Notes unit4 Big Data - Big data

B.tech (Dr. A.P.J. Abdul Kalam Technical University)

Scan to open on Studocu

Unit 4
Big Data

Hadoop Eco System and YARN:

Hadoop Ecosystem components

- Hadoop Ecosystem is a platform or a suite that provides various services to solve big data problems.
- It includes Apache projects and various commercial tools and solutions.
- There are four major elements of Hadoop i.e. HDFS, MapReduce, YARN, and Hadoop Common.
- Most of the tools or solutions are used to supplement or support these major elements.
- All these tools work collectively to provide services such as absorption, analysis, storage and maintenance of data etc.

Following are the components that collectively form a Hadoop ecosystem:

- HDFS: Hadoop Distributed File System
- YARN: Yet Another Resource Negotiator
- MapReduce: Programming based Data Processing
- Spark: In-Memory data processing
- PIG, HIVE: Query-based processing of data services
- HBase: NoSQL Database
- Mahout, Spark MLLib: Machine Learning algorithm libraries
- Solar, Lucene: Searching and Indexing
- Zookeeper: Managing cluster
- Oozie: Job Scheduling

Hadoop-Schedulers

1. Fifo scheduler

- As the name suggests FIFO i.e. First In First Out, therefore the tasks or application that comes first are going to be served first.
- This is the default Schedular we use in Hadoop.
- The tasks are placed during a queue and therefore the tasks are performed in their submission order.
- In this method, once the work is scheduled, no intervention is allowed.
- So sometimes the high priority process has got to wait an extended time since the priority of the task doesn't matter during this method.

2. Capacity schedulers

- In Capacity Scheduler we've multiple job queues for scheduling our tasks.
- The Capacity Scheduler allows multiple occupants to share an outsized size Hadoop cluster.
- In Capacity Schedular corresponding for every job queue, we offer some slots or cluster resources for performing job operations.
- Each job queue has its own slots to perform its task. just in case we've tasks to perform in just one queue then the tasks of that queue can access the slots of other queues also as they're liberal to use, and when the new task enters to another queue then jobs in running in its own slots of the cluster are replaced with its own job.
- Capacity Scheduler also provides A level of abstraction to understand which occupant is utilizing the more cluster resource or slots, so that the only user or application doesn’t take disappropriate or unnecessary slots within the cluster.
- The capacity Schedular mainly contains 3 sorts of the queue that are root, parent, and leaf which are wont to represent cluster, organization, or any subgroup, application submission respectively.

3. Fair Scheduler

- The Fair Scheduler is very much similar to that of the capacity scheduler. The

priority of the job is kept in consideration.

- With the help of Fair Scheduler, the YARN applications can share the resources in the large Hadoop Cluster and these resources are maintained dynamically so no need for prior capacity.
- The resources are distributed in such a manner that all applications within a cluster get an equal amount of time.
- Fair Scheduler takes Scheduling decisions based on memory, we can configure it to work with CPU also.
- As we told you it is similar to Capacity Scheduler but the major thing to notice is that in Fair Scheduler whenever any high priority job arises in the same queue, the task is processed in parallel by replacing some portion from the already dedicated slots.

Hadoop 2.0 New Features-NameNode high availability

- High Availability was a new feature added to Hadoop 2.x to solve the Single point of failure problem in the older versions of Hadoop.
- As the Hadoop HDFS follows the master-slave architecture where the NameNode is the master node and maintains the filesystem tree. So HDFS cannot be used without NameNode.
- This NameNode becomes a bottleneck. HDFS high availability feature addresses this issue.
- Before Hadoop 2.0.0, the NameNode was a single point of failure (SPOF) in an HDFS cluster.
- Each cluster had a single NameNode, and if NameNode fails, the cluster as a
- whole would be out of services.
- The cluster will be unavailable until the NameNode restarts or brought on a separate machine.
- Hadoop 2.0 overcomes this SPOF by providing support for many NameNode.
- HDFS NameNode High Availability architecture provides the option of running two redundant NameNodes in the same cluster in an active/passive configuration with a hot standby.

- **Active NameNode** – It handles all client operations in the cluster.
- **Passive NameNode** – It is a standby namenode, which has similar data as active NameNode. It acts as a slave, maintains enough state to provide a fast failover, if necessary.

If Active NameNode fails, then passive NameNode takes all the responsibility of active node and the cluster continues to work.

Issues in maintaining consistency in the HDFS High Availability cluster are as follows:

- Active and Standby NameNode should always be in sync with each other, i.e. they should have the same metadata. This permit reinstating the Hadoop cluster to the same namespace state where it got crashed. And this will provide us to have fast failover.
- There should be only one NameNode active at a time. Otherwise, two NameNode will lead to corruption of the data. We call this scenario a "**Split-Brain Scenario**", where a cluster gets divided into the smaller cluster. Each one believes that it is the only active cluster. "**Fencing**" avoids such scenarios. Fencing is a process of ensuring that only one NameNode remains active at a particular time.

