

Apache Ignite™

# Table of Contents

Basic Concepts .....	1
What is Ignite .....	2
Getting Started .....	2
Maven Setup .....	9
Ignite Life Cycle .....	11
Asynchronous Support.....	13
Clients and Servers .....	15
Performance Tips.....	21
Clustering .....	29
Cluster.....	30
Cluster Groups .....	32
Leader Election .....	38
Cluster Configuration .....	40
Zero Deployment .....	48
Amazon AWS Integration .....	50
Google Cloud Integration .....	52
JClouds Integration .....	53
Network Configuration .....	55
Docker Deployment.....	59
Mesos Deployment .....	67
YARN Deployment .....	75
SSL\TLS.....	78
Data Grid .....	82
Data Grid .....	83
JCache and Beyond .....	85
Cache Modes .....	88
Primary & Backup Copies .....	94
Near Caches.....	97
Cache Queries.....	98
SQL Queries.....	103
Continuous Queries.....	114
Transactions .....	116
Off-Heap Memory .....	121
Affinity Collocation .....	126
Persistent Store .....	129
Automatic Persistence .....	143

Data Loading.....	158
Eviction Policies.....	163
Expiry Policies .....	167
Data Rebalancing.....	168
Web Session Clustering .....	172
Hibernate L2 Cache.....	179
JDBC Driver .....	186
Spring Caching .....	189
Topology Validation .....	193
Interactive SQL.....	195
Ignite with Apache Zeppelin.....	196
Streaming & CEP.....	204
Streaming & CEP .....	205
Data Streamers.....	206
Sliding Windows .....	210
Word Count Example .....	213
JMS Data Streamer .....	217
Distributed Data Structures.....	220
Queue and Set.....	221
Atomic Types .....	224
CountDownLatch.....	227
ID Generator .....	227
Memcached .....	230
Memcached Support.....	231
PHP .....	231
Java .....	232
Python .....	233
Ruby .....	233
Compute Grid .....	235
Compute Grid .....	236
Distributed Closures .....	237
Executor Service .....	243
MapReduce & ForkJoin.....	244
Per-Node Shared State .....	250
Collocate Compute and Data.....	252
Fault Tolerance .....	253
Load Balancing.....	256
Checkpointing.....	258

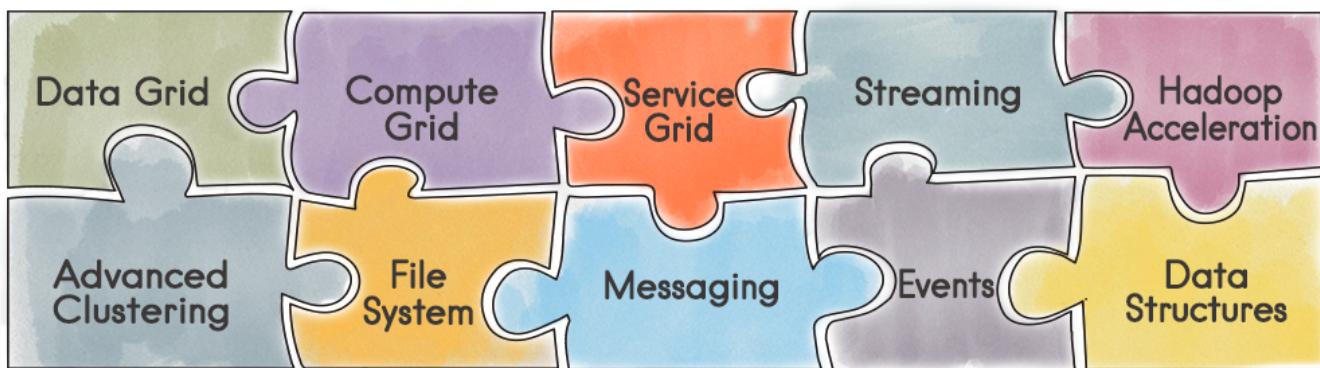
Job Scheduling .....	266
Task Deployment .....	269
Service Grid .....	276
Service Grid.....	277
Service Example.....	278
Cluster Singletons .....	282
Service Configuration.....	284
Distributed Messaging .....	287
Topic Based .....	288
Distributed Events .....	291
Local and Remote Events.....	292
Automatic Batching.....	295
HTTP .....	297
Rest API .....	298
Configuration .....	336
In-Memory File System.....	340
IGFS Native Ignite APIs .....	341
IGFS as Hadoop FileSystem.....	342
Hadoop FileSystem Cache .....	347
IGFS Modes .....	348
Hadoop Accelerator.....	350
Hadoop Accelerator .....	351
MapReduce .....	352
Installing on Apache Hadoop .....	355
Installing on Cloudera CDH.....	358
Installing on Hortonworks HDP.....	361
Ignite and Apache Hive .....	364
Spark Shared RDDs .....	367
Shared RDD Overview .....	368
IgniteRDD & IgniteContext .....	369
Installation & Deployment .....	371
Test Ignite with Spark-shell.....	372
Troubleshooting.....	376
Legal .....	377
Apache License .....	378
Copyright .....	382

# **Basic Concepts**

# What is Ignite

## In-Memory Data Fabric

Apache Ignite™ In-Memory Data Fabric is a high-performance, integrated and distributed in-memory platform for computing and transacting on large-scale data sets in real-time, orders of magnitude faster than possible with traditional disk-based or flash-based technologies.



## Features

You can view Ignite as a collection of independent, well-integrated, in-memory components geared to improve performance and scalability of your application. Some of these components include:

- Cluster
- Data Grid
- Streaming & CEP
- Compute Grid
- Service Grid
- IGFS Native Ignite APIs
- Queue and Set
- Topic Based
- Local and Remote Events
- Hadoop Accelerator
- Spark Shared RDDs

## Getting Started

This page will help you get started with Apache Ignite. You'll be up and running in a jiffy!

## Prerequisites

Apache Ignite was officially tested on:

Name	Value
JDK	Oracle JDK 7 and above
OS	Network
Linux (any flavor), Mac OSX (10.6 and up) Windows (XP and up), Windows Server (2008 and up)	No restrictions (10G recommended)
Hardware	No restrictions

## Installation

Here is the quick summary on installation of Apache Ignite:

- Download Apache Ignite as ZIP archive from <https://ignite.apache.org/>
- Unzip ZIP archive into the installation folder in your system
- Set `IGNITE_HOME` environment variable to point to the installation folder and make sure there is no trailing `/` in the path (this step is optional)

## Building From Source

If you downloaded the source package, you can build the binary using the following commands:

```
#Unpack the source package
$ unzip -q apache-ignite-1.3.0-incubating-src.zip
$ cd apache-ignite-1.3.0-incubating-src

#Build In-Memory Data Fabric release (without LGPL dependencies)
$ mvn clean package -DskipTests

#Build In-Memory Data Fabric release (with LGPL dependencies)
$ mvn clean package -DskipTests -Prelease,lgpl

#Build In-Memory Hadoop Accelerator release
#(optionally specify version of hadoop to use)
$ mvn clean package -DskipTests -Dignite.edition=hadoop [-Dhadoop.version=X.X.X]
```

## Start From Command Line

---

An Ignite node can be started from command line either with default configuration or by passing a configuration file. You can start as many nodes as you like and they will all automatically discover each other.

### With Default Configuration

To start a grid node with default configuration, open the command shell and, assuming you are in `IGNITE_HOME` (Ignite installation folder), just type this:

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

and you will see the output similar to this:

```
[02:49:12] Ignite node started OK (id=ab5d18a6)
[02:49:12] Topology snapshot [ver=1, nodes=1, CPUs=8, heap=1.0GB]
```

By default `ignite.sh` starts Ignite node with the default configuration: [config/default-config.xml](#).

### Passing Configuration File

To pass configuration file explicitly, from command line, you can type `ignite.sh <path to configuration file>` from within your Ignite installation folder. For example:

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

Path to configuration file can be absolute, or relative to either `IGNITE_HOME` (Ignite installation folder) or `META-INF` folder in your classpath.



**Interactive Mode.** To pick a configuration file in interactive mode just pass `-i` flag, like so: `ignite.sh -i`.

### Get It With Maven

---

Another easy way to get started with Apache Ignite in your project is to use Maven 2 dependency management.

Ignite requires only one `ignite-core` mandatory dependency. Usually you will also need to add `ignite-spring` for spring-based XML configuration and `ignite-indexing` for SQL querying.

Replace \${ignite-version} with actual Ignite version.

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



**Maven Setup.** See [Maven Setup](/docs/maven-setup) for more information on how to include individual Ignite maven artifacts.

## First Ignite Compute Application

Let's write our first grid application which will count a number of non-white-space characters in a sentence. As an example, we will take a sentence, split it into multiple words, and have every compute job count number of characters in each individual word. At the end we simply add up results received from individual jobs to get our total count.

*compute*

```
try (Ignite ignite = Ignition.start("examples/config/example-ignite.xml")) {
    Collection<IgniteCallable<Integer>> calls = new ArrayList<>();

    // Iterate through all the words in the sentence and create Callable jobs.
    for (final String word : "Count characters using callable".split(" "))
        calls.add(word::length);

    // Execute collection of Callables on the grid.
    Collection<Integer> res = ignite.compute().call(calls);

    // Add up all the results.
    int sum = res.stream().mapToInt(Integer::intValue).sum();

    System.out.println("Total number of characters is '" + sum + "'.");
}
```

*java7 compute*

```
try (Ignite ignite = Ignition.start("examples/config/example-ignite.xml")) {
    Collection<IgniteCallable<Integer>> calls = new ArrayList<>();

    // Iterate through all the words in the sentence and create Callable jobs.
    for (final String word : "Count characters using callable".split(" ")) {
        calls.add(new IgniteCallable<Integer>() {
            @Override public Integer call() throws Exception {
                return word.length();
            }
        });
    }

    // Execute collection of Callables on the grid.
    Collection<Integer> res = ignite.compute().call(calls);

    int sum = 0;

    // Add up individual word lengths received from remote nodes.
    for (int len : res)
        sum += len;

    System.out.println(">>> Total number of characters in the phrase is '" + sum + "'.");
}
```



**Zero Deployment.** Note that because of [Zero Deployment](#) feature, when running the above application from your IDE, remote nodes will execute received jobs without explicit deployment.

## First Ignite Data Grid Application

Now let's write a simple set of mini-examples which will put and get values to/from distributed cache, and perform basic transactions.

Since we are using cache in this example, we should make sure that it is configured. Let's use example configuration shipped with Ignite that already has several caches configured:

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

### Put and Get

```
try (Ignite ignite = Ignition.start("examples/config/example-ignite.xml")) {
    IgniteCache<Integer, String> cache = ignite.getOrCreateCache("myCacheName");

    // Store keys in cache (values will end up on different cache nodes).
    for (int i = 0; i < 10; i++)
        cache.put(i, Integer.toString(i));

    for (int i = 0; i < 10; i++)
        System.out.println("Got [key=" + i + ", val=" + cache.get(i) + ']');
}
```

## Atomic Operations

```
// Put-if-absent which returns previous value.  
Integer oldVal = cache.getAndPutIfAbsent("Hello", 11);  
  
// Put-if-absent which returns boolean success flag.  
boolean success = cache.putIfAbsent("World", 22);  
  
// Replace-if-exists operation (opposite of getAndPutIfAbsent), returns previous value.  
oldVal = cache.getAndReplace("Hello", 11);  
  
// Replace-if-exists operation (opposite of putIfAbsent), returns boolean success flag.  
success = cache.replace("World", 22);  
  
// Replace-if-matches operation.  
success = cache.replace("World", 2, 22);  
  
// Remove-if-matches operation.  
success = cache.remove("Hello", 1);
```

## Transactions

```
try (Transaction tx = ignite.transactions().txStart()) {  
    Integer hello = cache.get("Hello");  
  
    if (hello == 1)  
        cache.put("Hello", 11);  
  
    cache.put("World", 22);  
  
    tx.commit();  
}
```

## Distributed Locks

```
// Lock cache key "Hello".  
Lock lock = cache.lock("Hello");  
  
lock.lock();  
  
try {  
    cache.put("Hello", 11);  
    cache.put("World", 22);  
}  
finally {  
    lock.unlock();  
}
```

## Ignite Visor Admin Console

The easiest way to examine the content of the data grid as well as perform a long list of other management and monitoring operations is to use Ignite Visor Command Line Utility.

To start Visor simply run:

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

## Maven Setup

Apache Ignite is composed of multiple maven modules.

If you are using Maven to manage dependencies of your project, you can import individual Ignite modules a la carte.



In the examples below, please replace  `${ignite.version}`  with actual Apache Ignite version you are interested in.

## Common Dependencies

Ignite data fabric comes with one mandatory dependency on `ignite-core.jar`.

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

However, in many cases you may wish to have more dependencies, for example, if you want to use Spring configuration or SQL queries.

Here are the most commonly used optional modules:

- ignite-indexing (optional, add if you need SQL indexing)
- ignite-spring (optional, add if you plan to use Spring configuration)

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

## Importing Individual Modules A La Carte

You can import Ignite modules a la carte, one by one. The only required module is **ignite-core**, all others are optional. All optional modules can be imported just like the core module, but with different artifact IDs.

The following modules are available:

- **ignite-spring** (for Spring-based configuration support)
- **ignite-indexing** (for SQL querying and indexing)
- **ignite-geospatial** (for geospatial indexing)
- **ignite-hibernate** (for Hibernate integration)

- `ignite-web` (for Web Sessions Clustering)
- `ignite-schedule` (for Cron-based task scheduling)
- `ignite-log4j` (for Log4j logging)
- `ignite-jcl` (for Apache Commons logging)
- `ignite-jta` (for XA integration)
- `ignite-hadoop2-integration` (Integration with HDFS 2.0)
- `ignite-rest-http` (for HTTP REST messages)
- `ignite-scala` (for Ignite Scala API)
- `ignite-slf4j` (for SLF4J logging)
- `ignite-ssh` (for starting grid nodes on remote machines)
- `ignite-urideploy` (for URI-based deployment)
- `ignite-aws` (for seamless cluster discovery on AWS S3)
- `ignite-aop` (for AOP-based grid-enabling)
- `ignite-visor-console` (open source command line management and monitoring tool)

## Ignite Life Cycle

Manage life cycle of your Ignite nodes.

---

Ignite is JVM-based. Single JVM represents one or more logical Ignite nodes (most of the time, however, a single JVM runs just one Ignite node). Throughout Ignite documentation we use term Ignite runtime and Ignite node almost interchangeably. For example, when we say that you can "run 5 nodes on this host" - in most cases it technically means that you can start 5 JVMs on this host each running a single Ignite node. Ignite also supports multiple Ignite nodes in a single JVM. In fact, that is exactly how most of the internal tests run for Ignite itself.



Ignite runtime == JVM process == Ignite node (in most cases)

## Ignition Class

---

The `Ignition` class starts individual Ignite nodes in the network topology. Note that a physical server (like a computer on the network) can have multiple Ignite nodes running on it.

Here is how you can start grid node locally with all defaults

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

or by passing a configuration file:

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

Path to configuration file can be absolute, or relative to either `IGNITE_HOME` (Ignite installation folder) or `META-INF` folder in your classpath.

## LifecycleBean

Sometimes you need to perform certain actions before or after the Ignite node starts or stops. This can be done by implementing `LifecycleBean` interface, and specifying the implementation bean in `lifecycleBeans` property of `IgniteConfiguration` in the spring XML file:

```
<bean class="org.apache.ignite.IgniteConfiguration">
    ...
    <property name="lifecycleBeans">
        <list>
            <bean class="com.mycompany.MyLifecycleBean"/>
        </list>
    </property>
    ...
</bean>
```

`GridLifeCycleBean` can also configured programmatically the following way:

```
// Create new configuration.
IgniteConfiguration cfg = new IgniteConfiguration();

// Provide lifecycle bean to configuration.
cfg.setLifecycleBeans(new MyLifecycleBean());

// Start Ignite node with given configuration.
Ignite ignite = Ignition.start(cfg)
```

An implementation of `LifecycleBean` may look like the following:

```

public class MyLifecycleBean implements LifecycleBean {
    @Override public void onLifecycleEvent(LifecycleEventType evt) {
        if (evt == LifecycleEventType.BEFORE_NODE_START) {
            // Do something.
            ...
        }
    }
}

```

You can inject Ignite instance and other useful resources into a [LifecycleBean](#) implementation. Please refer to [Resource Injection](/docs/resource-injection) section for more information.

## Lifecycle Event Types

The following lifecycle event types are supported:

Event Type	Description
BEFORE_NODE_START	Invoked before Ignite node startup routine is initiated.
AFTER_NODE_START	Invoked right after Ignite node has started.
BEFORE_NODE_STOP	Invoked right before Ignite stop routine is initiated.
AFTER_NODE_STOP	Invoked right after Ignite node has stopped.

## Asynchronous Support

All distributed methods on all Ignite APIs can be executed either synchronously or asynchronously. However, instead of having a duplicate asynchronous method for every synchronous one (like [get\(\)](#) and [getAsync\(\)](#), or [put\(\)](#) and [putAsync\(\)](#), etc.), Ignite chose a more elegant approach, where methods don't have to be duplicated.

### IgniteAsyncSupport

[IgniteAsyncSupport](#) interface adds asynchronous mode to many Ignite APIs. For example, [IgniteCompute](#), [IgniteServices](#), [IgniteCache](#), and [IgniteTransactions](#) all extend [IgniteAsyncSupport](#) interface.

To enable asynchronous mode, you should call [withAsync\(\)](#) method which will return an instance of

the same API, but now with asynchronous behavior enabled.



**Method Return Values.** Note, that if async mode is enabled, actual synchronously returned values of methods should be ignored. The only way to obtain a return value from an asynchronous operation is from the `future()` method. ===== Compute Grid Example The example below illustrates the difference between synchronous and asynchronous computations.

### Synchronous

```
IgniteCompute compute = ignite.compute();

// Execute a job and wait for the result.
String res = compute.call(() -> {
    // Print hello world on some cluster node.
    System.out.println("Hello World");

    return "Hello World";
});
```

Here is how you would make the above invocation asynchronous:

### Asynchronous

```
// Enable asynchronous mode.
IgniteCompute asyncCompute = ignite.compute().withAsync();

// Asynchronously execute a job.
asyncCompute.call(() -> {
    // Print hello world on some cluster node and wait for completion.
    System.out.println("Hello World");

    return "Hello World";
});

// Get the future for the above invocation.
IgniteFuture<String> fut = asyncCompute.future();

// Asynchronously listen for completion and print out the result.
fut.listen(f -> System.out.println("Job result: " + f.get()));
```

## Data Grid Example

Here is the data grid example for synchronous and asynchronous invocations.

## Synchronous

```
IgniteCache<String, Integer> cache = ignite.cache("mycache");

// Synchronously store value in cache and get previous value.
Integer val = cache.getAndPut("1", 1);
```

Here is how you would make the above invocation asynchronous.

## Asynchronous

```
// Enable asynchronous mode.
IgniteCache<String, Integer> asyncCache = ignite.cache("mycache").withAsync();

// Asynchronously store value in cache.
asyncCache.getAndPut("1", 1);

// Get future for the above invocation.
IgniteFuture<Integer> fut = asyncCache.future();

// Asynchronously listen for the operation to complete.
fut.listen(f -> System.out.println("Previous cache value: " + f.get()));
```

## @IgniteAsyncSupported

Not every method on Ignite APIs is distributed and therefore does not really require asynchronous mode. To avoid confusion about which method is distributed, i.e. can be asynchronous, and which is not, all distributed methods in Ignite are annotated with `@IgniteAsyncSupported` annotation.



Note that, although not really needed, in async mode you can still get the future for non-distributed operations as well. However, this future will always be completed.

## Clients and Servers

Automatically distinguish between client and server nodes.

Ignite has an optional notion of **client** and **server** nodes. Server nodes participate in caching, compute execution, stream processing, etc., while the native client nodes provide ability to connect to the servers remotely. Ignite native clients allow to use the whole set of [Ignite APIs](#), including near caching, transactions, compute, streaming, services, etc. from the client side.

By default, all Ignite nodes are started as **server** nodes, and **client** mode needs to be explicitly enabled.

## Configuring Clients and Servers

You can configure a node to be either a client or a server via `IgniteConfiguration.setClientMode(...)` property.

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

```
IgniteConfiguration cfg = new IgniteConfiguration();

// Enable client mode.
cfg.setClientMode(true);

// Start Ignite in client mode.
Ignite ignite = Ignition.start(cfg);
```

Alternatively, for convenience, you can also enable or disable the client mode on the `Ignition` class itself, to allow clients and servers reuse the same configuration.

```
Ignition.setClientMode(true);

// Start Ignite in client mode.
Ignite ignite = Ignition.start();
```

## Creating Distributed Caches

Whenever creating caches in Ignite, either in XML or via any of the `Ignite.createCache(...)` or `Ignite.getOrCreateCache(...)` methods, Ignite will automatically deploy the distributed cache on all server nodes.



Once a distributed cache is created, it will be automatically deployed on all the existing and future **server** nodes.

```

// Enable client mode locally.
Ignition.setClientMode(true);

// Start Ignite in client mode.
Ignite ignite = Ignition.start();

CacheConfiguration cfg = new CacheConfiguration("myCache");

// Set required cache configuration properties.
...

// Create cache on all the existing and future server nodes.
// Note that since the local node is a client, it will not
// be caching any data.
IgniteCache<?, ?> cache = ignite.getOrCreateCache(cfg);

```

## Computing on Clients or Servers

By default `IgniteCompute` will execute jobs on all the server nodes. However, you can choose to execute jobs only on server nodes or only on client nodes by creating a corresponding cluster group.

### *Compute on Servers*

```

IgniteCompute compute = ignite.compute();

// Execute computation on the server nodes (default behavior).
compute.broadcast(() -> System.out.println("Hello Server"));

```

### *Compute on Clients*

```

ClusterGroup clientGroup = ignite.cluster().forClients();

IgniteCompute clientCompute = ignite.compute(clientGroup);

// Execute computation on the client nodes.
clientCompute.broadcast(() -> System.out.println("Hello Client"));

```

## Managing Slow Clients

In many deployments client nodes are launched outside of the main cluster on slower machines with worse network. In these scenarios it is possible that servers will generate load (such as continuous

queries notification, for example) that clients will not be able to handle, resulting in growing queue of outbound messages on servers. This may eventually cause either out-of-memory situation on server or blocking the whole cluster if back-pressure control is enabled.

To manage these situations you can configure the maximum number of allowed outgoing messages for client nodes. If the size of outbound queue exceeds this value, such a client node will be disconnected from the cluster preventing global slowdown.

Examples below show how to configure slow client queue limit in code and XML configuration.

```
IgniteConfiguration cfg = new IgniteConfiguration();

// Configure Ignite here.

TcpCommunicationSpi commSpi = new TcpCommunicationSpi();
commSpi.setSlowClientQueueLimit(1000);

cfg.setCommunicationSpi(commSpi);
```

```
<bean id="grid.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    <!-- Configure Ignite here. -->

    <property name="communicationSpi">
        <bean class="org.apache.ignite.spi.communication.tcp.TcpCommunicationSpi">
            <property name="slowClientQueueLimit" value="1000"/>
        </bean>
    </property>
</bean>
```

## Client Reconnect

Client node can disconnect from cluster in several cases:

- in case of network problems when client can not re-establish connection with server
- connection with server was broken for some time, client is able to re-establish connection with server, but server already dropped client node since server did not receive client heartbeats
- slow clients can be disconnected by server

When client determines that it disconnected from cluster it assigns to a local node new ID and tries to reconnect to cluster. Note: this has side effect and 'id' property of local `ClusterNode` will change in case of client reconnection.

While client is in disconnected state and attempt to reconnect is in progress all Ignite API throws special exception: `IgniteClientDisconnectedException`, this exception provides future which will be completed when client finish reconnect (`IgniteCache` API throws `CacheException` which has `IgniteClientDisconnectedException` as its cause). This future also can be obtained using method `IgniteCluster.clientReconnectFuture()`.

Also there are special events for client reconnect (these events are local, i.e. they are fired only on client node):

- `EventType.EVT_CLIENT_NODE_DISCONNECTED`
- `EventType.EVT_CLIENT_NODE_RECONNECTED`

Below are examples showing work with `IgniteClientDisconnectedException`.

### Compute

```
IgniteCompute compute = ignite.compute();

while (true) {
    try {
        compute.run(job);
    }
    catch (IgniteClientDisconnectedException e) {
        e.reconnectFuture().get(); // Wait for reconnect.

        // Can proceed and use the same IgniteCompute instance.
    }
}
```

## Cache

```
IgniteCache cache = ignite.getOrCreateCache(new CacheConfiguration<>());  
  
while (true) {  
    try {  
        cache.put(key, val);  
    }  
    catch (CacheException e) {  
        if (e.getCause() instanceof IgniteClientDisconnectedException) {  
            IgniteClientDisconnectedException cause =  
                (IgniteClientDisconnectedException)e.getCause();  
  
            cause.reconnectFuture().get(); // Wait for reconnect.  
  
            // Can proceed and use the same IgniteCache instance.  
        }  
    }  
}
```

Automatic client reconnect can be disabled using 'clientReconnectDisabled' property on `TcpDiscoverySpi`, if reconnect is disabled then client node is stopped when client determines that it disconnected from cluster.

Example below show how to disable client reconnect.

```
IgniteConfiguration cfg = new IgniteConfiguration();  
  
// Configure Ignite here.  
  
TcpDiscoverySpi discoverySpi = new TcpDiscoverySpi();  
  
discoverySpi.setClientReconnectDisabled(true);  
  
cfg.setDiscoverySpi(discoverySpi);
```

## Forcing Server Mode On Client Nodes

Client nodes will require alive server nodes in topology to start.

If it is a requirement to be able to start client node disregarding of server node presence you can force server mode discovery on client nodes this way:

```
IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setClientMode(true);

// Configure Ignite here.

TcpDiscoverySpi discoverySpi = new TcpDiscoverySpi();

discoverySpi.setForceServerMode(true);

cfg.setDiscoverySpi(discoverySpi);
```

In this case discovery will happen as if all nodes in topology were server nodes.



**Important Notice.** In this case all addresses discovery SPI uses on all nodes should be mutually reachable in order for discovery to work properly.

## Performance Tips

Simple cache configuration tips to optimize your cache performance.

Ignite In-Memory Data Grid performance and throughput vastly depends on the features and the settings you use. In almost any use case the cache performance can be optimized by simply tweaking the cache configuration.

### Disable Internal Events Notification

Ignite has rich event system to notify users about various events, including cache modification, eviction, compaction, topology changes, and a lot more. Since thousands of events per second are generated, it creates an additional load on the system. This can lead to significant performance degradation. Therefore, it is highly recommended to enable only those events that your application logic requires. By default, event notifications are disabled.

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <!-- Enable only some events and leave other ones disabled. -->
    <property name="includeEventTypes">
        <list>
            <util:constant static-field=
"org.apache.ignite.events.EventType.EVT_TASK_STARTED"/>
            <util:constant static-field=
"org.apache.ignite.events.EventType.EVT_TASK_FINISHED"/>
            <util:constant static-field=
"org.apache.ignite.events.EventType.EVT_TASK_FAILED"/>
        </list>
    </property>
    ...
</bean>

```

## Tune Cache Start Size

In terms of size and capacity, Ignite's internal cache map acts exactly like a normal Java HashMap: it has some initial capacity (which is pretty small by default), which doubles as data arrives. The process of internal cache map resizing is CPU-intensive and time-consuming, and if you load a huge dataset into cache (which is a normal use case), the map will have to resize a lot of times. To avoid that, you can specify the initial cache map capacity, comparable to the expected size of your dataset. This will save a lot of CPU resources during the load time, because the map won't have to resize. For example, if you expect to load 100 million entries into cache, you can use the following configuration:

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            ...
            <!-- Set initial cache capacity to ~ 100M. -->
            <property name="startSize" value="#{100 * 1024 * 1024}"/>
            ...
        </bean>
    </property>
</bean>

```

The above configuration will save you from  $\log_2(100,000,000) - \log_2(1024) \approx 16$  cache map resizes (1024 is an initial map capacity by default). Remember, that each subsequent resize will be on average 2 times longer than the previous one.

## Turn Off Backups

If you use **PARTITIONED** cache, and the data loss is not critical for you (for example, when you have a backing cache store), consider disabling backups for **PARTITIONED** cache. When backups are enabled, the cache engine has to maintain a remote copy of each entry, which requires network exchange and is time-consuming. To disable backups, use the following configuration:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            ...
            <!-- Set cache mode. -->
            <property name="cacheMode" value="PARTITIONED"/>
            <!-- Set number of backups to 0-->
            <property name="backups" value="0"/>
            ...
        </bean>
    </property>
</bean>
```



**Possible Data Loss.** If you don't have backups enabled for **PARTITIONED** cache, you will lose all entries cached on a failed node. It may be acceptable for caching temporary data or data that can be otherwise recreated. Make sure that such data loss is not critical for application before disabling backups.

## Tune Off-Heap Memory

If you plan to allocate large amounts of memory to your JVM for data caching (usually more than 10GB of memory), then your application will most likely suffer from prolonged lock-the-world GC pauses which can significantly hurt latencies. To avoid GC pauses use off-heap memory to cache data - essentially your data is still cached in memory, but JVM does not know about it and GC is not affected. To enable off-heap storage with unlimited size, use the following configuration:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            ...
            <!-- Enable off-heap storage with unlimited size. -->
            <property name="offHeapMaxMemory" value="0"/>
            ...
        </bean>
    </property>
</bean>
```

## Disable Swap Storage

Swap storage is disabled by default. However, in your configuration it might be enabled. If it is, keep in mind that using swap storage can significantly hurt performance. To disable swap storage explicitly, use the following configuration:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            ...
            <!-- Disable swap. -->
            <property name="swapEnabled" value="false"/>
            ...
        </bean>
    </property>
</bean>
```

## Disable Peer Class Loading

While peer class loading is very convenient in development, it does carry a certain overhead and should be turned off in production. To disable peer class loading, use the following configuration snippet:

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            ...
            <!-- Explicitly disable peer class loading. -->
            <property name="peerClassLoadingEnabled" value="false"/>
            ...
        </bean>
    </property>
</bean>

```

## Tune Eviction Policy

Evictions are disabled by default. If you do need to use evictions to make sure that data in cache does not overgrow beyond allowed memory limits, consider choosing the proper eviction policy. An example of setting the LRU eviction policy with maximum size of 100000 entries is shown below:

```

<bean class="org.apache.ignite.cache.CacheConfiguration">
    ...
    <property name="evictionPolicy">
        <!-- LRU eviction policy. -->
        <bean class="org.apache.ignite.cache.eviction.lru.LruEvictionPolicy">
            <!-- Set the maximum cache size to 1 million (default is 100,000). -->
            <property name="maxSize" value="1000000"/>
        </bean>
    </property>
    ...
</bean>

```

Regardless of which eviction policy you use, cache performance will depend on the maximum amount of entries in cache allowed by eviction policy - if cache size overgrows this limit, the evictions start to occur.

## Tune Cache Data Rebalancing

When a new node joins topology, existing nodes relinquish primary or back up ownership of some keys to the new node so that keys remain equally balanced across the grid at all times. This may require additional resources and hit cache performance. To tackle this possible problem, consider tweaking the following parameters:

- Configure rebalance batch size, appropriate for your network. Default is 512KB which means that by default rebalance messages will be about 512KB. However, you may need to set this value to be higher or lower based on your network performance.
- Configure rebalance throttling to unload the CPU. If your data sets are large and there are a lot of messages to send, the CPU or network can get over-consumed, which consecutively may slow down the application performance. In this case you should enable data rebalance throttling which helps tune the amount of time to wait between rebalance messages to make sure that rebalancing process does not have any negative performance impact. Note that application will continue to work properly while rebalancing is still in progress.
- Configure rebalance thread pool size. As opposite to previous point, sometimes you may need to make rebalancing faster by engaging more CPU cores. This can be done by increasing the number of threads in rebalance thread pool (by default, there are only 2 threads in pool).

Below is an example of setting all of the above parameters in cache configuration:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set rebalance batch size to 1 MB. -->
            <property name="rebalanceBatchSize" value="#{1024 * 1024}"/>

            <!-- Explicitly disable rebalance throttling. -->
            <property name="rebalanceThrottle" value="0"/>

            <!-- Set 4 threads for rebalancing. -->
            <property name="rebalanceThreadPoolSize" value="4"/>
            ...
        </bean>
    </property>
</bean>
```

## Configure Thread Pools

By default, Ignite has its main thread pool size set to the 2 times the available CPU count. In most cases keeping 2 threads per core will result in faster application performance, since there will be less context switching and CPU caches will work better. However, if you are expecting that your jobs will block for I/O or any other reason, it may make sense to increase the thread pool size. Below is an examples of how you configure thread pools:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <!-- Configure internal thread pool. -->
    <property name="publicThreadPoolSize" value="64"/>

    <!-- Configure system thread pool. -->
    <property name="systemThreadPoolSize" value="32"/>
    ...
</bean>
```

## Use Externalizable Whenever Possible

It is a very good practice to have every object that is transferred over network implement `java.io.Externalizable`. These may be cache keys or values, jobs, job arguments, or anything else that will be sent across network to other grid nodes. Implementing `Externalizable` may sometimes result in over 10x performance boost over standard serialization.

## Use Collocated Computations

Ignite enables you to execute MapReduce computations in memory. However, most computations usually work on some data which is cached on remote grid nodes. Loading that data from remote nodes is very expensive in most cases and it is a lot more cheaper to send the computation to the node where the data is. The easiest way to do it is to use `IgniteCompute.affinityRun()` method or `@CacheAffinityMapped` annotation. There are other ways, including `Affinity.mapKeysToNodes()` methods. The topic of collocated computations is covered in much detail in [Affinity Collocation](#), which contains proper code examples.

## Use Data Streamer

If you need to upload lots of data into cache, use `IgniteDataStreamer` to do it. Data streamer will properly batch the updates prior to sending them to remote nodes and will properly control number of parallel operations taking place on each node to avoid thrashing. Generally it provides performance of 10x than doing a bunch of single-threaded updates. See [Data Loading](#) section for more detailed description and examples.

## Batch Up Your Messages

If you can send 10 bigger jobs instead of 100 smaller jobs, you should always choose to send bigger

jobs. This will reduce the amount of jobs going across the network and may significantly improve performance. The same regards cache entries - always try to use API methods, that take collections of keys or values, instead of passing them one-by-one.

## Tune Garbage Collection

If you are seeing spikes in your throughput due to Garbage Collection (GC), then you should tune JVM settings. The following JVM settings have proven to provide fairly smooth throughput without large spikes:

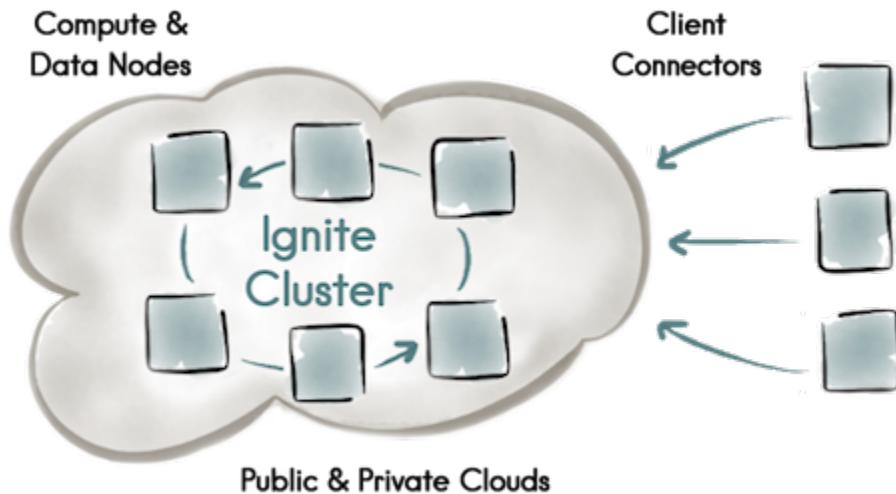
```
-XX:+UseParNewGC  
-XX:+UseConcMarkSweepGC  
-XX:+UseTLAB  
-XX:NewSize=128m  
-XX:MaxNewSize=128m  
-XX:MaxTenuringThreshold=0  
-XX:SurvivorRatio=1024  
-XX:+UseCMSInitiatingOccupancyOnly  
-XX:CMSInitiatingOccupancyFraction=60
```

# Clustering

# Cluster

Ignite has advanced clustering capabilities including logical cluster groups and auto-discovery.

Ignite nodes can automatically discover each other. This helps to scale the cluster when needed, without having to restart the whole cluster. Developers can also leverage from Ignite's hybrid cloud support that allows establishing connection between private cloud and public clouds such as Amazon Web Services, providing them with best of both worlds.



## Features

- Pluggable Design via [IgniteDiscoverySpi](#)
- Dynamic topology management
- Automatic discovery on LAN, WAN, and AWS
- On-demand and direct deployment
- Support for virtual clusters and node groupings

## IgniteCluster

Cluster functionality is provided via [IgniteCluster](#) interface. You can get an instance of [IgniteCluster](#) from [Ignite](#) as follows:

```
Ignite ignite = Ignition.ignite();
IgniteCluster cluster = ignite.cluster();
```

Through [IgniteCluster](#) interface you can:

- Start and stop remote cluster nodes
- Get a list of all cluster members
- Create logical [Cluster Groups](#)

## ClusterNode

The [ClusterNode](#) interface has very concise API and deals only with the node as a logical network endpoint in the topology: its globally unique ID, the node metrics, its static attributes set by the user and a few other parameters.

## Cluster Node Attributes

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:

```
<bean class="org.apache.ignite.IgniteConfiguration">
    ...
    <property name="userAttributes">
        <map>
            <entry key="ROLE" value="worker"/>
        </map>
    </property>
    ...
</bean>
```

Following example shows how to get the nodes where "worker" attribute has been set.

```
ClusterGroup workers = ignite.cluster().forAttribute("ROLE", "worker");
Collection<ClusterNode> nodes = workers.nodes();
```



All node attributes are available via [ClusterNode.attribute\("propertyName"\)](#) method.

## Cluster Node Metrics

Ignite automatically collects metrics for all cluster nodes. Metrics are collected in the background and

are updated with every heartbeat message exchanged between cluster nodes.

Node metrics are available via [ClusterMetrics](#) interface which contains over 50 various metrics (note that the same metrics are available for [Cluster Groups](#)).

Here is an example of getting some metrics, including average CPU load and used heap, for the local node:

```
// Local Ignite node.  
ClusterNode localNode = cluster.localNode();  
  
// Node metrics.  
ClusterMetrics metrics = localNode.metrics();  
  
// Get some metric values.  
double cpuLoad = metrics.getCurrentCpuLoad();  
long usedHeap = metrics.getHeapMemoryUsed();  
int numberOfcores = metrics.getTotalCpus();  
int activeJobs = metrics.getCurrentActiveJobs();
```

## Local Cluster Node

Local grid node is an instance of the [ClusterNode](#) representing **this** Ignite node.

Here is an example of how to get a local node:

```
ClusterNode localNode = ignite.cluster().localNode();
```

## Cluster Groups

Easily create logical groups of cluster nodes within your cluster.

[ClusterGroup](#) represents a logical grouping of cluster nodes.

In Ignite all nodes are equal by design, so you don't have to start any nodes in specific order, or assign any specific roles to them. However, Ignite allows users to logically group cluster nodes for any application specific purpose. For example, you may wish to deploy a service only on remote nodes, or assign a role of "worker" to some worker nodes for job execution.



Note that `IgniteCluster` interface is also a cluster group which includes all nodes in the cluster. You can limit job execution, service deployment, messaging, events, and other tasks to run only within some cluster group. For example, here is how to broadcast a job only to remote nodes (excluding the local node).

*broadcast*

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

IgniteCluster cluster = ignite.cluster();

// Get compute instance which will only execute
// over remote nodes, i.e. not this node.
IgniteCompute compute = ignite.compute(cluster.forRemotes());

// Broadcast to all remote nodes and print the ID of the node
// on which this closure is executing.
compute.broadcast(() -> System.out.println("Hello Node: " + ignite.cluster().localNode()
.id()));
```

*java7 broadcast*

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

IgniteCluster cluster = ignite.cluster();

// Get compute instance which will only execute
// over remote nodes, i.e. not this node.
IgniteCompute compute = ignite.compute(cluster.forRemotes());

// Broadcast closure only to remote nodes.
compute.broadcast(new IgniteRunnable() {
    @Override public void run() {
        // Print ID of the node on which this runnable is executing.
        System.out.println(">>> Hello Node: " + ignite.cluster().localNode().id());
    }
})
```

## Predefined Cluster Groups

You can create cluster groups based on any predicate. For convenience Ignite comes with some predefined cluster groups.

Here are examples of some cluster groups available on `ClusterGroup` interface.

## *Remote Nodes*

```
IgniteCluster cluster = ignite.cluster();

// Cluster group with remote nodes, i.e. other than this node.
ClusterGroup remoteGroup = cluster.forRemotes();
```

## *Cache Nodes*

```
IgniteCluster cluster = ignite.cluster();

// All nodes on which cache with name "myCache" is deployed,
// either in client or server mode.
ClusterGroup cacheGroup = cluster.forCache("myCache");

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

// All client nodes that access "myCache".
ClusterGroup clientGroup = cluster.forClientNodes("myCache");
```

## *Nodes With Attributes*

```
IgniteCluster cluster = ignite.cluster();

// All nodes with attribute "ROLE" equal to "worker".
ClusterGroup attrGroup = cluster.forName("ROLE", "worker");
```

## *Random Node*

```
IgniteCluster cluster = ignite.cluster();

// Cluster group containing one random node.
ClusterGroup randomGroup = cluster.forRandom();

// First (and only) node in the random group.
ClusterNode randomNode = randomGroup.node();
```

## *Host Nodes*

```
IgniteCluster cluster = ignite.cluster();

// Pick random node.
ClusterGroup randomNode = cluster.forRandom();

// All nodes on the same physical host as the random node.
ClusterGroup cacheNodes = cluster.forHost(randomNode);
```

## *Oldest Node*

```
IgniteCluster cluster = ignite.cluster();

// Dynamic cluster group representing the oldest cluster node.
// Will automatically shift to the next oldest, if the oldest
// node crashes.
ClusterGroup oldestNode = cluster.forOldest();
```

## *Local Node*

```
IgniteCluster cluster = ignite.cluster();

// Cluster group with only this (local) node in it.
ClusterGroup localGroup = cluster.forLocal();

// Local node.
ClusterNode localNode = localGroup.node();
```

## *Clients & Servers*

```
IgniteCluster cluster = ignite.cluster();

// All client nodes.
ClusterGroup clientGroup = cluster.forClients();

// All server nodes.
ClusterGroup serverGroup = cluster.forServers();
```

## **Cluster Groups with Node Attributes**

The unique characteristic of Ignite is that all grid nodes are equal. There are no master or server nodes, and there are no worker or client nodes either. All nodes are equal from Ignite's point of view -

however, users can configure nodes to be masters and workers, or clients 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:

```
<bean class="org.apache.ignite.IgniteConfiguration">
    ...
    <property name="userAttributes">
        <map>
            <entry key="ROLE" value="worker"/>
        </map>
    </property>
    ...
</bean>
```

```
IgniteConfiguration cfg = new IgniteConfiguration();
Map<String, String> attrs = Collections.singletonMap("ROLE", "worker");
cfg.setUserAttributes(attrs);

// Start Ignite node.
Ignite ignite = Ignition.start(cfg);
```



All environment variables and system properties are automatically registered as node attributes on startup.



Node attributes are available via `ClusterNode.attribute("propertyName")` method. Following example shows how to get the nodes where "worker" attribute has been set.

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

## Custom Cluster Groups

You can define dynamic cluster groups based on some predicate. 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 50% CPU utilization. Note that the nodes in this group will change over time based on their CPU load.

#### *custom group*

```
IgniteCluster cluster = ignite.cluster();

// Nodes with less than 50% CPU load.
ClusterGroup readyNodes = cluster.forPredicate((node) -> node.metrics().
getCurrentCpuLoad() < 0.5);
```

#### *java7 custom group*

```
IgniteCluster cluster = ignite.cluster();

// Nodes with less than 50% CPU load.
ClusterGroup readyNodes = cluster.forPredicate(
    new IgnitePredicate<ClusterNode>() {
        @Override public boolean apply(ClusterNode node) {
            return node.metrics().getCurrentCpuLoad() < 0.5;
        }
    }
);
```

## Combining Cluster Groups

You can combine cluster groups by nesting them within each other. For example, the following code snippet shows how to get a random remote node by combining remote group with random group.

```
// Group containing oldest node out of remote nodes.
ClusterGroup oldestGroup = cluster.forRemotes().forOldest();

ClusterNode oldestNode = oldestGroup.node();
```

## Getting Nodes from Cluster Groups

You can get to various cluster group nodes as follows:

```

ClusterGroup remoteGroup = cluster.forRemotes();

// All cluster nodes in the group.
Collection<ClusterNode> grpNodes = remoteGroup.nodes();

// First node in the group (useful for groups with one node).
ClusterNode node = remoteGroup.node();

// And if you know a node ID, get node by ID.
UUID myID = ...;

node = remoteGroup.node(myId);

```

## Cluster Group Metrics

Ignite automatically collects metrics about all the cluster nodes. The cool thing about cluster groups is that it automatically aggregates the metrics across all the nodes in the group and provides proper averages, mins, and maxes within the group.

Group metrics are available via [ClusterMetrics](#) interface which contains over 50 various metrics (note that the same metrics are available for individual cluster nodes as well).

Here is an example of getting some metrics, including average CPU load and used heap, across all remote nodes:

```

// Cluster group with remote nodes, i.e. other than this node.
ClusterGroup remoteGroup = ignite.cluster().forRemotes();

// Cluster group metrics.
ClusterMetrics metrics = remoteGroup.metrics();

// Get some metric values.
double cpuLoad = metrics.getCurrentCpuLoad();
long usedHeap = metrics.getHeapMemoryUsed();
int numberOfcores = metrics.getTotalCpus();
int activeJobs = metrics.getCurrentActiveJobs();

```

## Leader Election

Automatically select oldest or youngest cluster nodes.

When working in distributed environments, sometimes you need to have a guarantee that you always will pick the same node, regardless of the cluster topology changes. Such nodes are usually called **leaders**.

In many systems electing cluster leaders usually has to do with data consistency and is generally handled via collecting votes from cluster members. Since in Ignite the data consistency is handled by data grid affinity function (e.g. [Rendezvous Hashing](#), picking leaders in traditional sense for data consistency outside of the data grid is not really needed.

However, you may still wish to have a **coordinator** node for certain tasks. For this purpose, Ignite lets you automatically always pick either oldest or youngest nodes in the cluster.



**Use Service Grid.** Note that for most **leader** or **singleton-like** use cases, it is recommended to use the **Service Grid** functionality, as it allows to automatically deploy various [Cluster Singletons](#) and is usually easier to use.

## Oldest Node

Oldest node has a property that it remains constant whenever new nodes are added. The only time when the oldest node in the cluster changes is when it leaves the cluster or crashes.

Here is an example of how to select [\[undefined\]](#) with only the oldest node in it.

```
IgniteCluster cluster = ignite.cluster();

// Dynamic cluster group representing the oldest cluster node.
// Will automatically shift to the next oldest, if the oldest
// node crashes.
ClusterGroup oldestNode = cluster.forOldest();
```

## Youngest Node

Youngest node, unlike the oldest node, constantly changes every time a new node joins a cluster. However, sometimes it may still become handy, especially if you need to execute some task only on the newly joined node.

Here is an example of how to select [Cluster Groups](#) with only the youngest node in it.

```
igniteCluster cluster = ignite.cluster();

// Dynamic cluster group representing the youngest cluster node.
// Will automatically shift to the next oldest, if the oldest
// node crashes.
ClusterGroup youngestNode = cluster.forYoungest();
```



Once the cluster group is obtained, you can use it for executing tasks, deploying services, sending messages, and more.

## Cluster Configuration

Deploy Ignite in any environment with pluggable automatic node discovery.

In Ignite, nodes can discover each other by using [DiscoverySpi](#). Ignite provides [TcpDiscoverySpi](#) as a default implementation of [DiscoverySpi](#) that uses TCP/IP for node discovery. Discovery SPI can be configured for Multicast and Static IP based node discovery.

### Multicast Based Discovery

[TcpDiscoveryMulticastIpFinder](#) uses Multicast to discover other nodes in the grid and is the default IP finder. You should not have to specify it unless you plan to override default settings. Here is an example of how to configure this finder via Spring XML file or programmatically from Java:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="discoverySpi">
        <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
            <property name="ipFinder">
                <bean class=
                    "org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
                    <property name="multicastGroup" value="228.10.10.157"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

```
TcpDiscoverySpi spi = new TcpDiscoverySpi();

TcpDiscoveryVmIpFinder ipFinder = new TcpDiscoveryMulticastIpFinder();

ipFinder.setMulticastGroup("228.10.10.157");

spi.setIpFinder(ipFinder);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default discovery SPI.
cfg.setDiscoverySpi(spi);

// Start Ignite node.
Ignition.start(cfg);
```

## Static IP Based Discovery

For cases when Multicast is disabled, [TcpDiscoveryVmIpFinder](#) should be used with pre-configured list of IP addresses. You are only required to provide at least one IP address of a remote node, but usually it is advisable to provide 2 or 3 addresses of the grid nodes that you plan to start first for redundancy. Once a connection to any of the provided IP addresses is established, Ignite will automatically discover all other grid nodes.



You do not need to specify IP addresses for all Ignite nodes, only for a couple of nodes you plan to start first.

Here is an example of how to configure this finder via Spring XML file or programmatically from Java:

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="discoverySpi">
        <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
            <property name="ipFinder">
                <bean class=
"org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDiscoveryVmIpFinder">
                    <property name="addresses">
                        <list>
                            <value>1.2.3.4</value>

                            <!--
                                IP Address and optional port range.
                                You can also optionally specify an individual port.
                                -->
                            <value>1.2.3.5:47500..47509</value>
                        </list>
                    </property>
                </bean>
            </property>
        </bean>
    </property>
</bean>

```

```

TcpDiscoverySpi spi = new TcpDiscoverySpi();

TcpDiscoveryVmIpFinder ipFinder = new TcpDiscoveryVmIpFinder();

// Set initial IP addresses.
// Note that you can optionally specify a port or a port range.
ipFinder.setAddresses(Arrays.asList("1.2.3.4", "1.2.3.5:47500..47509"));

spi.setIpFinder(ipFinder);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default discovery SPI.
cfg.setDiscoverySpi(spi);

// Start Ignite node.
Ignition.start(cfg);

```



By default the `TcpDiscoveryVmIpFinder` is used in `non-shared` mode. In this mode the list of IP addresses should also contain the address of the local node, so the node can become the 1st node in the cluster, in case if other remote nodes have not been started yet.

## Multicast and Static IP Based Discovery

You can use both, Multicast and Static IP based discovery together. In this case, in addition to addresses received via multicast, if any, `TcpDiscoveryMulticastIpFinder` can also work with pre-configured list of static IP addresses, just like Static IP-Based Discovery described above. Here is an example of how to configure Multicast IP finder with static IP addresses:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="discoverySpi">
        <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
            <property name="ipFinder">
                <bean class=
"org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
                    <property name="multicastGroup" value="228.10.10.157"/>

                    <!-- list of static IP addresses-->
                    <property name="addresses">
                        <list>
                            <value>1.2.3.4</value>

                            <!--
                                IP Address and optional port range.
                                You can also optionally specify an individual port.
                            -->
                            <value>1.2.3.5:47500..47509</value>
                        </list>
                    </property>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

```

TcpDiscoverySpi spi = new TcpDiscoverySpi();

TcpDiscoveryVmIpFinder ipFinder = new TcpDiscoveryMulticastIpFinder();

// Set Multicast group.
ipFinder.setMulticastGroup("228.10.10.157");

// Set initial IP addresses.
// Note that you can optionally specify a port or a port range.
ipFinder.setAddresses(Arrays.asList("1.2.3.4", "1.2.3.5:47500..47509"));

spi.setIpFinder(ipFinder);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default discovery SPI.
cfg.setDiscoverySpi(spi);

// Start Ignite node.
Ignition.start(cfg);

```

## Apache jclouds Based Discovery

---

Refer to [JClouds Integration](#) documentation.

## Amazon S3 Based Discovery

---

Refer to [Amazon AWS Integration](#) documentation.

## Google Cloud Storage Based Discovery

---

Refer to [Google Cloud Integration](#) documentation.

## JDBC Based Discovery

---

You can have your database be a common shared storage of initial IP addresses. In this nodes will write their IP addresses to a database on startup. This is done via [TcpDiscoveryJdbcIpFinder](#).

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="discoverySpi">
        <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
            <property name="ipFinder">
                <bean class=
"org.apache.ignite.spi.discovery.tcp.ipfinder.jdbc.TcpDiscoveryJdbcIpFinder">
                    <property name="dataSource" ref="ds"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>

<!-- Configured data source instance. -->
<bean id="ds" class="some.Datasource">
    ...
</bean>
```

```

TcpDiscoverySpi spi = new TcpDiscoverySpi();

// Configure your DataSource.
DataSource someDs = MySampleDataSource(...);

TcpDiscoveryJdbcIpFinder ipFinder = new TcpDiscoveryJdbcIpFinder();

ipFinder.setDataSource(someDs);

spi.setIpFinder(ipFinder);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default discovery SPI.
cfg.setDiscoverySpi(spi);

// Start Ignite node.
Ignition.start(cfg);
```

## Failure Detection Timeout

Failure detection timeout is used to determine how long a cluster node should wait before considering a remote connection with other node failed. This timeout is the easiest way to tune discovery SPI's failure detection feature depending on the network and hardware conditions of your cluster.

The timeout automatically controls such configuration parameters of `TcpDiscoverySpi` as socket timeout, message acknowledgment timeout and others. If any of these parameters is set explicitly, then the failure timeout setting will be ignored. Failure detection timeout is configured with `IgniteConfiguration.setFailureDetectionTimeout(long failureDetectionTimeout)` method. Default value, that is equal to 10 seconds, is chosen in a way to make it possible for discovery SPI to work reliably on most of hardware and virtual deployments, but this has made failure detection time worse. However, for stable low-latency networks the parameter may be set to ~200 milliseconds in order to detect and react on failures quicker.



## Configuration

Following configuration parameters can be optionally configured on `TcpDiscoverySpi`.

<code>setIpFinder(TcpDiscoveryIpFinder)</code>	<b>IP finder that is used to share info about nodes IP addresses.</b>	<b>TcpDiscoveryMulticastIpFinder</b> <b>Provided implementations can be used:</b> <code>TcpDiscoverySharedFsIpFinder</code> <code>TcpDiscoveryS3IpFinder</code> <code>TcpDiscoveryJdbcIpFinder</code> <code>TcpDiscoveryVmIpFinder</code>
Setter Method	Description	Default
Default		<code>setLocalAddress(String)</code>
Sets local host IP address that discovery SPI uses.		If not provided, by default a first found non-loopback address will be used. If there is no non-loopback address available, then <code>java.net.InetAddress.getLocalHost()</code> will be used.
<code>setLocalPort(int)</code>	Port the SPI listens to.	47500
	<code>setLocalPortRange(int)</code>	Local port range. Local node will try to bind on first available port starting from local port up until local port + local port range.
100	100	<code>setHeartbeatFrequency(long)</code>

<code>setIpFinder(TcpDiscoveryIpFinder)</code>	<b>IP finder that is used to share info about nodes IP addresses.</b>	<code>TcpDiscoveryMulticastIpFinder</code> <b>Provided implementations can be used:</b> <code>TcpDiscoverySharedFsIpFinder</code> <code>TcpDiscoveryS3IpFinder</code> <code>TcpDiscoveryJdbcIpFinder</code> <code>TcpDiscoveryVmIpFinder</code>
Delay in milliseconds between heartbeat issuing of heartbeat messages. SPI sends messages in configurable time interval to other nodes to notify them about its state.	2000	2000
<code>setMaxMissedHeartbeats(int)</code>	Number of heartbeat requests that could be missed before local node initiates status check.	1
1	<code>setReconnectCount(int)</code>	Number of times node tries to (re)establish connection to another node.
2	2	<code>setNetworkTimeout(long)</code>
Sets maximum network timeout in milliseconds to use for network operations.	5000	5000
<code>setSocketTimeout(long)</code>	Sets socket operations timeout. This timeout is used to limit connection time and write-to-socket time.	2000
2000	<code>setAckTimeout(long)</code>	Sets timeout for receiving acknowledgement for sent message. If acknowledgement is not received within this timeout, sending is considered as failed and SPI tries to repeat message sending.
2000	2000	<code>setJoinTimeout(long)</code>

<code>setIpFinder(TcpDiscoveryIpFinder)</code>	<b>IP finder that is used to share info about nodes IP addresses.</b>	<code>TcpDiscoveryMulticastIpFinder</code> Provided implementations can be used: <code>TcpDiscoverySharedFsIpFinder</code> <code>TcpDiscoveryS3IpFinder</code> <code>TcpDiscoveryJdbcIpFinder</code> <code>TcpDiscoveryVmIpFinder</code>
Sets join timeout. If non-shared IP finder is used and node fails to connect to any address from IP finder, node keeps trying to join within this timeout. If all addresses are still unresponsive, exception is thrown and node startup fails. 0 means wait forever.	0	0
<code>setThreadPriority(int)</code>	Thread priority for threads started by SPI.	0
0	<code>setStatisticsPrintFrequency(int)</code>	Statistics print frequency in milliseconds. 0 indicates that no print is required. If value is greater than 0 and log is not quiet then stats are printed out with INFO level once a period. This may be very helpful for tracing topology problems.
true	true	

## Zero Deployment

---

The closures and tasks that you use for your computations may be of any custom class, including anonymous classes. In Ignite, the remote nodes will automatically become aware of those classes, and you won't need to explicitly deploy or move any .jar files to any remote nodes.

Such behavior is possible due to peer class loading (P2P class loading), a special **distributed ClassLoader** in Ignite for inter-node byte-code exchange. With peer-class-loading enabled, you don't have to manually deploy your Java or Scala code on each node in the grid and re-deploy it each time it changes.

A code example like below would run on all remote nodes due to peer class loading, without any explicit deployment step.

```

IgniteCluster cluster = ignite.cluster();

// Compute instance over remote nodes.
IgniteCompute compute = ignite.compute(cluster.forRemotes());

// Print hello message on all remote nodes.
compute.broadcast(() -> System.out.println("Hello node: " + cluster.localNode().id()));

```

Here is how peer class loading can be configured:

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <!-- Explicitly enable peer class loading. -->
    <property name="peerClassLoadingEnabled" value="true"/>
    ...
</bean>

```

```

IgniteConfiguration cfg = new IgniteConfiguration();
cfg.setPeerClassLoadingEnabled(true);

// Start Ignite node.
Ignite ignite = Ignition.start(cfg);

```

Peer class loading sequence works as follows: 1. Ignite will check if class is available on local classpath (i.e. if it was loaded at system startup), and if it was, it will be returned. No class loading from a peer node will take place in this case. 2. If class is not locally available, then a request will be sent to the originating node to provide class definition. Originating node will send class byte-code definition and the class will be loaded on the worker node. This happens only once per class - once class definition is loaded on a node, it will never have to be loaded again.



**Development vs Production.** It is recommended that peer-class-loading is disabled in production. Generally you want to have a controlled production environment without any magic. To deploy your classes explicitly, you can copy them into Ignite **libs** folder or manually add them to the classpath on every node.



**Auto-Clearing Caches for Hot Redeployment.** Whenever you change class definitions for the data stored in cache, Ignite will automatically clear the caches for previous class definitions before peer-deploying the new data to avoid class-loading conflicts.



**3rd Party Libraries.** When utilizing peer class loading, you should be aware of the libraries that get loaded from peer nodes vs. libraries that are already available locally in the class path. Our suggestion is to include all 3rd party libraries into class path of every node. This can be achieved by copying your JAR files into the Ignite **libs** folder. This way you will not transfer megabytes of 3rd party classes to remote nodes every time you change a line of code.

## Explicit Deployment

To deploy your JAR files into Ignite explicitly, you should copy them into the **libs** folder on every Ignite cluster member. Ignite will automatically load all the JAR files from the **libs** folder on startup.

## Amazon AWS Integration

Automatically discover cluster nodes on Amazon EC2 cloud.

Node discovery on AWS cloud is usually proven to be more challenging. Amazon EC2, just like most of the other virtual environments, has the following limitations:

- Multicast is disabled.
- TCP addresses change every time a new image is started.

Although you can use TCP-based discovery in the absence of the Multicast, you still have to deal with constantly changing IP addresses and constantly updating the configuration. This creates a major inconvenience and makes configurations based on static IPs virtually unusable in such environments.

## Amazon S3 Based Discovery

To mitigate constantly changing IP addresses problem, Ignite supports automatic node discovery by utilizing S3 store via [TcpDiscoveryS3IpFinder](#). On startup nodes register their IP addresses with Amazon S3 store. This way other nodes can try to connect to any of the IP addresses stored in S3 and initiate automatic grid node discovery.



Such approach allows to create your configuration once and reuse it for all EC2 instances.

Here is an example of how to configure Amazon S3 IP finder:

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="discoverySpi">
        <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
            <property name="ipFinder">
                <bean class=
"org.apache.ignite.spi.discovery.tcp.ipfinder.s3.TcpDiscoveryS3IpFinder">
                    <property name="awsCredentials" ref="aws.creds"/>
                    <property name="bucketName" value="YOUR_BUCKET_NAME"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>

<!-- AWS credentials. Provide your access key ID and secret access key. -->
<bean id="aws.creds" class="com.amazonaws.auth.BasicAWSCredentials">
    <constructor-arg value="YOUR_ACCESS_KEY_ID" />
    <constructor-arg value="YOUR_SECRET_ACCESS_KEY" />
</bean>

```

```

TcpDiscoverySpi spi = new TcpDiscoverySpi();

BasicAWSCredentials creds = new BasicAWSCredentials("yourAccessKey", "yourSecretKey");

TcpDiscoveryS3IpFinder ipFinder = new TcpDiscoveryS3IpFinder();

ipFinder.setAwsCredentials(creds);

spi.setIpFinder(ipFinder);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default discovery SPI.
cfg.setDiscoverySpi(spi);

// Start Ignite node.
Ignition.start(cfg);

```



Refer to [Cluster Configuration](#) for more information on various cluster configuration properties.

