Trying to have your cake and eat it too
Each phase pines for tasks with locality and their numbers on a tether
Alas within a phase, you get one, but not the other
Who gets what?
Stay tuned to find out

Shrideep Pallickara
Computer Science
Colorado State University
Topics covered in this lecture

- API differences
- Combiner Functions
- Hadoop Distributed File System

API DIFFERENCES
The old and new MapReduce APIs

- The new API favors abstract classes over interfaces
  - Makes things easier to evolve
- New API is in `org.apache.hadoop.mapreduce` package
  - Old API can be found in `org.apache.hadoop.mapred`
- New API makes use of context objects
  - Context unifies roles of JobConf, OutputCollector, and Reporter from the old API

- In the new API, job control is done using the Job class rather than using the JobClient

- Output files are named slightly differently
  - Old API: Both map and reduce outputs are named `part-nnnn`
  - New API: Map outputs are named `part-m-nnnn` and reduce outputs are named `part-r-nnnn`
The old and new MapReduce APIs

- The new API's reduce() method passes values as Iterable rather than as Iterator
  - Makes it easier to iterate over values using the for-each loop construct

```
for (VALUEIN value: values) {
    ...
}
```

MAPREDUCE TASKS & SPLIT STRATEGIES
Hadoop divides the input to a MapReduce job into fixed-sized pieces

- These are called **input-splits** or just splits
- Creates **one map task per split**
  - Runs **user-defined map function** for each **record** in the split

Split strategy: Having many splits

- Time taken to process split is small compared to processing the whole input
- Quality of **load balancing** increases as splits become **fine-grained**
  - Faster machines process proportionally more splits than slower machines
  - Even if machines are identical, this feature is desirable
    - Failed tasks get relaunched, and there are other jobs executing concurrently
Split strategy: If the splits are too small

- **Overheads** for managing splits and map task creation dominates total job execution time
- Good split size tends to be an HDFS block
  - This could be changed for a cluster or specified when each file is created

Scheduling map tasks

- Hadoop does its best to run a map task on the *node where input data resides* in HDFS
  - **Data locality**
- What if all three nodes holding the HDFS block replicas are busy?
  - Find free map slot on node in the same rack
  - Only when this is not possible, is an off-rack node utilized
    - Inter-rack network transfer
Why the optimal split size is the same as the block size...

- Largest size of input that can be stored on a single node
- If split size spanned two blocks?
  - Unlikely that any HDFS node has stored both blocks
  - Some of the split *will have to be transferred* across the network to node running the map task
    - Less efficient than operating on local data without the network movement

MANAGING OUTPUTS
Map task outputs

- Stored on the local disk
  - Not HDFS

- Once the job is complete, intermediate map outputs are thrown away
  - Storing in HDFS with replication is an overkill

Reduce tasks do not have the advantage of data locality

- Input to a single reduce task
  - Output from all the mappers
  - Sorted map outputs transferred over the network to node where reduce task is running
    - Merged and then passed to the reduce function

- Output of reduce task stored on HDFS
  - One replica of block is stored on local node, other replicas are stored on off-rack nodes
Number of reduce tasks

- Not governed by the size of the input
- Specified independently

When there are multiple reducers

- Maps **partition** their outputs
  - One partition for each reduce task
  - There can be many keys in each partition
  - Records for a given key are all in the same partition

- Partitioning controlled with a **partitioning function**
  - Default uses a hash function to bucket the key space
MapReduce Dataflow

Input HDFS

split 0 → Map → Sort → Copy → Merge → Reduce → Part 0

split 1 → Map → Merge → Reduce → Part 1

split 2 → Map → Merge → Reduce

Output HDFS

HDFS Replication

COMBINER FUNCTIONS

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT
CS455: Introduction to Distributed Systems
http://www.cs.colostate.edu/~cs455

COLORADO STATE UNIVERSITY
Combiner functions

- Many MapReduce jobs are limited by the available network bandwidth
  - Framework has mechanisms to minimize the data transferred between map and reduce tasks

- A combiner function is run on the map output
  - Combiner output fed to the reduce task

