APPDYNAMICS

Top 10 most common Java performance problems

Top 10 most common Java performance problems

Table of contents

Steven Haines is a technical architect at Kit Digital, working onsite at Disney in both an advisory role to the performance of Disney’s largest software endeavor to date as well as managing the development, quality assurance, and business analysis teams responsible for the ticketing and yet undisclosed systems. He has written three books: Java 2 From Scratch, Java 2 Primer Plus, and Pro Java EE Performance Management and Optimization, and has spent the better part of the past 14 years in the Application Performance Management (APM) space at both Quest Software and AppDynamics. Additionally he taught Java at both Learning Tree University and the University of California Irvine. You can find him teaching technical subjects on his website: www.geekcap.com. When not engrossed in performance tuning and enterprise architecture he spends his time with his two children: Michael (12) and Rebecca (3), and his wife Linda, frequenting amusement parks in Orlando, FL.

Introduction

My career in performance began, as you might guess, with a catastrophe. I was working as an architect at a marketing company that runs surveys, and one of our promotions took off. It was featured on the AOL homepage, and thousands of people began visiting our website. What started as a great success quickly turned into a crisis as our application servers began to fall over under the unprecedented load.

Our application environment was actually pretty heavy-duty for the time. We had a Cisco load balancer in front of four WebLogic instances running on Solaris with an Oracle cluster behind them. We were grossly underprepared, however, for the load we were about to receive. As soon as the promotion appeared on the AOL homepage I watched our session counts start climbing from 10 to 50, 500, and 5,000, at which point things started heading downhill quickly. I was restarting the WebLogic instances as fast as I could, like a terrible game of Whack-a-Mole. Eventually, however, it got out of our hands – we had to ask AOL to remove our survey from their homepage, because we simply couldn't handle the traffic.

When this happened I had virtually no experience with performance analysis, but I quickly realized how important performance was to everyone in the business. Over the next 13 years I learned all I could about Java performance and architecture so that I could help my company and clients to get their apps up to speed. Over those 13 years I saw my fair share of performance-related issues, and I noticed a disturbing trend: Most performance issues in Java can be attributed to a handful of root causes. Sure, occasionally I saw some bizarre corner cases that came out of the blue and wreaked havoc in an app, but for the most part performance issues in Java are all pretty cookie cutter. In this eBook, I'll talk about some of the most common problems I've encountered during my time as a performance analyst, along with how to recognize and address these issues to minimize their impact and prevent them from occurring in your application. I've sorted the most common issues into three main categories:

- Database Problems: Most applications of scale will eventually be backed by some form of relational or non-relational database, so for the purposes of this eBook I focus on three common relational database problems: persistence configuration (lazy vs. eager loading), caching, and database connection pool configuration.

- Memory Problems: Java memory management is challenging and can lead to all kinds of performance issues. I focus on what I have observed to be the two most common memory issues: garbage collection configuration and memory leaks.

- Concurrency Problems: As the complexity of applications increases we find ourselves writing code that performs more actions concurrently. In this section I focus on three common concurrency issues: thread deadlocks, thread gridlocks, and thread pool configuration issues.

-Steven Haines

Database

Database

The backbone of any modern web application is its data. Back in 1995, when businesses first began building web applications to house their marketing content, the database wasn't such a necessary feature of most web apps – the content was static and there was virtually no user interaction. Today, however, applications are much more complex and perform many more functions than before. As a result, the database is critical to the functionality and performance of the application.

It should come as no surprise that the database is, therefore, the biggest source of performance issues for Java applications. Problems can occur in many places: your application code may access the database inefficiently, or the database connection pool may be improperly sized, or the database itself may be missing indices or otherwise in need of tuning. In this section we'll focus on three of the most common performance issues on the application side of the equation (it's probably best to leave database tuning for another eBook):

- Excessive database queries (sometimes called the N+1 problem)
- Executing database queries that should be served from a cache
- An improperly configured database connection pool

1. Death by 1,000 cuts: The database N+1 problem

Back in the J2EE days when we were building Entity Beans, and specifically Bean Managed Persistence (BMP) Entity Beans, there was a problem that was referred to as the "N+1" problem. Let's say I wanted to retrieve the last 100 entries from my Order table. To do this, I'd have to first execute a query to find the primary keys of each item, such as:

```
SELECT id FROM Order WHERE ...
```

And then I'd execute one query for each record:

```
SELECT * FROM Order WHERE id = ?
```

In other words, it would take me 101 queries to retrieve 100 records (N+1). We'd call this problem "death by 1,000 cuts" because even though each individual query was fast, I was executing so many of them that the cumulative response time of the transaction was huge, and the database was subjected to enormous overhead.

Persistence technologies have improved since then, but the "N+1" problem hasn't completely disappeared, and neither has "death by 1,000 cuts." It's important to understand the implications of configuration options on both your database and your persistence engine.

Problem

A common problem with Java applications that access a database is that they sometimes access the database too often, resulting in long response times and unacceptable overhead on the database. Most of the time this is not deliberate. The above example describes the state of the database access a few years ago, but if you're not careful, this problem can reemerge in modern Hibernate and JPA code.

Hibernate and other JPA implementations provide fine-grained tuning of database access. The most common option for database tuning is the choice between eager and lazy fetching. Consider querying for a purchase order that contains 10 line items. If your application is configured to use lazy fetching, but the business requirement calls for the application to show all 10 line items, then the application will be required to execute an additional 10 database calls to load that purchase order. If the application is configured to use eager fetching, however, depending on the persistence technology, you may have one or two additional queries, but not 10.

Database cont'd

When an application requests an object from the database, that object may reference other objects. For example, a PurchaseOrder object may reference multiple LineItem objects. Eager fetching means that when the PurchaseOrder object is requested, all referenced LineItem objects will be retrieved from the database at that time. Lazy fetching means that when the PurchaseOrder object is requested, the referenced LineItems will not be retrieved from the database, but when a LineItem is accessed, it will be loaded from the database. Eager fetching reduces the number of calls that are made to the database, but those calls are more complex and slower to execute and they load more data into memory. Lazy fetching increases the number of calls that are made to the database, but each individual call is simple and fast and it reduces the memory requirement to only those objects your application actually uses.

So which strategy should you use? The answer, of course, depends on your use case. If you have no intention of looking at the line items 99% of the time then lazy fetching is the correct strategy because querying for a single line item is much faster than all 10. But if 90% of the time you're going to look at the line then eager fetching is your friend, because a couple slower queries are better than a hundred fast ones. The point is that you need to understand how your application will be used before you configure your persistence engine.

Symptoms

The primary symptom of the N+1 problem is an increased load on, and therefore a slower response time from, the database. The problem will be hard to detect under low user load because the database will have plenty of processing power, but as application load increases it will become more and more problematic. If the application load and database load increase at the same rate then you are probably making good use of your database, but if an increase in load on the application results in a disproportionate increase in load on the database then you might have an N+1 problem.

Impact

6

I only rate the impact of this problem as a 6 because, even though it's fairly common, you have an easy mitigation strategy: simply increase the capacity of your database. It it's not a long-term solution, but it'll do the job. The good news is that once you've diagnosed this problem, fixing it is usually pretty straightforward. Just be careful when making changes to your persistence engine configurations, because different use cases will have different consequences for application performance.

Troubleshooting

After observing the symptoms of this problem, troubleshooting the root cause can be challenging. In order to be able to effectively troubleshoot an N+1 problem you need the following information:

- Counters for the number of database calls
- Counters for the number of executed business transactions
- A correlation between a business transaction and the number of database calls it's making
- An understanding of the business rules governing the business transactions that are exhibiting the problem (you don't want to fix a performance problem just to learn that you just broke a business rule)

The business transaction and database counters can be used to confirm that you have this problem, but the most important piece of information is the correlation between business transactions and the database calls they make. One warning to be aware of is that the nature of this problem is in the persistence logic, so you may very well have multiple business transactions exhibiting this problem. If you can correlate business transactions to persistence logic then you'll be in a very good place to properly remediate the problem.

