

Improvements for large state in Apache Flink	2
Introduction to Online Machine Learning Algorithms	42
Machine Learning on Flink	97
Make the cloud work for you@Google	139
Near Real-Time Analytics@Uber	182
No shard left behind	222
Portable stateful big data processing in Apache Beam	251
Queryable State	286
Redesigning Apache Flink's Distributed Architecture	317
Runtime Improvements in Blink for Large Scale Streaming at Alibaba	344
Storage Reimagined for a Streaming World	372
Table & SQL API- unified APIs for batch and stream processing	427
The Stream Processor as a Database	461
Zero to Streaming	491
Apache Flink-The Latest and Greatest@dataArtisans	525
AthenaX-Streaming Processing Platform @Uber	550
Blink's Improvements to Flink SQL &Table API @alibaba	579
Building a Real-Time Anomaly-Detection System with Flink @ Mux	608
Comments on Streaming Deep Learning with Flink	630
Contributing to Apache Flink®	676
Dynamically Configured Stream Processing Using Flink & Kafka	705
Experiences in running Apache Flink® at large scale	754
EXPERIENCES WITH STREAMING & MICRO-BATCH FOR ONLINE LEARNING	798
Extending Flink's Streaming APIs	811
Flink, Queryable State, and High Frequency Time Series Data	862

Improvements for large state in Apache Flink



Stefan Richter
@stefanrrichter

dataArtisans

April 11, 2017



State in Streaming Programs

```
case class Event(producer: String, evtType: Int, msg: String)
case class Alert(msg: String, count: Long)
```

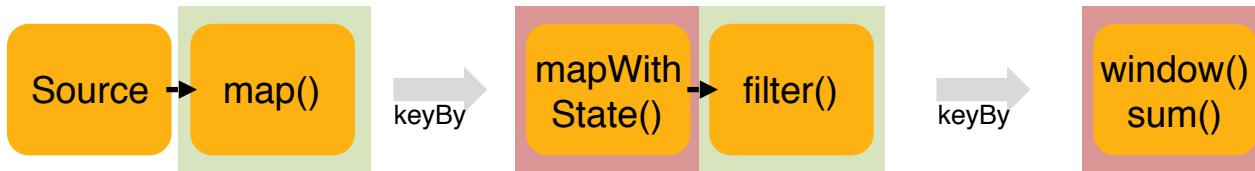


```
env.addSource(...)  
  .map(bytes => Event.parse(bytes) )  
  .keyBy("producer")  
  .mapWithState { (event: Event, state: Option[Int]) => {  
    // pattern rules  
  }  
  .filter(alert => alert.msg.contains("CRITICAL"))  
  .keyBy("msg")  
  .timeWindow(Time.seconds(10))  
  .sum("count")
```



State in Streaming Programs

```
case class Event(producer: String, evtType: Int, msg: String)  
case class Alert(msg: String, count: Long)
```



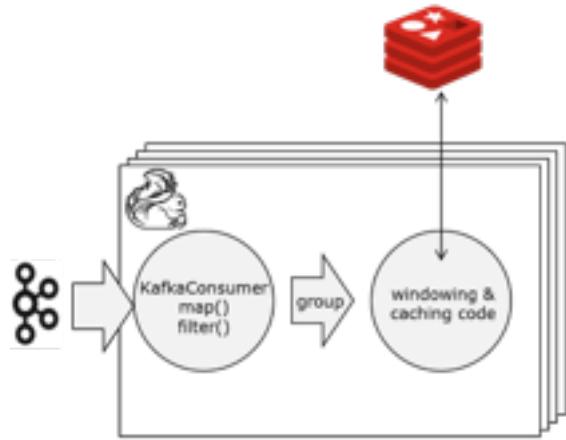
```
env.addSource(...)
```

```
.map(bytes => Event.parse(bytes) )  
.keyBy("producer")  
.mapWithState { (event: Event, state: Option[Int]) =>  
    // pattern rules  
}  
.filter(alert => alert.msg.contains("CRITICAL"))  
.keyBy("msg")  
.timeWindow(Time.seconds(10))  
.sum("count")
```

Stateless

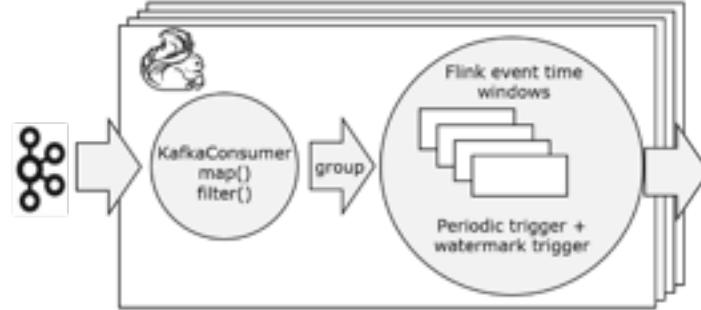
Stateful

Internal vs External State



External State

- State in a separate data store
- Can store "state capacity" independent
- Usually much slower than internal state
- Hard to get "exactly-once" guarantees

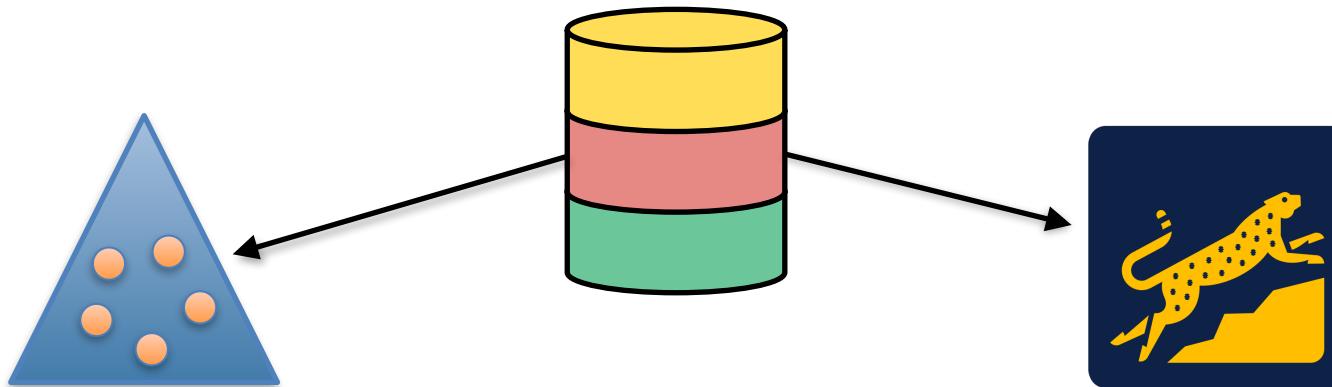


Internal State

- State in the stream processor
- Faster than external state
- Working area local to computation
- Checkpoints to stable store (DFS)
- Always exactly-once consistent
- Stream processor has to handle scalability



Keyed State Backends



HeapKeyedStateBackend

- State lives in memory, on Java heap
- Operates on objects
- Think of a hash map {key obj -> state obj}
- Async snapshots supported**

RocksDBKeyedStateBackend

- State lives in off-heap memory and on disk
- Operates on bytes, uses serialization
- Think of K/V store {key bytes -> state bytes}
- Log-structured-merge (LSM) tree
- Async snapshots
- Incremental snapshots**

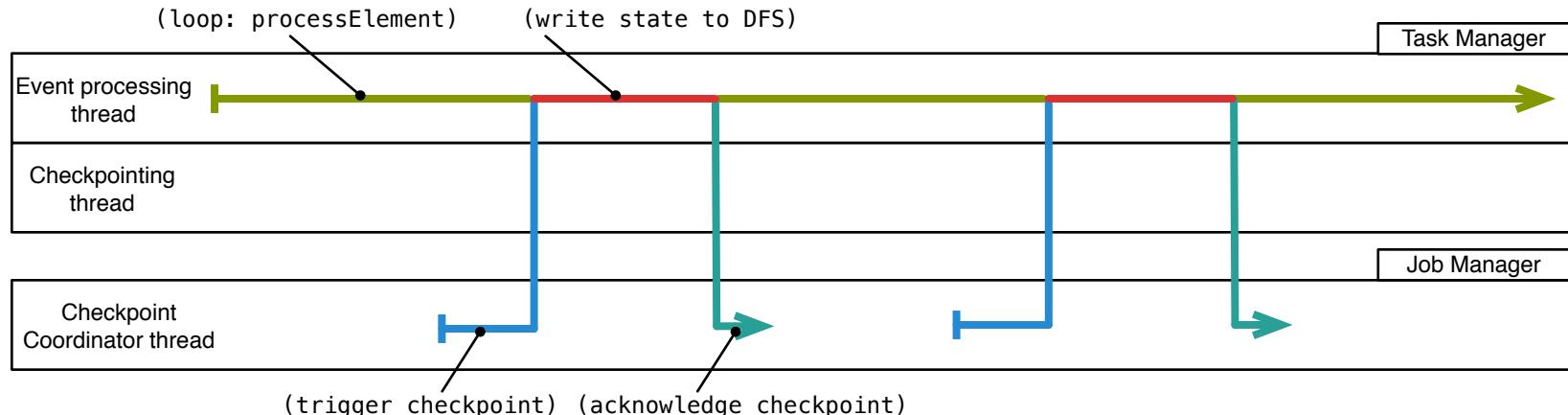


Asynchronous Checkpoints



Synchronous Checkpointing

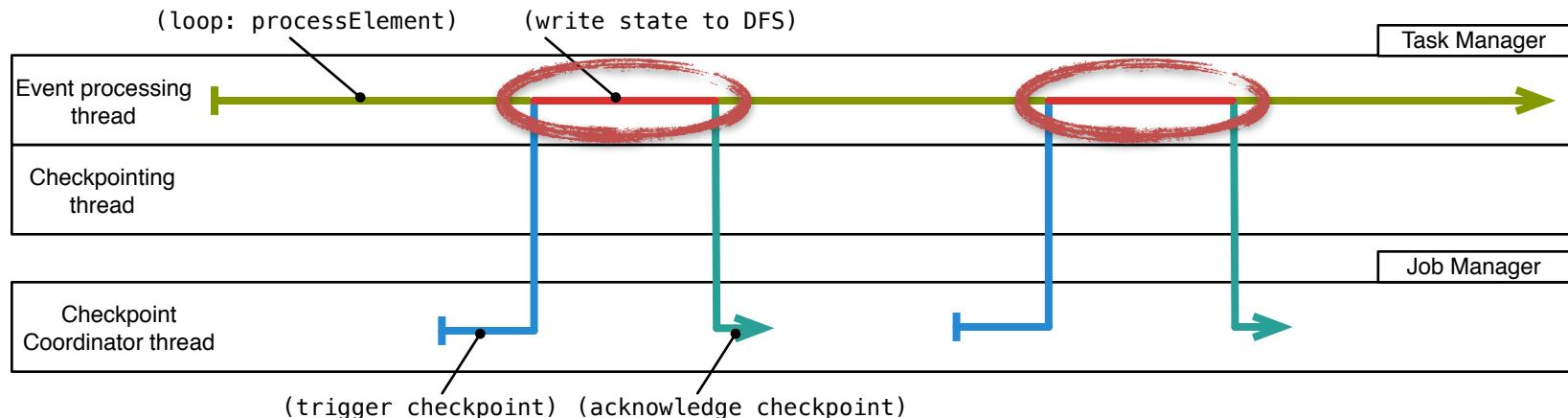
Why is async checkpointing so essential for large state?





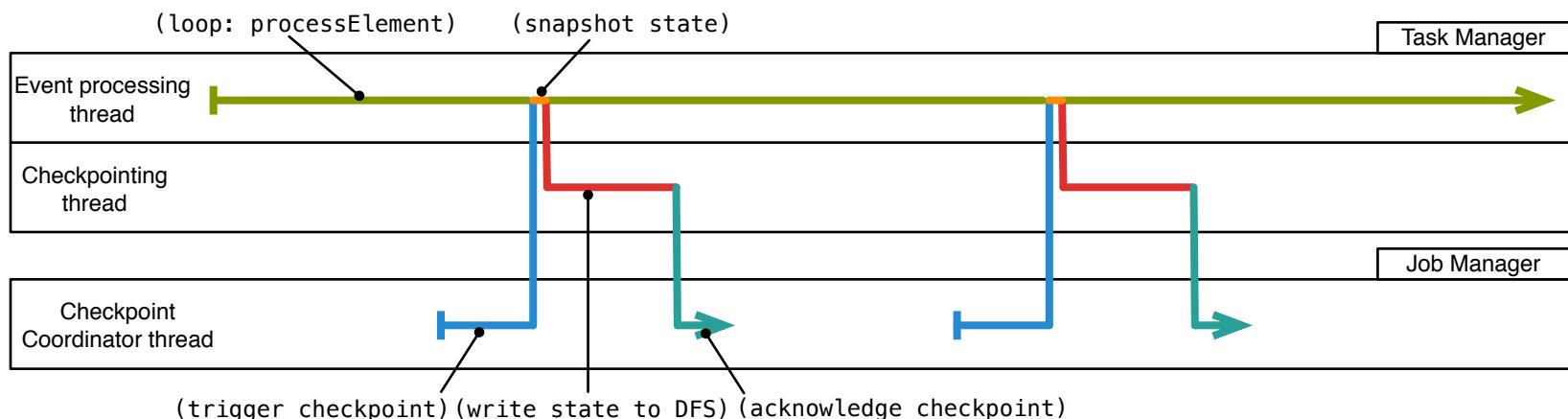
Synchronous Checkpointing

Problem: All event processing is on hold here to avoid concurrent modifications to the state that is written





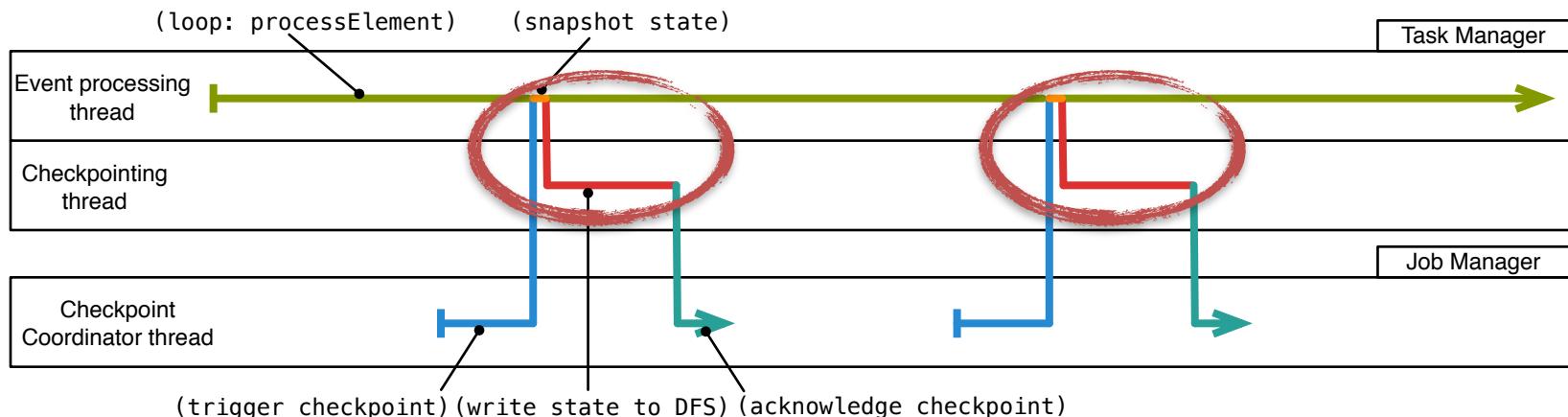
Asynchronous Checkpointing





Asynchronous Checkpointing

Problem: How to deal with concurrent modifications?





Incremental Checkpoints



What we will discuss

- What are incremental checkpoints?
- Why is RocksDB so well suited for this?
- How do we integrate this with Flink's checkpointing?

Driven by





Full Checkpointing

time

K	S
2	B
4	W
6	N

K	S
2	B
3	K
4	L
6	N

K	S
2	Q
3	K
6	N
9	S

K	S
2	B
4	W
6	N

K	S
2	B
3	K
4	L
6	N

K	S
2	Q
3	K
6	N
9	S

Checkpoint 1

Checkpoint 2

Checkpoint 3



Incremental Checkpointing

time

K	S
2	B
4	W
6	N

K	S
2	B
3	K
4	L
6	N

K	S
2	Q
3	K
6	N
9	S

K	S
2	B
4	W
6	N

$\Delta(-, c1)$

iCheckpoint 1

K	S
3	K
4	L

$\Delta(c1, c2)$

iCheckpoint 2

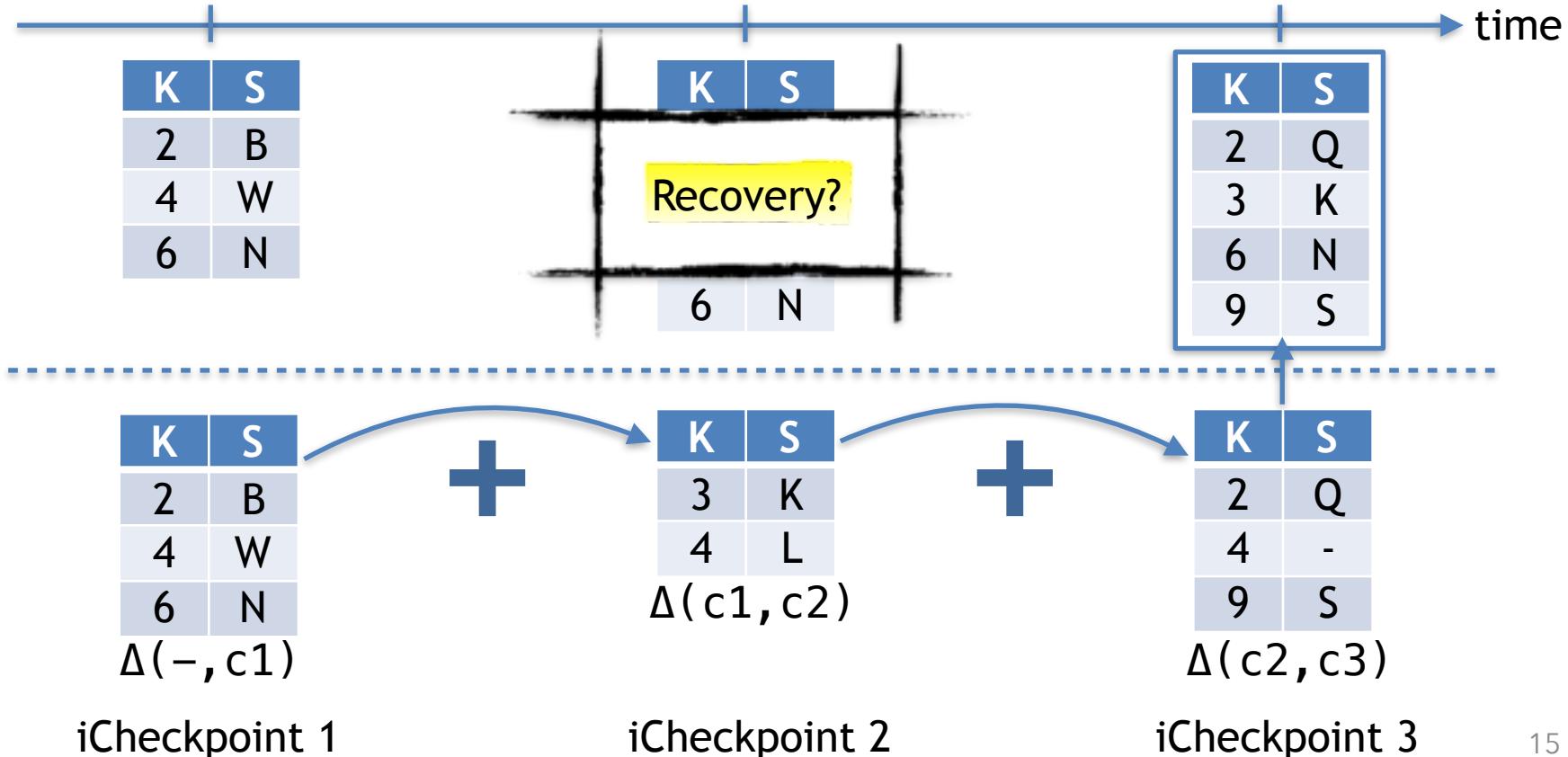
K	S
2	Q
4	-
9	S

$\Delta(c2, c3)$

iCheckpoint 3



Incremental Recovery

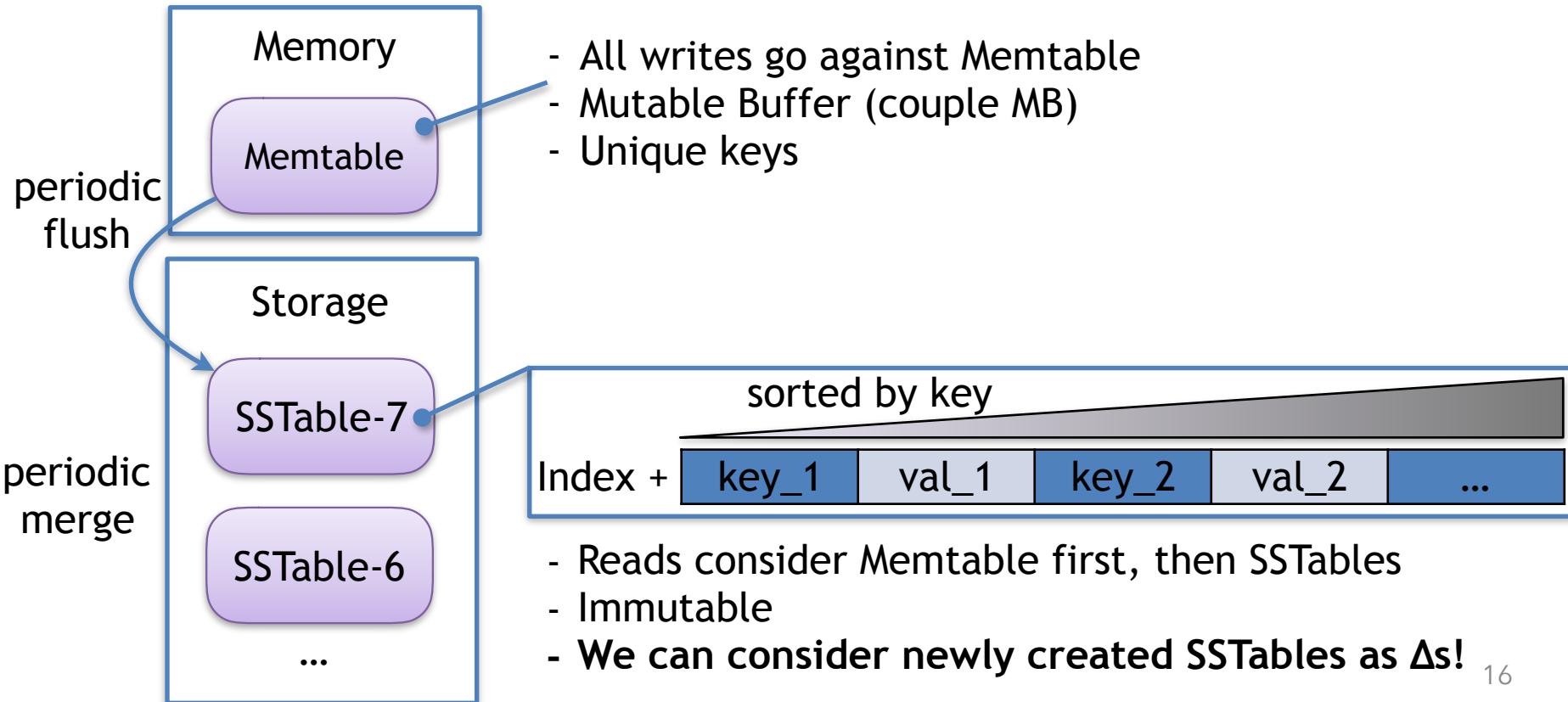


iCheckpoint 1

iCheckpoint 2

iCheckpoint 3

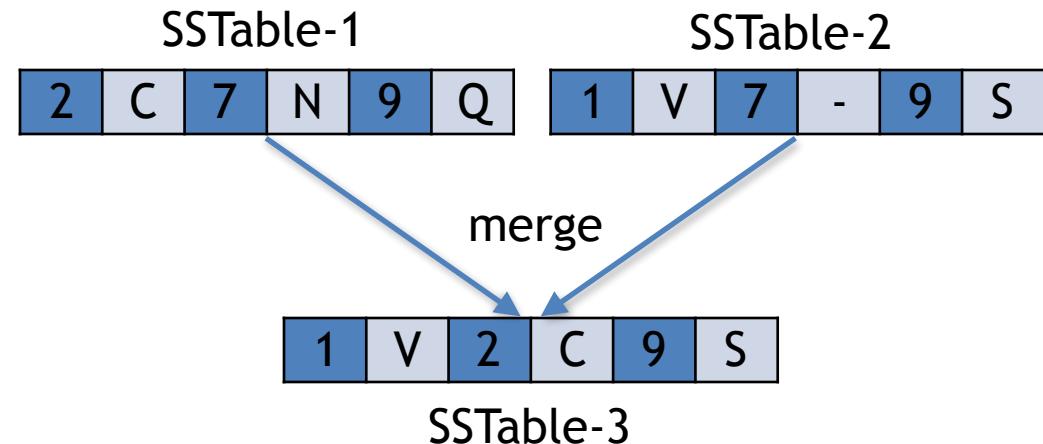
RocksDB Architecture (simplified)





RocksDB Compaction

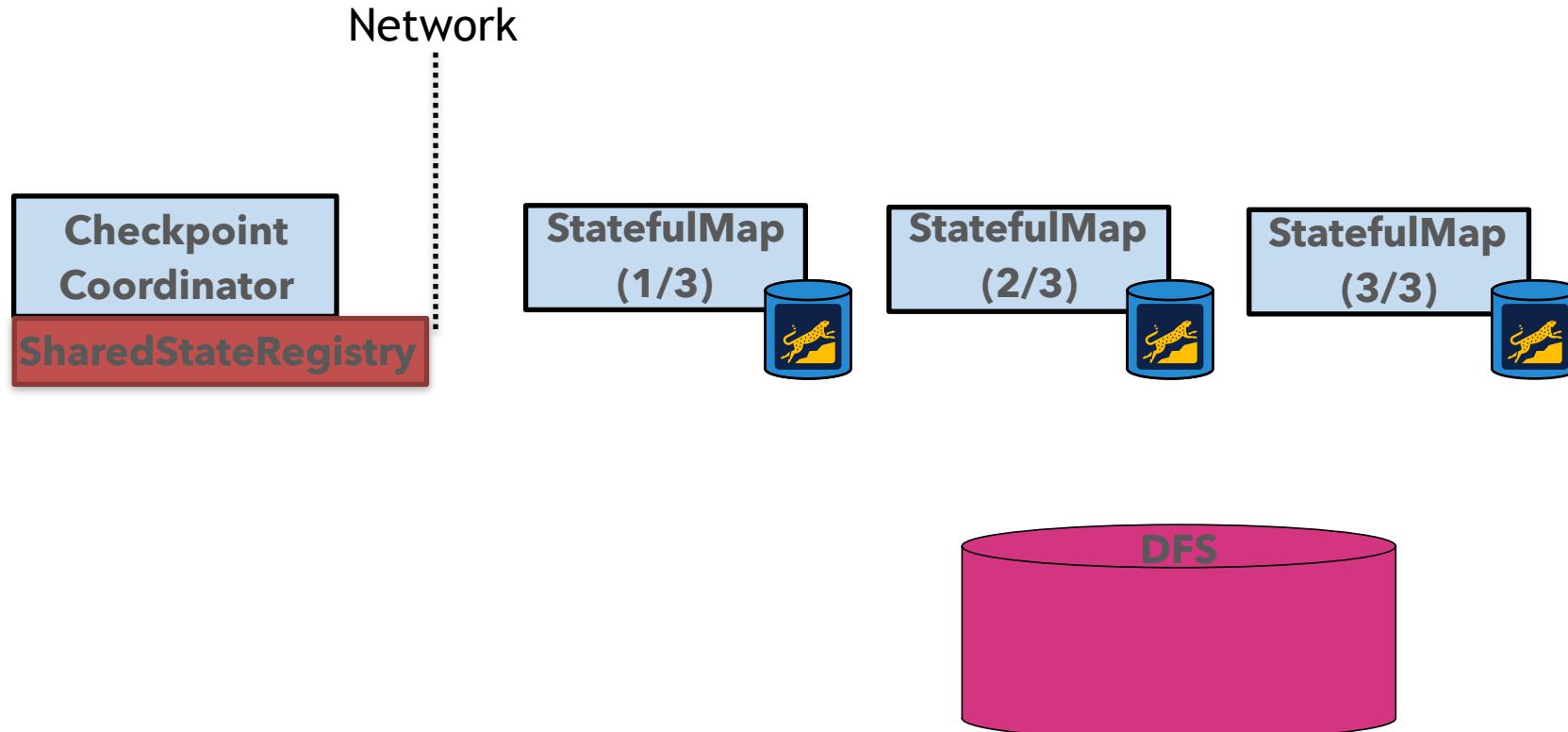
- Background Thread merges SSTable files
- Removes copies of the same key (latest version survives)
- Actually deletion of keys



Compaction consolidates our Δ s!

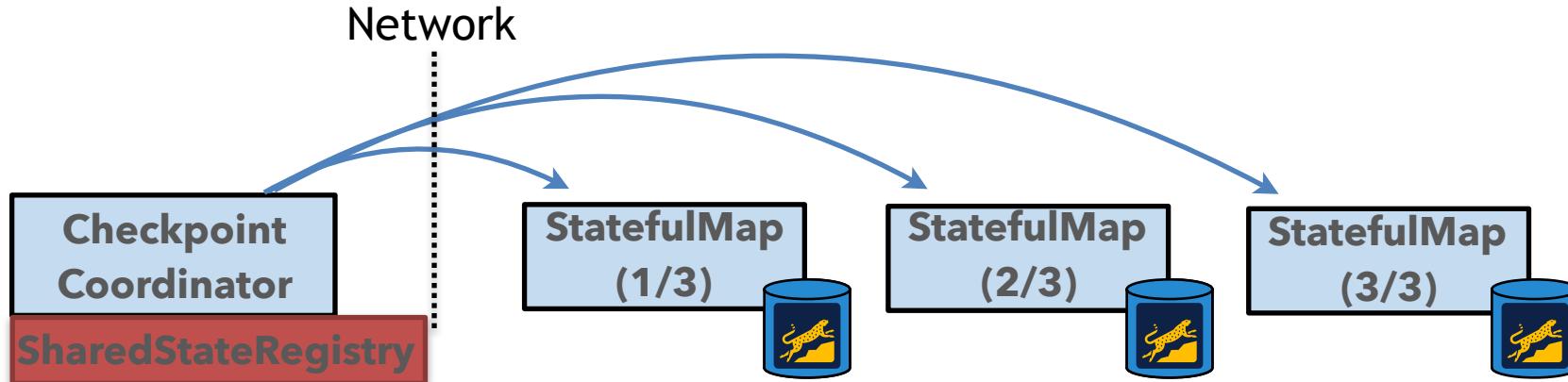


Flink's Incremental Checkpointing

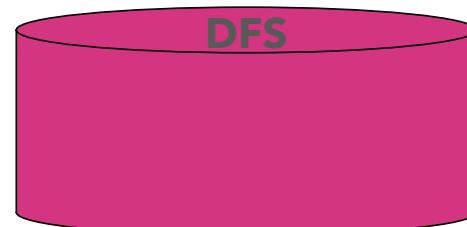




Flink's Incremental Checkpointing

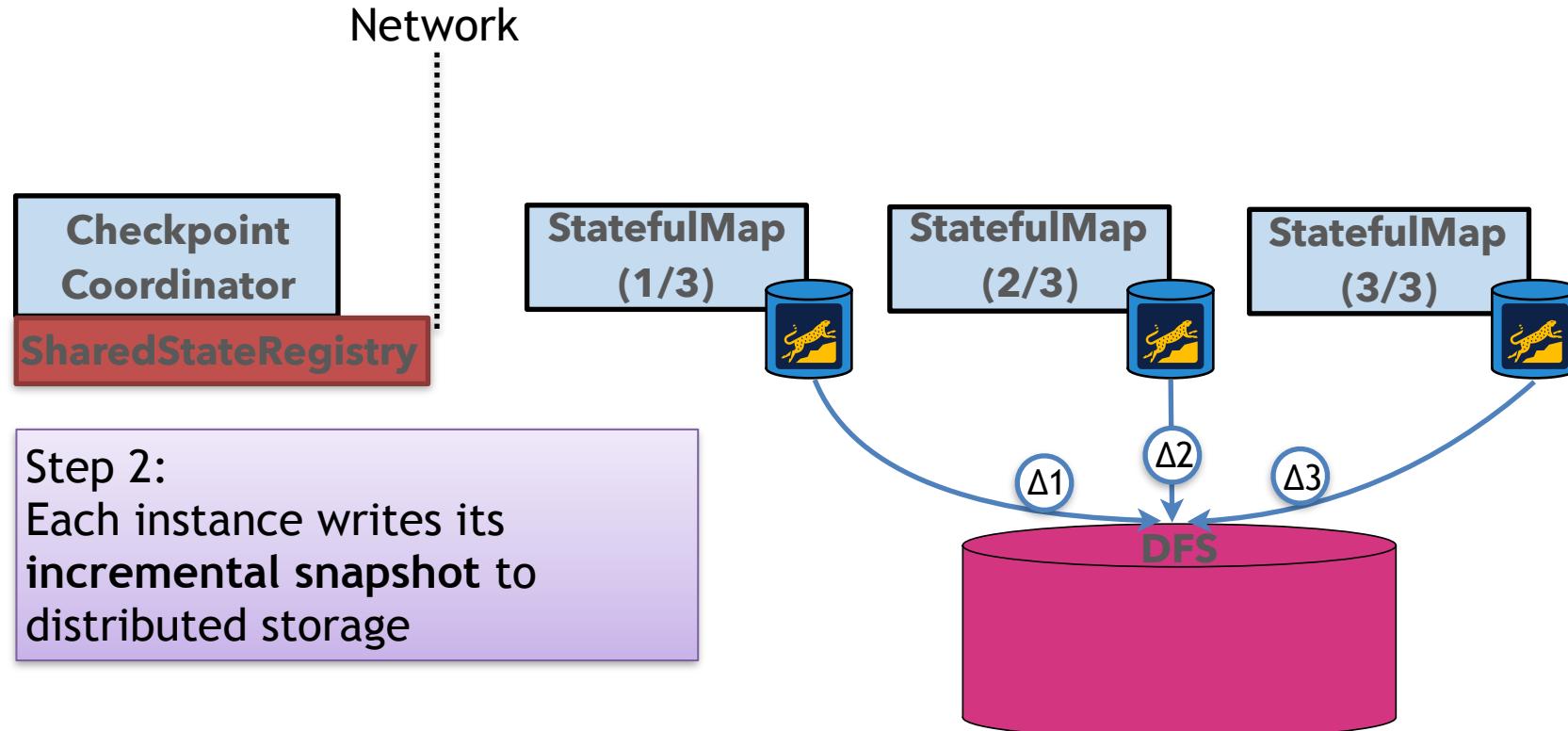


Step 1:
Checkpoint Coordinator sends
checkpoint barrier that triggers
a snapshot on each instance



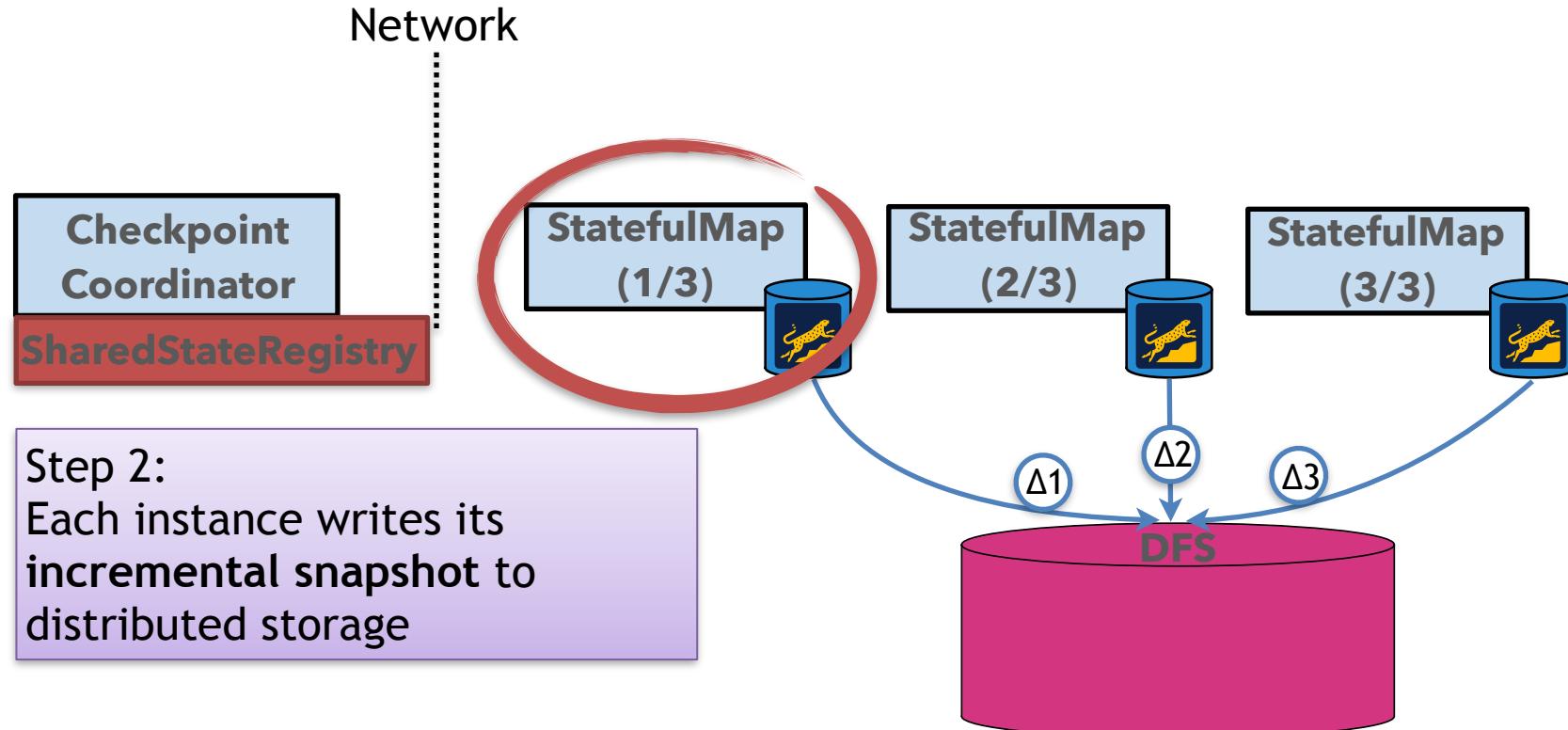


Flink's Incremental Checkpointing



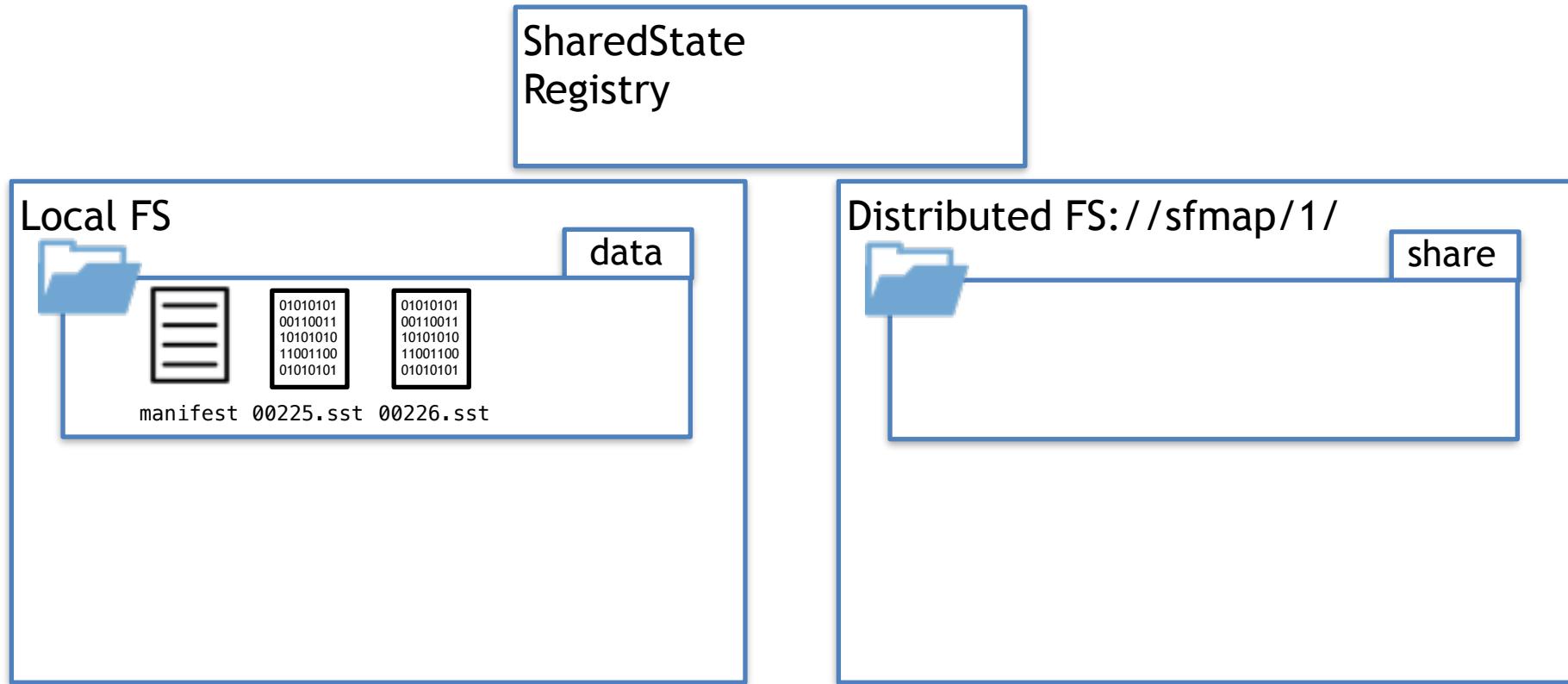


Flink's Incremental Checkpointing



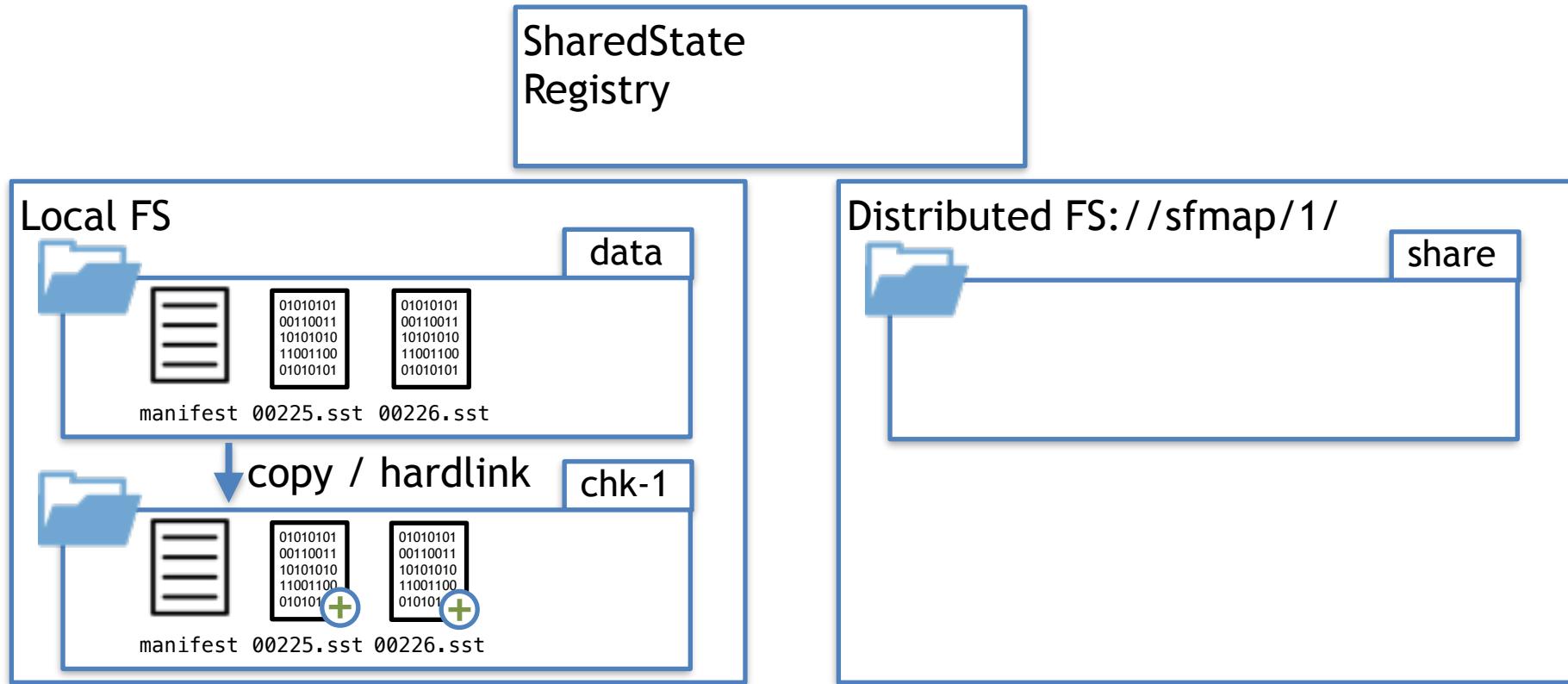


Incremental Snapshot of Operator



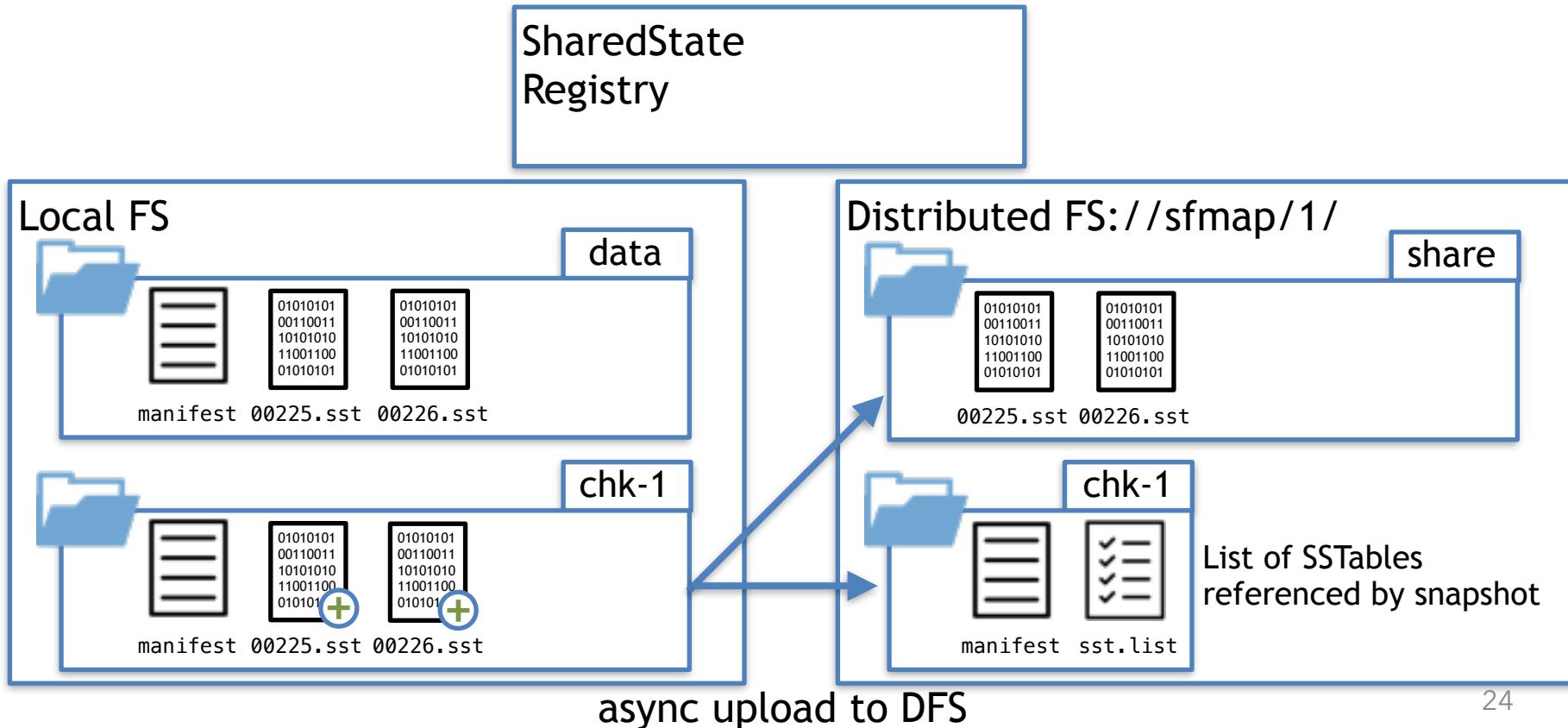


Incremental Snapshot of Operator



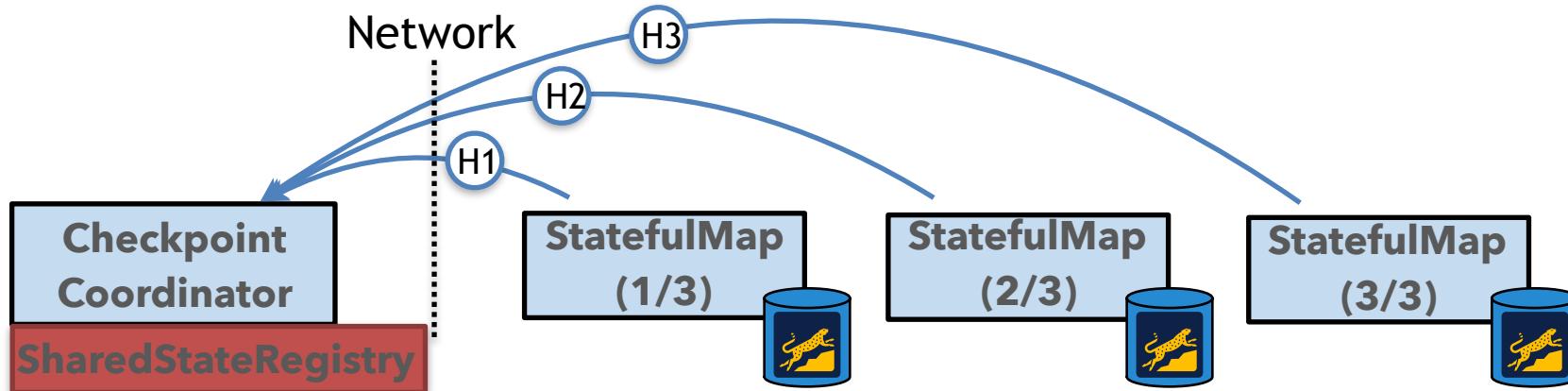


Incremental Snapshot of Operator

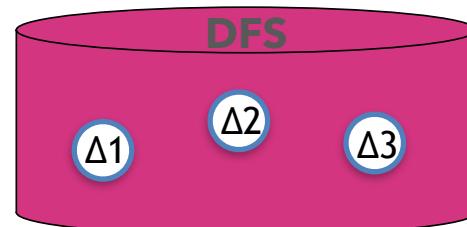




Flink's Incremental Checkpointing

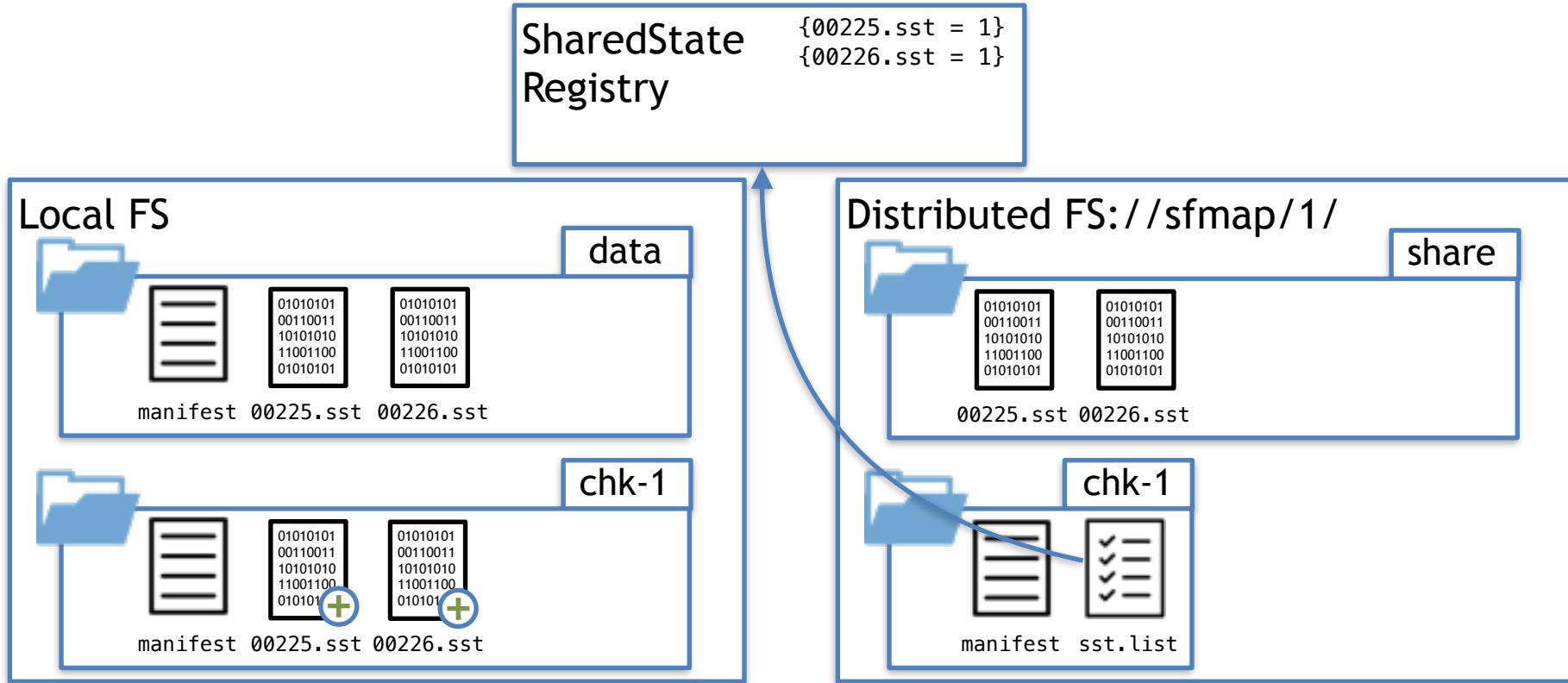


Step 3:
Each instance acknowledges and sends a handle (e.g. file path in DFS) to the Checkpoint Coordinator.



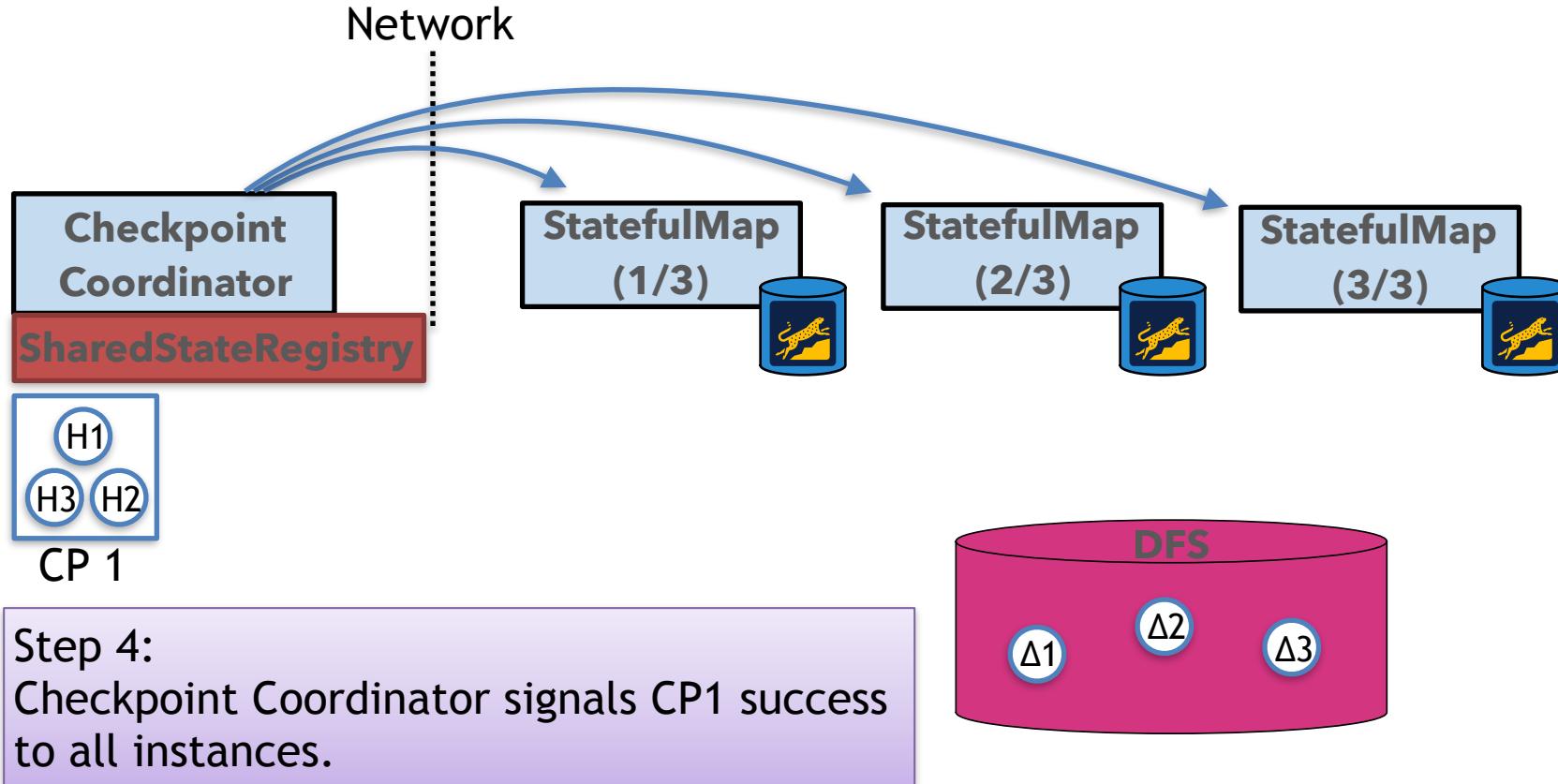


Incremental Snapshot of Operator





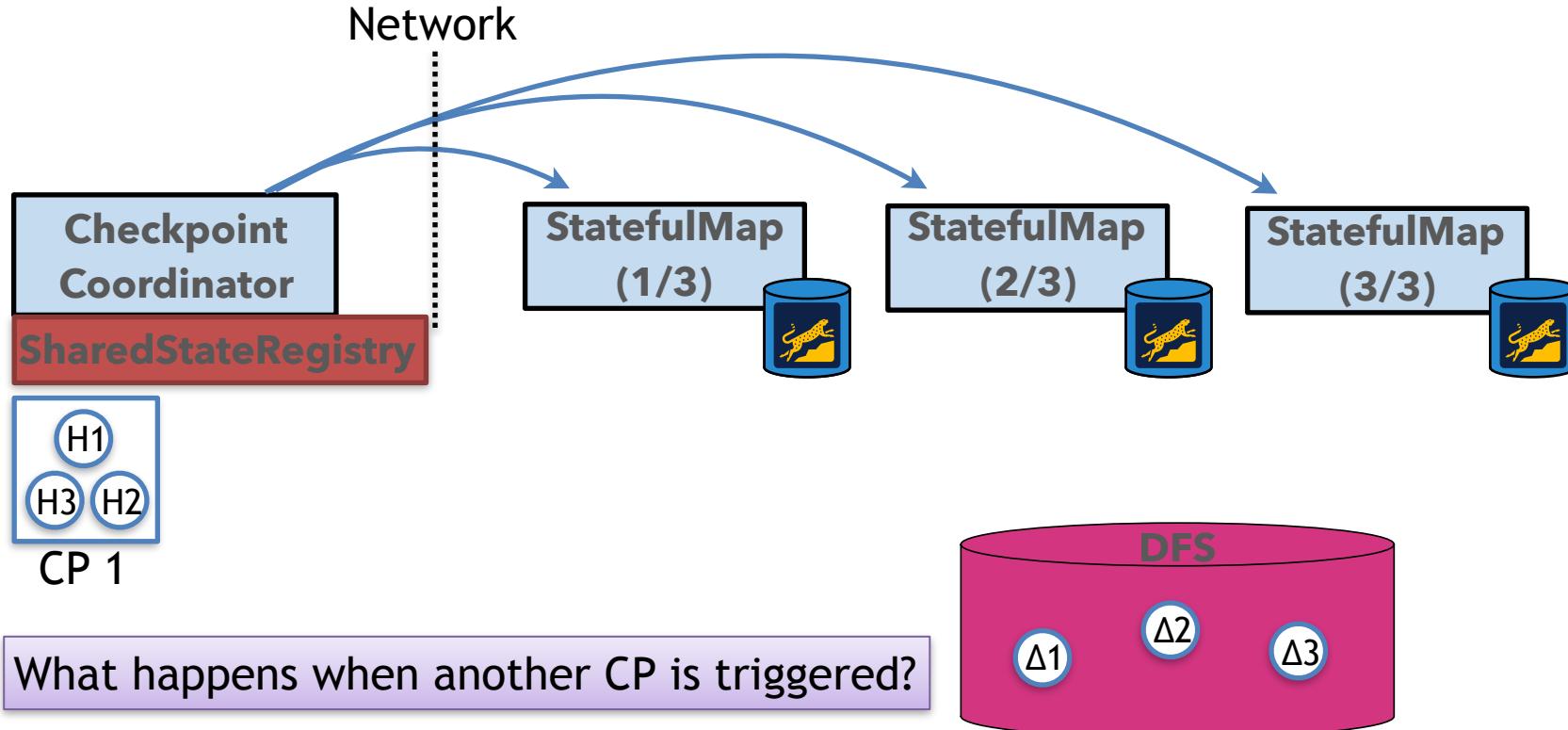
Flink's Incremental Checkpointing



Step 4:
Checkpoint Coordinator signals CP1 success
to all instances.

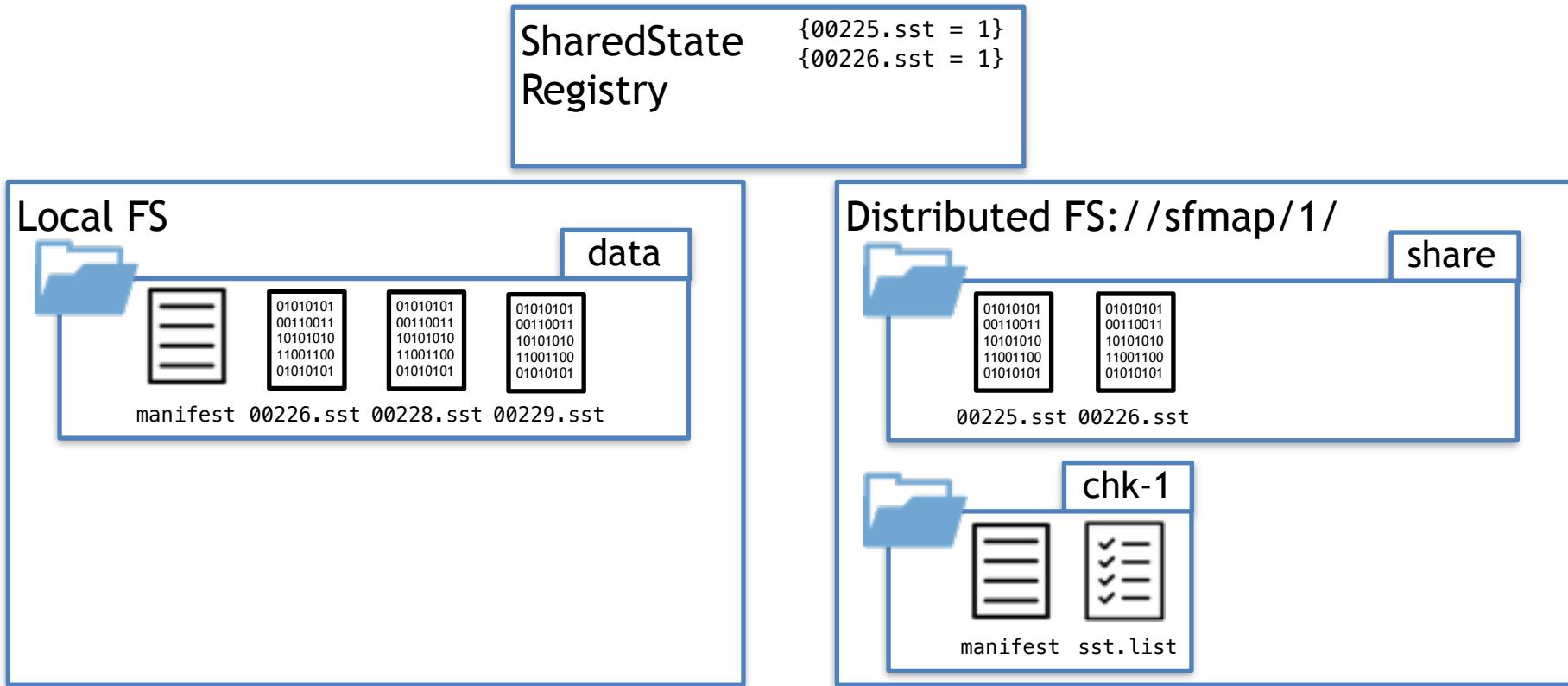


Flink's Incremental Checkpointing



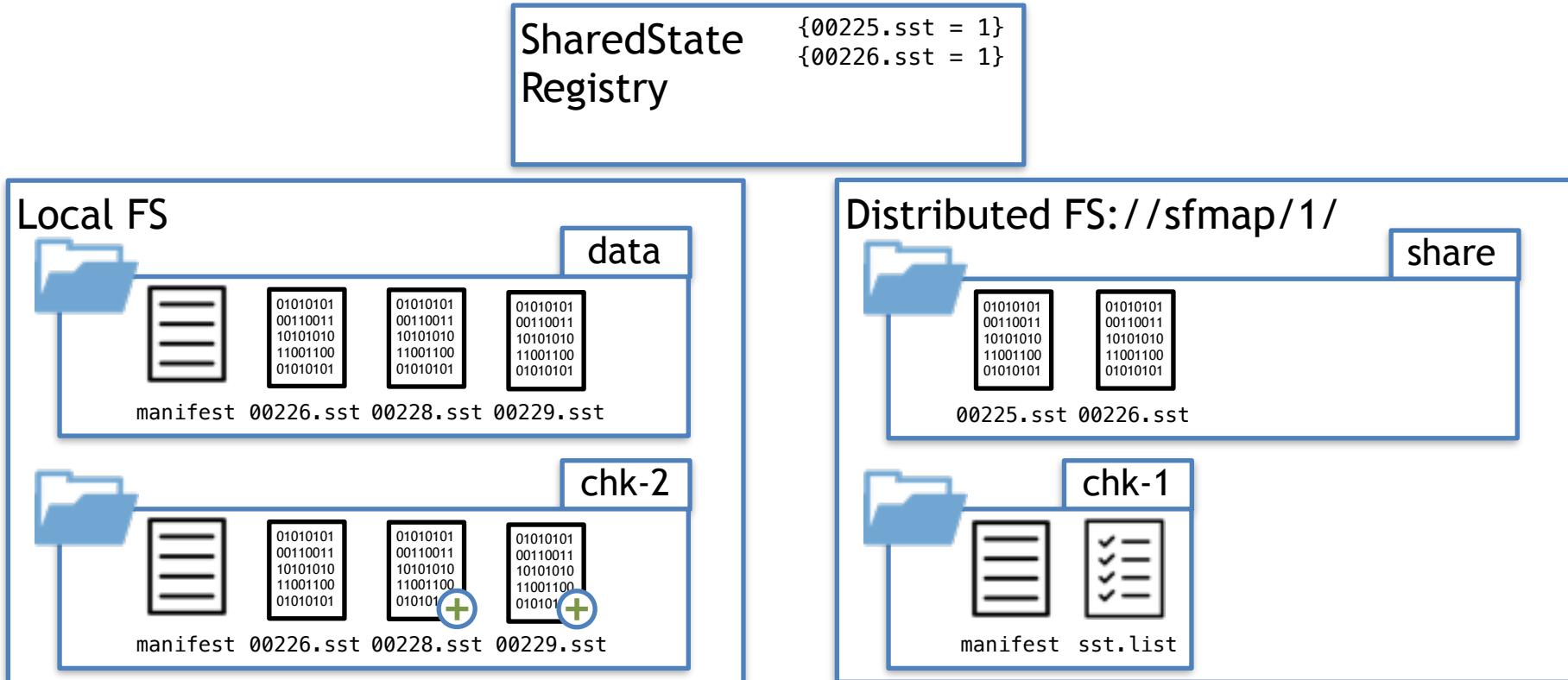


Incremental Snapshot of Operator



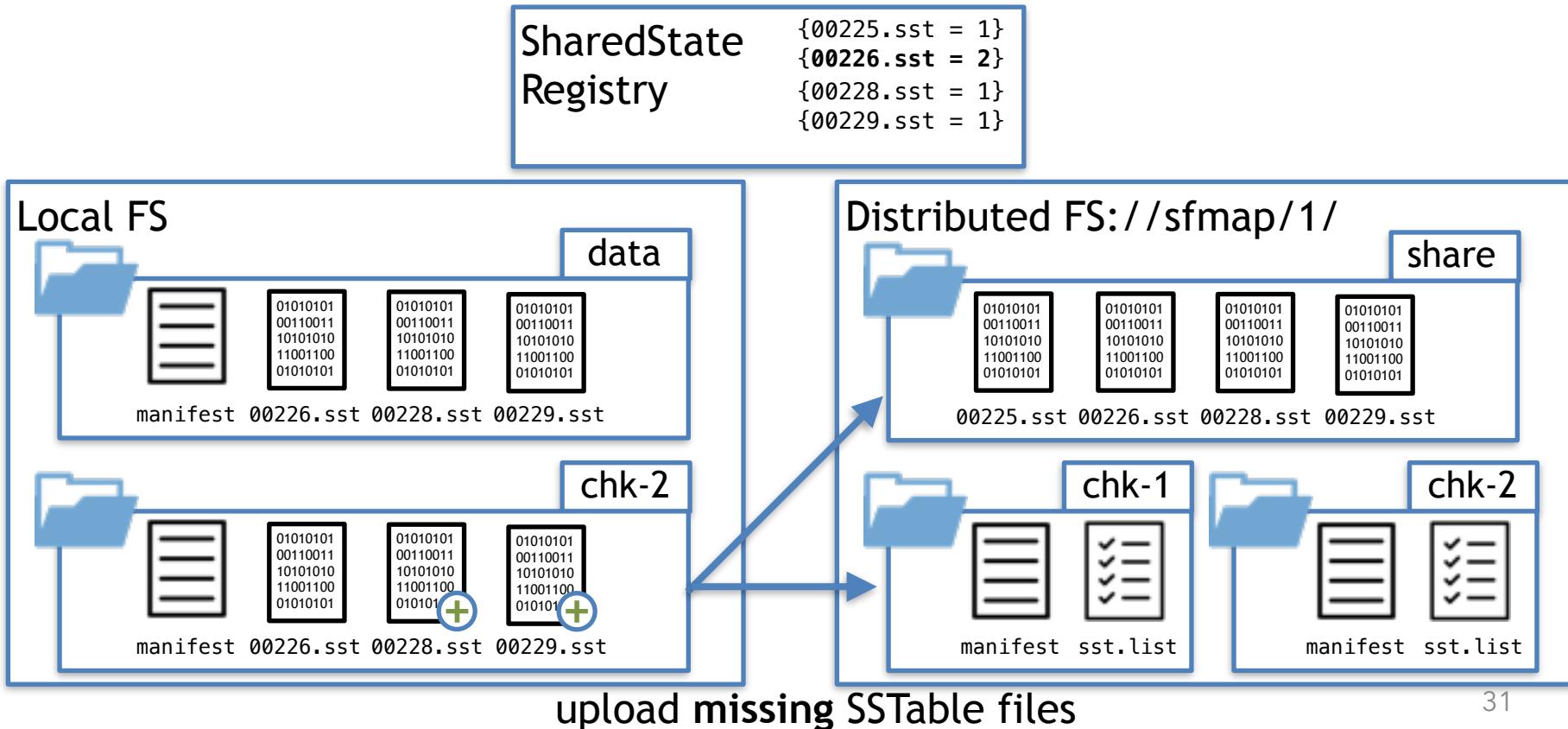


Incremental Snapshot of Operator



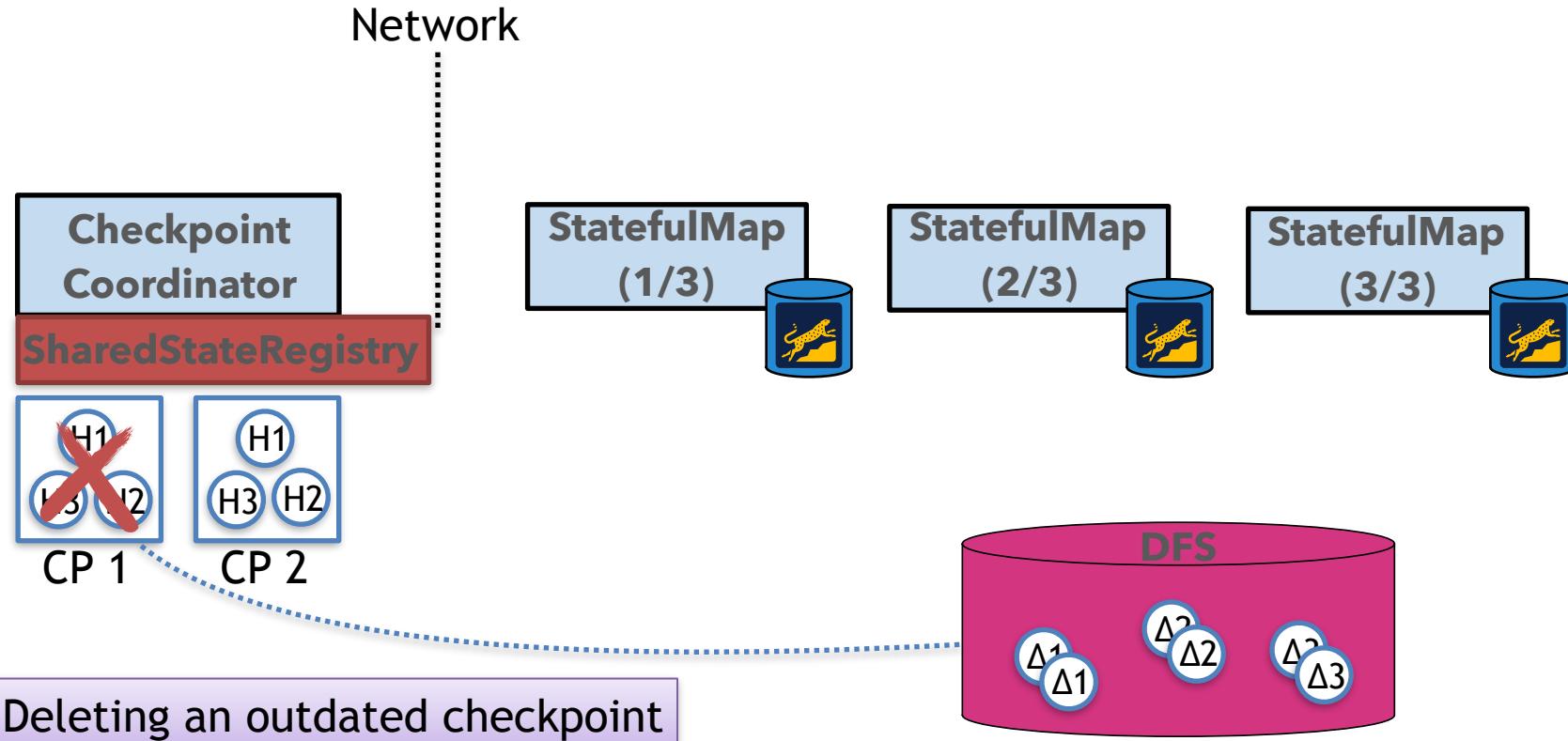


Incremental Snapshot of Operator



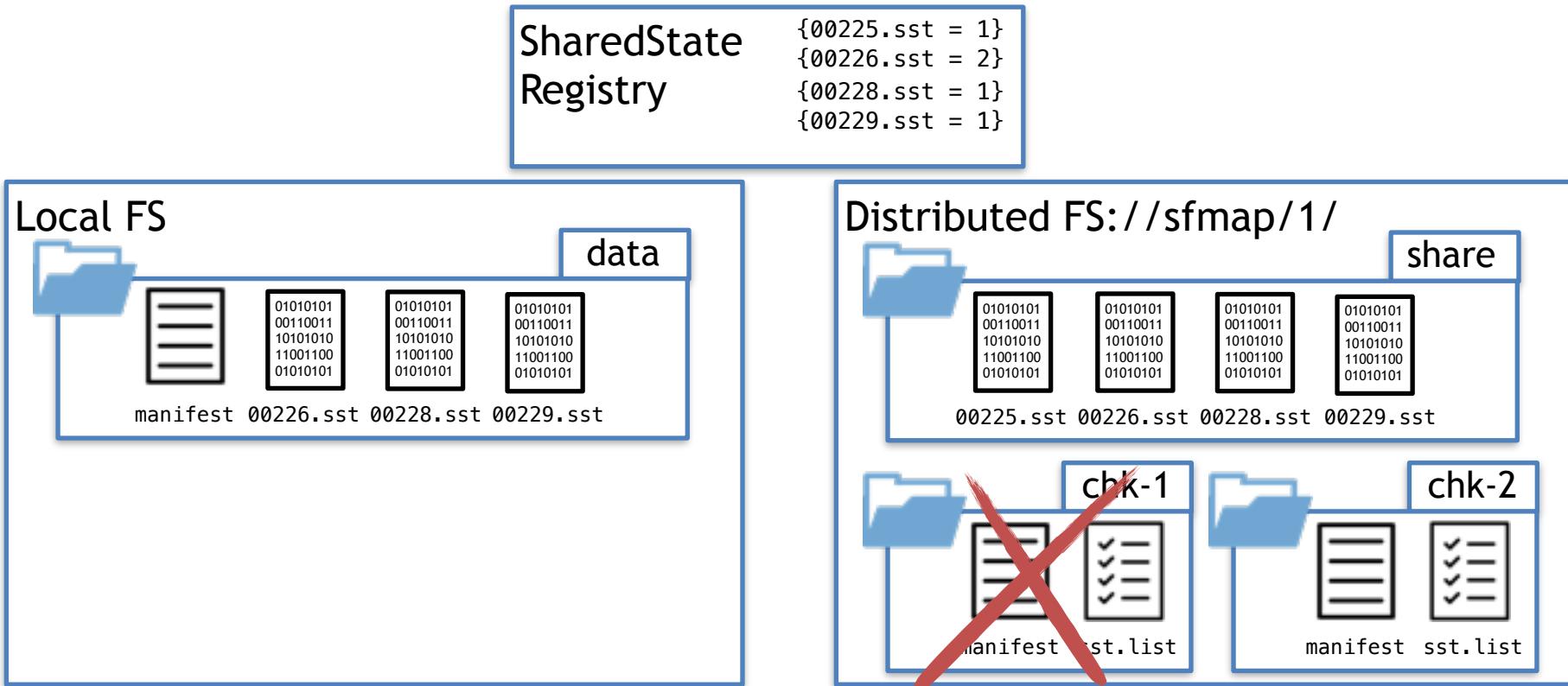


Deleting Incremental Checkpoints



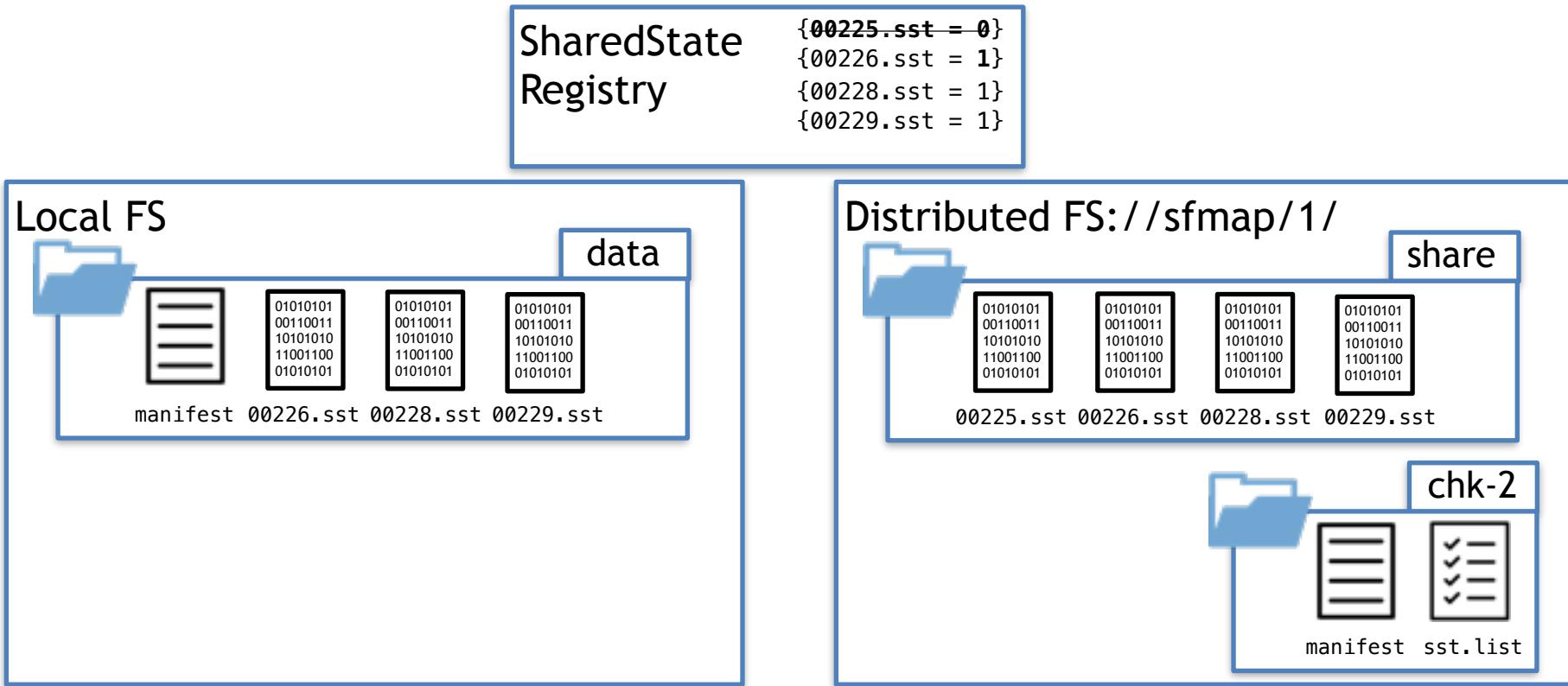


Deleting Incremental Snapshot



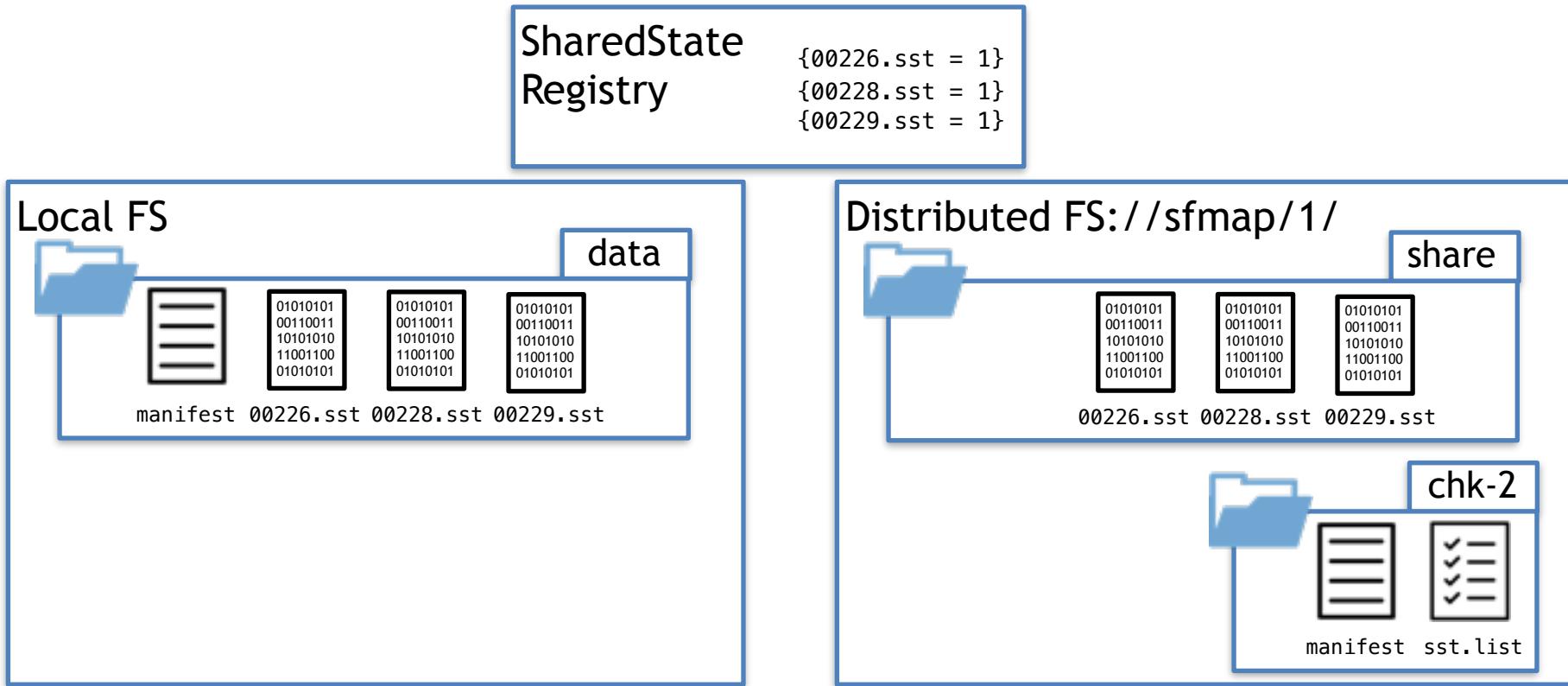


Deleting Incremental Snapshot





Deleting Incremental Snapshot





Wrapping up

Incremental checkpointing benefits



- Incremental checkpoints can dramatically reduce CP overhead for large state.
- Incremental checkpoints are async.
- RocksDB's compaction consolidates the increments. Keeps overhead low for recovery.

Incremental checkpointing limitations



- Breaks the unification of checkpoints and savepoints (CP: low overhead, SP: features)
- RocksDB specific format.
- Currently no support for rescaling from incremental checkpoint.

Further improvements in Flink 1.3/4



- *AsyncHeapKeyedStateBackend (merged)*
- *AsyncHeapOperatorStateBackend (PR)*
- *MapState (merged)*
- RocksDBInternalTimerService (PR)
- AsyncHeapInternalTimerService



Questions?



presentation starting soon... sit down

Introduction to Online Machine Learning Algorithms

Flink Forward- San Francisco

Trevor Grant
@rawkintrevo
April 11th, 2017

Table of Contents

Intro

- Who is this guy?
- Why should I care?
- What's going on here?
- This seems boring and mathy, maybe I should leave...

Buzzwords

Basic Online Learners

Challenges

Lambda Recommender

Conclusions



Branding

- Trevor Grant
- Things I do:
 - Open Source Technical Evangelist, IBM
 - PMC Apache Mahout
 - Blog: <http://rawkintrevo.org>
- Schooling
 - MS Applied Math, Illinois State
 - MBA, Illinois State
- How to get ahold of me:
 - @rawkintrevo
 - trevor.grant@ibm.com / rawkintrevo@apache.org
 - Mahout Dev and User Mailing Lists



Why does any of this matter?

- To disambiguate terms related to machine learning / streaming machine learning.
- Hopefully after this you
 - Won't keep using words wrong
 - Will know when someone else is
 - be pretentious
 - or don't
- Bonus material:
 - We build a fairly cool, yet super simple online recommender
 - Apache Flink + Apache Spark + Apache Mahout



Math. Eewww.

This talk invokes the following types of maths

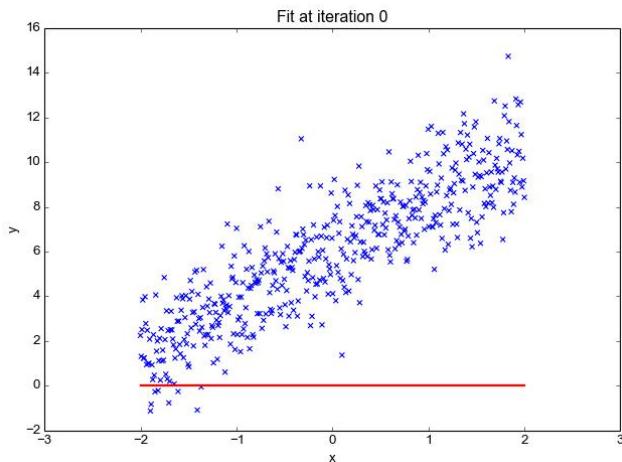
- Weighted Averaging
- Matrix Times Vector

Also there's pictures.



Types of Pictures

- Useful animations
- Unrelated animal pictures



<http://eli.thegreenplace.net/images/2016/regressionfit.gif>



Macs are good for keeping cat butts warm...
and not much else.



Table of Contents

Intro

Buzzwords

Basic Online Learners

Challenges

Lambda Recommender

Conclusions

- On the virtues of not throwing around buzzwords...
- Online vs. Offline
- Lambda vs. Kappa (w.r.t. machine learning)
- Statistical vs Adversarial
- Real-Time (one buzzword to rule them all)



Online vs. Offline

Online

- Input processed piece by piece in a serial fashion
- Each new piece of information generates an event
 - Not mini-batching
 - Possibly on a sliding window of record 1
- Not necessarily low latency

Offline

- Input processed in batches
- Not necessarily high latency



Fast offline, slow online and stack order

Slow Online

Stock broker in Des Moines Iowa writes Python program that gets EOD prices/statistics as they are published and then executes orders.

Fast Offline

HFT algorithm, executes trades based on tumbling windows of 15 milliseconds worth of activity

Online doesn't mean fast, online doesn't mean streaming, online ***only means that it processes information as soon as it is received.***

Consider an online algorithm (the slow online example), exists behind an offline EOD batch job.

- This is an extreme case, but no algorithm receives data as it is created.
- Best case- limited by speed of light (?)



Lambda vs. Kappa (Machine Learning)

Lambda

Learning happens (i.e. models are fitted)
offline

Model used by streaming engine to make
decisions online

Kappa

Learning happens (i.e. models are fitted)
online

Online decision model updates for each new
record seen

Model can change structure e.g. new words
in TF-IDF or new categories in ‘factor model’
‘linear regression’



Lambda with Novell Information

- A *trained model* expects structurally the same as training data.
- In linear regression, categorical features are “one-hot-encoded”. A feature with 3 categories expressed as a vector in 2 columns.
- What if a new category pops up?
 - Depends how you program it-
 - ignore the input
 - serve a bad response
- Consider clustering classification on text... new words?
 - Ignore: (probably what you'll do)
 - Word might be very important...



Kappa with Novell Information

- In Kappa, training happens with each new piece of data
 - Model data can account for structural change in data instantly
- New words can be introduced into TF-IDF
- New categories into a factor variable
- Both examples (and others) causes input vector to change.



Statistical vs. Adversarial

Traditional

Common statistical methods

- Supervised
- Unsupervised

Graded by

- Statistical Fitness Tests
- Out of core testing
- E.g.
 - Confusion Matrix, AuROC
 - MSE, MAPE, R2, MSE

Adversarial

Algorithm Versus Environment

- vs. Spammers
- vs. Hackers
- vs. Nature

Graded by

- Directionally can use some tests
- Really A/B testing
 - Adversaries may get smarter over time
 - Type of test where you automate adversary.



Real-time

- Subjective
- A good buzzword for something that:
 - Doesn't fall into any of the above categories cleanly
 - Doesn't fall into the category you want it to fall into
 - You're not really sure which buzzword to use, so you need a 'safe' word that no one can call you on.
 - Days
 - Weeks?
 - JJs



Table of Contents

Intro

- Streaming K-Means
- Streaming Linear Regression
- Why would I ever do with this?

Buzzwords

Basic Online Learners

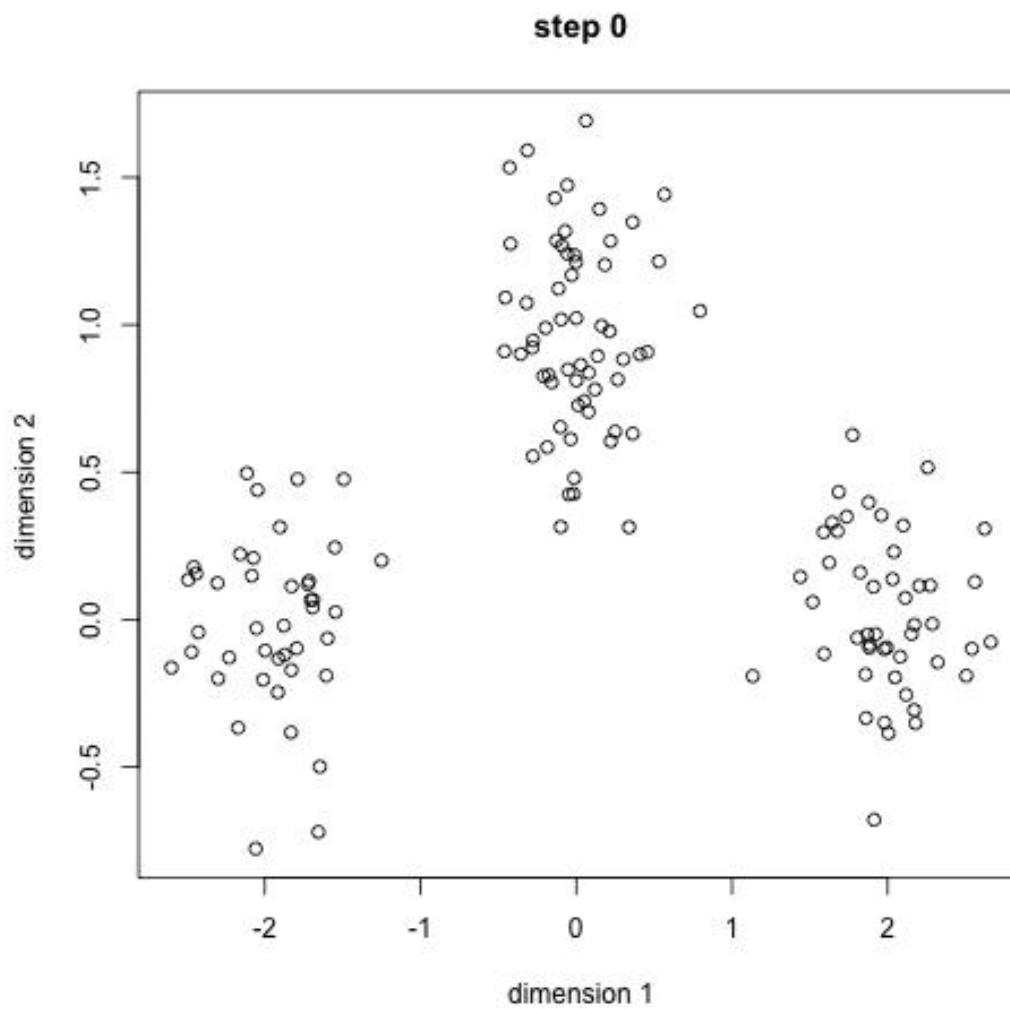
Challenges

Lambda Recommender

Conclusions



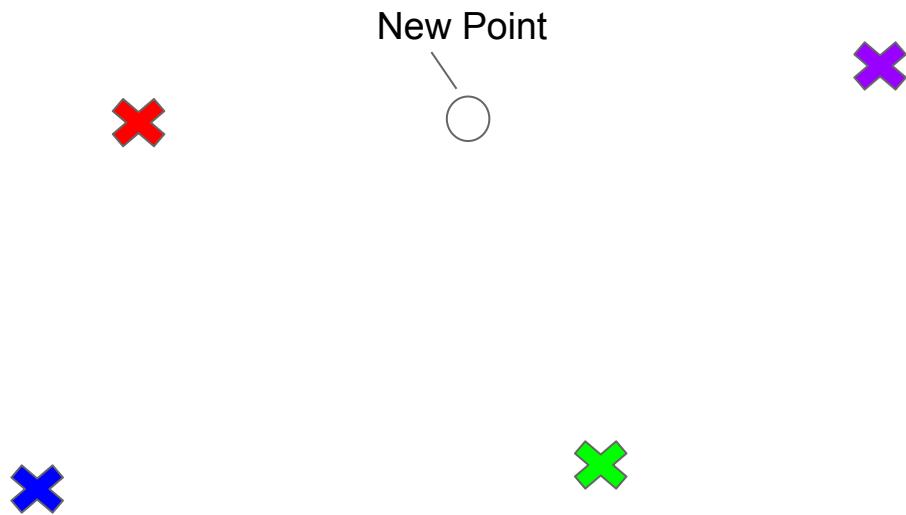
K-Means



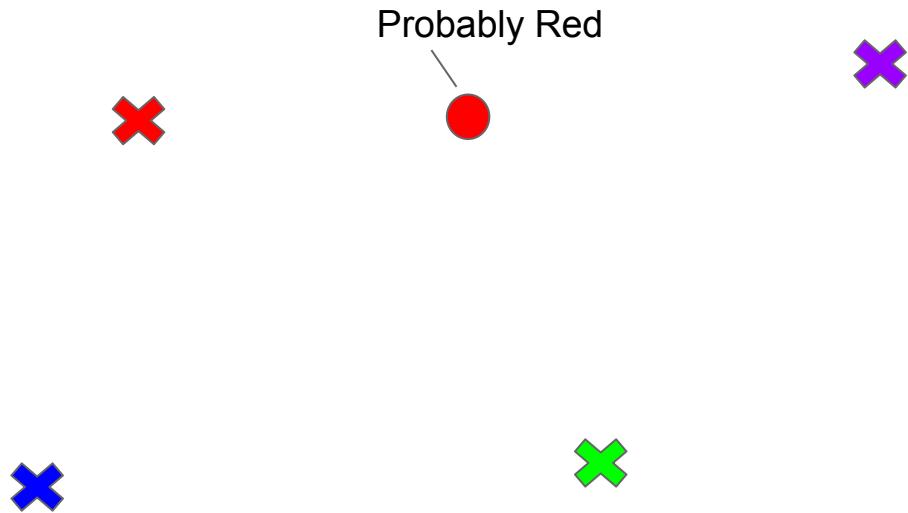
Online K-Means



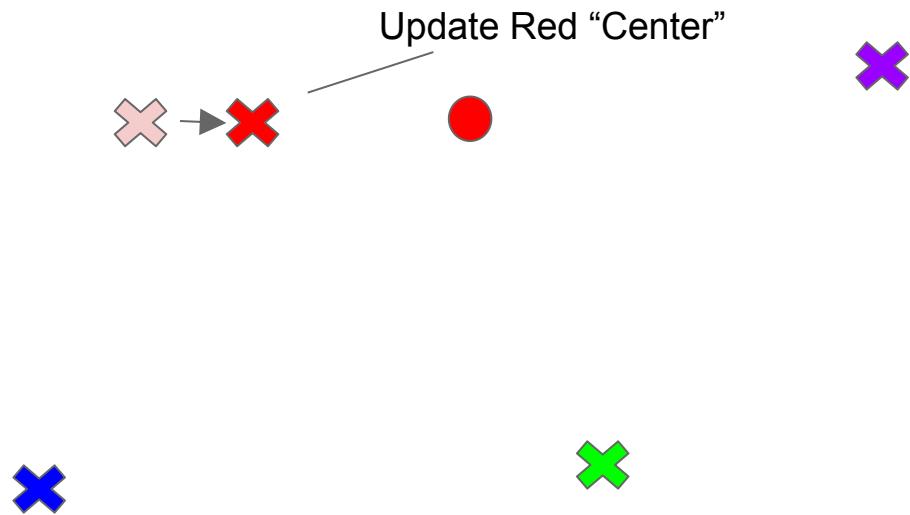
Online K-Means



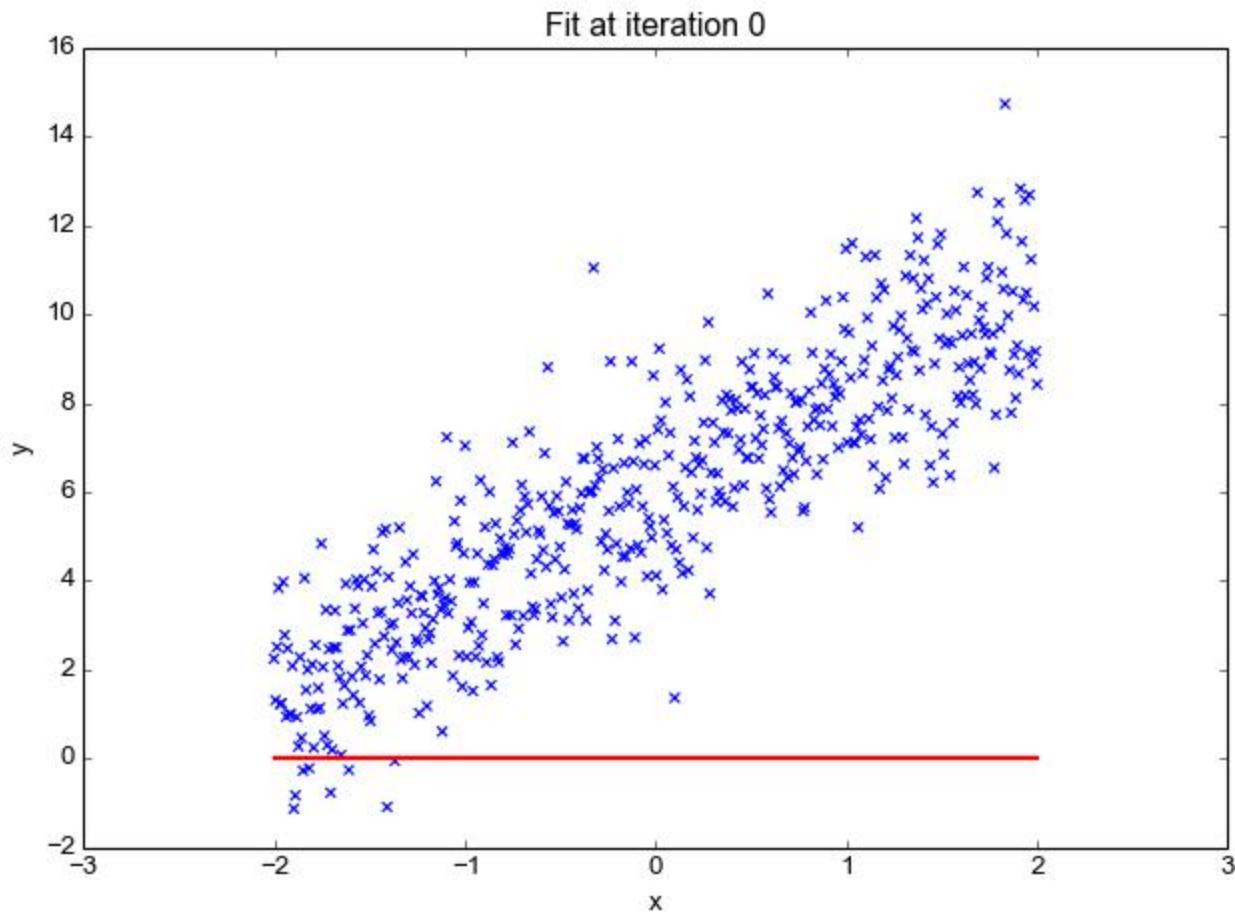
Online K-Means



Online K-Means



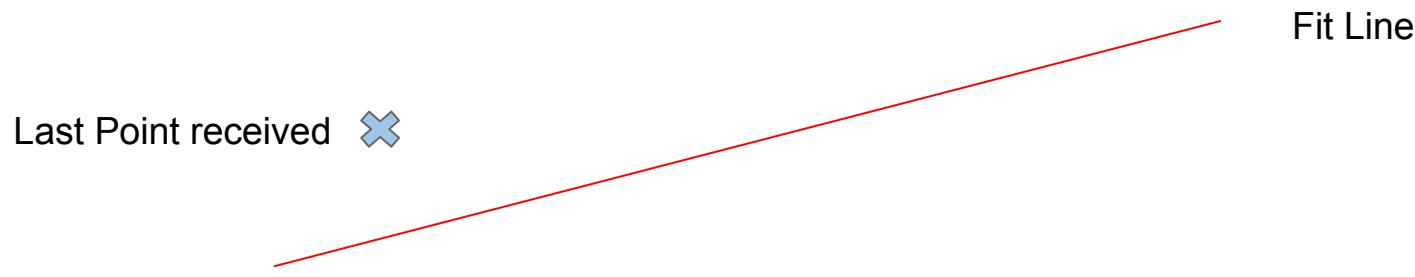
Linear Regression (Stochastic)



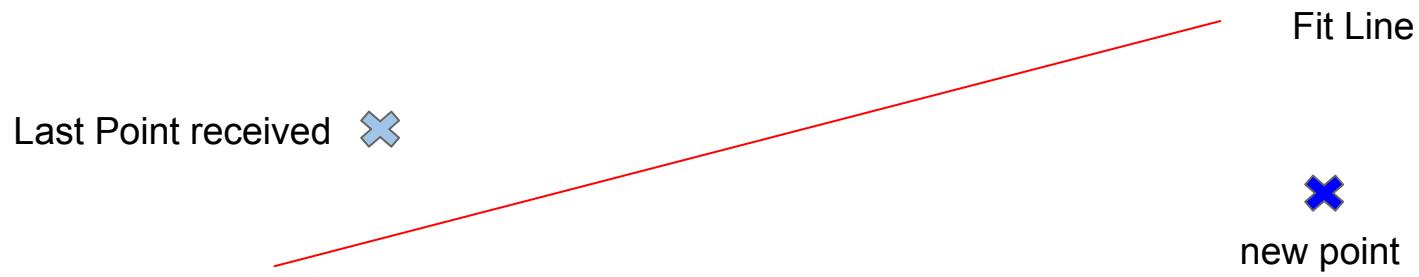
<http://eli.thegreenplace.net/images/2010/regression001.gif>



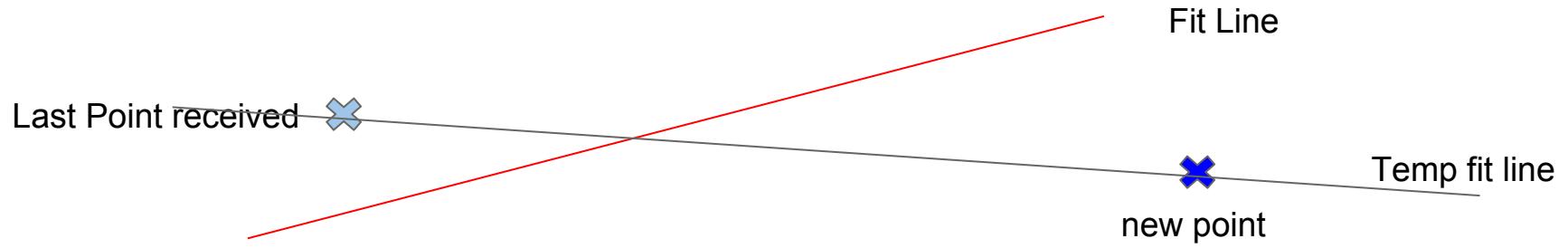
Online Linear Regression



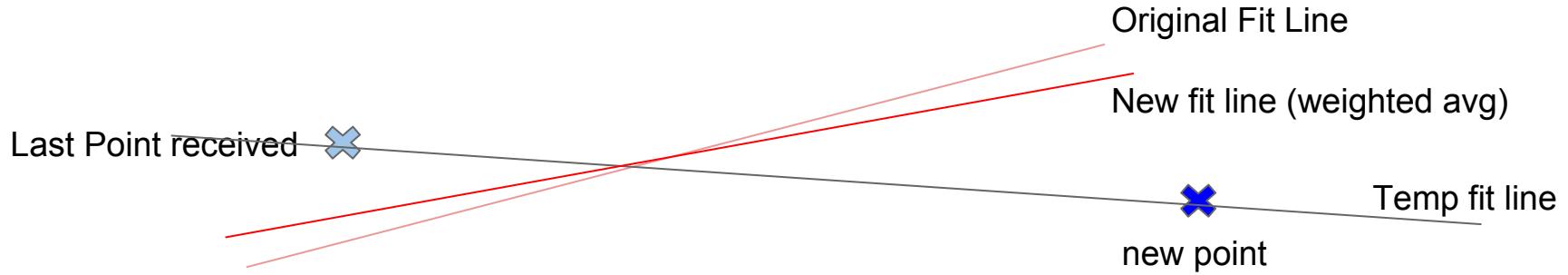
Online Linear Regression



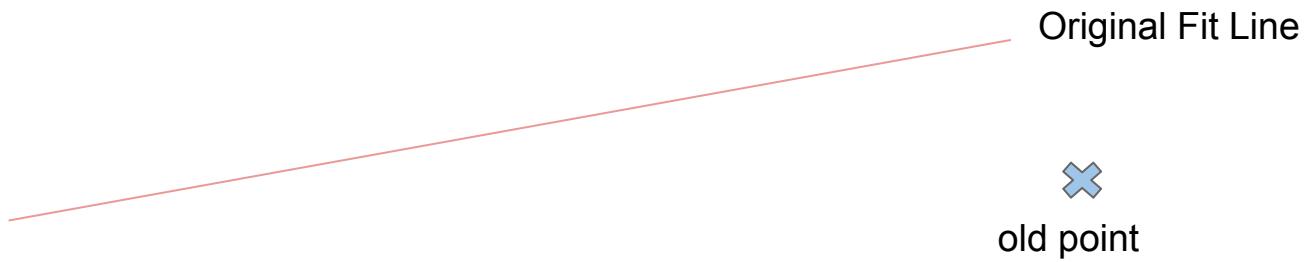
Online Linear Regression



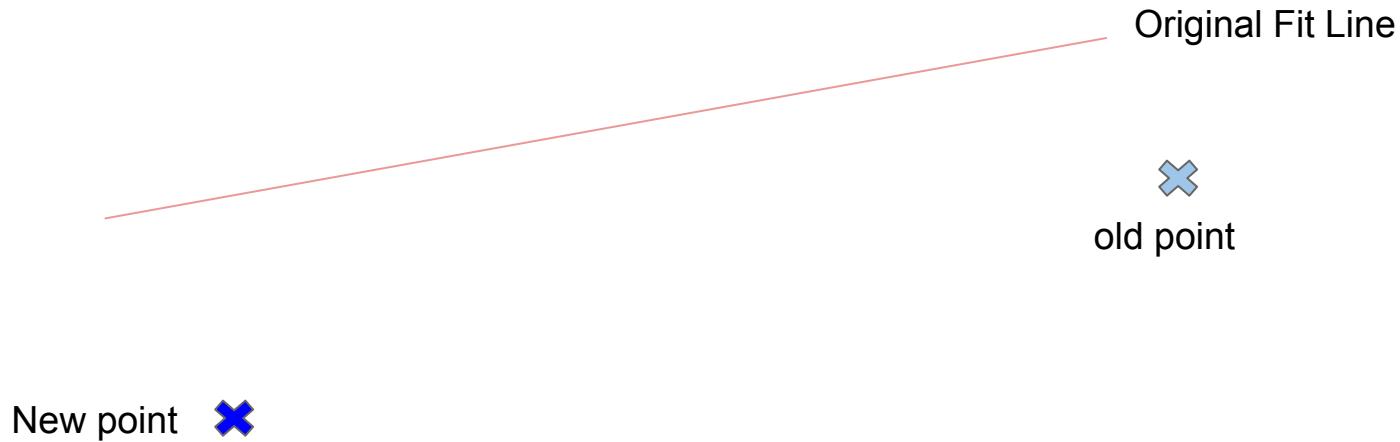
Online Linear Regression



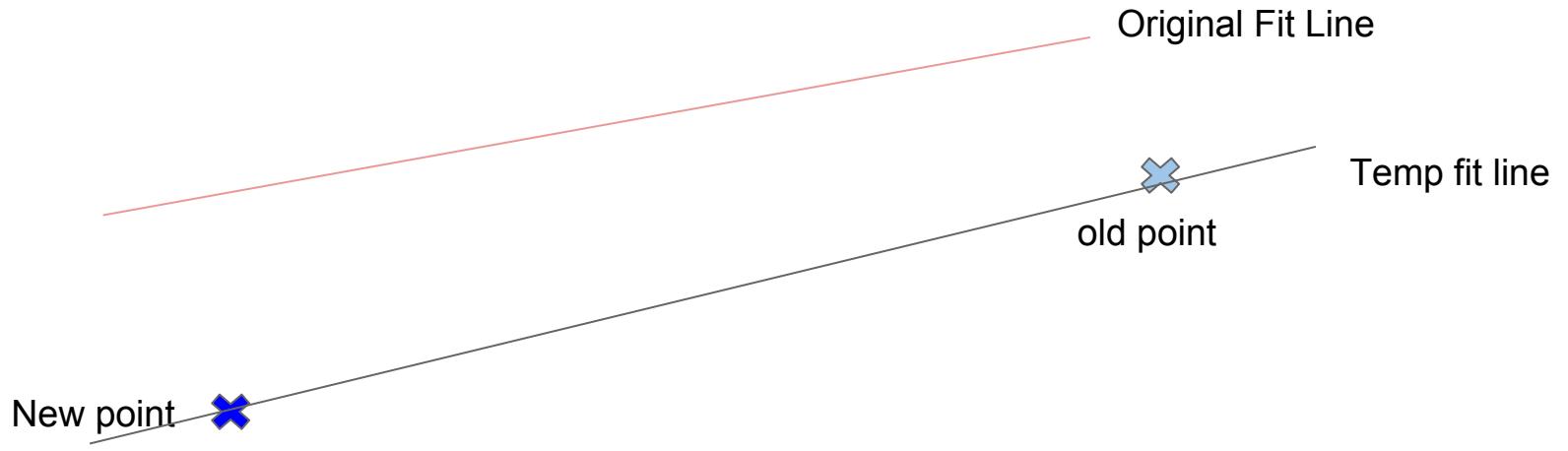
Online Linear Regression



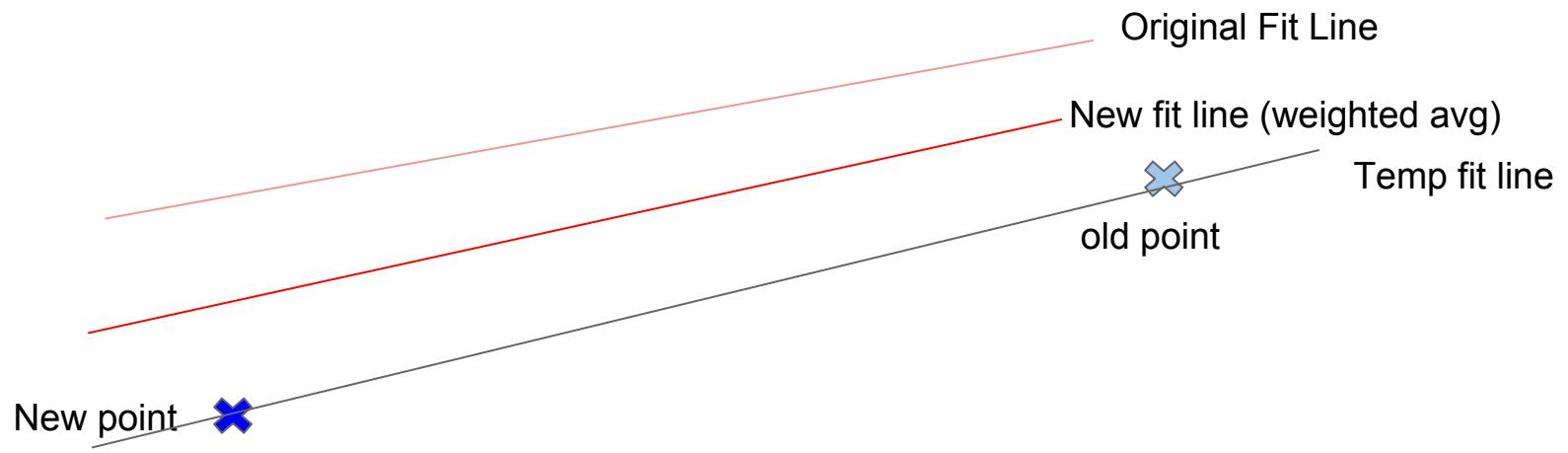
Online Linear Regression



Online Linear Regression



Online Linear Regression



Deep learning

This would work on neural networks too.

Also “Deep Learning” is another buzz word.



Why?

- Mostly Anomaly Detection (moving average, then something deviates)
 - A very popular use case of online/streaming algorithms (more talks today about this)
 - Algorithm learns what is normal (either online or offline)
 - When normality is sufficiently violated- the algorithm sounds an alarm
 - All anomaly detections some flavor of this. Usually referred to as:
_____ Anomaly Detection, only to specify what algorithm was used for defining normality (or lack there-of).
 - Architecture: online-offline training choices depend primarily on how fast ‘normality’ changes in your specific use case



Table of Contents

Intro

- Adversarial Analysis
- Scoring in Real Time (how do you know you're right?)
- A/B Tests

Buzzwords

Basic Online Learners

Challenges / Solutions

Lambda Recommender

Conclusions



Learning in real-time with supervised methods (challenge)

CHALLENGE:

How do you know how far you ‘missed’ prediction? In real life ‘correct’ answers may arrive later.

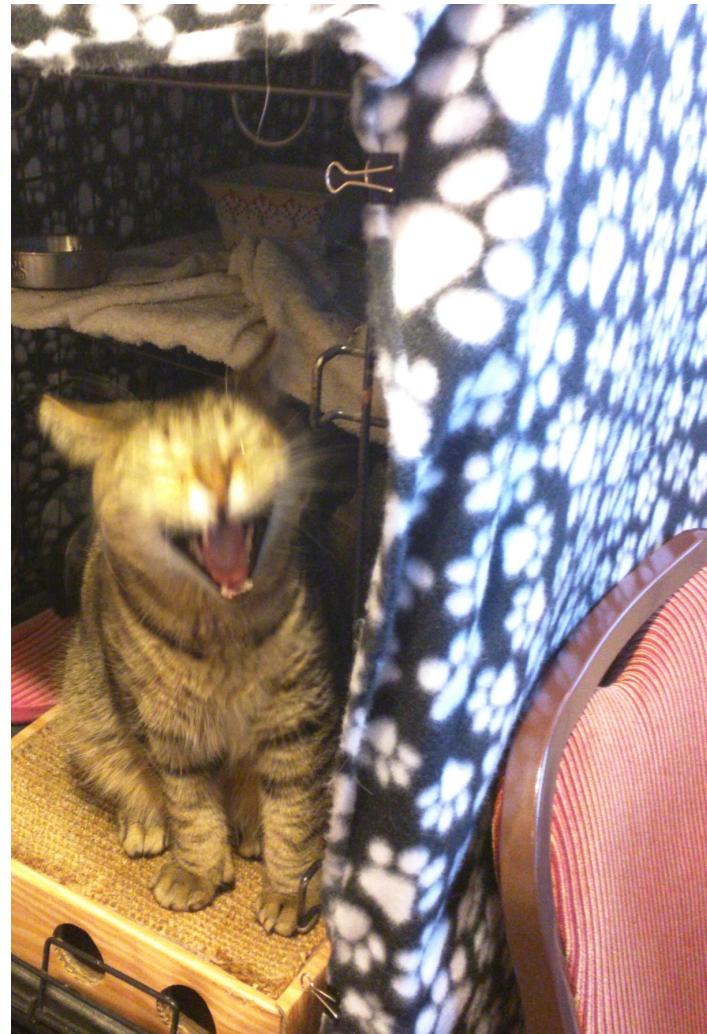
Corollary: If you have ‘correct’ answer why are you trying to predict it?

Not insurmountable, but prevents ‘one size fits all’ approaches (context dependence).



Latency and Normal Streaming Problems

You've only got so much hardware.



Adversarial Analysis

Simple Adversary-

How well does the algorithm do against “offline” version?

Consider Linear Regression with SGD

- Offline algorithm gets over full data set, then predicts
- Online model gets single pass to train and predict

How much worse is online than offline?



A/B Tests- The gold standard

Online algos are often *interacting* with the environment.

Learning rates, other knobs.



Table of Contents

Intro

- Correlated Co Occurrence – Brief Primer
- Architecture Overview
- Code walk through
- Looking at (pointless) results.

Buzzwords

Basic Online Learners

Challenges

Lambda Recommender

Conclusions



Correlated Co Occurrence Recommender: Overview / Benefits

- Overview of CCO
 - Collaborative Filtering (Like ALS, etc.)
 - Behavior Based (also like ALS)
 - Uses co-occurrence (no matrix factorization, unlike ALS)
 - Multi-modal: more than one behavior considered (unlike ALS / CO)
- Benefits of CCO
 - Many types of behaviors can be considered at once
 - Can make recommendations for users never seen before.



CCO Math

A Simple Co-Occurrence Recommender

▪

$$r = [P^T P] h_p$$

- r – recommendations
- P – history of all users on primary action (e.g. purchases)
 - Rows: user,
 - Columns: “Action” – e.g.(product1, product2, product3)
 - Then Row: Trevor, column: product2 => Trevor bought product 2
- $[P^T P]$ – Log Likelihood based correlation test
- h_p - A user’s history on behavior p (could be new user)



CCO Math

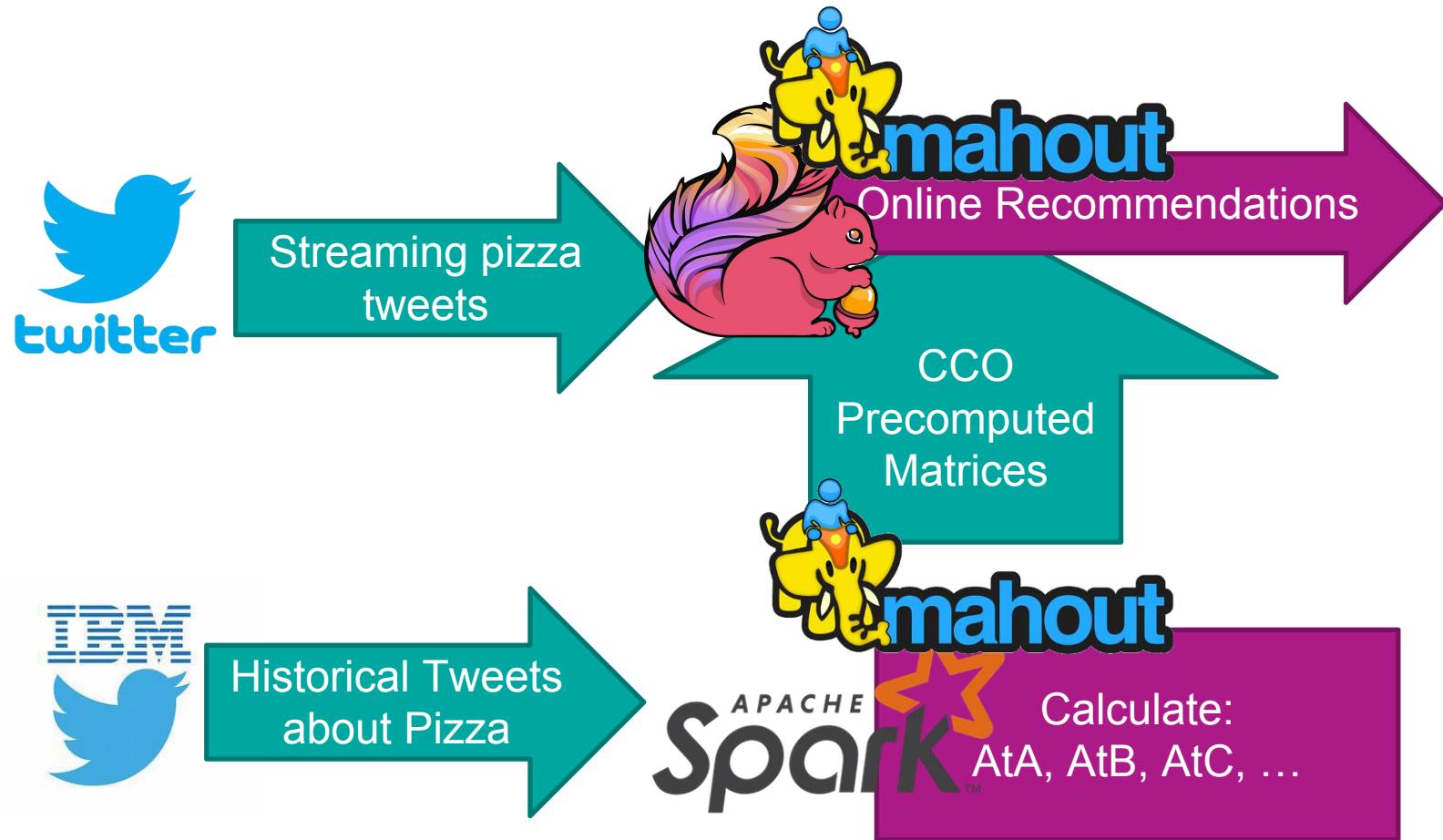
Correlated Co-Occurrence Recommender

$$r = [P^T P]h_p + [P^T A]h_a + [P^T B]h_b + \dots$$

- r – recommendations
- P – history of all users on primary action (e.g. purchases)
- $[P^T P]$ – Log Likelihood based correlation test
- A – history of all users on secondary action
 - Must have some rows (e.g. users)
- B – history of all users on tertiary action
 - Must have some rows (e.g. users)
- h_p - A user's history on behavior p (could be new user)



Architecture: Lambda CCO (Logo soup)



Python pulled in historical tweets and did this UserID - HashTag

561918328478785536,None
561918357851897858,None
561909179716481024,pizzagate
561909179716481024,gamergate
561949040011931649,None
561948991777038336,None
561947869805285377,superbowl
561947869805285377,pizzapizza
561918920282476545,None
561926796778565632,gunfriendly
561927577351503873,None



Python pulled in historical tweets and did this UserID - Words

561684486380068865,savethem000
561684486380068865,i
561684486380068865,dunno
561684486380068865,smiles
561684486380068865,want
561684486380068865,to
561684486380068865,get
561684486380068865,some
561684486380068865,pizza
561684486380068865,or
561684486380068865,something
561684441526194176,pizza
561684441526194176,de
561684441526194176,queso
561684441526194176,lista
561684441526194176,para



Some Spark Code

```
import org.apache.mahout.sparkbindings.indexeddataset.IndexedDatasetSpark
import org.apache.mahout.math.cf.SimilarityAnalysis

val baseDir = "/home/rawkintrevo/gits/ffsf17-twitter-recos/data"
// We need to turn our raw text files into RDD[(String, String)]

val userFriendsRDD = sc.textFile(baseDir + "/user-friends.csv")
.map(line => line.split(",")).filter(_.length == 2).map(a => (a(0), a(1)))
val userFriendsIDS = IndexDatasetSpark.apply(userFriendsRDD)(sc)

val userHashtagsRDD = sc.textFile(baseDir + "/user-ht.csv")
.map(line => line.split(",")).filter(_.length == 2).map(a => (a(0), a(1)))
val userHashtagsIDS = IndexDatasetSpark.apply(userHashtagsRDD)(sc)

val userWordsRDD = sc.textFile(baseDir + "/user-words.csv")
.map(line => line.split(",")).filter(_.length == 2).map(a => (a(0), a(1)))
val userWordsIDS = IndexDatasetSpark.apply(userWordsRDD)(sc)

val hashtagReccosLlrDrmListByUser = SimilarityAnalysis.cooccurrencesDSs(
Array(userHashtagsIDS, userWordsIDS, userFriendsIDS),
maxInterestingItemsPerThing = 100,
maxNumInteractions = 500,
randomSeed = 1234)
```



CCO Math

Spark+Mahout just Calculated these:

$$r = [P^T P]h_p + [P^T A]h_a + [P^T B]h_b + \dots$$

- r – recommendations
- P – history of all users on primary action (e.g. purchases)
- $[P^T P]$ – Log Likelihood based correlation test
- A – history of all users on secondary action
 - Must have some rows (e.g. users)
- B – history of all users on tertiary action
 - Must have some rows (e.g. users)
- h_p – A user's history on behavior p (could be new user)



Some Flink Code

```
streamSource.map(jsonString => {
    val result = JSON.parseFull(jsonString)

    val output = result match {
        case Some(e) => {
            //*****
            * Some pretty lazy tweet handling
            val tweet: Map[String, Any] = e.asInstanceOf[Map[String, Any]]
            val text: String = tweet("text").asInstanceOf[String]
            val words: Array[String] = text.split("\\s+").map(word => word.replaceAll("[^A-Za-z0-9]", "")).toLowerCase()

            val entities = tweet("entities").asInstanceOf[Map[String, List[Map[String, String]]]]
            val hashtags: List[String] = entities("hashtags").toArray.map(m => m.getOrElse("text", "").toLowerCase()).toList
            val mentions: List[String] = entities("user_mentions").toArray.map(m => m.getOrElse("id_str", "")).toList

            //*****
            * Mahout CCO
            val hashtagsMat = sparse(hashtagsProtoMat.map(m => svec(m, cardinality = hashtagsBiDict.size)): _*)
            val wordsMat = sparse(wordsProtoMat.map(m => svec(m, cardinality = wordsBiDict.size)): _*)
            val friendsMat = sparse(friendsProtoMat.map(m => svec(m, cardinality = friendsBiDict.size)): _*)

            val userWordsVec = listOfStringsToSVec(words.toList, wordsBiDict)
            val userHashtagsVec = listOfStringsToSVec(hashtags, hashtagsBiDict)
            val userMentionsVec = listOfStringsToSVec(mentions, friendsBiDict)

            val reccos = hashtagsMat %*% userHashtagsVec + wordsMat %*% userWordsVec + friendsMat %*% userMentionsVec

            //*****
            * Sort and Pretty Print
        }
    }
}
```



CCO Math

Flink+Mahout just Calculated these:

$$r = [P^T P] h_p + [P^T A] h_a + [P^T B] h_b + \dots$$

- r – recommendations
- P – history of all users on primary action (e.g. purchases)
- $[P^T P]$ – Log Likelihood based correlation test
- A – history of all users on secondary action
 - Must have some rows (e.g. users)
- B – history of all users on tertiary action
 - Must have some rows (e.g. users)
- h_p - A user's history on behavior p (could be new user)



Tweets

```
text: joemalicki josephchmura well i can make a pizza i bet he cant so there
userWordsVec: so a well i there can he make pizza cant
hashtags used: List()
hashtags recommended:
(ruinafriendshipin5words : 13.941270843461098)
(worstdayin4words : 8.93444123705558)
(recipes : 8.423061768672596)
```

```
text: people people dipping pizza in milk im done
userWordsVec: people in im done pizza
hashtags used: List()
hashtags recommended:
(None : 18.560367273335828)
(vegan : 10.84782189800353)
(fromscratch : 10.84782189800353)
```

*Results were cherry picked- no preprocessing, this was a garbage in-garbage out algo for illustration purposes only.



