Achieving Performance, Scalability, and Availability Objectives

This is an extract from Aleksandar Seovic's book on Oracle Coherence. He is founder and Managing Director at S4HC, Inc., where he leads professional services practice. Aleksandar lead the implementation of Oracle Coherence for .NET, a client library that allows applications written in any .NET language to access data and services provided by Oracle Coherence data grid.

Building a highly available and scalable system that performs well is no trivial task. In this chapter, we will look into the reasons why this is the case, and discuss what can be done to solve some of the problems.

It will also explain how Coherence can be used to either completely eliminate or significantly reduce some of these problems and why it is a great foundation for scalable applications.

Achieving performance objectives

There are many factors that determine how long a particular operation takes. The choice of the algorithm and data structures that are used to implement it will be a major factor, so choosing the most appropriate ones for the problem at hand is important.

However, when building a distributed system, another important factor we need to consider is network latency. The duration of every operation is the sum of the time it takes to perform the operation, and the time it takes for the request to reach the application and for the response to reach the client.

In some environments, latency is so low that it can often be ignored. For example, accessing properties of an object within the same process is performed at in-memory speed (nanoseconds), and therefore the latency is not a concern. However, as soon as you start making calls across machine boundaries, the laws of physics come into the picture.

PACKT
PUBLISHING

Dealing with latency

Very often developers write applications as if there is no latency. To make things even worse, they test them in an environment where latency is minimal, such as their local machine or a high-speed network in the same building.

When they deploy the application in a remote datacenter, they are often surprised by the fact that the application is much slower than what they expected. They shouldn't be, they should have counted on the fact that the latency is going to increase and should have taken measures to minimize its impact on the application performance early on.

To illustrate the effect latency can have on performance, let's assume that we have an operation whose actual execution time is 20 milliseconds. The following table shows the impact of latency on such an operation, depending on where the server performing it is located. All the measurements in the table were taken from my house in Tampa, Florida.

Location	**Execution time (ms)**	**Average latency (ms)**	**Total time (ms)**	**Latency (% of total time)**
Local host	20	0.067	20.067	0.3%
VM running on the local host	20	0.335	20.335	1.6%
Server on the same LAN	20	0.924	20.924	4.4%
Server in Tampa, FL, US	20	21.378	41.378	51.7%
Server in Sunnyvale, CA, US	20	53.130	73.130	72.7%
Server in London, UK	20	126.005	146.005	86.3%
Server in Moscow, Russia	20	181.855	201.855	90.1%
Server in Tokyo, Japan	20	225.684	245.684	91.9%
Server in Sydney, Australia	20	264.869	284.869	93.0%

As you can see from the previous table, the impact of latency is minimal on the local host, or even when accessing another host on the same network. However, as soon as you move the server out of the building it becomes significant. When the server is half way around the globe, it is the latency that pretty much determines how long an operation will take.

Of course, as the execution time of the operation itself increases, latency as a percentage of the total time will decrease. However, I have intentionally chosen 20 milliseconds for this example, because many operations that web applications typically perform complete in 20 milliseconds or less. For example, on my development box, retrieval of a single row from the MySQL database using **EclipseLink** and rendering of the retrieved object using **FreeMarker template** takes 18 milliseconds on an average, according to the **YourKit Profiler**.

PACKT
PUBLISHING

On the other hand, even if your page takes 700 milliseconds to render and your server is in Sydney, your users in Florida could still have a sub-second response time, as long as they are able to retrieve the page in a single request. Unfortunately, it is highly unlikely that one request will be enough. Even the extremely simple Google front page requires four HTTP requests, and most non-trivial pages require 15 to 20, or even more. Each image, external CSS style sheet, or JavaScript file that your page references, will add latency and turn your sub-second response time into 5 seconds or more.

You must be wondering by now whether you are reading a book about website performance optimization and what all of this has to do with Coherence. I have used a web page example in order to illustrate the effect of extremely high latencies on performance, but the situation is quite similar in low-latency environments as well.

Each database query, each call to a remote service, and each Coherence cache access will incur some latency. Although it might be only a millisecond or less for each individual call, it quickly gets compounded by the sheer number of calls.

With Coherence for example, the actual time it takes to insert 1,000 small objects into the cache is less than 50 milliseconds. However, the elapsed wall clock time from a client perspective is more than a second. Guess where the millisecond per insert is spent.

This is the reason why you will often hear advice such as "make your remote services coarse grained" or "batch multiple operations together". As a matter of fact, batching 1,000 objects from the previous example, and inserting them all into the cache in one call brings total operation duration, as measured from the client, down to 90 milliseconds!

