MapReduce

Dominic Duggan
Stevens Institute of Technology
Based on materials by
Srinivasan Seshan, A. Haeberlen, Z. Ives

MapReduce

- Programming model + infrastructure
- Write programs that run on lots of machines
- Automatic parallelization and distribution
- Fault-tolerance
- Scheduling, status and monitoring
- Application: **Big Data** processing

Parallelism

- Task Parallelism
  - Monitors, futures, etc
- Pipeline Parallelism
  - E.g. executing instruction on FPU while loading next instruction from memory
- **Data Parallelism**
  - Independent action on items in a collection

3

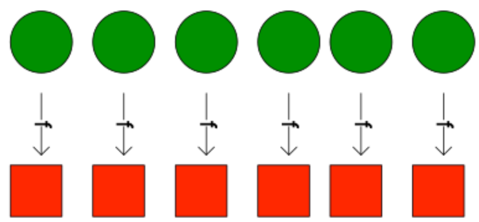
Functional Programming

- "Declarative" programming
- Referential transparency
  - No mutation of variables
  - Equational reasoning
- Implicit control flow
- Parallel functional programming
  - Data parallelism

4

Map

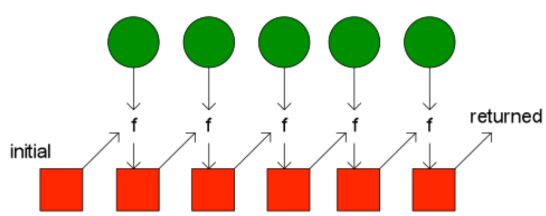
- Applies an operation *f* to each element of a list
- Returns a new list as its result
- Each element of new list is the result of applying the operation to the corresponding element of the input list



5

Reduce (aka Fold)

- Moves across a list, applying **operation** *f* to each element plus an **accumulator**.
- *f* returns the next accumulator value, which is combined with the next element of the list
- *f*: associative and commutative



6

Examples: What is “f” operation?

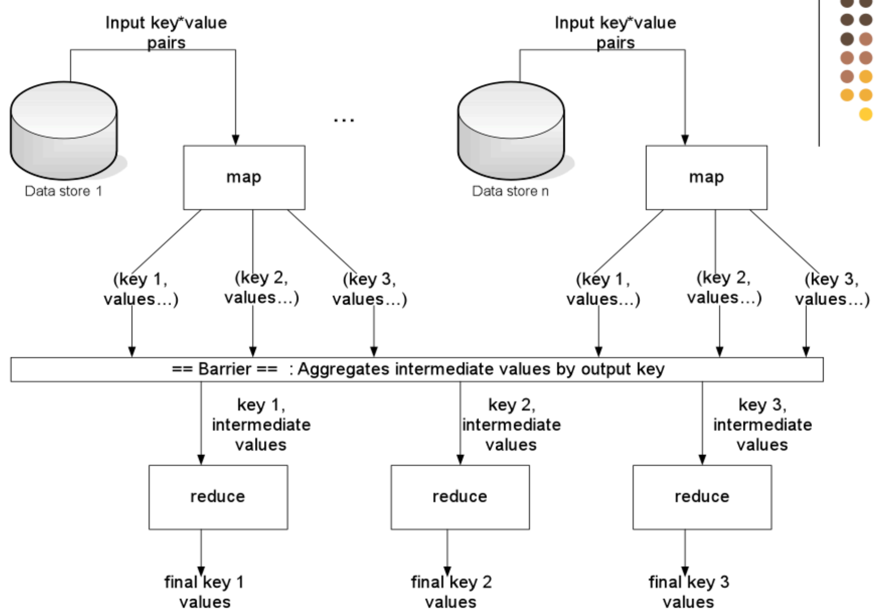
- Sum elements of a list
  - f(x,a) =
- Multiply elements of a list
  - f(x,a) =
- Compute length of a list
  - f(x,a) =

7

Examples: What is “f” operation?

