Common MapReduce Patterns

Chris K Wensel

BuzzWords 2011

Engineer, Not Academic

- Concurrent, Inc., Founder
  - Cascading support and tools
  - http://concurrentinc.com/

- Cascading, Lead Developer (started Sept 2007)
  - An alternative API to MapReduce
  - http://cascading.org/

- Formerly Hadoop mentoring and training
  - Sun - Apple - HP - LexisNexis - startups - etc

- Formerly Systems Architect & Consultant
  - Thomson/Reuters - TeleAtlas - startups - etc

Overview

- MapReduce
- Heavy Lifting
- Analytics
- Optimizations

MapReduce

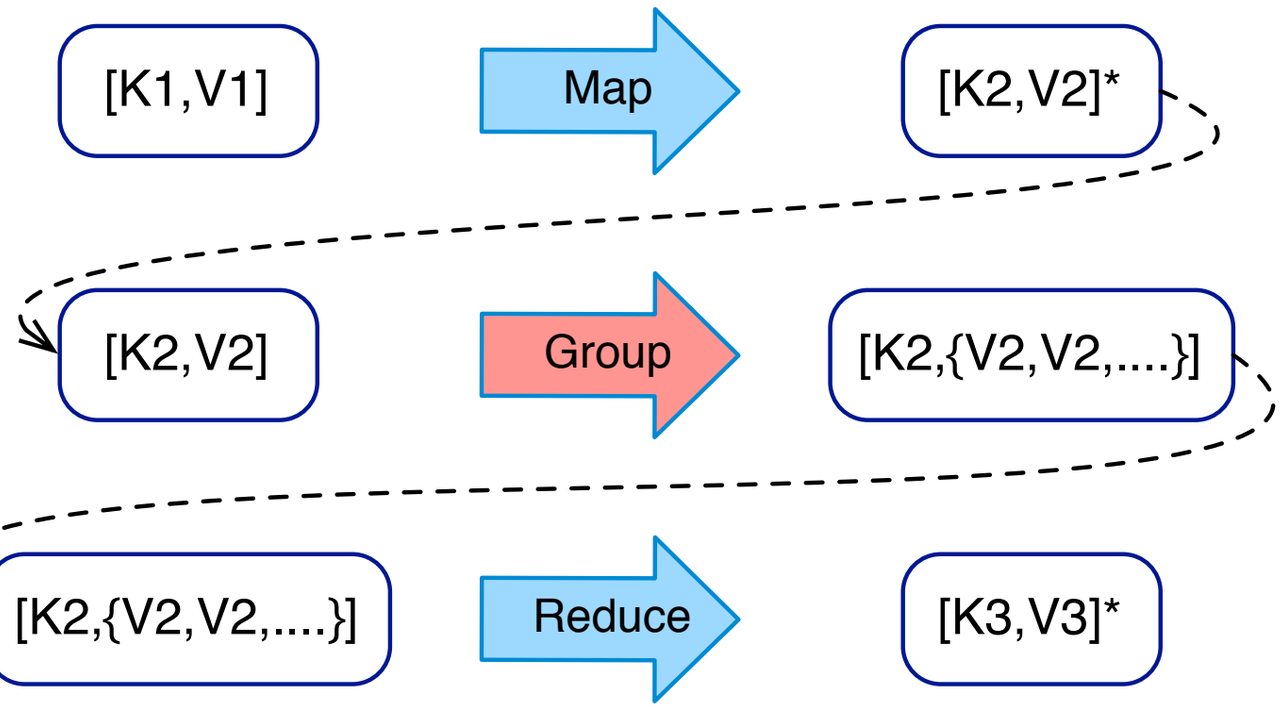
- A “divide and conquer” strategy for parallelizing workloads against collections of data

- Map & Reduce are two user defined functions chained via Key Value Pairs

- It’s really Map->Group->Reduce where Group is built in

Keys and Values

- Map translates input to keys and values to new keys and values

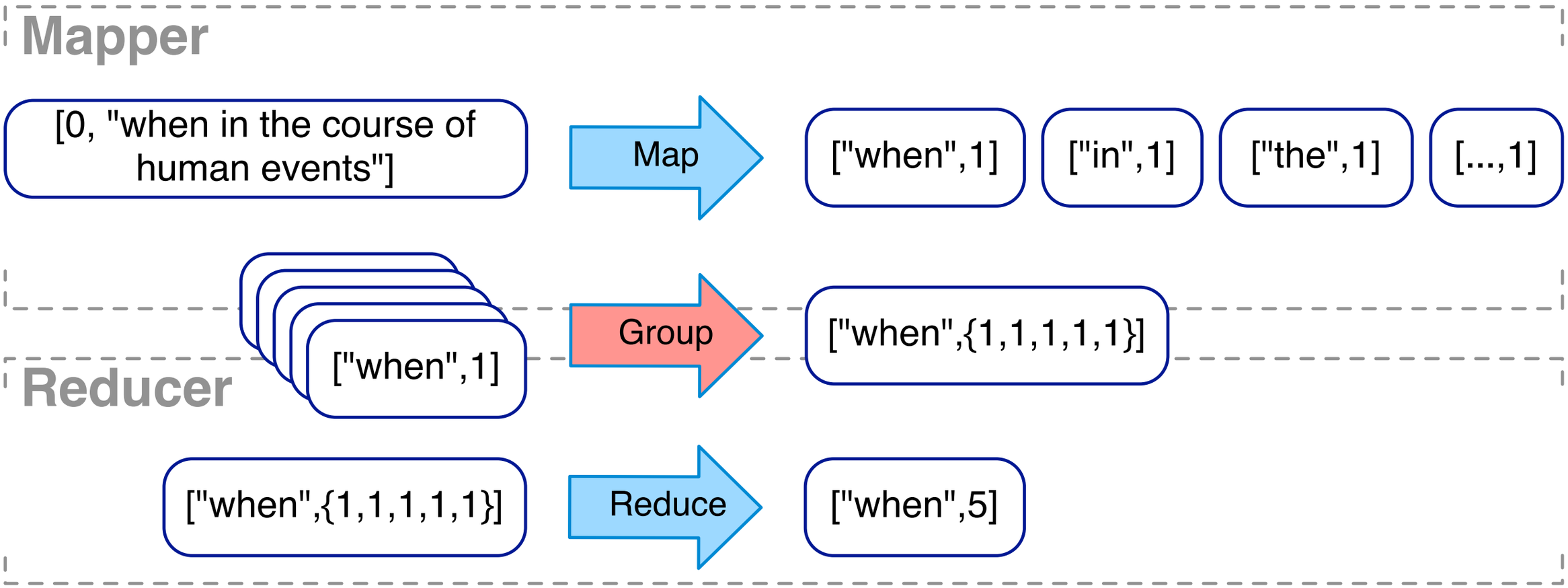


- System Groups each unique key with all its values

- Reduce translates the values of each unique key to new keys and values

* = zero or more

Word Count



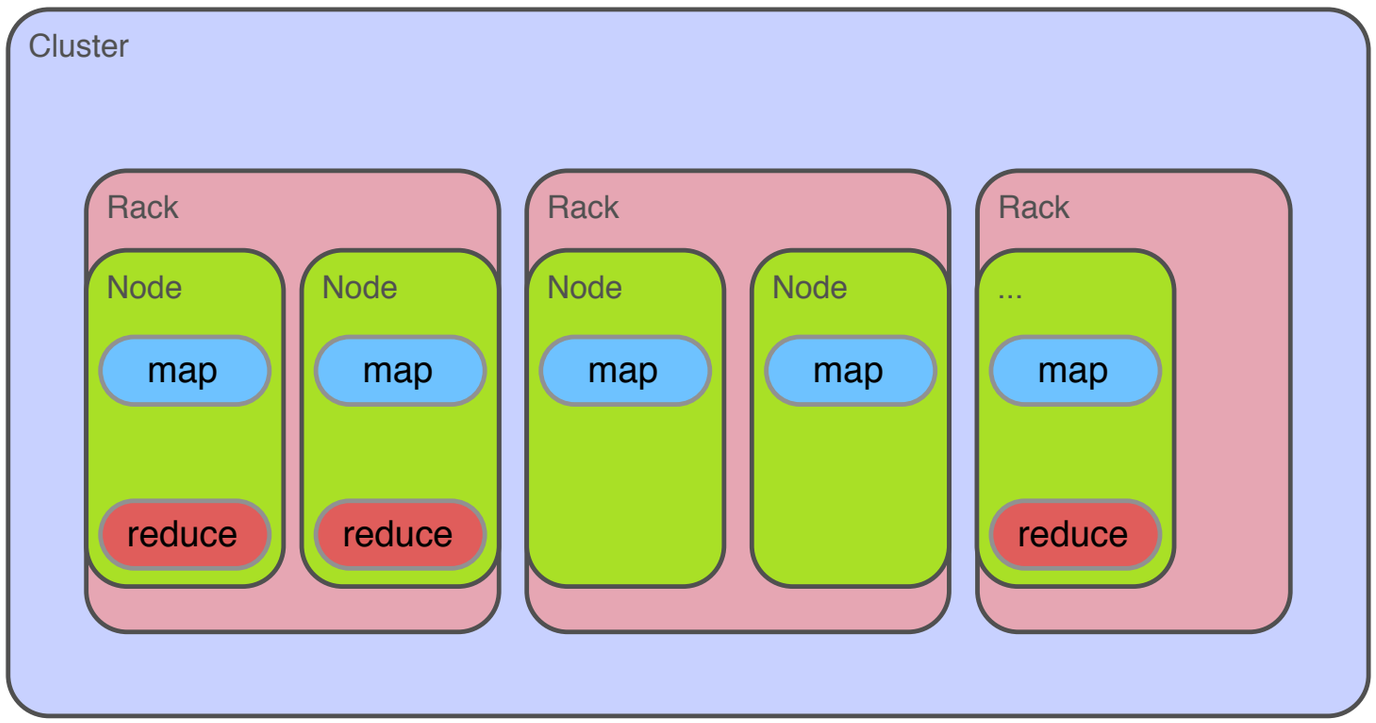
Divide and Conquer Parallelism

- Since the 'records' entering the Map and 'groups' entering the Reduce are independent

- That is, there is no expectation of order or requirement to share state between records/ groups