The bottom line is that if you are building a distributed application, and if you are reading this book you most likely are, you need to consider the impact of latency on performance when making design decisions.

Minimizing bandwidth usage

In general, bandwidth is less of an issue than latency, because it is subject to Moore's Law. While the speed of light, the determining factor of latency, has remained constant over the years and will likely remain constant for the foreseeable future, network bandwidth has increased significantly and continues to do so.

However, that doesn't mean that we can ignore it. As anyone who has ever tried to browse the Web over a slow dial-up link can confirm, whether the images on the web page are 72 or 600 DPI makes a big difference in the overall user experience.

PACKT PUBLISHING

So, if we learned to optimize the images in order to improve the bandwidth utilization in front of the web server, why do we so casually waste the bandwidth behind it? There are two things that I see time and time again:

- The application retrieving a lot of data from a database, performing some simple processing on it, and storing it back in a database.
- The application retrieving significantly more data than it really needs. For example, I've seen large object graphs loaded from database using multiple queries in order to populate a simple drop-down box.

The first scenario above is an example of the situation where moving the processing instead of data makes much more sense, whether your data is in a database or in Coherence (although, in the former case doing so might have a negative impact on the scalability, and you might actually decide to sacrifice performance in order to allow the application to scale).

The second scenario is typically a consequence of the fact that we try to reuse the same objects we use elsewhere in the application, even when it makes no sense to do so. If all you need is an identifier and a description, it probably makes sense to load only those two attributes from the data store and move them across the wire.

In any case, keeping an eye on how network bandwidth is used both on the frontend and on the backend is another thing that you, as an architect, should be doing habitually if you care about performance.

Coherence and performance

Coherence has powerful features that directly address the problems of latency and bandwidth.

First of all, by caching data in the application tier, Coherence allows you to avoid disk I/O on the database server and transformation of retrieved tabular data into objects. In addition to that, Coherence also allows you to cache recently used data in-process using its near caching feature, thus eliminating the latency associated with a network call that would be required to retrieve a piece of data from a distributed cache.

Another Coherence feature that can significantly improve performance is its ability to execute tasks in parallel, across the data grid, and to move processing where the data is, which will not only decrease latency, but preserve network bandwidth as well.

Leveraging these features is important. It will be much easier to scale the application if it performs well—you simply won't have to scale as much.

PACKT PUBLISHING

Achieving scalability

There are two ways to achieve scalability: by **scaling up** or **scaling out**.

You can scale an application up by buying a bigger server or by adding more CPUs, memory, and/or storage to the existing one. The problem with scaling up is that finding the right balance of resources is extremely difficult. You might add more CPUs only to find out that you have turned memory into a bottleneck. Because of this, the law of diminishing returns kicks in fairly quickly, which causes the cost of incremental upgrades to grow exponentially. This makes scaling up a very unattractive option, when the cost-to-benefit ratio is taken into account.

Scaling out, on the other hand, implies that you can scale the application by adding more machines to the system and allowing them to share the load. One common scale-out scenario is a farm of web servers fronted by a load balancer. If your site grows and you need to handle more requests, you can simply add another server to the farm. Scaling out is significantly cheaper in the long run than scaling up and is what we will discuss in the remainder of this section.

Unfortunately, designing an application for scale-out requires that you remove all single points of bottleneck from the architecture and make some significant design compromises. For example, you need to completely remove the **state** from the application layer and make your services **stateless**.

Stateless services do not exist

Well, I might have exaggerated a bit to get your attention. It is certainly possible to write a completely stateless service:

```
public class HelloWorldService {
  public String hello() {
    return "Hello world!";
  }
}
```

However, most "stateless" services I've seen follow a somewhat different pattern:

```
public class MyService {
  public void myServiceMethod() {
    loadState();
    doSomethingWithState();
    saveState();
  }
}
```

PACKT
PUBLISHING

Implementing application services this way is what allows us to scale the application layer out, but the fact that our service still needs state in order to do anything useful doesn't change. We haven't removed the need—we have simply moved the responsibility for state management further down the stack.

The problem with that approach is that it usually puts more load on the resource that is the most difficult and expensive to scale—a relational database.

Scaling a database is hard

In order to provide **ACID** (**atomicity**, **consistency**, **isolation**, and **durability**) guarantees, a relational database needs to perform quite a bit of locking and log all mutating operations. Depending on the database, locks might be at the row level, page level, or even table level. Every database request that needs to access locked data will essentially have to wait for the lock to be released.

