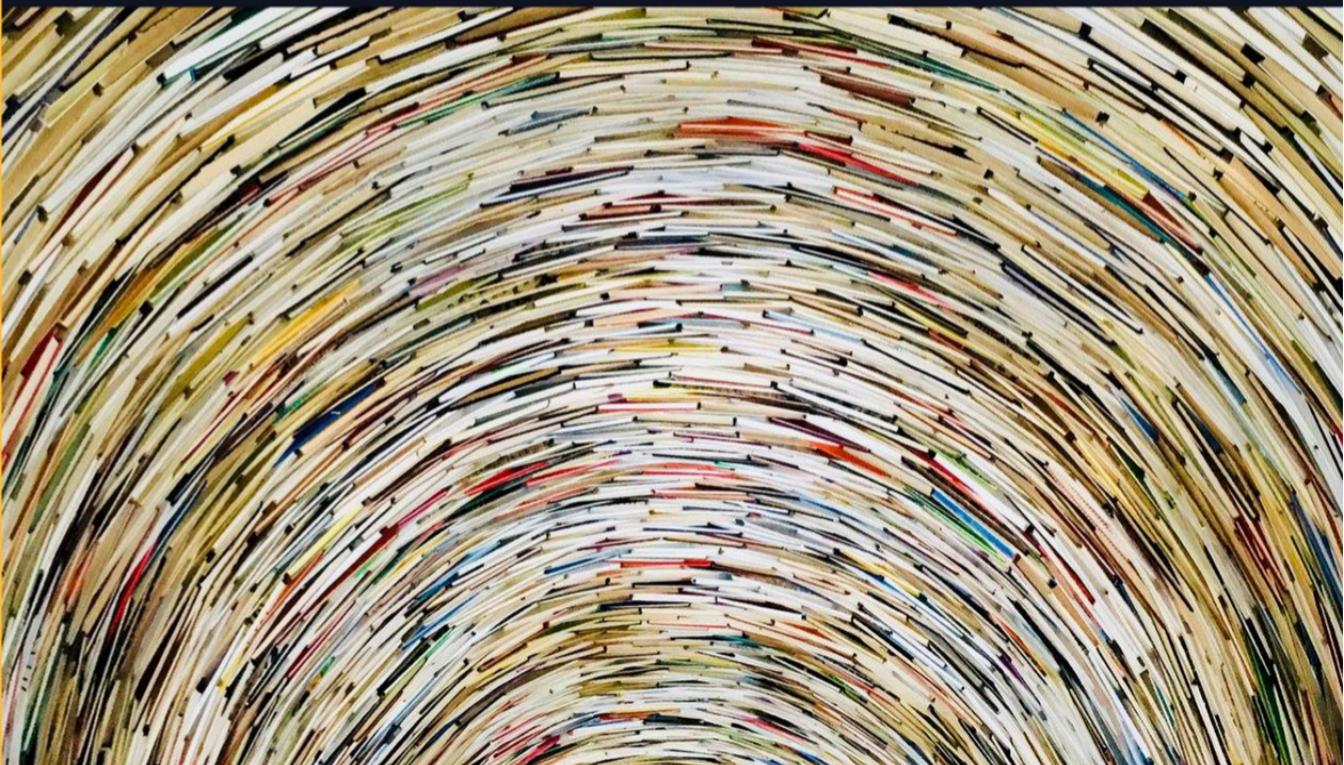


# THE APACHE IGNITE BOOK

The next phase of the distributed systems



The complete guide to learning everything  
you need to know about Apache Ignite

BY SHAMIM BHUIYAN & MICHAEL ZHELUDKOV

# The Apache Ignite book

The next phase of the distributed systems

Shamim Bhuiyan and Michael Zheludkov

This book is for sale at <http://leanpub.com/ignitebook>

This version was published on 2020-01-13

ISBN 978-0-359-43937-9



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2020 Shamim Bhuiyan

## **Tweet This Book!**

Please help Shamim Bhuiyan and Michael Zheludkov by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

Just purchased "The Apache Ignite Book" <https://leanpub.com/ignitebook> by @shamim\_ru  
#ApacheIgnite #IMDG #NoSQL #BigData #caching

*To my Mother & Brothers, thank you for your unconditional love.*

# Contents

Preface . . . . .	i
What this book covers . . . . .	i
Code Samples . . . . .	ii
Readership . . . . .	iii
Conventions . . . . .	iii
Reader feedback . . . . .	iv
About the authors . . . . .	v
Acknowledgments . . . . .	vi
Chapter 1. Introduction . . . . .	1
Chapter 2. Getting started with Apache Ignite . . . . .	7
Installing and setting up Apache Ignite . . . . .	9
Building from source code . . . . .	11
Running multiple instances of the Apache Ignite in a single host . . . . .	13
Running Apache Ignite in Docker containers . . . . .	14
Using Apache Ignite SQLLINE CLI . . . . .	17
Meet with Apache Ignite SQL engine: H2 database . . . . .	22
Using a universal SQL client IDE to work with Apache Ignite . . . . .	28
First Java application . . . . .	35
Apache Ignite thin client . . . . .	42
Using REST API for manipulating the Apache Ignite caches . . . . .	47
Configuring a multi-node Ignite cluster in different hosts . . . . .	53
A simple checklist for beginners . . . . .	55
Summary . . . . .	58
What's next? . . . . .	58
Chapter 3. Apache Ignite use cases . . . . .	59

## CONTENTS

Caching for fast data access . . . . .	60
HTAP . . . . .	61
High-volume transaction processing . . . . .	64
Fast data processing . . . . .	65
Lambda architecture . . . . .	66
Resilient web acceleration . . . . .	68
Microservices in distributed fashion . . . . .	69
Cache as a service . . . . .	70
Big Data accelerations . . . . .	71
In-memory machine learning . . . . .	72
In-memory geospatial . . . . .	74
Cluster management . . . . .	74
Summary . . . . .	75
What's next? . . . . .	75
<b>Chapter 4. Architecture deep dive . . . . .</b>	<b>76</b>
Functional overview . . . . .	77
Understanding the cluster topology: shared-nothing architecture . . . . .	78
Client and server node . . . . .	79
Embedded with the application . . . . .	81
Client and the server nodes in the same host . . . . .	82
Running multiple nodes within single JVM . . . . .	83
Real cluster topology . . . . .	84
Data partitioning in Ignite . . . . .	85
Understanding data distribution: DHT . . . . .	86
Rendezvous hashing . . . . .	90
Replication . . . . .	95
Master-Slave replication . . . . .	95
Peer-to-peer replication . . . . .	97
Partitioned mode . . . . .	98
Replicated mode . . . . .	101
Local mode . . . . .	102
Near cache . . . . .	102
Partition loss policies . . . . .	104
Caching strategy . . . . .	105
Cache a-side . . . . .	106
Read through and write through . . . . .	107
Write behind . . . . .	108

## CONTENTS

Ignite data model . . . . .	109
CAP theorem and where does Ignite stand in? . . . . .	112
Apache Ignite life cycle . . . . .	115
Memory-Centric storage . . . . .	118
Durable memory architecture . . . . .	124
Page . . . . .	124
Data Page . . . . .	125
Index pages and B+ trees . . . . .	126
Segments . . . . .	127
Region . . . . .	128
Data eviction . . . . .	130
Page based eviction . . . . .	130
Entry based eviction . . . . .	132
Data expiration . . . . .	134
Ignite read/write path . . . . .	137
Native persistence . . . . .	140
Write-Ahead-Log (WAL) . . . . .	143
Checkpointing . . . . .	155
Baseline topology . . . . .	157
Automatic cluster activation . . . . .	161
Split-brain protection . . . . .	163
Fast rebalancing and it's pitfalls . . . . .	165
Tool to control baseline topology . . . . .	168
Automate the rebalancing . . . . .	169
Discovery and communication mechanisms . . . . .	170
Discovery . . . . .	170
Communication . . . . .	178
Cluster groups . . . . .	179
Predefined cluster group . . . . .	180
Cluster group with node attributes . . . . .	181
Custom cluster group . . . . .	182
Data collocation . . . . .	183
Compute collocation with data . . . . .	184
Protocols and clients . . . . .	186
Resilience & Automatic failover . . . . .	187
Ignite client node . . . . .	188
Ignite thin client . . . . .	188
JDBC client . . . . .	189

## CONTENTS

Ignite Rest client . . . . .	189
Multi data center replication . . . . .	190
Key API's . . . . .	192
Summary . . . . .	193
What's next? . . . . .	194
<b>Chapter 5. Intelligent caching . . . . .</b>	<b>195</b>
Smart caching . . . . .	197
Caching best practices . . . . .	198
Design patterns . . . . .	199
Basic terms . . . . .	200
Database caching . . . . .	200
Hibernate caching . . . . .	203
MyBatis caching . . . . .	228
Memoization . . . . .	239
Web session clustering . . . . .	253
Classic solution 1: using a sticky session . . . . .	255
Classic solution 2: store sessions in DB . . . . .	255
Classic solution 3: use only stateless service . . . . .	256
Distributed web session clustering . . . . .	256
Prepare the caches correctly . . . . .	269
Summary . . . . .	272
What's next? . . . . .	272
<b>Chapter 6. Database . . . . .</b>	<b>273</b>
Ignite tables & indexes configuration . . . . .	275
Ignite queries . . . . .	277
SQL queries . . . . .	279
Query API . . . . .	297
SqlFieldsQuery . . . . .	298
SqlQuery . . . . .	307
How SQL queries works in Ignite . . . . .	308
Cache queries . . . . .	322
Scan queries . . . . .	324
Text queries . . . . .	329
Affinity collocation based data modeling . . . . .	334
Collocated distributed joins . . . . .	337
Non-collocated distributed joins . . . . .	340

## CONTENTS

Spring Data integration . . . . .	342
Apache Ignite with JPA . . . . .	355
Persistence . . . . .	369
Native persistence . . . . .	370
Persistence in 3 <sup>rd</sup> party database (MongoDB) . . . . .	378
Transaction . . . . .	394
Ignite transactions . . . . .	397
Transaction commit protocols . . . . .	398
Node types (NearNode/Remote/DHT) . . . . .	400
Concurrency Modes and Isolation Levels . . . . .	402
MVCC . . . . .	405
Performance impact on transaction . . . . .	406
Summary . . . . .	407
What's next? . . . . .	407
<b>Chapter 7. Distributed computing . . . . .</b>	<b>408</b>
Compute grid . . . . .	410
Distributed Closures . . . . .	412
MapReduce and Fork-join . . . . .	419
Per-Node share state . . . . .	427
Distributed task session . . . . .	436
Fault tolerance and checkpointing . . . . .	440
Collocation of computation and data . . . . .	451
Job scheduling . . . . .	457
Service Grid . . . . .	460
Developing services . . . . .	462
Cluster singleton . . . . .	467
Service management and configuration . . . . .	471
Developing microservices in Apache Ignite . . . . .	473
Summary . . . . .	484
What's next? . . . . .	484
<b>Chapter 8. Streaming and complex event processing . . . . .</b>	<b>485</b>
Kafka Streamer . . . . .	488
IgniteSinkConnector . . . . .	489
IgniteSourceConnector . . . . .	498
Camel Streamer . . . . .	510
Direct Ingestion . . . . .	512

## CONTENTS

Mediated Ingestion . . . . .	518
Flume sink . . . . .	522
Storm streamer . . . . .	533
Summary . . . . .	546
What's next? . . . . .	546
<b>Chapter 9. Accelerating Big data computing . . . . .</b>	<b>547</b>
Ignite for Apache Spark . . . . .	548
Apache Spark – a short history . . . . .	549
Ignite RDD . . . . .	552
Ignite DataFrame . . . . .	567
Summary . . . . .	574
What's next? . . . . .	575
<b>Chapter 10. Management and monitoring . . . . .</b>	<b>576</b>
Managing Ignite cluster . . . . .	577
Configuring Zookeeper discovery . . . . .	577
Managing Baseline topology . . . . .	582
Monitoring Ignite cluster . . . . .	594
VisualVM . . . . .	596
Grafana . . . . .	600
Summary . . . . .	612

# Preface

Apache Ignite is one of the most widely used open source memory-centric distributed, caching, and processing platform. This allows the users to use the platform as an in-memory computing framework or a full functional persistence data stores with SQL and ACID transaction support. On the other hand, Apache Ignite can be used for accelerating existing Relational and NoSQL databases, processing events & streaming data or developing Microservices in fault-tolerant fashion.

This book addressed anyone interested in learning in-memory computing and distributed database. This book intends to provide someone with little to no experience of Apache Ignite with an opportunity to learn how to use this platform effectively from scratch taking a practical hands-on approach to learning.

## What this book covers

**Chapter 1. Introduction:** gives an overview of the trends that have made in-memory computing such important technology today. By the end of this chapter, you will have a clear idea of what Apache Ignite is and why use Apache Ignite instead of others frameworks like *HazelCast*, *Ehcache*?

**Chapter 2. Getting started with Apache Ignite:** is about getting excited. This chapter walks you through the initial setup of an Ignite database and running of some sample application. You will implement your first Ignite application to read and write entries from the Cache at the end of the chapter. Also, you will learn how to install and configure an SQL IDE to run SQL queries against Ignite caches and use Apache Ignite Thin client to working with the Ignite database.

**Chapter 3. Apache Ignite use cases:** discusses various design decisions and use cases where Ignite can be deployed. These use cases detailed and explained through the rest of the book.

**Chapter 4. Architecture deep dive:** covers Ignite's internal plumbing. This chapter has a lot of useful design concepts if you have never worked with a distributed system. This chapter introduces Ignite shared nothing architecture, cluster topology, distributed hashing, Ignite replication strategy and durable memory architecture. It is a theoretical chapter; you may skip (not recommended) it and come back later.

**Chapter 5. Intelligent caching:** presents Ignite smart caching capabilities, Memoization, and Web-session clustering. This chapter covers developments and techniques to improve the performance of your existing web applications without changing any code.

**Chapter 6. Database:** guides you through the Ignite database features. This massive chapter explores: Ignite tables and index configurations, different Ignite queries, how SQL works under the cover, collocated/Non-collocated distributed joins, Spring data integration, using Ignite with JPA and Ignite native persistence. This chapter is for you if you are planning to use Ignite as a database.

**Chapter 7. Distributed computing:** focuses on more advanced Ignite features such as distributed computing and how Ignite can help you develop Micro-service like application, which will be performed in parallel fashion to gain high performance, low latency, and linear scalability. You will learn about Ignite inline MapReduce & ForkJoin, distributed closure execution, continuous mapping for data processing across multiple nodes in the cluster.

**Chapter 8. Streaming and complex event processing:** takes the next step and goes beyond using Apache Ignite to solve complex real-time event processing problem. This chapter covers how Ignite can be used easily with other Big data technologies such as Kafka, flume, storm, and camel to solve various business problems. We will guide you through with complete examples for developing real-time data processing on Apache Ignite.

**Chapter 9. Accelerating Big data computing:** is a full chapter about how to use Apache Spark Dataframe and RDD for processing massive datasets. We detailed by examples of how to share the application states in memory across multiple Spark jobs by using Ignite.

**Chapter 10. Management and monitoring:** explain the various tools that you can use to monitor and manage the Ignite cluster. For instance, configuring Zookeeper discovery, scaling up a cluster with Baseline topology. We provide a complete example of using Grafana for monitoring Ignite cluster at the end of this chapter.

## Code Samples

All code samples, scripts, and more in-depth examples can be found on the GitHub repository<sup>1</sup>.

---

<sup>1</sup><https://github.com/srecon/the-apache-ignite-book>

# Readership

The target audiences of this book are IT architect, team leaders or programmer with minimum programming knowledge. No excessive knowledge is required, though it would be good to be familiar with *Java*, *Spring framework* and tools like *Maven*. The book is also useful for any reader, who already familiar with Oracle Coherence, Hazelcast, Infinispan or Memcached.

# Conventions

The following typographical conventions are used in this book:

**Italic** and **Bold** indicates new terms, important words, URL's, filenames, and file extensions.

A block code is set as follows:

**Listing 1.1**

---

```
public class MySuperExtractor implements StreamSingleTupleExtractor<SinkRecord, String, S\\
tring> {

    @Override public Map.Entry<String, String> extract(SinkRecord msg) {
        String[] parts = ((String)msg.value()).split("_");
        return new AbstractMap.SimpleEntry<String, String>(parts[1], parts[2]+":"+parts[3]);
    }
}
```

---

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

```
[2018-09-30 15:39:04,479] INFO Kafka version : 2.0.0 (org.apache.kafka.common.utils.AppIn\\
foParser)
[2018-09-30 15:39:04,479] INFO Kafka commitId : 3402a8361b734732 (org.apache.kafka.common\\.utils.AppInfoParser)
[2018-09-30 15:39:04,480] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
```



## Tip

This icon signifies a tip, suggestion.



## Warning

This icon indicates a warning or caution.



## Info

This icon signifies general note.

## Reader feedback

We would like to hear your comment such as what you think, like or dislike about the content of the book. Your feedback will help us to write a better book and help others to clear all the concepts. To submit your feedback, please use the the feedback [link<sup>2</sup>](#).

---

<sup>2</sup>[https://leanpub.com/ignitebook/email\\_author/new](https://leanpub.com/ignitebook/email_author/new)

# About the authors

**Shamim Bhuiyan** is currently working as an Enterprise architect; where he's responsible for designing and building out highly scalable, and high-load middleware solutions. He received his Ph.D. in Computer Science from the University of Vladimir, Russia in 2007. He has been in the IT field for over 18 years and is specialized in Middleware solutions, Big Data and Data science. Also, he is a former SOA solution designer, speaker, and Big data evangelist. Actively participates in the development and designing of high-performance software for IT, telecommunication and the banking industry. In spare times, he usually writes the blog [frommyworkshop<sup>3</sup>](http://frommyworkshop) and shares ideas with others.

**Michael Zheludkov** is a senior programmer at AT Consulting. He graduated from the *Bau-man Moscow State Technical University* in 2002. Lecturer at BMSTU since 2013, delivering course *Parallel programming and distributed systems*.

---

<sup>3</sup><http://frommyworkshop.blogspot.ru/>

# Acknowledgments

I started writing this book more than a year ago in December 2017 in Istanbul and finished in Cuba this year. I have encouraged, motivated and inspired by a lot of amazing peoples in the journey to writing this book. At this time, my son *Mishel* grew up and sacrificed a lot of weekends and holidays so that I could write. Thanks a lot to Mishel for his patience.

I want to give special thanks to *Michael Zheludkov*; co-author of this book. He spent countless nights and weekends with me to discuss and endure my caprice during the entire process.

Many Ignite users and committers helped us to clear a lot of materials and made our life more comfortable. Thanks a lot you guys for your incredible contributions.

Special thanks go out to those who purchased and read our book at the early stage and provides useful commentaries and reviews. Thanks to Lisa & David for providing proof-reading and reviews of the book. I appreciate *Denis Magda* (GridGain Director of Product Management) and *Vladimir Grigorev* help for providing some early comments and encouragements.

Also, I would also like to thank my colleagues and friends at *AT Consulting, Sberbank*, who provided moral supports, encouragements, and technical suggestions.

# Chapter 1. Introduction



The Apache Ignite is a memory-centric distributed database, caching and computing platform for developing the data-intensive application to scale out to hundreds of millions of transactions per seconds and petabyte of in-memory data. This next generation memory-centric platform can function as an in-memory data grid, or it can be deployed as a full functional persistence data store with SQL and ACID transaction support.

The main mantra of the Apache Ignite platform is to keep the entire dataset into in-memory and on disk, boost application performance by providing caching capabilities and processing data in parallel. Such a paradigm has been identified as the answers to problems of traditional disk-based database performance since it can load and execute all the necessary datasets in memory. This paradigm eliminates the substantial number of disk I/O, which hamper transactions and create a performance bottleneck for the traditional database systems.

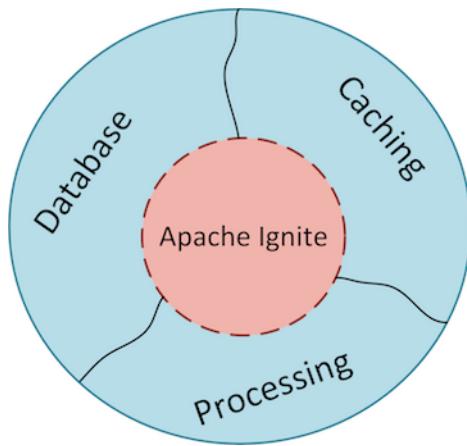


Figure 1.1

Besides the definition mentioned above, Apache Ignite can also:

1. Complete in-memory database: Apache Ignite can be deployed as a distributed

database with SQL support over non-relational data model whenever the native disk persistence is disabled.

2. Multi-model database: Apache Ignite stores data into the key-value store under the hood. Ignite stores data in special memory pages starting from the version 2.x, where the whole memory splits into pages that stores key-value entries inside. Apache Ignite provides SQL for modeling and accessing data, which allows the platform to act as a multi-model database.
3. Transactional database: Apache Ignite supports transactions at the key-value API level, which can span in different partitions on different servers. Apache Ignite will support transactions at the SQL level in the future version (partially supports in version 2.7).
4. Microservice platform: with the ability to store data and providing services for processing the data from the same computing resource makes the Apache Ignite to be a first citizen for developing fault-tolerant, scalable Microservices.
5. In-memory data fabrics: Apache Ignite also delivers functionality such as big-data accelerator, Web session clustering, Spring cache, and can be used as an implementation for cluster manager.

With these big promises, the best use-cases for Apache Ignite are as follows:

1. High-volume of ACID transactions processing.
2. Cache as a Service (CaaS).
3. Database Caching.
4. On-line fraud detection.
5. Complex event processing for IoT projects.
6. Real-time analytics.
7. HTAP business applications.
8. Fast data processing or implementing lambda architecture.

From the onset, Apache Ignite developed as an in-memory data grid. Apache Ignite evaluated from the in-memory computing platform to the memory-centric distributed database over time. The first time the product has evolved as a commercial solution named *GridGain*. Nikita Ivanov and Dmitriy Setrakyan first developed the project in 2005, and the initial release was in 2007. They incorporated the company in 2011 and raised the first investment in the United States. Realizing the potential of the open source product, they opened the most of the source code on GitHub, but control over the development of the product remained in the hands of the company. The open source part of the project transferred under the control of the Apache Software Foundation in 2015. So, the Apache Ignite was born as an in-memory

data grid and provides distributed cache (data grid), map-reduce (compute grid), streaming functionalities and so on.

The established concepts of in-memory *data* grid have ceased to reflect the essence of the project at some point (mid-2016). Then, there was the term *in-memory data fabric*, software for distributed in-memory computing, with numerous interfaces for different data sources and their consumers. In May 2017 Apache Ignite team *redesigned* the memory architecture of the Apache Ignite from scratch, where all the data and indexes stored in a completely new manageable off-heap memory that has no issues with memory fragmentation and Java GC pauses.

Apache Ignite released the version 2.1.0 and introduced the *Ignite persistence store* feature in July 2017. You no longer need to keep all the data in memory or warm up RAM after the whole cluster restart by having the persistence store enable. The persistent store keeps the superset of data and all the SQL indexes on disk making Apache Ignite fully operational from disk.

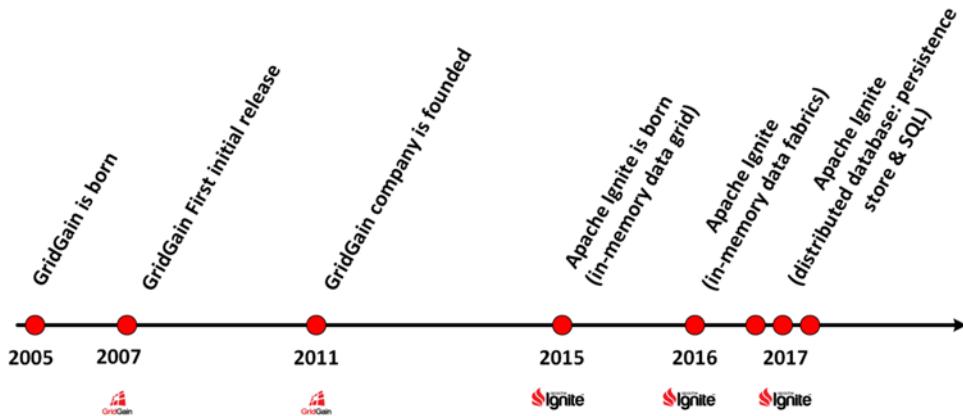


Figure 1.2

Apache Ignite 2.3.0 release brings new *SQL* capabilities into the platform in October of the same year. Apache Ignite team redesigned and optimized the performance of the *SQL*, added new features like Alter tables to DDL, and also introduced *SQLLINE* command-line tool for *SQL* based interaction. From this release, Apache Ignite team revisited the definition and purpose of the project. By their words, the definition *in-memory data fabrics* limit its capabilities, rather than the distributed database, caching, and processing platform.

So, Apache Ignite can store and process a large volume of data in memory and disk by its durable memory architecture. It can reload its state whenever Apache Ignite node restarts,

either from the hard drive or over the network from a persistence store. It's still an in-memory database despite of writing to the disk, because the disk is used merely as an append-only log for durability. Apache Ignite has the following key features:

1. Compute Grid.
2. Service Grid.
3. SQL Grid <sup>new</sup>.
4. Big Data accelerator.
5. Streaming grid.
6. Machine learning <sup>new</sup>.
7. Durable memory <sup>new</sup>.
8. 3<sup>rd</sup> party persistence store.
9. ORM support (Hibernate OGM & Spring Data) <sup>new</sup>.

The following figure illustrates the essential features of the Apache Ignite platform.

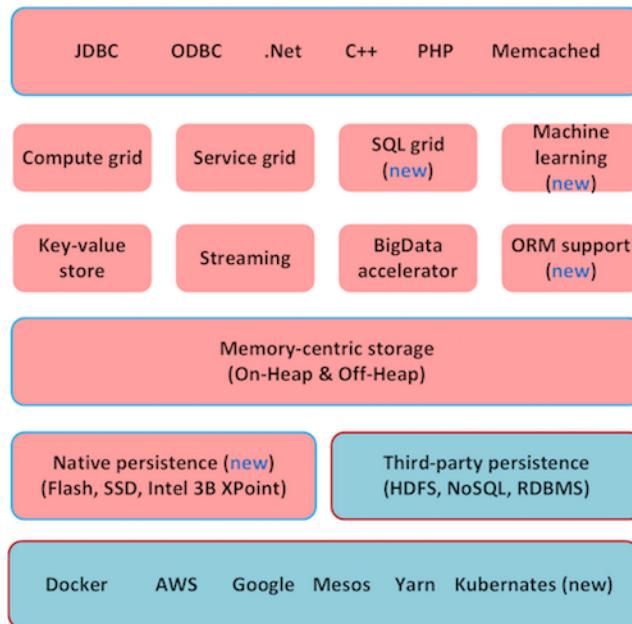


Figure 1.3

These are the core Apache Ignite technologies:

1. Written in Java and C++.

2. Integrates with Spring.
3. H2 database (SQL engine).
4. Apache Lucene (full-text search).
5. Supports JDBC, ODBC, .Net, C++, PHP etc.

The primary capabilities that Apache Ignite provides are:

- **Elasticity.** An Ignite cluster can grow horizontally by adding new nodes. Sometimes it's called scaling out or *horizontal scaling*. Apache Ignite scales to thousands of machines on commodity hardware.
- **Durable.** Apache Ignite provides robust mechanisms for preventing data loss by using disk-based durability. The disk has two significant advantages: they are permanent (if the power is turned off, the data will not be lost) and they have a lower cost per gigabyte than RAM. Apache Ignite continuously logs transactions into the append-only file (*WAL*) and periodically flush the changes to disk.
- **SQL support.** Apache Ignite specially designed for running SQL over a massive amount of data. Apache Ignite comes with *ANSI-99* compliant, horizontally scalable, and fault-tolerant distributed SQL database.
- **Decentralized.** There is no single point of failure or *SPOF*; every node in the cluster is identical and stores multiple copies of the data.
- **Fault tolerant.** Data is automatically replicated to multiple nodes by the *Backup copy* configuration. Failed nodes can be replaced with no downtime.
- **Cache as a Service (CaaS).** Apache Ignite supports *Cache-as-a-Service* across the organization which allows multiple applications from different departments to access managed in-memory cache instead of slow disk-based databases.
- **2<sup>nd</sup> Level Cache.** Apache Ignite is the perfect caching tier to use as a distributed 2<sup>nd</sup> level cache in Hibernate or MyBatis. This cache is not restricted to a single session but shared across sessions, so data is available to the entire application, not just for the current user. The 2<sup>nd</sup> level cache can significantly improve application performance because commonly used data can be held in memory in the application tier.
- **Share state in-memory across Spark applications.** Apache Ignite's memory-centric architecture enables efficient *RDD sharing* with *IgniteContext* and *IgniteRDD* to share RDDs between Spark apps. Ignite RDD allows easy sharing of states in-memory between different Spark jobs or applications. Any Spark application can put data into Ignite cache with Ignite in-memory shared RDDs, which is accessible by another Spark application later. Starting from the 2.4 version Ignite also supports Apache Spark modern *DataFrame* API. You can share data and state across Spark jobs with the support of Spark DataFrame, by writing and reading DataFrames to or from Ignite.

- **Distributed computing.** Apache Ignite provides a set of simple APIs that allows a user to distribute computation and data processing across multiple nodes in the cluster to gain high performance. Apache Ignite distributed services is very useful to develop and execute *microservice* like architecture.
- **Streaming.** Apache Ignite allows processing of continuous *never-ending* streams of data in a scalable and fault-tolerant fashion in-memory, rather than analyzing the data after it has stored in the database.
- **3<sup>rd</sup> party persistence store.** Apache Ignite can persist cache entries in RDBMS, even in NoSQL such as *MongoDB* or *Cassandra*.
- **Plugin support.** Apache Ignite built-in plugin system allowed third parties to extend the core functionality of the Apache Ignite. A user can implement features like *authentication* and *authorization* in Ignite by using the plugin system.

## Who uses Apache Ignite?

Apache Ignite is widely used around the world, and the list of consumers is growing all the time. Companies like Barclays, Misys, Sberbank (the 3<sup>rd</sup> largest bank in Europe), ING, JacTravel all use Ignite to power pieces of their architecture critical to the day-to-day operations. The vendor like TIBCO uses core caching data grid module of Apache Ignite with advanced indexing and SQL capability for their Master Data Management platform.

## Our hope

Just like any other distributed system, there is a lot to learn when diving into the Apache Ignite. The learning path can be complicated, but by the end of this book, we hope to have simplified it enough for you to not only build applications based on Apache Ignite but also apply all the use cases of in-memory databases for getting significant performance improvements from your application.

This book is a project-based guide, where each chapter focuses on the complete implementation of a real-world scenario. We will discuss the common challenges in each situation, along with tips, tricks and best practices on how to overcome them. We will introduce a complete application for every topic, which will help you hit the ground running.

# Chapter 2. Getting started with Apache Ignite

The best way to learn something new is to jump right in and start doing a simple example to play around it. Once you have experimented with the simple application, you get the necessary knowledge and the most common things that needed to use the particular technology. Whenever you had a decent overview of what you could do with this technology, you could always find more details later.

As a starting point, we provide basic installation of the Apache Ignite in this chapter. Later in this chapter, we cover a few different ways to query the Apache Ignite database so you can use your favorite editors and IDEs. Finally, we dive into the Apache Ignite platform with some examples.

In a nutshell, the following topics are covered in this chapter:

- Installing and setting up Apache Ignite.
- Running multiple instances of Apache Ignite in a single host.
- Running Apache Ignite from Docker.
- Using Apache Ignite SQLLINE command line-tool for querying tables/caches.
- Meet with Apache Ignite SQL engine: H2 database.
- Apache Ignite thin client.
- Using a universal SQL client IDE to work with Apache Ignite.
- Developing your first Java application with Apache Ignite.
- Using REST application to manipulate with the Apache Ignite caches.
- Configure a multi-node cluster in different hosts.
- A simple checklist for beginners.

## Before you start

Apache Ignite is a Java-Based application that runs on Java Virtual Machine (JVM). Because of this, a few considerations need to be taken into account when installing it. First, you need Java installed on your system for up and running Apache Ignite.

Nº	Name	Value
1	JDK	Oracle JDK 8 and above. It also supports Open JDK 8 and later, GraalVM, and IBM J8 and above.
2	OS	Linux, MacOS (10.7.3 and above), Windows XP and above, Windows Server (2008 and above), Unix (ex. Oracle Solaris)
3	Network	1G recommended
4	RAM	Default value 1 GB
5	ISA	x86, x64, SPARC, PowerPC

From a developer perspective, there are only a few things you need to get going:

- **JDK.** *Java 8* is probably the most widely used JDK/JRE right now, and is the preferred one to start with.
- **IDE.** IDEs like Eclipse, IntelliJ Idea, and NetBeans all work well with Ignite/Maven project. The sample code in the GitHub repository was tested in IntelliJ Idea, as well as from the command line.
- **Build tools.** Maven is one of the most popular build tools used with Ignite.

However, we recommend a workstation with the following configurations:

Nº	Name	Value
1	JDK	Oracle JDK 8 and above.
2	OS	Linux, MacOS (10.8.3 and above), Windows Vista SP2 and above
3	Network	No restriction (1G recommended)
4	RAM	Minimum 4GB of RAM
5	CPU	Minimum 2 core
5	IDE	Eclipse, IntelliJ Idea, NetBeans or JDeveloper
6	Apache Maven	Version 3.3.1 or above

As we have already mentioned, Apache Ignite uses JVM, therefore, make sure you have Java installed on your workstation. If you don't have Java installed on your system, please download the Java distribution from the [URL<sup>4</sup>](http://www.oracle.com/technetwork/java/javase/downloads/index.html), and follow the installation instruction to setup your environment.

---

<sup>4</sup><http://www.oracle.com/technetwork/java/javase/downloads/index.html>



## Info

The details of installing the JDK are beyond the scope of this book. The JDK can be downloaded from [here<sup>5</sup>](http://www.oracle.com/technetwork/java/javase/downloads/index.html). Therefore to download, go to the download page, follow the instructions for the type of your operating system that you have. After you have installed the JDK, you will be able to compile and run the Java programs. The JDK supplies two primary programs. The first is *javac*, which is used for compiling the Java programming code. The second is *java*, which is the standard java interpreter used for up and running Java application.

# Installing and setting up Apache Ignite

There are several ways to try Ignite, depending on how much you want to invest. In this section, we will show you the installation process of the Apache Ignite from the binary distribution and the source code.



## Warning

I am going to use the MacOS operating system for installing and running Apache Ignite through the book. Most of the commands I am using also runs on the Linux operating system and Windows.

Before starting, ensure `JAVA_HOME` environment variable is set and points to your JDK installation. Run the following command in your preferred terminal (command line console).

```
java -version
```

If Java already is installed in your system, you should get the following message on your terminal.

```
java version "1.8.0_45"  
Java(TM) SE Runtime Environment (build 1.8.0_45-b14)  
Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, mixed mode)
```

Having installed Java on our system, let's install, and run an Apache Ignite node as follows:

**Step 1.** Download the distribution archive from the Ignite [website<sup>6</sup>](https://ignite.apache.org/download.cgi#binaries).

**Step 2.** Extract distribution archive in any directory, for instance, `/opt/apache-ignite`

<sup>5</sup><http://www.oracle.com/technetwork/java/javase/downloads/index.html>

<sup>6</sup><https://ignite.apache.org/download.cgi#binaries>

```
unzip apache-ignite-fabric-{VERSION}-bin.zip -d /opt/apache-ignite
```

**Step 3.** Set an environment variable `IGNITE_HOME` to the created directory `apache-ignite` and add the bin directory of the *apache-ignite* to the PATH environment variable:

```
export IGNITE_HOME=YOUR_IGNITE_HOME_PATH  
export PATH=$PATH:$JAVA_HOME/bin:$IGNITE_HOME/bin:$HOME/bin
```

**Step 4. (Optional) Copy the folder `IGNITE_HOME/libs/optional/ignite-rest-http` to `IGNITE_HOME/libs`, this will enable the Apache Ignite REST interface.**

**Step 5.** Run the `ignite.sh` bash file in your favorite terminal or in a command prompt, which launches an Ignite node with default configuration. If you want to run an Ignite node with pre-configured caches, run the following command:

```
ignite.sh $IGNITE_HOME/examples/config/example-cache.xml
```

After executing the above command, it should print out the installed version of the Apache Ignite as shown below:

**Figure 2.1**



## Info

During the writing of this book, Apache Ignite version 2.6.0 is the latest stable release.

For now, an Apache Ignite node is up and running. Let's make a quick sanity test. Open the following URL in your favorite web browser.

```
http://localhost:8080/ignite?cmd=version
```

It should respond with a JSON representation of the Ignite version.

```
{"successStatus":0,"error":null,"sessionToken":null,"response":"2.6.0"}
```

The above response confirms that the Apache Ignite node is up and running, and the current version of the Apache Ignite is 2.6.0.

## Building from source code

You need to have [Apache Maven](#)<sup>7</sup> to be able to compile and build Ignite from the source code. If you already have a Maven installed on your system, please skip the first 4 steps.

**Step 1.** Download the Apache Maven binary distribution from this [link](#)<sup>8</sup>.

**Step 2.** Unzip the distribution archive in any directory, for example, /opt/maven.

```
tar xvf apache-maven-{version}-bin.tar.gz
```

**Step 3.** Add the *bin* directory of the created directory *apache-maven-{version}* to the PATH environment variable.

```
export PATH=$PATH:$JAVA_HOME/bin:$IGNITE_HOME/bin:$MAVEN_HOME/bin
```

**Step 4.** Run mvn -v in a new shell, the result should look similar to

---

<sup>7</sup><https://maven.apache.org/>

<sup>8</sup><https://maven.apache.org/download.cgi>

```
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4);      2014-08-12T00:58:10+03:00)
Java version: 1.8.0_45, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.11.5", arch: "x86_64", family: "mac"
```

**Step 5.** Download the Apache Ignite source release from the Ignite [website](#)<sup>9</sup>.

**Step 6.** Unzip the source package in any directory

```
unzip apache-ignite-{version}-src.zip
```

**Step 7.** Change the directory to the already extracted file with the command below:

```
cd apache-ignite-{version}-src
```

**Step 8.** Build the source code

```
mvn clean install -DskipTests
```

**Step 9.** (Optional) Build the source code with LGPL dependencies, such as Hibernate.

```
mvn clean install -DskipTests -Prelease, lGPL
```

**Step 10.** (Optional) Build in-memory Hadoop accelerator release with the command below:

```
mvn clean install -DskipTests -Dignite.edition=hadoop [-Dhadoop.version=X.X.X]
```

With the option `-DskipTests`, we skip the unit tests during compiling the source code, which can dramatically reduce the compilation time of the project. When building the source code for the first time, it will take a few minutes to download all the dependencies from the maven repositories and build the release.



## Tip

`skipTests` is a feature of the surefire plugin, compile the tests but skip running it.

On the other hand `maven.test.skip` is a feature of Maven itself, skip compile tests, run test, and ignore any test processes.

Therefore, after a successful compilation, type the following command in the same terminal to start an Ignite node.

---

<sup>9</sup><https://ignite.apache.org/download.cgi#sources>

`bin/ignite.sh`

The above command will print out various actions and start an Ignite node with the default configuration.

## Running multiple instances of the Apache Ignite in a single host

Apache Ignite allows cluster member to discover each other with multicast auto-discovery mechanism. It also helps to run multiple Ignite instances on a single host machine. Run the following commands in the different terminal to launch a few Ignite instances on a single host.

`ignite.sh`

```
shamim:~ shamim$ ignite.sh $IGNITE_HOME/examples/config/example
-cache.xml
[21:03:07]   / \ / \ / \ / \ / \ / \
[21:03:07]   \ / \ / (7 7) / \ / \ / \
[21:03:07]   / \ / \ / \ / \ / \ / \
[21:03:07]
[21:03:07] ver. 2.6.0#20180710-sha1:669feacc
[21:03:07] 2018 Copyright(C) Apache Software Foundation
[21:03:07]
[21:03:07] Ignite documentation: http://ignite.apache.org
[21:03:07]
[21:03:07] Quiet mode.
[21:03:07] ^-- Logging to file '/Users/shamim/Development/big
data/ignite/2.6.0-s3/work/log/ignite-1305b397.0.log'
[21:03:07] ^-- Logging by 'JavaLogger [quiet=true, config=nul
l]'
[21:03:07] ^-- To see **FULL** console log here add -DIGNITE_
QUIET=false or "-v" to ignite.{sh|bat}
[21:03:07]
[21:03:07] OS: Mac OS X 10.11.6 x86_64
[21:03:07] VM information: Java(TM) SE Runtime Environment 1.8.
[21:03:07] _45-b14 Oracle Corporation Java HotSpot(TM) 64-Bit Server VM 2
shamim:~ shamim$ ignite.sh $IGNITE_HOME/examples/config/example
-cache.xml
[21:06:47]   / \ / \ / \ / \ / \ / \
[21:06:47]   \ / \ / (7 7) / \ / \ / \
[21:06:47]   / \ / \ / \ / \ / \ / \
[21:06:47]
[21:06:47] ver. 2.6.0#20180710-sha1:669feacc
[21:06:47] 2018 Copyright(C) Apache Software Foundation
[21:06:47]
[21:06:47] Ignite documentation: http://ignite.apache.org
[21:06:47]
[21:06:47] Quiet mode.
[21:06:47] ^-- Logging to file '/Users/shamim/Development/big
data/ignite/2.6.0@work/log/ignite-0d8e321a.0.log'
[21:06:47] ^-- Logging by 'JavaLogger [quiet=true, config=nul
l]'
[21:06:47] ^-- To see **FULL** console log here add -DIGNITE_
QUIET=false or "-v" to ignite.{sh|bat}
[21:06:47]
[21:06:47] OS: Mac OS X 10.11.6 x86_64
[21:06:47] VM information: Java(TM) SE Runtime Environment 1.8.
[21:06:47] _45-b14 Oracle Corporation Java HotSpot(TM) 64-Bit Server VM 2
```

Figure 2.2

Apache Ignite node starts with a default configuration specified on the default-config.xml file by default. However, you can pass a configuration file explicitly from the command line. Just type `ignite.sh <path to configuration file>`, for instance.

`$ ignite.sh $IGNITE_HOME/examples/config/example-cache.xml`



## Warning

Make sure you have sufficient memory (RAM) on your host machine before running multiple Apache Ignite nodes on a single host. Normally, Apache Ignite consumes 1 GB memory per Ignite instance.

However, you can change the default JVM configuration in the `ignite.sh` file located in the `$IGNITE_HOME/bin` directory. Find the following fragment of the bash script in the `ignite.sh` file.

```
if [-z "$JVM_OPTS"]; then
    if [[ `"$JAVA" -version 2>&1 | egrep "1\.[7]. "` ]]; then
        JVM_OPTS="-Xms1g -Xmx1g -server -XX:+AggressiveOpts -XX:MaxPermSize=256m"
    else
        JVM_OPTS="-Xms1g -Xmx1g -server -XX:+AggressiveOpts -XX:MaxMetaspaceSize=256m"
    fi
fi
```

Replace the `-Xms1g -Xmx1g` parameters with `-Xms512m -Xmx512m`. Next time when you will restart the Ignite instance, your JVM will be started with Xms amount of memory and will be able to use a maximum of Xmx amount of memory. For instance, starting a JVM with `-Xms512m -Xmx512m` parameter will start the JVM with 512MB of memory, and will allow the process to use up to 512MB of memory.

## Running Apache Ignite in Docker containers

Docker<sup>10</sup> is a container technology. It has become immensely popular with both developer and system administrators. Docker simplifies creation, deployment, shipping and running of applications. It enables you to configure your application once and run it anywhere.

Apache Ignite provides Docker images for developers to learn Apache Ignite, to try new ideas, or test to and developing a prototype application based on Apache Ignite. It means that you do not need to set up your operating systems such as installing JDK or Maven; it automatically starts up a fully configured Apache Ignite node.

Once you have installed Docker and Docker machine running on your system, you can pull a Docker Image from the Docker hub and start the image. Apache Ignite Docker images are available at <https://hub.docker.com/r/apacheignite/ignite/tags/>. The latest version will be used by default.



### Info

We are not covering the installation procedure of the Docker. The [Docker community](#)<sup>11</sup> provides a comprehensive guide to install and run Docker for every operating system.

<sup>10</sup><https://www.docker.com>

<sup>11</sup><https://docs.docker.com>

**Step 1.** Pull the Docker image. Use the following command to pull the Apache Ignite Docker image.

```
docker pull apacheignite/ignite
```

It will download the latest Ignite image from the Docker hub by using default tag: **latest**.

```
shamim:~ shamim$ docker pull apacheignite/ignite
Using default tag: latest
latest: Pulling from apacheignite/ignite
4fe2ade4980c: Pull complete
6fc58a8d4ae4: Pull complete
819f4a45746c: Pull complete
a5770aa7884a: Pull complete
703fdbb3d9f4: Pull complete
faada2abf232: Pull complete
aa25326111cd: Pull complete
fd5aeda61ba6: Pull complete
f50830480165: Pull complete
87b89e6c5ff8: Pull complete
Digest: sha256:d7deab68b8fa447798eedaa71c6c4794fdd704e8400458d612af43d4d2264bc7
Status: Downloaded newer image for apacheignite/ignite:latest
```

Figure 2.3

**Step 2.** You can also pull Docker images by image tag version. For instance, the following command will pull the Apache Ignite image with version **2.2.0**.

```
docker pull apacheignite/ignite:2.2.0
```

**Step 3.** You can use `docker` command to get the full list of the downloaded images. In my case, I have two images downloaded and ready for the start.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
apacheignite/ignite	latest	4eca783f1bed	5 weeks ago	530.2 MB
<none>	<none>	107c5a038ae7	14 months ago	1.1 GB
cassandra-matview	latest	38f4e5bcf7f1	2 years ago	380.8 MB
cassandra	3.3	7514d930eadf	2 years ago	379.8 MB

Figure 2.4

**Step 4.** To run Ignite Docker container, use the following command:

```
docker run -it --net=host -e "CONFIG_URI=https://raw.githubusercontent.com/
apache/ignite/master/examples/config/example-cache.xml" apacheignite/ignite
```

It will start a single node of the Apache Ignite. You should see the following logs in the terminal.

```
[18:39:47] 2018 Copyright(C) Apache Software Foundation
[18:39:47]
[18:39:47] Ignite documentation: http://ignite.apache.org
[18:39:47]
[18:39:47] Quiet mode.
[18:39:47]   ^-- Logging to file '/opt/ignite/apache-ignite/work/log/ignite-81d98712.0.log'
[18:39:47]   ^-- Logging by 'JavaLogger [quiet=true, config=null]'
[18:39:47]   ^-- To see **FULL** console log here add -DIGNITE_QUIET=false or "-v" to ignite.{sh|bat}
[18:39:47]
[18:39:47] OS: Linux 4.1.19-boot2docker amd64
[18:39:48] VM information: OpenJDK Runtime Environment 1.8.0_181-b13 Oracle Corporation OpenJDK 64-Bit Server VM
[18:39:48] Please set system property '-Djava.net.preferIPv4Stack=true' to avoid possible problems in mixed
[18:39:48] Configured plugins:
[18:39:48]   ^-- None
[18:39:48]
[18:39:48] Configured failure handler: [hnd=StopNodeOrHaltFailureHandler [tryStop=false, timeout=0, super=Ab
reHandler [ignoredFailureTypes=[SYSTEM_WORKER_BLOCKED]]]]
[18:39:48] Message queue limit is set to 0 which may lead to potential OOMEs when running cache operations i
C or PRIMARY_SYNC modes due to message queues growth on sender and receiver sides.
[18:39:48] Security status [authentication=off, tls/ssl=off]
[18:39:50] Nodes started on local machine require more than 80% of physical RAM what can lead to significant
```

Figure 2.5

In addition to `docker run` command, the following configuration parameters can be passed as an environmental variable in the Docker container:

Name	Description	Default	Example
CONFIG_URI	URL to the Ignite configuration file (can also be relative to the META-INF folder on the class path). The downloaded config file will be saved to ./ignite-config.xml	N/A	
OPTION_LIBS	Ignite optional libs which will be included in the class path.	ignite-log4j, ignite-spring, ignite-indexing	
JVM_OPTS	Environment variables passed to Ignite instance in your docker command.	N/A	-Xms1g -Xmx1g -server -XX:+AggressiveOpts -XX:MaxPermSize=256m
EXTERNAL_LIBS	List of URL's to libs.	N/A	



## Tip

Apache Ignite Docker container reveals only port 11211, 47100, 47500, and 49112 for connecting from the host machine or another Docker container. Try to open the necessary ports on your host operating system if you encounter any problem to connect with the Ignite node from your host machine.

## Using Apache Ignite SQLLINE CLI

So far, we have installed and run Apache Ignite node. In this section, the first time we will create tables (under the hood, it will be caches), and run a few queries for retrieving data from them. Since 2.x version, Apache Ignite provides an SQL command-line (CLI) tool for creating tables and running queries over tables. Apache Ignite SQLLINE tool is a default CLI tool for SQL connectivity in Ignite.

To make it easy, we will create two tables DEPT (DEPARTMENT) and EMP (EMPLOYEE), and a few indexes on these tables. From the relational data model view, the relationship between the EMP and DEPT tables are *one-to-many*. It means that one department can relate to one or many employees (or even no employees) and that an employee is associated with only one department (or, sometimes, no department). I can restate this business rule: Each employee in a department may be in one and only one department, and that department must exist in the department table.

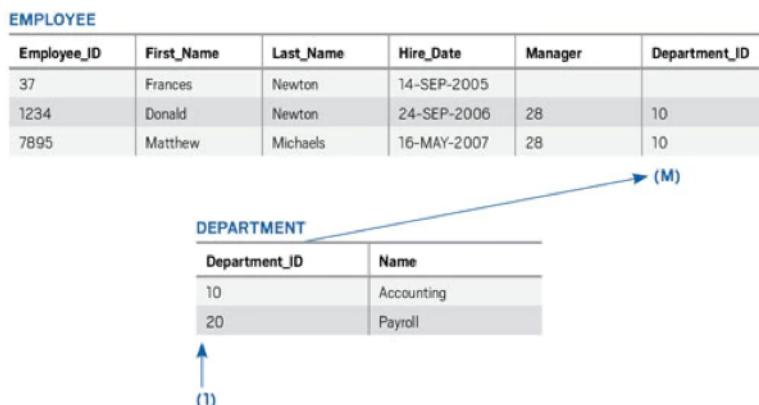


Figure 2.6



## Warning

Apache Ignite as a *Key-Value* database, there is no physical foreign key relationship between caches (tables).

First, make sure that an Apache Ignite node is up and running. If not, open the command shell, and run the following command from the IGNITE\_HOME folder.

bin/ignite.sh

**Step 1.** To connect SQLLINE to the Ignite cluster, run `sqlline.sh -u jdbc:ignite:thin:{host}` from your IGNITE\_HOME/bin directory. For instance:

```
./sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
```

```
shamim:~ shamim$ sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
issuing: !connect jdbc:ignite:thin://127.0.0.1/ '' '' org.apache.ignite.IgniteJdbcThinDriver
Connecting to jdbc:ignite:thin://127.0.0.1/
Connected to: Apache Ignite (version 2.6.0#20180710-sha1:669feacc)
Driver: Apache Ignite Thin JDBC Driver (version 2.6.0#20180710-sha1:669feacc)
Autocommit status: true
Transaction isolation: TRANSACTION_REPEATABLE_READ
sqlline version 1.3.0
```

Figure 2.7

Looking at the console log, we see that SQLLINE command tool applies JDBC thin client to connect to the Apache Ignite cluster.

**Step 2.** We can execute SQL statements to create or insert data into the tables after connecting to the Ignite cluster. Let's create two tables, which we have described earlier.

Listing 2.1

---

```
CREATE TABLE dept
(
    deptno LONG,
    dname VARCHAR,
    loc  VARCHAR,
    CONSTRAINT pk_dept PRIMARY KEY (deptno)
);
```

```
CREATE TABLE emp
```

```
(
    empno  LONG,
```

```

ename  VARCHAR,
job   VARCHAR,
mgr   INTEGER,
hiredate DATE,
sal    LONG,
comm   LONG,
deptno LONG,
CONSTRAINT pk_emp PRIMARY KEY (empno)
);

```

---



## Tip

You can find the DDL and DML scripts for the project chapter-2 on the [Github repository](#)<sup>12</sup>.

Execute the following SQLLINE command as follows:

```
0: jdbc:ignite:thin://127.0.0.1/> !tables
```

On the console logs, you should see the following information:

0: jdbc:ignite:thin://127.0.0.1/>	!tables		
TABLE_CAT	TABLE_SCHEM	TABLE_NAME	TABLE_TYPE
	PUBLIC	EMP	TABLE
	PUBLIC	DEPT	TABLE

Figure 2.8

You can also run a SQL script from the specified file by using the SQLLINE command **!run**. For instance,

```
0: jdbc:ignite:thin://127.0.0.1/> !run /users/opt/tiger.sql
```



## Tip

The foreign key constraint is not supported in Apache Ignite. If you try to add a foreign key constraint such as *constraint fk\_deptno foreign key (deptno) references dept (deptno)*, the command ends with the following error: Error: Too many constraints - only PRIMARY KEY is supported for CREATE TABLE (state=0A000,code=0)

<sup>12</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-2>

**Step 3.** Let's create a unique index as shown below:

**Listing 2.2**

---

```
CREATE INDEX ename_idx ON emp (ename);
```

---

The above statement creates a unique index on column *ename* on *emp* table.

**Step 4.** Insert some sample data into two tables as follows:

**Listing 2.3**

---

```
INSERT INTO dept (deptno, dname, loc)
```

```
VALUES (10,
```

```
    'ACCOUNTING',
```

```
    'NEW YORK');
```

---

```
INSERT INTO dept (deptno, dname, loc)
```

```
VALUES(20,
```

```
    'RESEARCH',
```

```
    'DALLAS');
```

---

```
INSERT INTO dept (deptno, dname, loc)
```

```
VALUES(30,
```

```
    'SALES',
```

```
    'CHICAGO');
```

---

```
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
```

```
VALUES(7839,
```

```
    'KING',
```

```
    'PRESIDENT',
```

```
    NULL,
```

```
    to_date('17-11-1981', 'dd-mm-yyyy'),
```

```
    5000,
```

```
    NULL,
```

```
    10);
```

---

```
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
```

```
VALUES(7698,
```

```
    'BLAKE',
```

```
    'MANAGER',
```

```
    7839,
```

```
    to_date('1-5-1981', 'dd-mm-yyyy'),
```

```
    2850,
```

```
NULL,
30);

INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
VALUES(7782,
  'CLARK',
  'MANAGER',
  7839,
  to_date('9-6-1981', 'dd-mm-yyyy'),
  2450,
  NULL,
  10);
```

---

**Step 5.** Now, we know how to execute the DDL/DML statement, let's take it one step further and execute some SELECT statements to query the tables.

**Listing 2.4**

---

```
select * from emp; //select all columns and rows from the emp table
```

---

This produces the following output:

```
0: jdbc:ignite:thin://127.0.0.1/> select * from emp;
+-----+-----+-----+-----+-----+
| EMPNO | ENAME | JOB   | M |
+-----+-----+-----+-----+
| 7782  | CLARK | MANAGER | 7839 |
| 7839  | KING  | PRESIDENT | null |
+-----+-----+-----+-----+
2 rows selected (0.043 seconds)
```

**Figure 2.9**

Finally, run the following query to find out the employees working in each department.

**Listing 2.5**

---

```
SELECT e.ename, e.empno, e.deptno, d.loc
FROM emp e, dept d
WHERE e.deptno = d.deptno
order by e.ename;
```

---

```
+-----+-----+-----+-----+-----+
| ENAME | EMPNO | DEPTNO | L  |
+-----+-----+-----+-----+
| CLARK | 7782  | 10    | NEW YORK |
| KING  | 7839  | 10    | NEW YORK |
+-----+-----+-----+-----+
2 rows selected (0.014 seconds)
```

**Figure 2.10**

As a last SQL select example, let's get all the managers from EMP and DEPT table per departments number 10, 20 and 30.

## Listing 2.6

```
SELECT d.deptno, d.dname, e.empno, e.ename  
FROM emp e  
    INNER JOIN dept d  
        ON (e.deptno = d.deptno)  
WHERE EXISTS (select 1 from emp t where t.mgr = e.empno)  
    AND d.deptno in (10,20,30);
```

Which will print the quintessential result:

```

0: jdbc:ignite:thin://127.0.0.1/> SELECT d.deptno, d.dname, e.empno, e.ename FROM emp e
..... > INNER JOIN dept d
..... > ON (e.deptno = d.deptno)
..... > WHERE EXISTS (select 1 from emp t where t.mgr = e.empno)
..... > AND d.deptno in (10,20,30);

+-----+-----+-----+-----+
| DEPTNO | DNAME | EMPNO | EN
+-----+-----+-----+-----+
| 10    | ACCOUNTING | 7839 | KING
+-----+-----+-----+-----+
1 row selected (0.008 seconds)

```

**Figure 2.11**

The same way you can use aggregated functions such as `count(*)`, `SUM()`, `AVG()` or system functions (`decode`, `NVL/2`) in your query. We will detail the Apache Ignite SQL functionalities very soon. If you are curious about how Apache Ignite runs SQL queries over caches, then wait for the next section.

## Meet with Apache Ignite SQL engine: H2 database

Under the cover, Apache Ignite uses H2<sup>13</sup> database for executing SQL queries over Ignite caches, and its usually conceal from the application developers. H2 database is a high-speed in-memory SQL database written in pure Java. H2 database can run in the embedded or server mode. Every Apache Ignite node runs one instance of the H2 database in the embedded mode. In this mode, H2 database instance runs with the same Apache Ignite process. The H2 database starts along with Ignite node and stops whenever the Ignite node dies or forced to stop.

<sup>13</sup><http://www.h2database.com/html/main.html>



## Info

An embedded H2 instance is always launched as a part of an Apache Ignite node process whenever an *ignite-indexing* module is added to the node's classpath. If the node is started from a command-line tool using the `ignite.sh{bat}` script, then the module will be added to the classpath automatically since it's located under the `{apache_ignite}\libs\` directory.

Also, H2 database supplies an H2 web console, which is a standalone application and includes its web server. This console allows you to access the H2 SQL database using a browser interface. This web console lets you know the internal structure of the tables and indexes. Moreover, with the H2 web console, you can analyze how a query is executed by the database, e.g., whether indexes are used or if the database has done an expensive full scan. This feature is crucial for optimizing the query performance.

To start the H2 web console, you have to set an environmental variable through the command console, and run the `ignite.sh{bat}` script. Set the variable as follows:

```
export IGNITE_H2_DEBUG_CONSOLE=true
```

Start an Ignite node from the same terminal as shown below:

```
IGNITE_HOME/bin/ignite.sh
```

The H2 web console URL should show on your terminal log.

```
[shamim:2.4 shamim$ ignite.sh
[10:13:18]   _/\_ / \_ / \_ / \_ / \_
[10:13:18]  / \_ / \_ / \_ / \_ / \_ / \_
[10:13:18]  / \_ \_ / \_ / \_ / \_ / \_ / \_
[10:13:18]  / \_ \_ / \_ / \_ / \_ / \_ / \_
[10:13:18] ver. 2.4.0#20180305-sha1:aa342270
[10:13:18] 2018 Copyright(C) Apache Software Foundation
[10:13:18]
[10:13:18] Ignite documentation: http://ignite.apache.org
[10:13:18]
[10:13:18] Quiet mode.
[10:13:18]  ^-- Logging to file '/Users/shamim/Development/bigdata/ignite/2.4/work/log/ignite-b3e856c8.0.log'
[10:13:18]  ^-- Logging by 'Javalogger [quiet=true, config=null]'
[10:13:18]  ^-- To see **FULL** console log here add -DIGNITE_QUIET=false or "-v" to ignite.{sh|bat}
[10:13:18]
[10:13:18] OS: Mac OS X 10.11.6 x86_64
[10:13:18] VM information: Java(TM) SE Runtime Environment 1.8.0_45-b14 Oracle Corporation Java HotSpot(TM) 64-Bit Server VM 25.45-b02
[10:13:18] Configured plugins:
[10:13:18]  ^-- None
[10:13:18]
[10:13:18] Message queue limit is set to 0 which may lead to potential OOMEs when running cache operations in FULL_ASYNC or PRIMARY_S
[10:13:18] s growth on sender and receiver sides.
[10:13:18] Security status [authentication=off, tls/ssl=off]
[10:13:19] H2 debug console URL: http://192.168.1.35:56723/frame.jsp?jsessionid=282bc4a8e45462eade038aa409997484
[10:13:20] Performance suggestions for grid (fix if possible)
[10:13:20] To disable, set -DIGNITE_PERFORMANCE_SUGGESTIONS_DISABLE=true
[10:13:20]  ^-- Enable G1 Garbage Collector (add '-XX:+UseG1GC' to JVM options)
[10:13:20]  ^-- Set max direct memory size if getting 'OOME: Direct buffer memory' (add '-XX:MaxDirectMemorySize=<size>[g|m|M|k|K]' )
[10:13:20]  ^-- Disable processing of calls to System.gc() (add '-XX:+DisableExplicitGC' to JVM options)
[10:13:20] Refer to this page for more performance suggestions: https://apacheignite.readme.io/docs/jvm-and-system-tuning
[10:13:20]
[10:13:20] To start Console Management & Monitoring run ignitevisorcmd.{sh|bat}
[10:13:20]
[10:13:20] Ignite node started OK (id=b3e856c8)
[10:13:20] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=1.0GB]
[10:13:20] Data Regions Configured:
[10:13:20]  ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
```

Figure 2.12

Copy the URL from the terminal and open it on any of your favorite browsers. A welcome page should pop up as shown in the screenshot below.

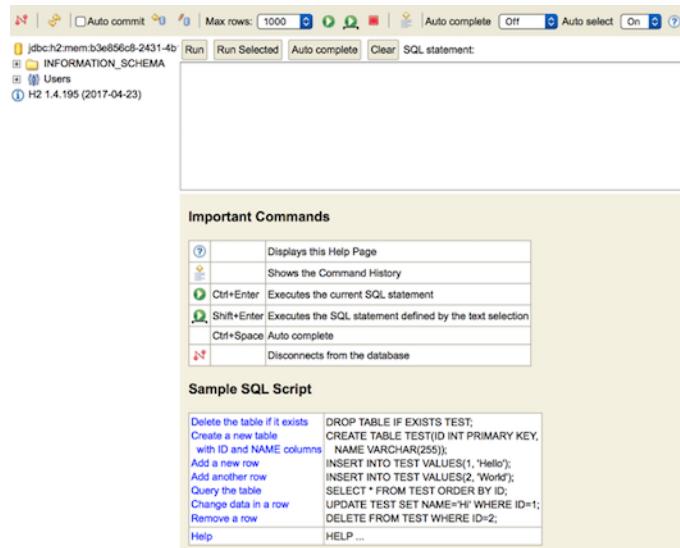


Figure 2.13



## Tip

The web page should be open automatically in your default web browser if the H2 web console started successfully. If you want to connect to the H2 web console from another computer, you need to provide the IP address with the port of the server, for instance, <http://192.168.1.35:56723>

As stated earlier, in H2 web console application, we can administrate the H2 database, running SQL queries and much more. Whenever we create any table in Ignite (through SQLLINE or Ignite JAVA API), a Meta-data information of the tables created and displayed in the H2 database. However, the data and the indexes are always stored in the Ignite caches that execute queries through the H2 database engine. Let's create the department (DEPT) tables through SQLLINE again, and see what will happen on the H2 database. Run the following script for the SQLLINE command line console to connect to the Ignite cluster:

```
sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
```

Next, create the DEPT table and insert a few rows.

**Listing 2.7**

```
CREATE TABLE dept
(
    deptno LONG,
    dname VARCHAR,
    loc  VARCHAR,
    CONSTRAINT pk_dept PRIMARY KEY (deptno)
);
```

```
INSERT INTO dept (deptno, dname, loc)
VALUES (10,
        'ACCOUNTING',
        'NEW YORK');
```

```
INSERT INTO dept (deptno, dname, loc)
VALUES (20,
        'RESEARCH',
        'DALLAS');
```

```
INSERT INTO dept (deptno, dname, loc)
VALUES (30,
```

---

```
'SALES',
'CHICAGO');
```

---

Now, we have a table named DEPT with some example data in Ignite.

TABLE_CAT	TABLE_SCHEM	TABLE_NAME
		PUBLIC
		DEPT

Figure 2.14

Let's get back to the H2 web console and refresh the H2 database objects panel (you should see a small refresh button on the H2 web console menu bar on the upper left side of the web page). A new table named *DEPT* should appear on the object panel after refreshing the panel. Expand the *DEPT* table, and you notice a few tables attributes as shown in figure 2.15 below.

The screenshot shows the H2 Web Console interface. At the top, there are various buttons for connection management and SQL execution. Below the header, the URL is displayed as `jdbc:h2:mem:e9752a16-da20-4cb6-ba26-3ed5`. The main area shows a tree view of database objects. Under the 'DEPT' table, several columns are listed: '\_KEY', '\_VAL', '\_VER', 'DEPTNO', 'DNAME', 'LOC', and 'Indexes'. There is also a folder icon labeled 'INFORMATION\_SCHEMA'.

Figure 2.15

From the above screenshot, we can discover that h2 DEPT table contains 3 extra columns with name *\_KEY*, *\_VAL*, *\_VER*. Apache Ignite as a key-value data store, always stores cache keys and values as *\_KEY* and *\_VAL* fields. H2 DEPT table column *\_key* and *\_val* corresponded to the Ignite internal *\_key* and *\_val* field of the cache. *\_VER* field assigns to the Ignite topology version and the node order.

Run the following query to unravel the mystery.

#### Listing 2.8

---

```
select _key, _val, _ver from dept;
```

---

Executing the above script in H2 web console SQL statement panel results in the following.

Run Run Selected Auto complete Clear SQL statement:		
select _key, _val, _ver from dept;		
KEY	VAL	VER
10	SQL_PUBLIC_DEPT_d8575d1b_87a8_4642_9b14_363153555594 [idHash=81635754, hash=-850331171, DNAME=ACCOUNTING, LOC=NEW YORK]	GridCacheVersion [topVer=132756184, order=1521276182264, nodeOrder=1]
20	SQL_PUBLIC_DEPT_d8575d1b_87a8_4642_9b14_363153555594 [idHash=1080543470, hash=-1365635177, DNAME=RESEARCH, LOC=DALLAS]	GridCacheVersion [topVer=132756184, order=1521276182266, nodeOrder=1]
30	SQL_PUBLIC_DEPT_d8575d1b_87a8_4642_9b14_363153555594 [idHash=938817198, hash=1109622257, DNAME=SALES, LOC=CHICAGO]	GridCacheVersion [topVer=132756184, order=1521276182268, nodeOrder=1]

(3 rows, 2 ms)

Figure 2.16

From the above figure, you can notice that every `_KEY` field hold the actual department number, and the rest of the information such as department name and the location hold by the `_VAL` field. H2 Dept table `deptno` field maps to the actual `_key` field of the cache, `dname` field maps to the `_VAL` field `dname` attribute and finally, `loc` field maps to the `_VAL` field `location` attribute.

If you expand the indexes object for the DEPT table on the H2 web console object panel, you should discover three different indexes created by the Ignite: `_key_PK`, `_key_PK_proxy` and `_key_PK_hash`.

`_key_PK` is the btree primary key index.

`_key_PK_hash` is the H2 primary key hash index. This index is an in-memory hash index, and it is usually faster than the regular index.

`_key_PK_proxy` is a proxy index (not a primary index), which allows delegating the calls to the underlying normal index.

We can do more than investigate the internal structure of the table such as executing a query plan with the H2 web console. Let's run the following SQL query on the SQL statement panel as shown below:

#### Listing 2.9

---

```
explain select * from dept d where d.deptno = 10;
```

---

The preceding SQL statement should give us the following query plan as shown in the following screenshot.

```
explain
select * from dept d where d.deptno = 10;
PLAN
SELECT
    D.DEPTNO,
    D.DNAME,
    D.LOC
FROM PUBLIC.DEPT D
/* PUBLIC."_key_PK_proxy": DEPTNO = 10 */
WHERE D.DEPTNO = 10
(1 row, 3 ms)
```

Figure 2.17

From the above screenshot, we can notice that the H2 SQL engine used the \_key\_PK\_proxy index to query the DEPT table. We are going to explain the query plans in detail in chapter 6.



## Tip

H2 database uses a cost-based (running time) optimizer. For simple queries and queries with medium complexity (less than 7 tables in the join clause), the expected SQL running time cost of all possible plans is calculated, and the plan with the lowest cost (running time) is used.

It's enough for getting started with this chapter; we explain a lot about Ignite SQL engine on the following chapters. Now, let's move to a new section where we discuss a few essential topics for the beginners who want to drive themselves into the realm of the Ignite platform.

## Using a universal SQL client IDE to work with Apache Ignite

Apache Ignite out-of-the-box shipped with a JDBC driver that allows you to connect to the Ignite cluster and execute SQL queries against Ignite caches. This feature makes it possible for a developer to connect to the Ignite cluster and process data from their favorite SQL tools. This section is for those who are comfortable with the declarative language like SQL rather than execute a bunch of Java or C# code for processing data. All you need to do is to configure a JDBC driver for the tool.

The first time we introduced the use of the universal SQL client DBeaver to connect to the Ignite cluster in the book [High performance in-memory computing with Apache Ignite](#)

Ignite<sup>14</sup>. After a while, the Apache Ignite community add full concise documentation into their manual to accomplish the basic configuration for data processing with Ignite through DBeaver. So, we decided not to repeat the information here in this section. However, while writing the book, we have got a few requests for describing how to configure the IntelliJ DataGrip<sup>15</sup> SQL client to communicate to the Ignite cluster.

IntelliJ DataGrip is a favorite commercial SQL IDE for professional SQL developer. It supports a large range of database vendors, and also let you connect to any database with JDBC thin client. This specific tool provides powerful SQL editor with smart code completion, allows you to execute SQL queries in various modes which made the tool very popular among developers.



## Tip

You can also try SQuirrel<sup>16</sup> SQL tool to connect to Ignite cluster for processing data. SQuirrel SQL tool also supports the JDBC client.

In the rest of the sub-section of this chapter, we cover the configuration of the JDBC driver for the *IntelliJ DataGrip* and show you how to execute SQL queries against Ignite database.

**Step 1.** Download and install IntelliJ DataGrip for an operating system of your choice. The installation process of the DataGrip is easy and well documented on the vendor web site.

**Step 2.** Once DataGrip is downloaded and install, lunch it, and select the File-> *data sources and driver* from the menu item.

**Step 3.** Press the + button to create a new data source and select the *driver and data source* option from the drop-down item.

**Step 4.** Fill out the required settings from the following screen as follows:

- Name: Apache Ignite
- URL: jdbc:ignite:thin://127.0.0.1/

---

<sup>14</sup><https://leanpub.com/ignite>

<sup>15</sup><https://www.jetbrains.com/datagrip>

<sup>16</sup><http://squirrel-sql.sourceforge.net>

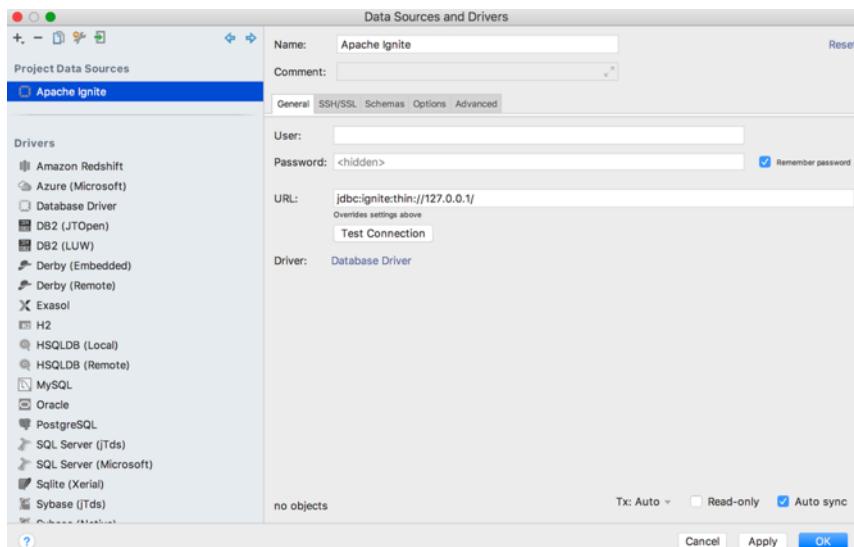


Figure 2.18

And click the *Database driver* URL to configure the driver manager.

**Step 5.** Add the following setting to the driver manager screen.

- Name: Apache Ignite database driver.
- Additional files: click on the + button and locate {apache-ignite-version}/libs/ignite-core-{version}.jar file that includes the Ignite's JDBC driver.
- Class: Select the `org.apache.ignite.IgniteDBCThinDriver` name in the drop-down menu.
- Dialect: Generic SQL

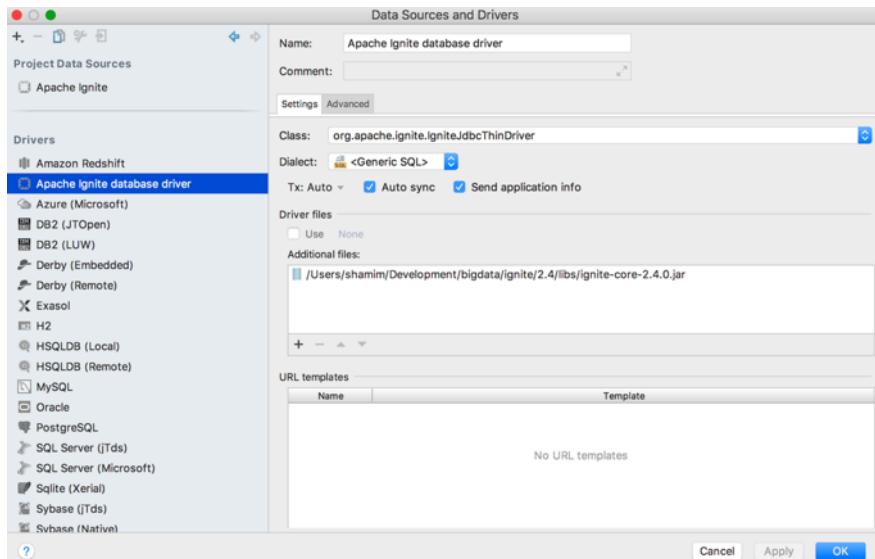


Figure 2.19

Click the *OK* button to complete the setup process. A new window will open with the SQL editor for editing and executing SQL queries.

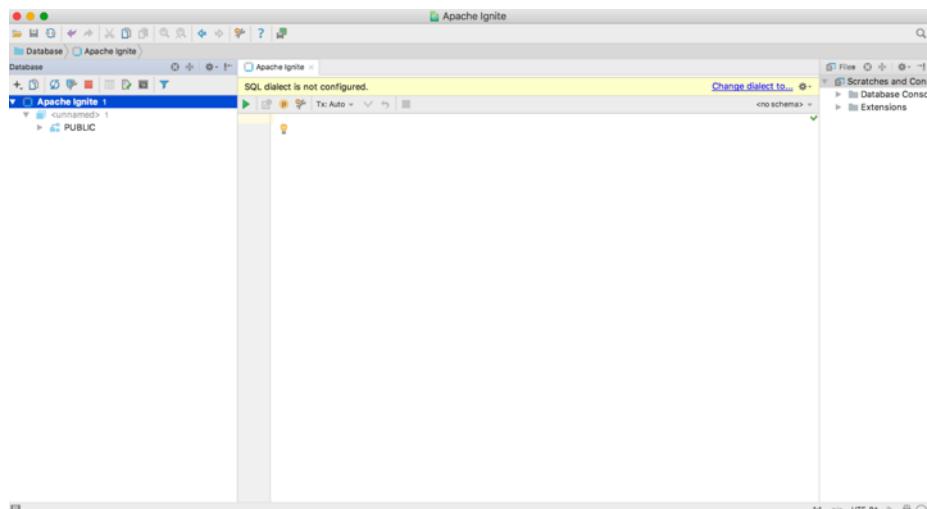


Figure 2.20

**Step 6.** Create the DEPT and EMP table using the following SQL statements.

**Listing 2.10**

---

```
DROP TABLE dept;

DROP TABLE emp;

CREATE TABLE dept
(
    deptno LONG,
    dname VARCHAR,
    loc  VARCHAR,
    CONSTRAINT pk_dept PRIMARY KEY (deptno)
);

CREATE TABLE emp
(
    empno  LONG,
    ename  VARCHAR,
    job   VARCHAR,
    mgr    INTEGER,
    hiredate DATE,
    sal    LONG,
    comm   LONG,
    deptno LONG,
    CONSTRAINT pk_emp PRIMARY KEY (empno)
);
```

---

And hit the *execute* button to run the SQL statements.



## Warning

In case of error like *DEPT/EMP table not found*, click the **ignore all** button to run the rest of the SQL scripts.

**Step 7.** Add a few records into the EMP and DEPT table through SQL INSERT statement.

**Listing 2.11**

---

```
INSERT INTO dept (deptno, dname, loc)
VALUES (10,
       'ACCOUNTING',
       'NEW YORK');

INSERT INTO dept (deptno, dname, loc)
VALUES(20,
      'RESEARCH',
      'DALLAS');

INSERT INTO dept (deptno, dname, loc)
VALUES(30,
      'SALES',
      'CHICAGO');

INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
VALUES(7839,
      'KING',
      'PRESIDENT',
      NULL,
      to_date('17-11-1981', 'dd-mm-yyyy'),
      5000,
      NULL,
      10);

INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
VALUES(7698,
      'BLAKE',
      'MANAGER',
      7839,
      to_date('1-5-1981', 'dd-mm-yyyy'),
      2850,
      NULL,
      30);

INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
VALUES(7782,
      'CLARK',
      'MANAGER',
      7839,
      to_date('9-6-1981', 'dd-mm-yyyy'),
      2450,
```

---

```
NULL,
10);
```

---

The SQL editor with the preceding DML statements should look as follows:

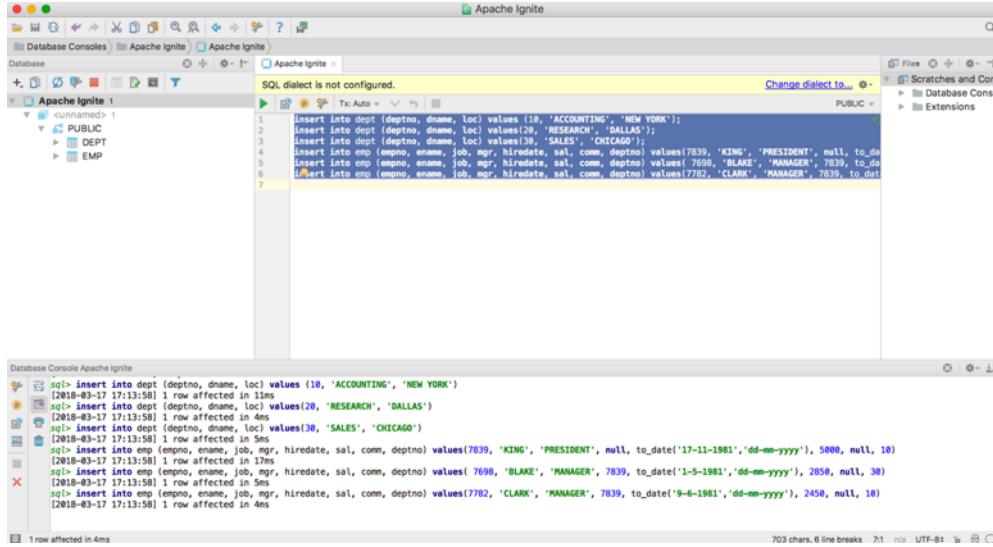


Figure 2.21

**Step 8.** Once we have inserted some data, we can perform SQL queries to retrieve the data from the Ignite database. Let's write SQL queries to get all the managers from EMP and DEPT tables by the department's numbers of 10, 20 and 30.

Listing 2.12

---

```
SELECT d.deptno,
       d.dname,
       e.empno,
       e.ename
FROM emp e
INNER JOIN dept d
    ON (e.deptno = d.deptno )
WHERE EXISTS (SELECT 1
    FROM emp t
    WHERE t.mgr = e.empno)
AND d.deptno IN ( 10, 20, 30 );
```

---

The above query should return the employee KING who works in department **Accounting**. The result is shown in the figure 2.22.

The screenshot shows the Apache Ignite DataGrip interface. The top window is titled "Apache Ignite" and displays a SQL query:

```

1 SELECT d.deptno, d.dname, e.empno, e.ename
2   FROM emp e
3      JOIN dept d
4     ON (e.deptno = d.deptno)
5 WHERE EXISTS (select 1 from emp t where t mgr = e.empno)
6   AND d.deptno IN (10,20,30);

```

The bottom window is titled "Database Console Apache Ignite" and shows the results of the query:

DEPTNO	DNAME	EMPNO	ENAME
10	ACCOUNTING	7839	KING

1 row retrieved starting from 1 in 27ms (execution: 8ms, fetching: 19ms)

Figure 2.22

Apache Ignite is completely *ANSI-99* compliant, and IntelliJ DataGrip let you run any SQL query like analytical or Ad-hoc queries. For more useful features such as running explain plan, refactoring, and on-the-fly code analysis, please see the IntelliJ DataGrip documentation.

## First Java application

In this section, we take one step further and guide you through the creation of the first Ignite application to write and read (put/get) from the distributed cache. As a first example, we make it as simple as possible to show you how to write an application in Java for manipulating the data of the Apache Ignite cluster. The applications shown in this section are available in the [GitHub repository<sup>17</sup>](#). You can clone or download the project from the GitHub, compile the application with Maven, and run it in your workstation. However, if you want to enter the programs manually, you are free to do so.

You follow these simple four steps:

1. Start an Ignite node.

<sup>17</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-2>

2. Create a Maven project or enter the program.
3. Compile the Java program.
4. Run the Java application.

**Step 1.** Start your Apache Ignite single node cluster if it is not started yet. Use the following command in your preferred terminal.

`IGNITE_HOME/bin/ignite.sh`

**Step 2.** Create a Maven project with the following command. Skip this step if you download the project from the GitHub repository.

**Listing 2.13**

---

```
mvn archetype:generate -DartifactId=chapter-two -DgroupId=com.blu.imdg -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

---

The above command will create a directory with the same name given as the `artifactId`. Change into the directory `chapter-two`. Under this directory, you will have the following standard project structure.

```
chapter-two
| -- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   |-- com
    |   |   |   |-- blu
    |   |   |   |   |-- imdg
    |   |   |   |   |   `-- App.java
    |   |-- test
    |   |   |-- java
    |   |   |   |-- com
    |   |   |   |   |-- blu
    |   |   |   |   |   |-- imdg
    |   |   |   |   |   |   `-- AppTest.java
```

The `src/main/java` directory contains the project source code, the `src/test/java` directory contains the test sources, and the `pom.xml` is the project's Project Object Model or POM. The `pom.xml` file is the core of the project's configuration in Maven. It is a single configuration file that

contains all the necessary information to compile and run the Java application. The pom.xml file could be complicated, but it's not necessary to understand all of the intricacies just yet to use it effectively.

**Step 3.** Add the following Ignite Maven dependency into the *pom.xml* file.

**Listing 2.14**

---

```
<dependency>
<groupId>org.apache.ignite</groupId>
<artifactId>ignite-core</artifactId>
<version>${ignite.version}</version>
</dependency>
```

---

Also, add the project properties section into the *pom.xml* file as shown below.

**Listing 2.15**

---

```
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<ignite.version>2.6.0</ignite.version>
</properties>
```

---

You can run the application from the command line with Maven. Alternatively, you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run the application. This makes it easy to ship, and deploy the service as an application throughout the development lifecycle, across different environments, and so on. Add two more plugins into the plugin section of the *pom.xml* to create a fat executable jar for running the application efficiently.

**Listing 2.16**

---

```
<build>
<plugins>
<plugin>
<groupId>com.jolira</groupId>
<artifactId>onejar-maven-plugin</artifactId>
<version>1.4.4</version>
<executions>
<execution>
<id>build-query</id>
<configuration>
<mainClass>com.blu.imdg.Helloignite</mainClass>
```

```
<attachToBuild>true</attachToBuild>
<classifier>onejar</classifier>
<filename>Helloignite-runnable.jar</filename>
</configuration>
<goals>
  <goal>one-jar</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

---

**Step 4.** In the `src/main/java/com/blu/imdg` directory, you can add any Java classes you want. To maintain consistency with the rest of this guide, create the following java class: `HelloIgnite.java` in this directory.

Listing 2.17

---

```
package com.blu.imdg;

public class Helloignite {}
```

---

**Step 5.** Add all the following libraries after the package statement.

Listing 2.18

---

```
import org.apache.ignite.Ignite;
import org.apache.ignite.IgniteCache;
import org.apache.ignite.Ignition;
import org.apache.ignite.configuration.IgniteConfiguration;
import org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi;
import org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder;
```

---

**Step 6.** Now that you have a Java class, copy the following lines of codes into the class.

**Listing 2.19**

---

```
public static void main(String[] args) {
    System.out.println("Hello Ignite");
    // create a new instance of TCP Discovery SPI
    TcpDiscoverySpi spi = new TcpDiscoverySpi();
    // create a new instance of tcp discovery multicast ip finder
    TcpDiscoveryMulticastIpFinder tcMp = new TcpDiscoveryMulticastIpFinder();
    tcMp.setAddresses(Arrays.asList("localhost")); // change your IP address here
    // set the multi cast ip finder for spi
    spi.setIpFinder(tcMp);
    // create new ignite configuration
    IgniteConfiguration cfg = new IgniteConfiguration();
    cfg.setClientMode(true);
    // set the discovery spi to ignite configuration
    cfg.setDiscoverySpi(spi);
    // Start ignite
    Ignite ignite = Ignition.start(cfg);
    // get or create cache
    IgniteCache<Integer, String> cache = ignite.getOrCreateCache("HelloWorld");
    // put some cache elements
    for (int i = 1; i <= 100; i++) {
        cache.put(i, Integer.toString(i));
    }
    // get them from the cache and write to the console
    for (int i = 1; i <= 100; i++) {
        System.out.println("Cache get:" + cache.get(i));
    }
    // close ignite instance
    ignite.close();
}
```

---

The program should be conversant to anyone who has some programming experience in Java. Let's carefully examine each part of the program. It has a `main()` method for executing the program. All Java program begins execution by calling the `main()` method. The next line of the code inside `main()` method outputs the string **Hello Ignite**. Next, we created the instance of a TCP Discovery SPI and set multicast IP finder instance on it.



## Tip

The word *SPI* stands for Service Provider API, which is the description of the classes or interfaces that you extend or implement to achieve a goal. Ignite **TcpDiscoverySpi** is a discovery SPI implementation that uses TCP/IP for node discovery.

Later, we set the multi-cast IP finder for the SPI. When TCP discovery starts, this finder sends a multicast request and waits for some time when others nodes reply to this request with messages containing their addresses. We then create an Ignite configuration instance and set the discovery SPI to the configuration.

After starting the Ignite instance, it joins with an existing Ignite cluster as a client. Next, we created a cache with the name `HelloWorld`, and put 100 entries into it. In the for-each loop, we read those 100 entries from the cache and prints on the console. Finally, we stopped the Ignite client instance.

Now that you have a Java project that is ready to be built with Maven, the next step is to build and run the application.

**Step 7.** To build the Java project, issue the following command in the terminal.

```
mvn clean install
```

The above command runs Maven, telling it to execute the `install` goal. This goal will compile, test, and package your project code, and then copy it into the local dependency repository. The build process will take a few minutes to complete. After successful compilation, an *executable* jar will be created in the project `target` directory.

**Step 8.** Run the application by typing the following command.

**Listing 2.20**

---

```
$ java -jar .\target\Helloignite-runnable.jar
```

---

You should see a lot of logs in the terminal. First, a new Ignite client instance will be created, and it will connect to the random node (in our case, there is only one single node) in the cluster. In the Ignite server console, you should see something like:

```
[17:04:43] Ignite node started OK (id=69608c1e)
[17:04:43] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=1.0GB]
[17:04:43]   ^-- Node [id=69608C1E-44DC-4FDC-979A-33627AF3D963, clusterState=ACTIVE]
[17:04:43] Data Regions Configured:
[17:04:43]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
[17:05:43] Topology snapshot [ver=2, servers=1, clients=1, CPUs=8, offheap=3.2GB, heap=4.6GB]
[17:05:43]   ^-- Node [id=69608C1E-44DC-4FDC-979A-33627AF3D963, clusterState=ACTIVE]
[17:05:43] Data Regions Configured:
[17:05:43]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
[17:05:44] Topology snapshot [ver=3, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=1.0GB]
[17:05:44]   ^-- Node [id=69608C1E-44DC-4FDC-979A-33627AF3D963, clusterState=ACTIVE]
[17:05:44] Data Regions Configured:
[17:05:44]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
```

Figure 2.23

On the other hand, Apache Ignite client node console prints the cache entries by 1 to 100.

```
Cache get>Hello Worlds: 92
Cache get>Hello Worlds: 93
Cache get>Hello Worlds: 94
Cache get>Hello Worlds: 95
Cache get>Hello Worlds: 96
Cache get>Hello Worlds: 97
Cache get>Hello Worlds: 98
Cache get>Hello Worlds: 99
Cache get>Hello Worlds: 100
[17:05:44] Ignite node_stopped OK [uptime=00:00:00.343]
```

Figure 2.24

Let's analyse the cache entries with the Ignite visor administrator console. Apache Ignite visor command line tool provides monitoring and administration capabilities for Ignite cluster. Launch the visor tool with the following command.

`IGNITE_HOME/bin/ignitevisorcmd.sh`



## Tip

Visor command line tool can be used to get statistics about nodes, caches, and tasks in the cluster. It shipped with the Ignite distribution archive and located in the `IGNITE_HOME/bin` directory.

Issue the following command into the visor command tool.

`cache -a`

It will return you the details of the cache statistics for our cache **HelloWorld** as shown in figure 2.25.

```

Cache 'testCache(@c0)':
+-----+
| Name(@)           | testCache(@c0) |
| Nodes             | 1           |
| Total size Min/Avg/Max | 100 / 100.00 / 100 |
| Heap size Min/Avg/Max | 0 / 0.00 / 0   |
| Off-heap size Min/Avg/Max | 100 / 100.00 / 100 |
+-----+

Nodes for: testCache(@c0)
+-----+
| Node ID@(), IP      | CPUs | Heap Used | CPU Load | Up Time    | Size     | Hi/Mi/Rd/Wr |
+-----+
| 69608C1E(@n0), 10.42.68.238 | 8    | 13.11 %   | 0.07 %   | 00:06:04.323 | Total: 100 | Hi: 0   |
|                               |       |            |           |             | Heap: 0   | Mi: 0   |
|                               |       |            |           |             | Off-Heap: 100 | Rd: 0   |
|                               |       |            |           |             | Off-Heap Memory: 0 | Wr: 0   |
+-----+

```

Figure 2.25

You notice that the *total cache size* is *100* and the *Offheap size* is also *100*. Apache Ignite stores cache entries into the Offheap memory by default from the version 2.0.1. In chapter 4, we will look at the OffHeap memory and the difference between the OnHeap and the OffHeap memory. Also note that, to make the example as simple as possible, we didn't use any spring related configuration here in this application.

## Apache Ignite thin client

From the version 2.4.0, Apache Ignite introduced a new way to connect to the Ignite cluster, which allows communication with the Ignite cluster without starting an Ignite client node. Historically, Apache Ignite provides two notions of nodes: client and server nodes. Ignite client node intended as a lightweight mode, which does not store data (however, it can store near cache), and does not execute any compute tasks. Mainly, the client node is used to communicate with the server remotely and allows manipulating the Ignite Caches using the whole set of Ignite API's. There are two main shortcomings with the Ignite Client node:

1. Whenever an Ignite client node connects to the Ignite cluster, it becomes the part of the cluster topology. The bigger the topology is, the harder it is for maintaining.
2. In a client mode, Apache Ignite node consumes a lot of resources for performing cache operations.

To solve the above problems, Apache Ignite provides a new binary client protocol for implementing thin client in any programming language or platforms.



### Info

Word **thin** means that it doesn't start any Ignite node for communicating with the Ignite cluster and doesn't implement any discovery or communication SPI logic

Thin client connects to the Ignite cluster through a TCP socket and performs CRUD operations using a well-defined binary protocol. The protocol is a fully socket-based; request-response style protocol. The protocol is designed to be strict enough to ensure standardization in the communication (such as connection handshake, message length, etc.), but still flexible enough for developers to expand upon the protocol to implement custom features. Apache Ignite provides brief data formats and communication details in the [documentation<sup>18</sup>](#) for using the binary protocol.

Ignite already supports .NET, and Java thin client builds on top of this protocol and plans to release a thin client for key languages such as *goLang*, *python*, etc. However, you can implement your thin client in any preferred programming language of your choice by using the binary protocol.



## Warning

The performance of the Apache Ignite thin client is slightly lower than Ignite client node as it works through an intermediary node. Assume that, you have two nodes of the Apache Ignite A, B and you are using a thin client C for retrieving data from the cluster. With the thin client C, you have connected to the node B, and whenever you try to retrieve any data that belongs to the node A, the requests always go through the client B. In case of the Ignite client node, it sends the request directly to the node A.

Most of the times, you should not care about how the message formats look like, or the socket handshake performs. Thin client for every programming language encapsulates the plain hard work under the hood for you. Anyway, if you want to have a deep dive into the Ignite binary protocol or have any issue to create your thin client, please refer to the Ignite documentation.

Before moving on to more advanced topics, let's have a look at a simple application to use Ignite thin client. In this application, I show you the bits and pieces you need to get started with the thin client. The source code for the examples is available at the [GitHub repository<sup>19</sup>](#), see chapter-2.

**Step 1.** Clone or download the project from the GitHub repository. If you are planning to develop the project from scratch, add the following Maven dependencies in your *pom.xml* file. The only `ignite-core` library is needed for the thin client, the rest of the libraries only used for logging.

<sup>18</sup><https://apacheignite.readme.io/docs/binary-client-protocol>

<sup>19</sup><https://github.com/srecon/the-apache-ignite-book>

Listing 2.21

---

```
<dependency>
  <groupId>org.apache.ignite</groupId>
  <artifactId>ignite-core</artifactId>
  <version>2.6.0</version>
</dependency>
<!-- Logging with SLF4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.6.1</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.0.1</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-core</artifactId>
  <version>1.0.1</version>
</dependency>
```

---

Step 2. Now, let's create a new Java class with the name HelloThinClient.

Step 3. Copy and paste the following source code into the newly created Java class. Do not forget to save the file.

Listing 2.22

---

```
import org.apache.ignite.IgniteException;
import org.apache.ignite.Ignition;
import org.apache.ignite.client.ClientCache;
import org.apache.ignite.client.IgniteClient;
import org.apache.ignite.configuration.ClientConfiguration;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloThinClient {
    private static final Logger logger = LoggerFactory.getLogger(HelloThinClient.class);
    private static final String HOST = "127.0.0.1";
    private static final String PORT = "10800";
    private static final String CACHE_NAME= "thin-cache";
```

```

public static void main(String[] args) {
    logger.info("Simple Ignite thin client example working over TCP socket.");

    ClientConfiguration cfg = new ClientConfiguration().setAddresses(HOST+":"+PORT);

    try (IgniteClient igniteClient = Ignition.startClient(cfg)) {

        ClientCache<String, String> clientCache = igniteClient.getOrCreateCache(CACHE\
_NAME);
        // put a few value
        clientCache.put("Moscow", "095");
        clientCache.put("Vladimir", "033");
        //get the region code of the Vladimir
        String val = clientCache.get("Vladimir");

        logger.info("Print value: {}", val);
    } catch(IgniteException e){
        logger.error("Ignite exception:", e.getMessage());
    } catch(Exception e){
        logger.error("Ignite exception:", e.getMessage());
    }
}
}

```

---

**Step 4.** Let's have a closer look at the application we have written above.

```

private static final Logger logger = LoggerFactory.getLogger(HelloThinClient.class);
private static final String HOST = "127.0.0.1";
private static final String PORT = "10800";
private static final String CACHE_NAME= "thin-cache";

```

First, we have declared a few constants: *logger*, *host IP address*, *port*, and the *cache name* that we are going to create. If you have a different IP address, you must change it here; Port **10800** is the default for Ignite thin client.

```
ClientConfiguration cfg = new ClientConfiguration().setAddresses(HOST+":"+PORT);
```

These are our next exciting line in the program. We have created an instance of the Ignite *ClientConfiguration* and passed the address we declared above. In the next try-catch block, we have defined a new cache with the name *thin-cache* and put two key-value pairs. We also used *Ignition.startClient()* method to initialize a connection to Ignite node.

```
try (IgniteClient igniteClient = Ignition.startClient(cfg)) {
    ClientCache<String, String> clientCache = igniteClient.getOrCreateCache(CACHE_NAME);
    // put a few value
    clientCache.put("Moscow", "095");
    clientCache.put("Vladimir", "033");
    // get the region code of the Vladimir
    String val = clientCache.get("Vladimir");
    logger.info("Print value: {}", val);

} catch(IgniteException e){

    logger.error("Ignite exception:", e.getMessage());
} catch(Exception e){
    logger.error("Ignite exception:", e.getMessage());
}
}
```

Later, we retrieved the value of the key **Vladimir** and printed the value in the console.

**Step 5.** Start your Apache Ignite single node cluster if it is not started yet. Use the following command in your preferred terminal.

```
$ IGNITE_HOME/bin/ignite.sh
```

**Step 6.** To try out the build, issue the following at the command line.

```
$ mvn clean install
```

After successful compilation, an executable jar named *HelloThinClient-runnable.jar* will be created in the project target directory.

**Step 7.** Run the application by typing the following command.

```
$ java -jar .\target\HelloThinClient-runnable.jar
```

You should see a lot of logs into the terminal. At the end of the log, you should find something like this.

```
2018-07-30 16:03:18 [main] INFO com.blu.imdg.HelloThinClient - Simple Ignite thin client example working over TCP socket.
2018-07-30 16:03:19 [main] INFO com.blu.imdg.HelloThinClient - Print value: 033
```

Figure 2.26

The application connected through the TCP socket to the Ignite node and performed a put and get operations on cache `thin-cache`. If you take a look at the Ignite node console, you should notice that Ignite cluster topology has not been changed.

```
[16:01:26] Ignite node started OK (id=784af67b)
[16:01:26] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=1.0GB]
[16:01:26]   ^-- Node [id=784AF67B-C199-4939-BE16-04583365C6FE, clusterState=ACTIVE]
[16:01:26] Data Regions Configured:
[16:01:26]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
```

Figure 2.27

## Using REST API for manipulating the Apache Ignite caches

Querying the database tables or caches (in Apache Ignite terms) is an essential part of any application. Apache Ignite out-of-the-box provides REST based API to query over Ignite caches. The REST services are the most straightforward way of accessing the Ignite data store. REST APIs can be used for performing diverse operations like read/write from/to cache, execute tasks, get various metrics and much more. Internally, every Apache Ignite node uses a single embedded *Jetty servlet container* to provide an HTTP server feature.



### Info

Copy and restart the Apache Ignite node if you have not yet copied the `$IGNITE_HOME\libs\optional\ignite-rest-http` modules to the `$IGNITE_HOME\libs\ignite-rest-http` directory from the *install and running Apache Ignite* section. Any extra configurations are needed. The Jetty server will start up and run on port **8080**.

You can use your favorite web browser or any REST client for executing a REST command. For instance, open your preferred browser and type the following URL into the address bar of your browser.

[http://IP\\_ADDR:8080/ignite?cmd=getOrCreate&cacheName=testCache](http://IP_ADDR:8080/ignite?cmd=getOrCreate&cacheName=testCache)

Where,

- *IP\_ADDR* is the IP address of your local machine or the Apache Ignite nodes;

- *cmd* is the Apache Ignite REST command parameter, for instance, *getOrCreate*. This command will create a new cache if it doesn't exist.
- *cacheName* is referred to the name of the cache which will be created or exists. In our case, this is *testCache*.

Inputting the above URL into the browser address bar will return you the following JSON response.

```
{
  "error": "",
  "response": null,
  "sessionToken": "",
  "successStatus": 0
}
```

Always keep in mind that, REST API regularly returns JSON object which is very similar for each command. This JSON object has the following structure:

Name	Type	Description	Example
affinityNodeId	string	Affinity node ID	2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37
error	string	The field contains a description of error if server could not handle the request	
response	jsonObject	The field contains result of the command	
successStatus	integer	Exit status code. It might have the following values: success = 0, failed = 1, authorization failed = 2, security check failed = 3	0



## Tip

If you are using Maven to manage dependencies of your project, you can add REST-HTTP module dependency like this (replace '\${ignite.version}' with actual Ignite version you are interested in).

**Listing 2.23**

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-rest-http</artifactId>
    <version>${ignite.version}</version>
</dependency>
```

---

Let's put a few cache elements (entries) by using the rest API. Type the following URL in your browser.

[http://IP\\_ADDR:8080/ignite?cmd=put&key=moscow&val=777&cacheName=testCache](http://IP_ADDR:8080/ignite?cmd=put&key=moscow&val=777&cacheName=testCache)

Where,

- cmd = REST command put. *Put* commands stores the given key-value pair into the cache.
- key = Key of the cache, it must be unique. In our case, it is *Moscow*.
- value = Value of the cache. In our case, it is the region code of the Moscow, code 777.
- cacheName = the name of the cache. For instance, *testCache*.

This above REST command will print the following output:

```
{
    "successStatus":0,
    "affinityNodeId":"20199df4-7cab-4587-9046-d0c74648fc39",
    "error":null,
    "response":true,
    "sessionToken":null
}
```

Now, we can use the Ignite command line tool `ignitevisor` for analyzing the Ignite cache. Open any terminal and run the following command.

```
$ ignitevisorcmd.sh
```

It should open up the console of the ignitevisor tool. Type the command *open* and type *0* again to select the configuration. And then type the `cache -scan` command, which will print the following messages.

```
visor> cache -scan
Time of the snapshot: 2018-04-07 11:10:27
+-----+
| # |     Name(@)    | Mode      | Size (Heap / Off-heap) |
+-----+
| 0 | testCache(@c0) | PARTITIONED | min: 1 (0 / 1)
|   |                   |           | avg: 1.00 (0.00 / 1.00)
|   |                   |           | max: 1 (0 / 1)
+-----+
Choose cache number ('c' to cancel) [c]: 0
Entries in cache: testCache
+-----+
|   Key Class    |   Key    |   Value Class   |   Value  |
+-----+
| java.lang.String | moscow | java.lang.String | 777    |
+-----+
visor> █
```

Figure 2.28

Now, cache size is *one*, because we have just put one element in it. If you change the *key-value* of the cache entry in the above rest API and type the URL again, it puts one more cache entry into the cache, and the cache size increases. For retrieving the element from the Ignite cache, type the following URL in the browser.

[http://IP\\_ADDR:8080/ignite?cmd=get&key=moscow&cacheName=testCache](http://IP_ADDR:8080/ignite?cmd=get&key=moscow&cacheName=testCache)

Where,

- cmd = REST command *get*. Get command retrieves value mapped to the specified key from the cache.
- key = unique key of the cache entry, for example, *Moscow*.
- cacheName = the name of the cache, in our case, it should be *testCache*.

After hitting the enter, you should get the following JSON output into the browser.

```
{
  "successStatus":0,
  "affinityNodeId":"20199df4-7cab-4587-9046-d0c74648fc39",
  "error":null,
  "response":true,
  "sessionToken":null
}
```

It's also possible to get more than one value from the cache with one rest API call. Here is a sample of the REST command `getall` as shown below.

[http://IP\\_ADDR:8080/ignite?cmd=getall&k1=moscow&k2=vladimir&k3=tver&cacheName=testCache](http://IP_ADDR:8080/ignite?cmd=getall&k1=moscow&k2=vladimir&k3=tver&cacheName=testCache)

The above URL will return the following result.

```
{
  "affinityNodeId":"",
  "error":"",
  "response": {
    "moscow":"777",
    "tver":"39",
    "vladimir":"33"
  },
  "sessionToken":"",
  "successStatus":0
}
```

As a part of the REST API, Apache Ignite provides a few ways that we can use for executing SQL queries over Ignite cache. Let's run an example of the SQL queries through the REST API. Assume, we have already created the `EMP` table and insert a few records into it. Please refer to the section *Using Apache Ignite SQLLINE command tool* for more information on how to create and manage tables into the Ignite.

Now that, we have the table `EMP`. Type the following URL into the address bar of the browser.

[http://IP\\_ADDR:8080/ignite?cmd=qryfldexe&pageSize=10&cacheName=SQL\\_PUBLIC\\_EMP&qry=select+1ename+job+from+emp](http://IP_ADDR:8080/ignite?cmd=qryfldexe&pageSize=10&cacheName=SQL_PUBLIC_EMP&qry=select+1ename+job+from+emp)

Where,

- cmd= REST command `qryfldexe`, runs SQL field query over Ignite cache.

- pageSize=10, number of the page size.
- cacheName= SQL\_PUBLIC\_EMP, the physical name of the cache with the schema name. SQL\_PUBLIC is the public schema for the EMP table.
- qry=select+ename+job+from+emp. Encoded SQL fields query which will be executed over table EMP.

The above command is the same as asking the Ignite: fetch all records from the table EMP.

```
{  
    "successStatus":0,  
    "error":null,  
    "response":{  
        "items": [  
            [  
                "BLAKE"  
            ],  
            [  
                "CLARK"  
            ],  
            [  
                "KING"  
            ]  
        ],  
        "last":true,  
        "queryId":2,  
        "fieldsMetadata": [  
            {  
                "schemaName": "",  
                "typeName": "EMP",  
                "fieldName": "JOB",  
                "fieldTypeName": "java.lang.String"  
            }  
        ]  
    },  
    "sessionToken": null  
}
```

However, the Ignite REST API is elegant and easy to use, and there are a few drawbacks of using RESTful API. First of all, through REST API, you are always connecting to a particular node of the cluster. If the node goes down, your application will start getting errors. This problem could be solved by using any reverse proxy server, for example, Nginx. Secondly

the performance of the REST API will be slower than the Ignite native client node, because the node that executes the API will be intermediary node if the asking dataset will reside on the other node. In this case, the dataset will be transferred through multiple Ignite nodes, and the network roundtrip will be increased.

Ignite REST API feature is the solution for you if you want to build your custom cluster management tool or develop your web application based on RESTful architecture. See the [Ignite API reference<sup>20</sup>](#) for full details with examples.

## Configuring a multi-node Ignite cluster in different hosts

So far, we have installed a single Ignite node on a single Linux or Mac machine and then start creating caches and *inserting/modifying/querying* data through API or SQL queries. That's fine, but Ignite is a distributed NoSQL database for scaling out and running across many different machines in a cluster or ring. You have to do a few exercises for installing and running Ignite in a multinode cluster. This section of this chapter gives you some guidance for configuring a multi-node Ignite cluster in different hosts.

This approach is very similar to configure a single-host cluster. However, if you are new to Apache Ignite and just now starting to kick the tires, it should hopefully guide you through the process.

**Step 1.** Install Java 1.8 on each host.

**Step 2.** Download and Install an Apache Ignite on each host. See the section «Installing and setting up Apache Ignite» for more details.

**Step 3.** Now, on each host, open (unblock) a few ports that allow the nodes to interconnect with each other. Open the `iptables` file if you are using Linux or Mac machine, and add the following commands.

---

<sup>20</sup><https://apacheignite.readme.io/docs/rest-api#sql-query-execute>

**Listing 2.24**

---

```
-A INPUT -m state --state NEW -m tcp -p tcp --dport 47500:47509 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 47400 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 47100 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 47101 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 48100 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 48101 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 31100 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 31101 -j ACCEPT
```

---

For Windows Vista, 7, and 2008, you can use the command `netsh advfirewall` for unblocking specific IP address.

**Step 4.** Restart the *iptable* service as well.

```
/etc/init.d/iptables restart
```

**Step 5.** Next, we have to make only one change in a spring-config file (for instance, `example-cache.xml`). We should replace the IP address `127.0.0.1` with the actual host IP address as follows.

**Listing 2.25**

---

```
<?xml version="1.0" encoding="UTF-8"?>
<bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticast\
\\ pFinder">
    <property name="addresses">
        <list>
            <value>HOST_IP_ADDRESS:47500..47509</value>
        </list>
    </property>
</bean>
```

---

Now, you can start/restart each instance of the Apache Ignite on the separate host machines, and every Ignite node should locate each other.



## Warning

If something went wrong, double check the *iptable* file settings, and the actual IP address of the host machine.

**Step 6.** You can use the Ignite visor command line tool for monitoring the cluster topology. Run the following command in the terminal.

```
$ IGNITE_HOME/bin/ignitevisorcmd.sh
```

Type the command *open* and choose the local configuration file such as *0* or *1*. Then, type the command *top* to see full cluster topology. The command should print the very similar results that are shown in the figure 2.29.

Int./Ext. IPs	Node IDB(@)	Node Type	OS	CPU	MACs	CPU Load
0:0:0:0:0:0:1	1: 7286CB34(@n0)	Server	Mac OS X x86_64 10.11.6	8	46:97:7F:66:5C:2C	0.23 %
127.0.0.1	2: 6B0F819D(@n1)	Server			80:E6:50:07:70:B6	
192.168.1.36						

Figure 2.29

Ignite Visor CLI can be employ to get statistics about nodes, caches, and running tasks in the cluster. Also, Visor command line interface allows you to start and stop remote nodes. To get help and get started, type *help* or *?* commands.

## A simple checklist for beginners

If you are running Apache Ignite for the first time, probably you would face some difficulties. You have just downloaded the Apache Ignite, run it for a few times, and get some issues. Mostly, these problems are solved in a similar fashion. Therefore, I decided to create a checklist, which provides recommendations to help you avoid issues in the development environments.

**1. Configuration files.** When Ignite starting on a standalone mode by executing the `ignite.sh|bat` file, Ignite uses the `$IGNITE_HOME/config/default-config.xml` configuration file. In this situation, to connect to the specified node from the *Visor* command line console, you should choose the `default-config.xml` file from the configuration file list. Most of the time, the `default-config.xml` file is the first file in the list.

You have to run the following command to execute an Ignite node with your *own Spring configuration* file:

```
{IGNITE_HOME}/bin/ignite.{bat|sh} FILE_PATH/my-ignite-example.xml
```

or copy the `my-ignite-example.xml` file in the `$IGNITE_HOME/example/config` directory and execute the `ignite.{bat|sh}` command as follows:

```
{IGNITE_HOME}/bin/ignite.{bat|sh} examples/config/my-ignite-example.xml
```

**2. Ports.** By default, Ignite uses the following local ports:

TCP/UDP	Port number	Description
TCP	10800	Default port for thin client connection
TCP	11211	Default JDBC port
TCP	47100	Default local communication port
UDP	47400	
TCP	47500	Default local discovery port
TCP	8080	Default port for REST API
TCP	49128	Default port for JMX connection
TCP	31100~31200	Default time server port
TCP	48100~48200	Default shared memory port

If you are using docker/virtual machine for up and running Ignite node, you should open the above ports to communicate from your host machine.

**3. Logs.** Log files are tracking events that happen when running Ignite. A log file is very useful to find out what happened with the Ignite application. If you have encountered a problem, and asked a question in Ignite forums, first of all, you will be asked for the log file. Ignite logging is enabled by default, but there are a few drawbacks. In default mode, Ignite writes not so much logging information's on the console (stdout). In the console, you see only the errors; everything else will be passed to the file. Ignite log files are located on the `$IGNITE_HOME/work/log` directory by default. Do not erase log files and keep logs as long as possible, and it would be handy for debugging any serious errors.

However, if you want to quickly find out the problems without digging into separates log files, you can execute Ignite in verbose mode.

```
$ ignite.sh -v
```

In verbose mode, Ignite writes all the logging information both on the console and into the files. Note that, Ignite runs slowly in verbose modes, and not recommended to use in a production environment.

**4. Network.** If you encountered strange network errors. For instance, if a network *could not connect* or *could not send the message*, most often you unfortunately hit by the IPv6 network problem. It can't be said that Ignite doesn't support the *IPv6* protocol, but at this moment, there are a few specific problems. The easiest solution is to disable the *IPv6* protocol. To disable the *IPv6* protocol, you can pass a Java option or property to the JVM as follows:

-Djava.net.preferIPv4Stack=true

The above JVM option forces the Ignite to use IPv4 protocols and solves a significant part of the problems related to the network.

**5. Ghost nodes.** One of the most common problems that many people encountered several times whenever they launched a new Ignite node. You have just executed a single node and encountered that you already have a two server Ignite nodes in your topology. Most often, it may happen if you are working on a home office network and one of your colleagues also run the Ignite server node at the same time. The fact is that by default, Ignite uses the multicast protocol to discover and communicate with other nodes. During startup, Ignite search for all other nodes that are in the same multicast group and located in the same subnetwork. Moreover, if it does, it tries to connect to the nodes.

The easiest way to avoid this situation is to configure static IP instead of *TcpDiscoveryMulticastIpFinder*. Therefore, use *TcpDiscoveryVmIpFinder* and write down all the IP address and ports to which you are going to connect. These particular configuration helps you to protect from the ghost nodes in a development environment.

**6. Baseline topology.** Ignite baseline topology was introduced on version 2.4.0 and became a convenient way to protect the durability of the data through native persistence. See more about Ignite native persistence and baseline topology in *chapter 4*. However, what's wrong with the Ignite Baseline topology? To answer the question, let's imagine the following scenario:

- We have launched a single Ignite node with native persistence enable (data will be written to the disk).
- We activated the cluster because we enable native persistence for the node.
- We have created a REPLICATED cache and loaded some data on it.
- Next, we launched two more nodes and start manipulating with data, insert/delete some data.

At this moment, each node contains the full copy of the data and works well. After a while, we decided to restart one of the nodes. If we stop the very first node from which we start, then everything breaks, and the data is lost. The reason for this strange behavior is the Ignite baseline topology, a set of server nodes that stores persistence data. In the rest of the nodes, data will not be persisted.

Set of the server nodes is determined for the first time at the moment of the cluster activation. So, the rest of the server nodes that you added later will no longer be included in the baseline

topology. Thus, in our case, the set of the baseline topology consists of only one server node and this node persists data on disk. Whenever you stop this server node, everything breaks. Therefore, to prevent from this surprise, start all the nodes of the cluster first, and only then activate the cluster.

So, we can point out the following shortlist for the beginners:

N	Check it out
1	Use proper configuration files to connect through Ignite Visor.
2	Open ports that you need to work with the Ignite node.
3	Configure & read logs.
4	Avoid IPv6.
5	Use TcpDiscoveryVmIpFinder on the home office network.
6	Keep track of the Baseline topology.

## Summary

Throughout this chapter, you have learned the basics of the Ignite distributed database. We have demonstrated the installation and setting up guidelines for beginners who are the newbie in the Ignite world. Later in this chapter, you learned how to write, compile and execute a simple application to work with Ignite cluster. Moreover, you have covered much ground to use various Ignite clients for querying Ignite database, such as SQL, Thin client or Rest API. We have introduced a few fundamental concepts like Ignite Visor command line interface for monitoring Ignite cluster.

## What's next?

In the next chapter, we will move away from the code samples you have seen so far, and discuss various use cases where you can use and deploy Ignite database for solving different kinds of architecture problems.

# Chapter 3. Apache Ignite use cases

The in-memory database use-cases are wide and varied. Roughly, any application that can benefit from a performance boost can potentially profit from using an in-memory database. Consider the following traditional three-tier application architecture as shown in figure 3.1.

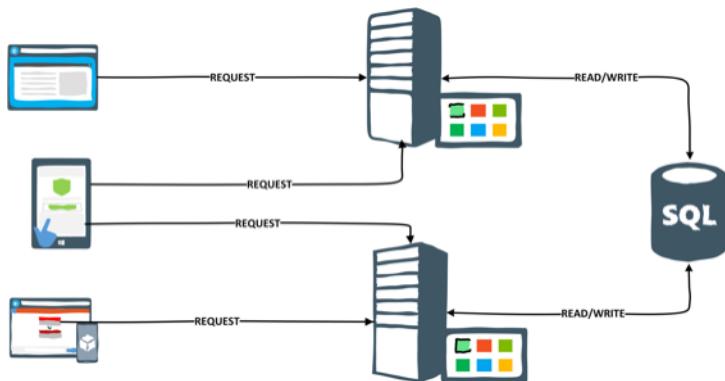


Figure 3.1

The traditional application architecture uses data stores which have synchronous read-write operations. This is useful for data consistency and data durability, but it can easily have a bottleneck if there are many transactions waiting in the queue.

An in-memory database is NOT just for caching and storing datasets; it can offer much more. An in-memory database can be used for acting in Real-Time. Real-Time event processing and fault-tolerant move the in-memory database to another level. In this chapter we are going to highlight a few use cases demonstrating how to use Apache Ignite to support high-performance, real-time applications, and without comprising the speed or safety of valuable data. And we will have a detailed look at most of these use cases and explains how to implements them in the later chapters.

## Caching for fast data access

The performance of the web application varies, and so do user expectations. Research shows that speed matters on every context. The hunger for speed on loading websites, mobile applications are increasing every year. Apache Ignite in-memory database can serve as a caching layer for your existing application, providing **fast recall of frequently accessed<sup>21</sup>** data. It sits between the application servers and the data store. Apache Ignite as an in-memory data grid uses a cache of frequently accessed data by the client in the active memory, and then can access the persistence store when needed, and even asynchronously send and receive updates from the persistence store. An application architecture for caching frequently read data is shown below.

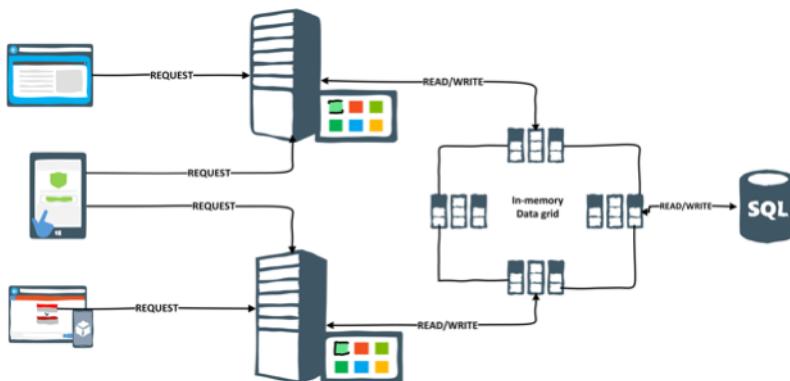


Figure 3.2

Frequently accessed data moves closer to the application endpoint by using Apache Ignite as a caching layer. This approach reduces the response times of the accessing data. Apache Ignite dynamically loaded the data into the memory when the data is first access by a client application. Usually, a client application is not required to implement any in-memory specific code. Most of the object-relational mapping (ORM) library already supports Apache Ignite as a 2<sup>nd</sup> level cache, such as Hibernate or MyBatis.

Often, no persistence is required in caching level, and the caches usually partitioned over the cluster nodes. These allow you to get the low latency up to ~200 microseconds of the response times.

The approaches mentioned above are not only applicable to the web application but can also be used for caching data in BPM or ESB (enterprise service bus) engine. Frequently, each BPM engine uses a database for storing business process context, which is employing to run

<sup>21</sup><https://gist.github.com/jboner/2841832>

instances of the business process. Caching such information in memory can boost the performance of the BPM engine. Similarly, using in-memory caching is also a straightforward way to pass data or share state from one service to another on any ESB application.

## HTAP

HTAP stands for Hybrid Transaction/Analytical Processing, a new term coined by the Gartner a few years ago. Basically, it's a single system that stands for both OLTP (operational) and OLAP (analytical) processing. Data is stored once in memory and so instantly available for analytics.

$$\text{OLTP} + \text{OLAP} = \text{HTAP}$$

Historically, every enterprise maintains multi-tiered database architecture for processing transactions and analytics for getting a business decision. Generally, the traditional data warehouse or OLAP system extracts data from various OLTP using ETL (Extraction Transformation and Loading) tool at a specified time interval per day or per hour load basics. The high-level architecture of the data warehouse system is shown in figure 3.3. This traditional approach has a few overheads:

1. The entire system is very expansive to maintain.
2. The data storage is getting bigger day by day because every business creates massive amounts of data and the success of the company depends on how those data are used efficiently. Data storage becomes the top issue since a massive amount of historical data is stored in the OLAP system.
3. Analytics on real-time data is very hard to achieve. Usually, traveling the data from OLTP to OLAP system through ETL takes longer. The current business trends suggest that even the delay of data per day can create a significant impact on the entire business.

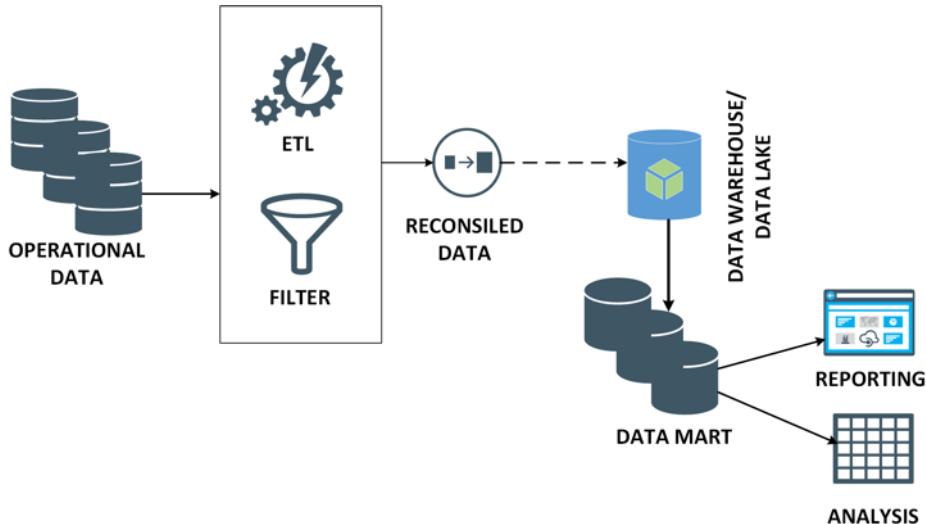


Figure 3.3

All these factors above make a considerable necessity to redesign the OLTP/OLAP system and combine them as a new database. The rise of in-memory and NoSQL databases; it is the right time to check for a new solution that suits both OLTP and OLAP sceneries. OLTP systems should be able to do high-volume transactions, and OLAP system should be able to do useful analysis over the same data, but both operations should not have interfered. An in-memory database like Apache Ignite might write enormously into the in-memory layer, and on the other hand, can persist the data into the disk or any NoSQL/RDBMS for analytical processing on real-time. The ETL stage goes away completely in this approach, which breaches the barrier of doing analytics on real-time data.

As we mentioned earlier, Apache Ignite provides native persistence of the highly transactional data on disk, and also ensure the ANSI SQL for querying the same database. Theoretically, you can build your HTAP class system based on Apache Ignite for any *small or medium-sized* business with these two features. You do not have any separate OLAP system for querying the data in this situation. Most modern BI tools like Tableau, Qlik or Pentaho have in-memory analytics, which is an improvement upon the OLAP data model.

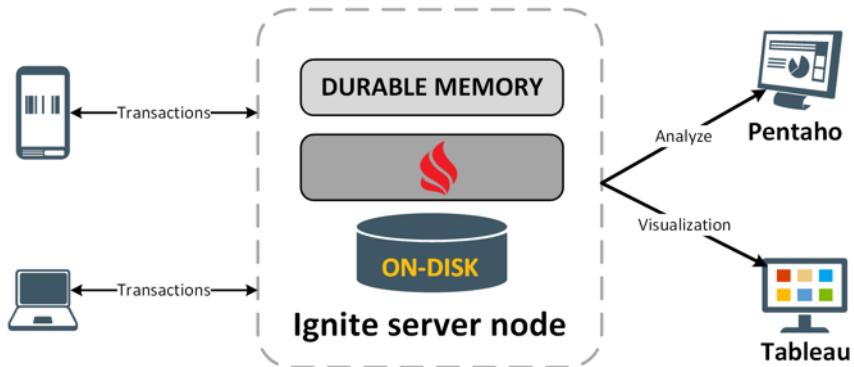


Figure 3.4

You can connect to an Ignite cluster from Tableau and analyze the data in different ways using the Ignite ODBC driver. However, you can also use Pentaho for connecting to the Ignite cluster through JDBC API to extract, transform, visualize, and analyze your data efficiently. Note that, sometimes this approach is also called NewSQL (buzzword for marketing).

HTAP system is getting complicated whenever you need data mining or OLAP cube for querying the data efficiently. Working with OLAP cubes often causes more challenges and require more careful architectural design. Apache Ignite cannot cope alone in such scenario. However, we can combine Apache Ignite with other NoSQL database such as Apache Cassandra to reach the goal. Apache Ignite 3<sup>rd</sup> party persistence feature becomes the foundation of this architecture design. In general, Apache Ignite involves as an OLTP system with full-fledged transactions, which stores the data in-memory and into the external NoSQL database (such as Cassandra) at the same time as shown in figure 3.5.



## Info

Note that *OLAP* cube is a multi-dimensional data structure that includes facts, measures (things you count or add) and dimensions (attributes of the data).

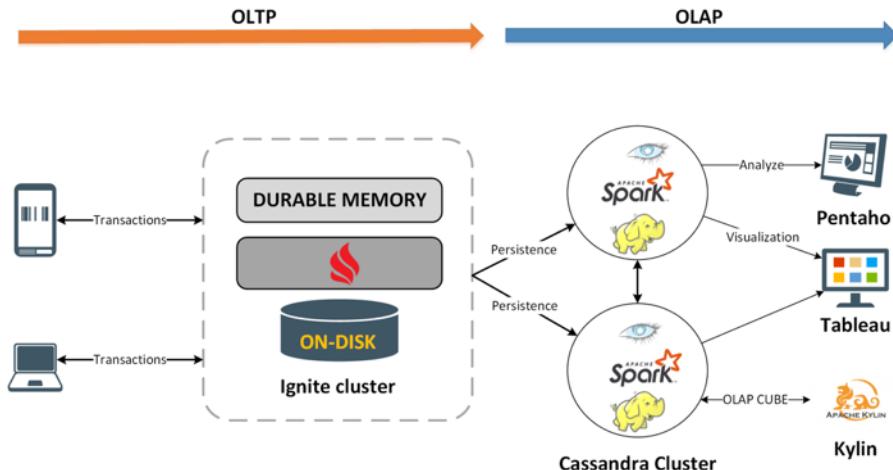


Figure 3.5

The main advantage of the above architecture design is the absence of the ETL stages. Data are immediately available into the Cassandra cluster after committing the transactions into the Ignite cluster. You can read the data from the Cassandra node even in the times of failure of the Ignite node. The second reason for considering having the separate cluster for data persistence is that you can import data from other external sources rather than the Ignite cluster. You can use commodity hardware for building the [data lake<sup>22</sup>](#) or [data marts<sup>23</sup>](#) on specific circumstances by using Cassandra and Spark stack. Such an OLAP system can live and grows independently; depends on your data size.

## High-volume transaction processing

The growth of the cloud computing and the mobile applications in the enterprise and beyond are all driving a need for robust database architecture that can manage real-time data transactions and high-volumes of simultaneous users accessing data around the world. Online shopping; credit or debit card processing always demand extremely high-performance data management. Apache Ignite can handle a large number of concurrent transactions involving petabytes of operational data by committing the transactions into memory in such cases. Apache Ignite also provides scalable architecture than traditional relational database management system, able to elastically scale as demand increases. This

<sup>22</sup>[https://en.wikipedia.org/wiki/Data\\_lake](https://en.wikipedia.org/wiki/Data_lake)

<sup>23</sup>[https://en.wikipedia.org/wiki/Data\\_mart](https://en.wikipedia.org/wiki/Data_mart)

approach reduces the response times and can lower transactions times from a few seconds to fractions of a second.

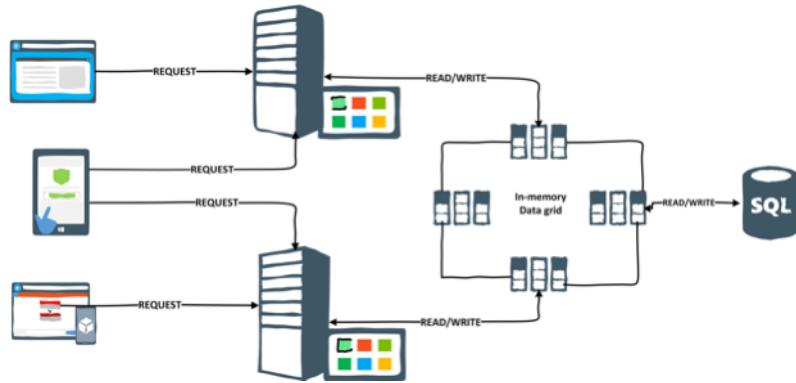


Figure 3.6

Moreover, Apache Ignite native persistence trade the full durability of the committed transactions for enhanced runtime performance for transactional workloads. Durability is provided via transactions log (WAL – write-ahead log) that are persisted on appropriate device (e.g., HDD or SSD). Each transaction that modified data is written into the log, and the log is used to restore the Ignite node state after a power failure or database/system restart.

## Fast data processing

*Big Data* is one of the talks of the town nowadays; however, it's not only about volume. Big data can be described by the characteristics of 4Vs: *Volume*, *Velocity*, *Variety*, and *Veracity*. The data velocity or flows of the data is one of the primary concern of the Big data. Data motion is massive and continuous in Big data. Data tells the story of what's happening with your business on the background right now. The opportunities to take advantage of the hot data is higher than ever using the IoT<sup>24</sup> as a continuous data source. This real-time data can help businesses make valuable decisions that provide strategic competitive advantages over others. Data processing in motion is often called Fast data.

$$\text{Fast data} = \text{Big Data} + \text{Event processing}$$

Fast Data requires a variety of processing techniques. Data processing could be *rule-based* or *Stream-based* event processing. Apache Ignite allows a few ways for processing continuous

<sup>24</sup>[https://en.wikipedia.org/wiki/Internet\\_of\\_things](https://en.wikipedia.org/wiki/Internet_of_things)

never-ending streams of data in scalable and fault-tolerant fashion in-memory, rather than analyzing data after it's reached the database. One of them is *Ignite RDD*; an implementation of the Spark native RDD and DataFrame API, which shares the state of the RDD across other Spark jobs. The high-level architecture of using IgniteRDD with Spark is shown in figure 3.7.

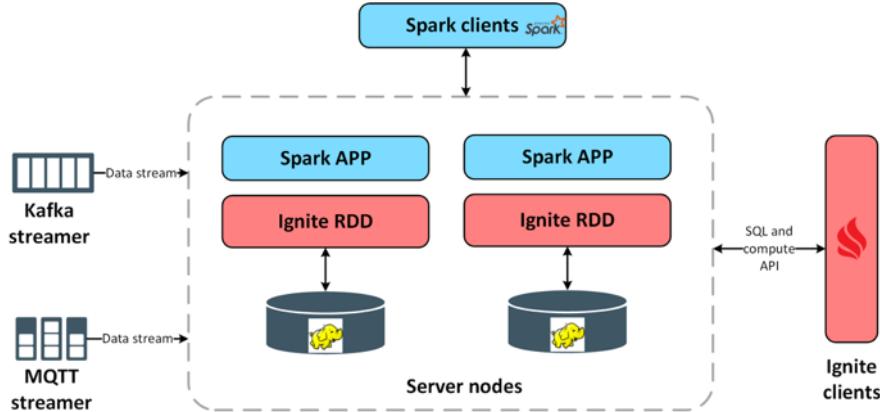


Figure 3.7

Such a design not only enable you to correlate relationships and detect meaningful patterns from the stream data but also help in processing it faster and much more efficiently. Apache Ignite in-memory data grid can manage a tremendous amount of incoming data and push notifications to the business application when changes occur with the server. The Apache Ignite *continuous queries* capability allows systems to quickly access a significant amount of never-ending incoming data and take actions.

## Lambda architecture

The purpose of this data-processing architecture is similar to the fast data processing with a little difference. [Lambda architecture<sup>25</sup>](#) proposed to handle massive quantities of data by taking advantages of both batch-processing and stream-processing methods, where fast data only refers to the stream-processing. Lambda architecture describes a system consisting of three different layers: batch processing, speed (real-time) processing and a serving layer for responding to queries. The high-level design of the lambda architecture is shown in figure 3.8.

<sup>25</sup>[https://en.wikipedia.org/wiki/Lambda\\_architecture](https://en.wikipedia.org/wiki/Lambda_architecture)

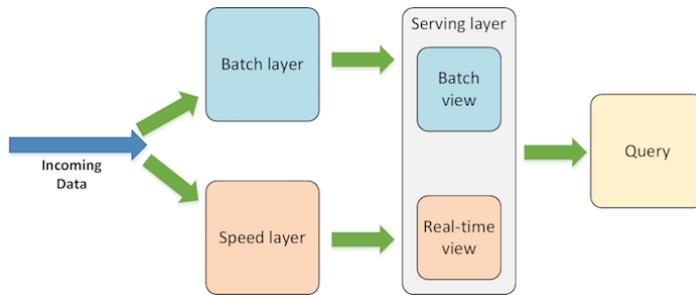


Figure 3.8

Lambda architecture has the following characteristics:

1. All data sent parallel to both the batch and the speed layer.
2. Master data is immutable
3. Batch layer pre-computes query functions from scratch; the result is called batch view.
4. Speed layer compensates for the high latency of updates to the batch views.
5. A query resolved by getting the result from both batch and real-time views.

There are various ways of implementing lambda architecture because, it is independent of underlying solutions for each of the layers. Each layer requires specific features of underlying implementation that could be solved by many different stack technologies:

1. Batch layer: write-once, bulk read many times.
2. Speed layer: random read-write, incremental computation.
3. Serving layer: random read, batch computation and write.

For instance, one of the possible implementations of the Lambda architecture might look as follows with the following stack technologies:

1. Batch layer: Hadoop, Spark
2. Speed layer: Strom or Apache Ignite
3. Serving layer: Hive or Apache Ignite
4. Input source: Kafka

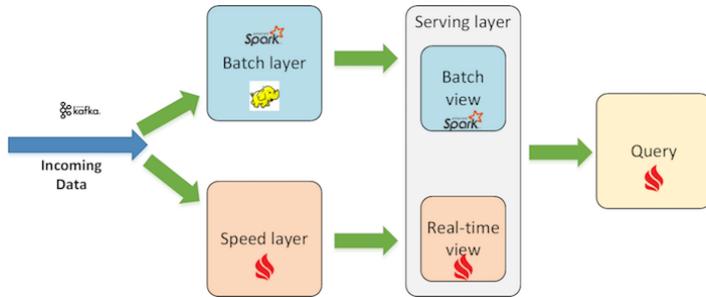


Figure 3.9

You can execute custom logic on incoming data in collocated fashion by using Apache Ignite StreamAPI such as *StreamTransformer* and *StreamVisitor*. You can change the data or add any pre-processing logic to it, before putting the data into the caches. Also, Apache Ignite continuous query can help you to execute queries and then get notifications about the data changes that fall into your query. Note that the above design is one of the possible ways of implementing the lambda architecture, you can use Apache Storm or another stack for your underlying requirements.

## Resilient web acceleration

A web application can easily be scaled out by adding more application server on the cluster. Load balancing between application servers usually provides by adding devices (hardware or software) such as a load balancer. However, web applications that use web sessions or providing stateful services introduce a new problem. A load balancer switches the current user to a new application server that creates a new web session when one of the application servers goes down, and the current web session is lost. This switching causes a few problems which include:

1. User session data has been lost, and any unsaved current data will also be lost.
2. Reduce the performance of the web application. Every time creating a sticky connection in a load balancer is a CPU intensive operation.

The solution to this problem is to provide web session clustering. Apache Ignite in-memory data grid feature provides web session clustering, which can provide fault tolerance to your web application and accelerate your web application's performance. You can share web sessions between web applications through Ignite caches *without changing any application logic code*.

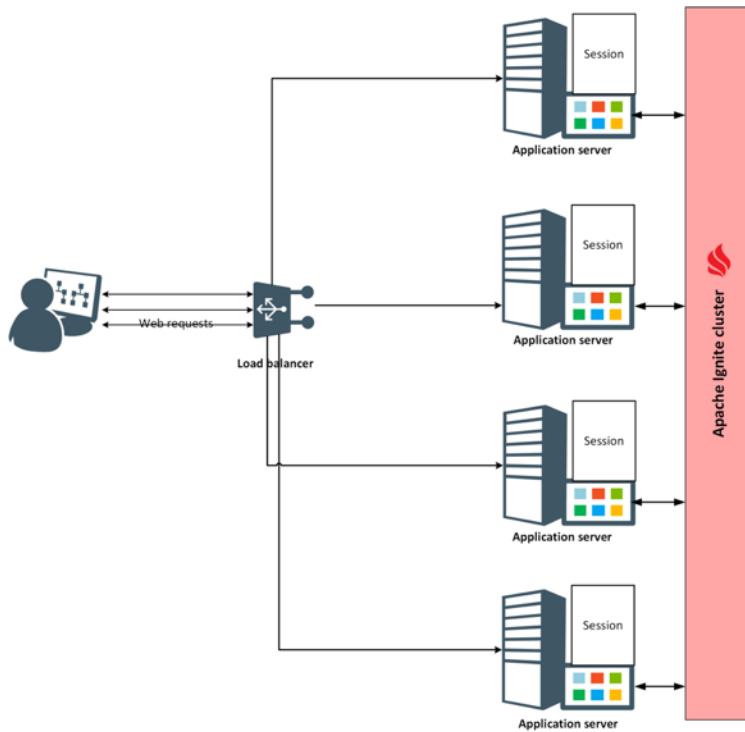


Figure 3.10

The above approach uses Apache Ignite caches as a distributed web session store, shared by all application server instances. In cases of an application server failure, the load balancer redirects the user request to a new application server which has access to the user session through Ignite distributed caches. This approach provides the highest level of high availability of a system and customer experience.

## Microservices in distributed fashion

Microservice architecture has a number of benefits, and enforces a level of modularity that is extremely difficult to achieve with a monolithic application. The idea behind the microservice architecture is developing a small subset of interconnected applications instead of creating a single large application. Every microservice has its own layered architecture similar to the monolith application.

Apache Ignite can provide independent cache nodes to corresponding microservices in the same distributed cluster and gives a few advantages over traditional approaches:

1. Business logic is running alongside with the data in the same node. This strategy reduces the redundant data transformation between an application server and the database.
2. Running as a light-weight Java thread and can use the same JVM resources.
3. Can provide automatic fault tolerance of the service.
4. Can shorten organizations time to market for new services.

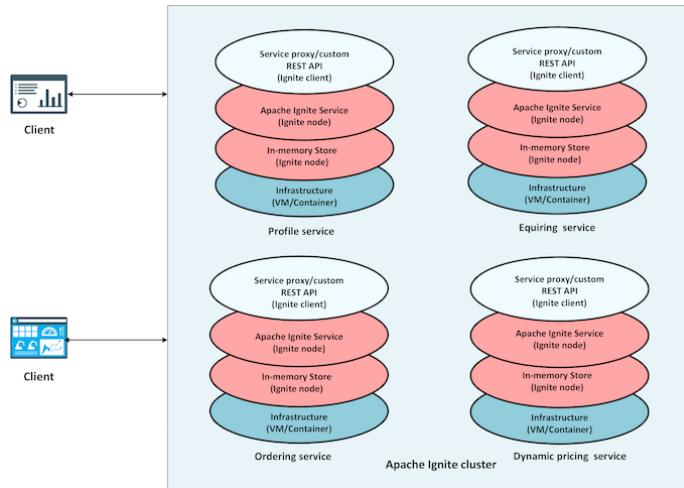


Figure 3.11

Services running on the in-memory cluster is much faster than the disk-based application server. Apache Ignite microservice based service grid provides a platform to automatically deploy any number of the distributed service instance in the cluster.

## Cache as a service

Generally, every organization maintains a few different information systems such as ERP, CRM or a portal for customer services. Such a system employs a separate caching mechanism and layers to speed up the application's performance at times. As the number of systems or services increases, those individual caching layers also increases and finally, it ends up with N numbers of a heterogeneous caching cluster.

Apache Ignite can provide a common caching layer across the organization, which can allow multiple applications to access the managed in-memory cache. You can isolate the caching layer from the applications by separating the caching layer from the applications.

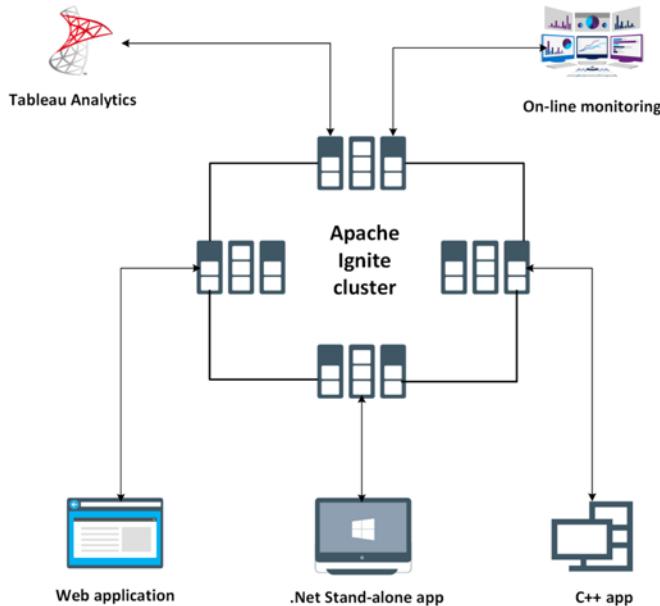


Figure 3.12

Any applications (Java, .Net, C++) across the organization can store and read data from the shared caching layer. It's not necessary to build and deploy local or private caching infrastructure for each application by using an in-memory data grid as a service. Applications can use Apache Ignite as cache aside or write behind to their database or load data from the database into the cache. It eliminates the complexity in the management of the hundred or more separate caching infrastructures under one organization.

## Big Data accelerations

Hadoop has been widely used for its ability to store and analyze large data sets economically and has long passed the point of being nascent technology. However, its batch scheduling overhead, and disk-based data storage have made it unsuitable for use in analyzing live, real-time data in the production environment. One of the main factors that limit performance scaling of Hadoop and Map/Reduce is the fact that Hadoop relies on a file system that generates a lot of input/output (I/O) files. An alternative is to store the needed distributed datasets within the memory or running Map/Reduce function in-memory with the data which can eliminate the I/O latency.

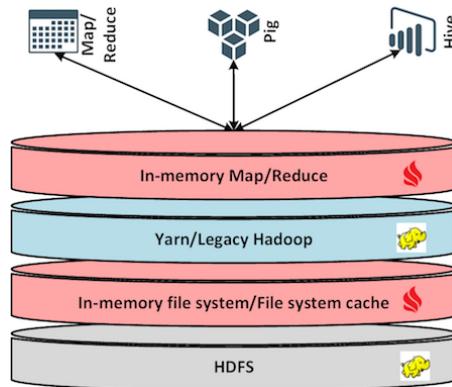


Figure 3.13

Apache Ignite has offered a set of useful components allowing in-memory Hadoop job executing and file system operations. Apache Ignite Hadoop accelerator component can automatically deploy all necessary executable programs and libraries for the execution of Map/Reduce across the JVMs, which significantly reduced startup time down to milliseconds. These speeds up by avoiding delays in accessing secondary storage. Also, key-value pairs, hosted within the data grid can be efficiently read into the execution engine to minimize access time because the execution engine is integrated with the in-memory data grid. This approach is convenient when you have an existing Hadoop Map/Reduce application, and you are willing to enhance the performance of the Map/Reduce execution without changing any code

## In-memory machine learning

For those (especially data scientist) who are working on training model against a massive volume of data deals with two major problems: data (after trained) movement from one environment to other, and the scalability problem of the environment. However, Apache Spark, Mahout or TensorFlow helps to train the model, but will not help you to deploy the model into the production environment. Running Machine Learning (ML) or Deep Learning (DL) algorithms against in-memory database solve the following problems:

1. Running ML or DL against live up-to-date data.
2. Using data locality on each node of the cluster by using co-located distributed processing for getting an extremely high performance of the algorithms.

Apache Ignite (Since version 2.0) provides a set of libraries for building predictive machine learning model to get the most out of the in-memory database. Note that, Apache Ignite machine learning grid is still in beta version at the moment of writing this book, but provides a bunch of machine learning algorithms or ML components such as Linear Regression, K-Means Clustering, Decision trees, etc. for production use.

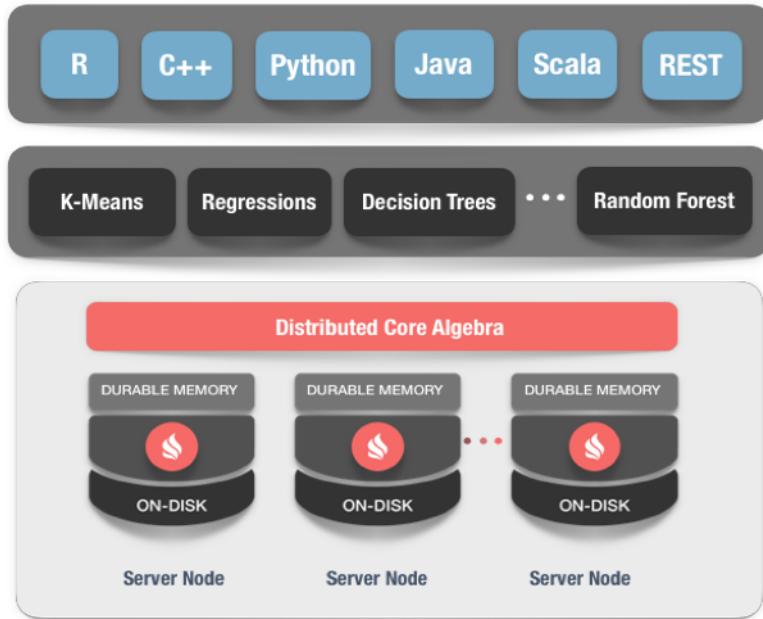


Figure 3.14

Furthermore, Apache Ignite provides two mains components for developing and running ML/DL algorithms on top of the distributed database:

1. *Distributed core algebra.* A set of libraries as a foundation of the ML/DL algorithms. You can develop your own implementation of any algorithms based on the Apache Ignite core algebra API.
2. *Pre-prepared ML/DL algorithms.* A set of pre-prepared production ready ML/DL algorithms and examples delivered with every Apache Ignite distribution. So, you can build and run the existing examples, study their outputs and keep developing your application.

## In-memory geospatial

Geospatial technology such as GIS and GPS is an integral part of our everyday life. The geospatial word indicates the data that has a geographical component to it. It means that the record in a dataset has location information attached to them in the form of coordinates, address, city or ZIP code. Most of them formed by the GIS system, where geographical data has been stored in layers and integrated with geographic software programs. Other geospatial data can originate from GPS data, satellite imagery, etc.

Although, there are a lot of varieties of [database<sup>26</sup>](#) that supported and suitable for geospatial data, but not many in-memory databases provide geospatial support. If you need geospatial support at scale and high throughput, Apache Ignite is a first-class citizen for storing and querying geospatial data. Apache Ignite provides a partial implementation of the [JTS topology suite<sup>27</sup>](#): a set of APIs of spatial predicates and functions for processing geometry. Apache Ignite geospatial library supports the following feature:

1. Datatypes: supports Points, Coordinates, etc.
2. Measurement functions: Polygon, Length, Area, etc.
3. Indexing: parallel and lock-free indexing.
4. Spatial joins: Caches can be joined by their relationships.

## Cluster management

As the last use case, Apache Ignite in-memory data grid feature can be used for managing cluster for your application. Commonly, a cluster manager supports various functions like:

1. Create a highly available environment to continue operating and provide resources to users when a failure occurs.
2. Discover and group membership of nodes in a cluster. For example, whenever a node joins or leaves the cluster.
3. Support distributed Map.
4. Support distributed locks and counters in a cluster.

---

<sup>26</sup>[https://en.wikipedia.org/wiki/Spatial\\_database](https://en.wikipedia.org/wiki/Spatial_database)

<sup>27</sup><http://www.tsusiatsoftware.net/jts/main.html>

So, I encourage you to use Apache Ignite as an implementation of the cluster manager if you are developing a platform or framework with the above requirements. Although, Apache Zookeeper is leading the market for maintaining cluster, you can make attention to the Apache Ignite, because Apache Ignite is small and very easy to use. A few application frameworks such as [Eclipse Vert.x<sup>28</sup>](#) for developing reactive applications uses Apache Ignite as an implementation of their cluster manager.

## Summary

We have explored a few fundamental use cases in this chapter, where in-memory database such as Apache Ignite could be employed. These use cases detailed through the rest of the book.

We have started from the use case of a database caching, which is one of the primary function of the Apache Ignite. We explained that a separate caching layer could move the data closer to the application, which can dramatically reduce the response times of the accessing data. Then we have described a few more Ignite uses cases such as HTAP, Web-session clustering, microservices, etc.

## What's next?

We will look at the Apache Ignite architecture in the next chapter, which will help you to have a better understand of the concepts and pitfalls of this distributed database.

---

<sup>28</sup><https://vertx.io/docs/vertx-ignite/java/>

# **Chapter 4. Architecture deep dive**

Apache Ignite is an open-source memory-centric distributed database, caching and computing platform. It was designed as an in-memory data grid for developing a high-performance software system from the beginning. So its core architecture design is slightly different from the traditional NoSQL databases. It can simplify the developing of modern applications with a flexible data model and simpler high availability and high scalability.

To understand how to properly design an application with any databases or framework, you must first understand the architecture of the database or framework itself. By getting a better idea of the system, you can solve different problems in your enterprise architecture landscape, can select a comprehensive database or framework that is appropriate for your application and can get maximum benefits from the system. This chapter gives you a look at the Apache Ignite architecture and core components to help you figure out the key reasons behind Ignite's success over other platforms.

## Functional overview

The first step in the journey of understanding the architecture of the Apache Ignite is to get a rough idea of the functional capabilities of the platform. Apache Ignite architecture has sufficient flexibility and advanced features that can be used in a large number of different architectural patterns and styles. You can view Ignite as a collection of independent, well-integrated components geared to improve the performance and scalability of your application. The following schematic view represents the basic functionalities of Apache Ignite.

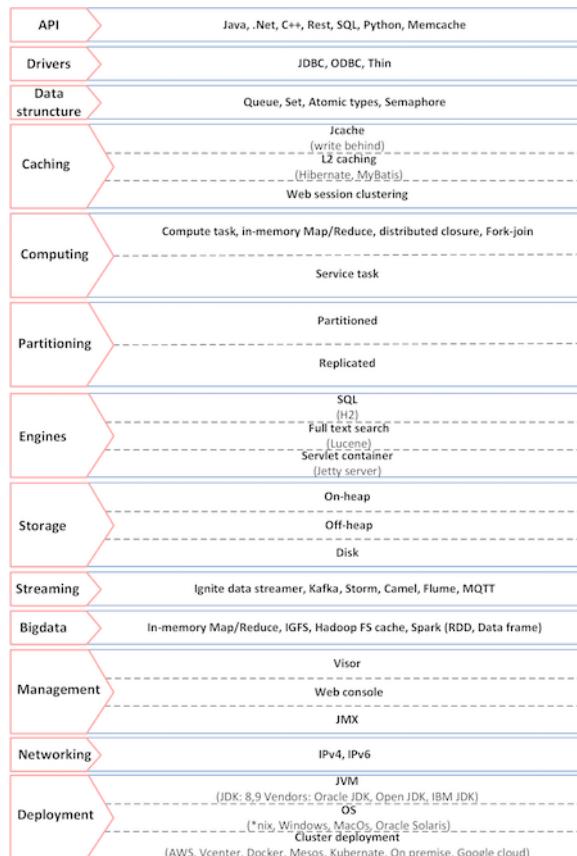


Figure 4.1

Apache Ignite is organized in a modular fashion and, in most cases, provides a single jar (or library) for each functionality. You only have to apply the desired library into your

application or project using Ignite. You can write applications in the language you choose (Java, .Net, C++) using the APIs. The applications connect to an Ignite cluster to perform read and write operations and run SQL queries with low latency and high throughput (millions of operations per second).

## Understanding the cluster topology: shared-nothing architecture

Apache Ignite is a grid technology, and its design implies that the entire system is both inherently available and massively scalable. Grid computing is a technology in which we utilize the resources of many computers (commodity, on-premise, VM, etc.) in a network towards solving a single computing problem in parallel fashion.

Note that there is often some confusion about the difference between grid and cluster. Grid computing is very similar to cluster computing, the big difference being that cluster computing consists of homogeneous resources, while grids are heterogeneous. Computers that are part of a grid can run different operating systems and have different hardware, whereas cluster computers all have the same hardware and OS. A grid can make use of spare computing power on a desktop computer, while the machines in a cluster are dedicated to working as a single unit and nothing else. Throughout this book, we use the terms *grid* and *cluster* interchangeably.

Apache Ignite also provides a [shared-nothing architecture<sup>29</sup>](#) where multiple identical nodes form a cluster with no *single master* or *coordinator*. All nodes in a shared-nothing cluster are identical and run the exact same process. In the Ignite grid, nodes can be added or removed nondisruptively to increase (or decrease) the amount of RAM available. Ignite internode communication allows all nodes to receive updates quickly without having any master coordinator. Nodes communicate using peer-to-peer message passing. The Apache Ignite grid is sufficiently resilient, allowing the nondisruptive automated detection and recovery of a single node or multiple nodes.

On the most fundamental level, all nodes in the Ignite cluster fall into one of two categories: *client* and *server*. There is a big difference between the two types of nodes, and they can be deployed in different ways. In the rest of this section, we will talk about the topology of the Ignite grid and how it can be deployed in real life.

---

<sup>29</sup>[https://en.wikipedia.org/wiki/Shared-nothing\\_architecture](https://en.wikipedia.org/wiki/Shared-nothing_architecture)

## Client and server node

An Ignite node is a single Ignite process running in a JVM. Apache Ignite nodes have an optional notion of client and server nodes as we mentioned before. Often, an Ignite client node also addresses as a *native client node*. Both client and server nodes are part of Ignite's physical grid and are interconnected with each other. The client and server nodes have the following characteristics.

Node	Description
Server	1. Acts as a container for storing data and computing. A server node contains data, participates in caching, computing and streaming. 2. Generally starts as a standalone Java process.
Client	1. Acts as an entry point to run operations like put/get into the cache. 2. Can store portions of data in the near cache, which is a smaller local cache that stores most recently and most frequently accessed data. 3. It is also used to deploy compute and service tasks to the server nodes and can participate in computation tasks (optional). 4. Usually embedded with the application code.



### Tip

You often encounter the term *data node* in the Ignite documentation. The terms *data node* and *server node* refer to the same thing and are used interchangeably.

All nodes in the Ignite grid start as server nodes by default, and client nodes need to be *explicitly* enabled. You can imagine the Ignite client node as a *thick client* (also called a fat client, e.g., Oracle OCI8). Whenever a client node connects to the Ignite grid or cluster, it is aware of the grid topology (data partitions for each node) and is able to send a request to the particular node to retrieve data. You can configure an Ignite node to be either a client or a server via a Spring or Java configuration, as shown below.

**Spring configuration:**

**Listing 4.1**


---

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <!-- Enable client mode. -->
    <property name="clientMode" value="true"/>
    ...
</bean>
```

