Beyond Ad-hoc Data Science

Oscar Boykin
@posco
oscar@twitter.com

…towards a solution for reliable data pipelines

Ad-hoc Data Science

Spark

tableau

R

python™

presto

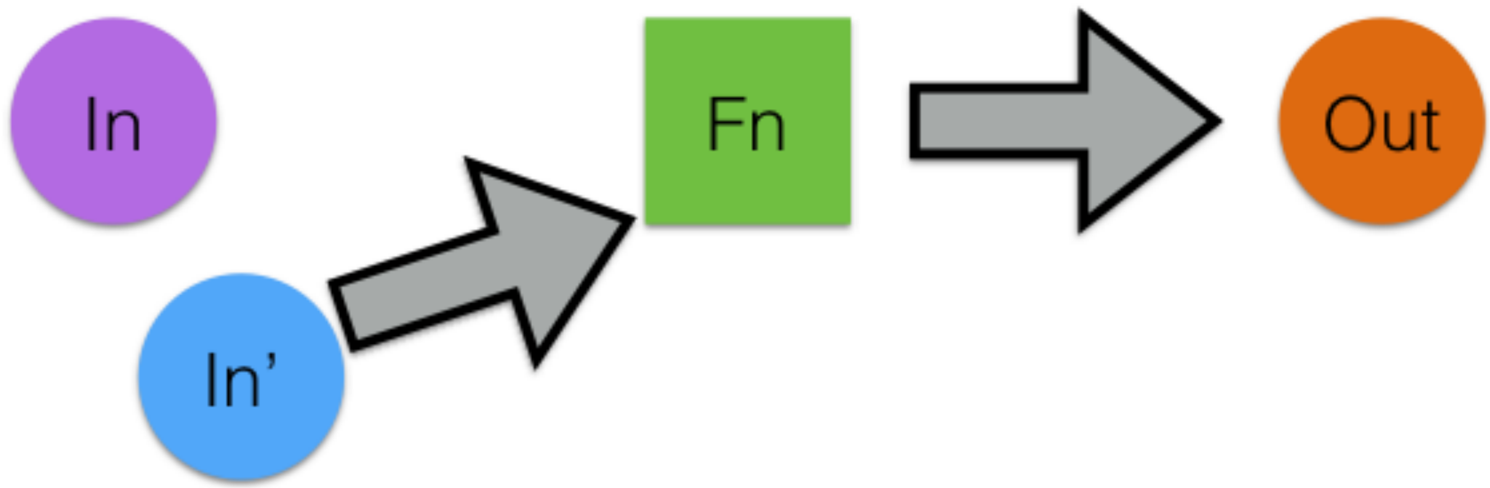
Scalding

ML/Data Science in Prod

- Dashboard of unique active users by geograph, 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?

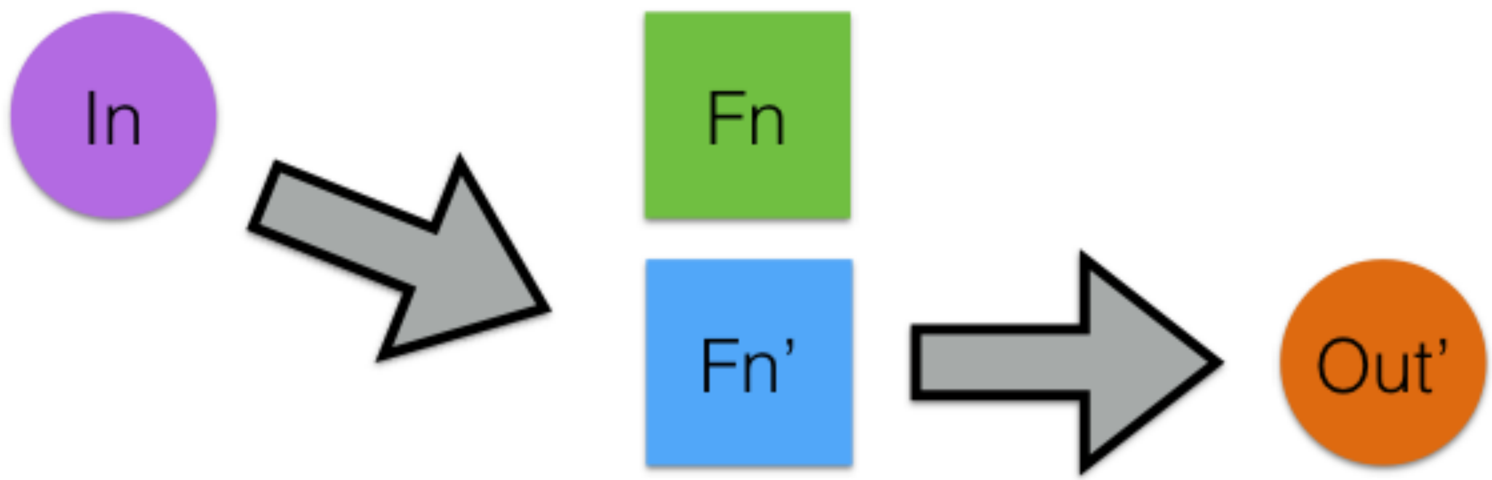
How do things change?



