Beyond Ad-hoc Data Science

Oscar Boykin
@posco
oscar@twitter.com
...towards a solution for reliable data pipelines
Ad-hoc Data Science

Tableau

Python

Scalding

Spark

R

presto
ML/Data Science in Prod

• Dashboard of unique active users by geography, client/ version.

• Process to train and evaluate click prediction models (joins impression features + user features and click events)

• Evaluate a classification model on an event to support search (show me news tweets).

• Eventually 100s - 1000s of jobs with complex dependencies.
How do things change?
How do things change?

Input changes
How do things change?
How do things change?
Basic Job Safety

• Tests! At Twitter, all production jobs should have tests. Virtually all jobs are in scalding (compiler checks and standard testing works).

• All jobs should be pure functions: output only depends on input, implies idempotency (safe to re-run the job).
How Inputs Change

- Data moves: new HDFS cluster, Kinesis to Kafka, private cluster to EMR,

- Product changes => logging changes.

- Data dependency is removed: some team stops producing a model/aggregate/feature.
How Output Changes

• Want to produce more detail/better model:
  
  • e.g. make a dashboard of unique users by geo/client+login frequency. \( \leq \) new data schema
  
  • consume more features to improve prediction
  
  • change modeling technique => rollout Deep Learning!!!
  
  • bugs are removed (and added)
$O(1000)$ batch jobs, $O(100)$ streaming jobs => things break all the time without some rigor.
It gets complicated
The Problem

- How to manage changing data and changing jobs.
- Somewhat unique to data science/ML which is interested in data-artifacts.
- Somewhat similar to the problem of compiling large code bases from complex source dependencies (but that generally has a simpler notion of time).
3 Solutions

- Restrict how schemas to evolve
- Decouple data locations from job logic
- Build a system to track data and job dependencies
Schemas

• a Good Thing!

• some folks like unstructured data.

• logic always requires some (subset of) a schema, better to be explicit.
Schemas

• At Twitter, all data has a schema which we can express using thrift. Written to HDFS/Kafka/etc…

• Thrift has boolean, numbers, strings, lists, sets, maps, structs, and unions. Structs members (columns) can be optional (nullable) or required (not nullable).

• Columns: (add|remove|switch) * (optional|required)
Thrift Example

struct UniqueUserCount {
    0: optional string geo;
    1: optional string client;
    2: required long timestamp;
    3: required long uniqueCount;
}
Thrift Example

struct UniqueUserCount {
  0: optional string geo;
  1: optional string client;
  2: required long timestamp;
  3: required long uniqueCount;
}

Can generalize this to other schema systems (SQL, protobuf, etc...)}
<table>
<thead>
<tr>
<th><strong>New Reader / Old Data</strong></th>
<th><strong>Ok (always absent)</strong></th>
<th><strong>Ok (ignored)</strong></th>
<th><strong>Fail (missing data)</strong></th>
<th><strong>Ok (ignored)</strong></th>
<th><strong>Ok (always present)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Old Reader / New Data</strong></td>
<td><strong>Ok (ignored)</strong></td>
<td><strong>Ok (always absent)</strong></td>
<td><strong>Fail (missing data)</strong></td>
<td><strong>Ok (ignored)</strong></td>
<td><strong>Ok (always present)</strong></td>
</tr>
</tbody>
</table>

- **add optional**
- **remove optional**
- **add required**
- **remove required**
- **opt -> req**
- **req -> opt**
<table>
<thead>
<tr>
<th>New Reader / Old Data</th>
<th>Old Reader / New Data</th>
<th>add optional</th>
<th>remove optional</th>
<th>add required</th>
<th>remove required</th>
<th>opt -&gt; req</th>
<th>req -&gt; opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ok (always absent)</td>
<td>Ok (always absent)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Ok (always absent)</td>
<td>Ok (always absent)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Ok (always absent)</td>
<td>Ok (always absent)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Ok (ignores)</td>
<td>Ok (ignores)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Fail (missing data)</td>
<td>Fail (missing data)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Fail (missing data)</td>
<td>Fail (missing data)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Fail (missing data)</td>
<td>Fail (missing data)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Fail (missing data)</td>
<td>Fail (missing data)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Ok (always present)</td>
<td>Ok (always present)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Fail (missing data)</td>
<td>Fail (missing data)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Fail (missing data)</td>
<td>Fail (missing data)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Fail (missing data)</td>
<td>Fail (missing data)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
<tr>
<td>Fail (missing data)</td>
<td>Fail (missing data)</td>
<td>add optional</td>
<td>remove optional</td>
<td>add required</td>
<td>remove required</td>
<td>opt -&gt; req</td>
<td>req -&gt; opt</td>
</tr>
</tbody>
</table>
How to make this work?

- Schemas live in the same repo as the code: jobs won’t compile if schema change out of sync with logic. Does not solve the problem of deployed jobs (can’t redeploy everything synchronously).

- Tool checks the previous version, ensures that all (public) schemas only add or remove optional fields. Otherwise, the check fails, and the code is not merged to master.

