HIGH PERFORMANCE IN-MEMORY COMPUTING WITH APACHE IGNITE

BUILDING LOW LATENCY, NEAR REAL-TIME APPLICATION

SHAMIM BHUIYAN, MICHAEL ZHELUDKOV
AND TIMUR ISACHENKO

High Performance in-memory computing with Apache Ignite

Building low latency, near real time application

Shamim Ahmed Bhuiyan, Michael Zheludkov and Timur Isachenko

This book is for sale at http://leanpub.com/ignite

This version was published on 2017-05-09

Tweet This Book!

Please help Shamim Ahmed Bhuiyan, Michael Zheludkov and Timur Isachenko by spreading the word about this book on Twitter!

The suggested hashtag for this book is #shamim_ru.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#shamim_ru

In memory of my Father. - Shamim Ahmed Bhuiyan

Contents

Preface

My first acquaintance with High load systems was at the beginning of 2007, and I started working on a real-world project since 2009. From that moment, I spent most of my office time with Cassandra, Hadoop, and numerous CEP tools. Our first Hadoop project (the year 2011-2012) with a cluster of 54 nodes often disappointed me with its long startup time. I have never been satisfied with the performance of our applications and was always looking for something new to boost the performance of our information systems. During this time, I have tried HazelCast, Ehcache, Oracle Coherence as in-memory caches to gain the performance of the applications. I was usually disappointed from the complexity of using these libraries or from their functional limitations.

When I first encountered Apache Ignite, I was amazed! It was the platform that I'd been waiting on for a long time: a simple spring based framework with a lot of awesome features such as DataBase caching, Big data acceleration, Streaming and compute/service grids.

In 2015, I had participated in Russian HighLoad++ conference[1] with my presentation and started blogging in Dzone/JavaCodeGeeks and in my personal blog[2] about developing High-load systems. They became popular shortly, and I received a lot of feedback from the readers. Through them, I clarified the idea behind the book. The goal of the book was to provide a guide for those who really need to implement an in-memory platform in their projects. At the same time, the idea behind the book is not writing a manual. Although the Apache Ignite platform is very big and growing day by day, we concentrate only on the features of the platform (from our point of view) that can really help to improve the performance of the applications.

We hope that *High-performance in-memory computing with Apache Ignite* will be the go-to guide for architects and developers: both new and at an intermediate level, to get up and to develop with as little friction as possible.

Shamim Ahmed

What this book covers

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 are and how can you design application with Apache Ignite for getting maximum performance from your application.

Chapter one - Installation and the first Ignite application walks you through the initial setup of an Ignite grid and running of some sample application. At the end of the chapter, you will implement

[1] http://www.highload.ru/2015/abstracts/1875.html

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

your first simple Ignite application to read and write entries from the Cache. You will also learn how to install and configure an SQL IDE to run SQL queries against Ignite caches.

Chapter two - Architecture overview covers the functional and architecture overview of the Apache Ignite data fabrics. Here you will learn the concepts and the terminology of the Apache Ignite. This chapter introduces the main features of Apache Ignite such as cluster topology, caching topology, caching strategies, transactions, Ignite data model, data collocation and how SQL queries works in Apache Ignite. You will become familiar with some other concepts like multi-datacenter replication, Ignite asynchronous support and resilience abilities.

Chapter three - In-memory caching presents some of the popular Ignite data grid features, such as 2nd level cache, java method caching, web session clustering and off-heap memory. This chapter covers developments and technics to improve the performance of your existing web applications without changing any code.

Chapter four - Persistence guides you through the implementation of transactions and persistence of the Apache Ignite cache. This chapter explores in depth: SQL feature and transaction of the Apache Ignite.

Chapter five - Accelerating Big Data computing, we focus on more advanced features and extensions to the Ignite platform. In this chapter, we will discuss the main problems of the Hadoop ecosystems and how Ignite can help to improve the performance of the exists Hadoop jobs. We detail the three main features of the Ignite *Hadoop accelerator*: in-memory Map/Reduce, IGFS, and Hadoop file system cache. We also provide examples of using Apache Pig and Hive to run Map/Reduce jobs on top of the Ignite in-memory Map/Reduce. At the end of the chapter, we show how to share states in-memory across different Spark applications easily.

Chapter six - 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 BigData technologies such as flume, storm, and camel to solve various business problems. We will guide you through with a few complete examples for developing real-time data processing on Apache Ignite.

Chapter seven - Distributive computing covers, how Ignite can help you to easily develop Microservice like application, which will be performed in parallel fashion to gain high performance, low latency, and linear scalability. You will learn about Ignite MapReduce & ForkJoin, Distributed closure execution, continuous mapping, etc. for data processing across multiple nodes in the cluster.

Code Samples

All code samples, scripts, and more in-depth examples can be found on GitHub at GitHub repo[3]

[3] https://github.com/srecon/ignite-book-code-samples

Support

Please reports bugs, errors and questions to authors by this link[4]. Bugs and errors in the code will be corrected and posted in a new version of the sample code of the book. Your feedback is very valuable and will be incorporated into the subsequent version of the book. Please do not hesitate to contact the authors, if you would like more information on some topics that have not been covered or explained in sufficient details.

[4] https://leanpub.com/ignite/email_author/new

About the authors

Shamim Ahmed Bhuiyan

He received his Ph.D. in Computer Science from the University of Vladimir, Russia in 2007. He is currently working as an Enterprise architect, where he is responsible for designing and building out highhighly scalable, and high load middleware solutions. He has been in the IT field for over 16 years and is specialized in Java 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[5] and shares ideas with others.

Michael Zheludkov

Is a senior programmer at AT Consulting. Graduated from the *Bauman Moscow State Technical University* in 2002.

Lecturer at BMSTU since 2013, delivering course *Parallel programming and distributed systems*.

Timur Isachenko

Is a Full Stack Developer working for AT-Consulting, passionate about web development and solving biga wide variety of related challenges.

Timur spends his time learning cutting-edge technologies every day to make a best developer out of himself.

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

Introduction

The term *high-performance computing* has recently become very popular in the IT world. *High-performance computing* refers to the practice of aggregating computing power in such a way that it delivers much higher performance than a typical desktop computer or workstation in order to solve large problems in science, engineering, or business.

High-performance computing is not only used to model complex physical phenomena such as weather or astronomical calculations. Very common use cases are to improve applications, reduce production costs and decrease development times. Also, as our ability to collect Big Data increases, the need to analyze the data also increases. High-performance computing allows processing this data as quickly as possible.

We can achieve 2-3x times the performance of a workstation by aggregating a few computers together in one grid. If someone wants a 4-5x performance, flash storage (SSD, Flash on PCI-E) can do the job easily. They are cheap and can provide a modest performance boost. However, if we have to achieve more than 10-20x performance, then we need to find another paradigm or solution: in-memory computing.

In plain English, `in-memory computing` primarily relies on keeping data in a server's RAM as a means of processing at faster speeds.

Why is it so popular? Because memory prices have come down a lot in recent times. In-memory computing can now be used to speed data-intensive processing. Imagine if you want to provide 16000 RPS for your customer internet banking portal or millions of transactions per second for online transaction processing system such as OW4. This is a common use case for in-memory computing that would be very hard to achieve with traditional disk-based computing. Best use cases for in-memory computing are as follows:

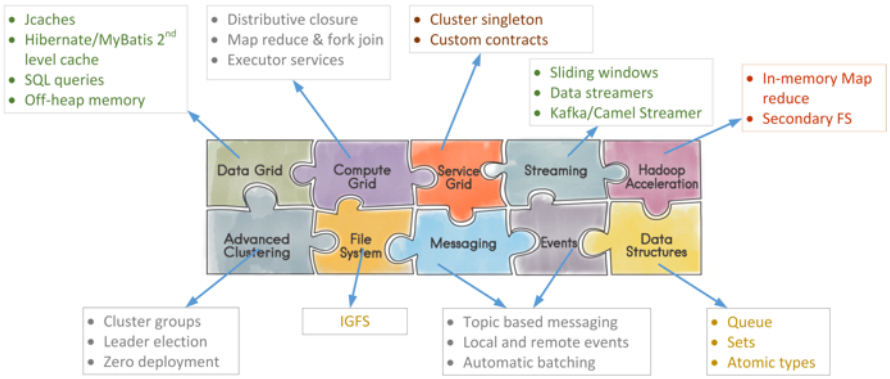
- High volume of ACID transactions processing.
- Cache as a Service (CaaS).
- Database caching.
- Complex event processing for IoT projects.
- Real-time analytics.
- HTAP business applications.

What is Apache Ignite?

Apache Ignite provides a convenient and easy-to-use interface for developers to work with large-scale data sets in real time and other aspects of in-memory computing. Apache Ignite has the following features:

1. Data grid.
2. Compute grid.
3. Service grid.
4. Bigdata accelerator;
5. and Streaming grid.

The following figure illustrates the basic features of Apache Ignite.



Ignite features

These are the core Apache Ignite technologies:

1. Open source.
2. Written in pure Java.
3. Supports java 7 and 8.
4. Based on Spring.
5. Supports .Net, C++ and PHP.

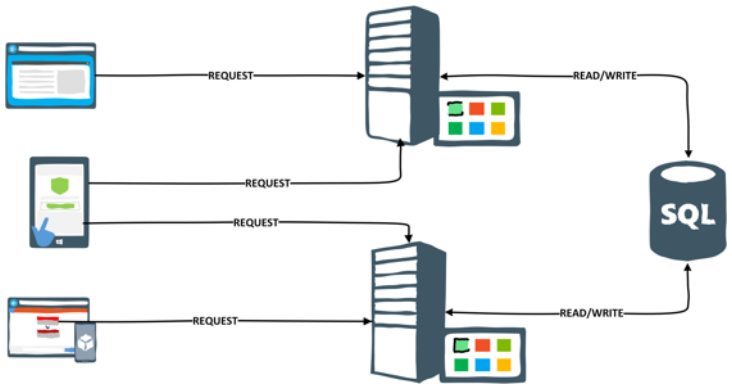
The primary capabilities that Apache Ignite provides are as follows

- Elasticity: An Apache Ignite cluster can grow horizontally by adding new nodes.
- Persistence: Apache Ignite data grid can persist cache entries in RDBMS, even in NoSQL like MongoDB or Cassandra.

- 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 base databases.
- 2nd Level Cache: Apache Ignite is the perfect caching tier to use as a 2nd level cache in Hibernate and MyBatis.
- High-performance Hadoop accelerator: Apache Ignite can replace Hadoop task tracker and job tracker and HDFS to increase the performance of big data analysis.
- Share state in-memory across Spark applications: Ignite RDD allows easily sharing of state in-memory between different Spark jobs or applications. With Ignite in-memory shared RDD's, any Spark application can put data into Ignite cache which will be accessible by another Spark application later.
- 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 continuous never-ending streams of data in scalable and fault-tolerant fashion in-memory, rather than analyzing the data after it has been stored in the database.

Modern application architecture with Apache Ignite

Let's take a quick look at an architecture of a traditional system. The traditional application architecture uses data stores which have synchronous read-write operations. This is useful for data consistency and data durability, but it is very easy to have a bottleneck if there are a lot of transactions waiting in the queue. Consider the following traditional architecture as shown below.

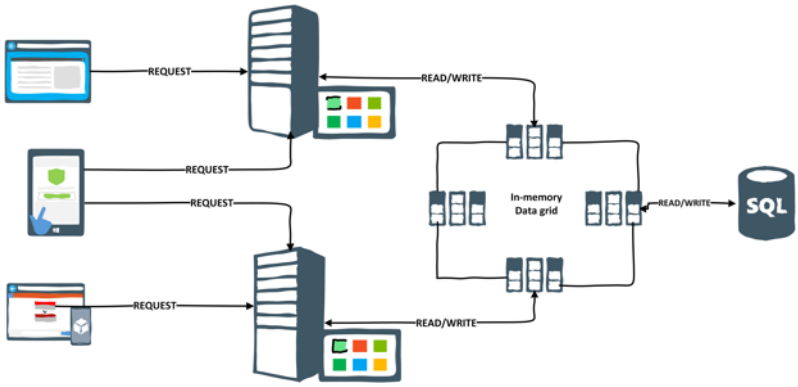


The traditional application architecture

High-volume transaction processing.

In-memory data grid adds an additional layer within an environment, which uses the Random-Access Memory (RAM) of the server to store most of all data required by the applications. In-memory

data grid sits between the application servers and the data store. 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 whenever needed and even asynchronously send and receive updates from the persistence store. An application architecture with in-memory data grid is shown below.

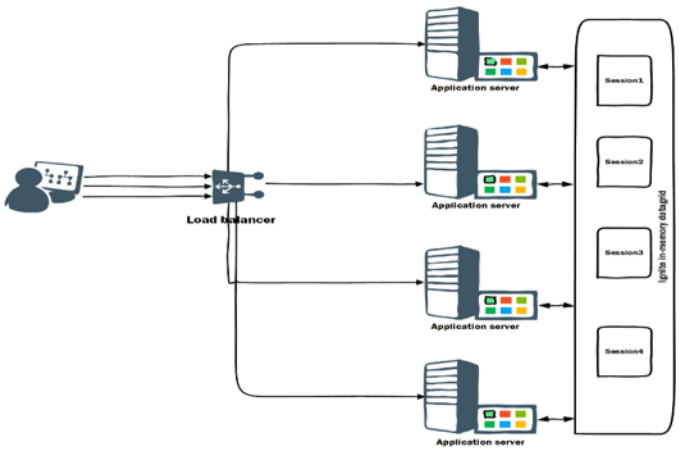


Application architecture with in-memory data grid

By using in-memory data grid, data moves closer to the application endpoints. This approach reduces the response times and can lower transactions times from a few seconds to fractions of a second. This way, the application can support extremely large numbers of concurrent transactions involving terabytes of operational data, providing a faster, more reliable transactional experience for customers. It is also a more modern scalable data management system than traditional RDBMS, able to elastically scale as demand increases.

Resilient web acceleration.

With in-memory data grid like Apache Ignite, you can provide fault tolerance to your web application and accelerate your web application's performance. Without changing any code, you can share session states between web applications through caches.



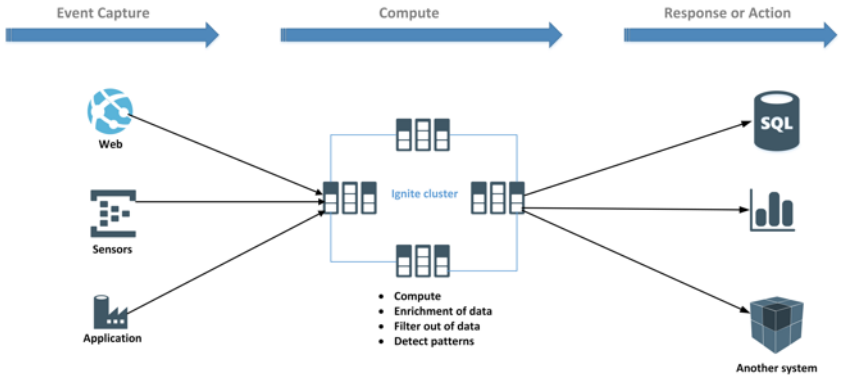
Fault tolerance web application

This above approach provides the highest level of high availability of a system and a customer

experience. Since Ignite is an in-memory solution, the performance of the web session clustering and replication mechanism of user web sessions are very high.

Event processing & real-time analysis.

Data tells the story of what's happing with your business on the background right now. With the IoT as a continuous data source, the opportunities to take advantage of the hot data is greater than ever. Traditional data management system cannot process big data fast enough to notify the business of important events as they occur: such as online credit card fraud detection or risk calculation. Apache Ignite allows processing continuous never-ending streams of data in scalable and fault-tolerant fashion in-memory, rather than analyzing data after it's reached the database.

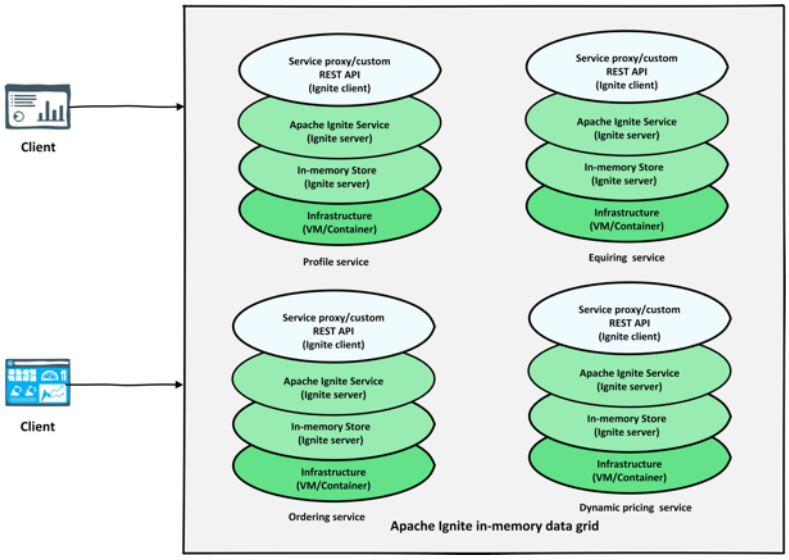


Complex event notification and processing

Not only does this enable you to correlate relationships and detect meaningful patterns from significantly more data but you can process 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 large amount of incoming never ending data and take action.

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 code base. In-memory data grid like Apache Ignite can provide independent cache nodes to corresponding microservices in the same distributed cluster and gives you a few advantages over traditional approaches.

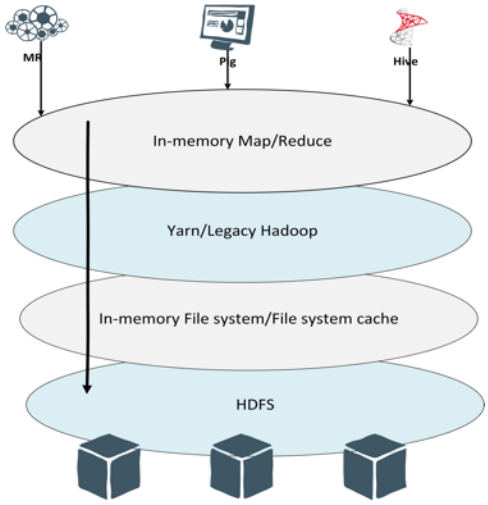


Apache Ignite microservice architecture

It allows you to make use to its maximum the data fabrics/grid resources. 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 distributed service instance in the cluster.

