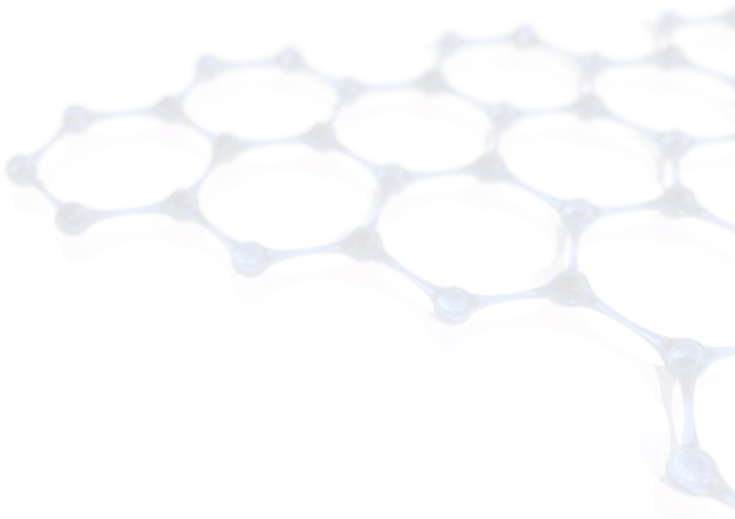




Real-time Learning for Fun and Profit

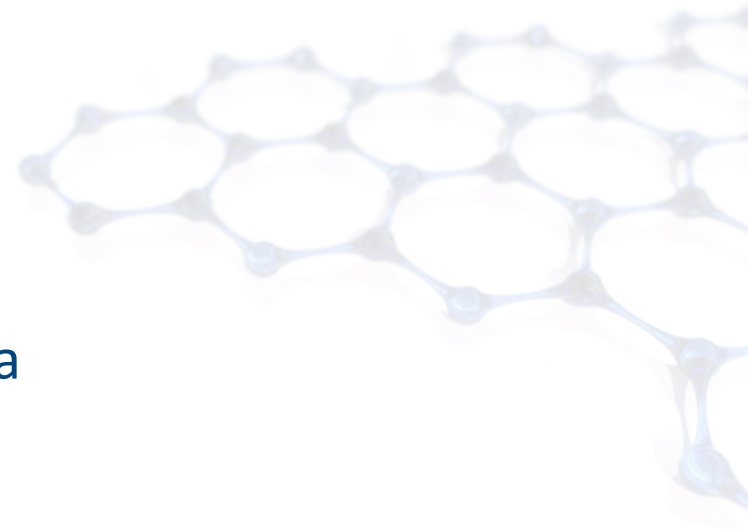
- 
- **Contact:**
 - tdunning@maprtech.com
 - @ted_dunning

 - **Slides and such (available late tonight):**
 - <http://slideshare.net/tdunning>

 - **Hash tags: #mapr #storm #bbuzz**

The Challenge

- Hadoop is great at processing vast amounts of data
 - But sucks for real-time (by design!)
- Storm is great for real-time processing
 - But lacks any way to deal with batch processing
- It sounds like there isn't a solution
 - Neither fashionable solution handles everything