# Google Cloud Integration

Automatically discover cluster nodes on Google Compute Engine cluster.

Nodes discovery on Google Compute Engine is usually proven to be more challenging. Google Cloud, just like most of the other virtual environments, has the following limitations:

- Multicast is disabled;
- TCP addresses change every time a new image is started.

Although you can use TCP-based discovery in the absence of the Multicast, you still have to deal with constantly changing IP addresses and constantly updating the configuration. This creates a major inconvenience and makes configurations based on static IPs virtually unusable in such environments.

## Google Cloud Storage Based Discovery

To mitigate constantly changing IP addresses problem, Ignite supports automatic node discovery by utilizing Google Cloud Storage store via [TcpDiscoveryGoogleStorageIpFinder](#). On startup each node registers its IP address in the storage. This way other nodes can try to connect to any address stored in the storage and initiate automatic grid node discovery.



Such approach allows to create your configuration once and reuse it for all Google Compute Engine instances. Here is an example of how to configure Google Cloud Storage based IP finder:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="discoverySpi">
        <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
            <property name="ipFinder">
                <bean class=
"org.apache.ignite.spi.discovery.tcp.ipfinder.gce.TcpDiscoveryGoogleStorageIpFinder">
                    <property name="projectName" ref="YOUR_GOOGLE_PLATFORM_PROJECT_NAME"/>
                    <property name="bucketName" value="YOUR_BUCKET_NAME"/>
                    <property name="serviceAccountId" value="YOUR_SERVICE_ACCOUNT_ID"/>
                    <property name="serviceAccountP12FilePath" value="PATH_TO_YOUR_PKCS12_KEY"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

```

TcpDiscoverySpi spi = new TcpDiscoverySpi();

TcpDiscoveryGoogleStorageIpFinder ipFinder = new TcpDiscoveryGoogleStorageIpFinder();

ipFinder.setServiceAccountId(yourServiceAccountId);
ipFinder.setServiceAccountP12FilePath(pathToYourP12Key);
ipFinder.setProjectName(yourGoogleCloudPlatformProjectName);

// Bucket name must be unique across the whole Google Cloud Platform.
ipFinder.setBucketName("your_bucket_name");

spi.setIpFinder(ipFinder);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default discovery SPI.
cfg.setDiscoverySpi(spi);

// Start Ignite node.
Ignition.start(cfg);

```



Refer to [Cluster Configuration](#) for more information on various cluster configuration properties.

## JClouds Integration

Cluster's nodes discovery on any cloud platform.

Nodes discovery on a cloud platform is usually proven to be more challenging because the most of such virtual environments, has the following limitations:

- Multicast is disabled;
- TCP addresses change every time a new image is started.

Although you can use TCP-based discovery in the absence of the Multicast, you still have to deal with constantly changing IP addresses and constantly updating the configuration. This creates a major inconvenience and makes configurations based on static IPs virtually unusable in such environments.

### Apache JClouds Based Discovery

To mitigate constantly changing IP addresses problem, Ignite supports automatic node discovery by

utilizing Apache jclouds multi-cloud toolkit via [TcpDiscoveryCloudIpFinder](#). For information about Apache jclouds please refer to [jclouds.apache.org](#).

The IP finder forms nodes addresses, that possibly running Apache Ignite, by getting private and public IP addresses of all virtual machines running on a cloud and adding a port number to them. The port is either the one that is set with 'TcpDiscoverySpi.setLocalPort(int)' or 'TcpDiscoverySpi.DFLT\_PORT' otherwise. This way all the nodes can try to connect to any formed IP address and initiate automatic grid node discovery.

Please refer to [Apache jclouds providers section](#) to get the list of supported cloud platforms.



All virtual machines must start Ignite instances on the same port, otherwise they will not be able to discover each other using this IP finder. Here is an example of how to configure Apache jclouds based IP finder:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="discoverySpi">
        <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
            <property name="ipFinder">
                <bean class=
"org.apache.ignite.spi.discovery.tcp.ipfinder.cloud.TcpDiscoveryCloudIpFinder"/>
                <!-- Configuration for Google Compute Engine. -->
                <property name="provider" value="google-compute-engine"/>
                <property name="identity" value="YOUR_SERVICE_ACCOUNT_EMAIL"/>
                <property name="credentialPath" value="PATH_YOUR_PEM_FILE"/>
                <property name="zones">
                    <list>
                        <value>us-central1-a</value>
                        <value>asia-east1-a</value>
                    </list>
                </property>
            </bean>
        </property>
    </bean>
</property>
</bean>
```

```

TcpDiscoverySpi spi = new TcpDiscoverySpi();

TcpDiscoveryCloudIpFinder ipFinder = new TcpDiscoveryCloudIpFinder();

// Configuration for AWS EC2.
ipFinder.setProvider("aws-ec2");
ipFinder.setIdentity(yourAccountId);
ipFinder.setCredential(yourAccountKey);
ipFinder.setRegions(Collections.<String>emptyList().add("us-east-1"));
ipFinder.setZones(Arrays.asList("us-east-1b", "us-east-1e"));

spi.setIpFinder(ipFinder);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default discovery SPI.
cfg.setDiscoverySpi(spi);

// Start Ignite node.
Ignition.start(cfg);

```



Refer to [Cluster Configuration](#) for more information on various cluster configuration properties.

## Network Configuration

Facilitates communication between nodes within the cluster.

[CommunicationSpi](#) provides basic plumbing to send and receive grid messages and is utilized for all distributed grid operations, such as task execution, monitoring data exchange, distributed event querying and others. Ignite provides [TcpCommunicationSpi](#) as the default implementation of [CommunicationSpi](#), that uses the TCP/IP to communicate with other nodes.

To enable communication with other nodes, [TcpCommunicationSpi](#) adds [TcpCommunicationSpi.ATTR\\_ADDRS](#) and [TcpCommunicationSpi.ATTR\\_PORT](#) local node attributes. At startup, this SPI tries to start listening to local port specified by [TcpCommunicationSpi.setLocalPort\(int\)](#) method. If local port is occupied, then SPI will automatically increment the port number until it can successfully bind for listening. [TcpCommunicationSpi.setLocalPortRange\(int\)](#) configuration parameter controls maximum number of ports that SPI will try before it fails.



**Local Port Range.** Port range comes very handy when starting multiple grid nodes on the same machine or even in the same VM. In this case all nodes can be brought up without a single change in configuration.

## Configuration

Following configuration parameters can be optionally configured on `TcpCommunicationSpi`:

Setter Method	Description	Default
<code>'setLocalAddress(String)'</code>	Sets local host address for socket binding.	Any available local IP address.
<code>setLocalPort(int)</code>	<code>setLocalPortRange(int)</code>	<code>setTcpNoDelay(boolean)</code>
<code>setConnectTimeout(long)</code>	<code>setIdleConnectionTimeout(long)</code>	<code>setBufferSizeRatio(double)</code>
<code>setMinimumBufferedMessageCount(int)</code>	<code>setDualSocketConnection(boolean)</code>	<code>setConnectionBufferSize(int)</code>
<code>setSelectorsCount(int)</code>	<code>setConnectionBufferFlushFrequency(long)</code>	<code>setDirectBuffer(boolean)</code>
<code>setDirectSendBuffer(boolean)</code>	<code>setAsyncSend(boolean)</code>	<code>setSharedMemoryPort(int)</code>
<code>setSocketReceiveBuffer(int)</code>	<code>setSocketSendBuffer(int)</code>	Sets local port for socket binding.
47100	Controls maximum number of local ports tried if all previously tried ports are occupied.	100
Sets value for <code>TCP_NODELAY</code> socket option. Each socket accepted or created will be using provided value. This should be set to true (default) for reducing request/response time during communication over TCP protocol. In most cases we do not recommend to change this option.	true	Sets connect timeout used when establishing connection with remote nodes.
1000	Sets maximum idle connection timeout upon which a connection to client will be closed.	30000

Setter Method	Description	Default
Sets the buffer size ratio for this SPI. As messages are sent, the buffer size is adjusted using this ratio.	0.8 or <code>IGNITE_COMMUNICATION_BUF_RESIZE_RATIO</code> system property value, if set.	Sets the minimum number of messages for this SPI, that are buffered prior to sending.
512 or <code>IGNITE_MIN_BUFFERED_COMMUNICATION_MSG_CNT</code> system property value, if set.	Sets flag indicating whether dual-socket connection between nodes should be enforced. If set to true, two separate connections will be established between communicating nodes: one for outgoing messages, and one for incoming. When set to false, single TCP connection will be used for both directions. This flag is useful on some operating systems, when <code>TCP_NODELAY</code> flag is disabled and messages take too long to get delivered.	false
This parameter is used only when <code>setAsyncSend(boolean)</code> is set to false. Sets connection buffer size for synchronous connections. Increase buffer size if using synchronous send and sending large amount of small sized messages. However, most of the time this should be set to 0 (default).	0	Sets the count of selectors to be used in TCP server.
Default count of selectors equals to the expression result - <code>Math.min(4, Runtime.getRuntime().availableProcessors())</code>	This parameter is used only when <code>setAsyncSend(boolean)</code> is set to false. Sets connection buffer flush frequency in milliseconds. This parameter makes sense only for synchronous send when connection buffer size is not 0. Buffer will be flushed once within specified period if there is no enough messages to make it flush automatically.	100

Setter Method	Description	Default
Switches between using NIO direct and NIO heap allocation buffers. Although direct buffers perform better, in some cases (especially on Windows) they may cause JVM crashes. If that happens in your environment, set this property to false.	true	Switches between using NIO direct and NIO heap allocation buffers usage for message sending in asynchronous mode.
false	Switches between synchronous and asynchronous message sending. This should be set to true (default) if grid nodes send large amount of data over network from multiple threads, however this maybe environment and application specific and we recommend to benchmark the application in both modes.	true
Sets port which will be used by <a href="#">IpcSharedMemoryServerEndpoint</a> . Nodes started on the same host will communicate over IPC shared memory (only for Linux and MacOS hosts). Set this to -1 to disable IPC shared memory communication.	48100	Sets receive buffer size for sockets created or accepted by this SPI. If not provided, default is 0 which leaves buffer unchanged after socket creation (i.e. uses Operating System default value).
0	Sets send buffer size for sockets created or accepted by this SPI. If not provided, default is 0 which leaves the buffer unchanged after socket creation (i.e. uses Operating System default value).	0

## Example

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="communicationSpi">
        <bean class="org.apache.ignite.spi.communication.tcp.TcpCommunicationSpi">
            <!-- Override local port. -->
            <property name="localPort" value="4321"/>
        </bean>
    </property>
    ...
</bean>

```

```

TcpCommunicationSpi commSpi = new TcpCommunicationSpi();

// Override local port.
commSpi.setLocalPort(4321);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default communication SPI.
cfg.setCommunicationSpi(commSpi);

// Start grid.
Ignition.start(cfg);

```

## Docker Deployment

Deploy Ignite within Docker containers

Docker allows to package Ignite deployment with all the dependencies into a standard container. Docker automates downloading the Ignite release, deploying users' code into Ignite, and configuring nodes. It also automatically starts up a fully configured Ignite node.

Ignite docker container can run it two modes:

### Start from User Git Repository

Ignite docker container will start in this mode if the `GIT_REPO` parameter is configured. In this case, the container will build user's project specified by the GIT repository and will start Ignite with user code deployed in it. Such integration allows users to deploy new code by simply restarting the Ignite docker container.

To pull the Ignite docker container use the following command:

```
sudo docker pull apacheignite/ignite-docker
```

To run Ignite docker container using `docker run`:

```
sudo docker run -it --net=host  
-e "GIT_REPO=$GIT_REPO"  
[-e "GIT_BRANCH=$GIT_BRANCH"]  
[-e "BUILD_CMD=$BUILD_CMD"]  
...  
apacheignite/ignite-docker
```

The configuration parameters are passed through environment variables in docker container. The following configuration parameters are available:

Name	Description	Optional	Default	Example
GIT_REPO	GIT_BRANCH	BUILD_CMD	URL to the GIT repository.	GIT branch to build.
Command which will be used to build the GIT project.	false	true	true	N/A
master	mvn clean package	https://github.com/bob/ignite-pojo	sprint-1	mvn clean package \ -DskipTests=true
true	IGNITE_CONFIG	URL to the Ignite configuration file (can also be relative to the META-INF folder on the class path). The downloaded config file will be saved to ./ignite-config.xml	N/A	https://raw.githubusercontent.com/bob/master/ignite-cfg.xml
LIB_PATTERN	If set then Ignite docker container will only copy the files which match this regex pattern.	true	copy all jar files from target folder	OPTION_LIBS

Name	Description	Optional	Default	Example
Ignite optional libs which will be included in the class path.	true	ignite-log4j,\ignite-spring,\ignite-indexing	ignite-aws,ignite-aop	libs.*

## Example

To run Ignite docker container from GIT repository, execute the following command:

```
sudo docker run -it --net=host
-e "GIT_REPO=https://github.com/TikhonovNikolay/docker-example.git"
apacheignite/ignite-docker
```

You should see the following print out in the logs:

```
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-ssh/ignite-ssh-1.1.2.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-ssh/jsch-0.1.50.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-ssh/licenses/apache-2.0.txt
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-ssh/licenses/jcraft-revised-bsd.txt
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/README.txt
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/commons-codec-1.6.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/ignite-urideploy-1.1.2.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/jtidy-r938.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/licenses/apache-2.0.txt
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/licenses/jtidy-mit-license.txt
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/spring-aop-4.1.0.RELEASE.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/spring-beans-4.1.0.RELEASE.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/spring-context-4.1.0.RELEASE.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/spring-core-4.1.0.RELEASE.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/spring-expression-4.1.0.RELEASE.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-urideploy/spring-tx-4.1.0.RELEASE.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-web/README.txt
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-web/ignite-web-1.1.2.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-web/javax.servlet-api-3.0.1.jar
inflating: ignite/gridgain-community-fabric-1.1.2/libs/optional/ignite-web/licenses/apache-2.0.txt
[14:44:35]
[14:44:35]   / \   / \   / / \ / \   / \   /
[14:44:35]   / \ / \ ( ) / \ / \ / \ / \ / \ / \
[14:44:35]   / \ / \ / \ / \ / \ / \ / \ / \ / \
[14:44:35]
[14:44:35] ver. 1.1.2#20150616-sha1:d1a21501
[14:44:35] 2015 Copyright(C) Apache Software Foundation
[14:44:35]
[14:44:35] Quiet mode.
[14:44:35]   ^-- Logging to file '/home/ignite_home/ignite/gridgain-community-fabric-1.1.2/work/log/ignite-1eeab765.log'
[14:44:35]   ^-- To see **FULL** console log here add -DIGNITE_QUIET=false or "-v" to ignite.{sh|bat}
[14:44:35]
[14:44:35] Configured plugins:
[14:44:35]   ^-- None
[14:44:35]
[14:44:37] To start Console Management & Monitoring run ignitevisorcmd.{sh|bat}
[14:44:37]
[14:44:37] Ignite node started OK (id=1eeab765)
[14:44:37] Topology snapshot [ver=1, nodes=1, CPUs=1, heap=1.0GB]
[14:44:45] New version is available at http://www.gridgain.com/download/editions: 1.1.4
```

## Start Bare Ignite Node

If `GIT_REPO` parameter is not configured, then Ignite docker container will download the specified or the latest Ignite distribution. By default the latest version is downloaded.

To pull the Ignite docker container use the following command:

```
sudo docker pull apacheignite/ignite-docker
```

To run Ignite docker container using `docker run`:

```
sudo docker run -it --net=host  
-e "IGNITE_VERSION=$IGNITE_VERSION"  
[-e "IGNITE_CONFIG=$IGNITE_CONFIG"]  
[-e "OPTION_LIBS=$OPTION_LIBS"]  
...  
apacheignite/ignite-docker
```

The configuration parameters are passed through environment variables in docker container. The following configuration parameters are available:

Name	Description	Default	Example
<code>IGNITE_URL</code>	<code>IGNITE_VERSION</code>	<code>IGNITE_SOURCE</code>	URL to the Ignite distribution.
Ignite version.	Ignite edition which will be downloaded. This parameter is ignored, if <code>IGNITE_VERSION</code> is set.	N/A	<code>latest</code>
<code>COMMUNITY</code>	<a href="http://apache-mirror.rbc.ru/pub/apache/incubator/ignite/1.1.0/apache-ignite-fabric-1.1.0-incubating-bin.zip">http://apache-mirror.rbc.ru/pub/apache/incubator/ignite/1.1.0/apache-ignite-fabric-1.1.0-incubating-bin.zip</a>	1.1.4	APACHE
<code>IGNITE_CONFIG</code>	URL to the Ignite configuration file (can also be relative to the META-INF folder on the class path). The downloaded config file will be saved to <code>./ignite-config.xml</code>	N/A	<a href="https://raw.githubusercontent.com/bob/master/ignite-cfg.xml">https://raw.githubusercontent.com/bob/master/ignite-cfg.xml</a>

Name	Description	Default	Example
OPTION_LIBS	Ignite optional libs which will be included in the class path.	ignite-log4j, ignite-spring, ignite-indexing	ignite-aws,ignite-aop

## Example

To run Ignite docker container with bare Ignite node, use the following command:

```
sudo docker run -it --net=host -e "IGNITE_VERSION=1.1.0" apacheignite/ignite-docker
```

Should see the following in logs:

```
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-ssh/ignite-ssh-1.1.0-incubating.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-ssh/jsch-0.1.50.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-ssh/licenses/apache-2.0.txt
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-ssh/licenses/jcraft-revised-bsd.txt
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/README.txt
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/commons-codec-1.6.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/ignite-urideploy-1.1.0-incubating.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/jtidy-r938.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/licenses/apache-2.0.txt
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/licenses/jtidy-mit-license.txt
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/spring-aop-4.1.0.RELEASE.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/spring-beans-4.1.0.RELEASE.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/spring-context-4.1.0.RELEASE.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/spring-core-4.1.0.RELEASE.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/spring-expression-4.1.0.RELEASE.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-urideploy/spring-tx-4.1.0.RELEASE.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-web/README.txt
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-web/ignite-web-1.1.0-incubating.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-web/javax.servlet-api-3.0.1.jar
inflating: ignite/apache-ignite-fabric-1.1.0-incubating-bin/libs/optional/ignite-web/licenses/apache-2.0.txt
[15:03:04]
[15:03:04]   /   /   /   |   /   /   /   /   /
[15:03:04]   -   /   (   7   )   /   /   /   /   /
[15:03:04]   /   \   /   /   |   /   /   /   /   /
[15:03:04]
[15:03:04] ver. 1.1.0-incubating#20150520-sha1:6da491f4
[15:03:04] 2015 Copyright(C) Apache Software Foundation
[15:03:04]
[15:03:04] Quiet mode.
[15:03:04]   ^-- Logging to file '/home/ec2-user/docker/ignite/apache-ignite-fabric-1.1.0-incubating-bin/work/log/ignite-3fc01526.log'
[15:03:04]   ^-- To see **FULL** console log here add -DIGNITE_QUIET=false or "-v" to ignite.{sh|bat}
[15:03:04]
[15:03:04] Configured plugins:
[15:03:04]   ^-- None
[15:03:04]
[15:03:05] To start Console Management & Monitoring run ignitevisorcmd.{sh|bat}
[15:03:05]
[15:03:05] Ignite node started OK (id=3fc01526)
[15:03:05] Topology snapshot [ver=1, nodes=1, CPUs=1, heap=1.0GB]
```

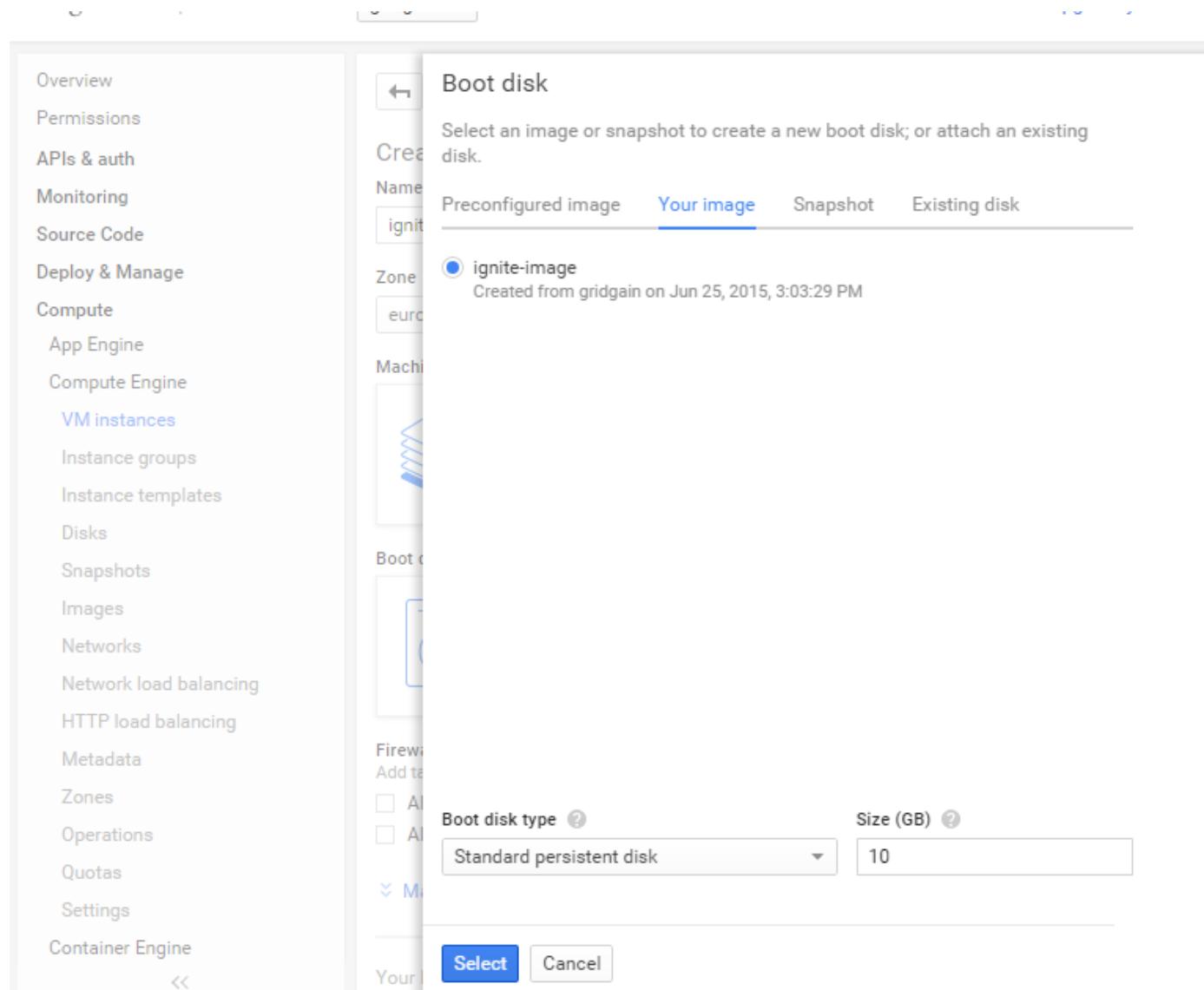
## Google Compute Deployment

1. To import the Ignite Image, execute the following command:

```
gcloud compute images  
  create <IMAGE_NAME>  
  --source-uri gs://ignite-media/ignite-google-image-1.0.0.tar.gz
```

For information please refer to [cloud.google.com](http://cloud.google.com)

1. Go to [Google Compute Console](#).
2. Go to [Compute](#) → [Compute Engine](#) → [VM instances](#) and click on [New instance](#).
3. Click on [Change](#) button in section [Boot disk](#).
4. Go to [Your image](#) and choose imported image. On the screenshot the image is with `ignite-name` name.



1. Click on [Management](#), [disk](#), [networking](#), [access & security options](#) and add any of the configuration parameters for the Ignite docker container.

Overview  
Permissions  
APIs & auth  
Monitoring  
Source Code  
Deploy & Manage  
Compute  
App Engine  
Compute Engine  
VM instances  
Instance groups  
Instance templates  
Disks  
Snapshots  
Images  
Networks  
Network load balancing  
HTTP load balancing  
Metadata  
Zones  
Operations  
Quotas  
Settings

**Metadata (Optional)**  
You can set custom metadata for an instance or project outside of the server-defined metadata. This is useful for passing in arbitrary values to your project or instance that can be queried by your code on the instance. [Learn more](#)

GIT_REPO	https://github.com/bob/ignite-pojo	X
GIT_BRANCH	sprint-1	X
<a href="#">+ Add item</a>		

**Availability policy**

**Preemptibility** ⓘ  
A preemptible VM costs much less, but lasts only 24 hours. It can be terminated sooner due to system demands. [Learn more](#)

Off (recommended)	▼
-------------------	---

**Automatic restart**  
Compute Engine can automatically restart VM instances if they are terminated for non-user-initiated reasons (maintenance event, hardware failure, software failure, etc.)

On (recommended)	▼
------------------	---

**On host maintenance**  
When Compute Engine performs periodic infrastructure maintenance it can migrate your VM instances to other hardware without downtime.

Migrate VM instance (recommended)	▼
-----------------------------------	---

[▲ Less](#)

1. Fill the required fields and run instances.
2. Connect to the instances.
3. To access the execution progress you need to know a `container id`. The following command will show the `container id`:

```
sudo docker ps
```

1. Show logs:

```
sudo docker logs -f CONTAINER_ID
```

1. Enter the docker container:

```
sudo docker exec -it container_id /bin/bash
```

## Amazon EC2 Deployment

1. Choose the required region and click on link in table below.

Region	US-WEST
US-EAST	EU-CENTRAL
ami-5b53a41f	ami-3fa45e54
ami-9c5f6481	Image

1. Choose an **Instance Type**.
2. Go to **Configure Instance** and expand **Advanced Details** section.
3. Add any of the configuration parameters for the Ignite docker container.

The screenshot shows the 'Configure Instance' step of an AWS EC2 instance creation wizard. It includes fields for IAM role, shutdown behavior, termination protection, monitoring, tenancy, and user data. The 'Advanced Details' section is expanded, showing user data input for a Docker container setup.

**IAM role:** None (dropdown) | Create new IAM role (button)

**Shutdown behavior:** Stop (dropdown)

**Enable termination protection:** Protect against accidental termination (checkbox)

**Monitoring:** Enable CloudWatch detailed monitoring (checkbox) | Additional charges apply.

**Tenancy:** Shared tenancy (multi-tenant hardware) (dropdown) | Additional charges will apply for dedicated tenancy.

**Advanced Details**

**User data:** As text (radio button selected) | As file (radio button) | Input is already base64 encoded (checkbox)

```
GIT_REPO=https://github.com/bob/ignite-pojo  
GIT_BRANCH=sprint-1
```

Cancel Previous Review and Create

1. On the Tag Instance, set the value for **Name** tag. For example `ignite-node`
2. Review and run instances.
3. Connect to the instances <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstances.html>
4. To access the execution progress, you need to know the **container id**. Use the following command:

```
sudo docker ps
```

## 1. Show logs:

```
sudo docker logs -f CONTAINER_ID
```

## 1. Enter the docker container:

```
sudo docker exec -it container_id /bin/bash
```

# Mesos Deployment

Deploy Ignite in Mesos cluster.

Apache Ignite Framework supports scheduling and running Apache Ignite nodes in a Mesos cluster. Apache Mesos is a cluster manager which provides a general runtime environment providing all the essentials to deploy, run and manage distributed applications. Its resource management and isolation helps getting the most out of servers. For information about Apache Mesos please refer to <http://mesos.apache.org/>

## Ignite Mesos Framework

Deploying Apache Ignite cluster typically involves downloading the Apache Ignite distribution, changing configuration settings and starting the nodes up. Apache Ignite Mesos Framework consists of **Scheduler** and **Task** and allows to greatly simplify cluster deployment.

- **Scheduler** registers itself at Mesos Master on scheduler startup. Once registration is successful the Scheduler will begin processing of resource requests from Mesos Master to utilize resources for Apache Ignite nodes. The Scheduler will maintain the Ignite cluster at desired total resources level (CPU, memory, etc.).
- **Task** - the entity that runs Ignite Node on slaves.

## Running Ignite Mesos Framework

For running Ignite Mesos Framework requires Apache Mesos Cluster configured and running. For information on how to set up a Mesos cluster please refer to <https://docs.mesosphere.com/getting-started/datacenter/install/>.



Make sure that masters and slaves node listen on correct ip addresses. In the other case there is no guarantee that Mesos Cluster will correctly work.

## Run the Framework via Marathon

Currently the recommended way to run the Framework is to run it via Marathon.

1. Install marathon. See <https://docs.mesosphere.com/getting-started/datacenter/install/> marathon section.
2. Download Apache Ignite and upload `libs\optional\ignite-mesos\ignite-mesos-<ignite-version>-jar-with-dependencies.jar` file to any cloud storage. (for example Amazon S3 storage and etc.).
3. Copy the following application definition (in JSON format) and save to `marathon.json` file. Update any parameters which would like to change. If doesn't set restriction on cluster then the framework will try to occupy all resources in Mesos cluster. See **Configuration** section below.

```
{  
  "id": "ignition",  
  "instances": 1,  
  "cpus": 2,  
  "mem": 2048,  
  "ports": [0],  
  "uris": [  
    "http://host/ignite-mesos-<ignite-version>-jar-with-dependencies.jar"  
,  
  "env": {  
    "IGNITE_NODE_COUNT": "4",  
    "MESOS_MASTER_URL": "zk://localhost:2181/mesos",  
    "IGNITE_RUN_CPU_PER_NODE": "2",  
    "IGNITE_MEMORY_PER_NODE": "2048",  
    "IGNITE_VERSION": "1.0.5"  
  },  
  "cmd": "java -jar ignite-mesos-<ignite-version>-jar-with-dependencies.jar"  
}
```

1. Send POST request with the application definition to Marathon by CURL or other tools.

```
curl -X POST -H "Content-type: application/json" --data-binary @marathon.json  
http://<marathon-ip>:8080/v2/apps/
```

1. In order to make sure that Apache Mesos Framework deployed correctly, do the following. Open Marathon UI at <http://<marathon-ip>:8080>. Make sure that exists application with name `ignition` and its status is `Running`.

The screenshot shows a web browser window with two tabs: 'Marathon' and 'Mesos'. The 'Marathon' tab is active, displaying the URL [52.8.121.33:8080/#apps](http://52.8.121.33:8080/#apps). The Marathon interface has a dark theme with a header containing the 'MARATHON' logo, 'Apps' (which is underlined), 'Deployments', 'About', and 'Docs'. Below the header is a green button labeled '+ New App'. A table lists the application details:

ID	Memory (MB)	CPUs	Tasks / Instances	Health	Status
/ignition	2048	2	1 / 1	<div style="width: 100%; background-color: #ccc; height: 10px;"></div>	Running

1. Open Mesos console at <http://<master-ip>:5050>. If everything works OK then tasks with name like `Ignite_node_N` should have state `RUNNING`. In this example N=4. See example `marathon.json` file -  
"IGNITE\_NODE\_COUNT": "4"

Marathon Mesos

52.8.121.33:5050/#

Mesos Frameworks Slaves Offers

**Cluster: (Unnamed)**  
**Server:** 52.8.121.33:5050  
**Version:** 0.22.1  
**Built:** 3 weeks ago by root  
**Started:** 20 minutes ago  
**Elected:** 20 minutes ago

**LOG**

**Slaves**

Activated	5
Deactivated	0

**Tasks**

Staged	5
Started	0
Finished	0
Killed	0
Failed	0
Lost	0

**Resources**

	CPU	Mem
Total	20	31.5 GB
Used	18	10.0 GB
Offered	0	0 B
Idle	2	21.5 GB

**Active Tasks**

ID	Name	State	Started ▾	Host
2	Ignite node 2	RUNNING	8 minutes ago	ec2-52-8-105-68.us-west-1.compute.amazonaws.com
1	Ignite node 1	RUNNING	8 minutes ago	ec2-52-8-121-33.us-west-1.compute.amazonaws.com
4	Ignite node 4	RUNNING	8 minutes ago	ec2-52-8-6-79.us-west-1.compute.amazonaws.com
3	Ignite node 3	RUNNING	8 minutes ago	ec2-52-8-118-95.us-west-1.compute.amazonaws.com
ignition.8e3f079a-0517-11e5-88f5-06239b67b81d	ignition	RUNNING	8 minutes ago	ec2-52-8-87-179.us-west-1.compute.amazonaws.com

**Completed Tasks**

ID	Name	State	Started ▾	Stopped	Host
No completed tasks.					

1. Mesos allows to retrieve tasks' logs from browser. To look through Ignite logs click on **Sandbox** in the Active Tasks table.

Marathon Mesos

52.8.121.33:5050/#/slaves/20150528-071451-1325795380-5050-1132-S2/browse?path=%2Ftmp%2Fmesos%2Fslaves%2F20150528-071451-1325795380-5050-1132-S2%2Fgridgain-community-fabric-1.0.6

Mesos Frameworks Slaves Offers

Master / Slave / Browse

/ tmp / mesos / slaves / 20150528-071451-1325795380-5050-1132-S2 / frameworks / 20150528-084538-561580084-5050-6691-0000 / executors / 3 / runs / 00f75243-2796-461c-828e-c044f318f3fc

mode	nlink	uid	gid	size	mtime	
drwxr-xr-x	8	root	root	4 KB	May 28 11:57	gridgain-community-fabric-1.0.6
-rw-r--r--	1	root	root	55 MB	May 28 11:57	gridgain-community-fabric-1.0.6.zip
-rw-r--r--	1	root	root	2 KB	May 28 11:57	ignite-default-config.xml
-rw-r--r--	1	root	root	17 KB	May 15 17:14	ignite-log4j-1.0.4-SNAPSHOT-sources.jar
-rw-r--r--	1	root	root	23 KB	May 15 17:14	ignite-log4j-1.0.4-SNAPSHOT.jar
-rw-r--r--	1	root	root	27 KB	May 28 11:57	ignite-mesos-1.1.0-SNAPSHOT.jar
-rw-r--r--	1	root	root	35 KB	May 28 11:57	libs.zip
-rw-r--r--	1	root	root	3 KB	May 28 11:57	stderr
-rw-r--r--	1	root	root	636 KB	May 28 11:57	stdout

1. Click on **stdout** to get stdout logs and on **stderr** to get stderr logs.

# Run the Framework via JAR file

1. Download Ignite package and go to `libs\optional\ignite-mesos\` folder.
  2. Run the framework.

```
java -jar ignite-mesos-<ignite-version>-jar-with-dependencies.jar
```

or

```
java -jar ignite-mesos-<ignite-version>-jar-with-dependencies.jar properties.prop
```

where `properties.prop` is a property file. If file is not provided then the framework will try to occupy all resources in Mesos cluster. Example property file:

```

#The number of nodes in the cluster.
IGNITE_NODE_COUNT=1

#Mesos ZooKeeper URL to locate leading master.
MESOS_MASTER_URL=zk://localhost:2181/mesos

#The number of CPU Cores for each Apache Ignite node.
IGNITE_RUN_CPU_PER_NODE=4

#The number of Megabytes of RAM for each Apache Ignite node.
IGNITE_MEMORY_PER_NODE=4096

#The version ignite which will be run on nodes.
IGNITE_VERSION=1.0.5

```

1. In order to make sure that Apache Mesos Framework deployed correctly, do the following. Open Mesos console at <http://<master-ip>:5050>. If everything works OK then tasks with name like **Ignite node N** should have state **RUNNING**. In this example N=1. See example **properties.prop** file - "IGNITE\_NODE\_COUNT": "1"

The screenshot shows the Apache Mesos web interface. At the top, there is a navigation bar with tabs for Mesos, Frameworks, Slaves, and Offers. The 'Mesos' tab is selected. Below the navigation bar, the URL is http://192.168.1.153:5050/#/. The main content area is divided into two sections: 'Active Tasks' and 'Completed Tasks'. The 'Active Tasks' section contains a table with one row:

ID	Name	State	Started	Host
1	Ignite node 1	RUNNING	3 minutes ago	192.168.1.153

The 'Completed Tasks' section shows a table with the message "No completed tasks.".

On the left side of the interface, there is a sidebar with the following information:

- Cluster:** (Unnamed)
- Server:** 192.168.1.153:5050
- Version:** 0.22.1
- Built:** 3 weeks ago by root
- Started:** 8 minutes ago
- Elected:** 8 minutes ago

Below the sidebar, there are sections for **LOG**, **Slaves**, and **Tasks**.

1. Mesos allows to retrieve tasks' logs from browser. To look through Ignite logs click on **Sandbox**.

Mesos	Frameworks	Slaves	Offers
-rw-r--r--	1	nikolay	nikolay
			59 KB
			May 26 19:37
			jcommander-1.32.jar
			<a href="#">Download</a>
-rw-r--r--	1	nikolay	nikolay
			316 KB
			May 26 19:37
			jcommon-1.0.21.jar
			<a href="#">Download</a>
-rw-r--r--	1	nikolay	nikolay
			1 MB
			May 26 19:37
			jfreechart-1.0.17.jar
			<a href="#">Download</a>
-rw-r--r--	1	nikolay	nikolay
			32 KB
			May 26 19:36
			jsr305-1.3.9.jar
			<a href="#">Download</a>
-rw-r--r--	1	nikolay	nikolay
			23 KB
			May 26 19:37
			minimal-json-0.9.1.jar
			<a href="#">Download</a>
-rw-r--r--	1	nikolay	nikolay
			118 KB
			May 26 19:36
			reflections-0.9.9-RC1.jar
			<a href="#">Download</a>
-rw-r--r--	1	nikolay	nikolay
			10 KB
			May 26 19:37
			stderr
			<a href="#">Download</a>
-rw-r--r--	1	nikolay	nikolay
			638 KB
			May 26 19:37
			stdout
			<a href="#">Download</a>
-rw-r--r--	1	nikolay	nikolay
			190 KB
			May 26 19:36
			xml-apis-1.3.04.jar
			<a href="#">Download</a>

1. Click on `stdout` to get stdout logs and on `stderr` to get stderr logs.

# Configuration

All configuration is handled through environment variables (this lends itself well to being easy to configure marathon to run the framework) or property file. Following configuration parameters can be optionally configured.

Name	Description	Default	IGNITE_RUN_CPU_PER_NODE
The number of CPU Cores for each Apache Ignite node.	UNLIMITED	Example	2

Name	Description	Default	IGNITE_RUN_CPU_PER_NODE
IGNITE_MEMORY_PER_NODE	The number of Megabytes of RAM for each Apache Ignite node.	UNLIMITED	1024
IGNITE_DISK_SPACE_PER_NODE	The number of Megabytes of Disk for each Apache Ignite node.	1024	2048
IGNITE_NODE_COUNT	The number of nodes in the cluster.	5	10
IGNITE_TOTAL_CPU	The number of CPU Cores for Ignite cluster.	UNLIMITED	5
IGNITE_TOTAL_MEMORY	The number of Megabytes of RAM for Ignite cluster.	UNLIMITED	16384
IGNITE_TOTAL_DISK_SPACE	The number of Megabytes of Disk for each Apache Ignite cluster.	UNLIMITED	5120
IGNITE_MIN_CPU_PER_NODE	The minimum number of CPU cores required to run Apache Ignite node.	1	4
IGNITE_MIN_MEMORY_PER_NODE	The minimum number of Megabytes of RAM cores required to run Apache Ignite node.	256	1024
IGNITE_VERSION	The version ignite which will be run on nodes.	latest	1.0.5
IGNITE_WORK_DIR	The directory which will be used for saving Apache Ignite distributives.	ignite-release	/opt/ignite/
IGNITE_XML_CONFIG	The path to Apache Ignite config file.	N/A	/opt/ignite/ignite-config.xml
IGNITE_USERS_LIBS	The path to libs which will be added to classpath.	N/A	/opt/libs/
MESOS_MASTER_URL	Mesos ZooKeeper URL to locate leading master.	zk://localhost:2181/mesos	zk://176.0.1.45:2181/mesos or 176.0.1.45:2181

Name	Description	Default	IGNITE_RUN_CPU_PER_NODE
IGNITE_CONFIG_XML_URL	The url to Apache Ignite config file.	N/A	<a href="https://example.com/default-config.xml">https://example.com/default-config.xml</a>
IGNITE_USERS_LIBS_URL	Comma separated list of urls to libs which will be added to classpath.	N/A	<a href="https://example.com/lib.zip">https://example.com/lib.zip</a> , <a href="https://example.com/lib1.zip">https://example.com/lib1.zip</a>

## YARN Deployment

Deploy Ignite in YARN cluster.

---

Integration with YARN supports scheduling and running Apache Ignite nodes in a YARN cluster. YARN is a resource negotiator which provides a general runtime environment providing all the essentials to deploy, run and manage distributed applications. Its resource manager and isolation helps getting the most out of servers. For information about YARN refer to <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>

### Ignite YARN Application

---

Deploying Apache Ignite cluster typically involves downloading the Apache Ignite distribution, changing configuration settings and starting the nodes up. Integration with YARN allows to avoid the actions. Apache Ignite Yarn Application allows to greatly simplify cluster deployment. The application consist from the following components:

- Client downloads ignte distributive, puts necessary resources to HDFS, creates the necessary context for launching the task, launches the ApplicationMaster process.
- **Application master**. Once registration is successful the component will begin requesting of resource from Resource Manager to utilize resources for Apache Ignite nodes. **The Application Master** will maintain the Ignite cluster at desired total resources level (CPU, memory, etc).
- **Container** - the entity that runs Ignite Node on slaves.

### Running Ignite YARN application

---

For running Ignite Application requires YARN and Hadoop cluster are configured and running. For information on how to set up a the cluster please refer to <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html>

1. Download Apache Ignite.

- Configure properties file. Update any parameters which would like to change. See **Configuration** section below.

```
#The number of nodes in the cluster.
IGNITE_NODE_COUNT=2

#The number of CPU Cores for each Apache Ignite node.
IGNITE_RUN_CPU_PER_NODE=1

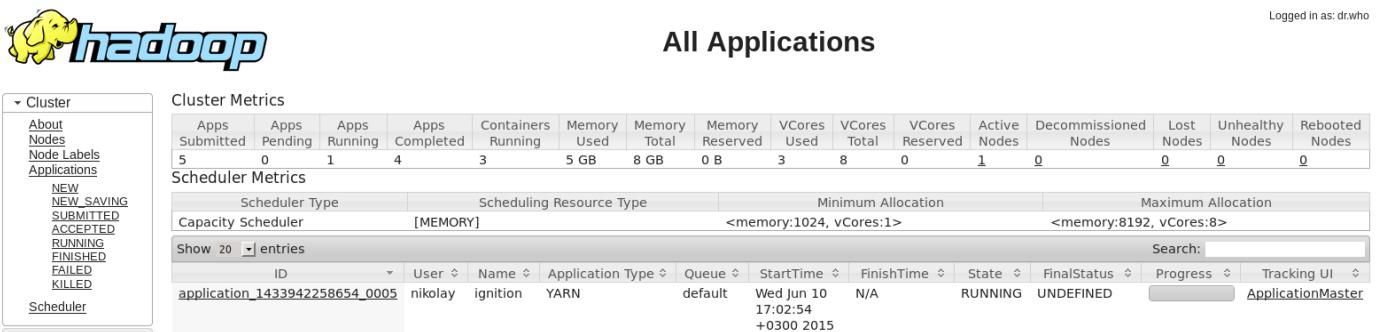
#The number of Megabytes of RAM for each Apache Ignite node.
IGNITE_MEMORY_PER_NODE=2048

#The version ignite which will be run on nodes.
IGNITE_VERSION=1.0.6
```

- Run the application.

```
hadoop java jar ignite-yarn-<ignite-version>.jar ./ignite-yarn-<ignite-version>.jar
cluster.properties
```

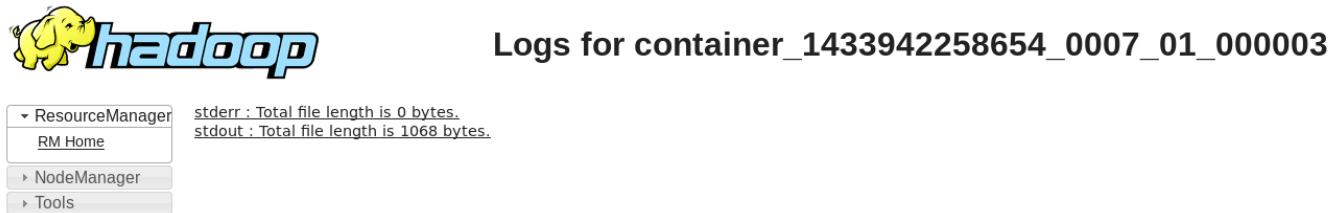
- In order to make sure that Application deployed correctly, do the following. Open YARN console at <http://<hostname>:8088/cluster>. If everything works OK then application with **Ignition** name.



The screenshot shows the Hadoop YARN Cluster Metrics page. On the left, there's a sidebar with a yellow elephant logo and navigation links for Cluster (About Nodes, Node Labels, Applications), Scheduler (Scheduler), and Resource Manager (RM Home, Node Manager, Tools). The main area is titled "All Applications". It displays a table of scheduler metrics with one entry:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	ApplicationMaster
application_1433942258654_0005	nikolay	ignition	YARN	default	Wed Jun 10 17:02:54 +0300 2015	N/A	RUNNING	UNDEFINED			

- Retrieve logs from browser. To look through Ignite logs click on **Logs** for any containers.



The screenshot shows the Hadoop ResourceManager page. On the left, there's a sidebar with a yellow elephant logo and navigation links for ResourceManager (RM Home, Node Manager, Tools). The main area is titled "Logs for container\_1433942258654\_0007\_01\_000003". It displays two log entries:

- stderr : Total file length is 0 bytes.
- stdout : Total file length is 1068 bytes.

- Click on **stdout** to get stdout logs and on **stderr** to get stderr logs.



## Configuration

All configuration is handled through environment variables or property file. Following configuration parameters can be optionally configured.

<b>IGNITE_XML_CONFIG</b>	<b>Name</b>	<b>Description</b>	<b>Default</b>
Example	The hdfs path to Apache Ignite config file.	N/A	/opt/ignite/ignite-config.xml
IGNITE_WORK_DIR	The directory which will be used for saving Apache Ignite distributives.	./ignite-release	/opt/ignite/
IGNITE_RELEASES_DIR	The hdfs directory which will be used for saving Apache Ignite distributives.	/ignite/releases/	/ignite-rel/
IGNITE_USERS_LIBS	The hdfs path to libs which will be added to classpath.	N/A	/opt/libs/
IGNITE_MEMORY_PER_NODE	The number of Megabytes of RAM for each Apache Ignite node.	2048	1024
IGNITE_HOSTNAME_CONSTRAINT	The constraint on slave hosts.	N/A	192.168.0.[1-100]
IGNITE_NODE_COUNT	The number of nodes in the cluster.	3	10
IGNITE_RUN_CPU_PER_NODE	The number of CPU Cores for each Apache Ignite node.	2	4

IGNITE_XML_CONFIG	Name	Description	Default
IGNITE_VERSION	The version ignite which will be run on nodes.	latest	1.0.5

## SSL\TLS

Ignite allows you to use SSL socket communication among all Ignite nodes. To use it, you need to set `Factory<SSLContext>` and configure the SSL section in the Ignite configuration. Ignite provides a default SSL context factory, `org.apache.ignite.ssl.SslContextFactory`, which uses configured keystore to initialize SSL context.

```
<bean id="cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    <property name="sslContextFactory">
        <bean class="org.apache.ignite.ssl.SslContextFactory">
            <property name="keyStoreFilePath" value="keystore/server.jks"/>
            <property name="keyStorePassword" value="123456"/>
            <property name="trustStoreFilePath" value="keystore/trust.jks"/>
            <property name="trustStorePassword" value="123456"/>
        </bean>
    </property>
</bean>
```

```
IgniteConfiguration igniteCfg = new IgniteConfiguration();

SslContextFactory factory = new SslContextFactory();

factory.setKeyStoreFilePath("keystore/server.jks");
factory.setKeyStorePassword("123456".toCharArray());
factory.setTrustStoreFilePath("keystore/trust.jks");
factory.setTrustStorePassword("123456".toCharArray());

igniteCfg.setSslContextFactory(factory);
```

In some cases it is useful to disable certificate validation of the client side (e.g. when connecting to a server with self-signed certificate). This can be achieved by setting a disabled trust manager to this factory, which can be obtained by `getDisabledTrustManager` method.

```

<bean id="cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    <property name="sslContextFactory">
        <bean class="org.apache.ignite.ssl.SslContextFactory">
            <property name="keyStoreFilePath" value="keystore/server.jks"/>
            <property name="keyStorePassword" value="123456"/>
            <property name="trustManagers">
                <bean class="org.apache.ignite.ssl.SslContextFactory" factory-method=
"getDisabledTrustManager"/>
            </property>
        </bean>
    </property>
</bean>

```

```

IgniteConfiguration igniteCfg = new IgniteConfiguration();

SslContextFactory factory = new SslContextFactory();

factory.setKeyStoreFilePath("keystore/server.jks");
factory.setKeyStorePassword("123456".toCharArray());
factory.setTrustManagers(SslContextFactory.getDisabledTrustManager());

igniteCfg.setSslContextFactory(factory);

```

If security is configured then the logs will include `communication encrypted=on`

```
INFO: Security status [authentication=off, communication encrypted=on]
```

## SSL and TLS

Ignite allows using different types of encryption. The following algorithms are supported <http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#SSLContext> and can be set by using the `setProtocol` method. **TLS** encryption is the default.

```

<bean id="cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    <property name="sslContextFactory">
        <bean class="org.apache.ignite.ssl.SslContextFactory">
            <property name="protocol" value="SSL"/>
            ...
        </bean>
    </property>
    ...
</bean>

```

```

IgniteConfiguration igniteCfg = new IgniteConfiguration();
SslContextFactory factory = new SslContextFactory();
...
factory.setProtocol("TLS");
igniteCfg.setSslContextFactory(factory);

```

## Configuration

Following configuration parameters can be configured on `SslContextFactory`.

Setter Method	Description	Default
<code>setKeyAlgorithm</code>	Sets key manager algorithm that will be used to create a key manager. Notice that in most cases default value suites well, however, on Android platform this value need to be set to <code>X509</code> .	<code>SunX509</code>
<code>setKeyStoreFilePath</code>	Sets path to the key store file. This is a mandatory parameter since ssl context could not be initialized without key manager.	<code>N/A</code>
<code>setKeyStorePassword</code>	Sets key store password.	<code>N/A</code>
<code>setKeyStoreType</code>	Sets key store type used in context initialization.	<code>JKS</code>
<code>setProtocol</code>	Sets protocol for secure transport.	<code>TLS</code>

<b>Setter Method</b>	<b>Description</b>	<b>Default</b>
<code>setTrustStoreFilePath</code>	Sets path to the trust store file.	N/A
<code>setTrustStorePassword</code>	Sets trust store password.	N/A
<code>setTrustStoreType</code>	Sets trust store type used in context initialization.	JKS
<code>setTrustManagers</code>	Sets pre-configured trust managers.	'N/A`

# Data Grid

# Data Grid

Replicate or partition your data in memory within the cluster.

---

Ignite in-memory data grid has been built from the ground up with a notion of horizontal scale and ability to add nodes on demand in real-time; it has been designed to linearly scale to hundreds of nodes with strong semantics for data locality and affinity data routing to reduce redundant data noise.

Ignite data grid is an **in-memory distributed key-value store** which can be viewed as a distributed partitioned hash map, with every cluster node owning a portion of the overall data. This way the more cluster nodes we add, the more data we can cache.

Unlike other key-value stores, Ignite determines data locality using a pluggable hashing algorithm. Every client can determine which node a key belongs to by plugging it into a hashing function, without a need for any special mapping servers or name nodes.

Ignite data grid supports local, replicated, and partitioned data sets and allows to freely cross query between these data sets using standard SQL syntax. Ignite supports standard SQL for querying in-memory data including support for distributed SQL joins.

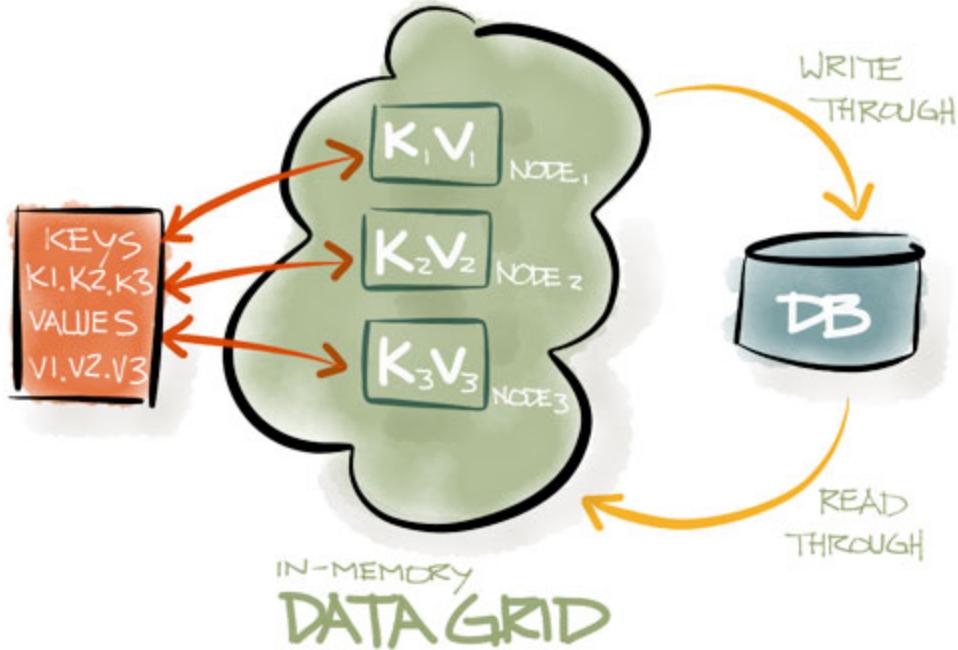
Ignite data grid is lightning fast and is one of the fastest implementations of transactional or atomic data in a cluster today.



**Data Consistency.** As long as your cluster is alive, Ignite will guarantee that the data between different cluster nodes will always remain consistent regardless of crashes or topology changes.



**JCache (JSR 107).** Ignite Data Grid implements [JCache and Beyond](#) specification.



## Features

- Distributed In-Memory Caching
- Lightning Fast Performance
- Elastic Scalability
- Distributed In-Memory Transactions
- Web Session Clustering
- Hibernate L2 Cache Integration
- Tiered Off-Heap Storage
- Distributed ANSI-99 SQL Queries with support for Joins

## IgniteCache

---

**IgniteCache** interface is a gateway into Ignite cache implementation and provides methods for storing and retrieving data, executing queries, including SQL, iterating and scanning, etc.

## JCache

**IgniteCache** interface extends `javax.cache.Cache` interface from JCache specification and adds additional functionality to it, mainly having to do with local vs. distributed operations, queries, metrics, etc.

You can obtain an instance of **IgniteCache** as follows:

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

// Obtain instance of cache named "myCache".
// Note that different caches may have different generics.
IgniteCache<Integer, String> cache = ignite.cache("myCache");
```

## Dynamic Cache

You can also create the cache on the fly, in which case Ignite will create and deploy the cache on all the server nodes in the cluster. Similarly, you can also destroy the cache dynamically across the server nodes within the cluster.

*Java*

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

CacheConfiguration cfg = new CacheConfiguration();

cfg.setName("myCache");
cfg.setAtomicityMode(TRANSACTIONAL);

// Create the cache with given name, if it does not exist.
IgniteCache<Integer, String> cache = ignite.getOrCreateCache(cfg);

// Destroy the cache with given name.
ignite.destroyCache("myCache");
```



**XML Configuration.** All caches defined in Ignite Spring XML configuration on any cluster member will also be automatically created and deployed on all the cluster servers (no need to specify the same configuration on each cluster member).

## JCache and Beyond

Apache Ignite data grid is an implementation of JCache (JSR 107) specification. JCache provides a very simple to use, but yet very powerful API for data access. However, the specification purposely omits any details about data distribution and consistency to allow vendors enough freedom in their own implementations.

With JCache support you get the following:

- Basic Cache Operations
- ConcurrentMap APIs

- Collocated Processing (EntryProcessor)
- Events and Metrics
- Pluggable Persistence

In addition to JCache, Ignite provides ACID transactions, data querying capabilities (including SQL), various memory models, queries, transactions, etc...

## IgniteCache

---

`IgniteCache` is based on **JCache (JSR 107)**, so at the very basic level the APIs can be reduced to `javax.cache.Cache` interface. However, `IgniteCache` API also provides functionality that facilitates features outside of JCache spec, like data loading, querying, asynchronous mode, etc.

You can get an instance of `IgniteCache` directly from `Ignite`:

```
Ignite ignite = Ignition.ignite();
IgniteCache cache = ignite.cache("mycache");
```

## Basic Operations

---

Here are some basic JCache atomic operation examples.

### Put & Get

```
try (Ignite ignite = Ignition.start("examples/config/example-cache.xml")) {
    IgniteCache<Integer, String> cache = ignite.cache(CACHE_NAME);

    // Store keys in cache (values will end up on different cache nodes).
    for (int i = 0; i < 10; i++)
        cache.put(i, Integer.toString(i));

    for (int i = 0; i < 10; i++)
        System.out.println("Got [key=" + i + ", val=" + cache.get(i) + ']');
}
```

## Atomic

```
// Put-if-absent which returns previous value.  
Integer oldVal = cache.getAndPutIfAbsent("Hello", 11);  
  
// Put-if-absent which returns boolean success flag.  
boolean success = cache.putIfAbsent("World", 22);  
  
// Replace-if-exists operation (opposite of getAndPutIfAbsent), returns previous value.  
oldVal = cache.getAndReplace("Hello", 11);  
  
// Replace-if-exists operation (opposite of putIfAbsent), returns boolean success flag.  
success = cache.replace("World", 22);  
  
// Replace-if-matches operation.  
success = cache.replace("World", 2, 22);  
  
// Remove-if-matches operation.  
success = cache.remove("Hello", 1);
```

## EntryProcessor

Whenever doing `puts` and `updates` in cache, you are usually sending full state object state across the network. `EntryProcessor` allows for processing data directly on primary nodes, often transferring only the deltas instead of the full state.

Moreover, you can embed your own logic into `EntryProcessors`, for example, taking previous cached value and incrementing it by 1.

### *invoke*

```
IgniteCache<String, Integer> cache = ignite.cache("mycache");  
  
// Increment cache value 10 times.  
for (int i = 0; i < 10; i++)  
    cache.invoke("mykey", (entry, args) -> {  
        Integer val = entry.getValue();  
  
        entry.setValue(val == null ? 1 : val + 1);  
  
        return null;  
   });
```

*java7 invoke*

```
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;
        }
    });
});
```



**Atomicity.** [EntryProcessors](#) are executed atomically within a lock on the given cache key.

## Asynchronous Support

Just like all distributed APIs in Ignite, [IgniteCache](#) extends [IgniteAsyncSupport](#) interface and can be used in asynchronous mode.

