CS 398 ACC
MapReduce Part 2

Prof. Robert J. Brunner

Ben Congdon
Tyler Kim

Change in Quiz Policy

Starting with Quiz 2:

- Only **two** attempts allowed
  - But, you can see all that you got right/wrong

MP1

How’s it going?

Tentative Auto-Grader Schedule

- Wednesday Evening (~9pm)
- Friday Evening (~9pm)
- Sunday Evening (~9pm)
- Monday Evening (~9pm)
- Tuesday Midday (~2pm)

- Results location: /mp1/grade_report.txt
- This will be posted on Piazza as well.

Outline

- MapReduce Programming
  - Word Count Implementation
  - Conventions / Pitfalls
- Execution Options
- MapReduce Use Cases

Outline

- **MapReduce Programming**
  - Word Count Implementation
  - Conventions / Pitfalls
- Execution Options
- MapReduce Use Cases

Reminders...

<key_input, val_input> ⇒ <key_inter, val_inter> ⇒ <key_out, val_out>

Map ***Reduce***

- **Map**:
  - **Input**: “Original” input data or key/value pairs from previous chained job
  - **Output**: Intermediate key/value pairs
- **Reduce**:
  - **Input**: Intermediate key/value pair (per key)
  - **Output**: Final key/value pairs

Word Count - Mapper

```
...

class WordCount(MRJob):
    def mapper(self, key, val):
        for word in WORD_REGEX.findall(val):
            yield (word, 1)

...
```

Word Count - Reducer

```python
def reducer(self, key, vals):
    total_sum = 0

    # Iterate and count all occurrences of the word
    for v in vals:
        total_sum += 1

    # Yield the word and number of occurrences
    yield key, total_sum
```

Reminders...

- **Python Iterators**

```python
def more_efficient(self, key, values):
    for v in values:
        yield key, v + 10

def inefficient(self, key, values):
    # List comprehension loads all values into memory
    plus_10 = [v + 10 for v in values]
    for v in plus_10:
        yield key, v
```

MapReduce - Common Conventions

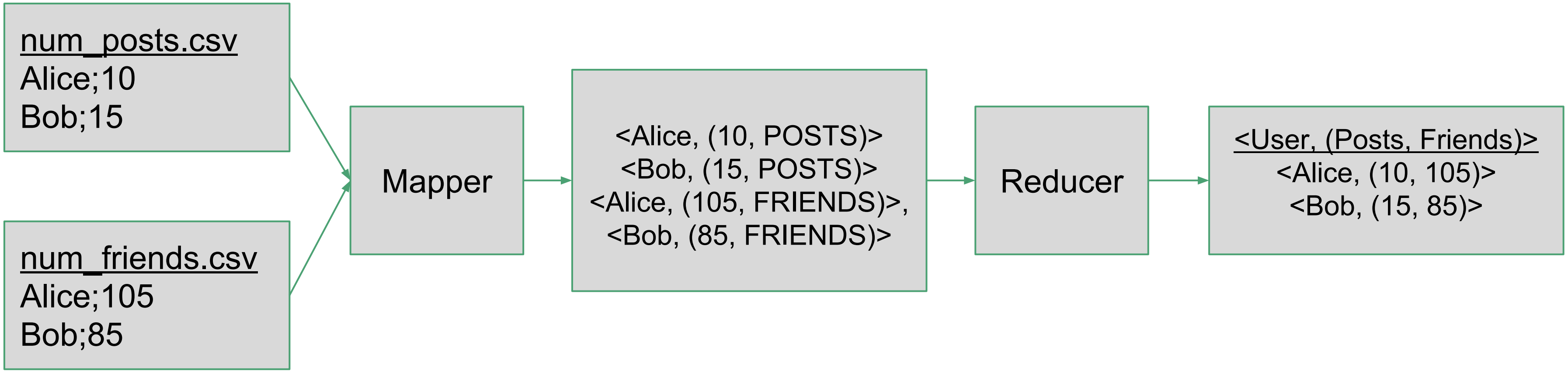
- **Composite Key**
  - Use more than one attribute in the construction of a key
    - i.e. <(city, state), population>

