HDFS – The Hadoop Distributed Filesystem

In high-performance computing (HPC) and many traditional cluster architectures, storage is viewed as a distinct and separate component from computation.

As dataset sizes increase, more computer capacity is required for processing.
But as compute capacity grows, the link between the compute nodes and the storage becomes a bottleneck. At that point, one could invest in higher performance but more expensive networks (e.g., 10 gigabit Ethernet). In most cases, this is not a cost-effective solution, as the price of networking equipment increases non-linearly with performance.

Alternatively, one could **abandon the separation of computation and storage nodes** as distinct components in a cluster. The distributed file system (DFS) that underlies MapReduce adopts exactly this approach.

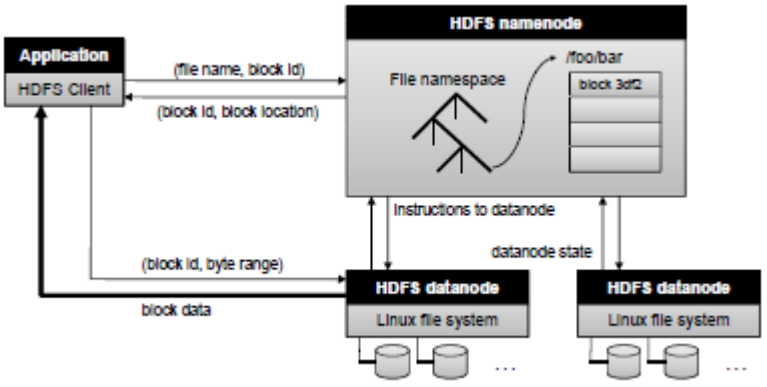
DISTRIBUTED FILE SYSTEM

The main idea is to divide user data into blocks and **replicate those blocks** across the local disks of nodes in the cluster.

- Write once, read many workloads
  - This implies no handle concurrency but allow replication
- Optimized for throughput (no random access) – Streaming data access
- Blocking data, of course, is not a new idea, but DFS blocks are significantly larger than block sizes in typical single-machine file systems (64 MB by default).
  - This avoids problems related to metadata managements
  - This makes transfer times larger than seek latency
- The system is built from unreliable but inexpensive commodity components. As a result, *failures are the norm rather than the exception*.

A MASTER-SLAVE ARCHITECTURE – NAMENODE AND DATANODES

The distributed file system adopts a master-slave architecture in which the master maintains the file namespace (*metadata, directory structure, file-to-block mapping, location of blocks, and access permissions*) and the slaves manage the actual data blocks.
In Hadoop the master is called **NAMENODE** and the slaves are called **DATANODES**.



Is single-master design a good choice?

A single-master design significantly simplifies implementation.
However, if the master goes down, the entire file system becomes unavailable, which trivially guarantees that the file system will never be in an inconsistent state.

The single-master design is a well-known weakness, since if the master goes offline, the entire file system and all MapReduce jobs running on top of it will grind to a halt. This weakness is mitigated in part by the lightweight nature of file system operations. Recall that no data is ever moved through the namenode. Because of this, the namenode rarely is the bottleneck. In practice, this single point of failure is not as severe a limitation as it may appear.

Furthermore, the Hadoop community is well-aware of this problem and has developed several reasonable workarounds – for example, a **warm standby namenode** that can be quickly switched over when the primary namenode fails.

BLOCK REPLICATION

By default, HDFS stores three separate copies of each data block to ensure both reliability, availability, and performance. In large clusters, the three replicas are spread across different physical racks, so HDFS is resilient towards two common failure scenarios:

- individual datanode crashes and
- failures in networking equipment that bring an entire rack offline.

Replicating blocks across physical machines also increases opportunities to co-locate data and processing in the scheduling of MapReduce jobs, since multiple copies yield more opportunities to exploit locality.

