Principles of Software Construction: Objects, Design, and Concurrency
Part 6: Concurrency and distributed systems

MapReduce

Christian Kästner  Charlie Garrod
Administrivia

- Homework 5c due tomorrow (Wednesday) night, 11:59 p.m.
- Homework 6 (MapReduce!) released by Friday
- Final exam Tuesday, May 5th, 1 – 4 p.m. DH 2210
  - Final exam review session Sunday, May 3rd, 4 – 6:30 p.m., Hamburg 1000
Key concepts from last Thursday
Some distributed system design goals

• The end-to-end principle
  – When possible, implement functionality at the ends (rather than the middle) of a distributed system

• The robustness principle
  – Be strict in what you send, but be liberal in what you accept from others
    • Protocols
    • Failure behaviors

• Benefit from incremental changes

• Be redundant
  – Data replication
  – Checks for correctness
Partitioning for scalability

- Partition data based on some property, put each partition on a different server

```
client
  → front-end
  → CMU server: {cohen:9, bob:42, ...}
  → Yale server: {alice:90, pete:12, ...}
  → MIT server: {deb:16, reif:40, ...}
```

client
  → front-end

Consistent hashing

• Goal: Benefit from incremental changes
  – Resizing the hash table (i.e., adding or removing a server) should not require moving many objects
• E.g., Interpret the range of hash codes as a ring
  – Each bucket stores data for a range of the ring
    • Assign each bucket an ID in the range of hash codes
    • To store item $X$ don't compute $X$.hashCode() \(\%\ n\). Instead, place $X$ in bucket with the same ID as or next higher ID than $X$.hashCode()
Master/tablet-based systems

- Dynamically allocate range-based partitions
  - Master server maintains tablet-to-server assignments
  - Tablet servers store actual data
  - Front-ends cache tablet-to-server assignments

```
client
front-end

client
front-end

Master:
{k-z:
{pete:12,
reif:42}}

Tablet server 1:
{k-z:
{pete:12,
reif:42}}

Tablet server 2:
a-c:
{alice:90,
ob:42,
cohen:9}
d-g:
{deb:16}
h-j:
{}

Tablet server 3:
d-g:
{deb:16}
h-j:
{}

Tablet server 4:
d-g:
{deb:16}
```
Today: Distributed system design

- MapReduce: A robust, scalable framework for distributed computation...
  - ...on replicated, partitioned data
Map from a functional perspective

- \( \text{map}(f, x[0...n-1]) \)
  - Apply the function \( f \) to each element of list \( x \)

- E.g., in Python:
  ```python
  def square(x): return x*x
  map(square, [1, 2, 3, 4]) would return [1, 4, 9, 16]
  ```
- Parallel map implementation is trivial
  - What is the work? What is the depth?
Reduce from a functional perspective

- **reduce**(f, x[0…n-1])
  - Repeatedly apply binary function f to pairs of items in x, replacing the pair of items with the result until only one item remains
  - One sequential Python implementation:
    ```python
def reduce(f, x):
    if len(x) == 1: return x[0]
    return reduce(f, [f(x[0],x[1])] + x[2:])
```
  - e.g., in Python:
    ```python
def add(x,y): return x+y
reduce(add, [1,2,3,4])
```
    would return 10 as
    ```python
reduce(add, [1,2,3,4])
reduce(add, [3,3,4])
reduce(add, [6,4])
reduce(add, [10]) -> 10
```
Reduce with an associative binary function

- If the function $f$ is associative, the order $f$ is applied does not affect the result

1 + ((2+3) + 4)  
1 + (2 + (3+4))  
(1+2) + (3+4)

- Parallel reduce implementation is also easy
  - What is the work? What is the depth?
Distributed MapReduce

- The distributed MapReduce idea is similar to (but not the same as!):
  \[ \text{reduce}(f_2, \text{map}(f_1, x)) \]

- Key idea: "data-centric" architecture
  - Send function \( f_1 \) directly to the data
    - Execute it concurrently
  - Then merge results with reduce
    - Also concurrently

- Programmer can focus on the data processing rather than the challenges of distributed systems
MapReduce with key/value pairs (Google style)

- **Master**
  - Assign tasks to workers
  - Ping workers to test for failures

- **Map workers**
  - Map for each key/value pair
  - Emit intermediate key/value pairs

- **Reduce workers**
  - Sort data by intermediate key and aggregate by key
  - Reduce for each key
MapReduce with key/value pairs (Google style)

- E.g., for each word on the Web, count the number of times that word occurs
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is a word, values is a list of the number of counts of that word

