**Micro Services**

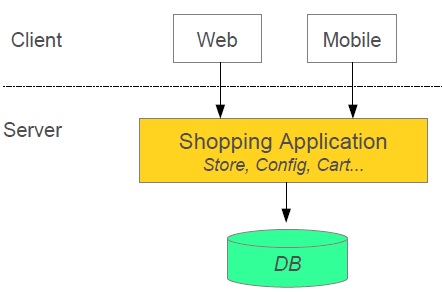
1. **What is Micro Services Architecture?**

Micro services architecture allows to avoid monolith application for large system. It provides loose coupling between collaborating processes which running independently in different environments with tight cohesion.

For example, imagine an online shop with separate micro services for user-accounts, product-catalog order-processing and shopping carts.

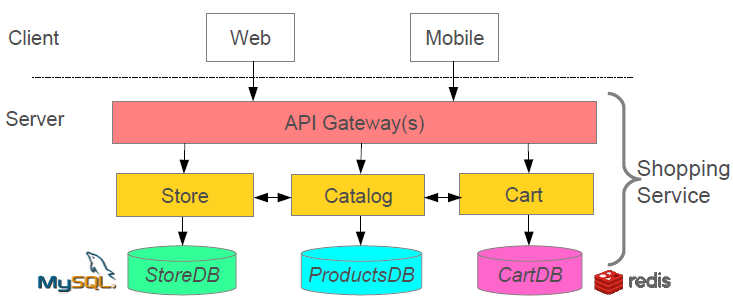
* **Shopping system without Micro services (Monolith architecture)**

In this architecture we are using Monolith architecture i.e. all collaborating components combine all in one application.



* **Shopping system with Micro services**

In this architecture style the main application divided in a set of sub applications called micro services. One large Application divided into multiple collaborating processes as below.



1. **Micro Services Benefits and Challenges?**

**Micro Services Benefits**

* Smaller code base is easy to maintain.
* Easy to scale as individual component.
* Technology diversity i.e. we can mix libraries, databases, frameworks etc.
* Fault isolation i.e. a process failure should not bring whole system down.
* Better support for smaller and parallel team.
* Independent deployment.
* Deployment reduce time.

**Micro Services Challenges**

* Difficult to achieve strong consistency across services
* ACID transactions do not span multiple processes.
* Distributed System so hard to debug and trace the issues
* Greater need for end to end testing
* Required cultural changes in across teams like Dev and Ops working together even in same team.

1. **Service Discovery / Registry**

**Implementing Service Discovery**

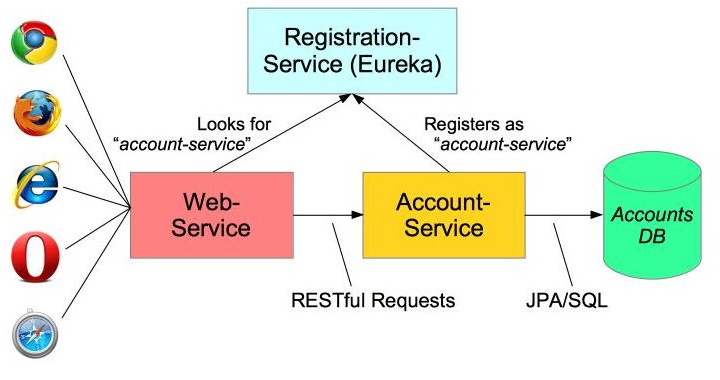
Use Eureka created by Netflix

**Client-side Load Balancing**

Each service typically deployed as multiple instances for fault tolerance and load sharing.

**Implementing Client-Side Load Balancing**

We will use Netflix Ribbon, it provide several algorithm for Client-Side Load Balancing. Spring provide smart **RestTemplate**for service discovery and load balancing by using ***@LoadBalanced***annotation with **RestTemplate**instance.

****

1. **Developing Simple Micro services Example**

**For build a simple micro services system following steps required**

1. Creating Discovery Service (Creating Eureka Discovery Service)
2. Creating Micro Service (the Producer)
   1. Register itself with Discovery Service with logical service.
3. Create Micro service Consumers find Service registered with Discovery Service
   1. Discovery client using a smart **RestTemplate**to find micro service.

**Maven Dependencies**

1. **<dependencies>**
2. **<dependency>**
3. **<groupId>**org.springframework.cloud**</groupId>**
4. **<artifactId>**spring-cloud-starter**</artifactId>**
5. **</dependency>**
6. **<dependency>**
7. **<groupId>**org.springframework.cloud**</groupId>**
8. **<artifactId>**spring-cloud-starter-eureka**</artifactId>**
9. **</dependency>**
10. **<dependency>**
11. **<groupId>**org.springframework.boot**</groupId>**
12. **<artifactId>**spring-boot-starter-web**</artifactId>**
13. **</dependency>**
14. **<dependency>**
15. **<groupId>**org.hsqldb**</groupId>**
16. **<artifactId>**hsqldb**</artifactId>**
17. **<scope>**runtime**</scope>**
18. **</dependency>**
19. **<dependency>**
20. **<groupId>**org.springframework.boot**</groupId>**
21. **<artifactId>**spring-boot-starter-test**</artifactId>**
22. **<scope>**test**</scope>**
23. **</dependency>**
24. **</dependencies>**

**Step 1: Creating Discovery Service (Creating Eureka Discovery Service)**

* Eureka Server using Spring Cloud
* We need to implement our own registry service as below.



**application.yml**

1. # Configure this Discovery Server
2. eureka:
3. instance:
4. hostname: localhost
5. client: #Not a client
6. registerWithEureka: **false**
7. fetchRegistry: **false**
8. # HTTP (Tomcat) port
9. server:
10. port: 8761

**or application.properties**

**eureka.client.register-with-eureka=false** [By Default , the registry will also attempt to register itself , so we’ll need to disable that.]

**eureka.client.fetch-registry=false**

**server.port=8761**

**DiscoveryMicroserviceServerApplication.java**

1. **package** com.doj.discovery;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
5. @SpringBootApplication
6. @EnableEurekaServer
7. **public** **class** DiscoveryMicroserviceServerApplication {
8. **public** **static** **void** main(String[] args) {
9. SpringApplication.run(DiscoveryMicroserviceServerApplication.**class**, args);
10. }
11. }

Spring Cloud’s @EnableEurekaServer to standup a registry that other applications can talk to. This is a regular Spring Boot application with one annotation added to enable the service registry.

**pom.xml**

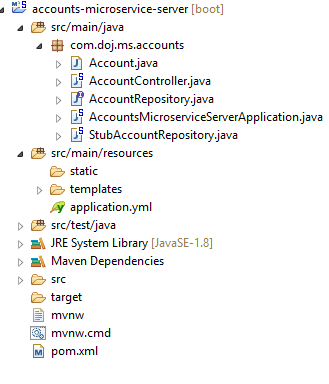
1. <!-- Eureka registration server -->
2. **<dependency>**
3. **<groupId>**org.springframework.cloud**</groupId>**
4. **<artifactId>**spring-cloud-starter-eureka-server**</artifactId>**
5. **</dependency>**

**Step 2: Creating Account Producer Micro Service**

Micro service declares itself as an available service and register to Discovery Server created in **Step 1**.

* Using ***@EnableDiscoveryClient***
* Registers using its application name

Let’s see the service producer application structure as below.



**application.yml**

1. ### Spring properties
2. # Service registers under this name
3. spring:
4. application:
5. name: accounts-microservice
6. # Discovery Server Access
7. eureka:
8. client:
9. serviceUrl:
10. defaultZone: http://localhost:8761/eureka/
11. # HTTP Server (Tomcat) Port
12. server:
13. port: 7777
14. # Disable Spring Boot's "Whitelabel" default error page, so we can use our own
15. error:
16. whitelabel:
17. enabled: **false**

or **application.properties**

server.port=7777

spring.application.name=accounts-microservice

#Eurkeka

eureka.client.serviceUrl.defaultZone= http://localhost:8761/eureka/

eureka.instance.preferIpAddress: false

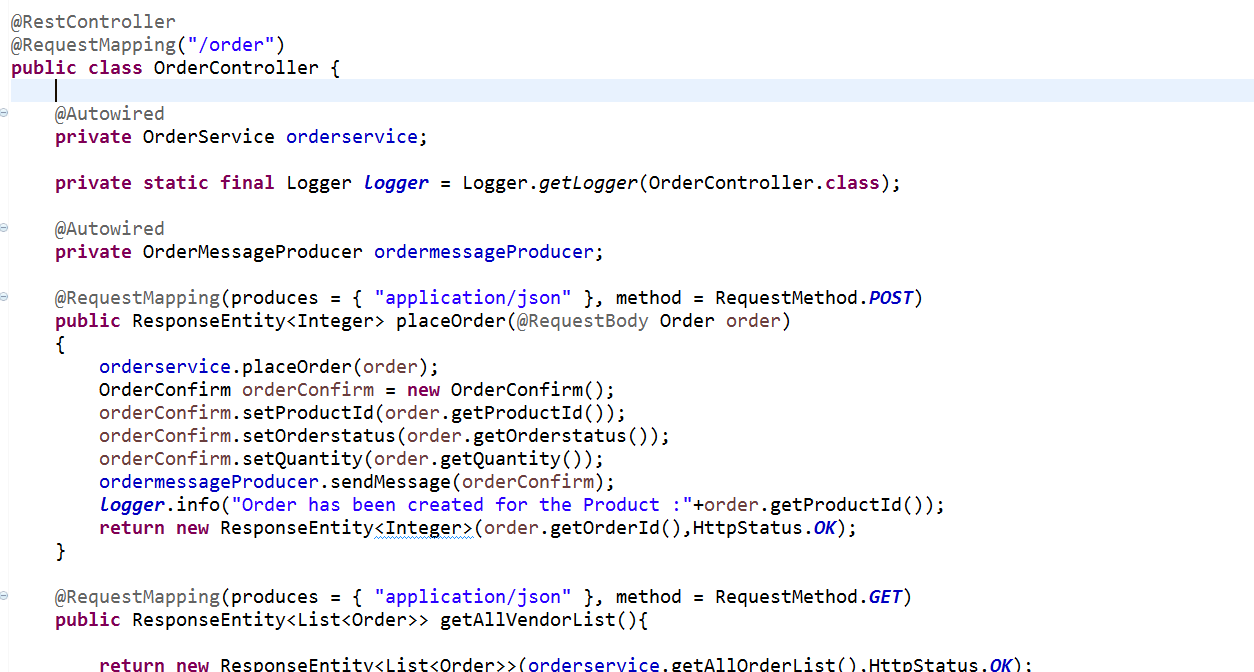
**AccountsMicroserviceServerApplication.java**

1. **package** com.doj.ms.accounts;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;
5. @SpringBootApplication
6. @EnableDiscoveryClient
7. **public** **class** AccountsMicroserviceServerApplication {
8. **public** **static** **void** main(String[] args) {
9. SpringApplication.run(AccountsMicroserviceServerApplication.**class**, args);
10. }
12. }

The @EnableDiscoveryClient activates the Netflix Eureka DiscoveryClient implementation.

**pom.xml**

1. **<dependencies>**
2. **<dependency>**
3. **<groupId>**org.springframework.cloud**</groupId>**
4. **<artifactId>**spring-cloud-starter**</artifactId>**
5. **</dependency>**
6. **<dependency>**
7. **<groupId>**org.springframework.cloud**</groupId>**
8. **<artifactId>**spring-cloud-starter-eureka**</artifactId>**
9. **</dependency>**
10. **<dependency>**
11. **<groupId>**org.springframework.boot**</groupId>**
12. **<artifactId>**spring-boot-starter-web**</artifactId>**
13. **</dependency>**
14. **</dependencies>**

****

**Using the @RestController Annotation**

Spring 4.0 introduced **@RestController**, a specialized version of the controller which is a convenience annotation that does nothing more than add the **@Controller** and **@ResponseBody** annotations. By annotating the controller class with **@RestController** annotation, you no longer need to add **@ResponseBody** to all the request mapping methods. The **@ResponseBody** annotation is active by default.

1. **What is Spring Boot?**

First of all Spring Boot is not a framework, it is a way to ease to create stand-alone application with minimal or zero configurations. It is approach to develop spring based application with very less configuration. It provides defaults for code and annotation configuration to quick start new spring projects within no time.

1. **How does it work? How does it know what to configure?**

Auto-configuration works by analyzing the classpath  
– If you forget a dependency, Spring Boot can't configure it  
– A dependency management tool is recommended  
– Spring Boot parent and starters make it much easier  
• Spring Boot works with Maven, Gradle, Ant/Ivy

1. **How are properties defined? Where?**

In spring boot, we have to define properties in the application.properties file exists in classpath of application as follow.  
Example: configure default DataSource bean

1. database.host=localhost
2. database.user=admin
3. **What embedded containers does Spring Boot support?**

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers.  
By default the embedded server will listen for HTTP requests on port 8080.

1. **What does @EnableAutoConfiguration do? What about @SpringBootApplication?**

**@EnableAutoConfiguration annotation** on a Spring Java configuration class

– Causes Spring Boot to automatically create beans it thinks you need  
– Usually based on classpath contents, can easily override

1. @Configuration
2. @EnableAutoConfiguration
3. **public** **class** MyAppConfig {
4. **public** **static** **void** main(String[] args) {
5. SpringApplication.run(MyAppConfig.**class**, args);
6. }
7. }

**@SpringBootApplication**was available from Spring Boot 1.2  
It is very common to use @EnableAutoConfiguration, @Configuration, and @ComponentScan together.

1. @Configuration
2. @ComponentScan
3. @EnableAutoConfiguration
4. **public** **class** MyAppConfig {
5. ...
6. }

**With @SpringBootApplication annotation**

1. @SpringBootApplication
2. **public** **class** MyAppConfig {
3. ...
4. }
5. **Can you control logging with Spring Boot? How?**

Yes, we can control logging with spring boot.  
 **Customizing default Configuration for Logging:**  
By adding **logback.xml** file to the application we can override the default logging configuration providing by the Spring Boot. This file place in the classpath (src/main/resources) of the application for Spring Boot to pick the custom configuration.  
  
**Spring Boot can control the logging level**  
– Just set it in application.properties  
• Works with most logging frameworks  
– Java Util Logging, Logback, Log4J, Log4J2

1. logging.level.org.springframework=DEBUG
2. logging.level.com.acme.your.code=INFO
3. **How to reload my changes on Spring Boot without having to restart server?**

Include following maven dependency in the application.

1. **<dependency>**
2. **<groupId>**org.springframework**</groupId>**
3. **<artifactId>**springloaded**</artifactId>**
4. **<version>**1.2.6.RELEASE**</version>**
5. **</dependency>**

**Automatic restart**

Applications that use spring-boot-devtools will automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a folder will be monitored for changes.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-devtools**</artifactId>**
4. **<optional>**true**</optional>**
5. **</dependency>**

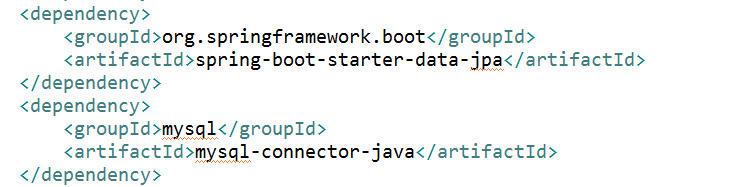
This can be achieved using DEV Tools. With this dependency any changes you save, the embedded tomcat will restart. Spring Boot has a Developer tools (DevTools) module which helps to improve the productivity of developers. One of the key challenge for the Java developers is to auto deploy the file changes to server and auto restart the server. Developers can reload changes on Spring Boot without having to restart my server. This will eliminates the need for manually deploying the changes every time. Spring Boot doesn’t have this feature when it has released it’s first version. This was a most requested features for the developers. The module DevTools does exactly what is needed for the developers. This module will be disabled in the production environment.

1. **How to implement security for spring boot application?**

Add spring security starter to the boot application

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-security**</artifactId>**
4. **</dependency>**
5. **How to configure datasource using Spring boot?**

* Use either spring-boot-starter-jdbc or **spring-boot-starterdata-jpa** and include a JDBC driver on classpath



• Declare properties in the application.properties

1. spring.datasource.url=jdbc:mysql://localhost/test
2. spring.datasource.username=dbuser
3. spring.datasource.password=dbpass
4. spring.datasource.driver-**class**-name=com.mysql.jdbc.Driver

**Spring Cache**

**Enable Caching**

1. **@EnableCaching** to enable caching because caching in spring is not enabled by default. The caching feature can be declaratively enabled by simply adding the **@EnableCaching** annotation to any of the configuration classes.
2. @SpringBootApplication
3. @EnableCaching
4. **public** **class** SpringBootCacheApplication {
5. **public** **static** **void** main(String[] args) {
6. SpringApplication.run(SpringBootCacheApplication.**class**, args);
7. }
8. }

In the XML we can you use following tag for enable caching.

1. **<cache:annotation-driven** **/>**

**Use Caching With Annotations**

After enabling the cache we can use following list of declarative annotations.

* @Cacheable
* @CacheEvict
* @CachePut
* @Caching
* @CacheConfig

**@Cacheable:**It is one of the most important and common annotation for caching the requests. If you annotate a method with @Cacheable, if multiple requests are received by the application, then this annotation will not execute the method multiple times, instead it will send the result from the cached storage.

The simplest way to enable caching behavior for a method is to demarcate it with @Cacheable and parameterize it with the name of the cache where the results would be stored:

1. @Cacheable(value="cities")
2. **public** List<City> findAllCity(){
3. **return** (List<City>) cityRepository.findAll();
4. }

The findAllCity() call will first checks the cache cities before actually invoking the method and then caching the result.

Spring framework also supports multiple caches to be passed as parameters:

1. @Cacheable(value={"cities","city-list"})
2. **public** List<City> findAllCity(){
3. **return** (List<City>) cityRepository.findAll();
4. }

**@CacheEvict:**

If we annotate all methods with**@Cacheable** then the size of cache may be some problem. **We don’t want to populate the cache with values that we don’t need often.** Caches can grow quite large, quite fast, and we could be holding on to a lot of stale or unused data. **@CacheEvict**annotation is used for removing a single cache or clearing the entire cache from the cache storage so that fresh values can be loaded into the cache again:

1. @CacheEvict(value="cities", allEntries=**true**)
2. **public** List<City> findAllCity(){
3. **return** (List<City>) cityRepository.findAll();
4. }

Here **allEntries** indicated whether all the data in the cache has to be removed.

**@CachePut:**

@CachePut annotation helps for updating the cache with the latest execution without stopping the method execution. The difference between @Cacheable and @CachePut is that @Cacheable will skip running the method, whereas @CachePut will actually run the method and then put its results in the cache.

1. @CachePut(value="cities")
2. **public** List<City> findAllCity(){
3. **return** (List<City>) cityRepository.findAll();
4. }

**@Caching:**

What if you want to use multiple annotations of the same type for caching a method? @Caching annotation used for grouping multiple annotations of the same type together when one annotation is not sufficient for the specifying the suitable condition. For example, you can put multiple @CacheEvict or @CachePut annotation inside @Caching to narrow down your conditions as you need.

1. @Caching(evict = {
2. @CacheEvict("cities"),
3. @CacheEvict(value="city-list", key="#city.name") })
4. **public** List<City> findAllCity(){
5. **return** (List<City>) cityRepository.findAll();
6. }

**@CacheConfig:**

You can annotate @CacheConfig at the class level to avoid repeated mentioning in each method. For example, in the class level you can provide the cache name and in the method you just annotate with @Cacheable annotation.

1. @CacheConfig(cacheNames={"cities"})
2. **public** **class** CityMasterService {
3. @Cacheable
4. **public** List<City>) findAllCity() {
5. **return** (List<City>)) cityRepository.findAll();
6. }
7. }

**Conditional Caching**

Sometimes we don’t want to cache some result there are no need to cache. For example – reusing our example from the @CachePut annotation – this will both execute the method as well as cache the results each and every time:

1. @CachePut(value="cities")
2. **public** List<City>) findAllCity(State state){
3. **return** (List<City>)) cityRepository.findAll(state.getStateCode());
4. }

**Condition Parameter:**

1. @CachePut(value="cities", condition="#state.stateName=='UP'")
2. **public** ListList<City>) findAllCity(State state){
3. **return** (List<City>)) cityRepository.findAll(state.getStateCode());
4. }

@CachePut can be parameterized with a condition parameter that takes a SpEL expression to ensure that the results are cached based on evaluating that expression.

**Unless Parameter:**

We can also control the caching based on the output of the method rather than the input – via the unless parameter

1. @CachePut(value="cities", **unless**="#result.length()>25")
2. **public** List<City>) findAllCity(String state){
3. **return** (List<City>)) cityRepository.findAll(state.getStateCode());
4. }

The above annotation would cache only those cities of state that have minimum 25 cities in the state.

**Agile**

* 1. **What are the differences between Agile and Traditional Project management?**

Agile encourages that a little of everything, including design, development and testing, is done at the same time, as opposed to the traditional approach to projects, where one phase is closed and completed before the next begins. Agile encourages short, frequent feedback loops and embraces changes to requirements. In [Waterfall](https://en.wikipedia.org/wiki/Waterfall_model), feedback is usually not collected until the very end of the project and changes are discouraged.

* 1. **What are the roles in Scrum?**

Scrum only prescribes 3 roles - **the Product Owner, Scrum Master and the Delivery team**. These roles should, ideally, be cross-functional and not shared among other projects.

Most Scrum Masters have not had the opportunity to work with a team that was cross-functional or dedicated due to the organization’s resistance or inability to allow for what some refer to as a “luxury”. This question sometimes leads them to ask how you would handle working with a team that did not have a designer or tester within the team or how you would handle a team that was not dedicated.  Be ready!

* 1. **What is the Daily Stand-Up?**

Every day, preferably in the morning, the team meets for no more than 15 minutes to answer three questions:

* + 1. What did you do yesterday?
    2. What do you plan on doing today?
    3. Are there any blocks or impediments that keep you from doing your work?

This Scrum ceremony is not meant to be a status meeting for stakeholders, but a way to energize the team and get them to set focus for the day.

* 1. **Describe what happens in the Sprint planning meeting.**

In Sprint planning, the Product Owner presents the goal of the sprint and discusses the high priority product backlog items. The Delivery team then chooses the amount of work for the next sprint.

* 1. **What is the role of the Scrum Master?**

A Scrum Master serves the team and shields them from any distractions that would prevent them from completing a sprint goal. They also remove blocks, teach the team to become self-organized, and serve as coach who teaches Agile and Scrum values and principles.

* 1. **Is there a difference between Agile and Scrum?**

Yes! Agile is the broader umbrella which Scrum falls under. [Agile has four main values and twelve principles](https://www.smartsheet.com/comprehensive-guide-values-principles-agile-manifesto). Scrum has its own set of values and principles and provides a lightweight “framework” to help teams becomeAgile.

* 1. **What are the four Main values and twelve principles of Agile?**

**The Four values of the Agile Manifesto**

* + 1. **Individuals and Interactions Over Processes and Tools**
    2. **Working Software Over Comprehensive Documentation**
       - Agile documents requirements as user stories, which are sufficient for a software developer to begin the task of building a new function.  
         The Agile Manifesto values documentation, but it values working software more.
    3. **Customer Collaboration Over Contract Negotiation**
       - Negotiation is the period when the customer and the product manager work out the details of a delivery, with points along the way where the details may be renegotiated. Collaboration is a different creature entirely. With development models such as Waterfall, customers negotiate the requirements for the product, often in great detail, prior to any work starting. This meant the customer was involved in the process of development before development began and after it was completed, but not during the process. The Agile Manifesto describes a customer who is engaged and collaborates throughout the development process, making. This makes it far easier for development to meet their needs of the customer. Agile methods may include the customer at intervals for periodic demos, but a project could just as easily have an end-user as a daily part of the team and attending all meetings, ensuring the product meets the business needs of the customer.
    4. **Responding to Change Over Following a Plan**
       - Traditional software development regarded change as an expense, so it was to be avoided. The intention was to develop detailed, elaborate plans, with a defined set of features and with everything, generally, having as high a priority as everything else, and with a large number of many dependencies on delivering in a certain order so that the team can work on the next piece of the puzzle.
       - With Agile, the shortness of an iteration means priorities can be shifted from iteration to iteration and new features can be added into the next iteration. Agile’s view is that changes always improve a project; changes provide additional value.

**The Twelve Agile Manifesto Principles**

1. **Customer satisfaction through early and continuous software delivery.**
2. **Accommodate changing requirements throughout the development process.**
3. **Frequent delivery of working software.**
4. **Collaboration between the business stakeholders and developers throughout the project.**
5. **Support, trust, and motivate the people involved.**
6. **Enable face-to-face interactions.**
7. **Working software is the primary measure of progress.**
8. **Agile processes to support a consistent development pace.**
9. **Attention to technical detail and design enhances agility.**
10. **Simplicity –**Develop just enough to get the job done for right now.
11. **Self-organizing teams encourage great architectures, requirements, and designs.**
12. **Regular reflections on how to become more effective.**
    1. **What is velocity?**

Velocity is the average number of points from that past 3-4 sprints and is used to help predict when backlog items will be delivered.

* 1. **What type of metrics or reports have you used?**

Sprint and release burn-down and burn-up charts are standard reports.

* 1. **What is a burn-down chart?**

A burn down chart displays the amount of work a team has burned through such as hours during the sprint.

A Scrum burn down chart should consist of

* + - * X-axis that displays working days
      * Y-axis that displays remaining effort
      * Ideal effort as guideline
      * Real Progress of effort
  1. **How many Scrum teams have you managed at one time?**

This is a popular question! Don’t offer that Scrum guidelines states only one Scrum Master per team as your answer! In this new role, you may be required to lead more than one team. Notice the use of the word “managed” versus “led” as Scrum Masters do not manage, they lead teams so be sure to use this word in your response.

* 1. **What type of requirements did you use for your teams?**

Requirements in Scrum are written as user stories using a standard, “As a , I want… so that I .”  As a Scrum Master, you don’t necessarily write user stories, but you would assist the Product Owner to ensure that user stories are written, prioritized and ready for the sprint.

* 1. **How much time should a person expect to spend on ScrumMaster activities?**

A ScrumMaster should make this role their top priority to focus on benefits of the overall team. Their load will vary from sprint to sprint depending on what impediments and issues the team is dealing with. Newly formed teams typically take more ScrumMaster time; 50%-100%, while experienced ScrumMasters with established well functioning teams might spend 50% or less time on the ScrumMaster role.

* 1. **What is difference between Epic, User stories & Tasks?**

**Epic** is a group of related user stories.  
**User Stories** define the actual business requirement. Generally created by the business owner.  
**Task:** To accomplish the business requirements, development team create tasks.

* 1. **How study Board can be defined in agile?**

A Story Board is a visual representation of a software project’s progress. There are generally four columns ‘To do’, In Progress’, ‘Test’, and ‘Done’. Different colored post, its notes are placed in each column indicating the progress of individual development items. A story board is typically used in agile development.

* 1. **How much time should a person expect to spend on ScrumMaster activities?**

A ScrumMaster should make this role their top priority to focus on benefits of the overall team. Their load will vary from sprint to sprint depending on what impediments and issues the team is dealing with. Newly formed teams typically take more ScrumMaster time; 50%-100%, while experienced ScrumMasters with established well functioning teams might spend 50% or less time on the ScrumMaster role.

* 1. **How the velocity of sprint is measured?**

If capacity is measured as a percentage of 40 hours weeks then completed  
= story points \* team capacity  
If capacity is measured in man hours then completed story points / team capacity.

* 1. **How to measure velocity if our iteration lengths change?**

You can’t measure it easily. Velocity’s value comes from its inherent consistency. A fixed iteration length helps drive the reliable rhythm of a project. Without this rhythm, you are constantly revising, re-estimating, and reconciling, and the ability to predict out in the future is minimized due to inconsistent results.  
If, on the other hand, almost everyone is going to be out a week for the holidays or a couple days for company-wide meetings then by all means adapt iteration dates or velocity accordingly. Like most agile practices, these are guidelines, not strict rules.

* 1. **List out what are the artifacts of Scrum process?**

Scrum process artifacts include

* + - * Sprint backlog
      * Product backlog
      * Velocity chart
      * Burn-down chart
  1. **Explain what is scrum of scrum?**

Scrum of scrum is used to refer the meeting after the daily scrum.  The responsible person from each team attends the meeting and discuss their work and answer the questions like

* + - * Since the last meeting, what is the progress of the team?
      * What your team is expected to do or should accomplish, before the next meeting?
      * What are the obstacles your team faced while completing the task?
      * Were you going to allot any of your work to the following team?
  1. **Explain when Scrum cannot be useful?**

Ideally scrum is useful to monitor work with 5 to 10 people, who are committed to achieving the sprint goal.  It does not go well with huge groups or team having more responsibilities.  For larger team, scrum can be applied by splitting the team into small groups and practice scrum.

* 1. **What is Spike?**

There may be some technical issues or design problem in the project which needs to be resolved first. To provide the solution of these problem “Spikes” are created. Spikes are of two types- Functional and Technical.

**Map Reduce and HDFS**

* 1. **How to compress mapper output but not the reducer output?**

With MR2, now we should set

conf.set("**mapreduce.map.output.compress**", true)

conf.set("**mapreduce.output.fileoutputformat.compress**", false)

By Default the property(mapreduce.map.output.compress) has been set as false.

