Who is this guy?

- Staff Engineer, Compute and Data Services, Ooyala
- Building multiple web-scale real-time systems on top of C*, Kafka, Storm, etc.
- Scala/Akka guy
- Very excited by open source, big data projects
- @evanfchan
• Ooyala and Big Data
• What problem are we trying to solve?
• Spark and Shark
• Our Spark/Cassandra Architecture
• Spark Job Server
OOYALA AND BIG DATA
OOYALA

Powering personalized video experiences across all screens.
COMPANY OVERVIEW

Founded in 2007

Commercially launch in 2009

230+ employees in Silicon Valley, LA, NYC, London, Paris, Tokyo, Sydney & Guadalajara

Global footprint, 200M unique users, 110+ countries, and more than 6,000 websites

Over 1 billion videos played per month and 2 billion analytic events per day

25% of U.S. online viewers watch video powered by Ooyala
We have a large Big Data stack

- > 250GB of fresh logs every day
- Total of 28TB of data managed over ~200 Cassandra nodes
- Traditional stack: Hadoop, Ruby, Cassandra, Ruby...
- Real-time stack: Kafka, Storm, Scala, Cassandra
- New stack: Kafka, Akka, Cassandra, Spark, Scala/Go
Becoming a big Spark user...

- Started investing in Spark beginning of 2013
- 2 teams of developers doing stuff with Spark
- Actively contributing to Spark developer community
- Deploying Spark to a large (>100 node) production cluster
- Spark community very active, huge amount of interest
WHAT PROBLEM ARE WE TRYING TO SOLVE?
From mountains of raw data...
To nuggets of truth...

- Quickly
- Painlessly
- At scale?
Today: Precomputed Aggregates

• Video metrics computed along several high cardinality dimensions
• Very fast lookups, but inflexible, and hard to change
• Most computed aggregates are never read
• What if we need more dynamic queries?
  • Top content for mobile users in France
  • Engagement curves for users who watched recommendations
  • Data mining, trends, machine learning
THE STATIC - DYNAMIC CONTINUUM

100% Precomputation

- Super fast lookups
- Inflexible, wasteful
- Best for 80% most common queries

100% Dynamic

- Always compute results from raw data
- Flexible but slow
WHERE WE WANT TO BE

Partly *dynamic*

- Pre-aggregate most common queries
- Flexible, fast dynamic queries
- Easily generate many materialized views
INDUSTRY TRENDS

• Fast execution frameworks
  • Impala
• In-memory databases
  • VoltDB, Druid
• Streaming and real-time
• Higher-level, productive data frameworks
WHY SPARK?
THROUGHPUT: MEMORY IS KING

- **C*, cold cache**
- **C*, warm cache**
- **Spark RDD**

Spark cached RDD 10-50x faster than raw Cassandra

6-node C*/DSE 1.1.9 cluster, Spark 0.7.0

---

Thursday, August 29, 13
DEVELOPERS LOVE IT

• “I wrote my first aggregation job in 30 minutes”
• High level “distributed collections” API
• No Hadoop cruft
• Full power of Scala, Java, Python
• Interactive REPL shell
SPARK VS HADOOP WORD COUNT

define a Spark program to perform word count:

```python
file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" ")).map(word => (word, 1))

.reduceByKey(_ + _)
```

Then, define a Java program to perform word count using Hadoop:

```java
package org.myorg;

import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>
    {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>
    {
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.waitForCompletion(true);
    }
}
```

Both approaches aim to count the words in a file, but Spark's implementation is generally more efficient and easier to use due to its in-memory processing capabilities.
• Fewer platforms == lower TCO
• Much higher code sharing/reuse
• Spark/Shark/Streaming can replace Hadoop, Storm, and Impala
• Integration with Mesos, YARN helps
OUR SPARK ARCHITECTURE
From raw events to fast queries

Raw Events → Ingestion → C* event store → View 1 → Spark → View 2 → Spark → View 3 → Spark → View 1 → Spark → Predefined queries

Raw Events → Ingestion → C* event store → View 2 → Shark → Ad-hoc HiveQL
Our Spark/Shark/Cassandra Stack

Spark Master

Job Server

Spark Worker

Shark

SerDe

InputFormat

Cassandra

Node1

Spark Worker

Shark

SerDe

InputFormat

Cassandra

Node2

Spark Worker

Shark

SerDe

InputFormat

Cassandra

Node3

Spark Master
# Cassandra Schema

## Event CF

<table>
<thead>
<tr>
<th></th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>2013-04-05</td>
<td>T00:00Z#id1</td>
<td>{event0: a0}</td>
<td>{event1: a1}</td>
<td>{event2: a2}</td>
</tr>
</tbody>
</table>

## EventAttr CF

<table>
<thead>
<tr>
<th></th>
<th>ipaddr: 10.20.30.40:t1</th>
<th>videoid:45678:t1</th>
<th>providerId:500:t0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>2013-04-05 T00:00Z#id1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INPUTFORMAT VS RDD

• You can easily use InputFormats in Spark using newAPIHadoopRDD().
• Writing a custom RDD could have saved us lots of time.