In order to improve concurrency, you need to ensure that each database write is committed or rolled back as fast as possible. This is why there are so many rules about the best ways to organize the disk subsystem on a database server. Whether it's placing log files on a different disk or partitioning large tables across multiple disks, the goal is to optimize the performance of the disk I/O as it should be. Because of durability requirements, database writes are ultimately disk bound, so making sure that the disk subsystem is optimally configured is extremely important.

However, no matter how fast and well-optimized your database server is, as the number of users increases and you add more web/application servers to handle the additional load, you will reach a point where the database is simply overwhelmed. As the data volume and the number of transactions increase, the response time will increase exponentially, to the point where your system will not meet its performance objectives anymore.

When that happens, you need to scale the database.

The easiest and the most intuitive approach to database scaling is to scale up by buying a bigger server. That might buy you some time, but guess what—if your load continues to increase, you will soon need an even bigger server. These big servers tend to be very expensive, so over time this becomes a losing proposition. One company I know of eventually reached the point where the incremental cost to support each additional user became greater than the revenue generated by that same user. The more users they signed up, the more money they were losing.

So if scaling up is not an answer, how do we scale the database out?

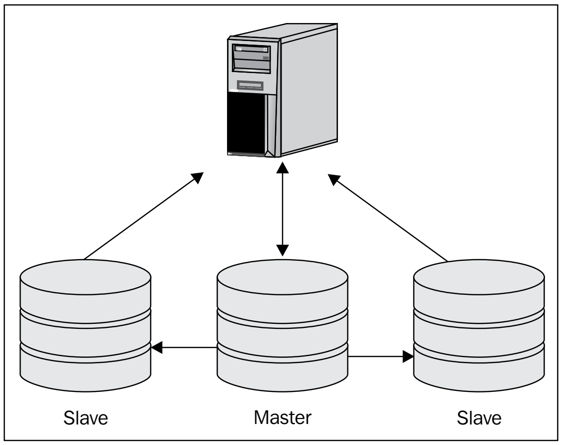
PACKT
PUBLISHING

Database scale-out approaches

There are three main approaches to database scale-out**: master-slave replication**, **clustering**, and **sharding**. We will discuss the pros and cons of each in the following sections.

Master-slave replication

Master-slave replication is the easiest of the three to configure and requires minimal modifications to application logic. In this setup, a single **master** server is used to handle all write operations, which are then replicated to one or more **slave** servers asynchronously, typically using log shipping:



This allows you to spread the read operations across multiple servers, which reduces the load on the master server.

From the application perspective, all that you need to do is to modify the code that creates the database connections to implement a load balancing algorithm. Simple round-robin server selection for read operations is typically all you need.

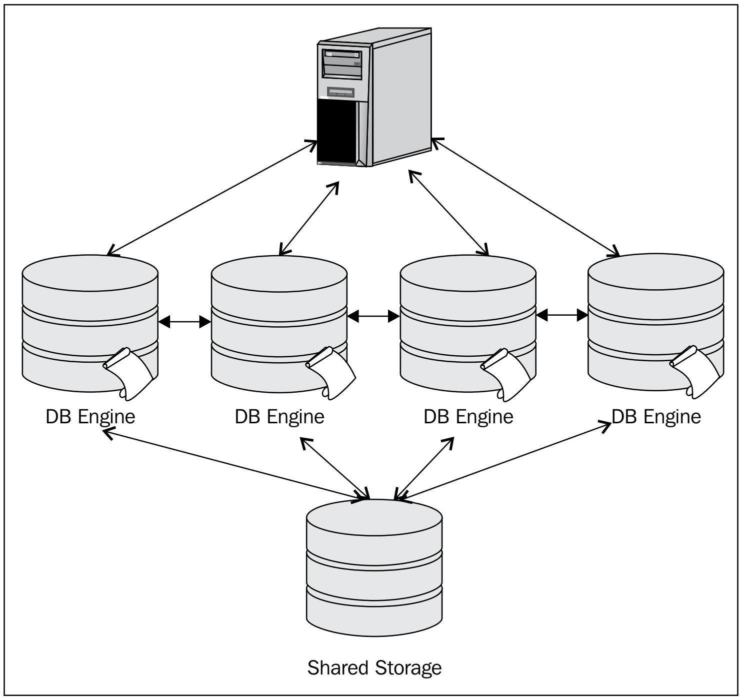
However, there are two major problems with this approach:

- There is a lag between a write to the master server and the replication. This means that your application could update a record on the master and immediately after that read the old, incorrect version of the same record from one of the slaves, which is often undesirable.
- You haven't really scaled out. Although you have given your master server some breathing room, you will eventually reach the point where it cannot handle all the writes. When that happens, you will be on your vendor's website again, configuring a bigger server.

