MapReduce I

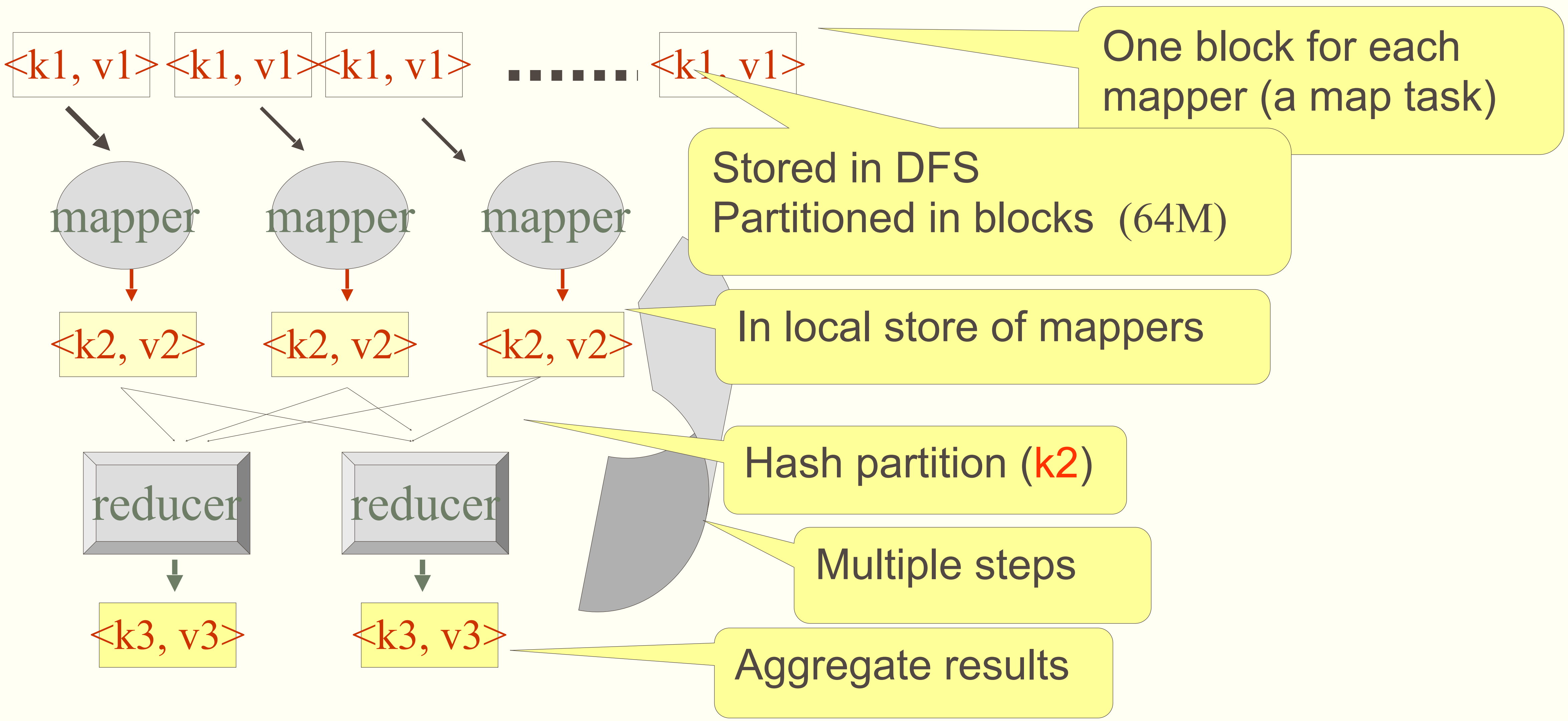
MapReduce

- MapReduce model
- MapReduce for relational operators
- MapReduce for graph querying

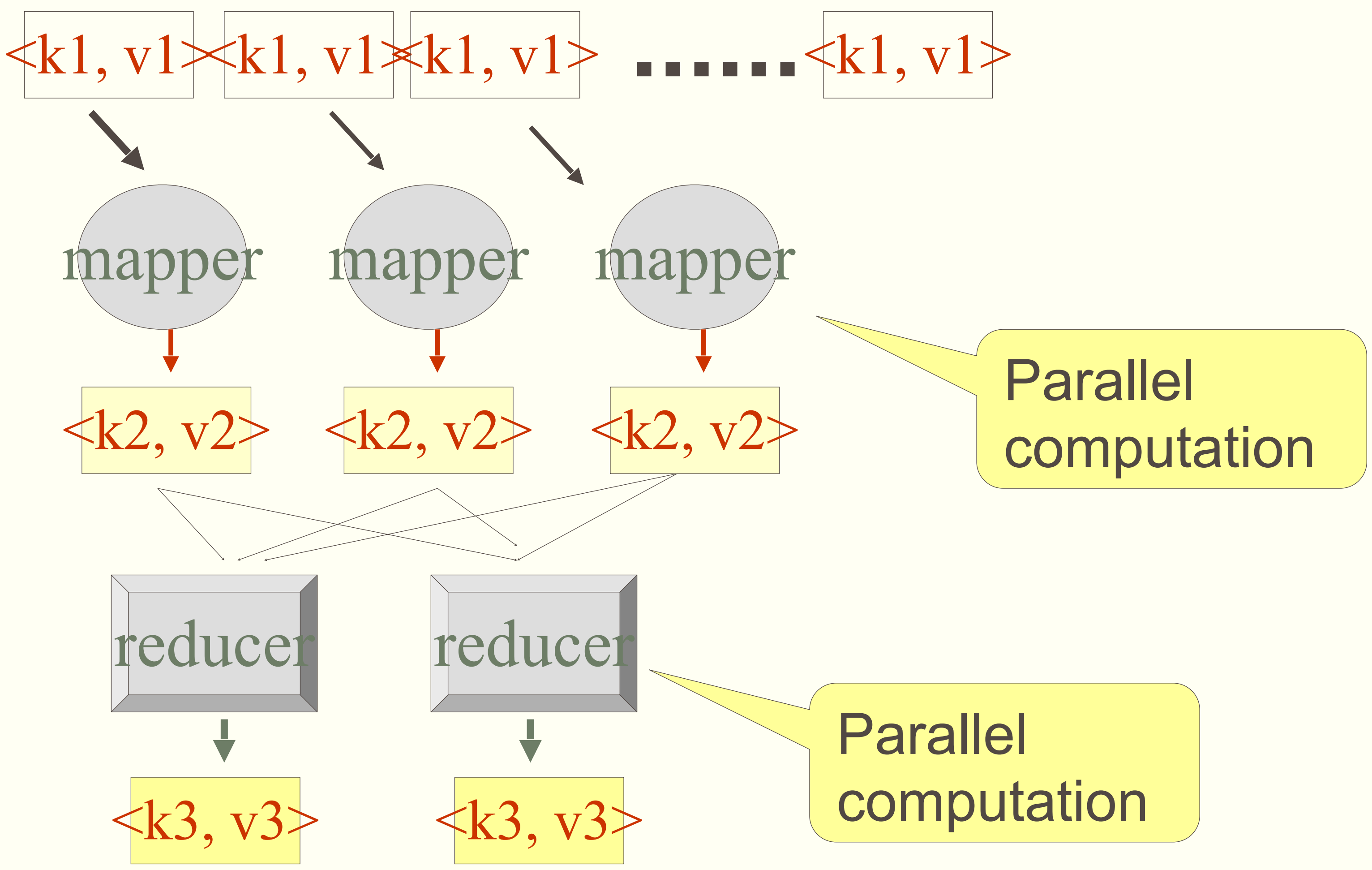
What is "MapReduce"

