Big Data Technologies

Agenda

- Apache Hadoop
  - HDFS
  - MapReduce

Apache Hadoop

Hadoop 1.x vs Hadoop 2.x

- Hadoop daemons are background processes implemented in Java.
- Hadoop 1.x Daemons
  - NameNode
  - DataNode
  - SeconadaryNameNode
  - JobTracker
  - TaskTracker
- Hadoop 2.x Daemons
  - NameNode
  - DataNode
  - SeconadaryNameNode
  - ResourceManager
  - NodeManager
- Hadoop 1.x - Cluster size
  - Max 4000 node cluster
- Hadoop 2.x - Cluster size
  - Max 10000 node cluster
- Hadoop 1.x - HDFS - Data blocks
  - Default block size 64 MB.

- Hadoop 2.x - HDFS - Data blocks
  - Default block size 128 MB.
- Hadoop 1.x - HDFS - Data Node failure
  - Data node failure handled by replication (default factor = 3).
  - Replication overheads = 200%.
- Hadoop 2.x - HDFS - Data Node failure
  - Data node failure handled by replication (or erasure encoding in Hadoop 3.x).
  - Erasure encoding overheads = 50%
- Hadoop 1.x - NameNode failure
  - NameNode is single point of failure.
  - Needs manual recovery of metadata from SeconadaryNameNode. The process is time-taking.
- Hadoop 2.x - NameNode failure
  - NameNode is not single point of failure.
  - In HA configuration, Standby NameNode actively takes backup and can become active namenode immediately in case of failure of NameNode.
- Hadoop 1.x - Distributed computing
  - Done by MapReduce components i.e. JobTracker and TaskTracker.
- Hadoop 2.x - Distributed computing
  - Uniform resource sharing done by YARN.
  - Job execution/tracking is done by MRAppMaster.
- Hadoop 1.x - Cluster management
  - Done by MapReduce components.
- Hadoop 2.x - Cluster management
  - Done by YARN. Resources (CPU and Memory) is shared fairly.
  - Hadoop 3.x YARN supports adding user defined resources like GPU.

Hadoop 2.x/3.x Installation

- Hadoop daemons are configurable via XML configuration files.
  - core-site.xml, hdfs-site.xml, mapred-site.xml, yarn-site.xml.
- Hadoop can be installed in 3 possible ways.
  - Local mode
  - Single node cluster

    - Multi node cluster
- It mainly differs in its applications and execution of Hadoop daemons.

Local mode

- All daemons runs in single Java process.
- Can access only LocalFileSystem (file:///).
- Used for unit testing of MR jobs & prototyping.

Pseudo distribution mode (Single node cluster)

- All daemons runs as independent Java processes on the single machine.
- Used as developer machine setup.

Full distribution mode (Multi node cluster)

- All daemons runs as independent Java processes on the multiple machines in the network.
- Typically NodeManager and DataNode runs on the same machine.
- This is production cluster setup to run jobs.

HDFS

- HDFS is "Write Once Read Multiple Times" type of File System.
- Files stored in HDFS are not editable.
- Hadoop 2.x enables appending contents to the file.

Using HDFS

- Before using HDFS need to be formatted. It create first (empty) file system image on NameNode.

```
hdfs namenode -format
```

- Start all HDFS daemons & verify them

```
start-dfs.sh
jps
```

- In Browser:
  - http://namenode:9870
- While metadata is loaded into NameNode memory, HDFS is not ready for use. This state is safe mode.

HDFS user commands

- https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html
- https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FileSystemShell.html
- syntax: hadoop fs genericoptions command

```
hadoop fs -help
```

- Generic options
  - -conf, -fs, ...
- HDFS user commands categories
  - ingestion/retrieval: put, get, getmerge
  - directory handling: ls, mkdir, rmdir
  - file data handling: cat, tail, rm, truncate, touchz, stat
  - metadata handling: chmod, chown, setrep
- HDFS admin commands

```
hdfs -help
hdfs dfsadmin -help
```

HDFS Replication

- Default replication factor for HDFS is 3.
    - hdfs-site.xml -- dfs.replication
    - Each data block is copied on 3 different data nodes.
- Data nodes are stored across the racks for more reliability. Data nodes are chosen by name node to balance the load.
- NameNode ensure availability of datanodes by the periodic heartbeat signal.
- If number of replicas are less than replication factor, it is under-replica.
- If number of replicas are more than replication factor, it is over-replica.
- Hadoop auto adjust replicas to the replication factor over the time by creating more replicas or deleting them depending on scenario.
- Replication is done while write operation.
- If no replica is available while read operation, it fails.