Database clustering

The second approach to database scale-out is **database clustering**, often referred to as the **shared everything approach**. The best known example of a database that uses this strategy is Oracle RAC.

This approach allows you to configure many database instances that access a shared storage device:



In the previous architecture, every node in the cluster can handle both reads and writes, which can significantly improve throughput.

From the application perspective, nothing needs to change, at least in theory. Even the load balancing is automatic.

However, database clustering is not without its own set of problems:

- Database writes require synchronization of in-memory data structures such as caches and locks across all the nodes in the cluster. This increases the duration of write operations and introduces even more contention. In the worst-case scenario, you might even experience negative scalability as you add nodes to the cluster (meaning that as you add the nodes, you actually decrease the number of operations you can handle).

- It is difficult to set up and administer, and it requires an expensive SAN device for shared storage.
- Even read operations cannot be scaled indefinitely, because any shared disk system, no matter how powerful and expensive it is, will eventually reach its limit.

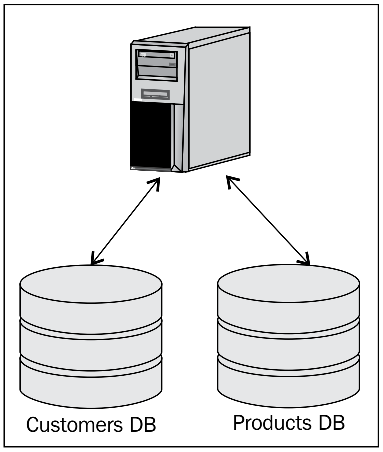
In general, database clustering might be a good solution for read-intensive usage scenarios, such as data warehousing and **BI** (**Business Intelligence**), but will likely not be able to scale past a certain point in write-intensive **OLTP** (**online transaction processing**) applications.

Database sharding

The basic idea behind **database sharding** is to partition a single large database into several smaller databases. It is also known as a **shared nothing approach**.

It is entirely up to you to decide how to actually perform partitioning. A good first step is to identify the groups of tables that belong together based on your application's querying needs. If you need to perform a join on two tables, they should belong to the same group. If you are running an e-commerce site, it is likely that you will end up with groups that represent your customer-related tables, your product catalog-related tables, and so on.

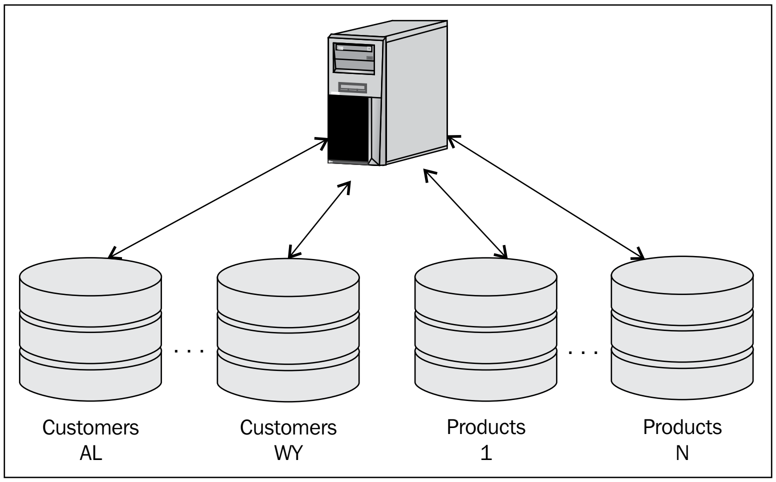
Once you identify table groups, you can move each group into a separate database, effectively partitioning the database by functional area. This approach is also called **vertical partitioning**, and is depicted in the following diagram:



PACKT PUBLISHING

Unfortunately, vertical partitioning by definition is limited by the number of functional areas you can identify, which imposes a hard limit on the number of shards you can create. Once you reach the capacity of any functional shard, you will either need to scale up or partition data horizontally.