```java
f1(String key1, String value):
    for each word w in value:
        EmitIntermediate(w, 1);

f2(String key2, Iterator values):
    int result = 0;
    for each v in values:
        result += v;
    Emit(key2, result);
```

Map: (key1, v1) \(\rightarrow\) (key2, v2)*
Reduce: (key2, v2*) \(\rightarrow\) (key3, v3)*

MapReduce: (docName, docText)* \(\rightarrow\) (word, wordCount)*
MapReduce architectural details

• Usually integrated with a distributed storage system
  – Map worker executes function on its share of the data

• Map output usually written to worker's local disk
  – Shuffle: reduce worker often pulls intermediate data from map worker's local disk

• Reduce output usually written back to distributed storage system
Handling server failures with MapReduce

- **Map worker failure:**
  - Re-map using replica of the storage system data

- **Reduce worker failure:**
  - New reduce worker can pull intermediate data from map worker's local disk, re-reduce

- **Master failure:**
  - Options:
    - Restart system using new master
    - Replicate master
    - ...
The beauty of MapReduce

• Low communication costs (usually)
  – The shuffle (between map and reduce) is expensive

• MapReduce can be iterated
  – Input to MapReduce: key/value pairs in the distributed storage system
  – Output from MapReduce: key/value pairs in the distributed storage system
MapReduce to count mutual friends

• E.g., for person in a social network graph, output the number of mutual friends they have
  ▪ For Map: key1 is a person, value is the list of her friends
  ▪ For Reduce: key2 is ???, values is a list of ???

f1(String key1, String value):

f2(String key2, Iterator values):

MapReduce: (person, friends)* → (pair of people, count of mutual friends)*
MapReduce to count mutual friends

- E.g., for person in a social network graph, output the number of mutual friends they have
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is a pair of people, values is a list of 1s, for each mutual friend that pair has

\[
\text{f1(String key1, String value):} \\
\quad \text{for each pair of friends in value:} \\
\quad \quad \text{EmitIntermediate(pair, 1)};
\]

\[
\text{f2(String key2, Iterator values):} \\
\quad \text{int result = 0;} \\
\quad \text{for each v in values:} \\
\quad \quad \text{result += v;} \\
\quad \text{Emit(key2, result);} \\
\]

MapReduce: (person, friends)* \rightarrow (pair of people, count of mutual friends)*
MapReduce to count incoming links

• E.g., for each page on the Web, count the number of pages that link to it
  ▪ For Map: key1 is a document name, value is the contents of that document
  ▪ For Reduce: key2 is ???, values is a list of ???

\[
\begin{align*}
\text{MapReduce: } & \quad \text{(docName, docText)}^* \rightarrow \text{(docName, number of incoming links)}^* \\
\text{f1(String key1, String value):} & \quad \text{f2(String key2, Iterator values):}
\end{align*}
\]
MapReduce to count incoming links

- E.g., for each page on the Web, count the number of pages that link to it
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is link, values is a list of 1s

f1(String key1, String value):
  for each link in value:
    EmitIntermediate(link, 1)

f2(String key2, Iterator values):
  int result = 0;
  for each v in values:
    result += v;
  Emit(key2, result);

MapReduce: (docName, docText)^* → (docName, number of incoming links)^*
MapReduce to create an inverted index

- E.g., for each page on the Web, create a list of the pages that link to it
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is ???, values is a list of ???

```java
f1(String key1, String value):
    for each link in value:
        EmitIntermediate(link, key1)
```

```java
f2(String key2, Iterator values):
    Emit(key2, values)
```

MapReduce: (docName, docText)* → (docName, list of incoming links)*
List the mutual friends

- E.g., for each pair in a social network graph, list the mutual friends they have
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is ???, values is a list of ???

\[
\begin{align*}
\text{f1(String key1, String value):} & \quad \text{f2(String key2, Iterator values):}
\end{align*}
\]

MapReduce: (person, friends)* \rightarrow (pair of people, list of mutual friends)*
List the mutual friends

- E.g., for each pair in a social network graph, list the mutual friends they have
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is a pair of people, values is a list of their mutual friends

```java
f1(String key1, String value):
    for each pair of friends in value:
        EmitIntermediate(pair, key1);
```

```java
f2(String key2, Iterator values):
    Emit(key2, values)
```

MapReduce: (person, friends)* → (pair of people, list of mutual friends)*
Count friends + friends of friends

• E.g., for each person in a social network graph, count their friends and friends of friends
  ▪ For Map: key1 is a person, value is the list of her friends
  ▪ For Reduce: key2 is ???, values is a list of ???

\[ \text{MapReduce: (person, friends)*} \rightarrow (\text{person, count of f + fof})* \]
Count friends + friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is ???, values is a list of ???

```java
f1(String key1, String value):
  for each friend1 in value:
    EmitIntermediate(friend1, key1)
  for each friend2 in value:
    EmitIntermediate(friend1, friend2);

f2(String key2, Iterator values):
  distinct_values = {}
  for each v in values:
    if not v in distinct_values:
      distinct_values.insert(v)
  Emit(key2, len(distinct_values))
```

MapReduce: (person, friends)* → (person, count of f + fof)*
Friends + friends of friends + friends of friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends and friends of friends of friends
  - For Map: key1 is a person, value is the list of her friends
  - For Reduce: key2 is ???, values is a list of ???

\[
f1(String\ \text{key1, String\ value}): \quad f2(String\ \text{key2, Iterator\ values}):\\
\]

\[
\text{MapReduce: (person, friends)* } \rightarrow \text{ (person, count of f + fof + fofof)*}\]
Problem: How to reach distance 3 nodes?

• Solution: Iterative MapReduce
  – Use MapReduce to get distance 1 and distance 2 nodes
  – Feed results as input to a second MapReduce process

• Also consider:
  – Breadth-first search
  – PageRank
  – ...
Dataflow processing

• High-level languages and systems for complex MapReduce-like processing
  – Yahoo Pig, Hive
  – Microsoft Dryad, Naiad
• MapReduce generalizations...
No class Thursday: Carnival