O'REILLY®

Compliments of Pivotal

Pivotal GemFire® — Powered by Apache Geode®

Scaling Data Services with Pivotal GemFire®

Getting Started with In-Memory Data Grids

Mike Stolz

Pivotal GemFire®

In-Memory Data Grid

Powered by Apache® Geode™

Fast

Speed access to data from your applications, especially for data in slower, more expensive databases.

Scalable

Continually meet demand by elastically scaling your application's data layer.

Available

Improve resilience to potential server and network failures with high availability.

Event-Driven

Provide real-time notifications to applications through a pub-sub mechanism, when data changes.

Learn more at pivotal.io/pivotal-gemfire
Download open source Apache Geode at geode.apache.org
Try GemFire on AWS at aws.amazon.com/marketplace

Pivotal®

Scaling Data Services with Pivotal GemFire®

Getting Started with In-Memory Data Grids

Mike Stolz

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Scaling Data Services with Pivotal GemFire®
by Mike Stolz

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Susan Conant and Jeff Bleiel
Production Editor: Justin Billing
Copyeditor: Octal Publishing, Inc.
Proofreader: Charles Roumeliotis

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

December 2017: First Edition

Revision History for the First Edition
2017-11-27: First Release

978-1-492-02755-3

[LSI]

Table of Contents

Foreword

In *Super Mario Bros*., a popular Nintendo video game from the 1980s, you can run faster and jump higher after catching a hidden star. With modern software systems, development teams are finding new kinds of star power: cloud servers, streaming data, and reactive architectures are just a few examples.

Could GemFire be the powerful star for your mission-critical, real-time, data-centric apps? Absolutely, yes! This book reveals how to upgrade your performance game without the head-bumping headaches.

More cloud, cloud, cloud, and more data, data, data. Sound familiar? Modern applications change how we combine cloud infrastructure with multiple data sources. We're heading toward real-time, data-rich, and event-driven architectures. For these apps, GemFire fills an important place between relational and single-node key–value databases. Its mature production history is attractive to organizations that need mature production solutions.

At Southwest Airlines, GemFire integrates schedule information from more than a dozen systems, such as passenger, airport, crew, flight, gate, cargo, and maintenance systems. As these messages flow into GemFire, we update real-time web UIs (at more than 100 locations) and empower an innovative set of decision optimization tools. Every day, our ability to make better flight schedule decisions benefits more than 500,000 Southwest Airlines customers. With our event-driven software patterns, data integration concepts, and distributed systems foundation (no eggs in a single basket), we're well positioned for many years of growth.

Is GemFire the best fit for all types of application problems? Nope. If your use case doesn't have real-time, high-performance requirements, or a reasonably constrained data window, there are probably better choices. One size does not fit all. Just like trying to store everything in an enterprise data warehouse isn't the best idea, the same applies for GemFire, too.

Here's an important safety tip. GemFire by itself is lonely. It needs the right software patterns around it. Without changing how you write your software, GemFire is far less powerful and probably even painful. Well-meaning development teams might gravitate back toward their familiar relational worldview. If you see teams attempting to join regions just like a relational database, remind them to watch the *Wizard of Oz*. With GemFire, you aren't in Kansas anymore! From my experience, when teams say, "GemFire hurts," it's usually related to an application software issue. It's easy to miss a nonindexed query in development, but at production scale it's a different story.

Event-driven or reactive software patterns are a perfect fit with GemFire. To learn more, the Spring Framework website is an excellent resource. It contains helpful documentation about noSQL data, cloud-native, reactive, and streaming technologies.

It's an exciting time for the Apache Geode community. I've enjoyed meeting new "friends-of-data" both within and outside of Southwest. I hope you'll build your Geode and distributed software friend network. Learning new skills is a two-way street. It won't be long before you're helping others solve new kinds of challenging problems.

When you combine GemFire with the right software patterns, right problems to solve, and an empowered software team, it's fun to deliver innovative results!

— Brian Dunlap
Solution Architect,
Operational Data
Southwest Airlines

Preface

Why Are We Writing This Book?

When Pivotal committed to an open source strategy for its products, we donated the code base for GemFire as Apache Geode. This means that Pivotal GemFire and Apache Geode are essentially the same product. In writing this book, we'll try to use GemFire, but we also sometimes use Geode.

We also decided that our products should have more information than is provided in the standard documentation, and we wanted to introduce GemFire to a wider audience. We're not unique in this thinking. Many other Apache Software Foundation projects have books, often published by O'Reilly Media.

Who Are "We"?

Wes Williams and Charlie Black, both GemFire gurus, proposed the idea of a GemFire/Geode book and outlined their ideas for the content. Mike Stolz, the GemFire product lead, contributed most of the material and edited much of the rest. Others contributed material, as well, and their names are listed in the upcoming Acknowledgments section and in the chapter for which they have written extensively.

Who Is the Audience?

This book is primarily aimed at Java developers, especially those who require lightning quick response times in their applications. Microservice application developers who could benefit from a cache for storage would also find this book useful, especially the chapter

on Pivotal Cloud Cache. You can profit from this book if you have no previous experience with in-memory data grids, GemFire, or Apache Geode. We also wrote this book so that IT managers can obtain a sound high-level understanding of how they can employ GemFire in their environments.

Acknowledgments

Mike Stolz is the primary author and deserves most of the credit.

We would also like to acknowledge the following contributors:

- Wes Williams and Charlie Black for their many contributions
- John Guthrie for the section on Spring Data GemFire
- Greg Green for sections on getting started and Lucene integrations
- Brian Dunlap for the Foreword
- Jacque Istok for prodding us to write the book
- Jagdish Mirani for the section on Pivotal Cloud Cache
- Swapnil Bawaskar for the section on security
- John Knapp for the section on the Greenplum-Gemfire Connector
- Jeff Bleiel, our editor at O'Reilly, for his many useful suggestions for improving this book
- Marshall Presser for providing internal editing and project management for the book

CHAPTER 1

Introduction to Pivotal GemFire In-Memory Data Grid and Apache Geode

Wes Williams, Mike Stolz, and Marshall Presser

Memory Is the New Disk

Prior to 2002, memory was considered expensive and disks were considered cheap. Networks were slow(er). We stored things we needed access to on disk and we stored historical information on tape.

Since then, continual advances in hardware and networking and a huge reduction in the price of RAM has given rise to memory clusters. At around the same time of this fall in memory prices, GemFire was invented, making it possible to use memory as we previously used disk. It also allowed us to use Atomic, Consistent, Isolated, and Durable (ACID) transactions in memory just like in a database. This made it possible for us to use memory as the system of record and not just as a "side cache," increasing reliability.

What Is Pivotal GemFire?

Is it a database? Is it a cache? The answer is "yes" to both of those questions, but it is much more than that. GemFire is a combined data and compute grid with distributed database capabilities, highly available parallel message queues, continuous availability, and an

event-driven architecture that is linearly scalable with a super-efficient data serialization protocol. Today, we call this combination of features an *in-memory data grid* (IMDG).

Memory access is orders of magnitude faster than the disk-based access that was traditionally used for data stores. The GemFire IMDG can be scaled dynamically, with no downtime, as data size requirements increase. It is a key–value object store rather than a relational database. It provides high availability for data stored in it with synchronous replication of data across members, failover, self-healing, and automated rebalancing. It can provide durability of its in-memory data to persistent storage and supports extremely high performance. It provides multisite data management with either an active–active or active–passive topology keeping multiple datacenters eventually consistent with one another.

Increased access to the internet and mobile data has accelerated the evolution of cloud computing. The sheer number of accesses by users and apps along with all of the data they generate will continue to expand. Apps must scale out to not only handle the growth of data but also the number of concurrent requests. Apps that cannot scale out will become slower to the point at which they will either not work or customers will move on to another app that can better serve their request.

A traditional web tier with a load balancer allowed applications to scale horizontally on commodity hardware. Where is the data kept? Usually in a single database. As data volumes grow, the database quickly becomes the new bottleneck. The network also becomes a bottleneck as clients transport large amounts of data across the network to operate on it. GemFire solves both problems. First, the data is spread out horizontally across the servers in the grid taking advantage of the compute, memory, and storage of all of them. Second, GemFire removes the network bottleneck by colocating application code with the data. Don't send the data to the code. It is much faster to send the code to the data and just return the result.

What Is Apache Geode?

When Pivotal embarked on an open source data strategy, we contributed the core of the GemFire codebase to the Apache Software Foundation where it is known as the Apache Geode top-level project. Except for some commercial extensions that we discuss

later, the bits are mostly the same, but GemFire is the enterprise version supported by Pivotal.

What Problems Are Solved by an IMDG?

There are two major problems solved by IMDGs. The first is the need for independently scalable application infrastructure and data infrastructure. The second is the need for ultra-high-speed data access in modern apps. Traditional disk-based data systems, such as relational database management systems, were historically the backbone of data-driven applications, and they often caused concurrency and latency problems. If you're an online retailer with thousands of online customers, each requesting information on multiple products from multiple vendors, those milliseconds add up to seconds of wait time, and impatient users will go to another website for their purchases.

Real GemFire Use Cases

The need for ultra-high-speed data access in modern applications is what drives enterprises to move to IMDGs. Let's take a look at some real customer use cases for GemFire's IMDG.

Transaction Processing

Transportation reservation systems are often subject to extreme spikes in demand. They can occur at special times of year. For instance, during the Chinese New Year, one sixth of the population of the earth travels on the China Rail System over the course of just a few days. The introduction of GemFire into the company's web and e-ticketing system made it possible to handle holiday traffic of 15,000 tickets sold per minute, 1.4 billion page views per day, and 40,000 visits per second. This kind of sudden increase in volume for a few days a year is one of the most difficult kinds of spikes to manage.

Similarly, Indian Railways sees huge spikes at particular times of day, such as 10 A.M. when discount tickets go on sale. At these times the demand can exceed the ability of almost any nonmemory-based system to respond in a timely fashion. India Railways suffered from serious performance degradation when more than 40,000 users would log on to an electronic ticketing system to book next-day

travel. Often it would take users up to 15 minutes to book a ticket and their connections would often time out. The IT team at India Railways brought in the GemFire IMDG to handle this extreme workload. The improved concurrency management and consistently low latency of GemFire increased the maximum ticket sale rate from 2,000 tickets per minute to 10,000 per minute, and could accommodate up to 120,000 concurrent user sessions. Average response time dropped to less than one second, and more than 50% of the responses now occur in less than half a second. The GemFire cluster is deployed behind the application server tier in the architecture with a write-behind to a database tier to ensure persistence of the transactions.