The namenode is in periodic communication with the datanodes to ensure proper replication of all the blocks: if there aren't enough replicas (e.g., due to disk or machine failures or to connectivity losses due to networking equipment failures), the namenode directs the creation of additional copies; if there are too many replicas (e.g., a repaired node re-joins the cluster), extra copies are discarded.

DATA FLOW

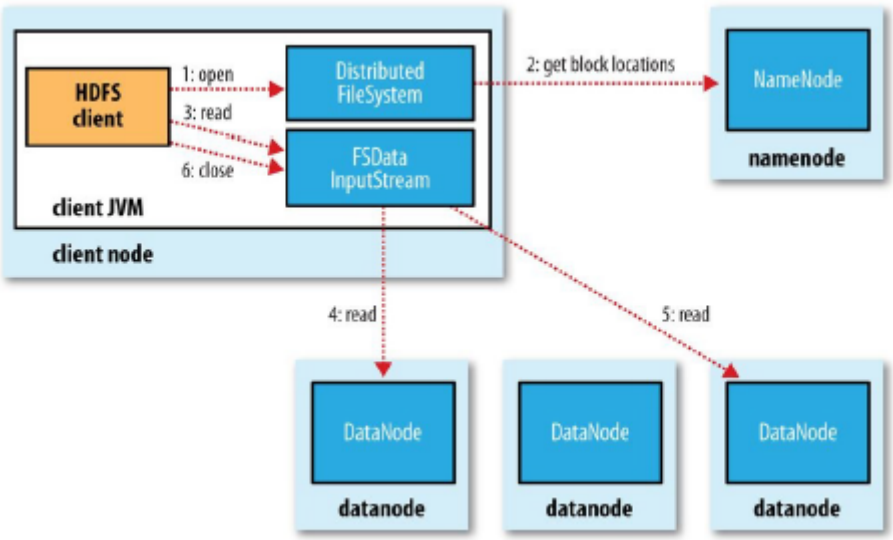
Anatomy of a File Read

In HDFS, an application client wishing to read a file (or a portion thereof) follows these steps:

1. First, the client opens the file he wants to read by calling `open()` on the `FileSystem` object, which for HDFS is an instance of `DistributedFileSystem` object.
2. `DistributedFileSystem` contacts the *namenode* to determine the locations of **the first few blocks in the file**.
    - The namenode returns for each block a **set** of datanodes that have o copy of the block. Blocks are sorted according to their proximity to the client.
    - The `DistributedFileSystem` returns a **`FSData`**`InputStream` object to the client to read data from. `FSDataInputStream` wraps a **`DFS`**`InputStream`, which manages the datanode and namenode I/O.

3. The client then calls `read()` on the stream. **DFS**`InputStream`, which has stored the datanode addresses of the *first blocks in the file*, then connects to the first (closest) datanode for the first block in the file. The client will call repeatedly `read()` on the stream object.
    - It's important to notice that **all data transfer occurs directly between clients and datanodes**; communications with the namenode only involves transfer of metadata.
4. When the end of the first block is reached, the connection to the datanode is closed and
5. A new connection to the best datanode for the next block is started by the **DFS**`InputStream`. *This happens transparently to the client, which from its point of view is just reading a continuous stream*.
    - Depending on the size of file to read, the **DFS**`InputStream` could call again the namenode to retrieve the set of datanode locations associated to the next set of blocks.
6. When the client has finished reading, he calls `close()` on the **FSData**`InputStream`.

During reading, if the **DFS**`InputStream` encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't retry them for later blocks.



