Now let's look at important program "design patterns" for MapReduce.

MapReduce Design Patterns

- This section is based on the book by Jimmy Lin and Chris Dyer
- Programmer can control program execution only through implementation of mapper, reducer, combiner, and partitioner
- No explicit synchronization primitives
- So how can a programmer control execution and data flow?

Taking Control of MapReduce

- Store and communicate partial results through complex data structures for keys and values
- Run appropriate initialization code at beginning of task and termination code at end of task
- Preserve state in mappers and reducers across multiple input splits and intermediate keys, respectively
- Control sort order of intermediate keys to control processing order at reducers
- Control set of keys assigned to a reducer
- Use "driver" program

(1) Local Aggregation

- Reduce size of intermediate results passed from mappers to reducers
  – Important for scalability: recall Amdahl's Law
- Various options using combiner function and ability to preserve mapper state across multiple inputs
- For example, consider Word Count with the document-based version of Map

Word Count Baseline Algorithm

```
map(docID a, doc d)
  for all term t in doc d do
    Emit(term t, count 1)
```

```
reduce(term t, counts [c1, c2,…])
  sum = 0
  for all count c in counts do
    sum += c
  Emit(term t, count sum);
```

- Problem: frequent terms are emitted many times with count 1

Tally Counts Per Document

```
map(docID a, doc d)
  H = new hashMap
  for all term t in doc d do
    H{t} ++
  for all term t in H do
    Emit(term t, count H{t})
```

- Same Reduce function as before
- Limitation: Map only aggregates counts within a single document
- Depending on split size and document size, a Map task might receive many documents
- Can we aggregate across all documents in the same Map task?

Tally Counts Across Documents

- Data structure H is a private member of the Mapper class
  - Local to a single task, i.e., does not introduce task synchronization issues
- Initialize is called when the task starts, i.e., before all map calls
  - Configure() in old API
  - Setup() in new API
- Close is called after the last document from the Map task has been processed
  - Close() in old API
  - Cleanup() in new API

```
Class Mapper {
 initialize() {
  H = new hashMap
 }

 map(docID a, doc d) {
  for all term t in doc d do
   H{t} ++
 }

 close() {
  for all term t in H do
   Emit(term t, count H{t})
 }
}
```


Design Pattern for Local Aggregation

- In-mapper combining
  - Done by preserving state across map calls in the same task
- Advantages over using combiners
  - Combiner does not guarantee if, when or how often it is executed
  - Combiner combines data *after* it was generated, in-mapper combining avoids generating it!
- Drawbacks
  - Introduces complexity and hence probability for bugs
  - Higher memory consumption for managing state
    - Might have to write memory-management code to page data to disk

(2) Counting of Combinations

- Needed for computing correlations, associations, confusion matrix (how many times does a classifier confuse Y_i with Y_j)
- Co-occurrence matrix for a text corpus: how many times do two terms appear near each other
- Main idea: compute partial counts for some combinations, then aggregate them
  - At what granularity should Map work?

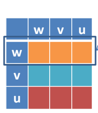
Pairs Design Pattern


```
map(docID a, doc d)
 for all term w in doc d do
  for all term u NEAR w do
   Emit(pair (w, u), count 1)

reduce(pair p, counts [c1, c2,…])
 sum = 0
 for all count c in counts do
  sum += c
 Emit(pair p, count sum)
```

- Can use combiner or in-mapper combining
- Good: easy to implement and understand
- Bad: huge intermediate-key space
  - Quadratic in number of distinct terms

Stripes Design Pattern




```
map(docID a, doc d)
 for all term w in doc d do
  H = new hashMap
  for all term u NEAR w do  H{u} ++
  Emit(term w, stripe H)

reduce(term w, stripes [H1, H2,…])
 Hout = new hashMap
 for all stripe H in stripes do  Hout = ElementWiseSum(Hout, H)
 Emit(term w, stripe Hout)
```

- Can use combiner or in-mapper combining
- Good: much smaller intermediate-key space
  - Linear in number of distinct terms
- Bad: more difficult to implement, Map needs to hold entire stripe in memory

Note About Stripes Map Code

- Pairs' Map code only needs a single sequential scan of the document, keeping the current term w and a "sliding window" of the nearby terms to its left and right
- Stripes can do the same, but then it does not aggregate counts across *multiple occurrences* of the same term w in document d, i.e., would mostly produce counts of 1 in the hash map
- To aggregate across all occurrences of w in d, Stripes would have to repeatedly scan the document, once for each distinct term w in d
  - Could create an index to find repeated occurrences of w faster
- Or use a two-dim. hash map H[w][u] in the Map function, allowing a single-scan solution at higher memory cost

Pairs versus Stripes

- *Without* combiner or in-mapper combining, Pairs could produce significantly more mapper output
  - ((w,u),1) per pair for Pairs, versus per-document aggregates for Stripes
- …but it would need a lot less memory
  - Pairs essentially needs no extra storage beyond the current "window" of nearby words, while Stripes has to store the hash map H

Pairs versus Stripes (cont.)