High-Speed Data Ingest and the Internet of Things

Increasingly, automobiles, manufacturing processes, turbines, and heavy-duty machinery are instrumented with myriad sensors. Disk-centric technologies such as databases are not able to quickly ingest new data and respond in subsecond time to sensor data. For example, certain combinations of pressure and temperature and observed faults predict conditions are going awry in a manufacturing process. Operator or automated intervention must be performed quickly to prevent serious loss of material or property.

For situations like these, disk-centric technologies are simply too slow. In-memory techniques are the only option that can deliver the required performance. The sensor data flows into GemFire where it is scored according to a model produced by the data science team in the analytical database. In addition, GemFire batches and pushes the new data into the analytical database where it can be used to further refine the analytic processes.

Offloading from Existing Systems/Caching

The increase in travel aggregator sites on the internet has placed a large burden on traditional travel providers for rapid information about availability and rates. The aggregator sites frequently give preference to enterprises that respond first. Traditionally, relational database systems were used to report this information. As the load grew due to the requests from the aggregators, response time to requests from the travel providers' own websites and customer agents became unacceptable. One of these travel providers installed GemFire as a caching layer in front of its database, enabling much

quicker delivery of information to the aggregators as well as offloading work from its transactional reservations system.

Event Processing

Credit card companies must react to fraudulent use and other misuse of the card in real time. GemFire's ability to store the results of complex decision rules to determine whether transactions should be declined means complex scoring routines can execute in milliseconds or better if the code and data are colocated. Continuous content-based queries allow GemFire to immediately push notifications to interested parties about card rejections. Reliable write-behind saves the data for further use by downstream systems.

Microservices Enabler

Modern microservice architectures need speedy responses for data requests and coordination. Because a basic tenet of microservices architectures is that they are stateless, they need a separate data tier in which to store their state. They require their data to be both highly available and horizontally scalable as the usage of the services increases. The GemFire IMDG provides exactly the horizontal scalability and fault tolerance that satisfies those requirements. Microservices-based systems can benefit greatly from the insertion of GemFire caches at appropriate places in the architecture.

IMDG Architectural Issues and How GemFire Addresses Them

IMDGs bring a set of architectural considerations that must be addressed. They range from simple things like horizontal scale to complicated things like ensuring that there are no single points of failure anywhere in the system. Here's how GemFire deals with these issues.

Horizontal Scale

Horizontal scale is defined as the ability to gain additional capacity or performance by adding more nodes to an existing cluster. GemFire is able to scale horizontally without any downtime or interruption of service. Simply start some more servers and GemFire will automatically rebalance its workload across the resized cluster.

Coordination

GemFire being an IMDG is by definition a distributed system. It is a cluster of members distributed across a set of servers working together to solve a common problem. Every distributed system needs to have a mechanism by which it coordinates membership. Distributed systems have various ways of determining the membership and status of cluster nodes. In GemFire, the Membership Coordinator role is normally assumed by the eldest member, typically the first `Locator` that was started. We discuss this issue in more detail in Chapter 2.

Organizing Data

GemFire stores data in a structure somewhat analogous to a database table. We call that structure in GemFire a *+Region+*. You can think of a `Region` as one giant Concurrent Map that spans nodes in the GemFire cluster. Data is stored in the form of keys and values where the keys must be unique for a given `Region`.

High Availability

GemFire replicates data stored in the `Regions` in such a way that primary copies and backup copies are always stored on separate servers. Every server is primary for some data and backup for other data. This is the first level of redundancy that GemFire provides to prevent data loss in the event of a single point of failure.

Persistence

There is a common misconception that IMDGs do not have a persistence model. What happens if a node fails as well as its backup copy? Do we lose all of the data? No, you can configure GemFire `Regions` to store their data not only in memory but also on a durable store like an internal hard drive or external storage. As mentioned a moment ago, GemFire is commonly used to provide high availability for your data. To guarantee that failure of a single disk drive doesn't cause data loss, GemFire employs a shared-nothing persistence architecture. This means that each server has its own persistent store on a separate disk drive to ensure that the primary and backup copies of your data are stored on separate storage devices so that there is no single point of failure at the storage layer.

CHAPTER 2

Cluster Design and Distributed Concepts

Mike Stolz

The Distributed System

Typically, a GemFire distributed system consists of any number of members that are connected to one another in a peer-to-peer fashion, such that each member is aware of the availability of every other member at any time. It is called a distributed system because the members of the cluster are distributed across many servers in order to provide high availability and horizontal scalability. Figure 2-1 shows a typical GemFire setup.

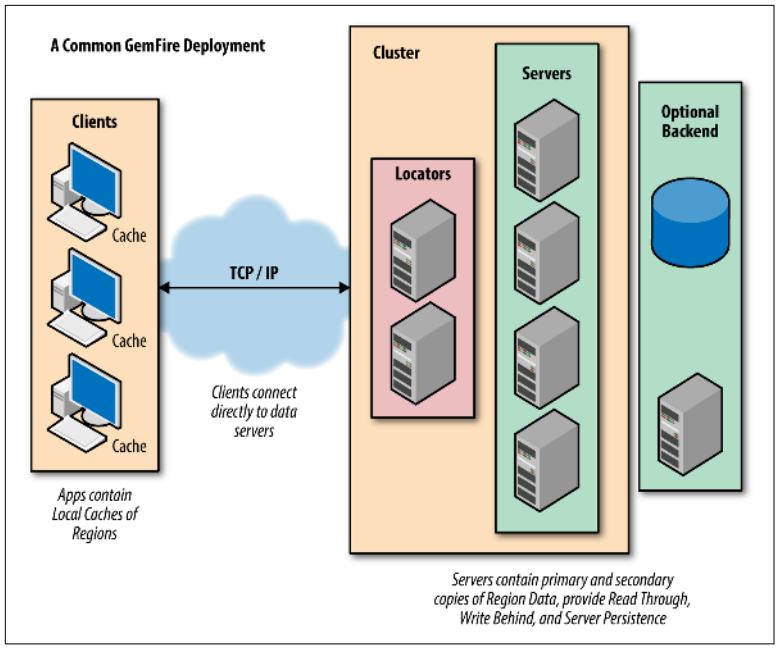

Figure 2-1. A common GemFire deployment

Cache

The `Cache` is the base abstraction of GemFire. It is the entry point to the entire system. Think of it as the place to define all the storage for the data you will put into the system. In some ways it is similar to the construct of "database" in the relational world. There is also a notion of a cache on the clients connected to the GemFire distributed system. We refer to this as a `ClientCache`. We usually recommend that this `ClientCache` be configured to be kept up-to-date automatically as data changes in the server-side cache.

Regions

`Regions` are similar to tables in a traditional database. They are the container for all data stored in GemFire. They provide the APIs that you put data into and retrieve data from GemFire. The `Region` API also provides many of the quality-of-service capabilities for data stored in GemFire such as eviction, overflow, durability, and high availability.

Locator

The GemFire `Locators` are members of the GemFire distributed system that provide the entry point into the cluster. The `Locators`' hostnames and ports are the only "well-known" addresses in a GemFire cluster. To provide high availability, we usually recommend that you configure and start three `Locators` per cluster. When any GemFire process starts (including a `Locator`), it first reaches out to one of the `Locators` to provide the new process's IP and port information and to join the distributed system. The membership coordinator that runs inside a `Locator` is responsible for updating the membership view and providing addresses of new members to all existing members, including the newly joined member.

When a GemFire client starts, it also connects to a `Locator` to get back the addresses of all of the data serving members in the cluster. Clients normally connect to all of those data serving members, affording them a single hop to access data that is hosted on any of the servers.

CacheServer

The `CacheServers` are what we have been referring to as data serving members up until now. Their primary purpose is to safely store the data that applications put into the cluster. `CacheServers` are the members in a GemFire cluster that host the `Regions`.

Dealing with Failures: The CAP Theorem

Having multiple components in a distributed system leads to a problem that single-node systems do not have, namely what happens in the case of a failure in which some nodes in the cluster cannot speak to others. A wise old man once said that there are two kinds of clusters: ones that have had failures and others that haven't had failures yet.

Let's take a break from the discussion of components and discuss this important topic and how GemFire clusters deal with it.

One scenario is that updates will be made to one `CacheServer` in the cluster that will not be replicated to some others because the net-

work connection between them is broken. Some of the members will have updated data and some will not.

This is referred to by Eric Brewer in his CAP theorem as the *Split Brain* problem. The CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees (see also Figure 2-2):

Consistency
: Every read receives the most recent write or an error.

Availability
: Every request receives a nonerror response, without guarantee that it contains the most recent write.

Partition tolerance
: The system continues to operate despite an arbitrary number of messages being dropped or delayed by the network between nodes.

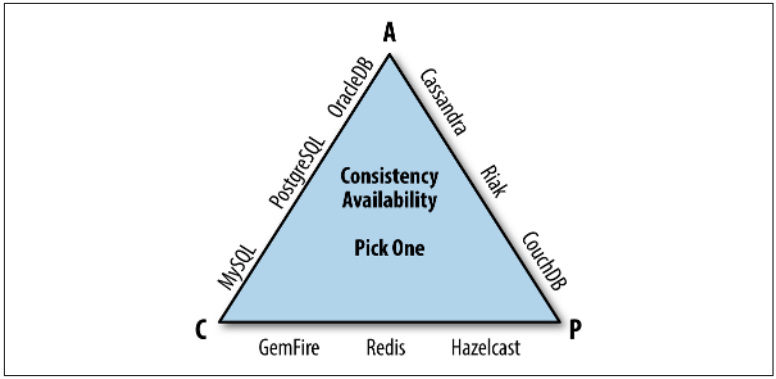

Figure 2-2. The CAP triangle

In other words, the CAP theorem states that in the presence of a network partition, you must choose between consistency and availability. In 2002, Seth Gilbert and Nancy Lynch of MIT published a formal proof of Brewer's conjecture, rendering it a theorem.

Mission-critical applications that deal with real property or use cases like flight operations require that they operate on *correct* data. This means that having an old copy of data available in the case of a network issue is not as good as getting an error when trying to access it. In many cases, there is a separate backing store behind the in-memory data grid (IMDG), which we can use as a secondary source

of truth in the event that some data is missing from the IMDG. For this reason, GemFire is biased toward consistency over availability. In the event of network segmentation, GemFire will always return the most recent successful write, or an error. To mitigate the potential for this kind of error, GemFire is usually configured to hold multiple copies of the data and to spread those copies across multiple availability zones, thereby reducing the possibility that all copies will be on the losing side of the network split.