*Async*

```
// Enable asynchronous mode.
IgniteCache<String, Integer> asyncCache = ignite.cache("mycache").withAsync();

// Asynchronously store value in cache.
asyncCache.getAndPut("1", 1);

// Get future for the above invocation.
IgniteFuture<Integer> fut = asyncCache.future();

// Asynchronously listen for the operation to complete.
fut.listenAsync(f -> System.out.println("Previous cache value: " + f.get()));
```

## Cache Modes

Setup different distribution models, backup copies, and near caches.

Ignite provides three different modes of cache operation: **PARTITIONED**, **REPLICATED**, and **LOCAL**. A cache mode is configured for each cache. Cache modes are defined in `CacheMode` enumeration.

## Partitioned Mode

---

**PARTITIONED** mode is the most scalable distributed cache mode. In this mode the overall data set is divided equally into partitions and all partitions are split equally between participating nodes, essentially creating one huge distributed in-memory store for caching data. This approach allows you to store as much data as can be fit in the total memory available across all nodes, hence allowing for multi-terabytes of data in cache memory across all cluster nodes. Essentially, the more nodes you have, the more data you can cache.

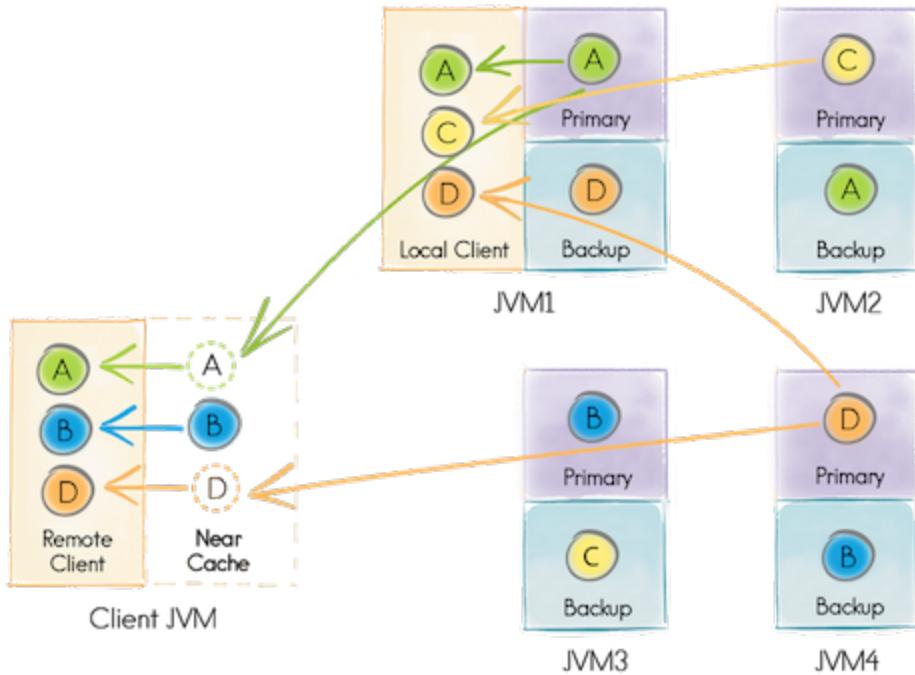
Unlike **REPLICATED** mode, where updates are expensive because every node in the cluster needs to be updated, with **PARTITIONED** mode, updates become cheap because only one primary node (and optionally 1 or more backup nodes) need to be updated for every key. However, reads become somewhat more expensive because only certain nodes have the data cached.

In order to avoid extra data movement, it is important to always access the data exactly on the node that has that data cached. This approach is called **affinity colocation** and is strongly recommended when working with partitioned caches.



- . Partitioned caches are ideal when working with large data sets and updates are frequent. The picture below illustrates a simple view of a partitioned cache. Essentially we have key A assigned to a node running in JVM1, B assigned to a node running in JVM3, etc...

## Partitioned Cache



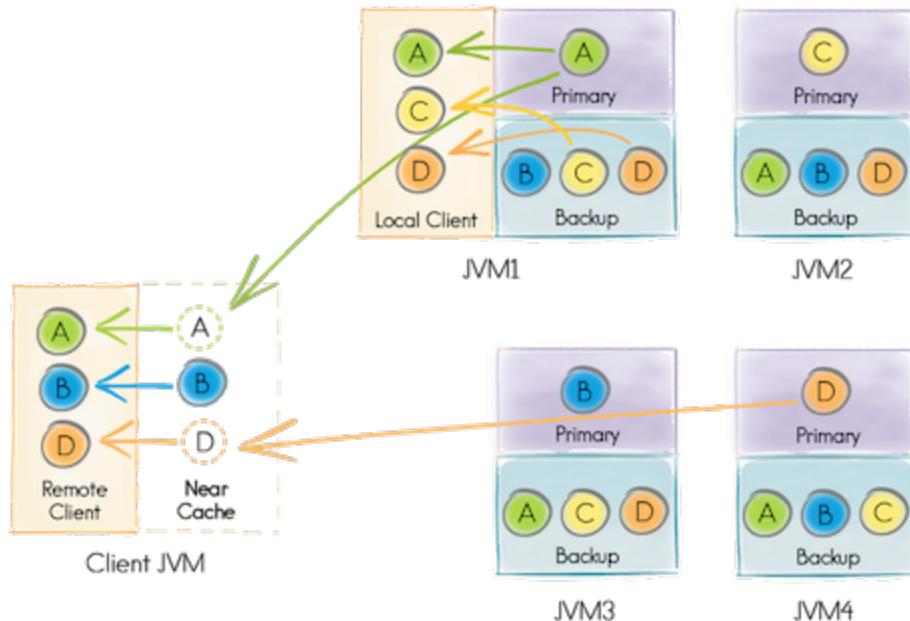
See [Configuration](#) section below for an example on how to configure cache mode.

## Replicated Mode

In **REPLICATED** mode, all the data is replicated to every node in the cluster. This cache mode provides the utmost availability of data as it is available on every node. However, in this mode every data update must be propagated to all other nodes which can have an impact on performance and scalability.

In Ignite, **replicated caches** are implemented using **partitioned caches** where every key has a primary copy and is also backed up on all other nodes in the cluster. For example, in the diagram below, the node running in JVM1 is a primary node for key A, but it also stores backup copies for all other keys as well (B, C, D).

## Replicated Cache



As the same data is stored on all cluster nodes, the size of a replicated cache is limited by the amount of memory available on the 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 data sets are small. If your system does cache lookups over 80% of the time, then you should consider using **REPLICATED** cache mode.



Replicated caches should be used when data sets are small and updates are infrequent.

## Local Mode

**LOCAL** mode is the most light weight mode of cache operation, as no data is distributed to other cache nodes. It is ideal for scenarios where data is either read-only, or can be periodically refreshed at some expiration frequency. It also works very well with **read-through** behavior where data is loaded from persistent storage on misses. Other than distribution, local caches still have all the features of a distributed cache, such as automatic data eviction, expiration, disk swapping, data querying, and transactions.

## Configuration

Cache modes are configured for each cache by setting the `cacheMode` property of `CacheConfiguration` like so:

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="cacheName"/>

            <!-- Set cache mode. -->
            <property name="cacheMode" value="PARTITIONED"/>
            ...
        </bean>
    </property>
</bean>

```

```

CacheConfiguration cacheCfg = new CacheConfiguration("myCache");

cacheCfg.setCacheMode(CacheMode.PARTITIONED);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);

```

## Atomic Write Order Mode

When using partitioned cache in `CacheAtomicityMode.ATOMIC` mode, one can configure atomic cache write order mode. Atomic write order mode determines which node will assign write version (sender or primary node) and is defined by `CacheAtomicWriteOrderMode` enumeration. There are 2 modes, `CLOCK` and `PRIMARY`.

In `CLOCK` write order mode, write versions are assigned on a sender node. `CLOCK` mode is automatically turned on only when `CacheWriteSynchronizationMode.FULL_SYNC` is used, as it generally leads to better performance since write requests to primary and backups nodes are sent at the same time.

In `PRIMARY` write order mode, cache version is assigned only on primary node. In this mode the sender will only send write requests to primary nodes, which in turn will assign write version and forward them to backups.

Atomic write order mode can be configured via `atomicWriteOrderMode` property of `CacheConfiguration`.

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="cacheName"/>

            <!-- Atomic write order mode. -->
            <property name="atomicWriteOrderMode" value="PRIMARY"/>
            ...
        </bean>
    </property>
</bean>
```

```
CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

cacheCfg.setAtomicWriteOrderMode(CacheAtomicWriteOrderMode.CLOCK);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```



For more information on **ATOMIC** mode, refer to [Transactions](/docs/transactions) section.

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="cacheName"/>

            <!-- Set cache mode. -->
            <property name="cacheMode" value="PARTITIONED"/>

            <!-- Number of backup nodes. -->
            <property name="backups" value="1"/>
            ...
        </bean>
    </property>
</bean>

```

```

CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

cacheCfg.setCacheMode(CacheMode.PARTITIONED);

cacheCfg.setBackups(1);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);

```

## Primary & Backup Copies

In **PARTITIONED** mode, nodes to which the keys are assigned to are called primary nodes for those keys. You can also optionally configure any number of backup nodes for cached data. If the number of backups is greater than 0, then Ignite will automatically assign backup nodes for each individual key. For example if the number of backups is 1, then every key cached in the data grid will have 2 copies, 1 primary and 1 backup.



By default, backups are turned off for better performance. Backups can be configured by setting `backups()` property of `CacheConfiguration`, like so:

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="cacheName"/>

            <!-- Set cache mode. -->
            <property name="cacheMode" value="PARTITIONED"/>

            <!-- Number of backup nodes. -->
            <property name="backups" value="1"/>
            ...
        </bean>
    </property>
</bean>

```

```

CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

cacheCfg.setCacheMode(CacheMode.PARTITIONED);

cacheCfg.setBackups(1);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);

```

## Synchronous and Asynchronous Backups

`CacheWriteSynchronizationMode` enum can be used to configure synchronous or asynchronous update of primary and backup copies. Write synchronization mode tells Ignite whether the client node should wait for responses from remote nodes, before completing a write or commit.

Write synchronization mode can be set in one of following 3 modes:

FULL_SYNC	FULL_ASYNC
PRIMARY_SYNC	Client node will wait for write or commit to complete on all participating remote nodes (primary and backup).
This is the default value. In this mode, client node does not wait for responses from participating nodes, in which case remote nodes may get their state updated slightly after any of the cache write methods complete or after <code>Transaction.commit()</code> method completes.	Client node will wait for write or commit to complete on primary node, but will not wait for backups to be updated.



**Cache Data Consistency.** Note that regardless of write synchronization mode, cache data will always remain fully consistent across all participating nodes. Write synchronization mode may be configured by setting `writeSynchronizationMode` property of `CacheConfiguration`, like so:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="cacheName"/>

            <!-- Set write synchronization mode. -->
            <property name="writeSynchronizationMode" value="FULL_SYNC"/>
            ...
        </bean>
    </property>
</bean>
```

```
CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

cacheCfg.setWriteSynchronizationMode(CacheWriteSynchronizationMode.FULL_SYNC);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

# Near Caches

Create local client-side caches.

A partitioned cache can also be fronted by a **Near** cache, which is a smaller local cache that stores most recently or most frequently accessed data. Just like with a partitioned cache, the user can control the size of the near cache and its eviction policies.

Near caches can be created directly on **client** nodes by passing **NearConfiguration** into the `Ignite.createNearCache(NearConfiguration)` or `Ignite.getOrCreateNearCache(NearConfiguration)` methods.

```
// Create distributed cache on the server nodes, called "myCache".
ignite.getOrCreateCache(new CacheConfiguration<MyKey, MyValue>("myCache"));

// Create near-cache configuration for "myCache".
NearCacheConfiguration<MyKey, MyValue> nearCfg = new NearCacheConfiguration<>();

// Use LRU eviction policy to automatically evict entries
// from near-cache, whenever it reaches 100_000 in size.
nearCfg.setEvictionPolicy(new LruEvictionPolicy<>(100_000));

// Create near-cache for "myCache".
IgniteCache<MyKey, MyValue> cache = ignite.getOrCreateNearCache("myCache", nearCfg);
```

In the vast majority of use cases, whenever utilizing Ignite with affinity colocation, near caches should not be used. If computations are collocated with the corresponding partition cache nodes then the near cache is simply not needed because all the data is available locally in the partitioned cache.

However, there are cases when it is simply impossible to send computations to remote nodes. For cases like this near caches can significantly improve scalability and the overall performance of the application.



**Near Caches on Server Nodes.** In rare cases, whenever accessing data from **PARTITIONED** caches on the server side in non-collocated fashion, you may need to configure near-caches on the server nodes as well via `CacheConfiguration.setNearConfiguration(...)` property.

## Configuration

Following are configuration parameters on **NearCacheConfiguration**.

<code>setNearEvictionPolicy(CacheEvictionPolicy)</code>	<b>Setter Method</b>	<b>Description</b>
Default	Eviction policy for the near cache.	None
<code>setNearStartSize(int)</code>	<code>375,000</code>	Eviction policy for near cache.

## Cache Queries

---

Ignite supports a very elegant query API with support for

- [Scan Queries](#)
- [SQL Queries](#)
- [Text Queries](#)

For SQL queries Ignite supports in-memory indexing, so all the data lookups are extremely fast. If you are caching your data in [Off-Heap Memory](#), then query indexes will also be cached in off-heap memory as well.

Ignite also provides support for custom indexing via [IndexingSpi](#) and [SpiQuery](#) class.

## Main Abstractions

---

[IgniteCache](#) has several query methods all of which receive some subclass of [Query](#) class and return [QueryCursor](#). [Query](#) abstract class represents an abstract paginated query to be executed on the distributed cache. You can set the page size for the returned cursor via [Query.setPageSize\(...\)](#) method (default is [1024](#)).

## QueryCursor

[QueryCursor](#) represents query result set and allows for transparent page-by-page iteration. Whenever user starts iterating over the last page, it will automatically request the next page in the background. For cases when pagination is not needed, you can use [QueryCursor.getAll\(\)](#) method which will fetch the whole query result and store it in a collection.



**Closing Cursors.** Cursors will close automatically if you call method [QueryCursor.getAll\(\)](#). If you are iterating over the cursor in a for loop or explicitly getting [Iterator](#), you must close() the cursor explicitly or use [AutoCloseable](#) syntax.

## Scan Queries

Scan queries allow for querying cache in distributed form based on some user defined predicate.

*scan*

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Find only persons earning more than 1,000.
try (QueryCursor cursor = cache.query(new ScanQuery((k, p) -> p.getSalary() > 1000))) {
    for (Person p : cursor)
        System.out.println(p.toString());
}
```

*java7 scan*

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Find only persons earning more than 1,000.
IgniteBiPredicate<Long, Person> filter = new IgniteByPredicate<>() {
    @Override public boolean apply(Long key, Person p) {
        return p.getSalary() > 1000;
    }
};

try (QueryCursor cursor = cache.query(new ScanQuery(filter))) {
    for (Person p : cursor)
        System.out.println(p.toString());
}
```

## SQL Queries

Ignite SQL queries are covered in a separate section [SQL Queries](#).

## Text Queries

Ignite also supports text-based queries based on Lucene indexing.

*text query*

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Query for all people with "Master Degree" in their resumes.
TextQuery txt = new TextQuery(Person.class, "Master Degree");

try (QueryCursor<Entry<Long, Person>> masters = cache.query(txt)) {
    for (Entry<Long, Person> e : masters)
        System.out.println(e.getValue().toString());
}
```

## Query Configuration by Annotations

Indexes can be configured from code by using `@QuerySqlField` annotations. To tell Ignite which types should be indexed, key-value pairs can be passed into `CacheConfiguration.setIndexedTypes(MyKey.class, MyValue.class)` method. Note that this method accepts only pairs of types, one for key class and another for value class.

```
public class Person implements Serializable {  
    /** Person ID (indexed). */  
    @QuerySqlField(index = true)  
    private long id;  
  
    /** Organization ID (indexed). */  
    @QuerySqlField(index = true)  
    private long orgId;  
  
    /** First name (not-indexed). */  
    @QuerySqlField  
    private String firstName;  
  
    /** Last name (not indexed). */  
    @QuerySqlField  
    private String lastName;  
  
    /** Resume text (create LUCENE-based TEXT index for this field). */  
    @QueryTextField  
    private String resume;  
  
    /** Salary (indexed). */  
    @QuerySqlField(index = true)  
    private double salary;  
  
    ...  
}
```

## Query Configuration by CacheTypeMetadata

Indexes and fields also could be configured with `org.apache.ignite.cache.CacheTypeMetadata`.

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
...
<!-- Cache configuration. -->
<property name="cacheConfiguration">
<list>
  <bean class="org.apache.ignite.configuration.CacheConfiguration">
    <property name="name" value="my_cache"/>
...
<!-- Cache types metadata. -->
<list>
  <bean class="org.apache.ignite.cache.CacheTypeMetadata">
    <!-- Type to query. -->
    <property name="valueType" value=
"org.apache.ignite.examples.datagrid.store.Person"/>
    <!-- Fields to be queried. -->
    <property name="queryFields">
      <map>
        <entry key="id" value="java.util.UUID"/>
        <entry key="orgId" value="java.util.UUID"/>
        <entry key="firstName" value="java.lang.String"/>
        <entry key="lastName" value="java.lang.String"/>
        <entry key="resume" value="java.lang.String"/>
        <entry key="salary" value="double"/>
      </map>
    </property>
    <!-- Fields to index in ascending order. -->
    <property name="ascendingFields">
      <map>
        <entry key="id" value="java.util.UUID"/>
        <entry key="orgId" value="java.util.UUID"/>
        <entry key="salary" value="double"/>
      </map>
    </property>
    <!-- Fields to index as text. -->
    <property name="textFields">
      <list>
        <value>resume</value>
      </list>
    </bean>
  </list>
...
</list>
</property>
...
</bean>
```

```

CacheConfiguration ccfg = new CacheConfiguration();
....
Collection<CacheTypeMetadata> types = new ArrayList<>();

CacheTypeMetadata type = new CacheTypeMetadata();
type.setValueType(Person.class.getName());

Map<String, Class<?>> qryFlds = type.getQueryFields();
qryFlds.put("id", UUID.class);
qryFlds.put("orgId", UUID.class);
qryFlds.put("firstName", String.class);
qryFlds.put("lastName", String.class);
qryFlds.put("resume", String.class);
qryFlds.put("salary", double.class);

Map<String, Class<?>> ascFlds = type.getAscendingFields();
ascFlds.put("id", UUID.class);
ascFlds.put("orgId", UUID.class);
ascFlds.put("salary", double.class);

Collection<String> txtFlds = type.getTextFields();
txtFlds.add("resume");

types.add(type);
...
ccfg.setTypeMetadata(types);
...

```

Note, annotations and `CacheTypeMetadata` are mutually exclusive.

For full example see [CacheQueryTypeMetadataExample](#).

## SQL Queries

Ignite supports free-form SQL queries virtually without any limitations. SQL syntax is ANSI-99 compliant. You can use any SQL function, any aggregation, any grouping and Ignite will figure out where to fetch the results from.

See example `SqlQuery` below.

## SQL Joins

Ignite supports distributed SQL joins. Moreover, if data resides in different caches, Ignite allows for cross-cache joins as well.

Joins between **PARTITIONED** and **REPLICATED** caches always work without any limitations. However, if you do a join between two **PARTITIONED** data sets, then you must make sure that the keys you are joining on are **collocated**.

See example **SqlQuery JOIN** below.

## Fields Queries

---

Instead of selecting the whole object, you can choose to select only specific fields in order to minimize network and serialization overhead. For this purpose Ignite has a concept of **fields queries**. Also it is useful when you want to execute some aggregate query.

See example **SqlFieldsQuery** below.

## Cross-Cache Queries

---

You can query data from multiple caches. In this case, cache names act as schema names in regular SQL. This means all caches can be referred by cache names in quotes. The cache on which the query was created acts as the default schema and does not need to be explicitly specified.

See example **Cross-Cache SqlFieldsQuery**.

### *SqlQuery*

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

SqlQuery sql = new SqlQuery(Person.class, "salary > ?");

// Find only persons earning more than 1,000.
try (QueryCursor<Entry<Long, Person>> cursor = cache.query(sql.setArgs(1000))) {
    for (Entry<Long, Person> e : cursor)
        System.out.println(e.getValue().toString());
}
```

## *SqlQuery JOIN*

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// SQL join on Person and Organization.
SqlQuery sql = new SqlQuery(Person.class,
    "from Person, Organization "
    + "where Person.orgId = Organization.id "
    + "and lower(Organization.name) = lower(?)");

// Find all persons working for Ignite organization.
try (QueryCursor<Entry<Long, Person>> cursor = cache.query(sql.setArgs("Ignite"))) {
    for (Entry<Long, Person> e : cursor)
        System.out.println(e.getValue().toString());
}
```

## *SqlFieldsQuery*

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Select with join between Person and Organization.
SqlFieldsQuery sql = new SqlFieldsQuery(
    "select concat(firstName, ' ', lastName), Organization.name "
    + "from Person, Organization where "
    + "Person.orgId = Organization.id and "
    + "Person.salary > ?");

// Only find persons with salary > 1000.
try (QueryCursor<List<?>> cursor = cache.query(sql.setArgs(1000))) {
    for (List<?> row : cursor)
        System.out.println("personName=" + row.get(0) + ", orgName=" + row.get(1));
}
```

## Cross-Cache SqlFieldsQuery

```
// In this example, suppose Person objects are stored in a
// cache named 'personCache' and Organization objects
// are stored in a cache named 'orgCache'.

IgniteCache<Long, Person> personCache = ignite.cache("personCache");

// Select with join between Person and Organization to
// get the names of all the employees of a specific organization.
SqlFieldsQuery sql = new SqlFieldsQuery(
    "select Person.name "
    + "from Person, \"orgCache\".Organization where "
    + "Person.orgId = Organization.id "
    + "and Organization.name = ?");

// Execute the query and obtain the query result cursor.
try (QueryCursor<List<?>> cursor = personCache.query(sql.setArgs("Ignite"))) {
    for (List<?> row : cursor)
        System.out.println("Person name=" + row);
}
```

## Configuring SQL Indexes by Annotations

Indexes can be configured from code by using `@QuerySqlField` annotations. To tell Ignite which types should be indexed, key-value pairs can be passed into `CacheConfiguration.setIndexedTypes` method like in example below. Note that this method accepts only pairs of types, one for key class and another for value class. Primitives are passed as boxed types.

```
CacheConfiguration<Object, Object> ccfg = new CacheConfiguration<>();

// Here we are setting 3 key-value type pairs to be indexed.
ccfg.setIndexedTypes(
    MyKey.class, MyValue.class,
    Long.class, MyOtherValue.class,
    UUID.class, String.class
);
```

## Making Fields Visible for SQL Queries

To make fields accessible for SQL queries you have to annotate them with `@QuerySqlField`. Field `age` will not be accessible from SQL. Note that none of these fields are indexed.

```

public class Person implements Serializable {
    /** Will be visible in SQL. */
    @QuerySqlField
    private long id;

    /** Will be visible in SQL. */
    @QuerySqlField
    private String name;

    /** Will NOT be visible in SQL. */
    private int age;
}

```

## Scala

```

case class Person (
    /** Will be visible in SQL. */
    @(QuerySqlField @field) id: Long,

    /** Will be visible in SQL. */
    @(QuerySqlField @field) name: String,

    /** Will NOT be visisble in SQL. */
    age: Int
) extends Serializable {
    ...
}

```



**Scala Annotations.** In Scala classes, the `@QuerySqlField` annotation must be accompanied by the `@field` annotation in order for a field to be visible for Ignite, like so: `@(QuerySqlField @field)`.

Alternatively, you can also use the `@ScalarCacheQuerySqlField` annotation from the `ignite-scalar` module which is just a type alias for the `@field` annotation. ===== Single Column Indexes To make fields not only accessible by SQL but also speedup queries you can index field values. To create a single column index you can annotate field with `@QuerySqlField(index = true)`.

```

public class Person implements Serializable {
    /** Will be indexed in ascending order. */
    @QuerySqlField(index = true)
    private long id;

    /** Will be visible in SQL, but not indexed. */
    @QuerySqlField
    private String name;

    /** Will be indexed in descending order. */
    @QuerySqlField(index = true, descending = true)
    private int age;
}

```

```

case class Person (
    /** Will be indexed in ascending order. */
    @QuerySqlField @field)(index = true) id: Long,

    /** Will be visible in SQL, but not indexed. */
    @QuerySqlField @field) name: String,

    /** Will be indexed in descending order. */
    @QuerySqlField @field)(index = true, descending = true) age: Int
) extends Serializable {
    ...
}

```



**Scala Annotations.** In Scala classes, the indexed `@QuerySqlField` annotation should look like so: `@(QuerySqlField @field)(index = true)`. ===== Group Indexes To have a multi-field index to speedup queries with complex conditions, you can use `@QuerySqlField.Group` annotation. It is possible to put multiple `@QuerySqlField.Group` annotations into `orderedGroups` if you want the field to participate in more than one group index.

For example of a group index in the class below we have field `age` which participates in a group index named `"age_salary_idx"` with group order 0 and descending sort order. Also in the same group index participates field `salary` with group order 3 and ascending sort order. On top of that field `salary` itself is indexed with single column index (we have `index = true` in addition to `orderedGroups` declaration). Group `order` does not have to be any particular number, it is needed just to sort fields inside of this group.

```

public class Person implements Serializable {
    /** Indexed in a group index with "salary". */
    @QuerySqlField(orderedGroups={@QuerySqlField.Group(
        name = "age_salary_idx", order = 0, descending = true)})
    private int age;

    /** Indexed separately and in a group index with "age". */
    @QuerySqlField(index = true, orderedGroups={@QuerySqlField.Group(
        name = "age_salary_idx", order = 3)})
    private double salary;
}

```

undefinedNote that annotating a field with `@QuerySqlField.Group` outside of `@QuerySqlField(orderedGroups={…})` will have no effect.

## Configuring SQL Indexes by CacheTypeMetadata

---

Indexes and fields also could be configured with `org.apache.ignite.cache.CacheTypeMetadata` which is convenient for XML configuration with Spring. Please refer to javadoc for details, basically it is equivalent to using `@QuerySqlField` annotation.

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
...
<!-- Cache configuration. -->
<property name="cacheConfiguration">
<list>
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    <property name="name" value="my_cache"/>
    <property name="typeMetadata">
        <!-- Cache types metadata. -->
        <list>
            <bean class="org.apache.ignite.cache.CacheTypeMetadata">
                <!-- Type to query. -->
                <property name="valueType" value=
"org.apache.ignite.examples.datagrid.store.Person"/>
                <!-- Fields to be queried. -->
                <property name="queryFields">
                    <map>
                        <entry key="id" value="java.lang.Long"/>
                        <entry key="orgId" value="java.util.UUID"/>
                        <entry key="firstName" value="java.lang.String"/>
                        <entry key="lastName" value="java.lang.String"/>
                        <entry key="resume" value="java.lang.String"/>
                        <entry key="salary" value="double"/>
                    </map>
                </property>
                <!-- Fields to index in ascending order. -->
                <property name="ascendingFields">
                    <map>
                        <entry key="id" value="java.util.UUID"/>
                        <entry key="orgId" value="java.util.UUID"/>
                        <entry key="salary" value="double"/>
                    </map>
                </property>
            </bean>
        </list>
    ...
</list>
</property>
...
</bean>

```

## How SQL Queries Work

There are two main ways of how query can be processed in Ignite:

1. If you execute the query against **REPLICATED** cache then Ignite assumes that all data available locally and run a simple local SQL query in H2 database engine. The same will happen for **LOCAL** caches.
2. If you execute the query against **PARTITIONED** cache, it work the following way: the query will be parsed and split into multiple map queries and a single reduce query. Then all the map queries are executed on all data nodes of participating caches, providing results to reducing node, which will in turn run reduce query over these intermediate results.

## Using EXPLAIN

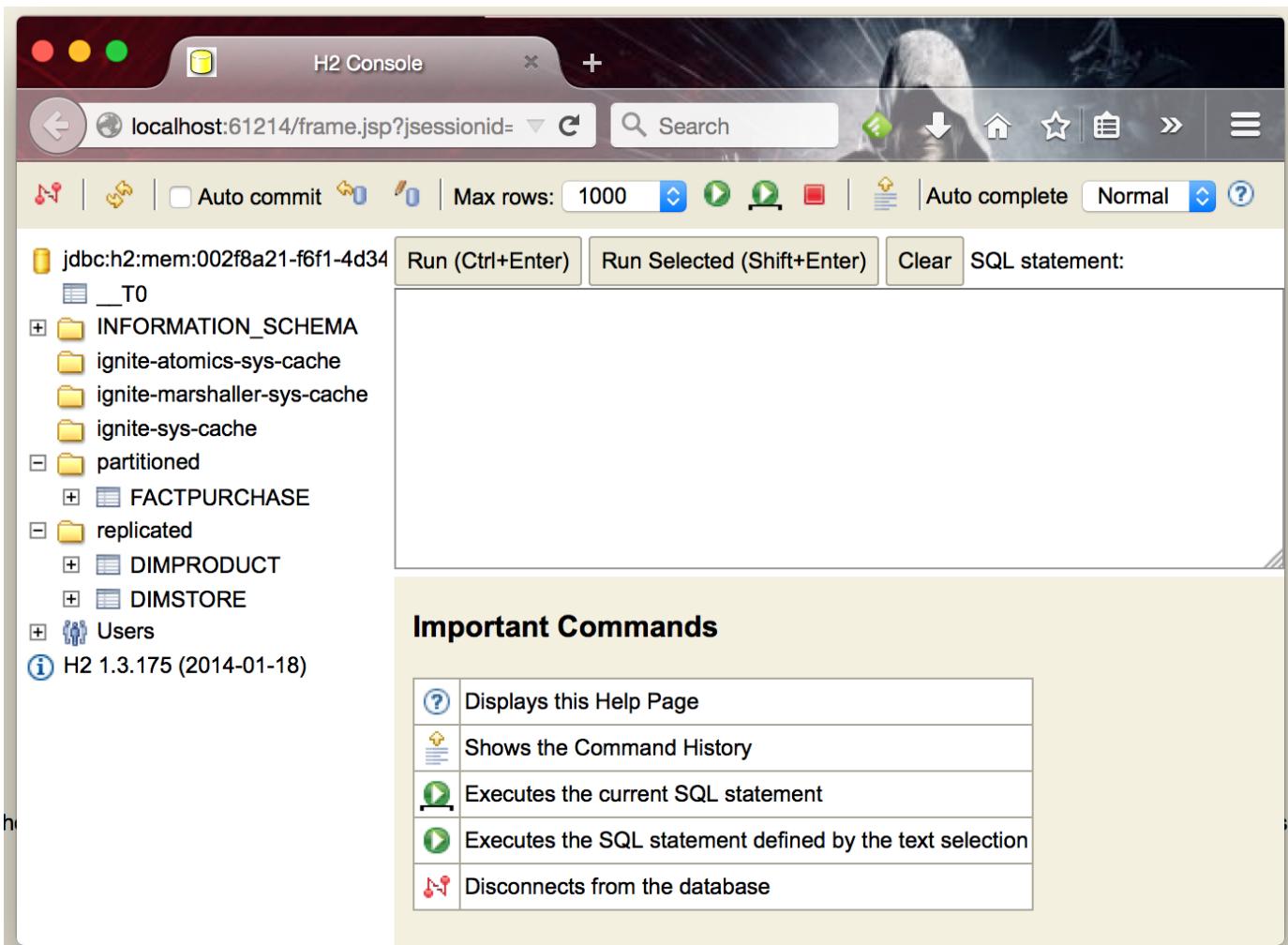
Ignite supports "EXPLAIN ..." syntax in SQL queries and reading execution plans is a main way to analyze query performance in Ignite. Note that plan cursor will contain multiple rows: the last one will contain query for reducing node, others are for map nodes.

```
SqlFieldsQuery sql = new SqlFieldsQuery(  
    "explain select name from Person where age = ?").setArgs(26);  
  
System.out.println(cache.query(sql).getAll());
```

The execution plan itself is generated by H2 as described here:  
[http://www.h2database.com/html/performance.html#explain\\_plan](http://www.h2database.com/html/performance.html#explain_plan)

## Using H2 Debug Console

When developing with Ignite sometimes it is useful to check if your tables and indexes look correctly or run some local queries against embedded in node H2 database. For that purpose Ignite has an ability to start H2 Console. To do that you can start a local node with **IGNITE\_H2\_DEBUG\_CONSOLE** system property or environment variable set to **true**. The console will be opened in your browser. Probably you will need to click **Refresh** button on the Console because it can be opened before database objects initialized.



## Off-heap SQL Indexes

Ignite supports placing index data to off-heap memory. This makes sense for very large datasets when keeping data on heap causes high GC activity and unacceptable response times.

SQL Indexes will reside on heap when property `CacheConfiguration.setOffHeapMaxMemory` is set to `-1`, otherwise off-heap indexes are always used. Note that it is the only property to enable or disable off-heap indexing, while for example `CacheConfiguration.setMemoryMode` does not have any effect on indexing.

To improve performance of SQL queries with off-heap enabled, you can try to increase value of property `CacheConfiguration.setSqlOnheapRowCacheSize` which can be low by default `10 000`.

```

CacheConfiguration<Object, Object> ccfg = new CacheConfiguration<>();

// Set unlimited off-heap memory for cache and enable off-heap indexes.
ccfg.setOffHeapMaxMemory(0);

// Cache entries will be placed on heap and can be evicted to off-heap.
ccfg.setMemoryMode(ONHEAP_TIERED);
ccfg.setEvictionPolicy(new RandomEvictionPolicy(100_000));

// Increase size of SQL on-heap row cache for off-heap indexes.
ccfg.setSqlOnheapRowCacheSize(100_000);

```

## Choosing Indexes

There are multiple things you should consider when choosing indexes for your Ignite application.

- Indexes are not free. They consume memory, also each index needs to be updated separately, thus your cache update performance can be lower if you have more indexes. On top of that optimizer can do more mistakes choosing wrong index to run query.

\*It is a bad strategy to index everything!\*

- Indexes are just sorted data structures. If you define an index on the fields (a,b,c) , the records are sorted first on a, then b, then c.



### Example of Sorted Index.

	A		B		C		1	2	3		1	4	2		1	4	4		2	3	5		2	4	4		2	4	5	
--	---	--	---	--	---	--	---	---	---	--	---	---	---	--	---	---	---	--	---	---	---	--	---	---	---	--	---	---	---	--

Any condition like `a = 1 and b > 3` can be viewed as a bounded range, both bounds can be quickly looked up in index in **log(N)** time, the result will be everything between.

The following conditions will be able to use the index: - `a = ?` - `a = ? and b = ?` - `a = ? and b = ? and c = ?`

Condition `a = ? and c = ?` is no better than `a = ?` from the index point of view. Obviously half-bounded ranges like `a > ?` can be used as well. - Indexes on a single fields are no better than group indexes on multiple fields starting with the same field (index on (a) is no better than (a,b,c)). Thus it is preferable to use group indexes.

## Performance and Usability Considerations

There are few common pitfalls that should be noticed when running SQL queries.

1. If the query is using operator **OR** then it may use indexes not the way you would expect. For example for query `select name from Person where sex='M' and (age = 20 or age = 30)` index on field `age` will not be used even if it is obviously more selective than index on field `sex` and thus is preferable. To workaround this issue you have to rewrite the query with UNION ALL (notice that UNION without ALL will return DISTINCT rows, which will change query semantics and introduce additional performance penalty) like `select name from Person where sex='M' and age = 20 UNION ALL select name from Person where sex='M' and age = 30`. This way indexes will be used correctly.
2. If query contains operator **IN** then it has two problems: it is impossible to provide variable list of parameters (you have to specify the exact list in query like `where id in (?, ?, ?)`, but you can not write it like `where id in ?` and pass array or collection) and this query will not use index. To workaround both problems you can rewrite the query in the following way: `select p.name from Person p join table(id bigint = ?) i on p.id = i.id`. Here you can provide object array (`Object[]`) of any length as a parameter and the query will use index on field `id`. Note that primitive arrays (`int[]`, `long[]`, etc..) can not be used with this syntax, you have to pass array of boxed primitives.

## Continuous Queries

Continuously obtain real-time query results.

---

Continuous queries are good for cases when you want to execute a query and then continue to get notified about the data changes that fall into your query filter.

Continuous queries are supported via `ContinuousQuery` class, which supports the following:

- ==== Initial Query Whenever executing continuous query, you have an option to execute initial query before starting to listen to updates. The initial query can be set via `ContinuousQuery.setInitialQuery(Query)` method and can be of any query type, [Scan](/docs/cache-queries#scan-queries), [SQL](/docs/cache-queries#sql-queries), or [TEXT](/docs/cache-queries#text-queries). This parameter is optional, and if not set, will not be used.
- ==== Remote Filter This filter is executed on the primary node for a given key and evaluates whether the event should be propagated to the listener. If the filter returns `true`, then the listener will be notified, otherwise the event will be skipped. Filtering events on the node on which they have occurred allows to minimize unnecessary network traffic for listener notifications. Remote filter can be set via `ContinuousQuery.setRemoteFilter(CacheEntryEventFilter<K, V>)` method.
- ==== Local Listener Whenever events pass the remote filter, they will be send to the client to notify the local listener there. Local listener is set via `ContinuousQuery.setLocalListener(CacheEntryUpdatedListener<K, V>)` method.

## *continuous query*

```
IgniteCache<Integer, String> cache = ignite.cache("mycache");

// Create new continuous query.
ContinuousQuery<Integer, String> qry = new ContinuousQuery<>();

// Optional initial query to select all keys greater than 10.
qry.setInitialQuery(new ScanQuery<Integer, String>((k, v) -> k > 10));

// Callback that is called locally when update notifications are received.
qry.setLocalListener((evts) ->
    evts.stream().forEach(e -> System.out.println("key=" + e.getKey() + ", val=" + e
.getValue())));

// This filter will be evaluated remotely on all nodes.
// Entry that pass this filter will be sent to the caller.
qry.setRemoteFilter(e -> e.getKey() > 10);

// Execute query.
try (QueryCursor<Cache.Entry<Integer, String>> cur = cache.query(qry)) {
    // Iterate through existing data stored in cache.
    for (Cache.Entry<Integer, String> e : cur)
        System.out.println("key=" + e.getKey() + ", val=" + e.getValue());

    // Add a few more keys and watch a few more query notifications.
    for (int i = 5; i < 15; i++)
        cache.put(i, Integer.toString(i));
}
```

## java7 continuous query

```
IgniteCache<Integer, String> cache = ignite.cache(CACHE_NAME);

// Create new continuous query.
ContinuousQuery<Integer, String> qry = new ContinuousQuery<>();

qry.setInitialQuery(new ScanQuery<Integer, String>(new IgniteBiPredicate<Integer, String>() {
    @Override public boolean apply(Integer key, String val) {
        return key > 10;
    }
}));

// Callback that is called locally when update notifications are received.
qry.setLocalListener(new CacheEntryUpdatedListener<Integer, String>() {
    @Override public void onUpdated(Iterable<CacheEntryEvent<? extends Integer, ? extends String>> evts) {
        for (CacheEntryEvent<Integer, String> e : evts)
            System.out.println("key=" + e.getKey() + ", val=" + e.getValue());
    }
});

// This filter will be evaluated remotely on all nodes.
// Entry that pass this filter will be sent to the caller.
qry.setRemoteFilter(new CacheEntryEventFilter<Integer, String>() {
    @Override public boolean evaluate(CacheEntryEvent<? extends Integer, ? extends String> e) {
        return e.getKey() > 10;
    }
});

// Execute query.
try (QueryCursor<Cache.Entry<Integer, String>> cur = cache.query(qry)) {
    // Iterate through existing data.
    for (Cache.Entry<Integer, String> e : cur)
        System.out.println("key=" + e.getKey() + ", val=" + e.getValue());

    // Add a few more keys and watch more query notifications.
    for (int i = keyCnt; i < keyCnt + 10; i++)
        cache.put(i, Integer.toString(i));
}
```

## Transactions

ACID compliant transactions ensuring guaranteed consistency.

Ignite supports 2 modes for cache operation, **transactional** and **atomic**. In **transactional** mode you are able to group multiple cache operations in a transaction, while **atomic** mode supports multiple atomic operations, one at a time. **Atomic** mode is more light-weight and generally has better performance over **transactional** caches.

However, regardless of which mode you use, as long as your cluster is alive, the data between different cluster nodes must remain consistent. This means that whichever node is being used to retrieve data, it will never get data that has been partially committed or that is inconsistent with other data.

## IgniteTransactions

**IgniteTransactions** interface contains functionality for starting and completing transactions, as well as subscribing listeners or getting metrics.



**Cross-Cache Transactions.** You can combine multiple operations from different caches into one transaction. Note that this allows to update caches of different types, like **REPLICATED** and **PARTITIONED** caches, in one transaction. You can obtain an instance of **IgniteTransactions** as follows:

```
Ignite ignite = Ignition.ignite();
IgniteTransactions transactions = ignite.transactions();
```

Here is an example of how transactions can be performed in Ignite:

```
try (Transaction tx = transactions.txStart()) {
    Integer hello = cache.get("Hello");

    if (hello == 1)
        cache.put("Hello", 11);

    cache.put("World", 22);

    tx.commit();
}
```

## Two-Phase-Commit (2PC)

Ignite utilizes 2PC protocol for its transactions with many one-phase-commit optimizations whenever applicable. Whenever data is updated within a transaction, Ignite will keep transactional state in a local transaction map until **commit()** is called, at which point, if needed, the data is transferred to

participating remote nodes.

For more information on how Ignite 2PC works, you can check out these blogs:

- [Two-Phase-Commit for Distributed In-Memory Caches](#)
- [Two-Phase-Commit for In-Memory Caches - Part II](#)
- [One-Phase-Commit - Fast Transactions For In-Memory Caches](#)



**ACID Compliance.** Ignite provides fully ACID (Atomicity, Consistency, Isolation, Durability) compliant transactions that ensure guaranteed consistency.

## Optimistic and Pessimistic

Whenever **TRANSACTIONAL** atomicity mode is configured, Ignite supports **OPTIMISTIC** and **PESSIMISTIC** concurrency modes for transactions. The main difference is that in **PESSIMISTIC** mode locks are acquired at the time of access, while in **OPTIMISTIC** mode locks are acquired during the **commit** phase.

Ignite also supports the following isolation levels:

- **READ\_COMMITTED** - data is always fetched from the primary node, even if it already has been accessed within the transaction.
- **REPEATABLE\_READ** - data is fetched from the primary node only once on first access and stored in the local transactional map. All consecutive access to the same data is local.
- **SERIALIZABLE** - when combined with **OPTIMISTIC** concurrency, transactions may throw **TransactionOptimisticException** in case of concurrent updates.

```
IgniteTransactions txs = ignite.transactions();

// Start transaction in optimistic mode with repeatable read isolation level.
Transaction tx = txs.txStart(TransactionConcurrency.OPTIMISTIC, TransactionIsolation
.REPEATABLE_READ);
```

## Integration With JTA

Ignite can be configured with a JTA transaction manager lookup class using **TransactionConfiguration#setTxManagerLookupClassName** method. Transaction manager lookup is basically a factory that provides Ignite with an instance of JTA transaction manager. When set, on each cache operation on a transactional cache Ignite will check if there is an ongoing JTA transaction. If JTA transaction is started, Ignite will also start a transaction and will enlist it into JTA transaction using its own internal implementation of **XAResource**. Ignite transaction will be prepared, committed or

rolledback altogether with corresponding JTA transaction. Below is an example of using JTA transaction manager together with Ignite.

```
// Get an instance of JTA transaction manager.  
TMService tms = appCtx.getComponent(TMService.class);  
  
// Get an instance of Ignite cache.  
IgniteCache<String, Integer> cache = cache();  
  
UserTransaction jtaTx = tms.getUserTransaction();  
  
// Start JTA transaction.  
jtaTx.begin();  
  
try {  
    // Do some cache operations.  
    cache.put("key1", 1);  
    cache.put("key2", 2);  
  
    // Commit the transaction.  
    jtaTx.commit();  
}  
finally {  
    // Rollback in a case of exception.  
    if (jtaTx.getStatus() == Status.STATUS_ACTIVE)  
        jtaTx.rollback();  
}
```

## Atomicity Mode

Ignite supports 2 atomicity modes defined in [CacheAtomicityMode](#) enum:

- [TRANSACTIONAL](#)
- [ATOMIC](#)

[TRANSACTIONAL](#) mode enables fully ACID-compliant transactions, however, when only atomic semantics are needed, it is recommended that [ATOMIC](#) mode is used for better performance.

[ATOMIC](#) mode provides better performance by avoiding transactional locks, while still providing data atomicity and consistency. Another difference in [ATOMIC](#) mode is that bulk writes, such as [putAll\(...\)](#) and [removeAll\(...\)](#) methods are no longer executed in one transaction and can partially fail. In case of partial failure, [CachePartialUpdateException](#) will be thrown which will contain a list of keys for which the update failed.



**Performance.** Note that transactions are disabled whenever **ATOMIC** mode is used, which allows to achieve much higher performance and throughput in cases when transactions are not needed.

## Configuration

Atomicity mode is defined in `CacheAtomicityMode` enum and can be configured via `atomicityMode` property of `CacheConfiguration`.

Default atomicity mode is **ATOMIC**.

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="myCache"/>

            <!-- Set atomicity mode, can be ATOMIC or TRANSACTIONAL. -->
            <property name="atomicityMode" value="TRANSACTIONAL"/>
            ...
        </bean>
    </property>

    <!-- Optional transaction configuration. -->
    <property name="transactionConfiguration">
        <bean class="org.apache.ignite.configuration.TransactionConfiguration">
            <!-- Configure TM lookup here. -->
        </bean>
    </property>
</bean>
```

```

CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

cacheCfg.setAtomicityMode(CacheAtomicityMode.ATOMIC);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Optional transaction configuration. Configure TM lookup here.
TransactionConfiguration txCfg = new TransactionConfiguration();

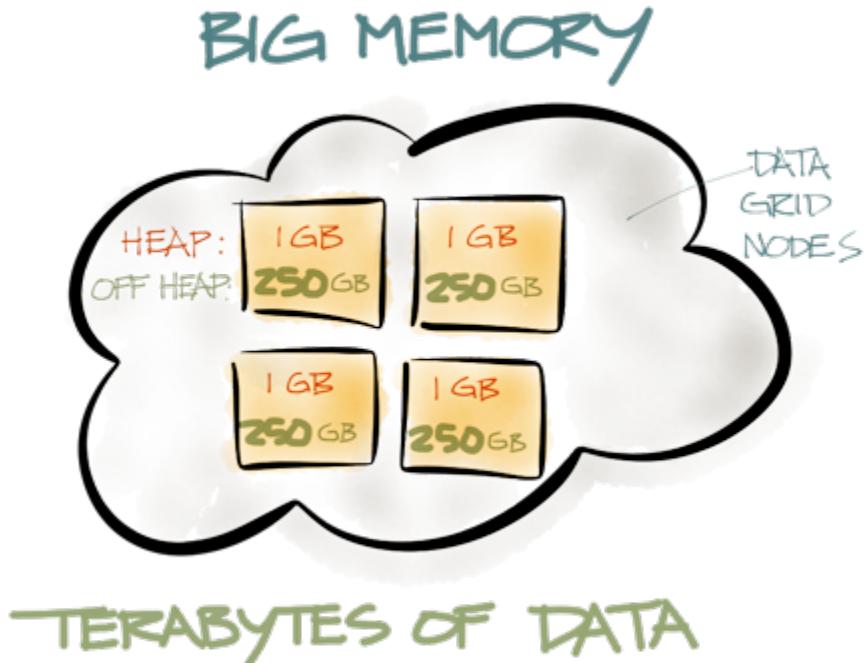
cfg.setTransactionConfiguration(txCfg);

// Start Ignite node.
Ignition.start(cfg);

```

## Off-Heap Memory

Off-Heap memory allows your cache to overcome lengthy JVM Garbage Collection (GC) pauses when working with large heap sizes by caching data outside of main Java Heap space, but still in RAM.





**Off-Heap Indexes.** Note that when off-heap memory is configured, Ignite will also store query indexes off-heap as well. This means that indexes will not take any portion of on-heap memory.



**Off-Heap Memory vs. Multiple Processes.** You can also manage GC pauses by starting multiple processes with smaller heap on the same physical server. However, such approach is wasteful when using REPLICATED caches as we will end up with caching identical **replicated** data for every started JVM process.

## Tiered Off-Heap Storage

Ignite provides tiered storage model, where data can be stored and moved between **on-heap**, **off-heap**, and **swap space**. Going up the tier provides more data storage capacity, with gradual increase in latency.

Ignite provides three types of memory modes, defined in [CacheMemoryMode](#), for storing cache entries, supporting tiered storage model:

Memory Mode	Description
ONHEAP_TIERED	Store entries on-heap and evict to off-heap and optionally to swap.
OFFHEAP_TIERED	<a href="#">OFFHEAP_VALUES</a>
Store entries off-heap, bypassing on-heap and optionally evicting to swap.	Store keys on-heap and values off-heap.

Cache can be configured to use any of the three modes by setting the [memoryMode](#) configuration property of [CacheConfiguration](#), as described below.

### ONHEAP\_TIERED

In Ignite, **ONHEAP\_TIERED** is the default memory mode, where all cache entries are stored on-heap. Entries can be moved from on-heap to off-heap storage and later to swap space, if one is configured.

To configure **ONHEAP\_TIERED** memory mode, you need to:

1. Set [memoryMode](#) property of [CacheConfiguration](#) to **ONHEAP\_TIERED**.
2. Enable off-heap memory (optionally).
3. Configure **eviction policy** for on-heap memory.

```

<bean class="org.apache.ignite.configuration.CacheConfiguration">
    ...
    <!-- Store cache entries on-heap. -->
    <property name="memoryMode" value="ONHEAP_TIERED"/>

    <!-- Enable Off-Heap memory with max size of 10 Gigabytes (0 for unlimited). -->
    <property name="offHeapMaxMemory" value="#{10 * 1024L * 1024L * 1024L}"/>

    <!-- Configure eviction policy. -->
    <property name="evictionPolicy">
        <bean class="org.apache.ignite.cache.eviction.fifo.CacheFifoEvictionPolicy">
            <!-- Evict to off-heap after cache size reaches maxSize. -->
            <property name="maxSize" value="100000"/>
        </bean>
    </property>
    ...
</bean>

```

```

CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setMemoryMode(CacheMemoryMode.ONHEAP_TIERED);

// Set off-heap memory to 10GB (0 for unlimited)
cacheCfg.setOffHeapMaxMemory(10 * 1024L * 1024L * 1024L);

CacheFifoEvictionPolicy evctPolicy = new CacheFifoEvictionPolicy();

// Store only 100,000 entries on-heap.
evctPolicy.setMaxSize(100000);

cacheCfg.setEvictionPolicy(evctPolicy);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);

```



**Eviction Policy.** Note that if you do not enable eviction policy in ONHEAP\_TIERED mode, data will never be moved from on-heap to off-heap memory.

## OFFHEAP\_TIERED

This memory mode allows you to configure your cache to store entries directly into off-heap storage, bypassing on-heap memory. Since all entries are stored off-heap, there is no need to explicitly configure an eviction policy. If off-heap storage size is exceeded (0 for unlimited), then LRU eviction policy is used to evict entries from off-heap store and optionally moving them to swap space, if one is configured.

To configure `OFFHEAP_TIERED` memory mode, you need to:

1. Set `memoryMode` property of `CacheConfiguration` to `OFFHEAP_TIERED`.
2. Enable off-heap memory (optionally).

```
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    ...
    <!-- Always store cache entries in off-heap memory. -->
    <property name="memoryMode" value="OFFHEAP_TIERED"/>

    <!-- Enable Off-Heap memory with max size of 10 Gigabytes (0 for unlimited). -->
    <property name="offHeapMaxMemory" value="#{10 * 1024L * 1024L * 1024L}"/>
    ...
</bean>
```

```
CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setMemoryMode(CacheMemoryMode.OFFHEAP_TIERED);

// Set off-heap memory to 10GB (0 for unlimited)
cacheCfg.setOffHeapMaxMemory(10 * 1024L * 1024L * 1024L);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

## OFFHEAP\_VALUES

Setting this memory mode allows you to store keys on-heap and values off-heap. This memory mode is useful when keys are small and values are large.

To configure `OFFHEAP_VALUES` memory mode, you need to:

1. Set `memoryMode` property of `CacheConfiguration` to `OFFHEAP_VALUES`.
2. Enable off-heap memory.
3. Configure **eviction policy** for on-heap memory (optionally).

```
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    ...
    <!-- Always store cache entries in off-heap memory. -->
    <property name="memoryMode" value="OFFHEAP_VALUES"/>

    <!-- Enable Off-Heap memory with max size of 10 Gigabytes (0 for unlimited). -->
    <property name="offHeapMaxMemory" value="#{10 * 1024L * 1024L * 1024L}"/>
    ...
</bean>
```

```
CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setMemoryMode(CacheMemoryMode.OFFHEAP_VALUES);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

## Swap Space

Whenever your data set exceeds the limits of on-heap and off-heap memory, you can configure swap space in which case Ignite will evict entries to the disk instead of discarding them.



**Swap Space Performance.** Since swap space is on-disk, it is significantly slower than on-heap or off-heap memory.

```
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    ...
    <!-- Enable swap. -->
    <property name="swapEnabled" value="true"/>
    ...
</bean>
```

```
CacheConfiguration cacheCfg = new CacheConfiguration();
cacheCfg.setSwapEnabled(true);

IgniteConfiguration cfg = new IgniteConfiguration();
cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

## Affinity Collocation

Collocate compute with data or data with data.

Given that the most common ways to cache data is in **PARTITIONED** caches, collocating compute with data or data with data can significantly improve performance and scalability of your application.

### Collocate Data with Data

In many cases it is beneficial to collocate different cache keys together if they will be accessed together. Quite often your business logic will require access to more than one cache key. By collocating them together you can make sure that all keys with the same **affinityKey** will be cached on the same processing node, hence avoiding costly network trips to fetch data from remote nodes.

For example, let's say you have **Person** and **Company** objects and you want to collocate **Person** objects with **Company** objects for which this person works. To achieve that, cache key used to cache **Person** objects should have a field or method annotated with **@AffinityKeyMapped** annotation, which will provide the value of the company key for collocation. For convenience, you can also optionally use **AffinityKey** class

## using PersonKey

```
public class PersonKey {  
    // Person ID used to identify a person.  
    private String personId;  
  
    // Company ID which will be used for affinity.  
    @AffinityKeyMapped  
    private String companyId;  
    ...  
}  
  
// Instantiate person keys with the same company ID which is used as affinity key.  
Object personKey1 = new PersonKey("myPersonId1", "myCompanyId");  
Object personKey2 = new PersonKey("myPersonId2", "myCompanyId");  
  
Person p1 = new Person(personKey1, ...);  
Person p2 = new Person(personKey2, ...);  
  
// Both, the company and the person objects will be cached on the same node.  
cache.put("myCompanyId", new Company(...));  
cache.put(personKey1, p1);  
cache.put(personKey2, p2);
```

## using AffinityKey

```
Object personKey1 = new AffinityKey("myPersonId1", "myCompanyId");  
Object personKey2 = new AffinityKey("myPersonId2", "myCompanyId");  
  
Person p1 = new Person(personKey1, ...);  
Person p2 = new Person(personKey2, ...);  
  
// Both, the company and the person objects will be cached on the same node.  
cache.put("myCompanyId", new Company(...));  
cache.put(personKey1, p1);  
cache.put(personKey2, p2);
```



**SQL Joins.** When performing [SQL distributed joins](/docs/cache-queries#sql-queries) over data residing in partitioned caches, you must make sure that the join-keys are collocated.

## Collocating Compute with Data

It is also possible to route computations to the nodes where the data is cached. This concept is known

as Collocation Of Computations And Data. It allows to route whole units of work to a certain node.

To collocate compute with data you should use `IgniteCompute.affinityRun(...)` and `IgniteCompute.affinityCall(...)` methods.

Here is how you can collocate your computation with the same cluster node on which company and persons from the example above are cached.

#### *affinityRun*

```
String companyId = "myCompanyId";  
  
// Execute Runnable on the node where the key is cached.  
ignite.compute().affinityRun("myCache", companyId, () -> {  
    Company company = cache.get(companyId);  
  
    // Since we collocated persons with the company in the above example,  
    // access to the persons objects is local.  
    Person person1 = cache.get(personKey1);  
    Person person2 = cache.get(personKey2);  
    ...  
});
```

#### *java7 affinityRun*

```
final String companyId = "myCompanyId";  
  
// Execute Runnable on the node where the key is cached.  
ignite.compute().affinityRun("myCache", companyId, new IgniteRunnable() {  
    @Override public void run() {  
        Company company = cache.get(companyId);  
  
        Person person1 = cache.get(personKey1);  
        Person person2 = cache.get(personKey2);  
        ...  
    }  
});
```

## IgniteCompute vs EntryProcessor

Both, `IgniteCompute.affinityRun(...)` and `IgniteCache.invoke(...)` methods offer ability to collocate compute and data. The main difference is that `invoke(...)` methods is atomic and executes while holding a lock on a key. You should not access other keys from within the `EntryProcessor` logic as it may cause a deadlock.

'affinityRun(...)' and 'affinityCall(...)', on the other hand, do not hold any locks. For example, it is absolutely legal to start multiple transactions or execute cache queries from these methods without worrying about deadlocks. In this case Ignite will automatically detect that the processing is collocated and will employ a light-weight 1-Phase-Commit optimization for transactions (instead of 2-Phase-Commit).



See [[JCache EntryProcessor](#)](/docs/jcache#entryprocessor) documentation for more information about `IgniteCache.invoke(...)` method.

## Persistent Store

Write-through or read-through data to and from persistent storage.

JCache specification comes with APIs for `javax.cache.integration.CacheLoader`.

While Ignite allows you to configure the `CacheLoader` and `CacheWriter` separately, it is very awkward to implement a transactional store within 2 separate classes, as multiple `load` and `put` operations have to share the same connection within the same transaction. To mitigate that, Ignite provides `org.apache.ignite.cache.store.CacheStore` interface which extends both, `CacheLoader` and `CacheWriter`.



**Transactions.** `CacheStore` is fully transactional and automatically merges into the ongoing cache transaction.



**CacheJdbcPojoStore.** Ignite ships with its own `CacheJdbcPojoStore` which automatically maps Java POJOs to database schema. See [Automatic Persistence](#) for more information.

## Read-Through and Write-Through

Providing proper cache store implementation is important whenever read-through or write-through behavior is desired. Read-through means that data will be read from persistent store whenever it's not available in cache, and write-through means that data will be automatically persisted whenever it is updated in cache. All read-through and write-through operations will participate in overall cache transaction and will be committed or rolled back as a whole.

To configure read-through and write-through, you need to implement `CacheStore` interface and set `cacheStoreFactory` as well as `readThrough` and `writeThrough` properties of `CacheConfiguration`, as shown in examples below.

## Write-Behind Caching

---

In a simple write-through mode each cache put and remove operation will involve a corresponding request to the persistent storage and therefore the overall duration of the cache update might be relatively long. Additionally, an intensive cache update rate can cause an extremely high storage load.

For such cases, Ignite offers an option to perform an asynchronous persistent store update also known as **write-behind**. The key concept of this approach is to accumulate updates and then asynchronously flush them to persistent store as a bulk operation. The actual data persistence can be triggered by time-based events (the maximum time that data entry can reside in the queue is limited), by queue-size events (the queue is flushed when its size reaches some particular point), or by using both of them in combination in which case either event will trigger the flush.



**Update Sequence.** With the write-behind approach only the last update to an entry will be written to the underlying storage. If cache entry with key `key1` is sequentially updated with values `value1`, `value2`, and `value3` respectively, then only single store request for `(key1, value3)` pair will be propagated to the persistent storage.



**Update Performance.** Batch store operations are usually more efficient than a sequence of single store operations, so one can exploit this feature by enabling batch operations in write-behind mode. Update sequences of similar types (put or remove) can be grouped to a single batch. For example, sequential cache puts of `(key1, value1)`, `(key2, value2)`, `(key3, value3)` will be batched into a single `CacheStore.putAll(...)` operation. Write-behind caching can be enabled via `CacheConfiguration.setWriteBehindEnabled(boolean)` configuration property. See [Configuration](#) section below for a full list of configuration properties that allow to customize the behavior of write-behind caching.

## CacheStore

---

`CacheStore` interface in Ignite is used to write and load data to and from the underlying data store. In addition to standard JCache loading and storing methods, it also introduces end-of-transaction demarcation and ability to bulk load a cache from the underlying data store.

### loadCache()

`CacheStore.loadCache()` method allows for cache loading even without passing all the keys that need to be loaded. It is generally used for hot-loading the cache on startup, but can be also called at any point after the cache has been started.

`IgniteCache.loadCache()` method will delegate to `CacheStore.loadCache()` method on every cluster member that is running the cache. To invoke loading only on the local cluster node, use

`IgniteCache.localLoadCache()` method.



In case of partitioned caches, keys that are not mapped to this node, either as primary or backups, will be automatically discarded by the cache. ===== `load()`, `write()`, `delete()` Methods `load()`, `write()`, and `delete()` on the `CacheStore` are called whenever methods `get()`, `put()`, and `remove()` are called correspondingly on the `IgniteCache` interface. These methods are used to enable **read-through** and **write-through** behavior when working with individual cache entries.

## loadAll(), writeAll(), deleteAll()

Methods `loadAll()`, `writeAll()`, and `deleteAll()` on the `CacheStore` are called whenever methods `getAll()`, `putAll()`, and `removeAll()` are called correspondingly on the `IgniteCache` interface. These methods are used to enable **read-through** and **write-through** behavior when working with multiple cache entries and should generally be implemented using batch operations to provide better performance.



. `CacheStoreAdapter` provides default implementation for `loadAll()`, `writeAll()`, and `deleteAll()` methods which simply iterates through all keys one by one. ===== `sessionEnd()` Ignite has a concept of store session which may span more than one cache store operation. Sessions are especially useful when working with transactions.

In case of `ATOMIC` caches, method `sessionEnd()` is called after completion of each `CacheStore` method. In case of `TRANSACTIONAL` caches, `sessionEnd()` is called at the end of each transaction, which allows to either commit or rollback multiple operations on the underlying persistent store.



`CacheStoreAdapter` provides default empty implementation of `sessionEnd()` method.

## CacheStoreSession

The main purpose of cache store session is to hold the context between multiple store invocations whenever `CacheStore` is used in a cache transaction. For example, if using JDBC, you can store the ongoing database connection via `CacheStoreSession.attach()` method. You can then commit this connection in the `CacheStore#sessionEnd(boolean)` method.

`CacheStoreSession` can be injected into your cache store implementation via `@GridCacheStoreSessionResource` annotation.

## CacheStore Example

Below are a couple of different possible cache store implementations. Note that transactional

implementation works with and without transactions.

#### *jdbc non-transactional*

```
public class CacheJdbcPersonStore extends CacheStoreAdapter<Long, Person> {
    // This method is called whenever "get(...)" methods are called on IgniteCache.
    @Override public Person load(Long key) {
        try (Connection conn = connection()) {
            try (PreparedStatement st = conn.prepareStatement("select * from PERSONS where id=?")) {
                st.setLong(1, key);

                ResultSet rs = st.executeQuery();

                return rs.next() ? new Person(rs.getLong(1), rs.getString(2), rs.getString(3)) :
null;
            }
        } catch (SQLException e) {
            throw new CacheLoaderException("Failed to load: " + key, e);
        }
    }

    // This method is called whenever "put(...)" methods are called on IgniteCache.
    @Override public void write(Cache.Entry<Long, Person> entry) {
        try (Connection conn = connection()) {
            // Syntax of MERGE statement is database specific and should be adopted for your
            // database.
            // If your database does not support MERGE statement then use sequentially update,
            // insert statements.
            try (PreparedStatement st = conn.prepareStatement(
                "merge into PERSONS (id, firstName, lastName) key (id) VALUES (?, ?, ?)")) {
                for (Cache.Entry<Long, Person> entry : entries) {
                    Person val = entry.getValue();

                    st.setLong(1, entry.getKey());
                    st.setString(2, val.getFirstName());
                    st.setString(3, val.getLastName());

                    st.executeUpdate();
                }
            }
        } catch (SQLException e) {
            throw new CacheWriterException("Failed to write [key=" + key + ", val=" + val + ']',
', e);
        }
    }
}
```

```

// This method is called whenever "remove(...)" methods are called on IgniteCache.
@Override public void delete(Object key) {
    try (Connection conn = connection()) {
        try (PreparedStatement st = conn.prepareStatement("delete from PERSONS where id=?"))
        {
            st.setLong(1, (Long)key);

            st.executeUpdate();
        }
    }
    catch (SQLException e) {
        throw new CacheWriterException("Failed to delete: " + key, e);
    }
}

// This method is called whenever "loadCache()" and "localLoadCache()"
// methods are called on IgniteCache. It is used for bulk-loading the cache.
// If you don't need to bulk-load the cache, skip this method.
@Override public void loadCache(IgniteBiInClosure<Long, Person> clo, Object... args) {
    if (args == null || args.length == 0 || args[0] == null)
        throw new CacheLoaderException("Expected entry count parameter is not provided.");

    final int entryCnt = (Integer)args[0];

    try (Connection conn = connection()) {
        try (PreparedStatement st = conn.prepareStatement("select * from PERSONS")) {
            try (ResultSet rs = st.executeQuery()) {
                int cnt = 0;

                while (cnt < entryCnt && rs.next()) {
                    Person person = new Person(rs.getLong(1), rs.getString(2), rs.getString(3));

                    clo.apply(person.getId(), person);

                    cnt++;
                }
            }
        }
    }
    catch (SQLException e) {
        throw new CacheLoaderException("Failed to load values from cache store.", e);
    }
}

// Open JDBC connection.
private Connection connection() throws SQLException {
    // Open connection to your RDBMS systems (Oracle, MySQL, Postgres, DB2, Microsoft

```

```

SQL, etc.)
    // In this example we use H2 Database for simplification.
    Connection conn = DriverManager.getConnection("jdbc:h2:mem:example;DB_CLOSE_DELAY=-1
");
    conn.setAutoCommit(true);

    return conn;
}
}

```

### *jdbc transactional*

```

public class CacheJdbcPersonStore extends CacheStoreAdapter<Long, Person> {
    /** Auto-injected store session. */
    @CacheStoreSessionResource
    private CacheStoreSession ses;

    // Complete transaction or simply close connection if there is no transaction.
    @Override public void sessionEnd(boolean commit) {
        try (Connection conn = ses.getAttached()) {
            if (conn != null && ses.isWithinTransaction()) {
                if (commit)
                    conn.commit();
                else
                    conn.rollback();
            }
        }
        catch (SQLException e) {
            throw new CacheWriterException("Failed to end store session.", e);
        }
    }

    // This method is called whenever "get(...)" methods are called on IgniteCache.
    @Override public Person load(Long key) {
        try (Connection conn = connection()) {
            try (PreparedStatement st = conn.prepareStatement("select * from PERSONS where
id=?")) {
                st.setLong(1, key);

                ResultSet rs = st.executeQuery();

                return rs.next() ? new Person(rs.getLong(1), rs.getString(2), rs.getString(3)) :
null;
            }
        }
        catch (SQLException e) {

```

```

        throw new CacheLoaderException("Failed to load: " + key, e);
    }
}

// This method is called whenever "put(...)" methods are called on IgniteCache.
@Override public void write(Cache.Entry<Long, Person> entry) {
    try (Connection conn = connection()) {
        // Syntax of MERGE statement is database specific and should be adopted for your
        database.

