Map Reduce

Priya Sudendra

What is Map Reduce?

- Programming model for processing large data sets with a parallel, distributed algorithm on a cluster
- Program is composed of a **Map()** procedure that performs filtering and sorting and a **Reduce()** procedure that performs a summary operation
- "MapReduce System" ("infrastructure" or "framework") works by marshaling distributed servers
    - Runs tasks in parallel
    - Manages all communications & data transfers between parts of system
    - Provides for redundancy and fault tolerance

Why Map Reduce?

- Bring processing to data
- Historically: programs have retrieved data from databases & then perform computations on separate server
    - OK for small amounts of data, but not modern data set scales
- Programming model that allows for parallelization
- Map-Reduce frameworks do all heavy lifting on running code across clusters of servers and ensuring it completes (i.e. handling failure)

Clusters

- If you are storing lots of data on clusters, have to organize processing differently so you can do it efficiently
- Allows for use of machines to spread the computation over
- Still need to try to reduce amount of data that needs to be transferred across the network by doing as much processing as you can on the same node as the data it needs
- Map-reduce pattern: way to organize processing, take advantage of multiple machines on cluster
  - Keep as much processing and the data it needs together on the same machine.

Map()

- Goal: Produce key/value pairs
- Each map function independent of others
    - Can be safely parallelized
- Data is co-located with processing
- Can be any arbitrarily complex function

Reduce()

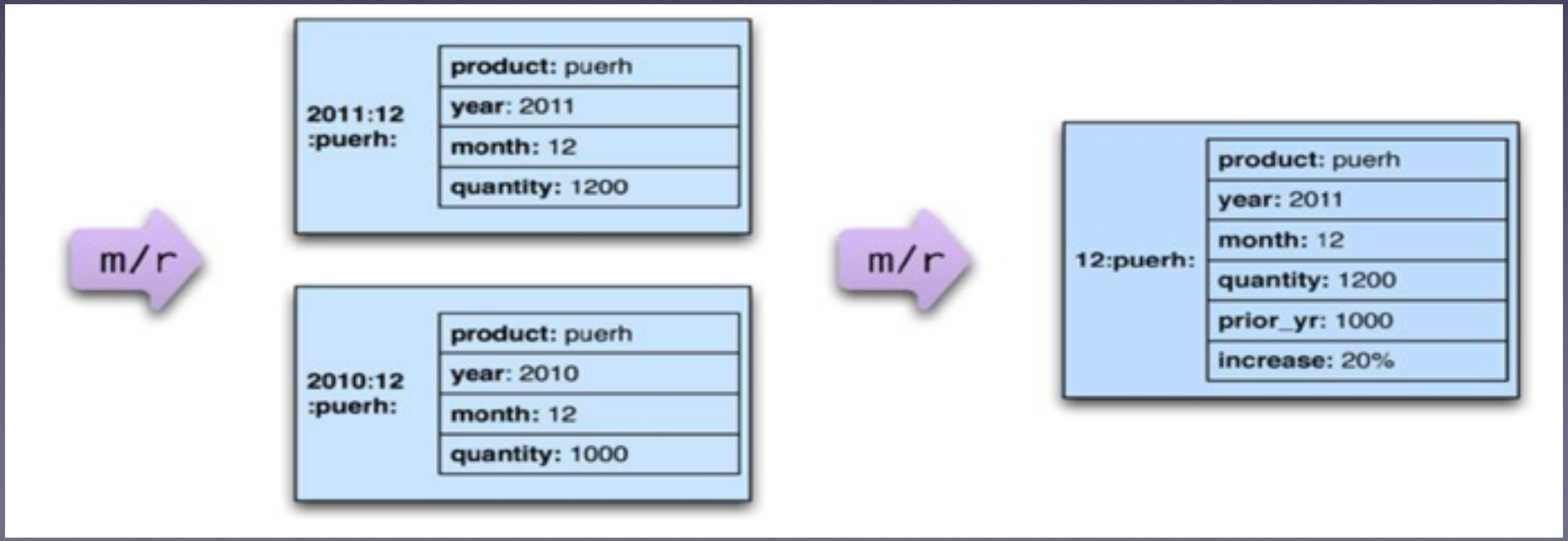
- Map() can only work on data from single aggregate,
- Reduce function can use all values emitted for a single key
- Map-reduce framework arranges for map tasks to be run on correct nodes to process all documents & for data to be moved to the reduce function
- Framework collects all values for a single pair & calls reduce function once with key and collection of all the values for that key