BigData accelerator.

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, it's 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 data within the memory. Placing Map/Reduce in- memory with the data it needs eliminates file I/O latency.

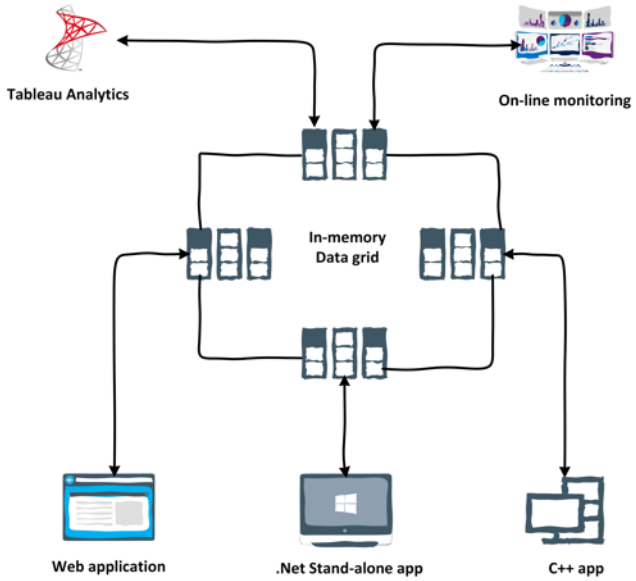


In-memory Map/Reduce

Apache Ignite has offered a set of useful components allowing in-memory Hadoop job executing and file system operations. Apache Ignite Hadoop accelerator can automatically deploy all necessary executable programs and libraries for the execution of Map/Reduce across the JVMs, which greatly reducing startup time down to milliseconds. This speeds up by avoiding delays in accessing secondary storage. Also, because the execution engine is integrated with the in-memory data grid, key/value pairs hosted within the data grid can be efficiently read into the execution engine to minimize access time.

Cache as a Service.

Data-driven applications that take too long to load are boring and frustrating to use. Four out of five online users will click away if a page stalls while loading. In-memory data grid can provide a common caching layer across the organization, which can allow multiple applications to access managed in-memory cache.



Cache as a Service

You can isolate the caching layer from the applications by separating the caching layer from the applications. Any applications (Java, .Net, C++) across the organization can store and read data from the cache. By using in-memory data grid as a service, it's not necessary to build and deploy local caching infrastructure for each application. 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.

These are some of the ways in-memory grids like Apache Ignite have served as an essential, architectural component for **transforming** the way businesses use their data to do business. But that's not all folks! We will cover a lot of in-memory data grid use cases and application architecture in more details through this book.

Who uses Apache Ignite?

Apache Ignite is widely used around the world and is growing all the time. Companies like Barclays, Misys, Sberbank (3rd largest bank in the Europe) all use Ignite to power pieces of their architecture that are critical to the day-to-day operations of those organizations.

Why Ignite instead of others?

There are a few others alternatives of Apache Ignite from other vendors such as HazelCast, Oracle, Ehcache, GemFire, etc. The main difference of Apache Ignite from the others is the quantity of functionality and simplicity of use. Apache Ignite provides a variety of functionalities, which you can use for different use cases. Unlike other competitors, Ignite provides a set of components called

Hadoop accelerator and Spark shared RDD that can deliver real-time performance to Hadoop & Spark users.

Our Hope

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

The book is a *project-based* guide, where each chapter focuses on the complete implementation of a real-world scenario. The frequent challenges in each scenario will be discussed, along with tips, tricks and best practices on how to overcome them. For every topic, we will introduce a complete sample running application, which will help you to hit the ground running.

Chapter two: Architecture overview

To better understand the functionality of Apache Ignite and use cases, it's very important to understand its architecture and topology. By getting a better understanding of Ignite's architecture, you can decide the topology or cache mode to solve different problems in your enterprise architecture landscape and get the maximum benefits from in-memory computing. Unlike master-slave designs, Ignite makes use of an entirely peer-to-peer architecture. Every node in the Ignite cluster can accept read and write, no matter where the data is being written. In this chapter we are going to cover the following topics:

- Apache Ignite functional overview.
- Different cluster topology.
- Cluster topology.
- Caching strategy.
- Clustering.
- Data model.
- Multi-datacenter replication.
- Asynchronous support.
- How SQL queries works in Ignite.
- Resilience.

Functional overview

The 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, in-memory components geared to improve the performance and scalability of your application. The following schematic represents the basic functionalities of Apache Ignite.

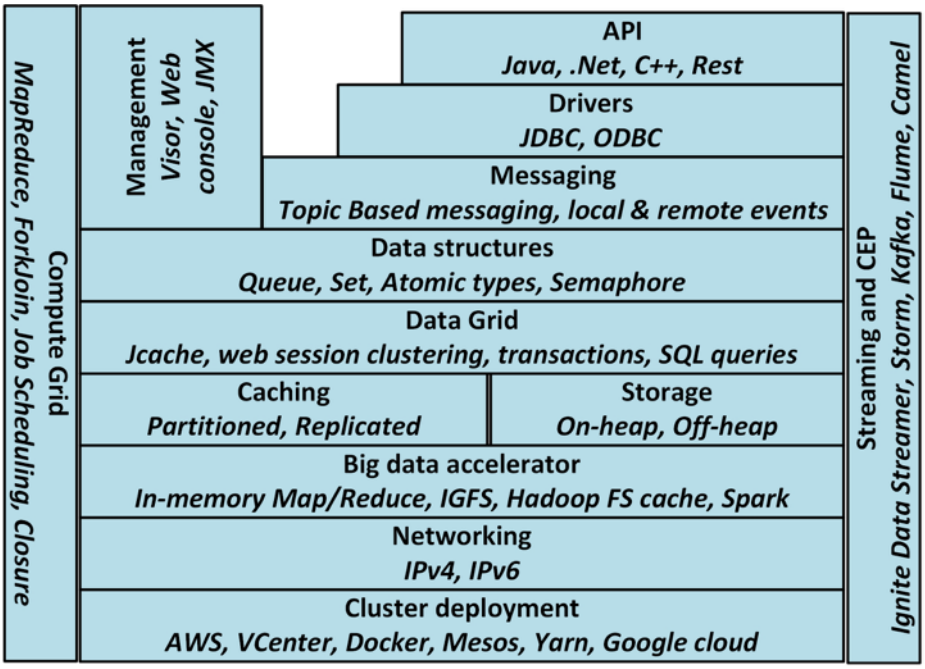


Figure 2.1. Functional architecture

Note that Apache Ignite contains a lot of features not shown in the above figure due to lack of space. Ignite is organized in a modular fashion and provides a single jar (library) for each functionality. You only have to apply the desired library into your project to use Ignite.

Cluster Topology

Ignite design implies that the entire system itself is both inherently available and massively scalable. Ignite internode communication allows all nodes to receive updates without the need for a **master coordinator** quickly. Nodes can be added or removed non-disruptively to increase the amount of *RAM* available. Ignite data fabrics are fully resilient, allowing non-disruptive automated detection and recovery of a single server or multiple servers.

> **Note:**
>
> In contrast to the monolithic and master-slave architectures, there are no special nodes in Ignite. All nodes are identical in the Ignite cluster.

Client and Server

However, Apache Ignite has an optional notion of servers and provides two types of nodes: Client and Server nodes.

Node	Description
Server	Contains Data, participates in caching, computations, streaming and can be part of the in-memory Map-Reduce tasks.
Client	Provides the ability to connect to the servers remotely to put/get elements into the cache. It can also store portions of data (near cache), which is a smaller local cache that stores most recently and most frequently accessed data.

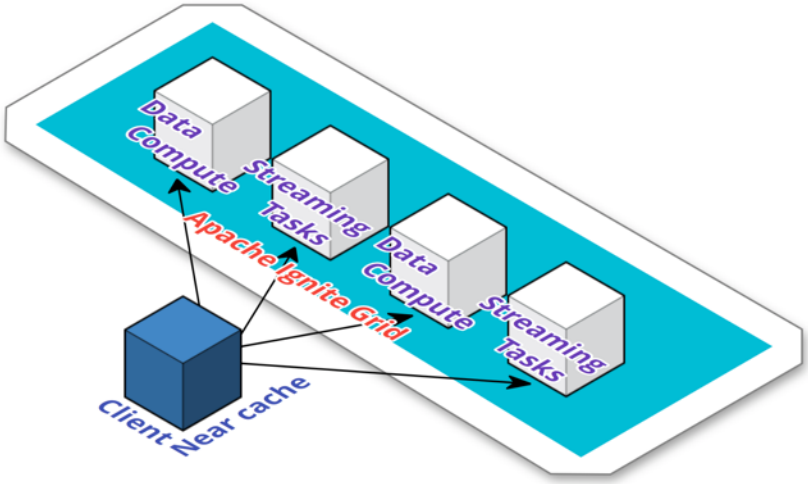


Figure 2.2. Ignite client and servers

The server node also can be grouped together in a cluster to perform work. Within a cluster, you can limit job execution, service deployment, streaming and other tasks to run only within 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 **testCache**. We shall discuss the clustering in more detail in the subsequent section of this chapter. By default, all nodes are started as server nodes. Client mode needs to be explicitly enabled. Note that you can't physically separate data nodes from compute nodes. In Apache Ignite, servers that contain data, are also used to execute computations.

Apache Ignite client nodes also participate in job executions. The concept might seem complicated at first glance, but let's try to clarify the concept.

Server nodes always store data and **by default** can participate in any computation task. On the other hand, the *Client* node can manipulate the server caches, store *local* data and participate in **computation tasks**. Usually, client nodes are used to put or retrieve data from the caches. This type of hybrid client nodes gives flexibility when developing a massive Ignite grid with many nodes. Both clients and server nodes are located in one grid, in some cases (as for example, high volume acid transactions in data node) you just do not want to execute any computation on a data node. In this case, you can choose to execute jobs only on client nodes by creating a corresponding cluster group. This way, you can separate the data node from the compute node in one grid. Compute on clients can be performed with the following pseudo code.

```
ClusterGroup clientGroup = ignite.cluster().forClients();
IgniteCompute clientCompute = ignite.compute(clientGroup);
// Execute computation on the client nodes.
clientCompute.broadcast(() -> System.out.println("sum of: " + (2+2)));
```

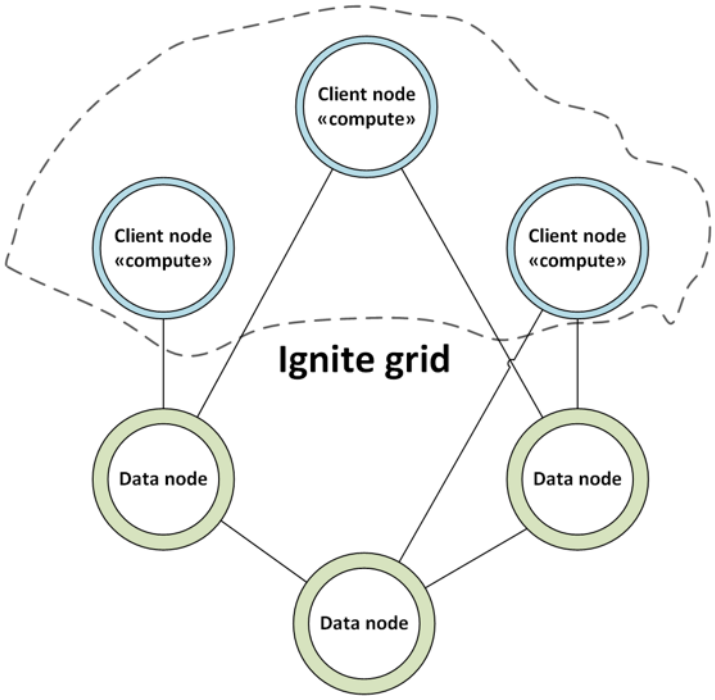


Figure 2.3. Client cluster group

There is one downside of this approach. With this approach, data will be allocated to separate nodes and for the computation of this data, all client nodes will need to retrieve the data from the server nodes. It can produce a lot of network connections and create latency. However, you can always run client and server nodes in separate JVM on one single host to decrease the network latency. We will discuss the different deployment approaches in more detail later in this chapter.

From the deployment point of view, Apache Ignite servers can be divided into the following groups.

- Embedded with application.
- Server in separate JVM.

Embedded with the application

With this approach, Apache Ignite node runs on the same JVM with the application. It can be any web application running on an application server or with standalone Java application. For example, our standalone **HelloIgnite** application from the chapter one – is an embedded Ignite server. Ignite server run along with the application in the same JVM and joins with other nodes of the grid. If

the application dies or is taken down, Ignite server will also shut down. This topology approach is shown in the following diagram:

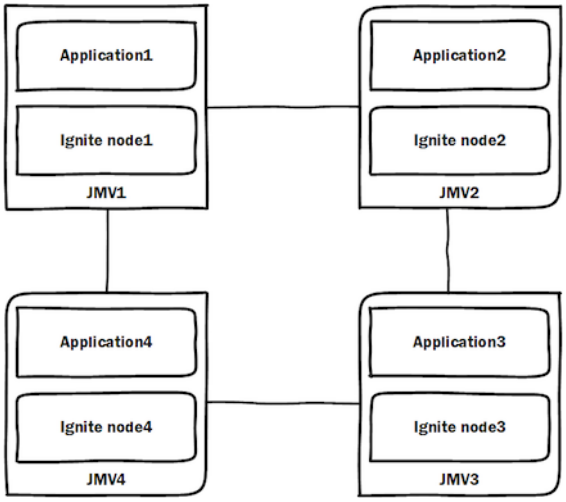


Figure 2.4. Embedded with the application

If you run the HelloIgnite application again and examine the logs of the server, you should find the following:

```
[18:36:05] Ignite node started OK (id=3113ed1e)
[18:36:05] Topology snapshot [ver=8, servers=1, clients=0, CPUs=8, heap=1.0GB]
[21:26:12] Topology snapshot [ver=9, servers=2, clients=0, CPUs=8, heap=4.5GB]
[21:26:13] Topology snapshot [ver=10, servers=1, clients=0, CPUs=8, heap=1.0GB]
```

Figure 2.5. Ignite log

HelloIgnite application run and joins to the cluster as a server. After completing the task, the application exits with the Ignite server from the Ignite grid.

Server in separate JVM (real cluster topology)

In this approach, server nodes will run in a separate JVM and client nodes remotely connects to the servers. Server nodes participate in caching, compute executions, streaming and much more. The client can also use REST API to connect to any individual node. By default, all Ignite nodes are started as server nodes; client nodes need to be explicitly enabled.

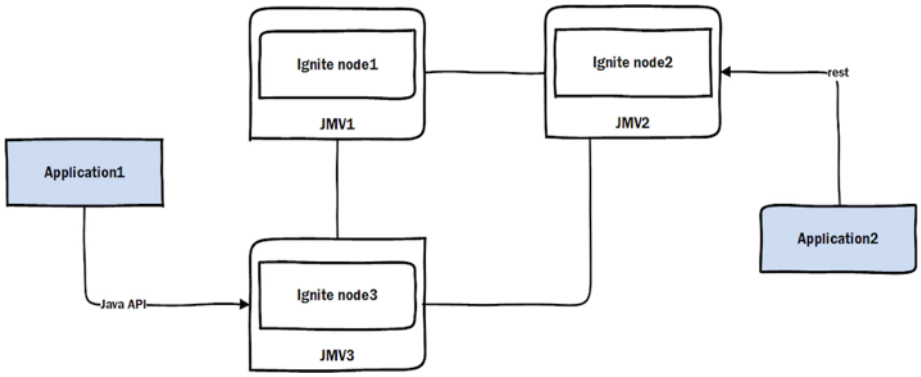


Figure 2.6. Real cluster topology

This is the most common approach, as it provides greater flexibility in terms of cluster mechanics. Ignite servers can be taken down and restarted without any impact to the overall application or cluster.

Client and Server in separate JVM on single host

You can consider this approach whenever you have a high volume of transactions on your data nodes and planning to perform some computations on this node. You can execute client and server in a separate JVM within a container such as Docker or OpenVZ. Containers can be located in the single host machine. The container will isolate the resources (cpu, ram, network interface, etc.) and the JVM will only use isolated resources assigned to this container.

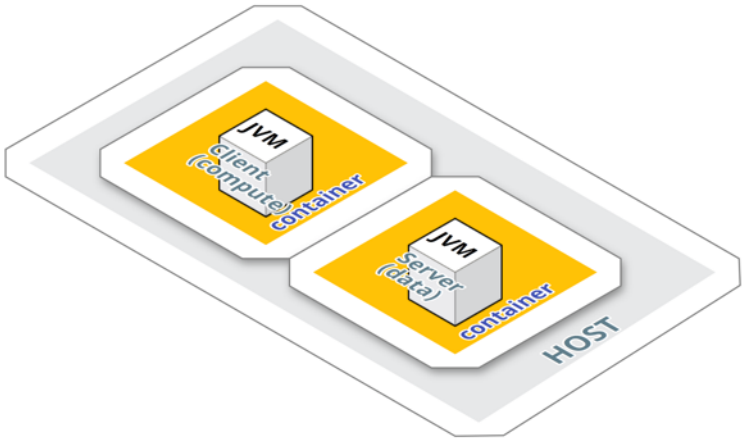


Figure 2.7. Client and server in separate JVM

This approach also has its own downside. During execution, the client (compute) node can retrieve data from any other data node that reside on other hosts and it can increase the network latency.

Caching Topology

Ignite provides three different approaches to caching topology: *Partitioned*, *Replicated* and *Local*. A *cache mode* is configured for each cache individually. Every caching topology has its own goal with pros and cons. The default cache topology is *partitioned*, without any backup option.