Anatomy of a File Write

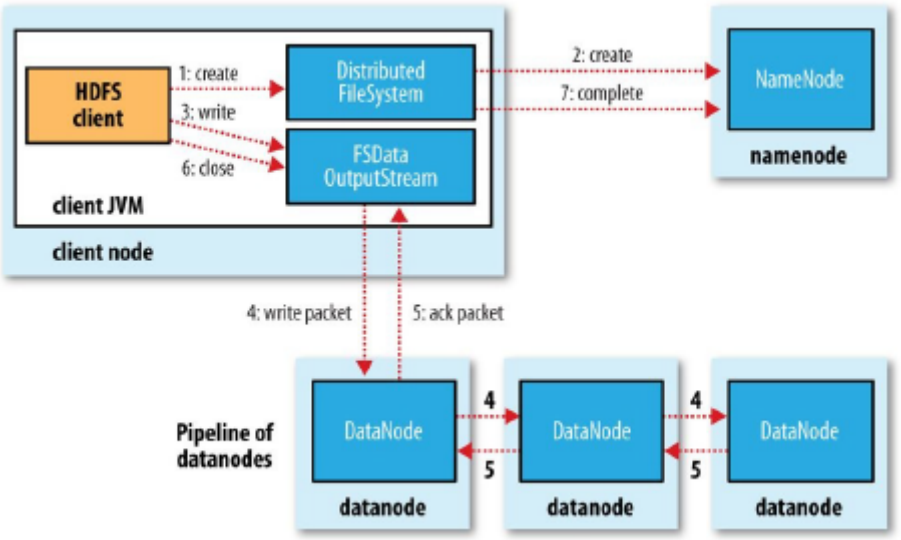
To create a new file and write data to HDFS:

1. The client creates the file by calling `create()` on `DistributedFileSystem`.
2. `DistributedFileSystem` makes a call to the namenode to create a new file in the filesystem' s namespace after checking that the file doesn't already exist and that the client has the right permissions to create the file. If checks fail, file creation fails.
    - The `DistributedFileSystem` returns an **FSData**`OutputStream` object for the client to start writing data to. Just in the read case, **FSData**`OutputStream` wraps a **DFS**`OutputStream`, which handles communication with the datanodes and namenode.
3. As the client writes data, the **DFS**`OutputStream` splits it into packets, which it writes to an internal queue called the **DATA QUEUE**.
4. The data queue is consumed by a `DataStreamer`, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas (3 by default).

The list of datanodes forms a pipeline. The `DataStreamer` streams the packets to the first datanode in the pipeline, which stores each packet and forwards it to the second datanode in the pipeline and so on.

5. The **DFS**`OutputStream` also maintains an internal queue of packets that are waiting to be acknowledged by datanode, called the **ACK QUEUE**. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline.
6. When the client has finished writing data, it calls `close()` on the **FSData**`OutputStream`. This actions flushed all the remaining packets to the datanode pipeline and waits for acknowledgements before contacting the namenode to signal that the file is complete.

In the most recent release of Hadoop, files are immutable – they cannot be modified after creation. There are current plans to officially support file appends in the near future.

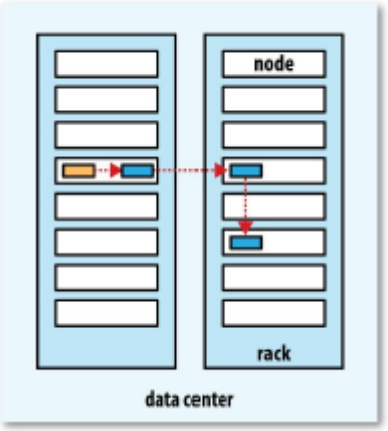


Replica Placement

How does the namenode choose which datanodes to store replicas on?
There is a trade-off between reliability and write bandwidth and read bandwidth.

Hadoop's default strategy is to place:

- The first replica ***on the same node as the client***.
- The second replica is placed ***on a different rack from the first*** (off-rack), chosen at random.
- The third replica is placed ***on the same rack as the second***, but on a different node chosen at random.
- Further replicas are placed ***on random nodes in the cluster***, although the system tries to avoid placing too many replicas on the same rack.



HDFS Coherency Model

A coherency model for a filesystem describes the data visibility of reads and writes for a file.
After creating a file, it is visible in the filesystem namespace as expected.

However, any content written to the file is not guaranteed to be visible, even if the stream is flushed (Flushing output on a buffered stream means transmitting all accumulated characters to the file). So, the file appears to have a zero length.
Once more than a block of data has been written, the first block will be visible to new readers. This is true of subsequent blocks, too: it is always the current block being written that is not visible to other readers.