- **Composite Value**
  - Use more than one attribute in the construction of a value
    - i.e. <user_id, (num_friends, num_posts)>

  - Can also use custom serialization methods for intermediate values
    - i.e. JSON, Python Pickling
    - Just be careful about size overhead (for bandwidth)

MapReduce - Common Conventions

- **Joins**
  - Idea: Use input datasets with more than one format
  - Mapper: Add flag to output to indicate value type
  - Reducer: Reconcile attributes by key into a single record



MapReduce - Common Pitfalls

- **“Leaky” Reducers**
  - **Source**: Reducers that use too much memory (i.e. keeping all values in memory)
    - Reducing functions have “too much” state
    - Might not be due to bad reducer design, but rather empirical workload/machine limitations

MapReduce - Common Pitfalls

- **"Leaky" Reducers**
  - **Example:**

```python
# Want to count number of unique values
def reduce(self, key, values):
    # 'unique' will store all distinct values we see
    unique = set()
    for v in values:
        if v not in unique:
            unique.add(v)
    # yield size of unique value set
    yield (key, len(unique))
```

MapReduce - Common Pitfalls

- **“Leaky” Reducers**
  - **Source**: Reducers that use too much memory (i.e. keeping all values in memory)
    - Reducing functions have “too much” state
    - Might not be due to bad reducer design, but rather empirical workload/machine limitations
  - **Solutions:**
    - Benchmark workload - Maybe it’s not an issue
    - Take advantage of secondary sort
    - Use the fact that values are passed as an iterator (Use a “stream” mindset)

MapReduce - Common Pitfalls

- **“Leaky” Reducers**
  - **Example (Fixed):**

```
# Want to count number of unique values
def reduce(self, key, values):
    prev, count = None, 0

    # Here we assume that values is sorted
    for v in values:
        if v != prev:
            count += 1
        prev = v

    yield (key, count)
```

MapReduce - Common Pitfalls

- **“Hot” Keys**
  - **Source:** Some keys may contain many, many more values than most other keys

MapReduce - Common Pitfalls

- **"Hot" Keys**
  - **Example:**
    - Google Web Indexing
    - Average Key Size: 300 KB
    - Some keys have 50+ GB

Reduce key	Reduce input size
*.blogspot.com	82.9G
cgi.ebay.com	58.2G
profile.myspace.com	56.3G
yellowpages.superpages.com	49.6G
www.amazon.co.uk	41.7G
average reduce input size for a given key	300K

Source: https://research.google.com/pubs/pub36249.html

MapReduce - Common Pitfalls

- **“Hot” Keys**
    - **Source:** Some keys may contain many, many more values than most other keys
    - **Solutions**:
        - Benchmark workload - Maybe it’s not an issue
        - Write a custom partitioner so that load is still distributed evenly across machines
            - Partitioner determines which intermediate keys go to which reducer
            - Goal: Distribute load evenly so “hot keys” don’t all go to one reducer

        - Can you split up keys further and recombine them in a chained MR step?

MapReduce - Common Pitfalls

- Python Specific: **Handling Types**
  - Input data is almost always `str` / `bytes`
  - Java allows you to define custom types and serialization
  - Python can do this too (i.e. with Pickling), but it is not required

Outline

- MapReduce Programming
    - Word Count Implementation
    - Conventions / Pitfalls
- **Execution Options**
- MapReduce Use Cases

MapReduce Execution Options

- Hadoop (native Java)
- Hadoop Streaming
- External Frameworks (MRJob)

MapReduce Programming Options

- **Hadoop MapReduce (native)**
  - Preferred, most performant method for writing Hadoop MapReduce jobs
  - Minimum Required Components per Job:
    - Mapper, Reducer, Job execution boilerplate
  - Additional Customizable Components:
    - **InputFormat**: Splits input files into chunks to be distributed to mappers
    - **Partitioner**: Controls which keys go to which reducers (default: HashPartitioner)
    - **OutputFormat / OutputComitter**: Handles the end of the reduce phase (usually writing job output to disk)