- *With* combiner or in-mapper combining, Map would produce about the same amount of data in both cases
  - Two-dimensional index Pairs[w][u] with per-task counts for each pair (w,u) is the same as one-dimensional index of one-dimensional indexes (Stripes[w])[u]
- …and would also require about the same amount of memory to store the two-dimensional count data structure

Pairs versus Stripes (cont.)

- Does the number of keys matter?
  - Assume we use the same number of tasks, then Pairs just assigns more keys per task
  - Master works with tasks, hence no conceptual difference between Pairs and Stripes
- More fine-grained keys of Pairs allow more flexibility in assigning keys to tasks
  - Pairs can emulate Stripes' row-wise key assignment to tasks
  - Stripes cannot emulate all Pairs assignments, e.g., "checkerboard" pattern for two tasks
- Greater number of distinct keys per task in Pairs tends to increase sorting cost, even if total data size is the same

Beyond Pairs and Stripes

- In general, it is not clear which approach is better
  - Some experiments indicate stripes win for co-occurrence matrix computation
- Pairs and Stripes are special cases of shapes for covering the entire matrix
  - Could use sub-stripes, or partition matrix horizontally and vertically into more square-like shapes etc.
- Can also be applied to higher-dimensional arrays
- Will see interesting version of this idea for joins

(3) Relative Frequencies

- Important for data mining
- E.g., for each species and color, estimate the probability of the color for that species
  - Probability of Northern Cardinal being red: P(color = red | species = N.C.)
    - Count f(N.C.) = the frequency of observations for N.C. (marginal)
    - Count f(N.C., red) = the frequency of observations for red N.C.'s (joint event)
    - Estimate P(red | N.C.) as f(N.C., red) / f(N.C.)
- Similarly: normalize word co-occurrence vector for word w

Bird Probabilities Using Stripes

- Use species as intermediate key
  - One stripe per species, e.g., stripe[N.C.]
  - (stripe[species])[color] stores f(species, color)
- Map: for each observation of (species S, color C) in an observation event, increment (stripe[S])[C]
  - Output (S, stripe[S])
- Reduce: for each species S, add all stripes for S
  - Result: stripeSum[S] with total counts for each color for S
  - Can get f(S) by adding all color-counts in stripeSum[S]
  - Emit (stripeSum[S])[C] / f(S) for each color C

Discussion, Part 1

- Stripe is great fit for relative frequency computation
- All values for computing the final result are in the stripe
- Any smaller unit would miss some of the joint events needed for computing f(S), the marginal for the species
  - So, this would be a problem for the pairs pattern

Bird Probabilities Using Pairs

- Intermediate key is (species, color)
- Map produces partial counts for each species-color combination in the input
- Reduce can compute f(species, color), the total count of each species-color combination
- But: it cannot compute the marginal f(S)
  - Reduce needs to sum f(S, color) for all colors for species S

Pairs-Based Solution, Take 1

- Make sure all values f(S, color) for the same species end up in the same reduce task
  - Define custom partitioning function on species
- Maintain state across different keys in the same reduce task: keep stripe[S] in memory as a variable in the Reduce task
  - This essentially simulates the stripes approach in the reduce task, requiring to keep a stripe in memory
- Can we avoid keeping a stripe?

Discussion, Part 2

- Pairs-based algorithm would work fine if marginal f(S) was known already
  - A Reduce function call computes f(species, color) and then outputs f(species, color) / f(species)
- We could compute the species marginals f(species) in a separate MapReduce job first
- Better: fold this into a single MapReduce job
  - Problem: easy to compute f(S) from all f(S, color), but how do we compute f(S) before knowing f(S, color)?

Bird Probabilities Using Pairs, Take 2

- Map: for each observation event, emit ((species S, color C), 1) and ((species S, dummyColor), 1) for each species-color combination encountered
- Use custom partitioner that partitions based on the species component only
- Use custom key comparator such that (S, dummyColor) is before all (S, C) for real colors C
  - Reduce call for dummyColor happens first and computes f(S) before any of the f(S, C)
    - Reducer needs to keep f(S) for the duration of the entire task
  - Reducer then computes f(S, C) for each C, outputting f(S, C) / f(S)
- Only needs counter f(S), not the entire stripe, in memory

Pairs-Based Code

```
map(…, observation: (species S, color C))
  Emit( (S, dummy), 1 )
  Emit( (S, C), 1 )

Partitioner: partition by species

Key comparator for (species, color):
- Sort by species first
- Make sure color “dummy” comes
  before all real colors
```

```
Class Reducer {
  int marginal

  reduce((S, C), counts [c1, c2,…]) {
    if C = dummy       // Compute marginal
      marginal = 0
      for all c in counts do
        marginal += c
    else               // Real color
      colorCnt = 0
      for all c in counts do
        colorCnt += c
      Emit( (S, C), colorCnt / marginal )
  }
}
```