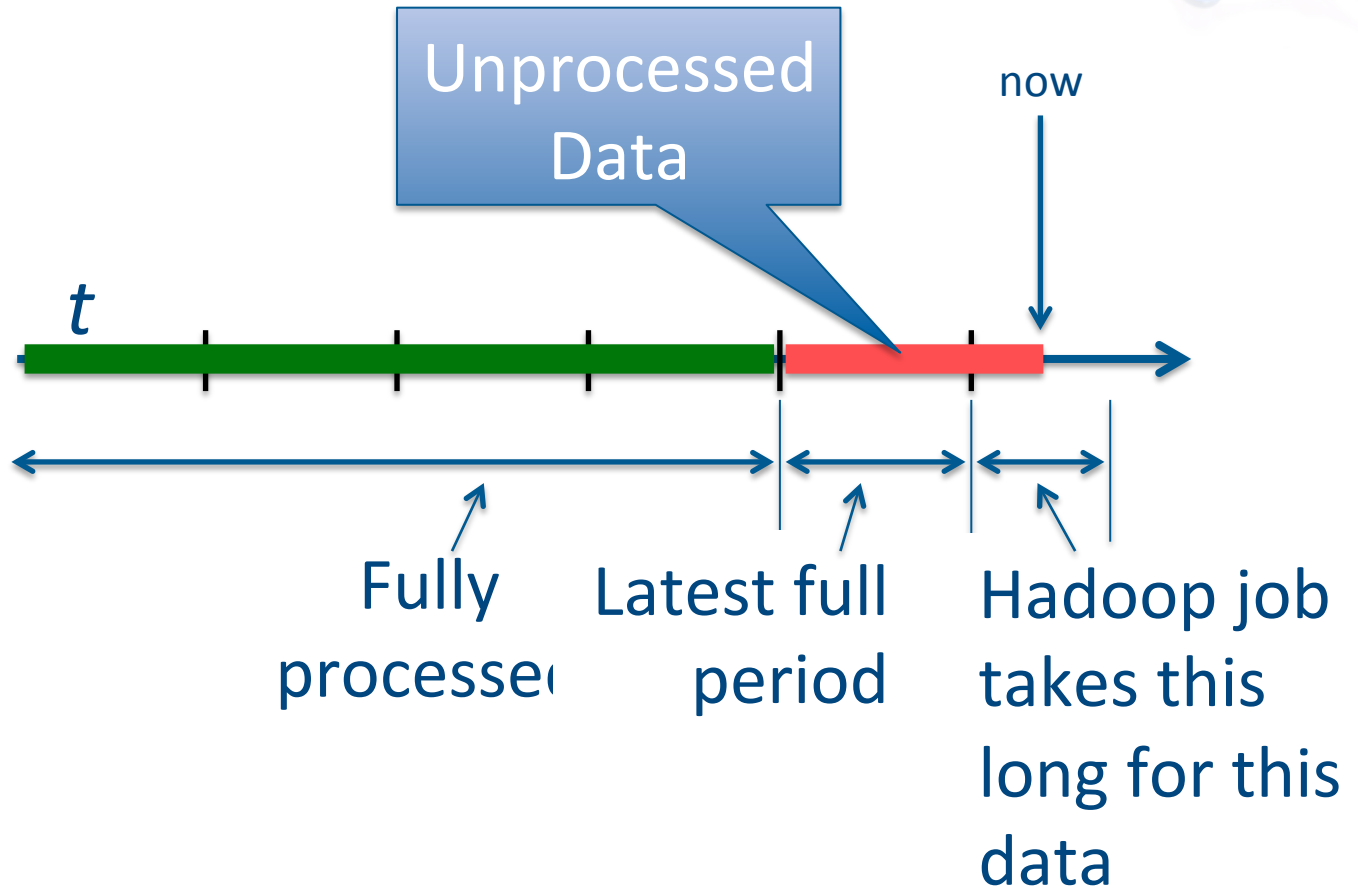




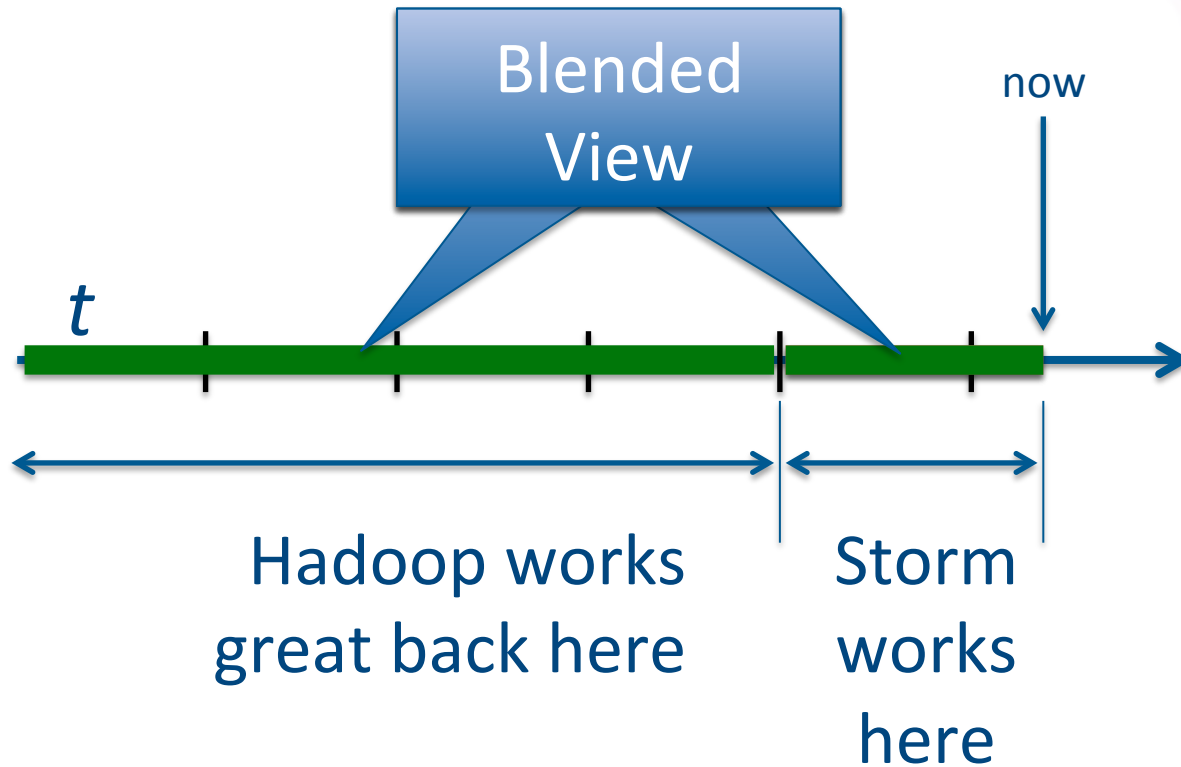
This is not a problem.

It's an opportunity!

