Introduction to Big Data

Pooya Jamshidi

pooya.jamshidi@ut.ac.ir

Ilam University

School of Engineering,
Computer Group

May 9, 2025

Ilam University

Data Streams: Infinite Data

Data Streams

- In many data mining situations, we do not know the entire data set in advance
- **Stream Management** is important when the input rate is controlled **externally:**
  - Google queries
  - Twitter or Facebook status updates
- We can think of the **data** as **infinite** and **non-stationary** (the distribution changes over time)

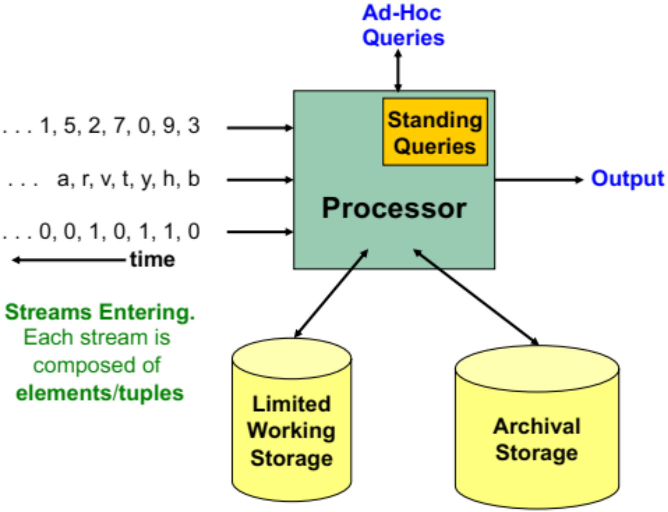
The Stream Model

- Input **elements** enter at a rapid rate, at one or more input ports (i.e., **streams**)
  - We call elements of the stream tuples
- **The system cannot store the entire stream accessibly**
- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

Side note: SGD is a Streaming Alg.

- **Stochastic Gradient Descent (SGD) is an example of a stream algorithm**
- **In Machine Learning we call this: Online Learning**
  - Allows for modeling problems where we have a continuous stream of data
  - We want an algorithm to learn from it and slowly adapt to the changes in data
- **Idea: Do slow updates to the model**
  - **SGD** (SVM, Perceptron) makes small updates
  - **So:** First train the classifier on training data.
  - **Then:** For every example from the stream, we slightly update the model (using small learning rate)

General Stream Processing Model



Problems on Data Streams

- **Types of queries one wants on answer on a data stream:** (we'll do these today)
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows**
    - Number of items of type x in the last k elements of the stream

Problems on Data Streams

- **Types of queries one wants an answer on a data stream:** (we'll do these next time)
  - **Filtering a data stream**
    - Select elements with property x from the stream
  - **Counting distinct elements**
    - Number of distinct elements in the last k elements of the stream
  - **Estimating moments**
    - Estimate avg./std. dev. of last k elements
  - **Finding frequent elements**

Applications (1)

- **Mining query streams**
  - Google wants to know what queries are more frequent today than yesterday
- **Mining click streams**
  - Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- **Mining social network news feeds**
  - E.g., look for trending topics on Twitter, Facebook

Applications (2)

- **Sensor Networks**
  - Many sensors feeding into a central controller
- **Telephone call records**
  - Data feeds into customer bills as well as settlements between telephone companies
- **IP packets monitored at a switch**
  - Gather information for optimal routing
  - Detect denial-of-service attacks

Sampling from a Data Stream

- Since **we can not store the entire stream**, one obvious approach is to store a sample
- **Two different problems:**
  - **(1)** Sample a fixed proportion of elements in the stream (say 1 in 10)
  - **(2)** Maintain a random sample of fixed size over a potentially infinite stream
    - At any "time" k we would like a random sample of s elements
    - – **What is the property of the sample we want to maintain?**
    - – For all time steps k, each of k elements seen so far has equal prob. of being sampled

Sampling a Fixed Proportion

- **Problem 1: Sampling fixed proportion**
- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as:** **How often did a user run the same query in a single day**
  - Have space to store $1/10^{th}$ of query stream
- **Naïve solution:**
  - Generate a random integer in **[0..9]** for each query
  - Store the query if the integer is 0, otherwise discard

Problem with Naïve Approach

- **Simple question: What fraction of queries by an average search engine user are duplicates?**
  - Suppose each user issues x queries once and d queries twice (total of $x + 2d$ queries)
    - **Correct answer:** $d/(x + d)$
- **Proposed solution: We keep 10% of the queries**
  - Sample will contain $x/10$ of the singleton queries and $2d/10$ of the duplicate queries at least once
  - But only $d/100$ pairs of duplicates
    - $d/100 = 1/10 \cdot 1/10 \cdot d$
  - Of d "duplicates" $18d/100$ appear exactly once
    - $18d/100 = ((1/10 \cdot 9/10) + (9/10 \cdot 1/10)) \cdot d$
- **So the sample-based answer is**

$$\frac{\frac{d}{100}}{\frac{x}{10} + \frac{d}{100} + \frac{18d}{100}} = \frac{d}{10x + 19d}$$

Solution: Sample Users

Solution:

- Pick $1/10^{\text{th}}$ of **users** and take all their searches in the sample
- Use a hash function that hashes the user name or user id uniformly into 10 buckets

Generalized Solution

- **Stream of tuples with keys:**
  - Key is some subset of each tuple's components
    - e.g., tuple is (user, search, time); key is **user**
  - Choice of key depends on application
- **To get a sample of a/b fraction of the stream:**
  - Hash each tuple's key uniformly into b buckets
  - Pick the tuple if its hash value is at most a

Hash table with b buckets, pick the tuple if its hash value is at most a.