* 1. **Explain what is distributed Cache in MapReduce Framework ?**

Distributed Cache is an important feature provided by map reduce framework. When you want to share some files across all nodes in Hadoop Cluster, DistributedCache  is used.  The files could be an executable jar files or simple properties file.

* 1. **Explain what is JobTracker in Hadoop? What are the actions followed by Hadoop?**

In Hadoop for submitting and tracking MapReduce jobs,  JobTracker is used. Job tracker run on its own JVM process .

Hadoop performs following actions in Hadoop

* + - * Client application submit jobs to the job tracker
      * JobTracker communicates to the Namemode to determine data location
      * Near the data or with available slots JobTracker locates TaskTracker nodes
      * On chosen TaskTracker Nodes, it submits the work
      * When a task fails, Job tracker notify and decides what to do then
      * The TaskTracker nodes are monitored by JobTracker
  1. **Explain what is Speculative Execution?**

It is the option for Hadoop to specify backup tasks if it detects that there are some slow tasks on a few of the cluster nodes.  The backup tasks will be preferentially scheduled on the faster nodes.  Whichever of the duplicate tasks finishes first becomes the one that is used in further operations.  
  
This is good to have on if you might have some slow nodes for whatever reason and there are available hardware resources to support additional tasks on the other nodes**.  Indeed it is on by default.**  However, you will want to turn it off if you are resource-limited.  You will also want to make sure it is off if you are copying any information into a distributed database to avoid duplicate entries.    
  
**To turn off speculative execution entirely, just set the values of the following to false:**

* **mapred.map.tasks.speculative.execution** (mapreduce.map.speculative in Hadoop 2.x)
* **mapred.reduce.tasks.speculative.execution** (mapreduce.reduce.speculative in Hadoop 2.x)

I additionally have a special case where I use mappers to write data to files outside of the standard output collection code.  If I need to append data to those files, speculative execution must be off or else I will get duplicate data in those files as well.

* 1. **Explain what is the function of MapReducer partitioner?**

The function of MapReducer partitioner is to make sure that all the value of a single key goes to the same reducer, eventually which helps evenly distribution of the map output over the reducers.

* 1. **Explain what is difference between an Input Split and HDFS Block?**

Logical division of data is known as Split while physical division of data is known as HDFS Block.

* 1. **Mention what are the main configuration parameters that user need to specify to run Mapreduce Job ?**

The user of Mapreduce framework needs to specify

* + - * Job’s input locations in the distributed file system
      * Job’s output location in the distributed file system
      * Input format
      * Output format
      * Class containing the map function
      * Class containing the reduce function
      * JAR file containing the mapper, reducer and driver classes
  1. **Explain what is sqoop in Hadoop ?**

To transfer the data between Relational database management (RDBMS) and Hadoop HDFS a tool is used known as Sqoop. Using Sqoop data can be transferred from RDMS like MySQL or Oracle into HDFS as well as exporting data from HDFS file to RDBMS.

* 1. **Explain how can you debug Hadoop code?**

The popular methods for debugging Hadoop code are:

* + - * By using web interface provided by Hadoop framework
      * By using Counters

# [**How to do I copy data from one HDFS to another HDFS?**](https://stackoverflow.com/questions/31862904/how-to-do-i-copy-data-from-one-hdfs-to-another-hdfs)

DistCp (distributed copy) is a tool used for copying data between clusters. It uses MapReduce to effect its distribution, error handling and recovery, and reporting. It expands a list of files and directories into input to map tasks, each of which will copy a partition of the files specified in the source list.

Usage: $ hadoop distcp <src> <dst>

example: $ hadoop distcp hdfs://nn1:8020/file1 hdfs://nn2:8020/file2

file1 from nn1 is copied to nn2 with filename file2

Distcp is the best tool as of now. Sqoop is used to copy data from relational database to HDFS and vice versa, but not between HDFS to HDFS.

If the clusters are running identical versions of hadoop, then the hdfs scheme is appropriate to use.

$ hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar

The data in /foo directory of namenode1 will be copied to /bar directory of namenode2. If the /bar directory does not exist, it will create it. Also we can mention multiple source paths.

Similar to rsync command, distcp command by default will skip the files that already exist. We can also use -overwrite option to overwrite the existing files in destination directory. The option -update will only update the files that have changed.

$ hadoop distcp -update hdfs://namenode1/foo hdfs://namenode2/bar/foo

distcp can also be implemented as a MapReduce job where the work of copying is done by the maps that run in parallel across the cluster. There will be no reducers.

If trying to copy data between two HDFS clusters that are running different versions, the copy will process will fail, since the RPC systems are incompatible. In that case we need to use the read-only HTTP based HFTP filesystems to read from the source. Here the job has to run on destination cluster.

$ hadoop distcp hftp://namenode1:50070/foo hdfs://namenode2/bar

* 1. **What do you mean by data locality?**

[***Data locality***](http://www.edureka.co/blog/mapreduce-tutorial/?utm_source=blog&utm_medium=content-link&utm_campaign=hadoop-interview-questions-mapreduce/#data_locality) talks about moving computation unit to data rather data to the computation unit. MapReduce framework achieves data locality by processing data locally i.e. processing of the data happens in the very node by Node Manager where data blocks are present.

* 1. **What happens when a DataNode fails during the write process?**

When a DataNode fails during the write process, a new replication pipeline that contains the other DataNodes opens up and the write process resumes from there until the file is closed. NameNode observes that one of the blocks is under-replicated and creates a new replica asynchronously.

* 1. **Explain the differences between a combiner and reducer. When is it suggested to use a combiner in a MapReduce job?**

Combiner can be considered as a mini reducer that performs local reduce task. It runs on the Map output and produces the output to reducers input. It is usually used for network optimization when the map generates greater number of outputs.

* + - * Unlike a reducer, the combiner has a constraint that the input or output key and value types must match the output types of the Mapper.
      * Combiners can operate only on a subset of keys and values i.e. combiners can be executed on functions that are commutative.
      * Combiner functions get their input from a single mapper whereas reducers can get data from multiple mappers as a result of partitioning.

Combiners are generally used to enhance the efficiency of a MapReduce program by aggregating the intermediate map output locally on specific mapper outputs. This helps reduce the volume of data that needs to be transferred to reducers. Reducer code can be used as a combiner, only if the operation performed is commutative. However, the execution of a combiner is not assured.

* 1. **Explain the process of spilling in MapReduce? Why the output of map tasks are stored (spilled) into local disc and not in HDFS?**

The output of a map task is written into a circular memory buffer (RAM). The default size of buffer is set to 100 MB which can be tuned by using **mapreduce.task.io.sort.mb** property. Now, spilling is a process of copying the data from memory buffer to disc when the content of the buffer reaches a certain threshold size. By default, a background thread starts spilling the contents from memory to disc after 80% of the buffer size is filled. Therefore, for a 100 MB size buffer the spilling will start after the content of the buffer reach a size of 80 MB.

**Note:** One can change this spilling threshold using **mapreduce.map.sort.spill.percent** which is set to 0.8 or 80% by default.

The outputs of map task are the intermediate key-value pairs which is then processed by reducer to produce the final aggregated result. Once a MapReduce job is completed, there is no need of the intermediate output produced by map tasks. Therefore, storing these intermediate output into HDFS and replicate it will create unnecessary overhead.

**HDFS**

* 1. **How the client communicates with Name node and Data node in HDFS?**

The communication mode for clients with name node and data node in HDFS is **SSH**.

**Secure Shell** (**SSH**) is a [cryptographic](https://en.wikipedia.org/wiki/Cryptography) [network protocol](https://en.wikipedia.org/wiki/Network_protocol) for operating network services securely over an unsecured network. SSH provides a [secure channel](https://en.wikipedia.org/wiki/Secure_channel) over an unsecured network in a [client-server](https://en.wikipedia.org/wiki/Client-server) architecture, connecting an [SSH client](https://en.wikipedia.org/wiki/SSH_client) application with an [SSH server](https://en.wikipedia.org/wiki/SSH_server).

## ****Explain the HDFS Architecture and list the various HDFS daemons in HDFS cluster?****

While listing various HDFS daemons, you should also talk about their roles in brief. Here is how you should answer this question:

Apache Hadoop [***HDFS Architecture***](http://www.edureka.co/blog/apache-hadoop-hdfs-architecture/?utm_source=blog&utm_medium=content-link&utm_campaign=hadoop-interview-questions-hdfs-2) follows a Master/Slave topology where a cluster comprises a single NameNode (Master node or daemon) and all the other nodes are DataNodes (Slave nodes or daemons).  Following daemon runs in HDFS cluster:

* + - * **NameNode:**It is the master daemon that maintains and manages the data block present in the DataNodes.
      * **DataNode:** DataNodes are the slave nodes in HDFS. Unlike NameNode, DataNode is a commodity hardware, that is responsible of storing the data as blocks.
      * **Secondary NameNode:**The Secondary NameNode works concurrently with the primary NameNode as a *helper daemon*. It performs checkpointing.

## ****Apache Hadoop HDFS**** ****Architecture****

## HDFS Topology - Apache Hadoop HDFS Architecture - Edureka

## **Apache HDFS** or **Hadoop Distributed File System** is a block-structured file system where each file is divided into blocks of a pre-determined size. These blocks are stored across a cluster of one or several machines. Apache Hadoop HDFS Architecture follows a Master/Slave Architecture, where a cluster comprises of a single NameNode (Master node) and all the other nodes are DataNodes (Slave nodes). HDFS can be deployed on a broad spectrum of machines that support Java. Though one can run several DataNodes on a single machine, but in the practical world, these DataNodes are spread across various machines.

## ****NameNode:****

## Apache Hadoop HDFS Architecture - Edureka

NameNode is the master node in the Apache Hadoop HDFS Architecture that maintains and manages the blocks present on the DataNodes (slave nodes). NameNode is a very highly available server that manages the File System Namespace and controls access to files by clients. The HDFS architecture is built in such a way that the user data never resides on the NameNode. The data resides on DataNodes only.

### **Functions of NameNode:**

* 1. It is the master daemon that maintains and manages the DataNodes (slave nodes)
  2. It records the metadata of all the files stored in the cluster, e.g. the location of blocks stored, the size of the files, permissions, hierarchy, etc. There are two files associated with the metadata:
     + - **FsImage:** It contains the complete state of the file system namespace since the start of the NameNode.
       - **EditLogs:** It contains all the recent modifications made to the file system with respect to the most recent FsImage.
  3. It records each change that takes place to the file system metadata. For example, if a file is deleted in HDFS, the NameNode will immediately record this in the EditLog.
  4. It regularly receives a Heartbeat and a block report from all the DataNodes in the cluster to ensure that the DataNodes are live.
  5. It keeps a record of all the blocks in HDFS and in which nodes these blocks are located.
  6. The NameNode is also responsible to take care of the **replication factor**of all the blocks.
  7. In **case of the DataNode failure**, the NameNode chooses new DataNodes for new replicas,balance disk usage and manages the communication traffic to the DataNodes.

## ****DataNode:****

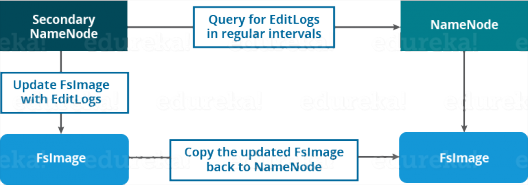
The DataNode is a block server that stores the data in the local file ext3 or ext4.

### **Functions of DataNode:**

* These are slave daemons or process which runs on each slave machine.
* The actual data is stored on DataNodes.
* The DataNodes perform the low-level read and write requests from the file system’s clients.
* They send heartbeats to the NameNode periodically to report the overall health of HDFS, by default, this frequency is set to 3 seconds.

## ****Secondary NameNode:****

Apart from these two daemons, there is a third daemon or a process called Secondary NameNode. The Secondary NameNode works concurrently with the primary NameNode as a **helper daemon.**And don’t be confused about the Secondary NameNode being a**backup NameNode because it is not.**



### **Functions of Secondary NameNode:**

* The Secondary NameNode is one which constantly reads all the file systems and metadata from the RAM of the NameNode and writes it into the hard disk or the file system.
* It is responsible for combining the EditLogs with FsImage from the NameNode.
* It downloads the EditLogs from the NameNode at regular intervals and applies to FsImage. The new FsImage is copied back to the NameNode, which is used whenever the NameNode is started the next time.

Hence, Secondary NameNode performs regular checkpoints in HDFS. Therefore, it is also called CheckpointNode.

## ****Blocks:****

Now, as we know that the data in HDFS is scattered across the DataNodes as blocks. **Let’s have a look at what is a block and how is it formed?**

Blocks are the nothing but the smallest continuous location on your hard drive where data is stored. In general, in any of the File System, you store the data as a collection of blocks. Similarly, HDFS stores each file as blocks which are scattered throughout the Apache Hadoop cluster. The default size of each block is 128 MB in Apache Hadoop 2.x (64 MB in Apache Hadoop 1.x) which you can configure as per your requirement.

HDFS ARCHITECTURE – HDFS TUTORIAL Introduction In this blog, we are going to talk about HDFS Architecture. From my previous blog, we already know that HDFS is a distributed file system which is deployed on low cost commodity hardware. I discussed many of its features too. So, its high time that we take a deep dive into Apache Hadoop HDFS Architecture and unlock its beauty. The topics that will be covered in this blog are as follows: • HDFS Master/Slave Topology • NameNode and DataNode • What is a block • Replication Management • Rack Awareness • HDFS Read/Write – Behind the scenes HDFS Architecture HDFS or Hadoop Distributed File System is a block-structured file system where each file is divided into blocks of a pre-determined size. These blocks are stored across a cluster of one or several machines. Apache Hadoop HDFS Architecture follows a Master/Slave Architecture, where a cluster comprises of a single NameNode (Master node) and all the other nodes are DataNodes (Slave nodes). HDFS is based on Java programming language, due to which HDFS can be deployed on broad spectrum of machines that support Java. Though one can run several DataNodes on a single machine, but in practical world, these DataNodes are spread across various machines. NameNode and DataNode NameNode: NameNode is the master of HDFS that maintains and manages the blocks present on the DataNodes (slave nodes). Think of the NameNode as a Lamborghini in midst of various other cars. Thus, like a Lamborghini, NameNode is a very highly available server that manages the File System Namespace and controls access to files by clients. I will be discussing this High Availability feature of Apache Hadoop HDFS in my next blog. The HDFS architecture is built in such a way that the user data is never stored in the NameNode. The data resides on DataNodes only. Functions of NameNode: • It is the master daemon that maintains and manages the DataNodes (slave nodes) • It records the metadata of all the files stored in the cluster, e.g. location of blocks stored, size of the files, permissions, hierarchy, etc. There are two files associated with metadata: o FsImage: An image of the file system on starting the NameNode. o EditLogs: A series of modifications done to the file system after starting the NameNode. • It records each change that takes place to the file system metadata. For example, if a file is deleted in HDFS, the NameNode will immediately record this in the EditLog. • It regularly receives a Heartbeat and a block report from all the DataNodes in the cluster to ensure that the DataNodes are live. • It keeps a record of all the blocks in HDFS and in which nodes these blocks are located. • The NameNode is also responsible to take care of the replication factor of all the blocks which we will discuss in detail later in this HDFS tutorial blog. • In case of a DataNode failure, the NameNode chooses new DataNodes for new replicas, balances disk usage and manages the communication traffic to the DataNodes. DataNode: DataNodes are the slave nodes in HDFS, just like any average car in front of a Lamborghini! Unlike NameNode, DataNode is a commodity hardware, that is, a non-expensive system which is not of high quality or high-availability. DataNode is a block server that stores the data in the local file ext3 or ext4. Functions of DataNode: • These are slave daemons or process which runs on each slave machine. • The actual data is stored on DataNodes. • DataNodes perform the low-level read and write requests from the file system’s clients. • They send heartbeats to the NameNode periodically to report the overall health of HDFS, by default, this frequency is set to 3 seconds. So, till now, you folks must have realized that the NameNode is pretty much important to us. If it fails, we are doomed. But don’t worry, we will be talking about how Hadoop solved this single point of failure problem in the next HDFS tutorial blog. So, just relax for now and let’s take one step at a time. Secondary NameNode Apart from these two daemons there is a third daemon or process called Secondary NameNode. The Secondary NameNode works concurrently with the primary NameNode as a helper daemon. And don’t confuse Secondary NameNode as a backup NameNode because it is not. Functions of Secondary NameNode: • The Secondary NameNode is one which constantly reads all the file systems and metadata from the RAM of the NameNode and writes it into the hard disk or the file system. • It is responsible for combining the EditLogs with FsImage from the NameNode. • It downloads the EditLogs from the NameNode at regular intervals and applies to FsImage. The new FsImage is copied back to the NameNode, which is used whenever the NameNode is started the next time. Hence, Secondary NameNode just performs regular checkpoints in HDFS. Therefore, it is also called CheckpointNode. Blocks Now as we know that the data in HDFS is scattered across the DataNodes as blocks. Let’s have a look on what is a block and how is it formed? Blocks are the nothing but smallest continuous location in your hard drive where data is stored. In general, in any of the File System the data are stored as collection of blocks. Similarly, HDFS stores each file as blocks which is scattered throughout the Apache Hadoop cluster. The default size of each block is 128MB in Apache Hadoop 2.x (64 MB in Apache Hadoop 1.x) which you can configure as per your requirement. It is not necessary that in HDFS, each file is stored in exact multiple of the configured block size (128MB, 256MB etc.). Let’s take an example where I have a file “example.txt” of size 514MB as shown in above figure. Suppose, we are using the default block size configuration which is 128Mb. So, how many blocks will be created? 5, Right. First four blocks will be of 128 MB. But, the last block will be of 2 MB size only. Now you must be thinking why we need to have such a huge blocks size i.e. 128MB? Well, whenever we talk about HDFS, we talk about huge data sets i.e. terabytes and petabytes of data. So, if we had a block size of let’s say 4KB as in Linux file system, we would be having too many of blocks and therefore too much of metadata. So, managing these no. of blocks and metadata will create huge overhead which is something, we don’t want. As you understood what a block is, lets understand how these blocks are places in the next section. Replication Management and Rack Awareness Replication Management: HDFS provides a reliable way to store huge data in a distributed environment as data blocks. The blocks are also replicated to provide fault tolerance. The default replication factor is 3 which is again configurable. So, if you want to store a file of 1GB in your HDFS, you will be consuming a space of 3GB (replication factor =3) and 24 (1GB/128MB=8 data blocks) data blocks in total, considering the default configuration. Don’t worry guys, if you didn’t get the math in one go. Take your time then move ahead. The NameNode collects block report from DataNode periodically to maintain the replication factor. Therefore, whenever a block is over-replicated or under-replicated the NameNode deletes or add replicas as needed. Rack Awareness: Anyways, moving ahead, let’s talk more about how replica are placed and what is rack awareness? Again, the NameNode also ensures that all the replicas are not stored on the same rack or a single rack. It follows an in-built Rack Awareness Algorithm to reduce latency. Considering the replication factor is 3, the Rack Awareness Algorithm says that the first replica of a block will be stored on a local rack and the next two replicas will be stored on a different (remote) rack but on a different DataNode within that (remote) rack. If you have more replicas, the rest of the replicas will be placed on random DataNodes provided not more than two replicas reside on the same rack, if possible. This is how an actual Hadoop production cluster looks like. Here, you have multiple racks populated with many DataNodes. Advantages of Rack Awareness: So now you will be thinking why do we need Rack Awareness algorithm? The reasons are: • To improve the network performance: The communication between nodes residing on different racks is directed via switch. In general, you will find greater network bandwidth between machines in the same rack than the machines residing in different rack. So, the Rack Awareness helps you to have reduce write traffic in between different racks and thus providing a better write performance. Also, you will be gaining increased read performance because you are using the bandwidth of multiple racks. • To prevent loss of data: We don’t have to worry about the data even if an entire rack fails because of the switch failure or power failure. And if you think about it, it will make sense, as it is said that never put all your eggs in the same basket. HDFS Read/ Write Architecture Now let’s talk about how the data read/write operations are performed on HDFS. HDFS follows Write Once - Read Many philosophy. So, you can’t edit files already stored in HDFS. But, you can append new data in a file. HDFS Write Architecture: Imagine a situation where a HDFS client, want to write a file named “example.txt” of size 248MB. Assume that the system block size is configured to 128 MB (default). So, the client will be dividing the file “example.txt” into 2 blocks – one of 128MB (Block A) and the other of 120 MB (block B). Now, the following protocol will be followed whenever the data is written into HDFS: • At first, the HDFS client will reach out to the NameNode for a Write Request against the two blocks, say, Block A & Block B. • The NameNode will then grant client the write permission and will provide the IP addresses of the DataNodes where the file blocks will be copied eventually. • The selection of IP addresses of DataNodes is purely randomized based on availability, replication factor and rack awareness that we have discussed earlier. • Let’s say the replication factor is set to default i.e. 3. Therefore, for each block the NameNode will be providing the client a list of (3) IP addresses of DataNodes. The list will be unique for each block. • Suppose, the NameNode provided following lists of IP addresses to the client: o For Block A, list A = {IP of DataNode 1, IP of DataNode 4, IP of DataNode 5} o For Block B, set B = {IP of DataNode 3, IP of DataNode 7, IP of DataNode 9} • Each block will be copied in three different DataNodes to maintain the replication factor consistent. • Now the whole data copy process will happen in three stages: 1. Set up of Pipeline 2. Data streaming and replication 3. Shutdown of Pipeline (Acknowledgement stage) 1. Set up of Pipeline: Before writing the blocks, client confirms whether the DataNodes present in each of the list of IPs are ready to receive the data or not. For doing so, the client creates a pipeline for each of the blocks by connecting the individual DataNodes in the respective list for that block. Let us consider Block A. The list of DataNodes provided by the NameNode is : For Block A, list A = {IP of DataNode 1, IP of DataNode 4, IP of DataNode 5}. So, for block A, the client will be performing following steps to create a pipeline:  Client will choose first DataNode in the list (DataNode IPs for Block A) which is DataNode 1 and will open a TCP/IP connection.  Client will inform DataNode 1 to be ready to receive the block. It will also provide the IPs of next two DataNodes (4 and 5) where the block will be replicated.  DataNode 1 will connect to DataNode 4. DataNode 1 will inform DataNode 4 to be ready to receive the block and will give it the IP of DataNode 5. Then, DataNode 4 will tell DataNode 5 to be ready for receiving the data.  Next, the acknowledgement of readiness will follow the reverse sequence i.e. From DataNode 5 to 4 and then to 1.  At last DataNode 1 will inform the client that all the DataNodes are ready and a pipeline is formed between client, DataNode 1, 4 and 5.  Now pipeline set up is complete and the client will finally begin the data copy or streaming process. 2. Data Streaming: As the pipeline has been created, the client will push the data into the pipeline. Now, don’t forget that in HDFS, data is replicated based on replication factor. So, here Block A will be stored to three DataNodes as the assumed replication factor is 3. Moving ahead, the client will copy the block (A) to DataNode 1 only. The replication is always done by DataNodes sequentially. So, following steps will take place during replication:  Once the block has been written to DataNode 1, it will connect to DataNode 4.  Then, it will push the block in the pipeline and data will be copied to DataNode 4.  Again, DataNode 4 will connect to DataNode 5 and will copy the block. 3. Shutdown of Pipeline or Acknowledgement stage: Once the block has been copied into all the three DataNodes, a series of acknowledgements will take place to ensure the client and NameNode that the data has been written successfully. And client will finally close the pipeline to end the TCP session. As shown in the figure, the acknowledgement happens in the reverse sequence i.e. from DataNode 5 to 4 and then to 1. Finally, the DataNode 1 will push three acknowledgements (including its own) into the pipeline and send it to client. Client will inform NameNode that data has been written successfully. NameNode will update its metadata and the client will shut down the pipeline. Similarly, Block B will also be copied into the DataNodes in parallel with Block A. But, following things are to be noticed here:  Client will copy Block A and Block B to the first DataNode simultaneously.  Therefore, in our case two pipelines will be formed for each of the block and all the process discussed above will happen in parallel in these two pipelines.  Client writes the block into first DataNode and then the DataNodes will be replicating the block sequentially. Folks! Its time for a quiz now: In HDFS, blocks of a file are written in parallel, however the replication of the blocks are done sequentially: a. True b. False True. Right guys. So, if you are still confused, go through the HDFS write architecture again. I am sure you will understand it this time. HDFS Read Architecture: HDFS Read architecture is comparatively easy to understand. Let’s the above example again where a HDFS clients want to read the file “example.txt” which is already there in HDFS. Now, following steps will be taking place while reading the file: • Client will reach out to NameNode asking for the block metadata for the file example.txt. • The NameNode will return the list of DataNodes where each block (Block A and B) are stored. • After that client, will connect to the DataNodes where the blocks are stored. • Client starts reading data parallel from the Data nodes (Block A from DataNode 1 and Block B from DataNode 5). • Once it gets all the required file blocks, it will combine these blocks to form a file. While serving read request of the client, HDFS selects the replica which is closest to the client. This reduces the read latency and the bandwidth consumption. Therefore, that replica is selected which resides on the same rack as the reader node, if possible. 

It is not necessary that in HDFS, each file is stored in exact multiple of the configured block size (128 MB, 256 MB etc.). Let’s take an example where I have a file “example.txt” of size 514 MB as shown in above figure.  Suppose that we are using the default configuration of block size, which is 128 MB. Then, how many blocks will be created? 5, Right. The first four blocks will be of 128 MB. But, the last block will be of 2 MB size only.

**Now, you must be thinking why we need to have such a huge blocks size i.e. 128 MB?**

Well, whenever we talk about HDFS, we talk about huge data sets, i.e. Terabytes and Petabytes of data. So, if we had a block size of let’s say of 4 KB, as in Linux file system, we would be having too many blocks and therefore too much of the metadata. So, managing these nos. of blocks and metadata will create huge overhead, which is something, we don’t want.

**Setting up Hadoop Cluster**

* 1. **Which are the modes in which Hadoop can run ?**

We have three modes in which Hadoop can run and that are:

* ***Standalone (local) mode***: Default mode of Hadoop, it uses the local file system for input and output operations. This mode is mainly used for debugging purpose, and it does not support the use of HDFS. Further, in this mode, there is no custom configuration required for **mapred-site.xml, core-site.xml, hdfs-site.xml** files. In stand-alone mode, there are no daemons, everything runs on a single JVM. It has no DFS and it utilizes the local file system. Stand-alone mode is suitable only for running MapReduce programs during development for testing. It is one of the most least used environments.
* ***Pseudo-distributed mode:*** In this case, you need configuration for all the three files mentioned above. In this case, all daemons are running on one node and thus, both Master and Slave nodes are on the same machine.
* ***Fully distributed mode:*** This is the production phase of Hadoop where data is distributed across several nodes on a Hadoop cluster. Separate nodes are allotted as Master and Slaves.
  + 1. **What is configured in /etc/hosts and what is its role in setting Hadoop cluster?**

/etc/hosts file contains the hostname and their IP address of that host. It maps the IP address to the hostname. In Hadoop cluster, we store all the hostnames (master and slaves) with their IP addresses in /etc/hosts so, that we can use hostnames easily instead of IP addresses.

## ****What are the default port numbers of NameNode, ResourceManager & MapReduce JobHistory Server?****

* + - * Namenode – ’50070’
      * ResourceManager – ’8088’
      * MapReduce JobHistory Server – ’19888’.

## ****What are the main Hadoop configuration files?****

4 main configuration files in Hadoop.

* **core-site.xml:**core-site.xml informs Hadoop daemon where NameNode runs on the cluster. It contains configuration settings of Hadoop core such as I/O settings that are common to HDFS & MapReduce.
* **hdfs-site.xml:** hdfs-site.xml contains configuration settings of HDFS daemons (i.e. NameNode, DataNode, Secondary NameNode). It also includes the replication factor and block size of HDFS.
* **mapred-site.xml**: mapred-site.xml contains configuration settings of the MapReduce framework like number of JVM that can run in parallel, the size of the mapper and the reducer, CPU cores available for a process, etc.
* **yarn-site.xml:**yarn-site.xml contains configuration settings of ResourceManager and NodeManager like application memory management size, the operation needed on program & algorithm, etc.

These files are in the **conf/hadoop/**directory inside Hadoop directory.

## ****5.What is command to extract the compressed file in tar.gz format?****

## tar -xvf /file\_location/filename.tar.gzcommand will extract the tar.gz compressed file.

## 6. ****What is the default replication factor and how will you change it?****

The default replication factor is 3.

♣ Tip: Default Replication Factor could be changed in three ways. Answering all the three ways will show your expertise.

1. By adding this property to **hdfs-site.xml**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <property>  <name>dfs.replication</name>  <value>5</value>  <description>Block Replication</description>  </property> |

1. Or you can change the replication factor on per file basis using following command:

**hadoop fs –setrep –w 3 /file\_location**

1. Or you can change replication factor for all the files in a directory using the following command:

**hadoop fs –setrep –w 3 -R /directory\_location**

## 7. What happens if you get a ‘connection refused java exception’ when you type hadoop fsck /?

If you get a ‘connection refused java exception’ when you type hadoop fsck, it could mean that the NameNode is not working on your VM.

## 8. How can we view the compressed files via HDFS command?

We can view compressed files in HDFS using **hadoop fs -text /filename**command.