Partitioned caching topology

The goal of this topology is to get extreme *scalability*. In this mode, the Ignite cluster transparently **partitions** the cached data to distribute the load across an entire cluster evenly. By partitioning the data evenly, the size of the cache and the processing power grows linearly with the size of the cluster. 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.

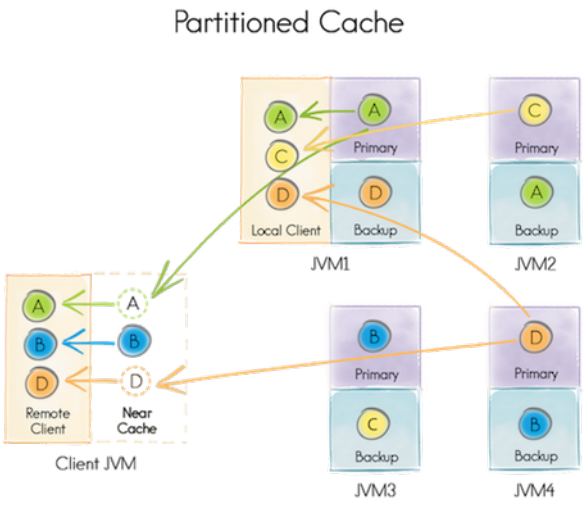


Figure 2.8. Partitioned caching topology

With partitioned cache topology, DML operations on the cache are extremely fast, because only one primary node (optionally 1 or more backup node) needs to be updated for every key. For high availability, a backup copy of the cache entry should be configured. The backup copy is the redundant copy of one or more primary copies, which will live in another node. There is a simple formula to calculate, how many backups copies you need for the high availability of your cluster.

Number of backup copies = N-1, where N is the total number of the nodes in the cluster.

Assume, you have a total number of 3 nodes in the cluster. If you always want to get a response from your cluster (when some of your nodes are unavailable), the number of backup copies should be not less than 2. In this case, 3 copies of the cache entry exist, 2 backup copies and 1 primary. Partitioned caches are ideal when working with large datasets and updates are very frequent. The

backup process can be synchronous or asynchronous. In synchronous mode, the client should wait for the responses from the remote nodes, before completing the commit or write.

Replicated caching topology

The goal of this approach is to get extreme **performance**. With this approach, cache data is *replicated* to all members of the cluster. Since the data is replicated to each cluster node, it is available for use without any waiting. This provides highest possible speed for read-access; each member accesses the data from its own memory. 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.

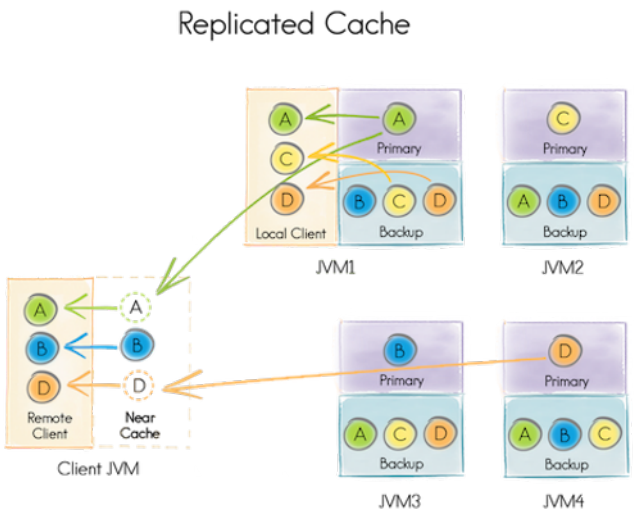


Figure 2.9. Replicated caching topology

In the above diagram, the same data is stored in all cluster nodes; the size of a replicated cache is limited by the amount of memory available on each node with the smallest amount of RAM. This mode is ideal for scenarios where cache reads are a lot more frequent than cache writes, and the data sets are small. The scalability of replication is inversely proportional to the number of members, the frequency of updates per member, and the size of the updates.

Local mode

This is a very primitive version of cache mode; with this approach, no data is distributed to other nodes in the cluster. As far as the Local cache 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 is mostly used in read-only operations. It also works very well for read/write-through behavior, where data is loaded from the data sources on cache misses. Unlike a

distributed cache, local cache still has all the features of distributed cache; it provides query caching, automatic data eviction and much more.

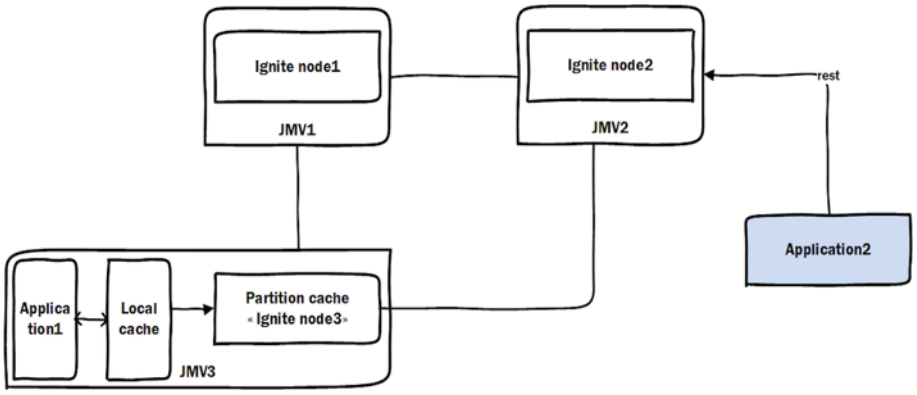


Figure 2.10. Local mode

Caching strategy

With the explosion of high transactions web applications and mobile apps, data storage has become the main bottleneck of performance. In most cases, persistence stores such as 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. It extends multiple servers (called a grid) to pool their memory together and keep the cache synchronized across all servers. There are two main strategies to use in a distributed in-memory cache:

Cache-aside

In this approach, an application is responsible for reading and writing from the persistence store. The cache doesn't interact with the database at all. This is called *cache-aside*. The cache behaves as a fast scaling in-memory data store. The application checks the cache for data before querying the data store. Also, the application updates the cache after making any changes to the persistence store.

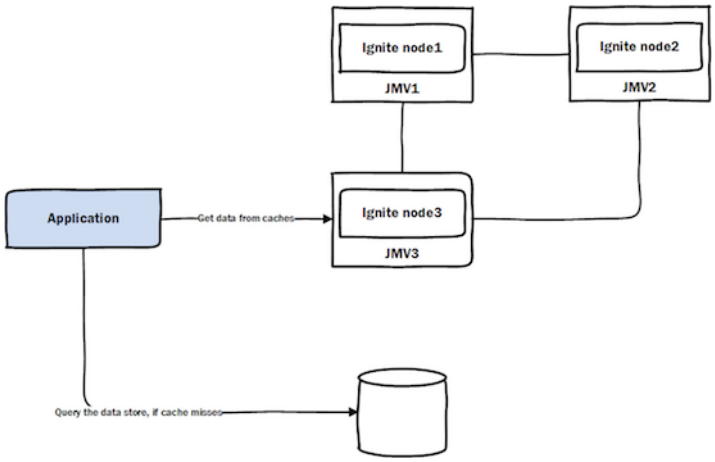


Figure 2.11. Cache-aside

However, even though cache-aside is very fast, there are quite a few disadvantages with this strategy. Application code can become complex and may lead to code duplication if multiple applications deal with the same data store. When there are cache data misses, the application will query the data store, update the caches and continue processing. This can result in multiple data store visits if different application threads perform this processing at the same time.

Read-through and Write-through

This is where application treats in-memory cache as the main data store, and reads data from it and writes data to it. In-memory cache is responsible for propagating the query to the data store on cache misses. Also, the data will be updated automatically whenever it is updated in the cache. All read-through and write-through operations will participate in the overall cache transaction and will be committed or rolled back as a whole.

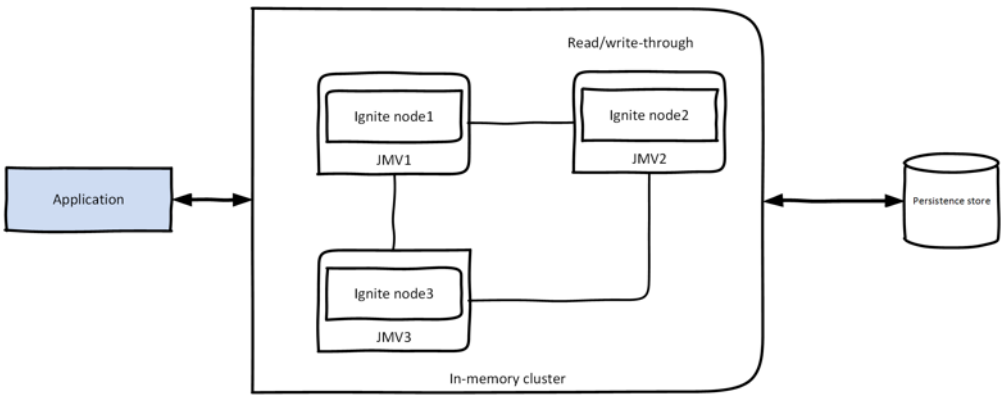


Figure 2.12. Read/Write-through

Read-through and write-through have numerous advantages over cache-aside. First of all, it simplifies application code. Read-through allows the cache to reload objects from the database when

it expires automatically. This means that your application does not have to hit the database in peak hours because the latest data is always in the cache.

Write behind

It is also possible to use write-behind to get better write performance. Write-behind lets your application quickly update the cache and return. It then aggregates the updates and asynchronously flushes them to persistence store as a bulk operation. Also with Write-behind, you can specify throttling limits, so the database writes are not performed as fast as the cache updates and therefore the pressure on the database is lower. Additionally, you can schedule the database writes to occur during off-peak hours, which can minimize the pressure on the Database.

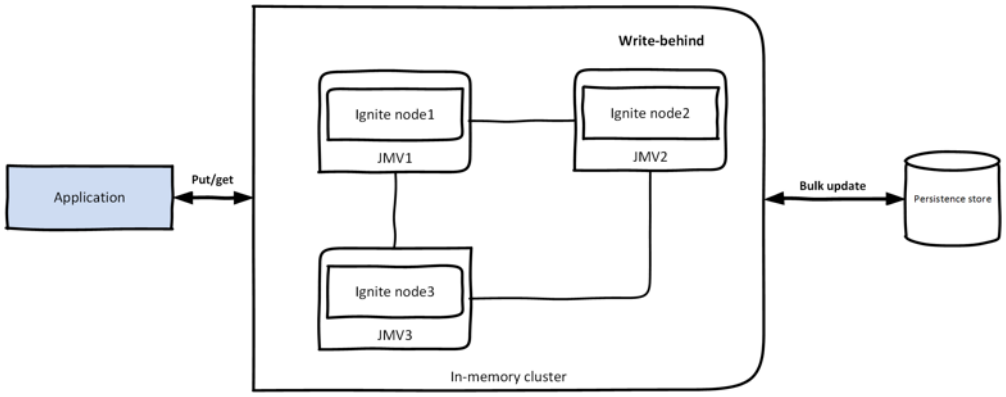


Figure 2.13. Write behind

Apache Ignite provides all the above caching strategies by implementing the Java JCache specification. In addition, Ignite provides Ignite *Cassandra* module, which implements persistent store for Ignite caches by utilizing Cassandra as a persistent storage for expired cache entries.

Data model

Apache Ignite implements a Key-Value data model, especially **JCache (JSR 107)** specification. JCache provides a common way for Java application to interact with the Cache. Terracotta has played the leading role in JSR107 development, acting as a specification lead. The JCache final specification was released on 18th March 2014. 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 basic storage. With the JSR107 specification, at last developers can program to a standard API instead of being tied to a single vendor.

From a design point of view, JCache provides a very simple key-value store. A key-value store is a simple Hashtable or Map, primarily used when you access the database table via a primary key. You

can take a key-value as a simple table in a traditional RDBMS with two columns such as 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). The application can provide a Key and Value and persist the pair. If the Key already exists, the value will be overwritten, otherwise, a new Value will be created. For clarity, we can compare the key-value store with Oracle terminology.

Oracle	**Apache Ignite**
Database Instance	Apache Ignite server
Table	Cache
Row	Key-value
RowID	Key

Key-value stores are the simplest data store in NoSQL world. It has very primitive operations like *put*, *get* or *delete* the value from the store. Since it always uses primary key access, they generally have a great performance and scalability.

Since 2014, JCache supports the following platforms:

JCache Delivery	**Target Platform**
Specification	Java 6+ (SE or EE)
Reference Implementation	Java 7+ (SE or EE)
Technology Compatibility kit	Java 7+ (SE or EE)
Demos and Samples	Java 7+ (SE or EE), Java 8+ (SE or EE)

Currently most caching projects and vendors implement the JCache specification as follows:

- Oracle Coherence
- Hazelcast
- Terracotta Ehcache
- Apache Ignite
- Infinispan

The Java caching API defines five core interfaces: CachingProvider, CacheManager, Cache, Entry and Expiry.

- A CachingProvider defines the mechanism to establish, configure, acquire, manage and control zero or more CacheManagers.
- A CacheManager defines the mechanism to establish, configure, acquire, manage and control zero or more uniquely named Caches all within the context of a CacheManager
- A Cache is a hash table like data structure that allows the temporary storage of key-based values. A Cache is owned by a single CacheManager.
- An entry is a single key-value pair stored in a Cache.

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

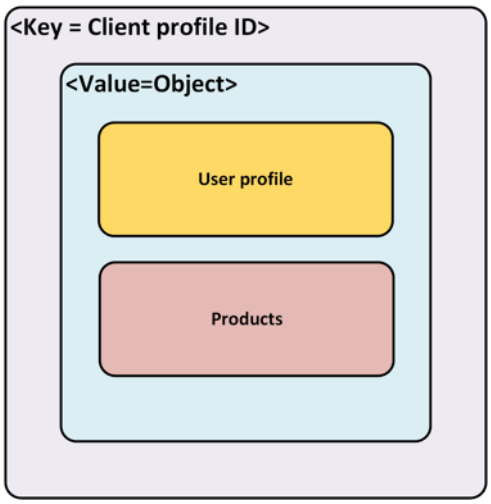


Figure 2.14. Key-value entry

With the key-value store, Java caching API also provides the following additional features:

- Basic Cache operations
- atomic operations, similar to java.util.ConcurrentMap
- read-through caching
- write-through caching
- Entry processor
- cache event listeners
- statistics
- caching annotations
- full generics API for compile time safety
- storage by reference (applicable to on heap caches only) and storage by value

In addition to JCache, Apache Ignite provides ACID transaction, SQL query capability, data loading, Asynchronous mode and various memory models. Apache Ignite provides the **IgniteCache** interface, which extends the Java Cache interface for working with the Cache. In the previous chapter, we already saw some basic operations of IgniteCache. Here is the pseudo code of the HelloWorld application from the previous chapter.

```
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 provides the JCache entry processor functionality for eliminating the network round trips across the network when doing puts and updates in the cache. JCache *EntryProcessor* allows for processing data directly on primary nodes, often transferring only the **deltas** instead of the full state.

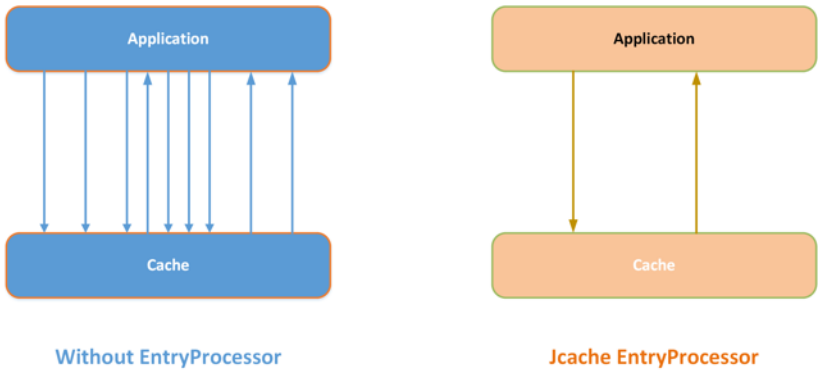


Figure 2.15. EntryProcessor

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

