MapReduce Design Patterns
MapReduce Restrictions

- Any algorithm that needs to be implemented using MapReduce must be expressed in terms of a small number of rigidly defined components that must fit together in very specific ways.
MapReduce Restrictions

- Any algorithm that needs to be implemented using MapReduce must be expressed in terms of a small number of rigidly defined components that must fit together in very specific ways.

- Synchronization is difficult. Within a single MapReduce job, there is only one opportunity for cluster-wide synchronization—during the shuffle and sort stage.
MapReduce Restrictions

- Any algorithm that needs to be implemented using MapReduce must be expressed in terms of a small number of rigidly defined components that must fit together in very specific ways.

- Synchronization is difficult. Within a single MapReduce job, there is only one opportunity for cluster-wide synchronization—during the shuffle and sort stage.

- Developer has little control over the following aspects:
  - Where a mapper or reducer runs (i.e., on which node in the cluster)
  - When a mapper or reducer begins or finishes
  - Which input key-value pairs are processed by a specific mapper
  - Which intermediate key-value pairs are processed by a specific reducer
MapReduce Techniques

- The ability to construct complex data structures as keys and values to store and communicate partial results.
MapReduce Techniques

- 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.
MapReduce Techniques

- 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.
MapReduce Techniques

- 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.
- The ability to iterate over multiple MapReduce jobs using a driver program.
MapReduce Techniques

- 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.
MapReduce Techniques

- 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.
- The ability to iterate over multiple MapReduce jobs using a driver program.
Local Aggregation

We will use the wordcount example to illustrate these techniques.

- **Use Combiners.** In Hadoop, combiners are considered optional optimizations so they cannot be counted on for correctness or to be even run at all.
Local Aggregation

We will use the wordcount example to illustrate these techniques.

- **Use Combiners.** In Hadoop, combiners are considered optional optimizations so they cannot be counted on for correctness or to be even run at all.
- **With the local aggregation technique,** we can incorporate combiner functionality directly inside the mappers (under our control) as explained below.
Local Aggregation

We will use the wordcount example to illustrate these techniques.

- **Use Combiners.** In Hadoop, combiners are considered optional optimizations so they cannot be counted on for correctness or to be even run at all.

- **With the local aggregation technique,** we can incorporate combiner functionality directly inside the mappers (under our control) as explained below.

- **In-Mapper Combining.** An associative array (e.g. Map in Java) 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.
In-Mapper Combining

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]) ▷ Tally counts for entire document
In-Mapper Combining Across Multiple Documents

- Prior to processing any input key-value pairs we initialize an associative array for holding term counts in the mapper’s initialize method. For example, in Hadoop’s new API, there is a \texttt{setup(...)} method that is called before processing any key-value pairs.
Prior to processing any input key-value pairs we initialize an associative array for holding term counts in the mapper’s initialize method. For example, in Hadoop’s new API, there is a `setup(...)` method that is called before processing any key-value pairs.

We can continue to accumulate partial term counts in the associative array across multiple documents, and emit key-value pairs only when the mapper has processed all documents.
In-Mapper Combining Across Multiple Documents

- Prior to processing any input key-value pairs we initialize an associative array for holding term counts in the mapper’s initialize method. For example, in Hadoop’s new API, there is a `setup(...)` method that is called before processing any key-value pairs.

- We can continue to accumulate partial term counts in the associative array across multiple documents, and emit key-value pairs only when the mapper has processed all documents.

- This requires an API hook that provides an opportunity to execute user-specified code after the Map method has been applied to all input key-value pairs of the input data split to which the map task was assigned.
In-Mapper Combining Across Multiple Documents

- Prior to processing any input key-value pairs we initialize an associative array for holding term counts in the mapper’s initialize method. For example, in Hadoop’s new API, there is a `setup(...)` method that is called before processing any key-value pairs.

- We can continue to accumulate partial term counts in the associative array across multiple documents, and emit key-value pairs only when the mapper has processed all documents.

