MapReduce

ECE4150 Cloud Computing

Joel Corporan, Ph.D.

This week...

MapReduce & AWS EMR

- Today (03/25):
  - MapReduce Summary
  - MR Application Examples
- Wednesday (03/27):
  - Lab 5: Batch Data Analysis using Hadoop, MapReduce, Pig & Hive with AWS EMR

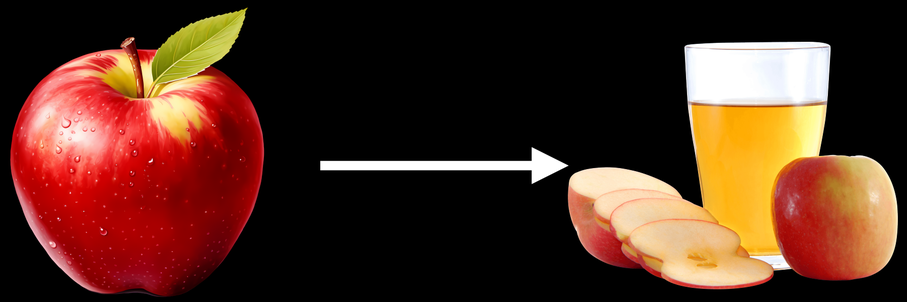
MapReduce
The Story Sam

One day, mom told Sam...

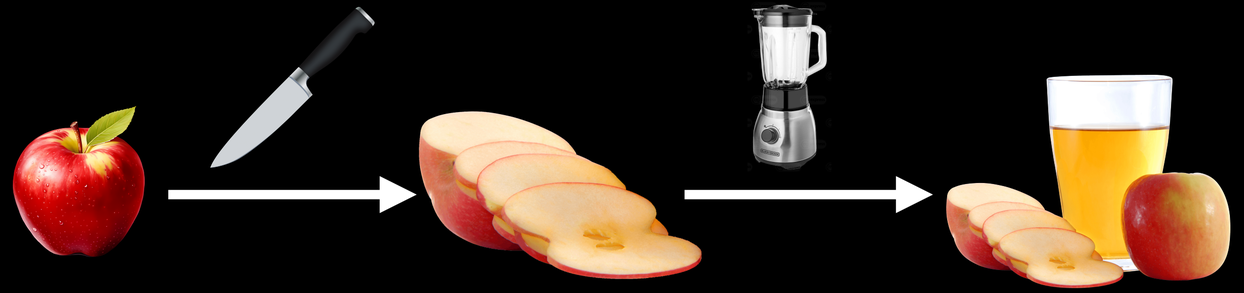
An apple a day keeps a doctor away!

But one day...