```
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;
    }
  });
```

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 in any NoSQL data store. 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[6] and means availability and partition tolerance, which are generally considered more important in NoSQL systems than consistency.

In 2000, Eric Brewer[7] 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. Each NoSQL data store can be classified by the CAP theorem as follows.

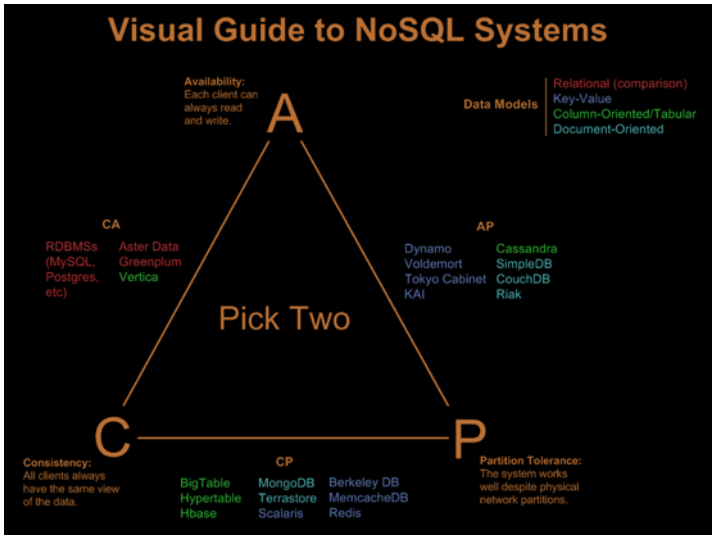


Figure 2.16. CAP theorem

The above illustration is taken from the blog post[8] of the Nathan Hurst. As you can see, a distributed system can only have two of the following three properties:

- Partition tolerance: meaning, if you chop the cable between two nodes, the system still works.
- Consistency: each node in the cluster has the same data.
- Availability: a node will always answer the queries if possible.

[6] https://en.wikipedia.org/wiki/CAP_theorem
[7] https://en.wikipedia.org/wiki/Eric_Brewer_(scientist)
[8] http://blog.nahurst.com/visual-guide-to-nosql-systems

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: the opposite of the CA system. In CP system availability is sacrificed for consistency and partition-tolerance. In the event of the node failure, some data will be lost.

AP system: This system is always available and partitioned. Also this system scales easily by adding nodes to the cluster. Cassandra is a good example of this type of system.

Now, we can return back to our question, where does Ignite stand in the CAP theorem? At first glance, Ignite can be classified by CP, because Ignite is fully ACID compliant distributed transactions with partitioned tolerance. But this is half part of the history. Apache Ignite can also be considered an AP system. But why does Ignite have two different classifications? Because it has two different transactional modes for cache operations, transactional and atomic.

In transactional mode, you can group multiple DML operations in one transaction and make a commit into the cache. In this scenario, Ignite will lock data on access by a pessimistic lock. If you configure backup copy for the cache, Ignite will use 2p commit protocol for its transaction. We will look at transactions in detail in chapter four.

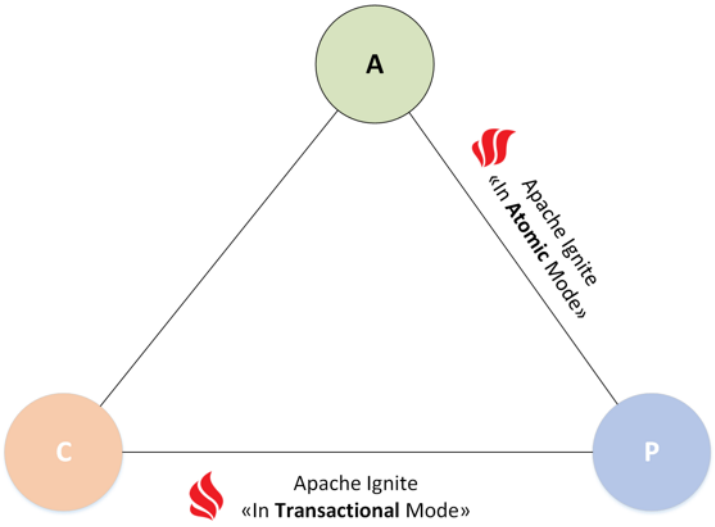


Figure 2.17. Ignite position in CAP theorem

On the other hand, in atomic mode Ignite supports multiple atomic operations, one at a time. In the 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 a higher performance than the transactional mode. When you make a write in Ignite cache, for every piece of data there will be a master copy in primary node and a backup copy (if defined). When you read data from Ignite grid, you always read from the Primary node, unless the Primary node is down, at which time data will be read from the backup. From this point of view, you gain the system availability and the partition-tolerance of the entire

system as an AP system. In the atomic mode, Ignite is very similar to Apache Cassandra.

However, 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.

Clustering

The design goal of the Apache Ignite is to handle high workloads across multiple nodes within a cluster. A cluster is arranged as a group of nodes. Clients can send read/write requests to any node in the cluster. Ignite nodes can automatically discover each other and data is distributed among all nodes in a cluster. This helps scale the cluster when needed, without restarting the entire cluster at a time. Ignite provides an easy way to create logical groups of cluster nodes within your grid and also collocate the related data into similar nodes to improve performance and scalability of your application. In the next few subsections, we will discover a few very important and yet unique features of Apache Ignite such as cluster group, data allocation (sometimes calls *affinity collocation*) and ability to scale and zero single points of failure.

Cluster group

Ignite `ClusterGroup` provides a simple way to create a logical group of nodes within a cluster. By design, all nodes in an Ignite cluster are the same. However, Ignite allows to logically group nodes of any application for a specific purpose. For instance, you can cluster all nodes together that service 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. You can limit job executions, service deployment, messaging, events, and other tasks to run only within some cluster groups.

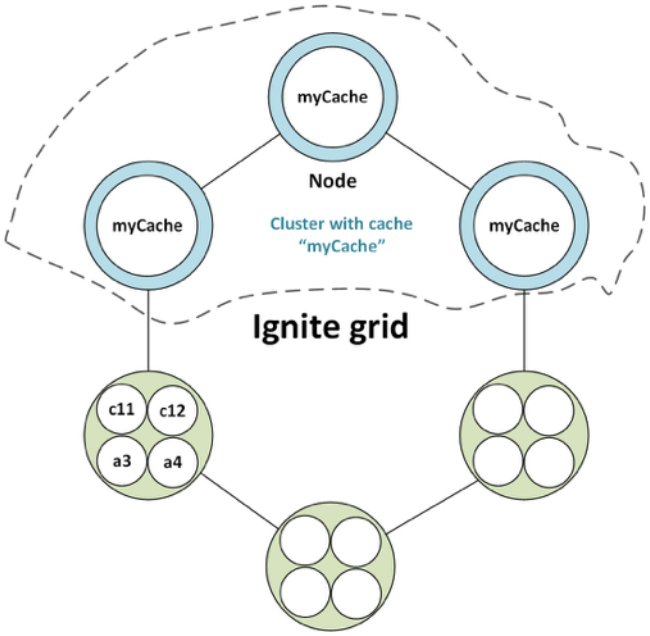


Figure 2.18. Cluster group

Ignite provides the following three ways to create a logical cluster into Ignite Grid:

Predefined cluster group. Ignite provides predefined implementations of ClusterGroup of an interface to create cluster group based on any predicate. Predicates can be Remote Node, Cache Nodes, Node with specified attributes and so on. Here is an example cluster group with all nodes caching data for cache *myCache*.

```
IgniteCluster cluster = ignite.cluster();
// All the data nodes responsible for caching data for "myCache".
ClusterGroup dataGroup = cluster.forDataNodes("myCache");
```

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:

```
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:

```
IgniteCluster cluster = ignite.cluster();
ClusterGroup workerGroup = cluster.forAttribute("ROLE", "master");
Collection<GridNode> workerNodes = workerGroup.nodes();
```

Custom cluster group. Sometimes it also calls 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. Note that the nodes in this group will change over time based on their heap memory used.

```
IgniteCluster cluster = ignite.cluster();
// Nodes with less than 256MB heap memory used
ClusterGroup readyNodes = cluster.forPredicate((node) -> node.metrics().getHeapMemoryUsed(\
) < 256);
```

Data collocation

Term *data collocation* means, allocation of same 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 node. In this approach, network roundtrips for the related data decrease and the client 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.

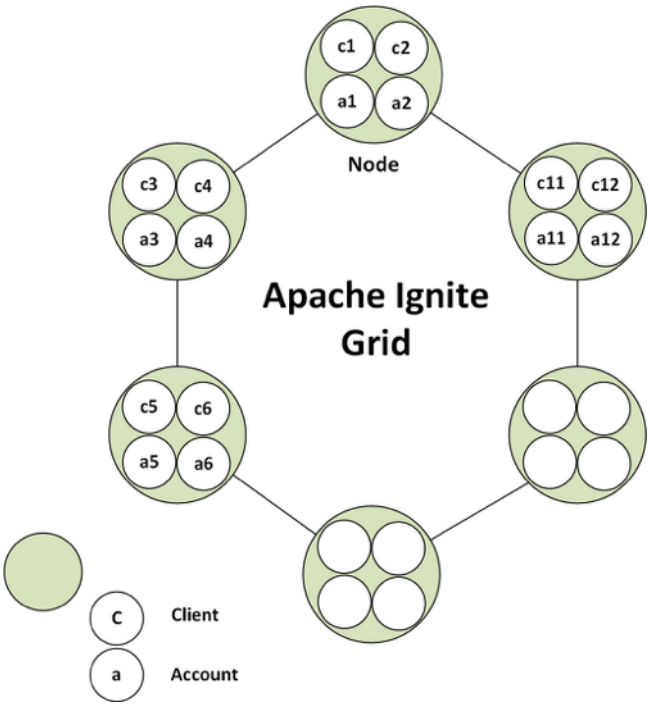


Figure 2.19. Data collocation

For example, 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 for collocation. For convenience, you can also optionally use *AffinityKey* class as follows:

```
Object clientKey1 = new AffinityKey("Client1", "accId");
Object clientKey2 = new AffinityKey("Client2", "accId ");

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("accId ", new Account(“credit card”));
cache.put(clientKey1, c1);
cache.put(clientKey2, c2);
```

To calculate the affinity function, you can use any set of fields, it is not necessary to use any kind of unique key. For example, to calculate the Client account affinity function, you can use the field *client ID* that owns the account id. 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 whole units of work to a certain node. To collocate computation with data you should use `IgniteCompute.affinityRun(...)` and `IgniteCompute.affinityCall(...)` methods.

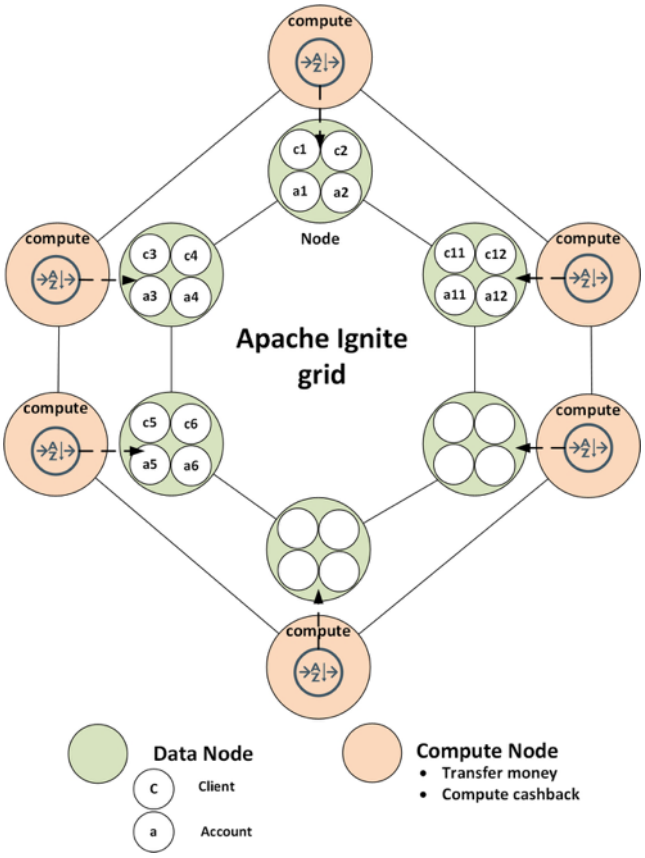


Figure 2.20. Compute collocation with Data

Here is how you can collocate your computation in the same cluster node on which the Client and its account information are allocated.

```
String accId = "acountId";
// Execute Runnable on the node where the key is cached.
ignite.compute().affinityRun("myCache", accId, () -> {
Account account = cache.get(accId);
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.

Apache Ignite out of the box ships with two affinity function implementations:

RendezvousAffinityFunction[9] - This function allows a bit of discrepancy in partition-to-node mapping (i.e. some nodes may be responsible for a slightly larger number of partitions than others).

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

However, it guarantees that when topology changes, partitions are migrated only to a joined node or only from a left node. No data exchange will happen between existing nodes in a cluster. This is the default affinity function used by the Apache Ignite.

FairAffinityFunction - This function tries to make sure that partition distribution among cluster nodes is even. This comes at a price of a possible partition migration between existing nodes in a cluster.

Later in this book, we will provide a complete real life example with the explanation in chapter seven.

Zero SPOF

In any distributed system, node failure should be expected, particularly as the size of the cluster grows. The Zero Single Point of Failure (SPOF) design pattern ensures that no single part of a system can stop the entire cluster or system from working. Sometimes, the system using master-slave replication or the mixed master-master system falls into this category. Prior to Hadoop 2.0.0, the Hadoop NameNode was an SPOF in an HDFS cluster. Netflix has calculated the revenue loss for each ms of downtime or latency, and it is not small at all. Most businesses do not want single points of failure for the obvious reason.

Apache Ignite, as a horizontally scalable distributed system, is designed in such way that all nodes in the cluster are equal, you can read and write from any node in the cluster. There are no master-slave communications in the Ignite cluster.

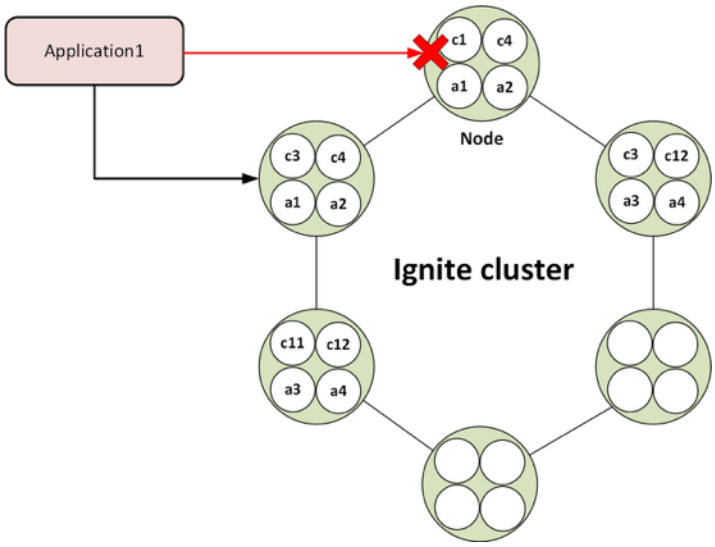


Figure 2.21. Zero SPOF

Data is backed up or replicated across the cluster so that failure of any node doesn't bring down the entire cluster or the application. This way Ignite provides a dynamic form of **High Availability**. Another benefit of this approach is the ease at which new nodes can be added. When new nodes

join the cluster, they can take over a portion of data from the existing nodes. Because all nodes are the same, this communication can happen seamlessly in a running cluster.

How SQL queries works in Ignite

In chapter one, we introduced Ignite SQL query feature very superficially. In chapter four, we will go into more details about Ignite SQL queries. It's interesting to know how a query processes under the hood of Ignite. There are two main approaches to process SQL queries in Ignite:

- In-memory Map-Reduce: If you are executing any SQL query against a Partitioned cache, Ignite under the hood splits the query into 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 in the cluster. Then all map queries are executed on all data nodes of the participating caches, providing 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.
- H2 SQL engine: if you are executing SQL queries against Replicated or Local cache, Ignite knows that all data is available locally and runs a simple local SQL query in the H2 database engine. Note that, in replicated caches, every node contains a replica data for other nodes. H2 database is a free database written in Java and can work in an embedded mode. Depending on the configuration, every Ignite node can have an embedded H2 SQL engine.

Multi-datacenter replication

In the modern world, multi-datacenter replication is one of the main 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 in order to provide a responsive exchange. In this case, data centers can be located in different geographical locations such as in different regions or in different countries. Data synchronization is completely transparent and bi-directional within data centers. The logic that defines which datacenter a user will be connected to resides in the application code.

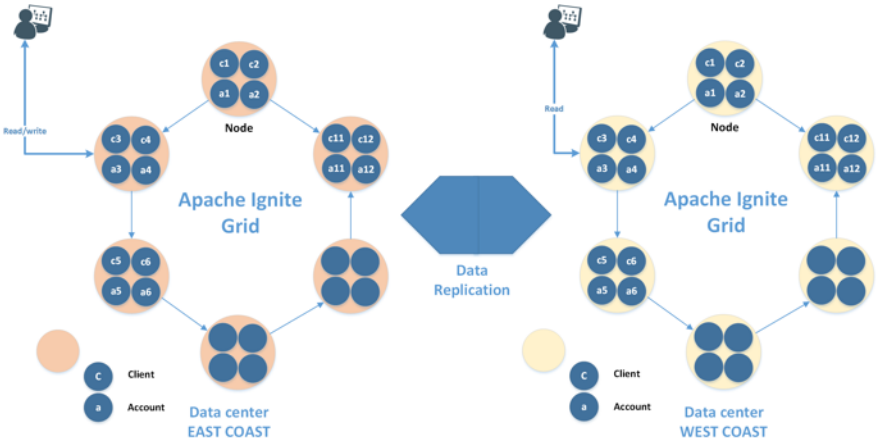


Figure 2.22. Geolocational data replication

Live backup scenario. In this scenario, most users use different datacenter as a live backup that can quickly be used 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.

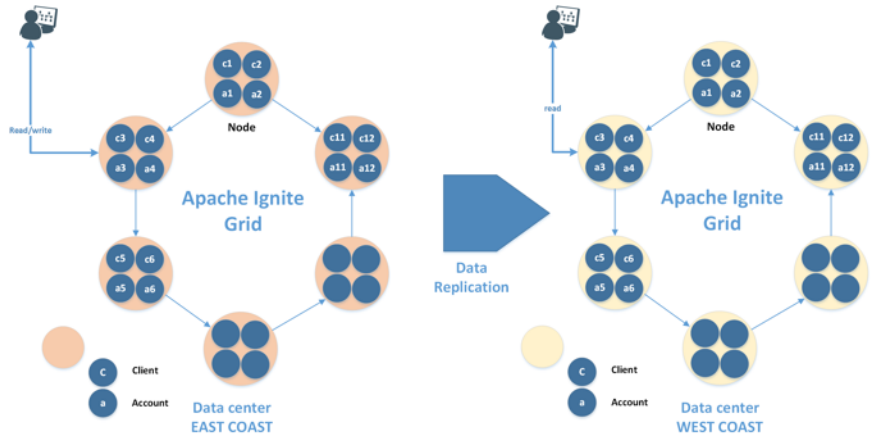


Figure 2.23. Realtime backup

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-data center replication. However, you can span Ignite nodes in different data centers (for instance, 10 nodes in one data center and other 10 nodes in another).

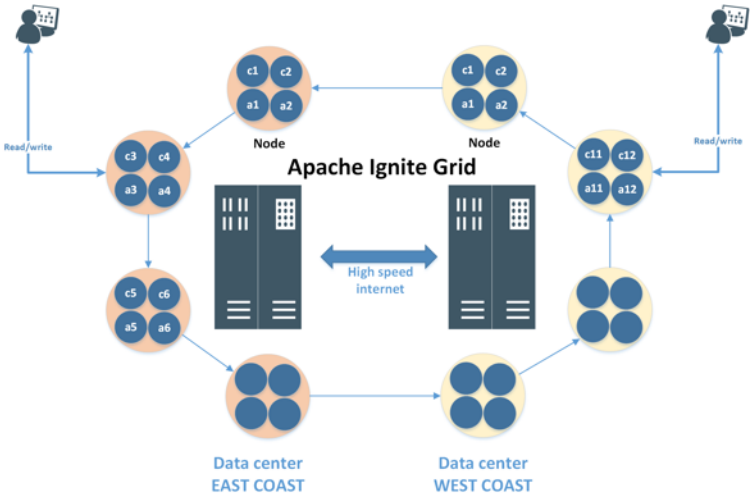


Figure 2.24. Span Ignite grid into multiple DC

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 is going to face very different latency for client operations.

However, GridGain[10](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 read a more detail here in GridGain site.

Asynchronous support

Usually, any plain(synchronous) put/get or execute call blocks the execution of the application until the result is available from the server. Afterward, the result can be iterated and consumed as needed by the client. This is probably the most common way to deal with any persistence store. However, asynchronous method execution can provide significant benefits and is a requirement of the modern system. Apache Ignite provides a flexible paradigm of using asynchronous and synchronous method call on all distributed APIs. It could be put/get any entries from the store or executes a job in a compute grid. Ignite asynchronous method execution is a non-blocking operation, and return an `IgniteFuture` object instead of the actual result. You can later get the result by calling `IgniteFuture.get()` method.

[10]https://gridgain.readme.io/docs/data-center-replication

> **Note:**
>
> Any asyncCache.future() call returns the IgniteFuture object immediately. IgniteFuture is general concurrency abstraction, also known as a promise, which promises to return a result in the near future.

Here is a very simple example of using asynchronous invocation for getting entries from the Ignite cache (see the github project-installation[11] for the complete example).

