Faster Workflows, Faster

Ken Krugler | President, Scale Unlimited
The Twitter Pitch

- Cascading is a solid, established workflow API
  - Good for complex custom ETL workflows
- Flink is a new streaming dataflow engine
  - 50% better performance by leveraging memory

\[ \text{Cascading} + \text{Flink} = \text{Love} \]
Perils of Comparisons

• Performance comparisons are clickbait for devs
  • “You won't believe the speed of Flink!”
• Really, really hard to do well
• I did “moderate” tuning of existing code base
  • Somewhat complex workflow in EMR
  • Dataset bigger than memory (100M...1B records)
TL;DR

- Flink gets faster by minimizing disk I/O
- Map-Reduce job always has write/read at job break
  - Can also spill map output, reduce merge-sort
- Flink has no job boundaries
  - And no map-side spills
- So only reduce merge-sort is extra I/O
• A very short intro to Cascading
• An even shorter intro to Flink
• An example of converting a workflow
• More in-depth results
In the beginning...

• There was Hadoop, and it was good
• But life is too short to write M-R jobs with K-V data
• Then along came Cascading...
Cascading
What is Cascading?

- A thin Java library on top of Hadoop
- An open source project (APL, 8 years old)
- An API for defining and running ETL workflows
30,000ft View

- Records (Tuples) flow through Pipes

[ 'clay', 'boxing', '5' ]
30,000ft View

• Pipes connect Operations

[ ‘term’, ‘link’, ‘distance’ ]

[ ‘clay’, ‘boxing’, ‘5’ ]

Filter by Distance
30,000ft View

• You do Operations on Tuples
30,000ft View

- Tuples flow into Pipes from Source Taps

```
Hfs Tap

[ 'term', 'link', 'distance' ]
[ 'clay', 'boxing', '5' ]
[ 'clay', 'olympics', '15' ]
```
30,000ft View

- Tuples flow from Pipes into Sink Taps

```
[ 'term', 'link', 'weight' ]
[ 'clay', 'boxing', '2.57' ]
[ 'clay', 'pottery', '5.32' ]
```

Hfs Tap
30,000ft View

• This is a data processing workflow (Flow)
Java API to Define Flow

```java
Pipe ipDataPipe = new Pipe("ip data pipe");
RegexParser ipDataParser = new RegexParser(new Fields("Data IP", "Country"), "([\d.]+)\t(.*)");
ipDataPipe = new Each(ipDataPipe, new Fields("line"), ipDataParser);

Pipe logAnalysisPipe = new CoGroup( logDataPipe, // left-side pipe
    new Fields("Log IP"), // left-side field for joining
    ipDataPipe, // right-side pipe
    new Fields("Data IP"), // right-side field for joining
    new LeftJoin()); // type of join to do

logAnalysisPipe = new GroupBy(logAnalysisPipe, new Fields("Country", "Status"));
logAnalysisPipe = new Every(logAnalysisPipe, new Count(new Fields("Count")));
logAnalysisPipe = new Each(logAnalysisPipe, new Count(new Fields("country"), new Not(new RegexFilter("null"))));

Tap logDataTap = new Hfs(new TextLine(), "access.log");
Tap ipDataTap = new Hfs(new TextLine(), "ip-map.tsv");
Tap outputTap = new Hfs(new TextLine(), "results");

FlowDef flowDef = new FlowDef().setName("log analysis flow")
    .addSource(logDataPipe, logDataTap).addSource(ipDataPipe, ipDataTap)
    .addTailSink(logAnalysisPipe, outputTap);

Flow flow = new HadoopFlowConnector(properties).connect(flowDef);
```
Things I Like

- “Stream Shaping” - easy to add, drop fields
- Field consistency checking in DAG
- Building blocks - Operations, SubAssemblies
- Flexible planner - MR, local, Tez
Time for a Change

• I've used Cascading for 100s of projects
• And made a lot of money consulting on ETL
• But ... it was getting kind of boring
Flink
Elevator Pitch

• High throughput/low latency stream processing
• Also supports batch (bounded streams)
• Runs locally, stand-alone, or in YARN
• Super-awesome team
Versus Spark? Sigh...OK

• Very similar in many ways
• Natively streaming, vs. natively batch
• Not as mature, smaller community/ecosystem
Similar to Cascading

• You define a DAG with Java (or Scala) code
• You have data sources and sinks
• Data flows through streams to operators
• The planner turns this into a bunch of tasks
It's Faster, But...

• I don't want to rewrite my code
• I use lots of custom Cascading schemes
• I don't really know Scala
  • And POJOs ad nauseam are no fun
• Same for Tuple21<Integer, String, String, ...>
Scala is the New APL

```scala
val input = env.readFileStream(fileName,100)
  .flatMap { _.toLowerCase.split(\\W+) filter { _.nonEmpty } }
  .timeWindowAll(Time.of(60, TimeUnit.SECONDS))
  .trigger(ContinuousProcessingTimeTrigger.of(Time.seconds(5)))
  .fold(Set[String]()){(r,i) => { r + i}}
  .map{x => (new Timestamp(System.currentTimeMillis()), x.size)}
```
public void reduce(Iterable<Tuple2<Tuple3<String, String, Integer>,
   Tuple2<String, Integer>>> tupleGroup,
   Collector<Tuple3<String, String, Double>> out) {
   for (Tuple2<Tuple3<String, String, Integer>, Tuple2<String, Integer>>
   tuple : tupleGroup) {
      Tuple3<String, String, Integer> idWC = tuple.f0;
      Tuple2<String, Integer> idTW = tuple.f1;
      out.collect(new Tuple3<String, String, Double>(idWC.f0, idWC.f1,
         (double)idWC.f2 / idTW.f1));
   }
}
Cascading-Flink
Cascading 3 Planner

• Converts the Cascading DAG into a Flink DAG
• Around 5K lines of code
• DAG it plans looks like Cascading local mode
Boundaries for Data Sets

- Speed == no spill to disk
- Task CPU is the same
- Other than serde time
How Painful?

• Use the FlinkFlowConnector
  • Flink Flow planner to convert DAG to job
• Uber jar vs. classic Hadoop jar
• Grungy details of submitting jobs to EMR cluster
Wikiwords Workflow

• Find association between terms and categories
• For every page in Wikipedia, for every term
  • Find distance from term to intra-wiki links
• Then calc statistics to find “strong association”
  • Prob unusually high that term is close to link
Timing Test Details

- EMR cluster with 5 i2.xlarge slaves
- ~1 billion input records (term, article ref, distance)
- Hadoop MapReduce took 148 minutes
- Flink took 98 minutes
- So 1.5x faster - nice but not great
  - Mostly due to spillage in many boundaries
Summary
If You're a Java ELT Dev

• And you have to deal with batch big data
• Then the Cascading API is a good fit
• And using Flink typically gives better performance
• While still using a standard Hadoop/YARN cluster
Status of Cascading-Flink

• Still young, but surprisingly robust
• Doesn't support Full or RightOuter HashJoins
• Pending optimizations
  • Tuple serialization
  • Flink improvements
Better Planning...

- Defer late-stage join
- Avoid premature resource usage
More questions?

• Feel free to contact me
  • http://www.scaleunlimited.com/contact/
  • ken@scaleunlimited.com

• Check out Cascading, Flink, and Cascading-Flink
  • http://www.cascading.org
  • http://flink.apache.org
  • http://github.com/dataArtisans/cascading-flink