Hybrid Lambda Architecture
Cognitive
Online Recommendations
GPU Accelerated
Adversarial
Algorithm

Also...

Don't do this in real life, probably. (you would use a service)



Table of Contents

Intro

- Trevor attempts to tie everything together into a cohesive thought
- Audience members asks easy questions
- Audience members buy speaker beer at after party

Buzzwords

Basic Online Learners

Challenges

Lambda Recommender

Conclusions



Final Thoughts

A lot of buzzwords have been flying around especially with respect to machine learning and streaming.

- Online
- Lambda / Kappa architecture
- Streaming machine learning
- Real time predictive model
- machine learning
- artificial/machine/cognitive intelligence
- cognitive
- blah- ^ pick 2.



Final Thoughts

Now that you've sat through this talk hopefully you can:

1. Call people out for trying to make their product/service/open source project/startup sound like a bigger deal than it is
2. Church up your product/service/open source project/startup to get clients/VC dummies excited about it without *technically* lying



Questions?

Buy trevor beers.

<https://github.com/rawkintrevo/fsf17-twitter-recos>

Machine Learning on Flink

Ted Dunning
MapR Technologies



Machine Learning on Flink (without Flink ML)

Ted Dunning
MapR Technologies



Machine Learning on Flink (without Flink ML*)

Ted Dunning
MapR Technologies



Machine Learning on Flink (without Flink ML*)

*well, with Flink ML if desired, but not just with Flink ML

Ted Dunning
MapR Technologies



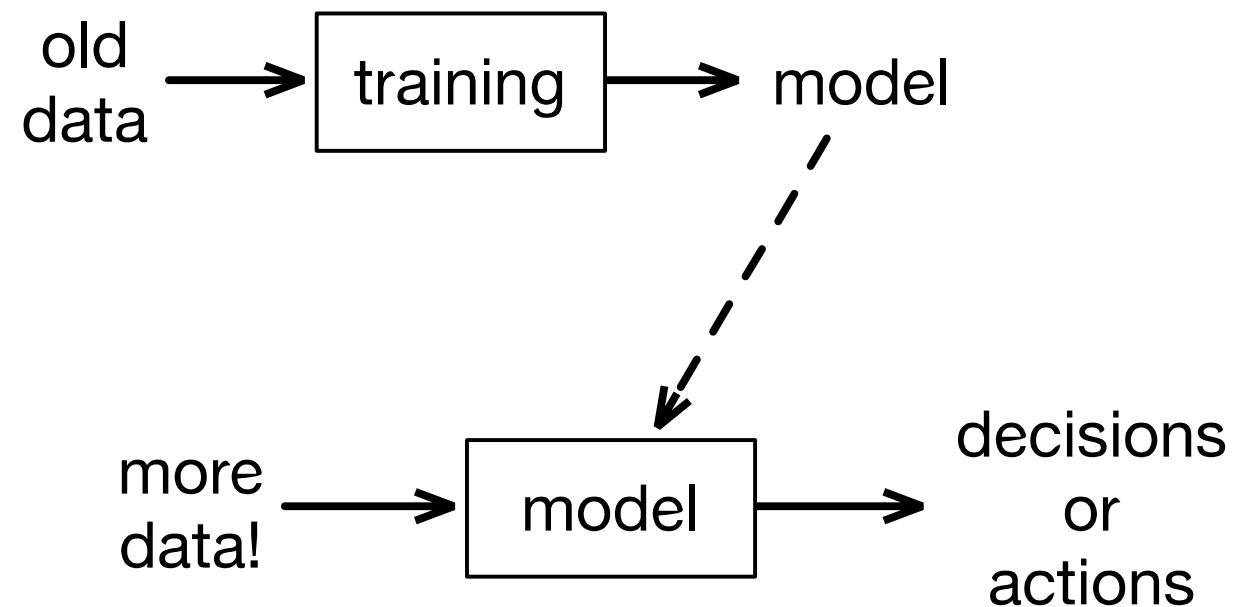
Where to Today?

- What is important in machine learning systems?
- What isn't like it seems?
- What can I/we/you do to improve it?

Traditional View



Traditional View



Not so much!

90% of effort is logistics,
not learning

Why Not?

- Just getting the training data is hard
 - Bootstrap problem for first models
 - First models and later can record input data
 - New kinds of observations force restarts
 - Requires a ton of domain knowledge
- The myth of the unitary model
 - You can't train just one
 - You will have dozens of models, likely hundreds to thousands
 - Handoff to new versions is tricky

Why Not? (continued)

- Recording raw-ish data is really a *big* deal
 - Data as seen by a model is worth gold
 - Data reconstructed later often has time-machine leaks
 - Databases were made for updates, streams are safer
- Real validation can be hard and takes time
 - Especially true if models effectively pick their own training data
 - Consider model over-ride for interesting data
 - Reject inference can be very difficult without a good framework
 - Thompson sampling can provide (near) optimal solution

How to Do Better ... Spoilers

- It's the data!
- Prioritize – put serious effort into infrastructure
- Persist – use streams to keep data around
- Measure – everything, and record it
- Meta-analyze – understand and see what is happening
- Containerize – make deployment repeatable, easy

- Oh... don't forget to do some machine learning, too

How to Do Better – Big Picture

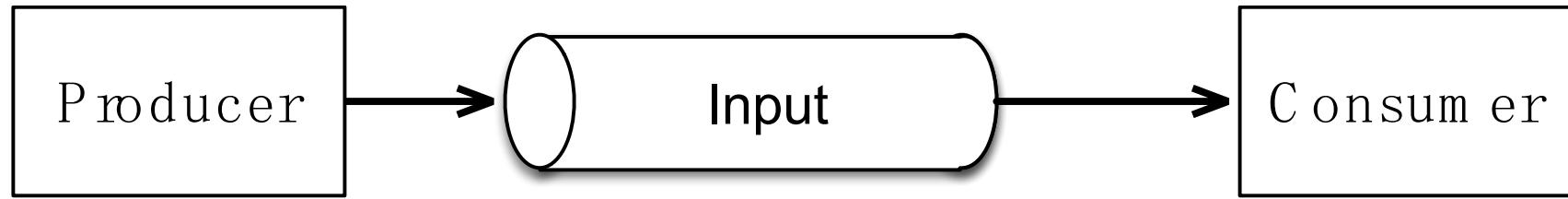
- The biggest improvements probably aren't machine learning
 - Algorithms are fun, but probably not the biggest bang
- Acquiring better data can make massive difference
 - Example: Clicks versus 30 second views
- Asking a better question can make massive difference
 - Example: When does new stock arrive versus stock recommender
- The 17th model revision will probably make <1% difference
- Consider \$/minute of effort, invest wisely

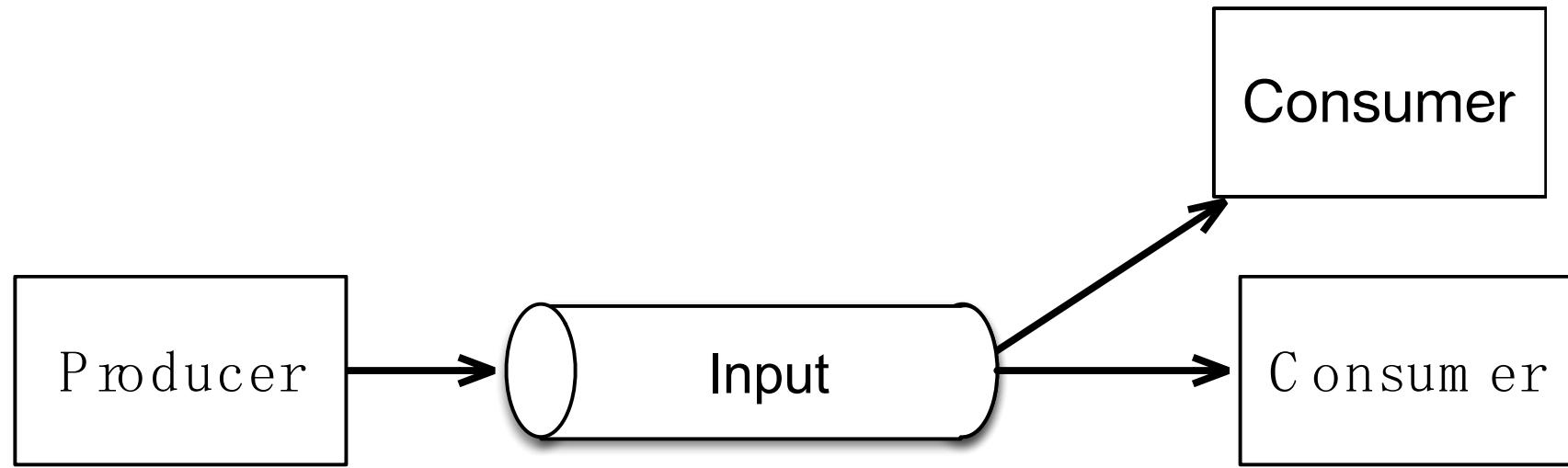
How to Do Better – Data and Deployment

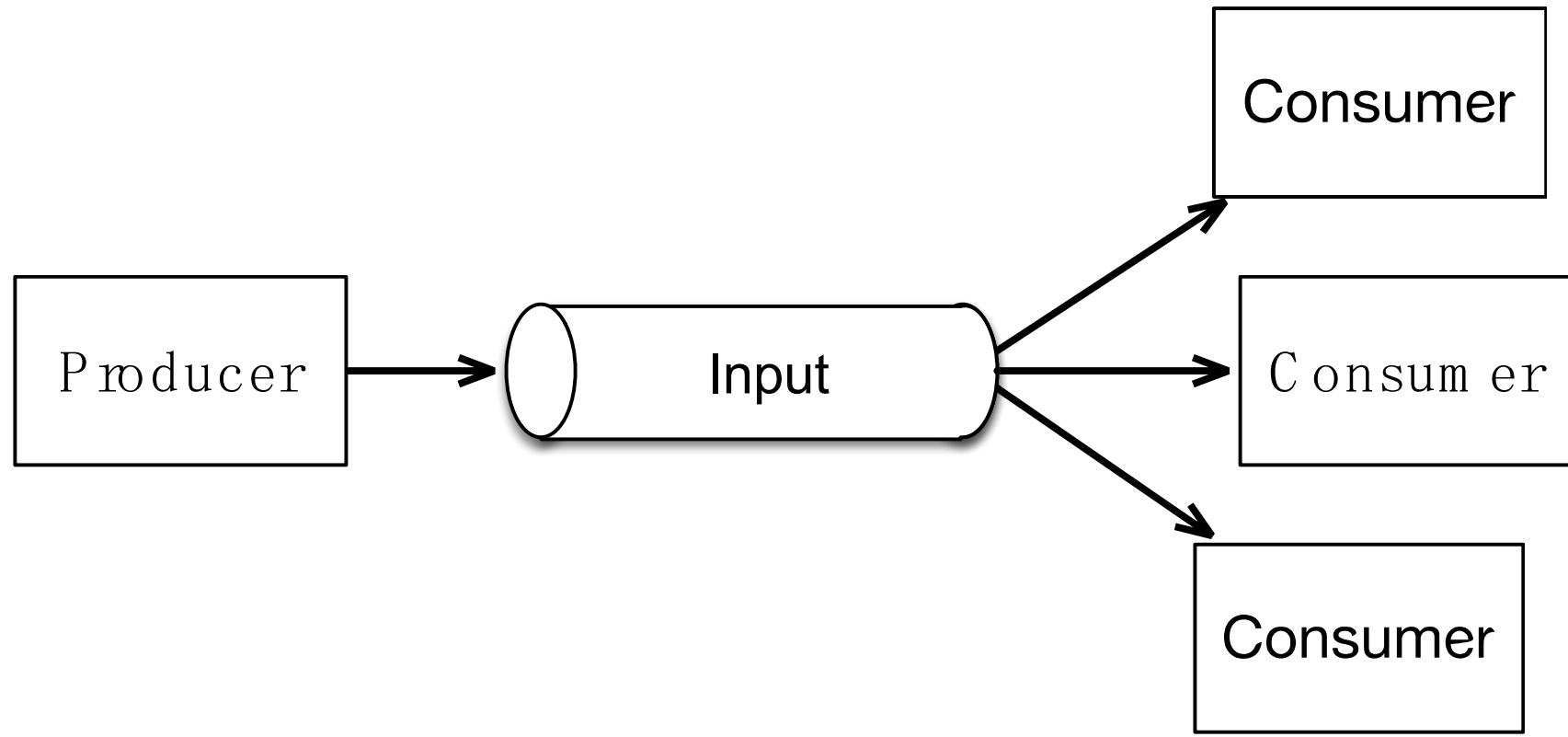
- Deploy a decoy model
 - Looks like a server, but it just archives inputs
 - Safe in a good streaming environment, less safe without good isolation
- Deploy a canary server
 - Keep an old model active as a reference
 - If it was 90% correct, difference with any better model should be small
 - Score distribution should be roughly constant
 - Many things can disturb the canary, but if it is stable, things are likely OK

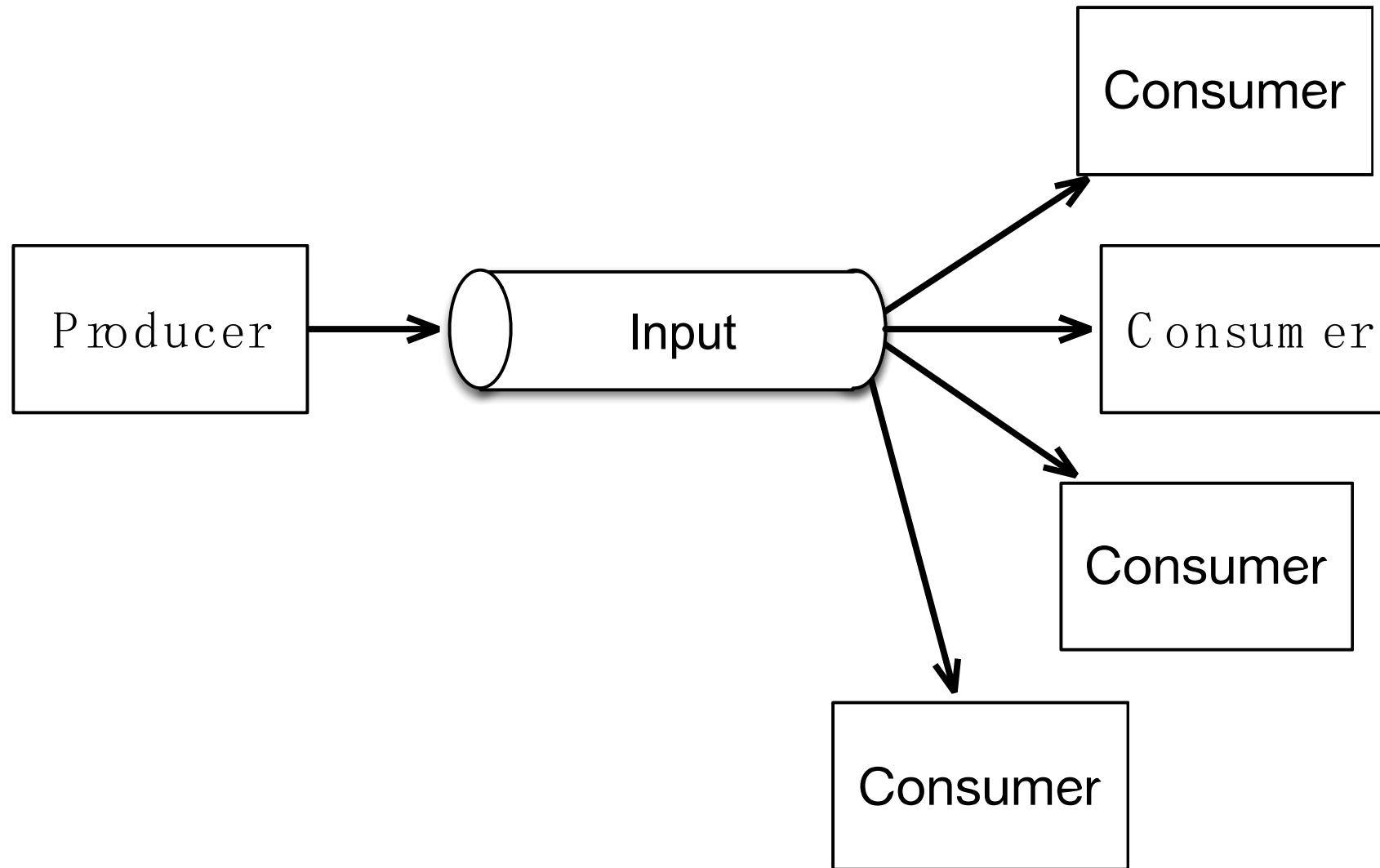
A decoy and a canary walked into a bar
and asked for a bottle of Club Maté..

Bartender said, “Wouldn’t you rather have a stream?”







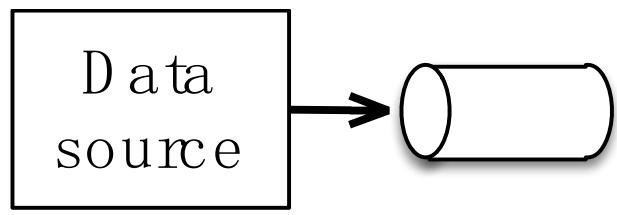


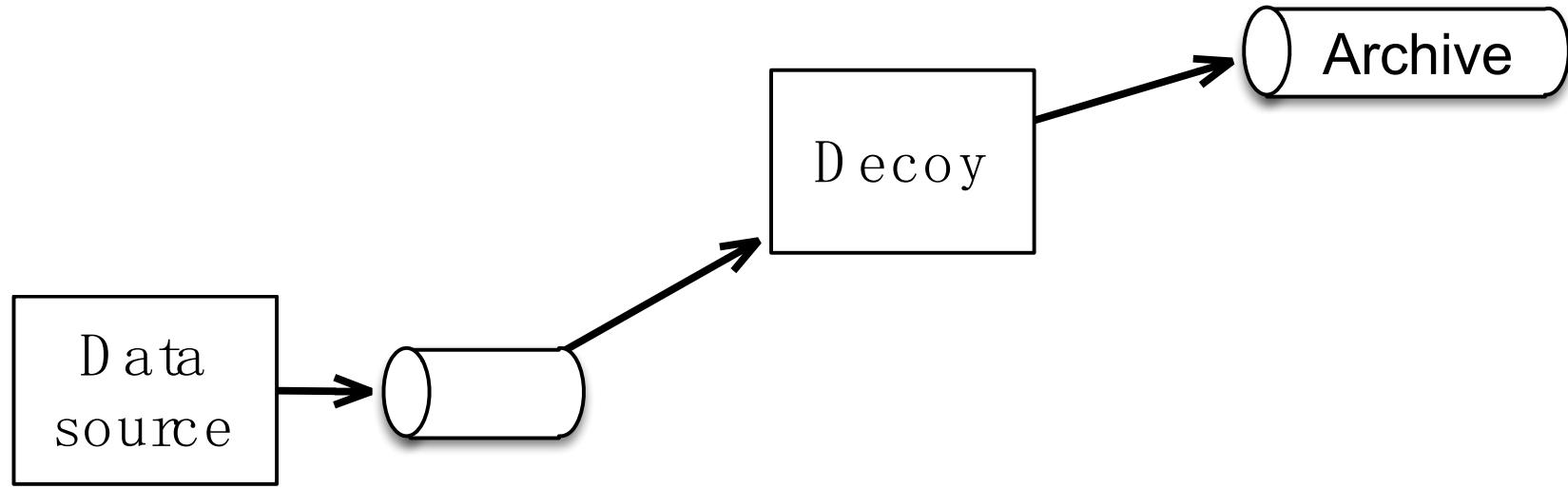
Features of Good Streaming

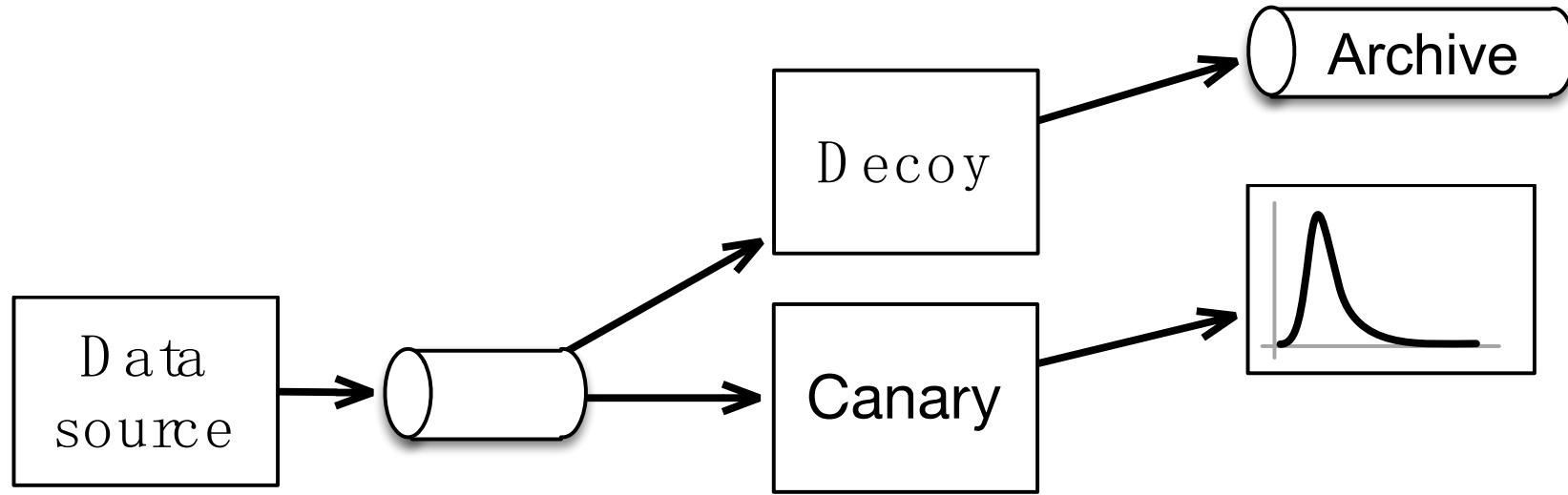
- It is Persistent
 - Messages stick around for other consumers
 - Consumers don't affect producers
- It is Performant
 - You don't have to worry if a stream can keep up
- It is Pervasive
 - It is there whenever you need it, no need to deploy anything
 - How much work is it to create a new file? Why harder for a stream?

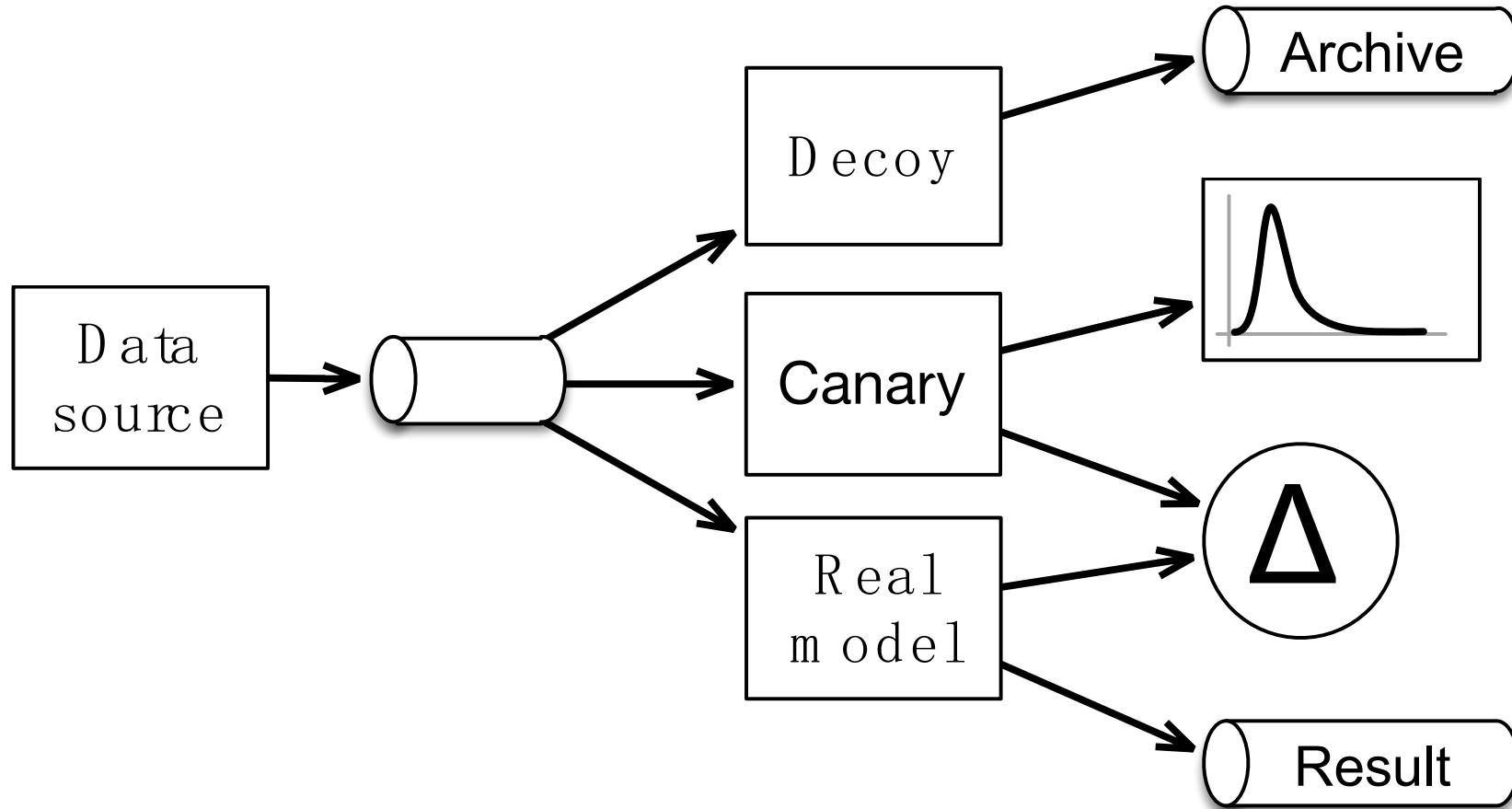
How to Do Better – Data and Deployment

- Deploy a decoy model
 - Looks like a server, but it just archives inputs
 - Safe in a good streaming environment, less safe without good isolation
- Deploy a canary server
 - Keep an old model active as a reference
 - If it was 90% correct, difference with any better model should be small
 - Score distribution should be roughly constant
 - Many things can disturb the canary, but if it is stable, things are likely OK



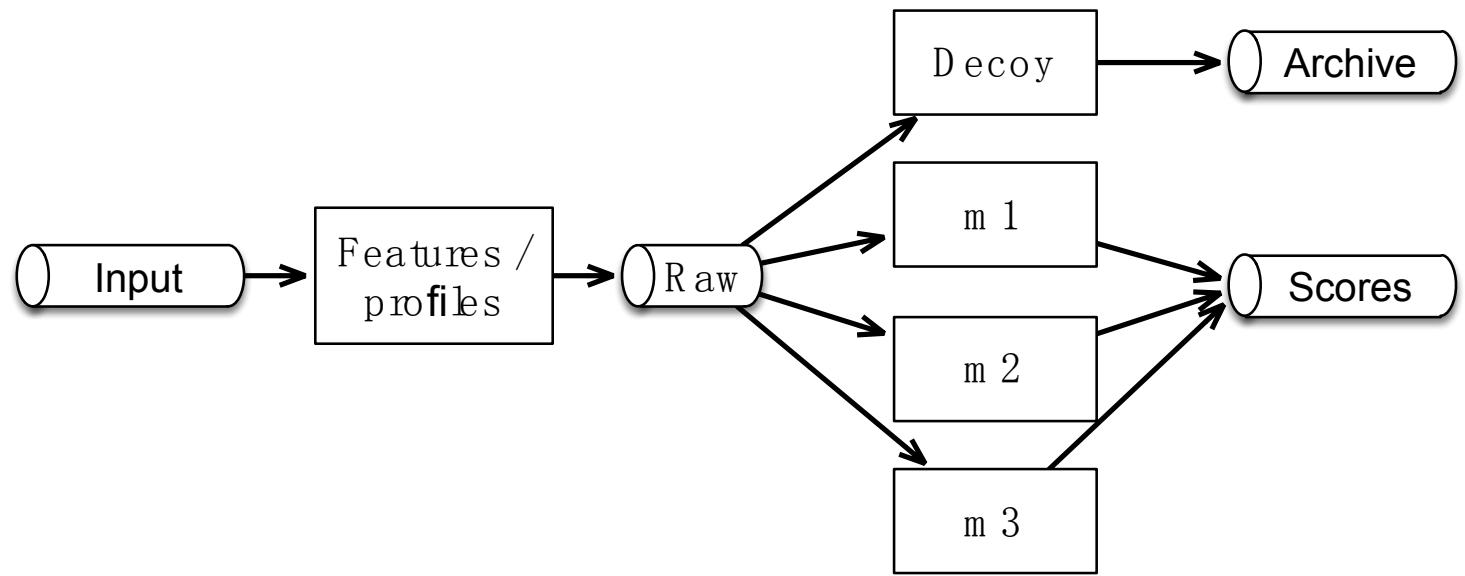






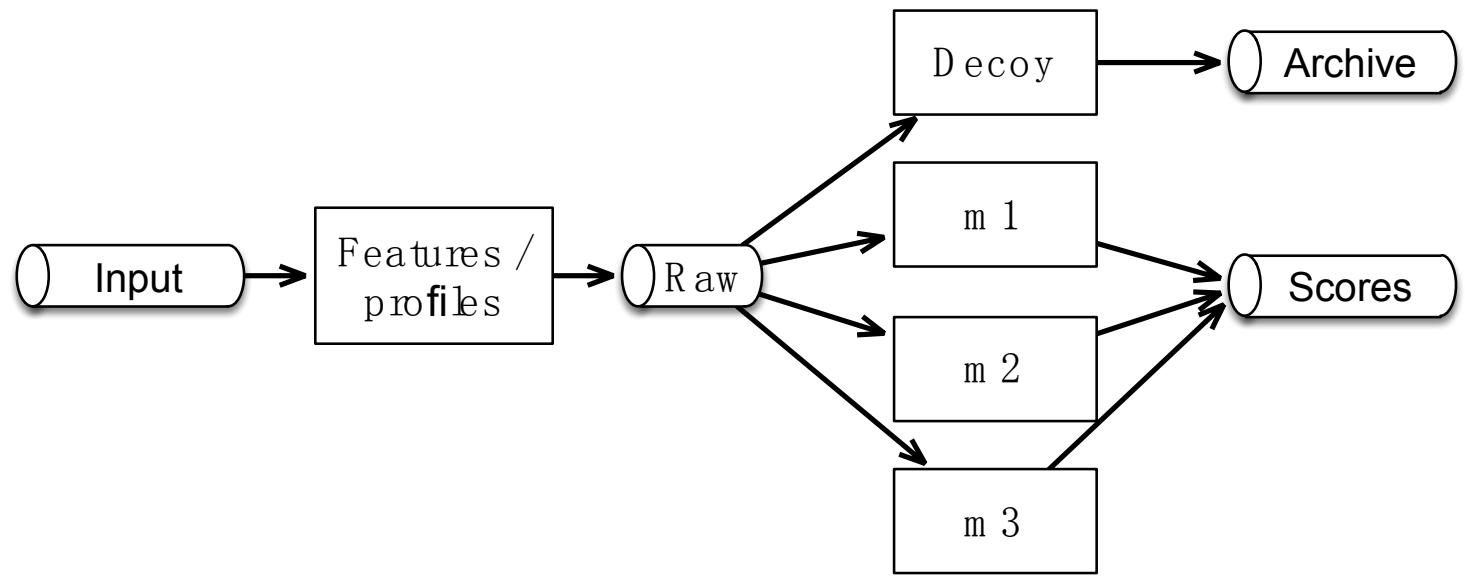
How to Do Better – Containers and Rendezvous

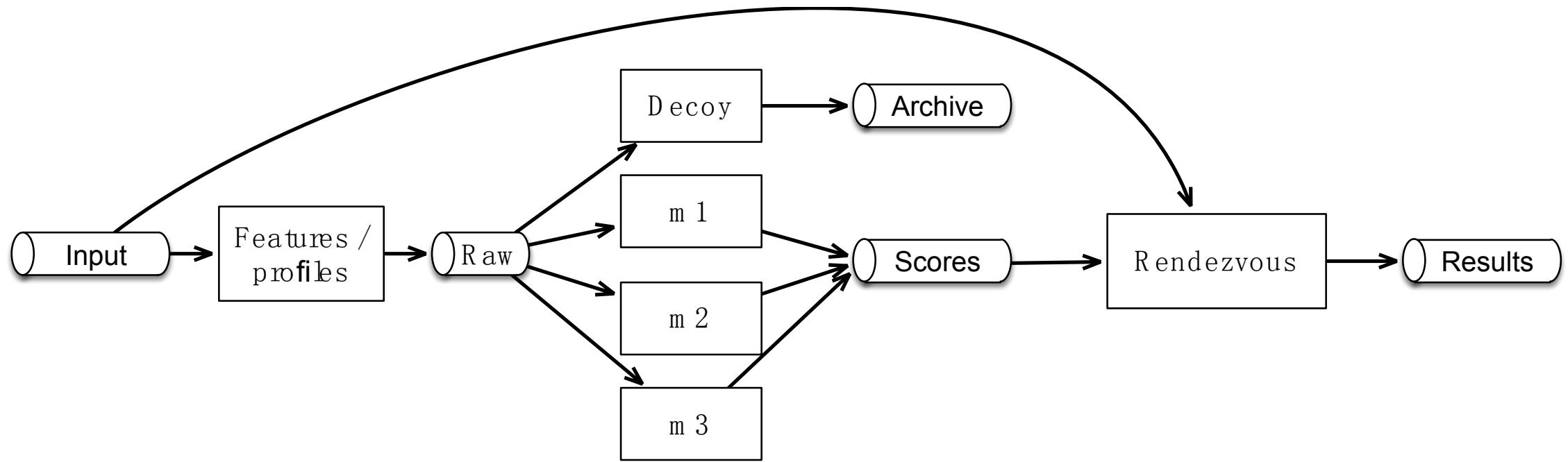
- Containerize all models
 - Safe deployment
 - Take input from stream
 - Write output to stream
 - Annotate output with meta-data about who, what and when
 - Pre-rig for standard operations like metrics, timing, annotation

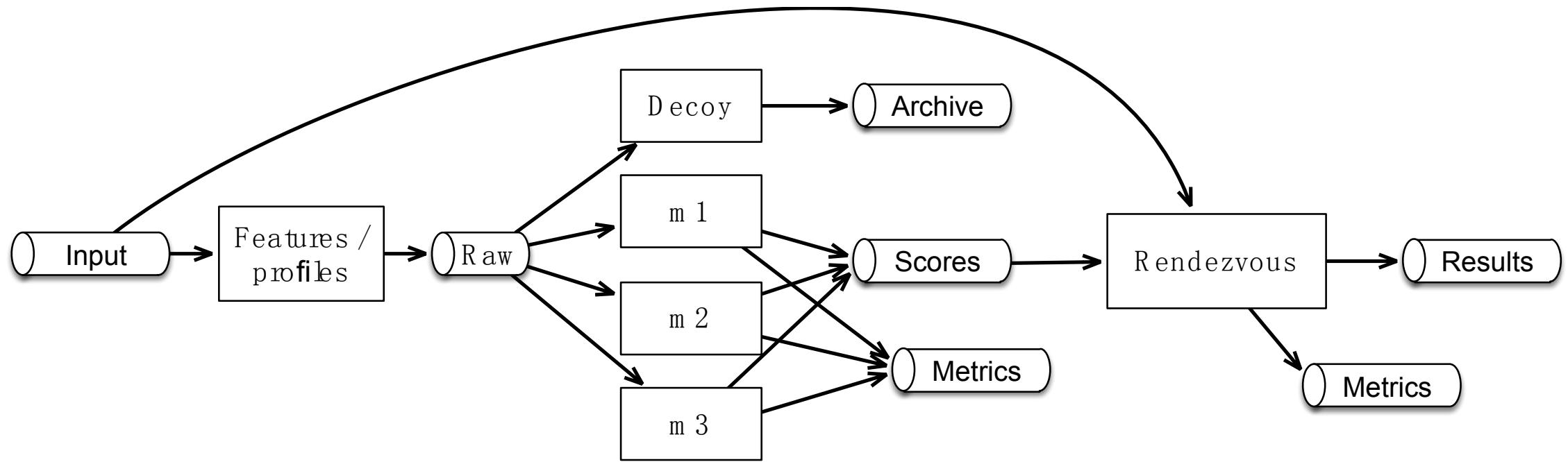


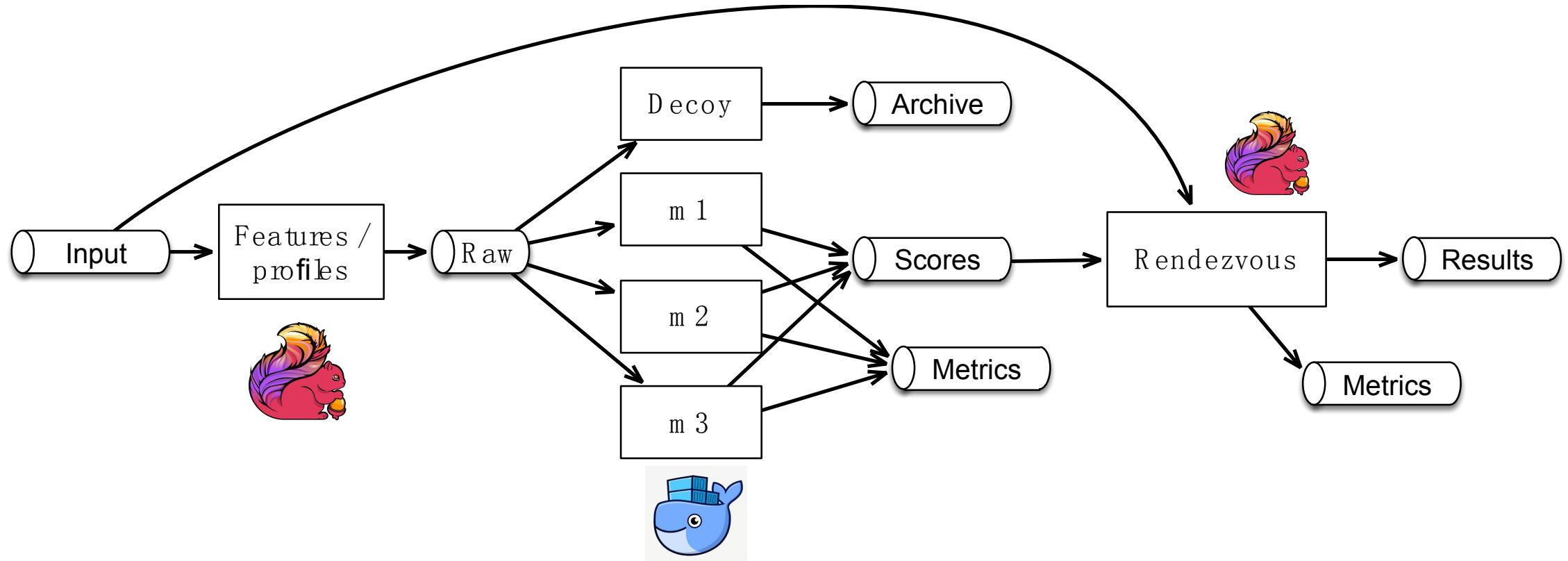
How to Do Better – Containers and Rendezvous

- Select desired model result
 - Thompson sampling, dithering
 - Defaults after extraordinary delays
- Rendezvous with original input
 - Or carry origin meta-data
- Record metrics
 - Latency
 - Score distributions
 - Selected model







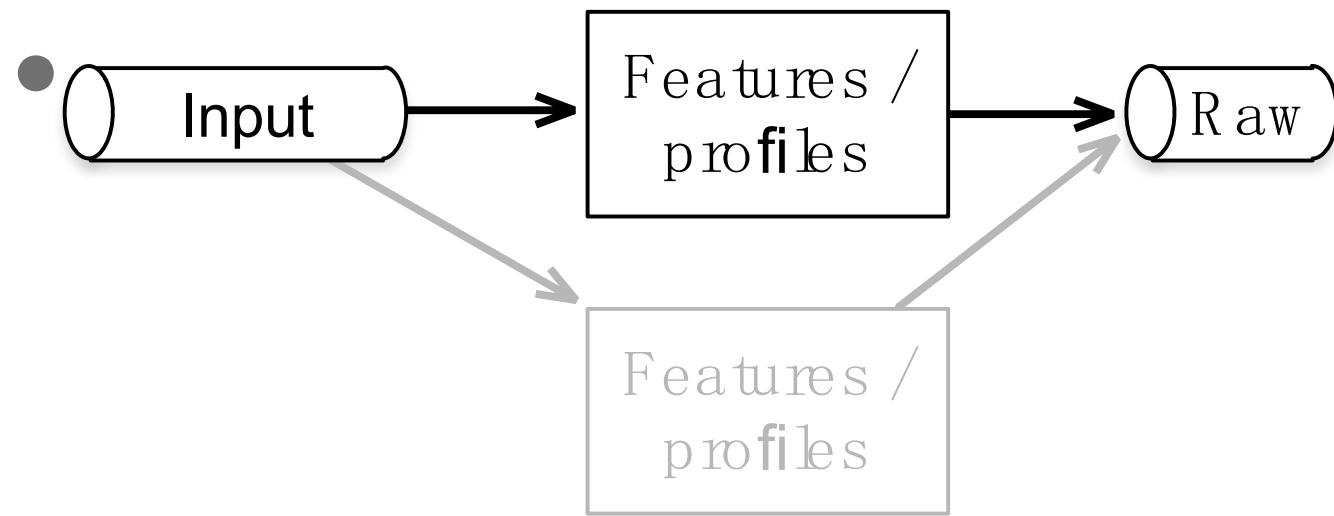


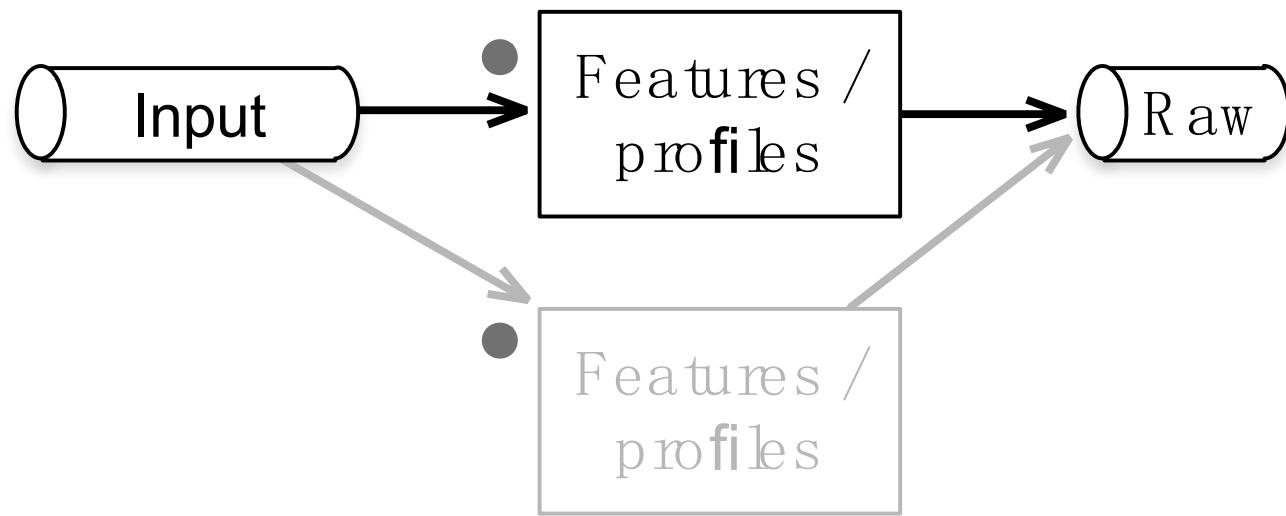
How to Do Better – Model Deployment as a Data Op

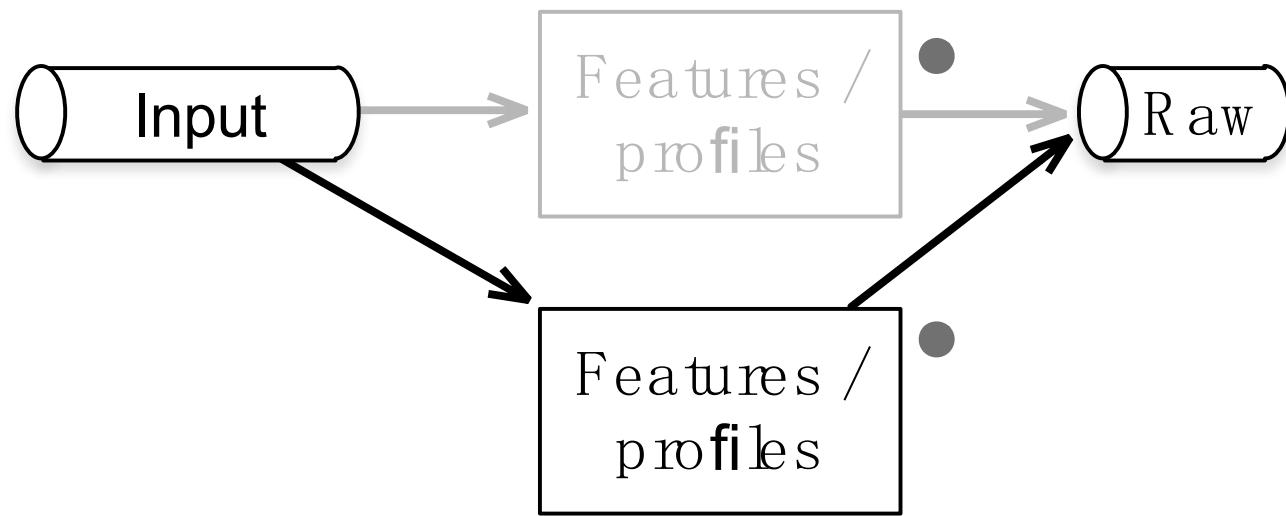
- Deploying a new model means spin some new containers
- Results will flow into the scoring stream
 - And will be ignored (initially)
- Tell the rendezvous service to use new model results
 - Start slowly
 - Check latencies, delta versus champion and canary
- Rendezvous service needs fallback rules
 - Use m3, after x ms use m2 if available, after y ms more, use m1
 - Must **always** give some answer within the SLA
 - Fancy statistics possible

Who Deploys the Deployer?

- Flink savepoints can be used to deploy new feature extractor, rendezvous service
- Underlying Chandy-Lamport algorithm can also be coded explicitly (not recommended for most folks)
- Hot swap token propagates through system to indicate when to switch







Takeaways

- It's the data!
- Prioritize – put serious effort into infrastructure
- Persist – use streams to keep data around
- Measure – everything, and record it
- Meta-analyze – understand and see what is happening
- Containerize – make deployment repeatable, easy
- Oh... don't forget to do some machine learning, too

Resources

- General principles of ML engineering with a Googlish accent:

<http://bit.ly/ml-best-practices-1>

- More good practices:

<http://bit.ly/ml-best-practices-2>

- Best short feature on feature engineering I know about:

<http://bit.ly/feature-engineering-1>

Who I am

Ted Dunning, Chief Applications Architect, MapR Technologies

Board of Directors, Apache Software Foundation

Email tdunning@mapr.com tdunning@apache.org

Twitter @Ted_Dunning

Q&A

Engage with us!

@mapr



maprtech

mapr-technologies



MapR

tdunning@mapr.com



maprtech



Make the cloud work for you

Easy, fast, and low-cost streaming with Apache Flink on Google Cloud Platform

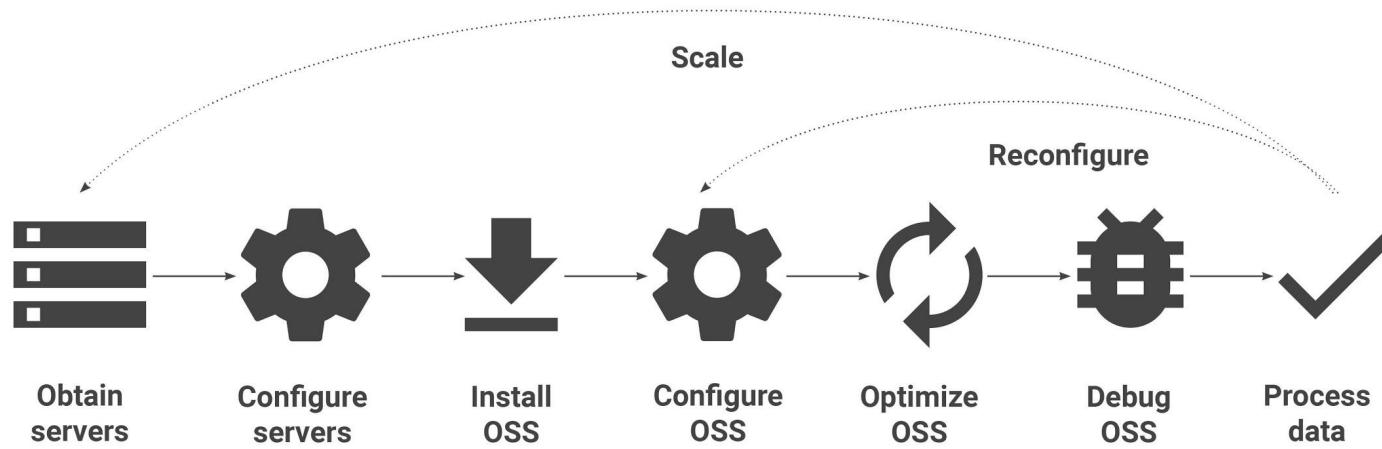
James Malone [jamesmalone@google.com]

Open source
powerful but complex

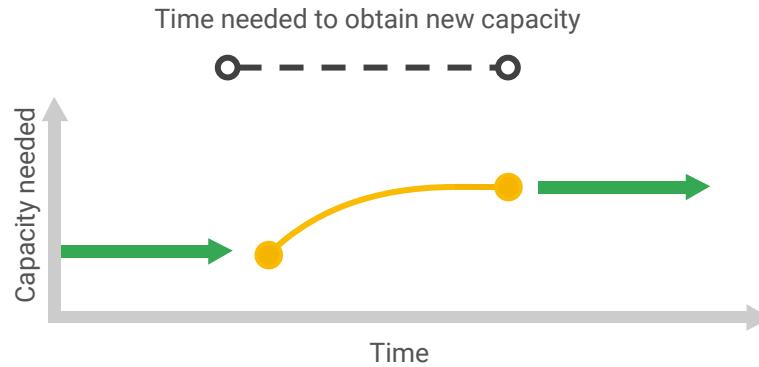
The Apache ecosystem



Typical OSS deployments

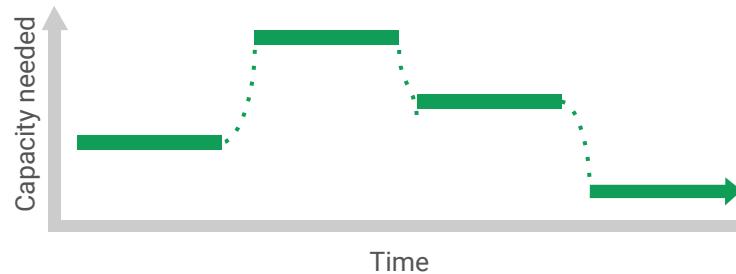


Scaling makes your life difficult



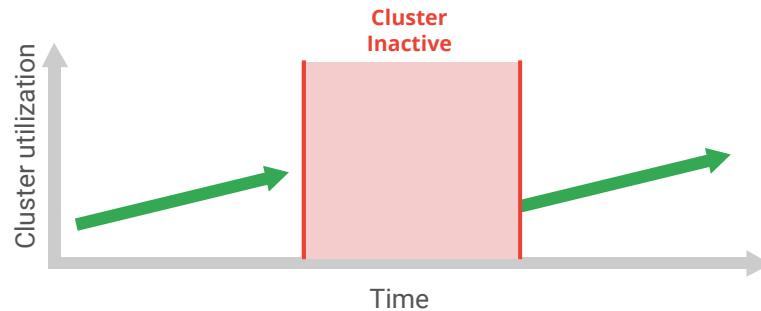
Scaling can take hours, days, or weeks to perform which may delay needed data processing

Scaling should be painless and fast



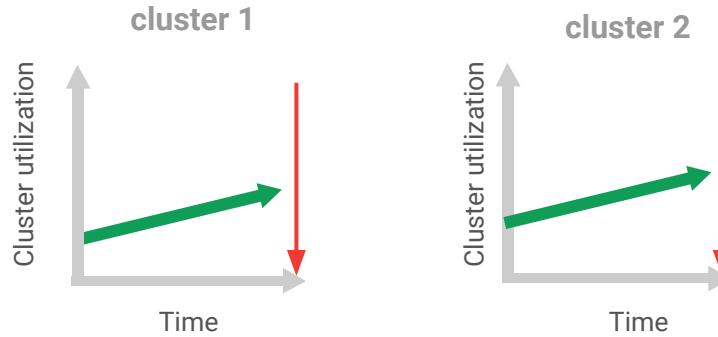
Things take seconds to minutes,
not hours or weeks.

You have to babysit utilization



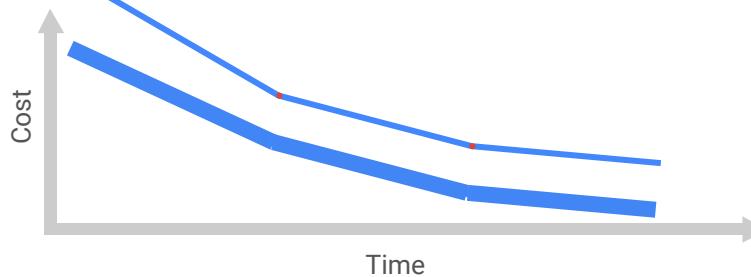
Requires effort to pack clusters
so the it does not have periods of
inactivity and wasted resources

Only use clusters when you need them



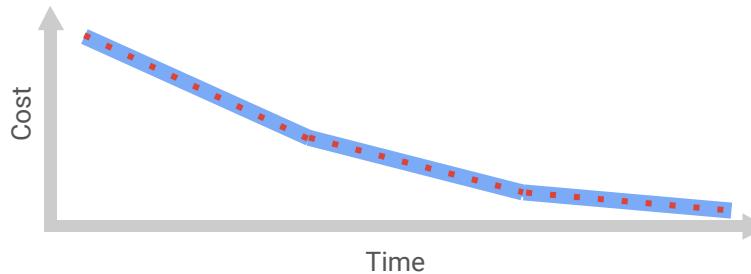
Be an expert with your data, not
your infrastructure

You are not paying for what you use



You are **paying** for more (spare) capacity than you **actually need** to process your data

Pay for exactly what you use

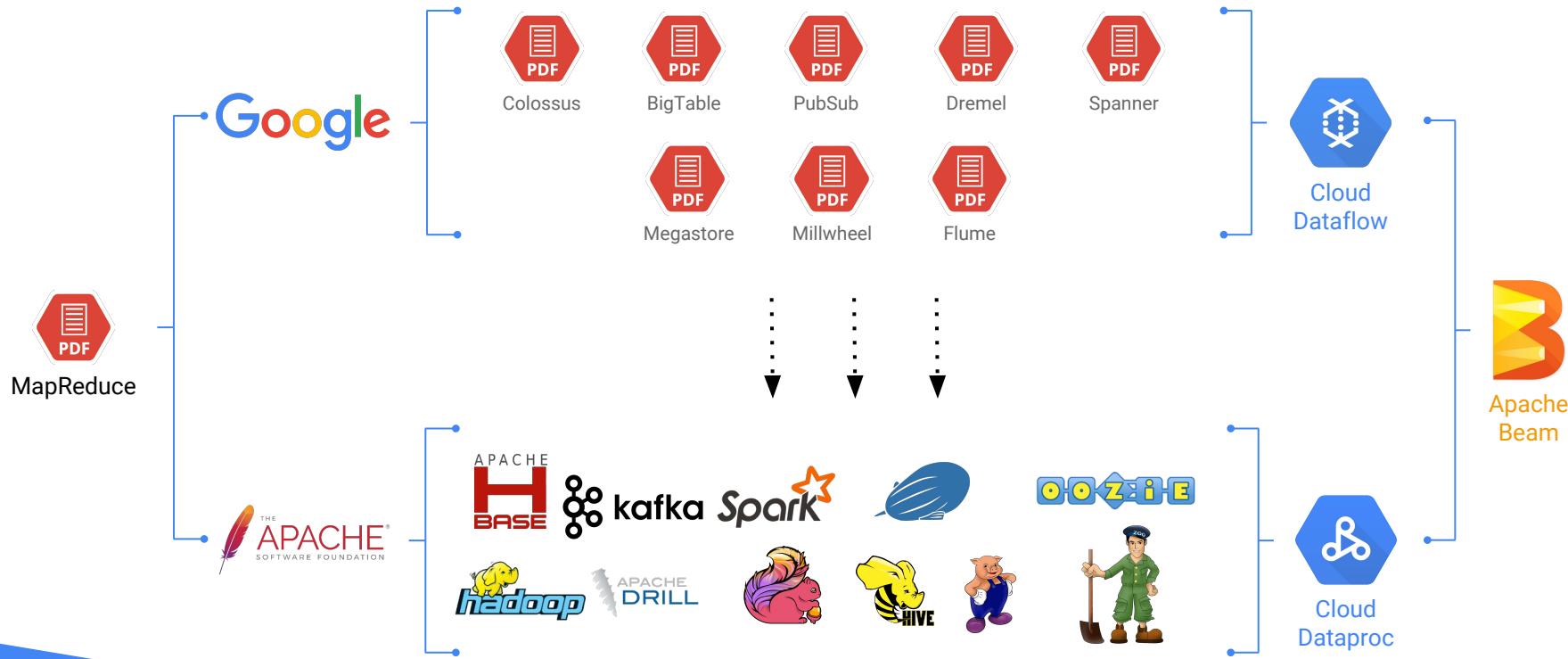


You only pay for resources when
you need them



Open source
on **Google Cloud**

Google is passionate about open source



What is Cloud Dataflow?

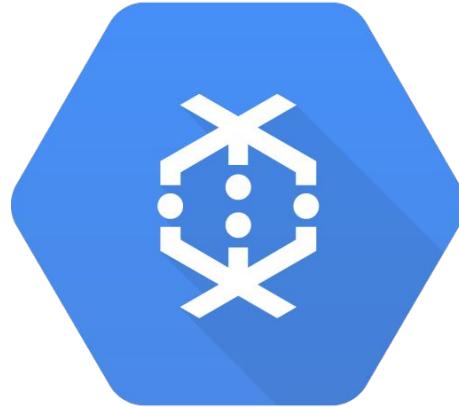
- Unified batch and streaming processing

- Fully managed, no-ops data processing

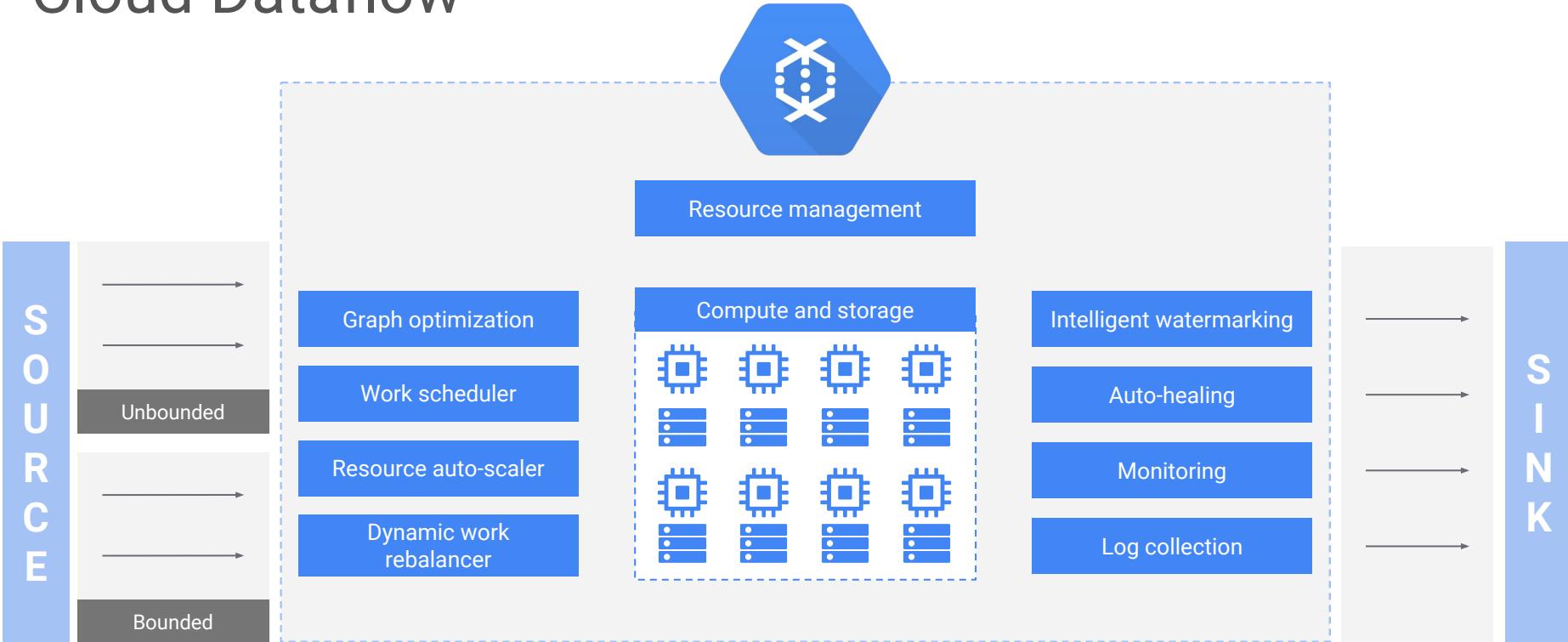
- Open source programming model



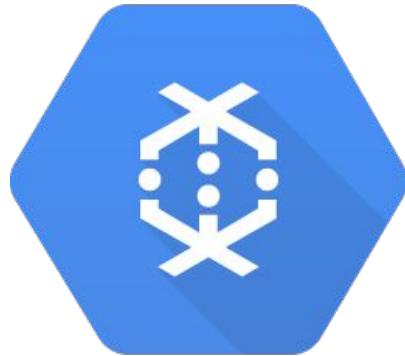
- Intelligently scales to millions of QPS



Cloud Dataflow



Cloud Dataproc offers a spectrum



Cloud Dataflow



Cloud Dataproc



Cloud Dataflow

Cloud Dataflow is a real-time data processing service for batch and stream data processing.

- Fully managed
- Unified programming model
- Integrated and open source
- Resource management
- Autoscaling
- Monitoring

What is Cloud Dataproc?

Google Cloud Dataproc is a fast, easy to use, low cost and fully-managed service, powered by Google Cloud Platform, that helps you take advantage of the Spark, Flink, and Hadoop ecosystem.

Google Cloud Dataproc

Fast

Things take seconds
to minutes, not
hours or weeks

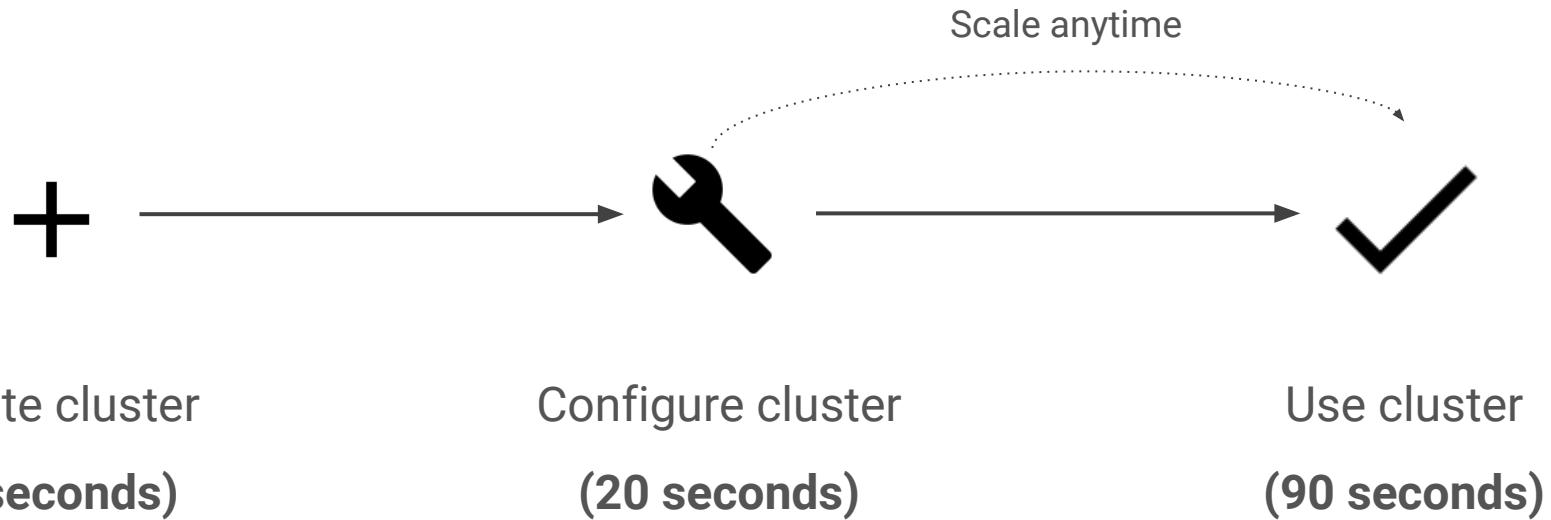
Easy

Be an expert with
your data, not your
data infrastructure

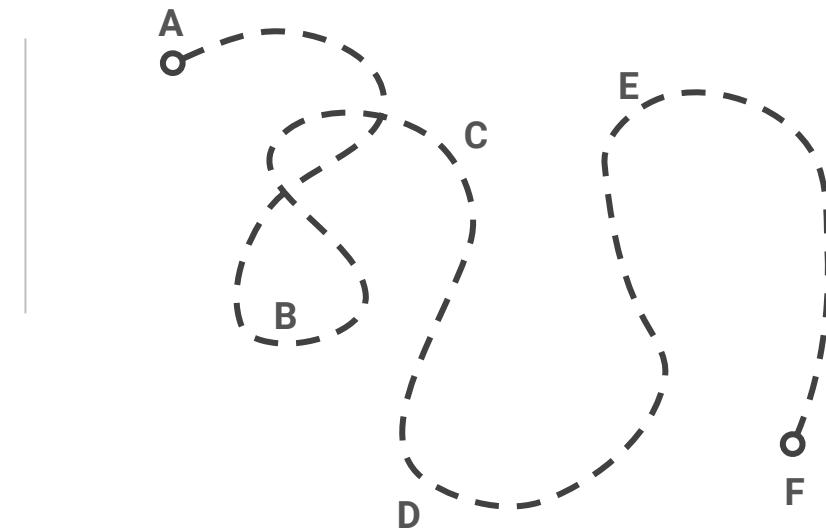
Cost-effective

Pay for exactly what
you use to process
your data, not more

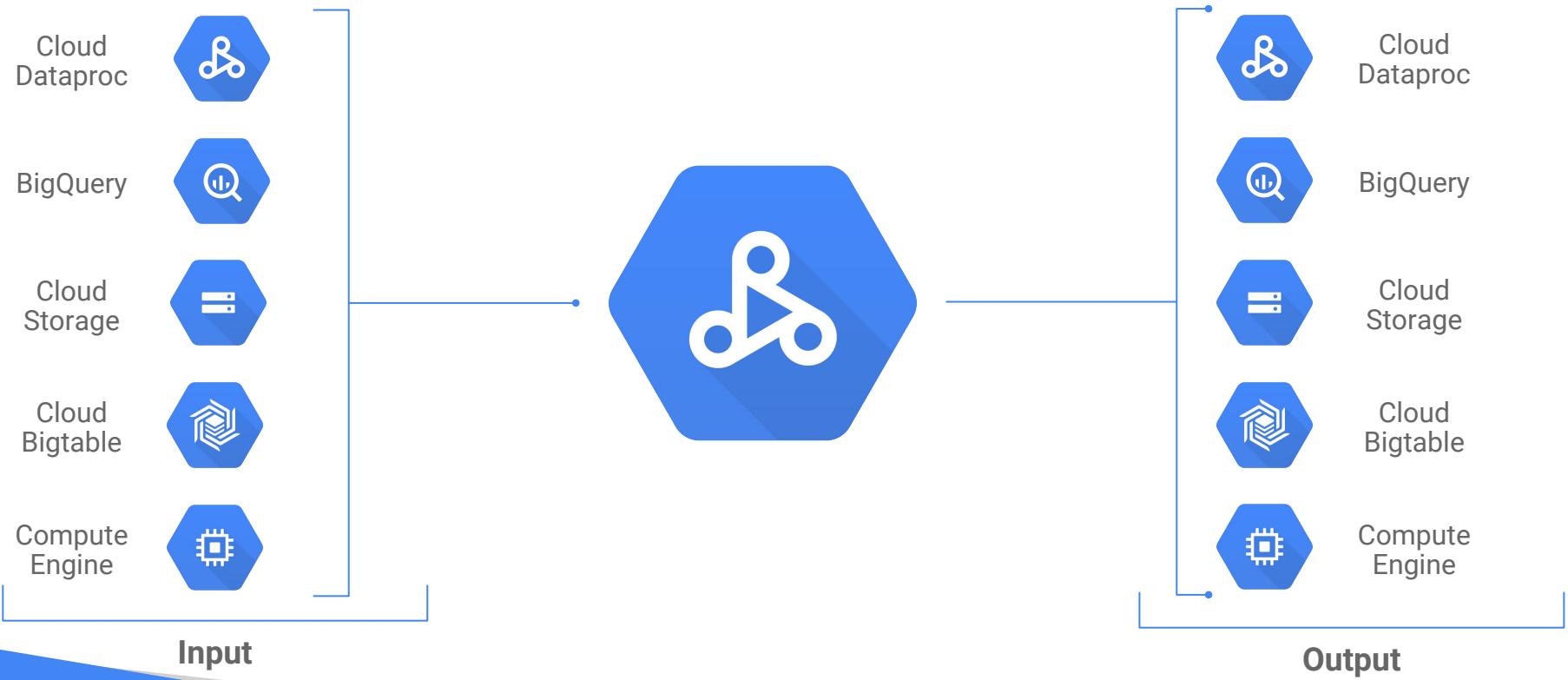
Cloud Dataproc clusters



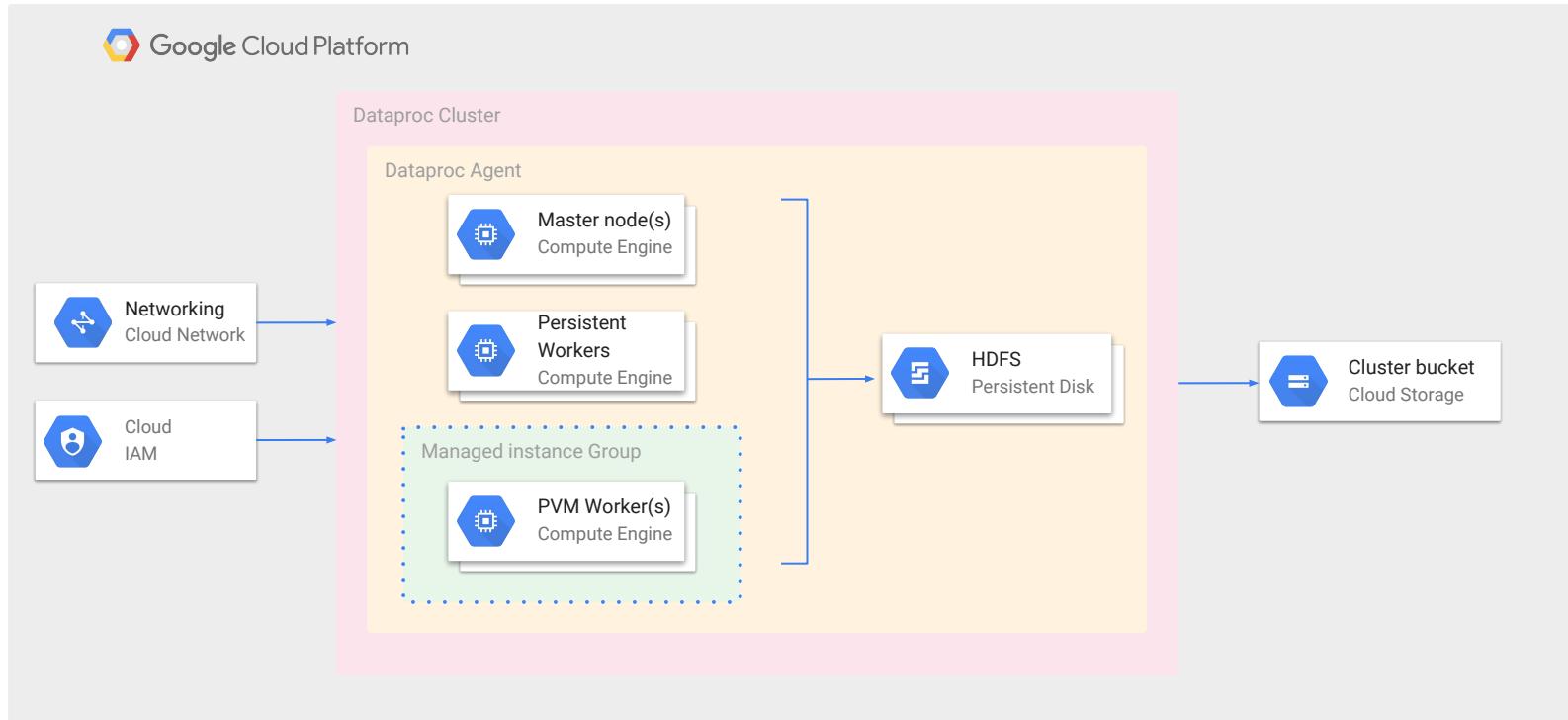
Cloud Dataproc offers a spectrum



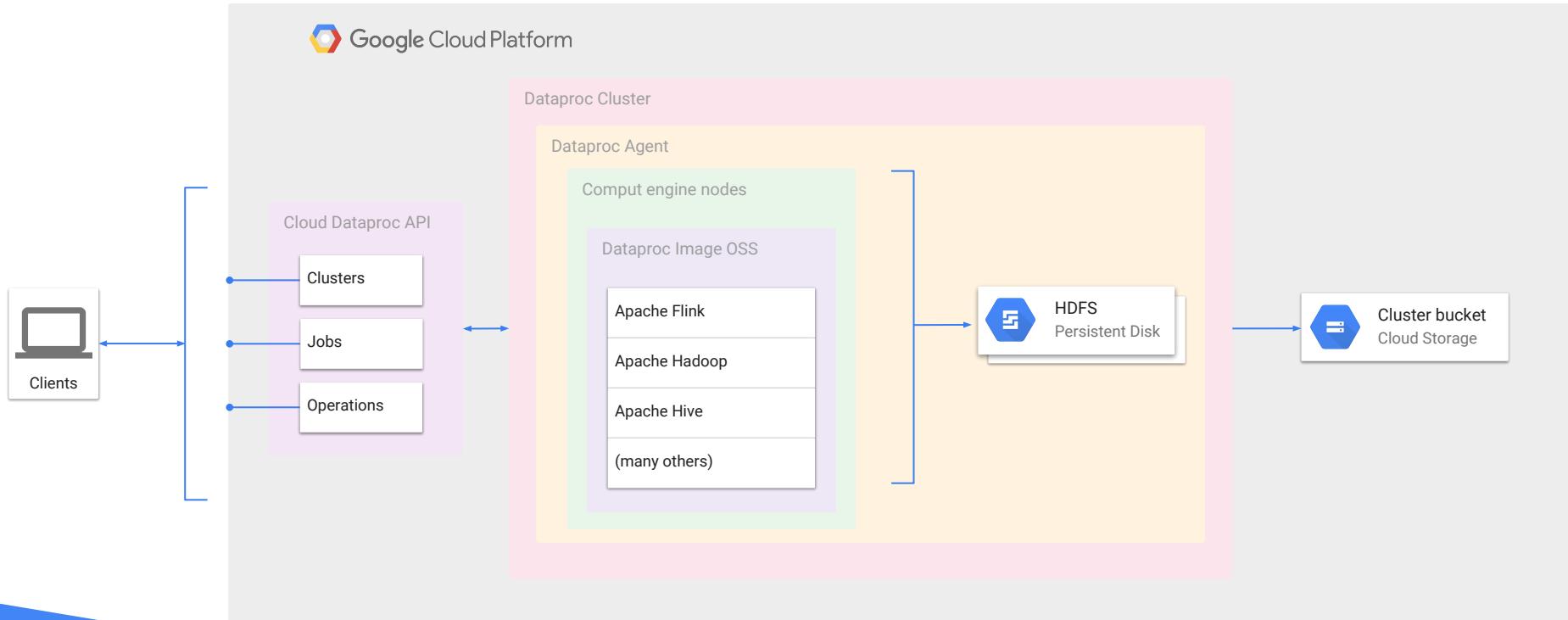
Connecting OSS to Cloud Platform



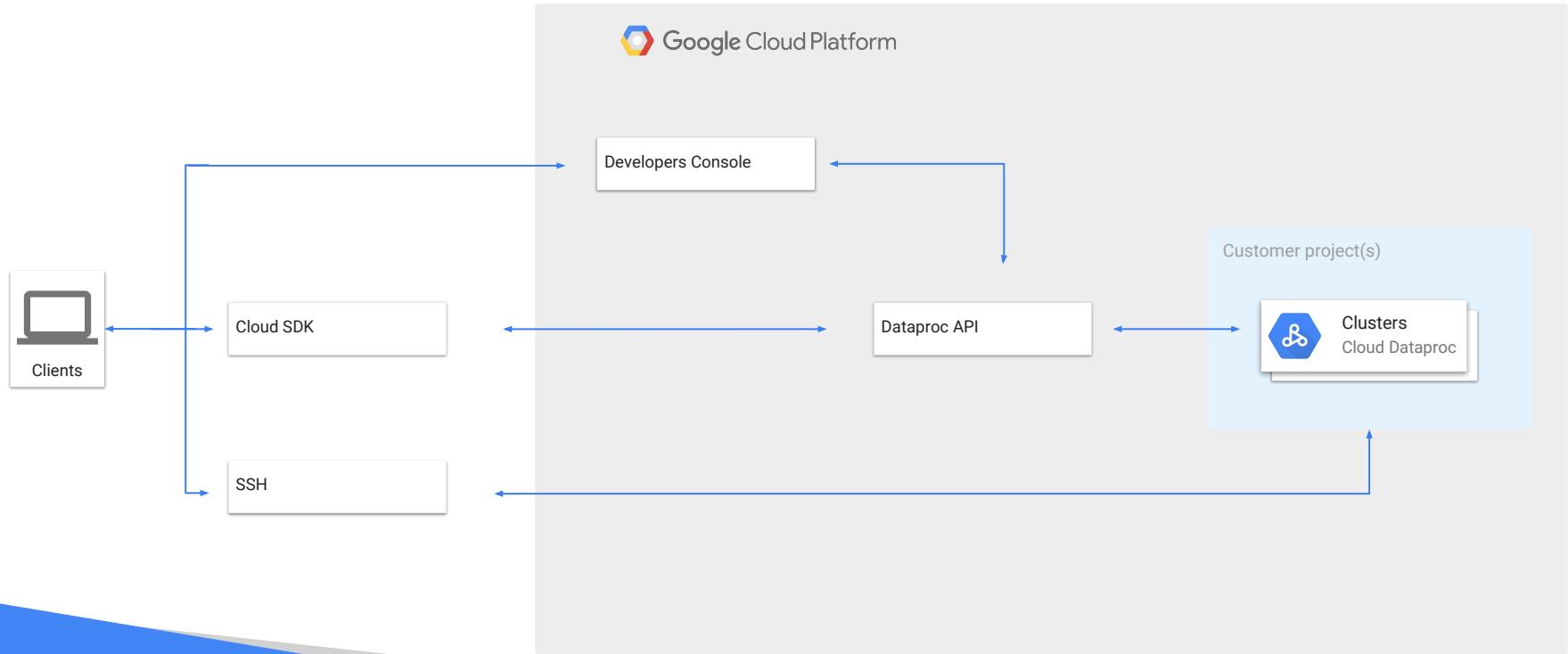
Cloud Dataproc - under the hood



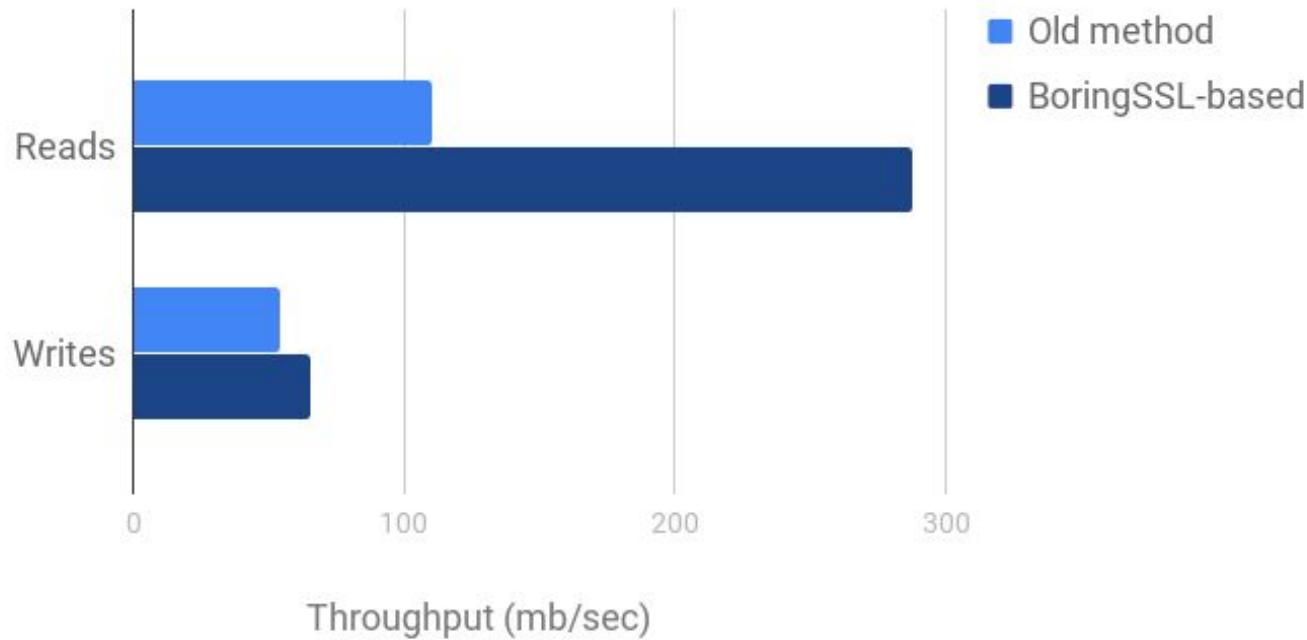
Cloud Dataproc - under the hood



Cloud Dataproc - under the hood



Cloud Storage performance (and improvement)



Cloud Dataproc **demo**

FEDERAL RESERVE NOTE

THE UNITED STATES OF AMERICA

THIS NOTE IS LEGAL TENDER
FOR ALL DEBTS, PUBLIC AND PRIVATE



H₃ 12

L 11180916 G

12 *Anna Escobedo Cabral*

Treasurer of the United States.



WASHINGTON

SERIES
2003
A

John W. Snow

Secretary of the Treasury.

ONE DOLLAR

12

FW H 57

L 11180916 G

WASHINGTON, D.C.



12

12

12

Example new features
and their impact

Restartable jobs (beta)

- Any job submitted through the Cloud Dataproc Jobs API can now be set to automatically restart on failure
- Very useful for both batch **and** streaming jobs. Jobs which checkpoint can also be automatically restarted
- Specified with the switch `--max_retries_per_hour` when using the Cloud SDK (`gcloud`) the `max_failures_per_hour` in the Jobs API

Clusters with GPUs (beta)

- Cloud Dataproc clusters support Compute Engine nodes with Nvidia Tesla K80 GPUs attached to them
- We expect GPU support will continue to grow in the open source data processing ecosystem throughout 2017
- Easily add GPUs to a Cloud Dataproc cluster with the switch `--master/worker_accelerator` with the Cloud SDK (`gcloud`)

Single-node clusters (beta)

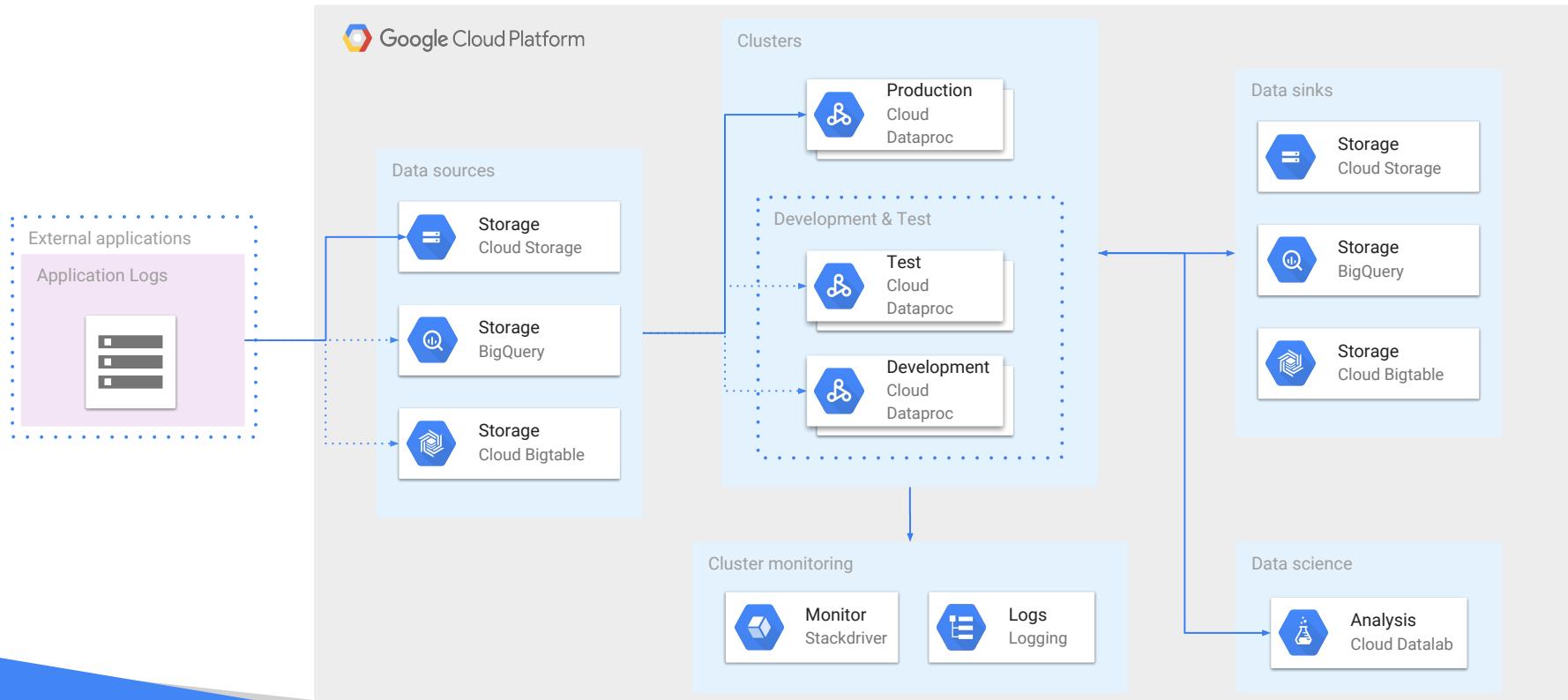
- Create a sandbox Cloud Dataproc “cluster” with only one node instead of the typical three node design (1 master, 2 workers)
- Great for lightweight data science, small-scale testing, proof of concept building, and education
- Use the `--single-node` argument in the Cloud SDK or select “Single node” when creating a cluster in the Google Cloud Console

Regional endpoints & private IP clusters (beta)

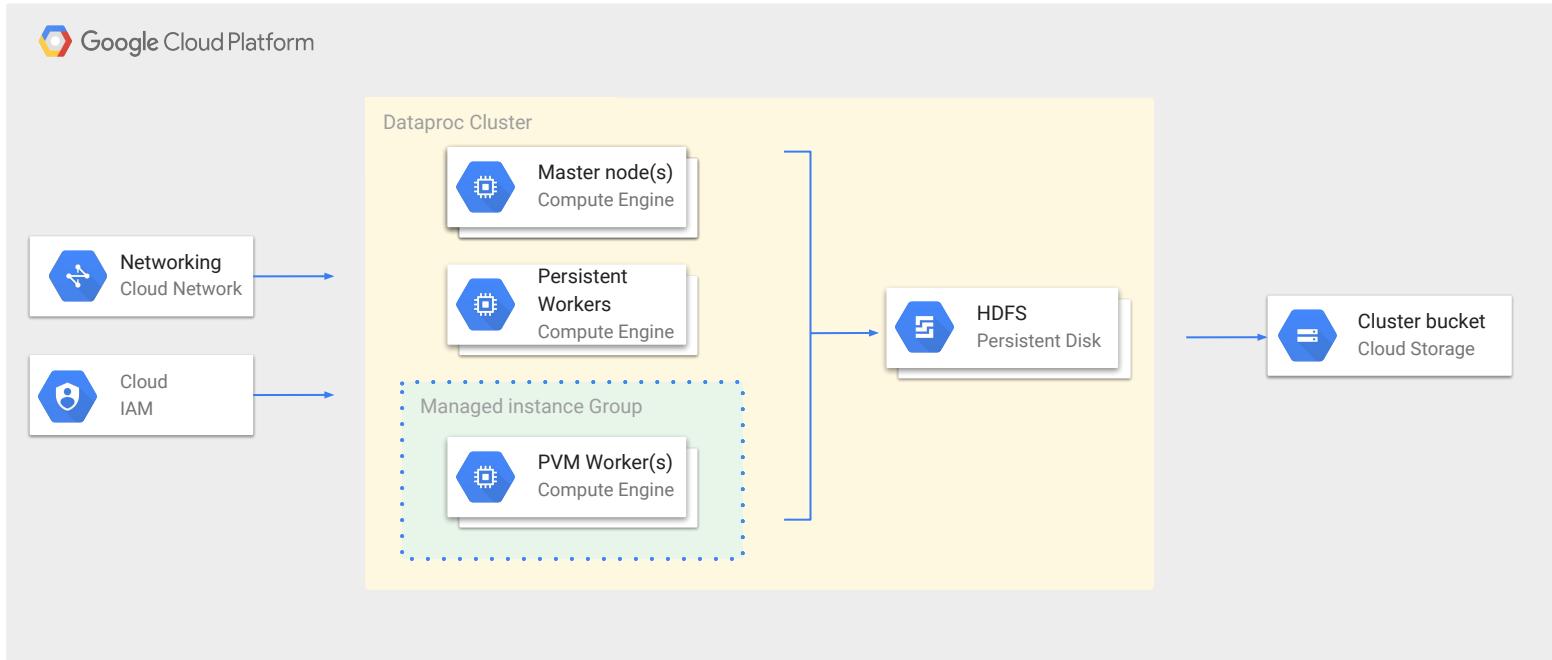
- Cloud Dataproc supports a “global” endpoint and “regional endpoints” in each Compute Engine region. This allows you to isolate Cloud Dataproc interactions to one specific region
- Traditionally clusters have needed a public IP attached to them. Cloud Dataproc now supports (easy to setup) “private IP only” clusters which do not require a public IP address on Compute Engine nodes

Cloud platform architecture concepts

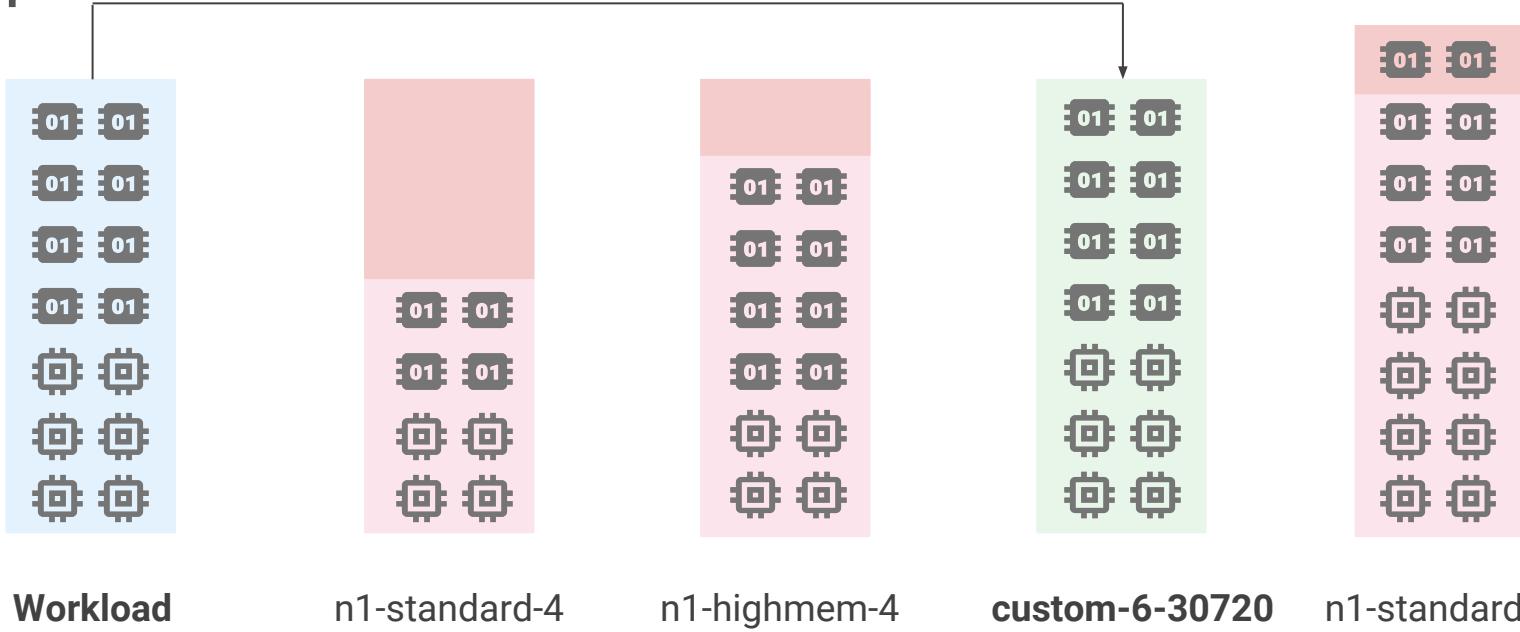
Disaggregation of storage and compute



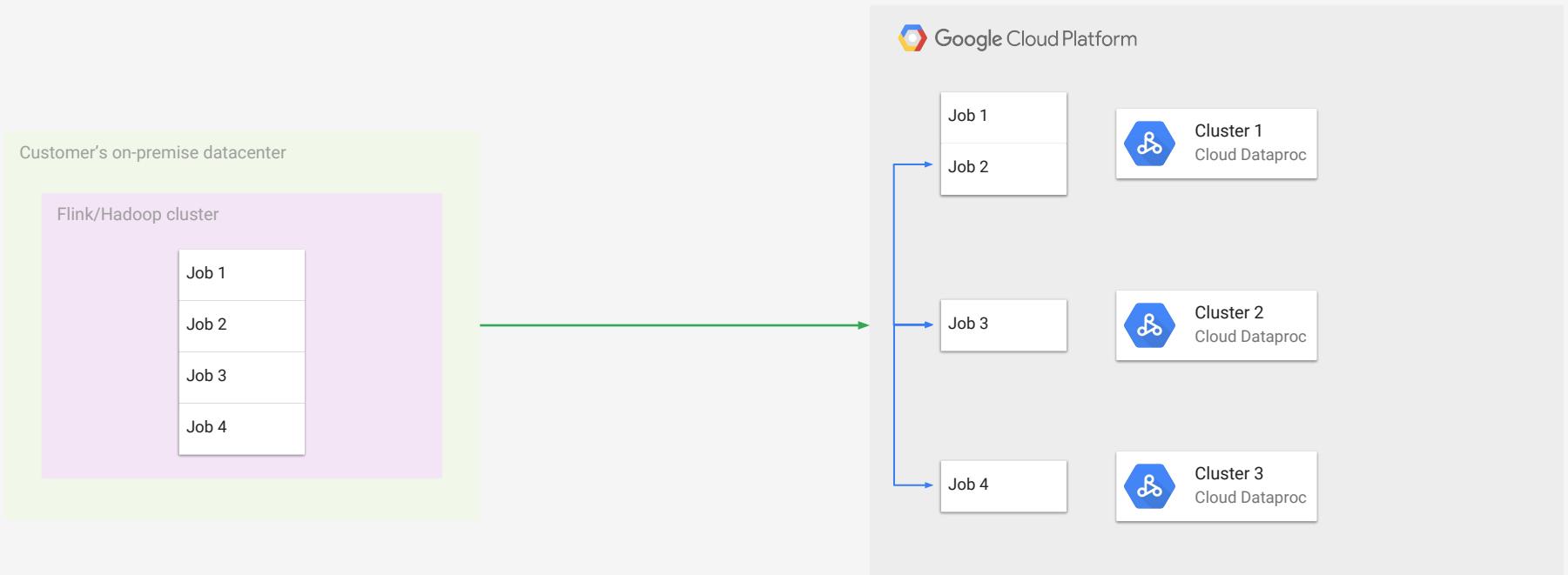
Cost savings through preemptible VMs



Right-sizing your hardware with custom machine types

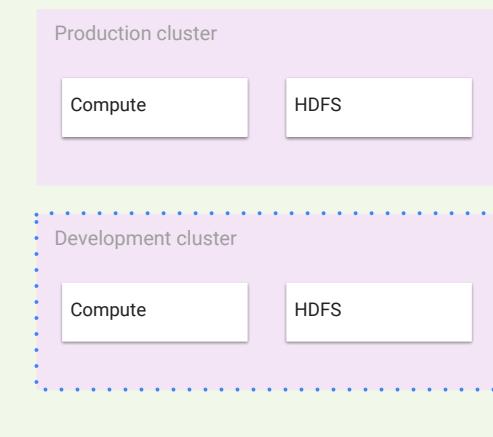


Split clusters and jobs

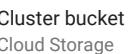
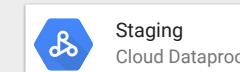
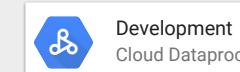
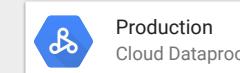


Separate development and production

Customer's on-premise datacenter



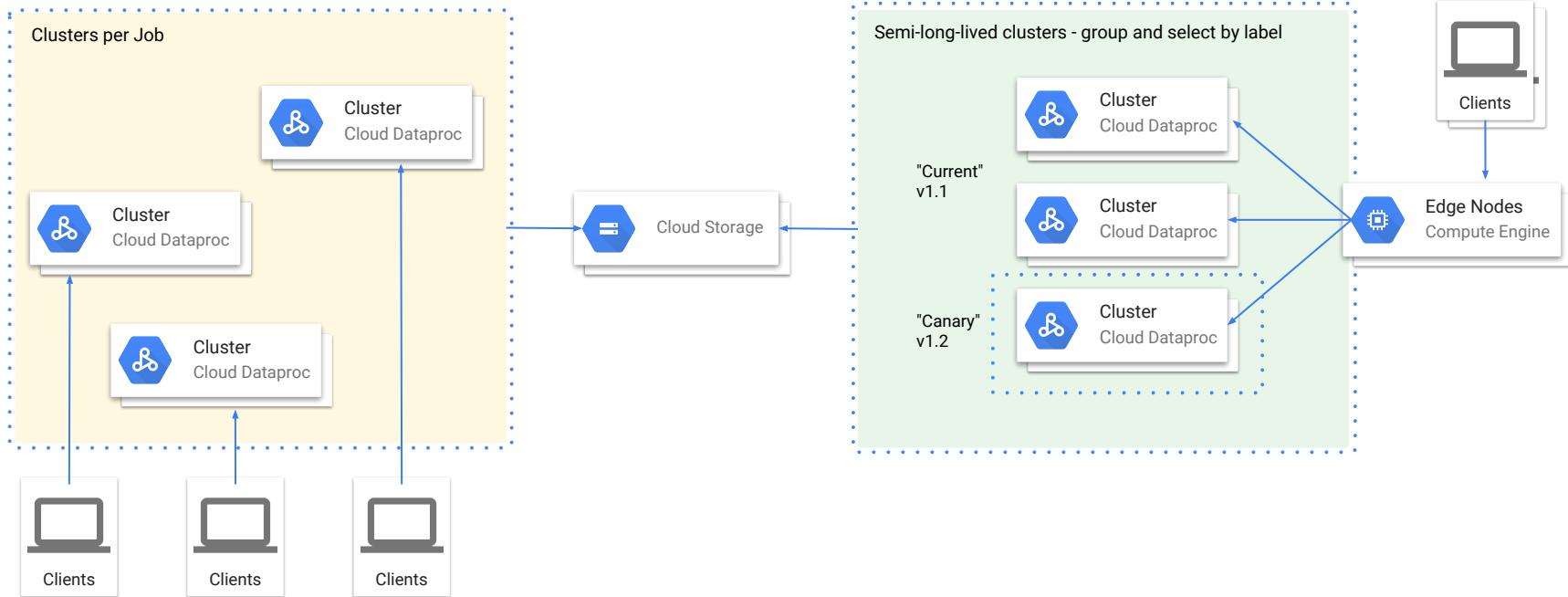
Google Cloud Platform



Read/write

Read only

Ephemeral and semi-long-lived clusters



Questions and **next steps**

Getting started

Codelabs - codelabs.developers.google.com/codelabs/cloud-dataproc-starter/

Cloud Dataproc quickstarts - cloud.google.com/dataproc/docs/quickstarts

Cloud Dataproc tutorials - cloud.google.com/dataproc/docs/tutorials

Cloud Dataproc initialization actions - github.com/GoogleCloudPlatform/dataproc-initialization-actions

Getting help

Cloud Dataproc documentation - cloud.google.com/dataproc/docs

Cloud Dataproc release notes - cloud.google.com/dataproc/docs

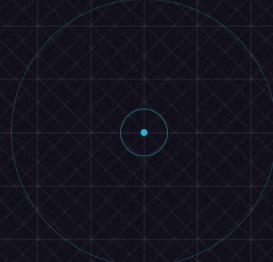
Stack Overflow - google-cloud-dataproc

Cloud Dataproc email discussion - cloud-dataproc-discuss@googlegroups.com

Google Cloud Support - cloud.google.com/support

Thank you

<https://goo.gl/Qyf5U7>



Near Real-Time Analytics

Challenges & Lessons at Uber Engineering



Quick Introduction



Chinmay Soman



@ChinmaySoman

- Staff Software Engineer @ Uber
- Tech Lead on Streaming Platform
- Background in distributed storage and filesystems
- Apache Samza Committer, PMC

Apache Kafka at Uber

Billion to Trillions

Messages/day

~ PB

bytes/day

Near Real-Time Analytics at Uber

Billions

Messages Processed / day

100s of TB - PB

Bytes Processed / day



What is near real-time ?



UBER



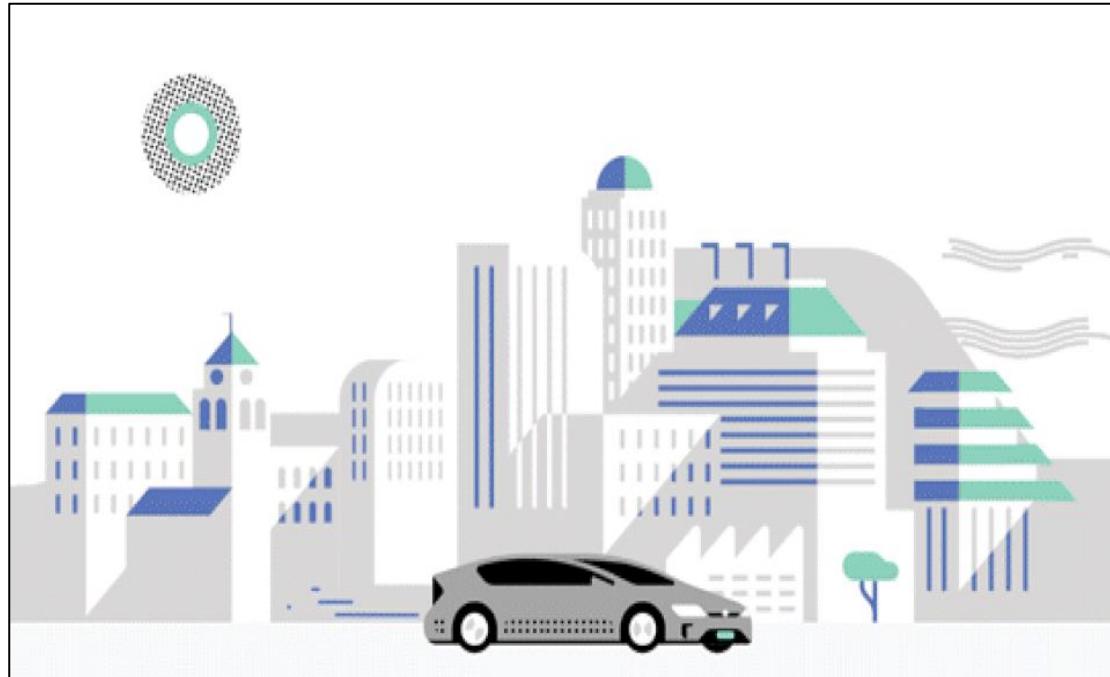
Agenda

- Evolution of Business Needs
- The case for SQL as building block
- New ecosystem using Flink
- The road ahead

Evolution of Business Needs



Case I - Growth Metrics



“How many **cars** are active right now ?”

“What **% of trips** have been **delayed** in the last 5 mins ?”

“What is the **% of Uber X trips** taken by **Android users** ?”



Events logged to Kafka



Rider eyeballs



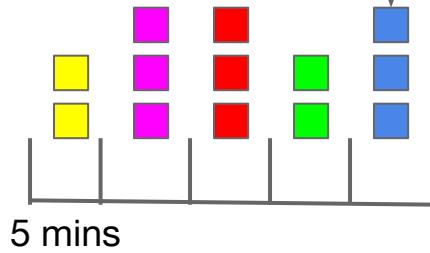
Trip updates



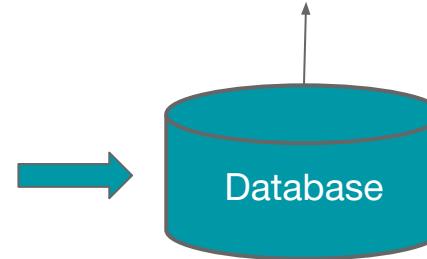
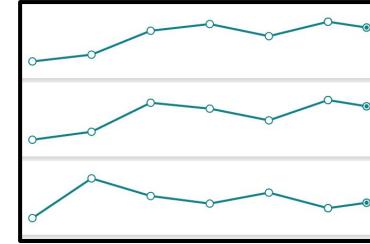


Artemis

Categorize by time



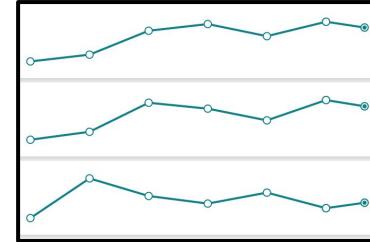
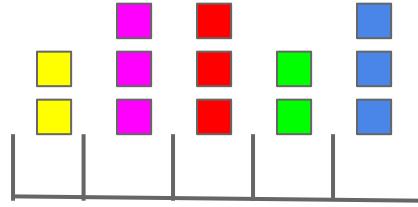
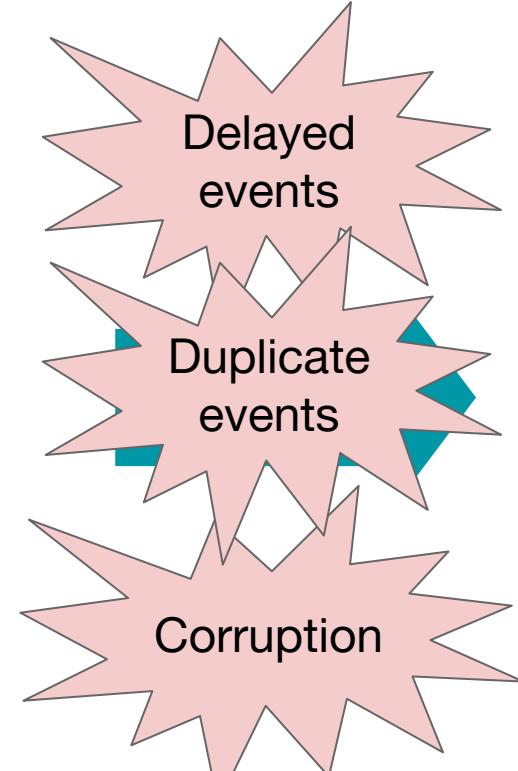
Aggregate value



UBER



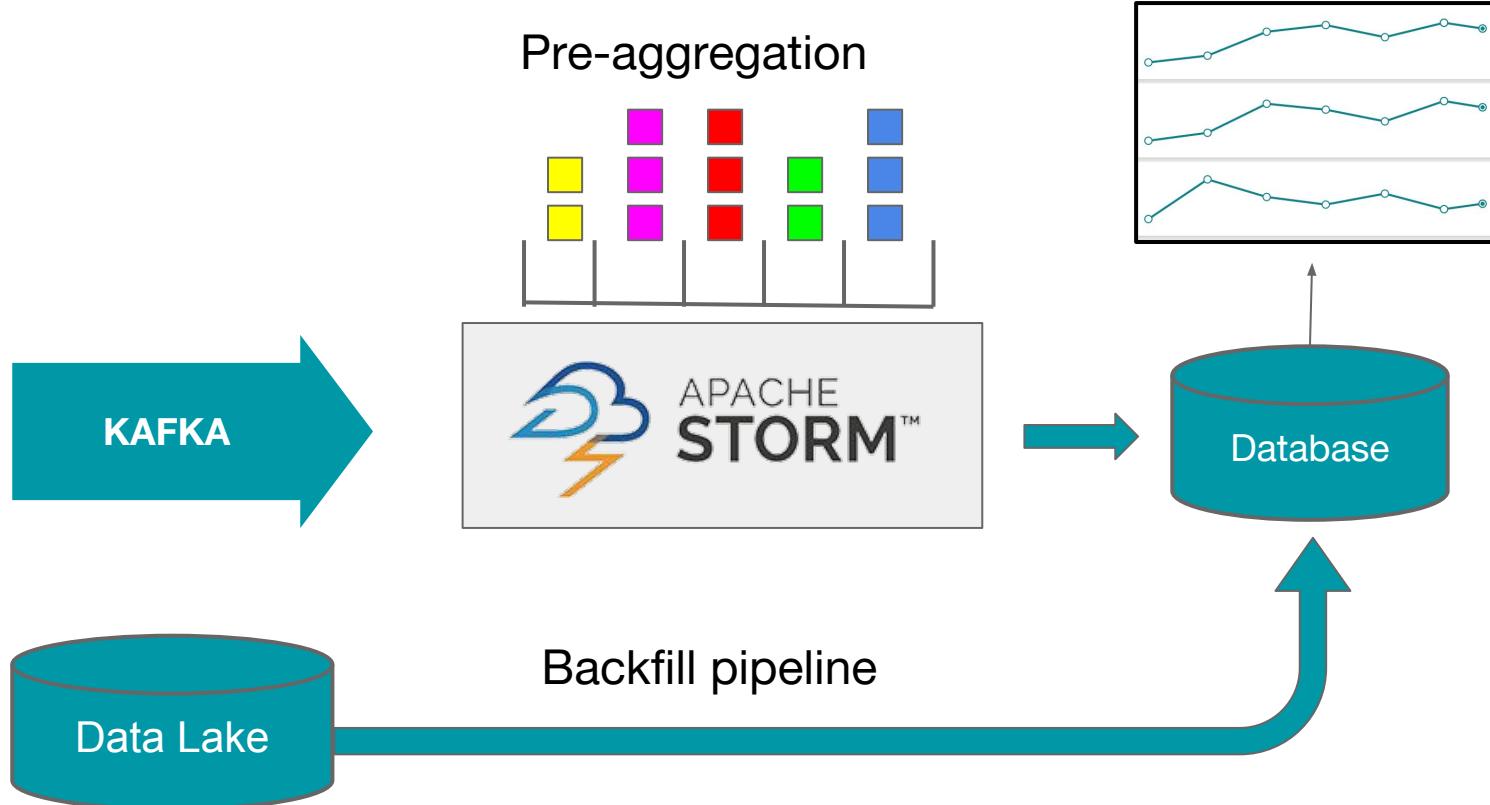
Artemis



UBER



Artemis

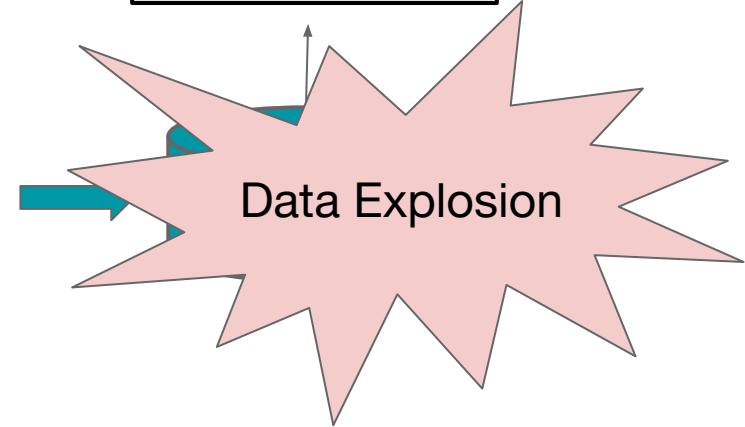
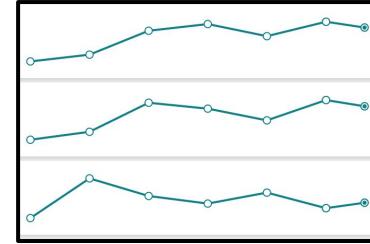
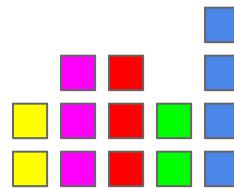


UBER



Artemis

Pre-aggregation
Per dimension



UBER

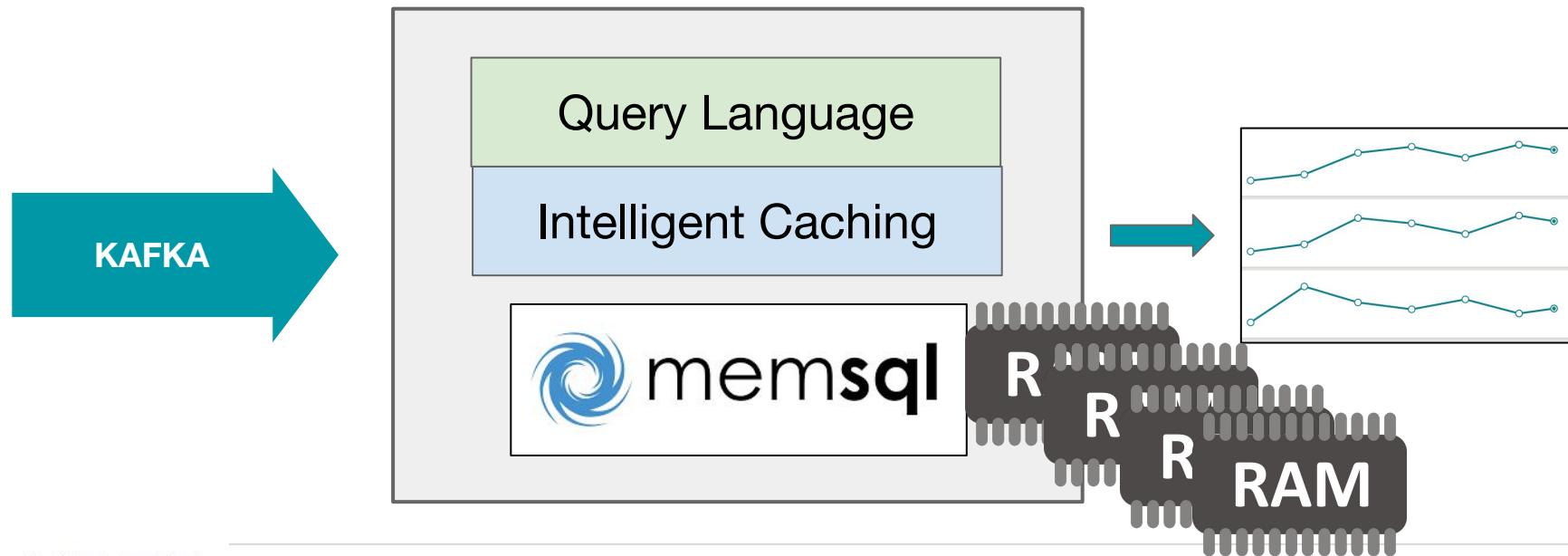


Apollo

✓ Fast

✓ Accurate

✓ Scalable

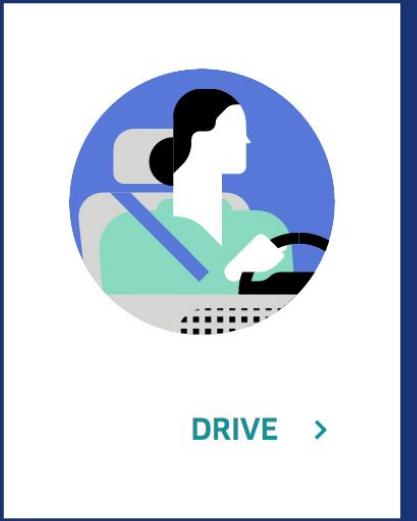
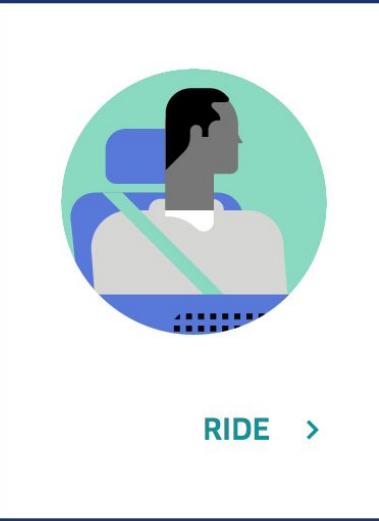


UBER



Case II - Event processing

Sign up to ride or drive



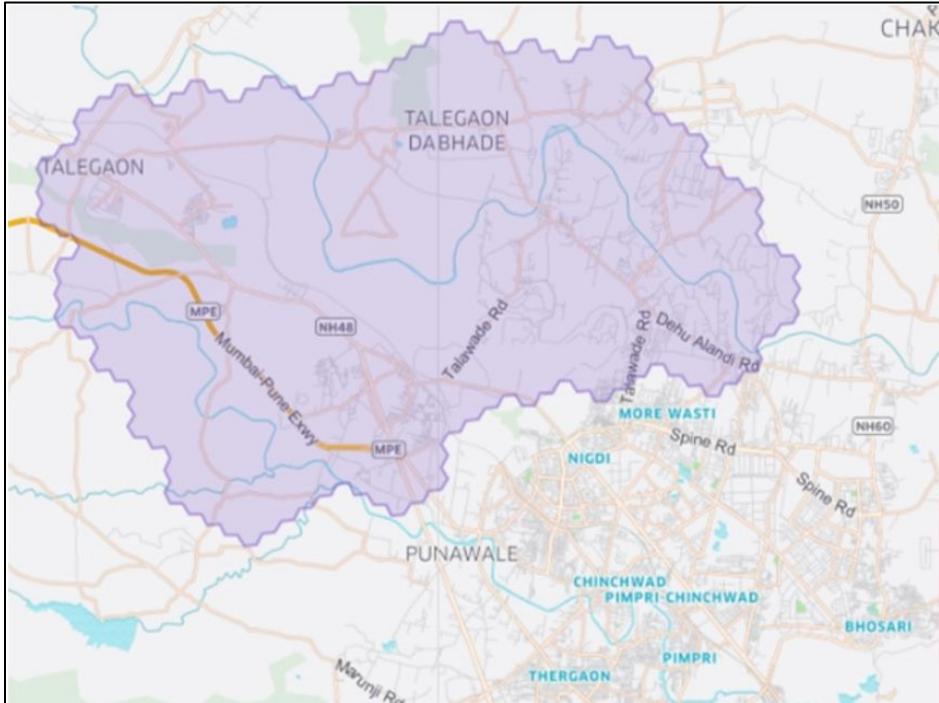
UBER

FRAUD

"If # Signups per device look suspicious -> Ban the driver/rider"



Case II - Event processing

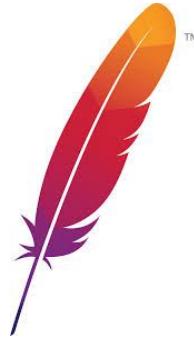


INTELLIGENT ALERTS

“Send me an alert if a leased vehicle **leaves a geo-fence**”



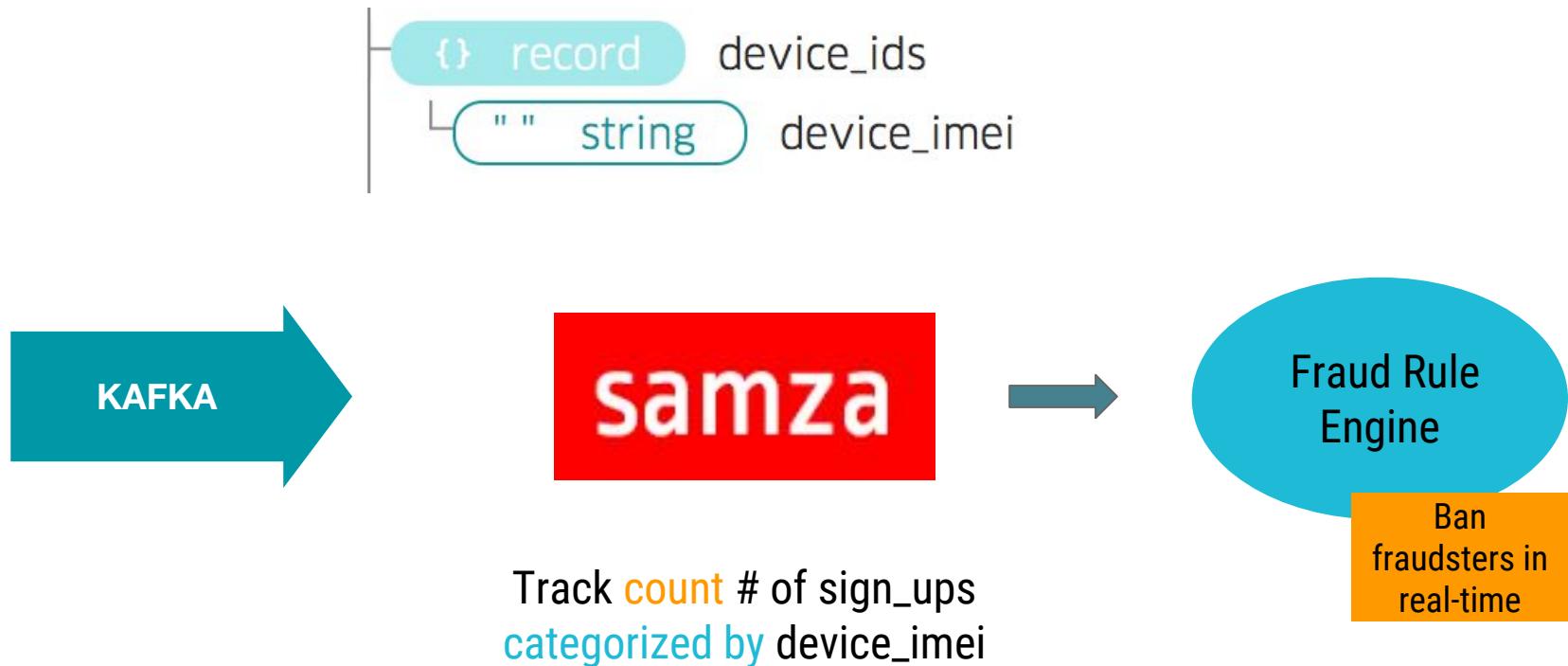
Athena platform using Apache Samza



- ★ Robust
- ★ Ease of operation
- ★ No backpressure issues
- ★ Built in state management

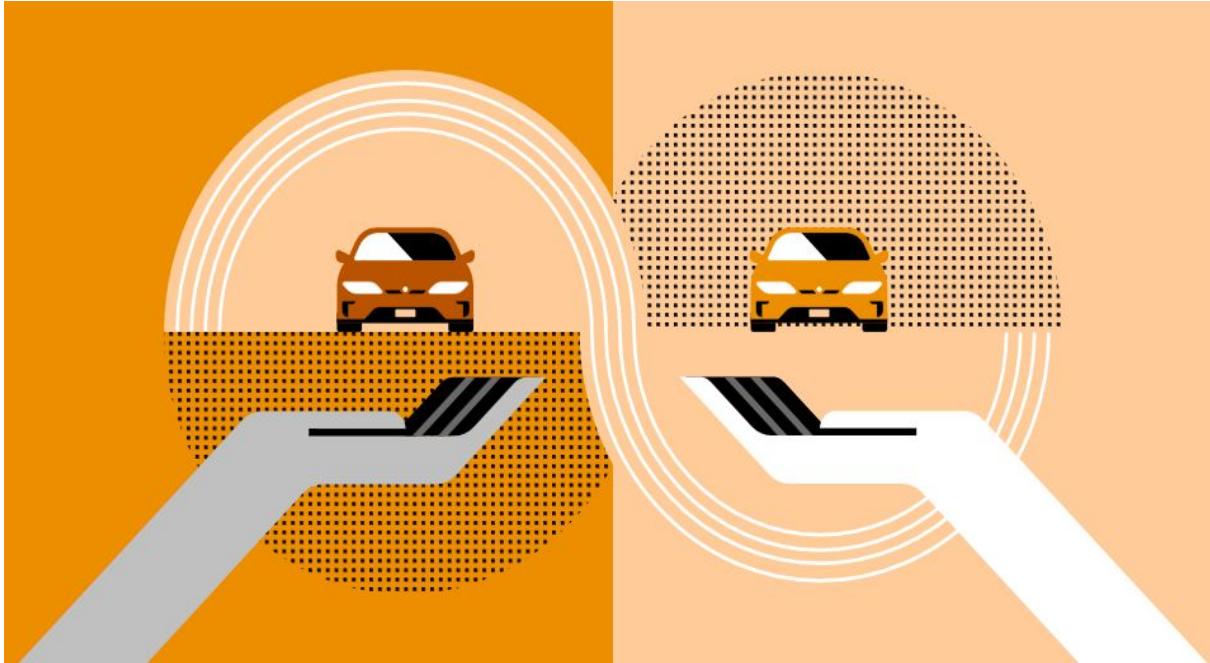


Event processing - Apache Samza





Case III - OLAP (OnLine Analytical Processing)

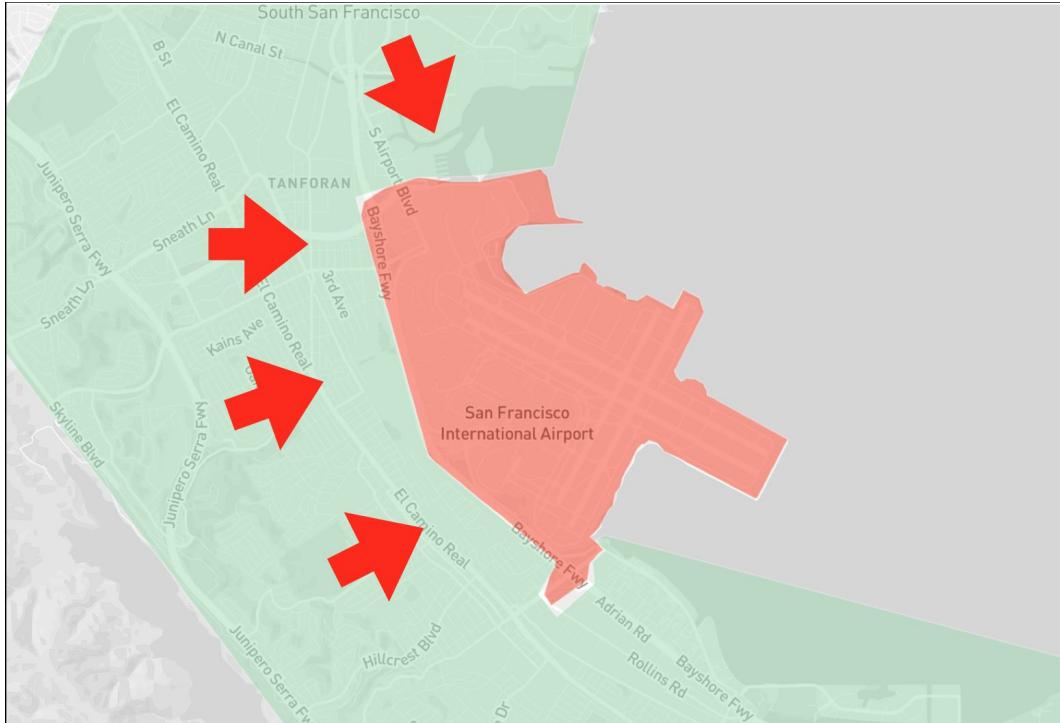


A / B Tests

See [progress](#) of
tests in [real-time](#)



Case III - OLAP use case



FORECASTING

“How many **first time riders** will be dropped off in a given geofence ?”



