NoSQL Systems

MapReduce (advanced topics)

Vinu Venugopal
ScaDS.ai Lab, IIIT Bangalore

- How a MapReduce job interacts with a Distributed File System.
- Optimizations for Improving Performance of MapReduce Jobs.
- Various Design Patterns for Text Processing in MapReduce Framework

Design of MapReduce

- The **programmer has little impact** on:
  - **WHERE** a mapper or reducer runs.
  - **WHEN** a mapper or reducer starts or finishes.
  - **WHICH** input key-value pairs are processed by a specific mapper.
  - **WHICH** input key-value pairs are processed by a specific reducer.
- But this is also good: MapReduce automatically takes care of all these issues for you!

Design of MapReduce

Techniques for controlling execution and managing the flow of data

- The ability to **construct complex data structures as keys and values** to store and communicate partial results.
- The ability to execute **user-specified initialization code** at the **beginning** of a map or reduce task, and the ability to execute user-specified termination code at the **end of a map or reduce task**.

Design of MapReduce

Techniques for controlling execution and managing the flow of data

- The ability to **construct complex data structures as keys and values** to store and communicate partial results.
- The ability to execute **user-specified initialization code** at the **beginning** of a map or reduce task, and the ability to execute user-specified termination code at the **end of a map or reduce task**.

Code that runs before the main map or reduce tasks, say setup()

- Useful for loading external data
- Establish DB connectivity for status updates
- Initialize data structures used during the task

Code that runs after the main map or reduce task has completed

- Closing DB connection/releasing resources
- Perform final aggregation on collected data

Design of MapReduce

Techniques for controlling execution and managing the flow of data

- The ability to **construct complex data structures as keys and values** to store and communicate partial results.
- The ability to execute **user-specified initialization code** at the **beginning** of a map or reduce task, and the ability to execute user-specified termination code at the **end of a map or reduce task**.
- The ability to **preserve state in both mappers and reducers** across multiple input or intermediate keys.
- The ability to **control the sort order of intermediate keys**, and therefore the order in which a reducer will encounter particular keys.
  - Process keys in **descending order** or any **custom-defined order**.
  - Implement Secondary Sorting
  - Ensure that the most important data is processed **first** or in a specific sequence.

Design of MapReduce

Techniques for controlling execution and managing the flow of data

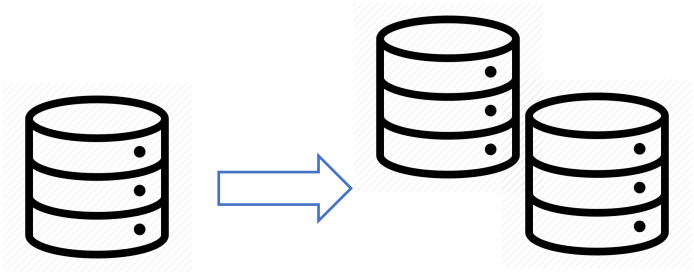
- The ability to **construct complex data structures as keys and values** to store and communicate partial results.
- The ability to execute **user-specified initialization code** at the **beginning** of a map or reduce task, and the ability to execute user-specified termination code at the **end of a map or reduce task**.
- The ability to **preserve state in both mappers and reducers** across multiple input or intermediate keys.
- The ability to **control the sort order of intermediate keys**, and therefore the order in which a reducer will encounter particular keys.
- The ability to **control the partitioning of the key space**, and therefore the set of keys that will be encountered by a particular reducer.

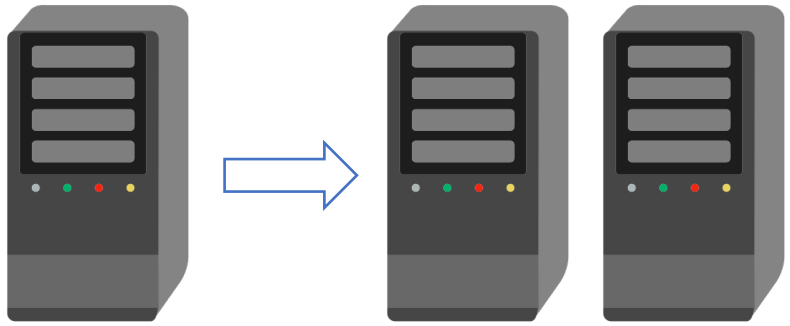
Scalability & Efficiency

- **Scalability**—ensuring that there are **no** inherent **bottlenecks** as algorithms are applied to increasingly **larger datasets**
- **Efficiency**—ensuring that algorithms **do not needlessly consume resources** and thereby reducing the cost of parallelization
- Two very principal aspects of **linear scalability**:
  - an algorithm running on twice the amount of data should take only twice as long.
  - an algorithm running on twice the number of nodes should only take half as long

Scalability & Efficiency

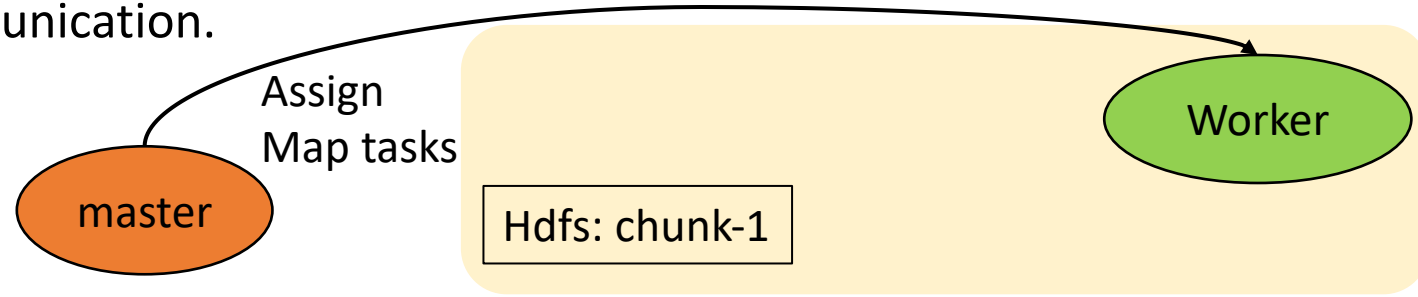
- **Scalability**—ensuring that there are **no** inherent **bottlenecks** as algorithms are applied to increasingly **larger datasets**
- **Efficiency**—ensuring that algorithms **do not needlessly consume resources** and thereby reducing the cost of parallelization
- Two very principal aspects of **linear scalability**:
  - an algorithm running on twice the amount of data should take only twice as long.
  - an algorithm running on twice the number of nodes should only take half as long





Data Splits & Record Readers

- By default, MapReduce will try to assign **Map tasks** to those worker nodes that also **hold the corresponding chunks of input data** in order to reduce network communication.



- That is, a Map task of a **worker node can directly read a data chunk** (usually 64 MB) from the part of the HDFS that this worker node manages.