        // If your database does not support MERGE statement then use sequentially update,
        insert statements.

        try (PreparedStatement st = conn.prepareStatement(
            "merge into PERSONS (id, firstName, lastName) key (id) VALUES (?, ?, ?)")) {
            for (Cache.Entry<Long, Person> entry : entries) {
                Person val = entry.getValue();

                st.setLong(1, entry.getKey());
                st.setString(2, val.getFirstName());
                st.setString(3, val.getLastName());

                st.executeUpdate();
            }
        }
    }
    catch (SQLException e) {
        throw new CacheWriterException("Failed to write [key=" + key + ", val=" + val + ']',
            e);
    }
}

// This method is called whenever "remove(...)" methods are called on IgniteCache.
@Override public void delete(Object key) {
    try (Connection conn = connection()) {
        try (PreparedStatement st = conn.prepareStatement("delete from PERSONS where id=?")) {
            st.setLong(1, (Long)key);

            st.executeUpdate();
        }
    }
    catch (SQLException e) {
        throw new CacheWriterException("Failed to delete: " + key, e);
    }
}

// This method is called whenever "loadCache()" and "localLoadCache()"
// methods are called on IgniteCache. It is used for bulk-loading the cache.
// If you don't need to bulk-load the cache, skip this method.

```

```

@Override public void loadCache(IgniteBiInClosure<Long, Person> clo, Object... args) {
    if (args == null || args.length == 0 || args[0] == null)
        throw new CacheLoaderException("Expected entry count parameter is not provided.");

    final int entryCnt = (Integer)args[0];

    try (Connection conn = connection()) {
        try (PreparedStatement st = conn.prepareStatement("select * from PERSONS")) {
            try (ResultSet rs = st.executeQuery()) {
                int cnt = 0;

                while (cnt < entryCnt && rs.next()) {
                    Person person = new Person(rs.getLong(1), rs.getString(2), rs.getString(3));

                    clo.apply(person.getId(), person);

                    cnt++;
                }
            }
        }
    }
    catch (SQLException e) {
        throw new CacheLoaderException("Failed to load values from cache store.", e);
    }
}

// Opens JDBC connection and attaches it to the ongoing
// session if within a transaction.
private Connection connection() throws SQLException {
    if (ses.isWithinTransaction()) {
        Connection conn = ses.getAttached();

        if (conn == null) {
            conn = openConnection(false);

            // Store connection in the session, so it can be accessed
            // for other operations within the same transaction.
            ses.attach(conn);
        }
    }

    return conn;
}
// Transaction can be null in case of simple load or put operation.
else
    return openConnection(true);
}

// Opens JDBC connection.

```

```

private Connection openConnection(boolean autocommit) throws SQLException {
    // Open connection to your RDBMS systems (Oracle, MySQL, Postgres, DB2, Microsoft
    // SQL, etc.)
    // In this example we use H2 Database for simplification.
    Connection conn = DriverManager.getConnection("jdbc:h2:mem:example;DB_CLOSE_DELAY=-1
");

    conn.setAutoCommit(autocommit);

    return conn;
}
}

```

### *jdbc bulk operations*

```

public class CacheJdbcPersonStore extends CacheStore<Long, Person> {
    // Skip single operations and open connection methods.
    // You can copy them from jdbc non-transactional or jdbc transactional examples.
    ...

    // This method is called whenever "getAll(...)" methods are called on IgniteCache.
    @Override public Map<K, V> loadAll(Iterable<Long> keys) {
        try (Connection conn = connection()) {
            try (PreparedStatement st = conn.prepareStatement(
                "select firstName, lastName from PERSONS where id=?")) {
                Map<K, V> loaded = new HashMap<>();

                for (Long key : keys) {
                    st.setLong(1, key);

                    try(ResultSet rs = st.executeQuery()) {
                        if (rs.next())
                            loaded.put(key, new Person(key, rs.getString(1), rs.getString(2)));
                    }
                }

                return loaded;
            }
        }
        catch (SQLException e) {
            throw new CacheLoaderException("Failed to loadAll: " + keys, e);
        }
    }

    // This method is called whenever "putAll(...)" methods are called on IgniteCache.
    @Override public void writeAll(Collection<Cache.Entry<Long, Person>> entries) {
        try (Connection conn = connection()) {

```

```

// Syntax of MERGE statement is database specific and should be adopted for your
database.

// If your database does not support MERGE statement then use sequentially update,
insert statements.

try (PreparedStatement st = conn.prepareStatement(
    "merge into PERSONS (id, firstName, lastName) key (id) VALUES (?, ?, ?)") {
    for (Cache.Entry<Long, Person> entry : entries) {
        Person val = entry.getValue();

        st.setLong(1, entry.getKey());
        st.setString(2, val.getFirstName());
        st.setString(3, val.getLastName());

        st.addBatch();
    }

    st.executeBatch();
}
}

catch (SQLException e) {
    throw new CacheWriterException("Failed to writeAll: " + entries, e);
}
}

// This method is called whenever "removeAll(...)" methods are called on IgniteCache.
@Override public void deleteAll(Collection<Long> keys) {
    try (Connection conn = connection()) {
        try (PreparedStatement st = conn.prepareStatement("delete from PERSONS where id=?")) {
            for (Long key : keys) {
                st.setLong(1, key);

                st.addBatch();
            }

            st.executeBatch();
        }
    }

    catch (SQLException e) {
        throw new CacheWriterException("Failed to deleteAll: " + keys, e);
    }
}
}

```

## Configuration

Following configuration parameters can be used to enable and configure **write-behind** caching via `CacheConfiguration`:

Setter Method	Description	Default
<code>setWriteBehindEnabled(boolean)</code>	<code>setWriteBehindFlushSize(int)</code>	<code>setWriteBehindFlushFrequency(long)</code>
<code>setWriteBehindFlushThreadCount(int)</code>	<code>setWriteBehindBatchSize(int)</code>	Sets flag indicating whether write-behind is enabled.
Maximum size of the write-behind cache. If cache size exceeds this value, all cached items are flushed to the cache store and write cache is cleared. If this value is 0, then flush is performed according to the flush frequency interval. Note that you cannot set both, flush size and flush frequency, to 0.	Frequency with which write-behind cache is flushed to the cache store in milliseconds. This value defines the maximum time interval between object insertion/deletion from the cache and the moment when corresponding operation is applied to the cache store. If this value is 0, then flush is performed according to the flush size. Note that you cannot set both, flush size and flush frequency, to 0.	Number of threads that will perform cache flushing.
Maximum batch size for write-behind cache store operations.	false	10240
5000 milliseconds	1	512

`CacheStore` interface can be set on `IgniteConfiguration` via a `Factory` in much the same way like `CacheLoader` and `CacheWriter` are being set.



For distributed cache configuration `Factory` should be serializable.

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.CacheConfiguration">
                ...
                <property name="cacheStoreFactory">
                    <bean class="javax.cache.configuration.FactoryBuilder$SingletonFactory">
                        <constructor-arg>
                            <bean class="foo.bar.MyPersonStore">
                                ...
                                </bean>
                            </constructor-arg>
                        </bean>
                    </property>
                    ...
                </bean>
            </list>
        </property>
    ...
</bean>
```

```

IgniteConfiguration cfg = new IgniteConfiguration();

CacheConfiguration<Long, Person> cacheCfg = new CacheConfiguration<>();

CacheStore<Long, Person> store;

store = new MyPersonStore();

cacheCfg.setCacheStoreFactory(new FactoryBuilder.SingletonFactory<>(store));
cacheCfg.setReadThrough(true);
cacheCfg.setWriteThrough(true);

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

## CacheJdbcBlobStore

**CacheJdbcBlobStore** implementation backed by JDBC. This implementation stores objects in underlying database in **BLOB** format. The **Store** will create table **ENTRIES** in the database to store data. Table will

have key and val fields. If custom DDL and DML statements are provided, table and field names have to be consistent for all statements and sequence of parameters have to be preserved.

Use [CacheJdbcBlobStoreFactory](#) factory to pass [CacheJdbcBlobStore](#) to [CacheConfiguration](#).

*Spring*

```
<bean id= "simpleDataSource" class="org.h2.jdbcx.JdbcDataSource"/>

<bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.CacheConfiguration">
                ...
                <property name="cacheStoreFactory">
                    <bean class="org.apache.ignite.cache.store.jdbc.CacheJdbcBlobStoreFactory">
                        <property name="user" value = "user" />
                        <property name="dataSourceBean" value = "simpleDataSource" />
                    </bean>
                </property>
            </bean>
        </list>
    </property>
    ...
</bean>
```

## CacheJdbcPojoStore

---

[CacheJdbcPojoStore](#) of CacheStore backed by JDBC and POJO via reflection. This implementation stores objects in underlying database using java beans mapping description via reflection.

Use [CacheJdbcPojoStoreFactory](#) factory to pass [CacheJdbcPojoStore](#) to [CacheConfiguration](#).

```
<bean id= "simpleDataSource" class="org.h2.jdbcx.JdbcDataSource"/>

<bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.CacheConfiguration">
                ...
                <property name="cacheStoreFactory">
                    <bean class="org.apache.ignite.cache.store.jdbc.CacheJdbcPojoStoreFactory">
                        <property name="dataSourceBean" value = "simpleDataSource" />
                    </bean>
                </property>
            </bean>
        </list>
    </property>
</bean>
```

## CacheHibernateBlobStore

**CacheHibernateBlobStore** implementation backed by Hibernate. This implementation stores objects in underlying database in **BLOB** format.

Use **CacheHibernateBlobStoreFactory** factory to pass **CacheHibernateBlobStore** to **CacheConfiguration**.

```
<bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.CacheConfiguration">
                <bean class=
"org.apache.ignite.cache.store.hibernate.CacheHibernateBlobStoreFactory">
                    <property name="hibernateProperties">
                        <props>
                            <prop key="connection.url">jdbc:h2:mem:</prop>
                            <prop key="hbm2ddl.auto">update</prop>
                            <prop key="show_sql">true</prop>
                        </props>
                    </property>
                </bean>
            </list>
        </property>
    ...
</bean>
```

## Automatic Persistence

Automatically read-through and write-through your domain model to and from database.

Ignite ships with its own database schema mapping wizard which provides automatic support for integrating with persistence stores. This utility automatically connects to the underlying database and generates all the required XML OR-mapping configuration and Java domain model POJOs.

Ignite also ships with `org.apache.ignite.cache.store.jdbc.CacheJdbcPojoStore`, which is out-of-the-box JDBC implementation of the [\[Persistence Store\]](#) interface, and automatically handles all the write-through and read-through logic.

### Database Schema Import

To start the wizard for generating database schema mapping, execute `bin/ignite-schema-import.sh` script:

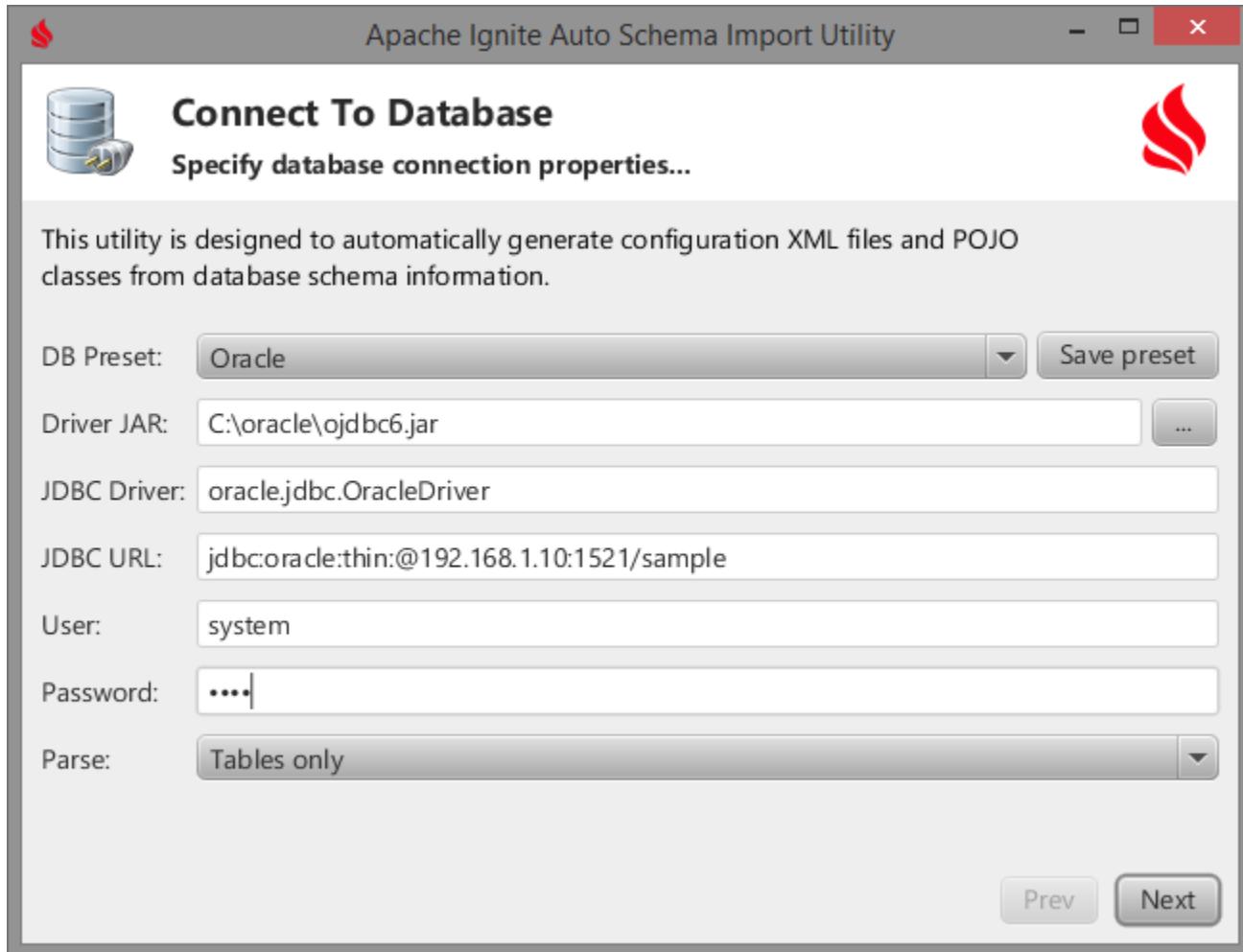
*Shell*

```
$ bin/ignite-schema-import.sh
```

This command will bring up a UI wizard which will take you through a couple of screens.

## Connect To Database

The first screen will ask you for database connectivity settings. Note that JDBC drivers **are not supplied** with the utility and should be provided separately.



## Generate XML Configuration and POJOs

The second screen allows to map database tables to domain model classes and automatically generates XML configurations and POJOs.

Apache Ignite Auto Schema Import Utility

## Generate XML And POJOs

jdbc:oracle:thin:@192.168.1.10:1521/sample

Schema / Table	Key Class Name	Value Class Name
<input checked="" type="checkbox"/> PUBLIC		
<input checked="" type="checkbox"/> BONUS	BonusKey	Bonus
<input checked="" type="checkbox"/> ORGANIZATION	OrganizationKey	Organization
<input checked="" type="checkbox"/> PERSON	PersonKey	Person
<input checked="" type="checkbox"/> PERSON_BONUS	PersonBonusKey	PersonBonus

Use	Key	AK	DB Name	DB Type	Java Name	Java Type
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	ID	INTEGER	id	int
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	ORG_ID	INTEGER	orgId	java.lang.Integer
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NAME	VARCHAR	name	java.lang.String

Package: org.apache.ignite.examples.datagrid.store.model

Output Folder: C:\Ignite\out

Include key fields into value POJOs

Generate constructors for POJOs

Write all configurations to a single XML file

Rename "Key class name", "Value class name" or "Java name" for selected tables

## Generated Artifacts

The utility generates the following artifacts:

- Java POJO key and value classes.
- XML `CacheTypeMetadata` configuration.
- `ConfigurationSnippet.java` (alternative to XML).

After you exit from the wizard, you should 1. Copy generated POJO java classes to your project source folder. 2. Copy XML declaration of `CacheTypeMetadata` to your Ignite XML configuration file under appropriate `CacheConfiguration` root. 3. Use `ConfigurationSnippet.java` in your Ignite initialization logic.

## POJO key class

```
/**  
 * PersonKey definition.  
  
 *  
 * Code generated by Apache Ignite Schema Import utility: 03/03/2015.  
 */  
public class PersonKey implements Serializable {  
    /** */  
    private static final long serialVersionUID = 0L;  
  
    /** Value for id. */  
    private int id;  
  
    /**  
     * Gets id.  
     *  
     * @return Value for id.  
     */  
    public int getId() {  
        return id;  
    }  
  
    /**  
     * Sets id.  
     *  
     * @param id New value for id.  
     */  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    /** {@inheritDoc} */  
    @Override public boolean equals(Object o) {  
        if (this == o)  
            return true;
```

```

    if (!(o instanceof PersonKey))
        return false;

    PersonKey that = (PersonKey)o;

    if (id != that.id)
        return false;

    return true;
}

/** {@inheritDoc} */
@Override public int hashCode() {
    int res = id;

    return res;
}

/** {@inheritDoc} */
@Override public String toString() {
    return "PersonKey [id=" + id +
        "]";
}
}

```

### *POJO value class*

```

/**
 * Person definition.
 *
 * Code generated by Apache Ignite Schema Import utility: 03/03/2015.
 */
public class Person implements Serializable {
    /**
     * private static final long serialVersionUID = 0L;
     */
    private int id;

    /**
     * Value for id.
     */
    private Integer orgId;
}

```

```
/** Value for name.*/
private String name;

/** 

* Gets id.

*

* @return Value for id.

*/
public int getId() {
    return id;
}

/** 

* Sets id.

*

* @param id New value for id.

*/
public void setId(int id) {
    this.id = id;
}

/** 

* Gets orgId.

*

* @return Value for orgId.

*/
public Integer getOrgId() {
    return orgId;
}

/** 

* Sets orgId.

*

```

```

* @param orgId New value for orgId.

*/
public void setOrgId(Integer orgId) {
    this.orgId = orgId;
}

/**

* Gets name.

*

* @return Value for name.

*/
public String getName() {
    return name;
}

/**

* Sets name.

*

* @param name New value for name.

*/
public void setName(String name) {
    this.name = name;
}

/** {@inheritDoc} */
@Override public boolean equals(Object o) {
    if (this == o)
        return true;

    if (!(o instanceof Person))
        return false;

    Person that = (Person)o;

    if (id != that.id)
        return false;

    if (orgId != null ? !orgId.equals(that.orgId) : that.orgId != null)
        return false;
}

```

```

        if (name != null ? !name.equals(that.name) : that.name != null)
            return false;

        return true;
    }

/** {@inheritDoc} */
@Override public int hashCode() {
    int res = id;

    res = 31 * res + (orgId != null ? orgId.hashCode() : 0);

    res = 31 * res + (name != null ? name.hashCode() : 0);

    return res;
}

/** {@inheritDoc} */
@Override public String toString() {
    return "Person [id=" + id +
        ", orgId=" + orgId +
        ", name=" + name +
        "]";
}
}

```

### *XML configuration*

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util.xsd">
    <bean class="org.apache.ignite.cache.CacheTypeMetadata">
        <property name="databaseSchema" value="PUBLIC"/>
        <property name="databaseTable" value="PERSON"/>
        <property name="keyType" value=
"org.apache.ignite.examples.datagrid.store.model.PersonKey"/>
        <property name="valueType" value=
"org.apache.ignite.examples.datagrid.store.model.Person"/>
        <property name="keyFields">
            <list>
                <bean class="org.apache.ignite.cache.CacheTypeFieldMetadata">

```

```

<property name="databaseName" value="ID"/>
<property name="databaseType">
    <util:constant static-field="java.sql.Types.INTEGER"/>
</property>
<property name="javaName" value="id"/>
<property name="javaType" value="int"/>
</bean>
</list>
</property>
<property name="valueFields">
    <list>
        <bean class="org.apache.ignite.cache.CacheTypeFieldMetadata">
            <property name="databaseName" value="ID"/>
            <property name="databaseType">
                <util:constant static-field="java.sql.Types.INTEGER"/>
            </property>
            <property name="javaName" value="id"/>
            <property name="javaType" value="int"/>
        </bean>
        <bean class="org.apache.ignite.cache.CacheTypeFieldMetadata">
            <property name="databaseName" value="ORG_ID"/>
            <property name="databaseType">
                <util:constant static-field="java.sql.Types.INTEGER"/>
            </property>
            <property name="javaName" value="orgId"/>
            <property name="javaType" value="java.lang.Integer"/>
        </bean>
        <bean class="org.apache.ignite.cache.CacheTypeFieldMetadata">
            <property name="databaseName" value="NAME"/>
            <property name="databaseType">
                <util:constant static-field="java.sql.Types.VARCHAR"/>
            </property>
            <property name="javaName" value="name"/>
            <property name="javaType" value="java.lang.String"/>
        </bean>
    </list>
</property>
</bean>
</beans>

```

## Java configuration

```

/**
 * ConfigurationSnippet definition.
 *

```

```

* Code generated by Apache Ignite Schema Import utility: 03/22/2015.

*/
public class ConfigurationSnippet {
    /** Configure cache store. */
    public static CacheStore store() {
        DataSource dataSource = null; // TODO create data source.

        CacheJdbcPojoStore store = new CacheJdbcPojoStore();
        store.setDataSource(dataSource);

        return store;
    }

    /** Configure cache types metadata. */
    public static Collection<CacheTypeMetadata> typeMetadata() {
        // Configure cache types.
        Collection<CacheTypeMetadata> meta = new ArrayList<>();

        // PERSON.
        CacheTypeMetadata type = new CacheTypeMetadata();
        type.setDatabaseSchema("PUBLIC");
        type.setDatabaseTable("PERSON");
        type.setKeyType("org.apache.ignite.PersonKey");
        type.setValueType("org.apache.ignite.Person");

        meta.add(type);

        // Key fields for PERSON.
        Collection<CacheTypeFieldMetadata> keys = new ArrayList<>();
        keys.add(new CacheTypeFieldMetadata("ID", Types.INTEGER, "id", int.class));
        type.setKeyFields(keys);

        // Value fields for PERSON.
        Collection<CacheTypeFieldMetadata> vals = new ArrayList<>();
        vals.add(new CacheTypeFieldMetadata("ID", Types.INTEGER, "id", int.class));
        vals.add(new CacheTypeFieldMetadata("FIRST_NAME", Types.VARCHAR, "firstName", String.class));
        vals.add(new CacheTypeFieldMetadata("LAST_NAME", Types.VARCHAR, "lastName", String.class));
        type.setValueFields(vals);

        return meta;
    }
}

```

## CacheJdbcPojoStore

After you generate XML and POJOs in wizard and copy all necessary stuff into your project you could use [CacheJdbcPojoStore](#) to load data from database into cache and write data from cache into database.

First you need to properly declare store in configuration:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
...
<!-- Cache configuration. -->
<property name="cacheConfiguration">
<list>
<bean class="org.apache.ignite.configuration.CacheConfiguration">
...
<!-- Cache store. -->
<property name="cacheStoreFactory">
<bean class="javax.cache.configuration.FactoryBuilder$SingletonFactory">
<constructor-arg>
<bean class="org.apache.ignite.cache.store.jdbc.CacheJdbcPojoStore">
<property name="dataSource">
<!-- Sample datasource: in-memory H2 database -->
<property name="dataSource">
<bean class="org.h2.jdbcx.JdbcConnectionPool" factory-method="create">
<constructor-arg value="jdbc:h2:tcp://localhost/mem:ExampleDb"/>
<constructor-arg value="sa"/>
<constructor-arg value="" />
</bean>
</property>
</property>
</bean>
</constructor-arg>
</bean>
</property>
<!-- Cache types metadata. -->
<property name="typeMetadata">
<list>
... Copy here types generated by wizard...
</list>
</property>

<!-- Enable store usage. -->
<!-- Sets flag indicating whether read from database is enabled. -->
<property name="readThrough" value="true"/>
```

```

<!-- Sets flag indicating whether write to database is enabled. -->
<property name="writeThrough" value="true"/>

<!-- Enable database batching. -->
<!-- Sets flag indicating whether write-behind is enabled. -->
<property name="writeBehindEnabled" value="true"/>
</List>
</property>
...
</bean>
```

```

IgniteConfiguration cfg = new IgniteConfiguration();
...
CacheConfiguration ccfg = new CacheConfiguration<>();

// Create data source for your database.
// For example: in-memory H2 database.
DataSource dataSource = org.h2.jdbcx.JdbcConnectionPool.create(
    "jdbc:h2:tcp://localhost/mem:ExampleDb", "sa", "");

// Create store.
CacheJdbcPojoStore store = new CacheJdbcPojoStore();
store.setDataSource(dataSource);

// Create store factory.
ccfg.setCacheStoreFactory(new FactoryBuilder.SingletonFactory<>(store));

// Configure cache to use store.
ccfg.setReadThrough(true);
ccfg.setWriteThrough(true);

// Enable database batching.
ccfg.setWriteBehindEnabled(true);

cfg.setCacheConfiguration(ccfg);

// Configure cache types.
Collection<CacheTypeMetadata> meta = new ArrayList<>();
... Paste here code generated by wizard ...
...

// Start Ignite node.
Ignition.start(cfg);
```

All operations defined in [\[Persistence Store\]](#) are available in [CacheJdbcPojoStore](#).

[CacheJdbcPojoStore](#) can effectively load data. To load **all** data from database for all types registered in

cache configuration - just call `IgniteCache.loadCache(null)` method. To load data with custom conditions you should pass key types and SQL queries to `IgniteCache.loadCache()` method. For example:

```
IgniteCache<Long, Person> c = node.jcache(CACHE_NAME);
c.loadCache(null, "java.lang.Integer", "select * from Person where id > 100");
```

## Example

Example `org.apache.ignite.examples.datagrid.store.auto.CacheAutoStoreExample` demonstrates usage of cache store.

In order to run example for automatic persistence you need to set `CacheNodeWithStoreStartup.STORE = AUTO` and run `CacheAutoStoreExample`.

## Demo

Lets do step-by-step demo. You can find demo sources in `examples/schema-import` folder.

We will use [H2 database](#).

- Start H2 server with following script:

*Shell*

```
$ examples/schema-import/bin/h2-server.sh
```

H2 Console will be started in your default browser.

- Connect to H2 with following settings:
- Select **Generic H2 (Server)** settings.
- Specify JDBC URL as `jdbc:h2:tcp://localhost/~/schema-import/demo`.
- Click **Connect**.

← → C 192.168.1.34:8082/login.jsp?jsessionid=f5a1d84

English ▾ Preferences Tools Help

Login

Saved Settings:

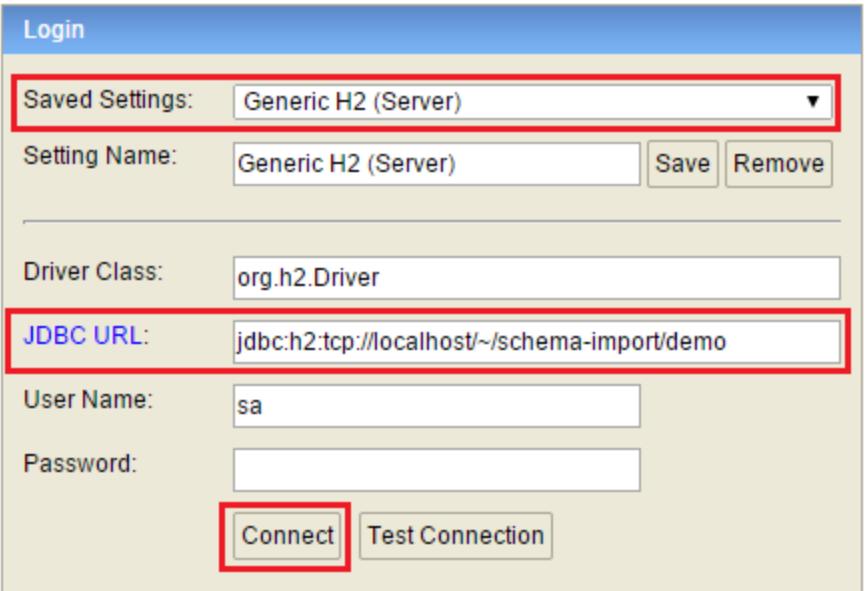
Setting Name:

Driver Class:

JDBC URL:

User Name:

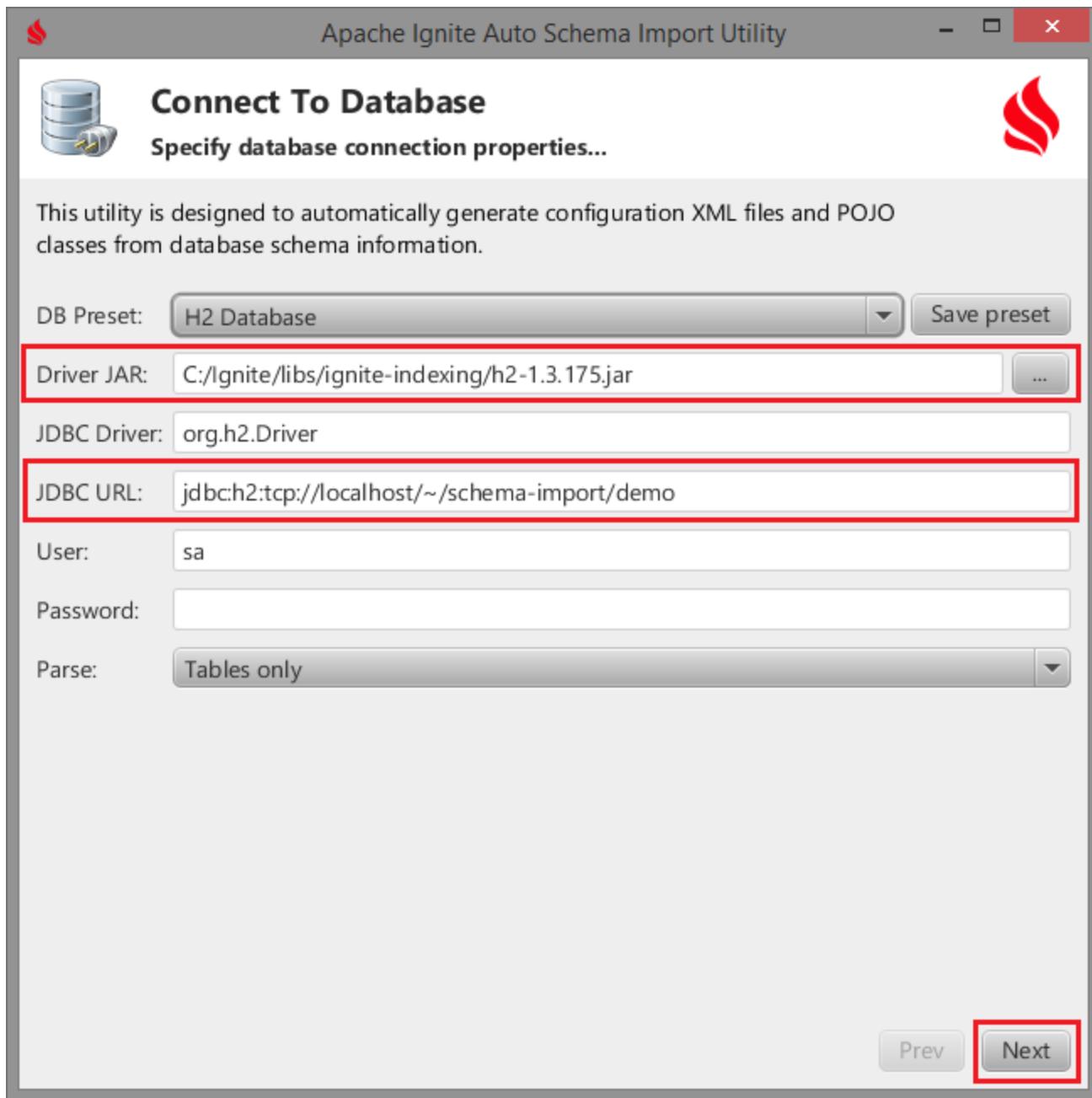
Password:



- Paste content of `examples/schema-import/bin/db-init.sql` into H2 Console and execute.
- Start Schema Import Wizard with properties configured for demo:

*Shell*

```
bin/ignite-schema-import.sh examples/schema-import/bin/schema-import.properties
```



- Click **Next**.

Apache Ignite Auto Schema Import Utility

## Generate XML And POJOs

jdbc:h2:tcp://localhost/C:\Ignite\examples\schema-import\demo

Schema / Table	Key Class Name	Value Class Name
<input checked="" type="checkbox"/> PUBLIC		
<input checked="" type="checkbox"/> PERSON	PersonKey	Person

Use	Key	AK	DB Name	DB Type	Java Name	Java Type
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	ID	INTEGER	id	int
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	FIRST_NAME	VARCHAR	firstName	java.lang.String
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	LAST_NAME	VARCHAR	lastName	java.lang.String

Package: org.apache.ignite.schema

Output Folder: C:\Ignite\examples\schema-import\src\main\java

Include key fields into value POJOs

Generate constructors for POJOs

Write all configurations to a single XML file

- Click **Generate**. Answer **Yes** for override warnings.

POJO files and java snippets will be generated in examples/schema-import/src/main/java/org/apache/ignite/schema folder.

- Import in your favorite IDE **examples/schema-import/pom.xml**.
- Run **Demo.java**.

## Data Loading

Load large amounts of data into cache.

Data loading usually has to do with initializing cache data on startup. Using standard cache `put(...)` or `putAll(...)` operations is generally inefficient for loading large amounts of data.

## IgniteDataStreamer

---

Data streamers are defined by [IgniteDataStreamer](#) API and are built to inject large amounts of continuous data into Ignite caches. Data streamers are built in a scalable and fault-tolerant fashion and achieve high performance by batching entries together before they are sent to the corresponding cluster members.



Data streamers should be used to load large amount of data into caches at any time, including pre-loading on startup. See [Data Streamers](#) documentation for more information.

## IgniteCache.loadCache()

---

Another way to load large amounts of data into cache is through `[CacheStore.loadCache()][docs/persistent-store#loadcache-]` method, which allows for cache data loading even without passing all the keys that need to be loaded.

`IgniteCache.loadCache()` method will delegate to `CacheStore.loadCache()` method on every cluster member that is running the cache. To invoke loading only on the local cluster node, use `IgniteCache.localLoadCache()` method.



In case of partitioned caches, keys that are not mapped to this node, either as primary or backups, will be automatically discarded by the cache. Here is an example of how `CacheStore.loadCache()` implementation. For a complete example of how a `CacheStore` can be implemented refer to [\[Persistence Store\]](#).

```

public class CacheJdbcPersonStore extends CacheStoreAdapter<Long, Person> {

    ...

    // This method is called whenever "IgniteCache.loadCache()" or
    // "IgniteCache.localLoadCache()" methods are called.
    @Override public void loadCache(IgniteBiInClosure<Long, Person> clo, Object... args) {
        if (args == null || args.length == 0 || args[0] == null)
            throw new CacheLoaderException("Expected entry count parameter is not provided.");

        final int entryCnt = (Integer)args[0];

        Connection conn = null;

        try (Connection conn = connection()) {
            try (PreparedStatement st = conn.prepareStatement("select * from PERSONS")) {
                try (ResultSet rs = st.executeQuery()) {
                    int cnt = 0;

                    while (cnt < entryCnt && rs.next()) {
                        Person person = new Person(rs.getLong(1), rs.getString(2), rs.getString(3));

                        clo.apply(person.getId(), person);

                        cnt++;
                    }
                }
            }
        }
        catch (SQLException e) {
            throw new CacheLoaderException("Failed to load values from cache store.", e);
        }
    }

    ...
}

```

## Partition-aware data loading

In the scenario described above the same query will be executed on all the nodes. Each node will iterate over the whole result set, skipping the keys that do not belong to the node, which is not very efficient.

The situation may be improved if partition ID is stored alongside with each record in the database. You can use `org.apache.ignite.cache.affinity.Affinity` interface to get partition ID for any key being stored into a cache.

Below is an example code snippet that determines partition ID for each `Person` object being stored into the cache.

```
IgniteCache cache = ignite.cache(cacheName);
Affinity aff = ignite.affinity(cacheName);

for (int personId = 0; personId < PERSONS_CNT; personId++) {
    // Get partition ID for the key under which person is stored in cache.
    int partId = aff.partition(personId);

    Person person = new Person(personId);
    person.setPartitionId(partId);
    // Fill other fields.

    cache.put(personId, person);
}
```

When **Person** objects become partition-ID aware, each node can query only those partitions that belong to the node. In order to do that, you can inject an instance of Ignite into your cache store and use it to determine partitions that belong to the local node.

Below is an example code snippet that demonstrates how to use **Affinity** to load only local partitions. Note that example code is single-threaded, however it can be very effectively parallelized by partition ID.

```

public class CacheJdbcPersonStore extends CacheStoreAdapter<Long, Person> {
    // Will be automatically injected.
    @IgniteInstanceResource
    private Ignite ignite;

    ...

    // This method is called whenever "IgniteCache.loadCache()" or
    // "IgniteCache.localLoadCache()" methods are called.
    @Override public void loadCache(IgniteBiInClosure<Long, Person> clo, Object... args) {
        Affinity aff = ignite.affinity(cacheName);
        ClusterNode locNode = ignite.cluster().localNode();

        try (Connection conn = connection()) {
            for (int part : aff.primaryPartitions(locNode))
                loadPartition(conn, part, clo);

            for (int part : aff.backupPartitions(locNode))
                loadPartition(conn, part, clo);
        }
    }

    private void loadPartition(Connection conn, int part, IgniteBiInClosure<Long, Person>
clo) {
        try (PreparedStatement st = conn.prepareStatement("select * from PERSONS where
partId=?")) {
            st.setInt(1, part);

            try (ResultSet rs = st.executeQuery()) {
                while (rs.next()) {
                    Person person = new Person(rs.getLong(1), rs.getString(2), rs.getString(3));

                    clo.apply(person.getId(), person);
                }
            }
        }
        catch (SQLException e) {
            throw new CacheLoaderException("Failed to load values from cache store.", e);
        }
    }

    ...
}

```



Note that key-to-partition mapping depends on the number of partitions configured in the affinity function (see [org.apache.ignite.cache.affinity.AffinityFunction](#)). If affinity function configuration changes, partition ID records in the database must be updated accordingly.

## Eviction Policies

---

Eviction policies control the maximum number of elements that can be stored in a cache on-heap memory. Whenever maximum on-heap cache size is reached, entries are evicted into [Off-Heap Memory](#), if one is enabled.

Some eviction policies support batch eviction and eviction by memory size limit.

If batch eviction is enabled than eviction starts when cache size becomes `batchSize` elements greater than the maximum cache size. In this cases `batchSize` entries will be evicted.

If eviction by memory size limit is enabled then eviction starts when size of cache entries in bytes becomes greater than the maximum memory size.



Batch eviction is supported only if maximum memory limit isn't set.

In Ignite eviction policies are pluggable and are controlled via [EvictionPolicy](#) interface. An implementation of eviction policy is notified of every cache change and defines the algorithm of choosing the entries to evict from cache.



If your data set can fit in memory, then eviction policy will not provide any benefit and should be disabled, which is the default behavior.

### Least Recently Used (LRU)

---

LRU eviction policy is based on [Least Recently Used \(LRU\)](#) gets evicted first.

Supports batch eviction and eviction by memory size limit.



LRU eviction policy nicely fits most of the use cases for caching. Use it whenever in doubt. This eviction policy is implemented by [LruEvictionPolicy](#) and can be configured via [CacheConfiguration](#).

```

<bean class="org.apache.ignite.cache.CacheConfiguration">
    <property name="name" value="myCache"/>
    ...
    <property name="evictionPolicy">
        <!-- LRU eviction policy. -->
        <bean class="org.apache.ignite.cache.eviction.lru.LruEvictionPolicy">
            <!-- Set the maximum cache size to 1 million (default is 100,000). -->
            <property name="maxSize" value="1000000"/>
        </bean>
    </property>
    ...
</bean>

```

```

CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

// Set the maximum cache size to 1 million (default is 100,000).
cacheCfg.setEvictionPolicy(new LruEvictionPolicy(1000000));

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);

```

## First In First Out (FIFO)

FIFO eviction policy is based on [First-In-First-Out \(FIFO\)](#) algorithm which ensures that entry that has been in cache the longest will be evicted first. It is different from [LruEvictionPolicy](#) because it ignores the access order of entries.

Supports batch eviction and eviction by memory size limit.

This eviction policy is implemented by [FifoEvictionPolicy](#) and can be configured via [CacheConfiguration](#).

```

<bean class="org.apache.ignite.cache.CacheConfiguration">
    <property name="name" value="myCache"/>
    ...
    <property name="evictionPolicy">
        <!-- FIFO eviction policy. -->
        <bean class="org.apache.ignite.cache.eviction.fifo.FifoEvictionPolicy">
            <!-- Set the maximum cache size to 1 million (default is 100,000). -->
            <property name="maxSize" value="1000000"/>
        </bean>
    </property>
    ...
</bean>

```

```

CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

// Set the maximum cache size to 1 million (default is 100,000).
cacheCfg.setEvictionPolicy(new FifoEvictionPolicy(1000000));

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);

```

## Sorted

Sorted eviction policy is similar to FIFO eviciton policy with the difference that entries order is defined by default or user defined comparator and ensures that the minimal entry (i.e. the entry that has integer key with smallest value) gets evicted first.

Default comparator uses cache entries keys for comparison that imposes a requirement for keys to implement [Comparable](#) interface. User can provide own comparator implementation which can use keys, values or both for entries comparison.

Supports batch eviction and eviction by memory size limit.

This eviction policy is implemented by [SortedEvictionPolicy](#) and can be configured via [CacheConfiguration](#).

```

<bean class="org.apache.ignite.cache.CacheConfiguration">
    <property name="name" value="myCache"/>
    ...
    <property name="evictionPolicy">
        <!-- Sorted eviction policy. -->
        <bean class="org.apache.ignite.cache.eviction.sorted.SortedEvictionPolicy">
            <!-- Set the maximum cache size to 1 million (default is 100,000) and use
            default comparator. -->
            <property name="maxSize" value="1000000"/>
        </bean>
    </property>
    ...
</bean>

```

```

CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

// Set the maximum cache size to 1 million (default is 100,000).
cacheCfg.setEvictionPolicy(new SortedEvictionPolicy(1000000));

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);

```

## Random

Random eviction policy which randomly chooses entries to evict. This eviction policy is mainly used for debugging and benchmarking purposes.

This eviction policy is implemented by `RandomEvictionPolicy` and can be configured via `CacheConfiguration`.

```

<bean class="org.apache.ignite.cache.CacheConfiguration">
    <property name="name" value="myCache"/>
    ...
    <property name="evictionPolicy">
        <!-- Random eviction policy. -->
        <bean class="org.apache.ignite.cache.eviction.random.RandomEvictionPolicy">
    <!-- Set the maximum cache size to 1 million (default is 100,000). -->
        <property name="maxSize" value="1000000"/>
    </bean>
</property>
...
</bean>

```

```

CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

// Set the maximum cache size to 1 million (default is 100,000).
cacheCfg.setEvictionPolicy(new RandomEvictionPolicy(1000000));

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);

```

## Expiry Policies

Expiry Policy specifies the amount of time that must pass before a cache entry is considered expired. Time can be counted from creation, last access or modification time.

Expiry Policy can be setup using any of the predefined implementations of `ExpiryPolicy`:

<code>CreatedExpiryPolicy</code>	<code>Used</code>	<code>AccessedExpiryPolicy</code>	<code>Used</code>
<code>ModifiedExpiryPolicy</code>	Used	<code>TouchedExpiryPolicy</code>	Used
Class name	creation time	<code>EternalExpiryPolicy</code>	
last access time	last update time		
Used		Used	

<a href="#">CreatedExpiryPolicy</a>	<b>Used</b>	<a href="#">AccessedExpiryPolicy</a>	<b>Used</b>
Used	Used		

Custom [ExpiryPolicy](#) implementations are also allowed.

Expiry Policy can be setup in [CacheConfiguration](#). This policy will be used for all entries inside cache.

```
cfg.setExpiryPolicyFactory(CreatedExpiryPolicy.factoryOf(Duration.ZERO));
```

Also, it is possible to change or set Expiry Policy for individual operations on cache.

```
IgniteCache<Object, Object> cache = cache.withExpiryPolicy(
    new CreatedExpiryPolicy(new Duration(TimeUnit.SECONDS, 5)));
```

This policy will be used for each operation invoked on the returned cache instance.

## Data Rebalancing

Preload data from other grid nodes to maintain data consistency.

---

When a new node joins topology, existing nodes relinquish primary or back up ownership of some keys to the new node so that keys remain equally balanced across the grid at all times.

If the new node becomes a primary or backup for some partition, it will fetch data from previous primary node for that partition or from one of the backup nodes for that partition. Once a partition is fully loaded to the new node, it will be marked obsolete on the old node and will be eventually evicted after all current transactions on that node are finished. Hence, for some short period of time, after topology changes, there can be a case when a cache will have more backup copies for a key than configured. However once rebalancing completes, extra backup copies will be removed from node caches.

### Rebalance Modes

---

Following rebalance modes are defined in [CacheRebalanceMode](#) enum.

SYNC	CacheRebalanceMode
Description	Synchronous rebalancing mode. Distributed caches will not start until all necessary data is loaded from other available grid nodes. This means that any call to cache public API will be blocked until rebalancing is finished.
Asynchronous rebalancing mode. Distributed caches will start immediately and will load all necessary data from other available grid nodes in the background.	ASYNC
In this mode no rebalancing will take place which means that caches will be either loaded on demand from persistent store whenever data is accessed, or will be populated explicitly.	NONE

By default, `ASYNC` rebalance mode is enabled. To use another mode, you can set the `rebalanceMode` property of `CacheConfiguration`, like so:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set synchronous rebalancing. -->
            <property name="rebalanceMode" value="SYNC"/>
            ...
        </bean>
    </property>
</bean>
```

```
CacheConfiguration cacheCfg = new CacheConfiguration();
cacheCfg.setRebalanceMode(CacheRebalanceMode.SYNC);

IgniteConfiguration cfg = new IgniteConfiguration();
cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

## Rebalance Message Throttling

When re-balancer transfers data from one node to another, it splits the whole data set into batches and sends each batch in a separate message. If your data sets are large and there are a lot of messages to send, the CPU or network can get over-consumed. In this case it can be reasonable to wait between rebalance messages so that negative performance impact caused by rebalancing process is minimized. This time interval is controlled by `rebalanceThrottle` configuration property of `CacheConfiguration`. Its default value is 0, which means that there will be no pauses between messages. Note that size of a single message can be also customized by `rebalanceBatchSize` configuration property (default size is 512K).

For example, if you want rebalancer to send 2MB of data per message with 100 ms throttle interval, you should provide the following configuration:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set batch size. -->
            <property name="rebalanceBatchSize" value="#{2 * 1024 * 1024}" />

            <!-- Set throttle interval. -->
            <property name="rebalanceThrottle" value="100" />
        ...
    </bean>
</property>
</bean>
```

```
CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setRebalanceBatchSize(2 * 1024 * 1024);

cacheCfg.setRebalanceThrottle(100);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

## Configuration

Cache rebalancing behavior can be customized by optionally setting the following configuration properties:

Setter Method	Description	Default
<code>setRebalanceMode</code>	Rebalance mode for distributed cache. See Rebalance Modes section for details.	<code>setRebalancePartitionedDelay</code>
Rebalancing delay in milliseconds. See Delayed And Manual Rebalancing section for details.	<code>setRebalanceBatchSize</code>	Size (in bytes) to be loaded within a single rebalance message. Rebalancing algorithm will split total data set on every node into multiple batches prior to sending data.
<code>setRebalanceThreadPoolSize</code>	Size of rebalancing thread pool. Note that size serves as a hint and implementation may create more threads for the rebalancing than specified here (but never less threads).	<code>setRebalanceThrottle</code>
Time in milliseconds to wait between rebalancing messages to avoid overloading of CPU or network. When rebalancing large data sets, the CPU or network can get over-consumed with rebalance messages, which consecutively may slow down the application performance. This parameter helps tune the amount of time to wait between rebalance messages to make sure that rebalancing process does not have any negative performance impact. Note that application will continue to work properly while rebalancing is still in progress.	<code>setRebalanceOrder</code>	<code>setRebalanceTimeout</code>
Order in which rebalancing should be done. Rebalance order can be set to non-zero value for caches with SYNC or ASYNC rebalance modes only. Rebalancing for caches with smaller rebalance order will be completed first. By default, rebalancing is not ordered.	Rebalance timeout (ms).	<code>ASYNC</code>
0 (no delay)	512K	2

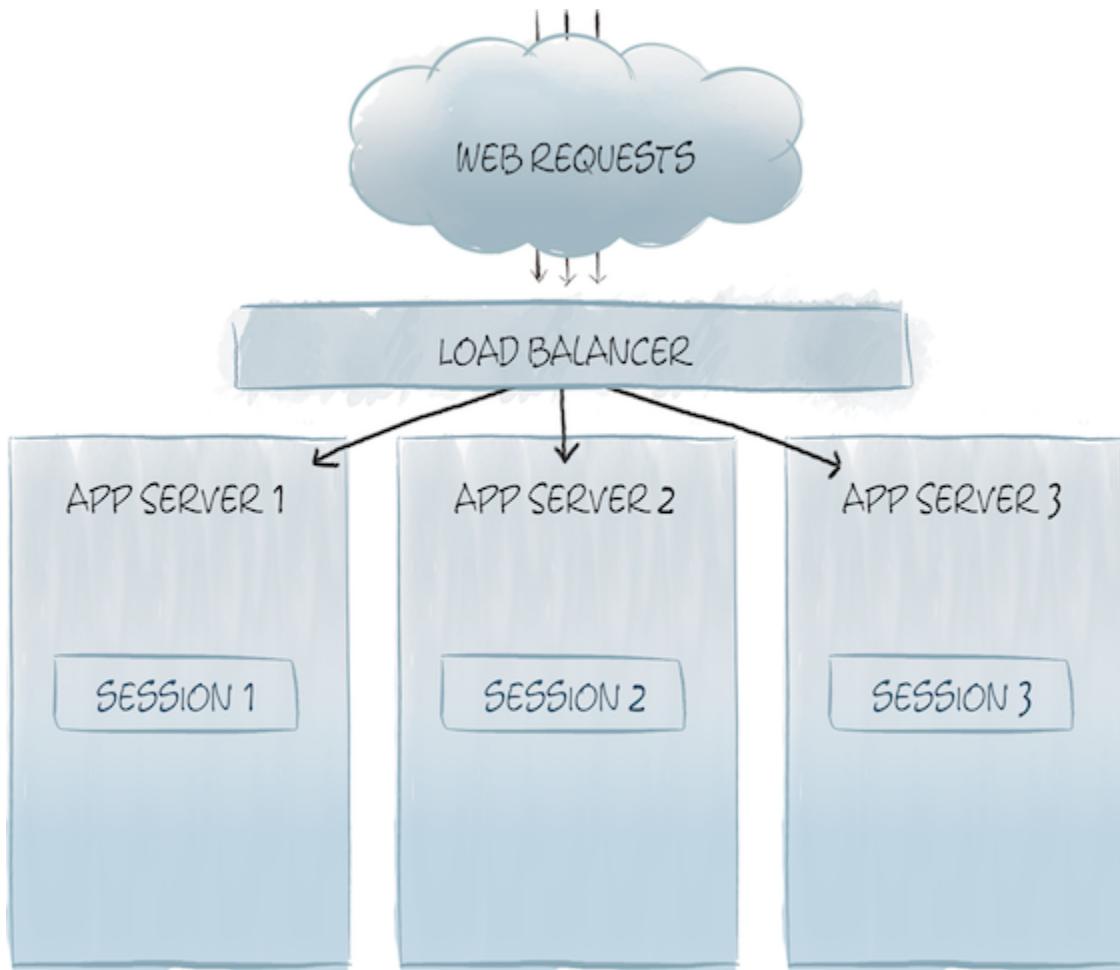
Setter Method	Description	Default
0 (throttling disabled)	0	10000

## Web Session Clustering

Cache all your web sessions in a fault-tolerant distributed cache.

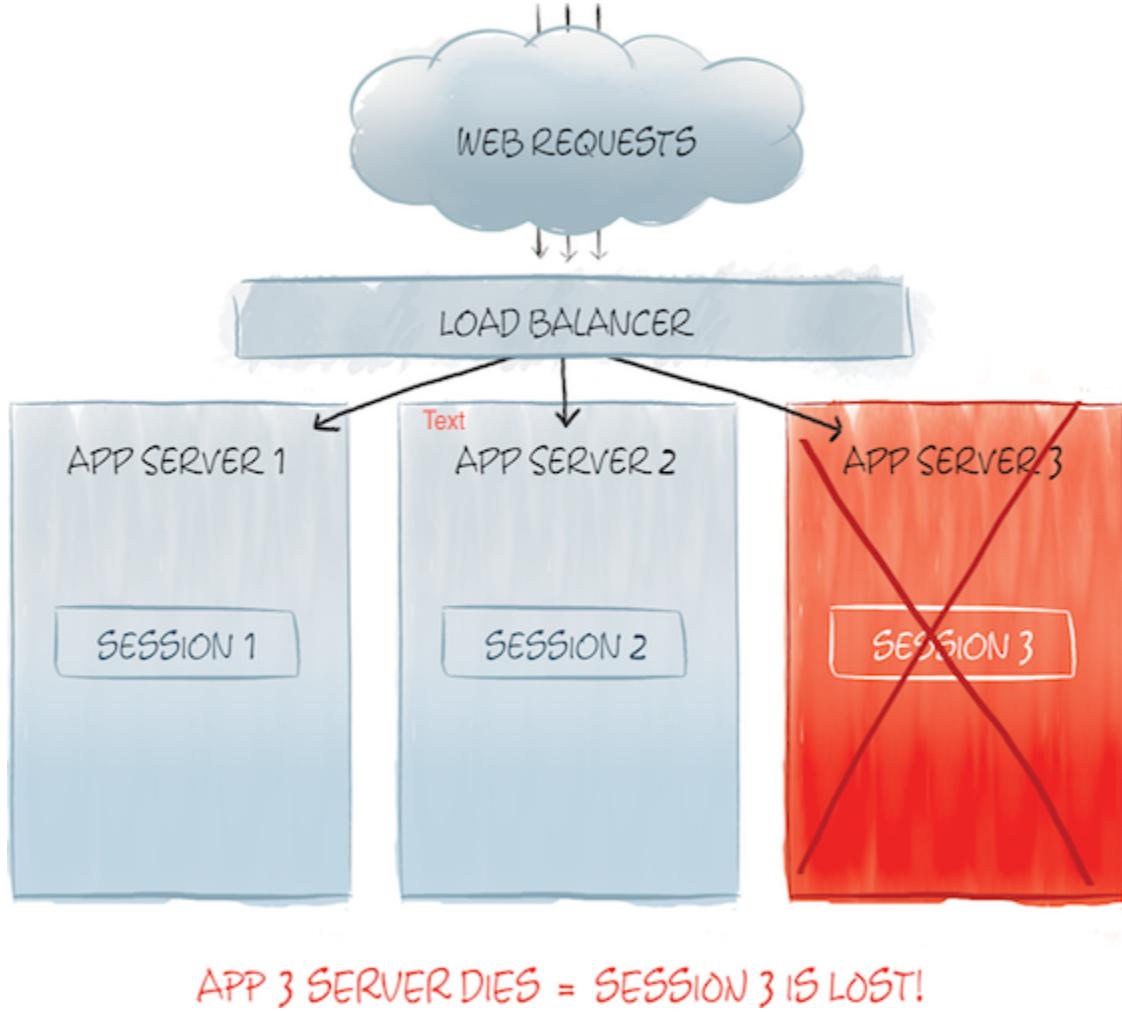
Ignite In-Memory Data Fabric is capable of caching web sessions of all Java Servlet containers that follow Java Servlet 3.0 Specification, including Apache Tomcat, Eclipse Jetty, Oracle WebLogic, and others.

Web sessions caching becomes useful when running a cluster of app servers. When running a web application in a servlet container, you may face performance and scalability problems. A single app server is usually not able to handle large volumes of traffic by itself. A common solution is to scale your web application across multiple clustered instances:

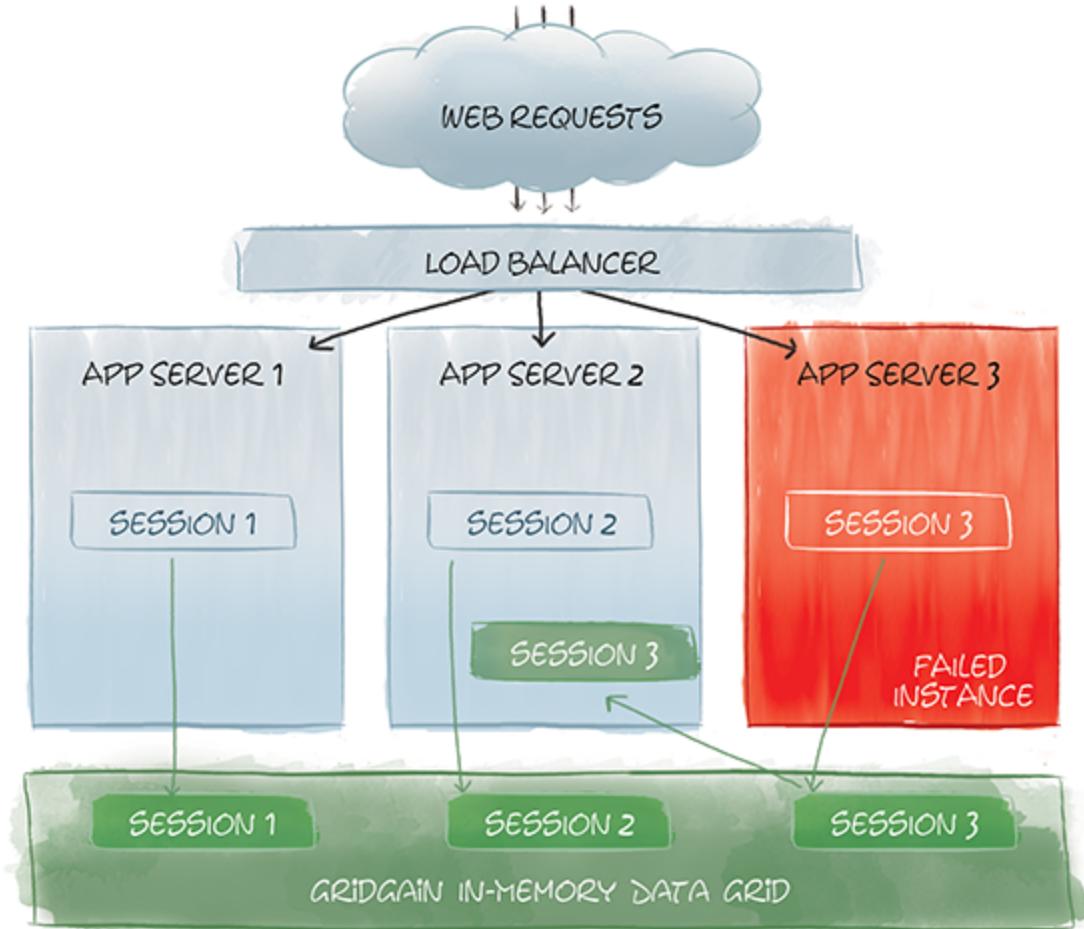


In the architecture shown above, High Availability Proxy (Load Balancer) distributes requests between multiple Application Server instances (App Server 1, App Server 2, ...), reducing the load on each

instance and providing service availability if any of the instances fails. The problem here is web session availability. A web session keeps an intermediate logical state between requests by using cookies, and is normally bound to a particular application instance. Generally this is handled using sticky connections, ensuring that requests from the same user are handled by the same app server instance. However, if that instance fails, the session is lost, and the user will have to create it anew, loosing all the current unsaved state:



A solution here is to use Ignite In-Memory Data Fabric web sessions cache - a distributed cache that maintains a copy of each created session, sharing them between all instances. If any of your application instances fails, Ignite will automatically restore the sessions, owned by the failed instance, from the distributed cache regardless of which app server the next request will be forwarded to. Moreover, with web session caching sticky connections become less important as the session is available on any app server the web request may be routed to.



In this chapter we give a brief architecture overview of Ignite's web session caching functionality and instructions on how to configure your web application to enable web sessions caching.

## Architecture

---

To set up a distributed web sessions cache with Ignite, you normally configure your web application to start a Ignite node (embedded mode). When multiple application server instances are started, all Ignite nodes connect with each-other forming a distributed cache.



Note that not every Ignite caching node has to be running inside of application server. You can also start additional, standalone Ignite nodes and add them to the topology as well.

## Replication Strategies

---

There are several replication strategies you can use when storing sessions in Ignite In-Memory Data Fabric. The replication strategy is defined by the backing cache settings. In this section we briefly cover

most common configurations.

## Fully Replicated Cache

This strategy stores copies of all sessions on each Ignite node, providing maximum availability. However with this approach you can only cache as many web sessions as can fit in memory on a single server. Additionally, the performance may suffer as every change of web session state now must be replicated to all other cluster nodes.

To enable fully replicated strategy, set cacheMode of your backing cache to **REPLICATED**:

```
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    <!-- Cache mode. -->
    <property name="cacheMode" value="REPLICATED"/>
    ...
</bean>
```

## Partitioned Cache with Backups

In partitioned mode, web sessions are split into partitions and every node is responsible for caching only partitions assigned to that node. With this approach, the more nodes you have, the more data can be cached. New nodes can always be added on the fly to add more memory.



With **Partitioned** mode, redundancy is addressed by configuring number of backups for every web session being cached. To enable partitioned strategy, set cacheMode of your backing cache to **PARTITIONED**, and set the number of backups with **backups** property of **CacheConfiguration**:

```
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    <!-- Cache mode. -->
    <property name="cacheMode" value="PARTITIONED"/>
    <property name="backups" value="1"/>
</bean>
```



See [Cache Modes](#) for more information on different replication strategies available in Ignite.

## Expiration and Eviction

Stale sessions are cleaned up from cache automatically when they expire. However, if there are a lot of long-living sessions created, you may want to save memory by evicting dispensable sessions from cache when cache reaches a certain limit. This can be done by setting up cache eviction policy and

specifying the maximum number of sessions to be stored in cache. For example, to enable automatic eviction with LRU algorithm and a limit of 10000 sessions, you will need to use the following cache configuration:

```
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    <!-- Cache name. -->
    <property name="name" value="session-cache"/>

    <!-- Set up LRU eviction policy with 10000 sessions limit. -->
    <property name="evictionPolicy">
        <bean class="org.apache.ignite.cache.eviction.lru.LruEvictionPolicy">
            <property name="maxSize" value="10000"/>
        </bean>
    </property>
    ...
</bean>
```



For more information about various eviction policies, see [Eviction Policies](#) section.

## Configuration

To enable web session caching in your application with Ignite, you need to:

1\. **Add Ignite JARs** - Download Ignite and add the following jars to your application's classpath ([WEB-INF/libs](#) folder):

- [ignite-core.jar](#)
- [cache-api-1.0.0.jar](#)
- [ignite-web.jar](#)
- [ignite-log4j.jar](#)
- [ignite-spring.jar](#)

Or, if you have a Maven based project, add the following to your application's pom.xml.

```

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

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

<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-log4j</artifactId>
    <version>${ignite.version}</version>
</dependency>

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

```

Make sure to replace \${ignite.version} with actual Ignite version.

2\. **Configure Cache Mode** - Configure Ignite cache in either **PARTITIONED** or **REPLICATED** mode (See [Replication Strategies](#)).

3\. **Update `web.xml`** - Declare a context listener and web session filter in `web.xml`:

```

...
<listener>
    <listener-class>
org.apache.ignite.startup.servlet.ServletContextListenerStartup</listener-class>
</listener>

<filter>
    <filter-name>IgniteWebSessionsFilter</filter-name>
    <filter-class>org.apache.ignite.cache.websession.WebSessionFilter</filter-class>
</filter>

<!-- You can also specify a custom URL pattern. -->
<filter-mapping>
    <filter-name>IgniteWebSessionsFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- Specify Ignite configuration (relative to META-INF folder or Ignite_HOME). -->
<context-param>
    <param-name>IgniteConfigurationFilePath</param-name>
    <param-value>config/default-config.xml </param-value>
</context-param>

<!-- Specify the name of Ignite cache for web sessions. -->
<context-param>
    <param-name>IgniteWebSessionsCacheName</param-name>
    <param-value>partitioned</param-value>
</context-param>

...

```

On application start, the listener will start a Ignite node within your application, which will connect to other nodes in the network, forming a distributed cache.

**4\. Set Eviction Policy (Optional)** - Set eviction policy for stale web sessions data lying in cache (See [Expiration and Eviction](#)).

## Configuration Parameters

`ServletContextListenerStartup` has the following configuration parameters:

<code>IgniteConfigurationFilePath</code>	<b>Path to Ignite configuration file (relative to <code>META-INF</code> folder or <code>IGNITE_HOME</code>).</b>	<code>/config/default-config.xml</code>
Default	Description	Parameter Name

`WebSessionFilter` has the following configuration parameters:

Parameter Name	Description	Default
<code>IgniteWebSessionsGridName</code>	Grid name for a started Ignite node. Should refer to grid in configuration file (if a grid name is specified in configuration).	null
<code>IgniteWebSessionsCacheName</code>	<code>IgniteWebSessionsMaximumRetriesOnFail</code>	Name of Ignite cache to use for web sessions caching.
null	Valid only for <code>ATOMIC</code> caches. Specifies number of retries in case of primary node failures.	3

## Supported Containers

---

Ignite has been officially tested with following servlet containers:

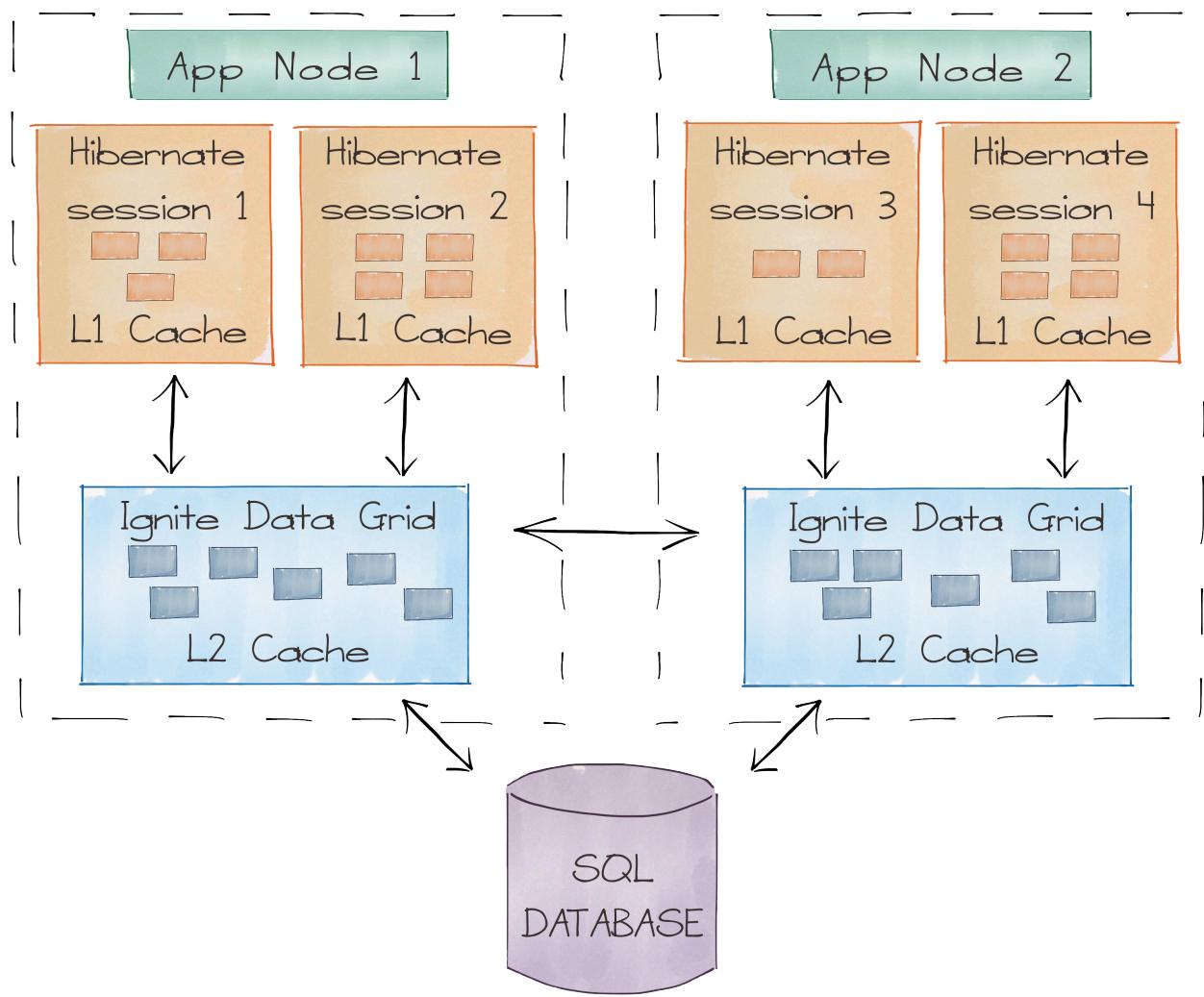
- Apache Tomcat 7
- Eclipse Jetty 9
- Apache Tomcat 6
- Oracle WebLogic >= 10.3.4

## Hibernate L2 Cache

---

Ignite In-Memory Data Fabric can be used as [Hibernate](#), which can significantly speed-up the persistence layer of your application.

[Hibernate](#). While interacting closely with an SQL database, it performs caching of retrieved data to minimize expensive database requests.



All work with Hibernate database-mapped objects is done within a session, usually bound to a worker thread or a Web session. By default, Hibernate only uses per-session (L1) cache, so, objects, cached in one session, are not seen in another. However, a global second-level (L2) cache may be used, in which the cached objects are seen for all sessions that use the same L2 cache configuration. This usually gives a significantly greater performance gain, because each newly-created session can take full advantage of the data already present in L2 cache memory (which outlives any session-local L1 cache).

While L1 cache is always enabled and fully implemented by Hibernate internally, L2 cache is optional and can have multiple pluggable implementations. Ignite can be easily plugged-in as an L2 cache implementation, and can be used in all access modes ([READ\\_ONLY](#), [READ\\_WRITE](#), [NONSTRICT\\_READ\\_WRITE](#), and [TRANSACTIONAL](#)), supporting a wide range of related features:

- caching to memory and disk, as well as off-heap memory.
- cache transactions, that make `TRANSACTIONAL` mode possible.
- clustering, with 2 different replication modes: [REPLICATED](#) and [PARTITIONED](#)

To start using GridGain as a Hibernate L2 cache, you need to perform 3 simple steps:

- Add Ignite libraries to your application's classpath.
- Enable L2 cache and specify Ignite implementation class in L2 cache configuration.
- Configure Ignite caches for L2 cache regions and start the embedded Ignite node (and, optionally, external Ignite nodes).

In the section below we cover these steps in more detail.

## L2 Cache Configuration

To configure Ignite In-Memory Data Fabric as a Hibernate L2 cache, without any changes required to the existing Hibernate code, you need to:

- Add dependency on **hibernate-ignite** module.
- Configure Hibernate itself to use Ignite as L2 cache.
- Configure Ignite cache appropriately.

## Maven Configuration



**Maven Dependency.** In order to enable Ignite hibernate integration, you need to add **ignite-hibernate** dependency to your project, or when starting from command line, copy **ignite-hibernate** module from **libs/optional** to **libs** folder. To add Ignite hibernate integration to your project, add the following dependency to your POM file:

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-hibernate</artifactId>
    <version>RELEASE</version>
</dependency>
```

## Hibernate Configuration Example

A typical Hibernate configuration for L2 cache with Ignite would look like the one below:

```

<hibernate-configuration>
    <session-factory>
        ...
        <!-- Enable L2 cache. -->
        <property name="cache.use_second_level_cache">true</property>

        <!-- Generate L2 cache statistics. -->
        <property name="generate_statistics">true</property>

        <!-- Specify GridGain as L2 cache provider. -->
        <property name="cache.region.factory_class"
>org.apache.ignite.cache.hibernate.HibernateRegionFactory</property>

        <!-- Specify the name of the grid, that will be used for second level caching.
-->
        <property name="org.apache.ignite.hibernate.grid_name">hibernate-grid</property>

        <!-- Set default L2 cache access type. -->
        <property name="org.apache.ignite.hibernate.default_access_type">
READ_ONLY</property>

        <!-- Specify the entity classes for mapping. -->
        <mapping class="com.mycompany.MyEntity1"/>
        <mapping class="com.mycompany.MyEntity2"/>

        <!-- Per-class L2 cache settings. -->
        <class-cache class="com.mycompany.MyEntity1" usage="read-only"/>
        <class-cache class="com.mycompany.MyEntity2" usage="read-only"/>
        <collection-cache collection="com.mycompany.MyEntity1.children" usage="read-only
"/>
        ...
    </session-factory>
</hibernate-configuration>

```

Here, we do the following:

- Enable L2 cache (and, optionally, the L2 cache statistics generation).
- Specify Ignite as L2 cache implementation.
- Specify the name of the caching grid (should correspond to the one in Ignite configuration).
- Specify the entity classes and configure caching for each class (a corresponding cache region should be configured in Ignite).

## Ignite Configuration Example

A typical Ignite configuration for Hibernate L2 caching looks like this:

```

<!-- Basic configuration for atomic cache. -->
<bean id="atomic-cache" class="org.apache.ignite.configuration.CacheConfiguration"
abstract="true">
    <property name="cacheMode" value="PARTITIONED"/>
    <property name="atomicityMode" value="ATOMIC"/>
    <property name="writeSynchronizationMode" value="FULL_SYNC"/>
</bean>

<!-- Basic configuration for transactional cache. -->
<bean id="transactional-cache" class="org.apache.ignite.configuration.CacheConfiguration"
abstract="true">
    <property name="cacheMode" value="PARTITIONED"/>
    <property name="atomicityMode" value="TRANSACTIONAL"/>
    <property name="writeSynchronizationMode" value="FULL_SYNC"/>
</bean>

<bean id="ignite.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
    <!--
        Specify the name of the caching grid (should correspond to the
        one in Hibernate configuration).
    -->
    <property name="gridName" value="hibernate-grid"/>
    ...
    <!--
        Specify cache configuration for each L2 cache region (which corresponds
        to a full class name or a full association name).
    -->
    <property name="cacheConfiguration">
        <list>
            <!--
                Configurations for entity caches.
            -->
            <bean parent="transactional-cache">
                <property name="name" value="com.mycompany.MyEntity1"/>
            </bean>
            <bean parent="transactional-cache">
                <property name="name" value="com.mycompany.MyEntity2"/>
            </bean>
            <bean parent="transactional-cache">
                <property name="name" value="com.mycompany.MyEntity1.children"/>
            </bean>

            <!-- Configuration for update timestamps cache. -->
            <bean parent="atomic-cache">
                <property name="name" value=
"org.hibernate.cache.spi.UpdateTimestampsCache"/>
            </bean>

```

```

<!-- Configuration for query result cache. -->
<bean parent="atomic-cache">
    <property name="name" value=
"org.hibernate.cache.internal.StandardQueryCache"/>
    </bean>
</list>
</property>
...
</bean>
```

Here, we specify the cache configuration for each L2 cache region:

- We use **PARTITIONED** cache to split the data between caching nodes. Another possible strategy is to enable **REPLICATED** mode, thus replicating a full dataset between all caching nodes. See Cache Distribution Models for more information.
- We specify the cache name that corresponds an L2 cache region name (either a full class name or a full association name).
- We use **TRANSACTIONAL** atomicity mode to take advantage of cache transactions.
- We enable **FULL\_SYNC** to be always fully synchronized with backup nodes.

Additionally, we specify a cache for update timestamps, which may be **ATOMIC**, for better performance.

Having configured Ignite caching node, we can start it from within our code the following way:

```
Ignition.start("my-config-folder/my-ignite-configuration.xml");
```

After the above line is executed, the internal Ignite node is started and is ready to cache the data. We can also start additional standalone nodes by running the following command from console:

```
$IGNITE_HOME/bin/ignite.sh my-config-folder/my-ignite-configuration.xml
```

For Windows, use the **.bat** script in the same folder.



The nodes may be started on other hosts as well, forming a distributed caching cluster. Be sure to specify the right network settings in GridGain configuration file for that.

## Query Cache

In addition to L2 cache, Hibernate offers a query cache. This cache stores the results of queries (either

HQL or Criteria) with a given set of parameters, so, when you repeat the query with the same parameter set, it hits the cache without going to the database.

Query cache may be useful if you have a number of queries, which may repeat with the same parameter values. Like in case of L2 cache, Hibernate relies on a 3-rd party cache implementation, and Ignite In-Memory Data Fabric can be used as such.



Consider using support for [SQL-based In-Memory Queries](/docs/cache-queries) in Ignite which should perform faster than going through Hibernate.

## Query Cache Configuration

The [L2 Cache Configuration](#) information above totally applies to query cache, but some additional configuration and code change is required.

### Hibernate Configuration

To enable query cache in Hibernate, you only need one additional line in configuration file:

```
<!-- Enable query cache. -->
<property name="cache.use_query_cache">true</property>
```

Yet, a code modification is required: for each query that you want to cache, you should enable `cacheable` flag by calling `setCacheable(true)`:

```
Session ses = ...;

// Create Criteria query.
Criteria criteria = ses.createCriteria(cls);

// Enable cacheable flag.
criteria.setCacheable(true);

...
```

After this is done, your query results will be cached.

### Ignite Configuration

To enable Hibernate query caching in Ignite, you need to specify an additional cache configuration:

```

<property name="cacheConfiguration">
    <list>
        ...
        <!-- Query cache (refers to atomic cache defined in above example). -->
        <bean parent="atomic-cache">
            <property name="name" value="org.hibernate.cache.internal.StandardQueryCache"/>
        </bean>
    </list>
</property>

```

Notice that the cache is made **ATOMIC** for better performance.

## JDBC Driver

Connect to Ignite using standard JDBC driver.

Ignite is shipped with JDBC driver that allows you to retrieve distributed data from cache using standard SQL queries and JDBC API.

### JDBC Connection

In Ignite JDBC connection URL has the following pattern:

```
jdbc:ignite:cfg://[<params>]@<config_url>
```

- **<config\_url>** is required and represents any valid URL which points to Ignite configuration file. See [Clients and Servers](#) section for details.
- **<params>** is optional part and have the following format:

```
param1=value1:param2=value2:...:paramN=valueN
```

The following parameters are supported:

Properties	Description	Default
cache	nodeId	local

Properties	Description	Default
<code>collocated</code>	Cache name. If it is not defined than default cache will be used. Note that the cache name is case sensitive.	ID of node where query will be executed. It can be useful for querying through local caches.
Query will be executed only on local node. Use this parameter with <code>nodeId</code> parameter in order to limit data set by specified node.	false	Flag that used for optimization purposes. Whenever Ignite executes a distributed query, it sends sub-queries to individual cluster members. If you know in advance that the elements of your query selection are collocated together on the same node, Ignite can make significant performance and network optimizations.



**Cross-Cache Queries.** Cache that the driver is connected to is treated as the default schema. To query across multiple caches, [Cross-Cache Query](/docs/cache-queries#cross-cache-queries) functionality can be used.



**Joins and Collocation.** Just like with [Cache Queries](#) for more details.



**Replicated vs Partitioned Caches.** Queries on `REPLICATED` caches will run directly only on one node, while queries on `PARTITIONED` caches are distributed across all cache nodes.

## Example

Ignite JDBC driver automatically gets only those fields that you actually need from objects stored in cache. For example you have `Person` class declared like this:

```
public class Person {
    @QuerySqlField
    private String name;

    @QuerySqlField
    private int age;

    // Getters and setters.
    ...
}
```

If you have instances of this class in cache, you can query individual fields (name, age or both) via standard JDBC API, like so:

### Java

```
// Register JDBC driver.  
Class.forName("org.apache.ignite.IgniteJdbcDriver");  
  
// Open JDBC connection (cache name is not specified, which means that we use default  
// cache).  
Connection conn = DriverManager.getConnection(  
    "jdbc:ignite:cfg://file:///etc/config/ignite-jdbc.xml");  
  
// Query names of all people.  
ResultSet rs = conn.createStatement().executeQuery("select name from Person");  
  
while (rs.next()) {  
    String name = rs.getString(1);  
    ...  
}  
  
// Query people with specific age using prepared statement.  
PreparedStatement stmt = conn.prepareStatement("select name, age from Person where age = ?");  
  
stmt.setInt(1, 30);  
  
ResultSet rs = stmt.executeQuery();  
  
while (rs.next()) {  
    String name = rs.getString("name");  
    int age = rs.getInt("age");  
    ...  
}
```

## Backward compatibility

For previous versions of Ignite (prior 1.4) JDBC connection URL has the following pattern:

```
jdbc:ignite://<hostname>:<port>/<cache_name>
```

See the corresponding [documentation](#) for details.

# Spring Caching

Use Spring Cache Abstraction to interact with Ignite cache

---

Ignite is shipped with the [SpringCacheManager](#) - an implementation of [Spring Cache Abstraction](#). It provides annotation-based way to enable caching for Java methods so that the result of a method execution is stored in the Ignite cache. If later the same method is called with the same set of parameters, the result will be retrieved from the cache instead of actually executing the method.



**Spring Cache Abstraction documentation.** For more information on how to use the Spring Cache Abstraction, including available annotations, refer to this documentation page: <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html>

## How to enable caching

---

Only two simple steps are required to plug in Ignite's cache into your Spring-based application:

- Start an Ignite node with proper configuration in embedded mode (i.e., in the same JVM where the application is running). It can already have predefined caches, but it's not required - caches will be created automatically on first access if needed.
- Configure [SpringCacheManager](#) as the cache manager in the Spring application context.

The embedded node can be started by [SpringCacheManager](#) itself. In this case you will need to provide a path to Ignite configuration XML file or [IgniteConfiguration](#) bean via [configurationPath](#) or [configuration](#) properties respectively (see examples below). Note that setting both is illegal and results in [IllegalArgumentException](#).

## Configuration path

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache
        http://www.springframework.org/schema/cache/spring-cache.xsd">
    <!-- Provide configuration file path. -->
    <bean id="cacheManager" class="org.apache.ignite.cache.spring.SpringCacheManager">
        <property name="configurationPath" value="examples/config/spring-cache.xml"/>
    </bean>

    <!-- Enable annotation-driven caching. -->
    <cache:annotation-driven/>
</beans>
```

## Configuration bean

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache
        http://www.springframework.org/schema/cache/spring-cache.xsd">
    <!-- Provide configuration bean. -->
    <bean id="cacheManager" class="org.apache.ignite.cache.spring.SpringCacheManager">
        <property name="configuration">
            <bean class="org.apache.ignite.configuration.IgniteConfiguration">
                ...
                </bean>
            </property>
        </bean>

        <!-- Enable annotation-driven caching. -->
        <cache:annotation-driven/>
    </beans>
```

It's possible that you already have an Ignite node running when the cache manager is initialized (e.g., it was started using `ServletContextListenerStartup`). In this case you should simply provide the grid name via `gridName` property. Note that if you don't set the grid name as well, the cache manager will try to use the default Ignite instance (the one with the `null` name). Here is the example:

## Using pre-started Ignite instance

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache
        http://www.springframework.org/schema/cache/spring-cache.xsd">
    <!-- Provide grid name. -->
    <bean id="cacheManager" class="org.apache.ignite.cache.spring.SpringCacheManager">
        <property name="gridName" value="myGrid"/>
    </bean>

    <!-- Enable annotation-driven caching. -->
    <cache:annotation-driven/>
</beans>
```



**Remote Nodes.** Keep in mind that the node started inside your application is an entry point to the whole topology it connects to. You can start as many remote standalone nodes as you need using `bin/ignite.{sh|bat}` scripts provided in Ignite distribution, and all these nodes will participate in caching the data.

## Dynamic Caches

While you can have all required caches predefined in Ignite configuration, it's not required. If Spring wants to use a cache that doesn't exist, the `SpringCacheManager` will automatically create it.

If otherwise not specified, a new cache will be created with all defaults. To customize it, you can provide a configuration template via `dynamicCacheConfiguration` property. For example, if you want to use `REPLICATED` caches instead of `PARTITIONED`, you should configure `SpringCacheManager` like this:

### Dynamic cache configuration

```
<bean id="cacheManager" class="org.apache.ignite.cache.spring.SpringCacheManager">
    ...
    <property name="dynamicCacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <property name="cacheMode" value="REPLICATED"/>
        </bean>
    </property>
</bean>
```

You can also utilize near caches on client side. To achieve this simply provide near cache configuration via `dynamicNearCacheConfiguration` property. By default near cache is not created. Here is the example:

#### Dynamic near cache configuration

```
<bean id="cacheManager" class="org.apache.ignite.cache.spring.SpringCacheManager">
    ...
    <property name="dynamicNearCacheConfiguration">
        <bean class="org.apache.ignite.configuration.NearCacheConfiguration">
            <property name="nearStartSize" value="1000"/>
        </bean>
    </property>
</bean>
```

## Example

Once you added `SpringCacheManager` to your Spring application context, you can enable caching for any Java method by simply attaching annotation to it.

Usually you would use caching for heavy operations, like database access. For example, let's assume you have a DAO class with `averageSalary(...)` method that calculates average salary of all employees in an organization. You can use `@Cacheable` annotation to enable caching for this method:

```
private JdbcTemplate jdbc;

@Cacheable("averageSalary")
public long averageSalary(int organizationId) {
    String sql =
        "SELECT AVG(e.salary) " +
        "FROM Employee e " +
        "WHERE e.organizationId = ?";

    return jdbc.queryForObject(sql, Long.class, organizationId);
}
```

When this method is called for the first time, `SpringCacheManager` will automatically create `averageSalary` cache. It will also lookup the pre-calculated average value in this cache and return it right away if it's there. If the average for this organization is not calculated yet, the method will be called and the result will be stored in cache. So next time you request average for this organization, you will not query the database.



**Cache Key.** Since `organizationId` is the only method parameter, it will be automatically used as a cache key. If the salary of one of the employees is changed, you may want to remove the average value for the organization this employee belongs to, because otherwise `averageSalary(...)` method will return obsolete cached result. This can be achieved with `@CacheEvict` annotation attached to a method that updates employee's salary:

```
private JdbcTemplate jdbc;

@CacheEvict(value = "averageSalary", key = "#e.organizationId")
public void updateSalary(Employee e) {
    String sql =
        "UPDATE Employee " +
        "SET salary = ? " +
        "WHERE id = ?";

    jdbc.update(sql, e.getSalary(), e.getId());
}
```

After this method is called, average value for provided employee's organization will be evicted from `averageSalary` cache. This will force `averageSalary(...)` to recalculate the value next time it's called.



**Spring Expression Language (SpEL).** Note that this method receives employee as a parameter, while average values are saved in cache by organization ID. To explicitly specify what is used as a cache key, we used `key` parameter of the annotation and [Spring Expression Language](#) method will be called on provided employee object and the returned value will be used as the cache key.

## Topology Validation

Topology validator is used to verify that cluster topology is valid for further cache operations.

The topology validator is invoked every time the cluster topology changes (either a new node joined or an existing node failed or left). If topology validator is not configured, then the cluster topology is always considered to be valid.

Whenever the `TopologyValidator.validate(Collection)` method returns true, then the topology is considered valid for a certain cache and all operations on this cache will be allowed to proceed. Otherwise, all update operations on the cache are restricted with the following exceptions: - `CacheException` will be thrown for all update operations (put, remove, etc) attempt. - `IgniteException` will be thrown for the transaction commit attempt.

After returning false and declaring the topology not valid, the topology validator can return to normal state whenever the next topology change happens.

## Example

### *Setup example*

```
...
for (CacheConfiguration cCfg : iCfg.getCacheConfiguration()) {
    if (cCfg.getName() != null) {
        if (cCfg.getName().equals(CACHE_NAME_1))
            cCfg.setTopologyValidator(new TopologyValidator() {
                @Override public boolean validate(Collection<ClusterNode> nodes) {
                    return nodes.size() == 2;
                }
            });
        else if (cCfg.getName().equals(CACHE_NAME_2))
            cCfg.setTopologyValidator(new TopologyValidator() {
                @Override public boolean validate(Collection<ClusterNode> nodes) {
                    return nodes.size() >= 2;
                }
            });
    }
}
...
}
```

In this example update operations will be allowed to cache with name - `CACHE_NAME_1` in case cluster contains exactly 2 nodes - `CACHE_NAME_2` in case cluster contain at least 2 nodes.

Configuration The topology validator can be configured either from code or XML via `CacheConfiguration.setTopologyValidator(TopologyValidator)` method.

# Interactive SQL

# Ignite with Apache Zeppelin

Run Ignite SQL interactively with Apache Zeppelin

---

## Overview

---

[Apache Zeppelin](#), a web-based notebook that enables interactive data analytics. You can make beautiful data-driven, interactive and collaborative documents with SQL, Scala and more.

You can use Zeppelin to retrieve distributed data from cache using Ignite SQL interpreter. Moreover, Ignite interpreter allows you to execute any Scala code in cases when SQL doesn't fit to your requirements. For example you can populate data into your caches or execute distributed computations.

## Zeppelin Installation and Configuration

---

In order to start using Ignite interpreters you should install Zeppelin in two simple steps: 1. Clone Zeppelin Git repository `git clone https://github.com/apache/incubator-zeppelin.git` 2. Build Zeppelin from sources `cd incubator-zeppelin mvn clean install -Dignite-version=1.2.0-incubating -DskipTests`



**Building Zeppelin with specific Ignite version.** You can use `ignite-version` property for build Zeppelin with specific Ignite version. Use version `1.1.0-incubating` or later.



**Adding Ignite Interpreters.** By default Ignite and Ignite SQL interpreters are already configured in Zeppelin. Otherwise you should add the following interpreters class names to the corresponding configuration file or environment variable (see "Configure" section of [Zeppelin installation guide](#)):

- `org.apache.zeppelin.ignite.IgniteInterpreter`
- `org.apache.zeppelin.ignite.IgniteSqlInterpreter`

**Note:** First interpreter become a default. Once Zeppelin are installed and configured you can start it using command `./bin/zeppelin-daemon.sh start` and open start page in your browser (default start page URL is <http://localhost:8080>).

# Welcome to Zeppelin!

Zeppelin is web-based notebook that enables interactive data analytics.

You can make beautiful data-driven, interactive, collaborative document with SQL, code and even more!



## Notebook

[Create new note](#)

- [Note 2AS1JPAMC](#)
- [Note 2AS624UKF](#)
- [Note 2AT41NT92](#)
- [Note 2ATRZTVTK](#)
- [Note 2AUCUEYVP](#)
- [Zeppelin Tutorial](#)

## Help

Get started with [Zeppelin documentation](#)

## Community

Please feel free to help us to improve Zeppelin,

Any contribution are welcome!

See also [Zeppelin installation documentation](#).

## Configuring Ignite Interpreters

---

Click on "Interpreter" menu item. This page contains settings for all configured interpreter groups. Scroll down to the "Ignite" section and modify properties as you need using "Edit" button. Click "Save" button to save changes in configuration. Don't forget restart interpreter after changes in configuration.



ignite %ignite , %ignitesql

[edit](#) [restart](#) [remove](#)

Properties		
name	value	action
ignite.addresses	127.0.0.1:47500..47509	<a href="#">x</a>
ignite.clientMode	true	<a href="#">x</a>
ignite.config.url		<a href="#">x</a>
ignite.jdbc.url	jdbc:ignite://localhost:11211/words	<a href="#">x</a>
ignite.peerClassLoadingEnabled	true	<a href="#">x</a>
		<a href="#">+</a>

[Save](#) [Cancel](#)

## #Configuring Ignite SQL Interpreter

Ignite SQL interpreter requires only `ignite.jdbc.url` property that contains JDBC connection URL. In our example we will use `words` cache. Edit `ignite.jdbc.url` property setting the following value: `jdbc:ignite://localhost:11211/words`.

See also [JDBC Driver](#) section for details.

## #Configuring Ignite Interpreter

For most simple cases Ignite interpreter requires the following properties:

- `ignite.addresses` - Comma separated list of Ignite cluster hosts. See [Cluster Configuration](#) section for details.
- `ignite.clientMode` - You can connect to the Ignite cluster as client or server node. See [Clients vs. Servers](#) section for details. Use `true` or `false` values in order to connect in client or server mode respectively.
- `ignite.peerClassLoadingEnabled` - Enables peer-class-loading. See [Zero Deployment](#) section for details. Use `true` or `false` values in order to enable or disable P2P class loading respectively.

For more complicated cases you can define own configuration of Ignite using `ignite.config.url` property that contains URL to Ignite configuration file. Note that if `ignite.config.url` property is defined then all aforementioned properties will be ignored.

# Using Ignite Interpreters

## #Starting Ignite cluster

Before using Zeppelin we need start Ignite cluster. Download [Apache Ignite In-Memory Data Fabric binary release](#) and unpack the downloaded archive: `unzip apache-ignite-fabric-1.2.0-incubating-bin.zip -d <dest_dir>`

Examples are shipped as a separate Maven project, so to start running you simply need to import provided `<dest_dir>/apache-ignite-fabric-1.2.0-incubating-bin/pom.xml` file into your favourite IDE.

Start the following examples:

- `org.apache.ignite.examples.ExampleNodeStartup` - starts one Ignite node. You can start one or more nodes.
- `org.apache.ignite.examples.streaming.wordcount.StreamWords` - starts client node that continuously streams words into `words` cache.

Now you are ready for using Zeppelin for accesing to our Ignite cluster.

## #Creating new note in Zeppelin

Create new (or open existing) note using "Notebook" menu item.

The screenshot shows the Zeppelin Notebook interface. At the top, there's a navigation bar with the Zeppelin logo, 'Notebook', 'Interpreter', and a 'Connected' status indicator. Below the navigation bar, a modal dialog is open for creating a new note. The dialog has a title 'ignite' and a subtitle '%ignite, %ignite'. It contains a properties section with the following entries:

Properties	Value	Action
name		edit
ignite.addresses	500..47509	remove
ignite.clientMode	true	remove
ignite.config.url		remove
ignite.jdbc.url	jdbc:ignite://localhost:11211/words	remove
ignite.peerClassLoadingEnabled	true	remove

At the bottom of the dialog, there are 'Save' and 'Cancel' buttons. A red box highlights the 'Create new note' button at the top right of the dialog.

After creating new note you should click "Notebook" menu item again and open created note. Click to note's name in order to rename it. Enter new title and press "Enter" key.

The screenshot shows the Zeppelin Notebook interface. At the top, there is a header bar with the Zeppelin logo, a "Notebook" dropdown, and an "Interpreter" button. A green "Connected" status indicator is on the right. Below the header, the title "Note 2ARR86ZSZ" is displayed in a red-bordered box. To the right of the title are several icons: a play button, a refresh, a save, a trash, and a circular icon. On the far right, there are help, settings, and a dropdown menu labeled "default". The main area below the title has a "READY" status and a toolbar with icons for play, refresh, save, and settings. The text input field is currently empty.

Since note is created you can input SQL query or Scala code and execute it clicking to "Execute" button (blue triangle icon).

The screenshot shows the Zeppelin Notebook interface with the title "Ignite example". The text input field at the bottom is highlighted with a red border. The rest of the interface is similar to the previous screenshot, with the "READY" status and a toolbar below the input field.

## #Using Ignite SQL interpreter

For execute SQL query use `%ignite.ignitesql` prefix and your SQL query. For example we can select top 10 words in our `words` cache using the following query:

```
%ignite.ignitesql select _val, count(_val) as cnt from String group by _val order by cnt desc limit 10
```

The screenshot shows the Zeppelin Notebook interface with the title "Ignite example". The text input field contains the query: `%ignite.ignitesql select _val, count(_val) as cnt from String group by _val order by cnt desc limit 10`. The "READY" status and toolbar are visible below the input field.

After executing this example you can see result as table or graph. Use corresponding icons to change view.

## Ignite example


? ⚙️ default ▾

```
%ignite.ignitesql select _val, count(_val) as cnt from String group by _val order by cnt desc limit 10
```

FINISHED ✖️ ⟳ 🖨️ ⚙️



_VAL	CNT
the	624
to	258
and	237
a	223
of	206
said	186
she	141
in	138
it	108

Took 1 seconds

## Ignite example


? ⚙️ default ▾

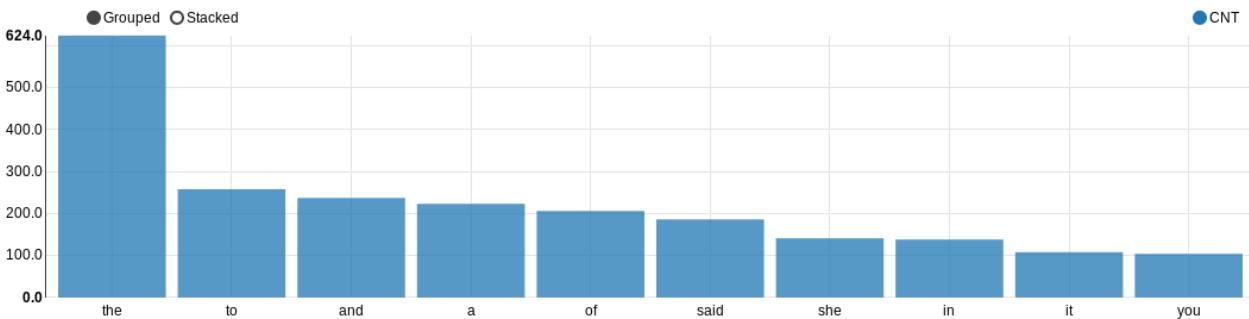
```
%ignite.ignitesql select _val, count(_val) as cnt from String group by _val order by cnt desc limit 10
```

FINISHED ✖️ ⟳ 🖨️ ⚙️



SETTINGS ▾

● CNT



Took 1 seconds

## Ignite example

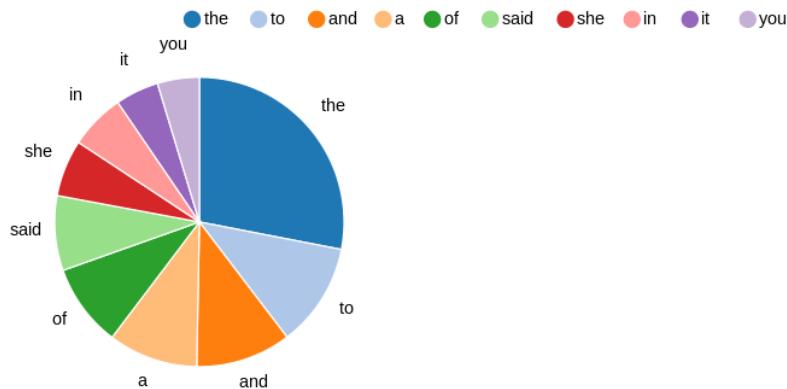
default ▾

```
%ignite.ignitesql select _val, count(_val) as cnt from String group by _val order by cnt desc limit 10
```

FINISHED



SETTINGS ▾



Took 1 seconds

## #Using Ignite interpreter

For execute Scala code snippet use `%ignite` prefix and your code snippet. For example we can select average, min and max counts among all the words:

```
%ignite
import org.apache.ignite._
import org.apache.ignite.cache.affinity._
import org.apache.ignite.cache.query._
import org.apache.ignite.configuration._

import scala.collection.JavaConversions._

val cache: IgniteCache[AffinityUuid, String] = ignite.cache("words")

val qry = new SqlFieldsQuery("select avg(cnt), min(cnt), max(cnt) from (select
count(_val) as cnt from String group by _val)", true)

val res = cache.query(qry).getAll()

collectionAsScalaIterable(res).foreach(println _)
```



## Ignite example



```
%ignite
import org.apache.ignite._
import org.apache.ignite.cache.affinity._
import org.apache.ignite.cache.query._
import org.apache.ignite.configuration._

import scala.collection.JavaConversions._

val cache: IgniteCache[AffinityUuid, String] = ignite.cache("words")

val qry = new SqlFieldsQuery("select avg(cnt), min(cnt), max(cnt) from (select count(_val) as cnt from String group by _val)", true)

val res = cache.query(qry).getAll()

collectionAsScalaIterable(res).foreach(println _)
```

READY ▶ ↻ 🔍 ⚙

After executing this example you will see output of Scala REPL.



## Ignite example



```
%ignite
import org.apache.ignite._
import org.apache.ignite.cache.affinity._
import org.apache.ignite.cache.query._
import org.apache.ignite.configuration._

import scala.collection.JavaConversions._

val cache: IgniteCache[AffinityUuid, String] = ignite.cache("words")

val qry = new SqlFieldsQuery("select avg(cnt), min(cnt), max(cnt) from (select count(_val) as cnt from String group by _val)", true)

val res = cache.query(qry).getAll()

collectionAsScalaIterable(res).foreach(println _)

import org.apache.ignite._

import org.apache.ignite.cache.affinity._

import org.apache.ignite.cache.query._

import org.apache.ignite.configuration._

import scala.collection.JavaConversions._

cache: org.apache.ignite.IgniteCache[org.apache.ignite.cache.affinity.AffinityUuid, String] = IgniteCacheProxy [delegate=GridDhtAtomicCache [updateReplyClos=or
qry: org.apache.ignite.cache.query.SqlFieldsQuery = SqlFieldsQuery [sql=select avg(cnt), min(cnt), max(cnt) from (select count(_val) as cnt from String group
res: java.util.List[java.util.List[_]] = [[3, 1, 220]]
[3, 1, 220]

Took 17 seconds
```

FINISHED ▶ ↻ 🔍 ⚙



Note that Ignite version of your Ignite cluster and Zeppelin installation must be equal.

# Streaming & CEP

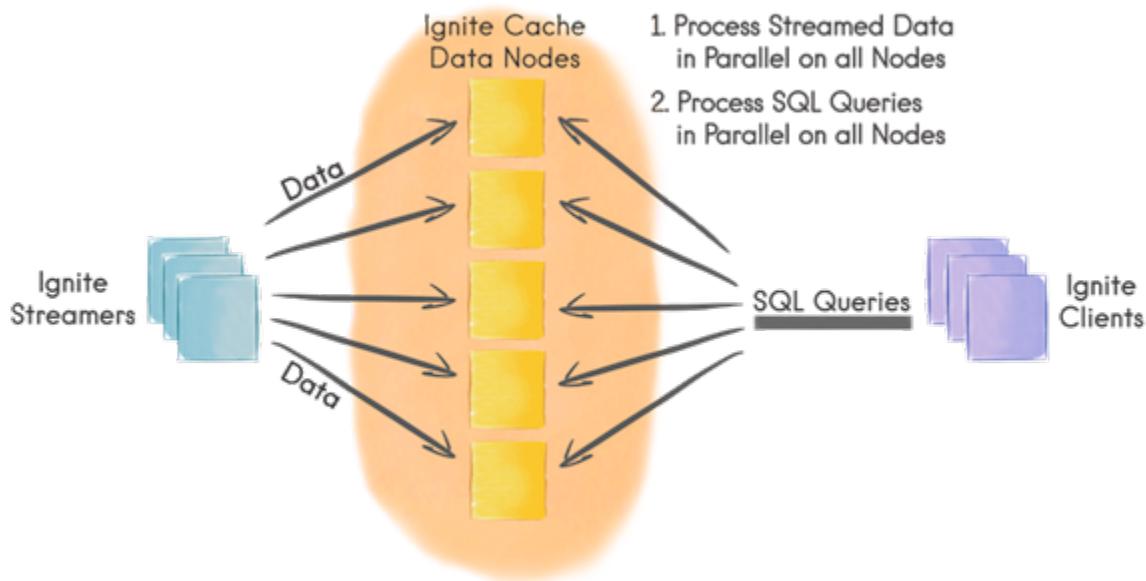
# Streaming & CEP

Easily stream large volumes of data into Ignite

Ignite streaming allows to process continuous never-ending streams of data in scalable and fault-tolerant fashion. The rates at which data can be injected into Ignite can be very high and easily exceed millions of events per second on a moderately sized cluster.

## How it Works

1. Client nodes inject finite or continuous streams of data into Ignite caches using Ignite [Data Streamers](#).
2. Data is automatically partitioned between Ignite data nodes, and each node gets equal amount of data.
3. Streamed data can be concurrently processed directly on the Ignite data nodes in collocated fashion.
4. Clients can also perform concurrent SQL queries on the streamed data.



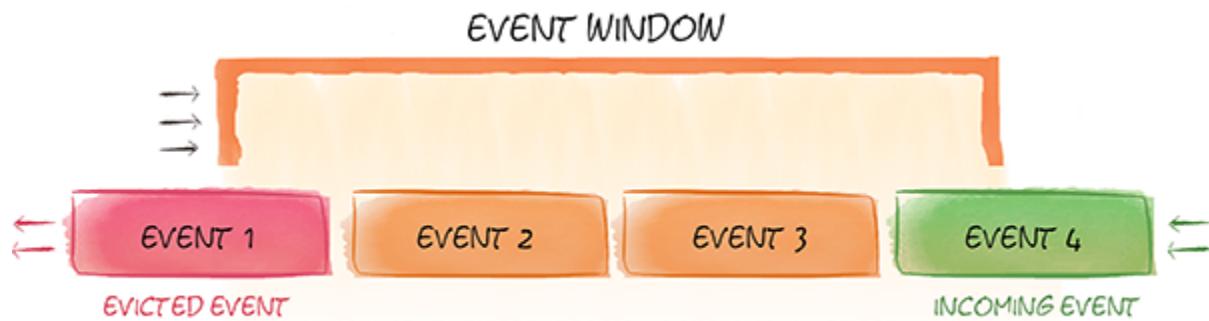
## Data Streamers

Data streamers are defined by [IgniteDataStreamer](#) API and are built to inject large amounts of continuous streams of data into Ignite stream caches. Data streamers are built in a scalable and fault-tolerant fashion and provide **at-least-once-guarantee** semantics for all the data streamed into Ignite.

### Data Streamers

## Sliding Windows

Ignite streaming functionality also allows to query into **sliding windows** of data. Since streaming data never ends, you rarely want to query the whole data set going back to the very beginning. Instead, you are more interested in questions like “What are the 10 most popular products over last 2 hours?”, or “What is the average product price in a certain category for the past day?”. To achieve this, you need to be able to query into **sliding data windows**.



Sliding windows are configured as Ignite cache eviction policies, and can be time-based, size-based, or batch-based. You can configure one sliding-window per cache. However, you can easily define more than one cache if you need different sliding windows for the same data.

## [Sliding Windows](#)

## Querying Data

You can use full set of Ignite data indexing capabilities, together with Ignite SQL, TEXT, and Predicate based cache queries, to query into the streaming data.

## [Cache Queries](#)

## Data Streamers

Stream large amounts of data into Ignite caches.

---

Data streamers are defined by [IgniteDataStreamer](#) API and are built to inject large amounts of continuous streams of data into Ignite caches. Data streamers are built in a scalable and fault-tolerant fashion and provide **at-least-once-guarantee** semantics for all the data streamed into Ignite.

## [IgniteDataStreamer](#)

---

The main abstraction for fast streaming of large amounts of data into Ignite is [IgniteDataStreamer](#), which internally will properly batch keys together and collocate those batches with nodes on which

the data will be cached.

The high loading speed is achieved with the following techniques:

- Entries that are mapped to the same cluster member will be batched together in a buffer.
- Multiple buffers can coexist at the same time.
- To avoid running out of memory, data streamer has a maximum number of buffers it can process concurrently.

To add data to the data streamer, you should call `IgniteDataStreamer.addData(...)` method.

```
// Get the data streamer reference and stream data.  
try (IgniteDataStreamer<Integer, String> stmr = ignite.dataStreamer("myStreamCache")) {  
    // Stream entries.  
    for (int i = 0; i < 100000; i++)  
        stmr.addData(i, Integer.toString(i));  
}
```

## Allow Overwrite

By default, the data streamer will not overwrite existing data, which means that if it will encounter an entry that is already in cache, it will skip it. This is the most efficient and performant mode, as the data streamer does not have to worry about data versioning in the background.

If you anticipate that the data may already be in the streaming cache and you need to overwrite it, you should set `IgniteDataStreamer.allowOverwrite(true)` parameter.

## StreamReceiver

---

For cases when you need to execute some custom logic instead of just adding new data, you can take advantage of `StreamReceiver` API.

Stream receivers allow you to react to the streamed data in collocated fashion, directly on the nodes where it will be cached. You can change the data or add any custom pre-processing logic to it, before putting the data into cache.



Note that `StreamReceiver` does not put data into cache automatically. You need to call any of the `cache.put(...)` methods explicitly.

## StreamTransformer

`StreamTransformer` is a convenience implementation of `StreamReceiver` which updates data in the stream cache based on its previous value. The update is collocated, i.e. it happens exactly on the cluster node where the data is stored.

In the example below, we use `StreamTransformer` to increment a counter for each distinct word found in the text stream.

*transformer*

```
CacheConfiguration cfg = new CacheConfiguration("wordCountCache");

IgniteCache<String, Long> stmCache = ignite.getOrCreateCache(cfg);

try (IgniteDataStreamer<String, Long> stmr = ignite.dataStreamer(stmCache.getName())) {
    // Allow data updates.
    stmr.allowOverwrite(true);

    // Configure data transformation to count instances of the same word.
    stmr.receiver(StreamTransformer.from((e, arg) -> {
        // Get current count.
        Long val = e.getValue();

        // Increment count by 1.
        e.setValue(val == null ? 1L : val + 1);

        return null;
    }));
}

// Stream words into the streamer cache.
for (String word : text)
    stmr.addData(word, 1L);
}
```

```

CacheConfiguration cfg = new CacheConfiguration("wordCountCache");

IgniteCache<Integer, Long> stmCache = ignite.getOrCreateCache(cfg);

try (IgniteDataStreamer<String, Long> stmr = ignite.dataStreamer(stmCache.getName())) {
    // Allow data updates.
    stmr.allowOverwrite(true);

    // Configure data transformation to count instances of the same word.
    stmr.receiver(new StreamTransformer<String, Long>() {
        @Override public Object process(MutableEntry<String, Long> e, Object... args) {
            // Get current count.
            Long val = e.getValue();

            // Increment count by 1.
            e.setValue(val == null ? 1L : val + 1);

            return null;
        }
    });

    // Stream words into the streamer cache.
    for (String word : text)
        stmr.addData(word, 1L);
}

```

## StreamVisitor

[StreamVisitor](#) is also a convenience implementation of [StreamReceiver](#) which visits every key-value tuple in the stream. Note, that the visitor does not update the cache. If the tuple needs to be stored in the cache, then any of the `cache.put(...)` methods should be called explicitly.

In the example below, we have 2 caches: "marketData", and "instruments". We receive market data ticks and put them into the streamer for the "marketData" cache. The [StreamVisitor](#) for the "marketData" streamer is invoked on the cluster member mapped to the particular market symbol. Upon receiving individual market ticks it updates the "instrument" cache with latest market price.

Note, that we do not update "marketData" cache at all, leaving it empty. We simply use for collocated processing of the market data within the cluster directly on the node where the data will be stored.

```

CacheConfiguration<String, Double> mrktDataCfg = new CacheConfiguration<>("marketData");
CacheConfiguration<String, Double> instCfg = new CacheConfiguration<>("instruments");

// Cache for market data ticks streamed into the system.
IgniteCache<String, Double> mrktData = ignite.getOrCreateCache(mrktDataCfg);

// Cache for financial instruments.
IgniteCache<String, Double> insts = ignite.getOrCreateCache(instCfg);

try (IgniteDataStream<String, Integer> mktStmr = ignite.dataStreamer("marketData")) {
    // Note that we do not populate 'marketData' cache (it remains empty).
    // Instead we update the 'instruments' cache based on the latest market price.
    mktStmr.receiver(StreamVisitor.from((cache, e) -> {
        String symbol = e.getKey();
        Double tick = e.getValue();

        Instrument inst = instCache.get(symbol);

        if (inst == null)
            inst = new Instrument(symbol);

        // Update instrument price based on the latest market tick.
        inst.setHigh(Math.max(inst.getLatest(), tick));
        inst.setLow(Math.min(inst.getLatest(), tick));
        inst.setLatest(tick);

        // Update instrument cache.
        instCache.put(symbol, inst);
    }));
}

// Stream market data into Ignite.
for (Map.Entry<String, Double> tick : marketData)
    mktStmr.addData(tick);
}

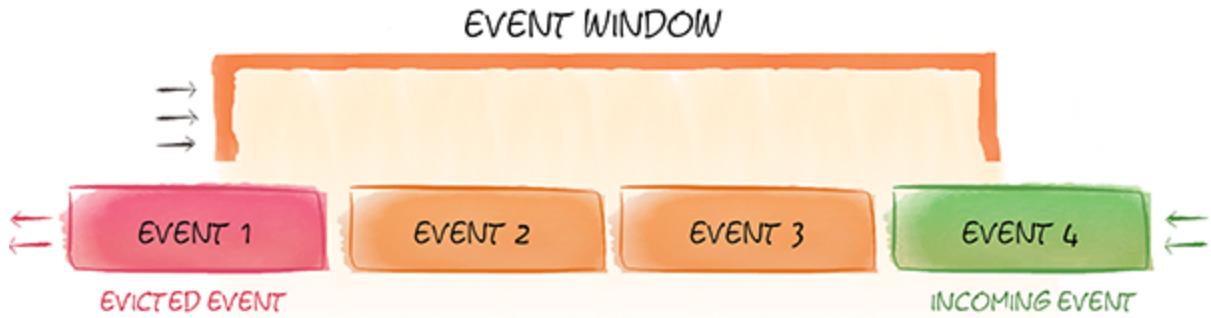
```

## Sliding Windows

Configure sliding windows into streaming data.

---

Sliding windows are configured as Ignite cache eviction policies, and can be time-based, size-based, or batch-based. You can configure one sliding-window per cache. However, you can easily define more than one cache if you need different sliding windows for the same data.



## Time-Based Sliding Windows

Time-based windows are configured using JCache-compliant [ExpiryPolicy](#). You can have streamed events expire based on **create time**, **last-access time**, or **update time**.

Here is how you can configure a 5-second sliding window based on creation time in Ignite.

