SIMPLIFYING YOUR LAMBA WITH KAPPA

OR: HOW I LEARNED TO STOP WORRYING AND LOVE THE STREAM

HI, I'M ADDISON

I WORK AT INSTRUCTURE

ON THE DATA AND ANALYTICS PLATFORM TEAM

A.K.A *THE BIG DATA TEAM*

SOME OF WHAT WE DO

- Run a large scale ETL job that processes ~10 TB a day
- Support teams and build tools to make data transformation easier
- Understand and try and design for future needs around data

(WE ARE HIRING BTW)

BEFORE WE DIG IN

- Any questions you have about flink/kappa you want me to answer?
- Feel free to ask questions anytime

TODAY'S TOPIC

WHY WE ARE MOVING AWAY FROM LAMBDA ARCHITECTURES AND TOWARDS KAPPA AND FLINK

THE DIFFICULTIES OF DATA

OLAP VS OLTP, NORMALIZED VS DENORMALIZED,

READS VS WRITES, RELATIONAL VS DOCUMENTS,

hierarchical vs flat, row oriented vs column oriented, etc, etc...

DATA MODELING: CHOOSE YOUR PATH

ALL PATHS ARE SCARY (IN VERY DIFFERENT WAYS)

WE INEVITABLY WILL NEED A DIFFERENT MODEL

(AND NOT WITH 24 HOURS LATENCY)

WE FACE LOTS OF CHALLENGES IN CHANGING

- The old model is still needed, which means a continual process
- Migrations with downtime often aren't an option
- Outside of many companies wheelhouse

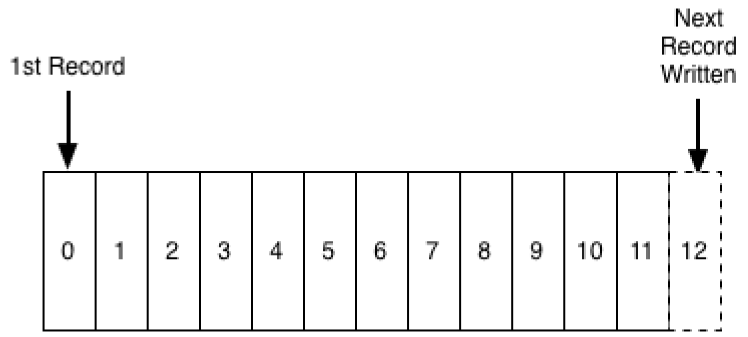
(SO WE USUALLY NEVER DO)

HOW TO STREAM

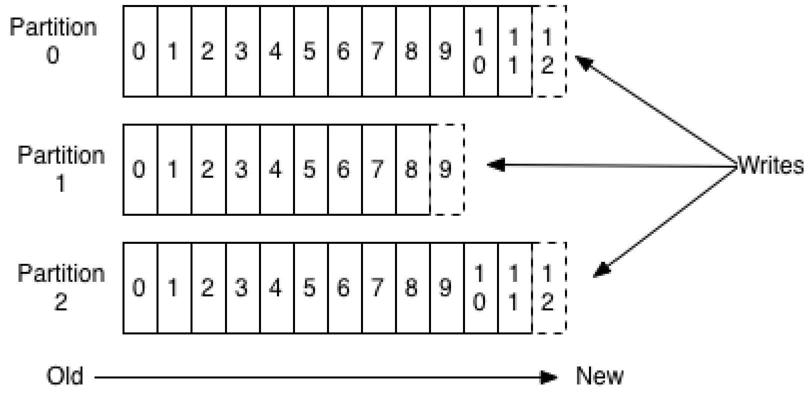
THE LOG

NOT THE PRINTLN ONES

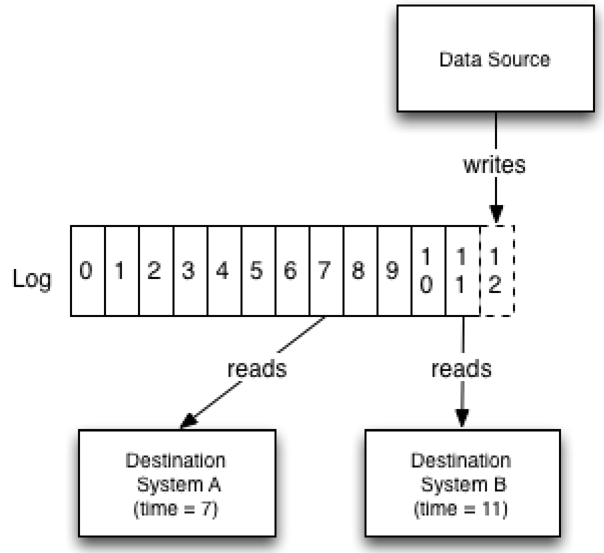
A Log is an append-only data structure ordered by time



