COMP5349 –Cloud Computing

Week 5: Map Reduce Design Patterns

A/Prof. Uwe Roehm
School of Information Technologies

Review Question 1

Which statement is true about MapReduce?

- **A:** several **map** tasks run in parallel on different nodes.
- **B**: there must be as many **map** tasks as **reduce** tasks.
- **C**: **map** tasks communicate directly with each other.
- **D**: the number of **map** tasks is determined by the cluster size.

http://tinyurl.com/lhmnjre

Review Question 2

How do Map and Reduce tasks exchange data?

- **A:** RPC
  (Remote Procedure Calls)
- **B**: local file system
- **C**: HDFS

http://tinyurl.com/k2a6cuv

Review Question 3

Can there be a map-only Hadoop job?

- **YES**

  or

- **NO**

http://tinyurl.com/m29zxbh

Outline

- **MR Design Patterns**
- Total Order Sorting in Hadoop
- Joining with MR
- More MR Design Patterns
- (YARN Resource Management and Scheduling)

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Input Transformation and Validation

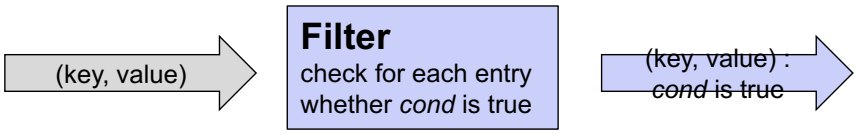
- Goal: Get Input into right format
- Pattern:
  - (a) Use corresponding InputReader
  - (b) Further input validation in Map task

- Example:
  - cf. Homework wk1: some input lines were incomplete

```
map(key, line):
   values = line.split()
   if len(values) > 2:
      movie_id = int(values[1])
      rating = int(values[2])
      if 1 <= rating <= 5: …
```

Filtering

- Goal: Filtering input data for certain conditions
- Pattern: Map task

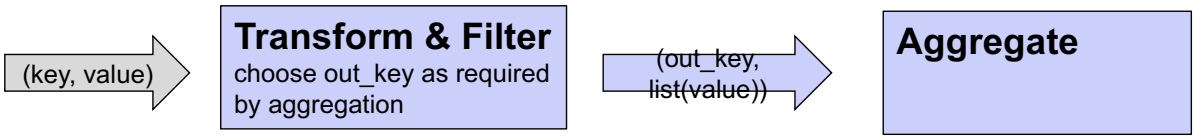


- Example:
  - cf. Homework wk1: Aggregate over those movies with movie id in a given range
  - **`map(key, line):`**

```
values = line.split()
movie_id = int(values[1])
if min_id <= movie_id <= max_id:
     emit(movie_id, int(values[2]))
else
     emit null
```


Aggregation

- Goal: Aggregate input values (for same key) to single result
  - eg. COUNT, SUM, AVERAGE
- Pattern: Reduce task with appropriate key chosen wisely



- Example:
  - cf. Homework wk1: Find average rating per movie
  - **`map(k,line)`** : output movie_id as out_key, rating as value
  - **`reduce(movie_id,ratings[]):`**

```
return (movie_id, sum(ratings)/ratings.count)
```

SQL Equivalence

- If you know SQL, this is basically equivalent to

```
SELECT out_key, AGGREGATE(value')
  FROM InputData(key, value)
 WHERE cond(key, value) = TRUE
 GROUP BY out_key
```

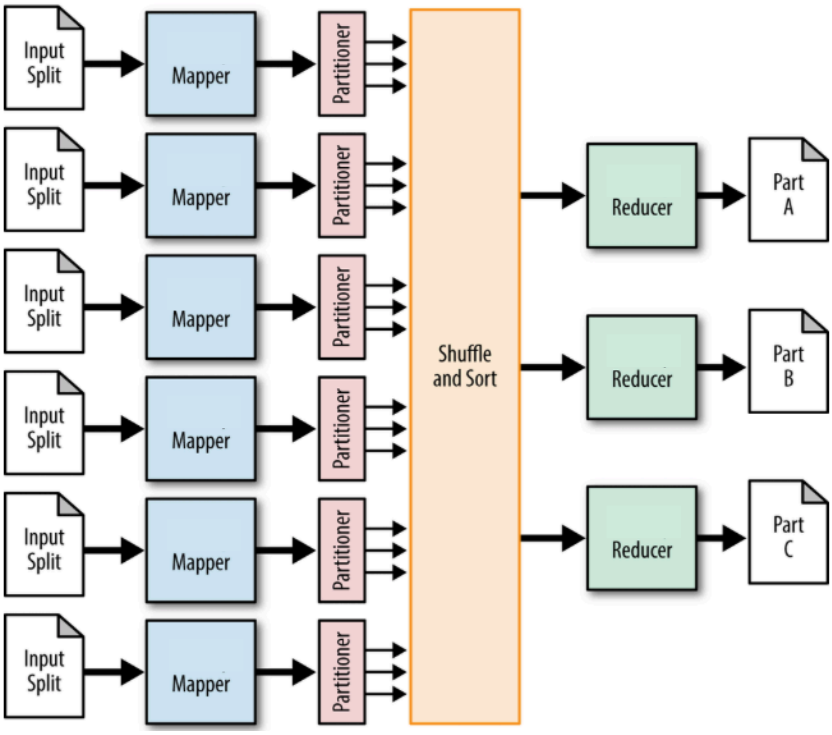

Sorting

- By *out_key*
  - done during shuffle phase
- Total order sorting?
  - equiv. ORDER BY clause of SQL
  - Needs second job

Agenda

- MR Design Patterns
- **Total Order Sorting in Hadoop**
- Joining with MR
- More MR Design Patterns
- (YARN Resource Management and Scheduling)

MapReduce Dataflow



[Figure from Donald Miner and Adam Shook, "Map Reduce Design Patterns", O'Reilly, 2013.]

