From Technologies to Solutions

Oracle Coherence 3.5

Create Internet-scale applications using Oracle's high-performance data grid

Foreword by Cameron Purdy,
Tangosol Founder, and Vice President of Development, Oracle Corporation

Aleksandar Seović

with Mark Falco
Patrick Peralta

PACKT PUBLISHING

Oracle Coherence 3.5

Create Internet-scale applications using Oracle's high-performance data grid

Aleksandar Seović

Mark Falco

Patrick Peralta

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Oracle Coherence 3.5

First published: April 2010

Production Reference: 1240310

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847196-12-5

`www.packtpub.com`

Cover Image by Vinayak Chittar (`vinayak.chittar@gmail.com`)

Credits

Authors
Aleksandar Seović
Mark Falco
Patrick Peralta

Reviewers
Rob Harrop
Jimmy Nilsson
Patrick Peralta
Steve Samuelson
Robert Varga

Acquisition Editor
James Lumsden

Development Editor
Dilip Venkatesh

Technical Editor
Arani Roy

Indexer
Rekha Nair

Editorial Team Leader
Mithun Sehgal

Project Team Leader
Lata Basantani

Project Coordinator
Srimoyee Ghoshal

Proofreader
Chris Smith

Graphics
Geetanjali Sawant

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

Foreword

There are a few timeless truths of software development that are near-universally accepted, and have become the basis for many a witty saying over the years. For starters, there's Zymurgy's First Law of Evolving Systems Dynamics, which states:

> *Once you open a can of worms, the only way to re-can them is to use a bigger can.*

And Weinberg's Second Law, which postulates that,

> *If builders built buildings the way that programmers wrote programs, the first woodpecker to come along would destroy civilization.*

There is true brilliance in this timeless wit, enjoyed and appreciated by generation after generation of software developers.

The largest set of challenges that the modern programmer faces, and thus the source of most of the wit that we as programmers revel in, revolves around the seemingly boundless growth of complexity. Hardware becomes more complex. Operating systems become more complex. Programming languages and APIs become more complex. And the applications that we build and evolve, become more and more complex.

The complexity of a system always seems to hover ever so slightly on the far side of manageable, just slightly over the edge of the cliff. And while our work reality is a world full of complexity—or perhaps because of that complexity—we gravitate toward the pristine and the simple. While our day-to-day lives may be focused on diagnosing failures in production systems, our guiding light is the concept of continuous availability. While we may have to manually correct data when things go wrong, our aspirations remain with data integrity and information reliability. While the complexity of the legacy applications that we manage forces us to adopt the most expensive means of adding capacity, our higher thoughts are focused on commodity scale-out and linear scalability. And while the complex, layered, and often twisted system designs result in hopelessly slow responses to user actions, we fundamentally believe that users should experience near-instant responses for almost any conceivable action they take.

In a word, we believe in the *ilities*.

Availability. Reliability. Scalability. Performance. These are attributes that we wish to endow each and every one of our systems with. If a system lacks continuous availability, its users will be directly impacted by failures within the system. If a system lacks information reliability, then users will never know if the information they are using can be trusted. If a system lacks scalability, its growing popularity will overwhelm and kill it—it will fail just as it begins to succeed! If a system lacks performance, it will inflict a dose of pain upon its users with each and every interaction.

We wish to achieve these *ilities* because we wish for our labors to be beneficial to others, and we hope that the value that we provide through these systems endures far longer than the passing whims and fads of technology and industry.

Perhaps no greater revolution has occurred in our industry than the World Wide Web. Suddenly, the systems we provide had a limitless audience, with instant access to the latest and greatest versions of our software. Users are so accustomed to instant responses from one application that failure to achieve the same will cause them to quickly abandon another. Downtime no longer represents an inconvenience—for major websites, their online foibles have become headline news on the printed pages of the Wall Street Journal!

At the same time, the competitive landscape has forced companies, and thus their IT departments, to act and react far more quickly than before. The instant popularity of a particular good, service, or website can bring mind-boggling hordes of unexpected—though generally not undesired—users. Companies must be able to roll out new features and capabilities quickly, to grow their capacity dynamically in order to match the increase in users, and to provide instantaneous responsiveness with correct and up-to-date information to each and every user.