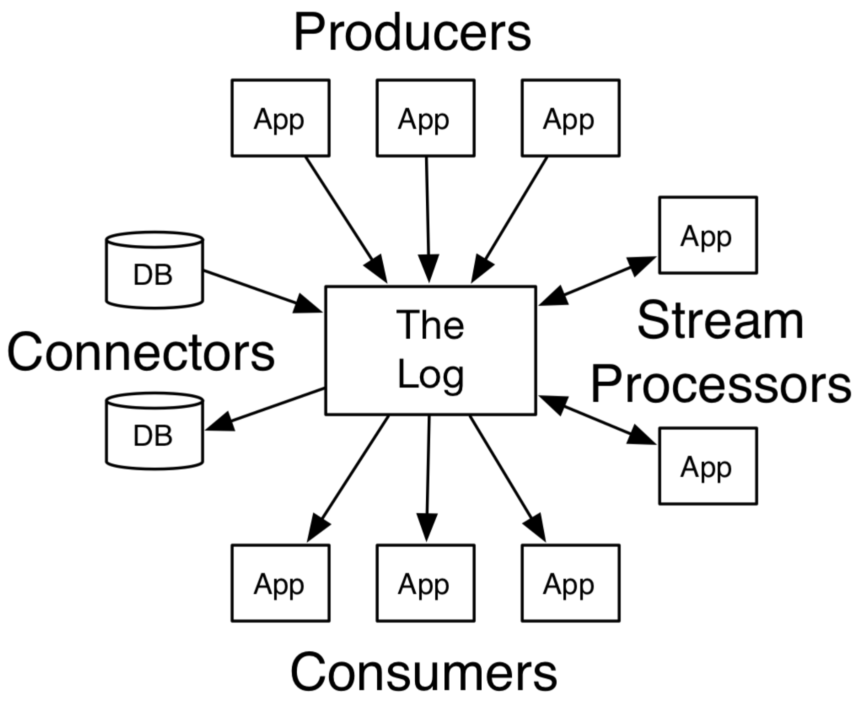
WE CAN PARTITION A LOG



IT SUPPORTS MULTIPLE READERS WHILE STILL ALLOWING WRITES


Data Source
writes
Log
0
1
2
3
4
5
6
7
8
9
10
11
12
reads
reads
Destination System A (time = 7)
Destination System B (time = 11)

IT CAN BE YOUR CORE MECHANISM FOR DATA EXCHANGE



YOU SHOULD READ THIS ARTICLE

https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

also at

https://goo.gl/et6TKf

TRANSFORMING DATA

WHAT IT NEEDS TO DO

A few main concerns:

- Scalability
- Low Latency
- Correctness
- Cost

MAKING TRADE-OFFS

It is pretty hard to do all of these

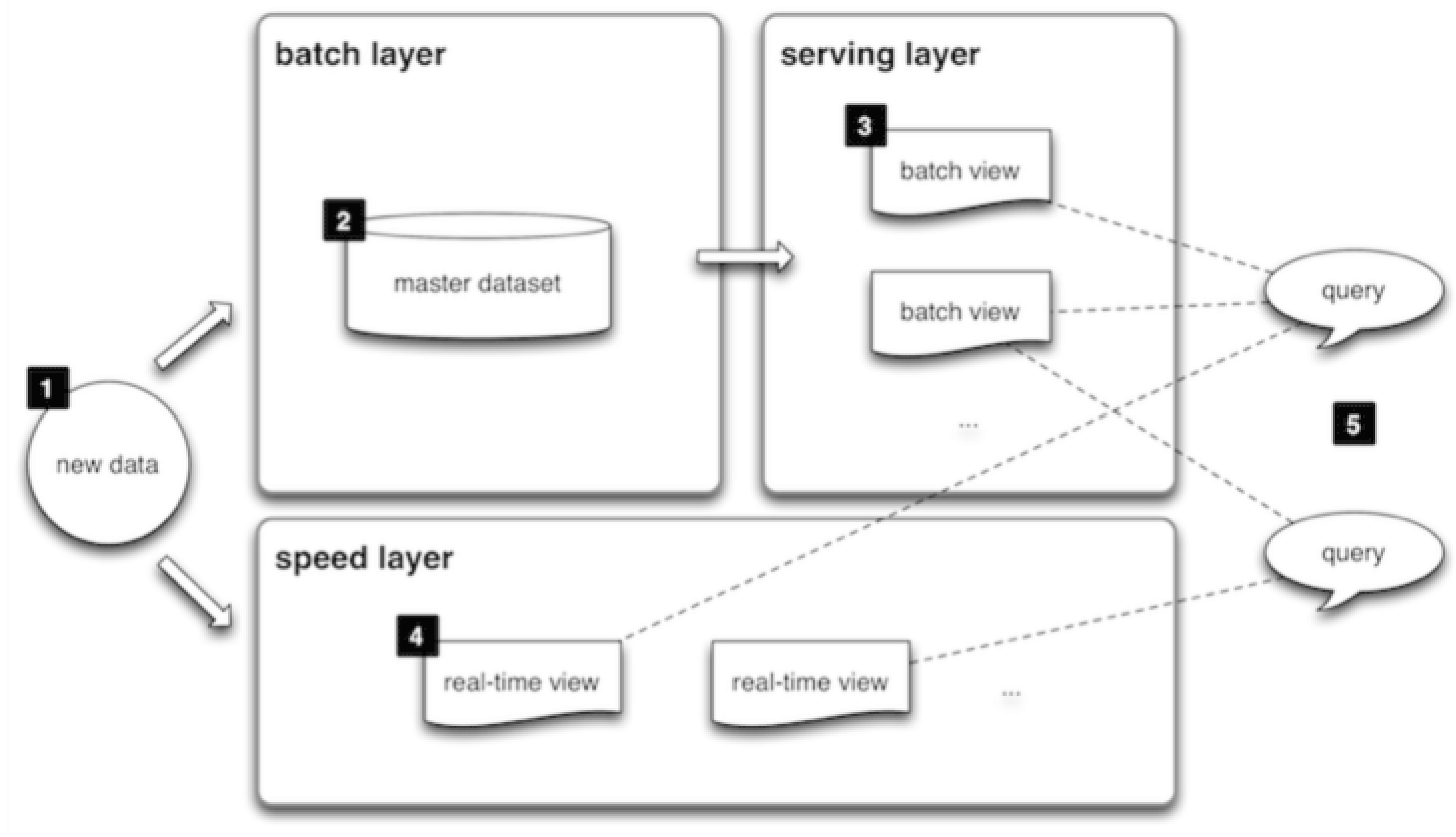
Any data transformation architecture should allow us to tune for our use case and to make trade-offs

LAMBDA ARCHITECTURE

TWO SYSTEMS IN ONE

- Batch layer - pure function that computes your view, lots of latency
- Speed layer - purpose-built stream processor to cover gap between batch layer and real-time

A DIAGRAM


batch layer
serving layer
3
batch view
2
master dataset
batch view
query
1
new data
...
5
speed layer
query
4
real-time view
real-time view
...

IS ALL THAT NECESSARY?

Two major reasons for two separate systems:

- It can be insanely hard to get accurate results in real-time
  - At-least-once semantics forces all state to be idempotent for accurate answers
  - Some aggregations require all state to compute
  - Some events may arrive out of order
- You will probably screw up and need to fix a bug which changes all data

LAMBDA ARCHITECTURE KNOBS (OR LACK THEREOF)

- We can get low latency, but likely inaccurate until batch layer reprocesses data (correctness vs cost and latency)
- Batch computation can be expensive if we have to recompute the world, incremental batch adds even more complexity (cost vs scalability)
- Not very scalable to have to build 2 systems

A NEW CHALLENGER

MAKING REAL-TIME EASIER

THE MAJOR PROBLEMS OF REAL-TIME

It's a distributed system...

1. events may arrive out of order, late, or not at all
2. machines fail (and in stream-processing often maintain state)
3. due to 1 and 2, most streaming systems have "at-least-once" message delivery semantics, this makes getting correct answers hard!