NAMENODE RESPONSABILITIES - SUMMARY

In summary, the HDFS namenode has the following responsibilities:

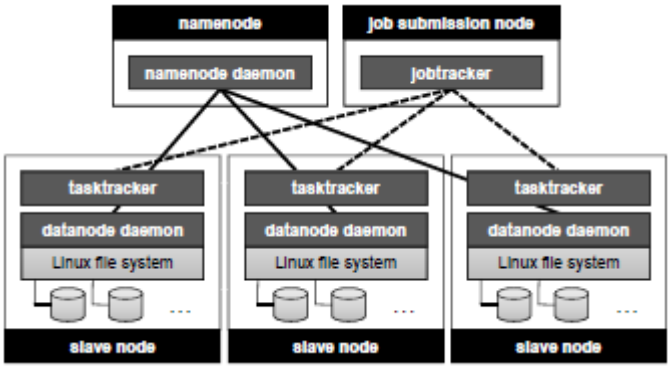
- **Namespace management**
  The namenode is responsible for maintaining the file namespace, which includes metadata, directory structure, file-to-block mapping, location of blocks, and access permissions. These data are held **in memory** for fast access and all mutations are persistently logged.
- **Coordinating file operations**
  The namenode directs application clients to datanodes for read operations, and allocates blocks on suitable datanodes for write operations. All data transfers occur directly between clients and datanodes. When a file is deleted, HDFS does not immediately reclaim the available physical storage; rather, blocks are slowly garbage collected.
- **Maintaining overall health of the file system and rebalancing operation**
  The namenode is in periodic contact with the datanodes via heartbeat messages to ensure the integrity of the system. If the namenode observes that a data block is under-replicated (fewer copies are stored on datanodes than the desired replication factor), it will direct the creation of new replicas. Finally, the namenode is also responsible for rebalancing the file system. During the course of normal operations, certain datanodes may end up holding more blocks than others; rebalancing involves moving blocks from datanodes with more blocks to datanodes with fewer blocks. This leads to better load balancing and more even disk utilization.

Hadoop MapReduce

HADOOP CLUSTER ARCHITECTURE

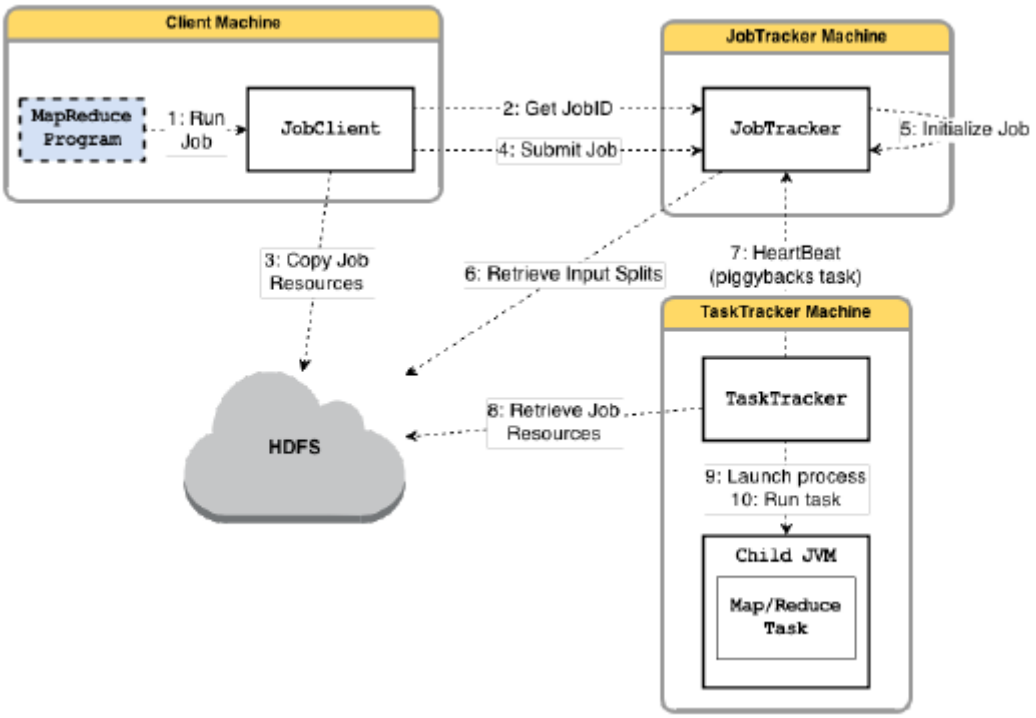
The architecture of a complete Hadoop cluster consists of three separate components:

- the HDFS master (called the **NAMENODE**)
- the job submission node (called the **JOBTRACKER**), and
- many slave nodes. Each of the slave nodes runs a **TASKTRACKER** for executing map and reduce tasks and a datanode daemon for serving HDFS data.



In particular, the **JOBTRACKER**, which is the single point of contact for a client wishing to execute a MapReduce job. The jobtracker monitors the progress of running MapReduce jobs and is responsible for coordinating the execution of the mappers and reducers.

ANATOMY OF A MAPREDUCE JOB RUN



Job Submission

The `runJob()` method on `JobClient` is a convenience method that creates a new `JobClient` instance and calls `submitJob()` on it. After having submitted the job, `runJob()` polls the job's progress once a second and reports the progress to the console of the client.

When the job is complete, if it was successful, the job counters are displayed; otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by `submitJob()` method does the following:

- Asks the job tracker for a new job ID (*step 2*).
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
- Computes the input splits for the job. If the splits cannot be computed (because the input paths don't exist, for example), the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job (*step 3*), including:
    - the job JAR file,
    - the configuration file, and
    - the computed input splits,

  to the jobtracker's filesystem in a directory named after the job ID.
  The job JAR is copied 10 times by default, so that there are lots of copies across the cluster for the task trackers to access when they run tasks for the job.
- Tells the jobtracker that the job is ready for execution by calling `submitJob()` on jobtracker (*step 4*).

Job Initialization

When the jobtracker receives a call to its `submitJob()` method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it.
Initialization involves creating an object to represent the job being run – a container – which encapsulate its tasks, and bookkeeping information to keep track of the tasks' status and progress (*step 5*).
In order to create the list of tasks to run, the job scheduler retrieves the input splits computed in the client from the share file system (*step 6*) and creates:

- a map task object for each split,
- as well as a number of reduce task objects specified by the programmer.

Tasks are given IDs at this point.

Task assignment

Tasktrackers run a simple loop that periodically sends **HEARTBEAT** method calls to the jobtracker.
This is used:

- to advice the jobtracker that a tasktracker is alive and
- to advice the jobtracker whether it is ready to run a new task (*step 7*)

If a tasktracker is ready to run, the jobtracker will allocate it a task. How? The jobtracker first needs to select a job by using a particular **SCHEDULING ALGORITHM** and then chooses a task for the job.

Each tasktracker has a fixed number of slots for map tasks and for reduce tasks. The default scheduler fills empty map tasks slots before reduce task. ***Map tasks have higher priority than reduce tasks*** since all the map tasks must complete before the sort phase of the reduce can start (Shuffle and Sort).

Reduce tasks can run anywhere in the cluster, but ***request for map tasks have data locality constraints*** that the scheduler tries to honor. In the optimal case, the task is data local – that is running on the same node that the split resides on. Alternatively, the task may be ***rack local***: on the same rack, but not the same node, as the split. Some tasks are neither local nor rack local and retrieve their data from a different rack than the one they are running on.

Task Execution

Now that the tasktracker has been assigned a task, the next step is for it to run the task.
Before it can run the task:

- it localized the resources that the task needs including the job configuration and the JAR file, and any files from the HDFS (*step 8*).
- Then the tasktracker creates a local working directory for the task and un-jars the content of the JAR into this directory
- Finally, the `TaskRunner` launch a Child JVM (*step 9*) to run the task (*step 10*).