## 9. ****What is the command to move into safe mode and exit safe mode?****

Safe Mode in Hadoop is a maintenance state of the NameNode during which NameNode doesn’t allow any changes to the file system. During Safe Mode, HDFS cluster is read-only and doesn’t replicate or delete blocks.

* To know the status of safe mode, you can use the command: ***hdfs dfsadmin -safemode get***
* To enter safe mode: ***hdfs dfsadmin -safemode enter***
* To exit safe mode: ***hdfs dfsadmin -safemode leave***

## 10. ****How can I restart Namenode?****

This question has two answers, answering both will give you a plus point. We can restart NameNode by following methods:

1. You can stop the NameNode individually using***.*** ***/sbin /hadoop-daemon.sh stop namenode*** command and then start the NameNode using***.*** ***/sbin/hadoop-daemon.sh start namenode***
2. Use .***/sbin/stop-all.sh*** and and then use .***/sbin/start-all.sh*** command which will stop all the daemons first and then start all the daemons.

## 11) What are the different commands used to startup and shutdown Hadoop daemons?

1. ***./sbin/start-all.sh*** to start all the Hadoop daemons and .***/sbin/stop-all.sh*** to stop all the Hadoop daemons.
2. Then you can start all the dfs daemons together using***.*** ***/sbin/start-dfs.sh***, yarn daemons together using***.*** ***/sbin/start-yarn.sh***and MR job history server using***.*** ***/sbin/mr-jobhistory-daemon.sh start historyserver***. To stop these daemons similarly we can use***.*** ***/sbin/start-yarn.sh***, ***./sbin/start-yarn.sh****&****.*** ***/sbin/mr-jobhistory-daemon.sh stop historyserver***.
3. The last way is to start all the daemons individually and stop them individually:

***./sbin/hadoop-daemon.sh start namenode***

***./sbin/hadoop-daemon.sh start datanode***

***./sbin/yarn-daemon.sh start resourcemanager***

***./sbin/yarn-daemon.sh start nodemanager***

***./sbin/mr-jobhistory-daemon.sh start historyserver***

and stop them similarly.

## 12) What does hadoop-metrics.properties file do?

## hadoop-metrics.properties is used for ‘Performance Reporting‘ purposes. It controls the reporting for Hadoop. The API is abstract so that it can be implemented on top of a variety of metrics client libraries. The choice of client library is a configuration option, and different modules within the same application can use different metrics implementation libraries. This file is stored inside /etc/hadoop.

## 13) What happens to a NameNode, when ResourceManager is down?

## When a ResourceManager is down, it will not be functional (for submitting jobs) but NameNode will be present. So, the cluster is accessible if NameNode is working, even if the ResourceManager is not working.

## 14) How can we format HDFS?

Hadoop distributed file system(HDFS) can be formatted using ***bin/hadoop namenode -format***command. This command formats the HDFS via NameNode. This command is only executed for the first time. Formatting the file system means initializing the directory specified by the **dfs.name.dir** variable. If you run this command on existing filesystem, you will lose all your data stored on your NameNode. Formatting a Namenode will not format the DataNode. It will format the **FsImage** and **edit logs** data stored on the NameNode and will lose the data about the location of blocks stored in HDFS.

Never format, up and running Hadoop filesystem. You will lose all your data stored in the HDFS.

### **15) What happens when two clients try to access the same file in the HDFS?**

HDFS supports exclusive writes only.

When the first client contacts the “NameNode” to open the file for writing, the “NameNode” grants a lease to the client to create this file. When the second client tries to open the same file for writing, the “NameNode” will notice that the lease for the file is already granted to another client, and will reject the open request for the second client.

### **16) State the reason why we can’t perform “aggregation” (addition) in mapper? Why do we need the “reducer” for this?**

This answer includes many points, so we will go through them sequentially.

* We cannot perform “aggregation” (addition) in mapper because sorting does not occur in the “mapper” function. Sorting occurs only on the reducer side and without sorting aggregation cannot be done.
* During “aggregation”, we need output of all the mapper functions which may not be possible to collect in the map phase as mappers may be running on different machine where the data blocks are stored.
* And lastly, if we try to aggregate data at mapper, it requires communication between all mapper functions which may be running on different machines. So, it will consume high network bandwidth and can cause network bottlenecking.

### 17) **How do “reducers” communicate with each other?**

The “MapReduce” programming model does not allow “reducers” to communicate with each other. “Reducers” run in isolation.

**Apache Hive**

## Apache Hive is a data warehouse system built on top of Hadoop and is used for analyzing structured and semi-structured data. Internally, these queries or HQL gets converted to map reduce jobs by the Hive compiler.

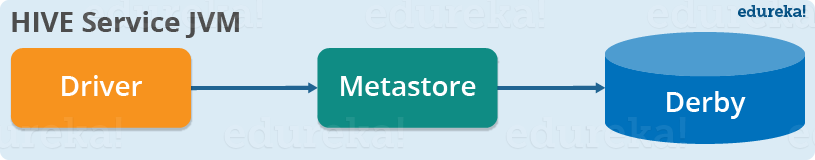
## ****Where does the data of a Hive table gets stored?****

## By default, the Hive table is stored in an HDFS directory – /user/hive/warehouse. One can change it by specifying the desired directory in hive.metastore.warehouse.dir configuration parameter present in the hive-site.xml.

## ****2) What is a metastore in Hive?****

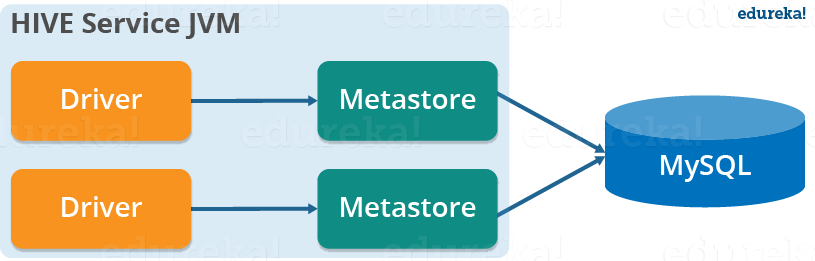
## Metastore stores the meta data information using RDBMS and an open source ORM (Object Relational Model) layer called Data Nucleus which converts the object representation into relational schema and vice versa. The reason for choosing RDBMS instead of HDFS is to achieve low latency. We can implement metastore in following three configurations:

### **Embedded Metastore:**



## Both the metastore service and the Hive service runs in the same JVM by default using an embedded Derby Database instance where metadata is stored in the local disk. This is called embedded metastore configuration. In this case, only one user can connect to metastore database at a time. If you start a second instance of Hive driver, you will get an error. This is good for unit testing, but not for the practical solutions.

### **Local Metastore:**



## This configuration allows us to have multiple Hive sessions i.e. Multiple users can use the metastore database at the same time. This is achieved by using any JDBC compliant database like MySQL which runs in a separate JVM or a different machine than that of the Hive service and metastore service which are running in the same JVM as shown above. In general, the most popular choice is to implement a MySQL server as the metastore database.

### **3. Remote Metastore:**

### Remote Mode Metastore - Hive Tutorial - Edureka

## In the remote metastore configuration, the metastore service runs on its own separate JVM and not in the Hive service JVM. Other processes communicate with the metastore server using Thrift Network APIs. You can have one or more metastore servers in this case to provide more availability. The main advantage of using remote metastore is you do not need to share JDBC login credential with each Hive user to access the metastore database.

## 3) ****What is the default database provided by Apache Hive for metastore?****

## By default, Hive provides an embedded Derby database instance backed by the local disk for the metastore. This is called the embedded metastore configuration.

## Suppose I have installed Apache Hive on top of my Hadoop cluster using default metastore configuration. Then, what will happen if we have multiple clients trying to access Hive at the same time?

The default metastore configuration allows only one Hive session to be opened at a time for accessing the metastore. Therefore, if multiple clients try to access the metastore at the same time, they will get an error. One has to use a standalone metastore, i.e. Local or remote metastore configuration in Apache Hive for allowing access to multiple clients concurrently.

Following are the steps to configure MySQL database as the local metastore in Apache Hive:

* One should make the following changes in **hive-site.xml**:
  + **javax.jdo.option.ConnectionURL** property should be set to jdbc:mysql://host/dbname?createDataba  
    seIfNotExist=true.
  + **javax.jdo.option.ConnectionDriverName** property should be set to **com.mysql.jdbc.Driver**.
  + One should also set the username and password as:
    - **javax.jdo.option.ConnectionUserName** is set to desired username.
    - **javax.jdo.option.ConnectionPassword** is set to the desired password.
* The JDBC driver JAR file for MySQL must be on the Hive’s classpath, i.e. The jar file should be copied into the Hive’s lib directory.
* Now, after restarting the Hive shell, it will automatically connect to the MySQL database which is running as a standalone metastore.

## ****Is it possible to change the default location of a managed table?****

## Yes, it is possible to change the default location of a managed table. It can be achieved by using the clause – LOCATION ‘<hdfs\_path>’.

## ****5)When should we use SORT BY instead of ORDER BY ?****

## We should use SORT BY instead of ORDER BY when we have to sort huge datasets because SORT BY clause sorts the data using multiple reducers whereas ORDER BY sorts all of the data together using a single reducer. Therefore, using ORDER BY against a large number of inputs will take a lot of time to execute.

## What is Tables, Partitions and Buckets?

## Hive’s data model provides a high-level, table-like structure on top of HDFS. It supports three data structures: tables, partitions, and buckets, where tables correspond to HDFS directories and can be divided into partitions, which in turn can be divided into buckets.

## What is the Temporary data file path?

## hive.exec.scratchdir: is the temporary data file path . By Deafult it is /tmp/hive-${user.name}.

## What are the commands from the Beeline to Hive.

## - Prepare the data as follows:

## -bash-4.1$ vi employee.txt

**Michael|Montreal,Toronto|Male,30|DB:80|Product:Developer^DLead**

**Will|Montreal|Male,35|Perl:85|Product:Lead,Test:Lead**

**Shelley|New York|Female,27|Python:80|Test:Lead,COE:Architect**

**Lucy|Vancouver|Female,57|Sales:89,HR:94|Sales:Lead**

- Log in to Beeline with the proper HiveServer2 hostname, port number, database name, username, and password:

**-bash-4.1$ beeline**

**beeline> !connect jdbc:hive2://localhost:10000/default**

**scan complete in 20ms Connecting to**

**jdbc:hive2://localhost:10000/default**

- Create a table using ARRAY, MAP, and STRUCT composite

data types ( It will create the Managed table/Internal Table)

**jdbc:hive2://> CREATE TABLE employee**

**. . . . . . .> (**

**. . . . . . .> name string,**

**. . . . . . .> work\_place ARRAY<string>,**

**. . . . . . .> sex\_age STRUCT<sex:string,age:int>,**

**. . . . . . .> skills\_score MAP<string,int>,**

**. . . . . . .> depart\_title MAP<string,ARRAY<string>>**

**. . . . . . .> )**

**. . . . . . .> ROW FORMAT DELIMITED**

**. . . . . . .> FIELDS TERMINATED BY '|'**

**. . . . . . .> COLLECTION ITEMS TERMINATED BY ','**

**. . . . . . .> MAP KEYS TERMINATED BY ':';**

**No rows affected (0.149 seconds)**

-Verify the table’s creation:

**jdbc:hive2://>!table employee**

**+---------+------------+------------+--------------+---------+**

**|TABLE\_CAT|TABLE\_SCHEMA| TABLE\_NAME | TABLE\_TYPE | REMARKS |**

**+---------+------------+------------+--------------+---------+**

**| |default | employee | MANAGED\_TABLE| |**

**+---------+------------+------------+--------------+---------+**

**jdbc:hive2://>!column employee**

**+--------------+-------------+---------------+---------------+**

**| TABLE\_SCHEM | TABLE\_NAME | COLUMN\_NAME | TYPE\_NAME |**

**+--------------+-------------+---------------+---------------+**

**| default | employee | name | STRING |**

**| default | employee | work\_place | array<string> |**

**| default | employee | sex\_age |**

**struct<sex:string,age:int>|**

**| default | employee | skills\_score | map<string,int>|**

**| default | employee | depart\_title | map<string,array<string>>**

**| +--------------+-------------+---------------+---------------+**

-Load data into the table: ( Load from the localpath)

**jdbc:hive2://>LOAD DATA LOCAL INPATH '/home/hadoop/employee.txt'**

**. . . . . . .>OVERWRITE INTO TABLE employee;**

**Hive Database**

Whenever a new database is created, Hive creates a directory for each database at **/user/hive/warehouse**, defined in **hive.metastore.warehouse.dir**. For example, the myhivebook database is located at **/user/hive/datawarehouse/myhivebook.db**.

**However, the default database doesn’t have its own directory.**

**Core DDL(Data Definition Language) for Hive Database.**

* Create the database without checking whether the database already exists:

**jdbc:hive2://> CREATE DATABASE myhivebook;**

* Create the database and check whether the database already exists:

**jdbc:hive2://> CREATE DATABASE IF NOT EXISTS myhivebook;**

* Create the database with location, comments, and metadata information:

**jdbc:hive2://> CREATE DATABASE IF NOT EXISTS myhivebook**

**. . . . . . .> COMMENT 'hive database demo'**

**. . . . . . .> LOCATION '/hdfs/directory'**

**. . . . . . .> WITH DBPROPERTIES ('creator'='dayongd','date'='2015-01-**

**01');**

* Drop the empty database:

**jdbc:hive2://> DROP DATABASE IF EXISTS myhivebook;**

Note that Hive keeps the database and the table in directory mode. In order to remove the parent directory, we need to remove the subdirectories first. By default, the database cannot be dropped if it is not empty, unless **CASCADE** is specified. **CASCADE** drops the tables in the database automatically before dropping the database.

Drop the database with **CASCADE**:

**jdbc:hive2://> DROP DATABASE IF EXISTS myhivebook CASCADE;**

* Alter the database properties.

The ALTER DATABASE statement can only apply to the table properties and roles (Hive 0.13.0 and later) on the table. The other metadata about the database cannot be changed:

**jdbc:hive2://> ALTER DATABASE myhivebook**

**. . . . . . .> SET DBPROPERTIES ('edited-by' = 'Dayong');**

**jdbc:hive2://> ALTER DATABASE myhivebook**

**SET OWNER user dayongd;**

**What is Hive Internal and External Tables?**

Each table associates with a directory configured in **${HIVE\_HOME}/conf/hive-site.xml** in HDFS. By default, it is **/user/hive/warehouse** in HDFS.

All the data in the table will be kept in the directory. The Hive table is also referred to as **internal or managed** tables.

When there is data already in HDFS, an external Hive table can be created to describe the data. It is called **EXTERNAL** because the data in the external table is specified in the **LOCATION** properties instead of the default warehouse directory. When keeping data in the internal tables, Hive fully manages the life cycle of the table and data. This means the data is removed once the internal table is dropped. If the external table is dropped, the table

metadata is deleted but the data is kept. Most of the time, an external table is preferred to avoid deleting data along with tables by mistake.

**DDLs for Hive internal and External table**

Create the internal table and load the data:

jdbc:hive2://> CREATE TABLE IF NOT EXISTS employee\_internal

. . . . . . .> (

. . . . . . .> name string,

. . . . . . .> work\_place ARRAY<string>,

. . . . . . .> sex\_age STRUCT<sex:string,age:int>,

. . . . . . .> skills\_score MAP<string,int>,

. . . . . . .> depart\_title MAP<STRING,ARRAY<STRING>>

. . . . . . .> )

. . . . . . .> COMMENT 'This is an internal table'

. . . . . . .> ROW FORMAT DELIMITED

. . . . . . .> FIELDS TERMINATED BY '|'

. . . . . . .> COLLECTION ITEMS TERMINATED BY ','

. . . . . . .> MAP KEYS TERMINATED BY ':'

. . . . . . .> STORED AS TEXTFILE;

No rows affected (0.149 seconds)

jdbc:hive2://> LOAD DATA LOCAL INPATH '/home/hadoop/employee.txt'

. . . . . . .> OVERWRITE INTO TABLE employee\_internal;

Create the external table and load the data:

jdbc:hive2://> CREATE EXTERNAL TABLE employee\_external

. . . . . . .> (

. . . . . . .> name string,

. . . . . . .> work\_place ARRAY<string>,

. . . . . . .> sex\_age STRUCT<sex:string,age:int>,

. . . . . . .> skills\_score MAP<string,int>,

. . . . . . .> depart\_title MAP<STRING,ARRAY<STRING>>

. . . . . . .> )

. . . . . . .> COMMENT 'This is an external table'

. . . . . . .> ROW FORMAT DELIMITED

. . . . . . .> FIELDS TERMINATED BY '|'

. . . . . . .> COLLECTION ITEMS TERMINATED BY ','

. . . . . . .> MAP KEYS TERMINATED BY ':'

. . . . . . .> STORED AS TEXTFILE

. . . . . . .> **LOCATION '/user/dayongd/employee';**

No rows affected (1.332 seconds)

jdbc:hive2://> LOAD DATA LOCAL INPATH '/home/hadoop/employee.txt'. . .

. . . .> OVERWRITE

INTO TABLE employee\_external;

**What will happen if folder does not exist in the Location Property.?**

If the folder in the path does not exist in the LOCATION property, Hive will create that folder. If there is another folder inside the folder specified in the LOCATION property, Hive will NOT report errors when creating the table, but will report an error when querying the table.

**What are the other format supported for STORE AS?**

For the STORE AS property, it is set to **AS TEXTFILE** by default. Other file format values, such as **SEQUENCEFILE**, **RCFILE**, **ORC**, **AVRO** (since Hive 0.14.0), and **PARQUET** (since Hive 0.13.0) can also be specified.

**What is the Create the table as Select (CTAS) and its restrictions.?**

jdbc:hive2://> CREATE TABLE ctas\_employee

. . . . . . .> AS SELECT \* FROM employee\_external;

No rows affected (1.562 seconds)

CTAS copies the data as well as table definitions. The table created by CTAS is atomic; this means that other users do not see the table until all the query results are populated.

CTAS has the following restrictions:

The table created cannot be a partitioned table

The table created cannot be an external table

The table created cannot be a list bucketing table

A CTAS statement will trigger a map job for populating the data;

**What will the Alter the table’s statement to rename the table.?**

jdbc:hive2://> ALTER TABLE cte\_employee RENAME TO c\_employee;

No rows affected (0.237 seconds)

**What will be the Alter the table’s file format statement?**

jdbc:hive2://> ALTER TABLE c\_employee SET FILEFORMAT RCFILE;

No rows affected (0.235 seconds)

**What will the Alter the table’s location statement?**

jdbc:hive2://> ALTER TABLE c\_employee

. . . . . . .> SET LOCATION

. . . . . . .> 'hdfs://localhost:8020/user/dayongd/employee';

No rows affected (0.169 seconds)

**What will be Alter the table’s enable/disable protection to NO\_DROP, which prevents a table from being dropped, or OFFLINE, which prevents data (not metadata) in a table from being queried?**

jdbc:hive2://> ALTER TABLE c\_employee ENABLE NO\_DROP;

jdbc:hive2://> ALTER TABLE c\_employee DISABLE NO\_DROP;

jdbc:hive2://> ALTER TABLE c\_employee ENABLE OFFLINE;

jdbc:hive2://> ALTER TABLE c\_employee DISABLE OFFLINE;

**What will be the Alter the table’s concatenation to merge small files into larger files.?**

**--Convert to the file format supported**

jdbc:hive2://> ALTER TABLE c\_employee SET FILEFORMAT ORC;

No rows affected (0.160 seconds)

**--Concatenate files**

jdbc:hive2://> ALTER TABLE c\_employee CONCATENATE;

No rows affected (0.165 seconds)

**--Convert to the regular file format**

jdbc:hive2://> ALTER TABLE c\_employee SET FILEFORMAT TEXTFILE;

No rows affected (0.143 seconds)

In Hive release 0.8.0, RCFile is added to support fast block-level merging of small RCFiles using the CONCATENATE command. In Hive release 0.14.0 ORC, the files that are added support fast stripe-level merging of small ORC files using the CONCATENATE command. Other file formats are not supported yet. In case of RCFiles, the merge

happens at block level and ORC files merge at stripe level thereby avoiding the overhead of decompressing and decoding the data. MapReduce is triggered when performing concatenation.

**What is Hive Buckets?**

Besides partition, bucket is another technique to cluster datasets into more manageable parts to optimize query performance. Different from partition, the bucket corresponds to segments of files in HDFS. For example, the employee\_partitioned table from the previous section uses the year and month as the top-level partition. If there is a further request to use the employee\_id as the third level of partition, it leads to many deep and small partitions and directories. For instance, we can bucket the employee\_partitioned table using employee\_id as the bucket column. The value of this column will be hashed by a user-defined number into buckets. The records with the same employee\_id will always be stored in the same bucket (segment of files). By using buckets, Hive can easily and efficiently do sampling and map side joins .

**--Create the buckets table**

**jdbc:hive2://> CREATE TABLE employee\_id\_buckets**

**. . . . . . .> (**

**. . . . . . .> name string,**

**. . . . . . .> employee\_id int,**

**. . . . . . .> work\_place ARRAY<string>,**

**. . . . . . .> sex\_age STRUCT<sex:string,age:int>,**

**. . . . . . .> skills\_score MAP<string,int>,**

**. . . . . . .> depart\_title MAP<string,ARRAY<string >>**

**. . . . . . .> )**

**. . . . . . .> CLUSTERED BY (employee\_id) INTO 2 BUCKETS**

**. . . . . . .> ROW FORMAT DELIMITED**

**. . . . . . .> FIELDS TERMINATED BY '|'**

**. . . . . . .> COLLECTION ITEMS TERMINATED BY ','**

**. . . . . . .> MAP KEYS TERMINATED BY ':';**

**No rows affected (0.104 seconds)**

Bucketing has close dependency on the underlying data loaded. To properly load data to a bucket table, we need to either set the maximum number of reducers to the same number of buckets specified in the table creation (for example, 2) or enable enforce bucketing as follows:

**jdbc:hive2://> set map.reduce.tasks = 2;**

**No rows affected (0.026 seconds)**

**jdbc:hive2://> set hive.enforce.bucketing = true;**

**No rows affected (0.002 seconds)**

To populate the data to the bucket table, we cannot use LOAD keywords such as what was done in the regular tables since LOAD does not verify the data against the metadata. Instead, INSERT should be used to populate the bucket table as follows:

**jdbc:hive2://> INSERT OVERWRITE TABLE employee\_id\_buckets**

**. . . . . . .> SELECT \* FROM employee\_id;**

**No rows affected (75.468 seconds)**

**--Verify the buckets in the HDFS**

**-bash-4.1$ hdfs dfs -ls /user/hive/warehouse/employee\_id\_buckets**

**Found 2 items**

**-rwxrwxrwx 1 hive hive 900 2014-11-02 10:54**

**/user/hive/warehouse/employee\_id\_buckets/000000\_0**

**-rwxrwxrwx 1 hive hive 582 2014-11-02 10:54**

**/user/hive/warehouse/employee\_id\_buckets/000001\_0**

**What is hive.fetch.task.conversion = more property ?**

SELECT \* scans the whole table/file without triggering MapReduce jobs, so it runs faster than SELECT <column\_name>. Since Hive 0.10.0, the simple SELECT statements, such as SELECT <column\_name> FROM

<table\_name> LIMIT n, can also avoid triggering the MapReduce job if the Hive fetch task conversion is enabled by setting hive.fetch.task.conversion = more.

**What is “*Correlating expression cannot contain unqualified column references”* error in Hive?**

If the alias (see the following example for the employee table) is not specified before columns (name) in the WHERE condition, Hive will report the error Correlating expression cannot contain unqualified column references. This is a limitation of the Hive subquery.

**jdbc:hive2://> SELECT name, sex\_age.sex AS sex**

**. . . . . . .> FROM employee a**

**. . . . . . .> WHERE a.name IN**

**. . . . . . .> (SELECT name FROM employee**

**. . . . . . .> WHERE sex\_age.sex = 'Male'**

**. . . . . . .> );**

**How to skip header while loading the csv file in the Hive table ?**

CREATE TABLE `warehouse`(`w\_warehouse\_sk` int,

`w\_warehouse\_id` string,

`w\_warehouse\_name` string,

`w\_warehouse\_sq\_ft` int,

`w\_street\_number` string,

`w\_street\_name` string,

`w\_street\_type` string,

`w\_suite\_number` string,

`w\_city` string,

`w\_county` string,

`w\_state` string,

`w\_zip` string,

`w\_country` string,

`w\_gmt\_offset` decimal(5,2)) row format delimited fields terminated by ',' stored as textfile **tblproperties("skip.header.line.count"="1")** ;

LOAD DATA Local INPATH '/home/cloud/data.csv' OVERWRITE INTO TABLE warehouse;

**How to connect to hive beeline?**

Beeline>!connect jdbc:hive2://192.168.145.57:10000 root Impetus@123# org.apache.hive.jdbc.HiveDriver

**How to execute a script file in Hive?**

beeline -u jdbc:hive2://192.168.145.57:10000/ndw\_base\_views -n root -p Impetus@123# -d org.apache.hive.jdbc.HiveDriver -f /syadav/Comcast/DDL/all\_prep\_ddl.hql

**How to extract Hive DDL?**

1. hive -e "use <databaseName>; show tables;" > all\_tables.txt

2. new shell script :

#!/bin/bash

cat all\_tables.txt |while read LINE

do

hive -e " use <databaseName>; show create table $LINE" >>tablesDDL.txt

done

**Why hive only supports Equal Join instead of unequal Join?**

Hive only supports equal JOIN instead of unequal JOIN, because unequal JOIN is difficult to be

converted to MapReduce jobs.

**Can JOIN operation be performed among more than two tables ?**

The JOIN operation can be performed among more tables (three tables in this case), as follows:

**jdbc:hive2://> SELECT emp.name, empi.employee\_id, emph.sin\_number**

**. . . . . . .> FROM employee emp**

**. . . . . . .> JOIN employee\_hr emph ON emp.name = emph.name**

**. . . . . . .> JOIN employee\_id empi ON emp.name = empi.name;**

**What is Self-Join?**

Self-join is a special JOIN where one table joins itself. When doing such joins, a different alias should be given to distinguish the same table:

**jdbc:hive2://> SELECT emp.name**

**. . . . . . .> FROM employee emp**

**. . . . . . .> JOIN employee emp\_b**

**. . . . . . .> ON emp.name = emp\_b.name;**

**Why it is suggested for JOIN statements to put the big table at the end.?**

It is suggested for JOIN statements to put the big table right at the end for better performance as well as

avoiding **Out Of Memory** (**OOM**) exceptions, because the last table in the sequence is streamed through the reducers where the others are buffered in the reducer by default.

Also, a hint, such as /\*+STREAMTABLE (table\_name)\*/, can be specified to tell which table is streamed as follows:

**jdbc:hive2://> SELECT /\*+ STREAMTABLE(employee\_hr) \*/**

**. . . . . . .> emp.name, empi.employee\_id, emph.sin\_number**

**. . . . . . .> FROM employee emp**

**. . . . . . .> JOIN employee\_hr emph ON emp.name = emph.name**

**. . . . . . .> JOIN employee\_id empi ON emph.employee\_id =**

**empi.employee\_id;**

**What is MAPJOIN ?**

The MAPJOIN statement means doing the JOIN operation only by map without the reduce job. The MAPJOIN statement reads all the data from the small table to memory and broadcasts to all maps. During the map phase, the JOIN operation is performed by comparing each row of data in the big table with small tables against the join conditions. Because there is no reduce needed, the JOIN performance is improved. When the

**hive.auto.convert.join** setting is set to true, Hive automatically converts the JOIN to MAPJOIN at runtime if possible instead of checking the map join hint.

**jdbc:hive2://> SELECT /\*+ MAPJOIN(employee) \*/ emp.name, emph.sin\_number**

**. . . . . . .> FROM employee emp**

**. . . . . . .> CROSS JOIN employee\_hr emph WHERE emp.name <> emph.name;**

**What is LOAD command in Hive?**

To *move* data in Hive, it uses the LOAD keyword. Move here means the original data is moved to the target table/partition and does not exist in the original place anymore.

The LOCAL keyword specifies where the files are located in the host. If the LOCAL keyword is not specified, the files are loaded from the full **Uniform Resource Identifier** (**URI**) specified after INPATH or the value from the **fs.default.name** Hive property by default. The OVERWRITE keyword is used to decide whether to append or

replace the existing data in the target table/partition.

**The examples to load files into Hive tables:**

**Load local data to the Hive table:**

**jdbc:hive2://> LOAD DATA LOCAL INPATH**

**. . . . . . .> '/home/dayongd/Downloads/employee\_hr.txt'**

**. . . . . . .> OVERWRITE INTO TABLE employee\_hr;**

**No rows affected (0.436 seconds)**

**Load local data to the Hive partition table:**

**jdbc:hive2://> LOAD DATA LOCAL INPATH**

**. . . . . . .> '/home/dayongd/Downloads/employee.txt'**

**. . . . . . .> OVERWRITE INTO TABLE employee\_partitioned**

**. . . . . . .> PARTITION (year=2014, month=12);**

**No rows affected (0.772 seconds)**

**Load HDFS data to the Hive table using the default system path:**

**jdbc:hive2://> LOAD DATA INPATH**

**. . . . . . .> '/user/dayongd/employee/employee.txt'**

**. . . . . . .> OVERWRITE INTO TABLE employee;**

**No rows affected (0.453 seconds)**

**Load HDFS data to the Hive table with full URI:**

**jdbc:hive2://> LOAD DATA INPATH**

**. . . . . . .>**

**'hdfs://[dfs\_host]:8020/user/dayongd/employee/employee.txt'**

**. . . . . . .> OVERWRITE INTO TABLE employee;**

**No rows affected (0.297 seconds)**

**What is Hive Partitions?**

By default, a simple query in Hive scans the whole Hive table. This slows down the performance when querying a large-size table. The issue could be resolved by creating Hive partitions, which is very similar to what’s in the RDBMS. In Hive, each partition corresponds to a predefined partition column(s) and stores it as a subdirectory in the table’s directory in HDFS. When the table gets queried, only the required partitions (directory) of data in the table are queried, so the I/O and time of query is greatly reduced.