Availability Zones/Redundancy Zones

Availability zones are a cloud construct that attempts to provide some level of assurance that two zones will not be taken down at the same time. Operations such as rolling restarts for maintenance are done by most cloud providers one availability zone at a time.

You can map availability zones onto GemFire's Redundancy Zone concept. Since GemFire is responsible for the high availability of your data, it should be configured to set its redundancy zones to match the cloud's availability zones. GemFire always makes sure not to store the primary copy and the backup copies for any data object in the same redundancy zone.

Cluster Sizing

Now that we understand the basic components, the next question that new GemFire administrators confront is sizing the cluster. Several considerations go into sizing a GemFire cluster. The first one is how much data you want to store in memory. That decision drives nearly everything else about how big the cluster needs to be.

Other important inputs to the sizing are how many copies of each object you want to keep in memory for high availability, how big the indexes on the data be, and how rapidly objects change in the system, causing the creation of garbage that needs to be collected.

How rapidly objects change is a tuning consideration to ensure that the Java Garbage Collector can keep up with the amount of garbage that is being created. It is common in Java-based applications for the Garbage Collector to be configured to kick in at 65% heap usage, so there is only 35% empty space available. However, GemFire is not a common Java-based application. It is primarily intended for storing your data in memory. Therefore, in many GemFire configurations

that small amount of empty space might not be sufficient. The second most important input into cluster sizing is how much space you want to leave unused in the cluster members in order to recover the data redundancy Service-Level Agreement (SLA) (i.e., number of copies) when a node eventually fails.

If you have only two members in the cluster, you cannot recover the redundancy SLA at all. If you have three members, you need to leave at least one-third of the memory unused in each member in order to recover the redundancy SLA. Also, if your redundancy SLA is three copies, even with three members you cannot recover your redundancy SLA.

So, you can see how with relatively small datasets it still makes sense to think about clusters with as many as nine members so that protecting against a single member failing requires only a small fraction of the memory of the overall system to be left empty.

Virtual Machines and Cloud Instance Types

Most IT organizations today run all of their workloads on some sort of virtual machine rather than bare metal. Sizing virtual machines can be a tricky business. There are a lot of things that you need to take into consideration to get the right settings. As you can see in Figure 2-3, which is excerpted from the VMware Best Practices Guide, the overall memory reservation needs to be set to the heap size, which is driven by all of the aforementioned considerations, plus the Perm Gen size, which is usually around 256 MB plus 192 k multiplied by the number of threads likely to be running in GemFire (100 is a good guess). There is also some other memory usage consumed by things like I/O buffers, file descriptors, and such. That can usually be thought of as around 1 GB for all of them.

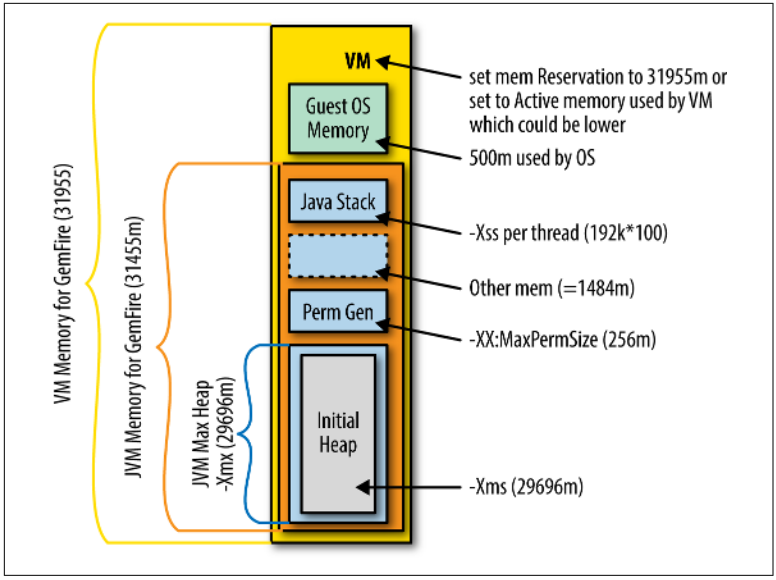

Figure 2-3. An example of memory usage in a VM hosting GemFire

Finally, there is the operating system (OS) itself, which is likely to consume about 500 MB. Thus, in the example in Figure 2-3, we're allocating 29.696 GB of memory to the Java heap, a total of 31.455 GB of memory to the GemFire Java Virtual Machine (JVM) as a whole and 0.5 GB of memory for the OS memory.

Two More Considerations about JVM Size

First, consider Java's ability to use smaller pointers when the JVM size is below 32 GB. This is known in the Java world as Compressed Oops. There is typically a significant savings from this. In fact, we have seen cases in which you cannot put more data into a GemFire cluster between 32 GB and 48 GB simply because the pointers consume so much more space.

Second, let's consider the nonuniform memory architecture (NUMA) of large-scale modern computers. It is easy now to procure single server-class machines with 256 or even 512 GB of memory. That memory is typically broken up into several NUMA nodes. There will be as many NUMA nodes as there are physical CPU sockets in the server. The idea behind NUMA is that each CPU will primarily execute accessing only memory on the NUMA node that

is directly connected to its socket. That connection is extremely fast and gives the best performance. If the CPU has to access memory that is in a different NUMA node, there is a significant penalty incurred, sometimes as much as a 30% penalty. So, it is important to size GemFire VMs so that they fit entirely in one NUMA node.

CHAPTER 3

Quickstart Example

Mike Stolz

Now that you have some understanding of the GemFire architecture, this chapter gets you started building a small GemFire instance on a single host. The product documentation illustrates how to build a more production-worthy cluster. In this chapter, we use the name GemFire, but the process is the same for the Geode version. We illustrate the process in Linux, but it is substantially the same in Windows or macOS.

Operating System Prerequisites

We have found that many problems building clusters arise from misconfigured operating system (OS) parameters. Please carefully follow these instructions. In particular, on some versions of OS X you must ensure that the hostname and IP of your machine is configured in your */etc/hosts* file in order for GemFire to operate correctly.

Confirm that your system meets the hardware and software requirements described in Host Machine Requirements.

If you have not already done so, download and install a compatible Java Development Kit (JDK) for your system.

Installing GemFire

You can download and install Pivotal GemFire binaries from *http://bit.ly/2zJFbUs*. On the Pivotal GemFire product page, locate Downloads. Download the ZIP distribution of GemFire.

Or, you can download and install Geode binaries from *https://geode.apache.org*.

Use the downloaded ZIP distribution to install and configure GemFire on every physical and virtual machine (VM) where you will run it.

Use the following procedure to install GemFire:

1. Navigate to the directory where you want GemFire to be installed, and then unzip the *.zip* file.
2. Configure the `JAVA_HOME` environment variable to a supported JDK installation. (You should find a bin directory containing the java binary under `JAVA_HOME`.)

To run GemFire and its utilities, you need to be running Java 1.8.

3. Set the `GEMFIRE` environment variable to the location where GemFire was installed. (You should find a bin directory containing `gfsh` in the directory to which you set the `GEMFIRE` variable.)
4. Add the path to the bin directory of the GemFire distribution to the end of your system `PATH` variable.
5. Set the `CLASSPATH` to point to the *geode-dependencies.jar* that supplies the rest of the dependencies.

It is best to put all of these settings into a script that you can run before starting `gfsh` and before starting any program using GemFire.

For example, in Unix, Linux, and macOS, the script would look something like this:

```
JAVA_HOME=/usr/java/jdk1.8.0_92 ; export JAVA_HOME
GEMFIRE=/opt/GemFire9.1.0 ; export GEMFIRE
PATH=$PATH:$GEMFIRE/bin ; export PATH
CLASSPATH=$GEMFIRE/lib/geode-dependencies.jar;export CLASSPATH
```

To run the preceding script, place it in a file named *genv.sh*, and then use the "." command to run the script in the context of the currently executing shell, as shown here:

```
$ . genv.sh
```

In Windows, place the script in a file named *genv.bat*, and then run it from the command line as usual. It will automatically run in the context of the current command shell:

```
set JAVA_HOME=c:\Program Files\Java\jdk1.8.0_92
set GEMFIRE=c:\GemFire9.1.0
set PATH=%PATH%;%GEMFIRE%\bin
set CLASSPATH=%GEMFIRE%\lib\geode-dependencies.jar
```

Starting the Cluster

After you have done that, you can set up a folder in which you will start a GemFire cluster, and then build a sample app using GemFire. You can find the examples in this book in a folder named *hello* in your home directory.

GemFire Shell

The GemFire SHell (`gfsh`) utility is a command-line tool that supports administration, debugging, and monitoring of GemFire and Geode. The GemFire shell is a Java Management Extensions (JMX) client to GemFire. A module referred to as the JMX manager handles the `gfsh` client connections:

```
$ gfsh
```


```
gfsh> start locator --name=locator
Starting a Geode Locator in /Users/mstolz/hello/locator...
Locator in /home/mstolz/hello/locator on myhost[10334] as
locator is currently online.
...A whole lot of output elided here for brevity...
```

```
Successfully connected to: JMX Manager
Cluster configuration service is up and running.
```

Now that we have a `Locator` running and we are connected to it, let's start a GemFire server to host our data:

```
gfsh> start server --name=server1
Starting a Geode Server in /Users/mstolz/hello/server1...
Server in /Users/mstolz/hello/server1 on myhost[40404] as
server1 is currently online.
...A whole lot of output omitted here for brevity...
```

Then, you can create a `Region`:

```
gfsh> create region --name=hello --type=REPLICATE
Member  | Status
------- | ----------------------------------
server1 | Region "/hello" created on "server1"
```

Something Fun: Time to One Million Puts

Now we can write a client application. This little sample application will write one million records into the GemFire `Region` named "hello" on the cluster we just started:

```
import org.apache.geode.cache.Region;
import org.apache.geode.cache.client.*;
import java.util.Date;

public class hello {
  public static void main(String[] args)
  throws Exception {
    ClientCache cache = new ClientCacheFactory()
        .addPoolLocator("localhost", 10334).create();
    Region<String, String> region =
        cache.<String,String>createClientRegionFactory
        (ClientRegionShortcut.CACHING_PROXY)
        .create("hello");

    System.out.println("Putting 1,000,000 entries");
    System.out.println("Start:  " + new Date());
    for(int i=1;i<1000001;i++)
        region.put(""+i, " " + i + "Hello World");
    System.out.println("Finish: " + new Date());

    cache.close();
  }
}
```