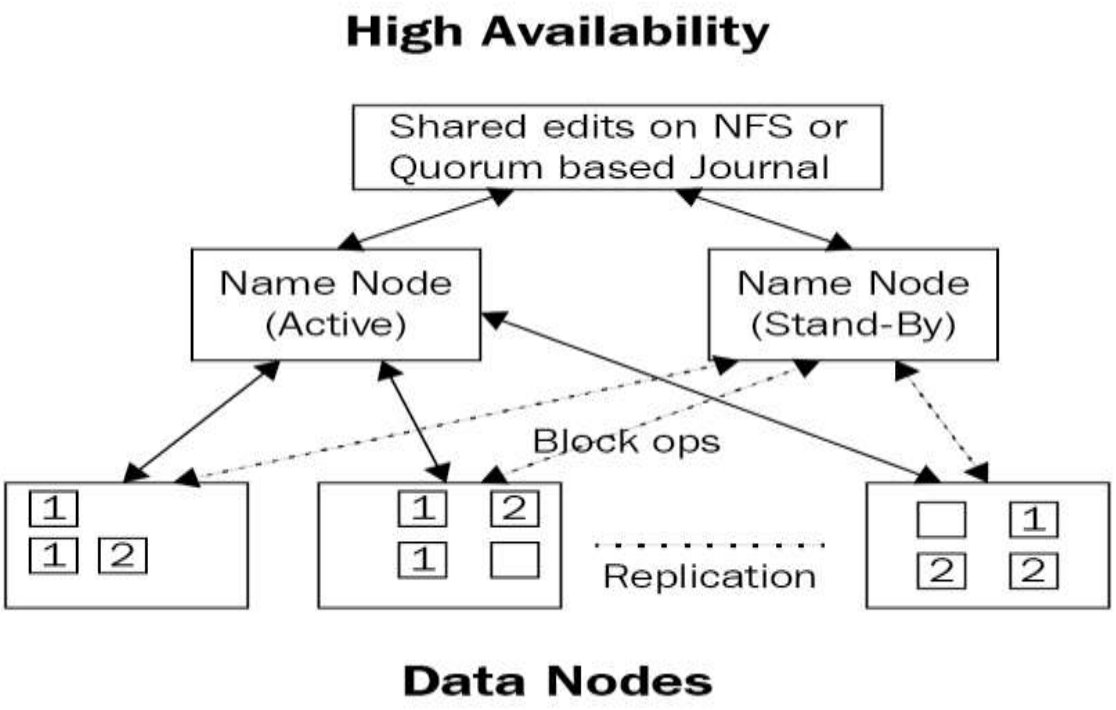
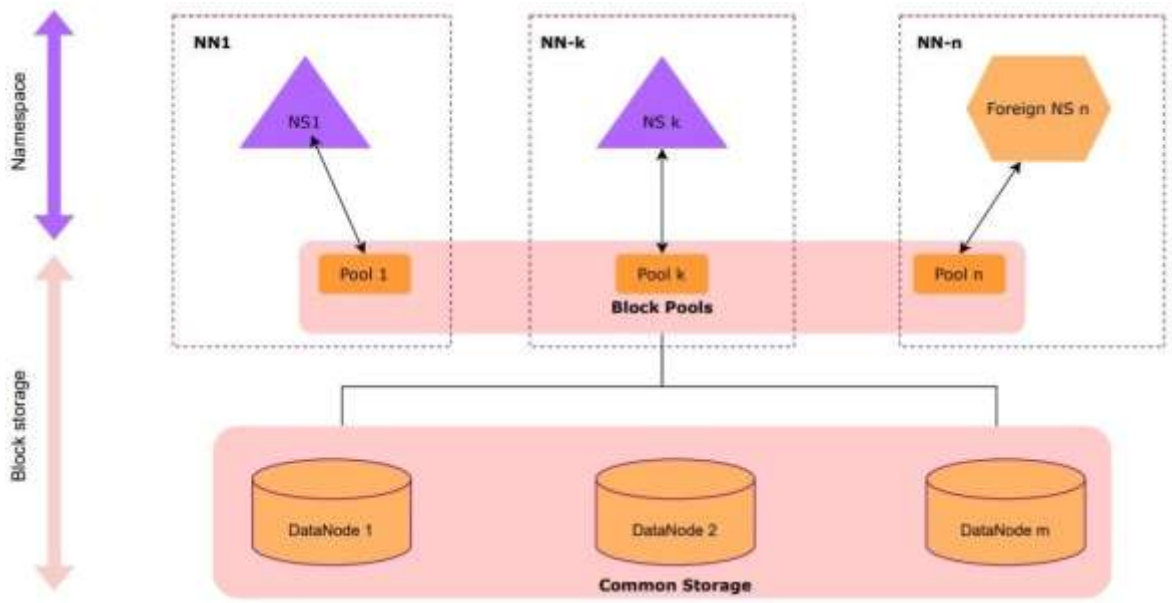


Fig: Name Node-high availability

HDFS Federation

HDFS Federation improves the existing HDFS architecture through a clear separation of namespace and storage, enabling a generic block storage layer. It enables support for multiple namespaces in the cluster to improve scalability and isolation. Federation also opens up the architecture, expanding the applicability of HDFS clusters to new implementations and use cases.

- HDFS Federation **enhances an existing HDFS architecture**
- HDFS Federation allows **more than one NameNode in a cluster.** Each NameNode has its own NameSpace. The **DataNodes are common** among all the NameNodes.
- The namenodes are federated, that is, the namenodes are independent and don't require coordination with each other.
- The datanodes are used as common storage for blocks by all the namenodes. Each datanode registers with all the namenodes in the cluster
- Datanodes send periodic heartbeats and block reports and handle commands from the namenodes.
- A Block Pool is a set of blocks that belong to a single namespace. Datanodes store blocks for all the block pools in the cluster.
- It is managed independently of other block pools. This allows a namespace to generate Block IDs for new blocks without the need for coordination with the other namespaces.
- The failure of a namenode does not prevent the datanode from serving other namenodes in the cluste
- A Namespace and its block pool together are called Namespace Volume. It is a self- contained unit of management.



Overview of Current HDFS

HDFS has two main layers:

- **Namespace** manages directories, files and blocks. It supports file system operations such as creation, modification, deletion and listing of files and directories.
- **Block Storage** has two parts:
  - **Block Management** maintains the membership of datanodes in the cluster. It supports block-related operations such as creation, deletion, modification and getting location of the blocks. It also takes care of replica placement and replication.
  - **Physical Storage** stores the blocks and provides read/write access to it.

