What is TSAR?
What is TSAR?

TSAR is a framework and service infrastructure for specifying, deploying and operating timeseries aggregation jobs.
TimeSeries Aggregation at Twitter
TimeSeries Aggregation at Twitter

A common problem

- Data products (analytics.twitter.com, embedded analytics, internal dashboards)
- Business metrics
- Site traffic, service health, and user engagement monitoring

Hard to do at scale

- 10s of billions of events/day in real time

Hard to maintain aggregation services once deployed -

- Complex tooling is required
TimeSeries Aggregation at Twitter
TimeSeries Aggregation at Twitter

Many time-series applications look similar

- Common types of aggregations
- Similar service stacks

Multi-year effort to build general solutions

- Summingbird - abstraction library for generalized distributed computation

TSAR - an end-to-end aggregation service built on Summingbird

- Abstracts away everything except application's data model and business logic
A typical aggregation

- Event logs
- Extract
- Group
- Measure
- Store
Event logs

[ ("/", 300, iPhone,...) 
, ("/favorites", 300, iPhone,...) 
, ("/replies", 300, Android,...) 
, ("/", 200, Web,...) 
, ("/favorites", 200, Web,...) 
, ("/", 200, iPhone,...) ]
[ ("/", 300),
  ("/favorites", 300),
  ("/replies", 300),
  ("/", 200),
  ("/favorites", 200),
  ("/", 200) ]
Example: API aggregates
Example: API aggregates

- Bucket each API call
- **Dimensions** - endpoint, datacenter, client application ID
- **Compute** - total event count, unique users, mean response time etc
- Write the output to Vertica
Example: Impressions by Tweet
Example: Impressions by Tweet

- Bucket each impressions by tweet ID
- Compute total count, unique users
- Write output to a key-value store
- Expose output via a high-SLA query service
- Write sample of data to Vertica for cross-validation
Problems
Problems

- **Service interruption**: Can we retrieve lost data?

- **Data schema coordination**: Store output as log data, in a key-value data store, in cache, and in relational databases

- **Flexible schema change**

- **Easy to backfill and update/repair historical data**

Most important: Solve these problems in a general way.
TSAR’s design principles
TSAR’s design principles

1) Hybrid computation: Build on Summingbird, process each event twice - in real time & in batch (at a later time)

- Gives stability and reproducibility of batch
- Streaming (recency) of realtime

Leverage the Summingbird ecosystem:
- Abstraction framework over computing platforms
- Rich library of approximation monoids (Algebird)
- Storage abstractions (Storehaus)
Figure 1.11 Lambda Architecture diagram
TSAR’s design principles
2) Separate event production from event aggregation

User specifies how to extract events from source data

Bucketing and aggregating events is managed by TSAR
TSAR’s design principles
TSAR’s design principles

3) Unified data schema:
   - Data schema specified in datastore-independent way
   - Managed schema evolution & data transformation

Store data on:
   - HDFS
   - Manhattan (key-value)
   - Vertica/MySQL
   - Cache

Easily extensible to other schemas (Cassandra, HBase, etc)
Data Schema Coordination

- Schema consistent across stores
- Update stores when schema evolves
TSAR’s design principles
4) Integrated service toolkit

- One-stop deployment tooling
- Data warehousing
- Query capability
- Automatic observability and alerting
- Automatic data integrity checks
Tweet Impressions in TSAR
Tweet Impressions in TSAR

- Annotate each tweet with an impression count
- Count = unique users who saw that tweet
- Massive scalability challenge:
  - > 500MM tweets/day
  - tens of billions of impressions
- Want realtime updates
- Production ready and robust
A minimal Tsar project

- Scala Tsar job
- Configuration File
- Thrift IDL
struct TweetAttributes
{
    1: optional i64 tweet_id
}
Tweet Impressions Example

Scala Tsar job
Tweet Impressions Example

```scala
aggregate {
  onKeys(
    (TweetId)
  ) produce (
    Count
  ) sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}
```
Tweet Impressions Example

Scala Tsar job
Tweet Impressions Example

```scala
aggregate {
  onKeys{
    (TweetId)
  } produce {
    Count
  } sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
```