- Sum elements of a list
  - f(x,a) = x + a
- Multiply elements of a list
  - f(x,a) = x * a
- Compute length of a list
  - f(x,a) = 1 + a

8



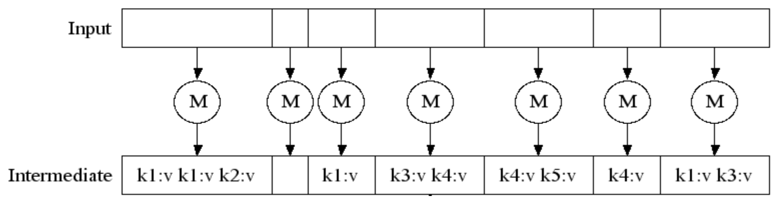
9

Map in MapReduce

- Records from the data source
  - lines out of files, rows of a database, etc
- Input as (key,value) pairs:
  - e.g., (filename, line).
- map() produces one or more *intermediate* values along with an output key

10

MapReduce: Example



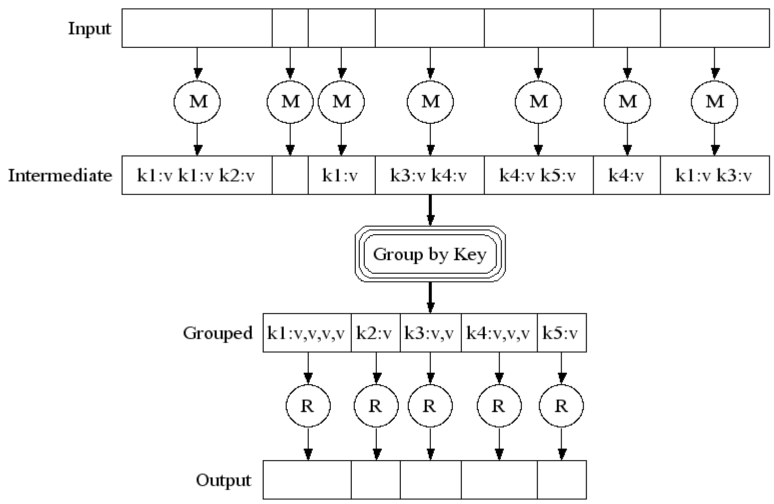
11

Reduce in MapReduce

- Intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
  - (in practice, usually only one final value per key)

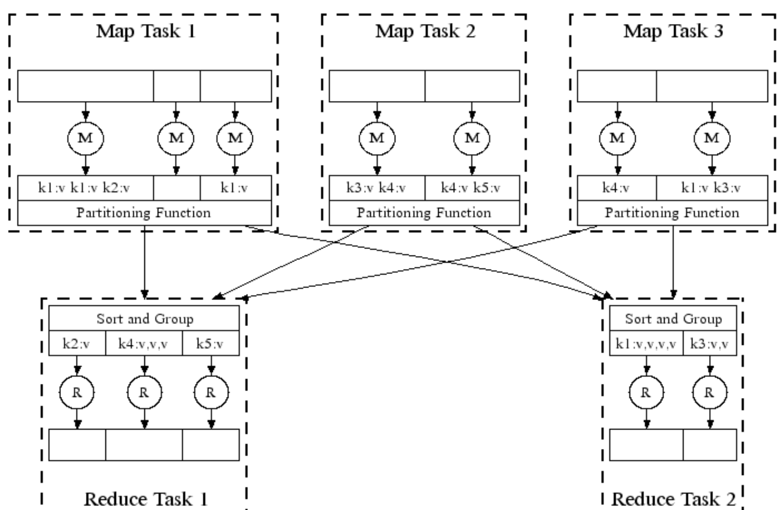
12

MapReduce: Example



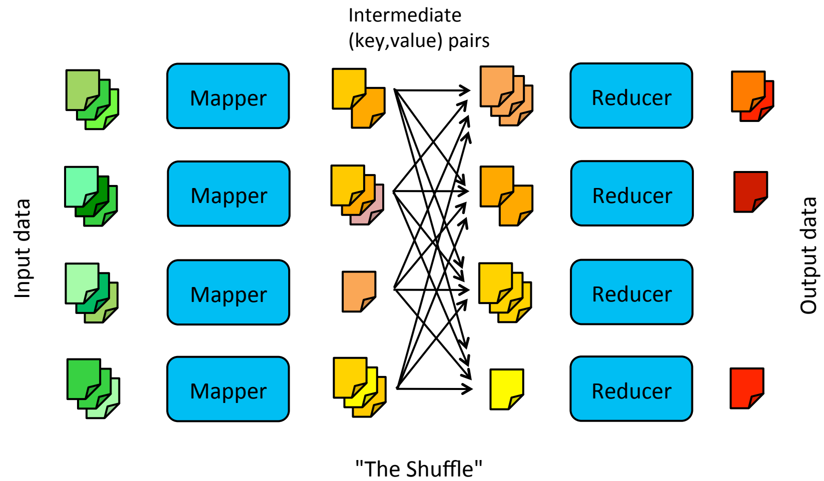
13

MapReduce in Parallel: Example



14

MapReduce Dataflow



MapReduce Programming Model

- Programmer writes 2 functions:

```
map (in_key, in_value) →
   list(out_key, intermediate_value)
```