Sam thought of “drinking” the apple instead

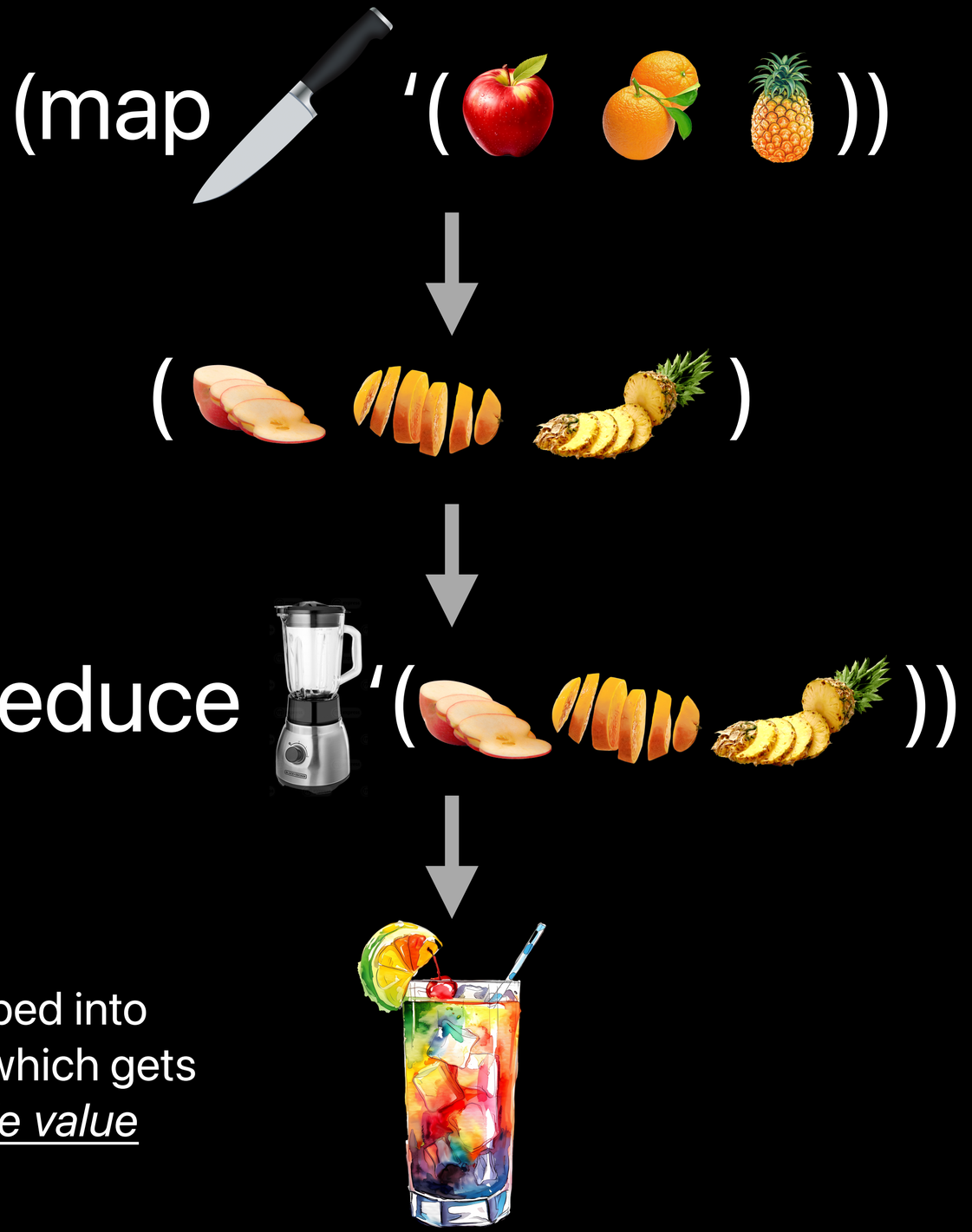


So, he used a  to cut the  and a  to make 



The next day...

Sam applied his invention to all the fruits he could find.



A *list of values* mapped into another *list of values*, which gets reduced into a *single value*

MapReduce
Programming Model

- An abstraction to express simple computations
  - From a set of input key-value pairs, its generates a set of output key-value pairs
    - $\langle k_i, v_i \rangle \rightarrow \langle k_o, v_o \rangle$
  - Expressed with two functions (or abstraction)
    - The Map task processes <k, v> pairs and produces an intermediate pairs
      - $\langle k_i, v_i \rangle \rightarrow \{\langle k_{int}, v_{int} \rangle\}$
    - The Reduce task combines intermediate values for a key and produces a merged set of outputs (may be also <k, v> pairs)
      - $\langle k_{int}, \{v_{int}\} \rangle \rightarrow \langle k_o, v_o \rangle$

