6 824

wu

September 15, 2022

Contents

1 map reduce

1.1 programming model

the computation takes a set of **input** key/value pairs, and produces a set of **output** key/value pairs. The user of the MapReduce library expresses the computation as two functions: **Map** and **Reduce**

Map, written by the user, takes an input pair and produces a set of **intermediate** key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the **Reduce** function

The **Reduce** function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation.

1.1.1 example

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1")

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    results += ParseInt(v)
  Emit(AsString(result))
```

The `map` function emits each word plus an associated count of occurrences. The `reduce` function sums together all counts emitted for a particular word

1.1.2 Types

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

1.1.3 More examples

Distributed Grep: the map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output

Count of URL Access Frequency: the map function processes logs of web page requests an outputs $\langle$`URL,1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL,total count`$\rangle$ pair

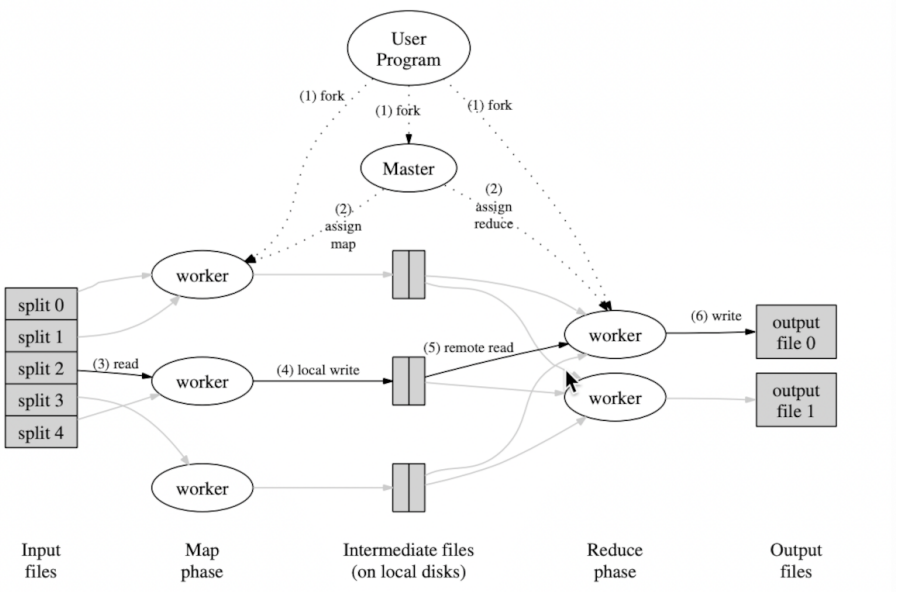
Term-vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$

pairs. The map function emits a ⟨`hostname,term vector`⟩ pair for each input document. The reduce function is passed all per-document term vectors for a given host. It add these term vectors together, throwing away infrequent terms, and then emits a final ⟨`hostname,term vector`⟩ pair

1.2 Implementation

1.2.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \mod R$). The number of partitions and the partitioning function are specified by the user



When the user program calls the `MapReduce` function, the following sequence of actions occurs

1. the MapReduce library in the user program first splits the input files into M pieces and starts up many copies of the program on a cluster of machines

2. one of the copies of the program is special - the master. The rest are workers that are assigned work by the master. there are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task

3. a worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory

4. periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers

5. when a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it **sorts** it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used

6. the reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition

7. when all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the *MapReduce* call in the user program returns back the user code

1.2.2 Master data structures

for each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), identity of the worker machine.

For each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed

1.2.3 Fault tolerance

1. worker failure The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed.

   Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible

2. master failure It is easy to make the master write periodic checkpoints of the master data structures described above.

3. Semantics in the presence of failures

2 Raft paper

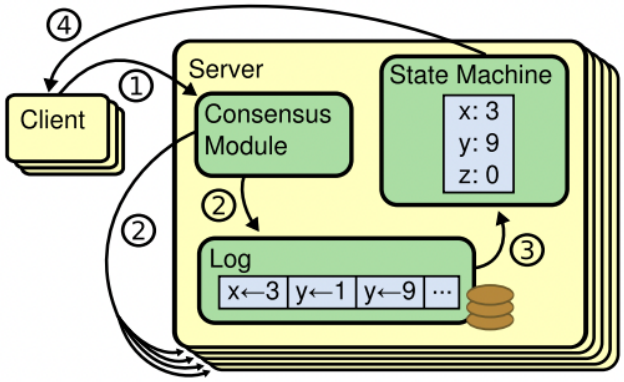
Raft is a consensus algorithm for managing a replicated log.

2.1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members

2.2 Replicated state machines

Replicated state machines are typically implemented using a replicated log, as shown in figure Each server stores a log containing a series of commands,



which its state machine executes in order. Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients.

Consensus algorithms for practical systems typically have the following properties:

- they ensure **safety** (never returning an incorrect result) under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering
- they are fully functional as long as any majority of the servers are operational and can communicate with each other and with clients. Thus a typical cluster of five servers can tolerate the failure of any two servers
- they do not depend on timing to ensure the consistency of the logs
- in the common case, a command can complete as soon as a majority of the cluster has responded to a single round of remote procedure calls

2.3 The Raft consensus algorithm

Raft implements consensus by first electing a distinguished **leader**, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tell servers when it is safe to apply log entries to their state machines.

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems:

- **leader election**
- **log replication**
- **safty** : the key safety property for Raft is the State Machine Safety Property: if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index
  - **Election Safety**: at most one leader can be elected
  - **Leader Append-Only**: a leader never overwrites or deletes entries in its log; it only append new entries

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.

- **Log Matching**: if two logs contain an entry with the same index and term, then logs are identical in all entries up through the given index
- **Leader Completeness**: if a log entry is commited in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms
- **State Machine Safety**: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index

2.3.1 Raft basics

At any given time each server is in one of three states: **leader**, **follower** or **candidate**. In normal operation there is exactly one leader and all of the other servers are followers.

- Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates.
- The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader)
- Candidate is used to elect a new leader

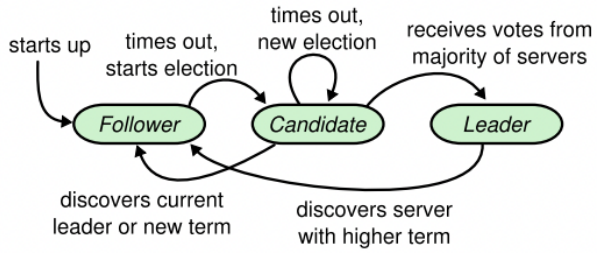


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.