– Processes <k,v> pairs
– Produces intermediate pairs

```
reduce (out_key, list(interim_val)) →
   list(out_value)
```

– Combines intermediate values for a key
– Produces a merged set of outputs (may be also <k,v> pairs)

Distributed Sort

- Map:
- Reduce

17

Distributed Grep

- Map:
- Reduce

18

Distributed Word Count

- Map:
- Reduce

19

Distributed Word Count

- Map:
  - Input: line of text
  - Output: (word, 1) for each word in line
- Reduce
  - Input: (word, [1,…,1])
  - Output: (word, frequency)

20

Coding MapReduce

These types depend on the input data

```
map(key:URL, value:Document)
{
  String[] words = value.split(" ");
  for each w in words
    emit(w, 1);
}

reduce(rkey:String, rvalues:Integer[])
{
  Integer result = 0;
  foreach v in rvalues
    result = result + v;
  emit(rkey, result);
}
```

Produces intermediate key-value pairs that are sent to the reducer

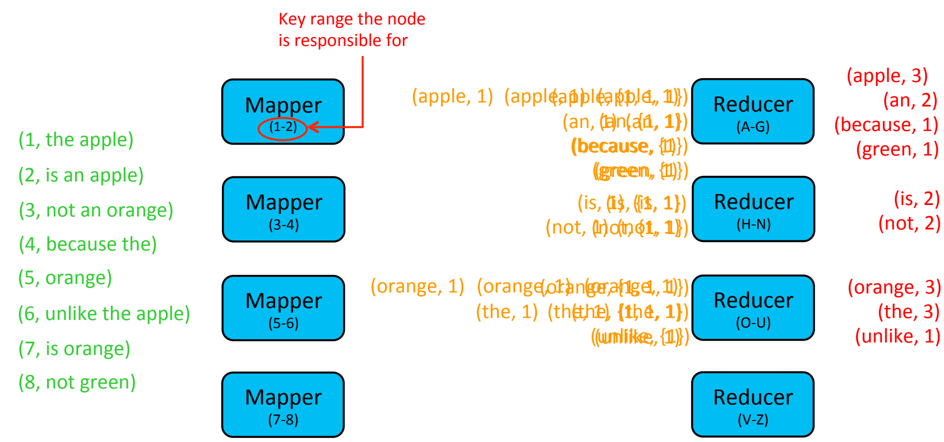
reduce gets all the intermediate values with the same rkey

These types can be (and often are) different from the ones in map()

Both map() and reduce() are stateless: Can't have a 'global variable that is preserved across invocations!

Any key-value pairs emitted by the reducer are added to the final output

Word Count (4)



1 Each mapper receives some of the KV-pairs as input

2 The mappers process the KV-pairs one by one

3 Each KV-pair output by the mapper is sent to the reducer that is responsible for it

4 The reducers sort their input by key and group it

5 The reducers process their input one group at a time

22

SINGLE-STAGE ALGORITHMS

23

Designing MapReduce algorithms

- map can only base output on each individual key-value pair
- map can emit more than one intermediate key-value pair for each incoming key-value pair
- reduce can aggregate data
  - – map must emit them using the same key!

24

Single-Stage algorithms

- Filter/collect/aggregate steps
  - Filter/collect: map
  - Collect/aggregate: reduce

- Chains of maps and reduces

Filtering algorithms

- Goal: Find lines/files/tuples with a particular characteristic

- Examples:
  - grep Web logs for requests to *.stevens.edu/*
  - find in the Web logs the hostnames accessed by 192.168.2.1
  - locate all the files that contain the words 'Apple' and 'Jobs'

- Generally: **`map`** does most of the work

Aggregation algorithms

- Goal: Compute the maximum, the sum, the average, ..., over a set of values
- Examples:
  - Count the number of requests to *.stevens.edu/*
  - Find the most popular domain
  - Average the number of requests per page per Web site
- Often: **`map`** may be simple or the identity