---

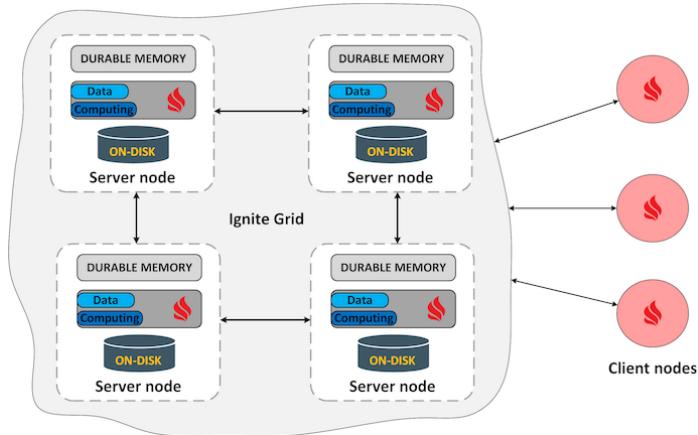
**Java configuration:****Listing 4.2**


---

```
IgniteConfiguration cfg1 = new IgniteConfiguration();
cfg1.setGridName("name1");
// Enable client mode.
cfg1.setClientMode(true);
// Start Ignite node in client mode
Ignite ignite1 = Ignition.start(cfg1);
```

---

Here is also a special type of logical node called a *compute node* in the Ignite cluster. A compute node is the node that usually participates in computing business logic. Basically, a server node that contains data is also used to execute computing tasks.

**Figure 4.2**

However, an Apache Ignite client node can also participate in computing tasks *optionally*. The concept might seem complicated at first glance, but let's try to clarify it.

Server nodes or Data nodes always stores data and participating in any computing task. On the other hand, the Client node can manipulate the server caches, store local data and optionally participate in computing tasks. Usually, client nodes are only used to put or retrieve data from the caches.

Why should you want to run any computing task on client nodes? In some cases (for instance high volume transactions in the server nodes), you do not want to execute any job or computing task on the server nodes. In such a case, you can choose to perform jobs only on client's nodes by creating a cluster group. This way, you can separate the server node (data node) from the nodes that are particular uses for computing in the same grid.

A cluster group is a *logical unit* of a few nodes (server or client node) that group together in a cluster to perform some work. Within a cluster group, you can limit job execution, service deployment, streaming and other tasks to run only within a cluster group. You can create a cluster group based on any predicate. For instance, you can create a cluster group from a group of nodes, where all the nodes are responsible for caching data for a cache named **testCache**. It's enough for now, and we will explore this distinction later in the subsequent sections of this chapter.

Ignite nodes can be divided into *two major groups* from the deployment point of view:

1. Embedded with the application.
2. Standalone server node.

## Embedded with the application

Apache Ignite as a Java application can be deployed embedded with other applications. It means that Ignite nodes will be runs on the same JVM that uses the application. Ignite node can be embedded with any Java web application artifact like WAR or EAR running on any application server or with any standalone Java application. Our *HelloIgnite* Java application from *chapter 2* is a perfect example of embedded Ignite server. We start our Ignite server as a part of the Spring application running on the same JVM and joins with other nodes of the grids in this example. In this approach, the life cycle of the Ignite node is tightly bound with the life cycle of the entire application itself. Ignite node will shut down if the application dies or is taken down. This topology is shown in figure 4.3.

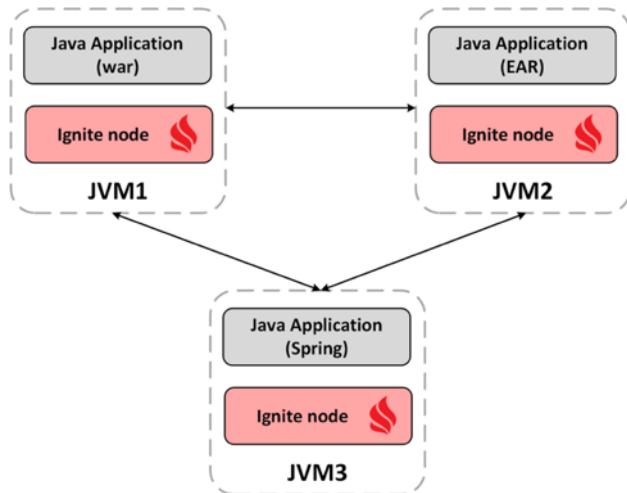


Figure 4.3

If you change the `IgniteConfiguration.setClientMode` property to `false`, and rerun the `HelloIgnite` application, you should see the following:

```
[23:55:22] Ignite node started OK (id=92c8bc87)
[23:55:22] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=3.6GB]
[23:55:22]   ^-- Node [id=92C8BC87-82A1-443E-BF6F-0F5963BECB65, clusterState=ACTIVE]
[23:55:22] Data Regions Configured:
[23:55:22]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
Cache get:Hello Worlds: 1
```

Figure 4.4

`HelloIgnite` Java application run and joins to the cluster as a server node. The application exists from the Ignite grid after inserting a few datasets. Another excellent example of using Ignite node as an embedded mode are implementing web session clustering. In this approach, you usually configure (web.xml file) your web application to start an Ignite node in embedded mode. When multiple application server instances are running, all embedded Ignite nodes connect with each other and forming an Ignite grid. Please see the *chapter 5 Intelligent caching* for more details of using web session clustering.

## Client and the server nodes in the same host

This is one of the typical cases when Ignite client and server nodes are running on different JVM in the same host. You can execute Ignite client and server nodes in separate containers such as Docker or OpenVZ if you are using container technology for running JVM. Both containers can be located in the same single host.

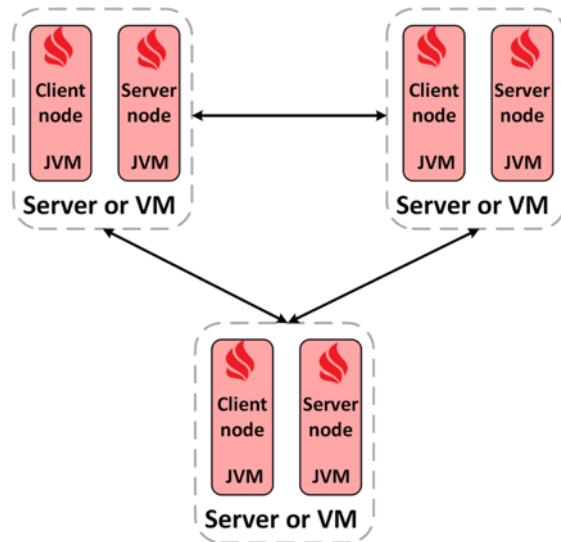


Figure 4.5

The container isolates the resources (CPU, RAM, Network interface) and the JVM only uses isolated resources assigned to this container. Moreover, the Ignite client and server node can be deployed in the separate JVM in the single host without containers, where they all use the shared resources assigned to this host machine. Host machine could be any on-premise, virtual machine or Kubernetes pods.

## Running multiple nodes within single JVM

It is possible to start multiple nodes from within a single JVM. This approach is very popular for unit testing among developers. Ignite nodes running on the same JVM connects with each other and forming an Ignite grid.

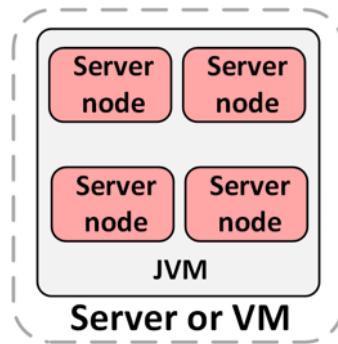


Figure 4.6

One of the *easiest* ways to run a few nodes within a single JVM is by executing the following code::

**Listing 4.3**

---

```
IgniteConfiguration cfg1 = new IgniteConfiguration();
cfg1.setGridName("g1");
Ignite ignite1 = Ignition.start(cfg1);
IgniteConfiguration cfg2 = new IgniteConfiguration();
cfg2.setGridName("g2");
Ignite ignite2 = Ignition.start(cfg2);
```

---

**Tip**

Such a configuration is only intended for developing process and not recommended for production use.

## Real cluster topology

In this approach Ignite client and server nodes are running on different hosts. These are the most common way to deploy a large-scale Ignite cluster for production environment because it provides greater flexibilities in term of cluster technics. Individual Ignite server node can be taken down or restarted without any impact to the overall cluster.

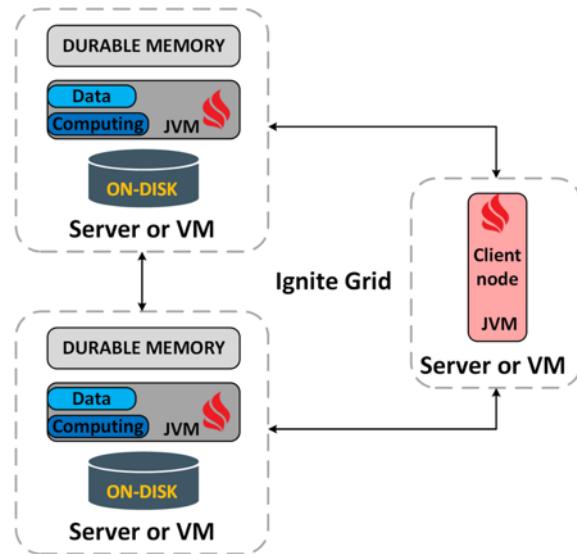


Figure 4.6.1

Such a cluster can be quickly deployed in and maintained by the [kubernetes<sup>30</sup>](#) which an open source system for automating deployment, scaling, and management of the containerized application. [VMWare<sup>31</sup>](#) is another common cluster management system rapidly used for the Ignite cluster.

## Data partitioning in Ignite

[Data partitioning<sup>32</sup>](#) is one of the fundamental parts of any distributed database despite its storage mechanism. Data partitioning and distribution technics are capable of handling large amounts of data across multiple data centers. Also, these technics allow a database system to become highly available because data has been spread across the cluster.

Traditionally, it has been difficult to make a database highly available and scalable, especially the relational database systems that have dominated the last couple of decades. These systems are most often designed to run on a single large machine, making it challenging to scale out to multiple machines.

At the very high level, there are two styles of data distribution models available:

<sup>30</sup><https://kubernetes.io>

<sup>31</sup><https://www.vmware.com/solutions/virtualization.html>

<sup>32</sup>[https://en.wikipedia.org/wiki/Partition\\_\(database\)](https://en.wikipedia.org/wiki/Partition_(database))

1. **Sharding:** it's sometimes called horizontal partitioning. Sharding distributes different data across multiple servers, so each server act as a single source for a subset of data. Shards are called *partitions* in Ignite.
2. **Replication:** replication copies data across multiple servers, so each portion of data can be found in multiple places. Replicating each partition can reduce the chance of a single partition failure and improves the availability of the data.



## Tip

There are also two types of partitions available in partitions strategy: *vertical partitioning* and *functional partition*. A detailed description of these partitioning strategies is out of the scope of this book.

Usually, there are several algorithms uses for distributing data across the cluster, a *hashing algorithm* is one of them. We will cover the Ignite data distribution strategy in this section, which will build a deeper understanding of how Ignite manages data across the cluster.

## Understanding data distribution: DHT

As you read in the previous section, Ignite shards are called partitions. Partitions are memory segments that can contain a large volume of a dataset, depends on the capacity of the RAM of your system. Partition helps you to spread the load over more nodes, which reduces contention and improves performance. You can scale out the Ignite cluster by adding more partitions that run on different server nodes. The next figure shows an overview of the horizontal partitioning or sharding.

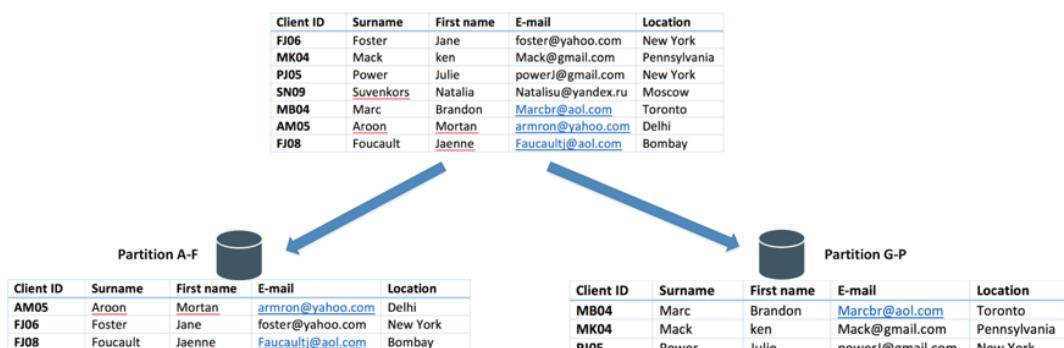


Figure 4.7

In the above example, the client profile's data are divided into partitions based on the client *Id* key. Each partition holds the data for a specified range of partition key, in our case, it's the range of the client ID key. Note that, partitions are shown here for the descriptive purpose. Usually, the partitions are not distributed in any order but are distributed randomly.

**Distributed Hash Table<sup>33</sup>** or DHT is one of the fundamental algorithms used in the distributed scalable system for partitioning data across the cluster. DHT is often used in web caching, P2P system, and distributed database. The first step to understand the DHT is *Hash Tables*. **Hashtable<sup>34</sup>** needs *key*, *value*, and one hash *function*, where hash function maps the key to a location (slot) where the value is located. According to this schema, we apply a hash function to some key attribute of the entity we are storing that becomes the partition number. For instance, if we have four Ignite nodes and 100 clients (assume that client Id is a numeric value), then we can apply the hash function `hash(Client Id) % 4`, which will return the node number where we can store or retrieve the data. Let's begin with some basic details of the Hashtable.

The idea behind the Hashtable is straightforward. For each element we insert, we have to have calculated the slot (technically, each position of the hash table is called slot) number of the element into the array, where we would like to put it. Once we need to retrieve the element from the array, we recalculate its slot again and returns it's as a single operation (something like return array [calculated index or slot]). That's why it has  $O(1)$ <sup>35</sup> time complexity. In short,  $O(1)$  means that the operation takes a certain (constant) amount of times, like 10 nanoseconds or 2 milliseconds. The process of calculating *unique* slot of each element is called *Hashing* and the algorithm how it's done called *Hash function*.

In a typical Hash table design, the Hash function result is divided by the number of array slots and the remainder of the division becomes the slot number of the array. So, the index or slot into the array can be calculated by `hash(o) % n`, where *o* is the object or key, and *n* is the total number of slots into the array. Consider the following illustration below as an example of the hash table.

---

<sup>33</sup>[https://en.wikipedia.org/wiki/Distributed\\_hash\\_table](https://en.wikipedia.org/wiki/Distributed_hash_table)

<sup>34</sup>[https://en.wikipedia.org/wiki/Hash\\_table](https://en.wikipedia.org/wiki/Hash_table)

<sup>35</sup>[https://en.wikipedia.org/wiki/Big\\_O\\_notation](https://en.wikipedia.org/wiki/Big_O_notation)



Figure 4.8

The value on the left represents keys in the preceding diagram, which are being hashed by the hash function for producing the slot where the value is stored. Based on the hash value computed, all the items placed in respective slots. Also, we can look up the client profile of a given client *Id* by calculating its hash and then accessing the resulting slot into the array.



## Info

Implementation of the Hash tables has some memory overhead. Hash tables need a lot of memory to accommodate the entire. Even if most of the table is empty, we need to allocate memory for the entire table. Often, this called a time-space tradeoff, and hashing gives the best performance for searching data at the expense of memory.

Hash table is well suited for storing data set allocated in one machine. However, when you have to accommodate a large number of keys, for instance, millions and millions of keys, DHT comes into play. A DHT is merely a key-value store distributed across many nodes in a cluster. You have to divide the keys into subsets of keys and map those keys to a *bucket*. Each bucket will reside in a separate node. You can assume a bucket as a separate hash table. In one word, using buckets to distribute the key-value pairs is DHT.

Another key objective of the hash function in a DHT is to map a key to the node that owns it, such that a request can be made to the correct node. Therefore, there are *two* hash functions for looking up the value of the key across the cluster in DHT. The first hash function will search for the appropriate bucket maps to the key, and the second hash function will return the slot number of the value for the key located in the node. We can visualize the schema as shown in figure 4.9.

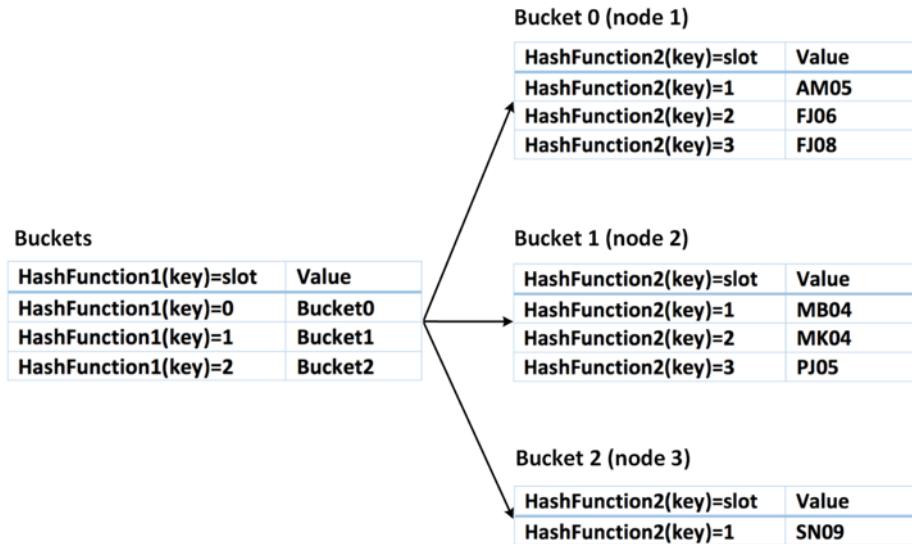


Figure 4.9

To illustrate this, we modified our previous hash table to store pointers to the bucket instead of values. If we have three buckets as shown in the preceding example, then  $key=1$  should go to the *bucket 1*,  $key=2$  will go to *bucket 2* and so on. Therefore, we have to need one more hash function to find out the actual value of the key-value pair inside a particular bucket. *HashFucntion2* is the second hash function for looking up the actual key-value pair from the bucket in this case.

Table named **Buckets** on the left-hand side in figure 4.9 sometimes called *partition table*. This tables stores the partition *IDs* and the node associated to that partition. The function of this table is to make all members of the entire cluster aware of this information, making sure that all members know where the data is.

The fundamental problem of DHT is that it effectively fixes the total number of the nodes in the cluster. Adding a new node or removing nodes from the cluster means changing the hash function which would require redistribution of the data and downtime of the cluster. Let's see what happens when we remove the bucket 2 (node 3) from the cluster, the number of buckets is now equal to two, i.e.,  $n=2$ . This changes the result of the hash function  $\text{hash}(\text{key}) \% n$ , causing the previous mapping to the node (bucket) unstable. The  $key=2$  which was previously mapped to bucket two now mapped to bucket 0 since  $key \% 2$  is equal to 0. We need to move the data between buckets to make it still work, which is going to be expensive in this hashing algorithm.

A workaround for this problem is to use *Consistence Hashing* or *Rendezvous hashing*. Often

Rendezvous hashing is also called *Highest Random Weight (HRW)* hashing. Apache Ignite uses the Rendezvous hashing, which guarantees that only the minimum amount of partitions will be moved to scale out the cluster when topology changes.



## Info

Consistence Hashing is also very popular among other distributive systems such as Cassandra, Hazelcast, etc. At the early stage, Apache Ignite also used consistent Hashing to reduce the number of partitions moving to different nodes. Still, you can find Java class *GridConsistentHash* in the Apache Ignite codebase regards to the implementation of the Consistent Hashing.

## Rendezvous hashing

Rendezvous hashing<sup>36</sup> (aka highest random weight (HRW) hashing) was introduced by David Thaler and Chinya Ravishankar in 1996 at the University of Michigan. It was first used for enabling multicast clients on the internet to identify rendezvous points in a distributed fashion. It was used by Microsoft corporation for distributed cache coordination and routing a few years later. Rendezvous hashing is an alternative to the ring based, consistent hashing. It allows clients to achieve distributed agreement on which node a given key is to be placed in.

The algorithm is based on a similar idea of the [consistent hashing](#)<sup>37</sup> where nodes are converted into *numbers* with hash. The basic idea behind the algorithm is that the algorithm uses *weights* instead of projecting nodes and their replicas on a circle. For each combination of the *node (N)* and *key (K)*, a numeric value is created with a standard hash function  $\text{hash}(N_i, k)$  to find out which node should store a given key. The node that's picked is the one with the highest number. This algorithm is particularly useful in a system with some replication (we will detail the replication mechanism in the next section, for now, data replication is a term means to have redundancies data for high availability) since it can be used to agree on multiple options.

---

<sup>36</sup>[https://en.wikipedia.org/wiki/Rendezvous\\_hashing](https://en.wikipedia.org/wiki/Rendezvous_hashing)

<sup>37</sup>[https://en.wikipedia.org/wiki/Consistent\\_hashing](https://en.wikipedia.org/wiki/Consistent_hashing)

1. Hash (Node id || Key) -> Score
2. Node with the highest score wins

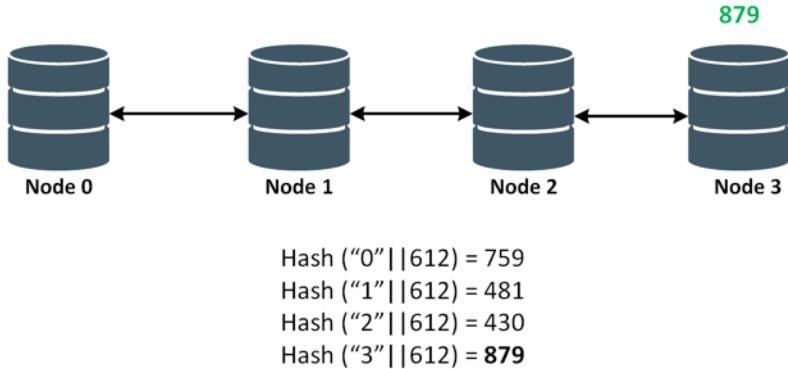


Figure 4.10

Both Consistent hashing and Rendezvous hashing algorithms can be used in a distributed database to decide the home node for any given key, and often can replace each other. However, Rendezvous hashing or HRW have some advantages over Consistent hashing (CH).

1. You do not have to pre-define any tokens for the nodes to create any circle for HRW hashing.
2. The biggest advantage of the HRW hashing is that it provides a very *even distribution* of keys across the cluster, even while nodes are being added or removed. For CH, you have to create a lot of virtual nodes (Vnodes) into each node to provide evenly distribution of keys on a small size of a cluster.
3. HRW hashing doesn't store any additional information for data distribution.
4. Usually, HRW hashing can provide different  $N$  servers for a given key  $K$ . This makes it very useful to support storing redundant data.
5. Finally, HRW hashing is simple to understand and code.

HRW hashing also has a few *disadvantages* as follows:

1. HRW hashing requires more than one hashing computation per key to maps key to a node. It can make a massive difference if you are using some sort of slow hashing function.

2. HRW hashing can be slower to run hash functions against each key node combinations instead of the just once with the CH algorithms.

Rendezvous Hashing or HRW hashing is the default algorithm in Apache Ignite for a key to node mapping since version 2.0. [RendezvousAffinityFunction<sup>38</sup>](#) class is the standard implementation of the Rendezvous Hashing in the Apache Ignite. This class provides affinity information for detecting which node (nodes) are responsible for the particular key in the Ignite grid.



## Info

Keys are not directly mapped to the node in Ignite. A given key always maps to the partition first. Then, the partitions are maps into nodes. Also, Ignite doesn't form any circle like network topology defined in *articles or documentation*.

Mapping of a given key in Ignite is a *three steps* operation. First, any given key will get an affinity key by using *CacheAffinityKeyMapper* function. Affinity key will be used to determine a node on which this key will be cached. The second step will map the affinity key to partition using *AffinityFunction.partition(object)* method. Here, a partition is simply a number from a limited set (0 to 1024), 1024 is default. A key to partition mapping does not change over the time. The third step will map an obtained partition to nodes for the current grid topology version. Partition to node mapping is calculated by using *assignPartitions()* method, which assigns a collection of nodes to each partition.

---

<sup>38</sup><https://github.com/apache/ignite/blob/master/modules/core/src/main/java/org/apache/ignite/cache/affinity/rendezvous/RendezvousAffinityFunction.java>

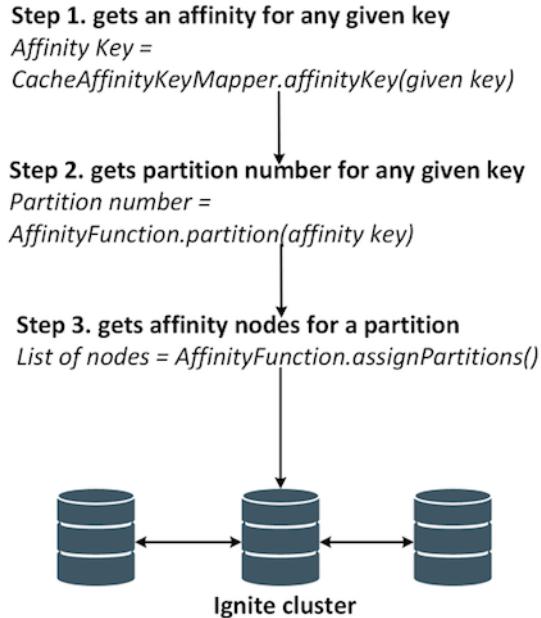


Figure 4.11

Apache Ignite affinity function (key to node mapping) is fully pluggable, and you can implement your version of Rendezvous Hashing or consistent hashing to determine an ideal mapping for the partition to nodes in the grids. You must have implemented the Java interface `AffinityFuction` and configure this function in the cache configuration as shown below:

Listing 4.4

---

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    <property name="cacheConfiguration">
        <list>
            <!-- Creating a cache configuration. -->
            <bean class="org.apache.ignite.configuration.CacheConfiguration">
                <property name="name" value="myCache"/>
                <!-- Creating the affinity function with custom setting. -->
                <property name="affinity">
                    <bean class="org.apache.ignite.cache.affinity.rendezvous.RendezvousAffinityFunction">
                        <property name="excludeNeighbors" value="true"/>
                        <property name="partitions" value="1048"/>
                    </bean>
                </property>
            </bean>
        </list>
    </property>
</bean>

```

```
</list>
</property>
</bean>
```

---

Note that, Apache Ignite *RendezvousAffinityFunction* class is also implemented the *AffinityFunction* interface. Ignite implementation of the Rendezvous hashing algorithm has a few advantages over other typical implementations of such algorithm:

1. Cache key affinity to nodes survives for full cluster restarts. That means, all keys should still map to the same nodes if you will restart the whole cluster and reload the data from the disk.
2. Ignite *RendezvousAffinityFunction* is pretty faster because partition to node mapping stage is almost constant and only calculates whenever cluster topology changes (add or remove nodes from the cluster).

The disadvantages of using *RendezvousAffinityFunction* is that it does not provide data distribution *evenly*. Some nodes in the cluster may have a slightly large number of partitions than other nodes. The inaccuracy of data distribution would be approximately 5-10%.

Note that, up to version 1.9, there was another hashing implementation called *FairAffinityFunction* in Ignite, which provided data distribution evenly. However, FairAffinityFunction did not perform well because it exchanged a high volume of data between nodes and provided inconsistent mapping for caches created on different topology versions. Also, data collocation was not working correctly in some scenario. So, Apache Ignite community decided to remove the FairAffinityFunction since 2.0 version.

Rendezvous affinity functions support a few configurations, such as:

1. *partitions* - a number of the partitions to spread across the grid, the default is 1024.
2. *excludeNeighbors* – it ensures that primary and backup partitions are not located on the same node. It will exclude the same host as a neighbor for storing primary and backup copies when it set true.
3. *backupFilter* – optional filter for backup nodes. The node that only passed this filter will be selected for backup copies if provided. It is handy in cases when you would like to store primary and backup copies of partitions on different racks.

## Replication

In this section you will learn one of the critical features of the distributed data store: *reliability* or *redundancy* of the data to make any sort of availability guarantee in the case of the node failure. Data replication or redundancy of the data enables high availability and reliability in a distributed system. Data is replicated in more nodes to give you fail-over safety in this approach. The redundant copies of the data will be used to continue any operations in case of node failure. In a distributed database world data replication comes into two forms:

1. Master-slave;
2. and Peer-to-peer.

We will now discuss these techniques starting at the simplest: first master-slave replication, then peer-to-peer replication and finally how Apache Ignite manages the replication strategy.

## Master-Slave replication

One of the nodes in the cluster is assigned as a master or primary node with master-slave replication. This master node is the authoritative source of the data and responsible for processing any update for the data. The rest of the nodes consider as a secondary or slave nodes.

Most of the time, all DML operations such as create/update/delete are made to the master node, and then replication process propagates these updates to the slave nodes for synchronization. Clients can read data from a master or slave nodes. Usually, a leader election process appointed a slave node as a new master after a failure of the master node. The new master node immediately starts accepting DML operations from the clients.

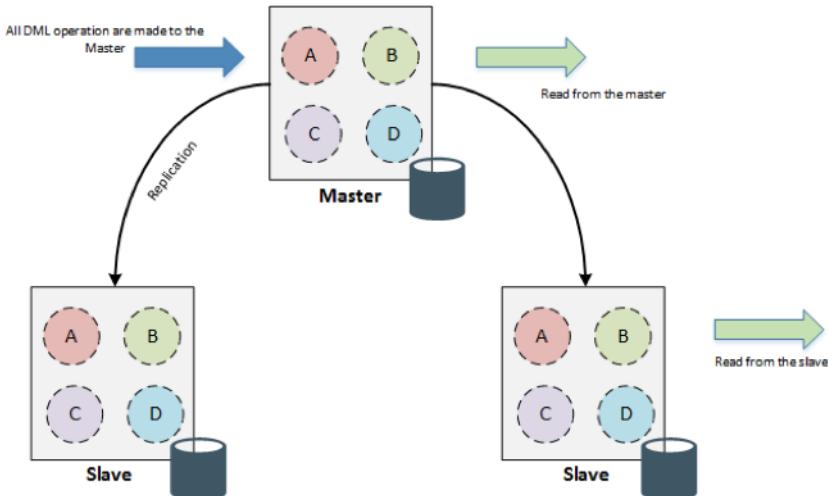


Figure 4.12

*MongoDB* is one of the popular NoSQL databases uses the master-slave replication for distributing data across the cluster. Apache Ignite does not have any master node and therefore **does not use master-slave replication** for distributing data. Anyway, there are a few pros and cons of master-slave replication:

#### Advantages:

1. Very suitable for read-intensive data set.
2. Slaves can handle the read requests in case of a failure of the master node.

#### Disadvantages:

1. The master is the single point of failure, and a bottleneck.
2. Not suitable for the heavy write-intensive data set.
3. Data may be inconsistency due to slow propagation to the slaves.

We are not going to details all the pitfalls of these replication techniques; however, I encourage you to check the MongoDB documentation if you are curious about the master-slave replication at details. In the next section we are going to discuss the peer-to-peer replication techniques used by Apache Ignite.

## Peer-to-peer replication

One of the most widespread replication approach among NoSQL databases. A distributed database like Apache Ignite, Cassandra, Riak, and HBase gracefully use this replication strategy in their architecture. The idea behind the peer-to-peer replication is straightforward, all nodes have equal weight and can receive any update. You can write the record into any node, and the record will be routed to the node responsible for storing that data when you want to update any datasets. The loss of any node does not prevent access to the data store.

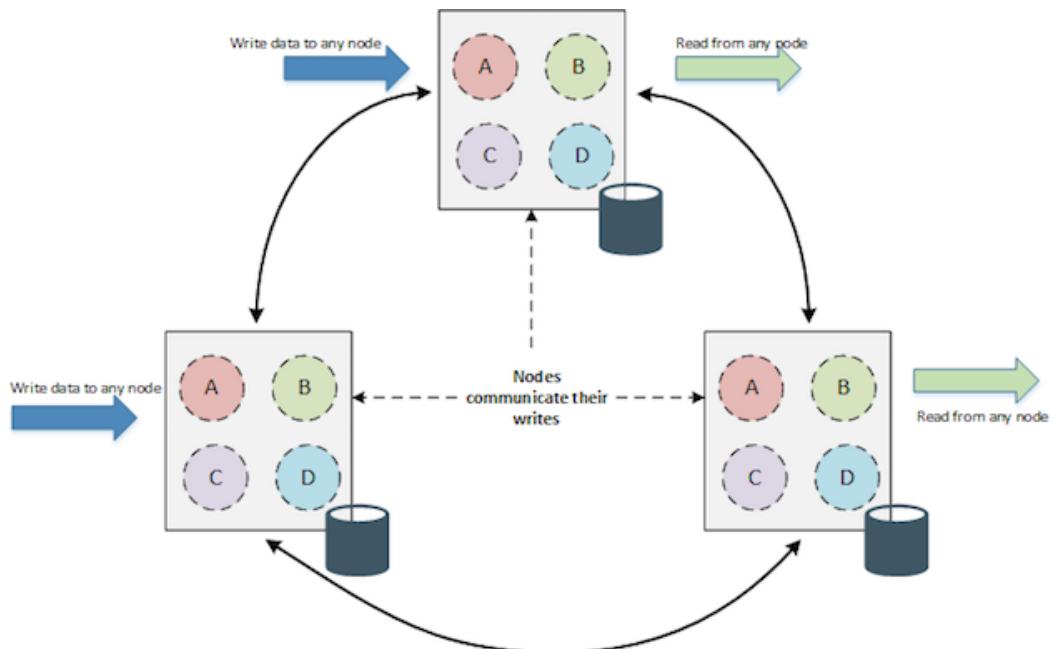


Figure 4.13

Although, there are a few advantages and disadvantages of using peer-to-peer replication and they are as follows.

### Advantages:

1. There is no single point of failure.
2. You can quickly add or remove nodes from the cluster.

### Disadvantages:

1. Data might be inconsistent due to slow propagation of changes to copies on different nodes.
2. Potential possibility of write-write conflict, when two users are trying to update different copies of the *same record* stored on different nodes at the same time.

Apache Ignite uses the combination of peer-to-peer replication and sharding strategies, unlike other NoSQL databases. In a scenario like this, you might have tens of hundreds of nodes in a cluster with data shared and replicated over them. For some caches (for now, think a cache is a container for storing key-value pair very similar at RDBMS table with two columns) you can use the **PARTITION** mode, and for others, you can apply **REPLICATED** mode. In PARTITION mode, nodes to which the *keys* are assigned to are called *primary node*. Optionally, you can also configure any number of *backup copies* for cached data. Ignite will automatically assign *backup nodes* for *each key* if the number of **backup copies** is higher than **0**. Ignite will copy the cache data into *each node* of the cluster in case of **REPLICATED** mode.

## Partitioned mode

The goal of this cache mode is to get extreme *scalability*. In this mode, Ignite transparently partitions the cached data for distributing the load across an entire cluster. The size of the cache and the processing power grows linearly with the size of the cluster by partitioning the data evenly. The responsibility for managing the data is automatically shared across the cluster. Every node or server in the cluster contains its primary data with a backup copy if *defined*.

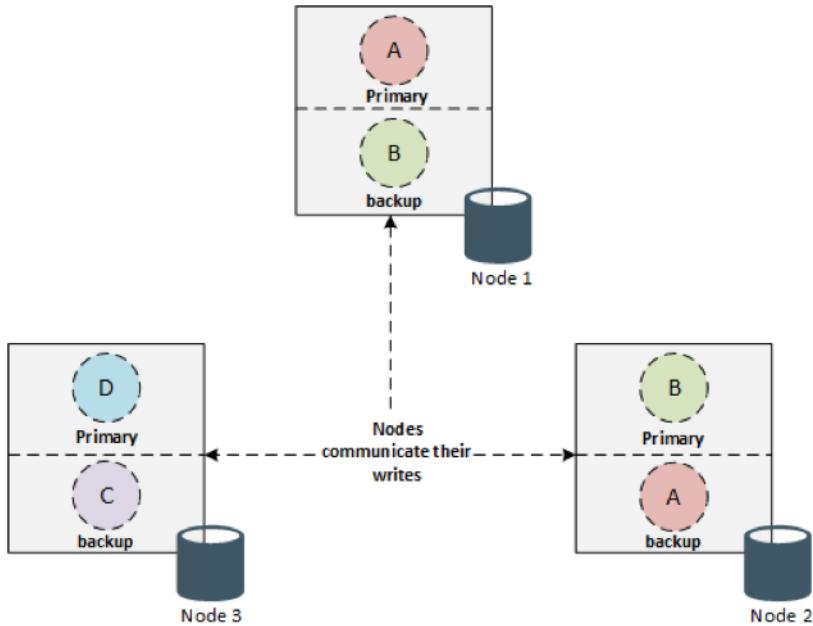


Figure 4.14

DML (create/update/delete) operations on the cache are extremely fast with partitioned cache topology because only one primary copy of the data needs to be updated for every key. A backup copy of the cache entry should be configured for high availability. As we mentioned before, the backup copy is the redundant copy of one or more primary copies, which lives in another node. There is a simple formula to calculate how many backup copies you need for the high availability of your cluster.

The number of backup copies =  $N-1$ , where  $N$  is the total number of the nodes in the cluster. Assuming, you have a total number of three nodes in the cluster. The number of backup copies should be not less than two if you always want to get a response from your cluster (when some of your nodes are unavailable). In this case, three copies of the cache entry exist, two backup copies and one primary.



## Warning

Term *Partitioned mode* or *Partitioned cached* are same and used interchangeably throughout this book.

A partitioned mode is ideal when working with large datasets and updates are very frequent. To get better query performance, it is important to always access the data from the node

that has that data cached. Sometimes this approach is also called affinity colocation and highly recommended when working with partitioned caches. Backup copies of the data can be configured by settings `backup()` property of `CacheConfiguration` as shown below:

Listing 4.5

---

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="cacheName" />
            <!-- Set cache mode. -->
            <property name="cacheMode" value="PARTITIONED" />
            <!-- Number of backup nodes. -->
            <property name="backups" value="1" />
        </bean>
    </property>
</bean>
```

---

The data replication process can be *synchronous* or *asynchronous*. The client should wait for the responses from the remote nodes, before completing the commit or write in synchronous mode. This mode is called *FULL\_SYNC*. This mode can increase the transaction time if you have a large number of backup copies configured. There is another variance of synchronous mode available: *PRIMARY\_SYNC*. Client node will wait for completing the commit on the primary copy of the data but will not wait for backups to be committed in this mode. *PRIMARY\_SYNC* is the default synchronization mode for the cache configuration and highly recommended for using in the high-load system. Synchronous mode is preferable when you would like to have a very high consistency data throughout the entire cluster.

In asynchronous mode, client node does not wait for responses from participating nodes, in which case remote nodes may get their state updated slightly after any of the caches write methods complete or after completing the `Transaction.commit()` method.



## Info

Ignite also provides API for reading backup copies from the nodes. Moreover, you can allocate backup copies in different racks to provide high availability.

## Replicated mode

The goal of this approach is to get extreme *read* performance. In this mode, cache data is replicated to every node of the cluster. Since the data is replicated to each node, it is available for use without any waiting. This provides the highest possible speed for read-access; a client can read data from any of the nodes of the cluster. The downside is that frequent writes are very expensive. Updating a replicated cache requires pushing the new version to all other cluster members. This will limit the scalability if there are a high frequency of updates.

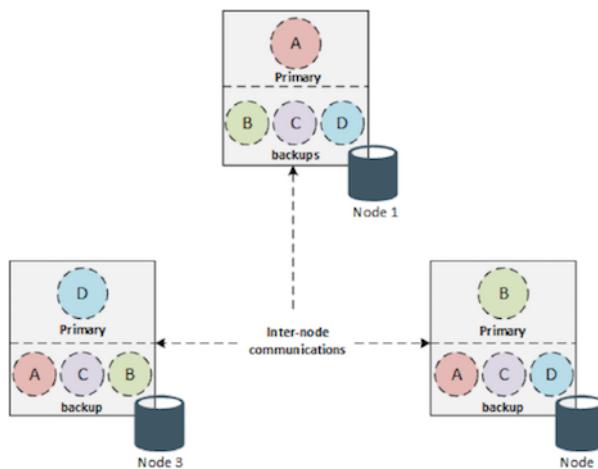


Figure 4.15

Replicated mode uses when the data sets are small, and updates are not frequent. Replicated caches are implemented in the way similar to partitioned caches where every key has a primary copy and is also backed up on all other nodes in the cluster in Ignite. For instance, in figure 4.15, the node running in JVM1 is a primary node for key A, but it also stores backup copies for all other keys as well (B, C, D).



## Warning

Up to 2.5 version, all read requests from the client node to replicated cache goes through the node that stores primary copy of the data. It was fixed as a bug<sup>39</sup> and resolved in version 2.5. If you are using Ignite version older than 2.5, you should update your cluster or patch the bug fix.

<sup>39</sup><https://issues.apache.org/jira/browse/IGNITE-5357>

## Local mode

This is a very primitive version of cache mode located in the Ignite server node. No data is distributed to other nodes in the cluster and does not have any replica or backup copies of the cache with this approach. As far as the cache with the Local mode does not have any replication or partitioning process, data fetching is very inexpensive and fast. It provides zero latency access to recently and frequently used data. The local cache mode is mostly used in read-only operations. It also works very well for read-through behavior, where data is loaded from the data sources on cache misses. Local cache mode still has all the features of a distributed cache and provides query caching, automatic data eviction and much more unlike a distributed cache.

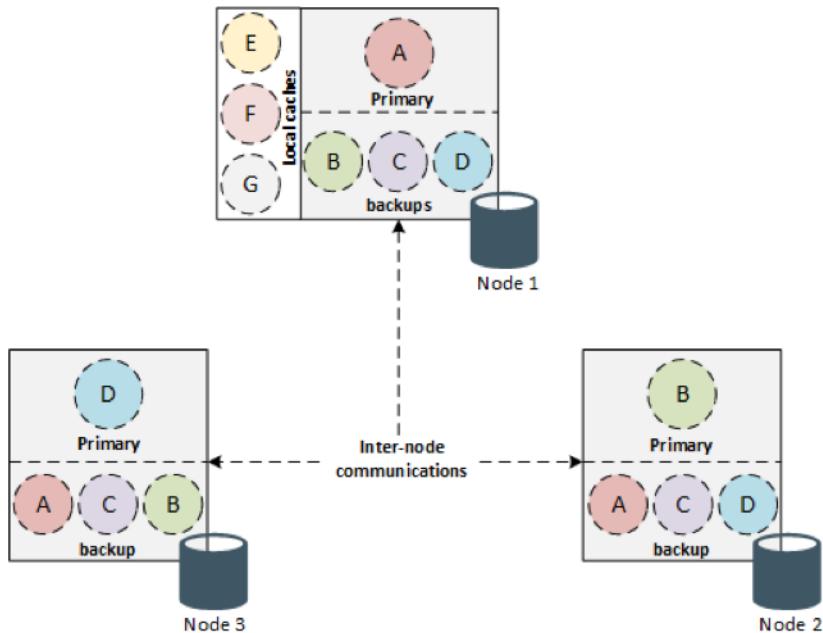


Figure 4.16

## Near cache

It's very similar to a local cache mode with one difference. Near cache can be located on *client nodes*. Near cache stores the most recently or frequently updated partitioned data and usually sited in front of the partitioned cache. It can improve query performance by caching frequently used data entries on the client node locally. Near cache allows not to go on remote

server node for querying data.

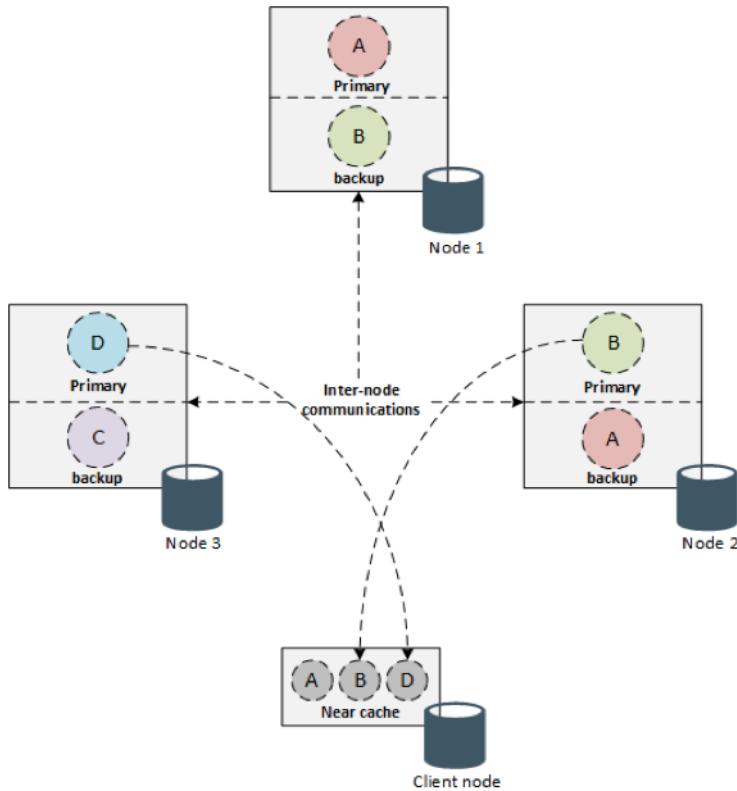


Figure 4.17

Also, near cache has no sense for server node on *REPLICATED* cache, because all the necessary (frequently accessed) dataset already available for each client node locally. So, if your remote server nodes located on the separate machine and you want to get a performance boost, you should configure your client node to use near cache.



## Warning

Near cache can only be used with *PARTITIONED* cache

You can control the size of the near cache and its eviction policy just like a *PARTITIONED* cache. Near cache can also participate in computation (see the chapter 7 for more details). Near cache are fully transactional and get updated or invalidated automatically whenever

the cache changes on the server node. There is a various way to configure near cache on client nodes. You can directly create near cache on client node by passing the property *NearConfiguration* into Ignite configuration or create both PARTITIONED cache and near cache dynamically on node startup.

## Partition loss policies

Data loss is not a rare case in distributed database system. It may occur if a few members of the cluster crash simultaneously that held a copy of the partitions. For instance, let's consider a situation with four nodes: N1, N2, N3 and N4 which contains a partition named *part-0*. N1 is the owner of the partition and held the primary copy of the part-0, and N2-4 are the backup copies respectively. Now, assume that, we have configured the backup copies *less than three* backups (1 primary and 1 backup copies of the data). In such a situation, if node N1 and N2 crash at the same time (simultaneously), part-0 loses its data, because both primary node and the backup of the part-0 have crashed. However, if we configure the backup number with *three* backups, it does not lose any data since a backup copy of part-0 data for the given partition also reside in N3 or N4.

Node failures can result from various causes such as hardware failures (for instance, without RAID to handle disk failure) and network outages. Apache Ignite provides a few *PartitionLossPolicies* and event handling that can be used for preventing minimum data loss according to your use case. Partition loss policy defines, how a cache will behave in a case when one or more partition is lost because of the failure of a node. For instance, some application treats this as an urgent issue, blocking all write operations that go to the lost partitions. While others might ignore this event at all because the lost data can be repopulated over time or partial data loss is not critical for their systems. Apache Ignite supports the following policies:

Policy	Description
READ_ONLY_SAFE	All writes to the cache will be failed with an exception, reads will only be allowed for keys in non-lost partitions. Reads from lost partitions will be failed with an exception.
READ_ONLY_ALL	All writes to the cache will be failed with an exception. All reads will proceed as if all partitions were in a consistent state. The result of reading from a lost partition is undefined and may be different on different nodes in the cluster.
READ_WRITE_SAFE	All reads and writes will be allowed for keys in valid partitions. All reads and writes for keys in lost partitions will be failed with an exception.
READ_WRITE_ALL	All reads and writes will proceed as if all partitions were in a consistent state. The result of reading from a lost partition is undefined and may be different on different nodes in the cluster.

Policy	Description
IGNORE	If partition is lost, reset it's state and do not clear intermediate data. The result of reading from a previously lost and not cleared partition is undefined and may be different on different nodes in the cluster. This mode is used by <b>default</b> .

The partition loss policy can be configured per cache level as follows:

**Listing 4.6**

```
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    <property name="name" value="LostPartitions"/>
    <property name="cacheMode" value="PARTITIONED"/>
    <property name="backups" value="1"/>
    <property name="partitionLossPolicy" value="READ_ONLY_ALL"/>
</bean>
```

Apache Ignite also provides several facilities for handling partition lost events. You can subscribe to the `EVT_CACHE_REBALANCE_PART_DATA_LOST` event for getting notified whenever any partition is lost. Moreover, you can use `IgniteCache.lostPartitions()` function to get information of all the lost partitions at some point in time.

Ignite Visor CLI supports two more new commands from the Apache Ignite 2.5 version: `cache -slp` and `cache -rlp` for displaying and reset lost partition for a cache. Apache Ignite Web console also delivers these functionalities through monitoring that shows the partition loss metrics.

## Caching strategy

As stated earlier, caching is a widely used technics that favors scalability, performance, and affordability against database bottleneck. Usually, relational databases cannot scale out perfectly by adding more servers. In this circumstance, in-memory distributed cache offers an excellent solution to data storage bottleneck. If done *right*, caches can reduce the response times, decrease the load of the database, save cost and money. Caches can be extended on multiple servers to pool their memory together and keep the cache synchronized across all servers.

There are several caching strategies available in the in-memory caching world and choosing the right one can make a big difference. The approach you want to implement for maintaining your cache depends upon the type of the data you are going to cache and the access pattern to that data. In other words, how the data is written and read. For instance,

- Write heavy and reads frequently (e.g. online transaction).
- Write less frequently and read heavy (e.g. user profile).
- Randomly access or data returns always unique (e.g. search queries).

A caching strategy for a retail bank portal should be different from the caching strategy for an online gaming website. Choosing the right caching strategy is the key design issue for improving the performance of your application. We discuss some common caching strategies, their advantages, and disadvantages and how Apache Ignite maintains each of them in the remaining part of this section.

## Cache a-side

Probably the most commonly used caching strategy. In this approach, an application is responsible for reading and writing from the persistence store. The cache sits on the side, and the application directly talks to both the cache and the data store.

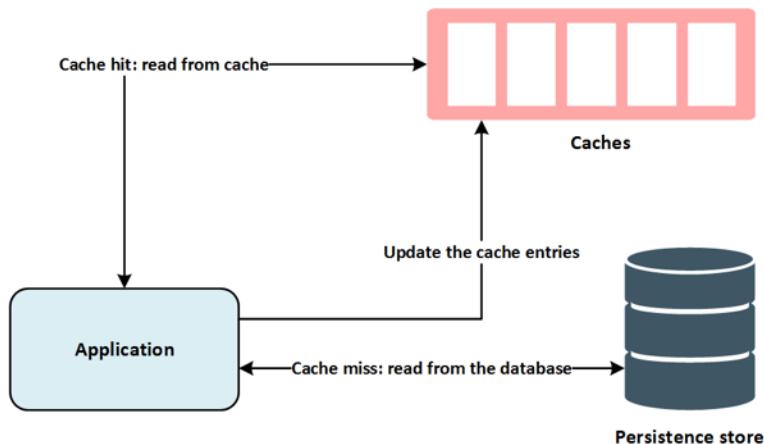


Figure 4.18

Here how it happens:

1. First, the application checks the cache by making a query to the cache.
2. If the cache data is found in the cache and isn't expired. Cache returns the data to the application. It's called *Cache hit*.
3. If the data is not in the cache or it's expired. Application request and retrieve the data from the database. It's called *Cache miss*.

4. Application updates the data into the cache so that subsequent reads for the same data results becomes a cache hit.

Cache-aside approach is usually general purpose and suitable for read-heavy workloads.

#### Advantages:

1. Cache-aside cache is resilient to cache failure. The application continues work by querying the database in case of cache cluster failure. However, it does not help much if cache server goes down during peak hour; response time can increase, and a result database can stop responding.
2. The data model in a cache can be different than the data model in the database.

#### Disadvantages:

1. Application code can become complicated and may lead to code duplication if multiple applications deal with the same data store.
2. When there are cache data misses, the application queries the data store, update the caches and continue processing. This can result in multiple datastore visits if different application threads perform this processing simultaneously.

Apache Ignite does not provide any transparent configuration to support *cache-aside* cache. You have the full responsibilities for implementing the logic and maintenance of the cache from the application developer perspective.

## Read through and write through

In this approach, application treats the cache as the primary data store, reads data from it and writes data to it. Read/Write-through cache sits between your application and the data store (database). In read-through strategy, when there is a cache miss, it retrieves the data from the database, updates the cache and returns the result to the client. The write-through strategy adds or updates the data in the cache whenever data is written to the database.

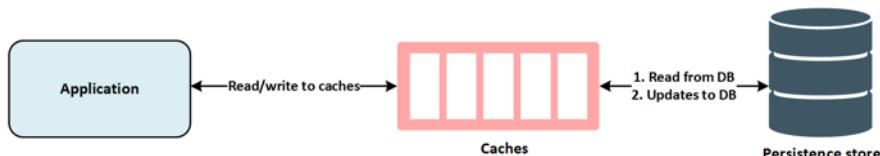


Figure 4.19

Both read/write-through strategy is convenient for read-heavy or write-heavy workloads.

#### Advantages:

1. Firstly, it simplifies the application logic. The logic for fetching data from the database and updating the cache is usually supported by the library or by the cache provider. Hibernate 2<sup>nd</sup> level cache provider or MyBatis cache provider is an example of such library.
2. Read-through cache allows the caches to reload data from the database whenever it expires automatically.
3. All read/write-through operations participate in the overall cache transaction and commit or rolled back as a whole.

#### Disadvantages:

1. Unlike Cache-aside, the data model in read/write-through cache cannot be different than that database.
2. Read/Write-through cache is not resilient to cache cluster failure. The application may respond with errors in case of cache cluster failure.

Apache Ignite data grid feature fully supports the read/write-through cache functionality. Ignite Hibernate/MyBatis L2 cache implementations and Ignite native data loading feature allows you to use both read-through and write-through caches for speed-up the persistence layer of your application. In *chapter 5*, we will look at the database catching in detail.

## Write behind

This is a more advanced variation of the write-through caching strategy. Write-behind strategy lets your application quickly update the cache and return. It aggregates the updates and asynchronously flushes them to the persistence store as a bulk operation unlike write-through caching strategy.

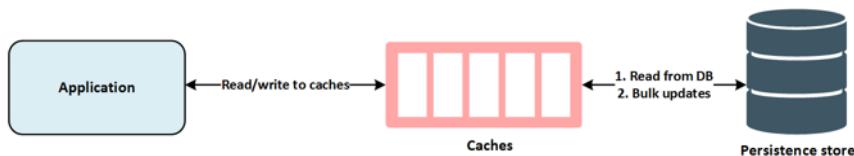


Figure 4.20

You can specify a *throttling* limit with Write-behind strategy, so the database writes are not performed as fast as the cache updates, and therefore the pressure on the database is lower. Also, you can schedule the database writes to occur during off-peak hours, which can also minimize the load on the database.

Apache Ignite provides these caching strategies by implementing the Java [JCache specification<sup>40</sup>](#). Also, Apache Ignite provides 3<sup>rd</sup> party persistence features which can be used as caching layer above an existing RDBMS and NoSQL such as Cassandra or MongoDB. We will cover the Ignite persistence feature more briefly in *chapter 6*.

## Ignite data model

Apache Ignite implements a Key-Value data model, especially JCache (JSR 107) specification. JCache provides a standard way for Java application to interact with the Cache. The JCache final specification was released on 18th March 2014, where Terracotta has played the leading role and acting as a specification lead. On September that year, Spring 4.1 was released with an implementation of the JSR107.

Why did we need another specification of Java? Because open source caching projects and commercial vendors like Terracotta and Oracle have been out there over a decade. Each project and vendor use a very similar hash table like API for primary storage. With the JSR107 specification, at last developers can write an application with a standard API instead of being tied to a specified vendor.

From a design point of view, JCache is a simple key-value store. You can imagine a cache as a simple table in a traditional relational database with two columns: *key* and *value*. The data type of the *value column* can be any primitive data type such as String, Integer or any complex Java object (or Blob – in Oracle Terms). Any application can provide a Key-value pair and persist these pair into any persistence store. The value will be overwritten if a Key already exists; otherwise, a new value will be created. We can compare the key-value store with Oracle terminology for simplicity.

Oracle	Apache Ignite
Database instance	Apache Ignite server node
Table	Cache
Row	Value
RowID	Key

<sup>40</sup><https://jcp.org/aboutJava/communityprocess/final/jsr107/index.html>

**Key-value stores<sup>41</sup>** are the simplest data store in the NoSQL world. It supports very primitive operations like put, get, or delete a value from the store. Generally, they have a high performance and scalability since it always uses primary key access.

A Key-value pair can be easily illustrated in a diagram. Consider the following illustration of an entry in a Cache.

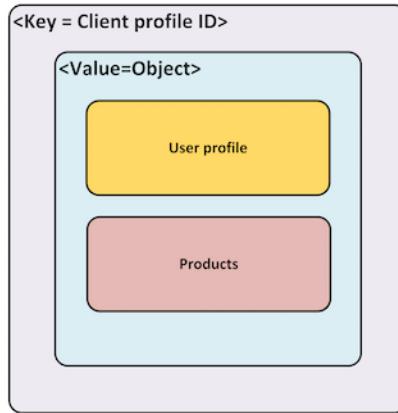


Figure 4.21

With a key-value store, JCache API provides the following features:

1. Basic Cache operations.
2. Atomic operations, similar to `java.util.ConcurrentMap`.
3. Read/Write-through caching.
4. Entry processor.
5. Cache event listeners.
6. Statistics.
7. Caching annotations.
8. Full generics API for compile time safety.
9. Storage by reference (applicable to on heap caches only) and storage by value.

In addition to JCache API, Apache Ignite provides *ACID transaction*, *SQL query* capability, data loading, *asynchronous executions* and various memory models. Apache Ignite provides the **IgniteCache<sup>42</sup>** interface, which extends the Java Cache interface for working with the Cache. We already saw some basic operations of IgniteCache in *chapter 2*. Here is the pseudo code of the *HelloWorld* application from the chapter 2.

<sup>41</sup>[https://en.wikipedia.org/wiki/Key-value\\_database](https://en.wikipedia.org/wiki/Key-value_database)

<sup>42</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/IgniteCache.html>

**Listing 4.7**

---

```
IgniteCache<Integer, String> cache = ignite.getOrCreateCache("testCache");
// put some cache elements
for(int i = 1; i <= 100; i++){
    cache.put(i, Integer.toString(i));
}
// get them from the cache and write to the console
for(int i =1; i<= 100; i++){
    System.out.println("Cache get:"+ cache.get(i));
}
```

---

Apache Ignite also implements the JCache *entry processor* functionality for eliminating the network round trips across the network when doing puts and updates into the cache. [JCache EntryProcessor<sup>43</sup>](#) allows processing of data directly on primary nodes, often transferring only the deltas instead of the full changed state.

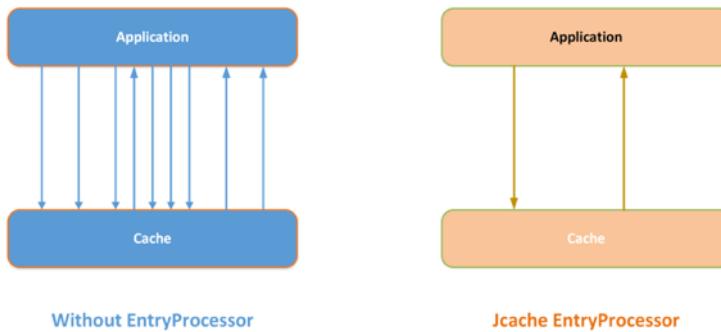


Figure 4.22

Moreover, you can add your own business logic into an *EntryProcessors*, for instance, taking the previously cached value and incrementing it by 1.

---

<sup>43</sup><https://static.javadoc.io/javax.cache/cache-api/1.0.0/javax/cache/processor/EntryProcessor.html>

**Listing 4.8**

---

```
IgniteCache<String, Integer> cache = ignite.jcache("mycache");
// Increment cache value 10 times.
for (int i = 0; i < 10; i++) {
    cache.invoke("mykey", new EntryProcessor<String, Integer, Void>() {
        @Override
        public Object process(MutableEntry<Integer, String> entry, Object... args) {
            Integer val = entry.getValue();
            entry.setValue(val == null ? 1 : val + 1);
            return null;
        }
    });
}
```

---

The key-value data model has some weakness as well while it gives a few flexibilities. One is that the model not provides any traditional database capabilities such as integrity constraint through foreign key or consistency when multiple transactions are executed simultaneously. The application itself must provide such capabilities. However, in addition to traditional key-value storage model, Apache Ignite provides a few additional opportunities. Some of its key characteristics are:

1. Indexing on data. You are not restricted on using the *key* for querying distributed data. You can index any part of the *value* as an index and use it for querying data.
2. Provides strong data affinity and co-location semantics by default.
3. Provides a data model for individual updates so updates can be accessed, locked and updated very effectively.
4. Provides SQL queries over key-value data.

However, if your data model is closer to the relational data model or you are willing to use a lot of complex SQL query such as ad-hoc queries, perhaps key-value data model is not the right choice for you.

## CAP theorem and where does Ignite stand in?

When I first started working with Apache Ignite, I wondered how on the one hand Ignite supports ACID transactions, and on the other hand, Ignite is also a highly available distributed system. Supporting ACID transactions and at the same time providing high availability is a challenging feature for any NoSQL databases. To scale horizontally, you

need strong network partition tolerance which requires giving up either consistency or availability. NoSQL system typically accomplishes this by relaxing relational availability or transactional semantics. A lot of popular NoSQL data stores like Cassandra and Riak still do not have transaction support and are classified as an AP system. The word AP comes from the famous [CAP theorem<sup>44</sup>](#) and it means availability and partition tolerance, which are generally considered more important in NoSQL systems than consistency.

In 2000, [Eric Brewer<sup>45</sup>](#) in his keynote speech at the ACM Symposium said that one could not guarantee consistency in a distributed system. This was his conjecture based on his experience with the distributed systems. This conjecture was later formally proved by Nancy Lynch and Seth Gilbert in 2002. Now, nearly each NoSQL databases can be classified by the CAP theorem as shown in the following figure.

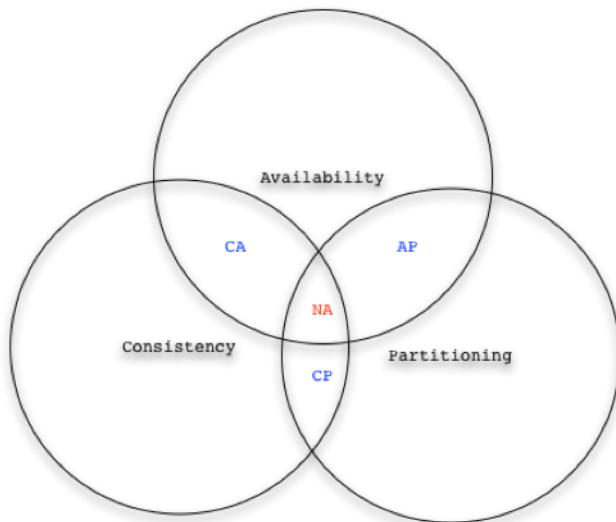


Figure 4.23

As you can see, a distributed system can only have two of the following three properties:

1. **Partition tolerance:** means that the system still works if you chop the cable between two nodes.
2. **Consistency:** each node in the cluster has the same data.
3. **Availability:** a node will always answer the queries if possible.

<sup>44</sup>[https://en.wikipedia.org/wiki/CAP\\_theorem](https://en.wikipedia.org/wiki/CAP_theorem)

<sup>45</sup>[https://en.wikipedia.org/wiki/Eric\\_Brewer\\_\(scientist\)](https://en.wikipedia.org/wiki/Eric_Brewer_(scientist))

So, let us see how choosing two out of three options affects the system behavior as follows.

### CA system.

In this approach, you sacrifice partition tolerance for getting consistency and availability. Your database system offers transactions, and the system is highly available. Most of the relational databases are classified as CA systems. This system has serious problems with scaling.

### CP system.

This is the opposite of the CA system. Availability is sacrificed for consistency and partition-tolerance in CP system. Some data will be lost in the event of the node failure.

### AP system.

This system is always available and partitioned. Also, this system scales easily by adding nodes to the cluster.



## Info

Real-World systems rarely fall neatly into all of these above categories, so it's more helpful to view CAP as a *continuum*. Most systems will make some effort to be consistent, available, and partition tolerant, and many can be tuned depending on what's most important.

Now, we can return to our question, where does Apache Ignite stand in the CAP theorem? At a glance, Ignite can be classified by CP, because Ignite is entirely ACID compliant distributed transactions with partitioned tolerance. However, this is half part of the history. Apache Ignite can also be considered an AP system because Ignite does not have any single point of failure and can be partitioned. But why does Ignite have two different classifications? Because, it has two different *transaction modes* for cache operations: **transactional** and **atomic**.

In *transactional* mode, you can group multiple DML operations in one transaction and make a single commit into the cache. In this scenario, Ignite will lock data on access by a pessimistic lock. We will look at transactions in details in chapter 6.

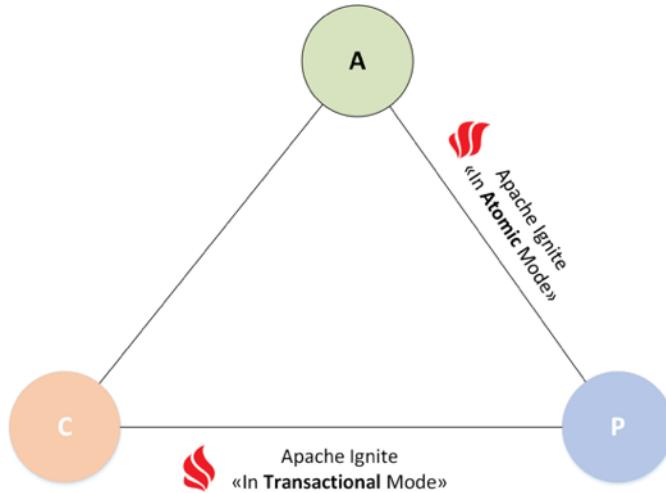


Figure 4.24

On the other hand, Ignite supports multiple atomic operations sequentially in *atomic* mode. In an atomic mode, each DML operation will either succeed or fail and, neither Read nor Write operation will lock the data at all. This mode gives better performance than the transactional mode. When you make a write in the Ignite cache, for every piece of data there might be a master copy and a backup copy (if defined). When you read data from the Ignite grid, you always read the master copy of the data, unless the node that stores the master copy is down (however, you can read backup copy explicitly through API), at which time data will be read from the other nodes. From this point of view, you gain the system availability and the partition-tolerance of the entire system as an AP system. Apache Ignite is very close to Apache Cassandra in an Atomic mode.

## Apache Ignite life cycle

As we have mentioned before, Apache Ignite node lives and runs on JVM, so Ignite's life cycle strictly depends on the JVM life cycle. The life cycle begins as soon as you call one of the following two methods:

### **Listing 4.9**

---

```
Ignite ignite = Ignition.start();
```

---

or from a configuration file as shown below:

**Listing 4.10**

```
ignite ignite = Ignition.start("config/example-cache.xml");
```

Also, you can use *ignitevisorcmd.sh* tool to stop an Ignite node: apply the `open` command to connect to the Ignite grid and then use the `kill` command to stop/restart an Ignite node.

You can also monitor and react to an event's in an Ignite lifecycle by defining *LifeCycleBean* objects whose methods get invoked when lifecycle event occurs. For instance, you can perform some action before or after Ignite node starts or stops. Ignite provides four different types of lifecycle events:

1. BEFORE\_NODE\_START. Invoked before node startup routine is initialized. In this event, Ignite node is not available yet.
2. AFTER\_NODE\_START. Fires just after the Ignite node start. This event can be used to load data into the cache from any 3<sup>rd</sup> party database or sources.
3. BEFORE\_NODE\_STOP. Before initiating the node stopped. Node is still available at this stage, and you can initialize any cleanup process such as clean up off-heap memory before node stop.
4. AFTER\_NODE\_STOP. After the Ignite node stops. Node is not available during this stage, and any Ignite node related code will not execute.

Lifecycle bean can be configured in several ways. One of the easiest ways to configure the bean is to implement the *LifeCycleBean* interface as shown below:

**Listing 4.11**

```
public class CustomLifecycleBean implements LifecycleBean {  
    @Override  
    public void onLifecycleEvent(LifecycleEventType lifecycleEventType)  
        throws IgniteException {  
  
        if(lifecycleEventType == LifecycleEventType.AFTER_NODE_START) {  
            }  
    }  
}
```

Here, we use the lifecycle event types to perform an action after the node starts. Next, we can pass the *CustomLifecycleBean* implementation to the *IgniteConfiguration* during the node startup as shown in listing 4.12.

**Listing 4.12**


---

```
IgniteConfiguration configuration = new IgniteConfiguration();
configuration.setLifecycleBeans(new CustomLifecycleBean());
Ignite ignite = Ignition.start(configuration);
```

---

Beside that, an Ignite *Lifecycle* beans can be injected using [dependency injection<sup>46</sup>](#) with Ignite resources. Ignite allows both *field* and *method-based* injection. You can inject or pass resources by either annotating a field or a method. There are a number of pre-defined Ignite resources that can be injected as a resource. We describe the most commonly used Ignite resources that can be useful for using with Ignite lifecycle bean in the following table.

Resources	Description
IgniteInstanceResource	Injects of a current Ignite instance.
SpringResource	Injects resource from Spring's ApplicationContext. Use it whenever you would like to access a bean specified in Spring's application context XML configuration.
SpringApplicationContextResource	Injects Spring ApplicationContext resource. When Ignite starts using Spring configuration, the Application Context for Spring Configuration is injected as this resource.
LoggerResource	Injects an Ignite logger resource.
TaskSessionResource	Injects instance of a ComputeTaskSession resource which defines a distributed session for a particular task execution.

In field-based dependency injection, fields or properties are annotated with the appropriate annotation. Ignite set these fields once the class is instantiated.

**Listing 4.13**


---

```
public class IgniteStartUp {
    @IgniteInstanceResource
    private Ignite ignite;
    Public void startup() {
        IgniteCache<Object, Object> cache = ignite.getOrCreateCache(CACHE_NAME);
        // do something
    }
}
```

---

This is the easiest way to pass dependency as it avoids adding boilerplate code and there is no need to declare a constructor for the class as you can see from the above pseudo code. There

<sup>46</sup>[https://en.wikipedia.org/wiki/Dependency\\_injection](https://en.wikipedia.org/wiki/Dependency_injection)

are a few drawbacks of using field-based annotation. Field-based dependency injection will not work on fields that are declared as `final` because this field must be instantiated at the class initialization.



## Tip

Note that, in Apache Ignite dependency injection design pattern are widely used for passing of dependency or a pre-defined resource into a corresponding task, job or SPI before it is initialized.

In the case of method-based annotation, setter methods are annotated with one of the appropriated annotations. Ignite will call these setter methods once the class is instantiated with the appropriate resource passed as an input parameter. Depending on your needs, you should primarily use methods injection or some mix of constructor injection.

## Memory-Centric storage

The challenge with Java and storing large dataset has been two-fold from a decade: how to get that much data into a JVM and how to handle more data that doesn't fit into single RAM? Memory-centric storage architecture is the answer to the above questions. Memory-centric architecture is designed as *memory first* and the latest data or more often used data resides both on disk and in memory to provides in-memory speed. The total dataset can exceed the amount of RAM and can be stored on disk using the memory-centric architecture. The system can process the data in memory or on the underlying disk storage while delivering tremendous performance.

The main difference between the memory-centric approach and the traditional disk-centric approach is that the memory is treated as fully functional storage, not just as a caching layer like most databases does. In this strategy, the database system can function in a pure in-memory mode, where all the datasets will reside in memory. Such a system can be treated as an In-Memory Database or In-Memory data Grid in one. The main drawback of the In-Memory Database is a potential loss of data during node failure and the limitation of the database size. On the other hand, when *persistence* is turned on, the database system begins to function as a memory-centric system where most of the processing happens in memory, but the data and indexes get persisted into the disk.

One of the advantages of using memory-centric architecture is that it provides companies more flexibility and control to balance performance and cost. The ability to exceed the

amount of memory means data can be optimized, so all the data resides on disk but the higher-demand, higher-value data also resides in-memory, while low-demand, low-value data resides only on the disk. This strategy, only available with memory-centric architectures, and can deliver optimal performance while minimizing infrastructure costs.

The other advantage of the memory-centric architecture is that it eliminates the need to wait for all data reloading into RAM in the event of a reboot of any nodes of the cluster. These reloading times can take more than an hour depending on the size of the dataset. The ability to read the data from the disk during node startup enable fast recovery. While initial system performance will be similar to disk-based systems, it will quickly speed up as the data is reloaded into memory and the system can once again perform all or most operations at in-memory speeds.

Starting with version 2.1, Apache Ignite is based on a memory-centric architecture that can be used as an In-Memory Data Grid or a fully functional distributed database with disk durability and strong consistency. The most important thing is that it is highly configurable even at runtime. You can classify Apache Ignite data storage in different modes depending on your application performance requirements:

1. In-Memory on JVM heap
2. In-Memory with Off-heap
3. In-Memory + Full copy on disk
4. 100% on the disk + In-Memory cache
5. In-Memory + 3<sup>rd</sup> party database

Let's start with the in-memory *on-heap* mode. The main reason for using this approach is to get the maximum performance from the Apache Ignite database. Reading and writing data that is purely in memory is faster than data stored outside the JVM heap or data stored on disk. For those that are not familiar with JVM heap, Java Heap memory is a place in RAM, where Java application stores their objects and arrays during execution. Java Heap size is limited for every Java application. The total amount of the heap memory for a JVM is determined by the value set to `-Xmx` parameter when starting the Java application. Java heap storage for objects reclaimed by an automatic storage management system as known as a [garbage collector](#)<sup>47</sup>.

Apache Ignite on-heap storage can be enabled via Spring configuration file or through Java Ignite Cache configuration as follows:

---

<sup>47</sup>[https://en.wikipedia.org/wiki/Garbage\\_collection\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

Listing 4.14

---

```
<!-- Enabling on-heap caching for this distributed cache. -->
<property name="onheapCacheEnabled" value="true"/>
```

---

You can use one of the on-heap eviction policies to manage the growing on heap cache once the on-heap cache is turned on. This mode is preferable for use cases like L2 database caching, web-session clustering, where you only have to store a small set of hot data. One of the corresponding disadvantages of this mode is that all the data must fit in memory and the heap size must not be very large. Because, although modern Garbage collector such as GC1 is a low pause, regionalized and generational, the overhead of the GC pause time exists in current days. The time it takes to complete the GC process will be detrimental to its performance if a cache requires large memory (in GBs). Large heap memory size can adversely impact its performance if the application performance is sensitive. Figure 4.25 shows the relative garbage pause time in the different Java heap size. So,

Big Heap == Slow GC

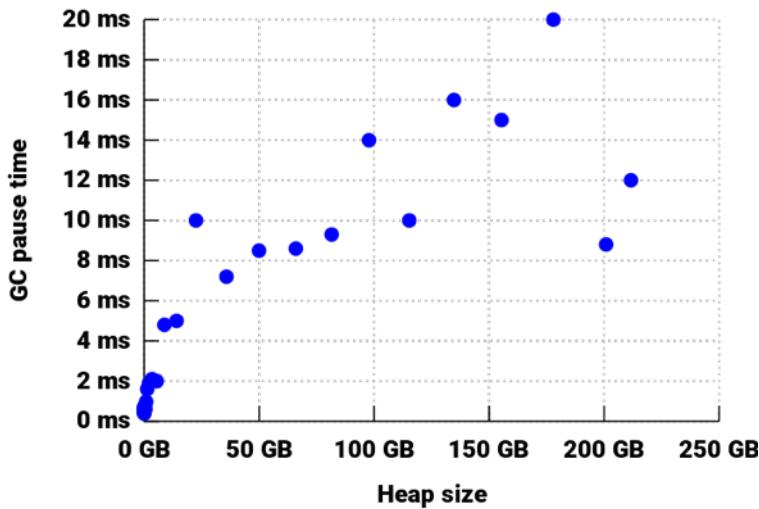


Figure 4.25

In order to mitigate the GC overhead, Apache Ignite provides an in-memory *off-heap* mode which uses the memory outside Java heap and hence reduce the Java heap memory uses and its size. This is the Apache Ignite default memory-centric storage mode. In this mode,

Apache Ignite keeps data in off-heap memory, outside of the Java Heap without the long garbage collection pauses that is common for large JVM heap size. If you are still curious what an off-heap memory is? Off-heap or Heap off memory is:

- A memory outside the Java heap.
- Off-heap memory stores serialized objects.
- No overhead of GC pauses.
- Sharing between processes, reducing duplication between JVMs, and making it easier to split JVMs.
- Scalability to large memory sizes, e.g., over 1 TB and larger than main memory.

There are a few disadvantages of using off-heap memory:

1. Object serialization overhead. Objects are serialized first and stored in the off-heap memory.
2. Retrieving objects are slower than retrieving objects from Java heap but much faster than disk.

In this mode, the entire dataset is stored in memory, and the data is never written to the disk. RAM is treated as the primary storage of the data. Apache Ignite off-heap design principle illustrated in figure 4.27.

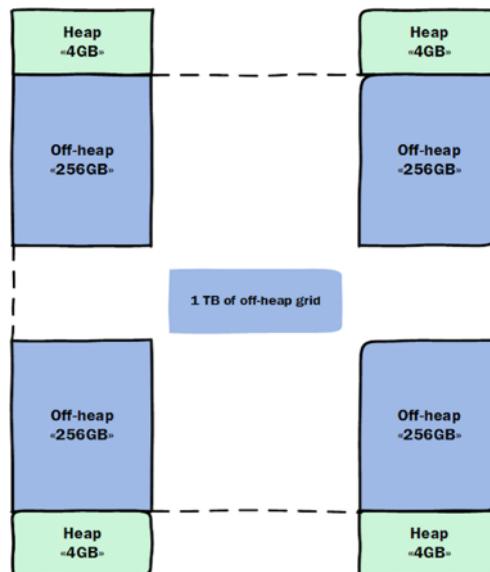


Figure 4.26

Ignite nodes consume 20% of the RAM available locally by default, and generally, you have to configure *dataStorageConfiguration* property for utilizing the rest of the free memory. It is recommended to configure a number of backup copies to prevent possible data loss when a node fails. Ignite also allows page-based eviction policy for the off-heap memory dataset. Whenever the maximum memory size is full, a memory page (memory page is the smallest unit of data which have fixed-length) is evicted from the off-heap memory. Another approach to prevent memory overflow is to use the operating system swap space. When swap space is enabled, Ignite stores data in memory mapped files (MF) which content will be swapped to the disk by the operating system.



## Warning

We highly discourage you not to use swap memory to store memory page. Since swap files always allocated on disk and can significantly reduce database performance.

The off-heap memory storage mode is the simplest, cost-effective way to get lightning-fast, predictable access to a terabyte of data in memory. This memory storage mode is highly recommended for in-memory data grids, in-memory computations, and real-time data processing use cases.

*In-Memory + 3<sup>rd</sup> party* database mode generally uses as a caching layer over any external resources such as Oracle database, PostgreSQL, Cassandra or HDFS. Usually, this mode is used to make persistence the in-memory data or accelerate the underlying database. It was the only way of data durability in Ignite until version 2.0. You can reload the data from the persistence store in the case of node failures. This approach imposed certain drawbacks such as:

1. In this mode, SQL query will be executed only on the data that is **in memory**. If you want to run SQL queries over entire (in memory + external resources) datasets, you have to pre-load the data from the external resources to in memory before executing SQL or Scan queries.
2. It is not possible to run any distributed computation on top of the data that is not in memory.
3. Transactions time is **higher** than usual because Ignite waits for a commit from the database before completing the entire transaction.
4. It is very hard to achieve an RTO (Recovery time objective) without any additional tools.

To address these above restrictions, in version 2.0 Apache Ignite provides a new memory architecture so called *Ignite durable memory* with own persistence implementation that allows storing and processing data and indexes both in memory and on disk when the Ignite native persistence feature is enabled. Now you can execute SQL queries, runs distributed computing on top of the in-memory data and over data on the disk that can be stored into several thousand nodes.

The next two memory modes *In-Memory + Full copy on Disk* and *100% on Disk + In-Memory cache* are belonging to the Ignite durable memory architecture and *only* available when the *Ignite Native Persistence* feature is enabled. In these modes, the dataset will be stored on both RAM and the Disk which reduces the infrastructure costs through maximum hardware utilization. These modes also provide high-performance, persistence storage for durability and ultra-fast restart.

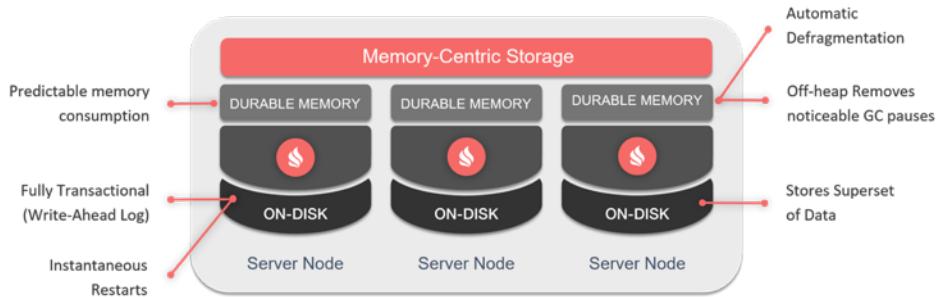


Figure 4.27

In the *In-Memory + Full copy on Disk* mode, the dataset is stored in memory and on Disk. The disk contains a redundant copy of the dataset which is used for data recovery purpose in case of node failure. You should explicitly enable the Ignite native persistence for using this feature. Whenever this option is enabled Ignite becomes a horizontally scalable database that guarantees full data consistency and is resilient to total cluster failure.

In the *100% on Disk + In-Memory cache* mode, the whole dataset is stored on Disk and a small subset of the hot data cached in memory. The disk serves as the primary data storage that survives any node failures or cluster restart. The more data is cached in memory; the faster is the performance. Ignite transparently executes SQL over the entire datasets on disk and in-memory. In this mode, the durable memory will continue to retain the hot data in RAM, automatically purging the raw data from memory when there is no more space is left.

Apache Ignite community redesign the in-memory storage architecture from scratch to build efficient hybrid storage for in memory and disk without duplicating a huge volume of source code. We are going to explore the Ignite durable memory internal architecture and native

persistence in details in the next few sections.

## Durable memory architecture

The Ignite new memory architecture as well as native persistence was debited on version 2.0 and distributed from the end of the last year. The data in memory and on disk has the same binary representation. This means that no additional conversion of the data is needed while moving from in memory to disk. Ignite new memory architecture provides off-heap data storage in a page format. Sometimes it's also called *page-based* memory architecture that is split into pages of fixed size. The pages are allocated in managed off-heap (outside of the Java heap) region of the RAM, and organized in a particular hierarchy. Let's start with the basic of the durable memory architecture: page, the smallest unit of the data with a fixed size.

### Page

A page is a basic storage unit of data that contains actual data or meta-data. Each page contains a fixed length and has a unique identifier: *FullPageId*. As mentioned earlier, Pages are stored outside the Java heap and organized in RAM. Pages interact with the memory using the *PageMemory* abstraction, which usually helps to read, write a page and even allocate a page *ID*.

When the allocated memory exhausted and the data are pushed into the persistence store, it happens page by page. So, a page size is crucial for performance, and it should not be too large. Otherwise, the efficiency of swapping will suffer seriously. When page size is small, there could be another problem of storing massive records that do not fit on a single page. Because, to satisfy a read, Ignite have to do a lot of expensive calls to the operating system for getting small pages with 10-15 records.

When the record does not fit in a single page, it spreads across several pages, and each of them stores only some fragments of the record. The downside of this approach is that Ignite has to look up the multiple pages to obtain the entire records. So, in such cases, you can configure the size of the memory page.

Size of the page can be configured via *DataStorageConfiguration.setPageSize(..)* method. It is highly recommended to use the same page size or not less than of your storage device (SSD, Flash, etc.) and the cache page size of your operating system. Try a 4 KB as page size if it's difficult to figure out the size of the cache page size of your operating system.,

Every *page* contains at least two sections: *header* and *page data*. Page header includes the following information's:

1. Type: size 2 bytes, defines the class of the page implementation (ex. DataPageIO, BplusIO).
2. Version: size 2 bytes, defines the version of the page.
3. CRC: size 4 bytes, defines the checksum.
4. PageId: unique page identifier.
5. Reserved: size 3\*8 bytes.

A high-level view of an Ignite memory page structure illustrated in figure 4.29.

Page -><PageIO>->	Header				Page Data			
	Type	Ver	CRC	Page ID				

Figure 4.28

Memory pages are divided into several types, and the most important of them are *Data Pages* and *Index Pages*. All of them are inherited from the *PageIO* Java class. We are going to details the Data Page and the Index Page in the next two subsections.

## Data Page

The *data pages* store the data you enter into the Ignite caches. If a single record does not fit into a single data page, it will be stored into several data pages. Generally, a single data page holds multiple key-values entries to utilize the memory as efficiently as possible for avoiding memory fragmentation. Ignite looks for an optimal data page that can fit the entire key-value pair when a new key-value entry is being added to the cache. It makes sense to increase the page size if you have many large entries in your application. One thing we have to remember is that data is swapped to disk page by page and the page is either entirely located in RAM or into Disk.

Page -><PageIO>->	Header				Page Data			
	Type	Ver	CRC	Page ID				
<b>Data Page</b> -><DataPageIO>->								
Header					Data Page Data			
	Free space	Direct counter		Indirect counter	it1	it2	it3	Free space V3 V2 V3

Figure 4.29

During an entry updates, if the entry size exceeds the free space available in the data page, then Ignite will look for a new data page that has enough space to store the entry and the new value will be moved there.

Data page has its header information in addition to the abstract page. Data page consist of two major sections: the *data page header* and *data page data*. Data page header contains the following information's and the structure of the data page is illustrated in figure 4.29.

1. Free space, refers to the max row size, which is guaranteed to fit into this data page.
2. Direct count.
3. Indirect count.

The next portions of data after the page header is *data page data*, and consists of *items* and *values*. Items are linked to the *key-value*. A link allows reading key-value pair as an  $N^{\text{th}}$  item in a page. Items are stores from the beginning to the end, and values are stores on reverse order: from the end to the beginning.

## Index pages and B+ trees

Index pages are stored in a structure known as a [B+ tree<sup>48</sup>](#), each of them can be distributed across multiple pages. All SQL and cache indexes are stored and maintained in a B+ tree data structure. For every unique index declared in SQL schema, Ignite initialized and managed a dedicated B+ tree instance. Unlike data pages, index pages are always stored in memory for quick access when looking for data.

A B+ tree structure is very similar to a *B tree* with the difference that an additional level is added at the bottom with linked leaves. The purpose of the B+ tree is to link and order the index pages that are allocated and stored within the durable memory. It means that only a small number of pointers or links traversal is necessary to search for value if the number of the keys in a node is very large. Finally, the index pages of the B+ tree all contain a next sibling pointer for fast iteration through a contiguous block of value.



### Info

Key duplication is not possible in B+ tree structure.

In B+ tree binary search is used to find out the required key. To search for an element into the tree, one look up the root nodes finds the adjacent keys that the searched-for value is between. If the required value is not found, it is compared with other values in the tree.

<sup>48</sup>[https://en.wikipedia.org/wiki/B%2B\\_tree](https://en.wikipedia.org/wiki/B%2B_tree)

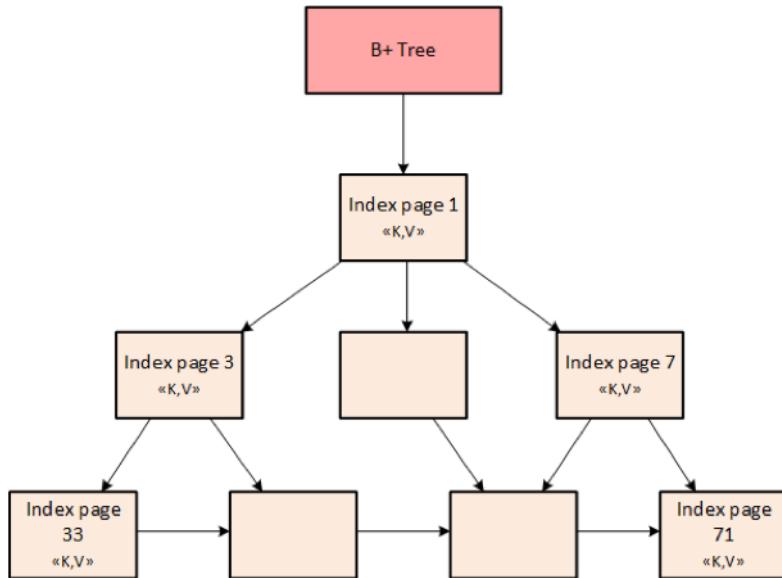


Figure 4.30

There is a high cost of allocating memory for a large number of pages including data or index pages, which solves through the next level of abstraction called *Segments*.

## Segments

Segments are a contiguous block of physical memory, which are the atomic units of the allocated memory. When the allocated memory runs out, the operating system is requested for an additional segment. Further, this segment is divided into pages of fixed size. All page types include data or index pages resides in the segment.



Figure 4.31

It is possible to allocate up to 16 memory segments for one dedicated memory region with the size of the segments at least 256 MB in the current version. Ignite uses a particular component for managing information about pages currently available in memory segment and page Id mapping to region address called *LoadedPagesTable*. *LoadedPagesTable* or *PageIdTable* manages mapping from Page ID to relative memory segment chunk (unsafe).

LoadedPagesTable uses [Robin Hood Hashing<sup>49</sup>](#) algorithm for maintaining HashMap of FullPageId since Ignite version 2.5.

When it comes about memory segment, it is necessary to mention the memory consumption limits. In Apache Ignite data are stored in caches. Obviously, we cannot keep the entire dataset forever in memory. Also, different data may have different storage requirements. To make it possible to set limits at the level of each cache, a hybrid approach was chosen that allows Ignite to define limits for groups of caches, which brings us to the next level of abstraction called memory *Region*.

## Region

The top level of the Ignite durable memory storage architecture is the data Region, a logical expandable area. Data region can have a few memory segments and can group segments that share a single storage area with their settings, constraints and so on. Durable memory architecture can consist of *multiple data regions* that can vary in size, evictions policies and can be persisted on disk.



### Tip

Ignite allocates a single data region (default data region) occupying up to 20% of the RAM available on a local cluster by default. The default data region is the data region that is used for all the caches and not explicitly assigned to any other data region.

Data region encapsulates all the data storage configuration for operational and historical data in Ignite and can have one or more caches or tables on a single region. With data region configuration you can manage more than one data region, which can be used for storing historical and operation data of your system. There are different cases when you might do this. The most trivial example is that when you have different non-related caches or tables with different limits.

---

<sup>49</sup><http://codecapsule.com/2013/11/17/robin-hood-hashing-backward-shift-deletion/>

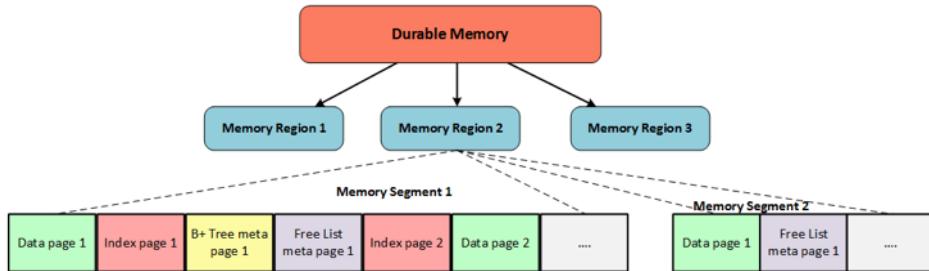


Figure 4.32

Let's assume that in our application we have *Product*, *Purchase history* entities stored in the *ProductCache* and the *PurchaseHistCache* caches respectively. Here, the Product data is operational and access by the application frequently. Moreover, the Purchase History data needed occasionally and not very critical. In this situation, we can define two different regions of memory with different sizes: *Data\_region\_4GB* and *Data\_region\_128GB* as shown in figure 4.33.

- *Data\_region\_128GB* is only 128 GB of memory and will store the operational or frequently accessed data such as Products.
- *Data\_region\_4GB* size is 4 GB and will be allocated for rarely accessed datasets like Purchase history.

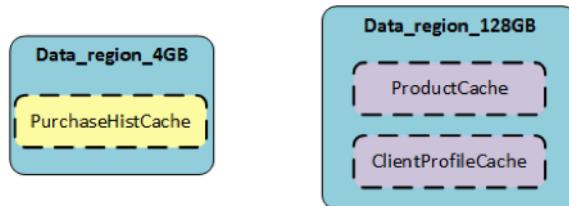


Figure 4.33

When we create caches, we should have specified the region, on which the cache will belong to. The limits here are applied to the data region level. When you put or insert something in your small cache, and if you exceed the maximum size of the data region (ex. 4 GB), you will get out of the memory (*IgniteOutOfMemory*) exception, even when the larger data region is empty. You can't use the memory that is allocated for the *Data\_region\_128GB* by the small caches, because it is assigned to the different data region.

So, you should remove or swap the stale data from the data region if you want to avoid this \_out of memory error-. For these circumstances, Ignite provides a few data eviction algorithms to remove unnecessary data from in memory.

## Data eviction

An eviction and expiration both are means of cleaning the cache of unused entries and thus guarding the memory against the OutOfMemory exception. With eviction, you set the maximum size of the cache or data region you want to keep in the cache, and if the limit exceeded, some entries are found to be evicted (remove) according to a chosen eviction policy. An eviction algorithm is a way of deciding which entries to evict (removed) when the cache is full.



### Info

With expiration, you set time criteria for entries, and how long you want to keep them in the cache. The entry eliminates from the cache once expired.

Apache Ignite supports two different types of data evictions starting with version 2.0:

- Page based eviction, newly added eviction strategy in the version 2.0, removing memory pages from the off-heap memory.
- Entry based eviction, removing entries from the Java heap memory or On-Heap memory.

The next section describes different types of data evictions in details.

### Page based eviction

If the RAM filled with pages and it is required to allocate new. It is required to evict a few pages from the memory. Evection occurs individually on a *per node* basic, rather than occurring as a cluster-wide operation. On each node, eviction modes tracks and analyze the size of the data page in-memory to determine which pages require eviction.

Note that, if durable memory operates with the disk, or in other words Ignite Native Persistence is enabled, page evictions is handled automatically (the oldest pages will be evicted from RAM automatically if there is not enough space available) by Ignite and explicitly page based eviction configuration for memory region will be ignored.

Ignite page-based eviction sets for a specific memory region. Only data pages that store key-value entries are eligible for eviction. The other types of pages such as Index pages or Meta pages are not competent for eviction. There are three different types of eviction modes

(policies) available to configure data region, which defines how the data evicts from the memory.

**Disable.** This is the default page eviction mode. In this mode, data eviction is disabled for the specified data region. In disabled mode, data page will not be purged from the memory when there is no more room for new entries. In such a situation, you will get an `OutOfMemory` exception. To enable page eviction from the memory, you have to use one from the next two modes: *Random-LRU* or *Random-2-LRU*.

**Random-LRU.** It enables the Random-LRU algorithms, deals with which pages need to be swapped out from the off-heap memory. Basically, it tracks (an off-heap array is allocated for each region) the latest access timestamps for each page that are currently stored in off-heap memory. When Ignite needs to evict one of the pages, it peeks 5 random pages out of all pages stored in memory. For those 5 pages, it peeked the one for eviction that was not used for the longest period of time. In other words, it peeked the page with the minimum value for the last access timestamp. If some of the pages related to non-data pages such as Index pages or Meta pages (B+ tree) then the algorithm picks other pages.

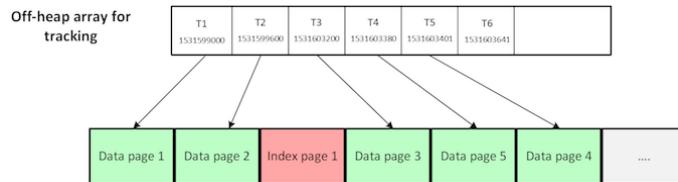


Figure 4.34

**Random-2-LRU.** This is the second option for evicting pages if Ignite native persistence is not *enabled*. This algorithm is distinguished from the Random-LRU in a way that two most recent access timestamps are stored in an array for each data page: *latest timestamp* and the *previous latest timestamp*. At the time of eviction, the algorithm randomly chooses 5 indexes from the tracking array, and the minimum between two latest timestamps is taken for further comparison with corresponding minimums of four other pages that are chosen as eviction candidates. In the case of eviction, the oldest timestamp is used for making eviction decision.



Figure 4.35

Random-2-LRU algorithm is used in a situation, when a data page is hit only once after a very long period of time (for example, by scan query occasionally) and you still want the page to be evicted. However, with the Random-LRU algorithm, the page will not be evicted a long time because the latest access timestamp is very recent. So, the data page will be stored in memory for a very long time regardless of you do not need it anymore. Random-2-LRU fixes this issue (one-hit-wonder) because the latest timestamp will be very recent but the previous timestamp that actually used by the algorithm to choose the data page for eviction is oldest.

Random-LRU works perfectly fine in most cases, but you may consider switching into the Random-2-LRU if you have a particular situation (one-hit-wonder) that we have described above.

## Entry based eviction

Entry-based eviction policy controls the maximum numbers of entries that can be stored in an on-heap memory. The caches in on-heap memory could be near caches (see the section Near cache in this chapter for more information) or any on-heap caches that are configured by `CacheConfiguration.setOnheapCacheEnabled(...)`. Entries are evicted from the Java heap whenever the maximum on-heap cache size is reached.



### Info

Entries that are evicted according to your eviction policy will remain in the persistence store if you are using 3<sup>rd</sup> party database like Cassandra or PostgreSQL as a persistence store. However, entries that are evicted from the on-heap memory will not be returned through Ignite SQL queries.

Apache Ignite out-of-the-box provides four different predefined entry-based eviction policies to control the eviction process of entries from the on-heap caches. Each of them implements

the pluggable `EvictionPolicy` interface, which can also notify whenever an element needs to be evicted from the cache. Apache Ignite ensures the thread-safe of the eviction process. All the entry-based eviction policies provide by the Ignite are very light-weight and lock-free (or very close to it), and do not create any internal tables, arrays or other expensive structures. The eviction order is preserved by attaching light-weight meta-data to existing cache entries.

Some entry-based eviction policies support eviction by *memory size limit* and *batch eviction*. If eviction by memory size limit is enabled, then the eviction starts when the size of the cache in bytes become larger than the maximum memory size. If a batch eviction is enabled then eviction starts when cache size (`batchSize`) is greater than the maximum cache size. In this case, `batchSize` entries will be evicted.



## Tip

Batch eviction is enabled only if maximum memory limit isn't set (`maxMemSize == 0`). The default `batchSize` value is 1.

Also, eviction policies should be used as required by your application to set up which entries are evicted and when an eviction occurs.

- LRU (Least recently used). This eviction policy is based on the [LRU algorithm](#)<sup>50</sup>. The oldest element is the Less Recently Used (LRU) element (i.e., the entry that has not been touched for the longest time) gets evicted first. The last used timestamp is updated when an element is put into the cache, or an element is retrieved from the cache with a get call. This algorithm takes a random sample of the elements and evicts the smallest. By using the sample size of 15 elements, empirical testing shows that an entry in the lowest quartile of use is evicted 99% of the time. This eviction policy supports batch eviction and eviction by memory size limits. This eviction policy is suitable for most of all applications and recommended by Apache Ignite. This eviction policy can be configured by Java and Spring configuration.
- FIFO (First In First Out). Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store), the element that was placed first (First-In) in the cache is the candidate for eviction (First-Out). This algorithm is used if the use of an element makes it less likely to be used in the future. An example here would be an authentication cache. It takes a random sample of the entries and evicts the smallest. By using the sample size of 15 entries, empirical testing shows that an entry in the lowest quartile

---

<sup>50</sup>[https://en.wikipedia.org/wiki/Cache\\_replacement\\_policies](https://en.wikipedia.org/wiki/Cache_replacement_policies)

of use is evicted 99% of the time. This implementation is very efficient since it does not create any additional table-like data structures. The ordering information is maintained by attaching ordering metadata to cache entries. This eviction policy supports batch eviction and eviction by memory size limit. This eviction policy is implemented by `FifoEvictionPolicy` and can be configured via `CacheConfiguration` property.

- **Sorted.** Sorted eviction policy is similar to FIFO eviction policy with the difference that the entries order is defined by default and ensures that the minimal entry (i.e., the entry that has integer key with the smallest value) gets evicted first. The user can provide their own comparator implementation which can use keys, values or both for entries comparison. This eviction policy supports both batch eviction and eviction by memory size limit. Sorted eviction policy is implemented by `SortedEvictionPolicy` and can be configured via `CacheConfiguration` property.
- **Random.** This cache eviction policy selects random cache entry for eviction if cache size exceeds the `maxSize` parameter. This implementation is extremely lightweight, lock-free, and does not create any data structures to maintain any order for eviction. Random eviction will provide the best performance over any elements in which every element has the same probability of being accessed. This eviction policy implementation doesn't support near cache and doesn't work on client nodes. This eviction policy is mainly used for debugging and benchmarking purposes. This eviction policy is implemented by `RandomEvictionPolicy` and can be configured via `CacheConfiguration` property.



## Tip

Random eviction policy is deprecated since 1.5 version and not recommended for use in a production environment.

## Data expiration

A cache expiration policy is a combination of concepts which define when a cache entry expires. The fundamental difference between the cache eviction and expiration is that, expire policy specifies the amount of the time that must pass before an entry is considered to be expired. A cache entry may be removed from the cache once it has expired. For instance, the cache could be configured to expire entries ten seconds after they are put in. Sometimes, it is called Time-to-live or TTL.

Apache Ignite provides five different predefined expiration policies as follows.

Policy name	Description	Creation time	Last access time	Last update time
CreatedExpiryPolicy	Defines the expiry of a Cache Entry based on when it was created. An update does not reset the expiry time. Sometimes it's also called absolute expiration.	Used		
AccessedExpiryPolicy	Defines the expiry of a Cache Entry based on the last time it was accessed. Accessed does not include a cache update.	Used	Used	
ModifiedExpiryPolicy	Defines the expiry of a Cache Entry based on the last time it was updated. Updating includes created and changing (updating) an entry.	Used		Used
TouchedExpiryPolicy	Defines the expiry of a Cache Entry based on when it was last touched. A touch includes creation update or access.	Used	Used	Used

Policy name	Description	Creation time	Last access time	Last update time
EternalExpiryPolicy	Specifies that Cache entries won't expire. This however doesn't mean they won't be evicted if an underlying implementation needs to free-up resources whereby it may choose to evict entries that are not due to expire.			
CustomExpiryPolicy	Implements javax.cache.expiry interface which defines functions to determine when cache entries will expire based on creation access and modification operations			

Expire policy can be configured in Ignite *CacheConfiguration*, which will be used for all entries inside the cache. Also, it is possible to configure expire policy for individual operations on the cache.



## Warning

There is no default expiry policy in Apache Ignite. You should configure expiry policy manually based on your business requirements.

Apache Ignite data expiration policy is a cluster-wide operation rather than node per basic. The specific expiration policy depends on the memory configuration of the entire cluster. The expiration actions depend on the Ignite memory configuration is illustrated in figure 4.36.

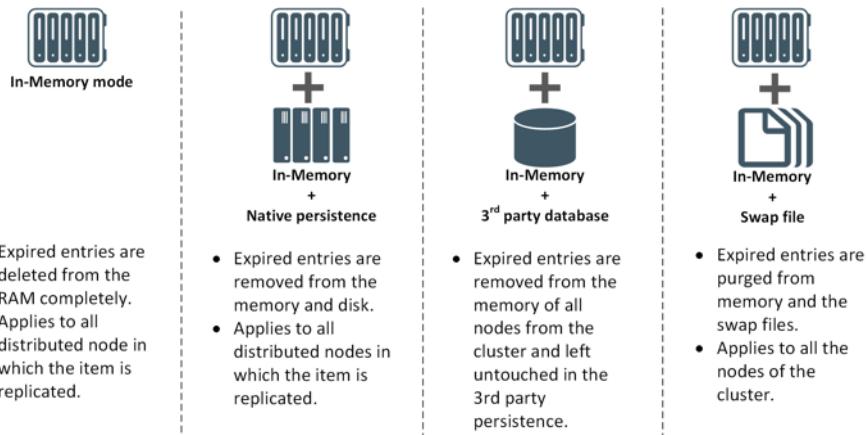


Figure 4.36

Apache Ignite allows removed expired entries actively from the cache. Ignite creates a single thread per nodes (depends on PARTITIONED and REPLICATED cache) to clean up expired entries in the background if there is at least one cache configured with *EgarTTL* enable. Anyway, it is also possible to preserve the data that is expiring. You can enable certain event types via configuration and catch them with a continuous query. However, it's not recommended to enable too many events for performance reasons.

## Ignite read/write path

Ignite uses a B+ tree index to find out the potential data pages to fulfil a read. Ignite processes read data at several stages on the read path to discover where the data is stored, starting looking up the key in the B+ tree and finishing with data page. The whole process can be split into the following steps:

1. On the client node, a cache method has been called `myCache.get(keyA)`.
2. Client node identifies the server node that is responsible for this given key `keyA` using the built-in affinity function and delegates the request to the server node over the network.
3. The server node determines the memory region that is responsible for the cache `myCache`.
4. In the corresponding memory region, a request goes to the meta page, which contains the entry points to a B+ tree by the key of this cache.
5. Based on the `keyA` hash code, the index page the key belongs to will be located in the B+ tree.

6. Ignite will return a null value if the corresponding index page is not found in the memory or on the disk.
7. If the index page exists, then it contains the reference to the data page of the entry keyA.
8. Ignite accesses the data page for keyA and returns the value to the client node.

The above schema for data looks up by the key can be illustrated as shown in figure 4.38.

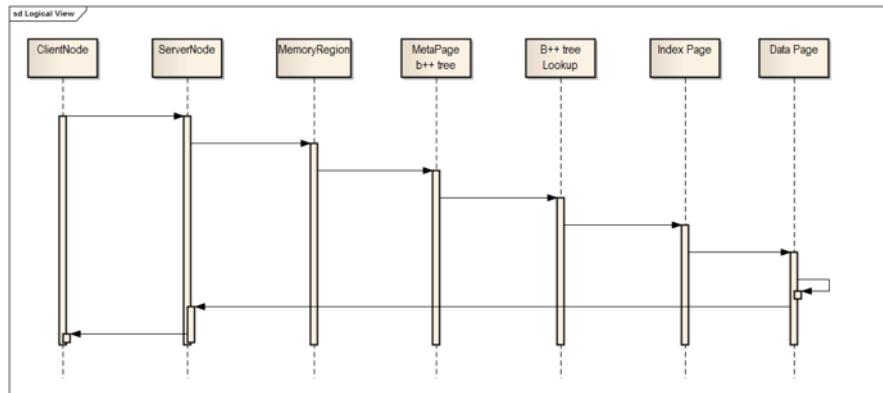


Figure 4.37

Similar to the read path, Ignite processes data at several stages on a write path. The only difference is that, when a write occurs, Ignite looks for the corresponding index page in the B+ tree. If the index page is not found, Ignite requests a new index page from one of the free lists. The same thing happens for the data page — a new data page also requests from the free list.

*Free List* is a list of pages, structured by an amount of space remained within a page. Ignite manages free lists to solve the problem of fragmentation in pages (not full page). Free lists make the allocation and deallocation operations of the data and index pages straightforward, and allow to keep track of free memory. For instance, the image in figure 4.39 shows a free list that stores all the data pages that have up to 15% free space available. Data and index pages are tracked in separate free lists. The list is traversed, and the data/index page that is large enough to store the data is returned when a request for a data/index pages is sent.

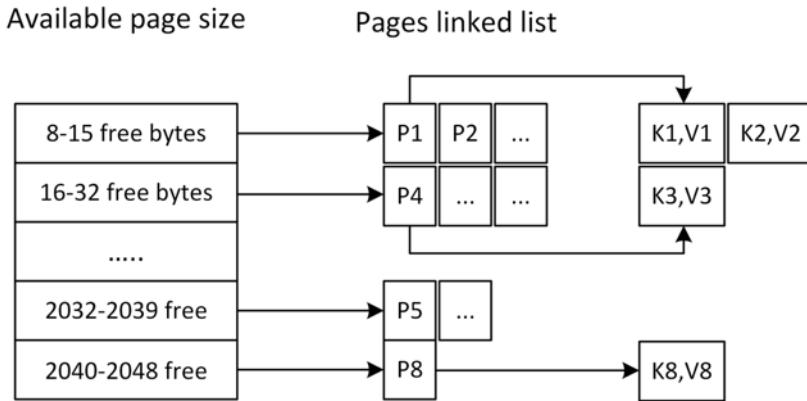


Figure 4.38

Let's see what's going under the hood when a `myCache.put(keyA, valueA)` request sent to the Ignite node:

1. A cache method `myCache.put(keyA, valueA)` has been called on the client node.
2. Client node identifies the server node that is responsible for this given key `keyA` using the built-in affinity function and delegates the request to the server node over the network.
3. The server node determines the memory region that is responsible for the cache `myCache`.
4. A request goes to the Meta page in the corresponding memory region, which contains the entry points to a B+ tree by the key of this cache.
5. Based on the `keyA` hash code, the index page the key belongs to will be located in the B+ tree.
6. If the corresponding index page is not found in the memory or on disk, then a new page will be requested from one of the free lists. Once the index page is provided, it will be added to the B+ tree.
7. If the index page is empty (i.e., does not refer to any data page), then the data page will be provided by one of the free lists, depending on the total cache entry size. During the selection of the data page for storing the new key-value pair, Ignite does the following:
  - Consult marshaller about size in bytes of this value pair.
  - Upper-round this value to be divisible by 8 bytes.
  - Use the value from the previous step to get page list from the free list.
  - Select some page from an appropriate list of free pages. This page will have required amount of free space.
  - A reference to the data page will be added to the index page.

8. The cache entry is added to the data page.

The Ignite write path with several stages illustrated in the following sequence diagram.

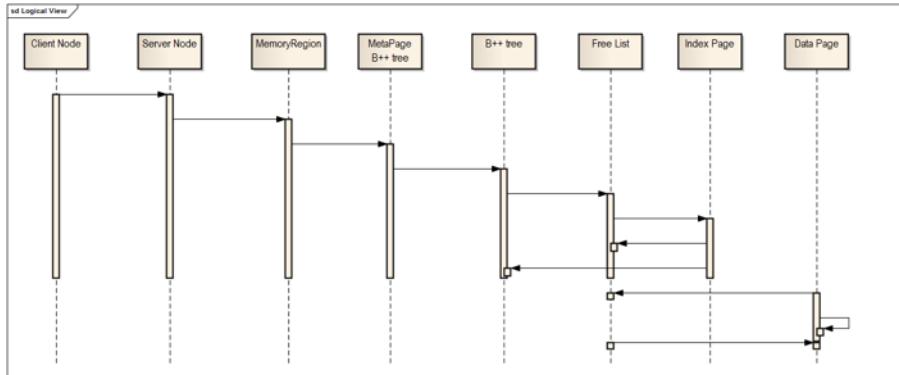


Figure 4.39

## Native persistence

We have talked a lot about Ignite durable memory and its architecture so far. The key idea behind the design can persist data, keeping all of the data on disk along with indexes such a way that, when we run an SQL query, Ignite fetches the data from disk if it is not available in memory. The way that Ignite persists data on disk is named Ignite native persistence. Ignite Native Persistence is the most flexible, scalable and convenient way of persisting data on the disk.

Thereby, from version 2.1.0, Apache Ignite provides ACID and SQL-compliant disk stores that transparently integrate with Ignite's durable memory as an optional disk layer storing data on SSD, Flash, 3D XPoint, and other types of non-volatile storage systems. Ignite persistence is an *optional* configuration and can be turned on and off. Apache Ignite stores a superset of data on disk, and a subset of data in RAM based on its capacity whenever the feature enables. The Ignite durable Memory will take it from the disk as shown below if a subset of data is missing in RAM:

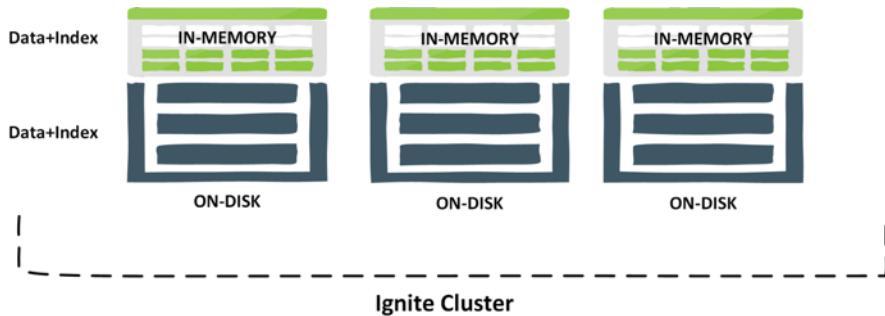


Figure 4.40

Data can also be stored in the central disk storage where all the Ignite nodes are connected as shown in figure 4.41.