Avoiding this problem

The best way to avoid this problem is to understand both your business domain and the persistence technology you're using. Making sure you're solving the correct business problem is up to you, but here's some general advice about the technology side of things:

- Do not accept the defaults without understanding what they are. In the case of eager versus lazy loading, most persistence engines default to "lazy," which may or may not be appropriate for your application. If it isn't, you may be inflicting unnecessary load of your database.

- Understand tuning options in your persistence tier. Eager vs. lazy loading is one example, but there are other options that can cause issues. To truly be effective, you need to pair up the configuration options of your chosen persistence technology with the business problem you're trying to solve.

2. Credit and debit only, no cache: The importance of caching

Years ago, Marc Fleury, the creator of JBoss, wrote a paper called "Why I Love EJBs." In it he argues that it's faster to read data from an entity bean in memory than it is to make a database call across a network. While I'm not as in love with EJBs as Marc, I can't deny that database calls can be very expensive from a performance standpoint. This is why, in recent years, many organizations have turned to caching to optimize the performance of their applications – it's much faster to read data from an in-memory cache than to make a database call across a network.

Problems with caching

1. No cache. It doesn't take a degree in rocket science to understand that it's faster to serve content from memory than to make a network trip to a database that has to execute a query to retrieve your data. Unless you have specific reasons not to cache, you should be caching.

2. The cache is not configured properly. There are various levels and various implementations of caching, from a level 2 cache that sits between your persistence engine and your database to a stand-alone distributed cache that holds arbitrary business objects. Your persistence technology, such as Hibernate, should have support for a level 2 cache that behaves as follows: when a request for an object is made, first check the cache to see if the object is already in memory; if it is and it hasn't expired, then return that cached object, otherwise make the database call, save the object to the cache, and return the object to the caller. In this capacity, frequently used objects will be resolved without requiring interaction with a database.

Caches are not the be-all end-all solution, but once you have decided to use a cache, there are a few things you need to consider:

- Caches are a fixed size
- Distributed caching is a non-trivial problem

Caches hold **stateful** objects, unlike pools, which hold **stateless** objects. For example, imagine a pool as the registers at a supermarket. When you're ready to check out, you go to whichever register is free – it doesn't matter which one you get. Caches, on the other hand, are like children at daycare. When you go to the daycare to pick up your child, you don't just pick up whichever child is available first – you're only interested in picking up your own child. Pools contain stateless objects, meaning it doesn't matter which connection you get – all connections are equal, but caches contain stateful objects because you go to a cache looking for a specific piece of data.

> *Stateful* objects represent specific object instances, such as a specific PurchaseOrder or a specific child. Stateless objects represent general objects, such as a Phillips head screwdriver or a supermarket checker. When a stateful object is accessed it is important to retrieve a specific object, but when a stateless object is accessed, any object of that type will do.

Because caches are stateful, you must configure them to a finite size so as not to exhaust memory. When the cache is full, then the cache must respond based on its configuration. For example, it might remove the least recently used object from the cache to make room for the new object. This means that sometimes the requested object may no longer be in the cache, resulting in a "miss." A miss typically results in a database call to find the requested object. The higher your miss ratio, therefore, the less you're taking advantage of the performance benefits of the cache. It's important to optimize your cache settings carefully, so that you maintain a good "hit ratio" without exhausting all the memory in your JVM.

3. Distributed caching. If you have multiple servers in a tier all writing to their own caches, how do they stay in sync? If you do not configure the caches to be distributed, then they won't. Depending on which server you hit, your results may vary (which is usually a bad thing). Most modern caches support a distributed paradigm so that when a cache is updated it will propagate its changes to other members in the cache. But depending on the number of cached nodes and the **consistency** of data you require, this can be expensive. Consistency refers to the integrity of your data at a point in time: if one cache node has one value for an object and another node has a different value then the two cache nodes are said to be inconsistent. On the loose end of the spectrum, caches can be "eventually consistent," meaning that your application can tolerate short periods of time when the caches on different nodes do not have the same values. For example, if you post a new status on Facebook you'll see it immediately, but your friends won't see it for a couple minutes. On the other end of the spectrum, you might require all cache nodes to have the same value before that update is considered committed. The performance challenge is to balance your distributed caching behavior with your business rules: try to opt for the loosest distribution strategy that satisfies your business requirements.

> Cache consistency, sometimes referred to as cache coherence, refers to the validity of data across your entire cache. A cache is consistent if every instance of the same object in the cache has the same value. In recent times, large-scale applications have adopted eventual consistency, which means that there will be periods of time when different instances of the same object will have different values, but eventually they will have the same value.

Database cont'd

Symptoms

The main symptom of an application that is not using a cache properly is increased database load and slow response times as a result of that load. Unlike the "N+1" performance problem, the relative database load increases in direct proportion to your application load. If you're using caching correctly, database load should not increase in proportion to application load, because the majority of requests should be hitting the cache. The negative impact of this problem is that as your load increases, the database machine is subjected to more load, which increases its CPU overhead and potentially disk I/O rate, which degrades the overall performance of all business transactions that interact with the database.

Impact

7

I only rate a missing cache as a 7 because adding a cache is really a performance enhancement, not a necessity. If you see increased load in your database and degraded performance, one of the biggest enhancements you can make for your application is adding a cache. It would be best to plan for a cache from the beginning, but caches can be added after the fact with minimal code rewrites. Depending on where you insert your cache and your cache provider's requirements, the code impact may vary, but it's rarely significant.

Troubleshooting

Identifying the need for a cache is accomplished by examining the performance of your database, its resource usage, and the amount of load that your application is sending to the database. If you observe problems with your database's resource utilization then you should examine your business requirements and determine whether or not a cache would be a good option.

Once you have determined that you need a cache, sizing the cache appropriately is the next issue. A cache adds the most value if the cache can service the majority of queries made to the cache, meaning that it contains the majority of the most frequently accessed objects. If the cache is sized too small then a significant number of queries will require a call to the backend data store because the cache doesn't contain the value. If the cache is sized too large then it could consume an excessive amount of memory. Caches frequently publish metrics, such as through JMX, about their performance. Two common metrics are the cache hit count or hit ratio and the cache miss count or miss ratio. A cache hit means that the cache serviced the request and a cache miss means that the cache did not service the request. If you observe a high miss count then the cache is sized too small.

Avoiding this problem

Plan, plan, plan! Whenever I develop an application, no matter how small, I always err on the side of performance and scalability. This is not to say that you should go to extraordinary lengths to overengineer your application. It just means that before you begin, you should ask yourself the question: if I face performance issues, is this an object that could be cached? If so, go ahead and build the object in such a way that it can be easily added to a cache later, by doing things like making the object **serializable**.

If you're building an application of substance then I would recommend implementing caching anywhere it seems appropriate. You'll avoid many problems further down the line for a relatively small investment.

Java defines the notion of serialization as follows: an object is serializable if it implements the java.io.Serializable interface and it only contains primitive types, Strings, and other serializable objects. Practically, serializable objects can be converted into a form that can be transported to and from other servers or even to disk. Most caching solutions leverage Java's support for serialization to send objects from one machine to another.

3. Does anyone have a connection I can borrow? Database connection pools

In the last chapter we compared pools to the registers or checkers in a supermarket. A set number of checkers are open at any given time, and the shopper doesn't care which checker they get, then they choose the first available checker so they can get out of the store as soon as possible.