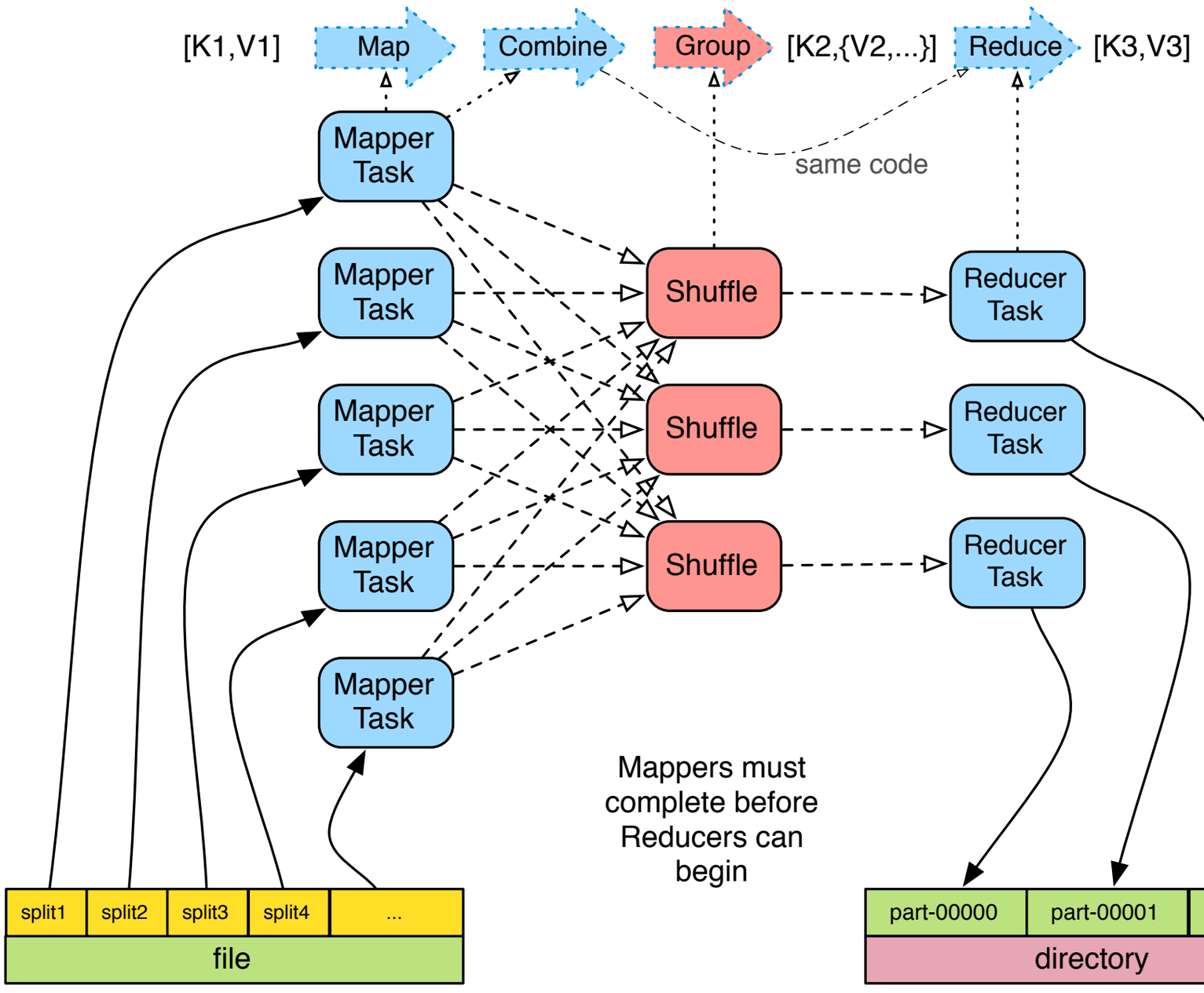
- Arbitrary numbers of Map and Reduce function instances can be created against arbitrary portions of input data

Cluster



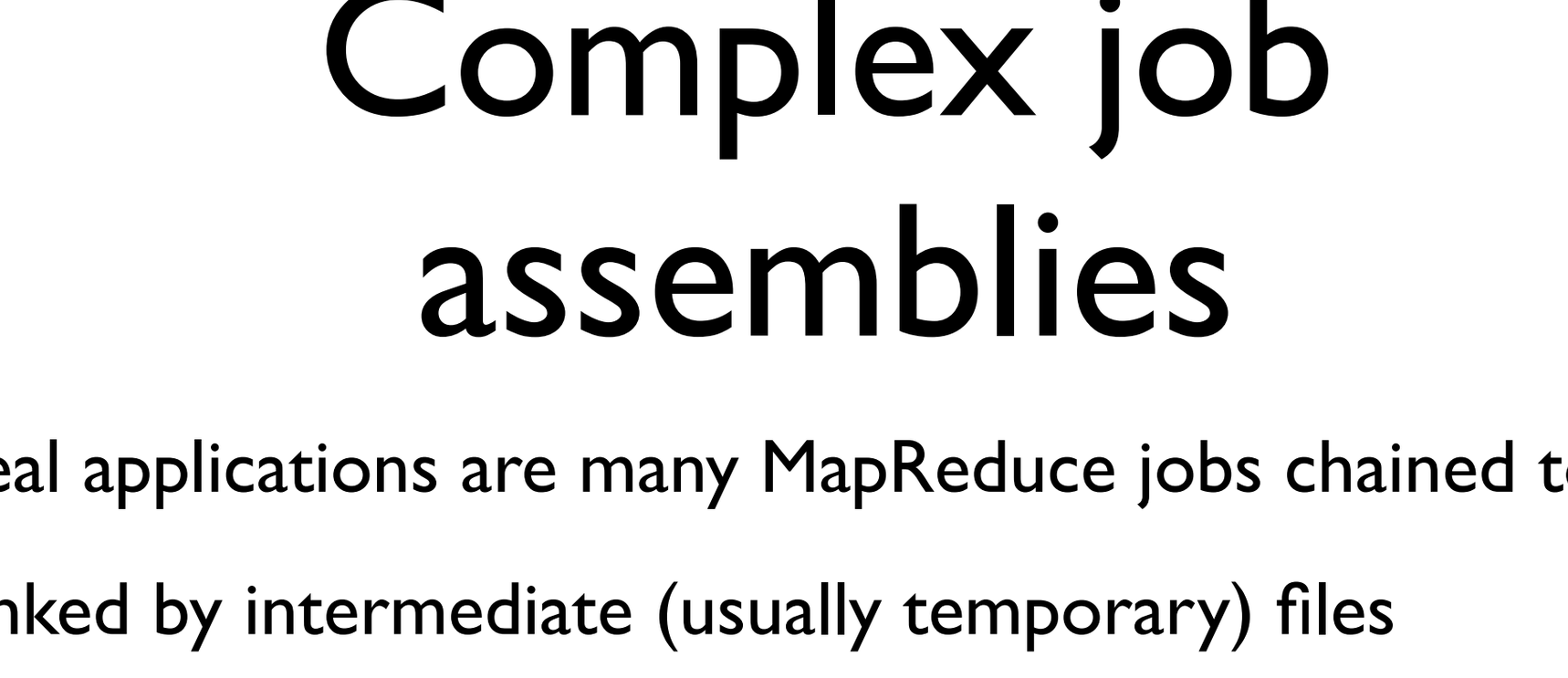
- Multiple instances of each Map and Reduce function are distributed throughout the cluster

Another View

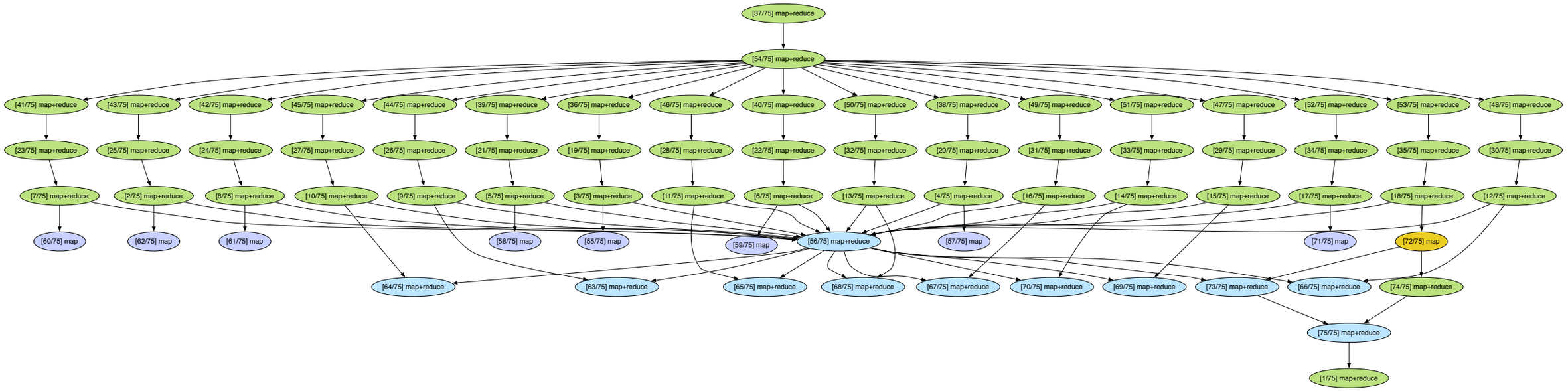


Complex job assemblies

- Real applications are many MapReduce jobs chained together
- Linked by intermediate (usually temporary) files
- Executed in order, by hand, from the 'client' application



Real World Apps



1 app, 75 jobs

green = map + reduce
purple = map
blue = join/merge
orange = map split

Heavy Lifting

- Thing we must do because data can be heavy
- These patterns are natural to MapReduce and easy to implement
- But have some room for composition/aggregation within a Map/Reduce (i.e., Filter + Binning)
- (leading us to think of Hadoop as an ETL framework)

- Record Filtering
- Parsing, Conversion
- Counting, Summing
- Unique
- Binning
- Distributed Tasks

Record Filtering

- Think unix ‘grep’

- Filtering is discarding unwanted values (or preserving wanted)

- Only uses a Map function, no Reducer

Parsing, Conversion

- Think unix 'sed'
- A Map function that takes an input key and/or value and translates it into a new format
- Examples:
  - raw logs to delimited text or archival efficient binary
  - entity extraction

Counting, Summing

- The same as SQL aggregation functions
- Simply applying some function to the values collection seen in Reduce
- Other examples:
  - average, max, min, unique

Merging

- Where many files of the same type are converted to one output path
- Map side merges
  - One directory with as many part files as Mappers
- Reduce side merges
  - Allows for removing duplicates or deleted items
  - One directory with as many part files as Reducers
- Examples
  - Nutch
  - Normalizing log files (apache, log4j, etc)

Binning

- Where the values associated w/ unique keys are persisted together
- Typically a directory path based on key's value
- Must be conscious of total open files, remember no appends
- Examples:
  - web log files by year/month/day
  - trade data by symbol

Distributed Tasks

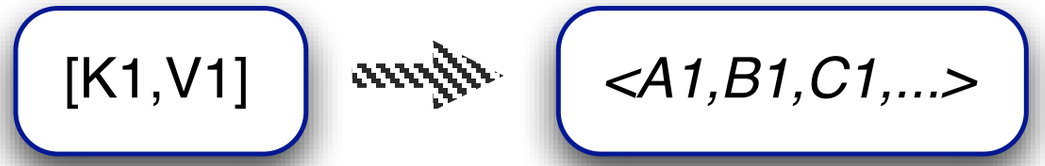
- Simply where a Map or Reduce function executes some ‘task’ based on the input key and value.
- Examples:
  - web crawling,
  - load testing services,
  - rdbms/nosql updates,
  - file transfers (S3),
  - image to pdf (NYT on EC2)

Basic Analytic Patterns

- Some of these patterns are unnatural to MapReduce
- We think in terms of columns/fields, not key value pairs
- (leading us to think of Hadoop as a RDBMS)

