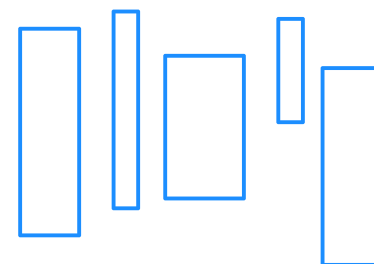


# Cascading

[www.cascading.org](http://www.cascading.org)

Chris K Wensel  
[chris@concurrentinc.com](mailto:chris@concurrentinc.com)



Concurrent, Inc.  
[www.concurrentinc.com](http://www.concurrentinc.com)

# In a nutshell

- An explicit way of describing ‘how’ data should be processed, independently from ‘what’ data will be processed.
- Process definitions are a pipeline of operations applied to a stream of tuples moving through it.
- At runtime, the process definition is combined with the data sources.
- The result is a set of interdependent MapReduce jobs, scheduled by dependency.
- A set of these resulting processes can be further organized and scheduled by dependencies.

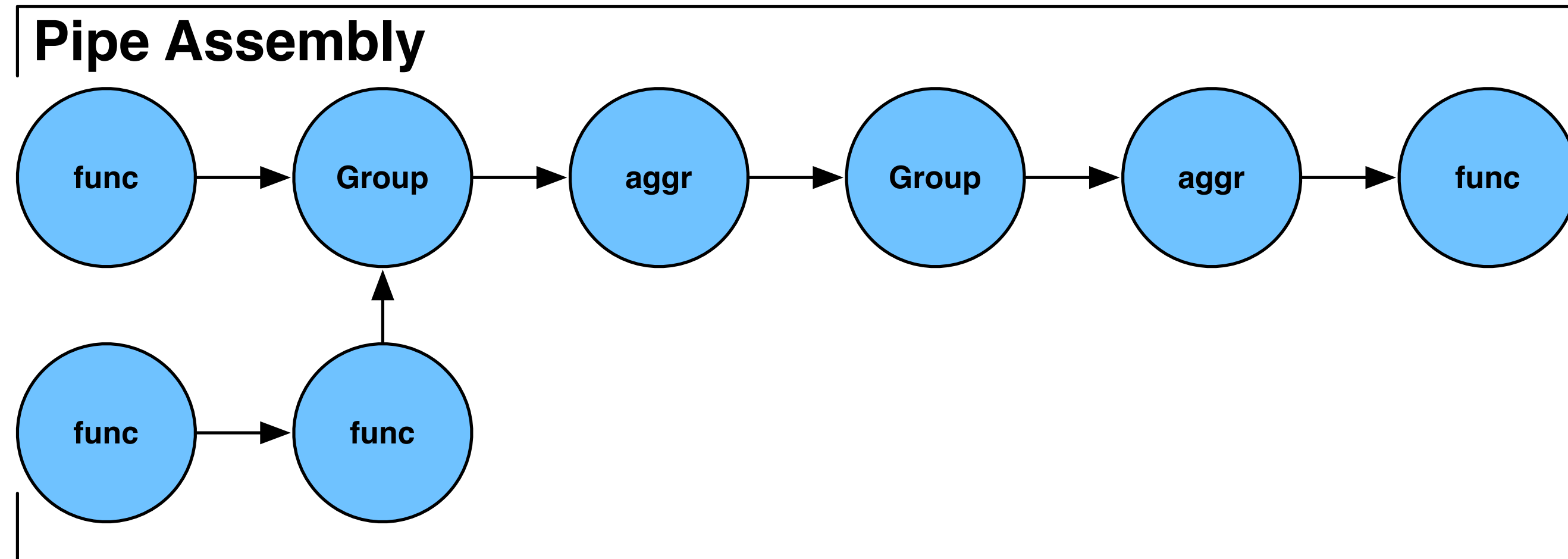
# Why this is good

- No need to think in MapReduce
- Processing definitions are composable and reusable
- Result data-sets defined by process definitions can be lazily (re)evaluated, or 'cached' for other processes