```
CacheConfiguration<Integer, Long> cfg = new CacheConfiguration<>("myStreamCache");

// Sliding window of 5 seconds based on creation time.
cfg.setExpiryPolicyFactory(FactoryBuilder.factoryOf(
    new CreatedExpiryPolicy(new Duration(SECONDS, 5))));
```

## FIFO Sliding Window

FIFO (first-in-first-out) sliding windows are configured via [FifoEvictionPolicy](#) in Ignite caches. This policy is size-based. Stream tuples are inserted into the window until cache size reaches its maximum limit. Then the oldest tuples start getting evicted automatically.

Here is how you can configure a FIFO sliding window holding 1,000,000 of stream tuples.

```
CacheConfiguration<Integer, Long> cfg = new CacheConfiguration<>("myStreamCache");

// FIFO window holding 1,000,000 entries.
cfg.setEvictionPolicyFactory(new FifoEvictionPolicy(1_000_000));
```

## LRU Sliding Window

LRU (least-recently-used) sliding windows are configured via [LruEvictionPolicy](#) in Ignite caches. This policy is size-based. Stream tuples are inserted into the window until cache size reaches its maximum limit. Then the least recently accessed tuples start getting evicted automatically.

Here is how you can configure LRU sliding window holding 1,000,000 of most recently accessed data.

```

CacheConfiguration<Integer, Long> cfg = new CacheConfiguration<>("myStreamCache");

// LRU window holding 1,000,000 entries.
cfg.setEvictionPolicyFactory(new LruEvictionPolicy(1_000_000));

```

## Querying Sliding Windows

Sliding windows can be queried in the same way as any other Ignite caches, using Predicate-based, SQL, or TEXT queries.

Here is an example of how a cache holding a sliding window of financial instruments streamed into the system can be queried using SQL queries.

First we need to create indexes based on the queried fields. In this case we are indexing the fields for **Instrument** class and the **String** keys.

```

CacheConfiguration<String, Instrument> cfg = new CacheConfiguration<>("instCache");

// Index some fields for querying portfolio positions.
cfg.setIndexedTypes(String.class, Instrument.class);

// Get a handle on the cache (create it if necessary).
IgniteCache<String, Instrument> instCache = ignite.getOrCreateCache(cfg);

```

Let's query top 3 best performing financial instruments. We do it via ordering by **(latest - open)** price and selecting top 3.

```

// Select top 3 best performing instruments.
SqlFieldsQuery top3qry = new SqlFieldsQuery(
    "select symbol, (latest - open) from Instrument order by (latest - open) desc limit 3"
);

// List of rows. Every row is represented as a List as well.
List<List<?>> top3 = instCache.query(top3qry).getAll();

```

Let's query total profit across all financial instruments. We do it by adding up all **(latest - open)** values across all instruments.

```
// Select total profit across all financial instruments.  
SqlFieldsQuery profitQry = new SqlFieldsQuery("select sum(latest - open) from Instrument  
");  
  
List<List<?>> profit = instCache.query(profitQry).getAll();  
  
System.out.printf("Total profit: %.2f%n", row.get(0));
```

## Word Count Example

Stream text into Ignite and count occurrences of each words.

---

In this example we will stream text into Ignite and count each individual word. We will also issue periodic SQL queries into the stream to query top 10 most popular words.

The example will work as follows: 1. We will setup up a cache to hold the words and their counts. 2. We will setup a 5 second **sliding window** to keep the word counts only for last 5 seconds. 3. **StreamWords** program will stream text data into Ignite. 4. **QueryWords** program will query top 10 words out of the stream.

### Cache Configuration

---

We define a **CacheConfig** class which will provide configuration to be used from both programs, **StreamWords** and **QueryWords**. The cache will use words as keys, and counts for words as values.

Note that in this example we use a sliding window of 5 seconds for our cache. This means that words will disappear from cache after 5 seconds since they were first entered into cache.

```

public class CacheConfig {
    public static CacheConfiguration<String, Long> wordCache() {
        CacheConfiguration<String, Long> cfg = new CacheConfiguration<>("words");

        // Index the words and their counts,
        // so we can use them for fast SQL querying.
        cfg.setIndexedTypes(String.class, Long.class);

        // Sliding window of 5 seconds.
        cfg.setExpiryPolicyFactory(FactoryBuilder.factoryOf(
            new CreatedExpiryPolicy(new Duration(SECONDS, 5))));

        return cfg;
    }
}

```

## Stream Words

We define a `StreamWords` class which will be responsible to continuously read words from a local text file ("alice-in-wonderland.txt" in our case) and stream them into Ignite "words" cache.

### Streamer Configuration

1. We set `allowOverwrite` flag to `true` to make sure that existing counts can be updated.
2. We configure a `StreamTransformer` which takes currently cache count for a word and increments it by 1.

```

public class StreamWords {
    public static void main(String[] args) throws Exception {
        // Mark this cluster member as client.
        Ignition.setClientMode(true);

        try (Ignite ignite = Ignition.start()) {
            IgniteCache<String, Long> stmCache = ignite.getOrCreateCache(CacheConfig.wordCache());
        }

        // Create a streamer to stream words into the cache.
        try (IgniteDataStream<String, Long> stmr = ignite.dataStreamer(stmCache.getName())) {
            // Allow data updates.
            stmr.allowOverwrite(true);

            // Configure data transformation to count instances of the same word.
        }
    }
}

```

```

    stmr.receiver(StreamTransformer.from((e, arg) -> {
        // Get current count.
        Long val = e.getValue();

        // Increment current count by 1.
        e.setValue(val == null ? 1L : val + 1);

        return null;
    }));

    // Stream words from "alice-in-wonderland" book.
    while (true) {
        Path path = Paths.get(StreamWords.class.getResource("alice-in-wonderland.txt")
            .toURI());

        // Read words from a text file.
        try (Stream<String> lines = Files.lines(path)) {
            lines.forEach(line -> {
                Stream<String> words = Stream.of(line.split(" "));

                // Stream words into Ignite streamer.
                words.forEach(word -> {
                    if (!word.trim().isEmpty())
                        stmr.addData(word, 1L);
                });
            });
        }
    }
}

```

## Query Words

We define a `QueryWords` class which will periodically query word counts from the cache.

## SQL Query

1. We use standard SQL to query the counts.
2. Ignite SQL treats Java classes as SQL tables. Since our counts are stored as simple `Long` type, the SQL query below queries `Long` table.
3. Ignite always stores cache keys and values as `_key` and `_val` fields, so we use this syntax in our SQL query.

```

public class QueryWords {
    public static void main(String[] args) throws Exception {
        // Mark this cluster member as client.
        Ignition.setClientMode(true);

        try (Ignite ignite = Ignition.start()) {
            IgniteCache<String, Long> stmCache = ignite.getOrCreateCache(CacheConfig.wordCache());
        }

        // Select top 10 words.
        SqlFieldsQuery top10Qry = new SqlFieldsQuery(
            "select _key, _val from Long order by _val desc limit 10");

        // Query top 10 popular words every 5 seconds.
        while (true) {
            // Execute queries.
            List<List<?>> top10 = stmCache.query(top10Qry).getAll();

            // Print top 10 words.
            ExamplesUtils.printQueryResults(top10);

            Thread.sleep(5000);
        }
    }
}

```

## Starting Server Nodes

In order to run the example, you need to start data nodes. In Ignite, data nodes are called **server** nodes. You can start as many server nodes as you like, but you should have at least 1 in order to run the example.

Server nodes can be started from command line as follows:

```
bin/ignite.sh
```

You can also start server nodes programmatically, like so:

```
public class ExampleNodeStartup {  
    public static void main(String[] args) throws IgniteException {  
        Ignition.start();  
    }  
}
```

## JMS Data Streamer

Ignite offers a JMS Data Streamer to consume messages from JMS brokers, convert them into cache tuples and insert them in Ignite caches.

### Features supported

This data streamer supports the following features:

- Consumes from queues or topics.
- Supports durable subscriptions from topics.
- Concurrent consumers are supported via the `threads` parameter.
- When consuming from queues, this component will start as many `Session` objects with separate `MessageListener` instances each, therefore achieving **natural** concurrency.
- When consuming from topics, obviously we cannot start multiple threads as that would lead us to consume duplicate messages. Therefore, we achieve concurrency in a **virtualized** manner through an internal thread pool.
- Transacted sessions are supported through the `transacted` parameter.
- Batched consumption is possible via the `batched` parameter, which groups message reception within the scope of a local JMS transaction (XA not used supported). Depending on the broker, this technique can provide a higher throughput as it decreases the amount of message acknowledgement round trips that are necessary, albeit at the expense possible duplicate messages (especially if an incident occurs in the middle of a transaction).
- Batches are committed when the `batchClosureMillis` time has elapsed, or when a Session has received at least `batchClosureSize` messages.
- Time-based closure fires with the specified frequency and applies to all `Session`'s in parallel.
- Size-based closure applies to each individual `Session` (as transactions are `Session-bound` in JMS), so it will fire when that `Session` has processed that many messages.
- Both options are compatible with each other. You can disable either, but not both if batching is enabled.

- Supports specifying the destination with implementation-specific `Destination` objects or with names.

We have tested our implementation against [Apache ActiveMQ](#).

## Instantiating a JMS Streamer

---

When you instantiate the JMS Streamer, you will need to concretualize the following generic types:

- `T` extends `Message` ⇒ the type of JMS `Message` this streamer will receive. If it can receive multiple, use the generic `Message` type.
- `K` ⇒ the type of the cache key.
- `V` ⇒ the type of the cache value.

To configure the JMS streamer, you will need to provide the following compulsory properties:

- `connectionFactory` ⇒ an instance of your `ConnectionFactory` duly configured as required by the broker. It can be a pooled `ConnectionFactory`.
- `destination` or (`destinationName` and `destinationType`) ⇒ a `Destination` object (normally a broker-specific implementation of the JMS `Queue` or `Topic` interfaces), or the combination of a destination name (queue or topic name) and the type as a `Class` reference to either `Queue` or `Topic`. In the latter case, the streamer will use either `Session.createQueue(String)` or `Session.createTopic(String)` to get a hold of the destination.
- `transformer` ⇒ an implementation of `MessageTransformer<T, K, V>` that digests a JMS message of type `T` and produces a `Map<K, V>` of cache entries to add. It can also return `null` or an empty `Map` to ignore the incoming message.

## Example of usage

---

The example in this section populates a cache with `String` keys and `String` values, consuming `TextMessage`'s with this format:

```
raulk,Raul Kripalani  
dsetrakyan,Dmitriy Setrakyan  
sv,Sergi Vladkyin  
gm,Gianfranco Murador
```

Here is the code:

```

// create a data streamer
IgniteDataStreamer<String, String> dataStreamer = ignite.dataStreamer("mycache"));
dataStreamer.allowOverwrite(true);

// create a JMS streamer and plug the data streamer into it
JmsStreamer<TextMessage, String, String> jmsStreamer = new JmsStreamer<>();
jmsStreamer.setIgnite(ignite);
jmsStreamer.setStreamer(dataStreamer);
jmsStreamer.setConnectionFactory(connectionFactory);
jmsStreamer.setDestination(destination);
jmsStreamer.setTransacted(true);
jmsStreamer.setTransformer(new MessageTransformer<TextMessage, String, String>() {
    @Override
    public Map<String, String> apply(TextMessage message) {
        final Map<String, String> answer = new HashMap<>();
        String text;
        try {
            text = message.getText();
        }
        catch (JMSException e) {
            LOG.warn("Could not parse message.", e);
            return Collections.emptyMap();
        }
        for (String s : text.split("\n")) {
            String[] tokens = s.split(",");
            answer.put(tokens[0], tokens[1]);
        }
        return answer;
    }
});
jmsStreamer.start();

// on application shutdown
jmsStreamer.stop();
dataStreamer.close();

```

To use this component, you must import the following module through your build system (Maven, Ivy, Gradle, sbt, etc.):

```

<dependency>
    <groupId>org.apache.ignite</groupId>
    <artifactId>ignite-jms11</artifactId>
    <version>${ignite.version}</version>
</dependency>

```

# Distributed Data Structures

# Queue and Set

Create and distribute Ignite queue and set entries across the cluster.

Ignite In-Memory Data Fabric, in addition to providing standard key-value map-like storage, also provides an implementation of a fast Distributed Blocking Queue and Distributed Set.

`IgniteQueue` and `IgniteSet`, an implementation of `java.util.concurrent.BlockingQueue` and `java.util.Set` interface respectively, also support all operations from `java.util.Collection` interface. Both, queue and set can be created in either collocated or non-collocated mode.

Below is an example of how to create a distributed queue and set.

## Queue

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

IgniteQueue<String> queue = ignite.queue(
    "queueName", // Queue name.
    0,           // Queue capacity. 0 for unbounded queue.
    null         // Collection configuration.
);
```

## Set

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

IgniteSet<String> set = ignite.set(
    "setName", // Queue name.
    null       // Collection configuration.
);
```

## Collocated vs. Non-Collocated Mode

If you plan to create just a few queues or sets containing lots of data, then you would create them in non-collocated mode. This will make sure that about equal portion of each queue or set will be stored on each cluster node. On the other hand, if you plan to have many queues or sets, relatively small in size (compared to the whole cache), then you would most likely create them in collocated mode. In this mode all queue or set elements will be stored on the same cluster node, but about equal amount of queues/sets will be assigned to every node. A collocated queue and set can be created by setting the `collocated` property of `CollectionConfiguration`, like so:

## Queue

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

CollectionConfiguration colCfg = new CollectionConfiguration();

colCfg.setCollocated(true);

// Create colocated queue.
IgniteQueue<String> queue = ignite.queue("queueName", 0, colCfg);
```

## Set

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

CollectionConfiguration colCfg = new CollectionConfiguration();

colCfg.setCollocated(true);

// Create colocated set.
IgniteSet<String> set = ignite.set("setName", colCfg);
```



- . Non-collocated mode only makes sense for and is only supported for **PARTITIONED** caches.

## Bounded Queues

Bounded queues allow users to have many queues with maximum size which gives a better control over the overall cache capacity. They can be either **collocated** or **non-collocated**. When bounded queues are relatively small and used in colocated mode, all queue operations become extremely fast. Moreover, when used in combination with compute grid, users can collocate their compute jobs with cluster nodes on which queues are located to make sure that all operations are local and there is none (or minimal) data distribution.

Here is an example of how a job could be send directly to the node on which a queue resides:

## Queue

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

CollectionConfiguration colCfg = new CollectionConfiguration();

colCfg.setCollocated(true);

final IgniteQueue<String> queue = ignite.queue("queueName", 20, colCfg);

// Add queue elements (queue is cached on some node).
for (int i = 0; i < 20; i++)
    queue.add("Value " + Integer.toString(i));

IgniteRunnable queuePoller = new IgniteRunnable() {
    @Override public void run() throws IgniteException {
        // Poll is local operation due to collocation.
        for (int i = 0; i < 20; i++)
            System.out.println("Polled element: " + queue.poll());
    }
};

// Drain queue on the node where the queue is cached.
ignite.compute().affinityRun("cacheName", "queueName", queuePoller);
```



Refer to [Collocate Compute and Data](#) section for more information on collocating computations with data.

## Cache Queues and Load Balancing

Given that elements will remain in the queue until someone takes them, and that no two nodes should ever receive the same element from the queue, cache queues can be used as an alternate work distribution and load balancing approach within Ignite.

For example, you could simply add computations, such as instances of `IgniteRunnable` to a queue, and have threads on remote nodes call `IgniteQueue.take()` method which will block if queue is empty. Once the `take()` method will return a job, a thread will process it and call `take()` again to get the next job. Given this approach, threads on remote nodes will only start working on the next job when they have completed the previous one, hence creating ideally balanced system where every node only takes the number of jobs it can process, and not more.

## Collection Configuration

Ignite collections can be configured in API via [CollectionConfiguration](#) (see above examples). The following configuration parameters can be used:

Setter Method	<code>setCollocated(boolean)</code>	Description
Default	false	Sets collocation mode.

## Atomic Types

Atomics in Ignite can be read and updated from any node in the cluster.

Ignite supports distributed **atomic long** and **atomic reference**, similar to [java.util.concurrent.atomic.AtomicLong](#) and [java.util.concurrent.atomic.AtomicReference](#) respectively.

Atomics in Ignite are distributed across the cluster, essentially enabling performing atomic operations (such as increment-and-get or compare-and-set) with the same globally-visible value. For example, you could update the value of an atomic long on one node and read it from another node.

## Features

- Retrieve current value.
- Atomically modify current value.
- Atomically increment or decrement current value.
- Atomically compare-and-set the current value to new value.

Distributed atomic long and atomic reference can be obtained via [IgniteAtomicLong](#) and [IgniteAtomicReference](#) interfaces respectively, as shown below:

### *AtomicLong*

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

IgniteAtomicLong atomicLong = ignite.atomicLong(
    "atomicName", // Atomic long name.
    0,           // Initial value.
    false        // Create if it does not exist.
)
```

## *AtomicReference*

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

// Create an AtomicReference.
IgniteAtomicReference<Boolean> ref = ignite.atomicReference(
    "refName", // Reference name.
    "someVal", // Initial value for atomic reference.
    true       // Create if it does not exist.
);
```

Below is a usage example of [IgniteAtomicLong](#) and [IgniteAtomicReference](#):

## *AtomicLong*

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

// Initialize atomic long.
final IgniteAtomicLong atomicLong = ignite.atomicLong("atomicName", 0, true);

// Increment atomic long on local node.
System.out.println("Incremented value: " + atomicLong.incrementAndGet());
```

## *AtomicReference*

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

// Initialize atomic reference.
IgniteAtomicReference<String> ref = ignite.atomicReference("refName", "someVal", true);

// Compare old value to new value and if they are equal,
// only then set the old value to new value.
ref.compareAndSet("WRONG EXPECTED VALUE", "someNewVal"); // Won't change.
```

All atomic operations provided by [IgniteAtomicLong](#) and [IgniteAtomicReference](#) are synchronous. The time an atomic operation will take depends on the number of nodes performing concurrent operations with the same instance of atomic long, the intensity of these operations, and network latency.



- . [IgniteCache](#) interface has [putIfAbsent\(\)](#) and [replace\(\)](#) methods, which provide the same CAS functionality as atomic types.

## **Atomic Configuration**

Atomics in Ignite can be configured via `atomicConfiguration` property of `IgniteConfiguration`. The following configuration parameters can be used :

<code>setBackups(int)</code>	<code>setCacheMode(CacheMode)</code>	<code>setAtomicSequenceReserveSize(int)</code>
Setter Method	Description	Default
Set number of backups.	0	Set cache mode for all atomic types.
<code>PARTITIONED</code>	Sets the number of sequence values reserved for <code>IgniteAtomicSequence</code> instances.	1000

## Example

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="atomicConfiguration">
        <bean class="org.apache.ignite.configuration.AtomicConfiguration">
            <!-- Set number of backups. -->
            <property name="backups" value="1"/>

            <!-- Set number of sequence values to be reserved. -->
            <property name="atomicSequenceReserveSize" value="5000"/>
        </bean>
    </property>
</bean>
```

```
AtomicConfiguration atomicCfg = new AtomicConfiguration();

// Set number of backups.
atomicCfg.setBackups(1);

// Set number of sequence values to be reserved.
atomicCfg.setAtomicSequenceReserveSize(5000);

IgniteConfiguration cfg = new IgniteConfiguration();

// Use atomic configuration in Ignite configuration.
cfg.setAtomicConfiguration(atomicCfg);

// Start Ignite node.
Ignition.start(cfg);
```

# CountDownLatch

Synchronize jobs on all Ignite nodes.

If you are familiar with `java.util.concurrent.CountDownLatch` for synchronization between threads within a single JVM, Ignite provides `IgniteCountDownLatch` to allow similar behavior across cluster nodes.

A distributed CountDownLatch in Ignite can be created as follows:

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

IgniteCountDownLatch latch = ignite.countDownLatch(
    "LatchName", // Latch name.
    10,          // Initial count.
    false        // Auto remove, when counter has reached zero.
    true         // Create if it does not exist.
);
```

After the above code is executed, all nodes in the specified cache will be able to synchronize on the latch named - `latchName`. Below is an example of such synchronization:

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

final IgniteCountDownLatch latch = ignite.countDownLatch("latchName", 10, false, true);

// Execute jobs.
for (int i = 0; i < 10; i++)
    // Execute a job on some remote cluster node.
    ignite.compute().run(() -> {
        int newCnt = latch.countDown();

        System.out.println("Counted down: newCnt=" + newCnt);
    });

// Wait for all jobs to complete.
latch.await();
```

# ID Generator

Sequentially generate unique Ids across the cluster.

Distributed atomic sequence provided by `IgniteCacheAtomicSequence` interface is similar to distributed atomic long, but its value can only go up. It also supports reserving a range of values to avoid costly network trips or cache updates every time a sequence must provide a next value. That is, when you perform `incrementAndGet()` (or any other atomic operation) on an atomic sequence, the data structure reserves ahead a range of values, which are guaranteed to be unique across the cluster for this sequence instance.

Here is an example of how atomic sequence can be created:

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

IgniteAtomicSequence seq = ignite.atomicSequence(
    "seqName", // Sequence name.
    0,         // Initial value for sequence.
    true       // Create if it does not exist.
);
```

Below is a simple usage example:

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

// Initialize atomic sequence.
final IgniteAtomicSequence seq = ignite.atomicSequence("seqName", 0, true);

// Increment atomic sequence.
for (int i = 0; i < 20; i++) {
    long currentValue = seq.get();
    long newValue = seq.incrementAndGet();

    ...
}
```

## Sequence Reserve Size

The key parameter of `IgniteAtomicSequence` is `atomicSequenceReserveSize` which is the number of sequence values reserved, per node . When a node tries to obtain an instance of `IgniteAtomicSequence`, a number of sequence values will be reserved for that node and consequent increments of sequence will happen locally without communication with other nodes, until the next reservation has to be made.

The default value for `atomicSequenceReserveSize` is `1000`. This default setting can be changed by modifying the `atomicSequenceReserveSize` property of `AtomicConfiguration`.



Refer to [Atomic Configuration](/docs/atomic-types#atomic-configuration) for more information on various atomic configuration properties, and examples on how to configure them.

# Memcached

# Memcached Support

Connect to Ignite using Memcached compatible client.

Ignite is [Memcached](#) compliant which enables users to store and retrieve distributed data from Ignite cache using any Memcached compatible client.



Currently, Ignite supports only binary protocol for Memcached. You can connect to Ignite using a Memcached client in one of the following languages:

- [PHP](#)
- [Java](#)
- [Python](#)
- [Ruby](#)

## PHP

To connect to Ignite using PHP client for Memcached, you need to [download Ignite](#) and -

1\. Start Ignite cluster with cache configured. For example :

```
bin/ignite.sh examples/config/example-cache.xml
```

2\. Connect to Ignite using Memcached client, via binary protocol.

```

// Create client instance.
$client = new Memcached();

// Set localhost and port (set to correct values).
$client->addServer("localhost", 11211);

// Force client to use binary protocol.
$client->setOption(Memcached::OPT_BINARY_PROTOCOL, true);

// Put entry to cache.
if ($client->add("key", "val"))
    echo "Successfully put entry in cache.\n";

// Check entry value.
echo("Value for 'key': " . $client->get("key") . "\n");

```

## Java

---

To connect to Ignite using Java client for Memcached, you need to [download Ignite](#) and -

1\. Start Ignite cluster with cache configured. For example:

```
bin/ignite.sh examples/config/example-cache.xml
```

2\. Connect to Ignite using Memcached client, via binary protocol.

```

MemcachedClient client = null;

try {
    client = new MemcachedClient(new BinaryConnectionFactory(),
        AddrUtil.getAddresses("localhost:11211"));
} catch (IOException e) {
    e.printStackTrace();
}

client.set("key", 0, "val");

System.out.println("Value for 'key': " + c.get("key"));

```

# Python

---

To connect to Ignite using Python client for Memcached, you need to [download Ignite](#) and -

- 1\. Start Ignite cluster with cache configured. For example:

```
bin/ignite.sh examples/config/example-cache.xml
```

- 2\. Connect to Ignite using Memcached client, via binary protocol.

```
import pylibmc

client = memcache.Client(["127.0.0.1:11211", binary=True])

client.set("key", "val")

print "Value for 'key': %s" %

client.get("key")
```

# Ruby

---

To connect to Ignite using Ruby client for Memcached, you need to [download Ignite](#) and -

- 1\. Start Ignite cluster with cache configured. For example:

```
bin/ignite.sh examples/config/example-cache.xml
```

- 2\. Connect to Ignite using Memcached client, via binary protocol.

```
require 'dalli'

options = { :namespace => "app_v1", :compress => true }

client = Dalli::Client.new('localhost:11211', options)

client.set('key', 'value')

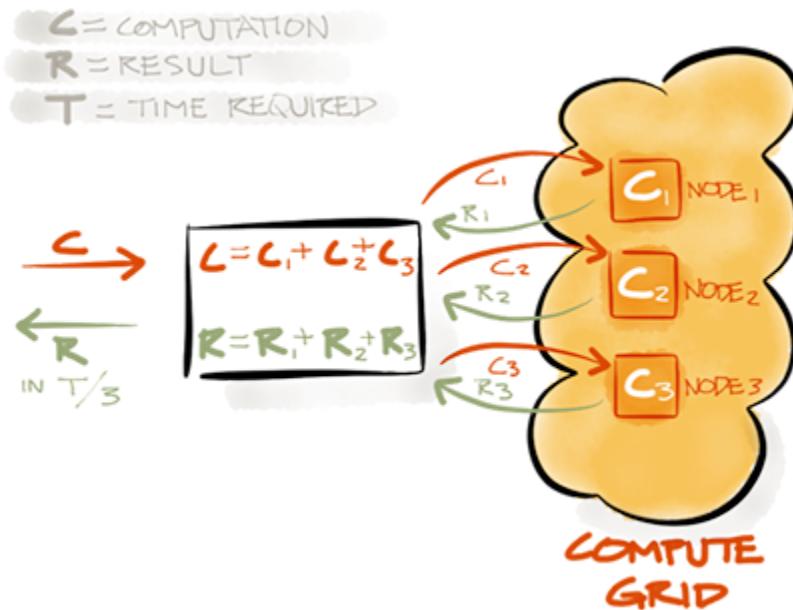
value = client.get('key')
```

# Compute Grid

# Compute Grid

Distribute your computations across cluster nodes.

Distributed computations are performed in parallel fashion to gain **high performance, low latency**, and **linear scalability**. Ignite compute grid provides a set of simple APIs that allow users distribute computations and data processing across multiple computers in the cluster. Distributed parallel processing is based on the ability to take any computation and execute it on any set of cluster nodes and return the results back.



## Features

- [Distributed Closures]
- MapReduce & ForkJoin
- Executor Service
- Collocate Compute and Data
- Load Balancing
- Fault Tolerance
- Checkpointing
- Job Scheduling

## IgniteCompute

`IgniteCompute` interface provides methods for running many types of computations over nodes in a

cluster or a cluster group. These methods can be used to execute Tasks or Closures in distributed fashion.

All jobs and closures are [Fault Tolerance](#) as long as there is at least one node standing. If a job execution is rejected due to lack of resources, a failover mechanism is provided. In case of failover, the load balancer picks the next available node to execute the job. Here is how you can get an [IgniteCompute](#) instance:

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

// Get compute instance over all nodes in the cluster.
IgniteCompute compute = ignite.compute();
```

You can also limit the scope of computations to a [Cluster Groups](#). In this case, computation will only execute on the nodes within the cluster group.

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

ClusterGroup remoteGroup = ignite.cluster().forRemotes();

// Limit computations only to remote nodes (exclude local node).
IgniteCompute compute = ignite.compute(remoteGroup);
```

## Distributed Closures

Broadcast and load-balance closure execution across cluster nodes.

---

Ignite compute grid allows to broadcast and load-balance any closure within the cluster or a cluster group, including plain Java [runnables](#) and [callables](#).

### Broadcast Methods

---

All [broadcast\(...\)](#) methods broadcast a given job to all nodes in the cluster or cluster group.

### *broadcast*

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

// Limit broadcast to remote nodes only.
IgniteCompute compute = ignite.compute(ignite.cluster().forRemotes());

// Print out hello message on remote nodes in the cluster group.
compute.broadcast(() -> System.out.println("Hello Node: " + ignite.cluster().localNode()
.id()));
```

### *async broadcast*

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

// Limit broadcast to remote nodes only and
// enable asynchronous mode.
IgniteCompute compute = ignite.compute(ignite.cluster().forRemotes()).withAsync();

// Print out hello message on remote nodes in the cluster group.
compute.broadcast(() -> System.out.println("Hello Node: " + ignite.cluster().localNode()
.id()));

ComputeTaskFuture<?> fut = compute.future();

fut.listenAsync(f -> System.out.println("Finished sending broadcast job."));
```

### *java7 broadcast*

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

// Limit broadcast to remote nodes only.
IgniteCompute compute = ignite.compute(ignite.cluster().forRemotes());

// Print out hello message on remote nodes in projection.
compute.broadcast(
    new IgniteRunnable() {
        @Override public void run() {
            // Print ID of remote node on remote node.
            System.out.println(">>> Hello Node: " + ignite.cluster().localNode().id());
        }
    });
);
```

*java7 async broadcast*

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

// Limit broadcast to remote nodes only and
// enable asynchronous mode.
IgniteCompute compute = ignite.compute(ignite.cluster().forRemotes()).withAsync();

// Print out hello message on remote nodes in the cluster group.
compute.broadcast(
    new IgniteRunnable() {
        @Override public void run() {
            // Print ID of remote node on remote node.
            System.out.println(">>> Hello Node: " + ignite.cluster().localNode().id());
        }
    }
);

ComputeTaskFuture<?> fut = compute.future();

fut.listenAsync(new IgniteInClosure<? super ComputeTaskFuture<?>>() {
    public void apply(ComputeTaskFuture<?> fut) {
        System.out.println("Finished sending broadcast job to cluster.");
    }
});
```

## Call and Run Methods

All `call(...)` and `run(...)` methods execute either individual jobs or collections of jobs on the cluster or a cluster group.

*call*

```
Collection<IgniteCallable<Integer>> calls = new ArrayList<>();

// Iterate through all words in the sentence and create callable jobs.
for (String word : "How many characters".split(" "))
    calls.add(word::length);

// Execute collection of callables on the cluster.
Collection<Integer> res = ignite.compute().call(calls);

// Add all the word lengths received from cluster nodes.
int total = res.stream().mapToInt(Integer::intValue).sum();
```

*run*

```
IgniteCompute compute = ignite.compute();

// Iterate through all words and print
// each word on a different cluster node.
for (String word : "Print words on different cluster nodes".split(" "))
    // Run on some cluster node.
    compute.run(() -> System.out.println(word));
```

*async call*

```
Collection<IgniteCallable<Integer>> calls = new ArrayList<>();

// Iterate through all words in the sentence and create callable jobs.
for (String word : "Count characters using callable".split(" "))
    calls.add(word::length);

// Enable asynchronous mode.
IgniteCompute asyncCompute = ignite.compute().withAsync();

// Asynchronously execute collection of callables on the cluster.
asyncCompute.call(calls);

asyncCompute.future().listenAsync(fut -> {
    // Total number of characters.
    int total = fut.get().stream().mapToInt(Integer::intValue).sum();

    System.out.println("Total number of characters: " + total);
});
```

*async run*

```
IgniteCompute asyncCompute = ignite.compute().withAsync();

Collection<ComputeTaskFuture<?>> futs = new ArrayList<>();

// Iterate through all words and print
// each word on a different cluster node.
for (String word : "Print words on different cluster nodes".split(" ")) {
    // Asynchronously run on some cluster node.
    asyncCompute.run(() -> System.out.println(word));

    futs.add(asyncCompute.future());
}

// Wait for completion of all futures.
futs.stream().forEach(ComputeTaskFuture::get);
```

*java7 call*

```
Collection<IgniteCallable<Integer>> calls = new ArrayList<>();

// Iterate through all words in the sentence and create callable jobs.
for (final String word : "Count characters using callable".split(" ")) {
    calls.add(new IgniteCallable<Integer>() {
        @Override public Integer call() throws Exception {
            return word.length(); // Return word length.
        }
    });
}

// Execute collection of callables on the cluster.
Collection<Integer> res = ignite.compute().call(calls);

int total = 0;

// Total number of characters.
// Looks much better in Java 8.
for (Integer i : res)
    total += i;
```

```
java7 async run
```

```
IgniteCompute asyncCompute = ignite.compute().withAsync();

Collection<ComputeTaskFuture<?>> futs = new ArrayList<>();

// Iterate through all words and print
// each word on a different cluster node.
for (String word : "Print words on different cluster nodes".split(" ")) {
    // Asynchronously run on some cluster node.
    asyncCompute.run(new IgniteRunnable() {
        @Override public void run() {
            System.out.println(word);
        }
    });

    futs.add(asyncCompute.future());
}

// Wait for completion of all futures.
for (ComputeTaskFuture<?> f : futs)
    f.get();
```

## Apply Methods

A closure is a block of code that encloses its body and any outside variables used inside of it as a function object. You can then pass such function object anywhere you can pass a variable and execute it. All apply(...) methods execute closures on the cluster.

*apply*

```
IgniteCompute compute = ignite.compute();

// Execute closure on all cluster nodes.
Collection<Integer> res = compute.apply(
    String::length,
    Arrays.asList("How many characters".split(" "))
);

// Add all the word lengths received from cluster nodes.
int total = res.stream().mapToInt(Integer::intValue).sum();
```

### *async apply*

```
// Enable asynchronous mode.  
IgniteCompute asyncCompute = ignite.compute().withAsync();  
  
// Execute closure on all cluster nodes.  
// If the number of closures is less than the number of  
// parameters, then Ignite will create as many closures  
// as there are parameters.  
Collection<Integer> res = asyncCompute.apply(  
    String::length,  
    Arrays.asList("How many characters".split(" ")))  
;  
  
asyncCompute.future().listenAsync(fut -> {  
    // Total number of characters.  
    int total = fut.get().stream().mapToInt(Integer::intValue).sum();  
  
    System.out.println("Total number of characters: " + total);  
});
```

### *java7 apply*

```
// Execute closure on all cluster nodes.  
Collection<Integer> res = ignite.compute().apply(  
    new IgniteClosure<String, Integer>() {  
        @Override public Integer apply(String word) {  
            // Return number of letters in the word.  
            return word.length();  
        }  
    },  
    Arrays.asList("Count characters using closure".split(" ")))  
;  
  
int sum = 0;  
  
// Add up individual word lengths received from remote nodes  
for (int len : res)  
    sum += len;
```

## Executor Service

[Compute Grid](#) provides a convenient API for executing computations on the cluster. However, you can also work directly with standard [ExecutorService](#) interface from JDK. Ignite provides a cluster-enabled

implementation of [ExecutorService](#) and automatically executes all the computations in load-balanced fashion within the cluster. Your computations also become fault-tolerant and are guaranteed to execute as long as there is at least one node left. You can think of it as a distributed cluster-enabled thread pool.

```
// Get cluster-enabled executor service.  
ExecutorService exec = ignite.executorService();  
  
// Iterate through all words in the sentence and create jobs.  
for (final String word : "Print words using runnable".split(" ")) {  
    // Execute runnable on some node.  
    exec.submit(new IgniteRunnable() {  
        @Override public void run() {  
            System.out.println(">>> Printing '" + word + "' on this node from grid job.");  
        }  
    });  
}
```

You can also limit the job execution with some subset of nodes from your grid:

```
// Cluster group for nodes where the attribute 'worker' is defined.  
ClusterGroup workerGrp = ignite.cluster().forAttribute("ROLE", "worker");  
  
// Get cluster-enabled executor service for the above cluster group.  
ExecutorService exec = ignite.executorService(workerGrp);
```

## MapReduce & ForkJoin

Execute MapReduce and ForkJoin tasks in memory.

[ComputeTask](#) is the Ignite abstraction for the simplified in-memory MapReduce, which is also very close to ForkJoin paradigm. Pure MapReduce was never built for performance and only works well when dealing with off-line batch oriented processing (e.g. Hadoop MapReduce). However, when computing on data that resides in-memory, real-time low latencies and high throughput usually take the highest priority. Also, simplicity of the API becomes very important as well. With that in mind, Ignite introduced the [ComputeTask](#) API, which is a light-weight MapReduce (or ForkJoin) implementation.



Use [ComputeTask](#) only when you need fine-grained control over the job-to-node mapping, or custom fail-over logic. For all other cases you should use simple closure executions on the cluster documented in [Compute Grid](#) section.

# ComputeTask

---

`ComputeTask` defines jobs to execute on the cluster, and the mappings of those jobs to nodes. It also defines how to process (reduce) the job results. All `IgniteCompute.execute(...)` methods execute the given task on the grid. User applications should implement `map(...)` and `reduce(...)` methods of `ComputeTask` interface.

Tasks are defined by implementing the 2 or 3 methods on `ComputeTask` interface

## Map Method

Method `map(...)` instantiates the jobs and maps them to worker nodes. The method receives the collection of cluster nodes on which the task is run and the task argument. The method should return a map with jobs as keys and mapped worker nodes as values. The jobs are then sent to the mapped nodes and executed there.



Refer to [Map Method](#) method. ===== Result Method Method `result(...)` is called each time a job completes on some cluster node. It receives the result returned by the completed job, as well as the list of all the job results received so far. The method should return a `ComputeJobResultPolicy` instance, indicating what to do next:

- `WAIT` - wait for all remaining jobs to complete (if any)
- `REDUCE` - immediately move to reduce step, discarding all the remaining jobs and unreceived yet results
- `FAILOVER` - failover the job to another node (see Fault Tolerance) All the received job results will be available in the `reduce(...)` method as well.

## Reduce Method

Method `reduce(...)` is called on reduce step, when all the jobs have completed (or REDUCE result policy was returned from the `result(...)` method). The method receives a list with all the completed results and should return a final result of the computation.

## Compute Task Adapters

---

It is not necessary to implement all 3 methods of the `ComputeTask` API each time you need to define a computation. There is a number of helper classes that let you describe only a particular piece of your logic, leaving out all the rest to Ignite to handle automatically.

## ComputeTaskAdapter

`ComputeTaskAdapter` defines a default implementation of the `result(...)` method which returns `FAILOVER`

policy if a job threw an exception and `WAIT` policy otherwise, thus waiting for all jobs to finish with a result.

## ComputeTaskSplitAdapter

`ComputeTaskSplitAdapter` extends `ComputeTaskAdapter` and adds capability to automatically assign jobs to nodes. It hides the `map(...)` method and adds a new `split(...)` method in which user only needs to provide a collection of the jobs to be executed (the mapping of those jobs to nodes will be handled automatically by the adapter in a load-balanced fashion).

This adapter is especially useful in homogeneous environments where all nodes are equally suitable for executing jobs and the mapping step can be done implicitly.

## ComputeJob

---

All jobs that are spawned by a task are implementations of the `ComputeJob` interface. The `execute()` method of this interface defines the job logic and should return a job result. The `cancel()` method defines the logic in case if the job is discarded (for example, in case when task decides to reduce immediately or to cancel).

## ComputeJobAdapter

Convenience adapter which provides a no-op implementation of the `cancel()` method.

## Example

---

Here is an example of `ComputeTask` and `ComputeJob` implementations.

### *ComputeTaskSplitAdapter*

```
IgniteCompute compute = ignite.compute();

// Execute task on the clustr and wait for its completion.
int cnt = grid.compute().execute(CharacterCountTask.class, "Hello Grid Enabled World!");

System.out.println(">>> Total number of characters in the phrase is '" + cnt + "'.");

/**
 * Task to count non-white-space characters in a phrase.
 */
private static class CharacterCountTask extends ComputeTaskSplitAdapter<String, Integer>
{
```

```

// 1. Splits the received string into to words
// 2. Creates a child job for each word
// 3. Sends created jobs to other nodes for processing.
@Override
public List<ClusterNode> split(List<ClusterNode> subgrid, String arg) {
    String[] words = arg.split(" ");

    List<ComputeJob> jobs = new ArrayList<>(words.length);

    for (final String word : arg.split(" ")) {
        jobs.add(new ComputeJobAdapter() {
            @Override public Object execute() {
                System.out.println(">>> Printing '" + word + "' on from compute job.");
                // Return number of letters in the word.
                return word.length();
            }
        });
    }

    return jobs;
}

@Override
public Integer reduce(List<ComputeJobResult> results) {
    int sum = 0;

    for (ComputeJobResult res : results)
        sum += res.<Integer>getData();

    return sum;
}

```

### *ComputeTaskAdapter*

```

IgniteCompute compute = ignite.compute();

// Execute task on the clustr and wait for its completion.
int cnt = grid.compute().execute(CharacterCountTask.class, "Hello Grid Enabled World!");

System.out.println(">>> Total number of characters in the phrase is '" + cnt + "'.");

/**
 * Task to count non-white-space characters in a phrase.

```

```

*/
private static class CharacterCountTask extends ComputeTaskAdapter<String, Integer> {
    // 1. Splits the received string into to words
    // 2. Creates a child job for each word
    // 3. Sends created jobs to other nodes for processing.
    @Override
    public Map<? extends ComputeJob, ClusterNode> map(List<ClusterNode> subgrid, String
arg) {
        String[] words = arg.split(" ");
        Map<ComputeJob, ClusterNode> map = new HashMap<>(words.length);
        Iterator<ClusterNode> it = subgrid.iterator();
        for (final String word : arg.split(" ")) {
            // If we used all nodes, restart the iterator.
            if (!it.hasNext())
                it = subgrid.iterator();
            ClusterNode node = it.next();
            map.put(new ComputeJobAdapter() {
                @Override public Object execute() {
                    System.out.println(">>> Printing '" + word + "' on this node from
grid job.");
                    // Return number of letters in the word.
                    return word.length();
                }
            }, node);
        }
        return map;
    }

    @Override
    public Integer reduce(List<ComputeJobResult> results) {
        int sum = 0;
        for (ComputeJobResult res : results)
            sum += res.<Integer>getData();
        return sum;
    }
}

```

## Distributed Task Session

Distributed task session is created for every task execution. It is defined by [ComputeTaskSession](#) interface. Task session is visible to the task and all the jobs spawned by it, so attributes set on a task or on a job can be accessed on other jobs. Task session also allows to receive notifications when attributes are set or wait for an attribute to be set.

The sequence in which session attributes are set is consistent across the task and all job siblings within it. There will never be a case when one job sees attribute A before attribute B, and another job sees attribute B before A.

In the example below, we have all jobs synchronize on STEP1 before moving on to STEP2.



**@ComputeTaskSessionFullSupport annotation.** Note that distributed task session attributes are disabled by default for performance reasons. To enable them attach [@ComputeTaskSessionFullSupport](#) annotation to the task class.

```
IgniteCompute compute = ignite.compute();

compute.execute(new TaskSessionAttributesTask(), null);

/**
 * Task demonstrating distributed task session attributes.
 *
 * Note that task session attributes are enabled only if
 *
 * @ComputeTaskSessionFullSupport annotation is attached.
 */

@SuppressWarnings
private static class TaskSessionAttributesTask extends ComputeTaskSplitAdapter<Object,
Object> {
    @Override
    protected Collection<? extends GridJob> split(int gridSize, Object arg) {
        Collection<ComputeJob> jobs = new LinkedList<>();

        // Generate jobs by number of nodes in the grid.
        for (int i = 0; i < gridSize; i++) {
            jobs.add(new ComputeJobAdapter(arg) {
                // Auto-injected task session.
                @TaskSessionResource
                private ComputeTaskSession ses;

                // Auto-injected job context.
            });
        }
    }
}
```

```

@JobContextResource
private ComputeJobContext jobCtx;

@Override
public Object execute() {
    // Perform STEP1.
    ...

    // Tell other jobs that STEP1 is complete.
    ses.setAttribute(jobCtx.get jobId(), "STEP1");

    // Wait for other jobs to complete STEP1.
    for (ComputeJobSibling sibling : ses.getJobSiblings())
        ses.waitForAttribute(sibling.get jobId(), "STEP1", 0);

    // Move on to STEP2.
    ...
}

@Override
public Object reduce(List<ComputeJobResult> results) {
    // No-op.
    return null;
}
}

```

## Per-Node Shared State

Share state between jobs or services on a cluster node.

Often it is useful to share a state between different compute jobs or different deployed services. For this purpose Ignite provides a shared concurrent **node-local-map** available on each node.

```

IgniteCluster cluster = ignite.cluster();

ConcurrentMap<String, Integer> nodeLocalMap = cluster.nodeLocalMap();

```

Node-local values are similar to thread locals in a way that these values are not distributed and kept only on the local node. Node-local data can be used by compute jobs to share the state between executions. It can also be used by deployed services as well.

As an example, let's create a job which increments a node-local counter every time it executes on some node. This way, the node-local counter on each node will tell us how many times a job had executed on that cluster node.

```
private IgniteCallable<Long> job = new IgniteCallable<Long>() {
    @IgniteInstanceResource
    private Ignite ignite;

    @Override
    public Long call() {
        // Get a reference to node local.
        ConcurrentMap<String, AtomicLong> nodeLocalMap = ignite.cluster().nodeLocalMap();

        AtomicLong cntr = nodeLocalMap.get("counter");

        if (cntr == null) {
            AtomicLong old = nodeLocalMap.putIfAbsent("counter", cntr = new AtomicLong());

            if (old != null)
                cntr = old;
        }

        return cntr.incrementAndGet();
    }
}
```

Now let's execute this job 2 times on the same node and make sure that the value of the counter is 2.

```
ClusterGroup random = ignite.cluster().forRandom();

IgniteCompute compute = ignite.compute(random);

// The first time the counter on the picked node will be initialized to 1.
Long res = compute.call(job);

assert res == 1;

// Now the counter will be incremented and will have value 2.
res = compute.call(job);

assert res == 2;
```

# Collocate Compute and Data

Collocate your computations with the data.

Collocation of computations with data allow for minimizing data serialization within network and can significantly improve performance and scalability of your application. Whenever possible, you should always make best effort to colocate your computations with the cluster nodes caching the data that needs to be processed.

## Affinity Call and Run Methods

`affinityCall(...)` and `affinityRun(...)` methods co-locate jobs with nodes on which data is cached. In other words, given a cache name and affinity key these methods try to locate the node on which the key resides on Ignite the specified Ignite cache, and then execute the job there.

`affinityRun`

```
IgniteCache<Integer, String> cache = ignite.cache(CACHE_NAME);

IgniteCompute compute = ignite.compute();

for (int key = 0; key < KEY_CNT; key++) {
    // This closure will execute on the remote node where
    // data with the 'key' is located.
    compute.affinityRun(CACHE_NAME, key, () -> {
        // Peek is a local memory lookup.
        System.out.println("Co-located [key= " + key + ", value= " + cache.localPeek(key)
+ "]");
    });
}
```

### *async affinityRun*

```
IgniteCache<Integer, String> cache = ignite.cache(CACHE_NAME);

IgniteCompute asyncCompute = ignite.compute().withAsync();

List<IgniteFuture<?>> futs = new ArrayList<>();

for (int key = 0; key < KEY_CNT; key++) {
    // This closure will execute on the remote node where
    // data with the 'key' is located.
    asyncCompute.affinityRun(CACHE_NAME, key, () -> {
        // Peek is a local memory lookup.
        System.out.println("Co-located [key= " + key + ", value= " + cache.peek(key) + "]");
    });
}

futs.add(asyncCompute.future());
}

// Wait for all futures to complete.
futs.stream().forEach(IgniteFuture::get);
```

### *java7 affinityRun*

```
final IgniteCache<Integer, String> cache = ignite.cache(CACHE_NAME);

IgniteCompute compute = ignite.compute();

for (int i = 0; i < KEY_CNT; i++) {
    final int key = i;

    // This closure will execute on the remote node where
    // data with the 'key' is located.
    compute.affinityRun(CACHE_NAME, key, new IgniteRunnable() {
        @Override public void run() {
            // Peek is a local memory lookup.
            System.out.println("Co-located [key= " + key + ", value= " + cache.peek(key)
+ "]");
        }
    });
}
```

## Fault Tolerance

Automatically fail-over jobs to other nodes in case of a crash.

Ignite supports automatic job failover. In case of a node crash, jobs are automatically transferred to other available nodes for re-execution. However, in Ignite you can also treat any job result as a failure as well. The worker node can still be alive, but it may be running low on CPU, I/O, disk space, etc. There are many conditions that may result in a failure within your application and you can trigger a failover. Moreover, you have the ability to choose to which node a job should be failed over to, as it could be different for different applications or different computations within the same application.

The `FailoverSpi` is responsible for handling the selection of a new node for the execution of a failed job. `FailoverSpi` inspects the failed job and the list of all available grid nodes on which the job execution can be retried. It ensures that the job is not re-mapped to the same node it had failed on. Failover is triggered when the method `ComputeTask.result(...)` returns the `ComputeJobResultPolicy.FAILOVER` policy. Ignite comes with a number of built-in customizable Failover SPI implementations.

## At Least Once Guarantee

As long as there is at least one node standing, no job will ever be lost.

By default, Ignite will failover all jobs from stopped or crashed nodes automatically. For custom failover behavior, you should implement `ComputeTask.result()` method. The example below triggers failover whenever a job throws any `IgniteException` (or its subclasses):

```
public class MyComputeTask extends ComputeTaskSplitAdapter<String, String> {  
    ...  
  
    @Override  
    public ComputeJobResultPolicy result(ComputeJobResult res, List<ComputeJobResult>  
        rcvd) {  
        IgniteException err = res.getException();  
  
        if (err != null)  
            return ComputeJobResultPolicy.FAILOVER;  
  
        // If there is no exception, wait for all job results.  
        return ComputeJobResultPolicy.WAIT;  
    }  
  
    ...  
}
```

## Closure Failover

Closure failover is by default governed by [ComputeTaskAdapter](#), which is triggered if a remote node either crashes or rejects closure execution. This default behavior may be overridden by using [IgniteCompute.withNoFailover\(\)](#) method, which creates an instance of [IgniteCompute](#) with a **no-failover flag** set on it. Here is an example:

```
IgniteCompute compute = ignite.compute().withNoFailover();

compute.apply(() -> {
    // Do something
    ...
}, "Some argument");
```

## AlwaysFailOverSpi

[AlwaysFailoverSpi](#) always reroutes a failed job to another node. Note, that at first an attempt will be made to reroute the failed job to a node that the task was not executed on. If no such nodes are available, then an attempt will be made to reroute the failed job to the nodes that may be running other jobs from the same task. If none of the above attempts succeeded, then the job will not be failed over and null will be returned.

The following configuration parameters can be used to configure [AlwaysFailoverSpi](#).

Setter Method	Description	Default
<a href="#">setMaximumFailoverAttempts(int)</a>	Sets the maximum number of attempts to fail-over a failed job to other nodes.	5

```
<bean id="grid.custom.cfg" class="org.apache.ignite.IgniteConfiguration" singleton="true">
    ...
    <bean class="org.apache.ignite.spi.failover.always.AlwaysFailoverSpi">
        <property name="maximumFailoverAttempts" value="5"/>
    </bean>
    ...
</bean>
```

```

AlwaysFailoverSpi failSpi = new AlwaysFailoverSpi();

IgniteConfiguration cfg = new IgniteConfiguration();

// Override maximum failover attempts.
failSpi.setMaximumFailoverAttempts(5);

// Override the default failover SPI.
cfg.setFailoverSpi(failSpi);

// Start Ignite node.
Ignition.start(cfg);

```

## Load Balancing

Load balancing component balances job distribution among cluster nodes. In Ignite load balancing is achieved via [LoadBalancingSpi](#) which controls load on all nodes and makes sure that every node in the cluster is equally loaded. In homogeneous environments with homogeneous tasks load balancing is achieved by random or round-robin policies. However, in many other use cases, especially under uneven load, more complex adaptive load-balancing policies may be needed.



**Data Affinity.** Note that load balancing is triggered whenever your jobs are not collocated with data or have no real preference on which node to execute. If [Collocate Compute and Data](#) is used, then data affinity takes priority over load balancing.

### Round-Robin Load Balancing

[RoundRobinLoadBalancingSpi](#) iterates through nodes in round-robin fashion and picks the next sequential node. Two modes of operation are supported: per-task and global.

#### Per-Task Mode

When configured in per-task mode, implementation will pick a random node at the beginning of every task execution and then sequentially iterate through all nodes in topology starting from the picked node. This is the default configuration. For cases when split size is equal to the number of nodes, this mode guarantees that all nodes will participate in the split.

#### Global Mode

When configured in global mode, a single sequential queue of nodes is maintained for all tasks and the

next node in the queue is picked every time. In this mode (unlike in per-task mode) it is possible that even if split size may be equal to the number of nodes, some jobs within the same task will be assigned to the same node whenever multiple tasks are executing concurrently.

```
<bean id="grid.custom.cfg" class="org.apache.ignite.IgniteConfiguration" singleton="true">
    ...
    <property name="loadBalancingSpi">
        <bean class=
"org.apache.ignite.spi.loadbalancing.roundrobin.RoundRobinLoadBalancingSpi">
            <!-- Set to per-task round-robin mode (this is default behavior). -->
            <property name="perTask" value="true"/>
        </bean>
    </property>
    ...
</bean>
```

```
RoundRobinLoadBalancingSpi = new RoundRobinLoadBalancingSpi();

// Configure SPI to use per-task mode (this is default behavior).
spi.setPerTask(true);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default load balancing SPI.
cfg.setLoadBalancingSpi(spi);

// Start Ignite node.
Ignition.start(cfg);
```

## Random and Weighted Load Balancing

`WeightedRandomLoadBalancingSpi` picks a random node for job execution by default. You can also optionally assign weights to nodes, so nodes with larger weights will end up getting proportionally more jobs routed to them. By default all nodes get equal weight of 10.

```

<bean id="grid.custom.cfg" class="org.apache.ignite.IgniteConfiguration" singleton="true"
">
    ...
    <property name="loadBalancingSpi">
        <bean class=
"org.apache.ignite.spi.loadbalancing.weightedrandom.WeightedRandomLoadBalancingSpi">
            <property name="useWeights" value="true"/>
            <property name="nodeWeight" value="10"/>
        </bean>
    </property>
    ...
</bean>
```

```

WeightedRandomLoadBalancingSpi = new WeightedRandomLoadBalancingSpi();

// Configure SPI to used weighted random load balancing.
spi.setUseWeights(true);

// Set weight for the local node.
spi.setWeight(10);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default load balancing SPI.
cfg.setLoadBalancingSpi(spi);

// Start Ignite node.
Ignition.start(cfg);
```

## Checkpointing

Checkpointing provides an ability to save an intermediate job state. It can be useful when long running jobs need to store some intermediate state to protect from node failures. Then on restart of a failed node, a job would load the saved checkpoint and continue from where it left off. The only requirement for job checkpoint state is to implement [java.io.Serializable](#) interface.

Checkpoints are available through the following methods on [GridTaskSession](#) interface:

- [ComputeTaskSession.loadCheckpoint\(String\)](#)
- [ComputeTaskSession.removeCheckpoint\(String\)](#)
- [ComputeTaskSession.saveCheckpoint\(String, Object\)](#)



`@ComputeTaskSessionFullSupport` annotation. Note that checkpointing is disabled by default for performance reasons. To enable it attach `@ComputeTaskSessionFullSupport` annotation to the task or closure class.

## Master Node Failure Protection

---

One important use case for checkpoint that is not readily apparent is to guard against failure of the "master" node - the node that started the original execution. When master node fails, Ignite doesn't have where to send the results of job execution to, and thus the result will be discarded.

To failover this scenario one can store the final result of the job execution as a checkpoint and have the logic re-run the entire task in case of a "master" node failure. In such case the task re-run will be much faster since all the jobs' can start from the saved checkpoints.

## Setting Checkpoints

---

Every compute job can periodically **checkpoint** itself by calling `ComputeTaskSession.saveCheckpoint(...)` method.

If job did save a checkpoint, then upon beginning of its execution, it should check if the checkpoint is available and start executing from the last saved checkpoint.

```

IgniteCompute compute = ignite.compute();

compute.run(new CheckpointsRunnable());

/** 

* Note that this class is annotated with @ComputeTaskSessionFullSupport

* annotation to enable checkpointing.

*/

@ComputeTaskSessionFullSupport
private static class CheckpointsRunnable implements IgniteRunnable() {
    // Task session (injected on closure instantiation).
    @TaskSessionResource
    private ComputeTaskSession ses;

    @Override
    public Object applyx(Object arg) throws GridException {
        // Try to retrieve step1 result.
        Object res1 = ses.loadCheckpoint("STEP1");

        if (res1 == null) {
            res1 = computeStep1(arg); // Do some computation.

            // Save step1 result.
            ses.saveCheckpoint("STEP1", res1);
        }

        // Try to retrieve step2 result.
        Object res2 = ses.loadCheckpoint("STEP2");

        if (res2 == null) {
            res2 = computeStep2(res1); // Do some computation.

            // Save step2 result.
            ses.saveCheckpoint("STEP2", res2);
        }

        ...
    }
}

```

## CheckpointSpi

In Ignite, checkpointing functionality is provided by [CheckpointSpi](#) which has the following out-of-the-box implementations:

Class	Description
Default	<a href="#">File System Checkpoint Configuration</a>
This implementation uses a shared file system to store checkpoints.	Yes
<a href="#">Cache Checkpoint Configuration</a>	This implementation uses a cache to store checkpoints.
<a href="#">Database Checkpoint Configuration</a>	This implementation uses a database to store checkpoints.
This implementation uses Amazon S3 to store checkpoints.	<a href="#">Amazon S3 Checkpoint Configuration</a>

[CheckpointSpi](#) is provided in [IgniteConfiguration](#) and passed into Ignition class at startup.

## File System Checkpoint Configuration

The following configuration parameters can be used to configure [SharedFsCheckpointSpi](#):

Setter Method	Description	Default
<code>setDirectoryPaths(Collection)</code>	Sets directory paths to the shared folders where checkpoints are stored. The path can either be absolute or relative to the path specified in <a href="#">IGNITE_HOME</a> environment or system variable.	<code>IGNITE_HOME/work/cp/sharedfs</code>

```

<bean class="org.apache.ignite.IgniteConfiguration" singleton="true">
    ...
    <property name="checkpointSpi">
        <bean class="org.apache.ignite.spi.checkpoint.sharedfs.SharedFsCheckpointSpi">
            <!-- Change to shared directory path in your environment. -->
            <property name="directoryPaths">
                <list>
                    <value>/my/directory/path</value>
                    <value>/other/directory/path</value>
                </list>
            </property>
        </bean>
    </property>
    ...
</bean>
```

```

IgniteConfiguration cfg = new IgniteConfiguration();

SharedFsCheckpointSpi checkpointSpi = new SharedFsCheckpointSpi();

// List of checkpoint directories where all files are stored.
Collection<String> dirPaths = new ArrayList<String>();

dirPaths.add("/my/directory/path");
dirPaths.add("/other/directory/path");

// Override default directory path.
checkpointSpi.setDirectoryPaths(dirPaths);

// Override default checkpoint SPI.
cfg.setCheckpointSpi(checkpointSpi);

// Starts Ignite node.
Ignition.start(cfg);
```

## Cache Checkpoint Configuration

[CacheCheckpointSpi](#) is a cache-based implementation for checkpoint SPI. Checkpoint data will be stored in the Ignite data grid in a pre-configured cache.

The following configuration parameters can be used to configure [CacheCheckpointSpi](#):

Setter Method	Description	Default
<code>setCacheName(String)</code>	Sets cache name to use for storing checkpoints.	<code>checkpoints</code>

## Database Checkpoint Configuration

---

`JdbcCheckpointSpi` uses database to store checkpoints. All checkpoints are stored in the database table and are available from all nodes in the grid. Note that every node must have access to the database. A job state can be saved on one node and loaded on another (e.g., if a job gets preempted on a different node after node failure).

The following configuration parameters can be used to configure `JdbcCheckpointSpi` (all are optional):

Setter Method	Description	Default
<code>setDataSource(DataSource)</code>	Sets DataSource to use for database access.	No value
<code>setCheckpointTableName(String)</code>	Sets checkpoint table name.	<code>CHECKPOINTS</code>
<code>setKeyFieldName(String)</code>	Sets checkpoint key field name.	<code>NAME</code>
<code>setKeyFieldType(String)</code>	Sets checkpoint key field type. The field should have corresponding SQL string type ( <code>VARCHAR</code> , for example).	<code>VARCHAR(256)</code>
<code>setValueFieldName(String)</code>	Sets checkpoint value field name.	<code>VALUE</code>
<code>setValueFieldType(String)</code>	Sets checkpoint value field type. Note, that the field should have corresponding SQL BLOB type. The default value is BLOB, won't work for all databases. For example, if using HSQL DB, then the type should be <code>longvarbinary</code> .	<code>BLOB</code>
<code>setExpireDateFieldName(String)</code>	Sets checkpoint expiration date field name.	<code>EXPIRE_DATE</code>
<code>setExpireDateFieldType(String)</code>	Sets checkpoint expiration date field type. The field should have corresponding SQL <code>DATETIME</code> type.	<code>DATETIME</code>
<code>setNumberOfRetries(int)</code>	Sets number of retries in case of any database errors.	2

Setter Method	Description	Default
<code>setUser(String)</code>	Sets checkpoint database user name. Note that authentication will be performed only if both, user and password are set.	No value
<code>setPassword(String)</code>	Sets checkpoint database password.	No value

## Apache DBCP

[Apache DBCP](#) project for more information.

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration" singleton="true">
    ...
    <property name="checkpointSpi">
        <bean class="org.apache.ignite.spi.checkpoint.database.JdbcCheckpointSpi">
            <property name="dataSource">
                <ref bean="anyPooledDataSourceBean"/>
            </property>
            <property name="checkpointTableName" value="CHECKPOINTS"/>
            <property name="user" value="test"/>
            <property name="password" value="test"/>
        </bean>
    </property>
    ...
</bean>
```

```
JdbcCheckpointSpi checkpointSpi = new JdbcCheckpointSpi();

javax.sql.DataSource ds = ... // Set datasource.

// Set database checkpoint SPI parameters.
checkpointSpi.setDataSource(ds);
checkpointSpi.setUser("test");
checkpointSpi.setPassword("test");

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default checkpoint SPI.
cfg.setCheckpointSpi(checkpointSpi);

// Start Ignite node.
Ignition.start(cfg);
```

## Amazon S3 Checkpoint Configuration

`S3CheckpointSpi` uses Amazon S3 storage to store checkpoints. For information about Amazon S3 visit <http://aws.amazon.com/>.

