NYU Center for Data Science

Week 03.3: Map-Reduce

DS-GA 1004: Big Data

Announcements

- Quiz #1 on SQL/RDBMS

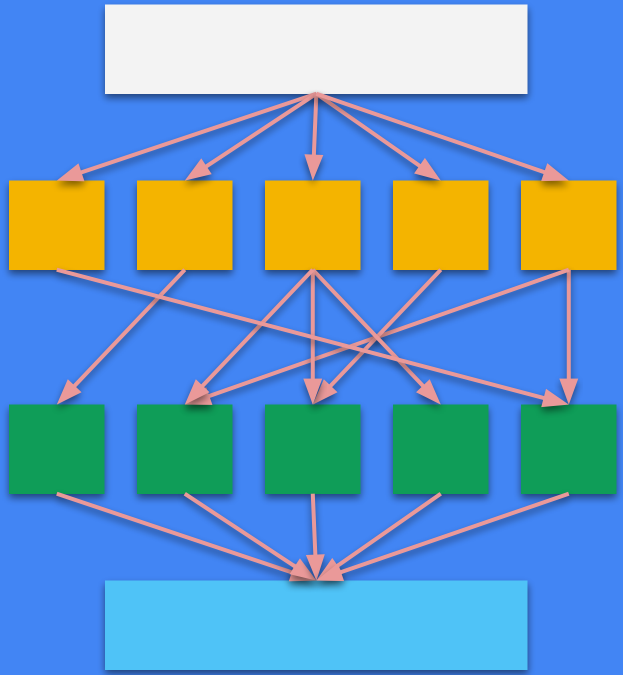
  Friday (02/11)

  - You have 24 hours to start the quiz
  - You have 1 hour to complete it
  - Open-book/paper/note
  - **You may not discuss with classmates / other people!**

Previously...

- Relational data model + DBMS
- Transactions for concurrent access

This week



1. Introduction to Map-Reduce (Dean & Ghemawat, 2008)
2. Criticisms of Map-Reduce (DeWitt & Stonebraker, 2008)

Why map-reduce?

- Distributed programming is really **difficult**!
- If we **restrict** how we program, parallelism becomes easier
- The **map** and **reduce** operations are surprisingly powerful!

Map-Reduce flow

1. Map phase
   a. Distribute data to mappers
   b. Generate intermediate results (*key*, *value*)

2. Sort / shuffle phase
   a. Assign intermediate results to reducers (by *key*)
   b. Move data from mappers to reducers

3. Reduce phase
   a. Execute reducers and collect output

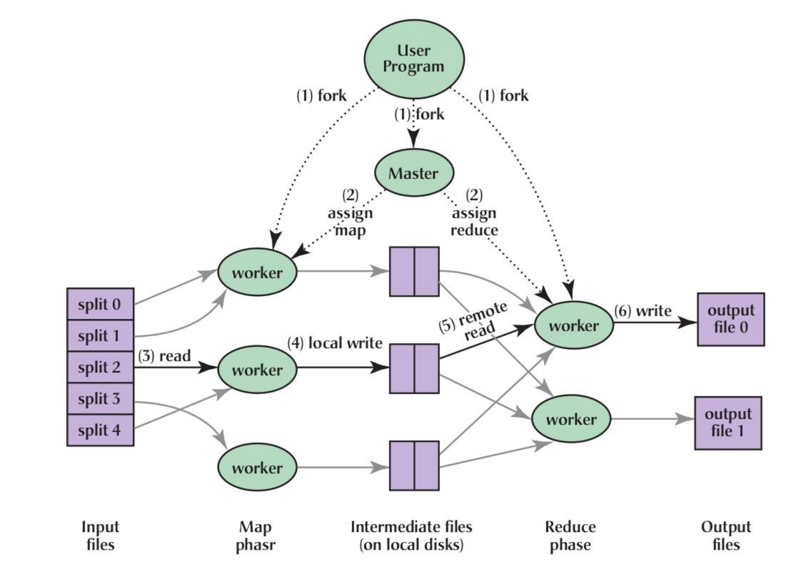


Figure adapted from Dean & Ghemawat, 2008

Why “map” and “reduce”?

- These are common operations in **functional programming**
  - E.g.: LISP, ML, Haskell, Scala…

- **map**(function f, values $[x_1, x_2, \ldots, x_n]$) $\rightarrow [f(x_1), f(x_2), \ldots f(x_n)]$
  - **map** : function, list $\rightarrow$ list

- **reduce**(function g, values $[x_1, x_2, \ldots, x_n]$) $\rightarrow g(x_1,$ **reduce**$(g, [x_2, \ldots, x_n]))$
  - **reduce** : function, list $\rightarrow$ item

Ex.: word counting in a document collection

```
mapper(doc_id, doc_contents):
  for word in doc_contents:
    emit word, 1
```

```
reducer(word, counts):
  total_count = 0
  for count in counts:
    total_count += count
  emit total_count
```

Key idea:

- Make the mapper as simple as possible
- Let the MR framework route the intermediate results
- Reduce can be simple as well

Combiner example: word count

```
mapper(doc_id, doc_contents):
  for word in doc_contents:
    emit word, 1
```

```
combiner(word, counts):
  partial_count = 0
  for count in counts:
    partial_count += count
  emit word, partial_count
```

Mapper node

```
reducer(word, counts):
  total_count = 0
  for count in counts:
    total_count += count
  emit total_count
```

This works because summation is **commutative** and **associative:**

A + B = B + A

A + B + C = (A + B) + C

When that happens, you can re-use the **reducer** code as a **combiner**!