HDFS Java API

- HDFS can be accessed or manipulated using Java API.
- DistributedFileSystem class represent HDFS, while LocalFileSystem class represent local file system.
- Mainly two types of APIs
    - FileSystem API
    - File-IO API

FileSystem API

- Deals with metadata & directories.
- FileStatus object contains metadata of file or directory.
- Most of FileSystem APIs don't need access to DataNode (as metadata is maintained on NameNode itself).

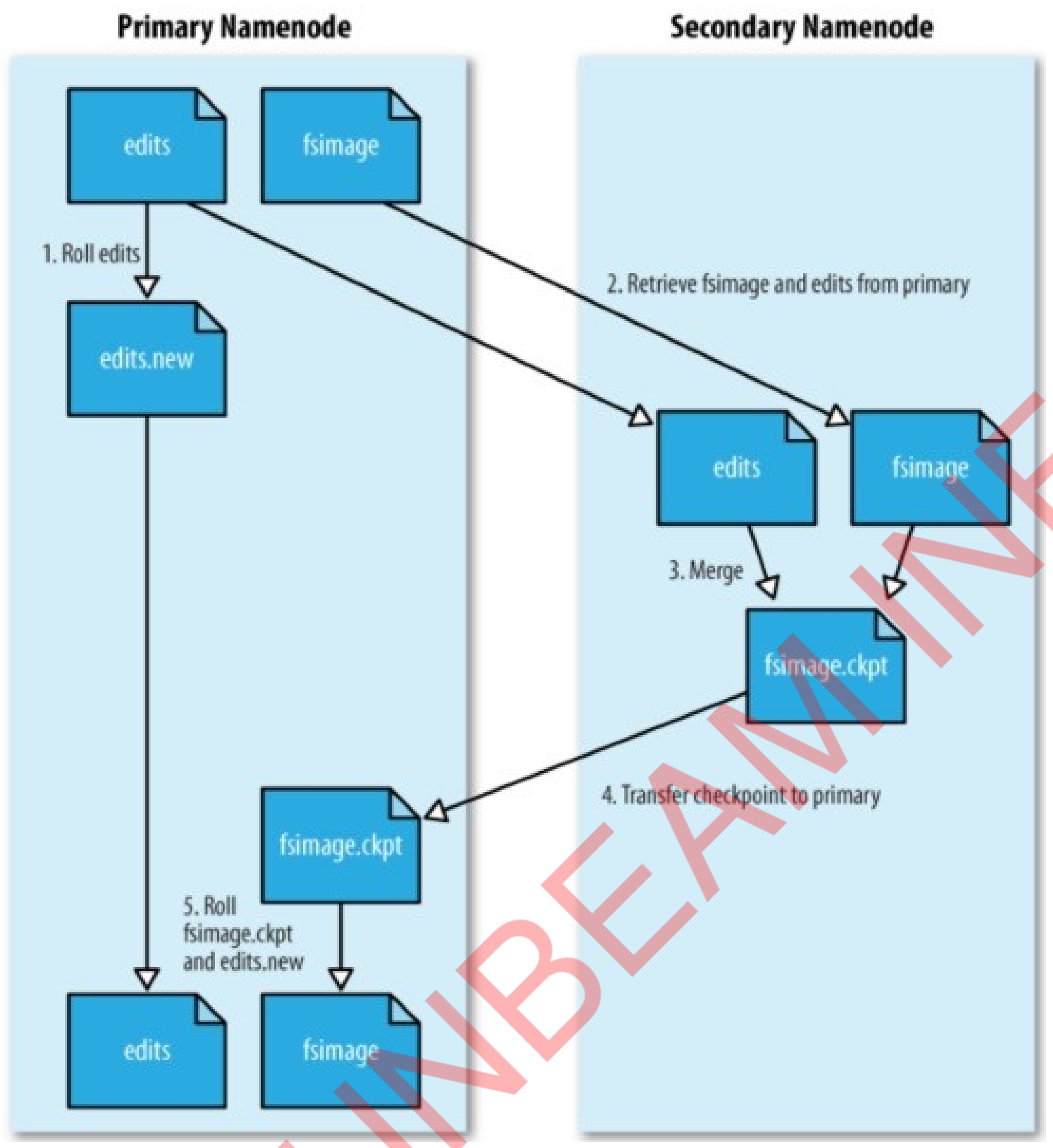
File IO API

- Deals with data of the files.
- FSDataInputStream class for reading the file, while FSDataOututStream class for writing the files.
- They provide abstraction like replication process, network access, etc.