Our integrated platform



TM
samza

- Filter events
- Merge streams
- Decorate with external data

UBER

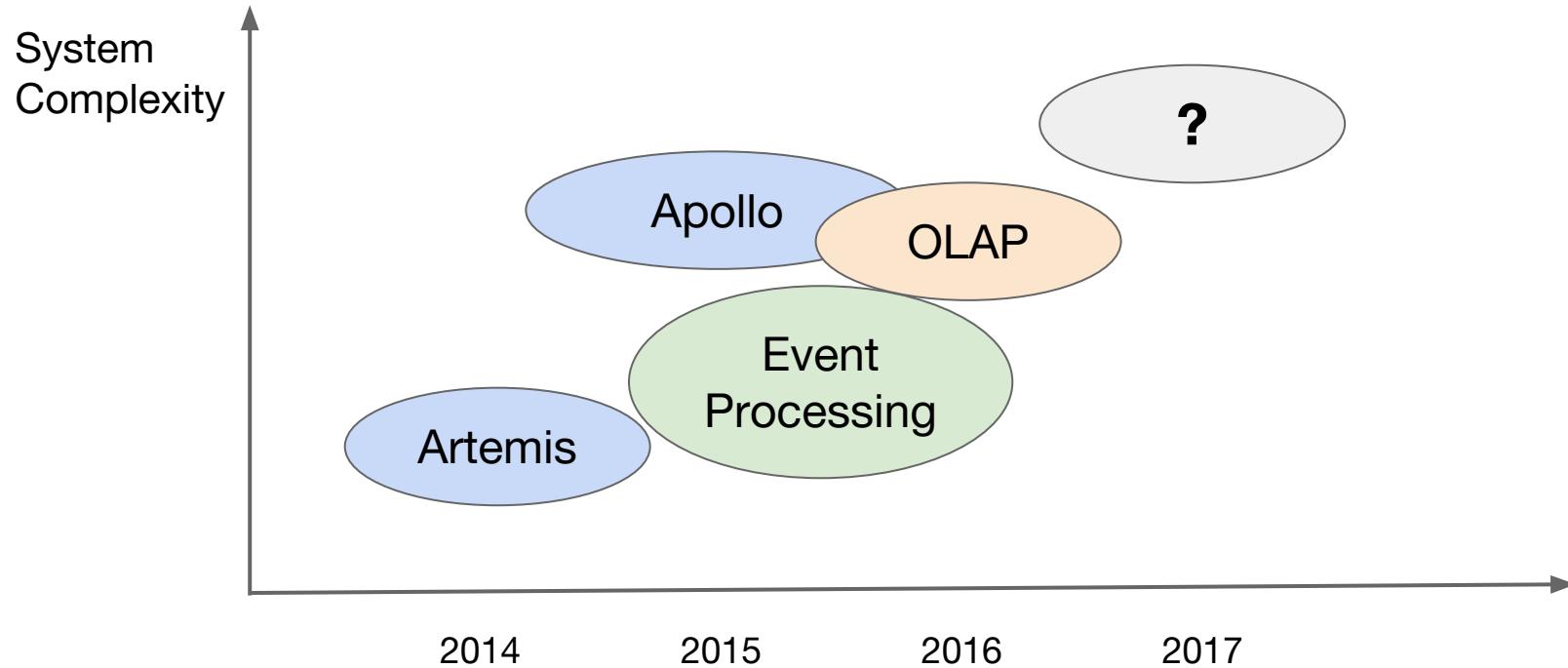
 **pinot**

 **elastic**

 **memsql**



Are we there yet ?





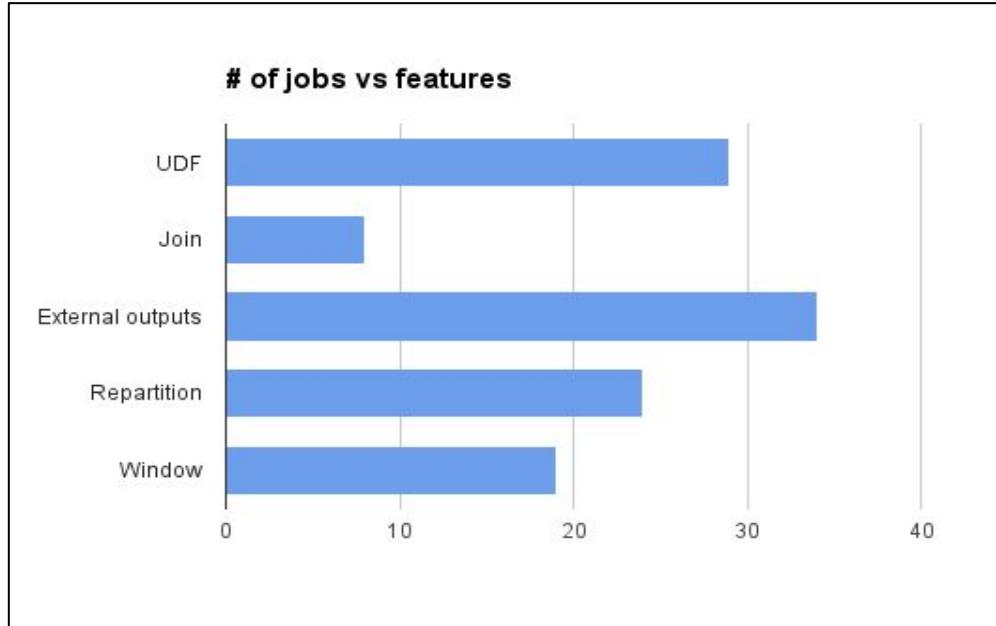
What's missing ?

- Cumbersome for data scientists / Ops people
- Redundant code
- Custom backfill pipelines

SQL as the building block



SQL + Stream Processing



70-80% of jobs can be implemented via SQL

SQL + Stream Processing: Powerful abstraction

Intelligent Promotions

Rule

“All trips worth $> 10\text{\$}$
in **San Francisco**
between Friday **5 pm**
and Sunday **9 pm**

Threshold

> 100

Action

“Give bonus of \$500”

UBER



SQL + Stream Processing: Powerful abstraction

Complicated rules

- “If number of hours online > 10 ...”
- “If amount earned > 700 in a given week, then ...”
- “If # uberPOOL rides >10, then ...”
- “If trip happens over some geo-fence 10 times in a given weekend, then ...”



SQL + Stream Processing: Powerful abstraction

Intelligent Promotions

Rule

Threshold

Action

```
select count(*) from hp_api_created_trips  
WHERE city_id = 1  
AND fare > 10  
AND request_at > 1491105600  
AND request_at <= 1491177600
```

> 100

trigger_payment()



SQL + Stream Processing: Powerful abstraction

Complicated rules

- “If number of hours online > 10 ...”
- “If amount earned > 700 in a given week, then ...”
- “If # Uber Pool rides >10, then ...”
- “If trip happens over some geo-fence 10 times in a given weekend, then ...”

What if we created **specific rules** for specific driver partners ?



SQL + Stream Processing: Powerful abstraction

Can be used for alerts as well:

“If a driver **X** is outside a **geofence**, then ...”

New eco-system: Athena X



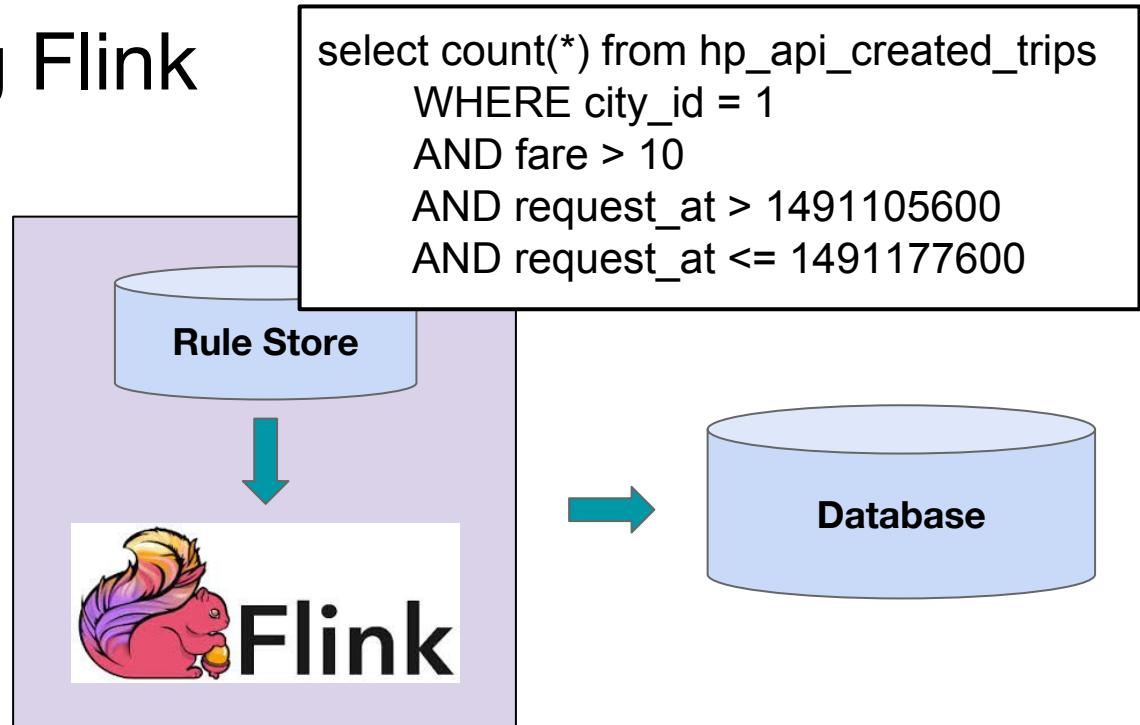
Enter Flink



- ★ Apache Calcite (SQL) Integration
- ★ Easy to manage and scale
- ★ No backpressure problem
- ★ Built in state management support
- ★ HDFS integration
- ★ Not dependent on Kafka

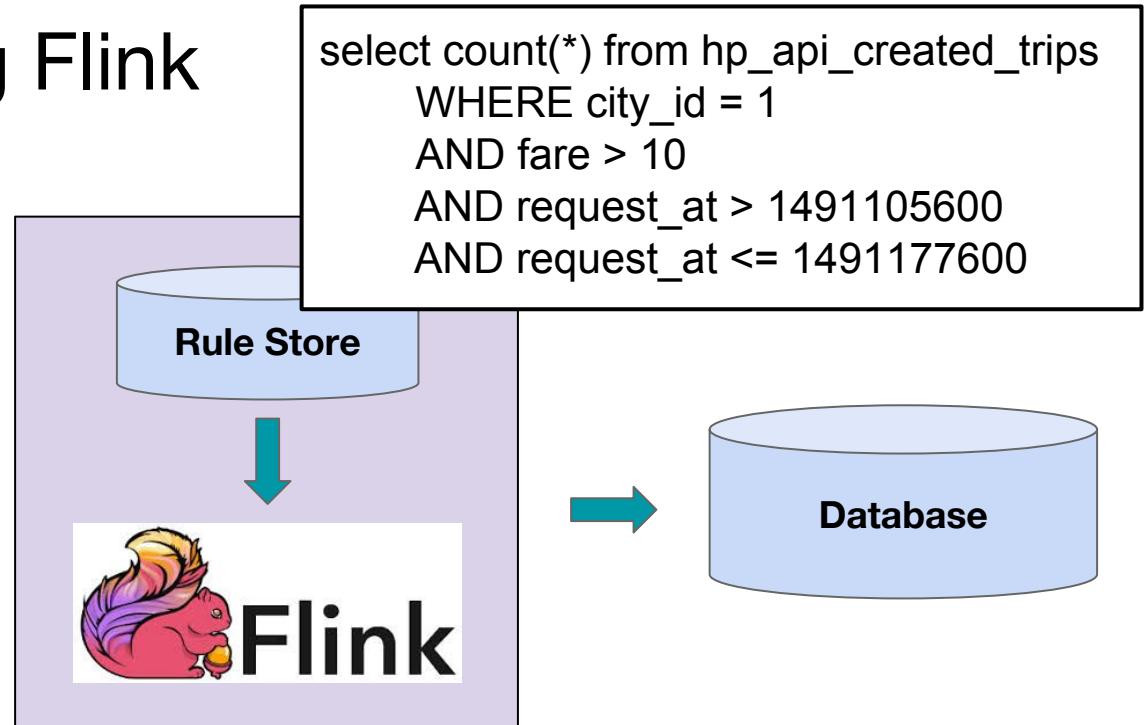


Promotions using Flink





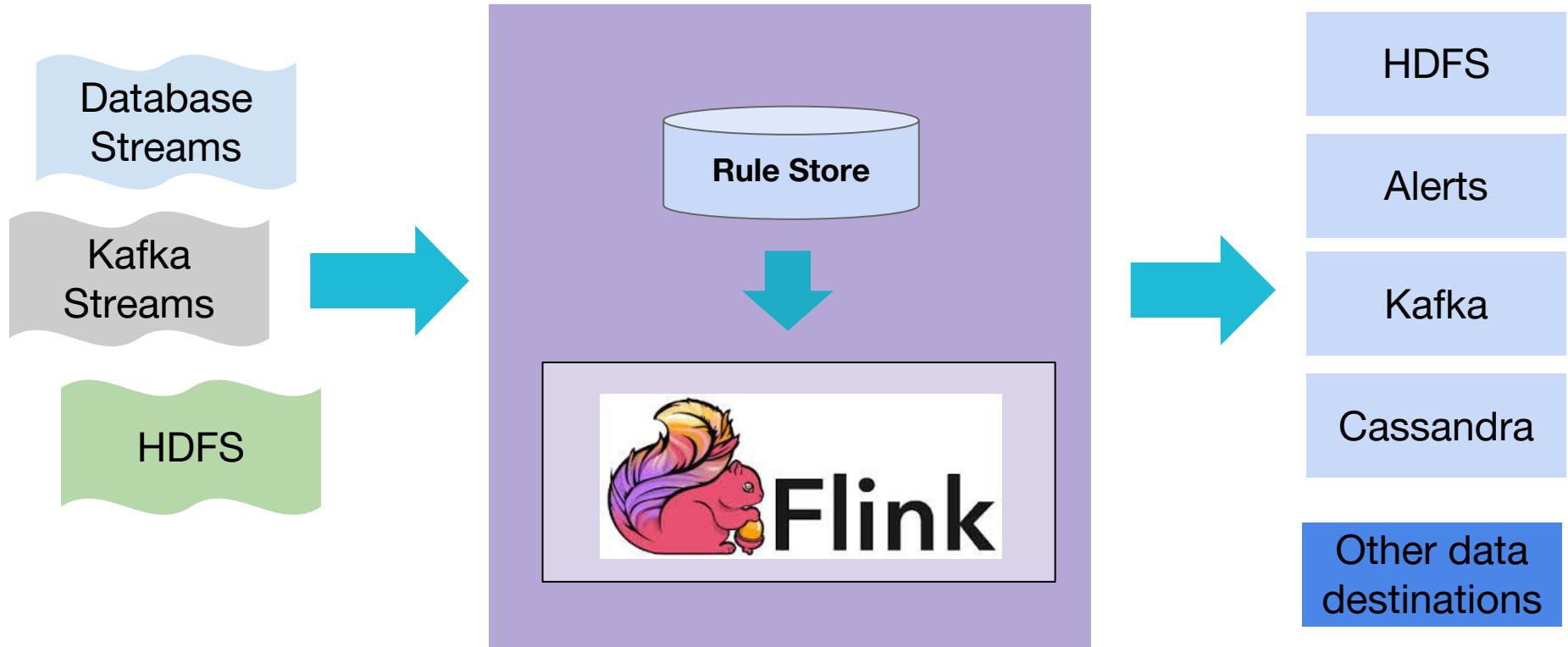
Promotions using Flink



UBER

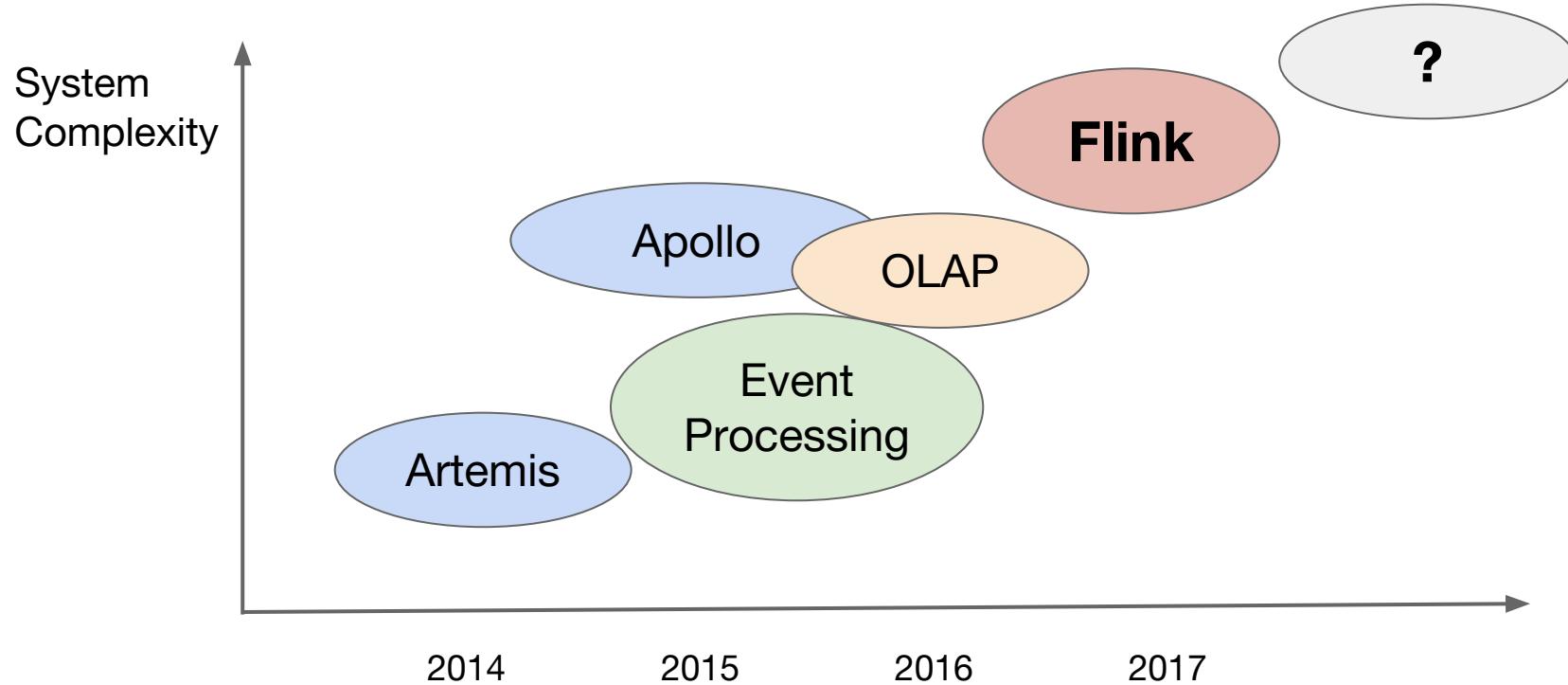


New Eco-system: Athena X





Are we there yet ?



The road ahead ...



Future Discussions

- To (Apache) Beam or not to Beam?
- Real-time Machine Learning
- Auto scaling

AthenaX - Flink deep dive

Haohui Mai
Bill Liu

(11:45 am)

Thank you

For more: eng.uber.com
Twitter: @UberEng



No shard left behind

Dynamic Work Rebalancing
and other adaptive features in
Apache Beam



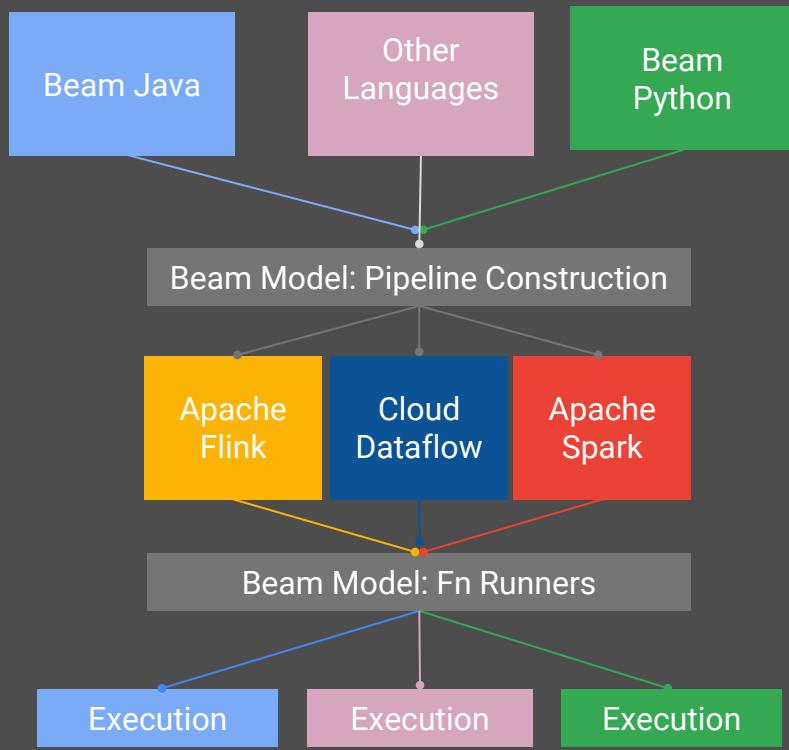
Malo Denielou (malo@google.com)



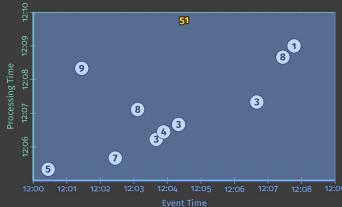
Apache Beam is a **unified** programming model designed to provide **efficient** and **portable** data processing pipelines.

Apache Beam

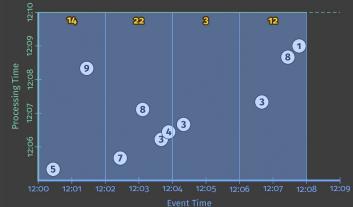
1. The Beam Programming Model
2. SDKs for writing Beam pipelines -- Java/Python/...
3. Runners for existing distributed processing backends
 - o Apache Flink
 - o Apache Spark
 - o Apache Apex
 - o Dataflow
 - o Direct runner (for testing)



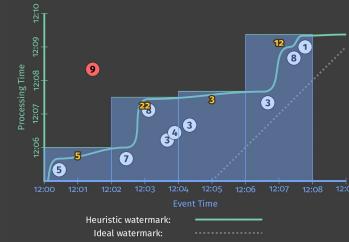
Apache Beam use cases



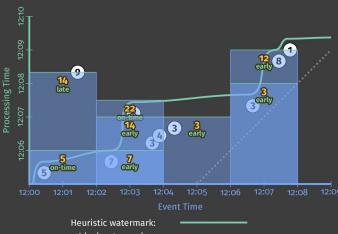
1. Classic Batch



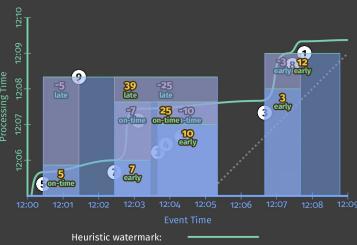
2. Batch with Fixed Windows



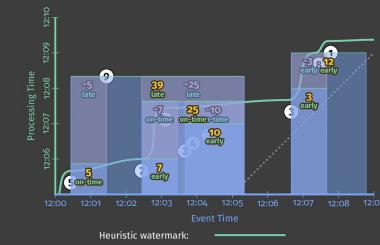
3. Streaming



4. Streaming with Speculative + Late Data

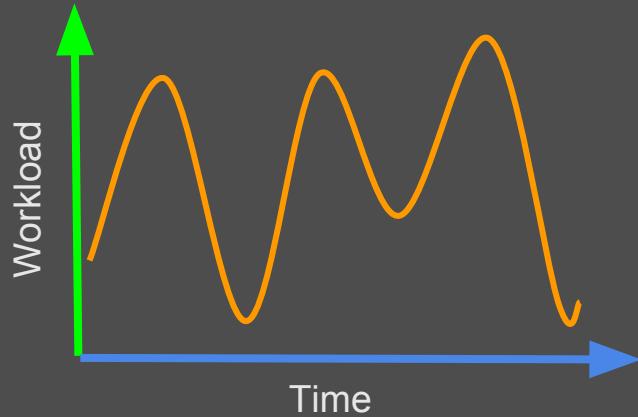


5. Streaming With Retractions

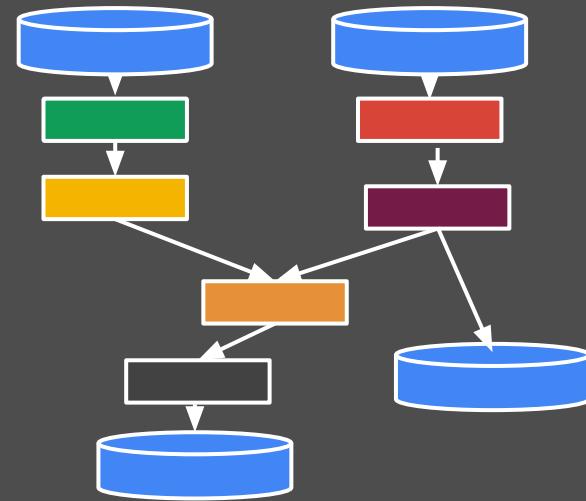


6. Streaming With Sessions

Data processing for realistic workloads

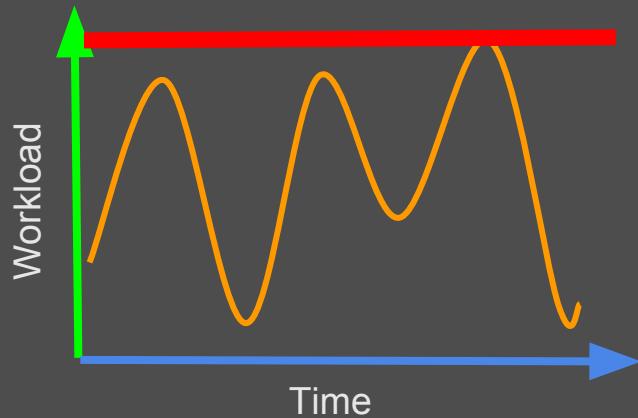


Streaming pipelines have variable input

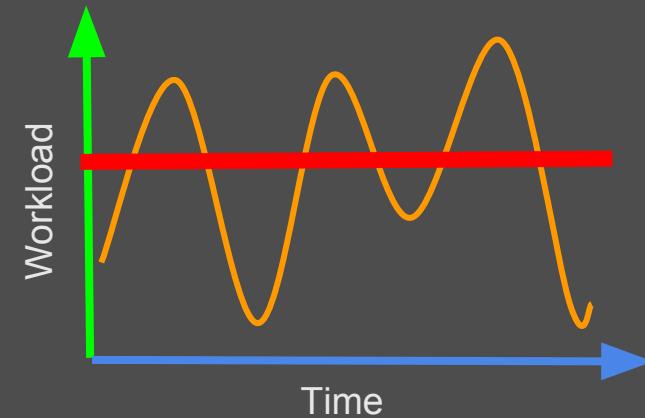


Batch pipelines have stages of different sizes

The curse of configuration



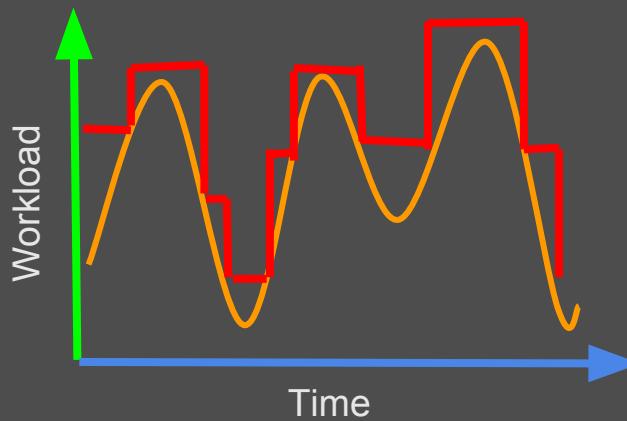
Over-provisioning resources?



Under-provisioning on purpose?

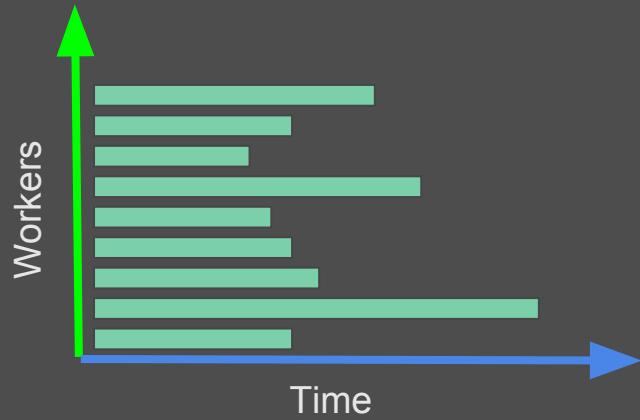
A considerable effort is spent to finely tune all the parameters of the jobs.

Ideal case



A system that adapts.

The straggler problem in batch



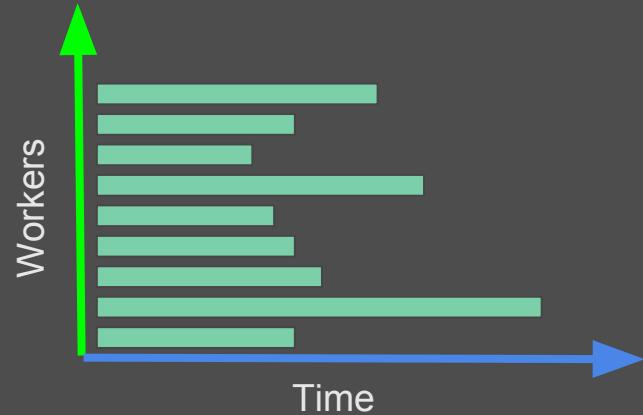
Tasks do not finish evenly on the workers.

- Data is not evenly distributed among tasks
- Processing time is uneven between tasks
- Runtime constraints

Effects are cumulative per stage!

Common straggler mitigation techniques

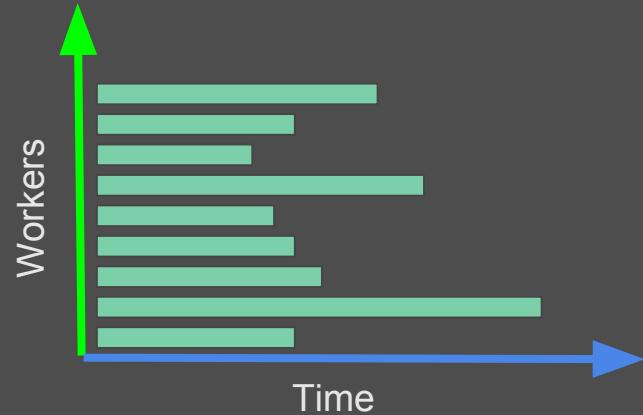
- Split files into equal sizes?
- Pre-emptively over split?
- Detect slow workers and reexecute?
- Sample the data and split based on partial execution



All have major costs, but do not solve completely the problem.

Common straggler mitigation techniques

- Split files into equal sizes?
- Pre-emptively over split?
- Detect slow workers and reexecute?
- Sample the data and split based on partial execution



All have major costs, but do not solve completely the problem.

« The most straightforward way to tune the number of partitions is experimentation:
Look at the number of partitions in the parent RDD and then keep multiplying that
by 1.5 until performance stops improving. »

From [blog]how-to-tune-your-apache-spark-jobs

No amount of upfront heuristic tuning (be it manual or automatic) is enough to guarantee good performance: the **system will always hit unpredictable situations** at run-time.

A system that's able to **dynamically adapt and get out of a bad situation** is much more powerful than one that **heuristically hopes to avoid** getting into it.

Fine-tuning execution parameters goes against having a truly **portable** and **unified** programming environment.



Beam abstractions empower runners

A **bundle** is group of elements of a PCollection processed and committed together.

APIs (ParDo/DoFn):

- `setup()`
- `startBundle()`
- `processElement()` n times
- `finishBundle()`
- `teardown()`

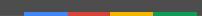
Streaming runner:

- **small bundles**, low-latency **pipelining** across stages, **overhead** of frequent commits.

Classic batch runner:

- **large bundles**, fewer **large commits**, more **efficient, long synchronous stages**.

Other runner strategies may strike a different balance.



Beam abstractions empower runners

Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"

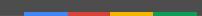
→ **user** decides

"Read from this source"

→ **runner** decides

APIs for portable Sources:

- long **getEstimatedSize()**
- List<Source> **splitIntoBundles(size)**



Beam abstractions empower runners

Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"

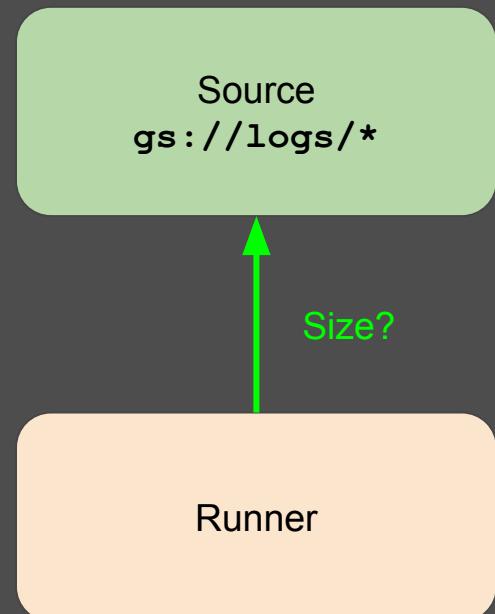
→ **user** decides

"Read from this source"

→ **runner** decides

APIs:

- long **getEstimatedSize()**
- List<Source> **splitIntoBundles(size)**



Beam abstractions empower runners

Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"

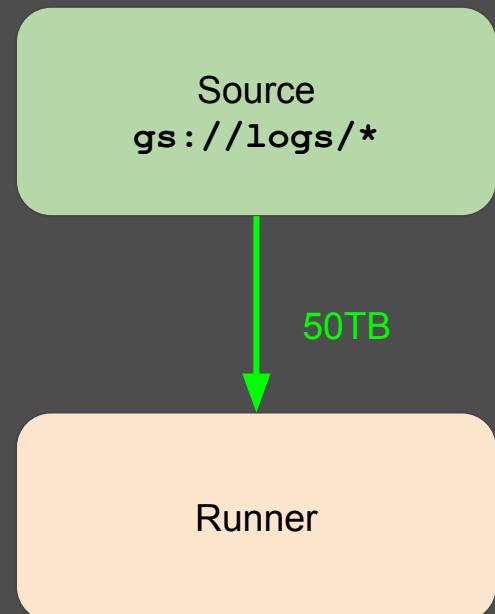
→ **user** decides

"Read from this source"

→ **runner** decides

APIs:

- long **getEstimatedSize()**
- List<Source> **splitIntoBundles(size)**



Beam abstractions empower runners

Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"

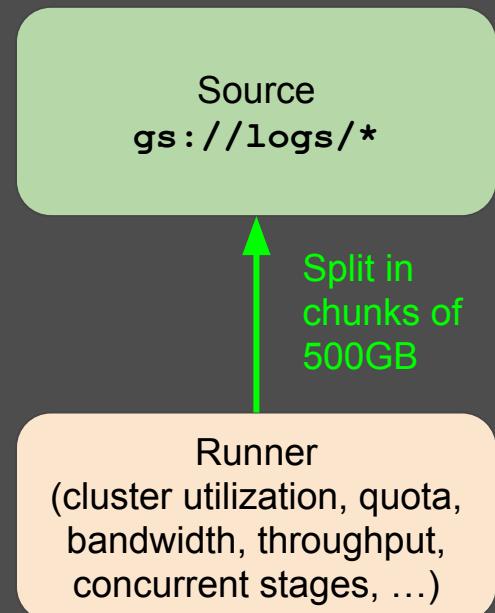
→ **user** decides

"Read from this source"

→ **runner** decides

APIs:

- long **getEstimatedSize()**
- List<Source> **splitIntoBundles(size)**



Beam abstractions empower runners

Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"

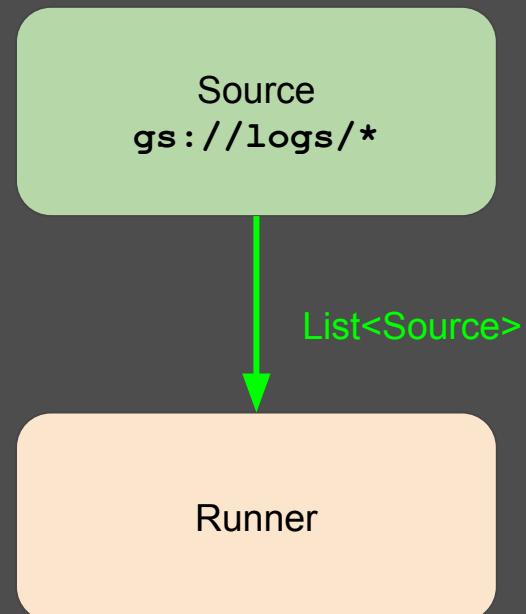
→ **user** decides

"Read from this source"

→ **runner** decides

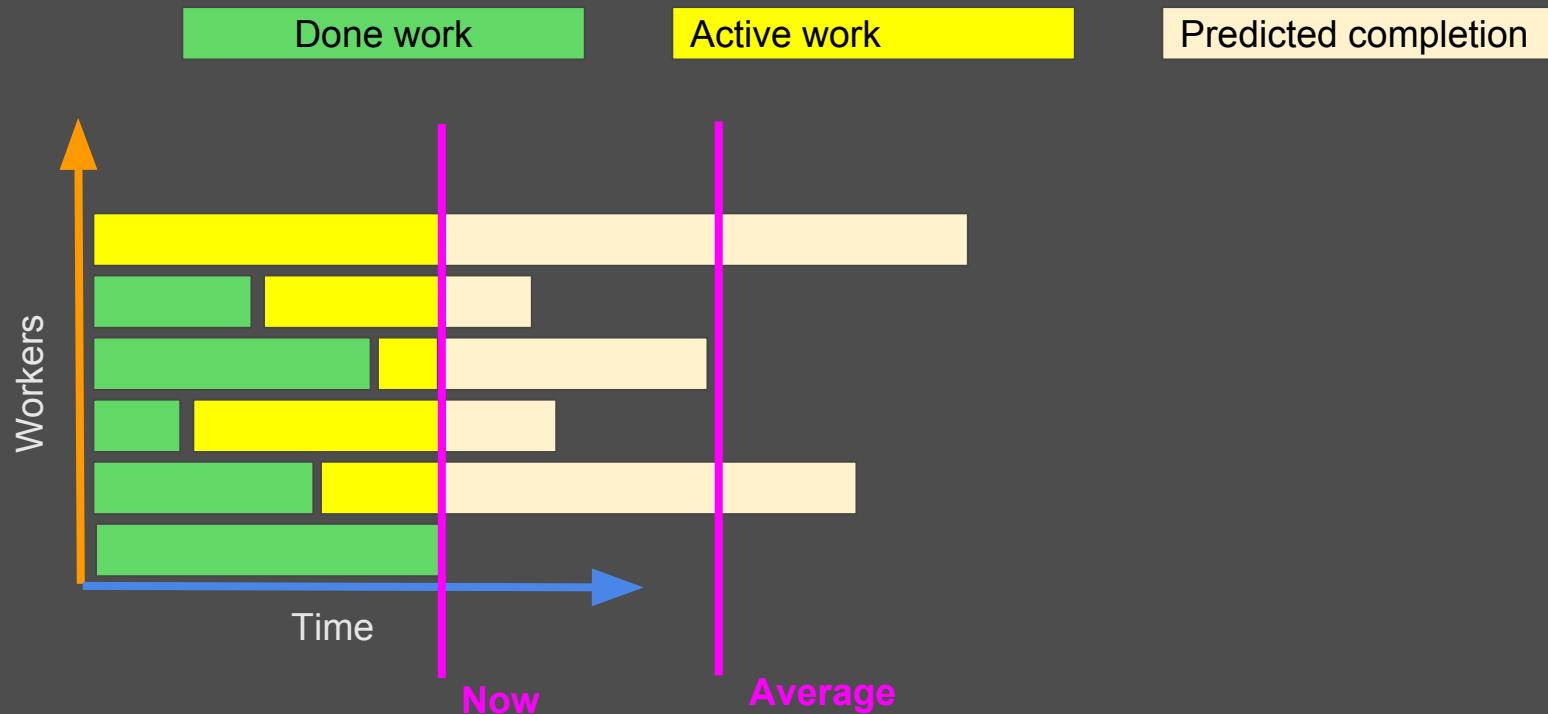
APIs:

- long **getEstimatedSize()**
- List<Source> **splitIntoBundles(size)**

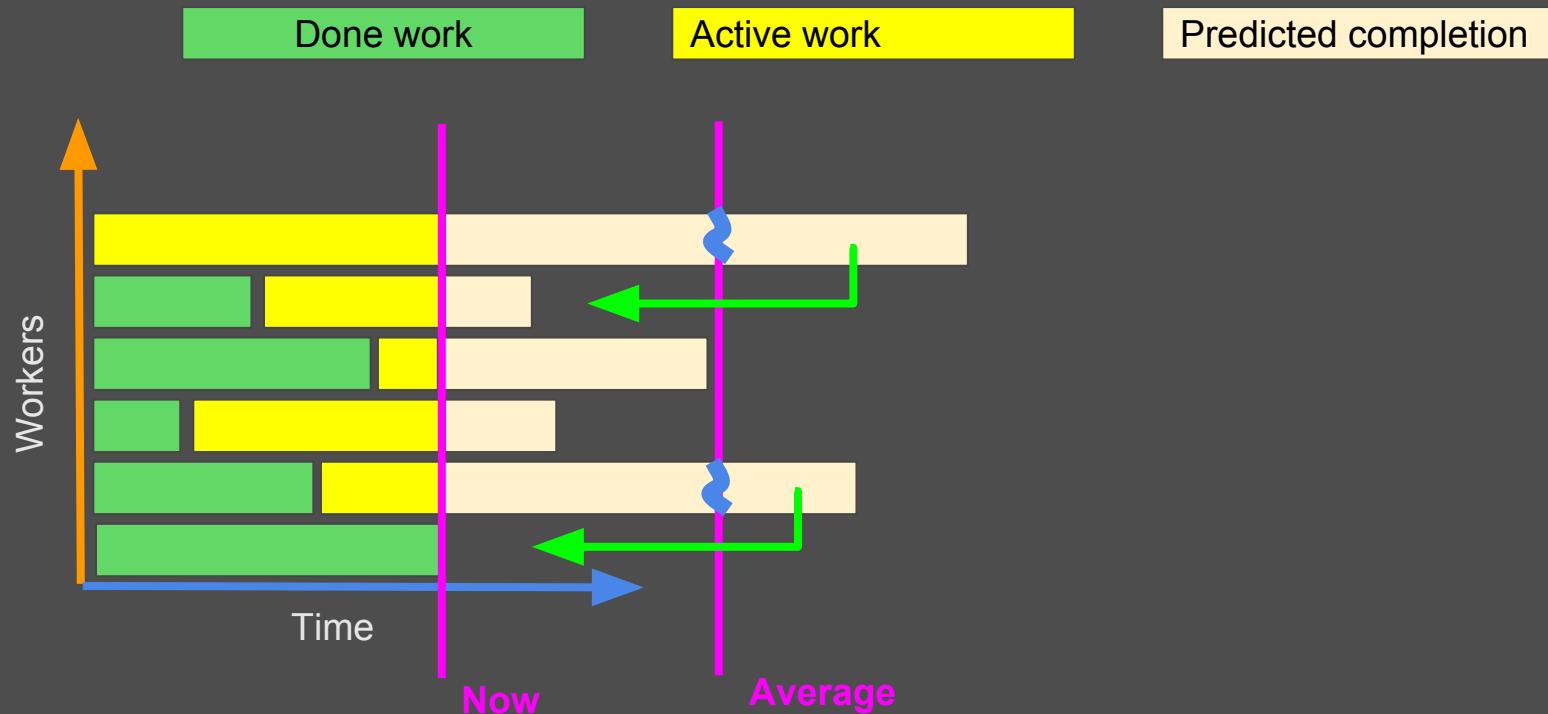


Solving the straggler problem: Dynamic Work Rebalancing

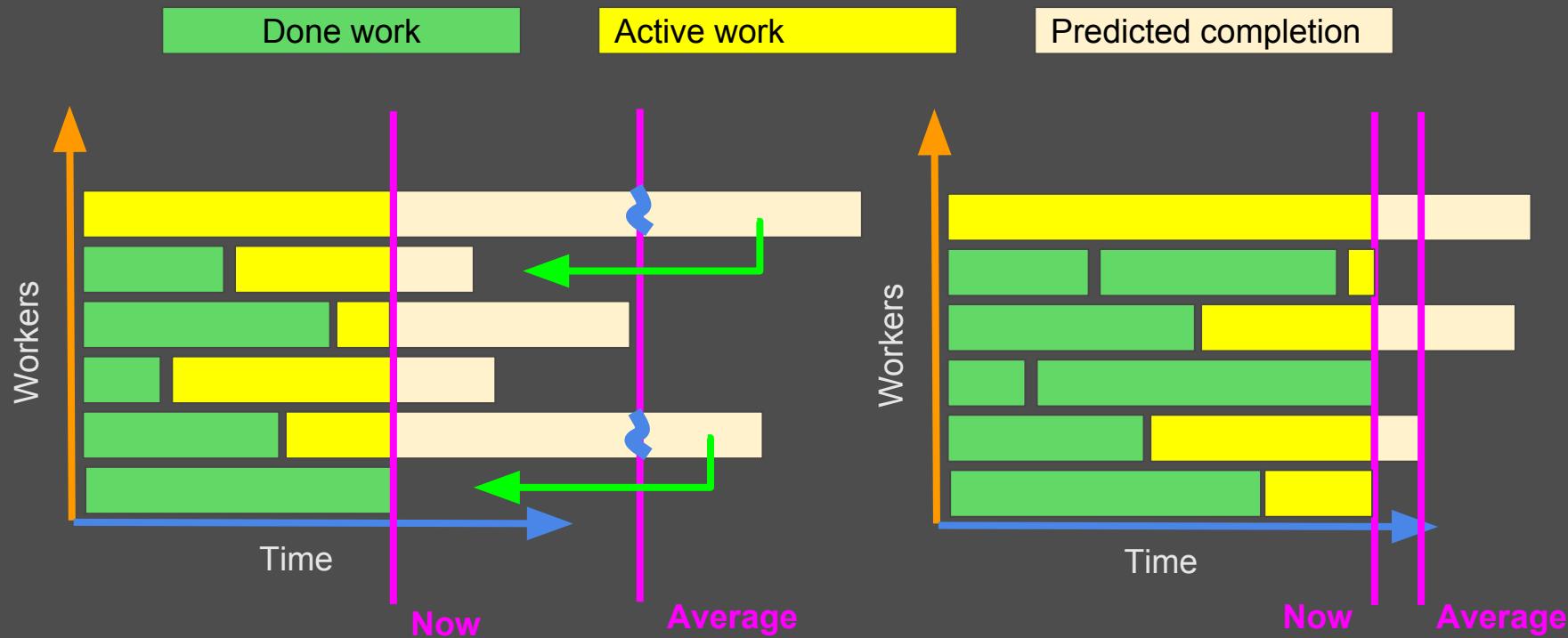
Solving the straggler problem: Dynamic Work Rebalancing



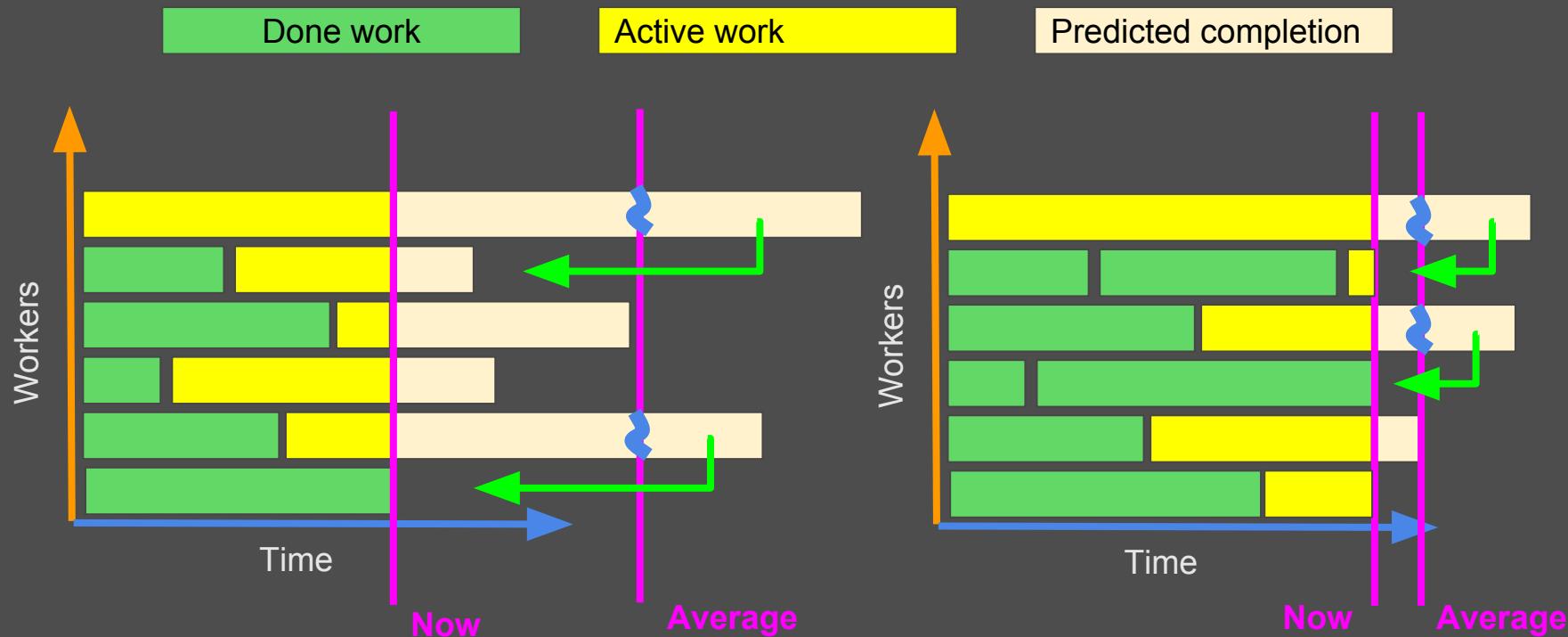
Solving the straggler problem: Dynamic Work Rebalancing



Solving the straggler problem: Dynamic Work Rebalancing



Solving the straggler problem: Dynamic Work Rebalancing



Dynamic Work Rebalancing in the wild



A classic MapReduce job (read from Google Cloud Storage, GroupByKey, write to Google Cloud Storage), 400 workers.

Dynamic Work Rebalancing disabled to demonstrate stragglers.

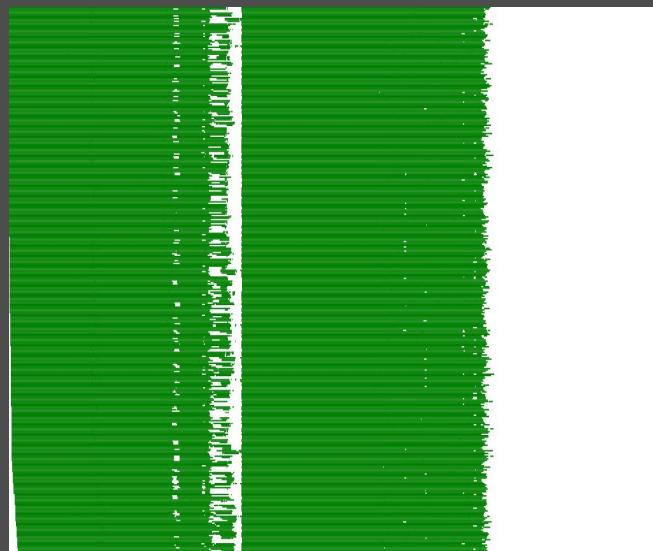
X axis: time (total ~20min.); Y axis: workers

Dynamic Work Rebalancing in the wild



A classic MapReduce job (read from Google Cloud Storage, GroupByKey, write to Google Cloud Storage), 400 workers.
Dynamic Work Rebalancing disabled to demonstrate stragglers.

X axis: time (total ~20min.); Y axis: workers



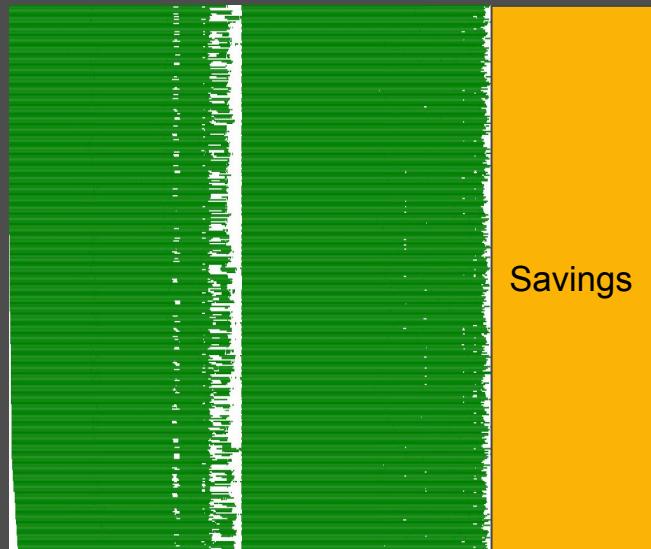
Same job,
Dynamic Work Rebalancing enabled.
X axis: time (total ~15min.); Y axis: workers

Dynamic Work Rebalancing in the wild



A classic MapReduce job (read from Google Cloud Storage, GroupByKey, write to Google Cloud Storage), 400 workers.
Dynamic Work Rebalancing disabled to demonstrate stragglers.

X axis: time (total ~20min.); Y axis: workers



Same job,
Dynamic Work Rebalancing enabled.
X axis: time (total ~15min.); Y axis: workers

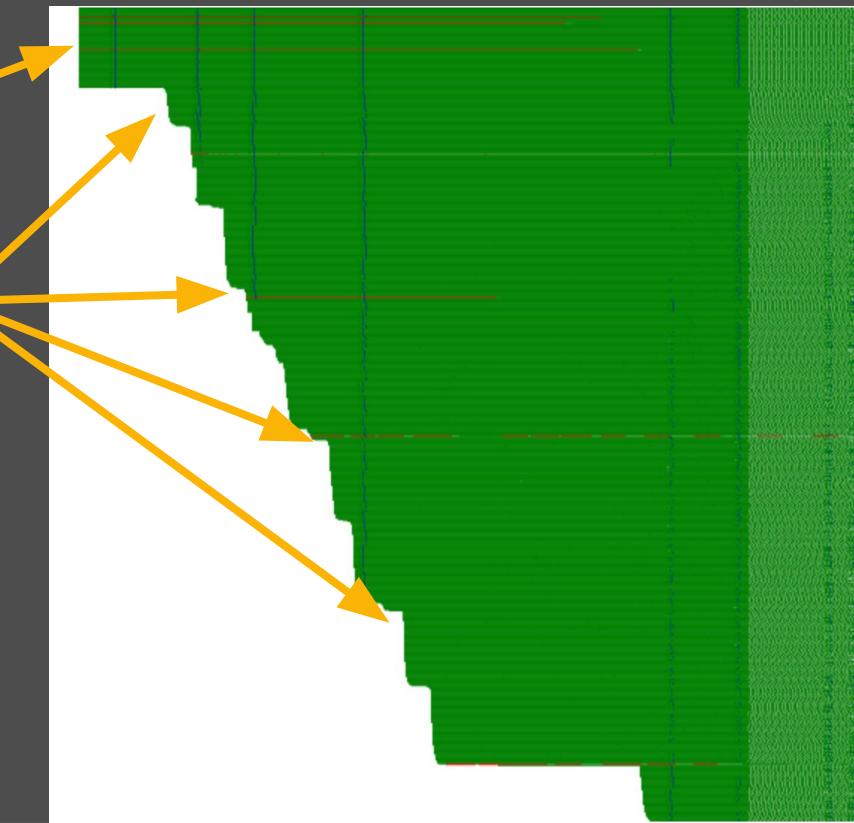
Dynamic Work Rebalancing with Autoscaling

Initial allocation of 80 workers, based on **size**

Multiple rounds of **upsizing**, enabled by dynamic work rebalancing

Upscales to 1000 workers.

- tasks are balanced
- no oversplitting or manual tuning



Apache Beam enable dynamic adaptation

Beam Source Readers provide **simple progress signals**, which enable runners to take action based on execution-time characteristics.

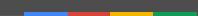
All Beam runners can implement Autoscaling and Dynamic Work Rebalancing.

APIs for how much work is pending.

- **bounded:** `double getFractionConsumed()`
- **unbounded:** `long getBacklogBytes()`

APIs for splitting:

- **bounded:**
 - `Source splitAtFraction(double)`
 - `int getParallelismRemaining()`
- **unbounded:**
 - Coming soon ...





Apache Beam is a **unified** programming model designed to provide **efficient** and **portable** data processing pipelines.

To learn more

Read our blog posts!

- No shard left behind: Dynamic work rebalancing in Google Cloud Dataflow
<https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow>
- Comparing Cloud Dataflow autoscaling to Spark and Hadoop
<https://cloud.google.com/blog/big-data/2016/03/comparing-cloud-dataflow-autoscaling-to-spark-and-hadoop>

Join the Apache Beam community!

<https://beam.apache.org/>



Portable stateful big data processing in Apache Beam

Kenneth Knowles

Apache Beam PMC
Software Engineer @ Google
klk@google.com / [@KennKnowles](https://twitter.com/KennKnowles)

<https://s.apache.org/ffsf-2017-beam-state>
Flink Forward San Francisco 2017

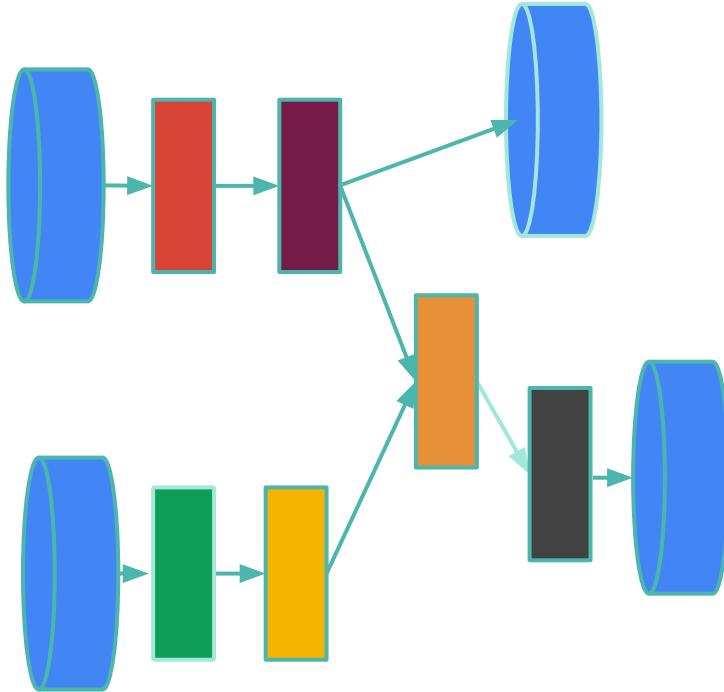
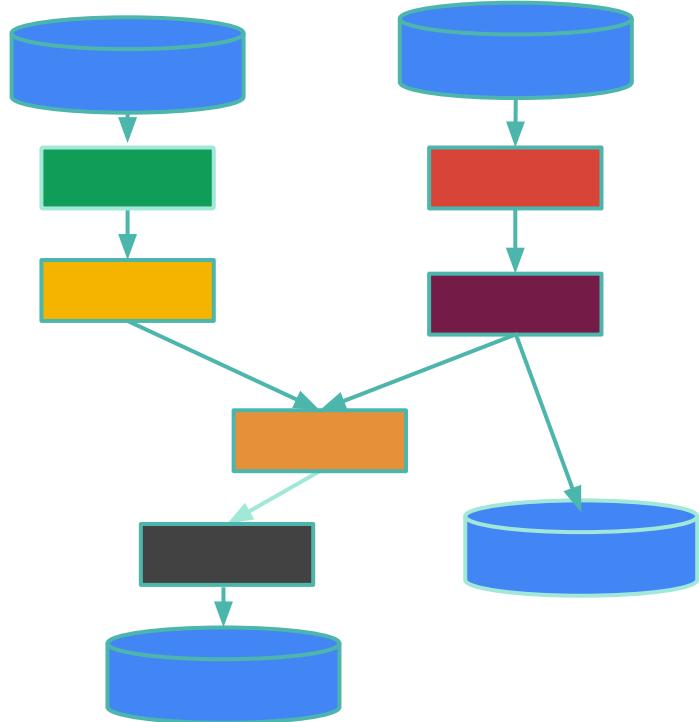
Agenda

1. What is Apache Beam?
2. State
3. Timers
4. Example & Little Demo

What is Apache Beam?

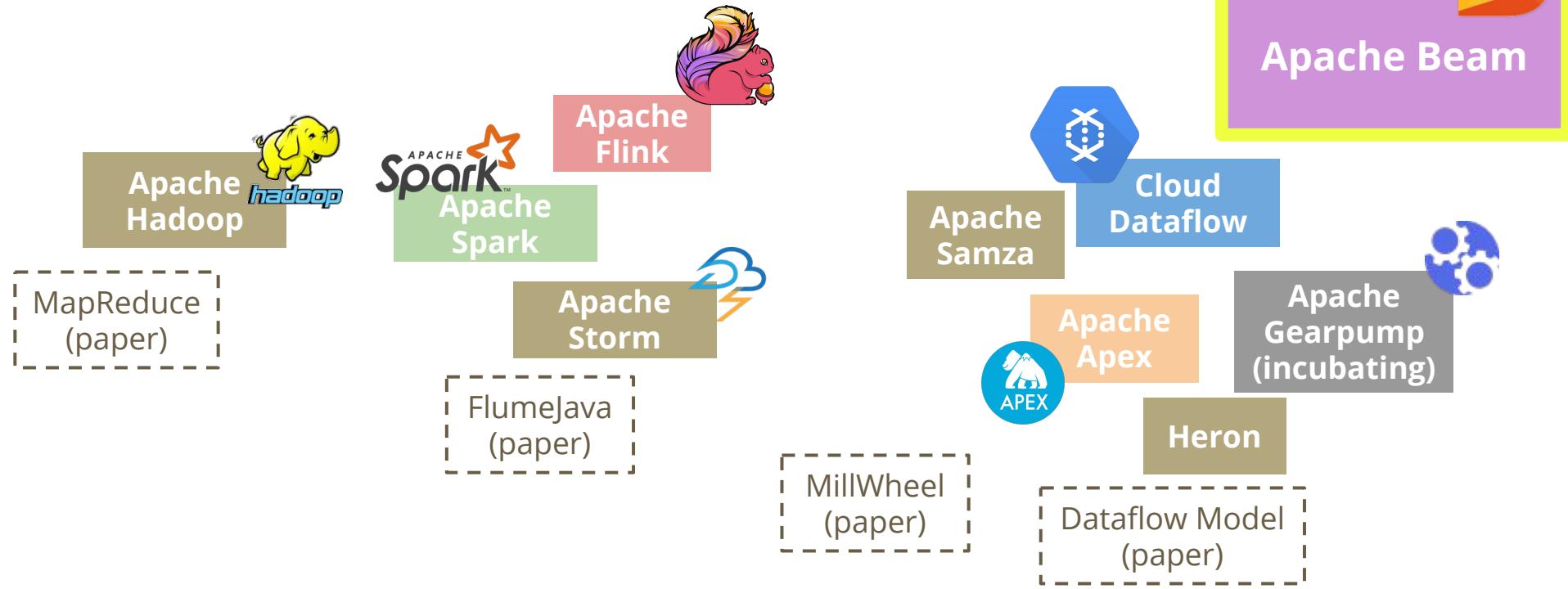


TL;DR



(Flink draws it more like this)

DAGs, DAGs, DAGs



2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016

The Beam Vision

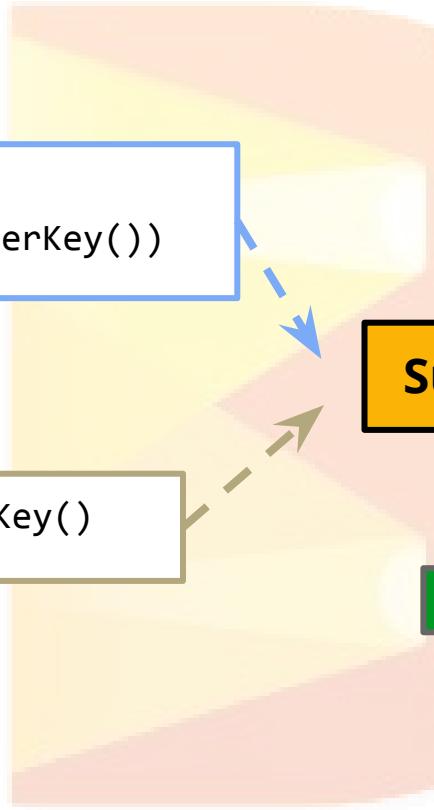
Java

```
input.apply(  
    Sum.integersPerKey())
```

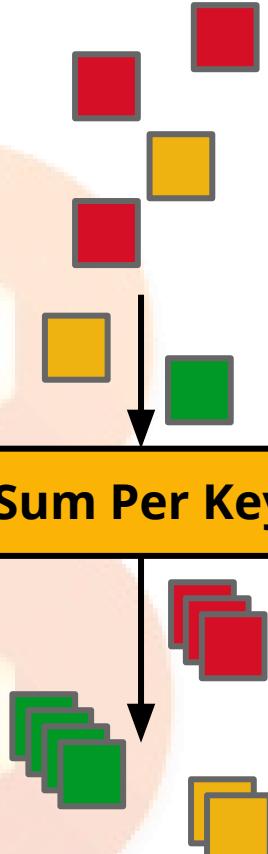
Python

```
input | Sum.PerKey()
```

⋮



Sum Per Key



Apache Flink
local, on-prem,
cloud



Cloud Dataflow:
fully managed



Apache Spark
local, on-prem,
cloud



Apache Apex
local, on-prem,
cloud



Apache
Gearpump
(incubating)

The Beam Vision

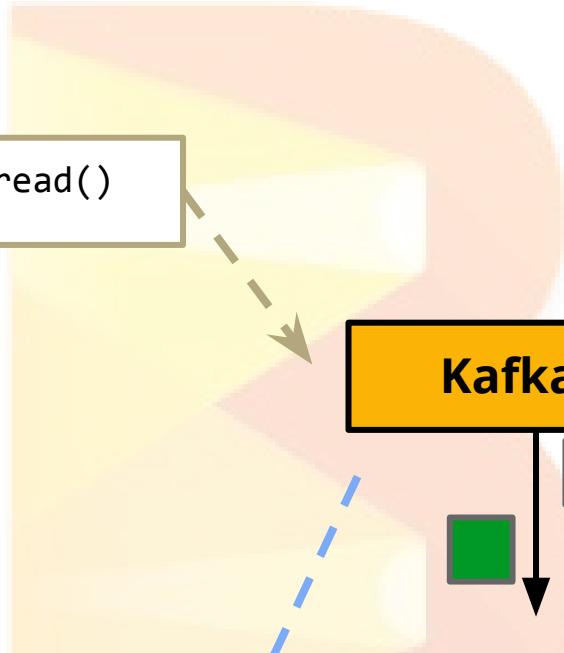
Python

```
input | KafkaIO.read()
```

⋮

Java

```
class KafkaIO extends  
UnboundedSource { ... }
```



Apache Flink
local, on-prem,
cloud



Cloud Dataflow:
fully managed



Apache Spark
local, on-prem,
cloud



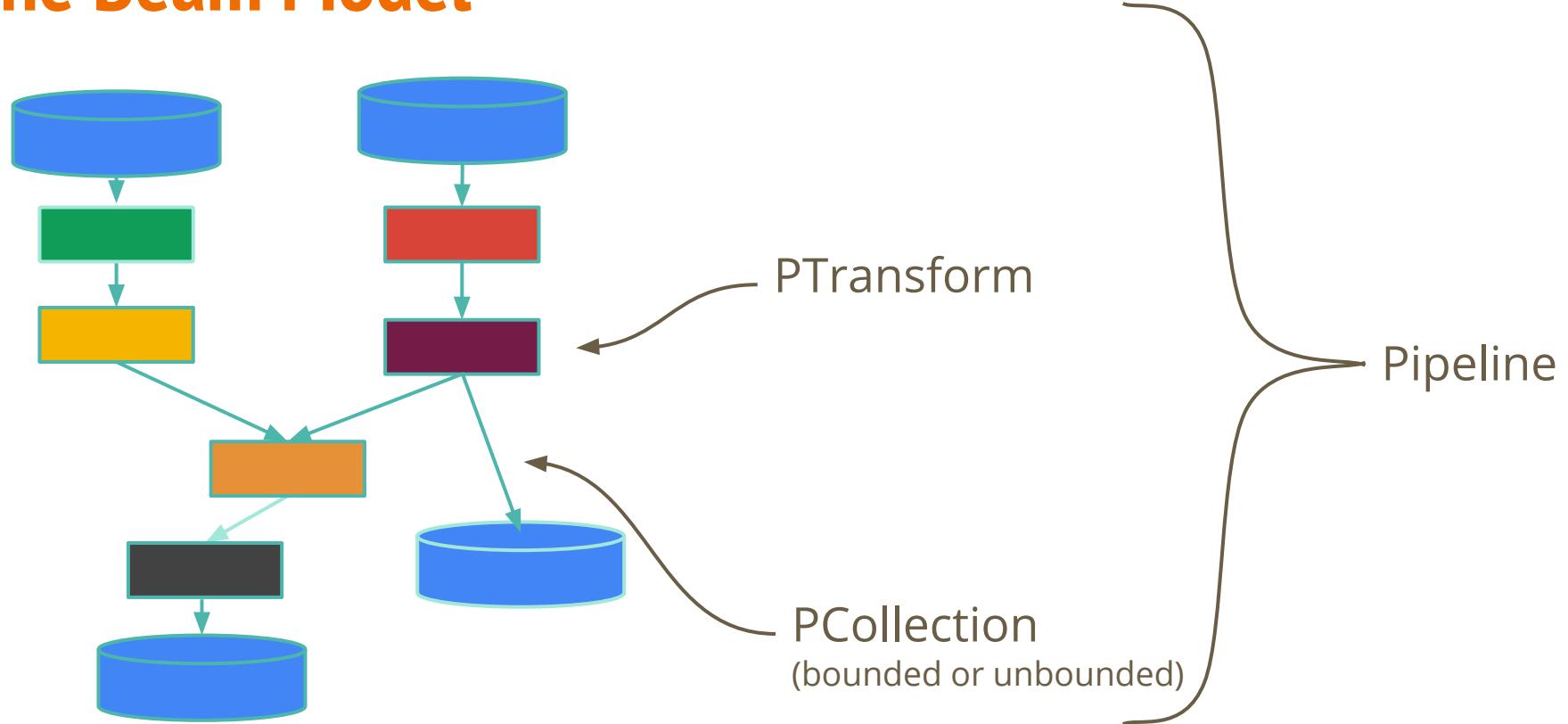
Apache Apex
local, on-prem,
cloud



Apache
Gearpump
(incubating)

⋮

The Beam Model



The Beam Model

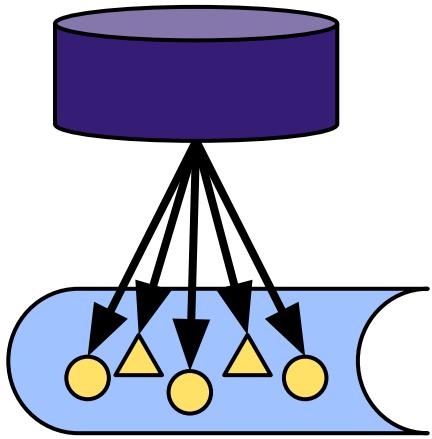
What are you computing? (read, map, reduce)

Where in event time? (event time windowing)

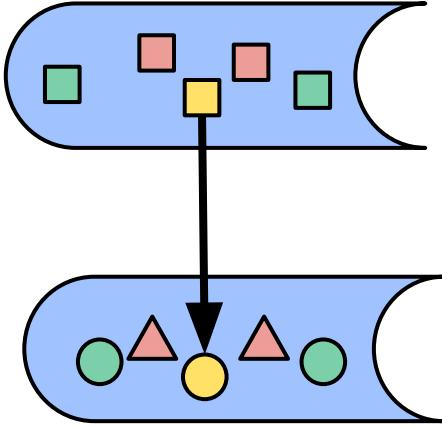
When in processing time are results produced? (triggers)

How do refinements relate? (accumulation mode)

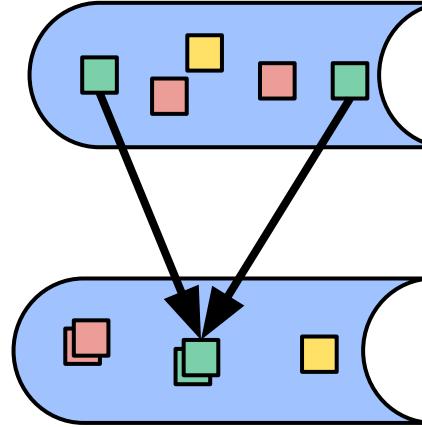
What are you computing?



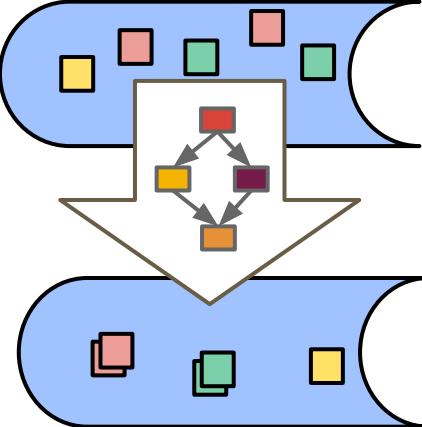
Read
Parallel connectors to external systems



ParDo
Per element "Map"



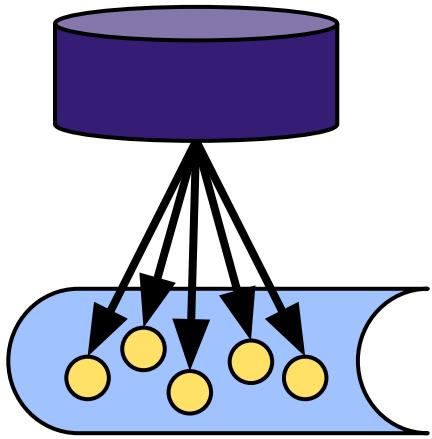
Grouping
Group by key, Combine per key, "Reduce"



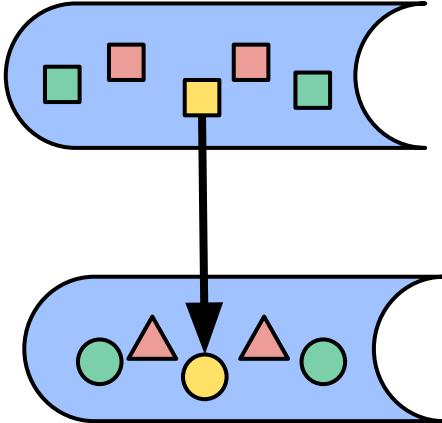
Composite
Encapsulated subgraph

State

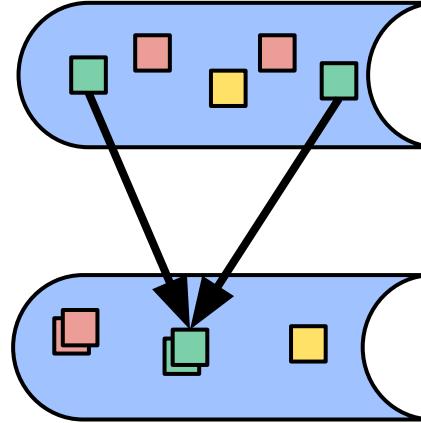
What are you computing?



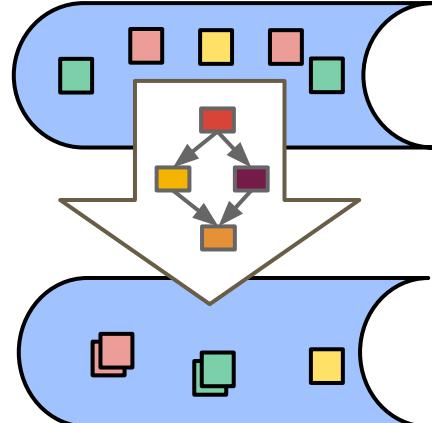
Read
Parallel connectors to external systems



ParDo
Per element "Map"

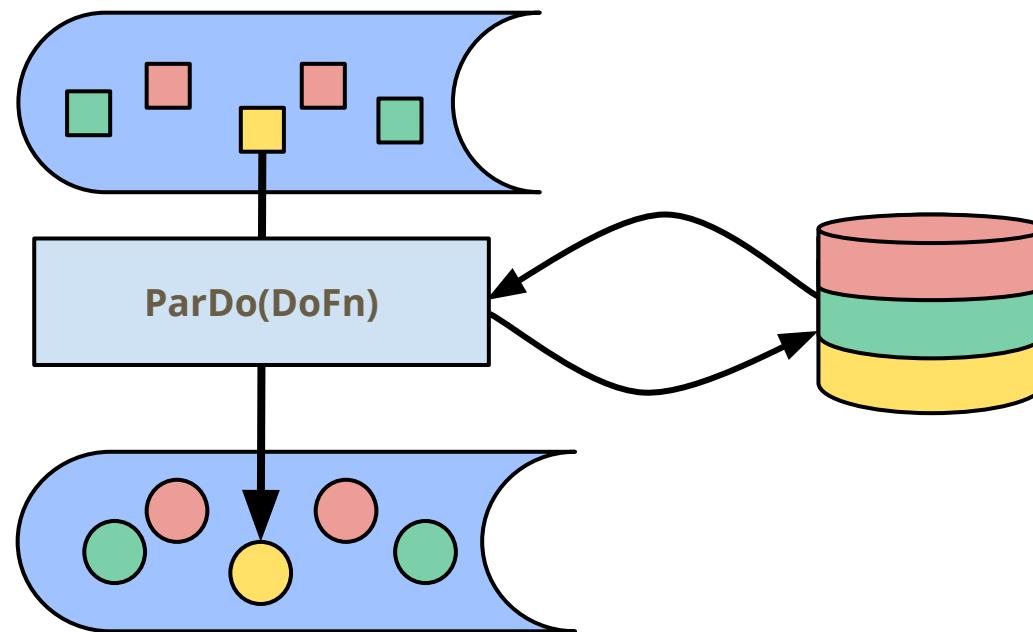


Grouping
Group by key, Combine per key, "Reduce"

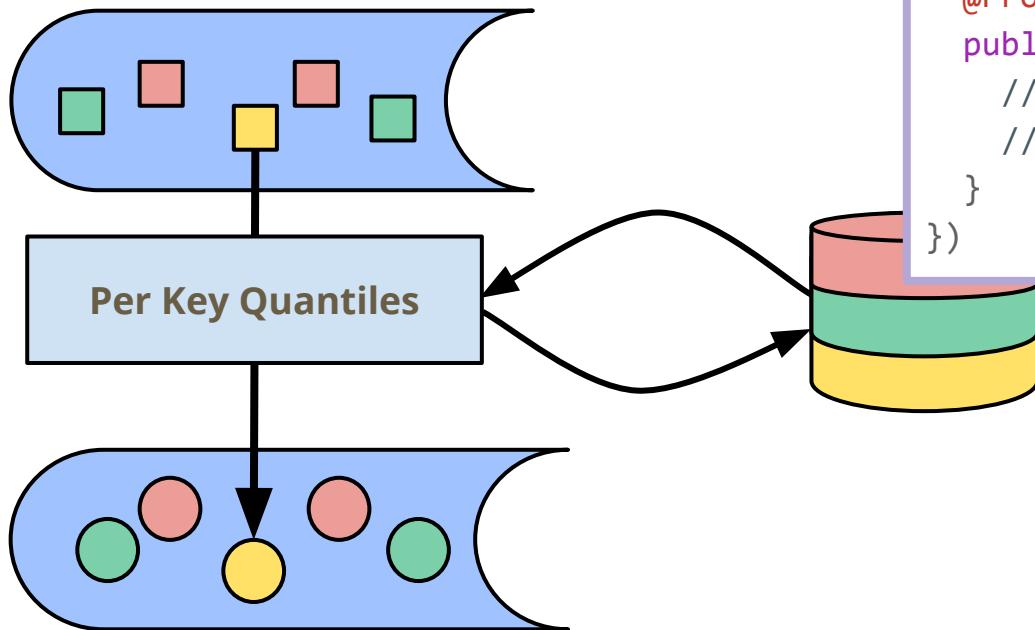


Composite
Encapsulated subgraph

State for ParDo

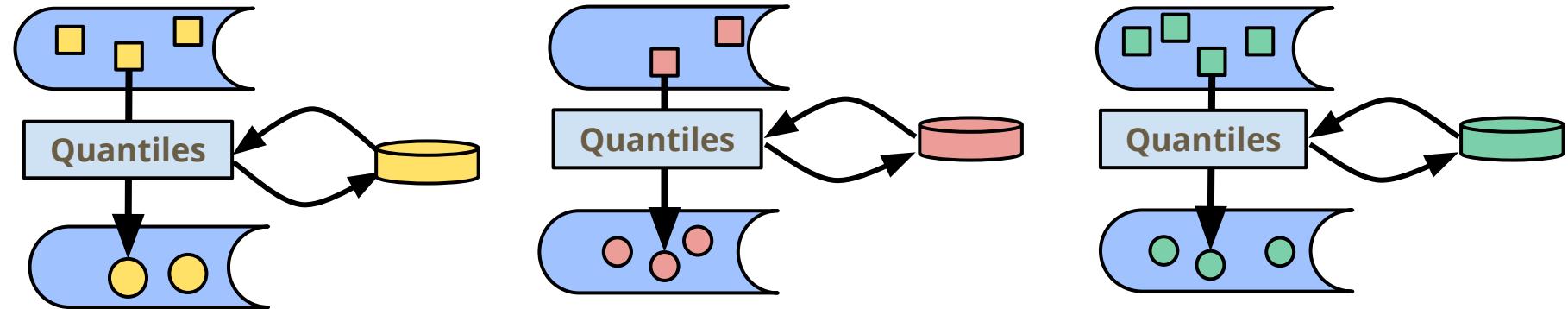


"Example"

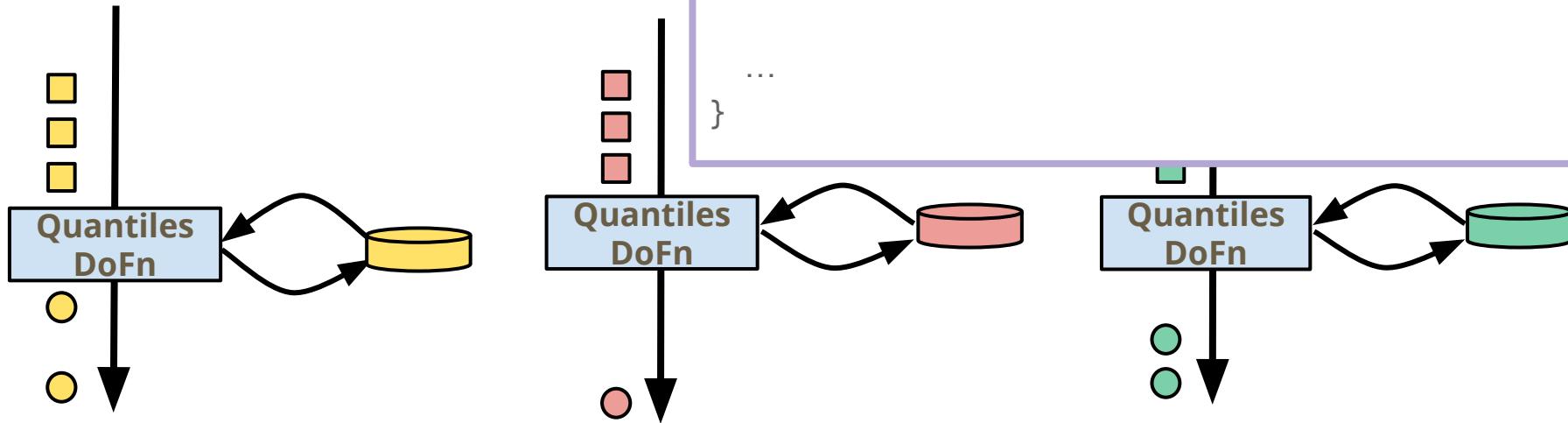


```
ParDo.of(new DoFn<...>() {  
    // declare some state  
  
    @ProcessElement  
    public void process(...) {  
        // update quantiles  
        // output if needed  
    }  
})
```

Partitioned

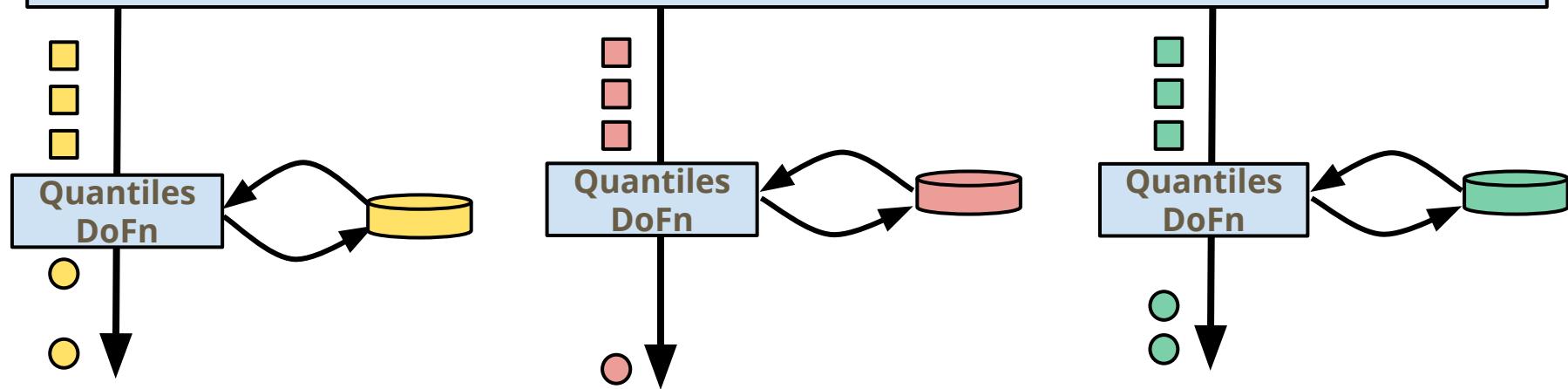


Parallelism!



Windowed

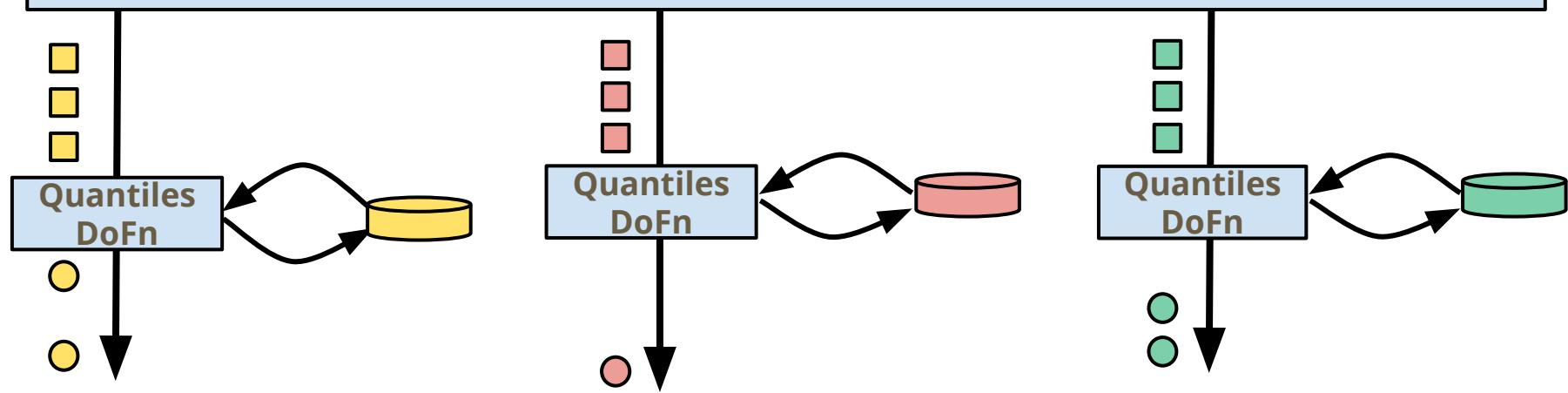
Window into Fixed windows of one hour



Expected result: Quantiles for each hour

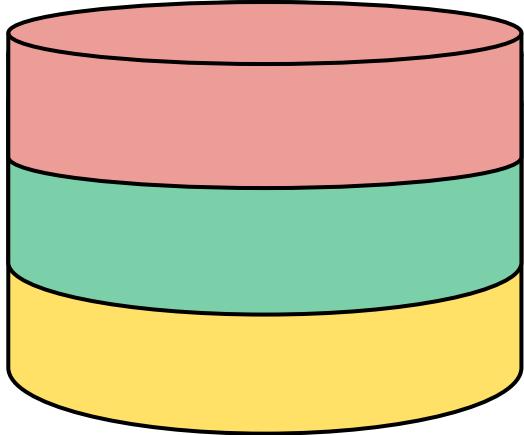
Windowed

Window into windows of 30 min sliding by 10 min



Expected result: Quantiles for 30 minutes sliding by 10 min

State is per key and window

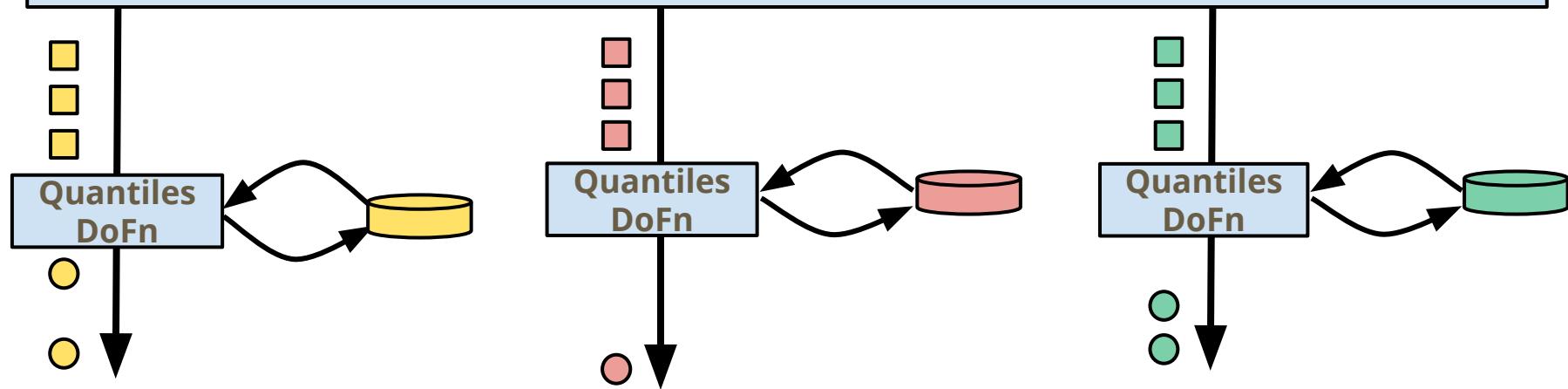


	$\langle k, w \rangle_1$	$\langle k, w \rangle_2$	$\langle k, w \rangle_3$...
"x"	3	7	15	
"y"	"fizz"	"7"	"fizzbuzz"	
...				

Bonus: automatically garbage collected when a window expires
(vs manual clearing of keyed state)

Windowed

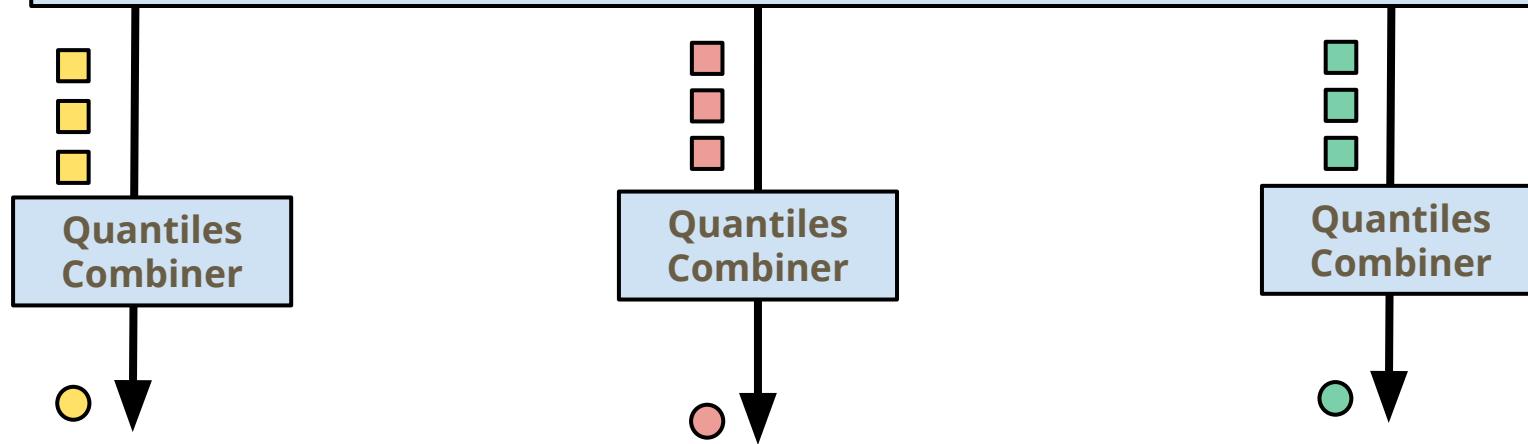
Window into Fixed windows of one hour



Expected result: Quantiles for each hour

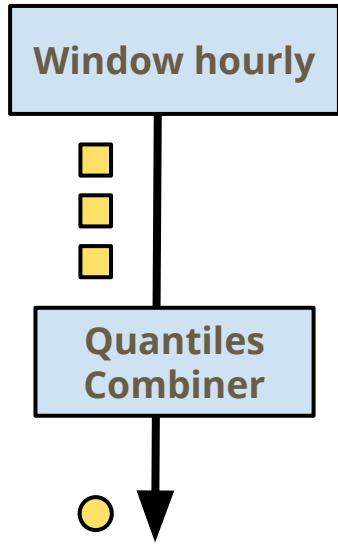
What about Combine?

Window into Fixed windows of one hour

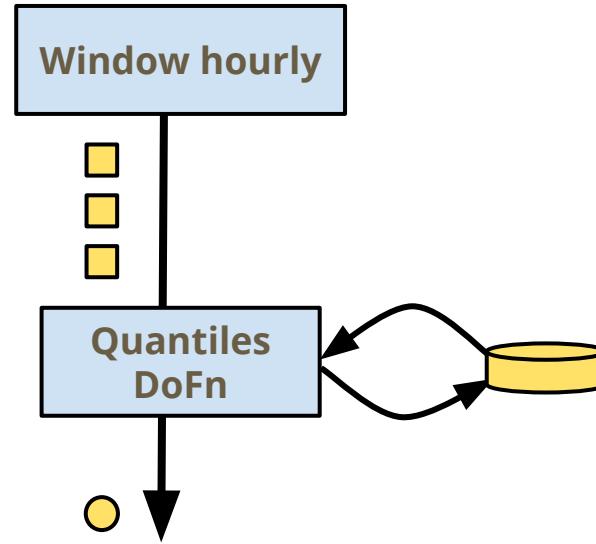


Expected result: Quantiles for each hour

Combine vs State (naive conceptualization)

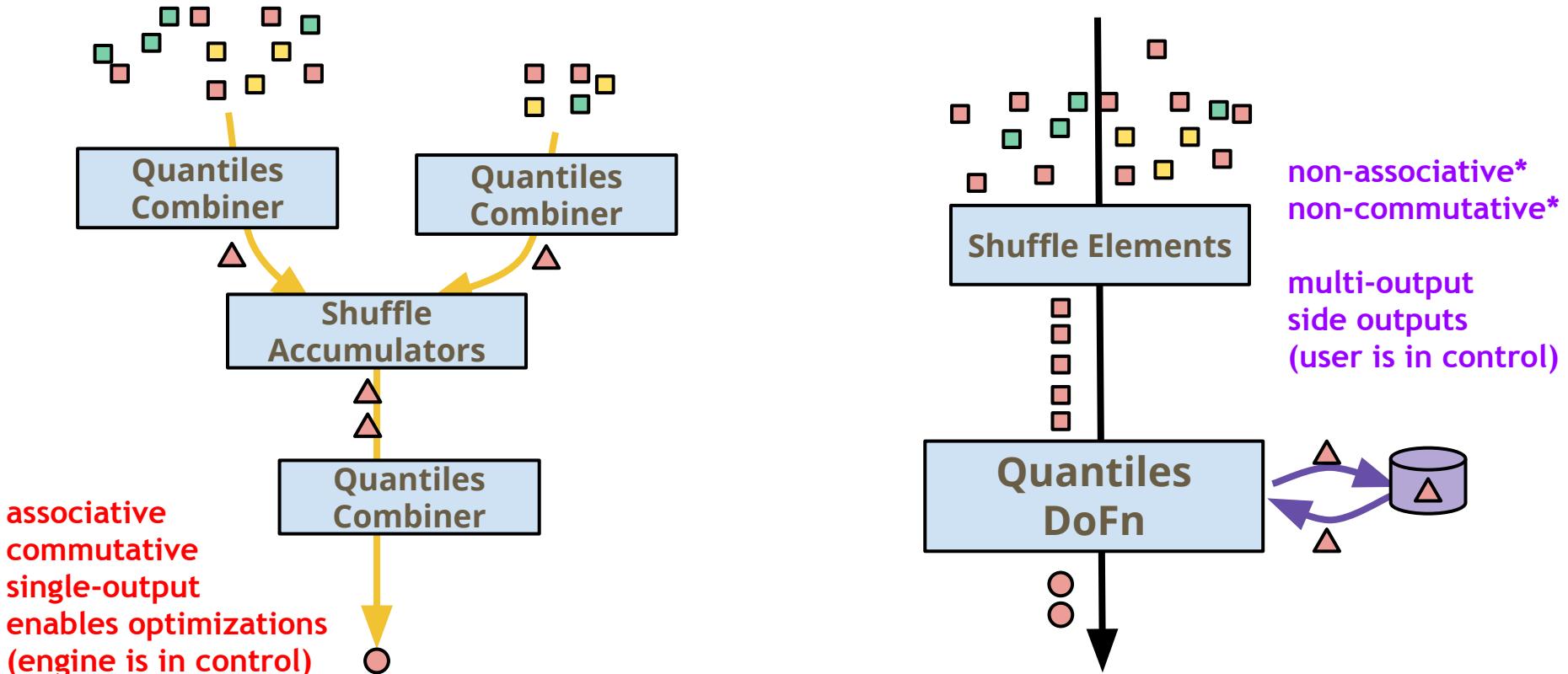


Expected result:
Quantiles for each hour

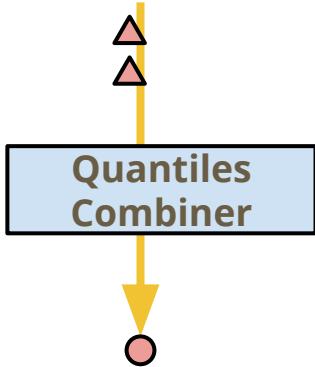


Expected result:
Quantiles for each hour

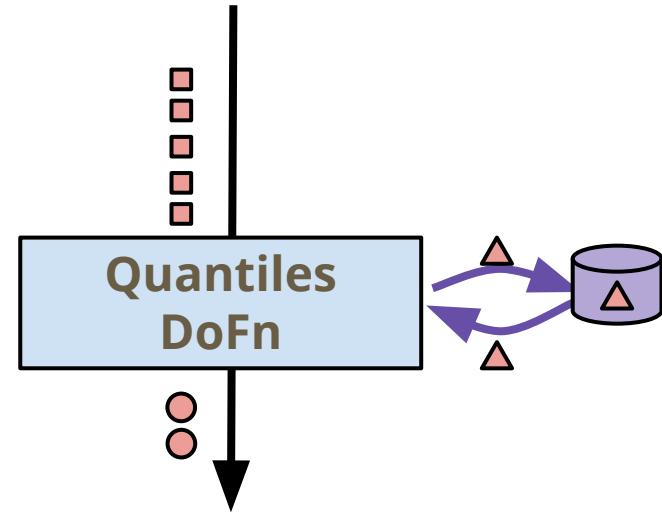
Combine vs State (likely execution plan)



Combine vs State

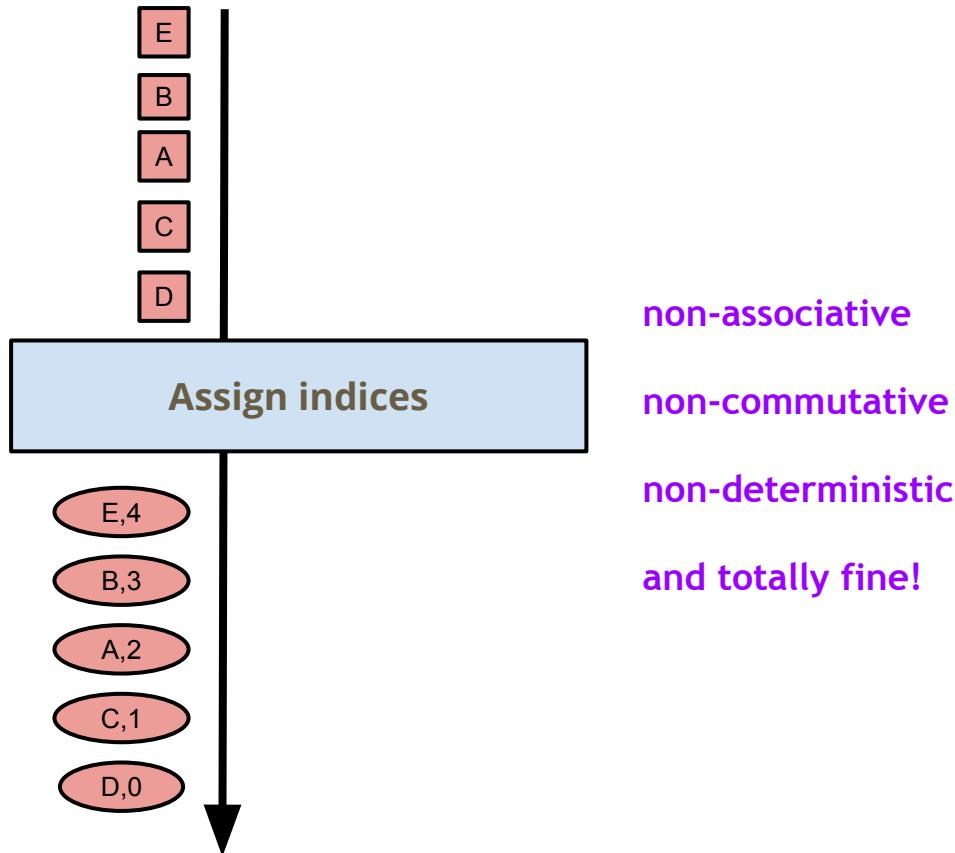


output governed by trigger
(data/computation unaware)



"output only when there's an interesting change"
(data/computation aware)

Example: Arbitrary indices



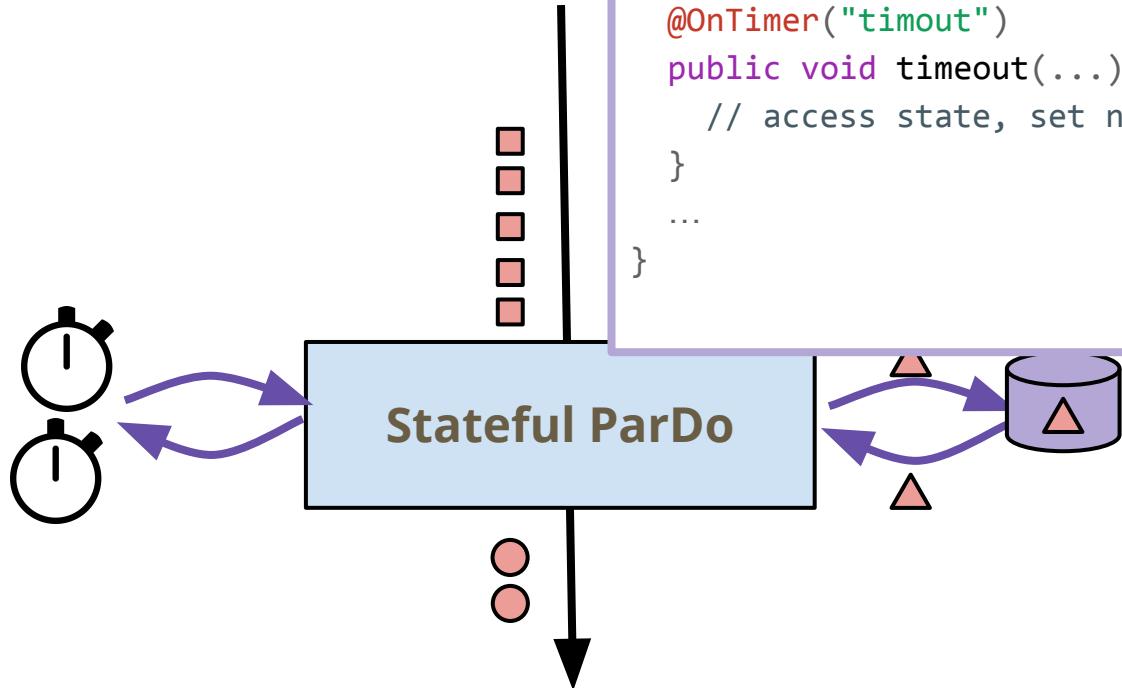
Kinds of state

- **Value** - just a mutable cell for a value
- **Bag** - supports "blind writes"
- **Combining** - has a CombineFn built in; can support blind writes and lazy accumulation
- **Set** - membership checking
- **Map** - lookups and partial writes

Timers

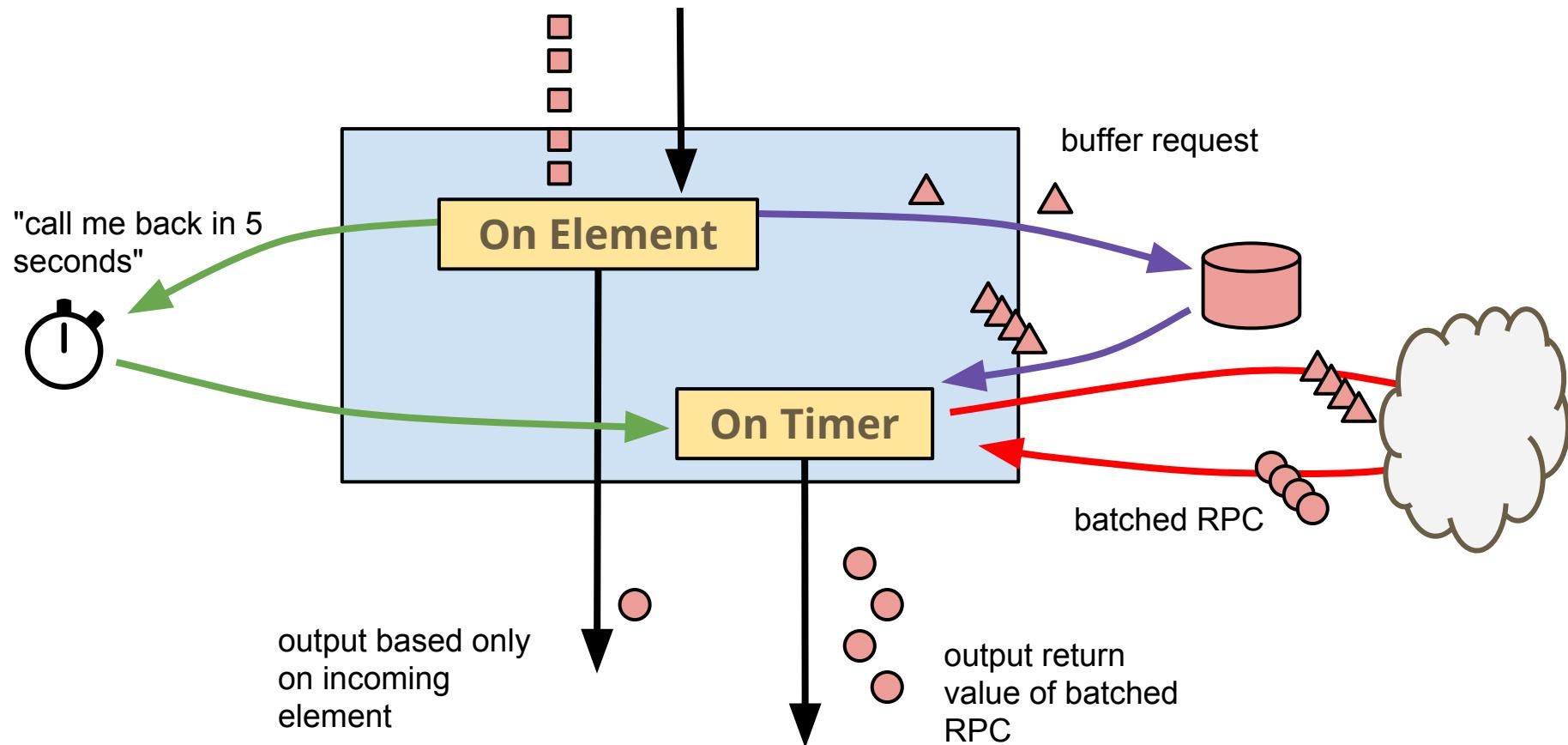
—

Timers for ParDo

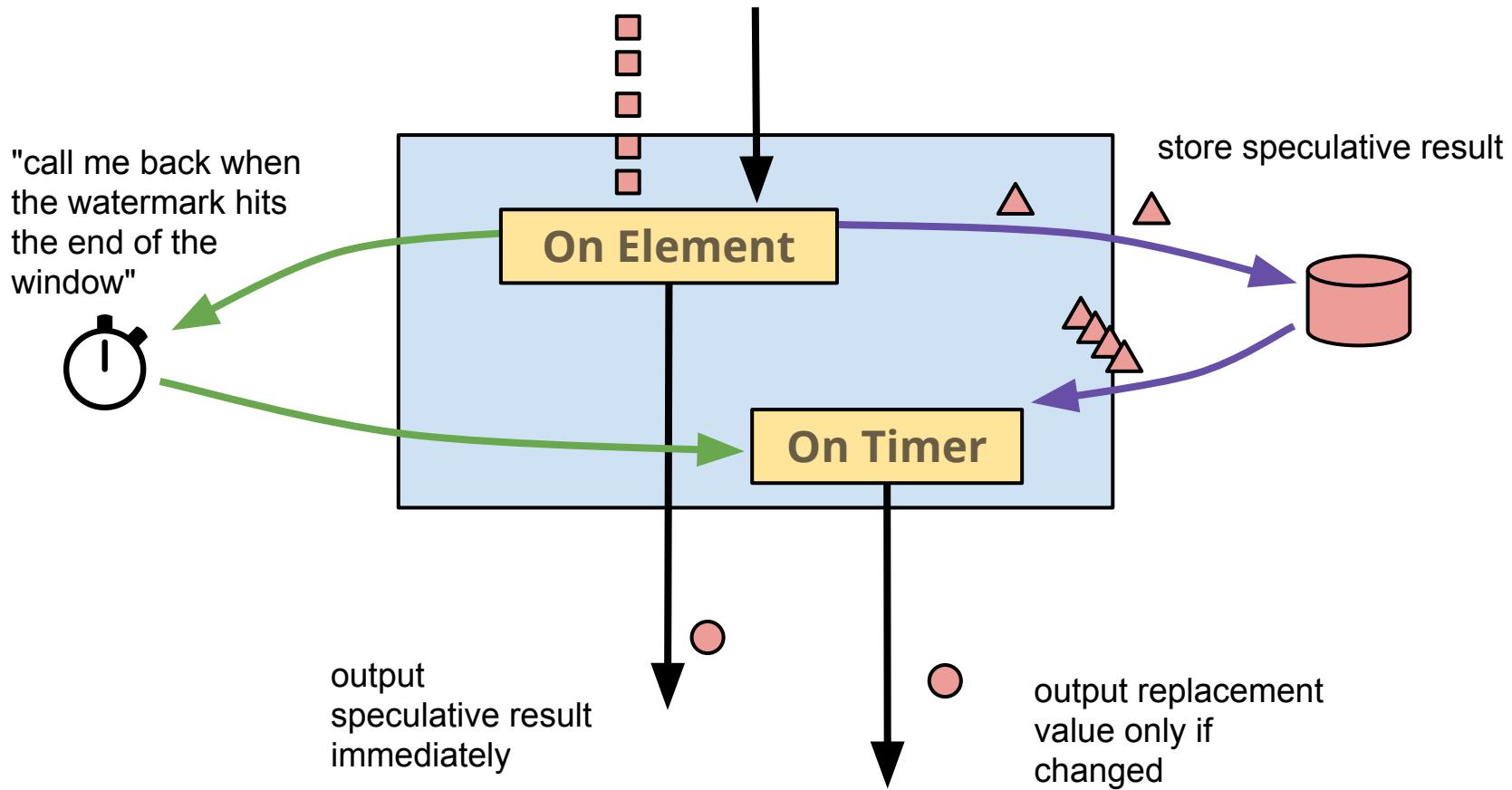


```
new DoFn<...>() {  
    @TimerId("timeout")  
    private final TimerSpec timeoutTimer =  
        TimerSpecs.timer(TimeDomain.PROCESSING_TIME);  
  
    @OnTimer("timeout")  
    public void timeout(...) {  
        // access state, set new timers, output  
    }  
    ...  
}
```

Timers in processing time



Timers in event time



More example uses for state & timers

- Per-key arbitrary numbering
- Output only when result changes
- Tighter "side input" management for slowly changing dimension
- Streaming join-matrix / join-biclique
- Fine-grained combine aggregation and output control
- Per-key "workflows" like user sign up flow w/ expiration
- Low-latency deduplication (let the first through, squash the rest)

Performance considerations (cross-runner)

- Shuffle to colocate keys
- Linear processing of elements for key+window
- Window merging
- Storage of state and timers
- GC of state

Demo

Summary

State and Timers in Beam...

- ... unlock new uses cases
- ... they "just work" with event time windowing
- ... are portable across runners (implementation ongoing)

Thank you for listening!

This talk:

- Me - [@KennKnowles](#)
- These Slides - <https://s.apache.org/ffsf-2017-beam-state>

Go Deeper

- Design doc - <https://s.apache.org/beam-state>
- Blog post - <https://beam.apache.org/blog/2017/02/13/stateful-processing.html>

Join the Beam community:

- User discussions - user@beam.apache.org
- Development discussions - dev@beam.apache.org
- Follow [@ApacheBeam](#) on Twitter

You can contribute to Beam + Flink

- New types of state
- Easy launch of Beam job on Flink-on-YARN
- Integration tests at scale
- Fit and finish: polish, polish, polish!
- ... and lots more!

<https://beam.apache.org>

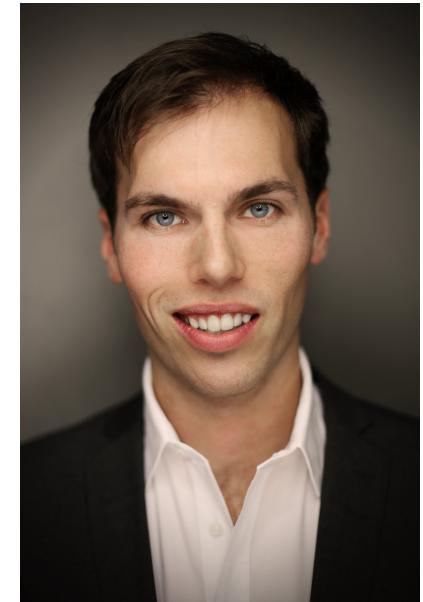
Queryable State

How to Build a Billing System Without a Database

Maximilian Bode, Konstantin Knauf



- Software Engineers with TNG Technology Consulting
- Focus on
 - Distributed Systems
 - Real-Time Processing
- Flink in Production for 1.5 y

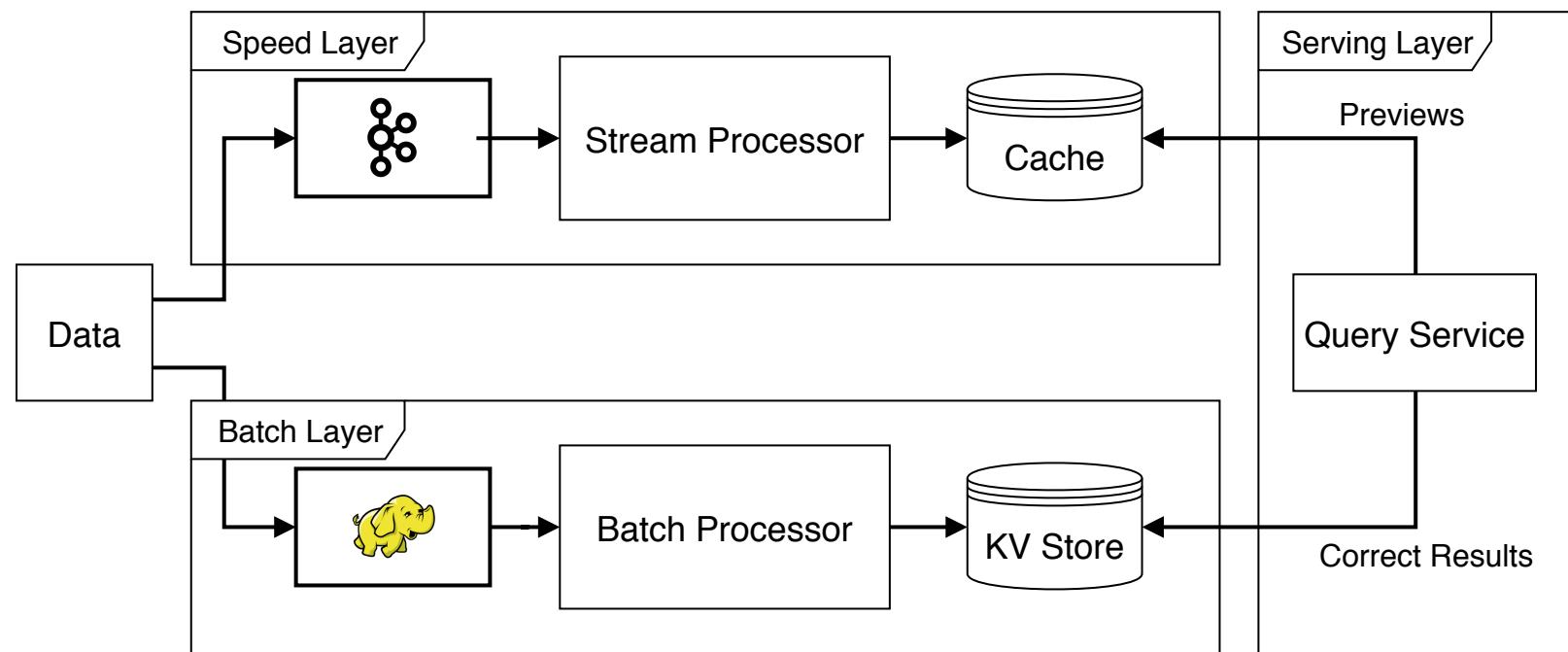


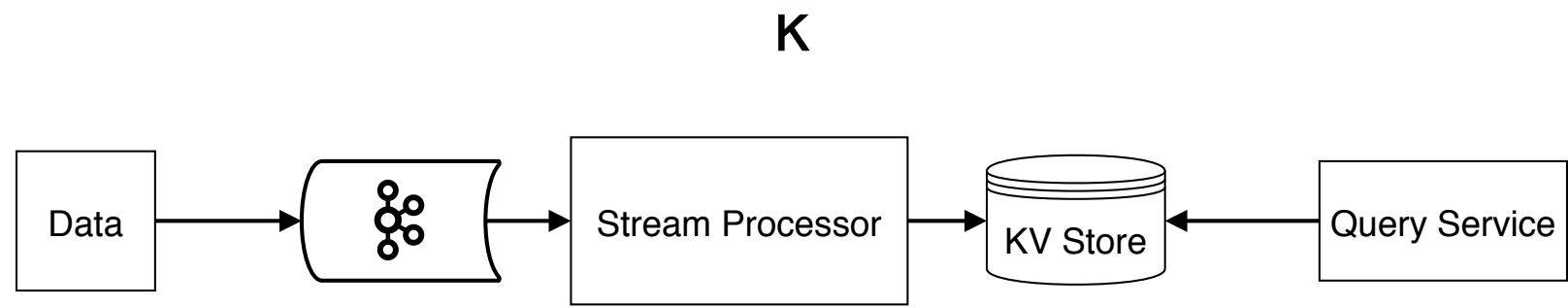
Agenda

1. Intro to Queryable State
2. Use Case: A Queryable Billing System
3. Live Demo of Our Prototype

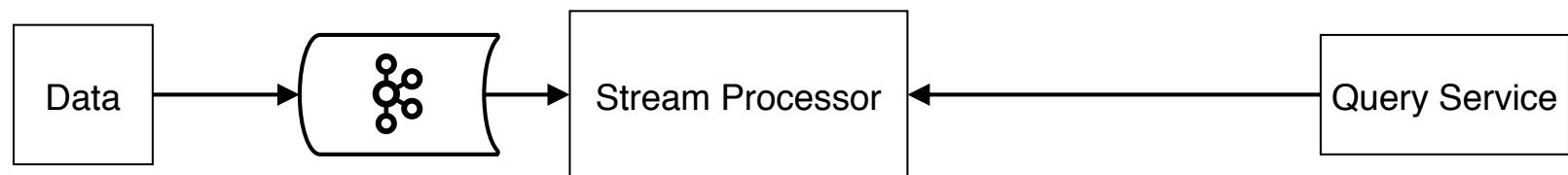
Intro to Queryable State

Evolution of Streaming Architecture

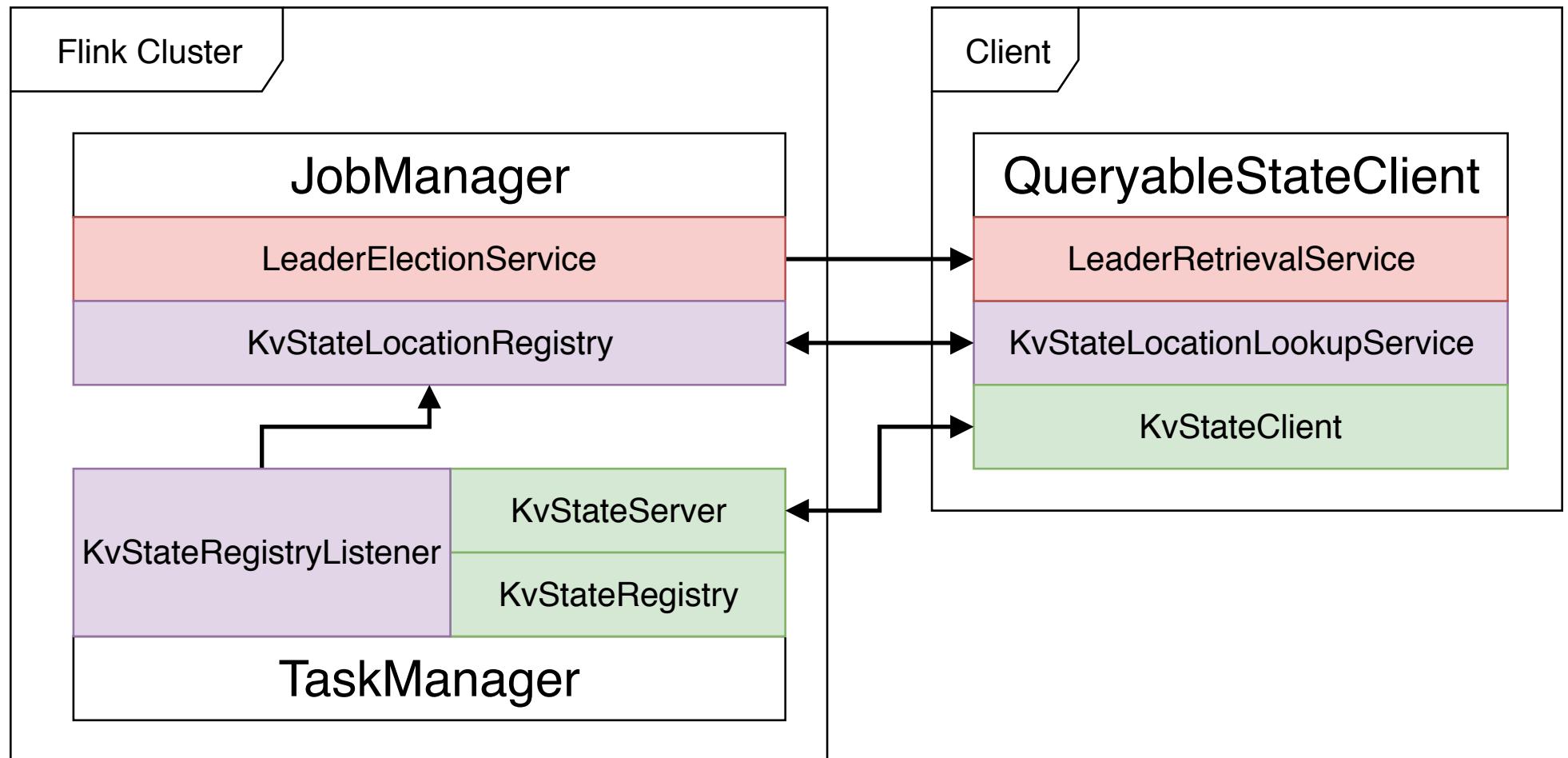
λ 



Queryable State

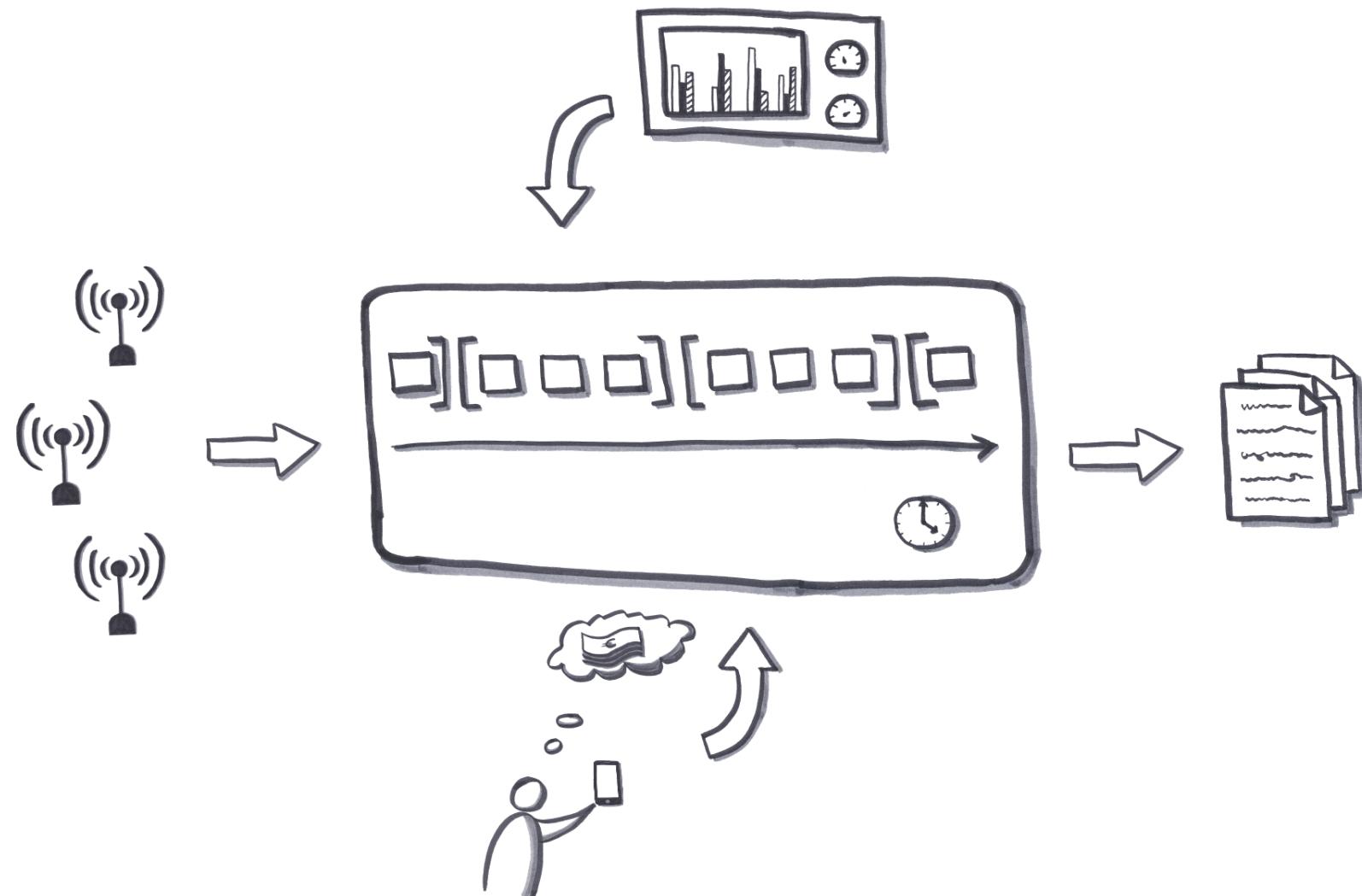


Implementation in Flink



Use Case: “Queryable Billing”

Requirements





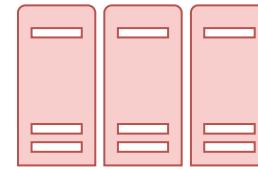
Correctness



Robustness

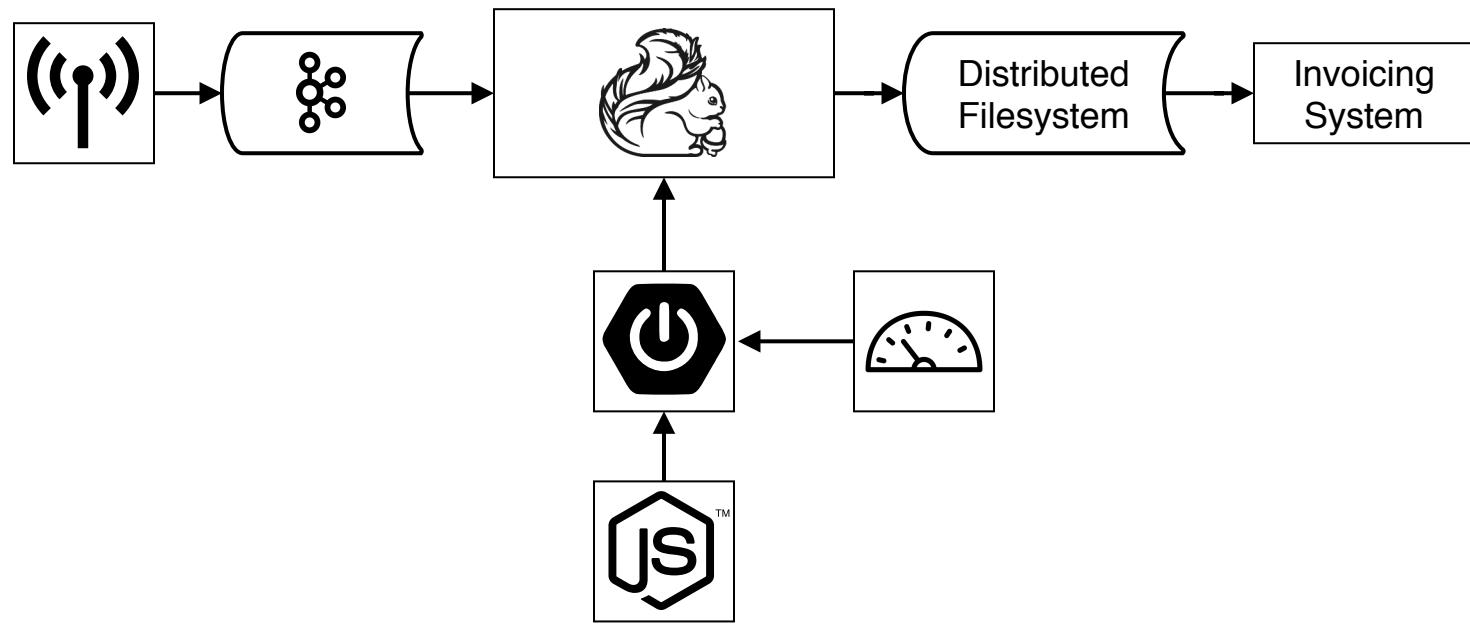


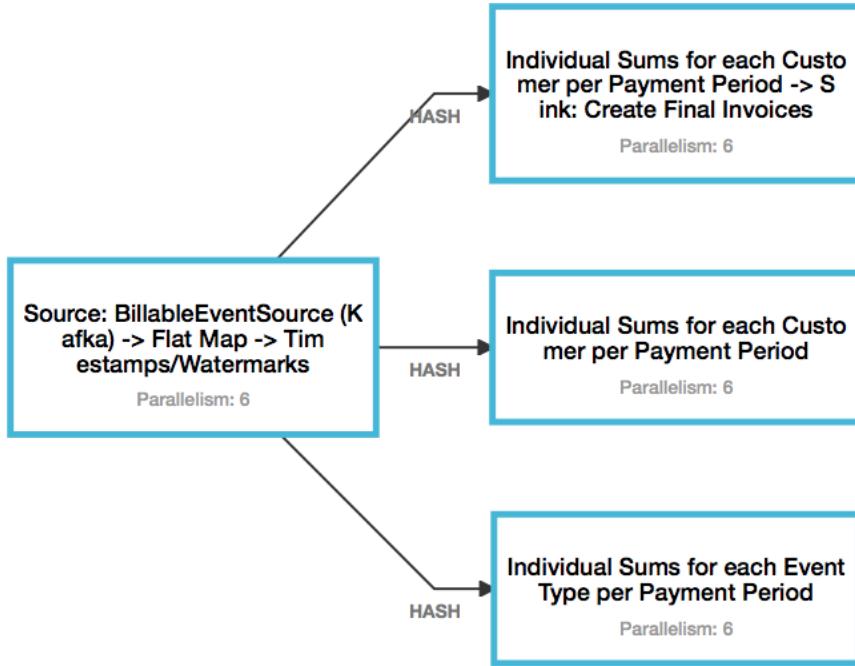
Availability



Scalability

Architecture





- Events from Kafka
- Final Invoices
 - TimeWindow with FoldFunction
 - BucketingSink
- Previews
 - TimeWindow with FoldFunction
 - CountTrigger.of(1)
 - Queryable State in WindowFunction

Demo

Requirements

- Invoices
- Live Updates
- Correctness
- Availability
- (Scalability)
- Robustness
 - Late Events
 - TaskManager
 - JobManager
 - Sink

Invoice Generation

Requirement

Requirement
Invoices

Demo

Demo


Live Updates

Correctness

()

Availability

Robustness

Late Events



TaskManager

JobManager

Sink

Live Updates

Requirement	Demo
Requirement Invoices	Demo ✓
Live Updates	✓
Correctness	(✓)
Availability	
Robustness	Late Events ✓
	TaskManager
	JobManager
	Sink

TaskManager Failure

Requirement	Demo
Requirement Invoices	Demo ✓
Live Updates	✓
Correctness	(✓)
Availability	(✓)
Robustness	Late Events ✓
	TaskManager ✓
	JobManager
	Sink

JobManager Failure

Requirement	Demo	
Requirements	Demo	
Live Updates	✓	
Correctness	(✓)	
Availability	(✓)	
Robustness	Late Events	✓
	TaskManager	✓
	JobManager	✓
	Sink	

Failure of Downstream Systems

Requirement	Demo	
Requirement	Demo	
Live Updates	✓	
Correctness	✓	
Availability	((✓))	
Robustness	Late Events	✓
	TaskManager	✓
	JobManager	✓
	Sink	✓

Current Limitations

- Native Support for Queryable State in Windows
- Improvements of Client API
- State Size
- Availability in Case of Job Failures

Questions?