# Features

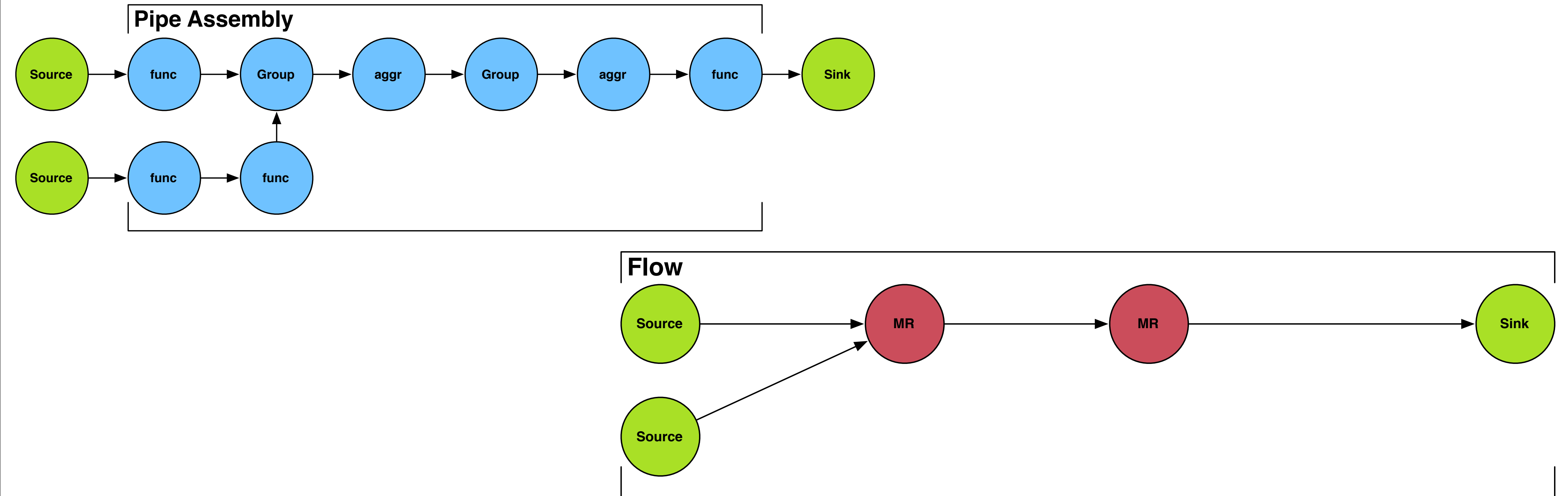


# Processing API



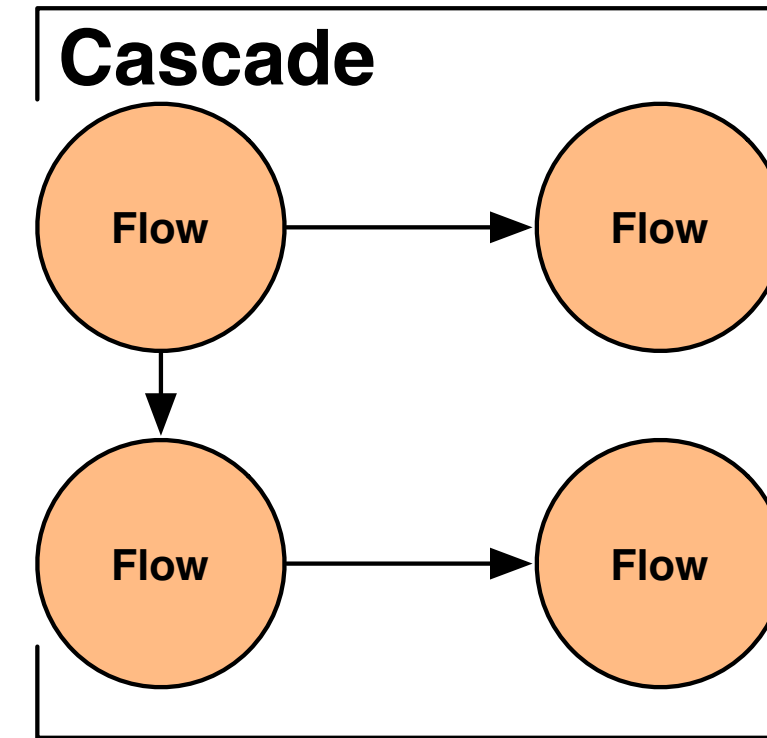
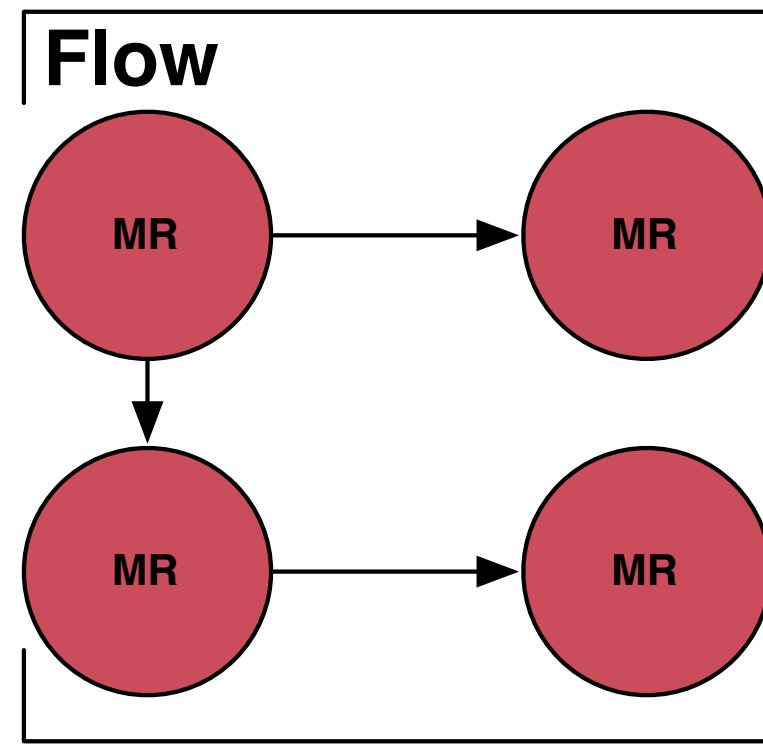
- Operations are chained together to define a Pipe assembly or a reusable sub-assembly