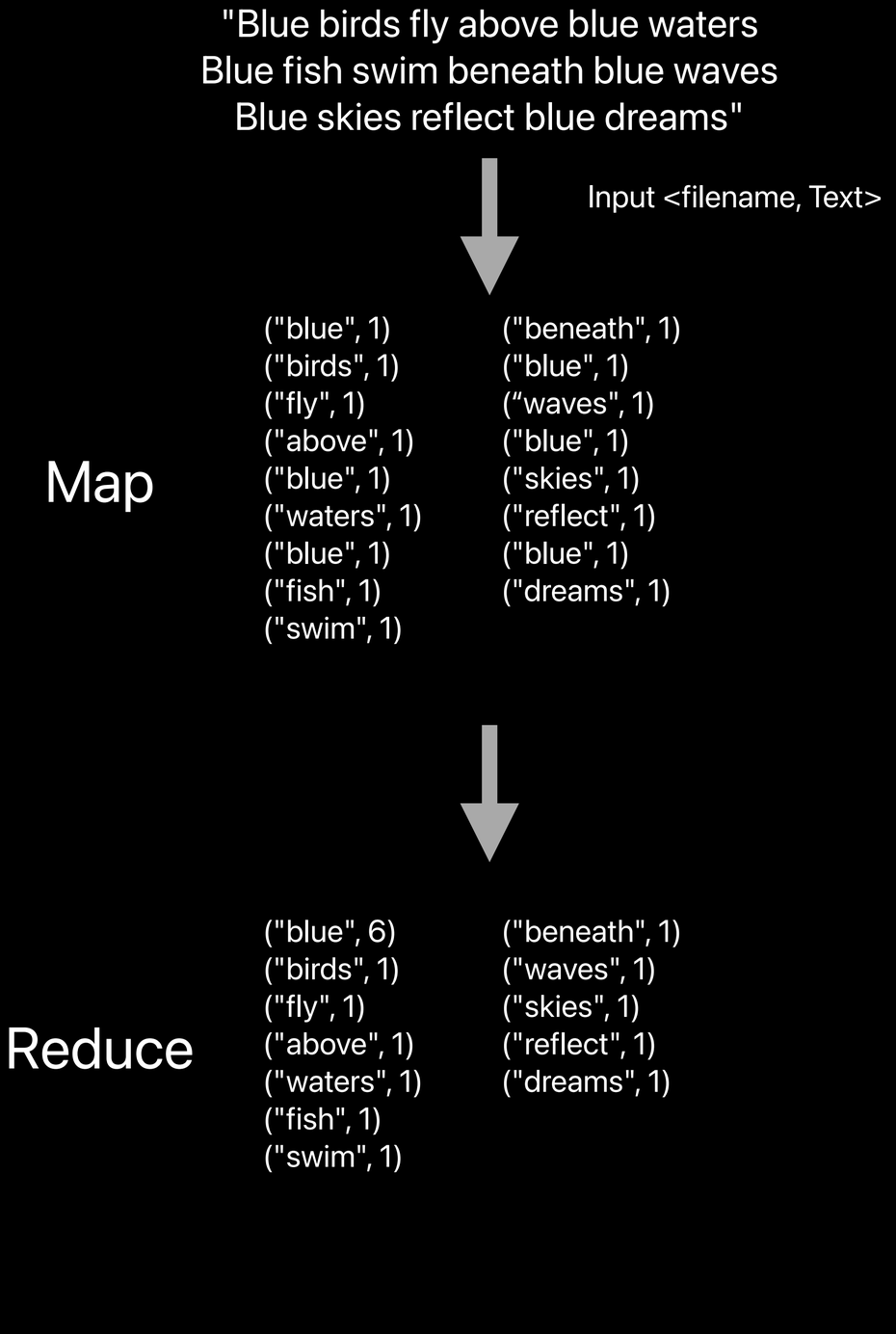
Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

MapReduce

Workcount Example

```
function map(filename, Text):
    for each word in Text:
       word = word.lower()
        emit(word, 1)

function reduce(Word, Counts):
    sum = 0 // Initialize a sum for each word
    for each count in Counts:
        sum += count
    emit(Word, sum)
```



18 years later...