27

A more complex example

- Goal: Billing for a CDN like Amazon CloudFront
  - Input: Log files from the edge servers. Two files per domain:
    - access_log-www.foo.com-20111006.txt: HTTP accesses
    - ssl_access_log-www.foo.com-20111006.txt: HTTPS accesses
    - Example line:
      158.130.53.72 - - [06/Oct/2011:16:30:38 -0400] "GET /largeFile.ISO HTTP/1.1" 200 8130928734 "-" "Mozilla/5.0 (compatible; MSIE 5.01; Win2000)"
    - Mapper receives (filename,line) tuples
  - Billing policy (simplified):
    - Billing is based on a mix of request count and data traffic
    - 10,000 HTTP requests cost $0.0075
    - 10,000 HTTPS requests cost $0.0100
    - One GB of traffic costs $0.12
  - Desired output is a list of (domain, grandTotal) tuples

Intersections and joins

- Goal: Intersect multiple different inputs on some shared values
  - Values can be equal, or meet a certain predicate
- Examples:
  - Find all professors and students in common courses and return the pairs (professor,student) for those cases

29

Joining Multiple Datasets

- Join could be inner, outer, left outer, cross product etc
  - Ex: Find all professors and students in common courses and return the pairs (professor,student) for those cases
  - Based on a common *join key*
- Join is a natural Reduce operation

30

Map in Join

- Input: *(Key_1,$Value_1$)* from A or B
- Output: *(Key_2, (Value2, A|B))*
  - *Key_2* is the *Join Key*

31

Reduce in Join

- Input: Lists of *($Value_2$, A|B)*
  - for each join key *Key_2*
- Operation depends on which kind of join
  - Inner join checks if key has values from both A & B
- Output: *(Key_2, JoinFunction($Value_2$,...))*

32

MR Join Performance

- Map Input =Total of A & B
- Map output = Total of A & B
- Shuffle & Sort
- Reduce input = Total of A & B
- Reduce output = Size of Joined dataset
- Filter and Project in Map

Join Special Cases

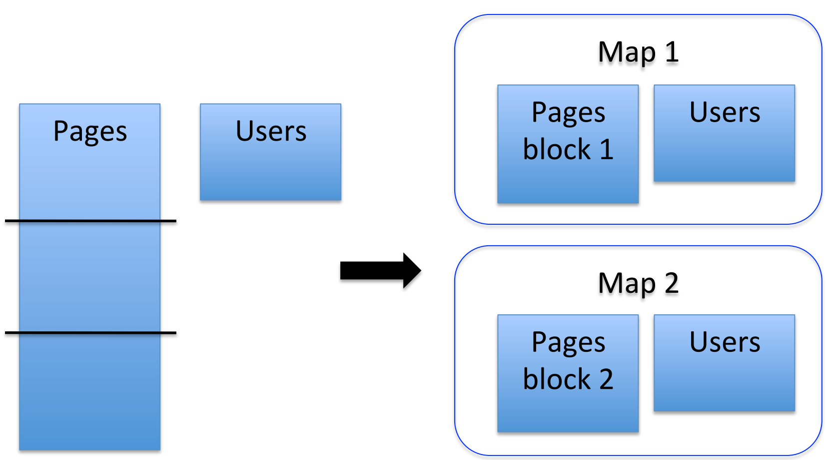
- Fragment-Replicate
  – 100GB dataset with 100 MB dataset
- Equipartitioned Datasets
  – Identically Keyed
  – Equal Number of partitions
  – Each partition locally sorted

Fragment-Replicate Join

- Fragment larger dataset
  - Specify as Map input
- Replicate smaller dataset
  - Use Distributed Cache
- Map-Only computation
  - No shuffle / sort

35

Fragment Replicate Join



36

Equipartitioned Join

- Datasets joined “before” input to mappers
- Input format: *CompositeInputFormat*
- *mapred.join.expr*

37

Equipartitioned Join

```
mapred.join.expr =
   inner (
      tbl (
         ....SequenceFileInputFormat.class,
         "hdfs://namenode:8020/path/to/data/A"
   ),
      tbl (
         ....SequenceFileInputFormat.class,
         "hdfs://namenode:8020/path/to/data/B"
   )
)
```

38

Partial Cartesian products

- Goal: Find some complex relationship, e.g., based on pairwise distance
- Examples:
  – Find all pairs of sites within 100m of each other
- Generally hard to parallelize
  – Divide the input into bins or tiles?
  – Link it to some sort of landmark?
  – Overlap the tiles?
  – Generate landmarks using clustering?