To take this analogy a step further, imagine now that only two registers are open during a busy time at the supermarket. What happens? If you've never experienced this (lucky you) then you can probably imagine that there would be a long line of angry customers. This is analogous to what happens when your database connection pool is too small. The number of connections to your database controls how many concurrent queries can be executed against it. If there are too few connections in the pool then you'll have a bottleneck in your application, increasing response times and angering end users (who, like Inigo Montoya, hate waiting).

Problem

Database connections are pooled for several reasons:

- Database connections are relatively expensive to create, so rather than create them on the fly we opt to create them beforehand and use them whenever we need to access the database.
- The database is a shared resource so it makes sense to create a pool of connections and share them across all business transactions.
- The database connection pool limits the amount of load that you can send to your database.

The first two points make sense because we want to pre-create expensive resources and share them across our application. The last point, however, might seem counter-intuitive. We pool connections to reduce load on the database because otherwise we might saturate the database with too much load and bring it to a screeching halt. The point is that not only do you want to pool your connections, but you also need to configure the size of the pool correctly.

If you do not have enough connections, then business transactions will be forced to wait for a connection to become available before they can continue processing. If you have too many connections, however, then you might be sending too much load to the database and then all business transactions across all application servers will suffer from slow database performance. The trick is finding the middle ground.

Symptoms

The main symptoms of a database connection pool that is sized too small are increased response time across multiple business transactions, with the majority of those business transactions waiting on a Datasource.getConnection() call, in conjunction with low resource utilization on the database machine. At first glance this will look like a database problem, but the low resource utilization reveals that the database is, in fact, under-utilized, which means the bottleneck is occurring in the application.

The symptoms of a database connection pool that is sized too large are increased response time across multiple business transactions, with the majority of those business transactions waiting on the response from queries, and high resource utilization in the database machine.

So while the external symptoms between these two conditions are the same, the internal symptoms are the opposite. In order to correctly identify the problem, you need to find out where your application is waiting on the database (for a connection to the database or on the execution of a query) and what the health of the database is.

Impact

8

The impact of a misconfigured database connection pool rates an 8 on my scale because the performance impact will be observable by your users. The fix is simple but will require time and effort: use load testing and a performance analysis tool to find the optimal value for the size of your database connection pool, and then make the configuration change.

Database cont'd

Troubleshooting

Identifying that you truly have a database connection pool configuration problem requires insight into what your application is doing:

- If your application is waiting on calls like Datasource.getConnection() and your database is underutilized then your connection pool is too small
- If your application is waiting on database query executions, such as PreparedStatement.execute() and the database is over-utilized then your connection pool is too large (or your database and your queries need to be tuned!)

Avoiding this problem

Database connection pool problems are really a combination of connection pool size tuning, SQL query tuning, and database tuning. If your queries are optimized and your database is properly tuned then your database can support more load than if your queries are sloppily written and the database is not tuned. Therefore I recommend the following approach:

1. Tune your SQL queries, either manually using your favorite SQL tuning book as a guide or automatically using a tool like SQL Optimizer. Ensure that your SQL is top-notch (and this includes your HQL and EJBQL as well).
2. Estimate the relative balance between the various queries your application will be executing (determined by your estimation of business transaction balance and your understanding of the queries executed by each business transaction).
3. Execute a load test against our database and tune your database to optimally support these queries.
4. Load test your application in a production-like environment (same number and same class of machine if possible). Run multiple iterations with different database connection pool settings and choose the best fit for your application. You want to ensure that you do not saturate the database, so if you can find the number of connections just below that saturation point then you have your golden value.

Memory

Memory

Today, most modern programming languages run in managed environments. When Java first entered the scene back in 1996, however, this wasn't the case. Other than a simplified syntax, managed memory was one of the biggest improvements that Java introduced. With Java's memory management, traditional C/C++ memory leaks could be easily avoided, making it easier than ever to build stable and secure applications. But managed memory was a mixed blessing for Java developers, who now had to deal with an entirely different beast – Java's garbage collector.
In the last 15 years, Java developers have established best practices for tuning garbage collection and properly managing objects in Java's managed memory model. However, not every developer is as intimately acquainted with object lifecycles and garbage collection behaviors as those that built applications without the aid of automatic memory management. As a result, memory-related problems still plague modern Java applications. In this section we'll take a look at the two most common memory issues in Java apps:

- Java Stop-the-World (STW) garbage collection
- Java Memory Leaks

Managed memory refers to the management of memory by the environment, or JVM in this case, and not the programmer. In other words, you, as the programmer, cannot directly manage memory by deleting objects that you no longer intend to use, but rather the JVM will delete memory on your behalf.

4. "I'll stop the world … and melt with you": STW garbage collections

It's nice when Modern English sings it, but "stop-the-world" can have much graver implications when it comes to your Java application. Stop-the-world garbage collection refers to a major garbage collection that freezes all running threads in the JVM in order to reclaim memory. It's actually a normal process, and it isn't a problem in itself unless it 1) takes too long to complete, or 2) occurs far too often.

Problem

To understand when and why garbage collection pauses the JVM, first we'll need to understand a little bit about how garbage collection works in Java. Different JVM implementations and different JVM garbage collection strategies manage heaps differently. For the purposes of this discussion I will focus on the Sun JVM. The Sun JVM is a generational JVM that divides the heap into two primary generations: the young generation and the old generation. Figure 4.1 shows the arrangement of the Sun heap.

Figure 4.1

As shown in Figure 4.1, the young generation is divided into three spaces: the Eden space and two Survivor Spaces (From space and To space), while the Tenured Space consumes the old generation. The object lifecycle can be defined as follows:

1. Objects are created in Eden
2. When Eden is full a minor mark-sweep garbage collection is performed
3. Objects that survive the mark-sweep garbage collection are copied to the To survivor space
4. If there are existing objects in the From survivor space then those are copied to the To survivor space
5. Once the To survivor space is full, all live objects left in Eden and the From survivor space are tenured, or copied to the Tenured Space

This lifecycle is summarized in Figure 4.2

Memory cont'd

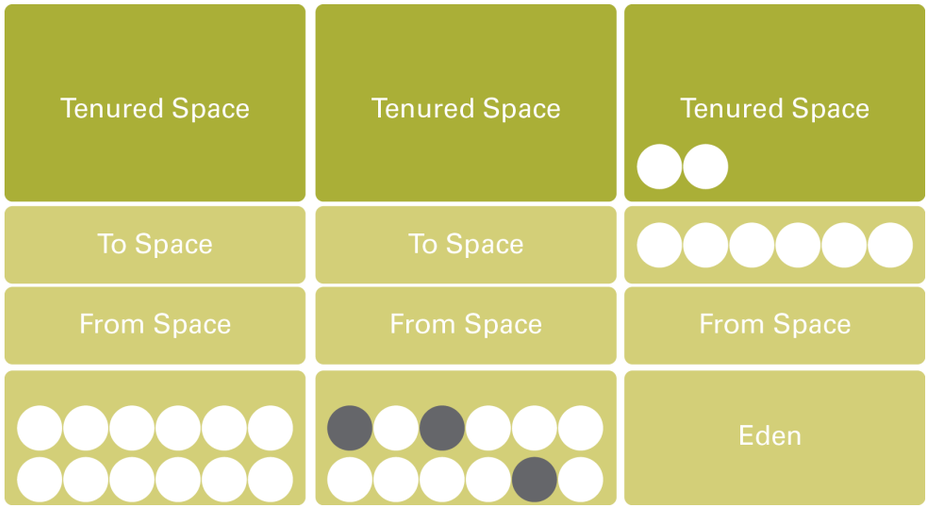

Figure 4.2

The next lifecycle that you need to be aware of is how objects are claimed during a major collection. A Major garbage collection, also known as a stop-the-world (STW) collection, does just that: it freezes all threads in the JVM, performs a mark-sweep collection across the entire heap, and then performs a compaction. The result is that the entire young generation is free and all live objects are compacted into the old generation. This is a very effective garbage collection, but it is both very slow (on the order of 10x minor collections) and impactful to your users because no threads are able to run during this collection. So in essence it "stops the JVM world" for however long it takes to run. And the amount of time will be directly proportional to the size of your heap. In a modest 2-3 gigabyte heap this might be 3-5 seconds, but if you are running a 30-gigabyte head this could be more on the order of 30 seconds.