MRv2

- Earlier version of map- reduce framework in Hadoop 1.0 is called as MRv1.
- The new version of MapReduce in Hadoop 2.0 is known as MRv2.
- The MRv1 framework includes client communication, job execution and management, resource scheduling and resource management
- MRv2 (aka YARN, "Yet Another Resource Negotiator") has a Resource Manager for each cluster, and each data node runs a Node Manager.

- For each job, one slave node will act as the Application Master, monitoring resources/tasks, etc.
- The Hadoop daemons associated with MRv1 are JobTracker and TaskTracker as shown in the following figure:

YARN

YARN stands for "Yet Another Resource Negotiator".

It was introduced in Hadoop 2.0 to remove the bottleneck on Job Tracker which was present in Hadoop 1.0. YARN was described as a "Redesigned Resource Manager" at the time of its launching

But it has now evolved to be known as a large-scale distributed operating system used for Big Data processing.

YARN also allows different data processing engines like graph processing, interactive processing, stream processing as well as batch processing to run and process data stored in HDFS (Hadoop Distributed File System) thus making the system much more efficient.

Through its various components, it can dynamically allocate various resources and schedule the application processing.

For large volume data processing, it is quite necessary to manage the available resources properly so that every application can leverage them

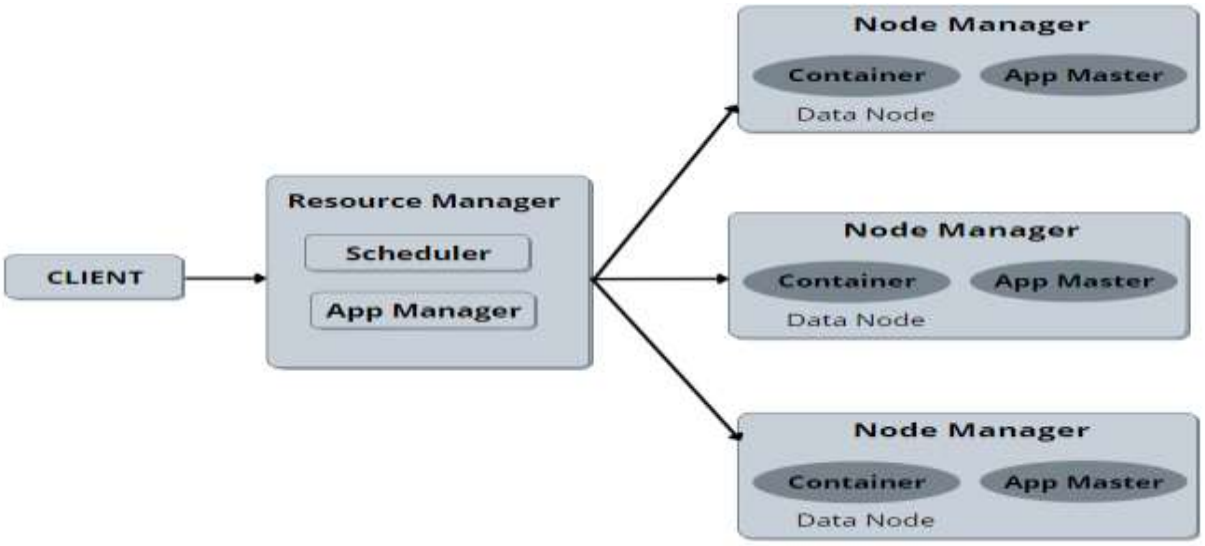


Fig: YARN Architecture

Running MRv1 in YARN.

- YARN uses the ResourceManager web interface for monitoring applications running on a YARN cluster.
- The ResourceManager UI shows the basic cluster metrics, list of applications, and nodes associated with the cluster.
- In this section, we'll discuss the monitoring of MRv1 applications over YARN.
- The Resource Manager is the core component of YARN – Yet Another Resource Negotiator. In analogy, it occupies the place of JobTracker of MRV1.
- Hadoop YARN is designed to provide a generic and flexible framework to administer the computing resources in the Hadoop cluster.
- In this direction, the YARN Resource Manager Service (RM) is the central controlling authority for resource management and makes allocation decisions ResourceManager has two main components: Scheduler and ApplicationsManager.

NoSQL Databases:

Introduction to NoSQL

- A term for any type of database that does not use SQL for the primary retrieval of data from the database.
- NoSQL databases have limited traditional functionality and are designed for scalability and high performance retrieve and append.
- Typically, NoSQL databases store data as key-value pairs, which works well for data that is unrelated.
- NoSQL databases can store relationship data—they just store it differently than relational databases do.
- When compared with SQL databases, many find modelling relationship data in NoSQL databases to be easier than in SQL databases, because related data doesn't have to be split between tables.
- NoSQL data models allow related data to be nested within a single data structure.

Here are the four main types of NoSQL database

- Document databases.
- Key-value stores.
- Column-oriented databases.
- Graph databases.

Document databases.

- A document database stores data in JSON, BSON, or XML documents (not Word documents or Google docs, of course).
- In a document database, documents can be nested. Particular elements can be indexed for faster querying.
- Documents can be stored and retrieved in a form that is much closer to the data objects used in applications, which means less translation is required to use the data in an application.
- SQL data must often be assembled and disassembled when moving back and forth between applications and storage.

- **Key-value stores**
  - Key-value databases are a simpler type of database where each item contains keys and values.
  - A value can typically only be retrieved by referencing its key, so learning how to query for a specific key-value pair is typically simple.
  - Key-value databases are great for use cases where you need to store large amounts of data but you don't need to perform complex queries to retrieve it.
  - Common use cases include storing user preferences or caching. Redis and DynamoDB are popular key-value databases.

