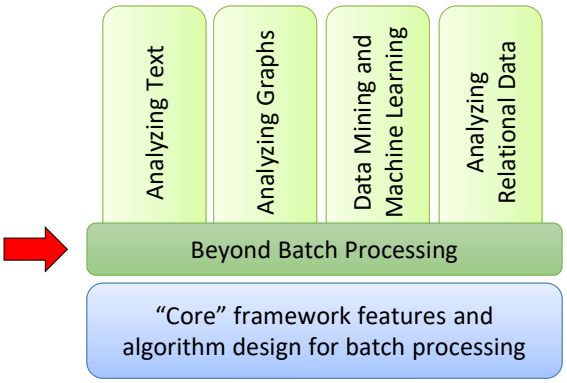
Data-Intensive Distributed Computing CS431/451/631/651

Module 8 – Beyond Batch Processing

Visual : Streaming

Structure of the Course

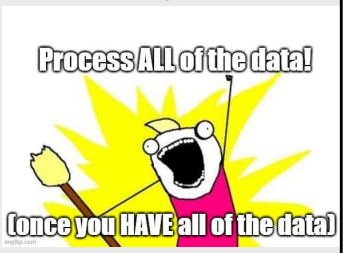


When you're out of green bars, simply make more.

Batch Processing vs Stream Processing

Batch Processing

- All The Data
- Not Real Time
- Long Wait



Stream Processing

- Process data as it arrives
- Real Time (ish)
- Low Latency

Remember Business Intelligence?

The analysists in my diagram said “meh” to stale data, but what about THIS

What does a BI Analyst Do?

Generate Reports

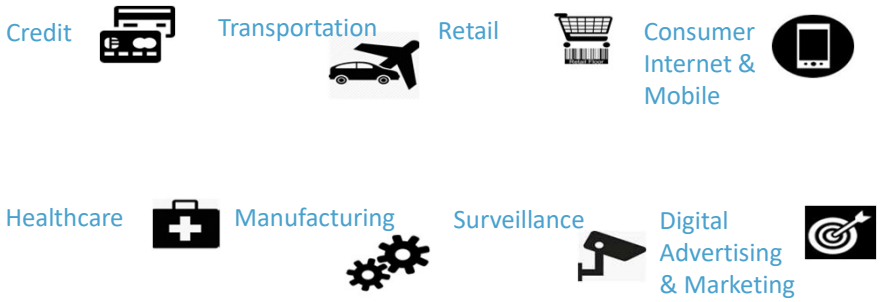
Create Monitoring Dashboards

Ad hoc analyses

- Descriptive – extract a description of the data
- Predictive – extract a model of the data that will predict future data

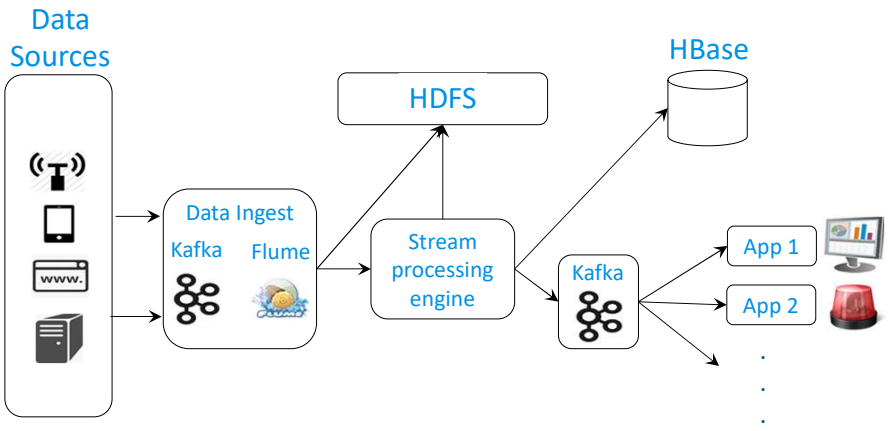
Should be real-time!

Use Cases Across Industries



Credit cards – Identify fraud / identity theft
Transportation – Rerouting vehicles – weather delays, road closures, detours, airport delays, runway closures…
Retail – Inventory management. In-store recommendations / offers (PC app for example?) e-commerce recommendations
Internet / Mobile – user engagement based on current behaviour
Healthcare – patient stats, identify at-risk patients, real-time emerg waiting times (GRH's has been offline for MONTHS ☹)
Manufacturing – equipment maintenance, identify failures and react instantly and automatically
Surveillance – realtime threat assessment, intrusion detection,
Advertising – optimize targeting advertising based on real-time data

Typical Datastream Pipeline



Huh? What are all those things?

Kafka – A distributed pub-sub broker designed for resilience and availability
Flume – A distributed log aggregator (designed for resilience and availability)
Stream-Processing-Engine – The top of this module
Hbase – Distributed key-value datastore, modelled after Google's BigTable and built on top of HDFS

So pixelated

Moses parting the data sea

So a data stream flows into a data lake?

…Yes

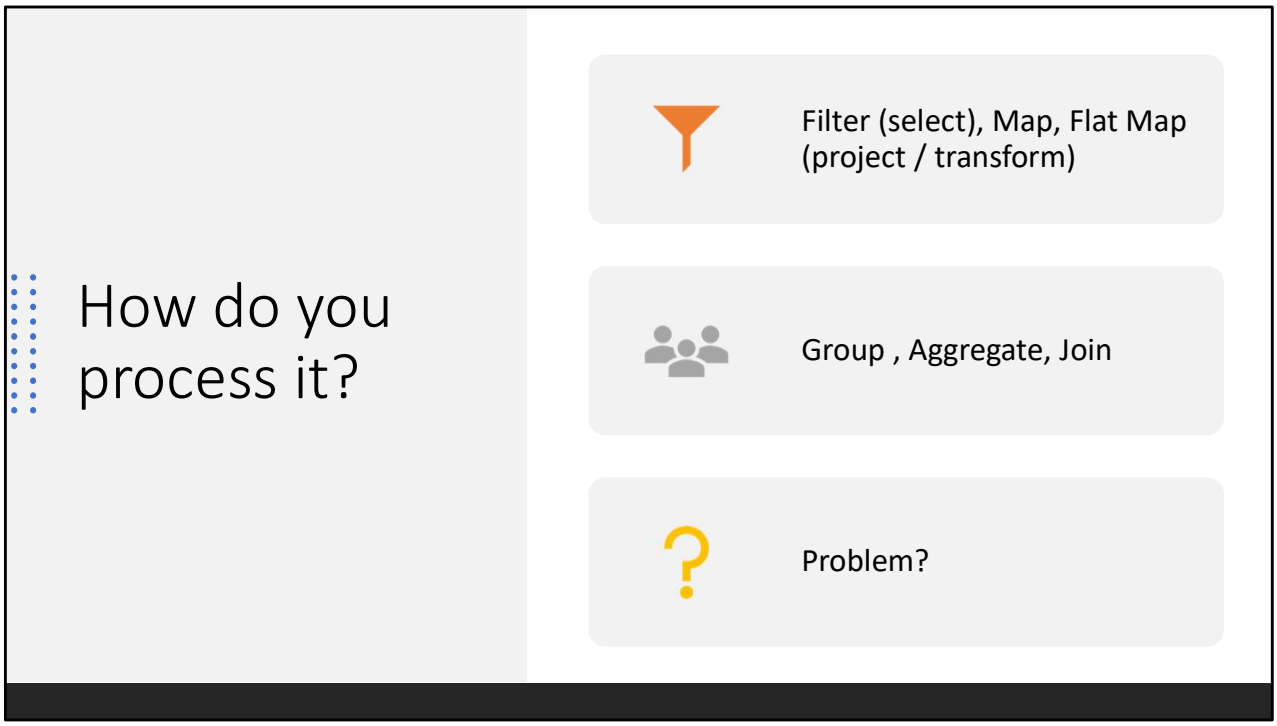
Does this mean there is also data rain? When data in the cloud condenses and…

Not yet, but, be the change you want to see in the world

But what IS a data stream?

A sequence of items (tuples)

- Structured
- Ordered (either a timestamp, or implicitly by arrival time)
- Continuously arriving
- High volume
  - Might not be possible to store all of it
  - Might not be possible to even examine all of it



The problem is that the “reduce-like” tasks – grouping, aggregating, and joining – rely on having all of the data.
How can we define them “continuously?”

Problems in Semantics

Aggregation / Grouping	Joining Stream to Static Data	Joining Stream to Stream
• When do you start? When do you stop?	• Easy Lookup, not a problem.	• How long do you wait for the corresponding key? • When do you stop joining?

Windows

Solution: Define all “reduce-like” transformations for a given window

- Based on ordering attribute (timestamp)
- Based on counts (last X records)
- Based on explicit markers
  - a.k.a. Punctuation

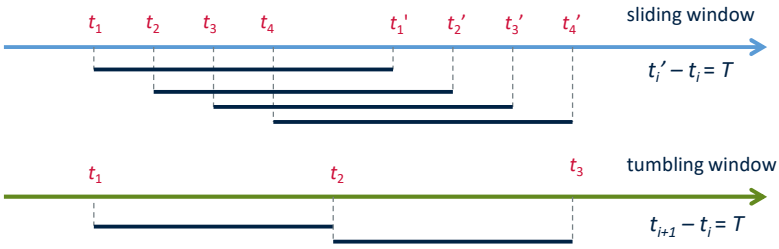
Sliding vs Tumbling

- Sliding
  - Last minute of data, updated every 5 seconds
  - Each window has most data in common with previous
  - Can your operation be reversed?
    - Can you easily remove values and add others without a full recompute?
- Tumbling
  - Last minute of data, updated every minute
  - Blank slate, no data in common between windows
  - You might be able to make a sliding window out of smaller tumbling windows!

Windows on Ordering Attributes

Assumes the existence of an attribute that defines the order of stream elements (e.g., time)

Let *T* be the window size in units of the ordering attribute



Windows Based on Counts

Windows of N records (sliding or tumbling) over the stream

Bursts:

- When stream is slow, windows are LONG
- When stream is fast, windows are very short

Windows Based on “Punctuation”

Sending Application inserts “End of Batch”

- UI Analytics, sends “End of Session” when user closes tab

PRO: application controls the semantics

CON: unpredictable window size (both time and number of records)

What does “application controls the semantics” mean?
In the context of analytics, a window of a single user session is PERFECT. Impossible to achieve through other methods. You can use time-based windows drawn from your “median session duration” statistics but that’s far from ideal.

Of course, not every sort of stream lends itself to inserting this kind of punctuation.