These are the systems that Oracle Coherence was designed to enable. These are the systems that this book will help you build.

If there was only one piece of advice that I could instill into the mind of a software architect or developer responsible for one of these systems, it would be this: architecture matters, and in systems of scale and systems that require availability, architecture matters absolutely! Failure to achieve a solid architecture will doom in advance any hope of significant scalability, and will leave the effects of failure within the system to pure chance.

No amount of brilliant programming can make up for a lack of architectural foresight. Systems do not remain available by accident, nor do they scale by accident. Achieving information reliability in a system that remains continuously available and provides high performance under varying degrees of load and scale is an outcome that results only when a systematic and well-conceived architecture has been laid down. Availability, reliability, scalability, and performance must be the core tenets of an architecture, and they must be baked into and throughout that architecture.

If there were a second piece of advice that I could confer, it would be this: as a craftsman or craftswoman, know your tools, and know them well. Using Oracle Coherence as part of a system does not ensure any of the *ilities* by itself; it is simply a powerful tool for simultaneously achieving those *ilities* as part of a great architecture. This book is an effort to condense a huge amount of experience and knowledge into a medium of transfer that you can rehydrate into instant knowledge for yourself.

And the last piece of advice is this: don't believe it until you see it; make sure that you push it until it fails. While testing, if you don't overload the system until it breaks, then you can't be certain that it will work. If you don't pull the plug while it's running, then you can't be certain that it will handle failure when it truly matters. Don't be satisfied until you understand the limits of your systems, and until you appreciate and understand what lies beyond those boundaries.

A word about the author

I first met Aleks Seović in 2005. I was attending the Javapolis (now Devoxx) conference in Antwerp with the express purpose of persuading Aleks to create the .NET implementation of Coherence. I had known of him through his work in creating the Spring.NET framework, and knew that there was only one person whom I wanted to lead the creation of our own product for .NET. As they say, the rest is history: We hit it off smashingly, and found a great deal of common ground in our experiences with enterprise systems, the challenges of distributed computing, architecting for scalable performance and high availability, and the need for seamless and reliable information exchange between Java and .NET applications.

Aleks has such a great ability to understand complex systems, and such a compelling manner of reducing complexity into simple concepts, that I was ecstatic when he told me that he was writing this book. Starting a book is no challenge at all, but *finishing* a book is a great feat. Many years of work have gone into these pages. May you enjoy and profit from this book as deeply as I have enjoyed and profited from my conversations with Aleks over these past years.

Cameron Purdy
Lexington, MA
January 2010

About the author

Aleksandar Seović is the founder of and managing director at S4HC, Inc., where he leads professional services practice. He works with customers throughout the world to help them solve performance and scalability puzzles and implement innovative solutions to complex business and technical problems.

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 an Oracle Coherence data grid. He was also one of the key people involved in the design and implementation of Portable Object Format (POF), a platform-independent object serialization format that allows seamless interoperability of Coherence-based Java, .NET, and C++ applications.

Aleksandar is Oracle ACE Director for Fusion Middleware, and frequently speaks about and evangelizes Coherence at conferences, Java and .NET user group events, and Coherence SIGs. He blogs about Coherence and related topics at `http://coherence.seovic.com`.

Acknowledgements

First and foremost, I'd like to thank Cameron Purdy and Alex Gleyzer for giving me the opportunity to work on Coherence. It has been quite a journey—I still remember the day when we got a .NET application to connect to the cluster for the first time. Guys, it has been a privilege working with you all these years to make a great product even better.

I've heard many times that a book is never a single person's creation. Only now, after I have written one myself, I truly understand what that means.

I cannot thank enough my co-authors, Mark Falco and Patrick Peralta, members of the Coherence engineering team, who contributed two great chapters to the book. Patrick's chapter covers pretty much everything you need to know in order to integrate Coherence with persistent data sources. As for Mark's chapter on using the Coherence C++ client, well, let's just say that I would've had a *really* hard time writing that one myself ☺.