# Job Planner



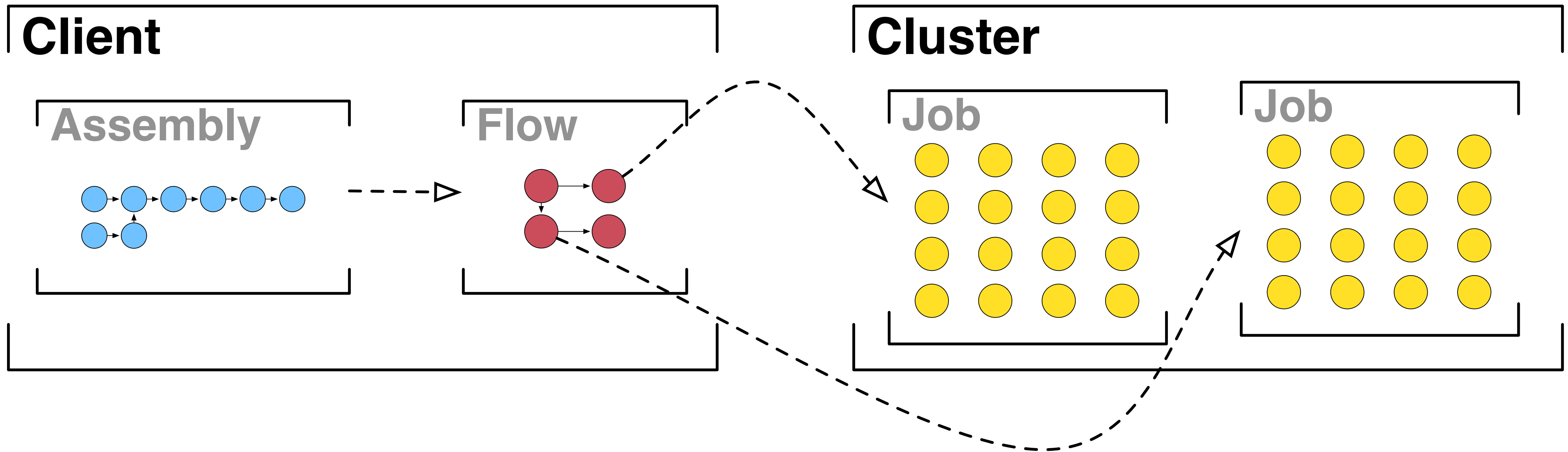
- Pipe Assemblies become Flows
- Translates a DAG of operations to a DAG of MapReduce jobs

# Topological Scheduler



- All MapReduce jobs in Flow scheduled in dependency order
- Flows can be combined into single process, a Cascade
- Flows scheduled by Cascade in dependency order
- Cascade will run only 'stale' Flows (pluggable behavior)
- Custom MapReduce jobs can participate in Cascade