Stream Challenges

Inherent Challenges

- Latency Requirements
- Memory / Storage
  - Big Data are Big

Framework Challenges

- Bursty nature of streams
- Load Balancing / Clustering
- Out-of-Order delivery
- Consistency Choices
  - At most once
  - Exactly once
  - At least once

Consistency choices

At most once: Every consumer will get a value from a producer at most once (but might not get it at all) -- easiest
Exactly once: Every consumer will get every value from a producer, and will never get any duplicates -- hardest
At least once: Every consumer will get every value from a producer, but may receive duplicates. Might be harmless if consumer can ignore duplicates. (Easy)

Two Hard Problems with Distributed Delivery

2. Exactly-Once Delivery
1. Out-of-Order Delivery
2. Exactly-Once Delivery

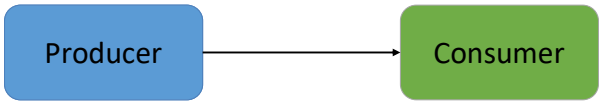
Producer/Consumers

How do consumers get data from producers?

Producer – Consumer model
Produces generate / gather / aggregate data
Consumers process it.

(A consumer might also be a producer, e.g. it's processing raw data and producing aggregated date for use in a dashboard, etc.)

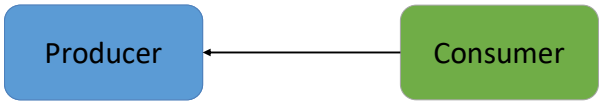
Producer/Consumers



Push / Callback method

Producer sends to consumer

Producer/Consumers

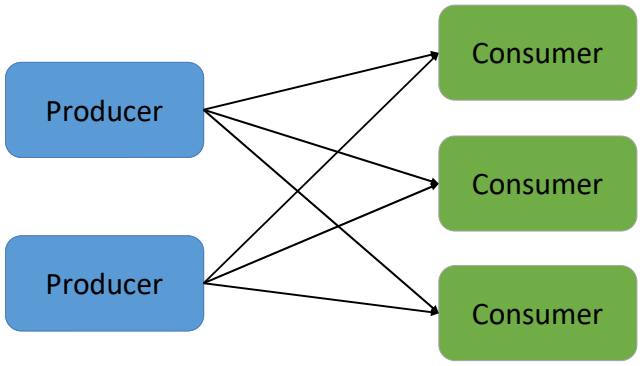


Pull / Poll method

Consumer requests data from producer

See also `tail –f` to do a polling stream from a UNIX file

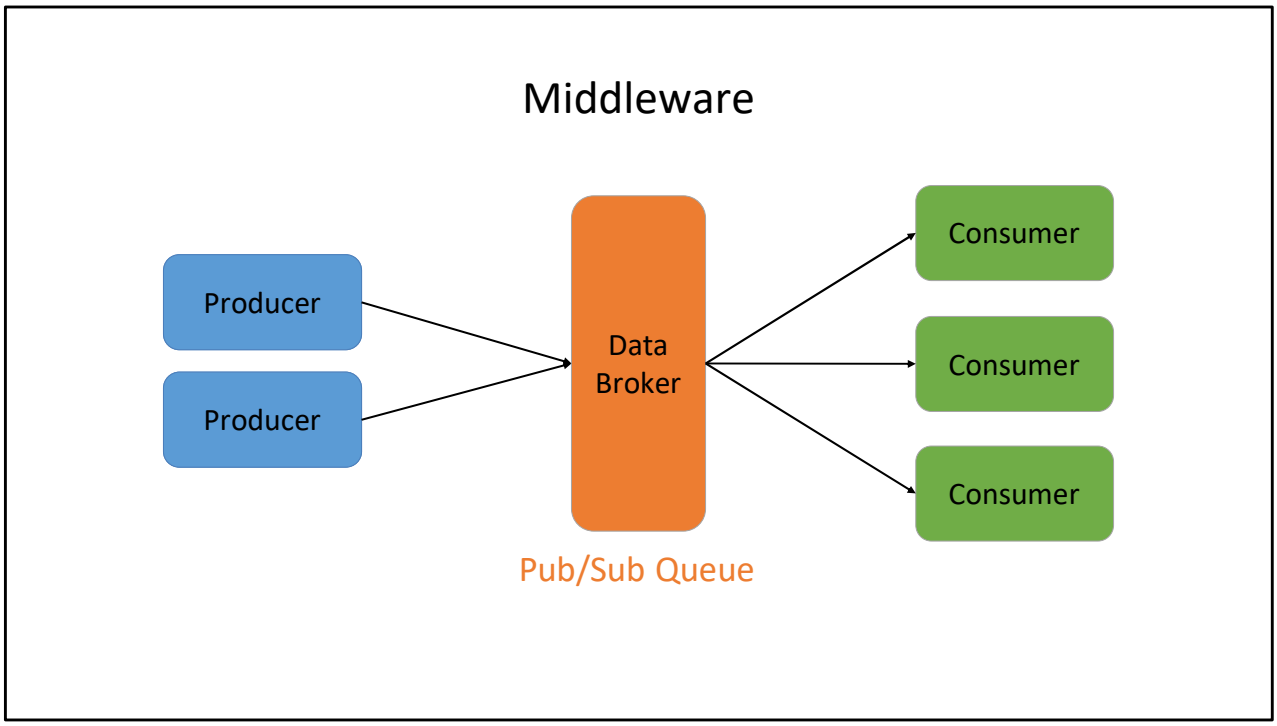
Producer/Consumers



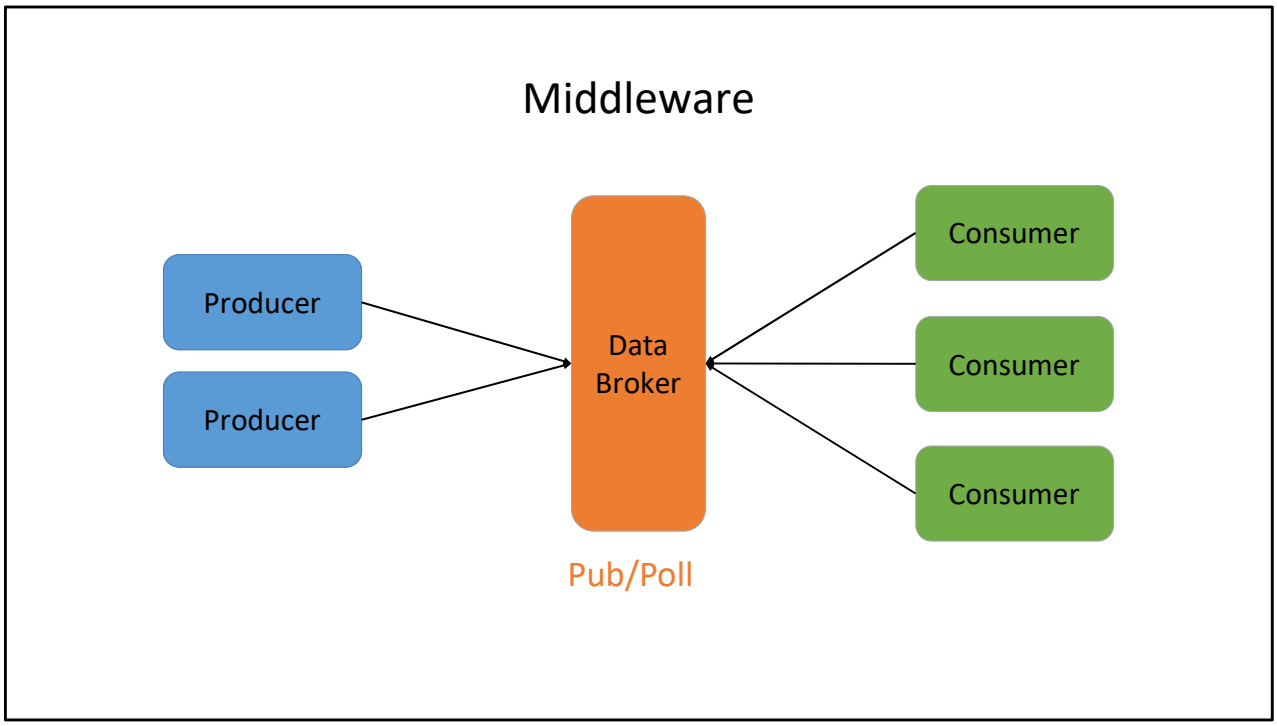
A consumer might be consuming data from multiple producers, and a producer might need to send to multiple consumers

Many:Many relationship in other words

How can you scale this to hundreds of consumers and dozens or producers?
How can you scale this even higher???



Producers push to the broker, broker pushes to consumers…



You could also have a push pull model: produces push to broker, consumers pull from the broker as needed

Advantage to consumer polling: Broker doesn't need to worry about spotty connections by consumers. When your phone has internet again, it will ask for updates.
Disadvantage – frequent polling for infrequently arriving data is wasteful