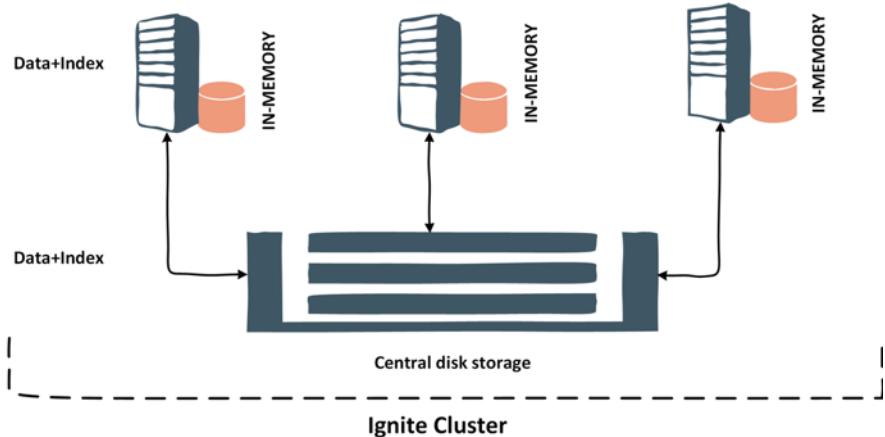


Figure 4.41



## Tip

Ignite becomes a pure in-memory data store when Ignite native persistence turned off. Ignite native persistence is disabled due to performance reasons by default.

It all started with the fact that, 3<sup>rd</sup> party persistence store such as DBMS was the only way to persistence data in Apache Ignite before the version 2.0. There are a few limitations of using traditional third-party DBMS as a persistence store in conjunction with Apache Ignite as described below:

1. **Inability to execute SQL queries over the whole data sets available in memory and 3<sup>rd</sup> party database.** Ignite SQL queries cannot span over the datasets stored in the 3<sup>rd</sup> party database. Because Ignite meta-data index and index pages do not have any idea what data is in it and which is not. So, if you want to execute SQL over data stored in 3<sup>rd</sup> party database, you have to have reloaded the datasets into the Ignite in-memory first.
2. **Increase the duration of the transaction time.** Whenever you use the 3<sup>rd</sup> party DBMS as a persistence store, duration of the ACID transaction will be increased. Because Ignite always waits for the commit phase from the DBMS. In some cases, it's caused a severe bottleneck for the write-intensive application.
3. **Warming up the data after Ignite node restart.** In case of Ignite node restart, you have to preload the data from the DBMS into the Ignite node before executing SQL. The preloading data process can take an immense amount of time for a large volume of data. This process is getting complicated whenever we need a complete Ignite cluster restart. For instance, it might take more than an hour to reload the 3 terabytes of data into the Ignite cluster.

Here, we have only described a few of the disadvantages of using 3r party DBMS. Besides that, it might be a single point of failure if you are using RDBMS for persisting data. So, Ignite implemented their native persistence mechanism which allows us to organize distributed, horizontally scalable storage to solve the problems described above. As a general summary, Ignite native persistence has the following characteristics and benefits:

1. *Optional persistence:* easy to configure how data will be stored in-memory, in-memory + disk or data will be stored as a caching layer.
2. *Storage scaled along with the cluster:* with the native persistence, Ignite node stores its portion of data not only in memory but also on disk. This gives near-linear scalability of the storage system.
3. *Data resiliency:* native persistence stores the full data sets and can survive cluster crashes and restarts without losing any data and still providing a strong transactional consistency. This allows you to make a backup copy of the entire disk space online. The backup copy can be used for the disaster recovery or deployed on a different cluster.
4. *Executing SQL queries over entire dataset:* therefore, Ignite durable memory tightly integrated with the native persistence and stores index and the data into in memory and the disk, the platform has comprehensive information of the location of data for performing SQL queries over the entire data sets.
5. *Caching hot data in memory:* when the native persistence feature is enabled, hot data can be kept in memory, and automatically purge cold data from the memory when memory space is reduced.

6. *Lazy loading of data and fast cluster restart*: with the native persistence feature, the Ignite cluster can start almost instantly. The in-memory data will be lazy loaded automatically, as the data starts getting accessed. This gives a significant speedup recovery after an accident (DR DTO), reduce the risk for business and costs of testing and development procedures by saving the time on the warm-up.
7. *Reducing unnecessary network interaction*: in this case, native persistence provides a higher level of performance than 3<sup>rd</sup> party DBMS due to lack of unnecessary network interaction.

Furthermore, the data transition between the in-memory and the disk drive storage layer is fluent and transparent. The entire process is very straightforward, and the developer does not need to know where the data is actually stored, which gives a huge difference from any other database system. Now, let's discuss two mechanisms that Ignite uses to manage persistence: *Write-ahead-log* (WAL) and *checkpointing*. We will begin with the WAL.

## Write-Ahead-Log (WAL)

The Write-Ahead-Log or WAL is a commonly used technique in the database system for maintaining atomicity and durability of writes. The key behind the WAL is that before making any changes to database state, first, we have to log the complete set of operations to the nonvolatile storage (e.g., disk). By writing the log into WAL first, we can guarantee the data durability. If the database crash during changes to the disk, we will be able to read and replay the instructions from the WAL to recover the mutation.



### Tip

WAL also known as the transaction log or redo log file. Practically every database management system has one.

From the Apache Ignite perspective, WAL is a dedicated partition file stored on each cluster node. The update is not directly written to the appropriate partition file but is appended to the end of the WAL file when data are updated in RAM. WAL provides superior performance when compared to in-place updates.

So, what exactly is a Write-Ahead-Log (WAL) file and how it works? Let's consider an application that's trying to change the value of A and B from the following four key-values:

$(K, V) = (A, 10);$   
 $(K, V) = (B, 10);$   
 $(K, V) = (C, 20);$   
 $(K, V) = (D, 30);$

The application is performing an addition of 10 within a single transaction as shown below.

$A := A + 10;$   
 $B := B + 10;$

The problem arises when there is a system failure during writing to the disk. Assume that, after  $output(A)$  on disk, there is a power outage, so  $output(B)$  does not get executed, and the value of B is now in the inconsistent state. Value of A on disk is 20, and the value of B is still 10. Therefore, the database system needs a mechanism to handle such failures since they cannot be prevented from any power outage or system crash.

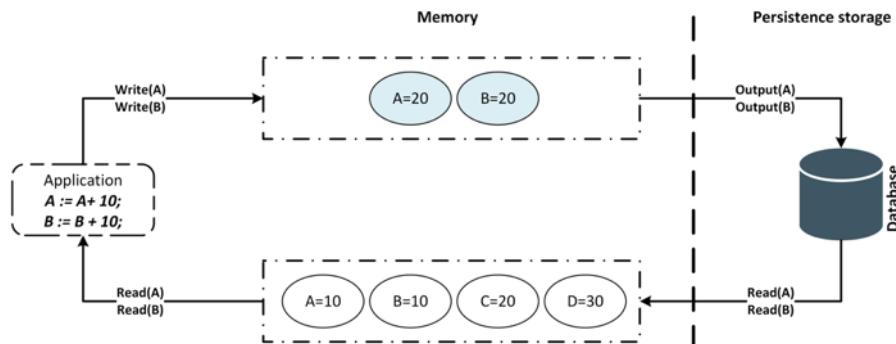


Figure 4.42

Most database system uses a log-based database recovery mechanism to solve the above problem. A *log* is the most commonly used structure for recording database modification. The DBMS has enough information available to recreate the original data changes after a crash after the log file has been flushed to disk.

The first log approach is the **UNDO log**. The purpose of the undo log is to reverse or undo the changes of an incomplete transaction. In our example, during recovery, we have to put the database in the state it was before this transaction, means that changes to A are undone, so A is once again 10 and  $A=B=10$ . The undo log file always written to the nonvolatile storage.

#### Undo logging rules:

1. Record a log in undo log file for every transaction T. Write (start T).

2. For every action, generate an undo log record with the old value. Write  $(T, X, V^{\text{old}})$ .
3. Flush the log to disk.
4. Write all the database changes to disk if transaction T commits.
5. Then write (commit T) to the log on disk as soon as possible.

An undo log looks very similar as shown in the figure 4.43.

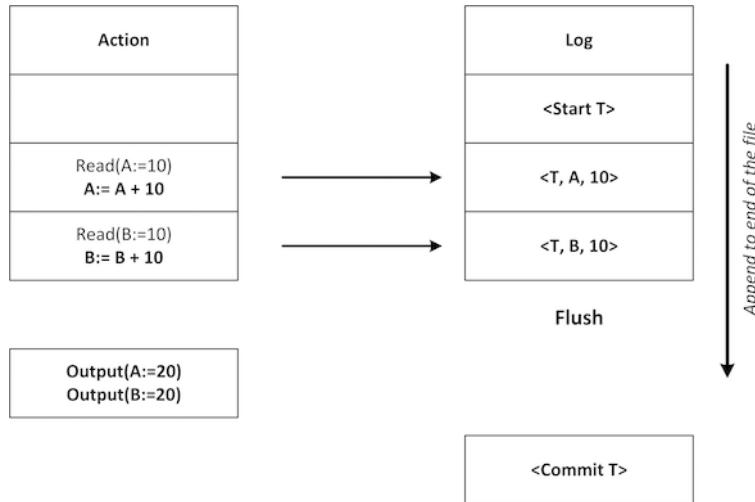


Figure 4.43

We log a record indicates that we have started the transaction before starting the transaction. When we update the value A, we also write a log indicates its old value 10. Similarly, we record its old value of 10 when we change the value of B from 10 to 20. We flush the undo log to disk before outputting values of A and B to disk. Then we output(A) and output(B) to disk, only after that, we can record (commit T) into undo log file.

#### Undo logging recovery rules:

1. We only undo the failed transaction. If there's already (commit T) or (abort T) record, do nothing.
2. For all  $(T, X, V^{\text{old}})$ :
  - $\text{output}(V^{\text{old}})$
3. write (abort T) to undo log.

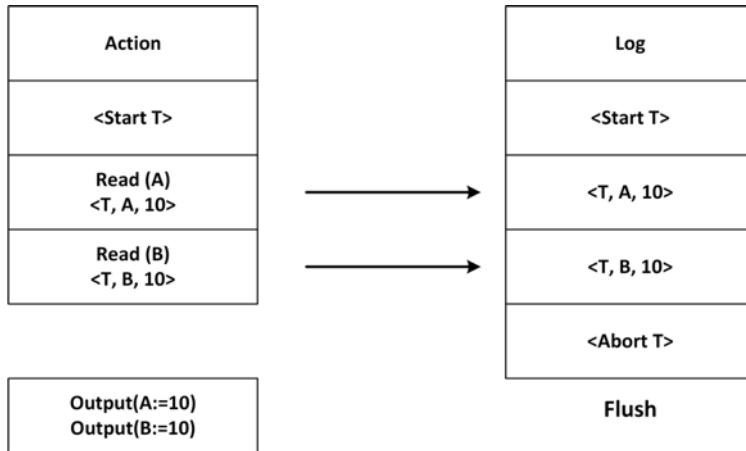


Figure 4.44

We read the undo log from the end to start, and looking for an incomplete transaction during the recovery process. Any records with (commit T) or (abort T) are ignored because we know that (commit T) or (abort T) can only be recorded after a successful output to disk. We cannot be sure that output was successful if there are no (commit T) record, so for every record, we use the old value  $V^{\text{old}}$  to revert the changes. So, (T, B, 10) sets B back to 10 and so on. Undo log records (abort T) to indicate that we aborted the transaction after making the changes.

The main disadvantage of the undo log is that it might be slower for heavy write-intensive application because for every transaction, we have to output the value to the disk before records a (commit T) log in the undo log file.

At this moment, we can get back to our starting point about WAL. The second log approach for protecting data-loss is the write-ahead log or WAL. Instead of undoing a change, WAL tries to reproduce a change. During the transaction, we write all the changes to WAL that we are intended to do, so we can rerun transaction in case of disaster and reapplying the changes if necessary. Before making any output (write to the disk), we must record the (commit T) record.

### WAL logging rules:

1. Record a log into the file for every transaction T. Write (start T) to the log.
2. Set its value to *New* if transaction modifies database record X. Write (T, X,  $V^{\text{new}}$ ) to the log.
3. Write (Commit T) to the log if transaction T commits.
4. Flush the log file to the disk.

5. And then, write the new value  $V^{new}$  for X to the disk.

A WAL log file looks something like shown in figure 4.45.

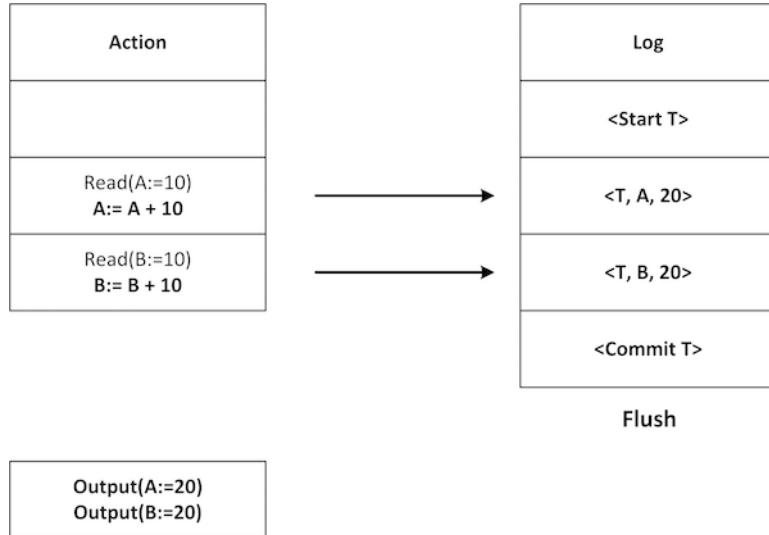


Figure 4.45

We record the new values for A and B then commit and flush the log to disk. Only after that, we output the values of A and B to the disk. This solves two main issues with disk I/O: buffering and randomly output to disk.

#### WAL logging recovery rules:

1. Do nothing if there's any incomplete transaction (no commit T) record.
2. If there is (commit T), for all (T, X,  $V^{new}$ ):
  - output( $V^{new}$ )

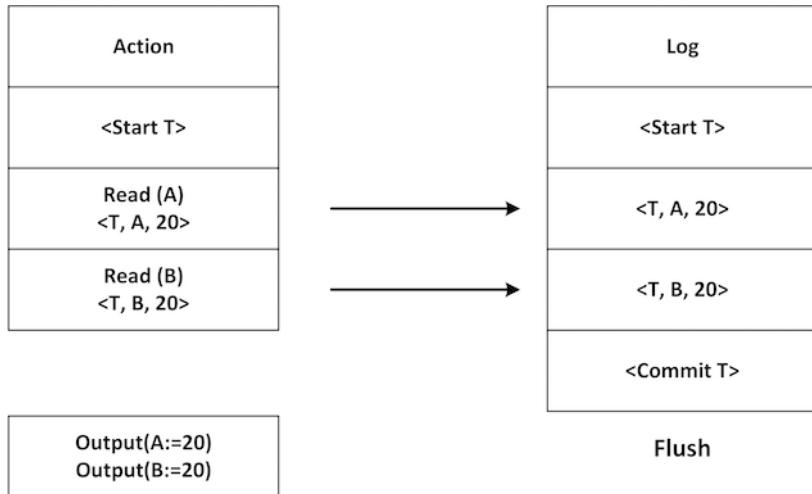


Figure 4.46

To recover with a WAL log file, we start from the beginning of the file scanning forwards (opposite of the undo log file). If we find any incomplete transaction (no commit T), we skip the transaction so that no output was done. We do not know whether the output was successful or not whenever we find any (commit T) record. In this case, we redo the changes, and even it is redundant. In our example, the value of A will be set to 20, and the value of B will also be set to 20.

Now that we have got the basics of the log structure, so let's move on to Ignite's WAL concept to see how the things organized under the cover. From the Ignite perspective, whenever the storage engine wants to make any changes to the data page, it writes the change to the RAM and then appends the changes to the WAL. Storage engine sends an acknowledgment to confirm the operation only after durably written the changes to WAL file on disk.

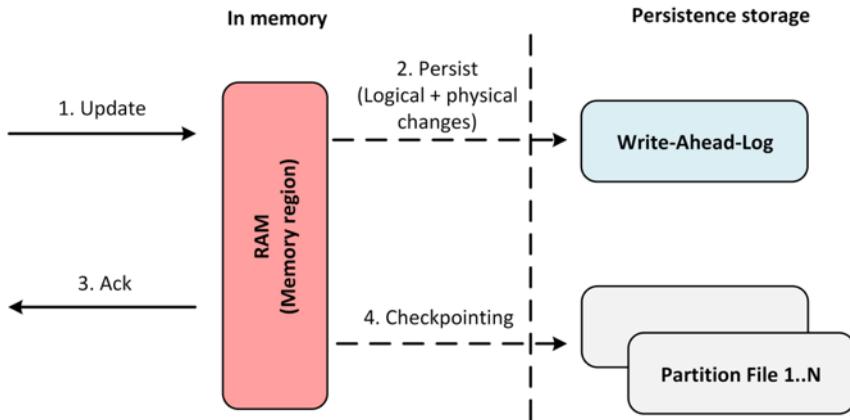


Figure 4.47

This makes the database changes reliably. If the node crashes while data was being appended to the WAL, no problem because dirty data pages have not been copied from RAM to disk. So, storage engine can read and reply WAL using already saved page set if it crashes while the data pages are being modified. The storage engine can restore to state, which was last committed state of the crashed process. In Ignite, restore is based on page store and WAL log. You may notice that Ignite native persistence is slightly different than the classical WAL log concept.



## Tip

Data changes are acknowledged only after the cache operations and page changes were logged into the WAL. Dirty data pages will be copied later by another process.

Physically, WAL log is split into several files, called segment. Each segment is fixed in size and numbered in order from 0 to 9. By default, each segment occupies approximately ~64 MB on disk; this size is configurable by the `walSegmentSize` property. These segments are filled sequentially. By default, ten segments are created while native persistence is enabled and this number is configurable by the `walSegments` property. So, WAL log for each node may grow up to ~640 MB of data. These WAL segments files are used as follows:

1. When the first segment is full, it is copied to a WAL archive file.
2. This WAL archive is kept for a configurable period of time. The number of files in the WAL archive depends on the `walHistSize` property, which is 20 by default.

3. While the 1<sup>st</sup> segment's content is being copied to the archive, the 2<sup>nd</sup> segment becomes the active WAL file. This process is repeated for each segment file.

```
-rw-r--r-- 1 shamil staff 67108864 Jul 27 14:23 0000000000000000.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000001.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000002.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000003.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000004.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000005.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000006.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000007.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000008.wal
-rw-r--r-- 1 shamil staff 0 Jul 24 12:00 0000000000000009.wal
```

Figure 4.48

Generally, the WAL log ends up in the critical path for all update operations, since we have to writes every update operation before copied the dirty pages to the partition files. This raises two questions: how to handle the expensive I/O operations to disk and how to provide consistency guarantees (we know that crash can happen any time: operating system failure, JVM or process crash)? We have to understand the mechanism of writing data to disk by the operating system to answer this particular question.

Data is not directly written to the disk in an operating system. For instance, files are cached in memory in the *page cache buffer* in Linux. These cache help improve performance since applications often read what they write recently, and applications do not always read and write in sequential order. Typically, Linux OS operates in a write-back mode, where the buffer cache is only flushed to disk after a certain amount of time, for instance, every  $\sim 30$  seconds. It means that even applications complete the write, data may not be on disk yet, and a power loss or OS crash at this moment could result in a data loss. You additionally have to call `fsync()`<sup>51</sup> to force the data flushed to disk. Usually, people put operation on a cache writing to the disk because full sync (`fsync`) mode is expensive and slow. However, Ignite provides this mode for higher data consistency.



## Info

The `fsync()` system calls basically means *do not return the IO call until the data has been written to disk or stable media*.

Each update is written to a memory buffer named WAL buffer before written to the WAL file for getting better performance. Each WAL record will be serialized to this memory buffer

---

<sup>51</sup>[https://en.wikipedia.org/wiki/Sync\\_\(Unix\)](https://en.wikipedia.org/wiki/Sync_(Unix))

before written to the WAL file on disk. WAL buffer size is equal to the WAL segment size if the memory mapped file is enabled by default. So, WAL buffer size=64 MB by default. Due to performance reasons, memory mapped file is enabled by default, and it can be turned off by using `IGNITE_WAL_MMAP` system property.

So, Ignite storage engine provides many different WAL write modes with various consistency guarantees, ranging from very strong consistency with no data loss to no consistency and potential data loss.

<b>WAL mode</b>	<b>Description</b>	<b>Implementation</b>	<b>Level of consistency guarantees</b>
FSYNC	Every individual atomic write or transactional commit will be written to WAL file.	fsync() on each commit operation	The highest level of data consistency for each node. Updates are never lost. Can survive OS/JVM process crash, even power failure.
LOG_ONLY	WAL default mode since 2.3.0 version. When a control is returned from the transaction commit operation, the changes are guaranteed to be forced to the OS buffer cache. It's up to the OS to decide when to flush its caches to disk.	File.write() on commit. The data synchronization to disk depends on OS	These writes survive process crash but no OS fail.

<b>WAL mode</b>	<b>Description</b>	<b>Implementation</b>	<b>Level of consistency guarantees</b>
BACKGROUND	When IGNITE_WAL_MMAP property is enabled (default), this mode behaves like LOG_ONLY mode. If the memory-mapped file approach is disabled (IGNITE_WAL_-MMAP=false) then the changes stay in node's internal buffer. This mode does not force application's buffer flush. The frequency of flushing the buffer to disk is defined via the DataStorageConfiguration.setWalFlushFrequency parameter.	Do nothing on commit.	Process crash may cause of several latest updates.
NONE	WAL is disabled. The changes are persisted only during the checkpointing process or graceful node shutdown. Use Ignite#active(false) to shut down the node gracefully.	WAL is disabled	Data is persisted only in case of graceful node shutdown

In most cases, the default mode (*LOG\_ONLY*) is the fastest and preferable option. However, you might need some benchmarking in your real environment (hardware and software) with your specific task before choosing any of the above WAL modes.

As mentioned earlier that, each update is appended to the WAL before flashing to the disk. Each update operation is uniquely identified with a Cache ID and Entry key. Therefore, the cluster can always be recovered to the latest successfully committed transaction or atomic update in the event of a crash or restart.

The WAL stores both logical and physical records. The logical record described the transaction behavior, and on the other hand physical record contains the information about page

snapshots. The WAL logical record structured is as follows:

1. Transactional record (TxRecord) – stores information on the transaction (begin prepare, prepared, commit, rollback).
2. Operation description (DataRecord) – stores information on the operation type (create, update, delete) that are intended to do and (Key, Value, Version).
3. Checkpoint record (CheckpointRecord) – stores begin checkpointing information.

Structure of the *DataRecord* is illustrated in the figure 4.49.

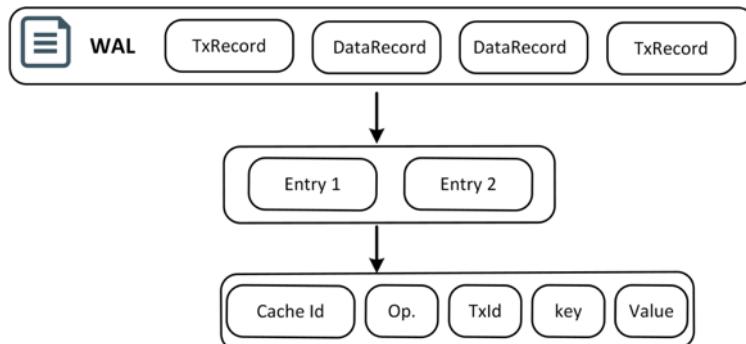


Figure 4.49

As we can see in figure 4.49, a DataRecord is consist of a list of entry operations (e.g., Entry 1, Entry 2). Each entry operation includes:

1. Cache Id
2. Operation type
3. Transaction Id
4. Key
5. And value

Operation type can be one of the following:

1. *Create()*. Put a new cache entry, contains (Key, Value).
2. *Update()*. Update any existing key, contains (key, Value).
3. *Delete()*. Delete or remove an entry, contains only Key.

Update and Create operation always contain the Value. If there are several updates for the same key within a transaction, these updates are merged into a single update using the latest value. This means that the latest value will be stored in the WAL.

The WAL physical records structure is as follows:

1. *Full page snapshot.* A record is logged when page state changes from clean to dirty state (PageSnapshot).
2. *Delta record.* Describes memory region change, page changes. The record contains the byte changed in the page, for instance, bytes 5-10 were changed to data page1.

Page snapshot and related deltas are combined during data recovery. The structure of the physical record is graphically illustrated in Figure 4.50.

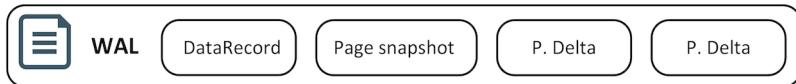


Figure 4.50

All the contents stored into the WAL file is in binary format and not human readable. Ignite provides a console utility for the developers, which allows to check WAL content and perform an analysis. This utility outputs all WAL records in human readable format to an output stream and could be redirected to a file.



## Info

*Eduard Shangareev* is the original creator of the WAL converter utility, and it's shipped with the Ignite source code.

The utility has the following characteristics:

1. It doesn't require Ignite node to be up and running.
2. Provides capability to an offline analysis of copied WAL files.
3. It also converts WAL archive segment contents to a human-readable format.

An example of the WAL record output is illustrated in figure 4.51.

```

super = [WALRecord [size=4129, chainSize=0, pos=FileWALPointer [idx=0, fileOff=1339310, len=4129], type=PAGE
[W] MetaPageUpdatePartitionDataRecord [updateCntr=1, globalRmvid=153243728144800000, partSize=1, state=1, a
[W] CheckpointRecord [cpId=05aa7b50-0faa-49db-afe5-b8cabc22ddc, end=false, cpMark=null, super=WALRecord [si
[W] DataRecord [writeEntries=[UnwrapDataEntry[k = 1, v = [ Hello Worlds: 1], super = [DataEntry [cacheId=-12
[W] PageSnapshot [fullPageId = FullPageId [pageId=000100010000006, effectivePageId=000000010000006, grpId=

```

Figure 4.51

First, we have inserted a few entries with value *Hello World* into a cache with Ignite native persistence enable. Then convert the WAL files through the utility and dumps it to file. Output consist of lines, every new record discovered in the WAL log converted into a new line. Record structure is straightforward as follows:

(Record source) (record string representation)  
 Record source - W - Work directory, A - Archive directory



## Tip

A pre-compiled WAL utility and converted WAL log dump file already uploaded to the book [GitHub repository](#)<sup>52</sup>.

The utility created for dealing with some issues in how Ignite deals partition size. However, this utility could be handy for debugging some weird things with recovery mechanism.

So far in this section, we have talked a lot about how the database changes are appended into a WAL log file. In the event of cluster failure, recovery could take a long time as the WAL file may be quite large. Ignite uses *checkpointing* to solve this issue.

## Checkpointing

Generally, moving the transactions back into the database is called a *Checkpointing* in the case of a cluster failure. Let's assume that the database crashed and it needs to perform recovery. The most straightforward approach would be replaying the whole WAL from the beginning to the end. We should get a complete and correct database at the end of the process. The main disadvantage of this process is, of course, the need to keep and replay the whole WAL. In a distributed database WAL file might be more significant than 1 TB, and it may take a considerable amount of time to recover the cluster by going through the file from the begging to the end.

<sup>52</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/tools/WAL-Reader>

However, all in-memory pages (updated) up to that position are flushed to disk if the storage engine could make a guarantee that all changes for a given WAL position. Then it could determine the location during the recovery process, and replay only the remaining part of the WAL file. In such a way it could significantly reduce the recovery time.

After updates were appended to the WAL file, the dirty pages (updated) in memory still need to be copied to the appropriate partition file. In Ignite, *Checkpointing* is the process of copying dirty pages from RAM to partition files into disk. The benefits of checkpointing are that pages on disk are kept up-to-date, and the WAL archive can be trimmed as old segments are removed.

Let's see the figure 4.52 as a checkpointing process at work.

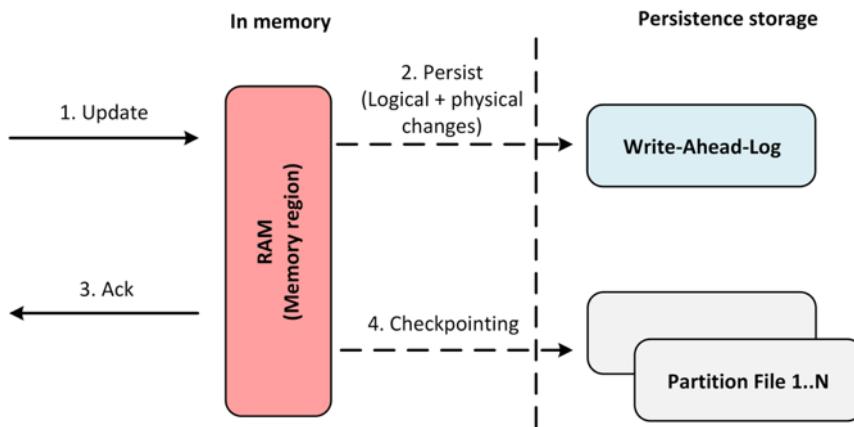


Figure 4.52

The entire message flows are as follows:

1. A cluster node receives an update request (1. Update). The node looks up the data page in RAM where the value is to be inserted or updated. The page is updated and marked as dirty.
2. The update (logical and physical changes) is appended to the end of the WAL (2. Persist).
3. The node sends an acknowledgment to the initiator of the update, confirming that the operation was successful (3. Ack).
4. Periodically, checkpointing is triggered (4. Checkpointing). The frequency of the checkpointing process is configurable. Dirty pages are copied from RAM to disk and passed on to specific partition files.

Theoretically, there are two approaches to implementation of checkpointing:

1. *Sharp checkpointing*. It is accomplished by flushing all the modified or dirty pages for a committed transaction to a particular partition file. If the checkpoint is completed, all data structure on the disk is considered consistency.
2. *Fuzzy checkpointing*. It flushes pages as time passes until it has flushed all pages that a sharp checkpoint would have done. The pages it flushed might not all be consistent with each other as of a single point in time, which is why it's called *fuzzy*.



## Tip

In Ignite *Sharp checkpointing* is used by default. Fuzzy checkpointing is planned for the future release. See the book [Gray and Reuter's classic text on transaction processing<sup>53</sup>](#) for more information about checkpointing.

Ignite uses a read-write lock to achieve the checkpoint consistency. Ignite holds the read lock whenever the cache updates. In a parallel process of writing dirty pages to disk, some threads may want to update data in the pages being written. In such cases, checkpoint pool is used for pages during updated in parallel. See the Apache Ignite wiki for more technical details.

## Baseline topology

Ignite *Baseline Topology* or BLT represents a set of server nodes in the cluster that persists data on disk.

$$\text{Baseline topology} = \{N1, N2, N5\} \subseteq \{N1, N2, N3, N4, N5, N6\}$$

Where,

- N1-2 and N5 server nodes are the member of the Ignite cluster with native persistence enable that persists data on disk.
- N3-4, N6 server nodes are the member of the Ignite cluster but not a part of the baseline topology.

---

<sup>53</sup><https://www.amazon.com/gp/product/1558601902/?tag=xaprb-20>

The nodes from the baseline topology are a regular server node, that store's data in memory and on the disk, and also participate in distributed computing. Ignite cluster can have different nodes that are not a part of the baseline topology such as:

- Server nodes that are not used by the Ignite native persistence to persist data on disk. Usually, they store data in memory or persists data to a 3<sup>rd</sup> party database or NoSQL. In the above equitation, node N3 or N4 might be one of them.
- Client nodes that are not stored shared data.

Let's start at the beginning and try to understand its goal and which problem it's solved to clear the baseline topology concept.

The database like Ignite is designed to support massive data storage and processing. Ignite database are highly scalable and fault-tolerant. This high scalability feature of the Ignite brings a few challenges for the database administrator, such as:

- How to manage a cluster?
- How to add/remove nodes correctly? or
- how to rebalance data after add/remove nodes?

Ignite cluster with a multitude of nodes can significantly increase the complexity of the data infrastructure. Let's look at it by the example of Apache Ignite. Ignite in-memory *mode* cluster concept is very simple. There are no master or dedicated node in the cluster, and every node is equal. Each node stores a subset of data and can be participated in distributed computing or deploy any services. In case of any node failures, client requests served by the other nodes, and the data of the failed nodes will be no longer available. In this mode, Ignite cluster management operations are very similar as follows:

1. To run a cluster, start all nodes.
2. To expand the cluster topology, add some nodes.
3. To reduce the cluster topology, remove some nodes.

Data redistributes between nodes automatically. Data partitions moves from one node to another depending on the backup copy configuration of the caches.

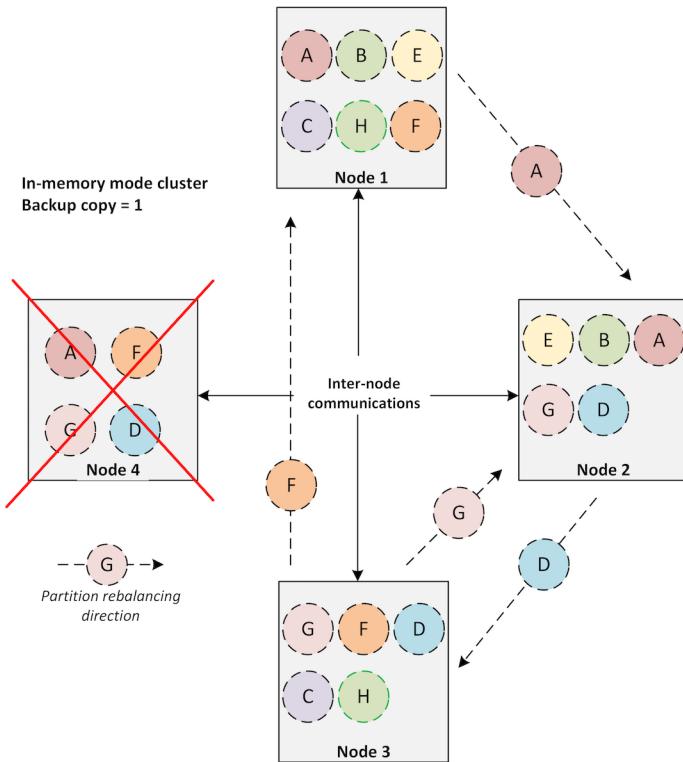


Figure 4.53

In the persistence mode, the node keeps their state even after the restart. Data is read from the disk and restores the node state during any read operation. Therefore, restart of a node in persistence mode does not need to redistribute data from one node to another unlike in-memory mode. The data during node failure will be restored from the disk. This strategy opens up the opportunities to not only prevent moving a massive amount of data during node failure but also reduce the startup times of the entire cluster after a restart. So, we need to distinguish somehow these nodes that can save their state after restart. In other words, the Ignite baseline topology provides this capability.

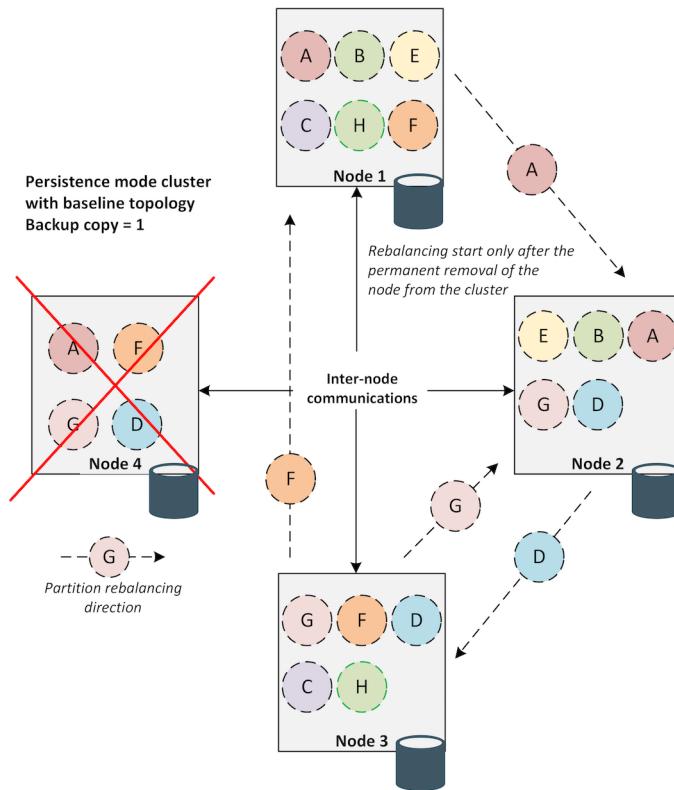


Figure 4.54

In a nutshell, Ignite baseline topology is a collection of nodes that have been configured for storing persistence data on disk. Baseline topology tracks the history of the topology changes and prevents data discrepancies in the cluster during recovery. Let's resume the goals of the baseline topology:

1. Avoid redundant data rebalancing if a node is being rebooted.
2. Automatically activate a cluster once all the nodes of the baseline topology have joined after a cluster restart.
3. Prevent the data inconsistencies in the case of split-brain.

Please note that, you can use persistence caches with the in-memory caches at the same time. In-memory caches will live same as before: consider all nodes are equals and begin redistribution of the partitions whenever a node goes down. Baseline topology will take action only on the persistence caches. Hence, Ignite baseline topology has the following characteristics:

1. Baseline topology defines a list of nodes which intended for storing data, and does not affect other functionalities such as data grid, compute grid etc. If a new node joined to the cluster where baseline topology is already defined, the data partitions is not started moving to the new node until the node is added to the baseline topology manually.
2. On each node, persistence Meta-data repository is used to store the history of the baseline topology.
3. For a newly created cluster (or cluster without baseline topology), a baseline topology is created for the first time during the first activation of the cluster. The administrator must explicitly do all the future changes (add/remove nodes) of the baseline topology.
4. If baseline topology is defined for a cluster, after restarting the cluster, the cluster will be activated automatically whenever all the nodes from the baseline topology are connected.

Now, let's details how Ignite storage engine achieves the abovementioned goals.

## Automatic cluster activation

A cluster can make on its own decision to activate the cluster in the persistence mode with baseline topology. After the first activation of the cluster, the first baseline topology is created and saved on the disk, which contains information about all nodes present in the cluster at the time of activation. Each node checks the status of the other nodes within the baseline topology after the cluster is rebooted. The cluster is activated automatically once all the nodes are online. This time the database administrator needs no manual intervention to activate the cluster.

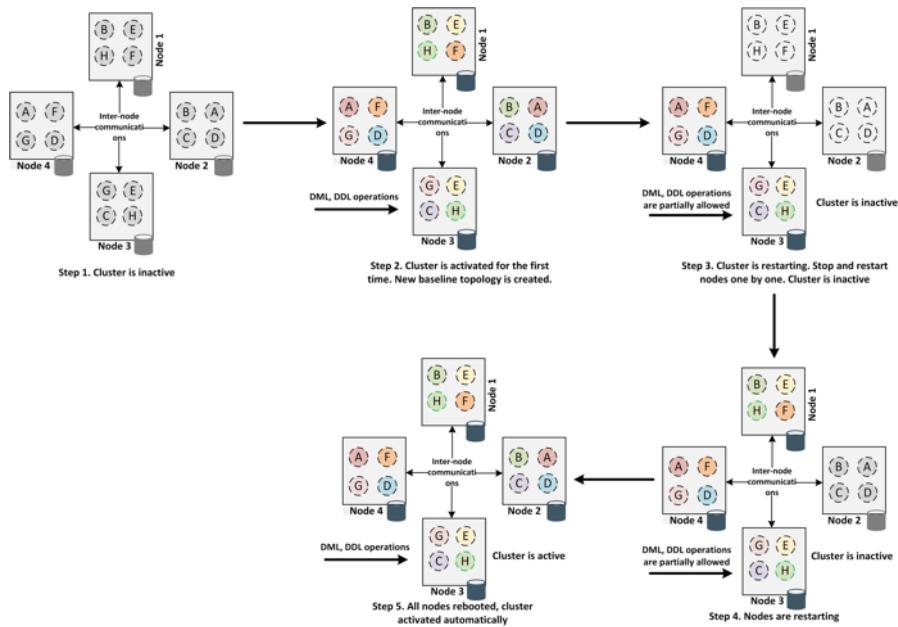


Figure 4.55

Let's go through the details of the automatic cluster activation when **Ignite persistence** is enabled:

- Step 1. All nodes started. The cluster is inactive state and can't handle any DDL/DML operations (SQL, Key-value API).
- Step 2. The cluster is activated by the database administrator manually. First baseline topology is created, added all the currently running server nodes to the baseline topology.
- Step 3. Database administrator decided to restart the entire cluster to perform any software or hardware upgrade. Administrator stopped or restarted each node one by one.
- Step 4. Nodes are started back one by one and joined to the cluster.
- Step 5. Once all the nodes of baseline topology booted, the cluster gets activated automatically.

Although, Apache Ignite is a horizontally scalable database and nodes can be added and removed from the cluster dynamically, baseline topology proceeds from the concept that, in persistence mode the user maintains a stable cluster in production.

## Split-brain protection

Split-brain<sup>54</sup> is one of the common problems of distributed systems, in which a cluster of nodes gets divided into smaller clusters of equal or nonequal numbers of nodes, each of which believes it is only the active cluster. Commonly, the split-brain situation is created during network interruption or cluster reformation. The cluster reforms itself with the available nodes when one or more node fails in a cluster. Sometimes instead of forming a single cluster, multiple mini clusters with an equal or nonequal of nodes may be formed during this reformation. Moreover, these mini cluster starts handling request from the application, which makes the data inconsistency or corrupted. How it may happen is illustrated in figure 4.56. Here's how it works in more details.

- Step 1. All nodes started. The cluster is inactive state and can't handle any DDL/DML operations (SQL, Key-value API).
- Step 2. The cluster is activated by the database administrator manually. First baseline topology is created, added all the currently running server nodes to the baseline topology.
- Step 3. Now let's say, a network interruption has occurred. Database administrator manually split the entire cluster into two different clusters: cluster A and cluster B. Activated the cluster A with a new baseline topology.
- Step 4. Database administrator activated the cluster B with a new baseline topology.
- Step 5-6. Cluster A and B are started getting updates from the application.
- Step 7. After a while, the administrator resolved the network problem and decided to merge the two different cluster into a single cluster. In this time baseline topology of the cluster A will reject the merge, and an exception will occur as follows:

Listing 4.15

---

```
class org.apache.ignite.spi.IgniteSpiException: BaselineTopology of joining node (4,3) is \
not compatible with BaselineTopology in the cluster. Branching history of cluster BIT ([\
11, 9]) doesn't contain branching point hash of joining node BIT (3). Consider cleaning p\
ersistent storage of the node and adding it to the cluster again.
```

---

The nodes of the cluster B will store their data during node startup when Ignite works in persistence mode. The data of the cluster B will be available as we started the cluster B again. So, different nodes may have different values for the same key after the cluster is restored to its primary state. Protection from this situation is one the task of baseline topology.

<sup>54</sup>[https://en.wikipedia.org/wiki/Split-brain\\_\(computing\)](https://en.wikipedia.org/wiki/Split-brain_(computing))

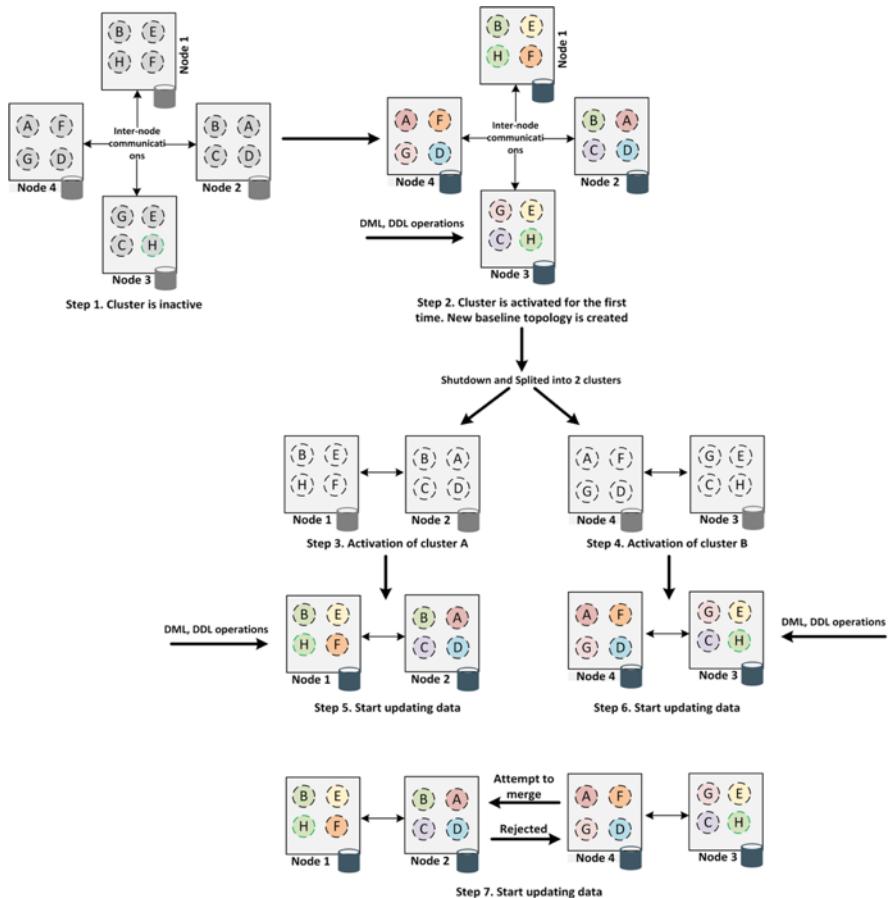


Figure 4.56

As stated earlier, a new baseline topology is created and saved on the disk, which contains information about all the nodes present in the cluster at the moment of activation when we activate the cluster first time. This information also includes a hash value based on the identifiers of the online nodes. If some nodes are missing in the topology during subsequent activation (for instance, the cluster was rebooted, and one node was removed permanently for disk outage), the hash value is recalculated for each node, and the previous value is stored in the activation history within the same baseline topology. Such a way, baseline topology supports a chain of hashes describing the cluster structure at the time of each activation.

In steps 3 and 4, the administrator manually activated the two incomplete cluster, and each baseline topology recalculated and updated the hash locally with a new hash. All nodes of each cluster will be able to calculate the same hashes, but they will be different in various

groups. Cluster A determined that nodes of the cluster B is activated independently of the node of the cluster A, and access was denied when the administrator tried to merge the two cluster into one. The logic is as follows:

**Listing 4.16**

---

```
if (!olderBaselineHistory.contains(newerBaselineHash))
    <join is rejected>
```

---



## Warning

Please note that this validation does not provide full protection against split-brain conflicts. However, it protects against conflicts in case of administrative errors.

## Fast rebalancing and its pitfalls

As described above, the rebalancing event occurs, and data starts moving between the nodes within the baseline topology whenever a new node joins or removes from the baseline topology explicitly by the database administrator. Generally, rebalancing is a time-consuming process, and the process can take quite a while depending on the amount of the data. In this section, we are going into details on the rebalancing process and its pitfalls.

First, let's dig deeper and try to estimate how much time it takes to rebalance the data. In the following table, we compared the performance time of the rebalancing data between in-memory and persistence mode. I used the following configuration for estimating the rebalancing time:

- Total amount of data: 4 GB
- Number of nodes: 3
- Off-heap memory per node: 2 GB
- Backup copy: 1
- Workstation: 4 core CPU MacBook pro, 16 GB of RAM

Actions	In-memory mode, rebalancing times in seconds	Native persistence mode (baseline topology), rebalancing times in seconds
Node restart (within 2 minutes)	247 sec.	51 sec*.
Node removal (total 2)	239 sec.	317 sec.
Add a new node (total 3)	234 sec.	322 sec.

You noticed that the baseline topology performs well when the downtime of the node is short from the preceding table. Rest of the times it takes more than 5 minutes for rebalancing data while adding or removing any node.

In the case of in-memory mode, the dataset that belongs to the departed node rebalanced between the other two nodes. The same thing happens whenever you add or remove a node permanently from the cluster. Data in memory is not persisted, restart of any nodes triggers data rebalancing between nodes automatically.



## Tip

The results from the preceding table may differ depending on your server configurations and the overall load of the cluster. In my case, I do not have any network overhead because all the interaction went through the local Ethernet interface.

The Estimated time of the Ignite native persistence with baseline topology became quite interesting unlike in-memory mode. The modified partitioned from other nodes moved to the restarted node within 51 seconds when the node restarted with a short downtime. Rest of the times rebalancing estimated time was longer than the in-memory mode. Why did it happen?

Firstly, the Ignite native persistence has one overhead: Write-Ahead-log (WAL). Every database modification must be written to the WAL file before they are applied. Written to the WAL means disk I/O. Disk I/O is an expensive operation for massive streams. Therefore, the performance of the rebalancing with baseline topology is much longer than others.

Secondly, The Ignite storage engine does not trigger the rebalancing process when a node restarted with a short downtime. In such a scenario, the Ignite storage engine handle the process in the other way. The Ignite storage engine doesn't copy the modified partitions from other nodes to the restarted node if during the node's downtime some data was updated. Instead of copying the partitions, it copies the *delta WAL file* from other nodes to the restarted node. Next, Ignite storage engine *replay* the delta WAL files and updates the data.

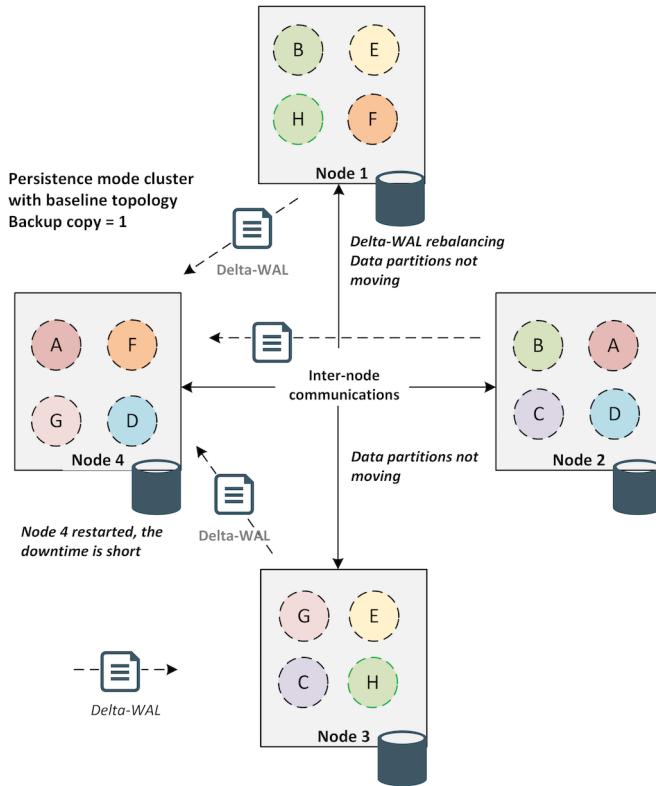


Figure 4.57

Sometimes, this process is also known as a *delta-WAL* rebalancing. These techniques seriously improve the data consistency and the overall performance of the Ignite cluster. Because, node failure is often transient due to the network outage for a short period of time in a large multi-node Ignite cluster. When a node comes back online after a short outage, it may have missed writes for the data it maintains. Delta-WAL rebalancing exists to recover missed data by copying the part (delta) of the WAL files from the other nodes.

As seen above, the rebalancing process puts a load on your cluster as your cluster resources are used to stream data to new nodes. This load reduces the read/write performance of the entire cluster, which slows down your application. The key to rebalancing is to finish as quick as possible. However, this is increasingly difficult as the cluster is under stress. Let's summarize what we have learned so far about the rebalancing:

1. Rebalancing is a resource consuming process, the key to rebalancing is to finish the process as quick as possible.

2. In persistence mode, if a node is restarted with a short downtime, Ignite storage engine doesn't trigger rebalancing process.
3. In persistence mode, if a node restarted with a short downtime, Ignite storage engine copies delta-WAL files from the others node.
4. In persistence mode, rebalancing process takes longer time than in-memory mode due to the WAL file.

## Tool to control baseline topology

Out-of-the-box, the Apache Ignite, provides three different ways to control the baseline topology. These three tools include Java API provides almost the same functionalities for managing baseline topology. Note that, the baseline topology is not set when the cluster with Ignite persistence enable. It should be activated manually for the first time. So, you can set the topology from the Java code, 3<sup>rd</sup> party tools like GridGain web console or command line tool, and let Ignite handle the automatic cluster routines going forward once all the nodes that should belong to the baseline topology are up and running. Let's have a closer look at the tools and their functionalities in the following table.

Tool	Description	Functionalities
Command line tool <code>control.sh bat</code>	Bash or batch script file located in the <code>\$IGNITE_HOME/bin</code> directory, that act like a command line tool to activate or deactivate the cluster.	The tool provides the following functionalities: 1. activate – Activate the cluster 2. deactivate – Deactivate the cluster 3. baseline add – add nodes to the baseline topology 4. baseline set – Set the baseline topology 5. baseline version – Set the baseline topology based on the version

Tool	Description	Functionalities
Java API	Set of Java API's for managing baseline topology from code.	The API provides the following core functionalities: IgniteCluster cluster = ignite.cluster(); 1. cluster.activate(true) - Activate the cluster. 2. cluster.setBaselineTopology(topVer) - Set the baseline topology that is represented by these nodes. 3. cluster.setBaselineTopology(baselineNodes) - Set the baseline topology to a specific Ignite cluster topology version.
GridGain Web console	An interactive configuration wizard, management and monitoring tool.	You can activate/deactivate a cluster from the web console. Once the cluster is active you can add nodes to the baseline topology.

## Automate the rebalancing

Let's assume that, you have 4 nodes cluster with the native persistence enable. Nodes are up and running on low-end servers or virtual machines. One of the nodes went down and did not reboot at the holiday. Its portion of the dataset became unavailable. All the other nodes continue serving their data and also stores newly updated data for the unavailable node. So, if the downtime is expected for a long time, then it's reasonable to remove the node from the baseline topology and trigger rebalancing data to avoid possible data loss. Therefore, at some point, you can use the Ignite *BaselineWatcher* Java application to trigger the rebalancing automatically if you want to automate the rebalancing process without any user intervention.

You can create an event listener to the cluster in *BaselineWatcher* Java application, and subscribe to the topology changes. If such an event occurs (for instance, any nodes went down for 10 minutes), the baseline topology changes on the current set of the server nodes. The data will be rebalanced automatically whenever the topology changes. You can add your own logic to this *BaselineWatcher* application, for instance, analyzing backup number or add any timeouts. See the Ignite documentation for more about BaselineWatcher. Note that in the near future, in Apache Ignite there will be a new feature named *Baseline topology changes* to manage the baseline topology automatically.

## Discovery and communication mechanisms

Any distributed system that scales linearly has at least two mechanisms: one for *communication* with each other, and another to *discover* other nodes in the cluster. These two mechanisms or techniques are the backbone of any cluster that scale-out horizontally when needed. Also, they are responsible for forming the cluster, adding new nodes, handling failures or passing messages to the nodes. In this section, we walkthrough the Ignite's discovery and communication SPI to figure out how it works.

### Discovery

Service discovery is an integral part of almost all the new elastically scaling software system. While modern runtime environments become more dynamic, the discovery of members becomes more complicated. Nodes of the cluster can join, or get restarted at any time. Also, the cluster has to adapt and discover new nodes automatically. Generally, Ignite discovery mechanism provides the following features:

1. Connect a new node to the cluster topology.
2. Disconnect a node from the cluster.
3. Maintains the order in which nodes connected/disconnected to or from the cluster.
4. Ability to send user messages through the cluster.
5. Ability to set an authenticator that will validate the connected nodes.

In addition to Ignite discovery, Ignite is transferring binary and object marshaller information, exchanging Meta-data information, sending tasks to grid service provider and transmit requests for changing the baseline topology. Ignite provides *DiscoverySpi* Java interface that allows discovering remote nodes in the cluster. Moreover, Ignite provides two specific DiscoverySpi implementations: **TcpDiscoverySpi** and **ZookeeperDiscoverySpi** for different scenarios (see the class diagram in figure 5.58).

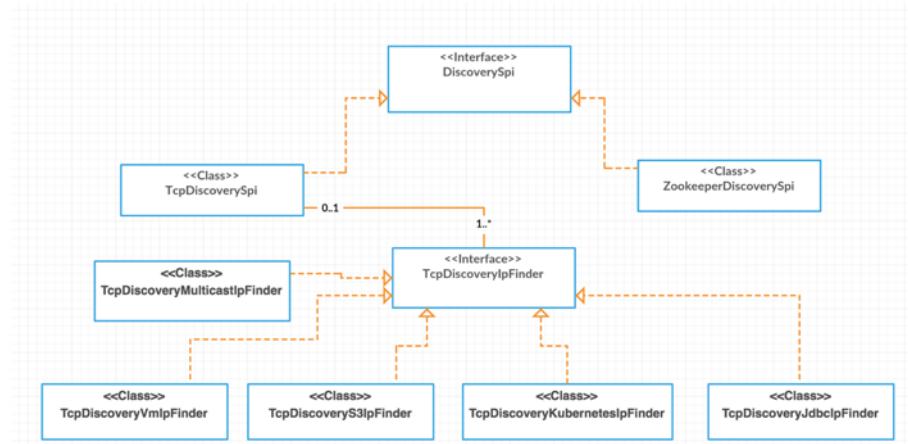


Figure 4.58

## TCP/IP discovery

`TcpDiscoverySpi`<sup>55</sup> is the default implementation of the Ignite DiscoverySpi interface that allows all nodes (with enabling multicast) to discover each other inside the same network. This specific implementation uses *TCP/IP* for node discovery, designed, and optimized for 10s and 100-300 of nodes deployment. When using this implementation Ignite forms a ring-shaped topology. So, almost all network exchanges (except few cases) is done through it.

`TcpDiscoverySpi` uses IP finder (`TcpDiscoveryIpFinder`) to share and store information about nodes IP addresses. At startup, `TcpDiscoverySpi` tries to send messages to random IP address taken from the `TcpDiscoveryIpFinder` about self-start.



### Tip

`TcpDiscoverySpi` starts in client mode as well if the node is configured as a client node. In this case, the node does not take its place in the ring, but it connects to a random node in the ring (IP address is taken from IP finder configured) and uses it as a router for discovery traffic.

<sup>55</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/spi/discovery/tcp/TcpDiscoverySpi.html>

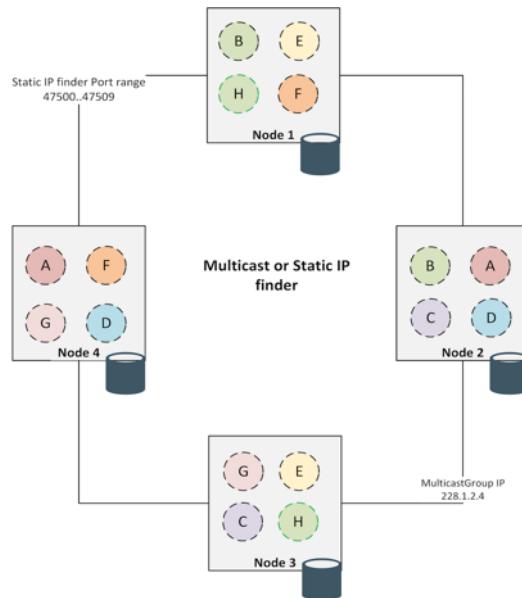


Figure 4.59

In most cases, `TcpDiscoveryIpFinder` can be configured for *Multicast* and *Static IP* based node discovery. Multicast is the default IP finder uses for the node discovery. Let's take a look at the following table for understating the pros and cons of these two approaches.

IP finder	Class name	Description
Multi cast IP finder	<code>TcpDiscoveryMulticastIpFinder</code>	When discovery SPI is configured as a multicast group, other nodes detect automatically inside the same network. When TCP discovery starts with this finder, it sends a multicast request and waits for some time when others nodes reply to this request with messages containing their addresses. This IP finder is the best option for the local or small home office network. If a multicastGroup property is not set, the default value 228.1.2.4 is used.

IP finder	Class name	Description
Static IP finder	TcpDiscoveryVmIpFinder	IP Finder which works only with a pre-configured list of IP addresses specified via the setAddresses method. By default, this IP finder is not shared, which means that all grid nodes have to be configured with the same list of IP addresses when this IP finder is used. It is the best option whenever the multicast is disabled, or you do not have any nodes with dynamic IP address.

TcpDiscoverySpi uses local port range to discover the nodes by default. The default local port range is 100. It is not necessary to open the entire range of discovery port in the range from 47500 to 47600 in each member of the Ignite cluster. TcpDiscoverySpi have localPortRange configuration property that allows customizing the range. For instance, you will have only 3 possible ports instead of 100 if you set localPortRange=3.



## Tip

Multicast IP address is not the same as localhost or 127.0.0.1. Usually, multicast IP address is the IP address in the range of 224.0.0.0..239.255.255.255. Multicasting is not enabled by default in some older version of MacOS. Please follow this [blog post](#)<sup>56</sup> to enable multicast on MacOS.

Note that you are only required to provide at least one IP address of a remote node with the *TcpDiscoveryVmIpFinder*, but usually, it is advisable to provide 2 or 3 addresses of grid nodes that you plan to start at some point of time in the future. Ignite will automatically discover all other grid nodes once a connection to any of the provided IP addresses is established.

The *TcpDiscoveryVmIpFinder* uses in non-shared mode by default. The list of IP addresses should contain an address of the local node as well if you plan to start a server node in this mode. It will let the node not to wait while other nodes join the cluster but instead it becomes the first cluster node and operate usually. Otherwise, you might get into the situation like that, where one node waits for the other nodes while joining to the cluster.

However, you can use the combination of both Multicast and Static IP based discovery together. It is possible to configure the list of static IP address with the multicast group as shown below:

<sup>56</sup><https://blogs.agilefaqs.com/2009/11/08/enabling-multicast-on-your-macos-unix/>

Listing 4.17

---

```
<property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
        <property name="ipFinder">
            <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
                <property name="multicastGroup" value="228.10.10.157" />
                <!-- list of static IP addresses-->
                <property name="addresses">
                    <list>
                        <value>1.2.3.4</value>
                        <value>1.2.3.5:47500..47509</value>
                    </list>
                </property>
            </bean>
        </property>
    </bean>
</property>
```

---

As stated above, TcpDiscoveryVmIpFinder uses IP address in non-shared mode. However, you can use your database or network shared directory as common shared storage of initial IP address. These behaviors are supported by the following TCP discovery IP finder:

- TcpDiscoveryJdbcIpFinder: a JDBC based IP finder, used as common shared storage for the IP address. Each node writes their IP address to the database table on startup and uses for node discovery by the other nodes. The database maintains one table which holds the IP address.
- TcpDiscoverySharedFsIpFinder: a shared file system base IP finder, uses a network shared directory as storage for nodes IP address. Each node writes their IP address to the file system on startup. Note that, to enable discovery over a network, you must provide a path to a shared directory explicitly.

So far, we have introduced only the discovery mechanism over a local network or home office network where the IP address of each node is almost static. However, node discovery on any cloud network or virtual machine environments such as *Amazon AWS* or *Google VMs* is complicated. Because most of the cloud platform does not support multicast, and static IP configuration is not flexible. In cloud platform virtual machine IP address may change from time to time and machine can be added or removed to a cluster dynamically. Ignite provides specific *TcpDiscoveryIpFinder* implementations for each platform to solves this problem.

- **TcpDiscoveryS3IpFinder.** Amazon AWS S3-based IP finder. This IP finder uses [Amazon S3<sup>57</sup>](#) store or [Amazon ELB<sup>58</sup>](#) to support automatic node discovery. Amazon S3-based discovery allows Ignite nodes to register their IP addresses on startup, with the S3 store. This way other nodes can try to connect to any of the IP addresses stored in S3 and initiate automatic grid node discovery. For more details, please follow the [documentation<sup>59</sup>](#) here.
- **TcpDiscoveryGoogleStorageIpFinder.** Google cloud storage-based IP finder. This IP finder uses Google cloud-storage as shared storage to support automatic node discovery. The IP finder creates a bucket with the provided name. On startup, each node registers its IP address in the bucket. This way other nodes can try to connect to any address stored in the bucket and initiate automatic cluster node discovery. For more details, please follow the [documentation<sup>60</sup>](#) here.
- **TcpDiscoveryKubernetesIpFinder.** IP finder for automatic lookup of Ignite nodes running in Kubernetes environment. All Ignite nodes have to be deployed as Kubernetes pods to be discovered. An application that uses Ignite client nodes as a gateway to the cluster is required to be containerized as well. Applications and Ignite nodes running outside of Kubernetes will not be able to reach the containerized counterparts. The implementation is based on a distinct Kubernetes service that has to be created and should be deployed before Ignite nodes startup. The service will maintain a list of all endpoints (internal IP addresses) of all containerized Ignite pods running so far. The name of the service must be equal to `setServiceName(String)` which is `ignite` by default. Please follow the [documentation<sup>61</sup>](#) here for details configuration of the IP finder.



## Warning

Storing data in AWS S3 services or Google cloud storage are not free. Choose another implementation of TcpDiscoveryIpFinder described above for local or home network tests.

The Apache Ignite TCP/IP Discovery SPI has a few major drawbacks. The transmission time of the messages between nodes is directly proportional to the number of the nodes in the cluster. It means that an Ignite cluster with more than 100 nodes may take a few more seconds for a system message to traverse through the cluster. As a result, the basic processing of the

<sup>57</sup><https://aws.amazon.com/s3/>

<sup>58</sup><https://aws.amazon.com/elasticloadbalancing/>

<sup>59</sup><https://apacheignite-mix.readme.io/docs/amazon-aws>

<sup>60</sup><https://apacheignite-mix.readme.io/docs/google-compute-engine>

<sup>61</sup><https://apacheignite-mix.readme.io/docs/kubernetes-discovery>

events such as joining of a new node or detecting a split-brain situation can take a while, which can affect the overall performance of the cluster. Therefore, this implementation is not an optimal solution for a large cluster.

### ZooKeeper discovery

It's proposed to move from the ring topology to the star topology to overcome the above disadvantages, in the center of which the [Zookeeper<sup>62</sup>](#) service is used. At the same time, the Zookeeper cluster appears as the connection and synchronization point for the Ignite cluster. Such an implementation allows scaling Ignite cluster to 100s and 1000s of nodes preserving linear scalability and performance.



### Info

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.

The basic idea behind the discovery service through Zookeeper is that for each node to be able to identify its current state and store that information into the centralized place. Primary usages of such storage are to provide as a minimum, IP and Port number of the node to all interested nodes that might need to communicate with it. This data is often extended with other types of the data such as sequence order of a node that how it connected to the cluster.

---

<sup>62</sup><https://zookeeper.apache.org/>

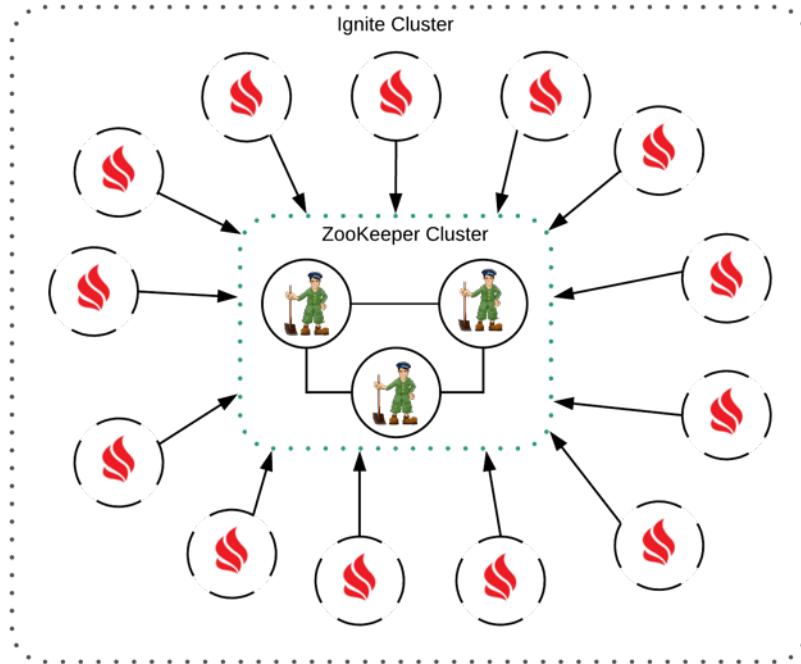


Figure 4.60

So, in this implementation, ZooKeeper cluster is used as primary storage of information for the current topology states. It also stores attributes of the nodes (user-defined attributes), the order that a node connected to the cluster, queues for storing user-defined events. All the messages about topology exchanges through Zookeeper, nodes are not communicated directly with each other.

So, ZooKeeper cluster is used as primary storage of information for the current topology states in this implementation. Also, it stores attributes of the nodes (user-defined attributes), the order that a node connected to the cluster, queues for storing user-defined events. All the messages about topology exchanges through Zookeeper, nodes are not communicated directly with each other.



## Warning

Configuring ZooKeeper service discovery with Ignite cluster needs particular knowledge and some extra efforts.

The main functionality of ZooKeeper is to provide a coordination and synchronization service for the distributed process through a hierarchical system. [ZooKeeper Overview<sup>63</sup>](#) provides an excellent overview of the main concepts. Here are some of the relevant ones:

1. All changes from the nodes will be applied in the order in which the cluster received them.
2. The actions of the ZooKeeper client nodes are transactional, i.e., either all are successful or canceled.
3. All ZooKeeper client behold the same state of the data in the service, regardless of the which ZooKeeper server they are connected to.

Another essential functionality of ZooKeeper is the mechanism of notifying clients about changes. ZooKeeper allows you to store arbitrary, client information directly in the service. The recorded information can be accessed by all ZooKeeper clients once it save.

ZooKeeper also provides opportunities to handle the split-brain scenario and network segmentation. Apache Ignite provides detailed [documentation<sup>64</sup>](#) described the entire process of handling network segmentation and split-brain situation. So, we highly recommended you to go through the documentation if you have an Ignite cluster with more than 100 nodes and need to use ZooKeeper as service discovery.

## Communication

Besides to discover nodes in a cluster, there are still needs for some direct communication between nodes for sending and receiving messages such as task execution, monitoring partition exchanges, etc. Ignite provides a *CommunicationSpi* which is responsible for *peer-to-peer* communication and data exchanges between nodes. Ignite CommunicationSpi is one of the most crucial SPI in Ignite. It is used heavily throughout the system and provides means for all data exchanges between nodes, such as internal implementation details and user-driven messages.

Ignite comes with a built-in [CommunicationSpi<sup>65</sup>](#) implementation: *TcpCommunicationSpi*, which uses TCP/IP protocols and [Java NIO<sup>66</sup>](#) to communicate with other nodes.

<sup>63</sup><https://zookeeper.apache.org/doc/current/zookeeperOver.html>

<sup>64</sup><https://apacheignite.readme.io/docs/zookeeper-discovery#section-failures-and-split-brain-handling>

<sup>65</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/spi/communication/CommunicationSpi.html>

<sup>66</sup><https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html>



## Info

ZooKeeper discovery is an alternative implementation of Ignite discovery SPI that does not affect Ignite Communication SPI.

TcpCommunicationSpi uses IP address and port of the local node attributes to communicate with other nodes. At startup, this SPI tries to start listening to a local port specified by the configuration. SPI will automatically increment the port number until it can successfully bind for listening if the local port is occupied.

TcpCommunicationSpi caches connections to remote nodes, so it does not have to reconnect every time a message is sent. Idle connections are kept active for 10 minutes by default, and they are closed. You can configure the idle connection timeout by the `setIdleConnectionTimeout` parameter.



## Tip

TcpCommunicationSpi is used by default and should be explicitly configured only if some SPI configuration parameters need to be overridden.

# Cluster groups

The design goal of the Apache Ignite is to handle high workloads across multiple nodes within a cluster. Also, by design, all nodes are equal in an Ignite cluster. However, in some cases, you may wish to create a *cache* to some specific nodes within a cluster or deploy a *service* only on particular nodes. In such cases, Ignite provides an easy way to create *logical groups* of nodes within your grid (entire cluster), and also collocate the related data into similar nodes to improve performance and scalability of your application. Creating a logical group of nodes into a cluster called *ClusterGroup* in Ignite. In the next few subsections, we will discover a few essential, and yet unique features of Apache Ignite such as cluster group, data allocation (sometimes calls affinity collocation).

Ignite provides a public interface `clusterGroup` that defines a cluster group of nodes which contains all or a subset of nodes. Cluster group allows to group cluster nodes into various subgroups to perform distributive operations on them. For instance, you can cluster a few nodes together that stores the cache with name *myCache*, or all client nodes that access the cache *myCache*. Moreover, you may wish to deploy a service only on remote nodes.

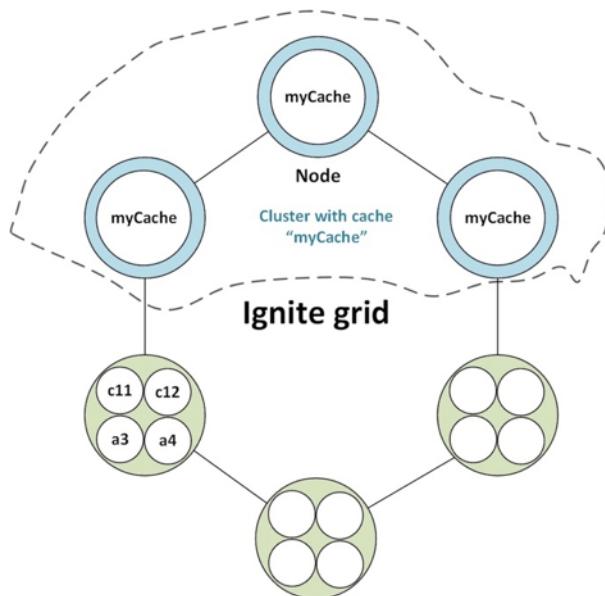


Figure 4.61

Apache Ignite provides the following few ways to create a cluster group into Ignite grid (cluster).

## Predefined cluster group

Ignite provides predefined implementations of *ClusterGroup* interface to create cluster group based on any predicate. Predicates can be Remote Node, Cache Nodes, Node with specified attributes and so on. For instance, `ClusterGroup.forClients()` creates a cluster group of nodes started in client mode. All `ClusterGroup.forXXX()` methods creates a child cluster group from the existing cluster group. If you create a new cluster group from the existing one, then the resulting cluster group will include a subset of nodes from the current one. Here is an example of some predefined cluster groups available in Ignite ClusterGroup interface.

**Listing 4.18**

---

```
Ignite ignite = Ignition.ignite();
IgniteCluster cluster = ignite.cluster();
// Cluster group over remote nodes.
ClusterGroup remoteNodes = cluster.forRemotes();
// Cluster group over all nodes with cache named "testCache" enabled.
ClusterGroup cacheNodes = cluster.forCacheNodes("testCache");
// Cluster group over all nodes that have the user attribute "group" set to the value "worker".
ClusterGroup workerNodes = cluster.forName("group", "worker");
```

---

**Tip**

The `IgniteCluster` interface also extends the `ClusterGroup` which makes an instance of the `IgniteCluster` into a group containing all cluster nodes.

There are a few `ClusterGroup.forXXX()` methods such as `ClusterGroup.forHost()`, `ClusterGroup.forNode()` that helps you to create a cache into particular subset of nodes. First of all, you have to create a cluster group consisting of the nodes running on the specified host. Then gets the predicate from this cluster node by the `ClusterGroup.predicate()` method. Next, you can define node filter for cache configuration using `CacheConfiguration.setNodeFilter(IgnitePredicate<ClusterNode> nodeFilter)` method. A cache will be started on each node where `ClusterNode` instance passed by the `IgnitePredicate<ClusterNode>`. You can also filter a node by node's static attributes or by any other node properties.

## Cluster group with node attributes

Although, every node into the cluster is same, a user can configure nodes to be master or worker and data nodes. All cluster nodes on startup automatically register all environment and system properties as node attributes. However, users can choose to assign their own node attributes through configuration:

**Listing 4.19**

---

```
IgniteConfiguration cfg = new IgniteConfiguration();
Map<String, String> attrs = Collections.singletonMap("ROLE", "master");
cfg.setUserAttributes(attrs);
// Start Ignite node.
Ignite ignite = Ignition.start(cfg);
```

---

After stating the node, you can group the nodes with the attribute master as follows:

**Listing 4.20**

---

```
IgniteCluster cluster = ignite.cluster();
ClusterGroup workerGroup = cluster.forName("ROLE", "master");
Collection<GridNode> workerNodes = workerGroup.nodes();
```

---

## Custom cluster group

Sometimes, it also called dynamic cluster group. You can define dynamic cluster groups based on some predicate; predicates can be based on any metrics such as CPU utilization or free heap space. Such cluster groups will always only include the nodes that pass the predicate. Here is an example of a cluster group over nodes that have less than 256 MB heap memory used.

**Listing 4.21**

---

```
IgniteCluster cluster = ignite.cluster();
// Nodes with less than 256MB heap memory used
ClusterGroup readyNodes = cluster.forName("ROLE", "master");
Collection<GridNode> workerNodes = readyNodes.nodes();
```

---



## Warning

Nodes in the above group will change over time based on their heap memory used.

Every `ClusterGroup.forName()` methods return a subset of the cluster group which allows you to combine cluster groups by nesting them within each other. For example, the following pseudo-code shows how to create a cluster group with the youngest nodes from the current server cluster group (nodes that started in server mode).

**Listing 4.22**

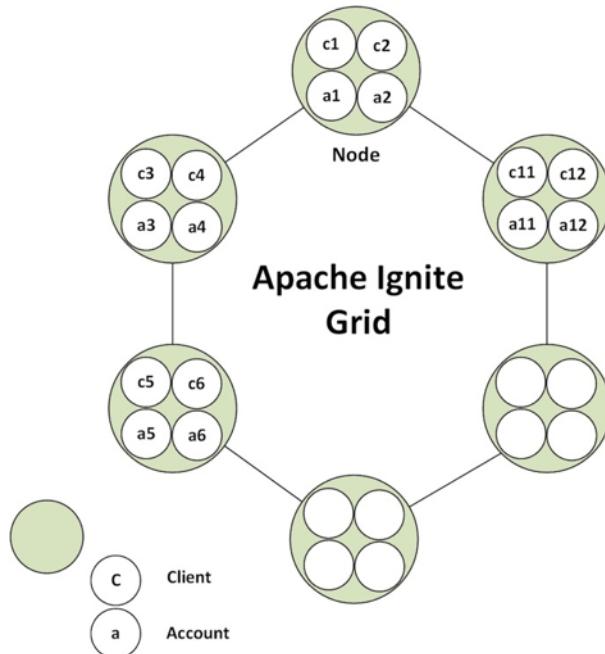

---

```
ClusterGroup youngestGroup = cluster.Servers().forYoungest();
ClusterNode youngestNode = youngestGroup.node();
```

---

## Data collocation

Term data collocation means an *allocation* of the *related* data into the same node. For instance, if we have one cache for *Clients profile* and another cache for *Clients transactions*. We can allocate the same clients profile and its transactions records in the *same* Ignite node. In this approach, network roundtrips for the related data decrease and the user application can get the data from a single node. In this case, multiple caches with the same set of fields are allocated to the same node.



**Figure 4.62**

Figure 4.62 illustrates that the client and its account information are located on the same Ignite host. To achieve that, the *cache key* used to cache Client objects should have a field or method annotated with `@AffinityKeyMapped` annotation, which will provide the value of the *account key* that belongs to that client, like so:

**Listing 4.23**

---

```
Object clientKey1 = new AffinityKey("Client1", "acclId1");
Object clientKey2 = new AffinityKey("Client2", "acclId2 ");
Client c1 = new Client (clientKey1, ...);
Client c2 = new Client (clientKey2, ...);
// Both, the client and the account information objects will be cached on the same node.
cache.put("acclId1", new Account("credit card1"));
cache.put("acclId2", new Account("credit card2"));
cache.put(clientKey1, c1);
cache.put(clientKey2, c2);
```

---

You can use any set of fields to calculate the affinity function, and it is not necessary to use any unique key. For instance, you can use the field *client Id* that owns the *account Id* to calculate the Client account affinity function. We will briefly describe the topics with a complete example in *chapter seven*.

## Compute collocation with data

Apache Ignite also provides the ability to route the data computation unit of work to the nodes where the desired data is cached. This concept is known as *Collocation Of Computations And Data*. It allows routing the whole units of work to a certain node. You should use `IgniteCompute.affinityRun(...)` and `IgniteCompute.affinityCall(...)` methods to collocate computation with data.

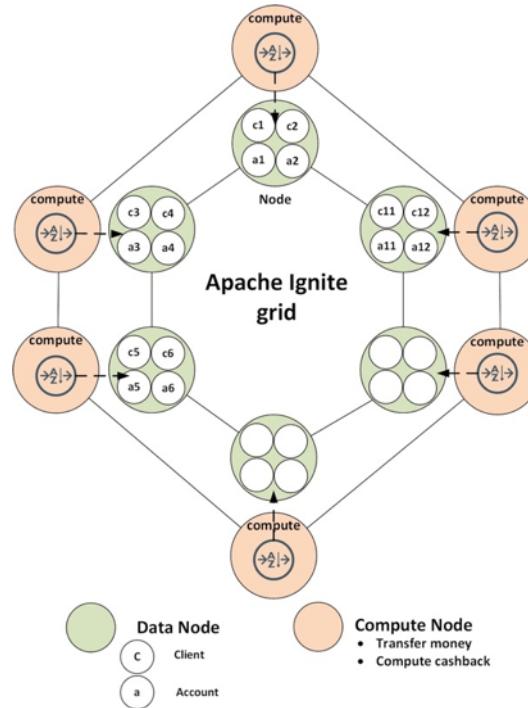


Figure 4.63

Here is how you can collocate your computation in the same cluster node on which the *Client* and its *account information* are allocated.

Listing 4.24

---

```

String acclId = "account1Id";
// Execute Runnable on the node where the key is cached.
ignite.compute().affinityRun("myCache", acclId, () -> {
    Account account = cache.get(acclId);
    Client c1 = cache.get(clientKey1);
    Client c2 = cache.get(clientKey2);
    ...
});
```

---

Here, the computation unit accesses to the *Client* data as local, this approach highly decreases the network roundtrip of the data across the cluster, and increases the performance of data processing.

## Protocols and clients

While working with Apache Ignite, often we came across a few protocols and clients to connect and work with an Apache Ignite cluster. In this section, we will discuss these clients and protocols in terms of performance and optimization. Let's start with the Ignite REST client.

**REST client.** As stated earlier, Apache Ignite provides an HTTP REST client that gives you the ability to communicate with the Ignite cluster over HTTP/HTTPS protocols. REST APIs can be used to perform different cache operations like read/write from/to caches or execute any compute task. It can also be used to execute SQL queries over HTTP.

*Use cases:*

1. Manipulating Ignite caches directly from the web application.

*Drawbacks:*

1. Does not support transactions.
2. Does not provide any failover mechanism.
3. REST client has much overhead of JSON marshaling/unmarshalling objects.
4. Query operation is always slower than the Ignite client node because cache operation always goes through a mediator node (if the data does not reside in the node that the client connected).

**JDBC/ODBC driver.** Ignite provides a JDBC/ODBC driver to connect and run SQL queries over Ignite cluster from the beginning. By using JDBC/ODBC protocol, you can configure your SQL IDE to connect with Ignite cluster and manipulate with the data. Moreover, through JDBC/ODBC driver you can connect any data visualization or data analyzing tool for performing business intelligence.

*Use cases:*

1. Connect any SQL IDE to execute SQL queries.
2. Data analysis over Ignite data.

*Drawbacks:*

1. JDBC driver (thin) is not thread-safe. JDBC objects such as *Connections*, *Statements*, and *ResultSet* are not thread-safe. You must not use all statements and results sets of a single JDBC Connection in multiple threads.
2. Query operation is always slower than the Ignite client node because cache operation always goes through a mediator node.

**Thin client.** Since version 2.4 Ignite provides a *lightweight* client that connects to the cluster via a standard TCP/IP socket connection. Unlike .Net client, it does not start any JVM, does not become a part of the cluster topology, and never holds any data. From the version 2.6 Ignite thin client supports failover operations.

*Use cases:*

1. Manipulating cache operation for platforms that do not directly supported by Ignite such as Rust, Golang, etc.

*Drawbacks:*

1. Does not support transactions (in version 2.6).
2. Query operation is always slower than the Ignite client node, because cache operation always goes through a mediator node.

## Resilience & Automatic failover

Ignite cluster is typically composed of multiple nodes. A node failure can happen anytime in a cluster. It's essential for an Ignite client to continue works with the cluster if an Ignite server node fails or becomes unreachable. Ignite client node can be disconnected from the cluster in several cases:

1. In the case of a network outage.
2. Server node dies or is restarted.
3. The server node can disconnect a slow client.

Out-of-the-box Ignite provides three different types of clients to connect and work with the Ignite cluster. Each of them handles the failover in different ways. We are going to take a look at how different Ignite clients manages the automatic failover in this section.

## Ignite client node

Ignite client node is entirely resilience by nature because whenever connected to the cluster it's become a part of the topology. When a client determines that a node disconnected from the cluster, it tries to re-establish the connection with the server. This time it assigns to a new node ID and tries to reconnect to the cluster. The connection re-establishment process is fully transparent for the client node.



### Tip

The word resilience means the ability of a system to recover quickly and continue its operation when there has been a failure.

## Ignite thin client

Unfortunately, Ignite does not support thin client failover on the server side. Usually, a thin client always connected to one Ignite server node at a time. However, Ignite thin client provides failover by automatically re-connecting to the another server node and retrying an operation if the server that the client is connected goes down. It is the application responsibility to configure the multiple server nodes to enable failover mechanism. Let's have a look at the following configuration.

Listing 4.25

---

```
try (IgniteClient client = Ignition.startClient(
    new IgniteClientConfiguration()
        .setAddresses("127.0.0.1:1080", "127.0.0.1:1081", "127.0.0.1:1082")
)) {
    ...
} catch (IgniteUnavailableException ex) {
    // All the servers are unavailable
}
```

---

With the above configuration, thin client randomly tries all the servers in the list and throws ClientConnectionException exception if all of the servers are unavailable.



## Warning

Thin client failover mechanism is almost transparent to the user code with one pitfall, failover queries might return duplicate entries. If the server that the client is connected goes down while the entries are being extracted, the client retries the query from the beginning from the other server node, which ends up with the duplicate entries in the result set.

## JDBC client

Any JDBC client can use the Ignite JDBC client driver to connect and query against the Ignite cluster which provides full failover support. From version 2.5, Ignite JDBC driver let you set multiple addresses in connection properties which allows reconnecting to another server node, in case of the node (old) failure. The new node takes care of the query distribution and result aggregation. Then the result is sent back to the client application.

## Ignite Rest client

Ignite rest client does not support failover and load balancing between several Ignite remote nodes. However, from the REST client perspective, the application is responsible for handling the failover functionalities. So, you can abstract the request (proxy URL) or handle the failover of the client connection by using 3<sup>rd</sup> party libraries such as [common-http-client-failover](#)<sup>67</sup>. The project implements a wrapper around Apache *commons-http-client* to provides automatic failover. The project is designed to gracefully handle the cases where:

1. Some remote hosts are down or hanged.
2. One of the remote hosts is going down as planned.

Another approach to control failover is using a load balancer such as *Nginx*, *Apache mod\_proxy*. If one of the remote hosts are down, load balancer redirects the request to another host from the list.

---

<sup>67</sup><https://github.com/exalead/commons-HttpClient-failover>

## Multi data center replication

In the modern IT landscape, multi-datacenter replication is one of the primary requirements for any database, including RDBMS and NoSQL. In simple terms, multi-datacenter replication means the replication of data between different data centers. Multiple datacenter replications can have a few scenarios.

### Geographical location scenario:

In this scenario, data should be hosted in different data centers depending on the user location to provide a responsive exchange. In this case, data centers can locate in different geographical locations such as different regions or different countries. Data synchronization is entirely transparent and bi-directional within data centers. The logic that defines which datacenter a user will be connected to resides in the application code.

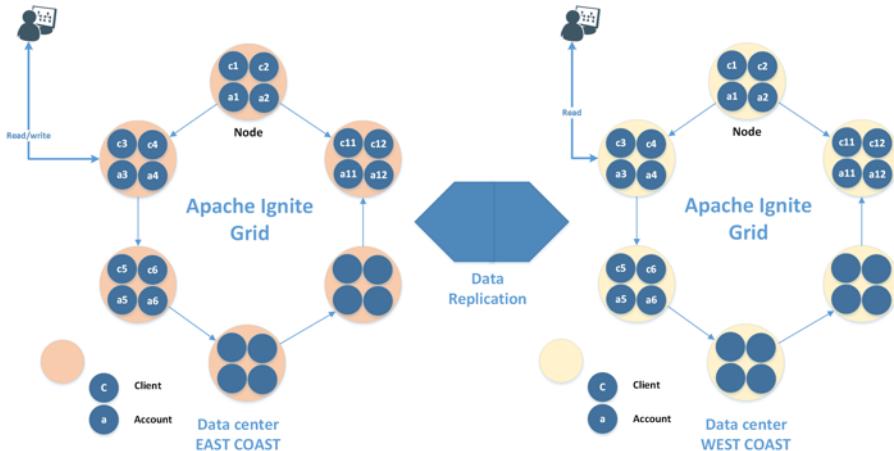


Figure 4.64

### Live backup scenario:

In this scenario, most users use different data center as a live backup that can quickly use as a fallback cluster. This use case is very similar to disaster recovery. Sometimes it's also called *passive replication*. In passive replication, replication occurs in one direction: from master to the replica. Clients can connect to the master database in one data center and perform all the CRUD operation on the database.

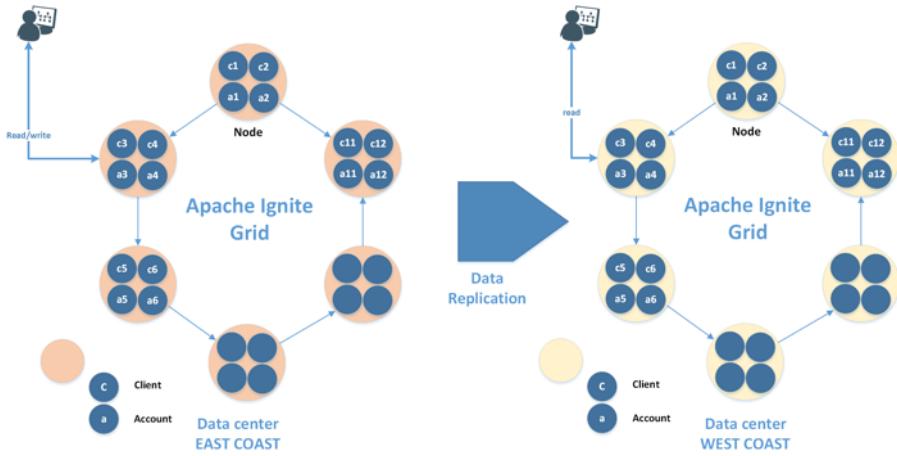


Figure 4.65

To ensure consistency between the two databases, the replica is started as a read-only database, where only transactions replicated from the master can modify the database contents.

Unfortunately, out-of-the-box, Apache Ignite doesn't support multi-datacenter replication. However, you can span Ignite nodes in different data centers (for instance, 10 nodes in one data center and other 10 nodes on another).

Span Ignite grid into multiple data center can introduce a few issues:

- Latency: this is going to hinder performance quite a bit on the server side. If any nodes from different datacenter have a backup copy of your master data, transaction time can be very high. In the long run, you could have a problem with data consistency also.
- Clients connected to different datacenters are going to face very different latency for client operations.

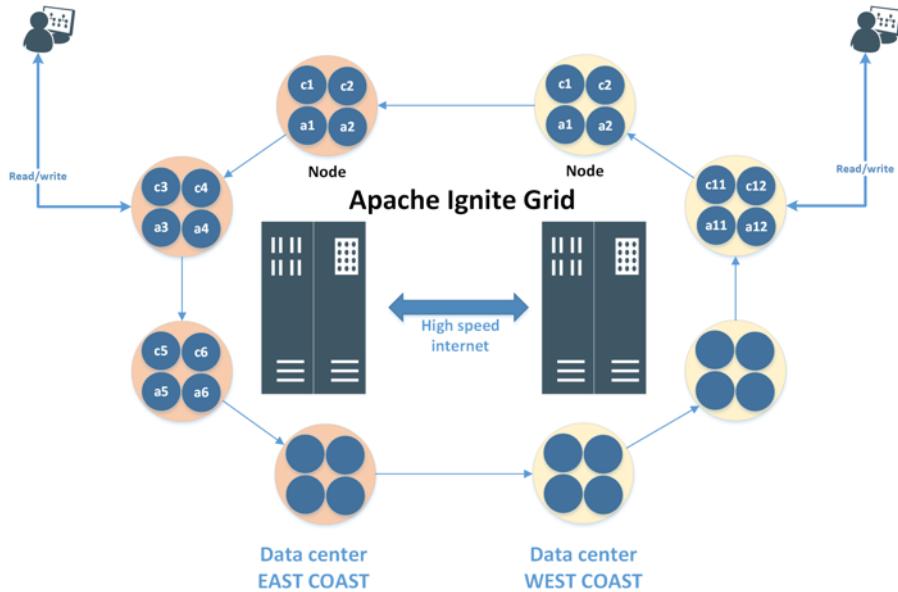


Figure 4.66

However, [GridGain<sup>68</sup>](#)(commercial version of Ignite) provides full functionality Data center replication in different geographical location. It also supports bi-directional, active-active data replication between data centers. You can also use Ignite Kafka streaming module for replicating data from one data center to another. Please, see [chapter 8](#) for a complete example of replicating data between data centers.

## Key API's

Apache Ignite provides a rich set of different APIs to work with the caches and data stored into the Apache Ignite cluster. The APIs are implemented in the form of native libraries for major languages and technologies like Java, .NET and C++. A concise list of dominant APIs of Apache Ignite is described below.

<sup>68</sup><https://docs.gridgain.com/docs/data-center-replication>

Name	Description
org.apache.ignite.Ignition	This class defines a factory for the Ignite API. It controls Ignite cluster life cycle such as start/stop the node and allows listening for cluster events.
org.apache.ignite.configuration.IgniteConfiguration	This class defines cluster runtime configuration. It defines all configuration parameters required to start a cluster instance.
org.apache.ignite.IgniteCache	The main cache interface, the entry point of all data grid APIs. This API extends javax.cache.Cache interface.
org.apache.ignite.cache.store	API for cache persistent storage for read-through and write-through behavior.
org.apache.ignite.cache.query.ScanQuery	Scan query over cache entries.
org.apache.ignite.cache.query.TextQuery	Query for Lucene based full-text search.
org.apache.ignite.cache.query.SqlFieldsQuery	SQL Fields query. This query can return specific fields of data based on SQL clause.
org.apache.ignite.transactions.Transaction	Ignite cache transaction interface, have a default 2PC behavior and supports different isolation levels.
org.apache.ignite.IgniteFileSystem	Ignite file system API, provides a typical file system view on the particular cache. Very similar to HDFS, but only on in-memory.
org.apache.ignite.IgniteDataStreamer	Data streamer is responsible for streaming external data into the cache. This streamer will stream data concurrently by multiple internal threads.
org.apache.ignite.IgniteCompute	Defines compute grid functionality for executing tasks and closures over nodes ClusterGroup.
org.apache.ignite.services.Service	An instance of the grid managed service. Whenever a service is deployed, Ignite will automatically calculate how many instances of this service should be deployed on each node within the cluster.

## Summary

By now, you are familiar with most of all the nuts and bolts of the Apache Ignite. We have discussed the fundamental concepts of the memory-centric database in this chapter: Ignite shared nothing architecture, data partitioning, caching strategies and Ignite data model. We also presented how CAP theorem governs the behavior of such databases. We went through

the Ignite data collocation techniques and briefly explained the Ignite durable memory architecture and Baseline topology concept. We also described the process of discovery and communication mechanisms and provided a comparative analysis of different protocols and clients.

It is understandable that it may be a lot to take in for someone new to NoSQL systems. It is okay if you do not have complete clarity at this point. As you start working with Apache Ignite, experimenting with it, and going through the rest of the book, you will start to come across stuff that you have read in this chapter and it will start to make sense, and perhaps you may want to come back and refer to this chapter to improve clarity.

## What's next?

At this point, we are ready to learn some of the common use cases of using Ignite. In the next chapter, we will look at the Ignite database caching features such as 2<sup>nd</sup> level cache, memoization, web session clustering and so on.

# Chapter 5. Intelligent caching

In computer terminology a cache is a high-speed data storage layer in front of the primary storage location which stores a subset of data so that future requests for that data served up as fast as possible. Primary storage could be any database or a file system that usually stores data on *non-volatile* storage. Caching allows you to reuse previously retrieved or computed data efficiently, and it is one of the secrets of high-scalability and performance of any enterprise level application.

You may wonder why we named the chapter intelligent caching! Because, from the last decades, the unbounded changes of the software architecture need not only correctly used of a caching strategy but also properly configured (cache eviction, expiration) and sizing the cache layer to achieve the maximum performance and high-scalability of an application. Caching can be used for speeding up requests on five main different layers or environments of your application architecture:

1. Client
2. Network
3. Web server
4. Application
5. Database

So, you should consider caching strategies for each layer of your application architecture to accomplish the high-performance of an application, and implements it's correctly. It should be noted that none of the caching platforms or framework are a silver bullet. Cache usages vary for different data sizes and scenarios. Firstly, you should measure the data sizes and requests on each layer, doing various tests to find out the bottleneck and then with the way of experiments you have to define a tool or framework for caching data before implementing any caching platform such as Ignite, Ehcache, Redis or Hazelcast on any application layer.

In this chapter, we want to focus primarily on things you need to know about data caching and demonstrate the use of Apache Ignite for accelerating application performance without changing any business logic code. So, we are going to cover the following topics throughout the entire chapter:

1. Different caching strategies and methods as a smart in-memory caching.

2. Read/Write through and write behind strategies examples based on Hibernate and MyBatis for database caching.
3. Memoization or application level caching.
4. Web session clustering.
5. Moreover, a list of recommendations to correctly prepare the caching layer.

## Smart caching

I often hear suggestion like this when it comes to a matter of performance: *Need for speed - Caching*. However, I believe that in-memory caching is *the last lines of defense* when all current optimization tricks reach a bottleneck. We have a lot of points or places for optimizing before considering a separate layer for caching data, such as:

- Optimizing SQL queries; runs a few SQL queries plans to define the bottleneck on the database level.
- Adding *necessary* indexes on tables.
- Optimizing and configuring connection pools on the application servers.
- Optimizing application code, such as fetching data by paging.
- Caching static data such as Java script, Images and CSS files on the Web server and the client side.

Consider the regular N-Tier JEE architecture for optimizing and caching data as shown in figure 5.1.

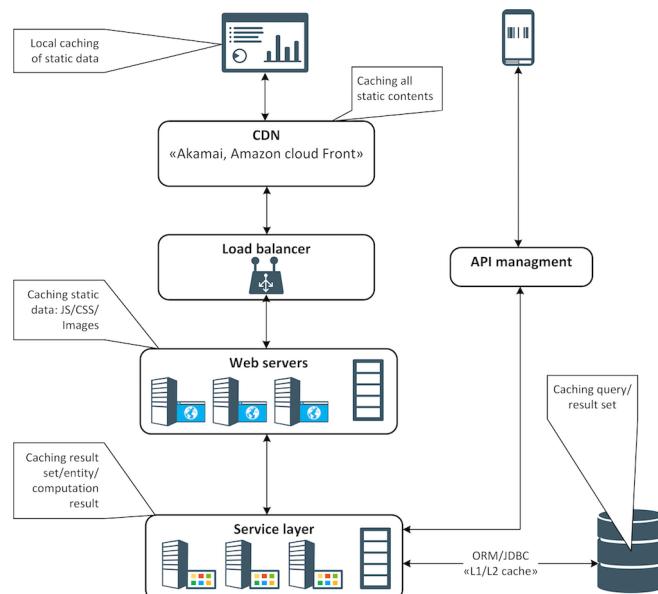


Figure 5.1

Caches can be applied and leveraged throughout the various layers of technology including browsers, network layers (Content delivery network and DNS), web applications and databases as shown in figure 5.1. Cached information can include the result of database queries, computationally intensive calculation, API request/responses, and web artifacts such as HTML, JavaScript, and multi-media files. Therefore, for getting a high throughput of an entire application, you should consider optimizations on other layers, and not only the use of in-memory caching layer. Such a way, you will get the maximum benefits of caching data and improves the overall performance of the application.

## Caching best practices

It's essential to consider a few best practices for using cache *smartly* when implementing a cache on any application layer. A smart caching ensure that you implement all (or most of them) the best practices whenever designing a cache. This subsection describes a few considerations for using a cache.

1. **Decide when and which data to cache.** Caching can improve performance; the more data you can cache, the higher the chance to reduce the latency and contention that's associated with handling large volumes of concurrent requests in the original data store. However, server resources are finite, and unfortunately, you could not cache all the resources you want. Consider caching data that are read frequently but modified rarely.
2. **The resilience of the caching layer.** Your application must continue its operation by using the primary data storage if the caching layer is unavailable, and you shouldn't lose any critical piece of information.
3. **Determine how to cache data effectively.** Most often, caching is less useful for dynamic data. The key to using a cache successfully lies in determining the most appropriate data to cache and caching it in the proper time. The data can be added to the cache on demand the first time it is fetched from the store by the application. Subsequent access to this data can be satisfied by using this cache. On the other hand, you can upload the data into the cache during the application startup, and sometimes it's called cache warm up.
4. **Managing data expiration in caches.** You can maintain a cache entry up-to-date by expiring a cache entry into the cache. When a cached data expires, it's removed from the cache, and a new cache entry will be added into the cache at the next time when it will be fetched from the primary data. You can set a default expiration policy when you configure the cache. However, consider the expiration period for the cache carefully. Cache entry expires too quickly if you make it too short, and you will reduce the

benefits of using the cache. On the other hand, you risk the data becoming stale if you make the period too long. Additionally, you should also consider to configure the cache eviction policy which will help you to evict entries from the cache whenever the cache is full and no more places exists to add a new entry.

5. **Update the caches when data changes on the primary data store.** Generally, a middle-tier caching layer duplicates some data from the central database server. Its goal is to avoid redundant queries to the database. The cache entry has to be updated or invalidated when the data updates in the database. You should consider the possibility to maintain a cache entry as up-to-date as possible when designing a caching layer. Many database vendors allow getting a notification whenever any entity updates into the database and updates the caches.
6. **Invalidate data in a client-side cache.** Data that is stored in a client-side cache (browser or any standalone application) is generally considered to be auspices of the service that provides the data to the client. A service cannot directly force a client to add or remove information from a client-side cache. This means that it's possible for a client that poorly configured the cache to continue using the stale information. However, a service that provides cache needs to ensure that each server response provides the correct HTTP header directives to instruct the browser on when and for how long the browser can cache the response.

## Design patterns

There might be two different strategies in a distributed computing environment when caching data:

- *Local or private cache.* The data is stored locally on the server that's running an instance of an application or service. Application performance is very high in this strategy, because, most often the cached data stored in the same JVM along with application logic. However, when the cache is resident on the same node as the application utilizing it, scaling may affect the integrity of the cache. Additionally, when local caches are used, they only benefit the local application that consuming the data.
- *Shared or distributed cache.* The cache served as the common caching layer that can be accessed from any application instances and architecture topology. Cached data can span multiple cache servers in this strategy, and be stored in a central location for getting the benefit of all the consumers of the data. This approach is especially relevant in a system where application nodes can be dynamically scaled in and out.



## Tip

With the Apache Ignite you can implement either or both of the above strategies when caching data.

## Basic terms

There are a few basic terms related to caching, frequently used throughout this book. I strongly believe that you are already familiar with these terms. However, it will be useful for those who are not familiar with these terms and getting all the information in a single place.

Terms	Description
Cache entry	A single cache value, consists of a key and its mapped data value within the cache.
Cache Hit	When a data entry is requested from the cache, and the entry exists for the given key. A more cache hit means that most of the requests are satisfied by the cache.
Cache Miss	When a data entry is requested from the cache, and the entry does not exist for the given key.
Hot data	Data that has recently been used by an application is very likely to be reassessed soon. Such data is considered hot. A cache may attempt to keep the hottest data most quickly available while trying to choose the least hot data for eviction.
Cache eviction	The removal of entries from the cache in order to make room for newer entries, typically when the cache has run out of data storage capacity.
Cache expiration	The removal of entries from the cache after some amount of time has passed, typically as a strategy to avoid stale data in the cache.

## Database caching

There are many challenges that disk-based databases (especially RDBMS) can pose to your application when developing a distributed system that requires low latency and horizontal scaling. A few common challenges are as follows:

1. *Expensive query processing.* Database queries can be slow and require serious system resources because the database system needs to perform some computation to fulfill the query request.

2. *Database hotspots.* It's likely that a small subset of data such as a celebrity profile or popular product (before Christmas) will be accessed more frequently than others in many applications. The SQL queries on such favorite products can result in hot spots in your database and maybe overprovisioning of database resources (CPU, RAM) based on the throughput requirements for the most frequently used data.
3. *The cost to scale.* Most often, RDBMS are only scaling vertically (anyway, [Oracle 18c](#)<sup>69</sup> and [Postgres-XL](#)<sup>70</sup> can scaling horizontally but needs tremendous effort to configure). Scaling databases for extremely high reads can be costly and may require many databases read replicas to match the current business needs.

Most database servers are configured by default for optimal performance. However, each database vendors provides various optimizations tips and tricks to help engineers get the most out of their databases. These guidelines for database optimization observe a law similar to the funnel law that is illustrated in figure 5.2 and described below:

1. Reducing data access.
2. Returning less data.
3. Reducing interaction with the underlayer.
4. Reducing CPU overhead and using more machine resources.

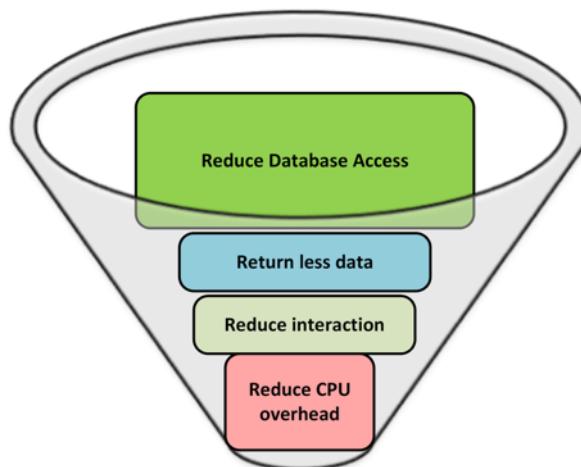


Figure 5.2

<sup>69</sup><https://www.oracle.com/technetwork/database/database-technologies/sharding/overview/index.html>

<sup>70</sup><https://www.postgres-xl.org>

Architects and engineers should make a great effort in squeezing as much performance as they can out of their database as mentioned earlier, because database caching should be implemented when all existing optimization tools reach a database bottleneck. A database cache supplements your primary database by removing unnecessary pressure on it (very close to the *reduce data access layer*), typically in the form of frequently accessed read data. The cache itself can live in some areas including your database, application or as a standalone layer.

The basic paradigm when querying data from a relational database from an application includes executing SQL statement through ORM tools or JDBC API and iterating over the returned *ResultSet* object cursor to retrieve the database rows. There are a few techniques you can apply based on your data access tools and patterns when wanting to cache the returned data.

In this section, we are going to discuss how Ignite in-memory cache can be used as a 2<sup>nd</sup> level caches in different data access tools such as *Hibernate* and *Mybatis*, which can significantly reduce the data access times of your application and improve overall application performance.

A 2<sup>nd</sup> level cache is a local or distributed data store of entity data managed by the persistence provider to improve application performance.

A second level cache can improve application performance by avoiding expensive database calls, keeping the data bounded (locally available) to the application. A 2<sup>nd</sup> level cache is fully managed by the persistence provider and typically transparent to the application. That is, application reads, writes and commits data through the entity manager without knowing about the cache.

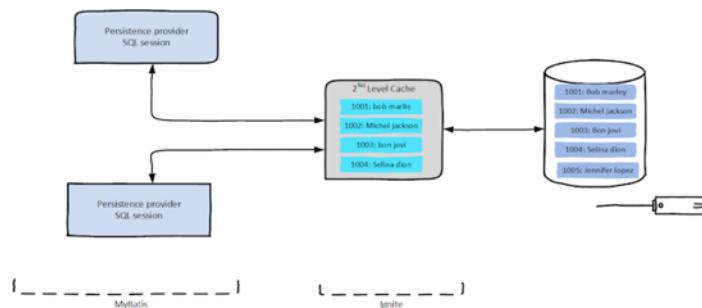


Figure 5.3

There is also a **Level 1** cache based on the persistence provider, such as MyBatis or Hibernate. Level 1 is used to cache objects retrieved from the database within the current database

session. All operations performed during the session will share the L1 cache, so the same object will not retrieve twice from the database. Objects retrieved from the database will not be available after closing the database session.



## Tip

In most persistence providers, *level 1* cache is always enabled *by default*.

So, in a nutshell, the 2<sup>nd</sup> level cache provides the following benefits:

1. Boost performance by avoiding expensive database calls.
2. Data are kept transparent to the application.
3. CRUD operation can be performed through standard persistence manager functions.
4. You can accelerate applications performance by using 2<sup>nd</sup> level cache, without changing the code.

## Hibernate caching

We are going to develop a **CRUD**<sup>71</sup> application in **Hibernate**<sup>72</sup> and use the Apache Ignite as an in-memory caching layer between Hibernate and the persistence store in this section. Unlike any other ORM framework, Hibernate provides three types of caching:

- **First-level or Session Cache.** As we mentioned before, a first level cache is the session cache, it's enabled by default, and every request to the database must pass through this level. The scope of this level is the session. Its cached objects are within the current database session.
- **2<sup>nd</sup> level cache.** The 2<sup>nd</sup> level cache is responsible for caching objects (entity) across sessions. Objects will be first searched in the cache when it's turned on, and a database query will be executed if they are not found. Note that the second level cache will be used only when the objects are loaded using their object ID. Hibernate 2<sup>nd</sup> level cache stores the entity data, not the entity or POJO itself. The data is stored in a serialized format, where the key is the entity identity, and the value is the list of primitive types. It is also possible to store data in the 2<sup>nd</sup> level cache in a more human-readable format. This can be useful if you would like to be able to browse the cache entries directly in your cache but does have a *performance impact*.

<sup>71</sup>[https://en.wikipedia.org/wiki/Create,\\_read,\\_update\\_and\\_delete](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete)

<sup>72</sup><http://hibernate.org/>

- **Query Cache.** The query cache is used to cache the result of a *query*. It can be Hibernate *SQL* query or *Hibernate named query (HQL)* with property `setCacheable(true)`. When the query cache is enabled, the results of the query are stored in the cache with a combination of the *query and parameters*. The cache manager checks for the query result and the associated parameter in the cache every time the query executes, if they are found in the caches, the cache manager returns the result from the cache. Otherwise, the cache manager propagates the query to the database.

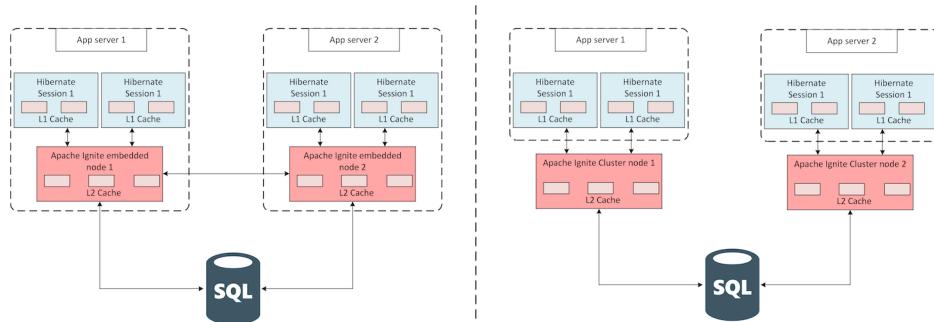


Figure 5.4

The Hibernate L2 and query cache features are optional and require a few additional physical cache regions that hold the cached query results and the timestamps when a table (or cache) was last updated. This feature is handy for queries that are frequently run with the same parameters. Figure 5.4 illustrated the high-level view of the Hibernate caches and its implementation in Apache Ignite. Apache Ignite can be used as an embedded node with an application or used separately.

Let's get started developing a Hibernate CRUD application in full swing. We will go through the different steps involved in creating an XML web service (web application) using Hibernate and Spring. This application is managing a list of employees and expose a web service endpoint for adding employee's and retrieving existing employee's profile. Next, we will configure the hibernate 2<sup>nd</sup> level entity cache and the query caches to the application and measure the performances of the web application. The full source code is available at [GitHub repository<sup>73</sup>](#).

**Environment.** Tools and technologies used for this application are:

1. Java 1.8
2. Maven 3.5.4

<sup>73</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-5/hibernate>

3. Ignite 2.6
4. Ignite-hibernate 5.1
5. Postgres 9.6.1.0
6. Hibernate 5.1.5
7. Spring ORM 4.3.18
8. Spring 4.3.16
9. Chrome [Wizdler<sup>74</sup>](#) extension or SoapUI.



## Warning:

In the moment of writing this book, Apache Ignite does not support Hibernate version 5.2.\*.

**Step 1.** Hibernate is LGPL licensed, so *Ignite-hibernate 5.1* version Java library is not available in the central maven repository. You need to build them from the Apache Ignite source code and install in your local maven repository. Therefore, download the source code from the Apache Ignite site, and follow the steps 1-9 from the *chapter 2* section *Building from source code*. You should have installed all the required artifacts in your local repository after completing *step 9*.

**Step 2.** Download and install *Postgres SQL server*. The installation process is out of the scope of this book; you can find a lot of tutorials/articles on the internet which shows you how to install and configure Postgres SQL server in the various environment. Anyway, you have to make a few changes in *resources/JDBC properties* files if you are planning to use your already configured RDBMS.

**Step 3.** Follow the steps from the *chapter 2* section *First Java application* for creating a new Maven project from scratch.

**Step 4.** Add the following maven dependencies and properties in your project *pom.xml* file.

---

<sup>74</sup><https://chrome.google.com/webstore/detail/wizdler/oebpmncolmhiapingjaagmapififiakb>

Listing 5.1

---

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <hibernate.version>5.1.5.Final</hibernate.version>
    <ignite.version>2.6.0</ignite.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-core</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-spring</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <!-- Spring orm-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>4.3.18.RELEASE</version>
    </dependency>
    <!-- Hibernate libs-->
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-hibernate_5.1</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-log4j</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>${hibernate.version}</version>
    </dependency>
    <!-- PostgreSQL-->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
```

```
<version>42.2.4</version>
</dependency>
</dependencies>
```

---

Note that, in the *hibernate/pom.xml* file of the GitHub project, we do not explicitly specify the ignite version for most of the Ignite libraries, because in our Github project we are using parent *pom.xml* file, where we have specified all the dependencies version we have used throughout the entire project. This gives a high degree of control when you work with multi-module projects and applications that consist of tens or hundreds of modules.

Finally, we can run a simple maven command which allows us to see a complete dependency tree of the project to understand all the libraries which we are added to the project. Here is a command which we can use:

**Listing 5.2**

---

```
mvn dependency:tree
```

---

When we run this above command, it shows the following dependency tree of the current project.

**Listing 5.3**

---

```
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ hibernate ---
[INFO] com.blu.imdg:hibernate:jar:1.0-SNAPSHOT
[INFO] +- org.apache.ignite:ignite-core:jar:2.6.0:compile
[INFO] |  +- javax.cache:cache-api:jar:1.0.0:compile
[INFO] |  +- org.jetbrains:annotations:jar:13.0:compile
[INFO] |  \- org.gridgain:ignite-shmem:jar:1.0.0:compile
[INFO] +- org.apache.ignite:ignite-spring:jar:2.6.0:compile
[INFO] |  +- org.apache.ignite:ignite-indexing:jar:2.6.0:compile
[INFO] |  |  +- commons-codec:commons-codec:jar:1.11:compile
[INFO] |  |  +- org.apache.lucene:lucene-core:jar:5.5.2:compile
[INFO] |  |  +- org.apache.lucene:lucene-analyzers-common:jar:5.5.2:compile
[INFO] |  |  +- org.apache.lucene:lucene-queryparser:jar:5.5.2:compile
[INFO] |  |  |  +- org.apache.lucene:lucene-queries:jar:5.5.2:compile
[INFO] |  |  \- org.apache.lucene:lucene-sandbox:jar:5.5.2:compile
[INFO] |  \- com.h2database:h2:jar:1.4.195:compile
[INFO] +- org.springframework:spring-core:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-aop:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-beans:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-context:jar:4.3.16.RELEASE:compile
```

```
[INFO] | +- org.springframework:spring-expression:jar:4.3.16.RELEASE:compile
[INFO] | +- org.springframework:spring-tx:jar:4.3.16.RELEASE:compile
[INFO] | +- org.springframework:spring-jdbc:jar:4.3.16.RELEASE:compile
[INFO] | \- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- org.springframework:spring-orm:jar:4.3.18.RELEASE:compile
[INFO] +- org.apache.ignite:ignite-hibernate_5.1:jar:2.6.0:compile
[INFO] | +- org.apache.ignite:ignite-hibernate-core:jar:2.6.0:compile
[INFO] | \- org.hibernate:hibernate-core:jar:5.1.5.Final:compile
[INFO] |   +- antlr:antlr:jar:2.7.7:compile
[INFO] |   +- org.jboss:jandex:jar:2.0.3.Final:compile
[INFO] |   \- com.fasterxml:classmate:jar:1.3.0:compile
[INFO] +- org.apache.ignite:ignite-log4j:jar:2.6.0:compile
[INFO] | \- log4j:log4j:jar:1.2.17:compile
[INFO] +- org.hibernate:hibernate-entitymanager:jar:5.1.5.Final:compile
[INFO] | +- org.jboss.logging:jboss-logging:jar:3.3.0.Final:compile
[INFO] | +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |   \- xml-apis:xml-apis:jar:1.0.b2:compile
[INFO] | +- org.hibernate.common:hibernate-commons-annotations:jar:5.0.1.Final:compile
[INFO] | +- org.hibernate.javax.persistence:hibernate-jpa-2.1-api:jar:1.0.0.Final:compile
[INFO] | +- org.javassist:javassist:jar:3.20.0-GA:compile
[INFO] | \- org.apache.geronimo.specs:geronimo-jta_1.1_spec:jar:1.1.1:compile
[INFO] \- org.postgresql:postgresql:jar:42.2.4:compile
[INFO] -----
```

---

Quite interesting? So many dependencies were added by just adding seven dependencies into the project. Apache Ignite and hibernate collects all the related (transitive) dependencies itself. The most significant advantage is that all these dependencies are guaranteed to be compatible with each other.

**Step 5.** Create the *EMP* and *DEPT* table by using the following SQL statements. You can use your favorite SQL IDE to run the SQL queries.

**Listing 5.4**

---

```
CREATE TABLE dept
(
    deptno INTEGER,
    dname TEXT,
    loc TEXT,
    CONSTRAINT pk_dept PRIMARY KEY (deptno)
);

CREATE TABLE emp
(
    empno INTEGER,
    ename TEXT,
    job TEXT,
    mgr INTEGER,
    hiredate DATE,
    sal INTEGER,
    comm INTEGER,
    deptno INTEGER,
    CONSTRAINT pk_emp PRIMARY KEY (empno),
    CONSTRAINT fk_deptno FOREIGN KEY (deptno) REFERENCES dept (deptno)
);

CREATE UNIQUE INDEX ename_idx
ON emp (ename);
```

---

**Step 6.** Add a few records into the *EMP* and *DEPT* table through SQL INSERT statement.

**Listing 5.5**

---

```
insert into dept
values(10, 'ACCOUNTING', 'NEW YORK');
insert into dept
values(20, 'RESEARCH', 'DALLAS');
insert into dept
values(30, 'SALES', 'CHICAGO');
insert into dept
values(40, 'OPERATIONS', 'BOSTON');

insert into emp
values( 7839, 'KING', 'PRESIDENT', null, to_date('17-11-1981','dd-mm-yyyy'),
5000, null, 10
);
```

```

insert into emp
values( 7698, 'BLAKE', 'MANAGER', 7839, to_date('1-5-1981','dd-mm-yyyy'),
2850, null, 30
);
insert into emp
values( 7782, 'CLARK', 'MANAGER', 7839, to_date('9-6-1981','dd-mm-yyyy'),
2450, null, 10
);
insert into emp
values( 7566, 'JONES', 'MANAGER', 7839, to_date('2-4-1981','dd-mm-yyyy'),
2975, null, 20
);
insert into emp
values( 7788, 'SCOTT', 'ANALYST', 7566, to_date('13-07-87','dd-mm-rr') - 85,
3000, null, 20
);
insert into emp
values( 7902, 'FORD', 'ANALYST', 7566, to_date('3-12-1981','dd-mm-yyyy'),
3000, null, 20
);
insert into emp
values( 7369, 'SMITH', 'CLERK', 7902, to_date('17-12-1980','dd-mm-yyyy'),
800, null, 20
);
-- Rest of the part is omitted. Please use the full source code from chapter-5/scripts directory.

commit;

```

---

**Step 7.** To keep things simple, we start with a tiny well-known domain *Employee* with following attributes.

#### Listing 5.6

---

```

@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Table( name = "emp" )
public class Employee implements Serializable{
    @Id
    //@GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "empno")
    private Integer empno;

```

```
@Column(name = "ename")
private String ename;
@Column(name = "job")
private String job;
@Column(name = "mgr")
private Integer mgr;
@Column(name = "hiredate")
private Date date;
@Column(name = "sal")
private Integer sal;
@Column(name = "deptno")
private Integer deptno;
@Column(name = "comm")
private Integer comm;

public Employee() {
}
// setters and getters are omitted
}
```

---

We omitted all the setter and getter methods for simplicity. See the `com.blu.imdg.dto.Employee` Java class for full source code.

1. Firstly, we've annotated the class with `@Entity`, which tells Hibernate that this class represents an object that we can persist. Any entity class can extend either entity class or a non-entity user-defined class. It can contain constructors, methods, fields and nested types with any access modifier.
2. The `@Table(name = "emp")` annotation allows you to specify the details of the table that will be used to persist the entity in the database. Any additional table also might be specified by the annotation `SecondaryTable`.
3. The `@Column(name = "empno")` annotation is used to map this property to the EMPNO column in the emp table.
4. The `@Cacheable` annotation enables caching for the given entity.
5. The `@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)` annotation defines the concurrent caching strategy, in our case it's `READ_WRITE`. A strategy can be `TRANSITIONAL`, `READ_ONLY`, etc., depends on the entity manager.
6. The `@Id` annotation is used to define a unique identifier of the entity instances, similar to a primary key. Primary key values are unique per entity class.

**Step 8.** In this step, we are going to define a service interface which will act as a contract for the implementation and represents all the actions our Service must support. These actions will be related to creating new employee and getting information related to the objects in the database. Here is the Employee Data Access Layer (DAO) interface contract we will be using:

**Listing 5.7**

---

```
public interface EmpDao {  
    List<Employee> getAllEmployees();  
    Employee getEmployeeById (int empno);  
    void create(Integer empno, String ename, String job, Integer mgr );  
    List<Employee> getEmpByName(String ename);  
}
```

---

1. `getAllEmployees ()` – this method will return all the *Employee* records from the tables.
2. `create(Integer empno, String ename, String job, Integer mgr)` – will create a new employee with an employee number, employee name. Note that, *empno* will always be unique to create a new Employee.
3. `getEmpByName(String ename)` – returns Employee entity by employee name.
4. `getEmployeeById` – returns an Employee entity by employee Id.

**Step 9.** Similarly, we have a web service with four web methods to invoke from the end users.

**Listing 5.8**

---

```
@javax.jws.WebService(name = "BusinessRulesServices",  
    serviceName = "BusinessRulesServices",  
    targetNamespace = "http://com.blu.rules/services")  
public class WebService {  
    private EmpDao empDao;  
  
    @WebMethod(exclude = true)  
    public void setEmpDao(EmpDao empDao) {  
        this.empDao = empDao;  
    }  
  
    @WebMethod(operationName = "getAllEmployees")  
    public List < Employee > getAllEmployees() {  
        return empDao.getAllEmployees();  
    }
```

```

}

{@WebMethod(operationName = "addEmployee")
public void addEmployee(@WebParam(name = "Employee_ID") int empno, @WebParam(name = "\nName") String ename, @WebParam(name = "Job_title") String job, @WebParam(name = "Manager_\nID") Integer mgr) {
    empDao.create(empno, ename, job, mgr);
}

{@WebMethod(operationName = "getEmpByName")
public List < Employee > getEmpByName(@WebParam(name = "Employee_Name") String ename)\n{
    return empDao.getEmpByName(ename);
}

{@WebMethod(operationName = "getEmpByNo")
public Employee getEmpByNo(@WebParam(name = "Employee_ID") int empno) {
    return empDao.getEmployeeById(empno);
}
}

```

---

The `@javax.jws.WebService` marks this Java class as a Web service which will be managed by the spring and `@WebMethod` annotation marks that this Java method should be exposed as a Web method, which can be invoked by the end users.

All of the above four web methods delegates the web services call to the implementation of the DAO class. Let's take a look at a few DAO implementation methods from class `EmpDaoImpl`.

#### **Listing 5.9**

---

```

public List<Employee> getEmpByName(String ename) {
    Session session = sessionFactory.openSession();
    Query query = session.createQuery("from Employee e where e.ename=:ename");
    query.setParameter("ename", ename);
    List<Employee> employees = query.list();
    session.close();
    return employees;
}

```

---

In the above, preceding `getEmpByName()` method, we create a JDBC session from a session factory, then establishes an HQL (Hibernate Query) query from this session. Next set the parameter of this query, and then run the query by calling the `query.list()` function and returns the query results. For rest of the part of this DAO implementation, please refer to the

source code of the `com.blu.imdg.EmpDaoImpl.java`. Let's try another DAO implementation method in detail.

**Listing 5.10**

---

```
public Employee getEmployeeById(int empno) {
    Session session = sessionFactory.openSession();
    Employee emp = session.get(Employee.class, empno);
    session.close();
    return emp;
}
```

---

The method implementation is small and straightforward. Firstly, we open a session from the Hibernate session factory, then use the `session.get()` method to retrieve the *Employee* entity by the employee Id. The `get()` method returns the object by fetching it from the database or the L2 cache. The process of retrieving the data from the database or the cache is sufficiently transparent for the application code, and Hibernate entity manager is fully responsible for the entire process.



### Tip:

Hibernate session has another method named `load()`, which returns the reference of an object that might not actually exist, it loads the data from the database or only the cache when you access other properties of the object, for instance when you call the `Employee.getJob()` method.

**Step 10.** So far so good. At this moment, let's add the spring configuration, `spring-core.xml`. This file is too large to fit in an entire page. So, I will split the configurations into a few blocks for simplicity. Note that, spring configurations can be very compact and can import other spring configuration files. Firstly, we will ship the data source configuration as shown below:

**Listing 5.11**

---

```
<context:property-placeholder location="classpath:jdbc.properties"/>
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

---

In the above listing, we define all the data source configurations from the *jdbc.properties* file.

**Listing 5.12**

---

```
jdbc.driver=org.postgresql.Driver
jdbc.url=jdbc:postgresql://localhost:5432/postgres
jdbc.username=postgres
jdbc.password=postgres
```

---

Note that, you have to **change** this configuration if you are going to use another *RDBMS* rather than PostgreSQL. Now add the hibernate session factory configuration part as follows:

**Listing 5.13**

---

```
<!-- Hibernate session factory-->
<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBe
an">

    <property name="dataSource" ref="dataSource"/>

    <property name="packagesToScan">
        <list>
            <value>com.blu.imdg.dto</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.cache.use_second_level_cache">true</prop>
            <!-- Generate L2 cache statistics. -->
            <prop key="hibernate.generate_statistics">true</prop>

            <!-- Specify ignite as L2 cache provider. -->
            <prop key="hibernate.cache.region.factory_class">org.apache.ignite.cache.hibe\
```

```

rnate.HibernateRegionFactory</prop>

    <!-- Specify the name of the grid, that will be used for second level caching\

. -->
    <prop key="org.apache.ignite.hibernate.ignite_instance_name">hibernate-grid</
prop>
    <!-- Set default L2 cache access type. -->
    <prop key="org.apache.ignite.hibernate.default_access_type">READ_WRITE</prop>

    <!-- Enable query cache. prefix hibernate is necessary here.-->
    <prop key="hibernate.cache.use_query_cache">true</prop>
    <prop key="show_sql">true</prop>
    <prop key="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</prop>

</props>
</property>
</bean>

```

---

Take a moment to review the code comments to understand how it works. In the above configuration, we first include our data source properties bean. Then we ensure the packages to scan for *Hibernate entity classes* are available. Next, we enable the hibernate 2<sup>nd</sup> level cache and set the cache provider. In our case, the cache provider is Ignite `HibernateRegionFactory`. Also, we specify the name of the Ignite cache instance, which will be used for the second level cache, and set the cache access type to `READ_WRITE`. We also enable the query cache here. Note that, you have to use the hibernate prefix to enable query cache. Next, we will define our DAO implementation bean and the web service bean into the `spring-core.xml` file.

**Listing 5.14**

---

```

<bean id="empDAO" class="com.blu.imdg.EmpDaoImpl">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<!-- Service-->
<bean id="serviceBean" class="com.blu.imdg.ws.WebService">
    <property name="empDao" ref="empDAO"/>
</bean>

```

---

**Step 11.** Now it's time to add the *Ignite configuration*. We have moved the Ignite configuration in the separate file named `ignite-l2-config.xml`.

**Listing 5.15**

---

```
<bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    <!-- Set to true to enable distributed class loading for examples, default is false. -->
    <property name="peerClassLoadingEnabled" value="true"/>

    <property name="igniteInstanceName" value="hibernate-grid"/>
    <!-- Enable client mode. -->
    <property name="clientMode" value="false"/>

    <property name="cacheConfiguration">
        <list>
            <!--
                Configurations for entity caches.
            -->
            <bean parent="transactional-cache">
                <property name="name" value="com.blu.imdg.dto.Employee"/>
            </bean>
            <!-- Query cache (refers to atomic cache defined in above example). -->
            <bean parent="atomic-cache">
                <property name="name" value="org.hibernate.cache.internal.StandardQueryCache"/>
            </bean>
            <!-- Configuration for update timestamps cache, it's also necessary for query cache-->
            <bean parent="atomic-cache">
                <property name="name" value="org.hibernate.cache.spi.UpdateTimestampsCache"/>
            </bean>
        </list>
    </property>
</bean>
```

---

Here, we first set the Ignite instance name property to *hibernate-grid*, then we *disable* the client mode. Finally, we set the *transactional cache mode* for our Employee entity. Also, we set a *Hibernate standard query cache* and *timestamp* cache to Ignite *atomic* cache.

**Tip:**

The Hibernate `UpdateTimestampsCache` will store the *time* whenever the cache entry has been updated.

There is also a basic configuration for transactional cache and atomic cache. These configurations are shown below.

**Listing 5.16**

---

```
<bean id="atomic-cache" class="org.apache.ignite.configuration.CacheConfiguration" abstract="true">
    <property name="cacheMode" value="PARTITIONED"/>
    <property name="atomicityMode" value="ATOMIC"/>
    <property name="writeSynchronizationMode" value="FULL_SYNC"/>
</bean>
<!-- Basic configuration for transactional cache. -->
<bean id="transactional-cache" class="org.apache.ignite.configuration.CacheConfiguration" abstract="true">
    <property name="cacheMode" value="PARTITIONED"/>
    <property name="atomicityMode" value="TRANSACTIONAL"/>
    <property name="writeSynchronizationMode" value="FULL_SYNC"/>
</bean>
```

---

We used *PARTITIONED* cache to distribute the data between Ignite nodes. Another possible strategy is to enable *REPLICATED* mode. A full dataset will be available at each node in a *REPLICATED* mode. Also, we indicate *TRANSACTIONAL* atomicity mode to take advantage of cache transactions. Moreover, we enable *FULL\_SYNC* mode to be always fully synchronized with backup nodes. Please refer to the [chapter 4](#) for more information about Ignite cache mode.

**Step 12.** We have also defined a `resources/log4j.xml` file to enable *logging*. The `log4j.xml` is the main configuration file having all runtime configuration used by the *log4j framework*. This file contains *appenders* information and log level information. The fragment of this file is shown below.

**Listing 5.17**

---

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern"
            value="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n" />
    </layout>
</appender>
<!-- Log SQL statements -->
<logger name="org.hibernate.SQL" additivity="false">
    <level value="debug"/>
    <appender-ref ref="console" />
```

```
</logger>
<!-- Log cache -->
<logger name="org.hibernate.cache" additivity="false">
    <level value="debug"/>
    <appender-ref ref="console" />
</logger>
<root>
    <level value="INFO" />
    <appender-ref ref="console" />
</root>
```

---

The interesting point of the above pseudo XML is that we defined the log level information to *DEBUG* for the Hibernate SQL and cache packages, the rest of the packages logs messages with level *INFO* or above. All the log messages will be displayed on the application console.

**Step 13.** Our setup is almost ready. Let's write a simple Java class to boot our web service.

**Listing 5.18**

```
public class RunWebService {
    public static void main(String[] args) {
        Ignite ignite = Ignition.start("ignite-l2-config.xml");
        ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-core.xml");
        WebService service = (WebService) ctx.getBean("serviceBean");
        Endpoint.publish("http://localhost:7001/invokeRules", service);
        System.out.println("Server start in Port .. 7001");
        System.out.println("Web service WSDL is available at http://localhost:7001/invokeRules?wsdl");
    }
}
```

---

The above Java class contains the `main()` method which will run the entire application. Firstly, we start an Ignite server node in **embedded mode**. Next, we start a standalone Spring XML application context, taking the context definition files `spring-core.xml` from the classpath. This is a simple, one-shop convenience Spring application context, which is very useful for test harnesses. We get the `WebService` bean instance from the application context by the given bean name and set the endpoint URL to localhost with port 7001 in the next statement.



## Tip:

Ignite server (in embedded mode), or client node should be run on the same JVM along with web application. You have to configure Ignite as a client node in the `ignite-l2-config.xml` file if you have an Ignite cluster running on separates JVM.

**Step 14.** At this moment, we are ready to compile and run our application. Type the following command into the terminal to compile the project.

**Listing 5.19**

---

```
mvn clean install
```

---

We are going to use a *one-jar plugin* to run the web service. One-jar is a maven plugin, which let you package a Java application together with all the required dependencies into a single executable Jar file. See the plugins section in the *hibernate/pom.xml* file for more details. Run the web service from the folder *chapter-5/hibernate* by the following command.

**Listing 5.20**

---

```
java -jar ./target/hibernate-1.0-SNAPSHOT.one-jar.jar
```

---

You should see some output like this.