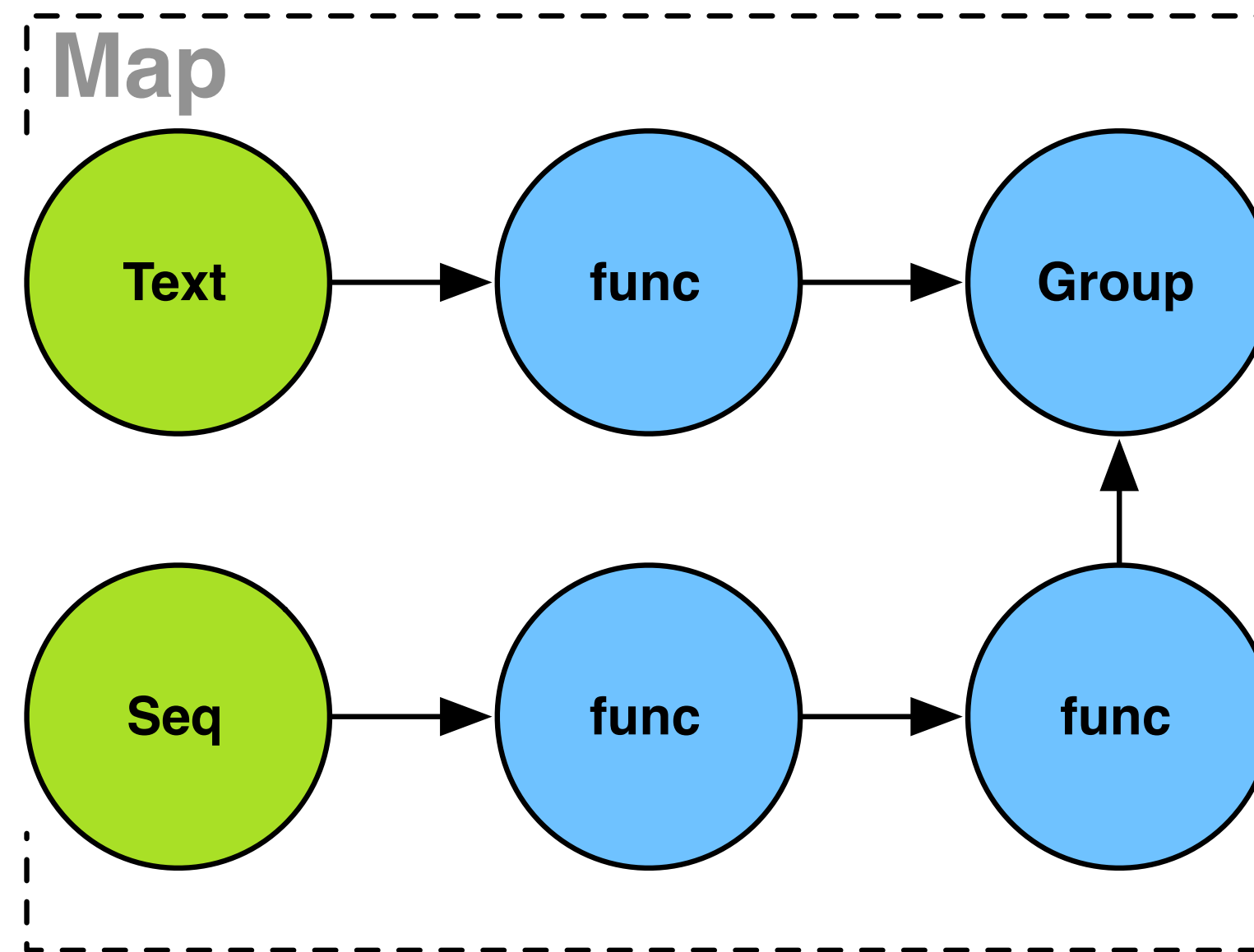
# All Together



# Joins and Merges

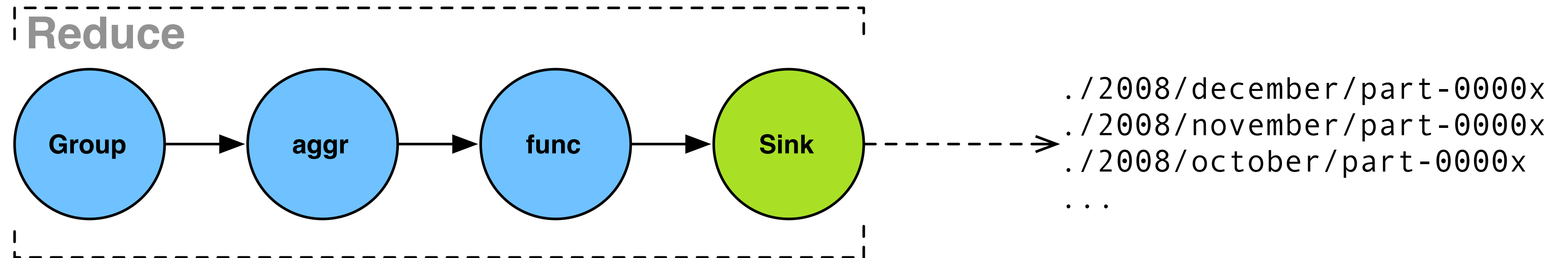
- Inner, Outer, Left, Right, and Mixed (N streams) Joins
- Merge of N number of heterogeneous streams into one normalized stream
- Self joins supported

# Heterogeneous Inputs



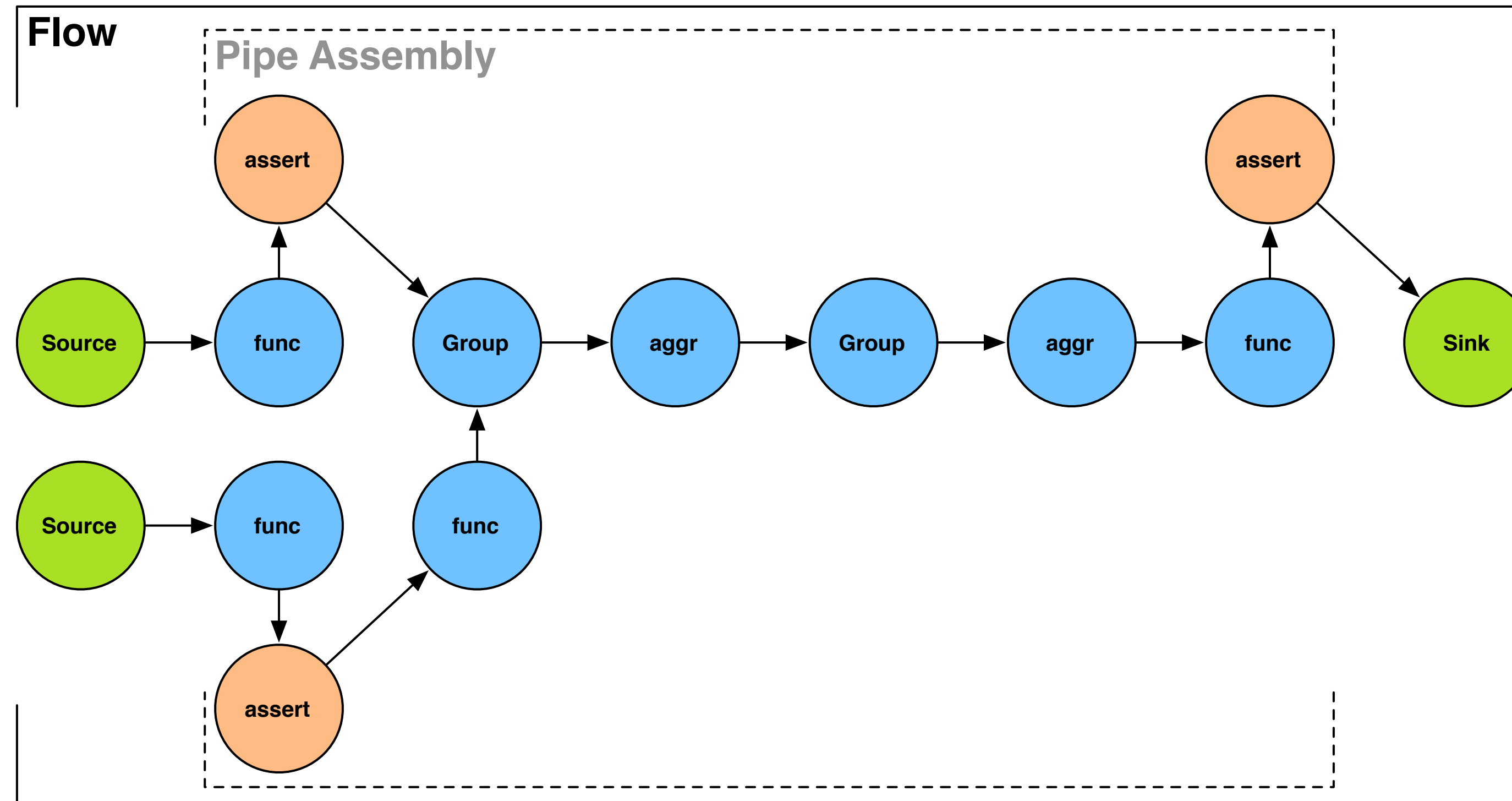
- A single mapper can read data from multiple InputFormats

