, and therefore, a single task cannot burn more than 6.25 percent of the available CPU. This is a false negative.

Response-Time Monitoring

Response-time monitoring tools (also known as *application-monitoring tools*) perform health checks based on synthetic transactions that are processed by *robots*. The tools measure the time taken by an application to process key transactions, and if the time exceeds an expected threshold value, they raise an alert. In other words, they exploit the infrastructure as users do, and they complain about poor performance as users do. Since they probe the application from a user perspective, they are able to not only check single components but, and more important, check the whole application's infrastructure as well. For this reason, they are devoted to monitoring service level agreements.

Compulsive Tuning Disorder

Once upon a time, most database administrators suffered from a disease called *compulsive tuning disorder*.[3] The signs of this illness were the excessive checking of many performance-related statistics, most of them ratio-based, and the inability to focus on what was really important. They simply thought that by applying some "simple" rules, it was possible to tune their databases. History teaches us that results are not always as good as expected. Why was this the case? Well, all the rules used to check whether a given ratio (or value) was acceptable were defined independently of the user experience. In other words, false negatives or positives were the rule and not the exception. Even worse, an enormous amount of time was spent on these tasks.

For example, from time to time a database administrator will ask me a question like "On one of our databases I noticed that we have a large amount of waits on latch X. What can I do to reduce or, even better, get rid of such waits?" My typical answer is "Do your users complain because they are waiting on this specific latch? Of course not. So, don't worry about it. Instead, ask them what problems they are facing with the application. Then, by analyzing those problems, you will find out whether the waits on latch X are related to them or not." I'll elaborate on this in the next section.

Even though I have never worked as a database administrator, I must admit I suffered from compulsive tuning disorder as well. Today, I have, like most other people, gotten over this disease. Unfortunately, as with any bad illness, it takes a very long time to completely vanish. Some people are simply not aware of being infected. Others are aware, but after many years of addiction, it is always difficult to recognize such a big mistake and break the habit.

---

3. This wonderful term was first coined by Gaya Krishna Vaidyanatha. You can find a discussion about it in the book *Oracle Insights: Tales of the Oak Table* (Apress, 2004).

How Do You Approach Performance Problems?

Simply put, the aim of an application is to provide a benefit to the business using it. Consequently, the reason for optimizing the performance of an application is to maximize that benefit. This does not mean maximizing the performance, but rather finding the best balance between costs and performance. In fact, the effort involved in an optimization task should always be compensated by the benefit you can expect from it. This means that from a business perspective, performance optimization may not always make sense.

Business Perspective vs. System Perspective

You optimize the performance of an application to provide a benefit to a business, so when approaching performance problems, you have to understand the business problems and requirements before diving into the details of the application. Figure 1-2 illustrates the typical difference between a person with a *business perspective* (that is, a user) and a person with a *system perspective* (that is, an engineer).
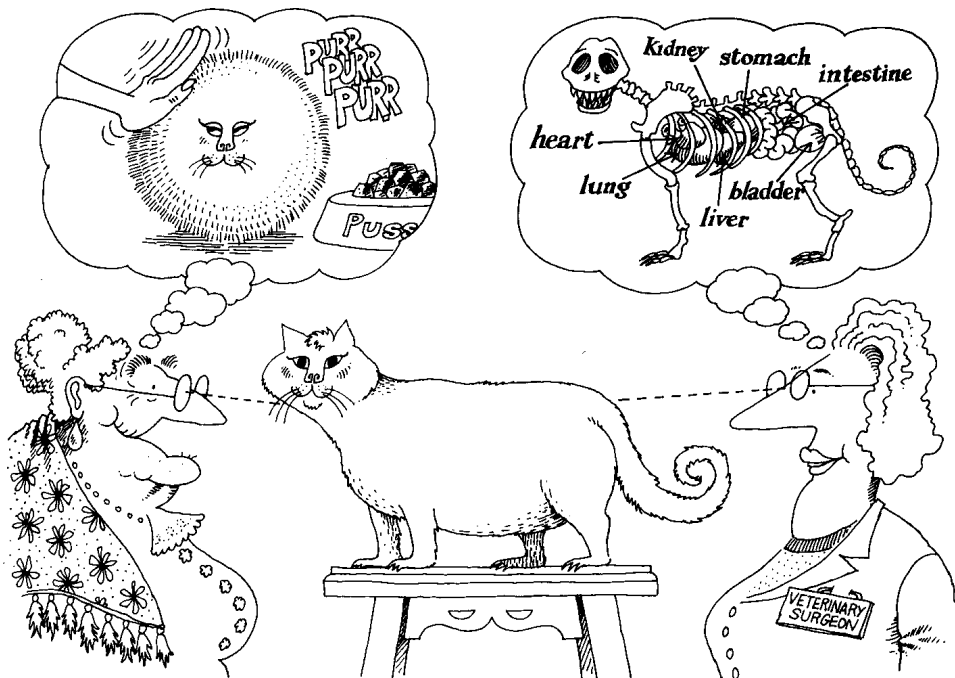

Figure 1-2. *Different observers may have completely different perspectives.*[4]

---

4. Booch, Grady, *Object-Oriented Analysis and Design with Applications,* page 42 (Addison Wesley Longman, Inc., 1994). Reproduced with the permission of Grady Booch. All rights reserved.