Order Inversion Design Pattern

- Occurs surprisingly often during data analysis
- Solution 1: use complex data structures that bring the right results together
  - Array structure used by Stripes pattern
- Solution 2: turn synchronization into ordering problem
  - Key sort order enforces computation order
  - Partitioner for key space assigns appropriate partial results to each reduce task
  - Reducer maintains task-level state across Reduce invocations
  - Enables use of simpler data structures and less reducer memory

(4) Secondary Sorting

- Recall the weather data: for simplicity assume observations are (date, temperature)
- Goal: find max temperature for each year
  - Reduce task should have all temperatures for a year: year as intermediate key
  - Temperatures in reduce input value list should be sorted in decreasing order by temperature
- Year as key does not sort by temperature
- (Year, temperature) as key creates different reduce calls for each temperature in a year

Can Hadoop Do The Sorting?

- We want to use year to partition the data, but (year, temperature) for sorting
- General value-to-key conversion design pattern
  - To partition by X and then sort each X-group by Y, make (X, Y) the key
  - Define key comparator to order by composite key (X, Y)
  - Define partitioner and grouping comparator for (X, Y) to consider only X for partitioning and grouping

Code for Secondary Sort

```
public class MaxTemperatureUsingSecondarySort
  extends Configured implements Tool {

  static class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, IntPair, NullWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

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

      parser.parse(value);
      if (parser.isValidTemperature()) {
        context.write(new IntPair(parser.getYearInt(), parser.getAirTemperature()),
            NullWritable.get());
      }
    }
  }
```

No value is output, because the year and temperature are in the key.


```
static class MaxTemperatureReducer
  extends Reducer<IntPair, NullWritable, IntPair, NullWritable> {

  @Override
  protected void reduce(IntPair key, Iterable<NullWritable> values,
      Context context) throws IOException, InterruptedException {

    context.write(key, NullWritable.get());
  }
}
```

The reducer only emits the first key, which due to secondary sorting, is the (year, temperature) pair with the maximum temperature for that year.

To get all temperatures for the year, we would have to emit temperatures as values in the map function.


```
public static class FirstPartitioner extends Partitioner<IntPair, NullWritable> {

  public int getPartition(IntPair key, NullWritable value, int numPartitions) {

    // multiply by 127 to perform some mixing
    return Math.abs(key.getFirst() * 127) % numPartitions;
  }
}
```

Make sure all records for the same year end up in the same partition.

```
// Controls how keys are sorted before they are passed to the reducer
public static class KeyComparator extends WritableComparator {
  protected KeyComparator() {
    super(IntPair.class, true);
  }
  public int compare(WritableComparable w1, WritableComparable w2) {
    IntPair ip1 = (IntPair) w1; IntPair ip2 = (IntPair) w2;
    int cmp = IntPair.compare(ip1.getFirst(), ip2.getFirst());
    if (cmp != 0) {
      return cmp;
    }
    return -IntPair.compare(ip1.getSecond(), ip2.getSecond()); //reverse
  } }

// Controls which keys are grouped into a single call of the reduce function
public static class GroupComparator extends WritableComparator {
  protected GroupComparator() {
    super(IntPair.class, true);
  }
  public int compare(WritableComparable w1, WritableComparable w2) {
    IntPair ip1 = (IntPair) w1; IntPair ip2 = (IntPair) w2;

    return IntPair.compare(ip1.getFirst(), ip2.getFirst());
  } }
```



```
  @Override
  public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
      return -1;
    }

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setPartitionerClass(FirstPartitioner.class);
    job.setSortComparatorClass(KeyComparator.class);
    job.setGroupingComparatorClass(GroupComparator.class);
    job.setReducerClass(MaxTemperatureReducer.class);
    job.setOutputKeyClass(IntPair.class);
    job.setOutputValueClass(NullWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

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


Design Pattern Summary

- In-mapper combining: do work of combiner in mapper
- Pairs and stripes: for keeping track of joint events
- Order inversion: convert sequencing of computation into sorting problem
- Value-to-key conversion: scalable solution for secondary sorting, without writing sort code

Tools for Synchronization

- Cleverly-constructed data structures for key and values to bring data together
- Preserving state in mappers and reducers, together with capability to add initialization and termination code for entire task
- Sort order of intermediate keys to control order in which reducers process keys
- Custom partitioner to control which reducer processes which keys

Issues and Tradeoffs

- Number of key-value pairs
  - Object creation overhead
  - Time for sorting and shuffling pairs across the network
- Size of each key-value pair
  - (De-)serialization overhead
- Local aggregation
  - Opportunities to perform local aggregation vary
  - Combiners can make a big difference
  - Combiners vs. in-mapper combining
  - RAM vs. disk vs. network