MQTT uses push-to-consumer
Kafka uses pull-from-broker

Pub/Sub at Dan's House

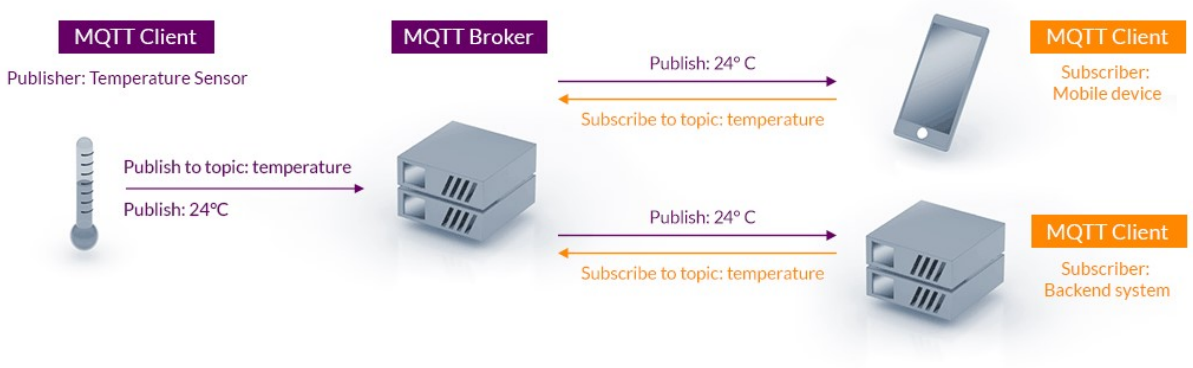


Image: from mqtt.org

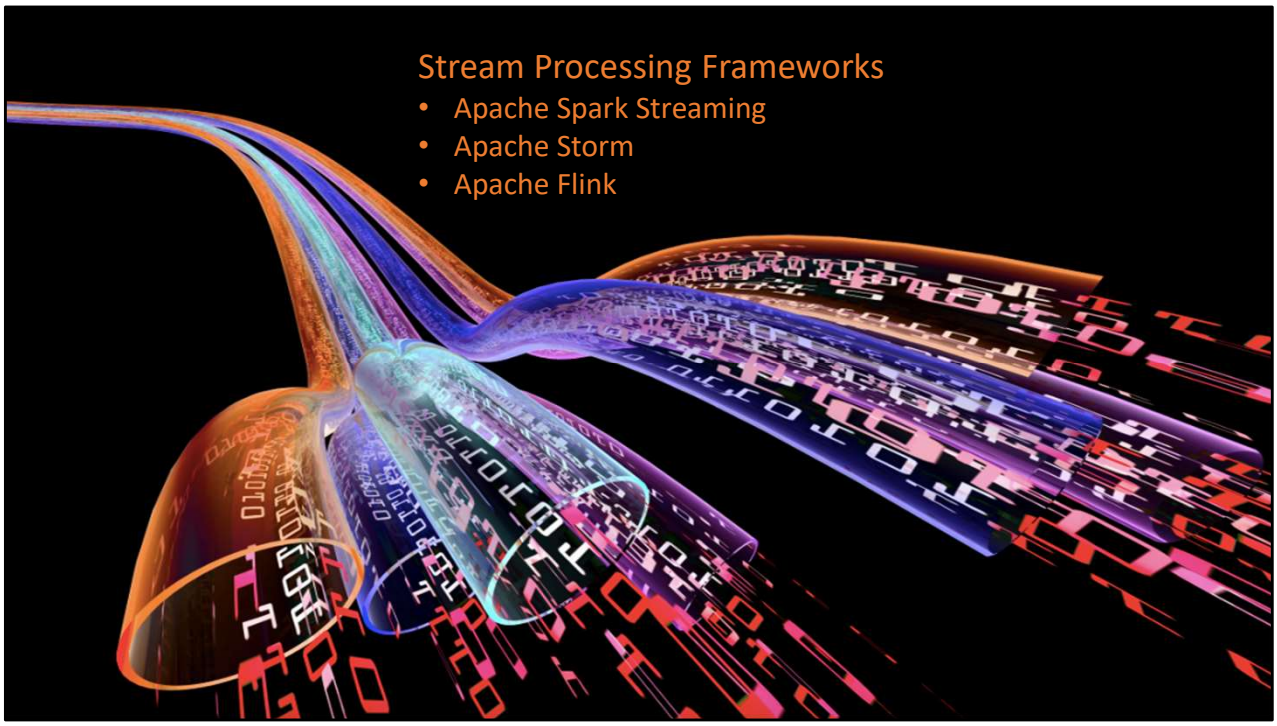
I've got a bunch of ESP8266 / ESP32 IoT thingies in: garage, attic, crawlspace, living room, deck that log temperature, humidity, air pressure (just the outdoor one)
They all push to an MQTT broker. The living room unit has an LCD display, so it subs to the deck sensor to display Inside and Outside temps.

The security system also pushes things like 'garage door open', 'living room motion sensor' to the MQTT broker.

HomeAssistant subscribes to these subjects, and can show house temperatures, security alerts.
The weather subjects also have a subscriber that tosses them into a postgres database. (A quick little C program I wrote that SEEMS to work)

MQTT is a very lightweight protocol meant for IoT applications. It's not as resilient as needed for Big Data…

Anyways…


Stream Processing Frameworks
• Apache Spark Streaming
• Apache Storm
• Apache Flink


Spark
Streaming

Spark Streaming: Discrete Streams (DStream)

- Tumbling Window with very small duration (typically 1 second)
  - As mentioned, you can make a sliding window out of a bunch of tumbling windows
- Every t seconds (usually t=1) collect all data into an RDD
- Process RDD through Spark Engine like any other RDD

Image source: spark.apache.org

Getting Started: Spark Streaming Context

Usually a variable called ssc.

- Specify the time slice when creating. Default is 1 second

ssc can be used to create DStreams

A DStream has the RDD transforms

It also has its own transforms and actions

The RDD transforms create a new Dstream. Each RDD "packet" in the source is transformed using the RDD transform. These form a new DStream

The DStream specific transforms tend to transform the entire stream.

Example: Tweet Streams

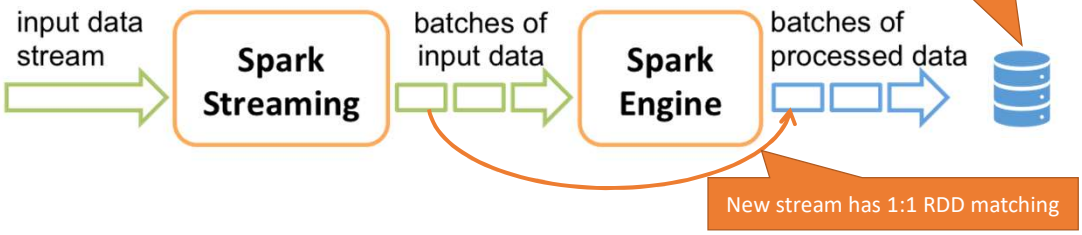
One of many stream constructors

Custom Receiver Class

```
tweets = ssc.receiverStream(TweetReceiver(username,password))
hashtags = tweets.flatMap(getTags)
hashtags.saveAsHadoopFiles(…)
```

flatMap applied to each batch RDD

RDDs saved to HDFS as they're created



New stream has 1:1 RDD matching

Key Concepts

DStream – sequence of RDDs representing a stream of data

Twitter, HDFS, Kafka, Flume, TCP sockets

Transformations – modify data from on DStream to another

Standard RDD operations – map, countByValue, reduce, join, …

Stateful operations – window, countByValueAndWindow, …

Output Operations – send data to external entity

saveAsHadoopFiles – saves to HDFS

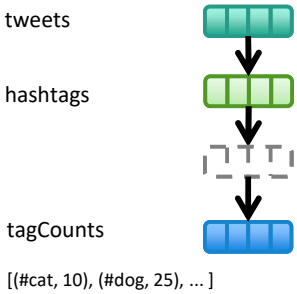
foreach – do anything with each batch of results

You can also output back to Kafka, etc.

Example: Tweet Streams

```
tweets = ssc.receiverStream(TweetReceiver(username,password))
hashtags = tweets.flatMap(getTags)
tagCounts = hashTags.countByValue()
```

Produces pair RDD: (value, count)



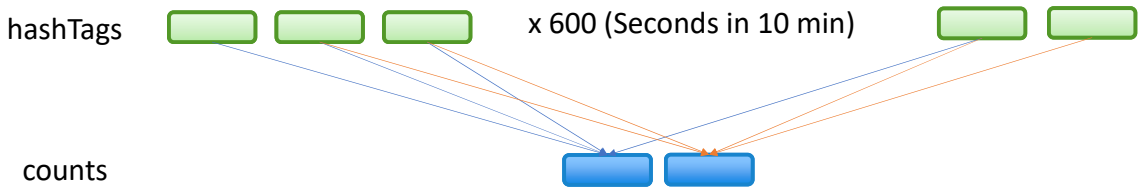
Repeat for each batch

Problem?

The problem is that this is only the hashtags for the last SECOND. That's a very small window.

Example: Tweet Streams

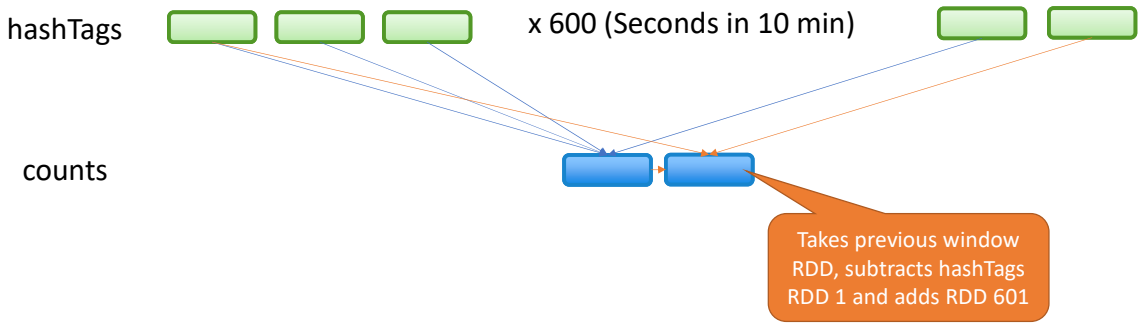
```
tweets = ssc.receiverStream(TweetReceiver(username,password))
hashtags = tweets.flatMap(getTags)
counts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



The first count RDD depends on the first 600 hashtag RDDS (call them 1 through 600). The second count depends on hashtag RDDs 2 through 601

Stream Smarter, not Harder

```
tweets = ssc.receiverStream(TweetReceiver(username,password))
hashtags = tweets.flatMap(getTags)
tagCounts = hashTags.countByValueAndWindow(Minutes(10), …)
```



Much more efficient!

Smart window-based reduce

Incremental counting generalizes to many reduce operations

Need a function to "inverse reduce" ("subtract" for counting)

```
val tagCounts = hashtags
    .countByValueAndWindow(Minutes(10), Seconds(1))
```