There are strategies to minimize the impact of stop-the-world collections, such as the Concurrent Mark Sweep (CMS) garbage collection strategy, which maintains an additional thread that is constantly marking and sweeping objects, and the new G1 garbage collection strategy, which attempts to collect popular sections of the heap, but neither of these completely eliminate stop-the-world compactions. Azul Systems claims to have built the first concurrent compaction garbage collector, so if you're running a huge heap and need to avoid garbage collection pauses then you might want to take a look at their Zing JVM.

Symptoms

The symptoms of stop-the-world garbage collection include periodic CPU spikes and aberrant response times. You might observe that your application is performing well most of the time but once or twice an hour your SLAs set off alerts. If you have a performance monitoring solution in place then you can also observe an increase in the frequency of major garbage collection occurrences as well as a pattern of increase in your old generation followed by a dip. Remember that objects in the old generation can only be reclaimed by major garbage collections, so whenever you see the old generation shrink, it means that a major garbage collection occurred.

Impact

The impact of stop-the-world garbage collection is a pause for the duration of the major garbage collection. In terms of user experience, this means that periodically users are going to see egregiously slow response times, the frequency and duration of which will be proportional to the size of your heap and to the behavior of your application. Furthermore, if your heap's configuration does not allow short-lived objects enough time to die in the young generation then the impact will be exacerbated.

Troubleshooting

There are several ways of troubleshooting major garbage collections:

- Enable verbose garbage collection logging by adding the –verbosegc to your Java startup and search the resultant log file for "FULL GC" entries. Each "FULL GC" entry corresponds to a major collection.
- Use a performance monitoring tool that shows you major garbage collection occurrence counts and look for frequent increases
- Use a performance monitoring tool that shows you the behavior of your CPU and heap usage (preferably partitioned by generation) and look for CPU spikes and frequent increases and dips in heap usage

Avoiding this problem

Avoiding major garbage collections is almost impossible, but there are some things that you can do to help mitigate the problem:

1. Ensure that your heap is sized in such a way that short-lived objects are given enough time to die. The recommendation here is to set the young generation to a little less than half the size of the heap (-XX:NewSize and –XX:MaxNewSize command line arguments) and then to set the survivor ratios to anywhere between 1/6th and 1/8th the size of the young generation (-XX:SurvivorRatio=4 or –XX:SurvivorRatio=6. It's a strange formula, but 4 equates to 1/6th of the young generation and 6 equates to 1/8th the size of the young generation). This will make minor garbage collections take longer to run but they can run without freezing JVM threads.
2. Consider enabling concurrent mark sweep (CMS). This increases the CPU overhead on the machine upon which the JVM is running, but it can reduce the pause time for major collections.
3. Cycle your JVMs: if you are running in an elastic environment, such as in a cloud like Amazon's EC2, and you regularly scale up and down throughout the day, you might consider cycling down older machines first. In this capacity you can maintain a much larger heap and shut down the JVM before a major garbage collector ever has a chance to run.
4. Explore Azul System's Zing JVM with its concurrent compaction garbage collector to see if it is a good fit for your business needs.

5. Where's my memory? Memory leaks

Hey, wait a minute. I thought that automatic garbage collection means no more memory leaks. What gives?

If you come from a programming language like C or C++, in which you were responsible for managing memory yourself, then this style of traditional memory leak is truly avoided by the garbage collector. But if this is true, then why do we continue to see memory leaks in Java?

Problem

The symptoms of C/C++ style memory leaks and Java memory leaks are the same, but the manner in which we arrive there is different. Memory leaks in Java are much more of a reference management issue than they are true memory leaks. Figure 5.1 shows a comparison of C/C++ style memory leaks and Java memory leaks.

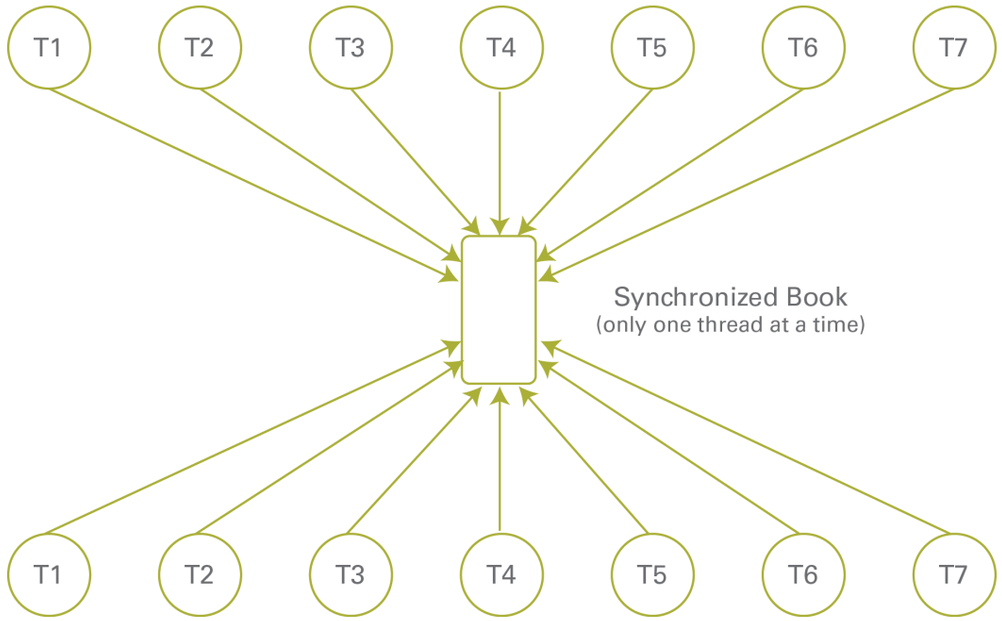

Figure 5.1 C++ style versus Java style memory leaks

Memory cont'd

In C/C++, a memory leak occurs when you allocate memory, assign the address of that memory to a reference variable, and then delete the reference to that memory without first de-allocating that memory. This is exactly what the Java garbage collector was designed to avoid, and from this perspective it works flawlessly. The left side of figure 5.1 shows an allocated chunk of memory that is lost when its reference is deleted. The JVM garbage collector handles this C/C++ style memory leak.

Java memory leaks are a different beast altogether: they occur when a Java application inadvertently maintains a reference to an object that it does not ever intend to use again. There is no definitive way for the garbage collector to assess the intentions of the developer who built the application, so if a reference to an object is maintained, the garbage collector will assume that someone intended to use it at some point in the future. Unfortunately this tends to occur in code that is frequently executed, causing the JVM to eventually exhaust its memory and throw the dreaded OutOfMemoryError, which usually means you have to restart your JVM.

To understand why this behavior truly is a problem, you need to understand the *reachability* test that the garbage collector performs as part of its *mark* phase. Referring again to Figure 5.1, when a garbage collection starts, the first thing it does is perform a reachability test. In this test the garbage collector looks at all objects directly visible by each running thread (these objects are collectively called the root set) and then walks over the heap from object to object until it has determined the set of objects that are *reachable*. All reachable objects are *marked* and then the garbage collector executes a *sweep* process that sweeps away all unmarked objects. In other words, it reclaims the memory referenced by all non-reachable objects. Therefore, if an object is reachable, or there exists a path from the root set to the object, then it cannot be reclaimed by the garbage collector.