Input changes

How do things change?

How do things change?



Output changes

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

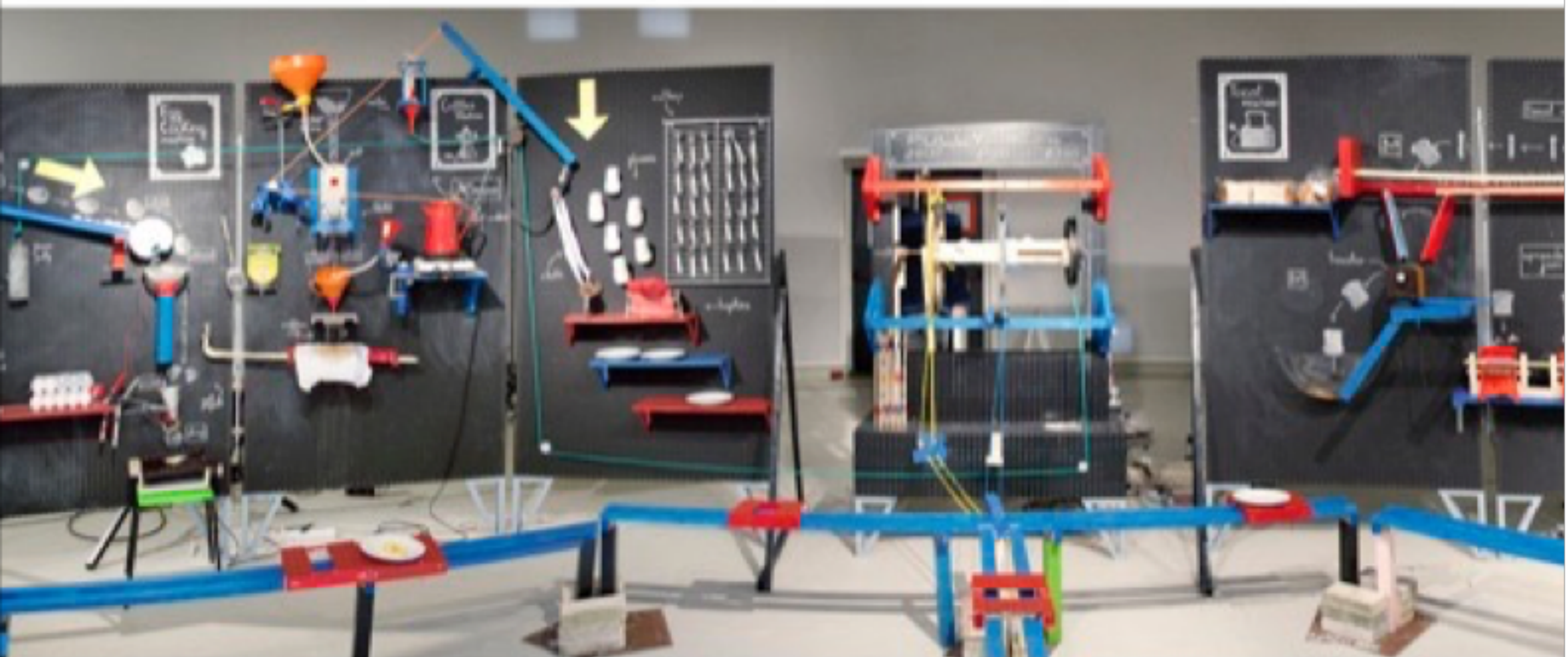
- Data moves: new HDFS cluster, Kinesis to Kafka, private cluster to EMR,
- Product changes => logging changes.
- Data dependency is removed: some team stops producing a model/aggregate/feature.

How Output Changes

- Want to produce more detail/better model:
    - e.g. make a dashboard of unique users by geo/client+login frequency. <= 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



Arne Hendrik's Breakfast Machine

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…)

	Old Reader / New Data	New Reader / Old Data
add optional	Ok (ignored)	Ok (always absent)
remove optional	Ok (always absent)	Ok (ignored)
add required	Ok (ignored)	**Fail** (missing data)
remove required	**Fail** (missing data)	Ok (ignored)
opt -> req	Ok (always present)	**Fail** (missing data)
req -> opt	**Fail** (missing data)	Ok (always present)

	Old Reader / New Data	New Reader / Old Data
add optional	Ok (ignored)	Ok (always absent)
remove optional	Ok (always absent)	Ok (ignored)
add required	Ok (ignored)	**Fail** (missing data)
remove required	**Fail** (missing data)	Ok (ignored)
opt -> req	Ok (always present)	**Fail** (missing data)
req -> opt	**Fail** (missing data)	Ok (always present)

How to make this work?