- **Column-oriented databases**
  - Wide-column stores store data in tables, rows, and dynamic columns.
  - Wide-column stores provide a lot of flexibility over relational databases because each row is not required to have the same columns.
  - Many consider wide-column stores to be two- dimensional key-value databases. Wide-column stores are great for when you need to store large

amounts of data and you can predict what your query patterns will be.

- Wide-column stores are commonly used for storing Internet of Things data and user profile data. Cassandra and HBase are two of the most popular wide-column stores.

- **Graph databases**

  - Graph databases store data in nodes and edges.
  - Nodes typically store information about people, places, and things while edges store information about the relationships between the nodes.
  - Graph databases excel in use cases where you need to traverse relationships to look for patterns such as social networks, fraud detection, and recommendation engines.
  - Neo4j and JanusGraph are examples of graph databases.

MongoDB:

Introduction:

MongoDB is an open-source document database and leading NoSQL database. MongoDB is written in C++. This tutorial will give you a great understanding of MongoDB concepts needed to create and deploy a highly scalable and performance-oriented database.

Data Types:

MongoDB supports many data types. Some of them are −

- String − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- Integer − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- Boolean − This type is used to store a boolean (true/ false) value.
- Double − This type is used to store floating-point values.
- Min/ Max keys − This type is used to compare a value against the lowest and highest BSON elements.
- Arrays − This type is used to store arrays or lists or multiple values into one key.
- Timestamp − timestamp. This can be handy for recording when a document has

been modified or added.

- Object − This data type is used for embedded documents.
- Null − This type is used to store a Null value.
- Symbol − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- Date − This data type is used to store the current date or time in UNIX time format. You can specify your own date time by creating an object of Date and passing day, month, a year into it.
- Object ID − This data type is used to store the document's ID.
- Binary data − This data type is used to store binary data.
- Code − This data type is used to store JavaScript code into the document.
- Regular expression − This data type is used to store regular expression.

CRUD Operation

- MongoDB provides a set of some basic but most essential operations that will help you to easily interact with the MongoDB server and these operations are called as CRUD operations

  Crud is

  - C - create
  - r - read
  - u - update
  - d – delete

Creating Document:

The create or insert operations are used to insert or add new documents in the collection

Method	Description
db.collection.insertOne()	It is used to insert a single document in the collection.
db.collection.insertMany()	It is used to insert multiple documents in the collection.
db.createCollection()	It is used to create an empty collection.

.

Read Operations –

The Read operations are used to retrieve documents from the collection, or in other words, read operations are used to query a collection for a document.

Method	Description
db.collection.find()	It is used to retrieve documents from the collection.

Updating Document

The update operations are used to update or modify the existing document in the collection

Method	Description
db.collection.updateOne()	It is used to update a single document in the collection that satisfy the given criteria.
db.collection.updateMany()	It is used to update multiple documents in the collection that satisfy the given criteria.
db.collection.replaceOne()	It is used to replace single document in the collection that satisfy the given criteria.

Deleting Documents

The update operations are used to update or modify the existing document in the collection.

Method	Description
db.collection.updateOne()	It is used to update a single document in the collection that satisfy the given criteria.
db.collection.updateMany()	It is used to update multiple documents in the collection that satisfy the given criteria.
db.collection.replaceOne()	It is used to replace single document in the collection that satisfy the given criteria.

Querying:

find() Method

To query data from MongoDB collection, you need to use MongoDB's find() method.

Syntax

The basic syntax of find() method is as follows –

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non-structured way.

pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

```
>db.COLLECTION_NAME.find().pretty()
```

findOne() method

Apart from the find() method, there is findOne() method, that returns only one document.

Syntax

```
>db.COLLECTIONNAME.findOne()
```

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND −

```
>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })
```

OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use $or keyword. Following is the basic syntax of OR −

```
db.mycol.find(
  {
    $or: [
      {key1: value1}, {key2:value2}
    ]
  }
).pretty()
```

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use the $not keyword. Following is the basic syntax of NOT −

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use the $not keyword following is the basic syntax of NOT −

```
db.COLLECTION_NAME.find(
        {
```

```
                $not: [
                        {key1: value1}, {key2:value2}
                ]
        }
)
```

Introduction to indexing

- Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement.
- If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.
- Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.
- The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations.

In addition, MongoDB can return sorted results by using the ordering in the index.

MongoDB supports indexes

- At the collection level
- Similar to indexes on

RDBMS Can be used for

- More efficient filtering
- More efficient sorting
- Index-only queries (covering

index)

Covered Queries

- When the query criteria and the projection of a query includes only the indexed fields, MongoDB will return results directly from the index without scanning any documents or bringing documents into memory.
- These covered queries can be very efficient.

Types of Indexes

→ Default _id Index

MongoDB creates a unique index on the _id field during the creation of a collection. The _id index prevents clients from inserting two documents with the same value for the
_id field.

Create an Index

To create an index, use
`db.collection.createIndex()`

- The db.collection.createIndex() method only creates an index if an index of the specification does not already exist.
- MongoDB indexes use a B-tree data structure.
- MongoDB provides several different index types to support specific types of data and queries.

Single Field

In addition to the MongoDB-defined _id index, MongoDB supports the creation of user-defined ascending/descending indexes on a single field of a document.

The following example creates an ascending index on the field orderDate.

db.collection.createIndex({ orderDate: 1 })

Compound Index

- MongoDB also supports user-defined indexes on multiple fields, i.e. compound indexes.
- The order of fields listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score: -1 }, the index sorts first by userid and then, within each userid value, sorts by score.

The following example creates a compound index on the orderDate field (in ascending order) and the zip code field (in descending order.)

```
db.collection.createIndex( { orderDate: 1, zipcode: -1 } )
```

Multikey Index

- MongoDB uses multikey indexes to index the content stored in arrays.
- If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array.
- These multikey indexes allow queries to select documents that contain arrays by matching on elements or elements of the arrays.
- MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

Remove Indexes

You can use the following methods to remove indexes:

db.collection.dropIndex() method

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

The above operation removes an ascending index on the item field in the items collection.