- A programming model with two primitive functions
  - Map: $< k_1, v_1 > \ \rightarrow list(k_2, v_2)$
  - Reduce: $< k_2, list(v_2) > \ \rightarrow list(k_3, v_3)$
- Input: a list $< k_1, v_1 >$ of key-value pairs
- Map:
  - Applied to each pair, computes key-value pairs $< k_2, v_2 >$
  - The intermediate key-value pairs are hash-partitioned based on k_2
  - Each partition $\left(k_2,\ list(v_2)\right)$ is sent to a reducer
- Reduce:
  - Takes a partition as input and computes key-value pairs $< k_3, v_3 >$
- This process any iterate – multiple map/reduce steps

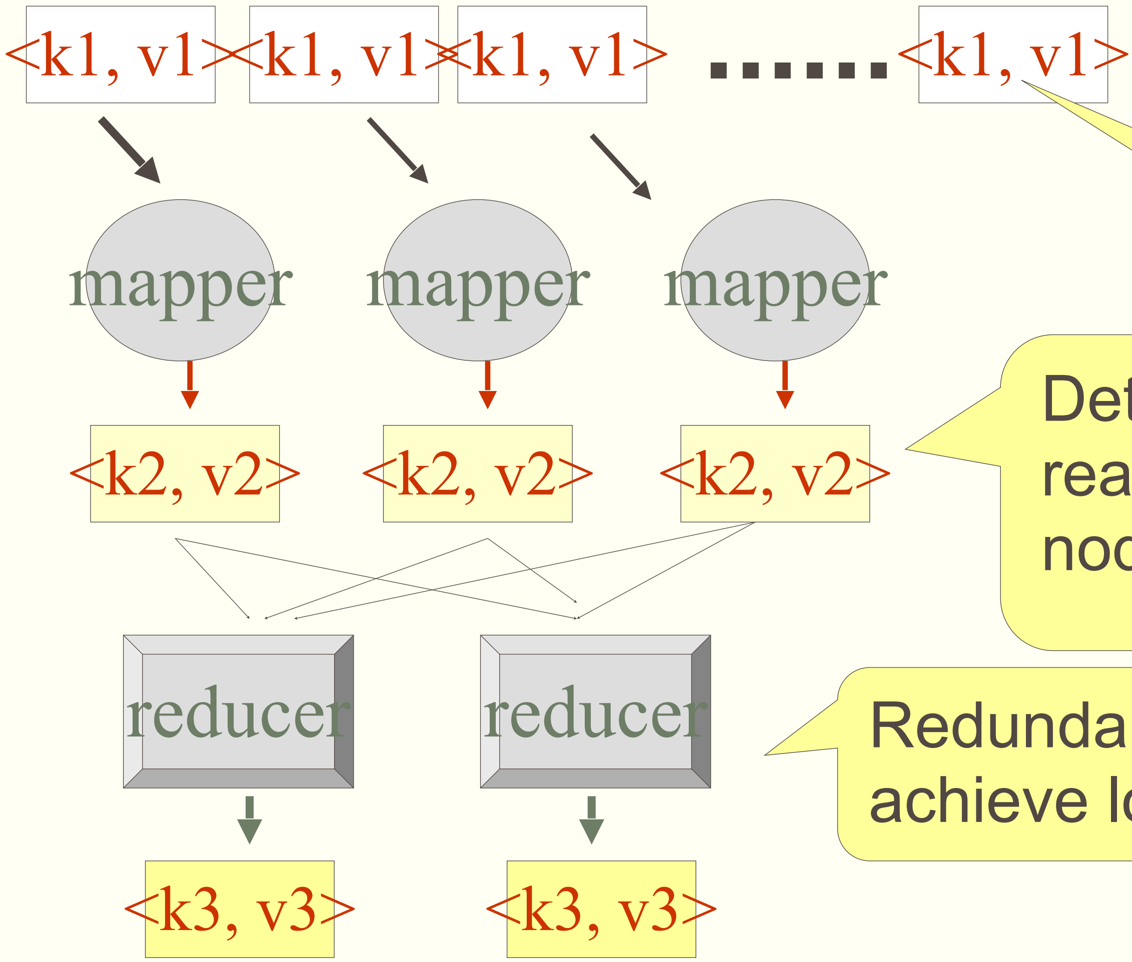
Architecture



Parallelism



Fault Tolerance



triplicated

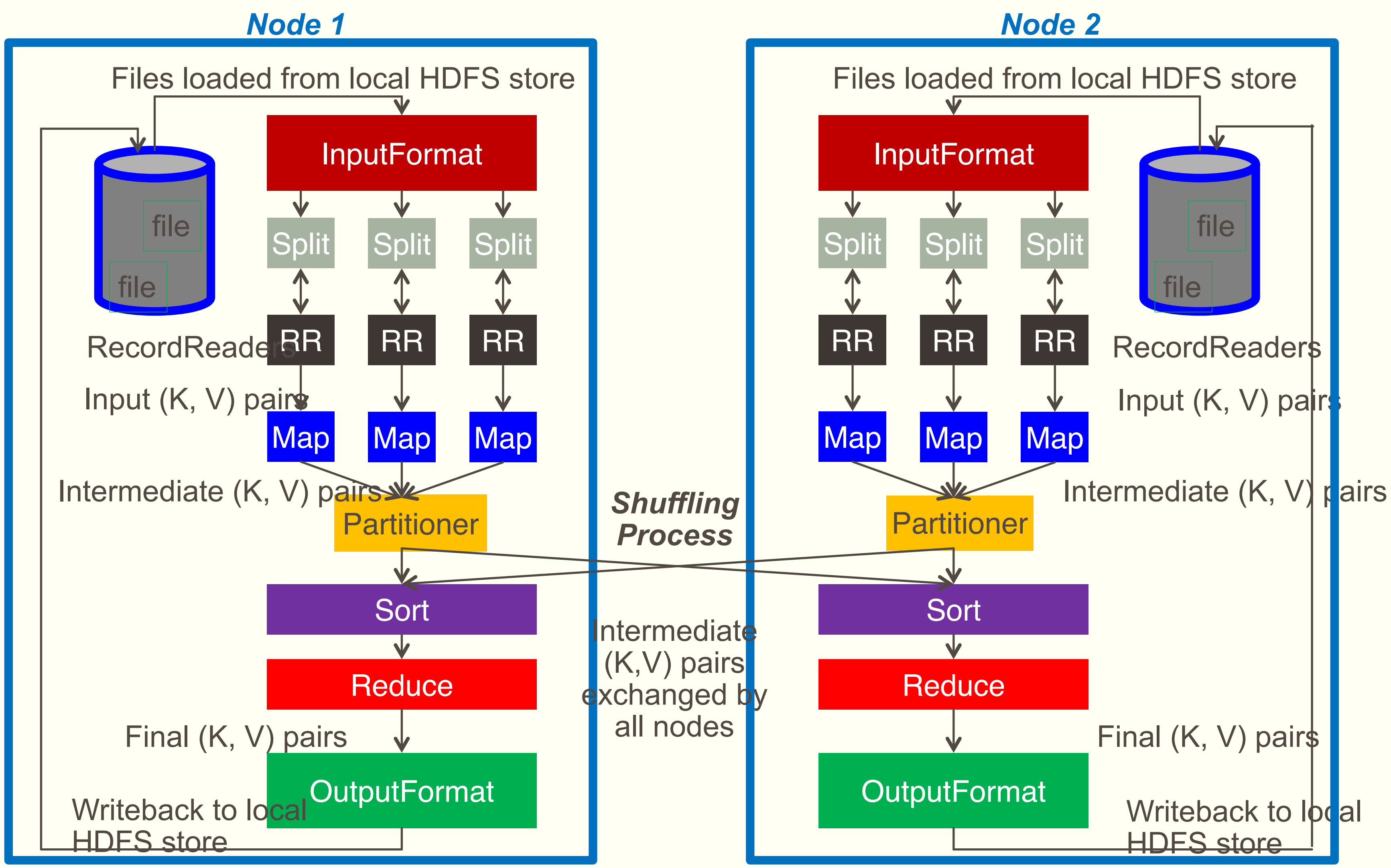
Detecting failures and reassigning the tasks of failed nodes to healthy nodes

Redundancy checking to achieve load balancing

Advantages of MapReduce

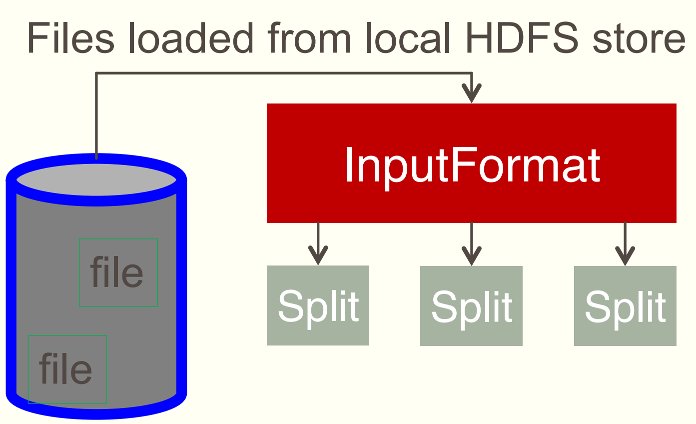
- Simple:
  - One only needs to define two functions