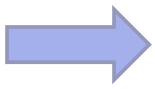
Sorting in Hadoop

- Hadoop has a standard TeraSort example; its source can be found in the Hadoop distribution
- Each reducer sorts its input **<key,v>** pairs based on the keys
  - Primitive key type such as **string**, **int** has its buildin comparator
  - *Custom key type* should provide a comparator
- Reducers are identified by id **[0,R-1]**
  - Reducers' outputs are named as **part-r-00000**, **part-r-00001**,…
  - The final output is a merge of all individual outputs in the order
- Default hash partitioner ensure map output keys are distributed to all reducers evenly
- To implement sort, a customized partitioner is required to make sure all keys in reducer **k-1** is always smaller than keys in reducer **k.**

Example: Sort Words Based on Frequency

- The initial output of *word count* is sorted on out_key (alphabetically-sorted by word)
- Q: How to get the output sorted on word frequency?

automobile	2700
aviation	1957
avion	1518
awesome	626
badajoz	734
bahia	701
barcelona	1305
barros	639
bc	2271
bcn	507
beach	6010

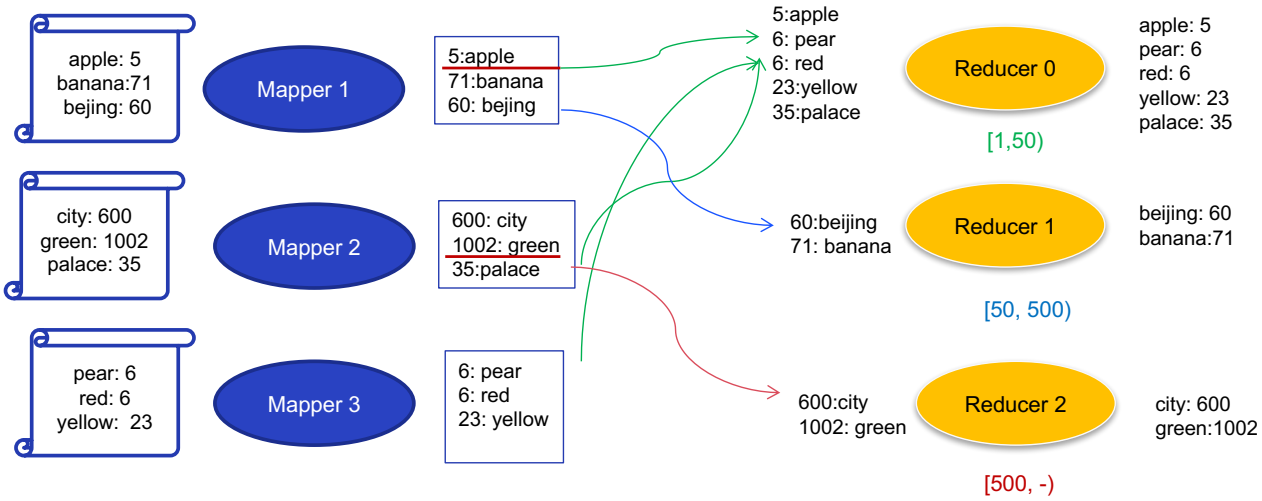


beach	6010
automobile	2700
bc	2271
aviation	1957
avion	1518
barcelona	1305
badajoz	734
bahia	701
barros	639
awesome	626
bcn	507

Sorting: A Second Job is Added

Map/Reduce tasks:

```
Map:
  (tag, freq) -> (k:freq, v:tag)
Reduce:
(k:freq,v:(t1,t2,..)) -> (K:t1, v: freq), (K:t2, v: freq),…
```



The Mapper and the Reducer

```
public class TagFreqMapper extends Mapper<Object, Text, IntWritable, Text> {
    private Text word = new Text();
    private IntWritable freq = new IntWritable();
    //just reverse the key and value
    public void map(Object key, Text value, Context context)
           throws IOException, InterruptedException{
           String[] tagFreqArray = value.toString().split("\t");
           if (tagFreqArray.length == 2){
                  word.set(tagFreqArray[0]);
                  try{
                      freq.set(Integer.parseInt(tagFreqArray[1]));
                      context.write(freq, word);
                  }catch(NumberFormatException e){
                      return; //not doing anything
                  }
           }
    }
}
```

```
public class TagFreqReducer extends Reducer<IntWritable,Text,Text,IntWritable> {
    //just reverse the key and value again!
    public void reduce(IntWritable key, Iterable<Text> values, Context context)
           throws IOException, InterruptedException{
           for (Text val : values) {
                    context.write(val, key);
           }
   }
}
```