To remove all indexes barring the _id index from a collection, use the operation above.

Db.collection.drop indexes()

Modify Indexes

To modify an index, first, drop the index and then recreate it.

Drop Index: Execute the query given below to return a document showing the operation status.

```
db.orders.dropIndex({ "cust_id" : 1, "ord_date" :-1, "items" : 1 })
```

Recreate the Index: Execute the query given below to return a document showing the status of the results.

```
db.orders.createIndex({ "cust_id" : 1, "ord_date" : -1, "items" : -1 })
```

Rebuild Indexes

- In addition to modifying indexes, you can also rebuild them.
- To rebuild all indexes of a collection, use the db.collection.reIndex() method.
- This will drop all indexes including _id and rebuild all indexes in a single operation.

Capped Collections:

- Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order.
- Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

Procedures

→ Create a Capped Collection

- You must create capped collections explicitly using the db.createCollection() method, which is a helper in the mongo shell for the create command.
- When creating a capped collection you must specify the maximum size of the collection in bytes, which MongoDB will pre-allocate for the collection.
- The size of the capped collection includes a small amount of space for internal overhead.

→ Query a Capped Collection

- If you perform a find() on a capped collection with no ordering specified, MongoDB guarantees that the ordering of results is the same as the insertion order.
- To retrieve documents in reverse insertion order, issue find() along with the sort() method with the $natural parameter set to -1, as shown in the following example:

```
db.cappedCollection.find().sort( { $natural: -1 } )
```

→ Check if a Collection is Capped

Use the isCapped() method to determine if a collection is capped, as follows:

```
db.collection.isCapped()
```

→ Convert a Collection to Capped

You can convert a non-capped collection to a capped collection with the convertToCapped command:

db.runCommand({"convertToCapped": "mycoll", size: 100000});

The size parameter specifies the size of the capped collection in bytes.

Spark:

Installing spark

1. Choose a Spark release: 3.1.2 (Jun 01 2021)3.0.3 (Jun 23 2021)
2. Choose a package type: Pre-built for Apache Hadoop 3.2 and later Pre-built for Apache Hadoop 2.7 Pre-built with user-provided Apache Hadoop Source Code
3. Download Spark: spark-3.1.2-bin-hadoop3.2.tgz
4. Verify this release using the 3.1.2 signatures, checksums and project release KEYS.

Note that, Spark 2.x is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12. Spark 3.0+ is pre-built with Scala 2.12.

Spark Applications:

1. Processing Streaming Data

- The most wonderful aspect of Apache Spark is its ability to process streaming data. Every second, an unprecedented amount of data is generated globally.

- This pushes companies and businesses to process data in large bulks and analyze it in real-time. The Spark Streaming feature can efficiently handle this function.

- By unifying disparate data processing capabilities, Spark Streaming allows developers to use a single framework to accommodate all their processing requirements.

Some of the best features of Spark Streaming are:

- Streaming ETL – Spark's Streaming ETL continually cleans and aggregates the data before pushing it into data repositories, unlike the complicated process of conventional ETL (extract, transform, load) tools used for batch processing in data warehouse environments – they first read the data, then convert it to a database compatible format, and finally, write it to the target database.
- Data enrichment – This feature helps to enrich the quality of data by combining it with static data, thus, promoting real-time data analysis. Online marketers use data enrichment capabilities to combine historical customer data with live customer behaviour data for delivering personalized and targeted ads to

customers in real-time.

- Trigger event detection – The trigger event detection feature allows you to promptly detect and respond to unusual behaviours or "trigger events" that could compromise the system or create a serious problem within it.
  While financial institutions leverage this capability to detect fraudulent transactions, healthcare providers use it to identify potentially dangerous health changes in the vital signs of a patient and automatically send alerts to the caregivers so that they can take the appropriate actions.
- Complex session analysis – Spark Streaming allows you to group live sessions and events (for example, user activity after logging into a website/application) together and also analyze them. Moreover, this information can be used to update ML models continually. Netflix uses this feature to obtain real-time customer behaviour insights on the platform and to create more targeted show recommendations for the users.

2. Machine Learning

- Spark has commendable Machine Learning abilities. It is equipped with an integrated framework for performing advanced analytics that allows you to run repeated queries on datasets. This, in essence, is the processing of Machine learning algorithms.
- Machine Learning Library (MLlib) is one of Spark's most potent ML components. This library can perform clustering, classification, dimensionality reduction, and much more.
- With MLlib, Spark can be used for many Big Data functions such as sentiment analysis, predictive intelligence, customer segmentation, and recommendation engines, among other things.
- Another mention-worthy application of Spark is network security. By leveraging the diverse components of the Spark stack, security providers/companies can inspect data packets in real-time inspections for detecting any traces of malicious activity. Spark Streaming enables them to check any known threats before passing the packets on to the repository.
- When the packets arrive in the repository, they are further analyzed by other Spark components (for instance, MLlib). In this way, Spark helps security providers to identify and detect threats as they emerge, thereby enabling them to solidify client security.

3. Fog Computing

- Fog computing is a computing architecture in which a series of nodes receives data from IoT devices in real time. These nodes perform real-time processing of the data that they receive, with millisecond response time.
- Fog Computing decentralizes data processing and storage.
- However, certain complexities accompany Fog Computing – it requires low latency, massively parallel processing of ML, and incredibly complex graph analytics algorithms.