The sample application for the book is the result of the hard work of my colleagues from Solutions for Human Capital: Ivan Cikić, Nenad Dobrilović, Marko Dumić, and Aleksandar Jević. They managed to deliver (once again) on a very tight schedule, and I am very proud of the final result. Thank you for all the help and for putting up with my last minute change requests—I know it wasn't easy at times, but you have done a wonderful job.

I can honestly say (and everyone who saw the first draft is my witness) that this book was significantly improved based on the feedback from many reviewers. The official reviewers for the book were Rob Harrop, Jimmy Nilsson, Patrick Peralta, Steve Samuelson, and Robert Varga, but Mark Falco, Cameron Purdy, Cristobal Soto, Phil Wheeler, and Andrew Wilson also provided invaluable feedback. Jimmy, it is mostly your "fault" that there is a sample application now demonstrating (among other things) how to build a clean, testable domain model that works well with Coherence.

Many members of Coherence engineering and Oracle's Architecture team were only an e-mail or phone call away when I had questions: Noah Arliss, Simon Bisson, Gene Gleyzer, Jason Howes, Rob Misek, Andy Nguyen, Brian Oliver, Jon Purdy, and Randy Stafford all helped at one point or another, whether they know it or not. Thank you for that, and thank you for building such a great product.

James Lumsden, Dilip Venkatesh, Rajashree Hamine, and Srimoyee Ghoshal from Packt Publishing provided necessary support when I needed it and helped me reach the finish line. James, I know the journey was much longer than either of us expected. Thank you for believing in me even when I didn't.

My partners at Solutions for Human Capital, Snejana Sevak, Aleksandar Jević, and Nebojša Peruničić, helped more than they realize by shielding me from the daily disturbances of running a business and allowing me to focus my energy on writing, and I thank them for that.

I would also like to thank all the clients and colleagues I worked with over the years. I learned something new from each one of you, and for that I am grateful.

Most importantly, I would like to thank my family. My wife Marija supported me from the moment I decided to work on this book, even though she knew that my already busy schedule would only get busier because of it. People who think astronauts' wives have it rough have never met the wife of a traveling geek who decides to write a book. Хвала, љубави.

To my children, Ana Maria and Novak, I am sorry I couldn't play with you as much as we all wanted, and promise to make it up to you. You are too young to understand why daddy was sleeping during the day and working at night, but that doesn't matter now. Daddy is back and loves you both very much!

About the co-authors

Mark Falco is a Consulting Member of Technical Staff at Oracle. He has been part of the Coherence development team since 2005 where he has specialized in the areas of clustered communication protocols as well as the Coherence for C++ object model. Mark holds a B.S. in computer science from Stevens Institute of Technology.

I would like to thank Aleks for the opportunity to contribute to this book and Tangosol for the years of fun and interesting work. Thank you Otti, Erika, and Mia for your encouragement and support.

Patrick Peralta is a Senior Software Engineer for Oracle (formerly Tangosol) specializing in Coherence and middleware Java. He wears many hats in Coherence engineering, including development, training, documentation, and support. He has extensive experience in supporting and consulting customers in fields such as retail, hospitality, and finance.

As an active member of the Java developer community he has spoken at user groups and conferences across the US including Spring One and Oracle Open World. Prior to joining Oracle, Patrick was a senior developer at Symantec, working on Java/J2EE based services, web applications, system integrations, and Swing desktop clients. Patrick has a B.S. in computer science from Stetson University in Florida.

He currently maintains a blog on Coherence and other software development topics at `http://blackbeanbag.net`.

I would like to express my appreciation and gratitude to those that provided valuable feedback, including Aleks Seović, Gene Gleyzer, Andy Nguyen, Pas Apicella, and Shaun Smith. Many thanks as well to my family, including my parents, siblings, and especially my wonderful wife Maria and son Isaac for providing me with joy and perspective on what is truly important in life.

About the reviewers

Rob Harrop is a respected speaker, author, entrepreneur, and technologist.

As Lead Engineer of SpringSource dm Server, Rob is driving SpringSource's enterprise middleware product line and ensuring that the company continues to deliver high-performance, highly scalable enterprise solutions.

With a thorough knowledge of both Java and .NET, Rob has successfully deployed projects across both platforms. He has extensive experience across a variety of sectors, in particular banking, retail, and government. Prior to joining SpringSource, he co-founded the UK-based software company Cake Solutions Limited and worked as a Lead Developer for a successful dotcom start-up.