```
// get or create cache
IgniteCache<Integer, String> cache = ignite.getOrCreateCache("testCache");
// get an asynchronous cache
IgniteCache<Integer, String> asynCache = cache.withAsync();

// put some cache elements
for(int i = 1; i <= 100; i++){
    cache.put(i, Integer.toString(i));
}

//get the first entries asynchronously
asynCache.withAsync().get(1);
// get the future promise
IgniteFuture<String> igniteFuture = asynCache.future();
// java 8 lamda expression
igniteFuture.listen(f-> System.out.println("Cache Value:" + f.get()));
```

In the above pseudo code, we have inserted 100 entries synchronously into the Ignite cache *testCache*. After that, asynchronously request the first entry and got future for the invocation. Next, we asynchronously listen for the operation to complete (see the GitHub project-installation for the complete example). Note that, the main thread of the Java main() method doesn't wait for the task to complete, rather it hands over the task to another thread and moves on.

Resilience

Ignite client node is completely resilient in nature. The word resilience means the ability of a server or system to recover quickly and continue operating when there has been a failure. Ignite client node can disconnect from the cluster in several cases:

- In the case of a network problem.

[11] https://github.com/srecon/ignite-book-code-samples/tree/master/chapters/chapter-installation

- Host machine dies or is restarted.
- Slow clients can be disconnected by the server.

When the 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.

Security

Apache Ignite is an open source project and does not provide any security features. However, the commercial version of Ignite has this functionality. The commercial version is called **GridGain Enterprise Edition**. GridGain enterprise edition provides extensible and customizable authentication and security features to satisfy a variety of security requirements for Enterprise.

Key API

Apache Ignite provides a rich set of different API's to work with the Data Fabrics. The APIs are implemented in the form of native libraries for major languages and technologies as Java, .NET and C++. A very short list of some useful APIs is described below.

Name	Description
org.apache.ignite.IgniteCache	The main cache interface, the entry point for all Data Grid API's. The interface extends javax.cache.Cache interface.
org.apache.ignite.cache.store.CacheStore	Interface for cache persistence storage for read-through and write-through behavior.
org.apache.ignite.cache.store.CacheStoreAdapter	The cache storage convenience adapter, implements interface CacheStore. It's provides default implements for bulk operations, such as writeAll and readAll.
org.apache.ignite.cache.query.ScanQuery	Scan query over cache entries.
org.apache.ignite.cache.query.TextQuery	Query for Lucene based fulltext 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.

Name	Description
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.
org.apache.ignite.IgniteMessaging	An interface that provides functionality for topic-based message exchange among nodes defined by ClusterGroup.

Conclusion

We covered the foundational concepts of the in-memory data grid. We first briefly described the Ignite functional overview and various caching strategies with different topologies. We also presented some techniques based on standard data access patterns such as caching-aside, read-through, and write-through. We went through the Ignite data collocation technics and briefly explained the Ignite key-value data store.

What's next

In the next chapter, we will look at the Ignite data grid implementations such as 2nd level cache, web session clustering and so on.

Chapter five: Accelerating Big Data computing

Hadoop has quickly become the standard for business intelligence on huge data sets. However, it's batch scheduling overhead, and disk-based data storage have made it unsuitable for use in analyzing live, real-time data in a 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. I/O adds latency that delays the Map/Reduce computation. An alternative is to store the needed distributed data within the memory. Placing Map/Reduce in-memory with the data it needs, eliminates file I/O latency.

Apache Ignite has offered a set of useful components allowing in-memory Hadoop job executing and file system operations. Even, Apache Ignite provides an implementation of Spark RDD which allows sharing state in-memory across Spark applications.

In this chapter, we are going to explore how big data computing with Hadoop/Spark can be performed by combining an in-memory data grid with an integrated, standalone Hadoop Map/Reduce execution engine, and how it can accelerate the analysis of large, static data sets.

Hadoop accelerator

Ignite in-memory Map/Reduce engine executes Map/Reduce programs in seconds (or less) by incorporating several techniques. By avoiding Hadoop's batch scheduling, it can start up jobs in milliseconds instead of tens of seconds. In-memory data storage dramatically reduces access times by eliminating data motion from the disk or across the network. This is the Ignite approach to accelerating Hadoop application performance without changing the code. The main advantages are, all the operations are highly transparent and that it is accomplished without changing a line of MapReduce code.

Ignite Hadoop in-memory plug and play accelerator can be grouped by into three different categories:

- **In-memory Map/Reduce**: It's an alternative implementation of Hadoop Job tracker and task tracker, which can accelerate job execution performance. It eliminates the overhead associated with job tracker and task trackers in a standard Hadoop architecture while providing low-latency, HPC-style distributed processing.
- **Ignite in-memory file system (IGFS)**: It's also an alternate implementation of Hadoop file system named *IgniteHadoopFileSystem*, which can store data sets in-memory. This in-memory file system minimizes disk I/O and improves performances.

- **Hadoop file system cache**: This implementation works as a caching layer above HDFS, every read and write operations would go through this layer and can improve Map/Reduce performance. See chapter two for more about read-write behind cache strategy.

Conceptual architecture of the Ignite Hadoop accelerator is shown in figure 5.1:

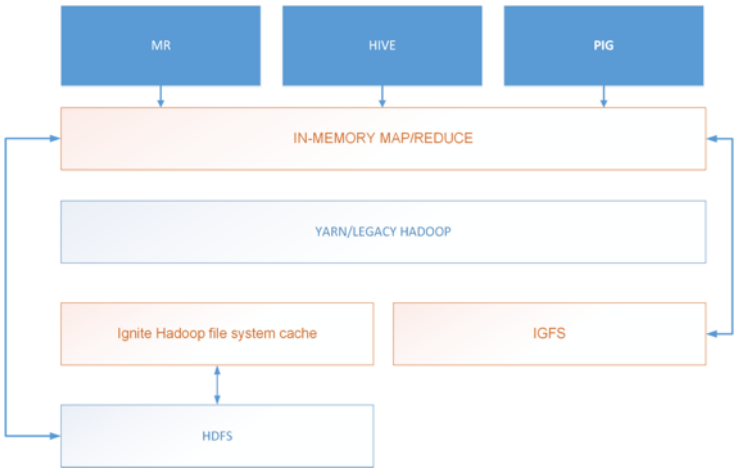


Figure 5.1

The apache Ignite Hadoop accelerator tool is especially very useful when you already have an existing Hadoop cluster up and running and want to get more performance with minimum efforts. I can still remember my first Hadoop project in the summer of the year 2011. After a few months of use of our 64 nodes Hadoop cluster, we stuck were with the problem of data motion between Hadoop nodes it was very painful to overcome. This time, Spark was not so much matured to use. In the long run, we made a serious change in our infrastructure, replaced most of the hard drive to SSD. As a consequence, it was very costly.

Note:

Hadoop running on commodity hardware is a *myth*. Most of the Hadoop process is I/O intensive and requires homogenous and mid-end servers to perform well.

When running Map/Reduce using one of the most popular open source distribution of Hadoop, Hadoop Map/Reduce introduces numerous I/O overhead that extends analysis times to minutes.

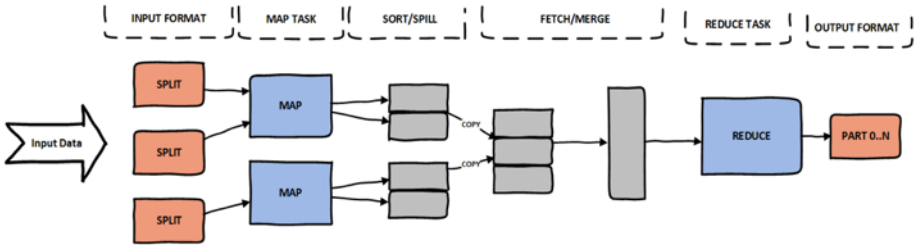


Figure 5.2

The generic phases of Hadoop job are shown in figure 5.2, phase sort, merge or shuffle are highly I/O intensive. These overheads are prohibitive when running real-time analytics that returns the result in milliseconds or seconds. With Ignite IGFS, you can replace the Hadoop HDFS and eliminate the I/O overhead, which can boost Map/Reduce performance. Because in-memory data grid hosts fast-changing data in memory, Map/Reduce application can input data directly from the grid and output results back to the grid. This speeds up the analysis by avoiding the delay in accessing HDFS during real-time analysis.

However, to decrease the access time for reading data into Hadoop from HDFS, the input data set can be cached within the Ignite cache storage. Ignite Hadoop file system (IGFS) cache provides a distributed caching feature that speeds access times by capturing data from HDFS or other data sources during Map/Reduce processing and working as a second level cache for HDFS. Ignite Hadoop accelerator is compatible with the latest versions of the most popular Hadoop platforms, including Apache, Cloudera and Horton. This means that you can run fully compatible MapReduce applications for any of these platforms on Ignite in-memory Map/Reduce engine.

Next, we are going to explore the details of the Hadoop accelerator tools.

In-memory Map/Reduce

This is the Ignite in-memory Map/Reduce engine, which is 100% compatible with Hadoop HDFS and Yarn. It reduces the startup and the execution time of the Hadoop Job tracker and the Task tracker. Ignite in-memory Map/Reduce provides dramatic performance boosts for CPU-intensive tasks while requiring an only minimal change to existing applications. This module also provides an implementation of weight based Map/Reduce planner, which assigns mappers and reducers based on their weights. Weight describes how much resources are required to execute the particular map and reduce task. This planning algorithm assigns mappers so that, total resulting weight on all nodes as minimal as possible. Reducers are assigned slightly different.

This approach minimizes the expensive data motion over the network. Reducer assigned to a node with mapper are called local. Otherwise, it is considered as remote. High-level architecture of the Ignite in-memory Map/Reduce is shown below:

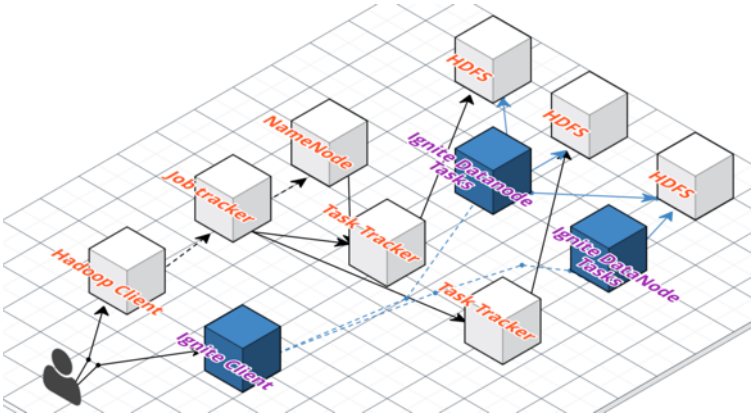


Figure 5.3

Ignite in-memory grid has pre-stage java based execution environment on all grid nodes and reuse it for multiple data processing. This execution environment consists of a set of Java virtual machines one on each server within the cluster. This JVM's forms the Ignite Map/Reduce engine as shown in figure 5.2. Also, the Ignite in-memory data grid can automatically deploy all necessary executable programs or libraries for the execution of the Map/Reduce across the grid, this greatly reduces the startup time down to milliseconds.

Now that, we have got the basics, let's try to configure the sandbox and execute a few Map/Reduce jobs in Ignite Map/Reduce engine.

For simplicity, we are going to install a Hadoop *Pseudo-Distributed* cluster in a single virtual machine and will run Hadoop famous *word count* example as a Map/Reduce job and then we will complicate our example to study the other features of the Ignite Hadoop accelerator tool.

> **Note:**
>
> Hadoop `Pseudo-Distributed` cluster means, Hadoop datanode, namenode, tasktracker/jobtracker, everything will be on one virtual (Host) machine.

Let's have a look at our sandbox configuration as shown below.

Sandbox configuration:

VM	VMWare
OS	RedHat enterprise Linux
CPU	2
RAM	2 Gb
JVM version	1.7_60
Ignite version	1.6, single node cluster

First of all, we are going to install and configure Hadoop and will proceed to Apache Ignite. Assuming that, Java has been installed, and JAVA_HOME is in the environment variables.

Step 1:

Unpack the Hadoop distribution archive and set the JAVA_HOME path in the `etc/hadoop/hadoop-env.sh` file as follows. This step is optional if your JAVA_HOME is properly configured in Linux box.

```
export JAVA_HOME=JAVA_HOME_PATH
```

Step 2:

Add the following configuration in the `etc/hadoop/core-site.xml` file.

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

Also append the following data replication strategy into the `etc/hadoop/hdfs-site.xml` file.

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

Step 3:

Setup password less or passphrase less ssh for your operating system.

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

Try the following command into your console.

```
$ ssh localhost
```

It shouldn't ask you for input password.

Step 4:

Format the Hadoop HDFS file system.

```
$ bin/hdfs namenode -format
```

Next, start the *namenode/datanode* daemon by the following command:

```
$ sbin/start-dfs.sh
```

Also, I would like to suggest you to add the `HADOOP_HOME` environmental variable to operating system.

Step 5:

Make a few directories in HDFS file system to run Map/Reduce jobs.

```
bin/hdfs dfs -mkdir /user
bin/hdfs dfs -mkdir /input
```

The above command will create two folder `user` and `input` in HDFS file system. Insert some text files in directory *input*.

```
bin/hdfs dfs -put $HADOOP_HOME/etc/hadoop /input
```

Step 6:

Run the Hadoop native Map/Reduce application to count the words of the file.

```
$ bin/hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar w\
ordcount /input/hadoop output
```

You can view the result of the words count by the following command:

```
bin/hdfs dfs -cat output/*
```

In my case, the file is huge with words and its number of count, let's see the fragment of the file.

```
want 1
warnings. 1
when 9
where 4
which 7
while 1
who 6
will 23
window 1
window, 1
with 62
within 4
without 1
work 12
writing, 27
```

At this stage, our Hadoop pseudo cluster is configured and ready to use. Now let's start configuring the Apache Ignite.

Step 7:

Unpack the distribution of the Apache Ignite somewhere in your sandbox and add the *IGNITE_HOME* path to the root directory of the installation. For getting *statistics about tasks and executions*, you have to add the following properties in your `/config/default-config.xml` file.

```
<bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
  ....
  <property name="includeEventTypes">
    <list>
      <util:constant static-field="org.apache.ignite.events.EventType.EVT_TASK_FAILED"/>
      <util:constant static-field="org.apache.ignite.events.EventType.EVT_TASK_FINISHED"/>
      <util:constant static-field="org.apache.ignite.events.EventType.EVT_JOB_MAPPED"/>
    </list>
  </property>
</bean>
```

Above configuration will enable the event task for statistics.

Warning:

By default, all events are disabled. Whenever these above events are enabled, you can use the command "tasks" in *ignitevisor* to get statistics about tasks executions.

IgniteVisor **tasks** command is very useful to get aggregated results of all executed tasks for a given

period of time. Check the *chapter-bigdata/src/main/config/ignite-inmemory* directory from GitHub repository[12] for complete configuration of the `defult-config.xml` file.

Step 8:

Add the following libraries in the `$IGNITE_HOME/libs` directory.

```
asm-all-4.2.jar
ignite-hadoop-1.6.0.jar
hadoop-mapreduce-client-core-2.7.2.jar
hadoop-common-2.7.2.jar
hadoop-auth-2.7.2.jar
```

Note that, *asm-all-4.2.jar* library version is dependent on your Hadoop version.

Step 9:

We are going to use the Apache Ignite default configuration, `config/default-config.xml` file to start the Ignite node. Start the Ignite node with the following command.

```
bin/ignite.sh
```

Step 10:

Add a few more staffs to use Ignite job tracker instead of Hadoop. Add the `HADOOP_CLASSPATH` to the environmental variables as follows.

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:$IGNITE_HOME/libs/ignite-core-1.6.0.jar:$IGNITE_\
HOME/libs/ignite-hadoop-1.6.0.jar:$IGNITE_HOME/libs/ignite-shmem-1.0.0.jar
```

Step 11:

In this stage, we are going to override the Hadoop `mapred-site.xml` file. For a quick start, add the following fragment of xml to the mapred-site.xml.