Scala Tsar job
Tweet Impressions Example

aggregate {
  onKeys {
    (TweetId)
  }
  produce {
    (TweetId)
  }
  sinkTo (Manhattan)
}

fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}
Tweet Impressions Example

aggregate {
  onKeys((TweetId))
  ) produce (Count)
  ) sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
  .filter { event => isImpressionEvent(event) }
  .map { event =>
    val impr = ImpressionAttributes(event.tweetId)
    (event.timestamp, impr)
  }
}

Dimensions for job aggregation
Tweet Impressions Example

Scala Tsar job
Tweet Impressions Example

```scala
aggregate {
  onKeys(
    (TweetId)
  ) produce (
    Count
  ) sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}
```
Tweet Impressions Example

```scala
aggregate {
  onKeys(
    (TweetId)
  ) produce (
    Count
  ) sinkTo (Manhattan)
}
fromProducer {
  ClientEventSource("client_events")
  .filter { event => isImpressionEvent(event) }
  .map { event =>
    val impr = ImpressionAttributes(event.tweetId)
    (event.timestamp, impr)
  }
}
```
aggregate {
  onKeys((TweetId))
  ) produce (Count)
  ) sinkTo (Manhattan)
}
fromProducer {
ClientEventSource("client_events")
  .filter { event => isImpressionEvent(event) }
  .map { event =>
    val impr = ImpressionAttributes(event.tweetId)
    (event.timestamp, impr)
  }
}

Metrics to compute
Tweet Impressions Example

Scala Tsar job
Tweet Impressions Example

```scala
aggregate {
  onKeys(
    (TweetId)
  ) produce (Count)
  sinkTo (Manhattan)
}
fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}
```
Tweet Impressions Example

aggregate {
  onKeys(
    (TweetId)
  ) produce (
    Count
  ) sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}
Tweet Impressions Example

Scala Tsar job
Tweet Impressions Example

```scala
aggregate {
  onKeys(
    (TweetId)
  ) produce (
    Count
  ) sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}
```
Tweet Impressions Example

aggregate {
    onKeys(
        (TweetId)
    )
    produce (
        Count
    )
    sinkTo (Manhattan)
} fromProducer {
    ClientEventSource("client_events")
        .filter { event => isImpressionEvent(event) }
        .map { event =>
            val impr = ImpressionAttributes(event.tweetId)
            (event.timestamp, impr)
        }
}
Tweet Impressions Example

aggregate {
  onKeys(
    (TweetId)
  ) produce (Count)
  sinkTo (Manhattan)
} fromProducer {
  ClientEventSource(“client_events”)
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}
Tweet Impressions Example

```scala
aggregate {
  onKeys(
    (TweetId)
  ) produce (Count)
  sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}

Summingbird fragment to describe event production.
```
Tweet Impressions Example

aggregate {
  onKeys {
    (TweetId)
  }
  produce {
    Count
  }
  sinkTo (Manhattan)
}
fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.tweetId)
      (event.timestamp, impr)
    }
}

Summingbird fragment to describe event production.

There is no aggregation logic specified here.
Config(
    base = Base(
        namespace = 'tsar-examples',
        name = 'tweets',
        user = 'tsar-shared',
        thriftAttributesName = 'TweetAttributes',
        origin = '2014-05-15 00:00:00 UTC',

        jobclass = 'com.twitter.platform.analytics.examples.TweetJob',

        outputs = [ 
            Output(sink = Sink.IntermediateThrift, width = 1 * Day),
            Output(sink = Sink.Manhattan1, width = 1 * Day)
        ],
    ...
)
What has been specified?
What has been specified?

- Our event schema (in thrift)
- How to **produce** these events
- **Dimensions** to aggregate on
- **Time granularities** to aggregate on
- **Sinks** (Manhattan / MySQL) to use
What do you not specify?