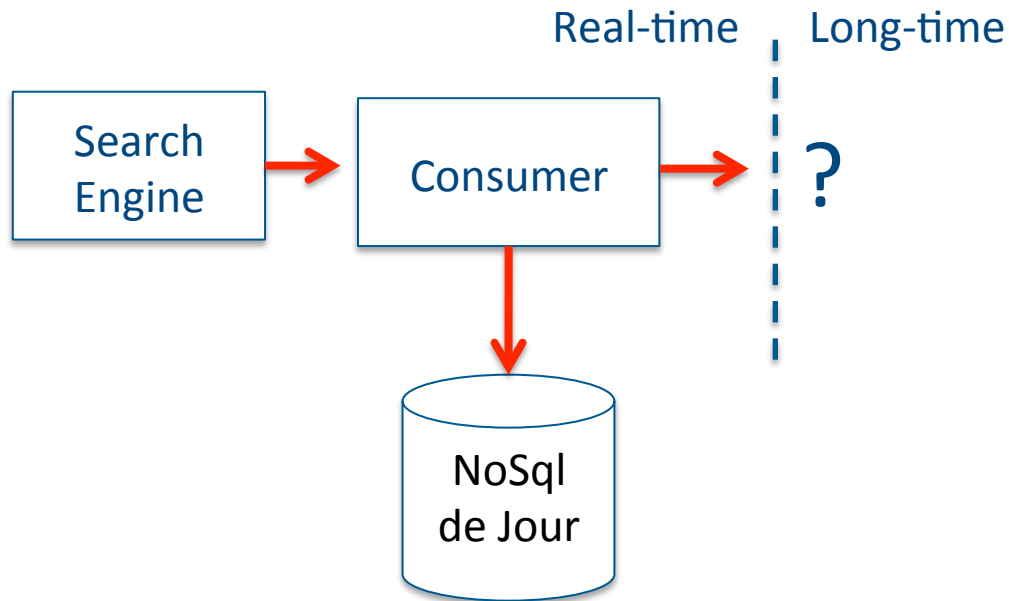
Hadoop is Not Very Real-time



Real-time and Long-time together



One Alternative



Problems

- Simply dumping into noSql engine doesn't quite work
- Insert rate is limited
- No load isolation
 - Big retrospective jobs kill real-time
- Low scan performance
 - Hbase pretty good, but not stellar
- Difficult to set boundaries
 - where does real-time end and long-time begin?

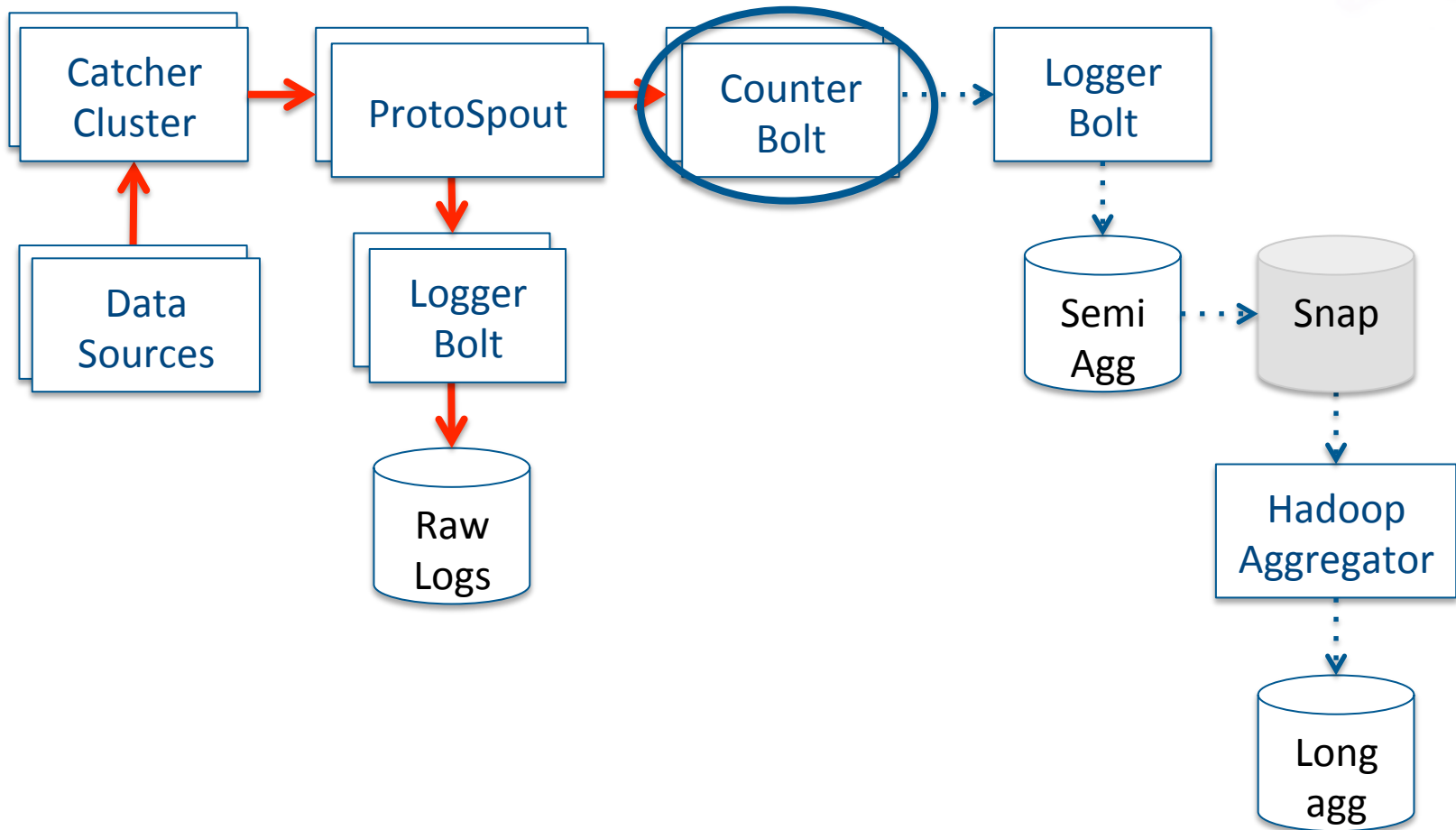
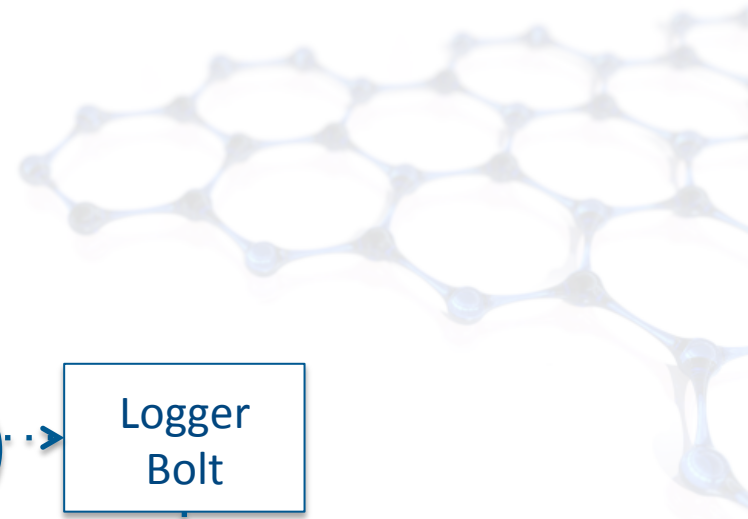
Almost a Solution