  - No need to worry about how the data is stored, distributed and how the operations are scheduled
- Scalability:
  - A large number of low-end machines
  - Scale Out (horizontally): Adding a new computer to a distributed software application; lost-cost "commodity"
  - Scale Up (vertically): Upgrade, add (costly) resources to a single node
- Independence
  - It can work with various storage layers
- Flexibility
  - Independent of data models and schema
- Fault tolerance

Hadoop MapReduce: A Closer Look



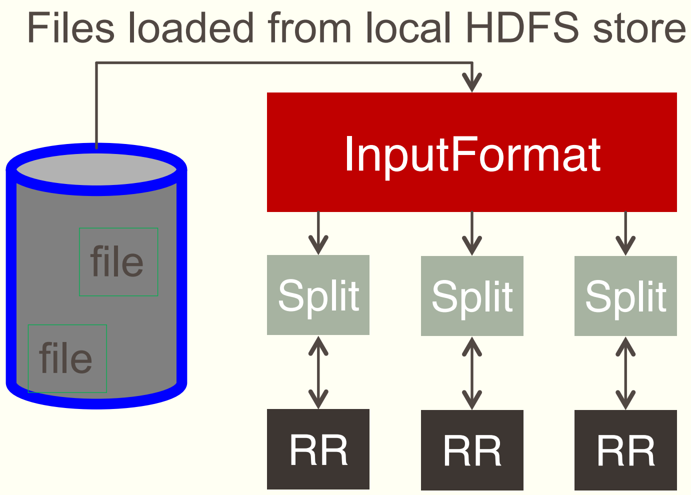
Input Splits

- An input split describes a unit of work that comprises a single map task in a MapReduce Program
- By default, the InputFormat breaks a file up into 64MB splits
- By dividing the file into splits, we allow several map task to operate on a single file in parallel
- If the file is very large, this can improve performance significantly through parallelism
- Each map task corresponds to a single input split