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

Decoupling Logic and Data Storage

- At Twitter, production data pipelines are written in with scalding.
- Problem: overly tight coupling of paths to jobs.

Example Job

```scala
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)

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))
  }
}
```

Physical paths
move data => breakage

Logical Names

Logical names

```
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))
  }
}
```

Logical Names

```scala
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))
  }
}
```

Logical names

Makes a service call at runtime, match schema, resolve paths

Tracking Logical to Physical

Mappings

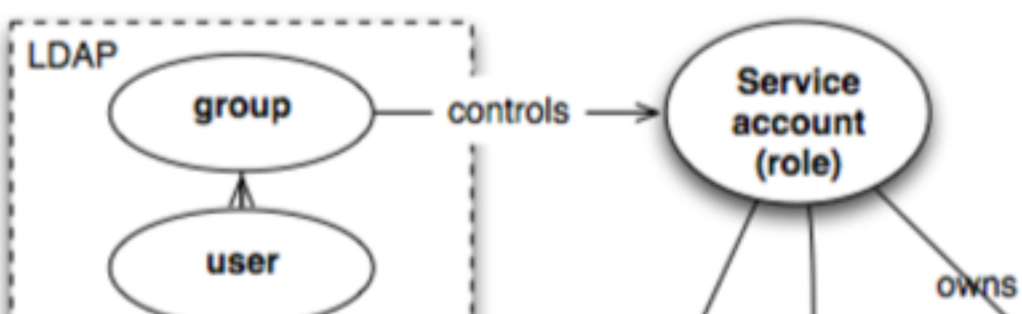
- Now we need a service that understands this mapping.
- Hive Metastore is a common choice (works well with Hive, Presto,…)
- Twitter has complex schemas, and needed better scalding support. Wanted a service that tracks not only data, but jobs.