Rob is the author of five books, including *Pro Spring*, a widely acclaimed, comprehensive resource on the Spring Framework.

Jimmy Nilsson has been working as a developer/architect for over 20 years.

He has authored *Applying Domain-Driven Design and Patterns* and *.NET Enterprise Design*.

Steve Samuelson has worked in IT for over 20 years across various industries including home building, finance, and education. Although experienced with Windows deployment and hardware, Steve prefers custom software development. Currently, Steve is the Chief Architect for an international education provider where he works with multiple technologies running under Unix and Windows. Steve's primary interest lies in Microsoft .NET development and tools, but he makes sure to keep up on emerging Java and Oracle technologies among others.

Robert Varga is a Lead Software Engineer at EPAM Systems. He has worked in various roles from Developer to Enterprise Architect on several large projects for various customers, mostly in the areas of insurance, online betting, online auctions, and finance. He is also an Oracle ACE since 2008.

Robert has worked on Java-based enterprise systems since 1998 and on various projects with the Coherence data grid since 2005. He is among the most active contributors to the Oracle Coherence support forums helping developers with questions about Oracle Coherence.

Table of Contents

Preface

As an architect of a large, mission-critical website or enterprise application, you need to address at least three major non-functional requirements: *performance*, *scalability*, and *availability*.

Performance is defined as the amount of time that an operation takes to complete. In a web application, it is usually measured as "time to last byte" (**TTLB**)—the amount of time elapsed from the moment the web server received a request, until the moment the last byte of response has been sent back to the client. Performance is extremely important, because experience has shown us that no matter how great and full-featured an application is, if it is slow and unresponsive, the users will hate it.

Scalability is the ability of the system to maintain acceptable performance as the load increases, or to support additional load by adding hardware resources. While it is relatively simple to make an application perform well in a single-user environment, it is significantly more difficult to maintain that level of performance as the number of simultaneous users increases to thousands, or in the case of very large public websites, to tens or even hundreds of thousands. The bottom line is, if your application doesn't scale well, its performance will degrade as the load increases and the users will hate it.

Finally, **availability** is measured as the percentage of time an application is available to the users. While some applications can crash several times a day without causing major inconvenience to the user, most mission-critical applications simply cannot afford that luxury and need to be available 24 hours a day, every day. If your application is mission critical, you need to ensure that it is highly available or the users will hate it. To make things even worse, if you build an e-commerce website that crashes during the holiday season, your investors will hate you as well.

The moral of the story is that in order to keep your users happy and avoid all that hatred, you as an architect need to ensure that your application is fast, remains fast even under heavy load, and stays up and running even when the hardware or software components that it depends on fail. Unfortunately, while it is relatively easy to satisfy any one of these three requirements individually and not too difficult to comply with any two of them, it is considerably more difficult to fulfill all three at the same time.

Introducing Oracle Coherence

Over the last few years, **In-Memory Data Grids** have become an increasingly popular way to solve many of the problems related to performance and scalability, while improving availability of the system at the same time.

Oracle Coherence is an In-Memory Data Grid that allows you to eliminate *single points of failure* and *single points of bottleneck* in your application by distributing your application's objects and related processing across multiple physical servers.

There are several important points in the definition above:

- Coherence manages *application objects*, which are ready for use within the application. This eliminates the need for repeated, and often expensive, loading and transformation of the raw data into objects.
- Coherence distributes application objects *across many physical servers* while ensuring that a coherent, **Single System Image** (**SSI**) is presented to the application.
- Coherence ensures that *no data or in-flight operations are lost* by assuming that any node could fail at any time and by ensuring that every piece of information is stored in multiple places.
- Coherence *stores data in memory* in order to achieve very high performance and low latency for data access.
- Coherence allows you to distribute not only application objects, but also *the processing* that should be performed on these objects. This can help you eliminate single points of bottleneck.

The following sections provide a high-level overview of Coherence features; the remainder of the book will teach you "how", and more importantly, "when" to use them.

Distributed caching

One of the easiest ways to improve application performance is to bring data closer to the application, and keep it in a format that the application can consume more easily.