- ✉ maximilian.bode@tng.tech
- ✉ konstantin.knauf@tng.tech
- in [LinkedIn/maxbode](https://www.linkedin.com/in/maxbode)
- 🐦 [@snntrable](https://twitter.com/snntrable)



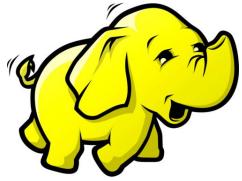
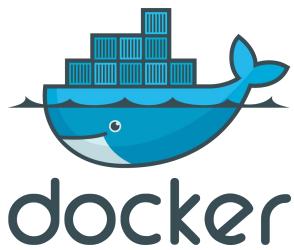


dataArtisans

ONE DOES NOT SIMPLY

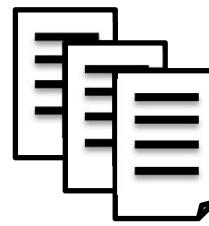


CHANGE ONE'S CLUSTER ENVIRONMENT

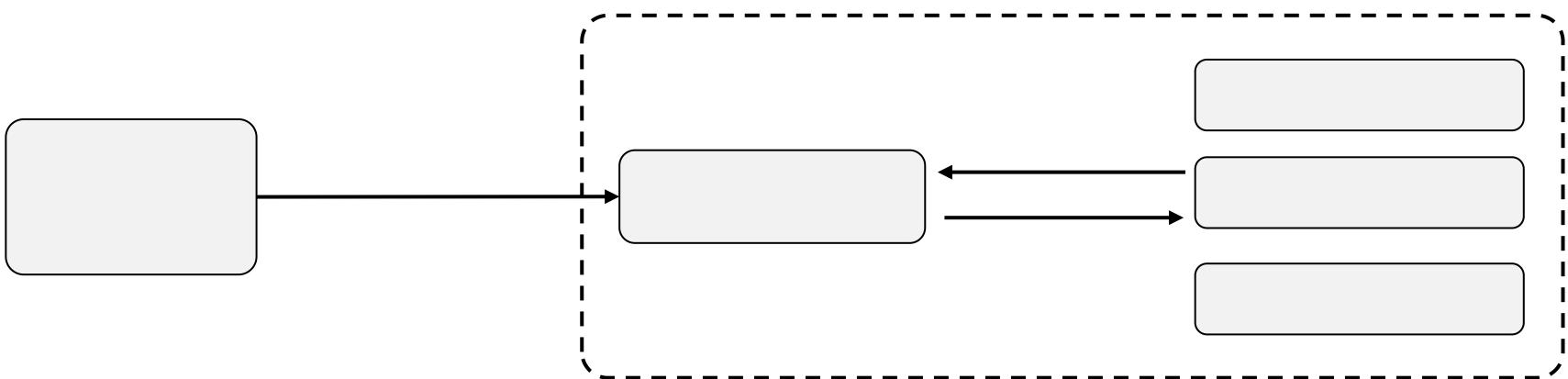


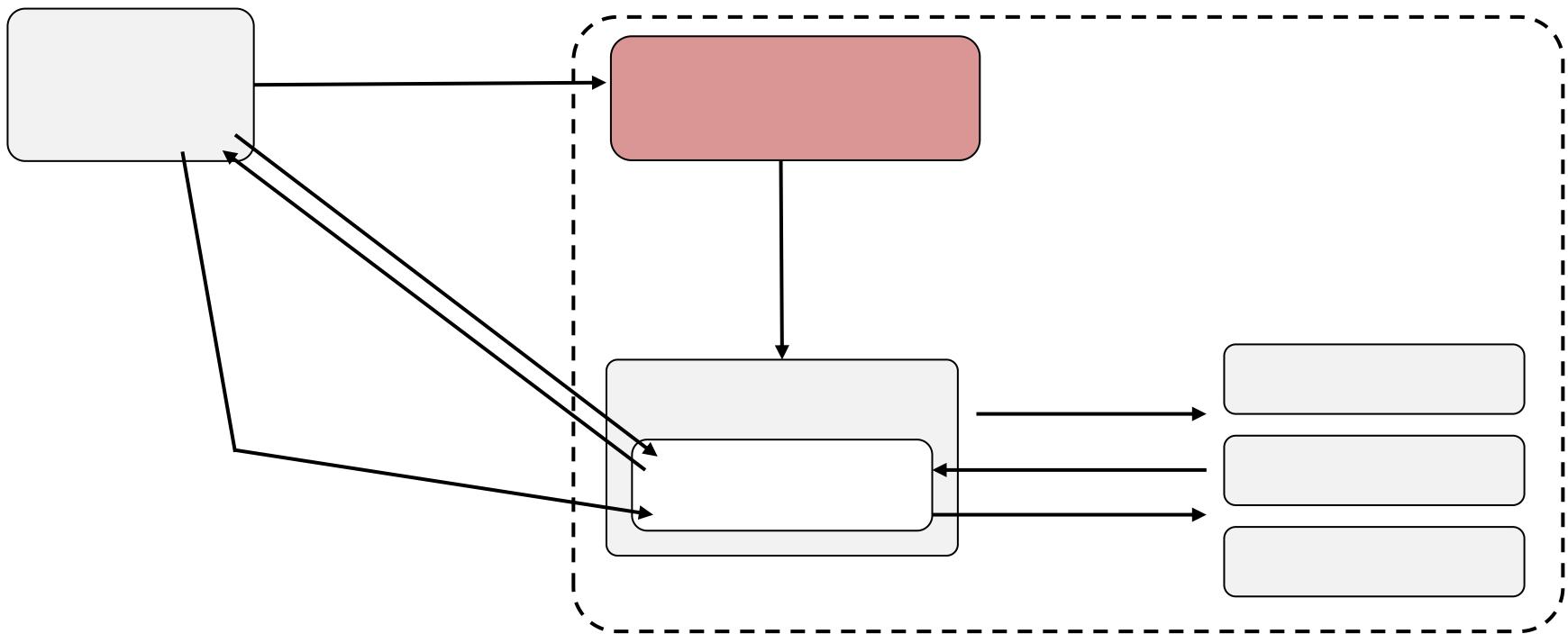
MESOS

















dataArtisans





•

•

•

•



•

•

•

•



•

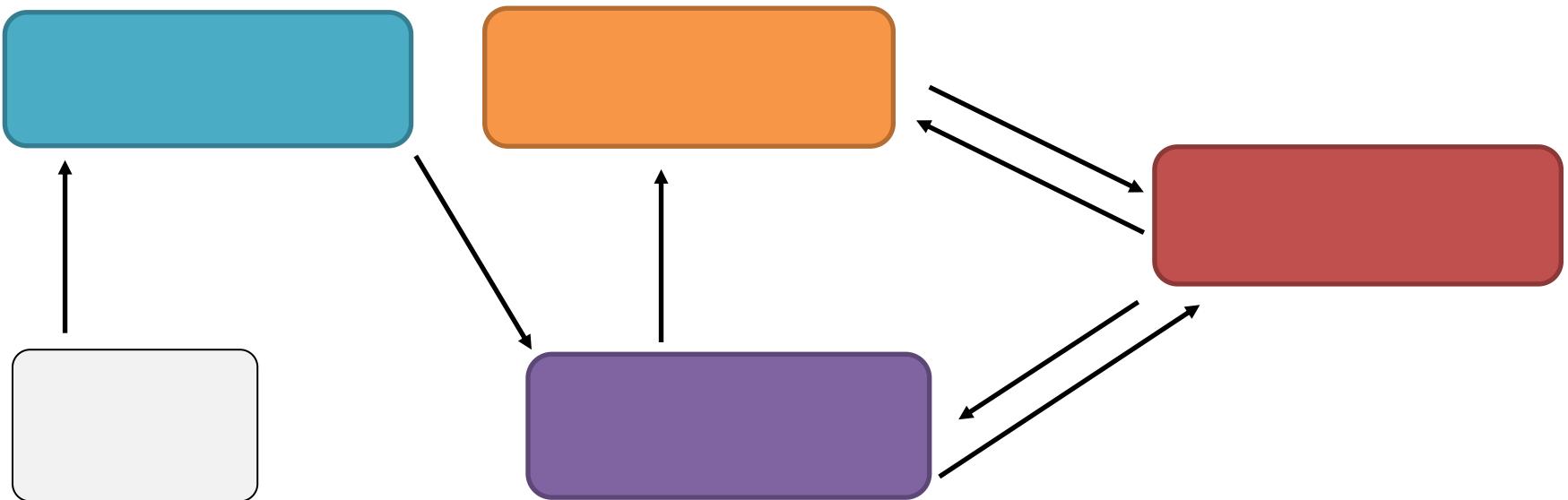
•

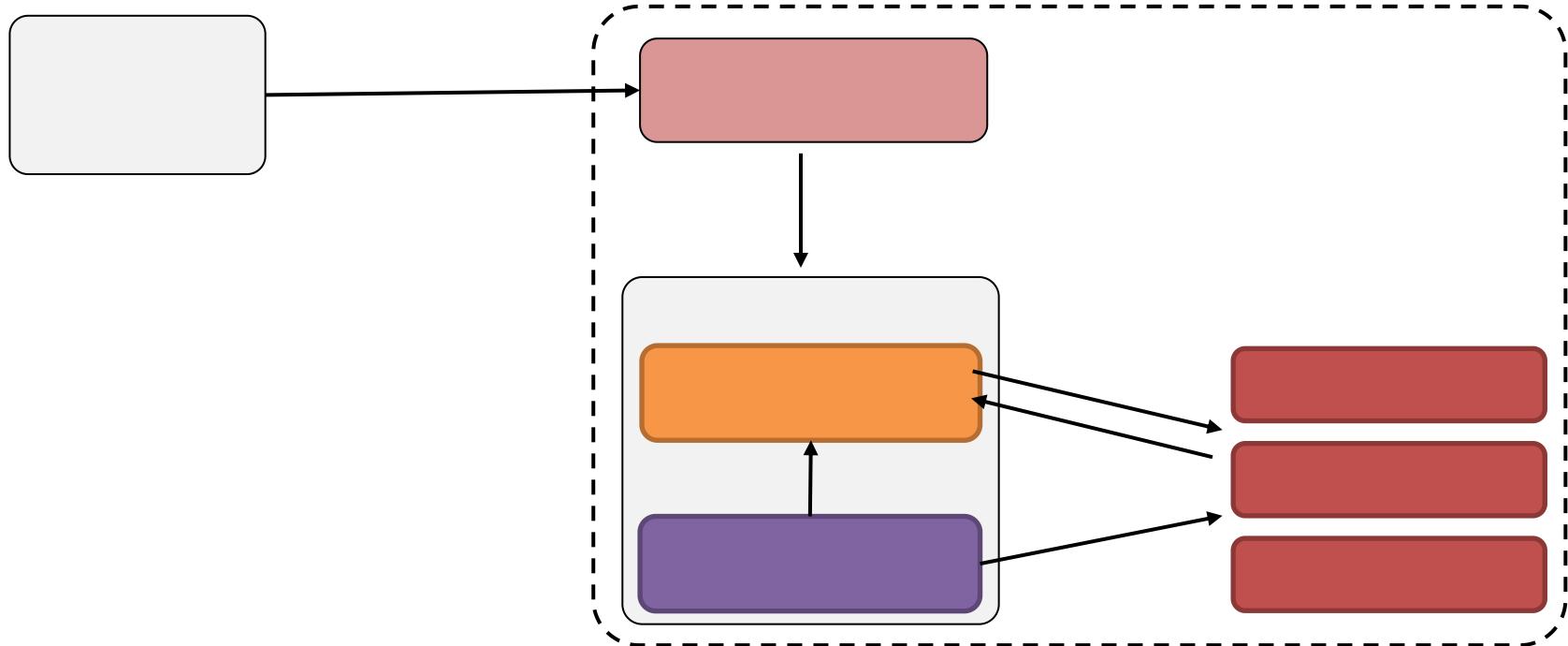
•



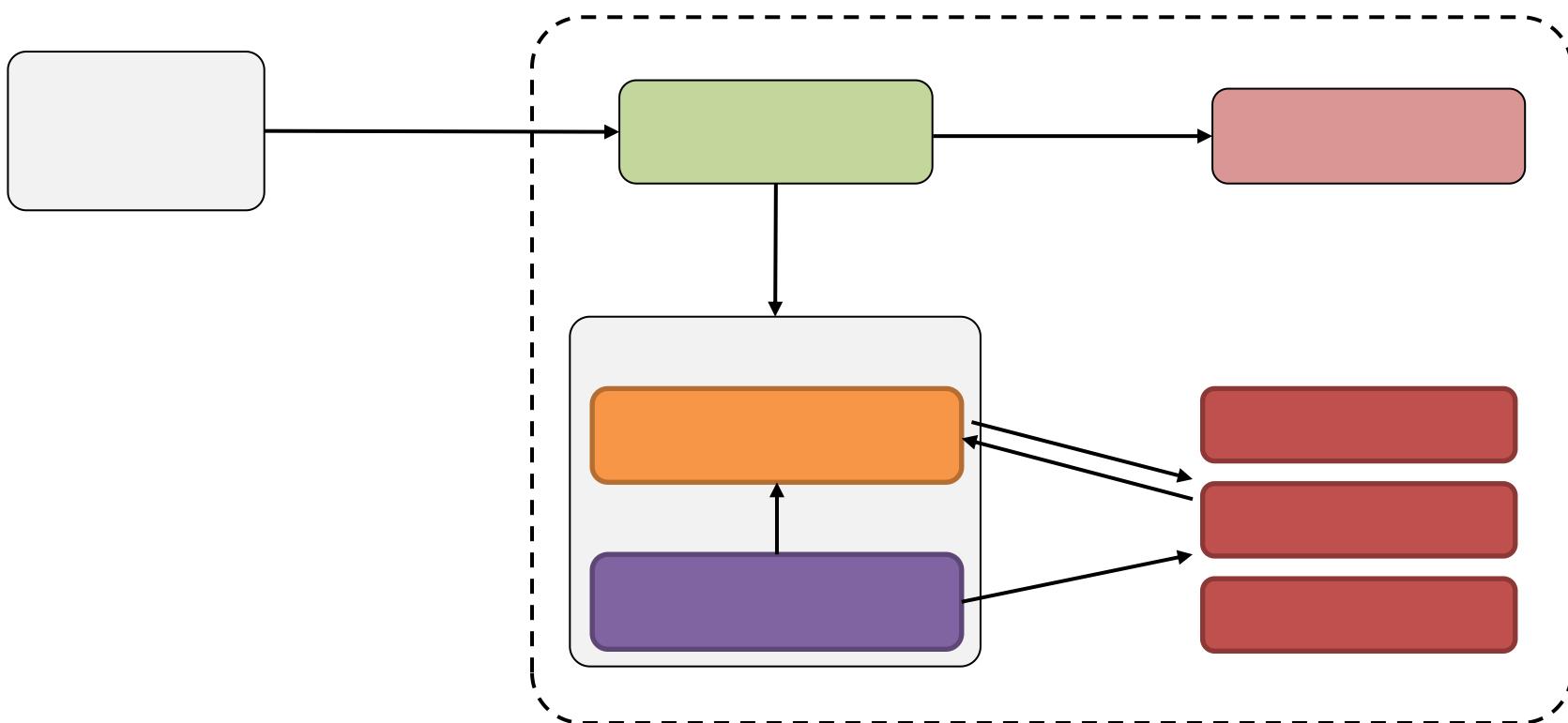
•

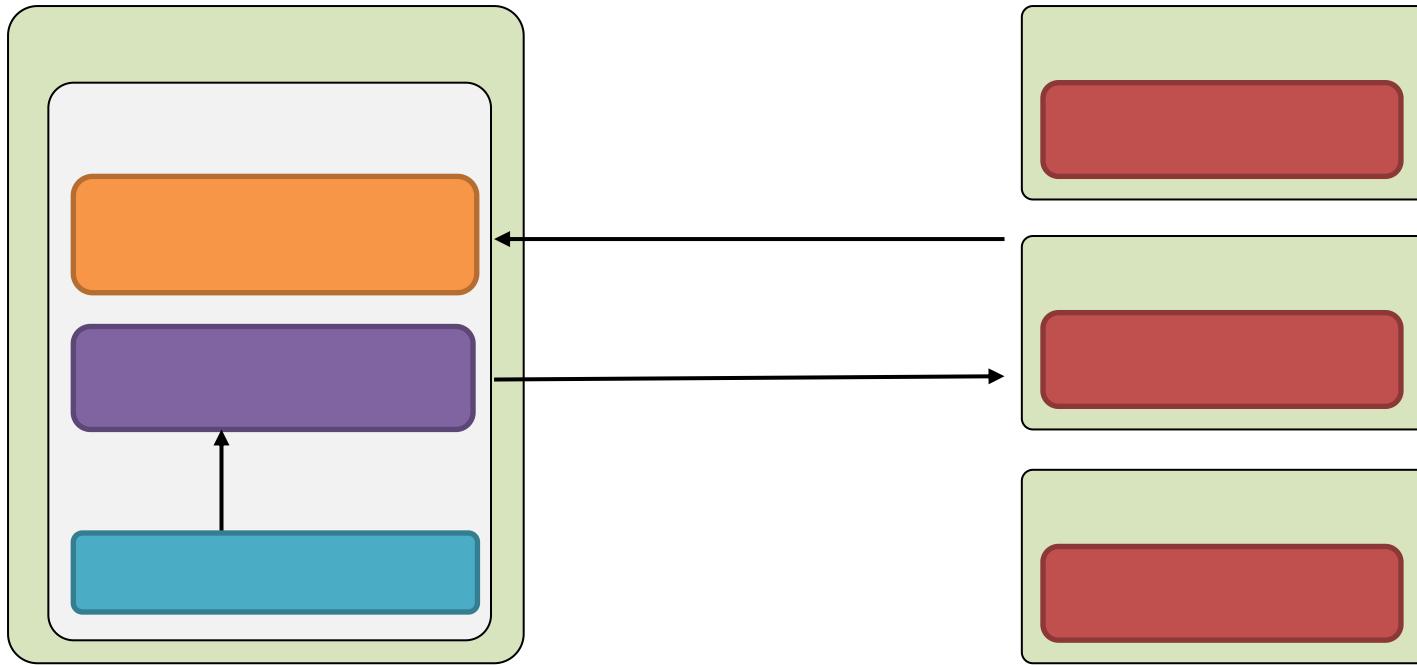
•





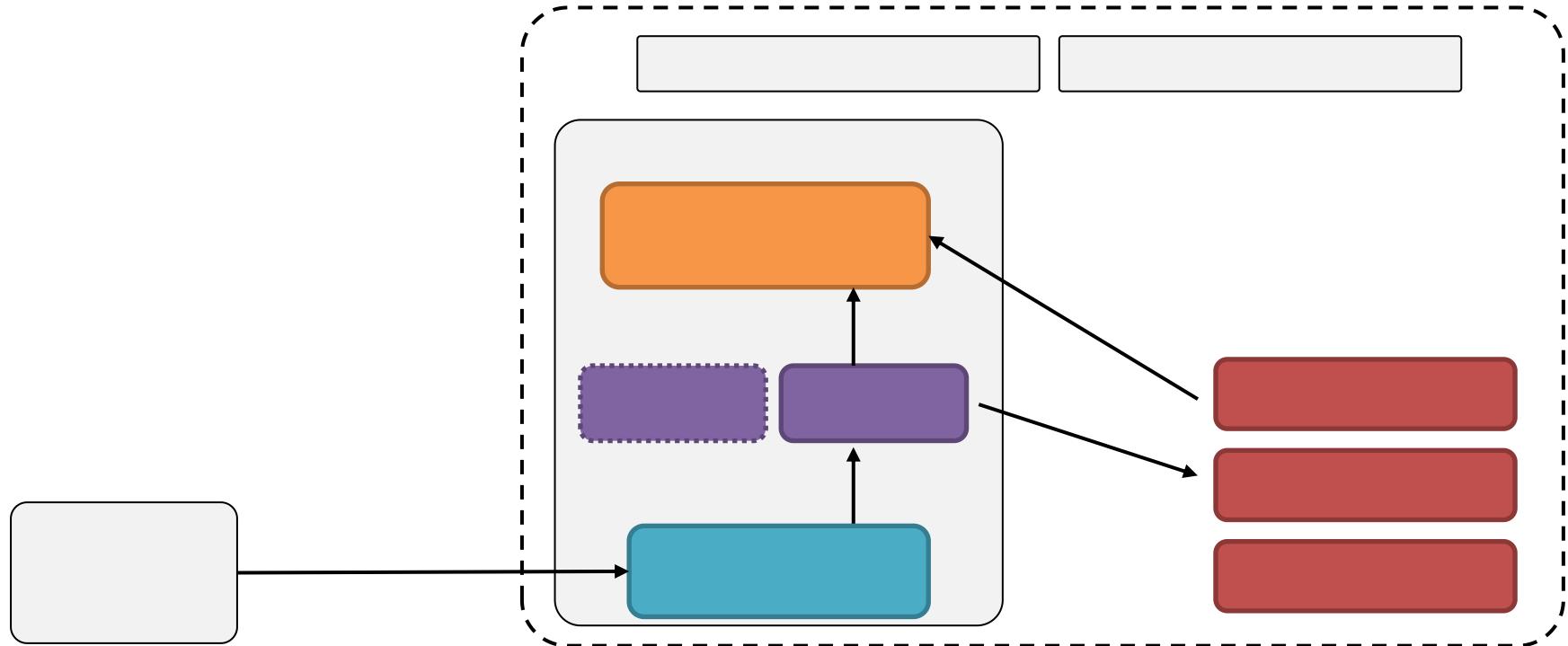


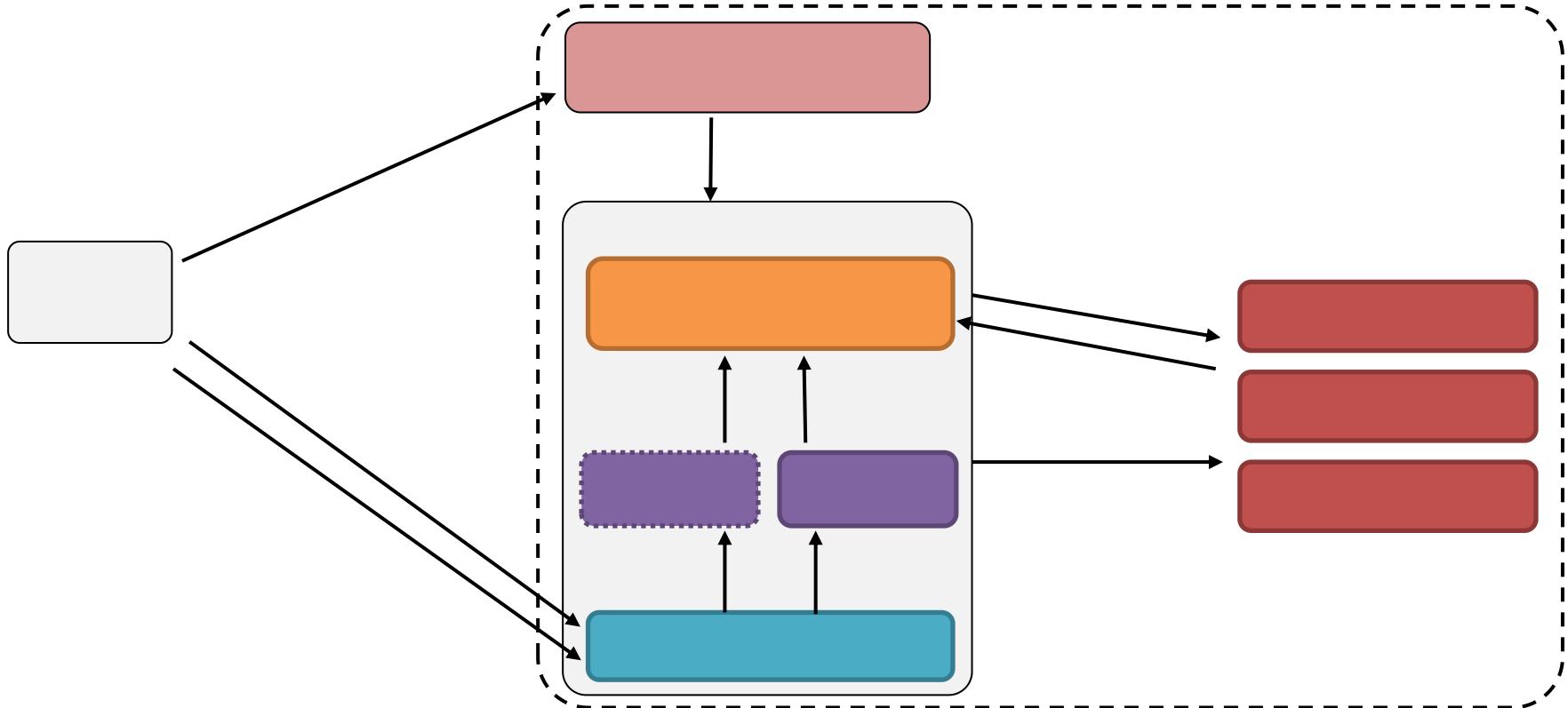




















O'REILLY®



Stream Processing with Apache Flink

FUNDAMENTALS, IMPLEMENTATION, AND OPERATION
OF STREAMING APPLICATIONS

Fabian Hueske & Vasiliki Kalavri

dataArtisans

Runtime Improvements in Blink for Large Scale Streaming at Alibaba

Feng Wang

Zhijiang Wang

April, 2017

Outline

1 Blink Introduction

2 Improvements to Flink Runtime

3 Future Plans

Blink Introduction

Section 1

Blink – Alibaba's version of Flink

✓ Looked into Flink two since 2 years ago

- best choice of unified computing engine
- a few of issues in flink that can be problems for large scale applications

✓ Started Blink project

- aimed to make Flink work reliably and efficiently at the very large scale at Alibaba

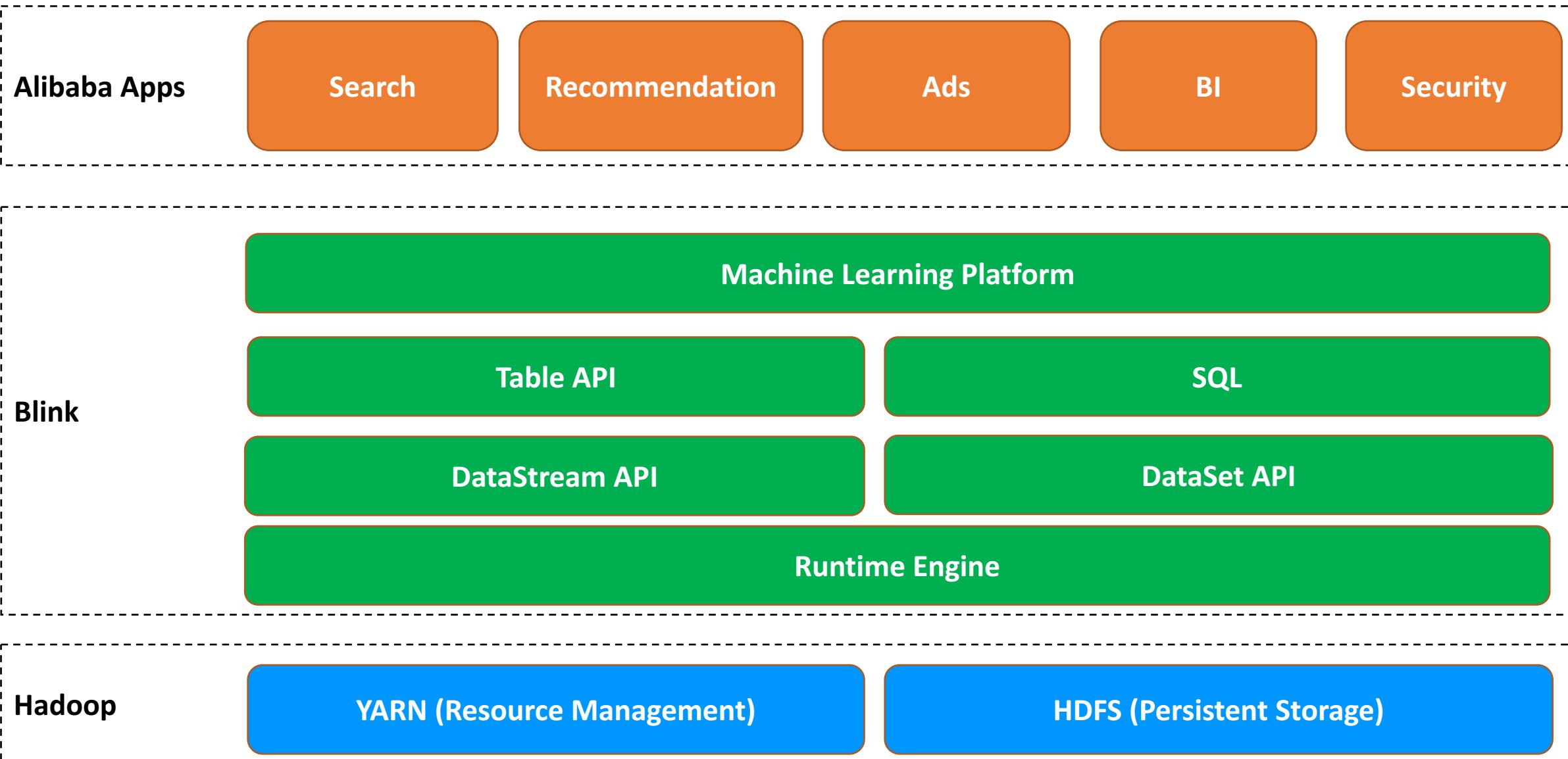
✓ Made various improvements in Flink runtime

- Runs natively on yarn cluster
- failover optimizations for fast recovery
- incremental checkpoint for large states
- async operator for high throughputs

✓ Working with Flink community to contribute changes back since last August

- several key improvements
- hundreds of patches

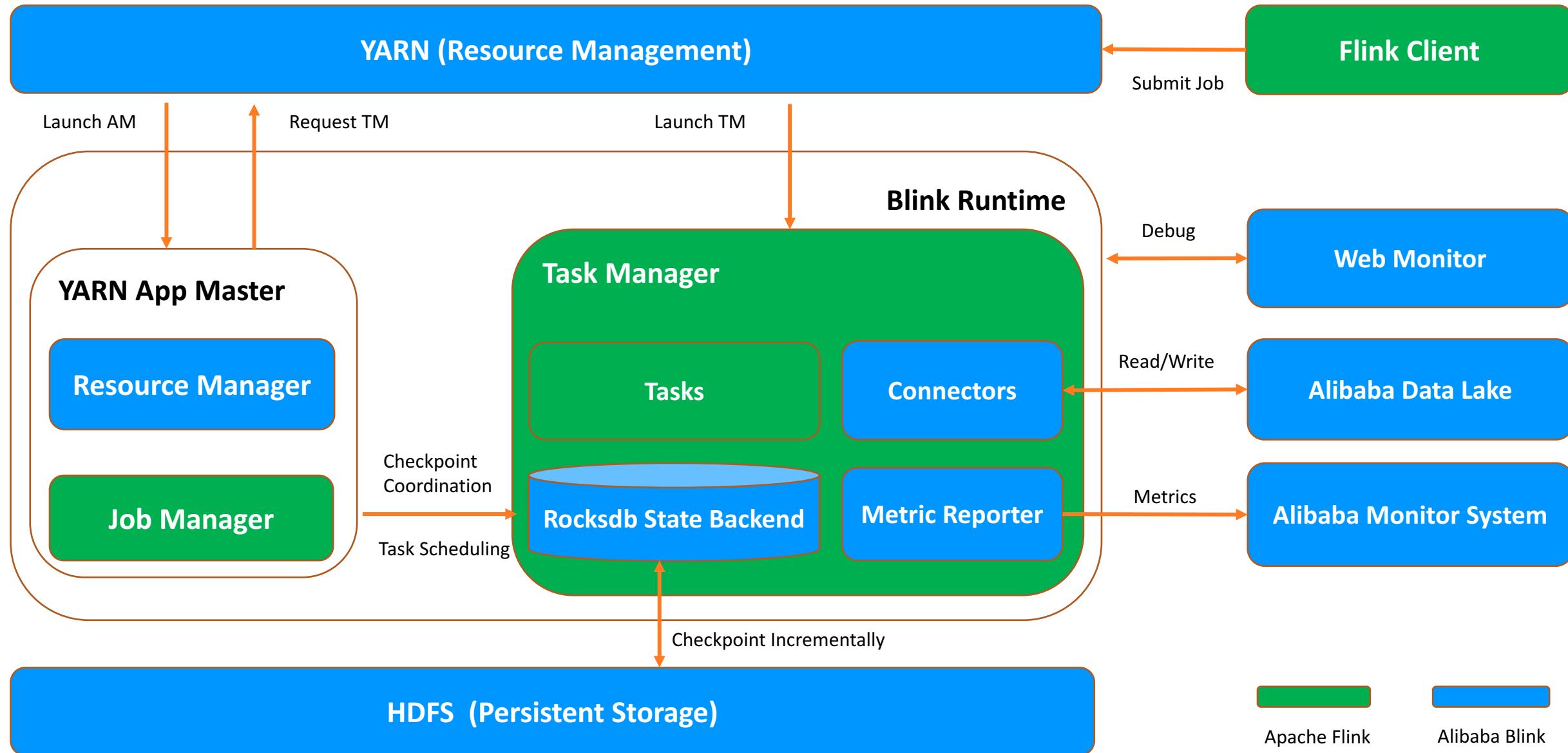
Blink Ecosystem in Alibaba



Blink in Alibaba Production

- ✓ In production for almost one year
- ✓ Run on thousands of nodes
 - hundreds of jobs
 - The biggest cluster is more than 1000 nodes
 - the biggest job has 10s TB states and thousands of subtasks
- ✓ Supported key production services on last Nov 11th, China Single's Day
 - China Single's Day is by far the biggest shopping holiday in China, similar to Black Friday in US
 - Last year it recorded \$17.8 billion worth of gross merchandise volumes in one day
 - Blink is used to do real time machine learning and increased conversion by around 30%

Blink Architecture



Improvements to Flink Runtime

Section 2

Improvements to Flink Runtime

✓ Native integration with Resource Management

- Take YARN for an Example

✓ Performance Improvements

- Incremental Checkpoint
- Asynchronous Operator

✓ Failover Optimization

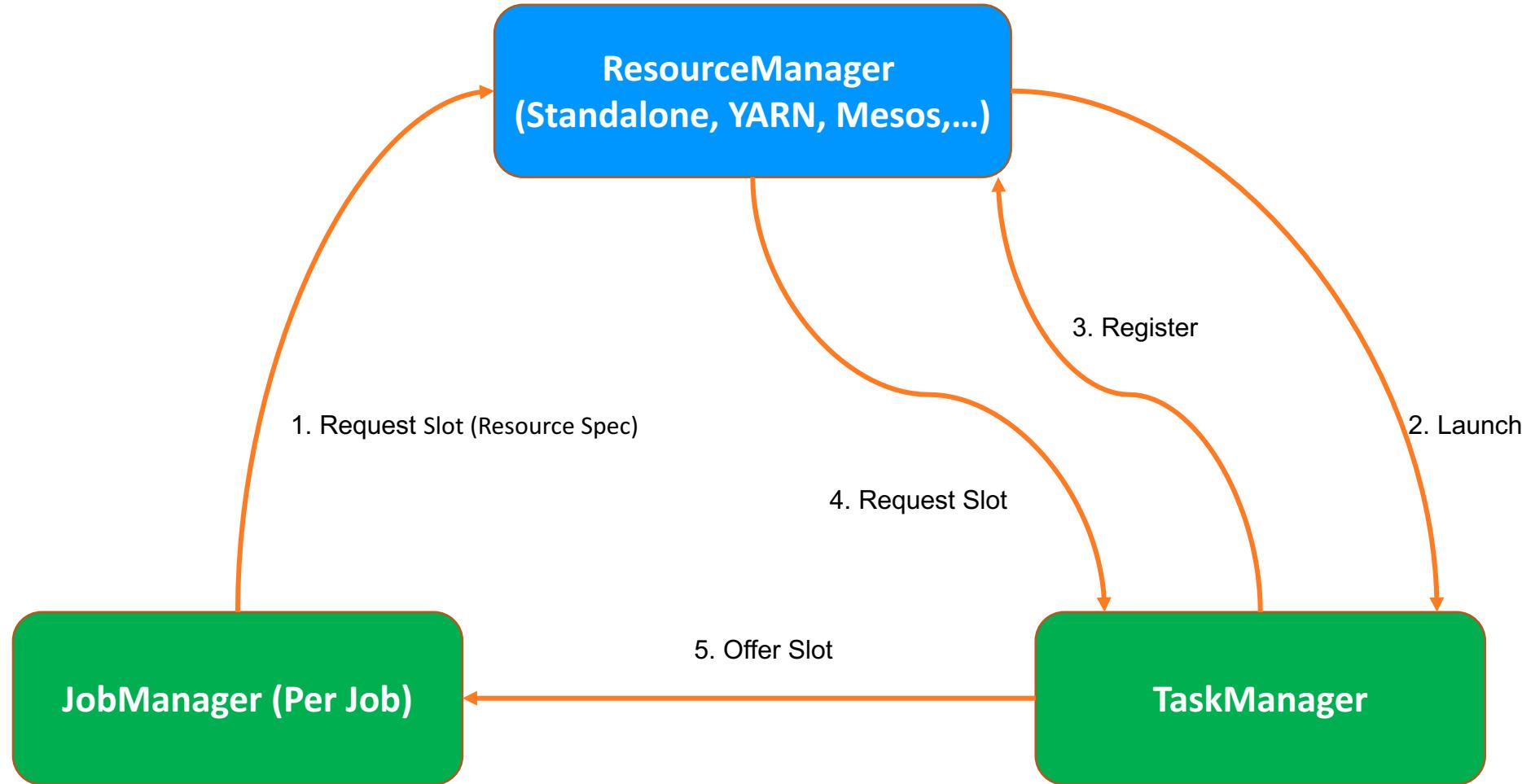
- Fine-grained Recovery for Task Failures
- Allocation Reuse for Task Recovery
- Non-disruptive JobManager failure recovery

Native integration with Resource Management

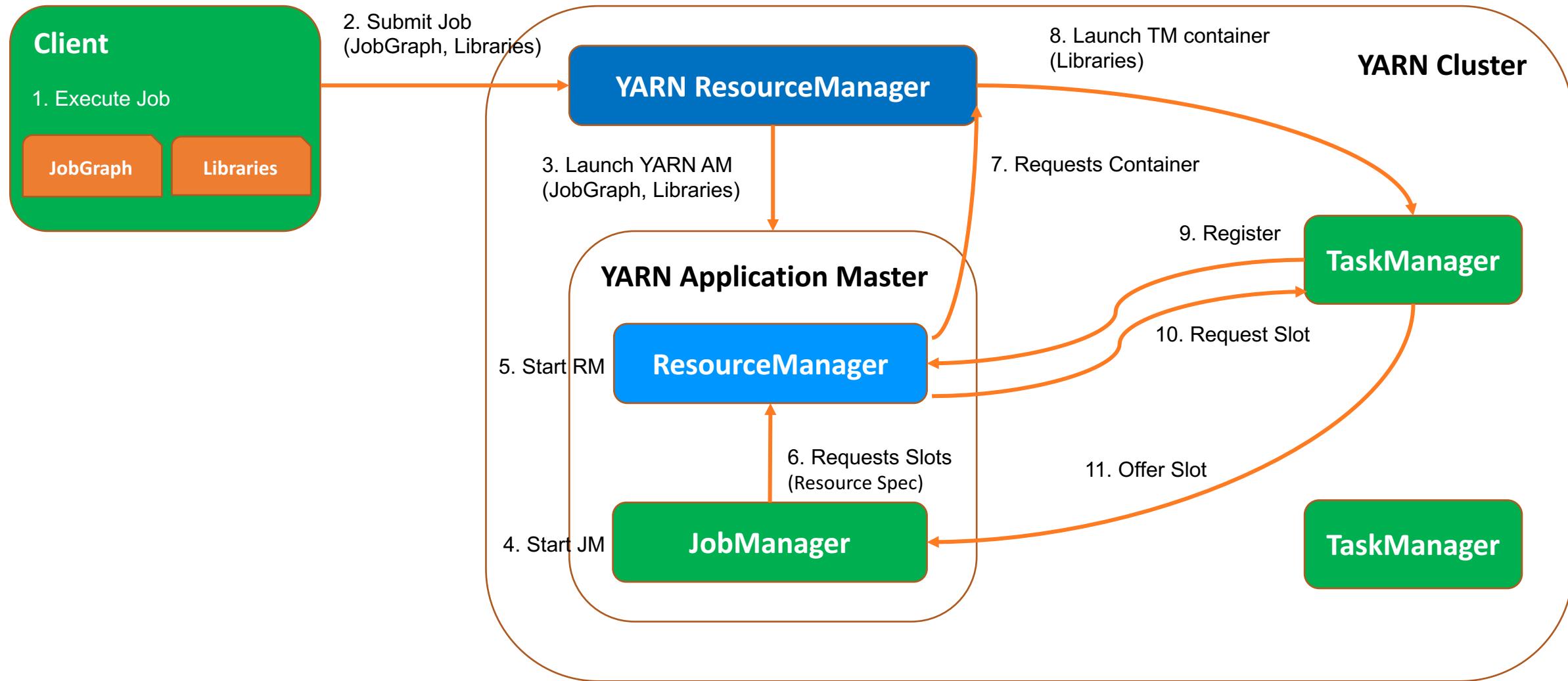
✓ Background

- Cluster resource is allocated upfront. The resource utilization can be not efficient
- A single JobManager handles all the jobs, which limits the scale of the cluster

Native integration with Resource Management



Native integration with YARN



Incremental Checkpoint

✓ Background

- The job state could be very large (many TBs)
- The state size of individual task can be many GBs

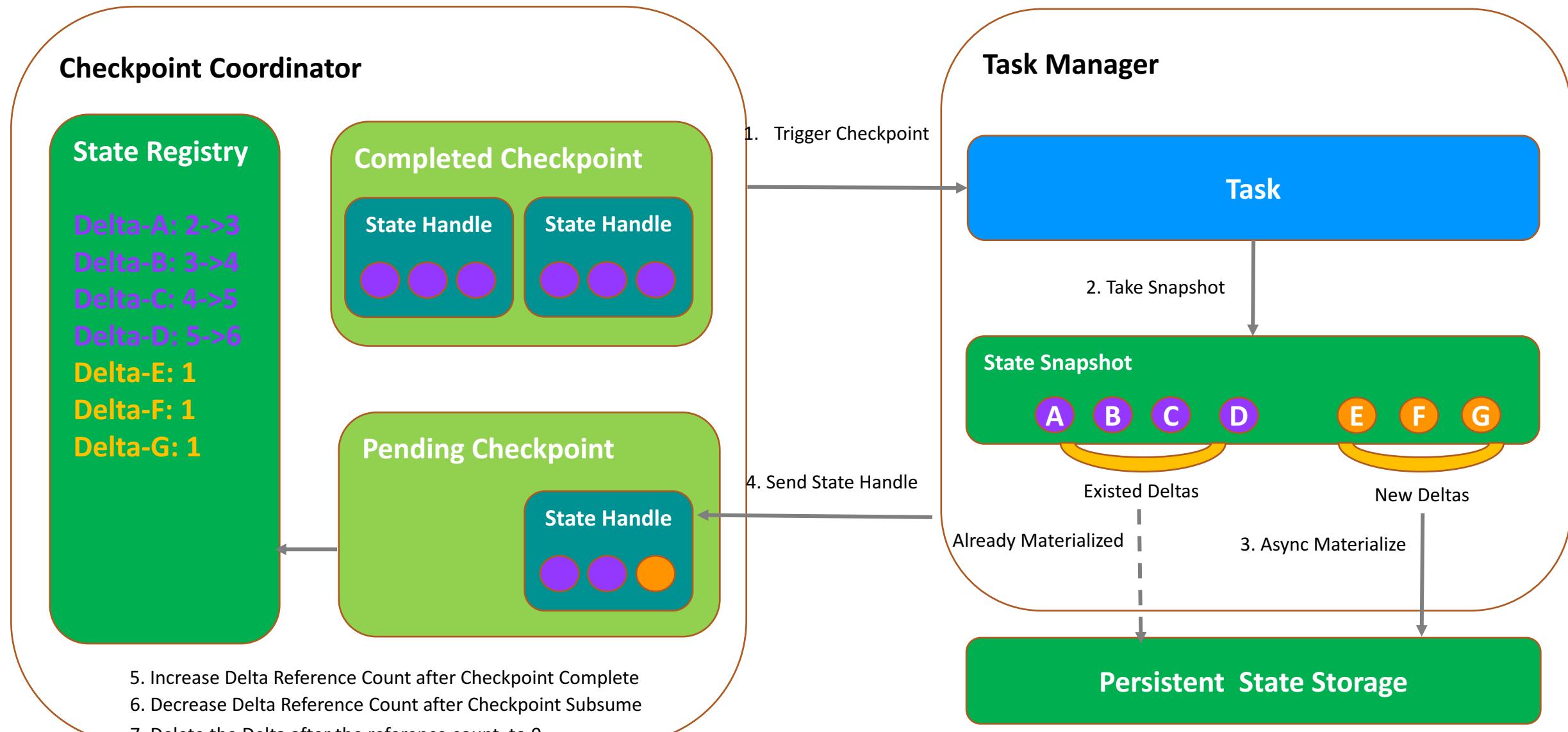
✓ The problems of Full Checkpoint

- Materialize all the states to persistent store at each checkpoint
- As the states get bigger, materialization may take too much time to finish
- One of the biggest blocking issues in large scale production

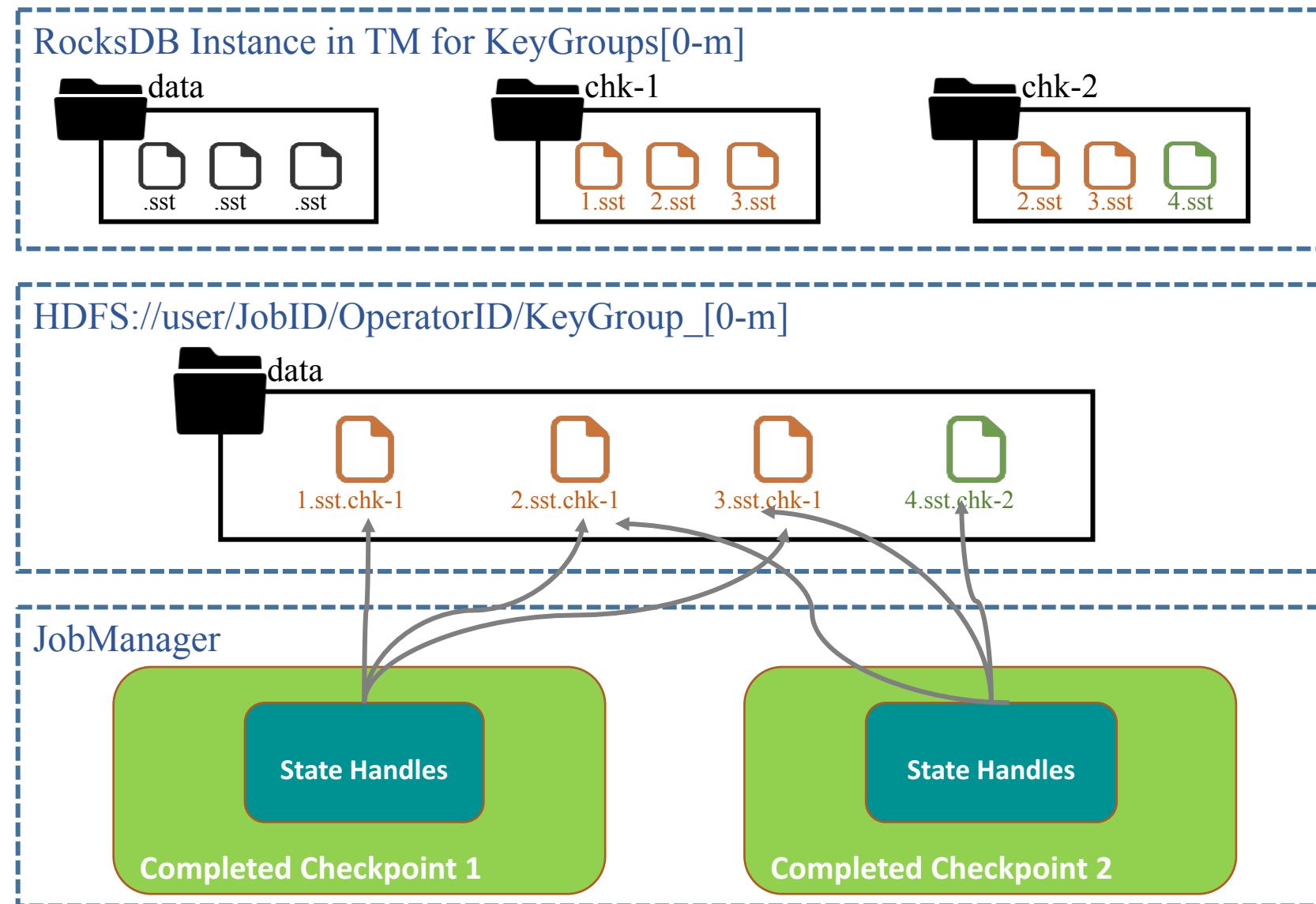
✓ Benefits of Incremental Checkpoint

- Only the modified states since last checkpoint need to be materialized
- The checkpoint will be faster and more efficient

Incremental Checkpoint – How It Works



Incremental Checkpoint - RocksDB State Backend Implementation



Asynchronous Operator

✓ Background

- Flink task use a single thread to process events
- Flink task sometimes need to access external services (hbase, redis,...)

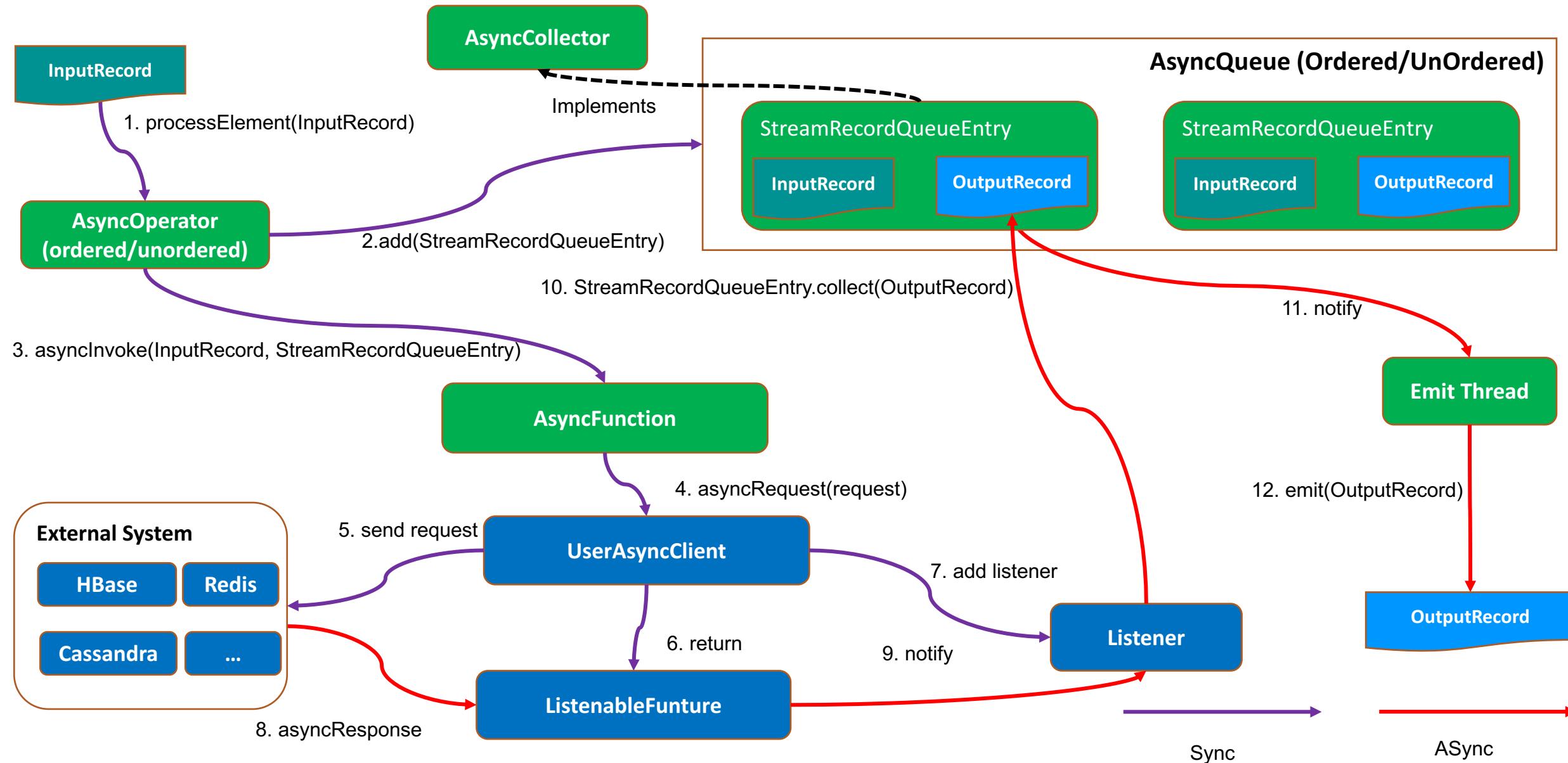
✓ Problem

- The high latency may block the event processing
- Throughput can be limited by latency

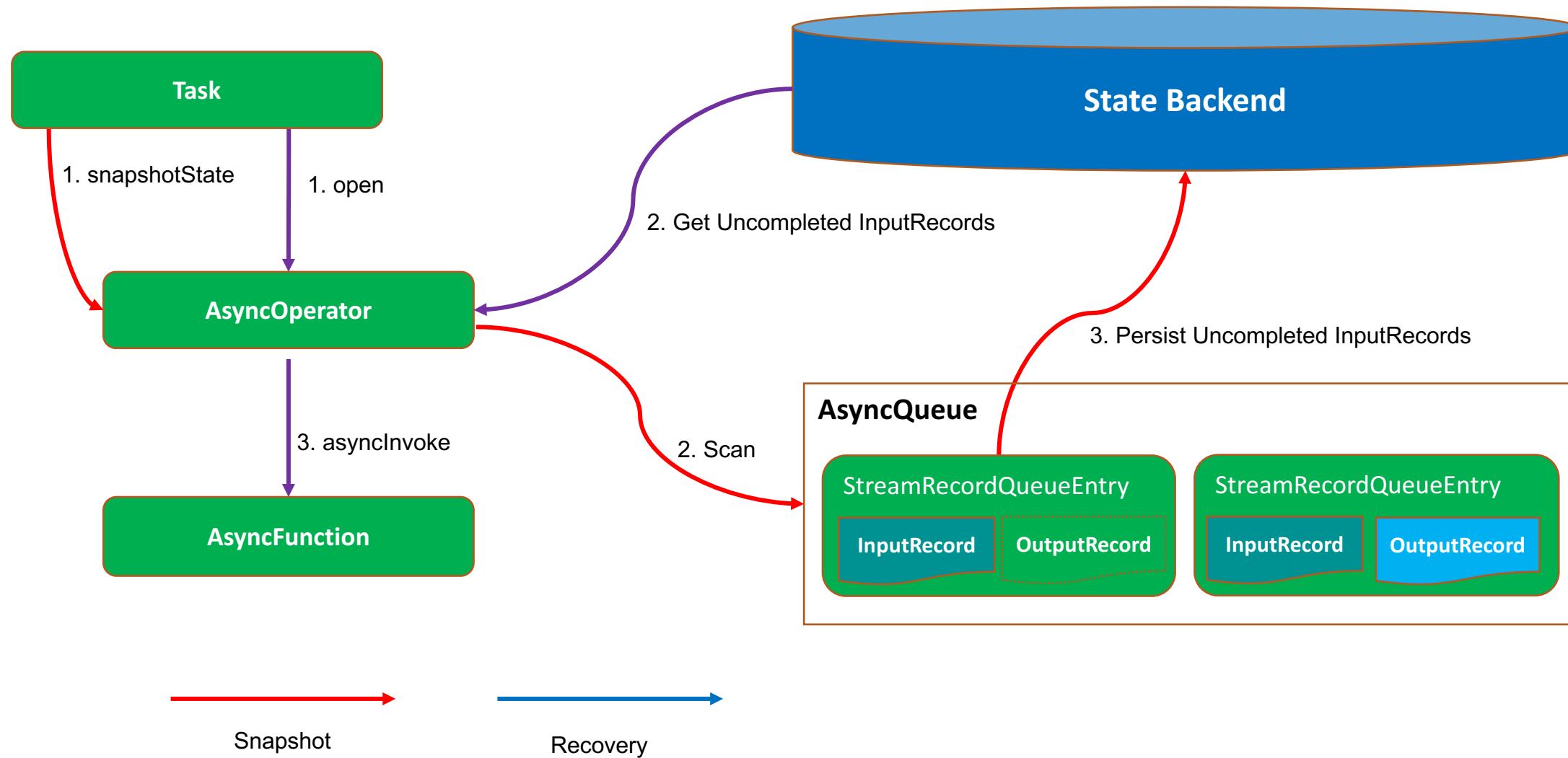
✓ Benefits of Asynchronous Operator

- Decouple the throughput of task from the latency of external system
- Improve the CPU utilization of the Flink tasks
- Simplify resource specification for Flink jobs

Asynchronous Operator – How It Works



Asynchronous Operator – How It Manages State



Fine-grained Recovery from Task Failures

✓ Status of batch job

- Large scale to thousands of nodes
- Node failures are common in large clusters
- Prefer non-pipelined mode due to limited resource

✓ Problems

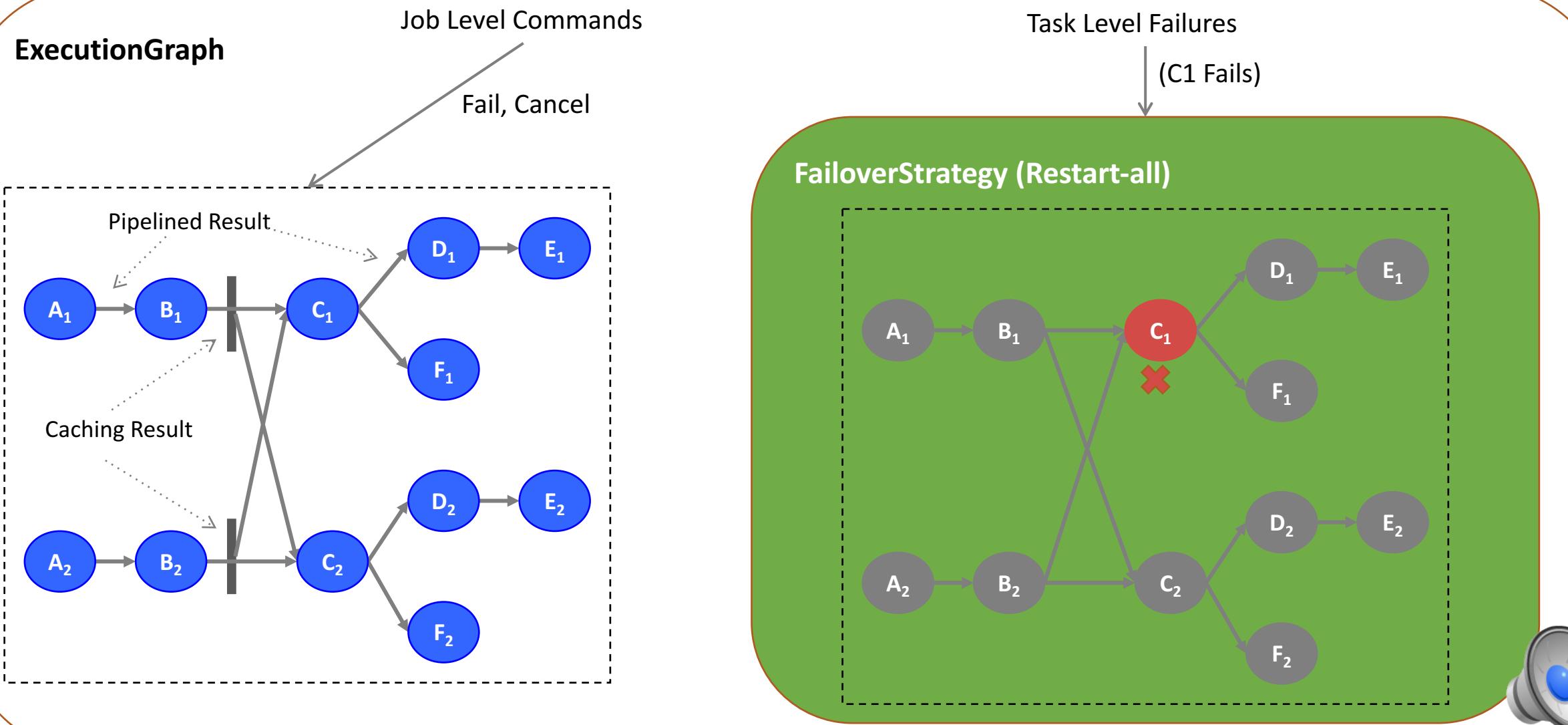
- One task failure needs to restart the entire execution graph
- It is especially critical for batch jobs

✓ Benefit

- Make recovery more efficient by restarting only what needs to be restarted

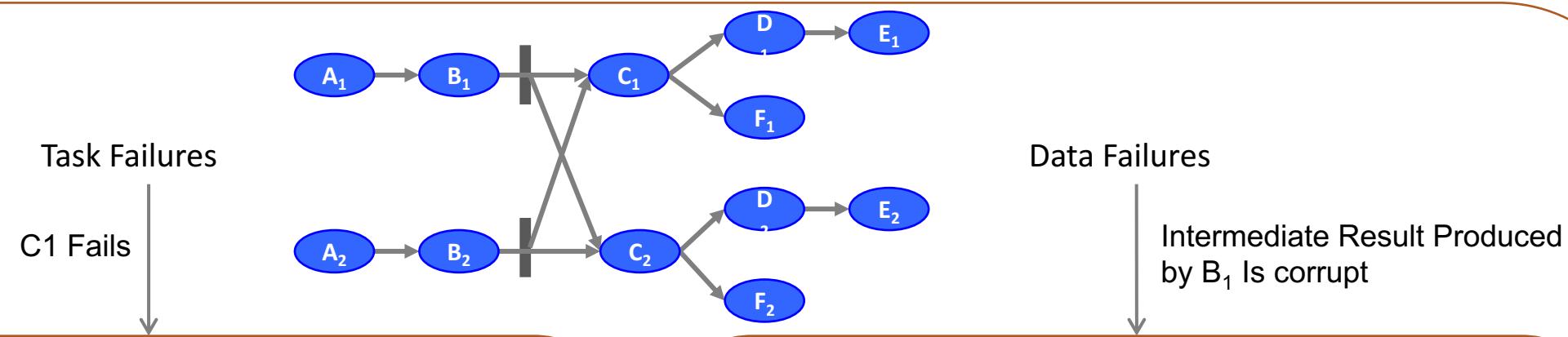


Fine-grained Recovery from Task Failures – Restart-all Strategy

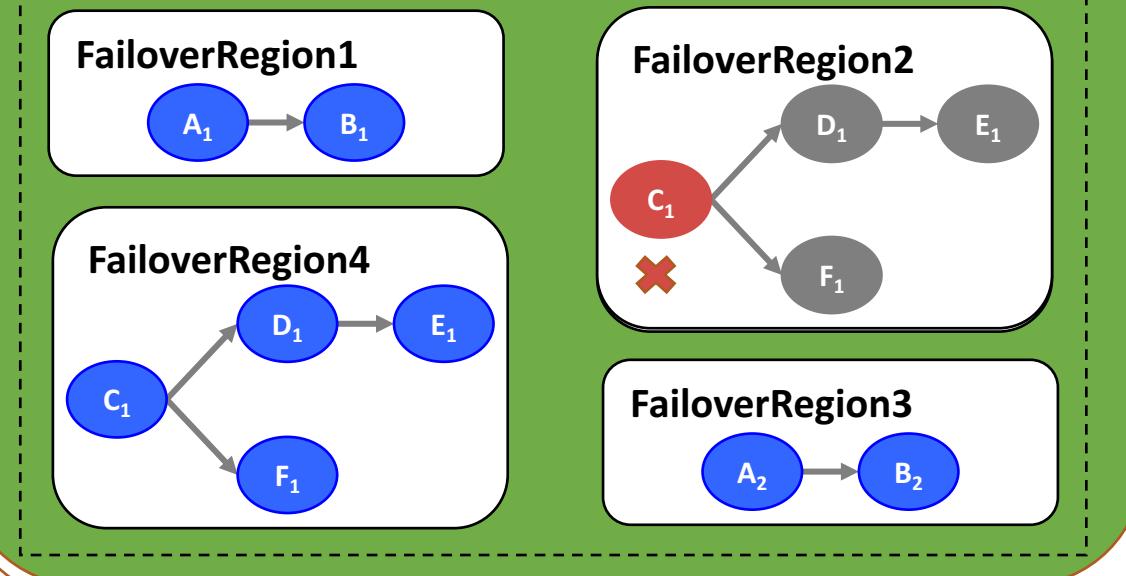


Fine-grained Recovery from Task Failures – Region-based Strategy

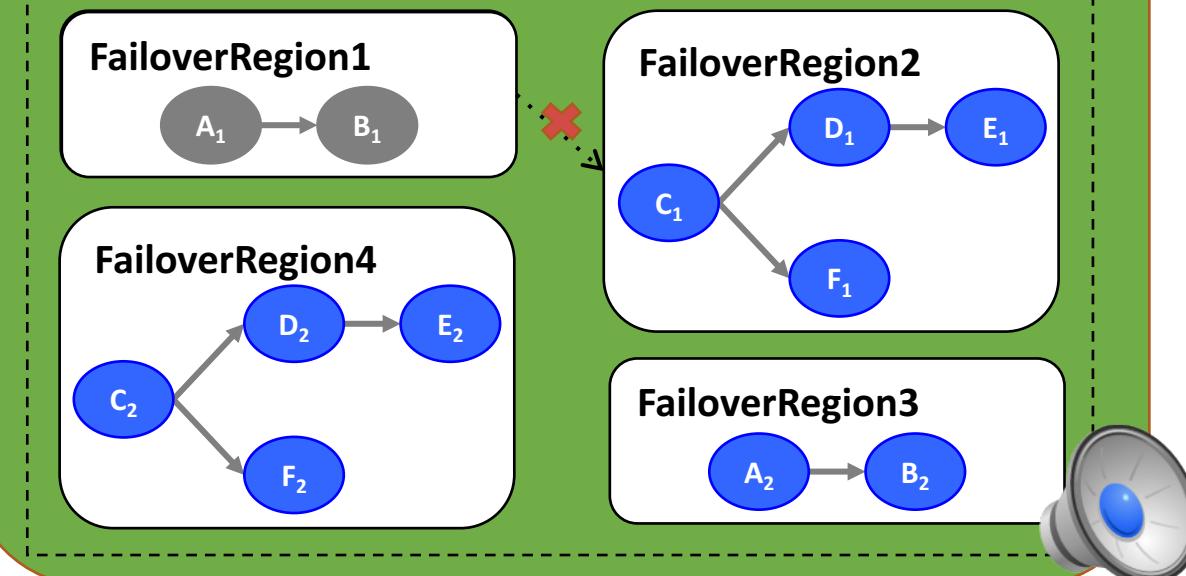
ExecutionGraph



FailoverStrategy (Region-based)



FailoverStrategy (Region-based)



Allocation Reuse for Task Recovery

✓ Background

- The job state can be very big in Alibaba, hence use RocksDB as state backend
- State restore by RocksDB backend involves in copying data from HDFS

✓ Problem

- It is expensive to restore state from HDFS during task recovery

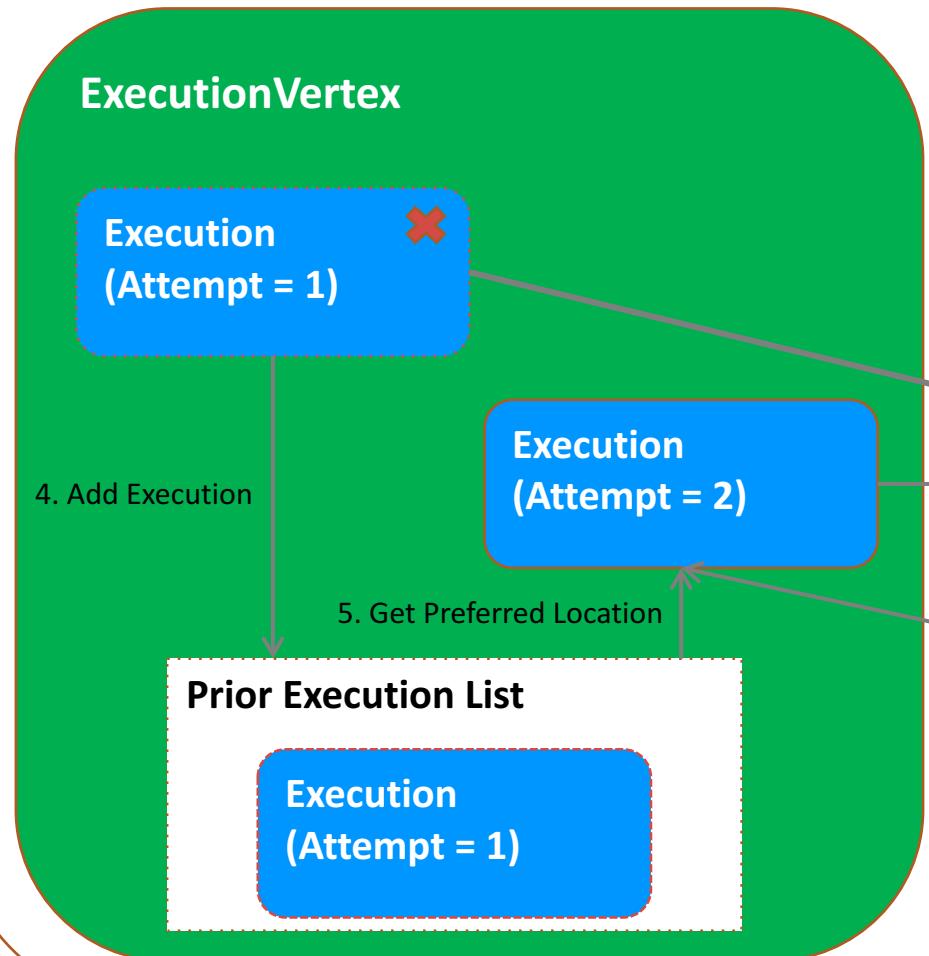
✓ Benefits of Allocation Reuse

- Deploy the restarted task in previous allocation to speed up recovery
- Restore state from local RocksDB to avoid copying data from HDFS



Allocation Reuse for Task Recovery – How It Works

JobManager



Non-disruptive JobManager Failures via Reconciliation

✓ Background

- The job is large scale to thousands of nodes
- The job state can be very big in TB level

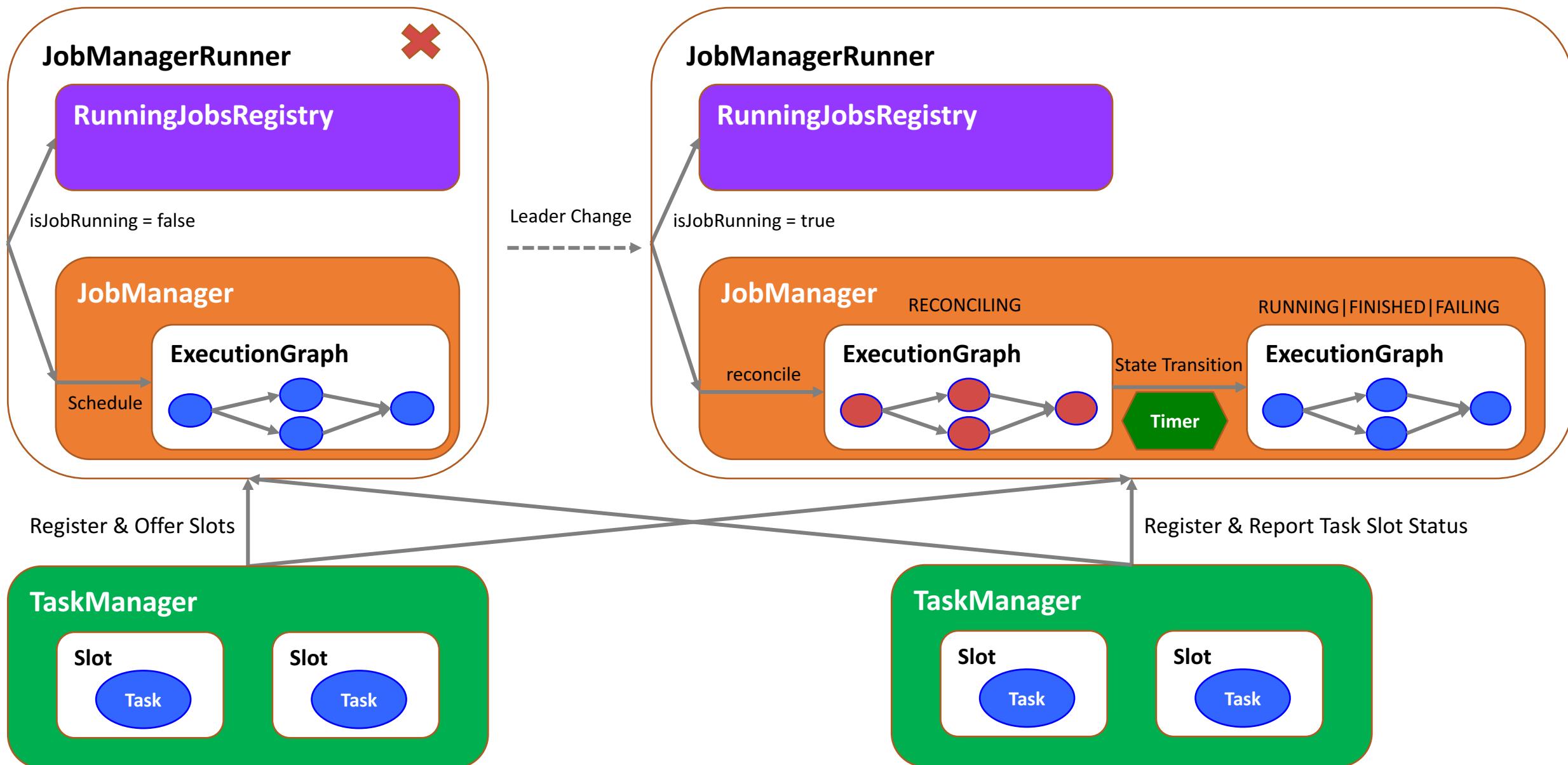
✓ Problems

- All the tasks are restarted for job manager failures
- The cost is high for recovering states

✓ Benefit

- Improve job stability by automatic reconciliation to avoid restarting tasks

Non-disruptive JobManager Failures via Reconciliation – How It works



Future Plans

Section 3

Future Plans

✓ Blink is already popular in the streaming scenarios

- more and more streaming applications will run on blink

✓ Make batch applications run on production

- increase the resource utilization of the clusters

✓ Blink as Service

- Alibaba Group Wide

✓ Cluster is growing very fast

- cluster size will double
- thousands of jobs run on production

Thanks

We are Hiring!

aliserach-recruiting@list.alibaba-inc.com

Pravega

Storage Reimagined for a Streaming World

Srikanth Satya & Tom Kaitchuck

Dell EMC Unstructured Storage

srikanth.satya@emc.com

tom.kaitchuck@emc.com

Streaming is Disruptive

How do you **shrink to zero** the time it takes to turn

Stateful processors born for streaming, like
Apache Flink, are **disrupting** how we think
about **data computing** ...

We think the world needs a complementary
technology ... to similarly **disrupt storage**.

- Ability to deliver **accurate results** processing continuously even with late arriving or out of order data

Introducing Pravega Streams

A new storage abstraction – a **stream** – for continuous and infinite data

- Named, durable, append-only, **infinite** sequence of bytes
- With low-latency appends to and reads from the tail of the sequence
- With high-throughput reads for older portions of the sequence

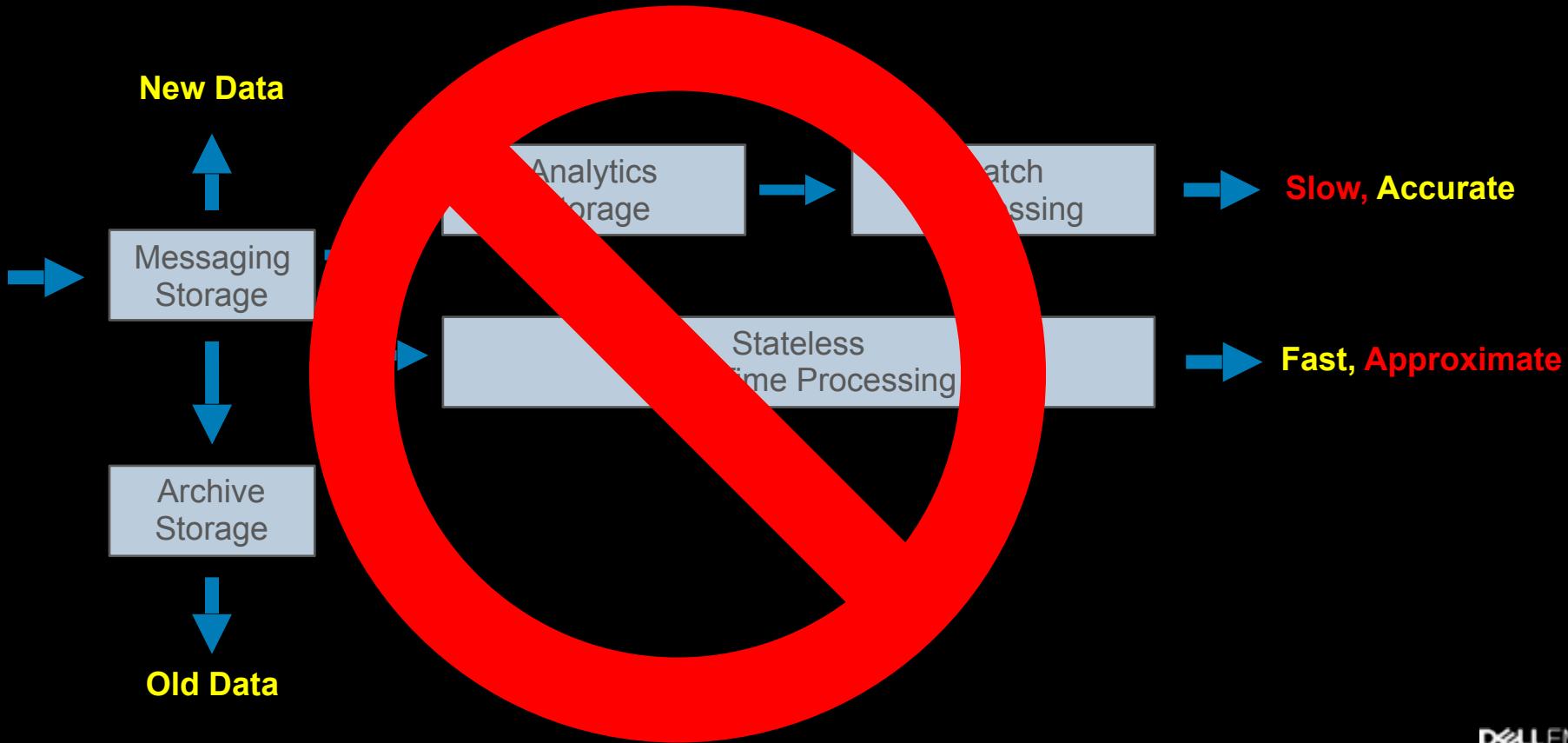
Coordinated scaling of stream storage and stream processing

- Stream writes partitioned by app key
- Stream reads independently and automatically partitioned by arrival rate SLO
- Scaling protocol to allow stream processors to scale in lockstep with storage

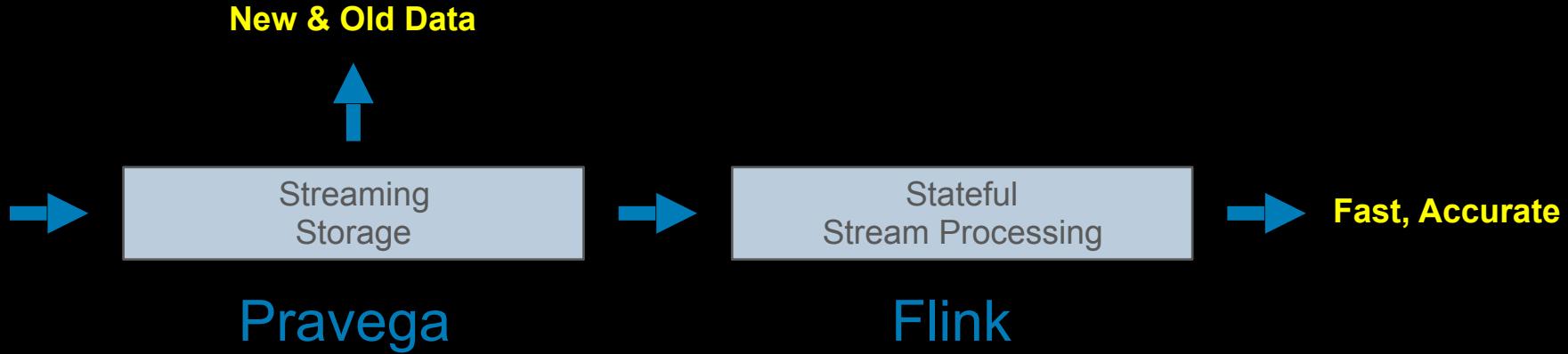
Enabling **system-wide exactly once** processing across multiple apps

- Streams are ordered and strongly consistent
- Chain independent streaming apps via streams
- Stream transactions integrate with checkpoint schemes such as the one used in Flink

In Place of All This ...

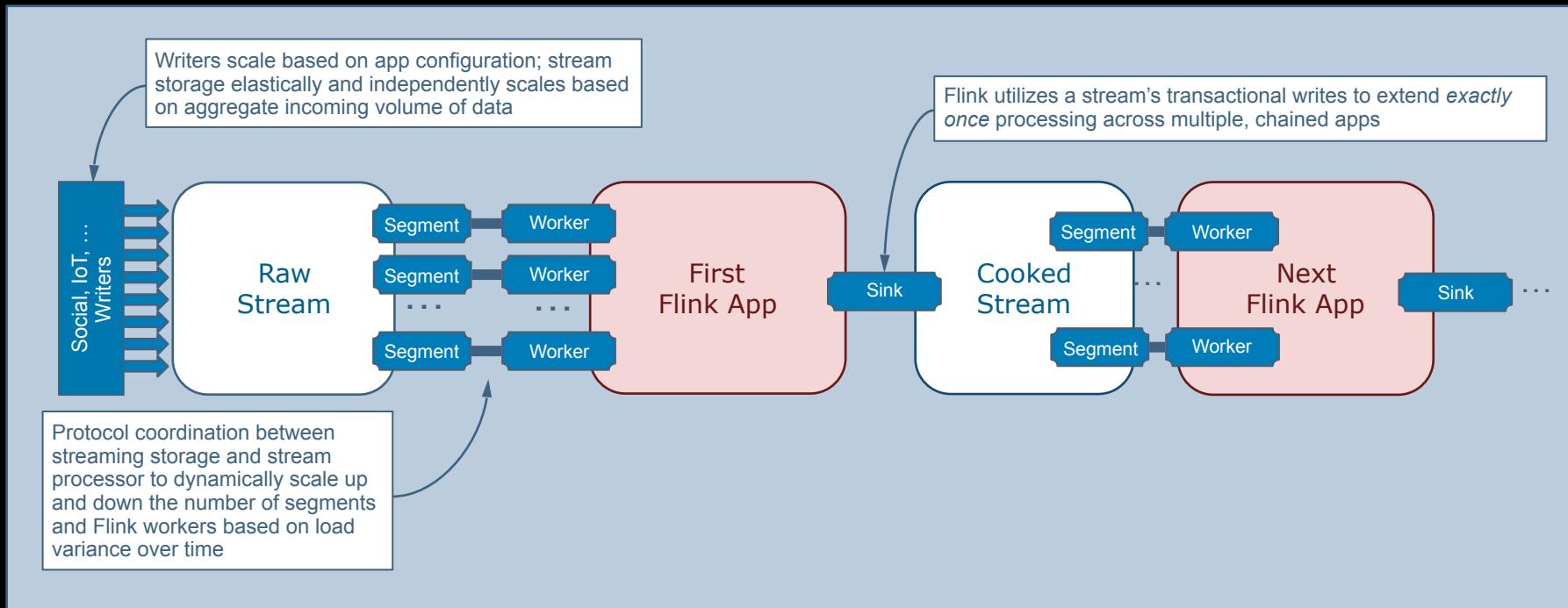


... Just Do This!



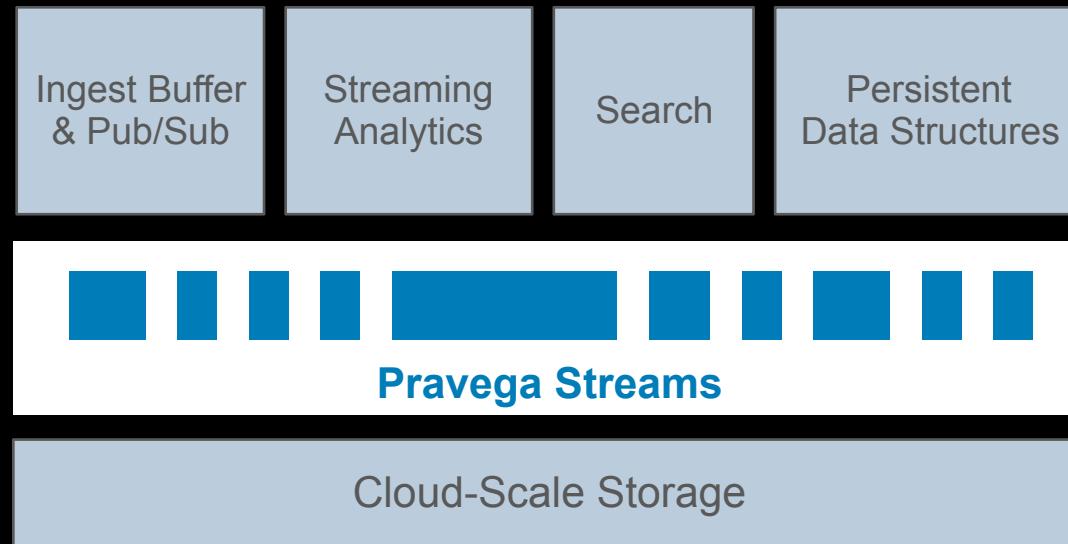
Each component in the combined system – writers, streams, readers, apps – is independently, elastically, and dynamically scalable in coordination with data volume arrival rate over time. Sweet!

Pravega Streams + Flink



And It's Just the Beginning ...

*Enabling a new generation of distributed middleware
reimagined as streaming infrastructure*



How Pravega Works

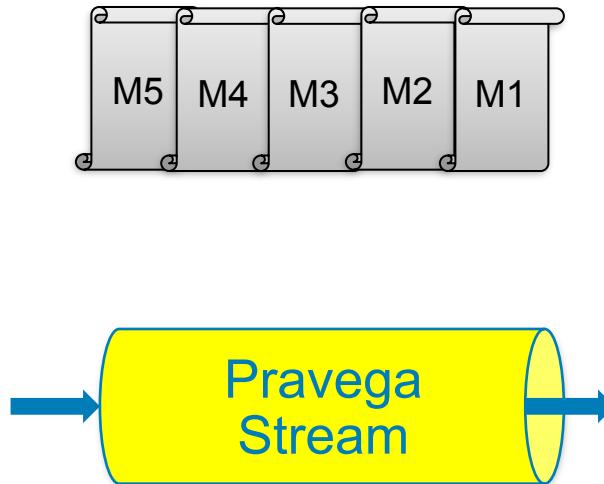
Architecture & System Design

Pravega Architecture Goals

- All data is durable
 - Data is replicated and persisted to disk before being acknowledged
- Strict ordering guarantees and exactly once semantics
 - Across both tail and catch-up reads
 - Client tracks read offset, Producers use transactions
- Lightweight, elastic, infinite, high performance
 - Support tens of millions of streams
 - Low (<10ms) latency writes; throughput bounded by network bandwidth
 - Read pattern (e.g. many catch-up reads) doesn't affect write performance
- Dynamic partitioning of streams based on load and throughput SLO
- Capacity is not bounded by the size of a single node

Streaming model

- Fundamental data structure is an ordered sequence of bytes
- Think of it as a durable socket or Unix pipe
- Bytes are not interpreted server side
- This implicitly guarantees order and non-duplication
- Higher layers impose further structure, e.g. message boundaries

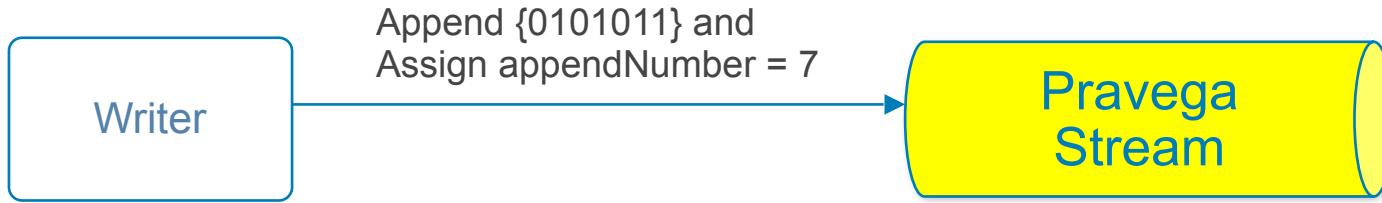


Cartoon API

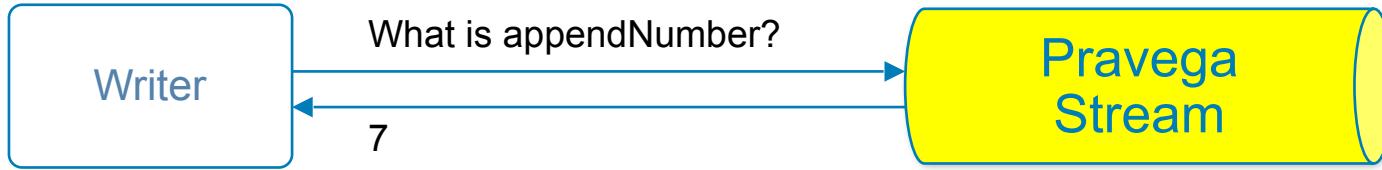
```
public interface SegmentWriter {  
  
    /** Asynchronously and atomically write data */  
  
    void write(ByteBuffer data);  
  
    /** Asynchronously and atomically write the  
     * data if it can be written at the provided offset */  
  
    void write(ByteBuffer data, long atOffset);  
  
    /** Asynchronously and atomically write all of  
     * the data from the provided input stream */  
  
    void write(InputStream in);  
  
}
```

```
public interface SegmentReader {  
  
    long fetchCurrentLength();  
  
    /** Returns the current offset */  
  
    long getOffset();  
  
    /** Sets the next offset to read from */  
  
    void setOffset(long offset);  
  
    /** Read bytes from the current offset */  
  
    ByteBuffer read(int length);  
  
}
```

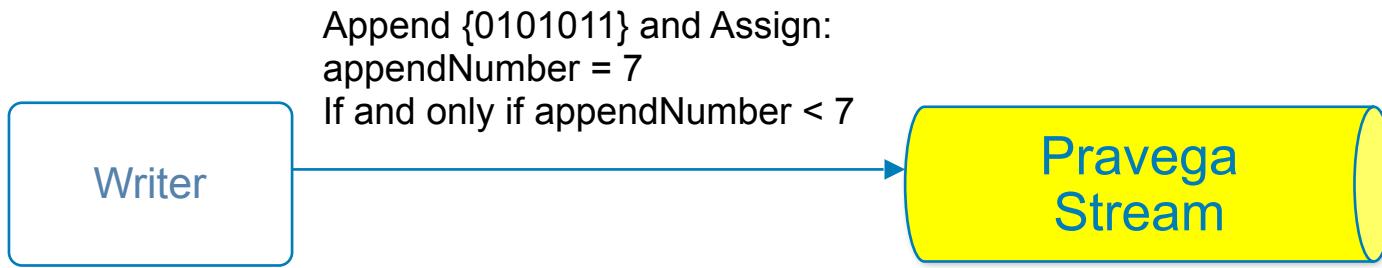
Idempotent Append



Idempotent Append



Idempotent Append



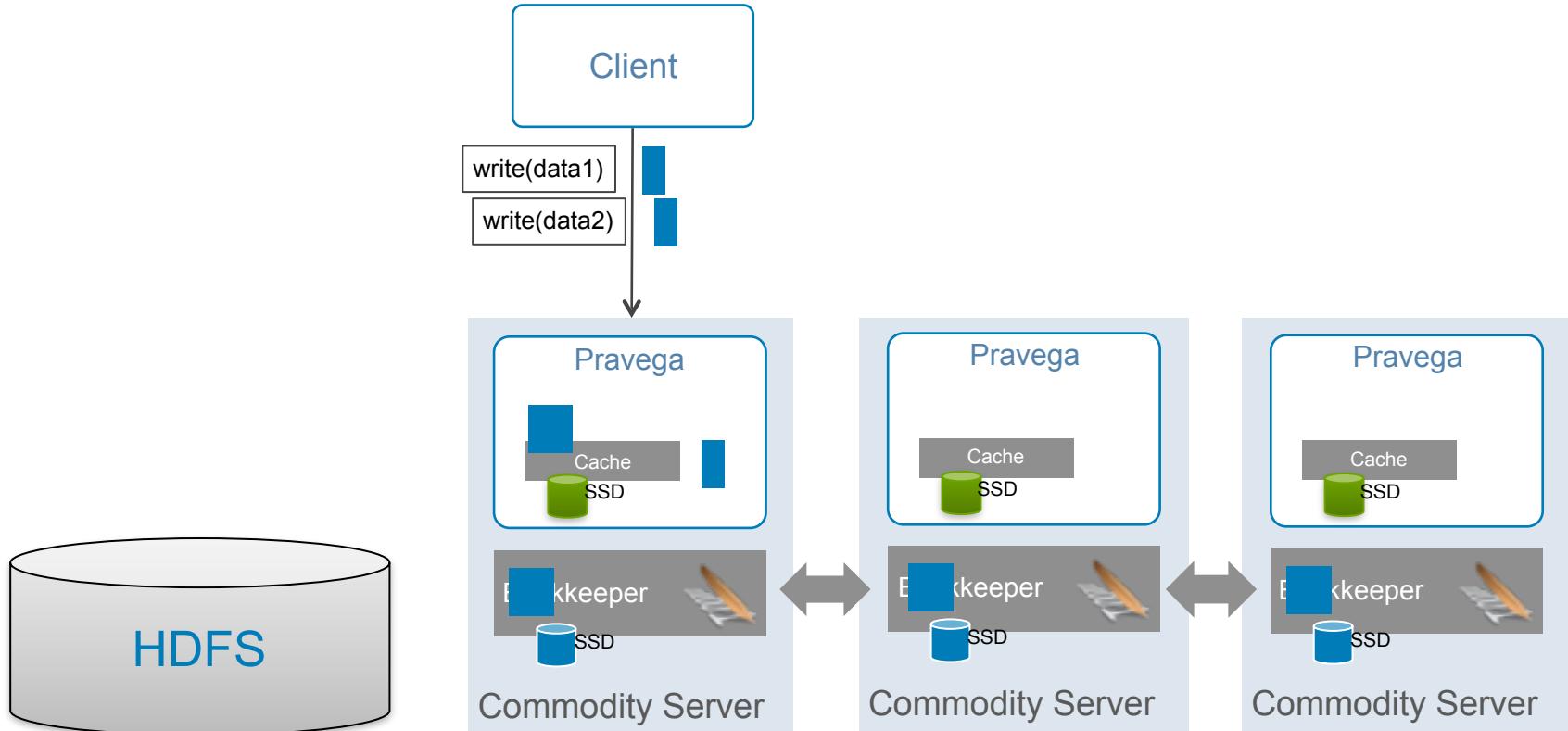
Idempotent output



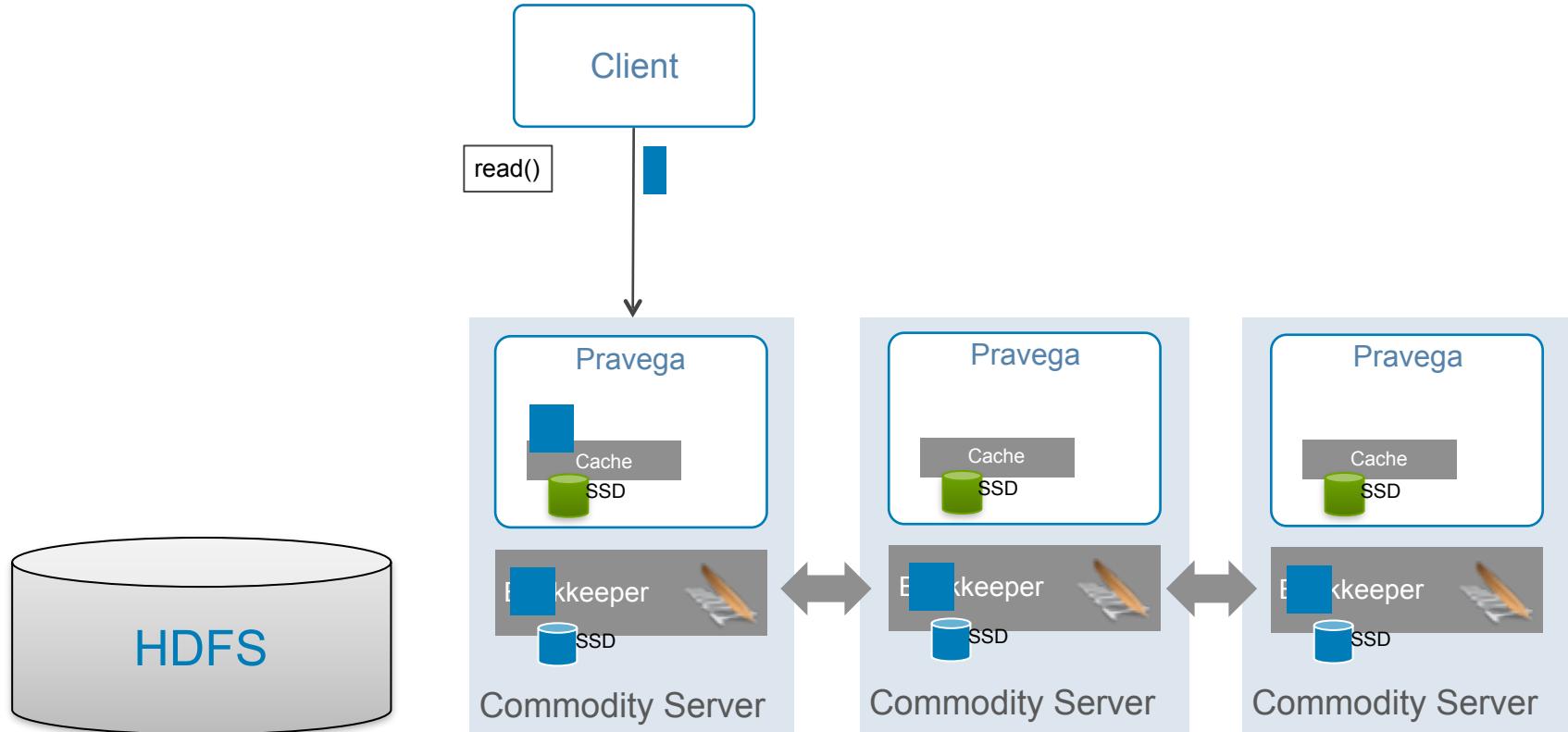
Idempotent output



Architecture overview - Write

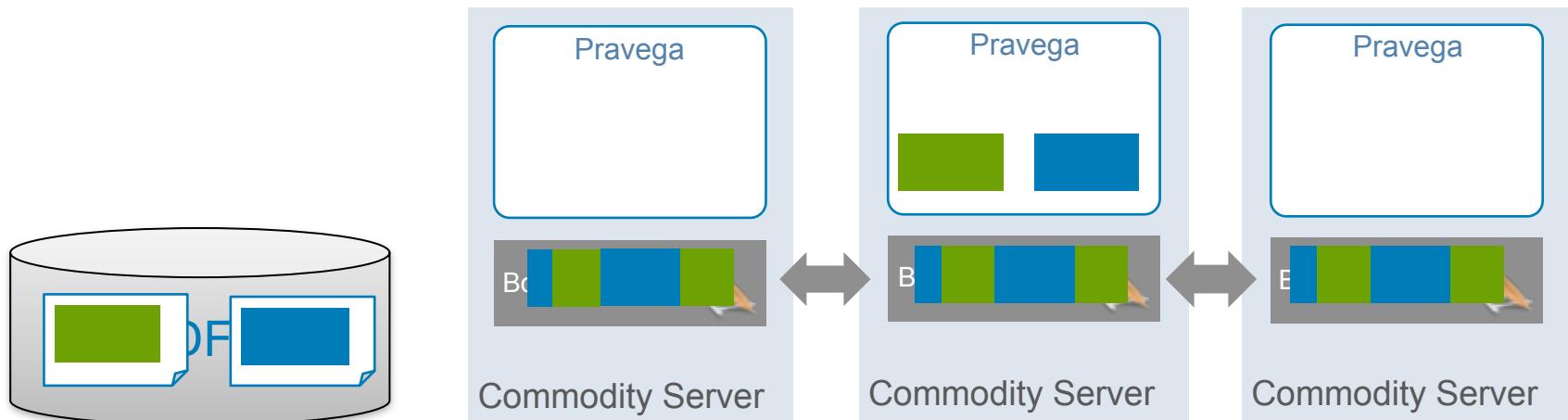


Architecture overview - Read

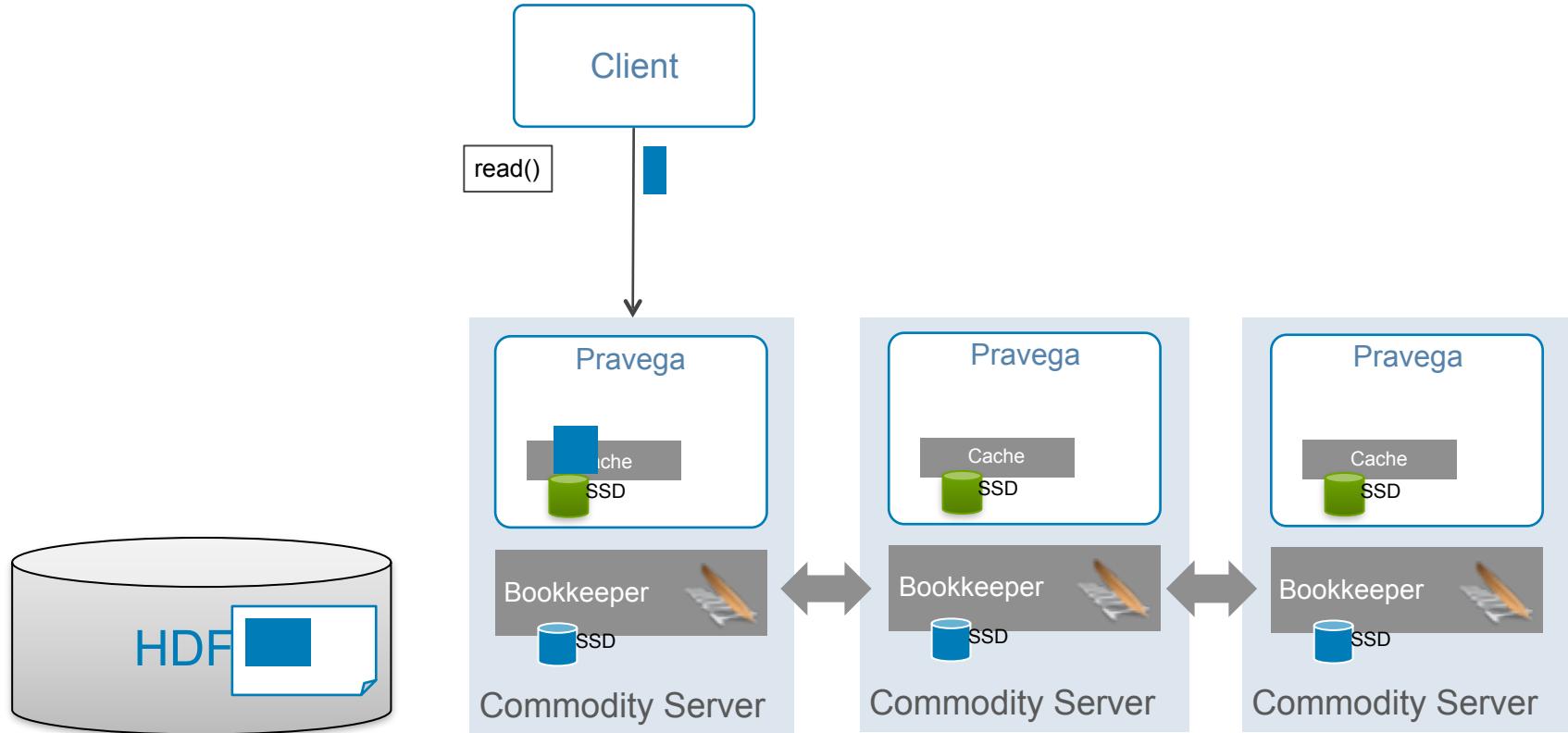


Architecture overview - Evict

- Files in HDFS are organized by Stream Segment
- Read-ahead cache optimizations are employed

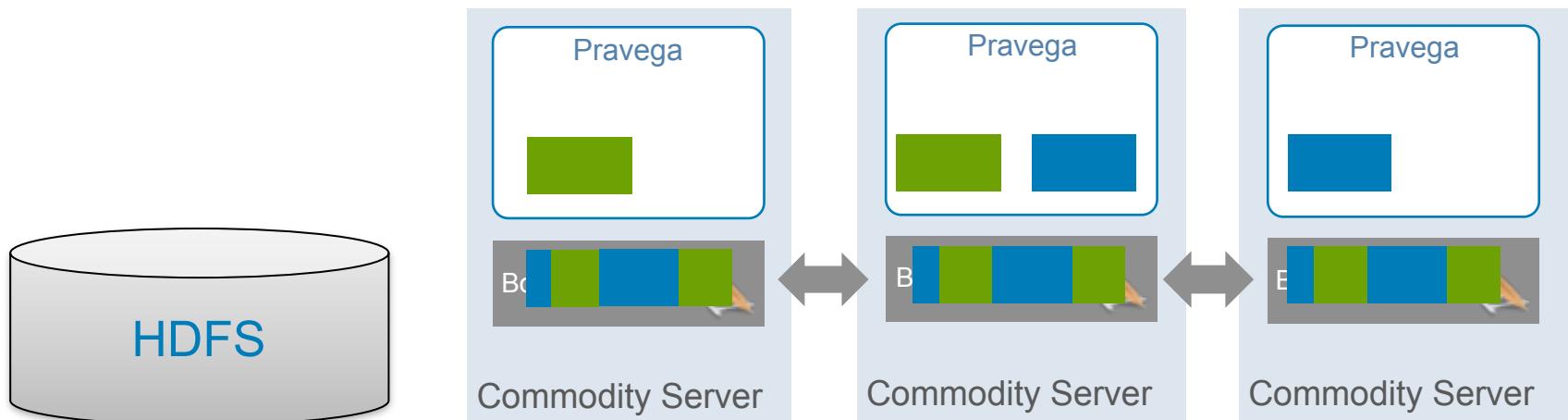


Architecture overview - Read



Architecture overview - Recover

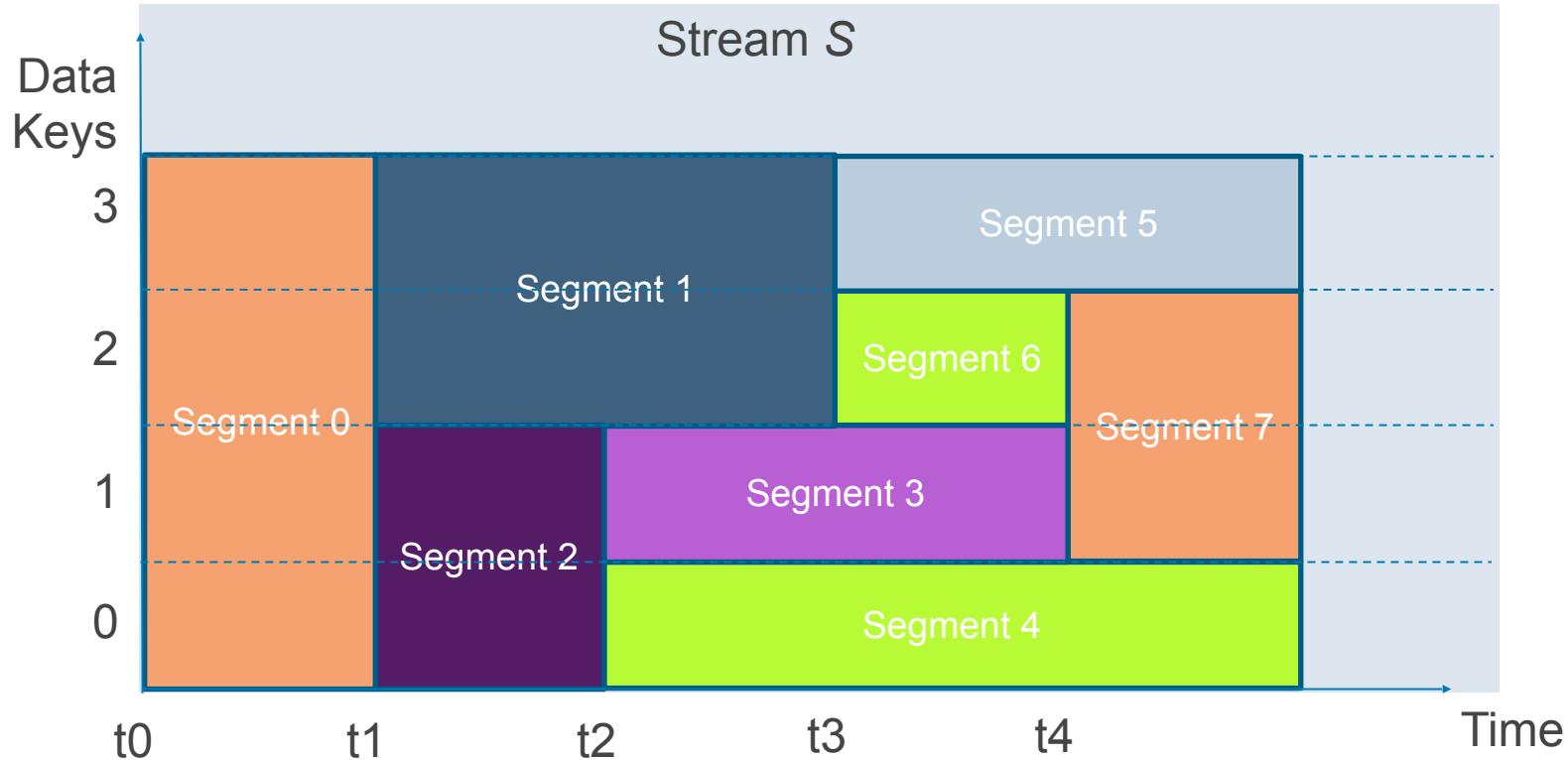
- Data is read from Bookkeeper only in the case of node failure
- Used to reconstitute the cache on the remaining hosts



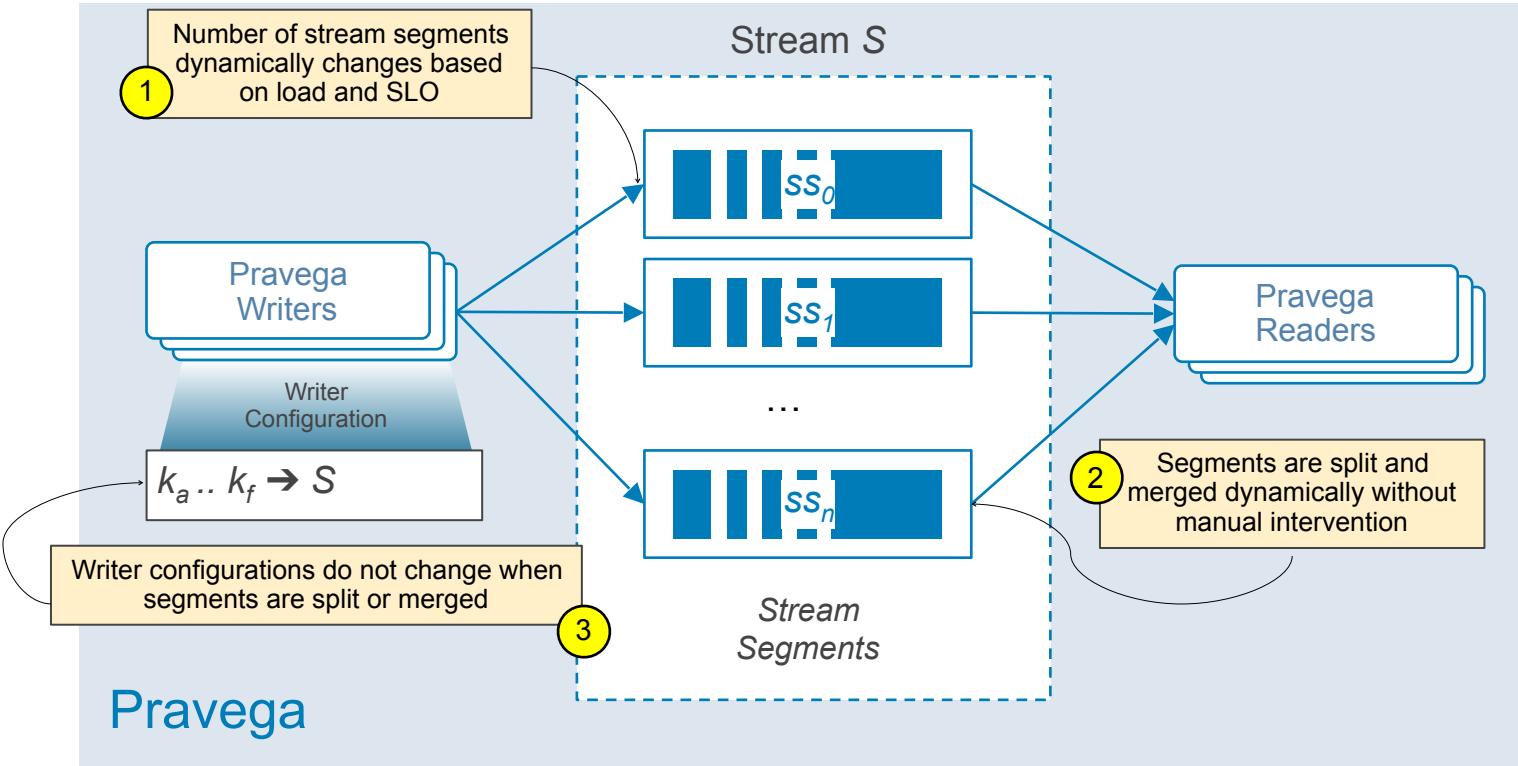
Performance Characteristics

- Fast appends to Bookkeeper
 - Data is persisted durably to disk 3x replicated consistently <10ms
- Big block writes to HDFS
 - Data is mostly cold so it can be erasure encoded and stored cheaply
 - If data is read, the job is likely a backfill so we can use a large read-ahead
- A stream's capacity is not limited by the capacity of a single machine
- Throughput shouldn't be either ...