**Syntax**

CREATE [EXTERNAL] TABLE table\_name (col\_name\_1 data\_type\_1, ....)

PARTITIONED BY (col\_name\_n data\_type\_n [COMMENT col\_comment], ...);

**What are the advantages of Partitioning?**

* Partitioning is used for distributing execution load horizontally.
* As the data is stored as slices/parts, query response time is faster to process the small part of the data instead of looking for a search in the entire data set.
* For example, In a large user table where the table is partitioned by country, then selecting users of country ‘IN’ will just scan one directory ‘country=IN’ instead of all the directories.

**What are the Limitations of Partitioning?**

* Having too many partitions in table creates large number of files and directories in HDFS, which is an overhead to NameNode since it must keep all metadata for the file system in memory only.
* Partitions may optimize some queries based on *Where clauses*, but may be less responsive for other important queries on *grouping clauses*.
* In Mapreduce processing, Huge number of partitions will lead to huge no of tasks (which will run in separate JVM) in each mapreduce job, thus creates lot of overhead in maintaining JVM start up and tear down. For small files, a separate task will be used for each file. In worst scenarios, the overhead of JVM start up and tear down can exceed the actual processing time.

**What are the ways to create a partitioned table for the use case where Customer/user details are partitioned by country/state or department for fast retrieval of subset data pertaining to some category?**

**Sample User records file**

**first\_name,last\_name,address,country,city,state,post,phone1,phone2,email,web**

Rebbecca,Didio,171 E 24th St,AU,Leith,TA,7315,03-8174-9123,0458-665-290,rebbecca.didio@didio.com.au,http://www.brandtjonathanfesq.com.au

###### Observation of Input Data

Input data has below fields or columns.

* First Name
* Last Name
* Address
* Country
* City
* State
* Postal Code
* Phone Number
* Alternative Phone Number
* Email Id
* Website URL

#### **Creation of Partition Table**

CREATE TABLE partitioned\_user(

firstname VARCHAR(64),

lastname  VARCHAR(64),

address   STRING,

city   VARCHAR(64),

post      STRING,

phone1    VARCHAR(64),

phone2    STRING,

email     STRING,

web       STRING

)

PARTITIONED BY (country VARCHAR(64), state VARCHAR(64))

STORED AS SEQUENCEFILE;

Note that we didn’t include **country** and **state** columns in table definition but included in partition definition.

hive> DESCRIBE FORMATTED partitioned\_user;

##### **External Partitioned Tables**

We can create external partitioned tables as well, just by using the **EXTERNAL** keyword in the **CREATE** statement, but for creation of External Partitioned Tables, we do not need to mention LOCATION clause as we will mention locations of each partitions separately while inserting data into table.

### Inserting Data Into Partitioned Tables

Data insertion into partitioned tables can be done in two modes.

* Static Partitioning
* Dynamic Partitioning

### **Static Partitioning in Hive**

In this mode, input data should contain the columns listed only in table definition (for example, firstname, lastname, address, city, post, phone1, phone2, email and web) but not the columns defined in partitioned by clause (country and state).

#### Loading Data into Managed Partitioned Table From Local FS

**hive> LOAD DATA LOCAL INPATH '${env:HOME}/staticinput.txt'**

**INTO TABLE partitioned\_user**

**PARTITION (country = 'US', state = 'CA');**

This will create separate directory under the default warehouse directory in HDFS.

**/user/hive/warehouse/partitioned\_user/country=US/state=CA/**

Similarly we have to add other partitions, which will create corresponding directories in HDFS. Or else we can load the entire directory into Hive table with single command and can add partitions for each file with [**ALTER**](http://hadooptutorial.info/partitioning-and-bucketing-in-hive/#alter) command.

**hive> LOAD DATA LOCAL INPATH '${env:HOME}/inputdir'**

**INTO TABLE partitioned\_user;**

**Loading Partition From Other Table**

We can load or add partitions with query results from another table as shown below.

**hive> INSERT OVERWRITE TABLE partitioned\_user**

**PARTITION (country = 'US', state = 'AL')**

**SELECT \* FROM another\_user au**

**WHERE au.country = 'US' AND au.state = 'AL';**

##### Loading Data into External Partitioned Table From HDFS

There is alternative for bulk loading of partitions into hive table. As data is already present in HDFS and should be made accessible by Hive, we will just mention the locations of the HDFS files for each partition.

If our files are on Local FS, they can be moved to a directory in HDFS and we can add partition for each file in that directory with commands similar to below.

**hive> ALTER TABLE partitioned\_user ADD PARTITION (country = 'US', state = 'CA')**

**LOCATION '/hive/external/tables/user/country=us/state=ca'**

Similarly we need to repeat the above alter command for all partition files in the directory so that a meta data entry will be created in metastore, mapping the partition and table.

### Dynamic Partitioning in Hive

Instead of loading each partition with single SQL statement as shown above, which will result in writing lot of SQL statements for huge no of partitions, Hive supports dynamic partitioning with which we can add any number of partitions with single SQL execution. Hive will automatically splits our data into separate partition files based on the values of partition keys present in the input files.

It gives the advantages of easy coding and no need of manual identification of partitions.

For dynamic partition loading we will not provide the values for partition keys, as shown below for previously seen query.

**hive>INSERT INTO TABLE partitioned\_user**

**PARTITION (country, state)**

**SELECT  firstname ,**

**lastname  ,**

**address   ,**

**city      ,**

**post      ,**

**phone1    ,**

**phone2    ,**

**email     ,**

**web       ,**

**country   ,**

**state**

**FROM temp\_user;**

But by default, Dynamic Partitioning is disabled in Hive to prevent accidental partition creations. To use dynamic partitioning we need to set below properties either in **Hive Shell** or in **hive-site.xml** file.

**Hive-site.xml**

<property>

    <name>**hive.exec.dynamic.partition**</name>

    <value>true</value>

    <description>Whether or not to allow dynamic partitions in DML/DDL.</description>

  </property>

  <property>

    <name>**hive.exec.dynamic.partition.mode**</name>

    <value>nonstrict</value>

    <description>

      In strict mode, the user must specify at least one static partition

      in case the user accidentally overwrites all partitions.

      In nonstrict mode all partitions are allowed to be dynamic.

    </description>

  </property>

  <property>

    <name>**hive.exec.max.dynamic.partitions**</name>

    <value>1000</value>

    <description>Maximum number of dynamic partitions allowed to be created in total.</description>

  </property>

  <property>

    <name>**hive.exec.max.dynamic.partitions.pernode**</name>

    <value>1000</value>

    <description>Maximum number of dynamic partitions allowed to be created in each mapper/reducer node.</description>

  </property>

  <property>

We can set these through hive shell with below commands,

set hive.exec.dynamic.partition=true;

set hive.exec.dynamic.partition.mode=nonstrict;

set hive.exec.max.dynamic.partitions=1000;

set hive.exec.max.dynamic.partitions.pernode=1000;

**Above Example use case for the dynamic partitioned**

As this input file contains, partitions keys also as fields in each record, we need to create **temporary user with the all the columns present in the input file** and from that we need to extract the columns needed into partition table by keeping country and state columns as partition keys.

We can create this temporary table and partitioned table and also load partitioned table dynamically with the help of below HiveQL.

**set hive.exec.dynamic.partition=true;**

**set hive.exec.dynamic.partition.mode=nonstrict;**

**set hive.exec.max.dynamic.partitions.pernode=1000;**

**DROP TABLE IF EXISTS partitioned\_user;**

**CREATE TEMPORARY TABLE temp\_user(**

**firstname VARCHAR(64),**

**lastname  VARCHAR(64),**

**address   STRING,**

**country   VARCHAR(64),**

**city      VARCHAR(64),**

**state     VARCHAR(64),**

**post      STRING,**

**phone1    VARCHAR(64),**

**phone2    STRING,**

**email     STRING,**

**web       STRING**

**)**

**ROW FORMAT DELIMITED**

**FIELDS TERMINATED BY ','**

**LINES TERMINATED BY '\n'**

**STORED AS TEXTFILE;**

**LOAD DATA LOCAL INPATH '/home/siva/UserRecords.txt' INTO TABLE temp\_user;**

**SELECT firstname, phone1, city**

**FROM temp\_user**

**WHERE country='US' AND state='CA'**

**ORDER BY city**

**LIMIT 5;**

**CREATE TABLE partitioned\_user(**

**firstname VARCHAR(64),**

**lastname  VARCHAR(64),**

**address   STRING,**

**city   VARCHAR(64),**

**post      STRING,**

**phone1    VARCHAR(64),**

**phone2    STRING,**

**email     STRING,**

**web       STRING**

**)**

**PARTITIONED BY (country VARCHAR(64), state VARCHAR(64))**

**STORED AS SEQUENCEFILE;**

**INSERT INTO TABLE partitioned\_user**

**PARTITION (country, state)**

**SELECT  firstname ,**

**lastname  ,**

**address   ,**

**city      ,**

**post      ,**

**phone1    ,**

**phone2    ,**

**email     ,**

**web       ,**

**country   ,**

**state**

**FROM temp\_user;**

**SELECT firstname, phone1, city**

**FROM partitioned\_user**

**WHERE country='US' AND state='CA'**

**ORDER BY city**

**LIMIT 5;**

**When partitioning is not effective?**

Partitioning gives effective results when,

* There are limited number of partitions
* Comparatively equal sized partitions

But this may not possible in all scenarios, like when are partitioning our tables based geographic locations like country, some bigger countries will have large partitions (ex: 4-5 countries itself contributing 70-80% of total data) where as small countries data will create small partitions (remaining all countries in the world may contribute to just 20-30 % of total data). So, In these cases Partitioning will not be ideal.

**What is IMPORT and EXPORT in HIVE?**

When working with Hive, sometimes we need to migrate data among different environments. Or we may need to back up some data. Since Hive 0.8.0, EXPORT and IMPORT statements are available to support the import and export of data in HDFS for data migration or backup/restore purposes.

The EXPORT statement will export both data and metadata from a table or partition.

Metadata is exported in a file called **\_metadata**. Data is exported in a subdirectory called **data**:

**jdbc:hive2://> EXPORT TABLE employee TO '/user/dayongd/output3';**

**No rows affected (0.19 seconds)**

After EXPORT, we can manually copy the exported files to other Hive instances or use Hadoop **distcp** commands to copy to other HDFS clusters. Then, we can import the data in the following manner:

**Import data to a table with the same name. It throws an error if the table exists:**

jdbc:hive2://> IMPORT FROM '/user/dayongd/output3';

Error: Error while compiling statement: FAILED: SemanticException

[Error 10119]: Table exists and contains data files

(state=42000,code=10119)

**Import data to a new table:**

jdbc:hive2://> IMPORT TABLE empolyee\_imported FROM

. . . . . . .> '/user/dayongd/output3';

No rows affected (0.788 seconds)

**Import data to an external table, where the LOCATION property is optional:**

jdbc:hive2://> IMPORT EXTERNAL TABLE empolyee\_imported\_external

. . . . . . .> FROM '/user/dayongd/output3'

. . . . . . .> LOCATION '/user/dayongd/output4' ;

No rows affected (0.256 seconds)

**Export and import partitions:**

jdbc:hive2://> EXPORT TABLE employee\_partitioned partition

. . . . . . .> (year=2014, month=11) TO '/user/dayongd/output5';

No rows affected (0.247 seconds)

jdbc:hive2://> IMPORT TABLE employee\_partitioned\_imported

. . . . . . .> FROM '/user/dayongd/output5';

No rows affected (0.14 seconds)

**What is ORDER BY AND SORT BY in HIVE? Why Sort by should be used instead of Order by?**

**ORDER BY (ASC|DESC):** A sorted order is maintained across all of the output from every reducer. It performs the global sort using only one reducer, so it takes a longer time to return the result.

jdbc:hive2://> SELECT name FROM employee ORDER BY NAME DESC;

**SORT BY (ASC|DESC):** This indicates which columns to sort when ordering the reducer input records. This means it completes sorting before sending data to the reducer. The SORT BY statement does not perform a global sort and only makes sure data is locally sorted in each reducer unless we set **mapred.reduce.tasks=1**. In this

case, it is equal to the result of ORDER BY. It can be used as follows:

**--Use more than 1 reducer**

**jdbc:hive2://> SET mapred.reduce.tasks = 2;**

**No rows affected (0.001 seconds)**

**jdbc:hive2://> SELECT name FROM employee SORT BY NAME DESC;**

**What is DISTRIBUTE BY in HIVE?**

Rows with matching column values will be partitioned to the same reducer. When used alone, it does not guarantee sorted input to the reducer. The DISTRIBUTE BY statement is similar to GROUP BY in RDBMS in terms of deciding which reducer to distribute the mapper output to. **When using with SORT BY, DISTRIBUTE BY must be specified before the SORT BY statement**. And, **the column used to distribute must appear in the select column list**. It can be used as follows:

**jdbc:hive2://> SELECT name**

**. . . . . . .> FROM employee\_hr DISTRIBUTE BY employee\_id;**

**Error: Error while compiling statement: FAILED: SemanticException**

**[Error 10004]: Line 1:44 Invalid table alias or column reference**

**'employee\_id': (possible column names are: name)**

**(state=42000,code=10004)**

**jdbc:hive2://> SELECT name, employee\_id**

**. . . . . . .> FROM employee\_hr DISTRIBUTE BY employee\_id;**

**+----------+--------------+**

**| name | employee\_id |**

**+----------+--------------+**

**| Lucy | 103 |**

**| Steven | 102 |**

**| Will | 101 |**

**| Michael | 100 |**

**+----------+--------------+**

**4 rows selected (38.92 seconds)**

**--Used with SORT BY**

**jdbc:hive2://> SELECT name, employee\_id**

**. . . . . . .> FROM employee\_hr**

**. . . . . . .> DISTRIBUTE BY employee\_id SORT BY name;**

**+----------+--------------+**

**| name | employee\_id |**

**+----------+--------------+**

**| Lucy | 103 |**

**| Michael | 100 |**

**| Steven | 102 |**

**| Will | 101 |**

**+----------+--------------+**

**4 rows selected (38.01 seconds)**

**What is ClUSTER BY in Hive?**

This is a shorthand operator to perform DISTRIBUTE BY and SORT BY operations on the same group of columns. And, it is sorted locally in each reducer.

The CLUSTER BY statement does not support ASC or DESC yet. **Compared to ORDER BY, which is globally sorted, the CLUSTER BY operation is sorted in each distributed group.** To fully utilize all the available reducers when doing a global sort, we can do CLUSTER BY first and then ORDER BY. This can be used as follows:

**jdbc:hive2://> SELECT name, employee\_id**

**. . . . . . .> FROM employee\_hr CLUSTER BY name;**

**What are the configuration parameters need to set to set the transaction support in Hive?**

SET hive.support.concurrency = true;

SET hive.enforce.bucketing = true;

SET hive.exec.dynamic.partition.mode = nonstrict;

SET hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;

SET hive.compactor.initiator.on = true;

SET hive.compactor.worker.threads = 1;

For now, all the transactions are autocommuted and only support data in the **Optimized Row Columnar**

**(ORC)** file format and in bucketed tables.

# **What is the Hive SQL COALESCE function, what does it do, and why on earth is it useful?**

There is a function called COALESCE that exists in Hive SQL. It has the following syntax.

**COALESCE(field1, field2, … , fieldn)**

**What does it do?**

Of the fields above (field1, field2, … , fieldn), for each record returned it returns the value of the first field of the ones listed in the COALESCE function that is not NULL. If all of the fields are NULL, it returns NULL.

You use it to create proxy values for NULLs in a field.

It is very useful when you have dirty data and have to use another field as an approximation of the dirty data’s actual value. If you have a field that is full of NULLs, you can use another field to put values in for those NULLs that you think provide a good approximate value of what should be there.

**What is Aggregation Condition – HAVING ?**

**HAVING is added to support the conditional filtering of GROUP BY results. By using HAVING, we can avoid using a subquery after GROUP BY. The following is an example:**

**jdbc:hive2://> SELECT sex\_age.age FROM employee**

**. . . . . . .> GROUP BY sex\_age.age HAVING count(\*)<=1;**

**If we do not use HAVING, we can use a subquery for instance as follows:**

**jdbc:hive2://> SELECT a.age**

**. . . . . . .> FROM**

**. . . . . . .> (SELECT count(\*) as cnt, sex\_age.age**

**. . . . . . .> FROM employee GROUP BY sex\_age.age**

**. . . . . . .> ) a WHERE a.cnt<=1;**

**IMPORTANT:- What is Windowing Functions in Hive?**

**What are the features of Bucketing?**

* Bucketing concept is based on (hashing function on the bucketed column) mod (by total number of buckets). The hash\_function depends on the type of the bucketing column.
* Records with the same bucketed column will always be stored in the same bucket.
* We use CLUSTERED BY clause to divide the table into buckets.
* Physically, each bucket is just a file in the table directory, and Bucket numbering is 1-based.
* Bucketing can be done along with Partitioning on Hive tables and even without partitioning.
* Bucketed tables will create almost equally distributed data file parts

**What are the advantages of Bucketing?**

* Bucketed tables offer efficient sampling than by non-bucketed tables. With sampling, we can try out queries on a fraction of data for testing and debugging purpose when the original data sets are very huge.
* As the data files are equal sized parts, map-side joins will be faster on bucketed tables than non-bucketed tables. In Map-side join, a mapper processing a bucket of the left table knows that the matching rows in the right table will be in its corresponding bucket, so it only retrieves that bucket (which is a small fraction of all the data stored in the right table).
* Similar to partitioning, bucketed tables provide faster query responses than non-bucketed tables.
* Bucketing concept also provides the flexibility to keep the records in each bucket to be sorted by one or more columns. This makes map-side joins even more efficient, since the join of each bucket becomes an efficient merge-sort.

**What are the ways to create a bucketing table for the use case where Customer/user details are partitioned by country and bucketed by state and sorted in ascending order of cities?**

#### **Creation of Bucketed Tables**

We can create bucketed tables with the help of **CLUSTERED BY** clause and optional **SORTED BY** clause in CREATE TABLE statement. With the help of the below HiveQL we can create **bucketed\_user** table with above given requirement.

**CREATE TABLE bucketed\_user(**

**firstname VARCHAR(64),**

**lastname  VARCHAR(64),**

**address   STRING,**

**city   VARCHAR(64),**

**state     VARCHAR(64),**

**post      STRING,**

**phone1    VARCHAR(64),**

**phone2    STRING,**

**email     STRING,**

**web       STRING**

**)**

**COMMENT 'A bucketed sorted user table'**

**PARTITIONED BY (country VARCHAR(64))**

**CLUSTERED BY (state) SORTED BY (city) INTO 32 BUCKETS**

**STORED AS SEQUENCEFILE;**

Unlike partitioned columns (which are not included in table columns definition) , Bucketed columns are included in table definition as shown in above code for **state** and **city** columns.

#### **Inserting data Into Bucketed Tables**

Similar to partitioned tables, we can not directly load bucketed tables with **LOAD DATA (LOCAL) INPATH** command, rather we need to use **INSERT OVERWRITE TABLE … SELECT …FROM** clause from another table to populate the bucketed tables. For this, we will create one temporary table in hive with all the columns in input file from that table we will copy into our target bucketed table.

Lets assume we have created **temp\_user** temporary table, and below is the HiveQL for populating bucketed table with **temp\_user** table.

To populate the bucketed table, we need to set the property **hive.enforce.bucketing = true,**so that Hive knows to create the number of buckets declared in the table definition.

**set hive.enforce.bucketing = true;**

**INSERT OVERWRITE TABLE bucketed\_user PARTITION (country)**

**SELECT  firstname ,**

**lastname  ,**

**address   ,**

**city      ,**

**state     ,**

**post      ,**

**phone1    ,**

**phone2    ,**

**email     ,**

**web       ,**

**country**

**FROM temp\_user;**

Note:

* The property hive.enforce.bucketing = true similar to hive.exec.dynamic.partition=true property in partitioning. By Setting this property we will enable dynamic bucketing while loading data into hive table.
* It will automatically sets the number of reduce tasks to be equal to the number of buckets mentioned in the table definition (for example 32 in our case) and automatically selects the clustered by column from table definition.
* If we do not set this property in Hive Session, we have to manually convey same information to Hive that, number of reduce tasks to be run (for example in our case, by using set mapred.reduce.tasks=32) and CLUSTER BY (state) and SORT BY (city) clause in the above INSERT …SELECT statement at the end.

**Solution for the Example Use Case**

Below is the combined HiveQL along with script required for temporary hive table creation. Lets save this HiveQL into **bucketed\_user\_creation.hql** and save the input file provided in example use case section into **user\_table.txt** file in home directory.

**set hive.exec.dynamic.partition=true;**

**set hive.exec.dynamic.partition.mode=nonstrict;**

**set hive.exec.max.dynamic.partitions.pernode=1000;**

**set hive.enforce.bucketing = true;**

**DROP TABLE IF EXISTS bucketed\_user;**

**CREATE TEMPORARY TABLE temp\_user(**

**firstname VARCHAR(64),**

**lastname  VARCHAR(64),**

**address   STRING,**

**country   VARCHAR(64),**

**city      VARCHAR(64),**

**state     VARCHAR(64),**

**post      STRING,**

**phone1    VARCHAR(64),**

**phone2    STRING,**

**email     STRING,**

**web       STRING**

**)**

**ROW FORMAT DELIMITED**

**FIELDS TERMINATED BY ','**

**LINES TERMINATED BY '\n'**

**STORED AS TEXTFILE;**

**LOAD DATA LOCAL INPATH '/home/user/user\_table.txt' INTO TABLE temp\_user;**

**CREATE TABLE bucketed\_user(**

**firstname VARCHAR(64),**

**lastname  VARCHAR(64),**

**address   STRING,**

**city      VARCHAR(64),**

**state     VARCHAR(64),**

**post      STRING,**

**phone1    VARCHAR(64),**

**phone2    STRING,**

**email     STRING,**

**web       STRING**

**)**

**COMMENT 'A bucketed sorted user table'**

**PARTITIONED BY (country VARCHAR(64))**

**CLUSTERED BY (state) SORTED BY (city) INTO 32 BUCKETS**

**STORED AS SEQUENCEFILE;**

**set hive.enforce.bucketing = true;**

**INSERT OVERWRITE TABLE bucketed\_user PARTITION (country)**

**SELECT  firstname ,**

**lastname  ,**

**address   ,**

**city      ,**

**state     ,**

**post      ,**

**phone1    ,**

**phone2    ,**

**email     ,**

**web       ,**

**country**

**FROM temp\_user;**

**Output**

user@tri03ws-386:~$ hive -f bucketed\_user\_creation.hql

Logging initialized using configuration in jar:file:/home/user/bigdata/apache-hive-0.14.0-bin/lib/hive-common-0.14.0.jar!/hive-log4j.properties

OK

Time taken: 12.144 seconds

OK

Time taken: 0.146 seconds

Loading data to table default.temp\_user

Table default.temp\_user stats: [numFiles=1, totalSize=283212]

OK

Time taken: 0.21 seconds

OK

Time taken: 0.5 seconds

Query ID = user\_20141222163030\_3f024f2b-e682-4b08-b25c-7775d7af4134

Total jobs = 1

Launching Job 1 out of 1

Number of reduce tasks determined at compile time: 32

In order to change the average load for a reducer (in bytes):

  set hive.exec.reducers.bytes.per.reducer=<number>

In order to limit the maximum number of reducers:

  set hive.exec.reducers.max=<number>

In order to set a constant number of reducers:

  set mapreduce.job.reduces=<number>

Starting Job = job\_1419243806076\_0002, Tracking URL = http://tri03ws-386:8088/proxy/application\_1419243806076\_0002/

Kill Command = /home/user/bigdata/hadoop-2.6.0/bin/hadoop job  -kill job\_1419243806076\_0002

Hadoop job information for Stage-1: number of mappers: 1; **number of reducers: 32**

2014-12-22 16:30:36,164 Stage-1 map = 0%,  reduce = 0%

2014-12-22 16:31:09,770 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 1.66 sec

2014-12-22 16:32:10,368 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 1.66 sec

2014-12-22 16:32:28,037 Stage-1 map = 100%,  reduce = 13%, Cumulative CPU 3.19 sec

2014-12-22 16:32:36,480 Stage-1 map = 100%,  reduce = 14%, Cumulative CPU 7.06 sec

2014-12-22 16:32:40,317 Stage-1 map = 100%,  reduce = 19%, Cumulative CPU 7.63 sec

2014-12-22 16:33:40,691 Stage-1 map = 100%,  reduce = 19%, Cumulative CPU 12.28 sec

2014-12-22 16:33:54,846 Stage-1 map = 100%,  reduce = 31%, Cumulative CPU 17.45 sec

2014-12-22 16:33:58,642 Stage-1 map = 100%,  reduce = 38%, Cumulative CPU 21.69 sec

2014-12-22 16:34:52,731 Stage-1 map = 100%,  reduce = 56%, Cumulative CPU 32.01 sec

2014-12-22 16:35:21,369 Stage-1 map = 100%,  reduce = 63%, Cumulative CPU 35.08 sec

2014-12-22 16:35:22,493 Stage-1 map = 100%,  reduce = 75%, Cumulative CPU 41.45 sec

2014-12-22 16:35:53,559 Stage-1 map = 100%,  reduce = 94%, Cumulative CPU 51.14 sec

2014-12-22 16:36:14,301 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 54.13 sec

MapReduce Total cumulative CPU time: 54 seconds 130 msec

Ended Job = job\_1419243806076\_0002

Loading data to table default.bucketed\_user partition (country=null)

Time taken for load dynamic partitions : 2421

Loading partition {country=AU}

Loading partition {country=country}

Loading partition {country=US}

Loading partition {country=UK}

Loading partition {country=CA}

Time taken for adding to write entity : 17

Partition default.bucketed\_user{country=AU} stats: [numFiles=32, numRows=500, totalSize=78268, rawDataSize=67936]

Partition default.bucketed\_user{country=CA} stats: [numFiles=32, numRows=500, totalSize=76564, rawDataSize=66278]

Partition default.bucketed\_user{country=UK} stats: [numFiles=32, numRows=500, totalSize=85604, rawDataSize=75292]

Partition default.bucketed\_user{country=US} stats: [numFiles=32, numRows=500, totalSize=75468, rawDataSize=65383]

Partition default.bucketed\_user{country=country} stats: [numFiles=32, numRows=1, totalSize=2865, rawDataSize=68]

MapReduce Jobs Launched:

Stage-Stage-1: Map: 1  Reduce: 32   Cumulative CPU: 54.13 sec   HDFS Read: 283505 HDFS Write: 316247 SUCCESS

Total MapReduce CPU Time Spent: 54 seconds 130 msec

OK

Time taken: 396.486 seconds

user@tri03ws-386:~$

From the above box, we can see that mapreduce job initiated 32 reduce tasks for 32 buckets and four partitions are created by country.

#### **Table Sampling in Hive**

Table Sampling in hive is nothing but extraction small fraction of data from the original large data sets. It is similar to LIMIT operator in Hive.

But below are the difference between LIMIT and TABLESAMPLE in Hive.

* In many cases a LIMIT clause executes the entire query, and then only returns a limited results.
* But Sampling will only select a portion of data to perform query.

hive> SELECT firstname, country, state, city FROM bucketed\_user

> TABLESAMPLE(BUCKET 32 OUT OF 32 ON state);

hive> SELECT firstname, country, state, city FROM bucketed\_user TABLESAMPLE(1 PERCENT);

**What does it mean by TABLESAMPLE(BUCKET x OUT OF y)?**

When you create the table and bucket it using the clustered by clause into 32 buckets (as an example), hive buckets your data into 32 buckets using deterministic hash functions. Then when you use TABLESAMPLE(BUCKET x OUT OF y), hive divides your buckets into groups of **y** buckets and then picks the **x**'th bucket of each group. For example:

* If you use TABLESAMPLE(BUCKET 6 OUT OF 8), hive would divide your 32 buckets into groups of 8 buckets resulting in 4 groups of 8 buckets and then picks the 6th bucket of each group, hence picking the buckets 6, 14, 22, 30.
* If you use TABLESAMPLE(BUCKET 23 OUT OF 32), hive would divide your 32 buckets into groups of 32, resulting in only 1 group of 32 buckets, and then picks the 23rd bucket as your result.
* If you use TABLESAMPLE(BUCKET 3 OUT OF 64), hive would divide your 32 buckets into groups of 64 buckets, resulting in 1 group of 64 "half-bucket"s and then picks the half-bucket that corresponds to the 3rd full-bucket.

**What is EXPLAIN and ANALYZE statements in Hive?**

The EXPLAIN and ANALYZE statements that can be used as utilities to check and identify the performance of queries.