ENTER APACHE FLINK

WHAT IS FLINK?

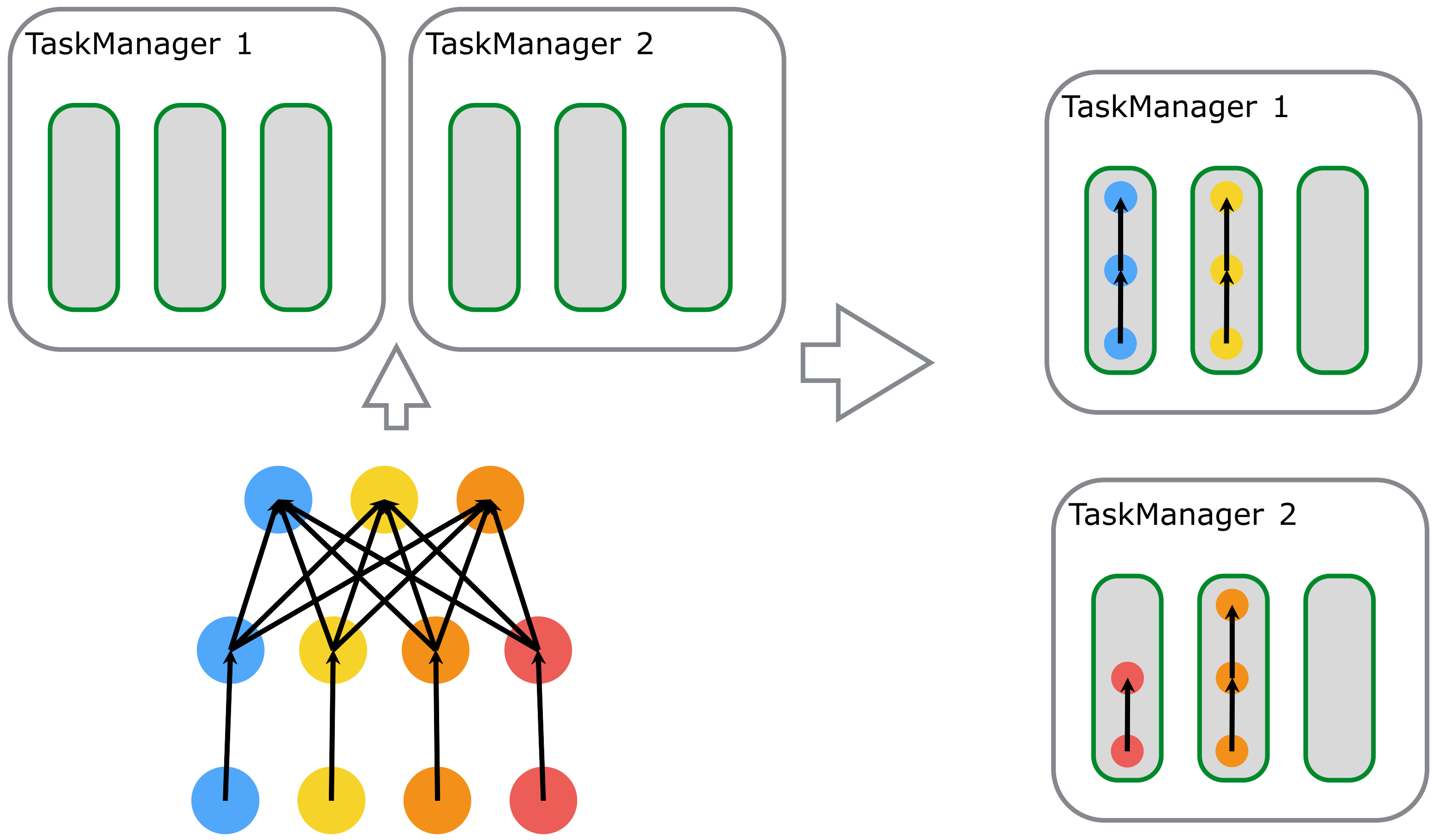
*Apache Flink® is an open-source stream processing framework for **distributed**, **high-performing**, **always-available**, and **accurate** data streaming applications.*

IN OTHER WORDS...