Sam got his first job with a juice making giant, called Joogle, for his talent in making juice

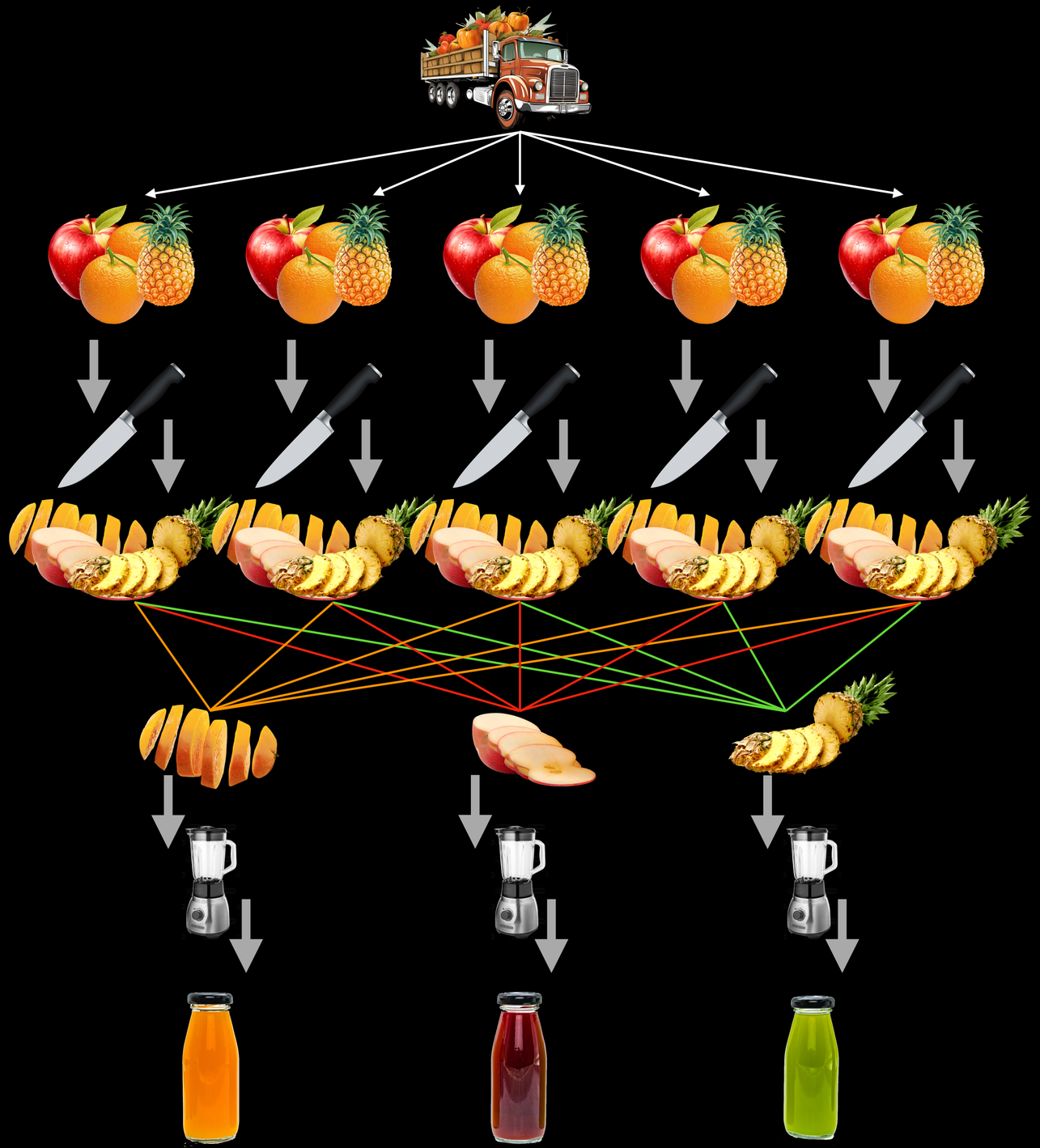
But...