**The EXPLAIN Statement**

The EXPLAIN command return a query execution plan without running the query. We can use an EXPLAIN command for queries if we have a doubt or a concern about performance.

**The syntax for EXPLAIN is as follows:**

**EXPLAIN [EXTENDED|DEPENDENCY|AUTHORIZATION] hive\_query**

* **EXTENDED**: This provides additional information for the operators in the plan, such as

file pathname and abstract syntax tree.

* **DEPENDENCY**: This provides a JSON format output that contains a list of tables and

partitions that the query depends on. It is available since HIVE 0.10.0.

* **AUTHORIZATION**: This lists all entities needed to be authorized including input and

output to run the Hive query and authorization failures, if any. It is

available since HIVE 0.14.0.

A typical query plan contains the following three sections.

**Abstract syntax tree (AST):** Hive uses a pacer generator called ANTLR to automatically generate a tree of syntax for HQL.

**Stage dependencies:** This lists all dependencies and number of stages used to run the query.

**Stage plans:** It contains important information, such as operators and sort orders, for running the job.

**The ANALYZE Statement**

Hive statistics are a collection of data that describe more details, such as the number of rows, number of files, and raw data size, on the objects in the Hive database. Statistics is a metadata of Hive data. Hive supports statistics at the table, partition, and column level.

These statistics serve as an input to the **Hive Cost-Based Optimizer (CBO)**, which is an optimizer to pick the query plan with the lowest cost in terms of system resources required to complete the query.

The statistics are gathered through the ANALYZE statement since Hive 0.10.0 on tables, partitions, and columns as given in the following examples:

**jdbc:hive2://> ANALYZE TABLE employee COMPUTE STATISTICS;**

No rows affected (27.979 seconds)

jdbc:hive2://> **ANALYZE TABLE employee\_partitioned**

. . . . . . .> PARTITION(year=2014, month=12) COMPUTE STATISTICS;

No rows affected (45.054 seconds)

jdbc:hive2://> **ANALYZE TABLE employee\_id COMPUTE STATISTICS**

**. . . . . . .> FOR COLUMNS employee\_id;**

No rows affected (41.074 seconds)

Once the statistics are built, we can check the statistics by the DESCRIBE EXTENDED/FORMATTED statement. From the table/partition output, we can find the statistics information inside the parameters, such as parameters:{numFiles=1,

COLUMN\_STATS\_ACCURATE=true, transient\_lastDdlTime=1417726247, numRows=4,

totalSize=227, rawDataSize=223}). The following is an example:

**jdbc:hive2://> DESCRIBE EXTENDED employee\_partitioned**

**. . . . . . .> PARTITION(year=2014, month=12);**

jdbc:hive2://> **DESCRIBE EXTENDED employee;**

… parameters:{numFiles=1, COLUMN\_STATS\_ACCURATE=true,

transient\_lastDdlTime=1417726247, numRows=4, totalSize=227,

rawDataSize=223}).

**What is the property to persist the Hive Statistics in the Metastore to avoid computing them every time.?**

jdbc:hive2://> SET hive.stats.autogather=ture;

**How many types of logs available in Hive?**

There are two types of logs available in Hive:

1) **System log** :- The system log contains the Hive running status and issues. It is configured in

**{HIVE\_HOME}/conf/hive-log4j.properties**. The following three lines for Hive log can be found:

hive.root.logger=WARN,DRFA

hive.log.dir=/tmp/${user.name}

hive.log.file=hive.log

To modify the status, we can either modify the preceding lines in **hive-log4j.properties** (applies to all users) or set from the Hive CLI (only applies to the current user and current session) as follows:

**hive --hiveconf hive.root.logger=DEBUG,console**

2) **Job log :-** The job log contains Hive query information and is saved at the same place, **/tmp/${user.name}**, by default as one file for each Hive user session. We can override it in **hive-site.xml** with the **hive.querylog.location** property. If a Hive query generates MapReduce jobs, those logs can also be viewed through the Hadoop JobTracker Web UI.

**What are the features available for the Performance Optimizations in HIVE?**

Since Hive 0.13.0, Hive includes the following new features for performance optimizations:

* **TEZ** :- Tez is an application framework built on Yarn that can execute complex directed acyclic graphs (DAGs) for general data-processing tasks. Tez further splits map and reduce jobs into smaller tasks and combines them in a flexible and efficient way for execution. Tez is considered a flexible and powerful

successor to the MapReduce framework. To configure Hive to use Tez, we need to overwrite the following settings from the default MapReduce:

**SET hive.execution.engine=tez;**

* **Vectorization :-** Vectorization optimization processes a larger batch of data at the same time rather than one row at a time, thus significantly reducing computing overhead. Each batch consists of a column vector that is usually an array of primitive types. Operations are performed on the entire column vector, which improves the instruction pipelines and cache usage. Files must be stored in the **Optimized Row**

**Columnar** (**ORC**) format in order to use vectorization.

**To enable vectorization, we need to do the following setting:**

**SET hive.vectorized.execution.enabled=true;**

**What is the property to set the Default File Format? What is the Default File Format?**

SET hive.default.fileformat=<File\_Format> --default file format for table

TEXTFILE is the default file format.

**How we can load the data in other file format other than TEXTFILE?**

We can load a text file directly to a table with the TEXTFILE format. To load data to the table with other file formats, we need to load the data to a TEXTFILE format table first.

Then, use **INSERT OVERWRITE TABLE <target\_file\_format\_table> SELECT \* FROM**

**<text\_format\_source\_table>** to convert and insert the data to the file format as expected.

**What is Hadoop Archive File (HAR)?**

**Hadoop Archive File** (**HAR**) is another type of file format to pack HDFS files into archives. This is an option (not a good option) for storing a large number of small-sized files in HDFS, as storing a large number of small-sized files directly in HDFS is not very efficient.

**How we can compress the intermediate files produced by Hive between multiple MapReduce Jobs?**

we need to set the following property (false by default) in the Hive CLI or the hive-site.xml file:

**jdbc:hive2://> SET hive.exec.compress.intermediate=true**

**what are the solutions to deal with too many small-file issues?**

* **Hadoop Archive and HAR :-** These are toolkits to pack small files.
* **SequenceFile Format :-** This is a format to compress small files to bigger files.
* **CombineFileInputFormat:-** A type of InputFormat to combine small files before

map and reduce processing. It is the default InputFormat for Hive.

* **HDFS federation** :- It makes namenodes extensible and powerful to manage more

files.

**What are the configurations required for merging files of query results to avoid recreating small files.?**

* **hive.merge.mapfiles**: This merges small files at the end of a map-only job. By default, it is true.
* **hive.merge.mapredfiles**: This merges small files at the end of a MapReduce job. Set it to true since its default is false.
* **hive.merge.size.per.task**: This defines the size of merged files at the end of the job. The default value is 256,000,000.
* **hive.merge.smallfiles.avgsize:** This is the threshold for triggering file merge. The default value is 16,000,000.

When the average output file size of a job is less than the value specified by **hive.merge.smallfiles.avgsize**, and both **hive.merge.mapfiles (for map-only jobs) and hive.merge.mapredfiles (for MapReduce jobs)** are set to true, Hive will start an additional MapReduce job to merge the output files into big files.

**What is JVM reuse?**

By default, Hadoop launches a new JVM for each map or reduce job and runs the map or reduce task in parallel. When the map or reduce job is a lightweight job running only for a few seconds, the JVM startup process could be a significant overhead. The MapReduce framework (version 1 only, not Yarn) has an option to reuse JVM by sharing the JVM to run mapper/reducer serially instead of parallel. JVM reuse applies to map or reduce tasks in the same job. Tasks from different jobs will always run in a separate JVM. To enable

the reuse, we can set the maximum number of tasks for a single job for JVM reuse using

the **mapred.job.reuse.jvm.num.tasks** property. **Its default value is 1**:

**jdbc:hive2://> SET mapred.job.reuse.jvm.num.tasks=5;**

**We can also set the value to –1 to indicate that all the tasks for a job will run in the same JVM.**

**How to enable the Parallel execution of stages in Hive?**

Hive queries commonly are translated into a number of stages that are executed by the default sequence. These stages are not always dependent on each other. Instead, they can run in parallel to save the overall job running time. We can enable this feature with the following settings:

**jdbc:hive2://> SET hive.exec.parallel=true;—default false**

**jdbc:hive2://> SET hive.exec.parallel.thread.number=16;**

**-- default 8, it defines the max number for running in parallel**

**What are User-defined functions?**

Hive defines the following three types of UDF:

* **UDFs**:- These are regular user-defined functions that operate row-wise and output one

result for one row, such as most built-in mathematic and string functions.

* **UDAFs**:- These are user-defined aggregating functions that operate row-wise or group-wise and output one row or one row for each group as a result, such as the MAX and COUNT built-in functions.
* **UDTFs:-**These are user-defined table-generating functions that also operate rowwise, but they produce multiple rows/tables as a result, such as the EXPLODE function.

UDTF can be used either after SELECT or after the LATERAL VIEW statement.

**What are the Steps to write the UDF and execute that UDF?**

**1)**Create a folder for the UDF like

$ mkdir -p src/main/java/com/snowplowanalytics/hive/udf

**2)**Create a UDF Java file (For example, **ToUpper.java**)

package com.snowplowanalytics.hive.udf;

2

3 import org.apache.hadoop.hive.ql.exec.UDF;

4 import org.apache.hadoop.hive.ql.exec.Description;

5 import org.apache.hadoop.io.Text;

6

7 @Description(

8 name = "toupper",

9 value = "\_FUNC\_(str) - Converts a string to uppercase",

10 extended = "Example:\n" +

11 " > SELECT toupper(author\_name) FROM authors a;\n" +

12 " STEPHEN KING"

13 )

14 public class ToUpper extends UDF {

15

16 public Text evaluate(Text s) {

17 Text to\_value = new Text("");

18 if (s != null) {

19 try {

20 to\_value.set(s.toString().toUpperCase());

21 } catch (Exception e) { // Should never happen

22 to\_value = new Text(s);

23 }

24 }

25 return to\_value;

26 }

27 }

This file defines our UDF, ToUpper. The package definition and imports should be self-explanatory; the @Description annotation is a useful Hive-specific annotation to provide usage information for our UDF in the Hive console.

All user-defined functions extend the Hive UDF class; a UDF sub-class must then implement one or more methods named “**evaluate**” which will be called by Hive. We implement an evaluate method which takes one Hadoop Text (which stores text using UTF8) and returns the same Hadoop Text, but now in upper-case.

## 3) Building and using our UDF

## Create a jar for the custom file and add to the Hive console and create a temporary function .

1 > add jar /path/to/ToUpper.jar;

2 > create temporary function to\_upper as 'com.snowplowanalytics.hive.udf.ToUpper';

And then finally you can use our new UDF in your HiveQL queries, something like this:

1 > SELECT to\_upper(author\_name) FROM authors a;

## What are the steps to write the UDAF and execute the UDAF?

## 1)Max.java

## import org.apache.hadoop.hive.ql.exec.UDAF;

**import** org.apache.hadoop.io.IntWritable;

**import** org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

@SuppressWarnings("deprecation")

**public** **class** **Max** **extends** **UDAF** {

**public** **static** **class** **MaxIntUDAFEvaluator** **implements** **UDAFEvaluator** {

**private** IntWritable output;

**public** **void** **init**()

{

output = **null**;

}

**public** **boolean** **iterate**(IntWritable maxvalue) // Process input table

{

**if** (maxvalue == **null**)

{

**return** **true**;

}

**if** (output == **null**)

{

output = **new** IntWritable(maxvalue.get());

}

**else**

{

output.set(Math.*max*(output.get(), maxvalue.get()));

}

**return** **true**;

}

**public** IntWritable **terminatePartial**()

{

**return** output;

}

**public** **boolean** **merge**(IntWritable other)

{

**return** iterate(other);

}

**public** IntWritable **terminate**() // final result

{

**return** output;

}

}

}

* **init**() – This method initializes the evaluator and resets its internal state.
* **iterate**() – This method is called every time there is a new value to be aggregated. The evaluator should update its internal state with the result of performing the aggregation.
* **terminatePartial**() – This method is called when Hive wants a result for the partial aggregation. The method must return an object that encapsulates the state of the aggregation.
* **merge**() – This method is called when Hive decides to combine one partial aggregation with another.
* **terminate**() – This method is called when the final result of the aggregation is needed.

## 2) Compile and create a jar file of your java project.

## 3) Add jar file in distributed cache, create a function and execute udaf function.

## hive> ADD JAR /home/hduser/Desktop/HIVE/MaxUDAF.jar;

## hive> CREATE TEMPORARY FUNCTION max AS 'Max';

## hive> SELECT max(Num) FROM Num\_list;

Will check the other details later

What is SerDe ? How to use the SerDe?

The SerDe interface allows you to instruct Hive as to how a record should be processed. A SerDe is a combination of a Serializer and a Deserializer (hence, Ser-De). The [Deserializer](http://javasourcecode.org/html/open-source/hive/hive-0.9.0/org/apache/hadoop/hive/serde2/Deserializer.html) interface takes a string or binary representation of a record, and translates it into a Java object that Hive can manipulate. The [Serializer](http://javasourcecode.org/html/open-source/hive/hive-0.9.0/org/apache/hadoop/hive/serde2/Serializer.html), however, will take a Java object that Hive has been working with, and turn it into something that Hive can write to HDFS or another supported system. Commonly, Deserializers are used at query time to execute SELECT statements, and Serializers are used when writing data, such as through an INSERT-SELECTstatement.

Tables can be configured to process data using a SerDe by specifying the SerDe to use at table creation time, or through the use of an ALTER TABLE statement. For example:

**ADD JAR /tmp/hive-serdes-1.0-SNAPSHOT.jar**

**CREATE EXTERNAL TABLE tweets (**

**...**

**retweeted\_status STRUCT&lt;**

**text:STRING,**

**user:STRUCT&lt;screen\_name:STRING,name:STRING&gt;&gt;,**

**entities STRUCT&lt;**

**urls:ARRAY&lt;STRUCT&lt;expanded\_url:STRING&gt;&gt;,**

**user\_mentions:ARRAY&lt;STRUCT&lt;screen\_name:STRING,name:STRING&gt;&gt;,**

**hashtags:ARRAY&lt;STRUCT&lt;text:STRING&gt;&gt;&gt;,**

**text STRING,**

**...**

**)**

**PARTITIONED BY (datehour INT)**

**<strong>ROW FORMAT SERDE 'com.cloudera.hive.serde.JSONSerDe'</strong>**

**LOCATION '/user/flume/tweets';**

The bolded red section of the above CREATE TABLE statement shows how a table is configured to use a SerDe. If the SerDe is not on the Hive classpath, it must be added at runtime using the ADD JARcommand.

**How we can develop a custom SerDe ?**

we can write a basic template for a SerDe, which utilizes the Hive serde2 API (**org.apache.hadoop.hive.serde2**)

package com.cloudera.hive.serde; import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.Properties;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hive.serde.Constants;

import org.apache.hadoop.hive.serde2.SerDe;

import org.apache.hadoop.hive.serde2.SerDeException;

import org.apache.hadoop.hive.serde2.SerDeStats;

import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;

import org.apache.hadoop.hive.serde2.typeinfo.StructTypeInfo;

import org.apache.hadoop.hive.serde2.typeinfo.TypeInfo;

import org.apache.hadoop.hive.serde2.typeinfo.TypeInfoFactory;

import org.apache.hadoop.hive.serde2.typeinfo.TypeInfoUtils;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.Writable;

/\*\*

\* A template for a custom Hive SerDe

\*/

public class **BoilerplateSerDe** implements **SerDe** {

private StructTypeInfo rowTypeInfo;

private ObjectInspector rowOI;

private List<String> colNames;

private List<Object> row = new ArrayList<Object>();

/\*\*

  \* An initialization function used to gather information about the table.

  \* Typically, a SerDe implementation will be interested in the list of

  \* column names and their types. That information will be used to help

  \* perform actual serialization and deserialization of data.

  \*/

@Override

public void **initialize**(Configuration conf, Properties tbl)

     throws SerDeException {

   // Get a list of the table's column names.

   String colNamesStr = tbl.getProperty(Constants.LIST\_COLUMNS);

   colNames = Arrays.asList(colNamesStr.split(","));

   // Get a list of TypeInfos for the columns. This list lines up with

   // the list of column names.

   String colTypesStr = tbl.getProperty(Constants.LIST\_COLUMN\_TYPES);

   List<TypeInfo> colTypes =

       TypeInfoUtils.getTypeInfosFromTypeString(colTypesStr);

   rowTypeInfo =

       (StructTypeInfo) TypeInfoFactory.getStructTypeInfo(colNames, colTypes);

   rowOI =

       TypeInfoUtils.getStandardJavaObjectInspectorFromTypeInfo(rowTypeInfo);

}

/\*\*

  \* This method does the work of deserializing a record into Java objects

  \* that Hive can work with via the ObjectInspector interface.

  \*/

@Override

public Object **deserialize**(Writable blob) throws SerDeException {

   row.clear();

   // Do work to turn the fields in the blob into a set of row fields

   return row;

}

/\*\*

  \* Return an **ObjectInspector** for the row of data

  \*/

@Override

public ObjectInspector **getObjectInspector**() throws SerDeException {

   return rowOI;

}

/\*\*

  \* Unimplemented

  \*/

@Override

public SerDeStats getSerDeStats() {

   return null;

}

/\*\*

  \* Return the class that stores the serialized data representation.

  \*/

@Override

public Class<? extends Writable> **getSerializedClass**() {

   return Text.class;

}

/\*\*

  \* This method takes an object representing a row of data from Hive, and

  \* uses the ObjectInspector to get the data for each column and serialize

  \* it.

  \*/

@Override

public Writable **serialize**(Object obj, ObjectInspector oi)

     throws SerDeException {

   // Take the object and transform it into a serialized representation

   return new Text();

}

}

* **initialize**() method is called only once and gathers some commonly-used pieces of information from the table properties, such as the column names and types. Using the type info of the row, you can instantiate an **ObjectInspector** for the row ([ObjectInspectors](http://javasourcecode.org/html/open-source/hive/hive-0.9.0/org/apache/hadoop/hive/serde2/objectinspector/ObjectInspector.html) are Hive objects that are used to describe and examine complex type hierarchies.)
* In a SerDe, the **serialize**() method takes a Java object representing a row of data, and converts that object into a serialized representation of the row. The serialized class is determined by the return type of **getSerializedClass**().

**What is Default build-in SerDe?**

**LazySimpleSerDe**: The default built-in SerDe (org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe) that’s used with the

TEXTFILE format. It can be implemented as follows:

**jdbc:hive2://> CREATE TABLE test\_serde\_lz**

**. . . . . . .> STORED AS TEXTFILE AS**

**. . . . . . .> SELECT name from employee;**

**How Hive Security has been achieved?**

Hive Security has been achieved through following areas: -

1) **Authentication**

Authentication is the process of verifying the identity of a user by obtaining the user’s

credentials. Hive has offered authentication since HiveServer2. In the previous

HiveServer, if we could access the host/port over the network, we could access the data.

In this case, the Hive Metastore Server can be used to authenticate thrift clients using

Kerberos.

**Kerberos**

Kerberos is a network authentication protocol developed by MIT. Kerberos authentication process: **client, server, and Key Distribution Center (KDC)**. All clients and servers registered to KDC are known as a realm, which is typically the domain’s DNS name in all caps.

**What are the configuration required to authenticate the clients with the Hive Metastore server using Kerberos?**

The following properties need to be configured in the **hive-site.xml**

* Enable the **Simple Authentication and Security Layer** (**SASL**) framework to

enforce client Kerberos authentication, as follows:

**<property>**

**<name>hive.metastore.sasl.enabled</name>**

**<value>true</value>**

**<description>If true, the metastore thrift interface will be secured**

**with SASL framework. Clients must authenticate with Kerberos.**

**</description>**

**</property>**

* Specify the Kerberos **keytab** that is generated. Override the following example if we want to keep the file in another place. Make sure the file access permissions are set to 400 implying only read permission for the owner to avoid their identity being stolen by others:

**<property>**

**<name>hive.metastore.kerberos.keytab.file</name>**

**<value>/etc/hive/conf/hive.keytab</value>**

**<description>The sample path to the Kerberos Keytab file containing**

**the metastore thrift server's service principal.</description>**

**</property>**

* Specify the Kerberos principal pattern string. The special string \_HOST will be replaced automatically with the correct hostnames. The YOUR-REALM.COM value should be replaced by the actual realm name:

**<property>**

**<name>hive.metastore.kerberos.principal</name>**

**<value>hive/\_HOST@YOUR-REALM.COM</value>**

**<description>The service principal for the metastore thrift server.**

**</description>**

**</property>**

**What are the HiveServer2 authentication modes and how to configure them?**

HiveServer2 supports the following authentications. To configure HiveServer2 to use one of these authentication modes, we can set the proper properties in **hive\_site.xml** as follows:

* **None authentication:** None authentication is what’s in the default settings. “None” here means Hive allows anonymous access as shown in the following setting:

**<property>**

**<name>hive.server2.authentication</name>**

**<value>NONE</value>**

**</property>**

* **Kerberos authentication:** If Kerberos authentication is used, authentication is supported between the thrift client and HiveServer2, and between HiveServer2 and secure HDFS. To enable Kerberos authentication for HiveServer2, we can set the following properties by overriding the keytab path (if we want to keep the file in another place) as well as changing YOUR-REALM.COM to the actual realm name:

**<property>**

**<name>hive.server2.authentication</name>**

**<value>KERBEROS</value>**

**</property>**

**<property>**

**<name>hive.server2.authentication.kerberos.keytab</name>**

**<value>/etc/hive/conf/hive.keytab</value>**

**</property>**

**<property>**

**<name>hive.server2.authentication.kerberos.principal</name>**

**<value>hive/\_HOST@YOUR-REALM.COM</value>**

**</property>**

Once Kerberos is enabled, the JDBC client (such as Beeline) must include the principal parameter in the JDBC connection string such as the following:

**jdbc:hive2://HiveServer2HostName:10000/default;principal=hive/HiveServe**[r2HostName@YOUR-REALM.COM](mailto:r2HostName@YOUR-REALM.COM)

* **LDAP authentication:** To configure HiveServer2 to use user and password

validation backed by LDAP, we can set the following properties:

**<property>**

**<name>hive.server2.authentication</name>**

**<value>LDAP</value>**

**</property>**

**<property>**

**<name>hive.server2.authentication.ldap.url</name>**

**<value>LDAP\_URL, such as ldap://ldaphost@company.com</value>**

**</property>**

**<property>**

**<name>hive.server2.authentication.ldap.Domain</name>**

**<value>Your Domain Name</value>**

**</property>**

* **Pluggable custom authentication:** Pluggable custom authentication provides a

custom authentication provider for HiveServer2. To enable it, configure the settings

as follows:

**<property>**

**<name>hive.server2.authentication</name>**

**<value>CUSTOM</value>**

**</property>**

**<property>**

**<name>hive.server2.custom.authentication.class</name>**

**<value>pluggable-auth-class-name</value>**

**<description> Custom authentication class name, such as**

**com.packtpub.hive.essentials.hiveudf.customAuthenticator**

**</description>**

**</property>**

The following is a sample of a customized class that implements the **org.apache.hive.service.auth.PasswdAuthenticationProvider** interface. The overridden Authenticate method has the core logic of how to authenticate a username and password. Make sure to copy the compiled JAR file to **$HIVE\_HOME/lib/** so that the preceding settings can work.

**customAuthenticator.java**

**package com.packtpub.hive.essentials.hiveudf;**

import java.util.Hashtable;

import javax.security.sasl.AuthenticationException;

import org.apache.hive.service.auth.PasswdAuthenticationProvider;

/\*

\* The customized class for HiveServer2 authentication

\*/

public class customAuthenticator implements

PasswdAuthenticationProvider {

Hashtable<String, String> authHashTable = null;

public customAuthenticator () {

authHashTable = new Hashtable<String, String>();

authHashTable.put("user1", "passwd1");

authHashTable.put("user2", "passwd2");

}

@Override

public void Authenticate(String user, String password)

throws AuthenticationException {

String storedPasswd = authHashTable.get(user);

if (storedPasswd != null && storedPasswd.equals(password))

return;

throw new AuthenticationException("customAuthenticator Exception:

Invalid user");

}

}

**What are the different modes of authorization in Hive?**

**Authorization** in Hive is used to verify if a user has permission to perform a certain action, such as creating, reading, and writing data or metadata.

Hive provides three authorization modes:

* legacy mode
* storage-based mode,
* SQL standard-based mode.

**Legacy Mode**

This is the default authorization mode in Hive, providing column and row-level authorization through HQL statements. However, it is not a completely secure authorization mode and has a couple of limitations. It can be mainly used to prevent good users from accidentally doing bad things rather than preventing malicious users’

operations. In order to enable the legacy authorization mode, we need to set the following properties in **hive-site.xml**:

<property>

<name>**hive.security.authorization.enabled**</name>

<value>**true**</value>

<description>**enables or disable the hive client authorization**</description>

</property>

<property>

<name>**hive.security.authorization.createtable.owner.grants**</name>

<value>**ALL**</value>

<description>**the privileges automatically granted to the owner whenever a**

**table gets created. An example like "select, drop" will grant select and**

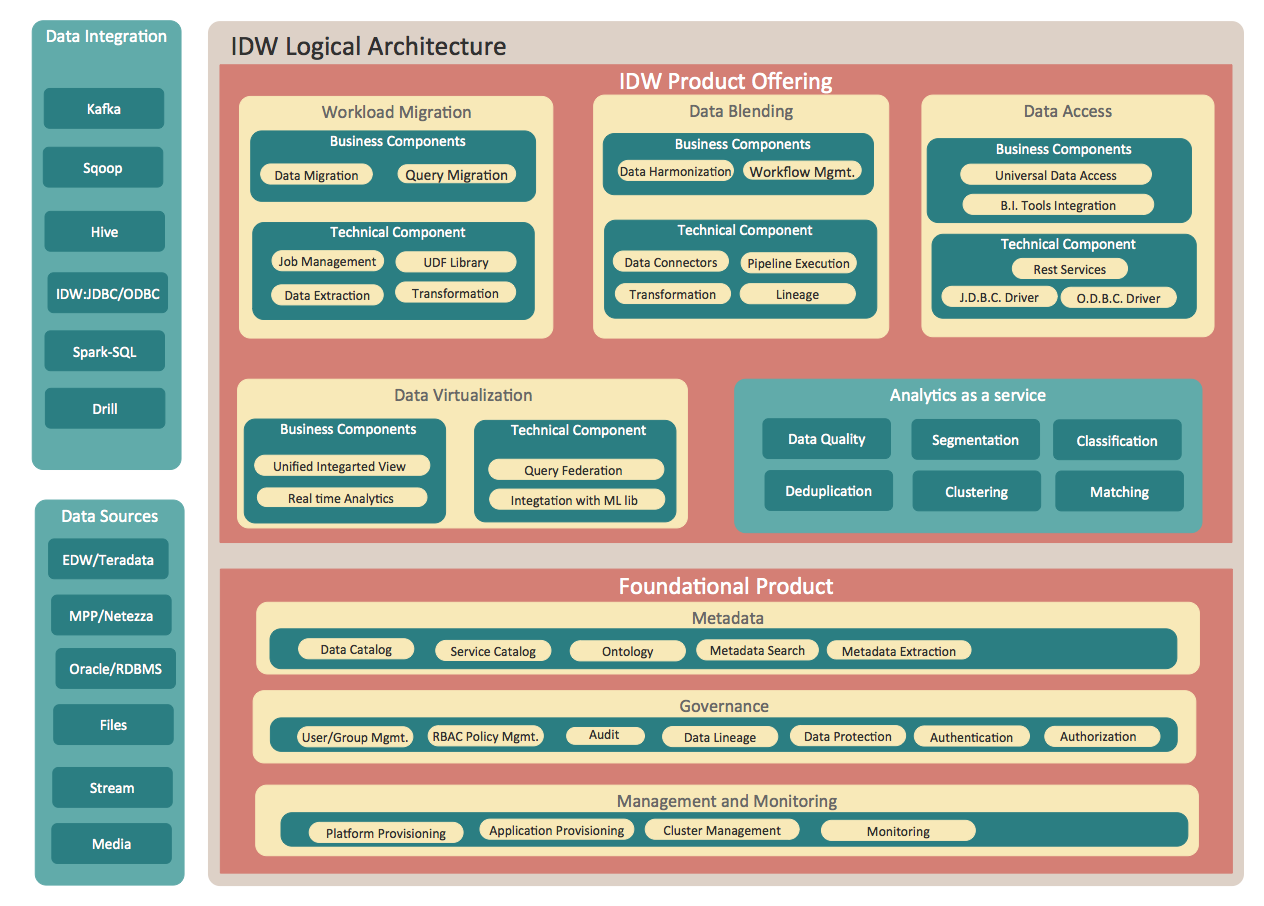
**drop privilege to the owner of the table.**