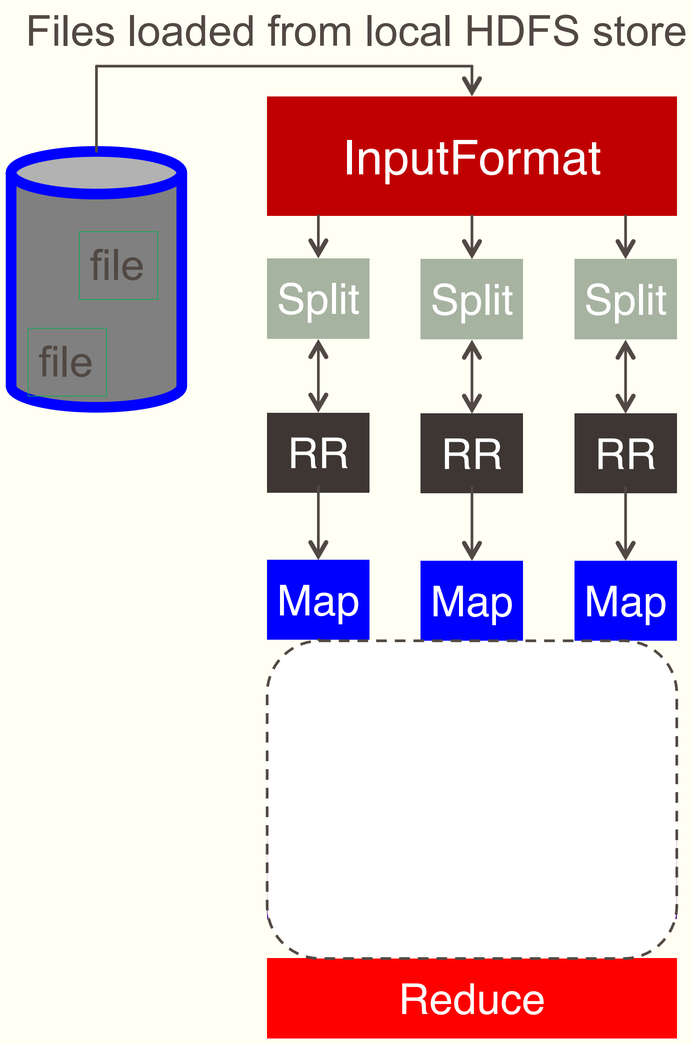
Record Reader

- The input split defines a slice of work but does not describe how to access it
- The RecordReader class actually loads data from its source and converts it into (K, V) pairs suitable for reading by Mappers
- The RecordReader is invoked repeatedly on the input until the entire split is consumed
- Each invocation of the RecordReader leads to another call of the map function defined by the programmer



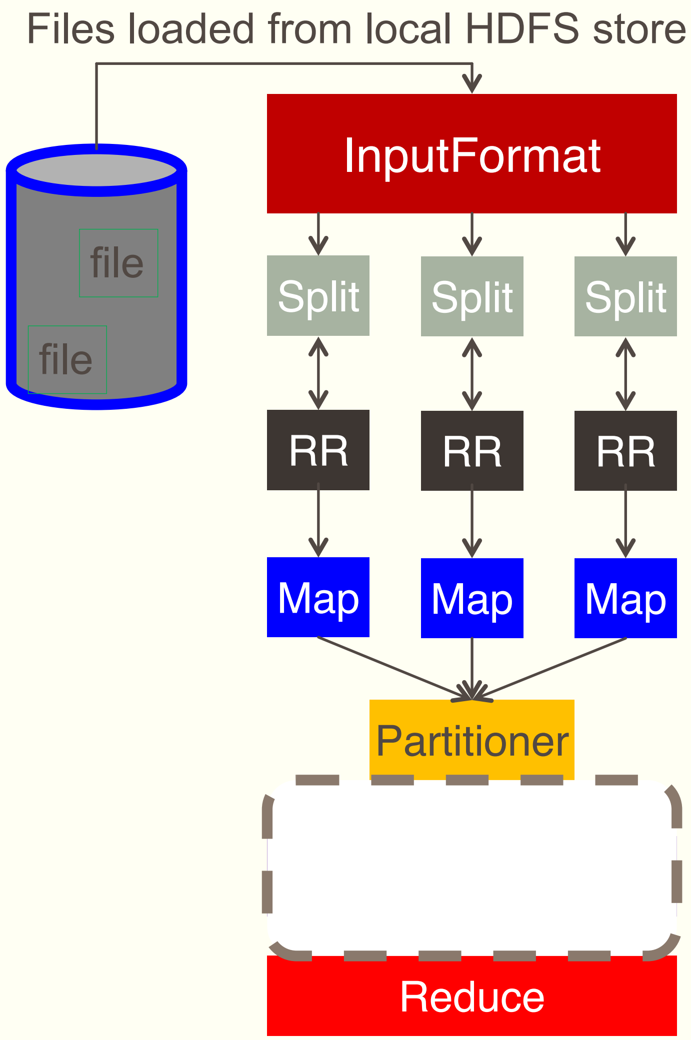
Mapper and Reducer

- The Mapper performs the user-defined work of the first phase of the MapReduce program
- A new instance of Mapper is created for each split
- The Reducer performs the user-defined work of the second phase of the MapReduce program
- A new instance of Reducer is created for each partition
- For each key in the partition assigned to a Reducer, the Reducer is called once



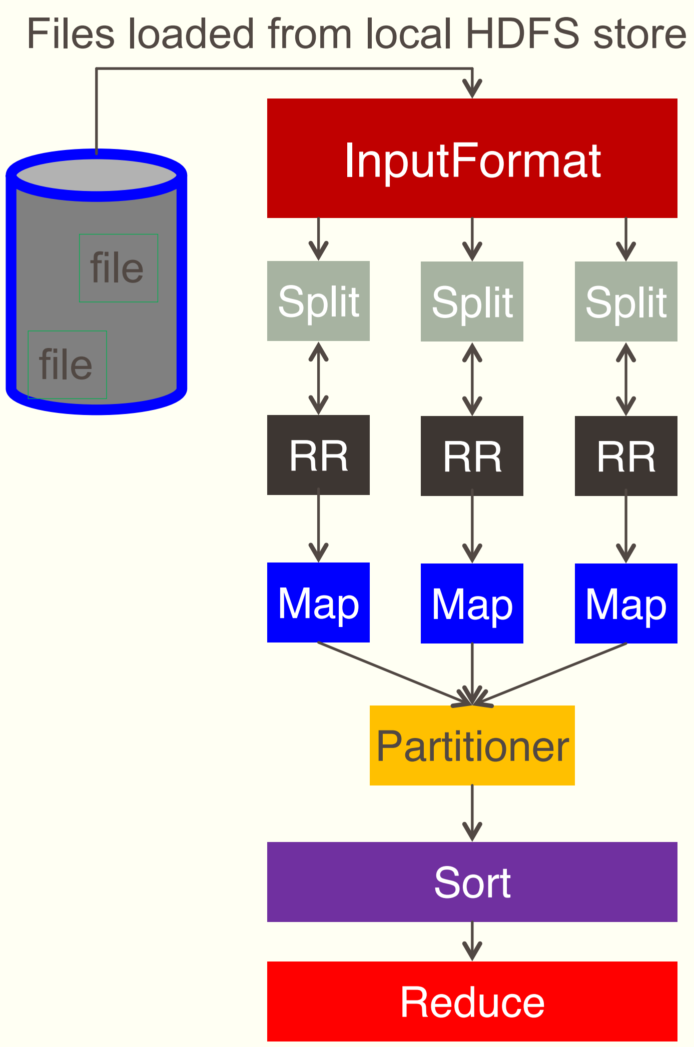
Partitioner

- Each mapper may emit (K, V) pairs to any partition
- Therefore, the map nodes must all agree on where to send different pieces of intermediate data
- The partitioner class determines which partition a given (K,V) pair will go to
- The default partitioner computes a hash value for a given key and assigns it to a partition based on this result