This means that you need to create multiple databases with identical schemas, and split all the data across them. It is entirely up to you to decide how to split the data, and you can choose a different partitioning strategy for each table. For example, you can partition customers by state and products using modulo of the primary key:



Implemented properly, database sharding gives you virtually unlimited scalability, but just like the other two strategies it also has some major drawbacks:

- For one, sharding significantly complicates application code. Whenever you want to perform a database operation, you need to determine which shard the operation should execute against and obtain a database connection accordingly. While this logic can (and should) be encapsulated within the Data Access Layer, it adds complexity to the application nevertheless.
- You need to size your shards properly from the very beginning, because adding new shards later on is a major pain. Once your data partitioning algorithm and shards are in place, adding a new shard or a set of shards requires you not only to implement your partitioning algorithm again, but also to undertake the huge task of migrating the existing data to new partitions, which is an error-prone and time-consuming process.

PACKT PUBLISHING

- Queries and aggregations that used to be simple are not so simple anymore. Imagine your customers are partitioned by state and you want to retrieve all female customers younger than 30, and sort them by the total amount they spent on makeup within the last six months. You will have to perform a distributed query against all the shards and aggregate the results yourself. And I hope you kept track of those makeup sales within each customer row, or you might spend a few long nights trying to collect that information from a partitioned orders table.
- It is likely that you will have to denormalize your schema and/or replicate reference data to all the shards in order to eliminate the need for cross-shard joins. Unless the replicated data is read-only, you will have the same consistency issues as with master-slave setup when it gets updated.
- Cross-shard updates will require either distributed (XA) transactions that can significantly limit the scalability, or compensating transactions that can significantly increase application complexity. If you avoid distributed transactions and implement your own solution, you will also run into data consistency issues, as updates to one shard will be visible before you update the others and complete a logical transaction.
- Failure of any single shard will likely render the whole system unusable. Unfortunately, the probability that one shard will fail is directly proportional to the number of shards. This means that you will have to implement an **HA** (**High Availability**) solution for each individual shard, effectively doubling the amount of hardware and the cost of the data layer.

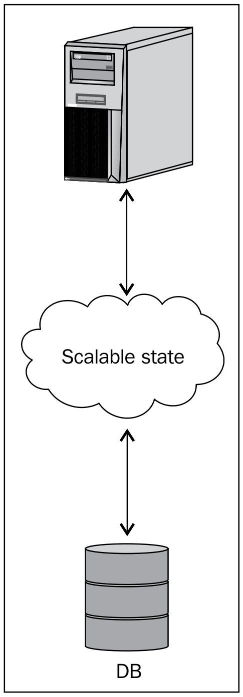
Even with all these drawbacks, sharding is the approach used by some of the largest websites in the world, from Facebook and Flickr, to Google and eBay. When the pain is great, any medicine that reduces it is good, regardless of the side effects.

In the next section we will look at the fourth option for database scaling—removing the need to scale it at all.

Return of the state

As I mentioned earlier, removal of the state from the application has a significant increase in database load as a consequence. This implies that there is a simple way to reduce the database load—put state back into the application.

Of course, we can't really put the state back into our **stateless** services, as that would make them **stateful** and prevent them from scaling out. However, nothing prevents us from introducing a new data management layer between our stateless application logic and the database. After all, as professor Bellovin said, "any software problem can be solved by adding another layer of indirection".



Ideally, this new layer should have the following characteristics:

- It should manage data as objects, because objects are what our application needs
- It should keep these objects in memory, in order to improve performance and avoid the disk I/O bottlenecks that plague databases
- It should be able to transparently load missing data from the persistent store behind it
- It should be able to propagate data modifications to the persistent store, possibly asynchronously
- It should be as simple to scale out as our stateless application layer

As you have probably guessed, Coherence satisfies all of these requirements, which makes it a perfect data management layer for scalable web applications.

PACKT PUBLISHING

Using Coherence to reduce database load

Many database queries in a typical application are nothing more than primary key-based lookups. Offloading only these lookups to an application-tier cache would significantly reduce database load and improve overall performance.

However, when your application-tier cache supports propagate queries and can also scale to support large datasets across many physical machines, you can offload even more work to it and let the database do what it does best—persist data and perform complex queries.

The company I mentioned earlier, which was at risk of going bankrupt because of the high cost of scaling up their database, saw database load drop more than 80% after they introduced Coherence into the architecture.

Furthermore, because they didn't have to block waiting for the response from the database for so long, their web servers were able to handle twice the load. In combination, this effectively doubled the capacity and ensured that no database server upgrades would be necessary in the near future.

This example is somewhat extreme, especially because in this case it literally meant the difference between closing the shop and being profitable, but it is not uncommon to see a 60% to 80% reduction in database load after Coherence is introduced into the architecture, accompanied with an increased capacity in the application layer as well.

Coherence and master-slave databases

Coherence effectively eliminates the need for master-slave replication, as it provides all of its benefits without any of the drawbacks.