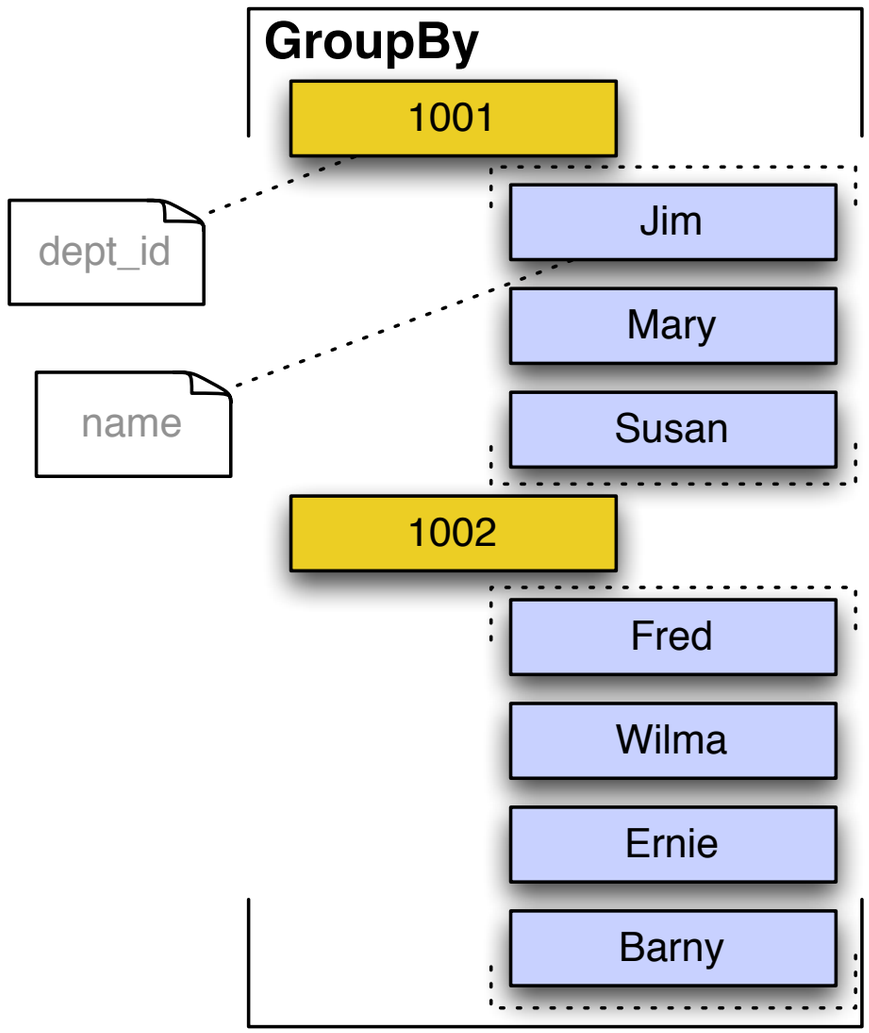
  - Group By
  - Unique
  - Secondary Sort
  - Secondary Unique
  - CoGrouping and Joining

Composite Keys/Values



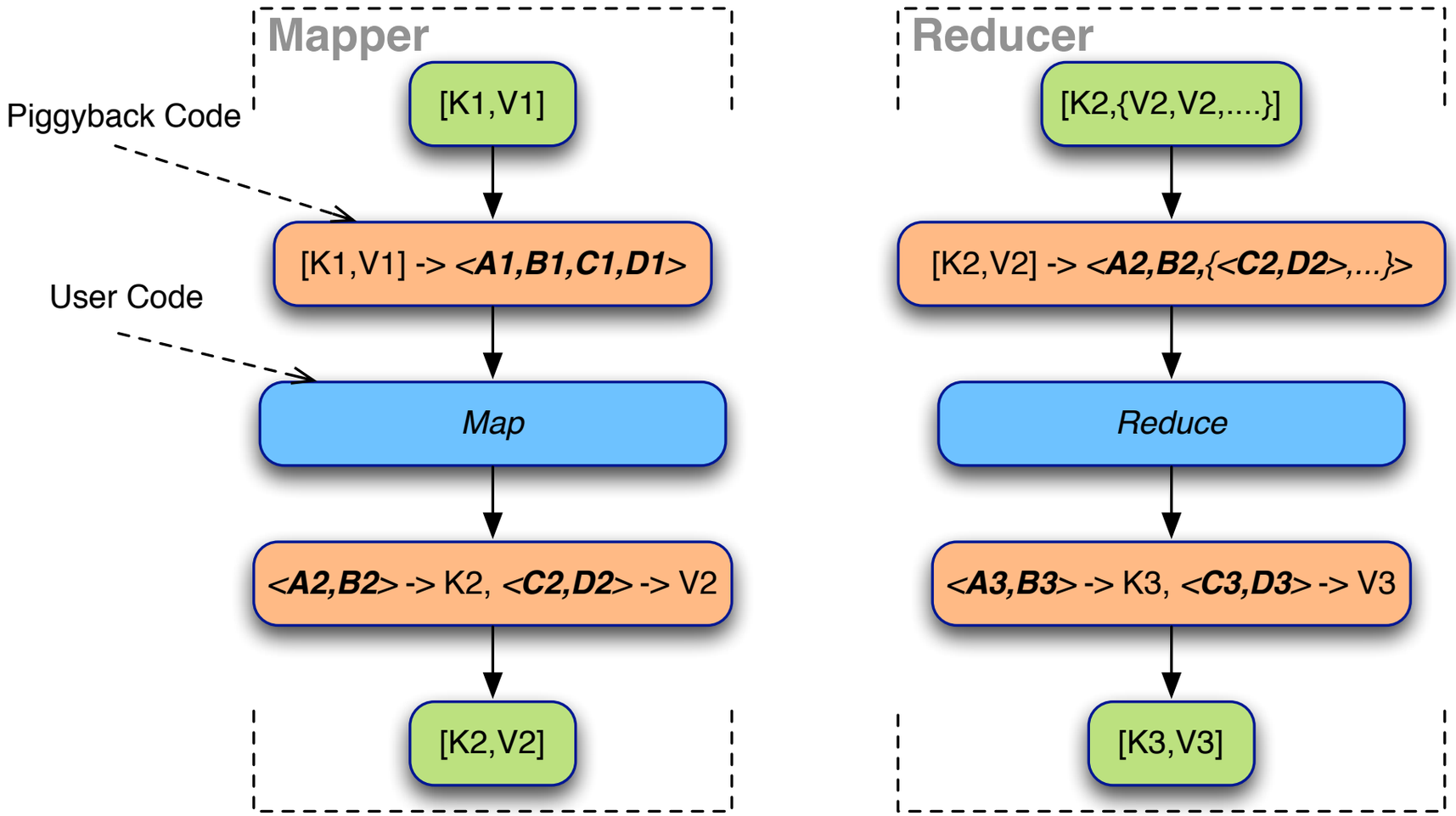
- It is easier to think in columns/fields
  - e.g. “firstname” & “lastname”, not “line”
- Whether a set of columns are Keys or Values is arbitrary
- Keys become a means to piggyback the properties of MR and become an impl detail

Group By



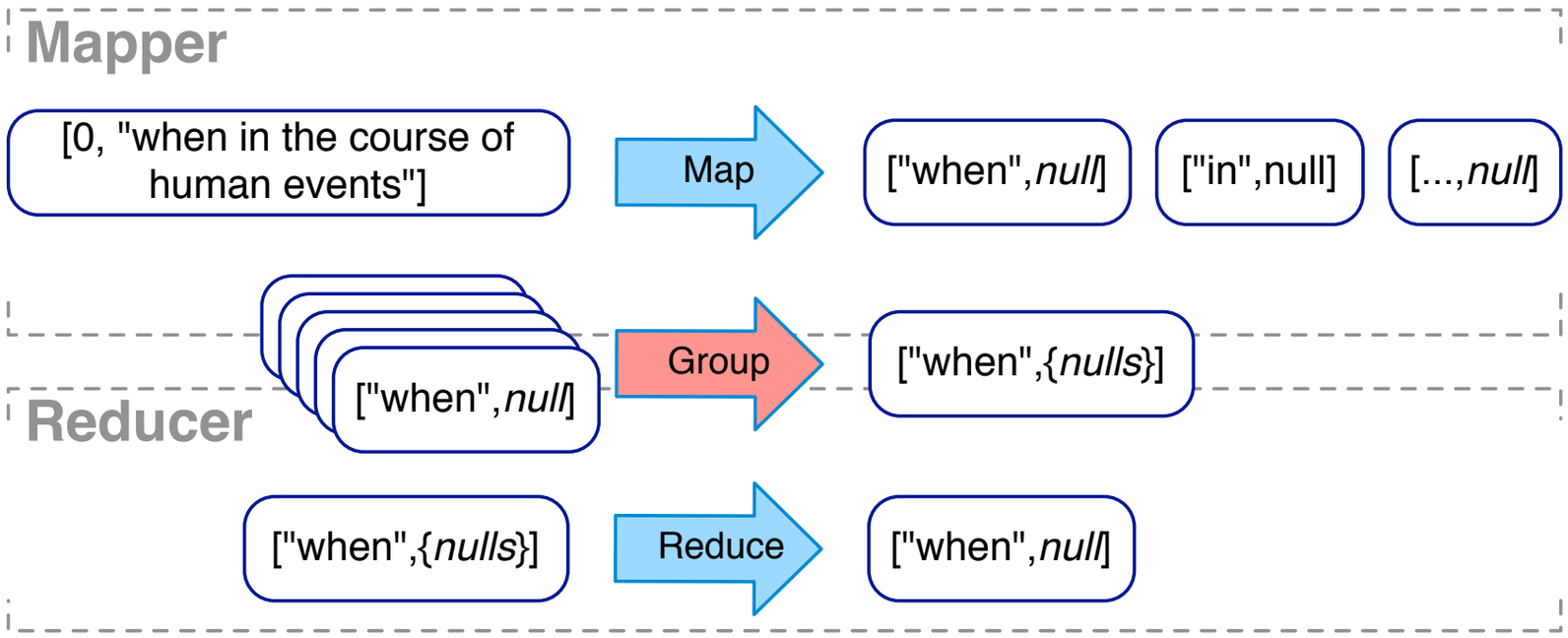
- Group By is where Value fields are grouped by Grouping fields
- Above, Map output key is “dept_id” and value is “name”

Group By



- So the K2 key becomes a **composite Key** of
  - **key**: [grouping], **value**: [values]

Unique



- Or Distinct (as in SQL)
- Globally finding all the unique values in a dataset
  - Usually finding unique values in a column
- Often used to filter a second dataset using a join