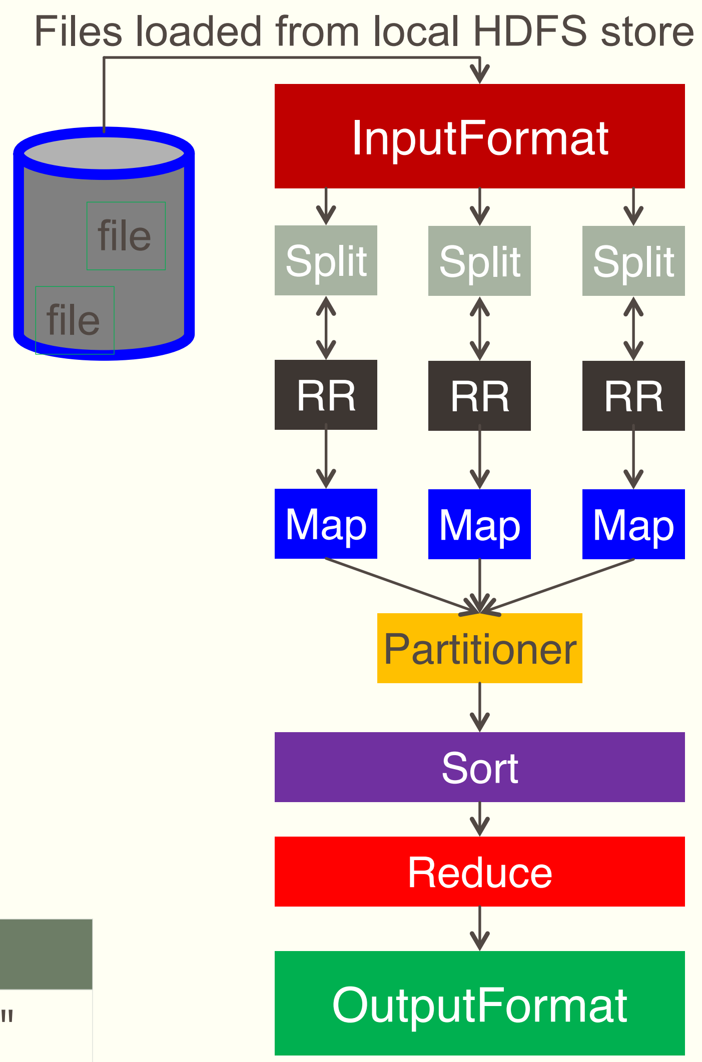
Sort

- Each Reducer is responsible for reducing the values associated with (several) intermediate keys
- The set of intermediate keys on a single node is automatically sorted by MapReduce before they are presented to the Reducer



Output Format

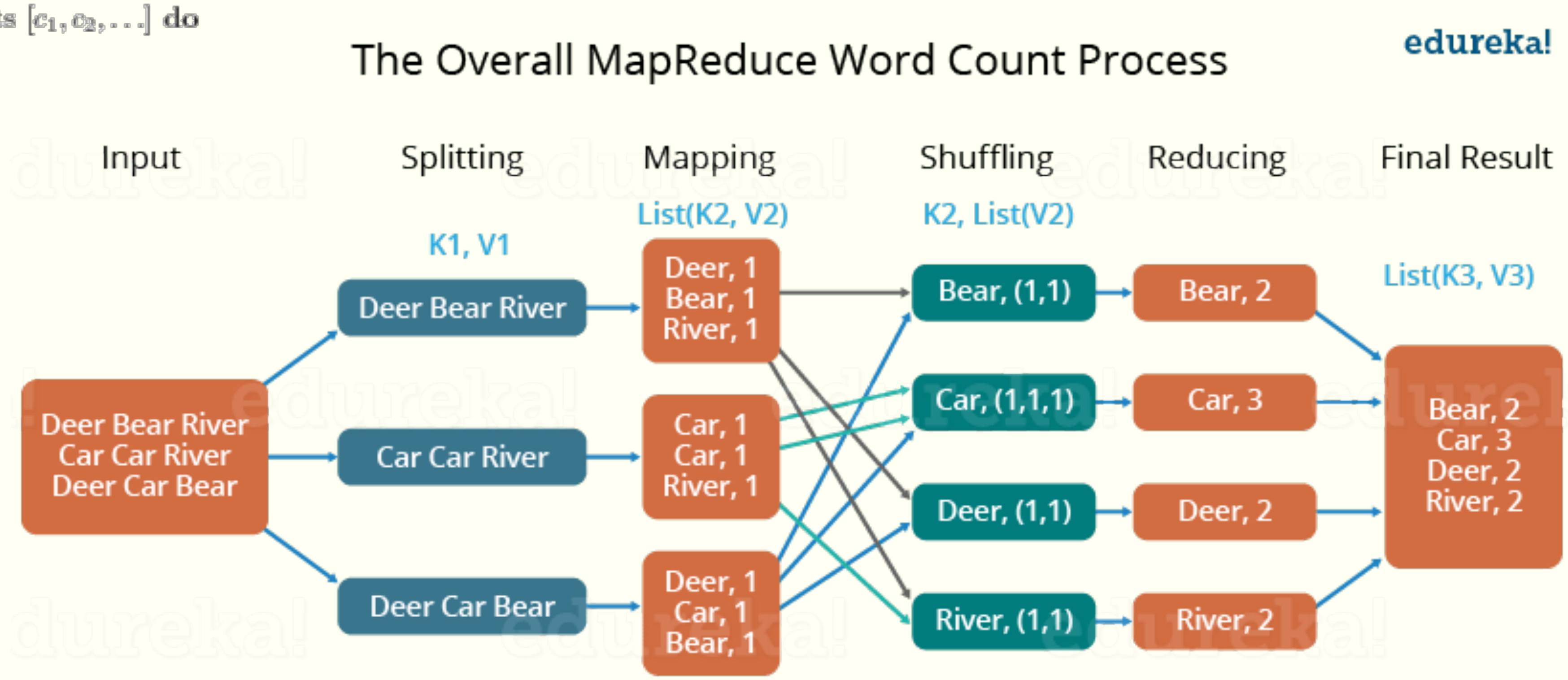
- The OutputFormat class defines the way (K,V) pairs produced by Reducers are written to output files
- The instances of OutputFormat provided by Hadoop write to files on the local disk or in HDFS
- Several OutputFormats are provided by Hadoop:



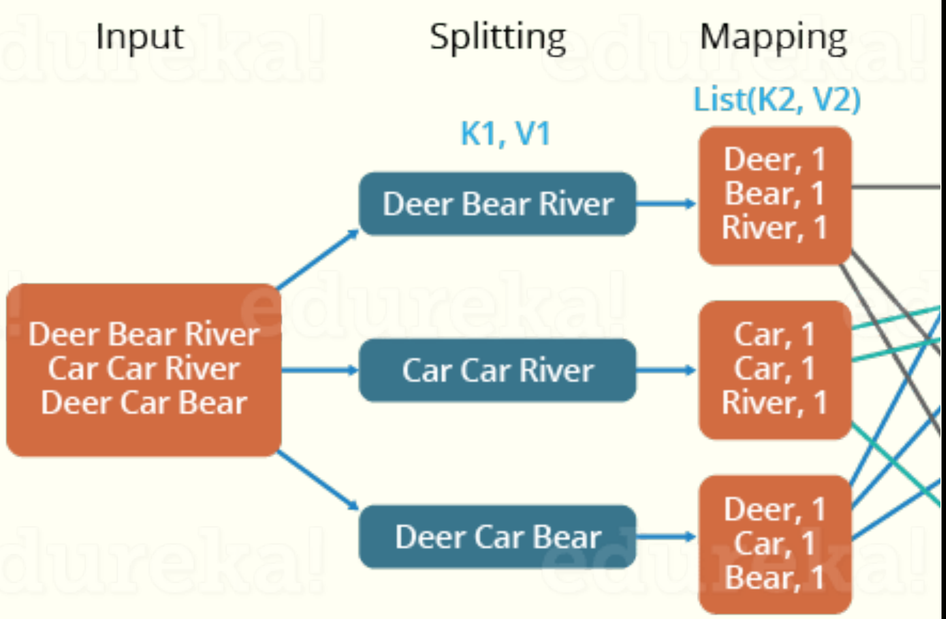
OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files

Example: Word Count

```
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 s)
```

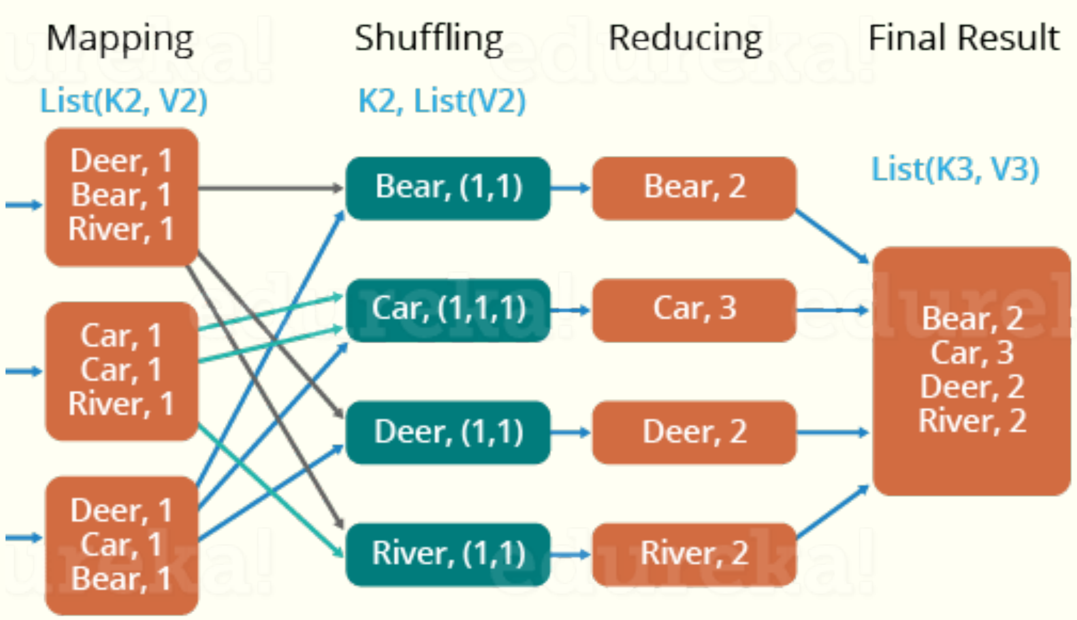


Java Implementation: Word Count




```java
import org.apache.hadoop.mapreduce.Mapper;
public class WordCountMapper extends Mapper{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    @Override
    protected void map(LongWritable key, Text value,
            Context context)
            throws IOException, InterruptedException {

        //Get the text and tokenize the word using space as separator.
        String line = value.toString();
        StringTokenizer st = new StringTokenizer(line," ");

        //For each token aka word, write a key value pair with
        //word and 1 as value to context
        while(st.hasMoreTokens()){
            word.set(st.nextToken());
            context.write(word, one);
        }
    }
}
```

WordCount: Reducer




```
import org.apache.hadoop.mapreduce.Reducer;
public class WordCountReducer extends Reducer{

   @Override
   protected void reduce(Text key, Iterable values,
         Context context)
         throws IOException, InterruptedException {

      int sum = 0;
      Iterator valuesIt = values.iterator();

      //
For each key value pair, get the value and adds to the sum
      // to get the total occurances of a word
      while(valuesIt.hasNext()){
         sum = sum + valuesIt.next().get();
      }

      // Writes the word and total occurances as
      // key-value pair to the context
      context.write(key, new IntWritable(sum));
   }
}
```

WordCount: Driver

```
import org.apache.hadoop.conf.Configured;
public class WordCount extends Configured implements Tool{

    public static void main(String[] args) throws Exception{
        int exitCode = ToolRunner.run(new WordCount(), args);
        System.exit(exitCode);
    }

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s needs two arguments, input
and output
files\n", getClass().getSimpleName());
            return -1;
        }

        // Create a new Jar and set the driver class(this class) as the
        // main class of jar
        Job job = new Job();
        job.setJarByClass(WordCount.class);
        job.setJobName("WordCounter");

        // Set the input and the output path from the arguments
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

```
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        //Set the map and reduce classes in the job
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        //Run the job and wait for its completion
        int returnValue = job.waitForCompletion(true) ? 0:1;

        if(job.isSuccessful()) {
            System.out.println("Job was successful");
        } else if(!job.isSuccessful()) {
            System.out.println("Job was not successful");