- Lambda architecture talks about function of long-time state
 - Real-time approximate accelerator adjusts previous result to current state
- Sounds good, but ...
 - How does the real-time accelerator combine with long-time?
 - What algorithms can do this?
 - How can we avoid gaps and overlaps and other errors?
- Needs more work

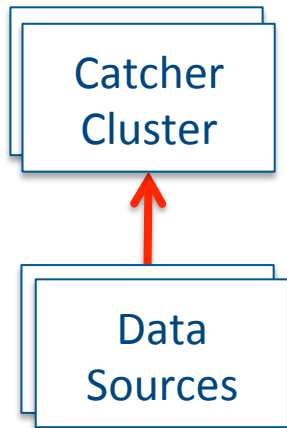
A Simple Example

- Let's start with the simplest case ... counting
- Counting = addition
 - Addition is associative
 - Addition is on-line
 - We can generalize these results to all associative, on-line functions
 - But let's start simple

Rough Design – Data Flow



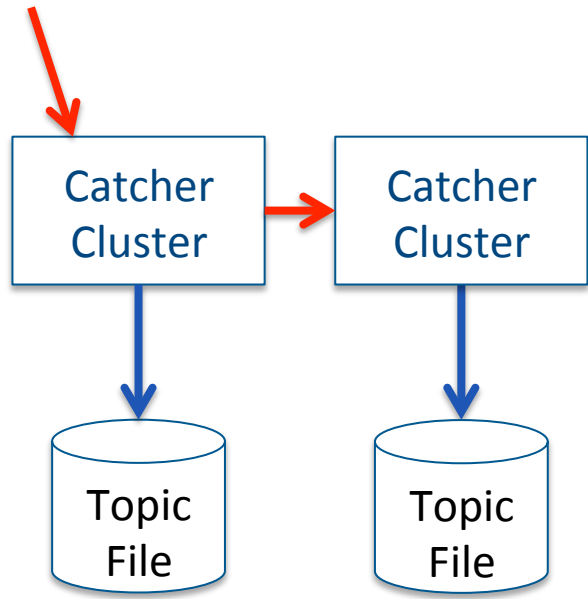
Closer Look – Catcher Protocol



The data sources and catchers communicate with a very simple protocol.

Hello() => list of catchers
Log(topic,message) =>
(OK|FAIL, redirect-to-catcher)

Closer Look – Catcher Queues

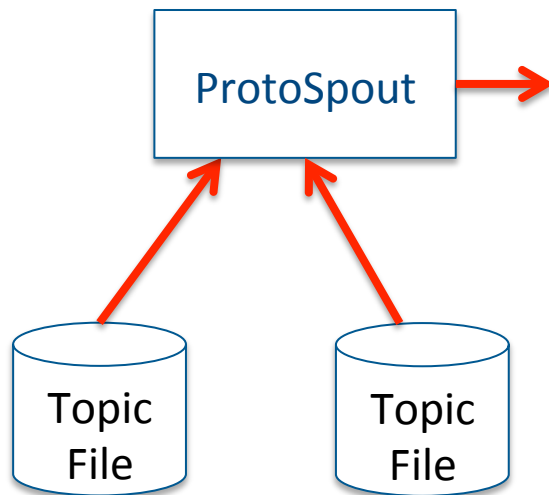


The catchers forward log requests to the correct catcher and return that host in the reply to allow the client to avoid the extra hop.

Each topic file is appended by exactly one catcher.

Topic files are kept in shared file storage.

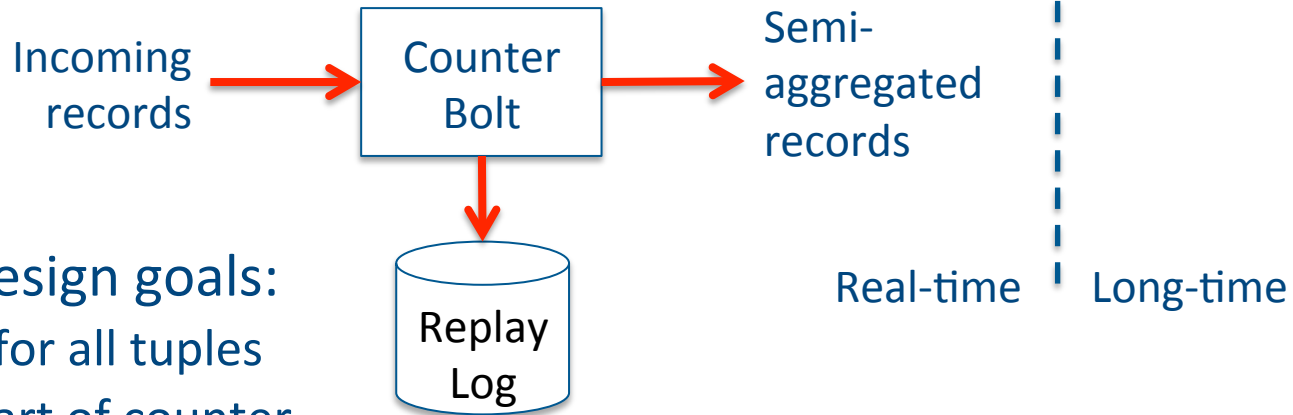
Closer Look – ProtoSpout



The ProtoSpout tails the topic files, parses log records into tuples and injects them into the Storm topology.