The following configuration parameters can be used to configure `S3CheckpointSpi`:

Setter Method	Description	Default
<code>setAwsCredentials(AWSCredentials)</code>	Sets AWS credentials to use for storing checkpoints.	No value (must be provided)
<code>setClientConfiguration(Client)</code>	Sets AWS client configuration.	No value
<code>setBucketNameSuffix(String)</code>	Sets bucket name suffix.	default-bucket

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration" singleton="true">
    ...
    <property name="checkpointSpi">
        <bean class="org.apache.ignite.spi.checkpoint.s3.S3CheckpointSpi">
            <property name="awsCredentials">
                <bean class="com.amazonaws.auth.BasicAWSCredentials">
                    <constructor-arg value="YOUR_ACCESS_KEY_ID" />
                    <constructor-arg value="YOUR_SECRET_ACCESS_KEY" />
                </bean>
            </property>
        </bean>
    </property>
    ...
</bean>
```

```
IgniteConfiguration cfg = new IgniteConfiguration();

S3CheckpointSpi spi = new S3CheckpointSpi();

AWS Credentials cred = new BasicAWS Credentials(YOUR_ACCESS_KEY_ID, YOUR_SECRET_ACCESS_KEY);

spi.setAwsCredentials(cred);

spi.setBucketNameSuffix("checkpoints");

// Override default checkpoint SPI.
cfg.setCheckpointSpi(cpSpi);

// Start Ignite node.
Ignition.start(cfg);
```

## Job Scheduling

Properly order your jobs for execution.

---

In Ignite, jobs are mapped to cluster nodes during initial task split or closure execution on the client side. However, once jobs arrive to the designated nodes, they need to be ordered for execution. By default, jobs are submitted to a thread pool and are executed in random order. However, if you need to have a fine-grained control over job ordering, you can enable [CollisionSpi](#).

### FIFO Ordering

---

[FifoQueueCollisionSpi](#) allows a certain number of jobs in first-in first-out order to proceed without interruptions. All other jobs will be put on a waiting list until their turn.

Number of parallel jobs is controlled by [parallelJobsNumber](#) configuration parameter. Default is number of cores times 2.

### One at a Time

Note that by setting [parallelJobsNumber](#) to 1, you can guarantee that all jobs will be executed one-at-a-time, and no two jobs will be executed concurrently.

```

<bean class="org.apache.ignite.IgniteConfiguration" singleton="true">
    ...
    <property name="collisionSpi">
        <bean class="org.apache.ignite.spi.collision.fifoqueue.FifoQueueCollisionSpi">
            <!-- Execute one job at a time. -->
            <property name="parallelJobsNumber" value="1"/>
        </bean>
    </property>
    ...
</bean>

```

```

FifoQueueCollisionSpi colSpi = new FifoQueueCollisionSpi();

// Execute jobs sequentially, one at a time,
// by setting parallel job number to 1.
colSpi.setParallelJobsNumber(1);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default collision SPI.
cfg.setCollisionSpi(colSpi);

// Start Ignite node.
Ignition.start(cfg);

```

## Priority Ordering

[PriorityQueueCollisionSpi](#) allows to assign priorities to individual jobs, so jobs with higher priority will be executed ahead of lower priority jobs.

#Task Priorities Task priorities are set in the [task session](/docs/compute-tasks#distributed-task-session) via [grid.task.priority](#) attribute. If no priority has been assigned to a task, then default priority of 0 is used.

Below is an example showing how task priority can be set.

```

public class MyUrgentTask extends ComputeTaskSplitAdapter<Object, Object> {
    // Auto-injected task session.
    @TaskSessionResource
    private GridTaskSession taskSes = null;

    @Override
    protected Collection<ComputeJob> split(int gridSize, Object arg) {
        ...
        // Set high task priority.
        taskSes.setAttribute("grid.task.priority", 10);

        List<ComputeJob> jobs = new ArrayList<>(gridSize);

        for (int i = 1; i <= gridSize; i++) {
            jobs.add(new GridJobAdapter() {
                ...
            });
        }
        ...

        // These jobs will be executed with higher priority.
        return jobs;
    }
}

```

Just like with [FIFO Ordering](#), number of parallel jobs is controlled by `parallelJobsNumber` configuration parameter.

## Configuration

```

<bean class="org.apache.ignite.IgniteConfiguration" singleton="true">
    ...
    <property name="collisionSpi">
        <bean class=
"org.apache.ignite.spi.collision.priorityqueue.PriorityQueueCollisionSpi">
            <!--
                Change the parallel job number if needed.
                Default is number of cores times 2.
            -->
            <property name="parallelJobsNumber" value="5"/>
        </bean>
    </property>
    ...
</bean>

```

```

PriorityQueueCollisionSpi colSpi = new PriorityQueueCollisionSpi();

// Change the parallel job number if needed.
// Default is number of cores times 2.
colSpi.setParallelJobsNumber(5);

IgniteConfiguration cfg = new IgniteConfiguration();

// Override default collision SPI.
cfg.setCollisionSpi(colSpi);

// Start Ignite node.
Ignition.start(cfg);

```

## Task Deployment

In addition to peer class loading Ignite has a deployment mechanism which is in charge of deploying tasks and classes from different sources in runtime.

---

### DeploymentSpi

---

Deployment functionality is provided via [DeploymentSpi](#) interface.

Class loaders that are in charge of loading task classes (and other classes) can be deployed directly by calling [register\(ClassLoader, Class\)](#) method or by SPI itself, for example by asynchronously scanning some folder for new tasks. When method [findResource\(String\)](#) is called by the system, SPI must return a class loader associated with given class. Every time a class loader gets (re)deployed or released, callbacks [DeploymentListener.onUnregistered\(ClassLoader\)](#) must be called by SPI.

If peer class loading is enabled, then it is usually enough to deploy class loader only on one grid node. Once a task starts executing on the grid, all other nodes will automatically load all task classes from the node that initiated the execution. Hot redeployment is also supported with peer class loading. Every time a task changes and gets redeployed on a node, all other nodes will detect it and will redeploy this task as well. Note that peer class loading comes into effect only if a task was not locally deployed, otherwise, preference will always be given to local deployment.

Ignite provides these following [DeploymentSpi](#) implementations out of the box:

- UriDeploymentSpi
- LocalDeploymentSpi



SPI methods should never be used directly. SPIs provide internal view on the subsystem and is used internally by Ignite. In rare use cases when access to a specific implementation of this SPI is required - an instance of this SPI can be obtained via Ignite.configuration() method to check its configuration properties or call other non-SPI methods. Note again that calling methods from this interface on the obtained instance can lead to undefined behavior and explicitly not supported.

## UriDeploymentSpi

---

This is the implementation of `DeploymentSpi` which can deploy tasks from different sources like file system folders, email and HTTP. There are different ways to deploy tasks in grid and every deploy method depends on selected source protocol. This SPI is configured to work with a list of URI's. Every URI contains all the data about protocol/transport plus configuration parameters like credentials, scan frequency, and others.

When SPI establishes a connection with an URI, it downloads deployable units to the temporary directory in order to prevent it from any changes while scanning. Use method `setTemporaryDirectoryPath(String)` to set custom temporary folder for downloaded deployment units. SPI will create folder under the path with name identical to local node ID.

SPI tracks all changes of every given URI. This means that if any file is changed or deleted, SPI will redeploy or delete corresponding tasks. Note that the very first apply to `findResource(String)` is blocked until SPI finishes scanning all URI's at least once.

There are several deployable unit types supported:

- GAR file.
- Local disk folder with structure of unpacked GAR file.
- Local disk folder containing only compiled Java classes.

### GAR file

GAR file is a deployable unit. GAR file is based on ZLIB compression format like simple JAR file and its structure is similar to WAR archive. GAR file has '.gar' extension.

GAR file structure (file or directory ending with '.gar'):

```
META-INF/
|
|- ignite.xml
|- ...
lib/
|
|-some-lib.jar
|- ...
xyz.class
...
...
```

- **META-INF/** entry may contain `ignite.xml` file which is a task descriptor file. The purpose of task descriptor XML file is to specify all tasks to be deployed. This file is a regular Spring XML definition file. **META-INF/** may also contain any other files specific for JAR format.
- **lib/** entry contains all library dependencies.
- Compiled Java classes must be placed in the root of a GAR file.

GAR file may be deployed without descriptor file. If there is no descriptor file, SPI will scan all classes in archive and instantiate those that implement `ComputeTask` interface. In that case, all grid task classes must have a public no-argument constructor. Use `ComputeTaskAdapter` adapter for convenience when creating grid tasks.

By default, all downloaded GAR files that have digital signature in **META-INF** folder will be verified and deployed only if signature is valid.

## Code Example

The following examples demonstrate how the deployment SPI can be used. Different protocols can be used together as well.

## *File Protocol*

```
// The example expects that you have a GAR file in  
// 'home/username/ignite/work/my_deployment/file' folder  
// which contains 'myproject.HelloWorldTask` class.  
  
IgniteConfiguration cfg = new IgniteConfiguration();  
  
DeploymentSpi deploymentSpi = new UriDeploymentSpi();  
  
deploymentSpi.setUriList(Arrays.asList("file:///home/username/ignite/work/my_deployment/  
file"));  
  
cfg.setDeploymentSpi(deploymentSpi);  
  
try(Ignite ignite = Ignition.start(cfg)) {  
    ignite.compute().execute("myproject.HelloWorldTask", "my args");  
}
```

## *Http Protocol*

```
// The example expects that you have a HTMP under  
// 'www.mysite.com:110/ignite/deployment' page which contains a link  
// on GAR file which contains 'myproject.HelloWorldTask` class.  
  
IgniteConfiguration cfg = new IgniteConfiguration();  
  
DeploymentSpi deploymentSpi = new UriDeploymentSpi();  
  
deploymentSpi.setUriList(Arrays.asList("http://username:password;freq=1000@www.mysite.co  
m:110/ignite/deployment"));  
  
cfg.setDeploymentSpi(deploymentSpi);  
  
try(Ignite ignite = Ignition.start(cfg)) {  
    ignite.compute().execute("myproject.HelloWorldTask", "my args");  
}
```

## XML Configuration

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="deploymentSpi">
        <bean class="org.apache.ignite.grid.spi.deployment.uri.UriDeploymentSpi">
            <property name="temporaryDirectoryPath" value="c:/tmp/grid"/>
            <property name="uriList">
                <list>
                    <value>http://www.site.com/tasks</value>
                    <value>file://freq=2000@localhost/c:/Program files/gg-deployment</value>
                </list>
            </property>
        </bean>
    </property>
</bean>
```

## Configuration

Property	Description	Optional	Default
<code>uriList</code>	List of URIs that should be scanned by SPI for the new tasks.	Yes	<code>file://\${IGNITE_HOME}/work/deployment/file</code> (note that <code>IGNITE_HOME</code> must be set if you're using the default folder).
<code>scanners</code>	Array of <code>UriDeploymentScanner</code> implementations which will be used to deploy resources.	Yes	<code>UriDeploymentFileScanner</code> and <code>UriDeploymentHttpScanner</code> .
<code>temporaryDirectoryPath</code>	Temporary directory path where scanned GAR files and directories are copied to.	Yes	<code>java.io.tmpdir</code> system property value.
<code>encodeUri</code>	Flag to control encoding of the <code>path</code> portion of URI.	Yes	<code>true</code>

## Protocols

Following protocols are supported in SPI out of the box:

- `file://` - File protocol
- `http://` - HTTP protocol

- https:// - Secure HTTP protocol



**Custom Protocols.** You can add support for additional protocols if needed. To do this implement `UriDeploymentScanner` interface and plug your implementation into the SPI via `setScanners(UriDeploymentScanner... scanners)` method. In addition to SPI configuration parameters, all necessary configuration parameters for selected URI should be defined in URI. Different protocols have different configuration parameters described below. Parameters are separated by ';' character.

## #File

For this protocol SPI will scan folder specified by URI on file system and download any GAR files or directories that end with .gar from source directory defined in URI. For file system URI must have scheme equal to file.

Following parameters are supported:

Parameter	Description	Optional	Default
<code>freq</code>	Scanning frequency in milliseconds.	Yes	5000 ms

## #File Uri Example

The following example will scan `c:/Program files/ignite/deployment` folder on local box every '2000' milliseconds. Note that since path has spaces, `setEncodeUri(boolean)` parameter must be set to true (which is default behavior).

```
file://freq=2000@localhost/c:/Program files/ignite/deployment
```

## #HTTP/HTTPS

URI deployment scanner tries to read the DOM of the HTML file it points to and parses out href attributes of all <a>-tags - this becomes the URL collection (to GAR files) to deploy: each 'A' link should be an URL to a GAR file. It's important that only HTTP scanner uses `URLConnection.getLastModified()` method to check if there were any changes since last iteration for each GAR-file before redeploying.

Following parameters are supported:

Parameter	Description	Optional	Default
<code>freq</code>	Scanning frequency in milliseconds.	Yes	300000 ms

## #Http Uri Example

The following example will download the page [www.mysite.com/ignite/deployment](http://www.mysite.com/ignite/deployment), parse it and download and deploy all GAR files specified by href attributes of <a> elements on the page using authentication `username:password` every '10000' milliseconds (only new/updated GAR-s).

```
http://username:password;freq=10000@www.mysite.com:110/ignite/deployment
```

## LocalDeploymentSpi

Local deployment SPI that implements only within VM deployment on local node via `register(ClassLoader, Class)` method. This SPI requires no configuration.

There is no point to explicitly configure `LocalDeploymentSpi` as it is used by default and has no configuration parameters.

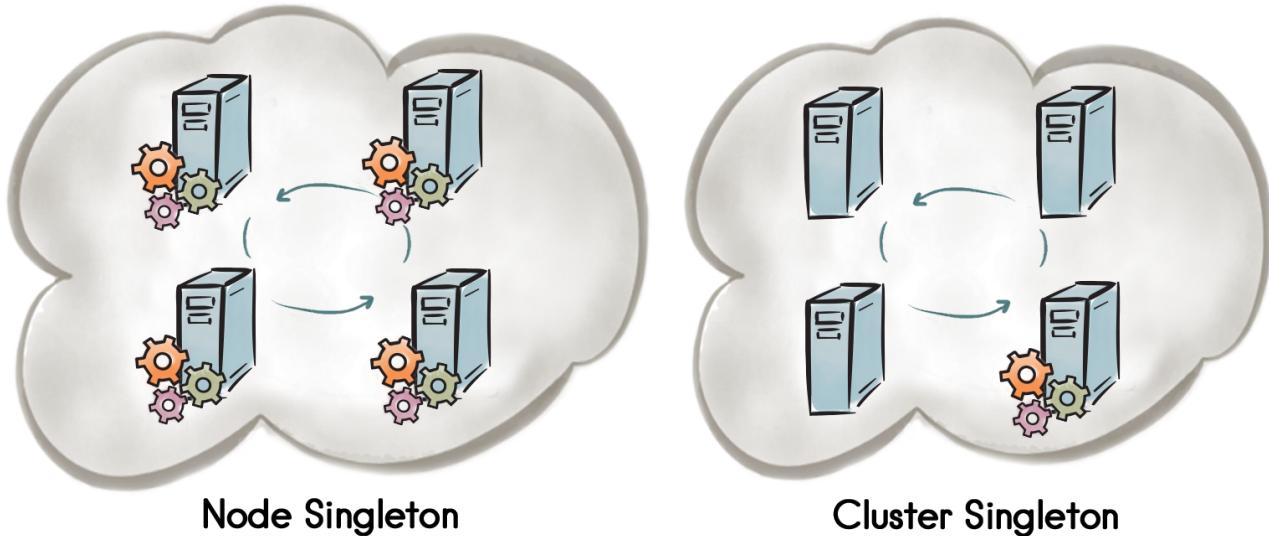
# Service Grid

# Service Grid

Cluster-enable any service or data structure.

Service Grid allows for deployments of arbitrary user-defined services on the cluster. You can implement and deploy any service, such as custom counters, ID generators, hierarchical maps, etc.

Ignite allows you to control how many instances of your service should be deployed on each cluster node and will automatically ensure proper deployment and fault tolerance of all the services .



## Features

- **Continuous availability** of deployed services regardless of topology changes or crashes.
- Automatically deploy any number of distributed service instances in the cluster.
- Automatically deploy **Cluster Singletons**, including cluster-singleton, node-singleton, or key-affinity-singleton.
- Automatically deploy distributed services on node start-up by specifying them in the configuration.
- Undeploy any of the deployed services.
- Get information about service deployment topology within the cluster.
- Create service proxy for accessing remotely deployed distributed services.



Please refer to [Service Example](#) for information on service deployment and accessing service API.



Note that it's required that all nodes have service classes deployed locally, otherwise Service Grid will not be fully-functional. Peer-deployment is not supported for services.

## IgniteServices

All service grid functionality is available via [IgniteServices](#) interface.

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

// Get services instance spanning all nodes in the cluster.
IgniteServices svcs = ignite.services();
```

You can also limit the scope of service deployment to a Cluster Group. In this case, services will only span the nodes within the cluster group.

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

ClusterGroup remoteGroup = ignite.cluster().forRemotes();

// Limit service deployment only to remote nodes (exclude the local node).
IgniteServices svcs = ignite.services(remoteGroup);
```

## Load Balancing

In all cases, other than singleton service deployment, Ignite will automatically make sure that about an equal number of services are deployed on each node within the cluster. Whenever cluster topology changes, Ignite will re-evaluate service deployments and may re-deploy an already deployed service on another node for better load balancing.

## Fault Tolerance

Ignite always guarantees that services are continuously available, and are deployed according to the specified configuration, regardless of any topology changes or node crashes.

## Service Example

Define and deploy your own service.

## Define Your Service Interface

As an example, let's define a simple counter service as a `MyCounterService` interface. Note that this is a simple Java interface without any special annotations or methods.

```
public interface MyCounterService {  
    /**  
  
     * Increment counter value and return the new value.  
     */  
    int increment() throws CacheException;  
  
    /**  
  
     * Get current counter value.  
     */  
    int get() throws CacheException;  
}
```

## Service Implementation

An implementation of a distributed service has to implement both, `Service` and `MyCounterService` interfaces.

We implement our counter service by storing the counter value in cache. The key for this counter value is the name of the service. This allows us to reuse the same cache for multiple instances of the counter service.

```
public class MyCounterServiceImpl implements Service, MyCounterService {  
    /** Auto-injected instance of Ignite. */  
    @IgniteInstanceResource  
    private Ignite ignite;  
  
    /** Distributed cache used to store counters. */  
    private IgniteCache<String, Integer> cache;  
  
    /** Service name. */  
    private String svcName;  
  
    /**
```

```

* Service initialization.

*/
@Override public void init(ServiceContext ctx) {
    // Pre-configured cache to store counters.
    cache = ignite.cache("myCounterCache");

    svcName = ctx.name();

    System.out.println("Service was initialized: " + svcName);
}

/** 

* Cancel this service.

*/
@Override public void cancel(ServiceContext ctx) {
    // Remove counter from cache.
    cache.remove(svcName);

    System.out.println("Service was cancelled: " + svcName);
}

/** 

* Start service execution.

*/
@Override public void execute(ServiceContext ctx) {
    // Since our service is simply represented by a counter
    // value stored in cache, there is nothing we need
    // to do in order to start it up.
    System.out.println("Executing distributed service: " + svcName);
}

@Override public int get() throws CacheException {
    Integer i = cache.get(svcName);

    return i == null ? 0 : i;
}

@Override public int increment() throws CacheException {
    return cache.invoke(svcName, new CounterEntryProcessor());
}

/** 

```

```

* Entry processor which atomically increments value currently stored in cache.

*/
private static class CounterEntryProcessor implements EntryProcessor<String, Integer,
Integer> {
    @Override public Integer process(MutableEntry<String, Integer> e, Object... args) {
        int newVal = e.exists() ? e.getValue() + 1 : 1;

        // Update cache.
        e.setValue(newVal);

        return newVal;
    }
}
}

```

## Service Deployment

We should deploy our counter service as per-node-singleton within the cluster group that has our cache "myCounterCache" deployed.

```

// Cluster group which includes all caching nodes.
ClusterGroup cacheGrp = ignite.cluster().forCache("myCounterService");

// Get an instance of IgniteServices for the cluster group.
IgniteServices svcs = ignite.services(cacheGrp);

// Deploy per-node singleton. An instance of the service
// will be deployed on every node within the cluster group.
svcs.deployNodeSingleton("myCounterService", new MyCounterServiceImpl());

```

## Service Proxy

You can access an instance of the deployed service from any node within the cluster. If the service is deployed on that node, then the locally deployed instance will be returned. Otherwise, if service is not locally available, a remote proxy for the service will be created automatically.

#Sticky vs Not-Sticky Proxies Proxies can be either **sticky** or not. If proxy is sticky, then Ignite will always go back to the same cluster node to contact a remotely deployed service. If proxy is **not-sticky**, then Ignite will load balance remote service proxy invocations among all cluster nodes on which the service is deployed.

```

// Get service proxy for the deployed service.
MyCounterService cntrSvc = ignite.services().
    serviceProxy("myCounterService", MyCounterService.class, /*not-sticky*/false);

// Invoke a method on 'MyCounterService' interface.
cntrSvc.increment();

// Print latest counter value from our counter service.
System.out.println("Incremented value : " + cntrSvc.get());

```

## Access Service from Computations

For convenience, you can inject an instance of service proxy into your computation using `@ServiceResource` annotation.

```

IgniteCompute compute = igntie.compute();

compute.run(new IgniteRunnable() {
    @ServiceResource(serviceName = "myCounterService");
    private MyCounterService counterSvc;

    public void run() {
        // Invoke a method on 'MyCounterService' interface.
        int newValue = cntrSvc.increment();

        // Print latest counter value from our counter service.
        System.out.println("Incremented value : " + newValue);
    }
});

```

## Cluster Singletons

Define various guaranteed cluster singletons.

`IgniteServices` facade allows to deploy any number of services on any of the grid nodes. However, the most commonly used feature is to deploy singleton services on the cluster. Ignite will manage the singleton contract regardless of topology changes and node crashes.



Note that in case of topology changes, due to network delays, there may be a temporary situation when a singleton service instance will be active on more than one node (e.g. crash detection delay).

## Cluster Singleton

You can deploy a cluster-wide singleton service. Ignite will guarantee that there is always one instance of the service in the cluster. In case the cluster node on which the service was deployed crashes or stops, Ignite will automatically redeploy it on another node. However, if the node on which the service is deployed remains in topology, then the service will always be deployed on that node only, regardless of topology changes.

```
IgniteServices svcs = ignite.services();
svcs.deployClusterSingleton("myClusterSingleton", new MyService());
```

The above method is analogous to calling

```
svcs.deployMultiple("myClusterSingleton", new MyService(), 1, 1)
```

## Node Singleton

You can deploy a per-node singleton service. Ignite will guarantee that there is always one instance of the service running on each node. Whenever new nodes are started within the cluster group, Ignite will automatically deploy one instance of the service on every new node.

```
IgniteServices svcs = ignite.services();
svcs.deployNodeSingleton("myNodeSingleton", new MyService());
```

The above method is analogous to calling

```
svcs.deployMultiple("myNodeSingleton", new MyService(), 0, 1);
```

## Cache Key Affinity Singleton

You can deploy one instance of this service on the primary node for a given affinity key. Whenever topology changes and primary key node assignment changes, Ignite will always make sure that the service is undeployed on the previous primary node and is deployed on the new primary node.

```
IgniteServices svcs = ignite.services();

svcs.deployKeyAffinitySingleton("myKeySingleton", new MyService(), "myCache", new
MyCacheKey());
```

The above method is analogous to calling

```
IgniteServices svcs = ignite.services();

ServiceConfiguration cfg = new ServiceConfiguration();

cfg.setName("myKeySingleton");
cfg.setService(new MyService());
cfg.setCacheName("myCache");
cfg.setAffinityKey(new MyCacheKey());
cfg.setTotalCount(1);
cfg.setMaxPerNodeCount(1);

svcs.deploy(cfg);
```

## Service Configuration

Service grid configuration.

---

In addition to deploying managed services by calling any of the provided `IgniteServices.deploy(...)` methods, you can also automatically deploy services on startup by setting `serviceConfiguration` property of `IgniteConfiguration`:

```

<bean class="org.apache.ignite.IgniteConfiguration">
    ...
    <!-- Distributed Service configuration. -->
    <property name="serviceConfiguration">
        <list>
            <bean class="org.apache.ignite.services.ServiceConfiguration">
                <property name="name" value="MyClusterSingletonSvc"/>
                <property name="maxPerNodeCount" value="1"/>
                <property name="totalCount" value="1"/>
                <property name="service">
                    <ref bean="myServiceImpl"/>
                </property>
            </bean>
        </list>
    </property>
</bean>

<bean id="myServiceImpl" class="foo.bar.MyServiceImpl">
    ...
</bean>

```

```

ServiceConfiguration svcCfg1 = new ServiceConfiguration();

// Cluster-wide singleton configuration.
svcCfg1.setName("MyClusterSingletonSvc");
svcCfg1.setMaxPerNodeCount(1);
svcCfg1.setTotalCount(1);
svcCfg1.setService(new MyClusterSingletonImpl());

ServiceConfiguration svcCfg2 = new ServiceConfiguration();

// Per-node singleton configuration.
svcCfg2.setName("MyNodeSingletonSvc");
svcCfg2.setMaxPerNodeCount(1);
svcCfg2.setService(new MyNodeSingletonImpl());

IgniteConfiguration igniteCfg = new IgniteConfiguration();

igniteCfg.setServiceConfiguration(svcCfg1, svcCfg2);
...

// Start Ignite node.
Ignition.start(gridCfg);

```

## Deploying After Startup

You can configure and deploy services after the startup of Ignite nodes. Besides multiple convenience methods that allow deployment of various [Cluster Singletons](#), you can also create and deploy service with custom configuration.

```
ServiceConfiguration cfg = new ServiceConfiguration();

cfg.setName("myService");
cfg.setService(new MyService());

// Maximum of 4 service instances within cluster.
cfg.setTotalCount(4);

// Maximum of 2 service instances per each Ignite node.
cfg.setMaxPerNodeCount(2);

ignite.services().deploy(cfg);
```

# Distributed Messaging

# Topic Based

Exchange custom messages between nodes across the cluster.

Ignite distributed messaging allows for topic based cluster-wide communication between all nodes. Messages with a specified message topic can be distributed to all or sub-group of nodes that have subscribed to that topic.

Ignite messaging is based on publish-subscribe paradigm where publishers and subscribers are connected together by a common topic. When one of the nodes sends a message A for topic T, it is published on all nodes that have subscribed to T.



Any new node joining the cluster automatically gets subscribed to all the topics that other nodes in the cluster (or [cluster group](/docs/cluster-groups)) are subscribed to. Distributed Messaging functionality in Ignite is provided via [IgniteMessaging](#) interface. You can get an instance of [IgniteMessaging](#), like so:

```
ignite = Ignition.ignite();

// Messaging instance over this cluster.
IgniteMessaging msg = ignite.message();

// Messaging instance over given cluster group (in this case, remote nodes).
IgniteMessaging rmtMsg = ignite.message(ignite.cluster().forRemotes());
```

## Publish Messages

Send methods help sending/publishing messages with a specified message topic to all nodes. Messages can be sent in **ordered** or **unordered** manner. ===== Ordered Messages [sendOrdered\(...\)](#) method can be used if you want to receive messages in the order they were sent. A timeout parameter is passed to specify how long a message will stay in the queue to wait for messages that are supposed to be sent before this message. If the timeout expires, then all the messages that have not yet arrived for a given topic on that node will be ignored. ===== Unordered Messages [send\(...\)](#) methods do not guarantee message ordering. This means that, when you sequentially send message A and message B, you are not guaranteed that the target node first receives A and then B.

## Subscribe for Messages

Listen methods help to listen/subscribe for messages. When these methods are called, a listener with

specified message topic is registered on all (or sub-group of ) nodes to listen for new messages. With listen methods, a predicate is passed that returns a boolean value which tells the listener to continue or stop listening for new messages. ===== Local Listen `localListen(...)` method registers a message listener with specified topic only on the local node and listens for messages from any node in **this** cluster group. ===== Remote Listen `remoteListen(...)` method registers message listeners with specified topic on all nodes in **this** cluster group and listens for messages from any node in **this** cluster group .

## Example

Following example shows message exchange between remote nodes.

### *Ordered Messaging*

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

IgniteMessaging rmtMsg = ignite.message(ignite.cluster().forRemotes());

// Add listener for unordered messages on all remote nodes.
rmtMsg.remoteListen("MyOrderedTopic", (nodeId, msg) -> {
    System.out.println("Received ordered message [msg=" + msg + ", from=" + nodeId + ']'
);

    return true; // Return true to continue listening.
});

// Send ordered messages to remote nodes.
for (int i = 0; i < 10; i++)
    rmtMsg.sendOrdered("MyOrderedTopic", Integer.toString(i));
```

## Unordered Messaging

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

IgniteMessaging rmtMsg = ignite.message(ignite.cluster().forRemotes());

// Add listener for unordered messages on all remote nodes.
rmtMsg.remoteListen("MyUnOrderedTopic", (nodeId, msg) -> {
    System.out.println("Received unordered message [msg=" + msg + ", from=" + nodeId + ']');

    return true; // Return true to continue listening.
});

// Send unordered messages to remote nodes.
for (int i = 0; i < 10; i++)
    rmtMsg.send("MyUnOrderedTopic", Integer.toString(i));
```

## java7 ordered

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

// Get cluster group of remote nodes.
ClusterGroup rmtPrj = ignite.cluster().forRemotes();

// Get messaging instance over remote nodes.
IgniteMessaging msg = ignite.message(rmtPrj);

// Add message listener for specified topic on all remote nodes.
msg.remoteListen("myOrderedTopic", new IgniteBiPredicate<UUID, String>() {
    @Override public boolean apply(UUID nodeId, String msg) {
        System.out.println("Received ordered message [msg=" + msg + ", from=" + nodeId + ']');

        return true; // Return true to continue listening.
    }
});

// Send ordered messages to all remote nodes.
for (int i = 0; i < 10; i++)
    msg.sendOrdered("myOrderedTopic", Integer.toString(i), 0);
```

# Distributed Events

# Local and Remote Events

Get notified of any state change or event occurring cluster-wide.

Ignite distributed events functionality allows applications to receive notifications when a variety of events occur in the distributed grid environment. You can automatically get notified for task executions, read, write or query operations occurring on local or remote nodes within the cluster.

Distributed events functionality is provided via [IgniteEvents](#) interface. You can get an instance of [IgniteEvents](#) from Ignite as follows:

```
Ignite ignite = Ignition.ignite();
IgniteEvents evts = ignite.events();
```

## Subscribe for Events

Listen methods can be used to receive notification for specified events happening in the cluster. These methods register a listener on local or remotes nodes for the specified events. Whenever the event occurs on the node, the listener is notified. ===== Local Events [localListen\(…\)](#) method registers event listeners with specified events on local node only. ===== Remote Events [remoteListen\(…\)](#) method registers event listeners with specified events on all nodes within the cluster or cluster group. Following is an example of each method:

### *local listen*

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

// Local listener that listens to local events.
IgnitePredicate<CacheEvent> locLsnr = evt -> {
    System.out.println("Received event [evt=" + evt.name() + ", key=" + evt.key() +
        ", oldVal=" + evt.oldValue() + ", newVal=" + evt.newValue());

    return true; // Continue listening.
};

// Subscribe to specified cache events occurring on local node.
ignite.events().localListen(locLsnr,
    EventType.EVT_CACHE_OBJECT_PUT,
    EventType.EVT_CACHE_OBJECT_READ,
    EventType.EVT_CACHE_OBJECT_REMOVED);

// Get an instance of named cache.
final IgniteCache<Integer, String> cache = ignite.cache("cacheName");

// Generate cache events.
for (int i = 0; i < 20; i++)
    cache.put(i, Integer.toString(i));
```

### *remote listen*

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

// Get an instance of named cache.
final IgniteCache<Integer, String> cache = ignite.jcache("cacheName");

// Sample remote filter which only accepts events for keys
// that are greater than or equal to 10.
IgnitePredicate<CacheEvent> rmtLsnr = evt -> evt.<Integer>key() >= 10;

// Subscribe to specified cache events on all nodes that have cache running.
ignite.events(ignite.cluster().forCacheNodes("cacheName")).remoteListen(null, rmtLsnr,
    EventType.EVT_CACHE_OBJECT_PUT,
    EventType.EVT_CACHE_OBJECT_READ,
    EventType.EVT_CACHE_OBJECT_REMOVED);

// Generate cache events.
for (int i = 0; i < 20; i++)
    cache.put(i, Integer.toString(i));
```

*java7 listen*

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

// Get an instance of named cache.
final IgniteCache<Integer, String> cache = ignite.jcache("cacheName");

// Sample remote filter which only accepts events for keys
// that are greater than or equal to 10.
IgnitePredicate<CacheEvent> rmtLsnr = new IgnitePredicate<CacheEvent>() {
    @Override public boolean apply(CacheEvent evt) {
        System.out.println("Cache event: " + evt);

        int key = evt.key();

        return key >= 10;
    }
};

// Subscribe to specified cache events occurring on
// all nodes that have the specified cache running.
ignite.events(ignite.cluster().forCacheNodes("cacheName")).remoteListen(null, rmtLsnr,
EVT_CACHE_OBJECT_PUT,
EVT_CACHE_OBJECT_READ,
EVT_CACHE_OBJECT_REMOVED);

// Generate cache events.
for (int i = 0; i < 20; i++)
    cache.put(i, Integer.toString(i));
```

In the above example `EVT_CACHE_OBJECT_PUT`, `EVT_CACHE_OBJECT_READ`, and `EVT_CACHE_OBJECT_REMOVED` are pre-defined event type constants defined in `EventType` interface.



`EventType` interface defines various event type constants that can be used with listen methods. Refer to [javadoc](#) for complete list of these event types.



Event types passed in as parameter in `localListen(...)` and `remoteListen(...)` methods must also be configured in `IgniteConfiguration`. See [Configuration](#) example below.

## Query for Events

All events generated in the system are kept locally on the local node. `IgniteEvents` API provides methods to query for these events. ===== Local Events `localQuery(...)` method queries for events on the

local node using the passed in predicate filter. If all predicates are satisfied, a collection of events happening on the local node is returned. ===== Remote Events `remoteQuery(...)` method asynchronously queries for events on remote nodes in this projection using the passed in predicate filter. This operation is distributed and hence can fail on communication layer and generally can take much longer than local event notifications. Note that this method will not block and will return immediately with future.

## Configuration

To get notified of any tasks or cache events occurring within the cluster, `includeEventTypes` property of `IgniteConfiguration` must be enabled.

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <!-- Enable cache events. -->
    <property name="includeEventTypes">
        <util:constant static-field="org.apache.ignite.events.EventType.EVT_CACHE"/>
    </property>
    ...
</bean>
```

```
IgniteConfiguration cfg = new IgniteConfiguration();

// Enable cache events.
cfg.setIncludeEventTypes(EventType.EVT_CACHE);

// Start Ignite node.
Ignition.start(cfg);
```

By default, event notifications are turned off for performance reasons.



Since thousands of events per second are generated, it creates an additional load on the system. This can lead to significant performance degradation. Therefore, it is highly recommended to enable only those events that your application logic requires.

## Automatic Batching

Batch notifications to help attain high cache performance and low latency.

Ignite automatically groups or batches event notifications that are generated as a result of cache events occurring within the cluster.

Each activity in cache can result in an event notification being generated and sent. For systems where cache activity is high, getting notified for every event could be network intensive, possibly leading to a decreased performance of cache operations in the grid.

In Ignite, event notifications can be grouped together and sent in batches or timely intervals. Here is an example of how this can be done:

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

// Get an instance of named cache.
final IgniteCache<Integer, String> cache = ignite.jcache("cacheName");

// Sample remote filter which only accepts events for keys
// that are greater than or equal to 10.
IgnitePredicate<CacheEvent> rmtLsnr = new IgnitePredicate<CacheEvent>() {
    @Override public boolean apply(CacheEvent evt) {
        System.out.println("Cache event: " + evt);

        int key = evt.key();

        return key >= 10;
    }
};

// Subscribe to cache events occuring on all nodes
// that have the specified cache running.
// Send notifications in batches of 10.
ignite.events(ignite.cluster().forCacheNodes("cacheName")).remoteListen(
    10 /*batch size*/, 0 /*time intervals*/, false, null, rmtLsnr, EVTS_CACHE);

// Generate cache events.
for (int i = 0; i < 20; i++)
    cache.put(i, Integer.toString(i));
```

# **HTTP**

# Rest API

Connect to Ignite over HTTP REST protocol.

---

Ignite provides an HTTP REST client that gives you the ability to communicate with the grid over HTTP and HTTPS protocols using REST approach. REST APIs can be used to perform different operations like read/write from/to cache, execute tasks, get various metrics and more.

- [Returned value](#)
- [Log](#)
- [Version](#)
- [Decrement](#)
- [Increment](#)
- [\[Cache Metrics\]](#)
- [Compare-And-Swap](#)
- [Prepend](#)
- [Append](#)
- [Replace](#)
- [\[Get and Replace\]](#)
- [Replace Value](#)
- [Remove all](#)
- [\[Remove Value\]](#)
- [Remove](#)
- [Get and remove](#)
- [Add](#)
- [Put all](#)
- [Put](#)
- [Get all](#)
- [Get](#)
- [Contains key](#)
- [Contains keys](#)
- [Get and put](#)
- [Put if absent](#)

## Get and put if absent

- Cache size
- Get or create cache
- Destroy cache
- Node
- Topology
- Execute
- Result
- Sql query execute
- Sql fields query execute
- Sql query fetch
- Sql query close

## Returned value

---

HTTP REST request returns JSON object which has similar structure for each command. This object has the following structure:

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
affinityNodeId	string	Affinity node ID.	2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37
error	string	The field contains description of error if server could not handle the request	specifically for each command
response	jsonObject	The field contains result of command.	specifically for each command
successStatus	integer	Exit status code. It might have the following values: * success = 0 * failed = 1 * authorization failed = 2 * security check failed = 3	0

## Log

---

-

**Log** command shows server logs.

```
http://host:port/ignite?cmd=log&from=10&to=100&path=/var/log/ignite.log
```

## Request Parameters

name	type	optional	description	example
cmd	Should be <b>log</b> lowercase.		string	No
from	integer	Yes	Number of line to start from. Parameter is mandatory if <b>to</b> is passed.	0
path	string	Yes	The path to log file. If not provided, will be used the following value <b>work/log/ignite.log</b>	log/cache_server.log
to	integer	Yes	Number to line to finish on. Parameter is mandatory if <b>from</b> is passed.	1000

## Response example:

```
{  
  "error": "",  
  "response": "[14:01:56,626][INFO ][test-runner][GridDiscoveryManager] Topology  
snapshot [ver=1, nodes=1, CPUs=8, heap=1.8GB]",  
  "successStatus": 0  
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	["[14:01:56,626][INFO ][test-runner][GridDiscovery Manager] Topology snapshot [ver=1, nodes=1, CPUs=8, heap=1.8GB]"]	string	logs

## Version

---

**Version** command shows current Ignite version.

```
http://host:port/ignite?cmd=version
```

## Request Parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
-------------	-------------	-----------------	--------------------	----------------

## Response example

```
{
  "error": "",
  "response": "1.0.0",
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	1.0.0	string	Ignite version

## Decrement

---

**Decrement** command subtracts and gets current value of given atomic long.

```
http://host:port/ignite?cmd=decr&cacheName=partitionedCache&key=decrKey&init=15&delta=10
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>decr</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	key
string	The name of atomic long.	counter	init	long
Yes	Initial value.	15	42	Number to be subtracted.

## Response example

```
{  
    "affinityNodeId": "e05839d5-6648-43e7-a23b-78d7db9390d5",  
    "error": "",  
    "response": -42,  
    "successStatus": 0  
}
```

name	type	description	example
response	long	Value after operation.	-42

## Increment

**Increment** command adds and gets current value of given atomic long.

```
http://host:port/ignite?cmd=incr&cacheName=partitionedCache&key=incrKey&init=15&delta=10
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>incr</b> lowercase.	cacheName

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	key
string	The name of atomic long.	counter	init	long
Yes	Initial value.	15	42	Number to be added.

## Response example

```
{
  "affinityNodeId": "e05839d5-6648-43e7-a23b-78d7db9390d5",
  "error": "",
  "response": 42,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	long	Value after operation.	42

## Cache metrics

**Cache metrics** command shows metrics for Ignite cache.

```
http://host:port/ignite?cmd=cache&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>cache</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId

## Response example

```
{  
    "affinityNodeId": "",  
    "error": "",  
    "response": {  
        "createTime": 1415179251551,  
        "hits": 0,  
        "misses": 0,  
        "readTime": 1415179251551,  
        "reads": 0,  
        "writeTime": 1415179252198,  
        "writes": 2  
    },  
    "successStatus": 0  
}
```

name	type	description	example
response	jsonObject	The JSON object contains cache metrics such as create time, count reads and etc.	{ "createTime": 1415179251551, "hits": 0, "misses": 0, "readTime": 1415179251551, "reads": 0, "writeTime": 1415179252198, "writes": 2 }

## Compare-And-Swap

**CAS** command stores given key-value pair in cache only if the previous value is equal to the expected value passed in.

```
http://host:port/ignite?cmd=cas&key=casKey&val2=casOldVal&val1=casNewVal&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>cas</b> lowercase.	cacheName

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key
val	val2	string	string	string
			Key to store in cache.	Value associated with the given key.

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if replace happened, false otherwise.	true

## Prepend

**Prepend** command prepends a line for value which is associated with key.

```
http://host:port/ignite?cmd=prepend&key=prependKey&val=prefix_&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>prepend</b> lowercase.	cacheName

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key
val	string	string		

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if replace happened, false otherwise.	true

## Append

**Append** command appends a line for value which is associated with key.

```
http://host:port/ignite?cmd=append&key=appendKey&val=_suffix&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>append</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key
val	string	string		

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if replace happened, false otherwise.	true

## Replace

Replace command stores a given key-value pair in cache only if there is a previous mapping for it.

```
http://host:port/ignite?cmd=rep&key=repKey&val=newValue&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>rep</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
val	string	string		

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if replace happened, false otherwise.	true

## Get and replace

**Get and replace** command stores a given key-value pair in cache only if there is a previous mapping for it.

```
http://host:port/ignite?cmd=getrep&key=repKey&val=newValue&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>getrep</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key
val	string	string		

## Response example

```
{  
    "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",  
    "error": "",  
    "response": oldValue,  
    "successStatus": 0  
}
```

name	type	description	example
response	jsonObject	The previous value associated with the specified key.	{"name": "Bob"}

## Replace Value

**Replace Value** command replaces the entry for a key only if currently mapped to a given value.

```
http://host:port/ignite?cmd=repval&key=repKey&val=newValue&val2=oldVal&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>repval</b> lowercase.	cacheName
string	Yes		Cache name. If not provided, default cache will be used.	partitionedCache
string		oldValue		key
val	string	string		
Key to store in cache.	Value associated with the given key.	destId	string	Yes

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if replace happened, false otherwise.	true

## Remove all

**Remove all** command removes given key mappings from cache.

```
http://host:port/ignite?cmd=rmvall&k1=rmKey1&k2=rmKey2&k3=rmKey3&cacheName=partitionedCache
&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>rmvall</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	k1...kN

## Response example

```
{
    "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
    "error": "",
    "response": true,
    "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if replace happened, false otherwise.	true

## Remove value

**Remove value** command removes the mapping for a key only if currently mapped to the given value.

```
http://host:port/ignite?cmd=rmvval&key=rmvKey&val=rmvVal&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>rmvval</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	val
string	Value expected to be associated with the specified key.	oldValue		key
string		Key whose mapping is to be removed from the cache.	destId	string

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	False if there was no matching key	true

## Remove

**Remove** command removes the given key mapping from cache.

```
http://host:port/ignite?cmd=rmv&key=rmvKey&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>rmv</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if replace happened, false otherwise.	true

## Get and remove

**Get and remove** command removes the given key mapping from cache and returns previous value.

```
http://host:port/ignite?cmd=getrmv&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>getrmv</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": value,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	Value for the key.	{"name": "bob"}

## Add

**Add** command stores a given key-value pair in cache only if there isn't a previous mapping for it.

```
http://host:port/ignite?cmd=add&key=newKey&val=newValue&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>add</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key
string	Key to be associated with the value.	name	val	string

## Response example

```
{  
    "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",  
    "error": "",  
    "response": true,  
    "successStatus": 0  
}
```

name	type	description	example
response	boolean	True if value was stored in cache, false otherwise.	true

## Put all

**Put all** command stores the given key-value pairs in cache.

```
http://host:port/ignite?cmd=putall&k1=putKey1&k2=putKey2&k3=putKey3&v1=value1&v2=value2&v3=value3&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>putall</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	k1...kN
string	Keys to be associated with values.	name	v1...vN	string

## Response example

```
{
  "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if values was stored in cache, false otherwise.	true

## Put

**Put** command stores the given key-value pair in cache.

```
http://host:port/ignite?cmd=put&key=newKey&val=newValue&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>put</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key
string	Key to be associated with values.	name	val	string

## Response example

```
{  
    "affinityNodeId": "1bcbac4b-3517-43ee-98d0-874b103ecf30",  
    "error": "",  
    "response": true,  
    "successStatus": 0  
}
```

name	type	description	example
response	boolean	True if value was stored in cache, false otherwise.	true

## Get all

**Get all** command retrieves values mapped to the specified keys from cache.

```
http://host:port/ignite?cmd=getall&k1=getKey1&k2=getKey2&k3=getKey3&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>get</b> lowercase.	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	k1...kN

## Response example

```
{
  "affinityNodeId": "",
  "error": "",
  "response": {
    "key1": "value1",
    "key2": "value2"
  },
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	The map of key-value pairs.	{"key1": "value1", "key2": "value2"}

## Get

**Get** command retrieves value mapped to the specified key from cache.

```
http://host:port/ignite?cmd=get&key=getKey&cacheName=partitionedCache&destId=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>get</b> lowercase.	cacheName

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
string	Yes	Cache name. If not provided, default cache will be used.	partitionedCache	destId
string	Node ID for which the metrics are to be returned.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	key

## Response example

```
{
  "affinityNodeId": "2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37",
  "error": "",
  "response": "value",
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	Value for the given key.	{"name": "bob"}

## Contains key

**Contains key** command determines if cache contains an entry for the specified key.

```
http://host:port/ignite?cmd=conkey&key=getKey&cacheName=partitionedCache
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd	string	Should be <b>conkey</b> lowercase.	cacheName	string
Yes	Cache name. If not provided, default cache will be used.	partitionedCache	key	string
Key whose presence in this cache is to be tested.	testKey	destId	string	Yes

## Response example

```
{  
    "affinityNodeId": "2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37",  
    "error": "",  
    "response": true,  
    "successStatus": 0  
}
```

name	type	description	example
response	boolean	True if this map contains a mapping for the specified key.	true

## Contains keys

**Contains keys** command determines if cache contains an entries for the specified keys.

```
http://host:port/ignite?cmd=conkeys&k1=getKey1&k2=getKey2&k3=getKey3&cacheName=partitionedCache
```

## Request parameters

cmd	string	name	type	optional
description	example	Should be <b>conkeys</b> lowercase.	cacheName	string
Yes	Cache name. If not provided, default cache will be used.	partitionedCache	k1...kN	string
Key whose presence in this cache is to be tested.	key1, key2, ..., keyN	destId	string	Yes

## Response example

```
{
  "affinityNodeId": "2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if this cache contains a mapping for the specified keys.	true

## Get and put

**Get and put** command stores the given key-value pair in cache and returns an existing value if one existed.

```
http://host:port/ignite?cmd=getput&key=getKey&val=newVal&cacheName=partitionedCache
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd	string	Should be <b>getput</b> lowercase.	cacheName	string
Yes	Cache name. If not provided, default cache will be used.	partitionedCache	key	string
Key to be associated with value.	name	val	string	Value to be associated with key.
Jack	destId	string	Yes	Node ID for which the metrics are to be returned.

## Response example

```
{
  "affinityNodeId": "2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37",
  "error": "",
  "response": "value",
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	Previous value for the given key.	{"name": "bob"}

## Put if absent

**Put if absent** command stores given key-value pair in cache only if cache had no previous mapping for it.

```
http://host:port/ignite?cmd=putifabs&key=getKey&val=newVal&cacheName=partitionedCache
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd	string	Should be <b>putifabs</b> lowercase.	cacheName	string
Yes	Cache name. If not provided, default cache will be used.	partitionedCache	key	string
Key to be associated with value.	name	val	string	Value to be associated with key.
Jack	destId	string	Yes	Node ID for which the metrics are to be returned.

## Response example

```
{
  "affinityNodeId": "2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37",
  "error": "",
  "response": true,
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	boolean	True if a value was set.	true

## Get and put if absent

**Get and put if absent** command stores given key-value pair in cache only if cache had no previous mapping for it. If cache previously contained value for the given key, then this value is returned.

```
http://host:port/ignite?cmd=getputifabs&key=getKey&val=newVal&cacheName=partitionedCache
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd	string	Should be <b>getputifabs</b> lowercase.	cacheName	string
Yes	Cache name. If not provided, default cache will be used.	partitionedCache	key	string
Key to be associated with value.	name	val	string	Value to be associated with key.
Jack	destId	string	Yes	Node ID for which the metrics are to be returned.

## Response example

```
{
    "affinityNodeId": "2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37",
    "error": "",
    "response": "value",
    "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	Previous value for the given key.	{"name": "bob"}

## Cache size

**Cache size** command gets the number of all entries cached across all nodes.

```
http://host:port/ignite?cmd=size&cacheName=partitionedCache
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd	string	Should be <b>size</b> lowercase.	cacheName	string

## Response example

```
{
    "affinityNodeId": "",
    "error": "",
    "response": 1,
    "successStatus": 0
}
```

<b>response</b>	<b>number</b>	<b>name</b>	<b>type</b>
description	example	Number of all entries cached across all nodes.	5

## Get or create cache

**Get or create cache** command creates cache with given name if it does not exist.

```
http://host:port/ignite?cmd=getOrCreate&cacheName=partitionedCache
```

### Request parameters

name	type	optional	description	example
cmd	string	Should be <b>getOrCreate</b> lowercase.	cacheName	string

### Response example

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

## Destroy cache

**Destroy cache** command destroys cache with given name.

```
http://host:port/ignite?cmd=destCache&cacheName=partitionedCache
```

### Request parameters

name	type	optional	description	example
cmd	string	Should be <b>destCache</b> lowercase.	cacheName	string

### Response example

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

## Node

**Node** command gets information about a node.

```
http://host:port/ignite?cmd=node&attr=true&mtr=true&id=c981d2a1-878b-4c67-96f6-70f93a4cd241
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>node</b> lowercase.	mtr
boolean	Yes	Response will include metrics, if this parameter has value true.	true	id
string	This parameter is optional, if ip parameter is passed. Response will be returned for node which has the node ID.	8daab5ea-af83-4d91-99b6-77ed2ca06647		attr
boolean	Response will include attributes, if this parameter has value true.	true	ip	string

## Response example

```
{
  "error": "",
  "response": {
    "attributes": null,
    "caches": {},
    "consistentId": "127.0.0.1:47500",
    "defaultCacheMode": "REPLICATED",
    "metrics": null,
    "nodeId": "2d0d6510-6fed-4fa3-b813-20f83ac4a1a9",
    "replicaCount": 128,
    "tcpAddresses": ["127.0.0.1"],
    "tcpHostNames": [],
    "tcpPort": 11211
  },
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	Information about only one node.	{ "attributes": null, "caches": {}, "consistentId": "127.0.0.1:47500", "defaultCacheMode": "REPLICATED", "metrics": null, "nodeId": "2d0d6510-6fed-4fa3-b813-20f83ac4a1a9", "replicaCount": 128, "tcpAddresses": ["127.0.0.1"], "tcpHostNames": [], "tcpPort": 11211 }

## Topology

**Topology** command gets information about grid topology.

```
http://host:port/ignite?cmd=top&attr=true&mtr=true&id=c981d2a1-878b-4c67-96f6-70f93a4cd241
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>top</b> lowercase.	mtr
boolean	Yes	Response will include metrics, if this parameter has value true.	true	id
string	This parameter is optional, if ip parameter is passed. Response will be returned for node which has the node ID.	8daab5ea-af83-4d91-99b6-77ed2ca06647	Yes	attr
boolean	Response will include attributes, if this parameter has value true.	true	ip	string

## Response example

```
{  
  "error": "",  
  "response": [  
    {  
      "attributes": {  
        ...  
      },  
      "caches": {},  
      "consistentId": "127.0.0.1:47500",  
      "defaultCacheMode": "REPLICATED",  
      "metrics": {  
        ...  
      },  
      "nodeId": "96baebd6-dedc-4a68-84fd-f804ee1ed995",  
      "replicaCount": 128,  
      "tcpAddresses": ["127.0.0.1"],  
      "tcpHostNames": [""],  
      "tcpPort": 11211  
    },  
    {  
      "attributes": {  
        ...  
      },  
      "caches": {},  
      "consistentId": "127.0.0.1:47501",  
      "defaultCacheMode": "REPLICATED",  
      "metrics": {  
        ...  
      },  
      "nodeId": "2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37",  
      "replicaCount": 128,  
      "tcpAddresses": ["127.0.0.1"],  
      "tcpHostNames": [""],  
      "tcpPort": 11212  
    }  
  ],  
  "successStatus": 0  
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	Information about grid topology.	[ { "attributes": { ... }, "caches": {}, "consistentId": "127.0.0.1:47500", "defaultCacheMode": "REPLICATED", "metrics": { ... }, "nodeId": "96baebd6-dedc-4a68-84fd-f804ee1ed995", ... "tcpPort": 11211 }, { "attributes": { ... }, "caches": {}, "consistentId": "127.0.0.1:47501", "defaultCacheMode": "REPLICATED", "metrics": { ... }, "nodeId": "2bd7b049-3fa0-4c44-9a6d-b5c7a597ce37", ... "tcpPort": 11212 } ]

## Execute

**Execute** command executes given task on grid.

```
http://host:port/ignite?cmd=exe&name=taskName&p1=param1&p2=param2&async=true
```

### Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>exe</b> lowercase.	name
string		Name of the task to execute.	summ	p1...pN
string	Argument of task execution.	arg1...argN	async	boolean

## Response example

```
{  
    "error": "",  
    "response": {  
        "error": "",  
        "finished": true,  
        "id": "~ee2d1688-2605-4613-8a57-6615a8cbcd1b",  
        "result": 4  
    },  
    "successStatus": 0  
}
```

name	type	description	example
response	jsonObject	JSON object contains message about error, unique identifier of task, result of computation and status of computation.	{ "error": "", "finished": true, "id": "~ee2d1688-2605-4613-8a57-6615a8cbcd1b", "result": 4 }

## Result

Result command returns computation result for the given task.

```
http://host:port/ignite?cmd=res&id=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

name	type	optional	description	example
cmd	string		Should be res lowercase.	id

## Response example

```
{
  "error": "",
  "response": {
    "error": "",
    "finished": true,
    "id": "69ad0c48941-4689aae0-6b0e-4d52-8758-ce8fe26f497d~4689aae0-6b0e-4d52-8758-
ce8fe26f497d",
    "result": 4
  },
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	JSON object contains message about error, ID of task, result of computation and status of computation.	{ "error": "", "finished": true, "id": "~ee2d1688-2605-4613-8a57-6615a8cbcd1b", "result": 4 }

## Result

**Result** command returns computation result for the given task.

```
http://host:port/ignite?cmd=res&id=8daab5ea-af83-4d91-99b6-77ed2ca06647
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>res</b> lowercase.	id

## Response example

```
{
  "error": "",
  "response": {
    "error": "",
    "finished": true,
    "id": "69ad0c48941-4689aae0-6b0e-4d52-8758-ce8fe26f497d~4689aae0-6b0e-4d52-8758-ce8fe26f497d",
    "result": 4
  },
  "successStatus": 0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	JSON object contains message about error, ID of task, result of computation and status of computation.	{ "error": "", "finished": true, "id": "~ee2d1688-2605-4613-8a57-6615a8cbcd1b", "result": 4 }

## Sql query execute

**Sql query execute** command runs sql query over cache.

```
http://host:port/ignite?cmd=qryexe&type=Person&pzs=10&cacheName=Person&arg1=1000&arg2=200
&qry=salary+%3E+%3F+and+salary+%3C%3D+%3F
```

## Request parameters

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
cmd		string	Should be <b>qryexe</b> lowercase.	type
string		Type for the query	String	pageSize
number		Page size for the query	3	cacheName
string	Yes	Cache name. If not provided, default cache will be used.	testCache	arg1...argN
string	Qyery arguments	1000,2000	qry	strings

## Response example

```
{  
    "error": "",  
    "response": {  
        "fieldsMetadata": [],  
        "items": [  
            {"key": 3, "value": {"name": "Jane", "id": 3, "salary": 2000}},  
            {"key": 0, "value": {"name": "John", "id": 0, "salary": 2000}}],  
        "last": true,  
        "queryId": 0},  
    "successStatus": 0  
}
```

name	type	description	example
response	jsonObject	JSON object contains result items for query, flag for last page and queryId for query fetching.	{ "fieldsMetadata":[], "items": [ {"key":3,"value": {"name": "Jane", "id": 3, "salary": 2000}}, {"key":0,"value": {"name": "John", "id": 0, "salary": 2000}}], "last": true, "queryId":0 }

## Sql fields query execute

**Sql fields query execute** command runs sql fields query over cache.

```
http://host:port/ignite?cmd=qryfldexe&pzs=10&cacheName=Person&qry=select+firstName%2C+lastName+from+Person
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>qryfldexe</b> lowercase.	pageSize
number		Page size for the query	3	cacheName

<b>name</b>	<b>type</b>	<b>optional</b>	<b>description</b>	<b>example</b>
string	Yes	Cache name. If not provided, default cache will be used.	testCache	arg1...argN
string	Query arguments	1000,2000	qry	strings

## Response example

```
{
  "error": "",
  "response":{
    "fieldsMetadata": [{"fieldName":"FIRSTNAME", "fieldTypeName":"java.lang.String",
"schemaName":"person", "typeName":"PERSON"}, {"fieldName":"LASTNAME", "fieldTypeName":
"java.lang.String", "schemaName":"person", "typeName":"PERSON"}],
    "items": [[["Jane", "Doe"], ["John", "Doe"]]],
    "last":true,
    "queryId":0
  },
  "successStatus":0
}
```

<b>name</b>	<b>type</b>	<b>description</b>	<b>example</b>
response	jsonObject	JSON object contains result items for query, fields query metadata, flag for last page and queryId for query fetching.	{   "fieldsMetadata": [{"fieldName":"FIRSTNAME", "fieldTypeName":"java.lang.String", "schemaName":"person", "typeName":"PERSON"}],   "items": [[["Jane", "Doe"], ["John", "Doe"]]],   "last":true, "queryId":0 }

## Sql query fetch

**Sql query fetch** command gets next page for the query.