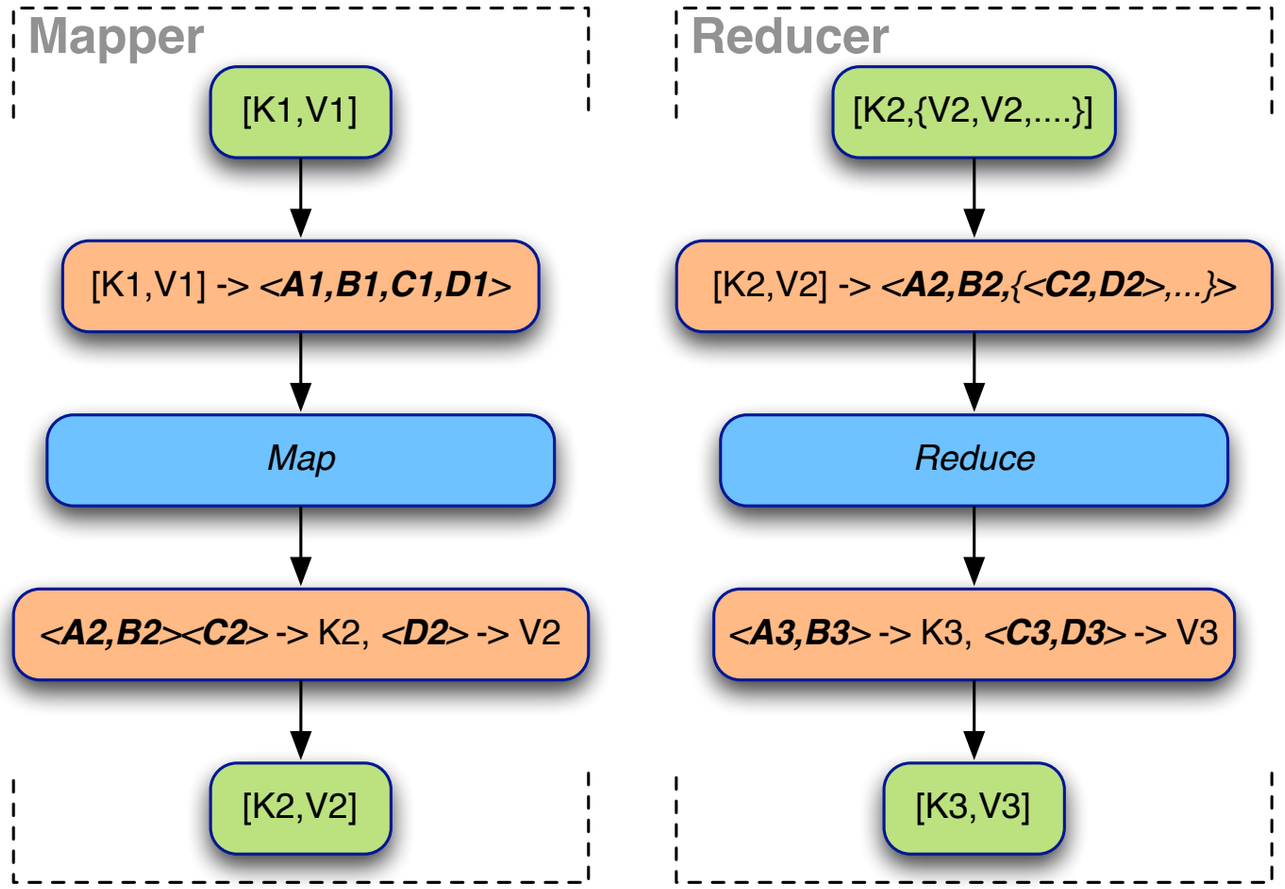
Secondary Sort

(group) *(sorted value)* *(remaining value)*

Date	*Time*	*Url*
08/08/2008,	1:00:00,	http://www.example.com/foo
08/08/2008,	1:01:00,	http://www.example.com/bar
08/08/2008,	1:01:30,	http://www.example.com/baz

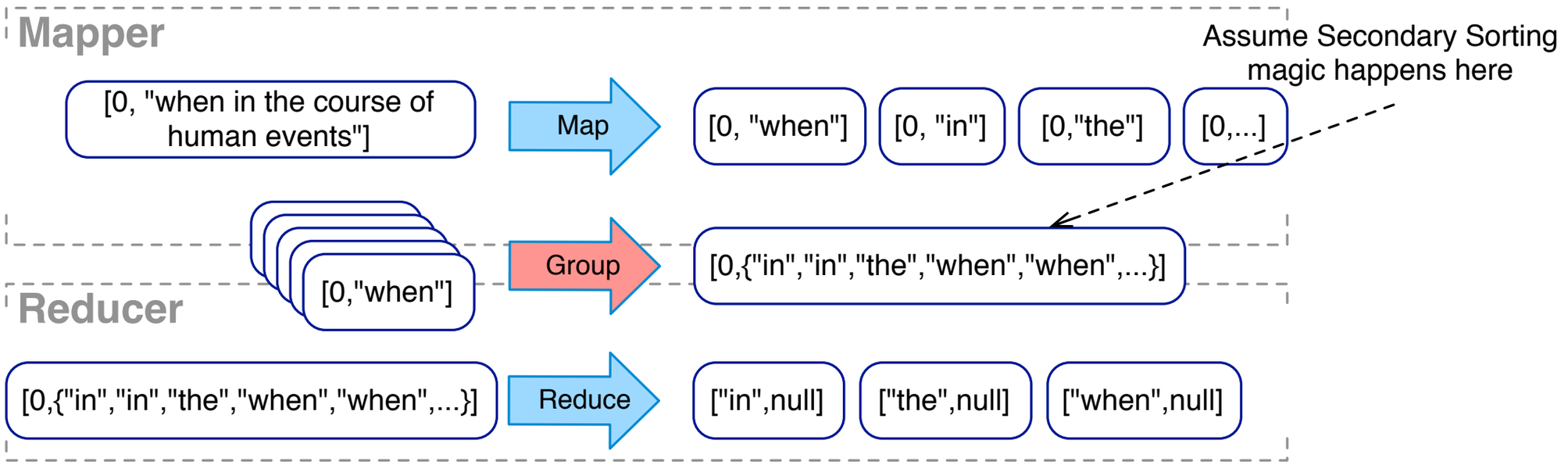
- Secondary Sorting is where
  - Some Fields are grouped on, and
  - Some of the remaining Fields are sorted within their grouping

Secondary Sort



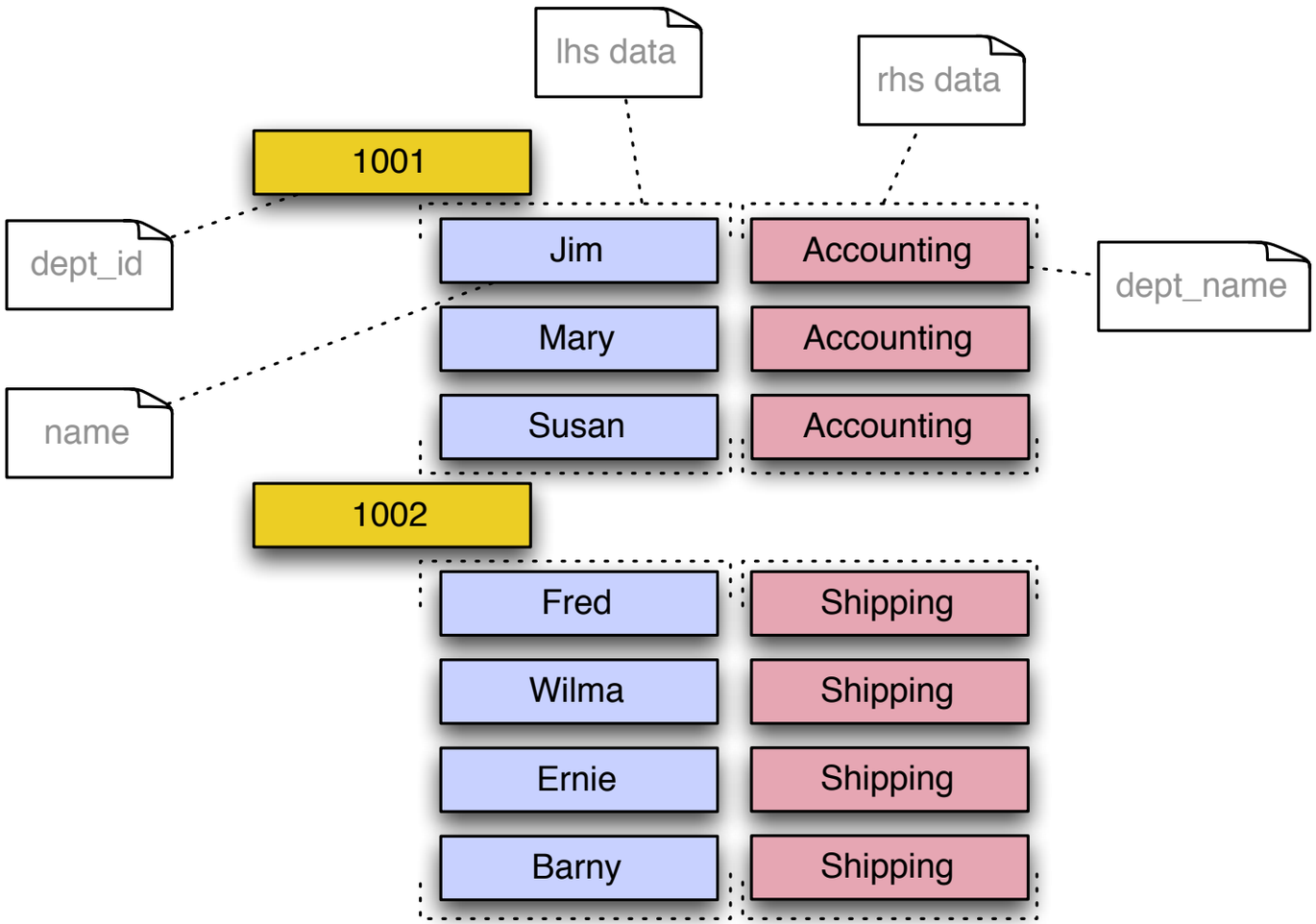
- So the K2 key becomes a **composite Key** of
  - **key**: [grouping, secondary], **value**: [remaining values]
- The trick is to piggyback the Reduce sort yet not be compared during the unique key comparison

Secondary Unique



- Secondary Unique is where the grouping values are uniqued
  - .... in a "scale free" way
- Perform a Secondary Sort...
- Reducer removes duplicates by discarding every value that matches the previous value
  - since values are now ordered, no need to maintain a Set of values

Joining



- Where two or more input data sets are 'joined' by a common key
  - Like a SQL join

Join Definitions

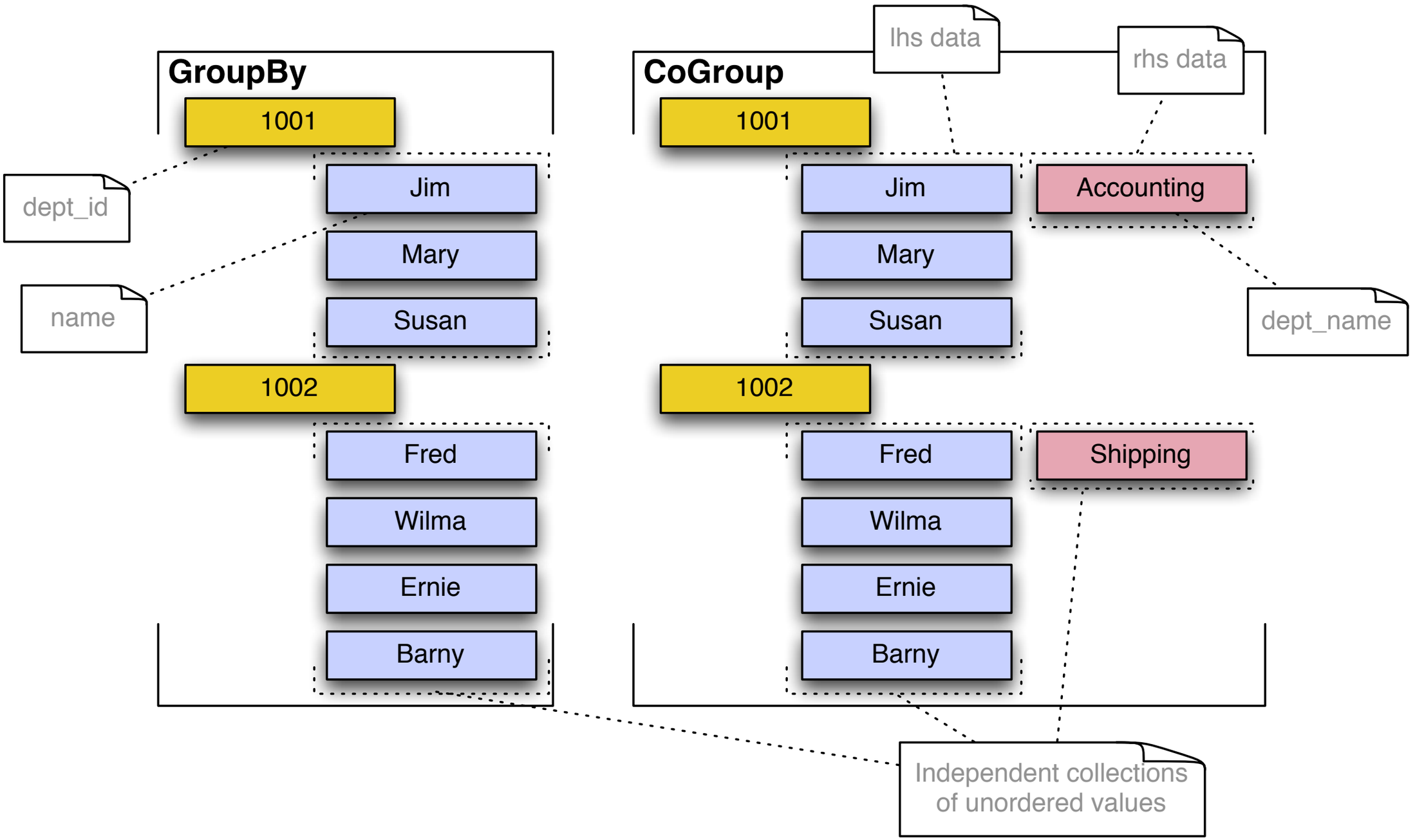
- Consider the input data **[key, value]**:
  - **`LHS = [0,a] [1,b] [2,c]`**
  - **`RHS = [0,A] [2,C] [3,D]`**
- Joins on the key:
  - Inner
    - **`[0,a,A] [2,c,C]`**
  - Outer *(Left Outer, Right Outer)*
    - **`[0,a,A] [1,b,null] [2,c,C] [3,null,D]`**
  - Left *(Left Inner, Right Outer)*
    - **`[0,a,A] [1,b,null] [2,c,C]`**
  - Right *(Left Outer, Right Inner)*
    - **`[0,a,A] [2,c,C] [3,null,D]`**