- Write/Read text files
  - HDFS Java API: write operation
    - HDFS Write
  - HDFS Java API: read operation
    - HDFS Read

HDFS Architecture

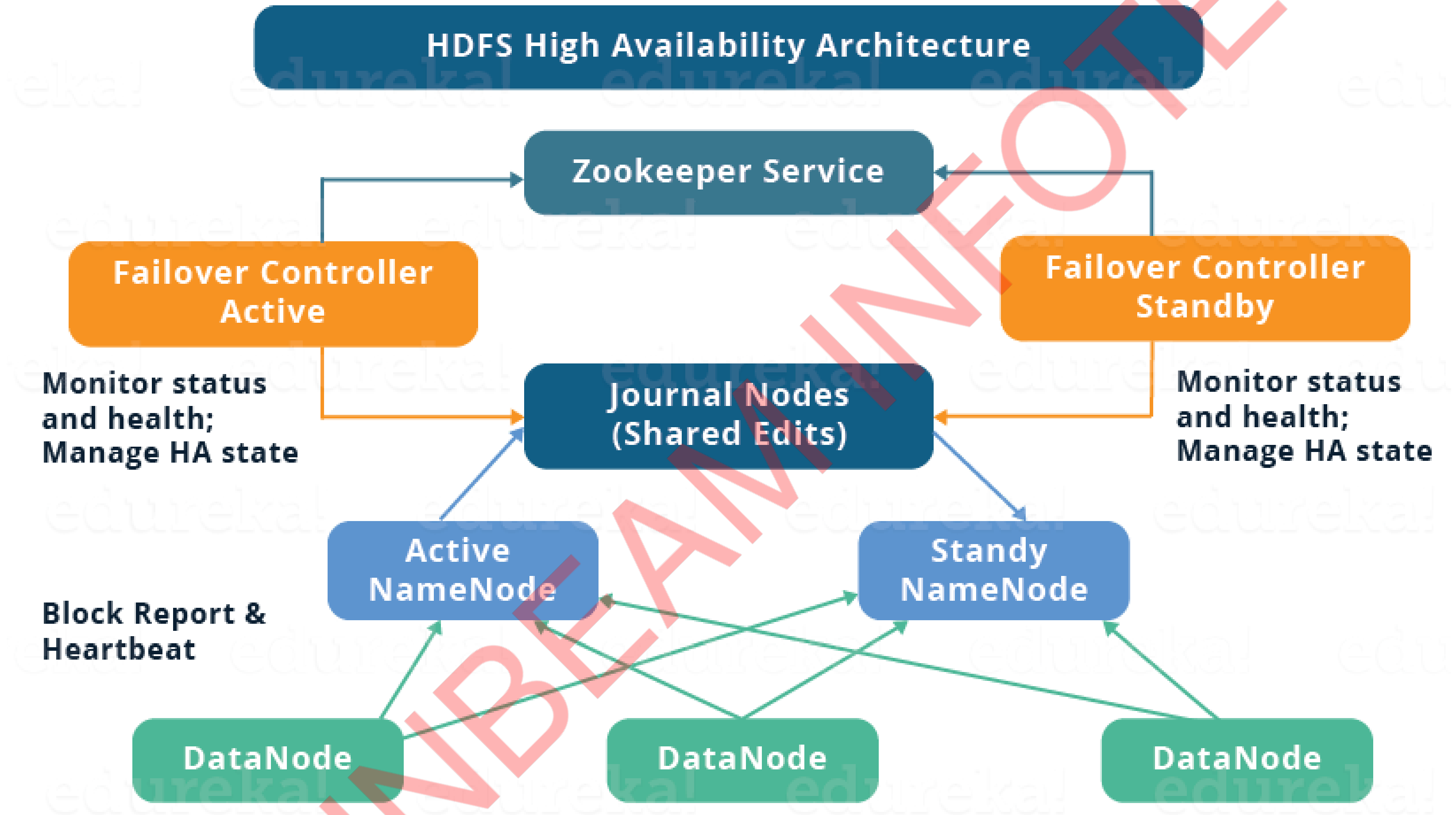
HDFS Architecture/Internals



-
- NameNode loads all metadata into RAM from HDFS fsimage.
- Each change in metadata is fetched by SNN with last fsimage.
- Applying those changes in fsimage, SNN creates next fsimage checkpoint (.ckpt)
- This checkpoint is transferred to NameNode.

- NameNode rename it to consider as new fsimage and deletes old fsimage.
- This ensures that NN & SNN maintains same metadata.