Combiner function

- No guarantees on how many times Hadoop will call this on a map output record
  - The combiner should, however, result in the same output from the reducer

- Contract for the combiner constrains the type of function that can be used
Combiner function: Let’s look at the maximum temperature example

[1/2]

(1950, 0)
(1950, 20)
(1950, 10)
Map 1

(1950, [0, 20, 10, 25, 15])
Reduce
(1950, 25)

(1950, 25)
(1950, 15)
Map 2

Combiner

Combiner

Reduce
(1950, 25)

Combiner function: Let’s look at the maximum temperature example

[2/2]

(1950, 0)
(1950, 20)
(1950, 10)
Map 1

(1950, [20, 25])
Reduce
(1950, 25)

(1950, 25)
(1950, 15)
Map 2

Combiner

Combiner
A closer look at the function calls

- \( \text{max}(0, 20, 10, 25, 15) = \text{max}(20, 25) = 25 \)

- Functions with this property are called **commutative** and **associative**
  - Commutative: Order of operands \((5+2) = (2 + 5)\)
  - Associative: Order of operators \(5 \times (5 \times 3) = (5 \times 5) \times 3\)
    - Division and subtraction are not commutative
    - Vector cross products are not

Not all functions possess the commutative and associative properties

- What if we were computing the mean temperatures?
  - We cannot use mean as our combiner function

\[
\text{mean}(0, 20, 10, 25, 15) = 14
\]

\[
\text{mean}(\text{mean}(0, 20, 10), \text{mean}(25, 15)) = \text{mean}(10, 20) = 15
\]
Combiner: Summary

- The combiner does not replace the reduce function
  - Reduce is still needed to process records from different maps
- But it is useful for cutting down traffic from maps to the reducer

Specifying a combiner function

```java
public class MaxTemperatureWithCombiner {
    public static main(String[] args) throws Exception {
        Job job = Job.getInstance();
        job.setJarByClass(MaxTemperature.class);
        job.setJobName("Max temperature");
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(MaxTemperatureMapper.class);
        job.setCombinerClass(MaxTemperatureReducer.class);
        job.setReducerClass(MaxTemperatureReducer.class);
        job.setOutputKey(Text.class);
        job.setOutputValueClass(IntWritable.class);
        System.exit(job.waitForCompletion(true) ? 0: 1);
    }
}
```
### Rationale

- Datasets often outgrow storage capacity of a single machine
  - Necessary to **partition** data across multiple machines

- File systems managing storage access *across* a network of machines
  - Distributed file systems
HDFS is designed for storing …

- **Very large** files
  - File sizes are in the order of 100s of GB or a few TB

- With **streaming data access** patterns
  - Write-once, read many times pattern
  - Each analysis involves a large portion of the dataset
    - Time to read dataset is more important than latency for the first record

- On **commodity hardware**

---

What is HDFS not suitable for? [1/2]

- **Low-latency** data access

- Lots of **small files**
  - Name nodes holds file system metadata in memory
  - Each file, directory, and block takes about 150 bytes
    - If there were $10^6$ files each of which had 1 block
      - 300 MB of memory
    - Millions of files are feasible but not billions of files
What is HDFS not suitable for? [2/2]

- **Multiple writers**, arbitrary file modifications

- HDFS does not support:
  - Multiple concurrent writers
  - Modifications at arbitrary offsets

---

Block

- Filesystems for a single disk, deal with data in blocks
  - Integral number of the HDD block size

- Block sizes
  - Filesystem blocks are a few KB
  - Disk blocks are normally 512 bytes
HDFS Blocks

- Has a much larger size: **128 MB** [default]
- Files are **broken** into block-sized **chunks**
  - Each chunk is stored as an independent unit
- If the last chunk is less than the HDFS block size?
  - No space is wasted because the blocks are themselves stored as files

Why is the block-size so big?

