Building Scale-Free Applications with Hadoop and Cascading

Chris K Wensel
Concurrent, Inc.

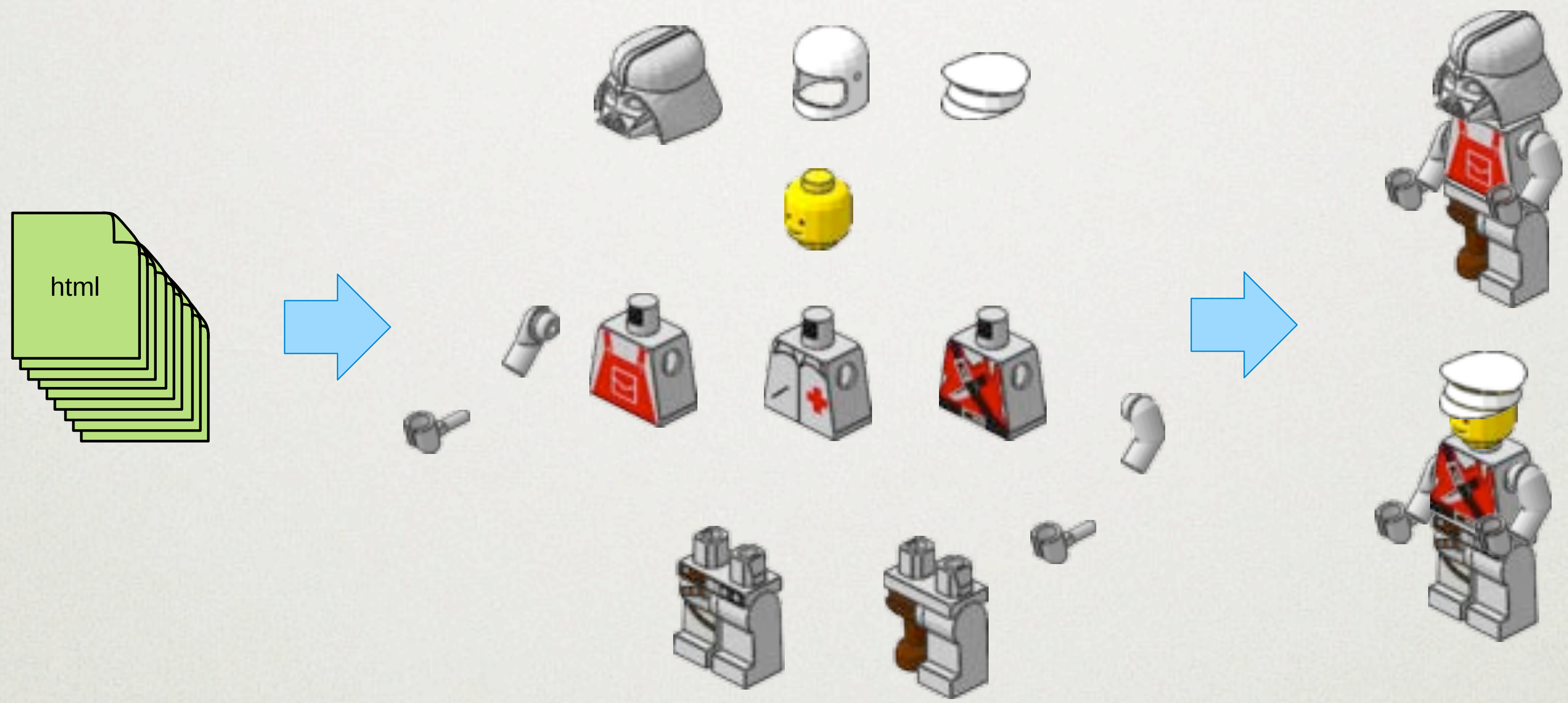
Introduction

Chris K Wensel

chris@wensel.net

- Cascading, *Lead developer*
  - http://cascading.org/
- Concurrent, Inc., *Founder*
  - Hadoop/Cascading support and tools
  - http://concurrentinc.com
- Scale Unlimited, *Principal Instructor*
  - Hadoop related Training and Consulting
  - http://scaleunlimited.com

My First Hadoop App



- Parse 10G of web content (400k pages)
- Extracted names, gender, & profession

When Done

- Was more than one Hadoop MapReduce “job”
- Used HashMaps and JSON to link jobs
- AWS, 20 Nodes, ~6 hours (dirt slow)

What I Realized

- MapReduce is hard to "think in"
- Map->Reduce->Map... is brittle
- POJO's and Maps are inefficient

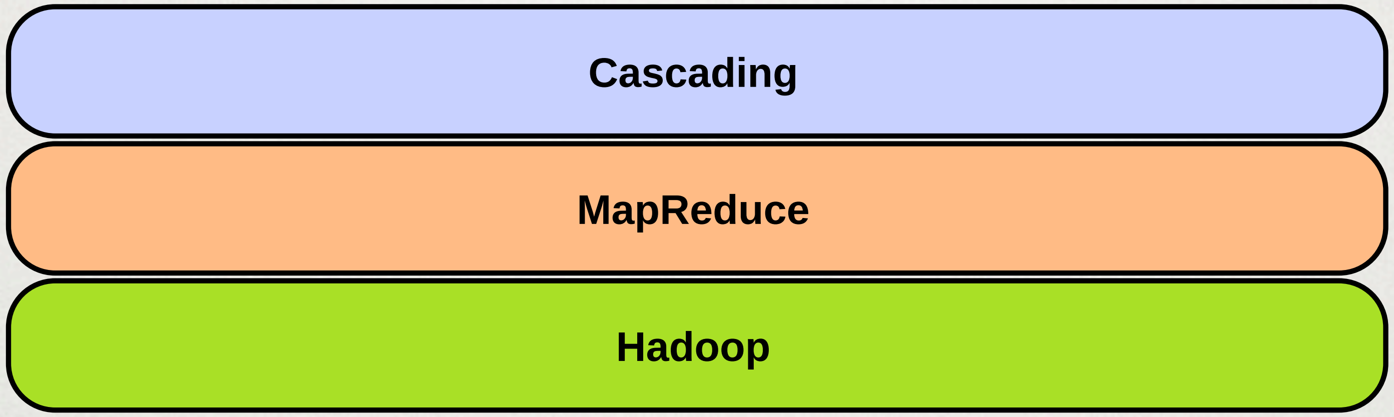
As Practitioners

- Syntax is for humans, APIs for software
- Integration is First Class

Thus Cascading

An alternative API to MapReduce

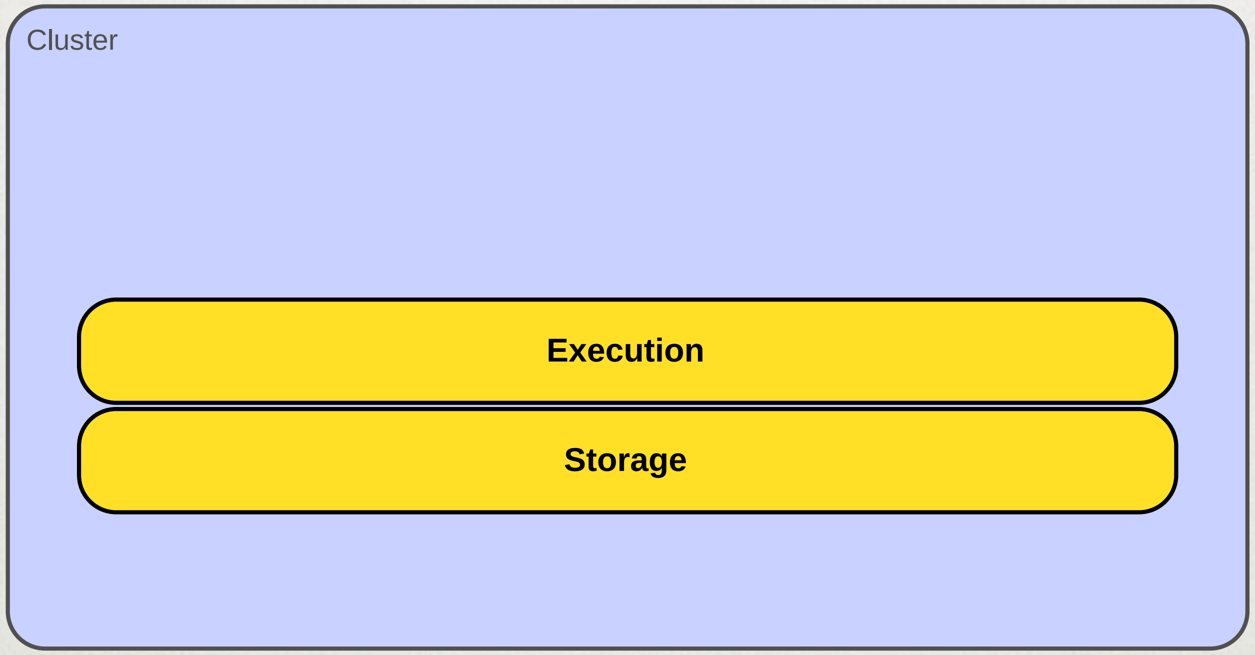
Topics



A rapid introduction to Hadoop, MapReduce patterns, and best practices with Cascading.

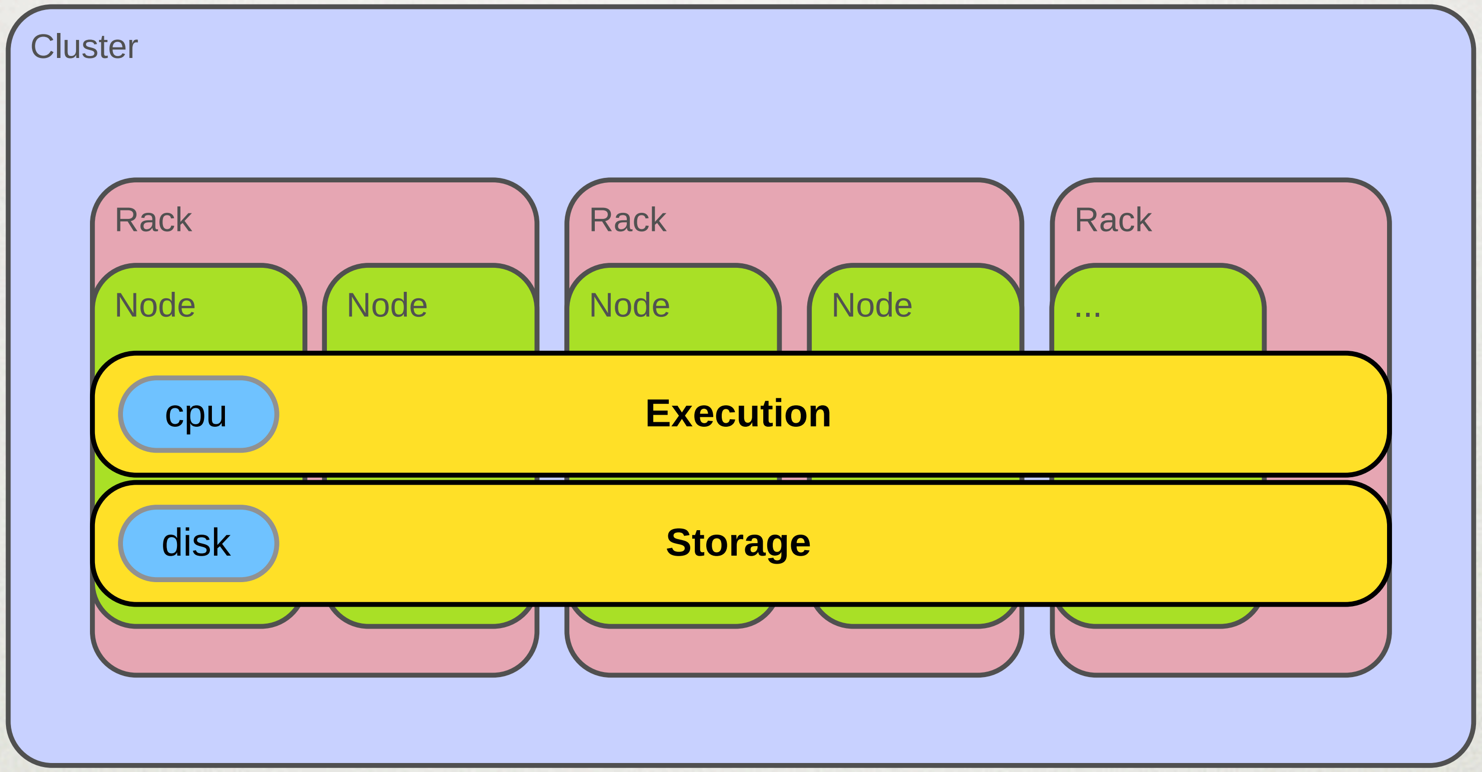
What is Hadoop?