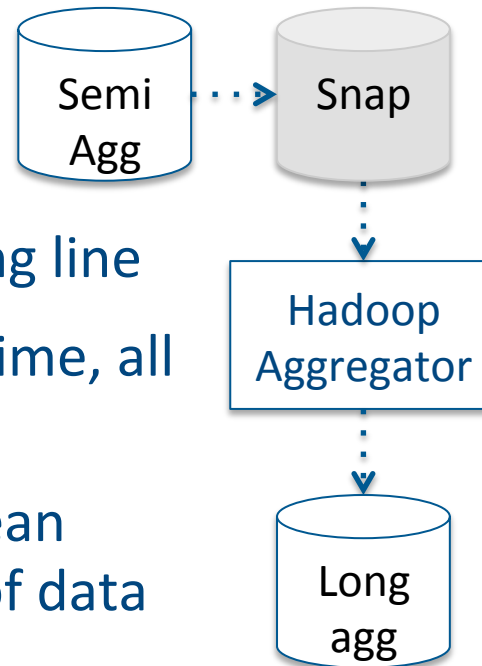
Last fully acked position stored in shared, transactionally correct file system.

Closer Look – Counter Bolt



- Critical design goals:
 - fast ack for all tuples
 - fast restart of counter
- Ack happens when tuple hits the replay log (10's of milliseconds, group commit)
- Restart involves replaying semi-agg's + replay log (very fast)
- Replay log only lasts until next semi-aggregate goes out

A Frozen Moment in Time



- Snapshot defines the dividing line
- All data in the snap is long-time, all after is real-time
- Semi-agg strategy allows clean combination of both kinds of data
- Data synchronized snap not needed

Guarantees

- Counter output volume is small-ish
 - the greater of k tuples per 100K inputs or k tuple/s
 - 1 tuple/s/label/bolt for this exercise
- Persistence layer must provide guarantees
 - distributed against node failure
 - must have either readable flush or closed-append
- HDFS is distributed, but provides no guarantees and strange semantics
- MapRfs is distributed, provides all necessary guarantees

Presentation Layer

- Presentation must
 - read recent output of Logger bolt
 - read relevant output of Hadoop jobs
 - combine semi-aggregated records
- User will see
 - counts that increment within 0-2 s of events
 - seamless and accurate meld of short and long-term data

The Basic Idea

- Online algorithms generally have relatively small state (like counting)
- Online algorithms generally have a simple update (like counting)
- If we can do this with counting, we can do it with all kinds of algorithms

Summary – Part 1

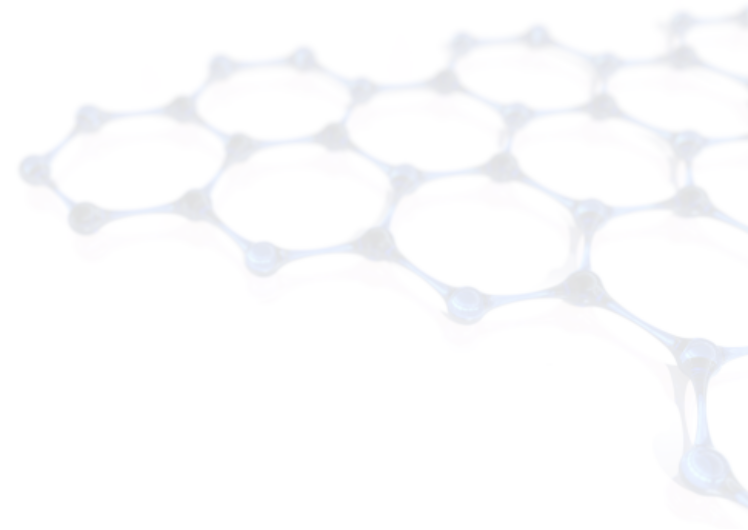
- Semi-agg strategy + snapshots allows correct real-time counts
 - because addition is on-line and associative
- Other on-line associative operations include:
 - *k*-means clustering (see Dan Filimon’s talk at 16.)
 - count distinct (see hyper-log-log counters from streamlib or kmv from Brickhouse)
 - top-*k* values
 - top-*k* (count(*)) (see streamlib)
 - contextual Bayesian bandits (see part 2 of this talk)

Example 2 – AB testing in real-time

- I have 15 versions of my landing page
- Each visitor is assigned to a version
 - Which version?
- A conversion or sale or whatever can happen
 - How long to wait?
- Some versions of the landing page are horrible
 - Don't want to give them traffic

