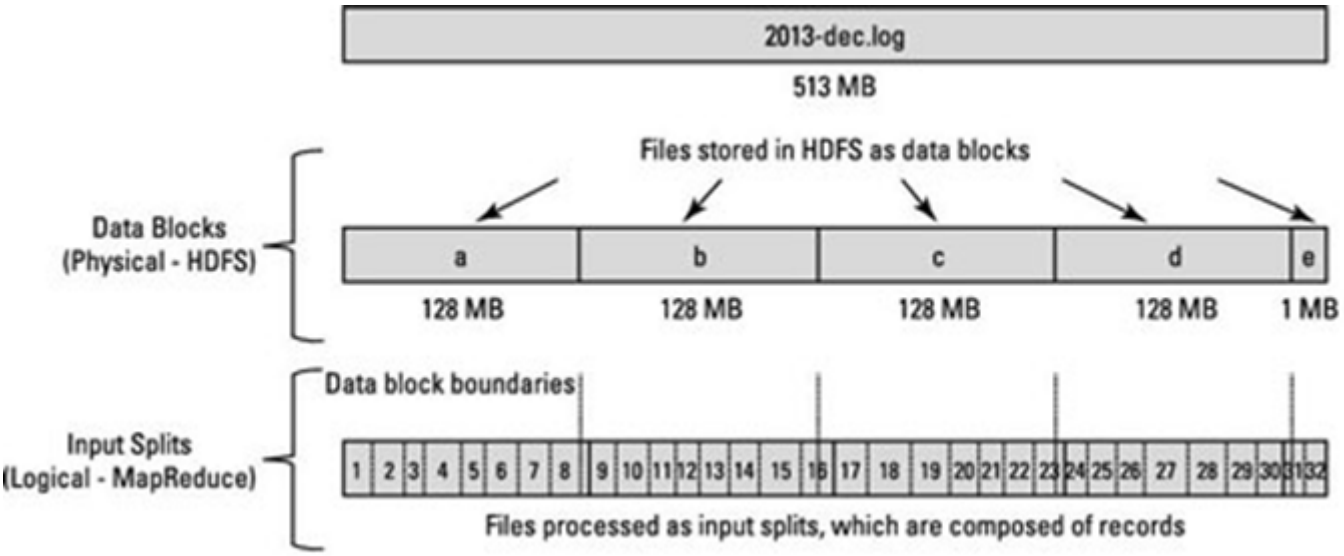
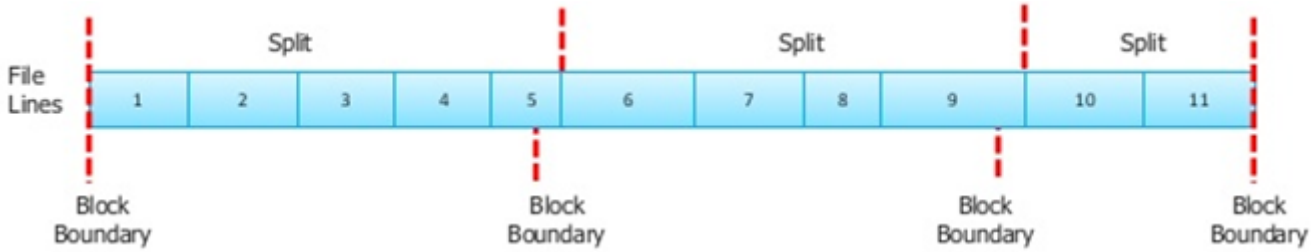
InputSplit and HDFS Blocks

- Input Split – A chunk of data processed by one mapper object.
  - It is further divided into records. Map function processes these records.
  - **Input splits** – logical record boundaries
  - **HDFS blocks** – physical boundaries (size)
- Input splits do not neatly fit into HDFS blocks.
- For example, a TextInputFormat's logical records are lines, which might cross HDFS block boundaries.
- It means that data-local maps (that is, maps that are running on the same host as their input data) will perform some remote reads. The slight overhead this causes is not normally significant.





Combiners

Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase. We can motivate the need for combiners by considering the word count algorithm, which emits a key-value pair for each word in the collection. These key-value pairs need to be copied across the network, and so the amount of intermediate data will be larger than the input collection itself. This is clearly inefficient. One solution is to perform local aggregation on the

output of each mapper, i.e., to compute a local count for a word over all the documents processed by the mapper. With this modification, the number of intermediate key-value pairs will be at most the number of unique words in the collection times the number of mappers (and typically far smaller because each mapper may not encounter every word).

One can think of combiners as "mini-reducers" that take place on the output of the mappers, prior to the shuffle and sort phase. Each combiner operates in isolation and therefore does not have access to intermediate output from other mappers. The combiner is provided keys and values associated with each key (the same types as the mapper output keys and values).