CoGrouping

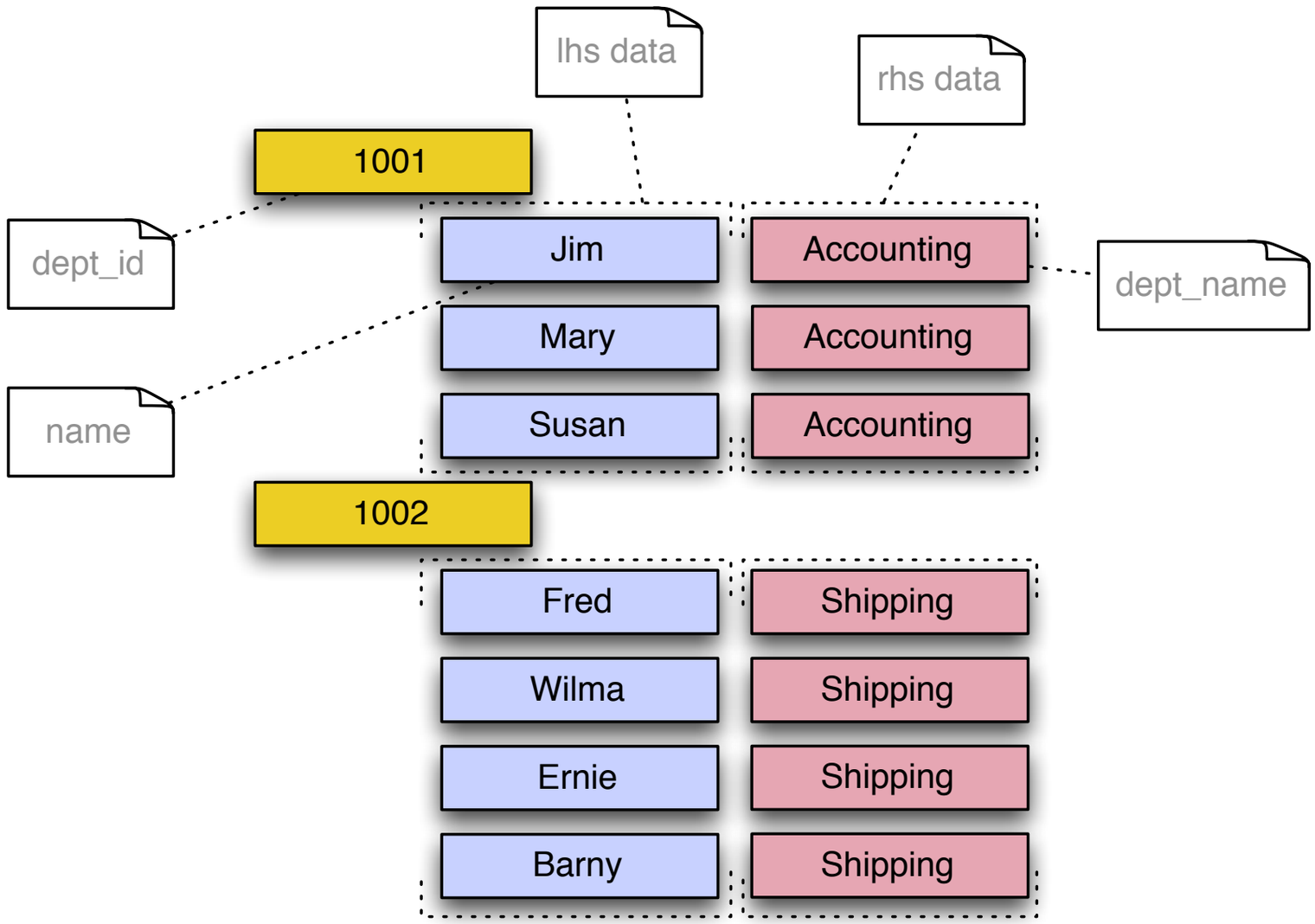
- Before Joining, CoGrouping must happen