Most enterprise applications are written in one of the object-oriented languages, such as Java or C#, while most data is stored in relational databases, such as Oracle, MySQL or SQL Server. This means that in order to use the data, the application needs to load it from the database and convert it into objects. Because of the impedance mismatch between tabular data in the database and objects in memory, this conversion process is not always simple and introduces some overhead, even when sophisticated O-R mapping tools, such as Hibernate or EclipseLink are used.

Caching objects in the application tier minimizes this performance overhead by avoiding unnecessary trips to the database and data conversion. This is why all production-quality O-R mapping tools cache objects internally and short-circuit object lookups by returning cached instances instead, whenever possible.

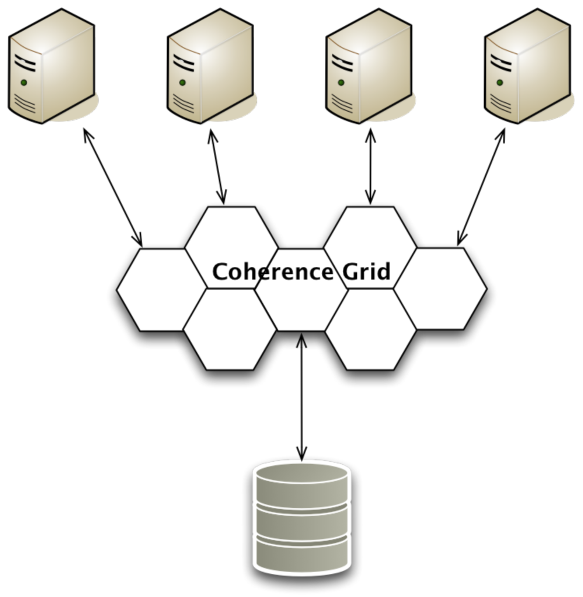
However, when you scale out your application across multiple servers, you will start running into cache synchronization issues. Each server will cache its own copy of the data, and will have no way of knowing if that same data has been changed on another server—in this case, the locally cached copy should be invalidated and evicted from the cache.

Oracle Coherence solves this problem by allowing you to distribute your cache across a cluster of machines, while providing a unified, fully coherent view of the data. This means that you can configure Coherence as an L2 cache for Hibernate or EclipseLink, and forget about distributed cache synchronization!

If this was all Coherence did, it would be impressive enough. However, it actually does so much more that I don't recommend using it purely as an L2 cache, unless you have an existing application that you need to scale out. While Coherence works like a charm as an L2 cache behind an O-R mapper, this architecture barely scratches the surface of what Coherence can do. It is like "killing an ox for a pound of meat", as the Serbian proverb says.

It is much more powerful to use Coherence as a logical persistence layer of your application, which sits between the application logic and the physical data store. Whenever the application needs data, it asks Coherence for it. If the data is not already in the cache, Coherence will transparently load it from the data store, cache it, and return it to the application. Similarly, when the application needs to store data, it simply puts objects into the cache, and Coherence updates the underlying data store automatically.

This architecture is depicted in the following diagram and is the basis for the architecture we will use throughout the book:



Although Coherence is not really a *persistent* store in the preceding scenario, the fact that the application *thinks* that it is decouples the application from the data store and enables you to achieve very high scalability and availability. You can even configure Coherence so the application will be isolated from a complete data store failure.

Distributed queries

Having all the data in the world is meaningless unless there is a way to find the information you need, when you need it. One of the many advantages of In-Memory Data Grids over clustered caches, such as Memcached, is the ability to find data not just by the primary key, but also by executing queries and aggregations against the cache.

Coherence is no exception—it allows you to execute queries and aggregations in parallel, across all the nodes in the cluster. This allows for the efficient processing of large data sets within the grid and enables you to improve aggregation and query performance by simply adding more nodes to the cluster.

In-place and parallel processing

In many situations, you can improve performance enormously if you perform the processing where the data is stored, instead of retrieving the data that needs to be processed. For example, while working with a relational database, you can use bulk update or a stored procedure to update many records without moving any data across the network.