Scaling: Segment Splitting & Merging



Scaling: Write Parallelism



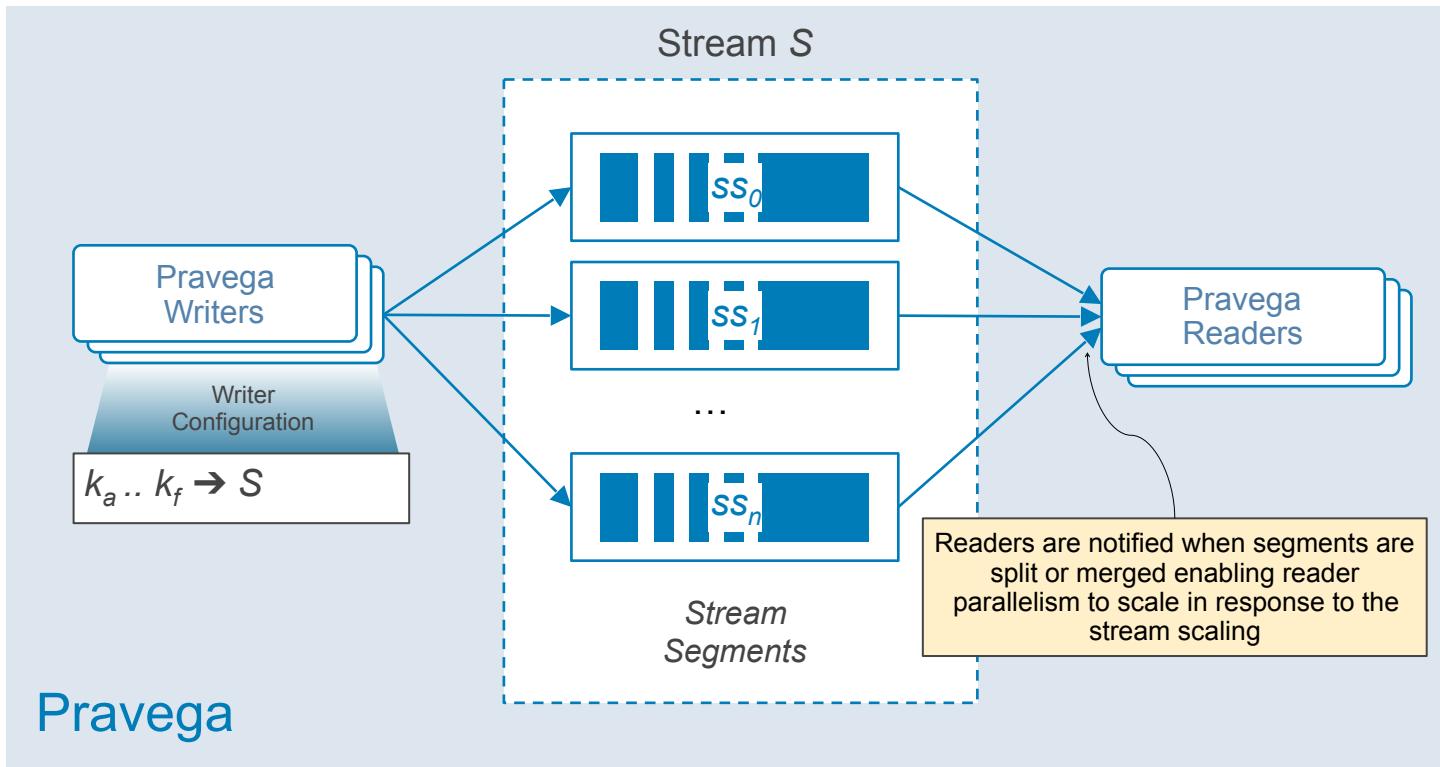
EventWriter API

```
/** A writer can write events to a stream. */
public interface EventStreamWriter {

    /** Send an event to the stream. Event must appear in the stream exactly once */
    AckFuture writeEvent(String routingKey, Type event);

    /** Start a new transaction on this stream */
    Transaction<Type> beginTxn(long transactionTimeout);
}
```

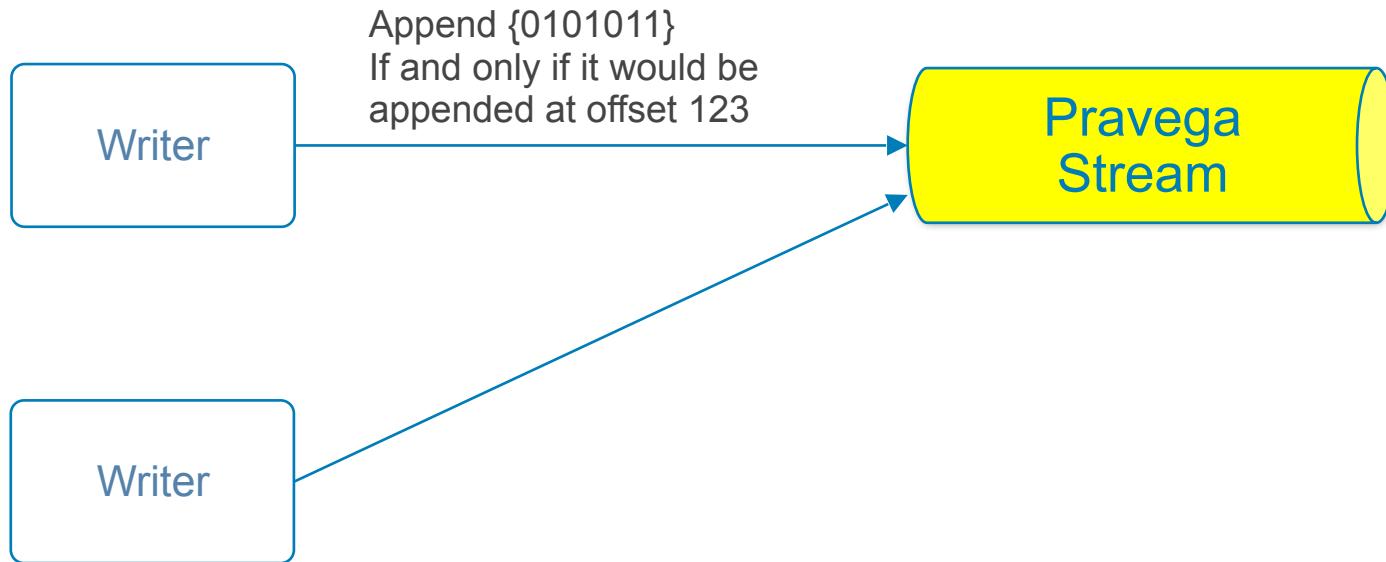
Scaling: Read Parallelism



EventReader API

```
public interface EventStreamReader<T> extends AutoCloseable {  
  
    /** Read the next event from the stream, blocking for up to timeout */  
  
    EventRead<T> readNextEvent(long timeout);  
  
    /**  
     * Close the reader. The segments owned by this reader will automatically be  
     * redistributed to the other readers in the group.  
     */  
  
    void close()  
}
```

Conditional Append



Synchronizer API

```
/** A means to synchronize state between many processes */
public interface StateSynchronizer<StateT> {

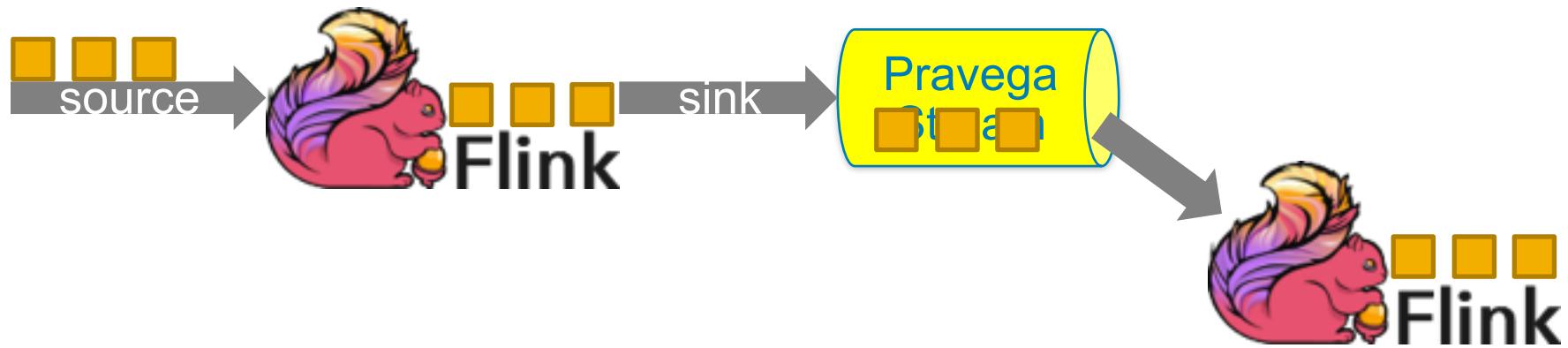
    /** Gets the state object currently held in memory */
    StateT getState();

    /** Fetch and apply all updates to bring the local state object up to date */
    void fetchUpdates();

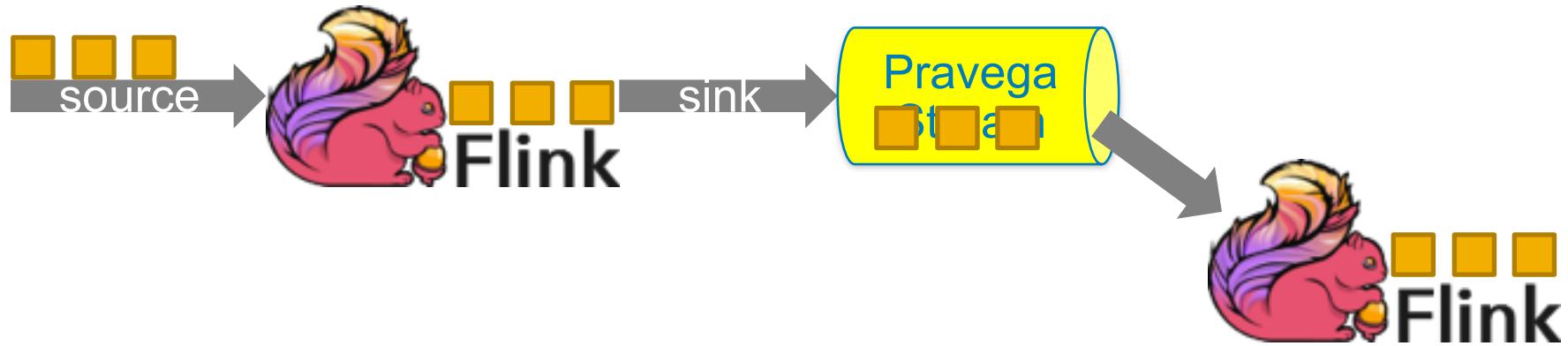
    /** Creates a new update for the latest state object and applies it atomically */
    void updateState(Function<StateT, Update<StateT>> updateGenerator);

}
```

Transactional output



Transactional output



EventWriter and Transaction API

```
/** A writer can write events to a stream. */
public interface EventStreamWriter {

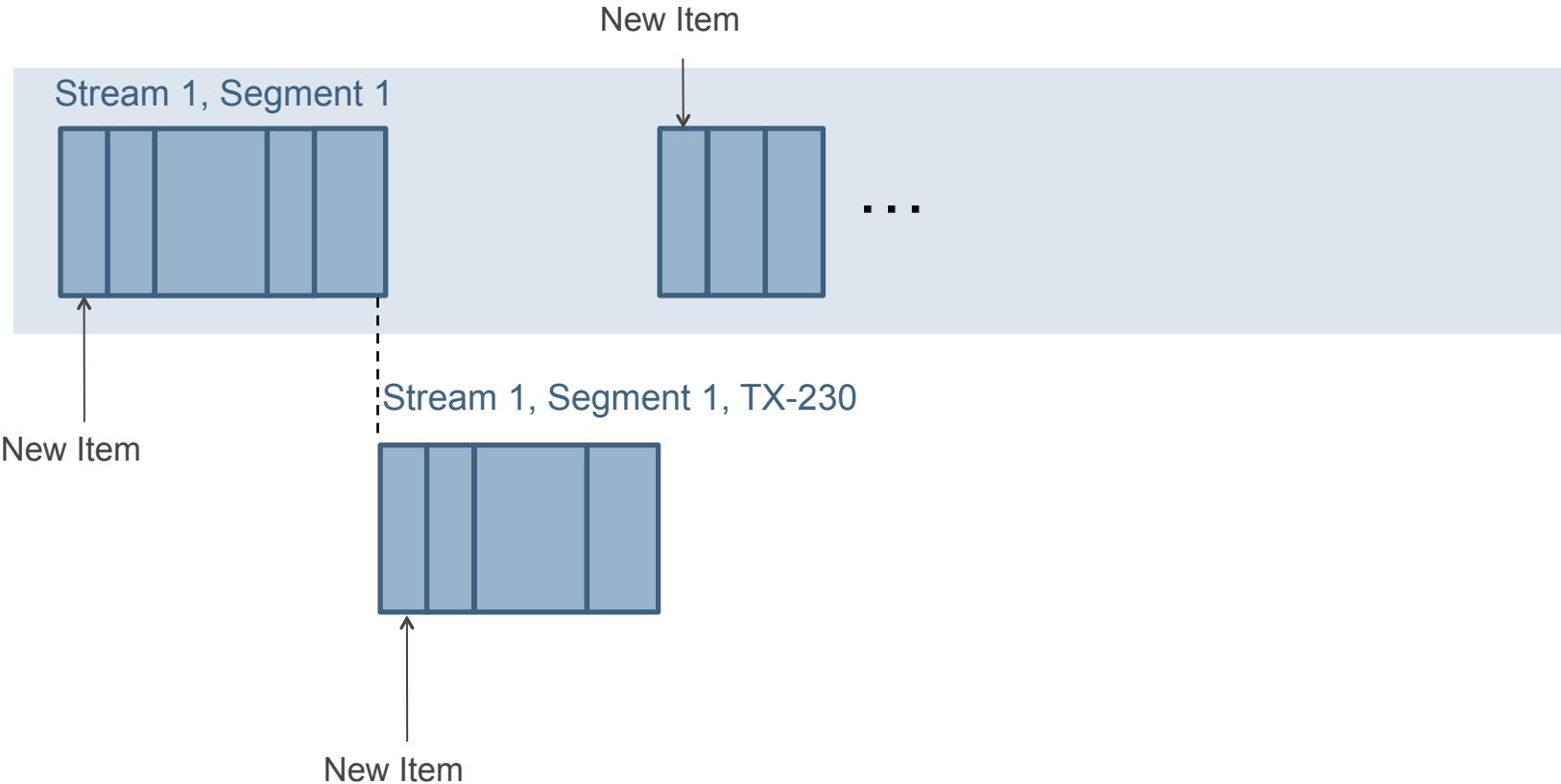
    /** Send an event to the stream. Event must appear in the stream exactly once */
    AckFuture writeEvent(String routingKey, Type event);

    /** Start a new transaction on this stream */
    Transaction<Type> begin();

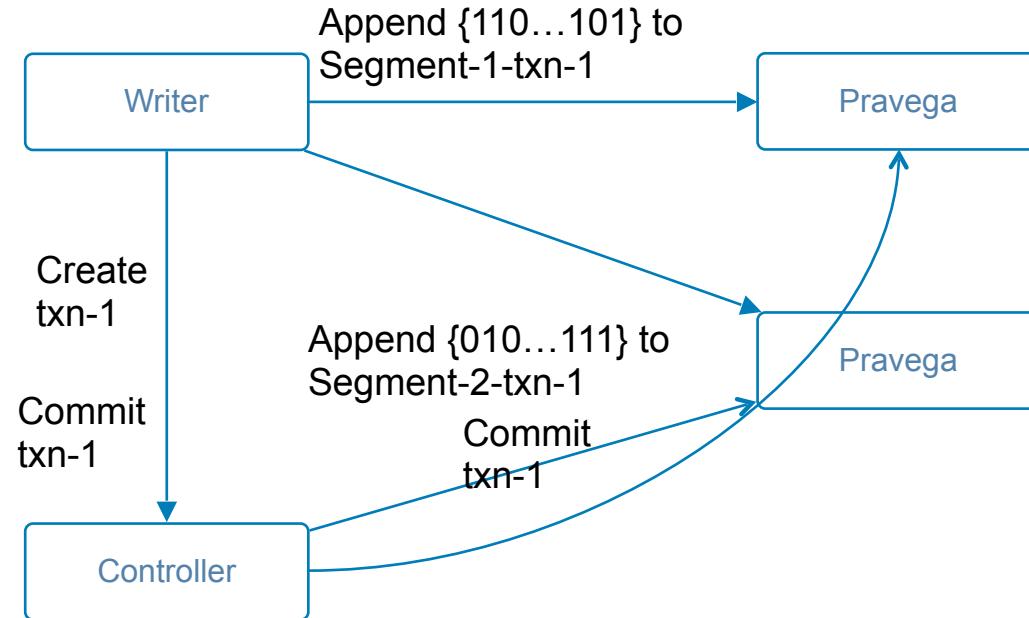
}

void writeEvent(String routingKey, Type event) throws TxnFailedException;
void commit() throws TxnFailedException;
void abort();
}
```

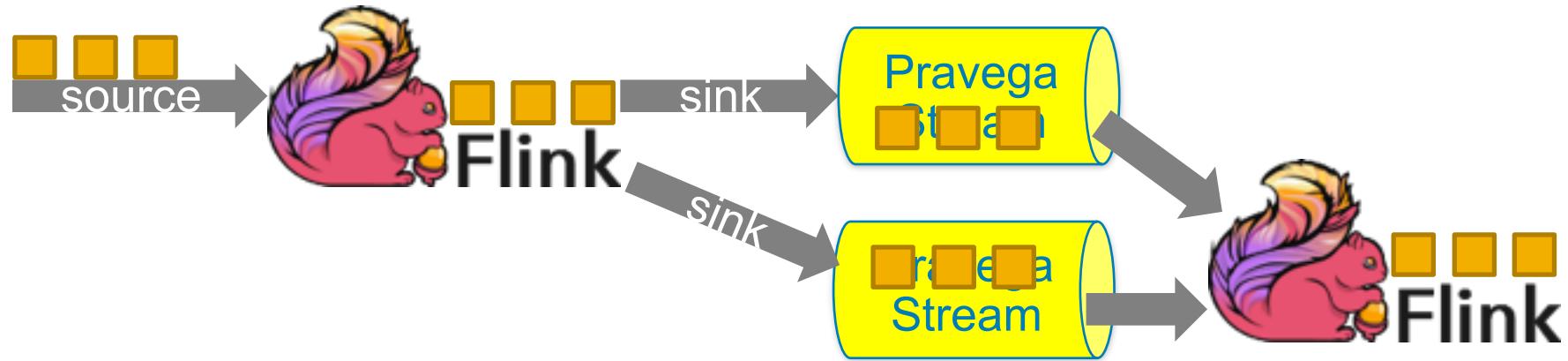
Transactions



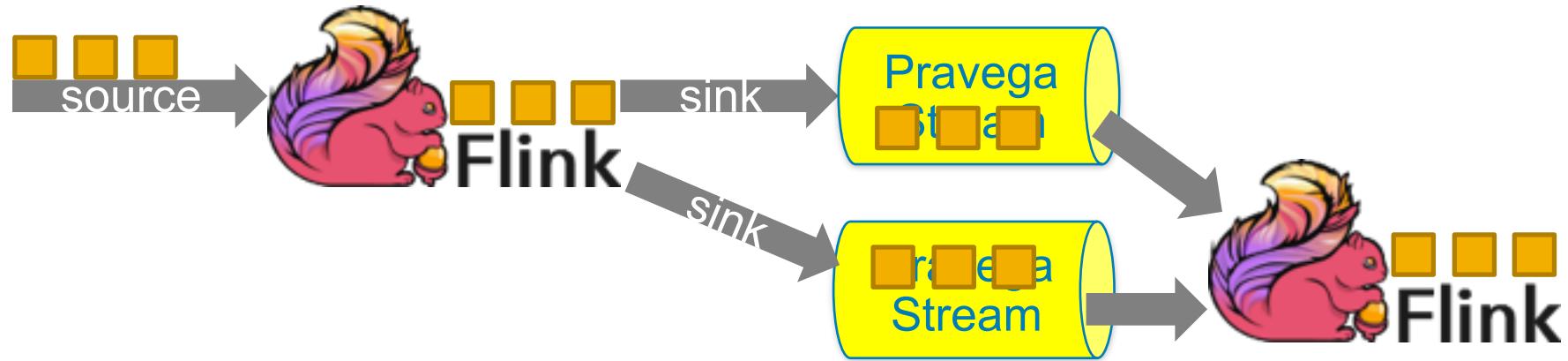
Transactions



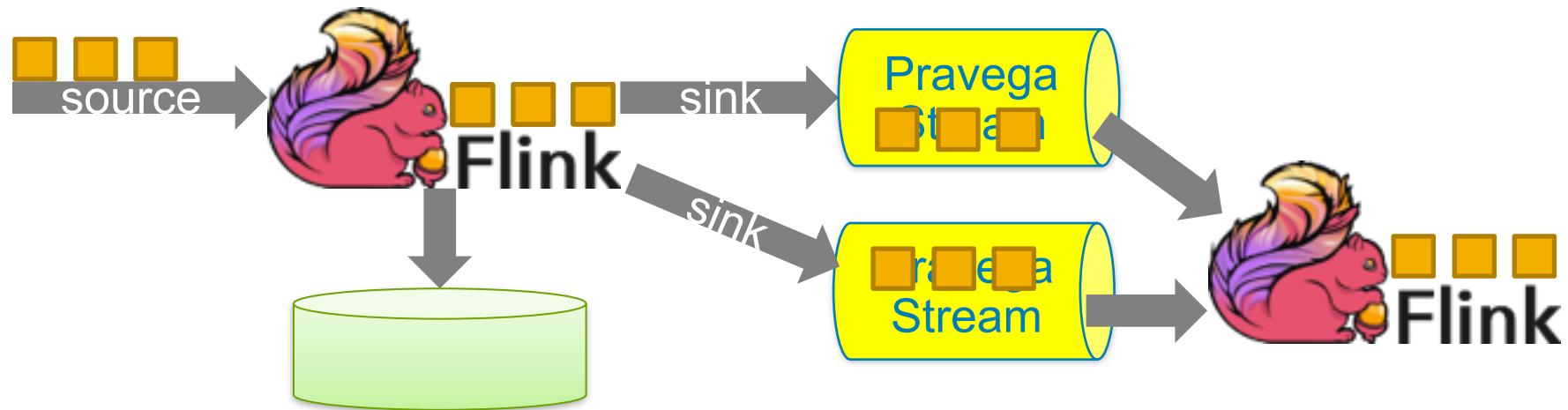
Transactional output



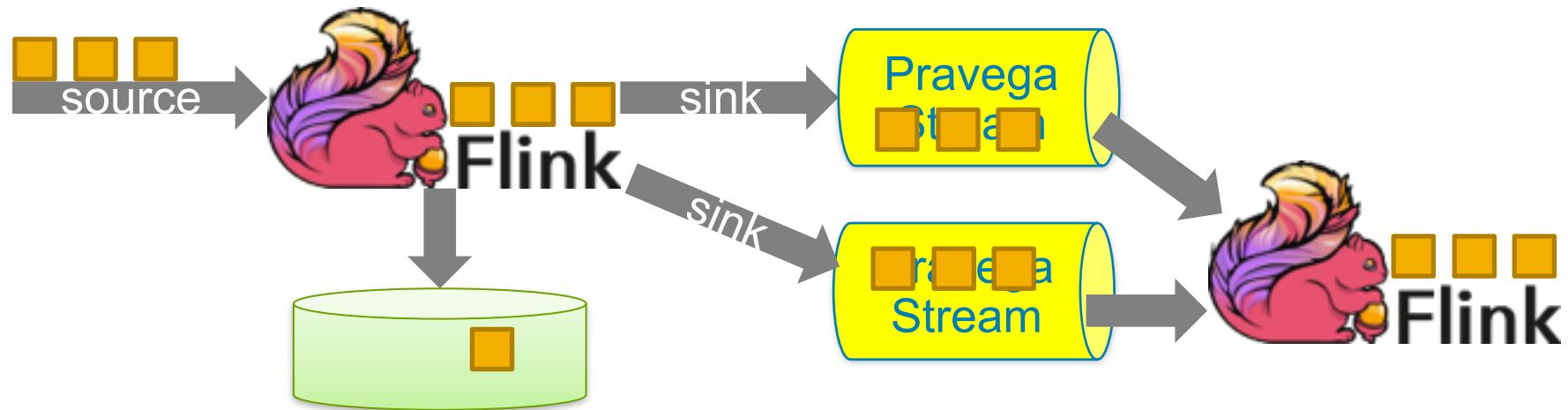
Transactional output



Transactional output



Transactional output



Pravega: Streaming Storage for All

- Pravega: an open source project with an open community
 - To be launched @ Dell EMC World this May 10th
 - Includes infinite byte stream primitive
 - Plus an Ingest Buffer with Pub/Sub built on top of streams
 - And Flink integration!
- Visit the Dell EMC booth here @ Flink Forward to learn more
- Contact us at pravega@emc.com for even more information!

Pravega

BB-8 Drawing

- Stop by the Dell EMC booth and enter to win
- Winner will be chosen after the closing Keynote
 - Must be present to win

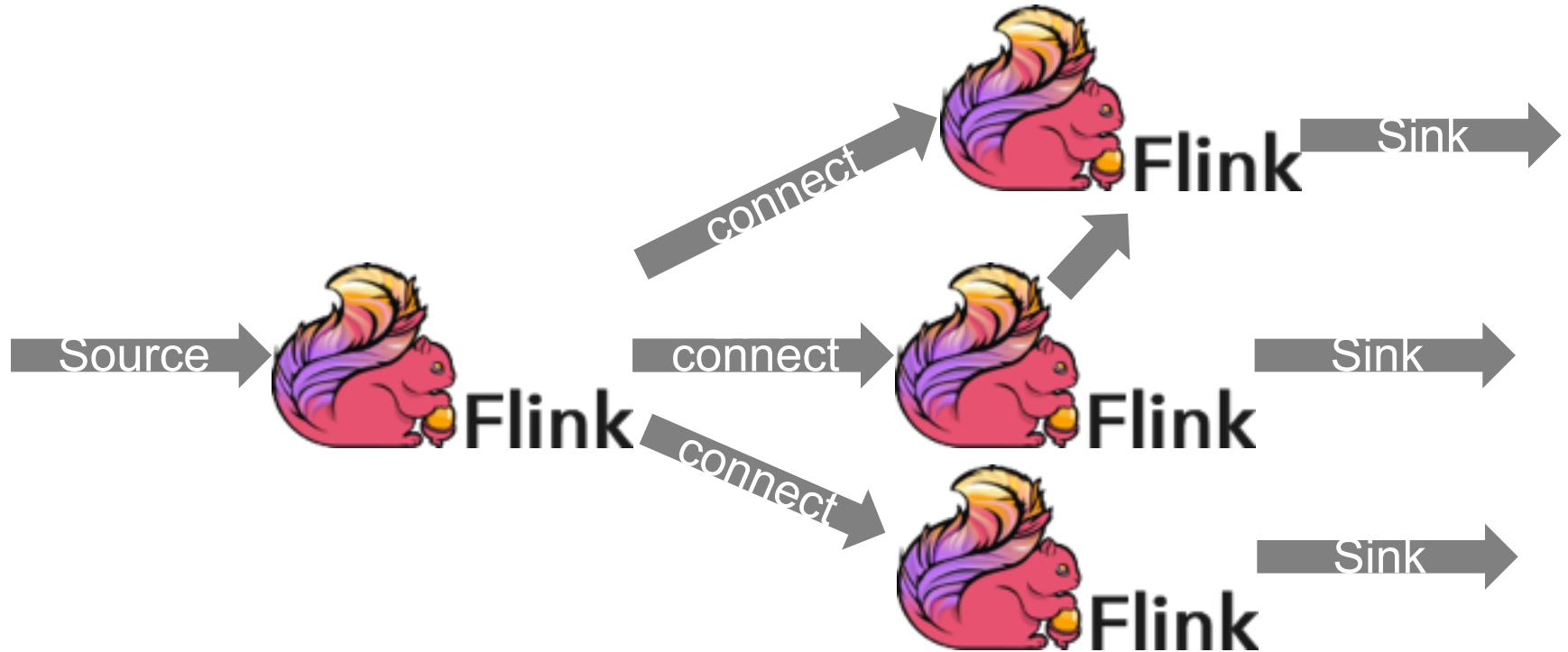


Email Pravega@emc.com for the latest news and information on Pravega!





Why a new storage system?



Why a new storage system?

Connector	Real Time	Exactly once	Durability	Storage Capacity	Notes
HDFS	No	Yes	Yes	Years	
Kafka	Yes	Source only	Yes* (Flushed but not synced)	Days	Writes are replicated but may not persisted to durable media. (flush.messages=1 bounds this but is not recommended)
RabbitMQ	Yes	Source only	Yes* (slowly)	Days	Durability can be added with a performance hit
Cassandra	No	Yes* (If updates are idempotent)	Yes	Years	App developers need to write custom logic to handle duplicate writes.
Sockets	Yes	No	No	None	

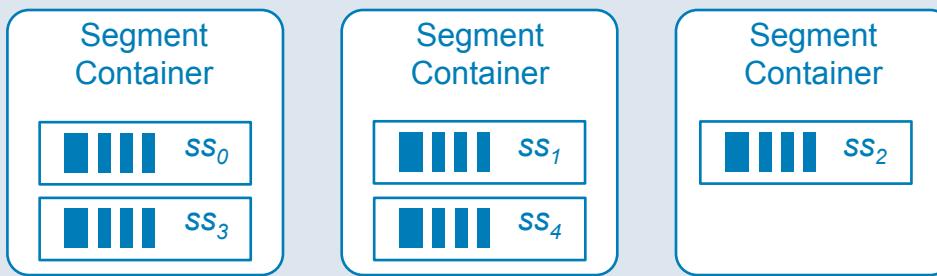
Flink storage needs

	Flink	Implications for storage
Guarantee	Exactly once	Exactly once, consistency
Latency	Very Low	Low latency writes (<10ms)
Throughput	High	High throughput
Computation model	Streaming	Streaming model
Overhead of fault tolerance mechanism	Low	Fast recovery Long retention
Flow control	Natural	Data can backlog Capacity not bounded by single host
Separation of application logic from fault tolerance	Yes	Re-reading data provides consistent results
License	Apache 2.0	Open Source and linkable

Shared config

```
public class SharedConfig<K extends Serializable, V extends Serializable> {  
  
    public V getProperty(K key);  
  
    public V putPropertyIfAbsent(K key, V value);  
  
    public boolean removeProperty(K key, V oldValue);  
  
    public boolean replaceProperty(K key, V oldValue, V newValue);  
  
}
```

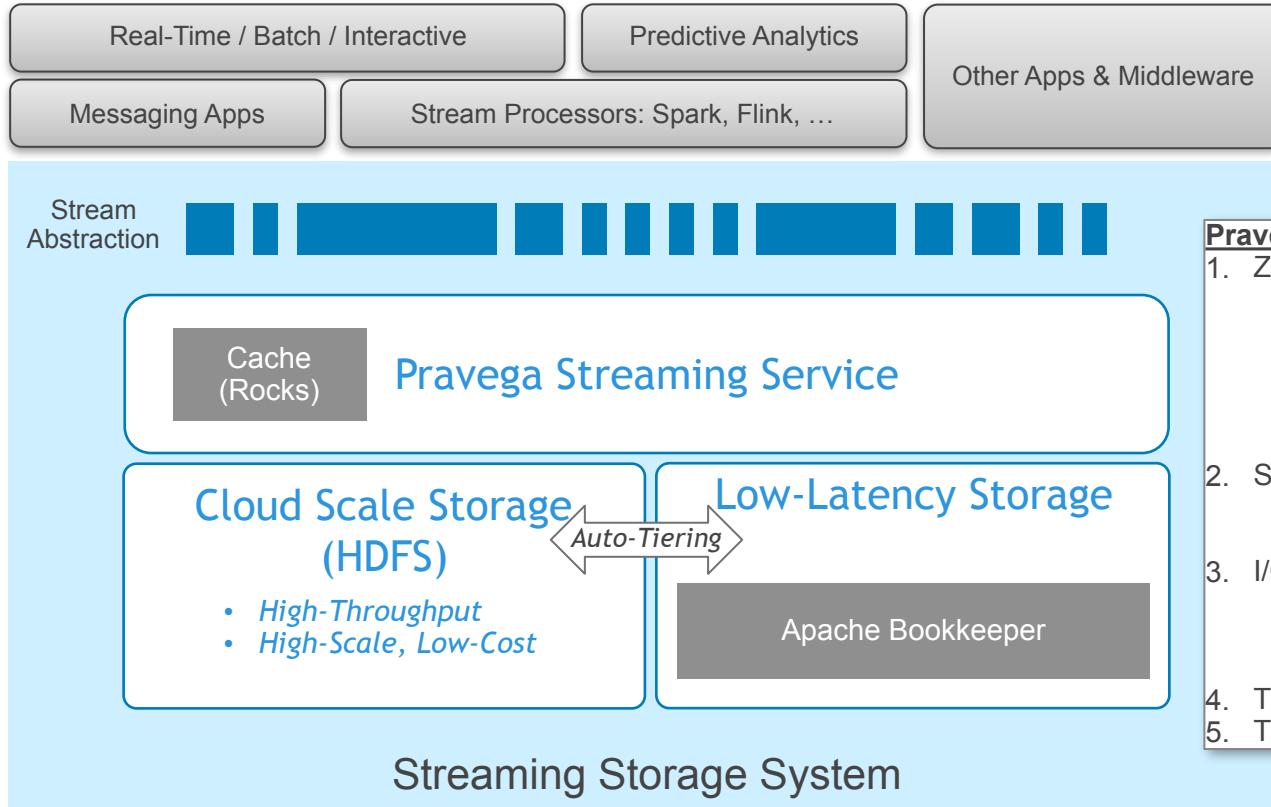
Smart Workload Distribution



The hot segment is automatically “split,” and the “child” segments are re-distributed across the cluster relieving the hot spot while maximizing utilization of the cluster’s available IOPs capacity

Pravega

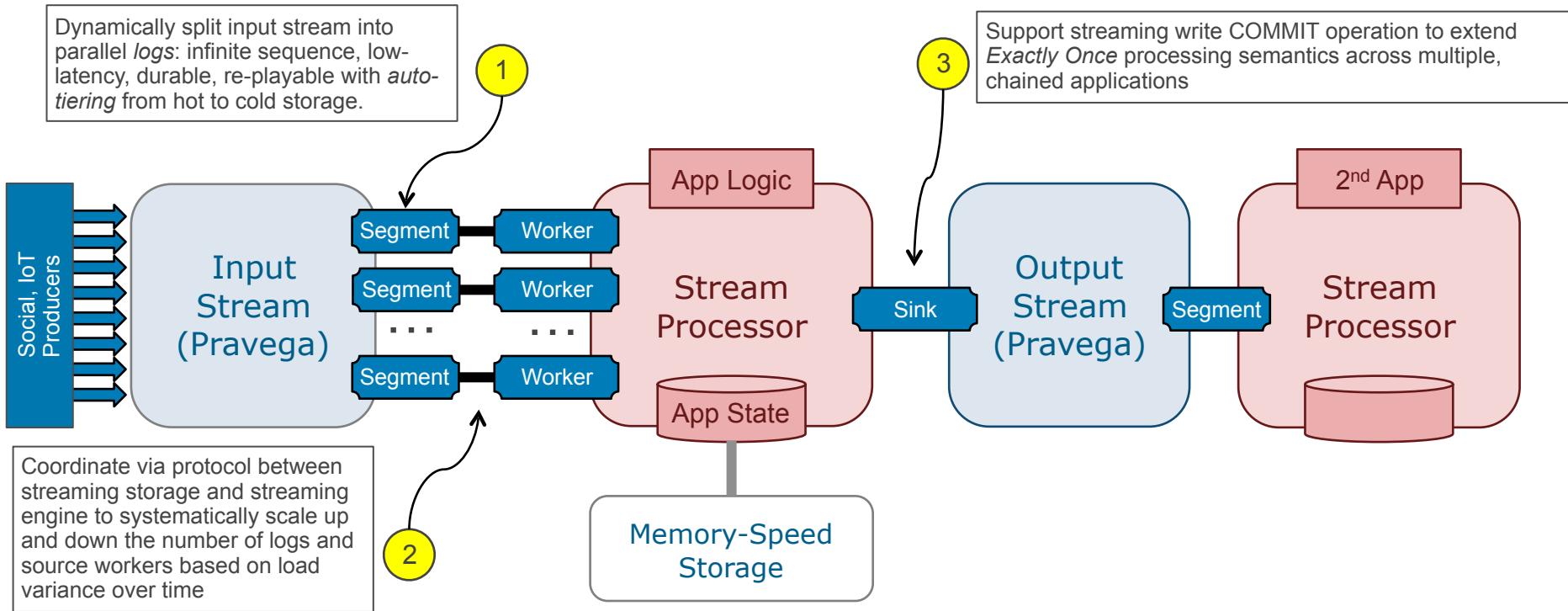
Architecture



Pravega Design Innovations

1. Zero-Touch Dynamic Scaling
 - Automatically scale read/write parallelism based on load and SLO
 - No service interruptions
 - No manual reconfiguration of clients
 - No manual reconfiguration of service resources
2. Smart Workload Distribution
 - No need to over-provision servers for peak load
3. I/O Path Isolation
 - For tail writes
 - For tail reads
 - For catch-up reads
4. Tiering for “Infinite Streams”
5. Transactions For “Exactly Once”

Pravega Optimizations for Stream Processors



Comparing Pravega and Kafka Design Points

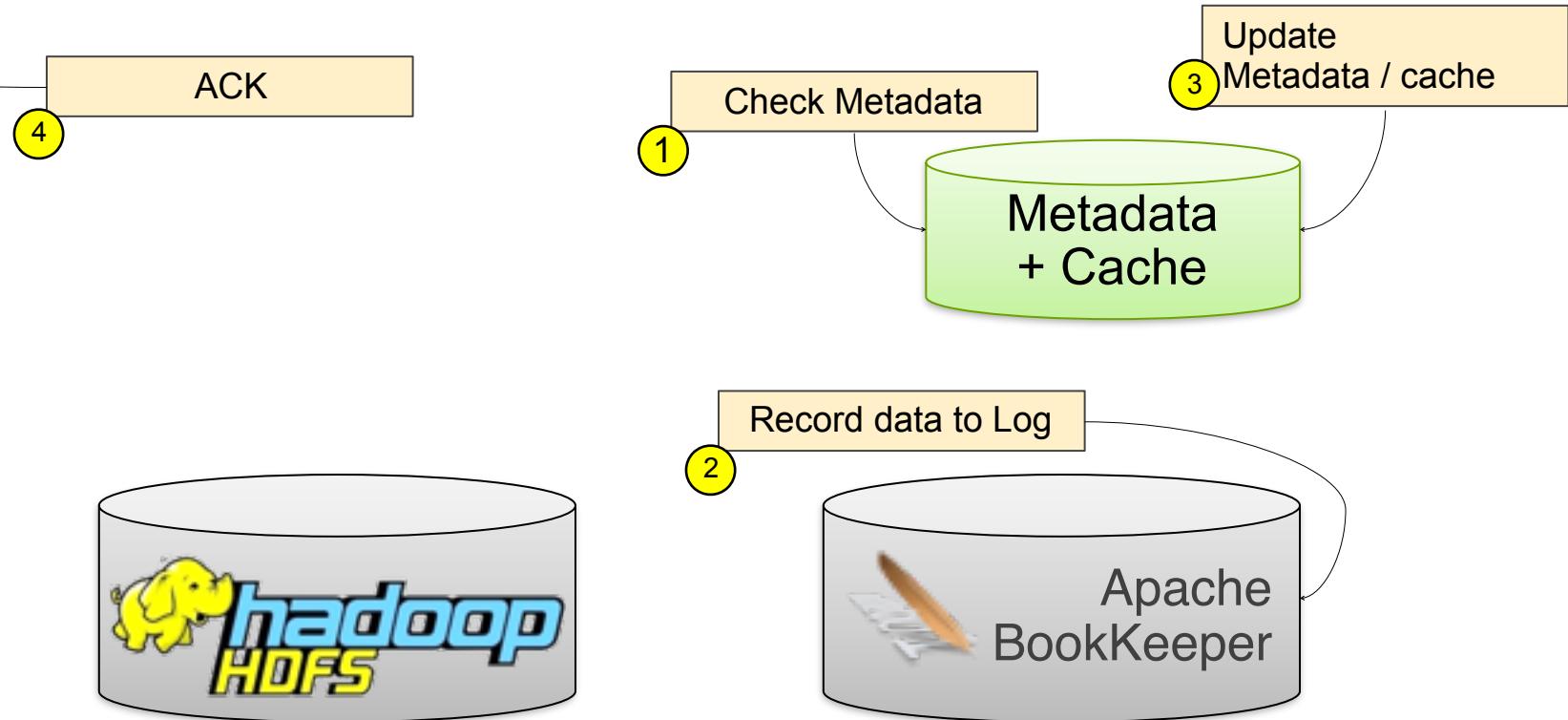
Unlike Kafka, Pravega is designed to be a durable and permanent storage system

Quality	Pravega Goal	Kafka Design Point
Data Durability	Replicated and persisted to disk before ACK	Replicated but not persisted to disk before ACK X
Strict Ordering	Consistent ordering on tail and catch-up reads	Messages may get reordered X
Exactly Once	Producers can use transactions for atomicity	Messages may get duplicated X
Scale	Tens of millions of streams per cluster	Thousands of topics per cluster X
Elastic	Dynamic partitioning of streams based on load and SLO	Statically configured partitions X
Size	Log size is not bounded by the capacity of any single node	Partition size is bounded by capacity of filesystem on its hosting node X
	Transparently migrate/retrieve data from Tier 2 storage for older parts of the log	External ETL required to move data to Tier 2 storage; no access to data via Kafka once moved X
Performance	Low (<10ms) latency durable writes; throughput bounded by network bandwidth	Low-latency achieved only by reducing replication/reliability parameters X
	Read pattern (e.g. many catch-up readers) does not affect write performance	Read patterns adversely affects write performance due to reliance on OS filesystem cache X

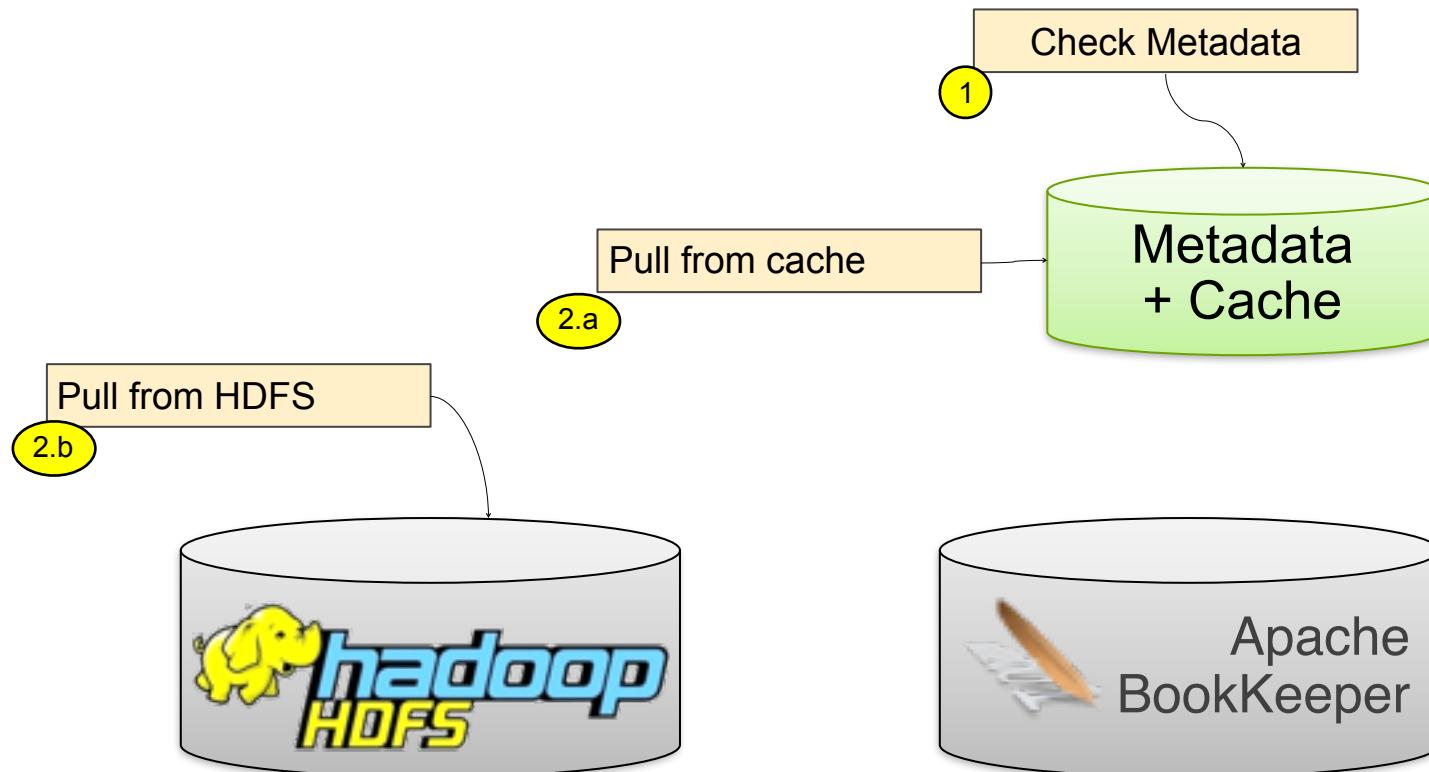
Attributes

Connector	Streaming	Exactly once	Durability	Storage Capacity
HDFS	No	Yes	Yes	Years
Kafka	Yes	Source only	Yes* (Flushed but not synced)	Days
Pravega	Yes: Byte oriented and event oriented	Yes. With either idempotent producers, or transactions	Yes. Always flushed and synced, with low latency.	As much as you can fit in your HDFS cluster.

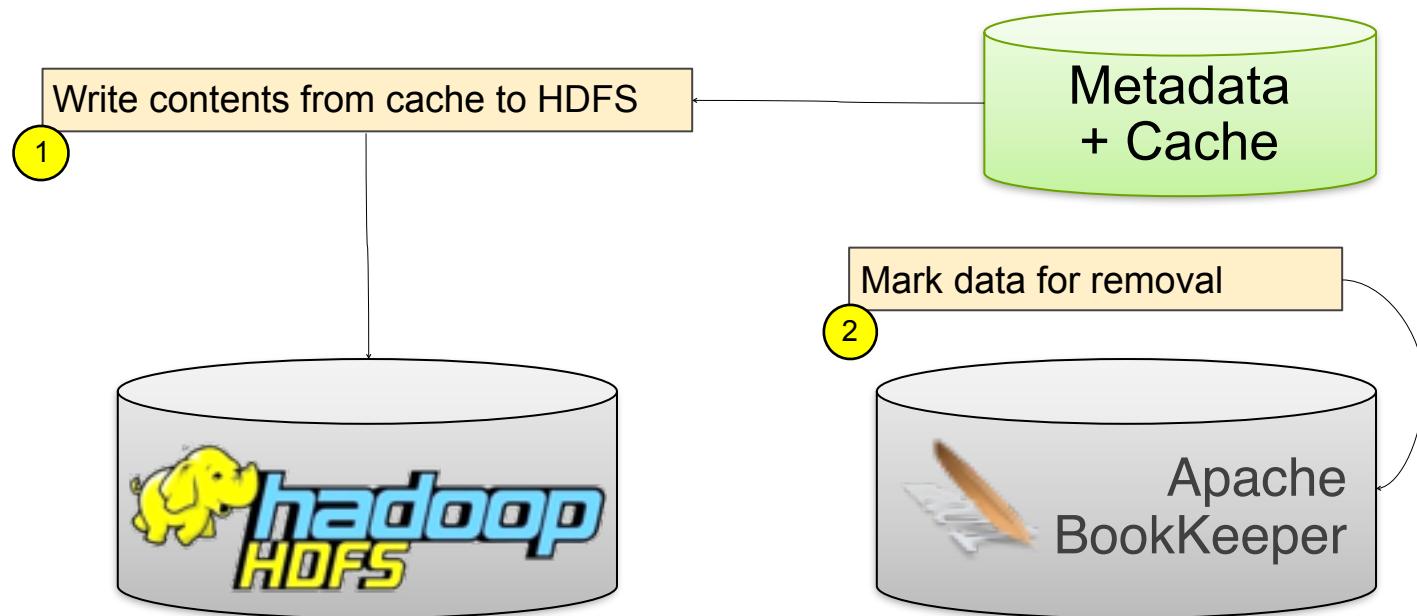
Architecture overview - Write



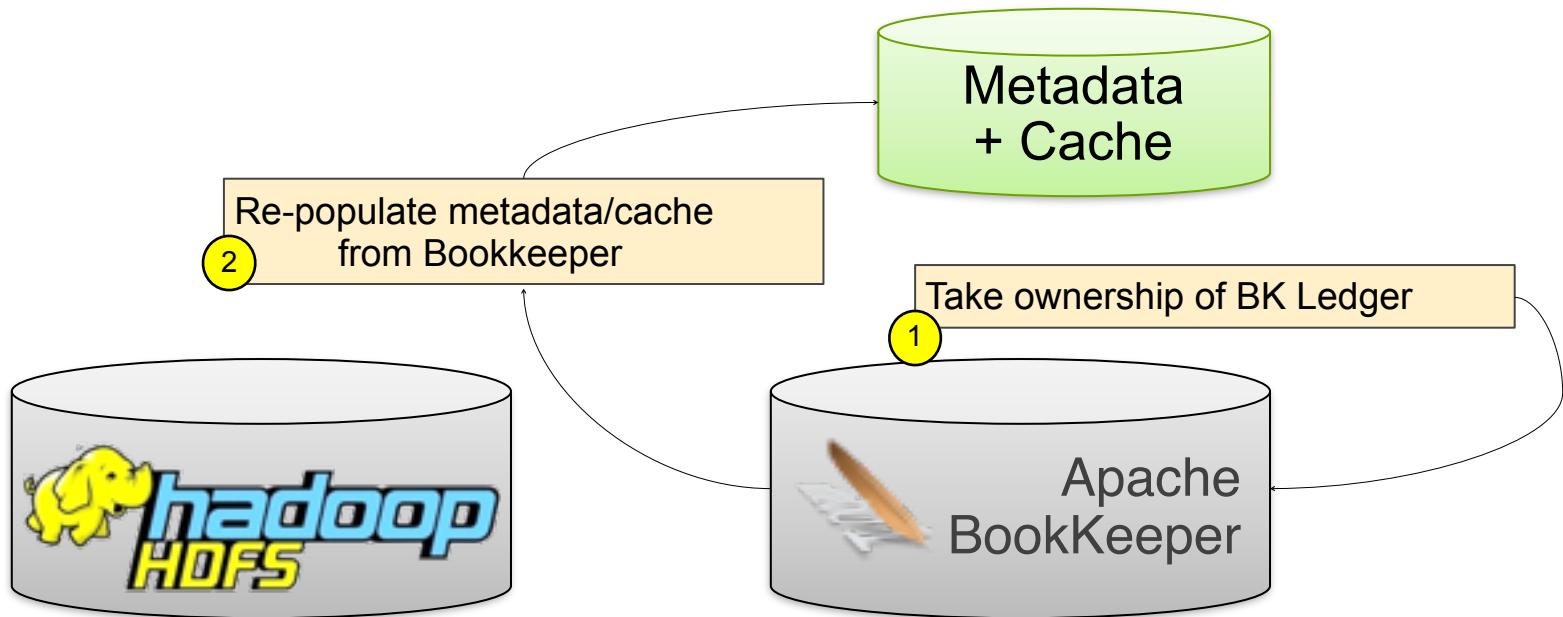
Architecture overview - Read



Architecture overview - Evict



Architecture overview - Recover





dataArtisans





















```
val tEnv = TableEnvironment.getTableEnvironment(env)

// configure your data source
val customerSource = CsvTableSource.builder()
    .path("/path/to/customer_data.csv")
    .field("name", Types.STRING).field("prefs", Types.STRING)
    .build()

// register as a table
tEnv.registerTableSource("cust", customerSource)

// define your table program
val table = tEnv.scan("cust").select('name.lowerCase(), myParser('prefs))
val table = tEnv.sql("SELECT LOWER(name), myParser(prefs) FROM cust")

// convert
val ds: DataStream[Customer] = table.toDataStream[Customer]
```

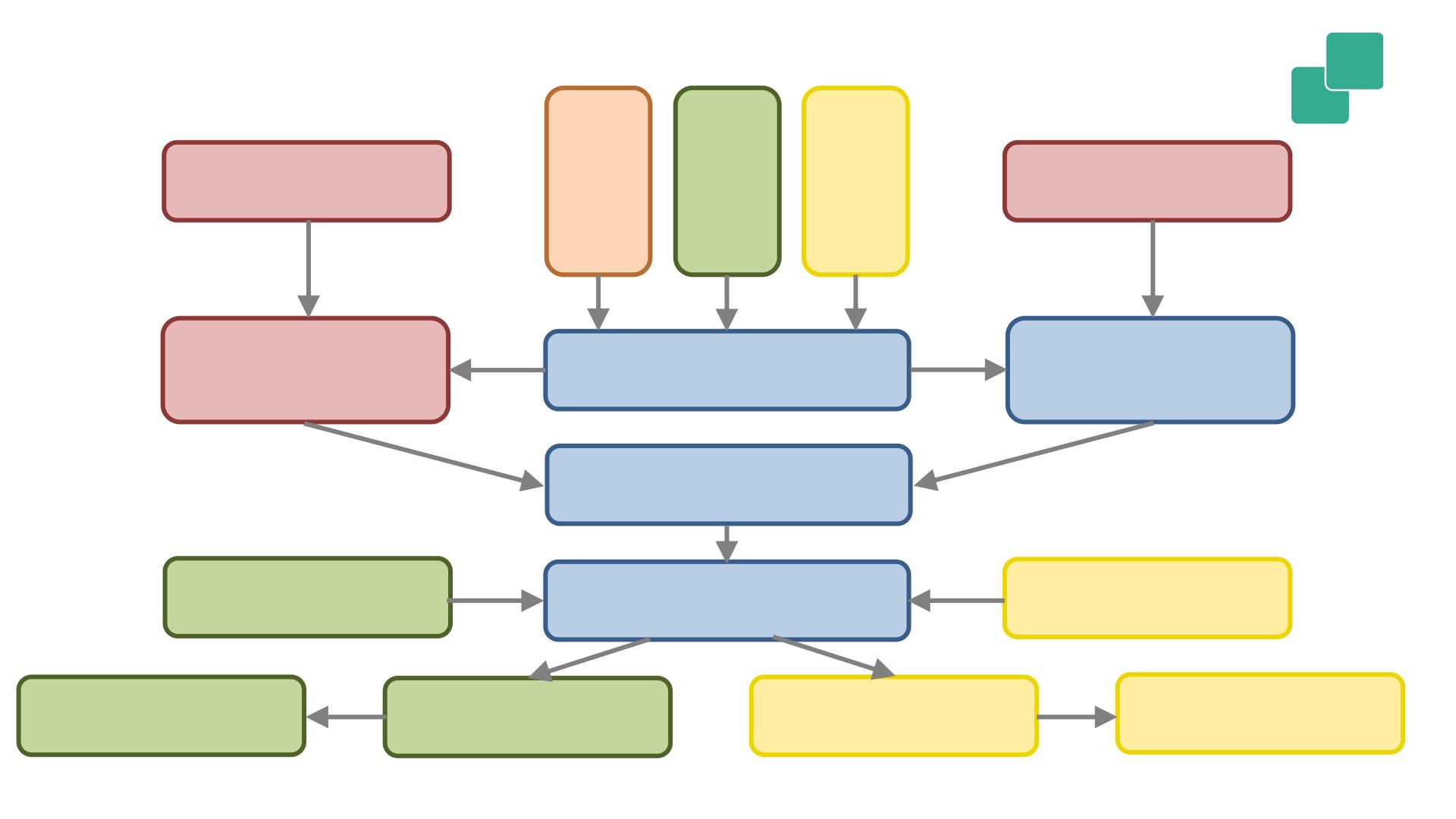


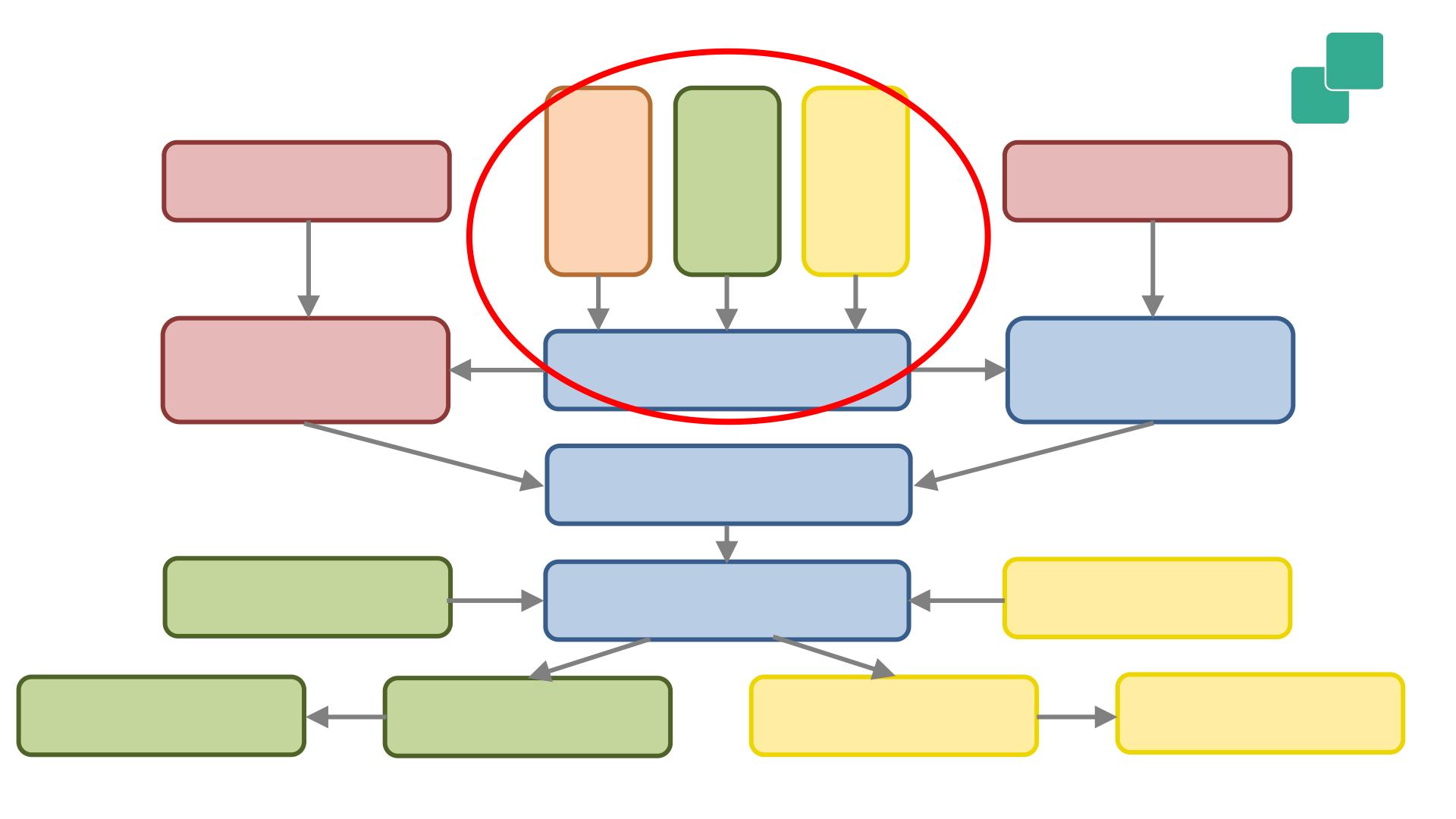
```
val sensorData: DataStream[(String, Long, Double)] = ???  
  
// convert DataStream into Table  
val sensorTable: Table = sensorData  
    .toTable(tableEnv, 'location, 'rowtime, 'tempF)  
  
// define query on Table  
val avgTempCTable: Table = sensorTable  
    .window(Tumble over 1.day on 'rowtime as 'w)  
    .groupBy('location, 'w)  
    .select('w.start as 'day,  
            'location,  
            ('tempF.avg - 32) * 0.556) as 'avgTempC)  
    .where('location like "room%")
```

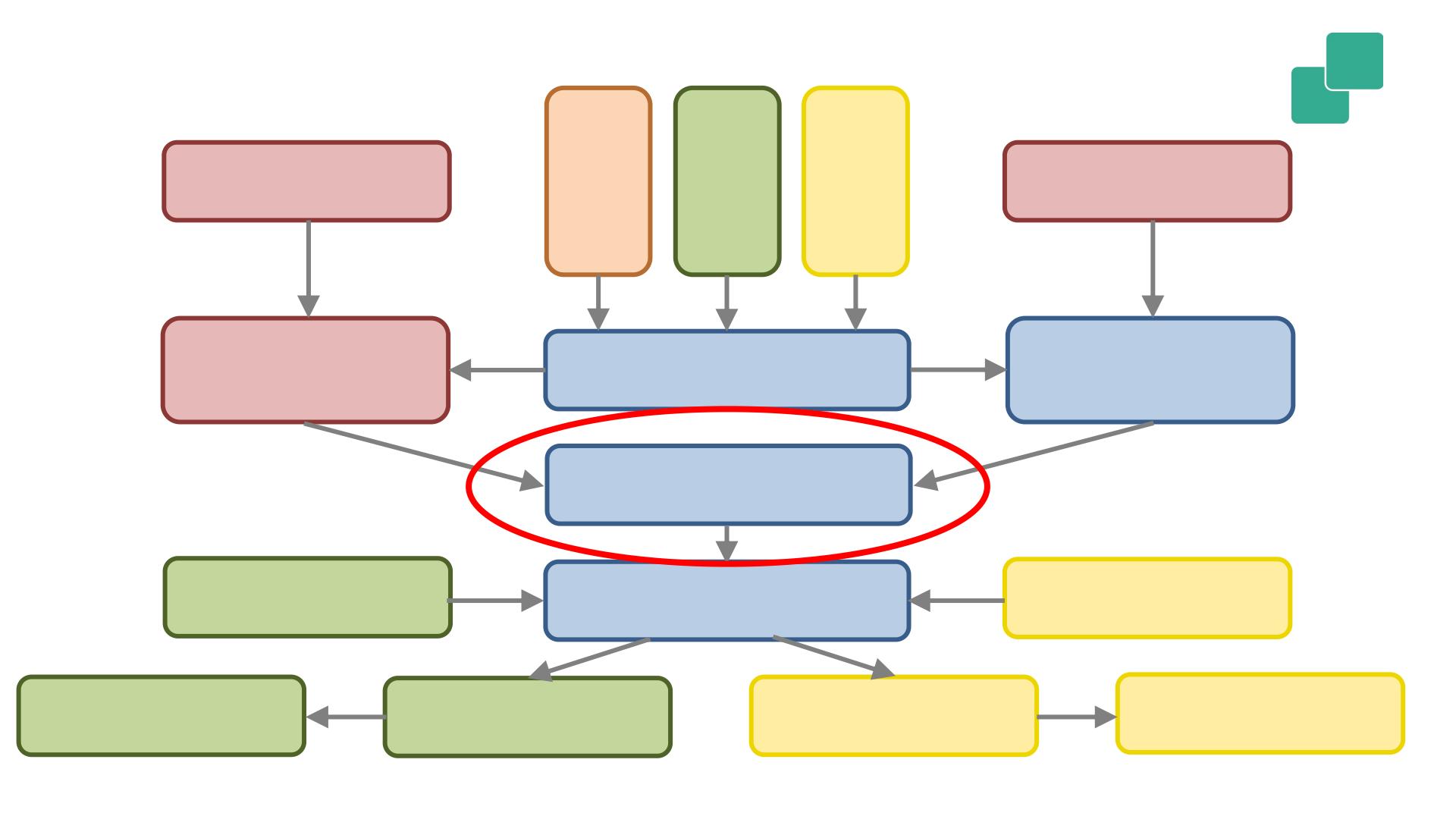


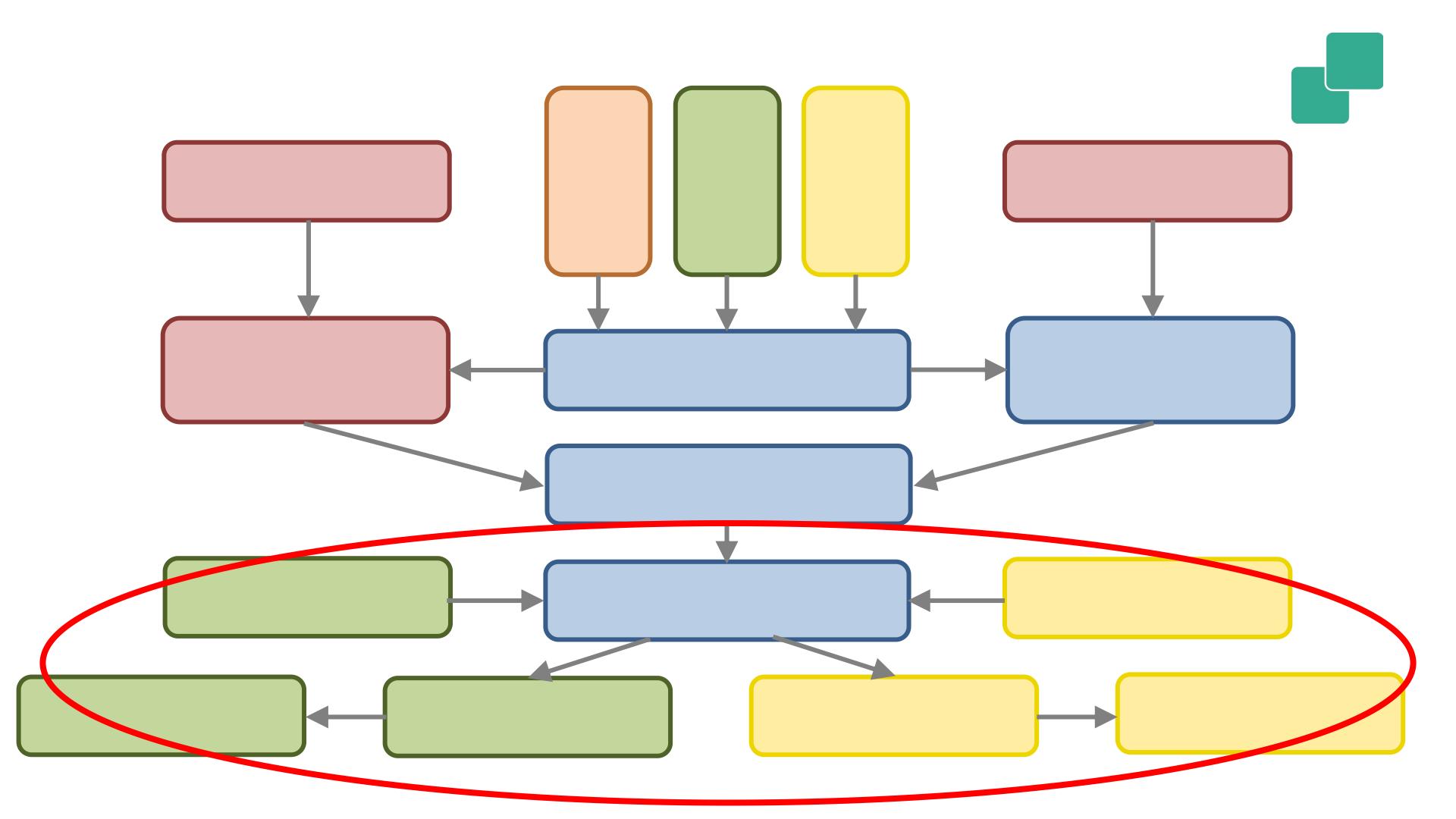
```
val sensorData: DataStream[(String, Long, Double)] = ???  
  
// register DataStream  
tableEnv.registerDataStream(  
    "sensorData", sensorData, 'location, 'rowtime, 'tempF)  
  
// query registered Table  
val avgTempCTable: Table = tableEnv.sql("""  
    SELECT TUMBLE_START(TUMBLE(time, INTERVAL '1' DAY) AS day,  
        location,  
        AVG((tempF - 32) * 0.556) AS avgTempC  
    FROM sensorData  
    WHERE location LIKE 'room%'  
    GROUP BY location, TUMBLE(time, INTERVAL '1' DAY)  
    """))
```











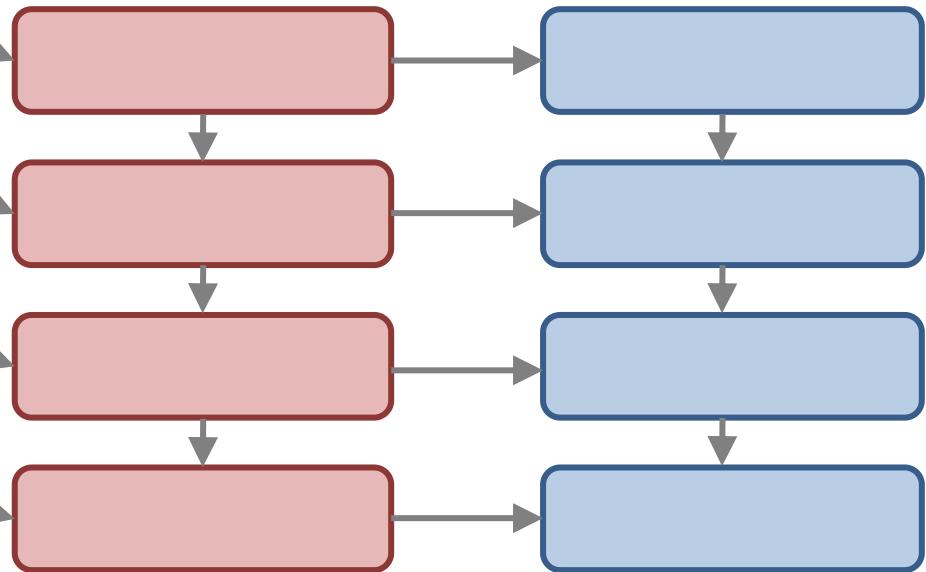


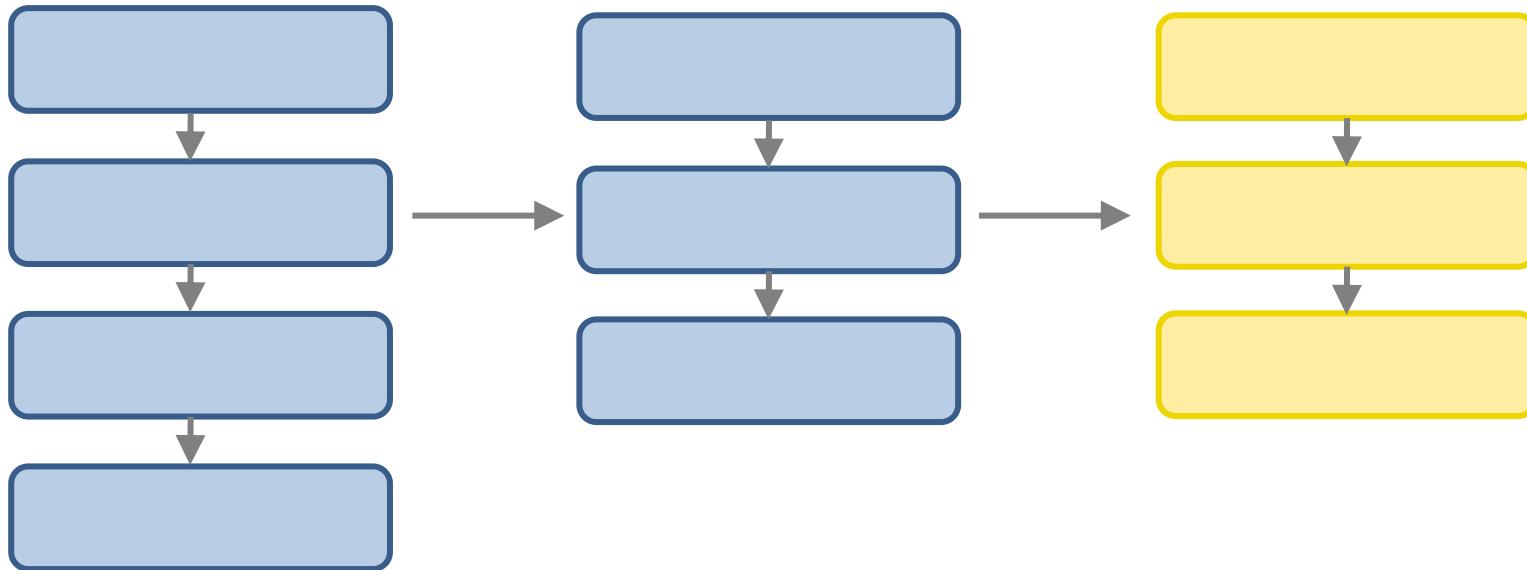
sensorTable

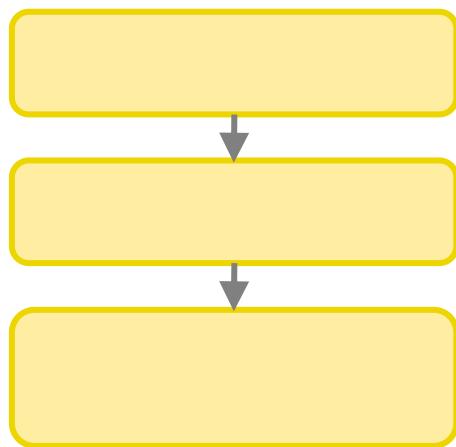
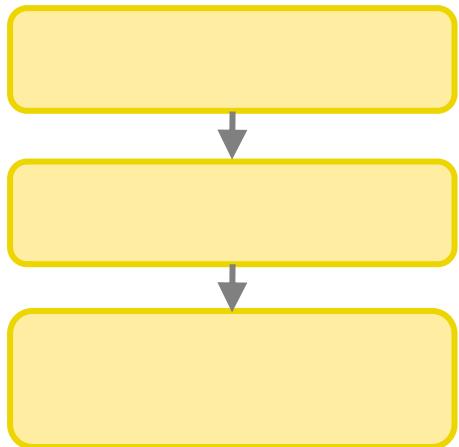
.window(Tumble over 1.day on 'rowtime' as 'w')
.groupBy('location', 'w')

.select(
 'w.start' as 'day',
 'location',
 ((tempF.avg - 32) *
 0.556) as 'avgTempC')

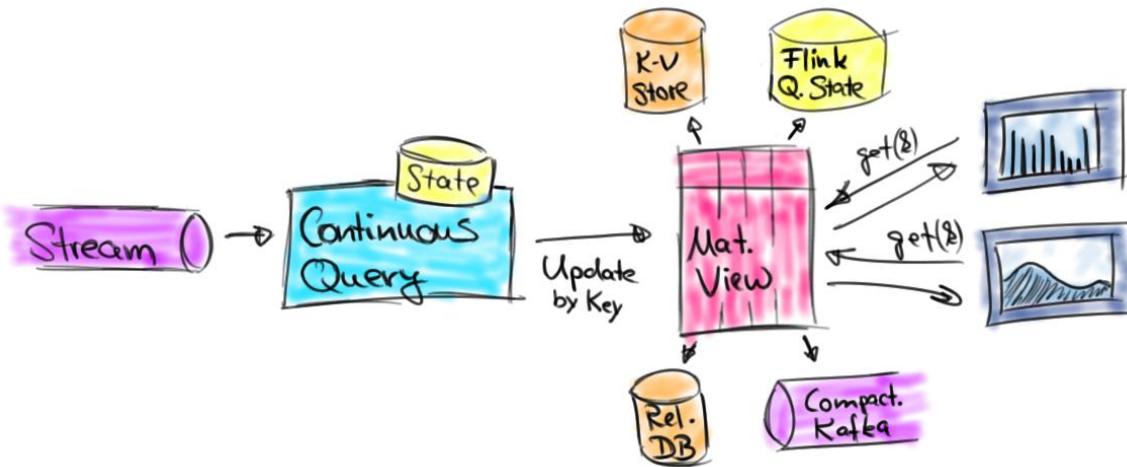
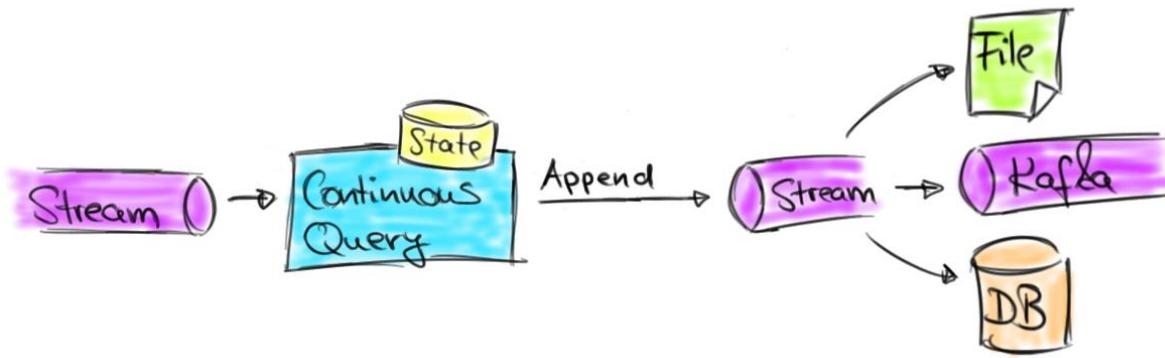
.where('location like "room%"')



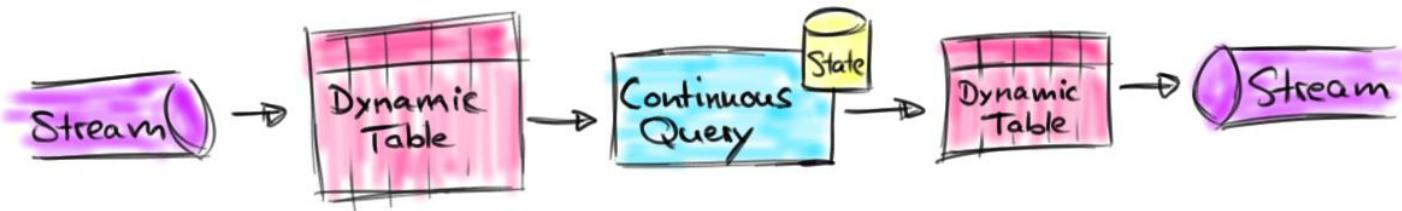












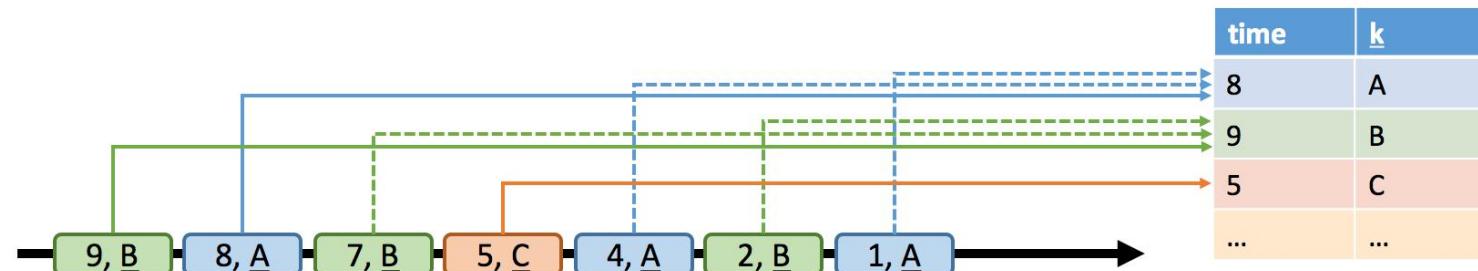
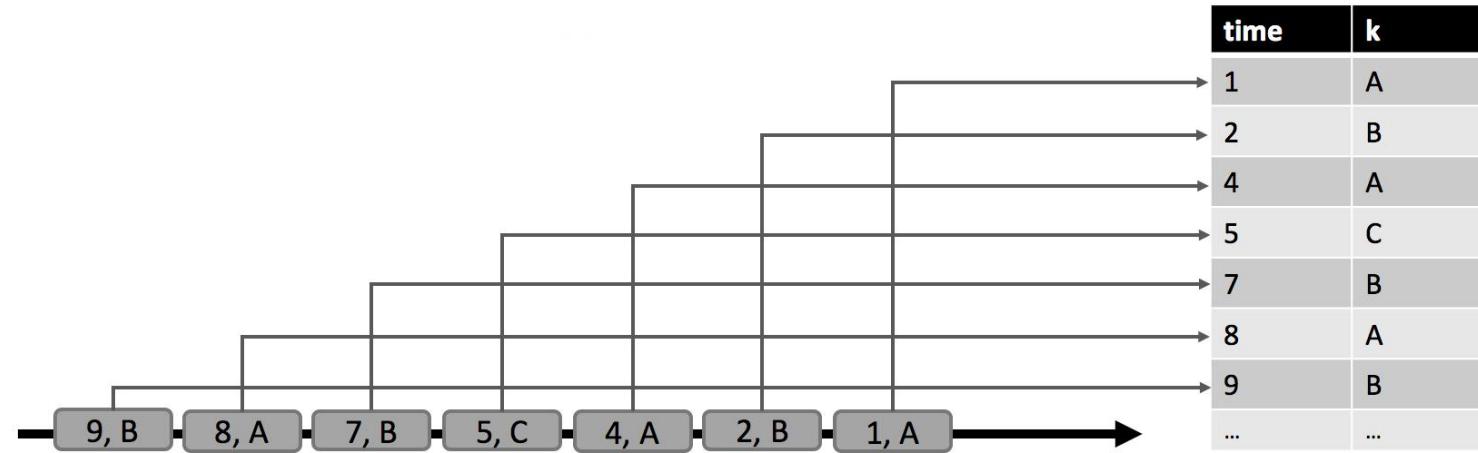
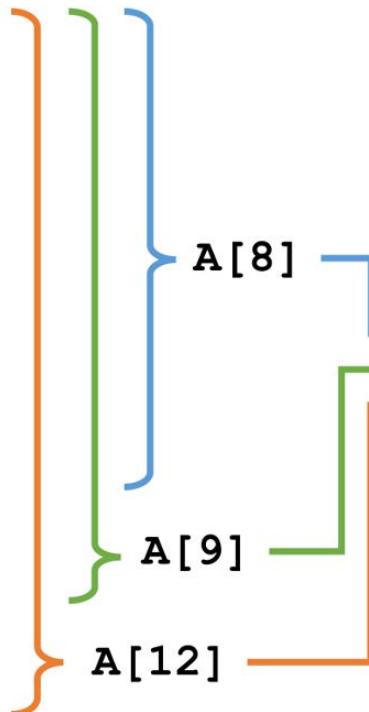






Table A	
time	k
1	A
2	B
4	A
5	C
7	B
8	A

9	B
12	C



q:
SELECT
k,
COUNT(k) as cnt
FROM A
GROUP BY k

q(A[8])

k	cnt
A	3
B	2
C	1

q(A[9])

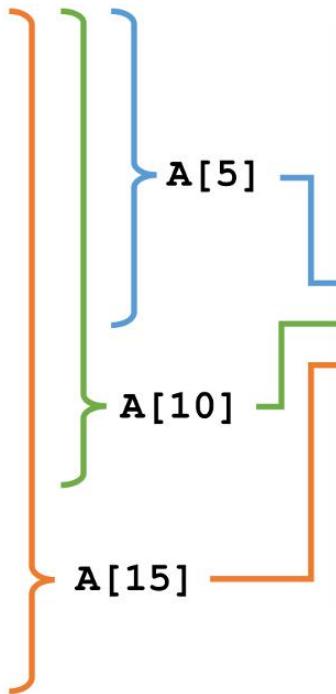
k	cnt
A	3
B	3
C	1

q(A[12])

k	cnt
A	3
B	3
C	2



Table A	
time	k
1	A
2	B
4	A
5	C
7	B
8	A
9	B
11	A
12	C
14	C
15	A



```
q:  
  
SELECT  
    k,  
    COUNT(k) AS cnt,  
    TUMBLE_END(  
        time,  
        INTERVAL '5' SECONDS)  
    AS endT  
FROM A  
GROUP BY  
    k,  
    TUMBLE(  
        time,  
        INTERVAL '5' SECONDS)
```

q(A)		
k	cnt	endT
A	2	5
B	1	5
C	1	5
A	1	10
B	2	10
A	2	15
C	2	15





•



•



•





Table A	
time	k
1	A
2	B
4	A
5	C
7	B
8	A
...	...

```
SELECT  
    k,  
    COUNT(k) AS cnt  
FROM A  
GROUP BY k
```

+ (INSERT), - (DELETE)





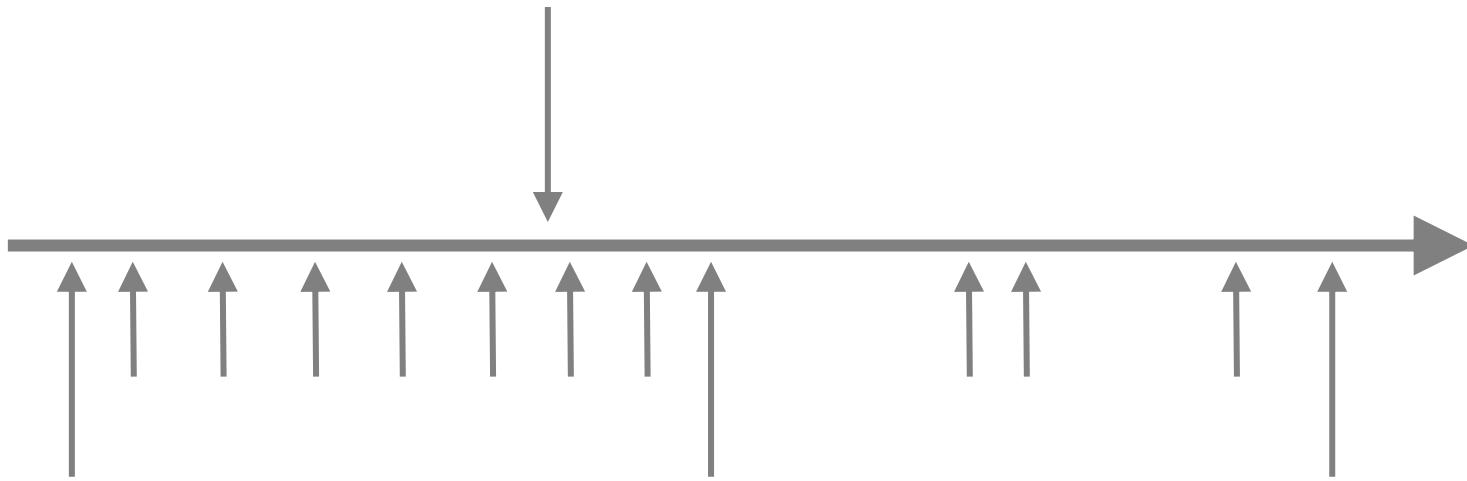
■

Table A	
time	k
1	A
2	B
4	A
5	C
7	B
8	A
...	...

```
SELECT  
    k,  
    COUNT(k) AS cnt  
FROM A  
GROUP BY k
```

+(INSERT), *(UPDATE by KEY), -(DELETE by KEY)









The Stream Processor as a Database

Ufuk Celebi

@iamuce

dataArtisans



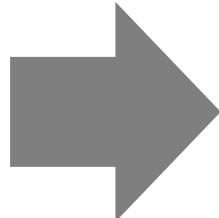
Realtime Counts and Aggregates

The (Classic) Use Case

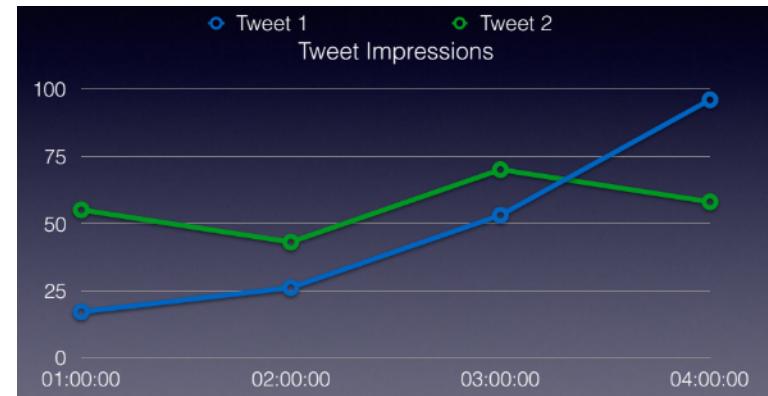
(Real-)Time Series Statistics



Stream
tweet-id: 1, event: url-click, time: 01:01:01
tweet-id: 2, event: url-click, time: 01:01:02
tweet-id: 1, event: impression, time: 01:01:03
tweet-id: 2, event: url-click, time: 02:01:01
tweet-id: 1, event: impression, time: 02:02:02

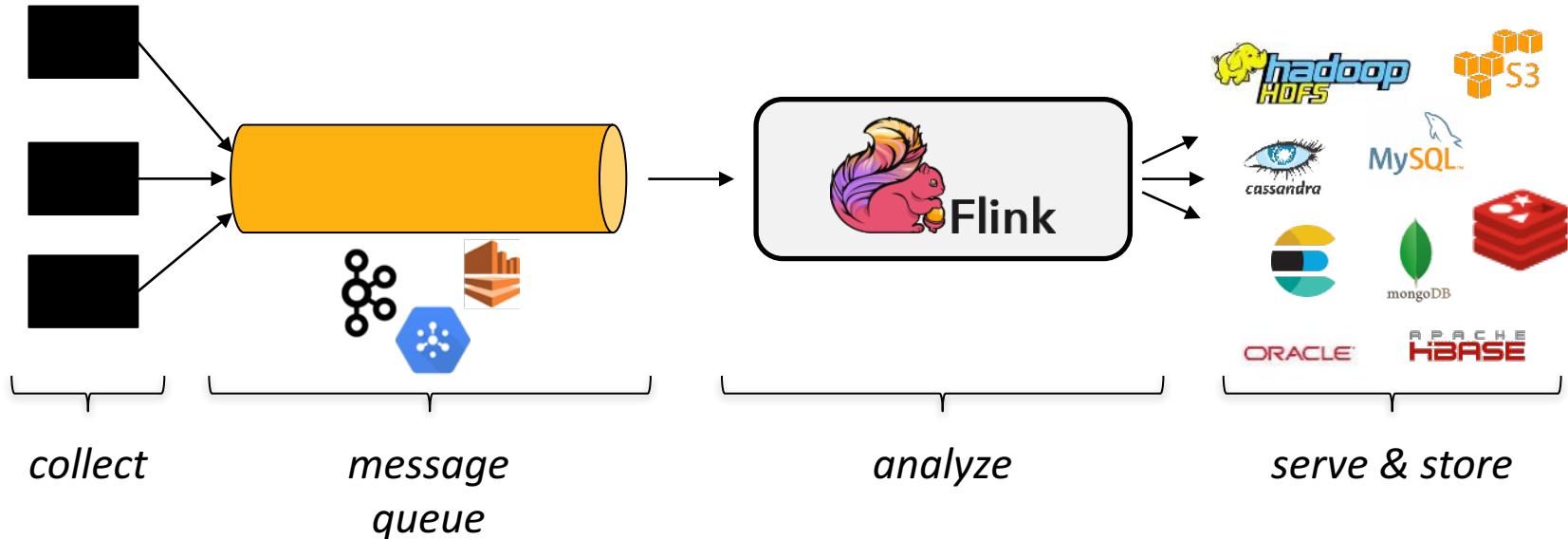


Stream of Events



Real-time Statistics

The Architecture



The Flink Job



```
case class Impressions(id: String, impressions: Long)

val events: DataStream[Event] = env
    .addSource(new FlinkKafkaConsumer09(...))

val impressions: DataStream[Impressions] = events
    .filter(evt => evt.isImpression)
    .map(evt => Impressions(evt.id, evt.numImpressions))

val counts: DataStream[Impressions] = stream
    .keyBy("id")
    .timeWindow(Time.hours(1))
    .sum("impressions")
```

The Flink Job



```
case class Impressions(id: String, impressions: Long)

val events: DataStream[Event] = env
    .addSource(new FlinkKafkaConsumer09(...))

val impressions: DataStream[Impressions] = events
    .filter(evt => evt.isImpression)
    .map(evt => Impressions(evt.id, evt.numImpressions))

val counts: DataStream[Impressions]= stream
    .keyBy("id")
    .timeWindow(Time.hours(1))
    .sum("impressions")
```

The Flink Job



```
case class Impressions(id: String, impressions: Long)

val events: DataStream[Event] = env
    .addSource(new FlinkKafkaConsumer09(...))

val impressions: DataStream[Impressions] = events
    .filter(evt => evt.isImpression)
    .map(evt => Impressions(evt.id, evt.numImpressions))

val counts: DataStream[Impressions]= stream
    .keyBy("id")
    .timeWindow(Time.hours(1))
    .sum("impressions")
```

The Flink Job



```
case class Impressions(id: String, impressions: Long)

val events: DataStream[Event] = env
    .addSource(new FlinkKafkaConsumer09(...))

val impressions: DataStream[Impressions] = events
    .filter(evt => evt.isImpression)
    .map(evt => Impressions(evt.id, evt.numImpressions))

val counts: DataStream[Impressions]= stream
    .keyBy("id")
    .timeWindow(Time.hours(1))
    .sum("impressions")
```

The Flink Job



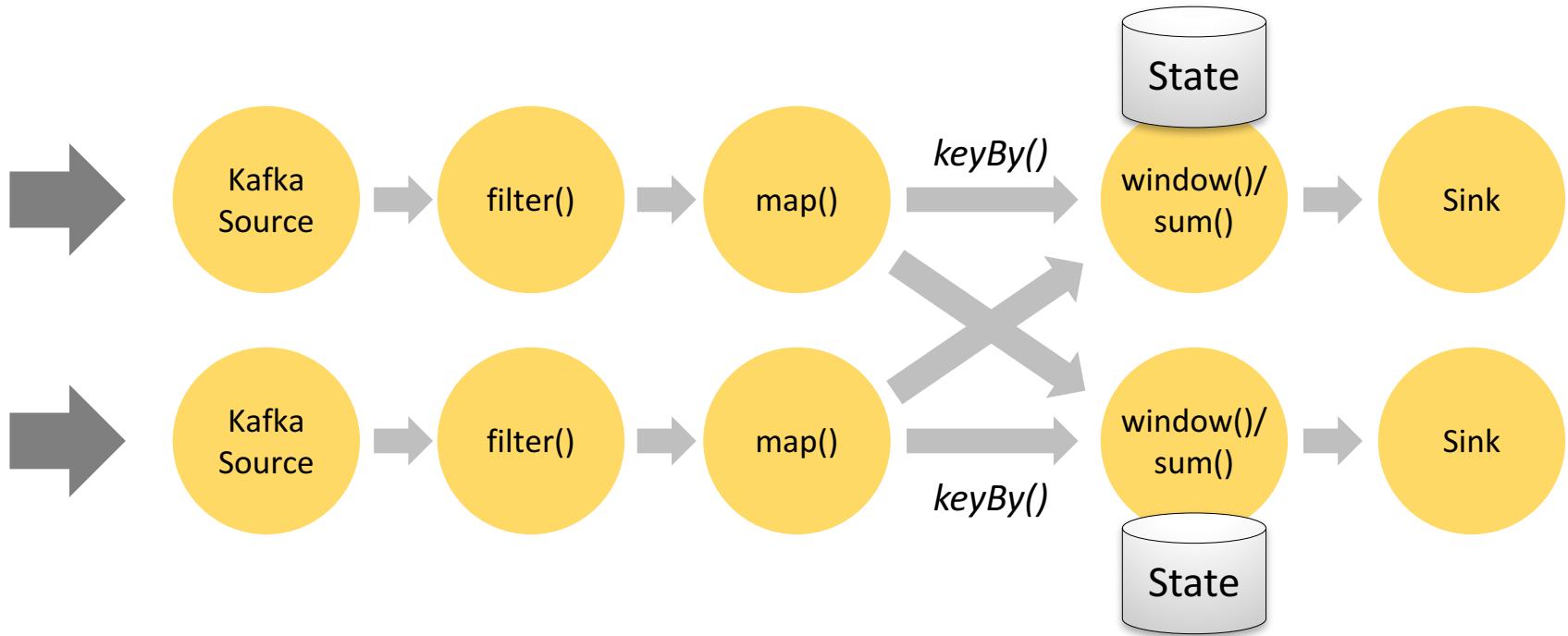
```
case class Impressions(id: String, impressions: Long)

val events: DataStream[Event] = env
    .addSource(new FlinkKafkaConsumer09(...))

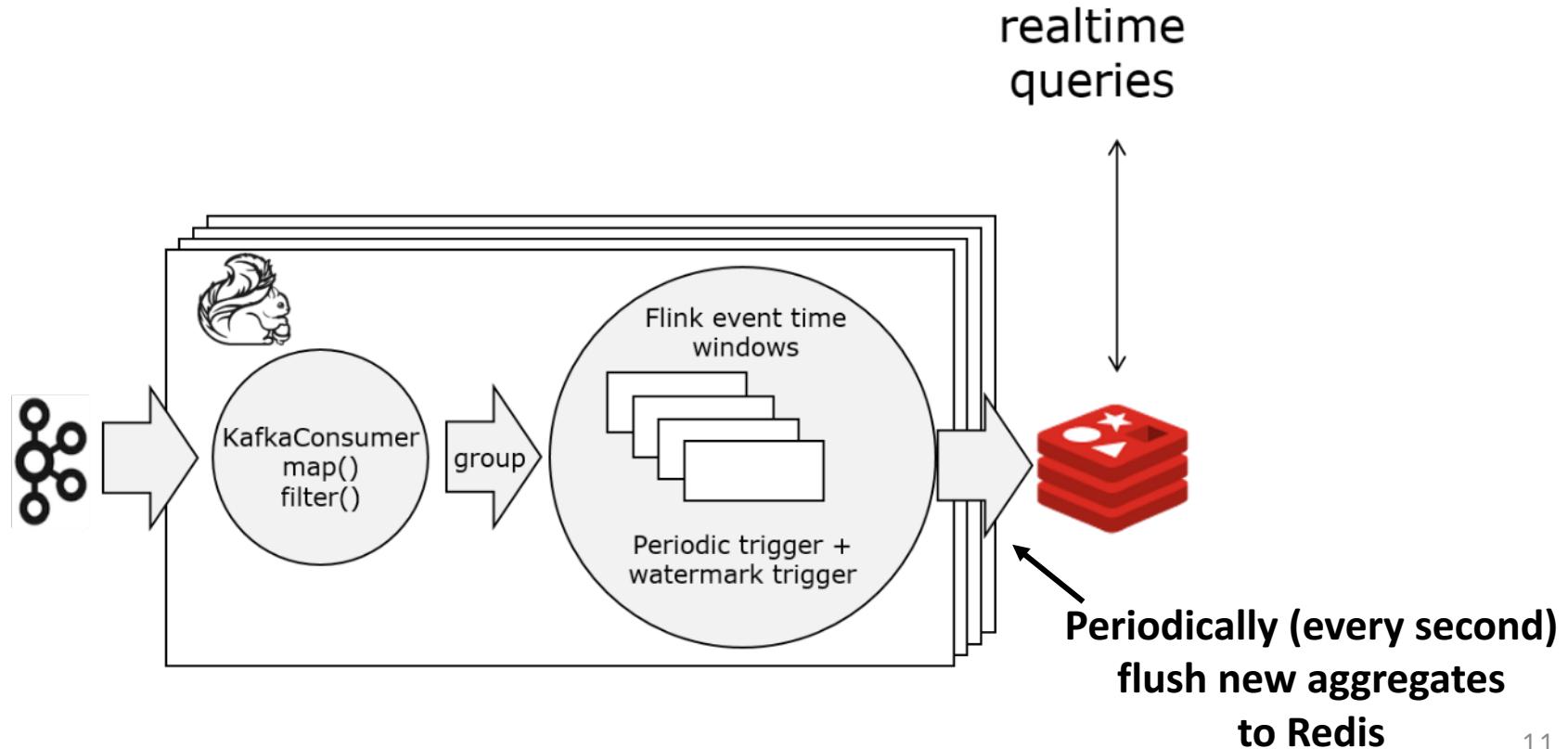
val impressions: DataStream[Impressions] = events
    .filter(evt => evt.isImpression)
    .map(evt => Impressions(evt.id, evt.numImpressions))

val counts: DataStream[Impressions]= stream
    .keyBy("id")
    .timeWindow(Time.hours(1))
    .sum("impressions")
```

The Flink Job

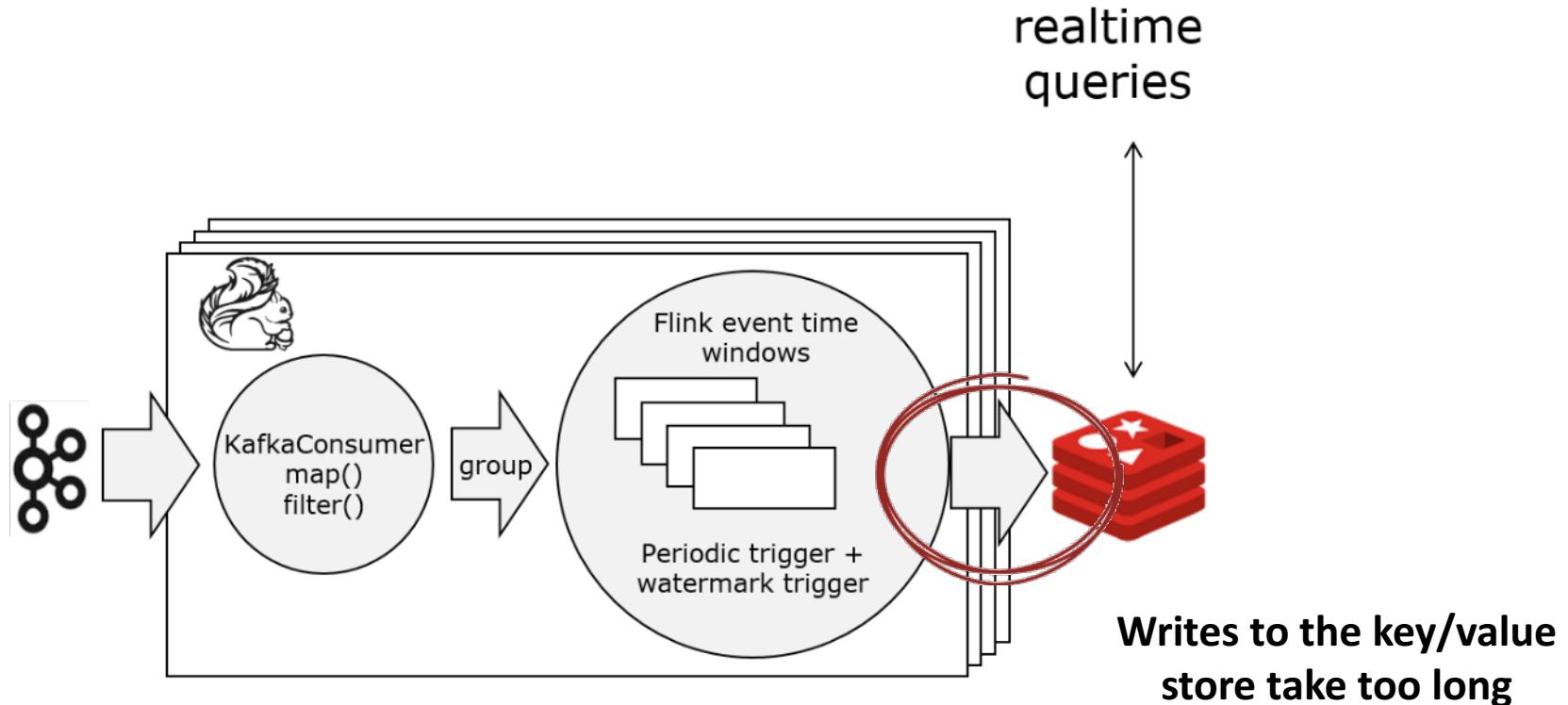


Putting it all together





The Bottleneck

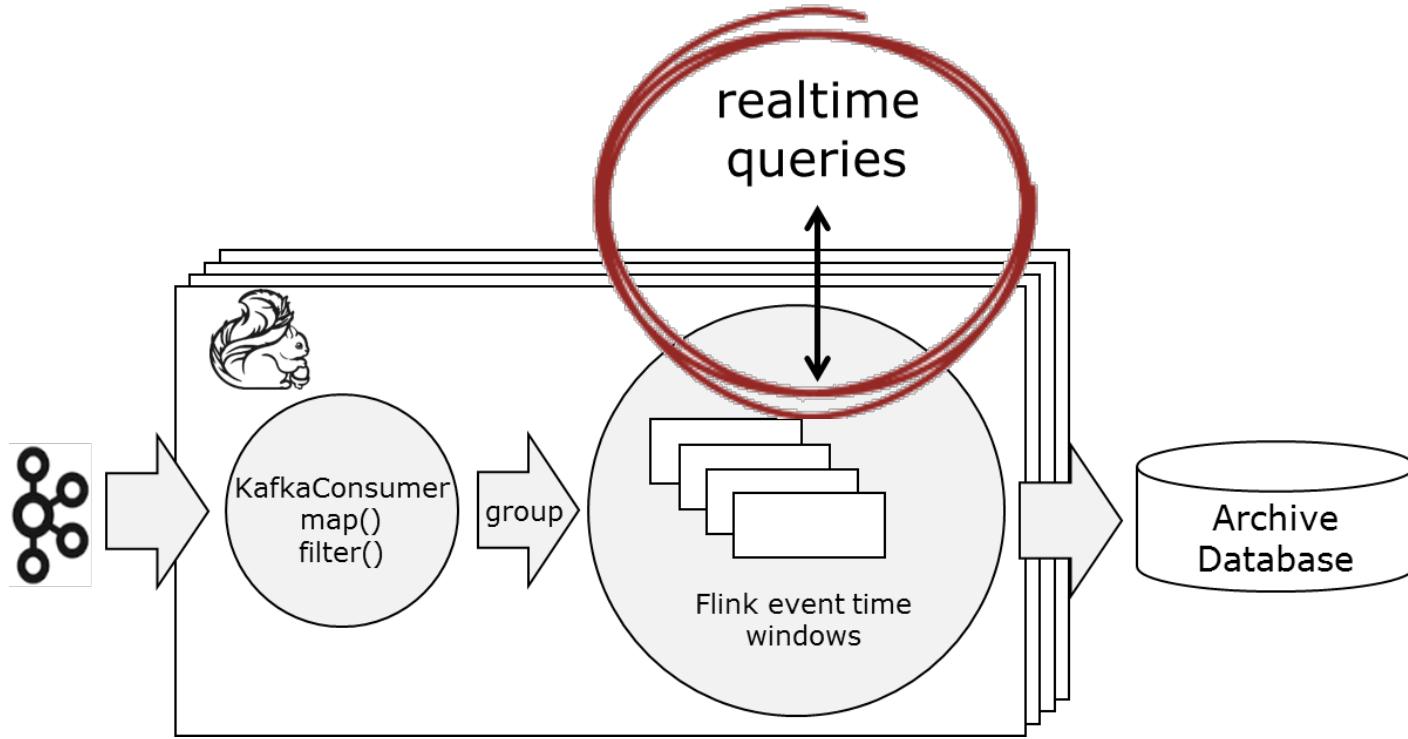




Queryable State

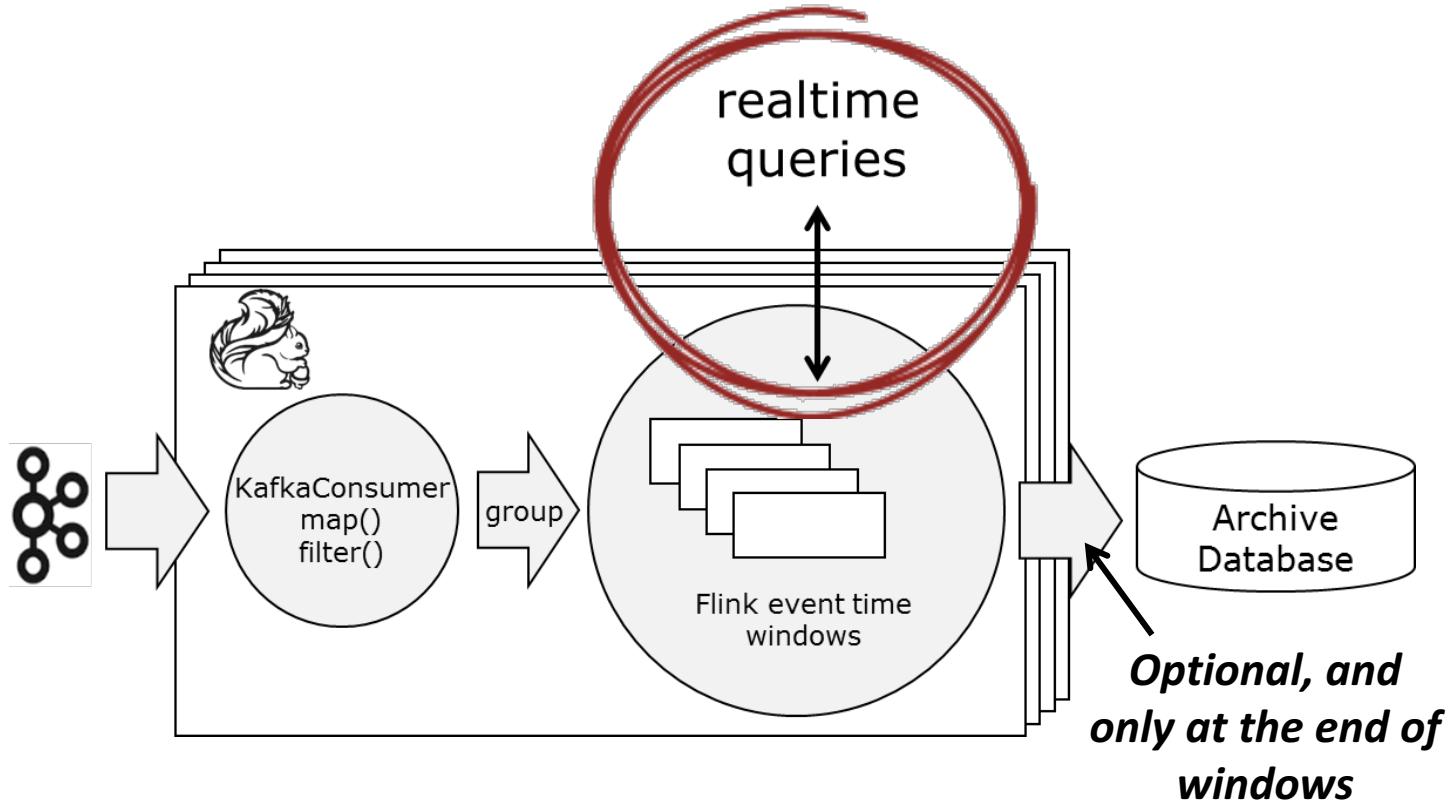


Queryable State

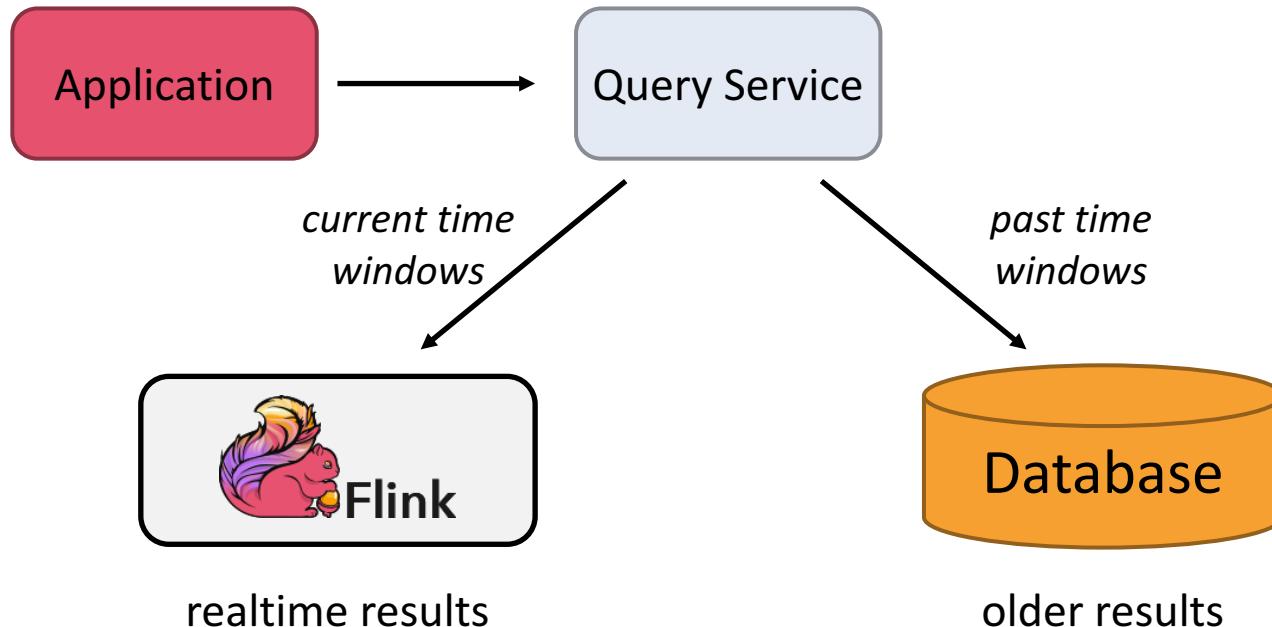




Queryable State



Queryable State: Application View



Queryable State Enablers

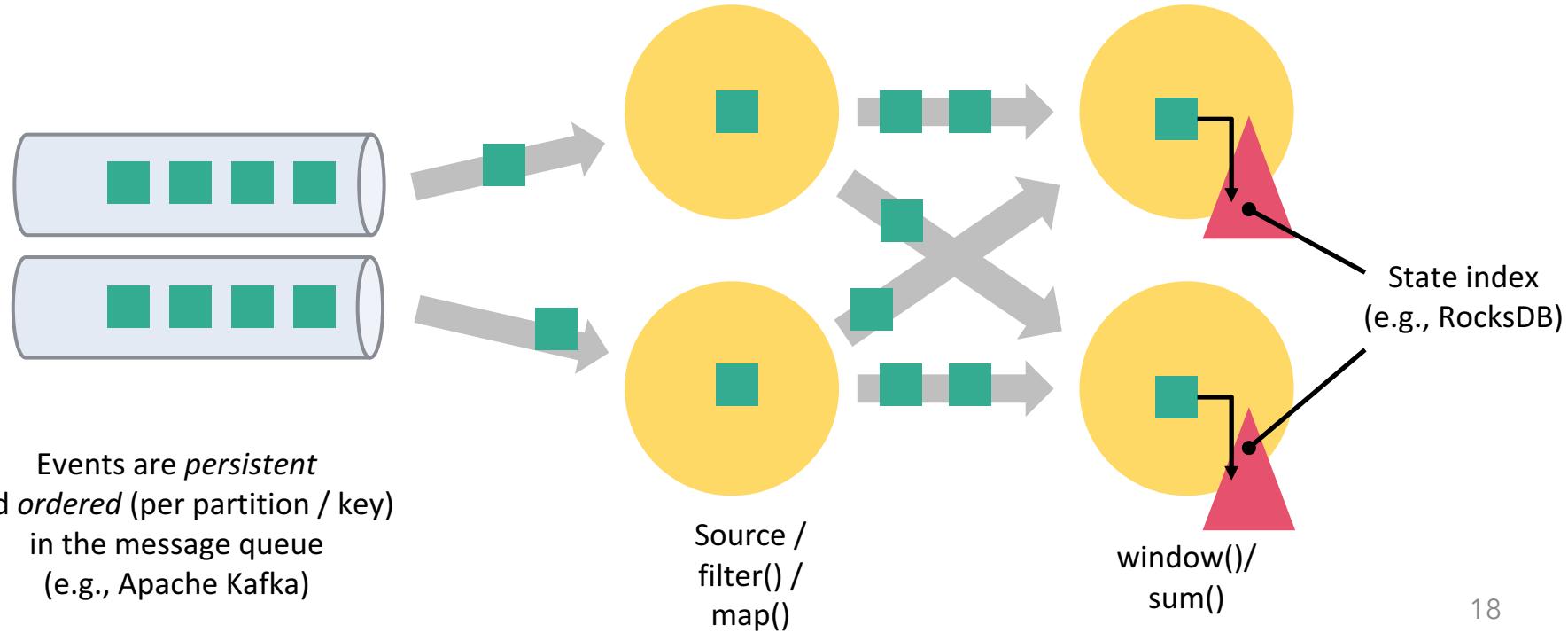


- Flink has state as a **first class citizen**
- State is **fault tolerant** (exactly once semantics)
- State is **partitioned** (sharded) together with the operators that create/update it
- State is **continuous** (not mini batched)
- State is **scalable**

State in Flink



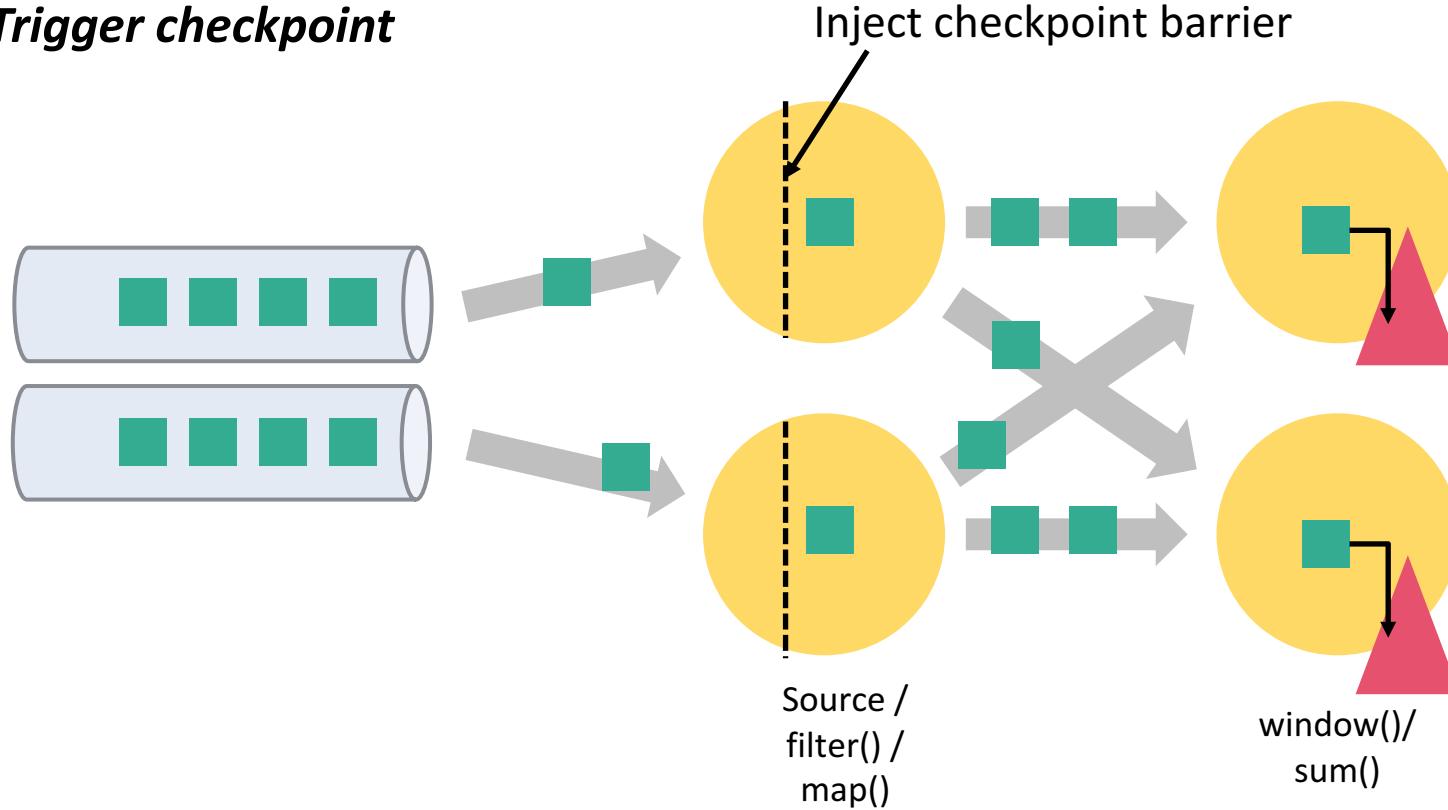
*Events flow **without replication or synchronous writes***



State in Flink



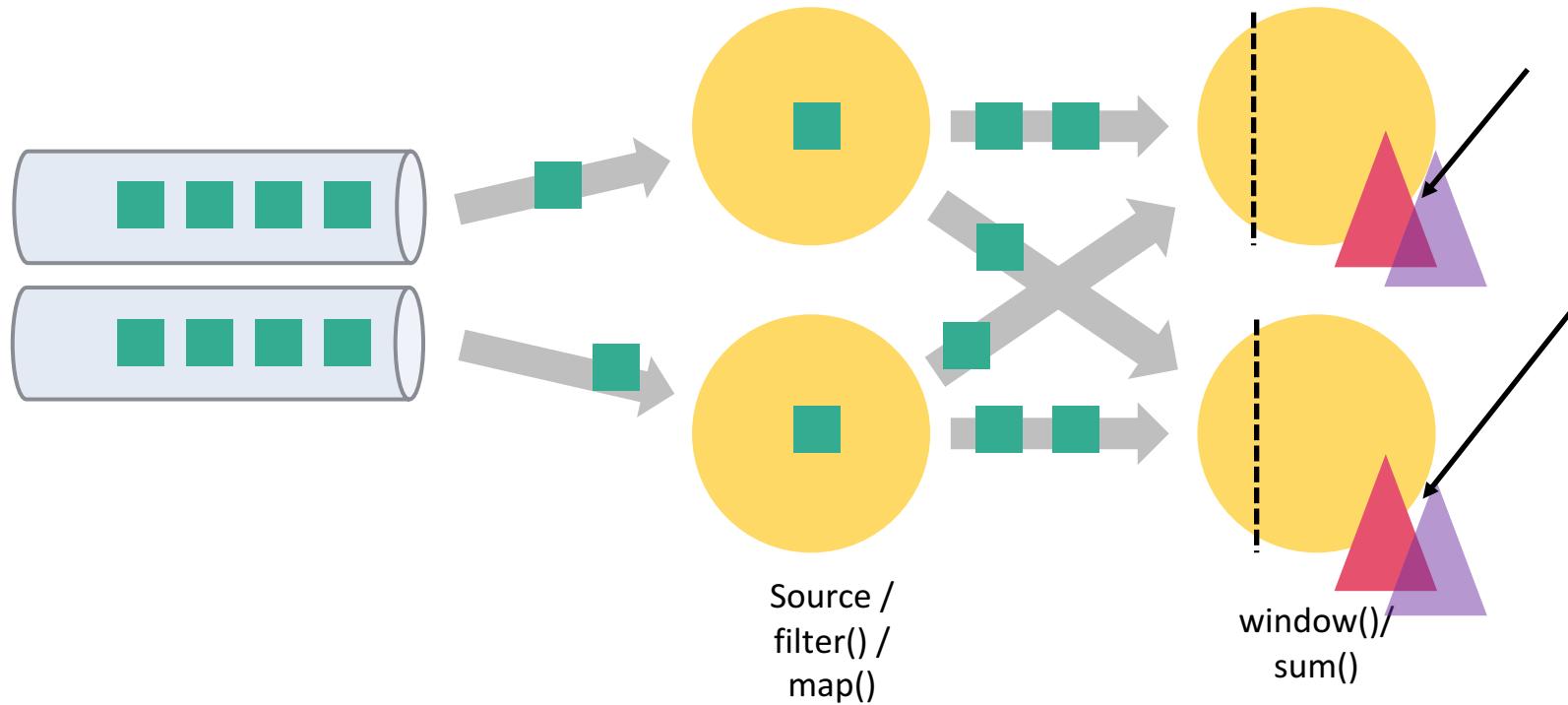
Trigger checkpoint



State in Flink



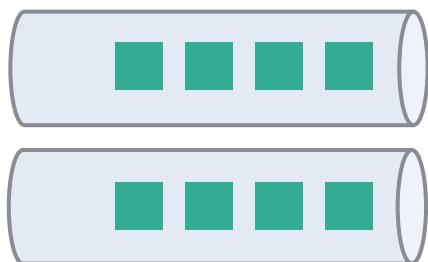
Take state snapshot



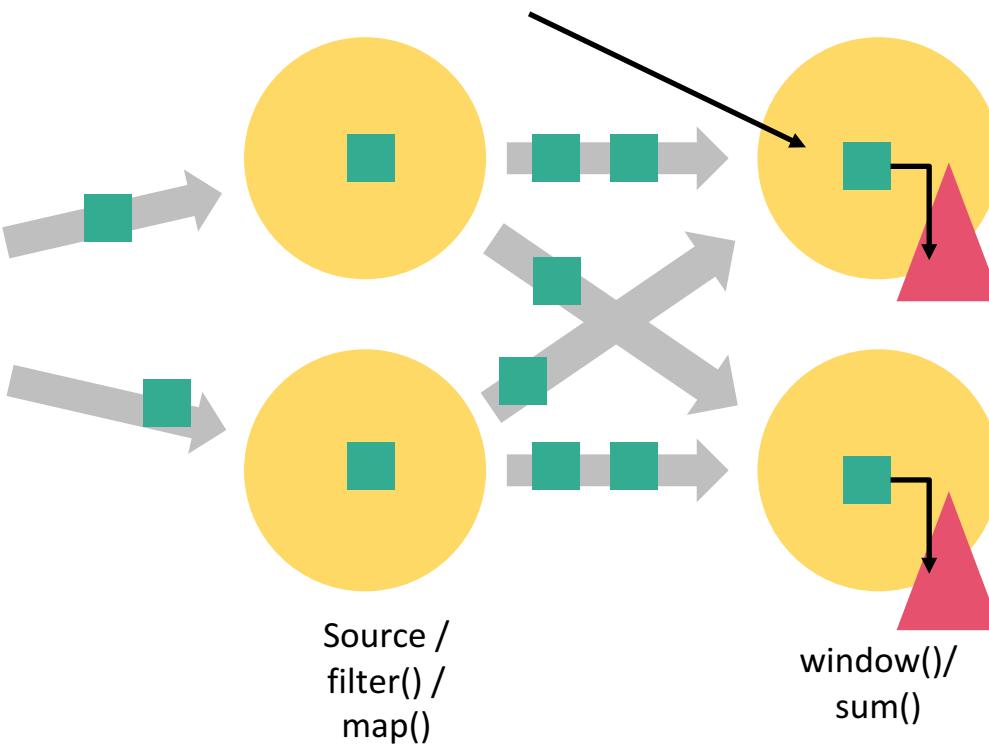
State in Flink



Persist state snapshots



Processing pipeline continues



Durably persist snapshots asynchronously

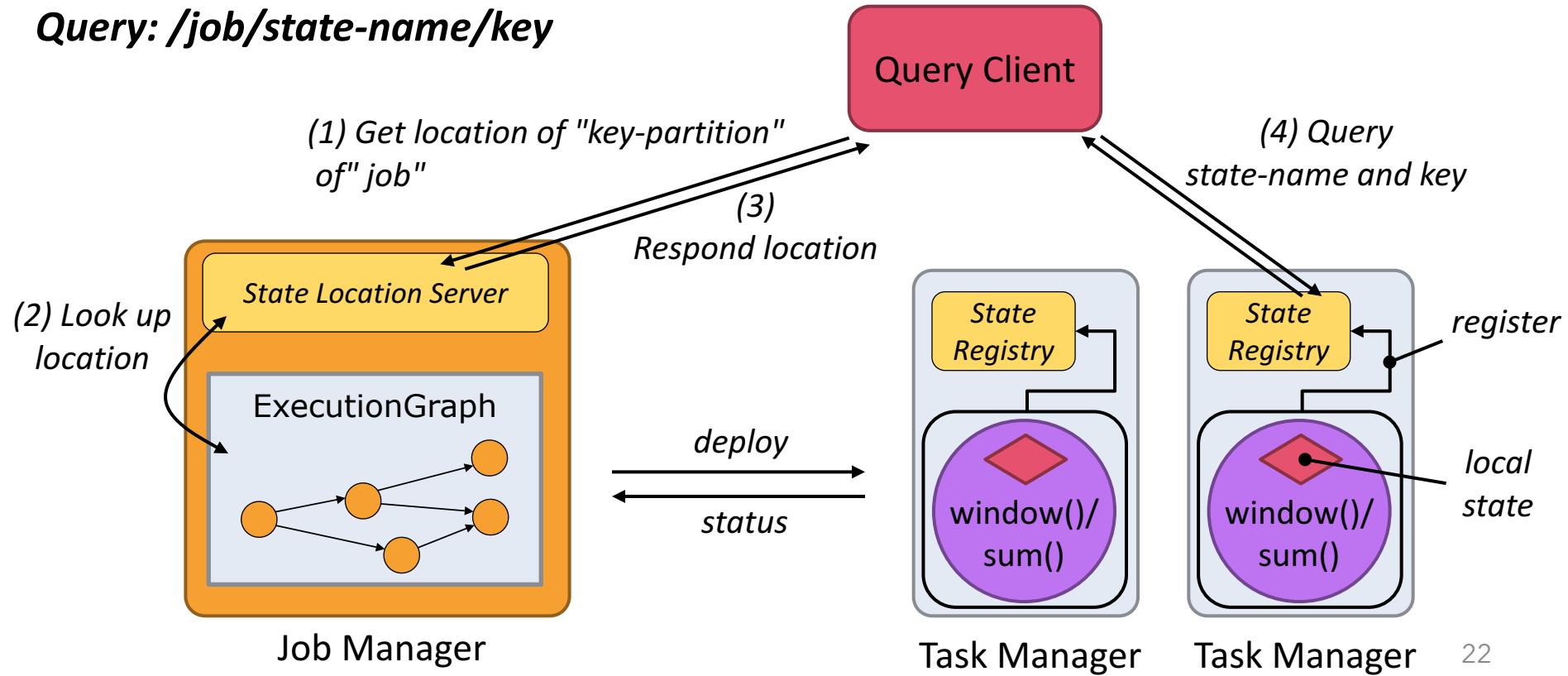


window()
sum()

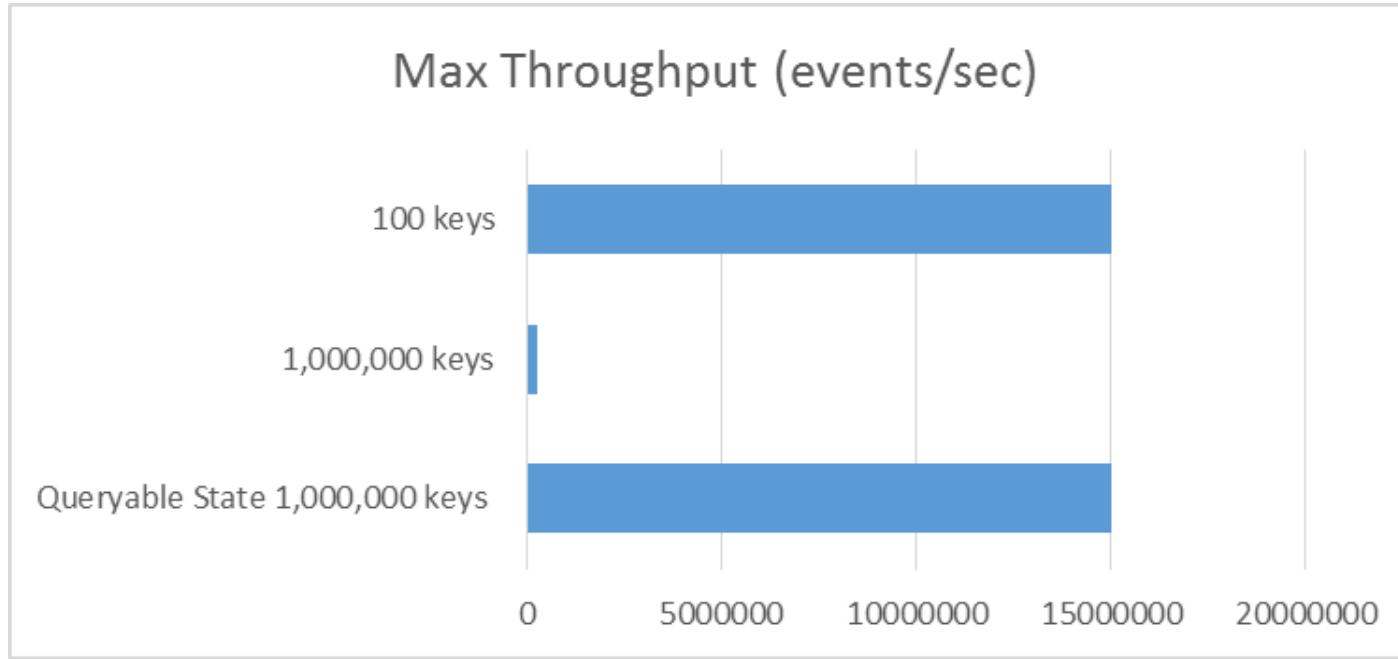
Queryable State: Implementation



Query: /job/state-name/key



Queryable State Performance





Conclusion



Takeaways

- Streaming applications are often not bound by the stream processor itself. **Cross system interaction is frequently biggest bottleneck**
- **Queryable state mitigates a big bottleneck:** Communication with external key/value stores to publish realtime results
- **Apache Flink's sophisticated support for state makes this possible**

Takeaways



Performance of Queryable State

- Data persistence is fast with logs
 - Append only, and streaming replication
- Computed state is fast with local data structures and no synchronous replication
- Flink's checkpoint method makes computed state persistent with low overhead

Questions?

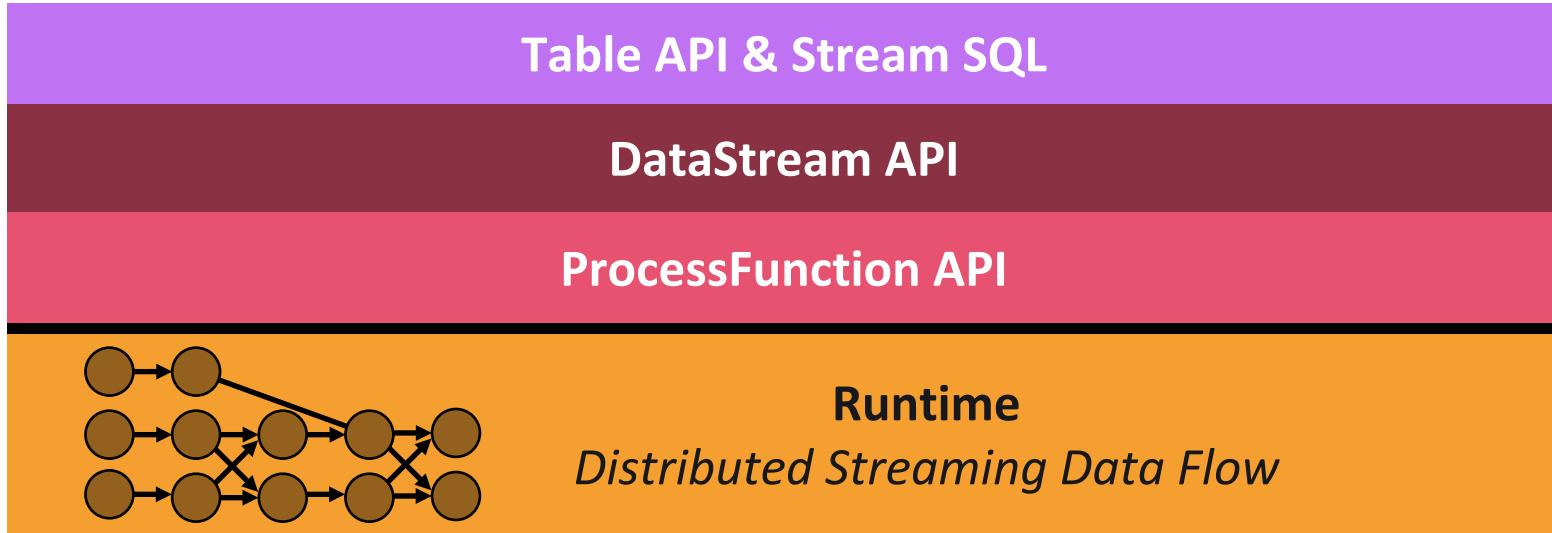


- eMail: uce@apache.org
- Twitter: @iamuce
- Code/Demo: https://github.com/dataArtisans/flink-queryable_state_demo



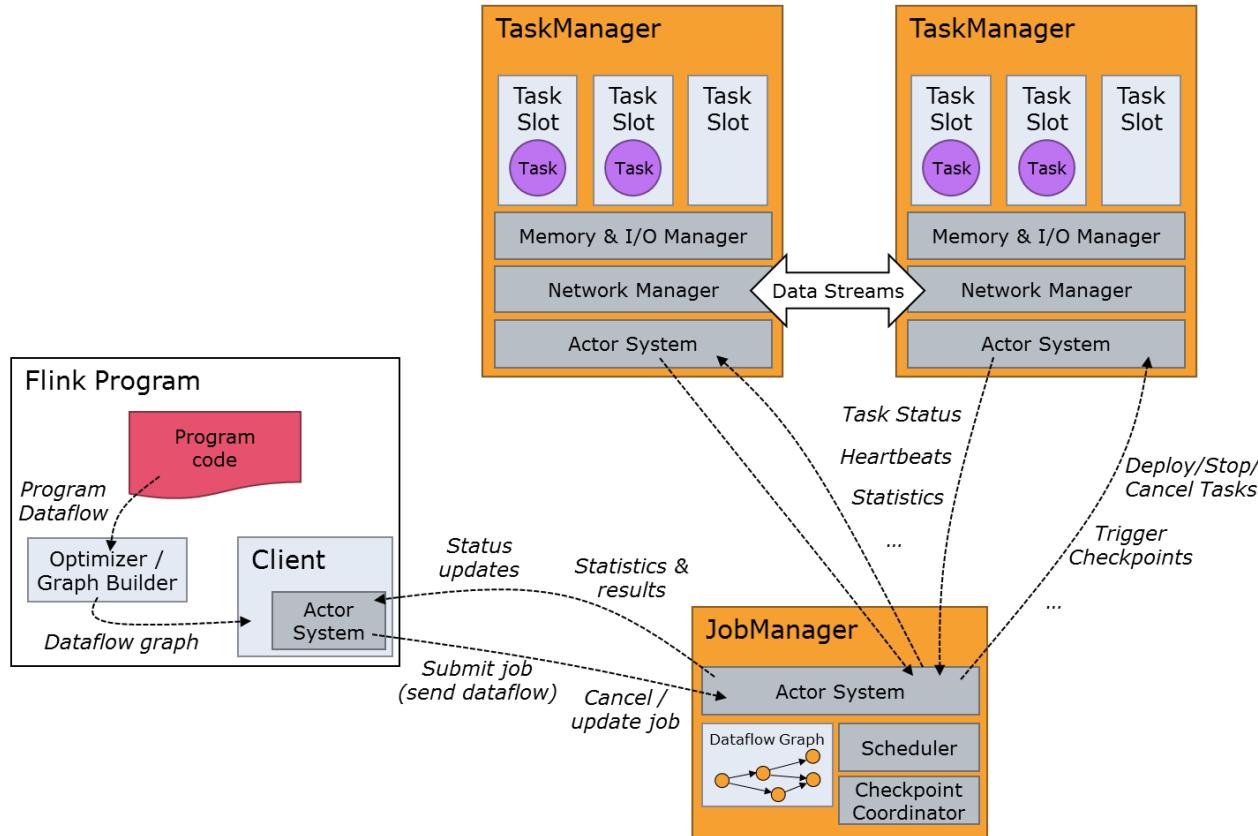
Appendix

Flink Runtime + APIs



Building Blocks: **Streams, Time, State**

Apache Flink Architecture Review



Zero to Streaming

Getting to Production with Apache Flink

April 6, 2017



Strength in Numbers

MediaMath

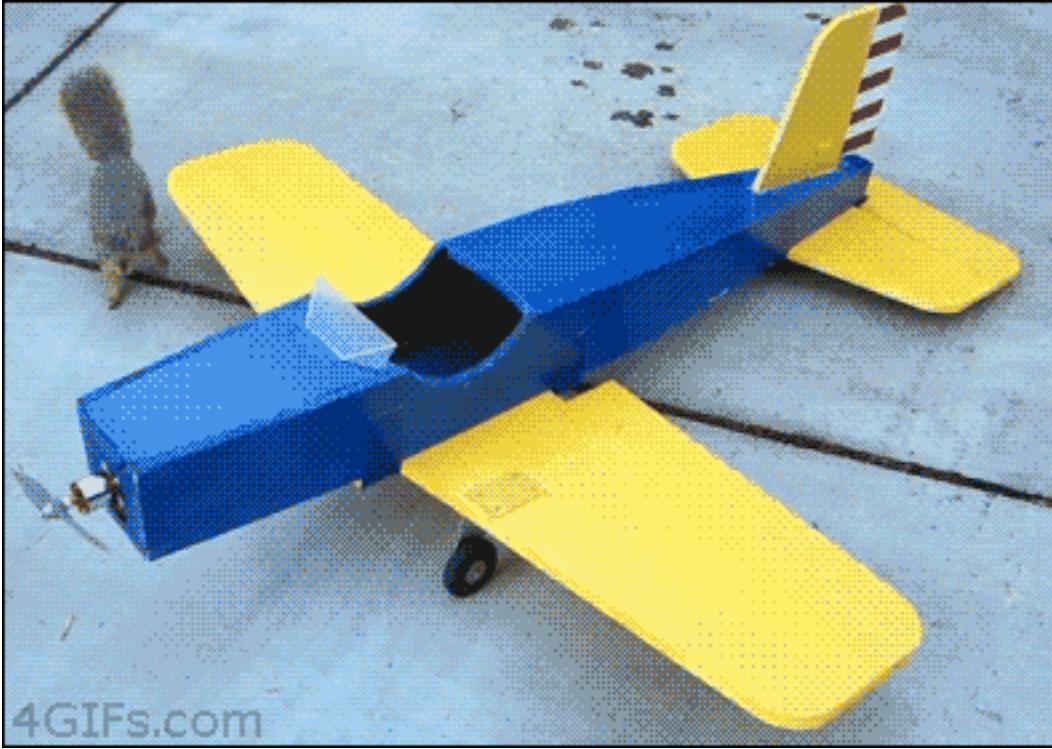
MediaMath's technology and services help brands and their agencies drive business outcomes through programmatic marketing. We believe that good advertising is customer-centric, delivering relevant and meaningful marketing experiences across channels, formats and devices. Powered by advanced machine learning algorithms that buy, optimize and report in real time, our platform gives sophisticated marketers access to first-, second- and third-party data and trillions of digital impressions across every media channel. Clients are supported by solutions and services experts that make it simple to activate our technology. Since launching the first Demand Side Platform (DSP) in 2007, MediaMath has grown to a global company of nearly 700 employees in 15 locations in every region of the world. MediaMath's clients include all major holding companies and operating agencies as well as leading brands across top verticals.