You get a framework for building a graph of stateful transformations that can re-distribute and shuffle data and provides an exactly-once model

It turns those into a graph of operations that get distributed across a cluster and handles fault tolerance and restarts for you

A DIAGRAM



OTHER HIGHLIGHTS

Easily wins the contest of best apache animal logo

FLINK'S SOLUTIONS TO EASY REAL-TIME PROCESSING

PROBLEM: LATE-ARRIVING EVENTS

SOLUTION: TIME TRAVELING WITH EVENT TIME

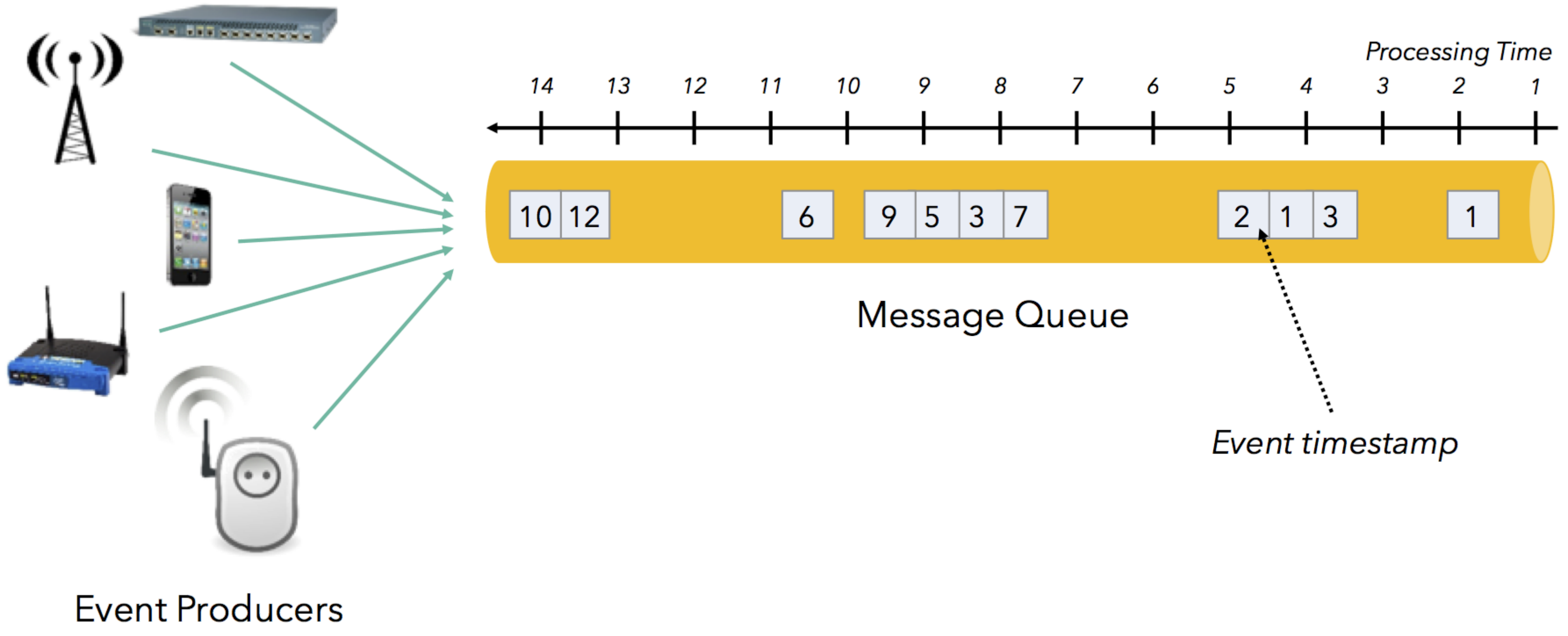
As usual, Google has a solution which is discussed in their MillWheel and DataFlow papers

Allows for us to deal with out of order and late messages

WHAT IS TIME?