Coherence allows you to achieve the same thing. Instead of retrieving the whole dataset that needs to be processed and iterating over it on a single machine, you can create an *entry processor*—a class that encapsulates the logic you want to execute for each object in a target dataset. You can then submit an instance of the processor into the cluster, and it will be executed locally on each node. By doing so, you eliminate the need to move a large amount of data across the network. The entry processor itself is typically very small and allows processing to occur in parallel.

The performance benefit of this approach is tremendous. Entry processors, just like distributed queries, execute in parallel across grid nodes. This allows you to improve performance by simply spreading your data across more nodes.

Coherence also provides a grid-enabled implementation of CommonJ Work Manager, which is the basis for JSR-237. This allows you to submit a collection of work items that Coherence will execute "in parallel" across the grid. Again, the more nodes you have in the grid, the more work items can be executed in parallel, thereby improving the overall performance.

Cache events

In many applications, it is useful to know when a particular piece of data changes. For example, you might need to update a stock price on the screen as it changes, or alert the user if a new workflow task is assigned to them.

The easiest and the most common solution is to periodically poll the server to see if the information on the client needs to be updated. This is essentially what Outlook does when it checks for new e-mail on the POP3 mail server, and you (the user) control how often the polling should happen.

The problem with polling is that the more frequently it occurs, the more load it puts on the server, decreasing its scalability, even if there is no new information to be retrieved.

On the other hand, if the server knows which information you are interested in, it can push that information to you. This is how Outlook works with Exchange Server—when the new mail arrives, the Exchange Server notifies Outlook about this event, and Outlook displays the new message in your inbox.

Coherence allows you to register interest in a specific cache, a specific item, or even a specific subset of the data within the cache using a query. You can specify if you are interested in cache insertions, updates or deletions only, as well as whether you would like to receive the old and the new cache value with the event.

As the events occur in the cluster, your application is notified and can take the appropriate action, without the need to poll the server.

Coherence within the Oracle ecosystem

If you look at Oracle marketing material, you will find out that Coherence is a member of the Oracle Fusion Middleware product suite. However, if you dig a bit deeper, you will find out that it is not just another product in the suite, but a foundation for some of the high-profile initiatives that have been announced by Oracle, such as Oracle WebLogic Application Grid and Complex Event Processing.

Coherence is also the underpinning of the "SOA grid"—a next-generation SOA platform that David Chappell, vice president and chief technologist for SOA at Oracle, wrote about for *The SOA Magazine* [SOAGrid1&2].

I believe that over the next few years, we will see Coherence being used more and more as an enabling technology within various Oracle products, because it provides an excellent foundation for fast, scalable, and highly-available solutions.

Coherence usage scenarios

There are many possible uses for Coherence, some more conventional than the others.

It is commonly used as a mechanism to off-load expensive, difficult-to-scale backend systems, such as databases and mainframes. By fronting these systems with Coherence, you can significantly improve performance and reduce the cost of data access.

Another common usage scenario is **eXtreme Transaction Processing** (**XTP**). Because of the way Coherence partitions data across the cluster, you can easily achieve throughput of several thousand transactions per second. What's even better is that you can scale the system to support an increasing load by simply adding new nodes to the cluster.

As it stores all the data in memory and allows you to process it in-place and in parallel, Coherence can also be used as a computational grid. In one such application, a customer was able to reduce the time it took to perform risk calculation from eighteen hours to twenty minutes.

Coherence is also a great integration platform. It allows you to load data from multiple data sources (including databases, mainframes, web services, ERP, CRM, DMS, or any other enterprise system), providing a uniform data access interface to client applications at the same time.

Finally, it is an excellent foundation for applications using the Event Driven Architecture, and can be easily integrated with messaging, ESB, and **Complex Event Processing** (**CEP**) systems.

That said, for the remainder of the book I will use the "conventional" web application architecture described earlier, to illustrate Coherence features—primarily because most developers are already familiar with it and also because it will make the text much easier to follow.

Oracle Coherence editions

Coherence has three different editions—Standard, Enterprise, and Grid Editions. As is usually the case, each of these editions has a different price point and feature set, so you should evaluate your needs carefully before buying.

The Coherence client also has two different editions—Data Client and Real-Time Client. However, for the most part, the client edition is determined by the server license you purchase—Standard and Enterprise Edition give you a Data Client license, whereas the Grid Edition gives you a Real-Time Client license.