It is important to recognize that there is a cause-effect relationship between these two perspectives. Although the effects must be recognized from the business perspective, the causes must be identified from the system perspective. So if you do not want to troubleshoot nonexistent or irrelevant problems (compulsive tuning disorder), it is essential to understand what the problems are from a business perspective—even if more subtle work is required.

Cataloging the Problems

The first steps to take when dealing with performance problems are to identify them from a business perspective and to set a priority and a target for each of them, as illustrated in Figure 1-3.
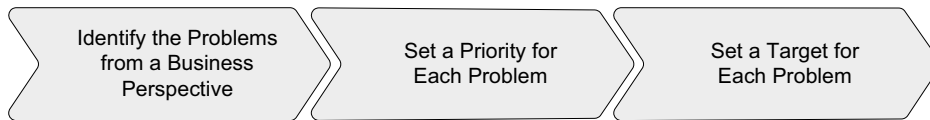
Figure 1-3. *Tasks to carry out while cataloging performance problems*

Business problems cannot be found by looking at system statistics. They have to be identified from a business perspective. If a monitoring of service level agreements is in place, the performance problems are obviously identified by looking at the operations not fulfilling expectations. Otherwise, there is no other possibility but to speak with the users or those who are responsible for the application. Such discussions can lead to a list of operations, such as registering a new user, running a report, or loading a bunch of data that is considered slow.

Once you know the problematic operations, it is time to give them a priority. For that, ask questions like "If we can work on only five problems, which should be handled?" Of course, the idea is to solve them all, but sometimes the time or the budget is limited. In addition, it is not possible to leave out cases where the measures needed to fix different problems conflict with each other. It is important to stress that to set priorities, the current performance could be irrelevant. For example, if you are dealing with a set of reports, it is not always the slowest one that has the highest priority. Possibly the fastest one is also the one that is executed more frequently. It might therefore have the highest priority and should be optimized first. Once more, business requirements are driving you.

For each operation, you should set a quantifiable target for the optimization, such as "When the Create User button is clicked, the processing lasts at most two seconds." If the performance requirements or even service level agreements are available, it is possible that the targets are already known. Otherwise, once again, you must consider the business requirements to determine the targets. Note that without targets you do not know when it is time to stop investigating for a better solution. In other words, the optimization could be endless. Remember, the effort should always be balanced by the benefit.

Working the Problems

Troubleshooting a whole system is much more complex than troubleshooting single components. Therefore, whenever possible, you should work one problem at a time. Simply take the list of problems and go through them according to their priority level.

For each problem, the three questions shown in Figure 1-4 must be answered:

Where is time spent? First, you have to identify where time goes. For example, if a specific operation takes ten seconds, you have to find out which module or component most of these ten seconds are used up in.

How is time spent? Once you know where the time goes, you have to find out how that time is spent. For example, you may find out that the application spends 4.2 seconds on CPU, 0.4 seconds doing I/O operations, and 5.1 seconds waiting for dequeuing a message coming from another component.

How can time spent be reduced? Finally, it is time to find out how the operation can be made faster. To do so, it is essential to focus on the most time-consuming part of the processing. For example, if I/O operations take 4 percent of the overall processing time, it makes no sense to start tuning them, even if they are very slow.

Figure 1-4. *To troubleshoot a performance problem, you need to answer these three questions.*

It is important to note that thanks to beneficial side effects, sometimes measures implemented to fix a particular problem will also fix another one. Of course, the opposite can happen as well. Measures taken may introduce new problems. It is essential therefore to carefully consider all the possible side effects that a specific fix may have. Clearly, all changes have to be carefully tested before implementing them in production.

On to Chapter 2

In this chapter, we looked at some key issues of dealing with performance problems: why it is essential to approach performance problems at the right moment and in a methodological way, why understanding business needs and problems is absolutely important, and why it is necessary to agree on what *good performance* means.

Before describing how to answer the three questions in Figure 1-4, it is essential that I introduce some key concepts that I'll be using in the rest of the book. For that purpose, Chapter 2 will describe the processing performed by the database engine to execute SQL statements. In addition, I'll define several frequently used terms.