Data Splits & Record Readers

```
public static void main(String[] args) throws Exception {

    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "wordcount");

    job.setMapperClass(TokenizerMapper.class);
    // job.setCombinerClass(IntSumReducer.class); // enable to use 'local aggregation'
    job.setReducerClass(IntSumReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

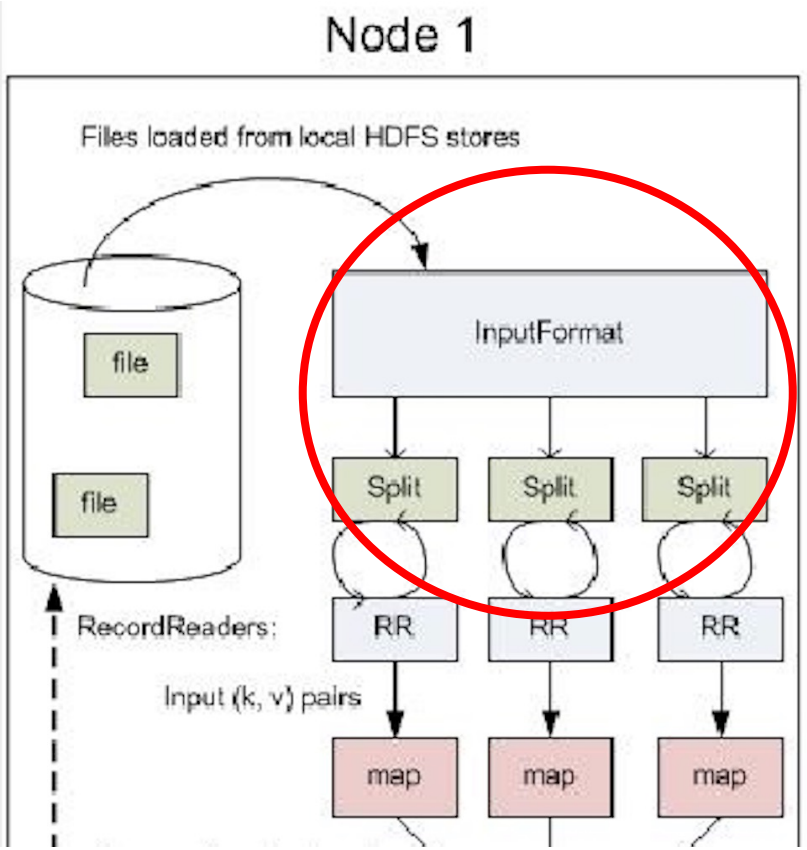
Reading from a data chunk

- This is taken care of by the InputFormat class that is registered in the driver function of the MapReduce job.
- This class in turn creates a RecordReader class which **creates the individual key-value pairs** that serve as input for the Map function.

InputFormat

Hadoop relies on the InputFormat of the job to do three things:

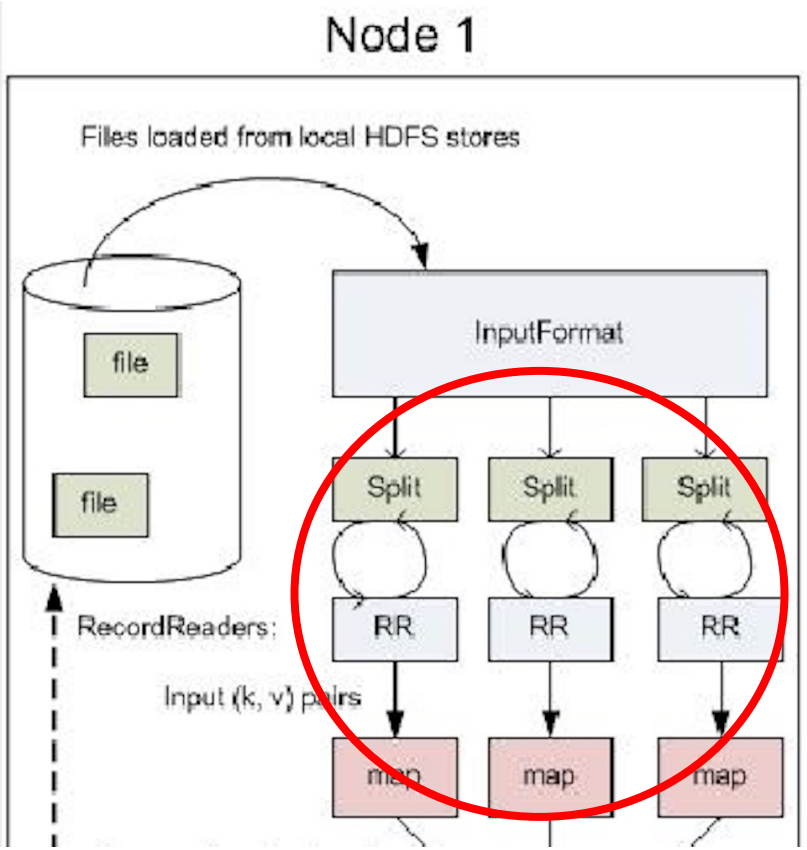
1. **Validate the input configuration** for the job (i.e., checking that the data is there).
2. **Split the input** blocks and files into **logical chunks** of type InputSplit, each of which is assigned to a map task for processing.
3. **Create the** RecordReader implementation to be used to **create key/value pairs from the raw** InputSplit. These pairs are **sent one by one to their mapper**.



Source: http://developer.yahoo.com/hadoop/tutorial/

RecordReader

- A RecordReader **uses the data within the logical boundaries** created by the input split to **generate key/value pairs**.
- **Start** and **End** denotes the logical boundaries
- **Start**: is the byte position in the file where the RecordReader should start generating key/value pairs.
- **End**: is where it should stop reading records.
- These are **not hard boundaries as far as the API** is concerned—there is nothing stopping from reading the entire file for each map task.
- While reading the entire file is not advised, **reading outside of the boundaries it often necessary** to ensure that a complete record is generated.



Source: http://developer.yahoo.com/hadoop/tutorial/

Customized Split Sizes & Record Readers

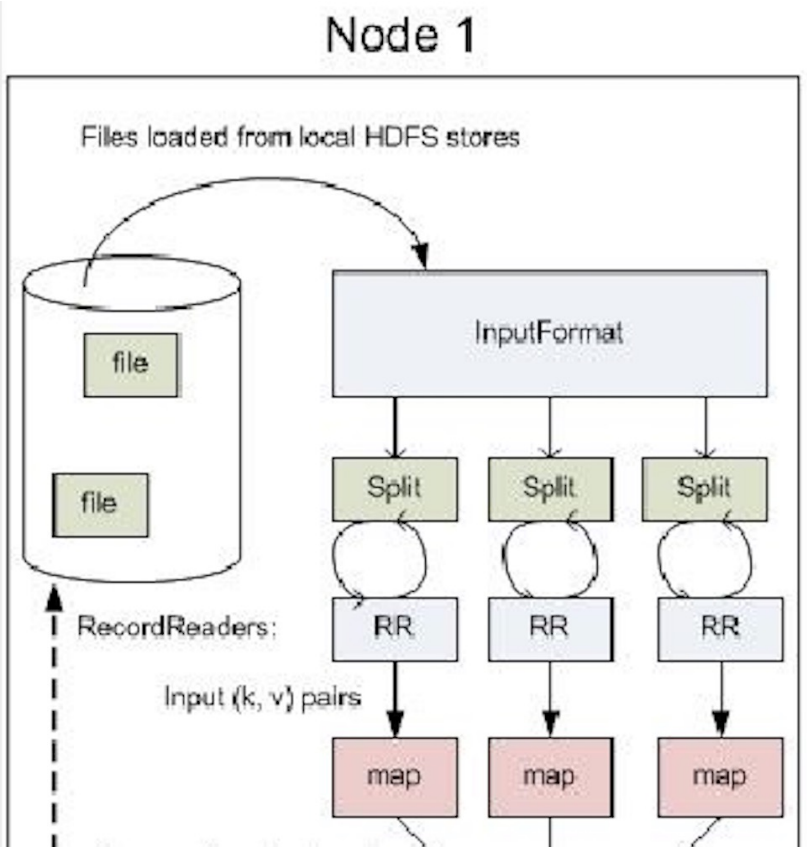
- The Hadoop MapReduce framework **spawns one map task for each InputSplit** generated by the InputFormat for the job.

- For **TextInputFormat** the default split size is one chunk/block, and for **LineRecordReader** the default record size is one line of the file.

• The **default key-value pair** passed to the mapper by **TextInputFormat** with **LineRecordReader** is:

Key: The **byte offset** of the line in the file (e.g., 0, 50, 100, etc.).

Value: The actual **line of text**.



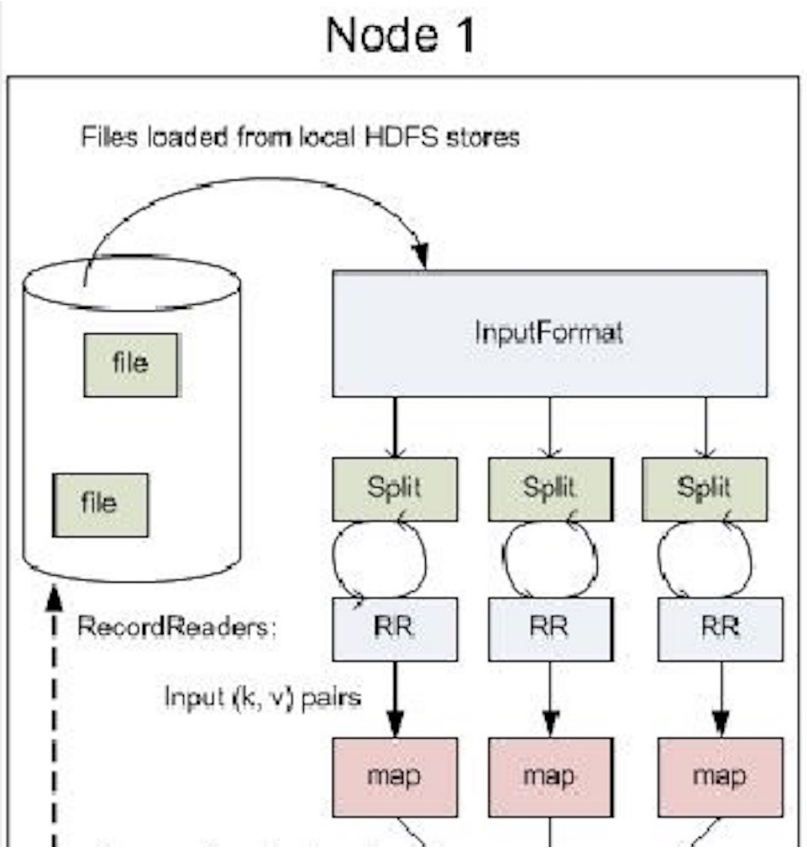
Source: http://developer.yahoo.com/hadoop/tutorial/
https://examples.javacodegeeks.com/enterprise-java/apache-hadoop/apache-hadoop-recordreader-example/

Customized Split Sizes & Record Readers

Both **input splits** and **record sizes** can be **customized** to handle different types of data:

Splits: You can adjust the split size by setting properties like mapreduce.input.fileinputformat.split.maxsize and mapreduce.input.fileinputformat.split.minsize.

Record Sizes: You can create **custom InputFormats** and **RecordReaders** to process complex input formats like **database records**, **log messages**, **JSON/XML**, or even **binary data**.



- See
  - CustomWordCount.java
  - CustomFileInputFormat.java
  - CustomLineRecordReader.java

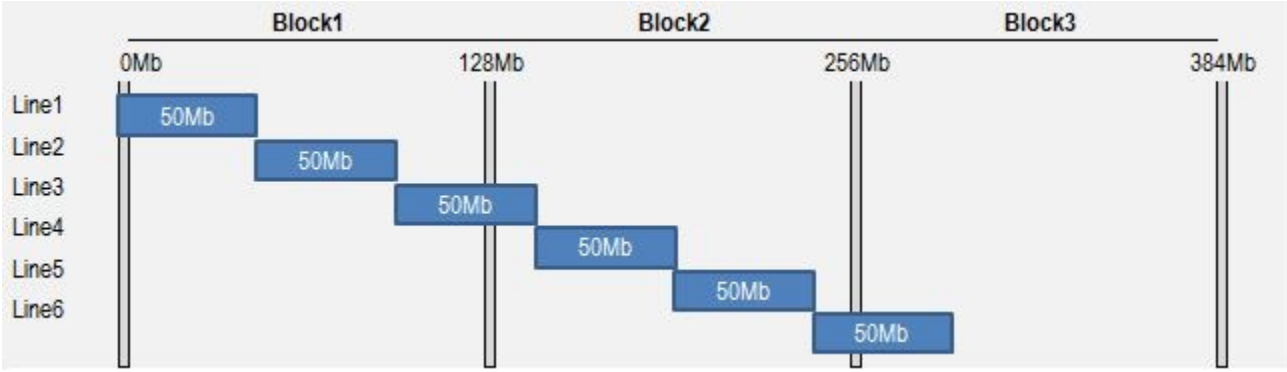
Source: http://developer.yahoo.com/hadoop/tutorial/
https://examples.javacodegeeks.com/enterprise-java/apache-hadoop/apache-hadoop-recordreader-example/

LineRecordReader

Dataset: 300MB file, spanned over 3 different blocks (blocks of 128Mb), and 1 InputSplit for each block.

3 scenarios

Scenario 1: File is composed on 6 lines of 50Mb each



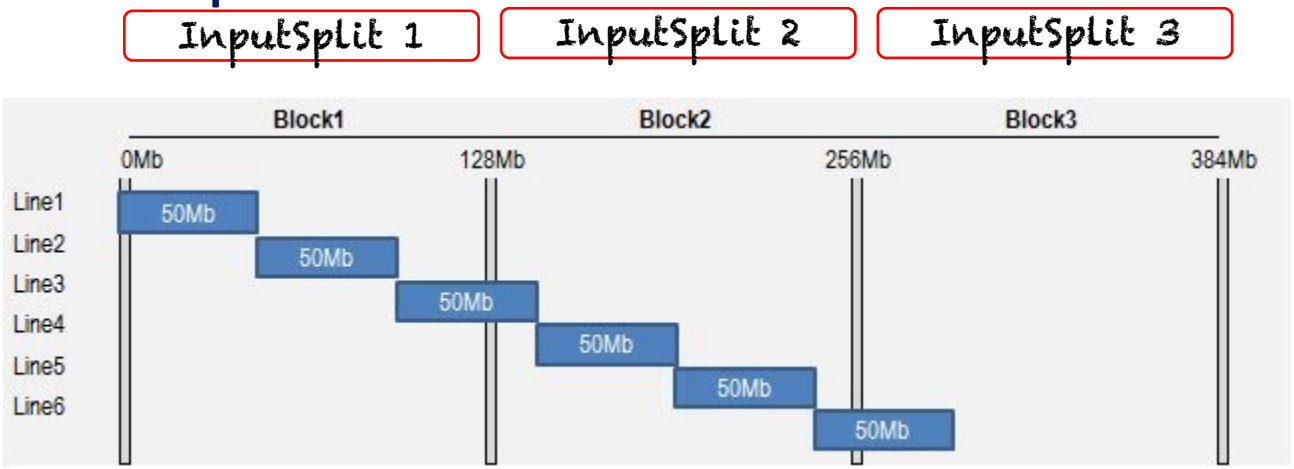
Source: https://hadoopi.wordpress.com

LineRecordReader

Dataset: 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and 1 InputSplit for each block.

3 scenarios

Scenario 1: File is composed on 6 lines of 50Mb each



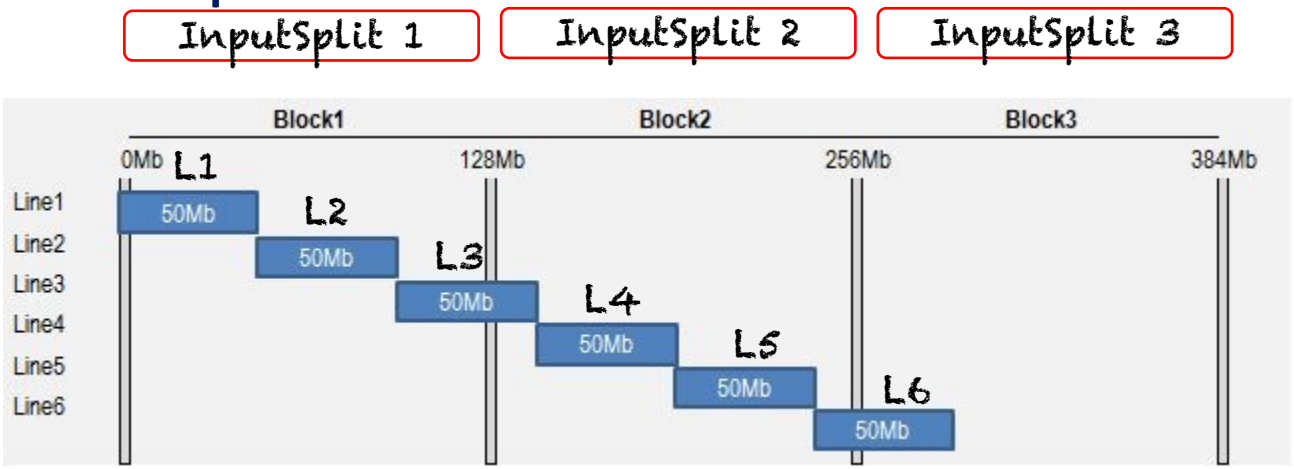
LineRecordReader

Dataset: 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and 1 InputSplit for each block.

3 scenarios

M1 M2 M3

Scenario 1: File is composed on 6 lines of 50Mb each



RReader 1: starts from block B1, position 0; at 50Mb and 100Mb sees 2 EOL; Read L1 and L2 *locally*; From 100Mb reach the end of Split without finding EOL; Read L3 *remotely* (by the mean of **FSDataInputStream**) from B2 until it finds an EOL

LineRecordReader

Dataset: 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and 1 InputSplit for each block.

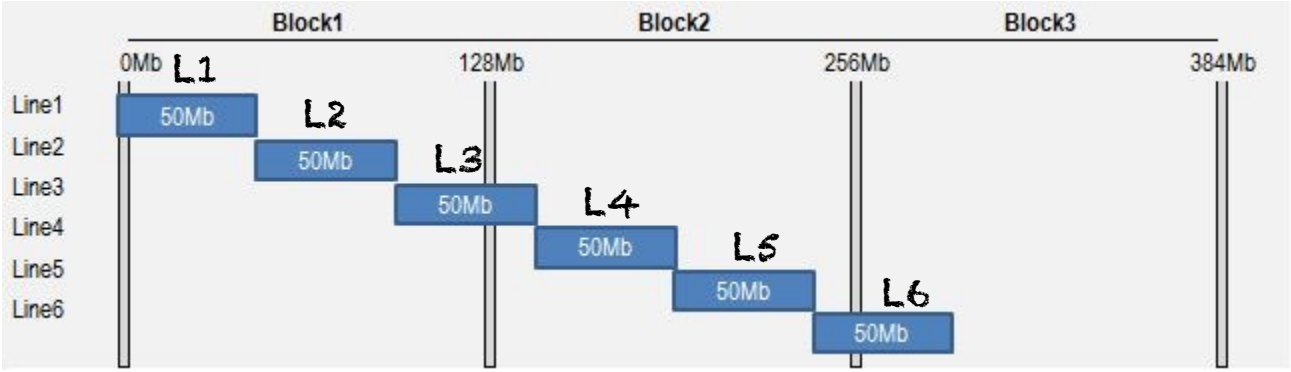
3 scenarios

M1 M2 M3

LRR1 LRR2 LRR3

Scenario 1: File is composed on 6 lines of 50Mb each

InputSplit 1 InputSplit 2 InputSplit 3



RReader 1: starts from block B1, position 0; at 50Mb and 100Mb sees 2 EOL; Read L1 and L2 *locally*; From 100Mb reach the end of Split without finding EOL; Read L3 *remotely* (by the mean of **FSDataInputStream**) from B2 until it finds an EOL

LineRecordReader

Dataset: 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and 1 InputSplit for each block.

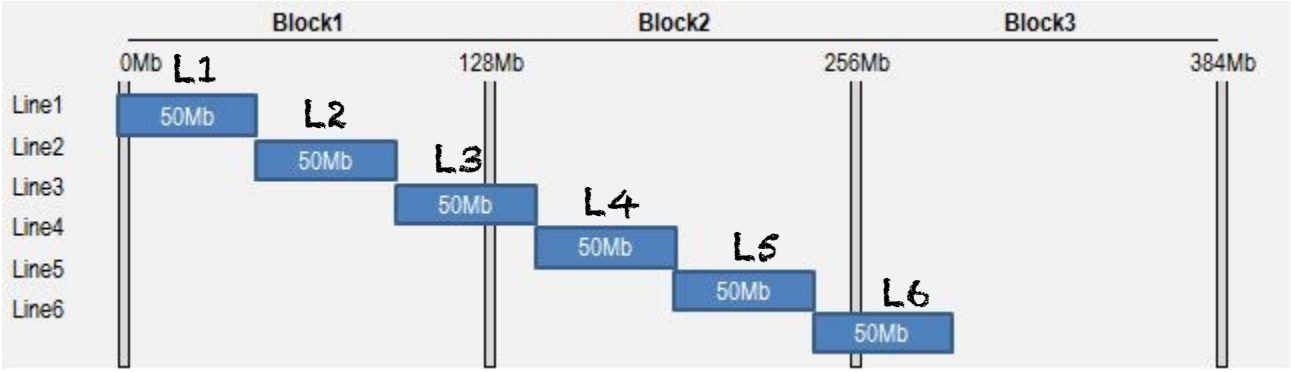
3 scenarios

M1 M2 M3

LRR1 LRR2 LRR3

Scenario 1: File is composed on 6 lines of 50Mb each

InputSplit 1 InputSplit 2 InputSplit 3



RReader 1: starts from block B1, position 0; at 50Mb and 100Mb sees 2 EOL; Read L1 and L2 *locally*; From 100Mb reach the end of Split without finding EOL; Read L3 *remotely* (by the mean of **FSDataInputStream**) from B2 until it finds an EOL