Reporting on Flink

Next Generation Reporting Infrastructure

- Generate user facing reports from raw log data
 - Currently one dataset with 600 million to 1 billion records a day
 - More datasets to come soon
- Data collected from multiple datacenters around the world
- Work is done in AWS

Proof of Concept

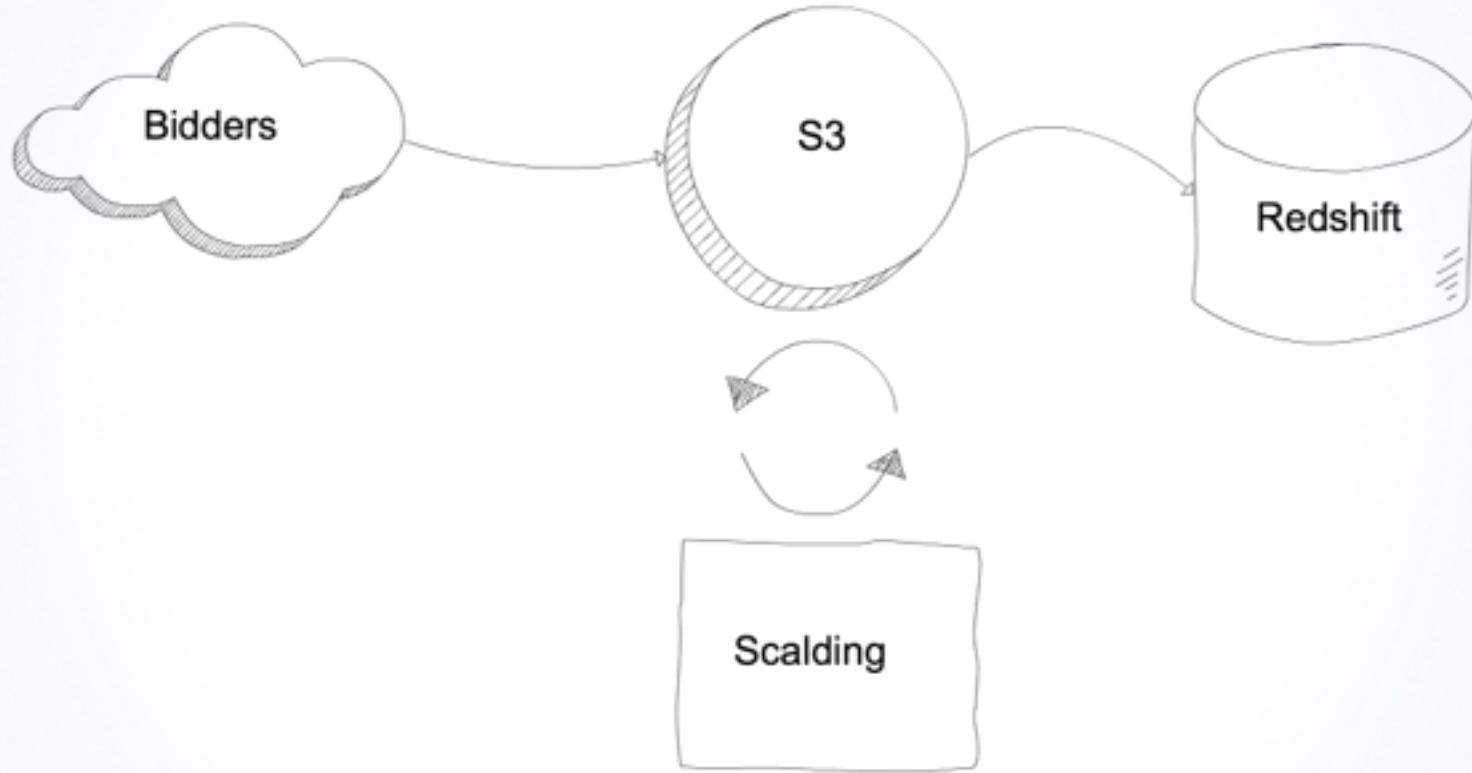


4GIFs.com

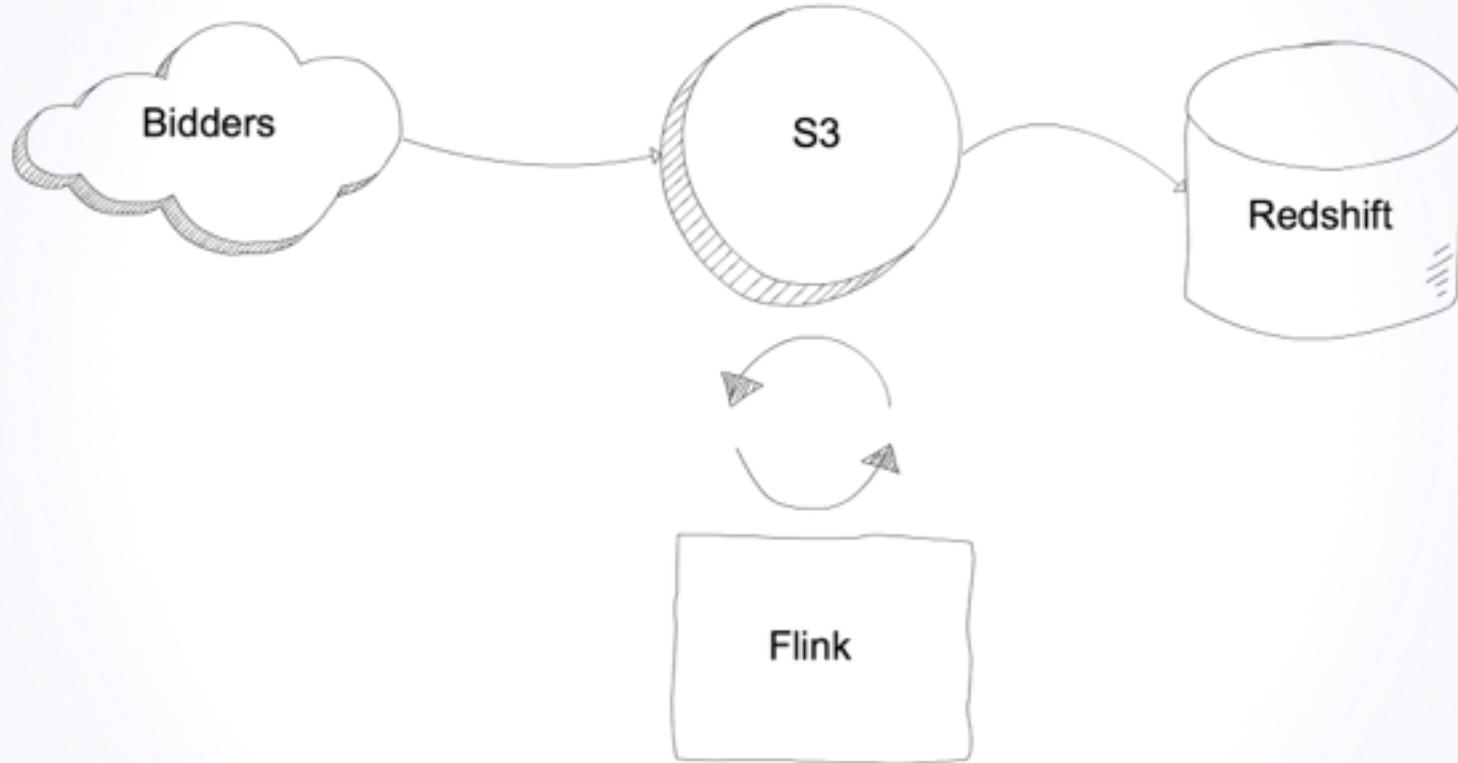
Requirements

- Not a lambda architecture
- Fit into the existing stack
- Can't break the bank
- Must align with my sleep schedule

Current Stack



New Stack



Window Schema

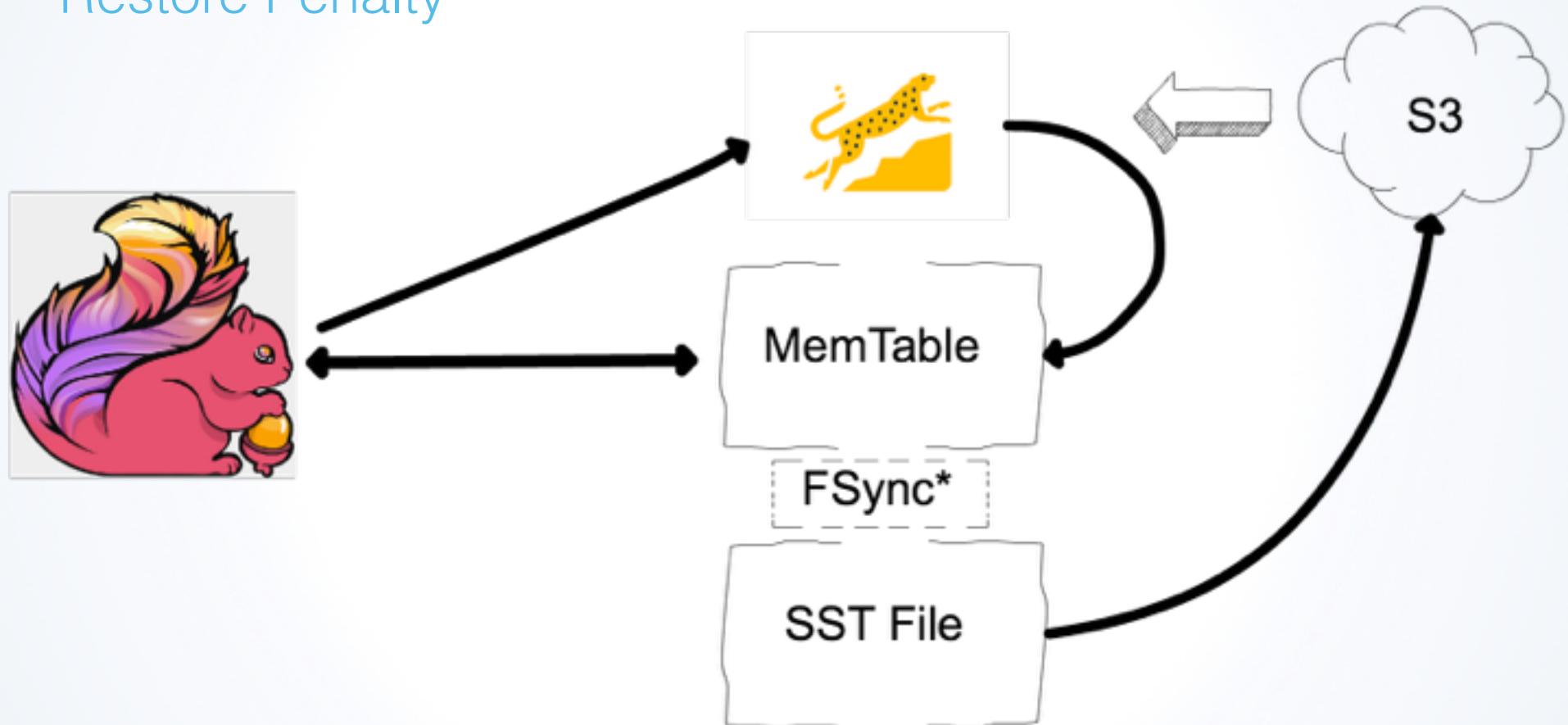
Append

- Simple to reason about
- Difficult to guarantee idempotent updates

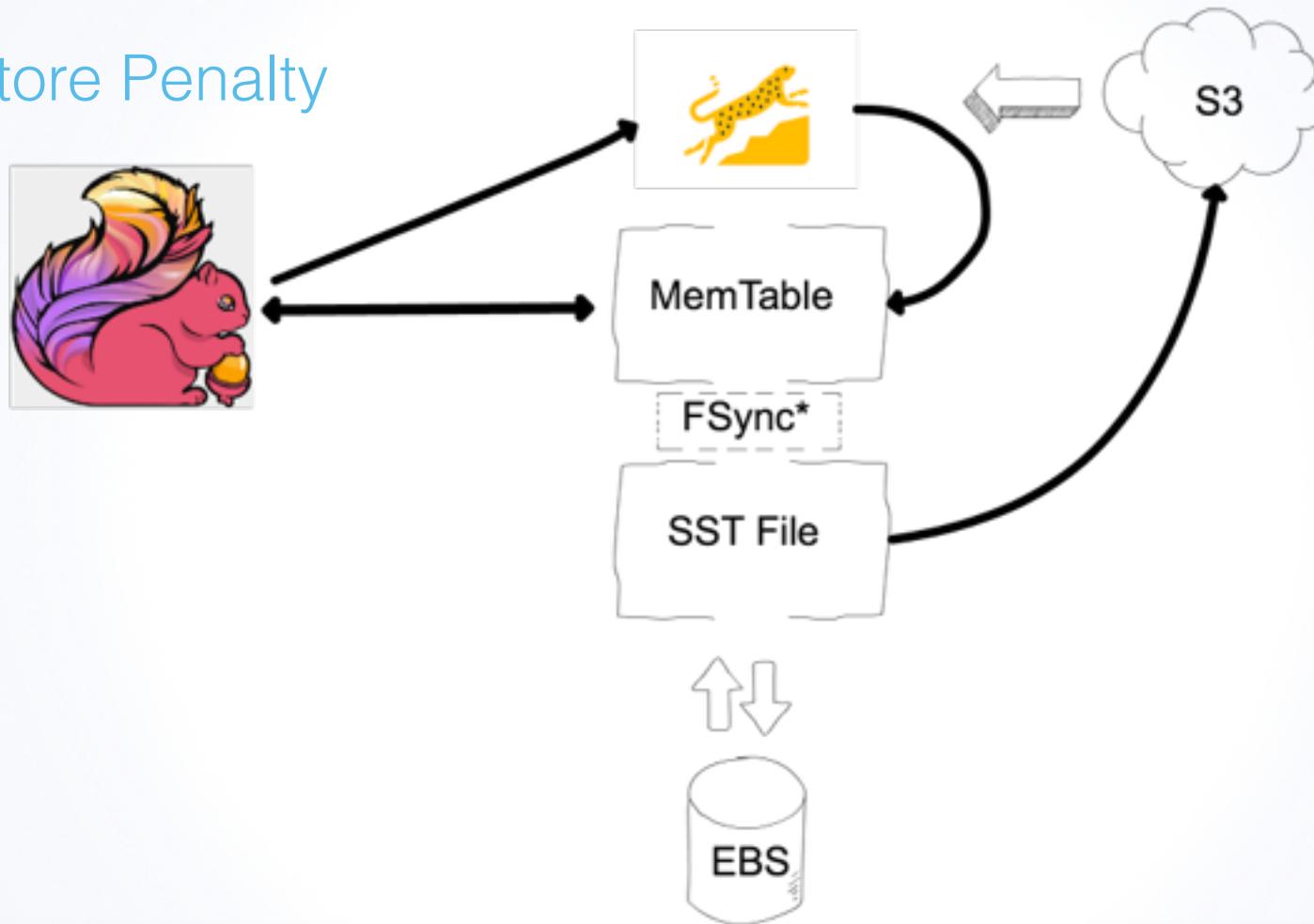
Upsert

- Better aligns with the streaming model
- Results in a larger state

Restore Penalty



Restore Penalty



S3

- S3 is not a file system
- 99.99999999% durability at the cost of eventual consistency
- What does Amazon promise?
 - read-after-write consistency for PUTS of new objects
 - eventual consistency for overwrite PUTS and Deletes

Consistent View

- EMR provides a consistent view of S3
 - DynamoDb is used as a source of truth
- Can we create our own source of truth?
- Flink is expected to be in a consistent state

Roll our own File System

- Treat Flink as a source of truth
- If Flink and S3 disagree
 - Assume S3 is inconsistent
 - Back off
 - Try again
- Works to an extent
 - S3 has no upper-bound on time to consistency

The BucketingSink and S3

```
649     LOG.debug("Moving pending files to final location for checkpoint {}", pastCheckpointId  
650  
651     for (String filename : pendingPaths) {  
652         Path finalPath = new Path(filename);  
653         Path pendingPath = getPendingPathFor(finalPath);  
654  
655         fs.rename(pendingPath, finalPath);  
656         LOG.debug(  
657             "Moving pending file {} to final location having completed checkpoint {}.",  
658             pendingPath,  
659             pastCheckpointId);  
660     }
```



Roll our own Sink

- Treat S3 like a key-value store
- Write all files locally
- Copy to S3 once per checkpoint
- Sink writes are all or nothing

Where did the checkpoint barriers go

- Went nearly 3 months without seeing a checkpoint
- Dug into Flink's internals
- Built tools to better understand what was happening

A small number of windows were never
checkpointing

Taking a step back

- We needed to relearn our dataset
 - 8 hours looks very different than 15 minutes
- We have massive data skew
- 1/3 of all elements belong to 0.3% of the key-space

Handling data skew

- Constrain the key-space
 - Flink scales with windows
- We have three keys common across all reports

Primary Key	
Secondary Key	Metrics
...	...
...	...

Handling data skew

- Reduce the number of elements moving across the network
- Can we build something similar Hadoop's combiner?
- Low level control through the `SingleInputOperatorInterface`

Flink in AWS

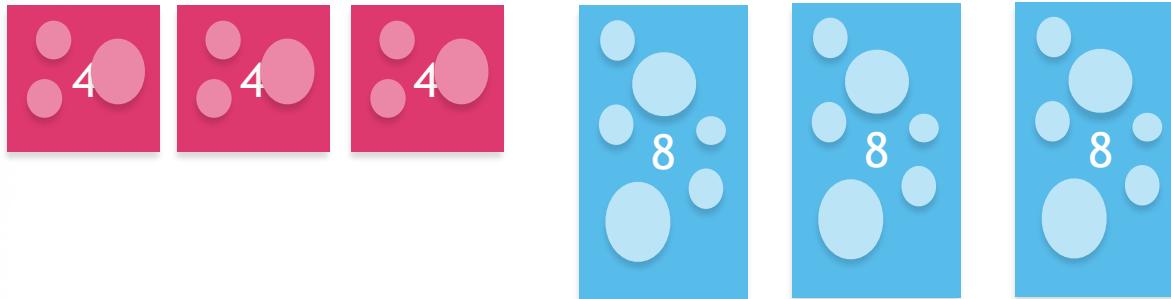
- Cluster
 - Standalone or YARN or ...?
- Storage
 - HDFS or EFS or S3?
- Lease
 - Reserved, On-Demand or Spot?
 - EMR?

Spot rides

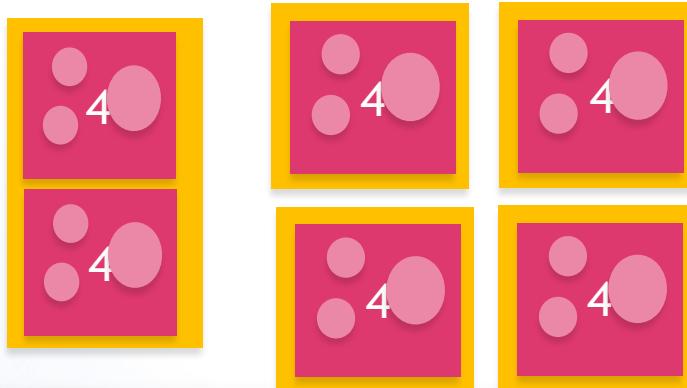


Why YARN?

- Flink Standalone (Homogeneous instances)



- Flink on YARN (Heterogeneous instances)

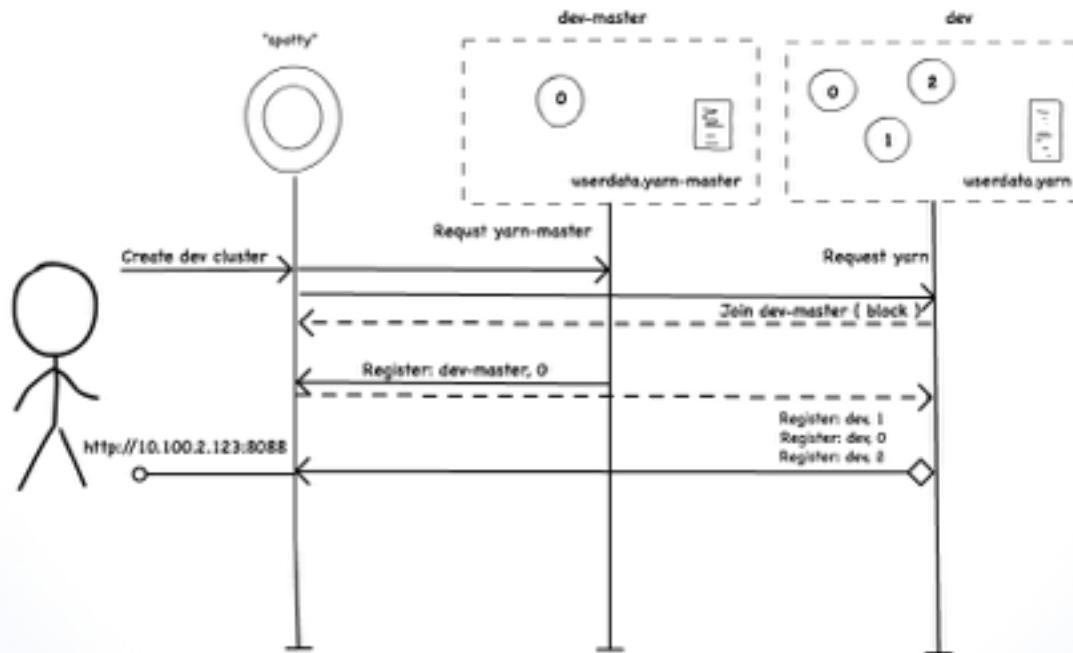


Cluster Initialization Concepts

- Cluster is defined as one or more instances of a Config, keyed by Name
- Config is “userdata”, a user supplied shell script called as root on instance init.
- Cluster “Join” - like Workflow join - means that one cluster can block on another’s init.

Spot Cluster Creation

Initialize "dev" YARN cluster



Running Flink

- Initialized by shell process, managed, monitored with YARN REST apis
- Flink Job created and managed with Flink REST apis (poll for events).
- Job exceptions and Checkpoint events are logged to database.

Monitoring

■ Health checks:

- Use http pings and YARN REST api to verify cluster health.
- Expose and publish Flink job Current Low Watermark
 - Alert on Watermark lag.
 - Trick! Use jolokia:
 - javaagent:/opt/jolokia-jvm-1.3.5-agent.jar=port=0,host=0.0.0.0
- Publish checkpoint times and durations
 - Alert on checkpoint lag.
- collectd/graphite for machine stats (memory, load, disk).

Spotty-agent

- Watches for spot reclaim (2 minute warning).
- Posix Passthrough: remote exec scripts and shell commands (init checks, stats).

Cluster Failover

1. Agent receives warning and sends notification (first one wins).
2. Monitor initiates Flink “cancel_with_savepoint”.
3. Monitor requests a whole new cluster.
4. When new cluster is up:
 1. Verify final checkpoint (may or may not be from cancel).
 2. Restart Flink Session
 3. Restart Flink job.
 4. Delete old clusters’ EC2 instances (may already be gone).

Task Manager on YARN config

- EC2 compute (C-type) instances are appx 1.75 gig ram/VCPU.
(reserving 600-700MB for linux os)
- Min YARN process = 1.75G
- TM of four slots = $4 \times 1.75 = 7G$
- After 5% Flink “yarn tax”, each YARN process = ~6.65G
- Set TM Heap cut-off to .25, leaving 1.6G for RocksDB

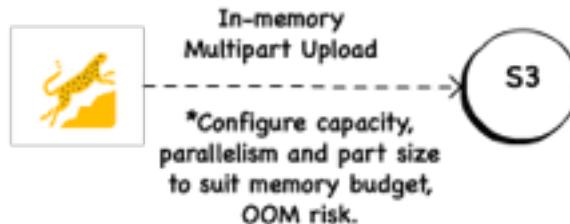


RocksDB Snapshot to S3

- S3N and S3A default:



- S3A Fast Upload:



- Hadoop 3.0 code looks very promising!

Dev Tools

- Live metrics

- Provide context to checkpoint and other pipeline problems

- Remote debugging

- Effective, but limited scope

- Cluster Logging?

- Continuous process - cannot use YARN log aggregation

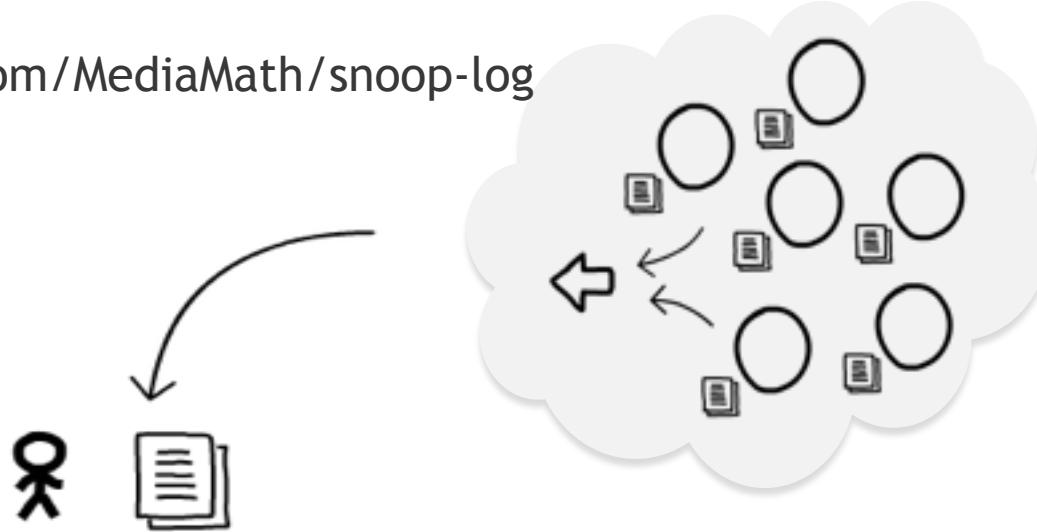
- Many (100s) of slots - each with own operator DAG

- Spot Instances - log aggregators not easy to integrate, overkill for problem

Snoop “Loggy” Log

Tail your ephemeral cluster ephemerally!

<https://github.com/MediaMath/snoop-log>



flink/lib/logback-snoop.jar

Hacking Flink

- (demo)

THANK YOU!

Seth Wiesman

Cliff Resnick

4 World Trade Center, 45th Floor
New York, NY 10007



Strength in Numbers

Apache Flink: The Latest and Greatest



Jamie Grier
@jamiegrier

dataArtisans

data-artisans.com

dataArtisans



Original creators of **Apache Flink®**



Providers of the
dA Platform, a supported
Flink distribution

The Latest Features

- ProcessFunction API
- Queryable State API
- Excellent support for advanced applications that are:
 - Flexible
 - Stateful
 - Event Driven
 - Time Driven

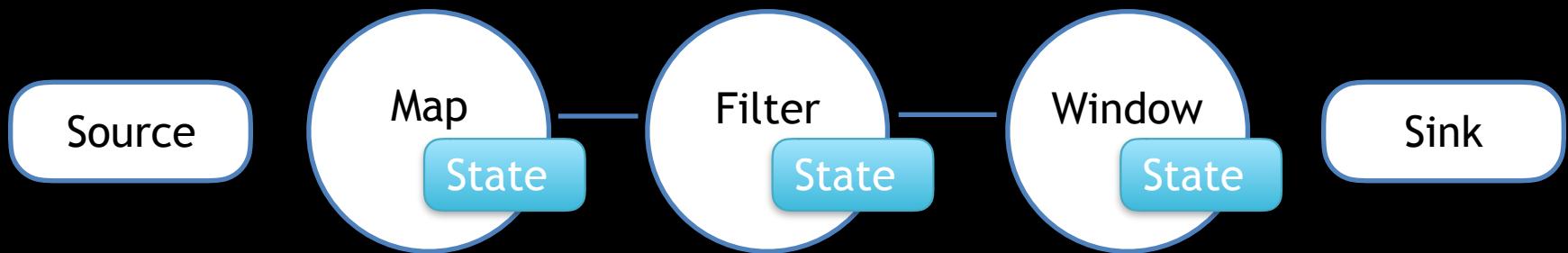
The Latest Features - Quick Overview

- Rescalable State
- Async I/O Support
- Flexible Deployment Options
- Enhanced Security

Rescalable State

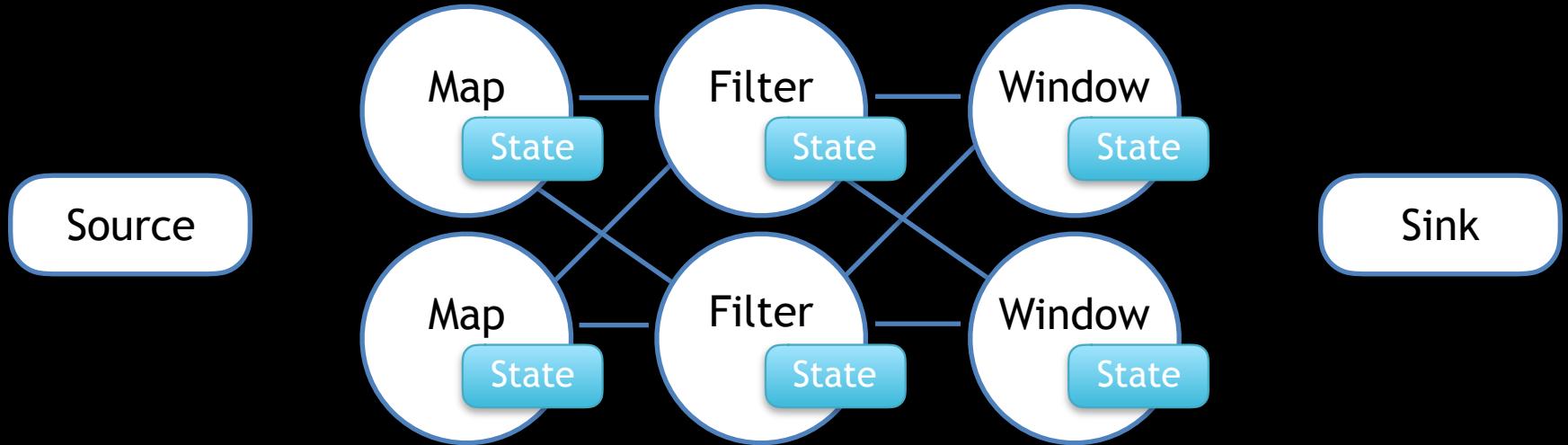
- Separates state parallelism from task parallelism
- Enables autoscaling integrations while maintaining stateful computations
- Handled efficiently via key groups

Rescalable State



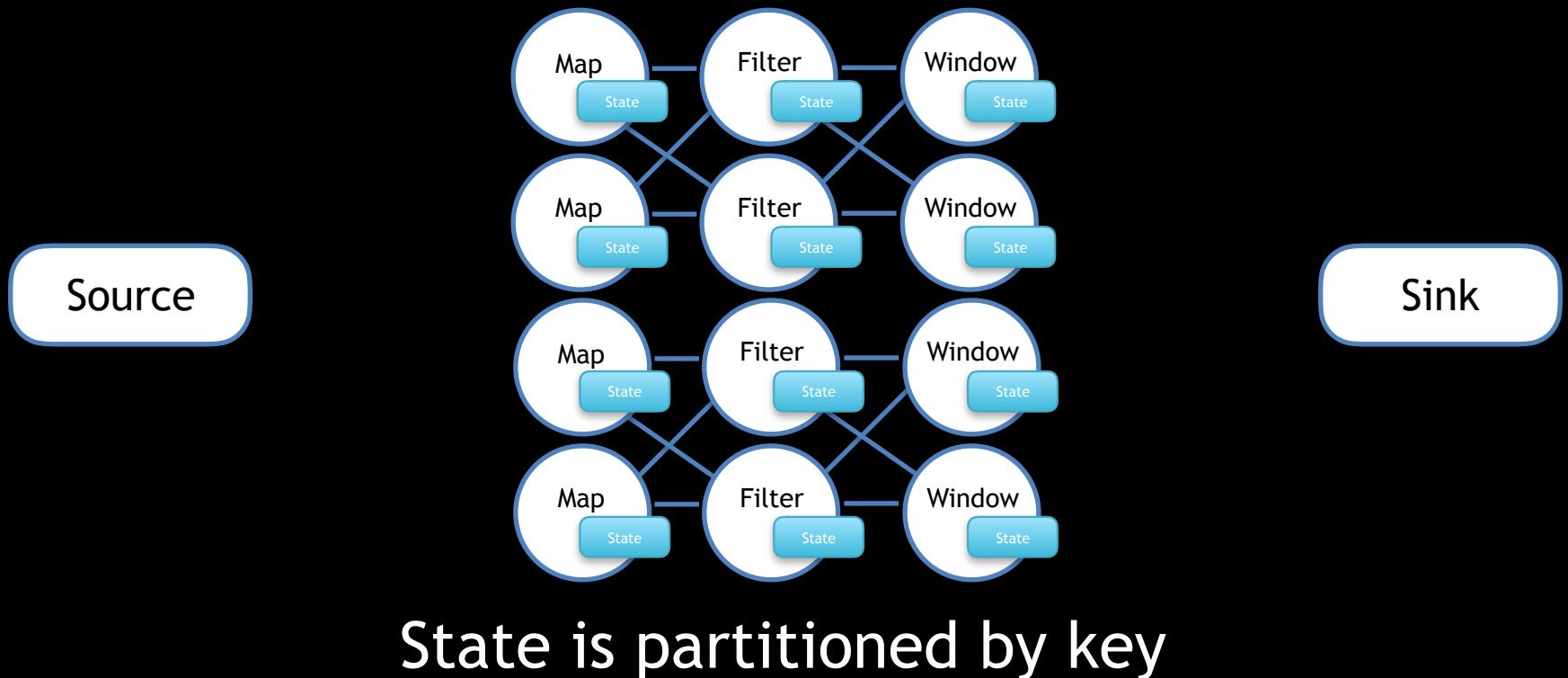
State is partitioned by key

Rescalable State

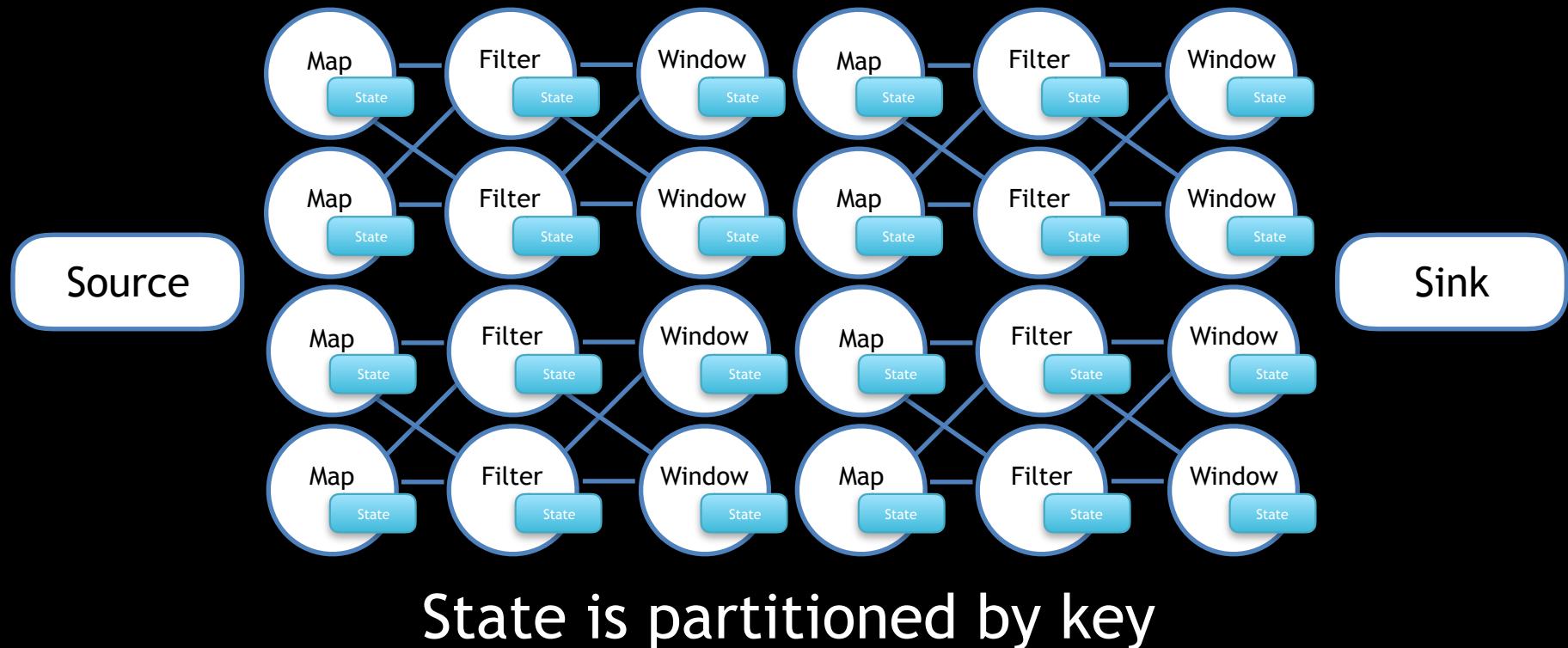


State is partitioned by key

Rescalable State



Rescalable State



Flexible Deployment Options

- YARN
- Mesos
- Docker Swarm
- Kubernetes



Flexible Deployment Options

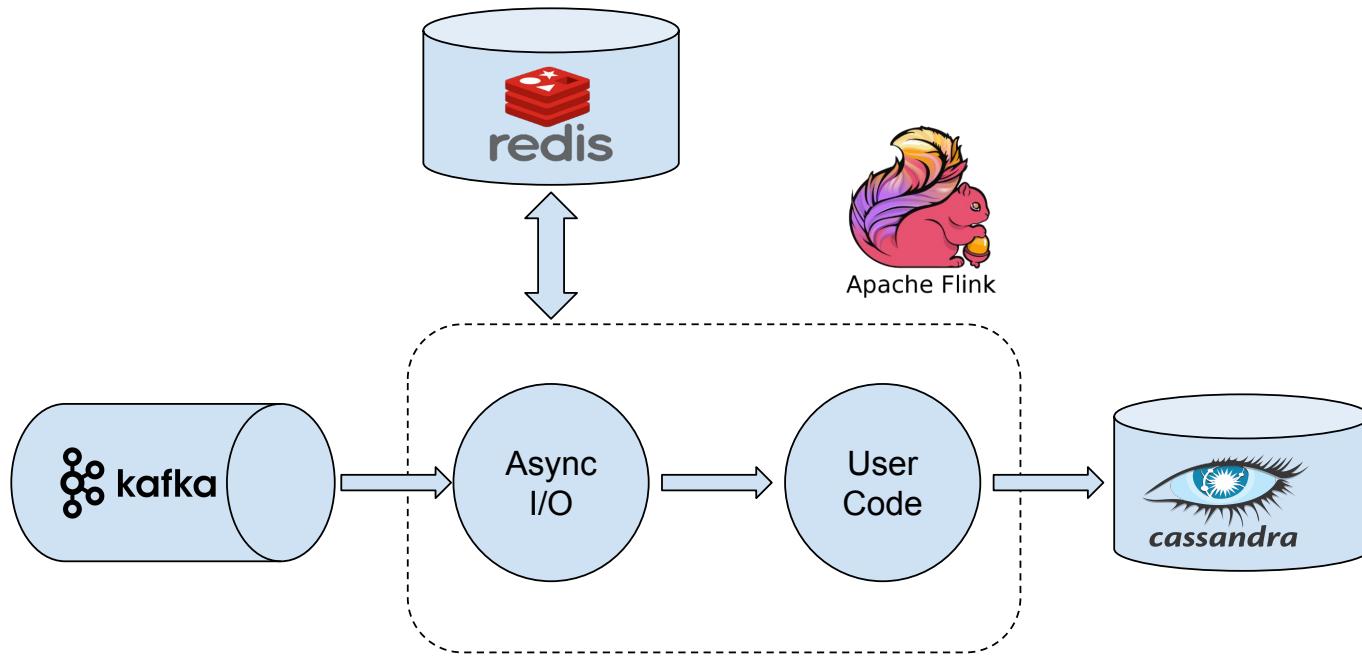
- DC/OS
- Amazon EMR
- Google Dataproc



Asynchronous I/O Support

- Make asynchronous calls to external services from streaming job
- Efficiently keeps configurable number of asynchronous calls in flight
- Correctly handles failure scenarios - restarts failed async calls, etc

Asynchronous I/O Support



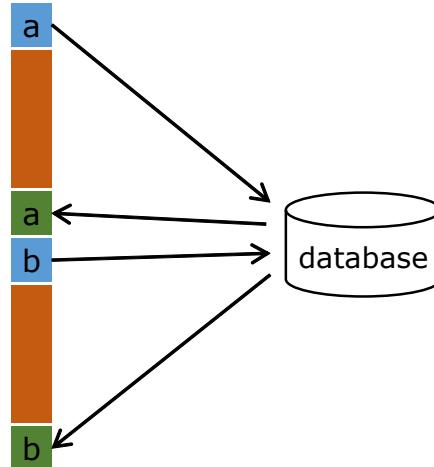
Asynchronous I/O Support

Little's Law:

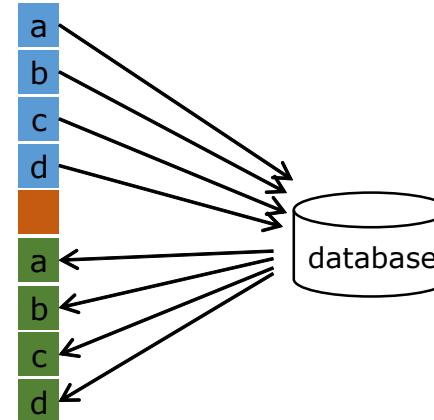
throughput = occupancy / latency

Asynchronous I/O Support

Sync. I/O



Async. I/O

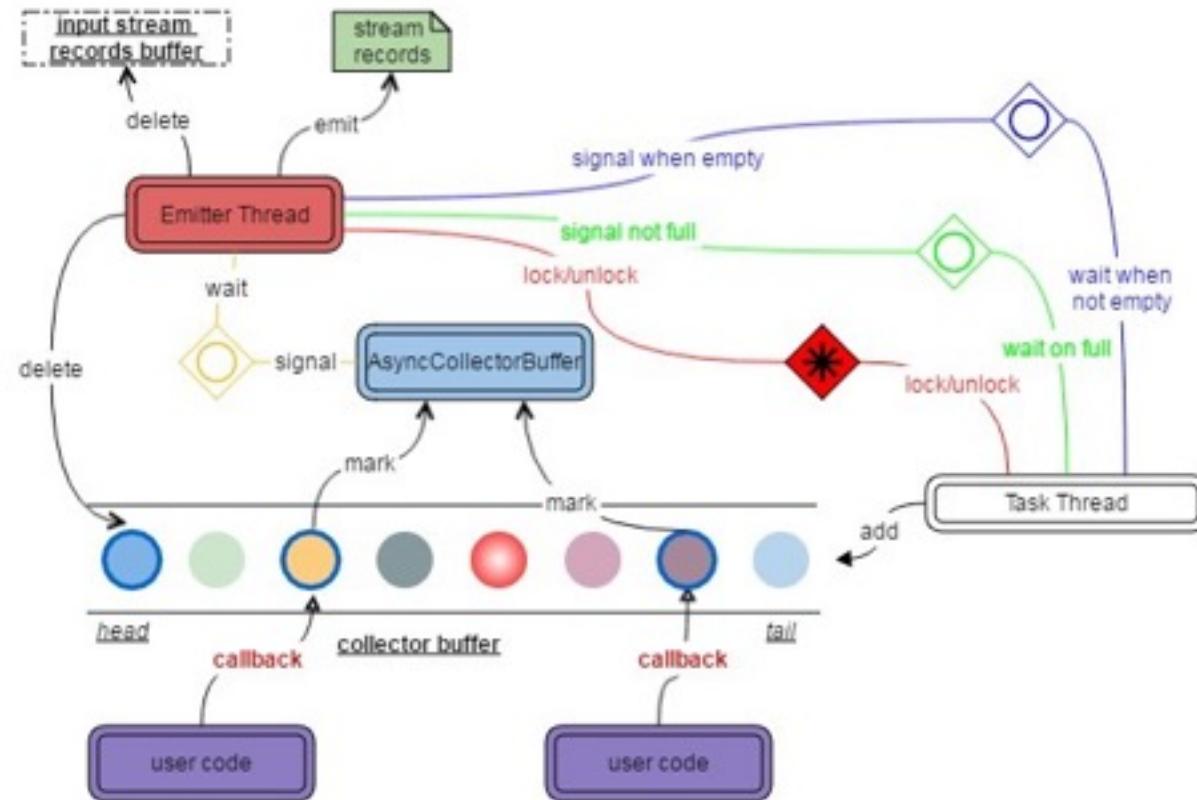


sendRequest(x)

receiveResponse(x)

wait

Asynchronous I/O Support



Asynchronous I/O Support

```
// create the original stream  
val stream: DataStream[String] = ...
```

```
// apply the async I/O transformation  
val resultStream: DataStream[(String, String)] =
```

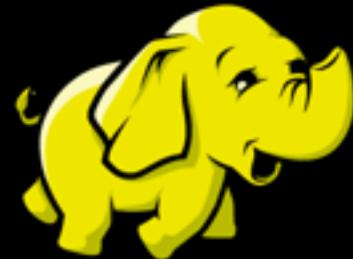
```
AsyncDataStream.unorderedWait(  
    input = stream,  
    asyncFunction = new AsyncDatabaseRequest(),  
    timeout = 1000,  
    timeUnit = TimeUnit.MILLISECONDS,  
    concurrentRequests = 100)
```

Asynchronous I/O Support

```
class AsyncDatabaseRequest extends AsyncFunction[String, (String, String)] {  
  
    override def asyncInvoke(str: String, asyncCollector: AsyncCollector[(String, String)]): Unit = {  
  
        // issue the asynchronous request, receive a future for the result  
        val resultFuture: Future[String] = client.query(str)  
  
        // set the callback to be executed once the request by the client is complete  
        // the callback simply forwards the result to the collector  
        resultFuture.onSuccess {  
            case result: String => asyncCollector.collect(Iterable((str, result)));  
        }  
    }  
}
```

Enhanced Security

- SSL
- Kerberos
 - Kafka
 - Zookeeper
 - Hadoop



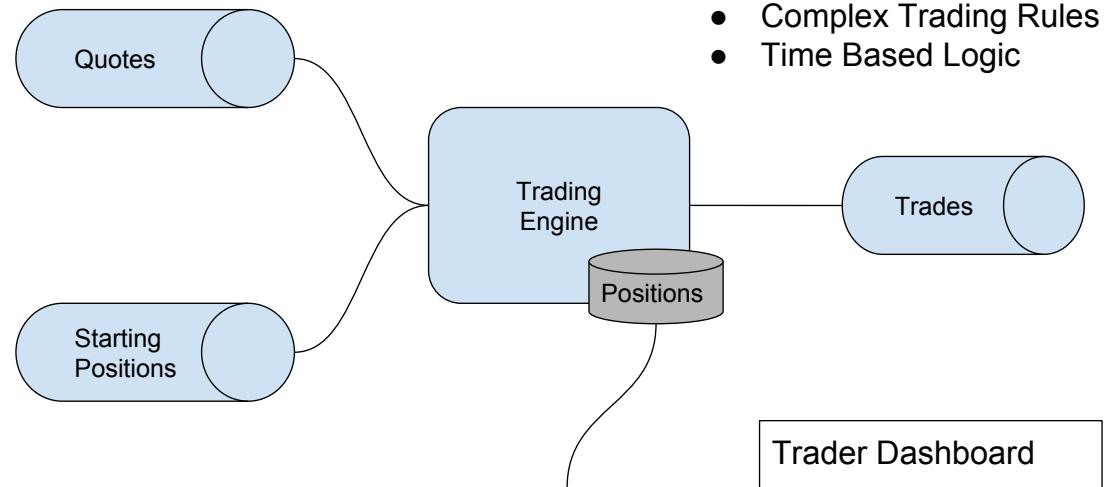
Advanced Event-Driven Applications

- ProcessFunction API
- Queryable State API
- Excellent support for advanced applications that are:
 - Flexible
 - Stateful
 - Event Driven
 - Time Driven

Example: FlinkTrade

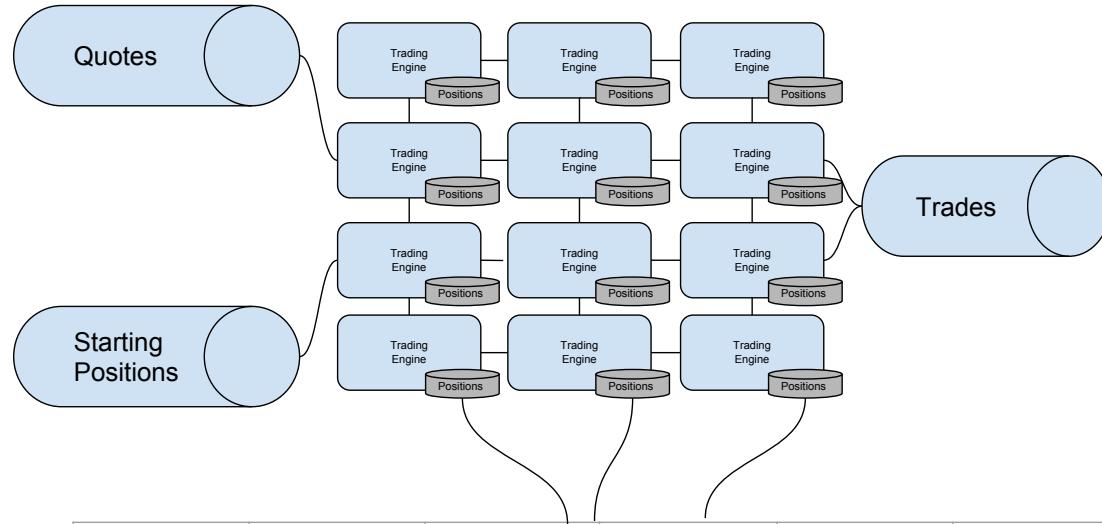
- Overall Requirements:
 - Consume “starting position” and “quote” streams
 - Process complex, time-oriented, trading rules
 - Trade out of positions to our advantage if possible
 - Provide a dashboard of currently held positions to traders and asset managers
- Complex Rules:
 - We only make trades where the Bid Price is above our current Ask Price
 - When a trade is made we increase our Ask Price – looking to optimize our profits
 - Positions have a set time-to-live until we try to trade out of them more aggressively by decreasing the Ask Price over time

Example: FlinkTrade



SYMBOL	SHARES	BUY PRICE	ASK PRICE	LAST TRADE PRICE	Profit
AAPL	10,000	140.40	140.50	140.40	\$10,921.00
GOOG	20,000	846.81	846.91	846.81	\$12,021.00
TWTR	8,000	15.12	15.22	15.12	\$4,032.00

Example: FlinkTrade



SYMBOL	SHARES	BUY PRICE	ASK PRICE	LAST TRADE PRICE	Profit
AAPL	10,000	140.40	140.50	140.40	\$10,921.00
GOOG	20,000	846.81	846.91	846.81	\$12,021.00
TWTR	8,000	15.12	15.22	15.12	\$4,032.00

Example: FlinkTrade

Let's look at
the code



We are hiring!

data-artisans.com/careers

@jamiegrier

@ApacheFlink

@dataArtisans

AthenaX: Streaming Processing Platform @Uber

Bill Liu, Haohui Mai

APRIL 11, 2017

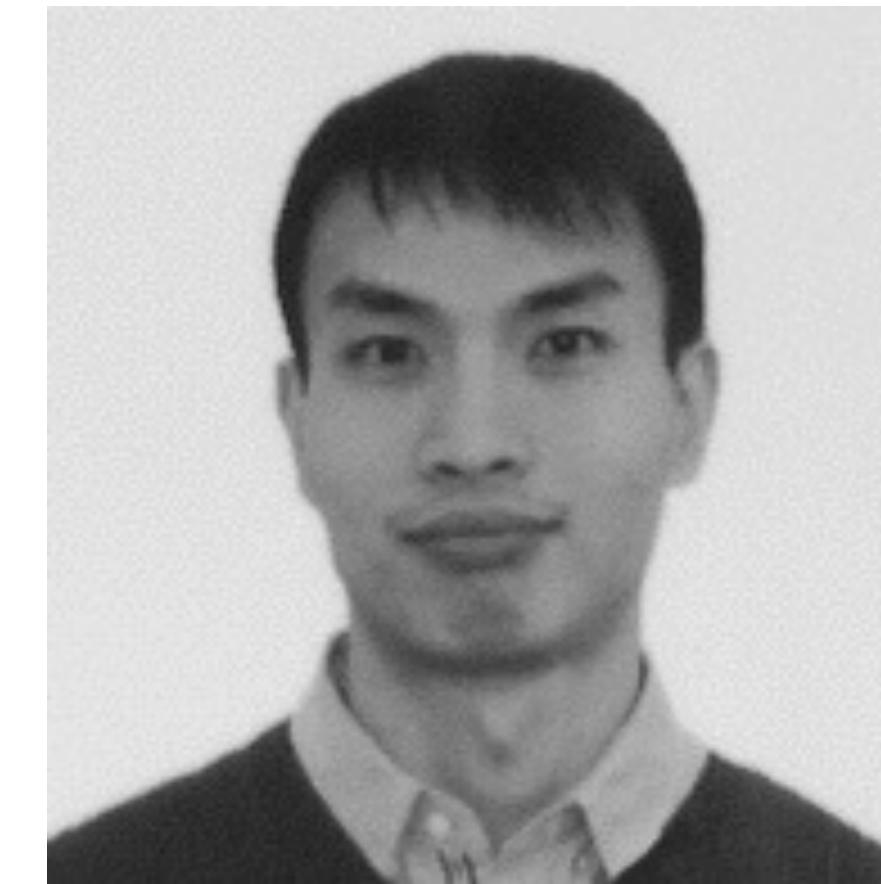


Speakers



Bill Liu

- Senior Software Engineer @ Uber

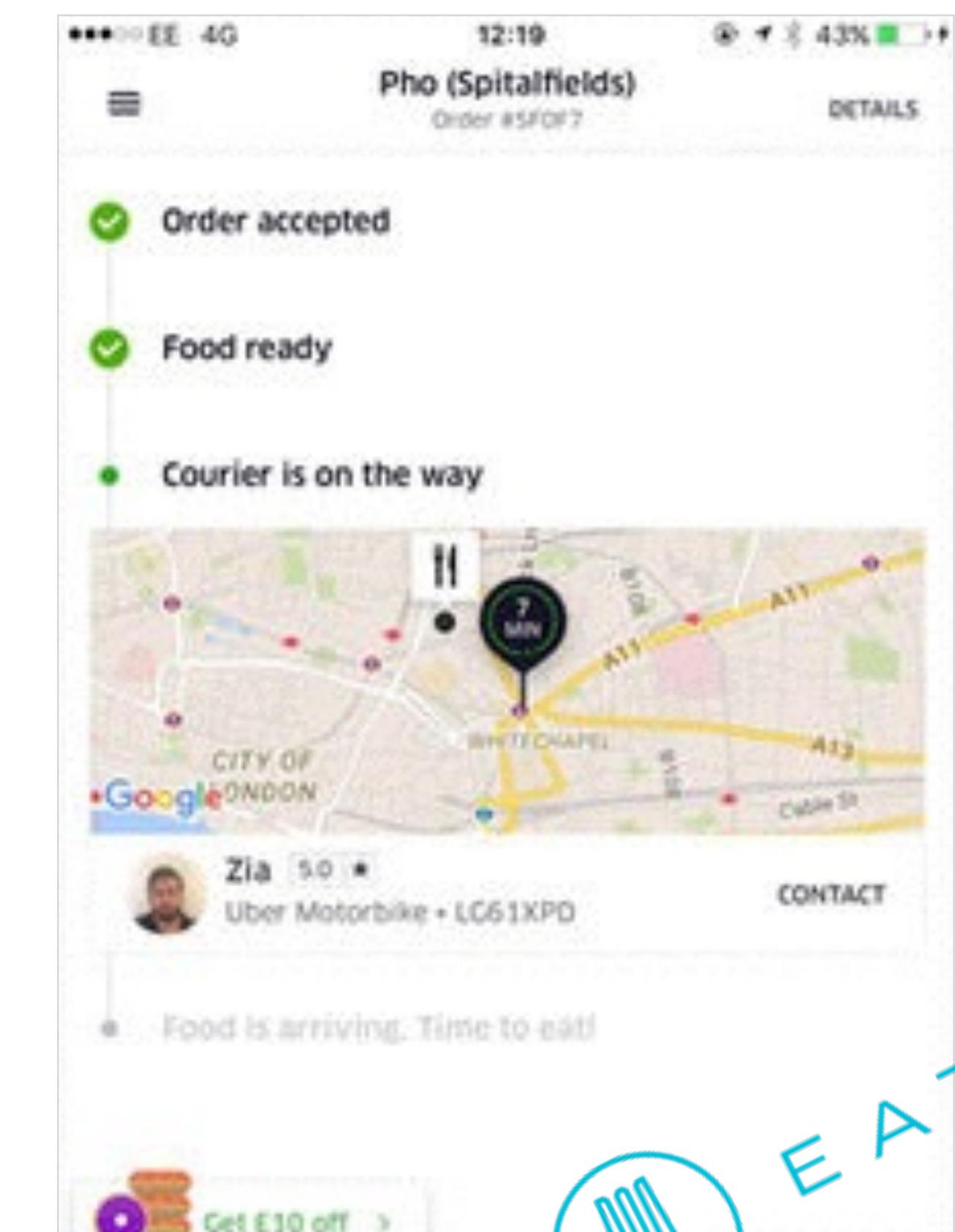


Haohui Mai, @wheat9

- Senior Software Engineer @ Uber
- PMC, Apache Hadoop & Storm

Uber business is real-time

- Uber: Transport A → B on demand reliably
- Dynamic marketplace
- Example: UberEATS



Challenges

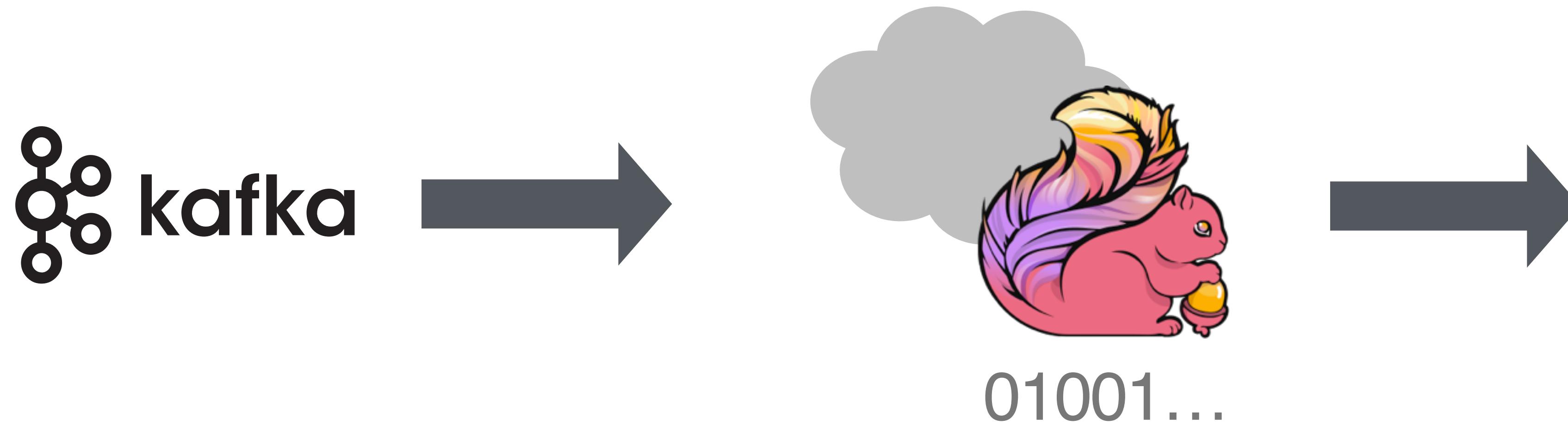
Infra.: Reliability & scalability

- 99.99% SLA on latency
- At-least-once processing
- Billions of messages
- Multiple PB / day

Solutions: Productivity

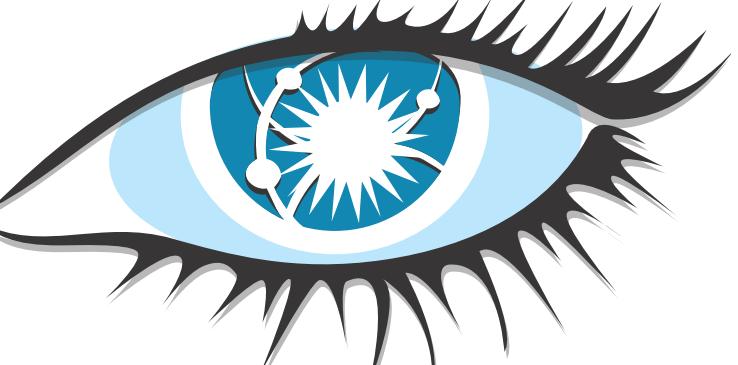
- Audiences: majority of employees use SQL actively
- Abstractions: Flink / DSL?
- Integrations: data management, monitoring, reporting, etc.

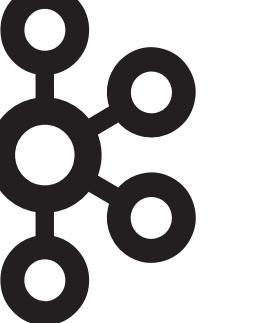
Building streaming applications

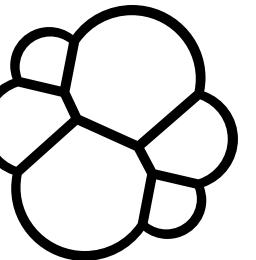


- Framework-specific
- Ad-hoc management over the life-cycles

 memsql

 cassandra

 kafka

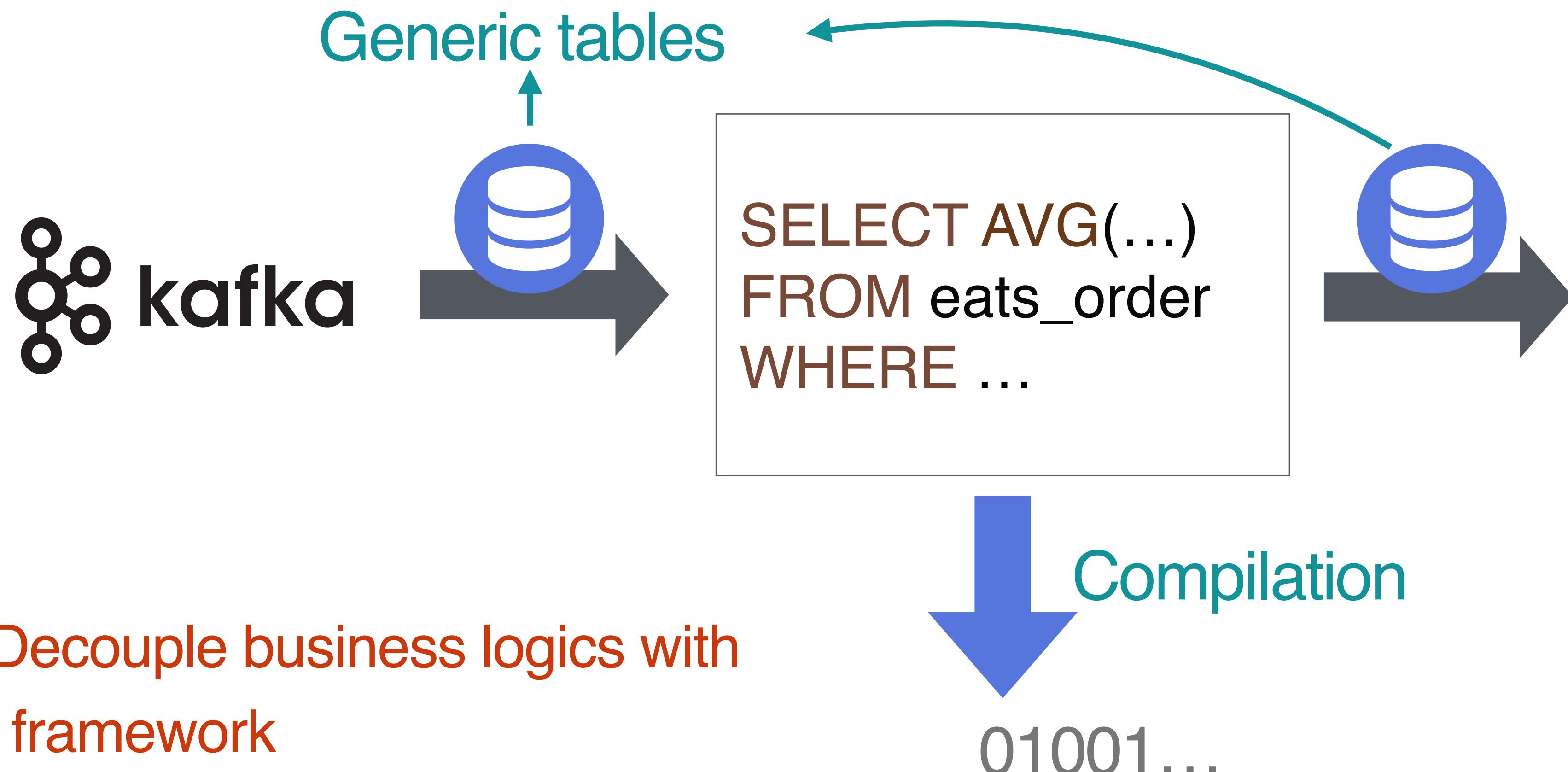
 elastic

Thrift

 MySQL®

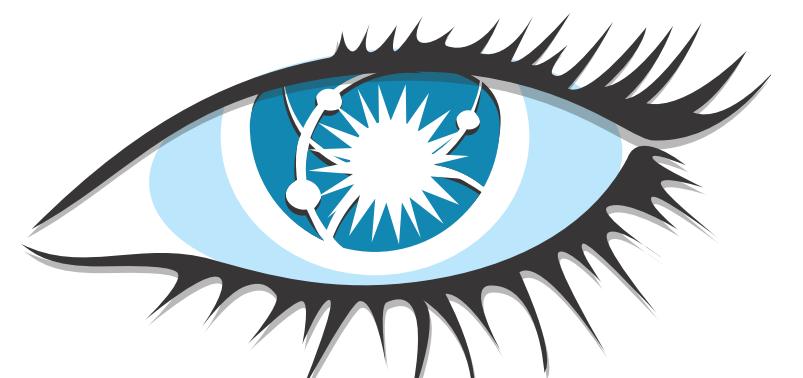
The AthenaX approach

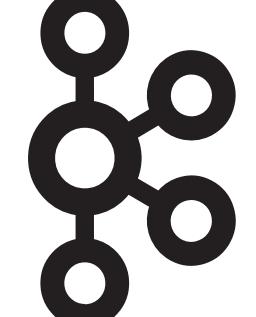
Write SQLs to build streaming applications

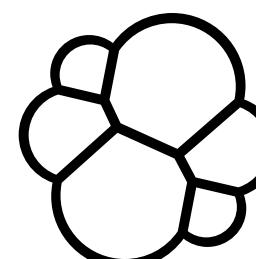


- Decouple business logics with framework
- Unified integration & management

memsql

 cassandra

 kafka

 elastic

Thrift

 MySQL®

AthenaX: Streaming processing platform @ Uber

- Write SQLs to build streaming applications
 - **Insight: generic table**
- Reliable, scalable processing based on Apache Flink
- Develop & deploy streaming applications in production in hours instead of weeks

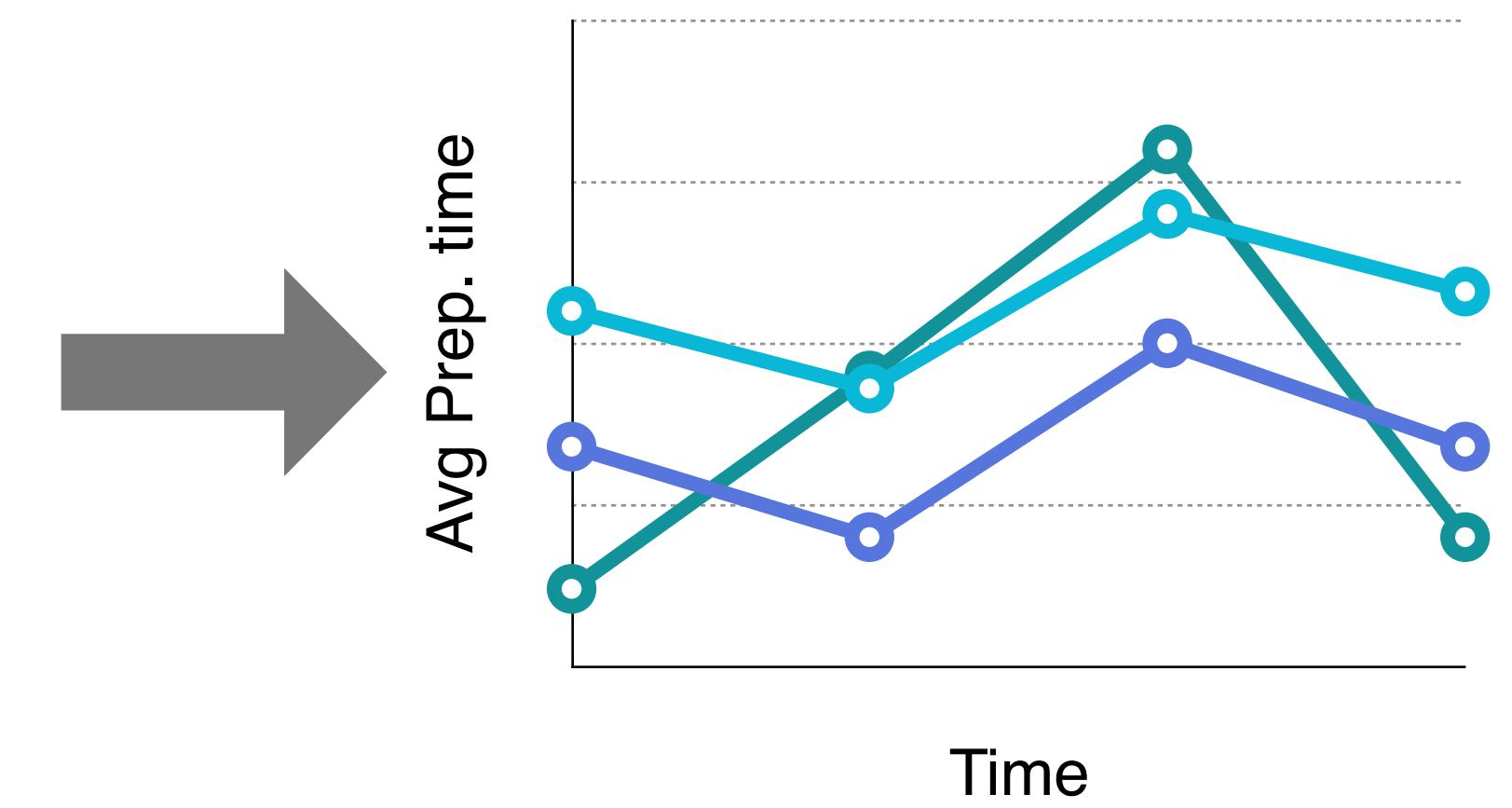
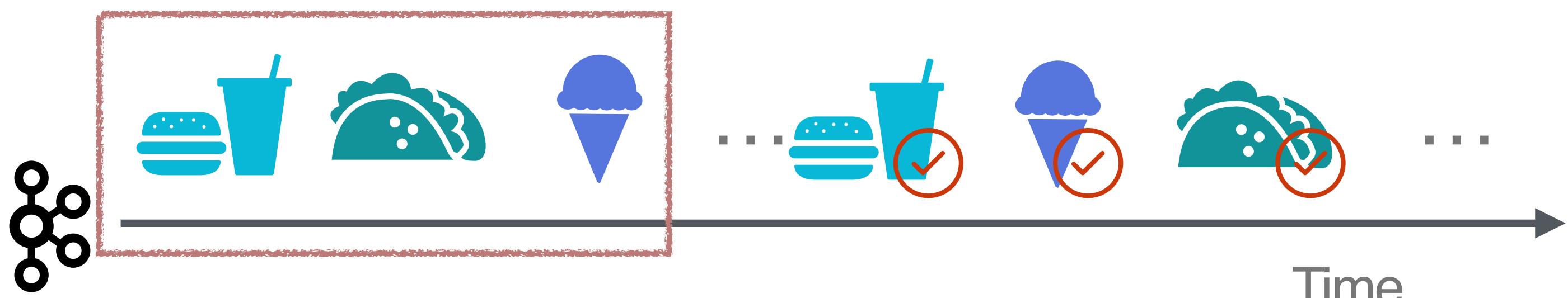


Agenda

- Motivating example
- Case study: ETD in UberEATS
- Implementation
- Current status
- Conclusion

Example

Real-time dashboard for restaurants



```
SELECT meal_id, AVG(meal_prep_time)  
FROM eats_order
```

```
GROUP BY meal_id, HOP(proctime(),  
INTERVAL '1' MINUTE,  
INTERVAL '15' MINUTE)
```

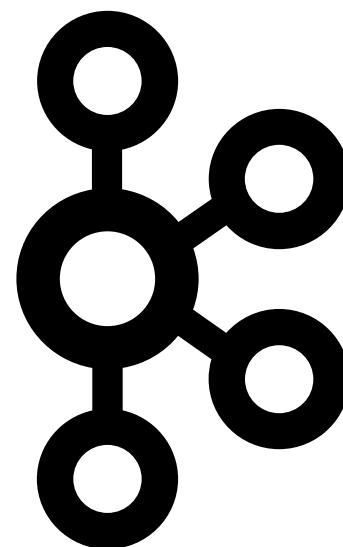
Example (cont.)

Building streaming processing applications with SQL

```
SELECT AVG(meal_prep_time) FROM  
eats_order  
  
GROUP BY meal_id, HOP(proctime(),  
INTERVAL '1' MINUTE,  
INTERVAL '15' MINUTE)
```

Example (cont.)

Building streaming processing applications with SQL



```
SELECT * FROM (
    SELECT EXPECTED_TIME(meal_id)
AS e, meal_id,
    AVG(meal_prep_time) AS t
    FROM eats_order
    GROUP BY meal_id, HOP(proctime(),
    INTERVAL '1' MINUTE,
    INTERVAL '15' MINUTE)
```



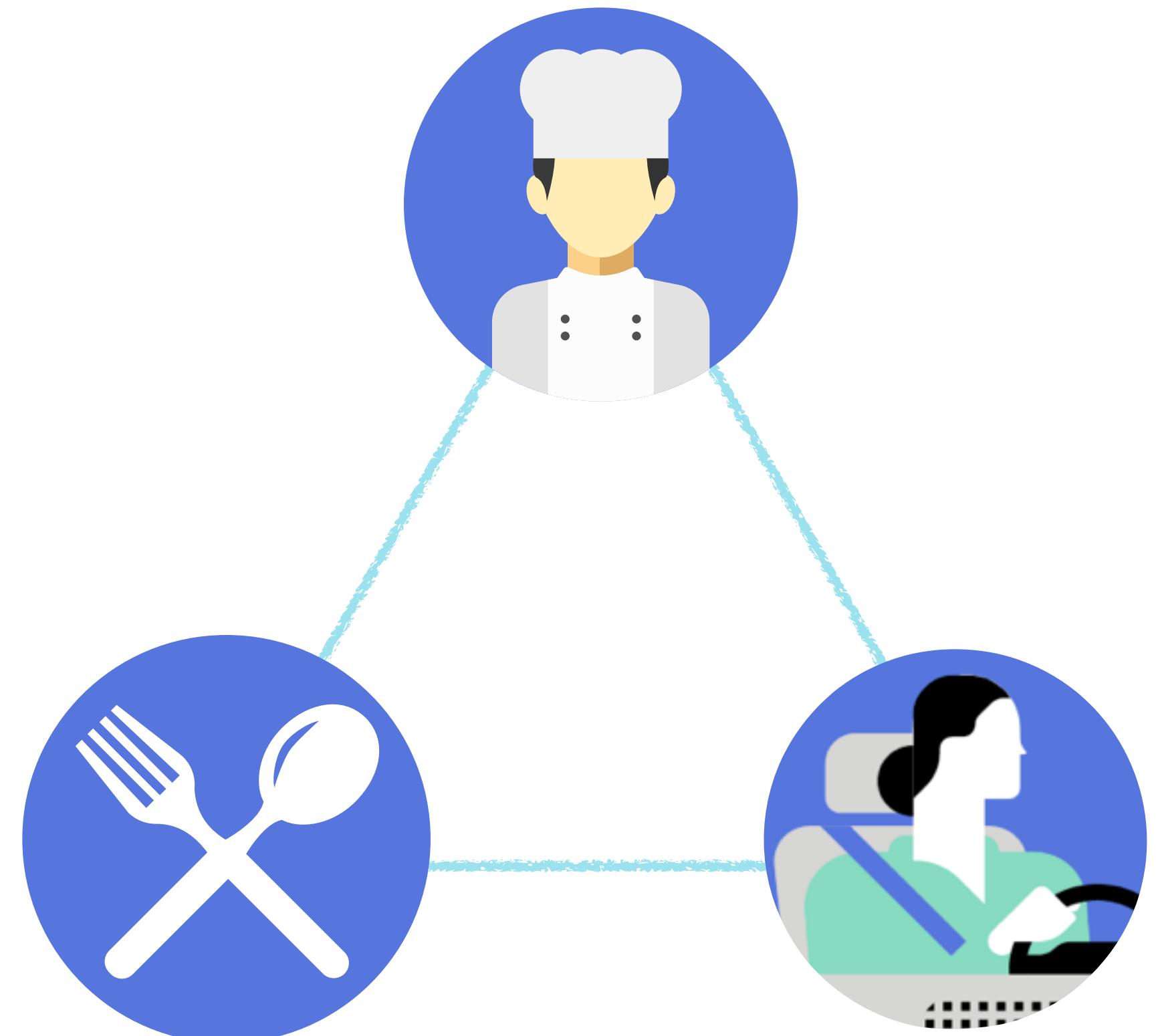
Tables are more generic than analytical stores

Agenda

- Motivating example
- Case study: ETD in UberEATS
- Implementation
- Current status
- Conclusion

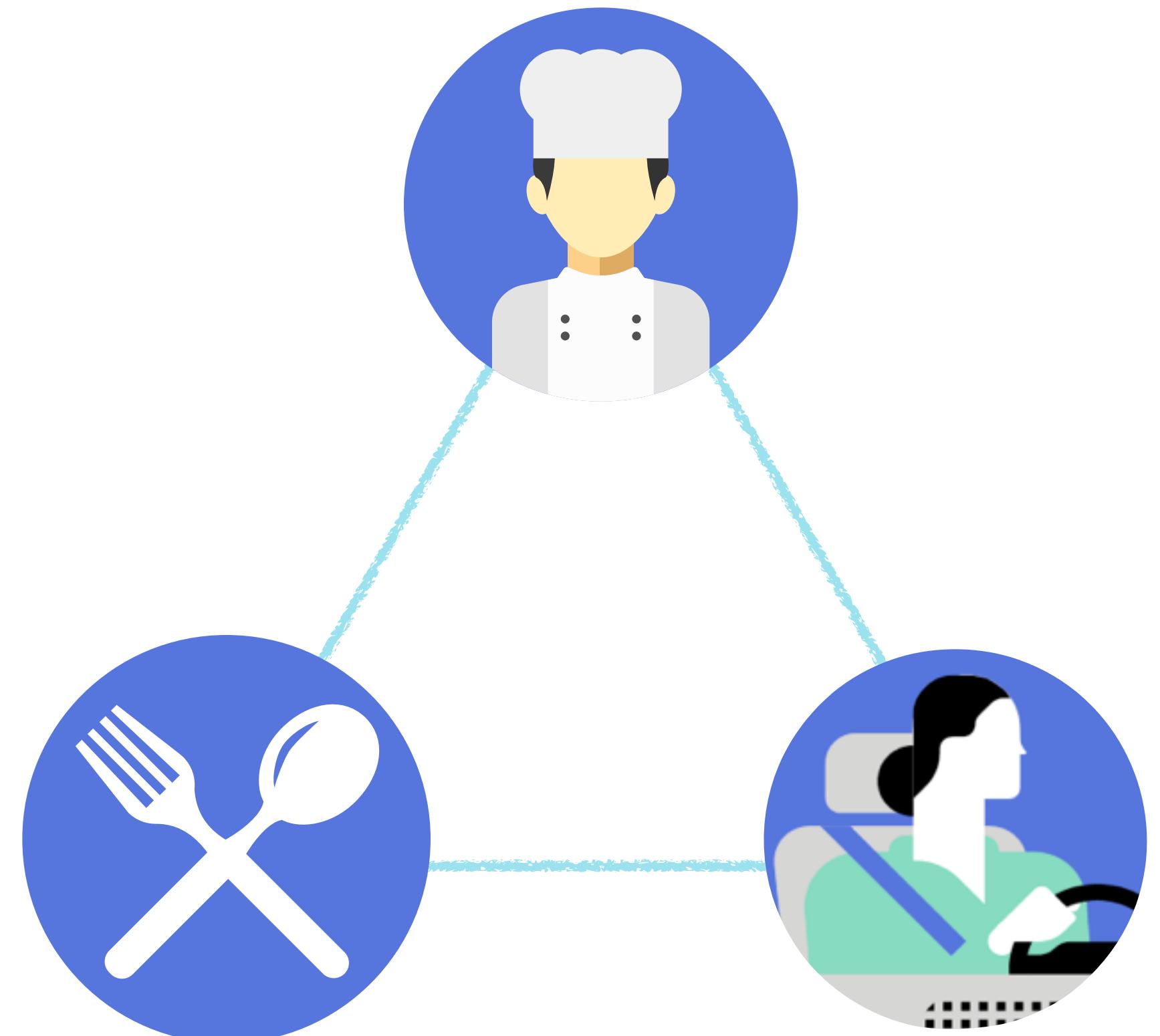
The case of UberEATS

- Three-way marketplace
- Real-time metrics
 - Estimated Time to Delivery (ETD)
 - Transactions
 - Demand forecasts



The case of UberEATS

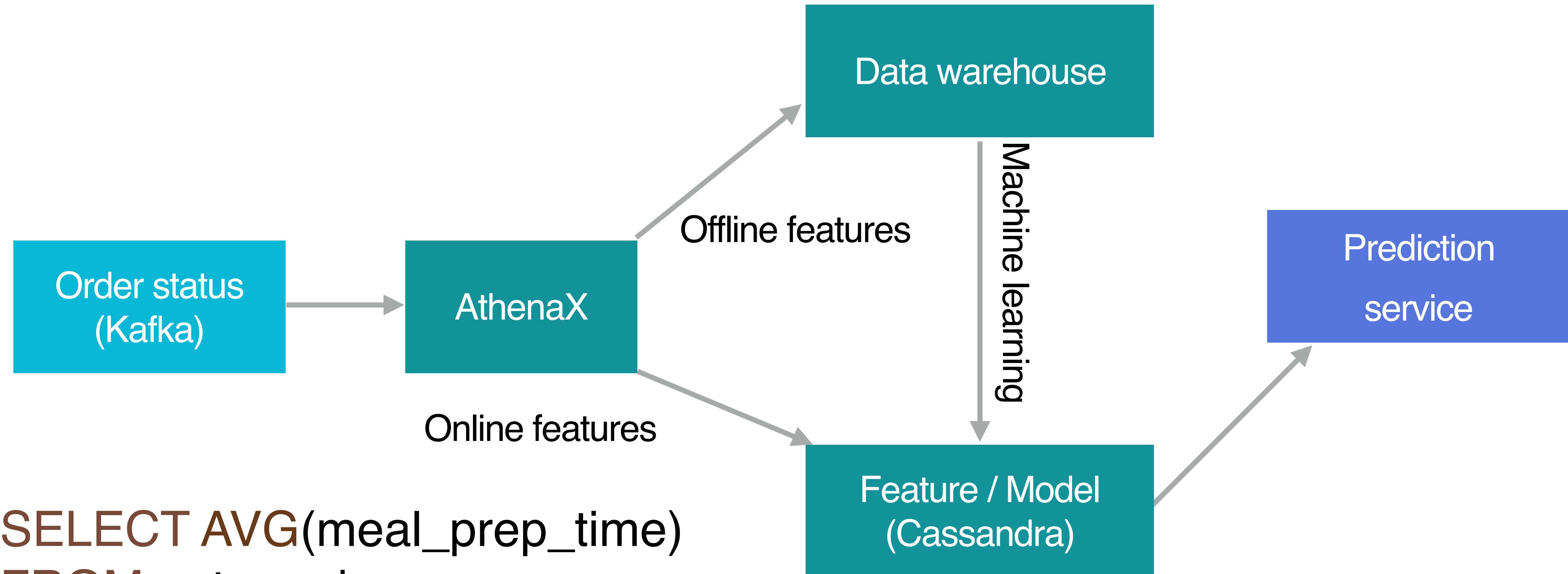
- Three-way marketplace
- Real-time metrics
 - Estimated Time to Delivery (ETD)
 - Transactions
 - Demand forecasts



Predicting the ETD

- Key metric: time to prepare a meal(t_{prep})
- Learn a function f : (order status) $\rightarrow t_{\text{prep}}$ *periodically*
- Predict the ETD for current orders using f
- AthenaX extracts features for both learnings and predictions

Architecture of the ETD service

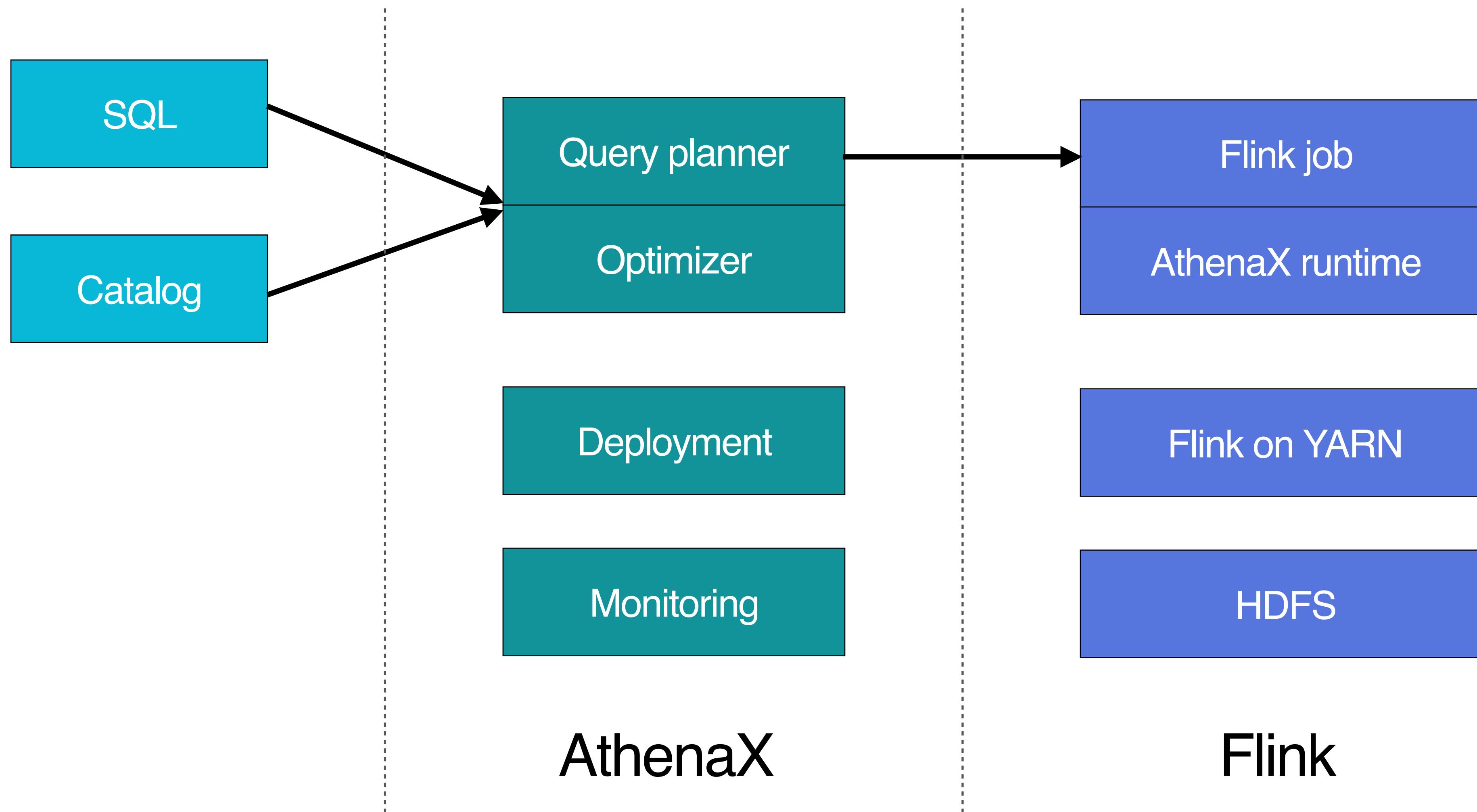


```
SELECT AVG(meal_prep_time)  
FROM eats_order  
GROUP BY meal_id,  
HOP(proctime(),  
INTERVAL '1' MINUTE,
```

Agenda

- Motivating example
- Case study: ETD in UberEATS
- **Implementation**
- Current status
- Conclusion

Architecture



Executing AthenaX applications

Compile SQLs to Flink applications

- Compilation + Code generation
 - Flink SQL APIs: SQL → Logical plans → Flink applications
 - Leverage the Volcano optimizer in Apache Calcite
- Challenges: exposing streaming semantics

Query planner

Optimizer

Deployment

Monitoring

AthenaX as a self-serving platform

Self-serving production support end-to-end

- Metadata / catalog management
- Job management
- Monitoring
- Resource management and elastic scaling
- Failure recovery

Query planner

Optimizer

Deployment

Monitoring

Agenda

- Motivating example
- Case study: ETD in UberEATS
- Implementation
- **Current status**
- Conclusion

Current status

- Pilot jobs in production
 - In the process of full-scale roll outs
- Based on Apache Flink 1.3-SNAPSHOT
 - Projection, filtering, group windows, UDF
 - Streaming joins not yet supported

Embrace the community

Contributions to the upstream

- Group window support for streaming SQL
 - CALCITE-1603, CALCITE-1615
 - FLINK-5624, FLINK-5710, FLINK-6011, FLINK-6012
- Stability fixes
 - FLINK-3679, FLINK-5631
- Table abstractions for Cassandra / JDBC (WIP)
- Available in the upcoming 1.3 release



Agenda

- Motivating example
- Case study: ETD in UberEATS
- Implementation
- Current status
- Conclusion

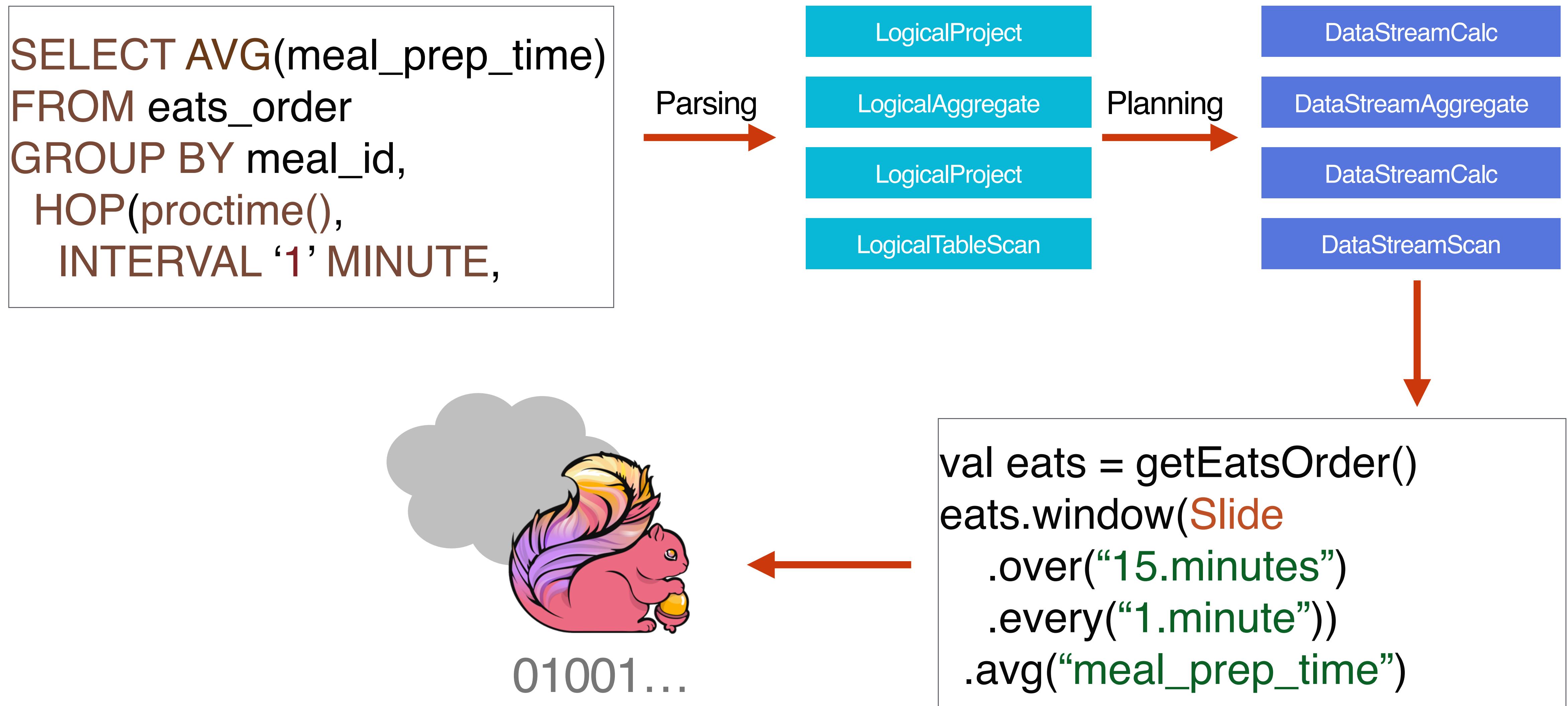
Conclusion

- AthenaX: write SQLs to build streaming applications
 - Treat table as a generic concept
 - Productivity: development → production in hours
- The AthenaX approach
 - SQL on streams as a platform
 - Self-serving production support end-to-end

Thank you

UBER

Compiling SQL



Lazy deserialization

Example of SQL optimization



```
SELECT  
AVG(meal_prep_time)  
FROM eats_order
```

0 record prepTime ADD DESCRIPTION
long min ADD DESCRIPTION
long max ADD DESCRIPTION

Blink's Improvements to Flink SQL &Table API

Shaoxuan Wang,

Xiaowei Jiang

{xiaowei.jxw,shaoxuan.wang}
@alibaba-inc.com

FlinkForward 2017.4



About Us

■ **Xiaowei Jiang**

- 2014-now Alibaba
- 2010-2014 Facebook
- 2002-2010 Microsoft
- 2000-2002 Stratify

■ **Shaoxuan Wang**

- 2015-now Alibaba
- 2014-2015 Facebook
- 2010-2014 Broadcom



Agenda

- **Background**
- **Why SQL & Table API**
- **Blink SQL & Table API (Selected Topics)**



Background About Alibaba

■ Alibaba Group

- Operates the world's largest e-commerce platform
- Recorded GMV of \$394 Billion in year 2015, \$17.8 billion worth of GMV on Nov 11, 2016

■ Alibaba Search

- Personalized search and recommendation
- Major driver for user's traffic

A search bar with placeholder text "I'm shopping for..." and a dropdown menu for "All Categories".A search bar with placeholder text "Products" and a dropdown menu. To its right is another search bar with placeholder text "What are you looking for..." and a large orange "SEARCH" button.A search bar with a magnifying glass icon and a red "搜索" (Search) button.A search bar with placeholder text "惠及颠覆你的生活" and a red "搜索" (Search) button. Below it is a horizontal menu with items: 连衣裙夏 | 智能手机 | t恤男 | 亲子装 | 凉鞋 | 夏凉被 | 睡衣女夏 | 防晒衣 | 电风扇.

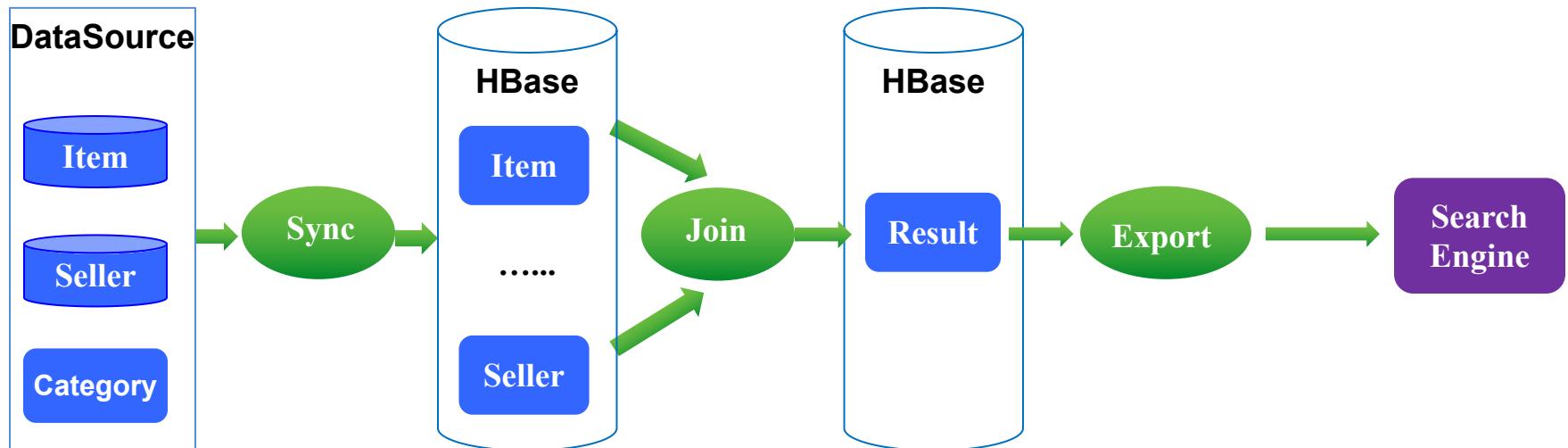


What is Blink?

- **Blink – A project to make Flink work well for large scale production at Alibaba**
 - Run on Thousands of Nodes In Production
 - Support Key Production Scenarios, such as Search and Recommendation
 - Compatible with Flink API and Ecosystem



Documents Building for Search





Why Flink SQL & Table API

■ Unify batch and streaming

- Flink currently offers DataSet API for batch and DataStream API for streaming
- We want a single API that can run in both batch and streaming mode

■ Simplify user code

- Users only describe the semantics of their data processing
- Leave hard optimization problems to the system
- SQL is proven to be good at describing data processing
- Table API makes multi-statement data processing easy to write
- Table API also makes it possible/easy to extend standard SQL when necessary



Stream-Table Duality

Stream

word	count
Hello	1
World	1
Hello	2
Bark	1
Hello	3

Apply



Dynamic Table

word	count
Hello	3
World	1
Bark	1

Changelog





Dynamic Tables

■ Apply Changelog Stream to Dynamic Table

- **Append Mode:** each stream record is an insert modification to the dynamic table. Hence, all records of a stream are appended to the dynamic table
- **Update Mode:** a stream record can represent an insert, update, or delete modification on the dynamic table (append mode is in fact a special case of update mode)

■ Derive Changelog Stream from Dynamic Table

- **REDO Mode:** where the stream records the new value of a modified element to redo lost changes of completed transactions
- **REDO+UNDO Mode:** where the stream records the old and the new value of a changed element to undo incomplete transactions and redo lost changes of completed transactions



Stream SQL



**Dynamic Tables generalize the concept of Static Tables
SQL serves as the unified way to describe data processing in
both batch and streaming**

There is no such thing as Stream SQL



Blink SQL & Table API

- Stream-Stream Inner Join
- User Defined Function (UDF)
- User Defined Table Function (UDTF)
- User Defined Aggregate Function (UDAGG)
- Retract (stream only)
- Over Aggregates



A Simple Query: Select and Where

id	name	price	sales	stock
1	Latte	6	1	1000
8	Mocha	8	1	800
4	Breve	5	1	200
7	Tea	4	1	2000
1	Latte	6	2	998

```
SELECT id, name, price, sales, stock  
FROM myTable WHERE name = 'Latte'
```



id	name	price	sales	stock
1	Latte	6	1	1000
1	Latte	6	2	998



Stream-Stream Inner Join

id1	name	stock
1	Latte	1000
8	Mocha	800
4	Breve	200
3	Water	5000
7	Tea	2000

id2	price	sales
1	6	1
8	8	1
9	3	1
4	5	1
7	4	1

```
SELECT id1 AS id, name, price, sales, stock  
FROM table1 INNER JOIN table2 ON id1 = id2
```



id	name	price	sales	stock
1	Latte	6	1	1000
8	Mocha	8	1	800
4	Breve	5	1	200
7	Tea	4	1	2000

*This is proposed and discussed in
FLINK-5878*



Blink SQL & Table API

- Stream-Stream Inner Join
- User Defined Function (UDF)
- User Defined Table Function (UDTF)
- User Defined Aggregate Function (UDAGG)
- Retract (stream only)
- Over Aggregates



User Defined Function (UDF)

Create and use a UDF is very simple and easy:

```
object AddFunc extends ScalarFunction {
    def eval(a: Long, b: Long): Long = a + b
    @varargs
    def eval(a: Int*): Int = a.sum
}

tEnv.registerTable("MyTable", table)
tEnv.registerFunction("addFunc", AddFunc)
val sqlQuery =
    "SELECT addFunc(long1,long2), addFunc(int1,int2,int3) FROM MyTable"
```

We recently have enhanced UDF/UDTF to let them support variable types and variable arguments (FLINK-5826)



User Defined Table Function (UDTF)

line

Tom#23 Jark#17 David#50

Scalar → Table (multi rows and columns)



	name	age
	Tom	23
	Jack	17
	David	50

```
SELECT name, age  
FROM myTable,  
LATERAL TABLE(splitFunc(line))  
AS T(name, age)
```

```
case class User(name: String, age: Int)  
class SplitFunc extends TableFunction[User] {  
    def eval(str: String): Unit = {  
        str.split(" ").foreach{ e =>  
            val subSplits = e.split("#")  
            collect(User(subSplits(0), subSplits(1).toInt))  
        }  
    }  
}
```

We have shipped UDTF in flink 1.2 (FLINK-4469).



User Defined Aggregate Function (UDAGG) - Motivation

Flink has built-in aggregates (count, sum, avg, min, max) for SQL and table API

```
SELECT name, SUM(price), COUNT(sales),
       MAX(price), MIN(price), AVG(price)
FROM myTable
GROUP BY name
```

What if user wants an aggregate that is not covered by built-in aggregates, say a weighted average aggregate?

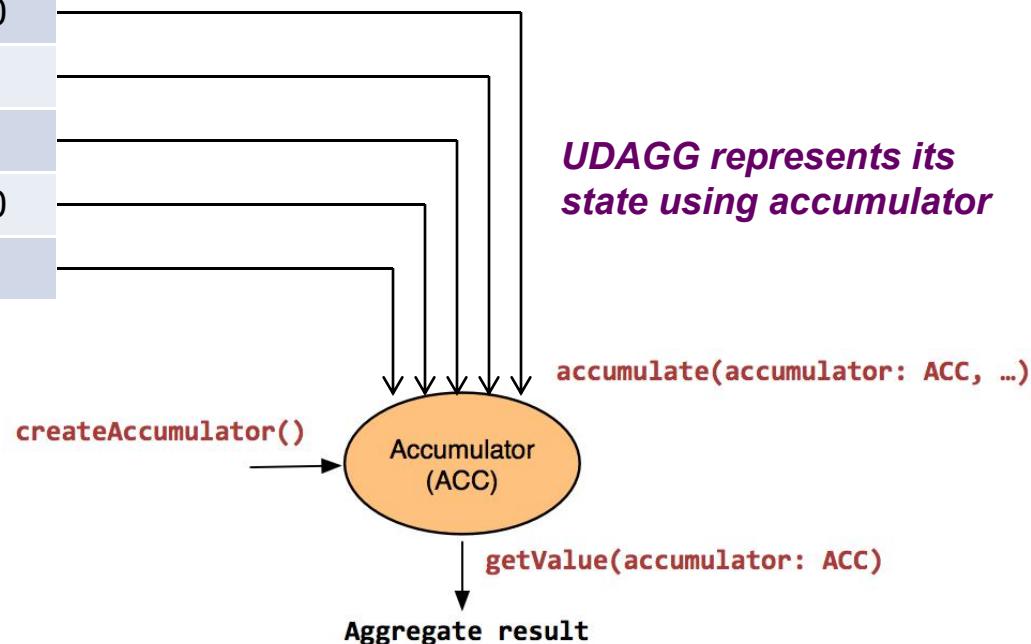
We need an aggregate interface to support user defined aggregate function.



UDAGG – Accumulator (ACC)

id	name	price	sales	stock
1	Latte	6	1	1000
8	Mocha	8	1	800
4	Breve	5	1	200
7	Tea	4	1	2000
1	Latte	6	2	998

```
SELECT name, SUM(price), COUNT(sales),
       MAX(price), MIN(price), AVG(price)
  FROM myTable
 GROUP BY name
```





UDAGG – Interface

UDAGG Interface

```
Abstract class AggregateFunction[T, ACC] extends UserDefinedFunction {  
    def createAccumulator(): ACC  
    def getValue(accumulator: ACC): T  
}  
/* The implementations of accumulate must be declared publicly,  
not static and named exactly as "accumulate". accumulate method  
can be overloaded */  
def Accumulate(accumulator: ACC, [user defined inputs]): Unit
```

SQL Query

```
SELECT type, weightAvgFun(weight, cnt)  
FROM myTable  
GROUP BY type
```

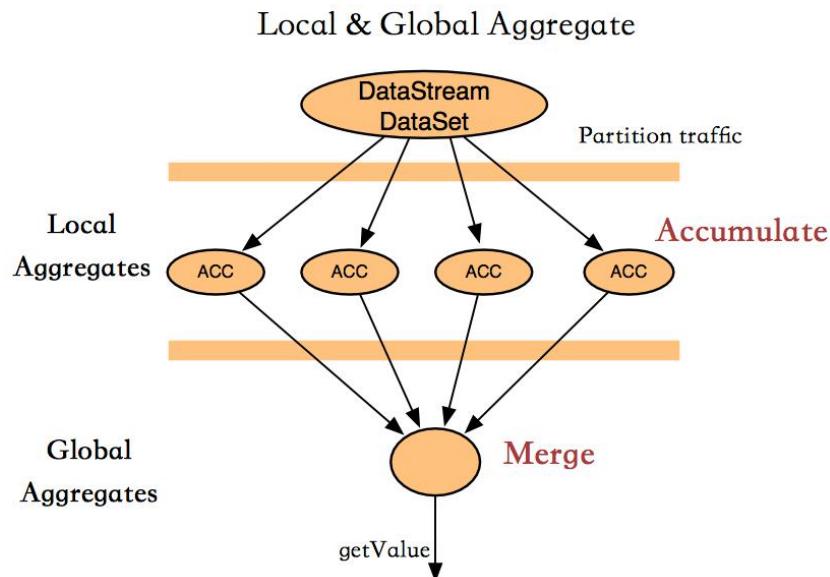
UDAGG example: a weighted average

```
public static class WeightedAvgAccum {  
    public long sum = 0; public int count = 0; }  
public static class WeightedAvg  
extends AggregateFunction<Long, WeightedAvgAccum> {  
    @Override  
    public WeightedAvgAccum createAccumulator() {  
        return new WeightedAvgAccum(); }  
    @Override  
    public Long getValue(WeightedAvgAccum accumulator) {  
        if (accumulator.count == 0) return null;  
        else return accumulator.sum/accumulator.count; }  
    public void accumulate(  
        WeightedAvgAccum accumulator, long iValue, int iWeight) {  
        accumulator.sum += iValue * iWeight;  
        accumulator.count += iWeight; }  
}
```



UDAGG – Merge

How to count the total visits on TaoBao web pages in real time?



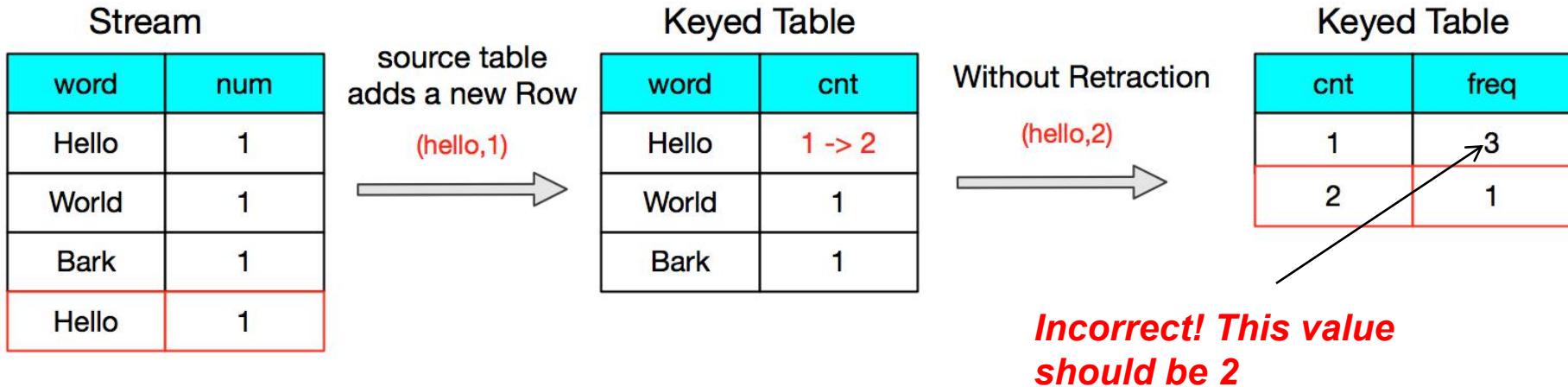
```
Abstract class AggregateFunction[T, ACC] extends UserDefinedFunction {  
    def createAccumulator(): ACC  
    def getValue(accumulator: ACC): T  
}  
/* The implementations of following methods: accumulate(MUST have), merge(OPTIONAL) must be declared publicly, not static and named exactly as "accumulate", "merge" */  
def Accumulate(accumulator: ACC, [user defined inputs]): Unit  
def merge(it: java.util.iterator[ACC]): ACC
```

Motivated by local & global aggregate (and session window merge etc.), we need a merge method which can merge the partial aggregated accumulator into one single accumulator



UDAGG - Retract - Motivations

```
SELECT  
cnt,  
COUNT(word) AS freq  
FROM (  
    SELECT word, COUNT(num) AS cnt  
    FROM Table GROUP BY word  
) GROUP BY cnt
```





UDAGG - Retract – Motivations

Stream

word	num
Hello	1
World	1
Bark	1
Hello	1

source table
adds a new Row
(hello,1)

Keyed Table

word	cnt
Hello	1 > 2
World	1
Bark	1

Without Retraction

(Hello,2,accumulate)



Incorrect! this value
should be 2

Keyed Table

cnt	freq
1	3
2	1

Keyed Table

cnt	freq
1	2
2	1

With Retraction

(Hello,1,retract)

(Hello,2,accumulate)



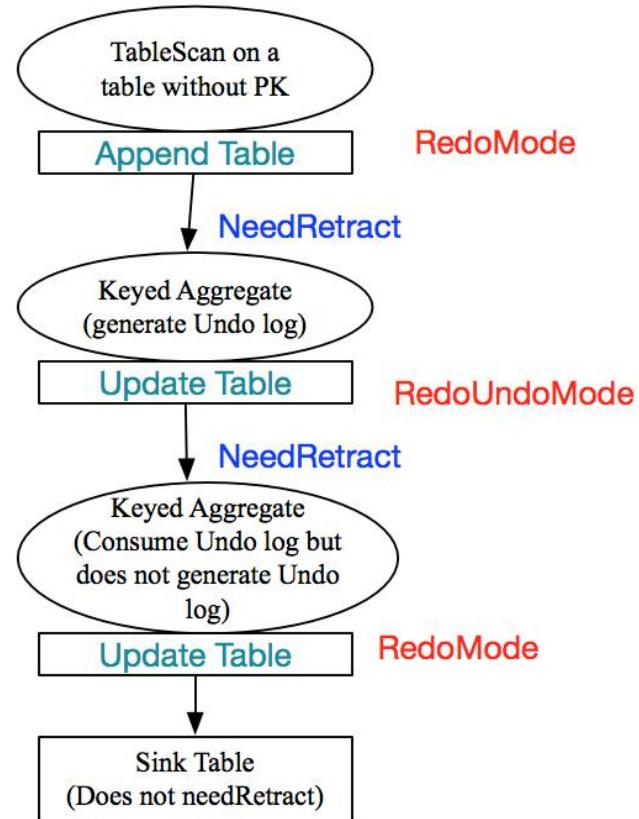
We need a retract method in UDAGG, which can retract the input values from the accumulator



Retract – Solution

- Retraction is introduced to handle updates
- We use query optimizer to decide where the retraction is needed.

The design doc and the progress of retract implementation are tracked in FLINK-6047. A FLIP for retract is on the way. We aim to release it in flink-1.3





UDAGG – Summary

```
Abstract class AggregateFunction[T, ACC] extends UserDefinedFunction {  
    def createAccumulator(): ACC  
    def getValue(accumulator: ACC): T  
}  
/* The implementations of following methods: accumulate(MUST have),  
merge(OPTIONAL), and retract(OPTIONAL) must be declared publicly, not  
static and named exactly as "accumulate", "merge", and "retract" */  
def accumulate(accumulator: ACC, [user defined inputs]): Unit  
def merge(it: java.util.iterator[ACC]): ACC  
def retract(accumulator: ACC, [user defined inputs]): Unit
```

Master JIRA for UDAGG is FLINK-5564. We plan to ship it in release 1.3.



Blink SQL & Table API

- Stream-Stream Inner Join
- User Defined Function (UDF)
- User Defined Table Function (UDTF)
- User Defined Aggregate Function (UDAGG)
- Retract (stream only)
- Over Aggregates



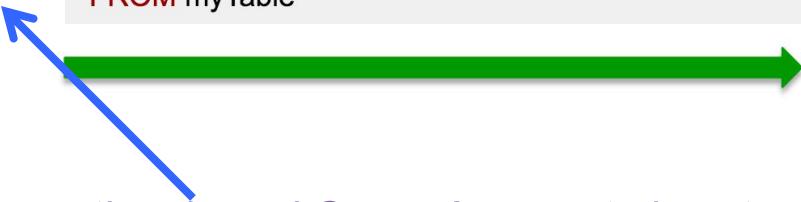
Over Aggregates

Calculate moving average (in the past 5 seconds), and emit the result for each record

time	itemID	price
1000	101	1
3000	201	2
4000	301	3
5000	101	1
5000	401	4
7000	301	3
8000	501	5
10000	101	1

SELECT

```
time, itemID, MovingAverage(price) OVER
(
    ORDER BY RowTime()
    RANGE
        BETWEEN INTERVAL '5' SECOND PRECEDING
        AND CURRENT ROW
) AS avgPrice
FROM myTable
```



time based Group Aggregate is not able to differentiate two records with the same row time.

time	itemID	avgPrice
1000	101	1
3000	201	1.5
4000	301	2
5000	101	2.2
5000	401	2.2
7000	301	2.6
8000	401	3
10000	101	2.8



Group/Over Aggregates

■ **Grouping methods:** Groupby / Over

■ **Window types:**

- Time/Count + TUMBLE/SESSION/SLIDE window;
- OVER Range/Rows window

■ **Time types:** Event time; Process time (only for stream)

We have been working closely with team dataArtisans, from the design to the implementation on FLIP11 (FLINK-4557). Upon now, except the unbounded group aggregate, all other group/over aggregates are fully supported via SQL query. We are working on the support for table API.



Current Status of Flink SQL & Table API

- Flink blog: “Continuous Queries on Dynamic Tables” (*posted at <https://flink.apache.org/news/2017/04/04/dynamic-tables.html>*)
- UDF (*several improvements will be released in 1.3*)
- UDTF (*FLINK-4469, released in 1.2*)
- UDAGG (*FLINK-5564, target for release 1.3*)
- Group/Over Window Aggregate (*FLINK-4557, target for release 1.3*)
- Retract (*FLINK-6047, target for release 1.3*)
- Unbounded Stream Group Aggregate (*FLINK-6216, bundled with retract design*)
- Stream-Stream Inner Join (*FLINK-5878, TBD*)

We will keep merging Blink SQL & Table API back to Flink



Q & A

Thank You!

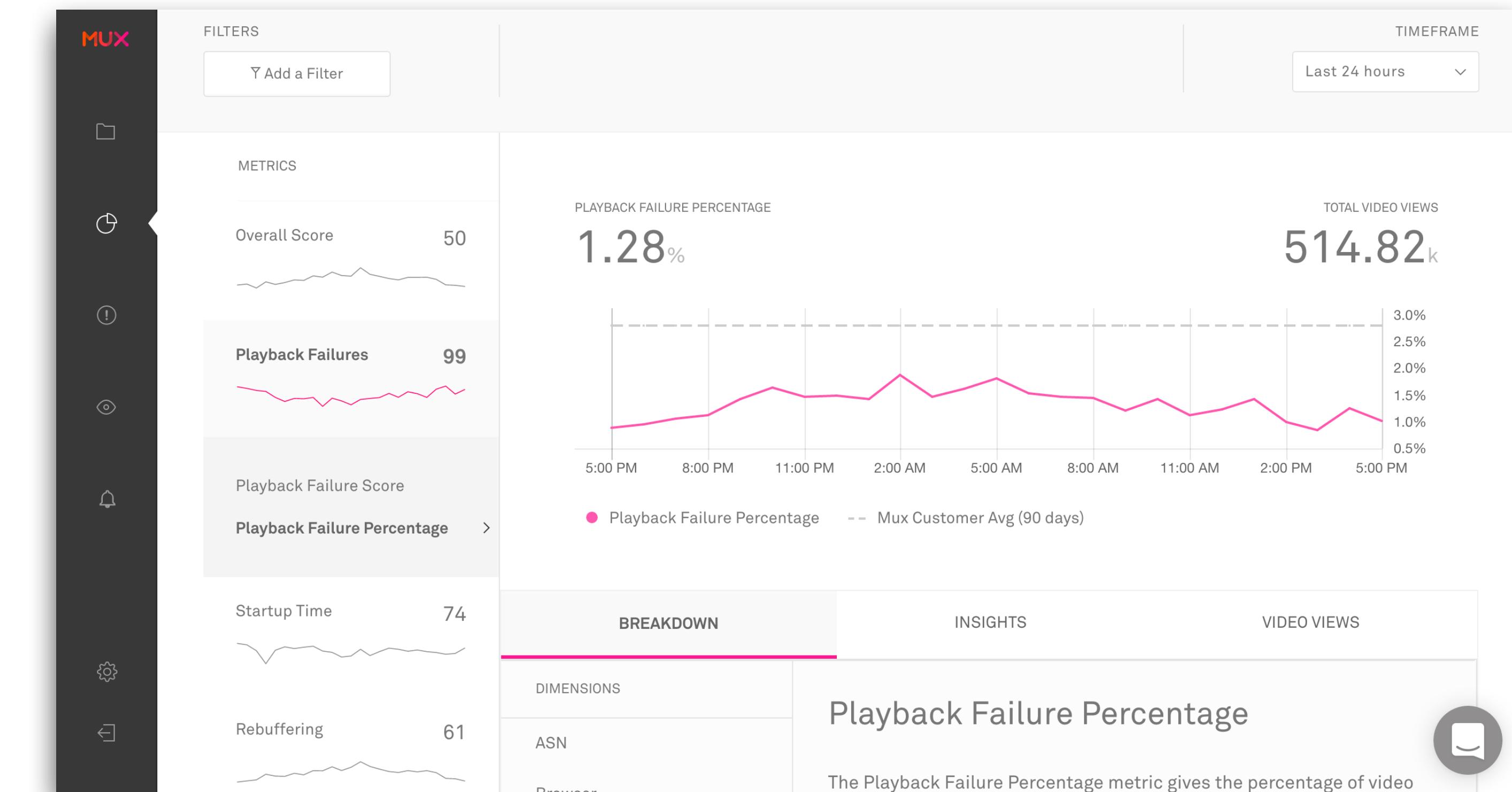
MUX

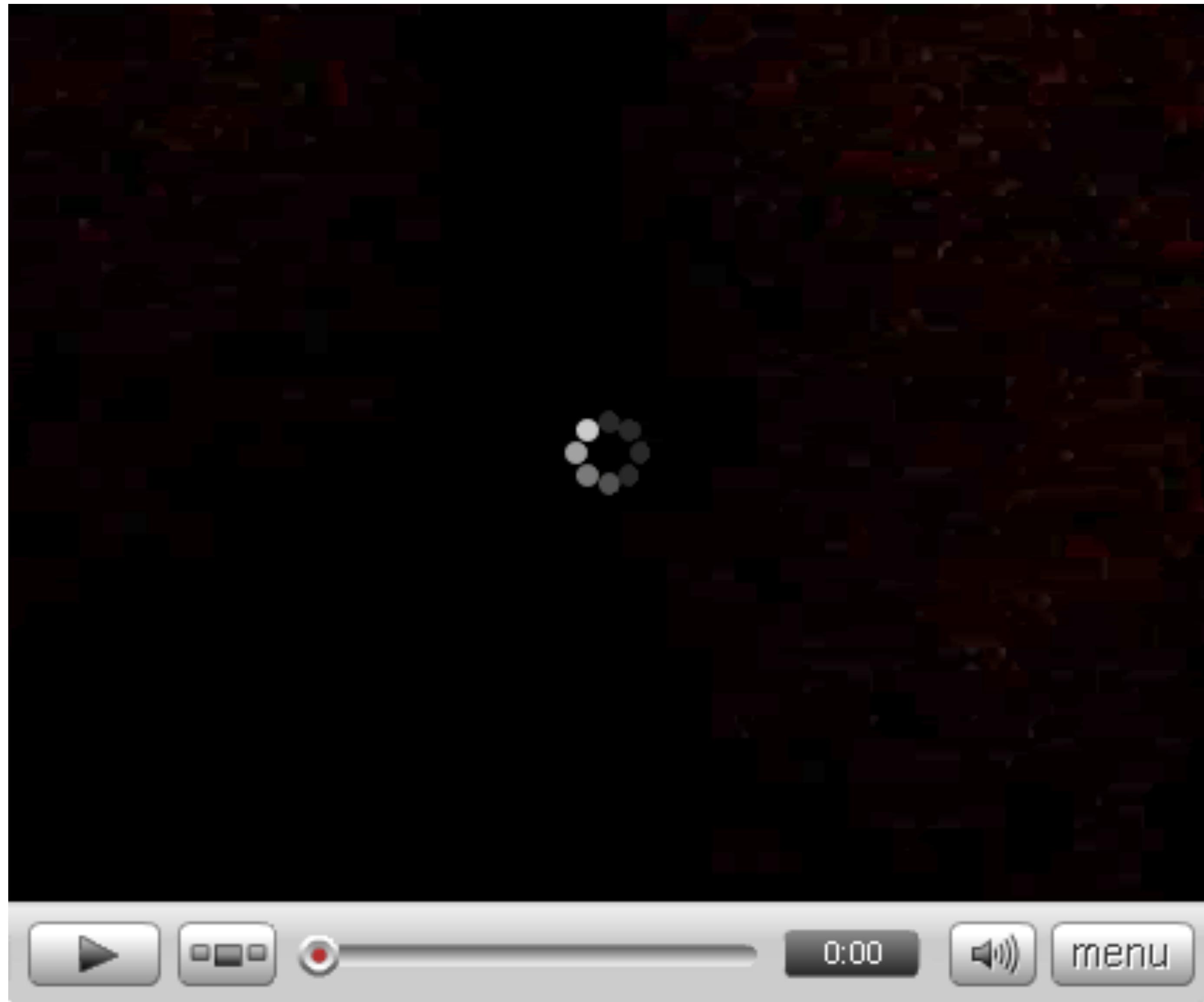
Building a Real-Time Anomaly-Detection System with Flink @ Mux

Scott Kidder
Software Engineer @ Mux

What is Mux?

- Real-time analytics for video
- Customers include PBS, Funny or Die, IGN, Wistia
- Track playback failures, start-up time, rebuffering, and more
- Process millions of video views per day





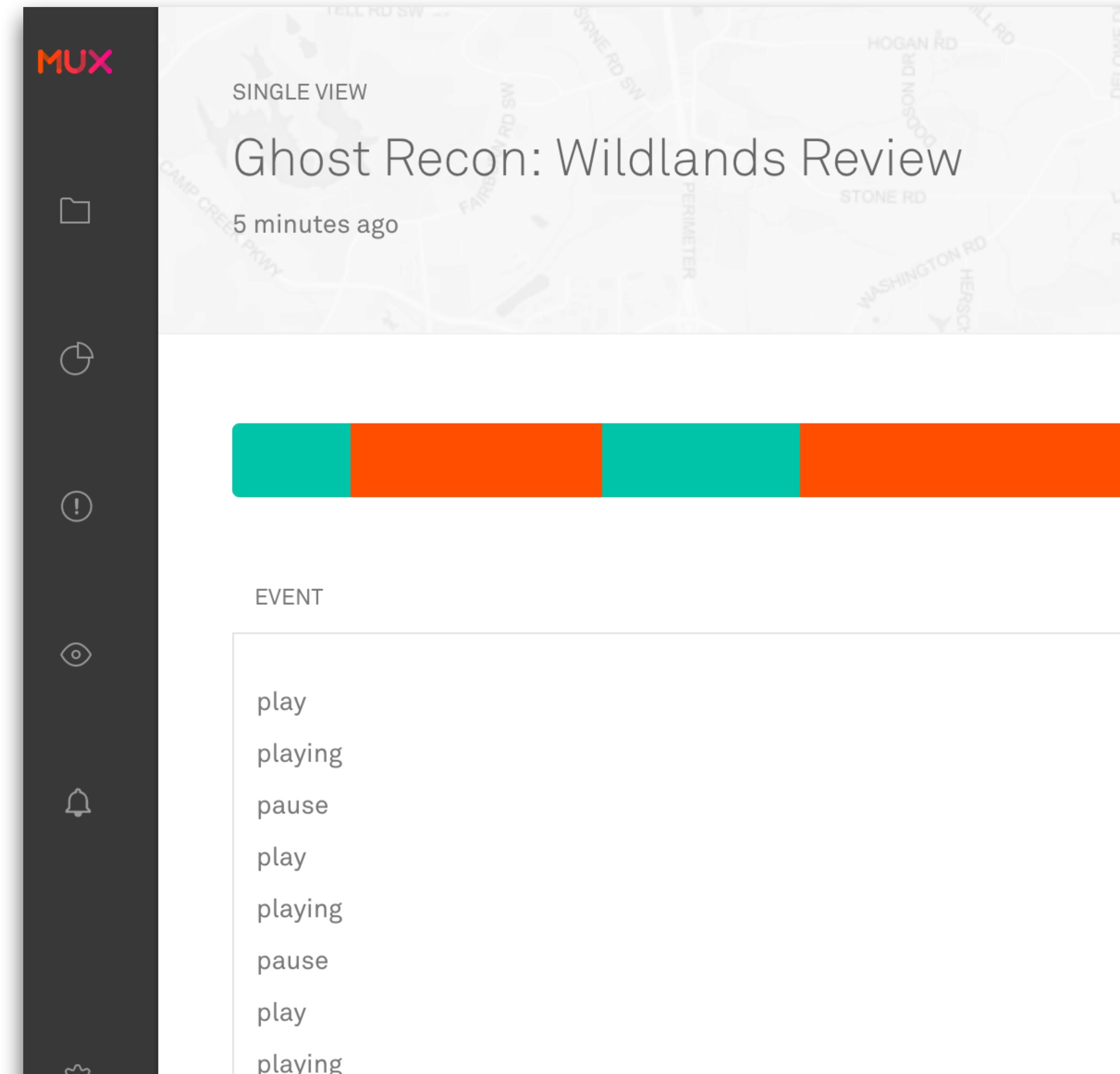
“All production flows have a basic characteristic: the material becomes more valuable as it moves through the process.”

— Andy Grove, High Output Management

What processing adds the most value?