Conceptually



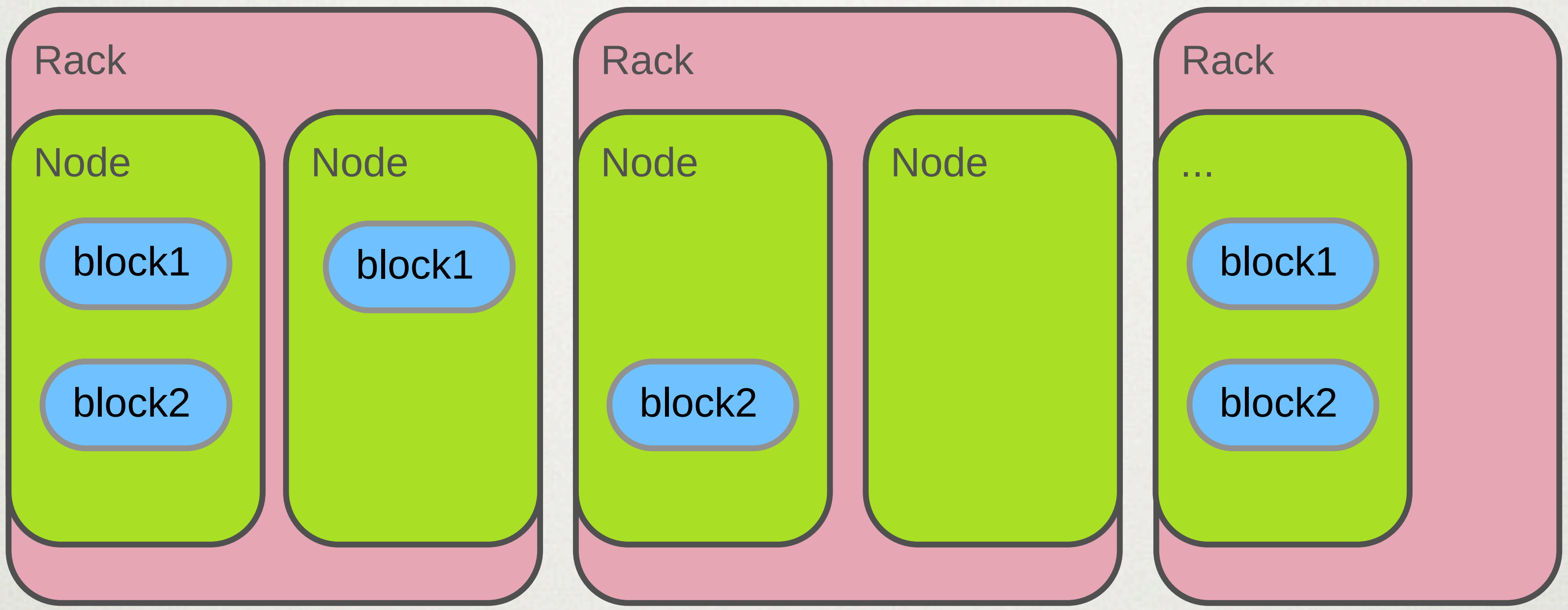
- Single FileSystem Namespace
- Single Execution-Space

CPU, Disk, Rack, and Node



- Virtualization of storage and execution resources
- Normalizes disks and CPU
- Rack and node aware
- Data locality and fault tolerance

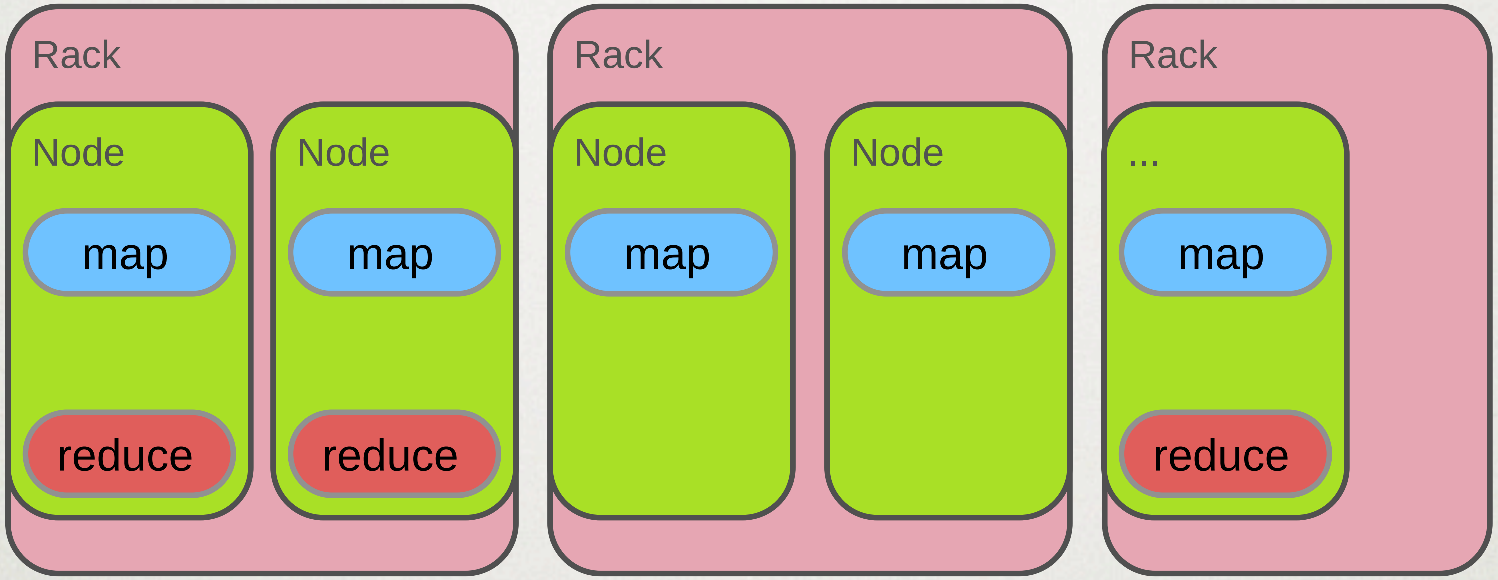
Storage



One file, two blocks large, 3 nodes replicated

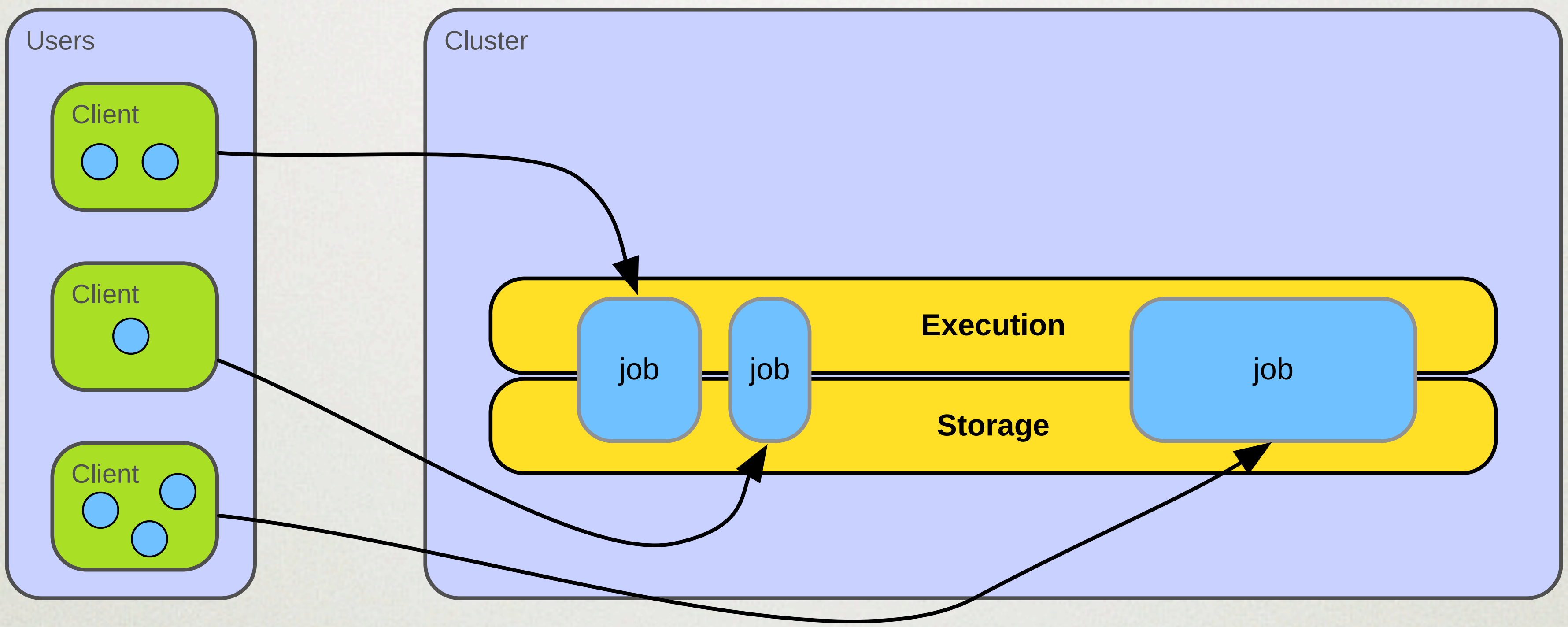
- Files chunked into blocks
- Blocks independently replicated

Execution



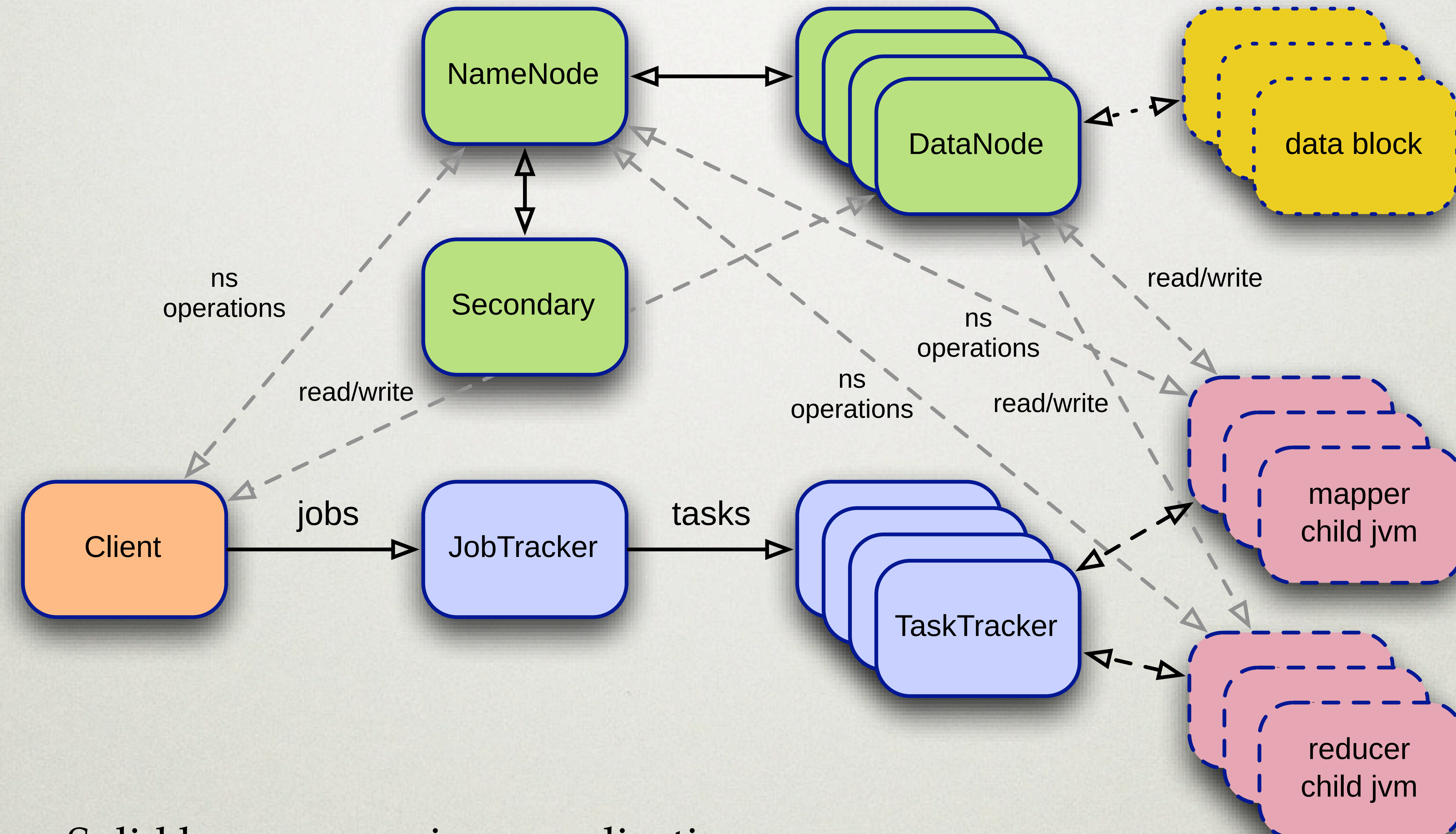
- Map and Reduce tasks move to data
- Or available processing slots