Composing Map-Reduce

- Way of thinking about concurrent processing that trades off flexibility in how you structure your computation for a relatively straightforward model for parallelization
- Within a map task, you can only operate on a single aggregate
- Have to structure your calculations around operations that fit in well with the notion of a reduce operation
- Example: Averages
    - Not composable: can’t combine separate groups’ averages alone
    - Need to take total amount & count of orders from each group, combine, and calculate the average from combined sum & count

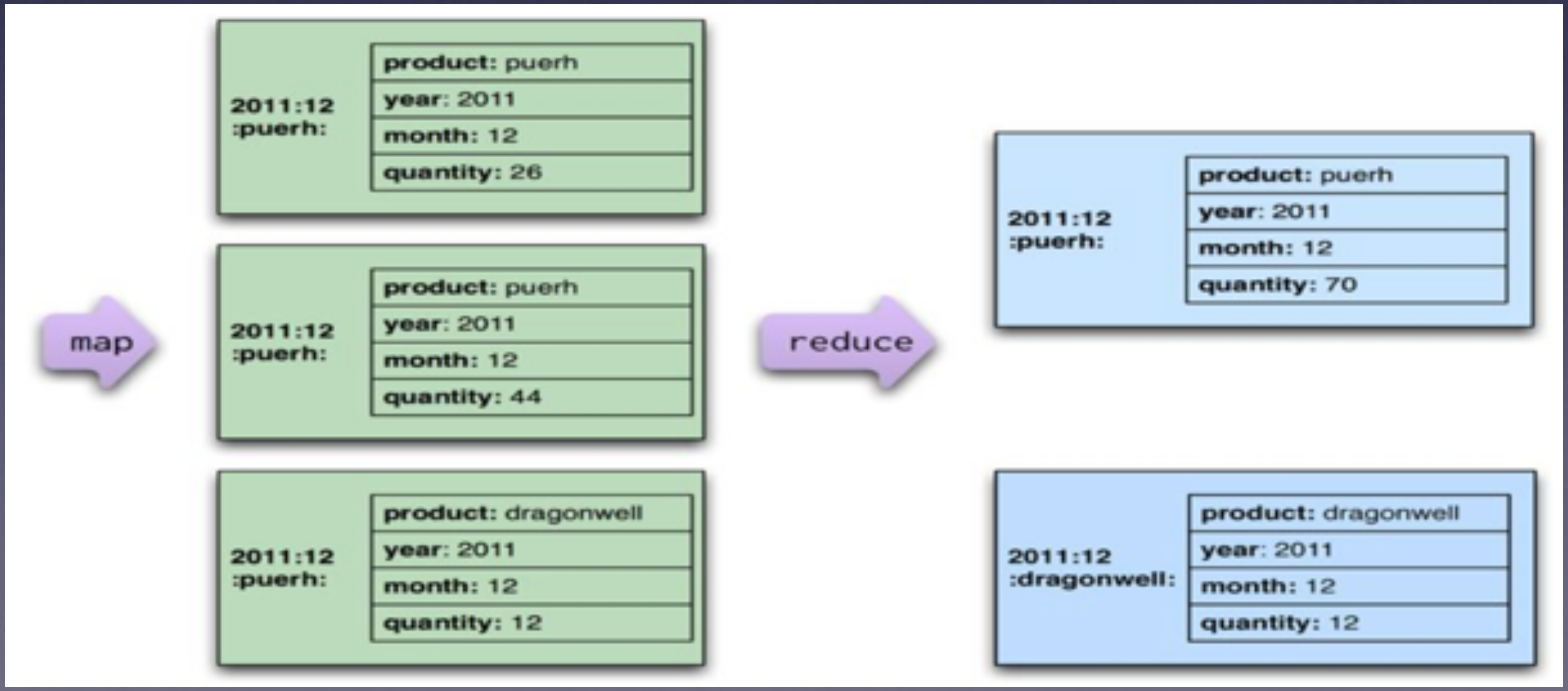
Two Stage Map Reduce

- As calculations get more complex, break them down into stages using a pipes-and-filters approach
    - output of one stage is input to the next



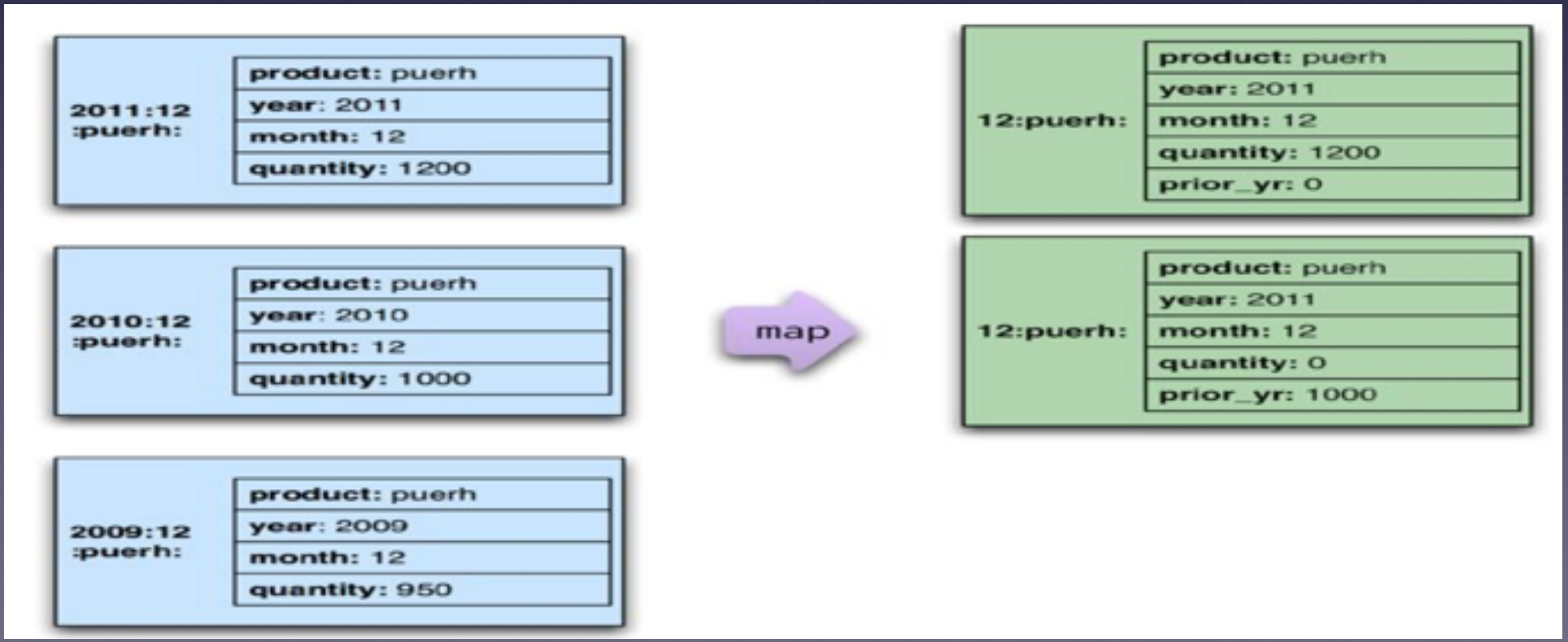
Two Stage Map Reduce

- Stage 1: Creating records for a monthly sales of a product.