A Very Simple Custom Partitioner

```
package comp5349sort;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;
public class SortPartitioner extends Partitioner<IntWritable, Text> {
   public int getPartition(IntWritable key, Text value, int arg2) {
   // a simple and static partitioner to partition the key into 3
   regions, assume we know before hand the number of reducer is 3
   // [1,50) => 0, [50, 500) => 1, [500,-) => 2
         int keyInt = key.get();
         if (keyInt <50)
                  return 0;
         if (keyInt < 500 )
                  return 1;
         else
                  return 2;
   }
}
```

Set the custom partitioner for this job:

```
job.setPartitionerClass(SortPartitioner.class);
```


Problem when using a Simple Partitioner

Contents of directory /user/zhouy/sortout

Goto : /user/zhouy/sortout go

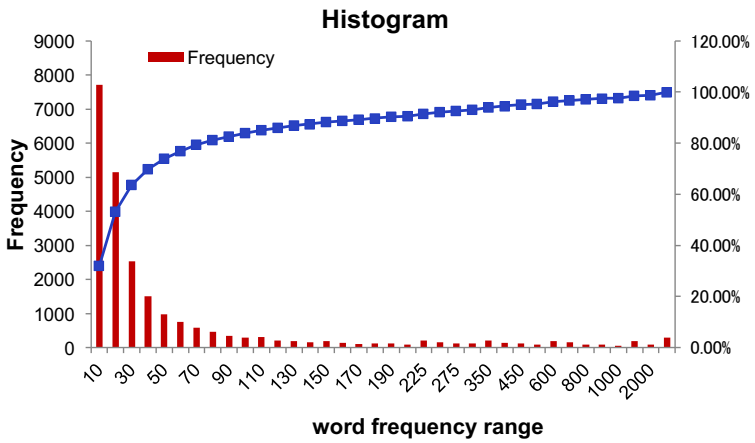
Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
logs	dir				2011-08-26 16:45	rwxr-xr-x	zhouy	hadoop
part-r-00000	file	742.79 KB	2	64 MB	2011-08-26 16:46	rw-r--r--	zhouy	hadoop
part-r-00001	file	221.17 KB	2	64 MB	2011-08-26 16:46	rw-r--r--	zhouy	hadoop
part-r-00002	file	42.75 KB	2	64 MB	2011-08-26 16:46	rw-r--r--	zhouy	hadoop

The keys are not evenly distributed!

- In the TeraSort algorithm, the boundaries of each region is determined dynamically based on data distribution
- The distribution is estimated by taking a small sample of the data
  - E.g. for example, the tag frequency distribution is like a zipf shape

Distribution of Word Frequencies



Bin	*Frequency*	*Cumulative %*
10	7719	31.87%
20	5153	53.15%
30	2524	63.57%
40	1503	69.78%

The bounds 10 and 30 would roughly divide the keys evenly among three reducers

Chaining Multiple Jobs

- If we want to find the word frequencies and sort words based on frequencies, we need to run two jobs
- It is easy to chain jobs together in a form of
  - Map1 -> Reduce 1 -> Map2 -> Reduce 2 …
  - The first job in the chain should write its output to a path which is then used as the input path for the second job

```
Job countJob = new Job(conf, "count Word");
countJob.setNumReduceTasks(3);
…
TextInputFormat.addInputPath(countJob, new Path(otherArgs[0]));
TextOutputFormat.setOutputPath(countJob, new Path("temp"));
countJob.waitForCompletion(true);
Job sortJob = new Job(conf,"sort WordFrequency");
TextInputFormat.addInputPath(sortJob, new Path("temp"));
TextOutputFormat.setOutputPath(sortJob, new Path(args[1]));
System.exit(sortJob.waitForCompletion(true) ? 0 : 1);
```

Outline

- MR Design Patterns
- Total Order Sorting
- **Joining with MapReduce**
- More MR Design Patterns
- (YARN Resource Management and Scheduling)

Implementing Join in Hadoop

- Two basic options
- **Option 1:** DistributedCache
  - load smaller join table into the 'distributed cache' of Hadoop
  - will be available via the 'Context' parameter to all reducers as a main-memory structure
  - Pro: Easy, fast
  - Con: Memory overhead; join data is replicated to all reducers, whether needed or not
  - ONLY use this for very small join tables
- **Option 2:** Reduce-side Join with heterogeneous mappers
  - preferred option, because it scales well
  - details on next slides