Read-only slaves are effectively replaced with a distributed cache that is able to answer the vast majority of read operations. On the other hand, updates are performed against the cached data and written into the database by Coherence, so there is no replication lag and the view of the data is fully coherent at all times.

Coherence and database clusters

By significantly reducing the total load on the database, Coherence will give your database cluster some breathing space and allow you to handle more users, or to reduce the size of the cluster.

In the best-case scenario, it might eliminate the need for the database cluster altogether, allowing you to significantly simplify your architecture.

Coherence and database sharding

This is by far the most interesting scenario of the three. Just like with database clustering, you might be able to completely eliminate the need for sharding and simplify the architecture by using a single database.

However, if you do need to use shards, Coherence allows you to eliminate some of the drawbacks we discussed earlier.

For one, distributed queries and aggregations are built-in features, not something that you need to write yourself. Finding all female customers younger than 30 is a simple query against the customers cache. Similarly, aggregating orders to determine how much each of these customers spent on makeup becomes a relatively simple parallel aggregation against the orders cache.

Second, Coherence allows you to protect the application against individual shard failures. By storing all the data within the grid and configuring Coherence to propagate data changes to the database asynchronously, your application can survive one or more shards being down for a limited time.

Coherence and scalability

Coherence is an ideal solution for scalable data management. Scaling both capacity and throughput can be easily achieved by adding more nodes to the Coherence cluster.

However, it is entirely possible for an application to introduce contention and prevent scalability, for example, by using excessive locking. Because of this, care should be taken during application design to ensure that artificial bottlenecks are not introduced into the architecture.

Achieving high availability

The last thing we need to talk about is **availability**. At the most basic level, in order to make the application highly available we need to remove all single points of failure from the architecture. In order to do that, we need to treat every single component as unreliable and assume that it will fail sooner or later.

[PACKT]
PUBLISHING

It is important to realize that the availability of the system as a whole can be defined as the product of the availability of its tightly coupled components:

$AS = A1 * A2 * ...* An$

For example, if we have a web server, an application server, and a database server, each of which is available 99% of the time, the expected availability of the system as a whole is only 97%:

$0.99 * 0.99 * 0.99 = 0.970299 = 97\%$

This reflects the fact that if any of the three components should fail, the system as a whole will fail as well. By the way, if you think that 97% availability is not too bad, consider this: 97% availability implies that the system will be out of commission 11 days every year, or almost one day a month!

We can do two things to improve the situation:

- We can add redundancy to each component to improve its availability.
- We can decouple components in order to better isolate the rest of the system from the failure of an individual component.

The latter is typically achieved by introducing asynchrony into the system. For example, you can use messaging to decouple a credit card processing component from the main application flow—this will allow you to accept new orders even if the credit card processor is temporarily unavailable.

As mentioned earlier, Coherence is able to queue updates for a failed database and write them asynchronously when the database becomes available. This is another good example of using asynchrony to provide high availability.

Although the asynchronous operations are a great way to improve both availability and scalability of the application, as well as perceived performance, there is a limit to the number of tasks that can be performed asynchronously in a typical application. If the customer wants to see product information, you will have to retrieve the product from the data store, render the page, and send the response to the client synchronously.

To make synchronous operations highly available our only option is to make each component redundant.

Adding redundancy to the system

In order to explain how redundancy helps improve availability of a single component, we need to introduce another obligatory formula or two (I promise this is the only chapter you will see any formulas in):

$F = F1 * F2 * \ldots * Fn$

Where F is the likelihood of failure of a redundant set of components as a whole, and F1 through Fn are the likelihoods of failure of individual components, which can be expressed as:

$Fc = 1 – Ac$

Going back to our previous example, if the availability of a single server is 99%, the likelihood it will fail is 1%:

$Fc = 1 – 0.99 = 0.01$

If we make each layer in our architecture redundant by adding another server to it, we can calculate new availability for each component and the system as a whole:

$Ac = 1 - (0.01 * 0.01) = 1 – 0.0001 = 0.9999 = 99.99\%$

$As = 0.9999 * 0.9999 * 0.9999 = 0.9997 = 99.97\%$

Basically, by adding redundancy to each layer, we have reduced the application's downtime from 11 days to approximately two and a half hours per year, which is not nearly as bad.

Redundancy is not enough

Making components redundant is only the first step on the road to high availability. To get to the finish line, we also need to ensure that the system has enough capacity to handle the failure under the peak load.

Developers often assume that if an application uses scale-out architecture for the application tier and a clustered database for persistence, it is automatically highly available. Unfortunately, this is not the case.