A high-level overview of edition differences can be found at `http://www.oracle.com/technology/products/coherence/coherencedatagrid/coherence_editions.html`, but you are likely to find the following documents available in the Coherence Knowledge Base much more useful:

- *The Coherence Ecosystem*, available at `http://coherence.oracle.com/display/COH35UG/The+Coherence+Ecosystem`
- *Coherence Features by Edition*, available at `http://coherence.oracle.com/display/COH35UG/Coherence+Features+by+Edition`

Throughout the book, I will assume that you are using the Grid Edition and Real-Time Client Edition, which provide access to all Coherence features.

The important thing to note is that when you go to Oracle's website to download Coherence for evaluation, you will find only one download package for Java, one for .NET, and one for each supported C++ platform. This is because all the editions are included into the same binary distribution; choosing the edition you want to use is simply a matter of obtaining the appropriate license and specifying the edition in the configuration file.

By default, Grid Edition features are enabled on the server and Real-Time Client features on the client, which is exactly what you will need in order to run the examples given in the book.

What this book covers

Chapter 1, Achieving Performance, Scalability, and Availability Objectives discusses obstacles to scalability, performance, and availability and also some common approaches that are used to overcome these obstacles. It also talks about how these solutions can be improved using Coherence.

Chapter 2, Getting Started teaches you how set up Coherence correctly in a development environment, and the basics of how to access Coherence caches, both by using the supplied command-line client and programmatically.

Chapter 3, Planning Your Caches covers various cache topologies supported by Coherence and provides guidance on when to use each one and how to configure them.

Chapter 4, Implementing Domain Objects introduces the sample application we will be building throughout the book and shows you how to design your domain objects to take full advantage of Coherence.

Chapter 5, Querying the Data Grid teaches you how to use Coherence queries and aggregators to retrieve data from the cache in parallel.

Chapter 6, Parallel and In-Place Processing covers Coherence features that allow you to perform in-place or parallel processing within a data grid.

Chapter 7, Processing Data Grid Events shows you how to use powerful event mechanisms provided by Coherence.

Chapter 8, Implementing Persistence Layer discusses options for integration with various data repositories, including relational databases.

Chapter 9, Bridging Platform and Network Boundaries covers the Coherence*Extend protocol, which allows you to access a Coherence cluster from remote clients and from platforms and languages other than Java, such as .NET and C++.

Chapters 10, Accessing Coherence from .NET and *Chapter 11, Accessing Coherence from C++* teach you how to access Coherence from .NET and C++ clients, respectively.

Chapter 12, The Right Tool for the Job, provides some parting thoughts and reiterates practices you should apply when building scalable applications.

Appendix, Coherent Bank Sample Application, describes how to set up the sample application that accompanies the book in your environment.

The main goal of this book is to provide the missing information that puts various Coherence features into context and teaches you when to use them. As such, it does not cover every nook and cranny Coherence has to offer, and you are encouraged to refer to the Coherence product documentation [CohDoc] for details.

On the other hand, real-world applications are not developed using a single technology, no matter how powerful that one technology is. While the main focus of the book is Coherence, it will also discuss how Coherence fits into the overall application architecture, and show you how to integrate Coherence with some popular open source frameworks and tools.

You are encouraged to read this book in the order it was written, as the material in each chapter builds on the topics that were previously discussed.

What you need for this book

In addition to some spare time, an open mind, and a desire to learn, you will need to have Java SDK 1.5 or higher in order to run Coherence and the examples given in this book. While Coherence itself will run just fine on Java 1.4, the examples use some features that are only available in Java 1.5 or higher, such as enums and generics.

To run .NET examples from Chapter 10, you will need .NET Framework 3.5 and Visual Studio 2008. Although you can access Coherence using .NET Framework 1.1 and higher, the examples use features such as generics and Windows Presentation Foundation, which are only available in the more recent releases of the .NET Framework.

Finally, to run the C++ examples from Chapter 11, you need an appropriate version of the C++ compiler and related tools depending on your platform (for details check `http://download.oracle.com/docs/cd/E14526_01/coh.350/e14513/cpprequire.htm#BABDCDFG`), a fast machine to compile and link examples on, and a lot of patience!