Clients and Jobs



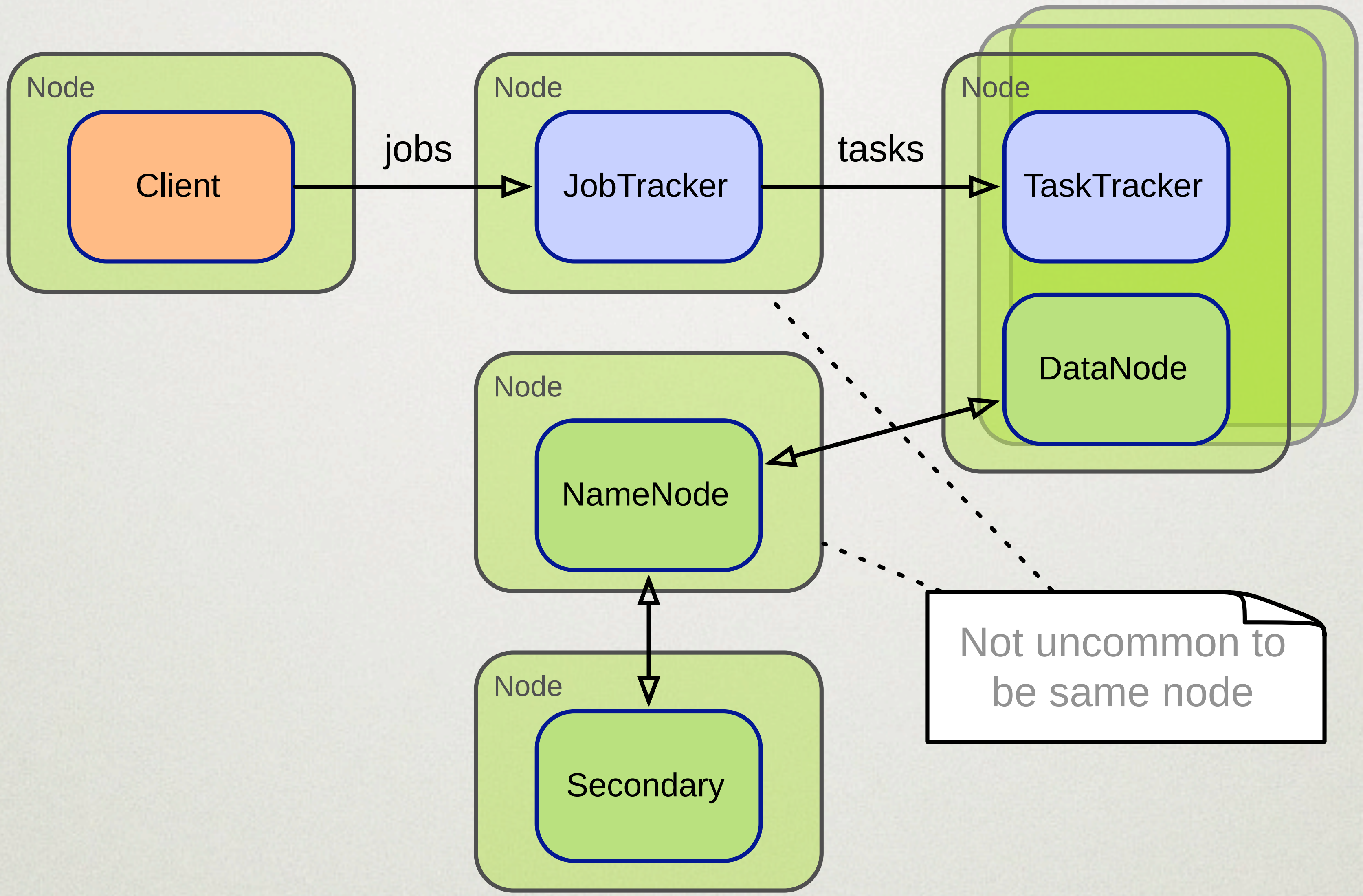
- Clients launch jobs into the cluster
- One at a time if there are dependencies
- Clients are not managed by Hadoop

Physical Architecture



- Solid boxes are unique applications
- Dashed boxes are child JVM instances on same node as parent
- Dotted boxes are blocks of managed files on same node as parent

Deployment



- TaskTracker and DataNode collocated

Why Hadoop?

Why It's Adopted

Big Data

PaaS

Big Data
Hard Problem

Platform as a Service
Wide Virtualization

How It's Used

Ad hoc
Queries

Querying / Sampling

Processing
Integration

Integration / Processing

Cascading and ...?

	Big Data	PaaS
Ad hoc Queries	Pig/Hive	?
Processing Integration	Cascading	Cascading

- Designed for Processing / Integration
- OK for ad-hoc queries (API, not Syntax)
- Should you use Hadoop for small queries?

ShareThis.com

Processing Integration

Big Data

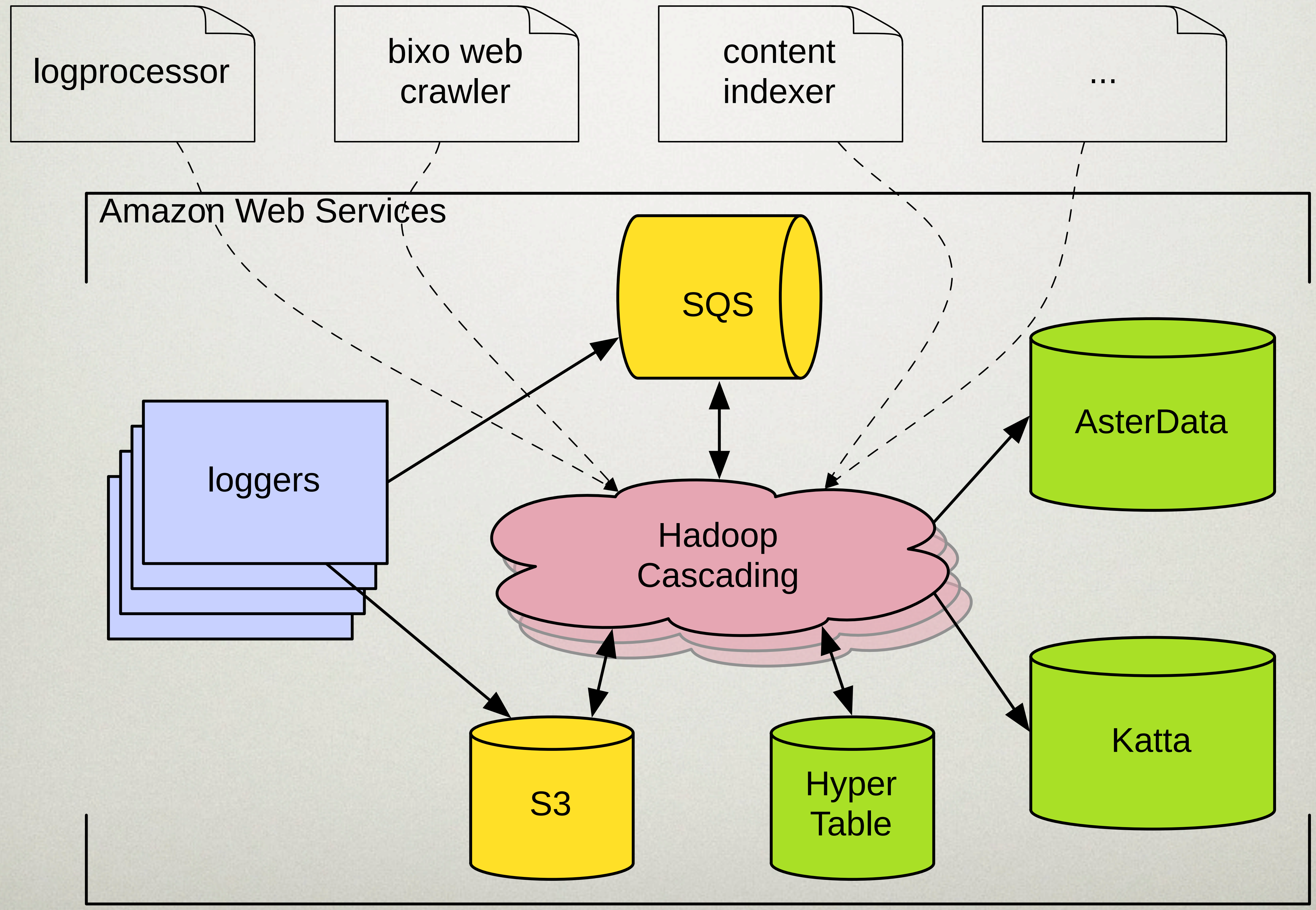
- Primarily integration and processing
- Running on AWS

