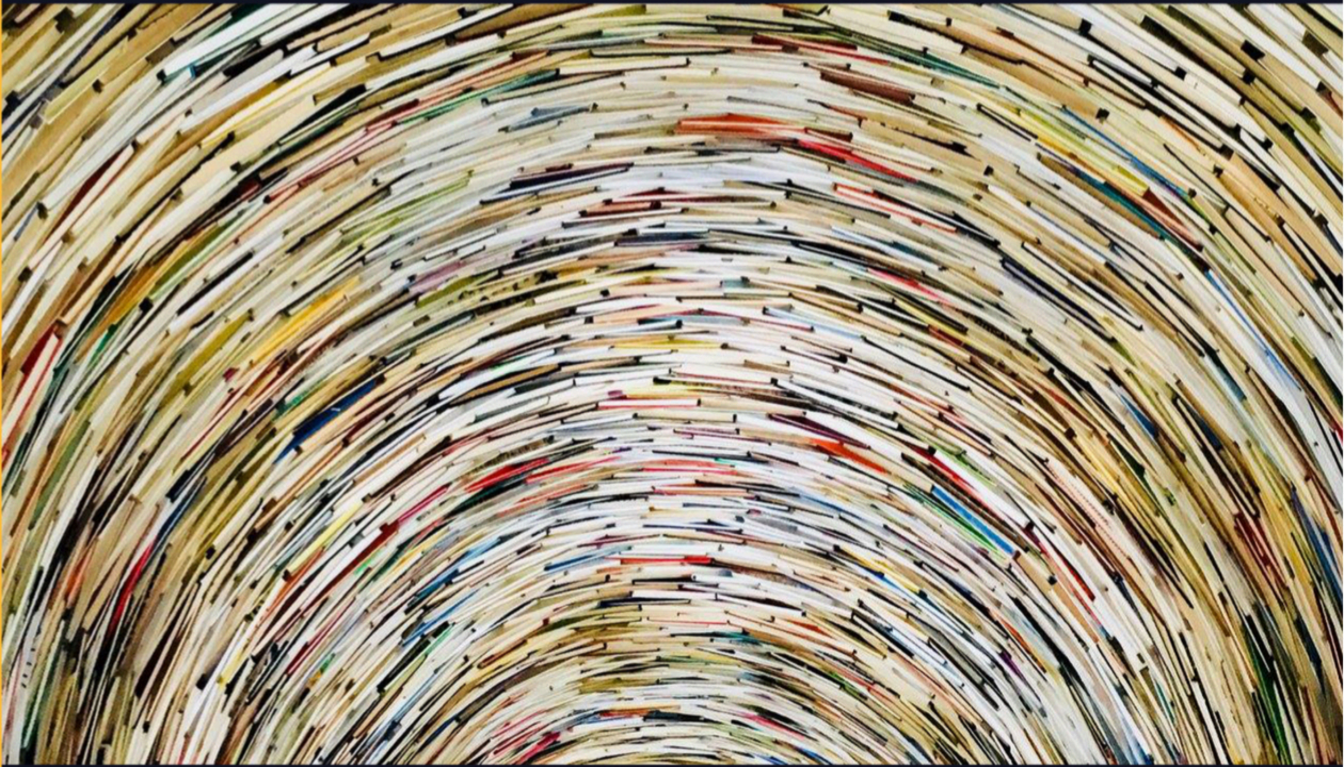
THE APACHE IGNITE BOOK

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 2023-02-12

ISBN 978-0-359-43937-9

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

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[1] and shares ideas with others.

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

[1]http://frommyworkshop.blogspot.ru/

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 that of the traditional NoSQL databases, able to simplify the building 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.

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[2] where multiple identical nodes

[2]https://en.wikipedia.org/wiki/Shared-nothing_architecture

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 also shut down if the application dies or is taken down. This topology is shown in figure 4.3.