</description>

</property>

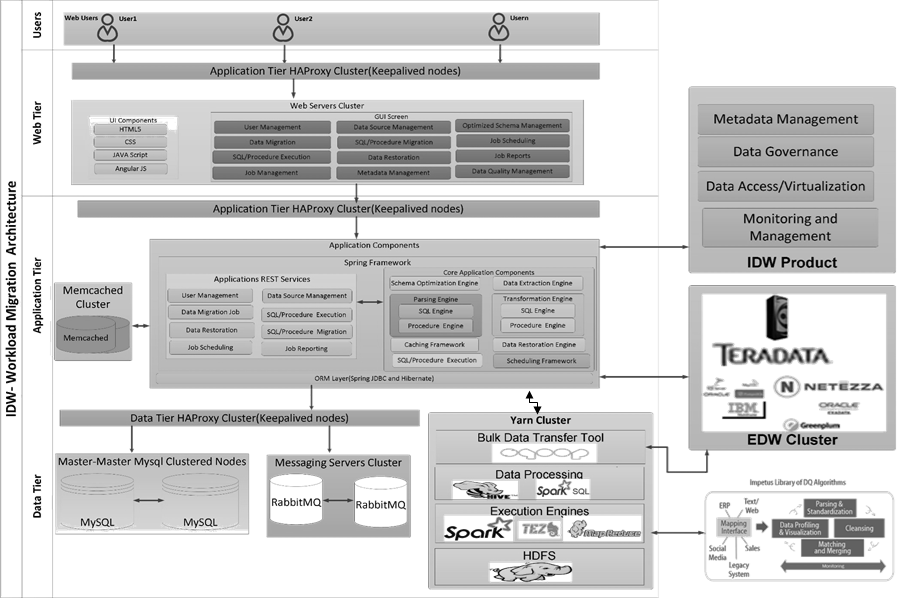
**IDW Architecture**

**What is the architecture of your current project (IDW)?**

****

**What is the architecture of your module Workload Migration (WM)? What are the Workload Migration Features?**

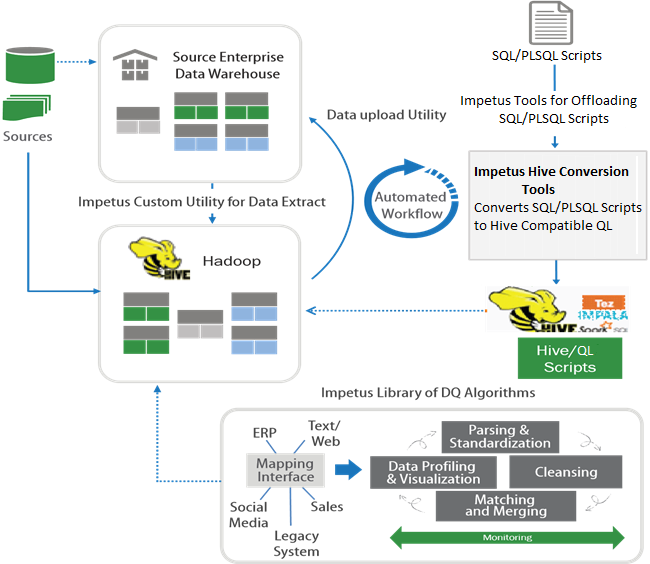
#################### PRINTED TILL HERE ###############################

****

* **Intelligent Identification of “Offload-able” Entities**
  + **Workload based tables classification**
  + **Usage based tables classification**
* **Data Migration**
  + **Recommendation for partitioning, clustering and number of buckets based on dataset**
  + **Migrate role based security**
* **Workload/Logic Transformation**
  + **Impetus UDF Library**
  + **Automatic conversation of SQL and PL/SQL scripts**
* **Workload Execution**
  + **Schedule migrated code**
* **Ease of Deployment**

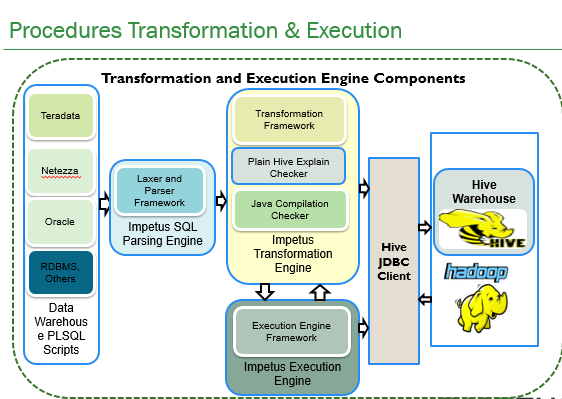
**- Packaged as standard java web application and easily deployable on Hortwonworks Edge node**

**How Workload Migration works?**

****

* **Step 1: Identification**
* **Step 2: Schema, Security and Data Migration**
* **Step 3: Logic Migration**
* **Step 4: Logic Execution**
* **Step 5: Data Quality Enhancement**
* **Step 6: Analytical Data Moved to EDW Datamart**

**What are the flow for the Procedures Transformation & Execution?**

****

* **WM Laxer and Parser parses PLSQL files for syntax compatibility and generate parsing tree**
* **Transformation Engine scan the generated tree and create equivalent Java Hive Objects and** 
  + **Transforms all SQL queries in the SQL file into Corresponding Hive QL**
  + **Auto apply Hive UDF/Custom EDW UDF for Hive QL**
  + **Auto converts EDW Data Types to Hive Data Type**
  + **Validates compatibility of generated HQL**
  + **Validates generated Java code for compilation errors**
* **Execution Engine** 
  + **Executes generated transformed Java code over Hive JDBC**
  + **Auto convert Dynamic SQL query into Hive QL**

**- Accepts custom Jar for the execution in case of the manual intervention**

**How WM different stages work from the code level.?**

**There are mainly 5 different stages in the workload migration.**

**1)Migration: -** In the Migration stage , the data from the different EDW data sources will be migrated to HDFS . Once this stage is configured providing the source datasource (like Oracle , Teradata etc.) and the destination datasource (Hive) **, a service(**PipelineServiceImpl) got called to save the migration stage and a consumer (OpenJMS consumer will be created ) and there will be a producer listening at the queue .

At the Migration stage , the data got migrated to different EDW datasources to HDFS . For this there are Sqoop scripts has been used to import and export the Data .

Sqoop Import command scripts has been used to import the data from RDBMS datasource to HDFS.

Sqoop Export command scripts has been used to export the data from HDFS to RDBMS datasource .

There are different velocity templates like (sqoop-import-oracle.vm) has been created with the placeholder to fill the required value at runtime. Then there are different class (OracleSqoopCommandGenerator) has been written to fill the placeholder. Then the Sqoop command got executed and Data got migrated. Once that stage got completed, then there will be a rest service call to show the results at the UI. If there are any issues with the stage , then also a rest service got called to show at the UI and it will hard stop the Pipeline and next stage will not run.

There is also one component with Migration stage for the Data validation where the Data got validated with the Max , count , Avg. etc. aggregation on the source and destination datasource.

**2)Transformation: -** This is the stage got executed after the Migration stage. In this stage , the SQL and PLSQL got transformed to HQL and java classes which got executed using the JDBC connection .

In this stage, SQL and PLSQL file got transferred using the ANTLR.

The Transformed files got executed at the HIVE and if it got executed successfully at the Hive. The Stage has been marked as successful.

**3)Execution: -**

**4) Data Generator: -**

**5) Query Level Validation: -**

**SPARK**

**Does Spark require Hadoop?**

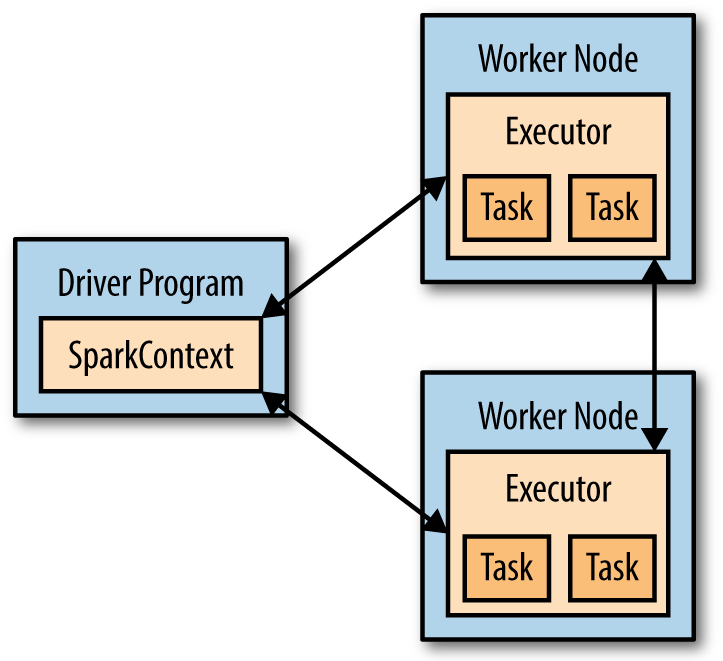
Spark does not require Hadoop; it simply has support for storage systems implementing the Hadoop APIs. Spark can create distributed datasets from any file stored in the Hadoop distributed filesystem (HDFS) or other storage systems supported by the Hadoop APIs (including your local filesystem, Amazon S3, Cassandra, Hive, HBase, etc.).

**How Spark application runs on a cluster?**

At a high level, every Spark application consists of a driver program that launches various parallel operations on a cluster. The driver program contains your application’s main function and defines distributed datasets on the cluster, then applies operations to them.

From the Spark Shell, the Driver Program was the Spark Shell and we could type in operations we want.

Driver programs access Spark through a **SparkContext** object, which represents a connection to a computing cluster. To run these operations, driver programs typically manage a number of nodes called ***executors***.



you can write code in a single driver program and automatically have parts of it run on multiple nodes.

**What is local and App name in the below statement**?

**val** conf **= new SparkConf**().setMaster("local").setAppName("My App")

**val** sc **= new SparkContext**(conf)

* A *cluster URL*, namely local in these examples, which tells Spark how to connect to a cluster. **local** is a special value that runs Spark on one thread on the local machine, without connecting to a cluster.
* An *application name*, namely My App in these examples. This will identify your application on the cluster manager’s UI if you connect to a cluster.

**Why Persist method is needed?**

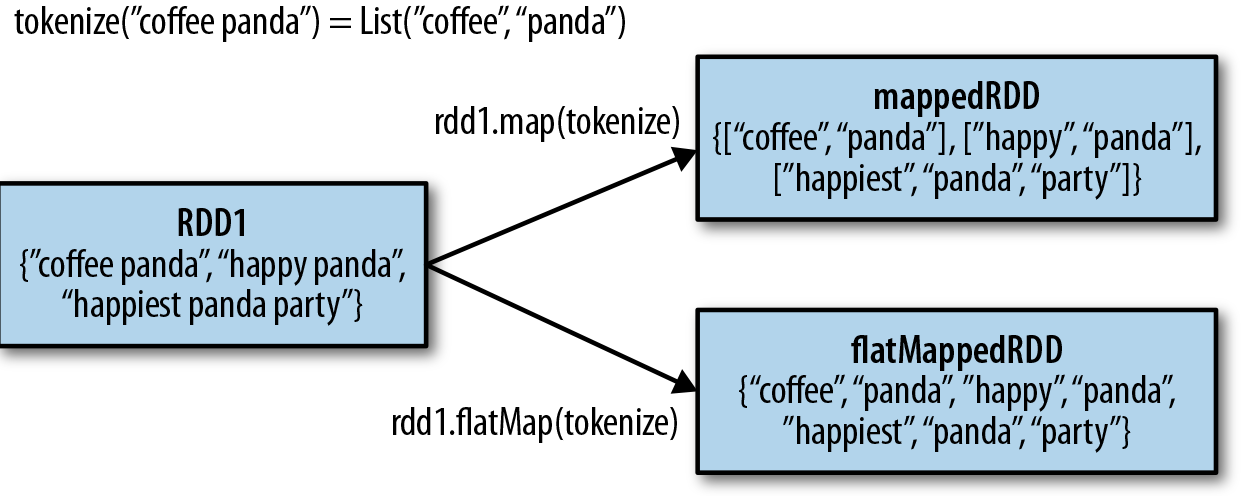
Spark’s RDDs are by default recomputed each time you run an action on them. If you would like to reuse an RDD in multiple actions, you can ask Spark to ***persist***it using **RDD.persist()**. After computing it the first time, Spark will store the RDD contents in memory (partitioned across the machines in your cluster), and reuse them in future actions.

default persist() will store the data in the JVM heap as unserialized objects.

**Why Collect () action should not be used for the large data sets?**

your entire dataset must fit in memory on a driver machine to use collect () on it, so collect () shouldn’t be used on large datasets. For the large data sets, it can through out of memory exception.

**What is the difference between map() and flatmap() transformations.?**



**tokenize is the function .**

Sometimes we want to produce multiple output elements for each input element. The operation to do this is called flatMap(). As with map(), the function we provide to flatMap() is called individually for each element in our input RDD. Instead of returning a single element, we return an iterator with our return values. **Rather than**

**producing an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators**. A simple usage of flatMap() is splitting up an input string into words, as shown in figure above.

**Why RDD.distinct() transformation is expensive operation?**

If we want only unique elements we can use the **RDD.distinct()** transformation to produce a new RDD with only distinct items. Note that **distinct()** is expensive, however, as it requires shuffling all the data over the network to ensure that we receive only one copy of each element.

**What is aggregate () function?**

With aggregate (), like fold (), we supply an initial zero value of the type we want to return**. We then supply a function to combine the elements from our RDD with the accumulator**. Finally, **we need to supply a second function to merge two accumulators, given that each node accumulates its own results locally**.

**val** result **=** input.aggregate((0, 0))(

(acc, value) **=>** (acc.\_1 + value, acc.\_2 + 1),

(acc1, acc2) **=>** (acc1.\_1 + acc2.\_1, acc1.\_2 + acc2.\_2))

**val** avg **=** result.\_1 / result.\_2.toDouble

**What are the different Persistence levels?**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Level** | **Space used** | **CPU time** | **In memory** | **On disk** | **Comments** |
| **MEMORY\_ONLY** | High | Low | Y | N |  |
| **MEMORY\_ONLY\_SER** | Low | High | Y | N |  |
| **MEMORY\_AND\_DISK** | High | Medium | Some | Some | Spills to disk if there is too much data to fit in  memory. |
| **MEMORY\_AND\_DISK\_SER** | Low | High | Some | Some | Spills to disk if there is too much data to fit in  memory. Stores serialized representation in  memory. |
| **DISK\_ONLY** | Low | High | N | Y |  |

If you attempt to cache too much data to fit in memory, Spark will automatically evict old partitions using a **Least Recently Used (LRU) cache policy**. For the memory only storage levels, it will recompute these partitions the next time they are accessed, while for the memory-and-disk ones, it will write them out to disk. In either case, this

means that you don’t have to worry about your job breaking if you ask Spark to cache too much data. However, caching unnecessary data can lead to eviction of useful data and more recomputation time.

Finally, RDDs come with a method called **unpersist**() that lets you manually remove them from the cache.

**What are the transformations on the Pair RDDs?**

***Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})***

|  |  |  |  |
| --- | --- | --- | --- |
| **Function Name** | **Purpose** | **Example** | **Result** |
| **reduceByKey(func)** | Combine values with  the same key. | rdd.reduceByKey(  (x, y) => x + y) | {(1, 2), (3, 10)} |
| **groupByKey()** | Group values with the  same key. | rdd.groupByKey() | {(1, [2]), (3, [4, 6])} |
| **mapValues(func)** | Apply a function to  each value of a pair  RDD without  changing the key. | rdd.mapValues(x => x+1) | {(1, 3), (3, 5), (3,7)} |
| **flatMapValues(func)** | Apply a function that  returns an iterator to  each value of a pair  RDD, and for each  element returned,  produce a key/value  entry with the old  key. Often used for  tokenization. | rdd.flatMapValues(x => (x to 5) | {(1, 2), (1,3), (1,4), (1,  5), (3,4), (3,5)} |
| **keys()** | Return an RDD of just  the keys. | rdd.keys() | {1, 3,3} |
| **values()** | Return an RDD of just  the values. | rdd.values() | {2, 4,6} |
| **sortByKey()** | Return an RDD sorted  by the key. | rdd.sortByKey() | {(1,2), (3,4), (3,6)} |

**What are the transformations on two pair RDDs?**

***(rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})***

|  |  |  |  |
| --- | --- | --- | --- |
| **Function Name** | **Purpose** | **Example** | **Result** |
| **subtractByKey** | Remove elements with a  key present in the other  RDD. | rdd.subtractByKey(other) | **{(1, 2)}** |
| **join** | Perform an inner join  between two RDDs. | rdd.join(other) | {(3, (4, 9)), (3,(6, 9))} |
| **rightOuterJoin** | Perform a join between two RDDs where the key must be present in the first RDD. | rdd.rightOuterJoin(other) | {(3,(Some(4),9)),  (3,(Some(6),9))} |
| **leftOuterJoin** | Perform a join between two RDDs where the key must be present in the other RDD. | rdd.leftOuterJoin(other) | {(1,(2,None)), (3,  (4,Some(9))), (3,  (6,Some(9)))} |
| **cogroup** | Group data from both RDDs  sharing the same key. | **rdd.cogroup(other)** | {(1,([2],[])), (3,  ([4, 6],[9]))} |

**What is combineByKey ()?**

**Will check the details later**

**What is Data Partitioning and show the advantages with a use case?**

**What is Custom Partitioning?**

**How Spark SQL connect to the existing Hive Installation?**

To connect Spark SQL to an existing Hive installation, you need to provide a Hive configuration. You do so by copying your ***hive-site.xml***file to Spark’s ***./conf/***directory.

Once you have done this, you create a **HiveContext** object, which is the entry point to Spark SQL, and you can write Hive Query Language (HQL) queries against your tables to get data back as RDDs of rows.

***Creating a HiveContext and selecting data in Scala***

**import org.apache.spark.sql.hive.HiveContext**

**val** hiveCtx **= new** org.apache.spark.sql.hive.**HiveContext**(sc)

**val** rows **=** hiveCtx.sql("SELECT name, age FROM users")

**val** firstRow **=** rows.first()

println(firstRow.getString(0)) *// Field 0 is the name*

***What are the Spark Best Practices?***

# **Don't use count() when you don't need to return the exact number of rows**

When you don't need to return the exact number of rows use:

DataFrame inputJson = sqlContext.read().json(...);

if (inputJson.take(1).length == 0) {}

instead of

if (inputJson.count() == 0) {}

In RDD you can use **isEmpty()**

def isEmpty(): Boolean = withScope { partitions.length == 0 || take(1).length == 0 }

## Hash-partition before transformation over pair RDD

## Before perform any transformation we should shuffle same key data at the same worker so for that we use Hash-partition to shuffle data and make partition using the key of the pair RDD let see the example of the Hash-Partition data

val wordPairsRDD = rdd.map(word => (word, 1)).

partitonBy(new HashPartition(4))

val wordCountsWithReduce = wordPairsRDD

.reduceByKey(\_ + \_)

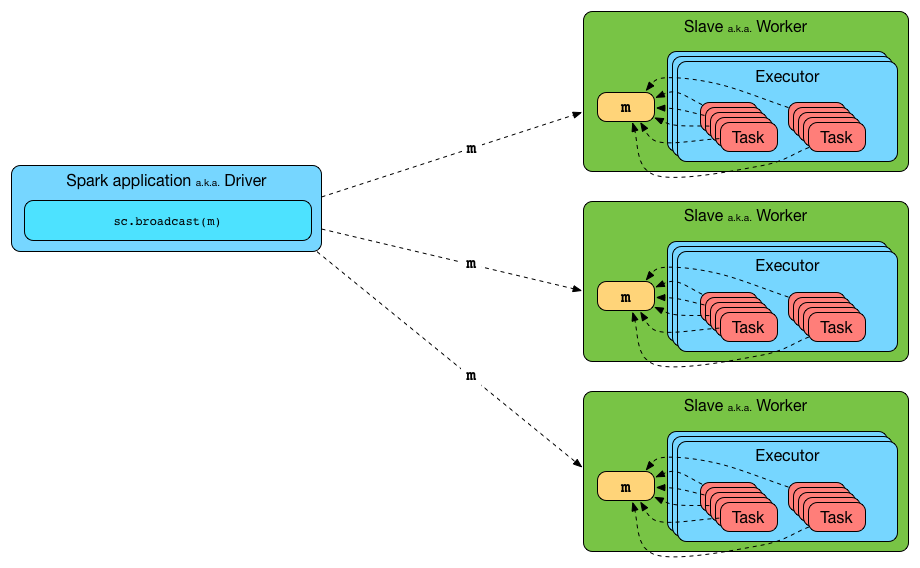
.collect()

## When we are using Hash-partition the data will be shuffle and all same key data will shuffle at same worker, Let see in diagram

## https://umbertogriffo.gitbooks.io/apache-spark-best-practices-and-tuning/content/p1.png

In the above diagram you can see all the data of “c” key will be shuffle at same worker node. So if we use transformation over pair RDD we should use hash-partitioning.

**When to use Broadcast Variable?**



Before running each tasks on the available executors, Spark computes the task’s closure. The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD.

If you have huge array that is accessed from Spark Closures, for example some reference data, this array will be shipped to each spark node with closure.

For example if you have 10 nodes cluster with 100 partitions (10 partitions per node), this Array will be distributed at least 100 times (10 times to each node).  
If you use broadcast it will be distributed once per node using efficient p2p protocol.

val array: Array[Int] = ??? // some huge array

val broadcasted = sc.broadcast(array)

And some RDD

val rdd: RDD[Int] = ???

In this case array will be shipped with closure each time

rdd.map(i => array.contains(i))

and with broadcast you'll get huge performance benefit

rdd.map(i => broadcasted.value.contains(i))

**What is Shuffle?**

The process of moving the data from partition to partition in order to aggregate, join, match up, or spread out in some other way, is known as **shuffling**. The aggregation/reduction that takes place before data is moved across partitions is known as a **map-side shuffle**.

The following operations are examples of shuffle inducing operations for RDDs:

* groupBy/subtractByKey/foldByKey/aggregateByKey/reduceByKey
* cogroup
* any of the join transformations
* distinct

**What is an external shuffle service and how it can optimize shuffling?**

ExternalShuffleService is an **external shuffle service** that serves shuffle blocks from outside an [Executor](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-Executor.html) process. It runs as a standalone application and manages shuffle output files so they are available for executors at all time. As the shuffle output files are managed externally to the executors it offers an uninterrupted access to the shuffle output files regardless of executors being killed or down.

You start ExternalShuffleService using [start-shuffle-service.sh shell script](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-ExternalShuffleService.html#start-script) and enable its use by the driver and executors using [spark.shuffle.service.enabled](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-ExternalShuffleService.html#spark.shuffle.service.enabled).

Enable INFO logging level for org.apache.spark.deploy.ExternalShuffleService logger to see what happens inside.

Add the following line to conf/log4j.properties:

log4j.logger.org.apache.spark.deploy.ExternalShuffleService=INFO

start-shuffle-service.sh shell script allows you to launch ExternalShuffleService. The script is under sbin directory.

|  |  |  |
| --- | --- | --- |
| **Spark Property** | **Default Value** | **Description** |
| **spark.shuffle.service.enabled** | false | Enables [External Shuffle Service](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-ExternalShuffleService.html#ExternalShuffleService). When true, the driver registers itself with the shuffle service.  Used to enable for [dynamic allocation of executors](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-dynamic-allocation.html) and in [CoarseMesosSchedulerBackend](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-mesos/spark-mesos.html#CoarseMesosSchedulerBackend) to instantiate [MesosExternalShuffleClient](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-mesos/spark-mesos.html#MesosExternalShuffleClient).  Explicitly disabled for LocalSparkCluster(and any attempts to set it are ignored). |
| **spark.shuffle.service.port** | 7337 |  |

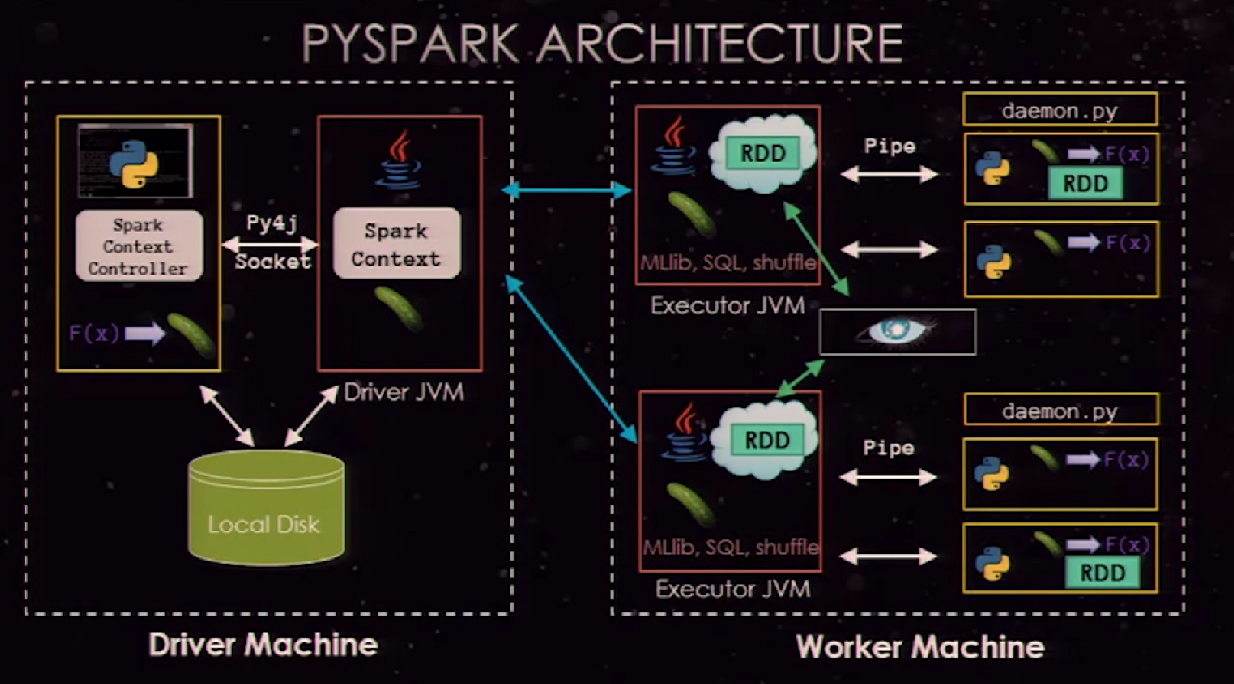
**How combineByKey() function works ?**

If it’s a new element, combineByKey() uses a function we provide, called createCombiner(), to create the initial value for the accumulator on that key. It’s important to note that this happens the first time a key is found in each partition, rather than only the first time the key is found in the RDD.

If it is a value we have seen before while processing that partition, it will instead use the provided function, mergeValue(), with the current value for the accumulator for that key and the new value.

Since each partition is processed independently, we can have multiple accumulators for the same key. When we are merging the results from each partition, if two or more partitions have an accumulator for the same key we merge the accumulators using the user-supplied mergeCombiners() function.

**PySpark Architecture**

****

**What are the different ways to provide the configuration to Spark?**

**1)**The Primary configuration mechanism in spark is the SparkConfclass like below



**2)**Spark allows setting configurations dynamically through the spark-submit tool



3) **spark-submit also supports loading configuration values from a file.**

This can be useful to set environmental configuration, which may be shared across multiple users, such as a default master. By default, spark-submit will look for a file called *conf/*

*spark-defaults.conf* in the Spark directory and attempt to read whitespace-delimited

key/value pairs from this file. You can also customize the exact location of the file using the --properties-file flag to spark-submit

****

****

****

**What is the precedence order for the configuration in the Spark?**

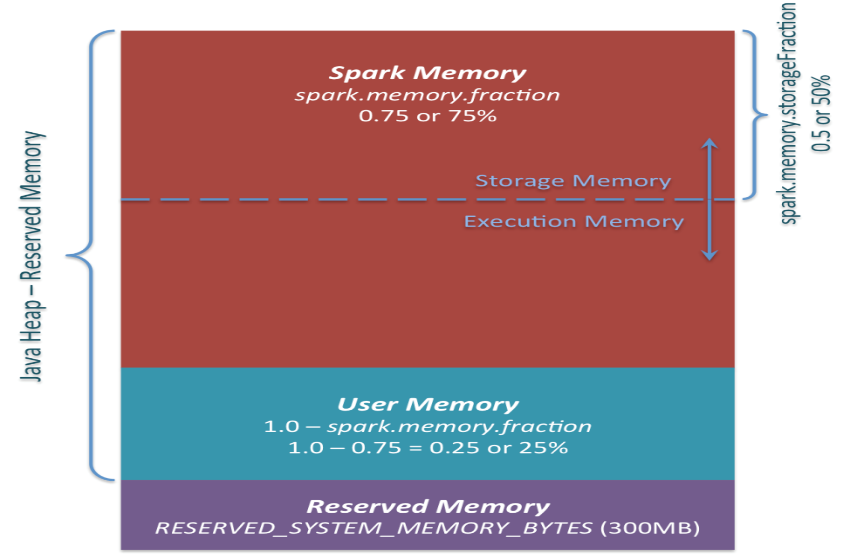
The highest priority is given to configurations declared explicitly in the user’s code using the **set()** function on a **SparkConf** object. Next are flags passed to **sparksubmit**, then values in the **properties file**, and finally default values.

**What is Thread Dump button feature for the Executor at the Spark Web UI?**

**Thread Dump button feature** in the executor’s page is the ability to collect a stack trace from executors. Visualizing the thread call stack of an executor can show exactly what code is executing at an instant in time. If an executor is sampled several times in a short time period with this feature, you can identify “hot spots,” or expensive sections, in user code.