Ad hoc Queries

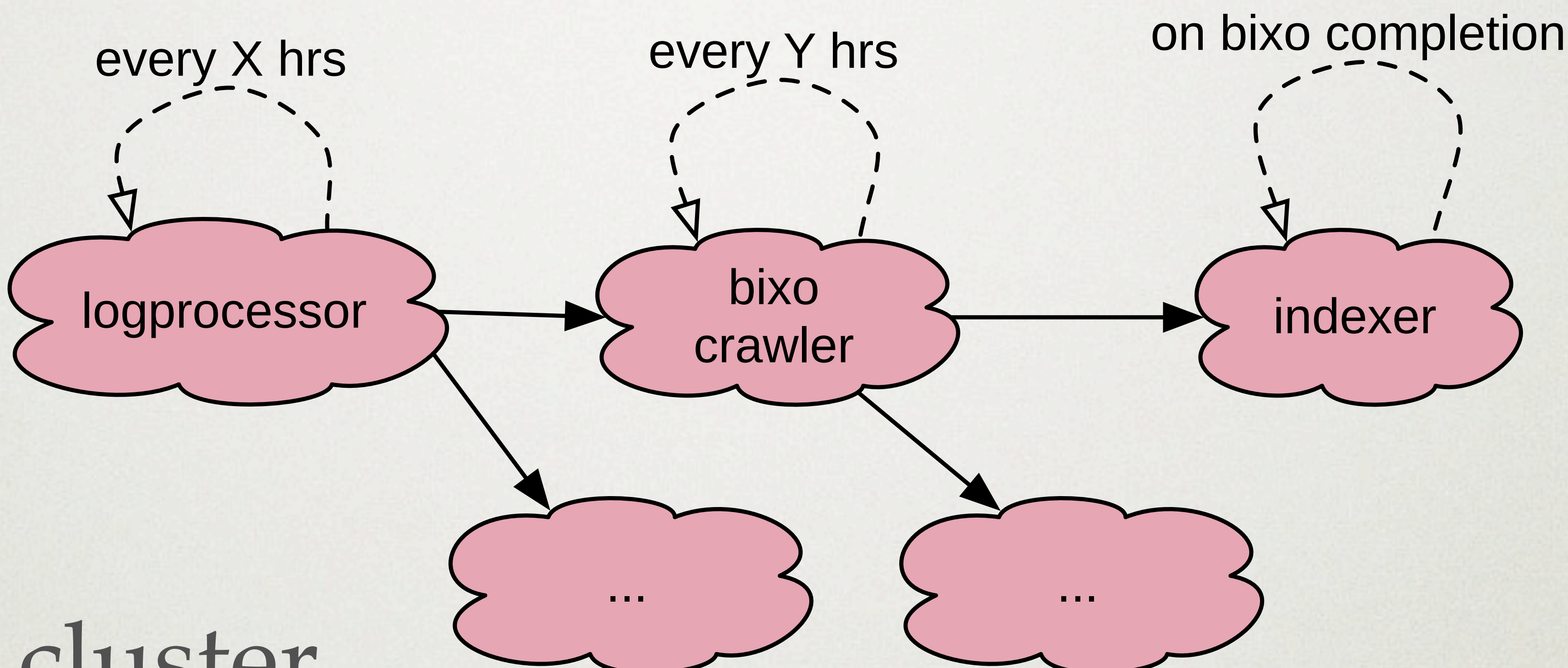
PaaS

- Ad-hoc queries (mining) performed on Aster Data nCluster, not Hadoop

ShareThis in AWS



Staged Pipeline



- Each cluster
  - is "on demand"
  - is "right sized" to load
  - is tuned to boot at optimum frequency

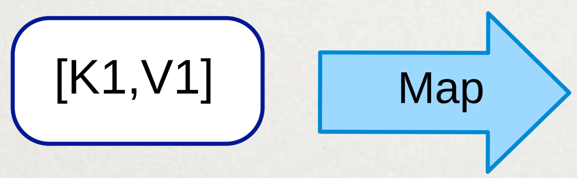
ShareThis: Best Practices

- Authoritative/Original data kept in S3 in its native format
- Derived data pipelined into relevant systems, when possible
- Clusters tuned to load for best utilization
- Loose coupling keeps change cases isolated
- GC'ing Hadoop not a bad idea

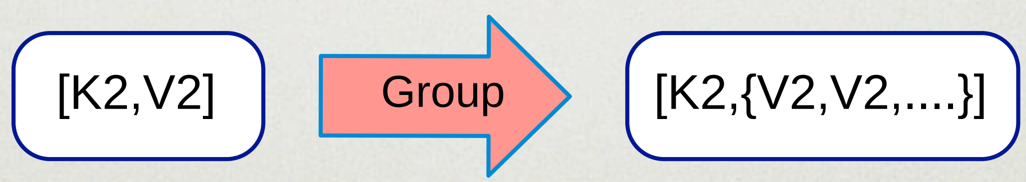
Thinking In MapReduce

It's really Map-Group-Reduce

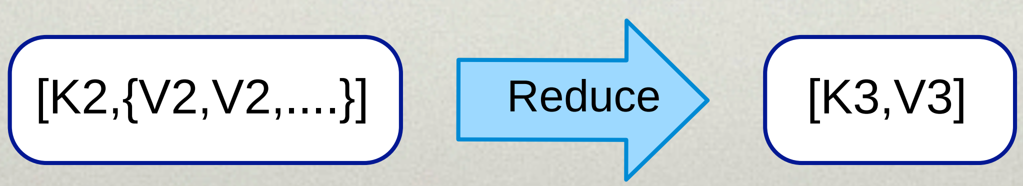
- *Map* and *Reduce* are user defined functions
- *Map* translates input Keys and Values to new Keys and Values



- System sorts the keys, and *groups* each unique Key with all its Values

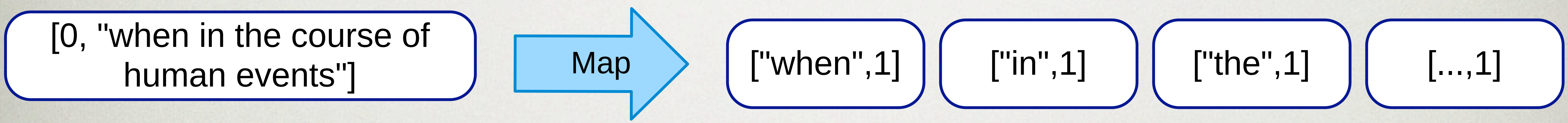


- *Reduce* translates the Values of each unique Key to new Keys and Values

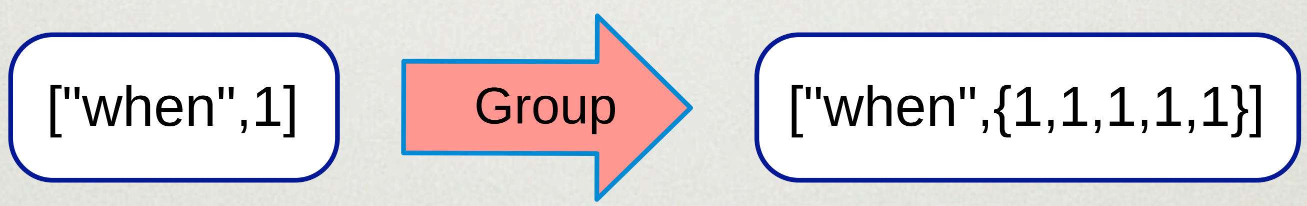


Word Count

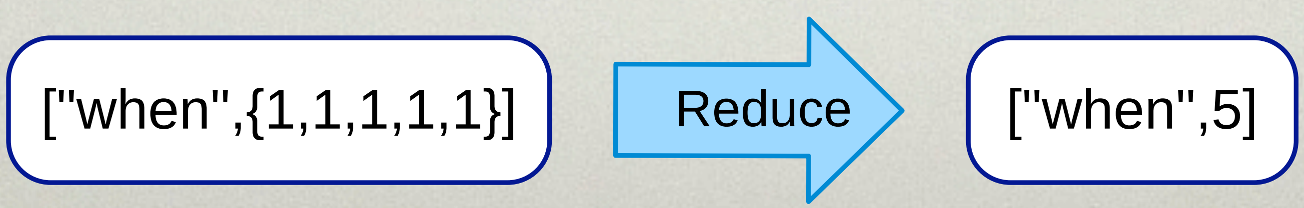
- Read a line of text, output every word



- Group all the values with each unique word



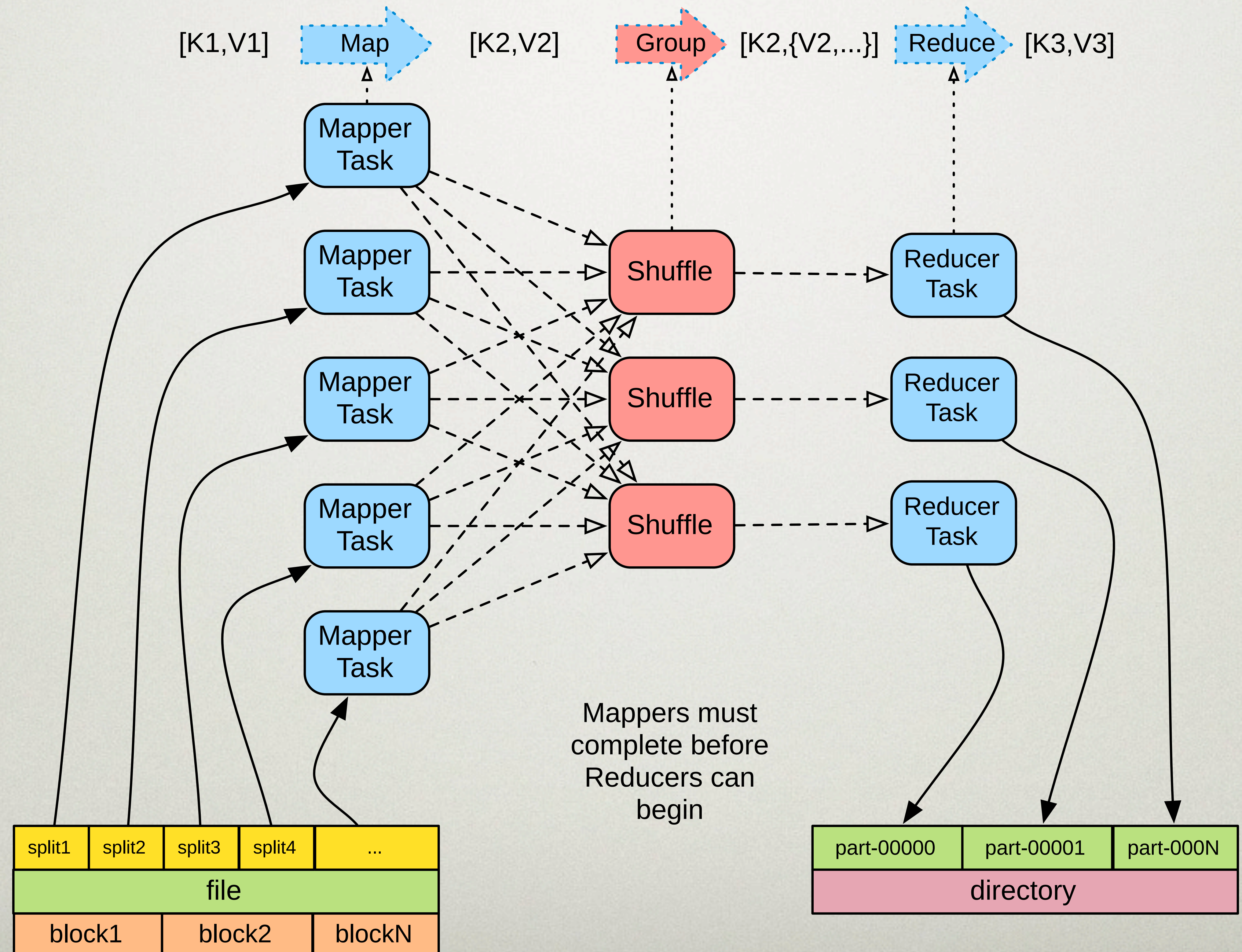
- Add up the values associated with each unique word



Know That...