A Table Join Example

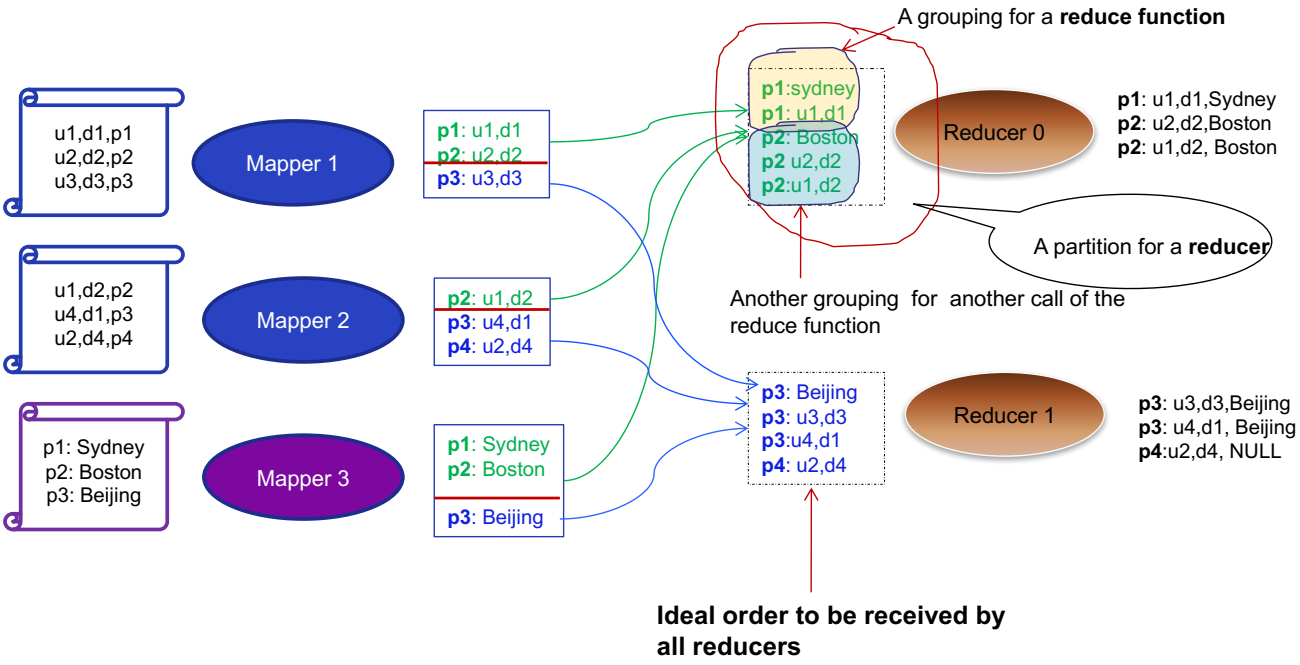
- Two input sources
  - **Photo:** `user \t date \t place_id \n`
  - **Place:** `place_id \t woeid \t lat \t longi \t place_name \t place_url`
- Output:
  - `Place_id \t user \t date \t place_name1`
  - `Place_id \t user \t date \t place_name2`
- This is like joining two tables on key **place_id**

```
Map:
 photo record -> (k:place_id, v: user \t date)
 place record ->(k:place_id, v: place_name)
Reduce:
(k:place_id,v:(place_name, user1 \t date1, user2 \t date2)) ->
         (K:place_id, v: user1 \t date1 \t place_name),
         (K:place_id, v: user2 \t date2 \t place_name),
```


Implementing "Join" in Hadoop

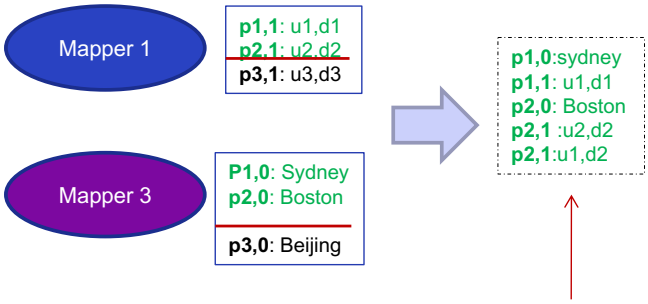
- Several new features
  - Multiple inputs with different formats and hence **different Mappers**
  - There should be a way to identify values coming from various input since the values are passed to a same reduce function
    - One easy way is to order the values in a particular way
    - Value from the **Place** table, which contains the place name should be the first in the value list
- Multiple input is supported in Hadoop framework
- Values are grouped based on key
  - Need a way to "label" the key to differentiate keys from different input data
  - Need special handling to ensure the labelled keys maintain their original partition position and the values from the say key are grouped as a list to send to the same reduce function

Example of Join Execution



Composite Key

- We want to sort the map output based on key as well as where it comes from
  - Label the key with additional information to indicate where the (key,value) pair comes from.