Critically, one cannot assume that a combiner will have the opportunity to process all values associated with the same key. The combiner can emit any number of key-value pairs, but the keys and values must be of the same type as the mapper output (same as the reducer input). In cases where an operation is both associative and commutative (e.g., addition or multiplication), reducers can directly serve as combiners. In general, reducers and combiners are not interchangeable.

Partitioners

Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. In other words, the partitioner specifies the task to which an intermediate key-value pair must be copied. Within each reducer, keys are processed in sorted order (which is how the "group by" is implemented). The simplest partitioner involves computing the hash value of the key and then taking the mod of that value with the number of reducers. This assigns approximately the same number of keys to each reducer.

The partitioner only considers the key and ignores the value therefore, a roughly-even partitioning of the key space may nevertheless yield large differences in the number of key-values pairs sent to each reducer (since different keys may have different numbers of associated values). This imbalance in the amount of data associated with each key is relatively common in many text processing applications due to the Zipfian distribution of word occurrences.

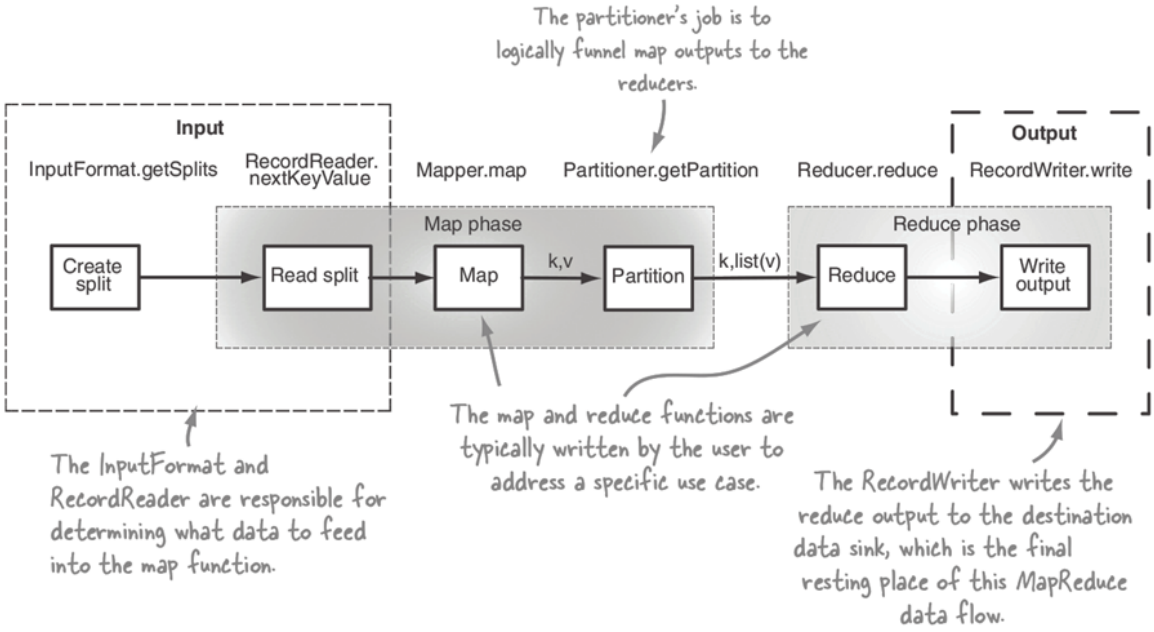
Map phase is done by mappers. ***Mappers*** run on unsorted input key/values pairs. Each mapper emits zero, one or multiple output key/value pairs for each input key/value pairs.

Combine phase is done by **Combiners**. Combiner should combine key/value pairs with the same key together. Each combiner may run zero, once or multiple times.

Shuffle and Sort phase is done by framework. Data from all mappers are grouped by the key, split among reducers and sorted by the key. Each reducer obtains all values associated with the same key. Programmer may supply custom compare function for sorting and **Partitioner** for data split.

Partitioner decides which Reducer will get a particular key value pair.

Reducer obtains sorted key/[values list] pairs sorted by the key. Value list contains all values with the same key produced by mappers. Each reducer emits zero, one or multiple output key/value pairs for each input key/value pair.



Advantages of the in-mapper combining design pattern

1) First, it provides control over when local aggregation occurs and how it exactly takes place.
In contrast, the semantics of the combiner is under specified in MapReduce. For example, Hadoop makes no guarantees on how many times the combiner is applied, or that it is even applied at all. The combiner is provided as a semantics-preserving optimization to the execution framework, which has the option of using it, perhaps multiple times, or not at all.