- Now, it is not just a couple a fruits, but a whole *container* of fruits
- Also, they product a *list* of juice type separately
- Sam had just one  and one 

Large data and list of values for output

After much thought...

Sam implemented a *parallel* version of his innovation.



MapReduce

Commodity clusters

- MapReduce is designed to efficiently process large volumes of data by connecting many commodity computers to work in parallel
- A *theoretical* 1000-CPU machine would cost a very large amount of money, far more than 1000 single-CPU or 250 quad-core machines
- MapReduce ties smaller and more reasonably priced machines together into a single cost-effective *commodity cluster*

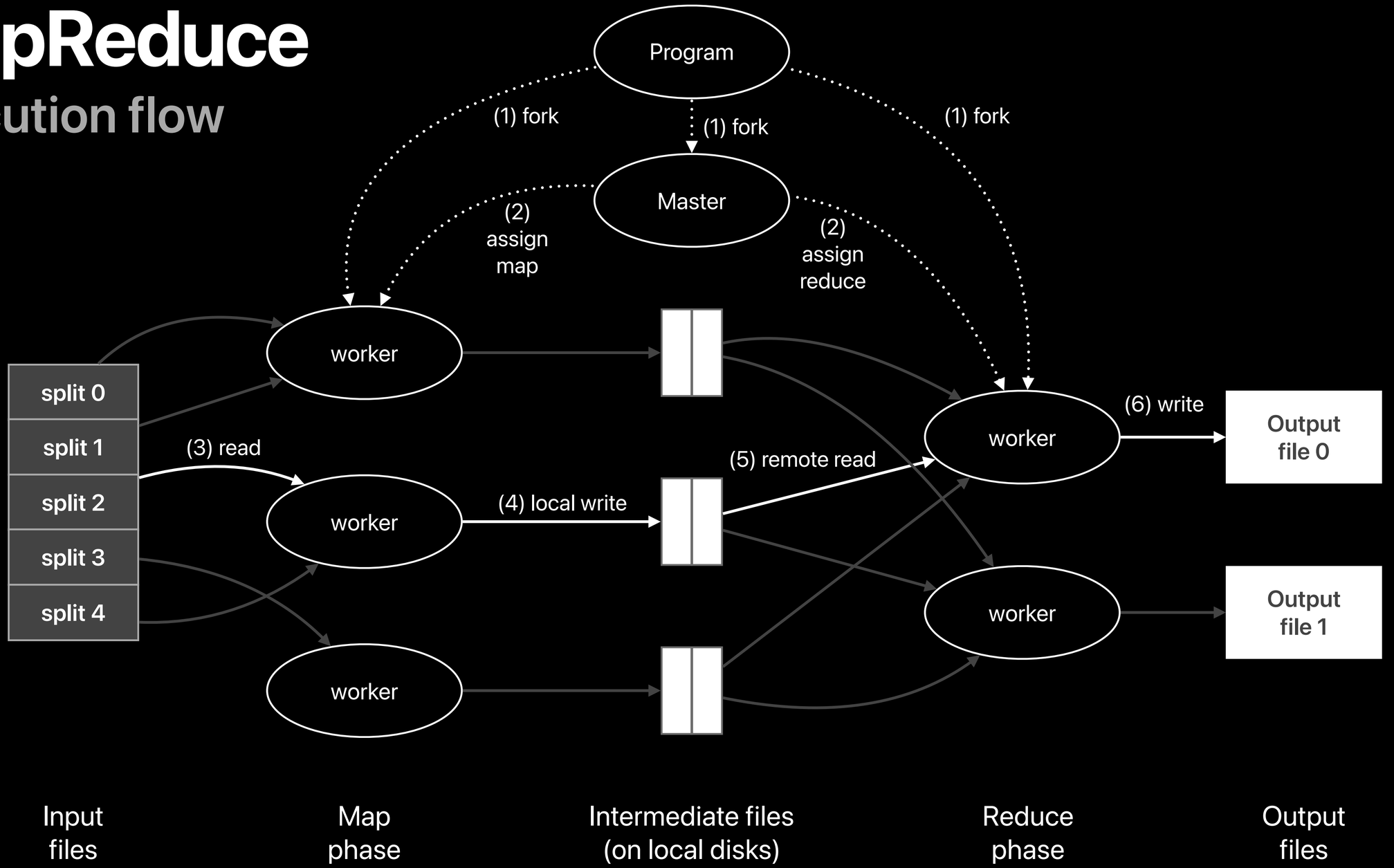
MapReduce

Isolated tasks

- MapReduce divides the workload into multiple *independent tasks* and schedules them across cluster nodes
- The work performed by each task is done *in isolation* from one another
- The amount of communication that tasks can perform is mainly limited for scalability reasons

MapReduce

Execution flow



Each input to a map is a **list of <key, value> pairs**

(e.g., <a, >, <o, >, <p, >, …)

Each output of the map is a **list of <key, value> pairs**

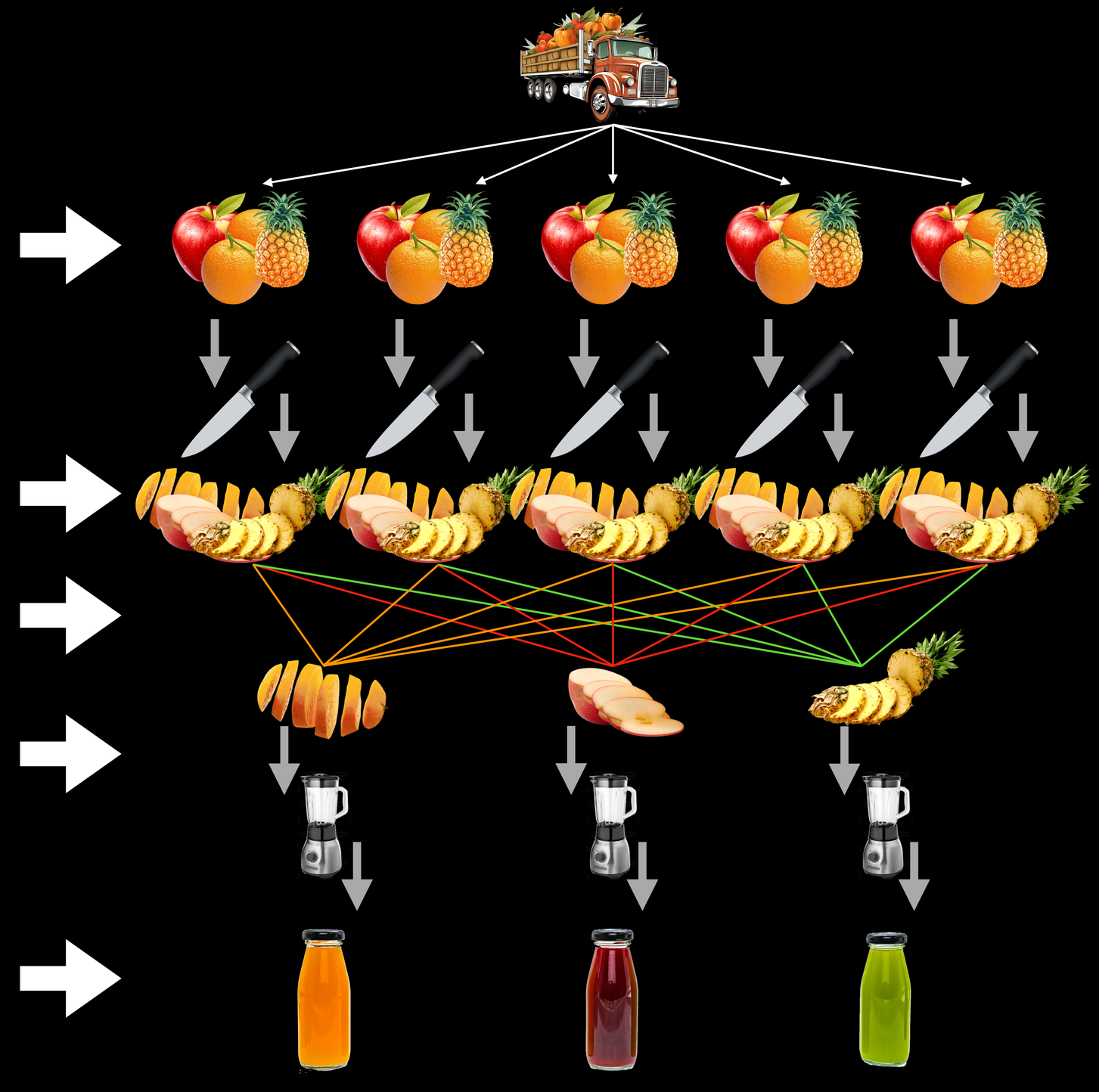
(e.g., <a', >, <o', >, <p, >, …)