You will want a *gemfire.properties* file to suppress printing of the client configuration:

```
log-level=warning
```

Build the Hello example:

```
$ javac hello.java
```

When we run the application, it connects to the GemFire cluster that we just started and puts a million objects into the `hello Region`:

```
$ java -cp $CLASSPATH:hello.class hello
Putting 1,000,000 entries
Start:  Fri Sep 08 13:44:02 PDT 2017
Finish: Fri Sep 08 13:45:08 PDT 2017
```

Check that the data actually got into the server. By using the `gfsh describe region` command we can see that there is a `Region` named `hello` with its `Data Policy` attribute set to `replicate`, hosted on `server1`, and its size is `1000000`:

```
gfsh> describe region --name=hello
------------------------------------------------
Name            : hello
Data Policy     : replicate
Hosting Members : server1

Non-Default Attributes Shared By Hosting Members

 Type  |    Name     | Value
------ | ----------- | ---------------
Region | data-policy | REPLICATE
       | size        | 1000000
       | scope       | distributed-ack

gfsh> query --query="select * from /hello limit 3"

Result     : true
startCount : 0
endCount   : 20
Rows       : 3

Result
------------------
 576875 Hello World
 532879 Hello World
 201441 Hello World

NEXT_STEP_NAME : END
```

Now, let's start another server:

```
gfsh> start server --name=server2 --server-port=40406
Starting a Geode Server in /Users/mstolz/hello/server2...
Server in /Users/mstolz/hello/server1 on myhost[40404] as
server2 is currently online.
Geode Version: 9.1.0
...A whole lot of output elided here for brevity...
gfsh>list members
 Name   | Id
------- | -----------------------------------------
locator | 192.168.1.12(locator:50290:locator)<ec><v0>:1024
server1 | 192.168.1.12(server1:50317)<v1>:1025
server2 | 192.168.1.12(server2:50381)<v2>:1026
```

After we start this, let's describe the `Region` again, and we will see that both `server1` and the new `server2` are hosting the `Region` now:

```
gfsh> describe region --name=hello
------------------------------------------------------
Name            : hello
Data Policy     : replicate
Hosting Members : server2
                  server1
```

So now if we stop `server1` and do the query again we will see that we still have our data being served up from `server2`:

```
gfsh> stop server --name=server1
Stopping Cache Server running in /Users/mstolz/hello/server1 on
myhost[40404] as server1...

gfsh> query --query="select * from /hello limit 3"
Result     : true
startCount : 0
endCount   : 20
Rows       : 3

Result
------------------
 576875 Hello World
 532879 Hello World
 201441 Hello World
```

Now, it's time to shut down the GemFire servers and `Locator`:

```
gfsh> shutdown --include-locators=true
As a lot of data in memory will be lost, including possibly
events in queues, do you really want to shutdown the entire
distributed system? (Y/n): y
Shutdown is triggered
gfsh>
No longer connected to myhost[1099].
gfsh> quit
```

```
Exiting...
$
```

You have built your first GemFire-based application. See how easy it is to get started? Next, let's take a look at the bigger picture of using Spring Data GemFire.

CHAPTER 4

Spring Data GemFire

John Guthrie

If you've been working with enterprise Java recently, you likely have heard of, and almost equally likely have used, Spring Framework (commonly referred to as just "Spring"). Spring is used for building modern enterprise applications. At its core, Spring is a lightweight dependency-injection architecture that allows for loose coupling of components. Its features bring all manner of benefits to the Spring developer. Spring also brings benefits to the application, including modularity and scalability (which plays right into the design of today's cutting-edge microservices architectures), testing and mocking, metrics, and a single configuration platform (Spring).

But Spring Framework extends well beyond its core. One of the most used and most long-lived projects under Spring Framework is Spring Data.

What Is Spring Data?

The Spring Data team explains it this way on its home page:

> Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.

Spring Data is all about accessing your data, and doing so in a consistent way, irrespective of how you store that data, be it in a relational database like MariaDB, a NoSQL database like MongoDB, or an in-memory data grid like GemFire.

This chapter is going to look at Spring Data, and the GemFire implementation of Spring Data, called, unsurprisingly, Spring Data GemFire. Spring Data GemFire lets Spring developers easily get up and running with GemFire by using constructs, patterns, and configurations that are familiar to them. Spring Data is an agnostic way to access backend data stores, but Spring Data GemFire provides hooks to GemFire-specific capabilities

Getting Started

So, let's revisit the Hello example from Chapter 3, but, to get you introduced to Spring with GemFire, we'll rewrite it to use Spring's configuration and annotation features. This isn't, strictly speaking, Spring Data Gemfire, but it's a good start in showing how Spring and GemFire work together. After this quick example, which uses Spring to create a local cache on the fly, we dive into Spring Data GemFire, and show a much more robust and Spring Data–oriented example.

For our example, we are going to use Java configuration and annotations to configure our Spring `ApplicationContext`; also, for simplicity's sake we repurpose our main class as the configuration class, as well. The code that follows, complete with a Maven *pom.xml* file,[1] is available on our GitHub site.

```
@Configuration
public class HelloWorld {
    @Bean
    CacheFactoryBean gemfireCache() {
        CacheFactoryBean gemfireCache =
            new CacheFactoryBean();
        return gemfireCache;
    }

    @Bean(name="hello")
    LocalRegionFactoryBean<String, String>
      getEmployee(GemFireCache cache) {
        LocalRegionFactoryBean<String, String> region =
              new LocalRegionFactoryBean<String, String>();
        region.setCache(cache);
        region.setName("hello");
        return region;
```

1 Spring and GemFire have many JAR dependencies. By far, the easiest way to manage these dependencies is with Maven or Gradle.

```
    }

    public static void main(String[] args) {
      ApplicationContext context =
         new AnnotationConfigApplicationContext(
             HelloWorld.class);
      Region<String, String> region =
           (Region<String, String>) context.getBean("hello");
      region.put("1", "Hello");
      region.put("2", "World");
      for (Map.Entry<String, String>  entry :
            region.entrySet()) {
              System.out.format("key = %s, value = %s\n",
              entry.getKey(), entry.getValue());
      }
      region.close();
      region.getCache().close();
       System.exit(0);
    }
}
```

If you're a Spring developer, this programming style should look very familiar to you. But what you've seen here is really just some Spring-ification of standard GemFire access patterns, and that is not Spring Data GemFire. Spring Data GemFire helps you access your (GemFire) data in the way that you would access any persistent data source. This starts with the idea that all data is contained in repositories; access to those disparate repositories has some commonalities that can be reflected in the way you access them in Spring. So, Spring Data GemFire is Spring Data with a GemFire repository.

Spring Data GemFire Features

What does Spring Data GemFire offer above the standard GemFire Java API? First, Spring Data GemFire lets you configure your GemFire cache in the same way that you configure the rest of your Spring application. Every configuration in GemFire's Java API is also available in Spring Data GemFire. Spring Data GemFire also lets you use all the configuration features and shortcuts built into Spring, including the following:

Profile support
: Spring profiles make it easy to set up different runtime profiles that can all be held in a common location.

Property placeholders
: Spring property placeholders allow properties to be stored externally and pulled in at configuration time. Properties are usually kept in XML, YAML, or standard *.properties* formats. Managing properties externally is a tenet of the 12-factor methodology used in microservices architectures.

Spring Expression Language (SpEL) support
: SpEL is an expression language that is evaluated at runtime; it can be used in Spring to configure GemFire beans (and others) based on values returned at runtime from other beans.

Let's look at an example with Spring Data GemFire. But first, let's get into a Spring Data frame of mind. Spring Data abstracts the underlying data layer and minimizes the emphasis on the persistence of data. It keeps the focus on the business domain; Spring Data GemFire exists to enhance our ability to maintain, query, and persist objects in that domain.

So, in this simple example, our domain is the art world; we will model the `Artist` object (note, again, full source code for this example is in our GitHub repository):

```
@Region("artist")
public class Artist {
    @Id
    public String name;
    public String style;
    public int yearOfBirth;
}
```

Savvy Java developers might cringe that we've made the object's attributes `public`, but we're trying to keep the example small. Notice, though, the `@Id` annotation. This is a Spring Data annotation used to indicate the ID, key, or primary key, in various data stores. With Spring Data, you can use this annotation; the framework will map that to whatever construct the data store uses.

Next, we will write our Java configuration class. This is a standard Spring methodology, and this class contains a mixture of Spring annotations and GemFire classes; in other words, it builds the underlying infrastructure for Spring Data Gemfire. Note that we've omitted the package declaration and imports for conciseness:

```
@Configuration
public class GemdemoConfiguration {

    @Bean
    Properties gemfireProperties() {
        Properties gemfireProperties = new Properties();
        gemfireProperties.setProperty("mcast-port", "0");
        return gemfireProperties;
    }

    @Bean
    CacheFactoryBean gemfireCache() {
     CacheFactoryBean gemfireCache = new CacheFactoryBean();
     gemfireCache.setClose(true);
        gemfireCache.setProperties(gemfireProperties());
        return gemfireCache;
    }

    @Bean(name="artist")
    LocalRegionFactoryBean<String, Artist>
        getArtist(final GemFireCache cache) {
        LocalRegionFactoryBean<String, Artist> artistRegion
            = new LocalRegionFactoryBean<String, Artist>();
        artistRegion.setCache(cache);
        artistRegion.setName("artist");
        return artistRegion;
    }
}
```

With our domain class done and our GemFire cache set up, we're now ready to tap into the elegance of Spring Data. As mentioned earlier, Spring Data abstracts various data stores into a common object called a *repository*. When you're creating a Spring Data (GemFire) application, you define the repository for your domain object, and how you want to query it. The magic here is that the definition you provide need only be an *interface*—Spring Data will infer what you're trying to accomplish, based upon method names and other introspection techniques, and will then generate the logic for you. So, in our simple example, we are going to define an `Artist` repository that can query on name (the ID/key), and can also query for artists born after a specified year. The `ArtistRepository` will look like this:

```
@Repository
public interface ArtistRepository extends
    GemfireRepository<Artist, String> {
    Artist findByName(String name);
    Artist findByStyle(String style);
    Iterable<Artist> findByStyleIn(
```

```
            Collection<String> modernStyles);
        Iterable<Artist> findByYearOfBirthGreaterThan(
            double yearOfBirth);
    }
```

Spring Data looks at each method in the interface and figures out what is needed. The `findByXxx` methods, where `Xxx` is a domain object attribute, are obvious, but Spring Data can also grok the more complex method names; for example, `findByStyleIn` will look for each domain object that has a style attribute equal to one of the ones in the parameter list. The `findByYearOfBirthGreaterThan` method is examined by Spring Data, which can parse out the "`YearOfBirth`" part as an attribute of `Artist`, and the "`GreaterThan`" as a relationship to be tested; it generates an implementation of the interface that does the right thing, as the right thing is defined by the backing data store (in our case, GemFire).