- **Time to transfer** data from disk can be made significantly larger than the time to seek first block
- If the seek time is 10 ms and transfer rate is 100 MB/sec?
  - To make seek time 1% of the transfer time, block size should be 100 MB
- Must be careful not to overdo block size increase
  - Since tasks operate on blocks, the number of tasks could reduce.
Benefits of the block abstraction in distributed file systems

- File can be **larger than any single disk** in the cluster
- Simplifies the storage subsystem
  - File metadata (including permissions) handled by another subsystem and not stored with the block

Blocks and replication

- Each block is replicated on a small number of **physically separate** machines
- If a block becomes unavailable?
  1. Copy **read from another location** transparently
  2. That block is also **replicated from its alternative locations** to other live machines
     - Bring replication factor back to the desired level
HDFS’ fsck command

- List blocks that make up each file in the filesystem

```
% hadoop fsck / -files -blocks
```
Namenode

- Manages filesystem namespace
- Maintains filesystem tree and metadata
  - For all files and directories in the tree
- Information stored persistently on local disk in two files
  - Namespace image and the edit log

Tracking location of blocks comprising files

- Namenode knows about datanodes on which all blocks of a file are located
- The locations of the blocks are not stored persistently
  - Information reconstructed from datanodes during start up
Interacting with HDFS

- HDFS presents a **POSIX-like** file system interface
- Client code does not need to know about the namenode and datanode to function

Datanodes

- Store and retrieve blocks
  - Initiated by the client or the namenode
- **Periodically reports** back to the namenode with the *list of blocks* that they store
Failure of the namenode

- Decimates the filesystem
- **All files on the filesystem are lost**
  - No way of knowing how to reconstitute the files from the blocks

Guarding against namenode failures

- **Backup** files comprising the persistent state of the filesystem metadata
  - Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems
    - Writes are synchronous and atomic
- Run a **secondary** namenode
  - Does not act as a namenode
  - Periodically merges namespace image with edit log
Secondary namenode

- Runs on a separate physical machine
  - Requires as much memory as the namenode to perform the merge operation
- Keeps a copy of the merged namespace image
  - Can be used if the namenode fails
- However, the secondary namenode lags the primary
  - Data loss is almost certain

HDFS Federation (introduced in 0.23)

- On large clusters with many files, memory is a limiting factor for scaling
- HDFS federation allows scaling with the addition of namenodes
  - Each manages a portion of the filesystem namespace
    - For e.g., one namenode for /user and another for /share
HDFS Federation [1/2]

- Each namenode manages a namespace volume
  - Metadata for the namespace and block pool

- Namespace volumes are **independent** of each other
  - No communications between namenodes
  - Failure of one namenode does not affect availability of another

HDFS Federation [2/2]

- Block pool storage is **not partitioned**

- Datanodes register with each namenode in the cluster
  - Store blocks from multiple blockpools
Recovering from a failed namenode

- Admin starts a new primary namenode
  - With one of the filesystem metadata replicas
  - Configure datanodes and clients to use this namenode

- New namenode unable to serve requests until:
  1. Namespace image is loaded into memory
  2. Replay of edit log is complete
  3. Received enough block reports from datanodes to leave safe mode

Recovering from a failed namenode

- Recovery can be really long
  - On large clusters with many files and blocks this can be about 30 minutes

- This also impacts routine maintenance
HDFS High Availability has features to cope with this

- Pair of namenodes in active standby configuration
- During failure of active namenode, standby takes over the servicing of client requests
  - In 10s of seconds

HDFS High-Availability:
Additional items to get things to work

- Namenodes use a highly-available *shared storage* to store the *edit log*
- Datanodes must send block reports to *both* namenodes
  - Block mappings stored in memory not disk
- Clients must be configured to handle namenode failover
HDFS HA: Dealing with ungraceful failovers

- Slow network or a network partition can trigger failover transition
  - Previously active namenode thinks it is still the active namenode
- The HDFS HA tries to avoid this situation using fencing
  - Previously active namenode should be prevented from causing corruptions

Fencing mechanisms: To shutdown previously active namenode

- Kill the namenode’s process
- Revoking access to the shared storage directory
- Disabling namenode’s network port
  - Using the remote management command
- STONITH
  - Use specialized power distribution unit to forcibly power down the host machine
The contents of this slide set are based on the following references