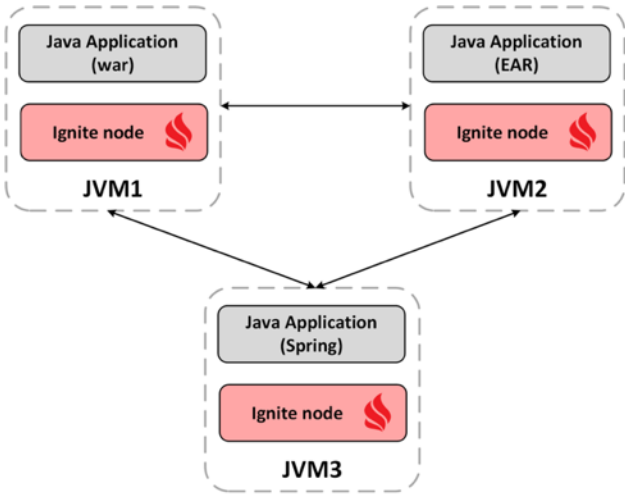
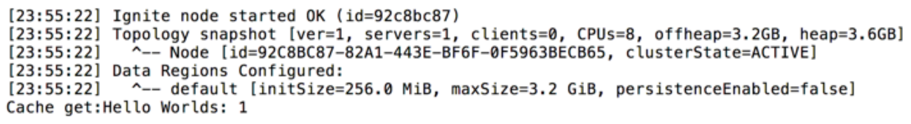


Figure 4.3

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 each other and forming an Ignite grid.

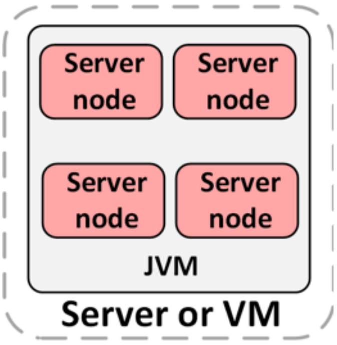


Figure 4.6

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 use 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.

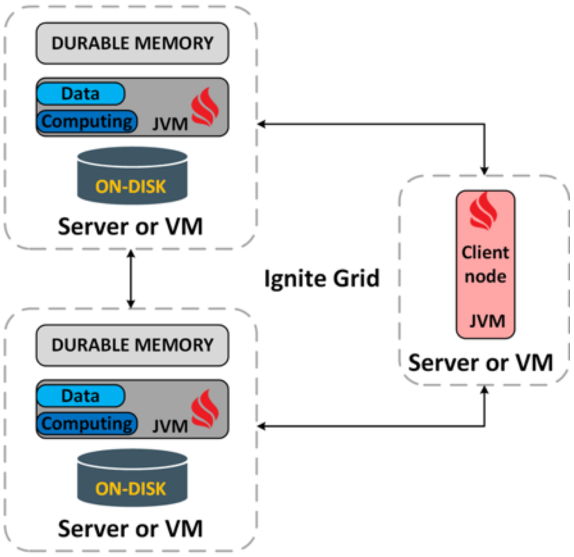


Figure 4.6.1

Such a cluster can be quickly deployed in and maintained by the kubernates[3] which an open source system for automating deployment, scaling, and management of the containerized application. VMWare[4] is another common cluster management system rapidly used for the Ignite cluster.

Data partitioning in Ignite

Data partitioning[5] 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:

[3]https://kubernetes.io
[4]https://www.vmware.com/solutions/virtualization.html
[5]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.