What you've seen here is the tip of the Spring Data GemFire iceberg. Spring Data GemFire also puts a nice Spring-based wrapper around GemFire functions, making their management much easier for Spring developers; Spring Data core also has many more features worth further investigation—the Spring Data home page is a great place to start when you want to find out more.

CHAPTER 5

Designing Data Objects in GemFire

Mike Stolz

> Three NoSQL developers walked into a bar. They left when they couldn't find a table.
>
> —*NoSQL data design is different.*

To make good use of any data tool, you need to know a bit about best practices for using it. GemFire is no different, especially if you're used to thinking in a relational database manner. This chapter outlines the principles of good data design in GemFire's in-memory data grid (IMDG).

The Importance of Keys

GemFire is primarily a key–value store. The strong implication you can and should derive from that statement is that the primary key for objects stored in GemFire is very important indeed.

The construct that GemFire exposes for data storage is called a `Region`. A `Region` is similar to a database table but it is an implementation of `ConcurrentMap`. So, you can think of it as just a *hashmap*. In a hashmap, the two primary API's are `put(key,value)` and `object=get(key)`.

The first requirement is that keys in GemFire `Region`s must be unique. In fact, this is a requirement of `ConcurrentMap` itself.

The second important characteristic is that the keys must be efficient. Although it is possible to use complex objects as keys, their

implementation of hashcode and equals needs to be very fast. For beginners, it is recommended that simple `String` keys are the easiest to manage.

In fact, there are many use cases for which the strings used as keys might even be user friendly and easily remembered. For instance, in the financial services community one of the commonly used constructs is to name tradable instruments in a user-friendly way. That's why simple constructs like "ticker" have become so pervasive. If you want to trade Apple, you use the ticker "APPL." For International Business Machines the ticker is "IBM." These are very efficient keys because they can use the default implementation of hashcode and equals on `java.lang.String` and still work very rapidly.

Sometimes, it is handy to put some additional information into the key. For instance, in financial services there are commonly used conventions for certain market data elements. An example is the London Interbank Offered Rate (LIBOR) rate, which is set once per day, every day, and is commonly used in the valuation of many trading instruments. In practical usage, the LIBOR rate is usually keyed to a particular currency such as the US dollar (USD). It also has a future date associated with it like three months or six months. So a really common naming convention for the LIBOR rate is "USD.LIBOR.3M" and "USD.LIBOR.6M."

Bad Keys

There are certain constructs for keys that are not usable in GemFire:

Nonunique
: Every key in a given `Region` must be unique.

Mutable
: Keys must be constant.

Partitioned Regions

GemFire achieves horizontal scale for the data that is stored in it via a partitioning (aka sharding) mechanism called a Partitioned Region.

The partitions in GemFire are sometime referred to as "buckets." This is based on the same notion as buckets in a hashmap. The buckets are spread out evenly across all the data-serving members of

the cluster for a given region. There are primary buckets and backup buckets. The backup buckets are also spread out evenly across all the data-serving members of the cluster. These backup buckets are used to provide high availability by storing multiple copies of data in the region, spread across the primary and backup buckets. In the specific case of a Partitioned Region, the data is partitioned by taking a hash of the key modulo the number of buckets in the system. The key will be in the same bucket number in both the primary set of buckets and the secondary set of buckets, but the primary and secondary buckets for a given key will never reside on the same server.

Partitioned Regions are the optimal way of storing data in GemFire because they give you horizontal scale at the data layer. Things like customer records, orders, shipments and payments, which can all be partitioned by the customer ID, are good candidates for Partitioned Regions. You would have a customer `Region`, an orders `Region`, a shipments `Region`, and a payments `Region`.

Colocation

Sometimes, various parts of a dataset are related in some way. For instance, customer records, orders, shipments, and payments are all related to one another through the customer. The customer has orders, shipments, and payments.

GemFire provides a mechanism by which you can tell it which `Region`s are related and the foreign key relationship. The easiest way to provide the foreign key relationship information is by using the built-in GemFire `StringPrefixPartionResolver`, which uses a simple naming convention to define the foreign key relationship. Given a customer ID C01234 and a shipment number S0002, the key for the shipment would be formatted as "C01234|S0002." Of course, the key for the customer record would be just be "C01234."

Replicated Regions

The GemFire Replicated Region is a very different way of organizing data. It is intended to solve for many-to-many relationships between related data elements. The data is spread across all the data-serving members of the cluster, but not via sharding or partitioning. Rather ALL of the data in a Replicated Region is present on all data-serving members.

Even though this doesn't scale horizontally at all, it doesn't tend to be too big of an issue because it is usually a much smaller dataset containing things like reference data.

Things like product data tend to be good candidates for storage in Replicated Regions, because all customers need access to all products, and you likely have far fewer products than you do customers.

Designing Optimal Data Types

There are a lot of considerations that go into designing data types in every programming context. For example, in the relational world, you break `Orders` down into `OrderHeading` and `OrderItems` and put those into separate tables with foreign key relationships. GemFire has its own considerations related to performance and manageability.

Operations on GemFire are automatically atomic, so if you usually operate on both the `OrderItems` and the `OrderHeader` as an atomic unit of work, then you will get best performance by using a containment relationship. The `Order` contains both the fields from the `OrderHeading` and the actual `OrderItems` in a list or map.

When you go to update within a transaction, the transaction needs to lock only one `Order` very briefly during the commit operation.

On the other hand, if you have an object that has a large number of slowly changing attributes and some smaller number of very rapidly changing attributes, you might find that you get the best performance by splitting the object up in two: the slowly changing attributes in one class, and the rapidly changing attributes in a separate class. These two different objects would be placed into separate colocated regions and would probably have the same exact key for each.

Using the stock ticker example discussed earlier, it is common practice to put the description of the instrument, which is slowly changing (or never changing), into one `Region` and the Bid, Ask, High, Low, Open, and Close fields into a second `Region`, both keyed by the "ticker" (e.g., AAPL), and mark those two `Regions` as "colocated."

There is another trade-off in object design for GemFire in terms of searching and querying. If your usage pattern isn't primarily key–

value as described in this chapter until now, you need to take into consideration the complexity of the searches/queries.

It is a common practice to put the searchable fields of your rich domain object into a top-level construct that is easily any quickly accessible by the Lucene indexing engine or the OQL engine, and to still keep the nested versions of those fields in the object hierarchy in their natural form. In fact, it is not uncommon to have completely separate searchable objects that live in a different `Region` from the rich hierarchical object that they represent. This of course presents a trade-off in a highly transactional system wherein both the searchable record and the rich hierarchical record would need to be locked in order to update both transactionally.

So, to summarize, in terms of performance, use cases that can be implemented using only key–value access will be the fastest in GemFire. Heavily transactional use cases benefit from rich hierarchical object graphs. Rapidly changing data elements might benefit most from being separate from their slower changing siblings. Search-heavy use cases will likely benefit from separating and flattening the search fields.

Portable Data eXchange Format

GemFire's Portable Data eXchange (PDX) wire format provides a mechanism by which it can serialize data in order to put it on-the-wire, and de-serialize again on the other side of the connection. Every data store needs some kind of serialization format.

The advantages that PDX bring are many. First, it is a language-neutral format. Data that is written into GemFire using the default `ReflectionBasedAutoSerialzer` can be automatically read in C#, for instance. Second, there is sufficient metadata stored in the PDX registry that it is possible to index and query PDX serialized objects in the GemFire server without the need for the domain classes in the server. Third, GemFire can do that indexing and/or querying without having to de-serialize the entire object. It just needs to de-serialize the fields in the predicates.

PDX can also help developers move more quickly because it handles simple schema changes for you. If you add or remove fields in your domain objects, GemFire will help you manage how those changes behave in your application.

For instance, suppose that you start out with a `Customer` class that contains these fields:

```
int    id,
String customerID,
String address1,
String address2,
String phone,
```

You can go ahead and store `Customer` objects into GemFire and write apps that use those fields. Now, suppose that as you develop further you decide that it would be a really good idea to add in a count of the number of `Orders` the customer has outstanding. Now the `Customer` class contains these fields:

```
int    id,
String customerID,
String address1,
String address2,
String phone,
int    order_count
```

GemFire's autoserializer will figure out that you have added the `order_count` to your class. Now you have a whole bunch of data already in GemFire that doesn't have that field. Thus, when GemFire delivers one of these old objects to a new app, it automatically sets the `order_count` to `0`, so the new app knows it needs to do a `select count *` query to fill in the correct `order_count` value. Now when this new app updates the record in GemFire, it will contain the corrected value. But now, you have some old apps still running and new objects in GemFire containing the `order_count` field that they don't know about.

When GemFire delivers a new object to an old app, it will hide the `order_count` field from the old app, and when the old app updates the record to change the `address2` value, the current value of `order_count` will be preserved during the update. We call this feature "automatic backward and forward schema migration."

Handling Dates in a Language-Neutral Fashion

Date representation is possibly the hardest part of trying to make a language-neutral representation of data. The components of Java date and .NET date are actually different. Starting with .NET Frame-

work version 2.0, the default `DateTime` is one whose `Kind` property returns `DateTimeKind.Local` and stores the `timezone` information from the computer on which it was created. The `java.util.Date` class does not store the `timezone` information. That makes sharing date objects between the two languages very difficult. The best idea is to always store dates/times in GemFire in Coordinated Universal Time (UTC), and possibly convert your dates/times to a long containing the number of milliseconds from some epoch such as January 1, 0001. That will make range searches much easier.

Start Slow: Optimize When and Where Necessary

In general, the rule of thumb with performance optimization is "don't optimize until there is a need for more speed." So, don't worry too much up front about what your domain objects and keys look like and how they perform. Design them in such a way that you are comfortable programming with them.