- 1 Job == 1 Map impl + 1 Reduce impl
- Map:
  - is required in a Hadoop Job
  - may emit 0 or more Key Value pairs [K2,V2]
- Reduce:
  - is optional in a Hadoop Job
  - sees Keys in sort order
  - but Keys will be randomly distributed across Reducers
  - the collection of Values per Key are unordered [K2,{V2..}]
  - may emit 0 or more Key Value pairs [K3,V3]

Runtime Distribution



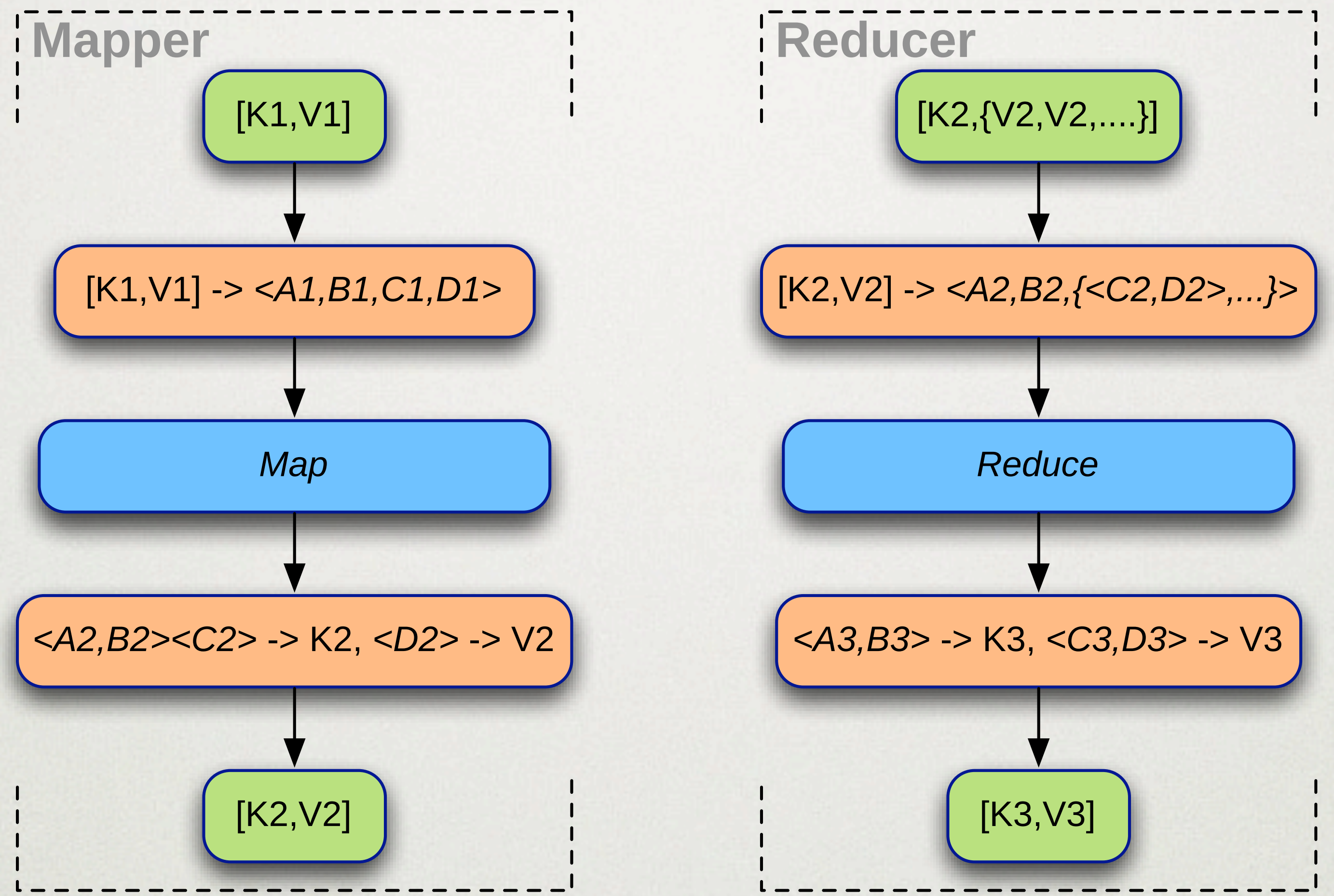
Common Patterns

- Filtering or "Grepping"
- Parsing, Conversion
- Counting, Summing
- Binning, Collating
- Distributed Tasks
- Simple Total Sorting
- Chained Jobs

Advanced Patterns

- Group By
- Distinct
- Secondary Sort
- CoGrouping / Joining
- Distributed Total Sort

For Example: Secondary Sort

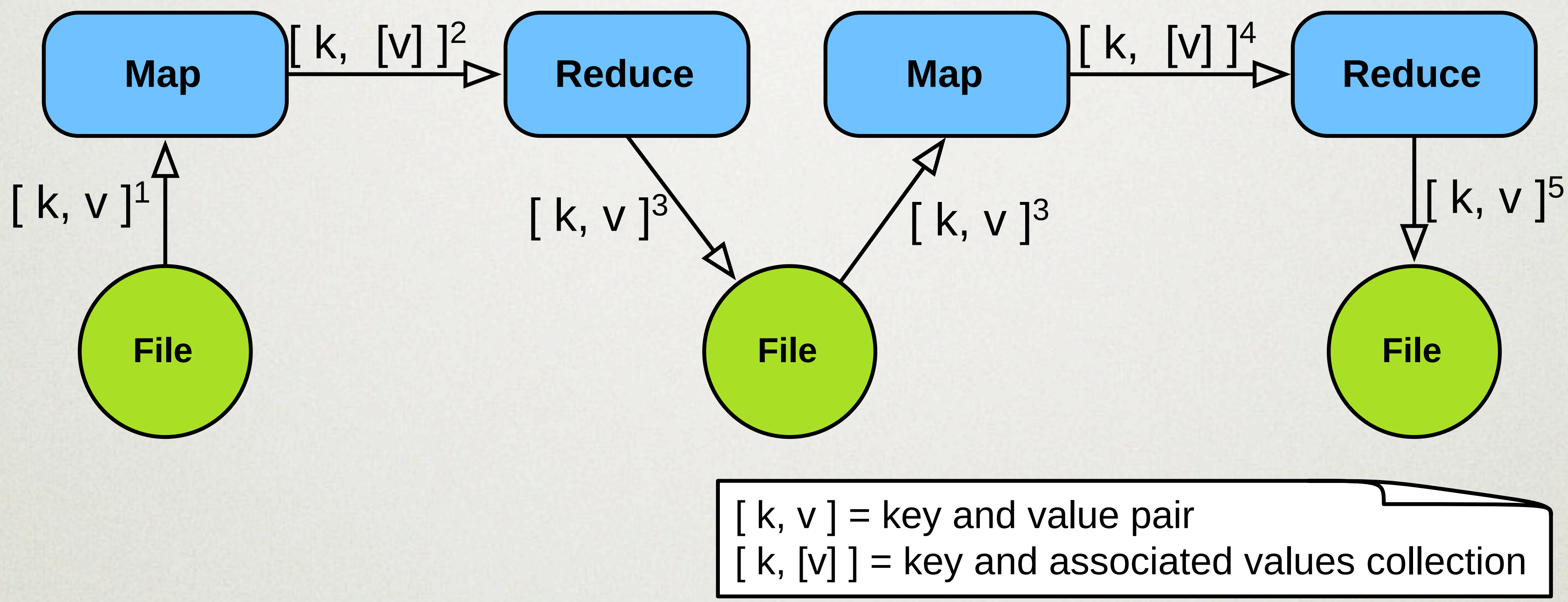


- The K2 key becomes a composite Key
  - Key: [grouping, secondary], Value: [remaining values]
- Shuffle phase
  - Custom Partitioner, must only partition on grouping Fields
  - Standard 'output key' Comparator, must compare on all Fields
  - Custom 'value grouping' Comparator, must compare grouping Fields

Very Advanced

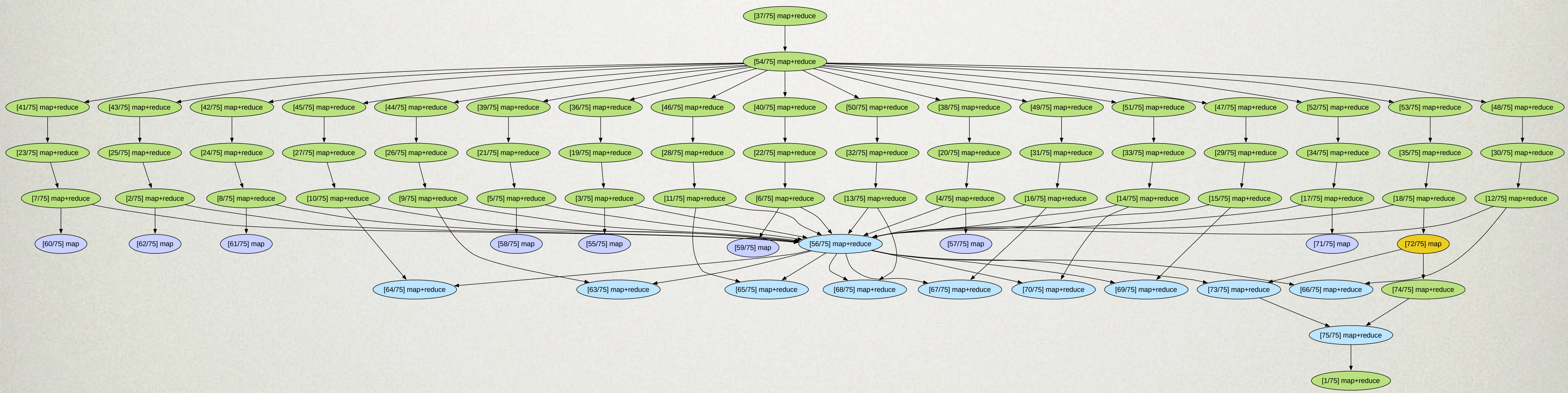
- Classification
- Clustering
- Regression
- Dimension Reduction
- Evolutionary Algorithms