```
http://host:port/ignite?cmd=qryfetch&pzs=10&qryId=5
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>qryfetch</b> lowercase.	
number		Page size for the query.	3	qryId

## Response example

```
{  
    "error": "",  
    "response": {  
        "fieldsMetadata": [],  
        "items": [["Jane", "Doe"], ["John", "Doe"]],  
        "last": true,  
        "queryId": 0  
    },  
    "successStatus": 0}
```

name	type	description	example
response	jsonObject	JSON object contains result items for query, flag for last page and queryId for query fetching.	{ "fieldsMetadata":[], "items": [["Jane", "Doe"], ["John", "Doe"]], "last": true, "queryId": 0 }

## Sql query close

**Sql query close** command closes query resources.

```
http://host:port/ignite?cmd=qrycls&qryId=5
```

## Request parameters

name	type	optional	description	example
cmd		string	Should be <b>qrycls</b> lowercase.	qryId

## Response example

```
{  
    "error": "",  
    "response":true,  
    "successStatus":0  
}
```

name	type	description	example
response	boolean	True if query closed successfully.	true

## Configuration

Internally, Ignite uses Jetty server to provide HTTP server features. HTTP REST client is configured with **ConnectorConfiguration** bean.

### General Configuration

Parameter name	Default value	Optional	Description
<b>setSecretKey(String)</b>	Defines secret key used for client authentication. When provided, client request must contain HTTP header <b>X-Signature</b> with Base64 encoded SHA1 hash of the string "[1];[2]", where [1] is timestamp in milliseconds and [2] is the secret key.	null	Yes

Parameter name	Default value	Optional	Description
<b>setPortRange(int)</b>	Port range for Jetty server. In case port provided in Jetty configuration or <b>IGNITE_JETTY_PORT</b> system property is already in use, Ignite will iteratively increment port by 1 and try binding once again until provided port range is exceeded.	<b>100</b>	Yes
<b>setJettyPath(String)</b>	Path to Jetty configuration file. Should be either absolute or relative to <b>IGNITE_HOME</b> . If not provided then GridGain will start Jetty server with simple HTTP connector. This connector will use <b>IGNITE_JETTY_HOST</b> and <b>IGNITE_JETTY_PORT</b> system properties as host and port respectively. In case <b>IGNITE_JETTY_HOST</b> is not provided, localhost will be used as default. In case <b>IGNITE_JETTY_PORT</b> is not provided, port 8080 will be used as default.	<b>null</b>	Yes

## Sample Jetty XML configuration

```

<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN"
"http://www.eclipse.org/jetty/configure.dtd">
<Configure id="Server" class="org.eclipse.jetty.server.Server">
  <Arg name="threadPool">
    <!-- Default queued blocking thread pool -->

```

```

<New class="org.eclipse.jetty.util.thread.QueuedThreadPool">
    <Set name="minThreads">20</Set>
    <Set name="maxThreads">200</Set>
</New>
</Arg>
<New id="httpCfg" class="org.eclipse.jetty.server.HttpConfiguration">
    <Set name="secureScheme">https</Set>
    <Set name="securePort">8443</Set>
    <Set name="sendServerVersion">true</Set>
    <Set name="sendDateHeader">true</Set>
</New>
<Call name="addConnector">
    <Arg>
        <New class="org.eclipse.jetty.server.ServerConnector">
            <Arg name="server"><Ref refid="Server"/></Arg>
            <Arg name="factories">
                <Array type="org.eclipse.jetty.server.ConnectionFactory">
                    <Item>
                        <New class="org.eclipse.jetty.server.HttpConnectionFactory">
                            <Ref refid="httpCfg"/>
                        </New>
                    </Item>
                </Array>
            </Arg>
            <Set name="host">
                <SystemProperty name="IGNITE_JETTY_HOST" default="localhost"/>
            </Set>
            <Set name="port">
                <SystemProperty name="IGNITE_JETTY_PORT" default="8080"/>
            </Set>
            <Set name="idleTimeout">30000</Set>
            <Set name="reuseAddress">true</Set>
        </New>
    </Arg>
</Call>
<Set name="handler">
    <New id="Handlers" class="org.eclipse.jetty.server.handler.HandlerCollection">
        <Set name="handlers">
            <Array type="org.eclipse.jetty.server.Handler">
                <Item>
                    <New id="Contexts" class=
"org.eclipse.jetty.server.handler.ContextHandlerCollection"/>
                </Item>
            </Array>
        </Set>
    </New>
</Set>
<Set name="stopAtShutdown">false</Set>

```

</Configure>

# In-Memory File System

# IGFS Native Ignite APIs

Distribute files and directories in-memory.

Ignite File System (IGFS) is an in-memory file system allowing work with files and directories over existing cache infrastructure. IGFS can either work as purely in-memory file system, or delegate to another file system (e.g. various Hadoop file system implementations) acting as a caching layer. In addition IGFS provides API to execute map-reduce tasks over file system data.

## IgniteFileSystem

`IgniteFileSystem` interface is a gateway into Ignite file system implementation. It provides methods for regular file system operations such as `create`, `delete`, `mkdirs`, etc., as well as methods for map-reduce tasks execution.

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

// Obtain instance of IGFS named "myFileSystem".
IgniteFileSystem fs = ignite.fileSystem("myFileSystem");
```

IGFS can be configured either through Spring XML file or programmatically.

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="fileSystemConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.FileSystemConfiguration">
                <!-- Distinguished file system name. -->
                <property name="name" value="myFileSystem" />
                <!-- Name of the cache where file system structure will be stored. Should be
                    configured separately. -->
                <property name="metaCacheName" value="myMetaCache" />
                <!-- Name of the cache where file data will be stored. Should be configured
                    separately. -->
                <property name="dataCacheName" value="myDataCache" />
            </bean>
        </list>
    </property>
</bean>
```

```
IgniteConfiguration cfg = new IgniteConfiguration();
...
FileSystemConfiguration fileSystemCfg = new FileSystemConfiguration();
fileSystemCfg.setName("myFileSystem");
fileSystemCfg.setMetaCacheName("myMetaCache");
fileSystemCfg.setDataCacheName("myDataCache");
...
cfg.setFileSystemConfiguration(fileSystemCfg);
```

## Work with files and directories

```
IgniteFileSystem fs = ignite.fileSystem("myFileSystem");

// Create directory.
IgfsPath dir = new IgfsPath("/myDir");

fs.mkdirs(dir);

// Create file and write some data to it.
IgfsPath file = new IgfsPath(dir, "myFile");

try (OutputStream out = fs.create(file, true)) {
    out.write(...);
}

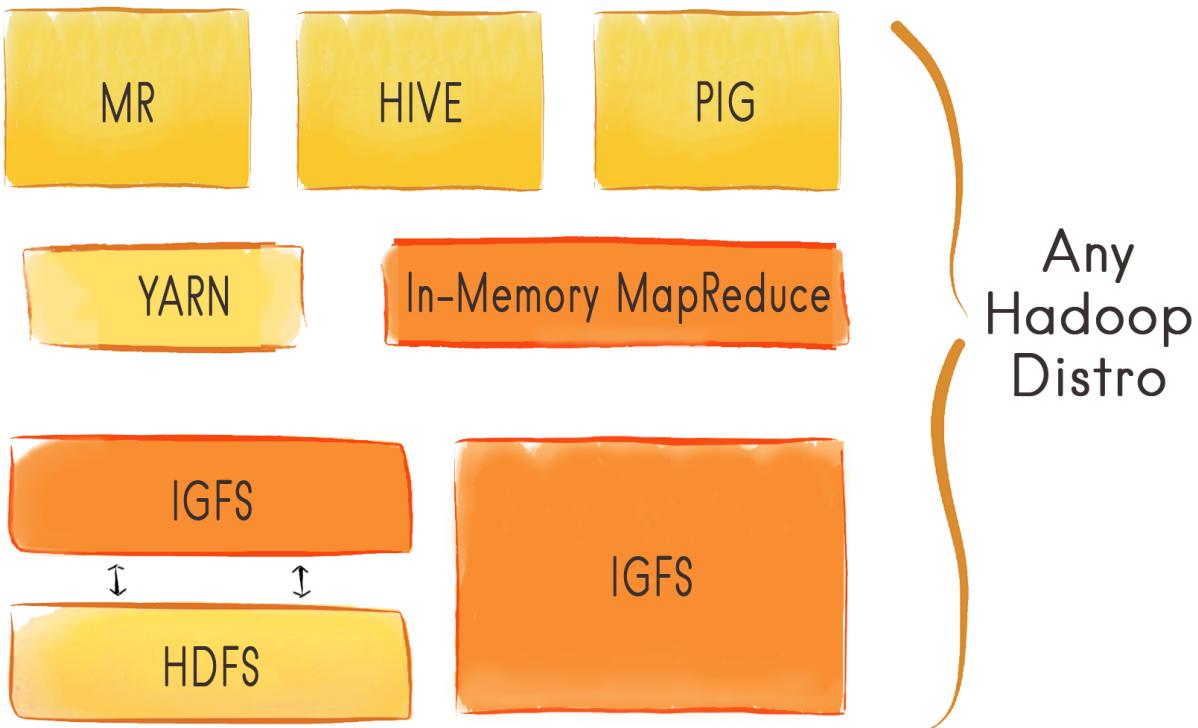
// Read from file.
try (InputStream in = fs.open(file)) {
    in.read(...);
}

// Delete directory.
fs.delete(dir, true);
```

## IGFS as Hadoop FileSystem

In-Memory Distributed Hadoop-Compliant File System

Ignite Hadoop Accelerator ships with Hadoop-compliant **IGFS File System** implementation called **IgniteHadoopFileSystem**. Hadoop can run over this file system in plug-n-play fashion and significantly reduce I/O and improve both, latency and throughput.



## Configure Ignite

Apache Ignite Hadoop Accelerator performs file system operations within Ignite cluster. Several prerequisites must be satisfied.

- 1) **IGNITE\_HOME** environment variable must be set and point to the root of Ignite installation directory.
- 2) Each cluster node must have Hadoop jars in CLASSPATH. See respective Ignite installation guide for your Hadoop distribution for details.
- 3) **IGFS** must be configured on the cluster node. See <http://apacheignite.readme.io/v1.0/docs/igfs> for details on how to do that.
- 4) To let **IGFS** accept requests from Hadoop, an endpoint should be configured (default configuration file is  `${IGNITE_HOME}/config/default-config.xml`). Ignite offers two endpoint types:
  - **shmem** - working over shared memory (not available on Windows);
  - **tcp** - working over standard socket API.

Shared memory endpoint is the recommended approach if the code executing file system operations is on the same machine as Ignite node. Note that **port** parameter is also used in case of shared memory

communication mode to perform initial client-server handshake:

```
<bean class="org.apache.ignite.configuration.FileSystemConfiguration">
    ...
    <property name="ipcEndpointConfiguration">
        <bean class="org.apache.ignite.igfs.IgfsIpcEndpointConfiguration">
            <property name="type" value="SHMEM"/>
            <property name="port" value="12345"/>
        </bean>
    </property>
    ...
</bean>
```

```
FileSystemConfiguration fileSystemCfg = new FileSystemConfiguration();
...
IgfsIpcEndpointConfiguration endpointCfg = new IgfsIpcEndpointConfiguration();
endpointCfg.setType(IgfsEndpointType.SHMEM);
...
fileSystemCfg.setIpcEndpointConfiguration(endpointCfg);
```

TCP endpoint should be used when Ignite node is either located on another machine, or shared memory is not available.

```
<bean class="org.apache.ignite.configuration.FileSystemConfiguration">
    ...
    <property name="ipcEndpointConfiguration">
        <bean class="org.apache.ignite.igfs.IgfsIpcEndpointConfiguration">
            <property name="type" value="TCP"/>
            <property name="host" value="myHost"/>
            <property name="port" value="12345"/>
        </bean>
    </property>
    ...
</bean>
```

```
FileSystemConfiguration fileSystemCfg = new FileSystemConfiguration();
...
IgfsIpcEndpointConfiguration endpointCfg = new IgfsIpcEndpointConfiguration();
endpointCfg.setType(IgfsEndpointType.TCP);
endpointCfg.setHost("myHost");
...
fileSystemCfg.setIpcEndpointConfiguration(endpointCfg);
```

If host is not set, it defaults to **127.0.0.1**. If port is not set, it defaults to **10500**.

If ipcEndpointConfiguration is not set, then shared memory endpoint with default port will be used for Linux systems, and TCP endpoint with default port will be used for Windows.

## Run Ignite

When Ignite node is configured start it:

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

## Configure Hadoop

To run Hadoop job using Ignite job tracker three prerequisites must be satisfied:

- 1) **IGNITE\_HOME** environment variable must be set and point to the root of Ignite installation directory.
- 2) Hadoop must have Ignite JARS  **\${IGNITE\_HOME}\libs\ignite-core-[version].jar** and  **\${IGNITE\_HOME}\libs\hadoop\ignite-hadoop-[version].jar** in CLASSPATH.

This can be achieved in several ways.

- Add these JARs to **HADOOP\_CLASSPATH** environment variable.
  - Copy or symlink these JARs to the folder where your Hadoop installation stores shared libraries.  
See respective Ignite installation guide for your Hadoop distribution for details.
- 3) Ignite Hadoop Accelerator file system must be configured for the action you are going to perform.  
At the very least you must provide fully qualified file system class name:

```
<configuration>
  ...
  <property>
    <name>fs.igfs.impl</name>
    <value>org.apache.ignite.hadoop.fs.v1.IgniteHadoopFileSystem</value>
  </property>
  ...
</configuration>
```

If you want to set Ignite File System as a default file system for your environment, then add the following property:

```
<configuration>
  ...
  <property>
    <name>fs.default.name</name>
    <value>igfs:///</value>
  </property>
  ...
</configuration>
```

Here value is an URL of endpoint of the Ignite node with IGFS. Rules how to this URL must look like provided at the end of this article.

There are several ways how to pass these configuration to your Hadoop jobs.

**First**, you may create separate `core-site.xml` file with these configuration properties and use it for job runs:

**Second**, you may override default `core-site.xml` of your Hadoop installation. This will force all Hadoop jobs to pick Ignite jobs tracker by default unless it is overriden on job level somehow. **Note that you will not be able to HDFS in this case.**

**Third**, you may set these properties for particular job programmatically:

```
Configuration conf = new Configuration();
...
conf.set("fs.igfs.impl", "org.apache.ignite.hadoop.fs.v1.IgniteHadoopFileSystem");
conf.set("fs.default.name", "igfs:///");
...
Job job = new Job(conf, "word count");
...
```

## Run Hadoop

How you run a job depends on how you have configured your Hadoop.

If you created separate `core-site.xml`:

```
hadoop --config [path_to_config] [arguments]
```

If you modified default `core-site.xml`, then `--config` option is not necessary:

```
hadoop [arguments]
```

If you start the job programmatically, then submit it:

```
...
Job job = new Job(conf, "word count");
...
job.submit();
```

## File system URI

URI to access **IGFS** has the following structure: `igfs://[igfs_name@][host]:[port]/`, where:

- **igfs\_name** - optional name of IGFS to connect to (as specified in `FileSystemConfiguration.setName(...)`). Must always ends with `@` character. Defaults to `null` if omitted.
- **host** - optional IGFS endpoint host (`IgfsIpcEndpointConfiguration.host`). Defaults to `127.0.0.1`.
- **port** - optional IGFS endpoint port (`IgfsIpcEndpointConfiguration.port`). Defaults to `10500`.

Sample URIs:

- `igfs://myIgfs@myHost:12345/` - connect to IGFS named `myIgfs` running on specific host and port;
- `igfs://myIgfs@myHost/` - connect to IGFS named `myIgfs` running on specific host and default port;
- `igfs://myIgfs@/` - connect to IGFS named `myIgfs` running on localhost and default port;
- `igfs://myIgfs@:12345/` - connect to IGFS named `myIgfs` running on localhost and specific port;
- `igfs://myHost:12345/` - connect to IGFS with `null` name running on specific host and port;
- `igfs://myHost/` - connect to IGFS with `null` name running on specific host and default port;
- `igfs://:12345/` - connect to IGFS with `null` name running on localhost and specific port;
- `igfs:///` - connect to IGFS with `null` name running on localhost and default port.

## Hadoop FileSystem Cache

Delegate operations to another file system.

Ignite Hadoop Accelerator contains implementation of **IGFS** secondary file system `IgniteHadoopIgfsSecondaryFileSystem` which allows read-through and write-through for any Hadoop `FileSystem` implementation. To use secondary file system set it in **IGFS** configuration:

```

<bean class="org.apache.ignite.configuration.FileSystemConfiguration">
    ...
    <property name="secondaryFileSystem">
        <bean class="org.apache.ignite.hadoop.fs.IgniteHadoopIgfsSecondaryFileSystem">
            <constructor-arg value="hdfs://myHdfs:9000"/>
        </bean>
    </property>
</bean>

```

```

FileSystemConfiguration fileSystemCfg = new FileSystemConfiguration();
...
IgniteHadoopIgfsSecondaryFileSystem hadoopFileSystem = new
IgniteHadoopIgfsSecondaryFileSystem("hdfs://myHdfs:9000");
...
fileSystemCfg.setSecondaryFileSystem(hadoopFileSystem);

```

## IGFS Modes

Setup different operation modes for file system paths.

IGFS is able to operate in 4 modes: **PRIMARY**, **PROXY**, **DUAL\_SYNC** and **DUAL\_ASYNC**. Mode can be configured either for the whole file system or for particular paths. Modes are defined in **IgfsMode** enumeration. By default file system operates in **DUAL\_ASYNC** mode.

If secondary file system is not configured, all paths configured as **DUAL\_SYNC** or **DUAL\_ASYNC** will fallback to **PRIMARY** mode.

```

<bean class="org.apache.ignite.configuration.FileSystemConfiguration">
    ...
    <!-- Set default mode. -->
    <property name="defaultMode" value="DUAL_SYNC" />
    <!-- Configure '/tmp' and all child paths to work in PRIMARY mode. -->
    <property name="pathModes">
        <map>
            <entry key="/tmp/.*" value="PRIMARY"/>
        </map>
    </property>
</bean>

```

```
FileSystemConfiguration fileSystemCfg = new FileSystemConfiguration();
...
fileSystemCfg.setDefaultMode(IgfsMode.DUAL_SYNC);
...
Map<String, IgfsMode> pathModes = new HashMap<>();
pathModes.put("/tmp/.*", IgfsMode.PRIMARY);
fileSystemCfg.setPathModes(pathModes);
```

## PRIMARY mode

---

In this mode IGFS serves as a primary standalone distributed in-memory file system. Secondary file system is not used.

## PROXY mode

---

IGFS is restricted to operate on paths in PROXY mode. Exception will be thrown if any operation is invoked on such path.

## DUAL\_SYNC mode

---

In this mode IGFS will synchronously read-through from the secondary file system whenever data is requested and is not cached in memory, and synchronously write-through to it whenever data is updated/created in IGFS. Essentially, in this case IGFS serves as an intelligent caching layer on top of the secondary file system.

## DUAL\_ASYNC mode

---

Same as [DUAL\\_SYNC](#), but seconadry file system reads and writes are performed asynchronously. There is a lag between IGFS updates and secondary file system updates, however the performance of updates is significantly better than with [DUAL\\_SYNC](#) mode.

# Hadoop Accelerator

# Hadoop Accelerator

In-Memory Plug-n-Play Hadoop Accelerator

---

Apache Ignite Hadoop Accelerator provides a set of components allowing for in-memory Hadoop job execution and file system operations.

## MapReduce

---

Hadoop Accelerator ships with alternate high-performant implementation of job tracker which replaces standard Hadoop MapReduce. Use it to boost your Hadoop MapReduce job execution performance.

[MapReduce](#)

## IGFS - In-Memory FileSystem

---

Hadoop Accelerator ships with an implementation of Hadoop [FileSystem](#) which stores file system data in-memory using distributed Ignite File System ([IGFS](#)). Use it to minimize disk IO and improve performance of any file system operations.

[IGFS as Hadoop FileSystem](#)

## Secondary File System

---

Hadoop Accelerator ships with an implementation of [SecondaryFileSystem](#). This implementation can be injected into existing IGFS allowing for read-through and write-through behavior over any other Hadoop [FileSystem](#) implementation (e.g. [HDFS](#)). Use it if you want your [IGFS](#) to become an in-memory caching layer over disk-based [HDFS](#) or any other Hadoop-compliant file system.

[Hadoop FileSystem Cache](#)

## Supported Hadoop distributions

---

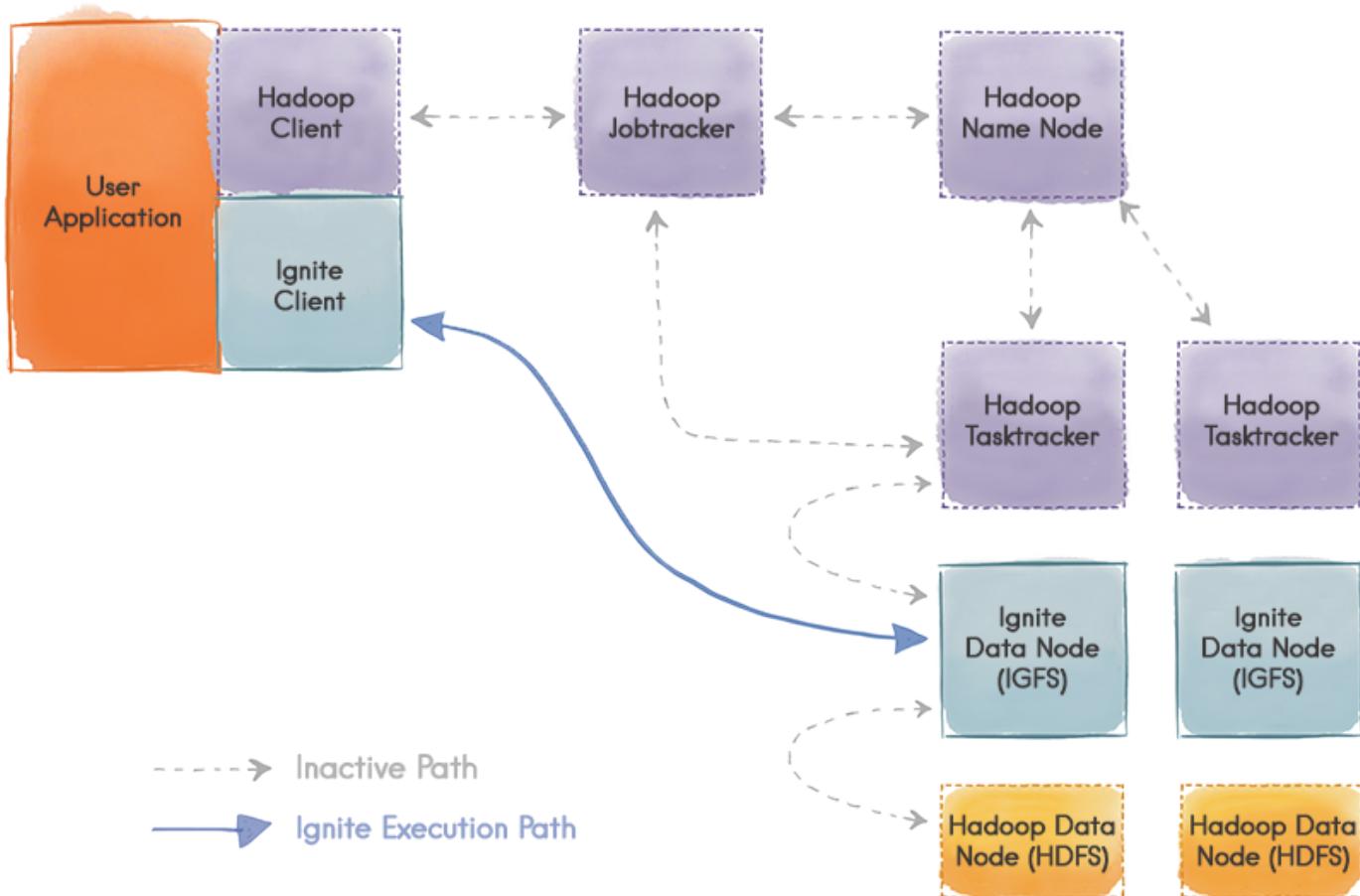
Apache Ignite Hadoop Accelerator can be used with a number of Hadoop distributions. Each distribution may require specific installation steps. See the following installation guides for more information:

- Installing on Apache Hadoop
- Installing on Cloudera CDH
- Installing on Hortonworks HDP
- Ignite and Apache Hive

## MapReduce

High Performance Real Time Hadoop MapReduce Engine.

Ignite In-Memory MapReduce allows to effectively parallelize the processing data stored in any Hadoop file system. It eliminates the overhead associated with job tracker and task trackers in a standard Hadoop architecture while providing low-latency, HPC-style distributed processing.



## Configure Ignite

Apache Ignite Hadoop Accelerator map-reduce engine processes Hadoop jobs within Ignite cluster. Several prerequisites must be satisfied.

- 1) `IGNITE_HOME` environment variable must be set and point to the root of Ignite installation directory.
- 2) Each cluster node must have Hadoop jars in CLASSPATH. See respective Ignite installation guide for your Hadoop distribution for details.
- 3) Cluster nodes accepts job execution requests listening particular socket. By default each Ignite node is listening for incoming requests on `127.0.0.1:11211`. You can override the host and port using `ConnectorConfiguration` class:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="connectorConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.ConnectorConfiguration">
                <property name="host" value="myHost" />
                <property name="port" value="12345" />
            </bean>
        </list>
    </property>
</bean>
```

## Run Ignite

When Ignite node is configured start it:

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

## Configure Hadoop

To run Hadoop job using Ignite job tracker several prerequisites must be satisfied:

- 1) `IGNITE_HOME` environment variable must be set and point to the root of Ignite installation directory.
- 2) Hadoop must have Ignite JARS ``${IGNITE_HOME}\libs\ignite-core-[version].jar`` and ```${IGNITE_HOME}\libs\hadoop\ignite-hadoop-[version].ja`` in CLASSPATH.

This can be achieved in several ways.

- Add these JARs to `HADOOP_CLASSPATH` environment variable.
- Copy or symlink these JARs to the folder where your Hadoop installation stores shared libraries. See respective Ignite installation guide for your Hadoop distribution for details.

3) Your Hadoop job must be configured to use Ignite job tracker. Two configuration properties are responsible for this:

- `mapreduce.framework.name` must be set to `ignite`
- `mapreduce.jobtracker.address` must be set to the host/port your Ignite nodes are listening.

This also can be achieved in several ways. **First**, you may create separate `mapred-site.xml` file with these configuration properties and use it for job runs:

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

**Second**, you may override default `mapred-site.xml` of your Hadoop installation. This will force all Hadoop jobs to pick Ignite jobs tracker by default unless it is overridden on job level somehow.

**Third**, you may set these properties for particular job programmatically:

```
Configuration conf = new Configuration();
...
conf.set(MRConfig.FRAMEWORK_NAME, IgniteHadoopClientProtocolProvider.FRAMEWORK_NAME);
conf.set(MRConfig.MASTER_ADDRESS, "127.0.0.1:11211");
...
Job job = new Job(conf, "word count");
...
```

## Run Hadoop

How you run a job depends on how you have configured your Hadoop.

If you created separate `mapred-site.xml`:

```
hadoop --config [path_to_config] [arguments]
```

If you modified default `mapred-site.xml`, then `--config` option is not necessary:

```
hadoop [arguments]
```

If you start the job programmatically, then submit it:

```
...
Job job = new Job(conf, "word count");
...
job.submit();
```

## Installing on Apache Hadoop

This article explains how to install Apache Ignite Hadoop Accelerator on Apache Hadoop distribution.

Please read the following articles first to get better understanding of product's architecture:

- <http://apacheignite.readme.io/v1.0/docs/overview>
- <http://apacheignite.readme.io/v1.0/docs/map-reduce>
- <http://apacheignite.readme.io/v1.0/docs/file-system>

### Ignite

- 1) Download the latest version of Apache Ignite Hadoop Accelerator and unpack it somewhere.
- 2) Set `IGNITE_HOME` variable to the directory where you unpacked Apache Ignite Hadoop Accelerator.
- 3) Ensure that `HADOOP_HOME` environment variable is set and valid. This is required for Ignite to find necessary Hadoop classes.
- 4) If you are going to use Ignite `FileSystem` implementation, configure `IGFS` in XML configuration (default configuration is  `${IGNITE_HOME}/config/default-config.xml`):

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="fileSystemConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.FileSystemConfiguration">
                <property name="metaCacheName" value="myMetaCache" />
                <property name="dataCacheName" value="myDataCache" />
            </bean>
        </list>
    </property>
    ...
</bean>

```

IGFS with this configuration will listen for incoming file system requests with default endpoint bound to **127.0.0.1:10500**. If you want to override it, provide alternate **ipcEndpointConfiguration** (see <http://apacheignite.readme.io/v1.0/docs/file-system>).

5) If you are going to use Ignite map-reduce engine for your jobs, no additional configuration is required, as node will listen for job execution requests with default endpoint bound to **127.0.0.1:11211**. If you want to override it, provide alternate 'ConnectorConfiguration' (see <http://apacheignite.readme.io/v1.0/docs/map-reduce>).

At this point Ignite node is ready to be started:

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

## Hadoop

1) Ensure that **IGNITE\_HOME** environment variable is set and points to the directory where you unpacked Apache Ignite Hadoop Accelerator.

2) Stop all Hadoop services.

3) Set Ignite JARs ``${IGNITE_HOME}\libs\ignite-core-[version].jar`` and ``${IGNITE_HOME}\libs\hadoop\ignite-hadoop-[version].jar`` to Hadoop CLASSPATH. You can either copy (or symlink) these JARs directly to Hadoop installation (e.g. to `'${HADOOP_HOME}/share/hadoop/common/lib'`)

```

ln -s ${IGNITE_HOME}/libs/ignite-core-1.0.0.jar
${HADOOP_HOME}/share/hadoop/common/lib/ignite-core-1.0.0.jar
ln -s ${IGNITE_HOME}/libs/hadoop/ignite-hadoop-1.0.0.jar
${HADOOP_HOME}/share/hadoop/common/lib/ignite-hadoop-1.0.0.jar

```

or set them to `HADOOP_CLASSPATH` environment variable:

```
export HADOOP_CLASSPATH=${IGNITE_HOME}/libs/ignite-core-1.0.0.jar:${IGNITE_HOME}/libs/ignite-hadoop/ignite-hadoop-1.0.0.jar
```

4) If you want to use Ignite `FileSystem`, configure it either in the separate `core-site.xml` file, or in default `core-site.xml` located in  `${HADOOP_HOME}/etc/hadoop`:

```
<configuration>
  ...
  <property>
    <name>fs.default.name</name>
    <value>igfs:///</value>
  </property>
  ...
  <property>
    <name>fs.igfs.impl</name>
    <value>org.apache.ignite.hadoop.fs.v1.IgniteHadoopFileSystem</value>
  </property>
  ...
</configuration>
```

Note that if you change `fs.default.name` to use Ignite FileSystem in default `core-site.xml`, Hadoop will not be able to work with `HDFS` anymore. If you want to use both Ignite `FileSystem` and `HDFS` at the same time, consider creating separate configuration file.

5) If you want to use Ignite `MapReduce` job tracker, configure it either in the separate `mapred-site.xml` file, or in default `mapred-site.xml` located in  `${HADOOP_HOME}/etc/hadoop`:

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

6) Start Hadoop.

7) At this point installation is finished and you can start running jobs. Run a job with separate `core-site.xml` and/or `mapred-site.xml` configuration files:

```
hadoop --config [path_to_config] [arguments]
```

Run a job with default configuration:

```
hadoop [arguments]
```

## Installing on Cloudera CDH

This article explains how to install Apache Ignite Hadoop Accelerator on Cloudera CDH distribution.

Please read the following articles first to get better understanding of product's architecture:

- <http://apacheignite.readme.io/v1.0/docs/overview>
- <http://apacheignite.readme.io/v1.0/docs/map-reduce>
- <http://apacheignite.readme.io/v1.0/docs/file-system>

### Ignite

- 1) Download the latest version of Apache Ignite Hadoop Accelerator and unpack it somewhere.
- 2) Set `IGNITE_HOME` variable to the directory where you unpacked Apache Ignite Hadoop Accelerator.
- 3) If you are going to use Ignite `FileSystem` implementation, configure `IGFS` in XML configuration (default configuration is  `${IGNITE_HOME}/config/default-config.xml`):

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="fileSystemConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.FileSystemConfiguration">
                <property name="metaCacheName" value="myMetaCache" />
                <property name="dataCacheName" value="myDataCache" />
            </bean>
        </list>
    </property>
    ...
</bean>

```

IGFS with this configuration will listen for incoming file system requests with default endpoint bound to **127.0.0.1:10500**. If you want to override it, provide alternate **ipcEndpointConfiguration** (see <http://apacheignite.readme.io/v1.0/docs/file-system>).

4) If you are going to use Ignite map-reduce engine for your jobs, no additional configuration is required, as node will listen for job execution requests with default endpoint bound to **127.0.0.1:11211**. If you want to override it, provide alternate 'ConnectorConfiguration' (see <http://apacheignite.readme.io/v1.0/docs/map-reduce>).

At this point Ignite node is ready to be started:

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

## CDH

1) Ensure that **IGNITE\_HOME** environment variable is set and points to the directory where you unpacked Apache Ignite Hadoop Accelerator.

2) Go to **Cloudera Manager** and stop all CDH services except of **Cloudera Management Services**.

3) Set Ignite JARs ``${IGNITE_HOME}\libs\ignite-core-[version].jar`` and ```${IGNITE_HOME}\libs\hadoop\ignite-hadoop-[version].jar`` to CDH CLASSPATH either copying these JARs or creating symlinks:

```

ln -s ${IGNITE_HOME}/libs/ignite-core-1.0.0.jar [path_to_CDH]/lib/hadoop/lib /ignite-
core-1.0.0.jar
ln -s ${IGNITE_HOME}/libs/hadoop/ignite-hadoop-1.0.0.jar
[path_to_CDH]/lib/hadoop/lib/ignite-hadoop-1.0.0.jar

```

4) If you want to use Ignite **FileSystem**, configure it in a separate **core-site.xml** file:

```
<configuration>
  ...
  <property>
    <name>fs.default.name</name>
    <value>igfs:///</value>
  </property>
  ...
  <property>
    <name>fs.igfs.impl</name>
    <value>org.apache.ignite.hadoop.fs.v1.IgniteHadoopFileSystem</value>
  </property>
  ...
</configuration>
```

Alternatively you can configure these properties in default CDH **core-site.xml**: **Cloudera Manager** → **YARN (MR2 Included)** → **Configuration** → **Service Wide** → **Advanced** → **YARN Service Advanced Configuration Snippet (Safety Valve)** for **core-site.xml**. Note that if you change **fs.default.name** to use Ignite FileSystem in default **core-site.xml**, CDH will not be able to work with **HDFS** anymore. If you want to use both Ignite **FileSystem** and **HDFS** at the same time, consider creating separate configuration file.

5) If you want to use Ignite **MapReduce** job tracker, configure it in a separate **mapred-site.xml** file:

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

Alternatively you can configure these properties in default CDH **mapred-site.xml**: **Cloudera Manager** → **YARN (MR2 Included)** → **Configuration** → **Service Wide** → **Advanced** → **YARN Service MapReduce Advanced Configuration Snippet (Safety Valve)**.

6) If you made any changes to default configuration(s), save them and re-deploy.

7) Start CDH services

8) At this point installation is finished and you can start running jobs. Run a job with separate `core-site.xml` and/or `mapred-site.xml` configuration files:

```
hadoop --config [path_to_config] [arguments]
```

Run a job with default configuration:

```
hadoop [arguments]
```

## Installing on Hortonworks HDP

This article explains how to install Apache Ignite Hadoop Accelerator on Hortonworks HDP distribution.

Please read the following articles first to get better understanding of product's architecture:

- <http://apacheignite.readme.io/v1.0/docs/overview>
- <http://apacheignite.readme.io/v1.0/docs/map-reduce>
- <http://apacheignite.readme.io/v1.0/docs/file-system>

### Ignite

1) Download the latest version of Apache Ignite Hadoop Accelerator and unpack it somewhere.

2) Set `IGNITE_HOME` variable to the directory where you unpacked Apache Ignite Hadoop Accelerator.

3) To let Ignite find required HDP JARs, create a file `/etc/default/hadoop` with the following content:

```
HDP=/usr/hdp/current  
export HADOOP_HOME=$HDP/hadoop-client/  
export HADOOP_COMMON_HOME=$HDP/hadoop-client/  
export HADOOP_HDFS_HOME=$HDP/hadoop-hdfs-client/  
export HADOOP_MAPRED_HOME=$HDP/hadoop-mapreduce-client/
```

4) If you are going to use Ignite `FileSystem` implementation, configure `IGFS` in XML configuration (default configuration is  `${IGNITE_HOME}/config/default-config.xml`):

```

<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="fileSystemConfiguration">
        <list>
            <bean class="org.apache.ignite.configuration.FileSystemConfiguration">
                <property name="metaCacheName" value="myMetaCache" />
                <property name="dataCacheName" value="myDataCache" />
            </bean>
        </list>
    </property>
    ...
</bean>

```

IGFS with this configuration will listen for incoming file system requests with default endpoint bound to **127.0.0.1:10500**. If you want to override it, provide alternate **ipcEndpointConfiguration** (see <http://apacheignite.readme.io/v1.0/docs/file-system>).

5) If you are going to use Ignite map-reduce engine for your jobs, no additional configuration is required, as node will listen for job execution requests with default endpoint bound to **127.0.0.1:11211**. If you want to override it, provide alternate 'ConnectorConfiguration' (see <http://apacheignite.readme.io/v1.0/docs/map-reduce>).

At this point Ignite node is ready to be started:

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

## HDP

- 1) Ensure that **IGNITE\_HOME** environment variable is set and points to the directory where you unpacked Apache Ignite Hadoop Accelerator.
- 2) Go to the **Ambari** web application and stop the following services: HDFS, MapReduce2 and YARN.
- 3) Set Ignite JARs  `${IGNITE_HOME}\libs\ignite-core-[version].jar` and  `${IGNITE_HOME}\libs\hadoop\ignite-hadoop-[version].jar`` to HDP CLASSPATH. To do this go to **HDFS → Configs → Advanced hadoop-env** section and edit the property **hadoop-env template**. Find the first export of 'HADOOP\_CLASSPATH' which typically looks as follows:

```
export HADOOP_CLASSPATH=${HADOOP_CLASSPATH}${JAVA_JDBC_LIBS}:${MAPREDUCE_LIBS}
```

Place the following after this line:

```
export HADOOP_CLASSPATH=${HADOOP_CLASSPATH}: ${IGNITE_HOME}/libs/ignite-core-1.0.0.jar: ${IGNITE_HOME}/libs/hadoop/ignite-hadoop-1.0.0.jar
```

4) If you want to use Ignite **FileSystem**, configure it in a separate **core-site.xml** file:

```
<configuration>
  ...
  <property>
    <name>fs.default.name</name>
    <value>igfs:///</value>
  </property>
  ...
  <property>
    <name>fs.igfs.impl</name>
    <value>org.apache.ignite.hadoop.fs.v1.IgniteHadoopFileSystem</value>
  </property>
  ...
</configuration>
```

Alternatively you can define **fs.igfs.impl** property in default **core-site.xml**: go to **HDFS → Configs → Custom core-site** and add the property **fs.igfs.impl** with value **org.apache.ignite.hadoop.fs.v1.IgniteHadoopFileSystem**.

Also, you can define **fs.default.name** property in default **core-site.xml**: go to **HDFS → Configs → Advanced core-site** and add the property **fs.default.name** with value **igfs://**. Note that if you change **fs.default.name** to use Ignite FileSystem in default **core-site.xml**, HDP will not be able to work with **HDFS** anymore. If you want to use both Ignite **FileSystem** and **HDFS** at the same time, consider creating separate configuration file.

5) If you want to use Ignite **MapReduce** job tracker, configure it in a separate **mapred-site.xml** file:

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

Alternatively you can configure these properties in default HDP `mapred-site.xml`:

- Create an empty archive anywhere in your system (e.g. with name `dummy.tar.gz`).
- Go to **MapReduce2 → Configs → Advanced mapred-site** section and change the following properties: `mapreduce.framework.name = ignite` `mapreduce.application.framework.path = /path/to/dummy.tar.gz` The latter property is required because HDP requires to define archive with Hadoop framework but in case of Ignite it doesn't make sense.
- Go to **MapReduce2 → Configs → Custom mapred-site** section and change the following property: `mapreduce.jobtracker.address = 127.0.0.1:11211`.

6) If you made any changes to default configuration(s), save them and restart all HDP services.

7) At this point installation is finished and you can start running jobs. Run a job with separate `core-site.xml` and/or `mapred-site.xml` configuration files:

```
hadoop --config [path_to_config] [arguments]
```

Run a job with default configuration:

```
hadoop [arguments]
```

## Ignite and Apache Hive

Running Apache Hive over "Ignited" Hadoop

---

This article explains how to properly configure and start Hive over Hadoop accelerated by Ignite. It also shows how to start HiveServer2 and a remote client with such configuration.

### Prerequisites

---

We assume that Hadoop is already installed and configured to run over Ignite, and Ignite node(s) providing `IGFS` file system and map-reduce job tracker functionality is up and running.

You will also need to install Hive: <http://hive.apache.org/>.

### Starting Hive

---

Here are the steps required to run Hive over "Ignited" Hadoop:

- Provide the location of correct `hadoop` executable. This can be done either with adding path to the executable file into `PATH` environment variable (note that this executable should be located in a folder named `bin/` anyway), or by specifying `HADOOP_HOME` environment variable.
- Provide the location of configuration files (`core-site.xml`, `hive-site.xml`, `mapred-site.xml`). To do this put all these files in a directory and specify the path to this directory as `HIVE_CONF_DIR` environment variable.



**Configuration Template.** We recommend to use Hive template configuration file `<IGNITE_HOME>/config/hadoop/hive-site.ignite.xml` to get Ignite specific settings.



There is a potential `issue` related to different `jline` library versions in Hive and Hadoop. It can be resolved by setting `HADOOP_USER_CLASSPATH_FIRST=true` environment variable. For convenience you can create a simple script that will properly set all required variables and run Hive, like this:

```
#Specify Hive home directory:  
export HIVE_HOME=<Hive installation directory>  
  
#Specify configuration files location:  
export HIVE_CONF_DIR=<Path to our configuration folder>  
  
#If you did not set hadoop executable in PATH, specify Hadoop home explicitly:  
export HADOOP_HOME=<Hadoop installation folder>  
  
#Avoid problem with different 'jline' library in Hadoop:  
export HADOOP_USER_CLASSPATH_FIRST=true  
  
${HIVE_HOME}/bin/hive "${@}"
```

This script can be used to start Hive interactive console:

```
$ hive-ig cli  
hive> show tables;  
OK  
u_data  
Time taken: 0.626 seconds, Fetched: 1 row(s)  
hive> quit;  
$
```

## Starting HiveServer2

You may also want to use [HiveServer2](#) for enhanced client features. To start it you can also use the script created above:

```
hive-ig --service hiveserver2
```

After the server is started, you can connect to it with any available [client](#). As a remote client, [beeline](#) can be run from any host, and it does not require any specific environment to work with "Ignited" Hive. Here is the example:

```
$ ./beeline
Beeline version 1.2.1 by Apache Hive
beeline> !connect jdbc:hive2://localhost:10000 scott tiger
org.apache.hive.jdbc.HiveDriver
Connecting to jdbc:hive2://localhost:10000
Connected to: Apache Hive (version 1.2.1)
Driver: Hive JDBC (version 1.2.1)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10000> show tables;
+-----+---+
| tab_name | 
+-----+---+
| u_data   | 
+-----+---+
1 row selected (0.957 seconds)
0: jdbc:hive2://localhost:10000>
```

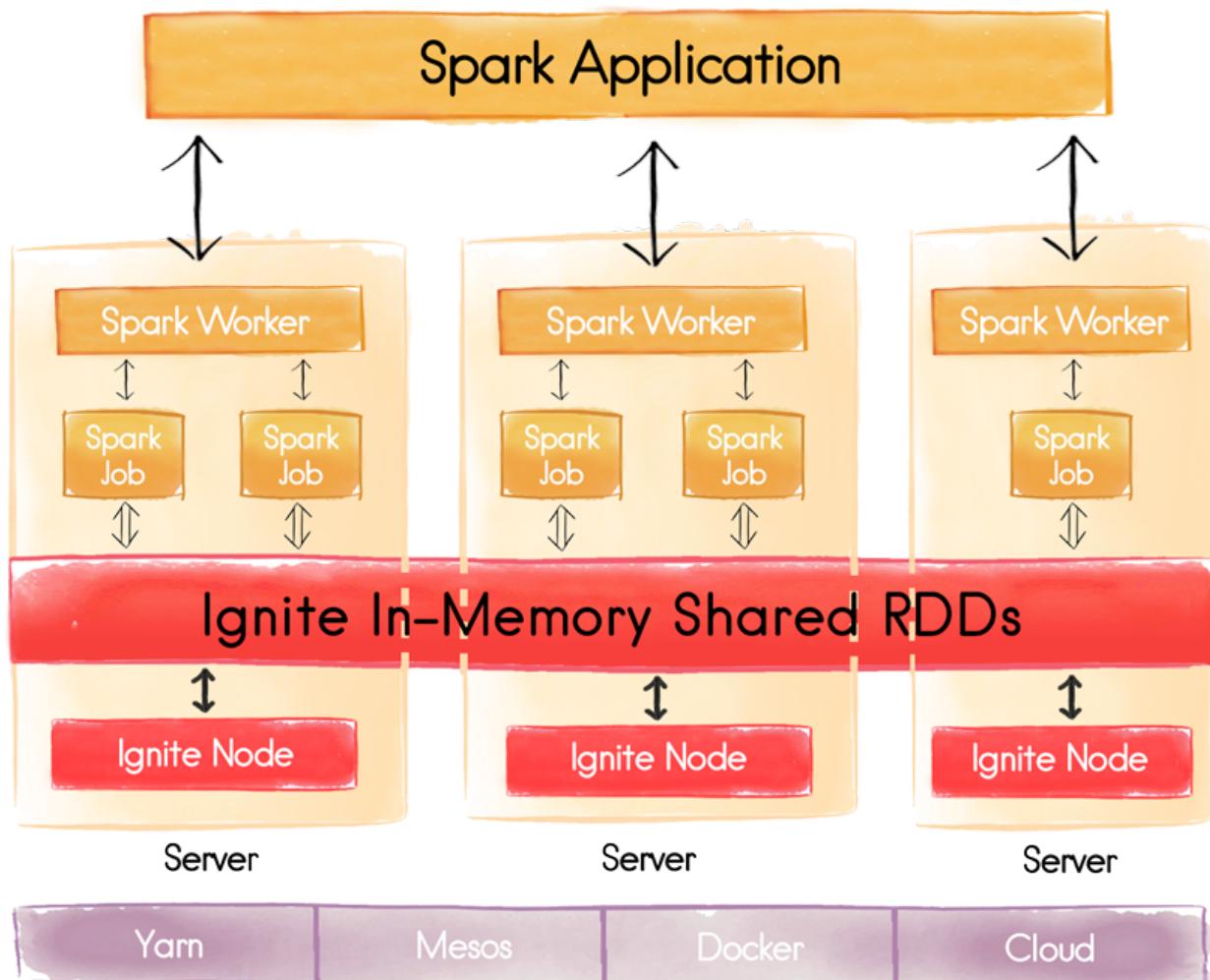
# Spark Shared RDDs

# Shared RDD Overview

Easily share state in memory across Spark jobs.

Apache Ignite provides an implementation of Spark RDD abstraction which allows to easily share state in memory across Spark jobs. The main difference between native Spark RDD and [IgniteRDD](#) is that Ignite RDD provides a shared in-memory view on data across different Spark jobs, workers, or applications, while native Spark RDD cannot be seen by other Spark jobs or applications.

The way IgniteRDD is implemented is as a view over a distributed Ignite cache, which may be deployed either within the Spark job executing process, or on a Spark worker, or in its own cluster. This means that depending on the chosen deployment mode the shared state may either exist only during the lifespan of a Spark application (embedded mode), or it may out-survive the Spark application (standalone mode) in which case the state can be shared across multiple Spark applications.



# IgniteRDD & IgniteContext

## IgniteContext

IgniteContext is the main entry point to Spark-Ignite integration. To create an instance of Ignite context, user must provide an instance of SparkContext and a closure creating [IgniteConfiguration](#) (configuration factory). Ignite context will make sure that server or client Ignite nodes exist in all involved job instances. Alternatively, a path to an XML configuration file can be passed to [IgniteContext](#) constructor which will be used to configure nodes being started.

When creating an [IgniteContext](#) instance, an optional boolean [client](#) argument (defaulting to [true](#)) can be passed to context constructor. This is typically used in a Shared Deployment installation. When [client](#) is set to [false](#), context will operate in embedded mode and will start server nodes on all workers during the context construction. This is required in an Embedded Deployment installation. See [Installation & Deployment](#) for information on deployment configurations.

Once [IgniteContext](#) is created, instances of [IgniteRDD](#) may be obtained using [fromCache](#) methods. It is not required that requested cache exist in Ignite cluster when RDD is created. If the cache with the given name does not exist, it will be created using provided configuration or template configuration.

For example, the following code will create an Ignite context with default Ignite configuration:

```
val igniteContext = new IgniteContext[Integer, Integer](sparkContext,  
    () => new IgniteConfiguration())
```

The following code will create an Ignite context configured from a file [example-cache.xml](#):

```
val igniteContext = new IgniteContext[Integer, Integer](sparkContext,  
    "examples/config/example-cache.xml")
```

## IgniteRDD

[IgniteRDD](#) is an implementation of Spark RDD abstraction representing a live view of Ignite cache. [IgniteRDD](#) is not immutable, all changes in Ignite cache (regardless whether they were caused by another RDD or external changes in cache) will be visible to RDD users immediately.

[IgniteRDD](#) utilizes partitioned nature of Ignite caches and provides partitioning information to Spark executor. Number of partitions in [IgniteRDD](#) equals to the number of partitions in underlying Ignite cache. [IgniteRDD](#) also provides affinity information to Spark via [getPreferredLocations](#) method so that

RDD computations use data locality.

## Reading values from Ignite

Since `IgniteRDD` is a live view of Ignite cache, there is no need to explicitly load data to Spark application from Ignite. All RDD methods are available to use right away after an instance of `IgniteRDD` is created.

For example, assuming an Ignite cache with name "partitioned" contains string values, the following code will find all values that contain the word "Ignite":

```
val cache = igniteContext.fromCache("partitioned")
val result = cache.filter(_.value.contains("Ignite")).collect()
```

## Saving values to Ignite

Since Ignite caches operate on key-value pairs, the most straightforward way to save values to Ignite cache is to use a Spark tuple RDD and `savePairs` method. This method will take advantage of the RDD partitioning and store value to cache in a parallel manner, if possible.

It is also possible to save value-only RDD into Ignite cache using `saveValues` method. In this case `IgniteRDD` will generate a unique affinity-local key for each value being stored into the cache.

For example, the following code will store pairs of integers from 1 to 10000 into cache named "partitioned" using 10 parallel store operations:

```
val cacheRdd = igniteContext.fromCache("partitioned")
cacheRdd.savePairs(sparkContext.parallelize(1 to 10000, 10).map(i => (i, i)))
```

## Running SQL queries against Ignite cache

When Ignite cache is configured with the indexing subsystem enabled, it is possible to run SQL queries against the cache using `objectSql` and `sql` methods. See [Cache Queries](#) for more information about Ignite SQL queries.

For example, assuming the "partitioned" cache is configured to index pairs of integers, the following code will get all integers in the range (10, 100):

```
val cacheRdd = igniteContext.fromCache("partitioned")
val result = cacheRdd.sql(
    "select _val from Integer where val > ? and val < ?", 10, 100)
```

# Installation & Deployment

---

## Shared Deployment

---

Shared deployment implies that Apache Ignite nodes are running independently from Apache Spark applications and store state even after Apache Spark jobs die. Similarly to Apache Spark there are three ways to deploy Apache Ignite to the cluster.

### Standalone deployment

In Standalone deployment mode Ignite nodes should be deployed together with Spark Worker nodes. Instruction on Ignite installation can be found [Getting Started](#). After you install Ignite on all worker nodes, start a node on each Spark worker with your config using `ignite.sh` script.

**#Adding Ignite libraries to Spark classpath by default == #Spark application deployment model allows dynamic jar distribution during application start.  
This model, however, has some drawbacks:**

- Spark dynamic class loader does not implement `getResource` methods, so you will not be able to access resources located in jar files.
- Java logger uses application class loader (not the context class loader) to load log handlers which results in `ClassNotFoundException` when using Java logging in Ignite.

There is a way to alter default Spark classpath for each launched application (this should be done on each machine of the Spark cluster, including master, worker and driver nodes).

1. Locate the `$SPARK_HOME/conf/spark-env.sh` file. If this file does not exist, create it from template using `$SPARK_HOME/conf/spark-env.sh.template`
2. Add the following lines to the end of the `spark-env.sh` file (uncomment the line setting `IGNITE_HOME` in case if you do not have it globally set):

```

#Optionally set IGNITE_HOME here.
#IGNITE_HOME=/path/to/ignite

IGNITE_LIBS="${IGNITE_HOME}/libs/*"

for file in ${IGNITE_HOME}/libs/*
do
    if [ -d ${file} ] && [ "${file}" != "${IGNITE_HOME}/libs/optional" ]; then
        IGNITE_LIBS=${IGNITE_LIBS}:${file}/*
    fi
done

export SPARK_CLASSPATH=$IGNITE_LIBS

```

You can verify that the Spark classpath is changed by running [bin/spark-shell](#) and typing a simple import statement:

```

scala> import org.apache.ignite.configuration._
import org.apache.ignite.configuration._

```

## Mesos deployment

Apache Ignite can be deployed on the Mesos cluster. Refer to the Mesos deployment instructions [Mesos Deployment](#).

## Embedded Deployment

---

Embedded deployment means that Apache Ignite nodes are started inside Apache Spark job processes and are stopped when job dies. There is no need for additional deployment steps in this case. Apache Ignite code will be distributed to the worker machines using Apache Spark deployment mechanism and nodes will be started on all workers as a part of [IgniteContext](#) initialization.

## Test Ignite with Spark-shell

Test Ignite RDD in Spark shell

---

## Starting up the cluster

Here we will briefly cover the process of Spark and Ignite cluster startup. Refer to [Spark](#)

[documentation](#) for more details.

For the testing you will need a Spark master process and at least one Spark worker. Usually Spark master and workers are separate machines, but for the test purposes you can start worker on the same machine where master starts.

1. Download and unpack Spark binary distribution to the same location (let it be `$SPARK_HOME`) on all nodes.
2. Download and unpack Ignite binary distribution to the same location (let it be `$IGNITE_HOME`) on all nodes.
3. On master node `cd` to `$SPARK_HOME` and run the following command:

```
sbin/start-master.sh
```

The script should output the path to log file of the started process. Check the log file for the master URL which has the following format: `spark://master_host:master_port`. Also check the log file for the Web UI url (usually it is `http://master_host:8080`).

1. On each of the worker nodes `cd` to `$SPARK_HOME` and run the following command:

```
bin/spark-class org.apache.spark.deploy.worker.Worker spark://master_host:master_port
```

where `spark://master_host:master_port` is the master URL you grabbed from the master log file. After workers has started check the master Web UI interface, it should show all of your workers registered in status `ALIVE`.

1. On each of the worker nodes `cd` to `$IGNITE_HOME` and start an Ignite node by running the following command:

```
bin/ignite.sh
```

You should see Ignite nodes discover each other with default configuration. If your network does not allow multicast traffic, you will need to change the default configuration file and configure TCP discovery.

## Working with spark-shell

Now, after you have your cluster up and running, you can run `spark-shell` and check the integration.

1. Start spark shell:
  - Either by providing Maven coordinates to Ignite artifacts (make sure to use `--repositories` if

working with GridGain community edition, otherwise `--repositories` may be omitted):

```
./bin/spark-shell  
--packages org.apache.ignite:ignite-spark:1.3.0  
--master spark://master_host:master_port  
--repositories http://www.gridgainsystems.com/nexus/content/repositories/external
```

- Or by providing paths to Ignite jar file paths using `--jars` parameter

```
./bin/spark-shell --jars path/to/ignite-core.jar,path/to/ignite-spark.jar,path/to/cache-  
api.jar,path/to/ignite-log4j.jar,path/to/log4j.jar --master  
spark://master_host:master_port
```

You should see Spark shell started up.

Note that if you are planning to use spring configuration loading, you will need to add the `ignite-spring` dependency as well:

```
./bin/spark-shell  
--packages org.apache.ignite:ignite-spark:1.3.0,org.apache.ignite:ignite-spring:1.3.0  
--master spark://master_host:master_port  
--repositories http://www.gridgainsystems.com/nexus/content/repositories/external
```

1. Let's create an instance of Ignite context using default configuration:

```
import org.apache.ignite.spark._  
import org.apache.ignite.configuration._  
  
val ic = new IgniteContext[Integer, Integer](sc, () => new IgniteConfiguration())
```

You should see something like

```
ic: org.apache.ignite.spark.IgniteContext[Integer, Integer] =  
org.apache.ignite.spark.IgniteContext@62be2836
```

An alternative way to create an instance of IgniteContext is to use a configuration file. Note that if path to configuration is specified in a relative form, then the `IGNITE_HOME` environment variable should be globally set in the system as the path is resolved relative to `IGNITE_HOME`

```
import org.apache.ignite.spark._  
import org.apache.ignite.configuration._  
  
val ic = new IgniteContext[Integer, Integer](sc, "config/default-config.xml")
```

1. Let's now create an instance of `IgniteRDD` using "partitioned" cache in default configuration:

```
val sharedRDD = ic.fromCache("partitioned")
```

You should see an instance of RDD created for partitioned cache:

```
shareRDD: org.apache.ignite.spark.IgniteRDD[Integer, Integer] = IgniteRDD[0] at RDD at  
IgniteAbstractRDD.scala:27
```

Note that creation of RDD is a local operation and will not create a cache in Ignite cluster.

1. Let's now actually ask Spark to do something with our RDD, for example, get all pairs where value is less than 10:

```
sharedRDD.filter(_._2 < 10).collect()
```

As our cache has not been filled yet, the result will be an empty array:

```
res0: Array[(Integer, Integer)] = Array()
```

Check the logs of remote spark workers and see how Ignite context will start clients on all remote workers in the cluster. You can also start command-line Visor and check that "partitioned" cache has been created.

1. Let's now save some values into Ignite:

```
sharedRDD.savePairs(sc.parallelize(1 to 100000, 10).map(i => (i, i)))
```

After running this command you can check with command-line Visor that cache size is 100000 elements.

1. We can now check how the state we created will survive job restart. Shut down the spark shell and repeat steps 1-3. You should again have an instance of Ignite context and RDD for "partitioned" cache. We can now check how many keys there are in our RDD which value is greater than 50000:

```
sharedRDD.filter(_.2 > 50000).count
```

Since we filled up cache with a sequence of number from 1 to 100000 inclusive, we should see **50000** as a result:

```
res0: Long = 50000
```

## Troubleshooting

- My Spark application or Spark shell hangs when I invoke any action on IgniteRDD

This will happen if you have created **IgniteContext** in client mode (which is default mode) and you do not have any Ignite server nodes started up. In this case Ignite client will wait until server nodes are started or fail after cluster join timeout has elapsed. You should start at least one Ignite server node when using **IgniteContext** in client mode.

- I am getting **java.lang.ClassNotFoundException org.apache.ignite.logger.java.JavaLoggerFileHandler** when using IgniteContext

This issue appears when you do not have any loggers included in classpath and Ignite tries to use standard Java logging. By default Spark loads all user jar files using separate class loader. Java logging framework, on the other hand, uses application class loader to initialize log handlers. To resolve this, you can either add **ignite-log4j** module to the list of the used jars so that Ignite would use Log4j as a logging subsystem, or alter default Spark classpath as described [Installation & Deployment](#)

# **Legal**

# Apache License

Apache License Version 2.0, January 2004 <http://www.apache.org/licenses/>

## TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

### 1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute

patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally

submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

## APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");  
you may not use this file except in compliance with the License.  
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software  
distributed under the License is distributed on an "AS IS" BASIS,  
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
See the License for the specific language governing permissions and  
limitations under the License.

## Copyright

Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements.  
See the NOTICE file distributed with this work for additional information regarding copyright  
ownership. The ASF licenses this file to you under the Apache License, Version 2.0 (the "License"); you  
may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is  
distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either  
express or implied. See the License for the specific language governing permissions and limitations  
under the License.