LineRecordReader

Dataset: 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and 1 InputSplit for each block.

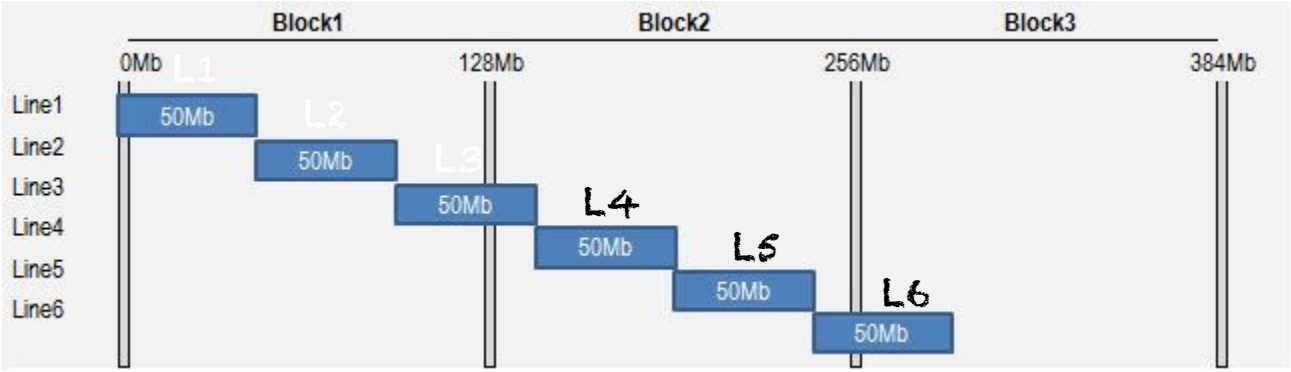
3 scenarios

M1 M2 M3

LRR1 LRR2 LRR3

Scenario 1: File is composed on 6 lines of 50Mb each

InputSplit 1 InputSplit 2 InputSplit 3



RReader 1: starts from block B1, position 0; at 50Mb and 100Mb sees 2 EOL; Read L1 and L2 *locally*; From 100Mb reach the end of Split without finding EOL; Read L3 *remotely* (by the mean of **FSDataInputStream**) from B2 until it finds an EOL

RReader2: will start from B2, at position 128Mb; skip the first record by jumping out to the next EOL; Similarly, read L4, L5 (locally) and finally L6 (remotely)

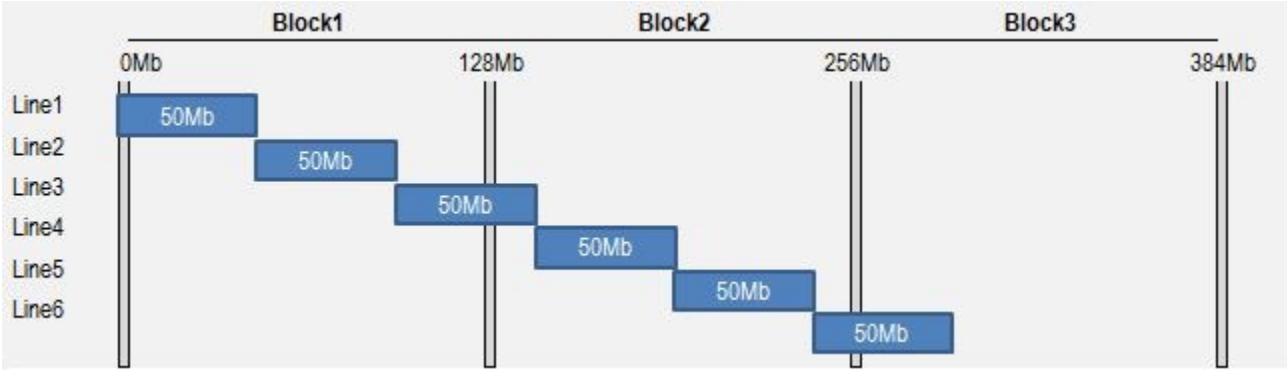
LineRecordReader

Dataset: 300Mb file, spanned over 3 different blocks (blocks of 128Mb), and 1 InputSplit for each block.

3 scenarios

	Start	Actual Start	End	Line(s)
Mapper1	B1:0	B1:0	B2:150	L1,L2,L3
Mapper2	B2:128	B2:150	B3:300	L4,L5,L6
Mapper3	B3:256	B3:300	B3:300	N/A

Scenario 1: File is composed on 6 lines of 50Mb each

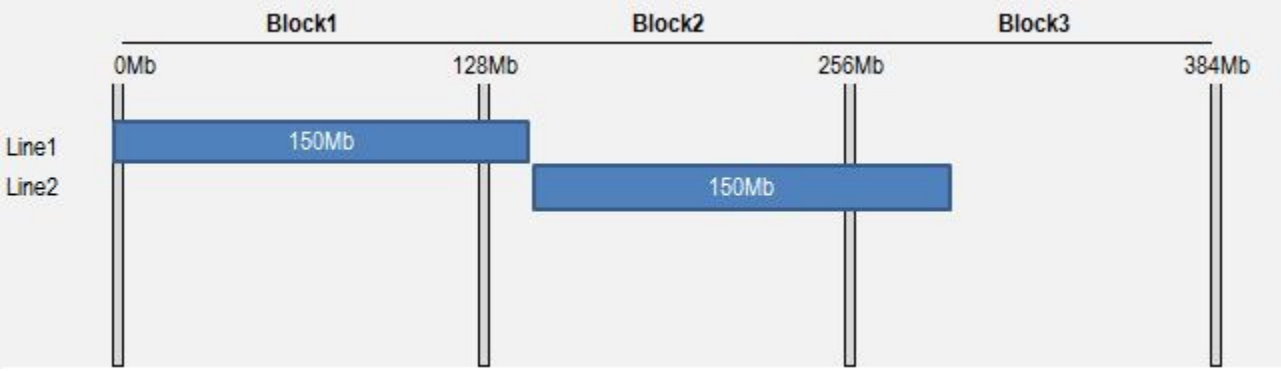


RReader 1: starts from block B1, position 0; at 50Mb and 100Mb sees 2 EOL; Read L1 and L2 *locally*; From 100Mb reach the end of Split without finding EOL; Read L3 *remotely* (by the mean of **FSDataInputStream**) from B2 until it finds an EOL

RReader2: will start from B2, at position 128Mb; skip the first record by jumping out to the next EOL; Similarly, read L4, L5 (locally) and finally L6 (remotely)

LineRecordReader

Scenario2: File composed on 2 lines of 150Mb each



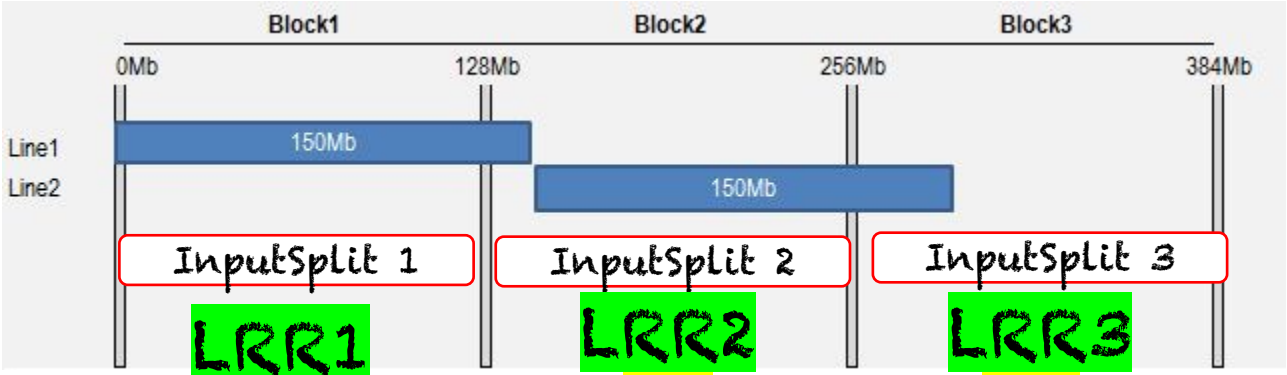
RReader 1: will start reading from block B1, position 0. It will read line L1 *locally* until end of its split (128Mb), and will then continue reading *remotely* on B2 until EOL (150Mb)

RReader 2: will not start reading from 128Mb, but from 150Mb, and until B3:300

	Start	Actual Start	End	Line(s)
Mapper1	B1:0	B1:0	B2:150	L1
Mapper2	B2:128	B2:150	B3:300	L2
Mapper3	B3:256	B3:300	B3:300	N/A

LineRecordReader

Scenario2: File composed on 2 lines of 150Mb each



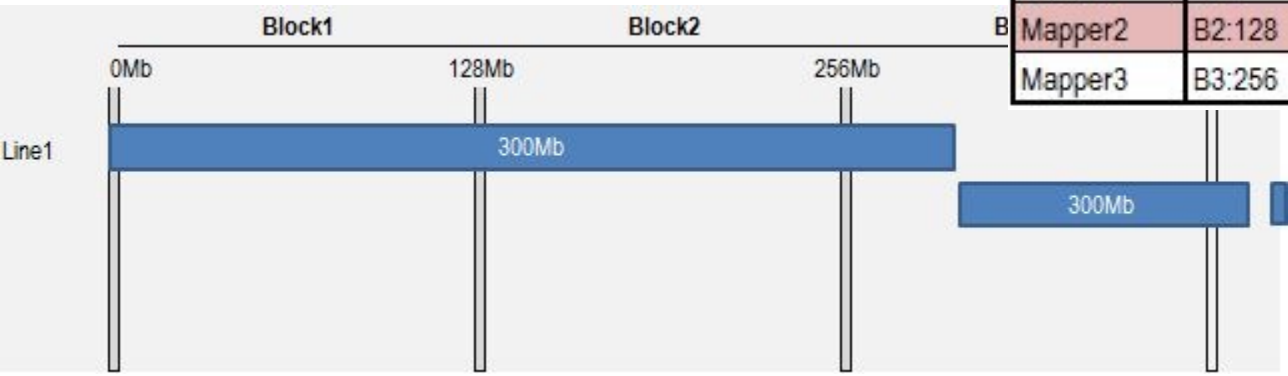
RReader 1: will start reading from block B1 position 0. It will read line L1 *locally* until end of its split (128Mb), and will then continue reading *remotely* on B2 until EOL (150Mb)

RReader 2: will not start reading from 128Mb, but from 150Mb, and until B3:300

	Start	Actual Start	End	Line(s)
Mapper1	B1:0	B1:0	B2:150	L1
Mapper2	B2:128	B2:150	B3:300	L2
Mapper3	B3:256	B3:300	B3:300	N/A

LineRecordReader

Scenario 3: File composed on 2 lines of 300Mb each



	Start	Actual Start	End	Line(s)
Mapper1	B1:0	B1:0	B3:300	L1
Mapper2	B2:128	B3:300	B2:256	N/A
Mapper3	B3:256	B3:300	B5:600	L2

RReader 1: will start reading *locally* from B1:0 until B1:128, then *remotely* all bytes available on B2, and finally *remotely* on B3 until EOL is reached (300Mb).

There is here **some overhead** as we're trying to **read a lot of data** that is **not locally available**

RReader 2: will start reading from B2:128 and will jump out to next available record located at B3:300. Its **new start** position (B3:300) is actually **greater than its maximum position** (B2:256). This reader will therefore **not provide Mapper 2 with any key / value.**

RReader 3: will start reading from B3:300 to B5:600

How Many Maps?

- The total number of map tasks will be equal to sum of **number of input splits** per file.

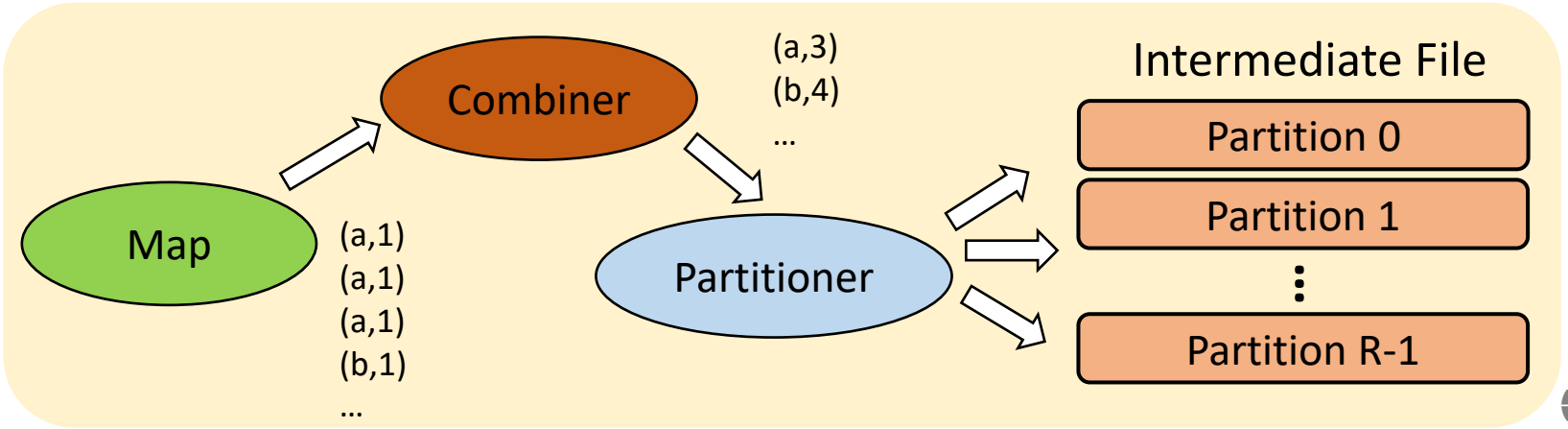
- The right level of parallelism for maps seems to be around 10-100 maps per-node.

Combiners & Partitioners

- **Combiners** are an **additional optimization** to **perform a "local" aggregation** before the key/value pairs are emitted through the network.

  Examples:

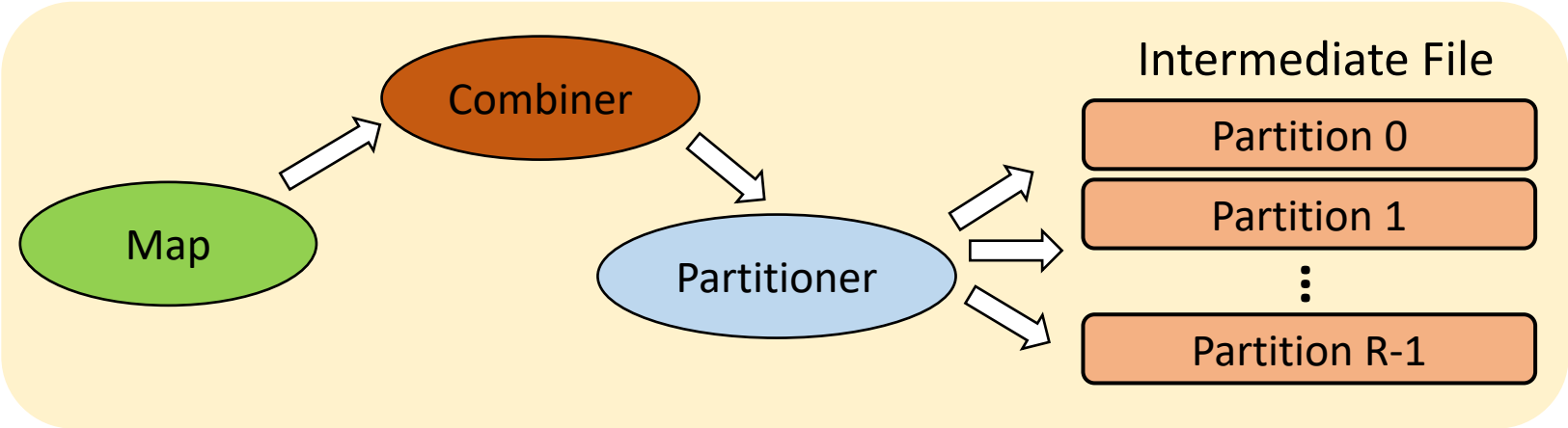
  - Compute the **partial sums of integers** at each mapper.
  - Compute the **partial counts of word occurrences** at each mapper.
- Users can optionally specify a combiner, via Job.setCombinerClass(Class)



Combiners & Partitioners

2nd optimization:

- **Partitioners** need to ensure that the workload of all reducers is balanced (i.e., the partitioner is the counterpart of the scheduler, which needs to ensure that the workload of all mappers is balanced).
  - Simple hash-partitioning function: hash(key) % #worker_nodes



How Many Reduces?

- The number of reduces for the job is set by the user via Job.setNumReduceTasks(int)

- Increasing the number of reducers, **increases the framework overhead**, but **increases load balancing and lowers the cost of failures**.

More Issues: Scheduling

- **Scheduling** (Map and Reduce tasks)
  - MapReduce jobs are divided into smaller units called **tasks** (e.g., map task may be responsible for processing an InputSplit).
  - In large jobs, the total number of **tasks may exceed** the number of tasks that can be **run on the cluster concurrently** – queue the tasks **assigned to nodes as they become available**.
  - A **dedicated task scheduler** has to ensure that all Mappers are balanced.

More Issues: Synchronization

- **Synchronization** (among Mappers & Reducers)
  - Reducers can only start once all Mappers are finished.
  - A few long-running **stragglers** may delay the overall MapReduce execution time substantially.
- **Stragglers** are often caused by circumstances that are local to the worker node on which the straggler task is running.
  - Overload on worker machine due to scheduler.