HDFS 2.x HA Configuration

- 
HDFS High Availability Architecture
Zookeeper Service
Failover Controller Active
Failover Controller Standby
Monitor status and health; Manage HA state
Journal Nodes (Shared Edits)
Monitor status and health; Manage HA state
Active NameNode
Standy NameNode
Block Report & Heartbeat
DataNode
DataNode
DataNode
- https://hadoop.apache.org/docs/r2.8.2/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html

Nodes configuration

- Hadoop is developed for commodity hardware (regular machine).
- DataNode/Worker machines needs regular config (e.g. 8 GB+ RAM, 4 CPUs+, 1 TB+ storage)
- NameNode serves all metadata from RAM (for high speed). So namenode machine must have higher RAM and also stable network.
- SeconadayNameNode/StandbyNameNode can be replaced as NameNode in case of NameNode failure. These nodes must have similar configuration as of NameNode.
- Journaling Nodes doesn't need very high config like NameNode.

Safe mode

- When Hadoop cluster is started, all metadata is loaded into NameNode RAM (from its disk fsimage) and the metadata is verified.
- This process may take several minutes (depending on amount of data). During this time, HDFS is said to be in safe mode. Any write operations are not allowed.
- If metadata information is not verified (due to failure of some components/file corruption), safe mode may take longer. Hadoop admin should manually check the system and repair it.

Commissioning and Decommissioning

- Adding new nodes in Hadoop cluster (on-the-fly) is called as Commissioning nodes.
- Removing few nodes from Hadoop cluster (on-the-fly) is called as Decommissioning nodes.

Map-Reduce

Map-Reduce thought process

- Design pattern for distributed computing.

- Word count program

  ```
  red green blue
  green blue black
  red red green
  green red green
  ```

- Read text line by line (1 line = 1 record)
- Mapper -- Process each record individually -- Split line into words and each word occurred once.
    - (Offset,Line)--> Mapper --> (Word,Count)

    ```
    red     1
    green   1
    blue    1
    green   1
    blue    1
    black   1
    red     1
    red     1
    green   1
    green   1
    red     1
    green   1
    ```

    - Logic/Pseudo code

    ```
    String[] words = line.split(" ");
    for(String word: words) {
        ctx.write(word, 1);
    }
    ```

- Sort and Group words

```
black   1
blue    1
blue    1
```

```
green   1
green   1
green   1
green   1
green   1
red     1
red     1
red     1
red     1
```

```
black   1
blue    1,1
green   1,1,1,1,1
red     1,1,1,1
```

- Reducer -- Count/Sum all 1s per group
  - (Word,Counts) --> Reducer --> (Word,Sum)

```
sum = 0;
for(int cnt:counts)
    sum = sum + cnt;
ctx.write(word, sum);
```

- Write the output record by record.
- Map Reduce Execution Overview
  - Step 1*: Read data record by record
  - Step 2: Mapper -- Process each record and produce key-value output
  - Step 3*: Sort by key and Group by key

- Step 4: Reducer -- Perform aggregate operation on each group and produce key-value output
- Step 5*: Write result record by record

Implementing MR job

- Implement Mapper class - To handle individual record
- Input from InputFormat record by record and key-value pair output to merge stage

```
public class MyMapper extends Mapper<KeyIn, ValueIn, KeyOut, ValueOut> {
    @Override
    public void map(KeyIn key, ValueIn value, Mapper.Context<> ctx) {
        // process each input key and value
        // produce output key and value
        // write it into hadoop context
    }
}
```

```
// Equivalent Java code
// Input    --> Long Offset, String Line
// Output   --> String Word, Integer Count(1)
public class WordCountMapper extends Mapper<Long,String,String,Integer> {
    @Override
    public void map(Long offset, String line, Mapper.Context<> ctx) {
        String[] words = line.split(" ");
        for(String word:words)
            ctx.write(word, 1);
    }
}
```

```
// Hadoop code
// Input    --> LongWritable Offset, Text Line
// Output   --> Text Word, IntWritable Count(1)
public class WordCountMapper extends Mapper<LongWritable,Text,Text,IntWritable> {
    @Override
    public void map(LongWritable offsetWr, Text lineWr, Mapper.Context<> ctx) {
        String line = lineWr.toString();
        String[] words = line.split(" ");
        for(String word:words) {
            Text wordWr = new Text(word);
            IntWritable oneWr = new IntWritable(1);
            ctx.write(wordWr, oneWr);
        }
    }
}
```