The order of reducer input kv pairs is determined by its map output key's `Comparator`;

The allocation of map output key to partition is determined by a given or default `Partitioner`.
The default `Partitioner` guarantees to put same key in a same partition

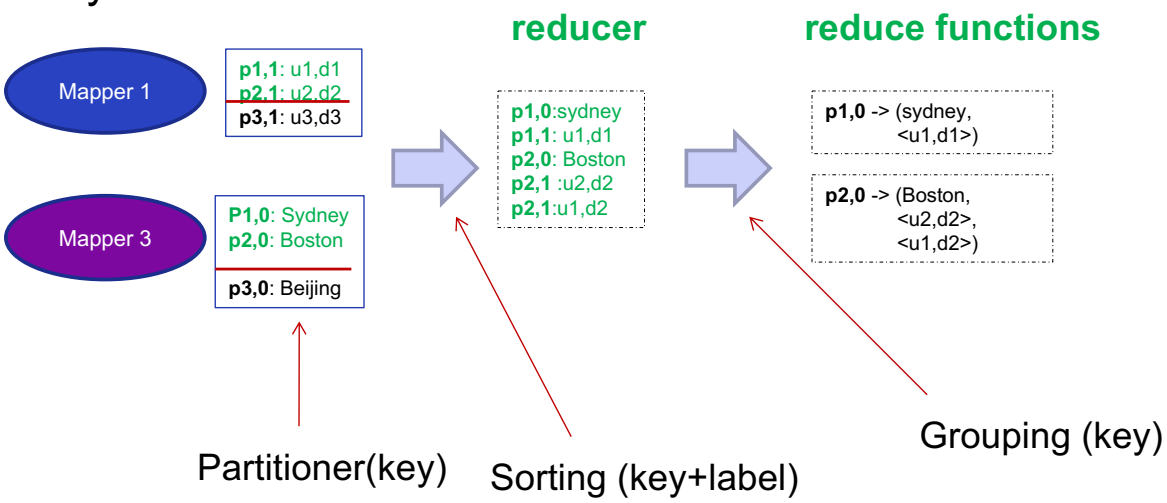
The grouping of key value pairs for each call of `reduce` function is determined by a grouping comparator; if not specified, the map output key's `Comparator` is used.

Sort by original key, then by the label

Using default partitioner, there is no guarantee that composite keys (**p1,0**) and (**p1,1**) go to the same partition, or composite keys (**p2,0**) and (**p2,1**) go to the same partition

Custom Partitioner and Grouping Comparator

- We should sort all reducer input based on composite key's comparator which takes into account both the original key and the additional label
- We should *partition* and *group* based on the original key only



The Composite Key

```
public class TextIntPair implements WritableComparable<TextIntPair>{

    private Text key;
    private IntWritable order;

    //standard accessor methods for the private fields are omitted; default constructor is omitted

    public TextIntPair(String key, int order){
           this.key = new Text(key);this.order = new IntWritable(order);
    }
    //serializing and deserializing methods
    public void readFields(DataInput in) throws IOException {key.readFields(in);order.readFields(in);}
    public void write(DataOutput out) throws IOException {key.write(out);order.write(out);}

    public int compareTo(TextIntPair other) { //this comparator will be used during sorting
           int cmp = key.compareTo(other.key);
           if (cmp != 0) {return cmp;}
           return order.compareTo(other.order);
    }

    public int hashCode() {return key.hashCode() * 163 + order.get();}

    public boolean equals(Object other) {
           if (other instanceof TextIntPair) {
                  TextIntPair tip = (TextIntPair) other;
                  return key.equals(tip.key) && order.equals(tip.order);
           }
           return false;
    }
}
```

Partitioner and Group Comparator

```
public class JoinPartitioner implements Partitioner<TextIntPair,Text> {
    public int getPartition(TextIntPair key, Text value, int numPartition) {
        return (key.getKey().hashCode() * 123) % numPartition;
    }
    public void configure(JobConf arg0) {
    }
}
```

```
public class JoinGroupComparator extends WritableComparator {

  protected JoinGroupComparator() {super(TextIntPair.class,true);}

/**
 * Only compare the key when grouping reducer input together
 */
  public int compare(WritableComparable w1, WritableComparable w2) {
        TextIntPair tip1 = (TextIntPair) w1;
        TextIntPair tip2 = (TextIntPair) w2;
        return tip1.getKey().compareTo(tip2.getKey());
  }
}
```

The full code including two mappers, one reducer and one driver can be downloaded from ELearning

Distributing Auxiliary Job data

- Auxiliary job data
  - In general a small file contains common background knowledge for processing map jobs
    - E.g. the stop word list for word counting, the dictionary for spelling check
  - All mappers need to read it
  - The file is small enough to fit in the memory of mappers
- Hadoop provides a mechanism for this purpose called the **`distributed cache`**.
  - Files put in the distributed cache is accessible by both mappers and reducers.
- Distributed cache can be used to provide an alternative, more efficient join if one join table is small enough to fit in the memory
  - Only the map stage is needed, sometimes called replicated join
  - See an example in the lab!

Summarizing MR programming features

- A Job = *m**Mapper + *r**Reducer
  - r could be zero
- To minimize shuffling bytes
  - A Job = *m** (Mapper + Combiner) + *r**Reducer
- We can manipulate data presented to reducer job by
  - Customized partitioner, composite key, group comparator, output key
- A complex workload can have multiple jobs:
  - previous job's output becomes the next job's input
- Small input files can be shared by all nodes
- We can have multiple input files with different formats

$$\text{A job} = \left\{ \begin{array}{l} m1 * \text{Mapper1} \\ m2 * \text{Mapper2} \\ m3 * \text{Mapper3} \end{array} \right\} + r*\text{Reducer}$$

Agenda

- MR Design Patterns
- Total Order Sorting
- Joining with MR
- **More MR Design Patterns**
- (YARN Resource Management and Scheduling)

Distinct Results

- Goal: How to make sure we aggregate some value only once?
  - For example: How many distinct ratings were given to a movie?