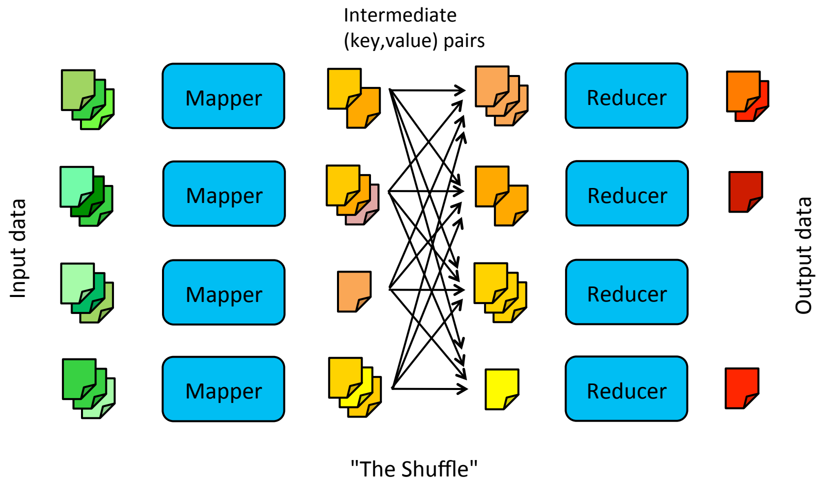
39

Sorting

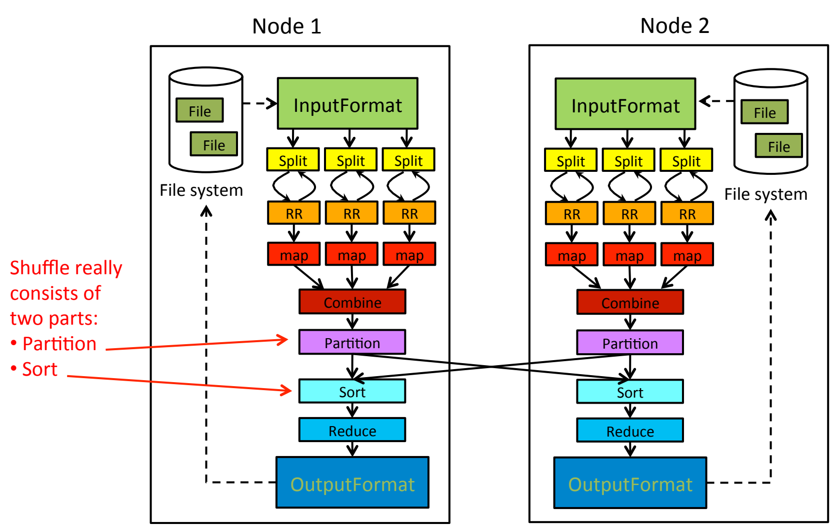
- Goal: Sort input
- Examples:
  – Return all the domains covered by Google's index and the number of pages in each, ordered by the number of pages
- Not supported by programming model…
  – …but supported by implementation

40

MapReduce Dataflow



Shuffle in More Detail



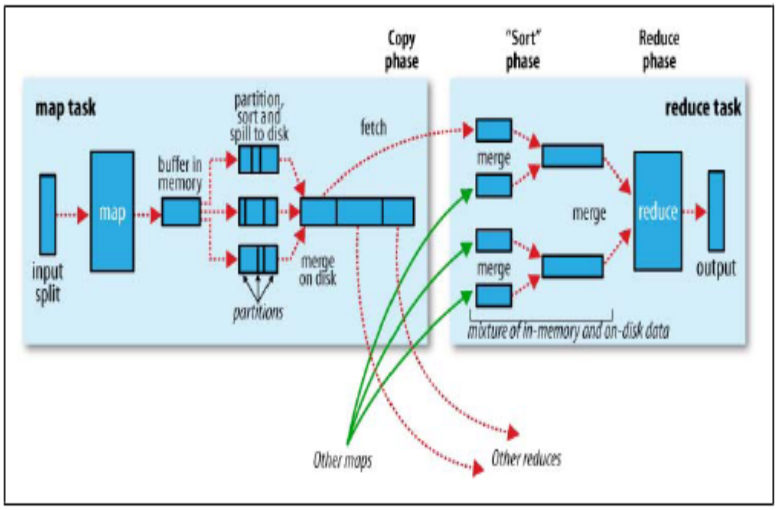


Figure 6-4. Shuffle and sort in MapReduce

43

Shuffle as a sorting mechanism

- Single reducer
- Multiple reducers: partly sorted output
  - Write a last-pass file that merges all of the part-r-000x files
  - Partition intermediate values so that *all keys in partition i < all keys in partition j* for i < j (TeraSort)