Making Video-Delivery a Monitored Service

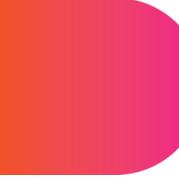
- Susan Fowler's book "Production-Ready Microservices" lists components of a well-monitored microservice:
 1. Logging
 2. Dashboards
 3. Alerting
 4. On-call Rotation
- Mux supported the first two in the form of video-event ingestion and a pretty web dashboard, but lacked alerting and an on-call rotation





Types of Error-Rate Alerts

- **Property Alerts**
 - Problems affecting an entire customer property
 - **Example:** CDN publishing for HTTP Live Streaming is broken, resulting in widespread live-streaming failures.
- **Video-Title Alerts**
 - Problems affecting specific video-titles within a customer property
 - **Example:** Poorly encoded/mislabeled video is published to catalog

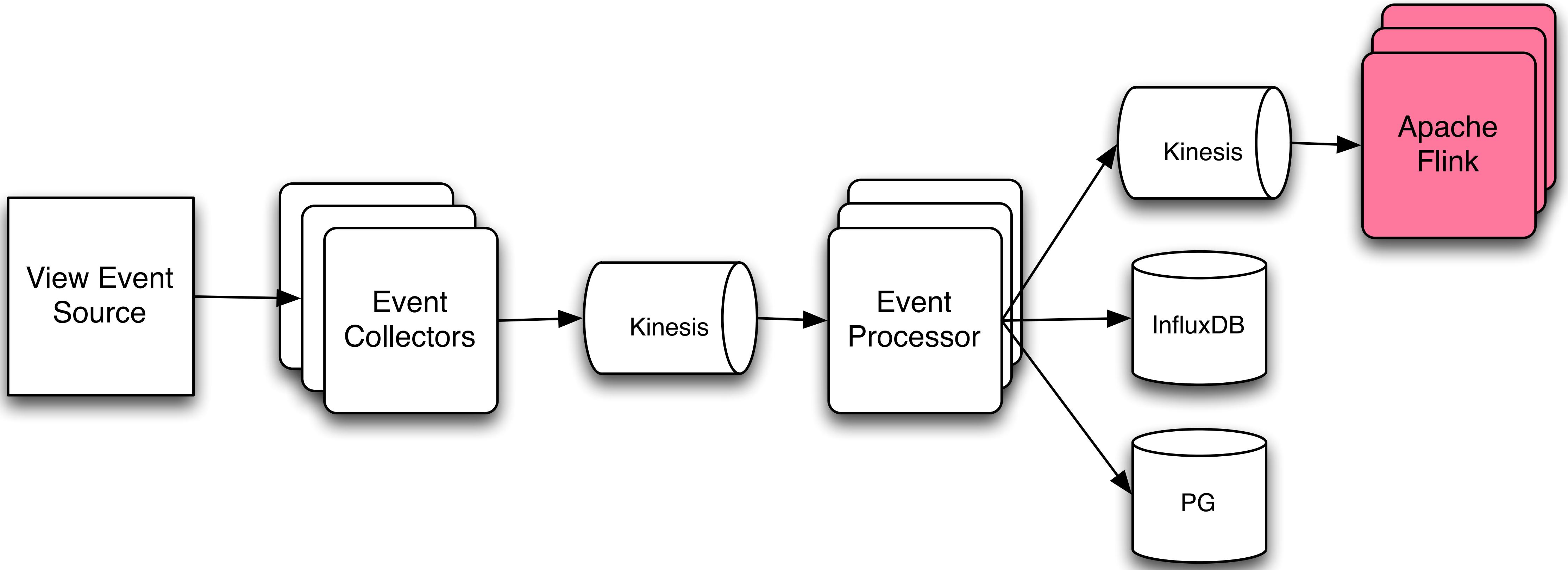


Alerting Technical Requirements

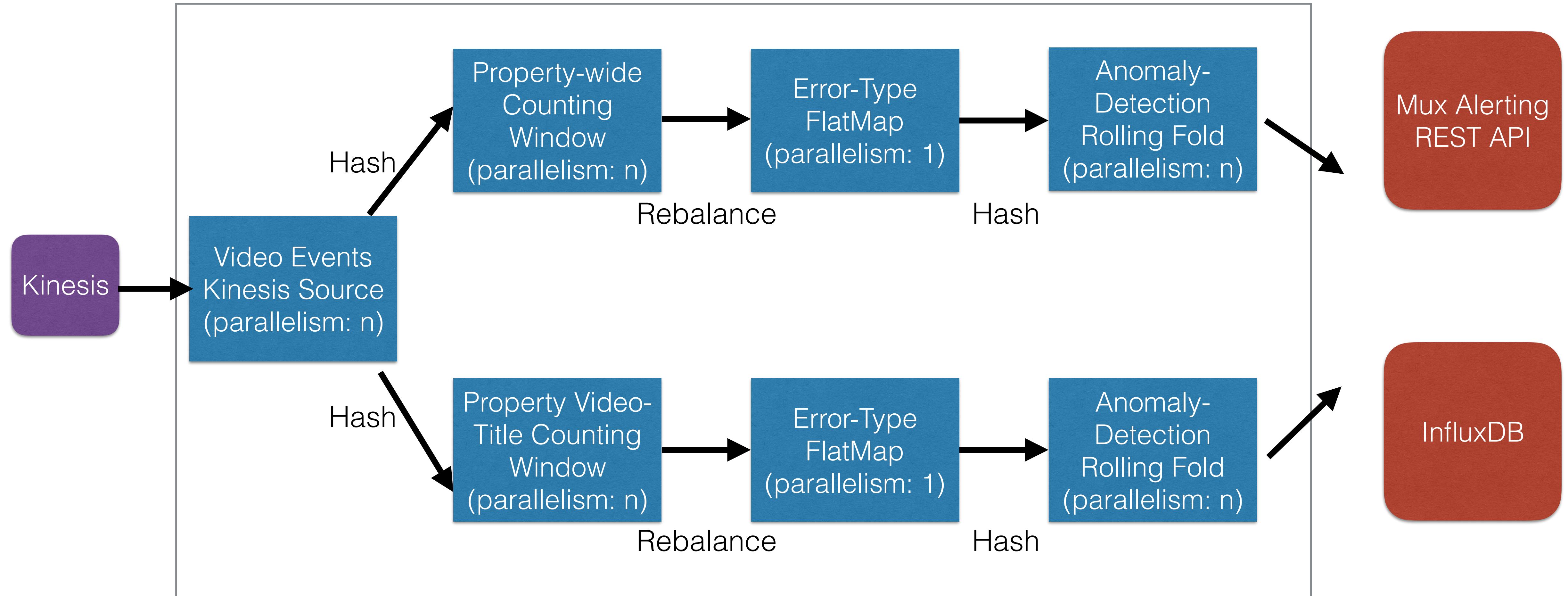
- Needed a system to detect error-rate anomalies in video-views across a customer property and for every video title
- Very low latency, high-availability
- Horizontally-scalable on AWS commodity hardware, preferably running in a Docker container
- Easy to use at every stage: prototyping, development, production
- Read from AWS Kinesis streams, but preferably support Kafka too

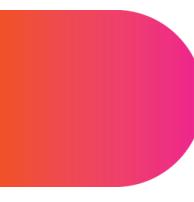
Application Design

Event Ingestion Architecture



Flink Execution Plan

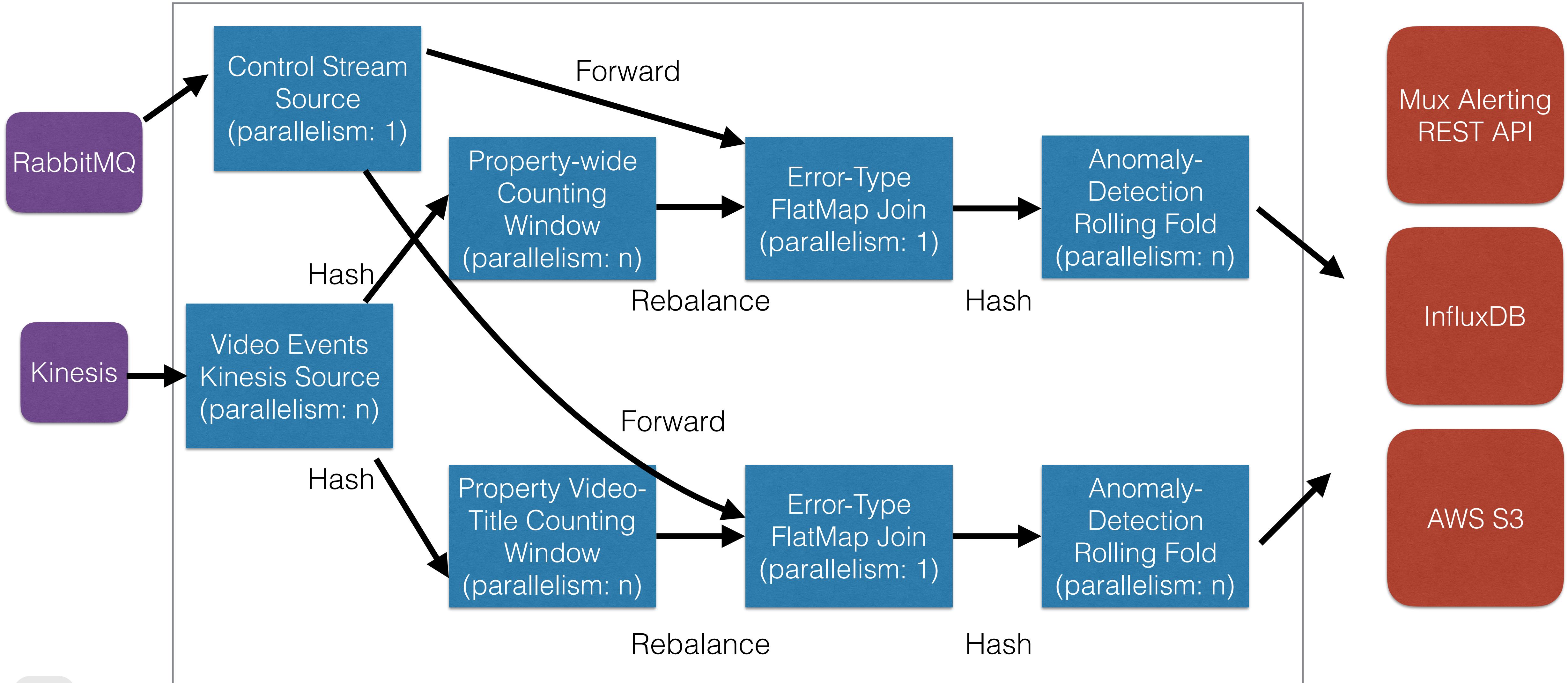




Introducing a Control Stream

- Data pipeline worked great, but needed ability to interact with running application without a restart
- Added a simple RabbitMQ stream source that accepts control messages
- Control messages feed into the FlatMap operator
- Control operations include:
 - Dump error-rates to S3 for each property/error-type permutation
 - Dump active alert-incident state to S3

Flink Execution Plan with Control Stream



Deployment and Operations

Docker

- All services at Mux are deployed in Docker containers
- Created a custom Docker image of Flink built from source
- Use BuildKite to build Docker image and push to Docker Hub
- Same image for Flink Job Manager & Task Manager
- Configure Flink using environment-variables
- Used with Alpine & Debian-Jessie base-images successfully
- Deploy Flink Standalone Cluster with Rancher

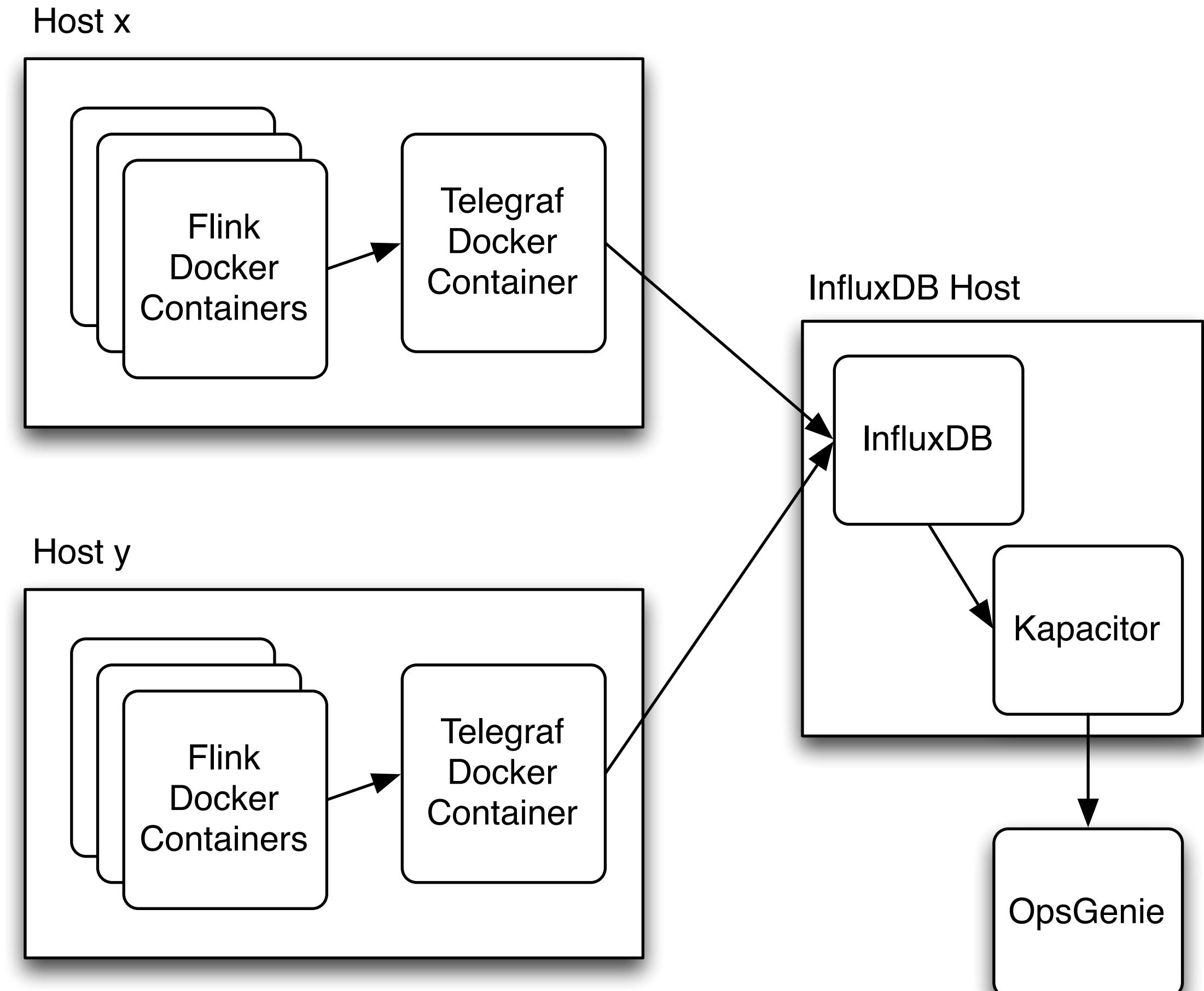


Builds & Behavioral Testing

- Use BuildKite to build our Flink application JAR
- BuildKite build runs in a Docker container on AWS EC2 instances
- Behavioral tests written in Cucumber (Ruby)
- Cucumber tests run against a set of Docker containers brought up using Docker-Compose: Flink, Kinesalite (Kinesis clone), Minio (S3 clone), RabbitMQ, InfluxDB
- Docker-managed networking to connect services

Internal Monitoring

- Use Statsd to emit Flink Metrics about Flink cluster
- Telegraf Docker container consumes Statsd metrics and writes to InfluxDB
- Kapacitor monitors InfluxDB writes
- Kapacitor scripts can trigger alerts(OpsGenie, PagerDuty, etc)



Mux Alerting UI

Listing of Alert Incidents

MUX

Alerts

See all incidents that have been alerted on

Notification Settings

OPEN INCIDENTS ALL INCIDENTS

ID	DETAILS	OPEN FOR	VIEWS IMPACTED	VIEWS IMPACTED / HR
13331	Video Title Grupo Formula 104.1 is failing at a significant rate (86.0%) due to an error of networkError	6 minutes	184	2660
13332	Overall error-rate is significantly high (22.4%) due to an error of networkError	8 minutes	300	3121
13323	Video Title Enlace is failing at a significant rate (86.0%) due to an error of networkError	an hour	97	532
13320	Video Title Bank of Montreal Rings the NYSE Opening Bell is failing at a significant rate (88.0%) due to an error of networkError	an hour	44	1022

Slack Notifications for Alerts

 [Open: Incident #13331](#)

Video Title
Grupo Formula 104.1

Error
networkError

Error Percentage	Rate
86.0%	11908 / hour

 <https://dashboard.mux.com/properties/177/incidents/13331>

 [Open: Incident #13332](#)

Error
networkError

Error Percentage	Rate
22.4%	5486 / hour

 <https://dashboard.mux.com/properties/177/incidents/13332>

Alert Incident Details

MUX

ERROR
networkError

INITIAL ERROR PERCENTAGE
22.4%

INITIAL RATE
5486 / hour

INCIDENT TRIGGER

Opened At	03/07/2017 7:25 AM
Error Percentage	22.4%
Alert Threshold	14.7%
Minimum Window	1000 views

CURRENT STATUS

Open For	12 minutes
Views Impacted	670 views
Impact/hr	3792 views

NOTIFICATIONS: ON

Silence Alert Notifications

INCIDENT TIMELINE ([View in metrics](#))

OTHER OCCURRENCES

an hour ago	Duration: 23 minutes	>
2 hours ago	Duration: 24 minutes	>
5 hours ago	Duration: an hour	>
8 hours ago	Duration: 2 hours	>
10 hours ago	Duration: an hour	>

Thank You!

Comments on Streaming Deep Learning with Flink

Dean Wampler, Ph.D.
dean@lightbend.com
@deanwampler

Flink Forward San Francisco, April 11, 2017



1

- Why DL for Streaming? Why DL for Flink?
- Model Training vs. Inference
- Practical Approaches with Flink
- The Future of DL with Flink

Outline – This is a status report of work in progress at Lightbend. It's far from complete and definitely not "battle tested", but this talk reflects our current thinking and approach.

Why DL for Streaming?
Why DL for Flink?

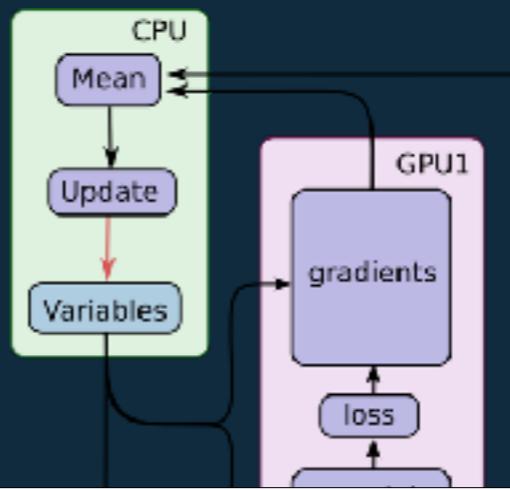
Because deep learning is “so hot right now.”



DeepMind



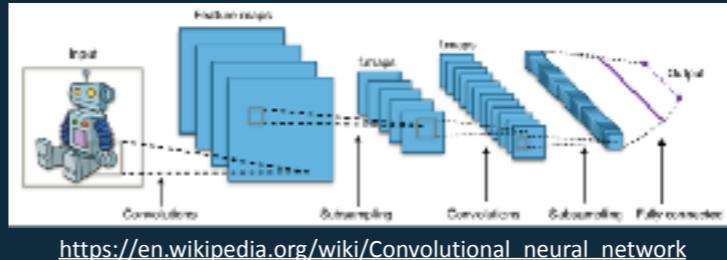
DL4J
DEEPMLEARNING4J



Images: <https://github.com/tensorflow/models/tree/master/inception>, <https://deeplearning4j.org>, <https://github.com/tensorflow>, and <https://deepmind.com/>

What Is deep learning?

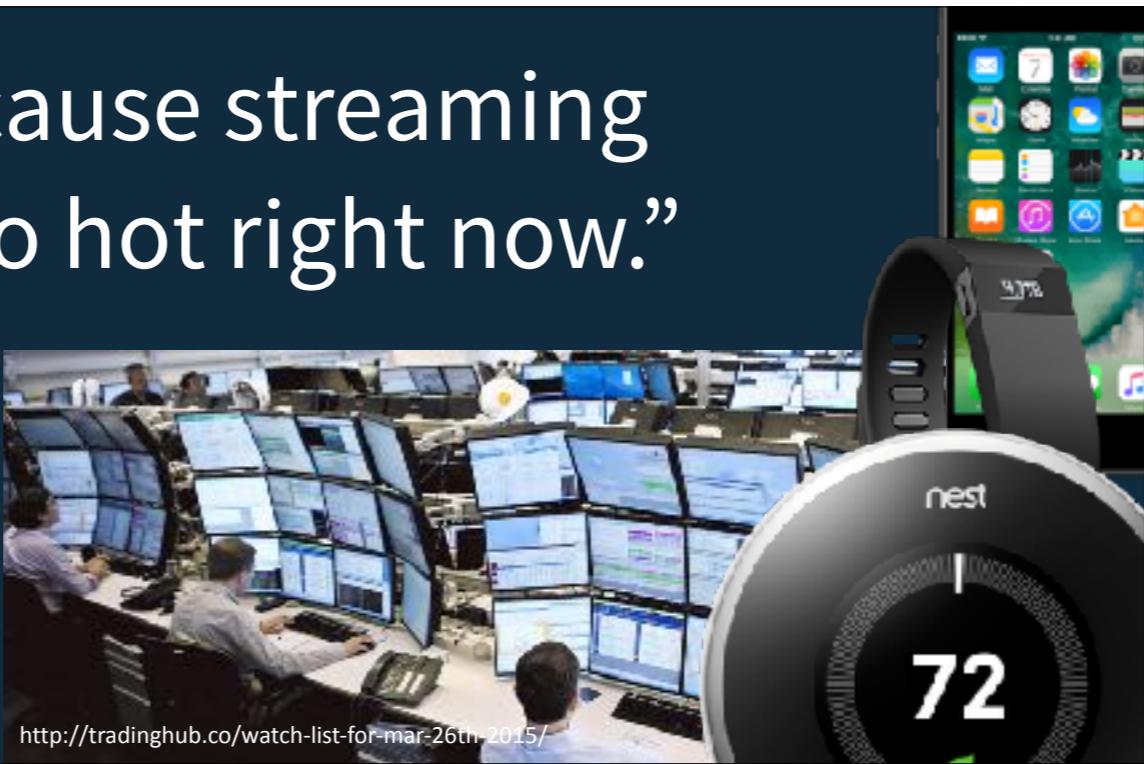
- A very sophisticated non-linear model
- A way to discover features, rather than engineer them



https://en.wikipedia.org/wiki/Convolutional_neural_network

I won't explain DL, but offer a few points. It's not mysterious. It's a sophisticated non-linear modeler. It's also a tool for learning the "features" (the characteristics of the data), rather than engineering them explicitly.

Because streaming
is “so hot right now.”

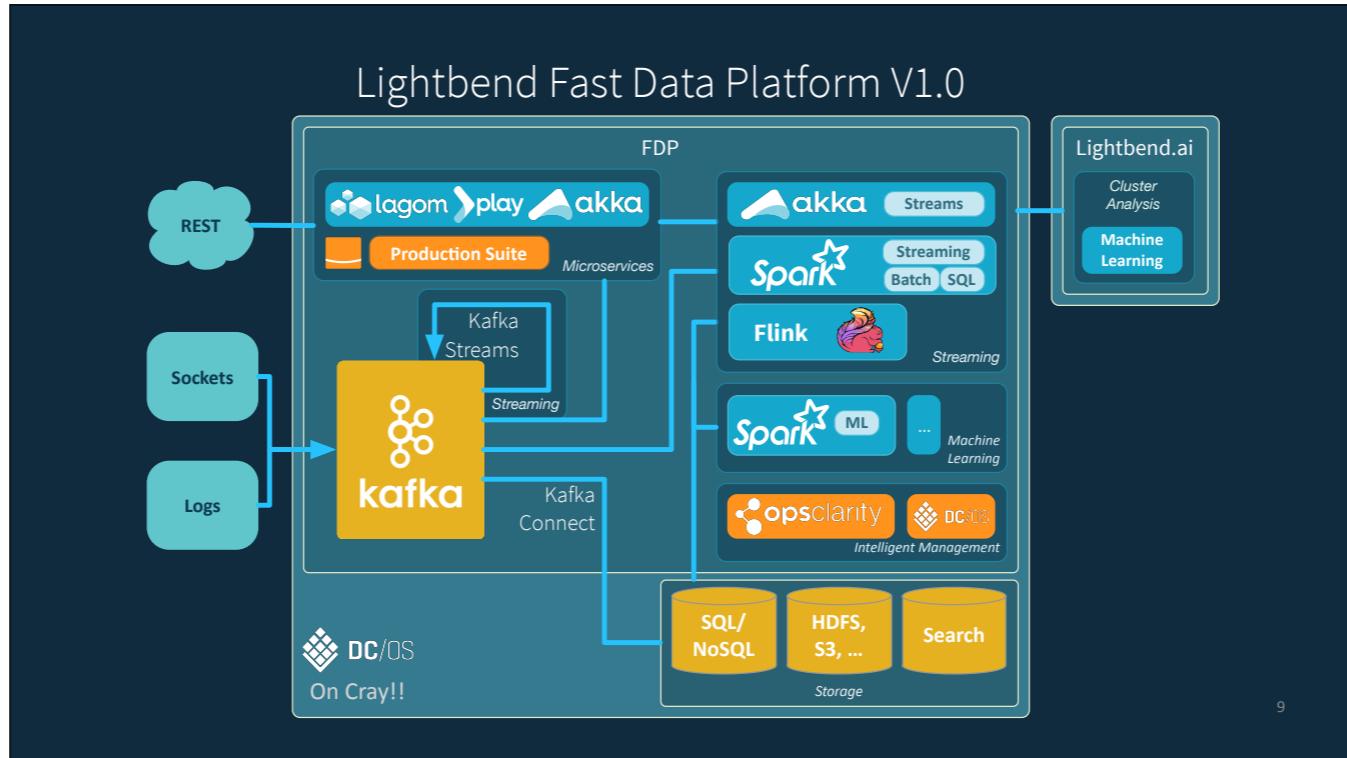


... and Flink provides large scale and low latency!

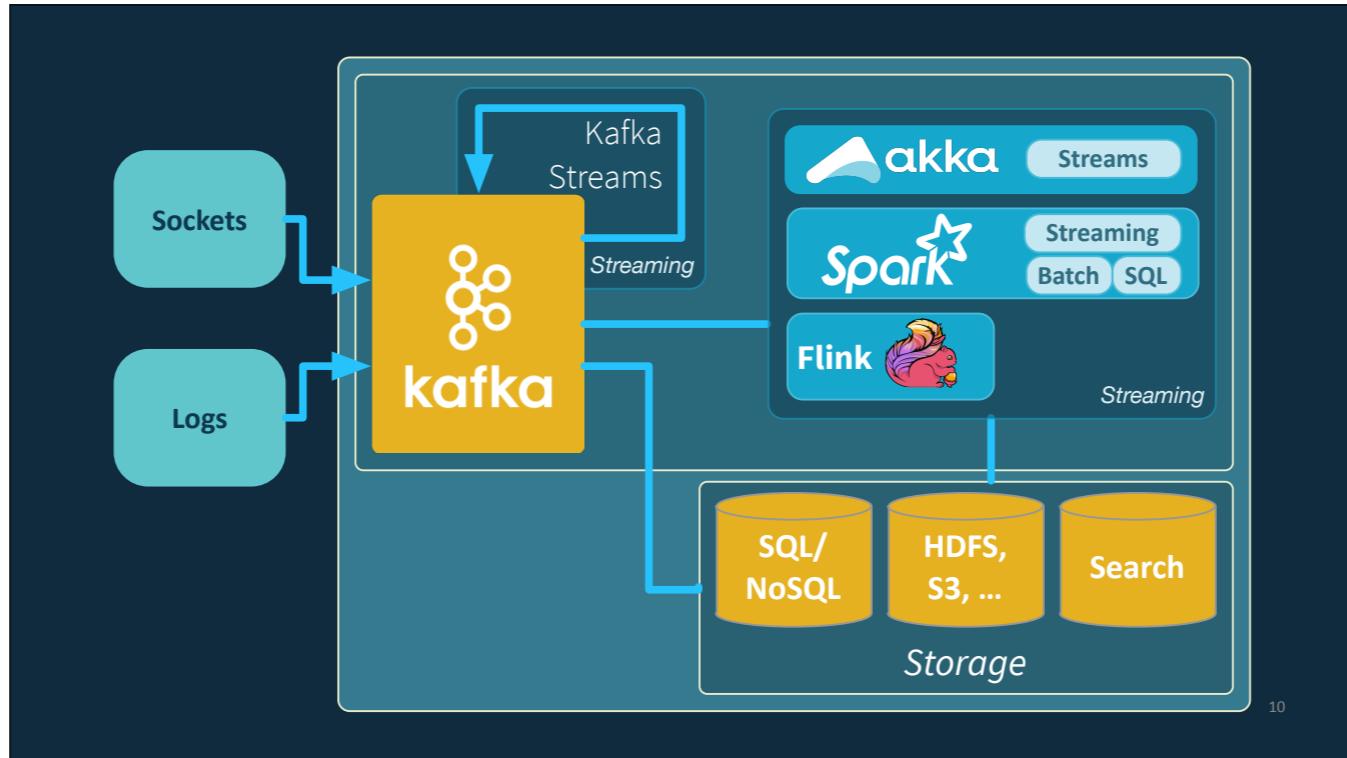


<http://tradinghub.co/watch-list-for-mar-26th-2015/>

Why Lightbend likes Flink

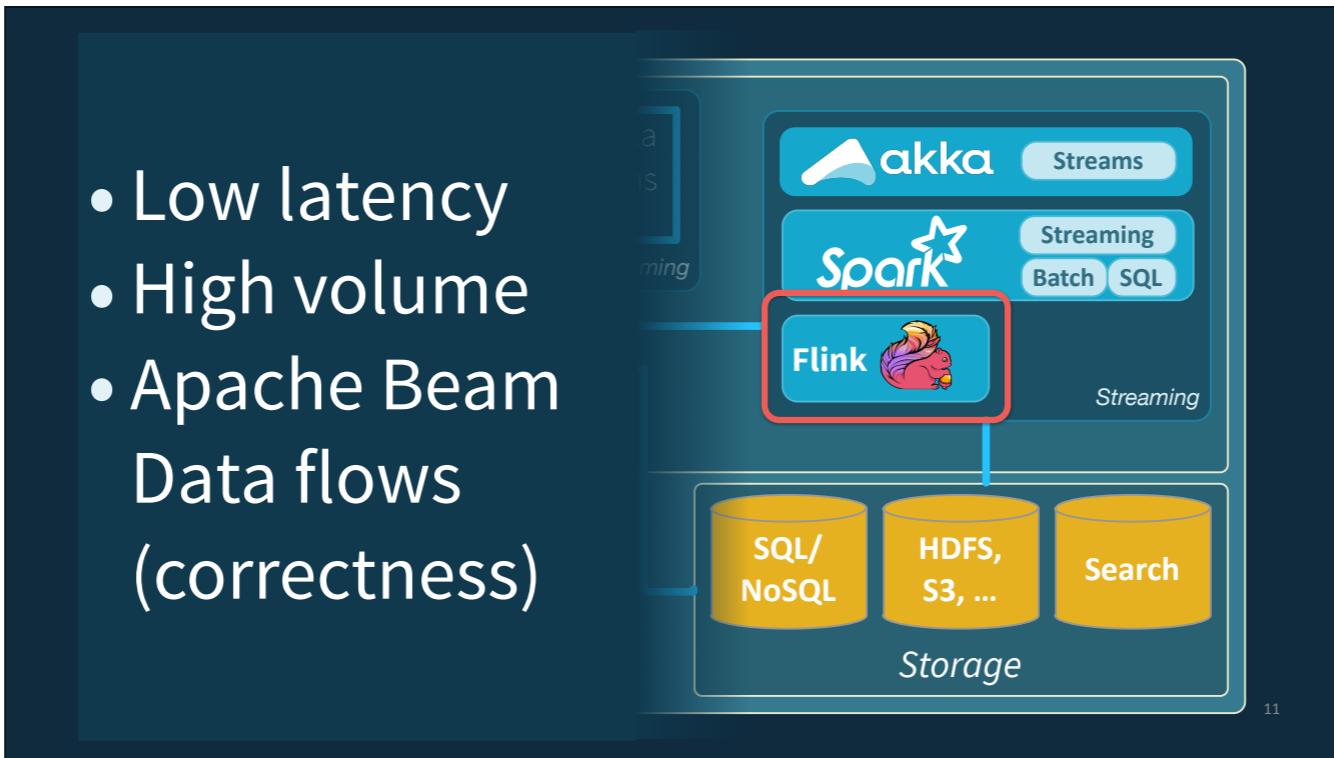


The Fast Data Platform that we're building as a commercial, streaming data processing stack. I won't go through the details (but I'll show a link at the end for more information). Rather, I'll focus on the streaming core...



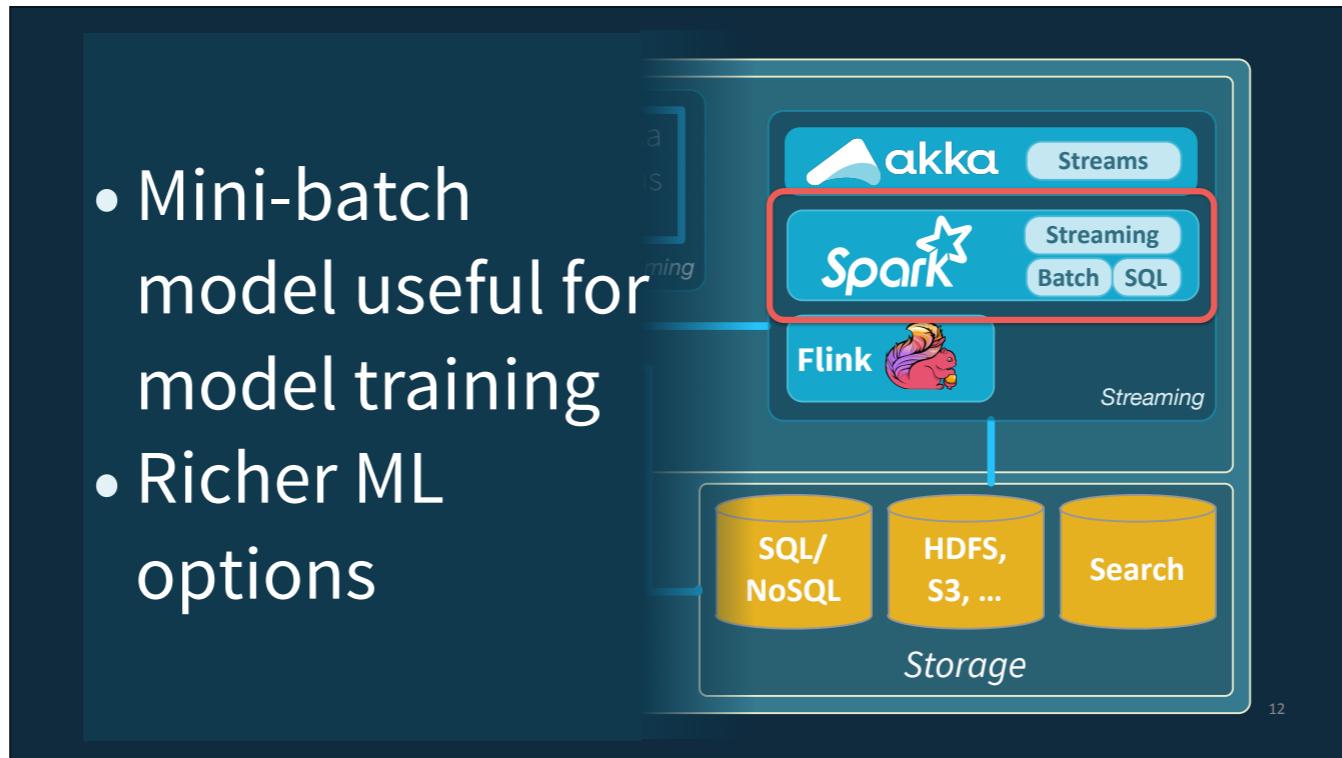
The core: It requires a backplane designed from streaming – Kafka here. To meet the full spectrum of streaming needs, we're bundling 4 stream processing engines, because no single engine meets all needs. Finally, persistence is required.

- Low latency
 - High volume
 - Apache Beam
- Data flows
(correctness)



Flink contrasts mostly with Spark. It fixes two (current) limitations: 1) low latency instead of medium latency from the current mini-batch streaming model in Spark, 2) Flink provides state of the art semantics for more sophisticated stream processing, especially when high accuracy (as opposed to approximate values) is important. These semantics are defined by Apache Beam (a.k.a., Google Dataflow). Flink can run Beam data flows (The open source Apache Beam requires a third-party runner). Both Spark and Flink provide excellent, scalable performance at high volumes.

- Mini-batch model useful for model training
- Richer ML options



Spark has medium latency (~0.5 seconds and up), optimized for excellent, scalable performance at high volumes, but it's not a true streaming engine. We would rather use Flink for low-latency model application (scoring), but training models incrementally through is one option we'll discuss. Also, the Spark ecosystem is ahead of Flink's in terms of integrations with ML tools.

Challenges Using Flink for Deep Learning

Use a Static Model for Inference?

- This is (relatively) straightforward. You load a model at startup and apply it (score data) in the stream.
- See Eron Wright's talk!

First, if you have a static model and you want to use it from Flink ("as a map function" – Eron Wright), this is relatively straightforward.

Train the Model and Do Inference?

- May not be reasonable to do training per record with low latency
- But see Swaminathan Sundararaman's talk

Harder problem is training the model on the stream and doing inference

Train the Model and Do Inference

- If too slow, could train mini-batches in windows with longer latency
 - Periodically update the model
 - Do inference with low latency using “old” model

Distributed Training

- You have to train on each data partition separately, then merge the model.
- Actually, DL4J does this, then averages the model parameters.

Model Training vs. Model Serving (Inference)

Kinds of Training

- Batch - all data at once
- Mini-batch - sample batches (with replacement!), iterate the model with each batch
- “Online” - iterate the model with each record



The “with replacement” means that by design, you keep sampling the same data set, so each record will appear in 0, 1, or more mini-batches.

Mini-batch Training

- Originally developed for offline use, where it's too expensive to train with all the data.
- Would “mini-batches” of stream data work?



That is, mini-batch training was originally invented as a more efficient, if potentially less accurate way to train offline, where each mini-batch samples the same data set repeatedly and the model improves with each iteration (this does work). What if the “mini-batch” is actually a group of N new records instead? By this definition, Online is a mini-batch where N = 1.

Online Training

- Go to Trevor Grant's talk next!



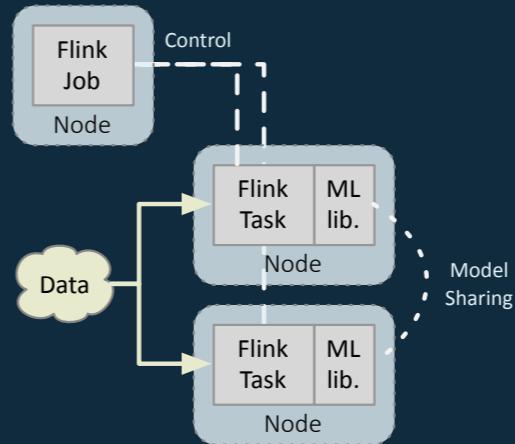
Trevor is going to discuss online learning scenarios in depth.

Practical Approaches with Flink

Design Approaches

1. Same Job - Call Library

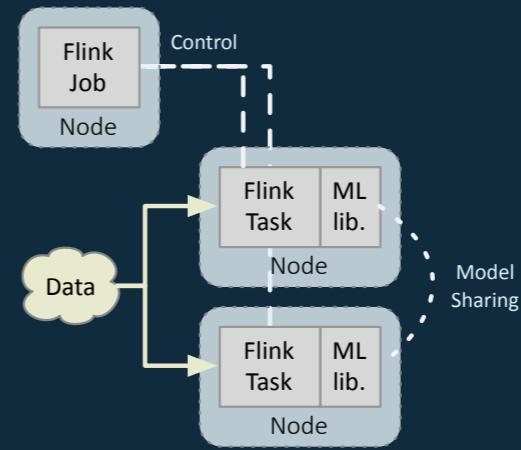
- Training and scoring using the same ML library, in the same job.



The main problem is how to share the model across the tasks running across a cluster. A model/parameter server is one answer.

1. Same Job - Call Library

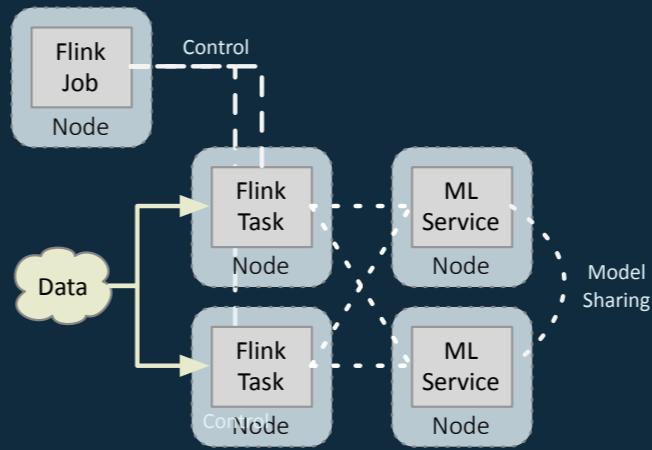
- But what does that mean in a distributed system like Flink?
- Must share model across the cluster & tasks.



The main problem is how to share the model across the tasks running across a cluster. A model/parameter server is one answer.

2. Same Job - Call Service

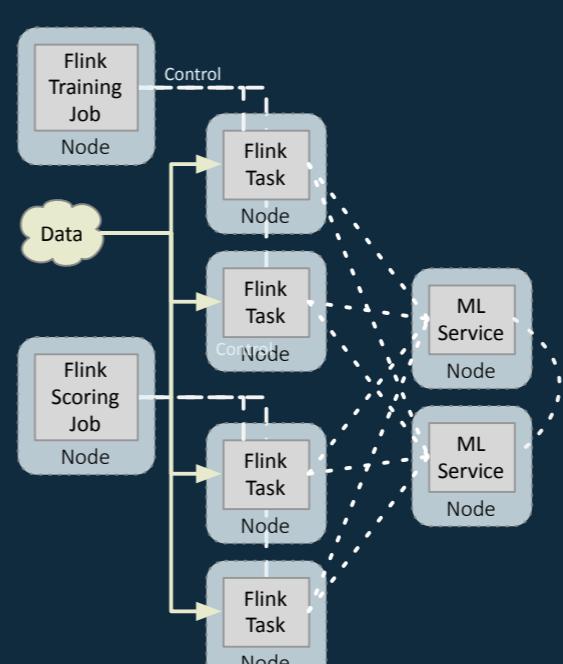
- Training and scoring are done using Async I/O to a separate service, but from the same Flink job.



This service might itself be distributed. It may or may not run on separate nodes from Flink tasks.

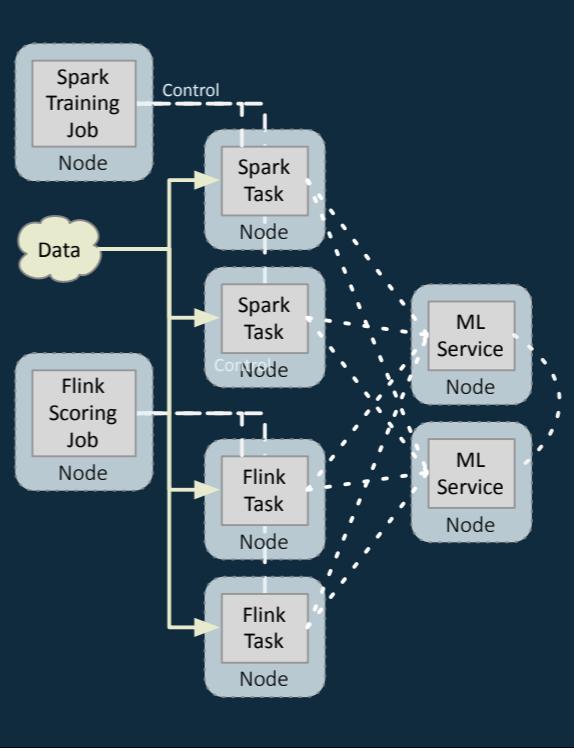
3. Two Flink Jobs

- Training and scoring using separate Flink jobs.
- Could either use a ML library or service.



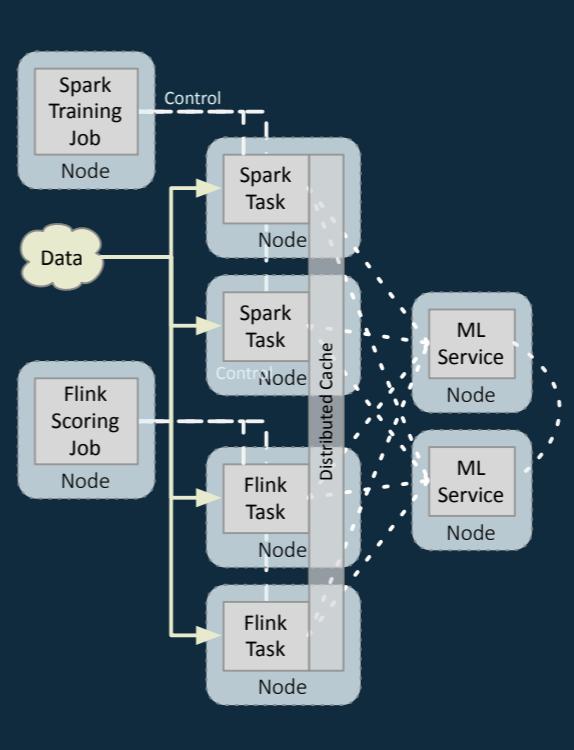
4. Two Different Systems

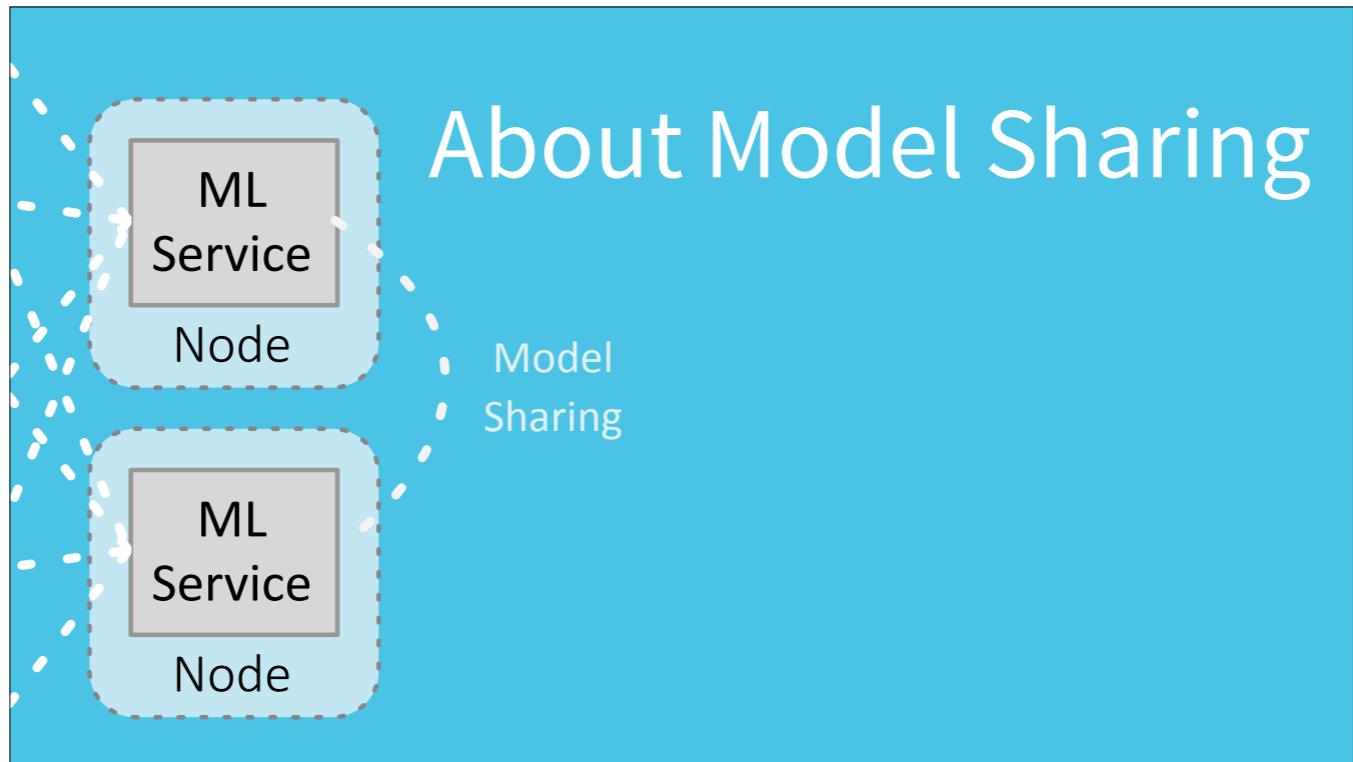
- Training with Spark Streaming or Batch (for example) and scoring with Flink.



5. Cache is King!

- Use a distributed cache
 - Alluxio (Tachyon)
 - Hazelcast
 - ...





Parameter Servers

- Scalable, low-latency.
- Serve both training and scoring engines.
 - ND4J parameter server
 - TensorFlow serving system
 - Clipper - prediction serving system
 - PredictionIO - ML server

ND4J provides multi-dimensional arrays for Java, targeting CPUs, GPUs, etc. Mimics Numpy, Matlab, and Scikit-learn.

ND4J parameter server: <https://github.com/deeplearning4j/nd4j/tree/master/nd4j-parameter-server-parent>

TensorFlow parameter server:

Clipper: <https://github.com/ucbrise/clipper>

Deeplearning4J

- JVM-based.
- Discussed at last year's Flink Forward.
- Use Flink Dataset to load data into DL4J's DataVec format, then feed to DL4J service.

<https://www.slideshare.net/FlinkForward/suneel-marthi-deep-learning-with-apache-flink-and-dl4j>

Leveraging Flink Features

Side Inputs

- Use case: Join a stream with slowly changing data.
 - So, for ML, what do you join with what?
 - Keyed or broadcast?
 - Windowed or not (“global window”)

<https://cwiki.apache.org/confluence/display/FLINK/FLIP-17+Side+Inputs+for+DataStream+API>

```
// Side Input sketch
// Example adapted from https://cwiki.apache.org/confluence/
// display/FLINK/FLIP-17+Side+Inputs+for+DataStream+API

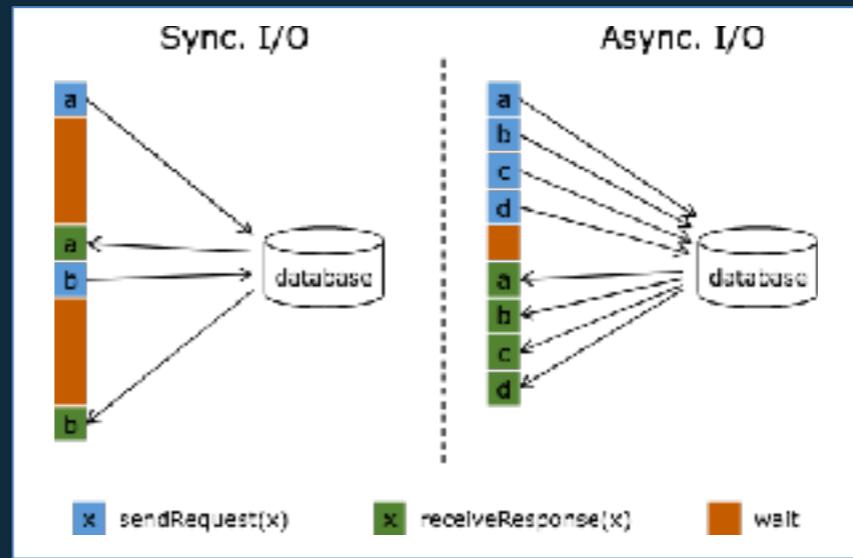
val dataStream: DataStream[Record] = ...
val modelStream: DataStream[Params] = ...

val modelInput = new SingletonSideInput(modelStream)

dataStream
  .map { record =>
    val params = getRuntimeContext().getSideInput(modelInput)
    val score  = model.withParams(params).score(records)
    (score, record)
  }
  .withSideInput(modelInput);
```

Async I/O

- Callback response handler



Async I/O

- Might be better for periodically retrieving model parameter updates.
- Calling to a model server for scoring could be too inefficient?

```
// Async I/O sketch
// Example adapted from https://ci.apache.org/projects/flink/
// flink-docs-release-1.2/dev/stream/asyncio.html

/**
 * Params is a type that encapsulates the notion of a
 * sequence of model parameters, each of which knows its
 * location in the model, etc.
 */
case class Params(...) extends Iterable[Param] {...}

/**
 * Handle asynchronous requests for the entire set of model
 * parameters, where a String key is used to specify the
 * model.
 */
```

```
/**  
 * Handle asynchronous requests for the entire set of model  
 * parameters, where a String key is used to specify the  
 * model.  
 */  
class AsyncMLModelParameterRequest  
  extends AsyncFunction[String, Params] {  
  
  /**  
   * An ML Model-specific client that can issue concurrent  
   * requests with callbacks. Would be customized for  
   * DL4J, TensorFlow, etc.  
   */  
  lazy val client = MLModelClient(host, port, ...)  
  
  /** The Flink context used for the future callbacks. */  
  implicit lazy val executor =  
    ExecutionContext.fromExecutor(Executors.directExecutor())
```

```
override def asyncInvoke(
    key: String,
    asyncCollector: AsyncCollector[Params]): Unit = {

    // Issue the asynchronous request, receive a
    // future for the result
    val paramsFuture: Future[Params] = client.getParams(key)

    // Set the callback to be executed once the request by the
    // client is complete. The callback simply forwards the
    // params to the collector, exploiting the fact that Params
    // implements Iterable.
    paramsFuture.onSuccess {
        case params: Params => asyncCollector.collect(params);
    }
}
```

The Future of Deep Learning with Flink

Current Flink ML Roadmap

- Today: Offline batch learning
- New: Google Doc investigating:
 - Offline training with Flink Streaming
 - Online training
 - Low-latency model serving

Flink ML: Do

- Focus on 3rd-party integrations:
 - DL4J (like the Spark integration)
 - TensorFlow
 - H2O
 - Mahout
 - ...

Flink ML: Do

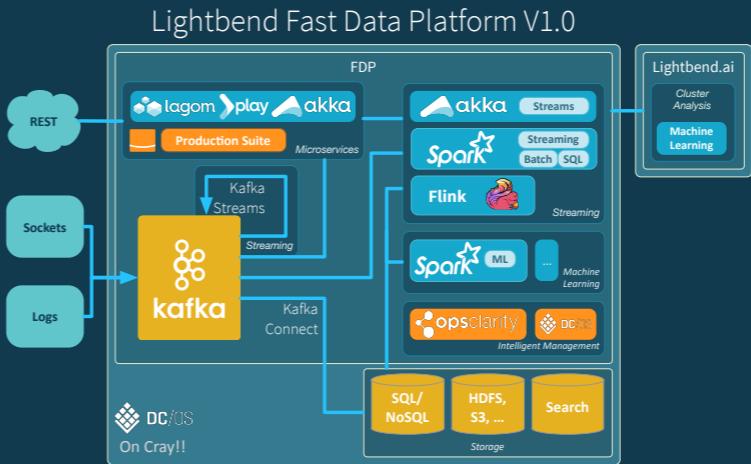
- Complete useful core capabilities, e.g.,
 - [FLINK-5782](#) (GPU support)
 - [FLINK-6131](#) (Side inputs)
- Remove anything in today's ML library that isn't "solid".

Flink ML: Don't

- PMML - doesn't work well enough. Not really that useful?
- Samoa - appears dead
- Reinvent the wheel...

Thank You!

Dean Wampler, Ph.D.
dean@lightbend.com
@deanwampler



lightbend.com/fast-data-platform

For more information on the Fast Data Platform.

Joining the Scurry of Squirrels: Contributing to Apache Flink®



Tzu-Li (Gordon) Tai

@tzulitai
tzulitai@apache.org

dataArtisans

Who?

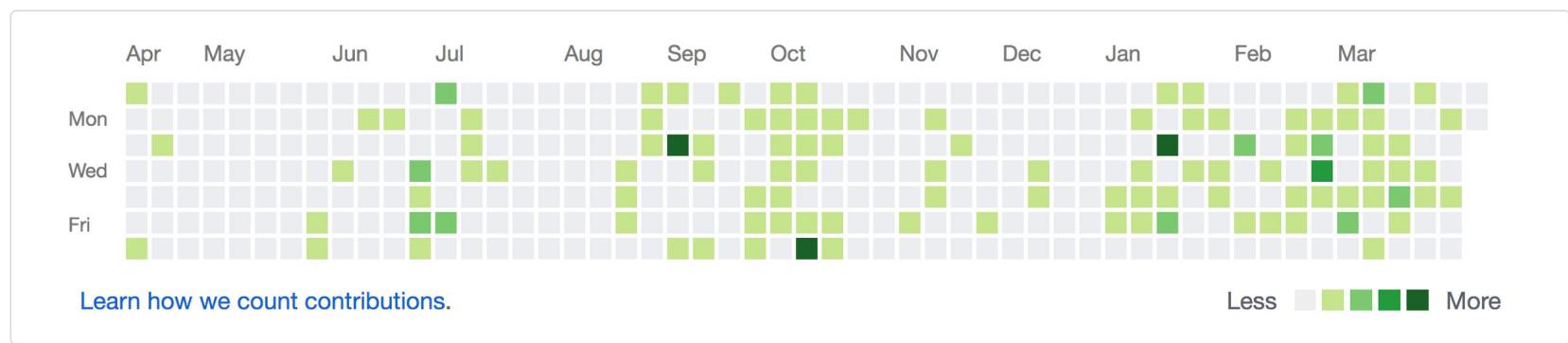


- Apache Flink Committer
 - Streaming connectors (Kafka, Kinesis, Elasticsearch)
- Software Engineer at data Artisans



Who?

- Definitely not a veteran contributor of Apache Flink ;-)

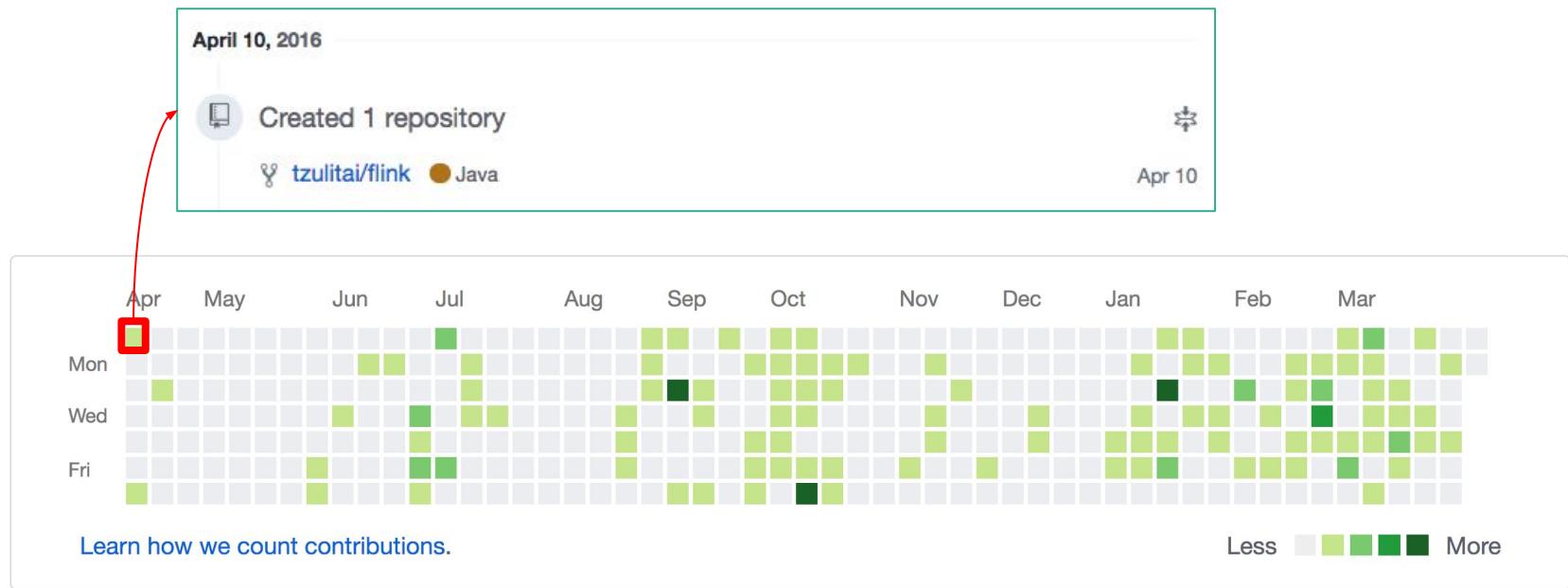


... my life of Apache Flink contribution ;-)

Who?



- Definitely not a veteran contributor of Apache Flink ;-)



... my life of Apache Flink contribution ;-)

The usual first question ...

How can I contribute?

It never is just about the code ...



 **Yehuda Katz** 
@wycats

 Following ▾

As a tech community, we must treat documentation, marketing, logistics, infrastructure, art, etc. work with as much respect as engineering

RETWEETS LIKES
1,004 **1,456**



10:11 AM - 12 Dec 2015

 33  1.0K  1.5K

Different forms of Contribution



- **As an user ...**
 - file bug reports
 - join discussions / propose new features on the mailing list
 - testing release candidates
 - talk about Flink

- **As a developer ...**
 - submit & review patches for features and fixes
 - help answer user questions

Know your resources!

Mailing Lists, JIRA, and Wiki

Know your resources!



- The resources *is* the community, and where everything happens
 - Mailing lists
 - JIRA
 - Wiki



Mailing lists

- Developer mailing list
 - dev@flink.apache.org
 - Ask questions about Flink development
 - Discuss new Flink features

- User mailing list
 - user@flink.apache.org
 - General questions about using Flink

JIRA board



- Issue tracking

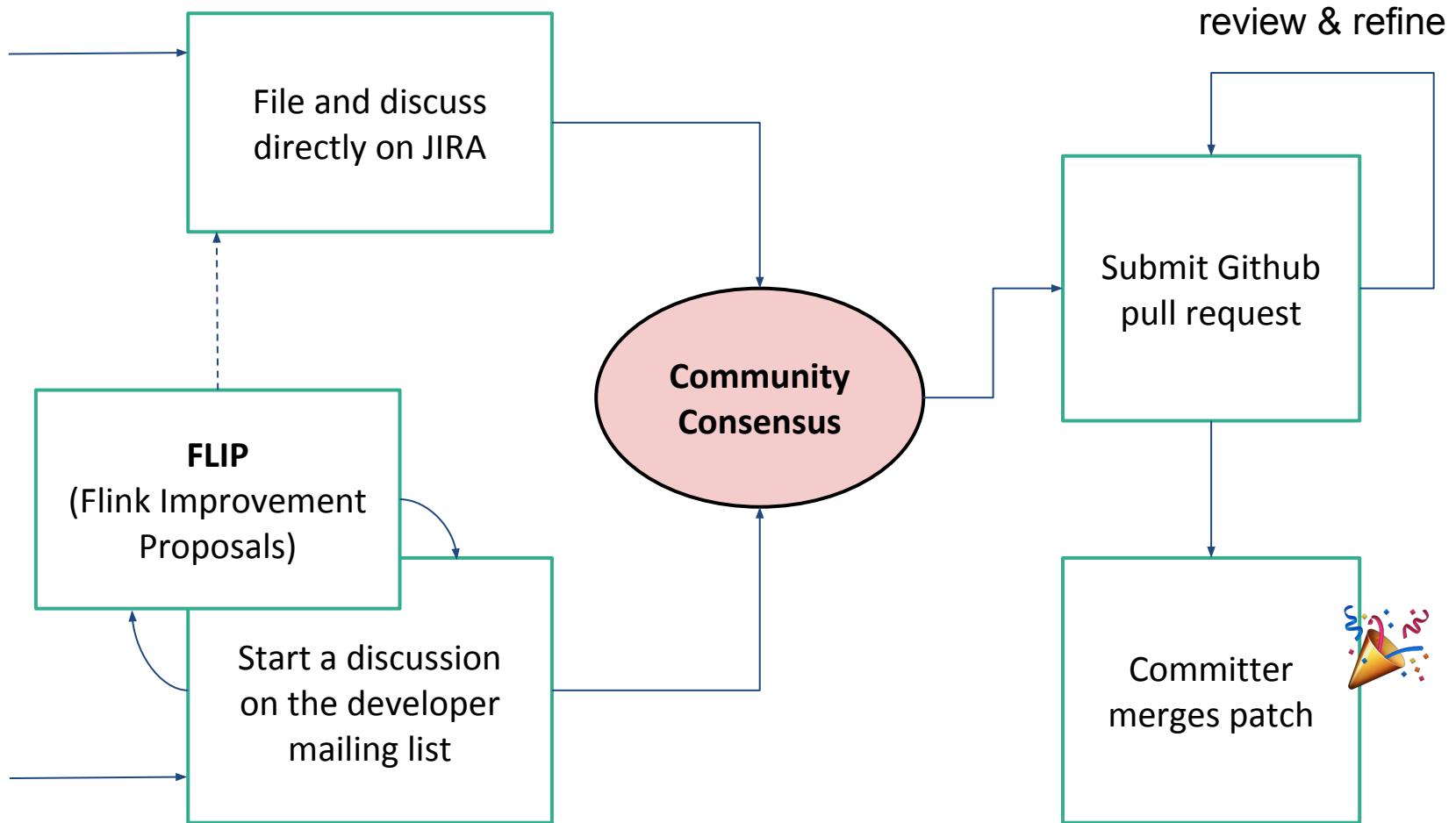
- <https://issues.apache.org/jira/browse/FLINK>
- Where bugs and new features / improvements are filed
- All code contributions must always have a corresponding JIRA issue ticket opened
- Do NOT ask questions here (--> mailing lists)



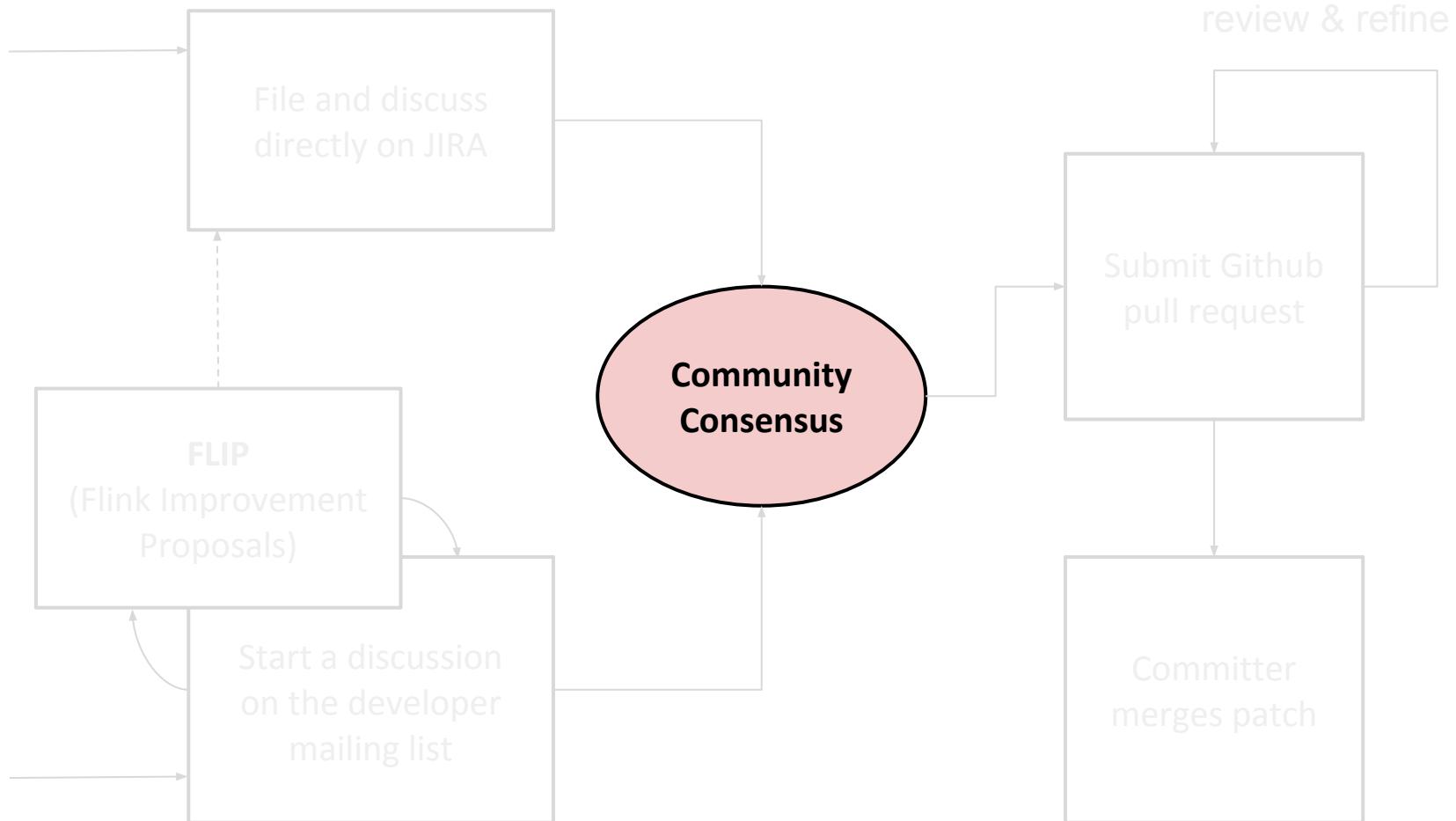
- General information about the project
 - <https://cwiki.apache.org/confluence/display/FLINK>
 - Mostly relevant for FLIPs (Flink Improvement Proposals)
 - Other info somewhat outdated

Apache Flink Patch Submission

Patch Submission Process



Patch Submission Process

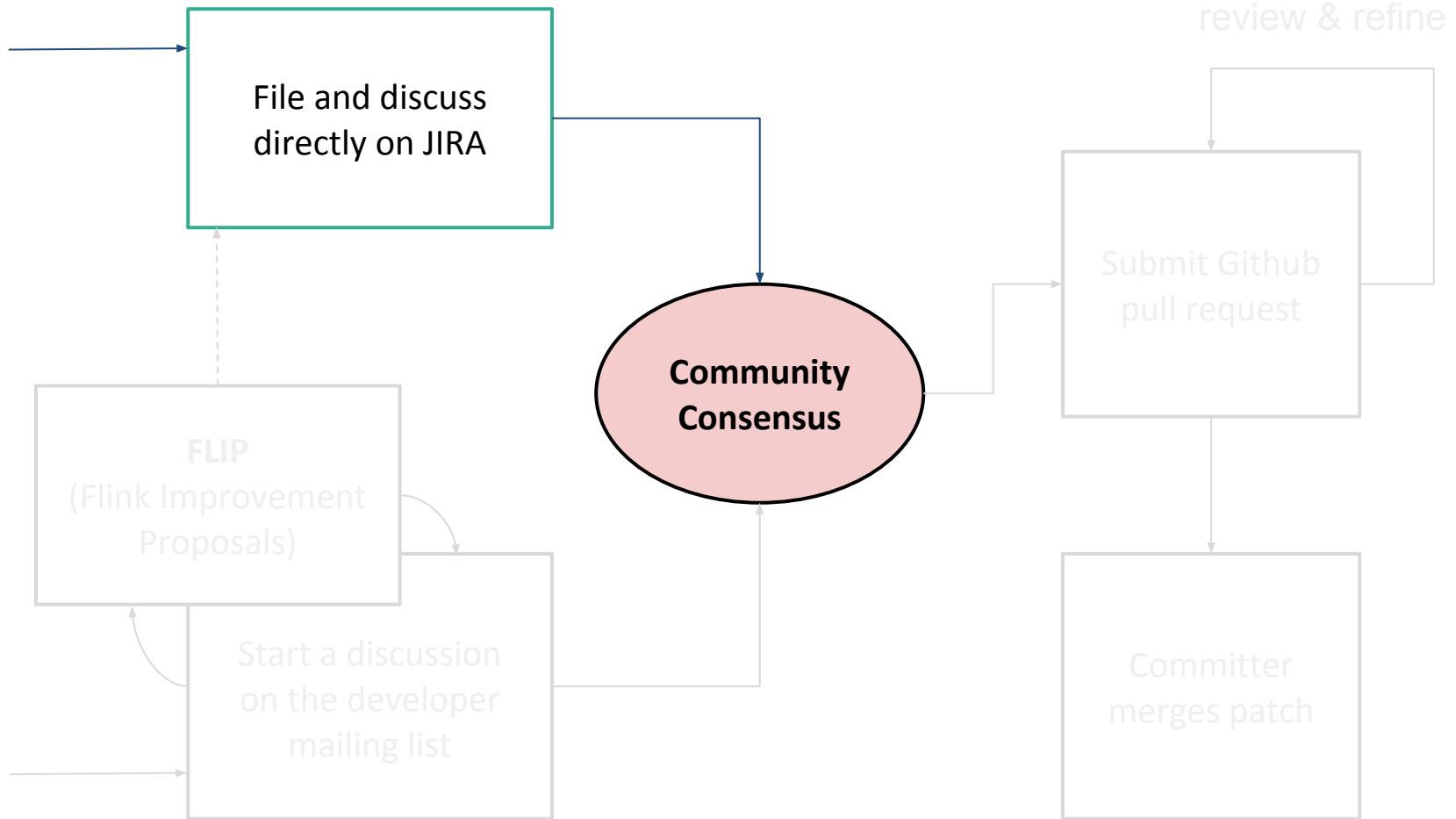


The Apache Way: Community Over Code



- Changes, especially new features, should be agreed upon by the community
 - Does not apply just to the developers
 - Beneficial to users as a whole
- Always seek to reach some level of consensus throughout the code contribution process
 - Avoid needing to go back to feature discussion after coding

Patch Submission Process



Filing a JIRA



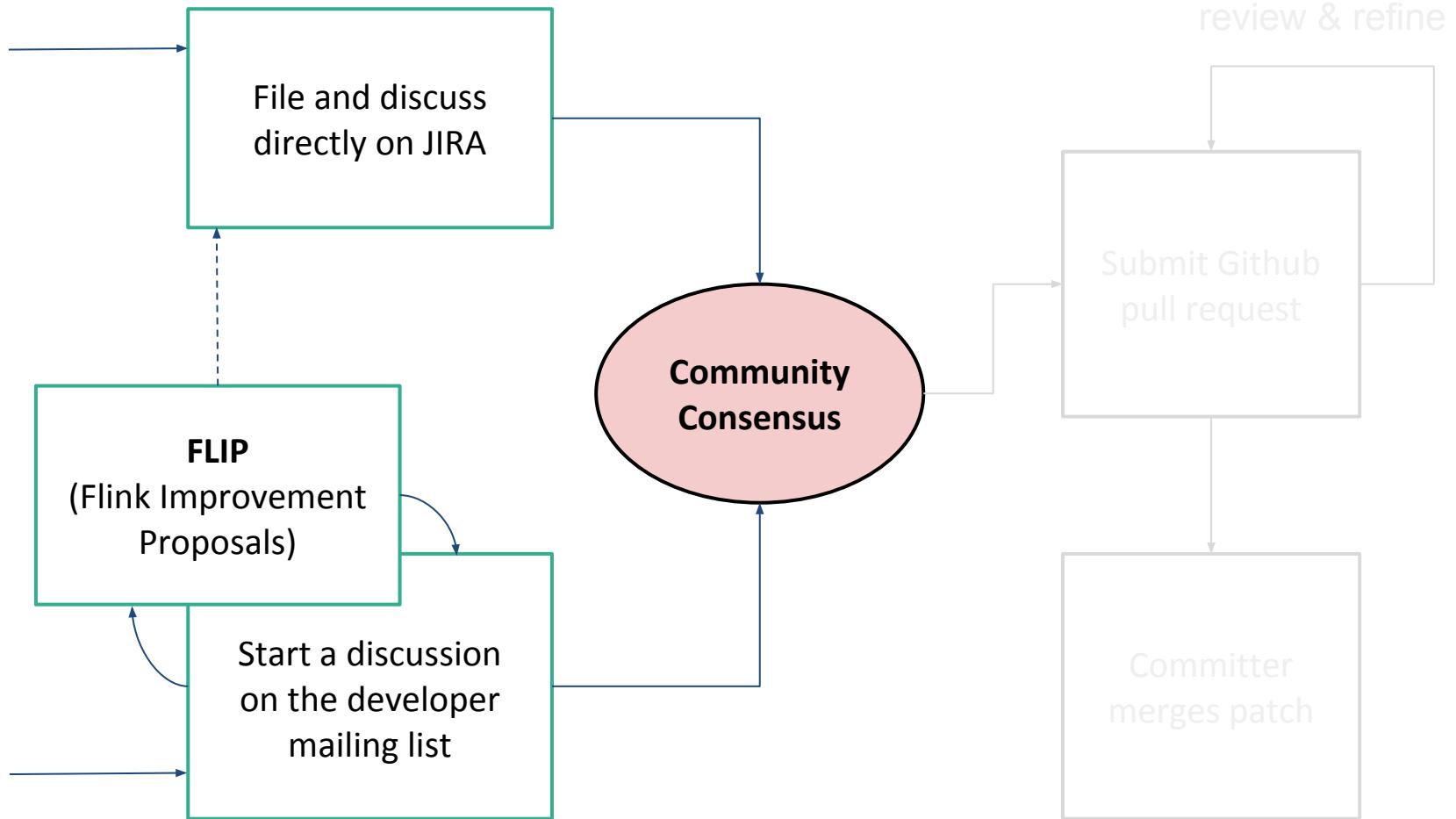
- Always have a good description of what the issue is
 - *Bugs* - the cause and the proposed solution
 - *Improvements and features* - bootstrap potential discussion with the implementation you have in mind
- Remember to appropriately set “Components”



Assign yourself to a JIRA

- Set the “Assignee” field to yourself
 - Notifies others that this issue is already in good hands
- Simply request JIRA permission on the developer mailing list

Patch Submission Process

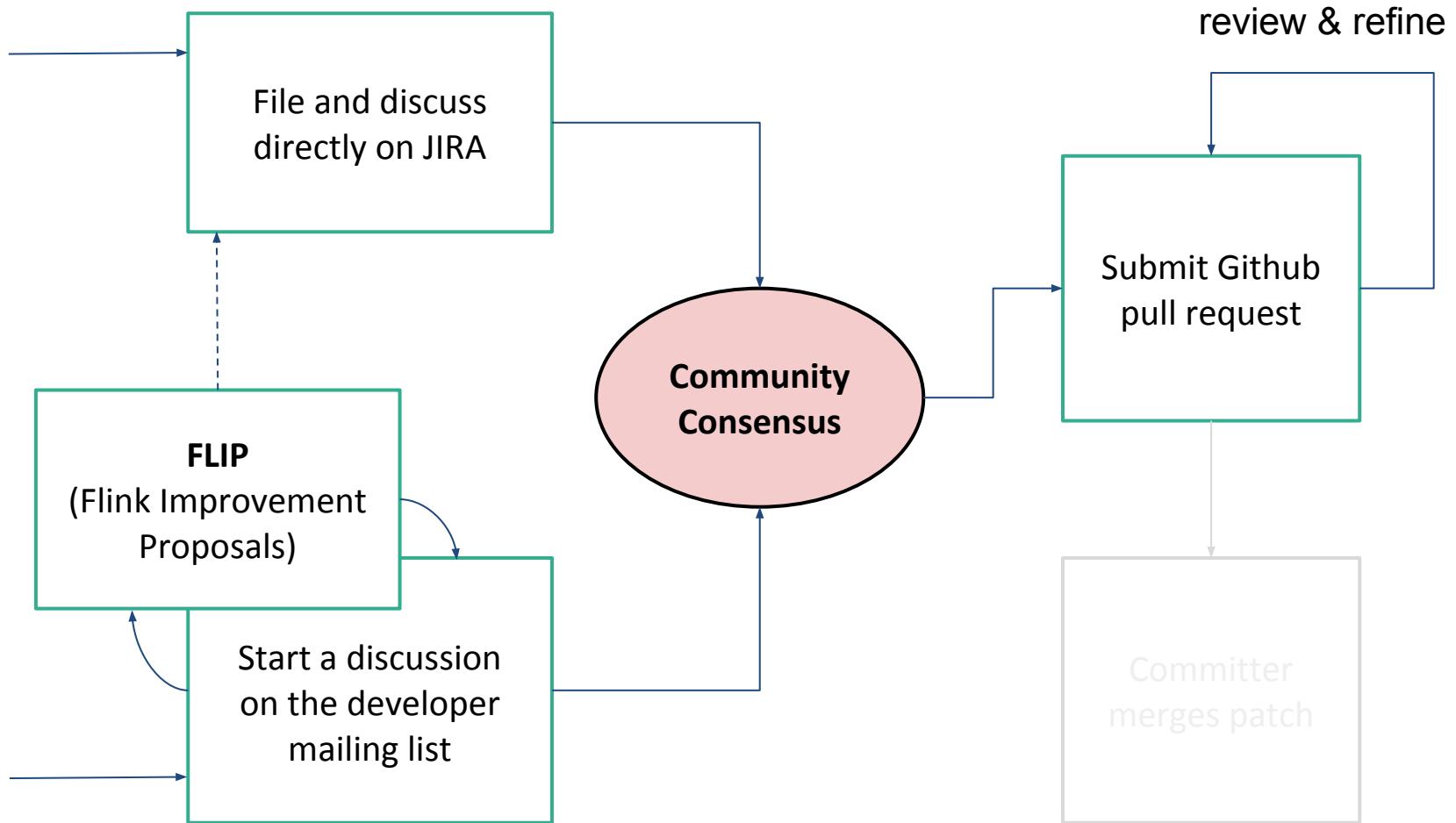


FLIP (Flink Improvement Proposals)



- Adopted from the Apache Kafka community
- Official design documentation for major features
 - Serves as the basis of discussion on the dev mailing list
 - Allows the community to decide on the best future-proof design

Patch Submission Process



Patch Submission Best Practices



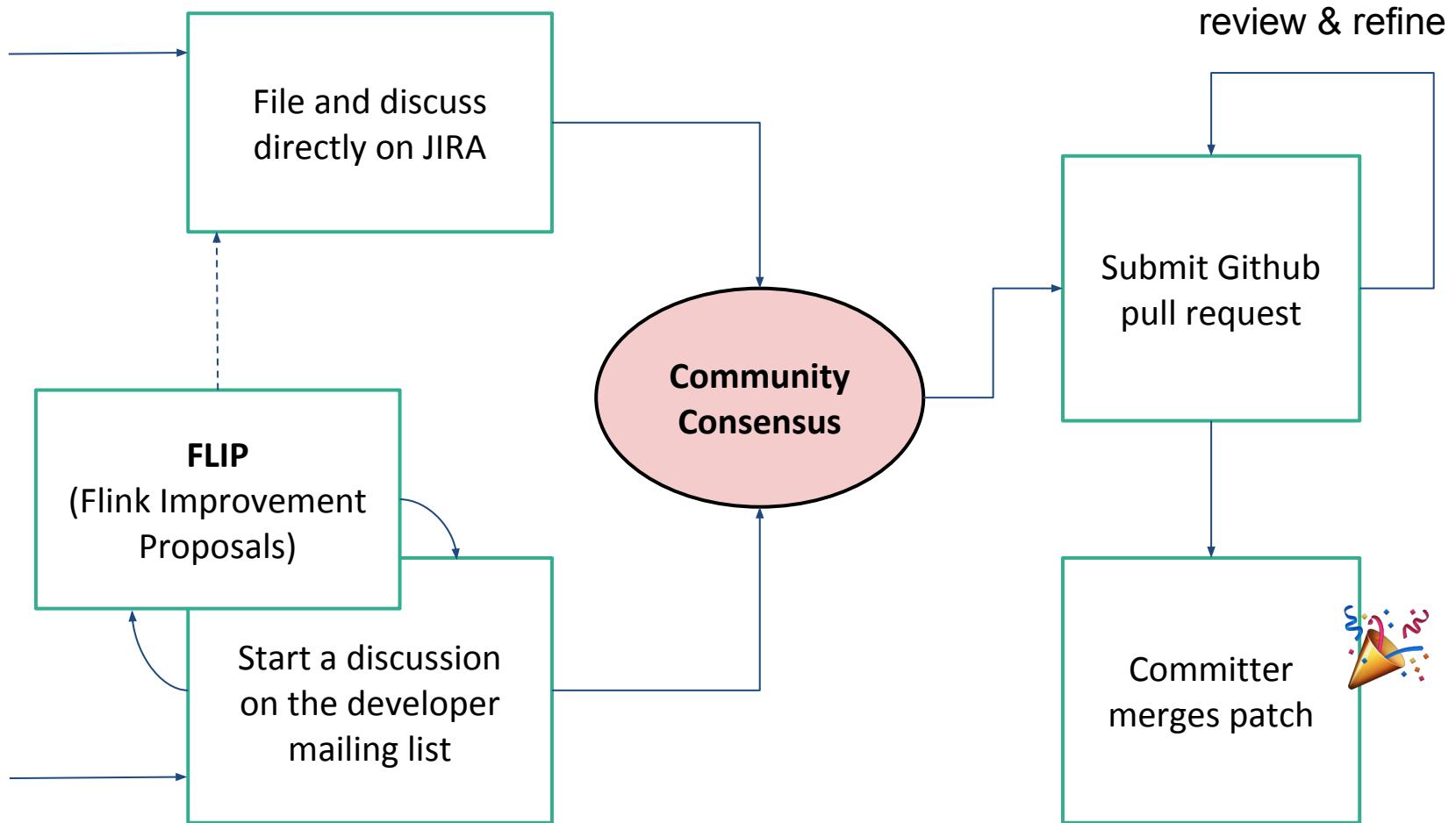
- Rebase onto latest master prior to opening pull request
 - `git pull --rebase origin master`
- Squash your commits to appropriate set
- Commit message example:
 - [FLINK-6025] [cep] Implement skip till next match strategy

Patch Submission Best Practices



- Usually the pull request naming follows the primary commit's message:
 - [FLINK-6025] [cep] Implement skip till next match strategy
- Address all discussions happening on the pull request
- Push follow-up commits
 - Retain history of the patch review
 - Also have meaningful msgs for follow-up commits

Patch Submission Process



Closing

Final Takeaways



- Development of Apache Flink is heavily centered around its community
 - The Apache Way - community over code
 - Contributing is never just about writing code
- Know the resources and do not hesitate to approach.
- Extra pair of helping hands is always welcome ;-)



Shoutouts

- Flink CEP
 - Complex event processing
 - Pattern detection for streams
 - → If interested, contact
Kostas Kloudas (kkloudas@apache.org)
- Flink ML
 - Online learning
 - Incremental learning
 - Model serving
 - → If interested, contact
Theodore Vasiloudis (tvas@apache.org)

Thank you!

@tzulitai
@ApacheFlink
@dataArtisans

Dynamically Configured Stream Processing Using Flink & Kafka

David Hardwick
Sean Hester
David Breloch

<https://github.com/breloch/FlinkForward2017>

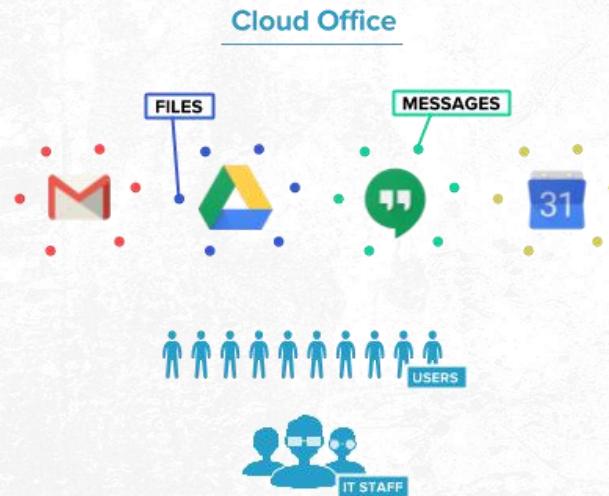
Multi-SaaS Management



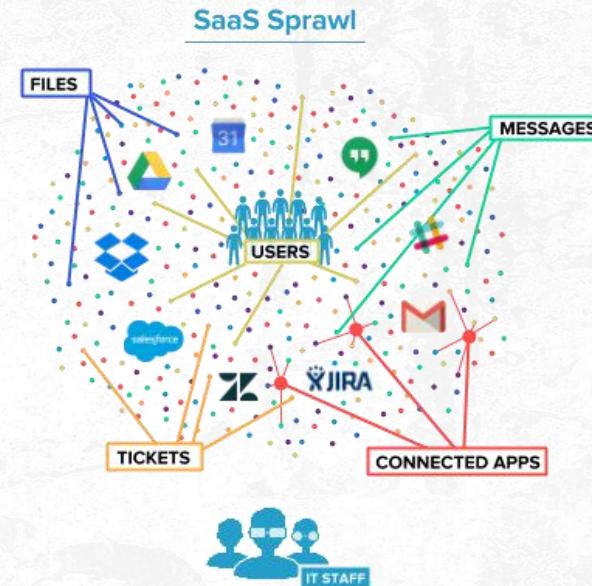
“What does that mean?”

Complex Interconnect of Apps, People and Data

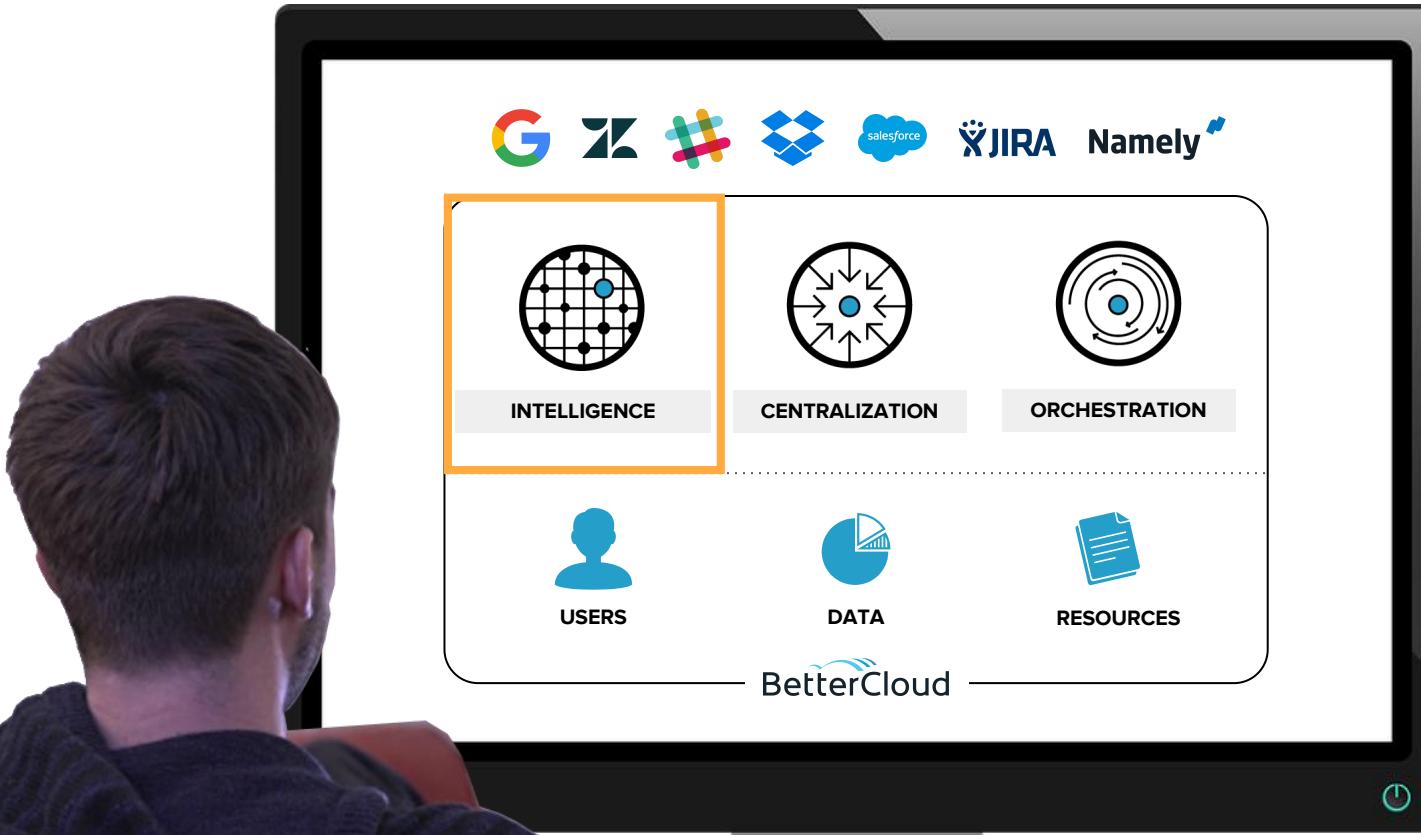
A Few Years Ago



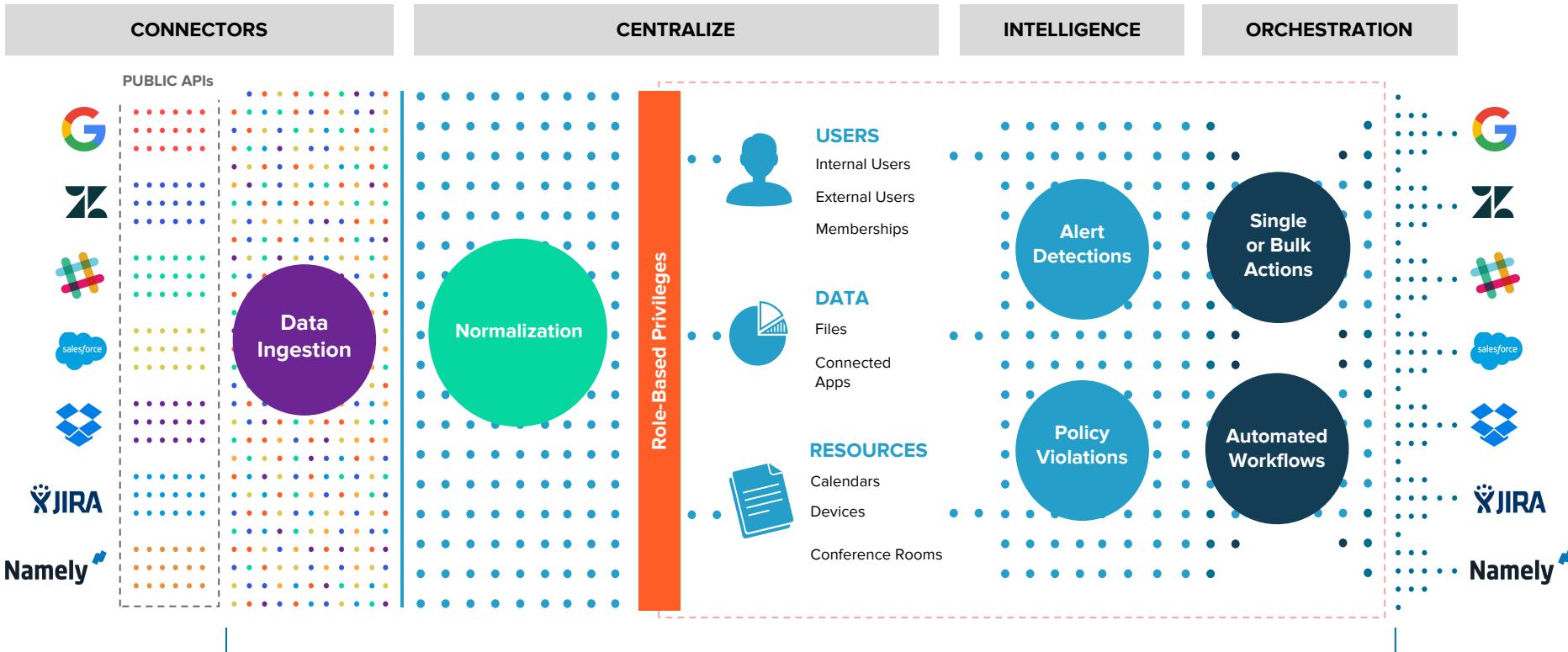
This is Now



“Multi-SaaS Management” is like Mission Control for IT Admins

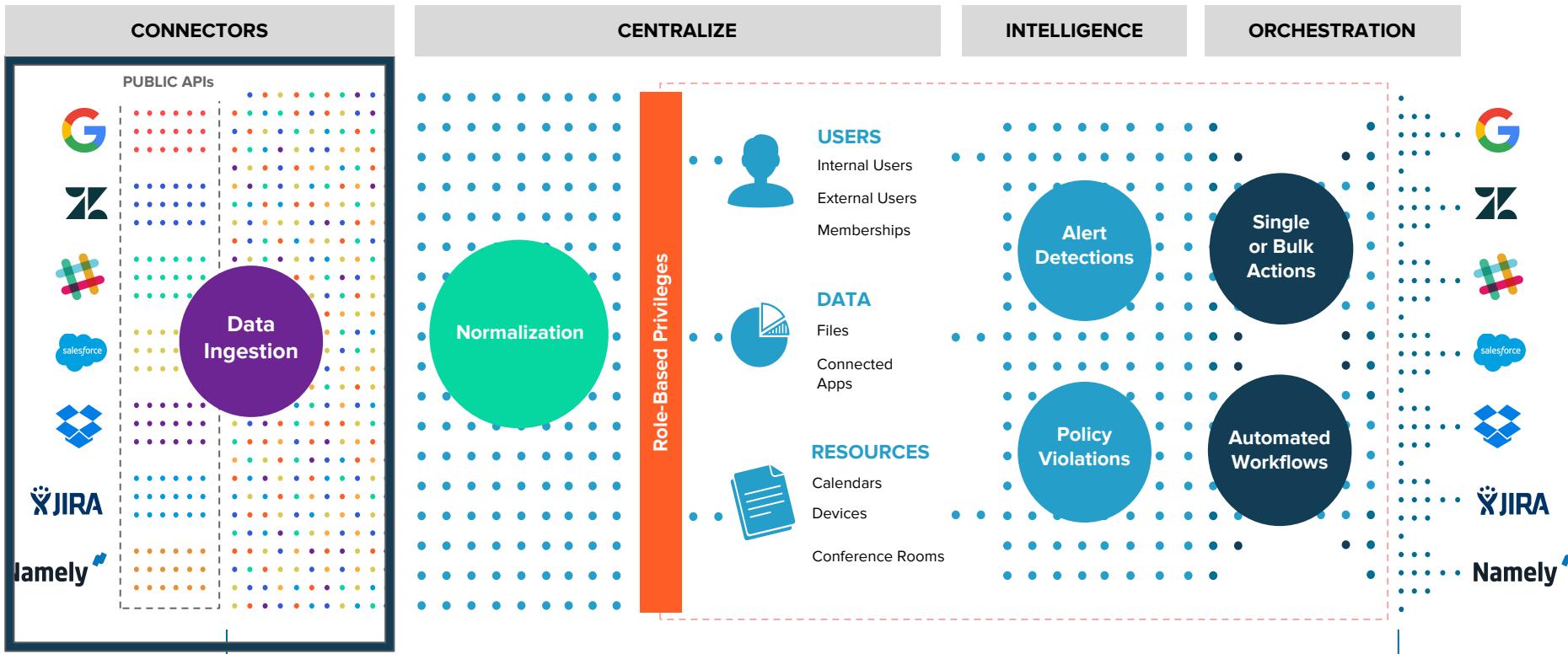


We process 100s millions of messages daily



Multi-SaaS Management

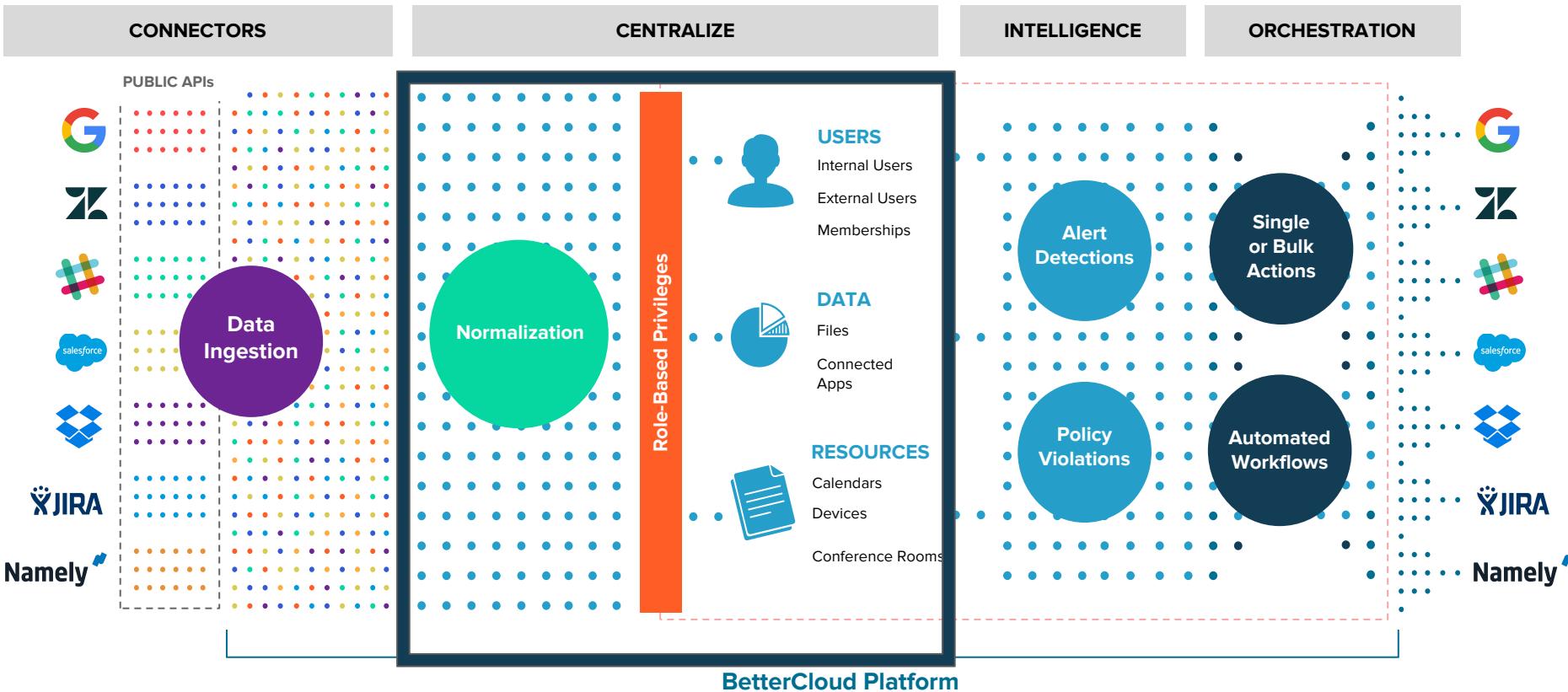
BetterCloud Platform



BetterCloud Platform

Multi-SaaS Management

BetterCloud Platform



USERS LIST



ALERTS



DIRECTORY



USERS



GROUPS



DATA MANAGEMENT



WORKFLOWS

Name	Email	Application	Status	Role
Albert (AJ) LeGaye	albert.legaye@demobettercloud.com	G	Active	Admin
Alex Miller	alex.miller@demobettercloud.com	G + Google Sheets Z	Active	Admin
Amanda McCarthy	amanda.mcCarthy@demobettercloud.com	G + Google Sheets Z	Active	Admin
Andrew McGonnigle	andrew.mcgonnigle@demobettercloud.com	G	Active	Admin
Ben Howard	ben.howard@demobettercloud.com	G + Google Sheets Z	Active	Admin
BetterCloud Notifications	notifications@demobettercloud.com	G	Active	End User
Bill Quinn	bill.quinn@demobettercloud.com	G	Active	Admin
Caitlin McDevitt	caitlin.mcdevitt@demobettercloud.com	G Z	Active	Admin
Caroline Thompson	caroline.thompson@demobettercloud.com	G + Google Sheets Z	Active	Admin
Chris Test	chris.test@demobettercloud.com	G	Active	End User
Christopher		G + Google Sheets		

< < 1 > >| View 25 ▾ per page.

Viewing 1 - 25 of 725

data centralization

Live Chat

ALERTS

DIRECTORY

USERS

GROUPS

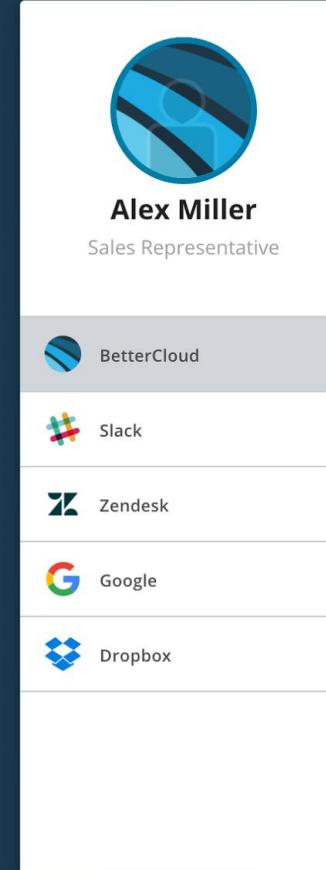
DATA MANAGEMENT

WORKFLOWS

CONNECTORS

PRIVILEGES

AUDIT LOGS



Summary

Memberships

Files

GROUP NAME	APPLICATION	ROLE	TYPE
admin_alerts		Member	group
ATL		—	group
bctest		—	group
BetterCloud Soccer Team		—	group
Customer List		—	group
Everyone at Demo BetterCloud		Member	group
general		Member	group
Human Resources		—	group

1 of 2 View 10 per page. Viewing 1 - 10 of 14

352 FILES

5 CALENDARS

4 INTEGRATIONS

32 GROUPS

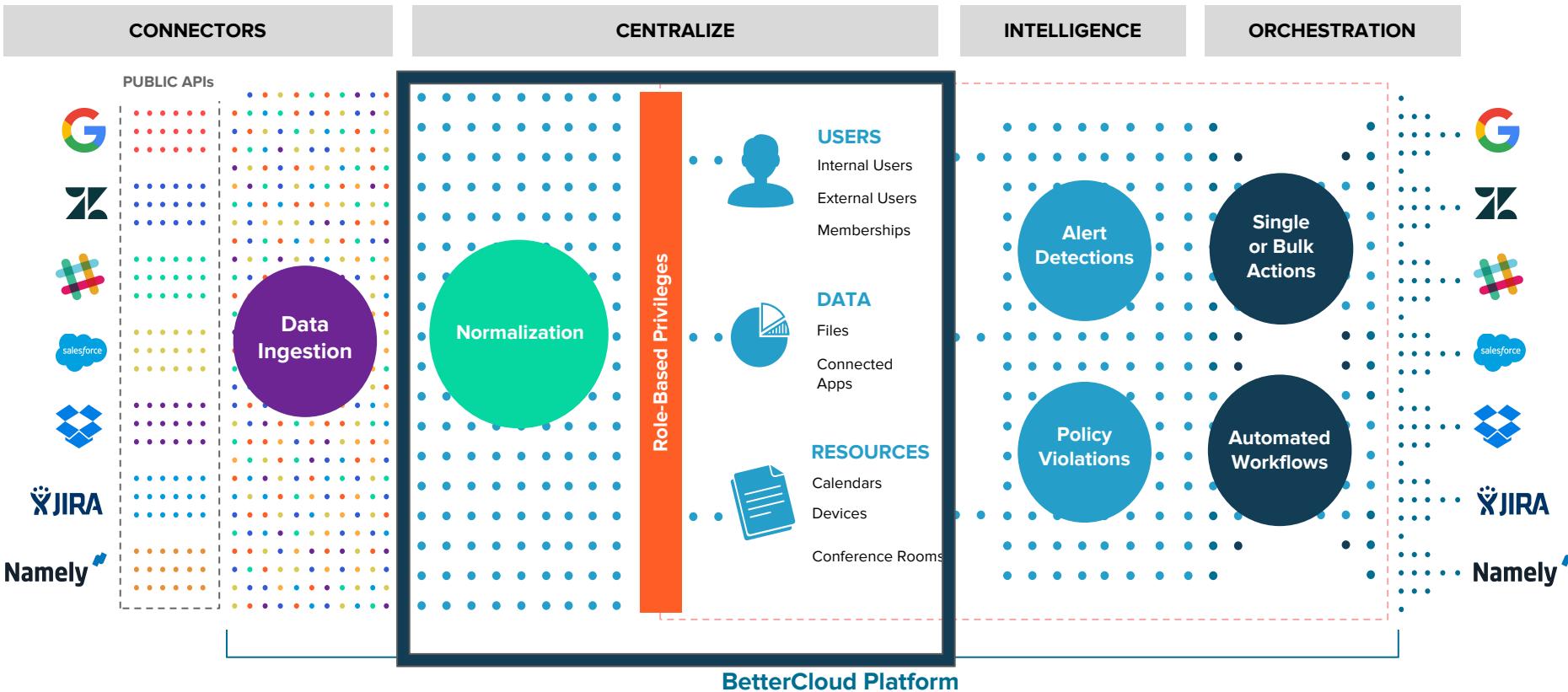
12% STORAGE USED

Live Chat

data centralization

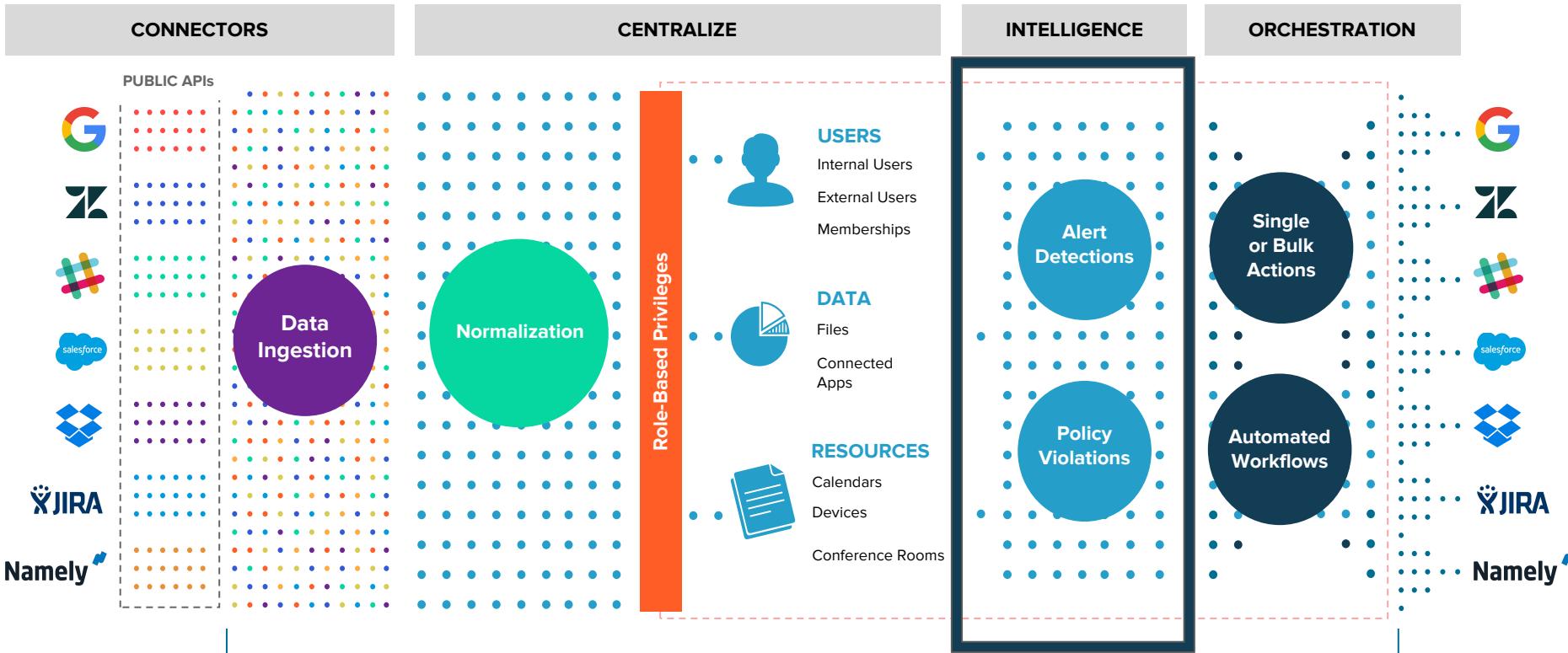
Multi-SaaS Management

BetterCloud Platform



Multi-SaaS Management

BetterCloud Platform



BetterCloud Platform

ALERTS

DIRECTORY

DATA MANAGEMENT

WORKFLOWS

CONNECTORS

PRIVILEGES

AUDIT LOGS

Users with No Depar

Empty Groups

Super Administrator

Super Administrator

Super Administrator

Users without Two-F

Administrator Count

Users with No Name

Users with No Title

Administrator Added

Users with No Phone

< < 1 of 1

Users without Two-Factor Authentication

Details

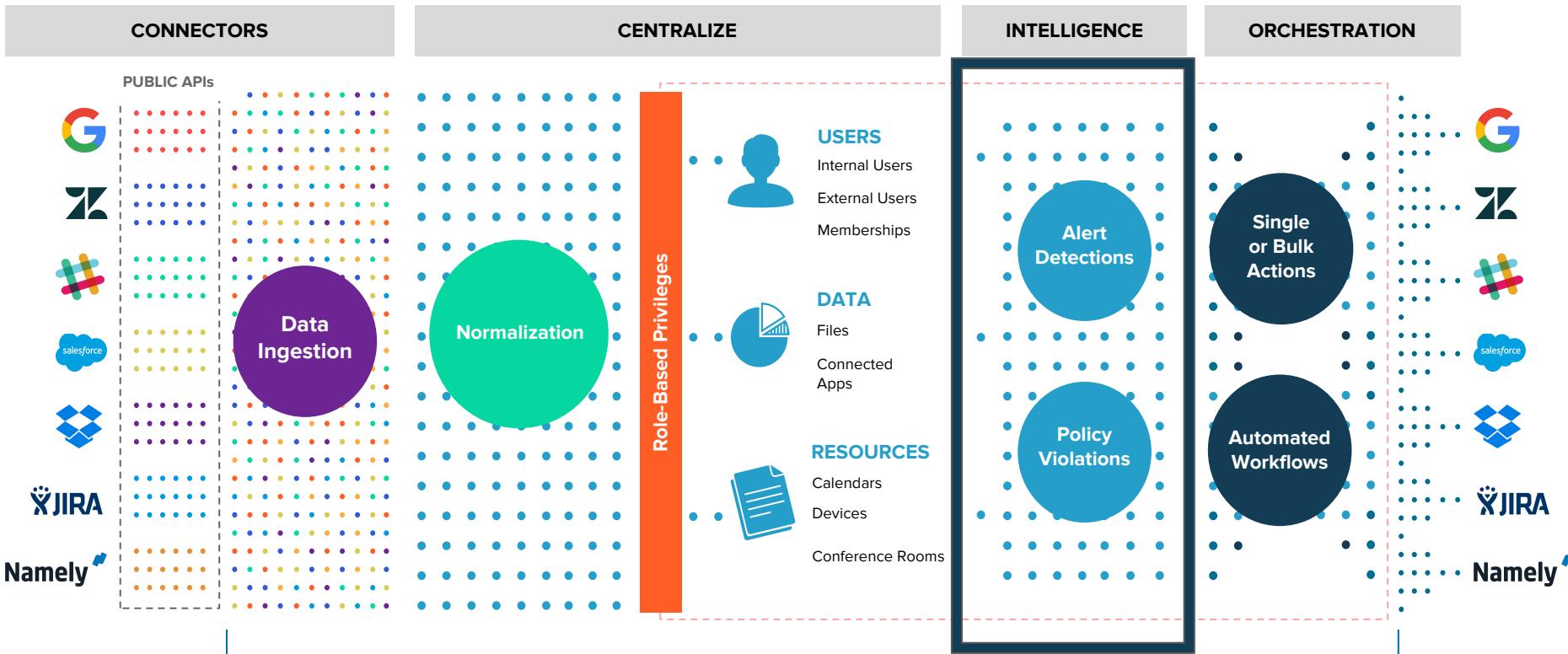
Severity:	Major
Application:	Slack
Date Occurred:	06/Feb/2017 10:35 PM
Description:	Users who do not have two-factor authentication set up
Threshold:	1
Count:	10

USER EMAIL	DATE/TIME TRIGGERED
michael.matmon@demobettercloud.com	06/Feb/2017 10:35 PM
caroline.thompson@demobettercloud.com	17/Jan/2017 5:40 PM
taylor.gould@demobettercloud.com	17/Dec/2016 6:29 PM

Live Chat

Multi-SaaS Management

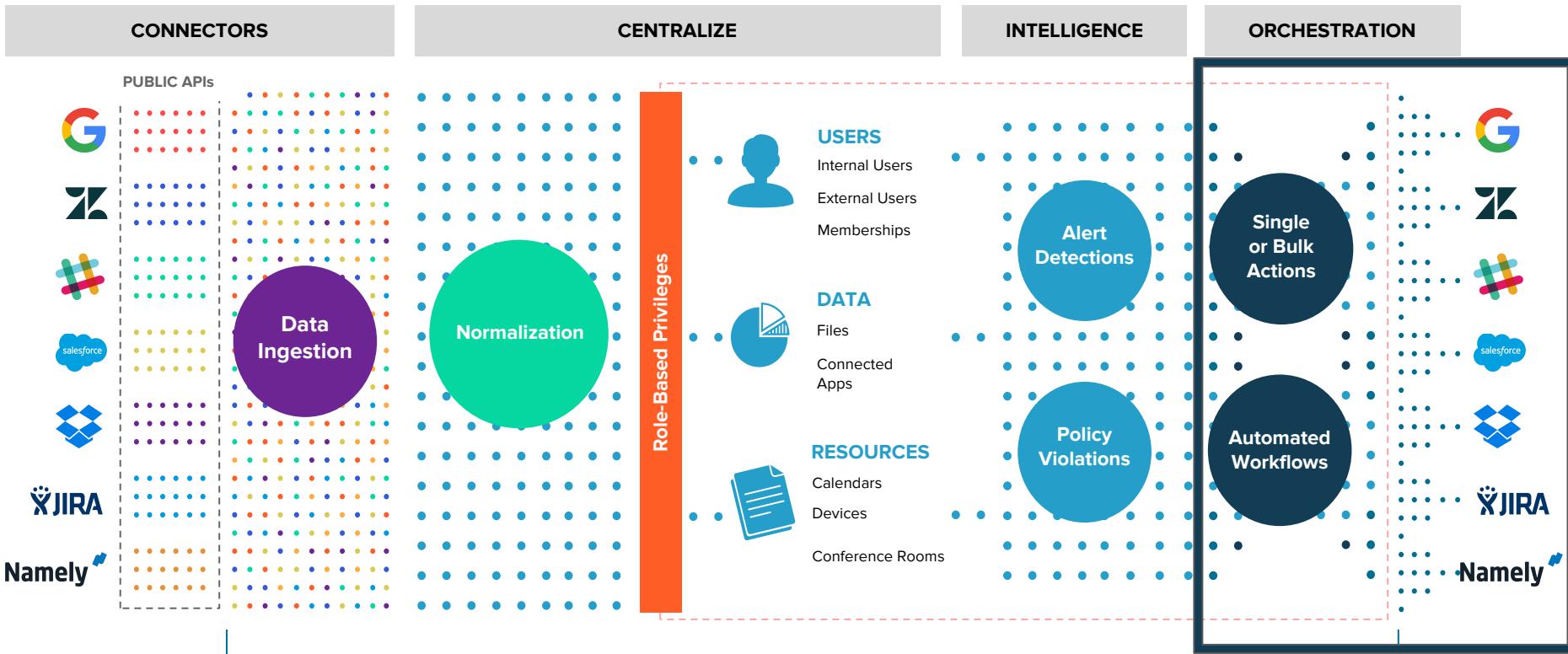
BetterCloud Platform



BetterCloud Platform

Multi-SaaS Management

BetterCloud Platform



BetterCloud Platform

5 Users

SELECT ACTION

ALERTS

DIRECTORY

USERS

GROUPS

DATA MANAGEMENT

WORKFLOWS

CONNECTORS

PRIVILEGES

AUDIT LOGS

	NAME ↑	EMAIL	APPLIANCE
<input type="checkbox"/>	asdfasdf3 asdfasdf	aa_asdfasdf3.asdfasdf-bb@0000ff.co	
<input checked="" type="checkbox"/>	brenda harden	brenda@0000ff.co	
<input checked="" type="checkbox"/>	Brenda Harden	brenda.harden@0000ff.co	
<input type="checkbox"/>	Brian Test	briantest@0000ff.co	
<input checked="" type="checkbox"/>	Christina Kang	christina.kang@0000ff.co	
<input checked="" type="checkbox"/>	David Hardwick	david@0000ff.co	
<input type="checkbox"/>	directory user	directory.user@0000ff.co	
<input checked="" type="checkbox"/>	Jason Lang	jason@0000ff.co	
<input type="checkbox"/>	john.mcintosh	john.mcintosh@0000ff.co	
<input type="checkbox"/>	Johnny McIntosh	mcintosh.john@0000ff.co	 Active
<input type="checkbox"/>	kevin willow	kevin.willow@0000ff.co	 Active

|< < 1 >> View 25 ▾ per page.

Hint: type a to quickly open and search this menu [close](#)

	Slack	Zendesk	Dropbox	Google
 Slack	Send Integration Logs			
 Slack	Set User's Phone Number			
 Zendesk	Send Direct Message			
 Dropbox	Remove from Channel			
 Google	Add To Channel			
	Create Reminder			
	Remove From User Group			
	Add To User Group			
	Set User's Title			
 Zendesk	Sign Out User			
	Reset Password			
	Suspend User			
	More Actions			

Viewing 1 - 23 of 23

 Submit a Support Ticket

action orchestration

Flink detects and evaluates

ACTIVE WORKFLOW
This workflow is active. Changes to this workflow will be reflected the next time it runs.

LIBRARY

WHEN IF THEN

ALL > Add To Channel +
Dropbox > G Add To Group +
Google > G Add To User Group +
Slack > G Archive Group +
Zendesk > G Assign License +
G Block Mobile Device +
G Change Role in Group +
G Copy Group Membership +

Add To Channel

This action adds a user to a public Slack channel.

1. Deprovisioning Users 23 / 50

THEN

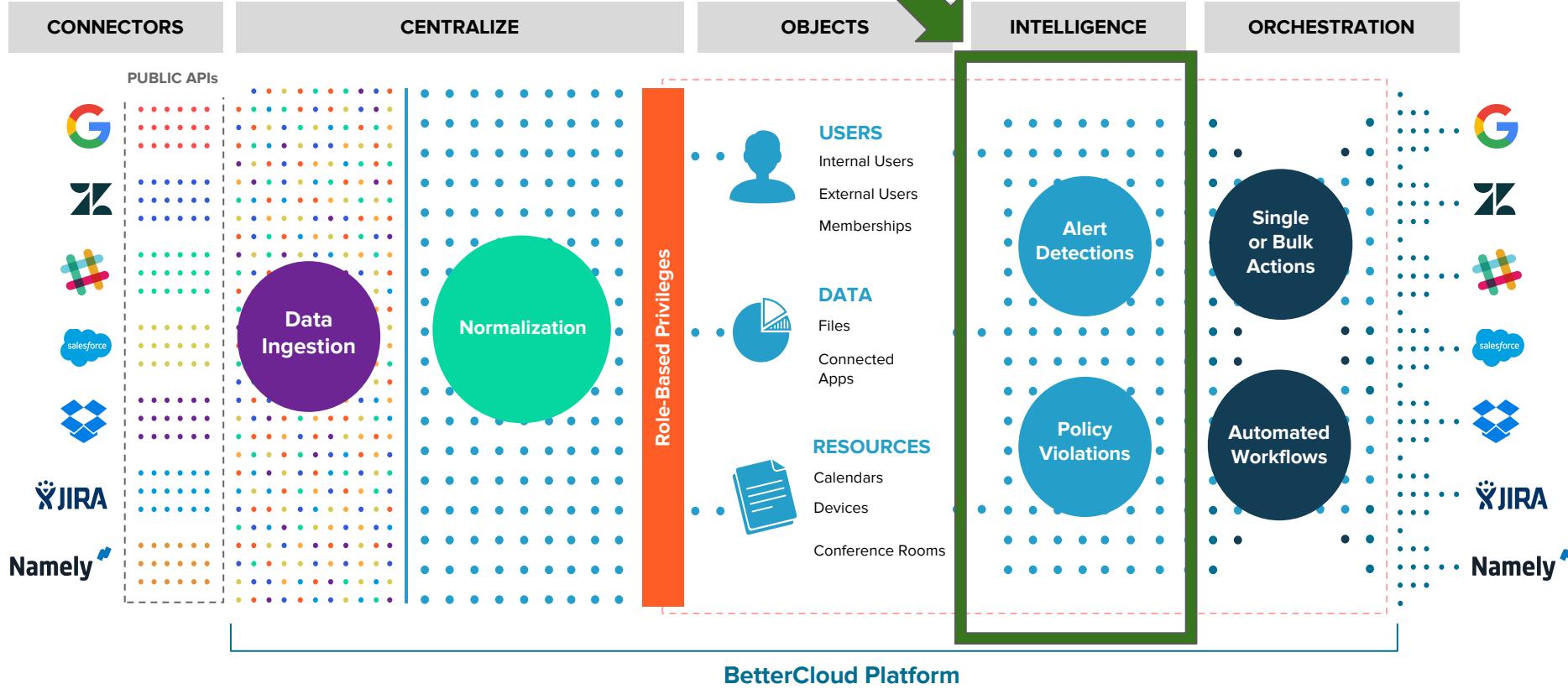
1. G HIDE user target user in directory
2. G Delete app-specific passwords for user target user
3. G Remove user target user from all groups group(s)
4. G Delete 2-step verification backup codes for user target user
5. G Transfer ownership of all files for target user to sharedinbox7@demobettercloud.com
6. G Suspend user target user
7. G Move user target user to Deprovision - Complete Org Unit

CANCEL SAVE

PROPERTIES

action orchestration

Heavy use of Flink here



Business Needs and Challenges

- Alert quickly < 2s latency from ingest to alert
- Scale tremendously ~100M events/day ...and growing
- Keep innovating daily deployments to production
- Show the power detect global alerts across all customers
- Let customers drive custom alerts for every customer
- Replay history “go back in time” for new alerts ...or alert bugs

“Okay, can you go deeper?”

Nope, but Sean and David can!

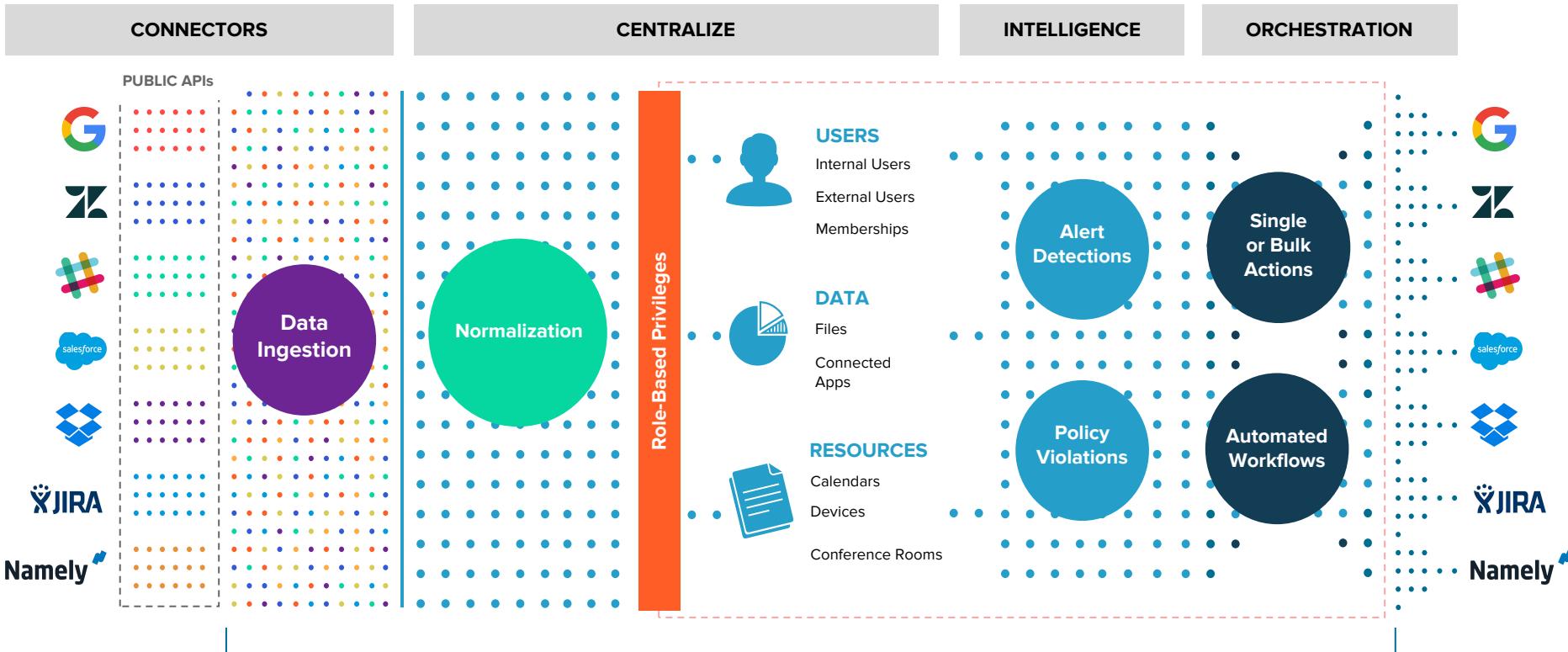
Foundational Prior Learnings

1. Event Stream Processing is a better model for Operational Intelligence than Ingest + Query

Foundational Prior Learnings

1. Event Stream Processing is a better model for Operational Intelligence than Ingest + Query
2. Rules Engines add significantly more value than hard-coded rules

We process 100s of millions of messages daily



Foundational Prior Learnings

1. Event Stream Processing is a better model for Operational Intelligence than Ingest + Query
2. Rules Engines add significantly more value than hard-coded rules

JsonPath

- Jayway JsonPath: <https://github.com/jayway/JsonPath>
- Executes queries again JSON documents
- For us, simplifies the creation of a rules interface for our non-technical users

JsonPath Continued

```
1  {  
2    "store": {  
3      "book": [  
4        {  
5          "category": "reference",  
6          "author": "Nigel Rees",  
7          "title": "Sayings of the Century",  
8          "price": 8.95  
9        },  
10       {  
11         "category": "fiction",  
12         "author": "Evelyn Waugh",  
13         "title": "Sword of Honour",  
14         "price": 12.99  
15       }]  
16     ],  
17     "bicycle": {  
18       "color": "red",  
19       "price": 19.95  
20     }  
21   },  
22   "expensive": 10  
23 }
```

Example Paths and Output:

`$.expensive => 10`

`$.store.bicycle.price => 19.95`

`$.store.book[0].author => "Nigel Rees"`

`$.store.book.length > 0 => true`

`$.store.book[?(@.category ==
'fiction')].price > 10.00 => { ... }`

JsonPath Wrapped in a User Interface

The screenshot shows the 'Edit Alert' interface in the BetterCloud platform. The main area is titled 'ALERT' and 'EDIT ALERT'. On the left sidebar, there are icons for Dropbox, Google, and Slack.

Alert Details:

- Title: Team Admin Added
- Description: Users that have had admin permissions added

Alert Trigger Conditions: This section is highlighted with a red box.