Thinking in MapReduce



- It's not just about Keys and Values
- And not just about Map and Reduce

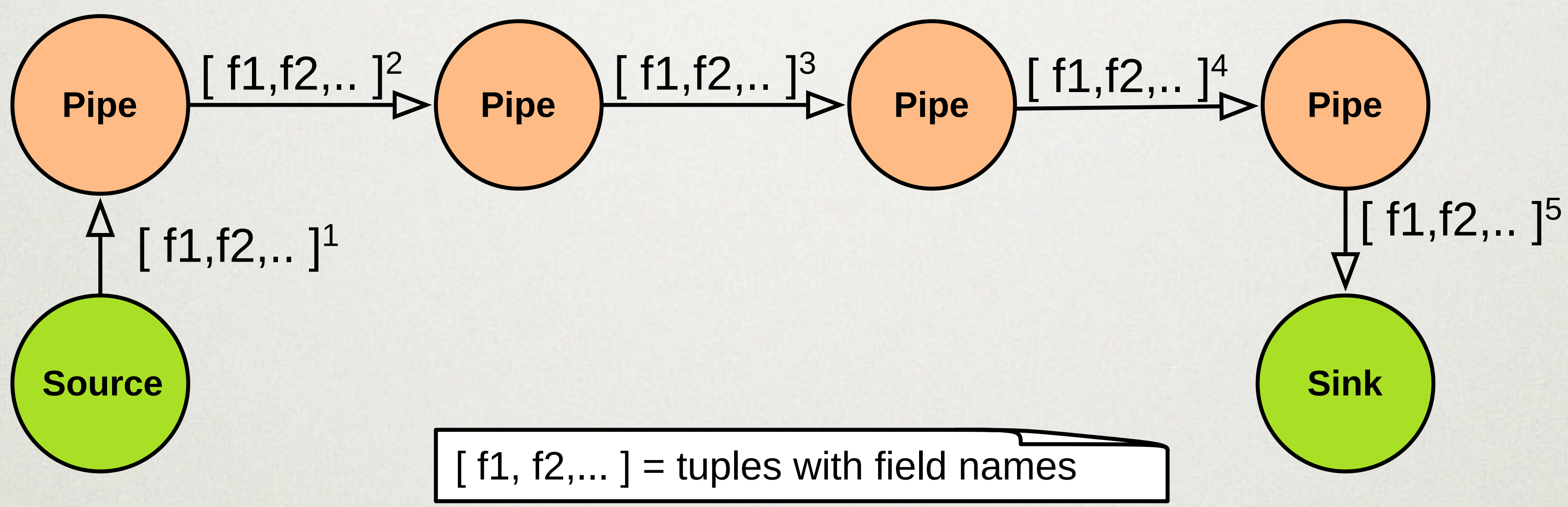
Real World Apps



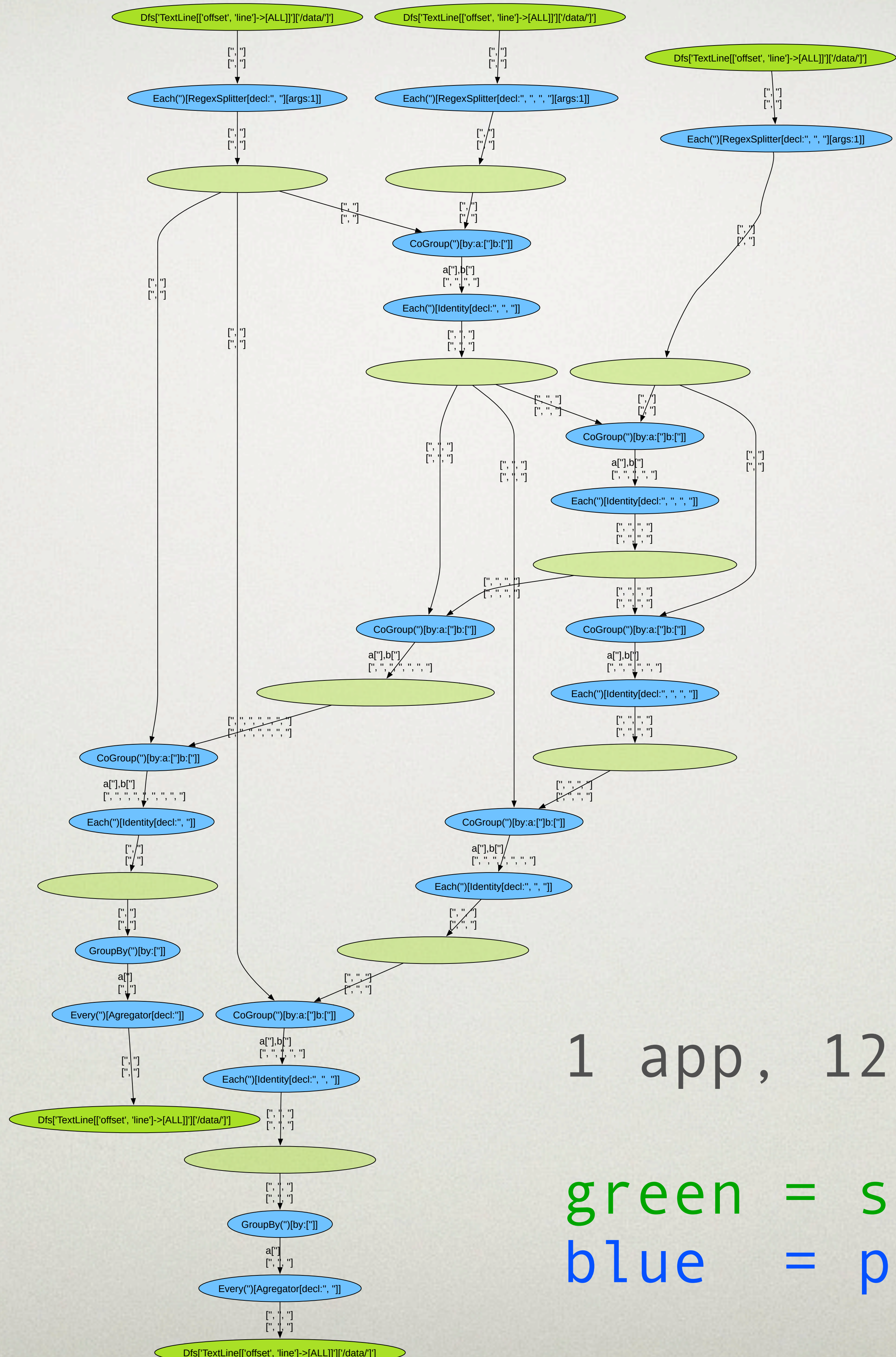
1 app, 75 jobs

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

Not Thinking in MapReduce



- Its easier with Fields and Tuples
- And Source/Sinks and Pipes



1 app, 12 jobs

green = source or sink

blue = pipe+operation

Cascading

Cascading

- A simpler alternative API to MapReduce
  - Fields and Tuples
  - Standard 'relational' operations
- Simplifies development of reusable
  - data operations,
  - logical process definitions
  - integration end-points, and
- Can be used by any JVM based language

Some Sample Code

http://github.com/cwensel/cascading.samples

- hadoop
- logparser
- loganalysis

bit.ly links on next pages

Raw MapReduce: Parsing a Log File

> read one key/value at a time from inputPath
> using some regular expression...
> parse value into regular expression groups
> save results as a TAB delimited row to the outputPath

http://bit.ly/RegexMap
http://bit.ly/RegexMain

http://bit.ly/RegexMap

```java
public class RegexParserMap extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text>
  {
  private Pattern pattern;
  private Matcher matcher;

  @Override
  public void configure( JobConf job )
    {
    pattern = Pattern.compile( job.get( "logparser.regex" ) );
    matcher = pattern.matcher( "" ); // lets re-use the matcher
    }

  @Override
  public void map( LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter )
    {
    matcher.reset( value.toString() );

    if( !matcher.find() )
      throw new RuntimeException( "could not match pattern: [" + pattern + "] with value: [" + value + "]" );

    StringBuffer buffer = new StringBuffer();

    for( int i = 0; i < matcher.groupCount(); i++ )
      {
      if( i != 0 )
        buffer.append( "\t" );

      buffer.append( matcher.group( i + 1 ) ); // skip group 0
      }

    // pass null so a TAB is not prepended, not all OutputFormats accept null
    output.collect( null, new Text( buffer.toString() ) );
    }
  }
```

http://bit.ly/RegexMain

```java
{
  public static void main( String[] args ) throws IOException
    {
    // create Hadoop path instances
    Path inputPath = new Path( args[ 0 ] );
    Path outputPath = new Path( args[ 1 ] );

    // get the FileSystem instances for the input path
    FileSystem outputFS = outputPath.getFileSystem( new JobConf() );

    // if output path exists, delete recursively
    if( outputFS.exists( outputPath ) )
      outputFS.delete( outputPath, true );

    // initialize Hadoop job configuration
    JobConf jobConf = new JobConf();
    jobConf.setJobName( "logparser" );

    // set the current job jar
    jobConf.setJarByClass( Main.class );

    // set the input path and input format
    TextInputFormat.setInputPaths( jobConf, inputPath );
    jobConf.setInputFormat( TextInputFormat.class );

    // set the output path and output format
    TextOutputFormat.setOutputPath( jobConf, outputPath );
    jobConf.setOutputFormat( TextOutputFormat.class );
    jobConf.setOutputKeyClass( Text.class );
    jobConf.setOutputValueClass( Text.class );

    // must set to zero since we have no redcue function
    jobConf.setNumReduceTasks( 0 );

    // configure our parsing map classs
    jobConf.setMapperClass( RegexParserMap.class );
    String apacheRegex = "^([^ ]*) +[^ ]* +[^ ]* + \\[([^]]*)\\] +\\\"([^ ]*) ([^ ]*) [^ ]* \\\" ([^ ]*) ([^ ]*).*$" ;
    jobConf.set( "logparser.regex" , apacheRegex );

    // create Hadoop client, must pass in this JobConf for some reason
    JobClient jobClient = new JobClient( jobConf );

    // submit job
    RunningJob runningJob = jobClient.submitJob( jobConf );

    // block until job completes
    runningJob.waitForCompletion();
    }
}
```

Cascading: Parsing a Log File

> read one "line" at a time from "source"
> using some regular expression...
> parse "line" into "ip, time, method, event, status, size"
> save as a TAB delimited row to the "sink"

http://bit.ly/LPMain

http://bit.ly/LPMain

```java
public static void main( String[] args )
  {
  String inputPath = args[ 0 ];
  String outputPath = args[ 1 ];

  // define what the input file looks like, "offset" is bytes from beginning
  TextLine scheme = new TextLine( new Fields( "offset", "line" ) );

  // create SOURCE tap to read a resource from the local file system, if input is not an URL
  Tap logTap = inputPath.matches( "^[^:]+://.*" ) ? new Hfs( scheme, inputPath ) : new Lfs( scheme, inputPath );

  // create an assembly to parse an Apache log file and store on an HDFS cluster

  // declare the field names we will parse out of the log file
  Fields apacheFields = new Fields( "ip", "time", "method", "event", "status", "size" );

  // define the regular expression to parse the log file with
  String apacheRegex = "^([^ ]*) +[^ ]* +[^ ]* +\\[([^]]*)\\] +\\\"([^ ]*) ([^ ]*) [^ ]* \\\" ([^ ]*) ([^ ]*).*$" ;

  // declare the groups from the above regex we want to keep. each regex group will be given
  // a field name from 'apacheFields', above, respectively
  int[] allGroups = {1, 2, 3, 4, 5, 6};

  // create the parser
  RegexParser parser = new RegexParser( apacheFields, apacheRegex, allGroups );

  // create the import pipe element, with the name 'import', and with the input argument named "line"
  // replace the incoming tuple with the parser results
  // "line" -> parser -> "ts"
  Pipe importPipe = new Each( "import", new Fields( "line" ), parser, Fields.RESULTS );

  // create a SINK tap to write to the default filesystem
  // by default, TextLine writes all fields out
  Tap remoteLogTap = new Hfs( new TextLine(), outputPath, SinkMode.REPLACE );

  // set the current job jar
  Properties properties = new Properties();
  FlowConnector.setApplicationJarClass( properties, Main.class );

  // connect the assembly to the SOURCE and SINK taps
  Flow parsedLogFlow = new FlowConnector( properties ).connect( logTap, remoteLogTap, importPipe );

  // optionally print out the parsedLogFlow to a DOT file for import into a graphics package
  // parsedLogFlow.writeDOT( "logparser.dot" );

  // start execution of the flow (either locally or on the cluster
  parsedLogFlow.start();

  // block until the flow completes
  parsedLogFlow.complete();
  }
```

What Changes When...

- We want to sort by "status" field?
- We want to filter out some "urls"?
- We want to sum the "size"?
- Do all the above?

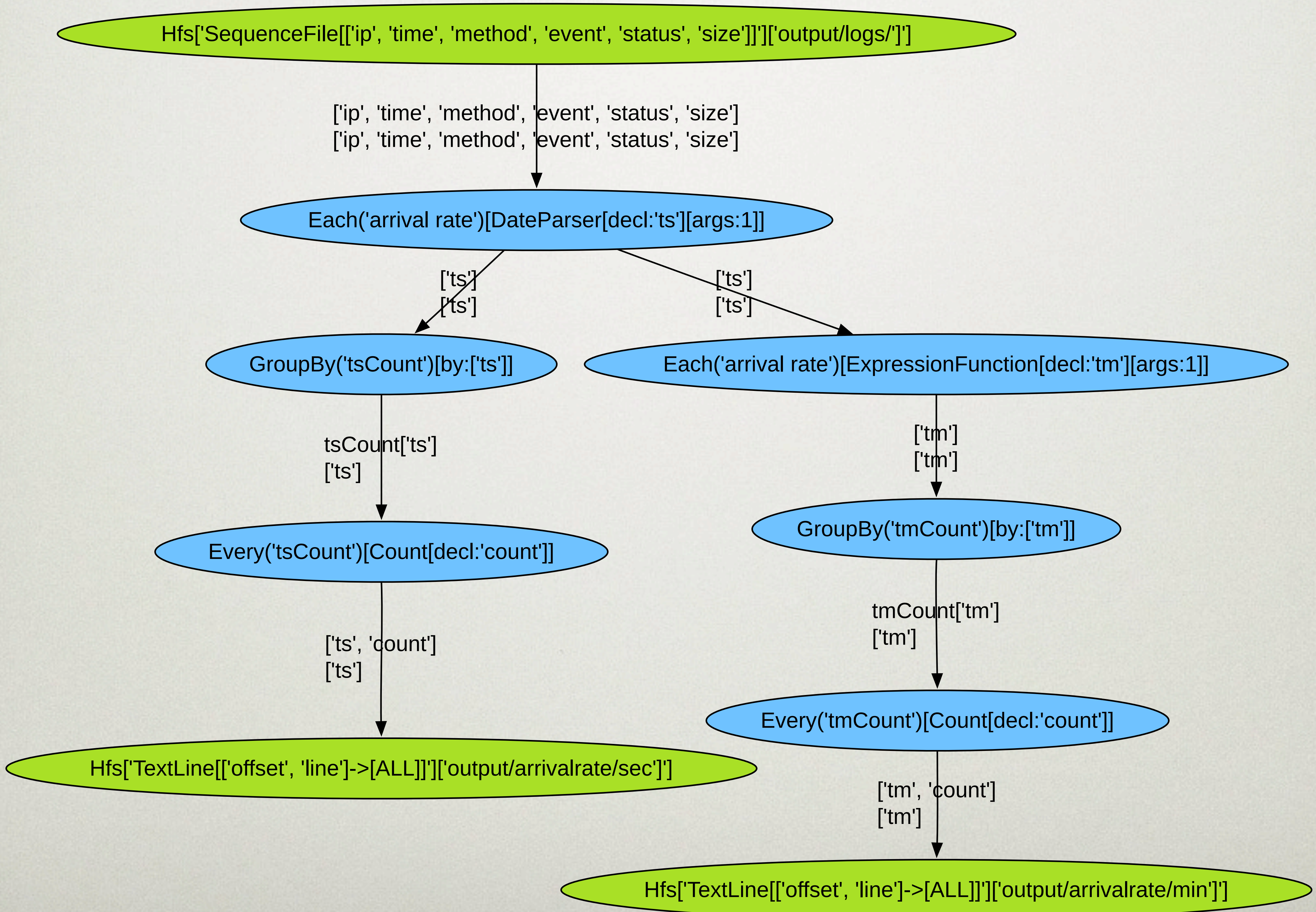
- We are thinking in 'fields' not Key/Values