Symptoms

The symptom of a Java memory leak is a gradual (or rapid, if it's a big memory leak) increase in memory usage until the heap eventually runs out of memory. Unfortunately there is no definitive way of differentiating between simply running out of memory and a memory leak, but there are ways to infer what is happening, which are summarized below.

Impact

9

Depending on the severity of your memory leak and how quickly it takes down your JVM, the impact can be severe. When a JVM runs out of memory, it must be killed and restarted before it can service any additional requirements. From an operations perspective, you'll find yourself switching between machines and restarting JVMs potentially several times a day.

Troubleshooting

Identifying the root cause of a memory leak is challenging but fortunately there are some behaviors of memory leaks that are common across the board. It may sound obvious, but memory leaks can only occur inside containers that support unbounded growth. Java natively supports unbounded growth through its Collection classes: it maintains lists, maps, and sets supported by linked lists, arrays, trees, hash tables, and more, which can all grow without bound. Consider a common scenario that leads to a memory leak: on a per-request basis (or on a subset of requests), the application adds an object to a collection but does not remove it from the collection when it's done. Because this occurs as a result of user interaction and the user continues to interact with your application, the end result is a memory leak.

The good thing is that because of this behavior, intelligent analysis can be performed to detect this behavior and point to suspected memory leaks. Figure 5.2 shows what this might look like in a production application with a memory leak.

Memory cont'd

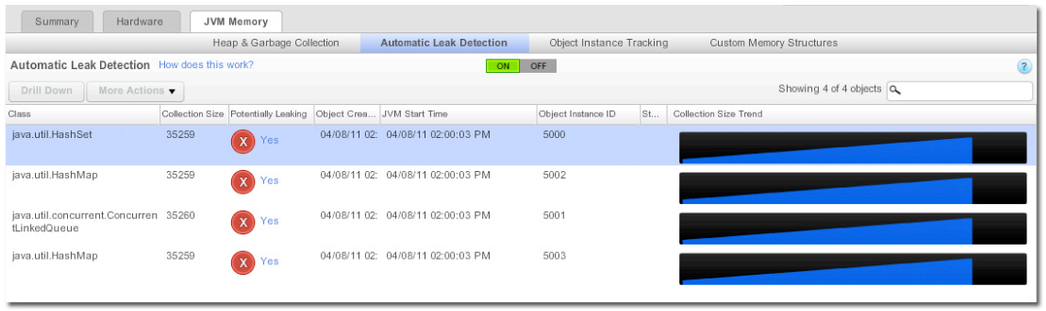

Figure 5.1 C++ style versus Java style memory leaks

Figure 5.2 shows the memory usage pattern by a specific collection instance. In this case the memory used by the HashSet is increasing over time and, at the time this diagram was built, the HashSet contained 35,259 elements, which points to the potential for a memory leak.

Finally, you can troubleshoot the root cause of a memory leak after the fact (which is not ideal) by requesting the JVM to capture a heap dump when an OutOfMemoryError occurs. This is accomplished by using the -XX:-HeapDumpOnOutOfMemoryError JVM argument, which is documented on Oracle's web site. Analyzing the resultant heap dumps can be a daunting task, but in the past I have found the Eclipse Memory Analyzer (MAT) to be a helpful tool.

Avoiding this problem

Avoiding memory leaks requires close scrutiny of code where memory leaks can occur, namely around Java collection classes. Additionally, I have found that memory leak-like symptoms can occur around session management: memory used by user sessions will eventually be reclaimed by the garbage collector, but if the heap exhausts its memory before the sessions expire, then session memory can cause OutOfMemoryErrors.

Here are some tips to help you avoid memory leaks:

1. Share common memory leak scenarios, such as managing data in Java Collection classes, with your developers and ask them to review their code in that light
2. Employ a Java memory profiler to analyze code during development. Execute a load test against our database and tune your database to optimally support these queries.
3. Monitor the application in production to detect potential memory leaks (including capabilities for observing changes in collection classes)
4. Configure your JVM to capture a heap dump when an OutOfMemoryError occurs and analyze that heap dump using a tool like Eclipse Memory Analyzer

Concurrency

Concurrency

It is rare to find an enterprise application that doesn't need to do more than one thing at once. **Concurrency** refers to executing several computations simultaneously. Every programming language has various strategies for managing concurrently running code. Running multiple simultaneous threads is a simple task as long as they do not interact with **mutable shared objects** (objects that are accessible by more than one thread and those objects can be changed).

Concurrency refers to the execution of several computations simultaneously, or, in other words, multithreaded programming. Concurrency is important because it allows your application to accomplish more work in less time.

Mutable shared objects are objects that are shared or accessible by multiple threads, but can also be changed by multiple threads. Ideally any objects that are shared between threads will be immutable, meaning that they cannot be changed – immutable shared objects do not pose challenges to multithreaded code.

Consider two threads that are simultaneously trying to increment the number 5 contained in variable A. If there are two increment operations then when all is said and done, A should have the value of 7 (5 + 1 + 1 = 7). Consider the situation when both threads run at the same time and the increment operator is not atomic (meaning that it is not guaranteed to run in a single operation.) In this case the increment operation retrieves the value from the variable A, adds 1 to it, and then assigns it back to the variable A. Consider that thread 1 reads the value of A (5) and then thread 2 reads the value of A (5), then thread 1 increments it to 6 and assigns it back to A followed by thread 2 that does the same thing (increments 5 to 6 and assigns it back to A.) So what is the value of A after this? A has the value of 6, which is wrong.

In Java we manage thread concurrency using synchronization. Each Object in Java has a lock and when a thread wants to execute the code in a synchronized block (denoted by the synchronized keyword) it must first obtain the object's lock. If the lock is not available then it means that another thread already has the lock so this thread must wait for the lock to be released. As you might guess, synchronization can lead to all kinds of functional and performance issues.

This section reviews three concurrency-related performance problems:

- Thread deadlocks
- Thread gridlocks
- Thread pool sizing issues

6. Hot fudge sundae standoff: Thread deadlocks

Consider two children that want to make hot fudge sundaes: the first child grabs the ice cream and the second child grabs the chocolate syrup. The first child has ice cream, but no chocolate, and refuses to give up his ice cream until he gets the chocolate. The second child has the chocolate, but no ice cream, and refuses to give up his chocolate until he gets the ice cream. In this scenario, who gets the hot fudge sundae?

I know what you're thinking: the bigger kid takes what he wants from the smaller kid. In Java programs, however, there is no bigger kid. Instead, they both starve.

Problem

Deadlocks occur when two or more threads need multiple shared resources to complete their task and they access those resources in a different order or a different manner. Java concurrency works under the notion of "locks." When a method or a block of code is *synchronized*, the executing thread obtains the lock for the object upon which the code is synchronized, executes the synchronized code, and then relinquishes that lock. If a second thread attempts to execute the synchronized code while the first thread has the lock then the second thread "waits" until the lock is available. If the lock is never released then that thread will wait forever, or until the JVM is restarted.

What happens in this scenario is thread 1, which has the first lock, needs to execute a block of synchronized code, but that lock is held by thread 2. And thread 2 is waiting on the lock held by thread 1. Thread 1 has the ice cream and thread 2 has the chocolate, so these two threads are deadlocked.

Code deadlocks occur when two or more threads each possess the lock for a resource the other thread needs to complete its task and neither thread is willing to give up the lock that it has already obtained.

In a synchronized code block, a thread must first obtain the lock for the code block before executing that code and, while it has the lock, no other thread will be permitted to enter the code block.

Concurrency cont'd

Symptoms

When your application has a deadlock in it, your JVM will eventually exhaust all or most of its threads. The application will appear to be accomplishing less and less work, but the CPU utilization of the machine on which the application is running will appear underutilized. Additionally, if you request a thread dump you will see reports of deadlocked threads.