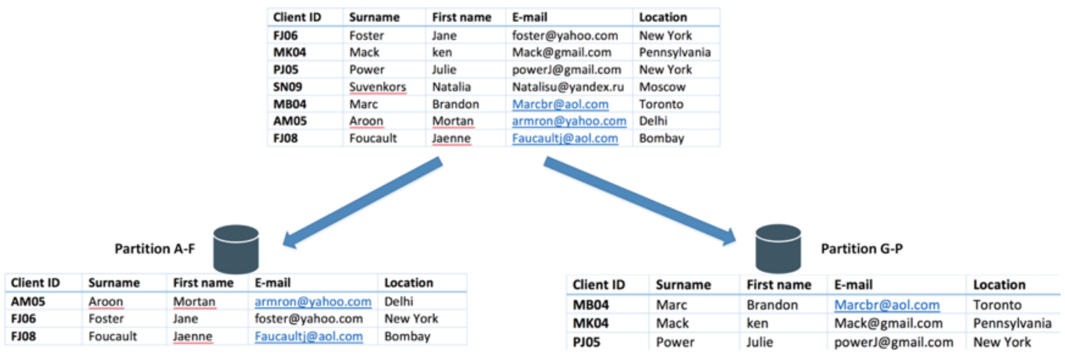
Tip

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.



Client ID	Surname	First name	E-mail	Location
FJ06	Foster	Jane	foster@yahoo.com	New York
MK04	Mack	ken	Mack@gmail.com	Pennsylvania
PJ05	Power	Julie	powerJ@gmail.com	New York
SN09	Suvenkors	Natalia	Natalisu@yandex.ru	Moscow
MB04	Marc	Brandon	Marcbr@aol.com	Toronto
AM05	Aroon	Mortan	armron@yahoo.com	Delhi
FJ08	Foucault	Jaenne	Faucaultj@aol.com	Bombay

Client ID	Surname	First name	E-mail	Location
AM05	Aroon	Mortan	armron@yahoo.com	Delhi
FJ06	Foster	Jane	foster@yahoo.com	New York
FJ08	Foucault	Jaenne	Faucaultj@aol.com	Bombay

Client ID	Surname	First name	E-mail	Location
MB04	Marc	Brandon	Marcbr@aol.com	Toronto
MK04	Mack	ken	Mack@gmail.com	Pennsylvania
PJ05	Power	Julie	powerJ@gmail.com	New York

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[6] 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[7] 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)[8] 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.

[6]https://en.wikipedia.org/wiki/Distributed_hash_table
[7]https://en.wikipedia.org/wiki/Hash_table
[8]https://en.wikipedia.org/wiki/Big_O_notation

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[9] (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 consistent hashing[10] 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. A numeric value is created with a standard hash function `hash(Ni, K)` to find out which node should store a given key, for each combination of the node (N) and key (K). 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.

[9]https://en.wikipedia.org/wiki/Rendezvous_hashing
[10]https://en.wikipedia.org/wiki/Consistent_hashing

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. It usually helps to read, write a page and even allocate a page *ID.*

When the allocated memory exhausted and the data are pushed to the persistence store, it happens page by page. So, a page size is crucial for performance, 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, 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, you can configure the size of the memory page in such cases.

Size of the page can be configured via *DataStorageConfiguration.setPageSize(..)* parameter. 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

Ignite memory page structure illustrated in the following figure 4.29.

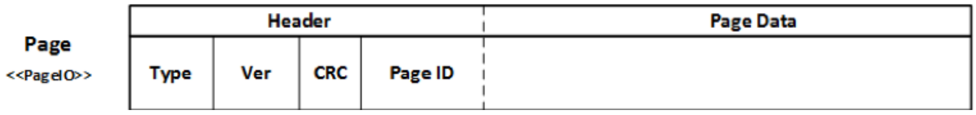


Figure 4.28

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*. 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 completely located in RAM or into Disk.

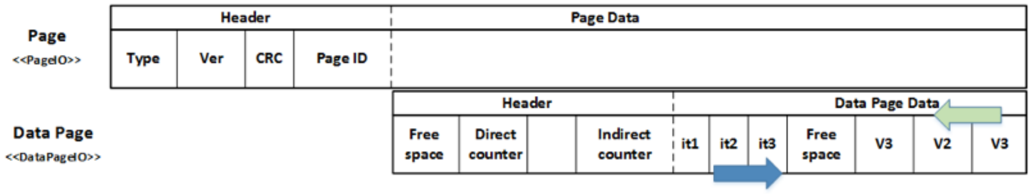


Figure 4.29

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.

Split-brain protection

Split-brain[14] 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:

[14] https://en.wikipedia.org/wiki/Split-brain_(computing)