If you determine during load testing that you need N servers to handle the peak load, and you would like the system to remain operational even if X servers fail at the same time, you need to provision the system with N+X servers. Otherwise, if the failure occurs during the peak period, the remaining servers will not be able to handle the incoming requests and either or both of the following will happen:

PACKT PUBLISHING

- The response time will increase significantly, making performance unacceptable
- Some users will receive "500 - Service Busy" errors from the web server

In either case, the application is essentially not available to the users.

To illustrate this, let's assume that we need five servers to handle the peak load. If we provision the system with only five servers and one of them fails, the system as a whole will fail. Essentially, by not provisioning excess capacity to allow for failure, we are turning "application will fail if all 5 servers fail" into "application will fail if any of the 5 servers fail". The difference is huge—in the former scenario, assuming 99% availability of individual servers, system availability is almost 100%. However, in the latter it is only 95%, which translates to more than 18 days of downtime per year.

Coherence and availability

Oracle Coherence provides an excellent foundation for highly available architecture. It is designed for availability; it assumes that any node can fail at any point in time and guards against the negative impact of such failures.

This is achieved by data redundancy within the cluster. Every object stored in the Coherence cache is automatically backed up to another node in the cluster. If a node fails, the backup copy is simply promoted to the primary and a new backup copy is created on another node.

This implies that updating an object in the cluster has some overhead in order to guarantee data consistency. The cache update is not considered successful until the backup copy is safely stored. However, unlike clustered databases that essentially lock the whole cluster to perform write operations, the cost of write operations with Coherence is constant regardless of the cluster size. This allows for exceptional scalability of both read and write operations and very high throughput.

However, as we discussed in the previous section, sizing Coherence clusters properly is extremely important. If the system is running at full capacity, failure of a single node could have a ripple effect. It would cause other nodes to run out of memory as they tried to fail over the data managed by the failed node, which would eventually bring the whole cluster down.

It is also important to understand that, although your Coherence cluster is highly available that doesn't automatically make your application as a whole highly available as well. You need to identify and remove the remaining single points of failure by making sure that your hardware devices such as load balancers, routers, and network switches are redundant, and that your database server is redundant as well. The good news is that if you use Coherence to scale the data tier and reduce the load on the database, making the database redundant will likely be much easier and cheaper.

As a side note, while there are many stories that can be used as a testament to Coherence's availability, including the one when the database server went down over the weekend without the application users noticing anything, my favorite is an anecdote about a recent communication between Coherence support team and a long-time Coherence user.

This particular customer has been using Coherence for almost 5 years. When a bug was discovered that affects the particular release they were using, the support team sent the patch and the installation instructions to the customer. They received a polite reply:

> *You have got to be kidding me!? We haven't had a second of downtime in the last 5 years. We are not touching a thing!*

Putting it all together

Coherence can take you a long way towards meeting your performance, scalability, and availability objectives, but that doesn't mean that you can simply add it to the system as an afterthought and expect that all your problems will be solved.

Design for performance and scalability

Achieving performance, scalability, and availability goals requires you to identify bottlenecks and single points of failure in the application and to eliminate them. Doing this requires careful consideration of the problems and evaluation of the alternatives.

Unfortunately, way too often developers have "one true way" of building applications, typically reflected in the choice of the same tool stack and a "carbon copy" architecture across their projects. While this approach might work for small, departmental applications, it will not work for large, high-load, mission-critical systems. I mean, I love Spring as much as the next guy, but I tend to use it more within the architecture than as the architecture.

PACKT PUBLISHING

Another problem that I see from time to time is that people tend to misinterpret the famous quote attributed to Donald Knuth:

> *Premature optimization is the root of all evil.*

Stated like that, this quote can be used as an excuse to avoid all kinds of hard work. The problem is that it is taken completely out of context. The full quote is:

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*

With that I agree, we should forget about **small** efficiencies. Trying to minimize the impact of a reflective method call when the method itself performs a database query or invokes a web service makes no sense.

However, the latency and scalability issues I described in this chapter are anything but small and need to be taken into account and addressed early on—addressing them later will vary from extremely hard to impossible. To quote Randy Shoup, distinguished architect at eBay:

> *Scalability is a prerequisite to functionality, a priority-0 requirement, if ever there was one.*

Ignorance is the root of all evil. Premature optimization is not even a close second.

Set performance goals at each level

In order to know when you are done, you need to set performance goals. Most often people set a performance goal such as "end-user response time of 2 seconds or less".