The Child runs in a dedicated JVM, so that any bugs in the user-defined map and reduce functions don't affect the tasktracker – by causing it to crash for example.

Obviously, the child process communicates with its parent in order to inform it of the task's progress every few seconds until the task is complete.

Streaming and Pipes

Streaming and Pipes runs special map and reduce tasks for the purpose of launching the user-supplied executable and communicating with it.
During execution of the task, the Java process passes input key-value pairs to the external process, which runs it through the user-defined map or reduce function and passes the output key-value pairs to the external back to the Java process.

Job Completion

When the jobtracker receives a notification that the last task for a job is complete, is changes the status for the job to "successful". Then, when the JobClient polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the `runJob()` method. Job statistics and counters are printed to the console at this point.

Finally, on job completion, the jobtracker clean up its working state for the job and instructs trasktracker to do the same (so intermediate output is deleted). Job information is archived by the job history server to enable later interrogation by users if desired.

FAILURES

One of the major benefits of using Hadoop is its ability to handle such failures and allow your job to complete successful.

Task Failure (is communicated to the Application Master)

The most common occurrence of this failure is when user code in the **map or reduce task throws a runtime exception**. If this happens, the child task JVM reports the error back to its parent tasktracker, before it exits. The tasktracker logs the error, marks the task attempt as failed and frees up a slot so its resources are available for another task.

Another failure mode is the **unexpected exit of the child task JVM**. In this case, the tasktracker notices that the process has exited and informs the application master so it can mark the attempt as failed.

Hanging (sospesi) tasks are dealt with differently. The tasktracker notices that it hasn't received a progress update for a while and proceeds to mark the task as failed and the child task JVM process will be killed. The timeout period after which a task is considered failed is normally 10 minutes but this parameter can be changed.

When the jobtracker is notified of a task attempt that has failed (through the tasktracker's heartbeat call), it will reschedule execution of the task. The jobtracker will try to avoid rescheduling the task on a taskmanager where it has previously failed. Furthermore, if a task fails four times (this parameter can be configured), it will not be retried again. By default, if any task fails four times, the whole job fails. Also this parameter can be configured.

TaskManager Failure

If a taskmanager fails by crashing or running very slowly, it will stop sending heartbeats to the jobtracker (or send them very infrequently). The jobtracker will notice a tasktracker that has stopped sending heartbeats if it hasn't received one for 10 minutes and remove it from its scheduling pool.

Any task or application master running on the failed tasktracker will be recovered. The jobtracker arranges also the map tasks completed successfully to be re-run, since their intermediate output residing on the failed tasktracker's local filesystem may not be accessible to the reduce task.

A tasktracker might be blacklisted if the number of failures for the application is too high.

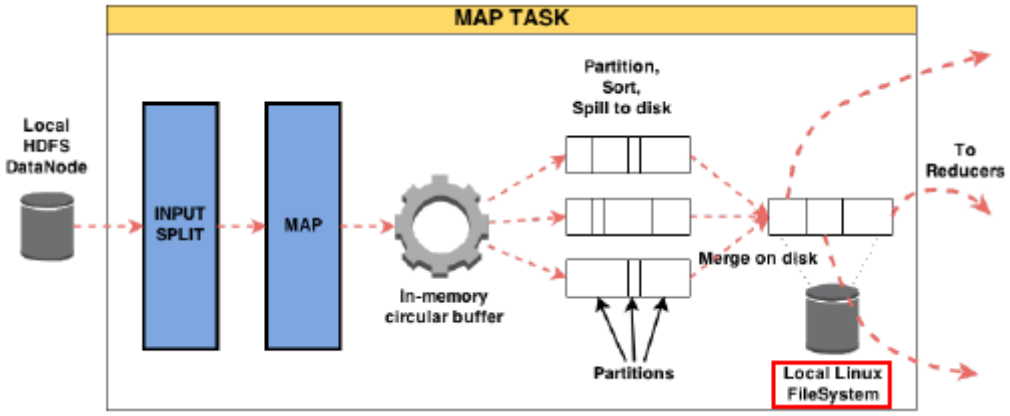
JobTracker Failure

Failure of the jobtracker is serious, because without it, neither jobs nor task containers can be launched.
To achieve high availability, it is necessary to run a pair of jobtracker in an active configuration. If the active jobtracker fails, then the standby can take over without significant interruption to the client.