```
2018-09-06 21:51:25 WARN  JdbcEnvironmentInitiator:132 - HHH000342: Could not obtain connection to query metadata : Connection to localhost:5432 refused. Check that the hostname and port are correct and that the postmaster is accepting TCP/IP connections.
2018-09-06 21:51:25 INFO  Dialect:156 - HHH000400: Using dialect: org.hibernate.dialect.PostgreSQLDialect
2018-09-06 21:51:25 INFO  LobCreatorBuilderImpl:63 - HHH000422: Disabling contextual LOB creation as connection was null
2018-09-06 21:51:25 INFO  BasicTypeRegistry:139 - HHH000270: Type registration [java.util.UUID] overrides previous : org.hibernate.type.UUIDBinaryType@3b94d659
2018-09-06 21:51:25 INFO  UpdateTimestampsCache:55 - HHH000250: Starting update timestamps cache at region: org.hibernate.cache.spi.UpdateTimestampsCache
2018-09-06 21:51:25 INFO  StandardQueryCache:74 - HHH000248: Starting query cache at region: org.hibernate.cache.internal.StandardQueryCache
Server start in Port .. 7001
Web service WSDL is available at http://localhost:7001/invokeRules?wsdl
2018-09-06 21:51:33 INFO  GridUpdateNotifier:566 - Your version is up to date.
```

**Figure 5.5**

Now, we can use any web service client like *SoapUI* or *Wizdler chrome extension* to check out the services. I am going to make use of the *Wizdlerchrome plugin*; which is easy to install and run through the browser. Let's invoke the web method `getEmpByNo` with *employee ID* 7788 (this is the ID for employee SCOTT) to watch the response time.



**Figure 5.6**

You should get the following response from the server if everything goes fine.

```

POST http://localhost:7001/invokeRules
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getEmpByNoResponse xmlns:ns2="http://com.blu.rules/services">
      <return>
        <date>0001-04-19T00:00:00+03:00</date>
        <deptno>20</deptno>
        <empno>7788</empno>
        <ename>SCOTT</ename>
        <job>ANALYST</job>
        <mgr>7566</mgr>
        <sal>3000</sal>
      </return>
    </ns2:getEmpByNoResponse>
  </S:Body>
</S:Envelope>

```

Request      Response      Done (86 ms).

Figure 5.7

Note that the response time is 86 ms. Let's deep dive into the application log and Ignite node to discover what's happening under the hood.

```

2018-09-06 22:05:05 DEBUG SQL:92 - select employee0_.empno as empno1_0_0_, employee0_.comm as comm2_0_0_, employee0_.hire
date as hiredate3_0_0_, employee0_.deptno as deptno4_0_0_, employee0_.ename as ename5_0_0_, employee0_.job as job6_0_0_,
employee0_.mgr as mgr7_0_0_, employee0_.sal as sal8_0_0_ from emp employee0_ where employee0_.empno=?
2018-09-06 22:05:05 INFO  StatisticalLoggingSessionEventListener:258 - Session Metrics {
  5442623 nanoseconds spent acquiring 1 JDBC connections;
  0 nanoseconds spent releasing 0 JDBC connections;
  5612646 nanoseconds spent preparing 1 JDBC statements;
  1921402 nanoseconds spent executing 1 JDBC statements;
  0 nanoseconds spent executing 0 JDBC batches;
  56515534 nanoseconds spent performing 1 L2C puts;
  0 nanoseconds spent performing 0 L2C hits;
  293789 nanoseconds spent performing 1 L2C misses;
  0 nanoseconds spent executing 0 flushes (flushing a total of 0 entities and 0 collections);
  0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}

```

Figure 5.8

Look at the output carefully, for the first time a new JDBC connection is acquired, and it's taken 5442623 nanoseconds. It's approximately 5.4 ms. Next Hibernate spent 1921402 nanoseconds for executing the JDBC statement. Then it has taken a few nanoseconds to put the entity into the cache.

Let's run the Ignite visor command line console to monitor the caches. Use the following command to run the Ignite visor console.

**Listing 5.19**


---

ignitevisor.sh

---

Choose the local configuration file config/default-config.xml by hitting the key 0. Next, enter the command cache -scan which will show you the following output.

```
visor> cache -scan
Time of the snapshot: 2018-09-06 22:27:24
+-----+
| # |           Name(@)          | Mode    | Size (Heap / Off-heap) |
+-----+
| 0 | com.blu.imdg.dto.Employee(@0) | PARTITIONED | min: 1 (0 / 1)
|   |                               |          | avg: 1.00 (0.00 / 1.00)
|   |                               |          | max: 1 (0 / 1)
+---+
| 1 | org.hibernate.cache.internal.StandardQueryCache(@c1) | PARTITIONED | min: 0 (0 / 0)
|   |                               |          | avg: 0.00 (0.00 / 0.00)
|   |                               |          | max: 0 (0 / 0)
+---+
| 2 | org.hibernate.cache.spi.UpdateTimestampsCache(@c2) | PARTITIONED | min: 0 (0 / 0)
|   |                               |          | avg: 0.00 (0.00 / 0.00)
|   |                               |          | max: 0 (0 / 0)
+-----+
```

**Figure 5.9**

It's clear from the output, that com.blu.imdg.dto.Employee cache contains an entry into the cache. If you choose the cache number to 0, it produces the following result.

```
+-----+
| o.a.i.i.binary.BinaryObjectImpl | org.apache.ignite.cache.hibernate.HibernateKeyWrapper [hash=410249197, key=7788, en
try=com.blu.imdg.dto.Employee, tenantId=null] | o.a.i.i.binary.BinaryObjectImpl | org.hibernate.cache.spi.entry.Stan
dCacheEntryImpl [hash=202351689, disassembledState=[Ljava.lang.Object@70c087e5, subclass=com.blu.imdg.dto.Employee, ve
rsion=null] |
```

**Figure 5.10**

Although the output of the console is not very readable, it's clear that Ignite stores the Employees entity in serialized format into the cache. We can also use the hibernate.cache.use\_structured\_entries Hibernate properties to force the Hibernate to store data in the 2<sup>nd</sup> level cache in a more human-readable format. Next time, if we are to rerun the above web method, Hibernate entity manager will return the result from the cache and the response time will be very small. Let's rerun the web method with the same Employee ID 7788.

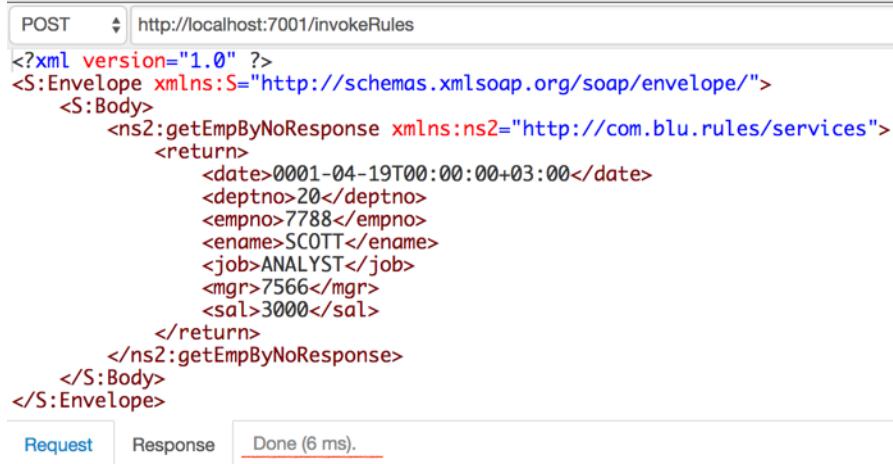


Figure 5.11

This time the response time decreased to 6 ms. Let's have a look at the application console log again.

```

2018-09-06 22:41:26 INFO  StatisticalLoggingSessionEventListerner:258 - Session Metrics {
  0 nanoseconds spent acquiring 0 JDBC connections;
  0 nanoseconds spent releasing 0 JDBC connections;
  0 nanoseconds spent preparing 0 JDBC statements;
  0 nanoseconds spent executing 0 JDBC statements;
  0 nanoseconds spent executing 0 JDBC batches;
  0 nanoseconds spent performing 0 L2C puts;
  377339 nanoseconds spent performing 1 L2C hits;
  0 nanoseconds spent performing 0 L2C misses;
  0 nanoseconds spent executing 0 flushes (flushing a total of 0 entities and 0 collections);
  0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}

```

Figure 5.12

No JDBC connection is acquired at this moment, and the entity returns from the cache (1 L2 cache hits). Hibernate entity manager spent 377339 nanoseconds or 0,3 ms for retrieving the cache entry. Impressive! Theoretically, this caching pattern called *read through caching*.

**Step 15.** Let's add a new (you can also uncomment the property in *hibernateProperties* section) Hibernate property in the `resources/spring-core.xml` file.

#### Listing 5.20

---

```
<prop key="hibernate.cache.use_structured_entries">true</prop>
```

---

What we have stated before, when this above property is true, it forces hibernate to store data in a human readable format in the 2<sup>nd</sup> level cache. Follow the step 14 to rebuild the application and invoke the web method `getEmpByNo` for a few times. Use the `cache -scan` command to display

the cache entries of the `com.blu.imdg.dto.Employee` cache. You should get very similar information displayed on the console as shown below.

```
| o.a.i.i.binary.BinaryObjectImpl | org.apache.ignite.cache.hibernate.HibernateKeyWrapper [hash=410249197, entry=com.blu.imdg.dto.Employee, tenantId=null, key=7788] | java.util.HashMap | {date=0001-04-19 00:00:00.0, _subklass=com.blu.imdg.dto.Employee, commnull, ename=SCOTT, mgr=7566, job=ANALYST, _version=null, deptno=20, sal=3000} |
```

Figure 5.13

*Employee cache* stores all the properties of the entity in a more human-friendly format at this moment. You probably should turn it *ON*, only if you plan to browse through the cache.



## Warning:

Setting the parameter `hibernate.cache.use_structured_entries` to `true` will generate a serious overhead in the L2 cache. So, use it carefully.

**Step 15.** Let's see what will happen if we add a new *Employee* through our web service. Theoretically, it will add a new Employee entry into the Ignite cache and also commit the Employee instance into the database. Let's invoke the web method `addEmployee` to create a new *Employee* with the following properties.

Listing 5.21

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <addEmployee xmlns="http://com.blu.rules/services">
      <Employee_ID xmlns="">1234</Employee_ID>
      <Name xmlns="">Test12</Name>
      <Job_title xmlns="">Mgr</Job_title>
      <Manager_ID xmlns="">7788</Manager_ID>
    </addEmployee>
  </Body>
</Envelope>
```

Next, run the `cache -scan` command in *ignitevisor* console to scan the caches.

#	Name(@)	Mode	Size (Heap / Off-heap)
0	com.blu.imdg.dto.Employee(@c0)	PARTITIONED	min: 1 (0 / 1) avg: 1.00 (0.00 / 1.00) max: 1 (0 / 1)
1	org.hibernate.cache.internal.StandardQueryCache(@c1)	PARTITIONED	min: 0 (0 / 0) avg: 0.00 (0.00 / 0.00) max: 0 (0 / 0)
2	org.hibernate.cache.spi.UpdateTimestampsCache(@c2)	PARTITIONED	min: 1 (0 / 1) avg: 1.00 (0.00 / 1.00) max: 1 (0 / 1)

Figure 5.14

Exactly what we expected, one new cache entry for Employee now allocated into the Ignite cache. We can reformat the cache entry to take a better look at it. Follow *step 15* to reformat the employee properties in human-readable format. We should be found an exciting thing if we take a closer look at our list of the caches into the Ignite. UpdateTimestampCache cache also has a cache entry for it. Let's scan the cache to observe the cache entry.

Entries in cache: org.hibernate.cache.spi.UpdateTimestampsCache			
Key Class	Key	Value Class	Value
java.lang.String	emp	java.lang.Long	1536511934341

Figure 5.15

Entries of this cache contain the timestamp value in milliseconds, which indicate the time when the table has been updated. It will show the exact time of the cache entry time if we convert the timestamp value to date, in my case, it's *Sunday, September 9, 2018, 8:36:13 PM GMT*. Being aware of this cache is very important whenever you are using query caching (details in the next section) because, Hibernate uses it to verify that cached query results are not stale.

We have reached the end of this section. This time we will detail how to cache the result of *HQL queries*. This is useful if you are frequently executing a query on entities that rarely changed. You have to do the following to enable query cache.

**Step 16.** Set the Hibernate property `hibernate.cache.use_query_cache` value to `TRUE` in the `spring-core.xml` file.

**Listing 5.22**


---

```
<!-- Enable query cache. prefix hibernate is necessary here.-->
<prop key="hibernate.cache.use_query_cache">true</prop>
```

---

**Step 17.** For each query, you have to explicitly set that the query is cacheable via an org.hibernate.cacheable query hint or Query.setCacheable(true) property. In our cases, we will be using `Query.setCacheable(true)` property. Let's modify the DAO method `getEmpByName()` in the `EmpDaoImpl` Java class as follows:

**Listing 5.23**


---

```
public List<Employee> getEmpByName(String ename) {
    Session session = sessionFactory.openSession();
    Query query = session.createQuery("from Employee e where e.ename=:ename");
    query.setParameter("ename", ename);
    // ENABLE QUERY CACHE
    query.setCacheable(true);
    List<Employee> employees = query.list();
    session.close();
    return employees;
}
```

---

In line *five* `query.setCacheable(true)` caches the query. Now, let's invoke our third web method: `getEmpByName()`. We are looking for the employee **SMITH** by his name.

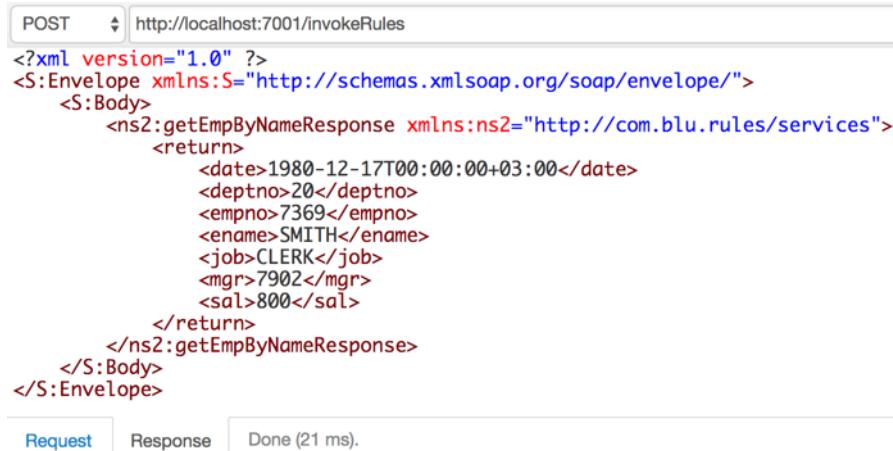


Figure 5.16

The query returns within 21 ms of times. Let's *rerun* the web method, and monitor the response time.

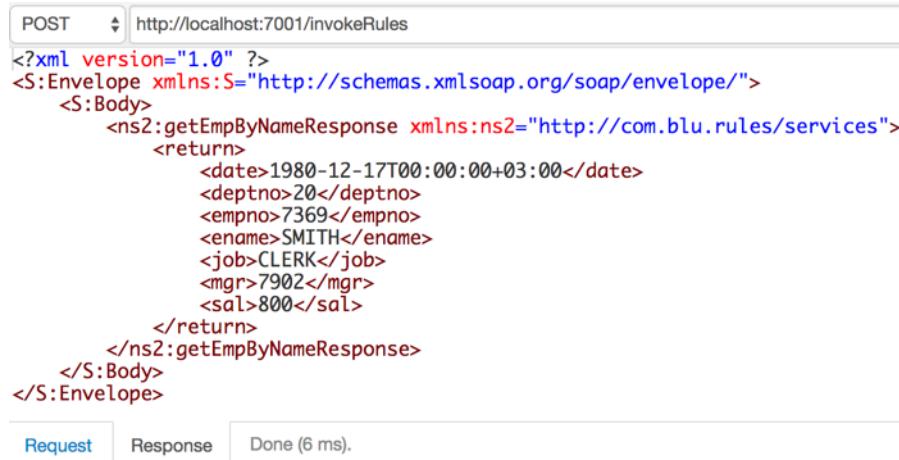


Figure 5.17

As we expected, it took only 6 ms of time to get a response. We should find the cache entry for our database query with parameter SMITH if we take a look at the cache via *IgniteVisor* CLI.

```

| o.a.i.i.binary.BinaryObjectImpl | org.hibernate.cache.spi.QueryKey [hash=337104888, sqlQueryString=select employee0_.empno as empno1_0_, employee0_.comm as comm2_0_, employee0_.hiredate as hiredate3_0_, employee0_.deptno as deptno4_0_, employee0_.ename as ename5_0_, employee0_.job as job6_0_, employee0_.mgr as mgr7_0_, employee0_.sal as sal8_0_ from emp employee0_ where employee0_.ename =?, positionalParameterTypes=[Ljava.lang.Object;@60f644db, positionalParameterValues=[Ljava.lang.Object;@52680d63, namedParameter swiename=org.hibernate.engine.spi.TypedValue [idHash=1871879885, hash=-187635181, type=org.hibernate.type.StringType [idHash=1176341699, hash=666966831, dictatedSize=org.hibernate.engine.jdbc.Size [idHash=337627284, hash=891931472, length=255, precision=19, scale=0], alterTypeDescriptor=org.hibernate.engine.jdbc.descriptor.AlterTypeDescriptor [idHash=1086362721, hash=1111111111], javaTypeDescriptor=org.hibernate.engine.jdbc.descriptor.java.StringDescriptor [idHash=1671235869, hash=-1088697, type=java.lang.String, mutabilityPlan=org.hibernate.type.descriptor.java.ImmutableMutabilityPlan [idHash=1229775376, hash=1], comparator=org.hibernate.internal.util.compare.ComparableComparator [idHash=595287791, hash=11], sqlTypes=[121], value=SMITH]], firstRow=null, maxRows=null, tenantIdentifier=null, filterKeys=null, customTransformer=org.hibernate.transform.CacheableResultTransformer [idHash=1742513415, hash=853949397, tupleLength=1, tupleSubsetLength=1, includeInTuple=[true], includeInTransformIndex=null]]] | java.util.ArrayList [ [1536519863357, 7369] ]
-----
```

Figure 5.18

Let's dig into the application log for more information. You should find logs very similar in the application console, as shown below.

```

2018-09-09 22:06:39 DEBUG StandardQueryCache:155 - Checking cached query results in region: org.hibernate.cache.internal.StandardQueryCache
2018-09-09 22:06:39 DEBUG StandardQueryCache:253 - Checking query spaces are up-to-date: [emp]
2018-09-09 22:06:39 DEBUG UpdateTimestampsCache:171 - [emp] last update timestamp: 1536517688332, result set timestamp: 1536519863
357
2018-09-09 22:06:39 DEBUG StandardQueryCache:178 - Returning cached query results
2018-09-09 22:06:39 INFO  StatisticalLoggingSessionEventListener:258 - Session Metrics {
    0 nanoseconds spent acquiring 0 JDBC connections;
    0 nanoseconds spent releasing 0 JDBC connections;
    0 nanoseconds spent preparing 0 JDBC statements;
    0 nanoseconds spent executing 0 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    855771 nanoseconds spent performing 3 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    0 nanoseconds spent executing 0 flushes (flushing a total of 0 entities and 0 collections);
    0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}

```

Figure 5.19

At this moment, Hibernate first checks the query cache result in the cache *StandardQueryCache* and also validate the last update timestamp of the cache *UpdateTimestampsCache*. Next, it returns the query result from the cache directly whenever the cache is valid. No JDBC connection has been created in this case, and Hibernate spent only 0.8 ms to return the cache query result.

So, by setting up an in-memory cache as a 2<sup>nd</sup> level cache we reduce the query execution time of 86 ms to 6 ms, and the cache entry retrieving time is approximately 0.3 ms. In the next section, we are going to cover the integration details with *MyBatis* and the Apache Ignite.

## MyBatis caching

[MyBatis](#)<sup>75</sup> provides two layers of caching unlike Hibernate: a *local cache* which is always enabled, and a 2<sup>nd</sup> level cache which is *optional*. MyBatis local cache is very similar to Hibernate L1 cache and stores data for the duration of a SQL session. However, it cannot optimize access to read-only or most read-only data. On the other hand, MyBatis L2 cache is consulted first, when performing queries. MyBatis, 2<sup>nd</sup> level cache has the following characteristics:

1. Shares data between SQL sessions and transactions.
2. Can use a 3<sup>rd</sup> party in-memory caches to store and retrieve cache entries into it.
3. By default, flushes the entire cache whenever an insert/update/delete statement is executed.

Ignite can be used as a MyBatis 2<sup>nd</sup> level cache for improving data access layer from version 1.5, which can give a performance boost to your application. In this section, we are going to demonstrate how to implement Apache Ignite over MyBatis and how to achieve an optimum configuration for it.

---

<sup>75</sup><http://www.mybatis.org>

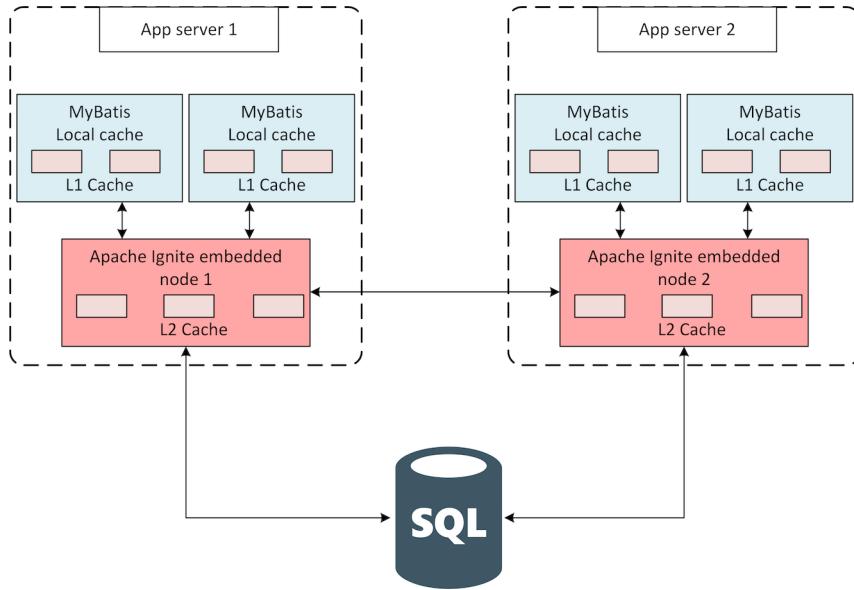


Figure 5.20

MyBatis stores the entity data in the 2<sup>nd</sup> level cache, but not the entity or objects itself. The data is stored in a serialized format, which looks like a HashMap where the key is the entity ID, and the value is a list of primitive values. Usually, key of each entry builds from a hash value consists of (MyBatis -namespace + stmt-name) + raw SQL + (actual parameter value). Here is an example of how the MyBatis cache entries look like in the Apache Ignite cache:

```

__Key__: org.apache.ibatis.cache.CacheKey [hash=2025310740, multiplier=37,
hashcode=421191021, checksum=6335551979, count=6,
updateList=[com.blu.imdg.mapper.UserMapper.getEmployee, 0, 2147483647, SELECT * FROM
emp WHERE ename = ?, SMITH, SqlSessionFactoryBean]]

__Value__: com.blu.imdg.dto.Employee [idHash=531482076, hash=924414128, empno=7369,
ename=SMITH, job=CLERK, mgr=7902, date=null, sal=800, deptno=20]

```

Figure 5.20.1

MyBatis 2<sup>nd</sup> level cache is very similar to Hibernate 2<sup>nd</sup> level Query cache that we have described in the previous section. Next, we will develop an application with MyBatis and Ignite to demonstrate the performance gain you can achieve. All the source code of our example application will be available at [GitHub repositories](#)<sup>76</sup>.

**Environment.** Tools and technologies used for this application are:

<sup>76</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-5/mybatis>

1. Java 1.8
2. Maven 3.5.4
3. Ignite 2.6
4. MyBatis 3.4.0
5. MyBatis Ignite 1.0.6
6. Postgres 9.6.1.0
7. Spring 4.3.16
8. Chrome Wizdler extension or SoapUI

**Step 1.** Create a Maven project. Follow the steps from the *chapter 2* section *First Java application* for creating a new Maven project from scratch.

**Step 2.** Simply add the following dependencies to your *pom.xml* file:

Listing 5.24

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-log4j</artifactId>
    <version>2.6.0</version>
</dependency>
<!-- MyBatis Dependencies-->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ignite</artifactId>
    <version>1.0.6</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>1.3.0</version>
</dependency>
<dependency>
```

```
<groupId>org.mybatis</groupId>
<artifactId>mybatis</artifactId>
<version>3.4.0</version>
</dependency>
<dependency>
    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.1-901.jdbc4</version>
</dependency>
```

---

In this project, we are using spring framework, so we added a few spring related dependencies. Also, we added a MyBatis core library and mybatis-ignite library to integrate with the Apache Ignite as a 2<sup>nd</sup> level cache. At compile time, Maven uses the information from the *pom.xml* file to look up all these above libraries in the Maven repository. Maven first looks in the repository on your local computer. If the libraries aren't there, then, it will download them from the Maven public repository and store them into your local repository for future use. Note that, I am going to use the *Postgres database version 9* as before.

**Step 3.** Run the following command to understand all the libraries that are used in this application.

#### **Listing 5.25**

---

```
mvn dependency:tree
```

---

The above command will display the following dependency tree of the current project.

#### **Listing 5.26**

---

```
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ mybatis ---
[INFO] com.blu.imdg:mybatis:jar:1.0-SNAPSHOT
[INFO] +- org.apache.ignite:ignite-core:jar:2.6.0:compile
[INFO] |  +- javax.cache:cache-api:jar:1.0.0:compile
[INFO] |  +- org.jetbrains:annotations:jar:13.0:compile
[INFO] |  \- org.gridgain:ignite-shmem:jar:1.0.0:compile
[INFO] +- org.apache.ignite:ignite-spring:jar:2.6.0:compile
[INFO] |  +- org.apache.ignite:ignite-indexing:jar:2.6.0:compile
[INFO] |  |  +- commons-codec:commons-codec:jar:1.11:compile
[INFO] |  |  +- org.apache.lucene:lucene-core:jar:5.5.2:compile
[INFO] |  |  +- org.apache.lucene:lucene-analyzers-common:jar:5.5.2:compile
[INFO] |  |  +- org.apache.lucene:lucene-queryparser:jar:5.5.2:compile
[INFO] |  |  +- org.apache.lucene:lucene-queries:jar:5.5.2:compile
```

```
[INFO] |  |  \- org.apache.lucene:lucene-sandbox:jar:5.5.2:compile
[INFO] |  \- com.h2database:h2:jar:1.4.195:compile
[INFO] +- org.springframework:spring-core:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-aop:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-beans:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-context:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-expression:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-tx:jar:4.3.16.RELEASE:compile
[INFO] +- org.springframework:spring-jdbc:jar:4.3.16.RELEASE:compile
[INFO] \- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- org.apache.ignite:ignite-log4j:jar:2.6.0:compile
[INFO] | \- log4j:log4j:jar:1.2.17:compile
[INFO] +- org.mybatis.caches:mybatis-ignite:jar:1.0.6:compile
[INFO] +- org.mybatis:mybatis-spring:jar:1.3.0:compile
[INFO] +- org.mybatis:mybatis:jar:3.4.0:compile
[INFO] \- postgresql:postgresql:jar:9.1-901.jdbc4:compile
```

---

**Step 4.** We are going to use the same table `EMP` and `DEPT` that we have used in the previous section. So, follow the *step 5-6* from the previous example to create the tables, if you have not done yet. Now the project setup is in place, let's go ahead with MyBatis implementation.

**Step 5.** Now, we have to add our *spring context XML* file in the `resources` directory of the project to add it to Java classpath. Let's have a detailed look at the `spring-core.xml` file.

**Listing 5.27**

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mapperLocations" value="classpath*:com/blu/imdg/dao/*Mapper.xml" />
</bean>
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="${jdbc.driver}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
  <property name="basePackage" value="com.blu.imdg.mapper" />
</bean>
<bean id="userServicesBean" class="com.blu.imdg.dao.UserServices">
  <property name="userMapper" ref="userMapper" />
</bean>
<bean id="servicesBean" class="com.blu.imdg.WebServices">
```

```
<property name="dao" ref="userServicesBean" />
</bean>
```

---

You need at least two things defined in the Spring application context to use MyBatis with Spring framework: a `SqlSessionFactory` and at least one mapper interface. In MyBatis Spring integration a `SqlSessionFactoryBean` is used to create a `SqlSessionFactory` interface. You use the `SqlSessionFactory` to create a `sqlSession`. Once you have a SQL session, you can use it for executing any SQL statements. With MyBatis you do not need to use `SqlSessionFactory` directly unlike Hibernate session factory from the previous section, because your beans can be injected with a thread-safe `SqlSession` that automatically commits, rollbacks and close the connection when needed. We also added the classpath of all the SQL mapper files in the XML through a `mapperLocations` property.

Note that, `SqlSessionFactory` requires a `DataSource`. We used the `SpringManagerDataSource` to configure the data source from the `jdbc.properties` file. Please do not forget to change the `jdbc.URL` and `jdbc.password` by your database configuration.

Next, we used `MapperScannerConfigurer` that searches recursively starting from a base package for interfaces and registers them as `MapperFactoryBean`. The `basePackage` property can contain more than one package name, separated by either commas or semicolons. Once everything is configured, we have defined two more beans: `userServicesBean` and `servicesBean`. `userServicesBean` bean provided the DAO layer and required an `userMapper` which will be injected automatically by the MyBatis at runtime. The `servicesBean` is our well-known web service implementation which provides a web method for invoking from any web service client.

**Step 6.** Now, it's time to register a mapper. You can register the mapper either using a standard XML configuration or the new 3.0+ Java Config (a.k.a. `@Configuration`). I prefer the classical way with XML configuration, because, I like to hold the configuration as separate as possible from the code. Create a few directories `com/blu/imdg/mapper` under `src/main/resources` folder and a new XML file with name `UserMapper.xml`. Add the following content as shown below:

**Listing 5.28**

---

```
<mapper namespace="com.blu.imdg.mapper.UserMapper">
    <cache type="org.mybatis.caches.ignite.IgniteCacheAdapter" />
    <select id="getEmployee" parameterType="String" resultType="com.blu.imdg.dto.Employee"\n    " useCache="true">
        SELECT * FROM emp WHERE ename = #{eName}
    </select>
</mapper>
```

---

The true power of MyBatis is in the mapper XML. This is where the magic happens. The mapper file is relatively simple. MyBatis was built to focus on the SQL and does it's best to stay out of your way in contrast with Hibernate. In this above snippet, we have first defined the mapper namespace as `com.blu.imdg.mapper.UserMapper`. This namespace is equal to the `UserMapper` Java interface we are going to define soon. Next, we specified the Ignite cache adapter as a cache type for the Mapper. At runtime, MyBatis will start an Ignite instance in the same JVM with this application and creates a cache with the mapper name. In this case, the cache name will be `com.blu.imdg.mapper.UserMapper`.

We added a `SQL SELECT` statement and specified that it uses caching in the last portion of the above XML configuration. We also define the result type and the parameter type of the `SELECT` statement. Result type of the statement is `com.blu.imdg.dto.Employee` and the parameter will be the employee name as `String` type.

**Step 7.** Let's start adding the domain class for `Employee` entity once all the declarative parts of the application completed. Add the following Java POJO under the `dto` folder of the project.

**Listing 5.29**

---

```
public class Employee implements Serializable{
    private Integer empno;
    private String ename;
    private String job;
    private Integer mgr;
    private Date date;
    private Integer sal;
    private Integer deptno;
    // setter and getter are omitted here.
}
```

---

The above Java class is very simple and defined all the necessary properties. Note that, we do not use any annotations in the above snippet.

**Step 8.** Add the following Java interface under the mapper folder.

**Listing 5.30**

---

```
public interface UserMapper {  
    Employee getEmployee (String eName);  
}
```

---

The `UserMapper` interface provides only one method named `getEmployee()`, which returns the `Employee` *object* by the employee name.

**Step 9.** Next, we have to define our DAO interface. Add a new Java class with name `UserServices` under the `dao` package and add the following contents on it.

**Listing 5.31**

---

```
public class UserServices {  
    private UserMapper userMapper;  
    public void setUserMapper(UserMapper userMapper) {  
        this.userMapper = userMapper;  
    }  
    public Employee getEmployee (String eName) {return userMapper.getEmployee(eName);}  
}
```

---

The DAO layer contains only one method `getEmployee()` and uses the MyBatis `UserMapper` interface to execute the SQL statement.

**Step 10.** Now that you've set up the project, you can proceed to create your Web service. Add a new Java class with name `WebServices` as shown below:

**Listing 5.32**

---

```
@WebService(name = "BusinessRulesServices",  
    serviceName="BusinessRulesServices",  
    targetNamespace = "http://com.blu.rules/services")  
public class WebServices {  
    private UserServices userServices;  
  
    @WebMethod(exclude = true)  
    public void setDao(UserServices userServices){  
        this.userServices = userServices;  
    }  
  
    @WebMethod(operationName = "getEmployee")  
    public Employee getEmployee (@WebParam(name = "EmployeeName") String eName) {return u  
serServices.getEmployee(eName);}
```

```
}
```

---

At this moment, the SOAP web service contains only one web method; `getEmployee()`. All the related web services annotations used in the above class are the same as before.

**Step 11.** To run the web service, we have used the same way as before. Create a Java class with the `main()` method and add the following contents:

**Listing 5.33**

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-core.xml");
WebServices services = (WebServices) ctx.getBean("servicesBean");

Endpoint.publish("http://localhost:7001/invokeRules", services);
System.out.println("Server start in Port .. 7001");
System.out.println("Web service WSDL is available at http://localhost:7001/invokeRules?wsdl");
```

---

We first start a standalone Spring XML application context, taking the context definition files `spring-core.xml` from the classpath. Then, we get the `WebService` bean instance from the application context by the given bean name and set the endpoint URL to localhost with port `7001` in the next statement.



## Info:

MyBatis cache adapter will automatically start an Ignite instance during application startup. You don't need to start any Ignite node explicitly.

**Step 12.** At this moment, we are ready to compile and run our application. Type the following command into your terminal to compile the project.

**Listing 5.34**

---

```
mvn clean install
```

---

We are going to use a `one-jar` Maven plugin to run the web service. Run the web service from the folder `chapter-5/mybatis` by the following command.

**Listing 5.35**

---

```
mvn clean install && java -jar ./target/mybatis-1.0-SNAPSHOT.one-jar.jar
```

---

You should see some output like this.

```
2018-09-12 23:07:08 INFO GridDhtPartitionsExchangeFuture:566 - finishExchangeOnCoordinator [topVer=AffinityTopologyVersion [topVer=1, minorTopVer=1], resVer=AffinityTopologyVersion [topVer=1, minorTopVer=1]]
2018-09-12 23:07:08 INFO GridDhtPartitionsExchangeFuture:566 - Finish exchange future [startVer=AffinityTopologyVersion [topVer=1, minorTopVer=1], endVer=AffinityTopologyVersion [topVer=1, minorTopVer=1], err=null]
2018-09-12 23:07:08 INFO time:566 - Finished exchange init [topVer=AffinityTopologyVersion [topVer=1, minorTopVer=1], crd=true]
2018-09-12 23:07:08 INFO GridCachePartitionExchangeManager:566 - Skipping rebalancing (nothing scheduled) [top=AffinityTopologyVersion [topVer=1, minorTopVer=1], evt=DISCOVERY_CUSTOM_EVT, node=b568db9f-f54f-4866-bbb7-9b54b6c60c6b]
log4j: Finalizing appender named [null].
Server start in Port .. 7001
Web service WSDL is available at http://localhost:7001/invokeRules?wsdl
```

Figure 5.21

Our web service is up and running on port 7001.

**Step 13.** Now, you can use any web service client in your choice. Let's invoke the web method `getEmployee()` with employee name **SMITH** as shown below.



The screenshot shows a SOAP UI interface. The request type is set to "POST" and the URL is "http://localhost:7001/invokeRules". The XML payload is as follows:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <getEmployee xmlns="http://com.blu.rules/services">
      <EmployeeName xmlns="">SMITH</EmployeeName>
    </getEmployee>
  </Body>
</Envelope>
```

Figure 5.22

You should get the following response from the server if everything goes fine.

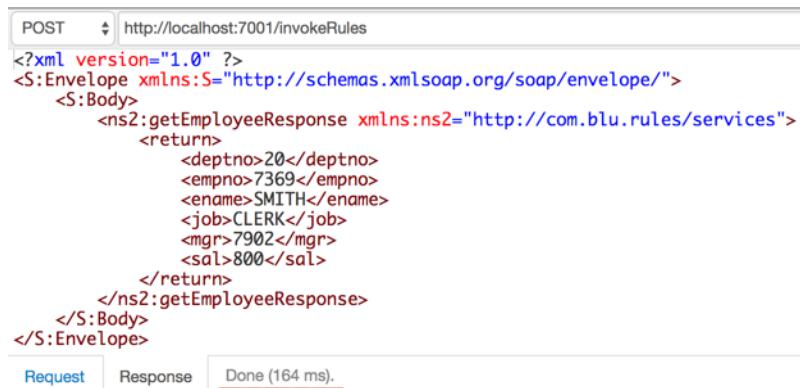


Figure 5.23

Note that, the response time is 164 ms. Let's reinvoke the web method again with the *same employee name*.

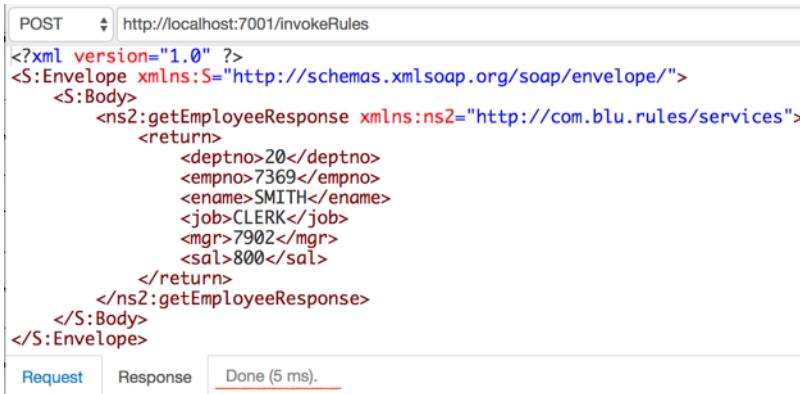


Figure 5.24

Now the response time is only 5 ms; MyBatis cache adapter returns the employee data directly from the cache at this moment. Let's run the Ignite visor CLI to monitor the caches. Use the following command to run the Ignite visor console.

#### Listing 5.36

---

ignitevisor.sh

---

Choose the local configuration file config/default-config.xml by hitting the key 0. Next, enter the command cache -scan which will show you the following output:

```
[visor> cache -scan
Time of the snapshot: 2018-09-12 23:18:13
+-----+
| # |           Name(@)          | Mode    | Size (Heap / Off-heap) |
+-----+
| 0 | com.blu.imdg.mapper.UserMapper(@c0) | PARTITIONED | min: 1 (0 / 1)
|   |                                         |           | avg: 1.00 (0.00 / 1.00)
|   |                                         |           | max: 1 (0 / 1)
+-----+
Choose cache number ('c' to cancel) [c]:
```

Figure 5.25

From the output, it's clear that the `com.blu.imdg.dto.UserMapper` cache contains **one** cache entry. It produces the following result if you choose the cache number to 0.

Key Class	Key	Value Class
Value		
+-----+		
o.a.i.i.binary.BinaryObjectImpl   org.apache.ibatis.cache.CacheKey {hash=2825310740, multiplier=37, hashCode=421191021, ch		
ecksum=6335551979, count=6, updateList=[com.blu.imdg.mapper.UserMapper.getEmployee, 0, 2147483647, SELECT * FROM emp WHERE e		
name = ?, SMITH, SqlSessionFactoryBean]   java.util.ArrayList   [com.blu.imdg.dto.Employee [idHash=2047256401, hash=9244141		
28, empno=7369, ename=SMITH, job=CLERK, mgr=7902, date=null, sal=800, deptno=20]]		

Figure 5.26

As expected, MyBatis stored the query result into the *UserMapper cache*, and in subsequent calls with the same employee name; returns the employee data from the cache.

So, when you run your query for the first time, your application will execute the query on the database and retrieve the results. The second time you access the employee list, your application will access the in-memory storage, and the response time is very quick. It takes approximately ~5 ms. We will discover another caching method called *memorization* in the next section, which can also improve the overall performance of your application dramatically.

## Memoization

Caching is everywhere in computer world. From the hardware to software, caching exists almost in each layer (physical or abstraction). For instance, L1, L2 or L3 caches in CPU, static files such as images, JavaScript caching in web servers, query result caching in an application server or HTTP cache control, etc. So far, we have introduced only the second level cache, which can reduce database call to speed up application performance. However, there could be one more caching tier between Java method and persistence manager. For instance, assume, you have a process that takes a significant time to execute, and method

result can only depend on its input parameters. In this situation, you can cache the result of the Java function, which will return the same result for the same input.

All the goals of the caching in each layer are the same, avoid doing the same works repeatedly for saving the resources and time. According to Wikipedia, [memoization<sup>77</sup>](#) is defined as an optimization technique used to speed up computer programs by storing the result of expensive function calls into the cache and returning the cached result when the same inputs occur again. Calculate the *factorial of numbers*, and the *Fibonacci numbers sequence* is vivid examples of an expensive function's calls.

In one word, *memoization* is a *caching* technique in the scope of the function. However, the function should be a pure function; a pure function is a function which, given the same input will always return the same result. Sometimes it also called *idempotent* operation regarding mathematics.

Anyway, you could use this caching technique not only in pure function but also in any functions which can become expensive to call multiple times. For example, suppose you have a function, which returns the current U.S dollar rate for any given regions of Russia (U.S dollar exchange rate is different in each region of Russia). All the exchange rates are stored in the database or somewhere in a Microsoft Excel file. With memoization technique, the cache manager will immediately return the value from the cache by region, and the function will not start computing business logic at all. Very straightforward, yeah! Rather than reprocessing all the business logic, returning the previously cached response can save a lot of CPU resources. Thus, you can increase the economy of your function's execution time up to 2-3 milliseconds.

Maybe you have wondered, why this 2-3 ms is so essential? At the end of 2014 in Russia, when the dollar exchange rate was changed on every hour, people rushed to the bank website to know the exchange rate. As a result, a lot of Russian bank portal was not able to handle the high traffic.

Java provides a few different approaches to implement the memoization technique, one of them is Java *Temporary Caching API*, *Guava API* from Google, and the *Spring cache abstraction* from Spring. Apache Ignite supports and shipped with an implementation of the Spring Cache Abstraction. It provides support for transparently adding caching into the Spring application, which will store the cache into Ignite node. Since 4.1, Spring cache abstraction is significantly improved and even supports annotations. The cache abstraction provides a set of Java annotations which includes:

1. `@Cacheable` - triggers cache population;

---

<sup>77</sup><https://en.wikipedia.org/wiki/Memoization>

2. `@CacheEvict` - triggers cache eviction;
3. `@CachePut` - updates the cache without interfering with the method execution;
4. `@Caching` - regroups multiple cache operations to be applied to a method;
5. `@CacheConfig` - shares some common cache-related settings at class-level.

Moreover, there is some direct support for synchronized caching from the Spring 4.3 version. Now `@Cacheable` annotation allows you to specify the `sync` attribute to ensure only a single thread is creating the cache value. We will explain it in details a very soon. Before we start developing an example, let's discuss the cases of using memoization.

**Use cases.** For now, we can define two use cases to turn a function to its memoized version.

1. A function is pure, CPU intensive and called repeatedly.
2. A function is pure and is recursive.

The 1<sup>st</sup> use case is evident. Calculating a sequence of the Fibonacci number is an excellent example of the 2<sup>nd</sup> use case.

**Enable caching.** Now that we have got the basics, let's build something useful. We have to do two simple steps to enable Spring cache with Ignite:

1. Start an Ignite server node with proper configuration in the embedded node (i.e., in the same JVM where the spring application is running). Ignite node can have predefined caches, but it's not required. Caches will be created automatically on first access it needs.
2. Configure a `SpringCacheManager` as the cache manager in the Spring application context. The Spring cache manager will create a whole new *cache* for each *Java function*.

## How does it work?

The approach is simple. In Ignite, a cache will be created, where the key is a function input parameter, and the values are the function result. The key consists of the parameter type and the name of the parameter. When the Java function is called for the first time, Spring cache manager stores the function result into the appropriate Ignite cache. Next time, when the function is recalled with the *same input parameter*. Spring cache manager check the key and the value into the Ignite cache, and it returns the function result directly from the cache if the key exists into the cache. Let's start developing an example to demonstrate how memoization works in the real world. The full source code of the project will be available at the [GitHub repository](#)<sup>78</sup>.

---

<sup>78</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-5/memoization>

**Step 1.** Create a new Maven project. Please refer to the *chapter 2* section *First Java application* if you faced any difficulties while creating a Maven project.

**Step 2.** Add the following Maven dependencies and properties in your project *pom.xml* file.

Listing 5.37

---

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <hibernate.version>5.1.5.Final</hibernate.version>
    <ignite.version>2.6.0</ignite.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-core</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-spring</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <!-- Spring orm-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>4.3.18.RELEASE</version>
    </dependency>
    <!-- Hibernate libs-->
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-log4j</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>${hibernate.version}</version>
    </dependency>
    <!-- PostgreSQL-->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.2.4</version>
```

```
</dependency>
</dependencies>
```

---



## Info:

Note that, we are going to use Hibernate for querying database only. Hibernate *L2 cache* or *query caches* will not be used in this example.

**Step 3.** Create the table `Exchangerate` and insert some data into it. The DML and DDL script for creating the table is available at [GitHub repository](#)<sup>79</sup> (see the files `exc-rate.ddl` and `exc-rate.dml`).

**Listing 5.38**

```
create table exchangerate(
    ratedate      date,
    usdollar     decimal,
    euro         decimal,
    gbp          decimal,
    region        text,
    primary key (ratedate, region)
);
INSERT INTO public.exchangerate (ratedate,usdollar,euro,gbp,region) VALUES (
TO_DATE('2015-05-01','YYYY-MM-DD'),69.7,75.9,103.01,'Vladimir');
INSERT INTO public.exchangerate (ratedate,usdollar,euro,gbp,region) VALUES (
TO_DATE('2015-05-02','YYYY-MM-DD'),71.7,77.1,102.03,'Tver');
INSERT INTO public.exchangerate (ratedate,usdollar,euro,gbp,region) VALUES (
TO_DATE('2015-05-02','YYYY-MM-DD'),71.7,77.1,102.03,'Moscow');
INSERT INTO public.exchangerate (ratedate,usdollar,euro,gbp,region) VALUES (
TO_DATE('2015-05-02','YYYY-MM-DD'),71.7,77.1,102.03,'Vladimir');
INSERT INTO public.exchangerate (ratedate,usdollar,euro,gbp,region) VALUES (
TO_DATE('2015-05-03','YYYY-MM-DD'),71.9,77.4,101.03,'Tver');
INSERT INTO public.exchangerate (ratedate,usdollar,euro,gbp,region) VALUES (
TO_DATE('2015-05-03','YYYY-MM-DD'),70.3,76.1,103.07,'Moscow');
INSERT INTO public.exchangerate (ratedate,usdollar,euro,gbp,region) VALUES (
TO_DATE('2015-05-03','YYYY-MM-DD'),71.1,77.4,101.01,'Vladimir');
commit;
```

---

**Step 4.** Create a simple Hibernate entity for the table `Exchangerate` in the package `com.blu.imdg.dto`.

<sup>79</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-5/scripts>

**Listing 5.39**

---

```
@Entity
@Table( name = "exchangerate" )
public class ExchangeRate {
    @Id
    @Column(name = "region")
    private String region;
    @Column(name = "ratedate")
    private Date rateDate;
    @Column(name = "usdollar")
    private double usdollar;
    @Column(name = "euro")
    private double euro;
    @Column(name = "gbp")
    private double gbp;

    public ExchangeRate() {
    }
    // setter and getter parts are omitted
}
```

---

We have annotated the class with `@Entity` annotation, which tells Hibernate that this class represents an object we are going to persist in the database. Next, the `@Table` annotation allows us to specify the details of the table that will be used to persist the entity. The rest of the part of the Java class is omitted for simplicity.

**Step 5.** Let's define a data access layer. Create a new Java interface named `ExchangeRateDao` or copy it from the source code.

**Listing 5.40**

---

```
public interface ExchangeRateDao {
    String getUSDollarExchangeRateByRegion(String region);
    void updateExchange(ExchangeRate e);
}
```

---

The interface provides only two methods:

- `getUSDollarExchangeRateByRegion()` – this method returns the exchange rate of the US dollar by region.

- `updateExchange()` – this method will update the exchange rates of the foreign currency by the region.

**Step 6.** Similarly, we will have a web service with two web methods to invoke from the end users as follows:

**Listing 5.41**

---

```
@javax.jws.WebService(name = "BusinessRulesServices",
    serviceName = "BusinessRulesServices",
    targetNamespace = "http://com.blu.rules/services")
public class WebService {
    private ExchangeRateDao exchangeRateDao;

    @WebMethod(exclude = true)
    public void setEmpDao(ExchangeRateDao empDao) {
        this.exchangeRateDao = empDao;
    }

    @WebMethod(operationName = "getExchangeRateByRegion")
    public String getExchangeRateByRegion(@WebParam(name = "RegionName") String str) {
        return exchangeRateDao.getUSDollarExchangeRateByRegion(str);
    }

    @WebMethod(operationName = "updateExchangeRate")
    public void updateExchangeRate(@WebParam(name = "RegionName") String region, @WebParam(
        name = "Date") Date rateDate, @WebParam(name = "USdollarRate") double usdollar, @WebParam(
        name = "EuroRate") double euro, @WebParam(name = "BritishPoundRate") double gbp) {
        ExchangeRate e = new ExchangeRate();
        e.setRegion(region);
        e.setUsdollar(usdollar);
        e.setEuro(euro);
        e.setGbp(gbp);
        e.setRateDate(rateDate);

        exchangeRateDao.updateExchange(e);
    }
}
```

---

As we done before, we have used the `@javax.jws.WebService` annotation for exposing the Java class as a web service and so on.

**Step 7.** In this step, we have to create a DAO implementation. Create a new Java class with name ExchangeRateDaoImpl, which implements the ExchangeRateDao interface. Let's take a look at the two DAO implementation methods snippet below.

**Listing 5.42**

---

```
@Cacheable(value = "exchangeRate")
public String getUSDollarExchangeRateByRegion(String region) {
    Session session = sessionFactory.openSession();
    // in real life, it should be current date time
    SQLQuery query = session.createSQLQuery("select * from exchangerate e where e.ratedat\\
e = TO_DATE('2015-05-02','YYYY-MM-DD') and e.region=:region");
    query.setParameter("region", region);
    query.addEntity(ExchangeRate.class);
    ExchangeRate res = (ExchangeRate) query.uniqueResult();
    session.close();
    return String.valueOf(res.getUsdollar());
}

@CacheEvict(value = "exchangeRate", key = "#e.region")
public void updateExchange(ExchangeRate e) {
    Session session = sessionFactory.openSession();
    session.beginTransaction();
    SQLQuery query = session.createSQLQuery("update exchangerate \n" +
        " set usdollar = :usdollar" +
        " where region = :region and ratedate = TO_DATE('2015-05-02','YYYY-MM-DD')");

    query.setParameter("region", e.getRegion());
    query.setParameter("usdollar", e.getUsdollar());
    query.addEntity(ExchangeRate.class);
    query.executeUpdate();
    session.beginTransaction().commit();
    session.close();
}
```

---

Pay attention to the two new annotations used in the above pseudo code: `@Cacheable` and `@CacheEvict`.

- `@Cacheable(value = “exchangeRate”)` annotation marks the Java method as memoized function and will create a new cache with name `exchangeRate`. Spring cache manager will reuse the cache if the cache already exists. Each time the function is invoked, the cache is checked to see whether the invocation has been already executed and doesn't have to repeat the execution block of the method. By default, only one cache

is declared, but this annotation allows multiple names of the cache to be specified. For instance: `@Cacheable({"exchangeRate", "dollars"})`. In such a case, each of the caches will be checked before executing the method. The associated value will be returned if at least one entry is found in any of the caches.

- `@CacheEvict(value = "exchangeRate", key = "#e.region")` allows eviction of the particular cache. This process is very useful to remove the unused or stale cache entries from the Ignite. There are two flavors of this eviction process: you can either evict the entire cache or evict only the cache entry that is stalled when using this annotation. In the preceding code, only the cache entry with the appropriate region will be evicted.

Note that, we are using, Spring *SpEL* to declare the cache key in the preceding example. To explicitly specify what is used for the cache key, we used the `key` parameter of the annotation. Spring SpEL indicate that we have to get the value from the `ExchangeRate` entity and use its code region to delete the cache entry. You can use the following annotation to evict the entire cache.

**Listing 5.43**

---

```
@CacheEvict(value = "exchangeRate", allEntries=true)
```

---

Also note that, It's not recommended to evict the entire cache because when your cache is empty, every method call will execute the business logic, which can decrease the performance of your application for a while. The rest of the part is similar to the one we have done before in section Hibernate caching. In a nutshell, we are using Hibernate native SQL query feature to query the database.

**Step 8.** Now, it's time to configure the Spring application context file, and we will be ready to go for the test. Let's add Spring cache manager into the `spring-core.xml` file.

**Listing 5.44**

---

```
<!-- Spring cache manager for Memoization -->
<bean id="cacheManager" class="org.apache.ignite.cache.spring.SpringCacheManager">
    <property name="igniteInstanceName" value="hibernate-grid" />
</bean>
<!-- Enable annotation-driven caching. -->
<cache:annotation-driven/>
```

---

The `SpringCacheManager` bean will be the cache manager for the spring application as we stated earlier. For every Java function, the cache manager will create an entire new cache.

Also, the `annotation-driven` is responsible for registering the necessary Spring components that enable the caching for the Java function.

With the Spring cache manager bean, you can configure your cache parameters. You can customize the cache configuration via the `dynamicCacheConfiguration` property. For instance, if you want to use the *REPLICATED* cache instead of *PARTITIONED*, you have to configure `SpringCacheManager` bean as shown below:

Listing 5.45

---

```
<bean id="cacheManager" class="org.apache.ignite.cache.spring.SpringCacheManager">
    <property name="dynamicCacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <property name="cacheMode" value="REPLICATED"/>
        </bean>
    </property>
</bean>
```

---

Moreover, you can utilize a *near* cache on Ignite client node. You can provide `dynamicNearCacheConfiguration` property to achieve this. See the example below.

Listing 5.46

---

```
<bean id="cacheManager" class="org.apache.ignite.cache.spring.SpringCacheManager">
    <property name="dynamicNearCacheConfiguration">
        <bean class="org.apache.ignite.configuration.NearCacheConfiguration">
            <property name="nearStartSize" value="1000"/>
        </bean>
    </property>
</bean>
```

---

**Step 9.** Next, we have to define our Ignite configuration file. Ignite configuration file will be very compact as shown below.

**Listing 5.47**

---

```
<bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    <!-- Set to true to enable distributed class loading for examples, default is false. -->
    <property name="peerClassLoadingEnabled" value="true"/>
    <!-- Ignite instance name -->
    <property name="igniteInstanceName" value="hibernate-grid"/>
    <!-- Enable client mode. -->
    <property name="clientMode" value="false"/>
    <!-- Explicitly configure TCP discovery SPI to provide list of initial nodes. -->
    <property name="discoverySpi">
        <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
            <property name="ipFinder">
                <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulti\
castIpFinder">
                    <property name="addresses">
                        <list>
                            <!-- In distributed environment, replace with actual host IP address. -->
                            <value>127.0.0.1:47500..47509</value>
                        </list>
                    </property>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

---

In the preceding configuration, we have just specified the Ignite node as a *server node*, *instance name* and *IP address* of the server node. As we described before, Spring cache manager will create the caches during first access of the Java function.

**Step 10.** Our setup is almost done, and we are going to use our previous *RunWebService* Java class to boot the web service. Our web service will be available at port 7001.

**Step 11.** To build and execute the web service, type the following command into any of your favorite terminals.

**Listing 5.48**

---

```
mvn clean install && java -jar ./target/memoization-1.0-SNAPSHOT.one-jar.jar
```

---

You should see some output similar to this.

```

log4j: Finalizing appender named [null].
2018-09-13 14:58:14 INFO Dialect:156 - HHH000400: Using dialect: org.hibernate.dialect.PostgreSQLDialect
2018-09-13 14:58:14 INFO LobCreatorBuilderImpl:63 - HHH000422: Disabling contextual LOB creation as connection was null
2018-09-13 14:58:14 INFO BasicTypeRegistry:139 - HHH000270: Type registration [java.util.UUID] overrides previous : org
inaryType@481a15ff
Server start in Port .. 7001
Web service WSDL is available at http://localhost:7001/invokeRules?wsdl

```

Figure 5.27

The web service is up and running on port 7001. Now, we can use any web service client like SoapUI or Wizdler chrome extension to check out the services. As usual, I am going to use the *Wizdler chrome plugin* to invoke the web service. Let's invoke the web method `getExchangeRateByRegion()` with the region name *Moscow*.

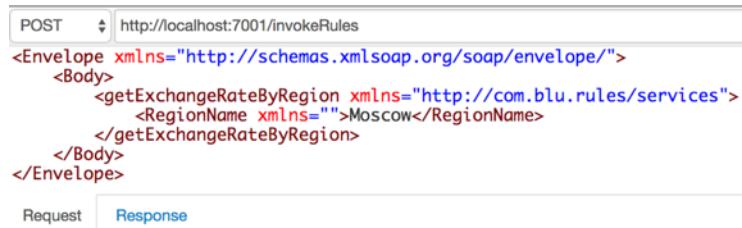


Figure 5.28

You should get the following response from the server.



Figure 5.29

Pay attention to the response time. It takes **129** ms to return the response.

**Step 12.** Let's check the cache entries by the command `cache -scan` in IgniteVisor.

```
[visor> cache -scan
Time of the snapshot: 2018-09-13 15:33:17
+-----+
| # |      Name(@)      | Mode      | Size (Heap / Off-heap) |
+-----+
| 0 | exchangeRate(@c0) | PARTITIONED | min: 1 (0 / 1)
|   |                   |           | avg: 1.00 (0.00 / 1.00)
|   |                   |           | max: 1 (0 / 1)
+-----+
[Choose cache number ('c' to cancel) [c]: 0
Entries in cache: exchangeRate
+-----+
| Key Class     | Key    | Value Class   | Value  |
+-----+
| java.lang.String | Moscow | java.lang.String | 71.7  |
+-----+
```

Figure 5.30

You should see that the exchange rate of the U.S. dollar for Moscow region is created into the cache. If we will re-invoke our web method again, spring cache manager will return the value from the cache.

Now recalled the web method again and watch the response time.



Figure 5.31

Now, the response time decreased to 6 ms.

**Step 13.** Let's update an exchange rate for some region and see what will happen with the cache. Invoke the `UpdateExchangeRate()` web method with the following parameters.

```

POST http://localhost:7001/invokeRules
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope">
  <Body>
    <updateExchangeRate xmlns="http://com.blu.rules/services">
      <RegionName>Moscow</RegionName>
      <Date>2015-05-03</Date>
      <USdollarRate>80.00</USdollarRate>
      <EuroRate>90.00</EuroRate>
      <BritishPoundRate>110.00</BritishPoundRate>
    </updateExchangeRate>
  </Body>
</Envelope>

```

Request      Response

Figure 5.32

The above web method call will update the database table row. U.S dollar rate changed to 80 ruble.

2015-05-02	111.01	77.1	102.03	Vladimir
2015-05-02	71.7	77.1	102.03	Tver
2015-05-02	80	77.1	102.03	MOSCOW

Figure 5.33

You should notice that the cache is empty if you scan the cache *exchangerate* into Ignite.

```

Entries in cache: exchangerate
+-----+
| Key Class | Key | Value Class | Value |
+-----+
| java.lang.String | Moscow | java.lang.String | 71.7 |
+-----+
[visor]> cache -scan
Time of the snapshot: 2018-09-13 15:45:31
+-----+
| # | Name(@) | Mode | Size (Heap / Off-heap) |
+-----+
| 0 | exchangeRate(@c0) | PARTITIONED | min: 0 (0 / 0)
| | | | avg: 0.00 (0.00 / 0.00)
| | | | max: 0 (0 / 0)
+-----+

```

Figure 5.34

The cache for the *Moscow* region has been evicted. Next time when the web method *getExchangeRateByRegion()* will be invoked again for *Moscow* region, the cache entry will appear in the cache *exchangerate*.

So, memoization is a specific type of caching. Memoization specifically involves caching the

return value of a function. If the function is get called with the same arguments as before, it just needs to return the cached result without actually doing the computation. We can use Spring caching with Apache Ignite to memoized any function.

This is the end of the section memoization. Next, we will go through an essential feature of the Apache Ignite – Web Session Clustering.

## Web session clustering

Web session clustering is a headache for every programmer and architects in the Microservice era. Breaking down a monolith into a collection of microservices has some wonderful benefits, but also has a few nasty downsides. One of them is *session management*.

Session management had always been a problem for enterprise Java. Native session management from application servers such as *Oracle WebLogic*, *IBM WebSphere* or *Tomcat* was error prone. Any of them could not replicate session very well to provide high availability of applications because always there is room for server maintenance.

Web session clustering is designed to provide a higher scalable solution for synchronizing session data across a cluster of web application servers. That means any web application can grow beyond a single server, in an application cluster. Apache Ignite provides an excellent way to store web sessions from the servlet containers to scale your web applications and handle extreme transaction load.

However, term session clustering may look like a complicated topic, but the basics are simple. Clustered architecture is used to solve one or more of the following problems in the computer world:

- A single server can't handle the high transactions load.
- A stateful application needs a way of preserving session data if its server fails.
- A developer needs the capability to deploy or update their applications without interrupting services.

A clustered architecture solves these above problems using a combination of the load balancer and some session replication. A classical architecture is shown in figure 5.35.

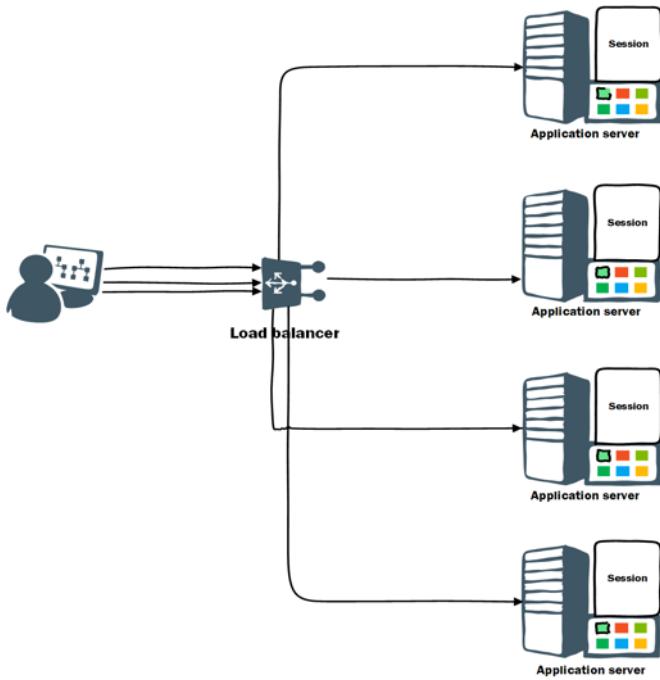


Figure 5.35



## Info:

We are talking about only standalone servlet containers such as Tomcat, Jetty, etc. Although, Oracle WebLogic cluster or IBM WebSphere provides session clustering and session replications, you can lose session data when you redeploy or update your application or doing server maintenance.

In the above architecture shown in figure 5.35, a load balancer distributes requests between multiple application server instances or servlet containers with algorithms such as round robin, reducing the load on each instance. The problem here is web session availability. A web session keeps an intermediate logical state between requests by using cookies and is generally bound to a particular application instance. There are a few classic solutions to solve this problem.

## Classic solution 1: using a sticky session

Sticky session<sup>80</sup> or session affinity ensures that the same application server handles request from the same user. Most of all web server, such as *Nginx*, *Apache Web server* can provide multiple algorithms and modules to provide such functionality. However, if the session is lost, and the user will face an unexpected result or error. Moreover, the sticky session is also unresponsive and can be slow.

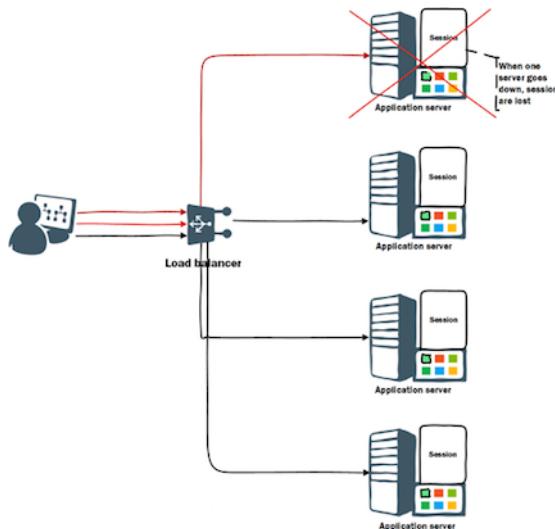


Figure 5.36

In all of these above cases, even for a given period of time, an application server that is still up and running may not accept new connections or handle new requests, or is entirely down. A client may be redirected to an alternate application server under these scenarios. The session affinity or sticky session mechanism will resume from that endpoint, but the original one will be abandoned. The sticky session is usually an optimization and not a replacement for long-lived connections.

## Classic solution 2: store sessions in DB

In this approach, the database can become both a bottleneck and a single point of failure. Servlet containers such as a Jetty or tomcat can take steps to reduce the load on the database. However, it is highly recommended not to use such a solution in a high loaded environment, because, in every request, the servlet container query the database.

<sup>80</sup>[https://en.wikipedia.org/wiki/Load\\_balancing\\_\(computing\)](https://en.wikipedia.org/wiki/Load_balancing_(computing))

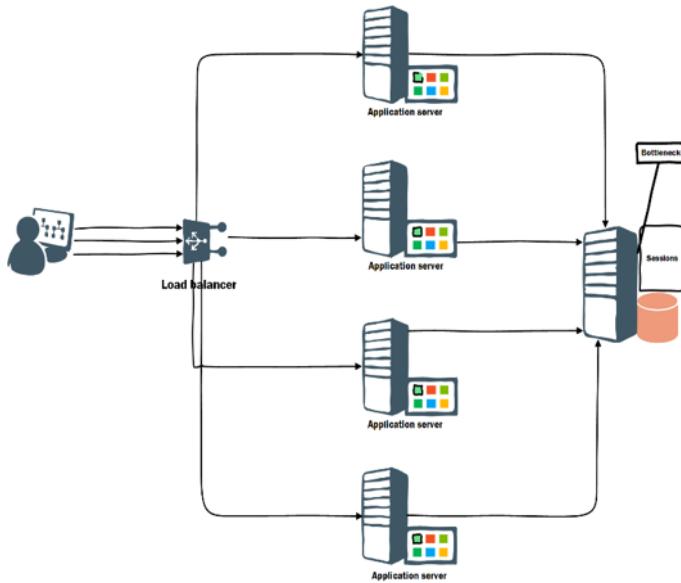


Figure 5.37

### Classic solution 3: use only stateless service

It is almost impossible to do all services stateless in the real world. For a simple e-commerce site, having a shopping cart or a simple captcha in your web form will force you to use stateful service.

### Distributed web session clustering

A solution here is to use distributed in-memory cache like Ignite for web session clustering, which maintains and store a copy of each created session, sharing them between all application server instances. In the event of an application server failure, the load balancer redirects to a new application server which has access to the user session in the Ignite cluster. The handoff to the new application server is seamless for the user.

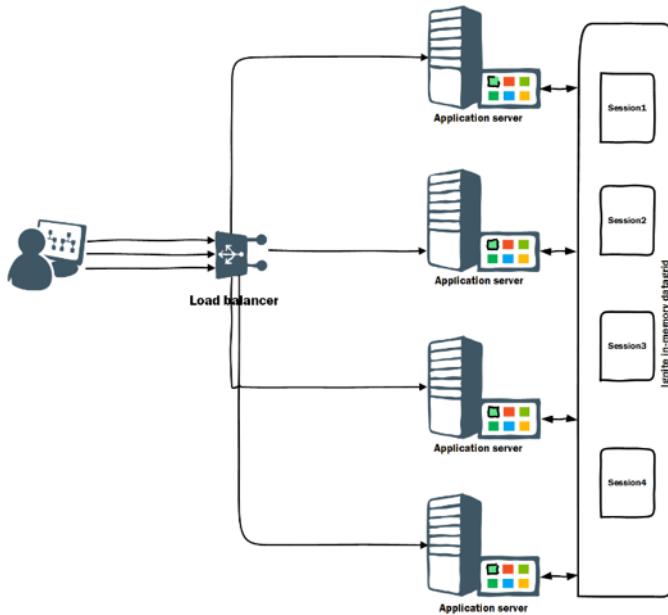


Figure 5.38

The above approach provides the highest level of high availability of a system and enjoyable customer experience. Since Ignite is an in-memory solution, the performance of this web session clustering and replication mechanism of a user web session is very high. Besides, Ignite need no modification to your application tier, so you can apply to Ignite web session clustering without modifying existing application code (only a few modifications in web.xml).

So far in this chapter, we have got a fundamental idea about web session clustering and how it could make our life easier in web development. Next, we are going to build a Spring MVC web application, which allows us to save some user session information in a **web session** stored in the Ignite in-memory cluster. Before we move on to start coding, let's have a detailed look at how the HTTP session works in a servlet container.

## HTTP Session

When a client visits the web application for the first time, the servlet container creates a new `Http Session` object, generate a long and unique id and stores it in a server's memory. The servlet container also set a `Cookie` in the `Set-Cookie` header of the `HTTP response` with `JSESSIONID` as cookie name and the unique session id as a cookie value.

The `HTTP session` lives until it has not been used for more than the sessions time, a setting you can specify in `web.xml`, which default to 30 minutes. So, when a client doesn't visit the

web application anymore for 30 minutes, then the servlet container remove the session from memory. Every subsequent request, even though with the cookie specified, will not have access to the same session anymore. The servlet container will create a new session.

In a nutshell,

- The HttpSession lives as long as the client is interacting with the web app with the same browser instance and the session hasn't timed out at the server side yet. It's been shared among all requests in the same session.
- The HttpServletRequest and HttpServletResponse live as long as the client has sent it until the complete response (the web page) arrives. It is not being shared elsewhere.
- Any Servlet, Filter, and Listener live as long as the web app lives. They are being shared among all requests in all sessions.

Let's prepare the sandbox for our web application. In this application we are going to use the following components:

1. Spring 4.3.16.RELEASE
2. Jetty-9.2.11
3. Ignite-2.6.0 version

To enable web session clustering in our web application with Apache Ignite, we need the following Ignite libraries:

1. ignite-core.jar
2. ignite-web.jar
3. ignite-log4j.jar
4. ignite-spring.jar

**Step 1.** Create a Maven *web* application project by using the following command.

**Listing 5.49**

---

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.3 -DinteractiveMode=false -DgroupId=com.bl\u00d7u.imdg -DartifactId=web-session-clustering-1
```

---

**Step 2.** Add these following libraries to the Maven project.

Listing 5.50

---

```
<properties>
    <springframework.version>4.3.16.RELEASE</springframework.version>
    <ignite.version>2.6.0</ignite.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${springframework.version}</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <!-- Ignite dependencies-->
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-core</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-web</artifactId>
        <version>${ignite.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.ignite</groupId>
        <artifactId>ignite-log4j</artifactId>
        <version>${ignite.version}</version>
    </dependency>
```

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>${ignite.version}</version>
</dependency>
</dependencies>
```

---

In the above maven project dependencies, we added a spring MVC library. To keep things simple, we use an embedded *Jetty servlet container* to execute our web application. Moreover, our web app is simple without any web form. In this web application, we have a couple of HTTP methods (`/putperson`), which puts session data into user session and another HTTP method (`/getperson`) to get the session data from the cache. If you are curious about the code, it's available at [GitHub project<sup>81</sup>](#).

**Step 3.** Let's create the session data. Create a new Java package `com.blu.imdg.wsession.bean` and add a Java class with name `Person` into it. Add the following contents into the newly added class.

**Listing 5.51**

```
public class Person implements Serializable{
    private String firstName;
    private int age;

    public Person(String firstName) {
        this.firstName = firstName;
    }

    public Person() {
    }
    // setter and getter methods are omitted
}
```

---

**Step 4.** In Spring's approach, a HTTP requests are handled by a controller. Create a MVC controller with name `WebSessionController` into the package `com.blu.imdg.wsession.controller`.

---

<sup>81</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-5/web-session-clustering>

**Listing 5.52**

---

```
@Controller
@RequestMapping("/")
public class WebSessionController {
    @RequestMapping(value = "/putperson", method = RequestMethod.GET)
    public String putSession(ModelMap model, HttpServletRequest request) {
        model.addAttribute("person", "Add Person to user session");
        String fName = request.getParameter("name");
        int age = Integer.valueOf(request.getParameter("age"));
        Person person = new Person();
        person.setFirstName(fName);
        person.setAge(age);
        // set the user session data
        request.getSession().setAttribute(fName, person);
        // return to the welcome.jsp view page
        return "welcome";
    }
    @RequestMapping(value = "/getperson", method = RequestMethod.GET)
    public String getSession(ModelMap model, HttpServletRequest request) {
        String fName = request.getParameter("name");
        // get the session data
        model.addAttribute("person", request.getSession().getAttribute(fName) != null ? "Age: " +
            + ((Person) request.getSession().getAttribute(fName)).getAge() : " Unknown");
        // view welcome.jsp
        return "welcome";
    }
}
```

---

This above controller is concise and simple, but there's plenty going on. Let's break it down step by step.

The `@Controller` annotation ensures that the given class is a Spring MVC controller.

When configuring Spring MVC, we need to specify the mappings between the HTTP requests and the handler methods. We use the `@RequestMapping` annotation to configure the mapping of the web requests. The `@RequestMapping` annotation can be applied to class-level and method-level in a controller. The class-level annotation maps a specific request path or pattern onto a controller. You can then apply additional method-level annotations to make mappings more specific to handler methods.

With the preceding code, request to `/putPerson` will be handled by the `putSession()` method while a request to the `/getPerson` will be handled by the `getSession()` method.

Also, the HTTP method `putSession()` parse the HTTP get parameters (name, age) from the URL and add the parameters values into the session. Similarly, HTTP method `getSession()` parse the parameter from the URL and prints the age of the Person into the web page.

**Step 5.** Next step is to create `spring-servlet.xml` to configure our MVC application. Add the `spring-servlet.xml` file into the `/webapp/WEB-INF` folder.

**Listing 5.53**

---

```
<context:component-scan base-package="com.blu.imdg.wsession.controller" />
<mvc:annotation-driven />
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/WEB-INF/views/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
```

---

In the above `spring-servlet.xml` configuration file, we have defined a `<context:component-scan>` tag. This will allow Spring to load all the components from the package `com.blu.imdg.wsession.controller` and all its child packages. Also, we defined a bean `viewResolver`. This bean will resolve the view and add prefix string `/WEB-INF/jsp/` and suffix `.jsp` to the view in `ModelAndView`. Note that, in our `WebSessionController` class, we have returned a `View` object with name `welcome`. This will be resolved to the path `/WEB-INF/jsp/welcome.jsp`.

**Step 6.** Create a new file `spring-ignite-cache.xml` and place it into the folder `/resources/META-INF`. Add the following content into the file.

**Listing 5.54**

---

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cache="http://www.springframework.org/schema/cache"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd ">

    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="peerClassLoadingEnabled" value="false"/>
        <property name="gridName" value="session-grid"/>
    
```

```
<!-- Enable client mode. -->
<property name="clientMode" value="false"/>
<property name="cacheConfiguration">
    <list>
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Cache mode. -->
            <property name="cacheMode" value="PARTITIONED"/>
            <property name="backups" value="1"/>
            <!-- Cache name. -->
            <property name="name" value="session-cache"/>
        </bean>
    </list>
</property>
<!-- Explicitly configure TCP discovery SPI to provide list of initial nodes. -->
<property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
        <property name="ipFinder">
            <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
                <property name="addresses">
                    <list>
                        <!-- In distributed environment, replace with actual host IP address. -->
                        <value>127.0.0.1:47500..47509</value>
                    </list>
                </property>
            </bean>
        </property>
    </bean>
</property>
</bean>
</beans>
```

File location of the above file is very **crucial**; for Ignite to start the cache instance, the path to the configuration file can be absolute, or relative to **IGNITE\_HOME**.

**Step 7.** This is the pre-final step to configure the Ignite web session clustering. Here, we add `web.xml` file into the `/WEB-INF` folder.

**Listing 5.55**

---

```
<web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>Ignite web session clustering example</display-name>
    <listener>
        <listener-class>org.apache.ignite.startup.servlet.ServletContextListenerStartup</listen\er-class>
    </listener>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <!--<param-value>classpath: META-INF/spring-ignite-cache.xml</param-value>-->
            <param-value>/WEB-INF/spring-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
<!-- Ignite -->
    <filter>
        <filter-name>IgniteWebSessionsFilter</filter-name>
        <filter-class>org.apache.ignite.cache.websession.WebSessionFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>IgniteWebSessionsFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
<!-- Ignite config from META-INF folder-->
    <context-param>
        <param-name>IgniteConfigurationFilePath</param-name>
        <param-value>spring-ignite-cache.xml</param-value>
    </context-param>
<!-- Specify the name of Ignite cache for web sessions. -->
    <context-param>
```

---

```

<param-name>IgniteWebSessionsCacheName</param-name>
<param-value>session-cache</param-value>
</context-param>
<context-param>
    <param-name>IgniteWebSessionsGridName</param-name>
    <param-value>session-grid</param-value>
</context-param>
</web-app>

```

---

From the above configuration, `DispatcherServlet` servlet will map the URL with pattern `/welcome.jsp`. One thing to note is the name of the servlet in the `<servlet-name>` tag in `web.xml` file. Once the `DispatcherServlet` is initialized, it will look for a file name `[servlet-name]-servlet.xml` in WEB-INF folder of web application. In this example, the Spring framework will look for a file called `spring-servlet.xml`.

We also initialized the Ignite servlet listener `ServletContextListenerStartup` into the `web.xml` listener section. The listener will start an Ignite node within the web application once the application starts, which will connect to the other nodes in the network forming a distributed cluster. In this preceding `web.xml` file, there are three most import configuration parameter (context param) relies on Ignite cluster as follows:

Parameter name	Description	Value
IgniteConfigurationFilePath	The path to Ignite configuration file (relative to META-INF folder or IGNITE_HOME). Make sure that, you have already put the file in META-INF folder	spring-ignite-cache.xml
IgniteWebSessionsCacheName	Name of the Ignite cache to use for web session caching. The name should be same as the Ignite cache name in <code>spring-ignite-cache.xml</code> .	session-cache
IgniteWebSessionsGridName	Name of the Ignite grid to use for web session caching. The name should be same as the Ignite cache name in <code>spring-ignite-cache.xml</code> .	session-grid

Please, recheck your configuration in the `web.xml` file twice if you got the following error during application startup.

Failed to find Spring configuration file, path provided should be either absolute, relative to `IGNITE_HOME`, or relative to `META-INF` folder.

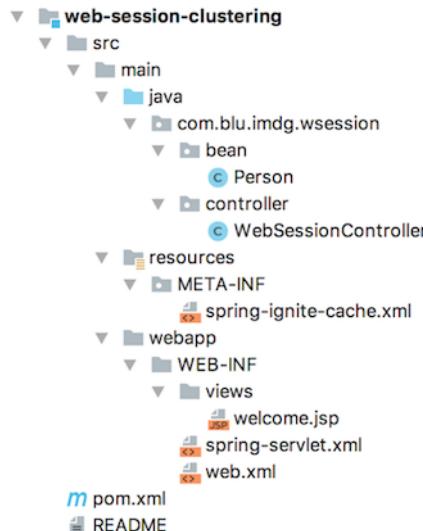
**Step 8.** Add a simple JSP page into the /webapp/WEB-INF/views folder and enter the following content.

**Listing 5.56**

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html/
4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Web session clustering example</title>
</head>
<body>
    Data : ${person}
</body>
</html>
```

The above JSP file is the view of the MVC pattern and relies on view technology. Spring parses the HTML template and evaluates the \${person} expression to render the value of the person parameter that was set in the controller during application runtime.

You should have the following project structure after completing the configuration.



**Figure 5.39**



## Info:

Note that, we didn't add any Ignite related code in the web application. Web session clustering in Ignite is fully *transparent* to the web application.

**Step 9.** Finally, we are ready to launch our web application. Let's run the web application in two separate terminals with different ports.

**Listing 5.57**

---

```
mvn -Djetty.port=8080 jetty:run
```

```
mvn -Djetty.port=8081 jetty:run
```

---

The following screenshot shows it in action.

The screenshot shows two terminal windows side-by-side. Both windows display log output from Ignite nodes starting up. The left window (port 8080) and right window (port 8081) show identical logs, indicating successful node startup and configuration of data regions and session clustering. The logs include details about the node ID, topology snapshot version, number of servers, clients, and heap size.

```
[22:42:07] Ignite node started OK [id=75195f1a, instance name=session-grid]
[22:42:07] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=3.6GB]
[22:42:07] ^-- Node [id=75195f1a-2614-49bf-8046-c1c110818009, clusterState=ACTIV
[22:42:07] [22:42:07] Data Regions Configured:
[22:42:07] ^-- default [initSize=256.0 MB, maxSize=3.2 GiB, persistenceEnabled=
false]
[INFO] Initializing Spring FrameworkServlet 'dispatcher'
[INFO] Started o.e.j.m.p.JettyWebAppContext@31ff6309[/test-session, file:/Users/shamim/Development/workshop/github/the-apache-ignite-book/chapters/chapter-5/web-session-clustering/src/main/webapp/,AVAILABLE]{file:/Users/shamim/Development/workshop/github/the-apache-ignite-book/chapters/chapter-5/web-session-clustering/src/main/webapp/}
[INFO] Started ServerConnector@5bdff890#7[HTTP/1.1]{0.0.0.0:8080}
[INFO] Started 9994sms
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 10 seconds.
[22:42:09] Topology snapshot [ver=2, servers=2, clients=0, CPUs=8, offheap=6.4GB, heap=7.1GB]
[22:42:09] ^-- Node [id=75195f1a-2614-49bf-8046-c1c110818009, clusterState=ACTIV
[22:42:09] [22:42:09] Data Regions Configured:
[22:42:09] ^-- default [initSize=256.0 MB, maxSize=3.2 GiB, persistenceEnabled=
false]
[INFO] Initializing Spring FrameworkServlet 'dispatcher'
[INFO] Started o.e.j.m.p.JettyWebAppContext@34465f1/test-session, file:/Users/shamim/Development/workshop/github/the-apache-ignite-book/chapters/chapter-5/web-session-clustering/src/main/webapp/,AVAILABLE}{file:/Users/shamim/Development/workshop/github/the-apache-ignite-book/chapters/chapter-5/web-session-clustering/src/main/webapp/}
[INFO] Started #9362ms
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 10 seconds.
```

**Figure 5.40**

Let's open a browser and go to the following URL.

**Listing 5.58**

---

<http://localhost:8080/test-session/putperson?name=shamim&age=38>

---

This will put a user-defined session in the Ignite cluster. If you open the *network* tab for developer view in Chrome/Firefox, you can also see the *JSESSIONID* of this request.

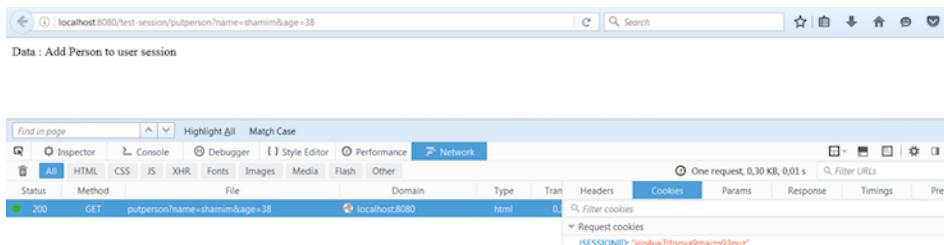


Figure 5.41

Now, check the entries in Ignite cache with *Ignitevisor*. Use the *cache -scan* command to display the entries.

```
Entries in cache: session-cache
+=====
|   Key Class    |       Key           |      Value Class      |
|                 |       Value          |                      |
+=====+
| java.lang.String | 1o8fvvnavqa12sv3i4lokplrb | o.a.i.i.websession.WebSessionEntity | WebSessionEntity [id=1o8fvvnavqa12sv3i4lokplrb, createTime=1537040909534, accessTime=1537040909566, maxInactiveInterval=1800, attributes=[shamim]] |
```

Figure 5.42

Impressing! Ignite stored the cache entity with the JSESSIONID and the value into the cache. We have two servlet container up and running, now, if we go to the next URL to get the value from the session as follows:

#### **Listing 5.59**

---

<http://localhost:8081/test-session/getperson?name=shamim>

---

It prints the Age: 38 in the web page. Note that, it's another instance of the servlet container, which doesn't hold our user session for Person Shamim. This web application looks for the session entries in the local session store, whenever it doesn't find the entries locally, it queries the Ignite cluster and gets the result from the cluster.

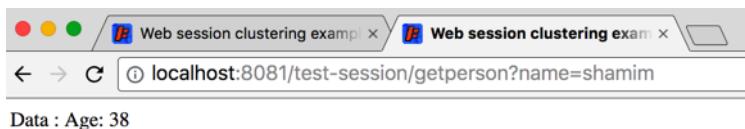


Figure 5.43

You can also use a Nginx load balancer to load balancing request between this two servlet container, and the result will be the same. No session data lose if one of the servlet containers goes down.

This is the end of the current section of this chapter. Here we learned that session clustering is used in currently most high traffic web applications. The feature offers an easy way to scale your application. Also, session clustering offers a way to handle application server failover and provides a way for the high availability of the application.

## Prepare the caches correctly

Any technology, no matter how advanced it is, will not be able to solve your problems if you implement it improperly. Caching, precisely when it comes to the use of a distributed caching, can only accelerate your application with the proper use and configurations of it. From this point of view, Apache Ignite is no different, and there are a few steps to consider before using it in the production environment. In this section, we describe various technics that can help you to plan and adequately use of Apache Ignite as cutting-edge caching technology.

**Do a proper capacity planning before using Ignite cluster.** Do paperwork for understanding the size of the cache, number of CPUs or how many JVMs will be required. Let's assume that you are using Hibernate as an ORM in 10 application servers and wish to use Ignite as an L2 cache. Calculate the total memory usages and the number of Ignite nodes you have to need for maintaining your SLA. An incorrect number of the Ignite nodes can become a bottleneck for your entire application. Please use the Apache Ignite [official documentation](#)<sup>82</sup> for preparing a system capacity planning.

**Select the best deployment option.** You can use Ignite as an embedded or a real cluster topology. All of them contains a few pros and cons. When Ignite is running in the same JVM (in embedded mode) with the application, the network roundtrip for getting data from the cache is minimum. However, in this case, Ignite uses the same JVM resources along with the application which can impact on the application performance. Moreover, in the embedded mode, if the application dies, the Ignite node also fails. On the other hand, when Ignite node is running on a separate JVM, there is a minimal network overhead for fetching the data from the cluster. So, if you have a web application with a small memory footprint, you can consider using Ignite node in the same JVM.

**Use on-heap caching for getting maximum performance.** By default, Ignite uses Java off-heap for storing cache entries. When using off-heap to store data, there is always some overhead of *de/serialization* of data. To mitigate the latency and get the maximum performance you can use on-heap caching. You should also take into account that, Java heap

---

<sup>82</sup><https://apacheignite.readme.io/docs/capacity-planning>

size is almost limited and there is a *GC (Garbage collection)* overhead whenever using on-heap caching. Therefore, consider using on-heap caching whenever you are using a small limited size of a cache, and the cache entries are almost constants.

**Use Atomic cache mode whenever possible.** If you do not need strong data consistency, consider using the atomic mode. In an atomic mode, each DML operation will either succeed or fail and, neither Read nor Write operation will lock the data. This mode gives a better performance than the transactional mode. An example of using an atomic cache configuration is shown below.

**Listing 5.60**

---

```
<property name="cacheConfiguration">
    <list>
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <property name="name" value="testCache" />
            <property name="atomicityMode" value="ATOMIC" />
        </bean>
    </list>
</property>
```

---

**Disable unnecessary internal event's notification.** Ignite has a rich event system to notify users/nodes about various events, including cache modification, eviction, compaction, topology changes, and a lot more. Since thousands of events per second are generated, it creates an additional load on the system. This can lead to significant performance degradation. Therefore, it is highly recommended to enable only those events that your application logic requires.

**Listing 5.61**

---

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    <!-- Enable events that you need and leave others disabled -->
    <property name="includeEventTypes">
        <list>
            <util:constant static-field="org.apache.ignite.events.EventType.EVT_TASK_STAR\
TED"/>
            <util:constant static-field="org.apache.ignite.events.EventType.EVT_TASK_FINI\
SHED"/>
            <util:constant static-field="org.apache.ignite.events.EventType.EVT_TASK_FAIL\
ED"/>
        </list>
    </property>
</bean>
```

---

**Turn off backups copy.** If you are using *PARTITIONED* cache and the data loss is not critical for you, consider disabling backups for the *PARTITIONED* cache. When backups are enabled, Ignite cache engine maintains a remote copy of each entry, which requires network exchanges. To turn off the backups copy, use the following cache configuration:

Listing 5.62

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set cache mode. -->
            <property name="cacheMode" value="PARTITIONED"/>
            <!-- Set number of backups to 0-->
            <property name="backups" value="0"/>
        </bean>
    </property>
</bean>
```

**Synchronizing the requests for the same key.** Let's explain by an example. Assume, your application has to handle 5000 requests per second. Most of them requested by one key. All the threads follow the following logic: If there is no value for the key in the cache, I query to the database. At the ends, each of the thread goes to the database and updates the value for the key into the cache. As a result, the application spends more times than if the cache was not at all. This is one of the common reasons when your application slows down whenever you are using a cache.

However, the solution to this problem is simple: synchronizing the requests for the same keys. From version 2.1, Apache Ignite support `@Cacheable` annotation with `sync` attributes which ensure that a single thread is forming the cache value. To achieve this, you have to add the `sync` attribute as follows:

Listing 5.63

```
@Cacheable(value = "exchangerate", sync = true)
public String getExchangerate(String region) {
}
```

**Turn off or tune durable memory.** Since version 2.1, Apache Ignite has its own persistence implementation. Unfortunately, persistence slows down the system. The *WAL* slows down the system even more. If you do not need the data durability, you can disable or turn off the *WAL* archiving. In Apache Ignite, starting from version 2.4, it is possible to disable *WAL* without restarting the entire cluster as shown below:

**Listing 5.64**

---

```
ALTER TABLE tableName NOLOGGING
ALTER TABLE tableName LOGGING
```

---

By the way, you can also tune the WAL logging level according to your requirements. By default, the WAL log level is enabled on DEFAULT mode, which guaranty the highest level of data durability. You can change the log to one of the following levels:

1. LOG\_ONLY.
2. BACKGROUND.
3. NONE.

Caching gives enormous performance benefits, saves unnecessary network roundtrips and reduce CPU costs. Many believe that caching is such an easy way to make everything faster and cooler. However, as practice shows, most often incorrect use of caching makes thing worse. Caching is the mechanism that only gives performance boosts when you use it *correctly*. So, remember this before implementing it in your project, take measurements before and after on all related cases.

## Summary

We have covered a lot of ground in this chapter by theory and practice. We have covered:

- The value of the intelligent caching and its best use cases.
- The concept of the 2<sup>nd</sup> level cache and how to implement it with Ignite.
- Memoization, which can improve the function execution time to ~4 ms.
- Web session clustering for high availability of the application.

At the end of this chapter, we also described how to prepare the cache properly to get the maximum performance from the caching layer.

## What's next?

In the next big chapter, we continue our study of the Ignite database features such as SQL queries, query cache, transactions and much more.

# Chapter 6. Database

Starting from version 2.0, Apache Ignite redesigned the storage engine and introduced a new memory-centric architecture that can be used as an In-memory database or a fully functional distributed database with disk durability and strong consistency. Ignite memory-centric storage helps to store data and indexes both in-memory and on-disk in the same data structure. It enables executing SQL queries over data stored in-memory and on-disk and delivers optimal performance while minimizing infrastructure costs. Apache Ignite data persistence on-disk mechanism is called *Native persistence* and is an *optional choice*. When native persistence is turned on, Ignite can store more data than can fit in the available memory, and act like a complete distributed SQL database. On the other hand, when the whole data set and indexes fit in memory, and the native persistence is disabled, Ignite function as an in-memory database supporting SQL, together with all the existing APIs for memory-only use cases.

Moreover, Apache Ignite supports [Spring Data<sup>83</sup>](#) that allows abstracting an underlying data storage from the application layer since 2.0 version. Spring Data also allows using out-of-the-box CRUD operations for accessing the database instead of writing boilerplate code. Additionally, you can use [Hibernate OGM<sup>84</sup>](#) that provides JPA (Java Persistence API) supports for Ignite database.

Another outstanding must-have feature for any enterprise-class system is the transaction. Support of *ACID* transactions guaranty you that, once changes to a dataset are made, the data is correct, and no data is lost. The modern enterprise often needs low latency transactional system with analytical capabilities on one single system with the ability of transaction. This type of Hybrid transactional processing often called *HTAP* (Hybrid transactional and analytical processing). HTAP enable you to process real-time analysis of your most recent data. In HTAP, data doesn't need to move from the operational database to separated data marts. In such a system, SQL support plays a vital role. De facto SQL is a natural language for data analysis and a productive language for writing queries. Therefore, in this chapter we are going to cover the following topics:

1. Ignite query capabilities.

---

<sup>83</sup><https://spring.io/projects/spring-data>

<sup>84</sup><http://hibernate.org/ogm/>

2. Spring Data integration.
3. Using Hibernate OGM with Ignite.
4. Distributed transactions.
5. Ignite persistence (native persistence and 3<sup>rd</sup> party).

However, before you start, you may find it helpful to know that:

- Each application presented in this chapter can be downloaded from the [GitHub repository<sup>85</sup>](#).
- There is no need for you to reconstruct each application unless you want to.

---

<sup>85</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6>

## Ignite tables & indexes configuration

Ignite stores data into caches under the hood. Tables and indexes created into Ignite are something like meta-data over cache entries which make it possible to run SQL queries over the data stored into the Ignite caches. You can configure tables and indexes in several ways in Apache Ignite:

**DDL statements.** Apache Ignite supports [Data Definition Language \(DDL\)](#)<sup>86</sup> statements for creating and removing SQL tables and indexes during run time. At the moment of writing this book, Apache Ignite provides following DDL statements:

1. *CREATE TABLE*. Create a new table and an underlying Ignite cache. A cache with a unique name will be created by the following format: `SQL_{SCHEMA_NAME}_{TABLE}`, where the `SCHEMA_NAME` will be always `PUBLIC`. For instance, if you create a table with name `EMP`, a cache with name `SQL_PUBLIC_EMP` will be also created and you can use the cache to work with it using key-value, compute or other APIs available in Ignite. It is worth mentioning that you can change the name of the cache created by the DDL statement.
2. *CREATE INDEX*. Create an index on the specified table. Indexes are stored in the internal *B+tree* data structures. The *B+tree* gets distributed across the cluster as well as the actual data. A cluster node stores a part of the index for the data it owns. See [chapter 4](#) for more information about internal structure of the Ignite indexes.
3. *ALTER TABLE*. Modify the structure of an existing table. You can add or remove several columns from a previously created table.
4. *DROP INDEX*. Drop an index on the specified table.
5. *DROP TABLE*. Drop the specified table and an underlying cache. This command drops an existing table in Ignite. The underlying distributed cache with all the data in it will also be destroyed.
6. *CREATE USER*. Creates a user with a given name and password.
7. *ALTER USER*. Changes an existing user's password.
8. *DROP USER*. Removes an existing user.

Note that, in the future Apache Ignite releases, you can expect to see support for additional, widely used DDL statements.

**Annotation-based.** In addition to the DDL commands, you can also annotate cache entry fields for SQL queries. Indexes, as well as queryable fields, can also be configured from Java code with the usage of `@QuerySqlField` annotation. All fields in Java class that will be involved in SQL clauses must have this annotation.

<sup>86</sup><https://apacheignite-sql.readme.io/docs/ddl>

**Listing 6.1**

---

```
public class Employee implements Serializable {
    /** Indexed field. Will be visible for SQL engine. */
    @QuerySqlField(index = true)
    private long empno;

    /** Queryable field. Will be visible for SQL engine. */
    @QuerySqlField
    private String ename;

    /** Will NOT be visible for SQL engine. */
    private int deptno;

    /**
     * Indexed field sorted in descending order.
     * Will be visible for SQL engine.
     */
    @QuerySqlField(index = true, descending = true)
    private float sal;
}
```

---

In the code above, both `empno` and `sal` are indexed fields. If you don't want to index a field but still need to use it in a SQL query, then the field has to be annotated as well omitting the `index = true` parameter. Such a field is called a *queryable field*. For instance, `ename` is defined as a queryable field above. Also, `deptno` is neither queryable nor indexed field and won't be accessible from SQL queries in Apache Ignite.

**QueryEntity Based Configuration.** You can also use a Spring XML file to configure indexes and *queryable fields* for caches. In this approach, you have to use `QueryEntity` class, which is a description of a cache entry (composed of key and value) in the way of how it must be indexed and can be queried. All concepts that are discussed as part of the annotation based configuration above are also valid for the `QueryEntity` based approach. Furthermore, fields that are configured with the `@QuerySqlField` annotation are internally turned into query entities. An example below shows how to define a query field and indexes by Spring XML configuration.

**Listing 6.2**

---

```
<?xml version="1.0" encoding="UTF-8"?>
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    <property name="name" value="EMP" />
    <!-- Configure query entities -->
    <property name="queryEntities">
        <list>
            <bean class="org.apache.ignite.cache.QueryEntity">
                <!-- Setting indexed type's key class -->
                <property name="keyType" value="java.lang.Long" />
                <!-- Key field name to be used in INSERT and SELECT queries -->
                <property name="keyFieldName" value="empno" />
                <!-- Setting indexed type's value class -->
                <property name="valueType" value="com.blu.imdg.Employee" />
                <!-- Defining fields that will be either indexed or queryable.
                     Indexed fields are added to 'indexes' list below.-->
                <property name="fields">
                    <map>
                        <entry key="ename" value="java.lang.String" />
                        <entry key="sal" value="java.lang.Long" />
                    </map>
                </property>
                <!-- Defining indexed fields.-->
                <property name="indexes">
                    <list>
                        <!-- Single field (aka. column) index -->
                        <bean class="org.apache.ignite.cache.QueryIndex">
                            <constructor-arg value="ename" />
                        </bean>
                    </list>
                </property>
            </bean>
        </list>
    </property>
</bean>
```

---

## Ignite queries

You can have several ways to query the data stored into the caches after creating the necessary tables or configuring caches for querying data. Apache Ignite provides a few

elegant ways like *Scan queries*, *SQL queries (ANSI-99)* and *Text queries* for retrieving data from Ignite caches. But the burning question would be, why Ignite provides API such as SQL queries over key-value stores? Because retrieving data using *Key* has some limitations: you have to handle the data querying manually at the application level, and a lack of integration with data analysis tools. Therefore, to address these shortcomings, Ignite provides declarative programming language like SQL for querying the data. Hence, SQL has become the universal interface for data retrieving and analysis.

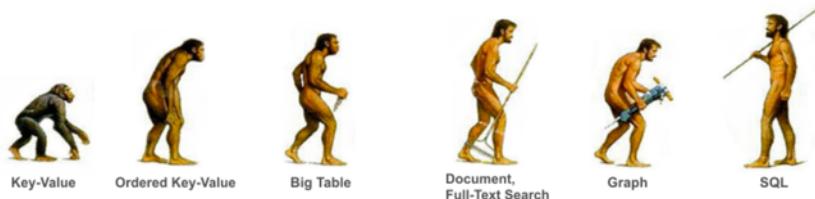


Figure 6.1

However, here are a few things you have to take into account about Ignite querying:

1. Internally Apache Ignite stores data in the key-value structure into caches.
2. Ignite loads data into memory before executing queries.
3. Ignite runs an H2 in-memory database engine on each node to execute the query under the hood.
4. You may write SQL like queries to query data on fields of our interest or retrieving the entire object.
5. You can use both native Apache Ignite SQL API's as well as JDBC or ODBC drivers to execute SQL queries.

Broadly categorizing, you may query Ignite database with the following ways:

1. SQL queries: SQL queries through *JDBC/ODBC*.
2. Ignite SQL API's: special or *native SQL API'S* ([SQLQuery<sup>87</sup>](#) and [SqlFieldsQuery<sup>88</sup>](#)).
3. Cache queries: predicate-base [Scan queries<sup>89</sup>](#) and [Text search<sup>90</sup>](#) queries.

In this last part of this section, we explore Ignite query capabilities one by one and explain by examples. All the queries using this section are representative of the typical database operations that are used in normal business processing.

---

<sup>87</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/cache/query/SqlQuery.html>

<sup>88</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/cache/query/SqlFieldsQuery.html>

<sup>89</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/cache/query/ScanQuery.html>

<sup>90</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/cache/query/TextQuery.html>

## SQL queries

Apache Ignite supports *JDBC/ODBC* connectivity to execute SQL queries from a long time before. However, this was too primitive to named the feature for production use. The release 2.0 version provides Data Definition Language (DDL) statements for creating and removing SQL tables and indexes at runtime which can be used by the JDBC/ODBC or Ignite SQL API's. With this functionality, you can define, what's more important, to *alter indexes* during runtime without a need to restart the Ignite cluster. This was a long time waiting feature for the Ignite users. Now users can leverage from this by using standard SQL commands like *CREATE* or *DROP* table/index and so on. In the subsequent releases, Ignite brings new SQL capabilities such as *DML* operations in JDBC/ODBC, *ALTER Table* command to add new table column at runtime. The most exciting feature was added in the 2.3 version: **SQLLine** tool. This fairly simple command line tool makes it easy to work with Apache Ignite as a database.



### Warning

JDBC driver for Apache Ignite was not entirely JDBC compliant prior to version 1.5. As a result, it was not possible to configure the JDBC driver connection pool (for instance, C3P0) to work with the Ignite driver.

Before we move on to start developing our applications, let's discuss some essential theories how Ignite handle SQL queries and joins. Historically, distributed JOINS against large dataset was very challenging because the overhead of an individual key lookup in different tables or caches is relatively large. Thus, most of the NoSQL database vendor doesn't support query JOINS. In this case, the user performs joins manually by combining multiple query results. However, Apache Ignite solves the problem of JOINS in a different fashion. Ignite provides a few different approaches for distributed joins: **Collocated joins**, **Non-collocated joins** and **cross-table join**.

When *collocated*, JOINs are executed on the nodes where the data is stored. The collocated JOINs omit data movement between cluster nodes expecting that the tables, that are already collocated, and the data is available locally on the node. This is the most efficient way to projection data in a distributed database like Apache Ignite. Ignite versions earlier than 1.7 can only perform collocated JOIN's, and for getting reliable query result, data should be collocated on the same node.

However, Apache Ignite version of 1.7 or later provides a new approach of distributed joins called *non-collocated* joins, where data collocation is no longer needed for the SQL

JOINS. There are other possibilities of cross joins of data from different datasets defined as *Cross-cache* queries. In cross-join queries, your data should be residing on the same node. Otherwise, you might get non-reliable query results. In the latter part of this section, we will details these two different joins by example.

We have to return to our caching topology of the Ignite again (chapter four) to help explain what we have discussed earlier. Caches are distributed in Ignite in two fashions: *Replicated* and *Partitioned*. All the nodes contain its primary dataset and the backup copy in the Partitioned mode. Datasets are replicated through the Ignite cluster in the Replicated mode. All the complexity of distributed JOINS comes in, whenever you are using the partitioned topology of the Ignite cluster. In the partitioned mode, Apache Ignite cannot give you the guarantee of getting a reliable query result if you are implementing any cross-join queries. For getting proper query result you have to collocate the related dataset in the same node or use the non-collocated distributed joins when it's tough or impossible to collocate all the data but you still need to execute a number of SQL queries such as *ad-hoc* queries over non-collocated caches.

With the Ignite SQL core concepts under your belt, you can tackle most of the SQL problems with Ignite. There are still more to cover, though, in this current subsection, we are going to cover the following topics:

1. DDL and DML statement to manipulate with Ignite tables.
2. Java application to retrieve data from the Ignite tables through pure JDBC API.
3. Configure JDBC connection pool to work with Ignite cluster.

#### Dataset:

The dataset that we are going to use is the *Employee* and *Department* entity. The structure of the department (dept) and Employee (emp) entity is straightforward, they are related to a one-to-many relationship with each other (see figure 6.2). This simple data structure will help us to cover all of our topics on SQL queries.

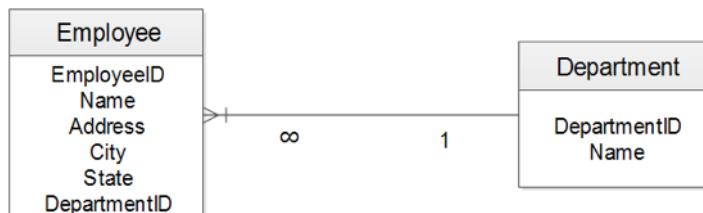


Figure 6.2

**Step 1.** Ensure that, one of your Apache Ignite node is up and running. If not, open the command shell and run the following command from the *IGNITE\_HOME* folder.

---

**Listing 6.3**

```
$ bin/ignite.sh
```

---

**Step 2.** Use the `SQLLINE` command line tool to connect to the Ignite cluster, run `sqlline.sh -u jdbc:ignite:thin:{host}` from your `IGNITE_HOME/bin` directory. For instance:

---

**Listing 6.4**

```
$ ./sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
```

---

**Step 3.** After connecting to the Ignite cluster, we can execute SQL statements to create or insert data into the tables. Now, let's create two tables, which we have described earlier.

---

**Listing 6.5**

```
CREATE TABLE dept
(
    deptno LONG,
    dname VARCHAR,
    loc  VARCHAR,
    CONSTRAINT pk_dept PRIMARY KEY (deptno)
);

CREATE TABLE emp
(
    empno  LONG,
    ename   VARCHAR,
    job    VARCHAR,
    mgr    INTEGER,
    hiredate DATE,
    sal     LONG,
    comm    LONG,
    deptno  LONG,
    CONSTRAINT pk_emp PRIMARY KEY (empno)
);
```

---

**Tip**

You can also use any JDBC compliant SQL tool such as *DBeaver* or *IntelliJ DataGrip* for executing DDL or DML statements. Please refer to [chapter 2](#) for more information about SQL tooling.

You can also download or copy the DDL and DML scripts for this project from [Github](#)<sup>91</sup>.

After executing the above two DDL statements, two new Ignite *caches* with name `SQL-PUBLIC_EMP` and `SQL_PUBLIC_DEPT` will be created and will also be defined two SQL *tables* on top of them. The caches stores the data in the *key-value* format while the table allows processing the data with SQL queries. For each cache, *Key* will be the field defined as a PRIMARY KEY in the CREATE table statement, and the rest of the fields stored as a *VALUE* in the underline cache.

By default, the cache `SQL_PUBLIC_EMP` and `SQL_PUBLIC_DEPT` will be *PARTITIONED*. KEYS of the caches will be `EMPNO` and `DEPTNO` value respectively. However, you can specify a PRIMARY KEY for the table that can be consist of multiple columns. The example below illustrates how to create the same table `EMP` with PRIMARY KEY consist of COLUMN `EMPNO` and `DEPTNO`.

#### Listing 6.6

---

```
CREATE TABLE emp
(
    empno  LONG,
    ename  VARCHAR,
    job    VARCHAR,
    mgr    INTEGER,
    hiredate DATE,
    sal    LONG,
    comm   LONG,
    deptno LONG,
    CONSTRAINT pk_emp PRIMARY KEY (empno, deptno)
);
```

---

A new table `EMP` with a composite primary key will be created once the above DDL statement executed. If you insert a few rows into `EMP` table and scan the cache by `ignitevisor` command line tool, you should have the following output.

---

<sup>91</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-2>

Entries in cache: SQL_PUBLIC_EMP		Key	Value
Value Class	Key Class		
I o.a.i.i.binary.BinaryObjectImpl   SQL_PUBLIC_EMP_fdd90af4_8840_4d11_850b_dd342e4eo55e.KEY [hash=2020639895, EMPNO=7698, DEPTNO=38]   o.a.i.i.binary.BinaryObjectImpl   SQL_PUBLIC_EMP_fdd90af4_8840_4d11_850b_dd342e4eo55e [hash=259097391, ENAME=BLAKE, JOB=MANAGER, MGR=7839, HIREDATE=1981-05-01, SAL=2850, COMM=null]	I o.a.i.i.binary.BinaryObjectImpl   SQL_PUBLIC_EMP_fdd90af4_8840_4d11_850b_dd342e4eo55e.KEY [hash=761930881, EMPNO=7782, DEPTNO=10]   o.a.i.i.binary.BinaryObjectImpl   SQL_PUBLIC_EMP_fdd90af4_8840_4d11_850b_dd342e4eo55e [hash=-165472427, ENAME=CLARK, JOB=MANAGER, MGR=7839, HIREDATE=1981-06-09, SAL=2450, COMM=null]	I o.a.i.i.binary.BinaryObjectImpl   SQL_PUBLIC_EMP_fdd90af4_8840_4d11_850b_dd342e4eo55e.KEY [hash=512379064, EMPNO=7839, DEPTNO=10]   o.a.i.i.binary.BinaryObjectImpl   SQL_PUBLIC_EMP_fdd90af4_8840_4d11_850b_dd342e4eo55e [hash=479490351, ENAME=KING, JOB=PRESIDENT, MGR=null, HIREDATE=1981-11-17, SAL=5000, COMM=null]	

Figure 6.3

The cache KEY value consist of two columns EMPNO and DEPTNO. Distinguishing from other RDBMS (for example, Oracle), Ignite table should have at least one PRIMARY KEY because Ignite is a Key-Value database and a VALUE must have a KEY related to it.



## Warning

If you try to create the EMP table without any PRIMARY KEY, you end up with a SQL exception like `java.sql.SQLException: No PRIMARY KEY defined for CREATE TABLE.`

So far, we have used only ANSI-99 compliant parameters to create tables. Apache Ignite DDL statement accepts additional distributed cache related parameters not defined by ANSI-99 SQL. Ignite cache related parameters are passed in the **WITH** clause of the DDL statement. If the **WITH** clause is omitted, then the cache will be created with default parameters set in the *CacheConfiguration* object. For instance, our *SQL\_PUBLIC\_EMP* cache created as a *PARTITIONED* cache by default. The following are the *key* optional parameters that you can specify in **WITH** clause of a CREATE TABLE DDL statement:

- **TEMPLATE=PARTITIONED|REPLICATED.** A template is an instance of the *CacheConfiguration* class registered with `Ignite.addCacheConfiguration` in the cluster. Use predefined *TEMPLATE=PARTITIONED* or *TEMPLATE=REPLICATED* templates to create the cache with the corresponding replication mode. By default, *TEMPLATE=PARTITIONED* replication mode is used if the template parameter is omitted or not explicitly specified.
- **BACKUPS=<number of backups>.** Sets the number of backup copies of data. If the *TEMPLATE* parameter is not set or the *BACKUPS* parameter is not set explicitly, a cache with 0 backup copies will be created.
- **ATOMICITY=ATOMIC|TRANSACTIONAL.** Sets the ATOMIC or TRANSACTIONAL mode for the underlying cache. By default, *ATOMICITY=ATOMIC* mode will be enabled for the newly created cache.

- CACHE\_NAME=<custom name of the new cache>. The name of the underlying cache created by the command.
- WRITE\_SYNCHRONIZATION\_MODE=PRIMARY\_SYNC|FULL\_SYNC|FULL\_ASYNC. Sets the write synchronization mode for the underlying cache. If neither this nor the TEMPLATE parameter is set, then the cache will be created with the FULL\_SYNC mode.
- AFFINITY\_KEY=<affinity key column name>. Specify an affinity key name which is a column of the PRIMARY KEY constraint. We detail the AFFINITY\_KEY by example in *Collocated distributed Joins* section of this chapter.

The next example of the DDL statement shows how to create the same *EMP* table with some cache related parameters:

**Listing 6.7**

---

```
CREATE TABLE IF NOT EXISTS EMP
(
    empno  LONG,
    ename  VARCHAR,
    job    VARCHAR,
    mgr    INTEGER,
    hiredate DATE,
    sal    LONG,
    comm   LONG,
    deptno LONG,
    CONSTRAINT pk_emp PRIMARY KEY (empno, deptno)
) WITH "template=replicated,backups=1,affinity_key=deptno,CACHE_NAME=MySuperEMPCache";
```

---

The above CREATE TABLE DDL statement creates a table called *EMP* which has eight columns. The DDL statement also creates an underlining cache called *MySuperEMPCache* with *REPLICATED* mode which has one backup copy. However, backup copy is *redundant* for replicated cache, because in replicated mode every node contains a copy of the replicated cache.

Please refer to the Ignite documentation<sup>92</sup> for additional parameters. Note that in each release, Ignite improves the DDL statements with new constraints and capabilities.

**Step 4.** Now, create a unique index as shown below.

---

<sup>92</sup><https://apacheignite-sql.readme.io/docs/create-table>

**Listing 6.8**

---

```
CREATE INDEX ename_idx ON emp (ename);
```

---

The above statement will create a unique index on column *ename* on *EMP* table.

**Step 5.** Next, insert some example data into two tables. Use the following DML statements to insert data:

**Listing 6.9**

---

```
insert into dept (deptno, dname, loc) values (10, 'ACCOUNTING', 'NEW YORK');
insert into dept (deptno, dname, loc) values(20, 'RESEARCH', 'DALLAS');
insert into dept (deptno, dname, loc) values(30, 'SALES', 'CHICAGO');
insert into emp (empno, ename, job, mgr, hiredate, sal, comm, deptno) values(7839, 'KING'\n, 'PRESIDENT', null, to_date('17-11-1981','dd-mm-yyyy'), 5000, null, 10);
insert into emp (empno, ename, job, mgr, hiredate, sal, comm, deptno) values( 7698, 'BLAK\N
E', 'MANAGER', 7839, to_date('1-5-1981','dd-mm-yyyy'), 2850, null, 30);
insert into emp (empno, ename, job, mgr, hiredate, sal, comm, deptno) values(7782, 'CLARK\N
', 'MANAGER', 7839, to_date('9-6-1981','dd-mm-yyyy'), 2450, null, 10);
```

---

**Step 6.** Create a new Maven project or clone it from the book [Github<sup>93</sup>](#) repository.

**Listing 6.10**

---

```
mvn archetype:generate -DgroupId=com.blu.imdg -DartifactId=jdbc-thin -DarchetypeArtifact\N
d=maven-archetype-quickstart -DinteractiveMode=false
```

---

Add the following *ignite-core* dependency into the Maven project.

**Listing 6.11**

---

```
<dependency>
  <groupId>org.apache.ignite</groupId>
  <artifactId>ignite-core</artifactId>
  <version>YOUR_IGNITE_VERSION</version>
</dependency>
```

---

**Step 7.** I'll start by creating a Java class using plain old JDBC to interact with the Ignite tables. Then, I'll make improvements to this approach by adding data sources and connection pool. Create a new Java class with name *JDBCThinClientExample* and add the following content on it.

---

<sup>93</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/jdbc-thin>

Listing 6.12

```
public class JDBCThinClientExample {  
    private static final String JDBC_URL = "jdbc:ignite:thin://127.0.0.1";  
    private static final String SQL_SALARY_MORE_THAN_3000 = "SELECT ename, deptno, job, s\\  
al\\n" +  
        "FROM EMP\\n" +  
        "WHERE (1.25 * sal) > 3000 ;";  
    private static final String SQL_SALARY_SMALLEST = "SELECT ename, deptno, sal \\n" +  
        "FROM emp \\n" +  
        "WHERE sal IN \\n" +  
        "( SELECT MIN(sal) \\n" +  
        "FROM emp \\n" +  
        "GROUP BY deptno \\n" +  
        ");";  
    private static final String SQL_SALARY_MORE_THAN_AVG = "SELECT ename, deptno, sal \\n\\  
" +  
        "FROM emp \\n" +  
        "WHERE sal > \\n" +  
        "( SELECT AVG(sal) \\n" +  
        "FROM emp \\n" +  
        ");";  
  
    public static void main(String[] args) throws ClassNotFoundException {  
        System.out.println("JDBC thin client example!!");  
        // Register JDBC driver.  
        Class.forName("org.apache.ignite.IgniteDBCThinDriver");  
  
        try (Connection conn = DriverManager.getConnection(JDBC_URL)) {  
  
            // Execute an query to list the employees whose salary is more than 3000 afte\\  
r giving 25%  
            ResultSet rs = conn.createStatement().executeQuery(SQL_SALARY_MORE_THAN_3000);  
  
            System.out.println("A list of the employees whose salary is more than 3000 af\\  
ter giving 25% !!!");  
            while (rs.next()) {  
                System.out.println("Employee name: " + rs.getString(1) + " Department: " \\  
+ rs.getString(2) + " Job:" + rs.getString(3) + " Salary: " + rs.getString(4));  
            }  
  
            // Execute an query to display the name (first name and last name), salary, \\  
department id for those employees who earn such amount of salary which is the smallest sa\\  
lary of any of the departments.
```

```
rs = conn.createStatement().executeQuery(SQL_SALARY_SMALLEST);

System.out.println("A list of employees who earn such amount of salary which \
is the smallest salary of any of the departments!!!");
while (rs.next()) {
    System.out.println("Employee name: " + rs.getString(1) + " Department: " \
+ rs.getString(2) + " Salary: " + rs.getString(3));
}

//Execute an query query to display the employee id, employee name (first nam\
e and last name ) for all employees who earn more than the average salary.
rs = conn.createStatement().executeQuery(SQL_SALARY_MORE_THAN_AVG);

System.out.println("A list of employees who earn more than the average salary\
!!!");
while (rs.next()) {
    System.out.println("Employee name: " + rs.getString(1) + " Department: " \
+ rs.getString(2) + " Salary: " + rs.getString(3));
}

} catch (SQLException e) {
    e.printStackTrace();
}

}
```

The above `JDBCThinClientExample` uses standard JDBC programming technics and displays how JDBC code is typically written.

From the beginning, we have declared *JDBC URL* to connect to the Ignite database and a few *SQL statements* that we are planning to execute later. In the `main()` method, first, we register the Ignite JDBC driver, and then we establish a connection to the Ignite database by the JDBC URL. When this class first attempts to establish a connection, it automatically loads the `IgniteJdbcThinDriver` drivers found within the classpath.



## Tip

We use `Class.forName()` method to register the JDBC driver. This is the most common way to load the driver's class file into the memory dynamically. There are a few other ways to register and a driver class automatically.

Next, we create a `Statement` object to execute a SQL statement. The `createStatement().executeQuery()` executes an SQL statement and returns a `ResultSet` object. We executes the following three SQL queries:

- fetch a list of the employees whose salary is over \$3000 after raising the 25%.
- fetch a list of employees who earn such amount of salary which is the *smallest* salary of any of the departments.
- fetch a list of employees who earn more than the *average* salary.

The Java `ResultSet` interface contains dozens of methods for getting the data from the current row. We iterate through the result set object and use `getString(columnIndex)` method to retrieve the data by specified column index and prints out into the console.

**Step 8.** Now, we are ready to build and run our application. To build the project, type the following command into the terminal.

**Listing 6.13**

---

```
mvn clean install
```

---

We are going to use a Maven `one-jar` plugin. Run the Java application from the folder *chapter-6/jdbc-thin* by the following command.

**Listing 6.14**

---

```
java -jar ./target/jdbc-thin-runnable.jar
```

---

This above command will start the application and you should see some output like this.

```
A list of the employees whose salary is more than 3000 after giving 25% !!!  
Employee name: BLAKE Department: 30 Job:MANAGER Salary: 2850  
Employee name: CLARK Department: 10 Job:MANAGER Salary: 2450  
Employee name: KING Department: 10 Job:PRESIDENT Salary: 5000  
A list of employees who earn such amount of salary which is the smallest salary of any of the departments!!!  
Employee name: BLAKE Department: 30 Salary: 2850  
Employee name: CLARK Department: 10 Salary: 2450  
A list of employees who earn more than the average salary!!!  
Employee name: KING Department: 10 Salary: 5000
```

**Figure 6.4**

The application is very primitive but shows you the full power of using JDBC over data stored in Ignite database. Note that, Ignite does not have full flagged transaction supports in version 2.6. At this moment, Ignite SQL queries provides only atomic mode transaction, which means that if there is a transaction that has already committed value A, but that is

still committing value B then an SQL query running in parallel will see A, but will not see B. In the upcoming Ignite release (version 2.7) a beta version of the MVCC and Transaction will be introduced.

The story is not over here, with Ignite JDBC thin driver you can configure connection pool to work with Ignite database. A *connection pool* is a cache of database connection objects. The objects represent physical database connections that can be used by an application to connect to a database. During run time, the application requests a connection from the pool. If the pool contains a connection that can satisfy the request, it returns the connection to the application. If no connections are found, a new connection is created and returned to the application. The application uses the connection to perform some work on the database and then returns the object back to the pool. The connection is then available for the next connection request.

Connection pools promote the reuse of connection objects and reduce the number of times that connection objects are created. Connection pools significantly improve performance for *database-intensive* applications because creating connection objects is costly both in terms of time and resources. Connection pooling is generally transparent to the developer. There is only one thing you need to do when you are using pooled connections:

- Use a `DataSource` object rather than the `DriverManager` class to get a connection.

The following code gets a `DataSource` object that produces connections to the Ignite database and uses to retrieve employees from the `EMP` table.

#### Listing 6.15

---

```
ComboPooledDataSource cpds = new ComboPooledDataSource();
try {
    cpds.setDriverClass("org.apache.ignite.IgniteJdbcThinDriver");
    cpds.setJdbcUrl(JDBC_URL);

    // pool size setting (optional), c3p0 can work with defaults
    cpds.setMinPoolSize(5);
    cpds.setAcquireIncrement(5);
    cpds.setMaxPoolSize(20);
    cpds.setMaxStatements(180);

} catch (PropertyVetoException e) {
    e.printStackTrace();
}
```

---

In the above pseudo code we set the minimum pool size to 5 and maximum size to 20. You can run the application by the following command from the `chapter-6\jdbc-thin` directory.

**Listing 6.16**

---

```
java -jar ./target/c3p0-runnable.jar
```

---

This command uses a connection pool to connect to the database and produces the same result as before. The full source code of the application is available at [Github<sup>94</sup>](#) repository.



## Tip

You can also use high-performance JDBC connection pool like [HikariCP<sup>95</sup>](#). See the `HikariCPDataSourceExample` Java class in the project repository for more details information.

Also, it is possible to set *multiple* JDBC connection endpoints in the connection string to enable automatic failover. JDBC driver randomly picks an address from the list to connect it. If the original connection fails, the driver will select another address from the list until the connection is restored. JDBC stops reconnecting and throws an Exception if all the endpoints are unreachable. A pseudo example below shows how to set multiple connection endpoints via a connection string.

**Listing 6.17**

---

```
HikariDataSource hds = new HikariDataSource();
hds.setJdbcUrl("jdbc:ignite:thin://127.0.0.1:1008,192.168.51.11:1008");
hds.setDriverClassName("org.apache.ignite.IgniteJdbcThinDriver");
```

---

The node that the driver connects to forward the SQL queries to the rest of the Ignite cluster nodes is acting as a gateway or mediator for the driver. This node cater for the query distribution and result aggregation in case of the distributed cache. Then the result is sent back to the client application. In this approach, the client application is not a part of the Ignite cluster and does not have any information about Ignite cluster topology. As a result, JDBC thin client has a *performance drawback* and requires *extra network round trip* through the gateway node for executing SQL queries. To overcome this issue, Ignite provides JDBC Client Driver that interacts with the cluster through an Ignite client node.

---

<sup>94</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/jdbc-thin>

<sup>95</sup><https://github.com/brettwooldridge/HikariCP>

**JDBC Client Driver.** The *JDBC Client Driver* connects to the Ignite cluster using its own fully established client node connection and *always be a part* of the Ignite cluster. The driver is more *heavyweight* than the thin client and contains information about the cluster topology and does not use any mediator or gateway node to execute queries. In addition, it can execute queries and aggregate the result of the queries from the application side. This driver requires the following configurations:

1. ignite-core-x.x.x, ignite-indexing-x.x.x and ignite-spring-x.x.x libraries in classpath.
2. A complete Spring XML configuration file on behalf of the Ignite client.

Following are the main advantages and disadvantages of the JDBC client driver, let's discuss them sequentially.

#### *Advantages:*

1. Connect as a client node to the cluster and contains cluster topology information.
2. Can send SQL queries to the specified node.
3. Can execute queries against local caches.
4. Can be used for bulk operation or streaming to load data into the cluster

#### *Disadvantages:*

1. Doesn't support all new SQL features.
2. Does not provide any ability to work with multiple tables at a time.

In this subsection we introduce some of the Ignite JDBC client driver functionality:

- We develop a new application from scratch, using JDBC client driver.
- We then configure the Spring XML configuration file and connect to the Ignite cluster.
- Finally, we execute some SQL queries to query the database.

We assume that you already have up and running an Ignite node and already created the *EMP* table with some rows and indexes that we have done on the previous section of this chapter.

**Step 1.** Create a Maven project or clone the `JDBC-client-driver` project from the [GitHub<sup>96</sup>](#) repository.

**Step 2.** Add the following libraries into the Maven `pom.xml` file.

---

<sup>96</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/jdbc-client-driver>

Listing 6.18

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>${ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-indexing</artifactId>
    <version>${ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>${ignite.version}</version>
</dependency>
```

---

**Step 3.** Create an XML file with name `ignite-jdbc.xml` under the `src\main\resources` folder. Add the following contents into the file.

Listing 6.19

---

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="grid.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="clientMode" value="true"/>
        <property name="peerClassLoadingEnabled" value="false"/>
        <property name="marshaller">
            <bean class="org.apache.ignite.internal.binary.BinaryMarshaller" />
        </property>
        <property name="discoverySpi">
            <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
                <property name="ipFinder">
                    <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDisco\
veryVmIpFinder">
                        <property name="addresses">
                            <list>
                                <value>127.0.0.1:47500..47549</value>
                            </list>
                        </property>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

---

```
</property>
</bean>
</property>
</bean>
</property>
</bean>
</beans>
```

---

We specified the client node runtime configuration. We made sure that the node is a client node and distributed class loading is not enabled. However, we explicitly enable the binary marshaller which lets you serialize and deserialize all objects in the binary format.

**Step 4.** Create a new Java class into the `com.blu.imdg` package with name `JDBCClientDriverExample`. Add the following content to the Java class.

**Listing 6.20**

```
public class JDBCClientDriverExample {
    private static final String SQL_SALARY_MORE_THAN_3000 = "SELECT ename, deptno, job, s\\
al\\n" +
        "FROM EMP\\n" +
        "WHERE (1.25 * sal) > 3000 ;";
    private static final String SQL_SALARY_SMALLEST="SELECT ename, deptno, sal \\n" +
        "FROM emp \\n" +
        "WHERE sal IN \\n" +
        "( SELECT MIN(sal) \\n" +
        "FROM emp \\n" +
        "GROUP BY deptno \\n" +
        ");";
    private static final String SQL_SALARY_MORE_THAN_AVG="SELECT ename, deptno, sal \\n" +
        "FROM emp \\n" +
        "WHERE sal > \\n" +
        "( SELECT AVG(sal) \\n" +
        "FROM emp \\n" +
        ");";
    public static void main(String[] args) throws ClassNotFoundException{
        System.out.println("JDBC client driver example!!!");
        // Register JDBC driver.
        Class.forName("org.apache.ignite.IgniteJdbcDriver");

        try{Connection conn = DriverManager.getConnection("jdbc:ignite:cfg://cache=SQL_PU\\
BLIC_EMP@file:///PHYSICAL_PATH/ignite-jdbc.xml");}
    }
}
```

```
ResultSet rs = conn.createStatement().executeQuery(SQL_SALARY_MORE_THAN_3000);

System.out.println("A list of the employees whose salary is more than 3000 af-
ter giving 25% !!!");
while (rs.next()){
    System.out.println("Employee name: " + rs.getString(1) + " Department: "+\
rs.getString(2) + " Job:" + rs.getString(3) + " Salary: " + rs.getString(4));
}

rs = conn.createStatement().executeQuery(SQL_SALARY_SMALLEST);

System.out.println("A list of employees who earn such amount of salary which \
is the smallest salary of any of the departments!!!");
while (rs.next()){
    System.out.println("Employee name: " + rs.getString(1) + " Department: "+\
rs.getString(2) + " Salary: " + rs.getString(3));
}

rs = conn.createStatement().executeQuery(SQL_SALARY_MORE_THAN_AVG);

System.out.println("A list of employees who earn more than the average salary\
!!!");
while (rs.next()){
    System.out.println("Employee name: " + rs.getString(1) + " Department: "+\
rs.getString(2) + " Salary: " + rs.getString(3));
}

} catch (SQLException e){
    e.printStackTrace();
}

}
```

The execution flow of the above class is similar to JDBC thin client. The main difference is the pattern of the JDBC connection URL that we have used here. The JDBC connection string has the following pattern:

**Listing 6.21**

---

```
jdbc:ignite:cfg://[<params>@]<config_url>
```

---

where,

- <config\_url>: is required and represents any valid URL that points to an Ignite configuration file for Ignite client node. At the moment of writing this book, `config_url` requires an absolute path of the Spring XML file.
- <params>: is optional and has the following format:

param1=value1:param2=value2:...:paramN=valueN

An example of the JDBC connection string is shown below:

```
jdbc:ignite:cfg://cache=SQL_PUBLIC_EMP@file:///Users/tms/chapters/chapter-6/jdbc-client-1  
driver/src/main/resources/ignite-jdbc.xml
```

Where,

- cache= name of the cache. SQL queries will be executed against the cache. If the cache name is not defined, the default cache will be used.
- @config\_url = absolute path of the Ignite client XML configuration.



## Warning

If you create tables in the **PUBLIC schema** (through JDBC/SQLLINE), define the full schema name into the cache name parameter.

**Step 5.** Build the project with Maven which creates a single executable JAR that contains all the necessary dependencies, classes, and resources.

**Listing 6.22**

---

```
mvn clean install
```

---

**Step 6.** Now you can run the JAR file:

**Listing 6.23**

---

```
java -jar ./target/jdbc-client-driver-runnable.jar
```

---

You should see the following output into the terminal:

```
[20:52:56] Ignite node started OK (id=9c824523, instance name=ignite-jdbc-driver-40baf06e\\
-e9c5-4e91-afeb-d929a0a717bf)
[20:52:56] Topology snapshot [ver=2, servers=1, clients=1, CPUs=8, offheap=3.2GB, heap=4.\\
6GB]
[20:52:56]  ^-- Node [id=9C824523-4378-439D-8ABD-A9213D4D3432, clusterState=ACTIVE]
A list of the employees whose salary is more than 3000 after giving 25% !!!
Employee name: CLARK Department: 10 Job:MANAGER Salary: 2450
Employee name: KING Department: 10 Job:PRESIDENT Salary: 5000
Employee name: BLAKE Department: 30 Job:MANAGER Salary: 2850
A list of employees who earn such amount of salary which is the smallest salary of any of\\
the departments!!!
Employee name: CLARK Department: 10 Salary: 2450
Employee name: BLAKE Department: 30 Salary: 2850
A list of employees who earn more than the average salary!!!
Employee name: KING Department: 10 Salary: 5000
[20:52:56] Ignite node stopped OK [name=ignite-jdbc-driver-40baf06e-e9c5-4e91-afeb-d929a0\\
a717bf, uptime=00:00:00.174]
```

The above output shows the result after executing the JDBCClientDriverExample application. Our application with an Ignite node has been started in client mode and joined to the cluster. Then the application executes three different SQL queries which we discussed earlier in this chapter. Finally, the client node stopped.

JDBC client driver has many features for configuring and customizing the JDBC connection URL, for instance, you can load data into Ignite cluster in a streaming mode (bulk mode) using the JDBC client driver. Also, the JDBC connection URL supports the following optional parameters:

Properties	Description	Default
cache	Cache name. If it is not defined, the default cache will be used.	
nodeId	ID of node where query will be executed. It can be useful for querying through local caches.	
local	Query will be executed only on a local node. Use this parameter with nodeId parameter in order to limit data set by specified node.	false
collocated	Flag that is used for optimization purposes. Whenever Ignite executes a distributed query, it sends sub-queries to individual cluster members. If you know in advance that the elements of your query selection are collocated together on the same node, Ignite can make significant performance and network optimizations.	false
distributedJoins	Allows distributed joins for non collocated data.	false
streaming	Turns on bulk data load mode via INSERT statements for this connection.	false
lazy	Lazy query execution. By default, Ignite attempts to fetch the whole query result set to memory, and send it to the client. For small and medium result sets, this provides optimal performance and minimize duration of internal database.	false

In the above table, we have skipped a few parameters related to the data streaming. Please refer to the [User Guide](#)<sup>97</sup> for more detail.

In addition to the JDBC drivers offered by the Ignite, Java developers can also use native SQL API's to query and modify data stored in the Ignite database. In the next section, we introduce you with the Ignite two different SQL API's to work with the Ignite database.

## Query API

Apache Ignite provides two different *Query API*'s to execute SQL queries over cache entries.

Name	Description
org.apache.ignite.cache.query.SqlQuery	Class that always returns the <i>whole key and the value</i> of the objects. It's very much similar to the Hibernate HQL. For example, you can run the following query to get all the employees with salaries between 1000 and 2000. <code>SqlQuery qry = new SqlQuery&lt;&gt;(Employee.class, "sal &gt; 1000 and sal &lt;= 2000");</code>

<sup>97</sup><https://apacheignite-sql.readme.io/docs/jdbc-client-driver>

Name	Description
org.apache.ignite.cache.query.SqlFieldsQuery	This query can return <i>specific fields of data</i> based on SQL select clause. You can choose to select only specific fields in order to minimize network and serialization overhead. Also it is useful when you want to execute some aggregate query. For example, <pre>SqlFieldsQuery qry = new SqlFieldsQuery("select avg(e.sal), d.dname " + "from Employee e, `"+ DEPARTMENT_CACHE_NAME + "`.Department d " + "where e.deptno = d.deptno " + "group by d.dname " + "having avg(e.sal) &gt; ?");</pre>

In the following section, we start from the `SqlFieldsQuery` and then explain `SqlQuery` API because, in real-life scenarios, you often work with the specified fields of any objects rather than the whole object from the database.

## SqlFieldsQuery

So far in this chapter, we have make use of SQL statements to query and modify data stored in the database. We create the table through standard DDL statements and use SELECT statements for querying the table. CREATE table DDL statement creates the table and appropriate cache into the database for further use. In this section, we are going to use the *Java specific annotation* to configure cache entries to use SQL queries against them.

Before using `SqlFieldsQuery` API to query the caches or tables, you have to do one of the following:

1. Create all the tables by using standard SQL DDL statements.
2. Use *Java specific annotation* to makes the entity fields accessible for the SQL queries.
3. Use a *query entity* description which tells Ignite how the entity must be indexed and can be queried.

Ignite provides `org.apache.ignite.cache.query.annotations.QuerySqlField` annotation to annotates Java class fields for SQL queries. All class *fields* that will be involved in SQL clauses must have this annotation. Let's start by looking at some examples. The next pseudo code shows the use of `@QuerySqlField` annotation.

**Listing 6.24**

---

```
public class Employee implements Serializable {
    @QuerySqlField
    private Integer empno;
    @QuerySqlField
    private String ename;
    @QuerySqlField
    private LocalDate hiredate;
    private Integer sal;
    @QuerySqlField
    private Integer deptno;
    //rest of the code is omitted
}
```

---

Let's have a detailed look at the above code. With the `@QuerySqlField` annotation, we have enabled the `empno`, `ename`, `hiredate` and `deptno` fields to be used by the SQL queries. Note that, property or field `sal` is not enabled for SQL queries. You can also use the `@QuerySqlField` annotation to *index* the field values for speeding up the query execution time. To create a single column index, you can annotate the field with `@QuerySqlField(index = true)`. This will create an index for the entity field. I create two indexes on fields: `empno` and `deptno` as shown in listing 6.25.

**Listing 6.25**

---

```
public class Employee implements Serializable {
    @QuerySqlField(index = true)
    private Integer empno;
    @QuerySqlField
    private String ename;
    private String job;
    @QuerySqlField
    private Integer mgr;
    @QuerySqlField
    private LocalDate hiredate;
    @QuerySqlField
    private Integer sal;
    @QuerySqlField(index = true)
    private Integer deptno;
    //rest of the part is omitted
}
```

---



## Warning

Indexes in Ignite has consumed places in memory. Also, each index needs to be updated separately, so your cache update can be slower if you have more indexes. You should index fields that are only participating in the *where clause* of the SQL query.

Also, it is possible to combine one or more indexes in one group to speed up the queries execution with a complex condition. In this case, you have to use `@QuerySqlField.Group` annotation. It is possible to put multiple `@QuerySqlField.Group` annotations into single `orderedGroups`, if you want the fields to participate in more than one group index.

**Listing 6.26**

---

```
public class Employee implements Serializable {
    /** Indexed in a group index with "hiredate and salary". */
    @QuerySqlField(orderedGroups = {
        @QuerySqlField.Group(
            name = "hiredate_salary_idx", order = 0, descending = true)
    })
    private LocalDate hireDate;
    /** Indexed separately and in a group index with "age". */
    @QuerySqlField(index = true, orderedGroups = {
        @QuerySqlField.Group(
            name = "hiredate_salary_idx", order = 3)
    })
    private Integer sal;
}
```

---



## Tip

Note that, there are two predefined fields for every entity: `_key` and `_val` which represents links to the whole key and value of the cache entries. These are useful, for example, when cache entry value is primitive and you want to filter by its value. For instance, execute a query like `SELECT * FROM Employee WHERE _key = 100`.

Now that, we have got the basics of using Java specific annotation to makes the entity fields accessible for the SQL queries. Let's quickly check some examples and see how they work. The code sample of this section can be found in [GitHub repository<sup>98</sup>](#).

---

<sup>98</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/sqlqueries>

**Step 1.** Clone or download the project from the GitHub repository. You need to add a couple of dependencies to the project to start: two for Ignite itself and one for the Spring library, which is used by Ignite under the covers:

Listing 6.27

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>${ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-indexing</artifactId>
    <version>${ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>${ignite.version}</version>
</dependency>
```

---

**Step 2.** Then, you need to define the Java classes with `@QuerySqlField` annotations on fields. You can imagine these classes as a JPA Entity class. Create two Java classes with name *Employee* and *Department*.