Spark jobs, stages and tasks

- **Job** - So, what Spark does is that as soon as action operations like collect(), count(), etc., is triggered, the driver program, which is responsible for launching the spark application as well as considered the entry point of any spark application, converts this spark application into a single job
- **Stage-** Whenever there is a shuffling of data over the network, Spark divides the job into multiple stages.

  Therefore, a stage is created when the shuffling of data takes place.

  These stages can be either processed parallelly or sequentially depending upon the dependencies of these stages between each other.

  If there are two stages, Stage 0 and Stage 1, and if they are not sequentially dependent, they will be executed parallelly.

- **Task** - it is a type of stage in the spark that produces data for shuffle operation.

  The output of this stage acts as an input for the other following stages.

  In the above code, Stage 0 will act as the ShuffleMapStage since it produces data for shuffle operation, which acts as an input for Stage 1.

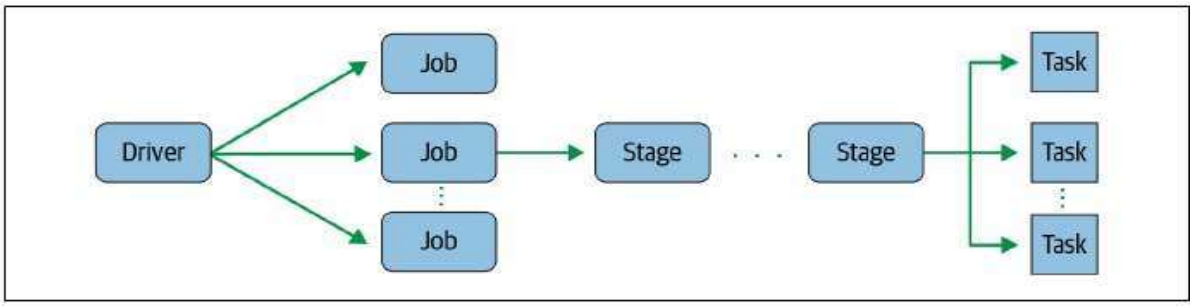


Fig: Concept of jobs, stage and tasks in SPARK

Resilient Distributed Databases

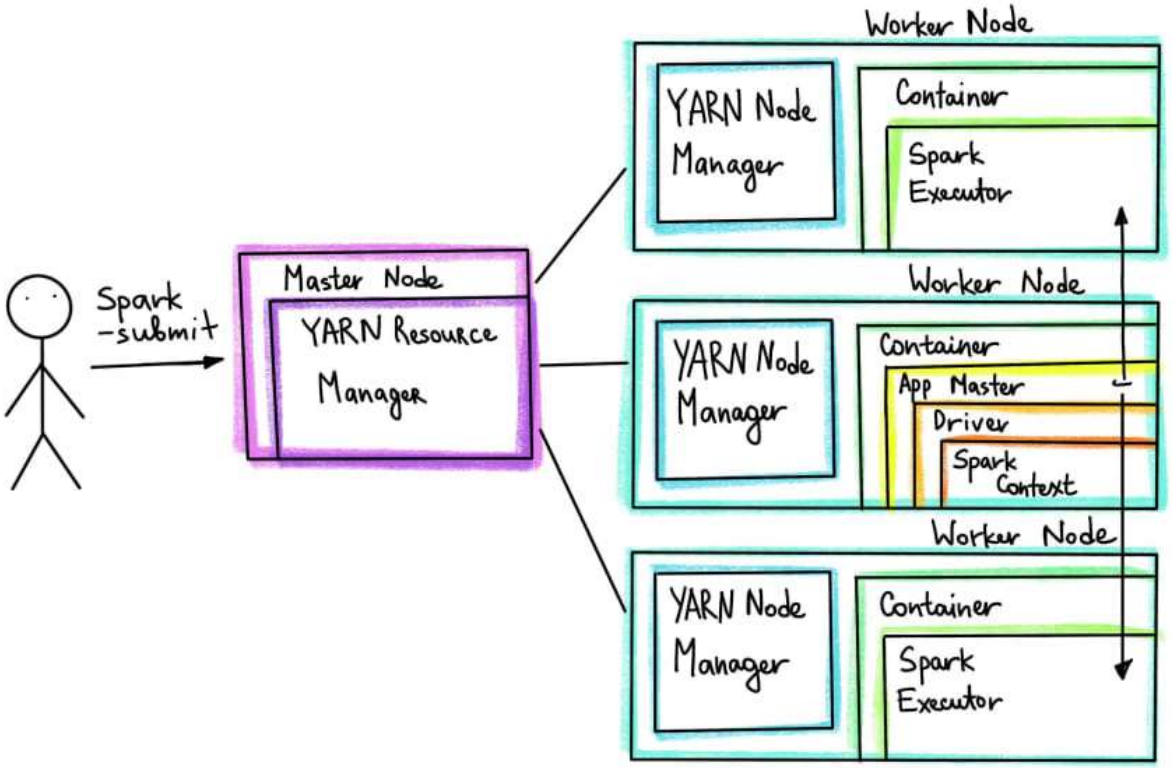
- Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects.
- Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.
- RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
- Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- There are two ways to create RDDs − parallelizing an existing collection in your driver program or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.
- Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations.

Anatomy of a Spark job run

- Spark application contains several components, all of which exists whether you are running Spark on a single machine or across a cluster of hundreds or thousands of nodes.
- The components of the spark application are Driver, the Master, the Cluster Manager and the Executors.
- All of the spark components including the driver, master, executor processes run

in java virtual machines(JVMs).

- A JVM is a cross-platform runtime engine that executes the instructions compiled into java bytecode. Scala, which spark is written in, compiles into bytecode and runs on JVMs.