Cascading: Arrival rate

> ..parse data as before..
> read parsed logs
> calculate “urls” per second
> calculate “urls” per minute
> save “sec interval” & “count” as a TAB’d row to “sink1”
> save “min interval” & “count” as a TAB’d row to “sink2”

http://bit.ly/LAMain

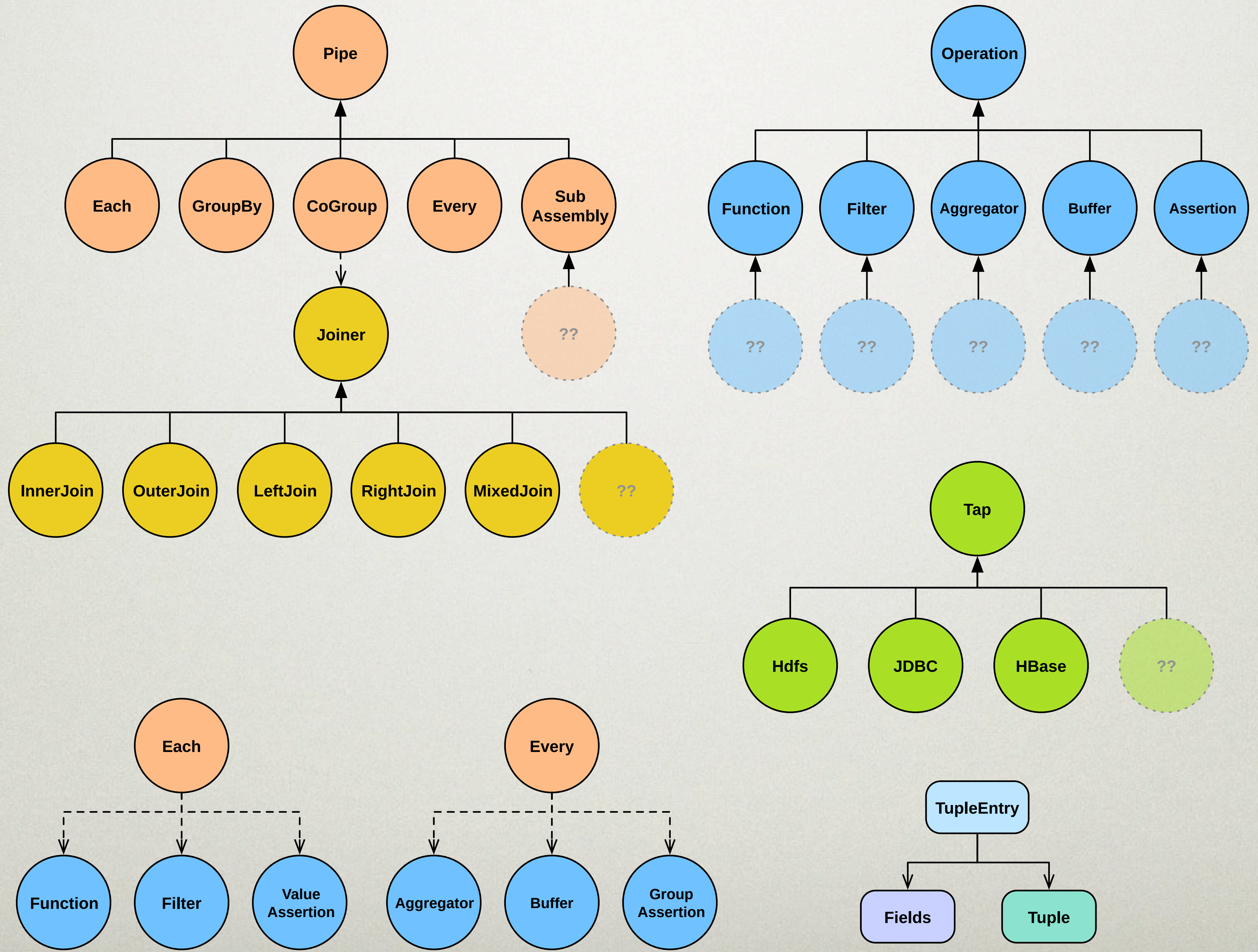
Arrival Rate Flow



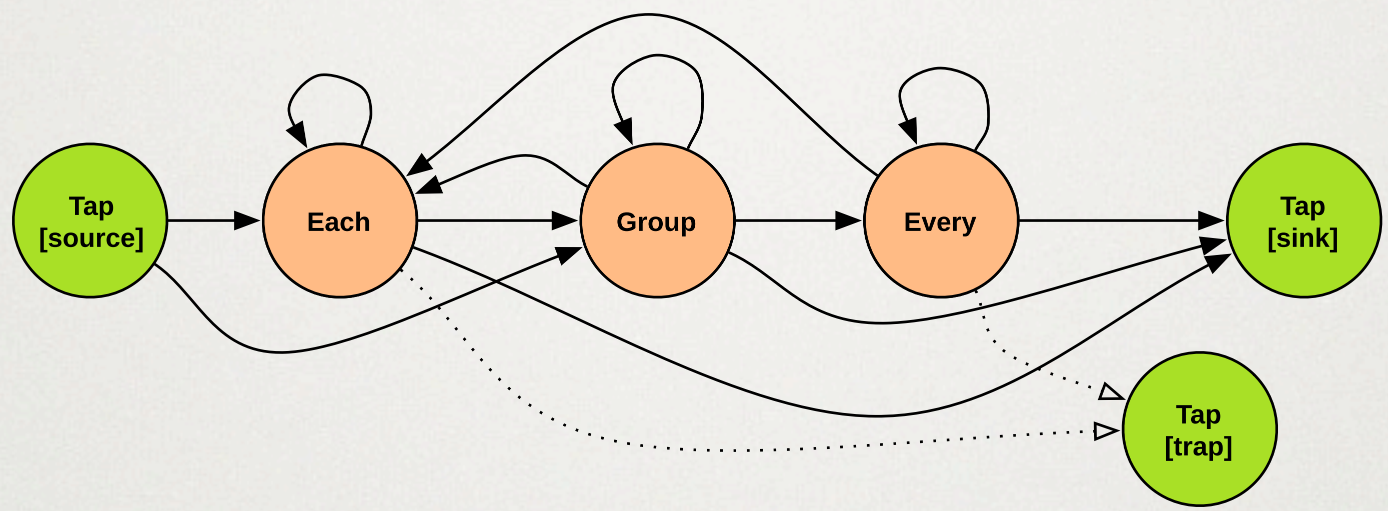
http://bit.ly/LAMain

```java
// set the current job jar
Properties properties = new Properties ();
FlowConnector .setApplicationJarClass ( properties , Main .class );

FlowConnector flowConnector = new FlowConnector ( properties );
CascadeConnector cascadeConnector = new CascadeConnector ();

String inputPath = args [ 0 ];
String logsPath = args [ 1 ] + "/logs/";
String arrivalRatePath = args [ 1 ] + "/arrivalrate/" ;
String arrivalRateSecPath = arrivalRatePath + "sec";
String arrivalRateMinPath = arrivalRatePath + "min";

// create an assembly to import an Apache log file and store on DFS
// declares: "time", "method", "event", "status", "size"
Fields apacheFields = new Fields ( "ip", "time", "method", "event", "status", "size" );
String apacheRegex = "^([^ ]*) +[^ ]* +[^ ]* +\\[([^]]*)\\] +\\\"([^ ]*) ([^ ]*) [^ ]*\\\" ([^ ]*) ([^ ]*).*$";
int[] apacheGroups = {1, 2, 3, 4, 5, 6};
RegexParser parser = new RegexParser ( apacheFields , apacheRegex , apacheGroups );
Pipe importPipe = new Each ( "import", new Fields ( "line" ), parser );

// create tap to read a resource from the local file system, if not an url for an external resource
// Lfs allows for relative paths
Tap logTap =
  inputPath .matches ( "^[^:]+://.*" ) ? new Hfs( new TextLine (), inputPath ) : new Lfs( new TextLine (), inputPath );
// create a tap to read/write from the default filesystem
Tap parsedLogTap = new Hfs( apacheFields , logsPath );

// connect the assembly to source and sink taps
Flow importLogFlow = flowConnector .connect ( logTap , parsedLogTap , importPipe );

// create an assembly to parse out the time field into a timestamp
// then count the number of requests per second and per minute

// apply a text parser to create a timestamp with 'second' granularity
// declares field "ts"
DateParser dateParser = new DateParser ( new Fields ( "ts" ), "dd/MMM/yyyy:HH:mm:ss Z");
Pipe tsPipe = new Each( "arrival rate" , new Fields ( "time" ), dateParser , Fields .RESULTS );

// name the per second assembly and split on tsPipe
Pipe tsCountPipe = new Pipe( "tsCount", tsPipe );
tsCountPipe = new GroupBy ( tsCountPipe , new Fields ( "ts" ) );
tsCountPipe = new Every ( tsCountPipe , Fields .GROUP , new Count () );

// apply expression to create a timestamp with 'minute' granularity
// declares field "tm"
Pipe tmPipe = new Each( tsPipe , new ExpressionFunction ( new Fields ( "tm" ), "ts - (ts % (60 * 1000))", long.class ) );

// name the per minute assembly and split on tmPipe
Pipe tmCountPipe = new Pipe( "tmCount", tmPipe );
tmCountPipe = new GroupBy ( tmCountPipe , new Fields ( "tm" ) );
tmCountPipe = new Every ( tmCountPipe , Fields .GROUP , new Count () );

// create taps to write the results the default filesystem, using the given fields
Tap tsSinkTap = new Hfs( new TextLine (), arrivalRateSecPath );
Tap tmSinkTap = new Hfs( new TextLine (), arrivalRateMinPath );

// a convenience method for binding taps and pipes, order is significant
Map<String , Tap> sinks = Cascades .tapsMap ( Pipe .pipes ( tsCountPipe , tmCountPipe ), Tap .taps ( tsSinkTap , tmSinkTap ) );

// connect the assembly to the source and sink taps
Flow arrivalRateFlow = flowConnector .connect ( parsedLogTap , sinks , tsCountPipe , tmCountPipe );

// optionally print out the arrivalRateFlow to a graph file for import into a graphics package
//arrivalRateFlow.writeDOT( "arrivalrate.dot" );

// connect the flows by their dependencies, order is not significant
Cascade cascade = cascadeConnector .connect ( importLogFlow , arrivalRateFlow );

// execute the cascade, which in turn executes each flow in dependency order
cascade .complete ();
```