Listing 6.28

---

```
public class Employee implements Serializable {
    private static final AtomicInteger GENERATED_ID = new AtomicInteger();

    @QuerySqlField(index = true)
    private Integer empno;
    @QuerySqlField
    private String ename;
    @QuerySqlField
    private String job;
    @QuerySqlField
    private Integer mgr;
    @QuerySqlField
    private LocalDate hiredate;
    @QuerySqlField
    private Integer sal;
```

```
@QuerySqlField(index = true)
private Integer deptno;
//rest of the part is omitted
}
```

---

**Listing 6.29**

```
public class Department implements Serializable {
    private static final AtomicInteger GENERATED_ID = new AtomicInteger();

    @QuerySqlField(index = true)
    private Integer deptno;

    @QuerySqlField
    private String dname;

    @QuerySqlField
    private String loc;
    //rest of the part is omitted
}
```

---

Instances of these two classes will be saved in the Ignite caches.

**Step 3.** Next, I choose a server node configuration to use for this application. Create an example-ignite.xml file inside the src/main/resources directory, and place the Ignite server node configuration from Listing 6.30 into it.

**Listing 6.30**

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="clientMode" value="false"/>
        <property name="discoverySpi">
            <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
                <property name="ipFinder">
                    <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDiscoveryVmIpFinder">
                        <property name="addresses">
                            <list>
```

```
        <value>127.0.0.1:47500</value>
    </list>
</property>
</bean>
</property>
</bean>
</property>
</bean>
</beans>
```

---

Nothing is special in the above configuration file. We have configured Ignite node as a server node and define the TCP discovery address.

**Step 4.** Now, create a *SqlQueryEmployees* Java class inside the `com.blu.imdg` package. This class bootstraps an Ignite server node, populates the caches, and executes a few *SqlFieldsQuery*. Note that, this Java class contains over 300 lines of code. Therefore, we will discuss only the main parts of the class. You can find the entire project on [GitHub repository<sup>99</sup>](#).

**Listing 6.31**

```
public class SqlQueryEmployees {
    .....
    public static void main(String[] args) throws Exception {
        .....
        try{
            IgniteCache<Integer, Department> deptCache = ignite.createCache(deptCacheC\
fg); IgniteCache<EmployeeKey, Employee> employeeCache = ignite.createCache(employeeCac\
heCfg)
        }{
            initialize();
            // Example for SQL-based fields queries that return only required
            // fields instead of whole key-value pairs.
            sqlFieldsQuery();
            // Example for SQL-based fields queries that uses joins.
            sqlFieldsQueryWithJoin();
            // Example for query that uses aggregation.
            aggregateQuery();
        }
    }
}
```

<sup>99</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/sqlqueries>

```
// Example for query that uses grouping.
groupByQuery();

}

}

}

....
```

---

The `main()` method is the entry point of the `SqlQueryEmployees` class. This class contains a few more static methods besides the `main()` method. Method `initialize()` populates the caches with test data, and rest of the methods executes some SQL-based fields queries.

Whenever the `try()` block runs, the Ignite SQL engine does some magic for you. It finds the `QuerySqlField` annotations in the `Employee` and `Department` POJO, and creates two caches. Moreover, it creates two tables into the H2 database to execute SQL queries over the cache entries. The schema for both tables will be `SqlQueryEmployees-CACHENAME`. You can query the tables through `SQLLINE` or any SQL tools as shown in Listing 6.32.

**Listing 6.32**

---

```
SELECT * FROM "SqlQueryEmployees-employees".EMPLOYEE;
```

---

Figure 6.5 shows the using of `SQLLINE` tool to query the `Employee` table.

0: jdbc:ignite:thin://127.0.0.1/> !tables				
TABLE_CAT	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE	
	SqlQueryEmployees-employees	EMPLOYEE	TABLE	
	SqlQueryEmployees-departments	DEPARTMENT	TABLE	
0: jdbc:ignite:thin://127.0.0.1/> SELECT * FROM "SqlQueryEmployees-employees".EMPLOYEE;				
EMPNO	ENAME	JOB	MGR	
14	Miller	Clerk	3	
2	Blake	Manager	1	
9	Ward	Salesman	2	
4	Jones	Manager	1	
11	Turner	Salesman	2	
6	Ford	Analyst	4	
13	Jones	Clerk	2	
1	King	President	null	
8	Allen	Salesman	2	
3	Clark	Manager	1	
10	Martin	Salesman	2	
5	Scott	Analyst	4	
12	Adams	Clerk	5	
7	Smith	Clerk	6	

**Figure 6.5**

Let's have a look at an example of the `SqlFieldsQuery` API. For instance, we take the method `sqlFieldsQueryWithJoin()`.

**Listing 6.33**

---

```
private static void sqlFieldsQueryWithJoin() {
    IgniteCache<< ?, ? > cache = Ignition.ignite().cache(EMPLOYEE_CACHE_NAME);

    // Create query to get names of all employees.
    SqlFieldsQuery qry = new SqlFieldsQuery(
        "select e.ename, d.dname " +
        "from Employee e, \"" + DEPARTMENT_CACHE_NAME + "\".Department d " +
        "where e.deptno = d.deptno");

    Collection< List << ? >> res = cache.query(qry).getAll();

    // Print persons' names and departments' names.
    logDecorated("==Names of all employees and departments they belong to (SQL join)==", \
    res);
}
```

---

First, we get the cache by its name. Next, we create an instance of the *SqlFieldsQuery* class with a SQL statement. Then executes the query by the `cache.query()` method and prints the employee's name and departments into the console.



## Warning

Apache Ignite does not support foreign key constraints between tables. You should do integrity check at the application level.

Similarly, you can also execute SQL queries with aggregate functions.

**Listing 6.34**

---

```
SqlFieldsQuery qry = new SqlFieldsQuery("select sum(sal), count(sal) from Employee");
```

---

The above query returns the *sum* of salaries and the number of *summed rows*. That's all there is to the application, and it can now build and tested. To begin, compile the project with Maven command as shown in Listing 6.35.

**Listing 6.35**

---

```
mvn clean install
```

---

At this point an executable jar file `sql-query-employees-runnable.jar` will be created. Run the application from the project *home directory* as follows:

**Listing 6.36**

---

```
java -jar ./target/sql-query-employees-runnable.jar
```

---

You should see something like this (with other stuff like queries as well):

```
21:44:03.607 [main] INFO c.b.i.SqlQueryEmployees -      ==Names of all employees and departments they belong to (SQL join)==  
21:44:03.607 [main] INFO c.b.i.SqlQueryEmployees -      [Miller, Accounting]  
21:44:03.607 [main] INFO c.b.i.SqlQueryEmployees -      [Blake, Sales]  
21:44:03.607 [main] INFO c.b.i.SqlQueryEmployees -      [Ward, Sales]  
21:44:03.607 [main] INFO c.b.i.SqlQueryEmployees -      [Jones, Research]  
21:44:03.607 [main] INFO c.b.i.SqlQueryEmployees -      [Turner, Sales]  
21:44:03.607 [main] INFO c.b.i.SqlQueryEmployees -      [Ford, Research]
```

You can also query the tables through the *SqlFieldsQuery* API. In this particular case, you have to explicitly specify the *SCHEMA* name by the `SqlFieldsQuery.setSchema("SCHEMA NAME")` method. Moreover, with *SqlFieldsQuery* API you can execute the DML SQL statements to modify the data. *SqlFieldsQuery* API supports a few more special method to take control over the query execution unlike other SQL API's. A few of them is as follows:

- `setLazy(boolean lazy)`. Ignite attempts to fetch the whole query result set to memory and send it to the client application by default. For small and medium result sets this provides optimal performance and minimize the duration of internal database locks. However, if the result set is too big to fit in available memory, this could lead to excessive GC pauses and even *OutOfMemoryError*. Use this flag as a hint for Ignite to fetch result set `lazily`<sup>100</sup>, thus minimizing memory consumption at the cost of a moderate performance hit. The default value is `false`, meaning that the whole result set is fetched to memory eagerly.
- `setTimeout(int timeout, TimeUnit timeUnit)`. Sets the query execution timeout. A query will be automatically canceled if the execution timeout is exceeded.

---

<sup>100</sup>[https://en.wikipedia.org/wiki/Lazy\\_loading](https://en.wikipedia.org/wiki/Lazy_loading)

- `setPageSize(int pageSize)`. Sets optional page size, if 0, the default is used. Query cursor results are paginated automatically while you iterate over the cursor. For example, if the page size is **1024** (default), you will never have more than 1024 entries in local memory. After you finish iteration through the first page, the second one will be requested, and so on. This allows avoiding out of memory issues in case of large result sets.
- `setCollocated(boolean collocated)`. Sets flag defining if this query is collocated. Collocation flag is used for optimization purposes of queries with GROUP BY statements. Ignite sends sub-queries to individual cluster members whenever it executes a distributed query. If you know in advance that the elements of your query selection are collocated together on the same node and you group by collocated key (primary or affinity key), then Ignite can make significant performance and network optimizations by grouping data on remote nodes. We explain the distributed joins a little bit later in this chapter.

The final piece of the API puzzle is the *SqlQuery* API. In the next section, we will go through the *SqlQuery* with examples.

## SqlQuery

*SqlQuery* API returns the entire object back in the result set. The API is very much similar to the Hibernate HQL. Ignite *SqlQuery* is translated by Ignite SQL engine into a conventional SQL query, which in turns performs an action on the database. You can imagine *SqlQuery* is an alternative to SQL query like `SELECT * from Employee`. Ignite *SqlQuery* is extremely simple to use, and the code is always self-explanatory. Let's have a look at a `SELECT` operation using *SqlQuery* API (the pseudo code is taken from the [GitHub repository<sup>101</sup>](#)).

**Listing 6.37**

---

```
private static void sqlQueryWithJoin() {
    IgniteCache<EmployeeKey, Employee> cache = Ignition.ignite().cache(EMPLOYEE_CACHE_NAME);

    SqlQuery<EmployeeKey, Employee> qry = new SqlQuery<>(Employee.class,
        "from Employee, " + DEPARTMENT_CACHE_NAME + ".Department " +
        "where Employee.deptno = Department.deptno " +
        "and lower(Department.dname) = lower(?)");
}

// Execute queries for find employees for different departments.
```

---

<sup>101</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/sqlqueries>

```
    logDecorated("==Following department 'Accounting' have employees (SQL join)==", cache\  
.query(qry.setArgs("Accounting")).getAll());  
    logDecorated("==Following department 'Sales' have employees (SQL join)==", cache.query\  
y(qry.setArgs("Sales")).getAll());  
}
```

---

The above SELECT query retrieves employee's details for departments `Sales` and `Accounting`. Here, the `EmployeeKey` is an affinity key which is used to collocated the employee with the same departments on the same node.



## Tip

`SqlQuery` only supports queries of the type `select * from ...` or `select alias.* from ...`, and does not support queries with a subset of columns. Because `SqlQuery` always returns all fields, you can omit the `SELECT` clause of the query, e.g. use `FROM Persons WHERE ...`, or provide only the `WHERE` clause, e.g. `salary > ?`.

Ignite `SqlQuery` also supports the `setTimeout()` method for canceling a query execution. Also, `SqlQuery` provides `QueryCursor.close()` method to close the entire cursor which in result cancel the query. You can find a few more examples to demonstrate the usage of `SqlQuery` API in our `SqlQueryEmployees` class located in the project code base.

In the next section, we are going to cover how SQL query works in Ignite which give you a clear understand what's going under the cover whenever you runs a query against the Ignite database.

## How SQL queries works in Ignite

In this chapter so far, we have discussed a lot about Ignite SQL capabilities. Perhaps it is the exact time to dive deep into the Ignite distributed SQL engine to explorer how it runs SQL queries under the hood. Most often Ignite executes SQL queries in two or more steps:

- Ignite distributed SQL engine parse and analyze the DDL/DML statements, creates appropriate caches from a non-ANSI part of the SQL queries and delegates the ANSI part to the H2 database engine.
- H2 executes a query locally on each node and passes a local result to the distributed Ignite SQL engine for aggregating the result or further processing.

Consequently, Apache Ignite SQL engine is firmly coupled with H2<sup>102</sup> database and uses particular version of the database (for instance, Ignite version 2.6 uses H2 1.4.195 version). Each Ignite node runs one instance of the H2 database in the embedded mode. In this mode, the H2 database instance runs within the same Ignite process as a in-memory database.



## Info

H2 SQL engine always executes SQL queries on local Ignite node.

We have already introduced you to the H2 database in *chapter 2*. However, in this section, we are going to answer the following questions:

1. How tables and indexes organize in H2 database?
2. How H2 database uses indexes?
3. How to use SQL statements execution plans for optimizing queries?
4. How SQL queries work for the PARTITIONED and REPLICATED caches?
5. How concurrent modification works?

For simplicity, we use our well-known dataset: *Employee* and *Department* tables. We also use the Ignite *SQLLINE* command line tool and H2 Web console to run SQL queries.

H2 database supplies a web console server to access the local database using a web browser. As we described earlier, the H2 console server allows you to run SQL queries against the embedded node and check how the tables and indexes look like internally. Moreover, with the H2 web console, you can analyze how a query is executed by the database, e.g., whether indexes are used or if the database has done an expensive full scan. This feature is crucial for optimizing the query performance. To do that, you have to start the Ignite local node with `IGNITE_H2_DEBUG_CONSOLE` system property or an environment variable set to *true* as shown below:

**Listing 6.38**

---

```
export IGNITE_H2_DEBUG_CONSOLE=true
```

---



## Info

Ignite *SQLLINE* also supports *SQL execution plan* (*EXPLAIN*), but through H2 web console you can execute *EXPLAIN* plan in a *particular* node which is very useful to investigating query performance for specific Ignite node.

---

<sup>102</sup><http://www.h2database.com/html/main.html>

Now, start the Ignite node from any terminal.

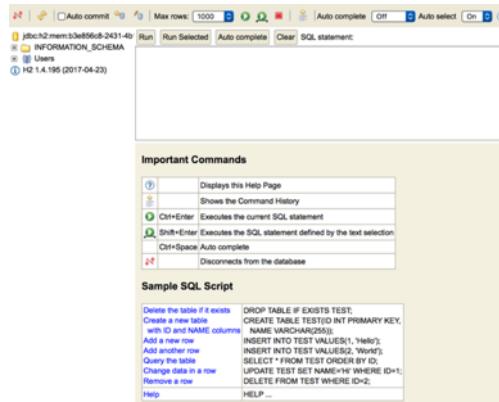
**Listing 6.39**

---

```
$IGNITE_HOME/bin/ignite.sh
```

---

A web console will be opened in your browser as shown in the next figure. As mentioned earlier, whenever we create any table (through SQLLINE or Ignite SQL JAVA API) in Ignite, a meta-data information of the tables created and displayed in the H2 database. However, the data, as well as the indexes, are always stored in the Ignite caches that execute queries through the H2 database engine.



**Figure 6.6**

Let's create the *EMP* table through the *SQLLINE* tool and observe what happen on the H2 database. Run the following script for running the *SQLLINE* command console to connect to the Ignite cluster:

**Listing 6.40**

---

```
$ ./sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
```

---

Next, create the *EMP* table with the following DDL script.

**Listing 6.41**

---

```
CREATE TABLE IF NOT EXISTS EMP
(
    empno LONG,
    ename VARCHAR,
    job VARCHAR,
    mgr INTEGER,
    hiredate DATE,
    sal LONG,
    comm LONG,
    deptno LONG,
    CONSTRAINT pk_emp PRIMARY KEY (empno)
) WITH "template=replicated,backups=1,CACHE_NAME=EMPcache";
```

---

Now, we have a table named *EMP* in Ignite as shown in the following figure.

0: jdbc:ignite:thin://127.0.0.1/	!tables	
TABLE_CAT	TABLE_SCHEMA	TABLE_NAME
	PUBLIC	EMP

Figure 6.7

Let's get back to the H2 web console and refresh the H2 database objects panel (you should see a small refresh button on the H2 web console menu bar on the upper left side of the web page). You would probably see a new table with name *EMP* appears on the object panel. Expand the *EMP* table, and you should get the following picture as shown in figure 6.8.

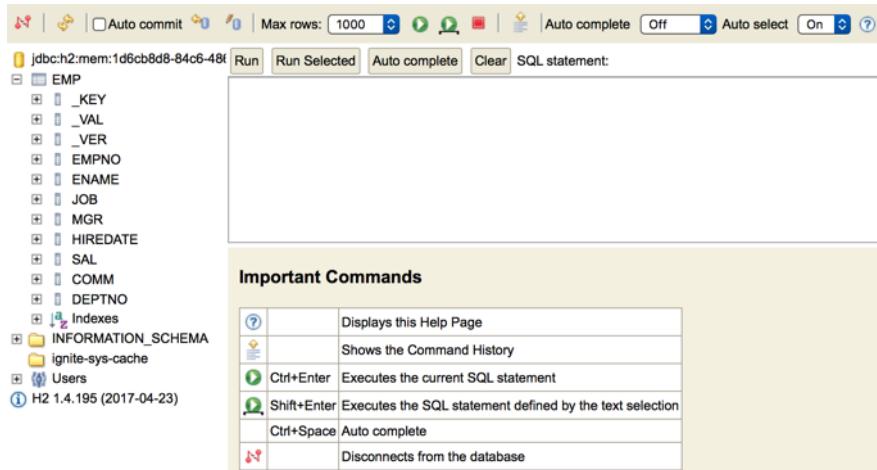


Figure 6.8

From the above screenshot, you can discover that H2 EMP table contains three extra columns with name `_KEY`, `_VAL`, `_VER`. Apache Ignite as a key-value data store, always stores cache keys and values as `_KEY` and `_VAL` fields. H2 EMP table column `_key` and `_val` corresponded to the Ignite internal `_key` and `_val` field of the `EMPCache` cache. The `_VER` field assigns the Ignite topology version and the node order. Let's have a look at the DDL scripts that H2 executed for creating the `EMP` table. Execute the following SQL query on the *SQL statement panel* of the H2 web console as shown below:

Listing 6.42

---

```
select table_catalog, table_name, table_type, storage_type, SQL from information_schema.tables
where table_name = 'EMP';
```

---

You should have the following query result on the panel.

select table_name, table_type, storage_type, SQL, table_class from information_schema.tables where table_name = 'EMP';				
TABLE_NAME	TABLE_TYPE	STORAGE_TYPE	SQL	TABLE_CLASS
EMP	EXTERNAL	MEMORY	<pre>CREATE MEMORY TABLE PUBLIC.EMP(     _KEY BIGINT INVISIBLE NOT NULL,     _VAL OTHER INVISIBLE ,     _VER OTHER INVISIBLE ,     EMPNO BIGINT,     ENAME VARCHAR,     JOB VARCHAR,     MGR INT,     HIREDATE DATE,     SAL BIGINT,     COMM BIGINT,     DEPTNO BIGINT ) ENGINE "org.apache.ignite.internal.processors.query.h2.H2TableEngine"</pre>	org.apache.ignite.internal.processors.query.h2.opt.GridH2Table

Figure 6.9

We used the H2 system table `TABLES` to get all the meta-data infotmation of the `EMP` table.



## Tip

The system tables of the H2 database in the schema `INFORMATION_SCHEMA` contain the metadata of all tables in the database as well as the current settings. Examples of the system tables are `TABLES`, `INDEXES`, `VIEWS`, etc.

The exciting part of the query result is `STORAGE_TYPE`, `TABLE_CLASS` and the `SQL` itself.

- **SQL:** See the listing 6.43. The `CREATE` table DDL statement indicates that the `EMP` table is a *MEMORY* table. A memory table is a persistence table, but the index of data is kept in the memory, so a memory table cannot be too large. The `EMP` table also uses custom table implementation which is specified by the `ENGINE` parameter. H2 uses the H2 table engine implementation of the class `org.apache.ignite.internal.processors.query.h2.H2TableEngine` to create the `EMP` table. H2 table engine creates the table using given connection, DDL clause for given type descriptor and list of indexes. Created `EMP` table has three individual columns with invisible column definition which made the columns hidden, i.e., it will not appear in `SELECT *` results.
- **TABLE\_CLASS:** `org.apache.ignite.internal.processors.query.h2.opt.GridH2Table`. The `EMP` table is not an H2 regular table. It's a user-defined or custom designed table which is the type of Ignite `GridH2Table`. The Ignite `H2TableEngine` uses the implementation of the `GridH2Table` for creating tables in the H2 database.
- **STORAGE\_TYPE:** `MEMORY`. The storage type of the table is `MEMORY`.

**Listing 6.43**


---

```
CREATE MEMORY TABLE PUBLIC.EMP(
    _KEY BIGINT INVISIBLE NOT NULL,
    _VAL OTHER INVISIBLE ,
    _VER OTHER INVISIBLE ,
    EMPNO BIGINT,
    ENAME VARCHAR,
    JOB VARCHAR,
    MGR INT,
    HIREDATE DATE,
    SAL BIGINT,
    COMM BIGINT,
    DEPTNO BIGINT
)
ENGINE "org.apache.ignite.internal.processors.query.h2.H2TableEngine"
```

---

Next, execute the following query into the H2 web console *SQL statement pane*.

**Listing 6.44**


---

```
select table_name, non_unique, index_name, column_name, primary_key, SQL, index_class fro
m information_schema.indexes where table_name = 'EMP';
```

---

We use the H2 system table *INDEXES* to discover the meta-data information about the indexes used by the *EMP* table. After executing the above SQL, you should have the following output demonstrated in the next table.

TABLE NAME	NON UNIQUE	INDEX NAME	COLUMN NAME	PRIMARY KEY	SQL	INDEX CLASS
EMP	FALSE	_key_PK_hash	_KEY	TRUE	CREATE PRIMARY KEY HASH PUBLIC."_key_- PK_hash" ON PUBLIC.EMP(_-	H2PkHashIndex
EMP	FALSE	_key_PK	_KEY	TRUE	KEY) CREATE PRIMARY KEY PUBLIC."_key_- PK" ON PUBLIC.EMP(_- KEY)	H2TreeIndex

TABLE NAME	NON UNIQUE	INDEX NAME	COLUMN NAME	PRIMARY KEY	SQL	INDEX CLASS
EMP	TRUE	_key_PK_proxy	EMPNO	FALSE	CREATE INDEX PUBLIC."_key_-PK_proxy" ON PUBLIC.EMP(EMPNO)	GridH2ProxyIndex

We have three different indexes generated for the table EMP by the H2 engine: *Hash*, *Tree* and *Proxy* index. Two different primary key indexes \_key\_PK\_hash and \_key\_PK are created on column \_KEY. The main interesting point is that, column EMPNO is not a primary key, rather than a proxy index \_key\_PK\_proxy is created on column EMPNO. A proxy index allows to delegates the calls to the underlying normal index. \_key\_PK is a B+ tree primary index created on column \_KEY. The \_key\_PK\_hash is an in-memory hash index and usually faster than the regular index. Note that, Hash index does not support range queries similar to a hash table.

Now that we have got an idea about the internal structure of the tables and indexes in H2 database. Let's insert a few rows into the EMP table and run some queries to know how data is organized into a table. First, insert some data into the EMP table as follows.

**Listing 6.45**

---

```
insert into emp (empno, ename, job, mgr, hiredate, sal, comm, deptno) values(7839, 'KING'\n, 'PRESIDENT', null, to_date('17-11-1981','dd-mm-yyyy'), 5000, null, 10);\ninsert into emp (empno, ename, job, mgr, hiredate, sal, comm, deptno) values( 7698, 'BLAK\\E', 'MANAGER', 7839, to_date('1-5-1981','dd-mm-yyyy'), 2850, null, 30);\ninsert into emp (empno, ename, job, mgr, hiredate, sal, comm, deptno) values(7782, 'CLARK\\', 'MANAGER', 7839, to_date('9-6-1981','dd-mm-yyyy'), 2450, null, 10);
```

---

Enter the following simple *SELECT* query into the H2 web console SQL statement panel.

**Listing 6.46**

---

```
select _key, _val, _ver from EMP;
```

---

The above query should return you the following output.

KEY	VAL	VER
7698	SQL_PUBLIC_EMP_da74a220_118a_4111_a7d7_-283d55dec3fb [idHash=1526052858, hash=-1417502761, ENAME=BLAKE, JOB=MANAGER, MGR=7839, HIREDATE=1981-05-01, SAL=2850, COMM=null, DEPTNO=30]	GridCacheVersion [topVer=153331122, order=1541851119645, nodeOrder=1]

KEY	VAL	VER
7782	SQL_PUBLIC_EMP_da74a220_118a_4111_a7d7_-283d55dec3fb [idHash=1412584148, hash=1453774085, ENAME=CLARK, JOB=MANAGER, MGR=7839, HIREDATE=1981-06-09, SAL=2450, COMM=null, DEPTNO=10]	GridCacheVersion [topVer=153331122, order=1541851119647, nodeOrder=1]
7839	SQL_PUBLIC_EMP_da74a220_118a_4111_a7d7_-283d55dec3fb [idHash=1343085706, hash=1848897643, ENAME=KING, JOB=PRESIDENT, MGR=null, HIREDATE=1981-11-17, SAL=5000, COMM=null, DEPTNO=10]	GridCacheVersion [topVer=153331122, order=1541851119643, nodeOrder=1]

From the preceding figure, you can notice that every `_KEY` field hold the actual employee number (`EMPNO`), and the rest of the information such as job, salary information hold by the `_VAL` field. H2 EMP table `EMPNO` field maps to the actual `_key` field of the cache, job field maps to the `_VAL` field job attribute and so on. Whenever you execute any SQL query, H2 engine uses the regular fields such as `ENAME`, `MGR`, `JOB` fields to fulfill the query request.

Before moving on to the query execution plan, let's clarify how Ignite process SQL queries under the cover. Ignite always uses *Map/Reduce* pattern to process SQL queries. Whenever Ignite SQL engine receives queries, it split the query into two parts: *Map* and *Reduce*. Map query runs on each participating cache node depends on the cache mode (PARTITIONED OR REPLICATED), and Reduce query aggregates results received from all nodes running map query. We can generalize the process of executing SQL queries based on the Cache/Table modes into two categories:

- REPLICATED/LOCAL CACHE/TABLE. If a SQL query is executed against Replicated or Local cache data, Ignite distributed SQL engine knows that all data is available locally and runs a local SQL map query in the H2 database engine, aggregates the result on the same node and returns the query result to the application. Map and Reduce parts executes on the same Ignite node, so the query performance is more impressive. It's worth mentioning that in replicated caches, every node contains a replica data for other nodes.

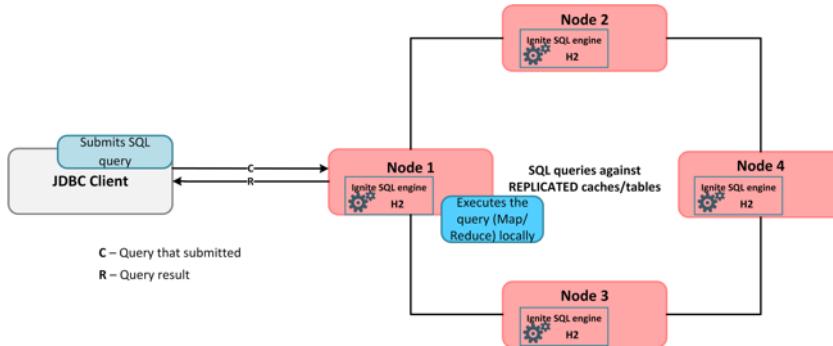


Figure 6.10

- PARTITIONED CACHE/TABLE. SQL queries against partitioned caches/tables is slightly complicated. If you are executing any SQL query against a Partitioned cache, Ignite under the hood splits the query into multiple in-memory map queries and a single reduce query. The number of map queries depends on the size of the partitions and number the partitions (or nodes) in the cluster. Then, all map queries are executed on all data nodes of the participating caches, relaying results to the reducing node, which will, in turn, run the reduce query over these intermediate results. If you are not familiar with the Map-Reduce pattern, you can imagine it as a Java Fork-join process.

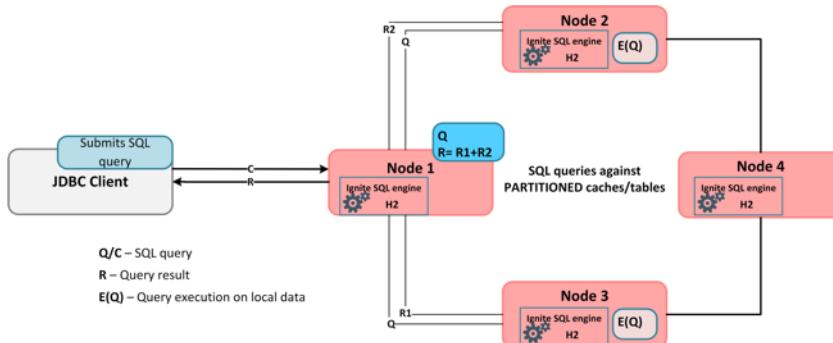


Figure 6.11

In addition, to study the internal structure of the tables and indexes in H2 databases, you can also see query execution plan against local data in H2 web console. Ignite SQLLINE tool also supports the *EXPLAIN* syntax for reading execution plans for the distributed query. In the last part of this sub-section, we use both of them to see how to read query execution plans and how to get benefit from it.

In the last part of this section, we use both H2 and SQLLINE SQL execution plans to see how to read query execution plans and how to get benefit from it.

Let's start from the definition and purpose of the Query Execution plan. A query plan is a set of steps that the DBMS executes to complete the query. The reason we have query plans is that the SQL you write may declare your intentions, but it does not tell the SQL engine the exact logic to follow. When a query is submitted to the database, the query optimizer<sup>103</sup> evaluates some of the different, correct possible plans for executing the query and returns what it considers the best option. In H2 database the process is called statement execution plan and provides two different syntaxes to run query execution plans: *EXPLAIN* and *EXPLAIN ANALYZE*.

H2 *EXPLAIN* statement displays the indexes and optimizations the database uses for the given SQL statement. This statement actually does not execute the query, rather than prepare it. Let's run the following query into the SQLLINE tool and then repeat the process into the H2 web console.

**Listing 6.47**

---

```
Explain select ename, job from emp where ename='KING';
```

---

After running the above query into the SQLLINE tool, you should have the following result on the console.

```
+-----+  
| PLAN |  
+-----+  
| SELECT  
| _Z0.ENAME AS _C0_0,  
| _Z0.JOB AS _C0_1  
FROM PUBLIC.EMP _Z0  
/* PUBLIC.EMP._SCAN */  
| SELECT  
| _C0_0 AS ENAME,  
| _C0_1 AS JOB  
FROM PUBLIC._T0  
/* PUBLIC."merge_scan" */ |  
+-----+
```

As we mentioned before, Ignite SQL engine receives the SQL queries and split this query into two parts: *Map* and *Reduce*. Map query runs on the single Ignite node because our *EMP*

---

<sup>103</sup>[https://en.wikipedia.org/wiki/Query\\_plan](https://en.wikipedia.org/wiki/Query_plan)

table is replicated. Reduce query aggregate the result received from the H2 database in the same node and return the result. So, in query execution plan we have two parts: Map parts with the Z0 prefix, and a reduce part with C0 prefix. If you run the above query through H2 web console, you should have only one part as shown below. This is because, H2 executes the entire queries as a whole and does not splits into parts.

```
SELECT
    ENAME,
    JOB
FROM PUBLIC.EMP
/* PUBLIC.EMP._SCAN_ */
WHERE ENAME = 'KING'
```

Let's get back to our explain plan result from the SQLLINE tool. Comment /\*PUBLIC.EMP.\_SCAN \*/ from the Map part of the query result indicates that the query does not use any indexes and executes a full table scan. These are the expected results because we did not create an index on column ENAME. Let's create a user-defined index on column *ENAME* through SQLLINE tool and re-run the explain plan query.

#### **Listing 6.48**

---

```
CREATE INDEX ename_idx ON emp (ename);
```

---

Now the SQL query to find the employee with name KING will use the index, and the explain plan query should return the following output.

```
+-----+
|   PLAN    |
+-----+
| SELECT
|   _Z0.ENAME AS _C0_0,
|   _Z0.JOB AS _C0_1
FROM PUBLIC.EMP _Z0
/* PUBLIC.ENAME_IDX: */
| SELECT
|   _C0_0 AS ENAME,
|   _C0_1 AS JOB
FROM PUBLIC._T0
/* PUBLIC."merge_scan" */
+-----+
```

H2 SQL engine uses only one index per table. If you create an index on column *SAL* and run the query explain plan using the condition `where ename='KING' and job='PRESIDENT' or sal>3000`. The query process would use a full table scan instead of first using the index on column ENAME and then the index on SAL.

```
+-----+  
|      PLAN      |  
+-----+  
| SELECT  
|   _Z0.JOB AS __C0_0,  
|   _Z0.SAL AS __C0_1  
| FROM PUBLIC.EMP _Z0  
| /* PUBLIC.EMP._SCAN_ */|  
| SELECT  
|   __C0_0 AS JOB,  
|   __C0_1 AS SAL  
| FROM PUBLIC.__T0  
| /* PUBLIC."merge_scan" */|  
+-----+
```

In such cases, it makes sense to write two queries and combines them using *UNION*. Thus each individual query uses a different index as follows:

**Listing 6.49**

---

```
Explain Select job, sal from emp where ename='KING' and job='PRESIDENT'  
UNION  
select job, sal from emp where sal>3000;
```

---

The preceding SQL statement should use two different indexes and give us the following query plan as shown in the following screenshot.

```
+-----+  
| PLAN |  
+-----+  
| SELECT  
|   _Z0.JOB AS _C0_0,  
|   _Z0.SAL AS _C0_1  
| FROM PUBLIC.EMP _Z0  
|   /* PUBLIC.ENAME_IDX: E */ |  
| SELECT  
|   _Z1.JOB AS _C1_0,  
|   _Z1.SAL AS _C1_1  
| FROM PUBLIC.EMP _Z1  
|   /* PUBLIC.IDX_SALARY: */ |  
| (SELECT  
|   _C0_0 AS JOB,  
|   _C0_1 AS SAL  
| FROM PUBLIC._T0  
|   /* PUBLIC."merge_scan" */)
```

Besides to EXPLAIN keywords, you can also use ANALYZE keyword which shows the scanned rows per table. The *EXPLAIN ANALYZE* keyword actually executes the SQL queries unlike *EXPLAIN* which only process it. This execution plan is useful to know how many rows will be affected by the queries. Unfortunately, Ignite SQLLINE tool does not support the *EXPLAIN ANALYZE* keyword. However, you can use this keyword through H2 web console. The following query scanned two rows while executing the plan.

```
SELECT  
  ENAME,  
  JOB  
FROM PUBLIC.EMP  
/* PUBLIC.ENAME_IDX: ENAME = 'KING' */  
/* scanCount: 2 */  
WHERE ENAME = 'KING'
```



## Warning

Not all execution plan is this simple and sometimes they might be challenging to read and understand.

**HOW SQL join works?**

The internal of any query operation can be reviewed in the query execution plan. There are several factors that involves how the data is extracted and queried from the source tables. Mainly there are two elements you need to be concerned about:

- *Seeks and Scans.* The first step to any query is to extract the data from the source tables. This is performed using one of two actions (and variations within): a *seek* or a *scan*. A seek is where the query engine can leverage an index to narrow down the data retrieval. If no index is available, then the engine must perform a scan of all records in the table. In general, seeks typically perform better than scans.
- *Physical Joins.* Once data is retrieved from the tables, it must be physically joined. Note, this is not the same as logical joins (INNER and OUTER). There are three physical *join operators*<sup>104</sup> that can be used by the engine: *Hash Join*, *Merge Join* and *Nested loop join*.

The query engine will evaluate several factors to determine which physical join will be the most efficient for the query execution. H2 uses *hash joins* during the query execution.

Another vital part of the query execution is the query optimizer. H2 SQL engine is a cost-based optimizer. Cost-based optimization involves generating multiple execution plans and selecting the lowest cost execution plans to fire and finally execute a query. The CPU, IO, and memory are some of the parameters H2 uses in recognizing the cost, and finally generating the execution plan.

Now you have a better understanding of how Ignite runs SQL queries under the cover. If you can read and understand query execution plans, it helps you to work on query tuning and optimization. In other words, you will understand which objects are in use within the database you are working.

## Cache queries

Apache Ignite provides a powerful Query API which supports *Predicate-base Scan Queries*, *Text search queries* and *Continuous queries*. Suppose one of your PARTITIONED cache contains *Company* objects as follows:

---

<sup>104</sup><https://www.periscopedata.com/blog/how-joins-work>

**Listing 6.50**

---

```
public class Company {  
    private Long id;  
    private String companyName;  
    private String email;  
    private String address;  
    private String city;  
    private String state;  
    private String zipCode;  
    private String phoneNumber;  
    private String faxNumber;  
    private String webAddress;  
}
```

---

Cache with the entries `company` will be partitioned and spread across the Ignite cluster. Definitely, you can iterate over the Cache entries and look for specific entries that you are interested in as we have done in *chapter two*. However, iterating over the whole cache is not very efficient because you will have to fetch the entire dataset and iterate through it locally. If the dataset is large, it will increase the network traffic. To solve such kind of issues, Apache Ignite provides a programming interface, which allows you to execute different types of cache queries against the cache entries. You can choose relevant API by your application requirements. For instance, if you want to execute a *full-text search* of the companies which address is similar to BLEECKER ST APT 51, Ignite *Text queries* is the best choice for you.

Apache Ignite delivers two main abstractions of Query API: *Query* class and *QueryCursor* interface.

Name	Description
Query	Base class for all Ignite cache queries. You have to use <code>SqlQuery</code> and <code>TextQuery</code> for SQL and text queries accordingly. This abstract class also provides methods such as <code>setLocal()</code> and <code>setPageSize()</code> to executes queries in local node and set the page size for the returned cursor.
QueryCursor	Interface, which represents the query result with page-by-page iteration. When pagination is not needed, you can use <code>QueryCursor.getAll()</code> method which will fetch the whole query result and store it in a collection. Note that, you must close the cursor explicitly, whenever iterating over the cursor in a for loop.



## Warning

`QueryCursor` interface is not thread-safe, you should use it from a single thread only.

Figure 6.12 illustrated the Ignite Query class and all extended classes derived from it.

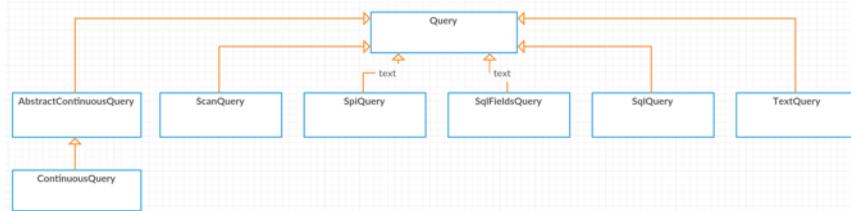


Figure 6.12

## Scan queries

Ignite Scan queries allow you to run distributed queries over cache entries. If no predicate is specified, queries return all the entries of the cache. You can define any predicate based on objects stored in the cache.



### Tip

A *predicate* is a specialized expression, which always returns a Boolean type.

The query will apply the predicates to all of the entries to find matches. For demonstrating the power of the scan queries, we are going to use the following dataset.

ID	CAT	COMPANY_NAME	EMAIL	ADDRESS	CITY	STATE	ZIPCODE	PHONE_NUMBER	FAX_NUMBER
230001	M	Medical-ID.com	bcourt@aol.com	115 Cooper Dr	VAILS GATE	NY	12584	null	null
230002	W	WEINBERG; KEITH	keith@groupdigital.com	272 BLEECKER ST APT 51	NEW YORK	NY	10013	null	null
230003	M	Maccio Physical Therapy	Maccio@macciophysicaltherapy.com	1 New Hampshire Ave	Troy	NY	12180-1754	5182730701	5182732121
230004	P	Prudential Douglas Elliman Re	thomas.uhlinger@prudentialelliman.com	10200 Main Rd / Po Box 1410	Mattituck	NY	11952	63129880000	-
230005	F	Fragomen Del Rey Bernsen & Low	sanfranciscoinfo@fragomen.com	515 Madison Ave # 15	New York	NY	10022-5493	9494400119	2126888555

Figure 6.13

The dataset contains all the companies contact information located in the state of New York. You can download the full source code from the [GitHub repository](https://github.com/srecon/the-apache-ignite-book/blob/master/chapters/chapter-6/textquery/src/main/resources/USA_NY_email_addresses.csv)<sup>105</sup>. We are going to load

<sup>105</sup>[https://github.com/srecon/the-apache-ignite-book/blob/master/chapters/chapter-6/textquery/src/main/resources/USA\\_NY\\_email\\_addresses.csv](https://github.com/srecon/the-apache-ignite-book/blob/master/chapters/chapter-6/textquery/src/main/resources/USA_NY_email_addresses.csv)

the entries from the CSV file into the Ignite cache, and apply some scan queries over the cache entries:

where,

- ID: Serial Number
- CAT: CAT (Company Name starting letter)
- COMPANY\_NAME: Name of the company
- EMAIL: email used by company or individuals
- ADDRESS: Street address or area
- CITY: City name
- STATE: State Name
- ZIPCODE: Zipcode details
- PHONE\_NUMBER: Phone number used by company or individuals
- FAX\_NUMBER: Fax number used by company or individuals

**Step 1.** Create a new Maven project with the following dependencies.

Listing 6.51

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>{ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>{ignite.version}</version>
</dependency>
```

---

**Step 2.** Add a new Java class with name `com.blu.imdg.model.Company` into the `src\main\java\com\blu\imdg\model` directory which represents the dataset.

**Listing 6.52**

---

```
public class Company implements Serializable {  
  
    private Long id;  
    private String cat;  
    private String companyName;  
    private String email;  
    private String address;  
    private String city;  
    private String state;  
    private String zipCode;  
    private String phoneNumber;  
    private String faxNumber;  
    // setter and getter are omitted  
}
```

---

Note that, the above Java class is a plain old Java (POJO) class without any annotations.

**Step 3.** Now, let's load the data from the CSV file into the Ignite cache. Create a new Java class named `com.blu.imdg.scanQueryExample` into the directory `/src/main/java/com/blu/imdg`. Add a new method named `initialized()` and use the following contents:

**Listing 6.53**

---

```
private static void initialize() throws InterruptedException, IOException {  
    IgniteCache<Long, Company> companyCache = Ignition.ignite().getOrCreateCache(COMPANY_CACHE_NAME);  
    // Clear caches before start.  
    companyCache.clear();  
    // Companies  
    try {  
        Stream<String> lines = Files.lines(Paths.get(TextQueryExample.class.getClassLoader().getResource("USA_NY_email_addresses.csv").toURI()));  
        lines  
            .skip(1)  
            .map(s1 -> s1.split("\\,"))  
            .map(s2 -> new Company(Long.valueOf(s2[0].replaceAll("\"", "")), s2[1], s2[2],  
                s2[3], s2[4], s2[5], s2[6], s2[7], s2[8], s2[9], s2[10], s2[11], s2[12].replaceAll("\\",  
                    ",")))  
            .forEach(r -> companyCache.put(r.getId(), r));  
    } catch (URISyntaxException | IOException e) {  
        log(e.getMessage());  
    }  
}
```

---

```
    }
    // Wait 1 second to be sure that all nodes processed put requests.
    Thread.sleep(1000);
}
```

---

Take some moments to read through the code to understand how this works. The code is very simple. First, we create an Ignite cache with name `companyCache`, which stores all the entries of the company. Then we use Java 8 Stream API to read and gets the content of the file `USA_NY_email_addresses.csv` as a stream. Then, we skip the first line of the CSV file, split every row by `,` and creates new `Company` objects to store them into the Ignite cache. The last line of the above code forces the application to wait for a second to be sure that all nodes of the cluster processed the `put` requests.

**Step 4.** Let's apply the Ignite *Scan queries* to the cache. Add a new Java method named `scanQuery()` in the class `ScanQueryExample` as follows:

**Listing 6.54**

```
private static void scanQuery() {
    IgniteCache<Long, Company> companyCache = Ignition.ignite().cache(COMPANY_CACHE_NAME);
    ME);

    // Query for all companies which the city 'NEW YORK' – State NewYork.
    QueryCursor<ScanQuery<Long, Company>> cursor = companyCache.query(new ScanQuery<Long, Company>((k, p) -> p
        .getCity().equalsIgnoreCase("NEW YORK")));

    for (Iterator<Object> ite = cursor.iterator(); ite.hasNext();) {
        CacheEntryImpl<Object> company = (CacheEntryImpl<Object>) ite.next();

        log(((Company) company.getValue()).getCompanyName());
    }
    cursor.close();
}
```

---

In the above pseudo code, first we get the Ignite cache for the `company` and create a new `ScanQuery` with the Java 8 lambda expression. We also pass the following predicates expression:

```
p.getCity().equalsIgnoreCase("NEW YORK")
```

This expression is looking for the company objects which city name equals to NEW YORK. Ignite applies the above predicates to all of the cache entries and return `QueryCursor` of that cache

entry, which city name equals to NEW YORK. Next, we iterate through the query cursor and print the *company name* on the console.



## Tip

Note that, after processing the cursor, we explicitly close the cursor by invoking the method `cursor.close()`.

**Step 5.** To build and execute the application, run:

**Listing 6.55**

---

```
mvn clean install && java -jar target/textquery-runnable.jar scanquery
```

---

Running this application yields output similar to this:

```
09:21:33.241 [main] INFO c.b.i.TextQueryExample - Yes Network
09:21:33.242 [main] INFO c.b.i.TextQueryExample - Launch Photography
09:21:33.242 [main] INFO c.b.i.TextQueryExample - ekstra design inc
09:21:33.242 [main] INFO c.b.i.TextQueryExample - New York Times
09:21:33.242 [main] INFO c.b.i.TextQueryExample - Lewis Brisbois
09:21:33.242 [main] INFO c.b.i.TextQueryExample - CafeCleo.com
09:21:33.242 [main] INFO c.b.i.TextQueryExample - Canfield media and arts; inc
09:21:33.242 [main] INFO c.b.i.TextQueryExample - DKO New York
09:21:33.242 [main] INFO c.b.i.TextQueryExample - Wedding Ring Originals
09:21:33.242 [main] INFO c.b.i.TextQueryExample - Slater
09:21:33.242 [main] INFO c.b.i.TextQueryExample - Text query example finished.
```

**Figure 6.14**

Figure 6.14 shows that Ignite scan query found all the company with the city name New York. It is important to note that, you can combine a few predicates expression in one statement. Take a look to the following pseudo code:

**Listing 6.56**

---

```
new ScanQuery<Long, Company>((k, p) -> p.getCity().equalsIgnoreCase("NEW YORK") && p.getEmail().equalsIgnoreCase("abc@yescompany.com"))
```

---

Here, we are looking for the companies, which city name belongs to the New Work and the individual's e-mail address equals abc@yescompany.com.

Sometimes, you need to work with only a few *properties* of the cache entries rather than the entire cache objects. This is useful, for instance, when you want to fetch only several fields out of a large object and want to minimize the network traffic. Scan query provides optional

transformer *closure*, which can transform the object and then returns the transformed object. For example, if we need only the *company Id* which is situated in the city *New York*, we can do this as follows:

#### Listing 6.57

```
List< Long > companyName = companyCache.query(new ScanQuery< Long, Company > (
    // Remote filter.
    new IgniteBiPredicate< Long, Company > () {
        @Override public boolean apply(Long k, Company p) {
            return p.getCity().equalsIgnoreCase("NEW YORK");
        }
    },
    // Transformer.
    new IgniteClosure< Cache.Entry< Long, Company > , Long > () {
        @Override public Long apply(Cache.Entry< Long, Company > e) {
            return e.getValue().getId();
        }
    }
).getAll();
```

The above pseudo code returns the List of the companies `id` which match the predicate expression `getCity==New York`. Next, we are going to explore another exciting feature of the Ignite cache queries: **full-text search**.

## Text queries

Text queries in Apache Ignite let you run *full-text* queries against character-based cache entries. Let's define why we need full-text search queries and how it differs from the Scan queries. For example, my spouse asked me to find a list of the `beauty salon` in the city New York. With scan queries, I can run the following queries:

```
QueryCursor cursor = companyCache.query(new ScanQuery< Long, Company >((k, p) -> p.getCompanyDescription().contains("beauty salon")));
```

The above query will return me the list of companies whose company description contains `beauty salon`. However, this approach has a few downsides:

1. To run the query, I have to know the properties (fields) of the company's object. In my case, it's the name or description of the company fields. However, there might be a situation when I don't have any idea about the object properties.

2. It will scan the *entire cache*, which is very inefficient if the cache contains a large range of datasets.
3. Scan query gets complicated whenever you have to use complex predicates based on the different fields or properties of the object. For example, if you have to find out the company which name is beauty saloon and situated in address abc street and index might be 4356.

## How text queries work in Ignite?

In daily life, we perform a lot of text search on the Web: Google or Bing. Most of the search engine works in principle of [full-text search<sup>106</sup>](#). We enter search items into the search box of Google. Google search engine returns the list of the websites or resources that contain the search item.

Apache Ignite supports *text-based queries* based on **Lucene** indexing. [Lucene<sup>107</sup>](#) is an open source full-text search library written in Java, which makes it easy to add search functionality to any application. Lucene does it by adding content to a full-text *index*. It then allows you to perform queries on this index. This type of index is called an *inverted* index because it inverts a page-centric data structure (page->words) to a keyword-centric data structure (word->pages) as shown in figure 6.15. You can imagine it as an index at the back of the book (unfortunately, we don't have any index part of this book due to the large size of the book).

In Lucene, a *Document* is the unit of index and search. An index can contain one or more documents. A Lucene document doesn't have to be a document in Microsoft word. For instance, if you are creating a Lucene index of *companies*, then each and every company will be representing a document in Lucene index. Lucene search can retrieve documents from the index via a Lucene *IndexSearcher*.

---

<sup>106</sup>[https://en.wikipedia.org/wiki/Full-text\\_search](https://en.wikipedia.org/wiki/Full-text_search)

<sup>107</sup>[https://en.wikipedia.org/wiki/Apache\\_Lucene](https://en.wikipedia.org/wiki/Apache_Lucene)

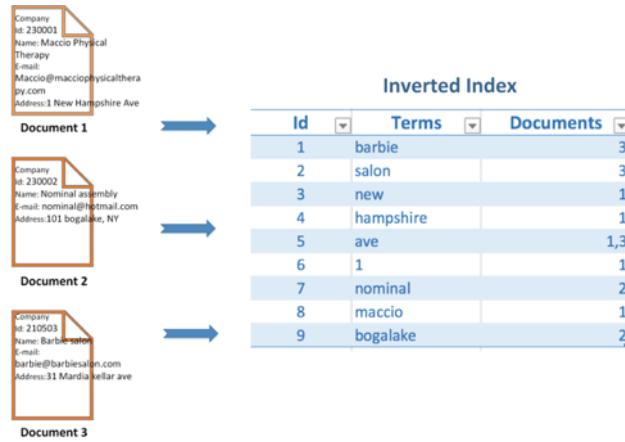


Figure 6.15

In Apache Ignite, each node contains a *local Lucene engine* that stores the index in memory. When any distributed full-text queries are executed, each node performs the search in local index via *IndexSearcher*, and send the result back to the client node, where the result is aggregated.



## Info

The Ignite cache doesn't contain the Lucene index, instead, Ignite provides an in-memory *GridLuceneDirectory* directory, which is the memory-resident implementation to store the Lucene index in memory. Ignite *GridLuceneDirectory* is very much similar to the Lucene *RAMDirectory*.

It's time to see some code and make Ignite Text-search clearer. We are going to use the same dataset from the previous subsection and extend the application to perform a few text search against cache entries. Let's modify the Maven project to add the full-text search functionality. The complete source code of the project is available at [GitHub repository<sup>108</sup>](https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/textquery).

**Step 1.** Add the following Maven dependencies into the project pom.xml file.

<sup>108</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/textquery>

**Listing 6.58**

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-indexing</artifactId>
    <version>{ignite.version}</version>
</dependency>
```

---

**Step 2.** Add `@QueryTextField` annotation to each field of the `com.blu.imdg.model.Company` class to be indexed for full-text search using Lucene as shown below.

**Listing 6.59**

---

```
public class Company implements Serializable {

    private Long id;
    private String cat;
    @QueryTextField
    private String companyName;
    @QueryTextField
    private String email;
    @QueryTextField
    private String address;
    @QueryTextField
    private String city;
    @QueryTextField
    private String state;
    @QueryTextField
    private String zipCode;
    private String phoneNumber;
    private String faxNumber;
    private String sicCode;
    private String sicDescription;
    @QueryTextField
    private String webAddress;
}
```

---

**Tip**

You can also use the annotation to *getter method* of the class. Field or getter method that doesn't have the annotation will not be indexed.

**Step 3.** Add a new static method called `textQuery()` in the `TextQueryExample` Java class with the following contents:

**Listing 6.60**

---

```
private static void textQuery() {
    IgniteCache<Integer, Company> cache = Ignition.ignite().cache(COMPANY_CACHE_NAME);

    // Query for all companies which has a text "John".
    TextQuery<Integer, Company> john = new TextQuery<>(Company.class, "John");

    // Query for all companies which has a text "beauty salon".
    TextQuery<Integer, Company> primavera = new TextQuery<>(Company.class, "beauty salon"\n);
    log("==So many companies with information about 'John'==", cache.query(john).getAll()\n);
    log("==A company which name is starting with 'Beauty salon'==", cache.query(primavera\n).getAll());
}
```

---

The above pseudo-code is very much similar to the `scanQuery()` method from the previous section. First, we retrieve the cache `company` and creates two text queries with text `John` and `beauty salon`. Then executes the query with the texts which returns the list of the companies and prints the results into the console. Under the hood, the Lucene engine looks for the text `Jhon`, and `beauty salon` in the indexes across all the fields marked `@QueryTextField`. All those records that have at least one field satisfying the text match condition will be returned.

**Step 4.** Build and run the application with the following command:

**Listing 6.61**

---

```
mvn clean install && java -jar target/textquery-runnable.jar textquery
```

---

It produces a trace similar to this.

```
14:28:06.784 [main] INFO c.b.i.TextQueryExample - ==A company which name with ' beauty salon'=
14:28:06.784 [main] INFO c.b.i.TextQueryExample -   Entry [key=233196, val=Company{id=233196, cat='B', companyName='Bardot Beauty', email='bardotbeauty@aol.com', ...
14:28:06.784 [main] INFO c.b.i.TextQueryExample -   Entry [key=2331457, val=Company{id=2331457, cat='0', companyName='Oriental Barber & Beauty Salon', email='spn@se...
14:28:06.784 [main] INFO c.b.i.TextQueryExample -   Entry [key=231951, val=Company{id=231951, cat='B', companyName='Bleak Beauty Books', email='cs1mt@aol.com', ad...
14:28:06.784 [main] INFO c.b.i.TextQueryExample -   Entry [key=234406, val=Company{id=234406, cat='1', companyName='J & D BEAUTY PRODUCTS', email='jdbtyar@aol.co...
14:28:06.784 [main] INFO c.b.i.TextQueryExample -   Entry [key=234218, val=Company{id=234218, cat='U', companyName='Ultimate Make-Up & Beauty Applications', email...
14:28:06.784 [main] INFO c.b.i.TextQueryExample -   Entry [key=231307, val=Company{id=231307, cat='1', companyName='JMC La Femina Beauty Salons Inc.', email='cyar...
14:28:06.784 [main] INFO c.b.i.TextQueryExample -   Entry [key=231419, val=Company{id=231419, cat='0', companyName='Devra Bader Skin Care & Beauty Spa', email='in...
```

**Figure 6.16**

You can modify this application with different search criteria. In real-world scenarios, the Ignite built-in text queries should be sufficient to perform the full-text search.

## Affinity collocation based data modeling

Ignite as a distributed database system does not support any arbitrary foreign key relationship between tables/caches which would introduce serious hurdles to how data is stored in a highly distributed environment. Instead, it supports the *affinity collocation* of data, which provides both a method for implementing logical relational constraints as well as a way to collocate related data for better query performance.

However, how do you tell Ignite to store related tables or caches on the same cluster node? You can optionally define parent-child relationships between tables if you want Apache Ignite to **physically** co-locate their data for efficient retrieval. For example, if you have an *Employee* table and a *Department* table, and your application frequently fetches all the employees for a department, you can define Employee tables as a child table of the Department table. In doing so, you are declaring a data locality relationship between two logically independent tables: you are telling Ignite to physically store all the employee's data with specified departments on the same node.



### Info

Similar concepts are adopted in other distributed databases like MemSQL, VoltDB or Google cloud spanner. In cloud spanner, the concept is called *interleaving of data*.

Apache Ignite provides *affinity key* concept which assists you to store related datasets on the same Ignite server nodes. For instance, assume *EMPLOYEE* and *DEPARTMENT* are PARTITIONED tables, by using `AffinityKey(int empno, int deptno)` function of the Employee and Department ID, Ignite storage engine makes sure that all the Employees data by specific Department will be stored on the same node like it's shown in the illustration below:

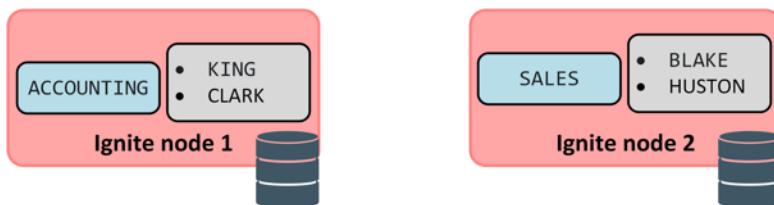


Figure 6.17

The figure 6.17 shows that all the employees regarding the department *Accounting* are stored on Ignite node 1, while the Employees related to the department *Sales* are stored

on Ignite node 2. You can define the affinity key using *CREATE TABLE DDL* statement or *SqlQueryField* annotation depending on the way you configure your tables or caches.

Let's define two tables through SQL to achieve such data distribution. We will start with the previously used Department table:

**Listing 6.62**

---

```
CREATE TABLE dept
(
    deptno LONG,
    dname VARCHAR,
    loc  VARCHAR,
    CONSTRAINT pk_dept PRIMARY KEY (deptno)
);
```

---

The department will be a partitioned table and distributed across the cluster node. The table contains one primary key column: *deptno*, which will be used by the Ignite storage engine to calculate a node in the cluster that will own a Department's entry. Next, we create the Employee table as shown below:

**Listing 6.63**

---

```
CREATE TABLE emp
(
    empno  LONG,
    ename  VARCHAR,
    job    VARCHAR,
    mgr    INTEGER,
    hiredate DATE,
    sal    LONG,
    comm   LONG,
    deptno LONG,
    CONSTRAINT pk_emp PRIMARY KEY (empno, deptno)
) with "affinityKey=deptno";
```

---

Note that, the primary key of the *EMP* table is composed of two columns: *empno* and *deptno*, to associate each employee with its department. The table *EMP* also has an additional non ANSI-99 *affinityKey* parameter set to column *deptno*. This key instructs Ignite storage engine to store an employee on the cluster node of its *deptno*.

You can achieve the same data model using Java with *@AffinityKeyMapped* annotation. Let's look at the *AffinityKey* Java class implementation:

**Listing 6.64**

---

```
public class EmployeeKey implements Serializable {  
  
    private final int empNo;  
  
    @AffinityKeyMapped  
    private final int deptNo;  
  
    public EmployeeKey(int empNo, int deptNo) {  
        this.empNo = empNo;  
        this.deptNo = deptNo;  
    }  
}
```

---

The *EmployeeKey* Java class contains two fields: *empNo* and *deptNo*. We used `@AffinityKeyMapped` annotation to specify custom key-to-node affinity. This annotation allows marking a field in the cache key object that will be used as an affinity key. Here, the `deptNo` key will be used to determine a node on which given key will be stored. Note that, a Java class can have only one field annotated with `@AffinityKeyMapped` annotation.

Now, we can define the Java class for the *Employee* cache.

**Listing 6.65**

---

```
public class Employee implements Serializable {  
  
    private static final AtomicInteger GENERATED_ID = new AtomicInteger();  
  
    @QuerySqlField(index = true)  
    private Integer empno;  
    @QuerySqlField  
    private String ename;  
    @QueryTextField  
    private String job;  
    @QuerySqlField  
    private Integer mgr;  
    @QuerySqlField  
    private LocalDate hiredate;  
    @QuerySqlField  
    private Integer sal;  
    @QuerySqlField(index = true)  
    private Integer deptno;
```

```
private transient EmployeeKey key;

//Affinity employee key
public EmployeeKey getKey() {
    if (key == null) {
        key = new EmployeeKey(empno, deptno);
    }
    return key;
}
```

---

Most part of the above Employee class is the same as before, except the affinity employee key section. Here, field `key` is a type of the `EmployeeKey` which we defined before. `EmployeeKey` type is the key mapped to the Employee ID and the Department ID, where department id (`deptno`) is the Employee's affinity key. Java class for the Department cache will remain unchanged. See the full source code of the Affinity key implementation on [GitHub repository<sup>109</sup>](#).

That's how data modeling for affinity collocation done right in Apache Ignite. Next, we discuss the Ignite distributed SQL joins in both *collocated* and *non-collocated* fashions.

## Collocated distributed joins

Ignite supports two types of SQL joins through JDBC SQL or SqlQuery API: *collocated* and *non-collocated*. The collocated joins perform SQL query against tables that are already collocated and stored on the same node. Therefore, it omits unnecessary data movement between cluster nodes. This approach makes the SQL JOIN's work faster.



### Info

Apache Ignite *collocated* JOIN is the most efficient and performant JOINs you can get in the distributed databases.

From the user perspective, you do not have to make any extra efforts to execute collocated distributed SQL joins. It is transparent if the tables or caches already created with an affinity key. Let's quickly run a few examples and see how they work.

**Step 1.** Clone or download the DDL/DML scripts for examples from the [GitHub repository<sup>110</sup>](#).

<sup>109</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/sqlqueries/src/main/java/com/blu/imdg/model>

<sup>110</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/distributed-joins>

**Step 2.** Start an Ignite server node and an SQLLINE command console on the different terminals as shown below.

**Listing 6.66**

---

```
$ ignite.sh
$ sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
```

---

**Step 3.** Run the following DDL script to create the *EMP* and *DEPT* tables.

**Listing 6.67**

---

```
0: jdbc:ignite:thin://127.0.0.1/> !run /PATH_TO_THE_SCRIPT/emp-dept-affinity-ddl.sql
```

---

The above `run` command executes the DDL script, create *Department* and *Employee* tables. Note that, the Employee table uses an affinity key on the `deptno` column as we described earlier.

**Step 4.** Next, insert some data into the tables. Execute the DML scripts through SQLLINE JDBC client as shown below.

**Listing 6.68**

---

```
0: jdbc:ignite:thin://127.0.0.1/> !run /PATH_TO_THE_SCRIPT/DEPT-DML.sql
0: jdbc:ignite:thin://127.0.0.1/> !run /PATH_TO_THE_SCRIPT/EMP-DML.sql
```

---

The `DEPT-DML.sql` script inserts 100 entries into the Department table, and the `EMP-DML.sql` script inserts 1000 entries into the Employee table respectively. Now, the tables are populated, and we are ready to execute SQL joins against them.

**Step 5.** Run the following SELECT statement to count the number of employees in each departments.

**Listing 6.69**


---

```
SELECT Count(empno),
    b.deptno,
    dname
FROM emp a,
    dept b
WHERE a.deptno = b.deptno
GROUP BY b.deptno,
    dname
ORDER BY b.deptno;
```

---

The following output shows the query result. The result is ordered by the department *id*.

COUNT(EMPNO)	DEPTNO	DNAME
8	1	Engineering
11	2	Human Resources
10	3	Research and Development
9	4	Human Resources
15	5	Services
8	6	Marketing

Figure 6.18

let's execute one more query to find out the employees working in *San Carlos*.

**Listing 6.69**


---

```
SELECT empno,
    ename,
    loc
FROM emp a,
    dept b
WHERE a.deptno = b.deptno
AND loc = 'San Carlos';
```

---

The query returns all the employee's name and id working in city San Carlos. Let's explain what's happened under the cover. The figure 6-19 illustrates what's going on Ignite cluster when you are executing SQL query with collocated tables.

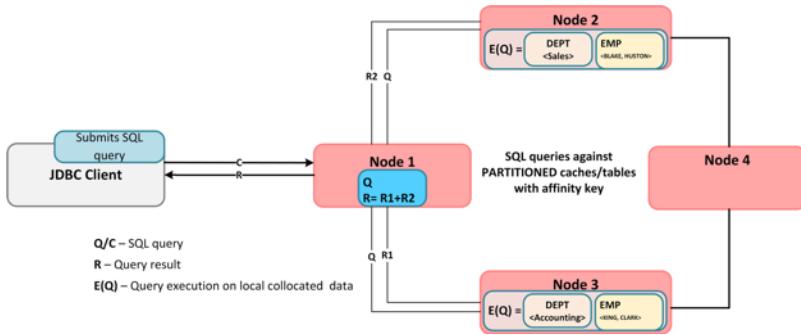


Figure 6.19

Let's details the execution flow:

- *Phase C.* JDBC client submits the SQL query to the Ignite node.
- *Phase Q.* Ignite node 1 splits the query into multiple in-memory map queries, and a single reduce query, because both EMP and DEPT tables are PARTITIONED.
- *Phase E(Q).* All the Ignite nodes that receive the SQL query runs the query against local data. So far we are using the affinity key, local data contains employees with their departments.
- *Phase R1/2.* All the nodes send their intermediate result set to the Ignite node 1.
- *Phase R.* Ignite node 1, in turn, appears as a reducer, aggregates all the intermediate result sets in a single result, and send back the final result to the client application.

## Non-collocated distributed joins

In real life scenarios, it's not always possible to collocate all the tables together on the same node. Executing queries over non-collocated (not related by affinity key) tables can return an incomplete and inconsistent query result because, at the join phase a node uses the data that is available locally only. In such a situation, you can use *non-collocated* distributed joins against tables that are not related by an affinity key. The non-collocated JOINS as the name suggests is a way to execute queries when there is no way to achieve affinity collocation for some tables, but they still need to be joined. This type of JOIN is slower because they might involve data movement between cluster nodes during JOIN time.

Non-collocated joins are not enabled by default. You can enable the non-collocated SQL joins for the specified query with two different ways:

- *SqlQuery API.* By setting the `sqlQuery.setDistributedJoins(true)` parameter.

- JDBC. By adding the `distributedJoins=true` parameter to a connection URL. An example of enabling non-collocated joins through SQLLINE `sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1?distributedJoins=true`.

When this parameter is enabled, the node to which the query was mapped will request for the missing data (data that are not presented locally) from the remote nodes by sending either broadcast or unicast request. The execution flow is illustrated in figure 6.20.

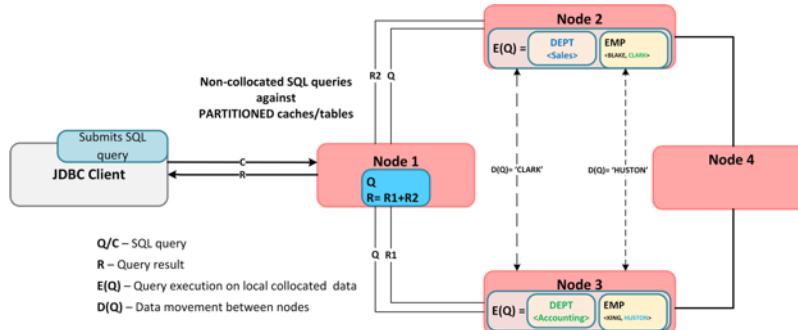


Figure 6.20

- Phase C. JDBC client submits the SQL query to the Ignite node.
- Phase Q. Ignite node 1 splits the query into multiple in-memory map queries, and a single reduce query, because both EMP and DEPT tables are PARTITIONED.
- Phase E(Q). All the Ignite nodes that receive the SQL query runs the query against local data.
- Phase D(Q). If any data is missing locally, it will be requested from the remote node by multicast or unicast request. For example, Ignite node 2 asked the employee HUSTON from node 3.
- Phase R1/2. All the nodes send their intermediate result set to the Ignite node 1.
- Phase R. Ignite node 1, in turn, appears as a reducer, aggregates all the intermediate result sets in a single result, and send back the final result to the client application.

If you execute the following query after creating the EMP and DEPT tables without an affinity key, you still get a complete result even though Employee is no longer collocated with its Department data.

```
select e.ename, d.dname from EMP e, DEPT d where e.deptno = d.deptno;
```

In this case, a broadcast request will be sent from a node to all other nodes (Phase D(Q)) in the cluster. However, you can force the Ignite SQL engine to switch from the broadcast messaging to *unicast messaging*. The following modification of the above query may enable the unicast messaging.

```
select e.ename, d.dname from EMP e, DEMT d where e.deptno = d._key;
```

With this query, if the SQL engine decides to execute the query against the Employee table by joining the Department table. First, it will send a unicast request to nodes that store Departments with `d._key`, where `_key` is a particular index that is used in Ignite SQL queries and it refers to tables primary key.

From Ignite 1.7.0 version, non-collocated distributed JOINS open the way to execute very complex analytical or ad-hoc queries over non-collocate data. However, do not overuse this approach in a high-load production environment because, with the non-collocated distributed query, there will be much more network roundtrip and data movement between the nodes during query execution.

## Spring Data integration

When we develop an application which interacts with a database, we write a set of SQL queries and bind them to the Java objects for saving and retrieving them. This standard approach to persistence via [Data Access Objects<sup>111</sup>](#) is monotonous and contains a lot of boilerplate code. [Spring Data<sup>112</sup>](#) simplifies this process of creating the data-driven application with a new way to access data by removing the DAO implementations entirely.

In a nutshell, Spring Data provides a unified and easy way to access the different persistence store, both relational database systems, and NoSQL data stores. It's adding another layer of abstraction and defining a standards-based design to support the persistence layer in a Spring context. You do not have to write any more code for CRUD operations with Spring Data. Instead, you define an interface, Spring Data generates (no CGLib nor byte-code generation) the actual implementation on the fly for you.

Spring Data introduced the concept of *Repository*. It acts as an adapter, takes the domain class to manage as well as the id type of the domain class as type arguments. This `Repository` interface is the central interface in Spring data repository concept. The primary purpose of

---

<sup>111</sup>[https://en.wikipedia.org/wiki/Data\\_access\\_object](https://en.wikipedia.org/wiki/Data_access_object)

<sup>112</sup><https://spring.io/projects/spring-data>

this interface is to capture the domain types to work, intercepts and routes all the calls to the appropriate implementation.



## Info

Note that, Spring Data does not generate any code. Instead, a JDK proxy instance is created programmatically using Spring's `ProxyFactory` API to back the interface, and a `MethodInterceptor` intercepts all the method calls and delegates them to the appropriate implementations.

Spring data provides various interfaces (abstraction) that enable you to work or connect with multiple data stores: `CrudRepository`, `PagingAndSortingRepository`. All of them extends the marker interface `Repository`.

**CrudRepository**. This is the main interface or abstraction in Spring Data. It provides generic CRUD operations of the underlying data store. It extends `Repository` interface which is the base class for all the repositories providing access to data stores. All the methods in the `CrudRepository` are shown below.

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

**PagingAndSortingRepository.** Another essential interface in Spring Data is *PagingAndSortingRepository*. This interface extends the CrudRepository and provides extra options of sorting and paginate your data using Pageable interface. The interface is shown in the next listing.

```
public interface PagingAndSortingRepository<T, ID extends Serializable>

    extends CrudRepository<T, ID> {

    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);

}
```

A high-level view of the Spring data architecture is shown in figure 6.21.

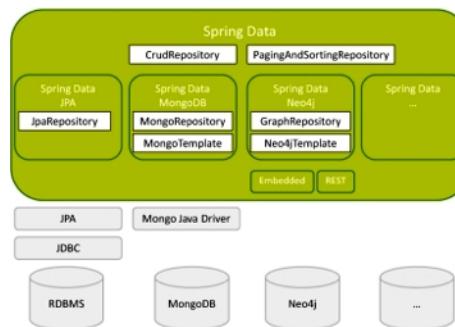


Figure 6.21

Apache Ignite provides an *IgniteRepository*<sup>113</sup> interface that extends the basic capabilities of the Spring data CrudRepository (see the figure 6.22) From version 2.0, which in turns supports:

1. Basic CRUD operations on a repository for a specific domain.
2. Access to the Apache Ignite database via Spring Data API.

---

<sup>113</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/springdata/repository/IgniteRepository.html>

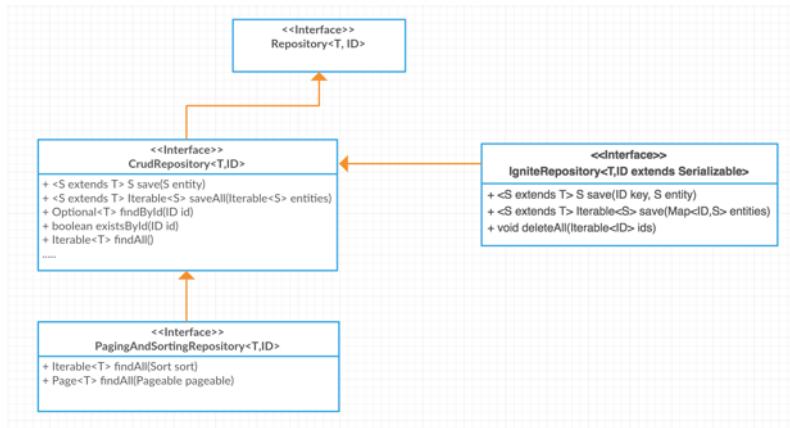


Figure 6.22

Each custom Spring Data repositories must be extended the *IgniteRepository* interface to store and query data located in an Ignite cluster. It means that you only need to write an interface with finder methods to query the objects. All the CRUD method for manipulating the objects will be delivered automatically. For instance:

#### Listing 6.70

---

```

@RepositoryConfig(cacheName = "DogCache")
public interface DogRepository extends IgniteRepository<Dog, Long> {
    List<Dog> getDogByName(String name);
    Dog getDogByld (Long id);
}
  
```

---

`@RepositoryConfig` annotation should be specified to map a repository to a distributed cache. In the above example, `DogRepository` is mapped to the `DogCache`. Methods like `getDogByName(String name)` or `getDogByld(Long id)` will be automatically processed by the Spring Data and turned into SQL queries when methods get executed. In addition to the `CrudRepository` interface, Apache Ignite `IgniteRepository` interface also provides three additional methods for working with the given domain:

- `save(ID key, S entity)`: Saves a given entity by using provided key. This method is suggested to use instead of `CrudRepository.save(Object)` which generate IDs (keys) that are not unique cluster-wide.
- `save(Map<ID, S> entities)`: Saves all given keys and entities combinations. This method is suggested to use instead of `CrudRepository.save(Iterable)` which generates IDs (keys) that are not unique cluster-wide.

- `deleteAll(Iterable<ID> ids)`: Deletes all the entities for the provided ids.



## Limitations

A few operations of the `CrudRepository` interface is not supported which auto-generates Keys that are not unique cluster-wide.

`save(S entity)`

`save(Iterable entities)`

`delete(T entity)`

`delete(Iterable entities)`

Instead of these operations you can use Ignite specific methods available via the `IgniteRepository` interface.

Armed with all the above notes, we can draw a high-level architecture of using `IgniteRepository` that are illustrated in figure 6.23. An application that uses `IgniteRepository` can access to Ignite cluster through the unified Spring Data API. Another beauty of using Spring data is its polyglot capacity; Spring Data helps you to avoid locking to a specific database vendor, making it easy to switch from one database to another with minimal efforts.

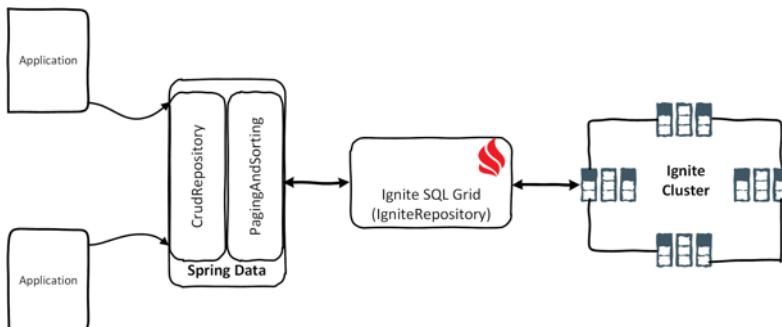


Figure 6.23

Note that, there are several Spring Data Modules specific to the data store you may want to use:

- Spring Data JPA. Connect to relational databases using ORM frameworks.
- Spring Data MongoDB. Repositories for MongoDB.

- Spring Data REST. Exposes HATEOAS RESTful resources around Spring Data repositories.
- Spring Data Redis. Repositories for Redis.

Let's look at how to build an application based on Spring data. The sample application allows us to persist a few entities of Dog and Breed into the Ignite database through Spring Data. Later we use the same application to query data from the Ignite database. The entire project is available in the [GitHub repository<sup>114</sup>](#). Before we start, let's cover the prerequisites of the project:

1. Java JDK 1.8.
2. Ignite version 2.6 or later.
3. Apache Maven version 3.2.3 or later.

**Domain model.** Our sample domain model consisted of two different entities: *Breed* and *Dog*.

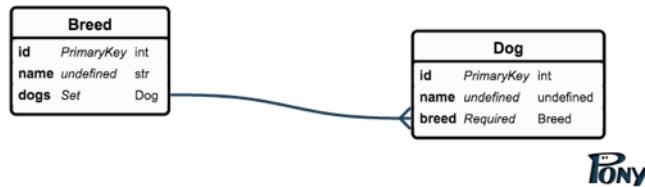


Figure 6.24

The association between Breed and Dog is *ManyToOne*. One Dog can have only one breed. Now let's get started.

**Step 1.** Let's set up the sandbox first. Create a Maven project or clone the project from the [GitHub repository<sup>115</sup>](#).

<sup>114</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/spring-data>

<sup>115</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/spring-data>

**Listing 6.71**

---

```
mvn archetype:generate -DgroupId=com.blu.imdg -DartifactId=spring-data -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

---

**Step 2.** Since we are using Spring data in this example, ignite-spring-data Maven dependency must be added.

**Listing 6.72**

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring-data</artifactId>
    <version></version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>{ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>{ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring-data</artifactId>
    <version>{ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-indexing</artifactId>
    <version>{ignite.version}</version>
</dependency>
```

---

All the Spring related dependency will be added automatically. If you are getting an error like `org.h2.result.RowFactory` is missing, add the maven H2 dependency explicitly.

**Step 3.** Now, let's map the domain model by creating the Java classes and annotating them with the required meta-information. We start with the *Breed* Java class.

**Listing 6.73**

---

```
public class Breed implements Serializable {  
  
    @QuerySqlField(index = true)  
    private Long id;  
  
    @QuerySqlField(index = true)  
    private String name;  
    // getter and setter methods goes here  
}
```

---

Here, we have a *Breed* class having two fields: the *ID*, and the *Name*. Both fields are annotated with `@QuerySqlField(index = true)` annotation so that they could be query using SQL. See the `SqlFieldsQuery` section of this chapter for more information. Create another Java class named *Dog* into the `com.blu.imdg.model` package, and add the following contents on it.

**Listing 6.74**

---

```
public class Dog implements Serializable {  
  
    @QuerySqlField(index = true)  
    private Long id;  
    @QuerySqlField(index = true)  
    private String name;  
    @QuerySqlField(index = true)  
    private Long breedid;  
    @QuerySqlField(index = true)  
    private Date birthdate;  
    // getter and setter methods goes here  
}
```

---

The *Dog* class contains four attributes: *ID*, *Name*, *BreedId*, and *birthdate*. All the fields are annotated with `@QuerySqlField(index = true)` annotation. The typical getters and setters have been omitted for brevity.

**Step 4.** Then, we will create the custom repositories for all the POJOs created earlier. Let's start with the `DogRepository` interface. Create a Java interface with name `DogRepsoitory` into the `com.blu.imdg.repositories` package as follows:

**Listing 6.75**

---

```
@RepositoryConfig(cacheName = "DogCache")
public interface DogRepository extends IgniteRepository<Dog, Long> {
    List<Dog> getDogByName(String name);
    Dog getDogById(Long id);
}
```

---

Here, the *DogRepository* extends the *IgniteRepository* interface. The type of the entity class (Dog), and ID (Long) are specified in the generic parameters on *IgniteRepository*. In other words, the type of the entity is Dog, and the type of the entity's id field is Long. By extending the *IgniteRepository*, *DogRepository* inherits several methods, including saving, deleting and finding Dog entities.

Spring Data also allows you to define other query methods by simply declaring their method signature. In the case of *DogRepository*, this is shown with `getDogByName(String name)` and the `getDogById(Long id)` method. The `@RepositoryConfig()` annotation mapped the *DogRepository* to the distributed *DogCache*.

**Tip**

Spring Data also provides the feature of custom query creation from a method name. `List<Dog> findByNameAndBreedId(String name, Long breedid)` would generate the appropriate query for the data store to return the *Dog* details.

Let's add a similar repository for the *Breed* domain as follows:

**Listing 6.76**

---

```
@RepositoryConfig(cacheName = "BreedCache")
public interface BreedRepository extends IgniteRepository<Breed, Long> {
    List<Breed> getAllBreedsByName (String name);
    @Query("SELECT id FROM Breed WHERE id = ?")
    List<Long> getById (long id, Pageable pageable);
}
```

---

As expected, the *BreedRepository* extends the *IgniteRepository* interface. The type of the entity and the ID that it works with (Breed and Long) are specified in the generic parameters on *IgniteRepository*. We also use the `@Query(queryString)` annotation, which can be used if a specific SQL query needs to be executed as a result of a method call.

Please refer to the [Spring Data documentation<sup>116</sup>](#) for more fine-grained control over the

<sup>116</sup><https://docs.spring.io/spring-data/data-jpa/docs/1.1.x/reference/html/#jpa.named-parameters>

creation of queries, such as *named queries*, *modifying queries* or applying *query hints*.

In a typical Java application, you would expect to write a class that implements DogRepository or BreedRepository. However, you do not have to write an implementation of the repository interface when using Spring Data: that makes the Spring Data elegant. Spring Data creates an implementation and analyses all the methods defined by the interfaces and tries to automatically generate queries from the method names on the fly when you run the application. Let's wire this up by creating a cache configuration.

**Step 5.** Create an Ignite cache configuration class and mark the application configuration with the `@EnableIgniteRepositories` annotation as shown below:

**Listing 6.77**

---

```
@Configuration
@EnableIgniteRepositories
public class SpringAppConfig {
    @Bean
    public Ignite igniteInstance() {
        IgniteConfiguration cfg = new IgniteConfiguration();

        // Setting some custom name for the node.
        cfg.setIgniteInstanceName("springDataNode");

        // Enabling peer-class loading feature.
        cfg.setPeerClassLoadingEnabled(true);

        // Defining and creating a new cache to be used by Ignite Spring Data
        // repository.
        CacheConfiguration ccfgDog = new CacheConfiguration("DogCache");
        CacheConfiguration ccfgBreed = new CacheConfiguration("BreedCache");

        // Setting SQL schema for the cache.
        ccfgBreed.setIndexedTypes(Long.class, Breed.class);
        ccfgDog.setIndexedTypes(Long.class, Dog.class);

        cfg.setCacheConfiguration(new CacheConfiguration[]{ccfgDog, ccfgBreed});

        return Ignition.start(cfg);
    }
}
```

---

`@EnableIgniteRepositories` annotation activates the Apache Ignite repositories. And the `igniteInstance()` method creates and passes the Ignite instance to `IgniteRepositoryFactoryBean`

to get access to the Apache Ignite cluster. We've also defined and set the DogCache and BreedCache configurations. The `setIndexedTypes()` method sets the SQL schema for the cache. We also enable the Ignite peer-class loading feature for inter-node byte-code exchange.

**Step 6.** Let's composed everything together by registering the configuration in a Spring application context once all the configurations and repositories are ready to be used. Here you create an *APP* Java class with all the components.

Listing 6.78

---

```
public class App
{
    private static AnnotationConfigApplicationContext ctx;
    private static BreedRepository breedRepository;
    private static DogRepository dogRepository;

    public static void main( String[] args )
    {
        System.out.println( "Spring Data Example!" );

        ctx = new AnnotationConfigApplicationContext();
        ctx.register(SpringAppConfig.class);
        ctx.refresh();

        breedRepository = ctx.getBean(BreedRepository.class);
        dogRepository = ctx.getBean(DogRepository.class);

        //fill the repository with data and Save
        Breed collie = new Breed();
        collie.setId(1L);
        collie.setName("collie");
        //save Breed with name collie
        breedRepository.save(1L, collie);

        System.out.println("Add one breed in the repository!");
        // Query the breed
        List<Breed> getAllBreeds = breedRepository.getAllBreedsByName("collie");

        for(Breed breed : getAllBreeds){
            System.out.println("Breed:" + breed);
        }
        //Add some dogs
        Dog dina = new Dog();
        dina.setName("dina");
```

```
dina.setId(1L);
dina.setBreedId(1L);
dina.setBirthdate(new Date(System.currentTimeMillis()));
//Save Dina
dogRepository.save(2L,dina);
System.out.println("Dog dina save into the cache!");

//Query the Dog Dina
List<Dog> dogs = dogRepository.getDogByName("dina");
for(Dog dog : dogs){
    System.out.println("Dog:"+ dog);
}
}
```

The above code snippet is very easy. First, we create a Spring annotated application context and registered our repositories. Then, we fetch the *BreedRepository* and the *DogRepository* from the Spring Application context. Next, we save a handful of *Breed* and *Dog* objects, demonstrating the `save()` method and setting up some data to work. Finally, we employ the `getDogByName("dina")` method to fetch Dog object with name **Dina** from the database.

**Step 7.** Let's build and run the application. Execute the following command:

**Listing 6.79**

---

```
mvn clean install && exec:java -Dexec.mainClass=com.blu.imdg.App
```

---

You should find a lot of log messages in the console as shown below:

```

es.
[21:48:33] Security status [authentication=off, tls/ssl=off]
[21:48:35] Performance suggestions for grid 'springDataNode' (fix if possible)
[21:48:35] To disable, set -DIGNITE_PERFORMANCE_SUGGESTIONS_DISABLED=true
[21:48:35]   ^-- Enable G1 Garbage Collector (add '-XX:+UseG1GC' to JVM options)
[21:48:35]   ^-- Specify JVM heap max size (add '-Xmx<size>[g|G|m|M|k|K]' to JVM options)
[21:48:35]   ^-- Set max direct memory size if getting 'OOME: Direct buffer memory' (add '-XX:MaxDirectMemorySize=<size>[g|G|m|M|k|K]' to JVM options)
[21:48:35]   ^-- Disable processing of calls to System.gc() (add '-XX:+DisableExplicitGC' to JVM options)
[21:48:35] Refer to this page for more performance suggestions: https://apacheignite.readme.io/docs/jvm-and-system-tuning
[21:48:35]
[21:48:35] To start Console Management & Monitoring run ignitevisorcmd.{sh|bat}
[21:48:35]
[21:48:35] Ignite node started OK (id=f510a337, instance name=springDataNode)
[21:48:35] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=3.6GB]
[21:48:35]   ^-- Node [id=F510A337-6FFF-4BF6-A159-6A3011C66D97, clusterState=ACTIVE]
[21:48:35] Data Regions Configured:
[21:48:35]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
Add one breed in the repository!
Breed:Breed{id='1', name='collie'}
Dog dina save into the cache!
Dog:Dog{id=1, name='dina', breedid=1, birthdate=2018-11-30}

```

Figure 6.25

The log messages confirm that two entries (*dina* and *breed-collie*) have been flushed into the Ignite cache and retrieved the dog *Dina* from the cache. Let's explore the cache through Ignite Visor.

#	Name(@)	Mode	Size (Heap / Off-heap)
0	BreedCache(@c0)	PARTITIONED	min: 1 (1 / 0) avg: 1.00 (1.00 / 0.00) max: 1 (1 / 0)
1	DogCache(@c1)	PARTITIONED	min: 1 (1 / 0) avg: 1.00 (1.00 / 0.00) max: 1 (1 / 0)

Figure 6.26

Two different Ignite caches have been created for the entities: *Breed* and *Dog*. If we scan the cache entries of the Dog cache, we should find the following entity on it.

Entries in cache: DogCache			
Key Class	Key	Value Class	Value
java.lang.Long	1	com.blu.ignite.BreedObjectImpl	{breedid=1, name=collie}
java.lang.Long	2	com.blu.ignite.DogObjectImpl	{birthdate=2000-07-31 00:00:00, breedid=1, name=dina}

Figure 6.27

Entity *Dina* has been persisted into the cache with the key of the *Breed Collie*. At these moments, in addition to Spring Data API, we have a few more alternatives to working with caches/tables created in Ignite database. First, we can use the *SQLLINE* tool for retrieving the same data:

```
0: jdbc:ignite:thin://127.0.0.1/> SELECT * FROM "DogCache".DOG;
+----+----+----+
| ID | NAME | BREEDID |
+----+----+----+
| 1  | dina | 1        |
+----+----+----+
1 row selected (0.006 seconds)
```

Figure 6.28

Alternatively, by using the standard Ignite SQL API as follows:

```
SqlFieldsQuery sql = new SqlFieldsQuery(
    "select * from Dog where name = 'dina'");
```

Finally, we can use Ignite cache API to fetch the Dog object from the cache as shown below:

```
IgniteCache<Integer, Dog> cache = ignite.cache("DogCache");
Dog dina = cache.get(2L);
```

This short but a complete application shows you the way of using Spring Data framework with Ignite database. It's not the end of the possibilities of using Spring Data with Ignite. You can use Ignite affinity key concept to store related datasets in the same Ignite node when using Spring Data. Moreover, you can execute Ignite collocated distributed SQL joins by using the `@Query(queryString)` annotation in the custom repository interface. In the next section of this chapter, we will use JPA with Ignite database for developing an application.

## Apache Ignite with JPA

The Java Persistence API (JPA) is a standard API for accessing databases from within Java applications and has been around for a while. The main advantage of JPA over other approaches is that in JPA data is represented by classes and objects rather than tables and records as in JDBC. With JPA you won't have to commit to a specific data store, and you can switch to another data store with minimum efforts.

JPA<sup>117</sup> is merely a specification, and it needs an implementation to interact with databases like Oracle, IBM DB2, MariaDB. The well-known JPA implementations are Hibernate, TopLink, MyBatis and Open JPA. These implementations provide an object-relational mapping between the data representation in the database (tables, views) and the Java application (classes and objects).

JPA was mainly focused on relational databases, and there are a few factors to consider before applying JPA into any NoSQL database:

<sup>117</sup>[https://en.wikipedia.org/wiki/Java\\_Persistence\\_API](https://en.wikipedia.org/wiki/Java_Persistence_API)

- Pure relational concepts may not apply well in NoSQL.
  - Table, Column, Joins.
- JPA queries may not be suitable for NoSQL. Application-specific access patterns typically drive NoSQL data modeling.



## Warning

Note that, if your dataset is by nature *non-domain* model centric, then JPA is not for you.

Apache Ignite is quite well fit for using JPA since it's a key-value data store. Other key-value datastore like Infispan, Oracle NoSQL, Ehcache also supported by JPA persistence as well. At this moment there are a few NoSQL JPA solutions available in today's market:

1. **Kundera**<sup>118</sup>
  - One of the first JPA implementations for NoSQL databases.
  - Supports Cassandra, MongoDB, HBase, Redis, Oracle NoSQL DB, etc.
2. **DataNucleus**<sup>119</sup>
  - Persistence layer behind Google App engine.
  - Supports MongoDB, Cassandra, Neo4J.
3. **Hibernate OGM**<sup>120</sup>
  - Using Hibernate ORM engine to persists entities into NoSQL database.
  - Supports **Apache Ignite**, MongoDB, Cassandra, Neo4j, Infispan, Ehcache.

*Hibernate OGM* is one of the most well-known and often used implementations of JPA for NoSQL datastore. It reuses Hibernate ORM's engine but persists entities into NoSQL datastore instead of an RDBMS. Hibernate OGM also supports several ways for searching entities and fetching them as Hibernate managed objects:

1. JP-QL queries.
2. Criteria query.
3. Datastore specific native queries.
4. Full-text queries, using Hibernate Search as indexing engine.

---

<sup>118</sup><https://github.com/Impetus/Kundera>

<sup>119</sup><http://www.datanucleus.org/>

<sup>120</sup><http://hibernate.org/ogm/>

Hibernate OGM talks to NoSQL database via store-specific dialects. Hibernate OGM community split the dialects into 2 categories:

- the ones maintained in the central repository: Infinispan, MongoDB, Neo4j.
- the ones maintained by the community which has been moved to separate repositories: Cassandra, CouchDB, EhCache, Apache Ignite, and Redis.

*Apache Ignite dialect* for Hibernate OGM first introduced on February 2016. Within three years of developments, it gets matured and almost ready for production use. I said almost ready for production use because, it contains multiple bugs and limitations on the current version. The most significant limitation is that the current version (5.3.1.Final) does not support Apache Ignite 2.6 or above. However, there is a pull request (patch-22) is waiting for approving.

Although, Apache Ignite JPA dialects for Hibernate OGM contains a few bugs, in this subsection we will try to use JPA over Ignite caches/tables. The project is available at [GitHub<sup>121</sup>](#), and any contributions are welcome. A high-level view of the Hibernate OGM shown in figure 6.29:

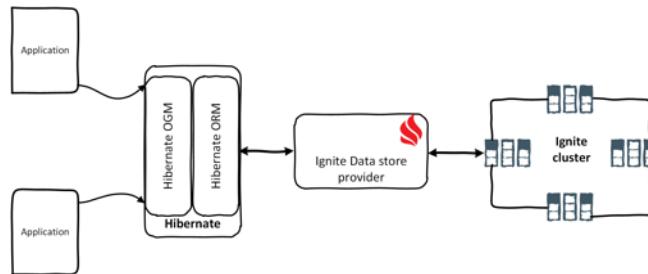


Figure 6.29

Hibernate OGM reusing the very mature Hibernate ORM core engine and trick the Ignite dialects to store data into a cluster. It plugs into Hibernate core via two main contracts named *Loader* and *Persister*. As you could expect, these load and manipulate (save/update/delete) entities into the Ignite database.

In this section, we cover the following topics:

1. Clone and build the `hibernate-ogm-ignite` project.
2. Apply a **patch** to support Ignite version 2.6.

<sup>121</sup><https://github.com/hibernate/hibernate-ogm-ignite>

3. Create an application based on JPA to persist, and query the Ignite database.

Let's have a quick look at the project prerequisites before diving into the details.

Name	Description
Apache Ignite	2.6
JDK	1.8 or above
Hibernate OGM	5.3.1.Final
Git	2.15 or above
Pull request	Upgrade Hibernate OGM to 5.4 and Apache Ignite to 2.6 #22

First, we will clone the *hibernate-ogm-ignite* project from the GitHub repository and apply a patch, and then, as usual, we develop an application to try the JPA over Ignite database. Note that, in the sample project we use the same entity model *EMP/DEPT* for all examples.

**Step 1.** Clone the *hibernate-ogm-ignite* project from the [Github repository](#)<sup>122</sup>. Open a terminal and change the current working directory to the location where you want the cloned directory to be made. Type the following command on the terminal as follows:

**Listing 6.80**

---

```
git clone https://github.com/hibernate/hibernate-ogm-ignite.git
```

---

Press enter, and your local clone copy of the project will be created.

**Step 2.** Hibernate OGM Ignite project uses Ignite version 2.4 at the moment of writing this book. Let's **apply a patch** to upgrade the project up to Ignite version 2.6. Download or copy the patch file from the [GitHub](#)<sup>123</sup> repository and save it in the same directory where you cloned the project in step 1. Apply the patch, use one of the following commands from the same directory as before.

---

<sup>122</sup><https://github.com/hibernate/hibernate-ogm-ignite>

<sup>123</sup><https://github.com/srecon/the-apache-ignite-book/blob/master/chapters/chapter-6/ignite-hibernate-ogm/22.patch>

**Listing 6.81**

---

```
git am 22.patch
```

or

---

```
git apply 22.patch
```



## Warning

Take a look at the *properties* section of the project POM.XML file. If it uses Ignite version >=2.6, then do not apply the patch.

**Step 3.** Build the Hibernate OGM Ignite project. Type the following command on the same terminal:

**Listing 6.82**

---

```
mvn clean install -Dmaven.test.skip=true -DskipDocs -DskipDistro -DskipITs -s ./settings-\nexample.xml
```

---

Maven downloads all the necessary libraries, build and deploy artifacts in your local Maven repository. Note that, we use a pre-configured Maven setting file (`settings-example.xml`) to download many multiple libraries from the JBoss central repository.

**Step 4.** Clone or download the `ignite-hibernate-ogm` project from the book [GitHub<sup>124</sup>](#) repository. If you create your individual Maven project, add these dependencies into your `pom.xml` file.

---

<sup>124</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/ignite-hibernate-ogm>

Listing 6.83

---

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-bom</artifactId>
    <version>5.4.0-SNAPSHOT</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-ignite</artifactId>
    <version>5.4.0-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.jboss.narayana.jta</groupId>
    <artifactId>narayana-jta</artifactId>
    <version>5.3.3.Final</version>
</dependency>
<dependency>
    <groupId>org.jboss.spec.javax.transaction</groupId>
    <artifactId>jboss-transaction-api_1.2_spec</artifactId>
    <version>1.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.7</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.7</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>2.6.0</version>
</dependency>
```

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-indexing</artifactId>
    <version>2.6.0</version>
</dependency>
```

---

The dependencies to focus on:

1. The Hibernate OGM Ignite version 5.4.0-SNAPSHOT for working with Apache Ignite. This maven dependency pulls all other required modules such as Hibernate OGM core and so on.
2. Hibernate JPA implementation for working with JPA.
3. JBoss transaction API specification and Narayana JTA implementation.

**Step 5.** Let's map the domain model by creating the entity classes and annotating them with the required meta-information. Create a Java class with name *Department* into the package com.blu.imdg.DTO as shown below.

**Listing 6.84**

```
@Entity(name = "DEPT")
public class Department implements Serializable {

    @Id
    @GeneratedValue
    private Integer deptno;

    private String dname;

    private String loc;

    public Department() {}

    public Department(String dname, String loc) {
        this.dname = dname;
        this.loc = loc;
    }
}
```

```
}

// setter/getter methods are omitted

}
```

---

The entity is marked as a JPA annotation of `@Entity`, while the `@Id` annotates other properties such as `deptno`. By the `@ID` annotation, Hibernate will take care to generate the primary key for the entity object. Here, we use the default ID generation type. This type can be numerical or UUID. In our cases, the type will be a numeric value and will be unique at the database level. Let's create and annotate the `Employee` entity.

Listing 6.85

```
@Entity(name = "EMP")
public class Employee implements Serializable{

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE,
        generator = "table-generator")
    @TableGenerator(name = "table-generator",
        table = "emp_ids",
        pkColumnName = "seq_id",
        valueColumnName = "seq_value")
    private Integer empno;

    private String ename;

    private String job;

    private Integer mgr;

    private LocalDate hiredate;

    private Integer sal;

    @ManyToOne()
    private Department dept;

    public Employee() { }

    public Employee(String ename, String job, Integer mgr, LocalDate hiredate,
        Integer sal, Department dept) {
        this.ename = ename;
        this.job = job;
```

```
this.mgr = mgr;
this.hiredate = hiredate;
this.sal = sal;
this.dept = dept;
}
// setter/getter methods are omitted
}
```

---

We annotated the Employee entity with `@Entity` and `@ID` annotation as well. Also, we add `@ManyToOne` annotation to make the association with the Department entity. Note that, we use `@TableGenerator` annotation for creating primary key automatically. The TableGenerator uses an underlying database table that holds segments of identifier generation values. We also customize the table name, `pkColumnName` and `valueColumnName` using the `@TableGenerator` annotation.



## Warning

`TableGenerator` strategy uses a database table for generating the primary key value automatically, and does not scale well and synchronized. So, this strategy can impact the negative effect on application performance.

**Step 6.** Let's define the persistence context into `META-INF/persistence.xml` file. Create the `persistence.xml` file into the `/src/resources/META-INF` directory.

**Listing 6.86**

```
<?xml version="1.0"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.c
om/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">

    <persistence-unit name="ogm-jpa-tutorial" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>
        <properties>

            <property name="com.arjuna.ats.jta.jtaTMImplementation" value="com.arjuna.ats\
.internal.jta.transaction.arjunacore.TransactionManagerImpl"/>
            <property name="com.arjuna.ats.jta.jtaUTImplementation" value="com.arjuna.ats\
```

```
.internal.jta.transaction.arjunacore.UserTransactionImpl"/>

<property name="hibernate.ogm.datastore.provider" value="org.hibernate.ogm.da\
tastore.ignite.impl.IgniteDatastoreProvider"/>
<!--<property name="hibernate.ogm.ignite.configuration_class_name" value="com\b\
.blu.imdg.exampleOgm.ConfigurationMaker"/-->
<property name="hibernate.ogm.ignite.configuration_resource_name" value="igni\
te-config.xml"/>
<property name="hibernate.ogm.ignite.instance_name" value="ignite-ogm"/>

</properties>
</persistence-unit>
</persistence>
```

---

If you are familiar with JPA, this persistence definition unit should look very common to you. The main difference of using the classic Hibernate ORM on top of a relational database is the provider class we need to specify for Hibernate OGM: `hibernate.ogm.datastore.provider`. Another mandatory property for the persistence context is `Ignite configuration`. Ignite configuration property can be applied in two different ways:

- `hibernate.ogm.ignite.configuration_resource_name`. A resource file containing all the Ignite configuration properties (`clientMode`, `discoverySpi`, `peerClassLoadingEnabled`). Usually, it is an Ignite Spring configuration file.
- `hibernate.ogm.ignite.configuration_class_name`. A Java class that returns the `IgniteConfiguration` instance to connect to the Ignite cluster.

We use `ignite-config.xml` file as a resource for connecting to the Ignite cluster. See the `ConfigurationMaker` Java class in the `com.blu.imdg.exampleOgm` package as an example of `hibernate.ogm.ignite.configuration_class_name` property. Also, note that we use `RESOURCE_LOCAL` instead of `JTA`. If you want to use `JTA`, you should provide a particular `JTA` implementation such as `JBoss`.

**Step 7.** Create an XML file named `ignite-config.xml` into the `resources` directory, and copy the following content into it.

**Listing 6.87**

---

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="clientMode" value="false"/>
        <property name="igniteInstanceName" value="ignite-ogm"/>

        <property name="peerClassLoadingEnabled" value="true"/>

        <property name="discoverySpi">
            <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
                <property name="ipFinder">
                    <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDiscoveryVmIpFinder">
                        <property name="addresses">
                            <list>
                                <value>127.0.0.1:47500..47509</value>
                            </list>
                        </property>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

---

Probably you have already familiar with all the configurations used in the above file.

**Step 8.** Let's store a set of entities, and retrieve them from the Ignite caches. Create a Java class with the name `App`, and add the following content:

**Listing 6.88**

---

```
public class App {
    final static Logger LOGGER = LogManager.getLogger(App.class.getName());

    public static void main(String[] args) {
        LOGGER.info("-----Ignite Hibernate JPA example-----");
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("ogm-jpa-tutori\
al");

        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();

        LOGGER.info("=====Populate DEPT table with rows=====");
        Department department_10 = new Department( "ACCOUNTING", "NEW YORK");
        Department department_20 = new Department("RESEARCH", "DALLAS");
        Department department_30 = new Department( "SALES", "CHICAGO");

        // persist Departments
        em.persist(department_10);
        em.persist(department_20);
        em.persist(department_30);

        LOGGER.info("=====Populate EMP table with rows=====");

        Employee emp_king = new Employee( "KING", "PRESIDENT", null, null, 5000, departme\
nt_10);
        Employee emp_blake = new Employee("BLAKE", "MANAGER", 7839, null, 2850, departmen\
t_30);
        Employee emp_clark = new Employee("CLARK", "MANAGER", 7839, null, 2450, departmen\
t_10);
        Employee emp_jones = new Employee("JONES", "MANAGER", 7839, null, 2975, departmen\
t_20);

        //persist Employees

        em.persist(emp_king);
        em.persist(emp_blake);
        em.persist(emp_clark);
        em.persist(emp_jones);

        Integer empNo = emp_king.getEmpno();
```

```
// commit the transaction
em.getTransaction().commit();

// Query some employees by JP-QL
LOGGER.info("=====Query by Employee ID=====");
Employee emp = em.find(Employee.class, empNo);

LOGGER.info("Employee KING:" + emp.getEname() + " Employee dept:" + emp.getDept()\n.getDname());

LOGGER.info("=====Query example=====");
Query query = em.createQuery("Select ename from EMP e where e.empno=:empno");
query.setParameter("empno", 1);
List<String> employeesName = query.getResultList();
employeesName.forEach(LOGGER::info);
query = em.createQuery("Select ename from EMP e where e.dept=:deptno");
query.setParameter("deptno", department_10);
employeesName = query.getResultList();
employeesName.forEach(LOGGER::info);

em.close();

}
}
```

---

First, we have created an EntityManagerFactory instance with parameter `ogm-jpa-tutorial`. Next, we derived our EntityManager from the factory; this EntityManager will be our entry point for the persistence entities. We opened a transaction from the EntityManager and populated some data of *departments* and *employees* to persist into the Ignite cache. We use the `persist()` method to store the generated departments and employees into the particular Ignite caches.

Then, we use the Java Persistence Query Langage (JP-QL) to query the Ignite caches. The JP-QL is a query language defined as part of the Java Persistence API (JPA) specification<sup>125</sup> and designed for working with entities to be database independent. First, we use the `find()` method to retrieve an employee by `id`. Next, we use JPA Query interface to execute a few SELECT queries, and returns the query results.

**Step 9.** Now, build and launch the application. Type the following command into a terminal from the project home directory.:

---

<sup>125</sup><http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-eval-oth-JSpec/>

**Listing 6.89**


---

```
mvn clean install && mvn exec:java -Dexec.mainClass=com.blu.imdg.App
```

---

The above command generates a lot of logs into the terminal.

```
[DEBUG] 06:13:24.158 [com.blu.imdg.App.main()] DotNode - getDataType() : dept -> org.hibernate.type.ManyToOneType(com.blu.imdg.DT
O.Department)
[DEBUG] 06:13:24.158 [com.blu.imdg.App.main()] DotNode - dereferenceShortcut() : property dept in com.blu.imdg.DTO.Employee does
not require a join.
[DEBUG] 06:13:24.167 [com.blu.imdg.App.main()] DotNode - Terminal getPropertyPath = [dept]
[DEBUG] 06:13:24.167 [com.blu.imdg.App.main()] FromReferenceNode - Resolved : e.dept -> employee0_.dept_deptno
[DEBUG] 06:13:24.167 [com.blu.imdg.App.main()] HqlSqlBaseWalker - select : finishing up [level=1, statement=select]
[DEBUG] 06:13:24.167 [com.blu.imdg.App.main()] HqlSqlWalker - processQuery() : ( SELECT {select clause} { employee0_.ename {syn
thetic-alias} ename } ) ( FromClause[level=1] EMP employee0_ ) ( where (= ( employee0_.dept_deptno employee0_.empno dept ) ? ) )
)
[DEBUG] 06:13:24.167 [com.blu.imdg.App.main()] JoinProcessor - Using FROM fragment [EMP employee0_]
[DEBUG] 06:13:24.167 [com.blu.imdg.App.main()] SyntheticAndFactory - Using unprocessed WHERE-fragment [null]
[DEBUG] 06:13:24.167 [com.blu.imdg.App.main()] SyntheticAndFactory - Using processed WHERE-fragment [null]
[DEBUG] 06:13:24.167 [com.blu.imdg.App.main()] HqlSqlBaseWalker - select >> end [level=1, statement=select]
[DEBUG] 06:13:24.178 [com.blu.imdg.App.main()] ErrorTracker - throwQueryException() : no errors
[DEBUG] 06:13:24.179 [com.blu.imdg.App.main()] QueryParser - Processed tree: (QUERY (QUERY_SPEC (SELECT_FROM (from (PERSISTER_SPA
CE (ENTITY_PERSISTER_REF com.blu.imdg.DTO.Employee e))) (Select (SELECT_LIST (SELECT_ITEM (PATH ename)))))) (where (<= (PATH (. e d
ept)) deptno)))
[DEBUG] 06:13:24.179 [com.blu.imdg.App.main()] QueryParser - Processed tree: (QUERY (QUERY_SPEC (SELECT_FROM (from (PERSISTER_SPA
CE (ENTITY_PERSISTER_REF com.blu.imdg.DTO.Employee e))) (Select (SELECT_LIST (SELECT_ITEM (PATH ename)))))) (where (<= (PATH (. e d
ept)) deptno)))
[DEBUG] 06:13:24.179 [com.blu.imdg.App.main()] SQL - SELECT e.ename as col_0 FROM EMP e WHERE e.dept_deptno=?1
[DEBUG] 06:13:24.182 [com.blu.imdg.App.main()] LogicalConnectionManagedImpl - Initiating JDBC connection release from afterTransa
ction
[INFO ] 06:13:24.182 [com.blu.imdg.App.main()] App - KING
[INFO ] 06:13:24.182 [com.blu.imdg.App.main()] App - CLARK
```

---

Figure 6.30

The log messages confirm that entries have been flushed into the Ignite cache, and then retrieve employees *KING* and *CLARK* from the cache. Let's explore the tables through Ignite *SQLLINE CLI*.

```
shamini@shamini5:~/git/ignite$ sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
issuing: !connect jdbc:ignite:thin://127.0.0.1/* org.apache.ignite.IgniteJdbcThinDriver
Connecting to: Apache Ignite (version 2.6.0#20180710-sha1:669feacc)
Connected to: Apache Ignite Thin JDBC Driver (version 2.6.0#20180710-sha1:669feacc)
Autocommit status: true
Transaction isolation: TRANSACTION_REPEATABLE_READ
sqlline version 1.3.0
0: jdbc:ignite:thin://127.0.0.1/> !tables
+-----+-----+-----+-----+
| TABLE_CAT | TABLE_SCHEM | TABLE_NAME | TABLE_TYPE |
+-----+-----+-----+-----+
|          | PUBLIC    | EMP        | TABLE      |
|          | PUBLIC    | DEPT      | TABLE      |
+-----+-----+-----+-----+
0: jdbc:ignite:thin://127.0.0.1/> Select * from emp;
+-----+-----+-----+-----+
| EMPNO | ENAME   | HIREDATE | JOB       |
+-----+-----+-----+-----+
| 1     | KING    |           | PRESIDENT |
| 2     | BLAKE   |           | MANAGER   |
| 3     | CLARK   |           | MANAGER   |
| 4     | JONES   |           | MANAGER   |
+-----+-----+-----+-----+
4 rows selected (0.107 seconds)
```

Figure 6.31

Two different tables created for the entities: *Department* and *Employee*. You can use ANSI-99 SQL queries to query the tables. Note that, the *hibernate-ogm-ignite* project is in progress and only a sub-set of JP-QL queries are available. These includes:

- simple comparisons using “<”, “ $\leq$ ”, “=”, “ $\geq$ ” and “>”
- IS NULL and IS NOT NULL
- the boolean operators AND, OR, NOT
- LIKE, IN and BETWEEN
- ORDER BY

Under the hood, queries using these above SQL operators transforms into equivalent native Ignite SQL queries and executes over Ignite caches.

## Persistence

In-memory approaches can achieve blazing speed by putting the working set of data into the system memory. When all the data is kept in memory, the need to deal with issues arising from the use of traditional spinning disks disappears. That means, for instance, there is no need to maintain additional cache copies of data and manage synchronization between them. However, there is also a downside to this approach because the data is in memory only, the data will not survive if the whole cluster gets terminated. Therefore, these types of data stores are not considered persistent at all.

In most cases, you can't (should not) store the whole data set in memory for your application, most often you should store a relatively small or active subset of data to increase the performance of the application. The rest of the data should be stored somewhere in low-cost disks or tape for archiving. There are two main in-memory database storage requirements available:

- Permanent media, to store committed transactions. Thereby maintaining durability and for recovery purpose if the in-memory database needs to be reloaded into the memory.
- Permanent storage, to hold a backup copy of the entire in-memory database.

Permanent storage or media can be any distributed or local file system, SAN, NoSQL database, or even RDBMS like Postgres or Oracle. Apache Ignite provides two elegant ways to persist data:

1. Persistence in 3<sup>rd</sup> party database. Since 1.5 version, Apache Ignite provides API to connect persistence data stores such as RDBMS or NoSQL DB like Mongo DB or Cassandra for storing data. Most often, persistence in an RDBMS will be bottlenecks, and you never got a horizontal scaling in your system.

2. Ignite native persistence. From the version 2.1.0, Apache Ignite provides ACID and SQL-compliant disk stores that transparently integrate with Ignite's durable memory as an optional disk layer storing data and indexes on SSD, Flash, 3D XPoint, and other types of non-volatile storage systems.

In this section, I explore Apache Ignite's both 3<sup>rd</sup> part and native persistence. Furthermore provide a clear, understandable picture of how Apache Ignite persistence mechanism works.

## Native persistence

The Apache Ignite native persistence uses new durable memory architecture that allows storing and processing data and indexes both in-memory and on disk. Whenever the feature enables, Apache Ignite stores a superset of data on disk, and a subset of data in RAM based on its capacity. If a subset of data or an index is missing in RAM, the Durable Memory takes it from the disk as shown in figure 6.32.

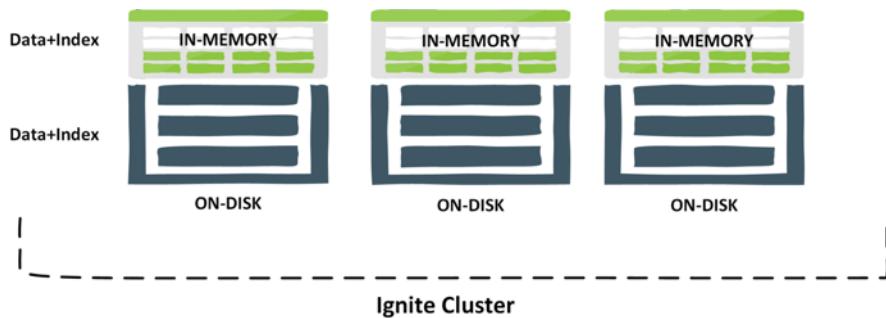


Figure 6.32

As we stated in *chapter 4*, data can also be stored in the central disk storage where all the Ignite nodes are connected. Please see native persistence section in chapter 4 for details about WAL archive and memory model.

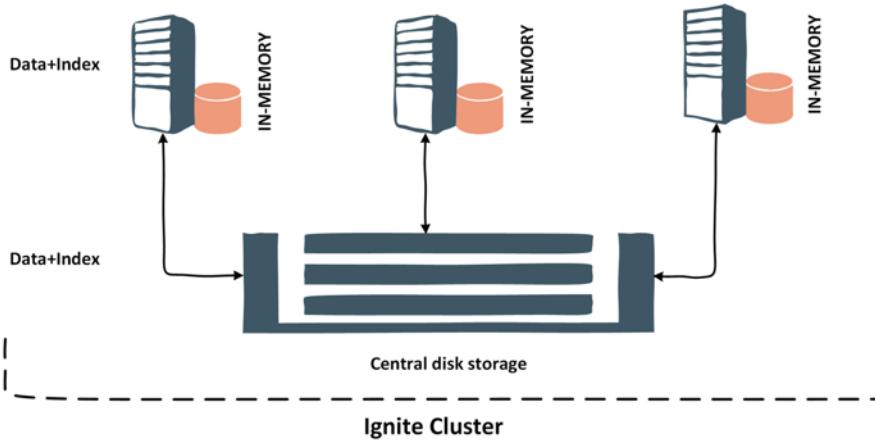


Figure 6.33

In this section, we will launch an Ignite node with native persistence enable and learn how data stored in the file system. Let's cover the prerequisites of the project in our sandbox before we start:

Name	Description
Apache Ignite	>=2.6.0
JVM	1.8
OS	*nix

**Step 1.** Download the Apache Ignite binary distribution, and unzip the distribution somewhere in your sandbox. Skip this step if you already have an installed copy of Ignite in your system.

**Step 2.** You have to pass an instance of `DataStorageConfiguration` to a cluster node configuration to enable the Ignite native persistence. Ignite `DataStorageConfiguration` allocates a single expandable data region with default settings. It will map all the caches that will be configured in an application to this data region. Persistence for *data region* can be turned on with `DataRegionConfiguration.setPersistenceEnabled(boolean)` flag. You can find a pre-configured `DataStorageConfiguration` in the `example-persistent-store.xml` file which supplied with the Ignite distribution. Let's take a look at the portion of the `example-persistent-store.xml` file.

**Listing 6.90**

---

```
<bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    <!-- Enabling Apache Ignite Persistent Store. -->
    <property name="dataStorageConfiguration">
        <bean class="org.apache.ignite.configuration.DataStorageConfiguration">
            <property name="defaultDataRegionConfiguration">
                <bean class="org.apache.ignite.configuration.DataRegionConfiguration">
                    <property name="persistenceEnabled" value="true" />
                </bean>
            </property>
        </bean>
    </property>
</bean>
</property>

<!-- Additional setting. TCP discovery or Cache configuration-->
</bean>
```

---

**Tip**

The Persistent Store's files are located under the Ignite **work** directory by default. However, you can set a path to the directory where the Ignite persistent will persist data and indexes. Use the `persistenceStorePath` property of the `DataStorageConfiguration` class to set the path.

**Step 3.** Run the following command from the `IGNITE_HOME` directory.

**Listing 6.91**

---

```
ignite.sh $IGNITE_HOME/examples/config/persistentstore/example-persistent-store.xml
```

---

It should start an Ignite node and create a new folder named `WORK` under `IGNITE_HOME` directory. Log messages on the Ignite server console should also confirm that the server is started with *persistence enable* as shown in figure 6.34.

```
[22:19:27] Ignite node started OK (id=9afea73d)
[22:19:27] >>> Ignite cluster is not active (limited functionality available). Use control.(shlbat)
to activate.
[22:19:27] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=1.0GB]
[22:19:27]   ^-- Node [id=9AFEAT3D-AF9B-4FEC-940C-41185B2D86EB, clusterState=INACTIVE]
[22:19:27] Data Regions Configured:
[22:19:27]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=true]
```

Figure 6.34

Ignite also warned that the cluster is not activated yet, and you have to activate the cluster by using `control.sh` script.

**Step 4.** Now, Ignite node is up and running with persistence enable in the INACTIVE mode. Active the cluster by using the `control.sh|bat` script supplied with the Ignite distribution. Type the following command in any separate terminal.

**Listing 6.92**

---

```
control.sh --activate
```

---

If the command succeeds, you should see the following messages in the console.

```
Control utility [ver. 2.6.0#20180710-sha1:669feacc]
2018 Copyright(C) Apache Software Foundation
```

```
User: shamim
```

---

```
-----
```

```
Cluster activated
```

At this moment, you can also use the `--state` command to check the current cluster state. The `--state` command should return a message that the cluster is activated. Please refer to *chapter 10* for more details about Ignite cluster management and monitoring.

**Step 5.** Now, create a table and populate some data. We use the *SQLLINE* tool to connect to the cluster. Run the following command to launch the *SQLLINE* tool:

**Listing 6.93**

---

```
sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
```

---

Create a table named *EMP* and insert a few rows into the table. Use the following DDL script to create the *EMP* table as follows:

**Listing 6.94**

---

```
CREATE TABLE IF NOT EXISTS EMP
(
    empno  LONG,
    ename  VARCHAR,
    job    VARCHAR,
    mgr    INTEGER,
    hiredate DATE,
    sal    LONG,
    comm   LONG,
    deptno LONG,
    CONSTRAINT pk_emp PRIMARY KEY (empno)
) WITH "template=partitioned,CACHE_NAME=EMPcache";
```

---

Insert a few rows into the EMP table as follows.

**Listing 6.95**

---

```
INSERT INTO emp
(empno,
ename,
job,
mgr,
hiredate,
sal,
comm,
deptno)
VALUES  (7839,
'KING',
'PRESIDENT',
NULL,
To_date('17-11-1981', 'dd-mm-yyyy'),
5000,
NULL,
10);
```

```
INSERT INTO emp
(empno,
ename,
job,
mgr,
hiredate,
sal,
```

```
    comm,  
    deptno)  
VALUES ( 7698,  
    'BLAKE',  
    'MANAGER',  
    7839,  
    To_date('1-5-1981', 'dd-mm-yyyy'),  
    2850,  
    NULL,  
    30);
```

```
INSERT INTO emp  
(empno,  
ename,  
job,  
mgr,  
hiredate,  
sal,  
comm,  
deptno)  
VALUES (7782,  
    'CLARK',  
    'MANAGER',  
    7839,  
    To_date('9-6-1981', 'dd-mm-yyyy'),  
    2450,  
    NULL,  
    10);
```

---

**Step 6.** Let's use the *ignitevisor* command-line tool for scanning the EMPCache. Use the `cache -scan` command in the *ignitevisor* command tool. You should receive a similar output on your console as shown below. Here are all the three elements in the cache:

```

visor> cache -scan
Time of the snapshot: 2018-12-20 20:38:04
+-----+
| # | Name(@) | Mode | Size (Heap / Off-heap) |
+-----+
| 0 | EMPcache(@c0) | PARTITIONED | min: 3 (0 / 3) |
| | | | avg: 3.00 (0.00 / 3.00) |
| | | | max: 3 (0 / 3) |
+-----+
Choose cache number ('c' to cancel) [c]: 0
Entries in cache: EMPcache
+-----+
| Key Class | Key | Value Class | Value |
+-----+
| java.lang.Long | 7698 | o.a.i.i.binary.BinaryObjectImpl | SQL_PUBLIC_EMP_0bd87eaf_b503_4b06_83b6_25be1cdb893c
[hash=-1417502761, ENAME=BLAKE, JOB=MANAGER, MGR=7839, HIREDATE=1981-05-01, SAL=2850, COMM=null, DEPTNO=30] |
| java.lang.Long | 7782 | o.a.i.i.binary.BinaryObjectImpl | SQL_PUBLIC_EMP_0bd87eaf_b503_4b06_83b6_25be1cdb893c
[hash=1453774085, ENAME=CLARK, JOB=MANAGER, MGR=7839, HIREDATE=1981-06-09, SAL=2450, COMM=null, DEPTNO=10] |
| java.lang.Long | 7839 | o.a.i.i.binary.BinaryObjectImpl | SQL_PUBLIC_EMP_0bd87eaf_b503_4b06_83b6_25be1cdb893c
[hash=1848897643, ENAME=KING, JOB=PRESIDENT, MGR=null, HIREDATE=1981-11-17, SAL=5000, COMM=null, DEPTNO=10] |

```

Figure 6.35

**Step 7.** Now, let's discover what's happening under the hood. Run the following command from the *IGNITE\_HOME/work* directory:

#### Listing 6.96

---

```
du -h.
```

---

The command should return something similar as shown below.

```

shamim:work shamim$ du -h .
8.0K ./binary_meta/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b
8.0K ./binary_meta
144K ./db/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b/cache-EMPcache
32K ./db/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b/cache-ignite-sys-cache
40K ./db/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b/cp
60K ./db/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b/metastorage
280K ./db/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b
0B ./db/wal/archive/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b
0B ./db/wal/archive
64M ./db/wal/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b
64M ./db/wal
64M ./db
92K ./log
8.0K ./marshaller
64M .

```

If you goes through the directory `db/node00-46ef2525-eef7-4cd1-9b14-b81a1eb9cd1b` (in my case), you should find individual folder for each cache. The folder with name `cache-EMPCache` should have all the cache entries (3 elements) which we have just inserted.

```
shamim:cache-EMPCache shamim$ ls -la
total 288
-rw-r--r-- 1 shamim staff 5211 22:41 cache_data.dat
-rw-r--r-- 1 shamim staff 40960 20:35 index.bin
-rw-r--r-- 1 shamim staff 32768 20:35 part-530.bin
-rw-r--r-- 1 shamim staff 32768 20:35 part-614.bin
-rw-r--r-- 1 shamim staff 32768 20:35 part-671.bin
```

The file `index.bin` is the index file of the cache entries, and each cache element gets their individual partition or page file (`*.bin`). Why did this happen? Ignite architecture is a page based architecture, and a page is the smallest unit of the data with a fixed size. Pages are stored outside the Java heap and organized in RAM. When the allocated memory is exhausted and the data are pushed to the persistence store, it happens page by page. If you take a closer look at the pages size, all of them are about 33 KB. Whenever Ignite needs to load data from the disk, it just loads the page file, and so it's very fast.

**Step 8.** Now, restart the Ignite node, and check the cache `EMPCache` with `ignitevisor` CLI. You will end up with a surprise that there is no data in the cache.