Impact

10 

Deadlocks are serious business and will eventually cause your application to stop processing business transactions. Even worse, the only way to resolve the issue is to restart your JVM, which takes it out of availability to service your users. Finally, because deadlocks are the result of **race conditions** (multiple threads competing for resources and typically using them for a very short period of time) they are very difficult to reproduce in a non-production environment and hence very difficult to troubleshoot.

Troubleshooting

As mentioned above, deadlocks are nearly impossible to reproduce in a development environment so troubleshooting one is very challenging. The only way that I have successfully discovered the root cause of a deadlock is by capturing a thread dump while two threads were deadlocked and then examining the stack traces of the deadlocked threads. This strategy points out where a deadlock occurred but it does not provide sufficient application context to definitively determine why it occurred. In my case I then examined the code and built theories about what could have happened and because of my understanding of the code, I was able to derive a reasonable case for it happening. Luckily after resolving the potential issue we did not observe the deadlock again.

Avoiding this problem

Avoiding deadlocks is only really accomplished by making your application and its resources as immutable as possible. If you are interacting with immutable resources (resources that cannot change) then you won't need to synchronize access to those resources, avoiding deadlocks altogether. But, depending on your application business requirements, this may not be possible. My only recommendation, if you are unable to make your resources immutable, is to use thread synchronization as sparingly as you can and to search for potential interactions between threads that might enter two or more synchronized blocks in a single business transaction.

7. Accident ahead, merge right: Thread gridlocks

If you live in a busy metropolitan area and don't take public transportation, then chances are you could probably best describe your daily commute as "hurry up and wait." Traffic going in and out of big cities is pretty abysmal on an average day, but when an accident occurs things get dramatically worse – sometimes you can be stuck in gridlock for hours before you finally get past the bottleneck.

We just finished up talking about thread deadlocks, but there is another performance issue that we can face with thread synchronization: for lack of a formal term I'll call it gridlock. An application under heavy load is like a freeway during rush hour – things are moving a little slower than usual, but for the most part everything's working fine. If your application is "over-synchronized," however, then you've essentially merged all the lanes of your freeway down to one, resulting in a lot of slow and stalled threads (and some unhappy end users).

To give you a real-world example of this, in a previous company our rule engine was not thread-safe and almost every request was required to execute rules. Under small amounts of load this was not a problem, just like merging from two lanes to one lane with a handful of cars is not very problematic. But when load increased we saw a dramatic jump in response time, just like experiencing that same traffic merger during rush hour. So although we were able to support hundreds of simultaneous requests on the box, in the end we built a single-threaded lane through which all traffic needed to pass.

Concurrency cont'd

Problem

Thread synchronization is a powerful tool for protecting shared resources, but if major portions of your code are synchronized you might be inadvertently single-threading your application. Figure 7.1 shows what this might look like in your application.

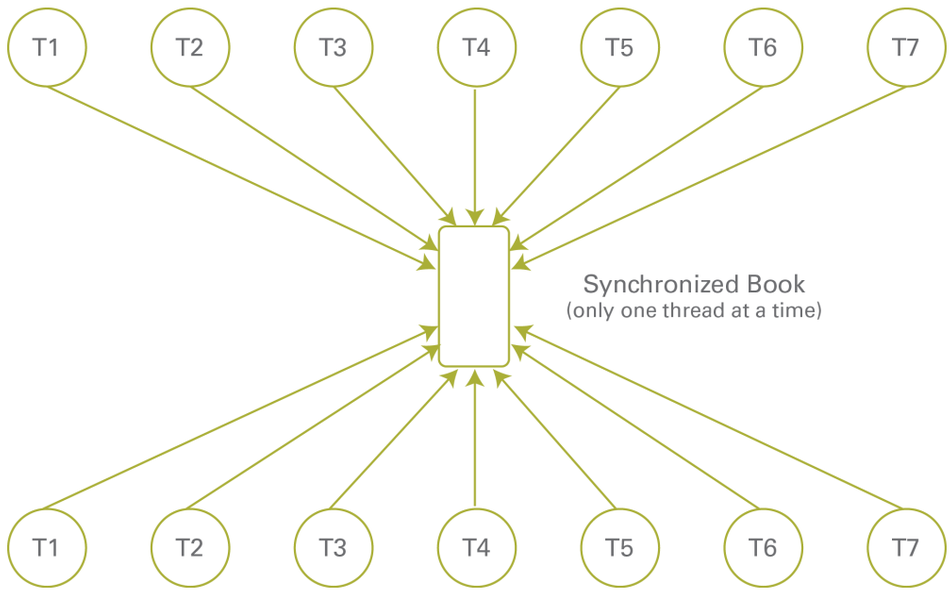

Figure 7.1 All threads merging through a single synchronized block

Symptoms

If your application has too much synchronization or synchronization through a core piece of functionality that is required by a large number of business transactions, then you will see slow response times with very low CPU utilization. The reason for this is that each of these threads is going to reach the synchronized code and go into a waiting state. So essentially one thread will be active at a time, which typically does not require much CPU to run. Additionally, if you capture a thread dump then you will see a large number of threads in a legitimate wait state: WAITING or TIMED_WAITING

Impact

Over-synchronization can have a dramatic impact on performance and if you are experiencing it, things are not going to get any better until you remove that block. And worse still, the problem gets worse as load increases. I remember that this problem was one that caused me many late nights and overtime.

Troubleshooting

In order to identify the root cause of over-synchronization you will need either a method-level view of the performance of your business transactions or frequently captured thread dumps. You need to know where your code is waiting and why each of those threads is waiting. Once you know where your threads are waiting, it's up to you to refactor the code to eliminate that synchronized requirement. The refactoring effort is why I mentioned that this led to late nights and overtime: it's not a quick fix.

Avoiding this problem

The same advice for deadlocks is applicable for gridlocks: try to use immutable resources whenever possible, use synchronization sparingly, and perform deep analysis of synchronized code blocks to determine their impact on your business transactions. Thread synchronization issues are really solved at the architectural level rather than at the implementation level; your application has to be architected to avoid these types of problems.

8. Please hold, your call will be answered in the order received: Thread pool configuration gridlocks

Have you ever had the unpleasant experience of calling your cable provider to troubleshoot a problem? If you call to upgrade your service you'll be helped in a matter of seconds, but if you're trying to get help you'd better block off the better part of your day. Call centers, like thread pools, are of a finite size, and when they're too small for the amount of traffic they're required to service you can end up with some angry end users.

If your application is running in an application server or web container, it will have a thread pool configured to control how many requests your application can concurrently process. Figure 8.1 shows the behavior of a thread pool in a web or enterprise application.

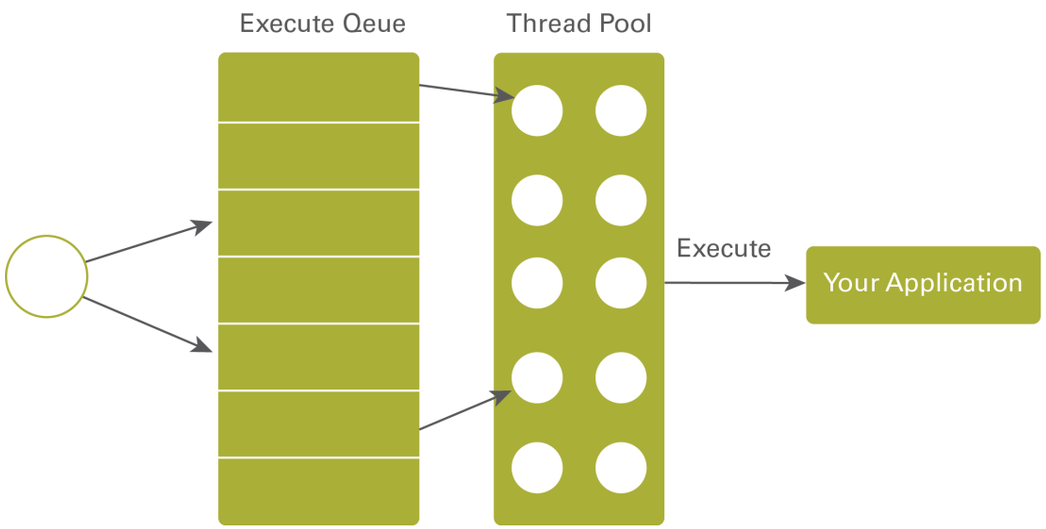

Figure 8.1 server request threading model

The server receives a request by a socket listener, places it in an execution queue, and then returns to listen for the next request to arrive on the socket. The execution queue is serviced by a thread pool. When a thread is available in the thread pool, a request is removed from the execution queue and passed to a thread for processing.