- Simply concurrent GroupBy operations on each input data set

GroupBy vs CoGroup

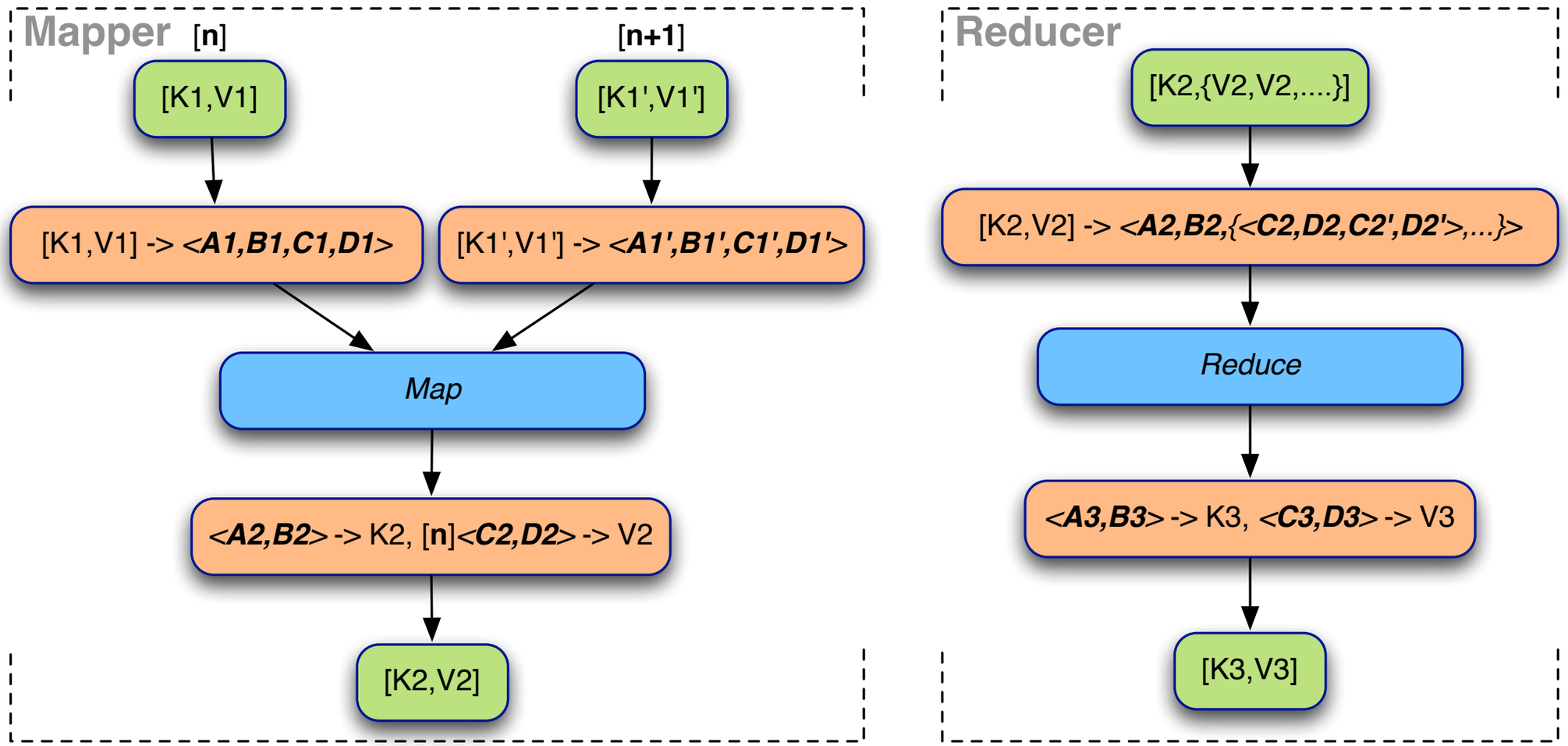


CoGroup Joined



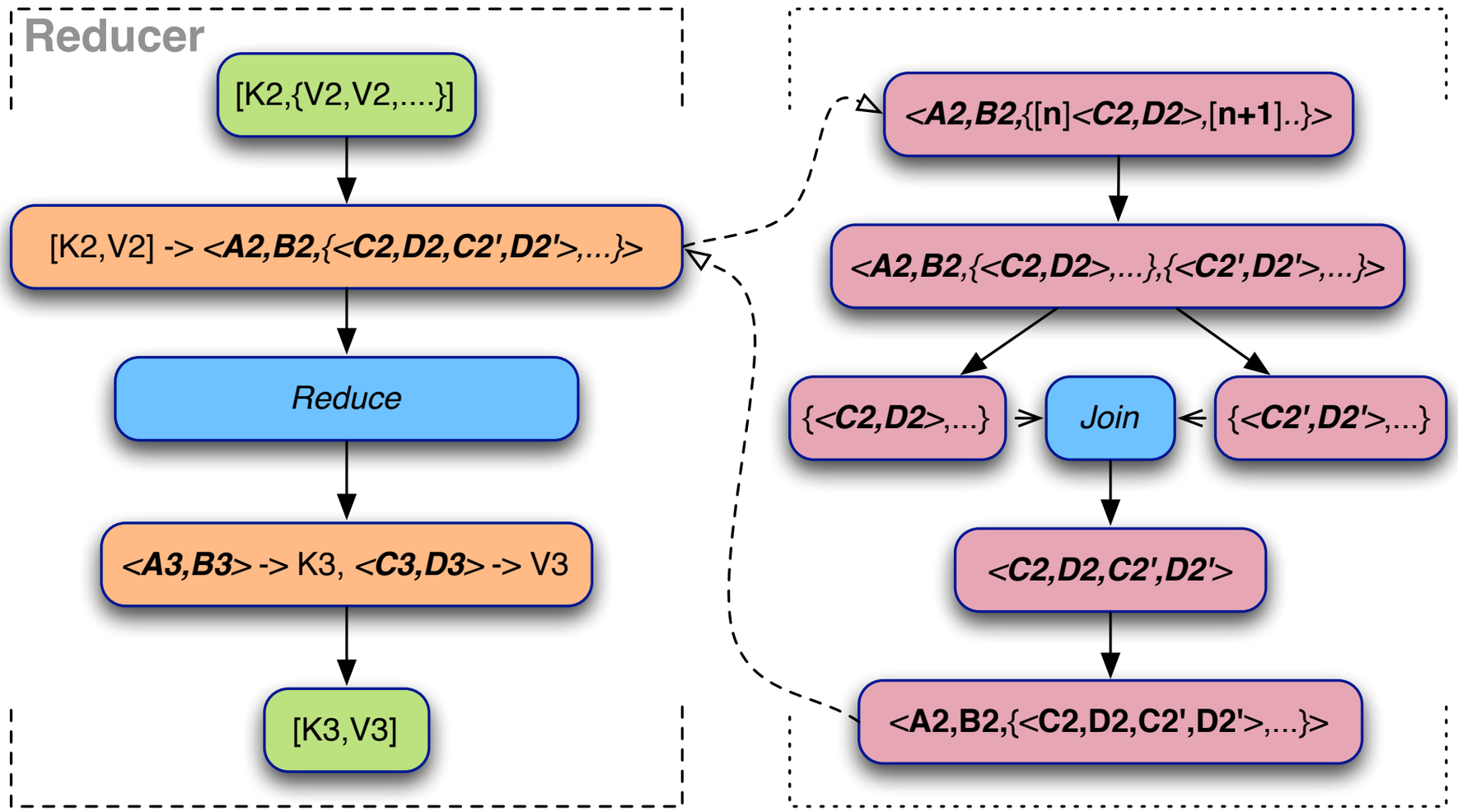
- Considering the previous data, a typical Inner Join

CoGrouping



- Maps must run for each input set in same Job (n, n+1, etc)
- CoGrouping must happen against each common key

Joining



- The CoGroups must be joined
- Finally the Reduce can be applied

Optimizations

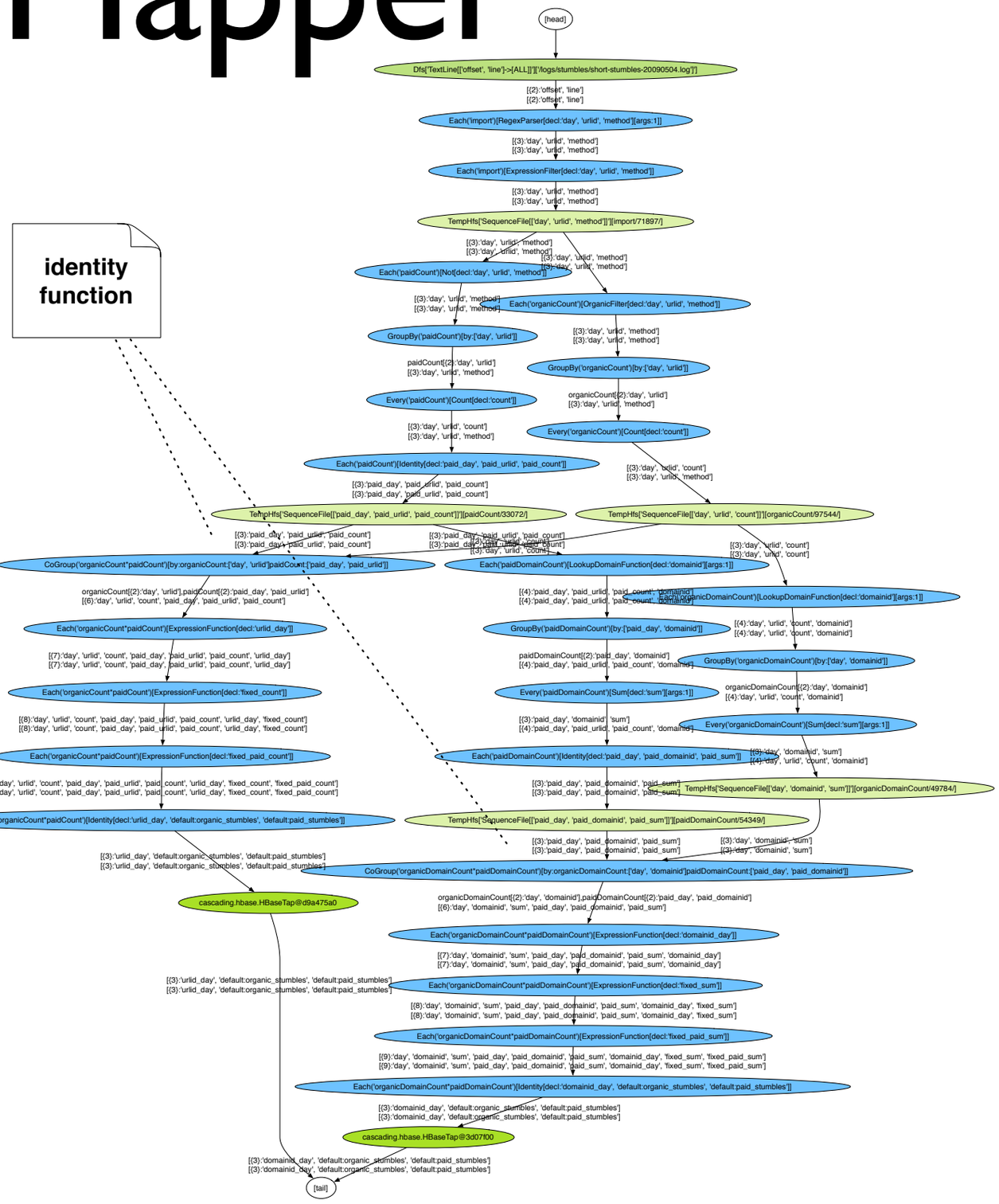
- Patterns for reducing IO

- Identity Mapper
- Map Side Join
- Combiners

- Partial Aggregates
- Similarity Joins

Identity Mapper

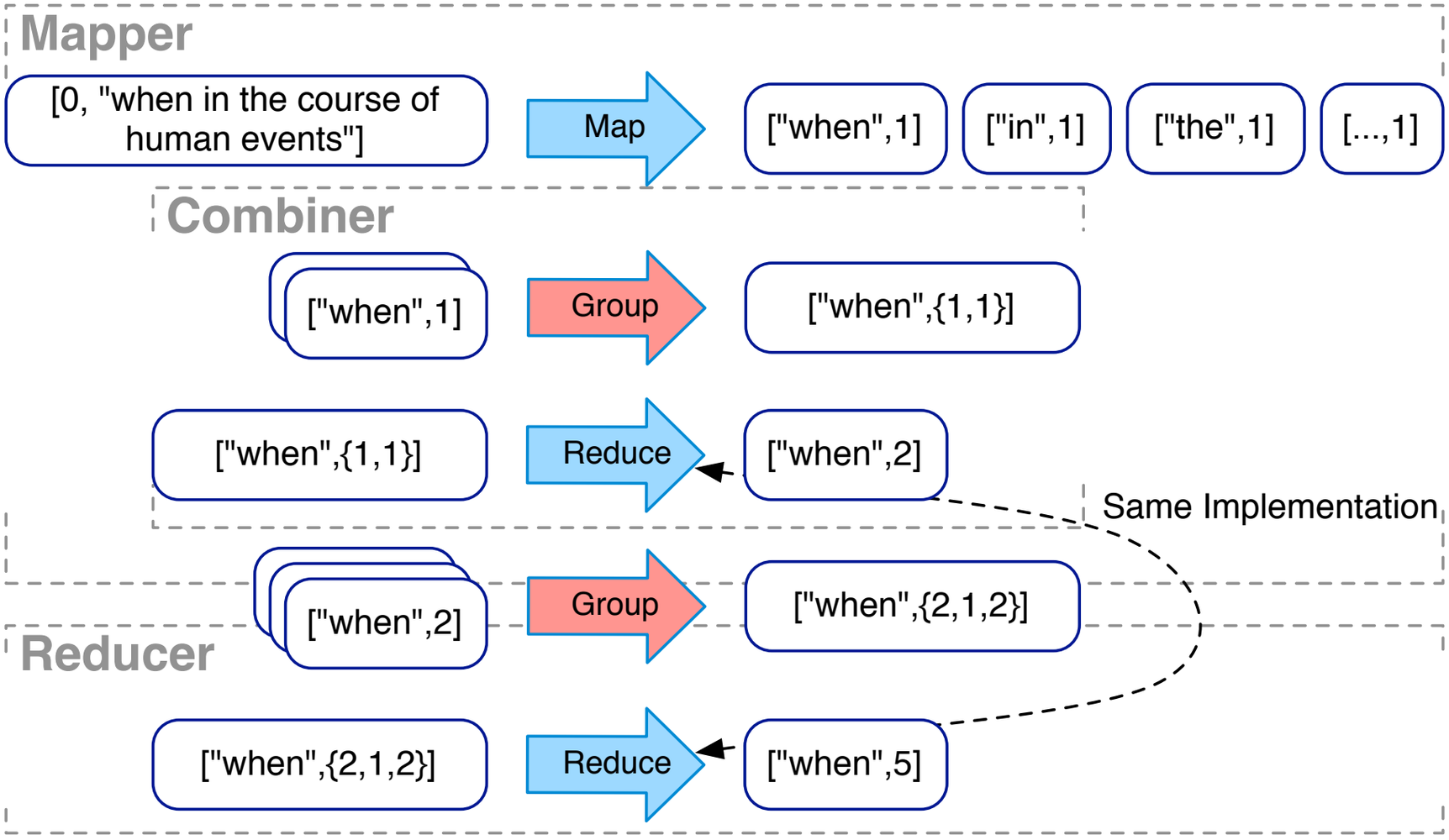
- Move Map operations to the previous Reduce
- Replace with an Identity function
- Assumes Map operations reduce the data