  - Frequent (but recoverable) disk errors.
- Solution:
  - Abort stragglers when MapReduce computation is near its end (progress monitored by a master node).
  - For each aborted straggler, schedule **backup** (replacement) **task** on another worker node.
- Can significantly improve overall completion time.

More Issues

- **Fault-tolerance** (failure management of tasks)
  - In large clusters, disk failures are common and RAM experiences more errors than one might expect
  - **Backup tasks** are started to replace the original task when the MapReduce job is close to finishing.

Note on Sharing Data

- A static **DistributedCache** (Job.addCacheFile() in Hadoop 3.3.1) object allows for sharing larger data volumes (via file- based access) among all instances of Mapper and Reducer classes.
- DistributedCache **provides a container** for an entire set of files that are to be shipped to the Mappers and Reducers.
- The framework will **copy the necessary files to the worker node before** any tasks for the job are executed on that node.
- Its efficiency stems from the fact that the **files are only copied once per job** and the ability to cache archives which are un-archived on the workers.
- Example: WordCount2.java, available on Moodle for download.

Useful Design Patterns for Text Processing with MapReduce*

1. Local Aggregation

Implement your own Combiner function to pre-aggregate values that belong to the same key – to minimize the amount of partial results that need to be copied across the network.

Useful Design Patterns for Text Processing with MapReduce*

1. Local Aggregation

Implement your own Combiner function to pre-aggregate values that belong to the same key.

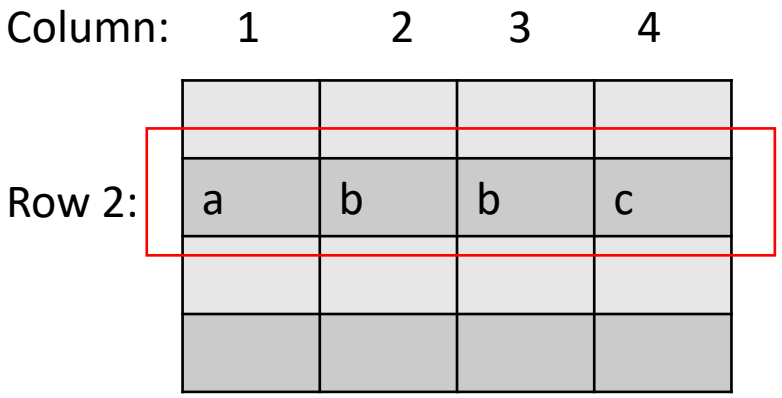
2. Pairs and Stripes

Two common design patterns useful in building *word co-occurrence matrices* on large text corpora.

Pairs

Key, value

((2,1), a)
((2,2), b)
((2,3), b)
((2,4), c)



Useful Design Patterns for Text Processing with MapReduce*

1. Local Aggregation

Implement your own Combiner function to pre-aggregate values that belong to the same key.

2. Pairs and Stripes

Two common design patterns useful in building *word co-occurrence matrices* on large text corpora.

3. Order Inversion

An algorithm may require data in some fixed order: by controlling *how keys are sorted* and *how the key space is partitioned*, we can present data to the reducer in the order necessary to perform the proper computations. This reduces the number of partial results that the reducer needs to hold in memory.

Useful Design Patterns for Text Processing with MapReduce*

1. Local Aggregation

Implement your own Combiner function to pre-aggregate values that belong to the same key.

2. Pairs and Stripes

Two common design patterns useful in building *word co-occurrence matrices* on large text corpora.

3. Order Inversion

An algorithm may require data in some fixed order: by controlling *how keys are sorted* and *how the key space is partitioned*, we can present data to the reducer in the order necessary to perform the proper computations. This reduces the number of partial results that the reducer needs to hold in memory.

4. Secondary Sorting (a.k.a. “value-to-key conversion” design pattern”)

What if in addition to sorting by key, we also need to sort by value?

Useful Design Patterns for Text Processing with MapReduce*

1. Local Aggregation

Implement your own Combiner function to pre-aggregate values that belong to the same key.

2. Pairs and Stripes

Two common design patterns useful in building *word co-occurrence matrices* on large text corpora.

3. Order Inversion

An algorithm may require data in some fixed order: by controlling *how keys are sorted* and *how the key space is partitioned*, we can present data to the reducer in the order necessary to perform the proper computations. This reduces the number of partial results that the reducer needs to hold in memory.

4. Secondary Sorting (a.k.a. "value-to-key conversion" design pattern")

What if in addition to sorting by key, we also need to sort by value?

5. Relational Joins

Map-side and *Reduce-side* join techniques.

Useful Design Patterns for Text Processing with MapReduce*

Other relevant topics:

- **Inverted Indexing** Using MapReduce to create an inverted index from a collection of documents. (Read by yourself)

*Following [2]: "*Data-Intensive Text Processing with MapReduce.*"

{d1, d2, d3,….}

Inverted Index:

a → [d1, d3,..]
b → [d5, d8,..]
c → [d1, d9,..]

	a	b	c	…
d1	**1**	**0**	**1**	**…**
d2	0	0	0	…
d3	1	0	0	…
⋮	⋮	⋮	⋮	

1. Local Aggregation

- In Hadoop, **intermediate results are written to local disk** before being sent over the network. Since **network and disk latencies are relatively expensive** compared to other operations, **reductions in the amount of intermediate data** translate into increases in **algorithmic efficiency**.

1. Local Aggregation

- In Hadoop, **intermediate results are written to local disk** before being sent over the network. Since **network and disk latencies are relatively expensive** compared to other operations, **reductions in the amount of intermediate data** translate into increases in **algorithmic efficiency**.

- Various techniques for local aggregation: COMBINERS and IN-MAPPER COMBINING

1. Local Aggregation

- In Hadoop, **intermediate results are written to local disk** before being sent over the network. Since **network and disk latencies are relatively expensive** compared to other operations, **reductions in the amount of intermediate data** translate into increases in **algorithmic efficiency**.