```
visor> cache -scan
(wrn) <visor> No caches found
```

Use the Ignite `SQLLINE` tool to connect to the node. Execute the following `SELECT` statement to list the employees who do not belong to department 30:

**Listing 6.97**

---

```
SELECT *
FROM emp
WHERE deptno NOT IN ( 30 );
```

---

Apache Ignite can execute SQL queries over the data that is both in memory and on disk. Whenever any read request occurs, Ignite first checks the data into the memory. If the dataset doesn't exist in memory, Ignite immediately retrieve the cache entries from the disk and loads into the memory. Also, note that all entries into the memory are in off-heap.

The main benefits of the Ignite native persistence are the data durability. However, it can also be handy to perform backups for the data recovery. Denis Magda wrote a comprehensive article<sup>126</sup> on data recovery by using Ignite native persistence. In addition, you can replicate data from one cluster to another in real-time by using Ignite native persistence. You can use any standard disk-based data replication tools to copy the changed dataset from the primary data center to the stand-in data center or another Ignite cluster.

## Persistence in 3<sup>rd</sup> party database (MongoDB)

Apache Ignite uses *read-through* and *write-through* caching strategies to persist data into 3<sup>rd</sup> part databases. Ignite provides out-of-the-box implementation to read and write database records from any RDBMS and Apache Cassandra. Apache Ignite implements the JCache interface for `CacheLoader` and `CacheWriter`, which are used for read-through and write-through to and from any underlying persistence media. Ignite provides `org.apache.ignite.cache.store.CacheStore` interface, which extends both `CacheLoader` and `CacheWriter` interface.



### Info

For other NoSQL databases like *MongoDB* or *Neo4J*, Ignite provides APIs to implement custom cache store. You can implements the `org.apache.ignite.cache.store.CacheStore` interface yourself (custom) to store cache entries to any external resources such as *local file system*, *SAN/NAS* or *Hadoop HDFS*.

Besides, Ignite `CacheStore` is fully transactional and automatically merges into the ongoing cache transaction. However, there is a tradeoff when using the write-through approach. For every single transaction, Ignite tries to save or update the data into persistence store or on-disk database, and therefore the overall duration of the cache update time might be relatively high. Here, Ignite is the same as an on-disk database.

---

<sup>126</sup><https://dzone.com/articles/apache-ignite-native-persistence-enables-data-reco>

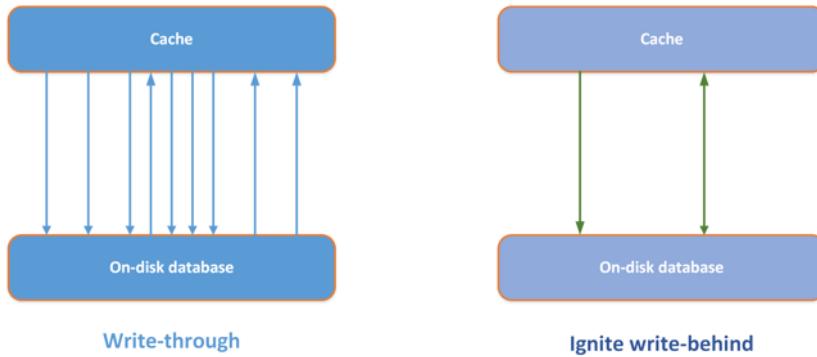


Figure 6.36

An intensive cache update rate can cause an extremely high storage load on the on-disk database. For such cases, Ignite offers a bulk update option to update asynchronously named *write-behind* as shown in figure 6.36. The main idea behind this approach is to aggregate updates and asynchronously flush it into the persistence store. Ignite write-behind implementation is very similar to the JCache entry processor which we described in *chapter four*. Ignite can flush the aggregated data with different criteria:

- **Time-based events:** the maximum time that data entry can reside in the queue.
- **Queue-size events:** the queue is flushed when its size reaches some particular point.
- **Both of them in combination.**



## Warning

With the *write-behind* approach, only the last update of an entry will be written to the underlying storage. If the cache entry with key `key1` is sequentially updated with values `value1`, `value2`, and `value3` respectively, then only a single store request for (`key1`, `value3`) pair will be propagated to the persistent storage.

In addition to *JCache* storing and loading method, Ignite `CacheStore` interface provides the ability to bulk store and load caches from the persistence store. Also, Ignite offers a cache storage convenience adapter named `CacheStoreAdapter` which implements the `CacheStore` interface and provides a default implementation for bulk operations. Abstract class `CacheStoreAdapter` has the following methods, which you have to implement for working with the persistence store:

Method names	Descriptions
load(), write(), delete()	Methods inherited from interfaces JCache.CacheLoader, and JCache.CacheWrite which allows you to put, get or delete cache entries from the persistence store. These methods are used to enable <i>read-through</i> and <i>write-through</i> behavior when working with individual cache entries.
loadAll(), writeAll(), deleteAll()	Methods inherited from the interfaces JCache.CacheLoader, and JCache.CacheWriter allows you to perform bulk operations. These methods are used to enable write-through and read-through behavior when working with multiple cache entries. These methods should be implemented for batch operation on persistence store to get better performance.
loadCache()	Method inherited from the Ignite.CacheStore interface, and used to loads all values from underlying persistence storage. It is generally used for hot-loading the cache on startup but can also be called at any point after the cache has been started.
sessionEnd()	An optional method which provides default empty implementation for ending transactions. Ignite uses this method to store session which may span through more than one cache store operation. This method is useful when working with transaction.

Class CacheConfiguration defines Ignite cache configuration, and offers a set of methods to enable write/read-through and write-behind caching.

Methods	Descriptions	default
setCacheStoreFactory()	Set cache store factory for persistence storage. Cache store factory should be implementation of the CacheStoreAdapter abstract class.	
setReadThrough(boolean)	Enables the read-through persistence approach.	False
setWriteThrough(boolean)	Enables the write-through persistence approach.	False
setWriteBehindEnabled(boolean)	Enables the write-behind persistence approach.	False
setWriteBehindFlushSize(int)	The maximum size of the write-behind cache. If cache size exceeds this value, all cached items are flushed to the cache store and write cache is cleared. If this value is 0, then flush is performed according to the flush frequency interval. Note that you cannot set both flush size and flush frequency to 0.	10240

Methods	Descriptions	default
setWriteBehindFlushFrequency(long)	The frequency with which write-behind cache is flushed to the cache store in milliseconds. This value defines the maximum time interval between object insertion/deletion from the cache and the moment when corresponding operation is applied to the cache store. If this value is 0, then flush is performed according to the flush size. Note that, you cannot set both flush size and flush frequency to 0.	5 seconds
setWriteBehindFlushThreadCount(int)	The number of threads that will perform cache flushing.	1
setWriteBehindBatchSize(int)	The maximum batch size for write-behind cache store operations.	512

Now that we have got the basics of storing data into the 3<sup>rd</sup> party database, let's install a NoSQL database and configure Ignite to store and load caches to and from the database. Our persistence store will be [MongoDB<sup>127</sup>](#), an open source prevalent NoSQL document-oriented database. The main benefit of using NoSQL or MongoDB as a persistence store is that they are very suitable for scaling out. You can store the cold or historical data directly from the presentation tier as JSON without any data conversion by using MongoDB. In this section, we are going to populate the following sceneries:

- Write-through: write cache entries into MongoDB.
- Write-behind: aggregate the updates into a single operation to update the stores.
- Read-through: read from the persistence store, when cache entries are not found in the cache.

To accomplish the above sceneries, we have to fulfill the following prerequisites:

1. Install and configure the MongoDB server.
2. Implements the Ignite CacheStoreAdapter to store and load data to and from MongoDB.

---

<sup>127</sup><https://www.mongodb.com/>

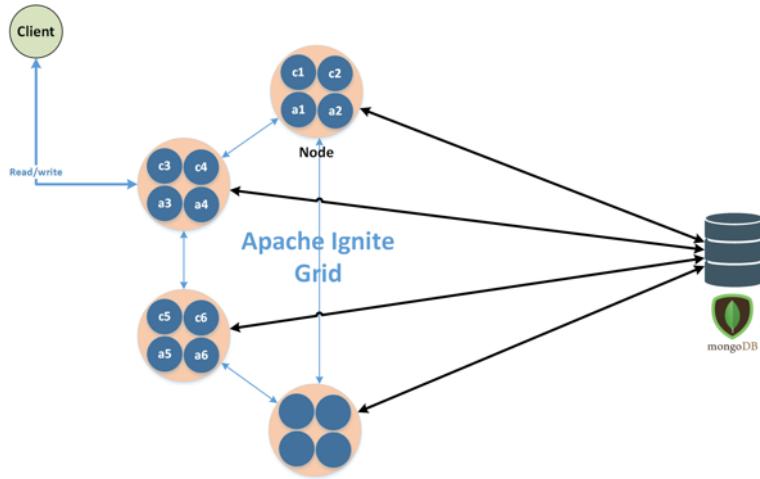


Figure 6.37

Figure 6.37 shows an overview of using MongoDB as a persistence store. A few words about MongoDB: MongoDB is an open source schemaless database. MongoDB stores all the data in BSON or binary JSON format. This gives MongoDB the flexibility to access his functionalities through Javascript, making the integration of data in certain types easier and faster. MongoDB also supports data *sharding* through the configuration of a shared cluster; it means you can scale MongoDB vertically and horizontally seamlessly. Typically, MongoDB is used for real-time analytics, where latency is low, and availability is very high.

The complete example of using MongoDB as a 3<sup>rd</sup> party persistence store can be downloaded from the [GitHub<sup>128</sup>](#) repository. There are a few different versions and distributions offers from the MongoDB. You can download MongoDB distribution from the MongoDB download page by yours choose. In my case, I will use the latest stable version 4.0.5. The full table of the prerequisites are:

N	Name	Value
1	MongoDB	4.0.5
2	Java	1.8
3	Apache Ignite	2.6
4	spring-data-mongodb library	1.10.17

We are going to install the MongoDB first, and then prepare the Maven project to accomplish the persistence of cache entries into the MongoDB. If you already have installed and configured a MongoDB database in your system, you can skip the first three steps and

<sup>128</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-6/persistence-store>

continue from step 4.

**Step 1.** Download the MongoDB distribution from [here<sup>129</sup>](#). I am going to use community edition, and downloaded the distribution for the MacOS operating system.

**Step 2.** Unzip the distribution somewhere in your file system. I use the Linux `tar` command.

**Listing 6.98**

---

```
tar -xvf mongodb-osx-ssl-x86_64-4.0.5.tgz
```

---

MongoDB stores its data in a **DATA directory**. You need to create this data directory first. Create a folder with name `/data/db` in your local file system to store the database files.

**Step 3.** To start MongoDB, use `mongod` command for Linux/Mac, or `mongod.exe` comamnd for Windows:

**Listing 6.99**

---

```
bin/mongod --dbpath PATH_TO_YOUR_DATA/DB_FOLDER
```

---

Note that the *data directory path* has been added to the command in listing 6.99. Once the *mongod* process has started, you should see a *waiting for connections* message in the console output which indicates that the *mongod* process is running successfully.

```
2018-12-21T22:18:27.083+0300 I CONTROL [initandlisten] ** WARNING: This server is bound to localhost.
2018-12-21T22:18:27.083+0300 I CONTROL [initandlisten] ** Remote systems will be unable to connect to
this server.
2018-12-21T22:18:27.083+0300 I CONTROL [initandlisten] ** Start the server with --bind_ip <address> t
o specify which IP
2018-12-21T22:18:27.083+0300 I CONTROL [initandlisten] ** addresses it should serve responses from, o
r with --bind_ip_all to
2018-12-21T22:18:27.083+0300 I CONTROL [initandlisten] ** bind to all interfaces. If this behavior is
desired, start the
2018-12-21T22:18:27.083+0300 I CONTROL [initandlisten] ** server with --bind_ip 127.0.0.1 to disable
this warning.
2018-12-21T22:18:27.083+0300 I CONTROL [initandlisten]
2018-12-21T22:18:27.291+0300 I FTDC [initandlisten] Initializing full-time diagnostic data capture with dir
ecitory '/Users/shamim/Development/bigdata/mongodb-4.0.5/data/db/diagnostic.data'
2018-12-21T22:18:27.293+0300 I NETWORK [initandlisten] waiting for connections on port 27017
```

**Figure 6.38**

As shown in figure 6.38, MongoDB server has been started and running on port 27017. Leave this terminal window open in the background.

**Step 4.** Create a Maven project or modify the downloaded project from the GitHub, and add all the following dependencies in the `pom.xml` file as shown below.

---

<sup>129</sup><https://www.mongodb.com/download-center?jmp=nav#community>

**Listing 6.100**

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>1.10.17.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>1.10.11.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>4.3.2.RELEASE</version>
</dependency>
```

---

We add *spring-data-mongodb* and *spring-data-jpa* dependency so that we are going to use the *Spring data MongoDB* framework to integrate with the MongoDB database. The key functional areas of Spring Data MongoDB are a POJO centric model for interacting with a MongoDB database, and easily writing a *Repository* style data access layer. Please, see the source *pom.xml* file for all dependencies.

**Step 5.** We are going to use a basic domain model for storing data into the MongoDB database. The domain model contains only one entity named `POSTS` (most often used by the twitter). The entity comprises a unique *ID of the post*, *title*, *author* of the post, and the post *creation date*. In the JSON format, the data model will appear as follows:

**Listing 6.101**

---

```
{  
    _id:"_1",  
    _class:"com.blu.imdg.nosql.model.MongoPost",  
    title:"title-1",  
    description:"description-1",  
    creationDate:"Sat Nov 05 00:00:00 MSK 2018",  
    author:"author-1"  
}
```

---

Let's create a Java class with name `MongoPost` into the `com.blu.imdg.nosql.model` package corresponding to the data model.

**Listing 6.102**

---

```
public class MongoPost {  
    @Id  
    private String id;  
    private String title;  
    private String description;  
    private LocalDate creationDate;  
    private String author;  
    // setter and getter methods are omitted  
}
```

---

**Step 6.** As stated earlier, we use the `spring-data-MongoDB` library from the Spring framework for working with the MongoDB database. The given library is a façade and makes more comfortable to work with MongoDB. We also made a spring context file to configure the MongoDB factory. Create an XML file named `mongo-context.xml` into the `main/resources` directory into the Maven project, and input the following contents.

**Listing 6.103**

---

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mongo="http://www.springframework.org/schema/data/mongo"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/data/mongo http://www.springframewor
ringframework.org/schema/data/mongo/spring-mongo.xsd
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans\
/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframewor
k.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <mongo:mongo id="mongo" host="localhost" port="27017"/>

    <mongo:db-factory id="mongoDbFactory" dbname="test" mongo-ref="mongo" />

    <mongo:repositories base-package="com.blu.imdg.nosql" mongo-template-ref="mongoTempla
te"/>
        <bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
            <constructor-arg ref="mongoDbFactory" />
        </bean>
        <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostPr
ocessor"/>

</beans>
```

---

Let's take a closer look at the above spring context file. The above XML file instructs the Spring to do following:

1. Specified the MongoDB host and port. You have to change the configuration if you have another hostname or port.
2. Set the MongoDB factory database name to test.
3. Convert the MongoPost DTO to mongo Bson.
4. Set the MongoDB template to the factory.

**Step 7.** Now that we have the DTO created and Spring application context file in place, it's time to implement the Ignite CacheStore. Create a new Java class named `MongoDBStore`, which extends the Ignite `CacheStoreAdapter` and implements the `LifecycleAware` interface.

**Listing 6.104**

---

```
public class MongoDBStore extends CacheStoreAdapter<String, MongoPost> implements LifecycleAware {

    @Autowired
    private PostRepository postRepository;
    @Autowired
    private MongoOperations mongoOperations;

    private static Logger logger = LoggerFactory.getLogger(MongoDBStore.class);

    @Override public MongoPost load(String key) throws CacheLoaderException {

        logger.info(String.valueOf(postRepository));
        return postRepository.findOne(key);
    }

    @Override public void write(Cache.Entry<? extends String, ? extends MongoPost> entry) throws CacheWriterException {
        MongoPost post = entry.getValue();
        logger.info(String.valueOf(postRepository));
        postRepository.save(post);
    }

    @Override public void delete(Object key) throws CacheWriterException {
        logger.info(String.valueOf(postRepository));
        postRepository.delete((String)key);
    }

    @Override public void start() throws IgniteException {
        ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("mongo-context.xml");
        postRepository = context.getBean(PostRepository.class);
        logger.info(String.valueOf(postRepository));
        mongoOperations = context.getBean(MongoOperations.class);
        if (!mongoOperations.collectionExists(MongoPost.class))
            mongoOperations.createCollection(MongoPost.class);
    }

    @Override public void stop() throws IgniteException { }
}
```

---

Here, we implements three CacheStore methods: *load()*, *write()*, and *delete()*.

- *load()* method looks for the documents in MongoDB server by the given key and returns the `MongoPost` instance.
- *Write()* method saves the single cache entry into the MongoDB test database.
- *Delete()* method deletes the BSON documents from the MongoDB by the given key.

**Step 8.** For writing and reading data to and from the MongoDB database, we programmatically configure the Ignite `cacheConfiguration`. I create a Java class with name `CacheStoreSample` into the `com.blu.imdg` package. Let's take a closer look at the cache configuration for the MongoDB again as shown in listing 6.105.

**Listing 6.105**

---

```
IgniteConfiguration cfg = new IgniteConfiguration();

CacheConfiguration configuration = new CacheConfiguration();
configuration.setName("mongoDynamicCache");
configuration.setAtomicityMode(CacheAtomicityMode.TRANSACTIONAL);

configuration.setCacheStoreFactory(FactoryBuilder.factoryOf(MongoDBStore.class));
configuration.setReadThrough(true);
configuration.setWriteThrough(true);

configuration.setWriteBehindEnabled(true);

log("Start. PersistenceStore example.");
cfg.setCacheConfiguration(configuration);
```

---

First, we create the Ignite `CacheConfiguration` instance, set the cache name to `mongoDynamicCache` and the atomicity mode to `transactional`. Then we set our `MongoDBStore` as the caching factory and enable the *Read-through* and *Write-Through* features. Note that we also enable the *Write-behind* feature.

Next, we put ten entries into the `mongoDynamicCache` as shown in the following pseudo-code:

**Listing 6.106**

---

```
try (Ignite ignite = Ignition.start(cfg)) {
    int count = 10;
    try (IgniteCache<String, MongoPost> igniteCache = ignite.getOrCreateCache(configuration)) {
        try (Transaction tx = ignite.transactions().txStart(PESSIMISTIC, REPEATABLE_READ)) {
            for (int i = 1; i <= count; i++) {
                igniteCache.put("_" + i, new MongoPost("_" + i, "title-" + i, "descriptio" +
                    n-" + i, LocalDate.now().plus(i, ChronoUnit.DAYS), "author-" + i));
            }
            tx.commit();
        }
    }
    log("PersistenceStore example finished.");
}

Thread.sleep(Integer.MAX_VALUE);
}
```

---

Here, we use the Ignite configuration created earlier and start a transaction before executing the put operation. We construct ten new `MongoPosts` documents and use the Ignite cache `PUT()` method to store the entries into the cache. We also commit the transaction after the operation.

**Step 9.** Now, build the Maven project and execute the application to store entities into MongoDB from the Ignite cache. Run the following command from the project home directory.

**Listing 6.107**

---

```
mvn clean install
```

---

Run the application as follows:

**Listing 6.108**

---

```
java -jar ./target/cache-store-runnable.jar mongodb
```

---

The application produces a lot of logs into the console. Here's an example of the output:

```
22:59:23.807 [flusher-0-#43] INFO c.b.i.n.MongoDBStore - org.springframework.data.mongodb.repository.support.SimpleMongoRepository@45735d2a
22:59:23.807 [flusher-0-#43] DEBUG o.s.d.m.c.MongoTemplate - Saving DBObject containing fields: [_class, _id, title, description, creationDate, author]
22:59:23.807 [flusher-0-#43] DEBUG o.s.d.m.c.MongoDbUtils - Getting Mongo Database name=[\test]
22:59:23.808 [flusher-0-#43] INFO c.b.i.n.MongoDBStore - org.springframework.data.mongodb.repository.support.SimpleMongoRepository@45735d2a
22:59:23.809 [flusher-0-#43] DEBUG o.s.d.m.c.MongoTemplate - Saving DBObject containing fields: [_class, _id, title, description, creationDate, author]
22:59:23.809 [flusher-0-#43] DEBUG o.s.d.m.c.MongoDbUtils - Getting Mongo Database name=[\test]
22:59:23.809 [flusher-0-#43] INFO c.b.i.n.MongoDBStore - org.springframework.data.mongodb.repository.support.SimpleMongoRepository@45735d2a
22:59:23.809 [flusher-0-#43] DEBUG o.s.d.m.c.MongoTemplate - Saving DBObject containing fields: [_class, _id, title, description, creationDate, author]
22:59:23.809 [flusher-0-#43] DEBUG o.s.d.m.c.MongoDbUtils - Getting Mongo Database name=[\test]
```

**Step 10.** Next, connect to the MongoDB instance and execute some query. You can use your terminal or command prompt to connect and run commands directly from the *MongoDB Shell*. The mongo shell is an interactive JavaScript interface to MongoDB, and it is included in the MongoDB distribution. You can use the mongo shell to query and update data, as well as perform administrative functions. The MongoDB Shell is located in the `bin` directory as the other binaries. So to run it, open a new terminal and enter the `mongo` command.

/bin/mongo

Once you have connected to the MongoDB instance, you can run queries. First switch to the database `test`.

use test

This results in the following message:

switched to db test

Execute the following query:

```
db.mongoPost.find();
```

Which should result in output like this:

```
{ "_id" : "_1", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-1", "de\\
scription" : "description-1", "creationDate" : ISODate("2018-12-21T21:00:00Z"), "author" \\
: "author-1" }
{ "_id" : "_2", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-2", "de\\
scription" : "description-2", "creationDate" : ISODate("2018-12-22T21:00:00Z"), "author" \\
: "author-2" }
{ "_id" : "_3", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-3", "de\\
scription" : "description-3", "creationDate" : ISODate("2018-12-23T21:00:00Z"), "author" \\
: "author-3" }
{ "_id" : "_4", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-4", "de\\
scription" : "description-4", "creationDate" : ISODate("2018-12-24T21:00:00Z"), "author" \\
: "author-4" }
{ "_id" : "_5", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-5", "de\\
scription" : "description-5", "creationDate" : ISODate("2018-12-25T21:00:00Z"), "author" \\
: "author-5" }
{ "_id" : "_6", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-6", "de\\
scription" : "description-6", "creationDate" : ISODate("2018-12-26T21:00:00Z"), "author" \\
: "author-6" }
{ "_id" : "_7", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-7", "de\\
scription" : "description-7", "creationDate" : ISODate("2018-12-27T21:00:00Z"), "author" \\
: "author-7" }
{ "_id" : "_8", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-8", "de\\
scription" : "description-8", "creationDate" : ISODate("2018-12-28T21:00:00Z"), "author" \\
: "author-8" }
{ "_id" : "_9", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-9", "de\\
scription" : "description-9", "creationDate" : ISODate("2018-12-29T21:00:00Z"), "author" \\
: "author-9" }
{ "_id" : "_10", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-10", "\\
description" : "description-10", "creationDate" : ISODate("2018-12-30T21:00:00Z"), "autho\\
r" : "author-10" }
```

It returns all ten documents stored in the MongoDB test database. You can filter the results down by providing only the criteria that you're interested in. For example, run the following command, if we're only interested in author-10 from the above collection:

```
db.mongoPost.find({ author : "author-10" })
```

Result:

```
{ "_id" : "10", "_class" : "com.blu.imdg.nosql.model.MongoPost", "title" : "title-10", "\n  description" : "description-10", "creationDate" : ISODate("2018-12-31T21:00:00Z"), "author"\n  r" : "author-10" }
```



## Tip

You can use the MongoDB `pretty()` method to format the results so that they're a bit easier to read. Just append `pretty()` method to the end of the above `find()` method, like this: `db.mongoPost.find({ author : "author-10" }).pretty()`.

**Step 11.** At this moment, the write-through feature is working. Let's check the *read-through* feature of the Ignite persistence store. To do this, first, we have to clean the cache and then use the Ignite cache GET API to retrieve an entry. We will use the `ignitevisor` command-line tool to clear the cache. Launch the `ignitevisor` CLI tool and run the following command:

```
cache -clear -c=mongoDynamicCache
```

The above command will produce the following output:

```
Cleared cache with name: mongoDynamicCache
+=====+
| Node ID8(@) | Cache Size Before | Cache Size After |
+=====+
| D11A070D(@n0) | 10          | 0          |
+-----+
```

Now `mongoDynamicCache` cache is empty. Let's try to retrieve a cache entry from the Ignite. Run the Java class `com.blu.imdg.ReadCacheEntries` from the project root directory as follows:

**Listing 6.108**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.ReadCacheEntries
```

---

`ReadCacheEntries` Java application connects to the Ignite server and executes the Ignite cache GET method for retrieving a cache entry with KEY `_9`. You can see the results of the execution as follows:

```
23:35:53.206 INFO MongoPost{id='_9', title='title-9', description='description-9', creati  
onDate=2018-12-30, author='author-9'}
```

Now apply the `cache -scan` visor command to display the cache entries.

```
visor> cache -scan
+=====+
| # | Name(@) | Mode | Size (Heap / Off-heap) |
+=====+
| 0 | mongoDynamicCache(@c0) | PARTITIONED | min: 0 (0 / 0) |
| | | avg: 0.50 (0.00 / 0.50) |
| | | max: 1 (0 / 1) |
+-----+
```

As you can see, the entry is obtained from the MongoDB and stored into the cache. Next, you can modify or extends the project according to your use case for storing cache entries into the MongoDB database and read through them.



## Tip

IgniteCache interface also provides `localPeek()` method that peeks at an in-memory cached value using default optional peek mode. Unlike the `GET` method, `localPeek()` method will not load value from any persistent store or from a remote node.

It is worth mentioning that, Ignite out-of-the-box provides 3<sup>rd</sup> party persistence into RDBMS. However, there is one drawback of using RDBMS as a persistence store: RDBMS is hard to scale out. On the other hand, using RDBMS also gives some advantages: you can use any existing BI tools such as Oracle BI, Tableau for analyzing the data easily. In the GitHub repository of the book you will find a complete example of using *PostgreSQL* as a persistence store for Ignite database.

It's also possible to use 3<sup>rd</sup> party persistence together with Ignite *native persistence* for a single cluster deployment starting with Apache Ignite 2.4. Apache Ignite does best effort to ensure consistency between Ignite and 3<sup>rd</sup> party storage when Ignite native persistence is enabled. Apache Ignite documentation briefly describe the entire process of enabling native persistence with 3<sup>rd</sup> party database, so we have decided not to cover those topics in this book. In the next final section of this chapter, we explore Ignite transaction ability and runs a few examples to see how it works.

## Transaction

So far in this chapter, we have learned the Apache Ignite different SQL features and API's interacting with it. Unlike other NoSQL databases, Apache Ignite provides ACID-compliant semantic when working with caches. You can create a transaction when working with one cache or across multiple caches. Caches with different cache modes, like **PARTITIONED** or **REPLICATED** can also participate in the same transaction. In 2.7 version, Ignite community also added experimental support for multi-version concurrency control with snapshot isolation available for both cache API and SQL.

Let's take a look at an example from a real-world scenario to explain what a transaction is. A classic example of the transaction is transferring money from one bank account to another. To do that, first, you have to withdraw an amount from one account and then deposit the money to the destination account. The operation either has to completed fully or not at all. If something goes wrong in halfway, the money will be lost. Another typical example of transactions is purchasing goods (books etc.) online. This seems like an everyday scenario but can be a relatively complex series of events. Whenever you are buying books online, you have to put the book in the shopping cart first; then you must fill the shipping and credit/debit card details before you confirm the purchase. If your credit/debit card is declined, the purchase will not be approved. The result of this operation is either of two states: either the purchase was confirmed, or the purchase was declined, leaving your bank card balance unchanged. Same with the money transfer scenario, transaction guarantees that there will NOT be a situation where money is withdrawn from one account, but not deposit to another.

Historically, Ignite provides two different modes for cache operations:

- transactional and
- atomic.

The smallest set of operations (withdraw an amount from one account or fill the shipping details) we described above is atomic, guaranteed to be either entirely fully or not at all. The series of events (a group of operations) to fulfill the entire process described before is the transaction. In a transactional term, all the activities either commit or rollback; either they all are completed, or they all fail – it's all or nothing. Let's make a comparison between Ignite atomic and transactional modes:

Atomic	Transactional
Supports atomic-only cache behavior doesn't provides distributed transaction and distributed locking.	Supports fully ACID-compliant transactions.
Supports multiple atomic operations, one at a time.	Support multiple cache operations in a transaction.
Bulk operations such as putAll(..), removeAll(..) and transformAll(..) methods can partially fail. Extremly fast, because explicit locking is not needed.	Supports bulk operations in a transaction.
Ignite default atomicity mode.	Doesn't allow high-performance and throughput because of distributed locking. Should be explicitly enabled.



## Tip

If you are not familiar with transactions, the [Wikipedia<sup>130</sup>](#) page is a good place to start learning about it.

The above atomicity modes are defined in Ignite `CacheAtomicityMode` ENUM. Cache atomicity mode controls whatever cache should maintain fully transactional semantics or more light-weight atomic behavior. Cache atomicity mode may be set via `CacheConfiguration.getAtomicityMode` configuration property.

An example of atomicity mode configuration is shown below in listing 6.109:

Listing 6.109

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="clientLoyalties"/>
            <!-- Set atomicity mode, can be ATOMIC or TRANSACTIONAL.
                 ATOMIC is default. -->
            <property name="atomicityMode" value="TRANSACTIONAL"/>
        </bean>
    </property>
</bean>
```

To help explain the difference between the two transaction modes in Ignite, let's consider the following pseudo-code:

<sup>130</sup><http://en.wikipedia.org/wiki/ACID>

Atmoic mode	Transactional mode
<pre>try (Ignite ignite = Ignition.start(cfg)) {     BankAccount bankAccount =         cacheAccount.get(accountNumber);     // Deposit into account.     bankAccount.deposit(amount);     // Store updated account in cache.     cacheAccount.put(accountNumber,         bankAccount);     cacheAudit.put(accountNumber,         bankAccount); }</pre> <p>In atomic mode, the two above cache operations can partially fail.</p>	<pre>try (Transaction tx =     Ignition.ignite().transactions().txStart()) {     BankAccount bankAccount =         cacheAccount.get(accountNumber);     // Deposit into account.     bankAccount.deposit(amount);     // Store updated account in cache.     cacheAccount.put(accountNumber,         bankAccount);     cacheAudit.put(accountNumber,         bankAccount);     tx.commit(); }</pre> <p>In transactional mode, the two above cache operations will either commit or rollback.</p>

Figure 6.39

Recently, Ignite represents another atomicity mode named *TRANSACTIONAL\_SNAPSHOT* which can be configured via the atomicityMode property of the *CacheConfiguration*. Note that, the *TRANSACTIONAL* and *TRANSACTIONAL\_SNAPSHOT* modes enable fully ACID-compliant transactions. The *TRANSACTIONAL* mode only supports key-value transactions executed via Java API. On the other hand, the *TRANSACTIONAL\_SNAPSHOT* mode supports both *key-value* transactions and *SQL* transactions, and enables multi-version concurrency control (MVCC) for both types of transactions.



## Warning

The multi-version concurrency control (MVCC) feature (in version 2.7) is not production ready yet, and data consistency is not guaranteed in case of node failures.

## Ignite transactions

As stated earlier, `IgniteTransactions` facade provides an ACID compliant semantic whenever working with caches. An example of a transaction in Ignite is shown below:

**Listing 6.110**

---

```
Transaction tx = Ignition.ignite().transactions().txStart();
try{
    Account acct = cache.get(acctId);

    // Current balance.
    double balance = acct.getBalance();

    // Deposit $100 into account.
    acct.setBalance(balance + 100);

    // Store updated account in cache.
    cache.put(acctId, acct);

    tx.commit();
}
```

---

Ignite `IgniteTransactions` are *autoclosable*, so they will automatically rollback if any exceptions occur. However, you always explicitly rollback any transaction. Here is an example of transaction rollback.

**Listing 6.111**

---

```
Transaction tx = Ignition.ignite().transactions().txStart();
try{
    Account acct = cache.get(acctId);
    // Current balance.
    double balance = acct.getBalance();
    // Deposit $100 into account.
    acct.setBalance(balance + 100);
    // Store updated account in cache.
    cache.put(acctId, acct);
    tx.commit();
} catch(){
    tx.rollback();
}
```

---

Ignite `IgniteTransactions` interface provides three different overloaded methods to start transactions:

- `txStart()`. Starts transaction with default isolation, concurrency, timeout, and invalidation policy.
- `txStart(TransactionConcurrency concurrency, TransactionIsolation isolation)`. Starts new transaction with the specified concurrency and isolation.
- `txStart(TransactionConcurrency concurrency, TransactionIsolation isolation, long timeout, int txSize)`. Starts transaction with specified isolation, concurrency, timeout, invalidation flag, and a number of participating entries.

There are two transaction concurrencies available in Ignite: *PESSIMISTIC* or *OPTIMISTIC*.



## Info

Enable `TRANSACTIONAL` or `TRANSACTIONAL_SNAPSHOT` mode only if you require ACID-compliant operation.

We will have a detailed look at the transaction concurrency very soon. For now, let's start with the Ignite transaction protocols.

## Transaction commit protocols

Ignite provides two types of commit protocols for utilizing transactions: *One-phase commit* (1p), *Two-phase commit* (2p). Ignite automatically decides when to use 1-phase commit or 2-phase commit protocol. At first glance, it may appear complicated to understand. The decision mainly depends on Ignite cache mode configuration: Partitioned or Replicated and data co-allocation. To help explain the commit protocols, let's imagine the following conditions:

1. We have three nodes Ignite cluster with 150 entries in the cache.
2. Cache mode will be either Replicated or Partitioned.

**1-phase commit.** 1-phase commit or 1p commit usually used in single node database and doesn't span through multiple systems. A high-level view of the 1p commit will appear as follows:

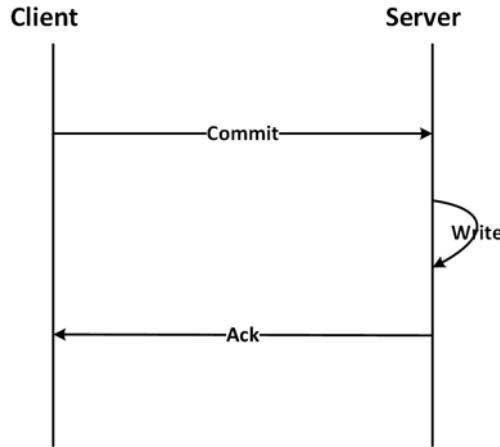


Figure 6.40

A client sends requests for commit and waiting for the acknowledgment from the server. The server on his side, either commit the transaction or rollback the data. Things are getting complicated when it comes to the distributed database like Ignite. For using 1p commit protocol in Ignite, we have to fulfill the following conditions:

1. Backup copy must be **0** in partition mode or a single Ignite server with the replicated mode.
2. Cache entries participate in a transaction must reside in the **same partition**.

To satisfy the second condition we can use an *affinity key* to co-allocate cache entries. Imagine that we have client profiles and transaction information which are stored in the different cache. So, we can use the client ID as an affinity key to bind the two different datasets. With the affinity key, Client with his transactions information will reside on the same node.

**2-phase commit.** Typically, a 2p commit is initialized whenever you have more than **1** backup copy of your primary data in the Ignite cluster. In 2p commit, there is one more phase named *prepare* appeared in the transaction process — the 2p commit protocol phases is shown in figure 6.41.

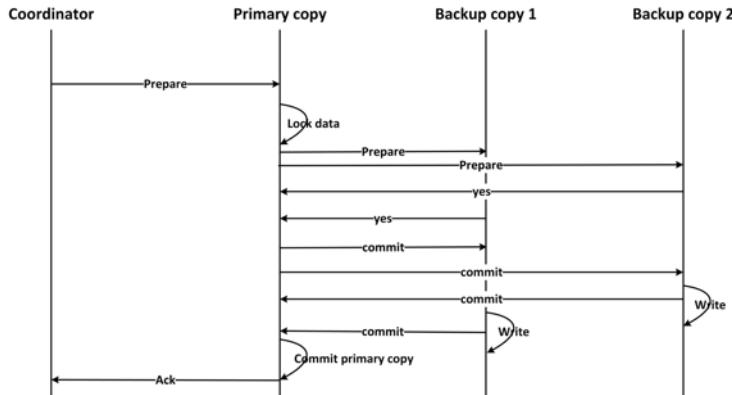


Figure 6.41

Whenever a user initiates a commit request to Ignite node, the coordinator node sends a *prepare* message to the primary nodes of the data. Primary node after acquiring the proper locks, synchronously send the prepare message to all the nodes holding the backup copy of the data. Every backup copy nodes reply with the messages *yes* on prepare request. Once every *yes* vote collected from the backup nodes, a commit message sent, and the transaction gets committed.

## Node types (NearNode/Remote/DHT)

If we consider the application from the point of view of the transaction, it is necessary to introduce another important aspect: **NearNode**. In the 2p commit algorithm, each transaction must have a node that will act as coordinator. According to Ignite terminology, its called NearNode. Regarding Ignite, this is the node where the transaction was initiated and called by `tx.start`. This node tracks the state of the transaction, the key's affected by the transaction, the nodes involved in the transaction, and other attributes of the transaction context as illustrated in figure 6.42.

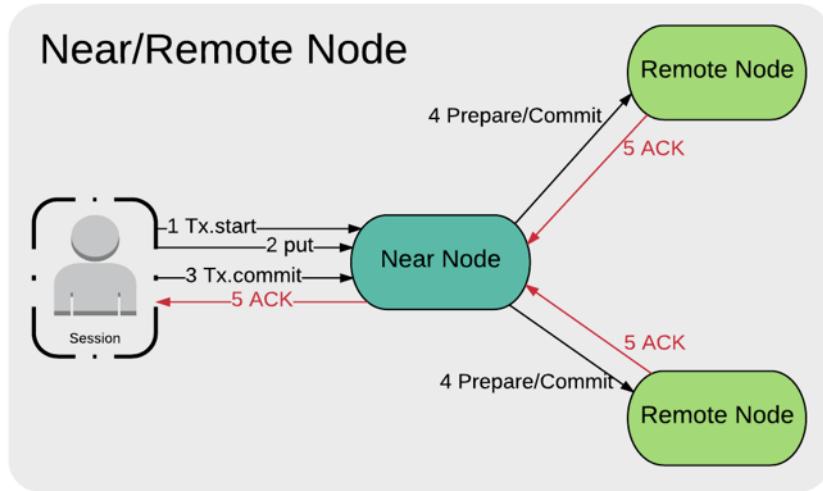


Figure 6.42

Most often, a user application connects to an Ignite cluster through an Ignite client node. In this case, the Ignite client node will be the *NearNode*. Alternatively, a user can use Ignite Thin client to connect to the cluster. In such cases, the connected server node will be the *NearNode*. The division of responsibility between the user application (Ignite client node/thin client), and the Near Node is as follows:

1. The user application (Ignite client node/thin client) calls the methods:
  - txStart()
  - cache.put()
  - cache.get()
  - tx.commit()
2. The Near Node manages all other operations:
  - Initiating a transaction.
  - Tracking the state of a transaction.
  - Sending prepare and commit messages.
  - Orchestrating the overall transaction process.

In contrast to Near node, other participants (Ignite server nodes) involved in a transaction are called *Remote node*. Physically, each remote node maintains a Distributed Hash Table (DHT) (see chapter 4 for more details) for partitions it owns. The DHT helps to look up partition owners (primary and backups) efficiently from any cluster node including the transaction coordinator.

## Concurrency Modes and Isolation Levels

One of the fundamental tasks of the database system is to provide the ability to handle multiple users accessing data simultaneously. When many users attempt to modify data in a single database at the same time, a system of controls must be implemented so that modifications made by one user do not adversely affect those of another user. This is called *concurrency control*. Concurrency control theory has two classifications for the methods of instituting concurrency control: *Pessimistic concurrency control* and *Optimistic concurrency control*.

Ignite supports OPTIMISTIC and PESSIMISTIC concurrency modes for transactions in *TRANSACTIONAL* atomicity mode like most other database systems. Concurrency mode determines locks, which can be acquired at the beginning of a transaction (pessimistic locking) or the end of a transaction (optimistic locking) before work is committed. Exactly locking prevents concurrent access to an object.

Among these two concurrency modes, Isolation determines how transaction integrity is visible to other users or transactions. Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation levels are determined by the following phenomena<sup>131</sup>:

- *Dirty Read*. A Dirty read is a situation when a transaction reads a data that has not yet been committed by the other transaction.
- *Non Repeatable read*. Non Repeatable read occurs when a transaction reads the same data set twice, and get a different value each time.
- *Phantom Read*. Phantom Read occurs when two same queries are executed, but the rows retrieved by the two are different.

Based on these above phenomena, Ignite supports *READ\_COMMITTED*, *REPEATABLE\_READ*, and *SERIALIZABLE* isolation levels.

**Pessimistic concurrency control.** A system of locks prevents users from modifying data in such a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control. In PESSIMISTIC control, locks are acquired during the first read or write access (depending on the isolation level) and held by the transaction until it is committed or rolled back. In this mode, locks are acquired on primary nodes first and then promoted to backup nodes during the prepare stage. The following isolation levels can be configured with PESSIMISTIC concurrency mode:

---

<sup>131</sup>[https://en.wikipedia.org/wiki/Isolation\\_\(database\\_systems\)](https://en.wikipedia.org/wiki/Isolation_(database_systems))

- READ\_COMMITTED. Data is read without a lock, and is never cached in the transaction itself. The data may be read from a backup node if this is allowed in the cache configuration. In this isolation, you can have the so-called Non-Repeatable Reads because, a concurrent transaction can change the data when you are reading the data twice in your transaction. The lock is only acquired at the time of first write access (this includes EntryProcessor invocation). This means that an entry that has been read during the transaction may have a different value by the time the transaction is committed. No exception will be thrown in this case.
- REPEATABLE\_READ. Entry lock is acquired, and data is fetched from the primary node on the first read or write access and stored in the local transactional map. All consecutive access to the same data is local and will return the last read or updated transaction value. This means no other concurrent transactions can make changes to the locked data, and you are getting Repeatable Reads for your transaction.
- SERIALIZABLE. In the PESSIMISTIC mode, this isolation level works the same way as REPEATABLE\_READ.

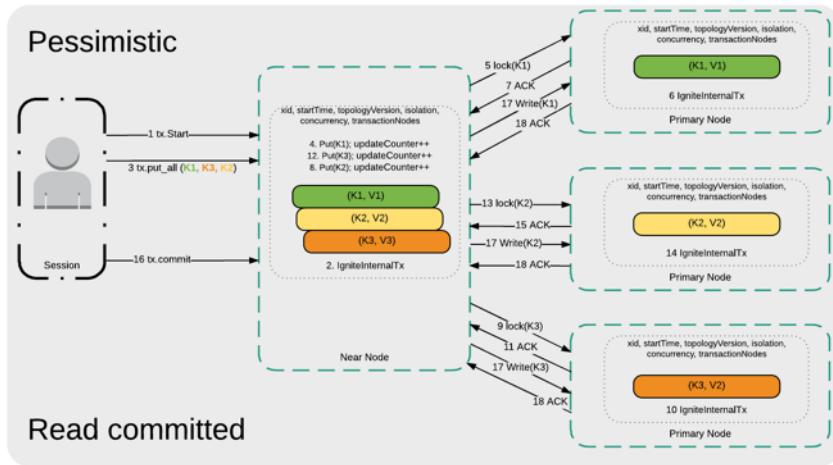


Figure 6.43

In figure 6.43 a message flow for READ\_COMMITTED isolation level is described. Note that, figure 6.35 obtained from the Ignite documentation because we didn't want to change the meaning and the contexts. Also consider that, in PESSIMISTIC mode, the order of locking is essential. Moreover, Ignite will acquire locks sequentially and precisely in the order provided by a user.



## Info

Most of the information about transaction obtained from the Ignite official [documentation](#)<sup>132</sup> and [Wiki pages](#)<sup>133</sup>. The information changes over time whenever a new Ignite version is released, so, for up-to-date information, please refer to the Ignite documentation.

**OPTIMISTIC concurrency control.** In optimistic concurrency control, *users* (In Ignite context it will be read or write request) do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called optimistic control. In *OPTIMISTIC* control, entry locks are acquired on primary nodes during the prepare step, then promoted to backup nodes and released once the transaction is committed. The locks are never acquired if the transaction is rolled back by the user and no commit attempt was made. This is how optimistic concurrency control works in Ignite as shown in figure 6.44.

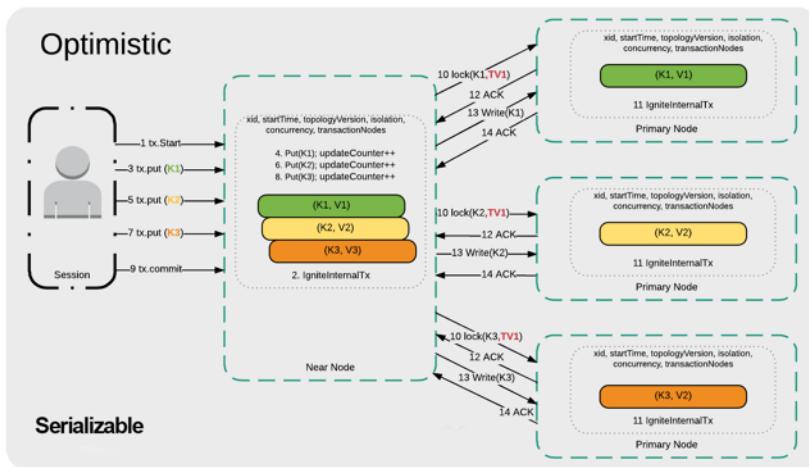


Figure 6.44

The following isolation levels can be configured with OPTIMISTIC concurrency mode:

- **READ\_COMMITTED.** Changes that should be applied to the cache are collected on the originating node and applied upon the transaction commit. Transaction data is

<sup>132</sup><https://apacheignite.readme.io/docs/concurrency-modes-and-isolation-levels>

<sup>133</sup><https://cwiki.apache.org/confluence/display/IGNITE/Ignite+Key+Value+Transactions+Architecture>

read without a lock and is never cached in the transaction. The data may be read from a backup node if this is allowed in the cache configuration. In this isolation you can have so-called Non-RePEATABLE Reads because a concurrent transaction can change the data when you are reading the data twice in your transaction. This mode combination does not check if the entry value has been modified since the first read or write access and never raises an optimistic exception.

- REPEATABLE\_READ. Transactions at this isolation level work similar to OPTIMISTIC READ\_COMMITTED transactions with only one difference, read values are cached on the originating node and all subsequent reads are guaranteed to be local. This mode combination does not check if the entry value has been modified since the first read or write access and never raises an optimistic exception.
- SERIALIZABLE. Stores an entry version upon first read access. Ignite will fail a transaction at the commit stage if the Ignite engine detects that at least one of the entries used as part of the initiated transaction has been modified. This is achieved by internally checking the version of an entry remembered in a transaction to the one actually in the grid at the time of commit. In short, this means that if Ignite detects that there is a conflict at the commit stage of a transaction, we fail such a transaction throwing TransactionOptimisticException & rolling back any changes made. The user should handle this exception and retry the transaction.

In contrast to pessimistic concurrency, optimistic concurrency delays lock acquisition. It may be better suited to applications where there is less contention of writes.

## MVCC

Apache Ignite introduced a new concurrency control named *Multiversion Concurrency Control (MVCC)* in version 2.7. MVCC is a method of controlling the consistency of data accessed by multiple users concurrently. MVCC rely on the snapshot and implements the snapshot isolation<sup>134</sup> guarantee which ensures that each transaction always sees a consistent snapshot of data.

Each transaction obtains a consistent snapshot of data when it starts and can only view and modify data in this snapshot. When the transaction updates an entry, Ignite verifies that the entry hasn't been updated by other transactions and creates a new version of the entry. The new version will become visible to other transactions only when if this transaction commits successfully. If the entry has been updated, the current transaction fails with an exception.

---

<sup>134</sup>[https://en.wikipedia.org/wiki/Snapshot\\_isolation](https://en.wikipedia.org/wiki/Snapshot_isolation)

The snapshots are not physical but logical snapshots that are generated by the MVCC-coordinator (a cluster node that coordinates transactional activity in the cluster). The coordinator keeps track of all active transactions and gets notified when each transaction ends. All operations with an MVCC-enabled cache request a snapshot of data from the coordinator.

**Enabling MVCC.** To enable MVCC for a cache, use the `TRANSACTIONAL_SNAPSHOT` atomicity mode in the cache configuration. If you create a table with the `CREATE TABLE` command, specify the atomicity mode as a parameter in the `WITH` part of the command:

**Listing 6.112**

---

```
CREATE TABLE EMP WITH "ATOMICITY=TRANSACTIONAL_SNAPSHOT"
```

---

**Concurrent Updates.** If an entry is read and then updated within a single transaction, there is a possibility that another transaction may hit in between the two operations and update that entry first. In this case, an exception is thrown when the first transaction attempts to update the entry, and the transaction is marked as *rollback only*. The user will have to retry the transaction.

This is how the user will know that an update conflict has occurred:

- When Java transaction API is used, a `CacheException` is thrown with the message *Cannot serialize transaction due to write conflict (transaction is marked for rollback)*.
- When SQL transactions are executed through the JDBC or ODBC driver, the user will get `SQLSTATE:40001` error code.

At this moment, the MVCC concurrency control is not recommended for production use and contains a few restrictions and limitations. Please refer to the Ignite official [documentation](#)<sup>135</sup> for more details about MVCC limitations.

## Performance impact on transaction

The following factors impacts on transactions in Apache Ignite:

1. *Number of the backup nodes:* number of the backup nodes directly impact on the transaction performance. The more you have (backup copy of data), the more time it will take to complete the transaction process.

---

<sup>135</sup><https://apacheignite.readme.io/docs/multiversion-concurrency-control#section-limitations>

2. *Data co-allocation*: if all the related datasets located on the same node, a transaction on such datasets will be faster than other datasets.
3. *Backup update mode (synchronous and asynchronous)*: in write *synchronization* mode, before completing a transaction, Ignite client node should wait for an acknowledgment from the remote nodes. This can also increase the transaction wait time on the client side. However, you can choose, `FULL_ASYNC` mode. In this case, the client node does not wait for responses from other nodes.

So, choose your cluster configuration and transaction modes carefully to get the desired performance.

## Summary

We have seen some of the exiting features in action: SQL queries, Query API, Scan and Text queries. We have also covered the Spring data integration and Hibernate OGM with Ignite. Later in this chapter, we described the Ignite native persistence capability and provided a complete example of persisting data into the 3<sup>rd</sup> party NoSQL database. We also explored the Ignite transaction on depth, and MVCC introduced in Ignite 2.7 version.

## What's next?

In the next chapter, we will focus on more advanced features of the Ignite platform. We are going to be familiarized with the Ignite compute grid, which provides a set of simple APIs that allow users to distribute computations and data processing across multiple Ignite nodes in the cluster.

# Chapter 7. Distributed computing

Generally, distributed computing is a system, designed to distribute application tasks over a cluster of computers and coordinate their tasks in parallel. The main principle of the distributed computing is to split a task into multiple parts and execute them on different nodes of the cluster in a parallel fashion. Figure 7.1 shows a very high-level view of a typical distributed system, where each computer has its own local memory, and information can be exchanged only by-passing messages from one node to another by using the available communication links.

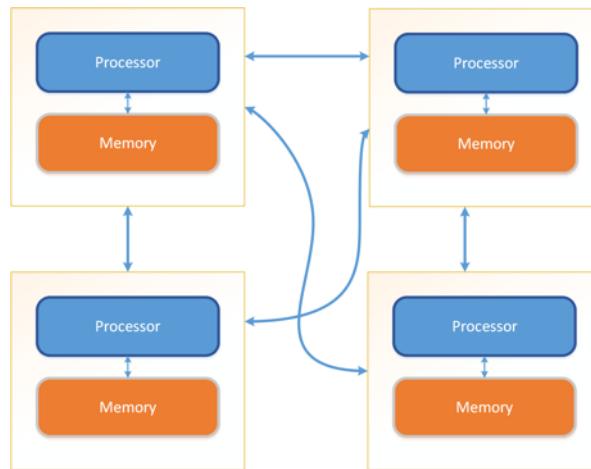


Figure 7.1

Here, the main benefit is that computations will be performed faster as it can now use resources from local node (local data) and also from all grid nodes in parallel. Apache Ignite as a compute grid offers a few advantages and differences over the classic approaches for creating a distributed computing system. Some of them are:

- The classic approach requires resources to be manually allocated and tasks distributed across clusters.
- You have to develop a system for monitoring the running tasks on the cluster and restart them in the case of errors in a classic way.
- Very difficult to use data locality with **compute task**.

Apache Ignite provides a set of simple APIs that allows the user to distribute computation and data processing across multiple nodes in the cluster to gain high performance. The key features of Apache Ignite distributed computing include:

1. *Data locality*: tasks can be bound to local data.
2. *Fork-join*: there is a standard mechanism for the fork-join framework, which allows reliable data transfer between nodes and restarts task in case of failure.
3. *Auto deployment*: no need to install your application/task code on a server, just connect to the server and any new code is automatically taken into account. Users usually are able to simply execute a task from one grid node, and as task execution penetrates the grid. All classes and resources are also automatically deployed.
4. *Checkpoint*: a task can save their states in a checkpoint and recover them from the checkpoint in case of failure. This feature is very useful for the long running task.
5. *Failover and recovery*: provide a built-in failover and recovery for all task in the cluster.
6. *Scalability*: scalability up to an arbitrary number of processing nodes.
7. *Asynchronous communication*: API supports asynchronous communication model for high level of concurrency.
8. *Load balancing*: provides a set of algorithms such as Round-robin, Random or Adaptive for load balancing over the cluster.
9. *Task's session*: through the session, separate job running in the same task can see and interact with each other.

Figure 7.2 is a schematic view of Apache Ignite distributed computing.

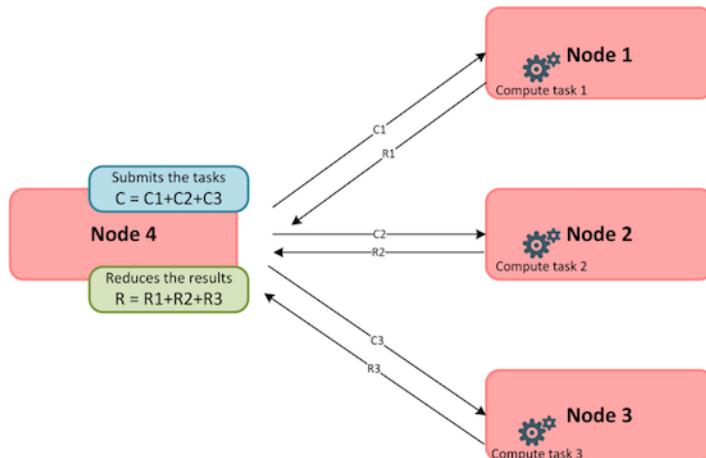


Figure 7.2

In a nutshell, Apache Ignite provides two different approaches to distributed computing: *Compute task* and *Service task*.



## Info

The main difference between *compute task* and *service task* is that **Service task** is a long running process, continuously available independent of topology changes or crashes. Distributed service can also be accessible through service proxy remotely. While, **Compute task** can execute very short live tasks and return an immediate result of the tasks.

For instance, calculating cash back for a client of the bank is the ideal example of a *compute task*. At every month, you execute compute tasks in the Ignite cluster to calculate cash back for a client of the internet bank. On the contrary, you can deploy a *distributed service* in the Ignite cluster to return the last *10 transactions* displayed on the client profile of the bank. Apache Ignite distributed services is very useful to develop and execute microservice like architecture. In the latter part of this chapter, we will describe briefly, the possibilities of Apache Ignite distributed services.

## Compute grid

So far in this chapter, we have learned that compute grid is used to split the task into multiple parts and runs on different nodes. However, Compute grid are useful even if you don't need to split your computation. They can help you improve overall scalability and fault-tolerance of your system by offloading your computations onto most available nodes. In this chapter, we are going to develop a step by step application with subsequent complication to study all the major features of the Ignite Compute grid. We will have the following use case:

Consider a system, where we have to validate *XML messages* with different *XSD schemas* depending on the message type, and executes a few business logics on messages.

A sample XML message for validation is as follows:

Listing 7.1

---

```
<t:message xmlns:t="http://test.msg/">
  <t:headers>
    <t:header>
      <t:name>header1</t:name>
      <t:value>transaction</t:value>
    </t:header>
    <t:header>
      <t:name>header2</t:name>
      <t:value>atm</t:value>
    </t:header>
  </t:headers>
  <t:priority>1</t:priority>
  <t:body>transaction ID: 12dsbfe231df, overdraft</t:body>
</t:message>
```

---

Where, *header1* indicates the type of the header, such as *transaction*, and the *priority tag* which provide the priority of the message. Also, the body part contains the transaction details. The XML schema looks like below:

Listing 7.2

---

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema targetNamespace="http://test.msg/" elementFormDefault="qualified"
  xmlns:xss="http://www.w3.org/2001/XMLSchema" xmlns:t="http://test.msg/">
  <xss:element name="message" type="t:messageType"/>
  <xss:complexType name="headersType">
    <xss:sequence>
      <xss:element type="t:headerType" name="header" maxOccurs="unbounded" minOccurs="0"/>
    </xss:sequence>
  </xss:complexType>
  <xss:complexType name="messageType">
    <xss:sequence>
      <xss:element type="t:headersType" name="headers"/>
      <xss:element type="xs:int" name="priority"/>
      <xss:element type="xs:string" name="body"/>
    </xss:sequence>
  </xss:complexType>
  <xss:complexType name="headerType">
    <xss:sequence>
      <xss:element name="name"/>
```

```
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:enumeration value="header1"/>
    <xs:enumeration value="header2"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element type="xs:string" name="value"/>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

---

Also, we have a simple validation script written in *JAVASCRIPT* shown in listing 7.3.

Listing 7.3

```
xpath("//*[local-name()='message']/*[local-name()='body']").length >0 && xpath("//*[local\
-name()='message']/*[local-name()='priority'])"==1
```

---

Validation script contains *XPATH expression* to check a simple business logic. The XPATH expression checks only the value of the *priority tag*; if the value does not equal to 1 the validation should fail. So, we will use two types of XML validations throughout the chapter: **document validation against XSD schema**, and **XML payload or content validation**.

Let's going to start from the Ignite's very primitive feature: *Distributive closure*.

## Distributed Closures

Distributive closures allow you to broadcast and execute business logic (computation) across cluster nodes, including plain Java runnable and callables. This is the simplest way to run jobs on the remote cluster node. The main entry point of the compute grid is the `IgniteCompute` interface. Interface `IgniteCompute` defines compute grid functionality for executing tasks and closures over nodes in the Ignite cluster or ClusterGroup. The given interface provides the following methods to execute different levels of jobs in the Ignite cluster or ClusterGroup.

Method name	Description
apply()	Execute IgniteClosure jobs over nodes in the cluster or the cluster group.
call()	Execute IgniteCallable jobs over nodes in the cluster or the cluster group.
run()	Execute IgniteRunnable jobs over nodes in the cluster or the cluster group.
broadcast()	Broadcast jobs to all nodes in the cluster or the cluster group.
affinityCall()	Collocate jobs with nodes on which a specified key is cached.



## Tip

If an attempt is made to execute a computation over an empty cluster group (i.e. cluster group that does not have any alive nodes), then Ignite will throw the `org.apache.ignite.cluster.ClusterGroupEmptyException` exception out of the result.

So far, we already have the XSD schema and the JAVASCRIPT validation file. So, let's execute a job on the Ignite cluster. All the sample code of this chapter is available on [GitHub](#)<sup>136</sup> for download. We are going to employ the module `example1` for this section.

**Step 1.** Create a new Maven project or modify the existing one. Add the following Maven dependencies in the project pom.xml file:

Listing 7.4

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-indexing</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-schedule</artifactId>
```

---

<sup>136</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7>

```
<version>1.0.0</version>
</dependency>
```

---

**Step 2.** Create a new Java class named `com.blu.imdg.example1.SimpleComputation` with the following content:

**Listing 7.5**

```
public class SimpleComputation {
    public static void main(String[] args) throws IOException {
        String sample1 = TestDataGenerator.getSample1();
        byte[] validateSchema = TestDataGenerator.getValidateSchema();
        String validateScript = TestDataGenerator.getValidateScript();

        try (Ignite ignite = Ignition.start(CLIENT_CONFIG)) {
            IgniteCompute compute = ignite.compute();

            Boolean result = compute.call(() -> {
                boolean validateXsdResult = XsdValidator.validate(sample1, validateSchema);
                boolean validateByJs = JSEvaluate.evaluateJs(sample1, validateScript);

                System.out.println("validateXsdResult=" + validateXsdResult);
                System.out.println("validateByJs=" + validateByJs);

                return validateXsdResult && validateByJs;
            });

            System.out.println("result=" + result);
        }
    }
}
```

---

Now, let's take a closer look at the above code. In the first three lines of the `main()` method, we initialize our XML schema, validation script, and the XML message itself. Next, we got an instance over all nodes in the cluster through `ignite.compute()` method, and creates a simple Ignite callable job and call the job with the method `igniteCompute.call()`. Then, we validate the *XML message* against *XML schema* and the *priority* value of the message in the callable job. After running the method, we printout the result of the computation into the console.

**Step 3.** Let's execute the Ignite first compute application. So, we have to build the project, and also need an available Ignite node to execute the job. To try out the build, issue the following at the command line:

**Listing 7.6**

---

```
mvn clean install
```

---

This will run Maven, telling it to execute the install goal. When it's finished, you should find the compiled Java \*.class files in the project *target* directory.

**Step 4.** All jobs and closures are guaranteed to be executed as long as there is at least one node standing. So, start an Ignite node. You can start an Ignite node by using the ignite.sh bash script as usual. Alternatively, you can use the com.blu.imdg.StartCacheNode Java class (supplied with the source code) to up and running an Ignite node. This Java class is located in the root of the com.blu.imdg package. Let's use the mvn exec:java command line version to start the Ignite node as follows:

**Listing 7.7**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

Invoking *mvn exec:java* on the command line will invoke the plugin to execute the com.blu.imdg.StartCacheNode class which in turn starts an Ignite node. You should see the following output in the console.

```
[17:58:12] Ignite node started OK [id=f41757cb]
[17:58:12] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=3.6GB]
[17:58:12]   ^-- Node [id=F41757CB-55F2-4B09-AE4-0FF3AFE1E25E, clusterState=ACTIVE]
[17:58:12] Data Regions Configured:
[17:58:12]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
Presse ENTER to exit!
```

**Figure 7.3**

**Step 5.** Execute the Ignite *SimpleComputation* application in the Ignite node. Run the following command from the project root directory (where the pom.xml file for chapter 7 is located):

**Listing 7.8**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example1.SimpleComputation
```

---

Let's see what has happened on the side of the Ignite cluster (I have two nodes Ignite Cluster).

```

validateXsdResult=true
validateByJs=true
[18:19:23] Topology snapshot [ver=5, servers=1, clients=0, CP
Us=8, offheap=3.2GB, heap=3.6GB]
[18:19:23]   ^-- Node [id=f41757CB-55F2-4B09-AEF4-0FF3AFE1E25
E, clusterState=ACTIVE]
[18:19:23] Data Regions Configured:
[18:19:23]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB
, persistenceEnabled=false]
[18:19:23] Ignite node started OK (id=eff2f49c)
[18:19:23] Topology snapshot [ver=4, servers=1, clients=1,
CPUs=8, offheap=3.2GB, heap=7.1GB]
[18:19:23]   ^-- Node [id=EFF2F49C-1BEB-4B2E-A63A-53368D462
C1C, clusterState=ACTIVE]
result=true
[18:19:23] Ignite node stopped OK [uptime=00:00:00.098]
[INFO] -----

```

Figure 7.4

One of the nodes of the Ignite cluster executed the computation and give back the result to the client. Ignite must select a node for a computation to be executed in all cases other than `broadcast(...)`. The node will be selected based on the underlying *LoadBalancingSpi*, which sequentially picks next available node from the underlying cluster group by default. Other load balancing policies, such as *random* or *adaptive*, can be configured as well.

Also, Ignite guarantees that as long as there is at least one grid node standing, every job will be executed. Jobs will automatically failover to another node if a remote node crashed or has rejected execution due to lack of resources. In the case of failover, next node will be picked for job execution by default. Also, jobs will never be re-routed to the nodes they have failed on. This performance can be changed by configuring any of the existing or a custom *FailoverSpi* in a grid configuration.

You can also execute an `IgniteClosure` through an `IgniteCompute.apply()` method in Ignite cluster. A closure is a block of code that encloses its body and any outside variables used inside of it as a function object. You can then pass such function object anywhere you can pass a variable and execute it. All `apply(...)` methods execute closures on the cluster. A new job is executed for every argument passed in the collection. The number of actual job executions will be equal to the size of the job arguments collection.

**Step 6.** Create a new Java class named `SimpleComputationClosure` in the `/src/main/com/blu/imdg/example1` directory. Add the following content to the Java class.

Listing 7.9

---

```

public class SimpleComputationClosure {
    public static void main(String[] args) throws IOException {
        try (Ignite ignite = Ignition.start(CLIENT_CONFIG)) {
            IgniteCompute compute = ignite.compute();
            // Execute closure on all cluster nodes.
            Collection<Integer> res = compute.apply(
                new IgniteClosure<String, Integer>() {
                    @Override
                    public Integer apply(String word) {
                        // Return number of letters in the sentence.
                        return word.length();
                    }
                }
            );
        }
    }
}

```

```

        }
    },
    Arrays.asList("Count characters using closure".split(" "))
);
int sum = 0;
//Add up individual word lengths received from remote nodes
for (int len : res)
    sum += len;
System.out.println("Length of the sentence: "+ sum);
}
}
}

```

---

The above code will define `IgniteClosure` with `apply()` function. As an argument of the function, we pass the `words` split by the `""`. We got the collection of word length in return, which we aggregate together and output into the console.

**Step 7.** Now, run the `SimpleComputationClosure` application as same before. Use the following command:

#### Listing 7.10

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example1.SimpleComputationClosure
```

---

Which should return the result of the sentence as follows.

```
[18:33:46] Ignite node started OK [id=cf52c2c]
[18:33:46] Topology snapshot [ver=6, servers=1, clients=1, CPUs=8, offheap=3.2GB, heap=7.1GB]
[18:33:46]   ^-- Node [id=CF52C2C-BF74-4265-A759-D892A3CFB36, clusterState=ACTIVE]
Length of the sentence: 27
[18:33:46] Ignite node stopped OK [uptime=00:00:00.066]
```

Figure 7.5

In the above examples, there is a problem with the client thread. A client thread always waits for the result from the Ignite cluster. Since the computation can take a long time, it is always preferable to use an *asynchronous* call. Apache Ignite provides a simple and convenient way to call remote computing by `callAsync()` method.

**Step 8.** Create an another Java class with name `AsyncComputation` in the `/src/main/com/blu/imdg/example2` directory. Add a `main()` method with the following contents.

**Listing 7.11**

---

```
public static void main(String[] args) throws IOException {
    String sample1 = TestDataGenerator.getSample1();
    byte[] validateSchema = TestDataGenerator.getValidateSchema();
    String validateScript = TestDataGenerator.getValidateScript();

    try (Ignite ignite = Ignition.start(CommonConstants.CLIENT_CONFIG)) {
        IgniteCompute compute = ignite.compute();

        IgniteFuture<Boolean> result = compute.callAsync(() -> {
            boolean validateXsdResult = XsdValidator.validate(sample1, validateSchema);
            boolean validateByJs = JSEvaluate.evaluateJs(sample1, validateScript);

            System.out.println("validateXsdResult=" + validateXsdResult);
            System.out.println("validateByJs=" + validateByJs);

            return validateXsdResult && validateByJs;
        });

        result.listen((r) -> {
            boolean res = (boolean) result.get();
            System.out.println("result=" + res);
        });
    }
}
```

---

The above code is very similar to others we have written before. The main difference here is the `compute.callAsync()` method. The `compute.callAsync()` method enable the Ignite asynchronous call to the Ignite cluster. Ignite `compute.callAsync()` method allows to not wait for the result after invocation and gets the result asynchronously when any of the nodes returns the result. If you execute the application with the following command:

**Listing 7.12**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example2.AsyncComputation
```

---

You should get the same result but, **asynchronously**.

## MapReduce and Fork-join

In this section, we are going to complicate our validation process of XML messages. We have to validate *a group of messages* and get the final result for the *entire group of a message*. In-memory MapReduce approach is the best of all in such condition. Apache Ignite in-memory MapReduce implementation is very close to the *Fork-join* paradigm and considers as a light-weight MapReduce implementation.

Apache Ignite provides `ComputeTask` interface that simplified the in-memory MapReduce and defines a task that can be executed on the Ignite grid. `ComputeTask` is responsible for splitting business logic into multiple compute jobs, receiving results from individual compute jobs executing on remote nodes, and reducing (aggregating) received results into final compute task result.

Ignite `ComputeTask` has three different phases as follows:

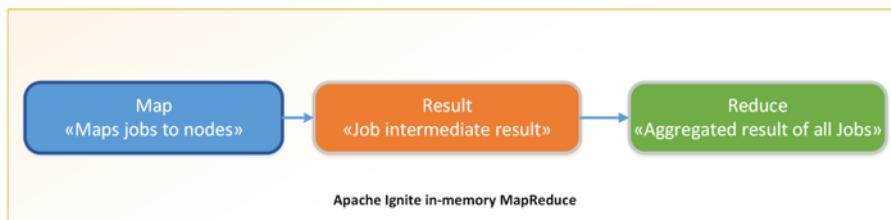


Figure 7.6

- *Map*. In this phase method `map(...)` instantiates the jobs and maps them to worker nodes. The method receives the collection of cluster nodes on which the task is run. The method should return a map with jobs as keys, and mapped worker nodes as values. The jobs are then sent to the mapped nodes and executed there.
- *Result*. It's an intermediate outcome of the completing job. Method `result(...)` is called each time a job completes on some cluster node. It receives the result returned by the completed job, as well as the list of all the job results received so far. The method should return a `ComputeJobResultPolicy` instance, indicating what to do next:
  - WAIT - wait for all remaining jobs to complete (if any).
  - REDUCE - immediately move to reduce step, discarding all the remaining jobs and unreceived results.
  - FAILOVER - failover the job to another node (see the Fault Tolerance section for details).
- *Reduce*. It is the aggregated result of all jobs. In this phase, method `reduce(...)` is called when all the jobs have been completed (or REDUCE result policy was returned from the

`result(...)` method). The method receives a list with all the completed results and should return a final result of the computation.

Figure 7.7 illustrates all the three execution phases in the Ignite grid.

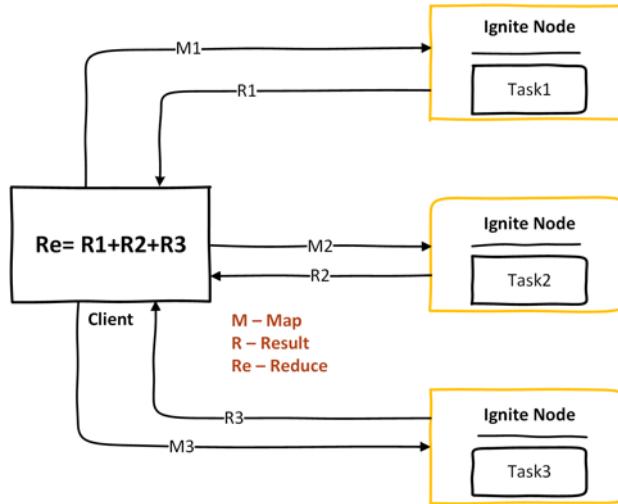


Figure 7.7

## Note:

ComputeTask is preferable over Distributed closure when you need fine-grained control over job to node mapping or custom failover logic.

Let's implement the full version of the ComputeTask, and run some validation of XML messages over Ignite cluster. However, the complete source code is available on [GitHub repository<sup>137</sup>](#) at project *example3*.

**Step 1.** Firstly, we have to add the structure of the *message*. Create a Java class with name `ValidateMessage` into directory `/src/main/java/com/blu/imdg/example3`. Add the following content to the Java class.

<sup>137</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example3>

Listing 7.13

---

```
public class ValidateMessage {  
    private String id;  
    private String msg;  
    private byte[] xsd;  
    private String js;  
    public ValidateMessage(String id, String msg, byte[] xsd, String js) {  
        this.id = id;  
        this.msg = msg;  
        this.xsd = xsd;  
        this.js = js;  
    }  
    public String getMsg() {  
        return msg;  
    }  
    public byte[] getXsd() {  
        return xsd;  
    }  
    public String getJs() {  
        return js;  
    }  
    public String getId() {  
        return id;  
    }  
}
```

---

It's a plain old Java class, which have the following fields:

- id: message identifier.
- msg: the content of the message of type String.
- xsd: the content of the XSD schema of type byte array.
- js: the content of the JAVASCRIPT validation script of type String.

**Step 2.** Add *three* XML documents into the `src/main/resources/META-INF/org/book/examples` directory with the similar contents as shown in listing 7.14.

**Listing 7.14**

---

```
<t:message xmlns:t="http://test.msg/">
  <t:headers>
    <t:header>
      <t:name>header1</t:name>
      <t:value>transaction</t:value>
    </t:header>
    <t:header>
      <t:name>header2</t:name>
      <t:value>atm</t:value>
    </t:header>
  </t:headers>
  <t:priority>1</t:priority>
  <t:body>transaction ID: 12dsbfe231df, overdraft</t:body>
</t:message>
```

---

In my case, I have renamed the XML documents as *sample1.xml*, *sample2.xml*, *sample3.xml*. You can easily create the XML documents from the XSD schema provided by the project. The *validate-schema.xsd* file also placed in the same folder of this project.

**Step 3.** Create or modify the Java class *ForkJoinComputation* in the */src/main/java/com/blu/imdg/example3* directory. Add the following content to the Java class:

**Listing 7.15**

---

```
public class ForkJoinComputation {

  public static void main(String[] args) throws IOException {
    try (Ignite ignite = Ignition.start(CommonConstants.CLIENT_CONFIG)) {

      IgniteCompute compute = ignite.compute();
      ValidateMessage[] validateMessages = TestDataGenerator.getValidateMessages();
      Boolean result = compute.execute(new ComputeTask<ValidateMessage[], Boolean>()
        ) {
        @LoadBalancerResource
        private ComputeLoadBalancer balancer;
        @Nullable
        @Override
        public Map<?, extends ComputeJob, ClusterNode> map(List<ClusterNode> list,
          @Nullable ValidateMessage[] validateMessages) throws IgniteException {
          Map<ComputeJob, ClusterNode> result = Maps.newHashMap();
          for (ValidateMessage msg : validateMessages) {
```

```

        ComputeJobAdapter job = new ForkJoinJobAdapter(msg);
        ClusterNode balancedNode = balancer.getBalancedNode(job, null);
        result.put(job, balancedNode);
    }
    return result;
}
@Override
public ComputeJobResultPolicy result(ComputeJobResult computeJobResult, List<ComputeJobResult> list) throws IgniteException {
    IgniteException e = computeJobResult.getException();
    if (e != null) {
        if (!(e instanceof ComputeExecutionRejectedException) && !(e instanceof ClusterTopologyException) && !e.hasCause(new Class[]{ComputeJobFailoverException.class})) {
            throw new IgniteException("Remote job threw user exception (override or implement ComputeTask.result(..) method if you would like to have automatic failover for this exception).", e);
        } else {
            return ComputeJobResultPolicy.FAILOVER;
        }
    } else {
        return ComputeJobResultPolicy.WAIT;
    }
}
@Nullable
@Override
public Boolean reduce(List<ComputeJobResult> list) throws IgniteException {
    return list.stream().reduce(true, (acc, value) -> acc && (Boolean) value.getData(), (a, b) -> a && b);
}, validateMessages);
System.out.println("result=" + result);
}
}
}

```

---

Let's go through the above code, line by line. In line 8:

```
ValidateMessage[] validateMessages = TestDataGenerator.getValidateMessages();
```

We have initialized and generate our sample XML documents. In the next line, we have

defined anonymous *ComputeTask* and implement the three functions: *Map()*, *Result()\_\_ and \_Reduce()*.

- In the Map method: we used the computing *Map<ComputeJob, ClusterNode>* object, create jobs for each XML message through *ForkJoinJobAdapter*. Also we defined the *ComputeLoadBalancer* interface that connects to the Ignite cluster and returns the next load balanced node according to the underlying load balancing policy.
- In the Result method: we are analyzing the execution result of the *ComputeJob*. At first, we check for any error by calling the method `computeJobResult.getException()`, then returns the *ComputeJobResultPolicy.WAIT* policy.
- In the Reduce method: gets all the result as a stream, and process them.

**Step 4.** Next, start a few Ignite nodes to run the `ForkJoinComputation` application. Run the following command from a few different terminals to start a few Ignite nodes as follows:

**Listing 7.16**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

I have started three different Ignite nodes with the above command.

**Step 5.** Run the `ForkJoinComputation` application from the command line through Maven `exec` command.

**Listing 7.17**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example3.ForkJoinComputation
```

---

You should discover the following output in the console if all your three XML messages are valid.

[3.2 GiB, persistenceEnabled=false] msg=3 validateXsdResult=true validateByJs=true [19:26:59] Topology snapshot [ver=5, servers=3, client s=0, CPUs=8, offheap=9.6GB, heap=11.0GB] [19:26:59] ^-- Node [id:EDF95CBE-4DF3-4E5D-8FCE-73ED F9F12009, clusterState=ACTIVE]	[e=3.2 GiB, persistenceEnabled=false] msg=2 validateXsdResult=true validateByJs=true [19:26:59] Topology snapshot [ver=5, servers=3, client s=0, CPUs=8, offheap=9.6GB, heap=11.0GB] [19:26:59] ^-- Node [id:68CAC994-68C4-40B7-8713-F2C1F24FCBF2, clusterState=ACTIVE]	[e=3.2 GiB, persistenceEnabled=false] msg=2 validateXsdResult=true validateByJs=true [19:26:59] Topology snapshot [ver=5, servers=3, client s=0, CPUs=8, offheap=9.6GB, heap=11.0GB] [19:26:59] ^-- Node [id:9D955832-9FE9-447A-A159-77 9196A2C77C, clusterState=ACTIVE]
---	--	---

**Figure 7.8**

You can notice that, for each XML document, one task is created in each Ignite node and all the XML messages are valid. Let's make a few changes in one of the XML messages so, it will not be valid, and then re-execute the application again. In my case, I have changed the XML namespace in the `sample2.xml` XML file as shown below:

```
<t:message xmlns:t="validate-schema.xsd">
```

Before executing the `ForkJoinComputation` application, do not forget to build the project, otherwise, any changes you made in the `sample2.xml` XML file may not take into action.

```
msg=3
validateXsdResult=false
validateByJ=ture
[19:37:56] Topology snapshot [ver=7, servers=3, client
s=0, CPUs=8, offheap=9.6GB, heap=11.0GB]
[19:37:56] ^-- Node [id=EDF5CBE-4DF3-4E5D-8FCE-73ED
F9F12009, clusterState=ACTIVE]
```

```
msg=1
validateXsdResult=true
validateByJ=ture
[19:37:56] Topology snapshot [ver=7, servers=3, cli
ents=0, CPUs=8, offheap=9.6GB, heap=11.0GB]
[19:37:56] ^-- Node [id=6BAC994-6BC4-40B7-8713-F
2C1F24FCBF2, clusterState=ACTIVE]
```

```
msg=2
validateXsdResult=true
validateByJ=ture
[19:37:56] Topology snapshot [ver=7, servers=3, clie
nts=0, CPUs=8, offheap=9.6GB, heap=11.0GB]
[19:37:56] ^-- Node [id=90955832-9FE9-447A-A150-77
9106A2C77C, clusterState=ACTIVE]
```

Figure 7.9

At this moment, validation of the XML message of the `sample2.xml` not passed. If we check the entire result of the validation process, you should find the next result in the console as shown below.

```
[19:37:56] Ignite node started OK (id=26d31537)
[19:37:56] Topology snapshot [ver=6, servers=3, clients=1, CPUs=8, offheap=9.6GB, heap=14.0GB]
[19:37:56] ^-- Node [id=26D31537-294C-4478-B493-A12098AC59CD, clusterState=ACTIVE]
result=false
```

Figure 7.10

The entire result is `false`, which is what we have expected. In this approach you can add thousands of XML messages to map over Ignite nodes and execute distributed computation. Most often you don't need the full implementation of the `ComputeTask`. There is a number of helper classes that let you provide only a particular piece of your logic, leaving out all the rest for Ignite to handle automatically. Let's start by the compute job adapter.

**Compute job adapter.** A convenience adapter for `ComputeJob` implementations. All jobs that are generated by a task are implementations of the `ComputeJob` interface. The `execute()` method of this interface defines the business logic and should return a result. `ComputeJobAdapter` provides the default implementation of `ComputeJob.cancel()` method and ability to check whenever a cancellation event occurred with `isCancel()` method. The adapter can get job arguments by method constructor or via `setArguments(Object...)` method. We explicitly used the compute job adapter (`com.blu.imdg.example3.ForkJoinJobAdapter`) to execute our XML messages validation in our last example.

```
ComputeJobAdapter job = new ForkJoinJobAdapter(msg);
```

Where `ForkJoinJobAdapter` extends the `ComputeJobAdapter` and implements the following business logic.

```
@Override  
public Boolean execute() throws IgniteException {  
    boolean validateXsdResult = XsdValidator.validate(msg.getMsg(), msg.getXsd());  
    boolean validateByJs = JSEvaluate.evaluateJs(msg.getMsg(), msg.getJs());  
    System.out.println("msg=" + msg.getId());  
    System.out.println("validateXsdResult=" + validateXsdResult);  
    System.out.println("validateByJs=" + validateByJs);  
    return validateXsdResult && validateByJs;  
}
```

The method `execute()` executes the two validations of the XML messages: one for the XML schema and another for the XML messages contents.

**Compute task adapter.** `ComputeTaskAdapter` is an implementation of `ComputeTask` interface and provides a default implementation of the `result()` method, which will wait for all jobs to complete before calling the `reduce()` method. A *FAILOVER* policy would be `return` if remote job threw an *exception*. On the other hand, *WAIT* policy will be `return`, if jobs are *waiting* for finished.

**Compute task split adapter.** This is the simplified adapter for `ComputeTask` interface. This adapter extends the `ComputeTaskAdapter` and adds the capability to assign jobs to nodes automatically. This adapter can be used when jobs can be randomly assigned to available grid nodes. This adapter is sufficient in most homogeneous environments where all nodes are equally suitable for executing grid job. It hides the `map()` method and adds a new `split()` method in which user only needs to provide a collection of the jobs to be executed. The `split()` method basically takes given an argument and splits it into a collection of compute jobs using provided grid size as an indication of how many nodes are available.



## Tip

The grid nodes will be reused and some jobs will end up on the same grid nodes if the number of jobs is greater than the number of grid nodes (i.e., grid size).

Dependencies between the above classes and interfaces can be expressed by the UML Class diagram. Take a look at the class diagram below.

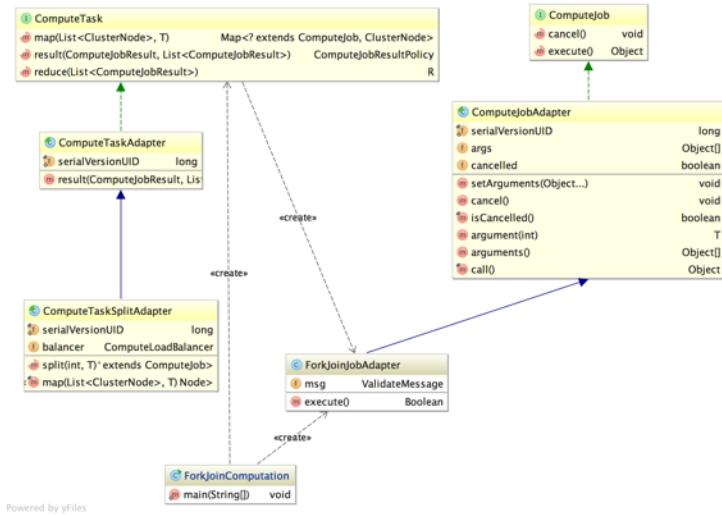


Figure 7.11

Where `ForkJoinJobAdapter` class and the `ForkJoinComputation` is our user-defined class. The `ComputeTask` interface, and the `ComputeJob` interface are the two separate interfaces working with Tasks and Jobs.

In the next few sections, we will broaden our XML validation example by using Compute task split adapter and will explore a few new features of the Ignite compute grid.

## Per-Node share state

Ignite provides access to special local data storage to each cluster node. This local data storage is available to all jobs running on this node. There is only one instance of local node storage per local node. The local node storage is based on `java.util.concurrent.ConcurrentMap` and safe for multi-threaded access. This local storage also provides the atomicity guarantees.

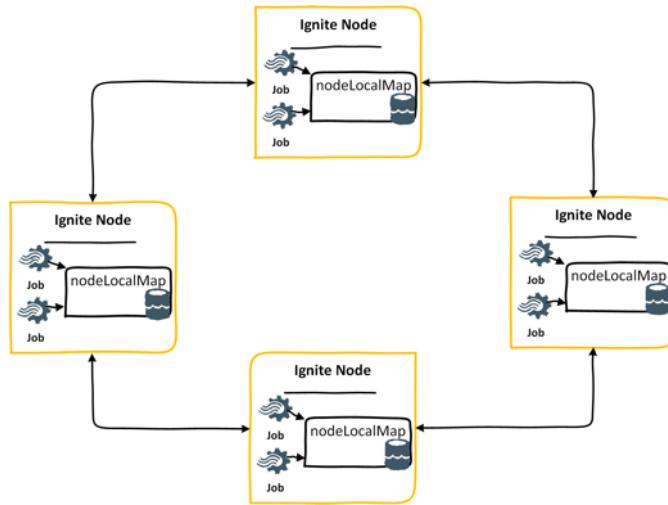


Figure 7.12

Node-local values are similar to thread locals in such a way that these values are not distributed and kept only on the local node (similar to thread local values are attached to the current thread only). *Node-local* values are used primarily by closures executed from the remote nodes to keep intermediate state on the local node between executions. This node local storage is useful to share a state between different compute jobs or different deployed services in a single node. For instance, we can create an HTTP connection pool per node in the local node storage for connecting to any third-party HTTP service. The process of initiating a HTTP connection from one host to another is quite complex and involves multiple packet exchanges between two endpoints, which can be quite time-consuming. The overhead of connection handshaking can be significant, especially for small HTTP messages. We can achieve a much higher data throughput if open connections can be re-used to execute multiple requests by using the local per-node HTTP connection pool.

Let's continue to modify our transactions validation application. This time, we want to organize the interaction of the message verification process with any external system. So we are going to send a few notifications of our validation result to an external HTTP server for audit. We have to keep a pool of HTTP connection in each Ignite node to effectively achieve this goal. A high-level view of this process is shown in figure 7.13.

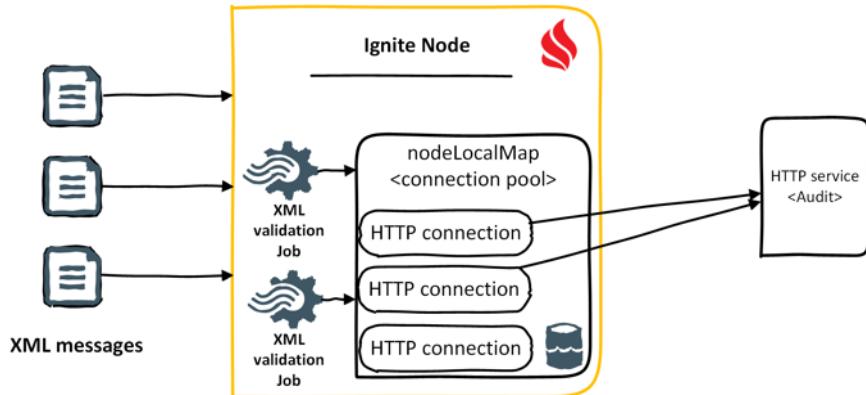


Figure 7.13

In the above scenario, there are three main components in our system.

- HTTP server: which can accept transaction validation result and act as a third-party service.
- HTTP connection pool: composed by the local node map and contain open connections. Whenever any job finished the XML validation and wants to send a notification to the HTTP server, it consumes a pre-created HTTP connection from the pool and sends the notification to the HTTP server rather than create an HTTP connection each time.
- XML validation job: an instance of a Compute task split adapter. The main purpose of this adapter is to validate incoming XML files and direct the result of the validations to the HTTP server

The full example is available at [GitHub repositories<sup>138</sup>](https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example4). You can execute the example by the following few commands. First, start the HTTP server from the project root directory as shown below:

#### **Listing 7.18**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.common.HttpAuditEmulator
```

---

Next, start an Ignite node, use the command shown in listing 7.19.

---

<sup>138</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example4>

**Listing 7.19**

---

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

Now, you can submit a job to Ignite node for validating XML files as follows:

**Listing 7.20**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example4.ForkJoinComputationExt
```

---

Note that, we used `ForkJoinComputationExt` as an implementation of the `Ignite ComputeTaskSplitAdapter` adapter. If everything goes fine, you should see the validation result in the HTTP server console as shown below:

```
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ chapter-7 ---
log message validation result msgId=3 result=true
log message validation result msgId=2 result=true
log message validation result msgId=1 result=true
```

Figure 7.14

Let's have a detailed look at the example of the chapter. Clone or download the `chapter-7/src/main/java/example4` project from the GitHub repository. If you create your own Maven project, add the below dependencies into your `pom.xml` file.

**Listing 7.21**

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.2</version>
</dependency>
```

---

The key dependencies are:

- Ignite-core: Ignite core module for working with Ignite compute grid. This maven dependency will pull all other required modules to work with the Ignite grid.
- HttpClient: is designed to work with any HTTP service on the web. It provides a pluggable HTTP transport abstraction so you can use any low-level library such as `java.net.HttpURLConnection`.

Let's create the *HTTP service* first. In our case, it will be a very simple HTTP REST service, which will accept the XML validation result by the following URL path `/audit/log`.

**Step 1.** Add the Java class with name `HttpAuditEmulator` in the `com.blu.imdg.common` package of the project.

**Listing 7.22**

---

```
@Path("/audit")
public class HttpAuditEmulator {
    @GET
    @Path("/log")
    @Produces(MediaType.TEXT_PLAIN)
    public String acceptResponse(
        @QueryParam("msgId") String msgId,
        @QueryParam("validationResult") boolean result
    ) {
        System.out.println("log message validation result msgId=" + msgId + " result=" + \
result);
        return "1";
    }

    public static void main(String[] args) {
        UriBuilder.fromUri("http://localhost/").port(9998).build();
        ResourceConfig config = new ResourceConfig(HttpAuditEmulator.class);
        HttpServer server = JdkHttpServerFactory.createHttpServer(baseUri, config);
    }
}
```

---

We are using the very basic of the *JAX-RS* annotation for creating an HTTP service. `HttpAuditEmulator` has one method `acceptResponse()`, which can accept HTTP GET request with the path `/audit/log` and can take query parameter `msgID`, and `validationResult`. The `acceptResponse()` method in response returns the HTTP result with value `1`. Note that the HTTP server runs on the port `9998`.

**Step 2.** At this time, we have to add another Java class to create *HTTP client factory*. Create a new Java class in the same package (com.blu.imdg.common) as before, and name the class as `HttpAuditClient`.

**Listing 7.23**

---

```
public class HttpAuditClient {  
  
    public static Boolean sendResult(HttpClient client, Boolean result, String messageId) \  
throws URISyntaxException, IOException {  
    URI uri = new UriBuilder().setScheme("http").setHost("localhost").setPort(9998).s\  
etPath("/audit/log")  
        .setParameter("msgId", messageId)  
        .setParameter("validationResult", result.toString())  
        .build();  
    HttpGet httpget = new HttpGet(uri);  
    try (CloseableHttpResponse resp = (CloseableHttpResponse) client.execute(httpget)) {  
        System.out.println("msg=" + messageId + " result=" + result);  
        return result;  
    }  
}  
  
    public static HttpClient createHttpClient(ConcurrentMap<Object, Object> nodeLocalMap) {  
        return (HttpClient) nodeLocalMap.computeIfAbsent("httpClient", new Function<Objec  
t, HttpClient>() {  
            @Override  
            public HttpClient apply(Object o) {  
                return createHttpClient();  
            }  
        });  
    }  
  
    public static HttpClient createHttpClient() {  
        PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager();  
        return HttpClients.custom()  
            .setConnectionManager(cm)  
            .build();  
    }  
}
```

---

The above code fragment illustrates the creation of HTTP clients with connection pooling

ability. Let's have a detailed look at the Java class. First, we create a Java method `createHttpClient()` without any parameter. The given class uses `PoolingHttpClientConnectionManager`, a more complex implementation that manages a pool of client connections and is able to service connection requests from multiple execution threads. By default, this implementation will create no more than two concurrent connections per given route and no more than 20 connections in total. However, you can increase the total connections for the connection pool as follows:

```
cm.setMaxTotal(200);
```

Next, we have another override method `createHttpClient(ConcurrentMap<Object, Object> nodeLocalMap)` with parameter of Ignite `nodeLocalMap`. In this method, we used the `ConcurrentMap.computeIfAbsent()` method to create an instance of the `HTTPClient` with name `httpClient` whenever needed. Our final method of this class is `sendResult()`. In a nutshell, this method constructs an HTTP *Get* request from the method parameters and send the request to the HTTP server.

**Step 3.** This is the final part of our application. In this step, we will create a *job adapter*, and a *compute task split adapter*. Let's create the Job adapter first.

**Listing 7.24**

---

```
public class ForkJoinJobAdapterExt extends ComputeJobAdapter {
    @IgniteInstanceResource
    private Ignite ignite;

    private ValidateMessage msg;

    public ForkJoinJobAdapterExt(ValidateMessage msg) {
        this.msg = msg;
    }

    @Override
    public Boolean execute() throws IgniteException {
        try {
            boolean validateXsdResult = XsdValidator.validate(msg.getMsg(), msg.getXsd());
            boolean validateByJs = JSEvaluate.evaluateJs(msg.getMsg(), msg.getJs());

            Boolean result = validateXsdResult && validateByJs;

            ConcurrentMap<Object, Object> nodeLocalMap = ignite.cluster().nodeLocalMap();

            HttpClient client = HttpAuditClient.createHttpClient(nodeLocalMap);
        }
    }
}
```

```
    return HttpAuditClient.sendResult(client, result, msg.getId());\n\n} catch (Exception err) {\n    throw new IgniteException(err);\n}\n}\n}
```

---

Basically, it is an instance of `ComputeJobAdapter` adapter, which we have seen before. In this adapter, we implemented our validation process and peek an HTTP client from the local node storage to send the result to HTTP server for audit. Also, note that we used the annotation `@IgniteInstanceResource` to auto-inject the Ignite current instance to the compute adapter. Ignite instance can be injected into grid tasks and grid jobs. The next few instructions performed the main setup of the HTTP client into the local node map as follows:

```
ConcurrentMap<Object, Object> nodeLocalMap = ignite.cluster().nodeLocalMap();\nHttpClient client = HttpAuditClient.createHttpClient(nodeLocalMap);\nreturn HttpAuditClient.sendResult(client, result, msg.getId());
```

Instance of the per-node shared state we got by calling the method `ignite.cluster().nodeLocalMap()`, and construct the HTTP connection pool with default HTTP client. Later we invoke the HTTP GET method by using the `HttpAuditClient.sendResult()` function.

Next, we are going to implements a Compute task split adapter, in which we only need to provide a collection of jobs to be executed.

#### Listing 7.25

```
public class ForkJoinComputationExt extends ComputeTaskSplitAdapter<ValidateMessage[], Boolean>\n    olean> {\n        @Override\n        protected Collection<? extends ComputeJob> split(int i, ValidateMessage[] messages) throws IgniteException {\n            return Arrays.stream(messages).map(ForkJoinJobAdapterExt::new).collect(Collectors.toList());\n        }\n\n        @Nullable\n        @Override\n        public Boolean reduce(List<ComputeJobResult> list) throws IgniteException {\n            return list.stream().reduce(true, (acc, value) -> acc && (Boolean) value.getData());\n        }\n    }
```

```
), (a, b) -> a && b);
}

public static void main(String[] args) throws IOException {
    try (Ignite ignite = Ignition.start(CLIENT_CONFIG)) {
        IgniteCompute compute = ignite.compute();

        ValidateMessage[] validateMessages = TestDataGenerator.getValidateMessages();
        Boolean result = compute.execute(new ForkJoinComputationExt(), validateMessages);
        System.out.println("result=" + result);
    }
}
```

---

Here, we added a new `split()` method in which we allow our `ForkJoinJobAdapterExt` adapter to be executed in the cluster. The `split()` method basically takes a few arguments and splits them into a collection of `ComputeJob`. Also, we override the `reduce()` method to compute the final result of the XML file validations. If any of the XML files doesn't pass the validation, the entire result will be invalid. Finally, we have the `main()` method to initialize the Ignite client and deploy the job into the Ignite compute grid.

**Step 4.** Rename the XML namespace of the `sample3.xml` file as follows:

```
<t:message xmlns:t="http://test.msg111/">
```

Now, the entire XML file is invalid against XSD schema. Run the application again.

#### Listing 7.25

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example4.ForkJoinComputationExt
```

---

You should notice the following log into the console.

```
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ chapter-7 ---
log message validation result msgId=1 result=true
log message validation result msgId=2 result=true
log message validation result msgId=3 result=false
```

Figure 7.15

If you check the console of the `ForkJoinComputationExt` application, you should discover that the entire validation result is false.

Ignite per-node share state is a very powerful feature and you can employ it to solve not only simple use cases but also very complex use cases to achieve a much higher data throughput such as HTTP connection pooling or database connection pooling etc. In the next section, we will explore the Ignite distributed task session to share state between different jobs.

## Distributed task session

Apache Ignite provides a distributed session for particular task execution. It is defined by the `ComputeTaskSession` interface. Task session is distributed across the parent task, and all grid jobs are spawned by it, so attributes set on a parent task or job can be viewed on other jobs. Task session also allows to receiving notifications when attributes are set or wait for an attribute to be set.

Distributed task session has two main features: *attribute* and *checkpoint management*. Both attributes and checkpoints can be used for the task itself from the jobs belonging to this task. Session attributes and checkpoints can be set from any task or job methods. Session attribute and checkpoint consistency is fault tolerant and is preserved whenever a job gets failed over to another node for execution. Whenever task execution ends, all checkpoints saved within a session with `ComputeTaskSessionScope` scope will be removed from the checkpoint storage. We will explore the Ignite checkpoint feature a little bit later.

The sequence in which session attributes are set is consistent across the task and all job siblings within it. There will never be a case when one job sees attribute A before attribute B, and another job sees attribute B before A. Attribute order is identical across all session participants. Attribute order is also fault tolerant and is preserved whenever a job gets failed over to another node.

With distributed task session, we can coordinate and synchronize all the jobs across the task. For instance, assuming we are getting bundle of XML messages with more than 100 messages in one package. First, we have to validate all the messages from the package, if all of them are validate then we delegate all the message for content validation through JAVASCRIPT (see section [Distributed closure](#) for more details about the validation process). We will stop the rest of the processing immediately and send a notification for audit if any of the XML messages will fail to pass the validation against XSD schema. To do that, in the grid environment, we could execute a few jobs which will start validating the XML messages against XSD schema one by one.

In the next example, we will stop the further validation process immediately and send the result to the HTTP server for audit if any of the XML messages will fail to pass the validation. Without session attribute, to coordinate this process would be much harder to solve. So, we have to slightly modify our previous `ComputeJobAdapter` adapter to use the task session. Also, we are going to use the `ComputeJobContext` to get the job done.



## Info

Unlike `ComputeTaskSession`, which distributes all attributes to each jobs in the task including the task itself, *job context attributes* belong to a job and do not get sent over the network unless a job moves from one node to another.

The full example of this sub-section is available at the [GitHub repositories](#)<sup>139</sup>. Now that we have got the basics, let's start creating the necessary classes.

**Step 1.** Create a new Java class with the name `ForkJoinWithSessionJobAdapter` in the `com.blu.imdg.example5` package, and extends it from the `ComputeJobAdapter` abstract class. Add the following content to the class.

**Listing 7.26**

---

```
public class ForkJoinWithSessionJobAdapter extends ComputeJobAdapter {
    @TaskSessionResource
    private ComputeTaskSession session;

    @JobContextResource
    private ComputeJobContext jobCtx;

    @IgniteInstanceResource
    private Ignite ignite;

    private ValidateMessage msg;

    public ForkJoinWithSessionJobAdapter(ValidateMessage msg) {
        this.msg = msg;
    }

    @Override
    public Boolean execute() throws IgniteException {
        try {
            boolean validateXsdResult = XsdValidator.validate(msg.getMsg(), msg.getXsd());
            session.setAttribute(jobCtx.getJobId(), validateXsdResult);
        }
    }
}
```

<sup>139</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example5>

```

if (!validateXsdResult) {
    System.out.println("force return result false!");
    return sendResultAndReturn(false);
}

for (ComputeJobSibling sibling : session.getJobSiblings()) {
    Boolean siblingStep1Result = session.waitForAttribute(sibling.getJobId(), \
0);
    if(!siblingStep1Result) {
        System.out.println("one sibling return false!");
        return sendResultAndReturn(false);
    }
}

boolean validateByJs = JSEvaluate.evaluateJs(msg.getMsg(), msg.getJs());
return sendResultAndReturn(validateByJs);

} catch (Exception err) {
    throw new IgniteException(err);
}
}

@NotNull
private Boolean sendResultAndReturn(Boolean result) throws URISyntaxException, IOExce\
ption {
    ConcurrentMap<Object, Object> nodeLocalMap = ignite.cluster().nodeLocalMap();
    HttpClient client = HttpAuditClient.createHttpClient(nodeLocalMap);
    return HttpAuditClient.sendResult(client, result, msg.getId());
}
}

```

---

Let's go through the code line by line. First, we *auto-inject* the following resources into the above adapter.

- ComputeTaskSession
- ComputeJobContext and
- Ignite

Next, we override the `execute()` method with our custom logics. First, we used our `XsdValidator` to validate the XML message and set the `session` attribute. Session attribute name will be the job identifier, and the value will be the result of the XML file validation. In the next few lines, we check the validation result, if it is *false*, we immediately return the result. Otherwise, we

go through all of the job siblings and waits for the attribute's value from other jobs. If any of the sibling jobs returns a *negative* result, we stop further processing of the XML validation and send the result to the HTTP server. Once all the message passed the XSD validation process, we start doing message *content validation* through *XPATH* expression.

Rest of the parts of this example is same as before. In the *compute task split adapter*, we split the task across the nodes in the grid. Let's run the application, first start the HTTP server as follows (if it is not already started).

**Listing 7.27**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.common.HttpAuditEmulator
```

---

Next, start the an Ignite node as done before:

**Listing 7.28**

---

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

Now, run the application with the following command::

**Listing 7.29**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example5.ForkJointWithSessionComputation
```

---

You should see the validation result in the Ignite server console as shown below.

```
Job id:cb2dd94b861-ea3cb254-9f85-4668-8375-c26fee2c7101
force return result false!
one sibling return false!
msg=3 result=false
one sibling return false!
msg=1 result=false
msg=2 result=false
```

**Figure 7.16**

One of the XML file validation has failed, and Ignite cancels the rest of the processing and returns the result. The result is negative in this case. This is very simple and straightforward approach of Apache Ignite to synchronize the state between jobs. In the next sub-section, we are going to explain the fault tolerance feature of the compute jobs.

## Fault tolerance and checkpointing

In a large-scale grid system, the probability of a job failure is much greater than the traditional parallel system. Therefore, fault tolerance has become a crucial area in grid computing. Ignite supports automatic job failover. In the case of a node crash, jobs are automatically transferred to other available nodes for re-execution. Designing of a fault tolerance system in a grid environment with optimized resource utilization and execution time is a critical and challenging task. A good fault tolerant job scheduling approach should be able to handle not only the complexity of the resources but also various faults occurring during the job execution.

For a long-running computationally intensive application like compressing a very large file, parsing tons of messages may require hours or even days to carry out the execution which they are prone to various types of fails. Different types of faults, classified based on several factors are mentioned below:

- Physical server faults: fault in CPU/memory/disk
- Network faults: fault due to network partition, packet loss.
- Processor faults: Operating system faults.
- Lifecycle faults: Legacy or versioning faults.
- Process faults: software bug, resource shortage.

There are many conditions that may result in failure within your application, and you can trigger a failover. Moreover, you have the ability to choose to which node a job should be failed over to, as it could be different for different applications or different computations within the same application. A high-level view of the fault tolerance support in Ignite is shown below:

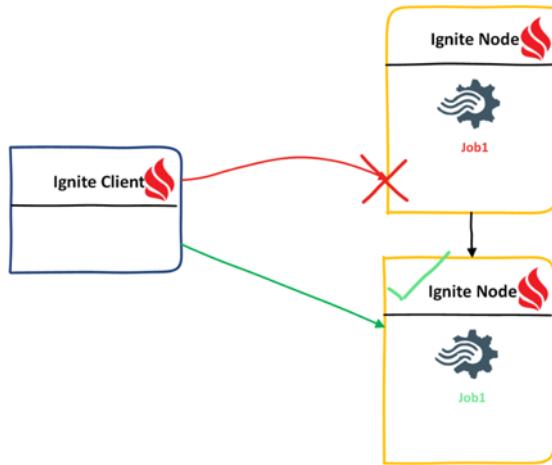


Figure 7.17

A checkpoint is important to take into account when designing fault tolerance system. It's very crucial to recover the job from the state, where it failed. Checkpointing is a technique which allows a process to preserve its state during an arbitrary time interval and resuming its normal operation to reduce faults during recovery process after failure. Note that, to improve the reliability and system availability in Grid computing, a commonly used fault tolerance technique is checkpointing.

However, in Ignite, checkpointing is *optional*; the default checkpointing is disabled for performance reasons. Without checkpointing, Ignite will restart the job in another node from the beginning. Job fault tolerance in Ignite with checkpoint shown in figure 7.18.

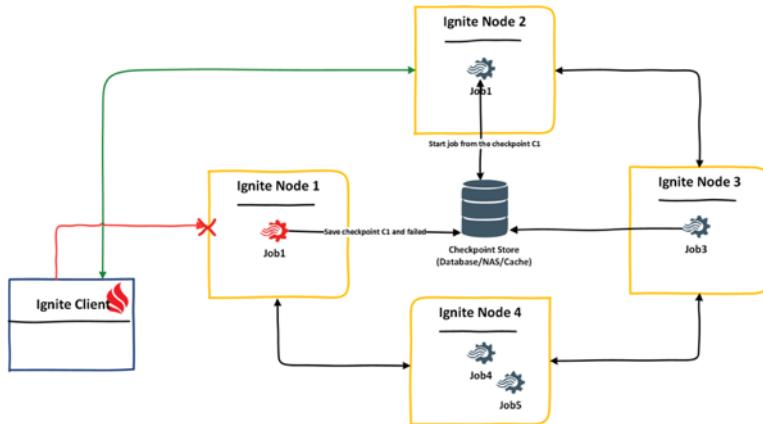


Figure 7.18

When a checkpoint is enabled, every job saves their state in checkpoint store such as Database, NAS or Ignite cache. When one of the jobs failed with an error, Apache Ignite restarts the same job in another node and retrieve the checkpoint from the store to start the job from this checkpoint. In the above figure, *Job1* save the state *C1* into the store and failed, Apache Ignite restarts the *Job1* in the *Node 2*, and start processing from the previously saved state *C1*.

Apache Ignite provides `FailoverSpi`, which is responsible for handling the selection of a new node for the execution of a failed job. Moreover, `FailoverSpi` interface offers developers the possibility to supply custom logic for handling failed execution of a grid job. In all cases of failure, `FailoverSpi` takes failed job (as failover context) and the list of all grid nodes and provides another node on which the job execution will be retried. It is up to failover SPI to make sure that job is not mapped to the node it failed on. Failover is triggered when the method `ComputeTask.result(...)` returns the `ComputeJobResultPolicy.FAILOVER` policy. Ignite comes with the following built-in failover SPI implementations:

- *NeverFailoverSpi* – failover SPI implementation that never fails over.
- *AlwaysFailoverSpi* - failover SPI that always reroutes a failed job to another node. This is the Ignite failover **default mode**.
- *JobStealingCollisionSpi* - Collision SPI that supports job stealing from over-utilized nodes to under-utilized nodes. This SPI is especially useful if you have some jobs within task complete fast, and others sitting in the waiting queue on slower nodes. In such case, the waiting jobs will be stolen from the slower node and moved to the fast under-utilized node.

- *JobStealingFailoverSpi* - job stealing failover SPI needs always to be used in conjunction with *JobStealingCollisionSpi* SPI. When *JobStealingCollisionSpi* receives a steal request and rejects jobs so they can be routed to the appropriate node. It is the responsibility of this *JobStealingFailoverSpi* SPI to make sure that the job is indeed re-routed to the node that has sent the initial request to steal it.



## Info

*AlwaysFailoverSpi* is enabled by default. In this mode, an attempt will be made to reroute the failed job to a node that was not part of initial split for a better chance of success. If no such nodes are available, then an attempt will be made to reroute the failed job to the nodes in the initial split minus the node the job has failed on. If none of the above attempts succeeded, then the job will not be failed over and null will be returned.

Ignite guarantee that, as long as there is at least one node standing, no job will ever be lost. To set the maximum number of attempts to execute a failed task on another node, use the following setter method:

Method name	Interface name
<code>setMaximumFailoverAttempts(int maxFailoverAttempts)</code>	<code>AlwaysFailoverSpi</code> interface
<code>setMaximumStealingAttempts(int maxStealingAttempts)</code>	<code>JobStealingCollisionSpi</code> interface

Let's get back to the checkpoint feature again. Checkpoints are available through the following methods on `ComputeTaskSession` interface:

- **saveCheckpoint(String key, Object state)** - saves intermediate state of a job or task to a storage. This way whenever a job fails over to another node, it can load its previously saved state via `loadCheckpoint(String)` method and continue with execution. Parameters:
  1. key - Key to be used to load this checkpoint in future.
  2. state - Intermediate job state to save. This method defaults checkpoint scope to `ComputeTaskSessionScope.SESSION_SCOPE`, and implementation will automatically remove the checkpoint at the end of the session.
- **saveCheckpoint(String key, Object state, ComputeTaskSessionScope scope, long timeout)** - the overloaded method of the `saveCheckpoint()` method. The life time of the checkpoint is determined by its timeout and scope. If `ComputeTaskSessionScope.GLOBAL_SCOPE` is used, the checkpoint will outlive its session, and can only be removed by calling `checkpointSpi.removeCheckpoint(String)` from Ignite or another task or job. Parameters:

1. key - Key to be used to load this checkpoint in future.
  2. state - Intermediate job state to save.
  3. scope - Checkpoint scope. If equal to `ComputeTaskSessionScope.SESSION_SCOPE`, then state will automatically be removed at the end of task execution. Otherwise, if scope is `ComputeTaskSessionScope.GLOBAL_SCOPE` then state will outlive its session and can be removed by calling `removeCheckpoint(String)` from another task or whenever timeout expires.
  4. timeout - maximum time this state should be kept by the underlying storage. Value 0 means that timeout will never expire.
- **saveCheckpoint(String key, Object state, ComputeTaskSessionScope scope, long timeout, boolean overwrite)** - the overloaded method of the `saveCheckpoint()` method. This method contains another extra parameter: overwrite, which determines, whether or not a checkpoint will be overwritten if it already exists.
  - **loadCheckpoint(String key)** - loads job's state previously saved via `saveCheckpoint(...)` method from an underlying storage for a given key. If the state was not previously saved, then null will be returned. Parameters:
    1. key - Key for intermediate job state to load.
  - **removeCheckpoint(String key)** - removes previously saved job's state for a given key from an underlying storage. Parameters:
    1. key - Key for intermediate job state to remove from the session.



## Warning

There is a mistake on Apache Ignite documentation, Checkpoints are available on interface `ComputeTaskSession` rather than `GridTaskSession`.

There are a few pre-defined Checkpoints implementations delivered by the Apache Ignite. Some of them are detailed below:

1. **NoopCheckpointSpi** – it's the default implementation of the `CheckpointSPI`, if nothing is specified, no-op checkpoint SPI is used.
2. **CacheCheckpointSpi** – this is the cache-based implementation for checkpoint SPI.
3. **JdbcCheckpointSpi** – this is the RDBMS-based implementation for checkpoint SPI, this implementation uses a database to store checkpoints.

4. **SharedFsCheckpointSpi** – this provides the shared file system CheckpointSPI implementation for the checkpoint SPI. All checkpoints are stored on shared storage and available for all nodes in the grid. Note that every node must have access to the shared directory. The reason the directory needs to be shared is because a job state can be saved on one node and loaded on another. Shared storage would be NAS or SAN file shared system. For high-performance SAN is preferable.
5. **S3CheckpointSpi** – this is the amazon S3-based implementation for checkpoint SPI.

*CheckpointSpi* is provided in *IgniteConfiguration* class and passed into Ignition class at startup. Here is an example of using *CacheCheckpointSPI* in Java on Ignite node startup with failover:

**Listing 7.30**

---

```
IgniteConfiguration cfg = new IgniteConfiguration();
//cache checkpointSPI implementation.
CacheCheckpointSpi cacheCheckpointSpi = new CacheCheckpointSpi();
cacheCheckpointSpi.setCacheName("checkpointCache");
// set the check point spi and failover spi in Ignite configuration
cfg.setCheckpointSpi(cacheCheckpointSpi).setFailoverSpi(new AlwaysFailoverSpi());
// Starts Ignite node.
Ignition.start(cfg);
```

---

Now that, we got an idea about the Ignite checkpoint and failover mechanism, let's build something useful. We are going to extend our XML validation application to be failover across the Ignite cluster. Assuming, we have got the following *non-functional* requirements from the system analyst:

1. Every job must save their state (checkpoint) after completing the XSD validation phase into the Ignite cache.
2. A new job will be executed in another Ignite node and load the state from the cache and continue execution if any of the jobs failed after completing the phase (*XML validation*).

Figure 7.19 illustrates the fail over process flow.

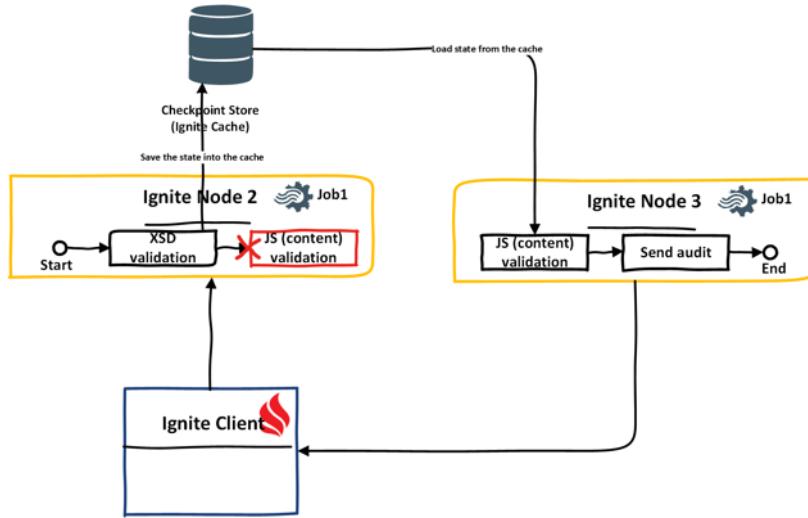


Figure 7.19

To implementation the fail-over through a checkpoint, we are going modify our previous application. If you are curious about the code, it's available at [GitHub repositories<sup>140</sup>](#).

Here, we have two main class: *ForkJoinWithCheckpointComputation* and *ForkJoinWithCheckpointJobAdapter*. *ForkJoinWithCheckpointComputation* is very similar to the previous example. *ForkJoinWithCheckpointComputation* is a compute task split adapter which implements the split and reduce methods for task splitting as well as reduce the overall result. Although, we have configured the cache checkpoint SPI to store the checkpoint into the cache and also explicitly configured the *AlwaysFailoverSpi* as follows:

Listing 7.31

---

```

try (Ignite ignite = Ignition.start(CLIENT_CONFIG)) {
    IgniteCompute compute = ignite.compute();
    CacheConfiguration cacheConfiguration = new CacheConfiguration("checkpoints");
    // explicitly uses of checkpoint
    CacheCheckpointSpi cacheCheckpointSpi = new CacheCheckpointSpi();
    cacheCheckpointSpi.setCacheName("checkpointCache");
    ignite.configuration().setCheckpointSpi(cacheCheckpointSpi)
        .setFailoverSpi(new AlwaysFailoverSpi());
    // create or get cache
    ignite.getOrCreateCache(cacheConfiguration);
}

```

---

<sup>140</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example6>



## Info

Checkpointing is disabled by *default* for performance reasons. Add `@ComputeTaskSessionFullSupport` annotation to the task to enable it.

Let's have a look at the next class: `ForkJoinWithCheckpointJobAdapter`. Most of the part of this class is very much similar to what we have done so far in this chapter.

**Listing 7.32**

```
public class ForkJoinWithCheckpointJobAdapter extends ComputeJobAdapter {
    @TaskSessionResource
    private ComputeTaskSession session;

    @JobContextResource
    private ComputeJobContext jobCtx;

    @IgniteInstanceResource
    private Ignite ignite;

    private ValidateMessage msg;

    public ForkJoinWithCheckpointJobAdapter(ValidateMessage msg) {
        this.msg = msg;
    }

    @Override
    public Boolean execute() throws IgniteException {
        try {
            Boolean validateXsdResult = session.loadCheckpoint(msg.getId());

            Boolean checkpointValue = validateXsdResult;
            System.out.println("validateXsdResult=" + validateXsdResult + " msg=" + msg.get
etId());
            if (validateXsdResult == null) {
                validateXsdResult = XsdValidator.validate(msg.getMsg(), msg.getXsd());
                session.setAttribute(jobCtx.getJobId(), validateXsdResult);
                session.saveCheckpoint(msg.getId(), true);
            }
            if (msg.getId().equals("1") && checkpointValue == null) {
                //emulate error
                System.out.println("throw err!!!");
                throw new ComputeJobFailoverException("err!!!");
            }
        }
    }
}
```

```
}

System.out.println("real execute msg=" + msg.getId());

if (!validateXsdResult) {
    System.out.println("force return result false!");
    return sendResultAndReturn(validateXsdResult);
}

for (ComputeJobSibling sibling : session.getJobSiblings()) {
    Boolean siblingStep1Result = session.waitForAttribute(sibling.getId(), \
0);
    if (!siblingStep1Result) {
        System.out.println("one sibling return false!");
        return sendResultAndReturn(false);
    }
}

boolean validateByJs = JSEvaluate.evaluateJs(msg.getId(), msg.getJs());
return sendResultAndReturn(validateByJs);

} catch (Exception err) {
    throw new IgniteException(err);
}
}

@NotNull
private Boolean sendResultAndReturn(Boolean result) throws URISyntaxException, IOExce\
ption {
    ConcurrentMap<Object, Object> nodeLocalMap = ignite.cluster().nodeLocalMap();
    HttpClient client = HttpAuditClient.createHttpClient(nodeLocalMap);
    return HttpAuditClient.sendResult(client, result, msg.getId());
}
```

---

Let's go through the `execute()` method of the above class. With the next pseudo codes, we are trying to load the checkpoint by *message id* if available.

```
Boolean validateXsdResult = session.loadCheckpoint(msg.getId());
Boolean checkpointValue = validateXsdResult;
System.out.println("validateXsdResult=" + validateXsdResult + " msg=" + msg.getId());
if (validateXsdResult == null) {
    validateXsdResult = XsdValidator.validate(msg.getMsg(), msg.getXsd());
    session.setAttribute(jobCtx.getJobId(), validateXsdResult);
    session.saveCheckpoint(msg.getId(), true);
}
```

If the `validateXsdResult=null`, we save the checkpoint with the *message id*. Also, note that we have set the job id and the result of the validation as a *session* attribute. Later, we will use this session attributes to check the entire job result. In the next few lines of code, we are trying to emulate the network error to interrupt the execution.

```
if (msg.getId().equals("1") && checkpointValue == null) {
    //emulate error
    System.out.println("throw err!!!");
    throw new ComputeJobFailoverException("err!!!");
}
```

The above code is very simple, for the message with id 1, and whenever there is no checkpoint available for this message, we throw compute job failover exception. This condition will execute only one time to emulate an error during job processing. The rest of the part of this method is as same as before. Let's run the application to ensure the fault tolerance of the application. First, start the HTTP server as follows (if it is not already started).

---

**Listing 7.33**

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.common.HttpAuditEmulator
```

---

Then, start the a few Ignite node with the following command (I am going to start 4 nodes):

---

**Listing 7.34**

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---



## Tip

You have to run more than one Ignite node for getting fault tolerance of an Ignite job. Because, if you run only one Ignite node, and whenever one of the jobs fails, Ignite couldn't re-execute the job on another node. If you run the application on single Ignite node, probably, you would get the following exception in console `org.apache.ignite.cluster.ClusterTopologyException: Failed to failover a job to another node (failover SPI returned null)`.

Now, run the application with the following command:

**Listing 7.35**

---

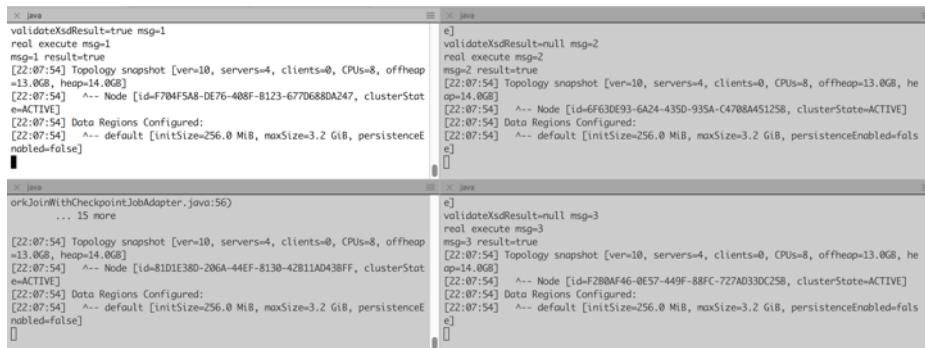
```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example6.ForkJoinWithCheckpointComputation
```

---

The entire result would be **true** as follows:

**final result=true**

Let's explain what's happened under the hood.



The figure shows three terminal windows, each with a title bar containing 'X: java'. The windows are arranged horizontally. Each window displays a Java stack trace. The top window shows a call to validateXsdResult with msg=1. The middle window shows a call to validateXsdResult with msg=2. The bottom window shows a call to validateXsdResult with msg=3. All three windows include the same code snippet from ForkJoinWithCheckpointJobAdapter.java:56, which includes a check for 'msg>1' and a recursive call to validateXsdResult with msg+1. The stack traces also show topology snapshots and data region configurations for Ignite nodes.

```

validateXsdResult=true msg=1
real execute msg=1
msg1 result=true
[22:07:54] Topology snapshot [ver=10, servers=4, clients=0, CPUs=8, offheap=13.0GB, heap=14.0GB]
[22:07:54]   ^-- Node [id=F704F5A8-DE76-408F-B123-677D688DA247, clusterState=ACTIVE]
[22:07:54] Data Regions Configured:
[22:07:54]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
e]

validateXsdResult=null msg=2
real execute msg=2
msg2 result=true
[22:07:54] Topology snapshot [ver=10, servers=4, clients=0, CPUs=8, offheap=13.0GB, heap=14.0GB]
[22:07:54]   ^-- Node [id=6F63DE93-6A24-435D-935A-C4708A451258, clusterState=ACTIVE]
[22:07:54] Data Regions Configured:
[22:07:54]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
e]

validateXsdResult=null msg=3
real execute msg=3
msg3 result=true
[22:07:54] Topology snapshot [ver=10, servers=4, clients=0, CPUs=8, offheap=13.0GB, heap=14.0GB]
[22:07:54]   ^-- Node [id=F2B80AF46-0E57-449F-88FC-727AD330C258, clusterState=ACTIVE]
[22:07:54] Data Regions Configured:
[22:07:54]   ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
e]

```

**Figure 7.20**

We have started 4 Ignite nodes for clarity. When we execute our application, it generated 3 jobs for each XML file. A job with the XML message of id *one* completed the XSD validation and failed. Ignite re-executed the job in the 4<sup>th</sup> node (upper right corner of figure 7.20) and completed the process.

One important use case for the checkpoint is not guarding against a failure of the master node; the node that started the original execution. When a master node fails, Ignite doesn't have any idea where to send the results of the job execution to, and thus the result will be discarded. To failover this scenario, one can store the final result of the job execution as a checkpoint and re-run the logic of the entire task in case of a node failure. In such a case, the re-run process of the task will be much faster since all the jobs can start from the saved checkpoints. We are going to study the compute grid affinity call (collocate the computation with the data) in the next section.

## Collocation of computation and data

Let's start with the definition of the collocation of data. The idea behind the data collocation is to keep all the related datasets in a single node. Datasets can be allocated into the different caches in the same node. With this approach, network roundtrip for the related data is decreasing, and the client application can retrieve the associated data from a single node. Composite columns of Cassandra database is an example of the collocation data technics.

As we mentioned before, Apache Ignite provides `@AffinityKeyMapped` annotation for key-to-node affinity. Affinity key is a key which will be used to determine a node on which given cache key will be stored. This annotation allows marking a field or a method in the cache key object that will be used as an affinity key (instead of the entire cache key object that is used for affinity by default). Note that a class can have only one field or method annotated with `@AffinityKeyMapped` annotation.

For instance, in bank account-engine, we have followed by two entities: account and transactions. Account entity contains all the information about the client, and the transaction entity holds all the transactions entries of a given client. If these two entities are always accessed together, then for better performance and scalability it makes sense to collocate entity transactions with their account entity when storing them into the cache. To accomplished this, we can provide a common field of these two entities as an affinity key. Figure 7.21 illustrates the graphical view of the affinity key. Note that we have some data duplication (account number) on each dataset.

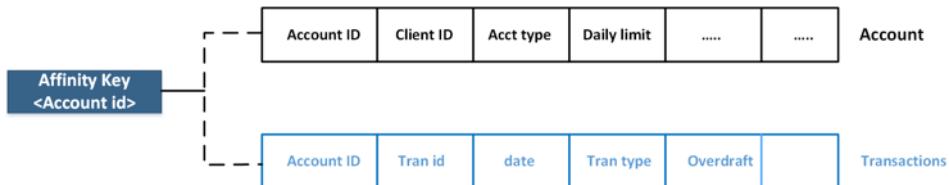


Figure 7.21

One of the main use cases for this `@AffinityKeyMapped` annotation is the routing of grid computations to the nodes where the data for this computation is cached. The concept is well known as Collocation of Computations and Data in Apache Ignite.

Apache Ignite provides two methods for executing a job on the node where data for provided affinity key is located: `affinityCall`, `affinityRun`. In other words, knowing a cache name and the affinity key, these methods will be able to find the node that is the primary for the given key and will execute a job there. The difference between the two methods is that `affinityRun` doesn't return any result. The two methods have the following signatures:

Method name	Description
cacheName	Name of the cache to use for affinity co-location.
affKey	Affinity key.
job	A job which will be co-located on the node with given affinity key.
Job result	Return a value of the job, only affinityCall() function can return value.

Now that we have dipped our toes into theory, let's build a (nearly) minimal example of an affinity co-location. To keep things simple, we will use the account and transactions entity as described before. We are going to build an application to calculate the Cashback for a given client. We will execute an affinity call function for each client, which will return the cash back amount for the client based on his transactions history. Let's start building the application from the scratch. The full source code is available on [GitHub<sup>141</sup>](#).

### Step 1.

Let's start with the `TransacionKey` class (this class is reside on package `common` at [GitHub<sup>142</sup>](#)) as follows:

Listing 7.36

---

```
public class TransactionKey implements AccountCacheKey, Serializable {
    @QuerySqlField(index = true)
    @AffinityKeyMapped
    private String account;
    private Date transactionDate;
    private String transactionId;
    public TransactionKey() {
    }
    ...
}
```

---

We annotate the field named `account` with the `@AffinityKeyMapped` annotation. This annotation specifies the affinity of the *transaction entity* with the *account number*. Note that, we also use the `@QuerySqlField` annotation to use this given field for querying through SQL. Create another Java class with name `AccountKey`, and add the following contents:

---

<sup>141</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example8>

<sup>142</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/common>

**Listing 7.37**

---

```
public class AccountKey implements AccountCacheKey, Serializable {
    @AffinityKeyMapped
    private String account;
    public AccountKey(String account) {
        this.account = account;
    }

    public String getAccount() {
        return account;
    }
    ...
}
```

---

Field *account* also mapped as an affinity key for the *AccountKey* class.

**Step 2.** Create another Java class for the *Transactions* data as follows:

**Listing 7.38**

---

```
public class TransactionData implements AccountCacheData {
    private String fromAccount;
    private String toAccount;
    private BigDecimal sum;
    private String transactionType;

    public TransactionData() {
    }
    ...
}
```

---

We will use the above Java class to create transaction dataset and store them into the cache. Appropriately, we have a Java class for storing account information as follows:

**Listing 7.38**

---

```
public class AccountData implements AccountCacheData {  
    private String account;  
    private String personId;  
    private String accountType;  
    private Date openedAt;  
  
    private BigDecimal dailyLimit;  
    private BigDecimal todayOperationSum;  
    private LocalDate today;  
    ....  
}
```

---

**Step 3.** We also need a utility class to generate some dataset for the application. Create a new Java class named `BankDataGenerator`, and add the following static methods to generate some test data.

**Listing 7.39**

---

```
public class BankDataGenerator {  
  
    public static final String ACCOUNT_CACHE = "accountCache";  
    public static final String SAVINGS_CACHE = "savingsDictionaryCache";  
    public static final String TEST_ACCOUNT = "0000*1111";  
  
    public static IgniteCache<AccountCacheKey, AccountCacheData> createBankCache(Ignite ignite) {  
        CacheConfiguration accountCacheCfg = new CacheConfiguration()  
            . setName(ACCOUNT_CACHE)  
            . setAtomicityMode(TRANSACTIONAL)  
            . setIndexedTypes(  
                AccountKey.class, AccountData.class,  
                TransactionKey.class, TransactionData.class,  
                String.class, CashBackDictionaryData.class  
            );  
  
        IgniteCache<AccountCacheKey, AccountCacheData> result = ignite.getOrCreateCache(a  
ccountCacheCfg);  
        result.removeAll();  
        return initData(result);  
    }  
    public static IgniteCache<String, CashBackDictionaryData> initSavigsCache(Ignite igni  
te) {  
        CacheConfiguration savingsCacheCfg = new CacheConfiguration()  
            . setName(SAVINGS_CACHE)  
            . setAtomicityMode(TRANSACTIONAL)  
            . setIndexedTypes(String.class, CashBackDictionaryData.class);  
        IgniteCache<String, CashBackDictionaryData> result = ignite.getOrCreateCache(s  
avingsCacheCfg);  
        result.removeAll();  
        return initData(result);  
    }  
}
```

---

```

te) {
    CacheConfiguration savingsCacheCfg = new CacheConfiguration(). setName(SAVINGS_CACHE);
    IgniteCache<String, CashBackDictionaryData> result = ignite.getOrCreateCache(savingsCacheCfg);
    result.removeAll();
    result.put("meal", new CashBackDictionaryData(new BigDecimal(0.01), "meal shopping"));
    result.put("entertainment", new CashBackDictionaryData(new BigDecimal(0.02), "entertainment"));
    return result;
}

private static IgniteCache<AccountCacheKey, AccountCacheData> initData(IgniteCache<AccountCacheKey, AccountCacheData> result) {
    //init data
    result.put(new AccountKey(TEST_ACCOUNT), new AccountData(TEST_ACCOUNT, "John Doe", "standard", new Date(), new BigDecimal(100)));
    result.put(new AccountKey(TEST_ACCOUNT), new AccountData(TEST_ACCOUNT, "Mr. Smith", "premium", new Date(), new BigDecimal(200)));

    result.put(
        new TransactionKey(TEST_ACCOUNT, new Date(), UUID.randomUUID().toString()),
        new TransactionData(TEST_ACCOUNT, "1111*2222", new BigDecimal(100), "meal"));
}

result.put(
    new TransactionKey(TEST_ACCOUNT, new Date(), UUID.randomUUID().toString()),
    new TransactionData(TEST_ACCOUNT, "3333*4444", new BigDecimal(100), "entertainment"));
};

return result;
}
}

```

---

Static method `createBankCache` initialize the cache named `accountCache`, and generate some test data for the account with transactions. Account number we are going to use for the test is `0000*1111`. We have another static method named `initSavingsCache`, which initialize another cache with name `savingsDictionaryCache` and filled some data for `cashback` percent.

**Step 4.** Now, it is time to execute some `affinityCall()` function in the Ignite node. Create a new Java class with a `main()` method. In my case, the class name is `TestAccountSavingsMain`. Add the following contents to the `main()` method:

Listing 7.40

```
public static void main(String[] args) {

    try (Ignite ignite = Ignition.start(CommonConstants.CLIENT_CONFIG)) {
        IgniteCompute compute = ignite.compute();

        IgniteCache<AccountCacheKey, AccountCacheData> cache = BankDataGenerator.createBankCache(ignite);
        IgniteCache<String, CashBackDictionaryData> savingsCache = BankDataGenerator.initSavingsCache(ignite);

        SqlFieldsQuery sql = new SqlFieldsQuery("select * from TransactionData where account = ?");
        BigDecimal result = compute.affinityCall(BankDataGenerator.ACCOUNT_CACHE, new AccountKey(BankDataGenerator.TEST_ACCOUNT), () -> {
            List<List<?>> data = cache.query(sql.setArgs(BankDataGenerator.TEST_ACCOUNT)).getAll();
            BigDecimal cashBack = new BigDecimal(0);
            for (List row : data) {
                TransactionData tr = (TransactionData) row.get(1);
                CashBackDictionaryData cashBackDictionaryData = savingsCache.get(tr.getTransactionType());
                cashBack = cashBack.add(tr.getSum()).multiply(cashBackDictionaryData.getCashBackPercent());
            }
            //System.out.println("savings="+cashBack);
            return cashBack;
        });

        System.out.println("CashBack="+result);
    }
}
```

Let's go through the codes. First, we start our application as an Ignite client. We initialized the caches and generated some data for the `affinityCall` function in the next two lines of code. We use a simple SQL query to retrieve all transactions for the account **0000\*1111**.

```
SqlFieldsQuery sql = new SqlFieldsQuery("select * from TransactionData where account = ?"\n);
```

In the next few lines of codes, we executed the `affinityCall()` function, where we provide the cache name as `accountCache`, affinity key as `0000*1111` (account number), and the job itself. In the execution block of the job, we executed the above SQL query to get all the transaction associated with the account `0000*1111`. Next, we iterate over the transactions data and calculate the cash back for all the transaction of this account number and returns the result. Let's run the application, start an Ignite node with the following command:

**Listing 7.41**

---

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

Then, run the application by the command as follows:

**Listing 7.42**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example8.TestAccountSavingsMain
```

---

The application should return to you the following result on the console:

```
savingsCache=org.apache.ignite.internal.processors.cache.GatewayProtectedCacheProxy@404e8\\n
c33
CashBack=3.0000000000000006245004513516505539882928133010864257812500
```

You can change the execution logic if you want to play with the `affinityCall()` function. From the version 1.8, Ignite provides consistency guaranty of `affinityCall()` function, which means, the partition, which the affinity key belongs to, will not be evicted from a node while a job triggered by `affinityCall(...)` or `affinityRun(...)` is being executed.

## Job scheduling

In Ignite, jobs are submitted to a thread pool of the Ignite JVM and are executed in random order (see the figure 7-22). However, Ignite provides the ability to use the custom logic in determining how jobs should be executed on a destination grid node. Job scheduling can be achieved by the `CollisionSpi` interface.

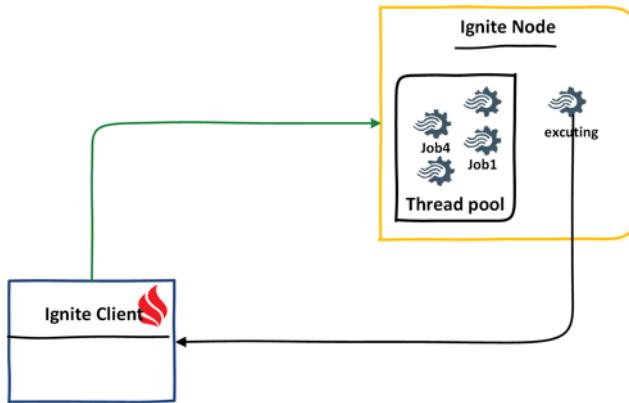


Figure 7.22

Collision SPI allows how grid jobs get executed when they arrive on a destination node for execution. Its functionality is similar to tasks management via customizable GCD (Great Central Dispatch) on Mac OS X as it allows the developer to provide custom job dispatching on a single node. In general, a grid node will have multiple jobs arriving at it for execution and potentially multiple jobs that are already executing or waiting for execution on it. There are multiple possible strategies dealing with this situation: all jobs can proceed in parallel, or jobs can be serialized, i.e., only one job can execute at any given point of time, or certain number or types of grid jobs can proceed in parallel, etc. Apache Ignite comes with the following ready made implementations for collision resolution that cover most popular strategies:

**FifoQueueCollisionSpi:** This class provides an implementation for *Collision SPI* based on FIFO queue. Jobs are ordered as they arrived, and only `getParallelJobsNumber()` number of jobs is allowed to execute in parallel. Other jobs will be buffered in the passive queue.



## Tip

By setting `parallelJobsNumber` to `one`, you can guarantee that all jobs will be executed one-at-a-time, and no two jobs will be executed concurrently.

*FifoQueueCollisionSpi* can be configured by Spring XML or Java. Let's check a simple *FifoQueueCollisionSpi* configuration by Spring XML.

