hack/reduce

Scalding and the ICGC Dataset

Scalding

- Developed by Twitter
- Scala DSL over the Cascading library
- Functional-oriented (uses Scala's Iterable trait)
- Cascading: data processing with Hadoop without the Map/Reduce paradigm
- Allows you to focus on **what** you want to do instead of **how**

Cascading Word Count

```
// define source and sink Taps.
Scheme sourceScheme = new TextLine( new Fields( "line" ) );
Tap source = new Hfs( sourceScheme, inputPath );

Scheme sinkScheme = new TextLine( new Fields( "word", "count" ) );
Tap sink = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );

// the 'head' of the pipe assembly
Pipe assembly = new Pipe( "wordcount" );

// For each input Tuple
// parse out each word into a new Tuple with the field name "word"
// regular expressions are optional in Cascading
String regex = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
Function function = new RegexGenerator( new Fields( "word" ), regex );
assembly = new Each( assembly, new Fields( "line" ), function );
```

continued...

```
// group the Tuple stream by the "word" value
assembly = new GroupBy( assembly, new Fields( "word" ) );

// For every Tuple group count the number of occurrences of "word" and store
// result in a field named "count"
Aggregator count = new Count( new Fields( "count" ) );
assembly = new Every( assembly, count );

// initialize app properties, tell Hadoop which jar file to use
Properties properties = new Properties();
FlowConnector.setApplicationJarClass( properties, Main.class );

// plan a new Flow from the assembly using the source and sink Taps
// with the above properties
FlowConnector flowConnector = new FlowConnector( properties );
Flow flow = flowConnector.connect( "word-count", source, sink, assembly );
// execute the flow, block until complete
flow.complete();
```

Scalding Word Count

```
import com.twitter.scalding._

class WordCountJob(args : Args) extends Job(args) {
  TextLine(args("input"))
    .flatMap('line -> 'word) { line : String => line.split("\\s+") }
    .groupBy('word) { _.size }
    .write(Tsv(args("output")))
}
```

Dataset

- **Simulated** dataset of variations and mutations
- Relational-oriented
  - ~15 TSV files
  - Approximately one SQL table per file
- Requires joining files all the time

Question

- Gene <-> Cancer interactions for people aged between 40 and 60
- Files required:
  - donor.tsv : age, cancer type (30K lines)
  - gene_info.tsv : gene symbol (42K lines)
  - ssm_m.tsv: donor mutations
  - ssm_p.tsv: mutation details (type, quality, location, etc.) 90M lines
  - ssm_s.tsv: mutation "consequence" : which gene is affected by the mutation

Steps

1. Get Scalding to work on the cluster
2. Join files
3. Group by gene,country count number of mutations
4. Make a graph to visualize
    a. nodes are genes and countries
    b. edges are weighted by mutation count

e.g.:

BRCA1,CA,456

BRCA1,FI,345

...

Code!

Results

- What we learned
  - Random really is random!
  - Nothing funky required to get Scalding to work
  - Cascading is very powerful for creating distributed / parallel file processing pipelines
  - Scalding removes **all** boilerplate from your code
  - A little bit of Scala
- What we **didn't** learn
  - Map/Reduce