44

MULTI-STAGE ALGORITHMS

45

Example: Unigrams

- Input: Huge text corpus
- Wikipedia Articles (40GB uncompressed)
- Output: List of words sorted in descending order of frequency

46

Unigrams

```
$ cat ~/wikipedia.txt | \
sed -e 's/ /\n/g' | grep . | \
sort | \
uniq -c > \
~/frequencies.txt
```

Stage I

```
$ cat ~/frequencies.txt | \
# cat | \
sort -n -k1,1 -r |
# cat > \
~/unigrams.txt
```

Stage II

47

MR for Unigrams: Stage I

```
mapper (filename, file-contents):
  for each word in file-contents:
    emit (word, 1)

reducer (word, values):
  sum = 0
  for each value in values:
    sum = sum + value
  emit (word, sum)
```

48

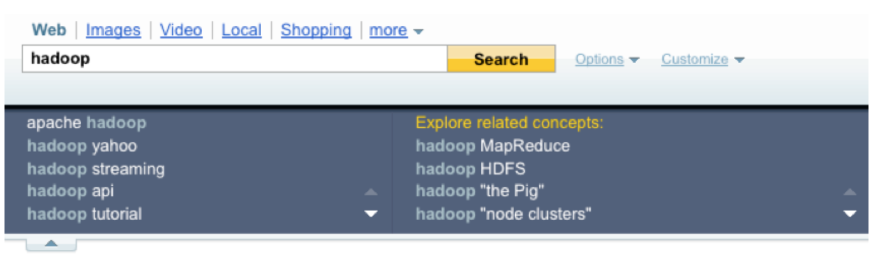
MR for Unigrams: Stage II

```
mapper (word, frequency):
  emit (frequency, word)

reducer (frequency, words):
  for each word in words:
    emit (word, frequency)
```

49

Bigrams



50

Bigrams

- Input: A large text corpus
- Output: List($word_1$,Top_K($word_2$))
- Two Stages:
  - Generate all possible bigrams
  - Find most frequent K bigrams for each word

51

Bigrams: Stage I Map

- Generate all possible Bigrams
- Map Input: Large text corpus
- Map computation
  - In each sentence, or each “word1 word2”
  - Output (word1, word2), (word2, word1)
- Partition & Sort by (word1, word2)

52

Bigrams: Stage I Reduce

- Input: List of (word1, word2) sorted and partitioned
- Output: List of (word1, [(freq, word2), …])
- Counting similar to Unigrams example

Bigrams: Stage II Map

- Input: List of (word1, [(freq,word2), …])
- Output: List of (word1, [(freq, word2), …])
- Identity Mapper (/bin/cat)
- Partition by word1
- Sort descending by (word1, freq)

Bigrams: Stage II Reduce

- Input: List of (word1, [(freq,word2), …])
  - partitioned by word1
  - sorted descending by (word1, freq)
- Output: Top_K(List of (word1, [(freq, word2), …]))
- For each word, throw away after K records

55

COMMON MISTAKES

56

Common Mistakes to Avoid

- Mapper and reducer should be stateless
  - Don't use static variables

```
HashMap h = new HashMap();
map(key, value) {
  if (h.contains(key)) {
    h.add(key,value);
    emit(key, "X");
  }
}
```

Wrong!

- Don't try to do your own I/O!
  - MapReduce uses HDFS

```
map(key, value) {
  File foo =
    new File("xyz.txt");
  while (true) {
    s = foo.readLine();
    ...
  }
}
```

Wrong!

Common Mistakes to Avoid

```
map(key, value) {
  emit("FOO", key + " " + value);
}
```

Wrong!

```
reduce(key, value[]) {
  /* do some computation on
  all the values */
}
```

- Mapper must not map too much data to the same key

Locality Optimization

- Problem: Network bandwidth
- Leverage GFS:
  - Schedule map task on machine that contains its input
  - Thousands of machines read input at local disk speed
- Without this, rack switches limit read rate

59

Redundant Execution

- Problem: Slow workers
- Near end of phase, spawn backup tasks
  - Increase utilization
  - Reduce completion time

60

Skipping Bad Records

- Problem: functions sometimes fail for particular inputs
- Fixing the bug might not be possible : Third Party Libraries
- On Error
  - Worker sends signal to Master
  - If multiple error on same record, skip record

61

Conclusion

- MapReduce:
  - Parallel functional programming
  - Batch processing of Big Data
- Many implementations e.g. Hadoop
- Base for cloud computing stack
  - Pig, Hive
  - NoSQL query processing

62