Who this book is for

The primary audience for this book is experienced architects and developers who are interested in, or responsible for, the design and implementation of scalable, high-performance systems using Oracle Coherence.

However, Coherence has features that make it useful even in smaller applications, such as applications based on Event Driven Architecture, or Service Oriented Applications that would benefit from the high-performance, platform-independent binary protocol built into Coherence.

Finally, this book should be an interesting read for anyone who wants to learn more about the implementation of scalable systems in general, and how Oracle Coherence can be used to remove much of the pain associated with the endeavor.

Who this book is not for

This book is not for a beginner looking forward to learning how to write computer software. While I will try to introduce the concepts in a logical order and provide background information where necessary, for the most part I will assume that you, the reader, are an experienced software development professional with a solid knowledge of object-oriented design, Java, and XML.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "As a matter of fact, such a class already exists within `coherence.jar`, and is called `AbstractEvolvable`".

A block of code is set as follows:

```
public interface QueryMap extends Map {
  Set keySet(Filter filter);
  Set entrySet(Filter filter);
  Set entrySet(Filter filter, Comparator comparator);
  ...
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Filter filter = new BetweenFilter(
                        new PropertyExtractor("time"),
                        from, to);
```

Any command-line input or output is written as follows:

```
$ . bin/multicast-test.sh -ttl 0
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **OK** button finishes the installation".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail to `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

> **Downloading the example code for the book**
>
> Visit `http://www.packtpub.com/files/code/6125_Code.zip` to directly download the example code.
>
> The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Achieving Performance, Scalability, and Availability Objectives

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.

I 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.

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**.

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.

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.

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();
  }
}
```

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.

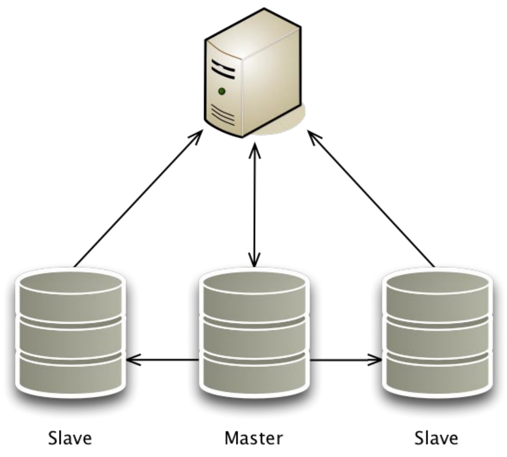
So if scaling up is not an answer, how do we scale the database out?

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.

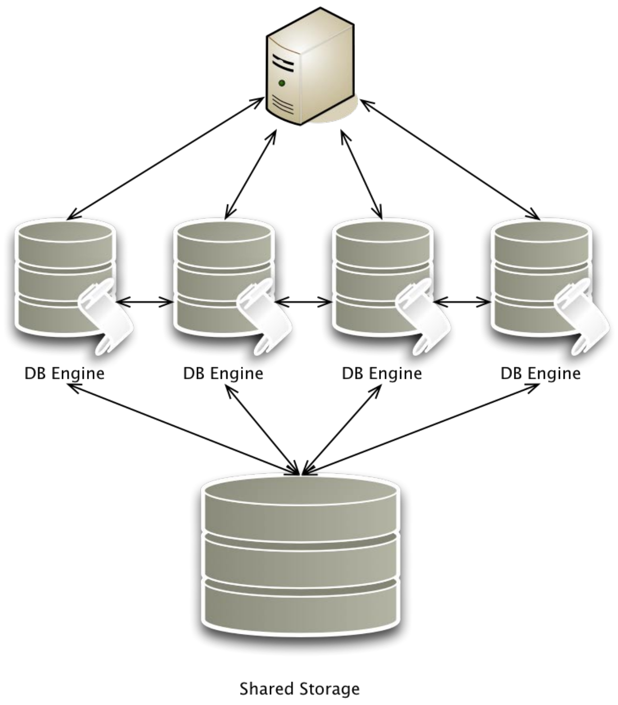
However, there are two major problems with this approach:

- 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).