When a...	Data Point *	Object *	Event Type *
Dropbox	Is Active	User	Team Admin Added Events
AND	Manager	Operator *	Operand *
	equals	is empty	true

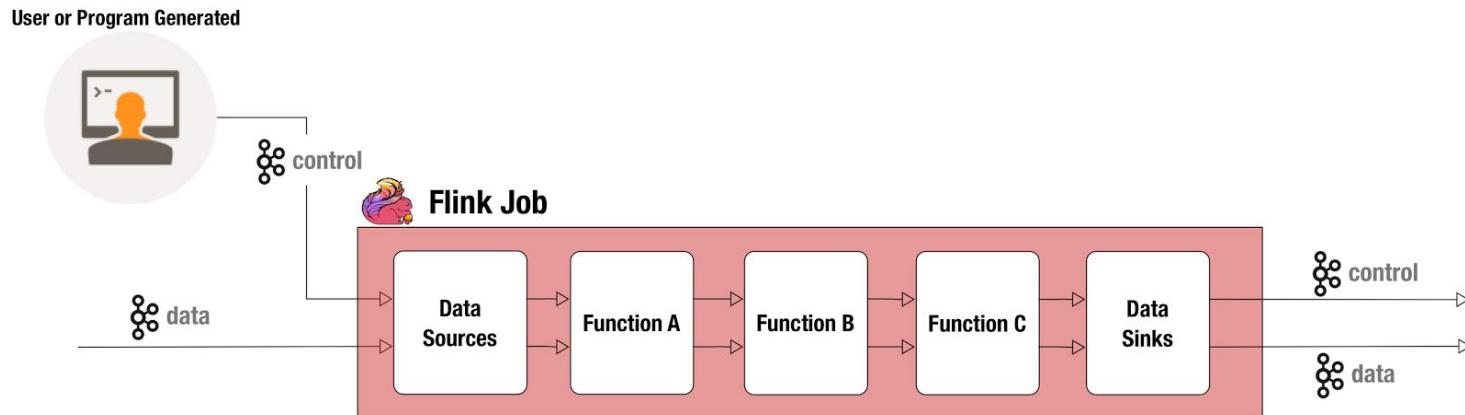
Timing & Thresholds:

Occurs...	Time Period	Operator	Threshold *
When total count Reached		Is Greater Than	1

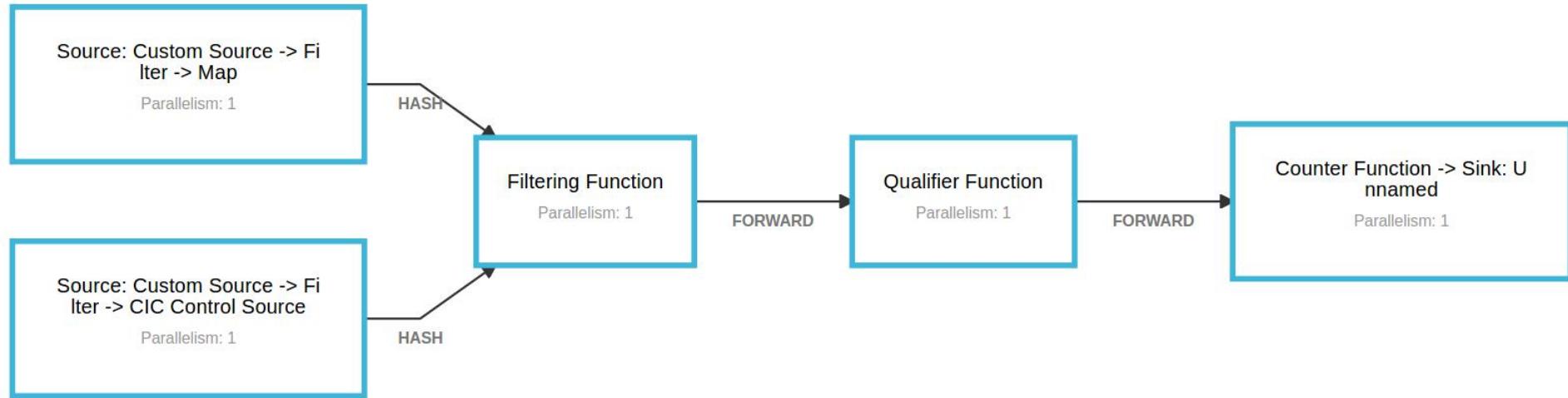
Buttons:

- ADD CONDITION

The Control Stream



The Solution



Models

```
case class CustomerEvent(customerId: UUID, payload: String)
```

```
case class ControlEvent(customerId: UUID, alertId: UUID, alertName: String, alertDescription: String, threshold: Int, jsonPath: String, bootstrapCustomerId: UUID)
```

Sources - Event Stream

```
val eventStream = env.addSource(new FlinkKafkaConsumer09("events", new CustomerEventSchema(),  
properties)  
.filter(x => x.isDefined)  
.map(x => x.get)  
.keyBy((ce: CustomerEvent) => { ce.customerId } )
```

Sources - Control Stream

```
val controlStream = env.addSource(new FlinkKafkaConsumer09("controls", new ControlEventSchema(),  
properties))  
    .filter(x => x.isDefined)  
    .map(x => x.get)  
    .name("Control Source")  
    .split((ce: ControlEvent) => {  
        ce.customerId match {  
            case Constants.GLOBAL_CUSTOMER_ID => List("global")  
            case _ => List("specific")  
        }  
    })
```

Sources - Control Stream Continued

```
val globalControlStream = controlStream.select("global").broadcast
```

```
val specificControlStream = controlStream.select("specific")
.keyBy((ce: ControlEvent) => { ce.customerId })
```

Sources - Join the Streams

```
// Join the control and event streams  
val filterStream = globalControlStream.union(specificControlStream)  
.connect(  
    eventStream  
)
```

Work on Streams - Filtering

```
class FilterFunction() extends RichCoFlatMapFunction[ControlEvent, CustomerEvent, FilteredEvent] {  
    var configs = new mutable.ListBuffer[ControlEvent]()  
  
    override def flatMap1(value: ControlEvent, out: Collector[FilteredEvent]): Unit = {  
        configs = configs.filter(x => (x.customerId != value.customerId) && (x.alertId != value.alertId))  
        configs.append(value)  
    }  
  
    override def flatMap2(value: CustomerEvent, out: Collector[FilteredEvent]): Unit = {  
        val eventConfigs = configs.filter(x => (x.customerId == value.customerId) || (x.customerId ==  
Constants.GLOBAL_CUSTOMER_ID))  
  
        if (eventConfigs.size > 0) {  
            out.collect(filteredEvent(value, eventConfigs.toList))  
        }  
    }  
}
```

Work on Streams - Qualifying

```
class QualifierFunction extends FlatMapFunction[FilteredEvent, QualifiedEvent] {  
    override def flatMap(value: FilteredEvent, out: Collector[QualifiedEvent]): Unit = {  
        Try(JsonPath.parse(value.event.payload)).map(ctx => {  
            value.controls.foreach(control => {  
                Try {  
                    val result: String = ctx.read(control.jsonPath)  
                    if (!result.isEmpty) {  
                        out.collect(QualifiedEvent(value.event, control))  
                    }  
                }  
            })  
        })  
    }  
}
```

Work on Streams - Counting

```
class CounterFunction extends FlatMapFunction[QualifiedEvent, ControlEvent] {  
  var counts = scala.collection.mutable.HashMap[String, Int]()  
  
  override def flatMap(value: QualifiedEvent, out: Collector[ControlEvent]): Unit = {  
    val key = s"${value.event.customerId}${value.control.alertId}"  
    if (counts.contains(key)) {  
      counts.put(key, counts.get(key).get + 1)  
      println(s"Count for ${key}: ${counts.get(key).get}")  
    } else {  
      counts.put(key, 1)  
      println(s"Count for ${key}: ${counts.get(key).get}")  
    }  
  }  
}
```

Wiring It Up

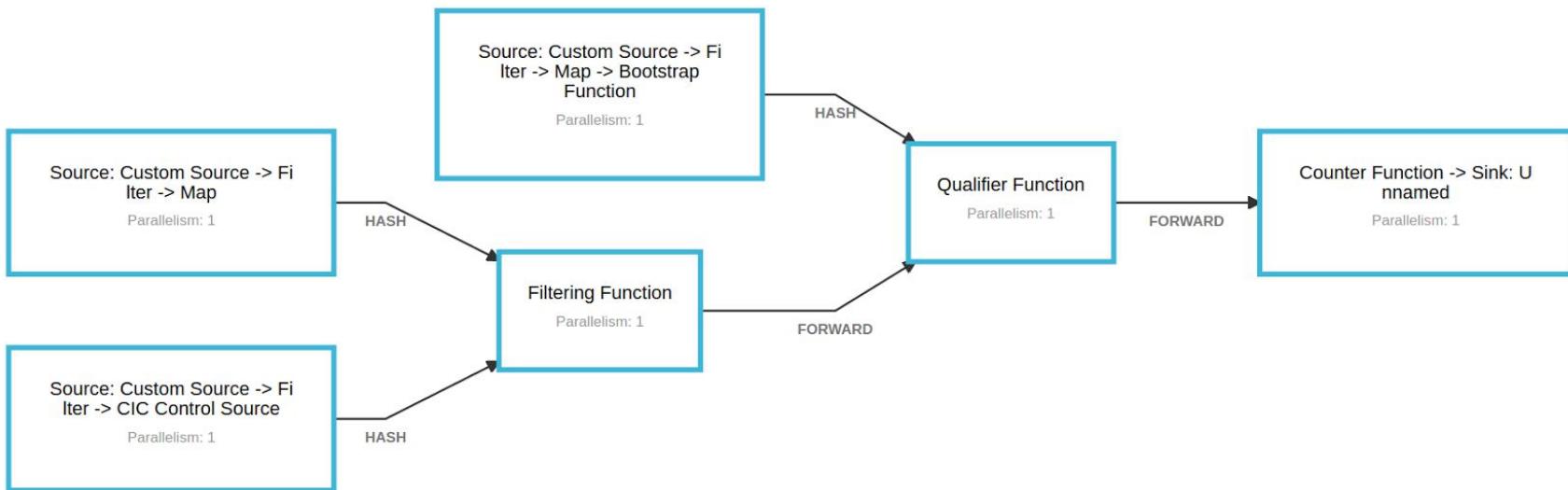
```
val filterFunction = new FilterFunction
val qualifierFunction = new QualifierFunction
val counterFunction = new CounterFunction

// Join the control and event streams
val filterStream = globalControlStream.union(specificControlStream)
.connect(
  eventStream
)
.flatMap(filterFunction).name("Filtering Function")
.flatMap(qualifierFunction).name("Qualifier Function")
.flatMap(counterFunction).name("Counter Function")
```

What about historical data?

- The current solution works great against the live stream of data, but...
how do you get to the current number when the events we need have already gone through the system?

Basic Implementation



Counter Function Updated

```
override def flatMap(value: QualifiedEvent, out: Collector[ControlEvent]): Unit = {
  val key = s"${value.event.customerId}${value.control.alertId}"
  if (counts.contains(key)) {
    counts.put(key, counts.get(key).get + 1)
    println(s"Count for ${key}: ${counts.get(key).get}")
  } else {
    val c = value.control
    counts.put(key, 1)
    out.collect(ControlEvent(c.customerId, c.alertId, c.alertName, c.alertDescription,
c.threshold, c.jsonPath, value.event.customerId))
    println(s"Bootstrap count for ${key}: ${counts.get(key).get}")
  }
}
```

Bootstrap Function

(“Go back in time”)

```
class BootstrapFunction extends FlatMapFunction[ControlEvent, FilteredEvent] {  
    override def flatMap(value: ControlEvent, out: Collector[FilteredEvent]): Unit = {  
        val stream = getClass.getResourceAsStream("/events.txt")  
  
        Source.fromInputStream(stream)  
            .getLines  
            .toList  
            .map(x => CustomerEvent(x))  
            .filter(x => x.customerId == value.bootstrapCustomerId)  
            .foreach(x => {  
                out.collect(FilteredEvent(x, List(value)))  
            })  
    }  
}
```

Wire up bootstrap

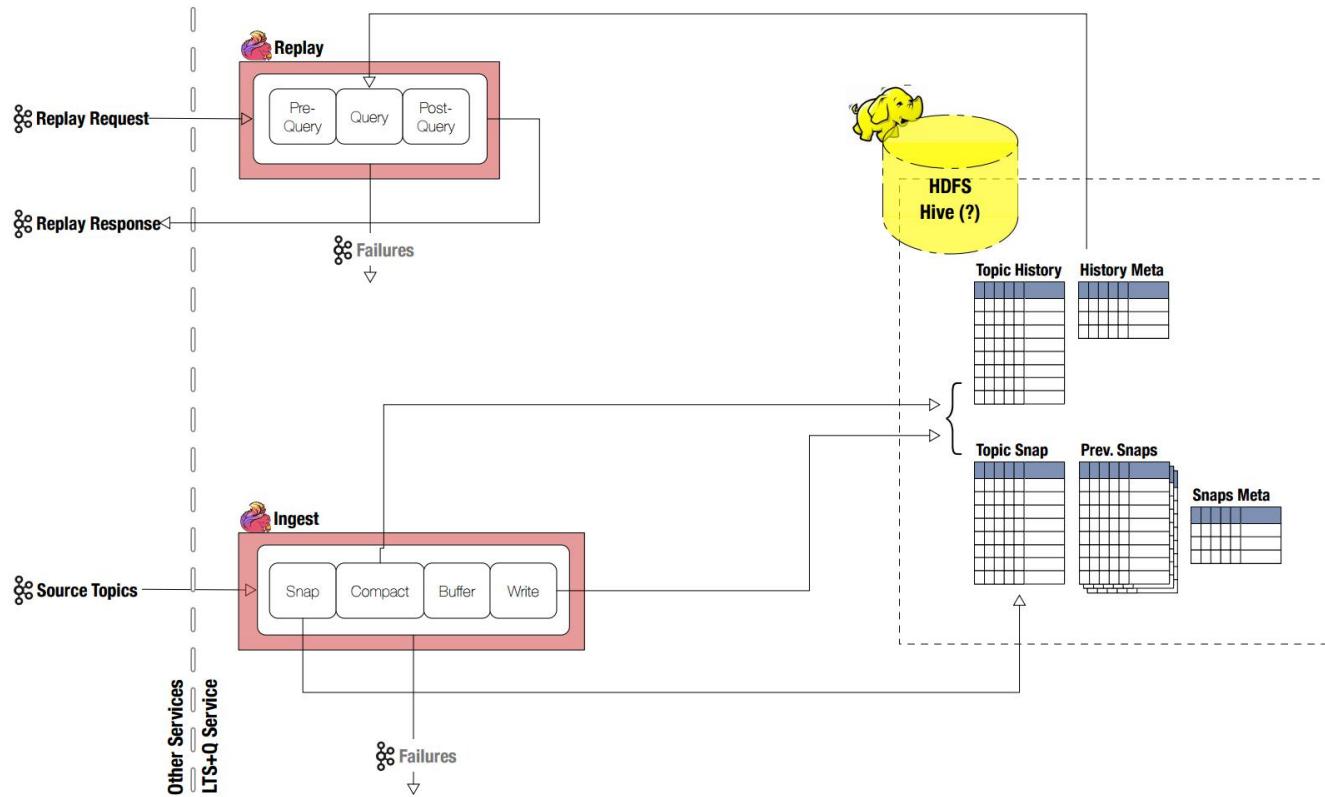
(“Go back in time”)

```
val bootstrapStream = env.addSource(new FlinkKafkaConsumer09("bootstrap", new ControlEventSchema(),  
properties)  
.filter(x => x.isDefined).map(x => x.get).flatMap(bootstrapFunction).name("Bootstrap Function")  
.keyBy((fe: FilteredEvent) => { fe.event.customerId } )  
  
val bootstrapSink = new FlinkKafkaProducer09("bootstrap", new ControlEventSchema(), properties)  
  
val filterStream = globalControlStream.union(specificControlStream)  
.connect(eventStream)  
.flatMap(filterFunction).name("Filtering Function")  
.union(bootstrapStream)  
.flatMap(qualifierFunction).name("Qualifier Function")  
.flatMap(counterFunction).name("Counter Function")  
.addSink(bootstrapSink)
```

- Extremely simplified compared to what you would want in production
 - A real system requires both ingestion of all events and some kind of query system (API, database, kafka, etc...)
 - Be careful with how you implement the query system. It is easy to block the whole stream with an API or database (Async I/O!!!!).
- There should be some kind of tracking on the event which triggered to bootstrap
 - You will need to get all events up until that event from the storage system
- If ordering does not matter:
 - Can just replay all previous events as in the example code

- If ordering does matter:
 - You will need to collect and block all incoming events for that specific key until the replay of old events completes

Our Solution For Historical Data - Long Term Storage



TO THE DEMO!



BetterCloud

Further Areas for Improvement

- Most expensive operations for us are JsonPath operations. It may be worth moving towards broadcasting all alerts instead of just global and move the keyed stream to as late as possible (we are switching to this)
- State is not being saved. At a minimum the controls and counts should be saved in state
- The current models only support incrementing
- JsonPath supports different types so you can have more operators than equals (>, <, etc...)
- Only a single property path is supported

We are hiring (go figure, right?)!!!

Seriously, we are always hiring. If you are interested talk to David Hardwick or visit
<https://www.bettercloud.com/careers/>

Questions?

<https://github.com/brelloch/FlinkForward2017>





Experiences in running Apache Flink® at large scale

Stephan Ewen (@StephanEwen)

dataArtisans



Lessons learned from running Flink at large scale

Including various things we never expected to become a problem and evidently still did...

Also a preview to various fixes coming in Flink...



What is large scale?

Large Data Volume
(events / sec)

Large Application State
(GBs / TBs)

Complex Dataflow
Graphs
(many operators)

High Parallelism
(1000s of subtasks)



Distributed Coordination

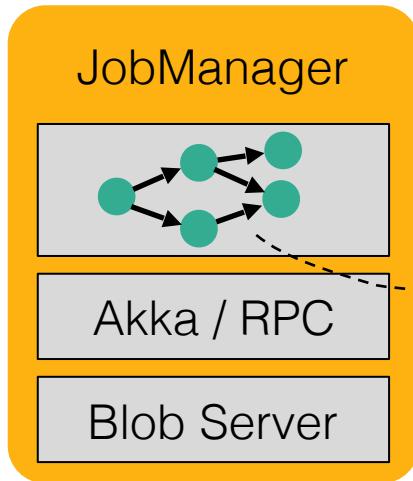


Deploying Tasks

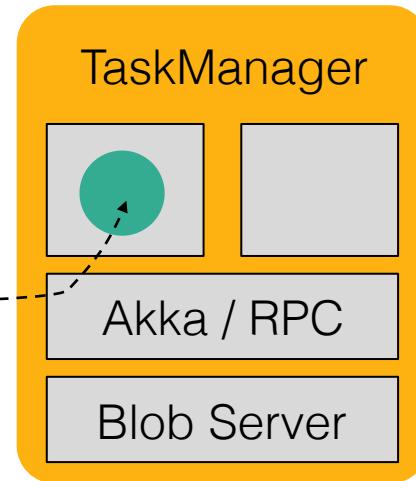
Happens during initial deployment and recovery

Contains

- Job Configuration
- Task Code and Objects
- Recover State Handle
- Correlation IDs



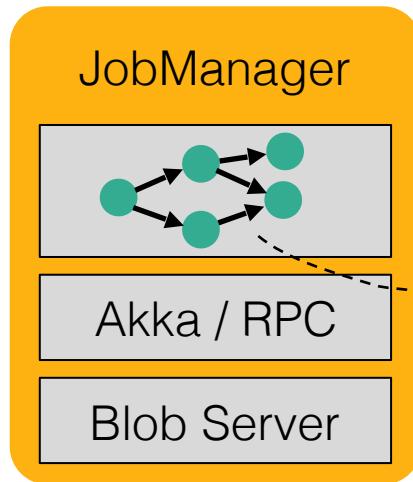
Deployment RPC Call





Deploying Tasks

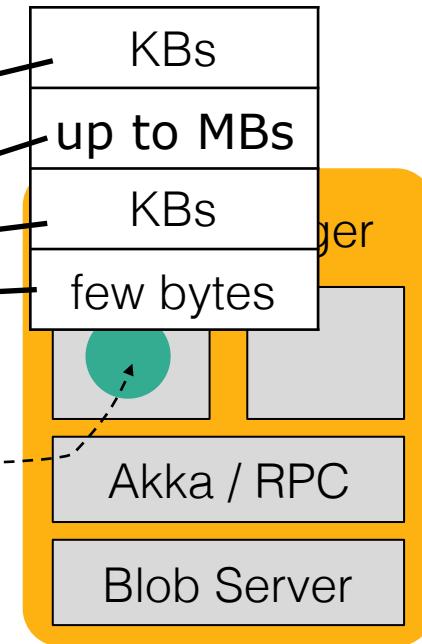
Happens during initial deployment and recovery



Contains

- Job Configuration
- Task Code and Objects
- Recover State Handle
- Correlation IDs

Deployment RPC Call





RPC volume during deployment

(back of the napkin calculation)

$$\begin{array}{l} \text{number of tasks} \times \text{parallelism} \times \text{size of task objects} = \text{RPC volume} \\ \\ 10 \times 1000 \times 2 \text{ MB} = 20 \text{ GB} \end{array}$$

~20 seconds on full 10 GBits/s net

> 1 min with avg. of 3 GBits/s net

> 3 min with avg. of 1GBs net



Timeouts and Failure detection

~20 seconds on full 10 GBits/s net

> 1 min with avg. of 3 GBits/s net

> 3 min with avg. of 1GBs net

Default RPC timeout: **10 secs**

default settings lead to **failed deployments with RPC timeouts**

Solution: Increase RPC timeout

Caveat: Increasing the timeout makes failure detection slower

Future: Reduce RPC load (next slides)



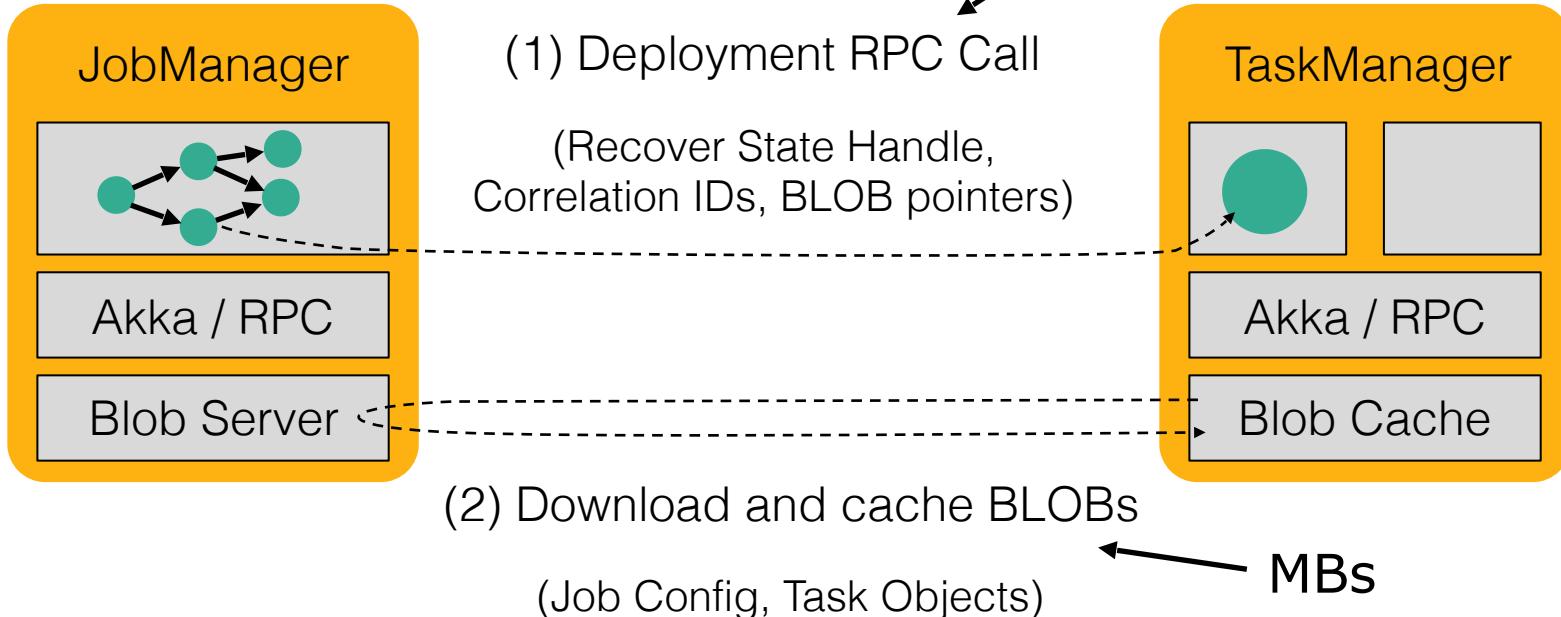
Dissecting the RPC messages

Message part	Size	Variance across subtasks and redeploys
Job Configuration	KBs	constant
Task Code and Objects	up to MBs	constant
Recover State Handle	KBs	variable
Correlation IDs	few bytes	variable



Upcoming: Deploying Tasks

Out-of-band transfer and caching of large and constant message parts





Checkpoints at scale



Robustly checkpointing...

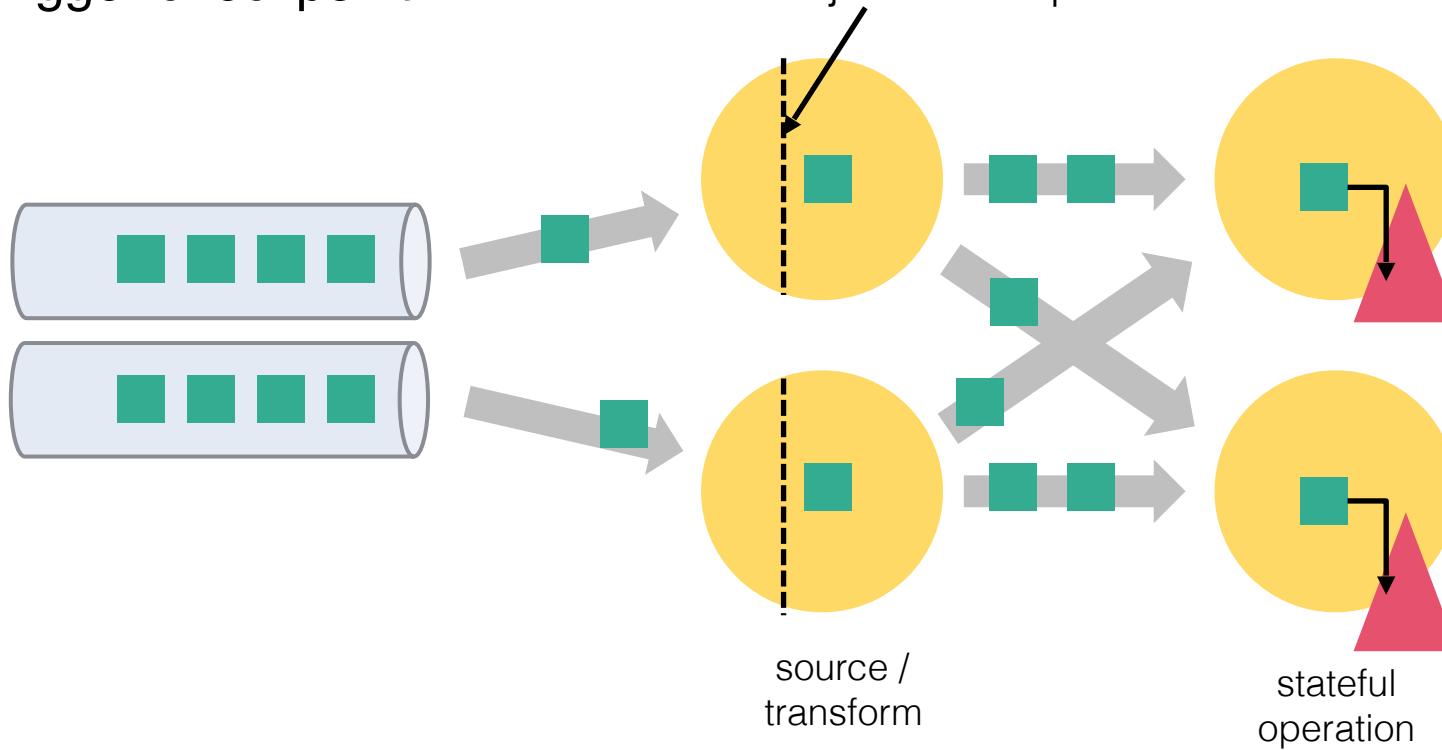
...is the most important part of
running a large Flink program



Review: Checkpoints

Trigger checkpoint

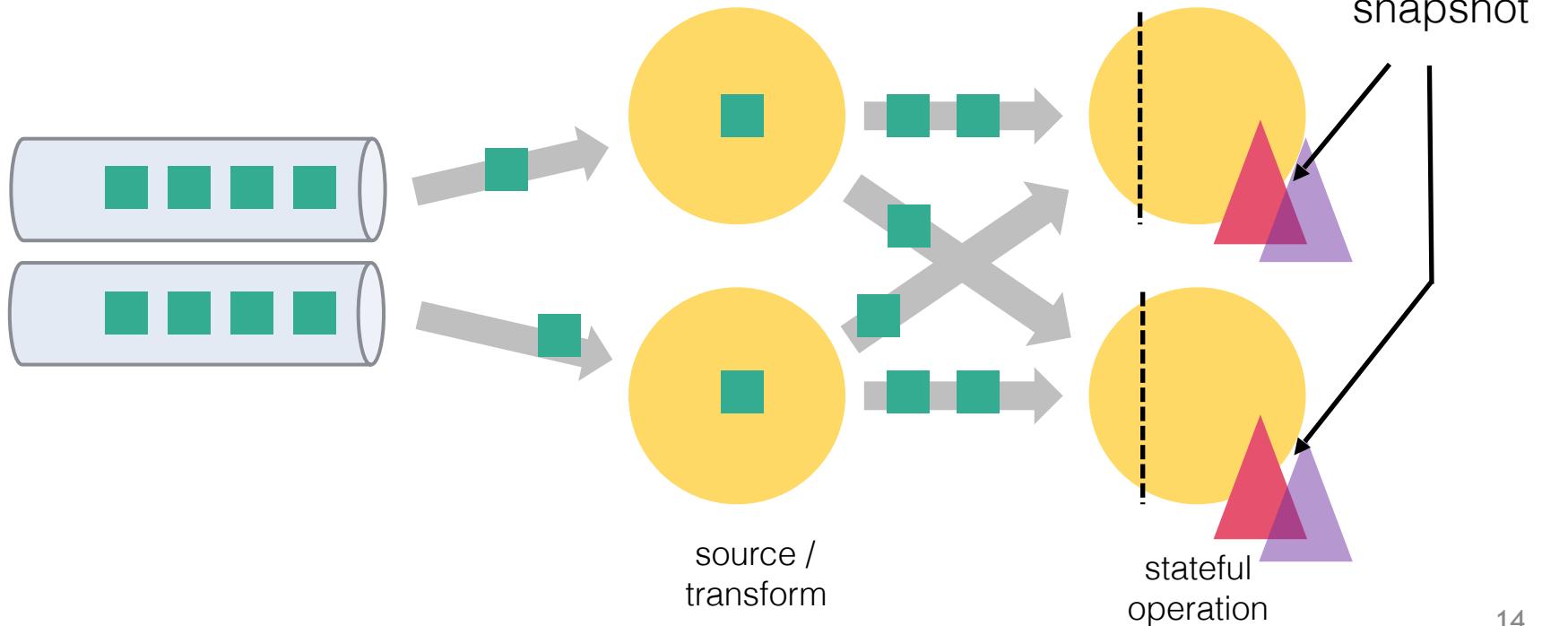
Inject checkpoint barrier





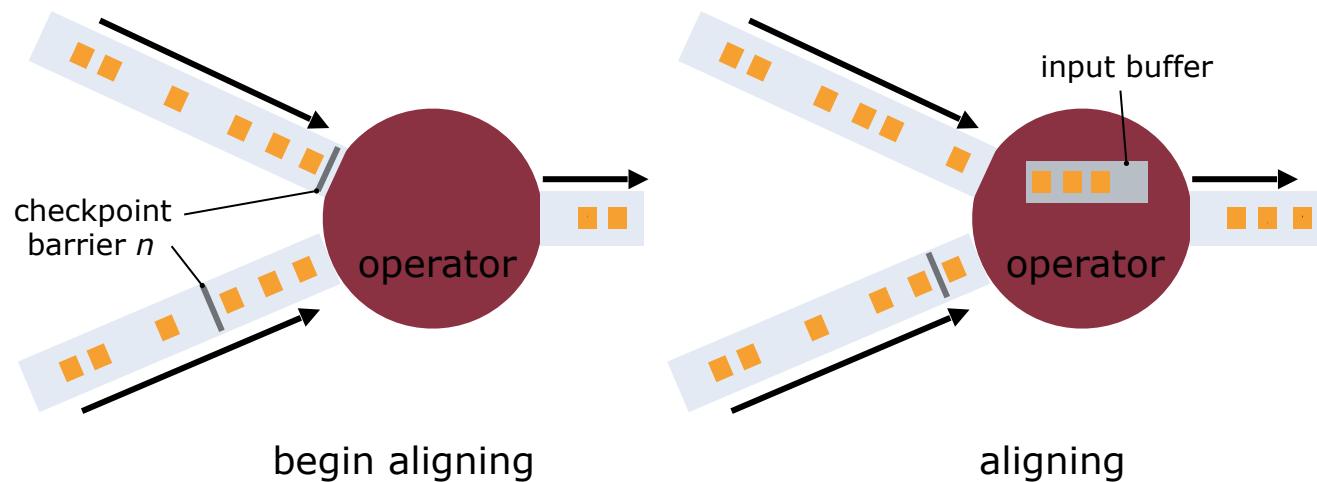
Review: Checkpoints

Take state snapshot



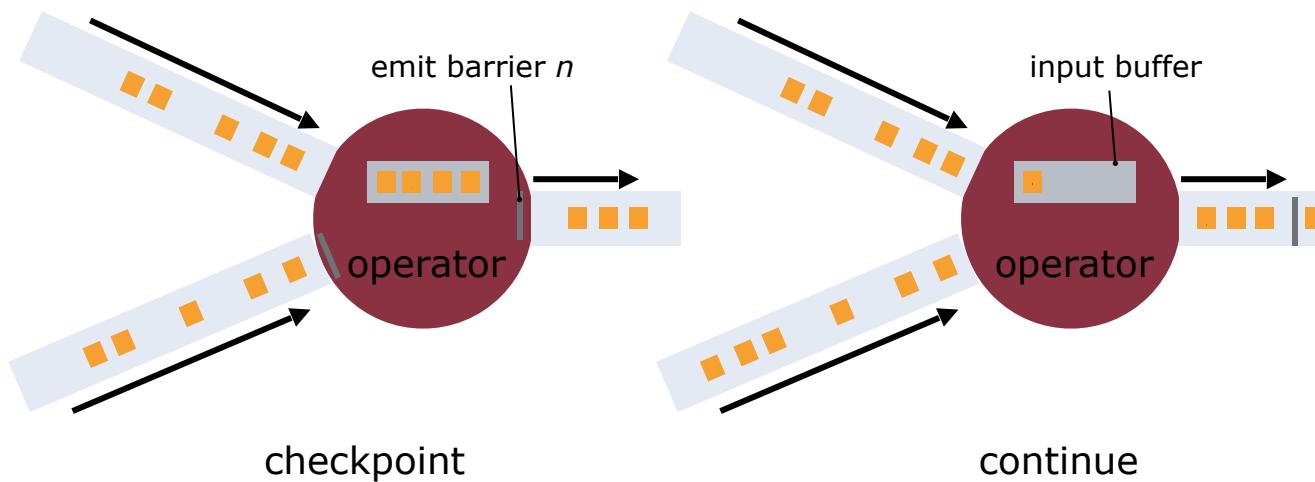


Review: Checkpoint Alignment





Review: Checkpoint Alignment





Understanding Checkpoints

Subtasks	TaskManagers	Metrics	Accumulators	Checkpoints	Back Pressure			
Overview	History	Summary	Configuration	Details for Checkpoint 4				
ID	Status	Acknowledged	Trigger Time	Latest Acknowledgment	End to End Duration	State Size	Buffered During Alignment	Discarded
4	✓ Completed	8/8 (100%)	15:42:26	15:42:26	15ms	96.2 KB	26.7 KB	No
Operators								
Name	Acknowledged	Latest Acknowledgment	End to End Duration	State Size	Buffered During Alignment			
Source: Custom Source	4/4 (100%)	15:42:26	14ms	48.5 KB	0 B	Show Subtasks ▾		
Flat Map -> Sink: Unnamed	4/4 (100%)	15:42:26	15ms	47.7 KB	26.7 KB	Show Subtasks ▾		



Understanding Checkpoints

delay =
end_to_end – sync – async

How long do
snapshots take?

How well behaves
the alignment?
(lower is better)

Source: Custom Source 4/4 (100%) 15:42:26 14ms 48.5 KB 0 B Hide Subtasks ▾						
	End to End Duration	State Size	Checkpoint Duration (Sync)	Checkpoint Duration (Async)	Alignment Buffered	Alignment Duration
Minimum	8ms	11.9 KB	0ms	0ms	0 B	0ms
Average	10ms	12.1 KB	0ms	0ms	0 B	0ms
Maximum	14ms	12.3 KB	0ms	1ms	0 B	0ms



Understanding Checkpoints

delay =
end_to_end – sync – async

How long do
snapshots take?

How well behaves
the alignment?
(lower is better)

Source: Custom Source 4/4 (100%) 15:42:26
long delay = under backpressure

under constant backpressure
means the application is
under provisioned

Average	10ms	State Size	1.9 KB
Maximum	14ms	12.1 KB	0ms

too long means
→ too much state
per node
→ snapshot store cannot
keep up with load
(low bandwidth)
changes with incremental
checkpoints

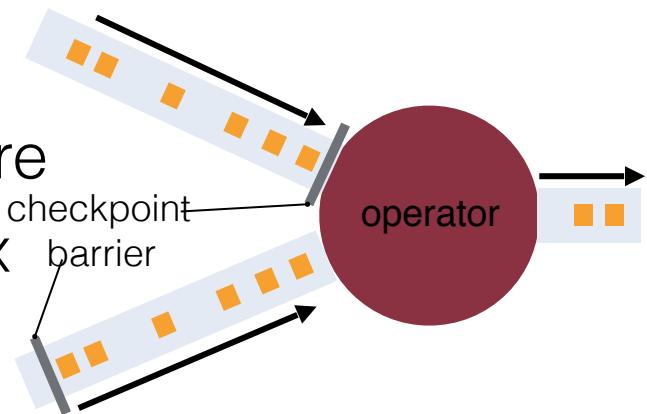
most important
metric

Checkpoint Duration (Sync)	Checkpoint Duration (Async)	Alignment Buffered	Alignment Duration
0ms	1ms	0 B	0ms



Alignments: Limit in-flight data

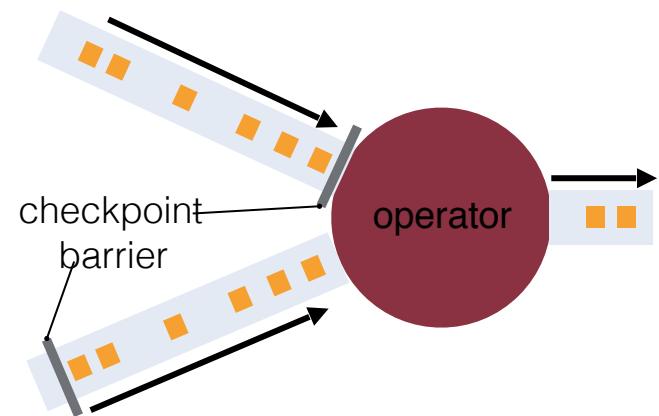
- In-flight data is data "between" operators
 - On the wire or in the network buffers
 - Amount depends mainly on network buffer memory
- Need some to buffer out network fluctuations / transient backpressure
- Max amount of in-flight data is max amount buffered during alignment





Alignments: Limit in-flight data

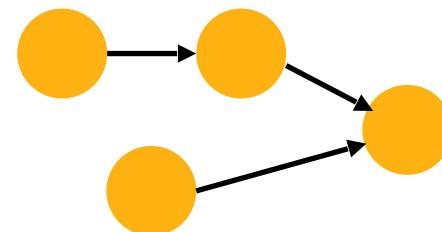
- Flink 1.2: Global pool that distributes across all tasks
 - Rule-of-thumb: set to $4 * \text{num_shuffles} * \text{parallelism} * \text{num_slots}$
- Flink 1.3: Limits the max in-flight data automatically
 - Heuristic based on channels and connections involved in a transfer step





Heavy alignments

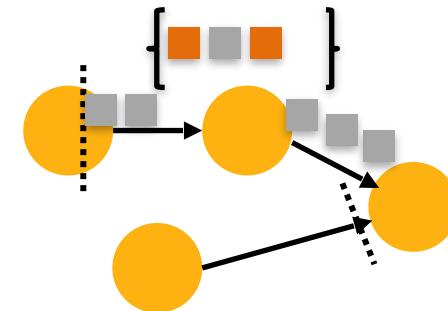
- A heavy alignment typically happens at some point
→ Different load on different paths
- Big window emission concurrent to a checkpoint
- Stall of one operator on the path





Heavy alignments

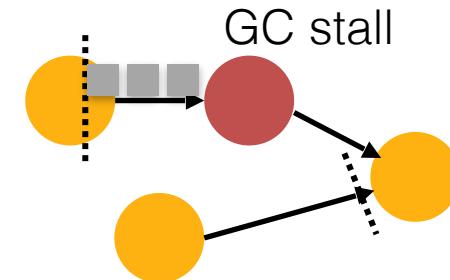
- A heavy alignment typically happens at some point
→ Different load on different paths
- Big window emission concurrent to a checkpoint
- Stall of one operator on the path





Heavy alignments

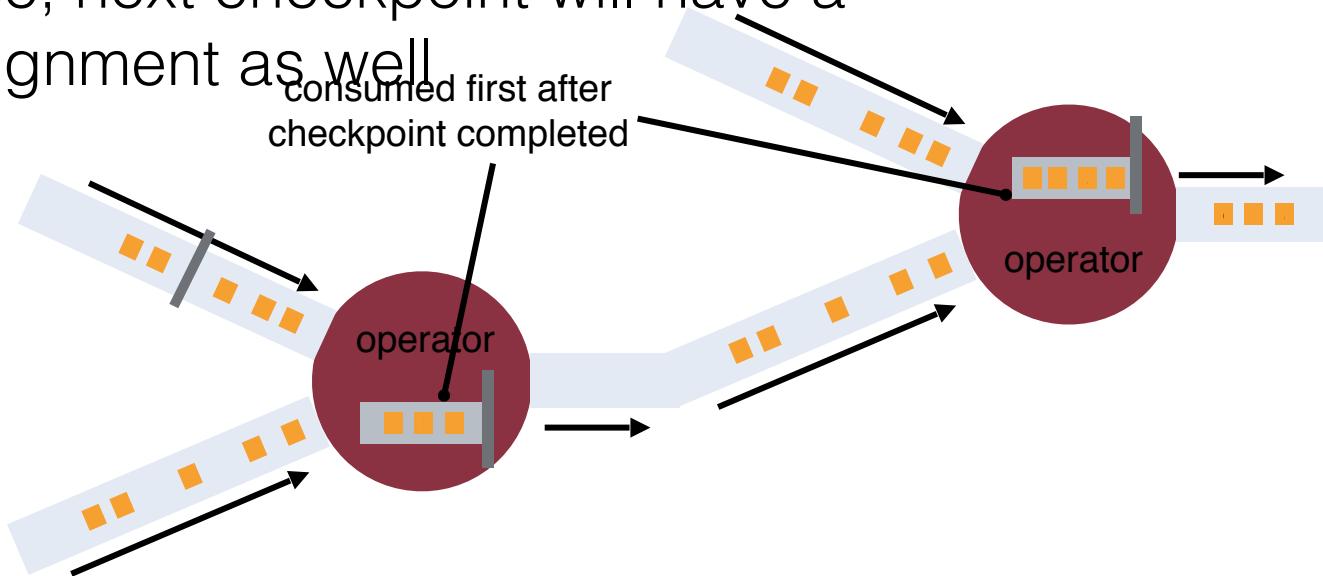
- A heavy alignment typically happens at some point
→ Different load on different paths
- Big window emission concurrent to a checkpoint
- **Stall of one operator on the path**





Catching up from heavy alignments

- Operators that did heavy alignment need to catch up again
- Otherwise, next checkpoint will have a heavy alignment as well



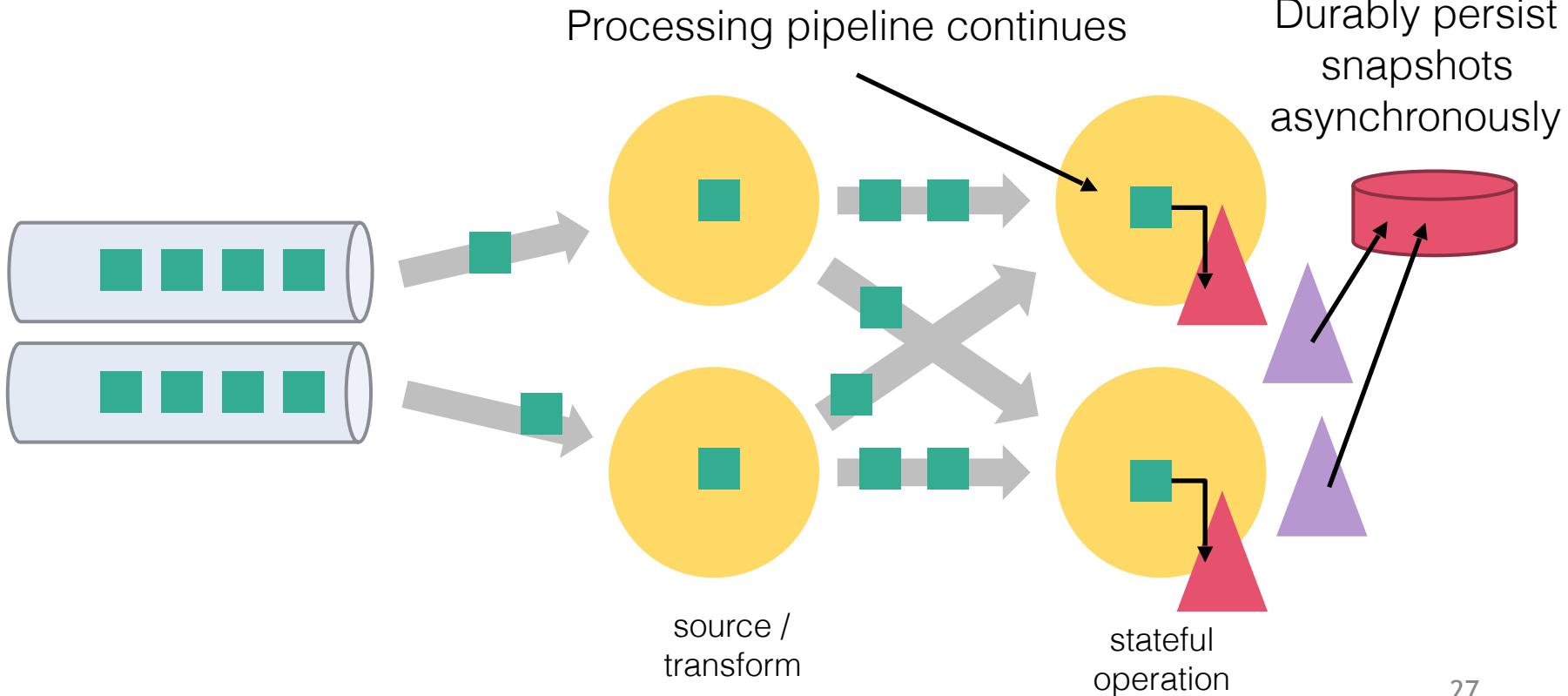


Catching up from heavy alignments

- Giving the computation time to catch up before starting the next checkpoint
 - Useful: Set the min-time-between-checkpoints
- Asynchronous checkpoints help a lot!
 - Shorter stalls in the pipelines means less build-up of in-flight data
 - Catch up already happens concurrently to state materialization



Asynchronous Checkpoints



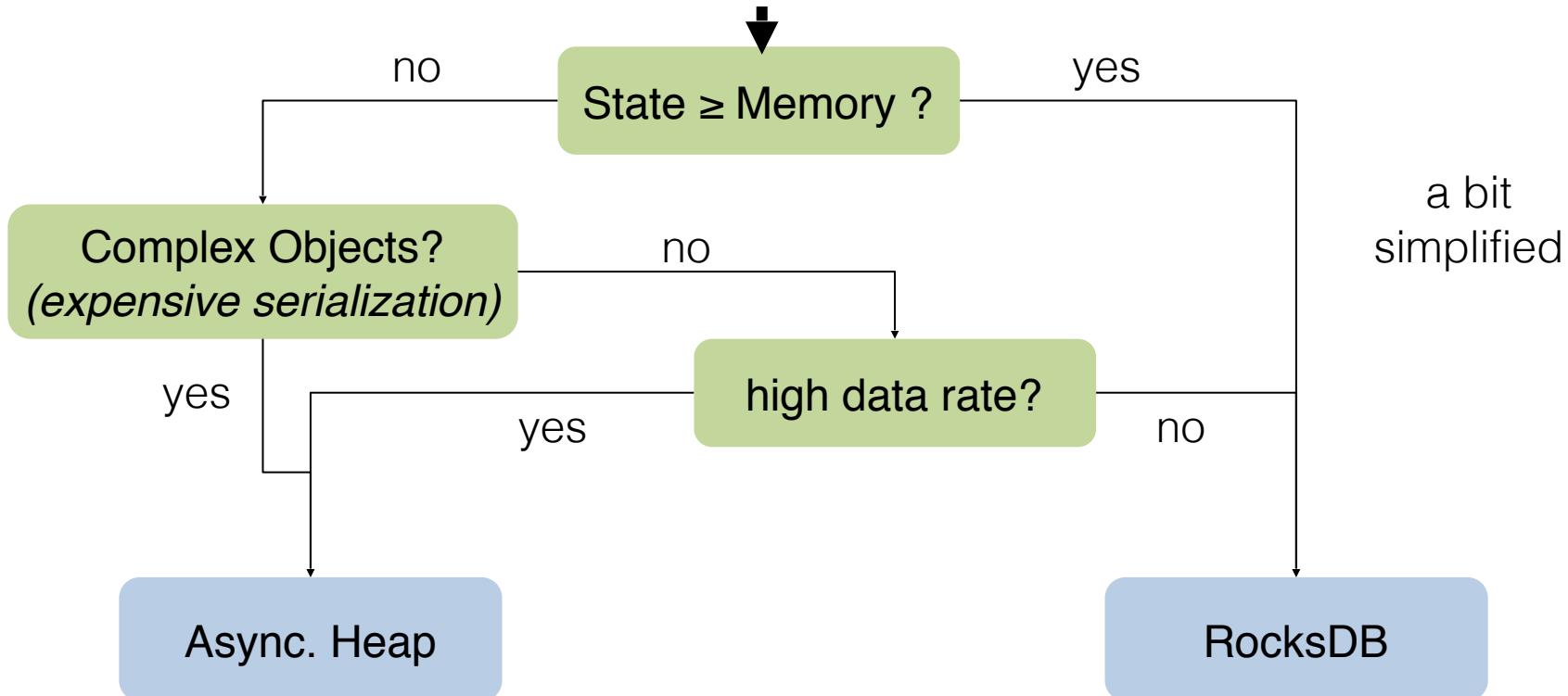


Asynchrony of different state types

State	Flink 1.2	Flink 1.3	Flink 1.3 +
Keyed state RocksDB	✓	✓	✓
Keyed State on heap	✗ (✓) (hidden in 1.2.1)	✓	✓
Timers	✗	✓/✗	✓
Operator State	✗	✓	✓



When to use which state backend?



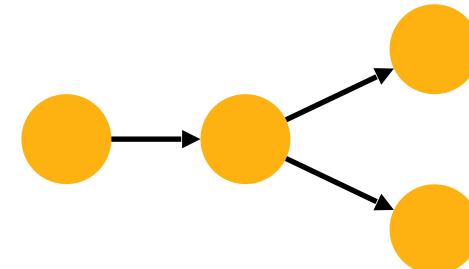


File Systems, Object Stores, and Checkpointed State



Exceeding FS request capacity

- Job size: 4 operators
- Parallelism: 100s to 1000
- State Backend: `FsStateBackend`
- State size: few KBs per operator, 100s to 1000 of files
- Checkpoint interval: few secs
- Symptom: S3 blocked off connections after exceeding 1000s HEAD requests / sec

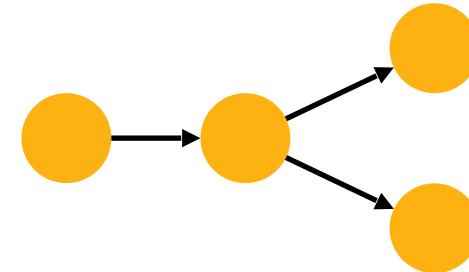




Exceeding FS request capacity

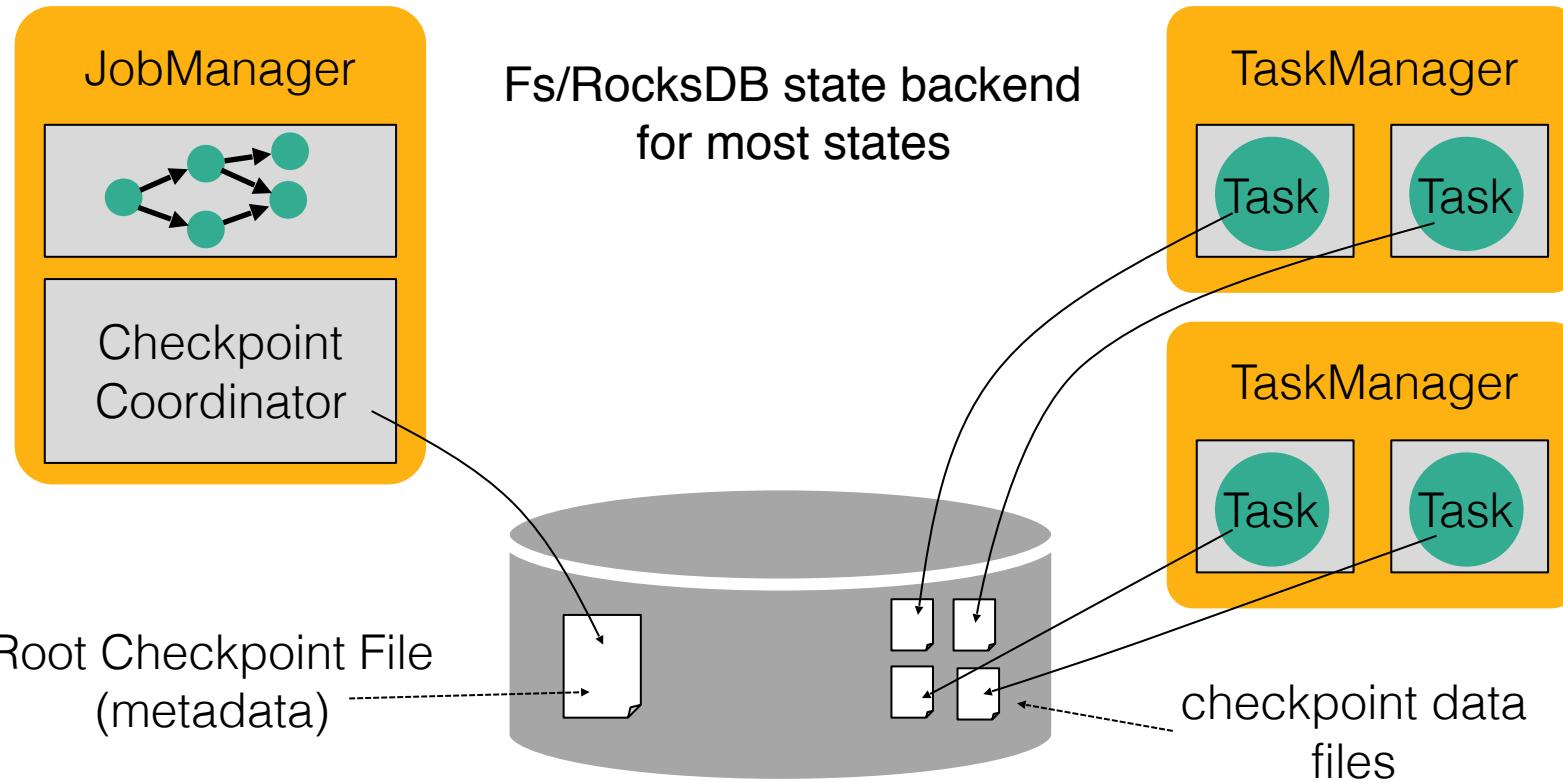
What happened?

- Operators prepare state writes, ensure parent directory exists
- Via the S3 FS (from Hadoop), each `mkdirs` causes 2 HEAD requests
- Flink 1.2: Lazily initialize checkpoint preconditions (dirs.)
- Flink 1.3: Core state backends reduce assumption of directories (PUT/GET/DEL), rich file systems support them as fast paths



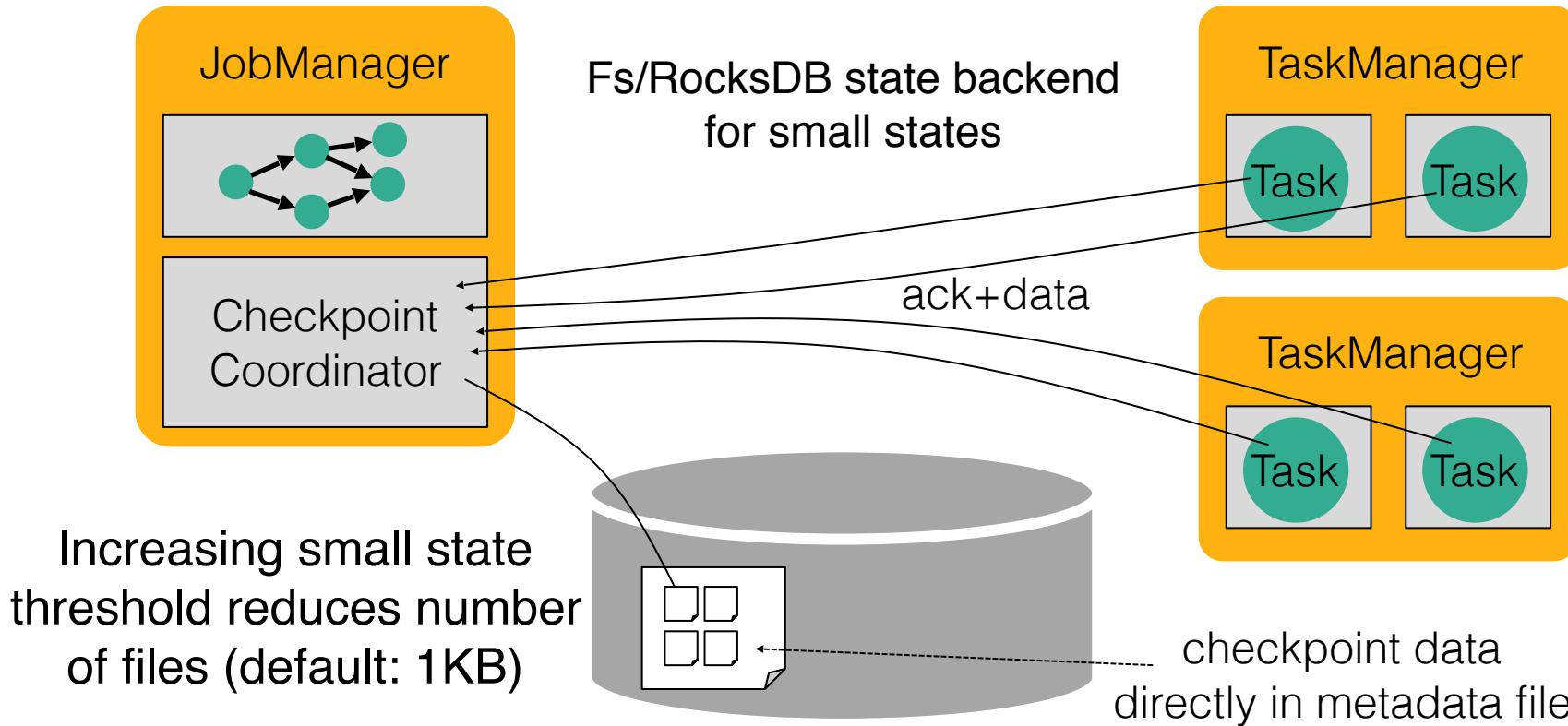


Reducing FS stress for small state





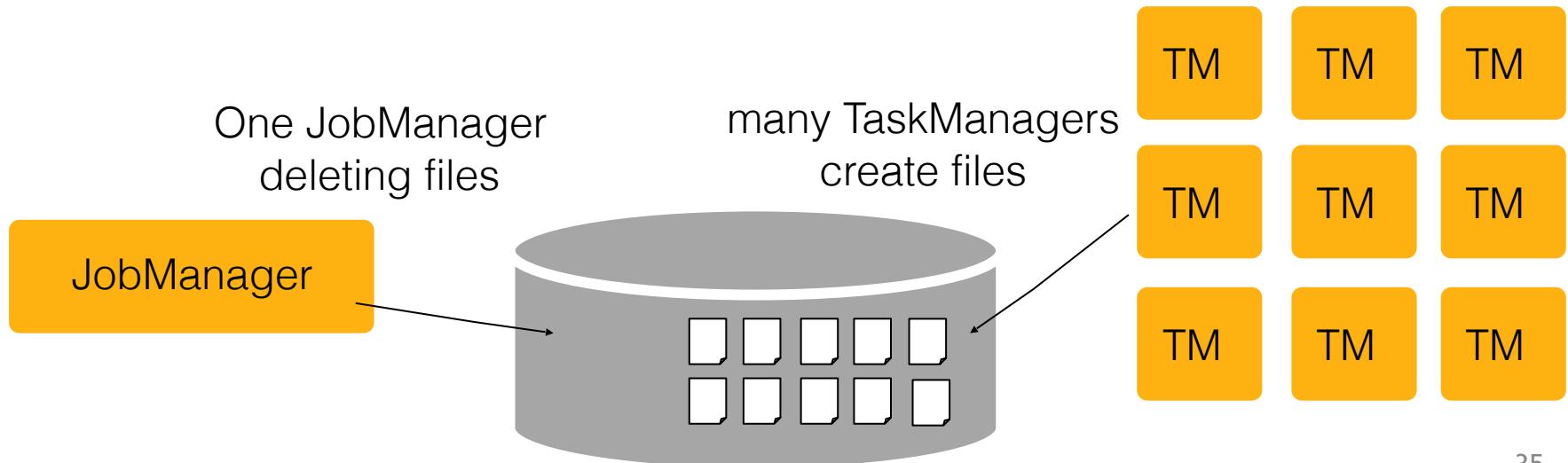
Reducing FS stress for small state





Lagging state cleanup

Symptom: Checkpoints get cleaned up too slow
State accumulates over time





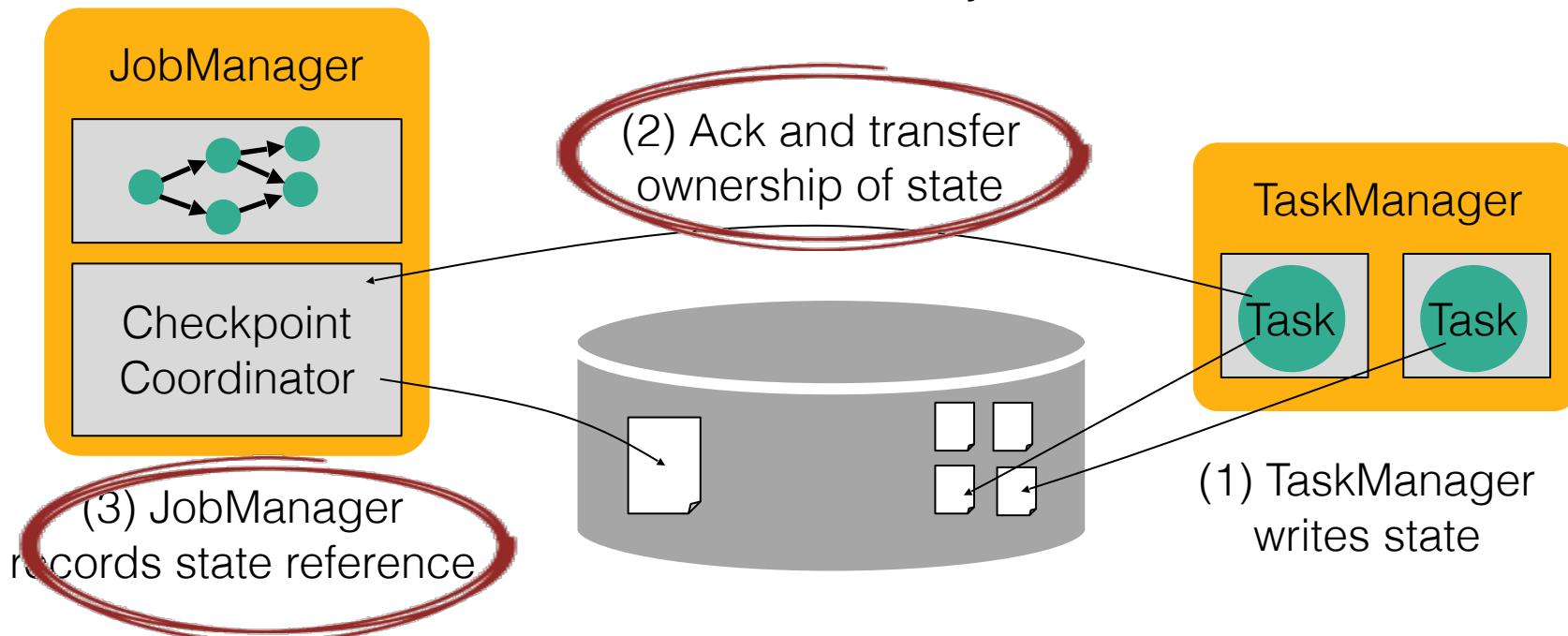
Lagging state cleanup

- **Problem:** FileSystems and Object Stores offer only synchronous requests to delete state object
Time to delete a checkpoint may accumulates to minutes.
- **Flink 1.2:** Concurrent checkpoint deletes on the JobManager
- **Flink 1.3:** For FileSystems with actual directory structure, use recursive directory deletes (one request per directory)



Orphaned Checkpoint State

Who owns state objects at what time?



Orphaned Checkpoint State



Upcoming: Searching for orphaned state

fs://checkpoints/job-61776516/chk-113

chk-129

chk-221

chk-271

It gets more complicated with incremental checkpoints...
Chk-272

4

11 252

38



Conclusion & General Recommendations



The closer your application is to saturating either network, CPU, memory, FS throughput, etc. the sooner an extraordinary situation causes a regression

Enough headroom in provisioned capacity means fast catchup after temporary regressions

Be aware that certain operations are spiky
(like aligned windows)

Production test always with checkpoints ;-)



Recommendations (*part 1*)

Be aware of the inherent scalability of primitives

- Broadcasting state is useful, for example for updating rules / configs, dynamic code loading, etc.
- Broadcasting does not scale, i.e., adding more nodes does not. Don't use it for high volume joins

- Putting very large objects into a **ValueState** may mean big serialization effort on access / checkpoint
- If the state can be mappified, use **MapState** – it performs much better



Recommendations *(part 2)*

If you are about recovery time

- Having spare TaskManagers helps bridge the time until backup TaskManagers come online
- Having a spare JobManager can be useful
 - Future: JobManager failures are non disruptive



Recommendations *(part 3)*

If you care about CPU efficiency, watch your serializers

- JSON is a flexible, but awfully inefficient data type
- Kryo does okay - make sure you register the types
- Flink's directly supported types have good performance
basic types, arrays, tuples, ...
- Nothing ever beats a custom serializer ;-)



Thank you!
Questions?



dataArtisans

EXPERIENCES WITH STREAMING & MICRO-BATCH FOR ONLINE LEARNING

Swaminathan Sundararaman
FlinkForward 2017



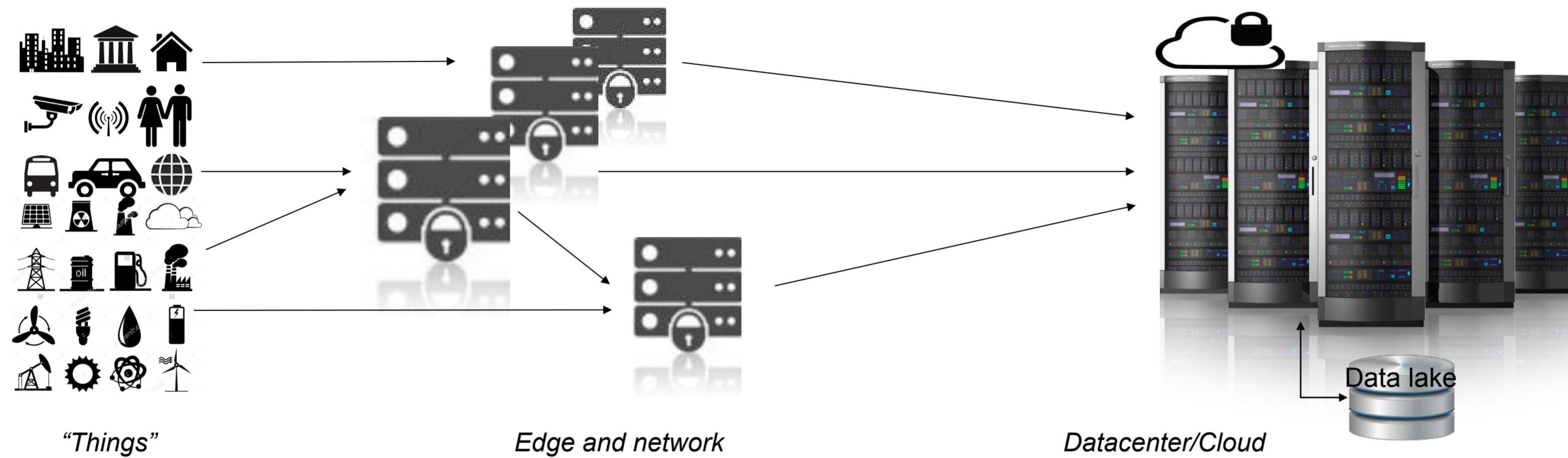
P/A

PARALLEL
MACHINES

Powering Machine Learning

The Challenge of Today's Analytics Trajectory

IoT is Driving Explosive Growth in Data Volume



Edges benefit from real-time online learning and/or inference

Real-Time Intelligence: Online Algorithm Advantages

- **Real-world data is unpredictable and bursty**
 - Data behavior changes (different time of day, special events, flash crowds, etc.)
- **Data behavior changes require retraining & model updates**
 - Updating models offline can be expensive (compute, retraining)
- **Online algorithms retrain on the fly with real-time data**
 - Lightweight, low compute and memory requirements
 - Better accuracy through continuous learning
- **Online algorithms are more accurate, especially with data behavior changes**

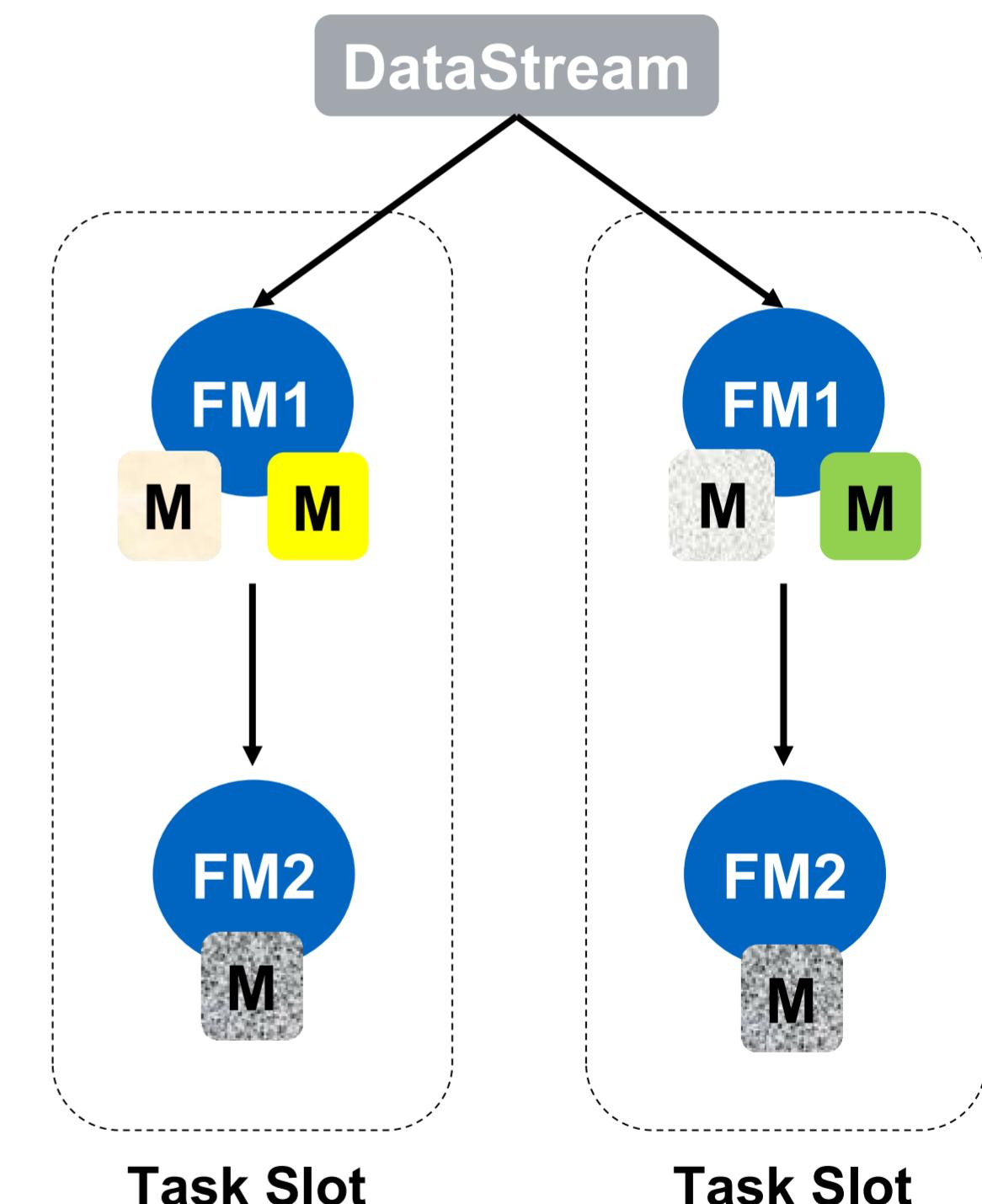
Experience Building ML Algorithms on Flink 1.0

- Built both Offline(Batch) and Online algorithms
 - Batch Algorithms (Examples: KMeans, PCA, and Random Forest)
 - Online Algorithms (Examples: Online KMeans, Online SVM)
- Uses many of the Flink DataStream primitives:
 - DataStream APIs are sufficient and primitives are generic for ML algorithms.
 - CoFlatMaps, Windows, Collect, Iterations, etc.
- We have also added Python Streaming API support in Flink and are working with dataArtisans to contribute it to upstream Flink.

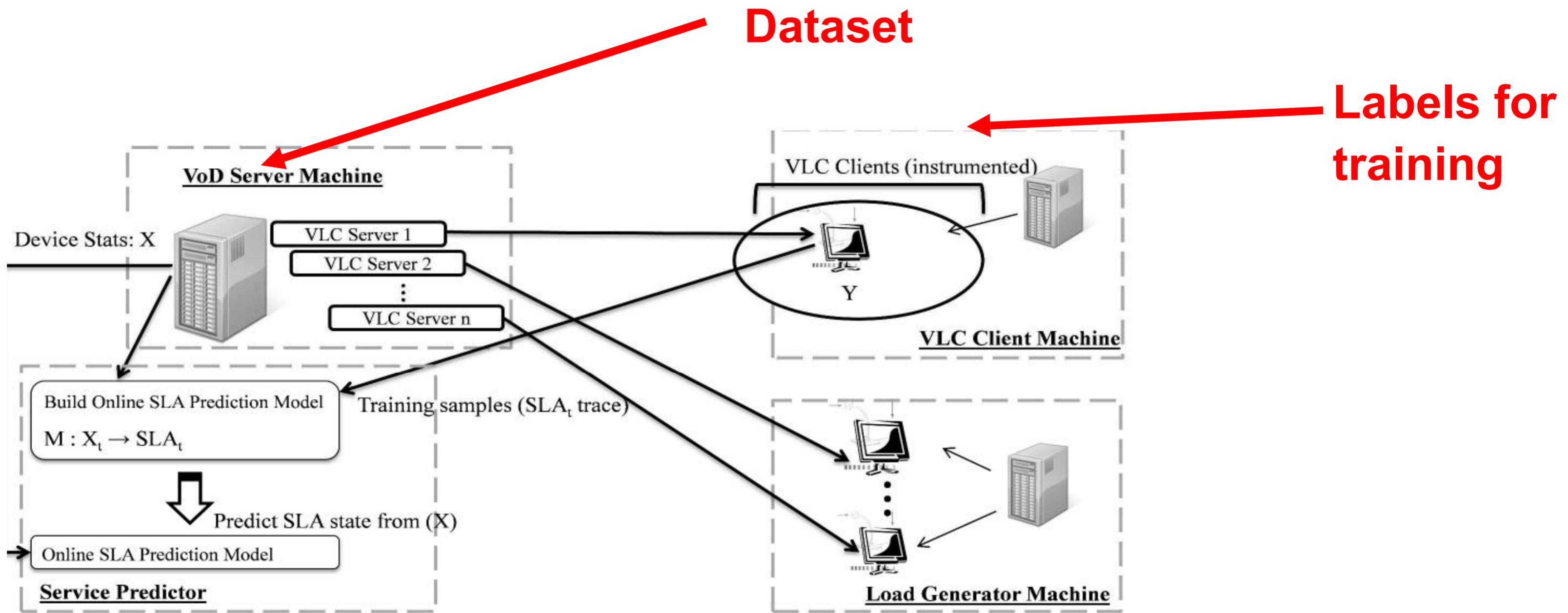
Example: Online SVM Algorithm

```
/* Co-map to update local model(s) when new data arrives and also  
create the shared model when a pre-defined threshold is met */  
private case class SVMModelCoMap(...) {  
    /* flatMap1 processes new elements and updates local model*/  
    def flatMap1(data: LabeledVector[Double],  
                 out: Collector[Model]) {  
        ...  
    }  
    /* flatMap2 accumulates local models and creates a new model  
(with decay) once all local models are received */  
    def flatMap2(currentModel: Model, out: Collector[Model]) {  
        ...  
    }  
}  
  
object OnlineSVM {  
    ...  
    def main(args: Array[String]): Unit = {  
        // initialize input arguments and connectors  
        ...  
    }  
}
```

Aggregated and local models combined with decay factor



Telco Example: Measuring SLA Violations



- A server providing VoD services to VLC (i.e., media player) clients
 - Clients request videos of different sizes at different times
 - Server statistics used to predict violations
- **SLA violation:** service level drops below predetermined threshold

Dataset

(<https://arxiv.org/pdf/1509.01386.pdf>)

CPU Utilization	Memory/Swap	I/O Transactions	Block I/O operations	Process Statistics	Network Statistics
CPU Idle	Mem Used	Read transactions/s	Block Reads/s	New Processes/s	Received packets/s
CPU User	Mem Committed	Write transactions/s	Block Writes/s	Context Switches/s	Transmitted Packets/s
CPU System	Swap Used	Bytes Read/s			Received Data (KB)/s
CPU IO_Wait	Swap Cached	Bytes Written/s			Transmitted Data (KB)/s
					Interface Utilization %

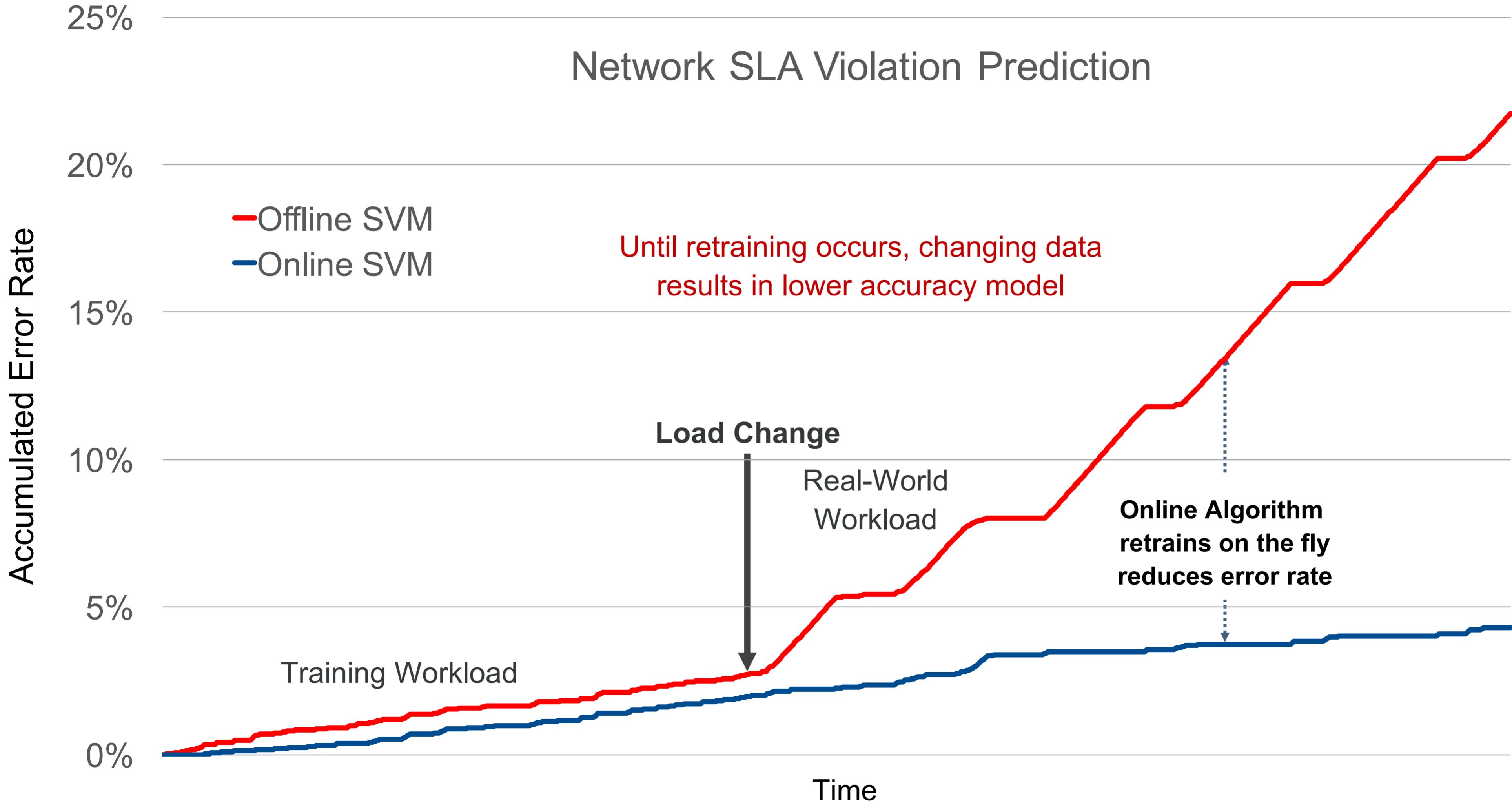
- Load patterns – Flashcrowd, Periodic
- Delivered to Flink and Spark as live stream in experiments

Fixed workloads – Online vs Offline (Batch)

Load Scenario	Offline (LibSVM) Accuracy	Offline (Pegasos) Accuracy	Online SVM Accuracy
flashcrowd_load	0.843	0.915	0.943
periodic_load	0.788	0.867	0.927
constant_load	0.999	0.999	0.999
poisson_load	0.963	0.963	0.971

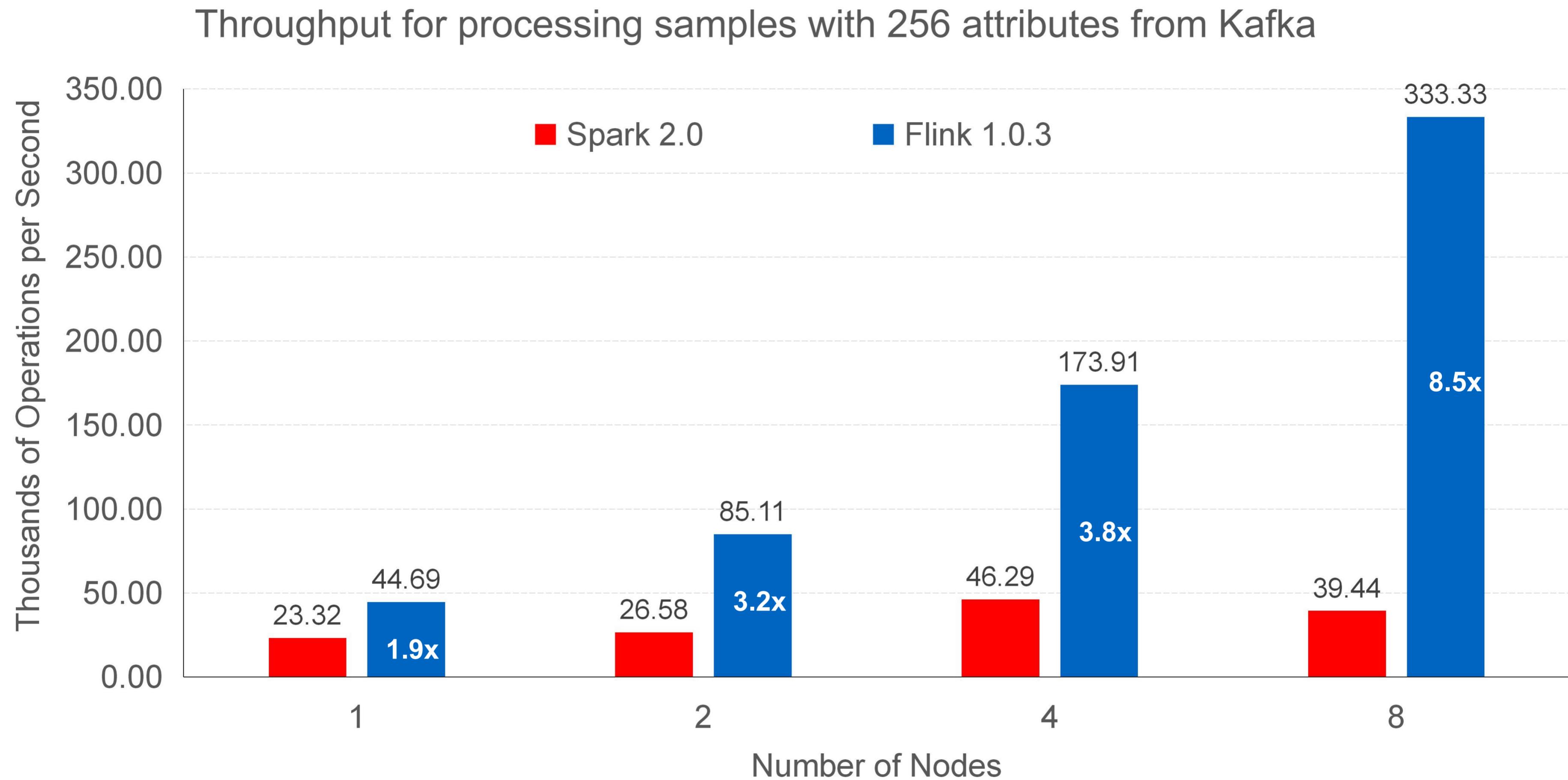
**When load pattern remains static (unchanged),
Online algorithms can be as accurate as Offline algorithms**

Online SVM vs Batch (Offline) SVM – both in Flink



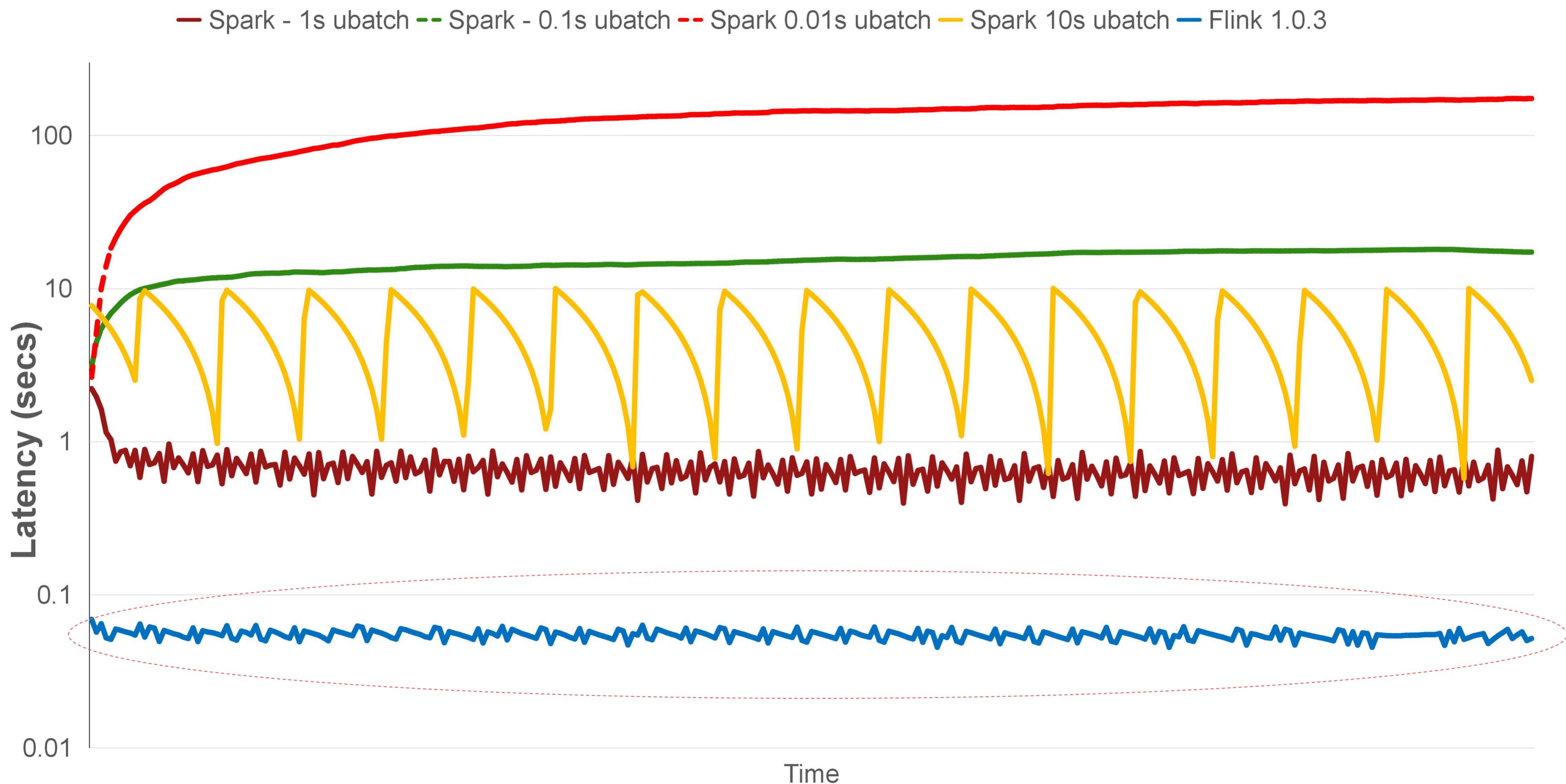
Online algorithms quickly adapt to workload changes

Throughput: Online SVM in Streams and Micro-Batch



Notable performance improvement over micro-batch based solution

Latency: Online SVM in Streams & Micro-batch



Low and predictable latency as needed in Edge

Conclusions

Edge computing & Online learning are needed for real-time analytics

- **Edge Computing:** minimizes the excessive latencies, reaction time
- **Online learning:** can dynamically adapt to changing data / behavior

Online machine learning with streaming on Flink

- Supports *low latency processing* with *scaling* across multiple nodes
- Using real world data, demonstrate improved accuracy over offline algorithms

Parallel Machines

The Machine Learning Management Solution

info@parallelmachines.com



dataArtisans

dataArtisans

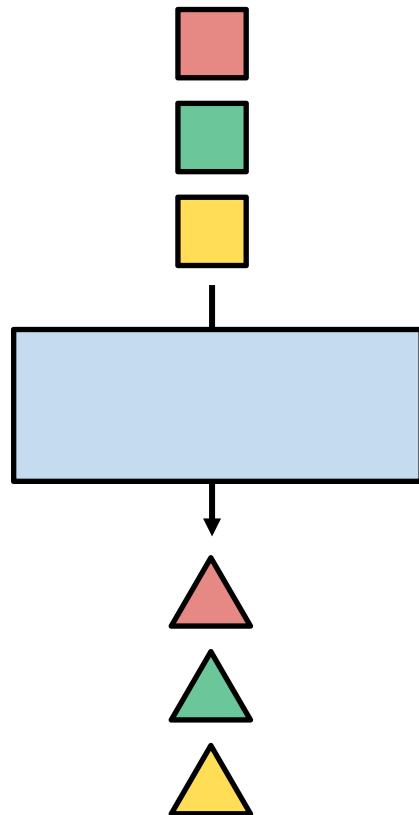


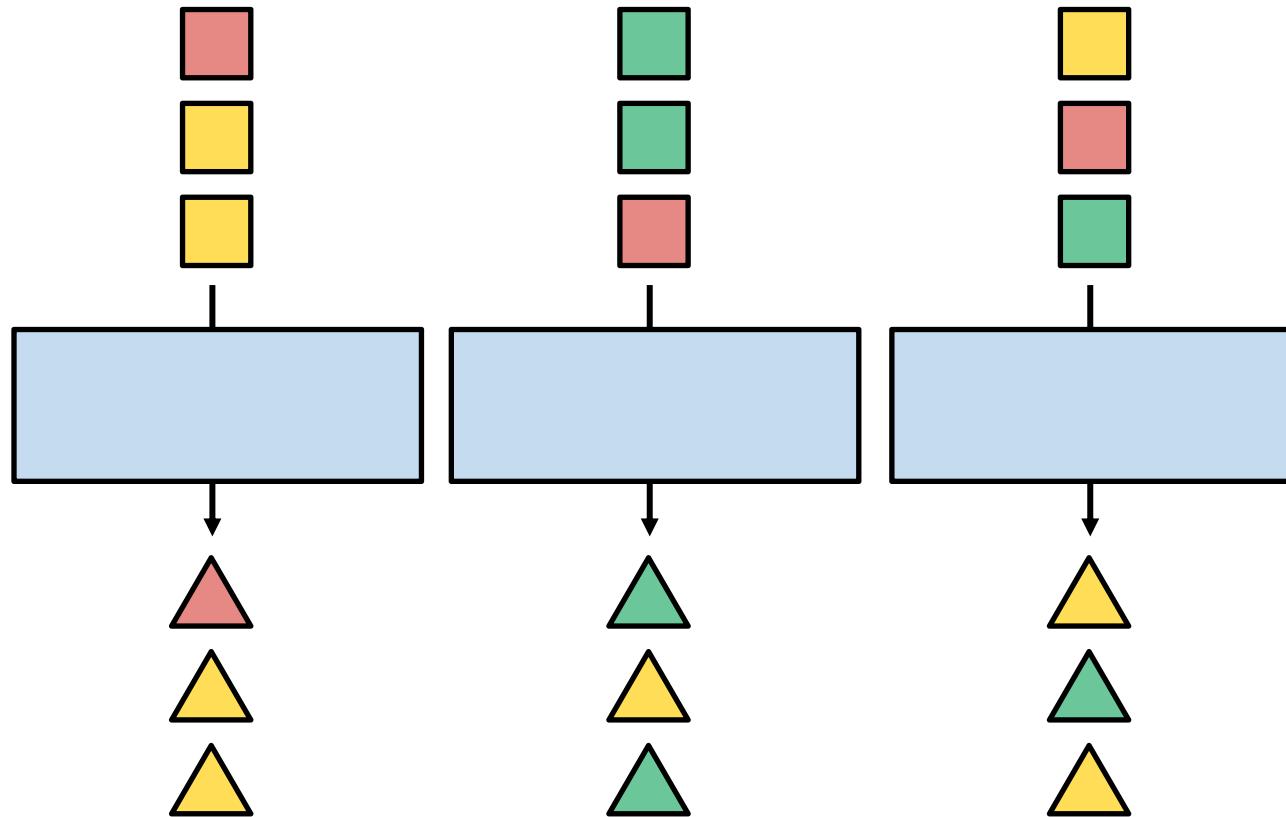
PLATFORM

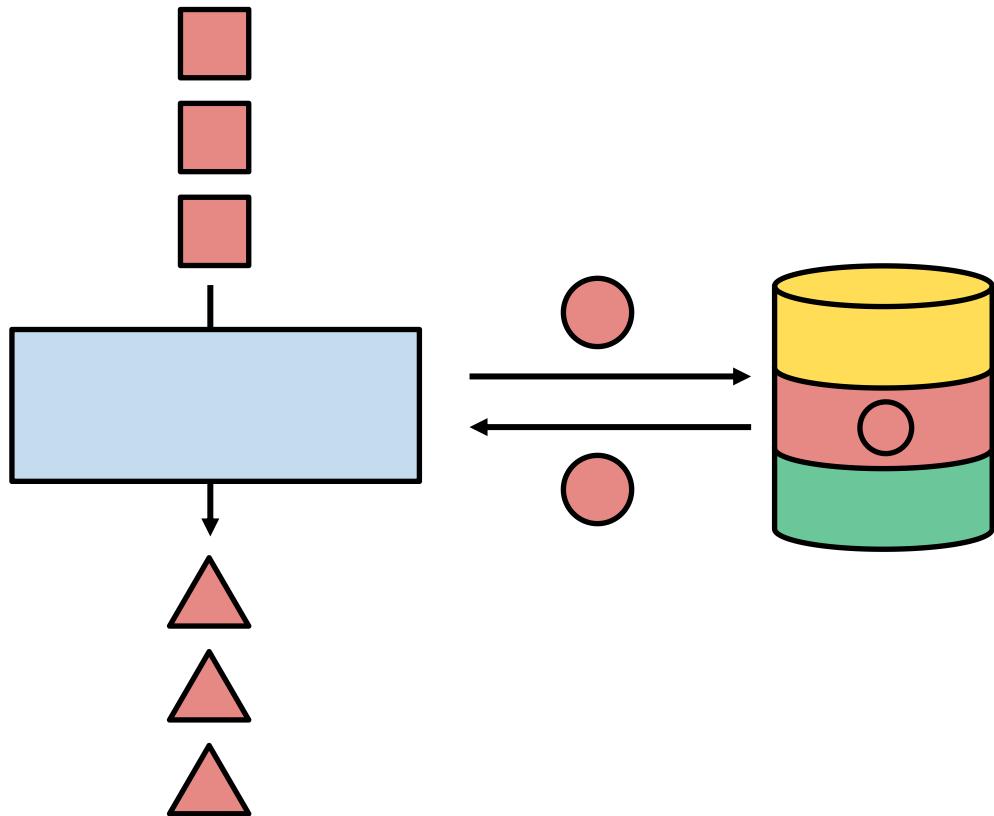






















```
/*
 * Process one element from the input stream.
 */
void processElement(I value, Context ctx, Collector<O> out) throws Exception;

/*
 * Called when a timer set using {@link TimerService} fires.
 */
void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out) throws Exception;
```



```
/*
 * Process one element from the input stream.
 */
void processElement(I value, Context ctx, Collector<0> out) throws Exception;

/*
 * Called when a timer set using {@link TimerService} fires.
 */
void onTimer(long timestamp, OnTimerContext ctx, Collector<0> out) throws Exception;
```



```
/*
 * Process one element from the input stream.
 */
void processElement(I value, Context ctx, Collector<O> out) throws Exception;

/*
 * Called when a timer set using {@link TimerService} fires.
 */
void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out) throws Exception;
```





ValueState





```
public class MyProcessFunction extends
    ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>> {

    // define your state descriptors

    @Override
    public void processElement(Tuple2<String, Long> value, Context ctx,
        Collector<Tuple2<String, Long>> out) throws Exception {
        // update our state and register a timer
    }

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx,
        Collector<Tuple2<String, Long>> out) throws Exception {
        // check the state for the key and emit a result if needed
    }
}
```



```
public class MyProcessFunction extends  
    ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>> {  
  
    // define your state descriptors  
    private final ValueStateDescriptor<CounterWithTS> stateDesc =  
        new ValueStateDescriptor<>("myState", CounterWithTS.class);  
  
}
```



```
public class MyProcessFunction extends
    ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>> {

    @Override
    public void processElement(Tuple2<String, String> value, Context ctx,
        Collector<Tuple2<String, Long>> out) throws Exception {

        ValueState<MyStateClass> state = getRuntimeContext().getState(stateDesc);
        CounterWithTS current = state.value();
        if (current == null) {
            current = new CounterWithTS();
            current.key = value.f0;
        }
        current.count++;
        current.lastModified = ctx.timestamp();
        state.update(current);
        ctx.timerService().registerEventTimeTimer(current.lastModified + 100);
    }
}
```



```
public class MyProcessFunction extends
    ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>> {

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx,
                        Collector<Tuple2<String, Long>> out) throws Exception {
        CounterWithTS result = getRuntimeContext().getState(stateDesc).value();
        if (timestamp == result.lastModified + 100) {
            out.collect(new Tuple2<String, Long>(result.key, result.count));
        }
    }
}
```



```
stream.keyBy("key")
    .process(new MyProcessFunction())
```







```
final OutputTag<String> outputTag = new OutputTag<String>("gt10"){};  
  
SingleOutputStreamOperator<Tuple2<String, Long>> mainStream = input.process(  
    new ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>>() {  
  
        @Override  
        public void onTimer(long timestamp, OnTimerContext ctx,  
            Collector<Tuple2<String, Long>> out) throws Exception {  
            CounterWithTS result = getRuntimeContext().getState(adStateDesc).value();  
            if (timestamp == result.lastModified + 100) {  
                out.collect(new Tuple2<String, Long>(result.key, result.count));  
            } else if (result.count > 10) {  
                ctx.output(outputTag, result.key);  
            }  
        }  
  
        DataStream<String> sideOutputStream = mainStream.getSideOutput(outputTag);
```



```
final OutputTag<String> outputTag = new OutputTag<String>("gt10"){};  
  
SingleOutputStreamOperator<Tuple2<String, Long>> mainStream = input.process(  
    new ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>>() {  
  
        @Override  
        public void onTimer(long timestamp, OnTimerContext ctx,  
            Collector<Tuple2<String, Long>> out) throws Exception {  
            CounterWithTS result = getRuntimeContext().getState(adStateDesc).value();  
            if (timestamp == result.lastModified + 100) {  
                out.collect(new Tuple2<String, Long>(result.key, result.count));  
            } else if (result.count > 10) {  
                ctx.output(outputTag, result.key);  
            }  
        }  
  
        DataStream<String> sideOutputStream = mainStream.getSideOutput(outputTag);
```



- CoProcessFunction







MapFunction





- MapFunction

-

-

-

-



Sync. I/O



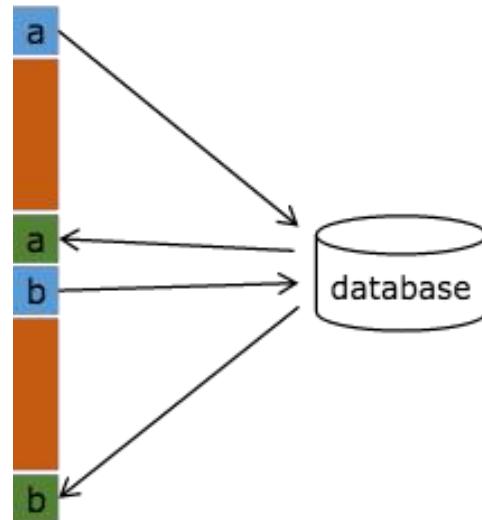
sendRequest(x)

receiveResponse(x)

wait



Sync. I/O



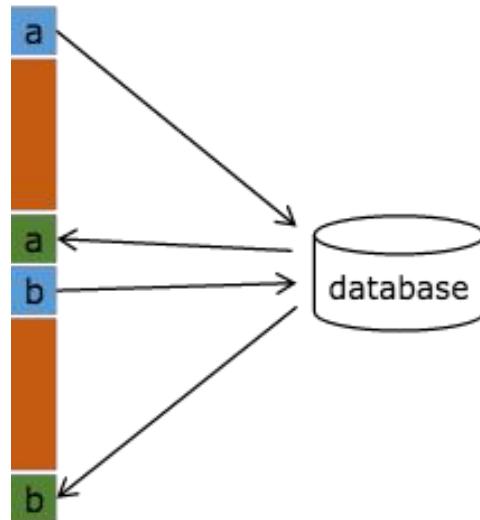
sendRequest(x)

receiveResponse(x)

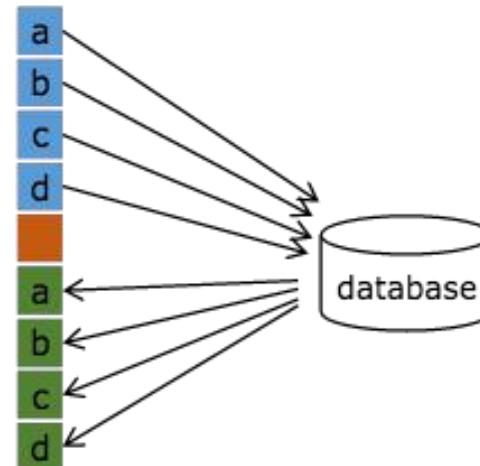
wait



Sync. I/O



Async. I/O



sendRequest(x)

receiveResponse(x)

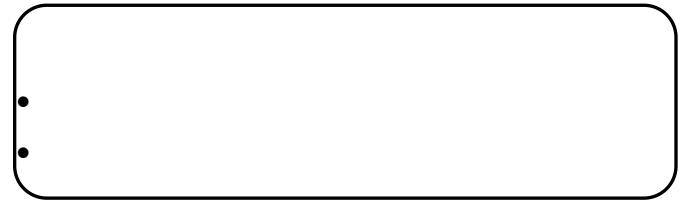
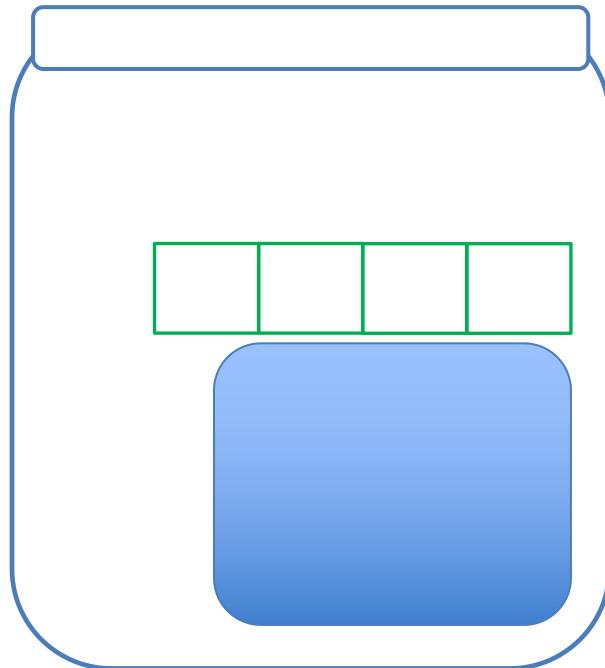
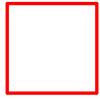
wait

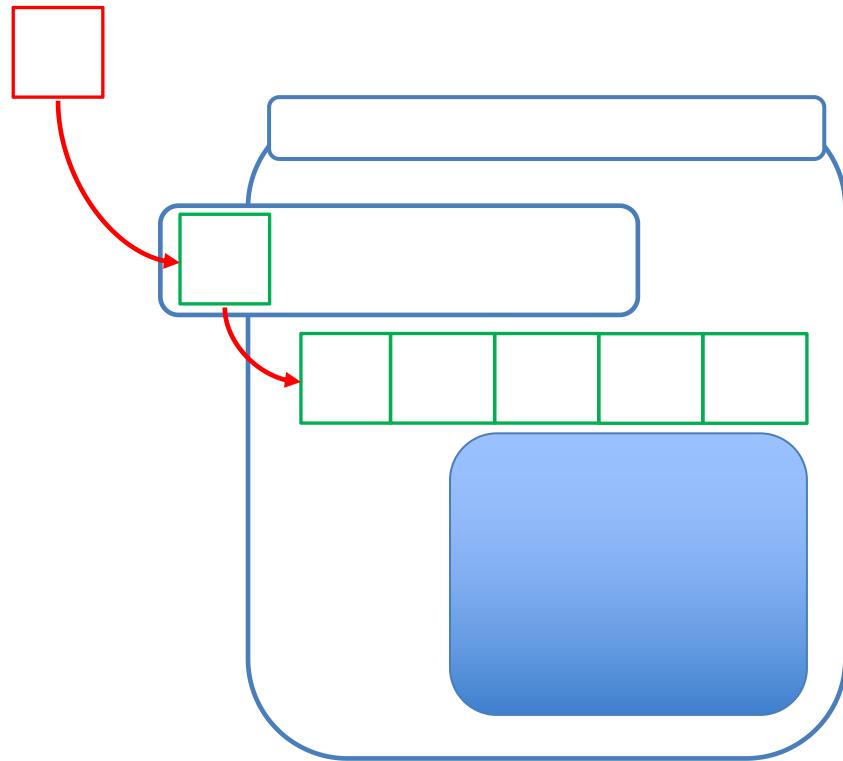


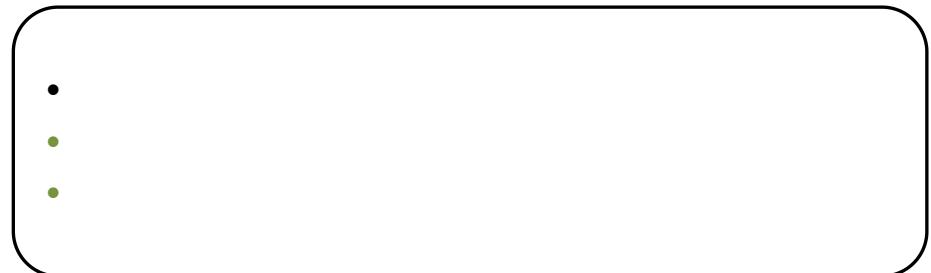
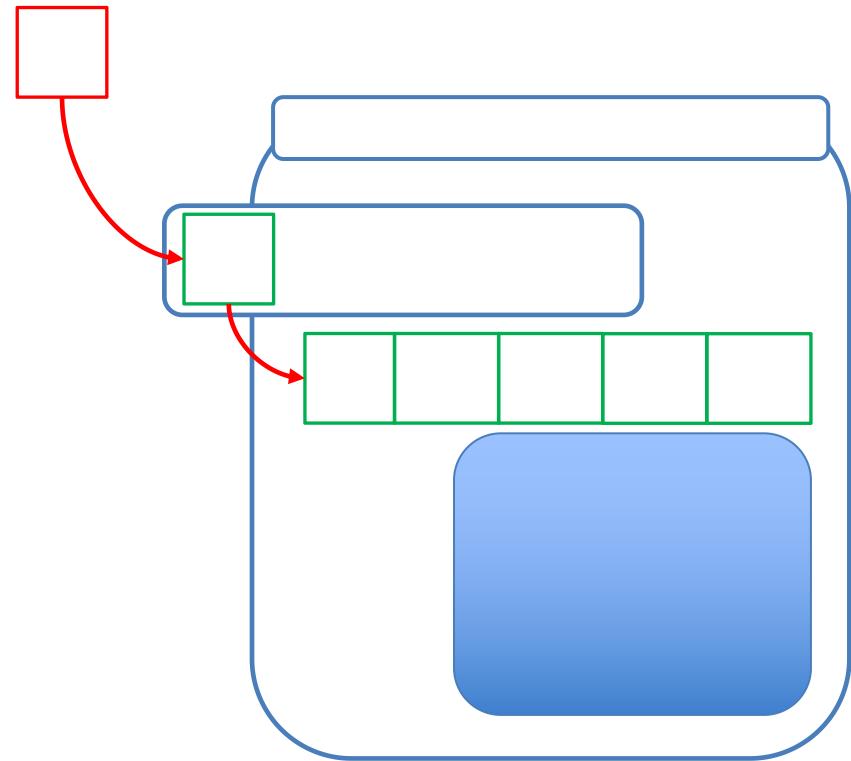


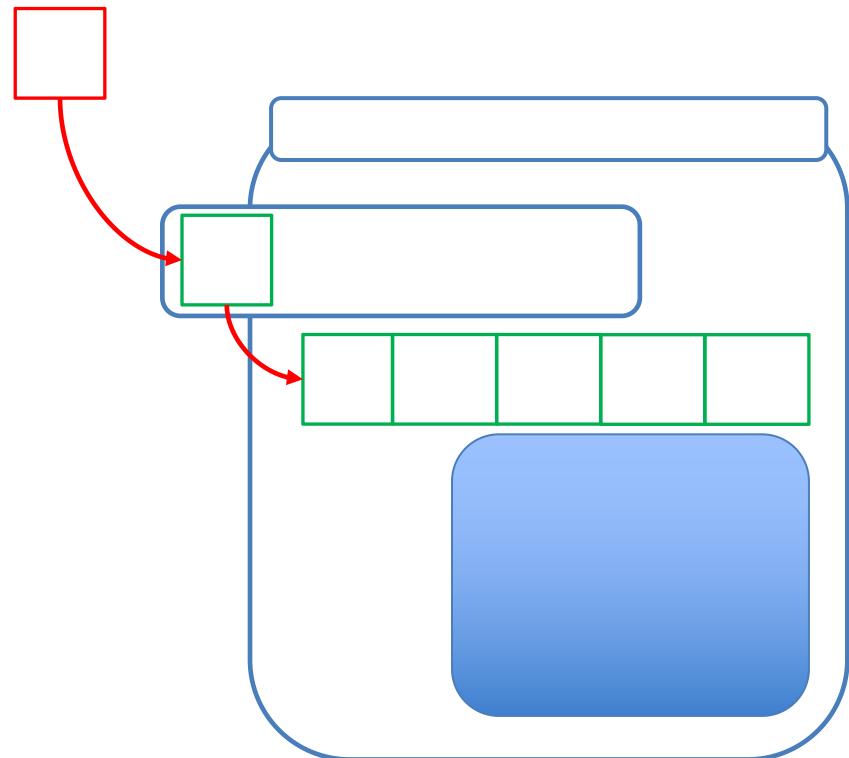
```
/*
 * Trigger async operation for each stream input.
 */
void asyncInvoke(IN input, AsyncCollector<OUT> collector) throws Exception;

/*
 * Example async function call.
 */
DataStream<...> result = AsyncDataStream.(un)orderedWait(stream,
    new MyAsyncFunction(), 1000, TimeUnit.MILLISECONDS, 100);
```



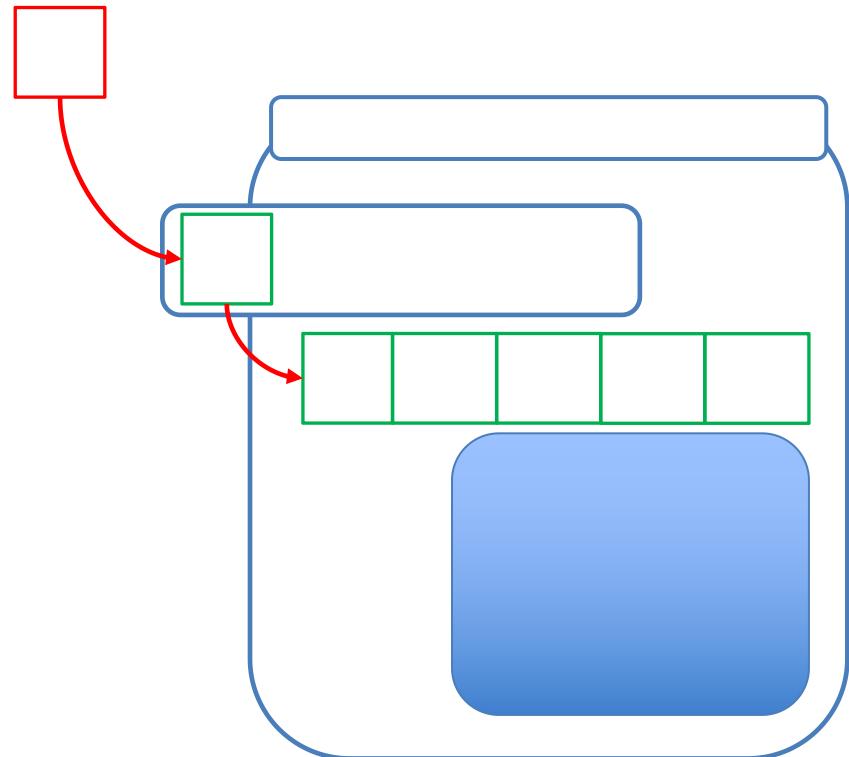






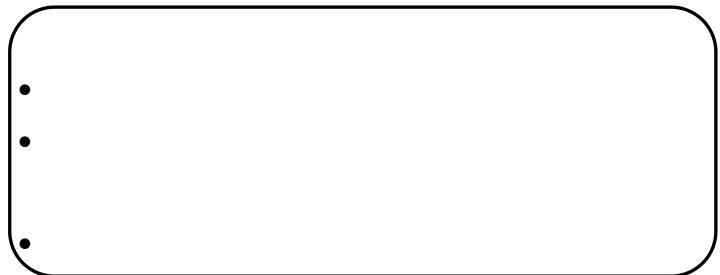
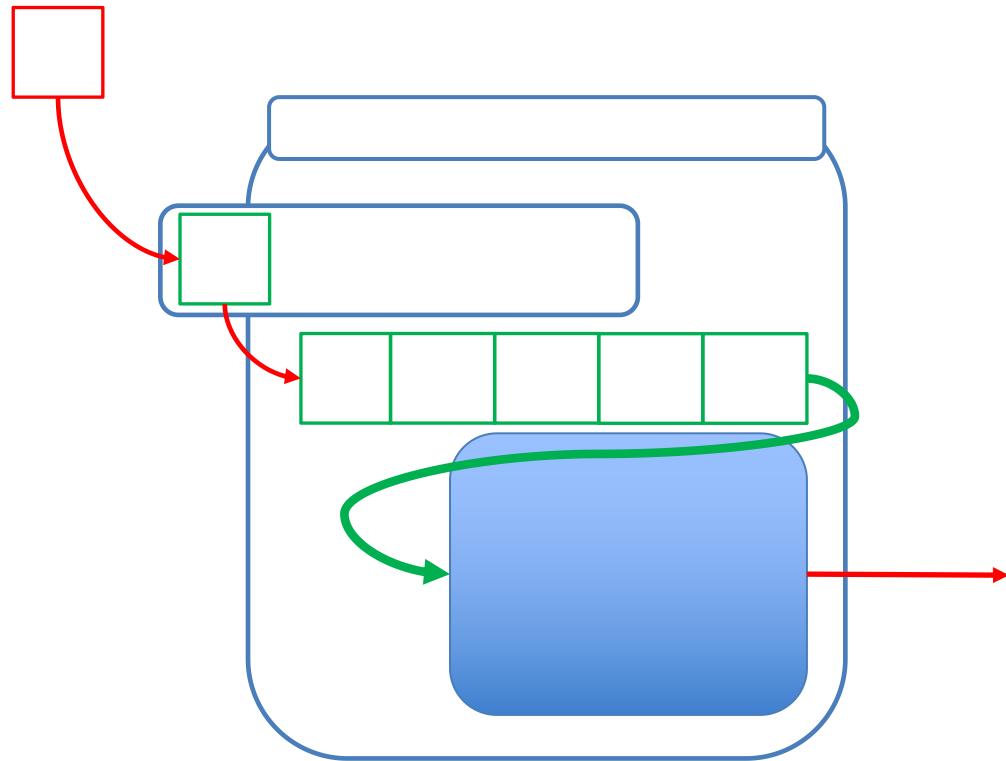
-
-
-

```
Future<String> future = client.query( );  
  
future.thenAccept((String result) -> {  
    .collect(  
        Collections.singleton(  
            new Tuple2<>( , result)));  
});
```



-
-
-

```
Future<String> future = client.query( );  
  
future.thenAccept((String result) -> {  
    .collect(  
        Collections.singleton(  
            new Tuple2<>( , result)));  
});
```





```
DataStream<Tuple2<String, String>> result =  
    AsyncDataStream.(un)orderedWait(stream,  
        new MyAsyncFunction(),  
        1000, TimeUnit.MILLISECONDS,  
        100);
```

- **asyncFunction**

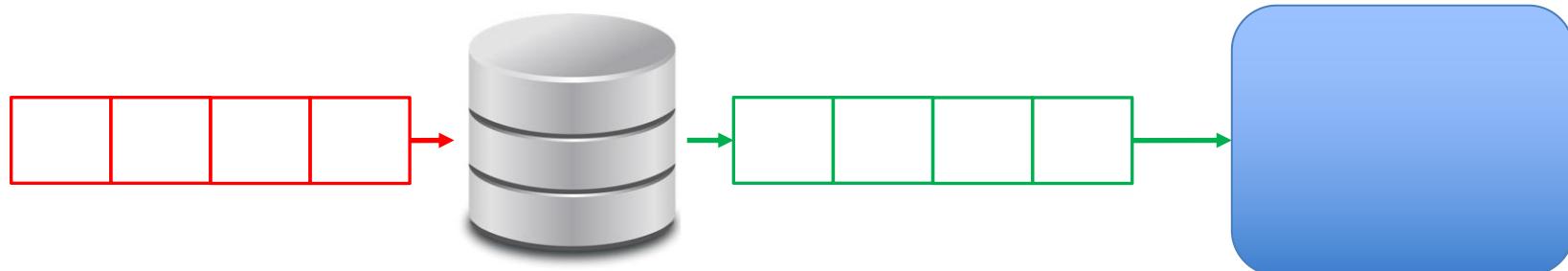




```
DataStream<Tuple2<String, String>> result =  
    AsyncDataStream.(un)orderedWait(stream,  
        new MyAsyncFunction(),  
        1000, TimeUnit.MILLISECONDS,  
        100);
```

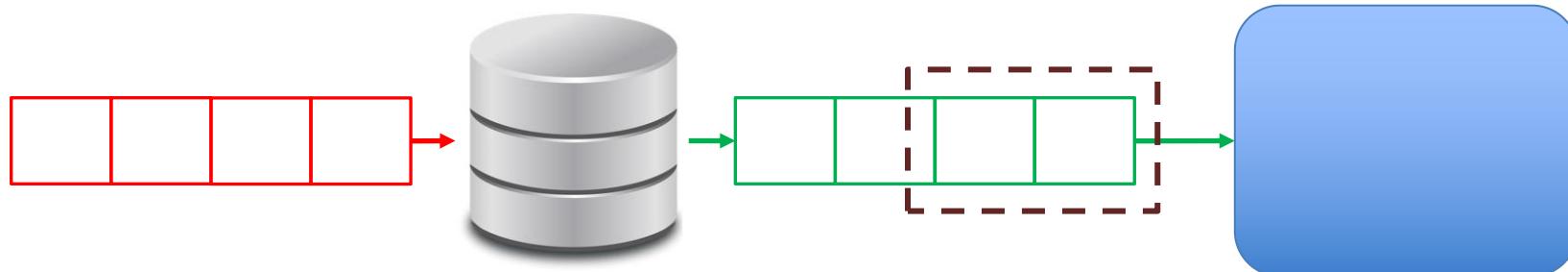


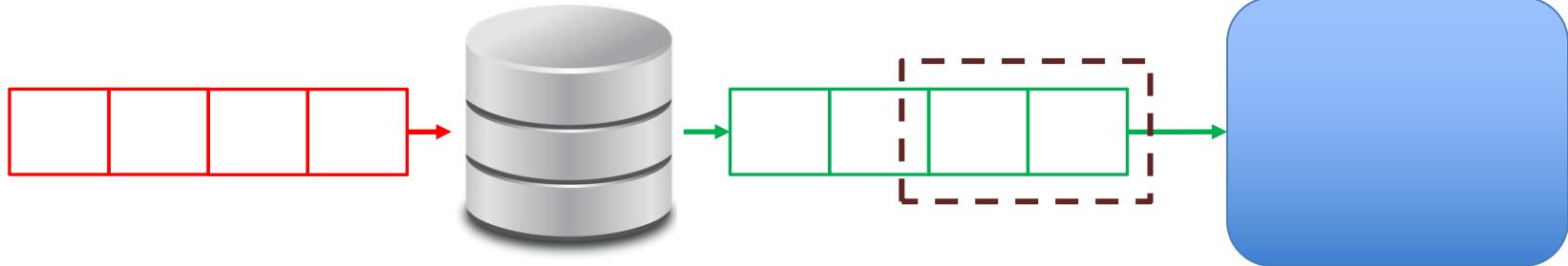
```
DataStream<Tuple2<String, String>> result =  
    AsyncDataStream.(un)orderedWait(stream,  
        new MyAsyncFunction(),  
        1000, TimeUnit.MILLISECONDS,  
        100);
```





```
DataStream<Tuple2<String, String>> result =  
    AsyncDataStream.unorderedWait(stream,  
        new MyAsyncFunction(),  
        1000, TimeUnit.MILLISECONDS,  
        100);
```



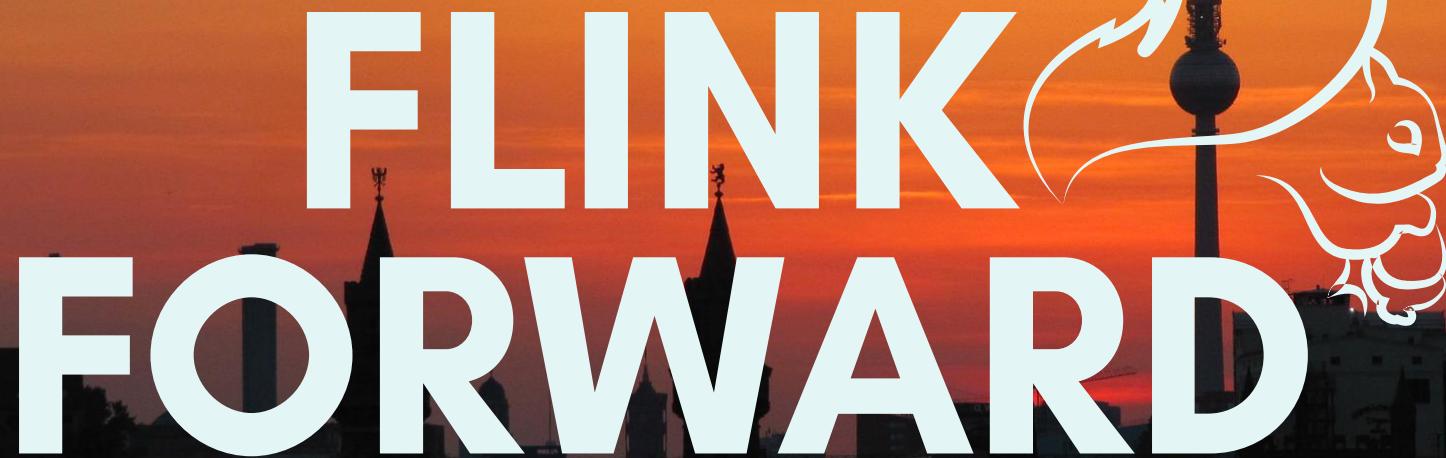


- `unorderedWait`
- `orderedWait`

■



FLINK FORWARD



FLINK FORWARD IS COMING BACK TO BERLIN

SEPTEMBER 11-13, 2017

BERLIN.FLINK-FORWARD.ORG

dataArtisans

Flink, Queryable State, and High Frequency Time Series Data

Joe Olson
Data Architect
PhysIQ
11Apr2017

About Us / Our Data....

- **What?** Tech company that collects, stores, enriches, and presents vitals data for a given patient (heart rate, O₂ levels, respiration rate, etc)
- **Why?** To build a predictive model of patient's state of health.
- **Who?** End users are patients and health care staff at care facilities (or at home!)
- **How?**
 - Data originates from wearable patches
 - Collected as a waveform – must be converted into friendly numeric types (think PLCs in a IoT type application)
 - Stream in in 1 second chunks (2KB – 6KB)
 - May represent data sampled anywhere from 1Hz to 200Hz
 - Treat data as a stream throughout all data flows

State in a Flink Stream...

- **Keyed State** – state associated with a partition determined by a keyed stream. One partition space per key.
- **Operator State** – state associated with an operator instance. Example: Kafka connector. Each parallel instance of the connector maintains its own state.

Both of these states exist in two forms:

- **Managed** – data structures controlled by Flink
 - **Raw** – user defined byte array
-
- Our use case leverages **managed keyed state**

Managed Keyed State

- `ValueState<T>`: a value that can be updated and retrieved. Two main methods:
 - `.update()` Set the value
 - `.value()` Get the value
- `ListState<T>`: a list of elements that can be added to, or iterated over
 - `.add(T)` Add an item to the list
 - `Iterable<T> get()` Use to iterate
- `ReducingState<T>`: a single value that represents an aggregation of all values added to the state
 - `.add(T)` add to the state using a provided `ReduceFunction`

How Is State Persisted?

- 3 back end options for preserving state:
- **MemoryStateBackend**
 - Stored on the Java heap
 - Aggregate state must fit into Job Manager RAM.
 - Good for small state situations / testing
- **FsStateBackend**
 - Data held in task manager RAM
 - Upon checkpointing, writes state to file system (must be shared for HA)
 - Good for larger states, and HA situations
- **RocksDBStateBackend**
 - All data stored on disk in a RocksDB – an optimized KV store
 - Upon checkpointing, the entire RocksDB is copied
 - Good for very large states!
- Persistence options can be defined on the job level

Putting it all together...

Here is our stream:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStateBackend(new RocksDBStateBackend(
    "hdfs://namenode:40010/flink/checkpoints"))
val stream = env.addSource(someConsumer)
    .keyBy(anObject => (anObject.id1, anObject.id2))
    .flatMap(new doSomethingClass(param1, param2))
```

Here is our managed state:

```
class doSomethingClass(p1:Long, p2:Long) extends RichFlatMapFunction[...]

private val listStateDescriptor = new ListStateDescriptor("list-example",
    TypeInformation.of(new TypeHint[java.lang.Long]{}))
private val listState = getRuntimeContext.getListState(listStateDescriptor)

override def flatMap(value:anObject, out: Collector[T]): Unit = {
    listState.add(value.somethingID)
    if (value.time > someTimeThreshold) {
        listState.clear()
    }
}
```

Queryable State

- The “Queryable State” feature refers to managed state that is accessible outside of the Flink runtime environment via a Flink Streaming API call.
- How?

```
aStateDescriptor.setQueryable("queryable-name")
```

- To access a managed state descriptor outside of Flink:

```
val config:Configuration = new Configuration();
config.setString(ConfigConst.JOB_MANAGER_IPC_ADDRESS_KEY, serverIP)
config.setInteger(ConfigConst.JOB_MANAGER_IPC_PORT_KEY, port)

val client:QueryableStateClient = new QueryableStateClient(config)

val key = (id1, id2)

// jobID: make a REST call to http://<serverIP>:8081/joboverview/running
val results = queryClient.executeQuery(key, jobID, "queryable-name")

// Not shown: deserializing results into a scala class
```

More thoughts

- Managed state is created within the Flink runtime context
- To access state, you'll need access to the runtime context (the Rich... classes in the Flink Streaming API)
- The windowing functionality in the API is not exposed to the runtime context

```
// No QS visibility into this until *after* apply finishes

val stream = env.addSource(aConsumer)
    .assignTimestampsAndWatermarks(new timeStampWatermark())
    .keyBy(x => (x._1, x._2))
    .window(GlobalWindows.create())
    .trigger(new customTrigger())
    .apply(new applyRule())
    .addSink(new dataSink())
```

- More complex state management – (e.g. maps)
- Partitioning managed state into more manageable chunks
 - State variables addressable by name

Our Use Cases

- Trying to move from batch mentality to stream mentality
- UC 1: ETL -> Kafka -> Flink -> (external) KV Store
 - Scalable, fault tolerant, etc
 - Replace traditional ETL stack
 - Much more scalable.
- UC 2: “Given T_1 and T_2 , find all places where we have data”
 - (i.e. “Show me the gaps in a big list of integers”)
 - Historically, calculated after the data is at rest
 - Now calculated in real time and served with Flink – fast, accurate
- UC 3: “Given data at 1s resolution, buffer it into Xs blobs”
 - Minimize reads (yeah, batch...)
 - Not currently in production
 - Need QS visibility into windows to service time series requests for data being blobbed
- Future Use Case? Replace (external) KV Store with Flink / Kafka?