Techique-1

- Writing a **custom combiner function** allows for preprocessing the output of a Mapper before the final key-value pairs are emitted.
- Combiner serves just like a "**Mini-Reducer**" that operates over the intermediate output of each Mapper.
- **Reduction in the # of intermediate key-value pairs** that need to be shuffled across the network—from the order of total number of terms in the collection to the order of the **number of unique terms in the collection**.
- Combiners in Hadoop are **treated as optional optimizations**, so there is **no guarantee** that the execution framework will take advantage of all opportunities for partial aggregation.

1. Local Aggregation

Techique-2 (~ same effect as Techique-1)

- An **associative array is introduced** inside the mapper to tally up term counts within a single document
- Instead of emitting a key-value pair for each term in the document, this version emits a **key-value pair for each unique term in the document**.

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

Basic word count algo in MR

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

1. Local Aggregation

Techique-3 (taken one step further)

- An **associative array is introduced** inside the mapper to tally up term counts within a single document
- Instead of emitting a key-value pair for each term in the document, this version **emits a key-value pair for each unique term in the document**.

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

- Prior to processing any input key-value pairs, the mapper's INITIALIZE method is called – API hook for user-specified code.
- To accumulate partial term counts in the associative array across multiple documents, initialize an associative array outside the Map method – preserving state across multiple calls to Map Method
- Note that the emission of intermediate data is deferred until the CLOSE method in the pseudo-code.

Local Aggregation

- With Techniques 2 and 3, we are in essence incorporating combiner functionality directly inside the mapper -- no need to run a separate combiner.
- Hence named ”in-mapper combining”
- In-mapper combining typically are more efficient than using actual combiners – combiners don’t reduce the no of key-value pairs that are emitted by the mappers implying unnecessary object creation and destruction (garbage collection), and other serialization and deserialization overheads.

Local Aggregation

Drawbacks of ”in-mapper combining”

- In-mapper combining pattern **violates the functional programming principles** that underpin MapReduce -- **preserves state** across multiple input key-value pairs within the same map task.

1. In functional programming, **operations are expected to be stateless**, meaning that **each input key-value pair is processed independently**.

2. By maintaining state across inputs, the behavior of the algorithm can **become dependent on the order** in which key-value pairs are processed. This **order-dependence** can lead to subtle bugs that are particularly **hard to detect and debug** when working with large datasets.

Local Aggregation

Drawbacks of ”in-mapper combining”

- There is a fundamental **scalability bottleneck** associated with the in-mapper combining pattern.

1. 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.

Local Aggregation

Drawbacks of ”in-mapper combining”

- There is a fundamental **scalability bottleneck** associated with the in-mapper combining pattern.

1. 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.

2. 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**. That is: instead of emitting intermediate data only after every key-value pair has been processed, **emit partial results after processing every n key-value pairs**.

Local Aggregation

Algorithmic correctness with local aggregation

- 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).

- if the reduce operation is both **commutative** and **associative**, you can use the **same function** for both the combiner and the reducer without modification. (Scenarios where neither the order nor the grouping affects the outcome.)

Local Aggregation

Algorithmic correctness with local aggregation

- 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).

- if the reduce operation is both **commutative** and **associative**, you can use the **same function** for both the combiner and the reducer without modification. (Scenarios where neither the order nor the grouping affects the outcome.)

  - **Calculating averages** isn't associative. You can't just sum partial averages to get the final average. Mean(1,2,3,4,5) ≠ Mean(Mean(1,2), Mean(3,4,5))

  - **String concatenation** is not commutative. The order in which strings are combined matters.

*taking into account the shuffle and sort; not exact match

Compute the mean of values with same key

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Compute the mean of values with same key

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

How can we properly take advantage of combiners?

Compute the mean of values with same key

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Combiner
2:     method Combine(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))          ▷ Separate sum and count

1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Compute the mean of values with same key

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)
```