2) Second, in-mapper combining will typically be more efficient than using actual combiners.
One reason is the additional overhead associated with actually materializing the key-value pairs. Combiners reduce the amount of intermediate data that is shuffled across the network, but don't actually reduce the number of key-value pairs that are emitted by the mappers in the first place.
This process involves unnecessary object creation and destruction (garbage collection takes time), and furthermore, object serialization and deserialization (when intermediate key-value pairs fill the in-memory buffer holding map outputs and need to be temporarily spilled to disk).
In contrast, with in-mapper combining, the mappers will generate only those key-value pairs that need to be shuffled across the network to the reducers.

Disadvantages of the in-mapper combining design pattern

First, it breaks the functional programming underpinnings of MapReduce, since state is being preserved across multiple input key-value pairs. Preserving state across multiple input instances means that algorithmic behavior may depend on the order in which input key-value pairs are encountered. This creates the potential for ordering-dependent bugs, which are difficult to debug on large datasets in the general case (although the correctness of in-mapper combining for word count is easy to demonstrate).

Second, there is a fundamental scalability bottleneck associated with the in-mapper combining pattern.
It critically depends on having sufficient memory to store intermediate results until the mapper has completely processed all key-value pairs in an input split.

In the word count example, the memory footprint is bound by the vocabulary size, since it is theoretically possible that a mapper encounters every term in the collection. Heap's Law (http://en.wikipedia.org/wiki/Heaps%27_law), a well-known result in information retrieval, accurately models the growth of vocabulary size as a function of the collection size

- the somewhat surprising fact is that the vocabulary size never stops growing. Therefore, the algorithm in Figure 3.3 will scale only up to a point, beyond which the associative array holding the partial term counts will no longer fit in memory.

One common solution to limiting memory usage when using the in-mapper combining technique is to "block" input key-value pairs and "flush" in-memory data structures periodically. Instead of emitting intermediate data only after every key-value pair has been processed, emit partial results after processing every n key-value pairs. This is straightforwardly implemented with a counter variable that keeps track of the number of input key-value pairs that have been processed.

As an alternative, the mapper could keep track of its own memory footprint and flush intermediate key-value pairs once memory usage has crossed a certain threshold. In both approaches, either the block size or the memory usage threshold needs to be determined empirically: with too large a value, the mapper may run out of memory, but with too small a value, opportunities for local aggregation may be lost.

In Hadoop physical memory is split between multiple tasks that may be running on a node concurrently; these tasks are all competing for finite resources, but since the tasks are not aware of each other, it is difficult to coordinate resource consumption effectively. In practice, however, one often encounters diminishing returns in performance gains with increasing buffer sizes, such that it is not worth the effort to search for an optimal buffer size.

In MapReduce algorithms, the extent to which efficiency can be increased through local aggregation depends on the size of the intermediate key space, the distribution of keys themselves, and the number of key-value pairs that are emitted by each individual map task. Opportunities for aggregation come from having multiple values associated with the same key (whether one uses combiners or employs the in-mapper combining pattern). In the word count example, local aggregation is effective because many words are encountered multiple times within a map task.

Algorithmic Correctness With Local Aggregation

Although use of combiners can yield dramatic reductions in algorithm running time, care must be taken in applying them. Since combiners in Hadoop are viewed as optional optimizations, the correctness of the algorithm cannot depend on computations performed by the combiner or depend on them even being run at all. In any MapReduce program, the reducer input key-value type must match the mapper output key-value type: this implies that the combiner input and output key-value types must match the mapper output key-value type (which is the same as the reducer input key-value type).

In cases where the reduce computation is both commutative and associative, the reducer can also be used (unmodified) as the combiner (as is the case with the word count example). In the general case, however, combiners and reducers are not interchangeable.

Pros and Cons of MapReduce Programming

In addition to preparing the input data, the programmer needs only to implement the mapper, the reducer, and optionally, the combiner and the partitioner.

However, this also means that any conceivable algorithm that a programmer wishes to develop must be expressed in terms of a small number of rigidly-defined components that must fit together in very specific ways.

Programmer has limited control over data and execution flow

All algorithms must be expressed in m, r, c, p

When NOT to use MapReduce?

There are few scenarios where MapReduce programming model cannot be employed.

- If the computation of a value depends on previously computed values, then MapReduce cannot be used.
- One good example is the Fibonacci series where each value is summation of the previous two values. i.e., f(k+2) = f(k+1) + f(k).
- Also, if the data set is small enough to be computed on a single machine, then it is better to do it as a single reduce(map(data)) operation rather than going through the entire map reduce process.
- MapReduce is not a database system and its main purpose is to process large amounts of unstructured data (e.g., web crawling) and generate meaningful structured data as its output.

Things under our control

- Construct complex data structures as keys and values to store and communicate partial results.
- Execute user-specified initialization/termination code in a map or reduce task
- Preserve state in both mappers and reducers across multiple input or intermediate keys
- Control sort order of intermediate keys, and hence the order of how a reducer processes keys
- Control partitioning of key space, and hence the set of keys encountered by a reducer