        }

        return returnValue;
    }
}
```

```
$ bin/hadoop jar wordcount.jar WordCount input output
```

MapReduce Implementation of Relational Operators

- Projection
- Selection
- Union
- Set Difference
- Join
    - Reduce-side Join
    - Map-side Join
    - In-memory Join
- Aggregation

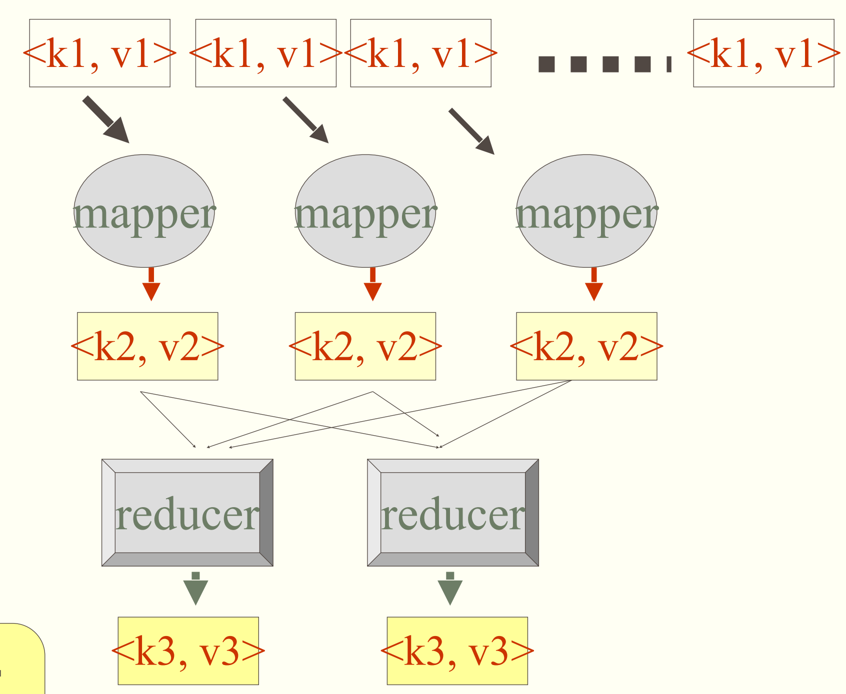
MapReduce: Projection

- Projection $\pi_A R$

not necessarily a key of R

- Input: for each tuple t in R, a pair (key, value), where value = t
- Map(key, t)
  - Emit (t.A, t.A)

Apply to each input tuple, in parallel; emit new tuples with projected attributes



- Reduce (hkey, hvalue[])
  - Emit(hkey, hkey)

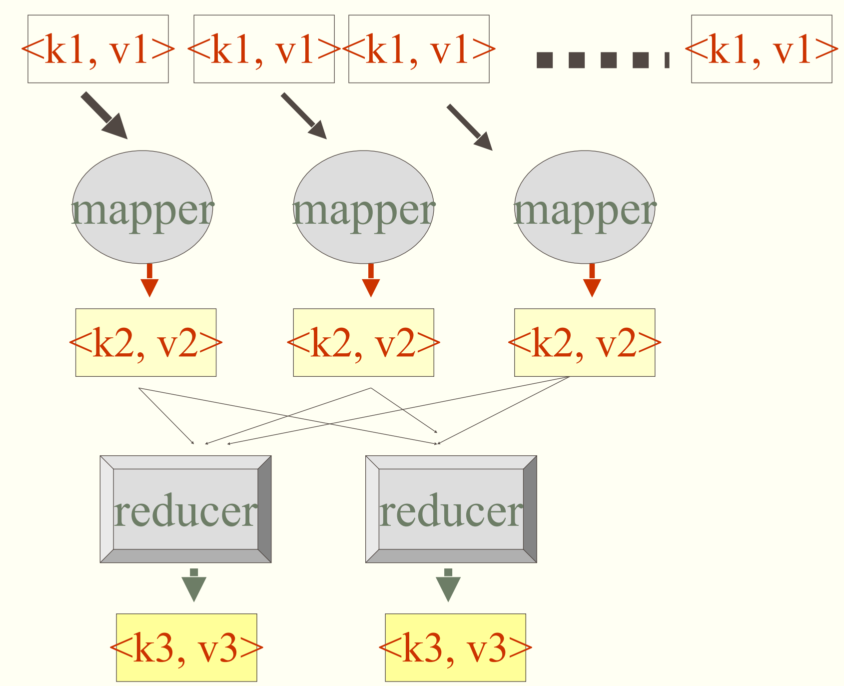
the reducer is not necessary; but it eliminates duplicates.

MapReduce: Selection

- Selection $\sigma_C R$
- Input: for each tuple t in R, a pair (key, value) where value = t
- Map (key, t)
  - If C(t), Then emit (t, “1”)

Apply to each input tuple, in parallel; select tuples that satisfy condition C

- Reduce (hkey, hvalue[])
  - emit(hkey, hkey)



MapReduce: Union

- Union $R_1 \cup R_2$

A mapper is assigned chunks from either R1 or R2

- Input: for each tuple t in R1 and s in R2, a pair (key, value)
- Map (key, t)
  - Emit (t, “1”)

A mapper just passes an input tuple to a reducer

- Reduce (hkey, hvalue[])
  - emit(hkey, hkey)

Reducers simply eliminate duplicates

MapReduce: Set Difference

- Set difference $R_1 - R_2$

distinguishable

- Input, for each tuple t in R1 and s in R2, a pair (key, value)
- Map (key, t)
  - If t is in R1, then emit (t, “1”), else emit (t, “2”)

tag each tuple with its source

- Reduce (hkey, hvalue[])
  - If only “1” appears in the list hvalue, then emit (hkey, hkey)

Reducers do the checking

Join Algorithms in MapReduce

- Reduce-side join
- Map-side join
- In-memory join
  - Striped variant
  - Memcached variant

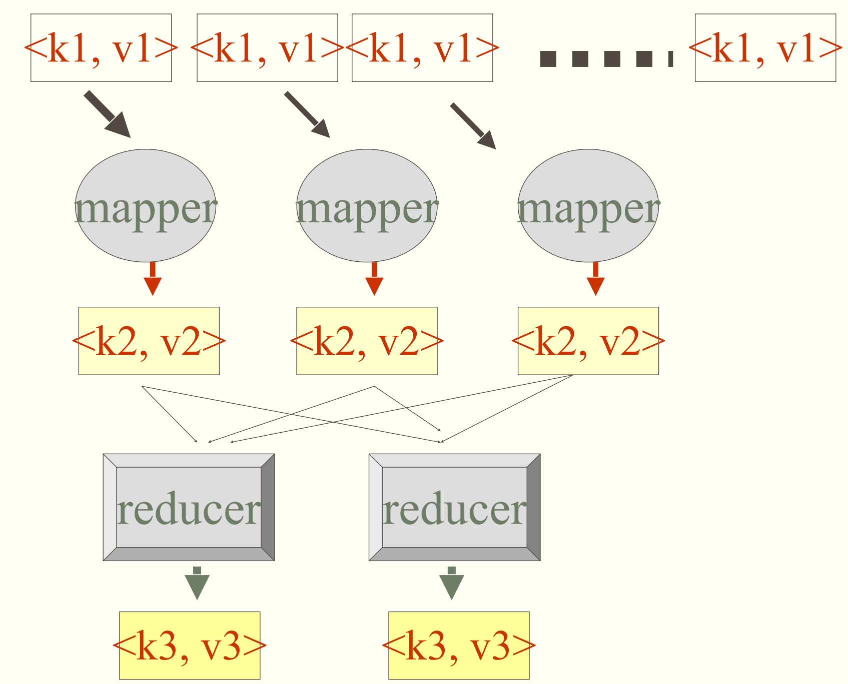
Reduce-side Join