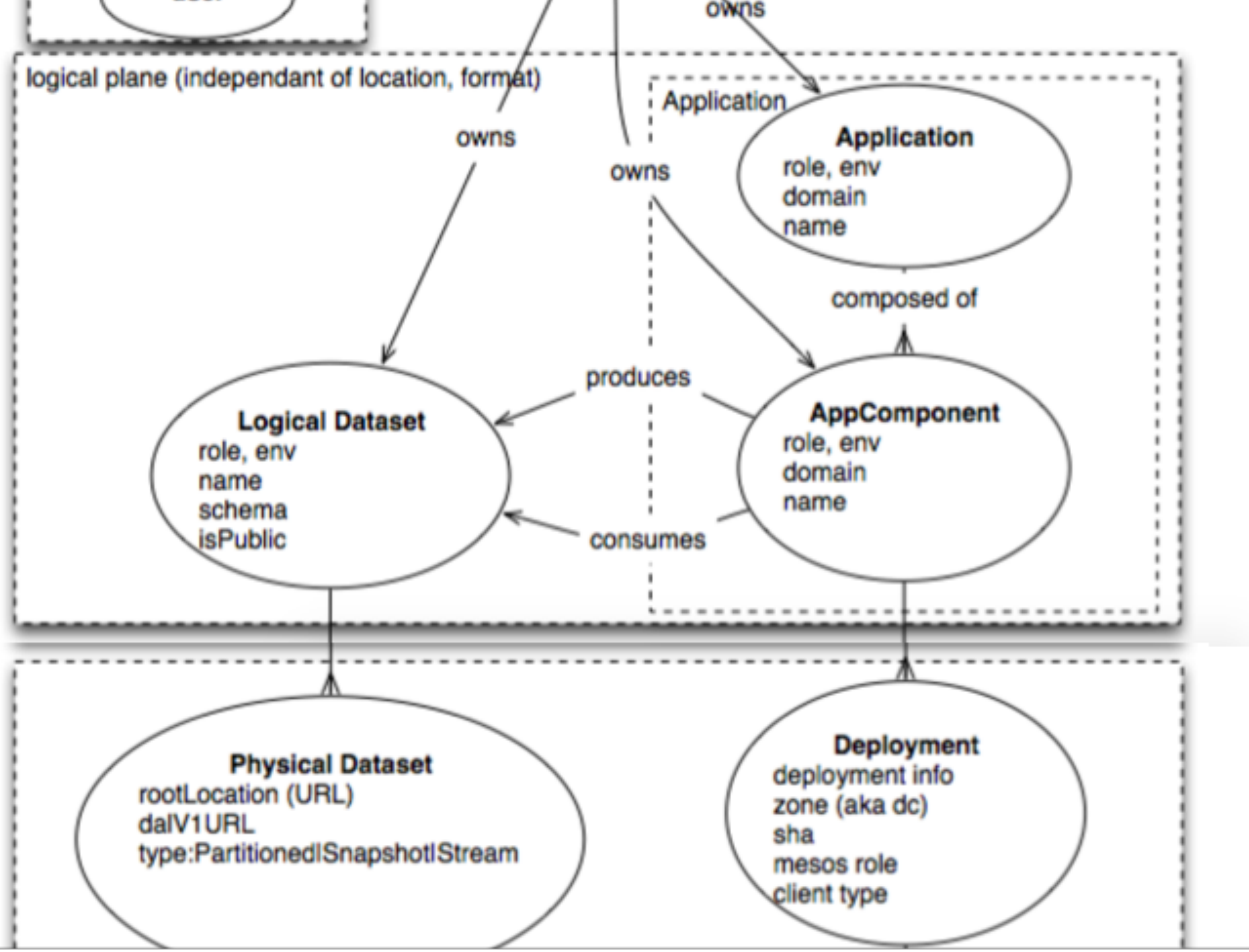
Tracking Job Evolution

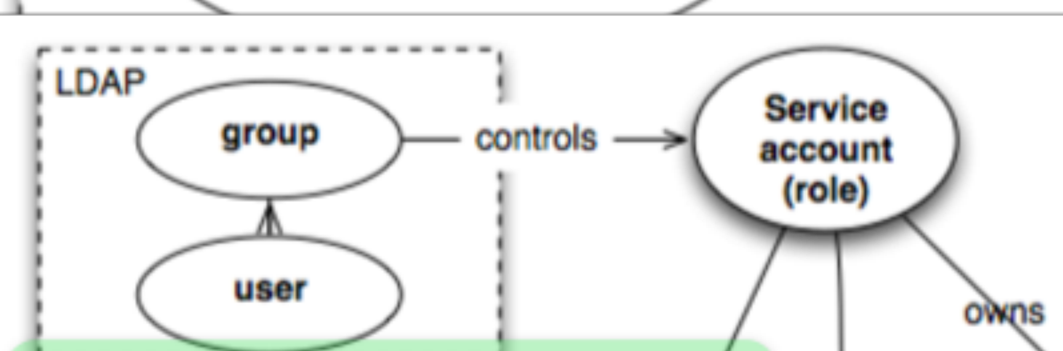
Tracking Job Evolution

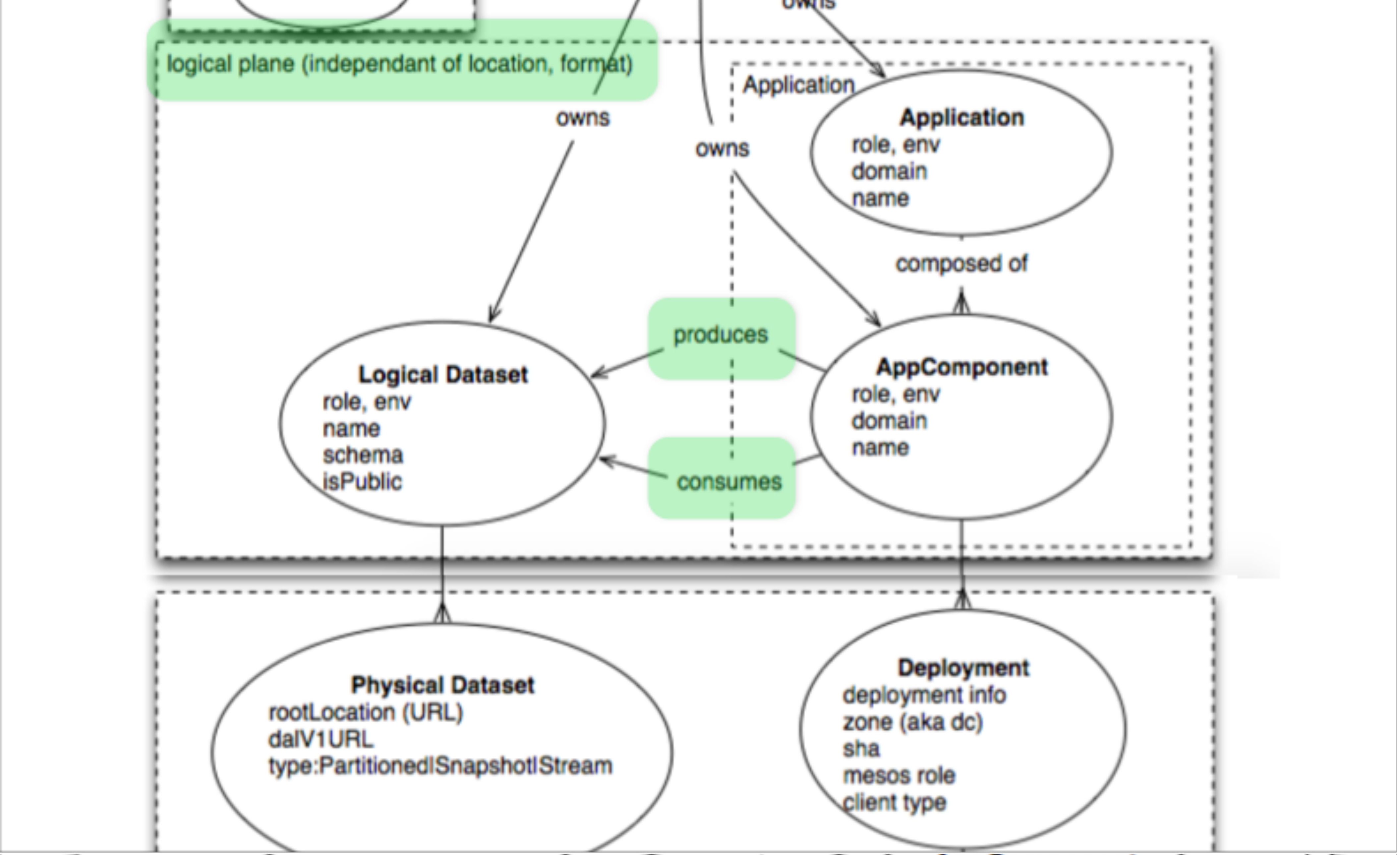
- Wanted a system that understands that jobs change, that schemas change.
- Wanted a full change log: append only structure (immutable records).
- How to handle bugfixes when you have an immutable log?