The thread executes the appropriate business transaction in your application code. When the thread completes processing the thread is returned to the thread pool and will be available to process another request.

Problem

The configuration of the size of that thread pool is going to be of paramount importance to the performance of your application. If the thread pool is sized too small then your requests are going to wait (much like you would while on hold for your cable provider) but if the thread pool is sized too large then too many threads are going to execute concurrently and take all of the machine's processing resources. When there are too many threads and the machine spends too much time context switching between threads, the threads will be "starved" of CPU cycles and will take longer to complete. You have a finite number of cores in your CPU and if threads need computing power (they aren't waiting) then there's only so much processing power to go around. The behavior of your application will dictate the optimal size of your thread pools.

Symptoms

When a thread pool is sized too small then you will see the thread pool utilization (number of active threads divided by the total number of threads) at or near 100%, requests backing up in the execution queue, and the CPU underutilized. The reason is that the application has used all available threads but is not using all of the processing capabilities available to it on the machine. And because it cannot process requests fast enough, requests back up.

When a thread pool is sized too large then you will see a moderately used thread pool, very few if any pending requests in the execution queue, but the CPU utilization will be at or near 100%. The reason for this is that the application has more threads than the machine is equipped to process.

Concurrency cont'd

Impact

7

Both situations are high impact on the performance of the application, and hence your user experience, but for different reasons. With a small thread pool, your application is not attempting to do as much as it can do (the under-performer). With a large thread pool, your application is trying to do too much (the over-achiever that takes on too much work and ends up not delivering).

Troubleshooting

Troubleshooting thread pool sizing problems is actually a lot easier than the other performance challenges presented in this eBook. The key is to look at two metrics:

- Thread pool utilization
- CPU utilization

If your thread pool utilization is high (at or near 100% with pending requests) and your CPU utilization is moderate then your thread pool is probably configured too small.

If your thread pool utilization is moderate, but your CPU utilization is high then your thread pool is probably configured too large.

Avoiding this problem

The way to avoid thread pool misconfigurations is to tune the thread pool. This sounds easy in theory, but it's time consuming in practice. In short, you want to size your pool large enough to properly utilize your machine's processing resources but not large enough to saturate them. But there's a complication: if your application accesses external resources, such as a database, your application should be configured to send enough load to those external resources to use them effectively, but not saturate them.

I have detailed a performance tuning strategy called Wait-Based Tuning that I have successfully used in customer environments. The essence of this strategy is, through an iterative process, to determine the capacity of your external resources, configure your application to send the appropriate volume of requests to those resources (using a gating control like a database connection pool) and then tune backwards all the way to your thread pool. The goal is to leave pending requests in the execution queue and streamline requests through your application architecture.

You can read more about Wai-Based Tuning on InfoQ.com.

DEFCON 1

DEFCON 1

The defense readiness condition (DEFCON) is a system that defines graduated levels of readiness for the U.S. Military, but because of its popularity in movies and on the news, it has come to connote the severity of some measured condition. DEFCON 5 refers to "the lowest state of readiness," or peace, while DEFCON 1 refers to "nuclear war is imminent," or the most heightened state of readiness. I titled this section DEFCON 1 because it presents performance issues, from a process perspective, that you can't afford to ignore. These are the things that, if you do not address them, will result in working long nights and weekends and maybe even losing your job.

Specifically this section addresses two DEFCON 1 issues:

- Considering performance as an afterthought
- Waiting for your users to tell you about performance problems

9. We don't need no stinking performance: Performance as an afterthought

If you wanted to build a racecar, would you start with a basic sedan and then make upgrades, or would you build it from scratch? You'd build it from scratch, because a sedan isn't built for speed, and the work and resources that would be required to turn a sedan into a racecar would far exceed those needed to build a racecar from the start. The same holds true of your application – if you didn't design it for performance, then it'll take a lot of time and effort to make it perform once it's built. The best solution is to consider performance as an important feature from the very beginning.

Problem

Software architects and developers are typically pretty good at coming up with elegant solutions to complex problems. After all, it's what they're paid to do. However, sometimes the most elegant or concise solution isn't always the best-performing solution, as anyone who's ever deployed new code to production will know. This is true both of high-level architecture as well as low-level implementation details throughout the entire project. Should you use a HashMap or a TreeMap here? What's the best algorithm to sort this data? These are questions that development teams face on a day-to-day basis, but their decisions don't always take into consideration the performance of their code in a production environment.

Aside from analyzing the performance and scalability implications of your application architecture for when you eventually deploy it to a production environment, a problem that you can tackle immediately is: when do you start measuring the performance of your code? If your answer is that you build the functionality of your application completely first and then look at refactoring your code for performance later, then you're going to find yourself in a heap of rework (you just built a Ford Pinto that you now need to turn into a 10-second street racing car!).

Note: If you find yourself in this camp, don't beat yourself up, you're in good company. The majority of the customers I visited in my performance tuning consulting practice followed the same strategy: make the application function properly first and then worry about performance. I'm here to encourage you to change this behavior, but just note that this is how most companies develop software.

Symptoms

Your company has spent anywhere between six months and several years building your application. You've been through program quality assurance (sometimes just called PQA) and it meets its functional requirements, but it falls over in load testing. You thought you were ready to ship the product but it turns out you're months behind because you need to analyze the performance of your application and refactor the components that do not meet performance requirements.

Impact

10 

The impact of this problem is a 10 because it will require many hours, days or even weeks of development work to be fixed. The amount of work and time required depends on the nature of the performance issues you've unveiled, but in my experience the problem is usually related to the architecture of the application, which requires substantial work to fix. When working with customers to discover these problems, I typically left them with a two-fold plan: (1) Identify and resolve the "quick fixes" to improve performance for the time being, and (2) Address the underlying architectural issues that you need to resolve in the long-term.

Troubleshooting

Troubleshooting the root cause of performance and scalability issues is a non-trivial task. My approach was always to first review the application architecture and understand how a request traverses through the application from when the web server receives the request to when a result is rendered back to the user. That architectural analysis identifies points of performance contention that I would then validate using performance analysis tools. Without performance analysis tools you're limited to adding additional logging and reviewing those logs – it can be done, but it is a time-consuming activity. For more information on understanding how your application architecture affects performance, review my performance-tuning article on InfoQ.com.

Avoiding this problem

The simple way to avoid this problem is to implement proactive performance analysis of your application code during development. A strategy that I've successfully employed is to measure the performance of code as it is being developed at the unit test level. If you write very granular unit test cases – which you should – they will provide a natural point for measuring performance. If you can capture the response time of unit tests as well as their memory utilization, then you can compare that across builds and detect changes in performance. There are commercial tools to help you do this, and I have written an article on doing this type of analysis using HProf, which is a profiler built into the JVM, that you can find on InformIT.com. The best strategy is to integrate this level of analysis into your continuous integration environment and perform the analysis programmatically. Manual analysis is tedious and, in the end, you'll find yourself skipping it when time is short, but if the analysis can be automated then you just need to review reports that are associated with your build, just as you would test failures.