- This requires an API hook that provides an opportunity to execute user-specified code after the Map method has been applied to all input key-value pairs of the input data split to which the map task was assigned.

- The Mapper class in the new Hadoop API provides this hook as the method named `cleanup(...)`. 
In-Mapper Combining Across Multiple Documents

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])

▷ Tally counts across documents
In-Mapper Combining Analysis

- **Advantages**: In-mapper combining will be more efficient than using Combiners since we have more control over the process and we save having to serialize/deserialize objects multiple times.
In-Mapper Combining Analysis

▶ **Advantages:** In-mapper combining will be more efficient than using Combiners since we have more control over the process and we save having to serialize/deserialize objects multiple times.

▶ **Drawbacks.**
  ▶ State preservation across mappers breaks the MapReduce paradigm. This may lead to ordering dependent bugs that are hard to track.
In-Mapper Combining Analysis

- **Advantages:** In-mapper combining will be more efficient than using Combiners since we have more control over the process and we save having to serialize/deserialize objects multiple times.

- **Drawbacks.**
  - State preservation across mappers breaks the MapReduce paradigm. This may lead to ordering dependent bugs that are hard to track.
  - Scalability bottlenecks if the number of keys we encounter cannot fit in memory.
In-Mapper Combining Analysis

- **Advantages**: In-mapper combining will be more efficient than using Combiners since we have more control over the process and we save having to serialize/deserialize objects multiple times.

- **Drawbacks**.
  - State preservation across mappers breaks the MapReduce paradigm. This may lead to ordering dependent bugs that are hard to track.
  - Scalability bottlenecks if the number of keys we encounter cannot fit in memory. This can be addressed by emitting partial results after every $n$ key-value pairs, or after certain fraction of memory has been used or when a certain amount of memory (buffer) is filled up.
In-Mapper Combiner: Another Example

Suppose we have a large data set where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key.
In-Mapper Combiner: Another Example

Suppose we have a large data set where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key.

A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session—the task would correspond to computing the mean session length on a per-user basis, which would be useful for understanding user demographics.
In-Mapper Combiner: Another Example

Suppose we have a large data set where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key.
A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session—the task would correspond to computing the mean session length on a per-user basis, which would be useful for understanding user demographics.

- Write MapReduce pseudo-code to solve the problem.
In-Mapper Combiner: Another Example

Suppose we have a large data set where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key.

A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session—the task would correspond to computing the mean session length on a per-user basis, which would be useful for understanding user demographics.

- Write MapReduce pseudo-code to solve the problem.
- Modify the solution to use Combiners. Note that

\[ \text{Mean}(1, 2, 3, 4, 5) \neq \text{Mean}(\text{Mean}(1, 2), \text{Mean}(3, 4, 5)) \]
In-Mapper Combiner: Another Example

Suppose we have a large data set where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key. A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session—the task would correspond to computing the mean session length on a per-user basis, which would be useful for understanding user demographics.

- Write MapReduce pseudo-code to solve the problem.
- Modify the solution to use Combiners. Note that

\[
\text{Mean}(1, 2, 3, 4, 5) \neq \text{Mean}(\text{Mean}(1, 2), \text{Mean}(3, 4, 5))
\]

- Modify the solution to use in-mapper combining.
Calculating Mean: Basic Solution

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})
Calculating Mean: With Combiners

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:     \( \text{sum} \leftarrow 0 \)
4:     \( \text{cnt} \leftarrow 0 \)
5:     **for all** pair (s, c) \( \in \) pairs \([(s₁, c₁), (s₂, c₂) ...] \) **do**
6:     \( \text{sum} \leftarrow \text{sum} + s \)
7:     \( \text{cnt} \leftarrow \text{cnt} + c \)
8:     Emit(string t, pair (sum, cnt))
Calculating Mean: Modified Reducer

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)
Calculating Mean: With In-Mapper Combining

1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S[t] ← S[t] + r
7:         C[t] ← C[t] + 1
8:     method Close
9:         for all term t ∈ S do
10:             Emit(term t, pair (S[t], C[t]))
Another example: Unique Items Counting

There is a set of records. Each record has field $F$ and arbitrary number of category labels $G = \{G_1, G_2, \ldots\}$. Count the total number of unique values of field $F$ for each subset of records for each value of any label.