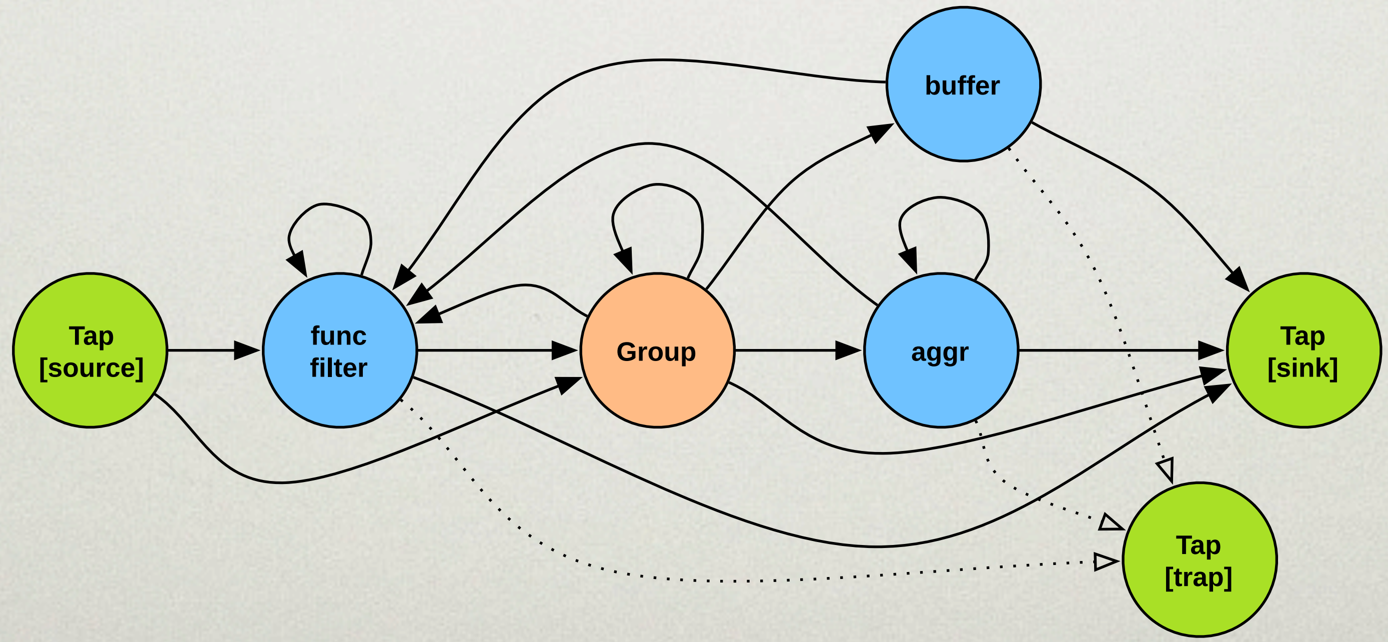
Cascading Model



Assembling Applications

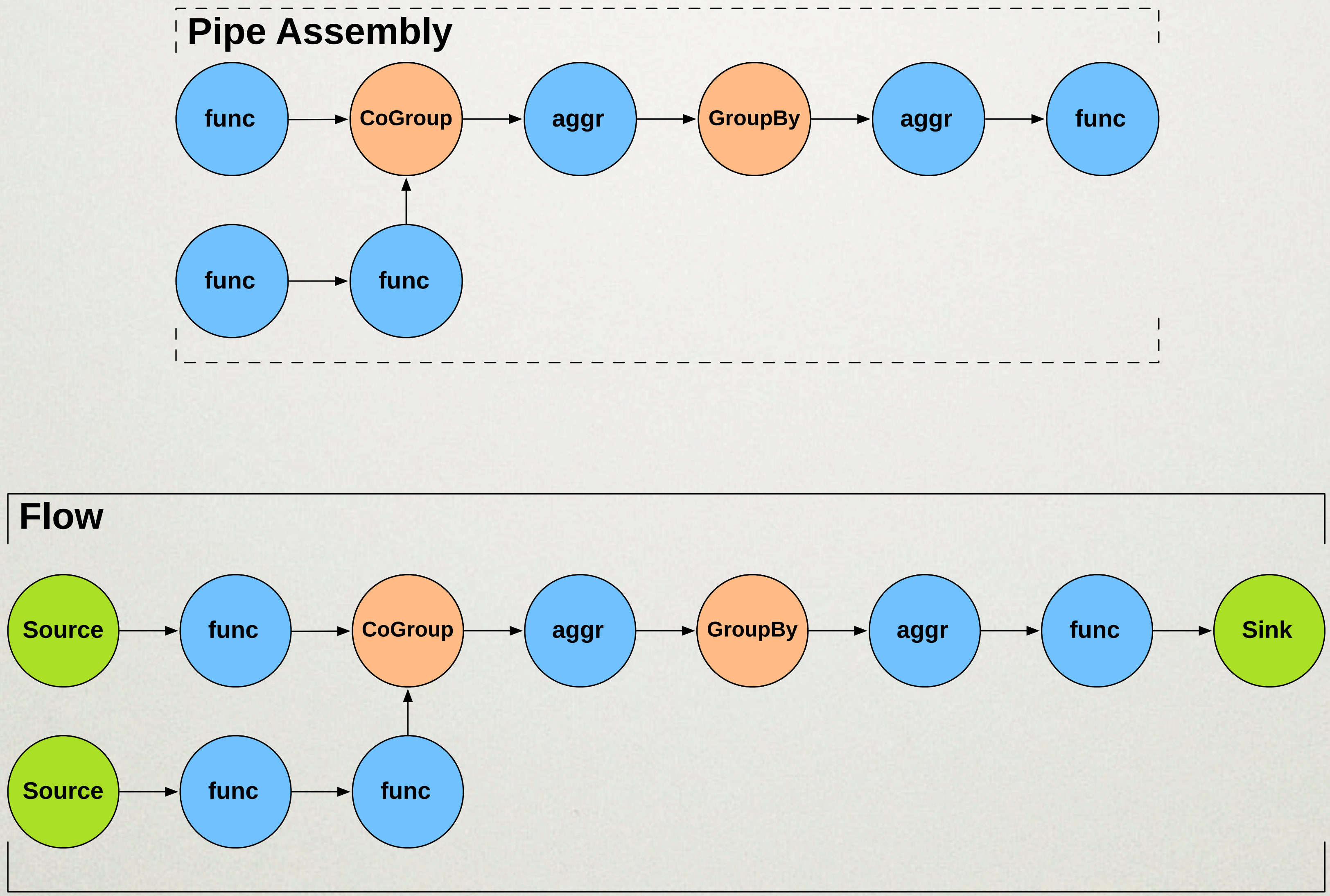


- Pipes can be arranged into arbitrary assemblies



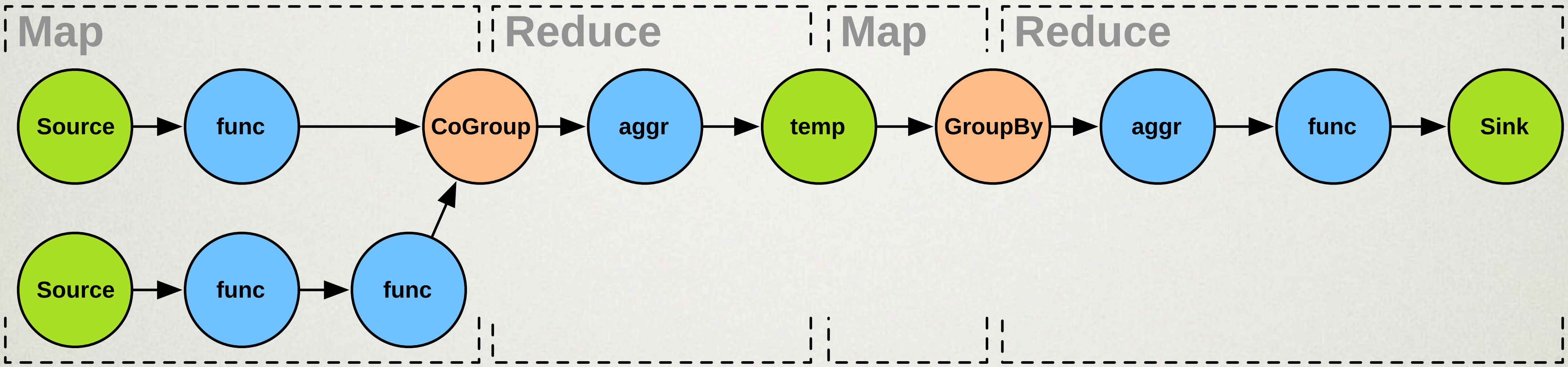
- Some limitations imposed by Operations

Assembly -> Flow



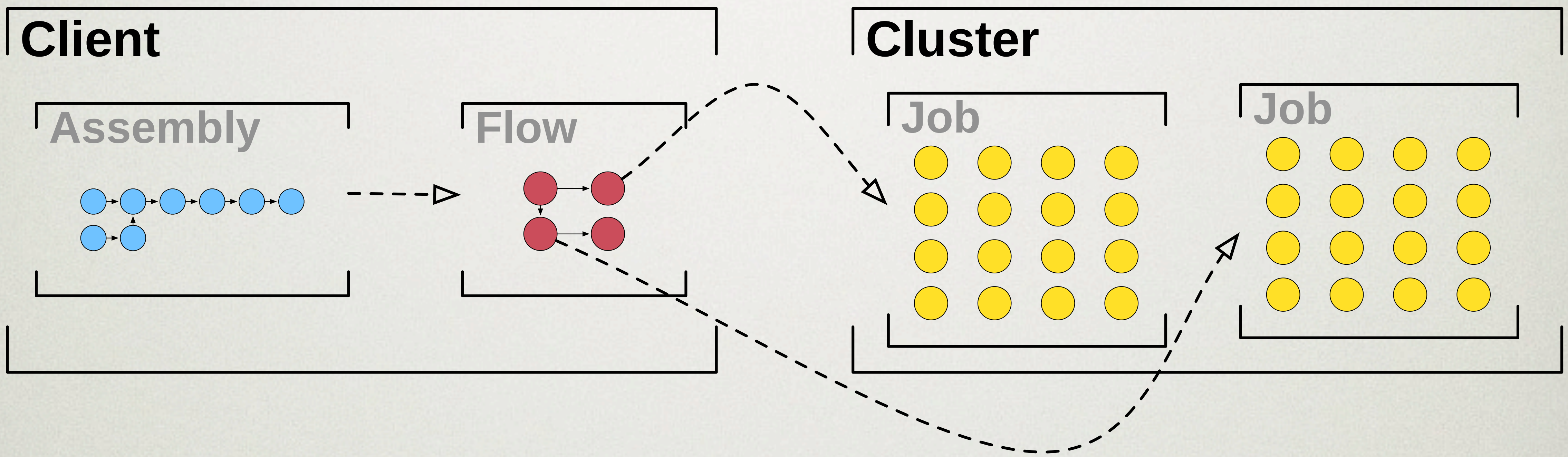
- Assemblies are planned into Flows

Flow Planner



- The Planner handles how operations are partitioned into MapReduce jobs

Assembly -> Cluster



Best Practices: Developing

- Organize Flows as med/small work-loads
  - Improves reusability and work-sharing
- Small/Medium-grained Operations
  - Improves reusability
- Coarse-grained Operations
  - Better performance vs flexibility
- SubAssemblies encapsulate responsibilities
  - Separate “rules” from “integration”
- Solve problem first, Optimize second
  - Can replace Flows with raw MapReduce

Best Practices: Testing

- Unit tests
  - Operations tested independent of Hadoop or Map/Reduce
- Regression / Integration Testing
  - Use built-in testing cluster for small data
  - Use EC2 for large regression data sets
- Inline "strict" Assertions into Assemblies
  - Planner can remove them at run-time
  - SubAssemblies tested independently of other logic

Best Practices: Data Processing

- Account for good/bad data
  - Split good/bad data by rules
  - Capture rule reasons with “bad” data
- Inline “validating” Assertions
  - For staging or untrusted data-sources
  - Again, can be “planned” out for performance at run-time
- Capture *really bad* data through “traps”
  - Depending on Fidelity of app
- Use s3n:// before s3://
  - S3:// stores data in proprietary format, never as default FS
  - Store compressed, in chunks (1 file per Mapper)

Cascading

- Open-Source/GPLv3
- 1.0 since January
- 1.1 Soon
- Growing community
- User contributed modules
- Elastic MapReduce

Upcoming

- **ScaleCamp - June 9th**
  - http://scalecamp.eventbrite.com/
- **Hadoop Summit - June 10th**
  - http://hadoopsummit09.eventbrite.com/

Resources

- **Cascading**
  - http://cascading.org
- **Enterprise tools and support**
  - http://concurrentinc.com
- **Professional Services and Training**
  - http://scaleunlimited.com