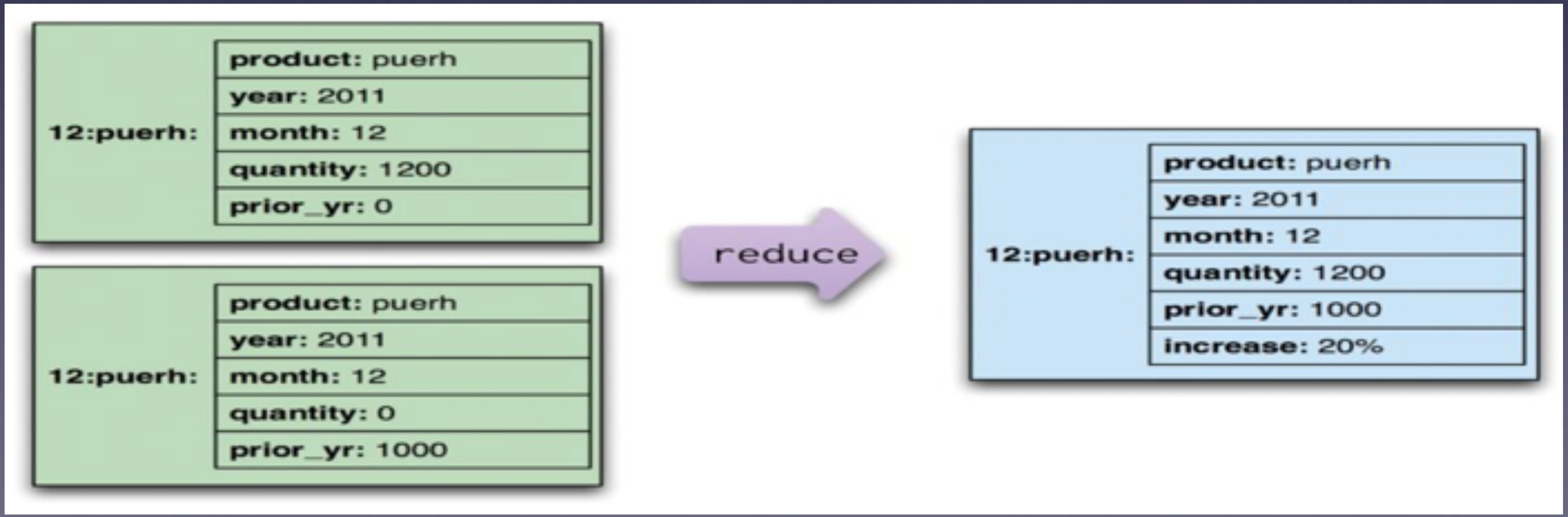
Two Stage Map Reduce

- Stage 2: Creates base records for year-on-year comparisons



Two Stage Map Reduce

- Stage 3: reduction step, merge of incomplete records.



Why?

- Decomposing report into multiple map-reduce steps makes it easier to write
- Once you’ve found a transformation framework that makes it easy to compose steps, it’s easier to compose many small steps together than try to cram heaps of logic into a single step
- Intermediate output may be useful for different outputs too, can get some reuse.
- Reduce operations often represent the heaviest amount of data access

Incremental Map Reduce

- Many map-reduce operations take a while to perform
  - New data keeps coming in, need to recompute.
  - Usually better to structure computations to allow for incremental updates
- Map stages are easy to handle incrementally
- Maps are isolated from each other, incremental updates straightforward
- Reduce more complex: pulls together outputs from many maps and chances to any of the map outputs can trigger a new reduction
  - If we partitioned the data for reduction, any unchanged partition does not need to be touched
- If our reducer is combinable, there are more opportunities for reducing computation
- If changes are additive, then we can just run the reduce with the existing result and new additions.
- If the changes are destructive (i.e. updates/deletes), then we can avoid computation by breaking up the reduce operation into steps and only recalculating steps whose inputs have changed