A Quick Diversion

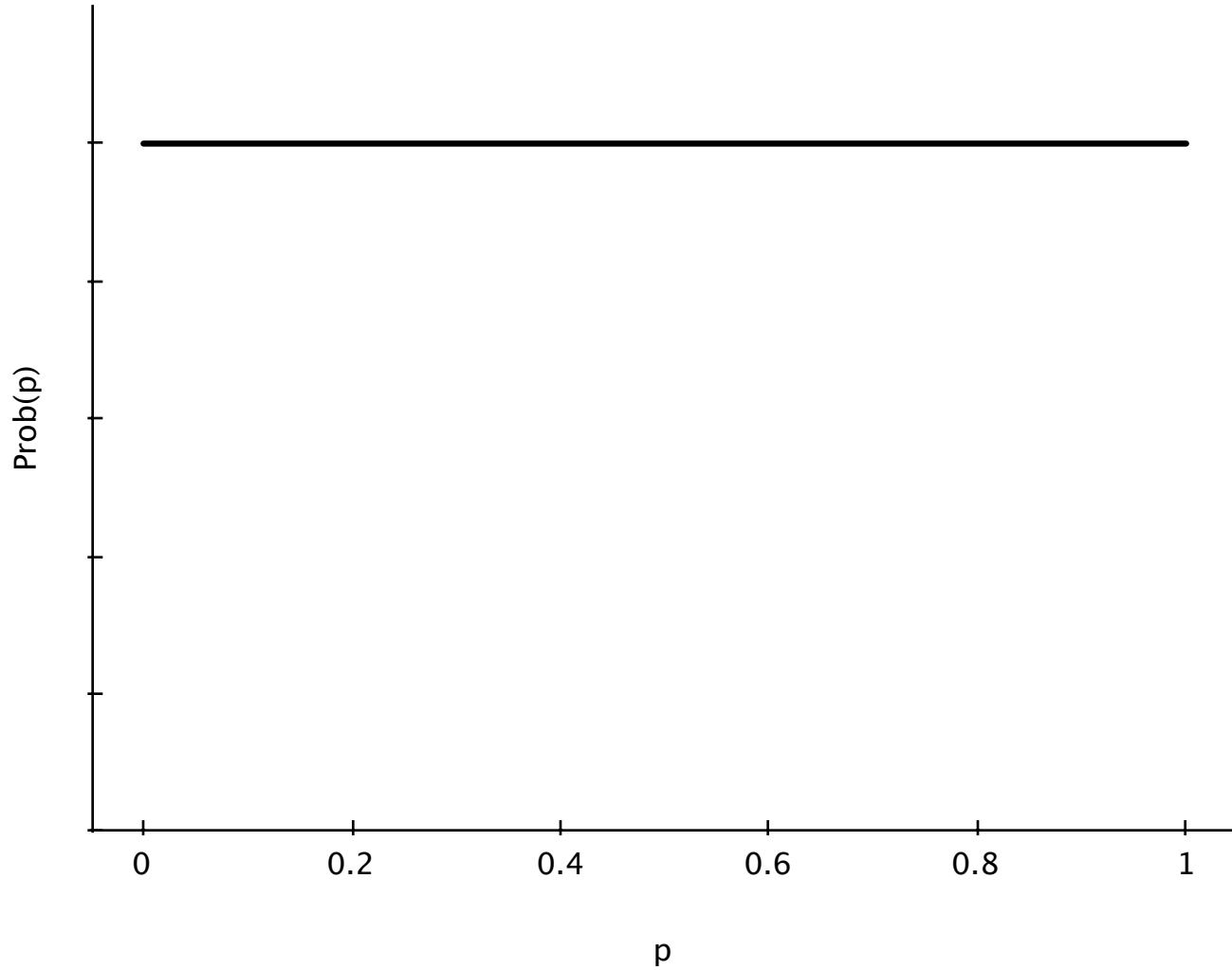
- You see a coin
 - What is the probability of heads?
 - Could it be larger or smaller than that?
- I flip the coin and while it is in the air ask again
- I catch the coin and ask again