```
val tagCounts = hashtags
    .reduceByKeyAndWindow(_ + _, _ - _, Minutes(10), Seconds(1))
```

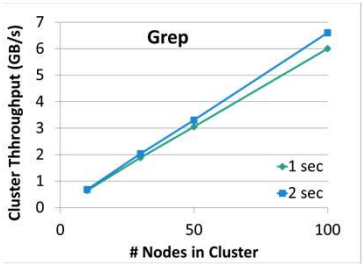
```
tagCounts = hashtags
    .reduceByKeyAndWindow(lambda x,y:x+y, lambda x,y:x-y,
Minutes(10), Seconds(1))
```

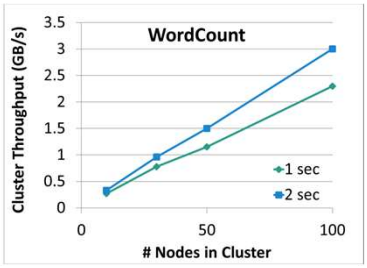
Last two: Scala vs Python

Performance

Can process **6 GB/sec (60M records/sec**) of data on 100 nodes at **sub-second** latencyTested

- with 100 streams of data on 100 EC2 instances with 4 cores each

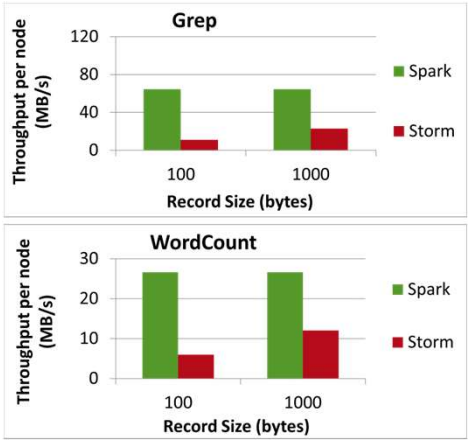




Comparison with Storm

Higher throughput than Storm

- Spark Streaming: 670k records/second/node
- Storm: 115k records/second/node



Data Brokers

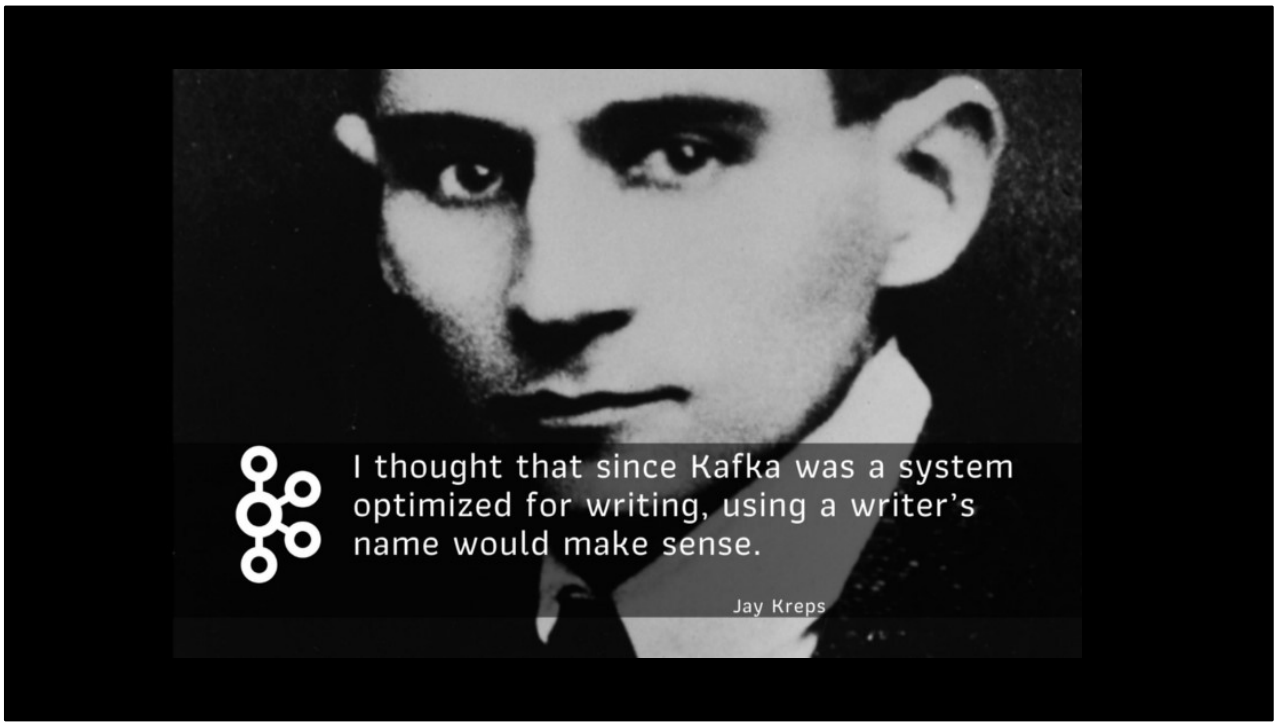
- Two popular ones are Kafka and Flume
- Flume is specifically for log aggregation, not general purpose
- Kafka is general purpose
- MQTT is lightweight for IoT uses. Not suitable for big data streaming
  - MQTT broker can collect data from IoT and forward to a Kafka broker


APACHE
kafka®

Kafka?

- http://kafka.apache.org/
- Originated at LinkedIn, open sourced in early 2011
- Implemented in Scala, some Java

Jay Kreps chose to **name** the software after the author Franz **Kafka** because it is "a system optimized for writing", and he liked **Kafka's** work.


I thought that since Kafka was a system optimized for writing, using a writer's name would make sense.
Jay Kreps

Kafka adoption and use cases

- **LinkedIn:** activity streams, operational metrics, data bus
  - 400 nodes, 18k topics, 220B msg/day (peak 3.2M msg/s), May 2014
- **Netflix**: real-time monitoring and event processing
- **Twitter**: as part of their Storm real-time data pipelines
- **Spotify**: log delivery (from 4h down to 10s), Hadoop
- **Loggly**: log collection and processing
- **Mozilla**: telemetry data
- Airbnb, Cisco, Uber, …

https://cwiki.apache.org/confluence/display/KAFKA/Powered+By

How fast is Kafka?

- **"Up to 2 million writes/sec on 3 cheap machines"**
  - Using 3 producers on 3 different machines, 3x async replication
    - Only 1 producer/machine because NIC already saturated

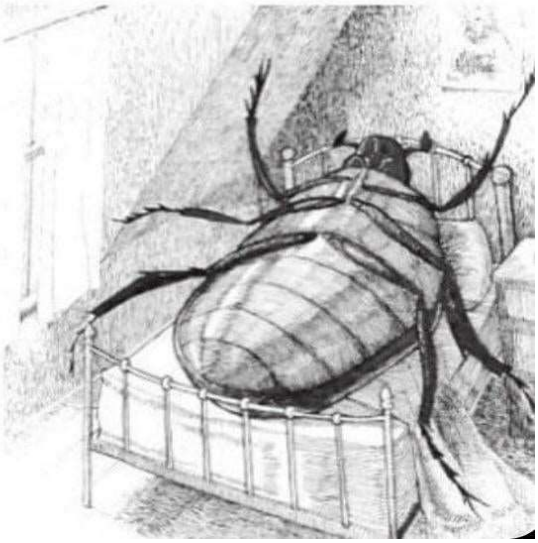
Why is Kafka so fast?

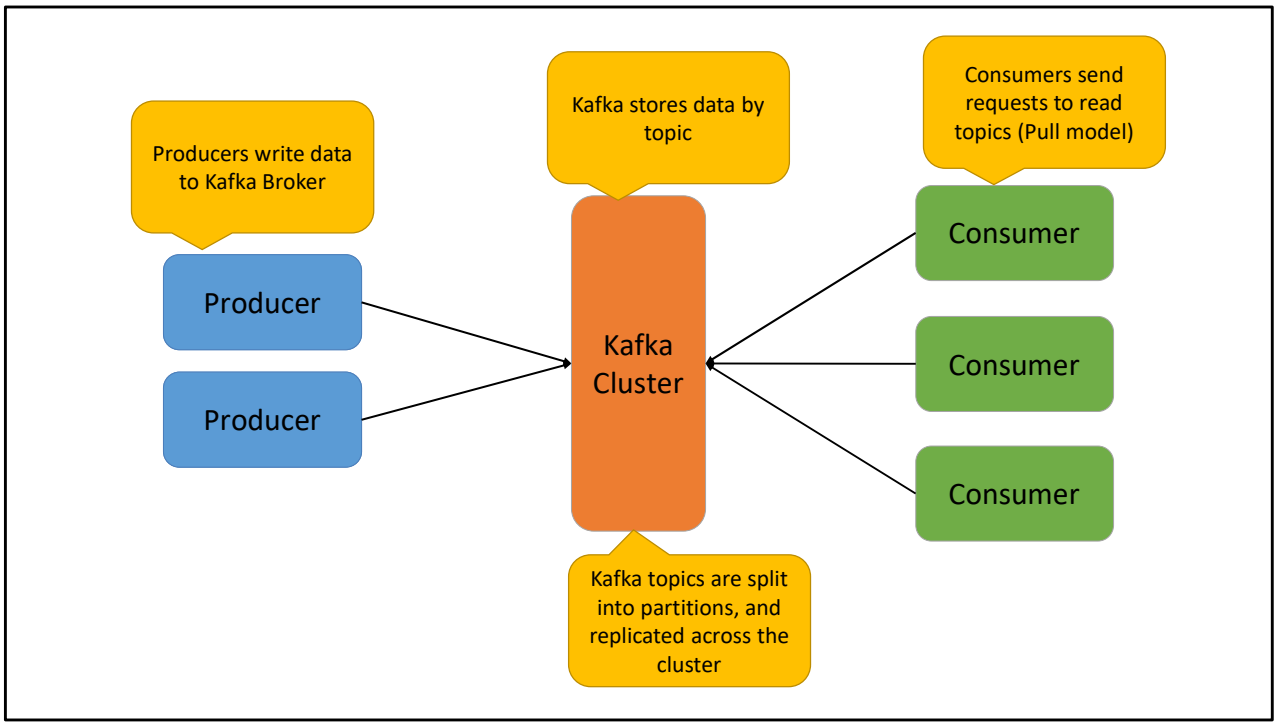
- Fast **writes**:
  - While Kafka persists all data to disk, essentially all writes go to the **page cache** of OS, i.e. RAM.

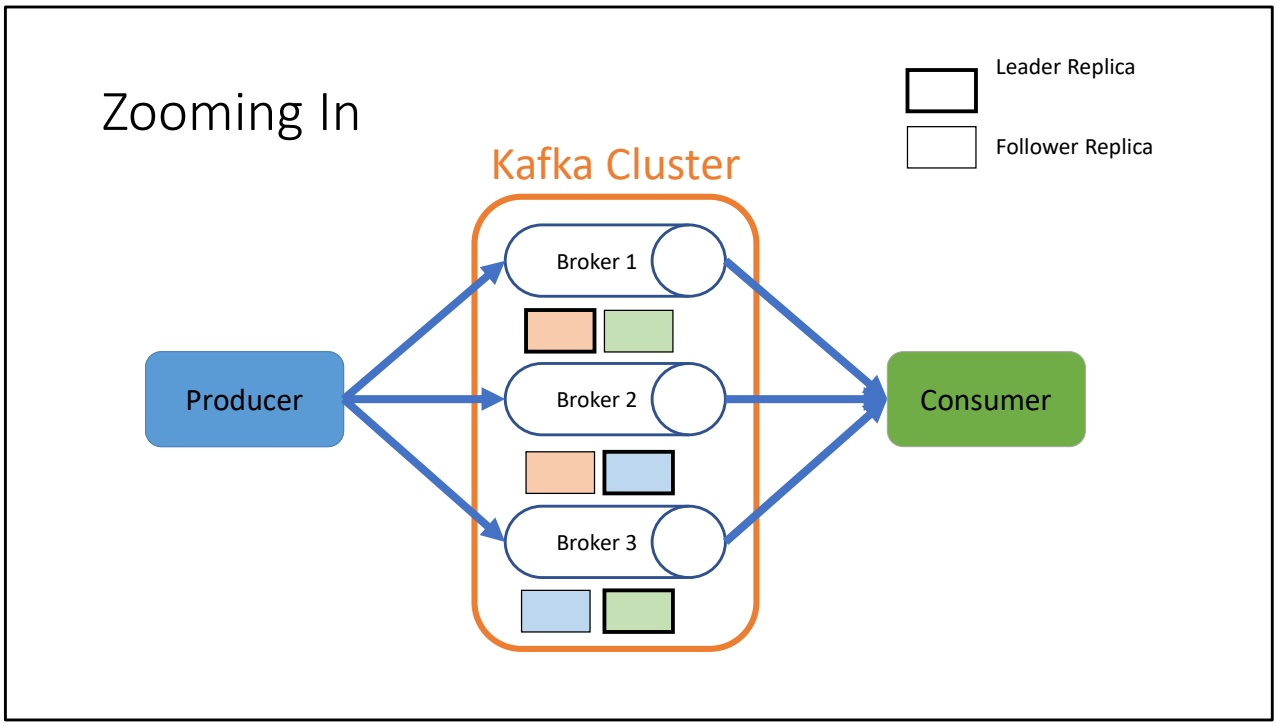
- Fast **reads**:
  - Very efficient to transfer data from page cache to a network **socket**
  - Linux: **`sendfile()`** system call

  Example (Operations): On a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks as they will be serving data entirely from cache. http://kafka.apache.org/documentation.html#persistence

ARE YA STREAMING KAFKA,




Producers write data to Kafka Broker
Kafka stores data by topic
Consumers send requests to read topics (Pull model)
Producer
Producer
Kafka Cluster
Consumer
Consumer
Consumer
Kafka topics are split into partitions, and replicated across the cluster



This diagram is for a specific topic, e.g. "/sensors/temperature"

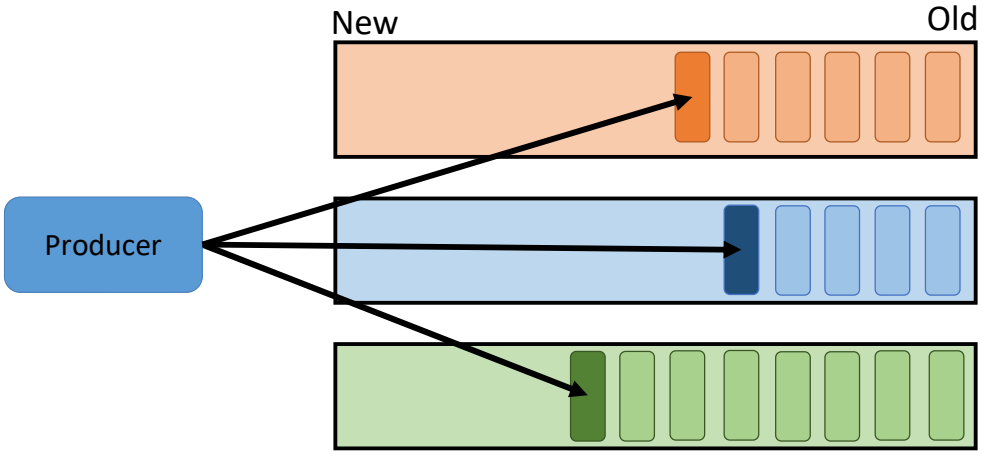
The producer will send messages (events in Kafka terminology) to the cluster, and they will be divided among the partitions.
For redundancy, each partition is replicated on other nodes in the cluster.

In this diagram, Broker 1 is the leader for partition Orange, and Broker 2 is a follower for Orange. Messages that Producer writes to Orangeare sent to Broker 1, which will forward them to Broker 2 to replicate

If Broker 1 goes offline, Broker 2 becomes the leader until Broker 1 is back online.

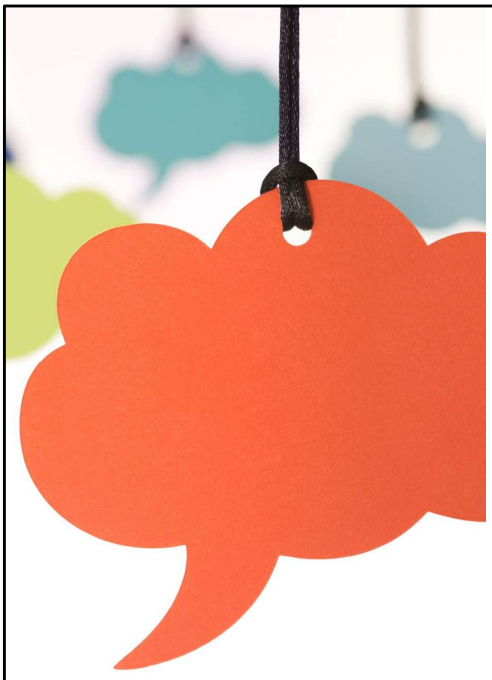
This diagram shows 3 brokers with 2x replication (R=2). A Kafka cluster can tolerate R-1 failures without data loss (in this case, any one Broker can go down without loss).

Writing to a partition



Messages from the producer are hashed (if the messages have a key) or balanced round-robin and appended to the partitions

(Think appending to files. Some docs refer to the partitions as “log files”)



Write Exactly Once