- Processing Time
- Ingestion Time
- Event Time

OUT OF ORDER EVENTS



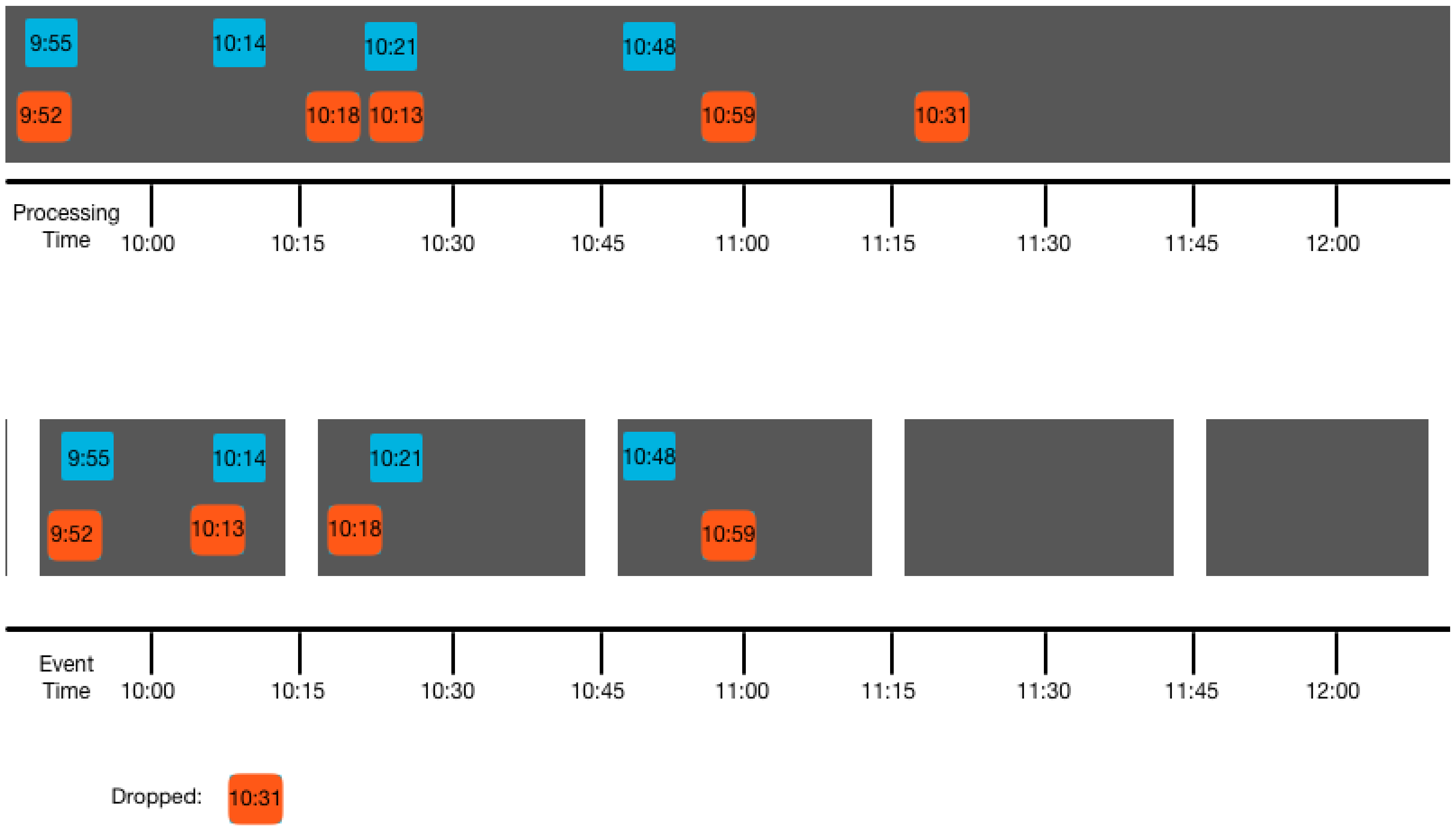
TRUST THE EVENT'S TIME

By using an event's notion of time, we can resolve out of order issues using "windows"