At some later time, you might realize that your use case fits one of the special patterns that we described earlier. Maybe you discover that it is a search-heavy use case, but you didn't know that up front. That is the time when you might need to do some large-scale refactoring. Maybe you decide to copy the searchable fields into a separate `Region` in a flattened form so that the Lucene index engine can give you blazing-fast searches. Of course, now you need to modify your update semantics to update those fields in both places during a transaction. And you will need to do a quick pass through the data already in GemFire to prepopulate those search fields for existing data and build the index.

The only "gotcha" that you need to watch out for is changing the *meaning* or *type* of a field. GemFire cannot deal with a situation in which you have changed the business meaning of a field. Nor can it deal with the fact that you changed the type of a field from an integer to a string. Those are breaking changes for which PDX's magic cannot help you.

CHAPTER 6

Multisite Topologies Using the WAN Gateway

Mike Stolz

You can deploy GemFire clusters in a single site or in a cloud deployment across nearby availability zones. But often the need arises for having two separate clusters in distinct locations. You can meet this need by replicating data between distant sites using GemFire's Asynchronous WAN Gateway, a bidirectional, fault-tolerant, store-and-forward replication subsystem. It is asynchronous because it is intended for use in long-haul replication situations, for example, New York, London, Tokyo. You can build mesh topologies wherein each site replicates data that originates there to all the other sites, or you can limit the locations to which data is replicated.

Example Use Cases for Multisite

The simplest and most often used design pattern for multisite deployment of two clusters is active/passive disaster recovery (DR). DR is still a very popular design pattern, and if you have several business units that update different data sets it is very easy to make this pattern active/active. Let's say you have an equity trading business and a fixed-income trading business. They trade different instruments, so they will never collide with each other. This means that you could host the equity business in one datacenter and the fixed-income business in the other datacenter, with both active, and both backing each other up for DR purposes, as illustrated in Figure 6-1.

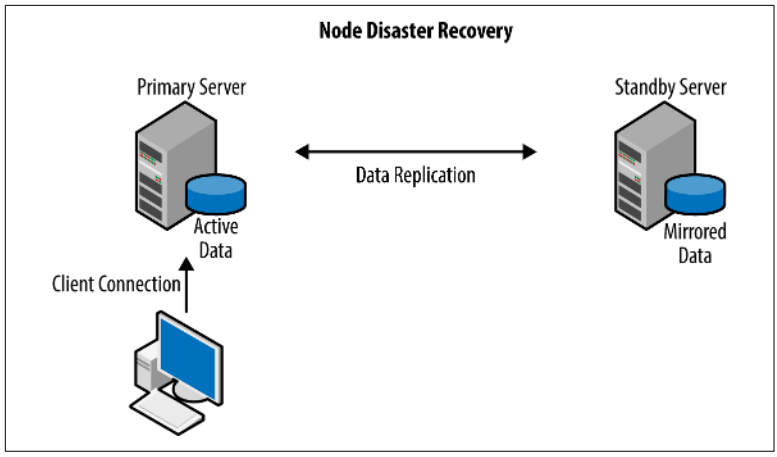

Figure 6-1. A simple bidirectional active/passive WAN Gateway configuration

Design Patterns for Dealing with Eventual Consistency

There are use cases for which the same data needs to be accessible in multiple sites, and possibly even the same data needs to be *updated* in multiple sites. This kind of multisite update capability forces a requirement to program somewhat differently to avoid collisions. Let's examine the design patterns for this now.

Inventory Allocation Pattern

The way this pattern works is that you split up the inventory that you can sell and give a portion of the inventory for every item to each of the venues. You can see a depiction of this patter in Figure 6-2. When a venue runs out of inventory for a particular item, the application might ask other venues for some of their inventory to replenish the empty venue. This pattern is very commonly used for trading municipal bonds. There are several different trading venues, and it is illegal to sell short, so each venue needs to sell only the inventory that has been allocated to it.

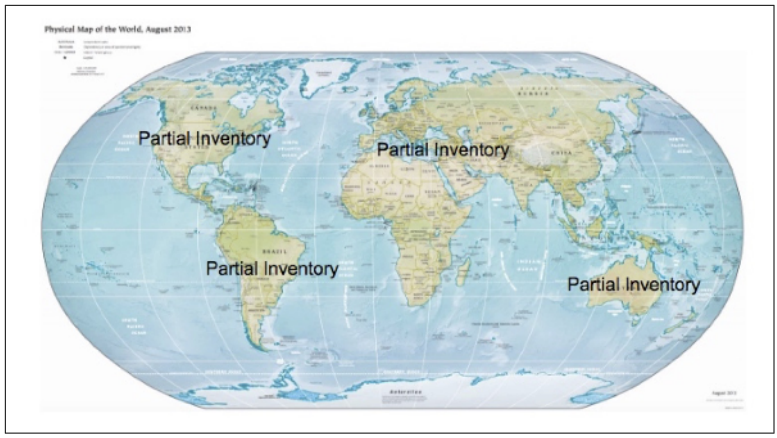

Figure 6-2. Inventory allocated to four different venues around the world (source: http://bit.ly/2k4YIHy)

Order-and-Fill Pattern

The order-and-fill pattern is commonly used by trading exchanges around the world. The idea is that you don't actually buy or sell things. You request to have them bought or sold on your behalf. So, the transaction actually happens under the control of a single transaction manager rather than having a multimaster situation in which applications in different sites are updating data freely as in the inventory allocation pattern. To implement this pattern, you need to put an order into the system. That order tells the transaction manager what you want to buy or sell and how much, and might contain other instructions including a time limit.

One of the additional bits of information contained in the order is who the order is being placed on behalf of, and how to notify that application that the order has been filled. The transaction manager itself can be code that runs inside the cluster. The orders can be sent to all possible venues where processing could occur, and the transaction manager will be running in just one venue at any given moment. When the transaction manager sees your order, it will process it, and it will reply to the requesting application with the results of processing. That reply goes through a "Response" `Region` with the key being the identity of the particular application or user that is awaiting the response. The "Response" `Region` is replicated back to the originator over the WAN Gateway in the opposite way that the order was replicated to the transaction manager. After the response

has been sent the order can be marked "filled." Responses can either be saved or destroyed as soon as they have been received.

A common extension of this order-and-fill pattern is a follow-the-sun pattern (Figure 6-3).

Figure 6-3. Follow-the-sun pattern (source: http://bit.ly/2Ac6LsI)

The transaction manager's responsibility can be moved from one location to another by passing a token on a "Control" `Region`, which identifies which venue should be processing transactions now. That token is replicated to all the venues over the WAN Gateway just as the orders are. When the various venues see the token arrive, they examine it to determine which venue has been elected to be the transaction manager.

Apology-Based Computing

This is the pattern that Max Feingold refers to when he says:

> At global scale, getting the truth is really, really expensive.

A little story is in order here. You have a business that sells sneakers online and guarantees same-day shipping on any pair of sneakers ordered anywhere in the world. Your transaction model is based on a single transaction manager in one site, so you always know *exactly* how much inventory you have.

One day you sell the last pair of sneakers of a certain kind, and as the forklift goes to put it on the truck it rolls over it and crushes it. Now you *must* have a way to apologize to your customer because

you are not going to be able to ship the same day. Well, if you need a way to apologize anyway, why not use that as the mechanism for fixing a problem when you actually do oversell? This is what is called *apology-based computing*. It doesn't work for everyone, but it might be good enough for selling sneakers online.

Gateway Components: Sender/Receiver

The Gateway Sender is the component that you configure for a Region to cause data to be sent to another cluster. The Sender is usually configured to operate in parallel for Partitioned Regions. This is in order to minimize the copying of data to a particular node where a serial sender would be. On Replicated Regions there is no notion of a primary, and all data is copied to all nodes, so it only makes sense to use a serial sender on Replicated Regions.

The Receivers are configured at the other end of the Gateway, and it is wise to configure at least two of them for high availability. By default, the Receiver will look at the version of each object received and compare it to the version of the object already present (or the tombstone if the object has been destroyed recently). If the incoming object is the same or older version it is automatically discarded.

Queue Persistence

WAN Gateways can overflow their data to a DiskStore so that we don't consume all available memory for the Gateway Queue. Used in conjunction with conflation this can allow a Gateway to be left running even in the event that the receiver is down for a prolonged period of time.

CHAPTER 7

Querying, Events, and Searching

Charlie Black, Mike Stolz, and Gregory Green

Object Query Language

The query language that is built in to GemFire is called Object Query Language (OQL). The OQL language was developed by the Object Data Management Group as a standard for object databases. Before we start on the virtues of OQL, we must talk about why all GemFire applications are not based on OQL.

When we issue an OQL statement, we can't be 100% certain which hosts contain the data that we are looking for. Therefore, the OQL query is sent to all members within the distributed system and is run on all CPUs; the results from all of the hosts have to be combined and presented in the result set. This might seem like a lot of work; however, OQL can be faster than disk-based architectures because GemFire normally operates out of memory-based storage rather than disk.

So, to get extreme scale we use GemFire for what it does best: key–value storage. Then, when we can't design our way to using what GemFire does best, we can use OQL and treat it like an object database. So, what does OQL look like? Let's see:

```
SELECT person FROM /people person
WHERE person.firstName = 'Charlie'
```

This query uses the `people` region, iterates through the values, and finds all of the people whose first name is `Charlie`. The interesting part comes into play when we begin thinking about the object side

of this equation. We can rewrite the query to iterate over the values of the hashmap like this:

```
SELECT person FROM /people.values person
WHERE person.firstName = 'Charlie'
```

How about a query for all of the keys for which the person's first name is `Charlie`:

```
SELECT entry.key FROM /people.entries entry
WHERE entry.value.firstName = 'Charlie'
```

OQL Indexing

Optimizing your queries with indexes requires a cycle of testing and tuning. Poorly defined indexes can degrade the performance of your queries instead of improving it. This section gives guidelines for index usage in OQL.

When creating indexes, keep in mind that indexes incur maintenance costs because they must be updated when the indexed data changes. An index that is updated frequently and is not used for queries very often can require more system resources than using no index at all.

You also need to be cognizant that indexes use memory. The index is always stored in memory even if the data being indexed eventually overflows to disk.

If you are creating multiple indexes on the same region, first define your indexes and then create the indexes all at once to avoid iterating over the region data multiple times.