[12] https://github.com/srecon/ignite-book-code-samples

```
<property>
  <name>mapreduce.framework.name</name>
  <value>ignite</value>
</property>
<property>
  <name>mapreduce.jobtracker.address</name>
  <value>127.0.0.1:11211</value>
</property>
```

Note that, we explicitly added the Map/Reduce framework to Ignite. Port *11211* is the default port to listening for the task.

> **Note:**
>
> You don't need to restart the Hadoop processes after making any change to `mapred-site.xml`.

Step 12:

Run the above example of the word count Map/Reduce example again.

```
$bin/hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wo\
rdcount /input/hadoop output2
```

The output should be very similar as shown in figure 5.4.

Figure 5.4

Now Execution time is faster than last time whenever we have used Hadoop task tracker. Let's examine the Ignite task execution statistics through Ignite visor:

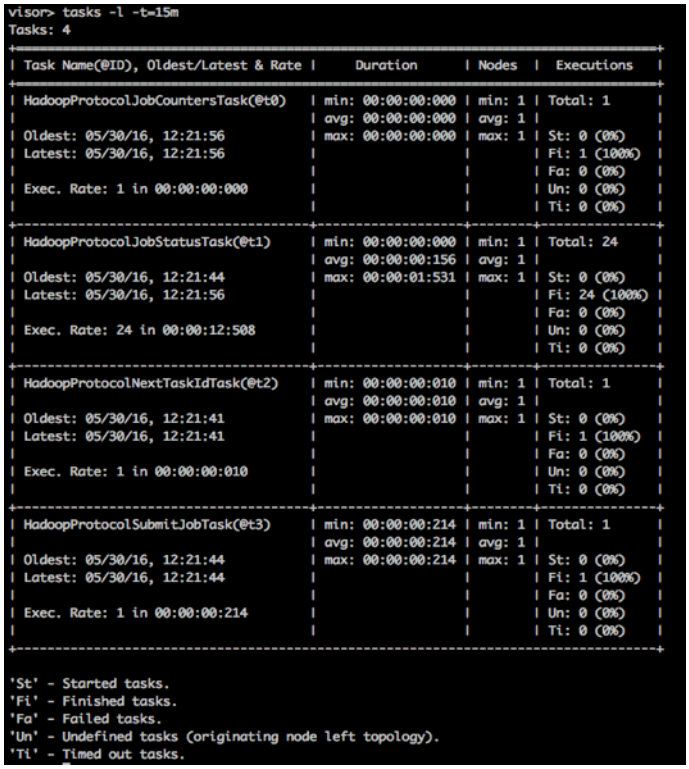


Figure 5.5

From the figure 5.5, we should notice that, the total executions and the durations times of the in-memory task tracker. In our case, total executions task (*HadoopProtocolJobStatusTask(@t1)*) is 24 and the execution rate are 12 second.

Benchmark

To demonstrate the performance advantage of the Ignite Map/Reduce engine, measurements were made of the familiar Hadoop WordCount and PI sample application. This program was run both on the standard Apache Hadoop distribution and on an Ignite grid that included a built-in Hadoop MapReduce execution engine.

The Hadoop distribution comes with a number of benchmarks, which are bundled in hadoop-*test*.jar and hadoop-*examples*.jar. The two benchmarks we are going to use are `WordCount` and `PI`. The full list of available options for hadoop-*examples*.jar are shown bellow:

```
$ bin/hadoop jar hadoop-*examples*.jar
An example program must be given as the first argument.
Valid program names are:
  aggregatewordcount: An Aggregate based map/reduce program that counts the words in the i\
nput files.
  aggregatewordhist: An Aggregate based map/reduce program that computes the histogram of \
the words in the input files.
  dbcount: An example job that count the pageview counts from a database.
  grep: A map/reduce program that counts the matches of a regex in the input.
  join: A job that effects a join over sorted, equally partitioned datasets
  multifilewc: A job that counts words from several files.
  pentomino: A map/reduce tile laying program to find solutions to pentomino problems.
  pi: A map/reduce program that estimates Pi using monte-carlo method.
  randomtextwriter: A map/reduce program that writes 10GB of random textual data per node.
  randomwriter: A map/reduce program that writes 10GB of random data per node.
  secondarysort: An example defining a secondary sort to the reduce.
  sleep: A job that sleeps at each map and reduce task.
  sort: A map/reduce program that sorts the data written by the random writer.
  sudoku: A sudoku solver.
  teragen: Generate data for the terasort
  terasort: Run the terasort
  teravalidate: Checking results of terasort
  wordcount: A map/reduce program that counts the words in the input files.
```

The wordcount example, for instance, will be used to count the word from the The `Complete Works of William Shakespeare` file. On the other hand, we are going to use the `PI` example to calculate the digit of pi using Monte-Carlo method. However, it's not recommended to use Hadoop pseudo-distributed server for benchmarking. We will do it for academic purposes.

Sandbox for the benchmark:

VM	**VMWare**
OS	RedHat enterprise Linux
CPU	2
RAM	2 Gb
JVM version	1.7_60
Ignite version	1.6, single node cluster
Hadoop version	2.7.2, pseudo cluster

First of all, we are going to use the `wordcount` example to count words from the *The Complete Works of William Shakespeare* file.

Step 1:

Create another input directory in HDFS to store the file `t8.shakespeare.txt`. You can download the file from the *input* directory of the GitHub project `chapter-bigdata/src/main/input/t8.shakespeare.txt`.

The file size is approximately 5,5 MB.

```
hdfs dfs -mkdir /wc-input
```

Step 2:

Store the file into the HDFS *wc-input* directory.

```
hdfs dfs -put /YOUR_PATH_TO_THE_FILE /t8.shakespeare.txt /wc-input
```

Step 3:

Comment the following fragment of the properties into the mapred-site.xml.

```
<property>
        <name>mapreduce.framework.name</name>
        <value>ignite</value>
</property>
<property>
        <name>mapreduce.jobtracker.address</name>
        <value>localhost:11211</value>
</property>
```

Step 4:

And before we start, here's a nifty trick for your tests: When running the benchmarks described in the following sections, you might want to use the Unix `time` command to measure the elapsed time. This saves you the hassle of navigating to the Hadoop *JobTracker* web interface to get the (almost) same information. Simply prefix every Hadoop command with `time` command as follows:

```
time hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wo\
rdcount /wc-input/ output6
```

Run the above job a few times. You should get the relevant value (real) into the console as shown below:

```
16/09/05 16:05:56 INFO mapred.LocalJobRunner: 2 / 2 copied.
16/09/05 16:05:56 INFO mapred.Task: Task attempt_local1864495227_0001_r_000000_0 is allowe\
d to commit now
16/09/05 16:05:56 INFO output.FileOutputCommitter: Saved output of task 'attempt_local1864\
495227_0001_r_000000_0' to hdfs://localhost:9000/user/user/output20/_temporary/0/task_loca\
l1864495227_0001_r_000000
16/09/05 16:05:56 INFO mapred.LocalJobRunner: reduce > reduce
16/09/05 16:05:56 INFO mapred.Task: Task 'attempt_local1864495227_0001_r_000000_0' done.
16/09/05 16:05:57 INFO mapreduce.Job:  map 100% reduce 100%
16/09/05 16:05:57 INFO mapreduce.Job: Job job_local1864495227_0001 completed successfully
16/09/05 16:05:57 INFO mapreduce.Job: Counters: 35
                File System Counters
                        FILE: Number of bytes read=2948198
                        FILE: Number of bytes written=5833532
                        FILE: Number of read operations=0
                Map-Reduce Framework
                        Map input records=128803
                        Map output records=926965
                        Map output bytes=8787114
                        Map output materialized bytes=1063186
                        Input split bytes=219
                Shuffle Errors
                        BAD_ID=0
                        CONNECTION=0
                        IO_ERROR=0
                        WRONG_LENGTH=0
                        WRONG_MAP=0
                        WRONG_REDUCE=0
                File Input Format Counters
                        Bytes Read=5596513
                File Output Format Counters
                        Bytes Written=724175
real            0m25.732s
user            0m14.582s
sys             0m0.616s
```

Step 5:

Next, run the Ignite version of the Map/Reduce for wordcount example. Uncomment or add the following fragment of the xml into `mapred-site.xml` file.

```
<property>
        <name>mapreduce.framework.name</name>
        <value>ignite</value>
</property>
<property>
        <name>mapreduce.jobtracker.address</name>
        <value>localhost:11211</value>
</property>
```

Execute the same command a few times as follows.

```
time hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wo\
rdcount /wc-input/ output7
```

In the console, you should get the following output.

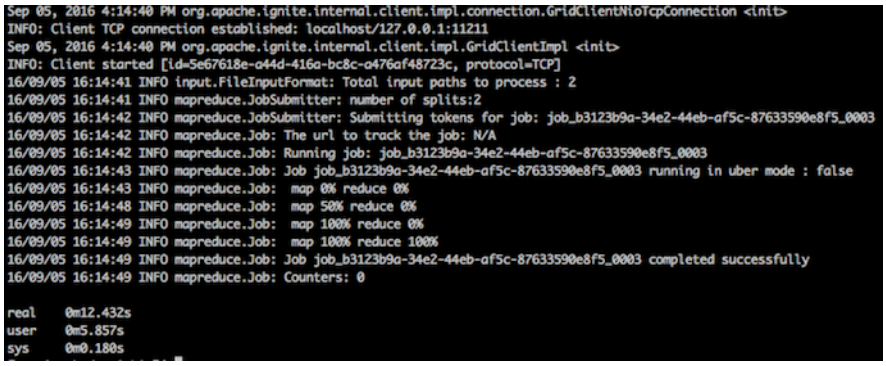


Figure 5.6

Step 6:

For the second benchmark test, we executes the `PI` example with 16 maps and 1000000 sample per map. Run the following command for Hadoop and Ignite Map/Reduce as follows:

```
time hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar pi\
 16 1000000
```

The output should be the same as shown below:

```
Sep 05, 2016 4:21:39 PM org.apache.ignite.internal.client.impl.connection.GridClientNioTcpConnection <init>
INFO: Client TCP connection established: localhost/127.0.0.1:11211
Sep 05, 2016 4:21:39 PM org.apache.ignite.internal.client.impl.GridClientImpl <init>
INFO: Client started [id=d011a8e1-8a96-4af4-afc6-baa7ca6f04be, protocol=TCP]
16/09/05 16:21:39 INFO input.FileInputFormat: Total input paths to process : 16
16/09/05 16:21:40 INFO mapreduce.JobSubmitter: number of splits:16
16/09/05 16:21:40 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_b3123b9a-34e2-44eb-af5c-87633590e8f5_0005
16/09/05 16:21:41 INFO mapreduce.Job: The url to track the job: N/A
16/09/05 16:21:41 INFO mapreduce.Job: Running job: job_b3123b9a-34e2-44eb-af5c-87633590e8f5_0005
16/09/05 16:21:42 INFO mapreduce.Job: Job job_b3123b9a-34e2-44eb-af5c-87633590e8f5_0005 running in uber mode : false
16/09/05 16:21:42 INFO mapreduce.Job:  map 0% reduce 0%
16/09/05 16:21:46 INFO mapreduce.Job:  map 31% reduce 0%
16/09/05 16:21:47 INFO mapreduce.Job:  map 100% reduce 0%
16/09/05 16:21:50 INFO mapreduce.Job:  map 100% reduce 100%
16/09/05 16:21:50 INFO mapreduce.Job: Job job_b3123b9a-34e2-44eb-af5c-87633590e8f5_0005 completed successfully
16/09/05 16:21:50 INFO mapreduce.Job: Counters: 0
Job Finished in 11.013 seconds
Estimated value of Pi is 3.14159125000000000000

real    0m17.814s
user    0m6.349s
sys     0m0.246s
```

Figure 5.7

After running the benchmark test, we can demonstrate the performance gain of the Ignite in-memory Map/Reduce. Two different test were run with same data sets 3 times and Ignite demonstrates a 1.7% performance gain on both tests as shown in figure 5.7 below.

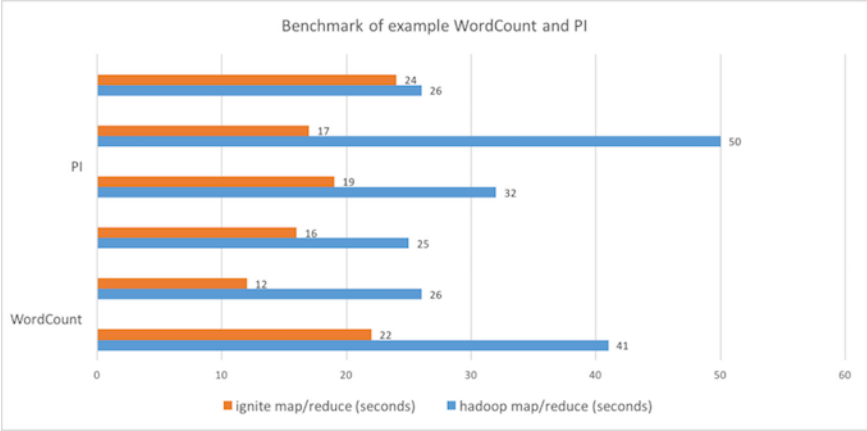


Figure 5.8

Note that, both of the tests executes on single node machine, and the result will be varying on real cluster. Anyway, we have seen how the Ignite can enable significant speed reduction in analysis time. Ignite also allows the input data set to be updated while the MapReduce analysis is in progress. Using the standard Hadoop MapReduce distribution, live updates are not possible since data in HDFS can only be appended and not updated.

Here, we are going to finish this section. In the next section, we will use Apache Pig for data analysis.

Using Apache Pig for data analysis

Apache Pig provides a platform for analyzing a very large-scale data set. With Apache Pig you can easily analyze your data from Hadoop HDFS. Apache Pig compiles instruction to sequences of Map/Reduce programs which will run on a Hadoop cluster. Apache Pig provides `PigLatin` scripting

language, which can perform operations like ETL or AdHoc data analysis. The Apache Pig was built to make programming Map/Reduce easier, before Pig, Java was the only way to process the data stored in HDFS.

Apache Ignite provides a transparent way to run PigLatin scripts in Ignite in-memory Map/Reduce. You have almost nothing to configure to run PigLatin script. Our setup from the previous section is enough to run in-memory Map/Reduce through PigLatin script. We just have to install Apache Pig in our environment to execute the PigLatin script.

Let's have a look at the features of Pig. Apache Pig comes with the following features:

- **Ease of programming** – PigLatin script is very much similar to SQL script. It's very easy to write a script in Pig if you are good at SQL.
- **Analytical functions** – Pig provides a lot of functions and operators to perform operations like `Filter`, `GroupBy`, `Sort`, etc.
- **Extensibility** – With existing operators, a user can develop their custom functions to read, write or process data. For example, we have developed a Pig function to parse XML and validate XML documents from HDFS.
- **Handles all kinds of Data** - Apache Pig can process or handle any kind of data, both structured or semi-structured. It can store the result into HDFS, Cassandra or Hbase.

Usually, Apache Pig is used by the data scientist or data analyst for performing `ad-hoc` scripting and quick prototyping. Apache Pig is hugely used in batch processing, such as

- Processing time sensitive data.
- Data analysis through sampling, for example weblogs.
- Data processing for search platform.

Although Map/Reduce is a powerful programming model, there is a significant difference between Apache Pig and Map/Reduce. The major difference between Pig and the Map/Reduce are shown below:

Pig	MapReduce
Apache Pig is a data flow language.	Map/Reduce is a data processing paradigm.
Apache Pig is a high-level language.	Map/Reduce is a low-level language, almost written in java or Python.
During execution of the PigLatin script, every Pig operator is converted internally into a Map/Reduce job.	Map/Reduce job have a long compilation process.

Now, we are going to use two different datasets for our examples, one of them are `Shakespeare all works` in a text file, and another one is the `movies_data.csv` CSV file, which contains a list of movies

name, release year, rating, etc. Both of them reside in the directory */chapter-bigdata/src/main/input*.

Next, we explains how to install, setup and run PigLatin scripts in-memory Map/Reduce.

It is essential that you have Java and Hadoop installed and configured in your sandbox to get up and running with Pig. Check the previous section to configure Hadoop and Ignite into your system.

Let's have a look at our sandbox configuration as shown below.

VM	**VMWare**
OS	RedHat enterprise Linux
CPU	2
RAM	2 Gb
JVM version	1.7_60
Ignite version	1.6, single node cluster
Hadoop version	2.7.2, pseudo cluster
Pig version	0.16.0

Next, we will download and install Apache Pig in our sandbox.

Step 1:

Download the latest version of the Apache Pig from the following link[13]. In our case, the Pig version is 0.16.0.

Step 2:

Untar the gz archive anywhere in your sandbox.

```
tar xvzf pig-0.16.0.tar.gz
```

And rename the folder for easier access as follows:

```
mv pig-0.16.0 pig
```

Step 3:

Add */pig/bin* to your path. Use export (bash, sh) or setenv (csh).

```
export PATH=/<my-path-to-pig>/pig/bin:$PATH
```

Step 4:

Test the pig installation with the following simple command.

[13] https://github.com/srecon/ignite-book-code-samples

```
pig --help
```

That's it, now you are ready to run PigLatin scripts. Apache Pig scripts can be executed in three different ways: an `interactive mode`, `batch mode` and, `embedded mode`.

1. Interactive mode or Grunt shell – In this shell, you can enter the PigLatin statements and get the output or results.
2. Batch mode or script mode – You can run Apache Pig scripts in batch mode by writing the Pig Latin scripts in a single file with extension *pig*.
3. Embedded mode - Apache Pig provides the provision of defining our own functions (User Defined Functions) in programming languages such as Java, and using them in your script.

You can run the Grunt shell in desired mode (local/cluster) using the –x option as follows:

Figure 5.9

With –x option, Grunt shell will start in cluster mode as above. In figure 5.9 we can notice that Pig is connected to Ignite in-memory job tracker at port 11211. Now, we execute the example of classic `wordcount` application with Apache Pig. For that, we are going to use our previously inserted `t8.shakespeare.txt` file from the Hadoop HDFS.