- Pattern:
  - Introduce a custom Hadoop CompositeKey type
    - similar to our solution forthe reduce-side Join
    - implements `org.apache.hadoop.io.WritableComparable<T>` interface
    - stores multiple components: out_key and a sub-out_key(s)
    - its `compareTo()` method should compare on all (sub-)components
      - => hence during Shuffle phase, sorting by out_key and sub-out_key
  - Custom Partitioner which only partitions by *out_key* but ignores rest
  - In reducer, input values now in sort order
    - Single scan through list; process each distinct value only once

Top-K?

- Goal: How to determine the top-*k* results?

- Idea: Similar to distinct
  - Follow the design pattern for determining distinct results
  - Instead of returning each distinct value only once, return the top-k results.

Binning

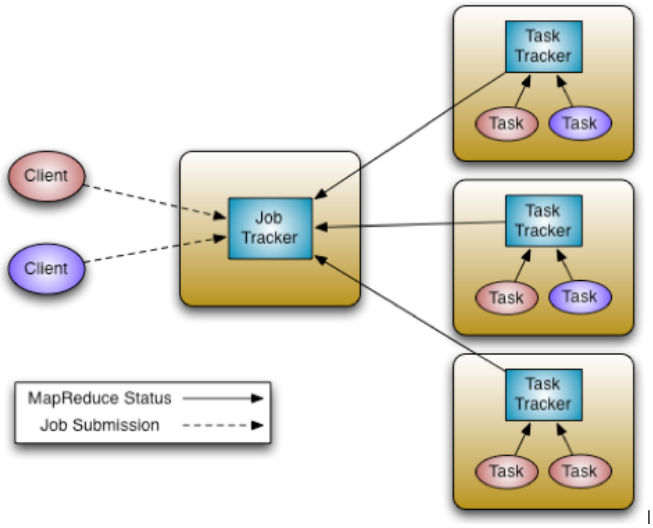
- to be done…

Outline

- MR Design Patterns
- Total Order Sorting
- Joining with MapReduce
- More MR Design Patterns
- **(YARN Resource Management and Scheduling)**

MRv1 Framework -- Overview

- The early version of Hadoop has a master node called JobTracker
  - The ***JobTracker*** knows before hand the number of mappers and number of *reducers* for each job.
  - It also knows the number of available *task slots* on each worker node
  - It is responsible for assigning tasks to node with vacant slot
  - The default and the simplest scheduling using **FIFO** queue



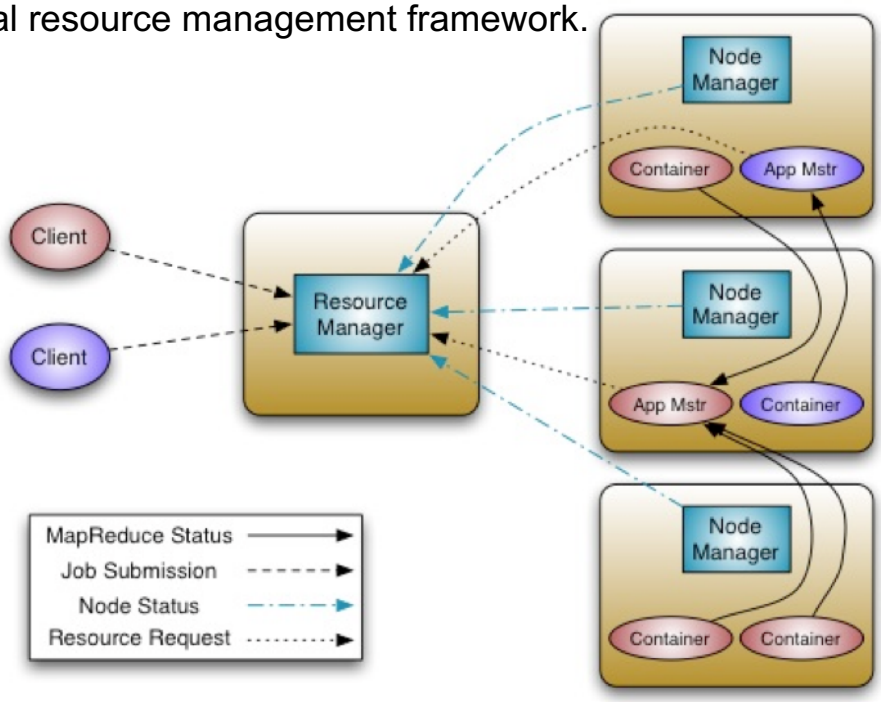
http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/

MRv1 Framework -- Issues

- Scalability
  - JobTracker has lots of responsibilities
    - Allocate resources
    - Monitor tasks progress and overall node status
      - Start speculative tasks
      - Kill tasks
      - Restart tasks
      - And so on..
    - MRv1 hits scalability bottlenecks in the region of 4,000 nodes and 40,000 tasks [https://developer.yahoo.com/blogs/hadoop/next-generation-apache-hadoop-mapreduce-3061.html]
- Resource Utilization
  - TaskTracker has fixed map slots and reduce slots
    - A node with two cores are usually configured to have two map slots and two reduce slots
    - Map tasks cannot be assigned to reduce slots and vice versa
    - There are cases when map slots might be full while reduce slots are empty and vice versa

YARN Framework

- Yet Another Resource Negotiator
  - A general resource management framework.