AN EXAMPLE

Event Time Example

10 Minute Windows, allow 10 minute late events



PROBLEM: FAULT TOLERANCE AND SCALABILITY

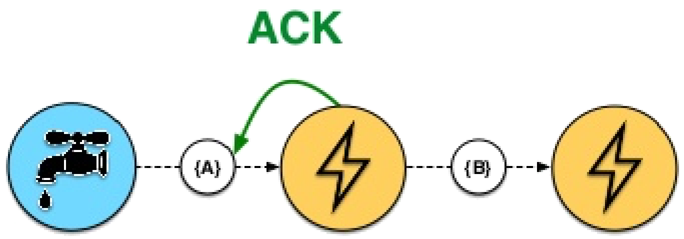
SOLUTION: DISTRIBUTED CHECKPOINTS

MESSAGE GUARANTEES

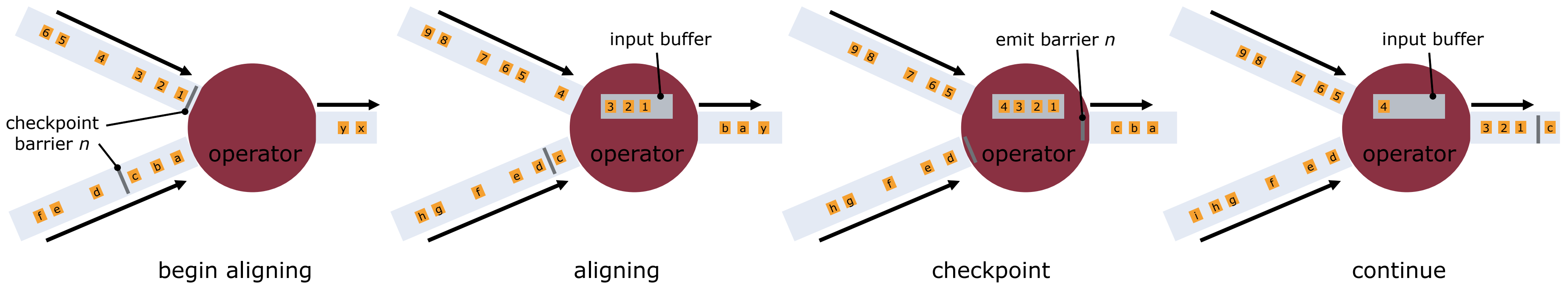
- **at-most-once**: you get the message either once, or not at all... this is can be useful when you don't care about correctness or consistency
- **at-least-once**: you get the message one or more times, useful if you need to process all messages, but makes some aggregations very difficult
- **exactly-once**: you will receive each message once and only once. This is what we want... but is it possible?

PAST STREAM SYSTEMS

Earlier systems like Apache Storm could offer at-least-once message guarantees through per message ACKing but it came at a high cost of low throughput



CLEVER SNAPSHOTS



PROBLEM: STATE IS HARD TO REASON ABOUT WITH AT-LEAST-ONCE

SOLUTION: EXACTLY-ONCE PROCESSING MAKES LIFE EASIER

ACCURATE RESULTS FOR MORTALS

Without exactly-once processing, we need to think harder about state

An example: count number of quiz submissions for a given quiz

SOME EXAMPLES

BASIC STREAMING

```
case class WordWithCount(word: String, count: Long)
val windowCounts = text
      .flatMap { w => w.split("\\s") }
      .map { w => WordWithCount(w, 1) }
      .keyBy("word")
      .timeWindow(Time.seconds(5))
      .sum("count")
```

MANAGED STATE

Flink's distributed snapshots can include user-defined state

In event of failure, your state rolls back to be consistent with the stream position

Allows for *logically* exactly-once processing

When paired with kafka transactions, you can get exactly-once across multiple jobs