Option: **Idempotence** Cool word.

Producer sends a unique key with each batch of messages.

Brokers log these and reject duplicate messages

An idempotent operation is one that if applied multiple times, has the same effect as if it was only applied once.

Since producers should be sending batches of messages anyway, the overhead of one extra UUID is per batch is quite low. (Apache claims it's negligible)

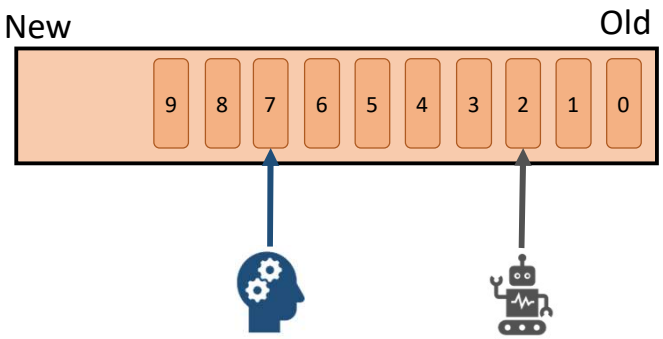
To Key or Not to Key

- Partitions are totally ordered, but topics are not
- If your events have a key then all events with the same key are in the same partition, and therefore are in order by arrival time
- If your events have no key, they are not in exact order anymore
- Takeaway: If you need totally ordered arrival, use a key!
  - Warning: If the key isn't good, the partitions won't be even ☹

Reading a Partition

A partition can be read by multiple consumers

Each consumer remembers its offset in each partition

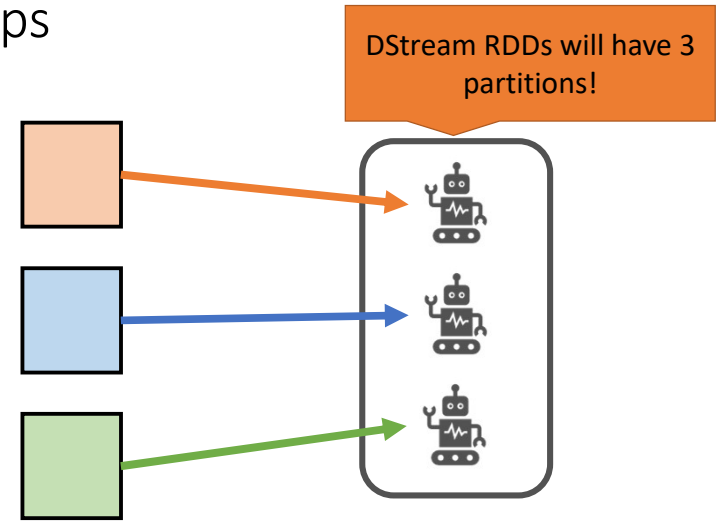


Consumers are responsible for their own bookkeeping. Kafka doesn't remember where the blue gearhead was at. If it asks for message 7 in partition Orange, that's what it gets! If a consumer crashes, it can ask for messages it's already "seen". Also allows it to recompute values if, e.g., a lookup table was wrong, there was a bug in the code. All kinds of reasons to rewind time.

Consumer Groups

Consumers can be grouped.

A consumer group gets all messages for a given topic, but each consumer only gets one partition's worth



Sound familiar? Sounds like if you have a topic with 6 partitions, then your Spark Stream would have 6 partitions per RDD? Yes!

The DStream would form a consumer group for that topic, and assign 1 RDD Partition (Task) per Kafka Partition

Read Exactly Once

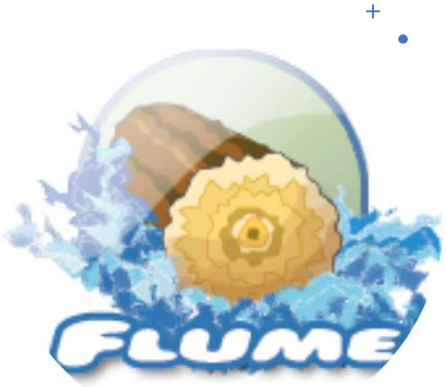
Consumer is given a batch of messages, including the index range

- Consumer should remember the range so it knows how to ask for the next ones
- Consumer should remember the range so it can discard duplicates


WHEN YOU'RE TALKING ABOUT KAFKA
AND SOMEONE SAYS "OH, I LOVE HIS WORK!"
imgflip.com

Apache Flume

- Distributed service for collecting, aggregating, and moving large amounts of log data
- Usually aggregates and writes to HDFS, but you can have other pipeline setups
- We won't use it, so this is the only slide it gets.



A flume is a chasm or trench with a stream in it. See, because loggers would dig a trench down a mountainside, and you put logs in it, and ZOOM, off they go. This is also where log rides came from. Sun's setting, take your final log and ride it down the log flume! Fun / deadly. I'm just assuming.

Anyways, it shouldn't need saying, but a FLUME is a LOG STREAM. Basically. This doesn't even count as a metaphor, it's literally what it is. Figuratively literally. Best project name.

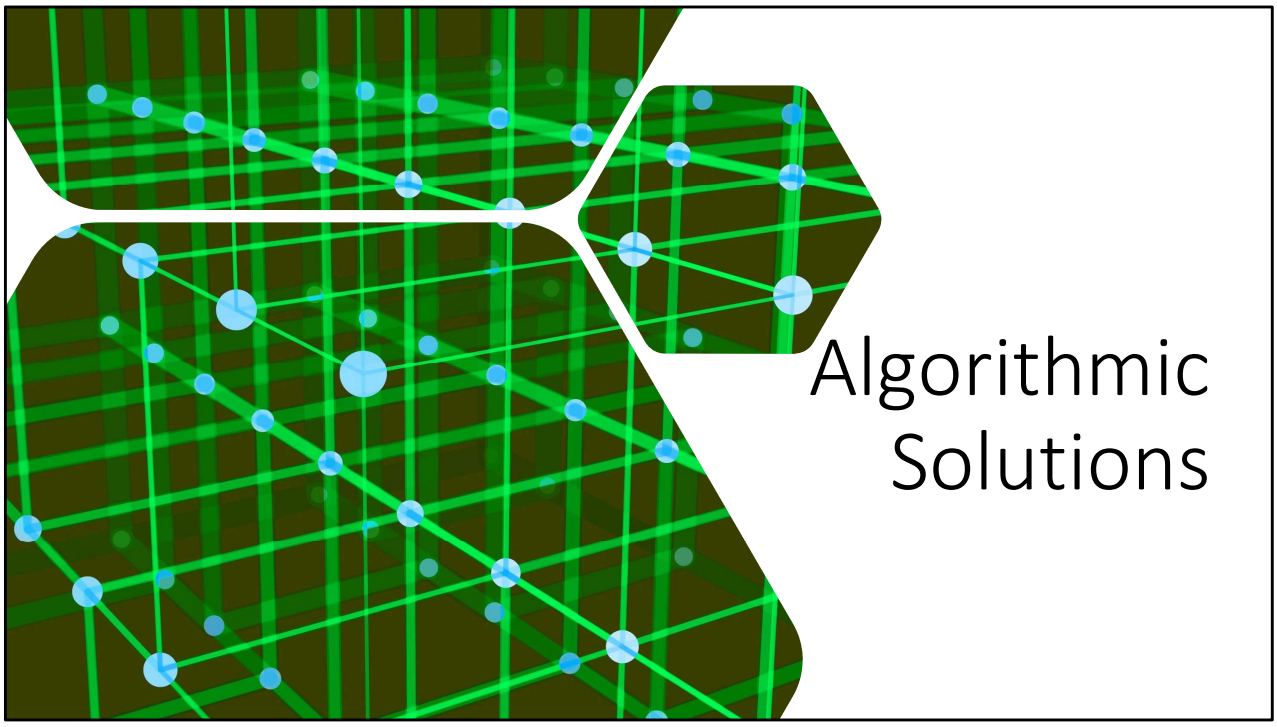
Back to real log flumes, they bring all the logs down to the river, where log drivers bundle them up and guide them down river to a sawmill.
https://www.youtube.com/watch?v=upsZZ2s3xv8

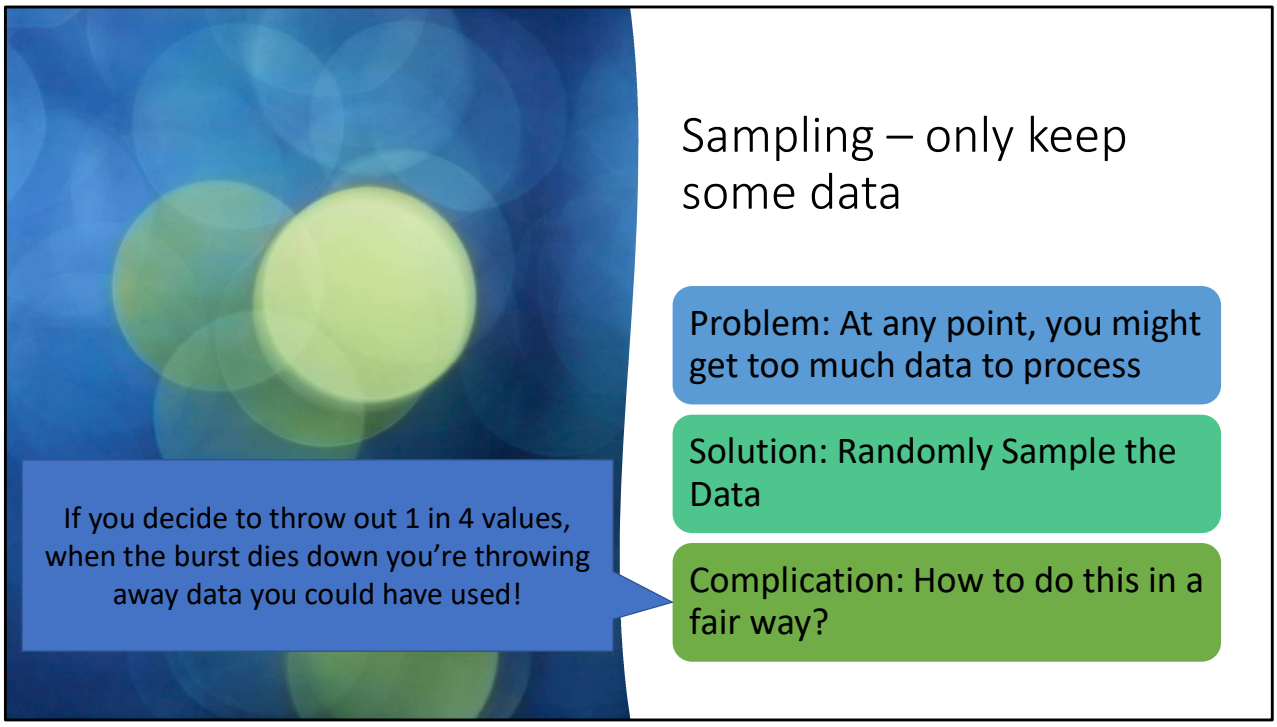
Remaining Issues…

- What if there's a BIG burst?
- If we can't examine all of it, what can we do so that the lost data causes us the least harm
  - It'll never be harmless
  - More = Better

Remember this slide?

- High volume
  - Might not be possible to store all of it
  - Might not be possible to even examine all of it


Algorithmic
Solutions


Sampling – only keep some data
Problem: At any point, you might get too much data to process
Solution: Randomly Sample the Data
Complication: How to do this in a fair way?
If you decide to throw out 1 in 4 values, when the burst dies down you're throwing away data you could have used!

Reservoir Sampling

Problem: Select S values uniformly at random from a stream of N values

Assumption: N is very big, and not known ahead of time (it's a stream!)

Solution:

- Store first S values
- When receiving k^{th} keep it with probability S/k
  - If keeping it, randomly discard one of the existing S elements to make room