# Dynamic Outputs



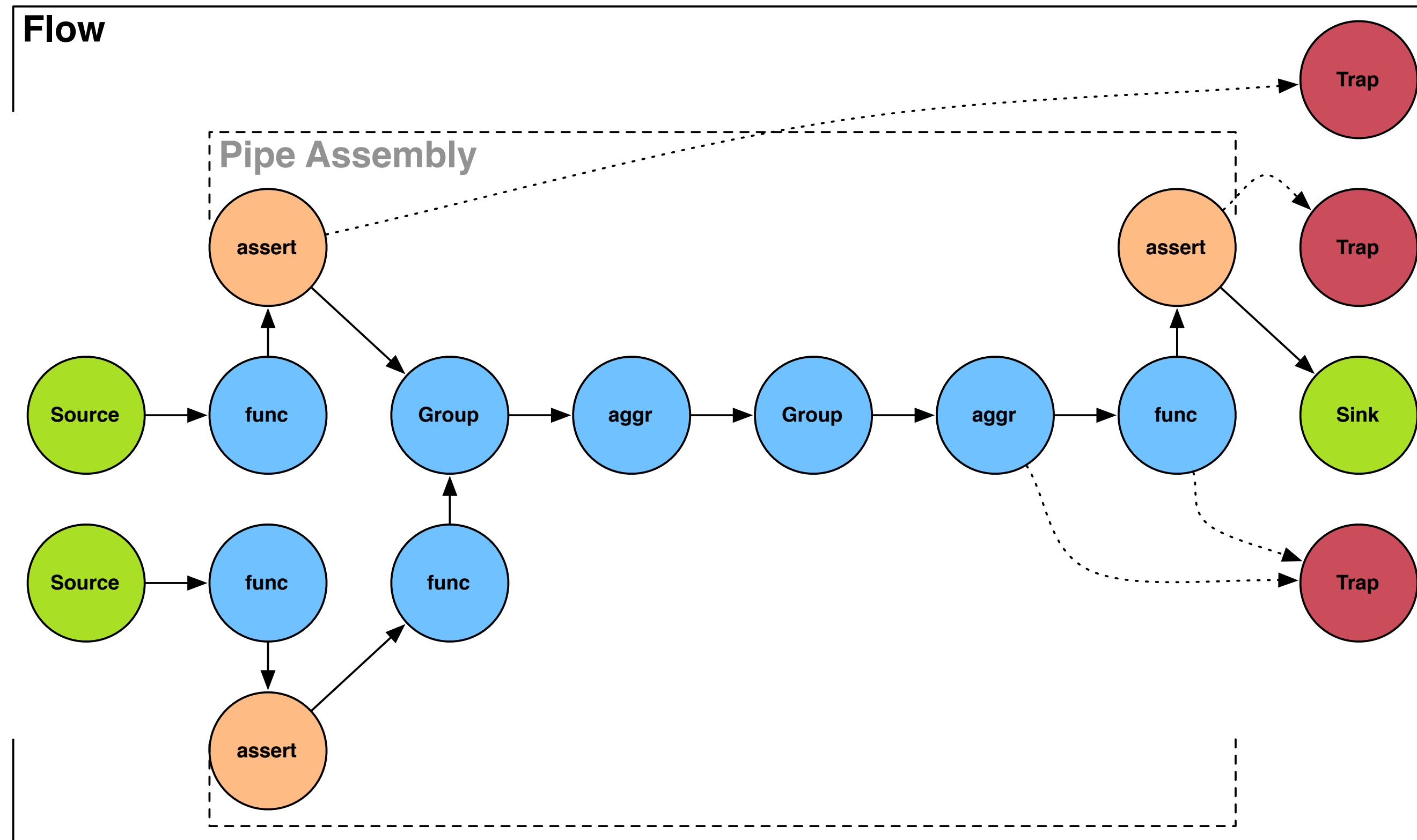
- Can sink to output folders named from Tuple values
- Even works Map side (must take care on num files)