MapReduce Programming Options

- **Hadoop Streaming**
  - Hadoop copies arbitrary binary executable(s) for mappers and reducers
  - Uses STDIN/STDOUT to stream data to mappers/reducers
    - **Mapper**: Each input record is a new line
    - **Reducer**: You receive a stream of arbitrary k/v pairs (sorted by key)
      - You (the program) have to figure out when you switch from one key to the next
  - Still parallel, but difficult to work in
    - Everything is text; key/values are usually just tab separated
  - Allows non-Java languages to be used on the Hadoop Framework

MapReduce Programming Options

- Aside: Local Unix Commands
    - `$ echo $DATA | ./mapper | sort -k1,1 | ./reducer > output`
    - Not parallel (or advisable), but useful for debugging Hadoop Streaming jobs
    - Similar to Hadoop Streaming in that you use an arbitrary executable as a mapper / reducer

MapReduce Programming Options

- **MRJob (External Framework)**
  - Python Framework for writing / running MapReduce jobs
  - Built by Yelp
  - Write Once, Run “Anywhere” (actually, though!)
  - Supported Execution Environments:
    - Local Execution (MP1)
    - Hadoop (MP2)
    - GCP Dataproc - Google’s Hosted MapReduce
    - AWS EMR - Amazon’s Hosted Hadoop

MapReduce Programming Options

- **MRJob (External Framework)**
  - How does it work?
    - Hadoop Streaming under-the-hood
    - Provides similar abstractions as the Native Java API

Outline

- MapReduce Programming
  - Word Count Implementation
  - Conventions / Pitfalls
- Execution Options
- **MapReduce Use Cases**

Distributed Grep

Used for: Filtering, Parsing, or Validation

- **Input**: Large set of files
- **Mapper:** Look at input and emit records containing the query term
- **Reducer**: Pass through all records unchanged

Graph Processing

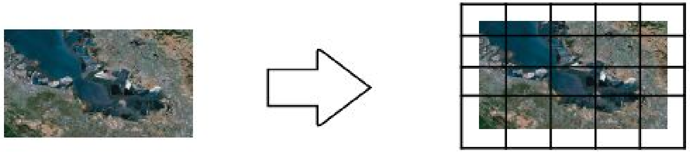
Web-Linked / Web Scraping Graph (Similar to Problem 2 of MP1)

- **Input**: HTML Text
- **Map output**: <target, source> pairs
    - i.e. Search for <a href=”...”>
- **Reduce output**: <target, list(source)> pairs

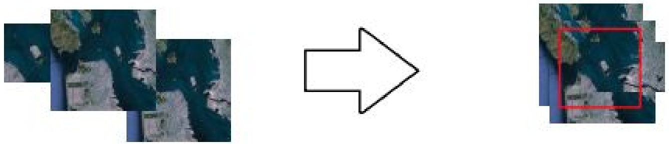
Used by Google for web search indexing

Geospatial / Satellite Data

- **Input**: Geospatial coordinates, satellite data
- **Map output**:
    - <map_tile, tile_information> pairs
    - “Chunk” geographic region by tile



- **Reduce output**:
    - <map_tile, final_tile_info> pairs



Used by Google Maps to reconcile satellite imagery over time

Source: https://research.google.com/pubs/pub36249.html

What is required:

Programmer:

1. Don’t need to know specifics about parallel/distributed computing/programming.
2. Know the data source/format
3. Write a map/reduce programs
4. Submit jobs and wait :)

What is required:

Framework/Library (e.g. mrjob, hadoop, etc.):

1. Parallelize Map (distribute to mapping machines)
2. Transfer Data / Shuffle Data
3. Parallelize Reduce (distribute to reducing machines)
4. Deal with failure, missing values
5. Implement data transfer. Input/Output/Artifacts. Interact with distributed file system

Next Week:

- Lets you run MapReduce on many computers for a single task.
- Can scales to 1000s of nodes
- Processes Petabytes with (relative) ease
- Get on course cluster!