This algorithm will not work!!

```
1: class Combiner
2:     method Combine(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))          ▷ Separate sum and count
```

```
1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Combiners must have the same input and output key-value type.

Also, the mapper output type and the reducer input type must match

Compute the mean of values with same key

- Instead of emitting a single aggregate value, we can emit **pairs of partial sums and counts** in the Mapper and Combiner classes.
- The final *Mean* value is computed in the **last step** of the Reducer class.
- Exercise: Translate the Combiner class into Hadoop using *configure* and *close* methods.

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))

1: class Combiner
2:     method Combine(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))

1: class Reducer
2:     method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

2. Pairs and Stripes

- Let's continue the idea of **emitting complex objects** (e.g., pairs of partial sums and counts in the previous example) to compare two common design patterns for processing large collections of documents.
- Suppose we would like to create a large **co-occurrence matrix** of words that occur together at a subsequent position in a document.
- For ***n* words**, this obviously results in a large ***n x n* matrix**.

$$X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

2. Pairs and Stripes

- Let's continue the idea of **emitting complex objects** (e.g., pairs of partial sums and counts in the previous example) to compare two common design patterns for processing large collections of documents.
- Suppose we would like to create a large **co-occurrence matrix** of words that occur together at a subsequent position in a document.
- For ***n* words**, this obviously results in a large ***n x n* matrix**.

$$X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

- The **Pairs algorithm** detects co-occurring words in each document and emits a count of 1 for each pair.
- The **Stripes algorithm** emits, for each individual word, an entire array (i.e., "stripe") of co-occurring words together with their local counts.

The Pairs Algorithm

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence
1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                             ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

- The Mapper employs a **nested loop** to create pairs of neighboring words, which are each emitted with a count of 1.
- The Reducer is identical to the earlier WordCount example, except that it obtains word pairs as keys.

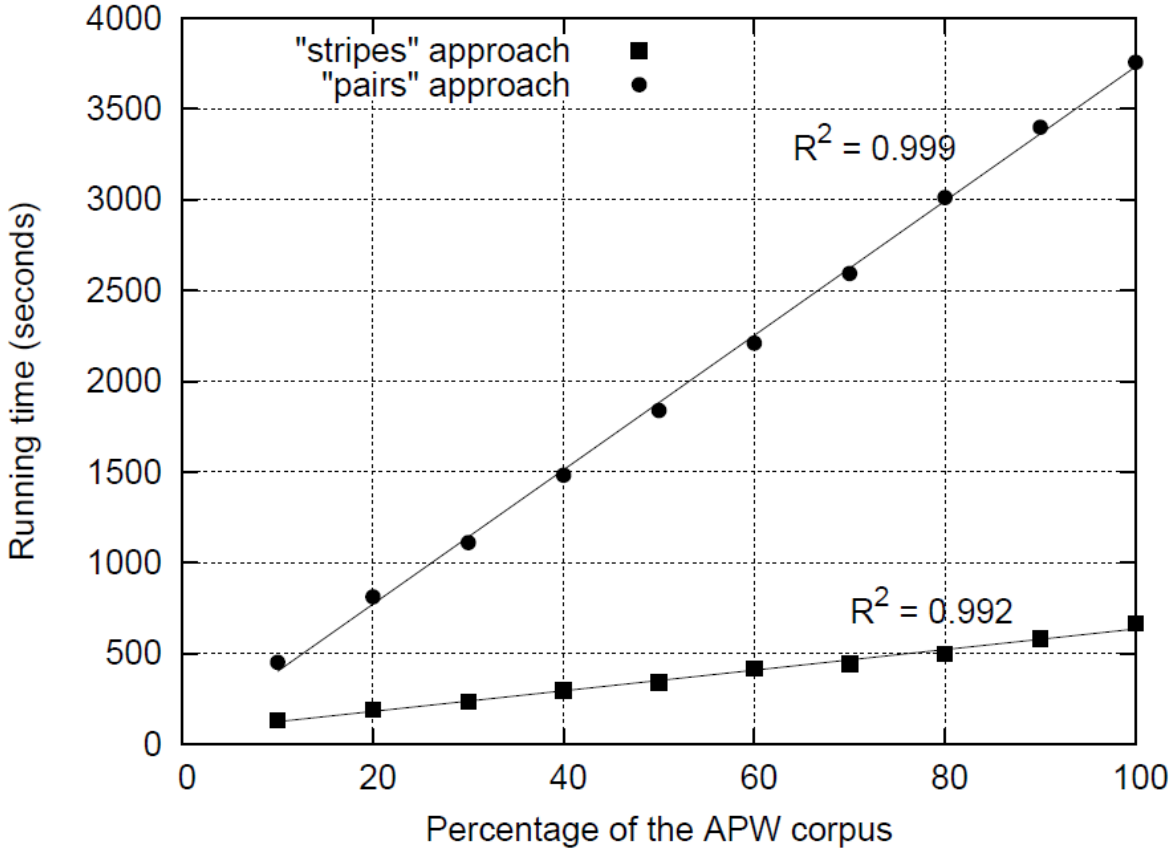
The Stripes Algorithm

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1          ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H_1, H_2, H_3, ...])
3:         H_f ← new AssociativeArray
4:         for all stripe H ∈ stripes [H_1, H_2, H_3, ...] do
5:             Sum(H_f, H)                  ▷ Element-wise sum
6:         Emit(term w, stripe H_f)
```