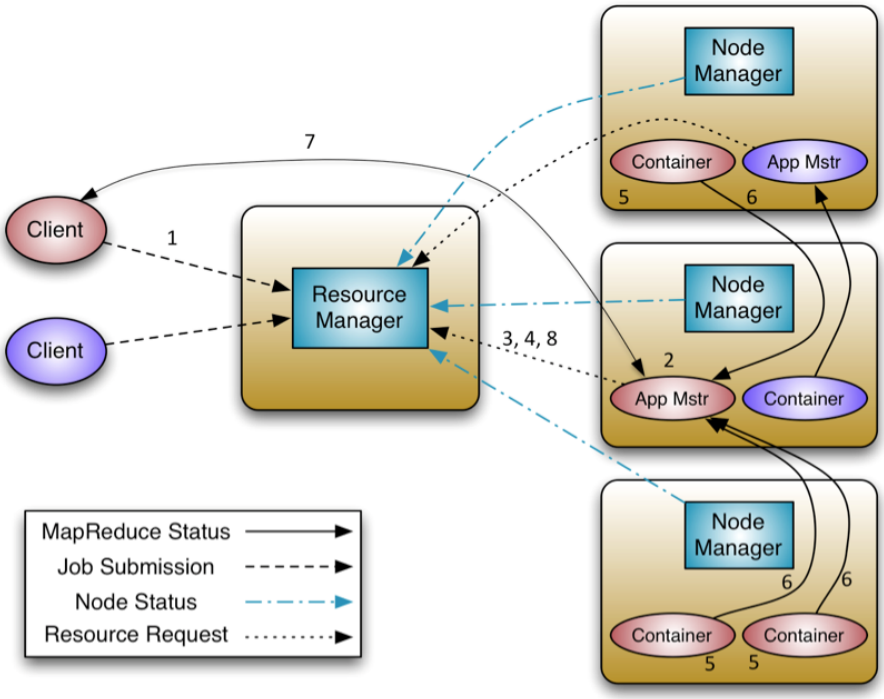
YARN Concepts

- YARN's world view consists of *applications* requesting *resources*
- Resource Manager (one per cluster)
  - "Is primarily, a *pure scheduler*. In essence, it's strictly limited to arbitrating available resources in the system among the competing applications"
- Node Manager (one per node)
  - Responsible for managing resources on individual node
- ApplicationMaster (one per application)
  - Responsible for requesting resources from the cluster on behalf of an application
  - Monitor the execution of application as well
  - Framework specific
    - MapReduce application's AM is different ot Spark application's AM

YARN Resource Concepts

- Resource Request
  - In the current model, resource requirements are expressed mainly in Memory and/or CUP cores
  - Depends on configured scheduler
- Container
  - Resource allocation is in the form of container
  - A Container grants rights to an application to use a specific amount of resources (memory, cpu etc.) on a specific host.
  - MR framework uses containers to run map or reduce tasks
  - Container is launched by NodeManager
- A MapReduce Job with 10 map tasks and 3 reduce tasks would need 14 containers. The first container runs its AM, which would request 10 containers for map tasks and 3 containers for reduce tasks.

YARN Walkthrough



YARN and MR memory configuration

- YARN needs to know
  - The total amount of memories allocated to YARN
  - How to break up the total resources into containers
    - Minimum container size
- ApplicationMaster for MapReduce application needs to know
  - Memory requirement for Map and Reduce tasks
    - We usually assume Reduce tasks need more memory
  - Each task is running on a JVM, the JVM heap size needs to be set as well

Memory Configuration Example

- Node capacity
  - 48G Ram, 12 core
- Memory allocation
  - Reserve 8G Ram for OS and others (e.g. HBase)
  - YARN can use up to 40G
  - Set Minimum container size to 2G
    - Each node may have at most 20 containers
  - Allow each map task to use 4G and reduce task to use 8G
    - Each node may run 10 map tasks or 5 reduce task or a combination of the two

Resource Scheduling

- Resource Scheduling deals the problem of which requests should get the available resources when there is a queue of resources
- Early MRv1 uses FIFO as default scheduling algorithm
  - Large job that comes first may starve small jobs
  - Users may experience long delay as their jobs are waiting in queue
- YARN is configured to support shared multi-tenant cluster
  - Capacity Scheduler
  - Faire Scheduler
  - FIFO Scheduler