Grouped by key

Each input to a reduce is a <key, value-list>

(e.g., <a', …)>

Reduced to a **list of values**



MapReduce

Applications

MapReduce Applications

Distributed Grep

- **Input:** large set of files
- **Output:** lines that match pattern
- **Importance:**
  - **Distributed Log Analysis**: Important for searching through massive logs spread across many servers to find errors or specific events.
  - **Data Mining**: Useful for quickly filtering large datasets and finding particular patterns or keyword occurrences.

MapReduce Applications

Distributed Grep — Input

Suppose we are searching for the pattern “ERROR” in a set of log files with the following lines:

INFO Starting process

ERROR Failed to open database

INFO Process completed

ERROR Unable to retrieve data

DEBUG Memory allocation successful

MapReduce Applications

Distributed Grep — Map phase

Map phase emits a line if it matches the supplied pattern. As a result, the output should be:

- *ERROR Failed to open database*
- *ERROR Unable to retrieve data*

MapReduce Applications

Distributed Grep — Reduce phase

Reduce phase copies the intermediate data to output. Therefore the out should be:

- *ERROR Failed to open database*
- *ERROR Unable to retrieve data*

MapReduce Applications

Distributed Grep — Pseudocode

```
function map(file_name, file_contents):
    for line in file_contents:
        if "ERROR" in line:
            emit(line)

function reduce(_, lines):
    for line in lines:
        output(line)
```

MapReduce Applications

Reverse Web-Link Graph

- **Input:** Web graph: tuples (a, b) where (page a → page b)
- **Output:** For each page, list of pages that link *to* it
- **Importance:**
  - **Backlink Analysis:** SEO evaluation of the volume and *quality* of links pointing to a particular website in another domain (e.g., huffpost.com → corporans.com)
  - **Fraud Detection:** Useful for detecting unnatural linking patterns that may suggest fraudulent activity.

MapReduce Applications

Reverse Web-Link Graph — Input

Suppose we have the following tuples representing links between pages (source → target):

- (PageA → PageB)
- (PageB → PageC)
- (PageA → PageC)
- (PageC → PageD)
- (PageB → PageD)
- (PageD → PageA)

MapReduce Applications

Reverse Web-Link Graph — Map phase

Map phase will process web log and for each input ***<source, target>***, it outputs ***<target, source>***. As a result, the output should be:

- *<PageB, PageA>*
- *<PageC, PageB>*
- *<PageC, PageA>*
- *<PageD, PageC>*
- *<PageD, PageB>*
- *<PageA, PageD>*

MapReduce Applications

Reverse Web-Link Graph — Reduce phase

Reduce phase emits <target, list(source)>. Therefore the out should be:

- *<PageA, [PageD]>*
- *<PageB, [PageA]>*
- *<PageC, [PageB, PageA]>*
- *<PageD, [PageC, PageB]>*

MapReduce Applications

Reverse Web-Link Graph — Pseudocode

```
function map(input_key, input_value):
    // input_key is source page
    // input_value is target page
    emit(input_value, input_key)

function reduce(target_page, sources):
    source_list = list(sources)
    emit(target_page, source_list)
```

MapReduce Applications

Count of URL access frequency

- **Input:** Log of accessed URLs (e.g., proxy server)
- **Output:** For each URL, % of total accesses for that URL
- **Importance:**
  - **Load Balancing**: It can help load balancing across servers by understanding the frequency of URL access.
  - **Caching Decisions**: Frequently accessed URLs can be cached more aggressively.
  - **Resource Allocation**: Servers and bandwidth can be allocated more efficiently based on URL access patterns.

MapReduce Applications

Count of URL access frequency — Input

Suppose we a log file of accessed page in a particular proxy server:

- http://example.com/page1 - accessed
- http://example.com/page2 - accessed
- http://example.com/page1 - accessed
- http://example.com/page3 - accessed
- http://example.com/page2 - accessed
- http://example.com/page2 - accessed

MapReduce Applications

Count of URL access frequency — Map phase

Map phase will parse each log entry to extract the URL and emit a key-value pair with the URL as the key and the number 1 as the value (e.g., ***<URL, 1>***). As a result, the output should be:

- *<http://example.com/page1, 1>*
- *<http://example.com/page2, 1>*
- *<http://example.com/page1, 1>*
- *<http://example.com/page3, 1>*
- *<http://example.com/page2, 1>*
- *<http://example.com/page2, 1>*

MapReduce Applications

Count of URL access frequency — Reduce phase

Reduce phase sums up all the 1's for each URL to get the total count of accesses for that URL. Therefore, the output should be:

- *<http://example.com/page1, 2>*
- *<http://example.com/page2, 3>*
- *<http://example.com/page3, 1>*

MapReduce Applications

Count of URL access frequency — (2) Map phase

(2) Map phase emits a dummy key (e.g., 1) and the key-value pair from the first Reduce phase. As a result, the output should be:

- *<1, (<http://example.com/page1, 2>)>*
- *<1, (<http://example.com/page2, 3>)>*
- *<1, (<http://example.com/page3, 1>)>*

MapReduce Applications

Count of URL access frequency — (2) Reduce phase

(2) Reduce phase does two passes. In first pass, sums up all *URL_count's* to calculate overall_count. In second pass calculates %'s. Therefore, the output should be:

- **First Pass:** it calculates the overall count
  - overall_count = 2 + 3 + 1 = 6
- **Second Pass:** it calculates the percentage
  - <http://example.com/page1, (2/6) * 100> → <http://example.com/page1, 33.33%>
  - <http://example.com/page2, (3/6) * 100> → <http://example.com/page2, 50.00%>
  - <http://example.com/page3, (1/6) * 100> → <http://example.com/page3, 16.67%>

MapReduce Applications

Count of URL access frequency — Pseudocode

```
// First MapReduce job
Map1(input_key, input_value):
  // input_value is each line from web log
  URL = extract_url(input_value)
  emit(URL, 1)

Reduce1(URL, values):
  URL_count = sum(values)
  emit(URL, URL_count)
```

MapReduce Applications

Count of URL access frequency — Pseudocode *(continuation)*

```
// Second MapReduce job
Map2(input_key, input_value):
  // input_key is the URL
  // input_value is the URL_count
  emit(1, (input_key, input_value))

Reduce2(dummy_key, values):
  overall_count = 0
  for value in values:
    overall_count += value.URL_count

  for value in values:
    percentage = (value.URL_count / overall_count) * 100
    emit(value.URL, percentage)
```