Reservoir Sampling, Example

S = 10

- Keep first 10 elements
- 11^{th} element is kept with P = 10/11
- 12^{th} element is kept with P = 10/12
- …

Proof?

Claim: For K >= S, all values V_1 … V_k have a probability S / K of being in the reservoir.

Proof will be by induction, of course! Who doesn't love a good induction?

Basis for Induction: K = S+1

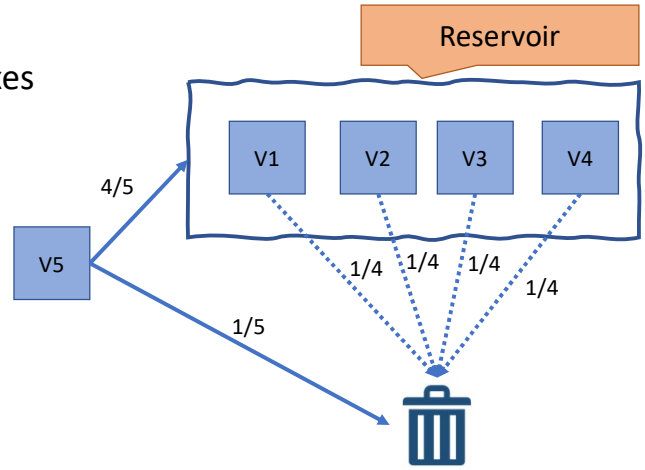
S = 4 because I don't want 11 boxes

Pr[Toss V5] = 1/5 (i.e. 1 – S/K)

Pr[Toss V1] = 1/S * Pr[Keep V5]
= 4/5 * 1/4 = 1/5

Pr[Toss V2] = 1/S * Pr[Keep V5]
= 4/5 * 1/4 = 1/5

…



Assumption

Assume that for some K > S, all K values have a S / K chance of being in the reservoir.

What are the chances the K+1 value is kept?

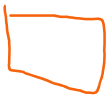
S / (K + 1)

What are the chances an existing item is kept?

1 – (S / (K + 1) x (1 / S)) = K / (K + 1)

What are the chances an item survives to K + 1?

S / K x K / (K + 1) = S / (K + 1)



So, by induction, blah blah blah, for all K > S, after K values have been seen, each value has a uniform S / K chance of being retained in the reservoir

Very hand-wavey induction

Hashing

Common Multiset operations that hashing can be used to estimate

- Cardinality: How many unique elements in the multiset?
- Membership: Is X a member of the multiset?
- Frequency: How often does X appear in the multiset?

HyperLogLog Counter (HLL)

Task: Estimate cardinality of a multiset (number of unique elements)

Observation: hash(item) -> vector of e.g. 32 bits

½ of items will have a hash code starting with 0
¼ of items will have a hash code starting with 00
…

How does this help us?

HyperLogLog Counter

If we've seen ~64 elements, we would Expect (As in E[…]) that

One of them would start with 000000 (6 zeros, 1 in 64 items)

All we need to do is record the longest string of leading 0s we've seen in any hash codes!

If we've seen x, our estimate is we've seen approximately 2^x unique items.

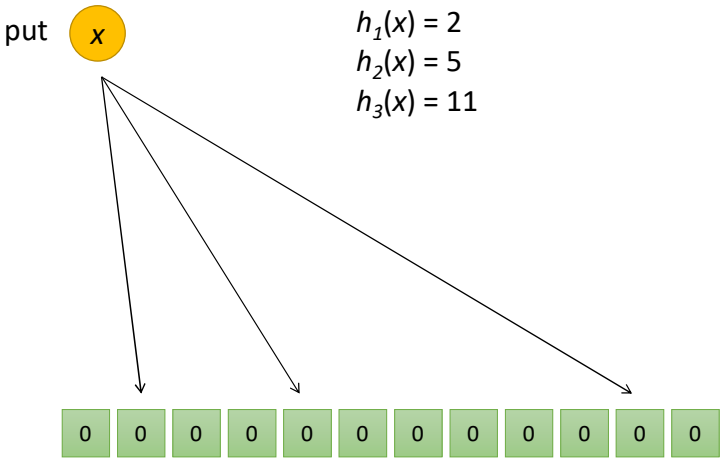
It's a Log Log counter because the number of leading zeros in a 32-bit number X is going to be around 32 – log2(X)
And we treat this as an estimator of the log2 of the cardinality of the set.

Bloom Filters

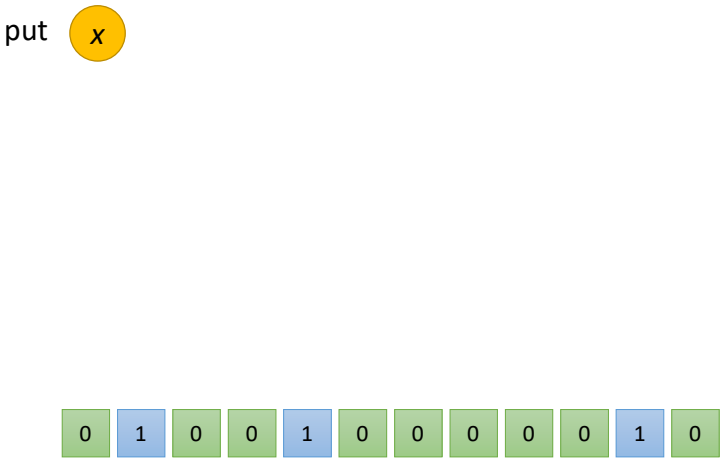
Problem: Set is too large to hold, want to know if it contains X

Solution: a bit-vector of *m* bits, and *k* unique hash functions.

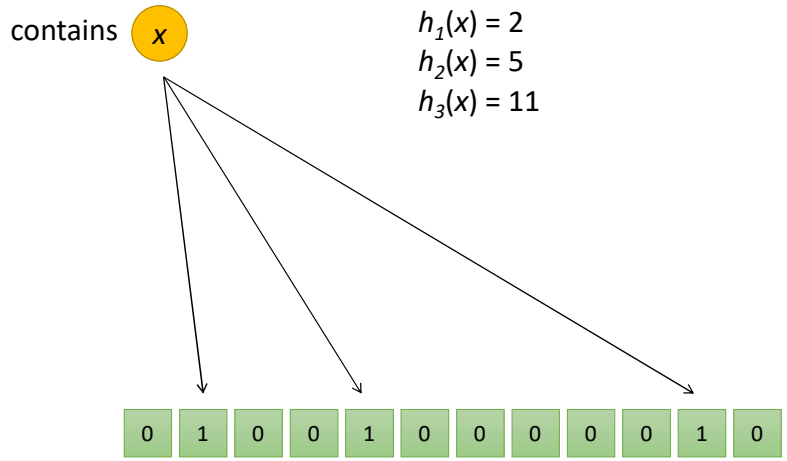
Bloom Filters: put



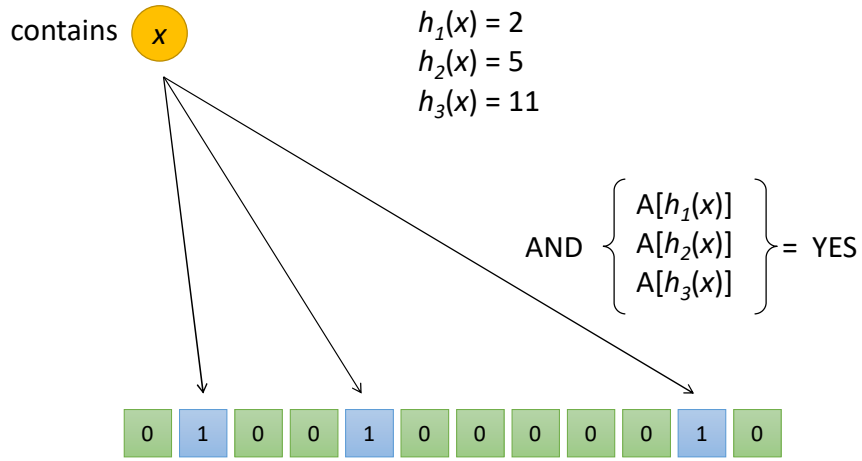
Bloom Filters: put



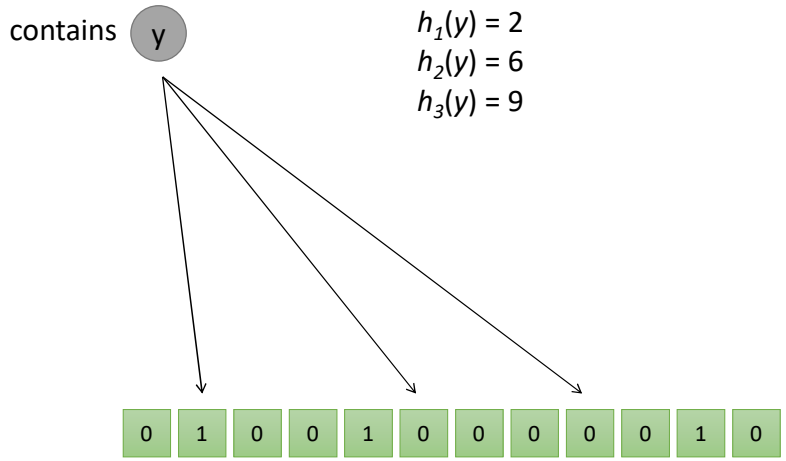
Bloom Filters: contains



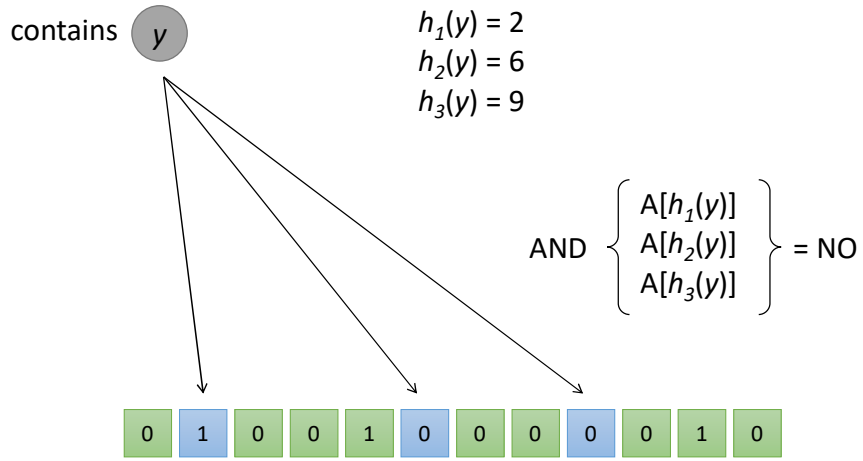
Bloom Filters: contains



Bloom Filters: contains



Bloom Filters: contains



Bloom Filters

No false negatives. If it's been seen, all of it's bits are set to 1

False positives: The more unique elements seen, the more bits are set to 1.

Can tune false positive rate by adjusting values for *m* and *k*

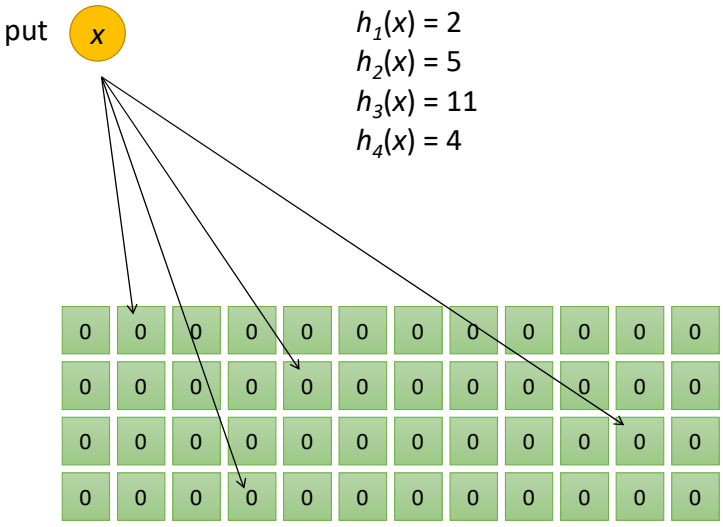
Count-Min Sketch (CM Sketch)

Counting the frequency of a value X within a multiset

Like Bloom Filters, we have a vector-length *m* and *k* independent hashes

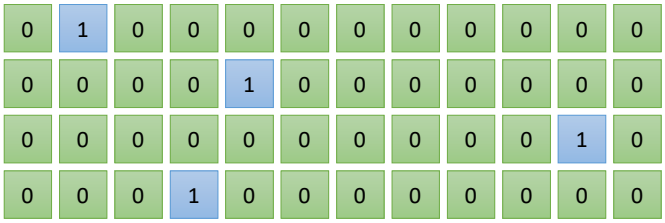
Unlike Bloom Filters, each hash has its own int-vector of *m* bits!

Count-Min Sketches: put

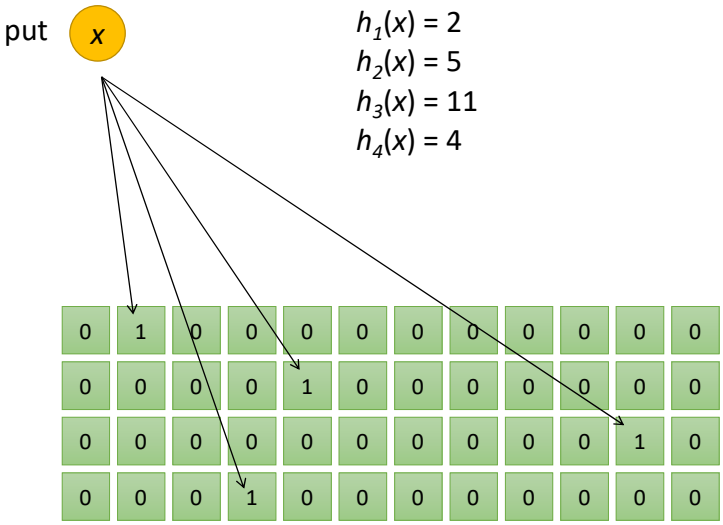


Count-Min Sketches: put

put *x*

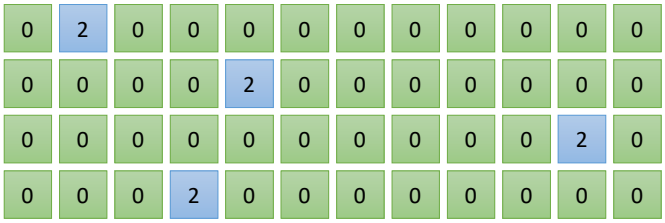


Count-Min Sketches: put

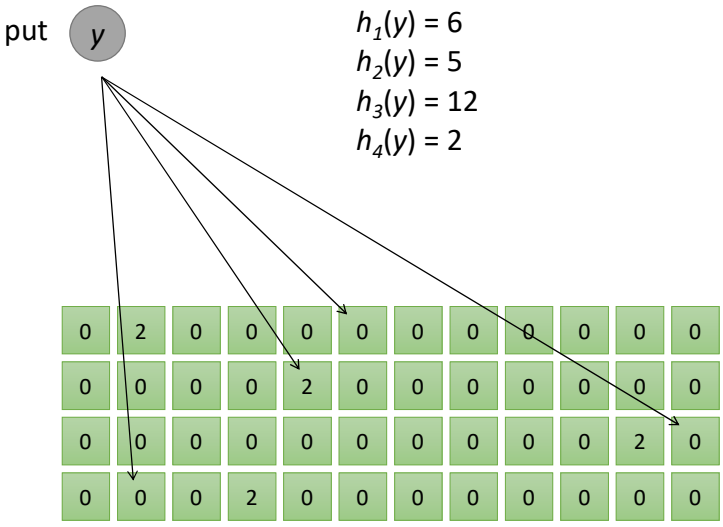


Count-Min Sketches: put

put *x*

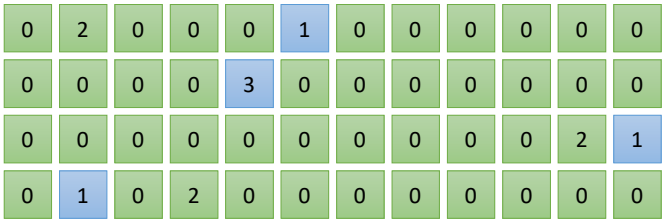


Count-Min Sketches: put

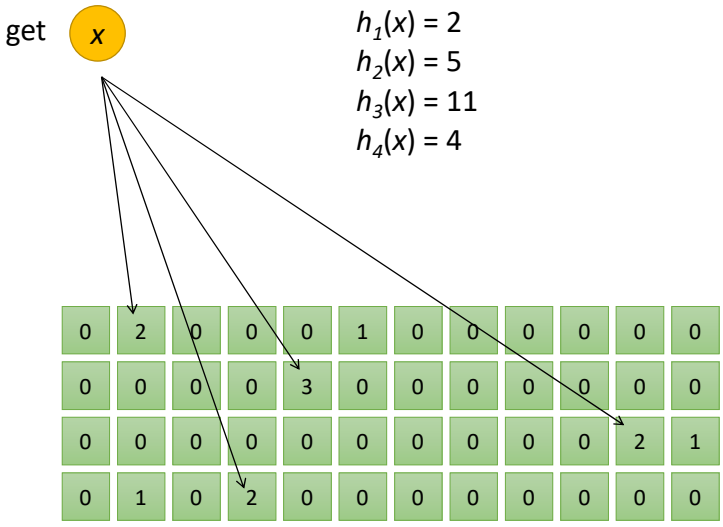


Count-Min Sketches: put

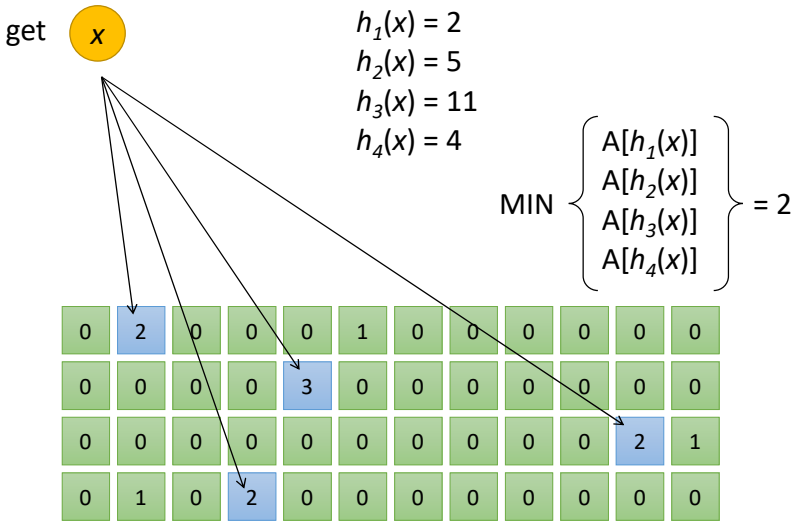
put y



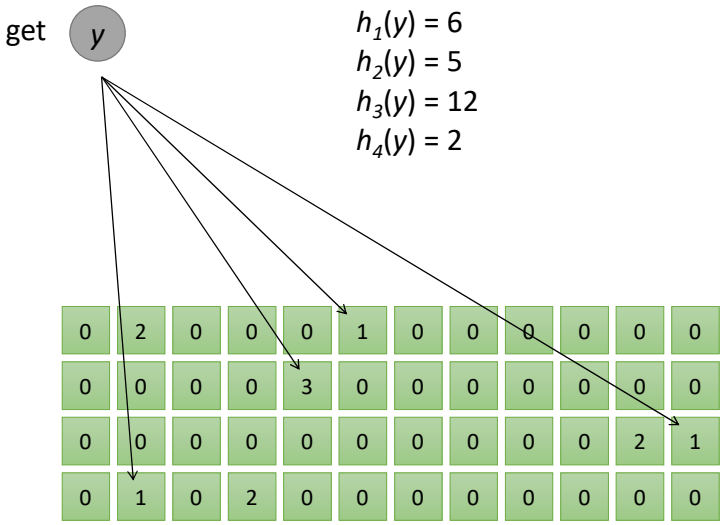
Count-Min Sketches: get



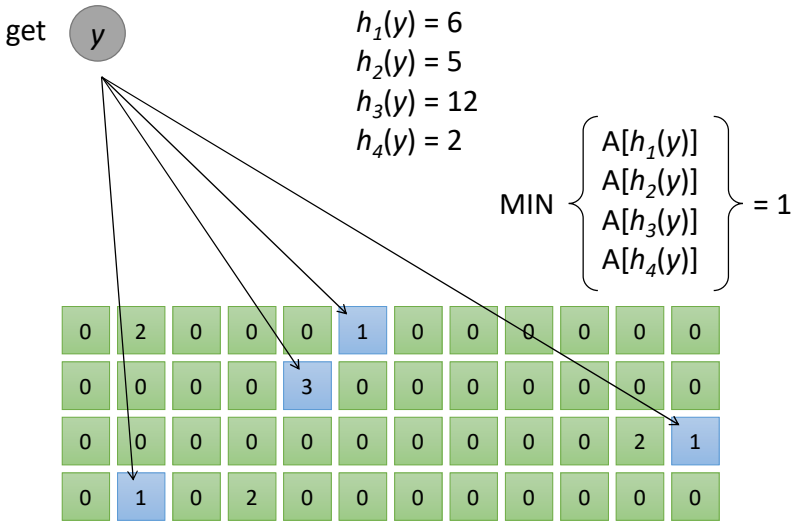
Count-Min Sketches: get

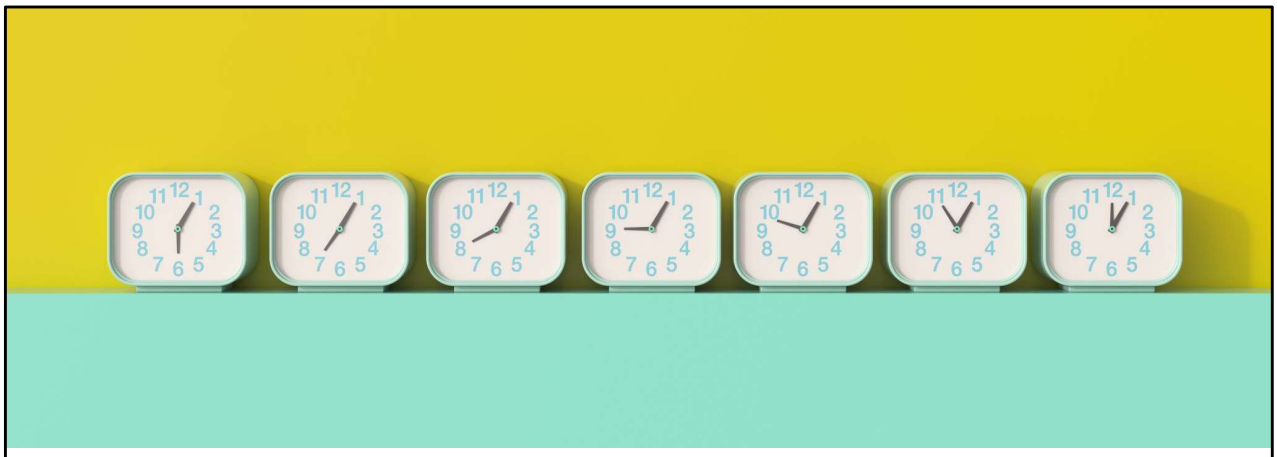


Count-Min Sketches: get



Count-Min Sketches: get





OPTIONAL KAFKA STUFF TO FILL TIME

Writing data to Kafka

- You use Kafka "producers" to write data to Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.
- A simple example producer:

```
1  Properties props = new Properties();
2  props.put("metadata.broker.list", "...");
3  ProducerConfig config = new ProducerConfig(props);
4
5  Producer p = new Producer(ProducerConfig config);
6  KeyedMessage<K, V> msg = ...; // cf. later slides
7  p.send(KeyedMessage<K,V> message);
```

87

Producers

- Two types of producers: "async" and "sync"

```
Properties props = new Properties();
props.put("producer.type", "async");
ProducerConfig config = new ProducerConfig(props);
```

- Same API and configuration, but slightly different semantics.
- What applies to a sync producer almost always applies to async, too.
- Async producer is preferred when you want higher throughput.

88

Producers

- Two aspects worth mentioning because they significantly influence Kafka performance:

  1. Message acking
  2. Batching of messages

1) Message acking

- Background:
  - In Kafka, a message is considered *committed* when "any required" replica for that partition have applied it to their data log.
  - Message acking is about conveying this "Yes, committed!" information back from the brokers to the producer client.
  - Exact meaning of "any required" is defined by `request.required.acks`.

- Only **producers** must configure acking
  - Exact behavior is configured via **`request.required.acks`**, which determines when a produce request is considered completed.
  - Allows you to trade **latency (speed)** <-> **durability (data safety)**.
  - Consumers: Acking and how you configured it on the side of producers do not matter to consumers because only committed messages are ever given out to consumers. They don't need to worry about potentially seeing a message that could be lost if the leader fails.

1) Message acking

- Typical values of `request.required.acks`
  - **`0`**: producer never waits for an ack from the broker.
    - Gives **the lowest latency** but the weakest durability guarantees.
  - **`1`**: producer gets an ack after the leader replica has received the data.
    - Gives better durability as the we wait until the lead broker acks the request. Only msgs that were written to the now-dead leader but not yet replicated will be lost.
  - **`-1`**: producer gets an ack after *all* replicas have received the data.
    - Gives **the best durability** as Kafka guarantees that no data will be lost as long as at least one replica remains.

2) Batching of messages

- Batching improves throughput
  - Tradeoff is data loss if client dies before pending messages have been sent.

- You have two options to "batch" messages:
  1. Use `send(listOfMessages)`.