<table>
<thead>
<tr>
<th>InputFormat</th>
<th>RDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supports Hadoop, HIVE, Spark, Shark</td>
<td>Spark / Shark only</td>
</tr>
<tr>
<td>Have to implement multiple classes</td>
<td>One class - simple API.</td>
</tr>
<tr>
<td>- InputFormat, RecordReader,</td>
<td></td>
</tr>
<tr>
<td>Writeable, etc. Clunky API.</td>
<td></td>
</tr>
<tr>
<td>Two APIs, and often need to</td>
<td>Just one API.</td>
</tr>
<tr>
<td>implement both (HIVE needs older...)</td>
<td></td>
</tr>
</tbody>
</table>
## Unpacking Raw Events

<table>
<thead>
<tr>
<th>Date</th>
<th>User ID</th>
<th>Video</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-04-05T00:00Z #id1</td>
<td>id1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>2013-04-05T00:00Z #id2</td>
<td></td>
<td>20</td>
<td>5</td>
</tr>
</tbody>
</table>

**UserID** | **Video** | **Type**
--- | --- | ---
id1 | 10 | 5

*Thursday, August 29, 13*
# Unpacking Raw Events

<table>
<thead>
<tr>
<th>UserID</th>
<th>Video</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>id1</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-04-05 T00:00Z#id {video: 10, type: 5}</td>
<td>{video: 11, type: 1}</td>
</tr>
<tr>
<td>2013-04-05 T00:00Z#id {video: 20, type: 5}</td>
<td>{video: 25, type: 9}</td>
</tr>
</tbody>
</table>
## Unpacking Raw Events

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>UserID</th>
<th>Video</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-04-05</td>
<td>00:00Z</td>
<td>id1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>2013-04-05</td>
<td>00:00Z</td>
<td>id1</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>2013-04-05</td>
<td>00:00Z</td>
<td>id2</td>
<td>20</td>
<td>5</td>
</tr>
</tbody>
</table>
## Unpacking Raw Events

<table>
<thead>
<tr>
<th></th>
<th>t0</th>
<th>t1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-04-05 T00:00Z#id</td>
<td>{video: 10, type: 5}</td>
<td>{video: 11, type: 1}</td>
</tr>
<tr>
<td>2013-04-05 T00:00Z#id</td>
<td>{video: 20, type: 5}</td>
<td>{video: 25, type: 9}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UserID</th>
<th>Video</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>id1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>id2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>id2</td>
<td>25</td>
<td>9</td>
</tr>
</tbody>
</table>
EXAMPLE: OLAP PROCESSING

C* events

Cached Materialized Views

OLAP Aggregates

Union

Query 1: Plays by Provider

Query 2: Top content for mobile

Spark

Spark

Spark

2013-04-05T00:00Z

{video: 10, type: 5}

2013-04-05T00:00Z

{video: 20, type: 5}

2013-04-05T00:00Z

{video: 30, type: 5}
PERFORMANCE #'S

Spark: C* -> OLAP aggregates
cold cache, 1.4 million events
130 seconds

C* -> OLAP aggregates
warmed cache
20-30 seconds

OLAP aggregate query via
Spark
(56k records)
60 ms

6-node C*/DSE 1.1.9 cluster,
Spark 0.7.0
OLAP WORKFLOW

Aggregate

Query

Result

Query

Result

REST Job Server

Aggregation Job

Dataset

Query Job

Query Job

Spark Executors

Cassandra
FAULT TOLERANCE

• Cached dataset lives in Java Heap only – what if process dies?

• Spark lineage – automatic recomputation from source, but this is expensive!

  • Can also replicate cached dataset to survive single node failures

• Persist materialized views back to C*, then load into cache -- now recovery path is much faster
SPARK JOB SERVER
JOB SERVER OVERVIEW

- Spark as a Service – Job, Jar, and Context management
- Run ad-hoc Spark jobs
- Great support for sharing cached RDDs across jobs and low-latency jobs
- Works with Standalone Spark as well as Mesos
- Jars and job history is persisted via pluggable API
- Async and sync API, JSON job results
- Contributing back to Spark community in the near future
/**
 * A super-simple Spark job example that implements the SparkJob trait and
 * can be submitted to the job server.
 */

object WordCountExample extends SparkJob {
    override def validate(sc: SparkContext, config: Config): SparkJobValidation = {
        Try(config.getString("input.string"))
            .map(x => SparkJobValid)
            .getOrElse(SparkJobInvalid("No input.string"))
    }

    override def runJob(sc: SparkContext, config: Config): Any = {
        val dd = sc.parallelize(config.getString("input.string").split(" ").toSeq)
        dd.map((_, 1)).reduceByKey(_ + _).collect().toMap
    }
}
SUBMITTING AND RUNNING A JOB

✦ curl --data-binary @../target/mydemo.jar localhost:8090/jars/demo
   OK[11:32 PM] ~

✦ curl -d "input.string = A lazy dog jumped mean dog" 'localhost:8090/jobs?
   appName=demo&classPath=WordCountExample&sync=true'

   {
     "status": "OK",
     "RESULT": {
       "lazy": 1,
       "jumped": 1,
       "A": 1,
       "mean": 1,
       "dog": 2
     }
   }
THANK YOU

And YES, We’re HIRING!!

oooyala.com/careers