Decoupling Logic and Data
Storage

- At Twitter, production data pipelines are written in with scalding.
- Problem: overly tight coupling of paths to jobs.
case class Impression(id: Long, contentId: Long, timestamp: Long)
case class CountRecord(start: Long, end: Long, count: Long)

object MegaCount {

case class ImpressionSource(dateRange: DateRange) extends
    HourlyDataSource[Impression]("/logs/impression/", dateRange)

case class CountSink(dateRange: DateRange) extends
    HourlyDataSink[CountRecord]("/aggregates/unique_users/", dateRange)

val OneDay = 1000 * 86400 /* (millis / sec) * (secs / day) */

def job(dr: DateRange) = {
    TypedPipe.from(ImpressionSource(dr))
        .map { imp ⇒ (dayOf(imp.timestamp), Set(imp.id)) }
        .sumByKey
        .mapValues { set ⇒ set.size }
        .map { case (start, size) ⇒ CountRecord(start, start + OneDay, size) }
        .write(CountSink(dr))
}

Example Job
case class Impression(id: Long, contentId: Long, timestamp: Long)
case class CountRecord(start: Long, end: Long, count: Long)
case class ImpressionSource(dateRange: DateRange)

val OneDay = 1000 * 86400 / (millis / sec) * (secs / day)

val job(dr: DateRange) = {
  TypedPipe.from(ImpressionSource(dr))
    .map { imp => (dayOf(imp.timestamp), Set(imp.id)) }
    .sumByKey()
    .mapValues { set => size }
    .map { case (start, size) => CountRecord(start, start + OneDay, size) }
    .write(CountSink(dr))
}
case class Impression(id: Long, contentId: Long, timestamp: Long)

case class CountRecord(start: Long, end: Long, count: Long)

object MegaCount {
  case class ImpressionSource(dr: DateRange) extends Source[Impression]("impressions", dr)
  case class CountSink(dr: DateRange) extends Sink[CountRecord]("counts", dr)

  val OneDay = 1000 * 86400 /* (millis / sec) * (secs / day) */

  def job(dr: DateRange) = {
    TypedPipe.from(ImpressionSource(dr))
      .map { imp ⇒ (dayOf(imp.timestamp), Set(imp.id)) } .sumByKey
      .mapValues { set ⇒ set.size } .map { case (start, size) ⇒ CountRecord(start, start + OneDay, size) }
      .write(CountSink(dr))
  }
}
case class Impression(id: Long, contentId: Long, timestamp: Long)
case class CountRecord(start: Long, end: Long, count: Long)

object MegaCount {
  case class ImpressionSource(dr: DateRange) extends Source[Impression]("impressions", dr)
  case class CountSink(dr: DateRange) extends Sink[CountRecord]("counts", dr)

  val OneDay = 1000 * 86400 /* (millis / sec) * (secs / day) */

  def job(dr: DateRange) = {
    TypedPipe.from(ImpressionSource(dr))
      .map { imp ⇒ (dayOf(imp.timestamp), Set(imp.id)) }
      .sumByKey
      .mapValues { set ⇒ set.size }
      .map { case (start, size) ⇒ CountRecord(start, start + OneDay, size) }
      .write(CountSink(dr))
  }
}
Mappings

- Now we need a service that understands this mapping.

- Hive Metastore is a common choice (works well with Hive, Presto,...)

- Twitter has complex schemas, and needed better scalding support. Wanted a service that tracks not only data, but jobs.

Tracking Job Evolution
• Wanted a system that understands that jobs change, that schemas change.

• Wanted a full change log: append only structure (immutable records).

• How to handle bugfixes when you have an immutable log?
Logical Dataset
- role, env
- name
- schema
- isPublic

Application
- role, env
domain
name

AppComponent
- role, env
domain
name

composed of

Logical Dataset
- produces
- consumes

Physical Dataset
- rootLocation (URL)
dalv1URL
type: Partitioned|Snapshot|Stream

Deployment
- deployment info
- zone (aka dc)
- sha
- mesos role
- client type

LDAP
- group
  - controls
  - Service account (role)
- user

owns

owns
• Each segment has a physical schema that was actually used to write it.

• All the schemas for a given dataset are always compatible (never allow incompatible schema changes).

• When jobs are moved to new clusters, we can replicate needed segments automatically, and record their existence.
If we discover a bug in a given git sha, we can query to see which data was produced. Can re-run the job and downstream jobs to produce new revisions of segments.
Currently, jobs are run by a mesos cron scheduler. Read the job state to see if dependencies are ready. Can move to a non-polling model where jobs are run as soon as dependencies are available.

Open question how to best express (time) dependencies in jobs.

Alerting is external to this: alert if a given job has not had a successful run in the last time interval. Can explain why: which Jobs/Segments are blocking.
• We want to get have an artifact build system that works for thousands of jobs, running 10-100k instances per day, with 100-1000s of users.

• Logic and data dependencies in the same place, independent of physical storage/format/etc… details.

• Understands version control, can easily (automatically?) schedule backfills when a job’s code changes.

• Has a manageable model of time, understands goal deadlines and priorities. Can predict when a goal deadline can’t be met, good alerts (actionable, early)
Thank You!

@posco

oscar@twitter.com