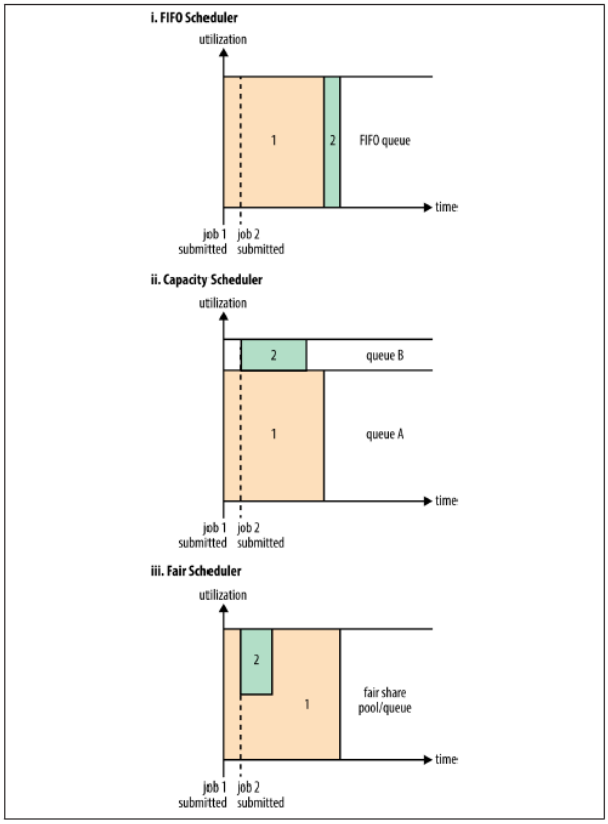


Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

Hadoop: the definitive guide, 4th edition, page 87

Capacity Scheduler

- Capacity Scheduler is designed for multi-tenancy situation
  - Each organization has a dedicated queue with a given fraction of the cluster capacity
    - Queues may be further divided to set up capacity allocation within organization
    - A single job does not use more resources than the queue capacity, but if there are more than one job in the queue, and there are idle resources, spare resources can be allocated to jobs in the queue
    - How much more resources can be allocated to a given queue is configurable
    - Which users can submit jobs to which queue is configurable

http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

Fair Scheduler

- The principle of fair scheduling is to allocate resources so that all running applications get similar share of resources
  - E.g. if there are R units of resources and J jobs, each job should get around R /J units of resources.
- YARN fair scheduler works between queues: each queue gets a fair share of the cluster resources
  - Within queue, the scheduling policy is configurable: FIFO or Fair

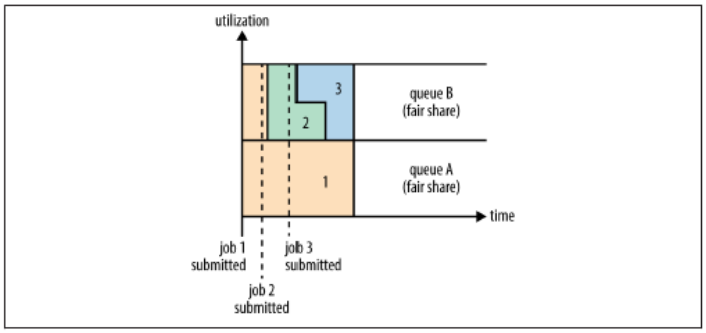


Figure 4-4. Fair sharing between user queues

Hadoop: the definitive guide, 4th edition, page 91

Fair Scheduler – Queue Placement

- Fair scheduler uses a rule based system to determine which queue an application should be placed in
  - Example queue placement policy

```
<queuePlacementPolicy>
    <rule name="specified" />
    <rule name="user" />
</queuePlacementPolicy>
```

- All queues in capacity scheduler needs to be specified beforehand by cluster administrator; Queues in fair scheduler can be created on the fly

Hadoop job execution profile

hadoop jar userTag.jar usertag.TagDriver /share/photo/n08.txt userN08Out

MapReduce Job job_1456812395129_0957

	Replicas	Map Task Running on
Block 0	soit-hdp-pro-[13,14,15].ucc.usyd.edu.au	soit-hdp-pro-15.ucc.usyd.edu.au
Block 1	soit-hdp-pro-[4,11,15].ucc.usyd.edu.au	soit-hdp-pro-[**15,8**].ucc.usyd.edu.au
Block 2	soit-hdp-pro-[8,10,15].ucc.usyd.edu.au	soit-hdp-pro-15.ucc.usyd.edu.au
Block 3	soit-hdp-pro-[6,12,15].ucc.usyd.edu.au	soit-hdp-pro-15.ucc.usyd.edu.au

References

- Tom White, Hadoop The Definitive Guide, 4th edition, O'Reilly, 2015
  - Chapter 4, YARN
  - Chapter 9 , MapReduce Features
- Donald Miner and Adam Shook, "Map Reduce Design Patterns", O'Reilly, 2013.
  - Available as e-book online in Usyd library
- Apache Hadoop Yarn Introduction
  - http://hortonworks.com/blog/introducing-apache-hadoop-yarn/
  - http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/
- Determine YARN and MapReduce Memory configuration Settings
  - http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.6.0/bk_installing_manually_book/content/rpm-chap1-11.html
- How to plan and configure YARN and MapReduce 2 in HDP
  - http://hortonworks.com/blog/how-to-plan-and-configure-yarn-in-hdp-2-0/
- Job Scheduling
  - FairScheduler: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html
  - CapacityScheduler: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html