MAP REDUCE SOME PRINCIPLES AND PATTERNS: IMPLEMENTING OPERATORS

GENOVEVA VARGAS SOLAR

FRENCH COUNCIL OF SCIENTIFIC RESEARCH, LIG-LAFMIA, FRANCE

Genoveva.Vargas@imag.fr

http://www.vargas-solar.com/teaching

MAP-REDUCE

- **Programming model** for expressing distributed computations on massive amounts of data
- **Execution framework** for large-scale data processing on clusters of commodity servers
- Market: any organization built around gathering, analyzing, monitoring, filtering, searching, or organizing content must tackle large-data problems
  - data- intensive processing is beyond the capability of any individual machine and requires clusters
  - large-data problems are fundamentally about ***organizing computations on dozens, hundreds, or even thousands of machines***

« Data represent the rising tide that lifts all boats—more data lead to better algorithms and systems for solving real-world problems »

DATA PROCESSING

- Process the data to produce other data: analysis tool, business intelligence tool, ...
- This means
- • Handle large volumes of data
- • Manage thousands of processors
- • Parallelize and distribute treatments
  - Scheduling I/O
  - Managing Fault Tolerance
  - Monitor /Control processes

Map-Reduce provides all this easy!

COUNTING WORDS

(URI, document) → (term, count)

```
see bob throw
see spot run
```

Map

```
see     1
bob     1
throw   1
see     1
spot    1
run     1
```

Shuffle/Sort

```
bob     <1>
run     <1>
see     <1,1>
spot    <1>
throw   <1>
```

Reduce

```
bob     1
run     1
see     2
spot    1
throw   1
```

MAP REDUCE EXAMPLE

- Input key-values pairs take the form of (docid, doc) pairs stored on the distributed file system,
  - the former is a unique identifier for the document
  - the latter is the text of the document itself
- The **mapper** takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for every word:
  - the word itself serves as the key, and the integer one serves as the value (denoting that we've seen the word once)
  - the MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer
- The reducer sums up all counts (ones) associated with each word
  - emits final key- value pairs with the word as the key, and the count as the value.
  - output is written to the distributed file system, one file per reducer

DESIGNING MAP REDUCE ALGORITHMS

PATTERNS AND EXAMPLES

BEYOND THE CONTROL OF PROGRAMMERS

- Where a mapper or reducer runs (i.e., on which node in the cluster)
- When a mapper or reducer begins or finishes
- Which input key-value pairs are processed by a specific mapper
- Which intermediate key-value pairs are processed by a specific reducer

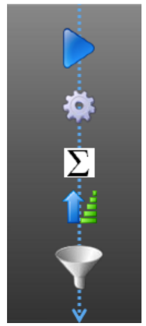
UNDER THE CONTROL OF PROGRAMMERS

- The ability to construct complex data structures as keys and values to store and communicate partial results.
- The ability to execute user-specified initialization code at the beginning of a map or reduce task, and the ability to execute user-specified termination code at the end of a map or reduce task.
- The ability to preserve state in both mappers and reducers across multiple input or intermediate keys.
- The ability to control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys.
- The ability to control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer.

MAP-REDUCE PHASES

- Initialisation
- **Map**: *record reader, mapper, combiner, and partitioner*
- **Reduce:** *shuffle, sort, reducer, and output format*

- Partition input (key, value) pairs into chunks run map() tasks in parallel
- After all map()'s have been completed consolidate the values for each unique emitted key
- Partition space of output map keys, and run reduce() in parallel



MAP SUB-PHASES

- *Record reader* translates an input split generated by input format into records
  - parse the data into records, but not parse the record itself
  - It passes the data to the mapper in the form of a key/value pair. Usually the key in this context is positional information and the value is the chunk of data that composes a record
- *Map* user-provided code is executed on each key/value pair from the record reader to produce zero or more new key/value pairs, called the intermediate pairs
  - The key is what the data will be grouped on and the value is the information pertinent to the analysis in the reducer
- *Combiner*, an optional localized reducer
  - Can group data in the map phase
  - It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in the small scope of that one mapper
- *Partitioner* takes the intermediate key/value pairs from the mapper (or combiner) and splits them up into shards, one shard per reducer

REDUCE SUB PHASES

- *Shuffle and sort* takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running.
  - These individual data pieces are then sorted by key into one larger data list
  - The purpose of this sort is to group equivalent keys together so that their values can be iterated over easily in the reduce task
- *Reduce* takes the grouped data as input and runs a reduce function once per key grouping
  - The function is passed the key and an iterator over all of the values associated with that key
  - Once the reduce function is done, it sends zero or more key/value pair to the final step, the output format
- *Output format* translates the final key/value pair from the reduce function and writes it out to a file by a record writer

GOLD STANDARD

- Linear scalability:
  - an algorithm running on twice the amount of data should take only twice as long
  - an algorithm running on twice the number of nodes should only take half as long
- Local aggregation: in the context of data-intensive distributed processing
  - the single most important aspect of synchronization is the exchange of intermediate results, from the processes that produced them to the processes that will ultimately consume them
  - Hadoop, intermediate results are written to local disk before being sent over the network
  - Since network and disk latencies are relatively expensive compared to other operations***, reductions in the amount of intermediate data translate into increases in algorithmic efficiency***
- Using the **combiner** and by taking advantage of the ability to **preserve state** across multiple inputs

  → it is possible to substantially reduce both the number and size of key-value pairs that need to be shuffled from the mappers to the reducers

COUNTING WORDS BASIC ALGORITHM

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        Emit(term t, count sum)
```

- the mapper emits an intermediate key-value pair for each term observed, with the term itself as the key and a value of one
- reducers sum up the partial counts to arrive at the final count

LOCAL AGGREGATION

Combiner technique

- Aggregate term counts across the documents processed by each map task
- Provide a general mechanism within the MapReduce framework to reduce the amount of intermediate data generated by the mappers
- Reduction in the number of intermediate key-value pairs that need to be shuffled across the network
  - from the order of total number of terms in the collection to the order of the number of ***unique*** terms in the collection

class MAPPER
    method MAP(docid a, doc d)
        $H \leftarrow$ new ASSOCIATIVEARRAY
        for all term $t \in$ doc d **do**
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ **do**
            EMIT(term t, count $H\{t\}$)

IN-MAPPER COMBINING PATTERN: ONE STEP FURTHER

- The workings of this algorithm critically depends on the details of how map and reduce tasks in Hadoop are executed
- Prior to processing any input key-value pairs, the mapper's `Initialize` method is called
  - which is an API hook for user-specified code
  - We initialize an associative array for holding term counts
  - Since it is possible to preserve state across multiple calls of the `Map` method (for each input key-value pair), we can
    - continue to accumulate partial term counts in the associative array across multiple documents,
    - emit key-value pairs only when the mapper has processed all documents
- Transmission of intermediate data is deferred until the `Close` method in the pseudo-code

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

Thanks Merci

Gracias

Contact: Genoveva Vargas-Solar, CNRS, LIG-LAFMIA

Genoveva.Vargas@imag.fr

http://www.vargas-solar.com/teaching

SOME BOOKS

- Hadoop The Definitive Guide – O'Reily 2011 – Tom White
- Data Intensive Text Processing with MapReduce – Morgan & Claypool 2010 –Jimmy Lin, Chris Dyer – pages 37-65
- Cloud Computing and Software Services Theory and Techniques– CRC Press 2011- Syed Ahson, Mohammad Ilyas – pages 93-137
- Writing and Querying MapReduce Views in CouchDB – O'Reily 2011 –Brandley Holt – pages 5-29
- NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence by Pramod J. Sadalage, Martin Fowler