- The **Mapper pre-aggregates the local counts** of each neighbor into an **associative array** and emits these stripes.
- The Reducer combines the partial counts into global counts. The results are the stripes of our matrix.

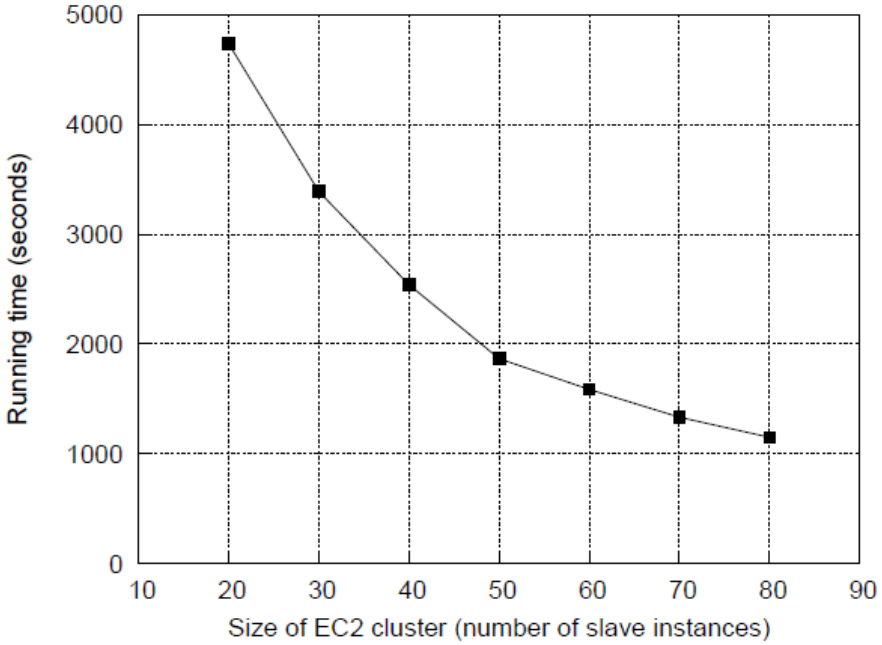
Performance Comparison

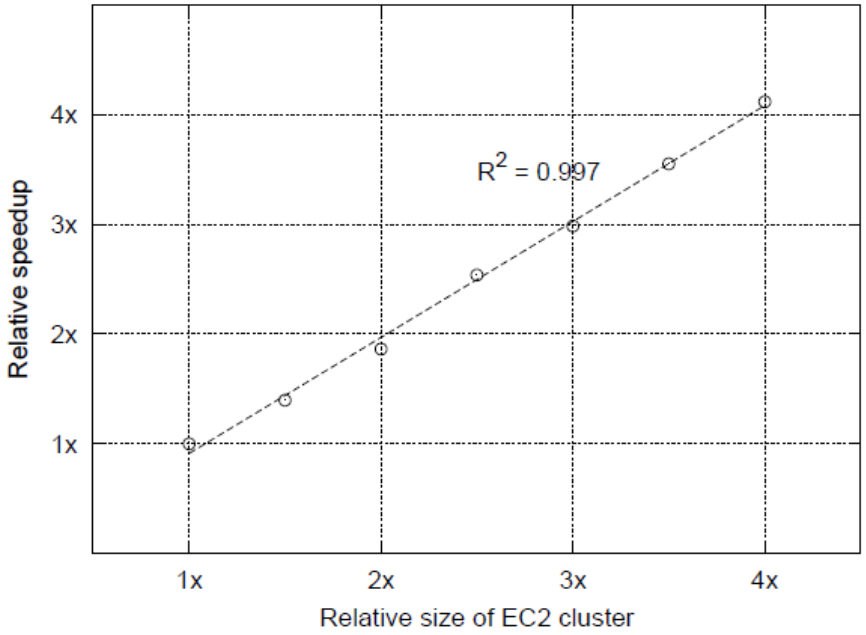


Both plots show a (nearly) perfect linear scale-out for larger data sizes.

- Comparison of the Pairs and Stripes algorithms on the Associated Press Worldstream (APW) **corpus with 2.27 million documents** and a size of 5.57 GB, performed on a **19-node Hadoop cluster**.

Performance Comparison





- The Stripes algorithm on the entire Associated Press Worldstream (APW) corpus over an increasing number of nodes using Amazon's EC2 compute cluster.
- Again, a nearly perfect linear scale-out for more nodes.

3. Secondary Sorting

- MapReduce **sorts intermediate key-value pairs by the keys** during the shuffle and sort phase, which is very convenient if computations inside the reducer rely on sort order
- What if in **addition to sorting by key**, we also **need to sort by value**?
- Example:
  - Sensor data from a scientific experiment: there are m sensors each taking readings on continuous basis, where m is potentially a large number -- where r_x after each timestamp t_x represents the actual sensor readings

$$
\begin{aligned}
&(t_1, m_1, r_{80521}) \\
&(t_1, m_2, r_{14209}) \\
&(t_1, m_3, r_{76042}) \\
&\ldots \\
&(t_2, m_1, r_{21823}) \\
&(t_2, m_2, r_{66508}) \\
&(t_2, m_3, r_{98347})
\end{aligned}
$$

3. Secondary Sorting

- MapReduce **sorts intermediate key-value pairs by the keys** during the shuffle and sort phase, which is very convenient if computations inside the reducer rely on sort order
- What if in **addition to sorting by key**, we also **need to sort by value**?
- Example:
  - Sensor data from a scientific experiment: there are m sensors each taking readings on continuous basis, where m is potentially a large number -- where r_x after each timestamp t_x represents the actual sensor readings
  - Suppose we wish to reconstruct the activity at each individual sensor over time.

$$m_1 \rightarrow (t_1, r_{80521})$$

  - This would bring all readings from the same sensor together in the reducer.
  - Makes no guarantees about the ordering of values associated with the same key, the sensor readings will not likely be in temporal order.