- How to represent the aggregated data
- How to represent the schema in MySQL / Manhattan
- How to perform the aggregation
- How to locate and connect to underlying services (Hadoop, Storm, MySQL, Manhattan, …)
Operational simplicity

End-to-end service infrastructure with a single command

$ tsar deploy --env=prod

- Launch Hadoop jobs
- Launch Storm jobs
- Launch Thrift query service
- Launch loader processes to load data into MySQL / Manhattan
- Mesos configs for all of the above
- Alerts for the batch & storm jobs and the query service
- Observability for the query service
- Auto-create tables and views in MySQL or Vertica
- Automatic data regression and data anomaly checks
Bird’s eye view of the TSAR pipeline
Seamless Schema evolution

Scala Tsar job
Seamless Schema evolution

Break down impressions by the client application (Twitter for iPhone, Twitter for Android etc)

aggregate {
  onKeys(
    (TweetId),
    (TweetId, ClientApplicationId)
  ) produce (
    Count
  ) sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) } 
    .map { event =>
      val impr = ImpressionAttributes(event.client, event.tweetId)
      (event.timestamp, impr)
    }
}
Seamless Schema evolution

Break down impressions by the client application (Twitter for iPhone, Twitter for Android etc)

aggregate {
  onKeys(
    (TweetId),
    (TweetId, ClientApplicationId)
  ) produce (Count)
  sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.client, event.tweetId)
      (event.timestamp, impr)
    }
}
Seamless Schema evolution

Break down impressions by the client application (Twitter for iPhone, Twitter for Android etc)

aggregate {
  onKeys(
    (TweetId),
    (TweetId, ClientApplicationId)
  ) produce (Count)
  sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(event.client, event.tweetId)
      (event.timestamp, impr)
    }
}
Backfill tooling
Backfill tooling

But what about historical data?
Backfill tooling

But what about historical data?

tsar backfill —start=<start> —end=<end>
Backfill tooling

But what about historical data?

tsar backfill —start=<start> —end=<end>

Backfill runs parallel to the production job

Useful for repairing historical data as well
Aggregating on different granularities
We have been computing only *daily aggregates*. We now wish to add *alltime aggregates*.
Aggregating on different granularities

We have been computing only *daily aggregates*.
We now wish to add *alltime aggregates*.

Output(sink = Sink.Manhattan, width = 1 * Day)
Output(sink = Sink.Manhattan, width = Alltime)
Aggregating on different granularities

We have been computing only daily aggregates
We now wish to add alltime aggregates

Output(sink = Sink.Manhattan, width = 1 * Day)
Output(sink = Sink.Manhattan, width = Alltime)
Aggregating on different granularities

We have been computing only *daily aggregates*
We now wish to add *alltime aggregates*

```
Output(sink = Sink.Manhattan, width = 1 * Day)
Output(sink = Sink.Manhattan, width = Alltime)
```

New aggregation granularity
Automatic metric computation

Scala Tsar job
Automatic metric computation

So far, only total view counts.
Now, add # unique users viewing each tweet

```
aggregate {
  onKeys(
    (TweetId),
    (TweetId, ClientApplicationId)
  ) produce (Count,
    Unique(UserId)
  ) sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(
        event.client, event.userId, event.tweetId
      )
      (event.timestamp, impr)
    }
}
Automatic metric computation

So far, only total view counts.
Now, add # unique users viewing each tweet

```
aggregate {
  onKeys(
    (TweetId),
    (TweetId, ClientApplicationId)
  ) produce (
    Count,
    Unique(UserId)
  ) sinkTo (Manhattan)
} fromProducer {
  ClientEventSource("client_events")
    .filter { event => isImpressionEvent(event) }
    .map { event =>
      val impr = ImpressionAttributes(
        event.client, event.userIds, event.tweetId
      )
      (event.timestamp, impr)
    }
```
Automatic metric computation

So far, only total view counts.
Now, add # unique users viewing each tweet

