[MapReduce 1](#_Toc433700523)

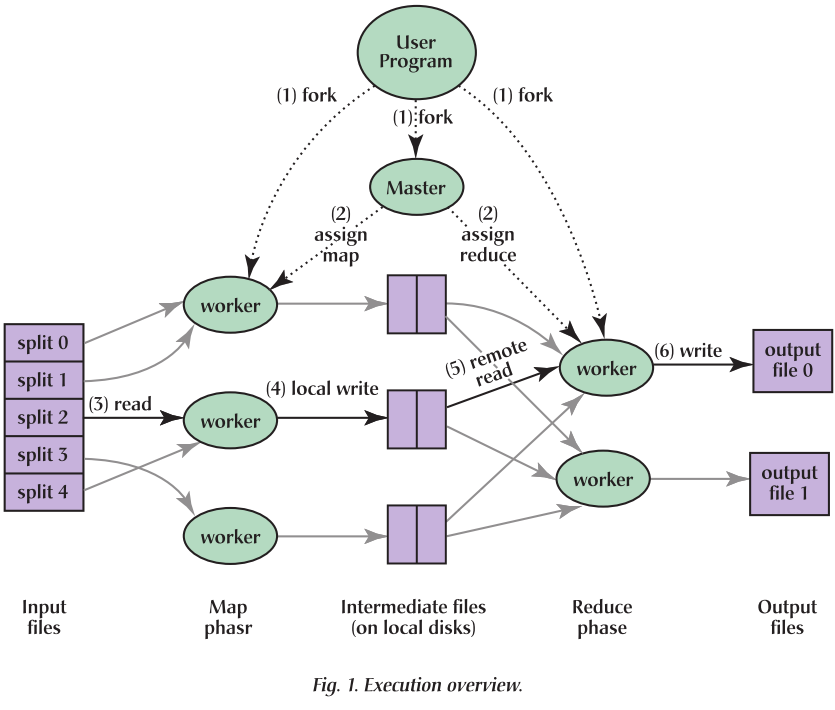
[From [Dean/Ghemawat 2008 - MapReduce: Simplified Data Processing on Large Clusters] 1](#_Toc433700524)

[From [https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html] 4](#_Toc433700525)

# MapReduce

### From [Dean/Ghemawat 2008 - MapReduce: Simplified Data Processing on Large Clusters]

* **Model**: map and reduce to parallelize large computations, use reexecution as primary mechanism for fault tolerance
* **Programming Model**:
  + Takes set of *input key/value pairs* and produces set of *output key/value pairs*
  + Two functions: *map* and *reduce*
  + Map:
    - Takes input pair and produces set of *intermediate key/value pairs*
    - MR library groups all intermediate values with same intermediate key and passes them to the reduce function
  + Reduce:
    - Accepts intermediate key and a set of values for that key
    - Merges values together to form a possibly smaller set of values
    - Typically: produces 0 – 1 output values per reduce invocation
    - Intermediate values are supplied to users reduce function via iterator
* **Implementation** on computing environment with large clusters of commodity PC’s connected together with switched Gigabit Ethernet



* + **Execution**:
    - **Map invocations**: distributed across multiple machines by automatically partitioning input data into set of M splits
    - **Reduce invocations**: distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g. hash(key)%R).
    - **Execution flow**:
      1. Split input files into M pieces (each 16 – 64MB, user controllable via optional parameter). Then starts many copies of the program on a cluster of machines
      2. **Master** copy of program: special, rest are Workers. Master assigns work to workers. M map tasks, R reduce tasks to assign. Master picks idle workers, assigns each a map or a reduce task.
      3. Worker who is assigned a map task: reads contents of corresponding input split, parses k/v pairs from input data, and passes each pair to the user defined map function. Intermediate k/v pairs from map function: buffered in memory
      4. Periodically write buffered pairs to local disk, partitioned into R regions by the partitioning function. Locations of buffered pairs are passed back to master, who is responsible for forwarding locations to reduce workers
      5. When reduce worker is notified by master about locations: uses RPC to read buffered data from local disks of map workers.
         * When reduce worker has read all intermediate data for its partition, it **sorts** it by the intermediate keys (such that all occurrences of the same key are grouped together). Sorting is needed: many different keys map to same reduce task. If intermediate data too large to fit in memory: external sort is used.
      6. Reduce worker iterates over sorted intermediate data
         * For each unique intermediate key, key & corresponding values are passed to reduce function
         * Ouput of reduce function is appended to final output file for this reduce partition.
      7. When all map and reduce tasks are completed: master wakes up user program. Returns back to the user code.
      8. On completion: R output files (one per reduce task, file names specified by users). May be used in another MR call, thus, not combined anymore.
  + **Master Data Structures**
    - Master: stores
      * State (idle, in-progress, completed)
      * Identity of worker machine (for nonidle tasks)
      * Locations/sizes of the R intermediate file regions produced by map task (updates on location/size as soon as map tasks are completed)
        + Information incrementally pushed to workers with in-progress reduce tasks
  + **Fault Tolerance**
    - **Handling Worker Failures**:
      * Master pings workers periodically (no response: mark as failed 🡪 map tasks (completed (as the node is inaccessible, the data cannot be recovered) or failed) of this worker are set to idle 🡪 map tasks can be scheduled on other workers
      * Completed reduce task don’t need to be reexecuted: they are stored on GLOBAL FS
      * If A executes map and fails and B executes A’s map, all workers executing **reduce** tasks are notified of the reexecution. Any reduce task that has not read data from A will read data from B.
    - **Semantics in the presence of failures**
      * Each in-progress task writes its output to private temp files (map produces R temp files and reduce task one temp file)
      * When map task completes, worker sends message to master & includes names of R temp files in message.
      * If master receives completion message for already completed map task, it ignores the message, otherwise, it records the names of R files in a master data structure.
    - **Locality**
      * Bandwidth restricted: schedule map task such that the machine contains a replica of the corresponding data
        + Else: attempts to schedule map task near replica of the tasks input data (same network switch machine)
      * Read locally, don’t use much bandwidth!
    - **Task Granularity**
    - **Backup Tasks**
      * Problems with slow machines (straggler): when MR operation is close to completion, schedule **backup executions** of remaining in-progress tasks. Task is marked as completed whenever either primary or backup execution completes 🡪 much faster.
  + **Refinements**
    - User specified partitioning functions for mapping of intermediate key values to R reduce shards
    - Ordering guarantees (increasing key order processing)
    - User specified combiner (partial combination of generated intermediate values with same key within same map task)
    - Custom input/output types
    - Mode for single machine execution (debugging, small-scale testing)

### From [https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html]

Question:

* Distributed File System? 🡪 nope, store it in DHT directly put/get (start simple, quick testing!!)
* YARN: Hadoop replaces initial map reduce engine with yarn. Manages multiple application and computing models and deployment of non-MR applications (no JobTracker/TaskTracker 🡪 new entities are independent of application)
* MR: input >> map >> **combine** >> reduce
  + Combine may be different but may often be the same as reducer, as it aggregates locally the values (e.g. map: <hello, 1>…<hello, 1> >> combine >> <hello, 2> on local node…). Combiner reduces data send to reducer jobs
* Reducer in Hadoop: Shuffle/Sort/secondary sort/reduce, see also reducer numbers in MapReduceTutorial/**how many reduces?**