**How Spark Memory Management works?**

Memory management model used in Apache Spark starting version 1.6.0, which is implemented as [UnifiedMemoryManager](https://github.com/apache/spark/blob/branch-1.6/core/src/main/scala/org/apache/spark/memory/UnifiedMemoryManager.scala).



*Apache Spark Unified Memory Manager introduced in v1.6.0+*

3 main memory regions

1. ***Reserved Memory***. This is the memory reserved by the system, and its size is hardcoded. As of Spark 1.6.0, its value is 300MB, which means that this 300MB of RAM does not participate in Spark memory region size calculations, and its size cannot be changed in any way without Spark recompilation or setting *spark.testing.reservedMemory*, which is not recommended as it is a testing parameter not intended to be used in production. Be aware, this memory is only called “reserved”, in fact it is not used by Spark in any way, but it sets the limit on what you can allocate for Spark usage. Even if you want to give all the Java Heap for Spark to cache your data, you won’t be able to do so as this “reserved” part would remain spare (not really spare, it would store lots of Spark internal objects). For your information, if you don’t give Spark executor at least *1.5 \* Reserved Memory = 450MB* heap, it will fail with “please use larger heap size” error message.
2. ***User Memory***. This is the memory pool that remains after the allocation of *Spark Memory*, and it is completely up to you to use it in a way you like. You can store your own data structures there that would be used in RDD transformations. For example, you can rewrite Spark aggregation by using mapPartitions transformation maintaining hash table for this aggregation to run, which would consume so called *User Memory*. In Spark 1.6.0 the size of this memory pool can be calculated as (“*Java Heap*” – “*Reserved Memory*”) \* (1.0 – *spark.memory.fraction*), which is by default equal to (“*Java Heap*” – 300MB) \* 0.25. For example, with 4GB heap you would have 949MB of *User Memory*. And again, this is the *User Memory* and its completely up to you what would be stored in this RAM and how, Spark makes completely no accounting on what you do there and whether you respect this boundary or not. Not respecting this boundary in your code might cause OOM error.
3. **Spark Memory**. Finally, this is the memory pool managed by Apache Spark. Its size can be calculated as (“Java Heap” – “Reserved Memory”) \* spark.memory.fraction, and with Spark 1.6.0 defaults it gives us (“Java Heap” – 300MB) \* 0.75. For example, with 4GB heap this pool would be 2847MB in size. This whole pool is split into 2 regions – Storage Memory and Execution Memory, and the boundary between them is set by spark.memory.storageFraction parameter, which defaults to **0.5**. The advantage of this new memory management scheme is that this boundary is not static, and in case of memory pressure the boundary would be moved, i.e. one region would grow by borrowing space from another one.

**3.1 Storage Memory**. This pool is used for both storing Apache Spark cached data and for temporary space serialized data “unroll”. Also all the “broadcast” variables are stored there as cached blocks. As you may see, it does not require that enough memory for unrolled block to be available – in case there is not enough memory to fit the whole unrolled partition it would directly put it to the drive if desired persistence level allows this. As of “broadcast”, all the broadcast variables are stored in cache with MEMORY\_AND\_DISK persistence level.

**3**.2 **Execution Memory**. This pool is used for storing the objects required during the execution of Spark tasks. For example, it is used to store [shuffle intermediate buffer on the Map side](https://0x0fff.com/spark-architecture-shuffle/) in memory, also it is used to store hash table for hash aggregation step. This pool also supports spilling on disk if not enough memory is available, but the blocks from this pool cannot be forcefully evicted by other threads (tasks).

Note :- Spark doesn’t cache the RDD by default , it caches only the Intermediate results from the stage jobs . To cache the RDD , we need to call cache() or persist() on the RDD .

**What is the difference between coalesce and repartition?**

Coalesce is used for either reducing or increasing the number of partitions.

**The Full method signature:**

**coalesce(numPartitions : Int , shuffle :Boolean = false)**

the second (optional) parameter specifies whether a shuffle should be performed (**false by default)** . If you want to increase the number of partitions, it’s necessary to set the shuffle parameter to true.

The Repartition transformation is just a coalesce with shuffle set to true.

Use coalesce if you decrease number of partition of the RDD instead of repartition. coalesce is useful because it’s not shuffle data over network.

**How we can destroy and unpersist broadcast variables?**

When a broadcast variable is no longer needed, you can destroy it. All information about it will be removed (from the executors and driver), and the variable will become unusable. If you try to access it after calling destroy, an exception will be thrown.

**myVarBroadcasted.destroy()**

Unpersist only removes the variable value from the cache in the executors. if you try to use if after unpersisting , it will sent to the executors again.

**myVarBroadcasted.unpersist(blocking = true)**

**What is RDD lineage?**

Every time a transformation is performed on an RDD, a new vertex (a new RDD) and a new edge (a dependency) are created. the new RDD depends on the old one, so the direction of the edge is from the child RDD to the parent RDD. This graph of dependencies is also called an RDD lineage.

# ***What is Checkpointing in Spark*** *and* [***What is the difference between spark checkpoint and persist to a disk***](https://stackoverflow.com/questions/35127720/what-is-the-difference-between-spark-checkpoint-and-persist-to-a-disk)*?*

RDD lineage can grow arbitrarily long , by chaining any number of transformations . Spark provides a way to persist the entire RDD to stable storage. Then in case of node failure, Spark doesn’t need to recompile the missing RDD pieces from the start. It uses the snapshot and computes the rest of the lineage from there. This feature is called Checkpointing.

During checkpointing , the entire RDD is persisted to disk – not just its data , as is the case with caching , but its lineage too. After checkpointing , the RDD’s dependencies are erased , as well as the information about its parent(s), because they won’t be needed for its recomputuation any more.

You can checkpoint an RDD by calling the checkpoint operation , but first you have to set the directory where the data will be saved by calling **SparkContext.setCheckpointDir() .** This directory is usually an HDFS directory , but it can also be a local one.

Persist/ cache keeps lineage intact while checkpoint breaks lineage.

import org.apache.spark.storage.StorageLevel

val rdd = sc.parallelize(1 to 10).map(x => (x % 3, 1)).reduceByKey(\_ + \_)

* cache / persist:

val indCache = rdd.mapValues(\_ > 4)

indCache.persist(StorageLevel.DISK\_ONLY)

indCache.toDebugString

// (8) MapPartitionsRDD[13] at mapValues at <console>:24 [Disk Serialized 1x Replicated]

// | ShuffledRDD[3] at reduceByKey at <console>:21 [Disk Serialized 1x Replicated]

// +-(8) MapPartitionsRDD[2] at map at <console>:21 [Disk Serialized 1x Replicated]

// | ParallelCollectionRDD[1] at parallelize at <console>:21 [Disk Serialized 1x Replicated]

indCache.count

// 3

indCache.toDebugString

// (8) MapPartitionsRDD[13] at mapValues at <console>:24 [Disk Serialized 1x Replicated]

// | CachedPartitions: 8; MemorySize: 0.0 B; ExternalBlockStoreSize: 0.0 B; DiskSize: 587.0 B

// | ShuffledRDD[3] at reduceByKey at <console>:21 [Disk Serialized 1x Replicated]

// +-(8) MapPartitionsRDD[2] at map at <console>:21 [Disk Serialized 1x Replicated]

// | ParallelCollectionRDD[1] at parallelize at <console>:21 [Disk Serialized 1x Replicated]

* checkpoint:

val indChk = rdd.mapValues(\_ > 4)

indChk.checkpoint

// indChk.toDebugString

// (8) MapPartitionsRDD[11] at mapValues at <console>:24 []

// | ShuffledRDD[3] at reduceByKey at <console>:21 []

// +-(8) MapPartitionsRDD[2] at map at <console>:21 []

// | ParallelCollectionRDD[1] at parallelize at <console>:21 []

indChk.count

// 3

indChk.toDebugString

// (8) MapPartitionsRDD[11] at mapValues at <console>:24 []

// | ReliableCheckpointRDD[12] at count at <console>:27 []

As you can see in the first case lineage is preserved even if data is fetched from the cache. It means that data can be recomputed from scratch if some partitions of indCache are lost. In the second case lineage is completely lost after the checkpoint and indChk doesn't carry an information required to rebuild it anymore.

**What is Speculative Execution of Tasks in Spark?**

If speculative execution is turned on, spark may try to run the slow running task for that partition on some other executor. If that happens and the new task finishes, Spark accepts the result of the new task and discards the old task’s results.

Speculative execution is turned off by default. Turned it on by setting the **spark.speculation** parameter to true. When turned on, spark checks every spark.speculation.interval setting to determine whether any of the tasks need to be restarted.

spark.speculation.quantile determines the percentage of tasks that need to complete before speculation is started for a stage , and spark.speculation.multiplier sets how many times a task needs to run before it needs to be restarted.

**What is Data-locality considerations in Spark and what are the levels of Data locality?**

Data locality means spark tries to run tasks as close to the data location as possible. Spark tries to maintain a list of preferred locations for each partition. A partition’s preferred location is a list of hostnames or executors where the partition’s data resides so that computation can be moved closer to the data.

There are five levels of data locality:

* **PROCESS\_LOCAL** – Execute a task on the executor that cached the partition.
* **NODE\_LOCAL** – Execute a task on the node where the partition is available.
* **RACK\_LOCAL** – Execute the task on the same rack as the partition if the rack information is available in the cluster (currently only on YARN)
* **NO\_PREF** – No Preferred locations are associated with the task.
* **ANY** – Default if everything else fails.

The amount of time the scheduler waits for each locality level before moving to the next is determined by the spark.locality.wait parameter . The default is 30 seconds. You can also set wait times for specific locality levels with spark.locality.wait.process , spark.locality.node and spark.locality.wait.rack.

If any of these parameters is set to 0, the corresponding level is ignored, and tasks won’t be assigned according to that level.

**How we can set the DRIVER MEMORY? How configuration can be set programmatically?**

You can set the memory for your driver with the spark.driver.memory parameter.

We can also set the **SPARK\_DRIVER\_MEMORY** system environment variable in the spark-env.sh file in the <SPARK\_HOME>/conf directory.

We can set Spark configuration parameters directly in the program by using the **SparkConf** class. For example,

**val conf = new org.apache.spark.SparkConf()**

**conf.set(“spark.driver.memory”, “16g”)**

**val sc = new org.apcahe.spark.SparkContext(conf)**

Spark configuration can’t be changed at runtime using this method, so you need to set up the SparkConf object with all the configuration options before creating the SparkContext object.

**How we can view all the configured parameters in the current Spark Context.?**

scala> sc.getConf.getAll.foreach(x => println(x.\_1 + “ : “ + x.\_2)

**How we can disable the Spark Web UI and what is the spark web UI default port and how we can change its port?**

We can disable the Spark Web UI by setting the **spark.ui.enabled** configuration parameter to false . The Default port is **4040** and we can change its port with the **spark.ui.port** parameter.

**What is Spark History Server and Event logging?**

Spark history server and event logging help to debug some strange behavior of the application.

When enabled, Spark logs events necessary for rendering the web UI in the folder specified by spark.eventLog.dir , which is /tmp/spark-events by default.

Event logging can be enabled by setting spark.eventLog.enabled to true.

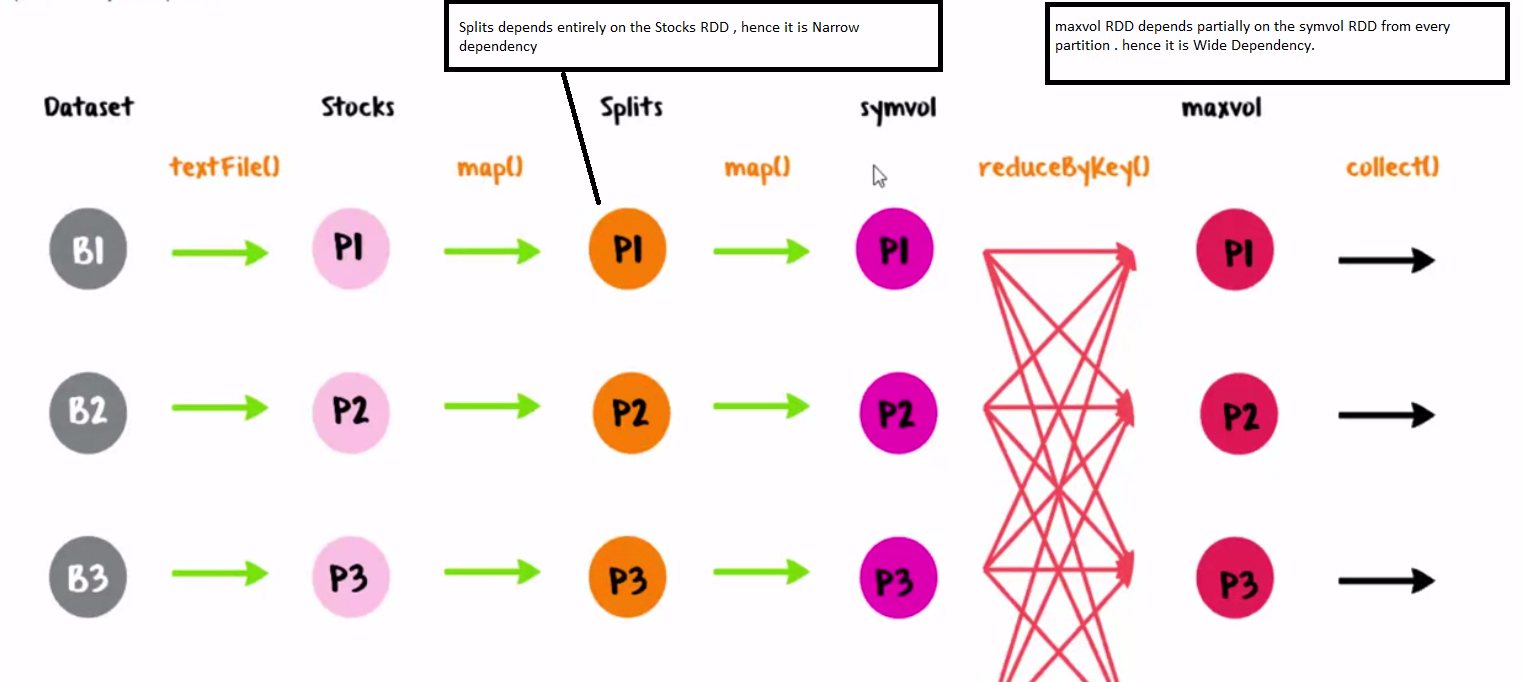
You can start the Spark History server with the script start-history-server.sh in the sbin directory and stop it with stop-history-server.sh . The default HTTP port is 18080 . You can change this with the spark.history.ui.port parameter.

If the application is killed before finishing, it may not appear in the History server UI because the History server expects to find a file name APPLOICATION\_COMPLETE in the application’s directory **(/tmp/spark-events/<application\_id>** by default)

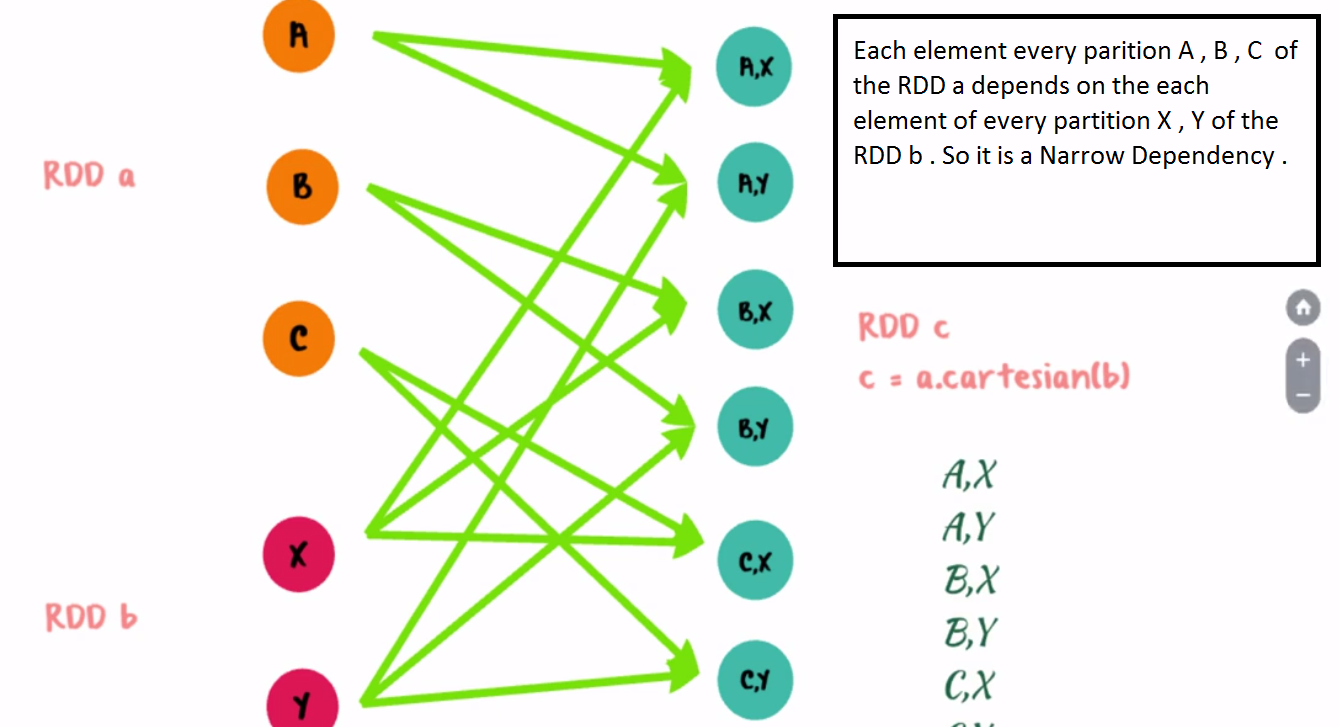
**What is Wide Dependency and Narrow Dependency? important**

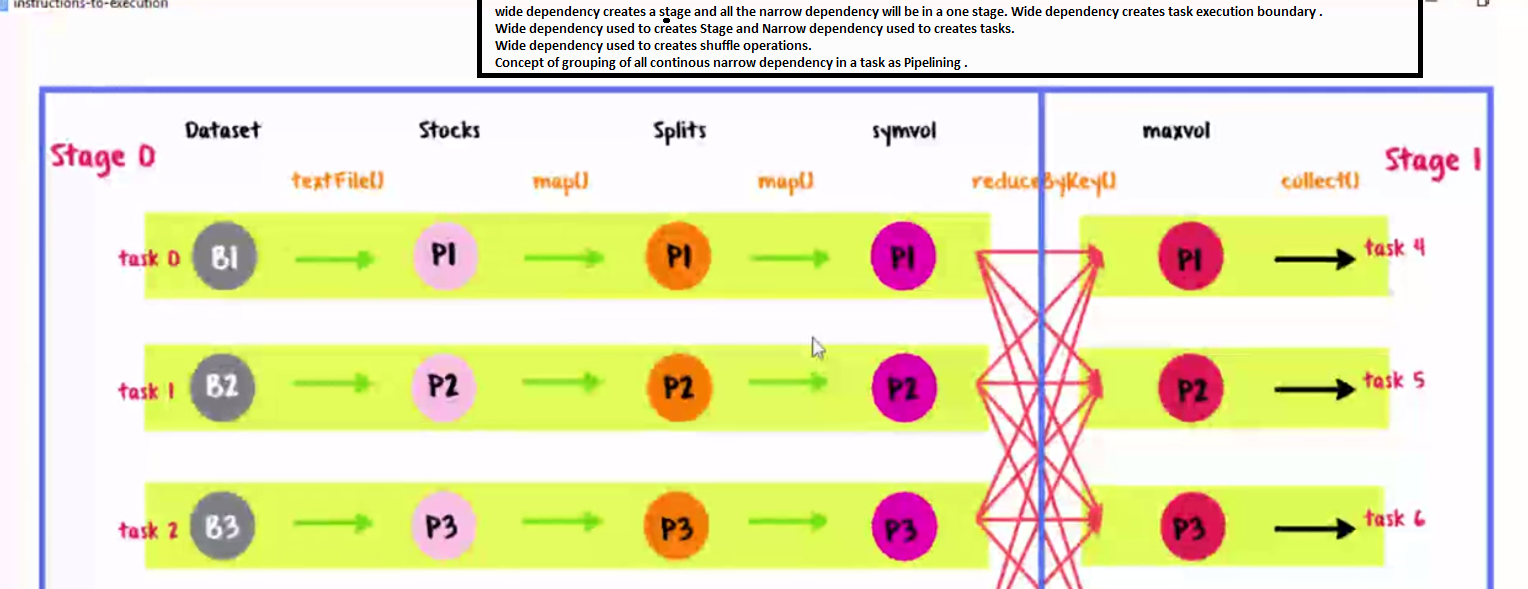
**Narrow Dependency: -** Child partition depends on the entire parent partition.

**Wide Dependency: -** Child partition depends on a portion of each of the parent partitions. Wide dependency always triggers the shuffle operations.

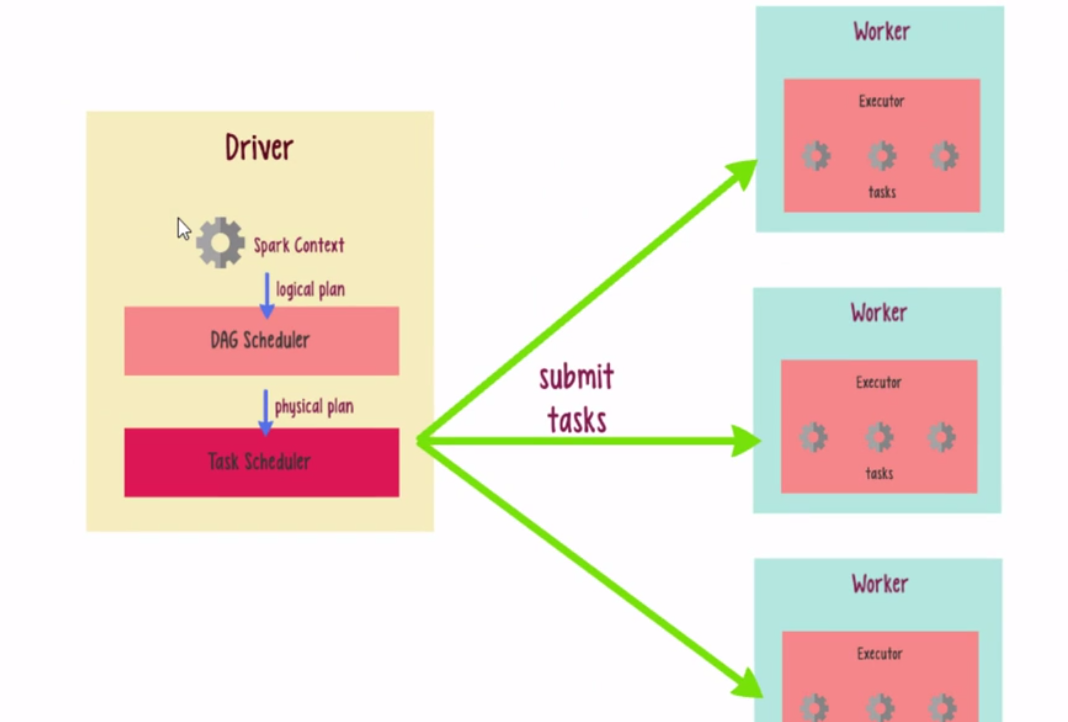
****

You can still have a narrow dependency even when a child partition has more than one parent partition. As long as the child partition depend on entire parent partitions.

****

****

**What happens behind the scene when the Action command is executed?**

****

The Spark Context hands over the logical plan to the DAG Scheduler. The DAG Scheduler takes the logical plan and converts it into the physical plan. The physical plan is the blue print of the execution which has the stages and tasks in each stage involved in the execution. The stages from the physical plan has been submitted to Task Scheduler and each task for the stage has been submitted to the different executor and once all the task has been complete, the result has been sent to the driver.

The DAG Scheduler creates the Job, Stages and Tasks from the logical plan.

The DAG Scheduler and Task Scheduler are local to the Driver.

**How to avoid Data loss in the Spark Streaming?**

Two techniques to deal with fault-tolerance in Spark Streaming: -

1)**Check pointing**: - Checkpoints from Spark Streaming are quite similar to the ones from batch oriented Spark. Both allow to save truncated (without dependencies) RDDs. But Spark Streaming checkpoints has another feature - the second type of checkpoints, called **metadata checkpoint**.

Metadata checkpoint saves information used to launch streaming context into reliable storage as HDFS or S3. Thanks to that, Spark Streaming can recover streaming context for failed driver node. As metadata are considered: streaming application configuration, DStream operations defining the application and not completed but queued batches.

The second type of checkpoint, **data checkpoint**, applies to generated RDDs. As in the case of metadata, they're stored in reliable storage. Data checkpoint is useful in stateful operations where data processed at time t depends on data generated at time t-1, t-2, until t-n where n is the definition of stateful operation's duration (for instance window duration). In this situation, the purpose of checkpoint is to store less data (without dependencies) than in the case of caching.

2)**Write Ahead Logs**: - Spark Streaming also has another protection against failures - a logs journal called **Write Ahead Logs** (WAL). Introduced in Spark 1.2, this structure enforces fault-tolerance by saving all data received by the receivers to logs file located in checkpoint directory. It can be enabled through spark.streaming.receiver.writeAheadLog.enable property.

WAL help to prevent against data loss, for instance in the case when data was received and not processed before driver's failure. It's because data is always written first to ahead logs and only after it's made available for processing. Thus the data is automatically available for reprocessing after streaming context recovery.

Please note that when ahead logs are activated, **cache level shouldn't make a replication**. WAL are already written to fault-tolerant and reliable filesystem, so additional overhead of cache replication is not necessary. Additional condition is the **reliability of receiver**. In fact, it should acknowledge data reception only after be sure to save it into ahead logs.

Logs are saved in receivedBlockMetadata/, located inside checkpoint directory. Files are suffixed by log-. Similarly to checkpoints, old logs are cleaned automatically by Spark. This activity can also be observed in logs:

**Configuration**

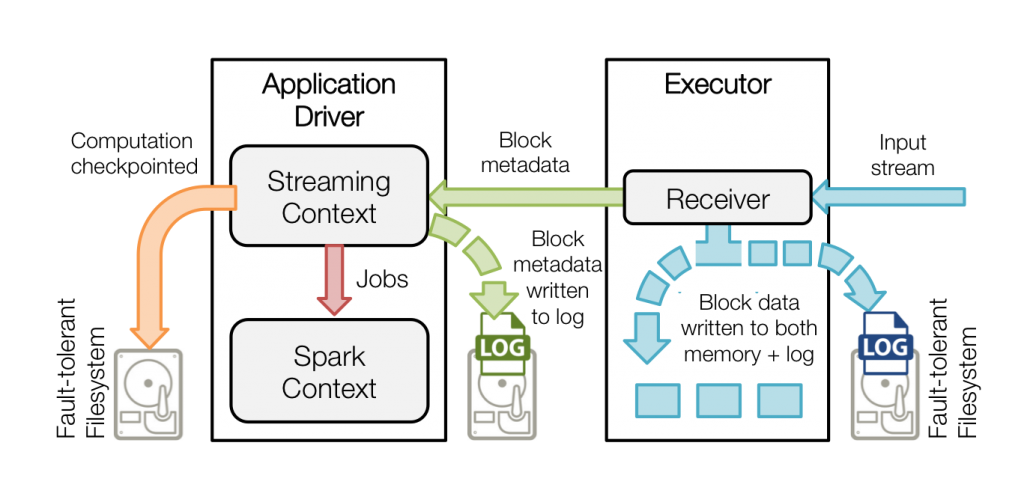
Write ahead logs can be enabled if required by do the following.

* Setting the checkpoint directory using streamingContext.checkpoint(path-to-directory). This directory can be set to any Hadoop API compatible file system, and is used to save both streaming checkpoints as well as write ahead logs.
* [Setting the SparkConf property](http://spark.apache.org/docs/latest/configuration.html#spark-properties) spark.streaming.receiver.writeAheadLog.enable to true (default is false).

**Implementation Details**

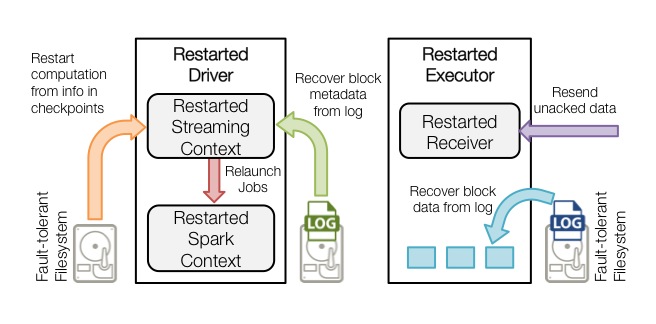
Understand how the write ahead logs work.

When a Spark Streaming application starts (i.e., the driver starts), the associated StreamingContext (starting point of all streaming functionality) uses the SparkContext to launch Receivers as long running tasks. These Receivers receive and save the streaming data into Spark’s memory for processing. The lifecycle of this data received through users is as follows (refer to the diagram below).