Map Side Joins

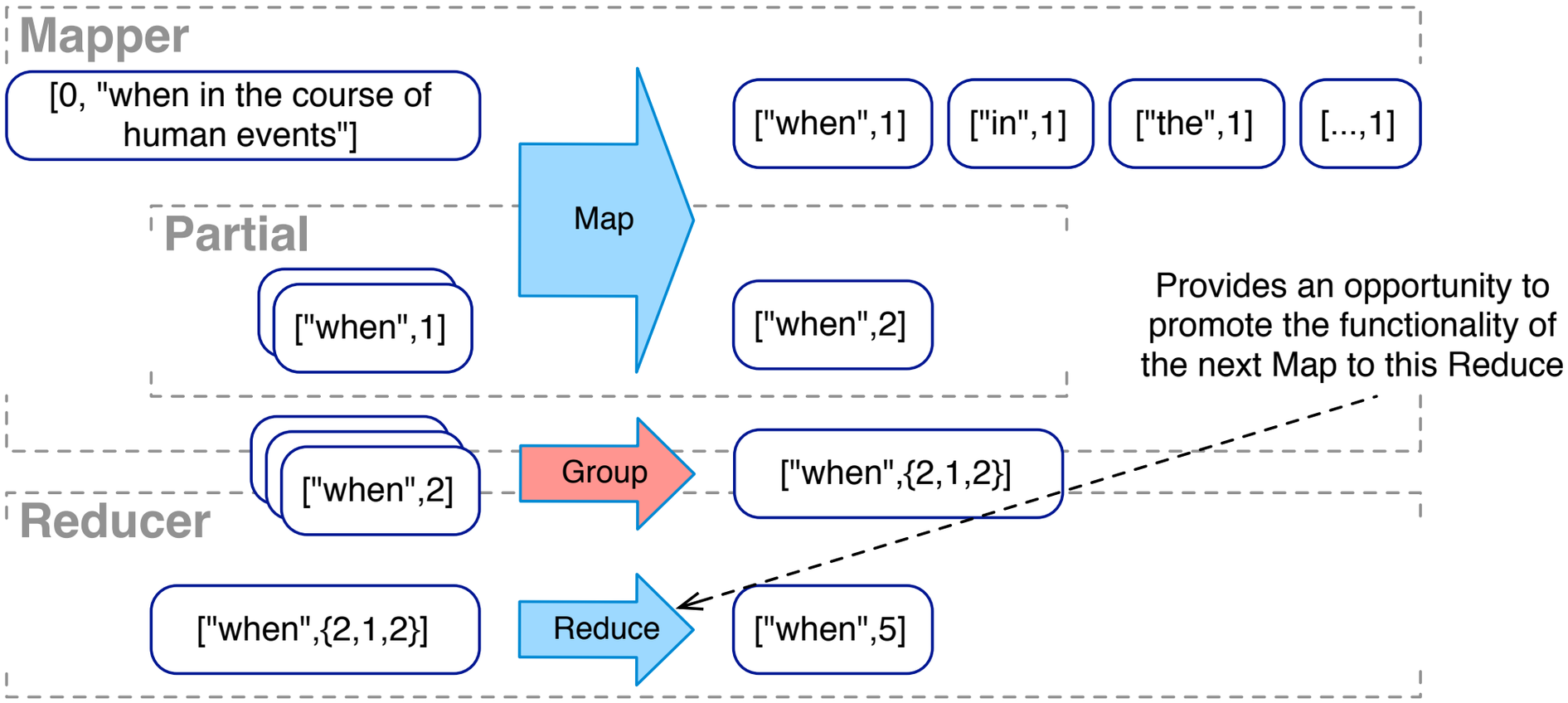
- Bypasses the (immediate) need for a Reducer
- Symmetrical
  - Where LHS and RHS are of equivalent size
  - Requires data to be sorted on key
- Asymmetrical
  - One side is small enough to fit in memory
  - Typically a hashtable lookup

Combiners



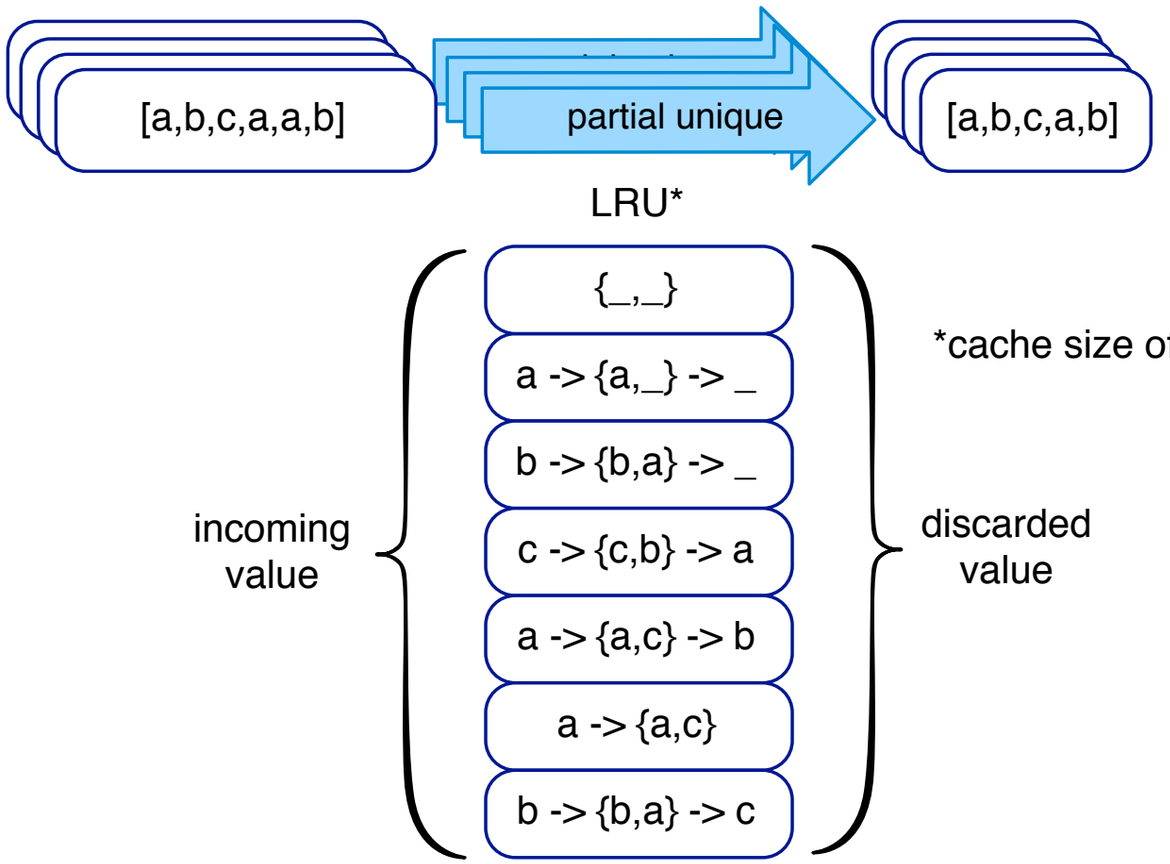
- Where Reduce runs Map side, and *again* Reduce side
- Only works if Reduce is *commutative* and *associative*
- Reduces bandwidth by trading CPU for IO
  - Serialization/deserialization during local sorting before combining

Partial Aggregates



- Supports any aggregate type, while being composable with other aggregates
- Reduces bandwidth by trading Memory for IO
  - Very important for a CPU constrained cluster
  - Use a bounded LRU to keep constant memory (requires tuning)

Partial Aggregates



- OK that dupes emit from a Mapper and across Mappers (or prev Reducers!)
- Final aggregation happens in Reducer
- Larger the cache, fewer dupes

Tradeoffs

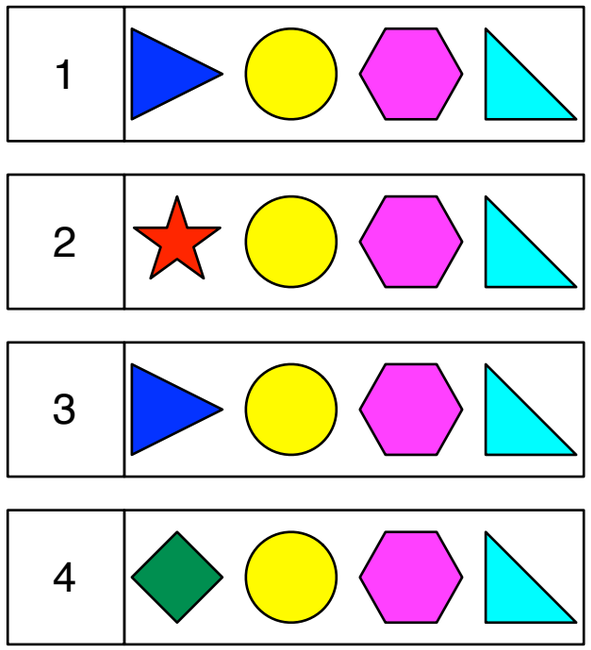
- CPU for IO == fault tolerance
- Memory for IO == performance

Similarity Join

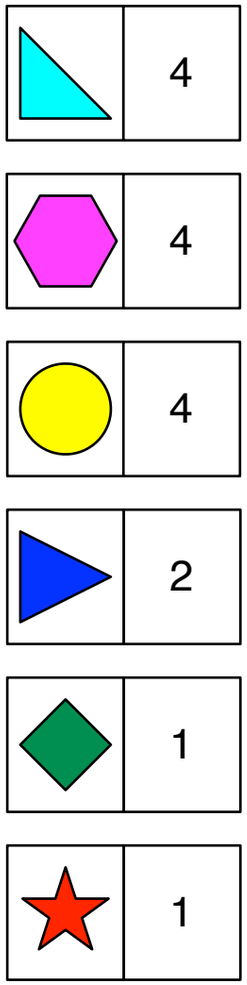
- Compare all values LHS to values RHS to find duplicates (or similar values)
- Naive approaches
  - Cross Join (all data through one reducer)
  - In-common features (very common features will bottleneck)

Set-Similarity Joining

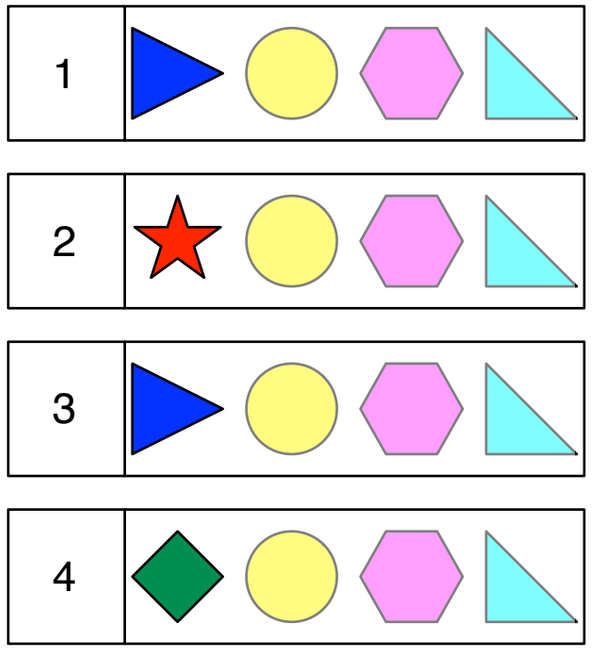
- "Efficient Parallel Set-Similarity Joins Using MapReduce" - R Vernica, M Carey, C Li
- Only compare candidate pairs
- Candidates share uncommon features



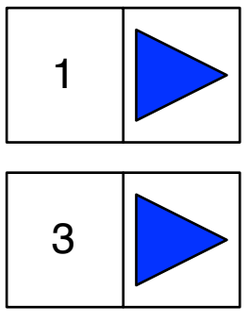
1: records



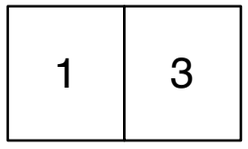
2: count tokens



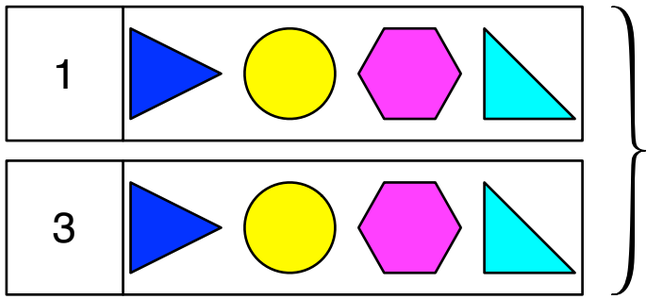
3: order by least frequent
discard common