Spark on YARN:

- Apache Spark is an in-memory distributed data processing engine and YARN is a cluster management technology.
- When running Spark on YARN, each Spark executor runs as a YARN container. Where MapReduce schedules a container and fires up a JVM for each task, Spark hosts multiple tasks within the same container. This approach enables several orders of magnitude faster task startup time.
- Spark supports two modes for running on YARN, “yarn-cluster” mode and “yarn- client” mode. Broadly, yarn-cluster mode makes sense for production jobs, while yarn-client mode makes sense for interactive and debugging uses where you want to see your application’s output immediately.

- In YARN, each application instance has an Application Master process, which is the first container started for that application.
- The application is responsible for requesting resources from the ResourceManager, and, when allocated them, telling NodeManagers to start containers on its behalf.
- In yarn-cluster mode, the driver runs in the Application Master. This means that the same process is responsible for both driving the application and requesting resources from YARN, and this process runs inside a YARN container.

SCALA:

Introduction

- Scala is a general-purpose, high-level, multi-paradigm programming language.
- It is a pure object-oriented programming language which also provides support to the functional programming approach.
- Scala programs can convert to bytecodes and can run on the JVM (Java Virtual Machine).
- Scala stands for Scalable language.

Classes and Objects

A class is a blueprint for objects. Once you define a class, you can create objects from the class blueprint with the keyword new. Through the object, you can use all functionalities of the defined class.

Class

In Scala, a class declaration contains the class keyword, followed by an identifier(name) of the class.

In general, class declarations can include these components, in order:

- **Keyword class:** A class keyword is used to declare the type class.
- **Class name:** The name should begin with a initial letter (capitalized by convention).

- **Superclass(if any):**The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Traits(if any):** A comma-separated list of traits implemented by the class, if any, preceded by the keyword extends. A class can implement more than one trait.
- **Body:** The class body is surrounded by { } (curly braces).

Syntax:

```
class Class_name{
// methods and fields
}
```

Objects

It is a basic unit of Object Oriented Programming and represents the real-life entities.

Syntax

Var obj = new objectname();

Example

var obj = new Dog("tuffy", "papillon", 5, "white");

How to print in SCALA?

Type the following text to the right of the Scala prompt and press the Enter key −

scala> println("Hello, Scala!");

Basic Types and Operators

Basic Data Types:

Value type	Description and range
Byte	8-bit signed 2's complement integer. It has minimum value of –128 and a maximum value of 127 (inclusive).
Short	16-bit signed 2's complement integer. It has a minimum value of –32,768 and maximum of 32,767 (inclusive).
Int	32-bit signed 2's complement integer. It has a minimum value of –2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).
Long	64-bit signed 2's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive).
Float	A single-precision 32-bit IEEE 754 floating point.
Double	A double-precision 64-bit IEEE 754 floating point.
Boolean	Two possible values: true and false.
Char	A single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

Operators:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Scala is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators

The following arithmetic operators are supported by the Scala language.

Operator	Description
+	Adds two operands
-	Subtracts second operand from the first
*	Multiplies both operands
/	Divides numerator by de-numerator

%	Modulus operator finds the remainder after division of one number by another

Relational Operators

The following relational operators are supported by the Scala language

Operator	Description
==	Checks if the values of two operands are equal or not, if yes then the condition becomes true.
!=	Checks if the values of two operands are equal or not, if values are not equal then the condition becomes true.
>	Checks if the value of the left operand is greater than the value of the right operand, if yes then the condition becomes true.
<	Checks if the value of the left operand is less than the value of the right operand, if yes then the condition becomes true.
>=	Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes then the condition becomes true.
<=	Checks if the value of the left operand is less than or equal to the value of the right operand, if yes then the condition becomes true.

Logical Operators

The following logical operators are supported by the Scala language.

Operator	Description
&&	It is called Logical AND operator. If both the operands are non zero then the condition becomes true.
||	It is called Logical OR Operator. If any of the two operands is non zero then the condition becomes true.
!	It is called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then the Logical NOT operator will make it false.

Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The bit positions of the value of the left operand are moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The bit positions of the left operand value are moved right by the number of bits specified by the right operand.
>>>	Shift right zero-fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.

Assignment Operators

There are the following assignment operators supported by Scala language −

Operator	Description
=	Simple assignment operator, Assigns values from right side operands to left side operand
+=	Add AND assignment operator, It adds the right operand to the left operand and assigns the result to the left operand
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assigns the result to left operand
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assigns the result to the left operand
/=	Divide AND assignment operator, It divides left operand with the right operand and assigns the result to left operand
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to the left operand

Functions and Closures

The method in scala is a part of a class that has a name, a signature, optionally some annotations, and some byte code whereas Scala's function is a complete object which can be assigned to a variable.

FunctionDeclarations:

It has the following syntax:

```
def function_name ([parameters list]) : [return type]
```

Function Definitions:

It has the following syntax:

```
def function_name ([parameters list]) : [return type] = {
//body of function
return [expression]
```

CallingFunctions:

Following syntax is used to call a function:

```
function_name( parameter list)
```

If the function is called using an instance of the object then use dot notation as follows:

```
[instance].function_name(parameters list)
```

ScalaClosures

It is a function in which the return value depends on the one or more variables values that are declared outside this function. **e.g.**

```
var value = 20
val sum = (i:Int) => i + value
```

Inheritance.

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in Scala by which one class is allowed to inherit the features(fields and methods) of another class.

