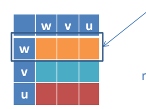
Pairs Design Pattern


```
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 (shuffling/sorting cost!)
  - 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

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, compute probability of 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)
    - P(red | N.C.) = f(N.C., red) / f(N.C.)
- Similarly: normalize word co-occurrence vector for word w by dividing it by w's frequency

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 stripeSum[S] values together
  - Get probability P(color = C | species = S) as (stripeSum[S])[C] / f(S)

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 input
- Reduce can compute f(species, color), the total count of each species-color combination
- But: cannot compute marginal f(S)
  - Reduce needs to sum f(S, color) for all colors for species S

210

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 same reduce task
- This essentially simulates the stripes approach in the reduce task, creating big reduce tasks when there are many colors
- Can we do better?

211

Discussion, Part 2

- Pairs-based algorithm would work better, if marginal f(S) was known already
  - Reducer computes f(species, color) and then outputs f(species, color) / f(species)
- We can 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)?

212

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
  - Reducer computes f(S) before the f(S, C)
    - Reducer keeps f(S) in state for duration of entire task
  - Reducer then computes f(S, C) for each C, outputting f(S, C) / f(S)
- Advantage: avoids having to manage all colors for a species together

213

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
  - Works for simpler pairs pattern, which uses simpler data structures and requires less reducer memory

214

(4) Secondary Sorting

- Recall the weather data: for simplicity assume observations are (date, stationID, temperature)
- Goal: for each station, create a time series of temperature measurements
- Per-station data: use stationID as intermediate key
- Problem: reducers receive huge number of (date, temp) pairs for each station
  - Have to be sorted by user code

215

Can Hadoop Do The Sorting?

- Use (stationID, date) as intermediate key
  - Problem: records for the some station might end up in different reduce tasks
  - Solution: custom partitioner, using only stationID component of key for partitioning
- 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
    - Grouping part is necessary if all dates for a station should be processed in the same Reduce invocation (otherwise each station-date combination ends up in a different Reduce invocation)

216

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 own sort code

217

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

218

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

219

Now that we have seen important design patterns and MapReduce algorithms for simpler problems, let's look at some more complex problems.

220

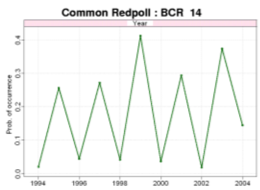
Joins in MapReduce

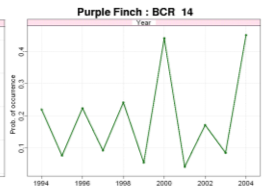
- Data sets $S=\{s_1,\ldots, s_{|S|}\}$ and $T=\{t_1,\ldots, t_{|T|}\}$
- Find all pairs (s_i, t_j) that satisfy some predicate
- Examples
  - Pairs of similar or complementary function summaries
  - Facebook and Twitter posts by same user or from same location
- Typical goal: minimize job completion time

221

Function-Join Pattern

- Find groups of summaries with certain properties of interest
  - Similar trends, opposite trends, correlations
  - Groups not known a priori, need to be discovered





Existing Join Support

- Hadoop has some built-in join support, but our goal is to design our own algorithms
  - Built-in support is limited
  - We want to understand important algorithm design principles
- “Join” usually just means equi-join, but we also want to support other join predicates
- Note: recall join discussion from earlier lecture

Joining Large With Small

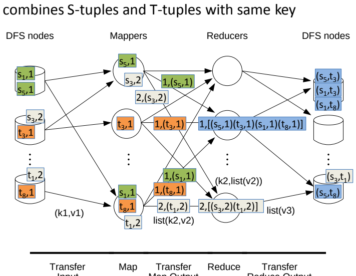
- Assume data set T is small enough to fit in memory
- Can run Map-only join
  - Load T onto every mapper
  - Map: join incoming S-tuple with T, output all matching pairs
    - Can scan entire T (nested loop) or use index on T (index nested loop)
- Downside: need to copy T to all mappers
  - Not so bad, since T is small

Distributed Cache

- Efficient way to copy files to all nodes processing a certain task
  - Use it to send small T to all mappers
- Part of the job configuration
- Hadoop still needs to move the data to the worker nodes, so use this with care
  - But it avoids copying the file for every task on the same node

Recall: Standard Equi-Join Algorithm

- Join condition: S.A=T.A
- Map(s) = (s.A, s); Map(t) = (t.A, t)
- Reduce combines S-tuples and T-tuples with same key

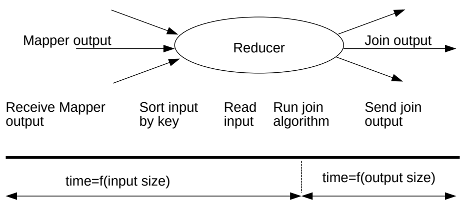


Problems With Standard Approach

- Degree of parallelism limited by number of distinct A-values
- Data skew
  - If one A-value dominates, reducer processing that key will become bottleneck
- Does not generalize to other joins

Reducer-Centric Cost Model

- Difference between join implementations starts with Map output



228

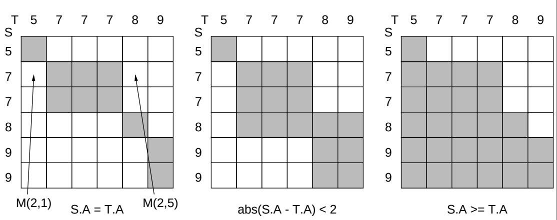
Optimization Goal: Minimal Job Completion time

- Assume all reducers are similarly capable
- Processing time at reducer is approximately monotonic in input and output size
- Hence need to minimize:
  - Max-reducer-input and/or
  - Max-reducer-output
- Join problem classification
  - Input-size dominated: minimize max-reducer-input
  - Output-size dominated: minimize max-reducer-output
  - Input-output balanced: minimize combination of both

229

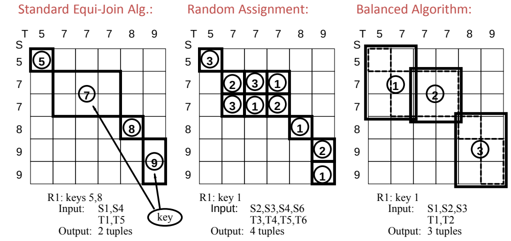
Join Model

- Join-matrix M: M(i, j) = *true*, if and only if (s_i, t_j) in join result
- Cover each *true*-valued cell by exactly one reducer



230

Standard Equi-Join Alg.: | Random Assignment: | Balanced Algorithm:



Standard Equi-Join Alg.:

R1: keys 5,8
Input: S1,S4
T1,T5
Output: 2 tuples

R2: key 7
Input: S2,S3
T2,T3,T4
Output: 6 tuples

R3: key 9
Input: S5,S6
T6
Output: 2 tuples

max-reducer-input = 5
max-reducer-output = 6

Random Assignment:

R1: key 1
Input: S2,S3,S4,S6
T3,T4,T5,T6
Output: 4 tuples

R2: key 2
Input: S2,S3,S5
T2,T4,T6
Output: 3 tuples

R3: key 3
Input: S1,S2,S3
T1,T2,T3
Output: 3 tuples

max-reducer-input = 8
max-reducer-output = 4

Balanced Algorithm:

R1: key 1
Input: S1,S2,S3
T1,T2
Output: 3 tuples

R2: key 2
Input: S2,S3
T3,T4
Output: 4 tuples

R3: key 3
Input: S4,S5,S6
T5,T6
Output: 3 tuples

max-reducer-input = 5
max-reducer-output = 4

231