- I *look* at the coin (and you don't) and ask again
- Why does the answer change?
 - And did it ever have a single value?



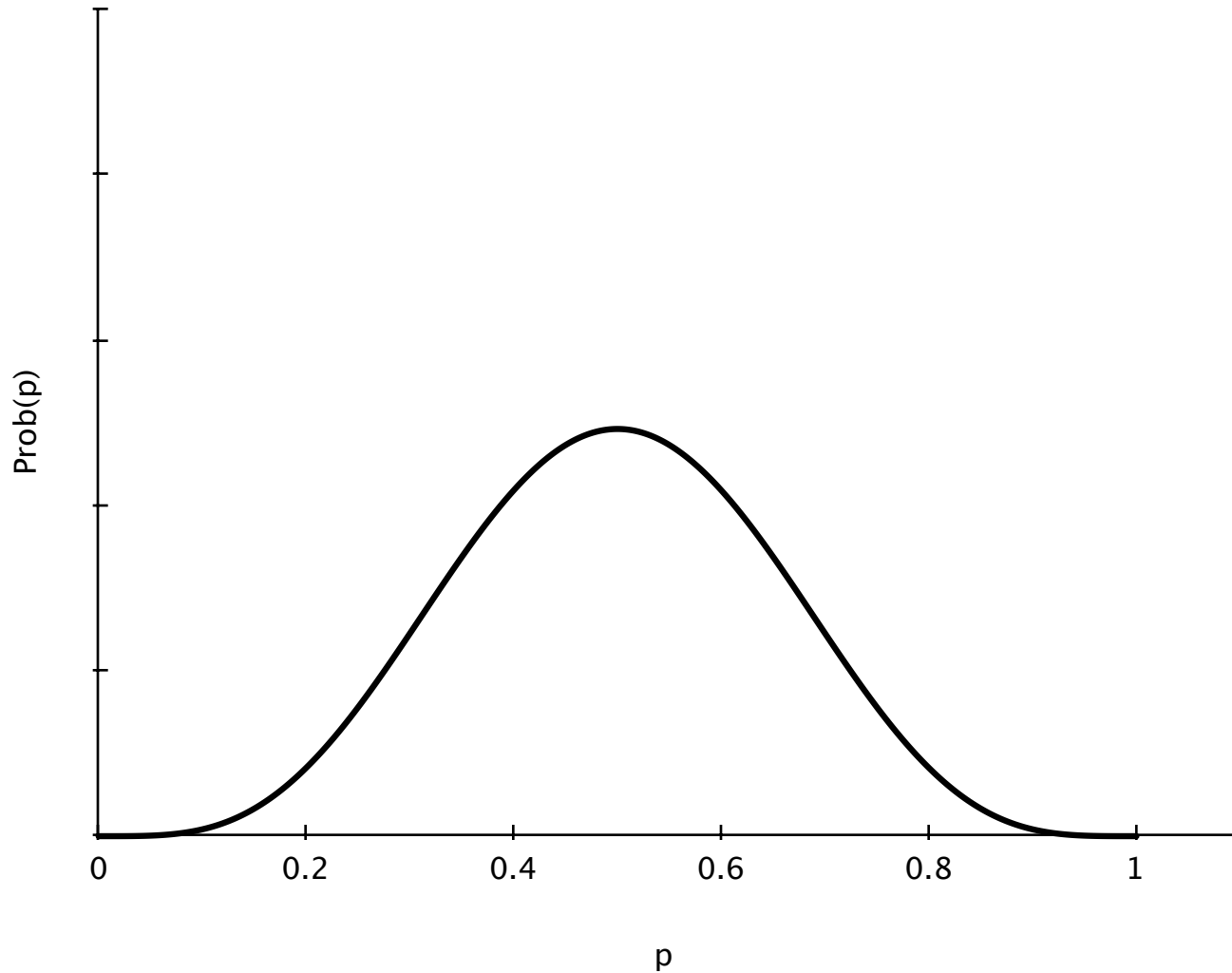
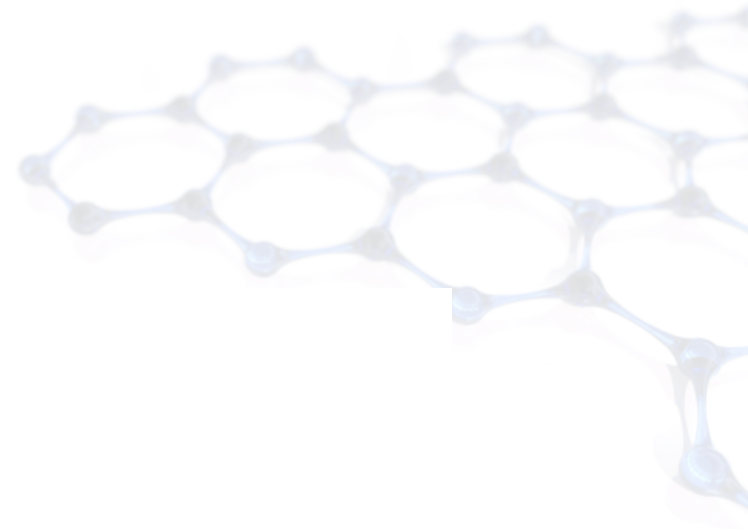
A Philosophical Conclusion