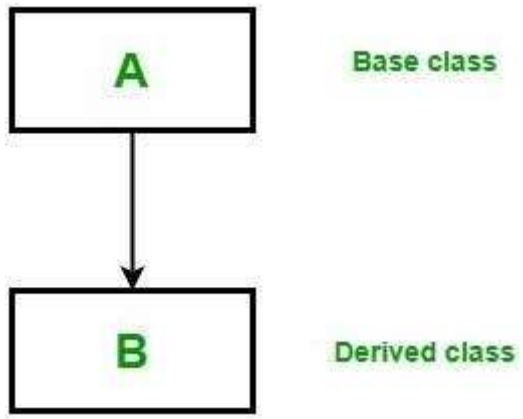
Important terminology:

- Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).
- Sub Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

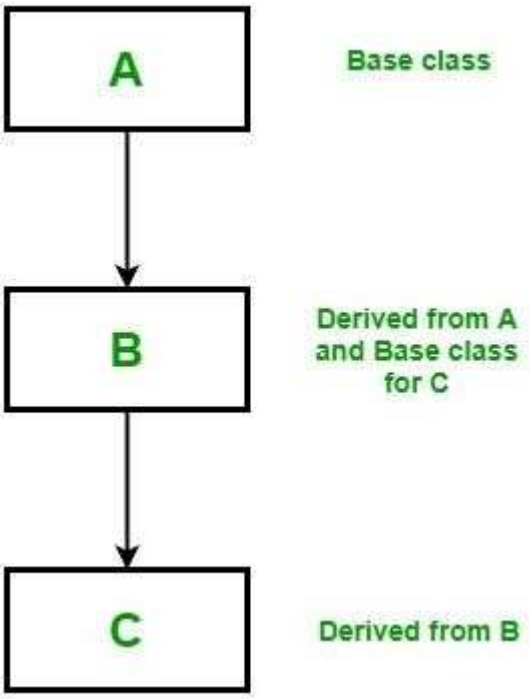
Type of inheritance

Below are the different types of inheritance which are supported by Scala.

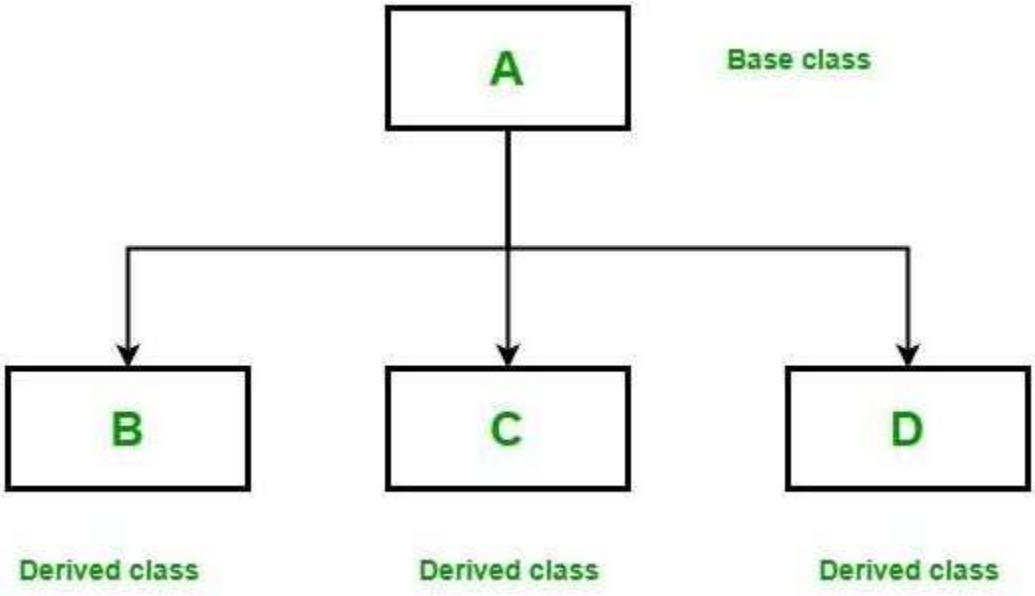
Single Inheritance: In single inheritance, derived class inherits the features of one base class. In the image below, class A serves as a base class for the derived class B.



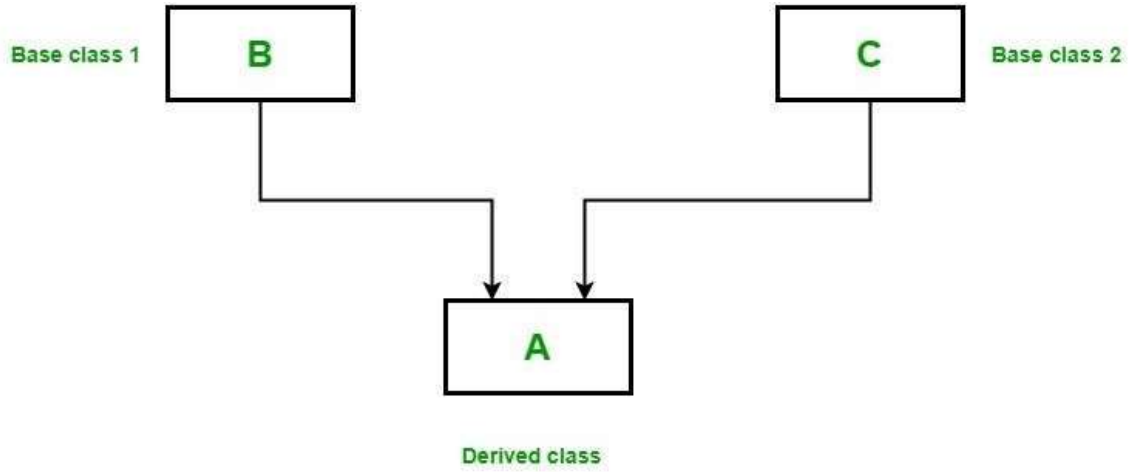
Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to another class. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



Hierarchical Inheritance: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.



Multiple Inheritance: In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. Scala does not support multiple inheritance with classes, but it can be achieved by traits.



Hybrid Inheritance: It is a mix of two or more of the above types of inheritance. Since Scala doesn't support multiple inheritance with classes, hybrid inheritance is also not possible with classes. In Scala, we can achieve hybrid inheritance only through traits.