Step 5:

Use the following command to load the data from HDFS.

```
A = load '/wc-input/t8.shakespeare.txt';
```

The above statement is made up of two parts. The part to the left of "=" is called the relation or alias. It looks like a variable but you should notice that this is not a variable. When this statement is executed, no Map/Reduce task is executed. On the other hand, all the data loads from the HDFS to variable `A`.

Step 6:

Let's generate data transformation based on the column of data. Sometimes, we want to eliminate nesting, this can be accomplished by `flattening` keywords.

```
B = foreach A generate flatten(TOKENIZE((chararray)$0)) as word;
```

Step 7:

Using the `group` keyword, we can group together all the tuples that have the same group key.

```
C = group B by word;
```

Step 8:

Use the `count` function to compute the number of elements in the bags.

```
D = foreach C generate COUNT(B), group;
```

That's is. Only four lines of code and the wordcount example is ready to go. At this moment, we can dump the results into the console or store the result into the HDFS file. Let's dump the output into the console.

```
E = limit D 100;
dump E;
```

Before dumping the result, we limit it to 100 rows for demonstration purpose. Whenever you execute the DUMP command, in-memory Map/Reduce will run and compute the count of all words in the text file and output the result to the console as shown below.

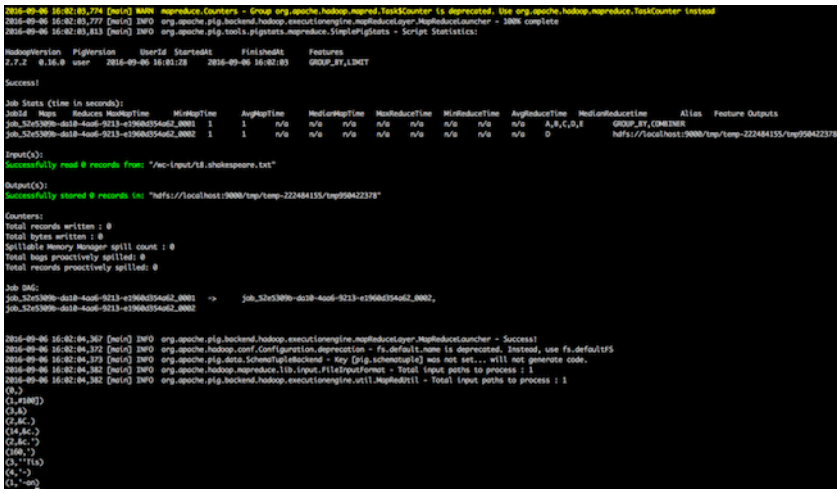


Figure 5.10

Let's have a look at the Ignite task statistics through *ignitevisor* as follows:

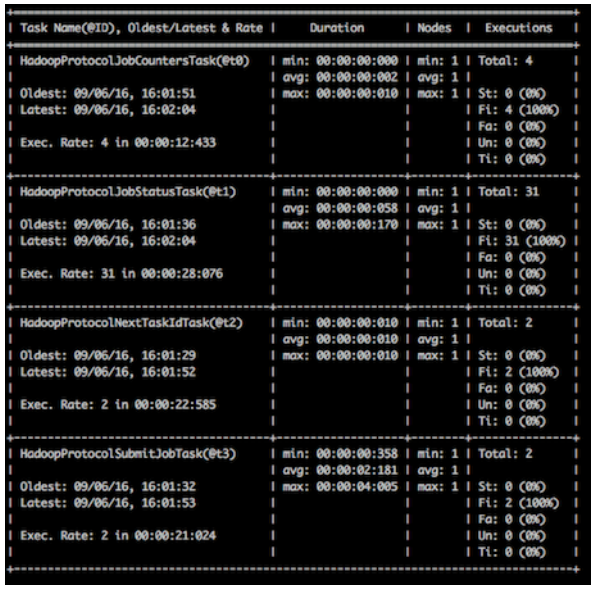


Figure 5.11

In figure 5.11, we can see the different statistics for different task executions. Total execution task is 39 with average 12 seconds.

Now that we have got the basics let's write more complex Pig script. Here we are going to use another dataset, *movies_data.csv*. The sample dataset is as follows:

```
1,The Nightmare Before Christmas,1993,3.9,4568
2,The Mummy,1932,3.5,4388
3,Orphans of the Storm,1921,3.2,9062
4,The Object of Beauty,1991,2.8,6150
5,Night Tide,1963,2.8,5126
6,One Magic Christmas,1985,3.8,5333
7,Muriel's Wedding,1994,3.5,6323
8,Mother's Boys,1994,3.4,5733
9,Nosferatu: Original Version,1929,3.5,5651
10,Nick of Time,1995,3.4,5333
```

Where the 2nd column is the name of the move, next column is the movie release year and the 4th column will be the overall rating of the certain movies. Let's upload the `movies_data.csv` file into Hadoop HDFS.

Step 1:

Upload `movies_data.csv` file into the HDFS.

```
bin/hdfs dfs -put /YOUR_PATH_TO_THE_FILE/movies_data.csv/wc-input
```

Step 2:

Execute the following pig statement to load the file from the HDFS.

```
movies = LOAD '/wc-input/hadoop/movies_data.csv' USING PigStorage(',') as (id:int,name:cha\
rarray,year:int,rating:double,duration:int);
```

Step 3:

Let's make a search of movies with the rating greater than four.

```
movies_rating_greater_than_four = FILTER movies BY (float) rating>4.0;
```

Step 4:

Dump the result into the console.

```
DUMP movies_rating_greater_than_four;
```

The output should be huge.

Step 5:

Search for the movie with the name `Mummy`;

```
movie_mummy = FILTER movies by (name matches '.*Mummy.*');
```

The output should be the same as shown below:

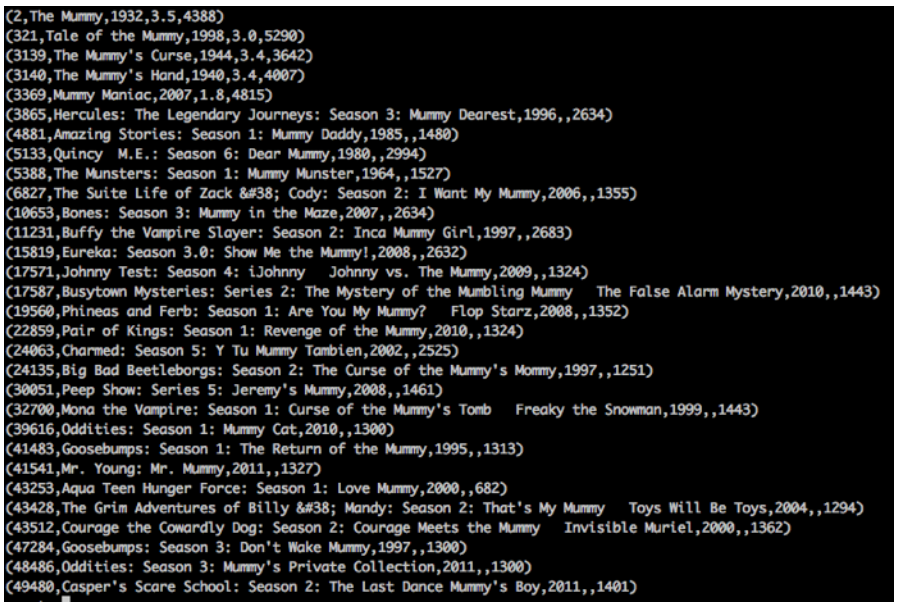


Figure 5.12

You can group movies by year or even count movies by year. The full source code of the Pig script is as follows:

```
movies = LOAD '/input/hadoop/movies_data.csv' USING PigStorage(',') as (id:int,name:charar\
ray,year:int,rating:double,duration:int);
movies_rating_greater_than_four = FILTER movies BY (float)rating > 4.0;
DUMP movies_rating_greater_than_four;
movie_mummy = FILTER movies by (name matches '.*Mummy.*');
grouped_by_year = group movies by year;
count_by_year = FOREACH grouped_by_year GENERATE group, COUNT(movies);
group_all = GROUP count_by_year ALL;
sum_all = FOREACH group_all GENERATE SUM(count_by_year.$1);
DUMP sum_all;
```

The dump command is only used to display the information on the standard output. If you need to store the result to a file, you can use the pig Store command as follows:

```
store sum_all into '/user/hadoop/sum_all_result;
```

In this section, we got a good feel of Apache Pig. We installed Apache Pig from scratch, loaded some data and executed some basic commands to query by in-memory Map/Reduce of Ignite. The next section will cover Apache Hive to analyze Big Data by Ignite in-memory Map/Reduce.

Near real-time data analysis with Hive

Apache Hive is a data warehouse framework for querying and analyzing data that is stored in Hadoop HDFS. Unlike Apache Pig, Hive provides SQL-like declarative language, called HiveQL, which is used for expressing queries. The Apache Hive was designed to appeal to a community comfortable with SQL. Its philosophy was that we don't need another scripting language. Usually, Hive engine compiles the HiveQL quires into Hadoop Map/Reduce jobs to be executed on Hadoop. In addition, custom Map/Reduce scripts can also be plugged into queries.

Hive operates on data stored in tables which consist of primitive data types and collection data types like arrays and maps. Hive query language is similar to SQL, where it supports subqueries. With Hive query language, it is possible to take a MapReduce joins across Hive tables. It has support for simple SQL-like functions- CONCAT, SUBSTR, ROUND, etc., and aggregation functions like SUM, COUNT, MAX, etc. It also supports GROUP BY and SORT BY clauses. It is also possible to write user defined functions in Hive query language.

Apache Ignite transparently supports Apache Hive for data analysis from the HDFS through in-memory Map/Reduce. No additional configurations to Ignite or the Hadoop cluster are necessary. Hive engines compile the HiveQL quires into Hadoop Map/Reduce jobs, which will be delegated into Ignite in-memory Map/Reduce jobs. Accordingly, the execution of the HiveQL quires is much faster than usual. Let's have a quick look at the features of Apache Hive. Hive provides the following features:

- It stores schema in a database such as derby and the processed data is stored in HDFS.

- It provides SQL-type language for scripting called HiveQL or HQL.
- It is fast, scalable and extensible.

Hive makes daily jobs easy when performing operations like:

- Ad-hoc quires.
- Analysis of huge datasets.
- Data encapsulation.

Unlike Apache Pig, Hive has a few important characteristics as follows:

1. In Hive, tables and databases are created first and then data is loaded into these tables.
2. The Apache Hive was designed for managing and querying only **structured** data that is stored in tables.
3. An important component of Hive i.e. Metastore used for storing schema information. This Metastore typically resides in a relational database.
4. For single user metadata storage, Hive uses derby database and for multiple users Metadata or shared Metadata case, Hive uses MYSQL.
5. Hive supports partition and buckets concepts for easy retrieval of data when the client executes the query.

For demonstration purpose, we are going to run `wordcount` example through HiveQL in the file `Shakespeare all works`. As we discussed before, we are not going to make any change to our Ignite or Hadoop cluster. In this section, we will explain how to properly configure and start Hive to execute HiveQL over Ignite Map/Reduce engine. Let's have a look at our sandbox configuration:

VM	**VMWare**
OS	RedHat enterprise Linux
CPU	2
RAM	2 Gb
JVM version	1.7_60
Ignite version	1.6, single node cluster
Hadoop version	2.7.2, pseudo cluster
Hive version	2.1.0

We are going to use Hive 2.1.0 version in our sandbox. You can download it by visiting the following link[14]. Let's assume that it gets downloaded into any of your local directory of the sandbox.

Step 1:

[14] http://apache-mirror.rbc.ru/pub/apache/hive/hive-2.1.0/

To install Hive, do the following:

```
tar zxvf apache-hive-2.1.0-bin.tar.gz
```

The above command will extract the archive into the directory called `hive-2.1.0-bin`. This directory will be your Hive home directory.

Step 2:

Let's create a new directory under `HADOOP_HOME/etc` with the following command and copy all the files from the hadoop directory. Execute the following command from the HADOOP_HOME/etc directory.

```
$ cd $HADOOP_HOME/etc
$ mkdir hadoop-hive
$ cp ./hadoop/*.* ./hadoop-hive
```

This `hadoop-hive` directory will be our `HIVE_CONF_DIRECTORY`.

Step 3:

Create a new file named `hive-site.xml` in the *hadoop-hive* directory with the following contents.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <!--
        Ignite requires query plan to be passed not using local resource.
    -->
    <property>
        <name>hive.rpc.query.plan</name>
        <value>true</value>
    </property>
</configuration>
```

Step 4:

Here, we will create a simple bash script, which will properly set all required variables and run Hive like this:

```
#!/usr/bin/env bash
# Specify Hive home directory:
export HIVE_HOME=<Hive installation directory>

# If you did not set hadoop executable in PATH, specify Hadoop home explicitly:
export HADOOP_HOME=<Hadoop installation folder>

# Specify configuration files location:
export HIVE_CONF_DIR=$HADOOP_HOME/etc/hadoop-hive

# Avoid problem with different 'jline' library in Hadoop:
export HADOOP_USER_CLASSPATH_FIRST=true

${HIVE_HOME}/bin/hive "${@}"
```

Place the bash script into the `$HIVE_HOME/bin` directory with name `hive-ig.sh`. Make the file runnable with the following command:

```
chmod +x ./hive-ig.sh
```

Step 5:

By default, Hive stores metadata information into Derby database. Before running Hive interactive console, run the following command to initialize Derby database.

```
schematool -initSchema -dbType derby
```

The above command will create a directory called `metastore-db`.

Warning:

If you have any existing directory called `metastore-db`, you have to delete it before you initialize the derby database.

Step 6:

Start your Hive interactive console with the following command:

```
hive-ig.sh
```

It will take a few moments to run the console. If everything goes fine, you should have the following screen into your console.

Figure 5.13

Step 7:

Now, you can run HiveQL into the console. Let's create a table in Hive and load the file `t8.shakespeare.txt`, then run the SQL query to count the words from the file.

```
CREATE TABLE input (line STRING);
```

Check the table by the following command.

```
hive> show tables;
OK
input
Time taken: 0.028 seconds, Fetched: 1 row(s)
```

Now, we can load the data from the host file system by Hive load command.

```
LOAD DATA LOCAL INPATH '/home/user/hadoop/t8.shakespeare.txt' OVERWRITE INTO TABLE input;
```

Note that, you have to change the `INPATH` of the file according to your local path. The above command loads the whole file into the column `line` in table `input`. Here, we are ready to execute any HiveQL quires against our data. Let's run the query to count the words from the table called `input`.

```
SELECT word, COUNT(*) FROM input LATERAL VIEW explode(split(line, ' ')) lTable as word GRO\
UP BY word;
```

After running the above query, you should have the following information into your console.

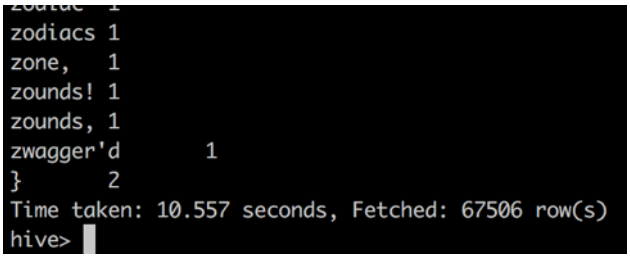


Figure 5.14

If you switch to the Ignite console, you should have a huge amount of logs into the console as follows:

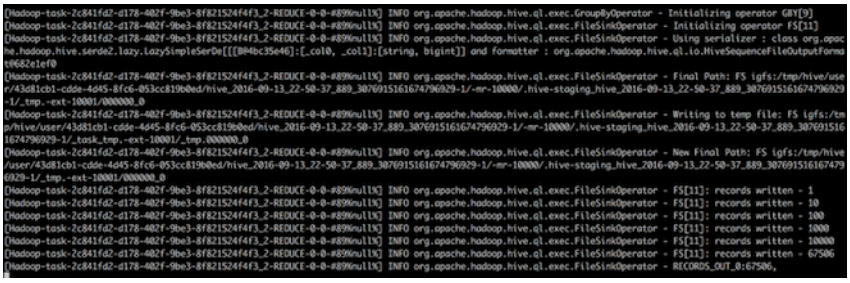


Figure 5.15

You can also use Limit keyword to restrict the maximum number of rows for the result set. With `Limit` clause, the above statement will look as follows:

```
SELECT word, COUNT(*) FROM input LATERAL VIEW explode(split(line, ' ')) lTable as word GRO\
UP BY word limit 5;
```

Now, let's try one more advanced example. We will use the dump of movies from the previous section to create a table into Hive and execute some HiveQL quires.

Step 8:

Create a new table into Hive with the following commands:

```
CREATE TABLE IF NOT EXISTS movies
 (id STRING,
 title STRING,
 releasedate STRING,
 rating INT,
 Publisher STRING)
 ROW FORMAT DELIMITED
 FIELDS TERMINATED BY ','
 STORED AS TEXTFILE;
```

Load some data from local directory as follows:

```
LOAD DATA local INPATH '/home/user/hadoop/hadoop-2.7.2/etc/hadoop/movies_data.csv' OVERWRI\
TE INTO TABLE movies;
```

You can get the *movies_data.csv* CSV data file from the GitHub project `/chapter-bigdata/src/main/input`.

Tip:

You can also load the data from the HDFS and IGFS file system. If you wish to load the data from the IGFS, you should configure the IGFS file system along with HDFS. In the next chapter, we will describe the complete process of deployment of the IGFS file system and run some Hive query.

Step 9:

Now that we have the data ready, let's do something interesting with it. The simple example is to see how many movies were released per year. We'll start with that, then see if we can do a bit more.

```
Select releasedate, count(releasedate) from movies m group by m.releasedate;
```

The above query should return the following output.

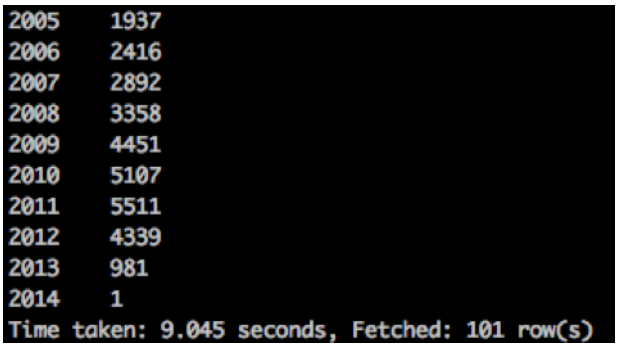


Figure 5.16

You can see the listing of years, along with the number of films released by that year. There's a lot more data in the set beyond years and films counts. Let's finding the films with the rating more than 3 per year. In Hive, this can be accomplished by the following query.