10. Monitoring on the super-cheap: Having your users tell you about performance problems

If you've worked in IT for long, you probably know the embarrassment of getting a call from your boss telling you that users have started reporting performance problems. It's far better to detect the start of a problem and resolve it before your users are ever aware there's a problem than it is to have to get a call from your boss telling you something's wrong. By the time the performance issue has been reported by your users, made it way up to executives and back down to your boss, your end users and your business have already been affected by the performance issue – not to mention your job security.

Problem

Performance problems rarely appear out of the blue like an F5 Tornado. Instead, they typically approach like a hurricane, with warning signs that an issue is developing long before your or your end users notice the problem. The challenge for you as an IT professional is to equip your systems to detect and alert on these signs so that you can begin addressing the problem before your end users start complaining.

There are various technologies that you can use to detect performance problems, including:

- Java Management Extensions (JMX): most application servers and web containers report metrics about their internal behavior through JMX. You will find things like thread pool usage, connection pool usage, cache hit/miss ratios, concurrent sessions, and more. JMX is a powerful technology and there are commercial and free tools that retrieve JMX metrics from servers for you. The benefit to JMX is that it exposes runtime metrics about the container/application server in which the application is running. The drawback to JMX is that it seldom presents response time metrics and does not tie requests to the response times of the methods that satisfy a request.
- Business Transaction Performance: It's important to at least capture the response time and execution counts for your **business transactions**. It would be even better to understand, on whatever level of granularity you want, the minimum, maximum, and average response times for business transactions as well as the standard deviation of response times so that you can more effectively assess the impact of outlier response times. From a business transaction perspective you can capture response times using AOP or even a simple Servlet Filter. There are free and commercial tools to help you capture this information without manual coding on your part. The benefit to capturing business transactions is that they reflect the behavior of your application as experienced by your users. The drawback to using business transactions alone is that while they show user behavior, they need to be combined with container metrics, such as JMX, in order to understand whether or not a method is slow because of the code or because of the environment.

DEFCON 1 cont'd

- Method-Level Response Times: When something goes wrong, you need to identify where it goes wrong. This is a little harder to capture without a commercial tool, but the end goal is to identify the response time of each method and associate that response time to individual business transactions. When a business transaction runs slow, identifying the methods contributing to the total response time will help you identify the root cause. The benefit to capturing method-level response times is that they identify hotspots in your application. The drawback, however, is that hotspots do not pinpoint the impact of the application on your users because they are not tied to business transactions.
- Thread dumps: If you do not have the luxury of having a commercial tool monitoring your environment then you do still have the option of capturing thread dumps and analyzing the time spent in methods that way. This is a challenging exercise and involves capturing thread dumps on a very frequent basis, such as every 50ms or 100ms, and then looking at methods that are still running between thread dumps. Your sampling period will dictate the granularity with which you can trust the results. But without modifying your code to associate a business transaction with a thread identifier, you will not be able to identify which business transaction a method belongs to. The benefit to using thread dumps is that they show exactly what each thread is doing at a point in time and analyzing thread dumps will identify threads that are stuck. The drawback is that you need multiple thread dumps in close proximity and it is a labor-intensive process to be able to interpret what your application is doing.
- Memory Dump Analysis: If you observe memory issues in your application then you can trigger memory dumps and use a tool like MAT to review the contents of the memory dump. Without tools this is a complex process. The benefit to capturing memory dumps is that they allow you to understand what objects are consuming the memory in your JVM. The drawback is that the analysis is very complex and would be better suited to an automated process.
- Packet Analysis: Depending on how deep you need to go, you can review individual network packets as they pass from machine to machine to gain insight into the behavior of your application – and even identify data being passed between your application and other services. This is not a job for the faint of heart, but it is very powerful. If you are interested in learning more, take a look at the free and open source Wireshark project and the book Practical Packet Analysis: Using Wireshark to Solve Real-World Network Problems to get you started. The benefit to packet analysis is that you can track the interactions between servers and their response times as well as track interactions using any transmission protocol. The drawback is that packet analysis only shows part of the behavior of your application and it requires a significant investment of your time to learn how to do it right.

A *business transaction* represents an interaction with your application, either from a user or a system. But more than that, a business transaction defines this interaction in terms of the functionality of your application. An example is a web request pattern that you can translate to a significant interaction with your application such as "Add an item to a shopping cart" (e.g. POST /cart) or "View Product" (e.g. GET /product).

Symptoms

It probably goes without saying, but the symptoms of ignoring performance until your users alert you to the problem are calls from users, the escalation of possibly trivial problems to executives in your company, and late nights and weekends of work as all other priorities are thrown to the wind until this performance issue is resolved!

Impact

10 

While a performance issue might not be the end of the world, bad publicity and loss of customer loyalty can do lasting damage to your company. The impact of this kind of issue varies greatly by industry and business model – for example, an eCommerce application will suffer much more than an internal finance or CRM application in a small business. A poor user experience is never a good thing, however, and you'll look much better to your boss if you can catch performance issues before the end users start calling.

DEFCON 1 cont'd

Troubleshooting

As mentioned above, there are various commercial and free tools to help you put some degree of performance monitoring in place. Consider the various technologies listed above, from JMX monitoring through full business transaction monitoring with method-level response time granularity, and determine what price point and featureset best suit your application and your business. Also be aware that while generic monitoring solutions are challenging to build, specific monitoring solutions are far easier. You might not want to re-engineer your entire application, but adding a Servlet filter to capture response times and a couple counters can go a long way toward helping you address these problems before you make the investment into a robust monitoring solution.

Avoiding this problem

After you assess the impact of performance issues in your environment, if you determine that bad press and a poor user experience could dramatically hurt your business, then the best way to manage performance is to invest in a performance monitoring tool. You can piece together technologies to help you, but if you have a revenue-generating application then some vendors can get you very detailed performance information in a matter of minutes. And if you do not want to invest in performance monitoring infrastructure then some vendors also provide monitoring solutions securely in a Software-as-a-Service (SaaS) model, so that your environment in only marginally impacted.

Regardless of whether you decide to purchase a commercial product or pull together a set of monitors to give you some insight into your application performance, the important thing is that you do something. Trust me, you won't regret detecting a performance problem and resolving it before your users are even aware that there was a problem to begin with!

FURTHER READING

If you found this information interesting, you can read more about these strategies in my book, Pro Java EE 5 Performance Management and Optimization.

Conclusion

You could easily spend years learning and refining your knowledge of performance analysis. This eBook just scratched the surface of common problems that you may have already observed in your own environment – it's not meant to be exhaustive, but rather to alert you to common problems and to raise awareness of the need to take performance seriously in every stage of the application lifecycle.

This eBook reviewed performance issues in four main categories:

- Database: Because databases ultimately back enterprise applications, we reviewed three common performance issues involving the database: database persistence configuration, caching, and database connection pool configuration.

- Memory: While automatic memory management was one of Java's strongest innovations, it is both a blessing and a curse that can lead to performance issues. This eBook reviewed the two most common memory-related performance issues: garbage collection configuration and memory leaks.

- Concurrency: As applications increase in comaplexity they need to accomplish more operations at the same time, which leads to concurrency concerns. This eBook reviewed three common concurrency and threading issues: thread deadlocks, thread gridlocks, and thread pool configuration.

- DEFCON 1: This eBook concluded by discussing two "big picture" no-nos: Deferring performance considerations to the end of your project and ignoring performance until your users complain about performance issues

I hope you found this eBook useful, but more importantly, I hope that you took this brief overview of common performance issues as a call to take performance seriously. Hopefully you walked away from this eBook armed with a little more knowledge about how to avoid the worst of performance issues in your application.

APPDYNAMICS

www.appdynamics.com