SHUFFLE AND SORT

MapReduce makes the guarantee that the input to every reducer is sorted by the key.
The system by which the system performs the sort – and transfers the map outputs to the reducers as inputs – is known as **SHUFFLE**.

The Map Side (The result is a single map output file for each mapper)



The map function receives an input split as input. Typically, the input split corresponds to a data block.
When the map function starts producing output, it is not simply written to disk.
Each map task has an in-memory circular buffer (100MB by default) that it writes the output to.
When the contents of the buffer reach a certain threshold size (which has the default value 80%), a background thread will start to **SPILL PHASE**. Map output will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete.

Before the thread writes to disk, it first divides the data into partitions corresponding to the reducers that they will be sent to. Within each partition, the background thread performs an in-memory sort by key and, if there is a combiner function, it is run on the output of the sort.

Each time the memory buffer reaches the spill threshold, a new spill file is created. Before the task is finished, the spill files are **merged** into a single partitioned and sorted output file that is finally written on the *LOCAL Filesystem*. The output file's partitions are made available to the reducers over HTTP.

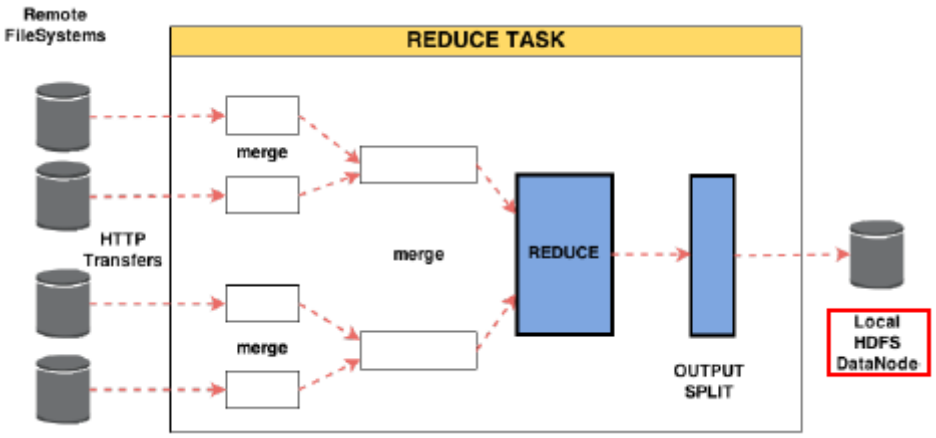
The Reduce Side

The map output file is sitting on the local disk of the machine that ran the map task. The reducer task needs the map output for its particular partition from several map tasks across the cluster.
The map tasks can finish at different times, so the reduce task starts copying their outputs as soon as each completes. This is known as the **COPY PHASE** of the reduce task and could be performed in memory or on disk depending on the outputs size.

How do reducers know which machines to fetch map output from?
As map tasks complete successfully, they notify their application master and so, for a given job, the application master knows the mapping between map outputs and hosts. On the other hand each reducer periodically asks the application master for the map outputs until it retrieved them all.

When all the map outputs have been copied, the reduce tasks moves into the **MERGE PHASE**, which merges the map outputs.

Unlike the previous case, the output of this phase is written directly to the HDFS.


Remote
FileSystems
REDUCE TASK
merge
HTTP
Transfers
merge
REDUCE
merge
OUTPUT
SPLIT
Local
HDFS
DataNode