```
producer.send(List<KeyedMessage<K,V>> messages);
```

     - Sync producer: will send this list ("batch") of messages *right now*. Blocks!
     - Async producer: will send this list of messages in background "as usual", i.e. according to batch-related configuration settings. Does not block!

  2. Use `send(singleMessage)` with async producer.

```
producer.send(KeyedMessage<K,V> message);
```

     - For async the behavior is the same as `send(listOfMessages)`.

92


Reading data from Kafka
93

Reading data from Kafka

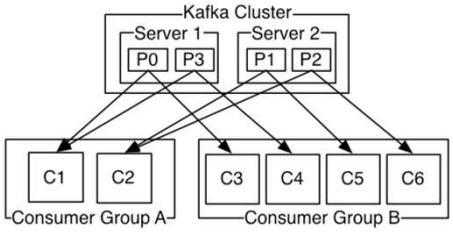
- You use Kafka “consumers” to write data to Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.

Reading data from Kafka

- Consumers *pull* from Kafka (there's no push)
  - Allows consumers to control their pace of consumption.
  - Allows to design downstream apps for **average** load, not peak load
- Consumers are responsible to track their read positions aka "offsets"

Reading data from Kafka

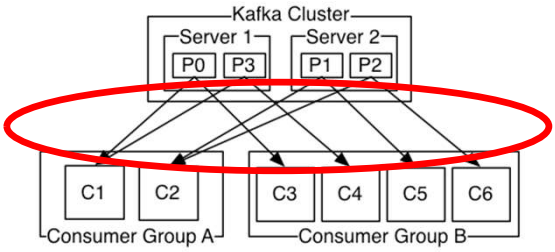
- Consumer “groups”
  - Allows multi-threaded and/or multi-machine consumption from Kafka topics.
  - Consumers “join” a group by using the same `group.id`
  - Kafka guarantees a message is only ever read by a single consumer in a group.
    - Kafka assigns the partitions of a topic to the consumers in a group so that each partition is consumed by exactly one consumer in the group.
    - Maximum parallelism of a consumer group: **#consumers** (in the group) <= **#partitions**



Guarantees when reading data from Kafka

- A message is only ever read by a single consumer in a group.
- A consumer sees messages in the order they were stored in the log.
- The order of messages is only guaranteed within a partition.

Rebalancing: how consumers meet brokers



- The assignment of brokers – via the partitions of a topic – to consumers is quite **important**, and it is **dynamic** at run-time.

98