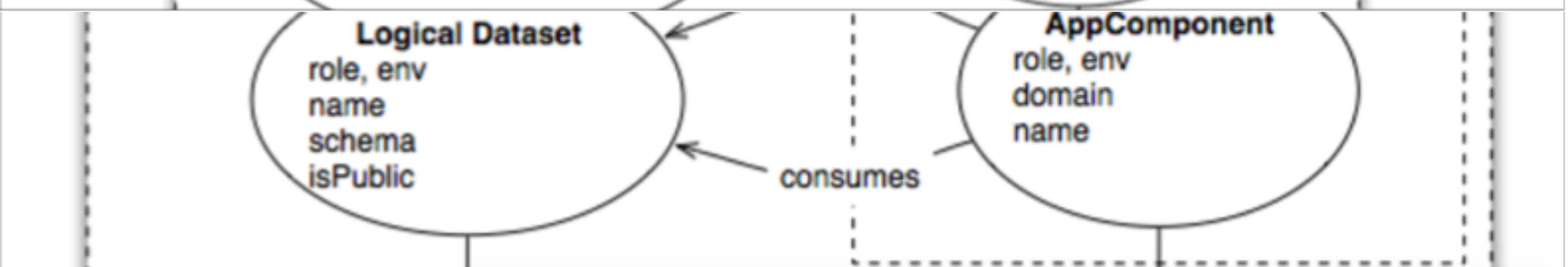
<design>

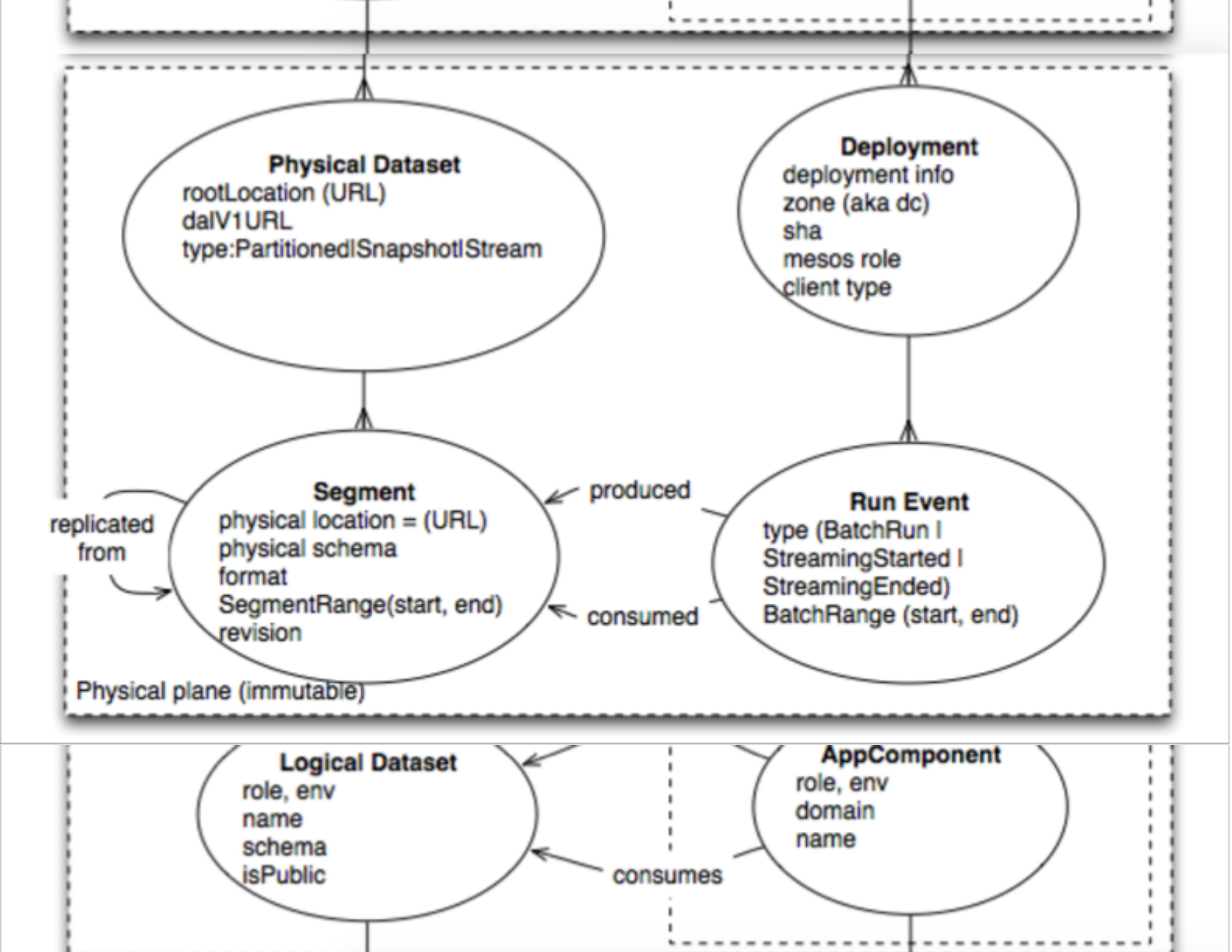



owns
logical plane (independant of location, format)
owns
owns
Application
Application
role, env
domain
name
composed of
produces
Logical Dataset
role, env
name
schema
isPublic
AppComponent
role, env
domain
name
consumes
Physical Dataset
rootLocation (URL)
dalV1URL
type:PartitionedlSnapshotlStream
Deployment
deployment info
zone (aka dc)
sha
mesos role
client type