STATEFUL EXAMPLE

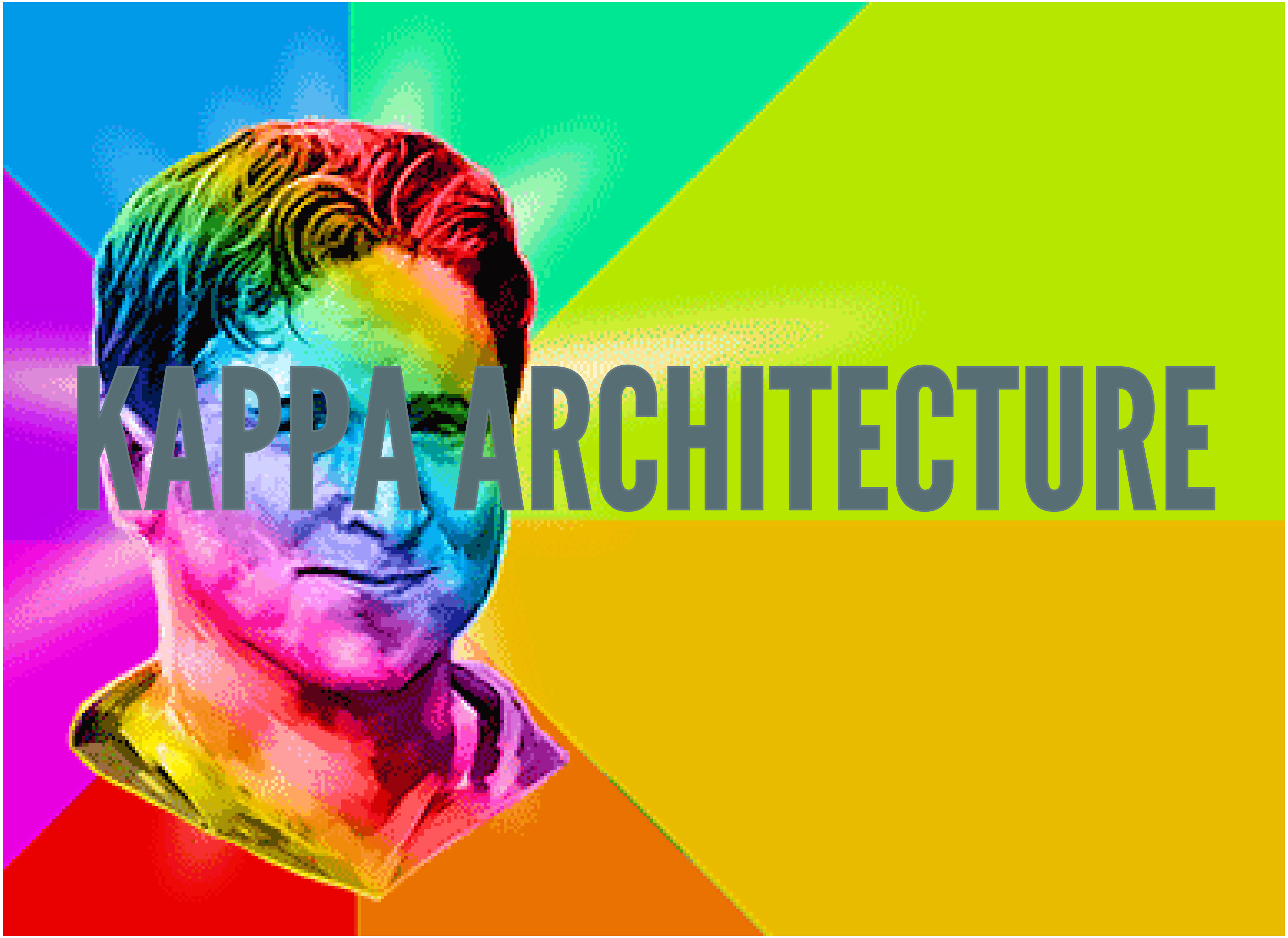
```
case class WordWithCount(word: String, count: Long)
val windowCounts = text
  .flatMap { w => w.split("\\s") }
  .map { w => WordWithCount(w, 1) }
  .keyBy("word")
  .mapWithState[WordWithCount, Long] {(w, state) =>
    val wordSum = state match {
      case None => WordWithCount(w.word, 1)
      case Some(oldCount) => WordWithCount(w.word, oldCount + 1)
    }
   (wordSum, Some(wordSum.count))
  }
```

WHAT IS FLINK REALLY?

A stream processor that:

1. is reliable and accurate with low programmer overhead
2. has really great performance
3. is pretty swell

KAPPA ARCHITECTURE



WHAT IS IT?

Use a log and a stream processor (like Flink) and write the results somewhere

That's it.

BETTER THAN LAMBA?

For a lot of things, yes!

With sufficiently powerful real-time tools, we get correct streaming results for most transformations

Which means we don't need a batch layer for consistency (woohoo!)

WHAT ABOUT MISTAKES?

One thing lambda architecture solves for is bugs and screw-ups

What about kappa architectures?

USE THE LOG

With a sufficiently long log (like back to the beginning of time), we just rewind the log to the beginning and start over

This also means consistency is easy as we don't need to coordinate between batch and speed layers

A SILVER BULLET?

Not quite...

Some things are still hard to do in real-time and some transformations might require a ton of state

But you should still explore it as it's almost always easier than a lambda architecture

THE FUTURE

FOR US, IT IS FLINK

- Sane model to give programmers
- Much easier to build one system
- Scales to our needs

THANKS!

QUESTIONS?