* Receiving data (blue arrows) – A Receiver chunks up the stream of data into blocks, that are stored in the memory of the executor. Additionally, if enabled, the data is also written to a write ahead log in a fault tolerant file systems.
* Notifying driver (green arrows) – The metadata of the received blocks are sent to the StreamingContext in the driver. This metadata includes  – (i) reference ids of the blocks for locating their data in the executor memory, (ii) offset information of the block data in the logs (if enabled).
* Processing the data (red arrow) – Every batch interval, the StreamingContext uses the block information to generate RDDs and jobs on them. The SparkContext executes these jobs by running tasks to process the in-memory blocks in the executors.
* Checkpointing the computation (orange arrow) – To recover, the streaming computation (i.e. the DStreams set up with the StreamingContext) is periodically checkpointed to another set of files in the same fault-tolerant file system.

When a failed driver is restart, the following occurs (see the following diagram).



* *Recover computation (orange arrow)* – The checkpointed information is used to restart the driver, reconstruct the contexts and restart all the receivers.
* *Recover block metadata (green arrow)* – The metadata of all the blocks that will be necessary to continue the processing will be recovered.
* *Re-generate incomplete jobs (red arrow)* – For the batches with processing that has not completed due to the failure, the RDDs and corresponding jobs are regenerated using the recovered block metadata.
* *Read the block saved in the logs (blue arrow)* – When those jobs are executed, the block data is read directly from the write ahead logs. This recovers all the necessary data that were reliably saved to the logs.
* *Resend unacknowledged data (purple arrow)* – The buffered data that was not saved to the log at the time of failure will be sent again by the source. as it had not been acknowledged by the receiver.

**How to Import the mysql data to HDFS using sqoop?**

**Steps used by me at cloudera VM**

1.At cloudera VM , install the mySql server from the below command

sudo yum install mysql-server

2. After successful installation, run below command to stop and start the Server.

sudo service mysqld stop

sudo service mysqld start

3.Run below command if you are getting the error (ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO))

mysql –u root -pcloudera (no space between -p and password)

it will open the mySql command prompt

4.Install the MySQL JDBC Driver

-Download the MySQL JDBC driver from <http://www.mysql.com/downloads/connector/j/5.1.html>.

- Extract the JDBC driver JAR file from the downloaded file. For example:

tar zxvf mysql-connector-java-5.1.31.tar.gz

- Copy the connector to /usr/lib/sqoop/lib folder with below command

sudo cp mysql-connector-java-5.1.31/mysql-connector-java-5.1.31-bin.jar /usr/lib/sqoop/lib/

5.Run the below command from the /usr/lib/sqoop path and enter the mysql password

sqoop import --connect jdbc:mysql://localhost/Edureka --table Employee --username root -P --target-dir /sqoopout1 -m 1

6.Check the ouput at the HDFS folder from the UI

http://localhost:50070/explorer.html#/

**How to Export the HDFS data to MySql table .**

Run the below command to export the HDFS data to Mysql table.

sqoop export --connect jdbc:mysql://localhost/Edureka --table Employee --username root -P --export-dir /sqoopout1/ -m 1

Where: -m denotes the number of mapper you want to run

Edureka :- Database name

Employee :- table name

**Sqoop**

**Importing a table**

The following command is used to import the **emp** table from MySQL database server to HDFS.

$ sqoop import \

--connect jdbc:mysql://localhost/userdb \

--username root \

--table emp --m 1

Where: -m denotes the number of mapper you want to run

## Importing into Target Directory

The following command is used to import **emp\_add** table data into ‘/queryresult’ directory.

$ sqoop import \

--connect jdbc:mysql://localhost/userdb \

--username root \

--table emp\_add \

--m 1 \

--target-dir /queryresult

## Import Subset of Table Data

The following command is used to import a subset of **emp\_add** table data. The subset query is to retrieve the employee id and address, who lives in Secunderabad city.

$ sqoop import \

--connect jdbc:mysql://localhost/userdb \

--username root \

--table emp\_add \

--m 1 \

--where “city =’sec-bad’” \

--target-dir /wherequery

## Incremental Import

Incremental import is a technique that imports only the newly added rows in a table. It is required to add ‘incremental’, ‘check-column’, and ‘last-value’ options to perform the incremental import.

$ sqoop import \

--connect jdbc:mysql://localhost/userdb \

--username root \

--table emp \

--m 1 \

--incremental append \

--check-column id \

-last value 1205

The following command is used to see the modified or newly added rows from the **emp** table.

$ $HADOOP\_HOME/bin/hadoop fs -cat /emp/part-m-\*1

# **Sqoop - Import All Tables**

The following command is used to import all the tables from the **userdb**database.

$ sqoop import-all-tables \

--connect jdbc:mysql://localhost/userdb \

--username root

**Note:** If you are using the import-all-tables, it is mandatory that every table in that database must have a primary key field.

# **Sqoop – Export**

To export data back from the HDFS to the RDBMS database. The target table must exist in the target database. The files which are given as input to the Sqoop contain records, which are called rows in table. Those are read and parsed into a set of records and delimited with user-specified delimiter.

The default operation is to insert all the record from the input files to the database table using the INSERT statement. In update mode, Sqoop generates the UPDATE statement that replaces the existing record into the database.

The following command is used to export the table data (which is in **emp\_data** file on HDFS) to the employee table in db database of Mysql database server.

$ sqoop export \

--connect jdbc:mysql://localhost/db \

--username root \

--table employee \

--export-dir /emp/emp\_data

# **Sqoop - Job**

Sqoop job creates and saves the import and export commands. It specifies parameters to identify and recall the saved job. This re-calling or re-executing is used in the incremental import, which can import the updated rows from RDBMS table to HDFS.

**sqoop job --create myjob2 -- import --connect jdbc:mysql://localhost/Edureka --table Employee --username root -P --target-dir /sqoopout2/ --m 1**

## Verify Job (--list)

$ sqoop job --list

**‘--list’** argument is used to verify the saved jobs.

## Inspect Job (--show)

**‘--show’** argument is used to inspect or verify particular jobs and their details.

$ sqoop job --show myjob

## Execute Job (--exec)

**‘--exec’** option is used to execute a saved job.

$ sqoop job --exec myjob

# **Sqoop - Eval**

It allows users to execute user-defined queries against respective database servers and preview the result in the console. So, the user can expect the resultant table data to import. Using eval, we can evaluate any type of SQL query that can be either DDL or DML statement.

**sqoop eval --connect jdbc:mysql://localhost/Edureka --username root --password cloudera --query "SELECT \* FROM Employee"**

## Insert Query Evaluation

**sqoop eval --connect jdbc:mysql://localhost/Edureka --username root --password cloudera --query "INSERT INTO Employee values('pp','bb')"**

**SCALA**

**What Scala is? Why Scala is in demand?**

Scala is a general-purpose programming language that runs on Java Virtual Machine (JVM) and .NET platforms.

It’s a feature-rich language that’s used in various types of applications, starting with building a large messaging

layer for social networking sites such as Twitter to creating an application build tool like SBT (Simple Build Tool). Because of this *scala-bility*, the name of the language is *Scala*.

Scala is one of those rare languages that successfully integrates both object-oriented and functional language

features.

**Who is the Creator of Scala and its History?**

**Martin Odersky** (the creator of Scala) and his team started development on Scala in 2001 in the programming methods laboratory at EPFL (École Polytechnique Fédérale de Lausanne). Scala made its public debut in

**January 2004** on the JVM platform and a few months later on the .NET platform.

**What is Functional Programming?**

*Functional programming* is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

**Is Scala a pure functional language?**

Scala, put simply, is not a pure functional language. In a pure functional language modifications are excluded, and variables are used in a mathematical sense, with identifiers referring to immutable and persistent values. An example of a pure functional language is Haskell.

Scala supports both types of variables: **single-assignment variables** (also called values) that don’t change their value throughout their lifetime and variables that point to a mutable state or could be reassigned to other objects. Even though you should use immutable objects whenever possible, Scala as a language doesn’t provide any restrictions. The restriction is purely conventional. A good rule of thumb is to always default to val and use variables when it’s absolutely necessary.

To me, the fundamental property of a functional language is treating functions as values, and Scala does that well.

**How the below code works in the Scala compiler?**

List(1, 2, 3).map((x: Int) => x + 1)

When the compiler encounters such a call, it replaces the function parameter with an object, as in the following:

List(1, 2, 3).map(new Function1[Int, Int]{ def apply(x:Int): Int = x + 1})

when the Scala compiler encounters functions with one parameter, it replaces that call with an instance of class

scala.Function1, which implements a method called apply.

**How Scala works for the concurrency issues?**

Scala takes a totally different approach to concurrency: the Actor model. An actor is a mathematical model of concurrent computation that encapsulates data, code, and its own thread of control and communicates asynchronously using immutable (no side effects) message-passing techniques. The basic Actor architecture relies on a shared-nothing policy and is lightweight in nature. The Scala Actor model is a better way to handle concurrency issues. Its shared-nothing architecture and asynchronous message-passing techniques make it an easy alternative to existing thread-based solutions.

To make parallel programming easier, Scala provides higher abstractions in the form of a parallel collections library that hides parallel algorithms. For example, to square up each element of a List in parallel, you can use parallel

collections like the following:

**List(1, 2, 3).par.map(x => x \* x)**

In this case the **.par** transforms the List into a parallel collection that implements the map method using a parallel algorithm. Behind the scenes a parallel collections library will fork threads necessary to execute the map method using all the cores available in a given host machine.

**What is println function in scala ?**

println is a function defined in scala.Console, which in turn uses System.out.println to print messages to the console.

**What is :cp , :load , :replay , :type command at the REPL Interpreter?**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **:cp** | Use this command to add a JAR file to the classpath for the Scala interpreter.  For example, :cp tools/junit.jar will try to find a JUnit JAR file relative to your current location and, if found, it will add the JAR file to your classpath so that you can refer to the classes inside the JAR file. |
| **:load or :l** | Allows you to load Scala files into the interpreter. If you want to investigate existing Scala code, you could load the file into the Scala interpreter, and all the definitions will be accessible to you. |
| **:replay or :r** | Resets the interpreter and replays all the previous commands. |
| **:type** | Displays the type of an expression without evaluating it. For example, :type 1 + 2 will determine the type of the expression to Int without performing the add operation. |

**What is multiline string literal in Scala?**

Scala also supports a special multiline string literal that’s enclosed in triple quotes ("""). The sequence of characters is arbitrary, except that it may not contain a triple quote, and it doesn’t even necessarily have to be printable:

scala> val multiLine = """This is a

| multi line

| string"""

multiLine: java.lang.String =

This is a

multi line

string

The output of the multiLine variable has leading whitespaces, and maybe you don’t want that. There’s an easy fix—invoking a method called stripMargin strips out the margin for you:

scala> val multiLine = """This is a

| |multi line

| |string""".**stripMargin**

multiLine: String =

This is a

multi line

string

**What is StringLike in Scala?**

**scala.collection.immutable.StringLike** is the wrapper class at top of java.lang.String

**How we can initialize the variable lazily in scala?**

scala> lazy val forLater = someTimeConsumingOperation()

forLater: Unit = <lazy>

The someTimeConsumingOperation() will be called when the variable forLater is used in any expression

Here’s another example that demonstrates the laziness:

scala> var a = 1

a: Int = 1

scala> lazy val b = a + 1

b: Int = <lazy>

scala> a = 5

a: Int = 5

scala> b

res1: Int = 6

In the last line, typing b forces the evaluation of the b, and because it wasn’t evaluated when b was declared, it uses the latest value of a. **The lazy keyword is allowed only with val; you can’t declare lazy var variables in Scala.**

**What is closure in scala?**

A *closure* is any function that closes over the environment in which it’s defined. For example, closure will keep track of any variable changes outside the function that are being referred to inside the function.

**What is this scala.collection.immutable.$colon$colon?**

If you call the getClass method on myList to see what type of object it is, you might be surprised. Instead of scala.collection.immutable.List, you’ll see

scala> myList.getClass

res42: java.lang.Class[\_] = class

scala.collection.immutable.$colon$colon

That’s because scala.collection.immutable.List is an abstract class, and it comes with two implementations: the scala.Nil class and scala.::. In Scala, :: is a valid identifier, and you could use it to name a class. Nil represents an empty list, and scala.:: represents any nonempty list.

**What happens when parameters are prefixed with var , val and without of these two ?**

When parameters are prefixed with var, Scala creates mutable instance variables. The val and var prefixes are optional, and when both of them are missing, they’re treated as private instance values, not accessible to anyone outside the class:

scala> class MongoClient(host:String, port:Int)

defined class MongoClient

scala> val client = new MongoClient("localhost", 123)

client: MongoClient = MongoClient@4089f3e5

scala> client.host

<console>:7: error: value host is not a member of MongoClient

client.host

**Note that when Scala creates instance values or variables, it also creates accessors for them. At no point in time are you accessing the field directly.** The following Mongo-Client definition is equivalent to the class MongoClient(val host:String, val port:Int)

definition.

class MongoClient(private val \_host:String, private val \_port:Int) {

def host = \_host

def port = \_port

}

**The Scala compiler doesn’t generate accessors by default. What val and var do is define a field and a getter for that field, and in the case of var an additional setter method is also created.**

**How do you add a setter method to a class?**

To add a setter, you have to suffix your setter method with **\_=.** In the following Person class, age is private so I’ll add both a getter and a setter:

class Person(var firstName:String, var lastName:String,

private var \_age:Int) {

def age = \_age

def age\_=(newAge: Int) = \_age = newAge

}

Now you can use the Person class and change its age value:

val p = new Person("Nima", "Raychaudhuri", 2)

p.age = 3

The assignment p.age = 3 could be replaced by p.age\_=(3). When Scala encounters an assignment like x = e, it checks whether there’s any method defined like x\_= and if so, it invokes the method.

**What is the base class for all reference types in scala?**

scala.AnyRef is the base class for all reference types

**What is Package Object in Scala? What is the main use case for the package objects?**

A *package* is a special object that defines a set of member classes and objects. The Scala package lets you segregate code into logical groupings or namespaces so that they don’t conflict with each other.

package com {

package scalainaction {

package mongo {

import com.mongodb.Mongo

class MongoClient(val host:String, val port:Int) {

require(host != null, "You have to provide a host name")

private val underlying = new Mongo(host, port)

def this() = this("127.0.0.1", 27017)

}}}}

**One more interesting point to note here is that Scala package declaration doesn’t have to match the folder structure of your filesystem.** You’re free to declare multiple packages in the same file:

package com.persistence {

package mongo {

class MongoClient

}

package riak {

class RiakClient

}

package hadoop {

class HadoopClient

}

}

If you save this code in a file called Packages.scala and compile it using the Scala compiler (scalac Packages.scala), you’ll notice that the Scala compiler generates class files in appropriate folders to match your package declaration. This ensures that your classes are compatible with the JVM, where package declaration has to match the folder structure in the file system.

With the help of the package object, you can put any kind of definition in a package, such as a class. For example, you can add a helper method in a package object that will be available to all members of the package. Each package is allowed to have one package object.

package object bar {

val minimumAge = 18

def verifyAge = {}

}

minimumAge and verifyAge will be available to all members of the package bar. The

following example uses verifyAge defined inside the package object:

package bar

class BarTender {

def serveDrinks = { verifyAge; ... }

}

The main use case for package objects is when you need definitions in various places inside your package, as well as outside the package when you use the API defined by the package.

**What is \_root\_ package in Scala?**

Consider the following example:

package monads { class IOMonad }

package io {

package monads {

class Console { val m = new monads.IOMonad }

}

}

If you try to compile this code, you’ll get an error saying that type IOMonad isn’t available. That’s because Scala is looking for the IOMonad type in the io.monads package, not in another top-level package called monads. To specify a top-level package you have to use \_root\_:

val m = new \_root\_.monads.IOMonad

**How we can hide a class to use?**

we can hide a class using import with the help of the underscore:

import java.sql.{Date => \_ }

The Date class from the java.sql package is no longer visible for use.

**What is Mixin in scala?**

A *mixin* is a class that provides certain functionality that could be used by other classes like traits.

**What is Class Linearization and what is Linearization Rules?**

Scala solves the **diamond problem** using the class linearization.

Diamond Problem :- The "diamond problem is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have [overridden](https://en.wikipedia.org/wiki/Method_overriding_(programming)), and D does not override it, then which version of the method does D inherit: that of B, or that of C?

**Following are the Linearization Rules?**

* 1. Traits are constructed left to right because a trait on the right is added “later” and thus “overrides” the previous traits.
  2. To construct a trait, its base traits must be constructed first, and of course if a trait is constructed, is not reconstructed again.
  3. No class or trait may appear more than once in the linearization.
  4. The construction order is the reverse of linearization.

Example: -

class A { println(“A”)}

trait H { println(“H”)}

trait S extends H { println(“S”)}

trait R { println(“R”)}

trait T extends R with H { println(“T”)}

class B extends A with T with S {println(“B”)}

new B

. B = L(S) => L(T) => L(A)

. B = S => H => L(T) => L(A)

. B = S => H => T => H => L(R) => L(A)

. B = S => H => T => H => R => L(A)

. B = S => H => T => H => R => A

Now start from right and remove the duplicate construct:

The Output would be :- ARHTSB

**How Trait files got generated on the JVM? (important)**

Depending on how you define a trait, the Scala compiler generates class files differently.

When you define a trait with only a method declaration and without any method body, it produces a Java interface. You could use javap –c <class file name> to inspect class files generated by Scala. For example, trait Empty { def e: Int } will produce the following class file:

public interface Empty{

public abstract int e();

}

When a trait declares concrete methods or code, Scala generates two class files: one for the interface (as shown in the previous code) and a new class file that contains the code. When a class extends a trait, the variables declared in the trait are copied to the class file, and the method defined in the trait becomes a façade method in the class. These façade methods in the class will call the methods defined in the trait code class.

**What is ScalaObject trait?**

ScalaObject trait is a marker (empty) trait. Scala always inserts a trait called scala.ScalaObject as a last mixin in all the classes you create in Scala.

**What is case class and what are the default characteristics of case class?**

scala> case class Person(firstName:String, lastName:String)

defined class Person

In this code example, you’re creating a Person case class with firstName and last-Name parameters. But when you prefix a class with case, the following things will happen automatically:

 Scala prefixes all the parameters with val, and that will make them public value. But remember that you still never access the value directly; you always access through accessors. You’re allowed to prefix the parameters to the case class with var if you want both accessors and mutators. Scala defaults it to val because it encourages immutability.

 Both equals and hashCode are implemented for you based on the given parameters.

 The compiler implements the toString method that returns the class name and its parameters.

 Every case class has a method named copy that allows you to easily create a modified copy of the class’s instance.

 A companion object is created with the appropriate apply method, which takes the same arguments as declared in the class.

 The compiler adds a method called unapply, which allows the class name to be used as an extractor for pattern matching .

 A default implementation is provided for serialization:

When you declare an abstract case class, Scala won’t generate the apply method in the companion object. That makes sense because you can’t create an instance of an abstract class. You can also create case objects that are singleton and serializable:

**How Case class could be used with the Pattern Matching?**

One of the most common reasons for creating case classes is the pattern-matching feature that comes free with

case classes.

Let’s take the Person case class , you’ll extract firstName and lastName from the object using pattern matching:

scala> case class Person(firstName:String, lastName: String)

defined class Person

scala> val p = Person("Matt", "vanvleet")

p: Person = Person(Matt,vanvleet)

scala> p match {

case Person(first, last) => println(">>>> " + first + ", " + last)

}

>>>> Matt, vanvleet

here you’re using a variable pattern in which the matching values get assigned to the first and last variables. Under the hood, Scala handles this pattern matching using a method called unapply. If you have to handcode the companion object that gets generated for Person, it will look like following:

object Person {

def apply(firstName:String, lastName:String) = {

new Person(firstName, lastName)

}

def unapply(p:Person): Option[(String, String)] =

Some((p.firstName, p.lastName))

}

The apply method is simple; it returns an instance of the Person class and it is called when you create an instance of a case class. The unapply method gets called when the case instance is used for pattern matching. Typically, the unapply method is supposed to unwrap the case instance and return the elements (parameters used to create the instance) of the case class.

**What is Sealed traits? (important)**

* sealed traits can only be extended in the same file
* this lets the compiler easily know all possible subtypes
* example: the compiler can emit warnings if a match/case expression is not exhaustive
* use sealed traits when the number of possibly subtypes is finite and known in advance
* they can be a way of creating something like an [enum in Java](https://alvinalexander.com/java/using-java-enum-examples-tutorial)

Examples of things that can be sealed traits are days in the week, months in a year, and the suits in a deck of cards.

sealed trait Answer

case object Yes extends Answer

case object No extends Answer

The compiler will emit a warning if a match is not exhaustive:

scala> val x: Answer = Yes

x: Answer = Yes

scala> x match {

| case No => println("No")

| }

<console>:12: warning: match is not exhaustive!

missing combination Yes

So you should use sealed traits (or sealed abstract class) if the number of possible subtypes is finite and known in advance.

**What is Named arguments?**

When you have methods or class constructors taking similar types of arguments, it becomes difficult to detect

errors if you swap them by mistake. Let’s take the example of Person. Instead of passing in an order of first name, last name, if we swap the order, Scala won’t complain:

scala> case class Person(firstName:String, lastName:String)

defined class Person

scala> val p = Person("lastname", "firstname")

p: Person = Person(lastname,firstname)

Unfortunately, both parameters are of the same type, and the compiler can’t detect the mistake at compile time. But now, using named style arguments, you can avoid the problem:

scala> val p = Person(lastName = "lastname", firstName = "firstname")

p: Person = Person(firstname,lastname)

when you use named arguments, the order doesn’t matter.

**What is private access modifiers?**

The private modifier can be used with any definition, which means it’s only accessible in an enclosed class, its companion object, or a companion class. In Scala, you can qualify a modifier with a class or a package name.

package outerpkg.innerpkg

class Outer {

class Inner {

private[Outer] def f() = "This is f"

private[innerpkg] def g() = "This is g"

private[outerpkg] def h() = "This is h"

}

}

Here, access to the f method can appear anywhere within the Outer class, but not outside it. The method g is accessible anywhere within outerpkg.innerpkg. It’s like the package private in Java. You could use it to make your methods available for unit tests in the same package. Because the h method is qualified with outerpkg, it can appear anywhere within outerpkg and its subpackages.

Scala also lets you qualify the private modifier with this: private[this]. In this case, it means object private. And object private is only accessible to the object in which it’s defined. When members are marked with private without a qualifier, they’re called *class-private*.

What is Value classes?

What is Implicit conversion with implicit classes?

Implicit conversion is a method that takes one type of parameter and returns another type.

The advantage of implicit conversion is that the compiler will find the appropriate implicit conversion and invoke it for you:

scala> val someInt: Int = 2.3

<console>:7: error: type mismatch;

found : Double(2.3)

required: Int

val someInt: Int = 2.3

^

scala> implicit def double2Int(d: Double): Int = d.toInt

warning: there were 1 feature warnings; re-run with -feature for details

double2Int: (d: Double)Int

scala> val someInt: Int = 2.3

someInt: Int = 2

The first time we assigned a Double value to an Int type variable it failed, but it succeeds the second time. When the compiler encounters a type error, it doesn’t immediately give up; instead, it looks for any implicit conversions that might fix the error. In this case, double2Int is used to convert Double to Int. The last

line will be rewritten by the compiler:

val someInt: Int = double2Int(2.3)

This conversion happens at compile time, and, if no appropriate conversion method is found, the compiler throws a compilation error. The compiler will also throw an error if there is ambiguity in an implicit resolution; for example, more than one implicit conversion is found that matches the given criteria.

**What is Option in Scala?**

Option is a collection type in Scala. Unlike other collection types, an Option contains a maximum of one element. It represents one of two possible values: None and Some. None means “no value” and Some represents “some value.” By returning Option from a method, you can communicate that the method might not return a value at all times.

**How we can build a function that works for all type of lists?**

Using type parameterization, we can make the type information configurable.

Here’s how the position function looks with type parameterization:

def position[A](xs: List[A], value: A): Int = {

xs.indexOf(value) <--- **Find index of first element occurrence**

}

Here A denotes a type that could only be determined when the function is invoked. Both List and the value you’re looking for need to be of the same type.

When this function is invoked with a list of integers, A represents the type Int. Similarly if you invoke it with a list of strings, the type parameter A is replaced with String.

scala> val xs = List("one", "two", "three")

xs: List[java.lang.String] = List(one, two, three)

scala> position(xs, "two")

res2: Int = 1

scala> val ys = List(20, 30, 40)

ys: List[Int] = List(20, 30, 40)

scala> position(ys, 40)

res6: Int = 2

scala> position[Int](ys, 300)

res11: Int = -1

**What is covariance and contravariance in scala?**

[**http://julien.richard-foy.fr/blog/2013/02/21/be-friend-with-covariance-and-contravariance/**](http://julien.richard-foy.fr/blog/2013/02/21/be-friend-with-covariance-and-contravariance/)

**Covariance**

**Contravariance**

**What is the usefulness of Nothing in Scala?**

In Scala all methods and functions need to have a return type. There’s no void in Scala. In situations where you have methods that don’t return anything, you use scala.Unit as a return type of the method. Scala uses scala.Nothing as a return type in situations where the return type isn’t clear, such as when an exception is thrown:

scala> def throwException = throw new RuntimeException(

"Always throws exception")

throwException: Nothing

When you see a method returning Nothing, that means that method won’t return successfully.

Here’s another example of invoking exit (System.exit) to terminate the runtime:

scala> def kill = sys.exit(1)

kill: Nothing

**What is Call-by-value and Call-by-name method invocation in Scala?**

**Call-by-value** :- It evaluates every function argument only once.

**Call-by-name** :- A function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body.

1)**A Normal scala method ( call-by-value)**

Object Test extends APP {

def time() : Long {

println(“In time()”)

system.nanoTime

}

def exec(t:Long) : Long = {

println(“Entered exec , calling t …”)

println(“t = “ + t)

println(“calling t again …”)

t

}

Println(exec(time()))

}

The output look like this :

In time()

Entered exec , calling t …

t= 1363909521286596000

calling t again …

1363909521286596000

As expected, the values for t are the same. This is because the value of t is determined when the below line is invoked .

Println(exec(time()))

2)**A “call-by-name” example**

Object Test extends APP {

def time() : Long {

println(“In time()”)

system.nanoTime

}

def exec(t :=> Long) : Long = {

println(“Entered exec , calling t …”)

println(“t = “ + t)

println(“calling t again …”)

t

}

Println(exec(time()))

}

The output look like this :

Entered exec , calling t …

In time()

t= 1363909521286596000

calling t again …

13639095212868008000

The two t invocations yield different results. This is because

“The argument is not evaluated at the point of function application, but instead is evaluated at each use within the function. That is, the argument is evaluated using call-by-name.”

* + Scala normally uses call-by-value.
  + But if the type of a fuction parameter starts with => it uses call-by-name

Another example

def custone(x:Int , y => Int ) = 1

custone(1+2 , loop) custone(loop , 1+2)

= =

Custone(3,loop) custone(loop , 1+2)

= =

1 Infinite loop

**How does head :: tail work?**

This pattern is called the *Infix Operation* pattern, where the expression head :: tail is shorthand for the constructor pattern ::(head, tail). The immutable List in Scala is defined as an abstract sealed class, and its only two subclasses are Nil and :: . Both are case classes. As you already know, one of the biggest benefits of Scala case classes is that they can participate in pattern matching. So ::(head,tail) matches the constructor of the :: case class that takes head as the first element

and the list as the second element, which in this case is tail.

**What is the difference between foldLeft and foldRight methods in Scala?**

**Will check again.**

**What are Lazy Collections : Views and Streams ?**

The more technical term for on-demand collections is *nonstrict collections.* Sometimes nonstrict collections are called *lazy collections.*

**Views**

Almost all collections expose a method called **view**, which returns a nonstrict view of the collection you’re working on.

scala> List(1, 2, 3, 4, 5).view.map( \_ + 1).head

res46: Int = 2

In this case, a call to map produces another view without doing the calculation, and the calculation is deferred until you invoke head on it.

**Streams**

The class Stream implements lazy lists in Scala where elements are evaluated only when they’re needed. Stream is like List, in which elements are stored in two parts, head and tail. The tail for Stream isn’t computed until it’s needed. If you want, you can build an infinite list in Scala using Stream, and it will consume memory based on your use.

Because Stream extends from LinearSeq, you have almost all the List methods available to you. The following example zips each element of the List with its index:

scala> List("zero", "one", "two", "three", "four", "five").zip(Stream.from(0))

res88: List[(java.lang.String, Int)] = List((zero,0), (one,1), (two,2),

(three,3), (four,4), (five,5))

In mathematics, the Fibonacci numbers are the numbers in the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

One way to implement the Fibonacci sequence is to build it using an infinite stream. If you take a Fibonacci sequence starting from 0 (0, 1, 1, 2, 3...) and zip with its tail, you get a sequence of pairs (tuples): ((0, 1), (1,

2), (2, 3)...). If you use map to sum the two parts of the pair, you get a number in the Fibonacci series. To implement this, use recursive streams. Here’s the code snippet:

val fib: Stream[Int] = Stream.cons(0, Stream.cons(1,fib.zip(fib.tail).map(t => t.\_1 + t.\_2)))

You’re using the apply method of the cons object with a head of 0 and another

Stream as a tail.

The question that remains is how Stream can manage to evaluate the tail when it’s

required and not eagerly evaluated. The answer is a call-by-name parameter.