- Natural Join: $R_1 \bowtie_{R_1.A=R_2.B} R_2$, where R1[A, C], R2[B, D]
- Input: for each tuple t in R1 and s in R2, a pair (key, value)
- Map (key, t)
  - If t is in R1
  - then emit (t.[A], (“1”, t.[C]))
  - Else emit (t.[B], (“2”, t.[D]))

Hashing on join attributes

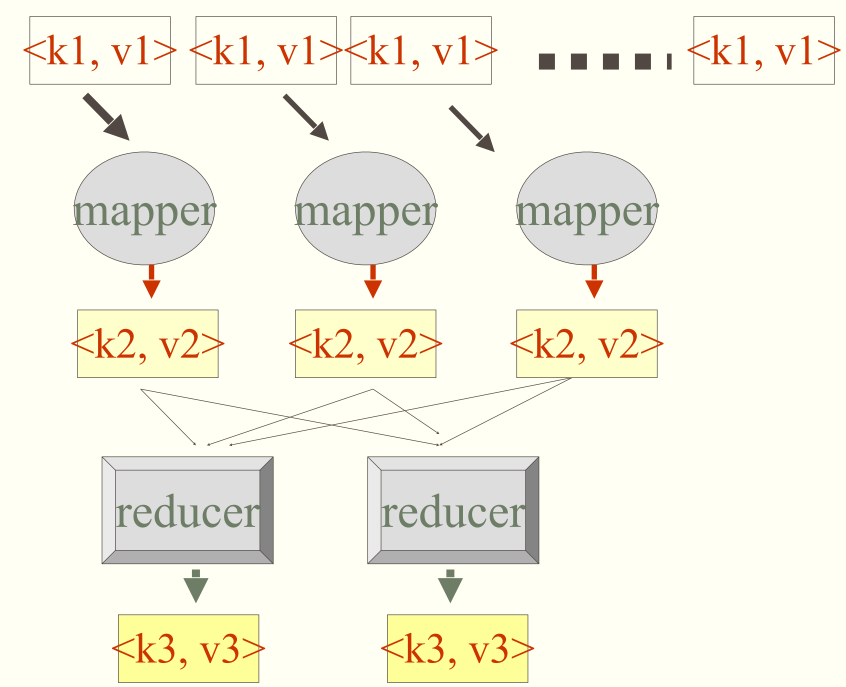
- Reduce (hkey, hvalue[])
  - For each (“1”, t[C]) and each (“2”, s[D]) in the list hvalue[]
  - Emit ((hkey, t[C], s[D]), hkey)

Nested loop



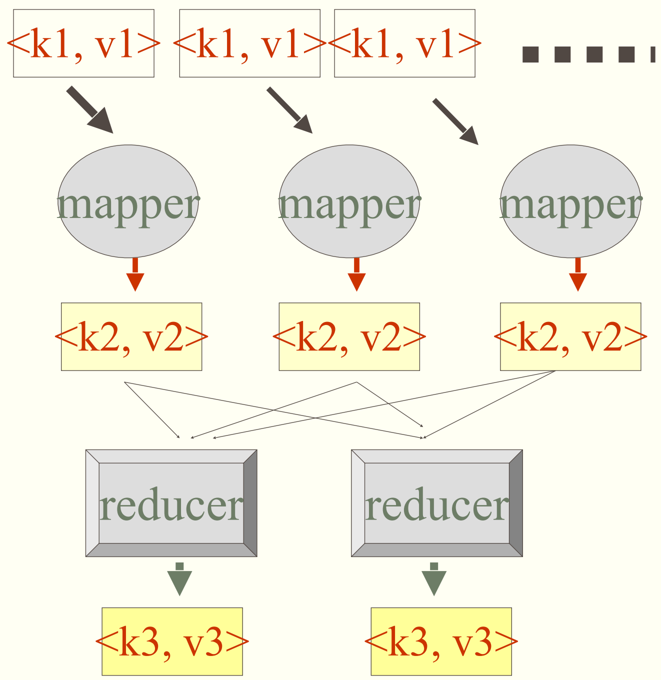
Map-Side Join

- Recall $R_1 \bowtie_{R_1.A=R_2.B} R_2$
  - Partition R1 and R2 into n partitions, by the same partitioning function (range/hash)
  - Compute $R_1^i \bowtie_{R_1.A=R_2.B} R_2^i$ locally
  - Merge the local results
- Map-side Join
  - Input relations are partitioned and sorted based on join keys
  - Map over R_1 and read from the corresponding partition of R_2
- Map(key, t)
  - Read R_2^i
  - For each tuple s in relation R_2^i
    - If t[A] = s[B] then emit ((t[A], t[C], s[D]), t[A])
- Reduce(hkey, hvalue[])
  - Emit (hkey, hkey)



In-Memory Join (Broadcast Join)

- Recall $R_1 \bowtie_{R_1.A<R_2.B} R_2$
  - Partition R1 into n partitions, by the same partitioning function (range/hash)
  - Replicate the other relation R2
  - Compute $R_1^i \bowtie_{R_1.A<R_2.B} R_2$ locally at each processor i
  - Merge the local results
- Broadcast Join
  - A smaller relation is broadcast to each node and stored in its local memory
  - The other relation is partitioned and distributed across mappers
- Map(key, t)
  - Read R_2
  - For each tuple s in relation R_2
    - If t[A] < s[B] then emit ((t[A], t[C], s[D]), t[A])
- Reduce(hkey, hvalue[])
  - Emit (hkey, hkey)



MapReduce: Aggregation

- R(A, B, C), compute sum(B) group by A
- Map (key, t)
  - Emit (t[A], t[B])

Grouping: done by MapReduce framework

- Reduce (hkey, hvalue[])
  - Sum = 0
  - For each value s in the list hvalue[]
    - Sum = Sum + 1
  - Emit (hkey, Sum)