Whenever possible, provide a hint to allow the query engine to prefer a specific index. You can use the hint keyword to allow the OQL query engine to prefer certain indexes. For cases in which one index is hinted in a query, the query engine filters off the hinted index (if possible) and then iterates and filters from the resulting values; for example:

```
<HINT 'IDIndex'> SELECT * FROM /Portfolios p WHERE p.ID > 10
    AND p.owner = 'XYZ'
```

If multiple indexes are added as hints, the query engine will try to use as many indexes as possible while giving a preference for the hinted indexes. Here's an example:

```
<HINT 'IDIndex', 'OwnerIndex'>
SELECT * FROM /Portfolios p WHERE
p.ID > 10 AND p.owner = 'XYZ' AND p.value < 100
```

OQL indexes and queries can perform cross-region joins (equi-joins), and can deal with nested object hierarchies and collections like maps.

To join across multiple regions, identify all equi-join conditions. Then, create as few indexes for the equi-join conditions as you can while still joining all regions. If there are equi-join conditions that redundantly join two regions (to more finely filter the data, for example), then creating redundant indexes for these joins will *negatively impact performance*. Create indexes only on one equi-join condition for each region pair. Consider this example query:

```
SELECT DISTINCT *
FROM /investors inv, /securities sc, /orders or,
inv.ordersPlaced inv_op, or.securities or_sec
WHERE inv_op.orderID = or.orderID AND or_sec.secID = sc.secID
```

Here all conditions are required to join the regions, so you would create four indexes, two for each equi-join condition:

```
inv_op.orderID and or.orderID
or_sec.secID and sc.secID
```

To assist with the quick lookup of multiple values in a Map (or HashMap) type field, you can create an index (sometimes referred to as a "map index") on specific (or all) keys in that field. For example, you could create a map index to support the following query:

```
SELECT * FROM /users u WHERE u.name['first'] = 'John'
    OR u.name['last'] = 'Smith'
```

Continuous Queries

Continuous queries are a mechanism whereby notifications can be sent to interested users when data changes in the cluster. Continuous queries look just like the OQL queries above except that continuous queries don't support projections and they can refer to only a single region. Here's a simple example of a good candidate for a continuous query:

```
SELECT * FROM /trades t
WHERE t.notional_amount > 1000000
```

This will return all trades whose notional amount is greater than one million, and then it will continue to be evaluated every time a trade is inserted or updated. If the notional amount of the trade is greater than one million you will be updated for that trade. You will also be notified if a trade is modified and no longer meets the `WHERE` clause criteria. So, if a trade *was* more than one million but drops under, you will receive a notification for both situations.

Listeners, Loaders, and Writers

GemFire has a very powerful built-in eventing model. There are callbacks that you can implement for handling cache misses (`CacheLoader`); for validating and accepting or rejecting data before it gets into the cache (`CacheWriter`); and to be notified right after data is inserted, updated, or destroyed (`CacheListener`).

There are also similar callbacks that work at the transaction level rather than the individual update level. You can validate a transaction and accept it or reject it by implementing the `TransactionWriter` callback. You can perform follow-up actions after a transaction is successful by implementing the `TransactionListener` callback.

One additional very useful callback that is used often is the `AsyncEventListener`. It is attached to an `AsyncEventQueue` and is used to implement lazy write-behind to some external backing store like a legacy database. This gives you the possibility of potentially batching the updates to the database rather than doing individual inserts. The `AsyncEventQueue` is a fault tolerant construct with primaries and secondaries distributed across the cluster. The `AsyncEventListener` is fault-tolerant in nature. There is a primary sender for some of the keys in the system, and there is a backup sender for the same keys running on another member. In the event that the primary fails, the secondary will take over sending data to the legacy store. Any events that were already in the queue when the primary failed will be marked with a "possible-duplicate" flag because there is no way to know for sure whether the backing store has already received those events.

Lucene Search

The embedded Lucene search feature allows users to perform various kinds of searches for region data using Apache Lucene. Apache Lucene is an open source search engine. It supports performing simple, wildcard, and fuzzy searches. Searching for region entries by name (for which people will not typically know the exact first name or last name) is a good use case for using Lucene.

Lucene searches can match text fields in region entries based on single character wildcard using "?". The query `N??A` will match the words "NYLA" as well as "NINA." Use "*" to perform multiple-character wildcard-based searches. The query `*EEN` will match any word that ends with "EEN". Lucene supports fuzzy searches using "~". The fuzzy query `Josiah~` will match words with similar spelling. Lucene also supports range searches and Boolean operations. See the Pivotal GemFire product documentation and the Apache Lucene documentation for a full range of its capabilities.

Apache Lucene indexes are the core component to facilitate searches. Lucene examines all of the words for region data based on the index definition. The index structure will store words along with pointers to where the words can be found. The indexes will be updated asynchronously whenever region entries are saved. Indexing asynchronously is used to optimize region `put` operations.

The embedded Lucene search feature provides search support through a Java API or the `gfsh` command-line utility:

```
gfsh>search lucene --region=/Users --keys-only=true
--name=userIndex
--queryStrings="+firstName:Greg +lastName:Green"
--defaultField="lastName"
```

This `gfsh` Lucene search query will search for region records based on the `firstName` and `lastName` fields on the `userIndex`. This search looks for region entries where the `firstName` field is Gregory and `lastName` is Green. The `+` means that both `firstName` and `lastName` field matches are required.

CHAPTER 8

Authentication and Role-Based Access Control

Swapnil Bawaskar

In this chapter, we look at using role-based access control (RBAC) in GemFire and securing the communication between various components by using Secure Sockets Layer (SSL).

Authentication and Authorization

Before we dig into how authentication and authorization works, let's try to look at the operations that you can perform in GemFire that would need to be authenticated and authorized.

Background

In GemFire you can start/stop locators and servers and you can alter their runtime to change log-level as well as other administrative actions. You are creating regions to store your data, defining indexes, and defining disk stores to persist your data; you then actually insert data, and access and query it.

We can broadly classify these actions into two categories based on the type of resource being worked on. Starting servers, altering runtime, and defining disk stores are operations that involve working on your `CLUSTER`, whereas `put()`, `get()` and queries work on `DATA` as the resource. The security framework classifies all operations in these two major categories.

Within each resource classification, we can further classify all commands as either accessing the resource (`READ`), writing to the resource (`WRITE`), or making changes to the resource (`MANAGE`). For example, `list members` just accesses the `CLUSTER` resource, whereas `stop server` manages it.

Table 8-1 shows classifications of some of the commands. You can find a comprehensive list in the product documentation.

Table 8-1. Classifications of operations

Read	Write	Manage
Cluster		
show metrics export logs list members	change log level	alter runtime start server shutdown
Data		
query region.get/getAll	region.put region.replace	create region destroy region

This Resource:Operation tuple forms the basic unit for authorization.

Implementation

GemFire's security framework is pluggable, enabling you to integrate with your existing infrastructure. The only interface you need to implement is `SecurityManager` from the `org.apache.geode.security` package. An implementation of `SecurityManager` only needs to implement the following method:

```
public Object authenticate(Properties)
```

This makes authorization optional. When this method is invoked, the `Properties` object that is passed in will have two entries, `security-username` and `security-password`, that you can use to authenticate the user with your existing infrastructure.

The authorize method, if you choose to implement it, has two parameters that are passed in the `principal` (the object that you returned from the `authenticate` method) and the `ResourcePermission` tuple for the requested operation:

```
boolean authorize(Object principal,
    ResourcePermission permission)
```

After you implement them, you need to tell GemFire about your `SecurityManager` by adding the following line to the *gemfire.properties* file:

```
security-manager=com.mycompany.MySecManager
```

Your client application will have to supply its credentials to the server in order to perform any operation. The way to do that is to implement the `AuthInitialize` interface in the `org.apache.geode.security` package and then make sure that the `getCredentials()` method returns properties with at least two entries: `security-username` and `security-password`.

Again, your client needs to tell the system about your `AuthInitialize` implementation by using the `security-client-auth-init` gemfire property.

Fine-Grained Authorization

In order to provide fine-grained control over the `CLUSTER` resource, it has been broken down further into `DISK`, `GATEWAY`, `QUERY`, `LUCENE`, and `DEPLOY`, allowing you to give only some users the ability to create WAN gateways, for example. All of these subresources also get finer control over `READ`/`WRITE`/`MANAGE` operations. Examples of permissions for some commands are as follows:

```
CLUSTER:MANAGE:DISK: create disk-store, alter disk-store
CLUSTER:WRITE:DISK: write to the disk store
CLUSTER:MANAGE:QUERY: create index, destroy index
CLUSTER:READ:QUERY: list indexes
```

For a full list, look at the product documentation.

In our discussion earlier, we talked only about operations working on one resource; however, it is possible to have operations working on multiple resources. Consider, for example, the creation of a persistent region. This operation acts on two resources, `DATA` and `DISK`, therefore you will need `DATA:MANAGE` as well as `CLUSTER:WRITE:DISK` permissions in order to create a persistent region.

SSL/TLS

To encrypt communication over the wire, let's look at the various channels of communication in the system:

1. Locator-to-locator
2. Locator-to-server
3. Server-to-server
4. Client-to-locator
5. Client-to-server
6. Between WAN gateways
7. REST API and Pulse
8. JMX communication

In many cases, you would want to secure only a subset of these communication channels. For example, your cluster might be protected by a firewall, so securing server-to-server communication might not be required, but your clients can be connecting to the servers from outside, in which case you would want to secure the client-to-server communication.

You can pick and choose which communication channels should be secured. You can specify the components that should be secured using the `ssl-enabled-components` GemFire property. Following are the values that this property accepts:

1. `locator`: for 1, 2, and 4 in the previous list
2. `server`: for 5
3. `cluster`: for 3
4. `gateway`: for 6
5. `web`: for 7
6. `jmx`: for 8
7. `all`: for securing everything

CHAPTER 9

Pivotal GemFire Extensions

John Knapp and Jagdish Mirani

GemFire-Greenplum Connector

Even though GemFire is built for rapid response time, Pivotal also produces an analytic relational database known as Greenplum. Greenplum was designed to provide analytic insights into large amounts of data. It was not designed for real-time response. Yet, many real-world problems require a system that does both. At Pivotal, we use GemFire for real-time requirements and the GemFire-Greenplum Connector to integrate the two.

The GemFire-Greenplum connector is a bidirectional, parallel data transfer mechanism. It is based on the Pivotal Greenplum Database's ability to view data residing outside the normal Greenplum storage (external tables) and Greenplum's parallel data transfer channel (gpfdist). Here's a brief description of how this works.