- Implement Reducer class -- To Perform aggregation on set of values (corresponding to each key)
- Input from merge stage in key-values pair and key-value pair output to OutputFormat

```
public class MyReducer extends Reducer<KeyIn,ValueIn,KeyOut,ValueOut> {
    @Override
    public void reduce(KeyIn key, Iterable<ValueIn> values, Reducer.Context<> ctx) {
        // perform aggregate operation on "values"
        // produce key-value result
        // write it into hadoop context
    }
}
```

```
// Equivalent Java code
// Input: String Word, Integer One(1)
// Output: String Word, Integer Sum
public class WordCountReducer extends Reducer<String,Integer,String,Integer> {
    @Override
    public void reduce(String word, Iterable<Integer> counts, Reducer.Context<> ctx) {
        int sum = 0;
        for(Integer cnt:counts)
            sum = sum + cnt;
        ctx.write(word, sum);
    }
}
```

```
// Hadoop code
// Input: Text Word, IntWritable One(1)
// Output: Text Word, IntWritable Sum
public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    @Override
    public void reduce(Text wordWr, Iterable<IntWritable> countWrs, Reducer.Context<> ctx) {
        int sum = 0;
        for(IntWritable cntWr:countWrs) {
            int cnt = cntWr.get();
            sum = sum + cnt;
        }
        IntWritable sumWr = new IntWritable(sum);
        ctx.write(wordWr, sumWr);
    }
}
```

- Implement Driver class -- to Submit MR job
  - Create Job and set Jar
  - Mapper class & its output
  - Reducer class & its output
  - Input & Output format

- Combiner, Partitioner
- Submit job

Map-Reduce Example: Jobwise Total Salary

- Mapper Implementation
  - (LongWritable,Text) --> Mapper --> (Text,DoubleWritable)
    - Input: LongWritable offset, Text line
    - Output: Text job, DoubleWritable sal

```
// (Pseudo code)
class EmpMapper extends Mapper<LongWritable, Text, Text, DoubleWritable> {
    // key = offset (byte number of file at which record starts)
    // value = one record = one line
    @Override
    public void map(LongWritable key, Text value, Context ctx) {
        String line = value.toString();
        String[] parts = line.split(",");
        String job = parts[2];
        double sal = Double.parseDouble(parts[5]);
        ctx.write(new Text(job), new DoubleWritable(sal));
    }
}
```

- Reducer Implementation
  - (Text,DoubleWriables) --> Reducer --> (Text,DoubleWritable)
    - Input: Text job, `Iterable<DoubleWritable>` sals
    - Output: Text job, DoubleWritable totalSal

```
// (Pseudo code)
class EmpReducer extends Reducer<Text,DoubleWritable,Text,DoubleWritable> {
    @Override
    public void reduce(Text key, Iterable<DoubleWritable> values, Context ctx) {
        double total = 0.0;
        for(DoubleWritable sal : values)
            total = total + sal.get();
        ctx.write(key, new DoubleWritable(total));
    }
}
```

- Driver Implementation (main class)
    - Create job and submit for execution

GenericOptionsParser

- Parses command-line arguments
- Extract generic-options from command-line args and create Configuration from it.

Configured class

- Associate configuration object with the driver
- getConf() and setConf()

Tool and ToolRunner

- Tool is standard way of implementing any processing on Hadoop – run() method
- ToolRunner is helper to execute the Tool.

Hadoop Writables

- Like java wrapper classes, but optimized for serialization over the network.
- IntWritable, ByteWritable, ShortWritable, LongWritable, DoubleWritable, BooleanWritable, Text
- ArrayWritable, MapWritable, NullWritable

Primitive	Java Wrapper	Hadoop Writables
byte	Byte	ByteWritable
short	Short	ShortWritable
int	Integer	IntWritable
long	Long	LongWritable
float	Float	FloatWritable
double	Double	DoubleWritable
boolean	Boolean	BooleanWritable
char	Character	-
String *	-	Text
null *	-	NullWritable
Array	-	ArrayWritable
Map	-	MapWritable

Execute MR Job

- Runnable Jar
  - `hadoop jar <jar-path> <generic-options> <cmd-line args to main-class>`
- Non-Runnable Jar
  - `hadoop jar <jar-path> <main-class> <generic-options> <cmd-line args to main-class>`
- Generic options:
  - -conf
  - -D
  - -fs
  - -jt

- -files
- -libjars

Assignments

1. Implement word count program using map reduce.
2. Implement subject-wise books total price (books.csv) using map reduce.
3. Implement job-wise avg sal (emp.csv) using map reduce.