```
aggregate {
    onKeys(
        (TweetId),
        (TweetId, ClientApplicationId)
    ) produce (
        Count,
        Unique(UserId)
    ) sinkTo (Manhattan)
} fromProducer {
    ClientEventSource("client_events")
       .filter { event => isImpressionEvent(event) }
       .map { event =>
            val impr = ImpressionAttributes(
                event.client, event.userId, event.tweetId
            )
            (event.timestamp, impr)
        }
```
Support for multiple sinks
Support for multiple sinks

So far, only persisting data to Manhattan

Persist data to MySQL as well
Support for multiple sinks

So far, only persisting data to Manhattan

Persist data to MySQL as well

Output(sink = Sink.Manhattan, width = 1 * Day)
Output(sink = Sink.Manhattan, width = Alltime)
Output(sink = Sink.MySQL, width = Alltime)
Support for multiple sinks

So far, only persisting data to Manhattan

Persist data to MySQL as well

Output(sink = Sink.Manhattan, width = 1 * Day)
Output(sink = Sink.Manhattan, width = Alltime)
Output(sink = Sink.MySQL, width = Alltime)
Support for multiple sinks

So far, only persisting data to Manhattan

Persist data to MySQL as well

Output(sink = Sink.Manhattan, width = 1 * Day)
Output(sink = Sink.Manhattan, width = Alltime)
Output(sink = Sink.MySQL, width = Alltime)
Tsar Workflow

Create Job

Deploy Job

Modify Job

Optional: Backfill Job
TSAR Optimizations
Cache TSAR Pipeline

Logs -> Intermediate Thrift -> Total Thrift

aggregate {
  onKeys {
    (TweetIdField),                               // Total
    (ClientApplicationIdField),                 // Total
    (TweetIdField, ClientApplicationIdField)    // Intermediate
  }
  produce {
    ...
  }
  sinkTo {
    ...
  }
}
Covering Template

aggregate {
  onKeys (
    (TweetIdField),
    (ClientApplicationIdField),
    (TweetIdField, ClientApplicationIdField) // Covering Template
  )
  produce ( ... )
  sinkTo ( ... )
}
Intermediate Sinks

```python
Config(
base = Base(
    namespace = 'tsar-examples',
    name = 'tweets',
    user = 'tsar-shared',
    thriftAttributesName = 'TweetAttributes',
    origin = '2014-05-15 00:00:00 UTC',
    jobclass = 'com.twitter.platform.analytics.examples.TweetJob',
    outputs = [
        Output(sink = Sink.IntermediateThrift, width = 1 * Day),
        Output(sink = Sink.TotalThrift, width = 1 * Day),
        Output(sink = Sink.Manhattan1, width = 1 * Day),
        Output(sink = Sink.Vertica, width = 1 * Day)
    ],
),
...
Trade space for time
Manhattan - Key Clustering

Reduces Manhattan queries for large time ranges
Manhattan - Key Clustering

2014-01-01 12:00:00 → 2014-01-01
2014-01-01 13:00:00

2014-01-02 12:00:00 → 2014-01-02
2014-01-02 13:00:00

2014-01-03 12:00:00 → 2014-01-03
2014-01-03 12:00:00

1 Day Clustering
### Value Packing - Key Indexer

Some keys are always queried together

<table>
<thead>
<tr>
<th>Location</th>
<th>Impressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>All campaigns</td>
<td>1,177,780</td>
</tr>
<tr>
<td>United States</td>
<td>610,968</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>186,313</td>
</tr>
<tr>
<td>Canada</td>
<td>127,818</td>
</tr>
<tr>
<td>Spain</td>
<td>69,910</td>
</tr>
<tr>
<td>France</td>
<td>58,335</td>
</tr>
<tr>
<td>Netherlands</td>
<td>49,483</td>
</tr>
<tr>
<td>Germany</td>
<td>41,632</td>
</tr>
<tr>
<td>Belgium</td>
<td>23,610</td>
</tr>
<tr>
<td>Ireland</td>
<td>8,676</td>
</tr>
</tbody>
</table>
Naive approach:

(CampaignIdField, CountryField) —> Impressions

Would have to query:

(CampaignIdField, USA) —> Impressions_USA
(CampaignIdField, UK) —> Impressions_UK
...

Better approach:

(CampaignIdField) —> Map[CountryField -> Impressions]

Would have to query:

(CampaignIdField) —> Map[USA -> Impressions_USA,
    UK  -> Impressions_UK,...]

- Limits key fanout

- Implicit index on CountryField - don’t need to know which countries to query for
TSAR Visualization
TSAR Visualization
Conclusion: Three basic problems
Conclusion: Three basic problems

- **Computation management**
  - Describe and execute computational logic
  - Specify aggregation dimensions, metrics and time granularities

- **Dataset management**
  - Define, deploy and evolve data schemas
  - Coordinate data migration, backfill and recovery

- **Service management**
  - Define query services, observability, alerting, regression checks, coordinate deployment across all underlying services

TSAR gives you all of the above
Key Takeaway
Key Takeaway

“The end-to-end management of the data pipeline is TSAR’s key feature. The user concentrates on the business logic.”
Thank you!

Questions?

@anirudhtodiani @ twitter