Group exercise 1

Degree counting

- Input consists of weighted edges in an undirected graph, e.g.:

```
# u, v, w
3, 5, 0.5
2, 3, 10
3, 1, 6.1
...
```

- Write a **mapper** and **reducer** to compute the **degree** (sum of edge weights) for each vertex
  - $\mathrm{deg}(u) = \sum_{u,v} w(u, v) + \sum_{v,u} w(u, v)$

slido

Write your mapper and reducer code here (one submission per group)

ⓘ Start presenting to display the poll results on this slide.

Solution: Degree counting

```
def mapper(u, v, w):
    emit u, w
    emit v, w
```

```
def reducer(u, weights):
    emit sum(weights)
```

Each edge touches two vertices, and contributes to both of their degree totals.

⇒ Mapper produces two intermediate keys per edge

Some tips...

- **Don't use floating point** / real numbers for **keys**
  - Keys need to hash consistently, and floating point equivalence can be subtle!
- Keep **map** and **reduce** simple!
  - Avoid loops if possible
  - Let sort do the work for you
- **Compare** your algorithm's complexity to the **simple / single-core solution**

Degree-counting, simple algorithm

```
def degree_count(V, E):
    deg ← defaultdict(0)
    for (u, v, w) in E:
        counts[u] += w
        counts[v] += w
    return deg
```

- Time is O(|E|)
- Space is O(|V|)

Degree-counting, MR algorithm

```
def mapper(u, v, w):
    emit u, w
    emit v, w

def reducer(u, weights):
    emit sum(weights)
```

- **Mapper**
  - Each call is O(1) time/space
  - Time O(|E| / K) for K mappers
  - **Total space: O(|E|)**
- **Shuffle**: O(|V|) keys, O(|E|) values
- **Reducer**
  - Each call is O(max-degree)
  - Total time depends on skew
  - Total space is O(|V|)

Group exercise 2

Directed degree counting

- Input consists of weighted edges in an **directed** graph, e.g.:

```
# u, v, w;  u → v
3, 5, 0.5
2, 3, 10
3, 1, 6.1
...
```

- Write a **mapper** and **reducer** to compute the **in-degree** and **out-degree** for each vertex
  - in-deg(u) = $\sum_{v,u} w(v, u)$
  - out-deg(u) = $\sum_{u,v} w(u, v)$

slido

Write your map-reduce code for directed degree counting here.

ⓘ Start presenting to display the poll results on this slide.

Solution: Directed degree counting

```
def mapper(u, v, w):
    emit (u, 'out'), w
    emit (v, 'in'), w
```

```
def reducer(key, weights):
    # u, direction ← key
    emit sum(weights)
```

The logic is nearly the same as in problem 1

We can pack information into the **intermediate keys** to keep things simple!

Sort/shuffle phase does the hard work for us.

5 criticisms

(DeWitt & Stonebraker, 2008)

1. Too low-level
   - Schemas, high-level languages
2. Poor implementation
   - No indexing, “brute-force” thinking
3. Not novel
4. Missing important features
   - Schemas, transactions, etc
5. No DBMS compatibility

Which of the five criticisms do you think are valid?

ⓘ Start presenting to display the poll results on this slide.

slido

[2] Which of the five criticisms do you think are valid?

ⓘ Start presenting to display the poll results on this slide.

MR is not an RDBMS…

- … but there is some overlap!

- Simple queries can often be translated to MR
    - SELECT some_id, sum(some_feature) from my_table
      WHERE some_clause ← mapper → (id, feature)
      GROUP BY some_id ← reducer → (id, sum(feature))

- Complex queries can be very difficult to translate
    - Multi-way joins are especially difficult, and common in practice

MR is not a general computation engine…

- … but it can still do a lot!
- Algorithms with **clear parallelism** and no/little looping are okay
- Iterative or recursive algorithms … not so much
  - Gradient descent (and other ML approaches, e.g. alternating solvers)

No transactions?

- Transactions exist to maintain consistency and validity of data
- Map-Reduce avoids these issues in a couple of ways:
  - Data is immutable
  - **MR programs must be deterministic**
- If a compute node fails, we can spawn extra instances of mappers / reducers, and we're guaranteed the same outputs
  - As soon as one node finishes, we can ignore any redundant copies

Next week

- The Hadoop distributed file-system (HDFS)
- Lab 2: programming with map-reduce

Audience Q&A Session

ⓘ Start presenting to display the audience questions on this slide.