```
Select releasedate, count(releasedate) from movies m where rating >3  group by m.releaseda\
te;
```

The execution time of all the queries in Hive is very impressive. Execution time is around 9 second for the last query. It's very hard to compare the executions time between the Hive and Pig because of the data processing paradigm is different.

This ends the Hive section . In this section, we described how to install and configure Hive for use with Ignite in-memory Map/Reduce. We also loaded a few samples data sets and executed HiveQL quires.

Anyway, we can improve the query performance by using IGFS file system instead of HDFS. In the next section, we will briefly discuss the benefits of IGFS and how to use it.

Chapter six: Streaming and complex event processing

There is no broadly or highly accepted definition of the term Complex Event Processing or CEP. What Complex Event Processing is may be briefly described as the following quote from the Wikipedia:

Complex Event Processing, or CEP, is primarily an event processing concept that deals with the task of processing multiple events with the goal of identifying the meaningful events within the event cloud. CEP employs techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events such as causality, membership, and timing, and event-driven processes.

For simplicity, Complex Event Processing (CEP) is a technology for low-latency filtering, aggregating and computing on real-world never ending or streaming event data. The quantity and speed of both raw infrastructure and business events are exponentially growing in IT environments. In addition, the explosion of mobile devices and the ubiquity of high-speed connectivity add to the explosion of mobile data. At the same time, demand for business process agility and execution has only grown. These two trends have put pressure on organizations to increase their capability to support event-driven architecture patterns of implementation. Real-time event processing requires both the infrastructure and the application development environment to execute on event processing requirements. These requirements often include the need to scale from everyday use cases to extremely high velocities or varieties of data and event throughput, potentially with latencies measured in microseconds rather than seconds of response time.

Apache Ignite allows processing continuous never-ending streams of data in scalable and fault-tolerant fashion in in-memory, rather than analyzing data after it's reached the database. Not only does this enable you to correlate relationships and detect meaningful patterns from significantly more data, you can do it faster and much more efficiently. Event history can live in memory for any length of time (critical for long-running event sequences) or be recorded as transactions in a stored database.

Apache Ignite CEP can be used in a wealth of industries area, the following are some first class use cases:

1. **Financial services**: the ability to perform real-time risk analysis, monitoring and reporting of financial trading and fraud detection.
2. **Telecommunication**: ability to perform real-time call detail record and SMS monitoring and DDoS attack.

3. **IT systems and infrastructure**: the ability to detect failed or unavailable application or servers in real time.
4. **Logistics**: ability to track shipments and order processing in real-time and reports on potential delays on arrival.

There are a few more industrials or functional areas, where you can use Apache Ignite to process streams event data such as Insurance, transportation and Public sector. Complex event processing or CEP contains three main parts of its process:

1. Event Capture or data ingesting.
2. Compute or calculation of these data.
3. Response or action.

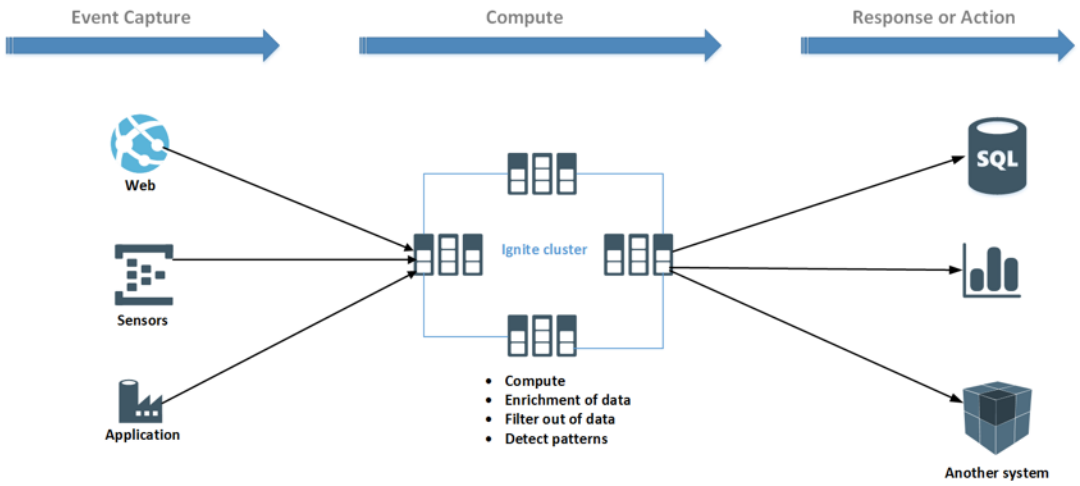


Figure 6.1

As shown in figure 6.1, data are ingesting from difference sources. Sources can be any sensors (IoT), web application or industry applications. Stream data can be concurrently processed directly on the Ignite cluster in collecting fashion. In addition, data can be enriched from other sources or filter out. After computing the data, computed or aggregated data can be exported to other systems for visualizing or taking an action.

Storm data streamer

Apache storm 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 data captured from network sensors, vehicles, physical devices, web, social data, transactional business data. With the volume of data being

generated, real-time computing has become a major challenge for most of the organization. In a short time, Apache Storm became a standard for 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 Clojure. It became a first choose for real-time analytics. Apache Ignite Storm streamer module provides a convenience way to streaming data via Storm to Ignite cache.

Although Hadoop and Storm frameworks are used for analyzing and processing big data, both of them complement each other and differ in a few aspects. Like Hadoop, Storm can process a huge amount of data but does it in real-time with guaranteed reliability, means, 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 is 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 the Storm. It's an ordered list of elements. Generally, tuple supports all primitives data types.

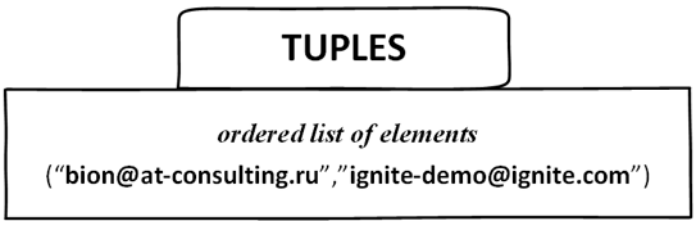


Figure 6.19

Streams – It's an unbound and un ordered sequence of tuples.

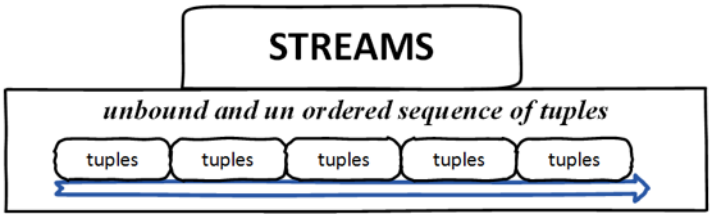


Figure 6.20

Spouts - Source of the streams, in simple terms, a spout reads the data from a source for useing in the topology. A spout can reliable or unreliable. A spout can talk with Queues, Web logs, event data etc.

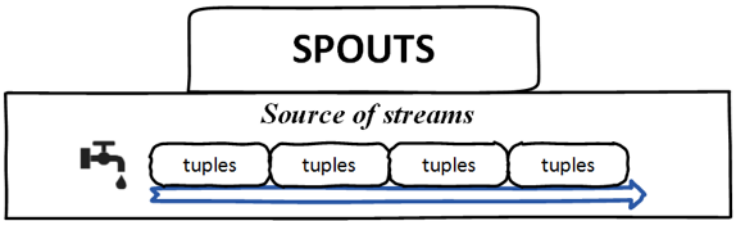


Figure 6.21

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 receives data from spout and emit to one or more bolts.

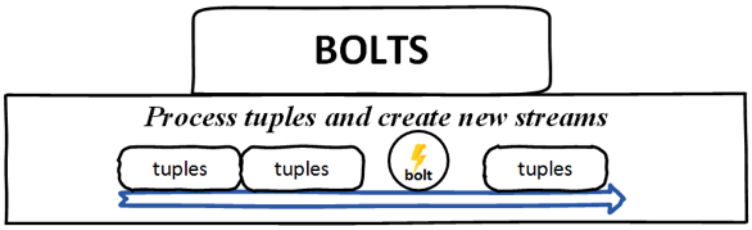


Figure 6.22

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, Storm keeps the topology running forever until you kill it. A simple topology starts with spouts, emit stream from the sources to bolt for processing data. Apache Storm main job is to run the any topology at given time.

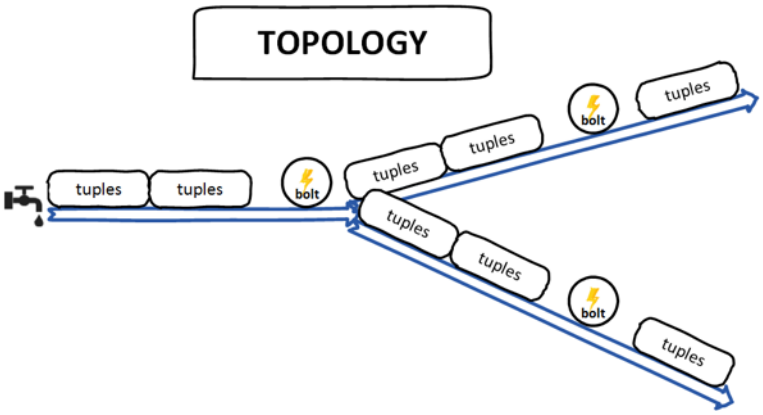


Figure 6.23

Ignite out of the box provides an implementation of Storm Bolt (StormStreamer) to 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 just have to implement *BaseBasicBolt* or *IRichBolt* Storm interface. However, if you decide to use StormStreamer, you have to configure a few properties to work the Ignite Bolt correctly. All mandatory properties are shown below:

№	Property Name	Description
1	CacheName	Cache name of the Ignite cache, in which the data will be store.
2	IgniteTupleField	Names of the Ignite Tuple field, by which tuple data is obtained in topology. By default the value is *ignite*.
3	IgniteConfigFile	This property will set the Ignite spring configuration file. Allows you to send and consume message to and from Ignite topics.
4	AllowOverwrite	It will enabling overwriting existing values in the cache, default value is false.
5	AutoFlushFrequency	Automatic flush frequency in milliseconds. Essentially, this is the time after which the streamer will make an attempt to submit all data added to remote nodes. Default is 10 sec.

Now, let's build something useful 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 huge amount of data from a traffic log files and trigger an alert when a specific value crosses a predefined threshold. 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 example 120km/h), Storm topology will send the data to 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.

```
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 taken from the *Dr. Dobbs* journal. Since this book is not for studying Apache Storm, I am going to keep the example 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 **chapter-cep/storm**. The above CSV file will be the source for the Storm topology.

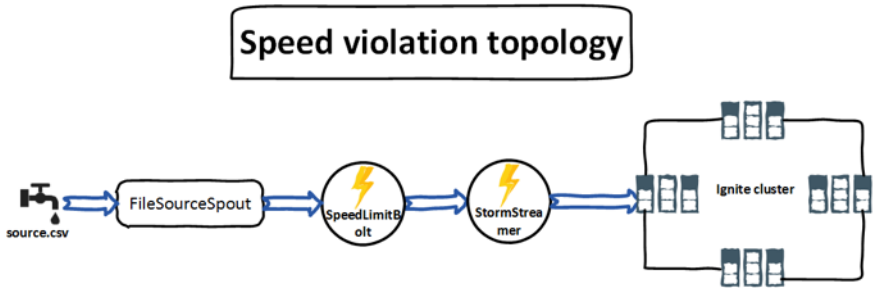


Figure 6.24

As shown in figure 6.24, 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:

Because this is a Storm topology, you must add the Storm and the Ignite *StormStreamer* dependency in the maven project.

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-storm</artifactId>
    <version>1.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-core</artifactId>
    <version>1.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-spring</artifactId>
    <version>1.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>0.10.0</version>
    <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 0.10.0 is only supported.

> **Note:**
>
> 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 `/chapter-cep/storm/src/resources/example-ignite.xml`) and make sure that it is available from the classpath. The content of the Ignite configuration is identical from the previous section of this chapter.

```
<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"/>
        <!-- Cache accessed from IgniteSink. -->
        <property name="cacheConfiguration">
            <list>
                <!-- Partitioned cache example configuration with configurations adjusted \
to server nodes'. -->
                <bean class="org.apache.ignite.configuration.CacheConfiguration">
                    <property name="atomicityMode" value="ATOMIC"/>

                    <property name="name" value="testCache"/>
                </bean>
            </list>
        </property>
        <!-- Enable cache events. -->
        <property name="includeEventTypes">
            <list>
                <!-- Cache events (only EVT_CACHE_OBJECT_PUT for tests). -->
                <util:constant static-field="org.apache.ignite.events.EventType.EVT_CACHE_\
OBJECT_PUT"/>
```

```
            </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.vm.TcpDiscov\
eryVmIpFinder">
                        <property name="addresses">
                            <list>
                                <value>127.0.0.1:47500</value>
                            </list>
                        </property>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

Step 3:

Create a *ignite-storm.properties* file to add the cache name, tuple name and the name of the Ignite configuration file as shown below.

```
cache.name=testCache
tuple.name=ignite
ignite.spring.xml=example-ignite.xml
```

Step 4:

Next, create *FileSourceSpout* Java class as shown below,

```
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 spoutO\
utputCollector) {
        this.outputCollector = spoutOutputCollector;
    }
@Override
    public void nextTuple() {
        try {
            Path filePath = Paths.get(this.getClass().getClassLoader().getResource("source\
```

```
.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 get 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, I am 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 *SpeedLimitBolt.java* class which implements the *BaseBasicBolt* interface.

```
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 line by line again.

- *execute():* This is the method where you implement the business logic of your bolt. In this case, I am splitting 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 FileSourceSpout. It declares that it is going to return Ignite tuple for further processing.

> **Note:**
>
> The tuple name `IGNITE` is important here, the StormStreamer will only process the tuple with name `ignite`.

Note that, Bolts can do anything, for example, 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 file named *SpeedViolationTopology.java* in the `src\main\java\com\blu\imdg\storm\topology` directory. Use the following as the contents of the file:

```
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 empt\
y");
            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 Fiel\
ds("trafficLog"));
        // set ignite bolt
        builder.setBolt("ignite-bolt", stormStreamer, STORM_EXECUTORS).shuffleGrouping("li\
mit");
        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().getResourceAsStrea\
m("ignite-storm.properties");
        try {
            properties.load(ins);
        } catch (IOException e) {
            e.printStackTrace();
```

```
            properties = null;
        }
        return properties;
    }
}
```

Let's go through line by line again. First, we read the *ignite-strom.properties* file to get all the necessary parameters to configure the StormStreamer bolt next. The storm topology is basically a Thrift structure. The *TopologyBuilder* class provides a simple and elegant way to build complex Storm topology. The TopologyBuilder class has methods to 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 submit 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 shutdown the cluster using *shutdown* method of *LocalCluster*.

Step 7:

Next, run a local node of Apache Ignite or cluster first. After building the maven project, use the following command to run the topology locally.

```
mvn compile exec:java -Dstorm.topology=com.blu.imdg.storm.topology.SpeedViolationTopology
```

The application will produce a lot of system logs as follows.

```
16886 [Thread-12-limit] INFO  c.b.i.s.b.SpeedLimitBolt - Speed violation found:BC 123 speed:170
16886 [Thread-12-limit] INFO  b.s.util - Async loop interrupted!
16886 [com.blu.imdg.storm.topology.SpeedViolationTopology.main()] INFO  b.s.d.executor - Shut down executor limit:[3 3]
16886 [com.blu.imdg.storm.topology.SpeedViolationTopology.main()] INFO  b.s.d.executor - Shutting down executor __acker:[1 1]
16886 [Thread-13-disruptor-executor[1 1]-send-queue] INFO  b.s.util - Async loop interrupted!
16886 [Thread-14-__acker] INFO  b.s.util - Async loop interrupted!
16893 [com.blu.imdg.storm.topology.SpeedViolationTopology.main()] INFO  b.s.d.executor - Shut down executor __acker:[1 1]
16893 [com.blu.imdg.storm.topology.SpeedViolationTopology.main()] INFO  b.s.d.executor - Shutting down executor __system:[-1 -1]
16893 [Thread-16-__system] INFO  b.s.util - Async loop interrupted!
16893 [Thread-15-disruptor-executor[-1 -1]-send-queue] INFO  b.s.util - Async loop interrupted!
16894 [com.blu.imdg.storm.topology.SpeedViolationTopology.main()] INFO  b.s.d.executor - Shut down executor __system:[-1 -1]
16894 [com.blu.imdg.storm.topology.SpeedViolationTopology.main()] INFO  b.s.d.executor - Shutting down executor spout:[4 4]
16895 [Thread-18-spout] INFO  b.s.util - Async loop interrupted!
```

Figure 6.25

Now, if we verify the Ignite cache through *ignitevisior*, we should get the following output into the console.

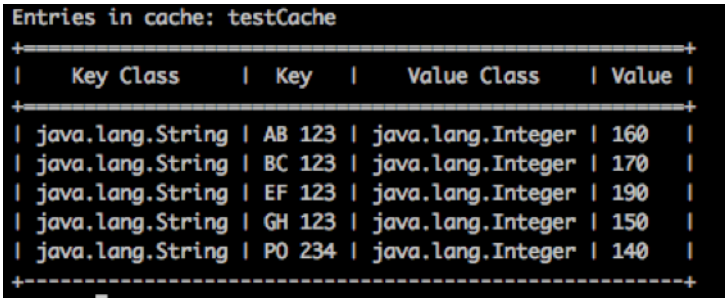


Figure 6.26

The output shows the result, what we 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.