- Probability as expressed by humans is subjective and depends on information and experience

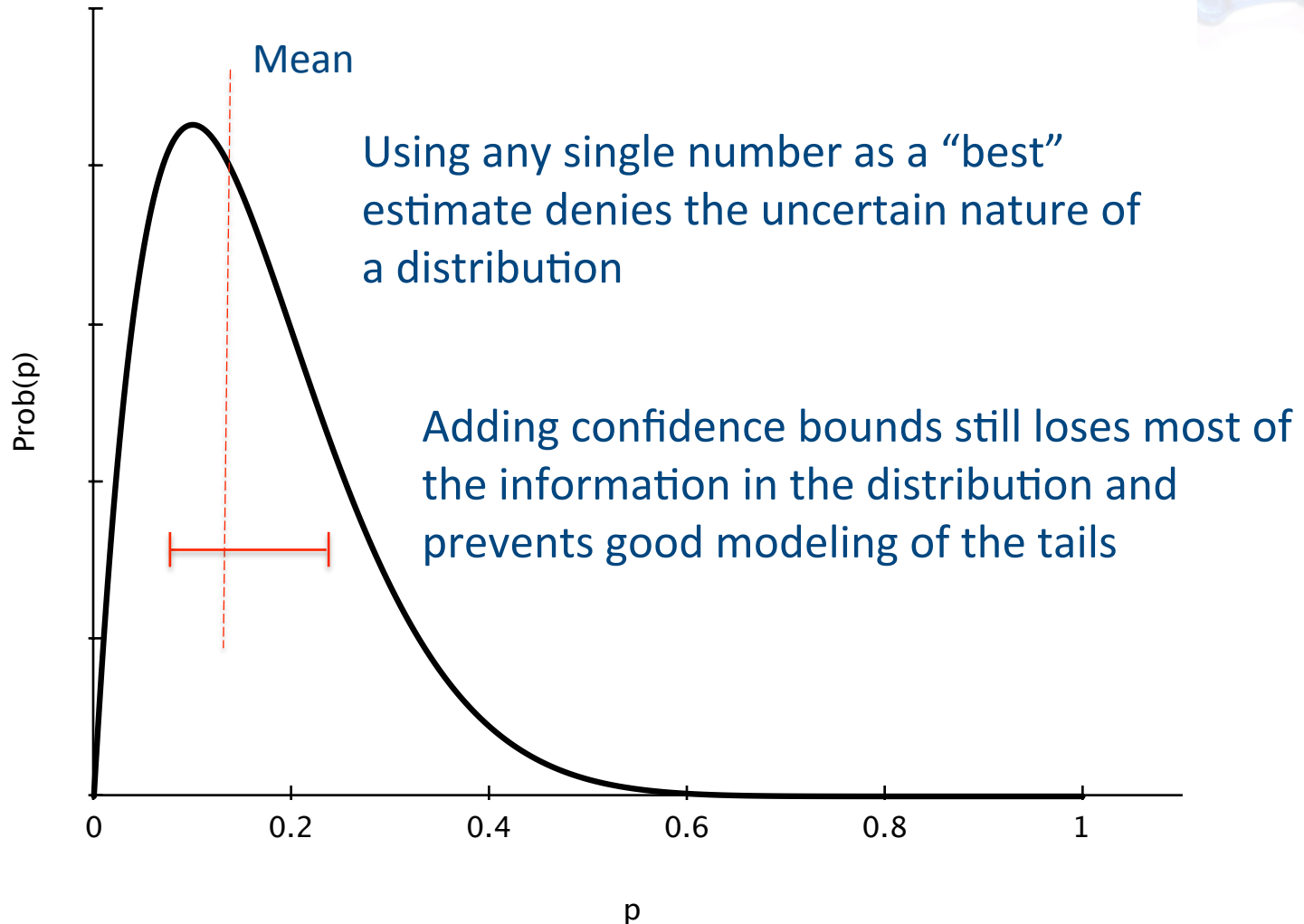
I Dunno



5 heads out of 10 throws



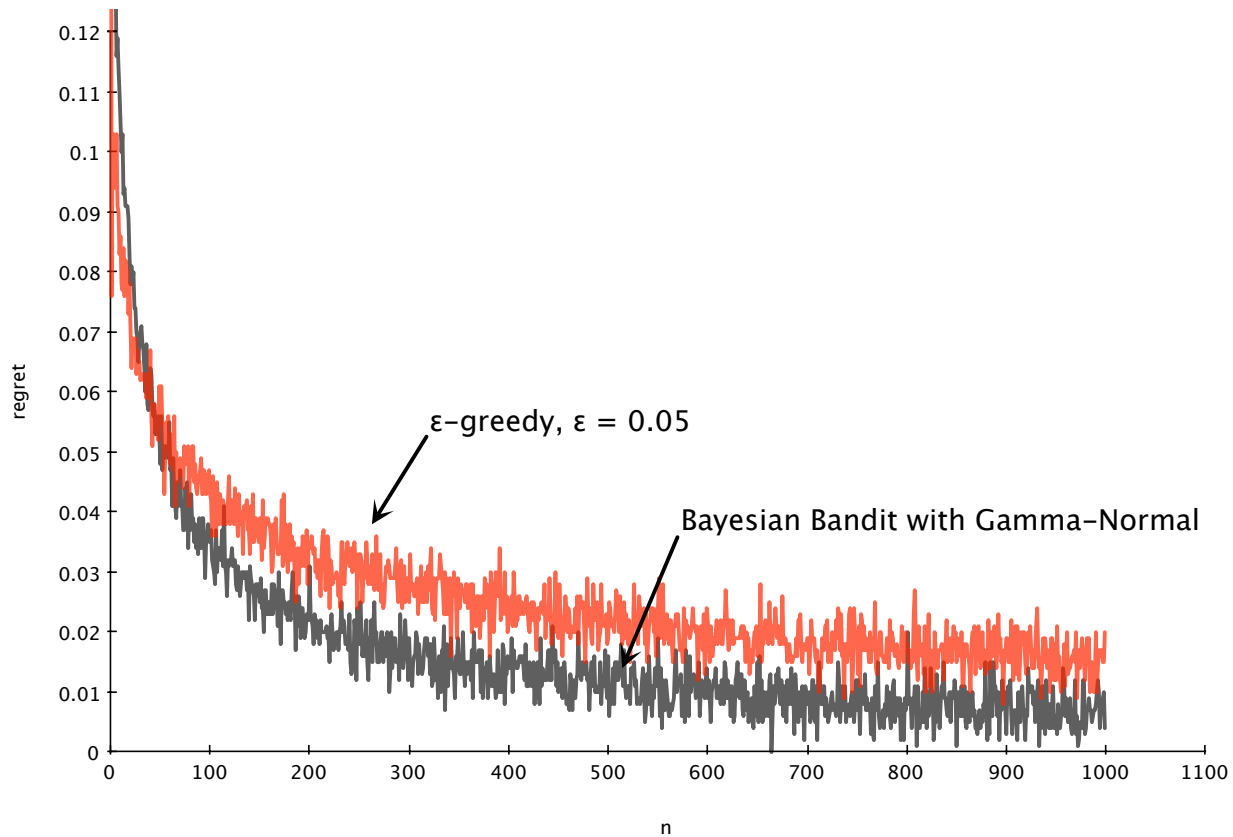
2 heads out of 12 throws



Bayesian Bandit

- Compute distributions based on data
- Sample p_1 and p_2 from these distributions
- Put a coin in bandit 1 if $p_1 > p_2$
- Else, put the coin in bandit 2

And it works!





Video Demo

The Code

- Select an alternative

```
n = dim(k)[1]
p0 = rep(0, length.out=n)
for (i in 1:n) {
  p0[i] = rbeta(1, k[i,2]+1, k[i,1]+1)
}
return (which(p0 == max(p0)))
```

- Select and learn

```
for (z in 1:steps) {
  i = select(k)
  j = test(i)
  k[i,j] = k[i,j]+1
}
return (k)
```

- But we already know how to *count!*

The Basic Idea

- We can encode a distribution by sampling
- Sampling allows unification of exploration and exploitation
- Can be extended to more general response models
- Note that learning here = counting = on-line algorithm

Generalized Banditry

- Suppose we have an infinite number of bandits
 - suppose they are each labeled by two real numbers x and y in $[0,1]$
 - also that expected payoff is a parameterized function of x and y

$$E[z] = f(x, y | \theta)$$

- now assume a distribution for θ that we can learn online
- Selection works by sampling θ , then computing f
- Learning works by propagating updates back to θ
 - If f is linear, this is very easy
- Don't just have to have two labels, could have labels and context

Caveats

- Original Bayesian Bandit only requires real-time
- Generalized Bandit may require access to long history for learning
 - Pseudo online learning may be easier than true online
- Bandit variables can include content, time of day, day of week
- Context variables can include user id, user features
- Bandit × context variables provide the real power

- 
- **Contact:**
 - tdunning@maprtech.com
 - @ted_dunning

 - **Slides and such (available late tonight):**
 - <http://slideshare.net/tdunning>

 - **Hash tags: #mapr #storm #bbuzz**

Thank You