4: uncommon features
in common



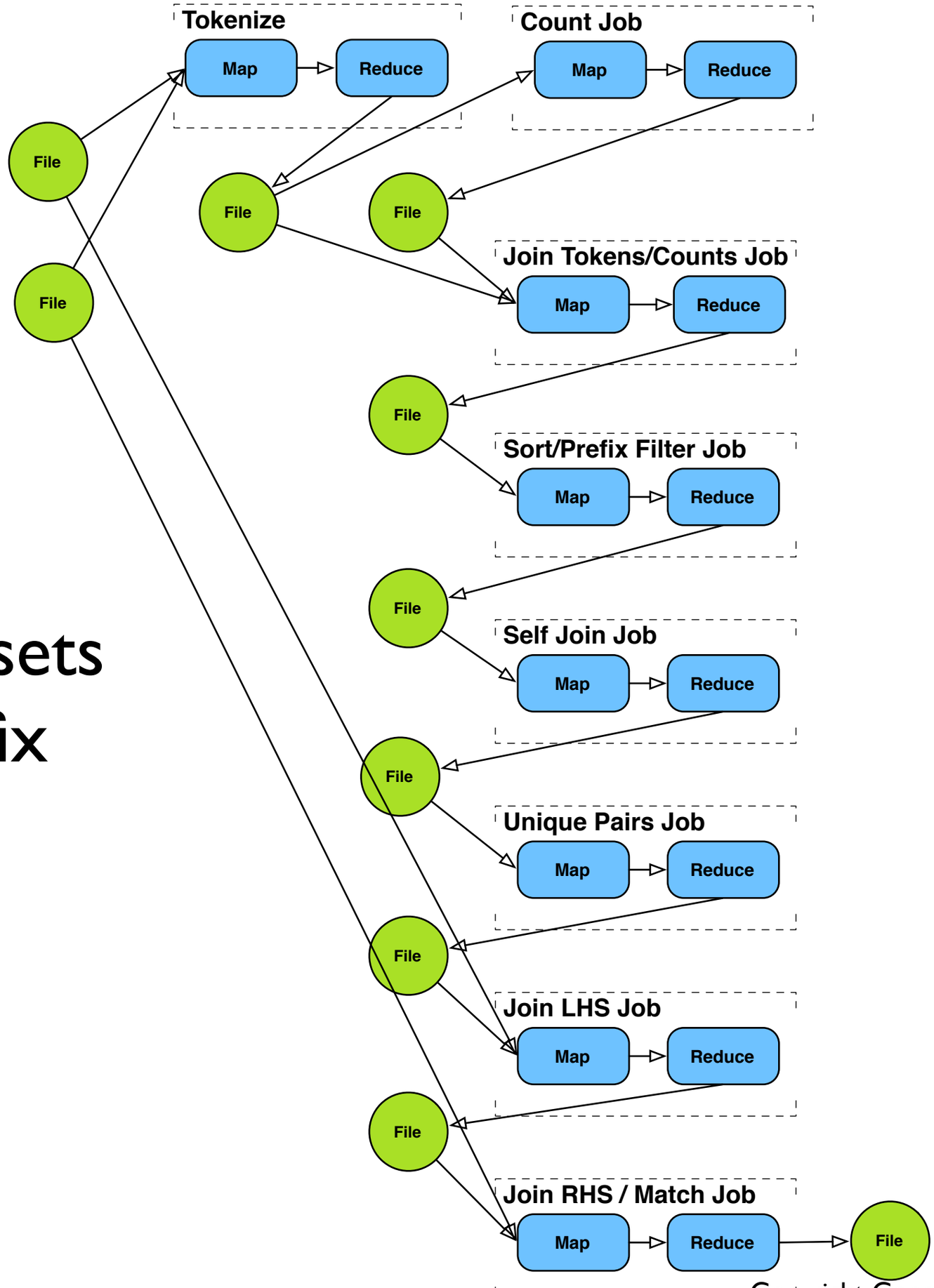
5: candidate pairs



6: final compare

- 1 and 3 share uncommon features
- thus are candidates for a full comparison

Match two sets using prefix filtering



Duality

- Note the use of the previous patterns to route data to implement a more efficient algorithm

Use a Higher Abstraction

- Command Line
  - Multitool - CLI for parallel sed, grep & joins

- API
  - Cascading - Java Query API and Planner
  - Plume - “approximate clone of FlumeJava”

- Interactive Shell
  - Cascalog - Clojure+Cascading query language (API also)
  - Pig - A text Syntax
  - Hive - Syntax + Infrastructure - SQL “like”

References

- Set Similarity
  - http://www.slideshare.net/ydn/4-similarity-joinshadoopsummit2010
  - http://asterix.ics.uci.edu/fuzzyjoin-mapreduce/

- MapReduce Text Processing
  - http://www.umiacs.umd.edu/~jimmylin/book.html

- Plume/FlumeJava
  - http://portal.acm.org/citation.cfm?id=1806596.1806638
  - http://github.com/tdunning/Plume/wiki

I'm Hiring

- Enterprise Java server and web client
- Language design, compilers, and interpreters
- No Hadoop experience required
- More info
  - http://www.concurrentinc.com/careers/

Resources

- Chris K Wensel
  - chris@wensel.net
  - @cwensel

- Cascading & Cascalog
  - http://cascading.org
  - @cascading

- Concurrent, Inc.
  - http://concurrentinc.com
  - @concurrent

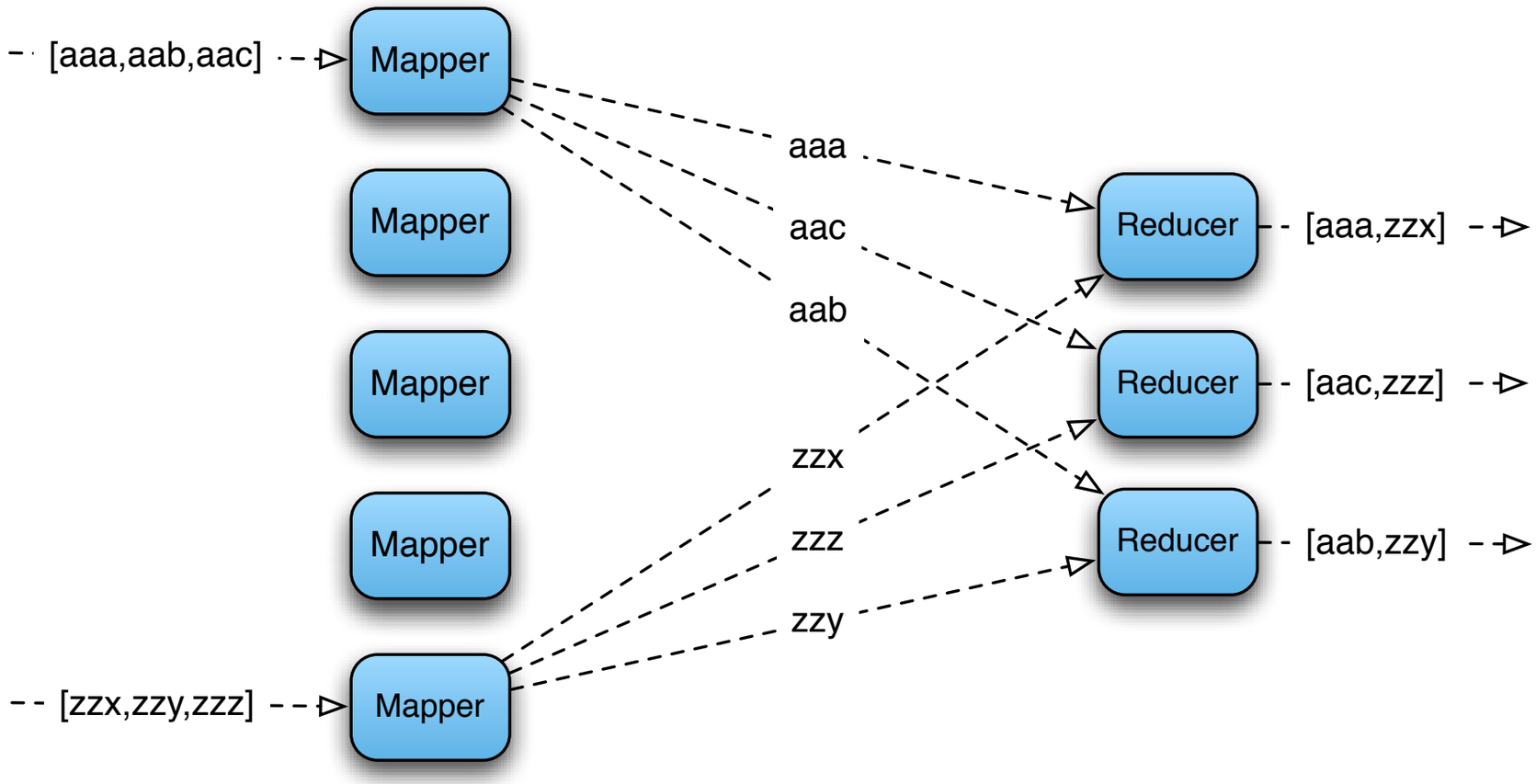
Appendix

Simple Total Sorting

- Where lines in a result file should be sorted

- Must set number of reducers to 1
  - Sorting in MR is local per Reduce, not global across Reducers

Why Sorting Isn't "Total"

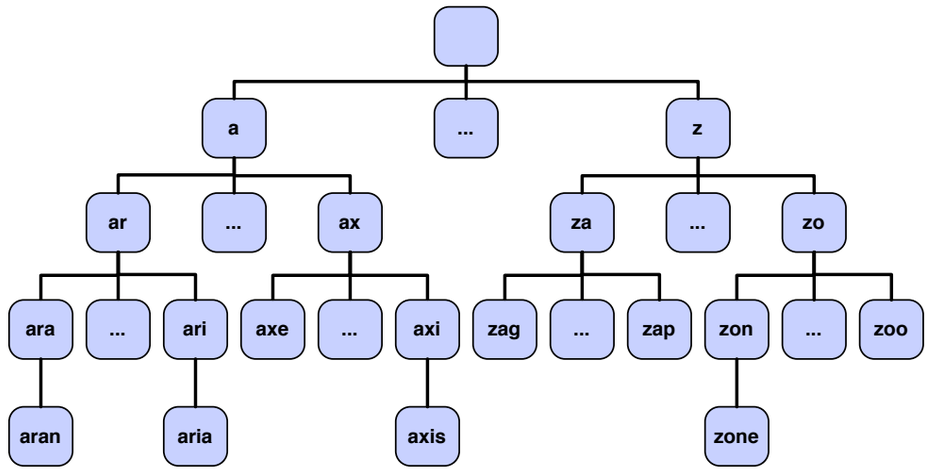


- Keys emitted from Map are naturally sorted at a given Reducer
- But are Partitioned to Reducers in a random way
- Thus, only one Reducer can be used for a total sort

Distributed Total Sort

- To work, the Shuffling phase must be modified with:
  - Custom Partitioner to partition on the *distribution* of ordered Keys
  - Custom Comparator for comparing Key types
    - Strings work by default

Distributed Total Sort - Details



- Sample all K2 values and build balanced distribution for num reducers
  - Sample all input keys and divide into partitions
  - Write out boundaries of partitions
- Supply Partitioner that looks up partition for current K2 value
  - Read boundaries into a Trie (pronounced 'try') data structure
- Use appropriate Comparator for Key type