Record 1: $F=1$, $G=\{a, b\}$
Record 2: $F=2$, $G=\{a, d, e\}$
Record 3: $F=1$, $G=\{b\}$
Record 4: $F=3$, $G=\{a, b\}$

Result:
- $a \rightarrow 3$ // $F=1$, $F=2$, $F=3$
- $b \rightarrow 2$ // $F=1$, $F=3$
- $d \rightarrow 1$ // $F=2$
- $e \rightarrow 1$ // $F=2$

▶ Come up with a two-pass solution.
▶ Come up with a one-pass solution that uses combining in the reducer.
Another example: Unique Items Counting

There is a set of records. Each record has field $F$ and arbitrary number of category labels $G = \{G_1, G_2, \ldots\}$. Count the total number of unique values of field $F$ for each subset of records for each value of any label.

Record 1: $F=1$, $G=\{a, b\}$
Record 2: $F=2$, $G=\{a, d, e\}$
Record 3: $F=1$, $G=\{b\}$
Record 4: $F=3$, $G=\{a, b\}$

Result:

$\begin{align*}
a &\rightarrow 3 \quad // \quad F=1, \ F=2, \ F=3 \\
b &\rightarrow 2 \quad // \quad F=1, \ F=3 \\
d &\rightarrow 1 \quad // \quad F=2 \\
e &\rightarrow 1 \quad // \quad F=2
\end{align*}$
Another example: Unique Items Counting

There is a set of records. Each record has field $F$ and arbitrary number of category labels $G = \{G_1, G_2, \ldots\}$. Count the total number of unique values of field $F$ for each subset of records for each value of any label.

Record 1: $F=1$, $G=\{a, b\}$
Record 2: $F=2$, $G=\{a, d, e\}$
Record 3: $F=1$, $G=\{b\}$
Record 4: $F=3$, $G=\{a, b\}$

Result:

a -> 3 // $F=1$, $F=2$, $F=3$
b -> 2 // $F=1$, $F=3$
d -> 1 // $F=2$
e -> 1 // $F=2$

▶ Come up with a two-pass solution.
Another example: Unique Items Counting

There is a set of records. Each record has field $F$ and arbitrary number of category labels $G = \{G_1, G_2, \ldots\}$. Count the total number of unique values of field $F$ for each subset of records for each value of any label.

Record 1: $F=1, G=\{a, b\}$
Record 2: $F=2, G=\{a, d, e\}$
Record 3: $F=1, G=\{b\}$
Record 4: $F=3, G=\{a, b\}$

Result:

- $a \rightarrow 3$ // $F=1, F=2, F=3$
- $b \rightarrow 2$ // $F=1, F=3$
- $d \rightarrow 1$ // $F=2$
- $e \rightarrow 1$ // $F=2$

- Come up with a two-pass solution.
- Come up with a one-pass solution that uses combining in the reducer.
Cross-Correlation

There is a set of tuples of items. For each possible pair of items calculate the number of tuples where these items co-occur. If the total number of items is $n$, then $n^2 = n \times n$ values should be reported.
Cross-Correlation

- There is a set of tuples of items. For each possible pair of items calculate the number of tuples where these items co-occur. If the total number of items is $n$, then $n^2 = n \times n$ values should be reported.

- This problem appears in text analysis (say, items are words and tuples are sentences), market analysis (customers who buy this tend to also buy that). If $n^2$ is quite small and such a matrix can fit in the memory of a single machine, then implementation is straightforward.
Cross-Correlation

- There is a set of tuples of items. For each possible pair of items calculate the number of tuples where these items co-occur. If the total number of items is $n$, then $n^2 = n \times n$ values should be reported.