# Stream Assertions



- Unit and Regression tests for Flows
- Planner can remove 'strict', 'validating', or all assertions

# Failure Traps



- Catch data causing Operations or Assertions to fail
- Allows processes to continue without data loss

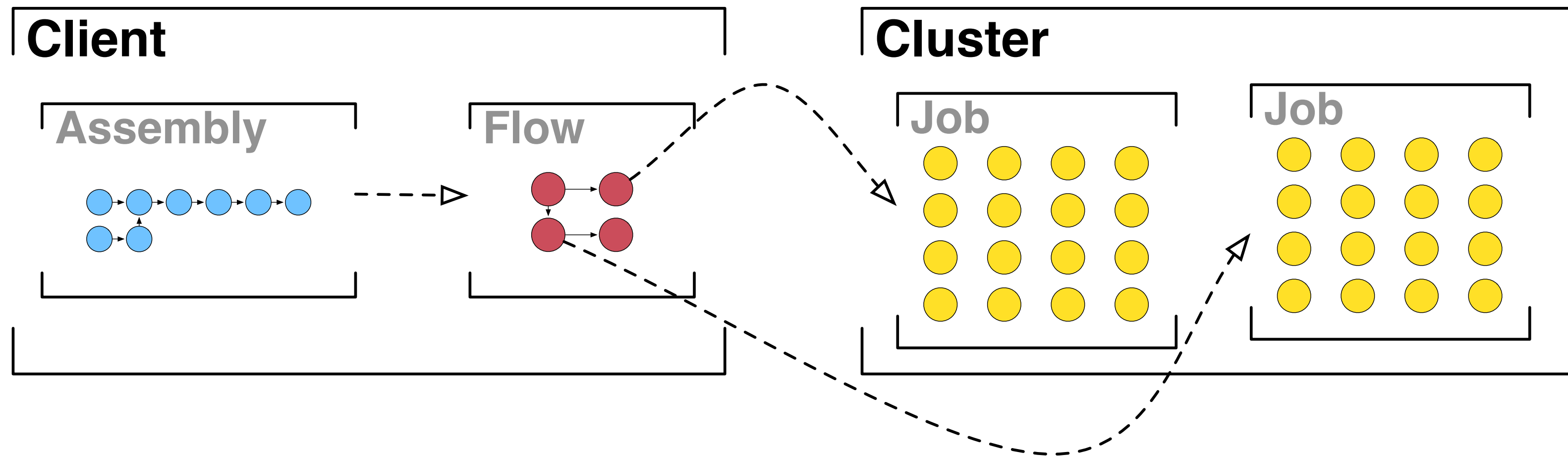
# Event Notification

- Listeners on Flows allow for notification for common events (start, complete, failed, and stopped)
- Useful for notifying external applications a process has completed

# Custom/Legacy MapReduce

- Anything that can be stuffed into a JobConf can be managed by a Cascade
- Useful for moving existing MR apps to Cascading
- Augmenting Cascading with a heavily optimized MR job
- Or overcoming any limitations of Cascading

# Scriptable



- Assembling jobs is not the same as executing jobs
- Operations run on the cluster
- Any language can be used to assemble and manage jobs