**Listing 7.43**

---

```
<bean class="org.apache.ignite.IgniteConfiguration" singleton="true">
    ...
    <property name="collisionSpi">
        <bean class="org.apache.ignite.spi.collision.fifoqueue.FifoQueueCollisionSpi">
            <!-- Execute one job at a time. -->
            <property name="parallelJobsNumber" value="1"/>
        </bean>
    </property>
    ...
</bean>
```

---

**PriorityQueueCollisionSpi:** This class provides an implementation for Collision SPI based on *priority queue*. Jobs are first ordered by their priority if one is specified, and only first `getParallelJobsNumber()` jobs are allowed to execute in parallel. Other jobs will be queued up. This SPI has the following optional configuration parameters:

1. A Number of jobs that can be executed in parallel (`setParallelJobsNumber(int)`). This number should usually be set to not greater than the number of threads in the execution thread pool.
2. Priority attribute session key (`getPriorityAttributeKey()`). Prior to returning from `ComputeTask.map(List, Object)` method, task implementation should set a value into the task session keyed by this attribute key.
3. Priority attribute job context key (`getJobPriorityAttributeKey()`). It is used for specifying job priority.
4. Default priority value (`getDefaultValue()`). It is used when no priority is set.
5. Default priority increase value (`getStarvationIncrement()`). It is used for increasing priority when job gets bumped down. This future is used for preventing starvation waiting for jobs execution.

Here is a Spring XML configuration example:

**Listing 7.44**

```
<property name="collisionSpi">
<bean class="org.apache.ignite.spi.collision.priorityqueue.PriorityQueueCollisionSpi">
    <property name="priorityAttributeKey" value="myPriorityAttributeKey"/>
    <property name="parallelJobsNumber" value="10"/>
</bean>
</property>
```

**JobStealingCollisionSpi:** Collision SPI that supports job stealing from over-utilized nodes to under-utilized nodes. This SPI is especially useful if you have some jobs within task complete fast, and others sitting in the waiting queue on slower nodes. In such case, the waiting jobs will be stolen from the slower node and moved to the fast under-utilized node. The design and ideas for this SPI are significantly influenced by Java *Fork/Join* Framework authored by Doug Lea and planned for Java 7. GridJobStealingCollisionSpi took similar concepts and applied them to the grid (as opposed to within VM support planned in Java 7). Quite often grids are deployed across many computers some of which will always be more powerful than others. This SPI helps you avoid jobs being stuck at a slower node, as they will be stolen by a faster node.

This is the end of the section *compute grid*. Next, we are going to explore one of the amazing features of Apache Ignite: Service grid.

## Service Grid

You won't find a clear description of what is service grid and how does it differ from a collection of services. In the enterprise application world, a service is *a standards-based way of encapsulating enterprise functionality and exposing it as a reusable component that can be combined with other services to meet new requirements* and provides the following set of ideas:

1. Service described by an interface and provides several methods.
2. The service may have state (stateful service) or can be stateless too.
3. The service has a life cycle.

From the Apache Ignite's point of view, a *service grid* is a collection of Ignite nodes, where you can deploy arbitrary user-defined reusable distributed services. You can imagine services in Apache Ignite as a precompiled execution block deployed in the Ignite cluster. You can

use service proxy (client) for accessing remotely deployed distributed services. A classic example of this kind of services would be a service that can return the last 20 transactions for displaying in the client transaction history page of an internet bank or a service that can check the cash withdrawal limit at ATM for a day. Figure 7.23 illustrates a high-level view of the service grid in the Ignite cluster.

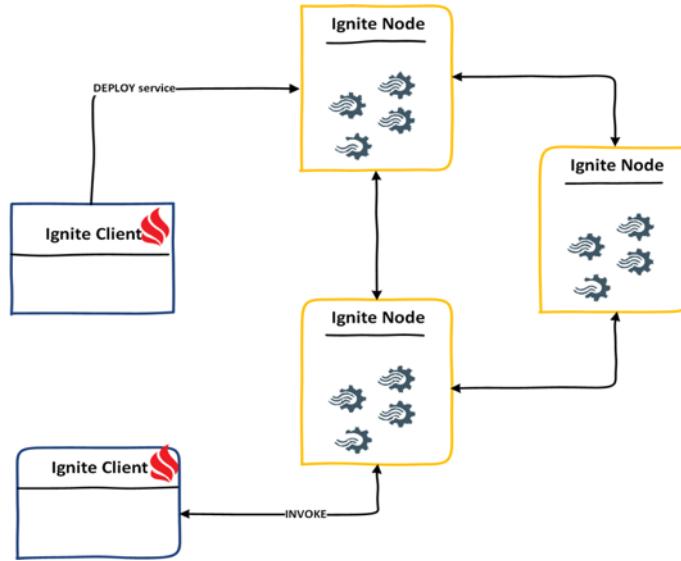


Figure 7.23

Ignite service grid provides the following features:

1. Continuous availability of deployed services regardless of topology changes or crashes.
2. Automatically deploy distributed services on node startup by specifying them in the configuration.
3. Automatically deploy singletons, including cluster-singleton, node-singleton, or key-affinity-singleton.
4. Automatically deploy any number of distributed service instances in the cluster.
5. Undeploy any of the deployed services.
6. Get information about service deployment topology within the cluster.
7. Develop and deploy on-demand *microservices* in distributed fashion.
8. Create service proxy for accessing remotely deployed distributed services.
9. Create a high-level wrapper (REST/WS) for service proxy for accessing remotely deployed distributed services.

**Disadvantages.** Just because something is new, doesn't mean it has no drawbacks. Here's a list of some potential drawbacks associated with the Apache Ignite service grid:

1. The developer should spend the time in writing boilerplate infrastructure code to deploy and manage services in Ignite grid.
2. CPU and RAM intensive services could crash the entire Ignite cluster. If you are planning to deploy services that consume a lot of RAM (e.g. text analysis, image processing, trained model), you should reconsider of using Apache Ignite service grid.
3. Peer class loading is not supported for service grid. It's required to have a Service class in the classpath of all the cluster nodes.

## Developing services

Apache Ignite provides two plain Java interfaces to develop and deploy services in Ignite.

**Interface Service:** an instance of grid-managed service. Whenever a service is deployed, Apache Ignite will automatically calculate how many instances of this service should be deployed on each node within the cluster. For developing service, you have to implement the interface Service and implements several methods as follows:

- method `init(...)`: Pre-initializes service before execution. This method is guaranteed to be called before service deployment is complete. In this method, we can initialize resources such as connection to any Database or 3<sup>rd</sup> party services.
- method `execute(...)`: Starts execution of this service. This method is automatically invoked whenever an instance of the service is deployed on a grid node. Note that service is considered deployed even after it exits the execute method and can be canceled (or undeployed) only by calling any of the cancel methods on IgniteServices API.
- method `cancel(...)`: Cancels this service. Ignite will automatically call this method whenever any of the cancel methods on IgniteServices API are called.

**Interface IgniteServices:** Provides functionality necessary to deploy distributed services on the Ignite grid. Java classes that implement interface Service can be deployed from IgniteServices façade. Interface IgniteServices defines a few override methods to deploy services in the grid.

- Method `deploy()`: Deploys multiple instances of the service on the grid according to provided configuration. Ignite will deploy a maximum amount of services equal to `cfg.getTotalCount()` [defines the maximum number of the instance] parameter making sure

that there are no more than `cfg.getMaxPerNodeCount()` [defines the maximum number of the instance on each node] service instances running on each node. Whenever topology changes, Apache Ignite will automatically rebalance the deployed services within the cluster to make sure that each node will end up with about equal number of deployed instances whenever possible.

- Method `deployClusterSingleton(...)`: Deploys a cluster-wide *singleton* service. Ignite will guarantee that there is always one instance of the service in the cluster. In case the grid node on which the service was deployed crashes or stops, Ignite will automatically redeploy it on another node. However, if the node on which the service is deployed remains in topology, then the service will always be deployed on that node only, regardless of topology changes. Method `deployNodeSingleton(...)`: Deploys a per-node singleton service. Ignite will guarantee that there is always one instance of the service running on each node. Whenever new nodes are started within the underlying cluster group, Ignite will automatically deploy one instance of the service on every new node.
- Method `deployKeyAffinitySingleton(...)`: Deploys one instance of this service on the primary node for a given affinity key. Whenever topology changes and primary node assignment changes, Ignite will always make sure that the service is undeployed on the previous primary node and deployed on the new primary node.
- Method `deployMultiple(...)`: Deploys multiple instances of the service on the grid. Ignite will deploy a maximum amount of services equal to `cfg.getTotalCount()` parameter making sure that there are no more than `cfg.getMaxPerNodeCount()` service instances running on each node.

So far in this chapter, we have created Ignite on-demand compute jobs to validate XML documents. We are going to reuse our previous two phase validation process as a service and deploy it as service on Ignite grid in this section. Whenever we have a need to validate any XML documents, we will create a proxy and invoke the service in Ignite grid to complete the validation. After deployment of the XML validation service in Ignite grid, it will live till undeployed from the Ignite grid. Full source code of this section is available on [GitHub<sup>143</sup>](#). Let's start with the creation of the interface.

**Step 1.** Create the `XsdValidatingService` interface with one method `isOk()`, which has parameter type `ValidateMessage` and can return the validation result. It's our custom interface for our service for validating XML message.

---

<sup>143</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example7>

**Listing 7.45**

---

```
public interface XsdValidatingService {  
    boolean isOk(ValidateMessage msg);  
}
```

---

**Step 2.** Next, create a new Java class with name `XsdValidatingServiceImpl` and implements the next two interfaces: `XsdValidatingService`, and `Service`.

**Listing 7.46**

---

```
public class XsdValidatingServiceImpl implements XsdValidatingService, Service {  
    private CloseableHttpClient auditClient;  
    @Override  
    public boolean isOk(ValidateMessage msg) {  
        Boolean validateXsdResult = XsdValidator.validate(msg.getId(), msg.getXsd());  
        sendResult(msg, validateXsdResult);  
        return validateXsdResult;  
    }  
    private void sendResult(ValidateMessage msg, Boolean validateXsdResult) {  
        try {  
            HttpAuditClient.sendResult(auditClient, validateXsdResult, msg.getId());  
        } catch (Exception err) {  
            err.printStackTrace();  
        }  
    }  
    //service methods  
    @Override  
    public void cancel(ServiceContext serviceContext) {  
        System.out.println("cancel service");  
        try {  
            auditClient.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
    @Override  
    public void init(ServiceContext serviceContext) throws Exception {  
        System.out.println("init service");  
        auditClient = (CloseableHttpClient) HttpAuditClient.createHttpClient();  
    }  
    @Override  
    public void execute(ServiceContext serviceContext) throws Exception {
```

---

```
    }  
}
```

---

Let's go through the above code line by line. Firstly, we have implemented the method `isOk()`, where, we have invoked our `XsdValidator.validate()` method to validated the XML document. Then, we send the result of the validation process for audit and return the result. Later, we override two Ignite services method:

- `Init()`: in this method, we initialize the `auditClient` instance for sending the validation process for audit.
- `Cancel()`: in this method, we released the `auditClient` connection and rest of all resources.

**Step 3.** Now that we have developed our service. Let's deploy it, and run some test to validate XML documents. Create a new Java class with a `main()` method and add the following contents to it.

**Listing 7.47**

```
public class TestXsdValidatingService {  
    private static final String VALIDATING_SERVICE = "validatingService";  
    public static void main(String[] args) throws IOException {  
        try (Ignite ignite = Ignition.start(CommonConstants.CLIENT_CONFIG)) {  
            String sample1 = TestDataGenerator.getSample1();  
            String sample2 = TestDataGenerator.getSample2();  
            byte[] validateSchema = TestDataGenerator.getValidateSchema();  
            String validateScript = TestDataGenerator.getValidateScript();  
            ignite.services().deployNodeSingleton(VALIDATING_SERVICE, new XsdValidatingSe  
rvicImpl());  
  
            XsdValidatingService xsdValidatingService = ignite.services().serviceProxy(VA  
LIDATING_SERVICE, XsdValidatingService.class, /*not-sticky*/false);  
            System.out.println("result=" + xsdValidatingService.isOk(new ValidateMessage(\n  
"1", sample1, validateSchema, validateScript)));  
            System.out.println("result2=" + xsdValidatingService.isOk(new ValidateMessage(\n  
("2", sample2, validateSchema, validateScript)));  
            ignite.services().cancel(VALIDATING_SERVICE);  
        }  
    }  
}
```

---

We are familiar with the first few lines of code from the previous section. We have generated two XML documents, get the XSD schema and the JS validator script from the `TestGenerator` class. Next, we deploy our service through `IgniteService` interface. As a parameter of the interface, we declare the service name and the implementation of our service as follows:

- Service name: `validatingService`
- Service implementation: `XsdValidatingServiceImpl`



## Info

We are using `deployNodeSingleton()` method to deploy our service, it means, there is always only one instance of the service running on each node.

In the next line, we create a *proxy* of the `XsdValidationService` and invoke the service method `isOk()` two times. We also pass the XML document for validating over XSD schema and JS validator during the invocation.



## Tip:

`ignite.services().serviceProxy(...)` method returns a remote handle on the service. If service is available locally, then the local instance is returned, otherwise, a remote proxy is dynamically created and provided for the specified service.

Also, note that we are using a **non-sticky** version of the proxy in our example. If the proxy is sticky, then Ignite will always go back to the same cluster node to contact a remotely deployed service. If the proxy is not sticky, then Ignite will load balance remote service proxy invocations among all cluster nodes on which the service is deployed. There are a few rules that need to be taken into account whenever use a sticky or non-sticky proxy. If you are developing *stateful* services then, you should consider using a sticky proxy to invoke the remote services, otherwise non-sticky proxy is the best choice for you. We cancel the service deployment and release all the resources of the service in the last line of the pseudo-code.

**Step 4.** Let's run the application. First, start an Ignite node with the following command:

**Listing 7.48**

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

Now, start the HTTP server as follows (if it is not already started).

**Listing 7.49**

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.common.HttpAuditEmulator
```

---

Next, run the application by the command as follows:

**Listing 7.50**

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example7.TestXsdValidatingService
```

---

If everything goes fine, you should have the following message on the console:

```
init service
msg=1 result=true
msg=2 result=true
cancel service
```

Very straightforward way to deploy and execute the services in Ignite grid. For simplicity, we use the same class to deploy and execute the service by proxy. However, in a real world, you should use a separate class for deploying and invoking the services as well. In the next section, we are going to study the different ways of deploying Ignite services.

## Cluster singleton

IgniteServices facade allows deploying any number of service instances on the grid automatically. Although, the most commonly used features is to deploy singleton services on the cluster. Ignite proposed automatically deploy singletons, including:

- Node-singleton
- Cluster-singleton
- Key-Affinity-singleton

**Node-singleton.** In the previous section, we used the node-singleton option to deploy our XsdValidation service. In this case, each node contains only one running instance of the service. Ignite will automatically deploy one instance of the service on every new node whenever new nodes are started within the underlying cluster group. Deploy node-singleton is accomplished with the following code:

```
ignite.services().deployNodeSingleton(VALIDATING_SERVICE, new XsdValidatingServiceImpl());
```

Where,

- VALIDATING\_SERVICE – is the name of the service.
- new XsdValidatingServiceImpl() – is the service instance.

The above method is analogous to calling

```
deployMultiple(VALIDATING_SERVICE, new XsdValidatingServiceImpl(), 0, 1)
```

Let's start two Ignite nodes and deploy the XsdValidation service with option node-singleton as follows:

**Listing 7.51**

---

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

Now, deploy the XsdValidation service by executing the Java class *DeployNodeSingleton* (see the *com.blu.imdg.example7* on GitHub for more details).

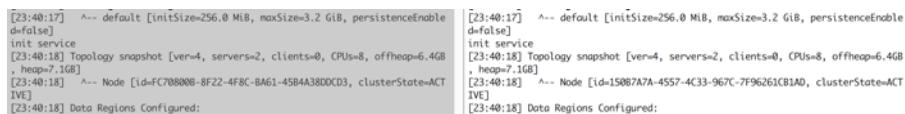
**Listing 7.52**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example7.DeployNodeSingleton
```

---

After the deployment, you should see the following messages on the console.



```
[23:40:17] ^--> default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
init service
[23:40:18] Topology snapshot [ver=4, servers=2, clients=0, CPUs=8, offheap=6.4GB
, heap=7.1GB]
[23:40:18] ^--> Node [Id=FC708008-8F22-4F8C-BA61-45B4A38DDCD3, clusterState=ACTIVE]
[23:40:18] Data Regions Configured:
[23:40:17] ^--> default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
init service
[23:40:18] Topology snapshot [ver=4, servers=2, clients=0, CPUs=8, offheap=6.4GB
, heap=7.1GB]
[23:40:18] ^--> Node [Id=150B7A7A-4557-4C33-967C-7F96261CB1AD, clusterState=ACTIVE]
[23:40:18] Data Regions Configured:
```

**Figure 7.24**

Above figure confirms that each node on the cluster has one instance of the XsdValidation service. If we invoke the remote service via proxy, one of the services will be invoked and returns the validation result. Let's run the Java class *RunXsdValidationService* and see the action as follows:

**Listing 7.53**

---

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.example7.RunXsdValidationService
```

---

If everything goes fine, you should have the following logs into the console.

```
msg=1 result=true  
msg=2 result=true
```

**Cluster-singleton.** A cluster singleton service is a service running on a cluster that is available on only one node of a cluster at a time. In case the grid node on which the service was deployed crashes or stops, Ignite will automatically redeploy it on another node. However, if the node on which the service is deployed remains in topology, then the service will always be deployed on that node only, regardless of topology changes.



## Warning:

In the case of topology changes, there may be a situation when a singleton service instance will be active on more than one node (e.g. crash detection delay) due to network delays.

Deploy cluster-singleton service can be achieved by the following code:

```
ignite.services().deployClusterSingleton(VALIDATING_SERVICE, new XsdValidatingServiceImpl()  
());
```

Where,

- VALIDATING\_SERVICE – is the name of the service, for instance *validatingService-cluster-singleton*.
- new XsdValidatingServiceImpl() – is the service instance.

Let's deploy the XsdValidation service by executing the Java class DeployClusterSingleton (see the example7 on GitHub for more details).

**Listing 7.54**

---

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.example7.DeployClusterSingleton
```

---

After the deployment, you should see the following messages on the console.

```
init service
```

In this time, only one instance of the service is deployed in the entire cluster. What will happen, if the node with the instance of the service crashed? If you force quite one of the Ignite node from the cluster, Ignite service grid will automatically redeploy the service on another node.

**Key-Affinity-singleton.** In this option, you can deploy one instance of this service on the primary node for a given affinity key. Whenever topology changes and primary node assignment changes, Ignite will always make sure that the service is undeployed on the previous primary node and deployed on the new primary node. The deploy option is very similar to the previous option as we have done before.

```
ignite.services().deployKeyAffinitySingleton(VALIDATING_SERVICE, new XsdValidatingService\  
Impl(), "myCache", new MyCacheKey());
```

Where,

- VALIDATING\_SERVICE – is the name of the service, for instance, *validatingService-affinity-singleton*.
- new XsdValidatingServiceImpl() – is the service instance.
- myCache - Name of the cache on which affinity for key should be calculated, null for default cache.
- new MyCacheKey() – affinity cache key.

The service deployment contains a value for the affinity key as well as a cache name (myCache) to which this key belongs to and Ignite service grid will deploy the service on a node that is primary for the given key. If the primary node changes throughout the time then the service will be re-deployed automatically as well.

## Service management and configuration

*IgniteService* interface provides `serviceDescriptors()` method to get metadata about all deployed services. `serviceDescriptors()` method returns an instance of the Service deployment descriptor, which contains all service deployment configuration, and also deployment topology snapshot as well as origin node ID. Ignite service descriptor provides a collection of methods that can return the following deployment configurations:

Method name	Description
<code>name()</code>	Name of the service
<code>serviceClass()</code>	Name of the service class
<code>totalCount()</code>	Maximum allowed total number of deployed services in the grid
<code>maxPerNodeCount()</code>	Maximum allowed number of deployed services on each node
<code>originNodeId()</code>	ID of the grid node that initiated the service deployment
<code>topologySnapshot()</code>	Service deployment topology snapshot. Service topology snapshot is represented by the number of service instances deployed on a node mapped to a node ID
<code>affinityKey()</code>	Affinity key used for key-to-node affinity calculation. This parameter is optional and is set only when key-affinity service was deployed.
<code>cacheName()</code>	Cache name used for key-to-node affinity calculation. This parameter is optional and is set only when key-affinity service was deployed.

To check how the Ignite service management works, create a Java class named `ServiceManagement`, and add the following contents to it:

Listing 55

---

```
public class ServiceManagement {
    public static void main(String[] args) throws IOException {
        try (Ignite ignite = Ignition.start(CommonConstants.CLIENT_CONFIG)) {
            for (ServiceDescriptor serviceDescriptor : ignite.services().serviceDescriptor
                s()) {
                System.out.println("Service Name: " + serviceDescriptor.name());
                System.out.println("MaxPerNode count: " + serviceDescriptor.maxPerNodeCou
                    nt());
                System.out.println("Total count: " + serviceDescriptor.totalCount());
                System.out.println("Service class Name: " + serviceDescriptor.serviceClas
                    s());
                System.out.println("Origin Node ID: " + serviceDescriptor.originNodeId());
            }
        }
    }
}
```

```
}
```

---

In the above class, we used the Ignite *ServiceDescriptor* to printout all the service metadata into the console. Let's execute an Ignite client to quick view the deployment configuration of our deployed services.

#### Listing 56

---

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.example7.ServiceManagements
```

---

See the example7 on [GitHub<sup>144</sup>](#) for the full source code of the Java class. The above command should return the following output on the console:

```
Service Name: validatingService
MaxPerNode count: 1
Total count: 1
Service class Name: class com.blu.imdg.example7.XsdValidatingServiceImpl
Origin Node ID: b7d6d0ef-c89e-4011-905f-bc1ec9347824
```

**Service configuration.** So far in this chapter, we have deployed service after the Ignite node startup. However, you can automatically deploy service on start up of the Ignite node by setting *serviceConfiguration* property of the IgniteConfiguration class. Uncomment the following spring configuration into the file `base-config.xml` (*chapters/chapter-7/src/main/resources/META-INF/org/book/examples/base-config.xml*).

#### Listing 7.57

---

```
<bean class="org.apache.ignite.IgniteConfiguration">
    ...
    <!-- Distributed Service configuration. -->
    <property name="serviceConfiguration">
        <list>
            <bean class="org.apache.ignite.services.ServiceConfiguration">
                <property name="name" value="validatingService"/>
                <property name="maxPerNodeCount" value="1"/>
                <property name="totalCount" value="1"/>
                <property name="service">
                    <ref bean="xsdValidatingServiceImpl"/>
                </property>
            </bean>
        </list>
    </property>
</bean>
```

---

<sup>144</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example7/ServiceManagements>

```
</bean>
</list>
</property>
</bean>
<bean id="xsdValidatingServiceImpl" class="com.blu.imdg.example7.XsdValidatingServiceImpl"
"/>
```

---

In the above spring configuration, we have defined all the necessary service configuration to deploy the service on startup. We set service name, total count, and the service class to deploy the service.



## Info

In this approach to deploy services, service class should be located on Ignite node classpath.

After changing the spring configuration, you have to restart the Ignite node as shown below:

**Listing 7.58**

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

During node startup you should notice that the service startup message (**init service**) on the console as shown in figure 7.28:

```
init service
[00:04:47] Performance suggestions for grid (fix if possible)
[00:04:47] To disable, set -DIGNITE_PERFORMANCE_SUGGESTIONS_DISABLED=true
```

**Figure 7.25**

After the node bootstrap, you can invoke the remote service through service proxy as described earlier in this chapter.

## Developing microservices in Apache Ignite

Microservices is a paradigm of breaking large software (monolith) projects into loosely coupled modules (fine grained), which communicate with each other through simple APIs. Sometimes Microservices are very analogous to Unix Utilities. The concept is the same, just different decade.

Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface - Doug McIlroy.

Unix executable: Does one thing and do it well	Microservice: Does one thing and do it well
--	---

Runs independently from the other commands	Runs independent of other microservice
Produces text-based response	Produces text-based response to clients

From my point of view word microservice misleads itself. In a nutshell, design model under this concept is as follows:

1. Service should be loosely coupled.
2. High cohesion in functionality.
3. Service should change with minimal effect on other services.
4. Automated in deployments.
5. Scaling out easily.
6. Can use a polyglot programming language and polyglot persistence store.

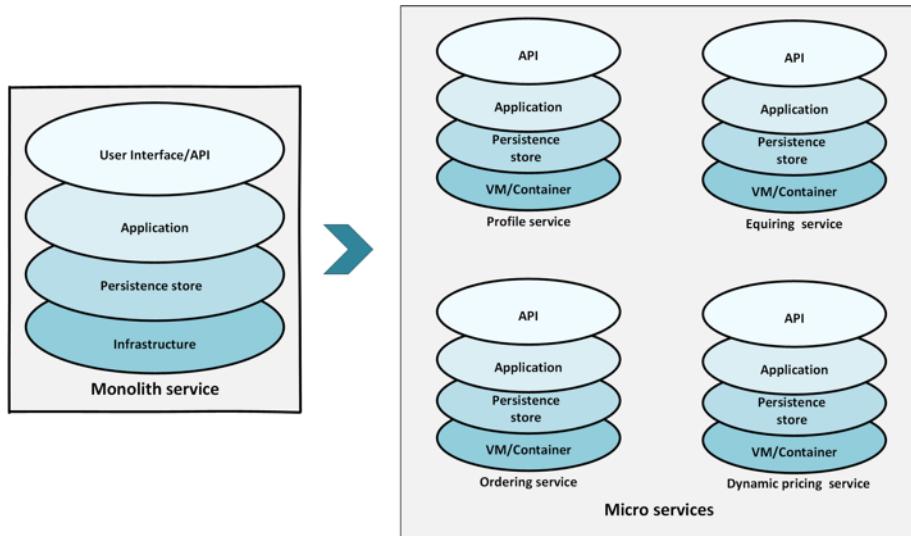


Figure 7.26

Most of these requirements are nothing new at all. Developers or IT Architect who are familiar with SOA already know all the above concepts very well. However, benefits of using

microservices over monolith system are very obviously. You can find the best explanation from Martin Fowler [article<sup>145</sup>](#).

As microservice architecture recommended independently deployable units with minimal shared data, in-memory data fabric is not completely aligned with this architectural approach. However, in-memory data fabrics/grid like Apache Ignite can provide independent cache nodes to corresponding microservices in the same distributed cluster and gives you the following advantages over traditional approaches:

1. Maximum uses of the data grid resources (nodes can be added/removed on need basis for scalability in the cluster). Services running on the in-memory cluster is much faster than the disk-based application server.
2. Services can be deployed alongside with the data in the same node, where services compute only the local data (data locality).
3. Automatically deploy any number of distributed services instances in the cluster.
4. Continuous availability of deployed services regardless of topology changes or crashes.

The figure 7.30 shows the main building block of the Apache Ignite microservice solutions. Let's explain all the layers shown below.

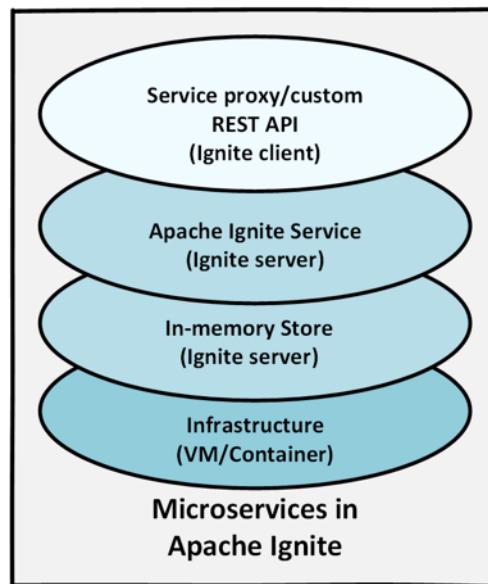


Figure 7.27

<sup>145</sup><http://martinfowler.com/articles/microservices.html#MicroservicesAndSoa>

**Service proxy/custom REST API:** this is the client-side proxy for the Ignite microservice. The service proxy enables the application to send and receive messages as a method call. Service proxy can be resided in the Ignite client node (separate node) and can provide high-level interface to interact with the Ignite microservice. Service proxies are created as needed, opened, used to call a service, and closed when no longer needed. Out-of-the-box, Apache Ignite service proxy is not a *REST style* proxy, you can create a wrapper REST client around the Ignite service proxy.

**Apache Ignite service:** each microservice in Ignite implements the Ignite service interface, which makes it inherently fault-tolerant and provides an easy way to call one microservice from another. Ignite microservice can also be called from the Ignite compute task as needed. Apache Ignite takes care of continuous availability of these deployed services regardless of topology changes or crashes.

**In-memory store:** it's an Ignite server node that holds the portion (or a full copy) of the datasets. It also enables the execution of the microservices.

**Infrastructure:** describes how the Ignite node will be deployed in the cluster. It can be any dedicated server or any container/VM to hold the Ignite node in the cluster. Most popular container to deploy and run Ignite node is the docker container.

In this last section of this chapter we are going to build an application that will allow us to validate the cash withdrawal limit of ATM for each user. Business requirements of our microservice are as follows:

- The microservice accept the customer account number and the transaction amount as input parameters.
- Microservices looks for the client by the customer account number.
- Check the amount of the transactions per day with a specified limit for the account. For instance, 3000\$ is the limit for basic clients.
- Add the amount to the total amount of the transactions per day (reset the number to zero in the case of the first transaction of the day).
- The Microservice also sends a request to the audit system.
- Return the result to the client.

The full source code of the application is available in the [GitHub repository<sup>146</sup>](#). Let's create a new Maven projects or modify the downloaded module.

**Step 1.** Let's start with the BankService interface.

---

<sup>146</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-7/src/main/java/com/blu/imdg/example9>

**Listing 7.59**

---

```
public interface BankService {  
    String NAME = "BANK_SERVICE";  
  
    boolean validateOperation (String account, BigDecimal sum) throws AccountNotFoundException,  
        LogServiceException;  
}
```

---

The `BankService` interface has one method: `validateOperation()`, which accept the client account number and the cash withdrawal amount as input parameters. The above method also throws `AccountNotFoundException`, whenever the account number is not available in the Ignite cache.

**Step 2.** Create another interface to log the client operation. Create a new Java interface with name `LogService` and add the following contents to it.

**Listing 7.60**

---

```
public interface LogService {  
    String NAME = "logService";  
  
    void logOperation (String operationCode, Boolean result) throws LogServiceException;  
}
```

---

The above interface also contains one method `logOperation()`, which accepts the `operationCode` and the result of the operation for sending to the audit system.

**Step 3.** Let's implements the `LogService` interface first. Add a new Java class with name `LogServiceImpl` and implements the following interfaces:

- `LogService`
- `Service`

Where, `Service` is the Ignite Service interface. Our `LogService` becomes a microservice by implementing the Ignite Service interface. Input the following contents into the `LogServiceImpl` class:

Listing 7.61

---

```
public class LogServiceImpl implements LogService, Service {
    private CloseableHttpClient auditClient;

    @Override
    public void logOperation(String operationCode, Boolean result) throws LogServiceException {
        try {
            HttpAuditClient.sendResult(auditClient, result, operationCode);
        } catch (Exception err) {
            throw new LogServiceException(err);
        }
    }

    @Override
    public void cancel(ServiceContext serviceContext) {
        try {
            auditClient.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void init(ServiceContext serviceContext) throws Exception {
        auditClient = (CloseableHttpClient) HttpAuditClient.createHttpClient();
    }

    @Override
    public void execute(ServiceContext serviceContext) throws Exception {
    }
}
```

---

The above implemented class is very similar to what we have done in the previous section. First, we implemented the method `logOperation()`, where, we are sending the result of the operation to the audit system. Then, we populated the `LogServiceImpl` as an Ignite service by implementing the methods: `init()`, `execute()` and `cancel()`.

**Step 4.** Now, create a new Java class with name `BankServiceImpl` and implement the following interface:

- BankService
- Service

Where *Service* is the Ignite Service *interface*.

Listing 7.62

---

```
public class BankServiceImpl implements BankService, Service {

    @ServiceResource(serviceName = LogService.NAME)
    private LogService logService;

    @IgniteInstanceResource
    private Ignite ignite;

    IgniteCache<AccountCacheKey, AccountCacheData> accountCache;

    @Override
    public boolean validateOperation(String account, BigDecimal sum) throws AccountNotFoundException, LogServiceException {
        AccountKey key = new AccountKey(account);
        Lock lock = accountCache.lock(key);
        try {
            lock.lock();
            AccountData accountData = (AccountData) accountCache.get(key);
            if (accountData == null) {
                throw new AccountNotFoundException(account);
            }

            //clean today operations
            if (!accountData.getToday().equals(LocalDate.now())) {
                accountData.setTodayOperationSum(new BigDecimal(0));
                accountData.setToday(LocalDate.now());
                accountCache.put(key, accountData);
            }

            BigDecimal newOperationSum = accountData.getTodayOperationSum().add(sum);
            if (newOperationSum.compareTo(accountData.getDailyLimit()) > 0) {
                logService.logOperation(account, false);
                return false;
            } else {
                accountData.setTodayOperationSum(newOperationSum);
                accountCache.put(key, accountData);
                logService.logOperation(account, true);
            }
        }
    }
}
```

```
    return true;
}

} finally {
    lock.unlock();
}
}

// implementation of the method init(), execute() and cancel() are omitted
}
```

---

First, we inject the Ignite LogService as a resource. If more than one LogService is deployed on a server, then the first available instance will be returned. Then, we injected the current Ignite instance as the resource. Note that, in this example, we are also using `AccountCacheKey` and `AccountCacheData` for data co-allocation. Please refer to the previous section for more information about data affinity. Later in this class, we implemented our business logic to validate the cash withdrawal operation. In the `validateOperation()` method, we first created the `AccountKey` instance from the given account number and locked the cache key. In the next few lines of code, we acquired the lock.



## Info:

If the lock is not available, then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired. This lock is an optimistic lock and guarantees that any other thread or operation on this cache entries will not interrupt our operation and the data.

The rest of the business logic's are as follows: we clean up the cache entry if the current operation is the first transaction of the day. If the new operation exceeded the limit of the cash withdrawal, we send a message to our microservice named *LogService*. Otherwise, we increased the total amount and registered the operation by sending a message to the LogService. Finally, we unlock the entries of the cache.

**Step 5.** Let's deploy the above two microservices and run some test. Create a new Java class with name `TestMicroServiceMain`, and add the following contents.

**Listing 7.63**

---

```
public class TestMicroServiceMain {  
  
    public static void main(String[] args) throws AccountNotFoundException, LogServiceExc  
eption {  
        try (Ignite ignite = Ignition.start(CommonConstants.CLIENT_CONFIG)) {  
  
            IgniteCache<AccountCacheKey, AccountCacheData> cache = BankDataGenerator.crea  
teBankCache(ignite);  
  
            IgniteServices services = ignite.services().withAsync();  
  
            services.deployNodeSingleton(LogService.NAME, new LogServiceImpl());  
            services.future().get();  
  
            services.deployNodeSingleton(BankService.NAME, new BankServiceImpl());  
            services.future().get();  
  
            BankService bankService = services.serviceProxy(BankService.NAME, BankService.  
.class, /*not-sticky*/false);  
  
            System.out.println("result=" + bankService.validateOperation(BankDataGenerato  
r.TEST_ACCOUNT, new BigDecimal(50)));  
            System.out.println("result1=" + bankService.validateOperation(BankDataGenerat  
or.TEST_ACCOUNT, new BigDecimal(40)));  
            System.out.println("result2=" + bankService.validateOperation(BankDataGenerat  
or.TEST_ACCOUNT, new BigDecimal(180)));  
  
            services.cancel(BankService.NAME);  
        }  
    }  
}
```

---

We deploy our two microservice as a singleton service in the above fragment of codes. Also created a service proxy *bankService* to invoke the *BankService*. Next, we generate some data for test and run the application. Next, we will start a few Ignite node and the audit service to test our application.

**Tip**

Ignite does not change the API for IgniteServices to asynchronous call. `services().withAsync()` call is still relevant for the Services API.

**Step 6.** Start the Ignite node with the following command:

**Listing 7.64**

---

```
mvn clean package exec:java -Dexec.mainClass=com.blu.imdg.StartCacheNode
```

---

Start the audit http service as follows:

**Listing 7.65**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.common.HttpAuditEmulator
```

---

Next, run the following Maven command to execute the `TestMicroServiceMain` application.

**Listing 7.66**

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example9.TestMicroServiceMain
```

---

If everything goes fine, you should have the following output into the console.

```
result=true  
result=true  
result=false
```

Note that the last result is *false* because our cash withdrawal limit exceeded. Our daily cash withdrawal limit is only 200\$, and we have asked for 180\$. Let's create a REST client for our microservice.

**Step 6.** Create a new Java class named `ServiceHttpClient`; it will be our HTTP client proxy for the *BankService*.

**Listing 7.67**

---

```
@Path("/service")  
public class ServiceHttpClient {  
    private static BankService bankService;  
    @GET  
    @Path("/withdrawlimit")  
    @Produces(MediaType.TEXT_PLAIN)  
    public boolean acceptResponse (  
        @QueryParam("accountnum") String accnum,  
        @QueryParam("amount") int amount
```

```
) throws Exception
{
    System.out.println("account number=" + accnum + " amount=" + amount);

    return bankService.validateOperation(accnum, new BigDecimal(amount));
}

public static void main(String[] args) {
    URI baseUri = UriBuilder.fromUri("http://localhost/").port(9988).build();
    // start the Ignite client
    Ignite ignite = Ignition.start(CommonConstants.CLIENT_CONFIG);
    IgniteServices services = ignite.services().withAsync();

    bankService = services.serviceProxy(BankService.NAME, BankService.class, /*not-sticky*/false);

    ResourceConfig config = new ResourceConfig(ServiceHttpClient.class);
    HttpServer server = JdkHttpServerFactory.createHttpServer(baseUri, config);
}
}
```

---

We used javax.ws.rs annotation to exposed the Java class as an HTTP service. The service will be available at [http://localhost:9988/service/withdraw?accountnum=0000\\*1111&amount=100](http://localhost:9988/service/withdraw?accountnum=0000*1111&amount=100). You can change the host and the port number in the source code. The service just wraps the Ignite service proxy and expose an HTTP interface to interact with it. Let's run the client with the following command:

#### Listing 7.68

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example9.ServiceHttpClient
```

---

Ensure that your Ignite node is running and **BankService** is deployed. If you just want to deploy the **BankService** in Ignite cluster run the following application.

#### Listing 7.69

---

```
mvn exec:java -Dexec.mainClass=com.blu.imdg.example9.DeployService
```

---

The `DeployService` application will just deploy the *BankService* and the *LogService* in the Ignite cluster. Now, open up your favorite browser and go to the next URL.

[http://localhost:9988/service/withdrawlimit?accountnum=0000\\*1111&amount=100](http://localhost:9988/service/withdrawlimit?accountnum=0000*1111&amount=100).

First two times, you should get the result *TRUE* in your web browser page, in the third time, it should return the result *FALSE* because the cash withdrawal limit is 200 dollar.

**Limitations.** In Microservices architecture, the number of microservices that you need to deal with is quite high. And also, their locations change dynamically, owing to the rapid and agile development/deployment nature of microservices. Therefore, you need to find the location of a microservice during the runtime. The solution to this problem is to use a *Service Registry*. Unfortunately, Apache Ignite doesn't provide any service registry. However, you can develop your own service registry and service discovery application with a little effort.

## Summary

In this chapter, we covered the following topics by examples:

- Developing and deploy distributed closure.
- Developing Map-Reduce and fork join.
- How to use Ignite Per-node share state.
- Compute task fault-tolerance and checkpointing.
- Distributed job scheduling.
- Developing and deploy Services in Ignite cluster.
- What microservice is and how Ignite service can differ from the traditional microservice.
- Developed and deployed microservice in Ignite cluster.
- Create high-level REST client to interact with Ignite microservice.

## What's next?

In the next chapter, we will describe the use of Data streaming and Complex event processing in Apache Ignite.

# Chapter 8. Streaming and complex event processing

Probably you often heard the terms: **Data-at-Rest** and **Data-in-Motion** whenever talking about BigData management. Data-at-rest refers mostly at static data collected from one and many data sources and followed by analysis. The Data-in-motion refers to a mode where all the similar data collection method is applied, and data get analyzed at the same time as it is generated. For instance, sensor data processing for the self-driving car from Google is an excellent example of data-in-motion. Sometimes, this type of data is also called *stream data*. Analysis of a Data-in-motion is often called *Stream processing*, *Real-time analysis* or *Complex event processing*.

Most often, Streaming data is generated continuously by thousands of data sources, which typically send the data records simultaneously and in small sizes. Streaming data includes a wide variety and velocities of data such as log files generated by mobile devices, user activities from e-commerce sites, financial trading floors or tracking information from car/bike sharing devices, etc.

This data needs to be processed *sequentially* and *incrementally*, and used for a wide variety of analytics including aggregation, filtering or business intelligence for taking any business decision with latencies measured in microseconds rather than seconds of response time. Apache Ignite allows loading and processing of continuous never-ending streams of data in a scalable and fault-tolerant fashion, rather than analyzing data after it has reached the database. This enables you to correlate relationships and detect meaningful patterns from significantly more data than you can process it faster and much more efficiently.

Apache Ignite streaming and CEP (Complex Event Processing) can be employed in a wealth of industries area; the following are some first-class use cases:

- **Financial services:** the ability to perform real-time risk analysis, monitoring ,and reporting of financial trading and fraud detection.
- **Telecommunication:** the ability to perform *real-time* call detail record, SMS monitoring, and *DDoS* attack.
- **IT systems and infrastructure:** the ability to detect failed or unavailable applications or servers in real-time.

- **Logistics:** the ability to track shipments and order processing in real-time and reports on potential delays on arrival.
- **In-game player activities:** the ability to collects streaming data about player-game interactions, and feeds the data into its gaming platform. It then analyzes the data in real-time, offers incentives and dynamic experiences to engage its players.

Basically, Apache Ignite Streaming techniques works as follows:

1. Clients inject streams of data into Ignite cluster.
2. Data is automatically partitioned between Ignite data nodes.
3. Data is concurrently processed across all cluster nodes, such as enrichment, filter extra.
4. Clients perform concurrent *SQL queries* on the streamed data.
5. Clients subscribe to *continuous queries* as data changes.

These above activities can be illustrated as shown in figure 8.1.

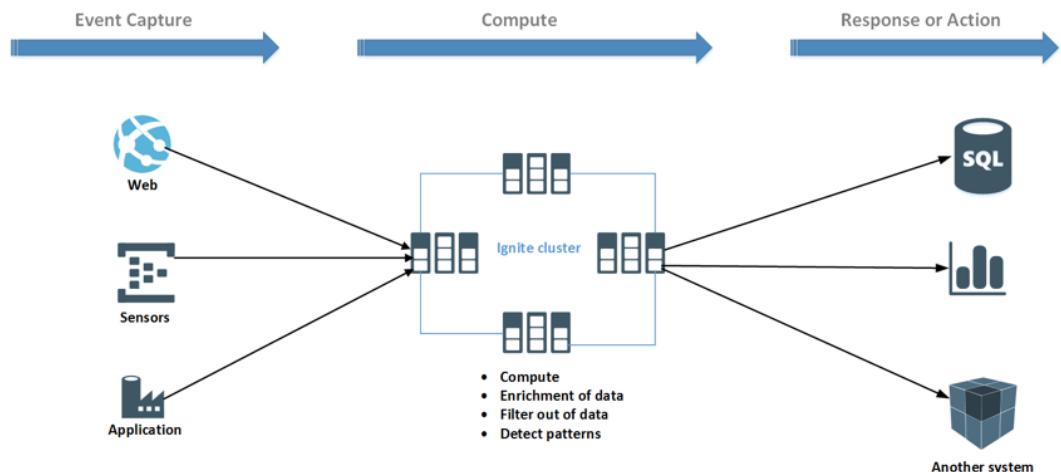


Figure 8.1

Data are ingesting from difference sources. Sources can be any sensors (IoT), web applications or industrial applications. Stream data can be concurrently processed directly on the Ignite cluster in a distributed fashion. Also, data can be computed on third-party CEP application like [confluent<sup>147</sup>](https://www.confluent.io/), [Apache Storm<sup>148</sup>](http://storm.apache.org/) and then aggregated data can be loaded into the Ignite cluster for visualization or for taking some actions.

<sup>147</sup><https://www.confluent.io/>

<sup>148</sup><http://storm.apache.org/>

Apache Ignite provides native data streamers for loading and streaming large amounts of data into Ignite cluster. Data streamers are defined by `IgniteDataStreamer` API and are built to ingest large amounts of endless stream data into Ignite caches. `IgniteDataStreamer` can ingest data from various sources such as files, FTP, queues, etc., but the users must develop the adapter for connecting to the sources. Also, Ignite integrates with major streaming technologies such as Kafka, Camel, Storm or Flume to bring even more advanced streaming capabilities to Ignite-based architectures. At the moment of writing this book, Ignite provides the following data streamers for streaming a large amount of data into the Ignite cluster:

- `IgniteDataStreamer`
- JMS Streamer
- Flume sink
- MQTT Streamer
- Camel Streamer
- Kafka Streamer
- Storm Streamer
- Flink Streamer
- ZeroMQ Streamer
- RocketMQ Streamer

In practice, most developers use the 3<sup>rd</sup> party framework such as Kafka, Camel for initial loading and streaming data into Ignite cluster, because they are well known multi-purpose technologies for complex event processing. So, in this chapter, first of all, we introduce the Kafka streamer and then goes through the rest of the favorite data streamers, and provides some real-world running example for each streamer.

## Kafka Streamer

Apache Ignite out-of-the-box provides *Ignite-Kafka* module with three different solutions (API) to achieve a robust data processing pipeline for streaming data from/to [Kafka<sup>149</sup>](#) topics into Apache Ignite.

Name	Description
IgniteSinkConnector	Consumes messages from Kafka topics and ingests them into an Ignite node.
KafkaStreamer	Fetching data from Kafka topics and injecting them into an Ignite node.
IgniteSourceConnector	Manages source tasks that listens to registered Ignite grid events and forward them to Kafka topics.

The main difference between `IgniteSinkConnector` and `KafkaStreamer` is that, `IgniteSinkConnector` implements [Kafka connector<sup>150</sup>](#) API which is designed for scalable and reliable streaming data between Apache Kafka and other data systems. It offers an API, and *REST Service* to enable developers to quickly define connectors that move large datasets into and out of Kafka. On the other hand, `KafkaStreamer` is based on [Kafka Streams<sup>151</sup>](#) – a lightweight library for creating stream processing applications. The library is lightweight in the sense that it builds on the primitives that are natively built within Kafka for solving problems that stream processing applications need to deal with: fault tolerance, and partitioning. In this section, we will start from the `IgniteSinkConnector` and details all the necessary steps to connect and build an application to ingest data from Kafka topics into Ignite caches.

### Pre-requisites

Apache Ignite *Ignite-Kafka* module and Kafka itself depends on a few frameworks and tools. So, a few considerations need to be taken into account before using it.

Name	Value
JVM	1.8
Kafka broker	2.11_2.0
ZooKeeper	3.4.12
Ignite	2.6.0

The underlying architecture of the Kafka is organized around a few fundamental terms: topics, producers, consumers, and brokers. We often use these terminologies throughout the section, so, before proceeding to more advanced topics, let's define these above terminologies

<sup>149</sup><https://kafka.apache.org/>

<sup>150</sup><http://kafka.apache.org/documentation.html#connect>

<sup>151</sup><http://kafka.apache.org/documentationstreams/>

in details.

Name	Description
Topics	All Kafka messages are organized into topics. If you wish to send a message you send it to a specific topic, and if you wish to read a message you read it from a specific topic.
Broker	Kafka is a distributed system, runs in a cluster. Each node in the cluster is called a Kafka broker.
Producer	Producers produces and push messages into a Kafka topic.
Consumer	A consumer pulls messages off of a Kafka topic.

## IgniteSinkConnector

Apache Ignite provides a new way for data processing based on Kafka connect from the version 1.6. Kafka connects, a new feature introduced in Apache Kafka 0.9 that enables scalable and reliable streaming data between Apache Kafka and other data systems. It's made easy to add new systems to your scalable and secure stream data pipelines in-memory.

*IgniteSinkConnector* provides the following features:

- *Configuration-driven*: *IgniteSinkConnector* is fully configuration-driven, no coding requires to install and run the connector.
- *Management*: provides *REST* full API to management and monitoring the connector. Through REST API you can restart, resume or pause the connector.



### Tip

A connector request is submitted on the command line in standalone mode. This mode is useful for getting status information, adding and removing connectors without stopping the process, and testing and debugging. The REST API is the primary interface to the cluster in distributed mode. Requests can be made to any cluster member where the REST API automatically forwards requests.

A high-level architecture of the *IgniteSinkConnector* is shown in figure 8.2.



Figure 8.2

The connector can be found in the `$IGNITE_HOME/optional/ignite-kafka` directory of the binary distribution. You can also build the connector from the source code. Now that we have a minimum idea about the Apache Kafka and its connector, let's install Kafka and *IgniteSinkConnector* to ingest data from the Kafka topics into Ignite caches. To keep things simple, we will load data from a *file* and streams the data into Ignite cluster through Kafka connector. After completing the configuration, we should have the following Kafka pipeline as shown below.

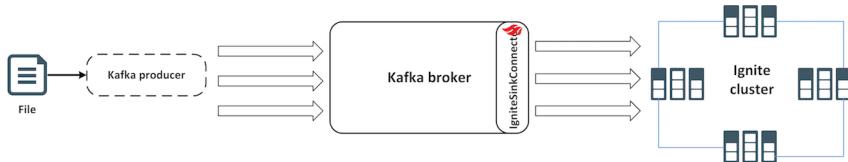


Figure 8.3

**Step 1.** Download and install the Apache Zookeeper. Yes, [Zookeeper<sup>152</sup>](#); Kafka uses Zookeeper to manage the cluster. Zookeeper is used for coordinating the broker topology. So, you need to start a ZooKeeper server first if you don't have one. Download the Zookeeper latest version from this [link<sup>153</sup>](#) and unarchive it somewhere in your workstation. This directory will be the Zookeeper home directory.

#### Listing 8.1

---

```
tar -xzf zookeeper-3.4.12.tar.gz
```

---

**Step 2.** Create a Zookeeper configuration file. Create a file with name `$ZOOKEEPER_HOME/conf/zoo.cfg` and copy the following text into it.

#### Listing 8.2

---

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# for example, /tmp directory
dataDir=ABSOLUTE_PATH_TO_DATA_DIRECTORY/data
```

---

<sup>152</sup><https://zookeeper.apache.org/>

<sup>153</sup><https://www.apache.org/dyn/closer.cgi/zookeeper/>

---

```
# the port at which the clients will connect  
clientPort=2181
```

---

Set the `dataDir` properties to any physical directory located in your operating system. This directory will be used to store a snapshot. Also, note that we are using port `2181` as a client port to connected to the Zookeeper server.

**Step 3.** Start the Zookeeper server in a standalone mode. Execute the following command into the terminal.

**Listing 8.3**

---

```
$ZOOKEEPER_HOME/bin/zkServer.sh start
```

---

This command will start a Zookeeper server in a background mode. You can monitor the status of the Zookeeper server by the following command.

```
$ZOOKEEPER_HOME/bin/zkServer.sh status
```

**Step 4.** Download the [Kafka 2.0<sup>154</sup>](#) version from the site, and un-tar it in your operating system. This directory will be the Kafka home directory.

**Listing 8.4**

---

```
tar -xzf kafka_2.11-2.0.0.tgz
```

---



## Tip

*kafka\_2.11* refers to the Scala version. There is another version of Kafka for the Scala version 2.12. This only matter if you are using Scala and you want a version built for the same version you are using.

**Step 5.** Now, start the Kafka server. Execute the following command from the `$KAFKA_HOME` directory.

---

<sup>154</sup>[https://www.apache.org/dyn/closer.cgi?path=/kafka/2.0.0/kafka\\_2.11-2.0.0.tgz](https://www.apache.org/dyn/closer.cgi?path=/kafka/2.0.0/kafka_2.11-2.0.0.tgz)

**Listing 8.5**

---

```
bin/kafka-server-start.sh config/server.properties
```

---

The above command will run a Kafka broker in a standalone mode with default configurations. You should have the following messages into your console.

```
[2018-09-30 15:39:04,479] INFO Kafka version : 2.0.0 (org.apache.kafka.common.utils.AppInfoParser)
[2018-09-30 15:39:04,479] INFO Kafka commitId : 3402a8361b734732 (org.apache.kafka.common.utils.AppInfoParser)
[2018-09-30 15:39:04,480] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
```

**Step 6.** Let's create a topic named `test` with a single partition and only one replica.

**Listing 8.6**

---

```
$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

---

You can use the `list` command to print out all the existing topics created on the Kafka broker.

**Step 7.** Kafka comes with a command line *producer* client to send messages to the Kafka broker. Each line sends as a separate message by default. Run the producer client in another terminal and then type a few messages into the console to send to the topic.

**Listing 8.7**

---

```
$KAFKA_HOME/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

---

```
k1,v1
k2,v2
```

---

**Step 8.** Kafka also has a command line consumer that will dump out messages to standard output. Start the consumer in a separate terminal with the following command.

**Listing 8.8**

---

```
$KAFKA_HOME/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test \
--from-beginning
```

---

This command should dump two messages into the console.

```
k1,v1
k2,v2
```

Now that our Zookeeper and Kafka servers are up and running, we can configure our *IgniteSinkConnector* to ingest data into Ignite node.

**Step 9.** Copy the following files from the \$IGNITE\_HOME/optional/ignite-kafka directory to \$KAFKA\_HOME/libs directory. \$KAFKA\_HOME/libs directory is the Kafka classpath directory.

**Listing 8.9**

---

```
ignite-kafka-2.6.0.jar
kafka_2.11-0.10.0.1.jar
connect-api-0.10.0.1.jar
```

---

Also, copy the *Ignite-Spring* related files from the \$IGNITE\_HOME/libs/ignite-spring directory into the Kafka classpath directory \$KAFKA\_HOME/libs.

**Listing 8.10**

---

```
spring-context-4.3.16.RELEASE.jar
ignite-spring-2.6.0.jar
spring-core-4.3.16.RELEASE.jar
spring-aop-4.3.16.RELEASE.jar
commons-logging-1.1.1.jar
spring-expression-4.3.16.RELEASE.jar
spring-beans-4.3.16.RELEASE.jar
spring-jdbc-4.3.16.RELEASE.jar
```

---

**Step 10.** Create a new folder named *myconfig* under \$KAFKA\_HOME directory. Create a new connector configuration file named *connect-standalone.properties* under this directory, and add the following configurations:

**Listing 8.11**

---

```
bootstrap.servers=localhost:9092

key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter
key.converter.schemas.enable=false
value.converter.schemas.enable=false

internal.key.converter=org.apache.kafka.connect.storage.StringConverter
internal.value.converter=org.apache.kafka.connect.storage.StringConverter
internal.key.converter.schemas.enable=false
internal.value.converter.schemas.enable=false

offset.storage.file.filename=/tmp/connect.offsets
offset.flush.interval.ms=10000
```

---

For simplicity, we are using `StringConverter` for the key-value of the messages. Let's create another file named `ignite-connector.properties` under the `$KAFKA_HOME/myconfig` directory, and add the following connection properties.

**Listing 8.12**

---

```
# connector
name=my-ignite-connector
connector.class=org.apache.ignite.stream.kafka.connect.IgniteSinkConnector
tasks.max=2
topics=test

# cache
cacheName=myCache
cacheAllowOverwrite=true
igniteCfg=$IGNITE_HOME/2.6.0/examples/config/example-cache.xml
```

---

The configuration is straightforward. We use the `IgniteSinkConnector` class as the connector class name, and topic `test` as a consumer topic to pull data. We also configure `myCache` as the sink cache and allow to overwrite the cache value if exists. Note that we explicitly set the Ignite configuration XML file to connect to the Ignite node. The `example-cache.xml` file contains the following cache configuration.

**Listing 8.13**

---

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="cacheConfiguration">
            <list>
                <bean class="org.apache.ignite.configuration.CacheConfiguration">
                    <property name="name" value="myCache"/>
                    <property name="atomicityMode" value="ATOMIC"/>
                    <property name="backups" value="1"/>
                </bean>
            </list>
        </property>

        <property name="discoverySpi">
            <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
                <property name="ipFinder">
                    <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
                        <property name="addresses">
                            <list>
                                <value>127.0.0.1:47500..47509</value>
                            </list>
                        </property>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

---

**Tip**

You can copy and paste the above Ignite configuration in any file located in your OS and point the file path to the `ignite-connector.properties` file.

You can also set `cachePerNodeDataSize` and `cachePerNodeParOps` to adjust per-node buffer and the maximum number of parallel stream operations for a single node.

**Step 11.** Restart the Kafka broker. Stop the server by command `CRTL+X`, and start again by executing the command from the *Step 5*.

**Step 12.** Now that we have configured our Kafka connector, we can start the connector in a standalone mode. Input the following command in any separate terminal from the Kafka home directory:

**Listing 8.14**

---

```
bin/connect-standalone.sh myconfig/connect-standalone.properties myconfig/ignite-connecto\
r.properties
```

---

You may assume that this will execute the `IgniteSinkConnector` with an Ignite node. The Ignite node will be executed in the embedded mode. Certainly, you can run a separate Ignite node in another JVM, for doing this, you have to apply the Ignite client node configuration into the `ignite-connector.properties` file.

**Step 13.** Start an another Kafka producer to put some messages into topic `test`. Execute the following command in any separate console, and type some messages.

**Listing 8.15**

---

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test --property parse.\nkey=true --property key.separator=,
```

---

```
k1,v1\nk2,v2\nk3,v3
```

---

**Step 14.** Now, we can use Ignite visor to check the values in the cache. Start the Ignite visor console and apply the `cache -scan` command to print the cache values.

```
Entries in cache: myCache
+-----+
|   Key Class    | Key | Value Class   | Value |
+-----+
| java.lang.String | k1  | java.lang.String | v1   |
| java.lang.String | k2  | java.lang.String | v2   |
| java.lang.String | k4  | java.lang.String | v4   |
+-----+
```

**Figure 8.4**

Writing data from the command console and ingesting it into the Ignite cache is a convenient place to start, but you probably want to use data from another sources. For instance, import data from a file. Let's use a file as a data source and import data from it.

**Step 15.** Create a file named `seed.txt` anywhere in your operating system and copy the following text into the file. In my case, I created the file in the Kafka home directory.

#### Listing 8.16

---

Shamim, Apache Ignite deep dive: SQL engine.

Denim martin, ZooKeeper internal engines.

Eric kleppmen, a journey to a NoSQL database.

---

**Step 16.** Start another producer which will read the file `seed.txt`, and then continue tailing for subsequently appended lines as follows:

#### Listing 8.17

---

```
tail -f -n +1 ./seed.txt | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test --property parse.key=true --property key.separator=,
```

---

Note that, in the above command you have to apply the absolute path of the `seed.txt` file. Now, if we re-check the values into the cache, we should discover the three lines of text into the Ignite cache named `myCache`.

Entries in cache: myCache				
Key Class	Key	Value Class	Value	
java.lang.String	Eric kleppmen	java.lang.String	a journey to a NOSql database.	
java.lang.String	K1	java.lang.String	v1	
java.lang.String	k2	java.lang.String	v2	
java.lang.String	K4	java.lang.String	v4	
java.lang.String	Shamim	java.lang.String	Apache Ignite deep dive: SQL engine.	
java.lang.String	Denim martin	java.lang.String	ZooKeeper internal engines.	

Figure 8.5

So far so good. However, what to do if you have to parse the incoming data and choose your new key-value pairs? In this situation, you can implement the business logics as a `StreamSingleTupleExtractor` and specify as `singleTupleExtractorCls` property into the `ignite-connector.properties` file. Below is an example of the implementation of the `StreamSingleTupleExtractor` interface.

**Listing 8.18**

---

```
public class MySuperExtractor implements StreamSingleTupleExtractor<SinkRecord, String, S\\
tring> {

    @Override public Map.Entry<String, String> extract(SinkRecord msg) {
        String[] parts = ((String)msg.value()).split(" ");
        return new AbstractMap.SimpleEntry<String, String>(parts[1], parts[2]+":"+parts[3]);
    }
}
```

---

You can directly add this class to the source code of the `ignite-Kafka` module and build the project. Alternatively, you can create a Maven project and add the class to the project. After building the project, you should modify the `ignite-connector.properties` file and add the following property.

**Listing 8.19**

---

```
singleTupleExtractorCls=com.blu.kafka.MySuperExtractor
```

---

That's all, `IgniteSinkConnector` loads data into the Ignite cache. Now, you are ready for further in-memory processing or analysis via Ignite SQL queries. However, Ignite Kafka sink connector has some limitations that you should consider whenever decided to use it in production.

- Ignite sink connector does not support **filtering**.
- Does not support **multiple caches** for ingesting data.
- Does not support **dynamic reconfiguration** since only one cache is supported.

## IgniteSourceConnector

The Apache `IgniteSourceConnector`<sup>155</sup> is used to subscribe to Ignite cache events and stream them to Kafka topic. In other words, it can be used to export data (changed datasets) from an Ignite cache into a Kafka topic. Ignite source connector listens to registered Ignite grid events such as `PUT` and forward them to Kafka topic. This enables data that has been saved into the Ignite cache to be easily turned into an event stream. Each event stream contains key and two values: *old* and *new*.

The `IgniteSourceConnector` can be used to support the following use cases:

---

<sup>155</sup><https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/stream/kafka/connect/IgniteSourceConnector.html>

1. To automatically notify any clients when a cache event occurs, for instance whenever there is a new entry into the cache.
2. To use an asynchronous event streaming from an Ignite cache to 1-N destinations. The destination can be any database or another Ignite cluster. *These enable you to data replication between two Ignite cluster through Kafka.*

The Apache *IgniteSourceConnector* ships together with the *IgniteSinkConnector*, and available in `ignite-kafka-x.x.x.jar` distribution. *IgniteSourceConnector* requires the following configuration parameters:

Name	Description	Mandatory/optional
igniteCfg	Ignite configuration file path.	Mandatory
cacheName	Name of the Cache.	Mandatory
topicNames	Kafka topics name where event will be streamed.	Mandatory
cacheEvs	Ignite cache events to be listened to, for example PUT.	Mandatory
evtBufferSize	Internal buffer size.	Optional
evtBatchSize	Size of one chunk drained from the internal buffer.	Optional
cacheFilterCls	User-defined filter class.	Optional

A high-level architecture of the *IgniteSinkConnector* is shown in figure 8.6.

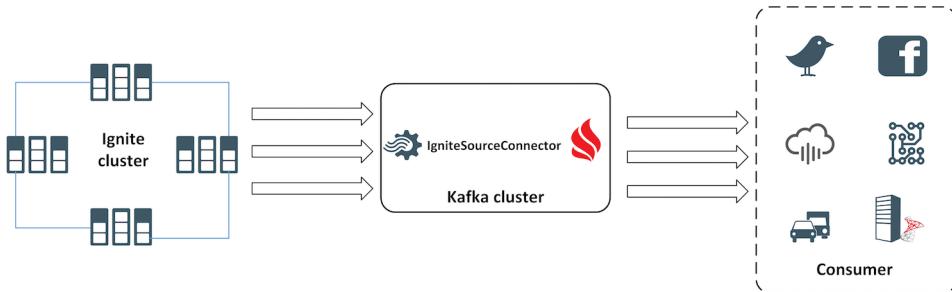


Figure 8.6

In this section, we are going to use both *IgniteSourceConnector* and *IgniteSinkConnector* for streaming event from one Ignite cluster to another. *IgniteSourceConnector* will stream the event from one Ignite cluster (source cluster) to Kafka topic, and the *IgniteSinkConnector* will stream the changes from the topic to the another Ignite cluster (target cluster). We will demonstrate the step by step instructions to configure and run both the Source and Sink connectors. To accomplish the data replication between Ignite clusters, we are going do the following:

1. Execute two isolated Ignite cluster in a single machine.

2. Develop a Stream extractor to parse the incoming data before sending to the Ignite target cluster.
3. Configure and start Ignite Source and Sink connectors in different standalone Kafka workers.
4. Add or modify some data into the Ignite source cluster.

After completing all the configurations, you should have a typical pipeline that is streaming data from one Ignite cluster to another as shown in figure 8.7.

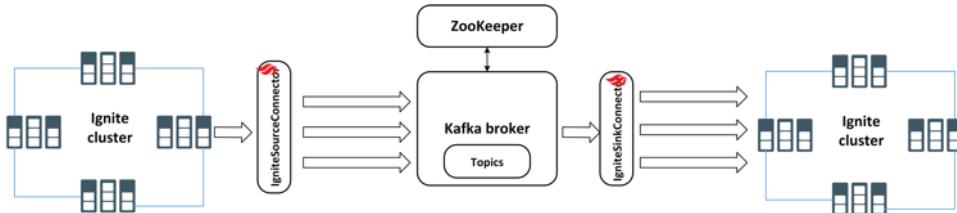


Figure 8.7

We will use the knowledge of Zookeeper and Kafka from the previous section to achieve the task. Let's start from the Ignite cluster configuration.

**Step 1.** We are going to start two isolated clusters on a single machine. To accomplish this, we have to use a different set of `TcpDiscoverySpi` and `TcpConfigurationSpi` to separate the two clusters on a single host. So, for the nodes from the first cluster we proceed to use the following `TcpDiscoverySpi` and `TcpConfigurationSpi` configurations:

**Listing 8.20**

---

```

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/util
           http://www.springframework.org/schema/util/spring-util.xsd">

    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="peerClassLoadingEnabled" value="true"/>
        <property name="cacheConfiguration">
            <list>

```

```
<!-- Partitioned cache example configuration (Atomic mode). -->
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    <property name="name" value="myCacheSource"/>
    <property name="atomicityMode" value="ATOMIC"/>
    <property name="backups" value="1"/>
</bean>
</list>
</property>
<!-- Enable cache events. -->
<property name="includeEventTypes">
    <list>
        <!-- Cache events. -->
        <util:constant static-field="org.apache.ignite.events.EventType.EVT_CACHE\_OBJECT_PUT"/>
    </list>
</property>

<property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
        <!-- Initial local port to listen to. -->
        <property name="localPort" value="48500"/>

        <!-- Changing local port range. This is an optional action. -->
        <property name="localPortRange" value="20"/>

        <!-- Setting up IP finder for this cluster -->
        <property name="ipFinder">
            <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDisco\veryVmIpFinder">
                <property name="addresses">
                    <list>
                        <value>127.0.0.1:48500..48520</value>
                    </list>
                </property>
            </bean>
        </property>
    </bean>
</property>

<!--
Explicitly configure TCP communication SPI changing local
port number for the nodes from the first cluster.
-->
```

```
<property name="communicationSpi">
    <bean class="org.apache.ignite.spi.communication.tcp.TcpCommunicationSpi">
        <property name="localPort" value="48100"/>
    </bean>
</property>
</bean>
</beans>
```

---

We have specified the local port 48500 to listen to and used the static IP finder for node discovering. Also, we have explicitly configured the TCP communication port to 48100. Each Ignite node start with this above configuration will only join to this cluster and will not visible to another cluster on the same host. Note that we also enable the `EVT_CACHE_OBJECT_PUT` event for getting PUT event notification for each entry into the cache. As a data source, we are going to use the *myCacheSource* replicated cache. Save the file with name `isolated-cluster-1-kafka-source.xml` into the `$IGNITE_HOME/examples/config` folder.

We have to use another set of ports for the nodes from the second cluster. The configuration will look like this:

Listing 8.21

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="peerClassLoadingEnabled" value="true"/>
        <property name="cacheConfiguration">
            <list>
                <!-- Partitioned cache example configuration (Atomic mode). -->
                <bean class="org.apache.ignite.configuration.CacheConfiguration">
                    <property name="name" value="myCacheTarget"/>
                    <property name="atomicityMode" value="ATOMIC"/>
                    <property name="backups" value="1"/>
                </bean>
            </list>
        </property>
        <property name="discoverySpi">
            <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
```

```
<!-- Initial local port to listen to. -->
<property name="localPort" value="49500"/>

<!-- Changing local port range. This is an optional action. -->
<property name="localPortRange" value="20"/>

<!-- Setting up IP finder for this cluster -->
<property name="ipFinder">
    <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDiscoveryVmIpFinder">
        <property name="addresses">
            <list>
                <value>127.0.0.1:49500..49520</value>
            </list>
        </property>
    </bean>
</property>
</bean>
</property>
<!--
Explicitly configure TCP communication SPI changing local port number
for the nodes from the second cluster.
-->
<property name="communicationSpi">
    <bean class="org.apache.ignite.spi.communication.tcp.TcpCommunicationSpi">
        <property name="localPort" value="49100"/>
    </bean>
</property>
</bean>
</beans>
```

---

We defined discovery port to 49500 for the nodes from the second cluster, and communication port to 49100. Difference between the two configurations is insignificant - only port numbers for SPIs and IP finder is vary. Save this configuration as a file with name isolated-cluster-1.xml and place the file into the folder \$IGNITE\_HOME/examples/config.

Let's test the configuration. Start two Ignite nodes in separate console with the different configuration file. Here is an example of how you would run the Ignite node.

**Listing 8.22**

```
ignite.sh $IGNITE_HOME/examples/config/isolated-cluster-1-kafka-source.xml
ignite.sh $IGNITE_HOME/examples/config/isolated-cluster-2.xml
```

Figure 8.8 shows the result from the above command. As expected, two separate Ignite nodes up and running in the different cluster.

```
[12:56:19] Configured failure handler: [hnd=StopNodeOrHaltFailureHandler [tryStop=false, timeout=0]] [12:56:19] Message queue limit is set to 0 which may lead to potential OOMEs when running cache operations in FULL_ASYNC or PRIMARY_SYNC modes due to message queues growth on sender and receiver sides. [12:56:19] Security status: [authenticationOff, tlsSslOff] [12:56:20] Performance suggestions for grid (fix if possible) [12:56:20] To disable, set IGNITE_PERFORMANCE_SUGGESTIONS_DISABLED=true [12:56:20] ^-- Disable grid events (remove 'includeEventTypes' from configuration) [12:56:20] ^-- Enable G1 Garbage Collector (add '-XX:+UseG1GC' to JVM options) [12:56:20] ^-- Set max direct memory size if getting 'OOME: Direct buffer memory' (add '-XX:+MaxDirectMemorySize=<size>[g|G|m|M|k|K]' to JVM options) [12:56:20] ^-- Disable processing of calls to System.gc() (add '-XX:+DisableExplicitGC' to JVM options) [12:56:20] ^-- Decrease number of backups (set 'backups' to 0) [12:56:20] Refer to this page for more performance suggestions: https://apachignite.readme.io/docs/jvm-and-system-tuning [12:56:20] [12:56:20] To start Console Management & Monitoring run ignitevisorcmd.sh[bat] [12:56:20] [12:56:20] Ignite node started OK ([id=1d2900b5] [12:56:20] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=-1.0GB] [12:56:20] ^-- Node [id=1d2900b5-6AD6-43D3-9816-F311B657C83F, clusterState=ACTIVE] [12:56:20] Data Regions Configured: [12:56:20] ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false] [12:56:31] Configured failure handler: [hnd=StopNodeOrHaltFailureHandler [tryStop=false, timeout=0]] [12:56:31] Configured failure handler: [hnd=StopNodeOrHaltFailureHandler [tryStop=false, timeout=0]] [12:56:32] Message queue limit is set to 0 which may lead to potential OOMEs when running cache operations in FULL_ASYNC or PRIMARY_SYNC modes due to message queues growth on sender and receiver sides. [12:56:32] Security status: [authenticationOff, tlsSslOff] [12:56:32] Performance suggestions for grid (fix if possible) [12:56:32] To disable, set IGNITE_PERFORMANCE_SUGGESTIONS_DISABLED=true [12:56:32] ^-- Enable G1 Garbage Collector (add '-XX:+UseG1GC' to JVM options) [12:56:32] ^-- Set max direct memory size if getting 'OOME: Direct buffer memory' (add '-XX:+MaxDirectMemorySize=<size>[g|G|m|M|k|K]' to JVM options) [12:56:32] ^-- Disable processing of calls to System.gc() (add '-XX:+DisableExplicitGC' to JVM options) [12:56:32] ^-- Decrease number of backups (set 'backups' to 0) [12:56:32] Refer to this page for more performance suggestions: https://apachignite.readme.io/docs/jvm-and-system-tuning [12:56:32] [12:56:32] To start Console Management & Monitoring run ignitevisorcmd.sh[bat] [12:56:32] [12:56:32] Ignite node started OK ([id=e85fa805] [12:56:32] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=-1.0GB] [12:56:32] ^-- Node [id=E85FA805-F630-41AA-9072-09973EE32886, clusterState=ACTIVE] [12:56:32] Data Regions Configured: [12:56:32] ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=false]
```

Figure 8.8

**Info**

All the listings and the configuration files are available on [GitHub repository<sup>156</sup>](https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-8/kafka/scripts).

**Step 2.** Next, you need to define the stream extractor for converting the data to Key-value tuple. Create a Maven project and add the following dependency into the pom.xml.

**Listing 8.23**

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-kafka</artifactId>
    <version>2.6.0</version>
</dependency>
```

We use ignite-kafka module as our dependency. Add the following Java class with name CsvStreamExtractor into the com.blu.imdg package which will implement the StreamSingleTupleExtractor interface as follows:

<sup>156</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-8/kafka/scripts>

**Listing 8.24**

---

```
public class CsvStreamExtractor implements StreamSingleTupleExtractor<SinkRecord, String> {

    public Map.Entry<String, String> extract(SinkRecord sinkRecord) {
        System.out.println("SinkRecord:" + sinkRecord.value().toString());

        String[] parts = sinkRecord.value().toString().split(",");
        String key = ((String[])parts[2].split("="))[1];
        String val= ((String[])parts[7].split("="))[1];

        return new AbstractMap.SimpleEntry<String, String>(key, val);
    }
}
```

---

The `extract()` method is the workhorse of the class `CsvStreamExtractor`. The business logic of the method is very simple: it retrieves the key and value from every tuple of an event, where each tuple is exposed as a `SinkRecord` in the stream. The `extract` method returns the key-value pair, which will be sent to the Ignite cluster (target) for storing into the cache.

Compile and build the project with Maven command `mvn clean install`. After successfully compilation the project a new library named `kafka-1.0.jar` should be created in the project target directory. Copy the library to the folder `$KAFKA_HOME/libs`.

**Step 3.** Now that, our Stream extractor is ready to use. Let's configure the Ignite *source* and *sink* connector and start them for replicating data. Let's create a file with name `ignite-connector-source.properties` into the `$KAFKA_HOME/myconfig` directory. Add the following properties and save the file.

**Listing 8.25**

---

```
# connector
name=my-ignite-source-connector
connector.class=org.apache.ignite.stream.kafka.connect.IgniteSourceConnector
tasks.max=2
topicNames=test2

# cache
cacheName=myCacheSource
cacheAllowOverwrite=true
cacheEvts=put
igniteCfg=PATH_TO_THE_FILE/isolated-cluster-1-kafka-source.xml
```

---

In the above connector configuration, we have defined the `kafka.connect.IgniteSourceConnector` as a connector class. We also specified `test2` as a topic name, where the stream event will be stored. Next, for cache configuration, we have defined the `put` event as grid remote event. In our case, we are using the `myCacheSource` as a source cache. Here, another critical property is `igniteCfg`, where we explicitly specified one of the isolated cluster configurations. *Cluster 1* will be our source of events.

Next, let's configure the Ignite sink connector. Create another file with the name `ignite-connector-sink.properties` into the `$KAFKA_HOME/myconfig` directory. Add the following properties from the listing 8.26.

#### Listing 8.26

---

```
# connector
name=my-ignite-sink-connector
connector.class=org.apache.ignite.stream.kafka.connect.IgniteSinkConnector
tasks.max=2
topics=test2

# cache
cacheName=myCacheTarget
cacheAllowOverwrite=true
igniteCfg=PATH_TO_THE_FILE/isolated-cluster-2.xml
singleTupleExtractorCls=com.blu.imdg.CsvStreamExtractor
```

---

The configuration is as same as before that we used in the previous section. The main difference is the `singleTupleExtractorCls` property where we have specified our Stream extractor that developed in *Step 2*.

**Step 4.** Start the Zookeeper and the Kafka broker (server) as described in the previous section. Go through the *Step 1-5* if you do not have installed Zookeeper and Kafka in your host machine.

**Step 5.** As you may guess, we have to create a new Kafka topic with name `test2`. Let's create the topic by the following command.

#### Listing 8.27

---

```
$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic test2
```

---

**Step 6.** Let's start the Source and Sink connector in a separate console. First, start the source connector, use the following command.

**Listing 8.28**

---

```
bin/connect-standalone.sh myconfig/connect-standalone.properties myconfig/ignite-connecto\\
r-source.properties
```

---

This will employ the default connector properties to start the source connector. Note that, this connector will also start an Ignite server node which will join to our Ignite *cluster 1*.

The final piece of the puzzle is the Sink connector. At this moment, we are ready to start the Sink connector. However, we have to change the REST port and the storage file name for the connector before starting another Kafka connector in standalone mode. Create a file with name `connect-standalone-sink.properties` into the `$KAFKA_HOME/myconfig` folder. Add the following properties to it.

**Listing 8.29**

---

```
bootstrap.servers=localhost:9092

key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter
key.converter.schemas.enable=false
value.converter.schemas.enable=false

internal.key.converter=org.apache.kafka.connect.storage.StringConverter
internal.value.converter=org.apache.kafka.connect.storage.StringConverter
internal.key.converter.schemas.enable=false
internal.value.converter.schemas.enable=false

offset.storage.file.filename=/tmp/connect-1.offsets
rest.port=8888
offset.flush.interval.ms=10000
```

---

Most of the configurations are the same as before, except the `rest.port` and the `storage.offset.storage.file.filename` configuration. We have explicitly defined a new port 8888 for this connector, and specified another file storage. Start the connector with this configuration from the `$KAFKA_HOME` directory.

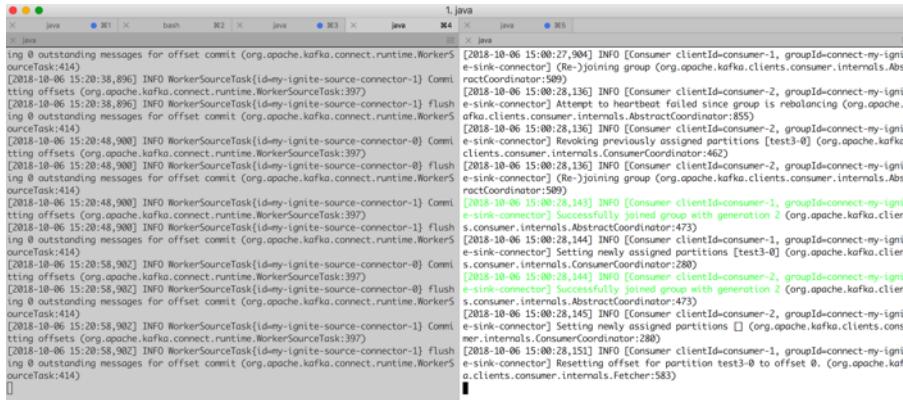
**Listing 8.30**


---

```
bin/connect-standalone.sh myconfig/connect-standalone-sink.properties myconfig/ignite-con\x
nector-sink.properties
```

---

The above command will start a *sink connector* on another console. Figure 8.9 shows a screenshot of the two connectors up and running on the separate console.



```
[2018-10-06 15:20:38,906] INFO WorkerSourceTask{id=my-ignite-source-connector-1} Committing offsets [org.apache.kafka.connect.runtime.WorkerSourceTask:397]
[2018-10-06 15:20:38,906] INFO WorkerSourceTask{id=my-ignite-source-connector-1} Flush ing 0 outstanding messages for offset commit [org.apache.kafka.connect.runtime.WorkerSourceTask:414]
[2018-10-06 15:20:48,980] INFO WorkerSourceTask{id=my-ignite-source-connector-0} Committing offsets [org.apache.kafka.connect.runtime.WorkerSourceTask:397]
[2018-10-06 15:20:48,980] INFO WorkerSourceTask{id=my-ignite-source-connector-0} Flush ing 0 outstanding messages for offset commit [org.apache.kafka.connect.runtime.WorkerSourceTask:414]
[2018-10-06 15:20:48,980] INFO WorkerSourceTask{id=my-ignite-source-connector-1} Committing offsets [org.apache.kafka.connect.runtime.WorkerSourceTask:397]
[2018-10-06 15:20:48,980] INFO WorkerSourceTask{id=my-ignite-source-connector-1} Flush ing 0 outstanding messages for offset commit [org.apache.kafka.connect.runtime.WorkerSourceTask:414]
[2018-10-06 15:20:48,980] INFO WorkerSourceTask{id=my-ignite-source-connector-0} Committing offsets [org.apache.kafka.connect.runtime.WorkerSourceTask:397]
[2018-10-06 15:20:48,980] INFO WorkerSourceTask{id=my-ignite-source-connector-0} Flush ing 0 outstanding messages for offset commit [org.apache.kafka.connect.runtime.WorkerSourceTask:414]
[2018-10-06 15:20:58,982] INFO WorkerSourceTask{id=my-ignite-source-connector-1} Committing offsets [org.apache.kafka.connect.runtime.WorkerSourceTask:397]
[2018-10-06 15:20:58,982] INFO WorkerSourceTask{id=my-ignite-source-connector-1} Flush ing 0 outstanding messages for offset commit [org.apache.kafka.connect.runtime.WorkerSourceTask:414]
[2018-10-06 15:20:58,982] INFO WorkerSourceTask{id=my-ignite-source-connector-2} Committing offsets [org.apache.kafka.connect.runtime.WorkerSourceTask:397]
[2018-10-06 15:20:58,982] INFO WorkerSourceTask{id=my-ignite-source-connector-2} Flush ing 0 outstanding messages for offset commit [org.apache.kafka.connect.runtime.WorkerSourceTask:414]
[2018-10-06 15:20:27,904] INFO [Consumer clientId=consumer-1, groupId=connect-my-ignite-e-sink-connector] (Re-)joining group [org.apache.kafka.clients.consumer.internals.AbstractCoordinator:509]
[2018-10-06 15:20:28,136] INFO [Consumer clientId=consumer-2, groupId=connect-my-ignite-e-sink-connector] (Re-)joining group [org.apache.kafka.clients.consumer.internals.AbstractCoordinator:855]
[2018-10-06 15:20:28,136] INFO [Consumer clientId=consumer-1, groupId=connect-my-ignite-e-sink-connector] Revoking previously assigned partitions [test3-0] [org.apache.kafka.clients.consumer.internals.AbstractCoordinator:462]
[2018-10-06 15:20:28,136] INFO [Consumer clientId=consumer-2, groupId=connect-my-ignite-e-sink-connector] Revoking previously assigned partitions [test3-0] [org.apache.kafka.clients.consumer.internals.ConsumerCoordinator:462]
[2018-10-06 15:20:28,136] INFO [Consumer clientId=consumer-2, groupId=connect-my-ignite-e-sink-connector] Setting newly assigned partitions [test3-0] [org.apache.kafka.clients.consumer.internals.ConsumerCoordinator:280]
[2018-10-06 15:20:28,144] INFO [Consumer clientId=consumer-1, groupId=connect-my-ignite-e-sink-connector] Setting newly assigned partitions [test3-0] [org.apache.kafka.clients.consumer.internals.ConsumerCoordinator:280]
[2018-10-06 15:20:28,144] INFO [Consumer clientId=consumer-2, groupId=connect-my-ignite-e-sink-connector] (Re-)joining group with generation 2 [org.apache.kafka.clients.consumer.internals.ConsumerCoordinator:473]
[2018-10-06 15:20:28,145] INFO [Consumer clientId=consumer-2, groupId=connect-my-ignite-e-sink-connector] Setting newly assigned partitions [] [org.apache.kafka.clients.consumer.internals.ConsumerCoordinator:280]
[2018-10-06 15:20:28,151] INFO [Consumer clientId=consumer-1, groupId=connect-my-ignite-e-sink-connector] Resetting offset for partition test3-0 to offset 0. [org.apache.kafka.clients.consumer.internals.Fetcher:583]
```

**Figure 8.9**

**Step 7.** Now that we have set up our connectors, it's time to test the stream pipeline. At this point, if we put some entries into the `myCacheSource` cache created on cluster 1, the entries should be replicated to the `myCacheTarget` cache on cluster 2. We have several ways to load some entries into the cache `myCacheSource`: by using Ignite REST API or Java client. Let's use the Ignite Java client `IsolatedCluster` from the *chapter 2*.

**Listing 8.31**


---

```
$ java -jar ./target/IsolatedClient-runnable.jar
```

---

This Java client loads 22 entries into the cache `myCacheSource`. Let's observe what's happens on Ignite clusters. Use two Ignite Visor tools to connect to the clusters, one for each cluster. Execute the `cache-scan` command to scan the cache, and you should get a very similar screenshot shown in figure 8.10.

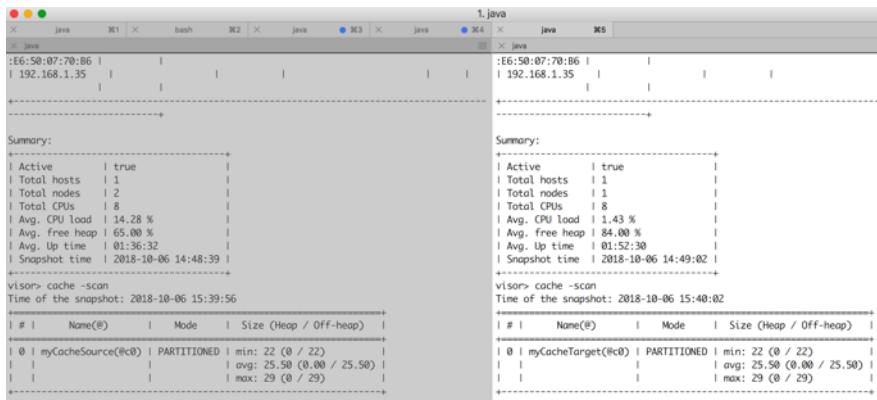


Figure 8.10

As shown in figure 8.10, you can notice that each cache in different clusters contains the same sets of entries. If you carefully look at the Ignite sink connector logs on the console, you should find logs similar as follows:

```

CacheEvent [cacheName=myCacheSource, part=64, key=Key:150, xid=null, lockId=GridCacheVersionion [topVer=150300733, order=1538826349084, nodeOrder=4], newVal>Hello World!!: 150, oldVal=null, hasOldVal=false, hasNewVal=true, near=false, subJId=572ac224-f48b-4a0c-a844-496f4d609b6a, cloClsName=null, taskName=null, nodeJId=fb6ae4b6, evtNodeJId=572ac224, msg=Cache event., type=CACHE_OBJECT_PUT, tstamp=1538829094472]

```

keyKey:150  
Val>Hello World!!:

Ignite source connector stream the cache *PUT* event into the topic *test2* as a tuple, which contains the metadata along with Key and Values: old and new values. Ignite Sink connector uses the `CsvStreamExtractor` extractor to retrieve the value from the tuple and stores the Key-Value pair into the cache `myCacheTarget`.

In the above sample, we have configured only one direction real-time data replication between Ignite clusters. However, Apache Ignite Kafka connectors pack a lot of power into a small module. You can develop powerful bi-directional data replication pipeline or notify any client application whenever any Cache event occurs into the Grid by taking advantage of its versatility and ease of use. Also, you can use any Kafka JDBC Sink connector along with Ignite source connector to push data into any RDBMS. However, Ignite source connector also has some limitations that you should take into account before using it in the production environment:

1. Ignite source connector is not working in parallel. It cannot split work, and one task instance handles the stream.
2. It does not handle multiple caches. In order to handle multiple caches, you have to define multiple connectors configured and running in Kafka.
3. Ignite source connector needs server node to be started in embedded mode to get notified event.
4. It does not support dynamic reconfiguration.

Beside these two connectors, Apache Ignite also provides a Stream adapter `KafkaStreamer` that subscribes to topic messages from Kafka broker, and streams it's to key-value pairs into a `IgniteDataStreamer` instance. The adapter is useful if you are using separate component to up and running a *Kafka streamer* instead of using *Kafka connector*. Anyway, this adapter does not carry any practical value because Kafka connector is easy to use and can be dynamically managed through REST API. So, if you are interested in using `KafkaStreamer` please see the [Ignite documentation<sup>157</sup>](#) for further information. In the next section of this chapter, we are going to explore the capabilities of Apache Camel, and its different ways of integrations with Apache Ignite.

## Camel Streamer

Integration and data exchange between several systems are one of the most challenging tasks for any industry. Different systems use separate interfaces, protocols to interact with each other which dramatically increase the complexities of communications with other systems. Traditional system integrations that we built them in the past decades require a lot of code to be created that has absolutely nothing to do with the higher-level integration problem. The majority of this is boilerplate code, dealing with common, repetitive tasks of setting up, and tearing down libraries for the messaging transports, connecting to the database, etc. Writing such boilerplate code for integration systems prevents developers to focus on business logic.

After publishing the book [Enterprise Integration Patterns: Designing, Building and deploying Messaging solution<sup>158</sup>](#), EIP became the industry standard for describing, documenting and implementing integration problems.

Apache Camel<sup>159</sup> is a powerful open source integration framework based on known Enterprise integration patterns.

<sup>157</sup><https://apacheignite-mix.readme.io/docs/kafka-streamer>

<sup>158</sup>[https://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley-ebook/dp/B007MQLL4E/ref=sr\\_1\\_1?ie=UTF8&qid=1538911330&sr=8-1&keywords=Enterprise+Integration+Patterns%3A+Designing%2C+Building+and+deploying+Messaging+solution&dpID=51pcuI4QqML&preST=\\_SY445\\_QL70\\_&dpSrc=src](https://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley-ebook/dp/B007MQLL4E/ref=sr_1_1?ie=UTF8&qid=1538911330&sr=8-1&keywords=Enterprise+Integration+Patterns%3A+Designing%2C+Building+and+deploying+Messaging+solution&dpID=51pcuI4QqML&preST=_SY445_QL70_&dpSrc=src&)

<sup>159</sup><http://camel.apache.org/>

Apache Camel framework hides all the complexity of integrations, so you can focus only on implementing your business logic. Apache Camel is also known as routing, and mediation engine as it effectively routes data between endpoints, while taking a heavy load of data transformation, endpoint connectivity and much more.

Apache Ignite provides camel streamer, which they described as a universal streamer because it allows consuming data from a variety of sources into Ignite cache. At the moment of writing the book, there are more than 200+ adapters and components available for Camel, which gives the ability to consume any type of data through the various protocol such as *JMS*, *SOAP*, *SFTP*, *SMTP*, *POP3*, etc. and provides opportunities to integrate with any kind of information system.

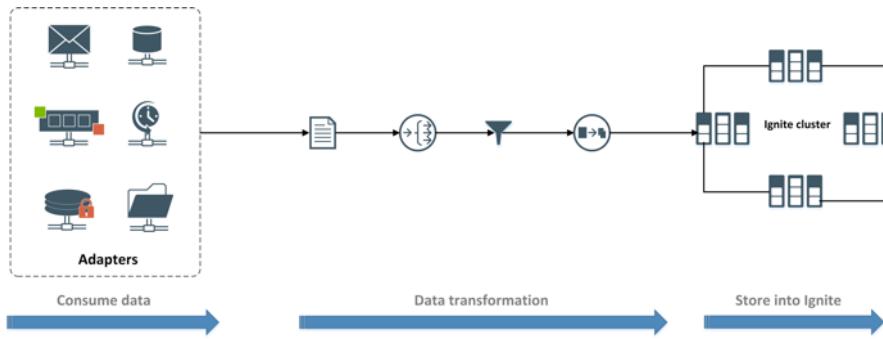


Figure 8.11

Apache Camel not only provides the opportunities to consume data but also, through camel you can route data from one endpoint to another, enrich data or transform data from one format to another. In Camel, you can also execute any custom logic into the data stream. There are two different components for Camel to work with Apache Ignite.

1. [Camel-Ignite<sup>160</sup>](http://camel.apache.org/ignite.html) component;
2. and [Ignite-camel<sup>161</sup>](https://apacheignite-mix.readme.io/docs/camel-streamer) streamer.

The Camel-Ignite library or component offers seamless integration of Camel with Ignite. If you already have an application developed on Camel, you can use these components to integrate with Ignite. The Camel-ignite component can help you to consume entries from caches or store any entries into the Ignite cache. This component offers several endpoints to cover most of the Ignite's functionality.

<sup>160</sup><http://camel.apache.org/ignite.html>

<sup>161</sup><https://apacheignite-mix.readme.io/docs/camel-streamer>

Nº	Endpoint Name	Description
1	Ignite Cache	Allows you to interact with Ignite Cache.
2	Ignite Compute	Allows you to compute operations on Ignite cluster, this endpoint only supports producer.
3	Ignite Messaging	Allows you to send and consume messages to and from Ignite topics.
4	Ignite Events	Allows you to consume events from Ignite cluster, this endpoint only works as consumers.
5	Ignite Sets	Allows you to manipulate with Ignite <i>Set</i> data structures.
6	Ignite Queues	Allows you to manipulate with Ignite <i>Queue</i> data structures.
7	Ignite Id Generator	Allows you to invoke the Ignite Id Generator.

This component lacks one functionality; it can't help you to work with the stream data. For working with streaming data, you have to use *Ignite camel streamer*. Ignite camel stream library can stream data with two different flavors.

1. Direct Ingestion.
2. Mediated Ingestion.

## Direct Ingestion

In the direct ingestion, you consume data from any source through camel endpoint and ingest the data directly into Ignite Cache. In other words, you can use this approach when you do not need to alter the incoming messages before storing them into Ignite cache. In this case, no data transformation or custom logic executed over the data. However, you have to extract the tuples and create the key and the values from tuples manually. Ignite provides two types of tuple extractor: **StreamSingleTupleExtractor** and **StreamMultipleTupleExtractor**.

- *StreamSingleTupleExtractor*, which extracts either no or one tuple out of the message and send them to Ignite to store.
- *StreamMultipleTupleExtractor*, which is capable of extracting multiple tuples out of a single message.

Next, we will develop an example of direct ingestion of streaming data into the Ignite cache. We will use the basic features of the Camel in our examples to keep things simple. To demonstrate the power of the Camel and the direct ingestion mechanism of Ignite, we will do the following:

1. Consume messages from the file system (directory named *input* in the Maven project) in JSON format.

2. Convert the JSON data into Java objects.
3. Directly store the Java object into an Ignite Cache.

Now, let's start developing the application.

**Step 1.** To start, create a new Maven project and add the following maven dependencies into the `pom.xml` file for working with the Ignite-camel streamer. The entire project can be found on [GitHub<sup>162</sup>](#). If you prefer, you can clone the project from the GitHub repository and edit with any text editor.

Listing 8.32

---

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-log4j</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-camel</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>2.6.6</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jackson</artifactId>
    <version>2.16.0</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.3</version>
</dependency>
```

---

<sup>162</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-8/camel>

Note that, the following library is **mandatory** for using Ignite-camel streamer.

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>2.6.0</version>
</dependency>
```



## Info

At the moment of writing this book, Apache Ignite supports *Camel 2.16.0* version.

**Step 2.** Because the component transforms data, a Java class (POJO) needs to be created to map the data from the file. Create a Java class named `MnpRouting` into the `com.blu.imdg.dto` package. The code in Listing 8.33 contains abbreviated source code for the `MnpRouting` class.

Listing 8.33

---

```
public class MnpRouting {
    @QuerySqlField(index = true)
    private String telephone;
    @QuerySqlField(index = true)
    private long credit;
    @QuerySqlField(index = true)
    private String routeTo;
    // setter and getter goes here.
    @Override
    public String toString() {
        return "Routing {" +
            "telephone=" + telephone + '\n' +
            ", credit=" + credit +
            ", routeTo=" + routeTo + '\n' +
            '}';
    }
}
```

---

Class `MnpRouting` has three properties: `telephone`, `credit`, and the `routeTo`, where the `telephone` number is the client telephone number and the `routeTo` properties is the telephone operator, where the call should be routed.

**Step 3.** Next, create a file which will be the source of the data stream. Camel file adapter will consume this file periodically and sends to the Camel Ignite streamer. Create a text file in any directory with the following text.

**Listing 8.34**

---

```
{"telephone": "89096860917", "credit": "129", "routeTo": "MTS"}
```

---

**Step 4.** Now, it's the time to add our main functionality. Add the following Java class named CamelStreamerDirectIngestion to the com.blu.imdg.camel package.

**Listing 8.35**

```
public class CamelStreamerDirectIngestion {  
    private static final String FILE_PROPERTIES= "camel.properties";  
    public static void main(String[] args) {  
        Ignition.setClientMode(true);  
        Ignite ignite = Ignition.start("example-ignite.xml");  
        if (!ExamplesUtils.hasServerNodes(ignite))  
            return;  
        if(getFileLocation() == null || getFileLocation().isEmpty())  
            System.out.println("Properties file is empty or null!");  
        return;  
    }  
    // camel_cache cache configuration  
    CacheConfiguration<String, MnpRouting> camel_cache_cfg = new CacheConfiguration<>\n("camel-direct");  
    camel_cache_cfg.setIndexedTypes(String.class, MnpRouting.class);  
    IgniteCache<String, MnpRouting> camel_cache = ignite.getOrCreateCache(camel_cache\\  
_cfg);  
    // Create an streamer pipe which ingests into the 'camel_cache' cache.  
    IgniteDataStreamer<String, MnpRouting> pipe = ignite.dataStreamer(camel_cache.get\\  
Name());  
    pipe.setAutoFlushFrequency(1l);  
    pipe.allowOverwrite(true);  
    // Create a Camel streamer and connect it.  
    CamelStreamer<String, MnpRouting> streamer = new CamelStreamer<>();  
    streamer.setIgnite(ignite);  
    streamer.setStreamer(pipe);  
    streamer.setEndpointUri("file://" + getFileLocation());  
    streamer.setSingleTupleExtractor(new StreamSingleTupleExtractor<Exchange, String,\\  
MnpRouting>()) {  
        @Override  
        public Map.Entry<String, MnpRouting> extract(Exchange exchange) {
```

```

ObjectMapper mapper = new ObjectMapper();
String msgBody = exchange.getIn().getBody(String.class);
MnpRouting obj = null;
try {
    obj = mapper.readValue(msgBody, MnpRouting.class);
} catch (IOException e) {
    e.printStackTrace();
}
if (obj != null){
    return new GridMapEntry<String, MnpRouting>(obj.getTelephone(), obj);
}
return new GridMapEntry<String, MnpRouting>(null, null);
}
});
streamer.start();
}
}

```

---

Let's have a more in-depth look at the class and go through it line by line. First of all, we declared this Ignite node as a client node by the statement of `Ignition.setClientMode(true)`. Then, we started the Ignite node and read the `camel.properties` file from the classpath. The `camel.properties` file is located in the project `resources` directory. Next, we declared the Cache configuration with name `came-direct`, and set the Index type for executing SQL queries on Cache. In the next few lines of code, we created an Ignite cache from the cache configuration and one Ignite data streamer with name `pipe`. Note the next statement:

```
pipe.autoFlushFrequency(1l)
```

This above statement indicates the auto flash interval for the data stream. Auto flash is disabled for data streamer by default. Set the value near 1 millisecond if you want to store the entries immediately.

Next, we created the camel streamer and set the `ignite-streamer` pipe on it. Also, we point the endpoint properties to our `input` directory for the camel streamer. A JSON files will be consumed from the input directory. Also, we added single tuple extractor to the camel streamer and implemented the method `extract()`. In the extract method, we retrieved message payload from the camel exchange object and converted the JSON text to `MnpRouting` Java object. Next, we filled the Map with the *telephone number* as a key, and the *MnpObject* as the value of the cache entry.

**Step 5.** Edit the `resources\camel.properties` file and set the value for the property `input.file.location`. The `input.file.location` property should have the **absolute directory** path of the file that you have created in *Step 3*. Now, compile and build the project with Maven by executing the following command:

**Listing 8.36**

---

```
mvn clean install
```

---

The above command will create a `camel-direct-ingestion-runnable.jar` executable jar in the Maven project `target` directory.

**Step 6.** To execute the application, follow the following steps:

1. Run the `node-runner-runnable.jar` executable jar to run an Ignite node from the directory `chapter-8/camel`: `java -jar ./target/node-runner-runnable.jar`
2. Put a JSON file into the `input` directory as described earlier. For example `mnp_input.txt` into `input` directory.
3. Run the `camel-direct-ingestion-runnable.jar` executable jar from the folder `chapter-8/camel` to run the camel Ignite streamer as follows: `java -jar ./target/camel-direct-ingestion-runnable.jar`

The following screenshot shows an example of running Camel streamer.

```
[15:49:03] Ignite node started OK (id=fc65e0f1)
[15:49:03] Topology snapshot [ver=3, servers=1, clients=1, CPUs=8, offheap=3.2GB, heap=7.1GB]
[15:49:03]   ^-- Node [id=FC65E0F1-5A67-4B17-AF94-96AD3C46A28A, clusterState=ACTIVE]
199 [main] INFO org.apache.camel.impl.converter.DefaultTypeConverter - Loaded 189 type converters
210 [main] INFO org.apache.camel.impl.DefaultCamelContext - Apache Camel 2.16.0 (CamelContext: camel-1)
210 [main] INFO org.apache.camel.management.ManagedManagementStrategy - JMX is enabled
367 [main] INFO org.apache.camel.impl.DefaultRuntimeEndpointRegistry - Runtime endpoint registry is in e
ing endpoints (cache limit: 1000)
367 [main] INFO org.apache.camel.impl.DefaultCamelContext - AllowUseOriginalMessage is enabled. If acces
turn this option off as it may improve performance.
367 [main] INFO org.apache.camel.impl.DefaultCamelContext - StreamCaching is not in use. If using stream
at http://camel.apache.org/stream-caching.html
367 [main] INFO org.apache.camel.impl.DefaultCamelContext - Total 0 routes, of which 0 is started.
368 [main] INFO org.apache.camel.impl.DefaultCamelContext - Apache Camel 2.16.0 (CamelContext: camel-1)
MsgBody>{"telephone" : "89096860917", "credit" : "129", "routeTo" : "MTS"}
```

**Figure 8.12**

Camel file adapter consumes the JSON file from the `input` directory and dispatch the data into the Ignite cache whenever you put a new file in the `input` directory. Now, we can scan the Ignite cache `camel-direct` by using the Ignite Visor tool to verify the cache entry.

```

Entries in cache: camel-direct
=====
+-----+
|   Key Class   |   Key    |       Value Class      |
|   Value        |          |       |
+-----+
+-----+
| java.lang.String | 89096860917 | o.a.i.i.binary.BinaryObjectImpl | com.blu.imdg.dto.MnpRouting [hash=-145921
6947, routeTo=MTS, telephone=89096860917, credit=129] |
+-----+
visor> ■
  
```

Figure 8.13

In this example above, we used only one JSON object to store entry into the cache for simplicity. However, you can modify the application to consume multiple JSON objects from one source file by using the `StreamMultipleTupleExtractor` extractor. In the next section, we will go through the mediated ingestion approach of the *Ignite-camel streamer* to streaming data into the Ignite cache.

## Mediated Ingestion

In real life, more often you need to do some complex computing or executing a few business logic on to streaming data rather than directly store them into the cache. The complex computing could be validating, splitting, routing or aggregating the incoming messages and ingest only the result of the pre-computed value into the Ignite cache. There are several basic patterns or components for message mediation, a more complex pattern can be built by combining the simple patterns. The basic mediation patterns are:

- Protocol switch
- Transform
- Enrich
- Aggregate
- Distribute
- Correlate

All of the above patterns can be combined into one Camel *route* or use separately whenever necessary. These patterns operate on one-way operations rather than on request-response pairs.

So far in this section, we have looked at minimal possibilities of Camel framework. In this section, we will modify our application for executing a few business logic on incoming data. We are going to validate the telephone number and the credit attributes from the JSON objects and finally store only those objects that passed the *validation* process. We are going

to use Camel *route* with a custom bean processor to achieve this functionality. Note that, validated objects will be dispatched to the `direct:ignite.ingest` endpoint, where the streamer is consuming the data. Now that we have discussed all the basics, let's start building our application.

**Step 1.** Create a new Java class named `RouteProcessor` into the `com.blu.imdg.processor` package and implements the Camel *Processor* interface as shown below.

Listing 8.37

---

```
public class RouteProcessor implements Processor {
    private static final long CREDIT_LIMIT = 100L;
    @Override
    public void process(Exchange exchange) throws Exception {
        MnpRouting mnpRouting = (MnpRouting) exchange.getIn().getBody();
        if(mnpRouting != null){
            // validate phone numbers of format "1234567890"
            if (!mnpRouting.getTelephone().matches("\\d{11}") || mnpRouting.getCredit() <\\
CREDIT_LIMIT){
                exchange.getOut().setBody("Message doesn't pass validation");
                exchange.getOut().setHeader("key", mnpRouting.getTelephone());
            } else{
                exchange.getOut().setBody(mnpRouting.toString());
                exchange.getOut().setHeader("key", mnpRouting.getTelephone());
            }
        }
    }
}
```

---

In the above code, business logic is pretty simple. We validate the `MnpRouting` objects: if the telephone number is **ambiguous** or the credit is less than **100**, then we pass the object and write an alert into the cache. Next, we will create another Java class for streaming data into the cache.

**Step 2.** Create another Java class named `CamelStreamerMediationIngestion` or modify the previous one, which we used for direct ingestion before.

Listing 8.38

```
public class CamelStreamerMediationIngestion {  
    private static final String FILE_PROPERTIES= "camel.properties";  
    public static void main(String[] args) throws Exception {  
        System.out.println("Camel Streamer Mediation ingestion!");  
        Ignition.setClientMode(true);  
        Ignite ignite = Ignition.start("example-ignite.xml");  
        if (!ExamplesUtils.hasServerNodes(ignite))  
            return;  
        if (getFileLocation() == null || getFileLocation().isEmpty()){  
            System.out.println("properties file is empty or null!");  
            return;  
        }  
        // camel_cache cache configuration  
        CacheConfiguration<String, String> camel_cache_cfg = new CacheConfiguration<>("ca  
mel-direct");  
        camel_cache_cfg.setIndexedTypes(String.class, String.class);  
        IgniteCache<String, String> camel_cache = ignite.getOrCreateCache(camel_cache_cfg\\  
);  
        // Create a streamer pipe which ingests into the 'camel_cache' cache.  
        IgniteDataStreamer<String, String> pipe = ignite.dataStreamer(camel_cache.getName\\  
());  
  
        pipe.setAutoFlushFrequency(1l);  
        pipe.allowOverwrite(true);  
  
        CamelStreamer<String, String> streamer = new CamelStreamer<>();  
        streamer.setIgnite(ignite);  
        streamer.setStreamer(pipe);  
        streamer.setEndpointUri("direct:ignite.ingest");  
        CamelContext context = new DefaultCamelContext();  
        context.addRoutes(new RouteBuilder() {  
            @Override  
            public void configure() throws Exception {  
                from("file://" + getFileLocation())  
                    .unmarshal().json(JsonLibrary.Jackson, MnpRouting.class)  
                    .bean(new RouteProcessor(), "process")  
                    .to("direct:ignite.ingest");  
            }  
        });  
        streamer.setCamelContext(context);  
        streamer.setSingleTupleExtractor(new StreamSingleTupleExtractor<Exchange, String,\\  
String>());  
    }  
}
```

```

@Override
public Map.Entry<String, String> extract(Exchange exchange) {
    String key = exchange.getIn().getHeader("key", String.class);
    String routeMsg = exchange.getIn().getBody(String.class);
    return new GridMapEntry<>(key, routeMsg);
}
});
streamer.start();
}
}

```

---

Most of the part of the above Java class has not changed, let us go through the most crucial part of this Java class. Here, we are using a new *endpoint* for the Ignite streamer: `direct:ignite.ingest`. Ignite streamer is now consuming data from the direct endpoint above. Next, we use the default Camel context and add one route to it. We are using camel *Java DSL* here for the ultimate clean and quick code.

```

context.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("file://" + getFileLocation())
            .unmarshal().json(JsonLibrary.JACKSON, MnpRouting.class)
            .bean(new RouteProcessor(), "process")
            .to("direct:ignite.ingest");
    }
});

```

Next, we consume the file from the *input* directory. Then, we *unmarshal* the JSON objects to Java objects and route to our `RouteProcessor` for validating the object. Note that, we are using `bean(new RouteProcessor(), "process")` bean binding approach, however, there are a few more ways to bind bean into the camel route. If you are curious about the Camel bean binding, please visit the Camel site for more information. After validating the objects, we dispatch it to Ignite streamer for storing into the cache.

**Step 3.** Let's compile and run the application to see it in action:

1. Build the project with `maven clean install` command.
2. Run the `node-runner-runnable.jar` executable jar to run an Ignite server node from the Maven project directory `chapter-8/camel`: `java -jar ./target/node-runner-runnable.jar`.

3. Put a few JSON files into the *input* directory as described earlier. Change the credit value less than 100.
4. Run the `camel-mediation-ingestion Runnable.jar` executable jar from the directory *chapter-8/camel* to run the camel Ignite streamer as follows: `java -jar ./target/camel-mediation-ingestion Runnable.jar`

```

541 [main] INFO org.apache.camel.impl.DefaultCamelContext - Route: route1 started and consuming from: Endpoint
[file:///Users/shamim/Development/workshop/github/the-apache-ignite-book/chapters/chapter-8/camel/input?delete
=true]
541 [main] INFO org.apache.camel.impl.DefaultCamelContext - Total 1 routes, of which 1 is started.
542 [main] INFO org.apache.camel.impl.DefaultCamelContext - Apache Camel 2.16.0 (CamelContext: camel-1) starte
d in 0.253 seconds
Process the bean MnpRouting!

```

Figure 8.14

We have added a few JSON files into the **input** directory with different credit information and telephone numbers. Camel file adapter starts consuming the files, and our custom Processor validates the objects. Let's verify the Ignite cache using the *ignitevisor* tool.

Entries in cache: camel-direct				
Key Class	Key	Value Class	Value	
java.lang.String	89096860917	Routing	{telephone='89096860917', credit=175, routeTo='VODAFON'}	
java.lang.String	89096860915	Message	doesn't pass validation	

Figure 8.15

We got, what we have expected, whenever the telephone number is not valid, or the credit is less than 100, we get an alert message into the cache *camel-direct*.

That's it, we have just completed our Camel Ignite streamer section. In the next section, we will deep dive into the Apache Flume: another framework to load a massive amount of data into the Ignite cache.

## Flume sink

Apache Flume<sup>163</sup> is a data ingestion mechanism for collecting, filtering, aggregating and transporting large amounts of streaming data such as logs, events from various sources to

---

<sup>163</sup><https://flume.apache.org/>

a centralized data store. A data store might be HDFS, RDBMS or in-memory data store. In plain English, words *flume* means channel. This tool is designed to manage the flow of the data, collected from different sources and send them to a centralized repository.

### Advantages:

- It is reliable, meaning if an event is introduced into a Flume event processing framework, it is guaranteed not to be lost.
- Flume supports any centralized data store, HDFS, RDBMS or In-memory data store. With customizing Flume sink, you can store data into any persistence store.
- It is scalable; by adding a greater number of processing agents, known as Flume agents, we can have scalability as far as processing is concerned. Thus, we can process more, if more agents are added.
- Flume provides the feature of contextual routing.
- When the rate of incoming data exceeds the rate at which data can be written to the destination, Flume acts as a mediator between data producers and the centralized stores, and provides a steady flow of data between them.
- Finally, it is feature rich and fully extensible; so we can extend the Flume framework to our needs.

Thus, Flume is a framework that is useful for moving data. What Flume lets us do is move data from *point A* to *point B*, and while moving the data, it lets us also transform the data. So, in that sense, it is more like an *ETL (Extract Transform Load)* tool.

### High-level architecture:

Let's describe the architecture of the Flume with the definitions of the basic concepts.

1. *Event*: A unit of data transmitted from place A to place B. An event has a payload and an optional set of string attributes.
2. *Flow*: The path of the movement of an event from place A to place B.
3. *Client*: Any application or source that ships the event to the Flume agent.
4. *Agent*: An independent Java process that hosts components such as sources, channels, and sinks; storing events and ship events to the next node.
5. *Source*: It's an interface, which can receive events from various sources through different protocols.
6. *Channel*: Temporary storage for events, the event is on the channel until the sink takes it.

7. *Sink*: Implementation of the interface, which takes events from the channel and transmits it to the destination. A Sink that conveys the event in the target store is called *final sink*. Examples of the final sink could be HDFS, Hive database or Solr.

Before we proceed to a more advanced topic, let's have a look at the high-level architecture of the Flume.

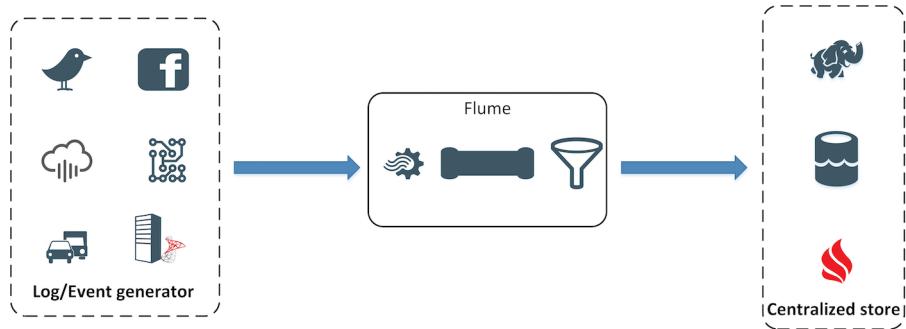


Figure 8.16

Historically, Flume is used for ingesting high volume of logs from various sources like Nginx web servers or Application servers to Hadoop file system, and it is more tightly integrated with the Hadoop ecosystems. So, it's a common use case is to act as a data pipeline to ingest a massive volume of data into the Hadoop file system. However, you can use Flume for ingesting any event data into any centralized store. For example, you can use Flume for ingesting transaction data from a credit card processing system like Way4 and store them into Apache Ignite for real-time analysis. Also, you can use Flume interceptor to do a variety of processing against incoming event as they pass through the Flume pipeline. For instance, you can calculate a basic *Travel score* or *anti-fraud* system to attempt to identify whether a bank customer is traveling while they are using their debit or credit card. The use case above is entirely fabricated, but Flume architecture can be used to apply virtually any online model or scoring while returning results in sub-second times.

Apache Ignite provides `IgniteSink` that extracts events from a flume channel and ingests into Ignite cache. At the time of writing of this book, IgniteSink support *Flume-1.7.0* version.

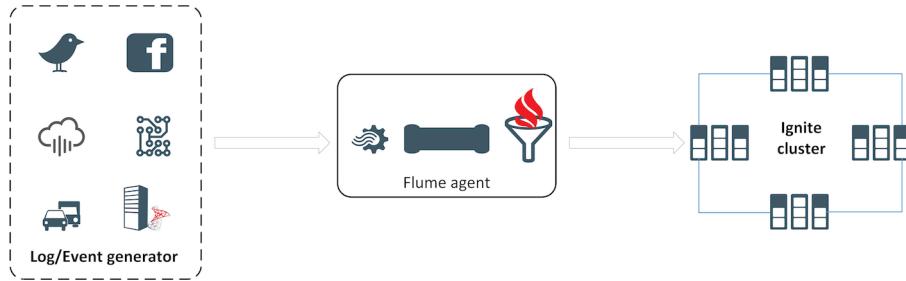


Figure 8.17

An *IgniteSink* is associated with precisely one channel as configured in the Flume configuration file. Also, there is one *SinkRunner* instance correlated with every configured sink, and when the Flume framework call *SinkRunner.start()* a new thread is created to run the *IgniteSink*.

Now that we have dipped our toes into Flume, let's build a (nearly) minimal example of IgniteSink to demonstrate the core functionality of the IgniteSink. To keep things simple, we are going to use two sources of events in our example: Linux Netcat utility and a custom RPC event generator. We will simulate ingesting transaction events into an Ignite cache. Through Netcat utility or RPC generator, we will generate transaction events in the format *string:string* for Flume source and the flume agent will ship this event through a channel to IgniteSink. IgniteSink will convert the list of events into cache entries.

**Step 1.** Create a Maven project and add the following IgniteSink Maven dependency into the pom.xml file. The source code of the entire porject is available at [GitHub<sup>164</sup>](#).

Listing 8.39

---

```

<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-flume</artifactId>
    <version>2.6.0</version>
</dependency>

```

---

**Step 2.** Firstly, we have to develop our own custom *EventTransformer* Java class by implementing the *EventTransformer* interface provided by IgniteSink. The main purpose of this class is to convert a list of Flume events to Ignite cache entries.

<sup>164</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-8/flume>



## Tip

Before implementing the interface *EventTransformer*, you have to specify the format and the type of the event. In our case, event type will be type *String*, and the format will be *String:String*.

**Listing 8.40**

---

```

public class FlumeEventTransformer implements EventTransformer <Event, String, Integer> {
    @Nullable
    @Override
    public Map<String, Integer> transform(List<Event> list) {
        final Map<String, Integer> map = new HashMap<>(list.size());
        for (Event event : list){
            map.putAll(transform(event));
        }
        return map;
    }
    /**
     * Event format - String:String
     * example - transactionId:amount [56102:232], where transactionId is the key and amount is the value
     */
    private Map<String, Integer> transform(Event event){
        final Map<String, Integer> map = new HashMap<>();
        String eventBody = new String(event.getBody());
        if(!eventBody.isEmpty()){
            // parse the string by the delimiter ":"
            String[] tokens = eventBody.split(":");
            map.put(tokens[0].trim(), Integer.valueOf(tokens[1].trim()));
        }
        return map;
    }
}

```

---

Let's have a detailed look at the above fragment of the code. First, we implemented the interface *EventTransformer* and override the method *transform*. We also overload the method *transform* with a single *Event*. In the overloaded method, we extract the event body as *String* and split the body by the delimiter `:`. The first part of the decoupled string will be our cache *key*, and the rest of the part will be the *value* of the cache entry.

**Step 3.** Compile and build the project with the following command:

**Listing 8.41**

---

```
mvn clean package
```

---

It will create a Jar file named *chapter-8/flume/target/flume-1.0-SNAPSHOT.jar*.

**Step 4.** Download the Apache Flume distributive from the following [page<sup>165</sup>](#). Apache Flume is distributed in several formats for your convenience. You can pick a ready-made binary distribution or a source archive if you intend to build Apache Flume yourself. In order to guard against corrupted downloads/installations, it is highly recommended to verify the signature of the release bundles against the public KEYS used by the Apache Flume developers.

**Step 5.** Ensure *JAVA\_HOME* environment variable is set and points to your JDK installation in your work station. In my case, I am using JVM 1.8. Extract the distribution archive into any directory.

**Listing 8.42**

---

```
tar xzvf apache-flume-1.7.0-bin.tar.gz
```

---

Move the content of the *apache-flume-1.7.0-bin.tar.gz* file to the directory */home/user/apache-flume-1.8.0-bin* as follows:

**Listing 8.43**

---

```
mv apache-flume-1.7.0-bin/* /home/user/apache-flume-1.7.0
```

---

In my case, */home/user/apache-flume-1.7.0* will be the *FLUME\_HOME* directory.

**Step 6.** Verify the installation of Apache Flume by browsing through the bin folder of the Flume home directory and typing the following command.

**Listing 8.44**

---

```
./flume-ng version
```

---

If you have successfully installed Flume, you will get the following information in your console as shown below:

---

<sup>165</sup><https://flume.apache.org/download.html>

Flume 1.7.0

Source code repository: <https://git-wip-us.apache.org/repos/asf/flume.git>

Revision: 99f591994468633fc6f8701c5fc53e0214b6da4f

Compiled by denes on Fri Sep 15 14:58:00 CEST 2017

From source with checksum fbb44c8c8fb63a49be0a59e27316833d

**Step 7.** Create a directory *plugins.d* inside the Flume home directory. In our case it should be /home/user/apache-flume-1.7.0/plugins.d. Create a *ignite-sink* directory under *plugins.d* directory. Create two more directories under *ignite-sink* directory:

- lib
- libext

Copy the previously build file *flume-1.0-SNAPSHOT.jar* to \${FLUME\_HOME}/plugins.d/ignite-sink/lib. Also copy other *Ignite-related* libraries files from the Apache Ignite distribution to \${FLUME\_HOME}/plugins.d/ignite-sink/libext folder as shown below.

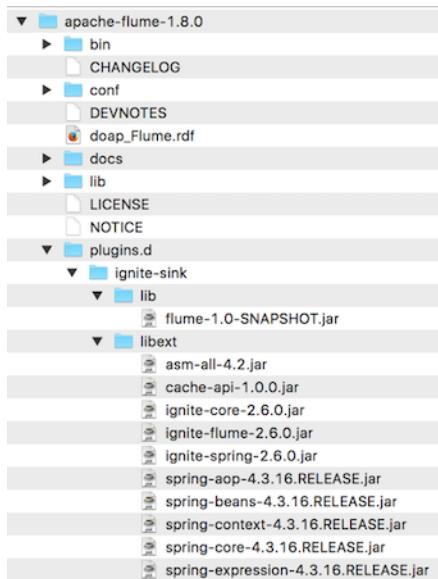


Figure 8.18



## Tip

Ignite documentation left the *asm-all-\*.jar* file which is also needed to run the Ignite sink with Flume agent.

**Step 8.** Copy the file `example-ignite.xml` and `demo.properties` from `/chapters/chapter-8/flume/src/main/java/com/blu/imdg/flume/config` to  `${FLUME_HOME}/conf` directory. Note that, in the `example-ignite.xml` file you have to specify the Ignite **cache name**, which will be the same as the cache name from the file `demo.properties`. In our case, the cache name is the `testCache`. At this moment, we have to configure the Flume agent properties into the file `demo.properties`, which is a Java property file having *key-value* pairs. In the Flume configuration file, we need to configure the following components:

- Name of the components for the current agent.
- A configuration of the source..
- A configuration of the channel.
- A configuration of the sink.
- Binding the source and the sink to the channel.

Generally, we can have multiple agents in the Flume. We can differentiate each agent by the unique name. Let's have a detailed look at the `demo.properties` file.

First of all, you have to give a name to the components such as sources, sinks, and the channels of the agent, as shown below.

```
# Name the components for this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1
```

Where, `a1` is the name of the agent and, `r1` is the name of the source and so on. Next, we have to configure the agent source; every source will have a separate list of properties.

```
# Describe/configure the source
#a1.sources.r1.type = netcat
a1.sources.r1.type = avro
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444
```

Property `type` is one of the most common properties of the source: it can be *Netcat*, *Avro* or *Twitter*. Also, the source should bind to a *Host* and with a *Port*. We are using *localhost* with port `44444`. In our example, we will use the source *Avro* to generate events.

Flume out-of-the-box provides a bunch of different channels to transfer data between sources and sinks. You have to set the required properties to describe the channel, as follows:

```
# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
```

Here, the maximum channel capacity of events is 1000 entries, and the maximum number of events stored in the channel per transaction is 100. Like source, sink must be configured in a separate list. The property named `type` is common to every sink, and it is used to specify the type of the sink we are using. Along with the property `type`, it also needs to provide the values of all the required properties of a particular sink as shown below.

```
# ignite sink
a1.sinks.k1.type = org.apacheignite.stream.flume.IgniteSink
a1.sinks.k1.igniteCfg = $FLUME_HOME/conf/example-ignite.xml
a1.sinks.k1.cacheName = testCache
a1.sinks.k1.eventTransformer = com.blu.imdg.flume.Transformer.FlumeEventTransformer
a1.sinks.k1.batchSize = 100
```

Here, the type of the sink is `org.apacheignite.stream.flume.IgniteSink`. The configuration of the Ignite sink is located in `$FLUME_HOME/conf` directory. Name of the cache for ingesting events is `testCache` and the event transformer class is our custom `FlumeEventTransformer` class.

So far, we have configured the source, channel, and the sink. The Flume sink channel connects the source and the sink together; it is required to bind both of them to channel as follows.

```
# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

**Step 9.** Now that we have configured all the necessary configuration and we are ready to run the Flume agent. First, we will have to start an Ignite node and then start the flume agent by executing the following command:

#### **Listing 8.45**

---

```
ignite.sh ~/the-apache-ignite-book/chapters/chapter-8/flume/src/main/java/com/blu/imdg/flume/config/example-ignite-server.xml
```

---

Now, start the Flume agent in a separate console from the `FLUME_HOME` directory.

**Listing 8.46**

---

```
bin/flume-ng agent --conf conf --conf-file /Users/shamim/Development/bigdata/apache-flume\1.7.0/conf/demo.properties --name a1 -Dflume.root.logger=INFO,console
```

---

If everything goes well, you should get a similar output in the console as shown below.

```
2018-10-08 20:22:40,502 [lifecycleSupervisor-1-0] [INFO - org.apache.flume.source.NetcatSource.start(NetcatSource.java:169)] Created serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:44444]
2018-10-08 20:22:40,693 [lifecycleSupervisor-1-1] [INFO - org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)] Loading XML bean definitions from URL [file:/Users/shamim/Development/bigdata/apache-flume-1.7.0/conf/example-ignite.xml]
2018-10-08 20:22:40,988 [lifecycleSupervisor-1-1] [INFO - org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:583)] Refreshing org.springframework.context.support.GenericApplicationContext@4af4677: startup date [Mon Oct 08 20:22:40 MSK 2018]; root of context hierarchy
[20:22:41]
[20:22:41]   / \   / \   / \   / \   / \
[20:22:41]   \ / \ / \ / \ / \ / \
[20:22:41]   / \ / \ / \ / \ / \
[20:22:41]
[20:22:41] ver. 2.6.0#20180710-sha1:669feacc
[20:22:41] 2018 Copyright(C) Apache Software Foundation
```

Figure 8.19

A Flume agent is started with an embedded Ignite client node. The Flume agent starts running the Flume source and sinks configured in the `demo.properties` file. A single agent named `a1` has been started with a source that listens for data on port `44444`. Also a channel up and running that buffers event data in memory, and a sink that stores data into the Ignite cache.

Now, we can run our Avro event generator (RPC) class to generate events. Run the `com.blu.imdg.flume.rpc.RunClient` Java class, with the following command.

**Listing 8.47**

---

```
java -jar ./target/flume-avro-client.jar
```

---

An Avro RPC client starts and generates 99 events to the Flume agent source. The format of the event is `transactionId: amount` as described earlier of this section.

Start an Ignite Visor tool and scan the cache named `testCache`, you should discover 99 generated events into the cache as shown in figure 8.20.

```
visor> cache -scan
Time of the snapshot: 2018-10-08 20:48:26
+-----+
| # | Name(@) | Mode | Size (Heap / Off-heap) |
+-----+
| 0 | testCache(@c0) | PARTITIONED | min: 99 (0 / 99) |
| | | | avg: 99.00 (0.00 / 99.00) |
| | | | max: 99 (0 / 99) |
+-----+
```

Figure 8.20

Next, we are going to **reconfigure** the Flume agent source to use Linux Netcat tool and generate some events by using `nc` tool for ingesting events into the Ignite cache. To keep things simple, we will edit the agent source section into our `demo.properties` file as shown below.

Listing 8.48

---

```
# Describe/configure the source
a1.sources.r1.type = netcat
#a1.sources.r1.type = avro
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444
```

---

That's it! We have disabled Avro source section in the properties file and enable the Flume source Netcat. Netcat is a simple Unix tool that reads and writes data across a network connection, widely used as a debugging and exploration tool. The rest of the configurations in the file will be untouched. Let's re-execute the Flume agent as follows.

```
bin/flume-ng agent --conf conf --conf-file /Users/shamim/Development/bigdata/apache-flume\1.7.0/conf/demo.properties --name a1 -Dflume.root.logger=INFO,console
```

You should notice that a Netcat source has been started on the specified host and ports in the console as shown in the figure 8.21.

```

2018-10-08 20:54:51,340 [lifecycleSupervisor-1-0] [INFO - org.apache.flume.source.NetcatSource.start(NetcatSource.java:169)] Created serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:44444]
2018-10-08 20:54:51,505 [lifecycleSupervisor-1-3] [INFO - org.springframework.beans.factory.xml.XmlBeanDefinitionReader.loadBeanDefinitions(XmlBeanDefinitionReader.java:317)] Loading XML bean definitions from URL [file:/Users/shamim/Development/bigdata/apache-flume-1.7.0/conf/example-ignite.xml]
2018-10-08 20:54:51,813 [lifecycleSupervisor-1-3] [INFO - org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractApplicationContext.java:583)] Refreshing org.springframework.context.support.GenericApplicationContext@6ef1d1af: startup date [Mon Oct 08 20:54:51 MSK 2018]; root of context hierarchy
[20:54:52] _____
[20:54:52] / \ / \ / \ / \
[20:54:52] / / (7 7) / / / / /
[20:54:52] / \ \ / | \ / \ / \ /
[20:54:52] ver. 2.6.0#20180710-sha1:669feacc

```

Figure 8.21

Let's start a new command line console and execute the following command.

Listing 8.49

---

```
nc YOUR_HOST 44444
```

---

At this moment, you can send any command by entering key *ENTER*. Let's type `transaction1:101` and *ENTER*. Now, have a look into the cache *testCache* by *ignitevisor* CLI to make sure that our event has been ingested into the Ignite cache.

java.lang.String   transaction6   <u>java.lang.Integer</u>   6
java.lang.String   transaction9   java.lang.Integer   9
java.lang.String   transaction8   java.lang.Integer   8
java.lang.String   transaction1   java.lang.Integer   101
java.lang.String   transaction3   java.lang.Integer   3
java.lang.String   transaction2   java.lang.Integer   2

Figure 8.22

In the *ignitevisor* console, we can confirm our transaction event in the cache. The Netcat tool is handy to debug event processing or ingestion data processing into the Ignite cache. In the next section of this chapter, we are going to describe one of the popular complex event processing framework for streaming data into the Ignite cache.

## Storm streamer

Apache Storm<sup>166</sup> is a distributed fault-tolerant real-time computing system. In the era of *IoT* (*Internet Of Things - the next big thing*), many companies regularly generate terabytes of data in their daily operations. The sources include everything from network sensors, vehicles,

<sup>166</sup><http://storm.apache.org/>

physical devices, the web, social data or transactional business data. With the volume of data being generated, real-time computing has become a significant challenge for most of the organization.

In a short time, Apache Storm became a standard for the distributed real-time processing system that allows you to process a large amount of data. Apache Storm project is open source and written in *Java* and *Cljure*. It became the first choice for real-time analytics. Apache Ignite *Storm streamer* module provides a convenient way to streaming data via Storm to Ignite cache.

Although Hadoop/Spark and Storm frameworks are used for analyzing and processing big data, both of them complement each other and differs in a few aspects. Like Hadoop, Storm can process a massive amount of data but does it in real-time with guaranteed reliability, meaning every message will be processed. It has these advantages as well:

1. Simple scalability, to scale horizontally, you simply add machines and change parallelism settings of the topology.
2. Stateless message processing.
3. It guarantees the processing of every message from the source.
4. It has fault tolerance.
5. The topology of Storm can be written in any languages, although mostly Java is used.

## Key concepts.

Apache Storm reads raw streams of data from the one end and passes it through a sequence of small processing units and output the processed information at the other end. Let's have a detailed look at the main components of Apache Storm:

*Tuples* – It is the main data structure of Storm. It's an ordered list of elements. Generally, tuple supports all primitives data types.

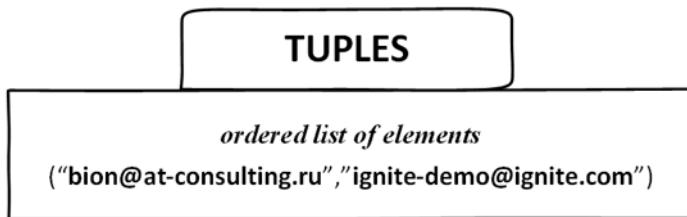


Figure 8.23

*Streams* – It's an unbound and unordered sequence of tuples.

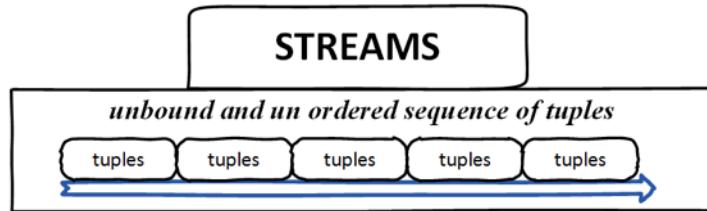


Figure 8.24

*Spouts* - Source of the streams; in other words, a spout reads the data from a source for use in the topology. A spout can be reliable or unreliable. A spout can talk with Queues, Weblogs, event data, etc.

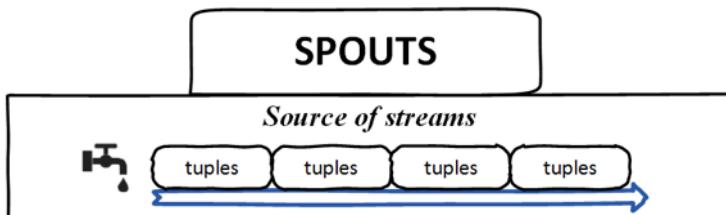


Figure 8.25

*Bolts* - Bolts are logical processing units, it is responsible for processing data and creating new streams. Bolts can perform the operations of filtering, aggregation, joining and interacting with files/database and so on. Bolts receive data from the spout and emit to one or more bolts.

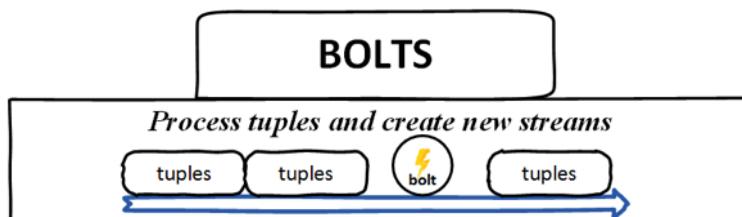


Figure 8.26

*Topology* – A topology is a directed graph of Spouts and Bolts, each node of this graph contains the data processing logic (bolts) while connecting edges define the flow of the data (streams).

Unlike Hadoop/Spark, Storm keeps the topology running forever until you kill it. A simple topology starts with spouts, emits stream from the sources to bolt for processing data. Apache Storm's main job is to run any topology at a given time.