- This problem appears in text analysis (say, items are words and tuples are sentences), market analysis (customers who buy this tend to also buy that). If $n^2$ is quite small and such a matrix can fit in the memory of a single machine, then implementation is straightforward.

- We will study two ways to solving this problem that illustrate two patterns: *pairs* versus *stripes*. 
Pairs and Stripes Patterns

- **Pairs pattern.** The mapper finds each co-occurring pair and outputs it with a count of 1. The reducer just adds up the frequencies for each pair. This requires the use of complex keys (a pair of words).

- **Stripes pattern.** Instead of emitting intermediate key-value pairs for each co-occurring word pair, co-occurrence information is first stored in an associative array, denoted $H$. The mapper emits key-value pairs with words as keys and corresponding associative arrays as values. The reducer performs an element-wise sum of all associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. The final associative array is emitted with the same word as the key. In contrast to the pairs approach, each final key-value pair encodes a row in the co-occurrence matrix.
Pairs and Stripes Patterns

- **Pairs pattern.** The mapper finds each co-occurring pair and outputs it with a count of 1. The reducer just adds up the frequencies for each pair. This requires the use of complex keys (a pair of words).

- **Stripes pattern.**
  - Instead of emitting intermediate key-value pairs for each co-occurring word pair, co-occurrence information is first stored in an associative array, denoted $H$. The mapper emits key-value pairs with words as keys and corresponding associative arrays as values.
Pairs and Stripes Patterns

- **Pairs pattern.** The mapper finds each co-occurring pair and outputs it with a count of 1. The reducer just adds up the frequencies for each pair. This requires the use of complex keys (a pair of words).

- **Stripes pattern.**
  - Instead of emitting intermediate key-value pairs for each co-occurring word pair, co-occurrence information is first stored in an associative array, denoted \( H \). The mapper emits key-value pairs with words as keys and corresponding associative arrays as values.
  - The reducer performs an element-wise sum of all associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. The final associative array is emitted with the same word as the key.
Pairs and Stripes Patterns

- **Pairs pattern.** The mapper finds each co-occurring pair and outputs it with a count of 1. The reducer just adds up the frequencies for each pair. This requires the use of complex keys (a pair of words).

- **Stripes pattern.**
  - Instead of emitting intermediate key-value pairs for each co-occurring word pair, co-occurrence information is first stored in an associative array, denoted $H$. The mapper emits key-value pairs with words as keys and corresponding associative arrays as values.
  - The reducer performs an element-wise sum of all associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. The final associative array is emitted with the same word as the key.
  - In contrast to the pairs approach, each final key-value pair encodes a row in the co-occurrence matrix.
Calculating Co-occurrences: With Pairs Pattern

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)
Calculating Co-occurrences: With Stripes Pattern

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 [H₁, H₂, H₃, ...])
3.      Hᵢ ← new AssociativeArray
4.  **for all** stripe H ∈ stripes [H₁, H₂, H₃, ...] do
5.      Sum(Hᵢ, H)  // Element-wise sum
6.      Emit(term w, stripe Hᵢ)
Pairs versus Stripes

- *Stripes* generates fewer intermediate keys than *Pairs* approach.
Pairs versus Stripes

- Stripes generates fewer intermediate keys than Pairs approach.
- Stripes benefits more from combiners and can be done with in-memory combiners.
Pairs versus Stripes

- *Stripes* generates fewer intermediate keys than *Pairs* approach.
- *Stripes* benefits more from combiners and can be done with in-memory combiners.
- *Stripes* is, in general, faster.
Pairs versus Stripes

- *Stripes* generates fewer intermediate keys than *Pairs* approach.
- *Stripes* benefits more from combiners and can be done with in-memory combiners.
- *Stripes* is, in general, faster.
- *Stripes* requires more complex implementation.
References

- Jimmy Lin and Chris Dyer. Chapter 3 in *Data-Intensive Text Processing with MapReduce*.