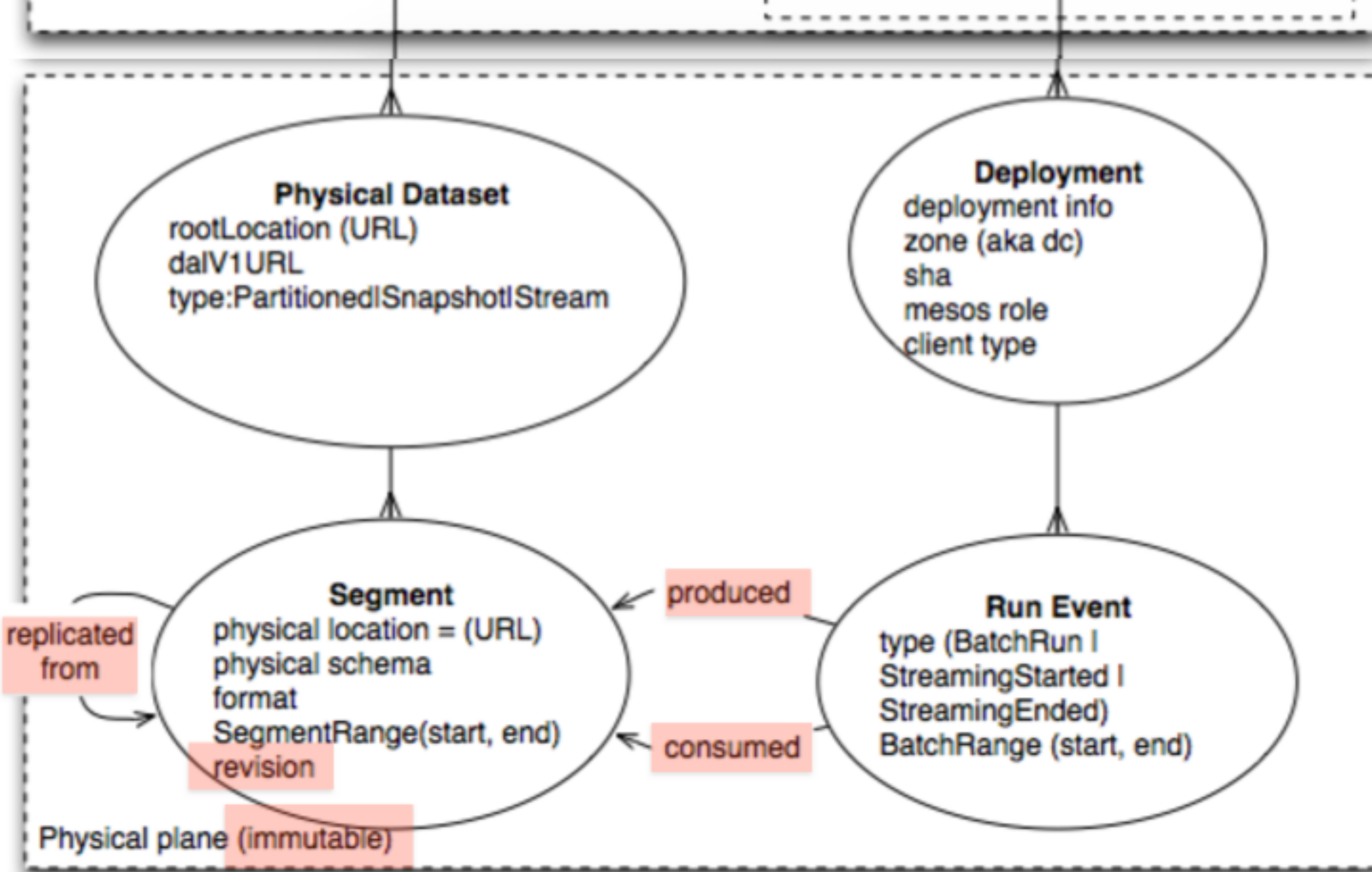

LDAP
group
controls
Service
account
(role)
user
owns


logical plane (independant of location, format)
owns
owns
owns
Application
Application
role, env
domain
name
composed of
produces
consumes
Logical Dataset
role, env
name
schema
isPublic
AppComponent
role, env
domain
name
Physical Dataset
rootLocation (URL)
dalV1URL
type:PartitionedISnapshotIStream
Deployment
deployment info
zone (aka dc)
sha
mesos role
client type


Logical Dataset
role, env
name
schema
isPublic
AppComponent
role, env
domain
name
consumes