The GemFire-Greenplum Connector (GGC) is an extension package built on top of GemFire that maps rows in Greenplum tables to plain-old Java objects (POJOs) in GemFire regions. With the GGC, the contents of Greenplum tables now can be easily loaded into GemFire, and entire GemFire regions likewise can be easily consumed by Greenplum. The upshot is that data architects no longer need to spend time hacking together and maintaining custom code to connect the two systems.

GGC functions as a bridge for bidirectionally loading data between Greenplum and GemFire, allowing architects to take advantage of

the power of two independently scalable massively parallel processing (MPP) data platforms while greatly simplifying their integration. GGC uses Greenplum's external table mechanisms to transfer data between all segments in the Greenplum cluster to all of the GemFire servers in parallel, preventing any single-point bottleneck in the process.

Supporting a Fraud Detection Process

Greenplum's horizontal scalability and rich analytics library (MADlib, PL/R, etc.) help teams quickly iterate on anomaly detection models against massive datasets. Using those models to catch fraud in real time, however, requires using them in an application. Depending on the velocity of data ingested through that application, a "fast data" solution might be required to classify the transaction as fraudulent in a timely manner. This activity involves a small dataset and real-time response. By connecting Greenplum and GemFire together, we can provide this fast-data solution hosted inside GemFire and have it be informed by its connection to the deep analytics performed in Greenplum.

Pivotal Cloud Cache

Pivotal Cloud Cache (PCC) is a caching service for Pivotal Cloud Foundry (PCF) powered by Pivotal GemFire. The PCF platform fosters the adoption of modern approaches to building and deploying software. Caching is finding new relevance in modern, cloud-native, distributed application architectures. As applications are split up into smaller separately deployed components, often on different systems in different locations, network latencies can severely limit application performance. Caching data locally can reduce network hops, reducing the impact from network latencies.

The sheer number of components in modern architectures introduces many points of failure. Adding highly available caches at critical points in the topology can dramatically increase the overall availability of the system.

Many of the core caching features in PCC are provided by technology that is already battle tested in GemFire. Many of the PCC features that can be traced back to GemFire are covered in this book, including high availability across servers and availability zones, hor-

izontal scalability and data partitioning and colocation, continuous query, subscribing to events by registering interest (pub/sub), role-based security, and more.

If you're interested in caching, and you're already running PCF, you're likely to be interested in PCC. PCC's unique purpose as a service on PCF is the result of significant investment in the platform integration of PCC. Services on PCF are designed to be easy for operators to set up and configure, and easy for developers to install and integrate with their apps. This is partly accomplished by delivering services that are opinionated, and preconfigured service plans by use case. By doing this we have eliminated a lot of setup and configuration steps. This use-case oriented approach is reflected in how PCC delivers the look-aside caching pattern and HTTP Session State Caching. Other patterns like in-line caching and multisite will be equally opinionated and therefore easy to configure.

PCC Service Instances On-Demand

A key goal of PCC, or any PCF service, is to make it easy for developers to self-serve on-demand service instances. However, uncontrolled utilization of resources can lead to escalating IT infrastructure costs. What's needed is a managed provisioning environment that provides developers with easy self-service access to caching as a resource, but also allows operators to maintain control through operator-defined plans, upgrade rules, and quotas. Operators can customize service plan definitions to support internal chargeback packages and enforce resource constraints.

On-Demand Service Broker

PCC's on-demand provisioning is built on the On-Demand Service Broker API, an abstraction that makes it easier to integrate and use services on PCF. The API provides support for provisioning, catalog management, binding, and updating instances.

Services like PCC are made available via the PCF Marketplace, which provides developers with a catalog of add-on services to enhance, secure, and manage applications.

On-demand services enable the flexibility to create instances in a scalable and cost-effective way. When an operator deploys the service, they do not preallocate virtual machine (VM) resources for service instances. Instead, they define an allowable range of VM

memory and CPU sizes, set quotas on the number of service instances that can be started, and create a dedicated network on the Infrastructure as a Service to host any required number of service instance VMs.

When a developer requests a service instance, it is provisioned on-demand by BOSH, a tool chain for release engineering, deployment, and lifecycle management of services. BOSH is a vital part of the PCF platform. The developer selects the service plan, and BOSH dynamically creates new dedicated VMs for the instance configured as per the service plan.

Cloud Agnostic

After a BOSH release is created, it's compatible with multiple clouds. BOSH abstracts the specifics of each cloud provider with a Cloud Provider Interface (CPI) abstraction. BOSH users now can realize the benefits of each provider, *without* in-depth knowledge of each.

High Availability

The BOSH layer monitors service nodes, removes unresponsive instances, and restores capacity by spinning up new instances or adds capacity on demand. PCC supports the notion of multiple availability zones mapped to GemFire redundancy zones, which allows smooth recovery from server failures.

Integration with Logging and Monitoring Services

For gaining visibility into the details of the PCC service operation, PCC streams its logs to PCF's Logregator Firehose. Standard monitoring and logging tools that integrate to the Firehose via a nozzle can be used for displaying this operational data in dashboards.

CHAPTER 10

More Than Just a Cache

Mike Stolz

Session State Cache

Web and mobile apps maintain the illusion of a user session even though they are connected over a sessionless communication protocol. This is achieved by caching the session state in a data layer separate from the app servers so that the load balancer is free to move load to any app server; it will still have access to the session state at all times.

Compute Grid

There are many use cases for which moving the compute to the data instead of moving the data to the compute can cause tremendous performance improvements. An example of such a use case is a financial risk-management system. In one benchmark of a position-keeping system calculating positions on 20 books of 10,000 European options, starting with five million trades per book, with 20,000 market data updates per second and 2,000 new trades per second, the mean time to "price the book" went down from 2.67 seconds using a separate data grid and compute grid to 0.035 seconds when the compute was done *in situ* with the data. That is an improvement in performance of 76 times, with half as much hardware just by moving the compute to the data. The key to this kind of performance gain is the use of GemFire's server-side data-aware function execution service. There are two ways that this service works. In both cases you program your custom function to act only on data

that is local to the member that it is running on, and you deploy it to the servers in the cluster. Then, you invoke it from a server or client.

In the "data-aware" function mechanism, when you invoke the function, you are invoking it on all of the members that host a specified `Region`, and you pass in a filter, which is a list of keys that are used to cause the function to run only on the members that are primaries for those keys.

There is also a notion of executing a function on specific members of the cluster, or a group of members who are part of a member group, or all members in the cluster.

In any of these cases, each of those members should be programmed to operate only on the data that is local to the member on which the function is running.

GemFire as System-of-Record

GemFire offers robust features to enable usage as the system-of-record for applications. A system-of-record is the authoritative source of truth for a given dataset. The implications of system-of-record are a set of features required to make the data safe. These features are high availability, disk durability, business continuity, snapshots, backup, incremental backup, and restore.

All of those features are present in GemFire. Let's take each of the features one-by-one and examine the implementation details.

High Availability

GemFire high availability is achieved via synchronous replication between the primary and backup members. For Partitioned Regions where the data is sharded across the members of the cluster, there is a notion of a primary for every object, and some number of backups. Every member is primary for some of the objects, and every member is backup for other objects. Because the replication between primaries and backups is synchronous and is completed before returning to the caller of `put(key, value)`, there is no chance of silent data loss in the event of primary failure.

Disk Durability

In keeping with the shared-nothing architecture of GemFire, both primaries and backups are saved to separate locally attached storage on separate hosts in separate redundancy zones, which are usually mapped to separate Infrastructure as a Service availability zones so there is no chance of a single point of failure causing data loss. For system-of-record use cases, it is recommended that you configure three redundancy zones and three copies of the mission-critical data.

Business Continuity

In addition to the availability zones, you also can configure GemFire to provide further business continuity in the event of a major disaster. This disaster recovery functionality is provided by the GemFire WAN Gateway, which is designed for replication of data between distant sites. Although it is important to note that it is asynchronous, when used in conjunction with multiple availability zones the likelihood of data loss due to the buffering in the WAN Gateway is still quite low.

It is also important to note that the WAN Gateway is able to be bidirectional. This is a critical feature for use in both active/active use cases and active/passive use cases. The active/active need for bidirectionality is obvious. Both sites are active and backing each other up. The active/passive use case also benefits from the bidirectionality of the WAN Gateway. When the primary site fails, and everything cuts over to the backup site, you want to queue up changes made on the backup until the primary comes back to life. This way when the primary comes back, it automatically gets brought back up to date on anything it missed while it was away. So GemFire's WAN Gateway makes *fail-back* just as easy as failover. Anybody can do failover, but fail-back is difficult without a feature like this.

Snapshots

GemFire has the ability to take a point-in-time snapshot of data in any `Region`, and the ability to import that data into the same or another GemFire system. When `gfsh` takes a snapshot, it also takes a copy of the PDX registry so that you will be able to apply the snapshot correctly in another system. In addition, there is a public API

that allows you to read the contents of the snapshot file, and do with it anything you may want, including modify the records before inserting them into another system.

Backup, Incremental Backup, and Restore

GemFire has built-in full backup, incremental backup, and restore mechanisms. For each member with persistent data, a full backup includes a full image of all of the saved state on all members in the cluster. An incremental backup saves the difference between the last backup and the current data. An incremental backup copies only op-logs that are not already present in the baseline directories for each member.

All these enterprise features make Apache Geode and its commercial counterpart GemFire suitable for even the most mission-critical use cases in many of the world's largest enterprises.

About the Authors

Mike Stolz is the product lead for Pivotal GemFire and is based on Long Island, New York. As product lead, he defines a roadmap that steers the product toward value, engages with a range of stakeholders both inside and outside the company to both listen and inform, shapes expectations of what the product is and is not good for, and engages deeply with the product managers internally to ensure Pivotal is delivering on the vision.

He was the first user to deploy GemFire in production back in 2003 when he was Director and Chief Architect for Fixed Income, Currencies, Commodities, Liquidity, and Risk Technology at Merrill Lynch. In fact, as of this writing, the global market data system that he and his team deployed way back then is still in continuous operation and is still running on the original version of GemFire that it was initially deployed on. It is the longest running GemFire based app in the world.

Mike took an early retirement package from Merrill Lynch and joined GemStone Systems, the creators of GemFire, as VP of Architecture in 2007 and has served in many roles on the GemFire team ever since.