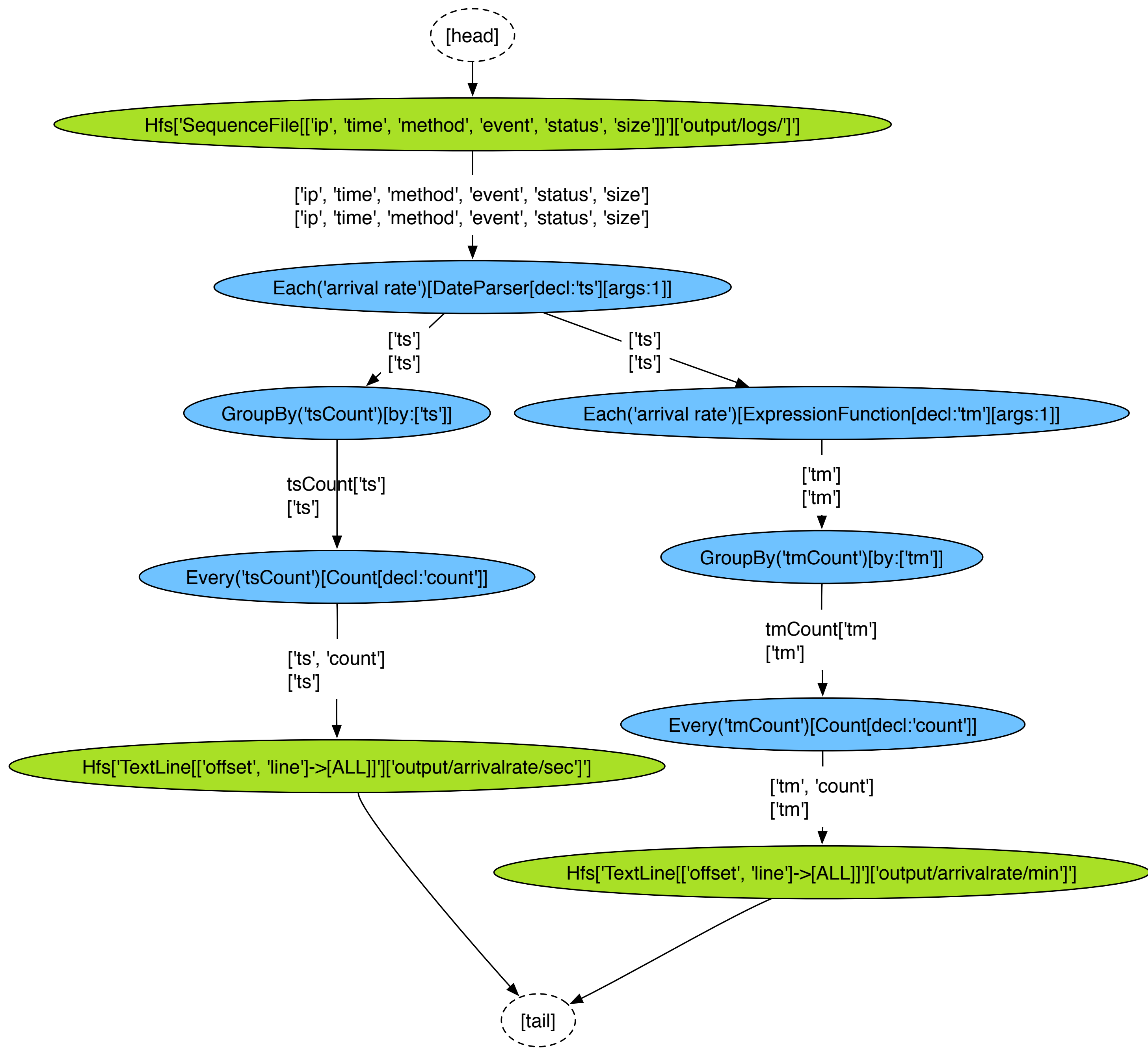
# Future

- IO vs CPU bound plans
- “Cost Modeling” for more efficient plans
  - Operation Generators vs. Filters
  - Operation Idempotency
  - Operation Response Time
- Multi-Core planning (decoupling from Hadoop)
- Pluggable “Other” target planning
- ...

# Recent News

- Cascading 75 / Pig 580
- Developer Support Contracts
- OEM/Commercial Licensing (mySql)