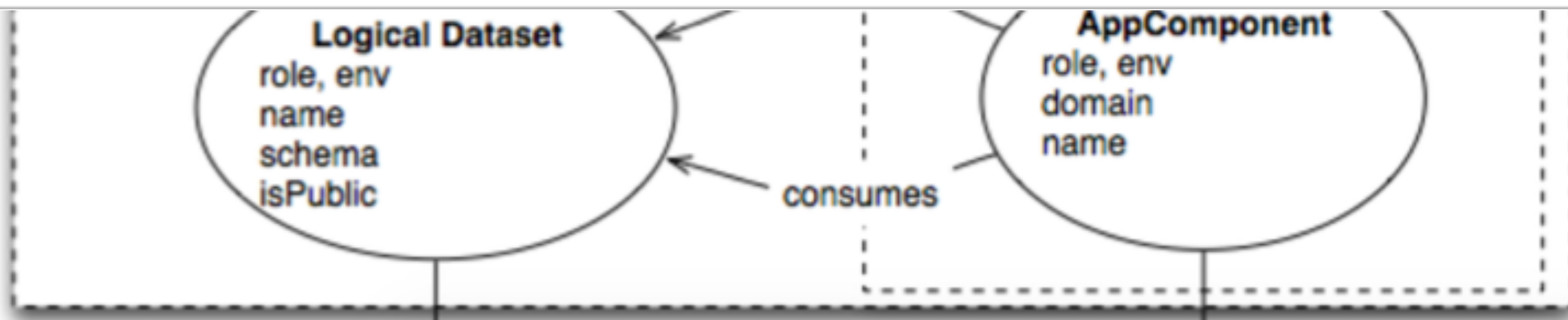


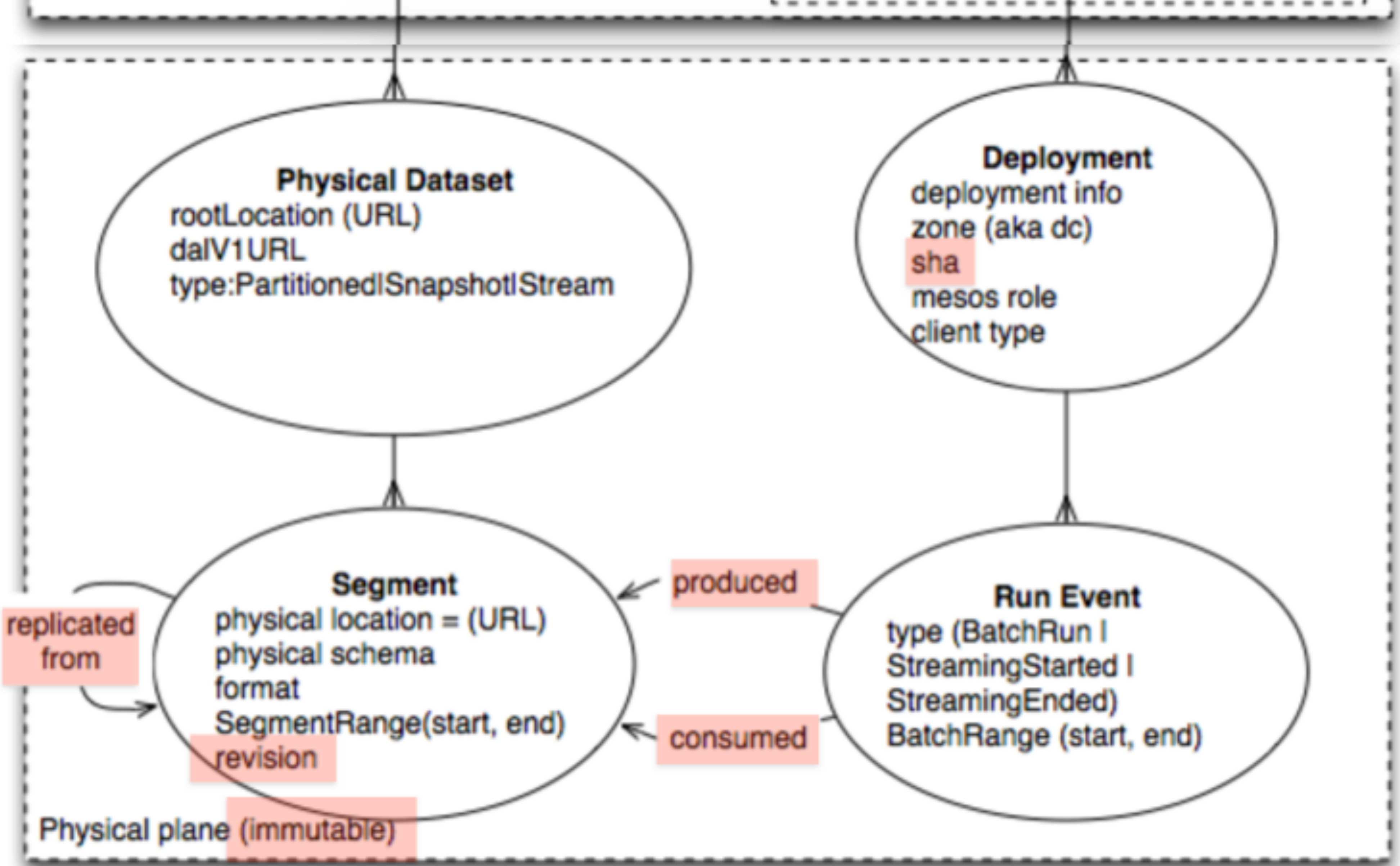
Physical Dataset
rootLocation (URL)
dalV1URL
type:PartitionedlSnapshotlStream
Deployment
deployment info
zone (aka dc)
sha
mesos role
client type
Segment
physical location = (URL)
physical schema
format
SegmentRange(start, end)
revision
replicated from
produced
consumed
Run Event
type (BatchRun l
StreamingStarted l
StreamingEnded)
BatchRange (start, end)
Physical plane (immutable)
Logical Dataset
role, env
name
schema
isPublic
AppComponent
role, env
domain
name
consumes



Physical Dataset
rootLocation (URL)
dalV1URL
type:PartitionedlSnapshotlStream
Deployment
deployment info
zone (aka dc)
sha
mesos role
client type
Segment
physical location = (URL)
physical schema
format
SegmentRange(start, end)
revision
Run Event
type (BatchRun l
StreamingStarted l
StreamingEnded)
BatchRange (start, end)
produced
consumed
replicated from
Physical plane (immutable)

- 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.



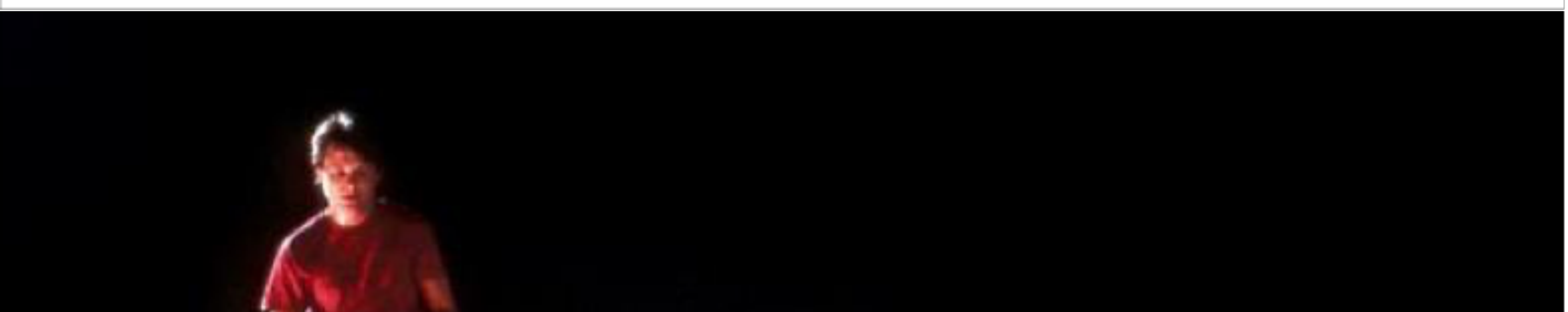

Physical Dataset
rootLocation (URL)
dalV1URL
type:PartitionedlSnapshotlStream
Deployment
deployment info
zone (aka dc)
sha
mesos role
client type
Segment
physical location = (URL)
physical schema
format
SegmentRange(start, end)
revision
replicated from
produced
consumed
Run Event
type (BatchRun l
StreamingStarted l
StreamingEnded)
BatchRange (start, end)
Physical plane (immutable)

- 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.

Other Issues

- 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.



The Future

- 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