3. Secondary Sorting

- MapReduce **sorts intermediate key-value pairs by the keys** during the shuffle and sort phase, which is very convenient if computations inside the reducer rely on sort order
- What if in **addition to sorting by key**, we also **need to sort by value**?

  Example: map(docId, doc) → [(term, (docId, count))]
- **Google's MapReduce has a built-in functionality** for setting a secondary sorting condition; Hadoop does not have this.
- Solution: Use **composite keys** plus a **custom Partitioner**.

  map(docId, doc) → [((term, docId), count)]

Custom Partitioner in Hadoop

Step 1: Define your own Partitioner class and override the *getPartition()* method:

```
public static class TermPartitioner  extends Partitioner<Text, Text> {
   @Override
   public int getPartition(Text key, Text value, int numReduceTasks)
   {
            String [] termDocIdPair = key.toString().split("\t");
            String term = termDocIdPair[0];
            return term.hashCode() % numReduceTasks;
    }
}
```

Step 2: Register your new Partitioner class in the driver:

```
job.setPartitionerClass(TermPartitioner.class);
```

4. Relational Joins

- Suppose we have two input relations S and T:

S	k_i, a_i, b_i	T	k_i, c_i, d_i
	k_1, a_1, b_1		k_1, c_1, d_1
	k_2, a_2, b_2		k_1, c_2, d_2
	k_3, a_3, b_3		

- k_i denotes the **join key** (shared among both relations), s_i and t_i are **identifiers of the tuples**, and ***S*$_i$ and *T*$_i$ are other attributes** of *S* and *T*.

- Obviously, the goal is to bring together (i.e., "join") tuples from *S* and *T* that share the same join key.

Reduce-Side Joins

- Map emits the **join key as the intermediate key** and the remaining tuple as the intermediate value (for both relations *S* and *T*).
- Reduce obtains all tuples belonging to the same join key and **combines all tuples from *S* with the tuples from *T*** via a **nested loop**.

$k_1 \rightarrow (s_1, S_1)$
$k_2 \rightarrow (s_2, S_2)$
$k_3 \rightarrow (s_3, S_3)$

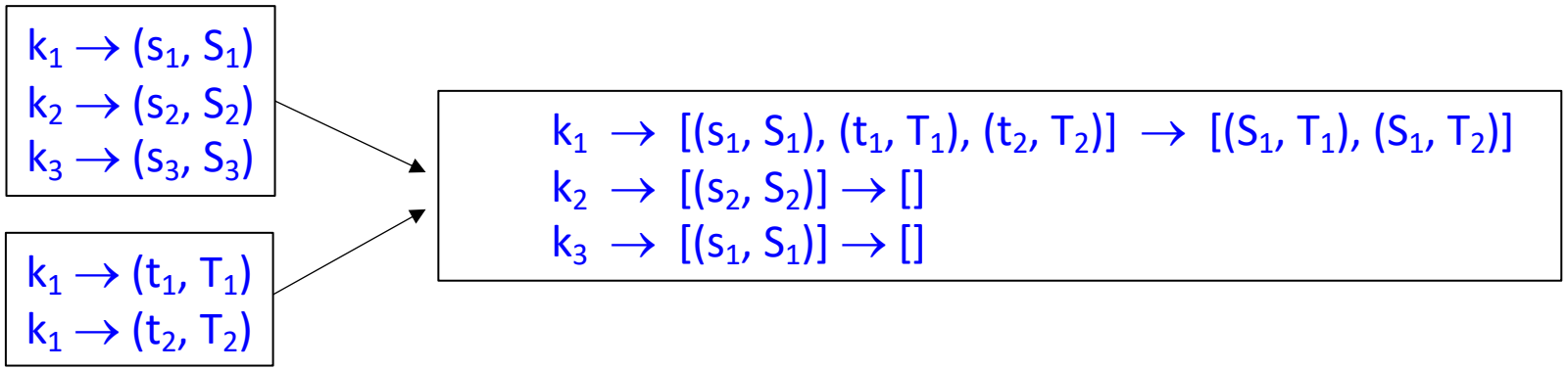
$k_1 \rightarrow (t_1, T_1)$
$k_1 \rightarrow (t_2, T_2)$

- **Optimize with secondary sorting idea** to group tuples from *S* and *T* together.

Note: You can configure **multiple mappers within a single job** using a custom InputFormat. This approach allows different Mappers to process different parts of the input.

Reduce-Side Joins

- Map emits the **join key as the intermediate key** and the remaining tuple as the intermediate value (for both relations *S* and *T*).
- Reduce obtains all tuples belonging to the same join key and **combines all tuples from *S* with the tuples from *T*** via a **nested loop**.



- **Optimize with secondary sorting idea** to group tuples from *S* and *T* together.

Note on Reduce-Side Joins

- The afore described algorithm is called "**parallel sort-merge join**" in databases.
- **Joining tuples at the reducer** however requires **shuffling both relations completely through the network**.
- Suppose each relation consists of **1 M tuples**, but only **1 tuple in *S* joins with 1 tuple in *R***. Then 1,999,998 tuples are shuffled in vain.
- So, what about a Map-side join?

Map-Side Joins

- For a Map-side join to be possible, we need to assume that both relations are already available **sorted and partitioned by their join keys**.
- We perform a **standard** (but parallel) **merge-join** by mapping over the larger relation and by reading the corresponding partition of the smaller relation directly inside the Map function. Tuples with matching join keys are again joined together via a nested-loop, but this time in the Map function.
- **No Reducer join is needed**, the output of the Map function are already the joined tuples.

A Third Technique: Memory-Backed Joins

- If one of the relations fits into main-memory, we can load that relation into a **shared main-memory data structure** which is then made accessible by all Mappers.
- We again map over the larger relation and **probe each key** of that relation via a random-access into the shared main-memory data structure.
- This is similar to a **standard** (but parallel) **hash join**.
- However, sharing large main-memory data structures across an entire compute cluster is tricky.

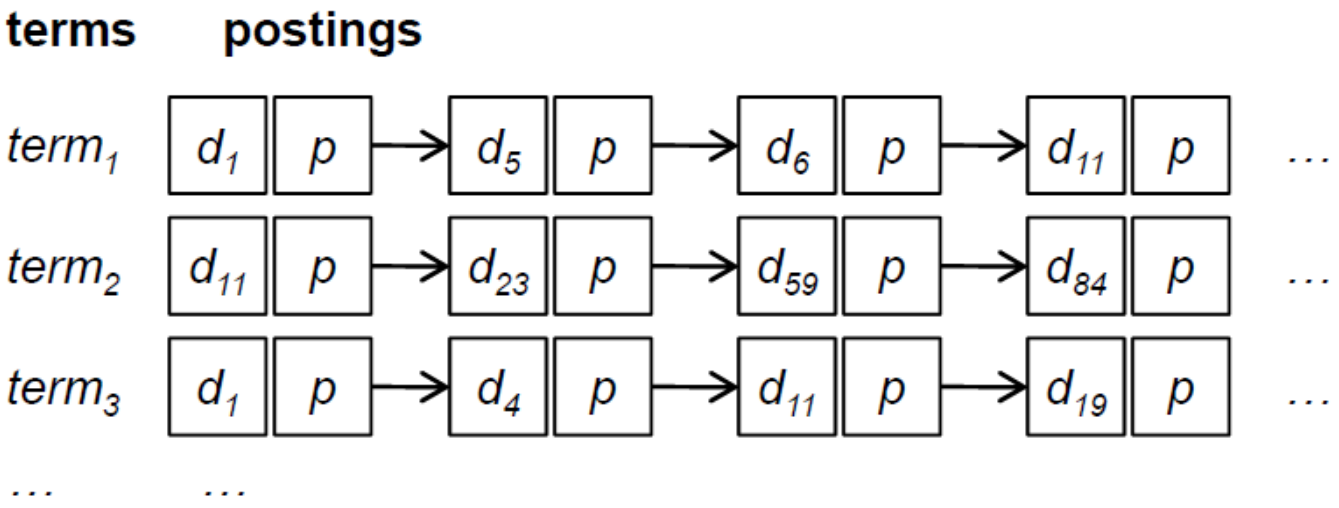
Credits

http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

https://hadoopi.wordpress.com/2013/05/27/understand-recordreader-inputsplit/

Inverted Indexes

- An inverted index contains, for each term, a list of *(docId, score)* pairs (so called "postings") with pointers to all documents that contain that term.



- We have already seen the benefit of using an inverted index for querying text documents.
- In MapReduce, the challenge lies in parsing the documents and to create positing lists that are sorted by *docId* efficiently.

Basic Inverted Index Implementation

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure Reduce(term t, postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)
7:         Emit(term t, postings P)
```

- But: assumes the posting list for each term to fit into memory.
- What about that secondary sorting idea, again?

Inverted Indexing with Secondary Sorting

- Instead of emitting this:

  map(docId, doc) → [(term, (docId, count))]

- We directly emit pairs of terms and document ids as keys in the Mapper:

  map(docId, doc) → [((term, docId), count)]

  (See also our earlier secondary sorting example)

```
1:  class Mapper
2:      method Map(docid n, doc d)
3:          H ← new AssociativeArray
4:          for all term t ∈ doc d do
5:              H{t} ← H{t} + 1
6:          for all term t ∈ H do
7:              Emit(tuple ⟨t, n⟩, tf H{t})

1:  class Reducer
2:      method Initialize
3:          t_prev ← ∅
4:          P ← new PostingsList
5:      method Reduce(tuple ⟨t, n⟩, tf [f])
6:          if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:              Emit(term t, postings P)
8:              P.Reset()
9:          P.Add(⟨n, f⟩)
10:         t_prev ← t
11:     method Close
12:         Emit(term t, postings P)
```