**Samples**



```
Operation parser = new DateParser( "dd/MMM/yyyy:HH:mm:ss Z" );
Pipe tsPipe = new Each( "arrival rate", new Fields( "time" ), parser );

Pipe tsCountPipe = new Pipe( "tsCount", tsPipe );
tsCountPipe = new GroupBy( tsCountPipe, new Fields( "ts" ) );
tsCountPipe = new Every( tsCountPipe, Fields.GROUP, new Count() );

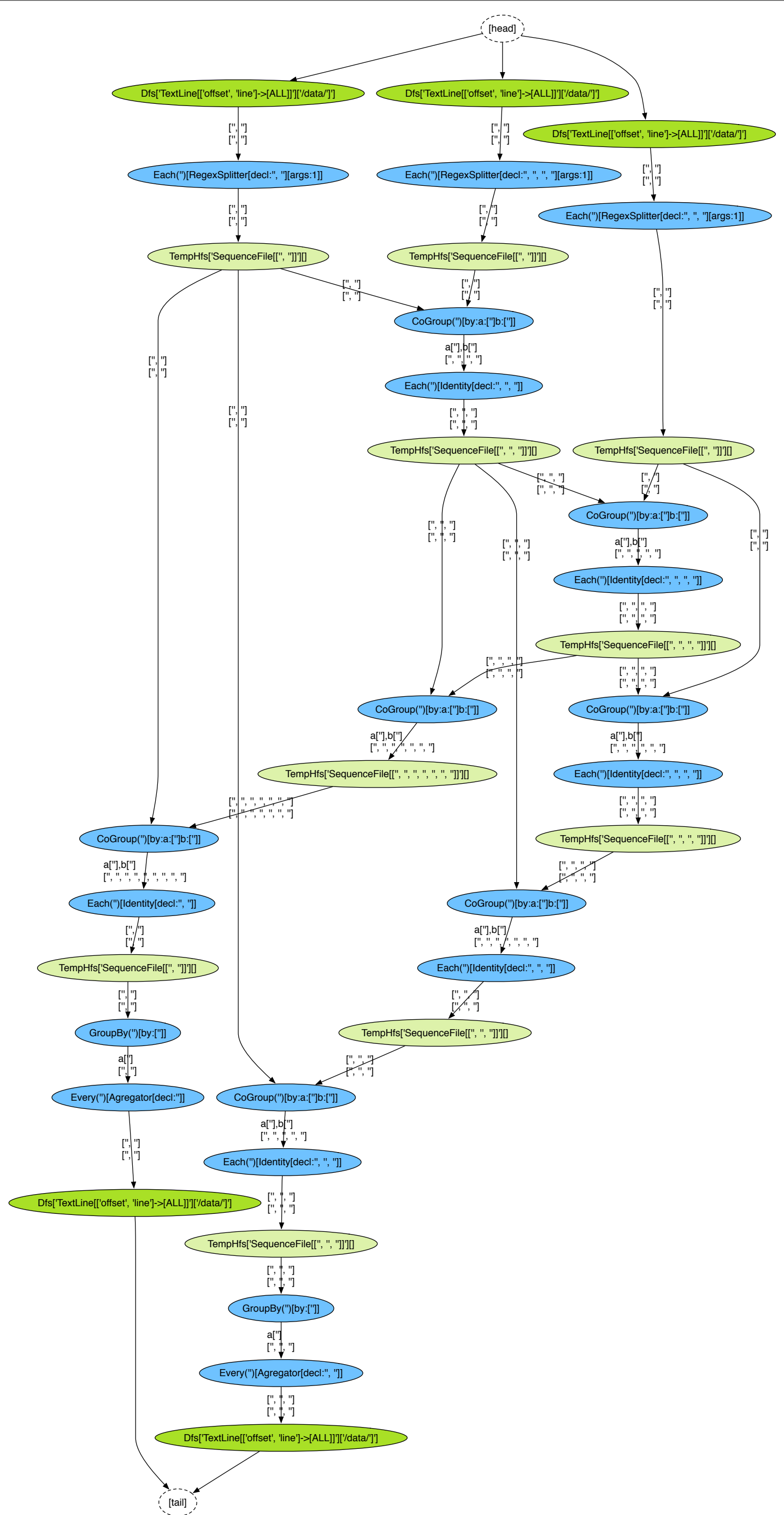
Operation expression =
    new ExpressionFunction( new Fields( "tm" ), "ts - (ts % (60 * 1000))", long.class );
Pipe tmPipe = new Each( tsPipe, expression );

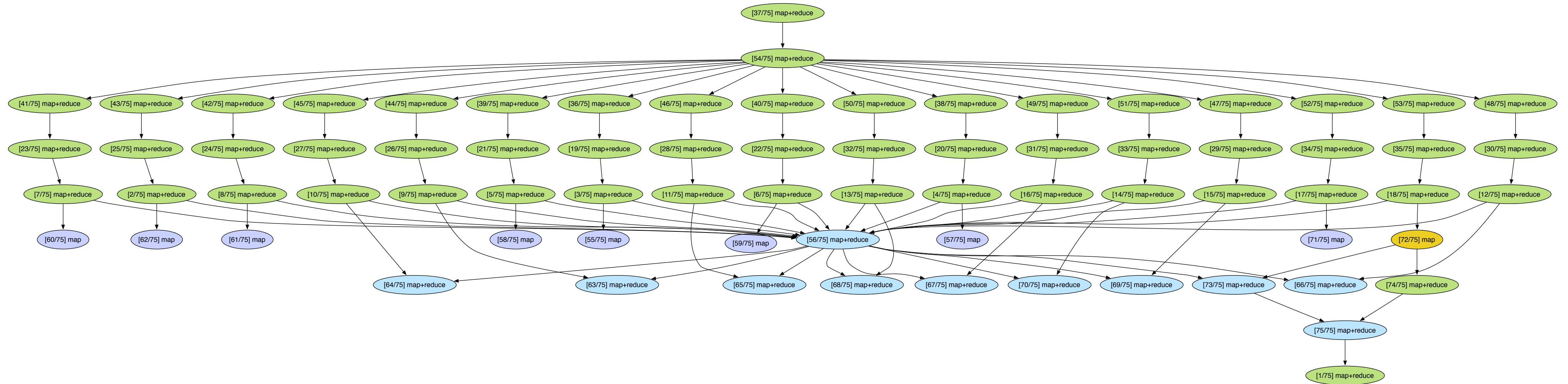
Pipe tmCountPipe = new Pipe( "tmCount", tmPipe );
tmCountPipe = new GroupBy( tmCountPipe, new Fields( "tm" ) );
tmCountPipe = new Every( tmCountPipe, Fields.GROUP, new Count() );

Tap tsSinkTap = new Hfs( new TextLine(), arrivalRateSecPath );
Tap tmSinkTap = new Hfs( new TextLine(), arrivalRateMinPath );

Pipe[] pipes = new Pipe[]{ tsCountPipe, tmCountPipe };
Tap[] taps = new Tap[]{ tsSinkTap, tmSinkTap };
Map<String, Tap> sinks = Cascades.tapsMap( pipes, taps );

Flow arrivalRateFlow =
    flowConnector.connect( parsedLogTap, sinks, tsCountPipe, tmCountPipe );
```





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

# Parting Thoughts

- Text is for Humans
- Data warehouses are caches
- MapReduce sucks (to think in)

# Productivity vs Efficiency