Even though this is a good start, it is not nearly enough, because it doesn't tell you how much time the application can spend where. What you need to do is define some more specific requirements, such as "end-user response time of 2 seconds or less and time-to-last-byte of 1 second or less".

Now we are getting somewhere. This tells us that we can spend one second on the backend, collecting data, executing business logic, rendering the response, and sending it to the browser. And one second on the frontend, downloading HTML, CSS, JavaScript files, images, and rendering the page.

Of course, you can (and probably should), break this down even further. Your request handler on the backend likely calls one or more services to retrieve the data it needs, and uses some kind of template engine to render the page. You could give 100ms to orchestration and rendering logic in your request handler (which is very generous, by the way), which leaves you with a performance goal of 900ms or less for individual services.

This forces you to think about performance issues at each level separately and brings performance bottlenecks to the surface. If you have a service that takes 2 seconds to execute, it will be immediately obvious that you won't be able to meet your objective of 900 milliseconds, and you will have to find a way to speed it up. Whether the solution is to cache intermediate results, parallelize execution of the service, invoke it asynchronously, or something else, more likely than not you will be able to solve the problem once you know that it exists, which brings us to the following topic.

Measure and monitor

> *Count what is countable, measure what is measurable, and what is not measurable, make measurable.*
>
> *– Galileo*

In order to know that the application is meeting its objectives, you need to measure. You need to measure how long each service takes to execute, page rendering time on the server, and page load time on the client.

You also need to measure the load a single server can handle, and how that number changes as you scale out. This will give you an idea of how well your application scales and will allow you to size the system properly. More importantly, it will allow you to predict the cost of scaling to support higher load levels.

During development, you can use a profiler such as YourKit (`www.yourkit.com`) to measure the performance of your server-side code and find hot spots that can be optimized. On the client, you can use FireBug (`getfirebug.com`), YSlow (`developer.yahoo.com/yslow`) and/or Page Speed (`code.google.com/p/page-speed`) to profile your web pages and determine opportunities for improvement.

For load testing, you can either use a commercial tool, such as HP LoadRunner, or an open source tool such as Apache JMeter (`jakarta.apache.org/jmeter`), The Grinder (`grinder.sourceforge.net`), or Pylot (`www.pylot.org`).

However, measuring performance and scalability should not be limited to development; you also need to monitor your application once it is deployed to the production environment. After all, the production system is the only one that will provide you with the realistic load and usage patterns.

In order to do that, you need to instrument your application and collect important performance metrics. You can do that using a standard management framework, such as JMX, or by using a tool such as ERMA (`erma.wikidot.com`), an open source instrumentation API developed at Orbitz.

When it comes to Coherence cluster monitoring, you can use JConsole to look at the statistics exposed through JMX, but more likely than not you will want to consider a commercial monitoring tool, such as Evident ClearStone Live (`www.evidentsoftware.com`) or SL RTView (`www.sl.com`).

In any case, the tools are many and you should choose the ones that best fit your needs. What is important is that you measure and monitor constantly and make course corrections if the measurements start to deviate.

Educate your team

It is important that everyone on the team understands performance and scalability goals, and the issues related to achieving them. While I tried to provide enough information in this chapter to set the stage for the remainder of the book, I barely scratched the surface.

If you want to learn more about the topics discussed in this chapter, I strongly recommend that you read Theo Schlossnagle's *Scalable Internet Architectures,* and Cal Henderson's *Building Scalable Websites*.

For client-side website optimization, *High Performance Websites: Essential Knowledge for Front-End Engineers* and *Even Faster Web Sites: Performance Best Practices for Web Developers* by Steve Souders provide some excellent advice.

Summary

In this chapter, we have discussed how to achieve performance, scalability, and availability, and how Coherence in particular can help.

In order to improve performance, you need to minimize both the latency and the amount of data that you move across the wire. Coherence features such as near caching and entry processors can help you achieve that.

Coherence really shines when it comes to improving the scalability of your application, as it allows you to scale the layer that is most difficult to scale—the data management layer. By adding Coherence to your architecture you can avoid the need to implement complex database scale-out strategies, such as database clustering or sharding.

To achieve high availability you need to remove single points of failure from the architecture. Although Coherence is highly available by design, you need to ensure that all other components are as well, by adding redundancy. Also, do not forget that you need to size the system properly and allow for failure. Otherwise, your "highly available" system might crash as soon as one of its components fails.

Finally, you have learned that performance, scalability, and availability cannot be added to the system as an afterthought, but need to be designed into the system from the very beginning, and measured and monitored both during development and in production.

PACKT
PUBLISHING