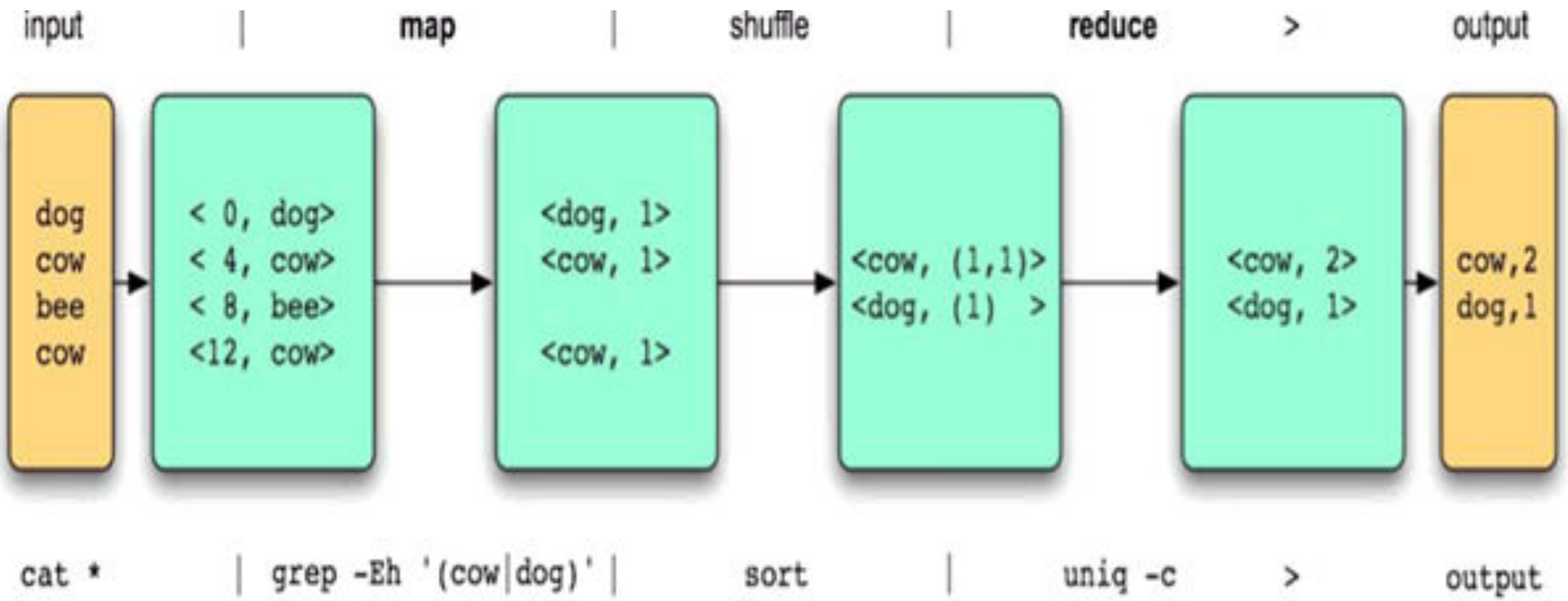
MapReduce Patterns



Intermediate Data

- **Written locally**
- **Transferred from mappers to reducers over network**
- **Issue**
  - Performance bottleneck
- **Solution**
  - Use combiners
  - Use **In-Mapper Combining**

Original Word Count

```
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)
```

- How many intermediate keys per mapper?
- How can we improve this?
- Is it a “real” improvement?

Stateless 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})
```

- Custom local aggregator
- Coding overhead
- Is it a “real” improvement?

Stateful In-Mapper Combining

```
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})
```

- Custom local aggregator
- Coding overhead
- Is it a “real” improvement?

In-Mapper Combining Analysis

- **Advantages:**
  - Complete local aggregation control (how and when)
  - Guaranteed to execute
  - Direct efficiency control on intermediate data creation
  - Avoid unnecessary objects creation and destruction (before combiners)
- **Disadvantages:**
  - Breaks the functional programming background (state)
  - Potential ordering-dependent bugs
  - Memory scalability bottleneck (solved by memory foot-printing and flushing)

Matrix Generation

- **Common problem**:

  - Given an input of size N, generate an output matrix of size N x N

- **Example**: word co-occurrence matrix

  - Given a document collection, emit the bigram frequencies

“Pairs”

```
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)
```

- We must use custom key type
- Intermediate overhead? Bottlenecks?
- Can we use the reducer as a combiner?
- Keys space?

“Stripes”

```
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:         Hf ← new AssociativeArray
4:         for all stripe H ∈ stripes [H1, H2, H3, ...] do
5:             Sum(Hf, H)                         ▷ Element-wise sum
6:         Emit(term w, stripe Hf)
```

- We must use custom key and value types
- Intermediate overhead? Bottlenecks?
- Can we use the reducer as a combiner?
- Keys space?