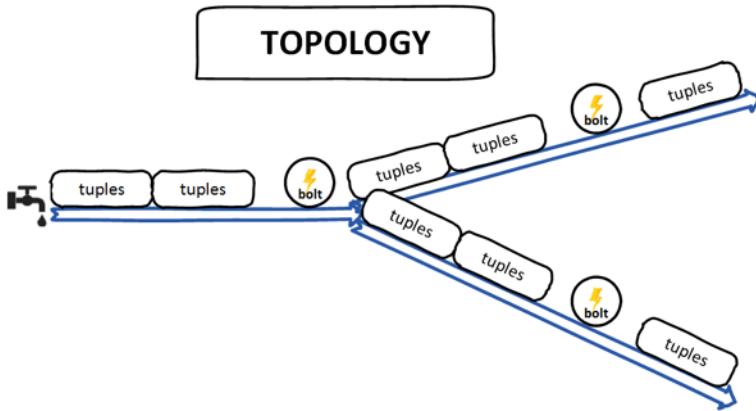


Figure 8.27

Ignite out-of-the-box provides an implementation of *Storm Bolt (StormStreamer)* for streaming the computed data into the Ignite cache. On the other hand, you can write your custom Strom Bolt to ingest stream data into Ignite. To develop a custom Storm Bolt, you have to implement `BaseBasicBolt` or `IRichBolt` Storm interface. However, you have to configure a few properties to work with the Ignite Bolt correctly if you decide to use `StormStreamer`. All mandatory properties are shown below:

Nº	Property Name	Description
1	CacheName	Cache name of the Ignite cache, in which the data will be stored.
2	IgniteTupleField	Names of the Ignite Tuple field, by which tuple data is obtained in topology. By default the value is <i>ignite</i> .
3	IgniteConfigFile	This property will set the Ignite spring configuration file. This allows you to send and consume message to and from Ignite topics.
4	AllowOverwrite	It will enable overwriting existing values in the cache; default value is <i>false</i> .
5	AutoFlushFrequency	Automatic flush frequency in <i>milliseconds</i> . Essentially, this is the time after which the streamer will make an attempt to submit all data added to remote nodes. The default value is <i>10 sec</i> .

Now, let's build something tangible to check how the Ignite *StormStreamer* works. The basic idea behind the application is to design one topology of spout and bolt that can process a massive amount of data from a traffic log files and trigger an alert when a specific value crosses a predefined threshold. By using a topology, the log file will be consumed *line by line*, and the topology is designed to monitor the incoming data. In our case, the log file will contain data, such as *vehicle registration number*, *speed* and the *highway name* from highway traffic camera. If the vehicle crosses the *speed limit* (for instance, 120km/h), Storm topology will send the data to an Ignite cache.

Next listing will show a CSV file of the type we are going to use in our example, which contain vehicle data information such as vehicle registration number, the speed at which the vehicle is traveling and the location of the highway.

**Listing 8.50**

---

```
AB 123, 160, North city
BC 123, 170, South city
CD 234, 40, South city
DE 123, 40, East city
EF 123, 190, South city
GH 123, 150, West city
XY 123, 110, North city
GF 123, 100, South city
PO 234, 140, South city
XX 123, 110, East city
YY 123, 120, South city
ZQ 123, 100, West city
```

---

The idea of the above example is in reference to the *Dr. Dobbs* journal. Since this book is not for studying Apache Storm, I am going to keep the example as simple as possible. Also, I have added the famous word count example of Storm, which ingests the word count value into Ignite cache through StormStreamer module. If you are curious about the code, it's available at [GitHub](#)<sup>167</sup>.

---

<sup>167</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-8/storm>

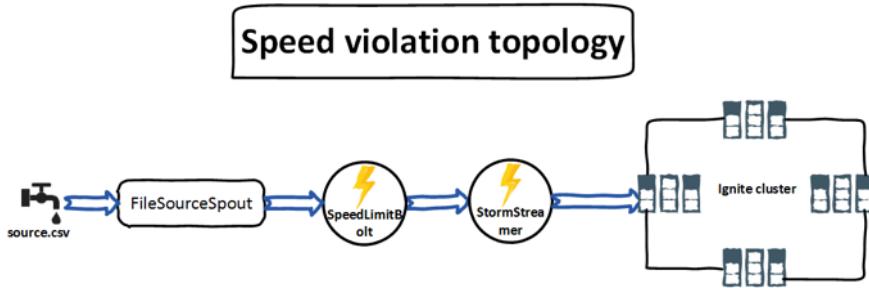


Figure 8.28

As shown in figure 8.28 above, the *FileSourceSpout* accepts the input CSV log file, reads the data line by line and emits the data to the *SpeedLimitBolt* for further threshold processing. Once the processing is done and found any car with exceeding the speed limit, the data is emitted to the Ignite *StormStreamer bolt*, where it is ingested into the cache. Let's dive into the detailed explanation of our Storm topology.

**Step 1.** You must add the Storm and the Ignite *StormStreamer* dependency in the Maven project because it is a Storm topology.

Listing 8.51

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-storm</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>1.1.1</version>
</dependency>
<exclusions>
    <exclusion>
        <groupId>log4j</groupId>
```

```
<artifactId>log4j</artifactId>
</exclusion>
<exclusion>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
</exclusion>
<exclusion>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
</exclusion>
<exclusion>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
</exclusion>
<exclusion>
    <groupId>org.slf4j</groupId>
    <artifactId>log4j-over-slf4j</artifactId>
</exclusion>
<exclusion>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
</exclusion>
</exclusions>
</dependency>
```

---

At the time of writing this book, Apache Storm version 1.1.1 is the only supported version.



## Info

You do not need any *Kafka* module to run or execute this example.

**Step 2.** Create an Ignite configuration file (see `example-ignite.xml` file in the folder `/chapter-8/storm/src/resources`) and ensure that it is available from the classpath. The content of the Ignite configuration is identical from the previous section of this chapter.

**Listing 8.52**

---

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">
    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <!-- Enable client mode. -->
        <property name="clientMode" value="true"/>

        <property name="cacheConfiguration">
            <list>
                <bean class="org.apache.ignite.configuration.CacheConfiguration">
                    <property name="atomicityMode" value="ATOMIC"/>
                    <property name="name" value="testCache"/>
                </bean>
            </list>
        </property>

        <property name="discoverySpi">
            <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
                <property name="ipFinder">
                    <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
                        <property name="addresses">
                            <list>
                                <value>127.0.0.1:47500..47509</value>
                            </list>
                        </property>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

---

**Step 3.** Create an `ignite-storm.properties` file to add the cache name, tuple name, and the name of the Ignite configuration file as shown below.

**Listing 8.53**

---

```
cache.name=testCache
tuple.name=ignite
ignite.spring.xml=example-ignite.xml
```

---

**Step 4.** Next, create the *FileSourceSpout* Java class as shown below.

**Listing 8.54**

---

```
public class FileSourceSpout extends BaseRichSpout {
    private static final Logger LOGGER = LogManager.getLogger(FileSourceSpout.class);
    private SpoutOutputCollector outputCollector;
    @Override
    public void open(Map map, TopologyContext topologyContext, SpoutOutputCollector spout\
OutputCollector) {
        this.outputCollector = spoutOutputCollector;
    }
    @Override
    public void nextTuple() {
        try {
            Path filePath = Paths.get(this.getClass().getClassLoader().getResource("sourc\
e.csv").toURI());
            try(Stream<String> lines = Files.lines(filePath)){
                lines.forEach(line ->{
                    outputCollector.emit(new Values(line));
                });
            } catch(IOException e){
                LOGGER.error(e.getMessage());
            }
        } catch (URISyntaxException e) {
            LOGGER.error(e.getMessage());
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("trafficLog"));
    }
}
```

---

The *FileSourceSpout* code has three important methods:

- **open()**: This method would be called at the start of the spout and will give you the context information.

- **nextTuple()**: This method would allow you to pass one tuple to Storm topology for processing at a time. In this method, we are reading the CSV file line by line and emitting the line as a tuple to the bolt.
- **declareOutputFields()**: This method declares the name of the output tuple. In our case, the name should be *trafficLog*.

**Step 5.** Now create a *SpeedLimitBolt.java* Java class which implements the *BaseBasicBolt* interface.

**Listing 8.55**

---

```
public class SpeedLimitBolt extends BaseBasicBolt {
    private static final String IGNITE_FIELD = "ignite";
    private static final int SPEED_THRESHOLD = 120;
    private static final Logger LOGGER = LogManager.getLogger(SpeedLimitBolt.class);
    @Override
    public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
        String line = (String)tuple.getValue(0);
        if(!line.isEmpty()){
            String[] elements = line.split(",");
            //we are interested in speed and the car registration number
            int speed = Integer.valueOf(elements[1].trim());
            String car = elements[0];
            if(speed > SPEED_THRESHOLD){
                TreeMap<String, Integer> carValue = new TreeMap<String, Integer>();
                carValue.put(car, speed);
                basicOutputCollector.emit(new Values(carValue));
                LOGGER.info("Speed violation found:" + car + " speed:" + speed);
            }
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields(IGNITE_FIELD));
    }
}
```

---

Let's go through it line by line:

- *execute()*: This is the method where you implement the business logic of your bolt. In this case, we split the line by the comma (,), and check the speed limit of the car. If the

speed limit of the given car is higher than the threshold, we are creating a new treemap data type from this tuple and emit the tuple to the next bolt. In our case, the next bolt will be the StormStreamer.

- *declareOutputFields()*: This method is similar to *declareOutputFields()* method in the *FileSourceSpout*. It declares that it is going to return Ignite tuple for further processing.



## Warning

The tuple name `IGNITE` is important here, the StormStreamer will only process the tuple with the name `ignite`.

Note that, Bolts can do anything, for instance, computation, persistence or talking to external components.

**Step 6.** It's the time to create our topology to run our example. Topology ties the spouts and bolts together in a graph, which defines how the data flows between the components. It also provides parallelism hints that Storm uses when creating instances of the components within the cluster. To implement the topology, create a new Java class named *SpeedViolationTopology* in the `src\main\java\com\blu\imdg\storm\topology` directory. Use the following as the contents of the file:

Listing 8.56

---

```
public class SpeedViolationTopology {
    private static final int STORM_EXECUTORS = 2;

    public static void main(String[] args) throws Exception {
        if (getProperties() == null || getProperties().isEmpty()) {
            System.out.println("Property file <ignite-storm.property> is not found or empty");
            return;
        }
        // Ignite Stream Ibolt
        final StormStreamer<String, String> stormStreamer = new StormStreamer<>();

        stormStreamer.setAutoFlushFrequency(10L);
        stormStreamer.setAllowOverwrite(true);
        stormStreamer.setCacheName(getProperties().getProperty("cache.name"));

        stormStreamer.setIgniteTupleField(getProperties().getProperty("tuple.name"));
        stormStreamer.setIgniteConfigFile(getProperties().getProperty("ignite.spring.xml"));
```

```
});
```

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new FileSourceSpout(), 1);
builder.setBolt("limit", new SpeedLimitBolt(), 1).fieldsGrouping("spout", new FieldIds("trafficLog"));
// set ignite bolt
builder.setBolt("ignite-bolt", stormStreamer, STORM_EXECUTORS).shuffleGrouping("limit");
Config conf = new Config();
conf.setDebug(false);
conf.setMaxTaskParallelism(1);
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("speed-violation", conf, builder.createTopology());
Thread.sleep(10000);
cluster.shutdown();
}

private static Properties getProperties() {
    Properties properties = new Properties();
    InputStream ins = SpeedViolationTopology.class.getClassLoader().getResourceAsStream("ignite-storm.properties");
    try {
        properties.load(ins);
    } catch (IOException e) {
        e.printStackTrace();
        properties = null;
    }
    return properties;
}
```

---

Let's go through the code line by line again. First, we read the *ignite-storm.properties* file to get all the necessary parameters to configure the StormStreamer bolt. The storm topology is a Thrift structure. The *TopologyBuilder* class provides a simple and elegant way to build complex Storm topology. The *TopologyBuilder* class has two methods: *setSpout()* and *setBolt()*. Next, we used the Topology builder to build the Storm topology and added the spout with name *spout* and parallelism hint of 1 executor. We also define the *SpeedLimitBolt* to the topology with parallelism hint of 1 executor. Next, we set the StormStreamer bolt with *shufflegrouping*, which subscribes to the bolt, and evenly distributes tuples (*limit*) across the instances of the StormStreamer bolt.

For demonstration purpose, we created a local cluster using *LocalCluster* instance and submitted the topology using the *submitTopology* method. Once the topology is submitted to the cluster, we will wait 10 seconds for the cluster to compute the submitted topology and then shut down the cluster using the *shutdown()* method of the *LocalCluster*.

**Step 7.** Next, run an Ignite node.

**Listing 8.57**

---

```
ignite.sh ~/the-apache-ignite-book/chapters/chapter-8/flume/src/main/java/com/blu/imdg/flume/config/example-ignite-server.xml
```

---

After building the maven project, use the following command to run the topology locally.

**Listing 8.58**

---

```
mvn compile exec:java -Dstorm.topology=com.blu.imdg.storm.topology.SpeedViolationTopology
```

---

The Storm application will produce a lot of system logs as follows.

```
64204 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:AB 123 speed:160
64204 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:BC 123 speed:170
64204 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:EF 123 speed:190
64204 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:GH 123 speed:150
64204 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:PO 234 speed:140
64205 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:AB 123 speed:160
64205 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:BC 123 speed:170
64205 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:EF 123 speed:190
64205 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:GH 123 speed:150
64205 [Thread-21-limit-executor[3 3]] INFO c.b.i.s.b.SpeedLimitBolt - Speed violation found:PO 234 speed:140
```

Figure 8.29

Now, if we verify the Ignite cache through *ignitevisor* CLI tool, we should get the following output into the console.

Entries in cache: testCache				
Key Class	Key	Value Class	Value	
java.lang.String   PO 234   java.lang.Integer   140				
java.lang.String   BC 123   java.lang.Integer   170				
java.lang.String   AB 123   java.lang.Integer   160				
java.lang.String   GH 123   java.lang.Integer   150				
java.lang.String   EF 123   java.lang.Integer   190				

Figure 8.30

The output shows the result of what was expected. From our `source.csv` log file, only five vehicles exceed the speed limit of 120 km/h. This is pretty much sums up the practical overview of the Ignite Storm Streamer.

## Summary

In this chapter, we became familiar with the Real-time data streaming and complex event processing. We went through the theory and practice of the following Ignite Data streamer components:

- Kafka Streamer.
- Camel Streamer.
- Flume Streamer.
- Storm Streamer.

For each data streamer, we provided a complete example and showed how to develop real-time data processing application with these Ignite modules. We also provide a practical example of real-time data replication between Ignite clusters by using Kafka.

## What's next?

In the next chapter, we will go through one more advanced features and extensions to the Ignite platform. We will discuss the main problems of the Hadoop ecosystems and how Ignite can help to improve the performance of the Apache Spark jobs.

# Chapter 9. Accelerating Big data computing

Hadoop and Spark have quickly become the standard for business intelligence on massive datasets. However, Hadoop's batch scheduling overhead and disk-based data storage have made it unfit for use in analyzing live, real-time data in a production environment. On the other hand, Spark is fast and optimized for real-time computing but cannot share state between different Spark Jobs or Spark context. Apache Ignite offers a set of useful components to solve these problems; allowing Hadoop job to executing in memory and use the Ignite in-memory file system as a primary caching layer to store HDFS file which can dramatically improve the Hadoop Map/Reduce job. Also, Ignite also provides an implementation of Spark *RDD* and *DataFrames* to share state in-memory across multiple Spark jobs to accelerate jobs performance.

In this chapter, we will explore how Big data computing with Apache Spark can be performed by combining an in-memory data grid, and how Ignite can speed up the analysis of large datasets by avoiding data movement across the network from Ignite to Spark.

## Ignite for Apache Spark

Apache Ignite provides several ways to improve the Apache Spark job's performance:

- **Ignite RDD:** represents the Ignite cache as Spark *RDD* abstraction. *Ignite RDD* allows sharing states easily in-memory between different Spark jobs or applications.
- **Ignite DataFrames:** an extension of the Spark *dataframes*, which simplifies the deployment and improving data access times whenever Ignite is used as memory-centric storage for Spark.
- **Ignite IGFS (in-memory file system):** which can be transparently plugged into the Spark deployments. IGFS delivers similar functionality to Hadoop HDFS, but only in memory. It can substitute Hadoop HDFS or can be deployed on top of HDFS, in which case it becomes a transparent caching layer for files stored in HDFS.

In this section, we are going to analyse the steps on how to use *Ignite RDD* and *Ignite DataFrames* to share data and states across Spark jobs by writing and reading RDD/-DataFrames to/from Ignite. A high-level view of using Ignite RDD/DataFrames is shown in figure 9.1 below.

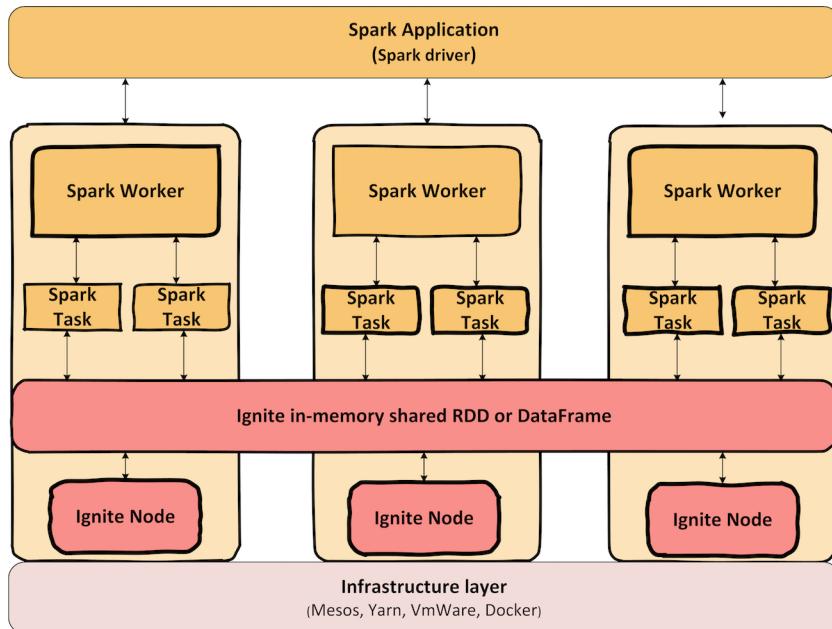


Figure 9.1

With Ignite in-memory shared RDD/DataFrame, any Spark job can put some data into Ignite cache which will be accessible by another Spark jobs later. Ignite RDD/DataFrame is implemented as a view over Ignite distributed cache, which may be deployed either within the Spark job execution process or on a Spark worker. Before we proceed to more advanced topics, let's review the history of the Spark and which problems can solve by the Ignite RDD/DataFrames.

## Apache Spark – a short history

Apache Spark<sup>168</sup> was invented by *AMPLab* for fast computing, built on top of the Hadoop Map/Reduce and it extends the Map/Reduce model to efficiently use operations like Interactive queries, and Stream processing. The Main difference between Spark and Hadoop Map/Reduce is that during execution Spark tries to keep data into memory, whereas Hadoop Map/Reduce shuffling data in and out of disk. For Hadoop Map/Reduce it takes a significant time to write intermediate data to disk and read them back. The elimination of this redundant disk operations makes Spark magnificently faster. Spark can store data (intermediately) into memory without any I/O, so you can keep operating on the same data very quickly.

Apache Spark provides a few particular API's (datasets) to store data into memory: Spark *RDD*, *DataFrames* and newly added *Datasets* - available in Apache [Spark 2.2](#)<sup>169</sup> and beyond. The beginning of the three API's is somewhat described below:

RDD (Spark 1.0) -> Data Frame (Spark 1.3) -> Dataset (Spark 1.6).

Let's start with the Spark RDD.

### Spark RDD

Spark RDD stands for Spark Resilient Distributed Dataset. Spark RDD are fundamental components of the Apache Spark for large-scale data processing framework. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions. You can imagine Spark RDD as a [Hadoop HDFS](#)<sup>170</sup> in memory.

Spark RDD supports two types of operations: *transformation*, which creates a new dataset from an existing one, and *actions*, which returns a value by performing a computation on the RDD as shown in figure 9.2 below.

---

<sup>168</sup><http://spark.apache.org>

<sup>169</sup><https://spark.apache.org/releases/spark-release-2-2-0.html>

<sup>170</sup>[https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html)

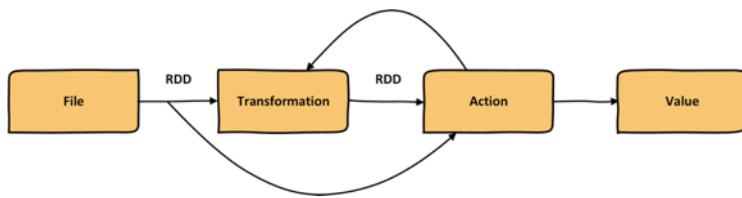


Figure 9.2

Spark RDD is created with the use of Spark transformation functions. Spark transformation functions can create Spark RDD from various sources, such as a text file. Also, Spark RDD may be created from external storage such as RDBMS, HBase, Cassandra or any other data source compatible with Hadoop input format.

Spark RDD can quickly and efficiently process both structured and unstructured data, but there are a few disadvantages for Spark RDD. Firstly, there is no built-in optimization engine for RDD. RDD's do not take advantage of Spark's advanced optimizers (catalyst optimizer and Tungsten execution engine) when working with structured data. Also, RDD's cannot share *states (jobs)* across Spark Jobs or SparkContext, because RDD is bound to a Spark application strictly. With the native Spark distribution, the only way to share RDD's between different Spark jobs is to write the dataset into HDFS or somewhere in the file system and then pull the RDD's within the other jobs.

### Spark DataFrames

DataFrames are immutable distributed collections of data in which the data is organized in a *relational manner*. Data is organized into named columns Unlike an RDD; like a table in a relational database, but with richer optimizations under the hood. DataFrames can be constructed from a wide variety of sources such as structured data files, tables in Hive, external databases, or *existing RDDs*.

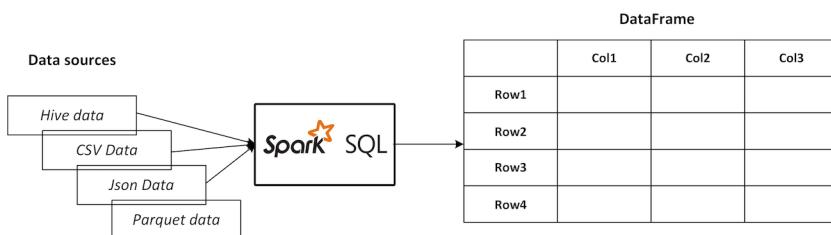


Figure 9.3

Spark Dataframes empower *Spark SQL queries* but also has a few disadvantages. Spark DataFrame API cannot share states between multiple Spark jobs and does not support compile-time safety for analysis, which limits the user during manipulating data when the

structure of the data is unknown.

## Spark Datasets

Starting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a *strongly-typed API* and an *untyped API*. A DataFrame can be seen as a collection of generic type Dataset [Row], where the Row can be a generic and untyped JVM object. Datasets are by default a collection of strongly typed JVM objects unlike DataFrames. In Java, they are mapped by class. In Scala, they are governed by case class.

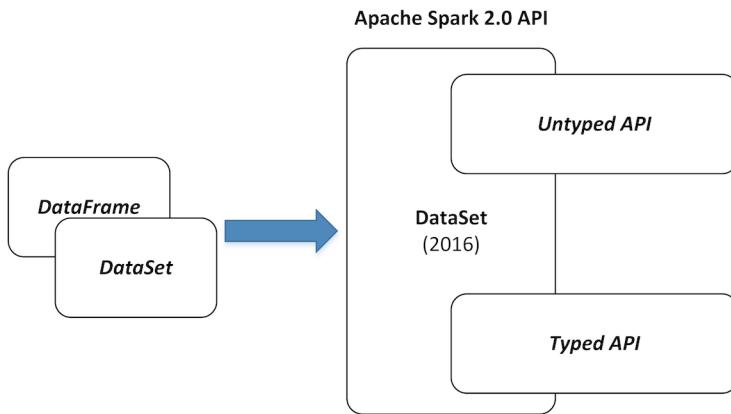


Figure 9.4

The main advantage of the Spark DataSets is that Spark DataSets provide static-type and runtime-type safety, which allow the developer to catch errors at compile time. Note that, DataFrames also support compile-time syntax error checking.

Apache Ignite as a memory-centric distributed database is complementary with Spark technology, and combining these two technologies it provides Spark users some significant benefits:

1. Share RDD/DataFrames across multiple Spark jobs.
2. Boost DataFrame and Spark SQL performances.
3. Avoiding data movement from a data source to Spark workers and applications.
4. Fast loading of massive amounts of data into Ignite cluster from different sources.

In this section we will focus on *Ignite RDD* and its implementation. In the next section of this chapter we will discuss the Ignite *DataFrame API*.

## Ignite RDD

Apache Ignite provides [IgniteRDD<sup>171</sup>](#) API to share RDD between Spark applications. IgniteRDD is an implementation of Spark RDD abstraction representing a live view of Ignite cache. IgniteRDD is not immutable, all changes in Ignite cache (regardless whether they were caused by another RDD or external changes in cache) will be visible to RDD users immediately.

IgniteRDD utilizes partitioned nature of Ignite caches and provides partitioning information to Spark executor. A number of partitions in IgniteRDD equals to the number of partitions in underlying Ignite cache. IgniteRDD also provides affinity information to Spark via `getPreferredLocations()` method so that RDD computations use data locality.

Apart from that, Ignite also provides [IgniteContext<sup>172</sup>](#) API, which is the main entry point to the Spark-Ignite integration. A user must provide an instance of the `SparkContext` and a closure creating `IgniteConfiguration` (configuration factory) to create an instance of an Ignite context. Ignite context will make sure that Ignite nodes exist in all involved job instances. Alternatively, a path to an XML configuration file can be passed to `IgniteContext` constructor which will be used to nodes being started.

Once `IgniteContext` is created, instances of IgniteRDD may be obtained using `fromCache()` methods. It is not required that requested cache exists in the Ignite cluster when RDD is created. If the cache with the given name does not exist, it will be created using provided configuration or template configuration.

Apache Ignite also provides three different deployments of Apache Ignite along with Apache Spark.

Deployment	Description
Shared deployment	Shared deployment implies that Apache Ignite nodes are running independently from Apache Spark applications and store state even after Apache Spark jobs die.
Standalone deployment	Ignite nodes should be deployed together with Spark Worker nodes in Standalone deployment mode. After you install Ignite on all worker nodes, start a node on each Spark worker with your config using <code>ignite.sh</code> script.

<sup>171</sup><https://github.com/apache/ignite/blob/master/modules/spark/src/main/scala/org/apache/ignite/spark/IgniteRDD.scala>

<sup>172</sup><https://github.com/apache/ignite/blob/master/modules/spark/src/main/scala/org/apache/ignite/spark/IgniteContext.scala>

Deployment	Description
Embedded deployment	Embedded deployment means that Apache Ignite nodes are started inside Apache Spark job processes and stopped when job dies. No need for additional deployment steps in this case. Apache Ignite code will be distributed to the worker machines using Apache Spark deployment mechanism, and nodes will be started on all workers as a part of IgniteContext initialization.



## Warning

Starting from 2.1 version embedded deployment mode is **deprecated** and will be eventually discontinued. Consider starting a separate Ignite cluster and using standalone mode to avoid data consistency and performance issues.

Next, we will install Spark and develop some application to see how we can use the Ignite RDD API to get benefits. We are going to do the following:

1. Install Spark in a standalone mode.
2. Run the `wordcount` example to verify the Spark installation.
3. Configure Apache Ignite to share RDD between Spark applications.
4. Run a Spark application through Spark Shell to use Ignite RDD.
5. Develop a Scala Spark application to put some Ignite RDD into the Ignite node and pull them from another Scala Spark application.

The full source code of the Spark section is available at [GitHub repository<sup>173</sup>](#).

### Preparing the sandbox

The easiest way to get started with Apache Ignite and Apache Spark is to use the standalone deployment. We need at least one Spark master node and a worker node running on a single host. I will have the following versions of Apache Ignite and Spark in my sandbox.

Name	Version
Apache Ignite	2.6.0
Apache Spark	2.3.1 with Hadoop 2.7
JDK	1.8.0
OS	Linux or MacOs

<sup>173</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-9/IgniteSparkApp>

**Step 1.** Download the Spark pre-build binary from this [link<sup>174</sup>](#) and extracts the archive in some location. Let's us call this directory `SPARK_HOME`. Add the `SPARK_HOME` directory to your PATH environmental variable as follows:

**Listing 9.1**

---

```
export SPARK_HOME=/Users/shamim/Development/bigdata/spark-2.3.1-bin-hadoop2.7
export IGNITE_HOME=/Users/shamim/Development/bigdata/ignite/2.6.0
export PATH=$IGNITE_HOME/bin:$SPARK_HOME/sbin:$PATH
```

---

**Step 2.** Start the Spark master node. Execute the following command into your preferred command line console.

**Listing 9.2**

---

```
start-master.sh
```

---

It should give a logging file info saying *starting org.apache.spark.deploy.master.Master, logging to /Users/shamim/Development/bigdata/spark-2.3.1-bin-hadoop2.7/logs/spark-shamim-org.apache.spark.deploy.master.Master-1-shamim.local.out*

Print the log file info with the Linux command `cat` as follows:

**Listing 9.3**

---

```
cat /Users/shamim/Development/bigdata/spark-2.3.1-bin-hadoop2.7/logs/spark-shamim-org.apache.spark.deploy.master.Master-1-shamim.local.out
```

---

The log file should show you the *master URL* and the *port* as shown in the next figure.

---

<sup>174</sup><http://spark.apache.org/downloads.html>

```

shamim:~ shamim$ cat /Users/shamim/Development/bigdata/spark-2.3.1-bin-hadoop2.7/logs/spark-shamim-org.apache.spark.de
ploy.master.Master-1-shamim.local.out
Spark Command: /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java -cp /Users/shamim/Development/
bigdata/spark-2.3.1-bin-hadoop2.7/conf:/Users/shamim/Development/bigdata/spark-2.3.1-bin-hadoop2.7/jars/* -Xmx1g org.
apache.spark.deploy.master.Master --host shamim.local --port 7077 --webui-port 8080
=====
2018-09-21 15:01:17 INFO Master:2611 - Started daemon with process name: 941@shamim.local
2018-09-21 15:01:17 INFO SignalUtils:54 - Registered signal handler for TERM
2018-09-21 15:01:17 INFO SignalUtils:54 - Registered signal handler for HUP
2018-09-21 15:01:17 INFO SignalUtils:54 - Registered signal handler for INT
2018-09-21 15:01:18 WARN NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using built-in
java classes where applicable
2018-09-21 15:01:18 INFO SecurityManager:54 - Changing view acls to: shamim
2018-09-21 15:01:18 INFO SecurityManager:54 - Changing modify acls to: shamim
2018-09-21 15:01:18 INFO SecurityManager:54 - Changing view acls groups to:
2018-09-21 15:01:18 INFO SecurityManager:54 - Changing modify acls groups to:
2018-09-21 15:01:18 INFO SecurityManager:54 - SecurityManager: authentication disabled; ui acls disabled; users with
view permissions: Set(shamim); groups with view permissions: Set(); users with modify permissions: Set(shamim); grou
ps with modify permissions: Set()
2018-09-21 15:01:18 INFO Utils:54 - Successfully started service 'sparkMaster' on port 7077.
2018-09-21 15:01:18 INFO Master:54 - Starting Spark master at spark://shamim.local:7077
2018-09-21 15:01:18 INFO Master:54 - Running Spark version 2.3.1
2018-09-21 15:01:18 INFO log:192 - Logging initialized @2112ms
2018-09-21 15:01:18 INFO Server:346 - jetty-9.3.z-SNAPSHOT
2018-09-21 15:01:18 INFO Server:414 - Started @2213ms
2018-09-21 15:01:18 INFO AbstractConnector:278 - Started ServerConnector@71cec39d[HTTP/1.1,[http/1.1]]{0.0.0.0:8080}
2018-09-21 15:01:18 INFO Utils:54 - Successfully started service 'MasterUI' on port 8080.
2018-09-21 15:01:18 INFO ContextHandler:781 - Started o.s.j.s.ServletContextHandler@4ce10f3f{/app,null,AVAILABLE,@Spa
rk}

```

**Figure 9.5**

**Step 3.** Start a Spark worker node. Run the following command to start a Spark worker node.

**Listing 9.4**


---

```
$SPARK_HOME/bin/spark-class org.apache.spark.deploy.worker.Worker spark://shamim.local:7077
```

---

Here, `$SPARK_HOME` is the home directory where the spark distribution is installed, and `spark://shamim.local:7077` is the hostname and port where the Spark master up and running.

This above command will start the Spark worker node and connect to the Spark master node as shown below.

```

shamim:~ shamim$ $SPARK_HOME/bin/spark-class org.apache.spark.deploy.worker.Worker spark://shamim.local:7077
2018-09-21 15:10:00 INFO Worker:2611 - Started daemon with process name: 1002@shamim.local
2018-09-21 15:10:00 INFO SignalUtils:54 - Registered signal handler for TERM
2018-09-21 15:10:00 INFO SignalUtils:54 - Registered signal handler for HUP
2018-09-21 15:10:00 INFO SignalUtils:54 - Registered signal handler for INT
2018-09-21 15:10:00 WARN NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using built-in java classes where applicable
2018-09-21 15:10:00 INFO SecurityManager:54 - Changing view acls to: shamim
2018-09-21 15:10:00 INFO SecurityManager:54 - Changing modify acls to: shamim
2018-09-21 15:10:00 INFO SecurityManager:54 - Changing view acls groups to:
2018-09-21 15:10:00 INFO SecurityManager:54 - Changing modify acls groups to:
2018-09-21 15:10:00 INFO SecurityManager:54 - SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(shamim); groups with view permissions: Set(); users with modify permissions: Set(shamim); groups with modify permissions: Set()
2018-09-21 15:10:01 INFO Utils:54 - Successfully started service 'sparkWorker' on port 60369.
2018-09-21 15:10:01 INFO Worker:54 - Starting Spark worker 192.168.1.35:60369 with 8 cores, 15.0 GB RAM
2018-09-21 15:10:01 INFO Worker:54 - Running Spark version 2.3.1
2018-09-21 15:10:01 INFO Worker:54 - Spark home: /Users/shamim/Development/bigdata/spark-2.3.1-bin-hadoop2.7
2018-09-21 15:10:01 INFO log:192 - Logging initialized @1444ms
2018-09-21 15:10:01 INFO Server:346 - jetty-9.3.z-SNAPSHOT
2018-09-21 15:10:01 INFO Server:414 - Started @1495ms
2018-09-21 15:10:01 INFO AbstractConnector:278 - Started ServerConnector@7379bec2[HTTP/1.1,[http/1.1]]{[0.0.0.0:8081]}
2018-09-21 15:10:01 INFO Utils:54 - Successfully started service 'WorkerUI' on port 8081.
2018-09-21 15:10:01 INFO ContextHandler:781 - Started o.s.j.s.ServletContextHandler@5f3980ac{/logPage,null,AVAILABLE,@Spark}

```

Figure 9.6

At this moment, we open the Spark Master web console (aka, WebUI or Spark UI) to monitor and inspect Spark job executions in a web browser. Open the following URL in your browser.

#### Listing 9.5

---

<http://localhost:8080>

---

You should see the following Spark master web page (WebUI) as follows:



The screenshot shows the Spark Master web interface. At the top, it displays the Spark logo and the text "Spark Master at spark://shamim.local:7077". Below this, there is a summary of cluster resources:

- URL: spark://shamim.local:7077
- REST URL: spark://shamim.local:6066 (cluster mode)
- Alive Workers: 1
- Cores in use: 8 Total, 0 Used
- Memory in use: 15.0 GB Total, 0.0 B Used
- Applications: 0 Running, 0 Completed
- Drivers: 0 Running, 0 Completed
- Status: ALIVE

Below this, there is a section titled "Workers (1)" with a table showing one worker entry:

Worker Id	Address	State	Cores	Memory
worker-20180921151001-192.168.1.35-60369	192.168.1.35:60369	ALIVE	8 (0 Used)	15.0 GB (0.0 B Used)

Figure 9.7

You can notice from the above web page that our Spark master and one worker is up and running. You can also start a few more worker on a single host, but we are going to use only one Spark worker.



## Warning

Spark master and the web interface are running on *different hostname and port*.

In my case, the Spark master node is up and running on host `shamim.local:7077`, and the web monitoring tool running on `localhost:8080`.

Let's do some sanity test once we have Spark cluster up and running. Let's run the `linecount` Spark application to count words from a text file.

**Step 4.** Switch to the code sample project `chapter-9` or download it from the [Github repository<sup>175</sup>](#). Build the project `linecount` with maven command as follows:

**Listing 9.6**

---

```
mvn clean install
```

---

It will create a snapshot library in the `target` folder of the project. Then, open a new command shell and submit the job into the Spark cluster through `spark-submit` command.

**Listing 9.7**

---

```
$YOUR_SPARK_HOME/bin/spark-submit --class com.blu.imdg.LineCount --master spark://shamim.local:7077 /Users/shamim/Development/workshop/github/the-apache-ignite-book/chapters/chapter-9/IgniteSparkApp/linecount/target/linecount-1.0-SNAPSHOT.jar ~/Downloads/README.md
```

---

This is the most common way to lunch spark application on the cluster. When using `spark-submit shell command`, the spark application need not be configured particularly for each cluster as the `spark-submit` shell script uses the cluster managers through a single interface. The `spark-submit` script has several *flags* that help to control the resources used by the Apache Spark application.



## Tip

You should adequately pass the spark master URL and the file for counting lines.

In my case, SPARK master URL is `spark://shamim.local:7077` and the file I passed for count lines are `README.md`.

After you hit the enter button, you should be followed by the following text in your command shell.

---

<sup>175</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-9/IgniteSparkApp/linecount>

```
2018-09-21 16:02:49 INFO TaskSetManager:54 - Starting task 0.0 in stage 3.0 (TID 3, 192.168.1.35, executor 0, partition 0, NODE_LOCAL, 7758 bytes)
2018-09-21 16:02:49 INFO BlockManagerInfo:54 - Added broadcast_4_piece0 in memory on 192.168.1.35:65254 (size: 3.8 KB, free: 366.2 MB)
2018-09-21 16:02:49 INFO MapOutputTrackerMasterEndpoint:54 - Asked to send map output locations for shuffle 1 to 192.168.1.35:65252
2018-09-21 16:02:49 INFO TaskSetManager:54 - Finished task 0.0 in stage 3.0 (TID 3) in 27 ms on 192.168.1.35 (executor 0) (1/1)
2018-09-21 16:02:49 INFO TaskSchedulerImpl:54 - Removed TaskSet 3.0, whose tasks have all completed, from pool
2018-09-21 16:02:49 INFO DAGScheduler:54 - ResultStage 3 (Count at LineCount.java:19) finished in 0.034 s
2018-09-21 16:02:49 INFO DAGScheduler:54 - Job 1 finished: count at LineCount.java:19, took 0.086797 s
Lines with a: 61, lines with b: 30
```

Figure 9.8

This Spark application counts the number of lines containing *a* and *b* in the *README.md* file. Note that you have to pass the file name with an *absolute path* into the Spark submit command line application. From the above screenshot, you should notice that lines with the letter *a* have been found 61 times, and lines with letter *b* has been spotted 30 times. Let's start an Ignite node and configure it for using Ignite RDD.

**Step 5.** To start an Ignite node with the default configuration, open a command shell and type the following (assume that, you have already configured the `IGNITE_HOME`):

**Listing 9.8**

---

```
ignite.sh
```

---

A new Ignite node with default configuration should be launched.



## Warning

If your network does not allow *multicast* traffic, you will need to change the default configuration file and configure TCP discovery.

**Step 6.** Now that we have our Spark cluster up and running and pass the sanity test. Let's execute the Spark interactive shell and run our first *Ignite RDD application*. Spark's shell provides a simple way to learn API as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus an excellent way to use existing Java libraries) or Python.

There are several ways to configure your Spark shell to integrate with Apache Ignite:

**Solution 1:** Start the Spark shell with the Maven coordinators to Ignite artifacts as follows:

**Listing 9.9**

```
$SPARK_HOME/bin/spark-shell --packages org.apache.ignite:ignite-spark:2.6.0 --master spark://shamim.local:7077 --repositories http://repo.maven.apache.org/maven2/org/apache/ignite/e
```

This is the best convenient solution if you have a internet connection from your Spark worker host machine. The above command will download a lot of artifacts from the Maven repository and starts the Spark shell.

		modules		artifacts	
conf	number	search dwnlded evicted		number dwnlded	
default	93	0   0   3		90   1	

Figure 9.9

In my case, it has been downloaded 1 artifact. However, you may encounter some problems whenever downloading artifacts from Maven repository as follows:

```
:::::::::::::::::::  
::          FAILED DOWNLOADS          ::  
:: ^ see resolution messages for details ^ ::  
:::::::::::::::::::  
:: com.jamesmurty.utils#java-xmlbuilder;0.4!java-xmlbuilder.jar  
:: org.xerial.snappy#snappy-java;1.0.4.1!snappy-java.jar(bundle)  
:::::::::::::::::::
```

Figure 9.10

First time Spark shell could not download two files from the Maven repository. In such cases, the easiest way to solve the problem is to download the artifacts *manually* and add them to the local Maven repository then rerun the Spark shell.

**Solution 2:** Start the Spark shell by providing *paths to Ignite libraries* as shown below:

**Listing 9.10**

---

```
./bin/spark-shell --jars path/to/ignite-core.jar,path/to/ignite-spark.jar,path/to/cache-a\\
pi.jar,path/to/ignite-log4j.jar,path/to/log4j.jar --master spark://master_host:master_por\\
t
```

---

**Solution 3:** Add Ignite libraries into the Spark classpath. Locate the `$SPARK_HOME/conf/spark-env.sh` file. If this file does not exist, create it from the template using `$SPARK_HOME/conf/spark-env.sh.template`. Include the following lines to the end of the `sparkenv.sh` file (uncomment the line setting `IGNITE_HOME` in case you do not have it globally set):

**Listing 9.11**

---

```
IGNITE_LIBS="${IGNITE_HOME}/libs/*"
for file in ${IGNITE_HOME}/libs/*
do
if [ -d ${file} ] && [ "${file}" != "${IGNITE_HOME}/libs/optional" ]; then
IGNITE_LIBS=${IGNITE_LIBS}:${file}/*
fi
done
export SPARK_CLASSPATH=$IGNITE_LIBS
```

---

This way, you alter the default Spark classpath for each launched application.

**Tip**

These above steps should be done on each machine of the Spark cluster, including a master, worker and driver nodes.

**Step 7.** We are ready to execute Scala commands after Spark shell is up and running. We are going to count the words in `README.md` file that locates in your Spark distribution and saves them into an Ignite cache through Ignite RDD. Next, we will run another Spark shell to read/filter the RDD states from the cache. Let's execute the following lines of codes into the Spark shell.

**Listing 9.12**

---

```
val lines = sc.textFile("PATH_TO_YOUR_SPARK_DISTR/README.md")  
  
val words = lines.flatMap(_.split("\\s+"))  
  
val wc = words.map(w => (w, 1)).reduceByKey(_ + _)
```

---

In the first line of codes, we read the file `README.md` from the Spark distribution directory. Next, we split each line into words and flatten the result into `words` variable. We Map each word into a pair and count them by word (key) in the last line of the code.

You can use the `count()` function to print the total amount of words found in the file. In my case, the result is 287.

```
scala> wc.count()  
res2: Long = 287
```

**Step 8.** Let's create an instance of Ignite context using default configuration:

**Listing 9.13**

---

```
import org.apache.ignite.spark._  
import org.apache.ignite.configuration._  
  
val ic = new IgniteContext(sc, () => new IgniteConfiguration())
```

---

First two lines of codes imports Ignite and Spark libraries. The last line of the code creates an Ignite context with default configuration. Note that there is an alternative way to create an instance of `IgniteContext` through a configuration file. If the path to the configuration file is specified in a relative form, then the `IGNITE_HOME` environment variable should be globally set in the system as the path is resolved relative to `IGNITE_HOME`.

**Listing 9.14**


---

```
import org.apache.ignite.spark._  
import org.apache.ignite.configuration._  
  
val ic = new IgniteContext(sc, "examples/config/spark/example-shared-rdd.xml")
```

---

**Step 9.** Let's create an Instance of the Ignite RDD with the cache name wordcount2 as follows:

**Listing 9.15**


---

```
val sharedRDD3 = ic.fromCache[String, Int]("wordcount2")
```

---

You should see an instance of the RDD created for wordcount2 cache:

`sharedRDD3: org.apache.ignite.spark.IgniteRDD[String,Int] = IgniteRDD[11] at RDD at Ignit\\eAbstractRDD.scala:32`

Note that, our cache key type is *String* and the value type is *Integer*.

**Step 10.** Let's save the result of the word count into the Ignite cache.

**Listing 9.16**


---

```
sharedRDD3.savePairs(wc)
```

---

After running the above command, a new cache wordcount2 will be created, and the cache size should be 287. Let's scan the cache through Ignite visor command. After running the visor cache -scan command, you should have the similar information into your console.

```
+---+-----+-----+-----+  
| 2 | wordcount2(@c2) | PARTITIONED | min: 287 (0 / 287) |  
|   |           |           | avg: 287.00 (0.00 / 287.00) |  
|   |           |           | max: 287 (0 / 287) |  
+---+-----+-----+  
  
Choose cache number ('c' to cancel) [c]: 2  
Entries in cache: wordcount2  
+-----+-----+-----+  
| Key Class | Key | Value Class | Value |  
+-----+-----+-----+  
| java.lang.String | <class> | java.lang.Integer | 47 |  
| java.lang.String | <class> | java.lang.Integer | 1 |  
| java.lang.String | your | java.lang.Integer | 1 |  
| java.lang.String | building | java.lang.Integer | 2 |  
| java.lang.String | sc.parallelize(1 | java.lang.Integer | 1 |  
| java.lang.String | Spark"](http://spark.apache.org/docs/latest/building-spark.html) | java.lang.Integer | 1 |  
| java.lang.String | you | java.lang.Integer | 4 |  
| java.lang.String | # | java.lang.Integer | 1 |
```

Figure 9.11

**Step 11.** Now, we can check how the state we created will survive Spark job restart. Shutdown the Spark shell by using the command `:q` and repeat **steps 8-9**. You should again have an instance of Ignite context and RDD for `wordcount2` cache again. Now, check how many lines contain word `Spark`. Run the following command:

**Listing 9.17**

---

```
sharedRDD3.filter(_.1 == "Spark").collect()
```

---

We filter out all the tuples where the first value (word) in the tuple is equal to word `Spark`. We should see `res14: Array[(String, Int)] = Array(Spark,16)` as a result. Similarly, we can check how many words are in our RDD which total value is greater than `10`.

**Listing 9.18**

---

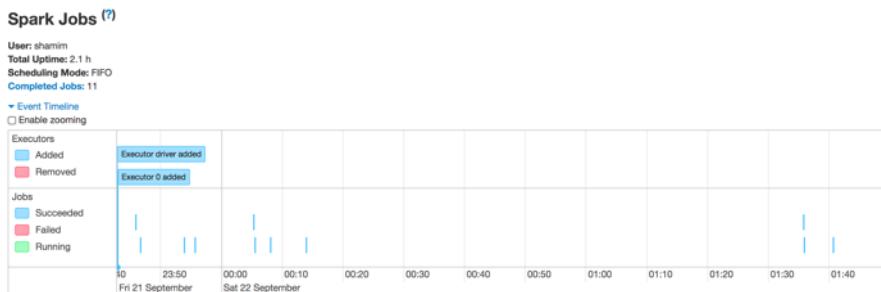
```
sharedRDD3.filter(_.2 > 10).count()
```

---

The above command should return the following result.

`res15: Long = 5`

If you switch back to the Spark jobs UI, you should perceive all of the stages executed by the spark shell. Note that Spark Job UI usually run on `localhost:4040`.



**Figure 9.12**

This way, you can share IgniteRDD between two different Spark shell. All the above scripts are available at the Github repository. Next, we are going to develop a Scala self-contained application to share RDD state between different spark applications. We will have two Scala applications, one of them will write tuples (key, value) of 1000 entries into an Ignite RDD and another application will perform some filtering and return a result for us.

The application will be very much similar to the example described in the Ignite documentation, but we develop it as a Scala application through Maven. In the future, you can use this Scala project for developing your own application with IgniteRDD. Note that, in this sample application, we only use *pure Scala* and *Maven jar plugin* to build the application.

**Step 12.** Download the project `ignite-spark-scala` from the [Github repository](#)<sup>176</sup>. Compile the project with the following command.

#### Listing 9.19

---

```
mvn clean install
```

---

It will take a few moments to build the application. During this time let's check the code snippet.

#### Listing 9.20

---

```
object RDDProducer extends App {  
    val conf = new SparkConf().setAppName("SparkIgniteProducer")  
    val sc = new SparkContext(conf)  
    val ic = new IgniteContext(sc, () => new IgniteConfiguration())  
    val sharedRDD: IgniteRDD[Int, Int] = ic.fromCache("IgniteRDD")  
    sharedRDD.savePairs(sc.parallelize(1 to 1000, 10).map(i => (i, i)))  
  
    ic.close(true)  
    sc.stop()  
}  
  
object RDDConsumer extends App {  
    val conf = new SparkConf().setAppName("SparkIgniteConsume")  
    val sc = new SparkContext(conf)  
    val ic = new IgniteContext(sc, () => new IgniteConfiguration())  
    val sharedRDD = ic.fromCache[Int, Int]("IgniteRDD")  
    val lessThanTwenty = sharedRDD.filter(_._2 < 20)  
  
    println("The count is::::::: "+lessThanTwenty.count())  
  
    ic.close(true)  
    sc.stop()  
}
```

---

<sup>176</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-9/IgniteSparkApp/ignite-spark-scala>

We have two Scala applications in the code above: *RDDProducer* and *RDDConsumer*. Let's go through the pseudo code line by line. In the *RDDProducer* application, we first create the Spark configuration with name *SparkConf* that includes the application name *SparkIgniteProducer*. Next, we create a Spark context based on the Spark configuration. Finally, we create an instance of the Ignite context by supplying the Spark context created earlier. Note that we are using Ignite default configuration for creating the Ignite context. After we successfully created the IgniteContext, a *sharedRDD* (*Ignite RDD*) is launched by invoking the method `fromCache("IgniterRDD")` with cache name IgniteRDD. Next, we store the integer values from *1 to 1000* into the Ignite RDD. The numbers are stored using ten parallel operations. Finally, we close the Ignite and Spark contexts.

Similarly, the *RDDConsumer* application retrieves the RDD's from the cache by cache name IgniteRDD, and applies the *transformation filter* for the pair having values less than *twenty* and prints it. Note that, in our Scala *RDDConsumer* application, the initialization and setup are identical to the Scala *RDDProducer*.

**Step 13.** After the successful compilation of the project, two Java libraries will be created with name `ignite-spark-scala-1.0-SNAPSHOT-RDDConsumer.jar` and `ignite-spark-scala-1.0-SNAPSHOT-RDDProducer.jar` into the project *target* directory. We can submit this application through Spark submit command. First, we will submit the *RDDProducer* as follows:

#### Listing 9.21

---

```
SPARK_HOME/bin/spark-submit --class com.blu.imdg.RDDProducer --master spark://HOST_NAME:PORT
ORT /PATH_TO_THE_PRODUCER_JAR_FILE/the-apache-ignite-book/chapters/chapter-9/IgniteSparkApp/ignite-spark-scala/target/ignite-spark-scala-1.0-SNAPSHOT-RDDProducer.jar
```

---

Note that, you have to modify the *IP address* and *port* number (*HOST\_NAME:PORT*) and the path (*/PATH\_TO\_THE\_PRODUCER\_JAR\_FILE*) for your environment. This command will run an Ignite client node for the Spark worker and generate a lot of logs into the console as shown below.

```
>>> +-----+
>>> Ignite ver. 2.6.0#20180710-sha1:b69feacc5d3a4e60efcd300dc8de99780f38eed stopped OK
>>> +-----+
>>> Grid uptime: 00:00:04.249

2018-09-22 12:39:16 INFO AbstractConnector:318 - Stopped Spark@38d5b107{HTTP/1.1}:[0.0.0.0:4040]
2018-09-22 12:39:16 INFO SparkUI:54 - Stopped Spark web UI at http://192.168.1.35:4040
2018-09-22 12:39:16 INFO StandaloneSchedulerBackend:54 - Shutting down all executors
2018-09-22 12:39:16 INFO CoarseGrainedSchedulerBackend$DriverEndpoint:54 - Asking each executor to shut down
2018-09-22 12:39:16 INFO MapOutputTrackerMasterEndpoint:54 - MapOutputTrackerMasterEndpoint stopped!
2018-09-22 12:39:17 INFO MemoryStore:54 - MemoryStore cleared
2018-09-22 12:39:17 INFO BlockManager:54 - BlockManager stopped
2018-09-22 12:39:17 INFO BlockManagerMaster:54 - BlockManagerMaster stopped
2018-09-22 12:39:17 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:54 - OutputCommitCoordinator stopped!
2018-09-22 12:39:17 INFO SparkContext:54 - Successfully stopped SparkContext
```

Figure 9.13

You can also start the command-line tool `ignitevisor`, and confirm that IgniteRDD cache had been created. There should be *1000* entries into the *IgniteRDD* cache.

```
visor> cache -scan
Time of the snapshot: 2018-09-22 11:28:03
+-----+
| # |     Name(@)      | Mode    |      Size (Heap / Off-heap)   |
+-----+
| 0 | IgniteRDD(@c3) | PARTITIONED | min: 1000 (0 / 1000)       |
|   |                   |           | avg: 1000.00 (0.00 / 1000.00) |
|   |                   |           | max: 1000 (0 / 1000)       |
+-----+
```

Figure 9.14

**Step 14.** Once the Spark *RDDProducer* job has been completed, we can run the Scala *RDDConsumer* application as follows:

#### Listing 9.22

---

```
SPARK_HOME/bin/spark-submit --class com.blu.imdg.RDDConsumer --master spark://HOST_NAME:PORT /PATH_TO_THE_PRODUCER_JAR_FILE/the-apache-ignite-book/chapters/chapter-9/IgniteSparkApp/ignite-spark-scala/target/ignite-spark-scala-1.0-SNAPSHOT-RDDConsumer.jar
```

---

The job submit process is the same as before, just that we have changed the class name and the Jar file name. Don't forget to modify the IP address and port number (*HOST\_NAME:PORT*) and the path (*/PATH\_TO\_THE\_PRODUCER\_JAR\_FILE*) for your environment. This produces the following output:

```
2018-09-22 11:51:43 INFO TaskSchedulerImpl:54 - Removed TaskSet 0.0, whose tasks have all completed, from pool
2018-09-22 11:51:43 INFO DAGScheduler:54 - ResultStage 0 (Count at SparkIgniteTest.scala:25) finished in 9.252 s
2018-09-22 11:51:43 INFO DAGScheduler:54 - Job 0 finished: count at SparkIgniteTest.scala:25, took 9.372597 s
The count is::::::::::: 19
2018-09-22 11:51:43 INFO IgniteContext:172 - Will stop Ignite nodes on 2 workers
2018-09-22 11:51:43 INFO SparkContext:54 - Starting job: foreachPartition at IgniteContext.scala:175
2018-09-22 11:51:43 INFO DAGScheduler:54 - Got job 1 (foreachPartition at IgniteContext.scala:175) with 2 output part
itions
```

Figure 9.15

*RDDConsumer* Spark application retrieves the RDD from the cache and filter for the pair having values less than twenty and prints it. At this moment, if we shut down our Spark cluster (worker and master). Our Ignite node will remain running, and the Ignite RDD is still available for use by other applications.

From here, you can develop your own Scala application to share RDD's between Spark jobs. In the next section of this chapter, we are going to develop another Scala application to demonstrate the capability of the Spark DataFrames API.

## Ignite DataFrame

Apache Ignite supports Spark *DataFrames* API to improve data access times starting from 2.4 version. As stated earlier, the Apache Spark DataFrame API introduced the notation of a schema to describe data. Spark manages the schema and organizes the data into a tabular format.

Apache Ignite expands Spark DataFrame API simplifying the development processes and improve the performance of the Spark SQL execution times. In addition, you can get the following benefits whenever running Ignite with Spark:

1. Boost DataFrame and Spark SQL performances. Ignite provides SQL with indexing so that Spark SQL can be accelerated over 100x.
2. Avoiding data movement from a data source to Spark workers and applications.
3. Fast loading of massive amounts of data into Ignite cluster from different sources.

In this section, we are going to develop two Java applications to see how we can use Ignite DataFrame from scratch. We are going to do the following:

1. Develop a Java application which will load some data from the CSV file and create a DataFrame that is stored in Ignite.
2. Develop another Java application that will read the DataFrame/DataSet from the Ignite caches and performs some SQL queries over the dataset.

Let's start with the DataFrame loader Java application.

**Step 1.** Download the *load-data-module* project from the [Github repository<sup>177</sup>](#). The project uses two Maven dependencies as shown below:

Listing 9.23

---

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spark</artifactId>
    <version>2.6.0</version>
</dependency>
```

---

Note that we added these above two dependencies in the parent POM.xml instead of the project POM.xml file because our second Java application will also use these two libraries

The entire project contains only one Java class. Let's take a closer look at the Java class named com.blu.imdg.dataframe.LoadingData.java.

Listing 9.24

---

```
public class LoadingData {

    private static final String IGNITE_CONFIG_XML = "ignite.config.xml";

    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: ignite.spark.load.module-1.0.jar PATH_TO/IgniteSparkApp");
            System.exit(1);
        }

        String config_path = args[0];

        IgniteSparkSession igniteSession = IgniteSparkSession.builder()
            .appName("Spark Ignite data loading example")
            .master("local")
```

---

<sup>177</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-9/IgniteSparkApp/load-data-module>

```
.config("spark.executor.instances", "2")
//Only additional option to refer to Ignite cluster.
.igniteConfig(config_path+"/"+IGNITE_CONFIG_XML)
.getOrCreate();

igniteSession.read()
.format("com.databricks.spark.csv")
.option("header", "true")
.option("mode", "DROPMALFORMED")
.csv("file://"+config_path+"/scripts"+"/664600583_T_ONTIME_sample2.csv")
.write()
.format(igniteDataFrameSettings.FORMAT_IGNITE())
.option(igniteDataFrameSettings.OPTION_CONFIG_FILE(), config_path+"/"+IGNITE_CONFIG_XML)
.option(igniteDataFrameSettings.OPTION_CREATE_TABLE_PRIMARY_KEY_FIELDS(),"id")
.option(igniteDataFrameSettings.OPTION_CREATE_TABLE_PARAMETERS(), "template=replicated")
.saveAsTable("FLIGHTS");

igniteSession.read()
.format("com.databricks.spark.csv")
.option("header", "true")
.option("mode", "DROPMALFORMED")
.csv("file://"+config_path+"/scripts"+"/L_AIRPORT_ID2.csv")
.write()
.format(igniteDataFrameSettings.FORMAT_IGNITE())
.option(igniteDataFrameSettings.OPTION_CONFIG_FILE(), config_path+"/"+IGNITE_CONFIG_XML)
.option(igniteDataFrameSettings.OPTION_CREATE_TABLE_PRIMARY_KEY_FIELDS(),"Code")
.option(igniteDataFrameSettings.OPTION_CREATE_TABLE_PARAMETERS(), "template=replicated")
.saveAsTable("AIRPORTS");

igniteSession.stop();
}
```

---

Let's analyze the above code line by line again. First, we create an `IgniteSparkSession` instance from the Ignite Spark session builder. That is the entry point for programming with Spark. The `IgniteSparkSession` includes the application name, Spark and Ignite node configurations.

Note that, in this example, we are executing the Spark master node as a local execution unit and initiate two Spark executor instances. If you want to use your separate Spark cluster to implement the data loading application, you can customize the configuration as follows:

**Listing 9.25**

---

```
IgniteSparkSession igniteSession = IgniteSparkSession.builder()
    .appName("Spark Ignite data loading example")
    .config("spark.master", "spark://HOST:IP")
    .igniteConfig(config_path+"/"+IGNITE_CONFIG_XML)
    .getOrCreate();
```

---

Alternatively, you may not specify the Spark configuration. In this case, you have to submit the Java application through Spark submit shell command and specify the master explicitly as shown below.

**Listing 9.26**

---

```
$SPARK_HOME/bin/spark-submit --class com.blu.imdg.dataframe.LoadingData --master spark://\nHOST:PORT
```

---

Next, we read a CSV file located in the `scripts` folder using `IgniteSparkSession.read()` method and write it to Ignite storage as a table `FLIGHTS`. We use the `com.databricks.spark.csv` format to parse the CSV file. Also, set the column `ID` as a primary key for the table `FLIGHTS`.



## Info

Under the cover, Ignite uses `igniteDataStreamer` to insert data into Ignite tables/caches.

In the subsequent line, we repeat the same process. Read the file `L_AIRPORT_ID2.csv` from the folder `scripts` and create DataFrames to store into the Ignite cache as a table `AIRPORTS`. In the `AIRPORTS` table column `CODE` uses as the primary key for the table. At the end of the Java application, we close the `IgniteSparkSession` to release the resources.

**Step 2.** Let's build the project with Maven.

**Listing 9.27**

---

```
mvn clean install
```

---

The above command will build the project and create a Java Jar file with name `ignite.spark.load.module-1.0.jar` into the project `target` directory.

**Step 3.** Start an Ignite node with the default configuration.

**Listing 9.29**

---

```
ignite.sh
```

---

**Step 4.** Now, we are ready to run our application to load some data into the Ignite cache. We will use the `spark-submit` shell command to execute the Java application. Run the following command as shown below:

**Listing 9.28**

---

```
$SPARK_HOME/bin/spark-submit PATH_TO_THE_LOADER_JAR/target/ignite.spark.load.module-1.0.jar  
ar PATH_TO_THE_PROJECT_HOME/IgniteSparkApp
```

---

The above command parameters are slightly different from the previous section. Here we omitted the `-class` and the `--master` parameter because we are using local Spark to execute the application. This simplifies the application developing process because you do not need to run Spark cluster with workers node explicitly. This above command will generate a lot of logs into the console and create two tables into the Ignite node.

```
2018-09-22 18:03:23 INFO  SparkUI:54 - Stopped Spark web UI at http://192.168.1.35:4040
2018-09-22 18:03:23 INFO  BlockManagerInfo:54 - Removed broadcast_5_piece0 on 192.168.1.35:57333 in memory
B, free: 366.2 MB)
2018-09-22 18:03:23 INFO  GridCacheProcessor:566 - Stopped cache [cacheName=ignite-sys-cache]
2018-09-22 18:03:23 INFO  GridCacheProcessor:566 - Stopped cache [cacheName=SQL_PUBLIC_FLIGHTS]
2018-09-22 18:03:23 INFO  GridCacheProcessor:566 - Stopped cache [cacheName=SQL_PUBLIC_AIRPORTS]
2018-09-22 18:03:23 INFO  IgniteKernal:566 -
>>> +-----+
>>> Ignite ver. 2.6.0#20180710-sha1:669feacc5d3a4e60efcdd300dc8de99780f38eed stopped OK
>>> +-----+
>>> Grid uptime: 00:00:08.050
```

**Figure 9.16**

**Step 5.** Let's use the Ignite SQLLINE command tools to explore the two tables. Use the `!tables` command to print out all tables in the database.

0: jdbc:ignite:thin://127.0.0.1/> !tables			
TABLE_CAT	TABLE_SCHEM	TABLE_NAME	TABLE_T
	PUBLIC	FLIGHTS	TABLE
	PUBLIC	AIRPORTS	TABLE

Figure 9.17

Please, refer to the *chapter 2* if you need any help to start the Ignite *SQLLINE tool* to connect to the Ignite database. Alternatively, you can also use the command `!columns TABLE_NAME` to prints all the columns for the specified table.

0: jdbc:ignite:thin://127.0.0.1/> !columns FLIGHTS			
TABLE_CAT	TABLE_SCHEM	TABLE_NAME	COLUMN_N
	PUBLIC	FLIGHTS	ID
	PUBLIC	FLIGHTS	YEAR
	PUBLIC	FLIGHTS	QUARTER
	PUBLIC	FLIGHTS	MONTH
	PUBLIC	FLIGHTS	DAY_OF_MONTH
	PUBLIC	FLIGHTS	DAY_OF_WEEK
	PUBLIC	FLIGHTS	FL_DATE
	PUBLIC	FLIGHTS	UNIQUE_CARRIER
	PUBLIC	FLIGHTS	AIRLINE_ID
	PUBLIC	FLIGHTS	CARRIER
	PUBLIC	FLIGHTS	TAIL_NUM
	PUBLIC	FLIGHTS	FL_NUM
	PUBLIC	FLIGHTS	ORIGIN_AIRPORT_ID
	PUBLIC	FLIGHTS	ORIGIN_AIRPORT_SE
	PUBLIC	FLIGHTS	ORIGIN_CITY_MARK
	PUBLIC	FLIGHTS	DEST_AIRPORT_ID
	PUBLIC	FLIGHTS	WHEELS_ON
	PUBLIC	FLIGHTS	ARR_TIME
	PUBLIC	FLIGHTS	ARR_DELAY
	PUBLIC	FLIGHTS	ARR_DELAY_NEW

Figure 9.18

**Step 6.** Now that we have some data in our Ignite database, we can use Spark to perform some SQL queries. Download the project *query-module* from the [Github repository](#)<sup>178</sup>. This Maven project also contains only one Java class to perform SQL queries. Here is the code for the `com.blu.imdg.dataframe.IgniteDataFrame.java` class:

Listing 9.29

---

```
public class IgniteDataFrame {

    private static final String IGNITE_CONFIG_XML = "ignite.config.xml";

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.err.println("Usage: ignite.spark.query-1.0.jar PATH_TO/IgniteSparkApp"
        );
        System.exit(1);
    }
}
```

<sup>178</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-9/IgniteSparkApp/query-module>

```
}

String config_path = args[0];

IgniteSparkSession igniteSession = IgniteSparkSession.builder()
    .appName("Spark Ignite dataframe example")
    .master("local")
    .config("spark.executor.instances", "2")
    //config("spark.master","spark://shamim.local:7077")
    //Only additional option to refer to Ignite cluster.
    .igniteConfig(config_path+"/"+IGNITE_CONFIG_XML)
    .getOrCreate();

igniteSession.catalog().listTables().show();

Dataset<Row> res = igniteSession.sql(""+ +
    "select f.ORIGIN_AIRPORT_ID, a.DESCRIPTION, count(f.ID) from FLIGHTS f " +
    "join AIRPORTS a on a.Code=f.ORIGIN_AIRPORT_ID" +
    " group by f.ORIGIN_AIRPORT_ID, a.DESCRIPTION order by count(f.ID)"
    );
res.foreach(v -> {
    System.out.println("Row="+v);
});
igniteSession.stop();
}
```

---

In the *IgniteDataFrame* application, the initialization and setup are identical to the *LoadingData* Java class. We use the `igniteSession.SQL()` function to perform an *SQL* query which should return a `DataSet` of *Airport ID*, *description* and *the total flights* of each airport. The result is then printed out.



## Info

Internally all Spark SQL query converts to *Ignite SQL query* if query consists of only Ignite tables.

**Step 7.** Finally, we are ready to test our application. Launch the `IgniteDataFrame` application through the `spark-submit` shell command.

**Listing 9.30**

---

```
$SPARK_HOME/bin/spark-submit PATH_TO_THE_LOADER_JAR/target/ignite.spark.load.module-1.0.jar PATH_TO_THE_PROJECT_HOME/IgniteSparkApp
```

---

The above command should produce the following output.

```
2018-09-22 18:36:55 INFO CodeGenerator:54 - Code generated in 55.25982 ms
Row=[15497,St. Augustine, FL: Northeast Florida Regional,1]
Row=[14222,Pago Pago, TT: Pago Pago International,1]
Row=[13964,North Bend/Coos Bay, OR: Southwest Oregon Regional,1]
Row=[13459,Marquette, MI: Sawyer International,2]
Row=[10165,Adak Island, AK: Adak,2]
Row=[11252,Daytona Beach, FL: Daytona Beach International,3]
Row=[14025,Plattsburgh, NY: Plattsburgh International,3]
Row=[12265,Niagara Falls, NY: Niagara Falls International,3]
Row=[13388,Mammoth Lakes, CA: Mammoth Lakes Airport,4]
Row=[10170,Kodiak, AK: Kodiak Airport,4]
Row=[10926,Cordova, AK: Merle K Mudhole Smith,4]
Row=[12177,Hobbs, NM: Lea County Regional,4]
Row=[14150,Pellston, MI: Pellston Regional Airport of Emmet County,4]
Row=[13873,Nome, AK: Nome Airport,5]
Row=[15008,St. Cloud, MN: St. Cloud Regional,5]
Row=[12016,Guam, TT: Guam International,5]
Row=[11898,Grand Forks, ND: Grand Forks International,5]
Row=[14905,Santa Maria, CA: Santa Maria Public/Capt. G. Allan Hancock Field,5]
Row=[15582,Vernal, UT: Vernal Regional,5]
Row=[14109,Hattiesburg/Laurel, MS: Hattiesburg-Laurel Regional,5]
Row=[13127,Lewiston, ID: Lewiston Nez Perce County,5]
```

Figure 9.19

This is what we expected. The `IgniteDataFrame` application execute the SQL query and print out the total flights for each airport. For self-practice, you can run a query such as *select last 5 Departing Flights* or *count all canceled flights group by airports*.

And with this, we come to an end of this chapter. That's not all, as there is a lot to learn when working with Spark DataFrames or DataSets. For instance, you can join Ignite DataFrame with Spark native DataFrames. In the subsequent releases, Ignite will add more advanced features such as `IgniteRelationProvider` or `IgniteSQLRelation` which can provide partitioning information to Spark.

## Summary

In this chapter, we have covered Ignite integration with Apache Spark. We described how Apache Ignite RDD and Spark new Dataframe API could be shared application states between Spark applications. We installed the Spark standalone cluster and configured Ignite to run some Scala applications which use Ignite DataFrame API.

## What's next?

The next chapter will deal with the various ways to monitor, recognize problems and troubleshoot the Ignite cluster in detail.

# **Chapter 10. Management and monitoring**

As a system or cluster grows, you may start getting hardware or virtual machine failures in your cloud/dedicated infrastructure. In such cases, you may need to do one of these things: add/remove nodes from the cluster or backup/repair nodes. These tasks come as an integral part of a system administrator daily work. Luckily all these tasks are relatively straightforward in Apache Ignite and partially documented in Ignite documentation.

In the last chapter of this book, we will go through Ignite's built-in and 3<sup>rd</sup> party tools to manage and monitor the Ignite cluster in the production environment. We divided the entire chapter into two parts: management and monitoring. In the management part, we will discuss different tools and technics to configure and manage the Ignite cluster. And we will cover the basic of monitoring Apache Ignite includes logging, inspection of JVM, etc. in the monitoring part.

## Managing Ignite cluster

Out-of-the-box Apache Ignite provides several tools for managing cluster. These tools include *web interface* or *command line interface* that allows you to perform various task such as start/stop/restart remote nodes or control cluster states (baseline topology). A table below shows all the built-in tools of the Apache Ignite to configure and managing Ignite cluster.

Name	Description
Ignite Web console	Allows configuring all the cluster properties, and managing the Ignite cluster through a web interface.
Control script	A command line script that allows to monitor and control cluster states includes Ignite baseline topology.



### Info

Please note that *Ignite Web console* also uses for monitoring cluster functionality like cache metrics as well as CPU and Memory heap usages.

However, I would like to discuss the Ignite cluster discovery through ZooKeeper before diving into the management tools. As we stated earlier in chapter 4 that Ignite offers Zookeeper discovery that allows scaling Ignite cluster to 100s and 1000s nodes. From my personal experience with Ignite cluster, I highly recommend you to use Zookeeper as a discovery mechanism for the Ignite cluster in the production environment. Because, firstly, Zookeeper is a proven technology for discovering and managing massive clusters like Hadoop or Spark and secondly, you decrease the transmission time of the messaging between Ignite nodes with Zookeeper discovery. These give you a highly reliable Ignite cluster and make you sleep at night knowing things are functioning the way you want them to. Let's start from the Zookeeper discovery for managing Ignite cluster and goes through the remaining tools.

## Configuring Zookeeper discovery

As we have mentioned before in chapter 4, Zookeeper discovery mechanism solves the Apache Ignite TCP/IP discovery problems. It's recommended to move from the ring topology to the star type topology, in the center of which the Zookeeper service is used. All the messages about cluster topology exchanges through Zookeeper. In this section, we are going

to install a standalone Zookeeper server and configure Ignite discovery service through this server. Also, we will discuss some drawbacks of using ZooKeeper as a node discovery.

**Step 1.** Download and install the Apache [Zookeeper](#)<sup>179</sup>. You need a Zookeeper server up and running before configuring Ignite node to use Zookeeper as a node discovery mechanism. Please skip these three steps if you already have a Zookeeper server installed. Download the Zookeeper latest version from this [link](#)<sup>180</sup>, and unarchive it somewhere in your workstation. This directory will be the *Zookeeper home* directory.

---

**Listing 10.1**

---

```
tar -xzf zookeeper-3.4.12.tar.gz
```

---

**Step 2.** Create a Zookeeper configuration file; create a file with name `$ZOOKEEPER_HOME/conf/zoo.cfg`, and copy the following content into it.

---

**Listing 10.2**

---

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# for example, /tmp directory
dataDir=ABSOLUTE_PATH_TO_DATA_DIRECTORY/data
# the port at which the clients will connect
clientPort=2181
```

---

Set the `dataDir` properties to any physical directory located in your operating system. This directory will be used to store a snapshot. Also, note that we are using port 2181 as a client port to connected to the Zookeeper server.

**Step 3.** Start the Zookeeper server in a standalone mode and execute the following command into the terminal.

---

<sup>179</sup><https://zookeeper.apache.org/>

<sup>180</sup><https://www.apache.org/dyn/closer.cgi/zookeeper/>

**Listing 10.3**

---

```
$ZOOKEEPER_HOME/bin/zkServer.sh start
```

---

This command will start a Zookeeper server in a *background* mode. You can monitor the status of the Zookeeper server by the command below.

```
$ZOOKEEPER_HOME/bin/zkServer.sh status
```

**Step 4.** Copy the folder `$IGNITE_HOME/libs/optional/ignite-zookeeper` to the `$IGNITE_HOME/libs` directory. This will enable the Zookeeper related libraries into the Ignite classpath.



## Warning

You should do this step for each Ignite node manually. Also, you can use Ignite visor deploy command to copies files or folders to a remote node.

**Step 5.** Create a new Ignite configuration file named `ignite-zoo-config.xml`, and copy the following text into it.

**Listing 10.4**

---

```
<?xml version="1.0" encoding="UTF-8"?>

<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="cacheConfiguration">
            <list>
                <bean class="org.apache.ignite.configuration.CacheConfiguration">
                    <property name="name" value="myCache"/>
                    <property name="atomicityMode" value="ATOMIC"/>
                    <property name="backups" value="1"/>
                </bean>
            </list>
        </property>
    </bean>

```

---

---

```

<property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.zk.ZookeeperDiscoverySpi">
        <property name="zkConnectionString" value="127.0.0.1:2181"/>
        <property name="sessionTimeout" value="30000"/>
        <property name="zkRootPath" value="/zkPath"/>
        <property name="joinTimeout" value="10000"/>
    </bean>
</property>
</bean>
</beans>

```

---

Alternatively, you can copy the `discoverySpi` property into your existing Ignite configuration. Let's have a detailed look at the above configurations:

Name	Description	Optional/Mandatory
<code>zkConnectionString</code>	ZooKeeper connection string. It keeps the list of addresses of ZooKeeper servers.	Mandatory
<code>sessionTimeout</code>	Implies the time after which an Ignite node will be considered disconnected if it doesn't react to events exchanged via Discovery SPI.	Mandatory
<code>zkRootPath</code>	Base path in Zookeeper for znodes created by SPI.	Optional
<code>joinTimeout</code>	A timeout for joining to the cluster topology (0 means wait forever).	Optional



## Tip

You can also use the property `isClientMode` value `true` if a node is in client mode.

**Step 6.** Start the Ignite node with the following command.

**Listing 10.5**

---

```
$IGNITE_HOME/bin/ignite.sh PATH_TO_THE/ignite-zoo-config.xml
```

---

If you open and search the Ignite server log with word `zookeeper`, you should find log messages very similar as shown below. Usually, Ignite server logs stores into the `$IGNITE_HOME/work/log` directory.

```
[14:13:50,048][INFO][main][ZookeeperDiscoverySpi] Start Zookeeper discovery [zkConnection]\nString=127.0.0.1:2181, sessionTimeout=30000, zkRootPath=/zkPath]\n[14:13:50,178][INFO][zk-null-EventThread][ZookeeperClient] ZooKeeper client state changed\\n[prevState=Disconnected, newState=Connected]\n[14:13:50,217][INFO][main][ZookeeperDiscoveryImpl] Node started join [nodeId=c0f43987-31e\\n1-49b4-ad4d-3e86c47de4ca, instanceName=null, zkSessionId=0x1000104ce5b0000, joinDataSize=\\n7238, consistentId=0:0:0:0:0:1,127.0.0.1,192.168.1.35:47100, initTime=101, nodePath=\\nzkPath/n/65e6b8bb-bbb2-4dc8-aa5d-a8c99f6a98b1:c0f43987-31e1-49b4-ad4d-3e86c47de4ca:80|000\\n0000003]\n[14:13:50,237][INFO][zk-null-EventThread][ZookeeperDiscoveryImpl] Previous cluster data c\\nleanUp time: 5\n[14:13:50,237][INFO][zk-null-EventThread][ZookeeperDiscoveryImpl] New cluster started [lo\\ncId=c0f43987-31e1-49b4-ad4d-3e86c47de4ca, clusterId=4a0adcc4-2d8a-4d58-ace5-4207f7c338bd,\\nstartTime=1538392430237]\n[14:13:50,277][INFO][zk-null-EventThread][ZookeeperDiscoveryImpl] Process alive nodes cha\\nge [alive=1]
```

The above log messages specified that an Ignite server node has been started and joined to Zookeeper server. Let's start an another Ignite server and check the cluster topology through Ignite visor.



## Warning

This time in Ignite visor, you have to specify the proper configuration file. In my case, the configuration file is in number **16** as shown in the figure below.

```
visor> open
Local configuration files:
+-----+
| # | Configuration File |
+-----+
| 0 | config/default-config.xml |
| 1 | benchmarks/config/ignite-base-config.xml |
| 2 | benchmarks/config/ignite-localhost-config.xml |
| 3 | benchmarks/config/ignite-multicast-config.xml |
| 4 | benchmarks/config/ignite-remote-config.xml |
| 5 | benchmarks/sources/config/ignite-base-config.xml |
| 6 | benchmarks/sources/config/ignite-localhost-config.xml |
| 7 | benchmarks/sources/config/ignite-multicast-config.xml |
| 8 | benchmarks/sources/config/ignite-remote-config.xml |
| 9 | (?) config/router/default-router.xml |
| 10 | examples/config/example-cache.xml |
| 11 | examples/config/example-data-regions.xml |
| 12 | examples/config/example-default.xml |
| 13 | (?) examples/config/example-ignite.xml |
| 14 | examples/config/filesystem/example-igfs.xml |
| 15 | examples/config/ignite-book-persistence.xml |
| 16 | examples/config/ignite-zoo-config.xml |
| 17 | examples/config/persistentstore/example-persistent-store.xml |
```

Figure 10.1

**Step 7.** Use the `top` command in the Ignite visor to print the current topology state. The command should print very similar information shows in the figure 10.2.

```
visor> top
Hosts: 1
+-----+
| Int./Ext. IPs | Node ID8(@) | Node Type | OS | CPUs | MACs | CPU Load |
+-----+
| 192.168.1.35 | 1: C0F43987(@n0) | Server | Mac OS X x86_64 10.11.6 | 8 | 3A:71:58:55:4F:AB | 0.15 % |
| 0:0:0:0:0:0:1 | 2: BA16911C(@n1) | Server | | | 80:E6:50:07:70:B6 |
| 127.0.0.1 | | | | | | |
+-----+
Summary:
+-----+
| Active | true |
| Total hosts | 1 |
| Total nodes | 2 |
| Total CPUs | 8 |
| Avg. CPU load | 0.15 % |
| Avg. free heap | 91.00 % |
| Avg. Up time | 00:10:35 |
| Snapshot time | 2018-10-01 14:24:24 |
+-----+
```

Figure 10.2

That's all, Ignite nodes are connected and communicating through Zookeeper server. You can also use Zookeeper *four later commands* to get details of the server and connected clients. However, it's not enough for production use. In the production environment, you have to deploy Zookeeper cluster with multiple servers and configure *leader election* for reliability. With leader election, if one of the Zookeeper server nodes goes down for some reason, another server rises to do its work.

## Managing Baseline topology

We have covered a lot about Ignite Baseline topology and details the concept in previous chapters. Ignite Baseline topology comes into action when Ignite native persistence is enabled. When the Ignite cluster is started for the first time with the persistence option, the cluster considered as inactive disallowing any CRUD operations. To perform any operations on the cluster, you have to activate it first. Apache Ignite provides a command line (CLI) tool that allows to monitor and manage a cluster Baseline topology. In this section, we will review several common scenarios of Baseline topology administration with this tool when Ignite persistence is used.

The `./control.sh` command line script can be found under `/bin` folder of an Apache Ignite distribution directory. The primary goal of this script (tool) is to *activate/deactivate*, and management of a set of nodes that represent the baseline topology. However, this tool is a multi-purpose tool and can be actively used for monitoring the cache states or detecting any transaction locks that could occur in the entire cluster. We summarize all the capabilities of

the *control.sh* tool in the table below.

Feature	Description
Baseline topology management	Activate/deactivate and management of a set of nodes that represent the baseline topology.
Cache state monitoring	A set of commands for monitoring caches, such as display list of caches with affinity parameters or all atomic sequences.
Contention detection in transactions	A command for monitoring multiple transactions that are contending in their attempt a lock for the same key.
Consistency checks	A set of commands that allow verifying internal data consistency invariants: verifications of partition checksums, SQL indexes consistency validation

**Preparing the sandbox.** As stated earlier, the script that runs the tool is located in the `{ignite-home}/bin` folder and called `control.sh`. There are versions of the script for Unix (`control.sh`) and Windows (`control.bat`). I will use the following configurations for demonstration purpose:

Name	Description
OS	MacOS, you can use Windows or Linux operating system by your choice.
Ignite version	2.6.0 or above.
Number of Ignite nodes	3 nodes in a single host.
JVM	1.8
TCP discovery	Multicast

**Step 1.** We are going to run three Ignite nodes on persistence mode in a single host. Ignite creates a *WORK* directory under the `IGNITE_HOME` folder for storing WAL archives and log files by defaults. Download the Ignite distribution and unarchive it in 3 different directories on your operating system, for instance `/usr/ignite/2.6.0-s1`, `/usr/ignite/2.6.0-s2`, `/usr/ignite/2.6.0-s3`. You should have a similar folder hierarchy as shown in figure 10.3 below.

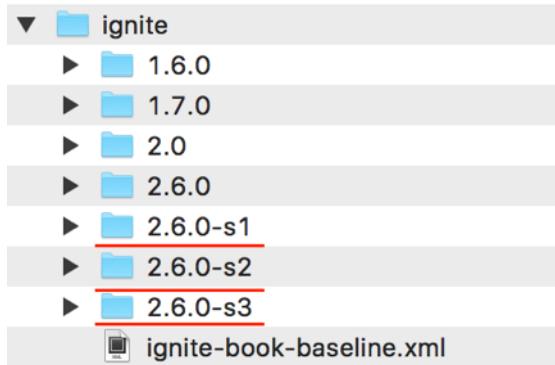


Figure 10.3

Note that it is the simplest way to run a few nodes with persistence enable in a single host without any extra configuration. However, you can configure Ignite in such a way that allows you to run a few Ignite nodes with different WAL archive folders.

**Step 2.** We use the Ignite data storage configuration through Spring to enable the persistence store. Create an XML file with name `ignite-book-baseline.xml`, and input the following content.

Listing 10.5

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
        <property name="cacheConfiguration">
            <list>
                <bean class="org.apache.ignite.configuration.CacheConfiguration">
                    <property name="name" value="TestCache"/>
                    <property name="atomicityMode" value="ATOMIC"/>
                    <property name="backups" value="1"/>
                </bean>
            </list>
        </property>
        <!-- Enabling Apache Ignite Persistent Store. -->
        <property name="dataStorageConfiguration">
            <bean class="org.apache.ignite.configuration.DataStorageConfiguration">
                <property name="defaultDataRegionConfiguration">
                    <bean class="org.apache.ignite.configuration.DataRegionConfiguration">
                        <property name="persistenceEnabled" value="true"/>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>

```

```
<property name="metricsEnabled" value="true"/>
</bean>
</property>
</bean>
</property>

<property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
        <property name="ipFinder">

            <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
                <property name="addresses">
                    <list>
                        <value>127.0.0.1:47500..47509</value>
                    </list>
                </property>
            </bean>
        </property>
    </bean>
</property>
</bean>
</beans>
```

---

Save the file somewhere in your filesystem.

**Step 3.** We will be starting each Ignite server node separately, starting with our first Ignite node. Open a terminal and change the `IGNITE_HOME` directory to the folder where you unarchive the Ignite distribution for the Ignite *node 1*.

**Listing 10.6**

---

```
export IGNITE_HOME=PATH_TO_THE_IGNITE_NODE_ONE/ignite/2.6.0-s1
```

---

Now, start the first Ignite node with the following command:

**Listing 10.7**

---

```
ignite.sh /PATH_TO_THE_SPRING_CONFIG_FILE/ignite/ignite-book-baseline.xml
```

---

Your output on the console should be similar to this:

```
[14:44:55] _____  
[14:44:55] / _/ __| | / / _\ _/ __/  
[14:44:55] _//(77 // / / / /  
[14:44:55] /__\_\_/_|/_\_\_/_/_/_/  
[14:44:55]  
[14:44:55] ver. 2.6.0#20180710-sha1:669feacc  
[14:44:55] 2018 Copyright(C) Apache Software Foundation  
[14:44:55]  
[14:44:55] Ignite documentation: http://ignite.apache.org  
[14:44:55]  
[14:44:55] Quiet mode.  
[14:44:55] ^-- Logging to file '/usr/ignite/2.6.0-s1/work/log/ignite-f0ef6ecc.0.log'  
...  
[14:44:57] Topology snapshot [ver=1, servers=1, clients=0, CPUs=8, offheap=3.2GB, heap=1.\  
0GB]  
[14:44:57] ^-- Node [id=F0EF6ECC-D692-4862-9414-709039FE00CD, clusterState=INACTIVE]  
[14:44:57] Data Regions Configured:  
[14:44:57] ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=true]
```

Inspect the logs displays on the console, log messages confirms that our first Ignite server is up and running and the persistence mode is enabled. Then, do the same thing again for the second Ignite node.

```
export IGNITE_HOME=PATH_TO_THE_IGNITE_NODE_ONE/ignite/2.6.0-s2  
ignite.sh /PATH_TO_THE_SPRING_CONFIG_FILE/ignite/ignite-book-baseline.xml
```

At this moment, you can see that the 2<sup>nd</sup> Ignite node started on persistence mode and joined to the cluster. You should see very similar messages in the terminal as shown below.

```
[16:13:35] >>> Ignite cluster is not active (limited functionality available). Use contro\  
l.(sh|bat) script or IgniteCluster interface to activate.  
[16:13:35] Topology snapshot [ver=2, servers=2, clients=0, CPUs=8, offheap=6.4GB, heap=2.\  
0GB]  
[16:13:35] ^-- Node [id=6DB02F31-115C-41E4-BECC-FDB6980F8143, clusterState=INACTIVE]  
[16:13:35] Data Regions Configured:  
[16:13:35] ^-- default [initSize=256.0 MiB, maxSize=3.2 GiB, persistenceEnabled=true]
```

Ignite also warned that the cluster is not *activated* yet and you have to activate the cluster by using *control.sh* script. Let's activate the cluster and creates a few tables for storing data.

**Step 4.** Let's consider specific features of the *control.sh* tool before we activate the cluster. The *control.sh* script currently supports the following commands:

Command	Description
-activate	This command switches the cluster into an active state. In this case, if there is no baseline topology exists in the cluster, a new baseline will be created during activation of the cluster. The new baseline topology will include all of the connected nodes in the cluster topology.
-deactivate	Deactivate the cluster. Limited functionality will be available in this state.
-state	Print the current cluster state.
-baseline	This command is designed to manage the baseline topology. When this command used without any parameters, it prints the current cluster baseline topology information. The following parameters can be used with this command: add, remove, set, and version.

To invoke a specific command, use the following pattern:

UNIX/LINUX/MacOS:

```
$IGNITE_HOME/bin/control.sh <command> <args>
```

Windows:

```
$IGNITE_HOME/bin/control.bat <command> <args>
```

Now, activate the cluster. Run the following command:

**Listing 10.8**

---

```
$IGNITE_HOME/bin/control.sh
```

---

If the command succeeds, you should see the following messages in the console.

```
Control utility [ver. 2.6.0#20180710-sha1:669feacc]
```

```
2018 Copyright(C) Apache Software Foundation
```

```
User: shamim
```

---

```
-----
```

```
Cluster activated
```

At this moment, you can also use the `--state` command to check the current cluster state. The `--state` command should return a message that the cluster is activated.

**Step 5.** Now, create a table and populate some data. We use the *SQLLINE* tool to connect to the cluster. Run the following command to start the *SQLLINE* tool:

**Listing 10.10**


---

```
sqlline.sh --color=true --verbose=true -u jdbc:ignite:thin://127.0.0.1/
```

---

Create a table named *EMP* and insert 1000 rows into the table. Use the following DDL script to create the *EMP* table as follows:

**Listing 10.11**


---

```
CREATE TABLE IF NOT EXISTS EMP
```

---

```
(  
    empno  LONG,  
    ename   VARCHAR,  
    job    VARCHAR,  
    mgr    INTEGER,  
    hiredate DATE,  
    sal     LONG,  
    comm    LONG,  
    deptno  LONG,  
    CONSTRAINT pk_emp PRIMARY KEY (empno)  
) WITH "template=partitioned,CACHE_NAME=EMPcache";
```

---

Next, use the *EMP\_001.sql* script from the GitHub repository to insert 1000 entries into the table.

**Listing 10.12**


---

```
0: jdbc:ignite:thin://127.0.0.1/> !run /PATH_TO_THE_FILE/the-apache-ignite-book/chapters/A  
chapter-10/baseline/EMP_001.sql
```

---

The command above inserts 1000 entries into the *EMP table* or *EMPcache*. Use the visor CLI tools to see the size of the cache into the entire cluster. Run the command `cache -a` in the Ignite Visor console. The command should return the following output as shown in figure 10.4.

Nodes for: EMPcache(@c1)								
Node ID	IP	CPU	Heap Used	CPU Load	Up Time	Size	Hi/Mi/Rd/Wr	
F0A6A794(@n0)	192.168.1.35	8	23.56 %	0.20 %	01:59:14.478	Total: 504	Hi: 0 Mi: 0 Rd: 0 Wr: 0	
						Heap: 0 Off-Heap: 504		
						Off-Heap Memory: 0		
6DB02F31(@n1)	192.168.1.35	8	14.30 %	0.20 %	01:57:39.172	Total: 496	Hi: 0 Mi: 0 Rd: 0 Wr: 0	
						Heap: 0 Off-Heap: 496		
						Off-Heap Memory: 0		

Figure 10.4

Take a look at the column named *SIZE*. This column clarifies the number of entries stored into each node. In our case, one of our nodes contains 504 entries, and the other one contains 496 entries into the EMPcache cache.

**Step 6.** So far, we have launched only two Ignite nodes and created a baseline topology in the cluster. Let's start another Ignite node. Do the same thing again (step 3) as before for the 3<sup>rd</sup> Ignite node.

```
export IGNITE_HOME=PATH_TO_THE_IGNITE_NODE_ONE/ignite/2.6.0-s3
ignite.sh /PATH_TO_THE_SPRING_CONFIG_FILE/ignite/ignite-book-baseline.xml
```

Logs on the console should assure you that the node is started on persistence mode successfully. Moreover, you should get a warning on the console that the local node is not included in Baseline Topology and will not be used for persistent data storage. Now we can play with the `--baseline` command. Let's run the command without any parameter as follows:

**Listing 10.13**

---

```
$IGNITE_HOME/bin/control.sh --baseline
```

---

The output might be as follows:

```
shamim:~ shamim$ control.sh --baseline
Control utility [ver. 2.6.0#20180710-sha1:669feacc]
2018 Copyright(C) Apache Software Foundation
User: shamim

-----
Cluster state: active
Current topology version: 6

Baseline nodes:
  ConsistentID=1640f655-4065-438c-92ca-478b5df91def, STATE=ONLINE
  ConsistentID=d8b04bc3-d175-443c-b53f-62512ff9152f, STATE=ONLINE

-----
Number of baseline nodes: 2

Other nodes:
  ConsistentID=3c2ad09d-c835-4f4b-b47a-43912d04d30e
Number of other nodes: 1
```

The above baseline information shows the cluster state, topology version, nodes with their consistent IDs that are part of the baseline topology as well as those that are not part of the

baseline topology. Here, the number of baseline nodes is two, and the baseline consists of our 1<sup>st</sup> and 2<sup>nd</sup> Ignite node.

Sometime it may occur that, during the first cluster activation the baseline topology was not created. In such cases, the --baseline command will return a message like “*Baseline nodes not found.*” In this situation, stop the 3<sup>rd</sup> node and waits for a few seconds. Then set the baseline topology manually by using the *numerical cluster topology version* as follows:

```
control.sh --baseline version topologyVersion
```

In the above command, replace the topologyVersion with the actual topology version. You can find the topology version in any Ignite node console as shown below:

```
Topology snapshot [ver=6, servers=3, clients=0, CPUs=8, offheap=9.6GB, heap=3.0GB]
```

Pick the latest topology snapshot version from the console.

In this stage, our 3<sup>rd</sup> Ignite node is *not the part* of our baseline topology. This node will not be used for persistent data storage. It means, if we will create any new tables and insert data into it, the node will not store any data for the new table. Let’s verify the concept.

**Step 7.** Create a new table DEPT with the following DDL script:

**Listing 10.14**

---

```
CREATE TABLE IF NOT EXISTS DEPT
(
    deptno LONG,
    dname VARCHAR,
    loc  VARCHAR,
    CONSTRAINT pk_dept PRIMARY KEY (deptno)
) WITH "template=partitioned,CACHE_NAME=DEPTcache";
```

---

Also, insert 100 departments by using the DEPT.SQL. The DEPT.SQL script is available at the GitHub repository<sup>181</sup>.

---

<sup>181</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-10/baseline>

**Listing 10.14**


---

```
0: jdbc:ignite:thin://127.0.0.1/> !run /PATH_TO_THE_FILE/github/the-apache-ignite-book/chapters/chapter-10/baseline/DEPT.sql
```

---

Now, run the command `cache -a` in the visor console which should print a similar output as shown in figure 10.5 below.

Nodes for: DEPTcache(@c2)								
	Node ID8(@), IP	CPU	Heaps Used	CPU Load	Up Time	Size	Hi/Mi/Rd/Wr	
1	70BF070B(@n2), 192.168.1.35	8	19.98 %	0.23 %	00:12:58.349	Total: 0	Hi: 0	
						Heap: 0	Mi: 0	
						Off-Heap: 0	Rd: 0	
						Off-Heap Memory: 0	Wr: 0	
1	F0A6A794(@n0), 192.168.1.35	8	21.12 %	0.17 %	02:27:40.466	Total: 44	Hi: 0	
						Heap: 0	Mi: 0	
						Off-Heap: 44	Rd: 0	
						Off-Heap Memory: 0	Wr: 0	
1	6DB02F31(@n1), 192.168.1.35	8	9.55 %	0.20 %	02:26:05.159	Total: 56	Hi: 0	
						Heap: 0	Mi: 0	
						Off-Heap: 56	Rd: 0	
						Off-Heap Memory: 0	Wr: 0	

Figure 10.5

The above figure confirms that the 3<sup>rd</sup> node does not contain any persistence data. However, the node that is not part of the baseline topology can participate in any *in-memory computing*.

**Step 8.** Next, let's add the new empty node to the baseline topology to hold persistence data. Invoke the command `--baseline add <node's consistentId>` to add the new node to the existing baseline.

**Listing 10.15**


---

```
control.sh --baseline add 3c2ad09d-c835-4f4b-b47a-43912d04d30e
```

---

In the command above, replace the consistent id `3c2ad09d-c835-4f4b-b47a-43912d04d30e` with your *consistentId* of the 3<sup>rd</sup> Ignite node. After completing the *–baseline add command*, a message will confirm that the new baseline topology contains three nodes.

Cluster state: active

Current topology version: 10

Baseline nodes:

```
ConsistentID=1640f655-4065-438c-92ca-478b5df91def, STATE=ONLINE
ConsistentID=3c2ad09d-c835-4f4b-b47a-43912d04d30e, STATE=ONLINE
ConsistentID=d8b04bc3-d175-443c-b53f-62512ff9152f, STATE=ONLINE
```

---

Number of baseline nodes: 3

Other nodes not found.

After forming the new baseline topology from three nodes, a data rebalancing will proceed immediately. The new empty node (in our case it's the 3<sup>rd</sup> node) will receive this portion of data from other nodes. You can confirm the data rebalancing if you run the `cache -a` command in Ignite Visor CLI again. The figure 10.6 shows the result of the data rebalancing after adding the 3<sup>rd</sup> node in the baseline topology.

Nodes for: EMPcache(@c1)								
Node ID@(), IP	CPU	Heaps Used	CPU Load	Up Time	Size	Hi/Mi/Rd/Wr		
70BF070B(@n2), 192.168.1.35   8   20.94 %   0.20 %   01:29:50.557   Total: 307   Hi: 0								
Heap: 0   Mi: 0								
Off-Heap: 307   Rd: 0								
Off-Heap Memory: 0   Wr: 0								
+-----+-----+-----+-----+-----+-----+-----+-----+-----+								
FC215913(@n1), 192.168.1.35   8   18.48 %   0.23 %   00:06:00.368   Total: 347   Hi: 0								
Heap: 0   Mi: 0								
Off-Heap: 347   Rd: 0								
Off-Heap Memory: 0   Wr: 0								
+-----+-----+-----+-----+-----+-----+-----+-----+-----+								
F0A6A794(@n0), 192.168.1.35   8   18.81 %   0.20 %   03:44:32.674   Total: 351   Hi: 0								
Heap: 0   Mi: 0								
Off-Heap: 351   Rd: 0								
Off-Heap Memory: 0   Wr: 0								

Figure 10.6

Now, each node stores almost evenly partition of entries (about 300 entries) for cache `EMPcache`. However, what will happen if one of the baseline topology nodes will be restarted? Let's stop one node and try to insert some data into the table `EMP`.

**Step 9.** Stop the 2<sup>nd</sup> node by hitting the key `CRTL+X`. Execute the command `--baseline` without any parameter to print the state of the baseline topology.

```
control.sh --baseline
```

The above command will display the current baseline topology status very similar to the next message:

---

Cluster state: active

Current topology version: 11

Baseline nodes:

ConsistentID=1640f655-4065-438c-92ca-478b5df91def, STATE=OFFLINE

ConsistentID=3c2ad09d-c835-4f4b-b47a-43912d04d30e, STATE=ONLINE

ConsistentID=d8b04bc3-d175-443c-b53f-62512ff9152f, STATE=ONLINE

---

Number of baseline nodes: 3

Other nodes not found

One of the nodes in offline as expected. Now try to insert some data into the EMP table by SQLLINE tool as follows:

```
insert into EMP (empno, ename, job, mgr, hiredate, sal, comm, deptno) values (2009, 'Sallie', 'Sales Associate', 96, null, 3619, 34, 78);
insert into EMP (empno, ename, job, mgr, hiredate, sal, comm, deptno) values (2010, 'Corinne', 'Human Resources Manager', 65, null, 1291, 86, 57);
insert into EMP (empno, ename, job, mgr, hiredate, sal, comm, deptno) values (2011, 'Myrtle', 'VP Quality Control', 88, null, 5103, 21, 48);
insert into EMP (empno, ename, job, mgr, hiredate, sal, comm, deptno) values (2012, 'Chesley', 'Desktop Support Technician', 46, null, 6352, 29, 21);
```

You should notice that a few inserts statement *failed with errors* which is shown in the next snippet.

```
Caused by: class org.apache.ignite.internal.cluster.ClusterTopologyServerNotFoundException
n: Failed to map keys for cache (all partition nodes left the grid).
    at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridNearAtomicSingleUpdateFuture.mapSingleUpdate(GridNearAtomicSingleUpdateFuture.java:562)
```

This error occurred because we have no backup copies for our EMP table; the node that should store the data has been stopped, and Ignite cannot be able to store the data. To avoid such a situation, consider a cache/table with at least one backup. If one node fails, it will lose no data. For now, we have a few options:

- Reboot the offline node as soon as possible with minimal downtime for preventing data loss.

- Remove the offline node from the baseline topology and rebalancing the data.

**Step 10.** Let's remove the offline node from the baseline topology. Execute the following command:

**Listing 10.16**

---

```
control.sh --baseline remove 1640f655-4065-438c-92ca-478b5df91def
```

---

After completing the remove command, the baseline topology changed excluding the stopped node. Note that by removing a node from the baseline topology, you acknowledge that you will no longer be able to use the data stored on that node after its restart. At this moment, no error will occur during data manipulation into the cluster. You can insert new entries or update existing entries into the cache successfully.



## Tip

Note that, the node that you want to remove from the baseline topology should be disconnected from the cluster before removing from the baseline. Otherwise, the error *Failed to remove nodes from baseline* occurs, specifying the nodes that you have to stop before deleting from the baseline.

In addition to topology management, `control.sh` script can also be used for monitoring and control a cluster state which is well documented in Ignite documentation. So, please refer to the control script section of the Ignite documentation for more information. In the next which is the final part of this chapter, we will discover a few technics for monitoring the Ignite cluster.

## Monitoring Ignite cluster

At this point, your Ignite cluster is configured and running. Applications are using the Ignite cluster for writing and reading data to and from it. However, Ignite is not a set-it-and-forget-it system. Ignite is a JVM based system and designed to *fast fail*. So, it requires monitoring for acting on time.

Ignite is built on JVM and JVM can use the [JMX<sup>182</sup>](#) or Java Management Extension. In other words, you can manage the system remotely by using JMX, gathering metrics (cache or

---

<sup>182</sup><https://docs.oracle.com/javase/8/docs/technotes/guides/jmx/intro.html>

memory) including the memory, CPU, threads, or any other part of the system that has been instrumented in JMX. Instrumentation enables the application or system to provide application-specific information to be collected by the external tools.



## Info

JMX was introduced in Java 5.0 release to manage & monitor the resources at runtime. Using JMX, we can monitor memory usage, garbage collection, loaded classes, thread count, etc. over time.

MBeans or Managed Beans are a particular type of JavaBeans that takes a resource inside the application or the JVM available externally. Figure 10.7 shows a high-level architecture of the JMX.

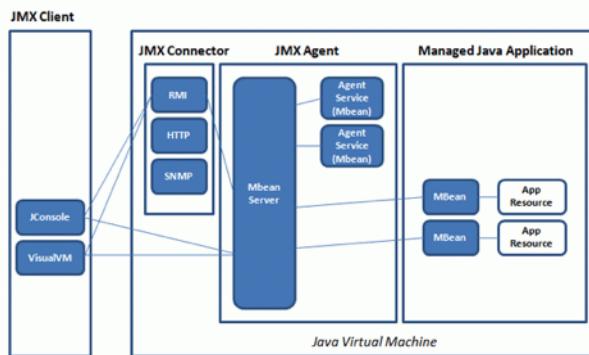


Figure 10.7

Apache Ignite provides a few JMX MBeans for collection and monitoring cache and memory metrics as follows:

- CacheMetricsMXBean. MBean that provides access to cache descriptor.
- CacheGroupMetricsMXBean. MBean that provides metrics for caches associated with a particular CacheGroup.
- DataRegionMetricsMXBean. MBean that provides access to *DataRegionMetrics* of a local Apache Ignite node.
- DataStorageMetricsMXBean. An MBean allowing to monitor and tune persistence metrics.

A few more new MBeans will be added in the subsequent Apache Ignite releases soon.

The standard tools that ships with Java for managing the MBeans is [JConsole<sup>183</sup>](#) or [VisualVM<sup>184</sup>](#). In the case of VisualVM you have to install the [VisualVM-MBeans plugin<sup>185</sup>](#). VisualVM is like JConsole but with more advanced monitoring featuring such as CPU profiling and GC visualization.



## Tip

Ignite allows running a VisualVM instance from IgniteVisor command. Use the `ignitevisor vvm` command to open VisualVM for an Ignite node in the topology.

## VisualVM

VisualVM is a GUI tool for monitor JVM. It helps the application developers and architects to track memory leaks, analyze the heap data, monitor the JVM garbage collector and CPU profiling. Moreover, after installing the *VisualVM-MBeans* plugin, you can manage and collect metrics from JMX MBeans provides by the application. VisualVM can be also used for monitoring the local and the remote Java process as well.

As we stated before, for monitoring the Ignite process you can lunch VisualVM with two different ways:

- Use `ignitevisor vvm` command to open a VisualVM instance or
- lunch a VisualVM instance manually by the `jvisualvm.exe|sh` from the `$JAVA_HOME/bin` folder.

IgniteVisor VVM command under the cover uses the default JDK installation to run the local VisualVM tool. Let's execute an Ignite node, create a table and populate some test data into the table. I am going to use the *EMP* table and data from the previous section. If you are having any trouble to create the table, please refer to step 5 of the previous section.

**Step 1.** Lunch the VisualVM application from the *JDK bin directory*. On top-left corner of the application tab, you can see different options like Local, Remote and Snapshots. Select the `org.apache.ignite.startup.cmdline.CommandLineStartup` application from the *Local* section as shown below.

<sup>183</sup><https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html>

<sup>184</sup><https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/>

<sup>185</sup><https://visualvm.github.io/plugins.html>

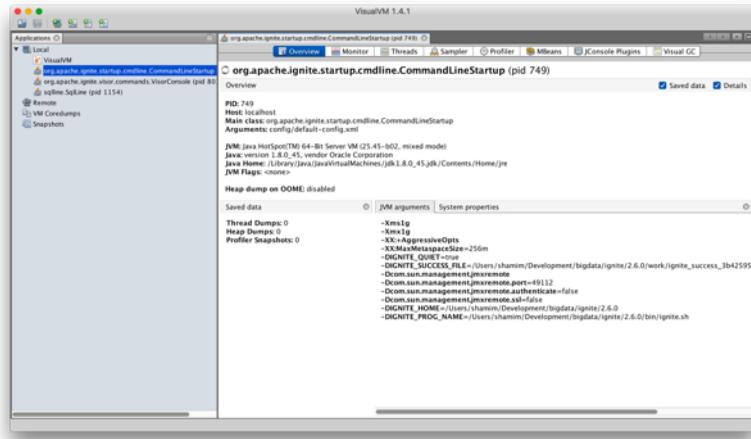


Figure 10.8

By default when Ignite node is started with `ignite.sh|bat` script, it picks up a random JMX port and binds to it. You can explicitly set the JMX port by setting the `IGNITE_JMX_PORT` environmental variable. In \*nix system it can be done in the following way:

```
export IGNITE_JMX_PORT=55555
```

However, if you run the Ignite node programmatically (i.e., by using Eclipse/IntelliJ IDEA), then the environmental variable `IGNITE_JMX_PORT` will not work. In such a situation, you need to pass the system parameters to your Java process that calls `Ignition.start` as follows:

```
-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port={PREFERRED_PORT}
-Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
```

The Memory, and Threads tabs are sets of graphs that provide insight into the current state of the application. The monitor tab consists of graphs about the current state of the Java heap, CPU, Classes and threads (see Figure 10.8).

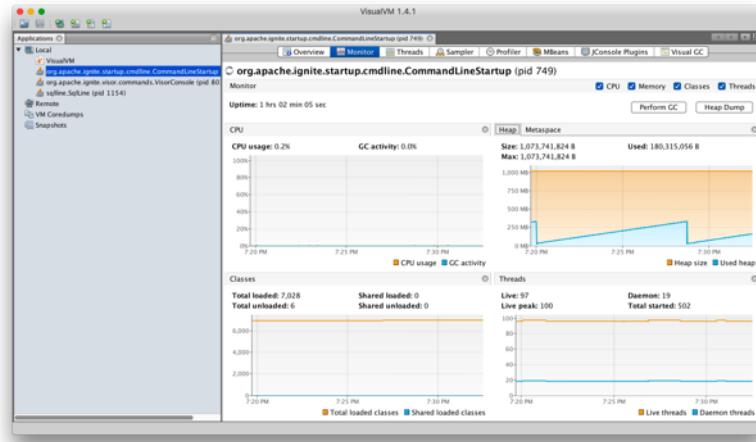


Figure 10.9

The Classes graph is merely a graph of how many classes are loaded into the JVM at the current time. One of the most important metrics to be aware of is your current heap usage. Ignite uses off-heap memory to store data from version 2.0 by default, so it is unnecessary to use a large heap size for Ignite node. By using small heap size, you reduce the memory footprint on the system and possibly speed up the GC process.

**Step 2.** Let's open the tab MBeans. There are many MBeans that are useful for assessing the state of the Ignite node. You will notice that there are few grouping here that can be expanded. All the Ignite MBeans classpath starts with the org.apache. Expand the group *Cache Group* under *18b4aac2* and click on the *EMPCache MBean* as shown in figure 10.10.

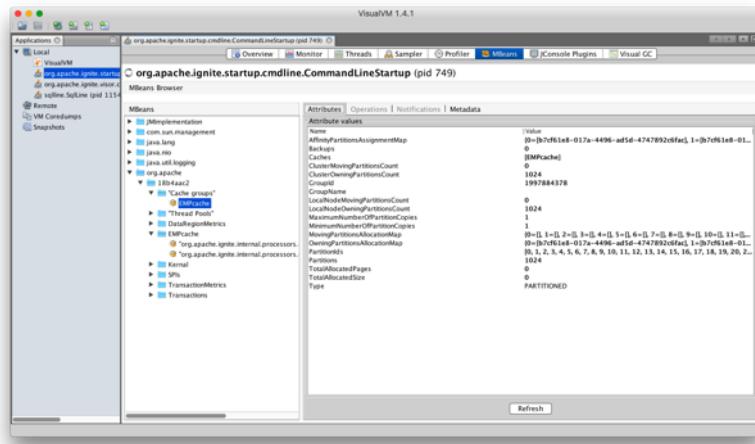


Figure 10.10

You should notice many import attributes shown for the EMPcache cache. Click on the value of the *LocalNodeOwningPartitionsCount* attribute, and a simple chart should pop up and show the current total number of partitions for the cache.

When you select an MBean in the tree, its MBeanInfo and MBean descriptor are displayed on the right-hand side of the window. If any additional attributes, operations or notifications are available, they appear in the tree as well below the selected MBean. As a high-level overview, they are broken down into the following categories:

- *Cache Groups*. The MBeans stored in this section cover everything about the actual data storage part of Ignite. This MBeans provides information about the caches itself: Affinity partitions assignment map, total backups, collections of the partitions as well as total partition number.
- *Kernel*. In the Kernel section, some MBeans cover the basic information about the node. For example, IgniteKernel MBean provides you the information about node Uptime, local node id or Peer class loading option.
- *SPIs*. The MBeans in the SPI's section covers the information about node discovery and internode communication. These include informations such as Node fails, Network timeout.
- *TransactionMetrics*. The metrics available in this section are closely related to transactions. These are things like LockedKeysNumber, TransactionRollbackNumber.

Each one of these sections of MBeans provides access to a large amount of information, giving you insight into both the system as a whole and the individual nodes. There is no

need to cover all of them as you can easily explore them on your own using VisualVm GUI interface.

Using JConsole/VisualVM to monitor a local application or Ignite node is useful for development or prototyping. Monitoring an Ignite cluster over 5 nodes by VisualVM or JConsole is unrealistic and time-consuming. Also, JMX does not provide any historical data. So, it is not recommended for production environments. Nowadays there is a lot of tools/software available for system monitoring. Most famous of them are:

- [Nagios<sup>186</sup>](#)
- [Zabbix<sup>187</sup>](#)
- Grafana, etc.

In the next section, we cover the Grafana for monitoring Ignite node and provide step-by-step instructions to install and configure the entire stack technology.

## Grafana

[Grafana<sup>188</sup>](#) is an open-source graphical tool dedicated to query, visualize and alert on for all your metrics. It brings your metrics together and lets you create graphs and dashboards based on data from various sources. Also, you can use Grafana to display data from different monitoring systems like Zabbix. It is lightweight, easy to install, easy to configure, and it looks beautiful.

Before we dive into the details, let's discuss the concept of monitoring large-scale production environments. Figure 10.11 illustrated a high-level overview of how the monitoring system looks like on production environments.

---

<sup>186</sup><https://www.nagios.org>

<sup>187</sup><https://www.zabbix.com>

<sup>188</sup><https://grafana.com/grafana>

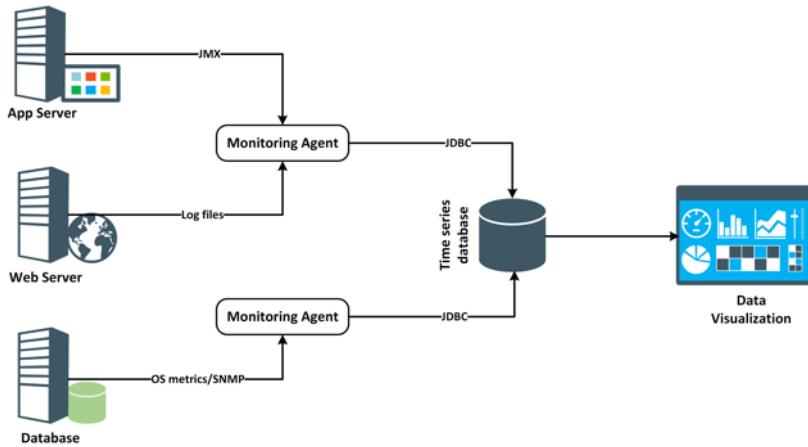


Figure 10.11

In the above architecture, data such as OS metrics, log files, and application metrics are gathering from various hosts through different protocols like JMX, SNMP into a single time-series database. Next, all the gathered data is used to display on a dashboard for real-time monitoring. However, a monitoring system could be complicated and vary in different environments, but the basic is the same for all.

Let's start at the bottom of the monitoring chain and work our way up. To avoid a complete lesson on monitoring, we will only cover the basics along with what the most common checks should be done as they relate to Ignite and its operation. The data we are planning to use for monitoring are:

- Ignite node Java Heap.
- Ignite cluster topology version.
- Amount of server or client nodes in cluster.
- Ignite node total up time.

The stack technologies we use for monitoring the Ignite cluster comprise three components: [InfluxDB<sup>189</sup>](https://www.influxdata.com/), [Grafana](#), and [jmxtrans<sup>190</sup>](https://github.com/jmxtrans/jmxtrans). The high-level architecture of our monitoring system is shown in figure 10.12 below.

<sup>189</sup><https://www.influxdata.com/>

<sup>190</sup><https://github.com/jmxtrans/jmxtrans>

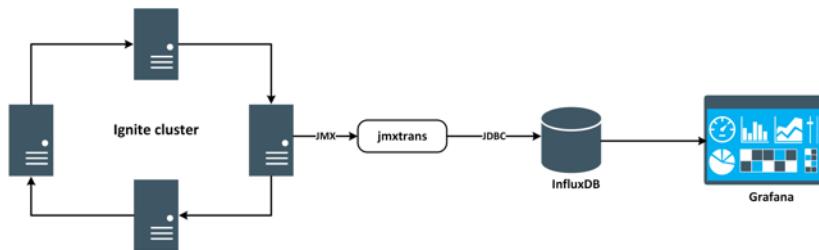


Figure 10.12

Ignite nodes does not send the MBeans metrics to the InfluxDB directly. We use *jmxtrans* which collects the JMX metrics and send to the InfluxDB. *Jmxtrans* is lightweight and running as a daemon to collect the server metrics. InfluxDB is an open-source time series database developed by InfluxData. It is written in Go and optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring and application metrics.

In the next sub-sections, we install and configure InfluxDB, Grafana, and *jmxtrans* to collect metrics from the Ignite cluster. We also compose a custom dashboard in Grafana that monitors Ignite cluster resources.

**Prerequisites.** To follow this instruction to configure the monitoring infrastructure, you need the following:

Name	Version
OS	MacOS, Windows, *nix
InfluxDB	1.7.1
Grafana	5.4.0
<i>jmxtrans</i>	271-SNAPSHOT

**Step 1.** The data store for all the metrics from the Ignite cluster will be [InfluxDB<sup>191</sup>](#). Let's install the InfluxDB first. I am using MacOS so; I will use Homebrew to install InfluxDB. Please visit the InfluxDB website and follow the instruction to install for other operating systems like Windows or Linux.

#### Listing 10.17

---

```
brew install influxdb
```

---

After completing the installation process, launch the database by the following command:

<sup>191</sup><https://docs.influxdata.com/influxdb/v1.7/introduction/downloading/>

**Listing 10.18**

```
influxd -config /usr/local/etc/influxdb.conf
```

---

InfluxDB running on `http://localhost:8086` and provides *REST API* for manipulating the database objects by default. Also, InfluxDB provides a command line tool named **influx** to interact with the database. Execute the *influx shell script* on another console which starts the CLI and automatically connects to the local InfluxDB instance. The output should look like this:

**Listing 10.19**

```
influx
```

```
Connected to http://localhost:8086 version v1.7.1
InfluxDB shell version: v1.7.1
Enter an InfluxQL query
```

---

A fresh install of InfluxDB has no *database*, so let's create a database to store the Ignite metrics. Enter the following Influx Query Language (a.k.a InfluxQL) statement to create the database.

**Listing 10.20**

```
create database ignitesdb
```

---

Now that the `ignitesdb` database is created, we can use the `SHOW DATABASES` statement to display all the existing databases.

**Listing 10.21**

```
show databases
```

```
name: databases
name
---
_internal
ignitesdb
```

---

Note that, the `_internal` database is created and used by InfluxDB to store internal runtime metrics. To insert or query the database, use `USE <db-name>` statement, which will automatically set the database for all future requests. For example:

**Listing 10.22**

---

```
USE ignitesdb
Using database ignitesdb
```

---

**Step 2.** Again, I am using Homebrew for installing *Grafana* as follows:

**Listing 10.23**

---

```
brew install grafana
```

---

It takes a few minutes to compile and install Grafana on the local machine. Alternatively, you can also use standalone MacOS/Darwin Binaries to install Grafana. We use the default settings for the purpose of this book. You can change Grafana settings in the following two locations:

/usr/local/etc/grafana/grafana.ini  
/usr/local/var/lib/grafana/grafana-server

A full guide of configuration options for Grafana can be found on their [website](#)<sup>192</sup>. Now that, you have completed the installation of Grafana, let's start the server by using the following command:

**Listing 10.24**

---

```
grafana-server --config=/usr/local/etc/grafana/grafana.ini --homepath /usr/local/share/grafana
cfg:default.paths.logs=/usr/local/var/log/grafana cfg:default.paths.data=/usr/local/var/lib/grafana
cfg:default.paths.plugins=/usr/local/var/lib/grafana/plugins
```

---

We can now browse the Grafana's web interface on `http://localhost:3000` and log in with `admin/admin` (the default credentials, which you must change at the first login).

**Step 3.** Next, we need to configure the data source. We do this by logging into Grafana using the URL `http://localhost:3000` and credentials above, clicking the Grafana logo at the top left corner, and then selecting data sources from the dropdown list. Then, select *Add new data source* option.

---

<sup>192</sup><https://grafana.com/grafana>

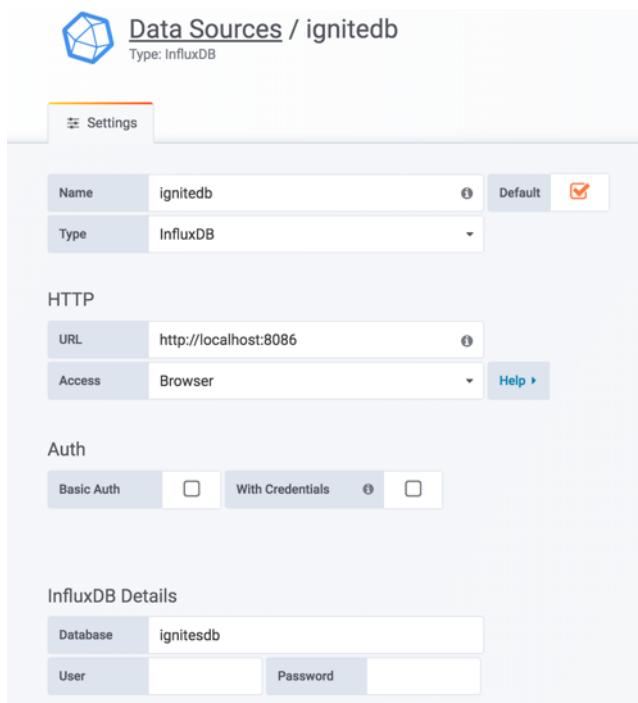


Figure 10.13

You should be greeted with a screen similar to the one above. We have to specify the database and fill some fields to get connectivity working.

- type: InfluxDB
- URL: http://localhost:8086
- Access: Browser
- Database: ignitesdb

Click the `save and test` button which saves and test the configuration. You should get a confirmation that the data source is working.

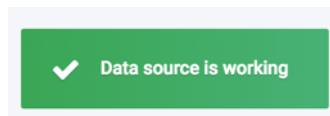


Figure 10.14

We have connected the InfluxDb and Grafana together, and now we need to push some data into InfluxDB.

**Step 4.** At this moment, we need to download and install *jmxtrans* on the local machine. Download a recent stable build from the [central maven repository](#)<sup>193</sup>. Unarchive the jmxtrans-VERSION\_NAME-dist.tar.gz distribution in your operating system. There is a jmxtrans.sh script in the *bin* directory included with the distribution. This script should be used to start the application running. Once installed, we need to configure it for gathering some JVM metrics. Create a JSON file with name *jvm-metric.json*, and add the following content on it.

**Listing 10.25**

```
{  
  "servers": [ {  
    "port": "55555",  
    "host": "localhost",  
    "alias": "Ignite-node-1",  
    "queries": [ {  
      "outputWriters": [  
        {  
          "@class": "com.googlecode.jmxtrans.model.output.InfluxDbWriterFactory",  
          "url": "http://localhost:8086/",  
          "database": "ignitesdb",  
          "username": "admin",  
          "password": "qwer4321"  
        },  
        {  
          "obj": "java.lang:type=Memory",  
          "attr": [ "HeapMemoryUsage", "NonHeapMemoryUsage" ],  
          "resultAlias": "jvmMemory"  
        },  
        {  
          "outputWriters": [ {  
            "@class": "com.googlecode.jmxtrans.model.output.InfluxDbWriterFactory",  
            "url": "http://localhost:8086/",  
            "database": "ignitesdb",  
            "username": "admin",  
            "password": "qwer4321"  
          } ],  
          "obj": "java.lang:type=Threading",  
          "attr": [ "DaemonThreadCount", "PeakThreadCount", "ThreadCount", "TotalStartedThreadsCount" ],  
          "resultAlias": "threads"  
        } ]  
      ]  
    } ]  
  }]
```

<sup>193</sup><http://central.maven.org/maven2/org/jmxtrans/jmxtrans/>

```
    }
    ],
    "numQueryThreads" : 2
  ]
}
```

---

The `port` and `host` fields are access details to the Ignite node. We are using JMX port 55555 for gathering information from the Ignite MBeans. The `alias` field will uniquely identify an Ignite nodes metrics in Grafana. The `resultAlias` is a container where metrics will be available from. We will start by collecting the JVM memory and threads metrics. Later we will add more metrics related to Ignite cache.

Now, start the *jmxtrans* daemon on background mode. Use the following command from the `jmxtrans_home` directory (directory, where you unarchive the distribution):

**Listing 10.26**

---

```
bin/jmxtrans.sh start /FILE_PATH_TO/jvm-metrics.json
```

---

For now, the entire pipeline is ready. We need to set a *JMX PORT* for the Ignite node and start it. Set the environmental variable as follows:

```
export IGNITE_JMX_PORT=55555
```

Start an Ignite node from the console.

```
ignite.sh
```

Once lauched, the *jmxtrans* start collecting metrics from the Ignite node and sends the metrics information into *InfluxDB*. Before creating any dashboard on Grafana, you can query the InfluxDB to make sure that the data has been passed to the database. Use the console where you have started the *influx CLI* and run the following queries:

**Listing 10.27**


---

```
use ignitesdb
SHOW MEASUREMENTS

name: measurements
name
---

jvmMemory
threads
```

---

**SHOW MEASUREMENTS** SQL statements show all the measurements having in the ignitesdb database. Now query the jvmMemory measurements by the following statements:

```
select * from jvmMemory
```

The above SQL statement should return query results similar as shown below:

name: <u>jvmMemory</u>	HeapMemoryUsage_committed	HeapMemoryUsage_init	HeapMemoryUsage_max	HeapMemoryUsed
1029177344	1073741824	1029177344	87948536	
1029177344	1073741824	1029177344	108812464	

Figure 10.15

**Step 5.** At this moment, we are ready to create a dashboard with different graphs for monitoring the Ignite server. Let's head back to the Grafana in our web browser. Click the dropdown denoted by *Home* at the top, and create a new dashboard by clicking *Create New* at the bottom of the screen:

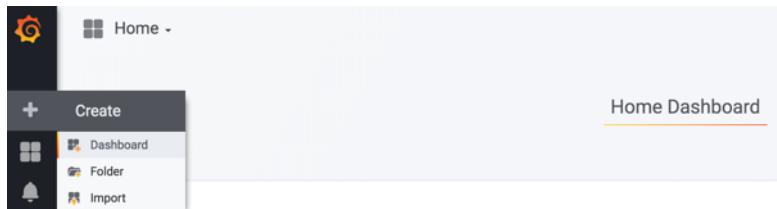


Figure 10.16

Next, click the tiny *Graph* button on the panel. Click on the *panel title* and select the *edit* option for configuring the panel with queries. First select the *ignitedb* as a data source and then select *jvmMemory* as a select measurement in the Metrics tab. Select *HeapMemoryUsage\_used*

as a field option. Set the `ALIAS BY` with `HeapUsages`. Add another query by clicking the *Add query button*. Select `threads` as a measurement and `ThreadCount` as a query field. After completing the configuration, you should have the following configuration shown in figure 10.16. Do not forget to save the panel configuration.

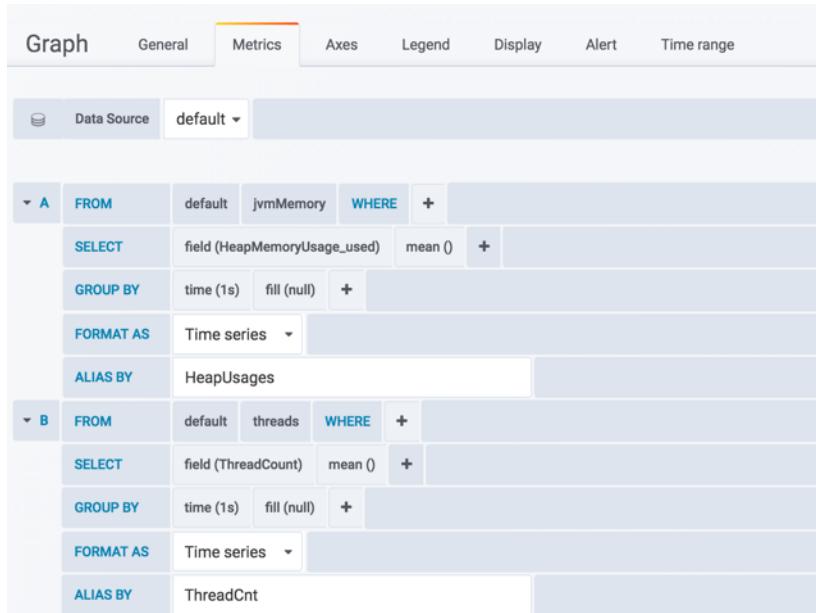


Figure 10.17

Now, when we have our Ignite node up and running, and the dashboard is ready to use, try to refresh the panel *ignite JVM metric*. After refreshing the panel, you should have the following view.

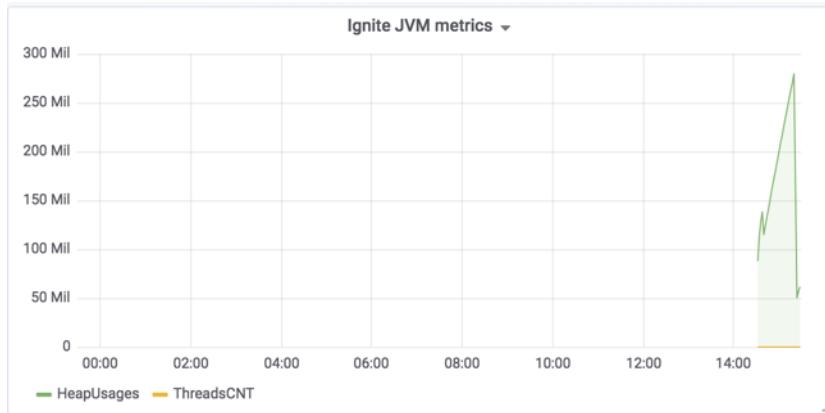


Figure 10.18

At this point, our entire pipeline from the Ignite cluster to Grafana is working. Now, we can add more output writers items and MBeans configuration into the jmxtrans jvm-metrics.json file for collecting more metrics from the Ignite node. Add the following output writers configuration after the output writers **threads**:

Listing 10.28

---

```
{
  "outputWriters": [
    {
      "@class": "com.googlecode.jmxtrans.model.output.InfluxDbWriterFactory",
      "url": "http://localhost:8086/",
      "database": "ignitesdb",
      "username": "admin",
      "password": "qwer4321"
    },
    {
      "obj": "org.apache:clsLdr=18b4aac2,group=Kernal,name=IgniteKernal",
      "attr": [ "UpTime", "StartTimestamp" ],
      "resultAlias": "IgniteKernal"
    },
    {
      "outputWriters": [
        {
          "@class": "com.googlecode.jmxtrans.model.output.InfluxDbWriterFactory",
          "url": "http://localhost:8086/",
          "database": "ignitesdb",
          "username": "admin",
          "password": "qwer4321"
        },
        {
          "obj": "org.apache:clsLdr=18b4aac2,group=Kernal,name=ClusterMetricsMXBeanImpl",
          "attr": [ "TopologyVersion", "TotalServerNodes" ],

```

```
"resultAlias" : "ClusterMetrix"
}
```

---

The full *jvm-metrics.json* file is available at the [Github project](#)<sup>194</sup>. Here, we have added two more MBeans configurations into the file as described below.

- org.apache:clsLdr=18b4aac2,group=Kernal,name=IgniteKernel. MBeans that provides access to Ignite Kernel information. The MBean provides Ignite node *UpTime* and *StartTimeStamp*.
- com.googlecode.jmxtrans.model.output.InfluxDbWriterFactory. MBean that provides access to aggregated cluster metrics. This MBean provides Ignite cluster *topology version* and *total server nodes*.

Save the file and **restart** the *jmxtrans* daemon.

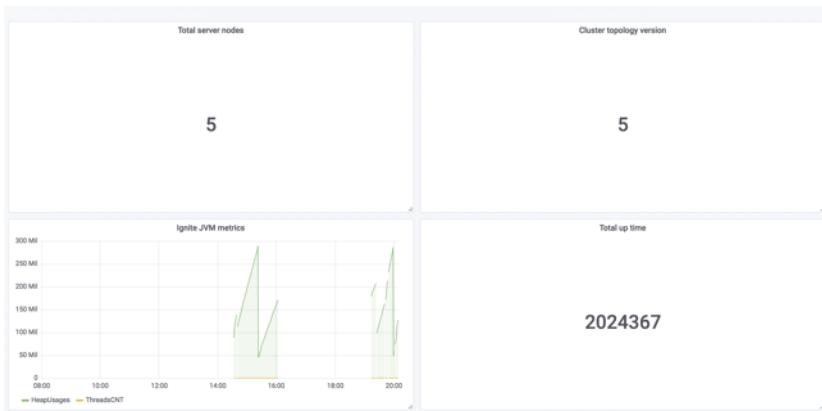
**Listing 10.29**

---

```
bin/jmxtrans.sh restart
```

---

**Step 6.** Add a few more *Singlestat* panel to display the Ignite topology version, total server nodes, and node uptime in the Grafana dashboard. For each panel, you have to configure the *data source*, *measurement*, and *fields*. If you do the configuration correctly, you will have a similar dashboard as shown in figure 10.18.



**Figure 10.19**

---

<sup>194</sup><https://github.com/srecon/the-apache-ignite-book/tree/master/chapters/chapter-10/grafana>

This has been a quick run through to get you set up. You will need to experiment with all aspects of the configuration to gain confidence in using jmxtrans and Grafana. However, there are tons of materials available on the internet for studying Grafana and jmxtrans. In conclusion, let's consider what are essential to monitor from Ignite perspective? The following table shows you the *key metrics* you can use in your dashboard to monitor the Ignite cluster.

Metric	Description
Topology version	The topology version changes when Ignite server or client nodes joins or leaves from the cluster.
Total nodes in cluster	A total amounts of servers and clients in the cluster.
Total Baseline nodes	A total amount of the nodes that participated in the baseline topology
Split Brain	Split brain occurs when the entire cluster is segmented — this information helps to determine the health of the cluster.
TCPDiscovery SPI	Message worker queue size.
TCPCommunication SPI	Outbound Messages Queue size.
GC pauses	How does garbage collector work for each Ignite node?
LRT	Long-running transactions. Transactions that are running in the cluster for a long time.
Checkpoint time	How long it took to write checkpoint on nodes?
Moving partitions	A number of partitions on nodes in moving state.
Locked key numbers	A total amount of blocked keys.
transactions holding lock number	The total amount of transactions that are blocked.

## Summary

Monitoring is an essential part of any application. With multiple tools at hand, you can wisely choose your weapon. Starting with VisualVM, which enables you to view the status of the Ignite internals, you may also configure and control some parameters through JMX. On the other hand, Ignite visor is a powerful utility. It can help you get the internal stats and performs many administrative tasks. However, it is a command-line tool. This is where Grafana comes into the action. With simple installation and the ability to store performance history and resource monitoring, it is one of the best options to monitor an Ignite cluster.