How to generate a 30% sample?

Hash into $b = 10$ buckets, take the tuple if it hashes to one of the first 3 buckets.

Maintaining a fixed-size sample

- **Problem 2: Fixed-size sample**
- **Suppose we need to maintain a random sample S of size exactly s tuples**
  - E.g., main memory size constraint
- **Why?** Don't know length of stream in advance
- **Suppose at time n we have seen n items**
  - Each item is in the sample S with equal prob. s/n

How to think about the problem: say $s = 2$
Stream: `a x c y z k d g e ...`

At $n = 5$, each of the first 5 tuples is included in the sample S with equal prob.
At $n = 7$, each of the first 7 tuples is included in the sample S with equal prob.

Impractical solution would be to store all the n tuples seen so far and out of them pick s at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first s elements of the stream to S
  - Suppose we have seen $n - 1$ elements, and now the n^{th} element arrives ($n > s$)
    - With probability s/n, keep the n^{th} element, else discard it
    - If we picked the n^{th} element, then it replaces one of the s elements in the sample S, picked uniformly at random
- **Claim:** This algorithm maintains a sample S with the desired property:
  - After n elements, the sample contains each element seen so far with probability s/n

Proof: By Induction

- **We prove this by induction:**
  - Assume that after n elements, the sample contains each element seen so far with probability s/n
  - We need to show that after seeing element $n+1$ the sample maintains the property
    - Sample contains each element seen so far with probability $s/(n+1)$
- **Base case:**
  - After we see $n = s$ elements the sample S has the desired property
    - Each out of $n = s$ elements is in the sample with probability $s/s = 1$

Proof: By Induction

- **Inductive hypothesis:** After n elements, the sample S contains each element seen so far with prob. s/n
- **Now element $n+1$ arrives**
- **Inductive step:** For elements already in S, probability that the algorithm keeps it in S is:

$$\underbrace{\left(1-\frac{s}{n+1}\right)}_{\text{Element } n+1 \text{ discarded}} + \underbrace{\left(\frac{s}{n+1}\right)}_{\text{Element } n+1 \text{ not discarded}} \underbrace{\left(\frac{s-1}{s}\right)}_{\text{Element in the sample not picked}} = \frac{n}{n+1}$$

- So, at time n, tuples in S were there with prob. s/n
- Time $n \to n+1$, tuple stayed in S with prob. $n/(n+1)$
- So prob. tuple is in S at time $n+1$:

$$\frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$$

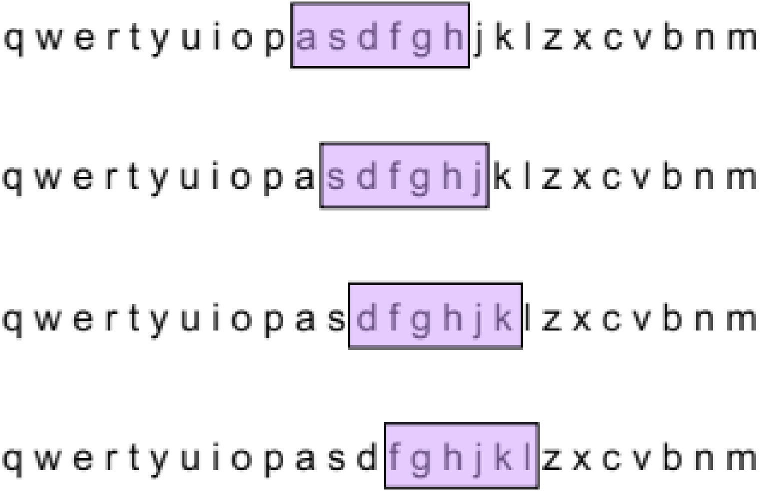
Sliding Windows

- A useful model of stream processing is that queries are about a **window** of length N
  - The N most recent elements received
- **Interesting case:** N is so large that the data cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
- **Amazon example:**
  - For every product $\mathbf{X}$ we keep 0/1 stream of whether that product was sold in the n^{th} transaction
  - We want answer queries, how many times have we sold $\mathbf{X}$ in the last k sales

Sliding Window: 1 Stream

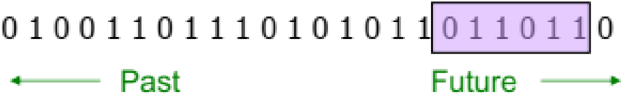
Sliding Window on a singel stream:

$N = 6$



Counting Bits (1)

- **Problem:**
  - Given a stream of 0s and 1s
  - Be prepared to answer queries of the form
    How many 1s are in the last k bits? where $k \leq N$
- **Obvious solution:**
  - Store the most recent N bits
  - When new bit comes in, discard the $N + 1^{\text{st}}$ bit

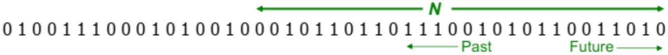


Counting Bits (2)

- You cannot get an exact answer without storing the entire window
- **Real Problem:**
  What if we cannot afford to store N bits?
  - **E.g.,** we're processing 1 billion streams and $N = 1$ billion
- **But we are happy with an approximate answer**

An Attempt: Simple Solution

- **Q: How many 1s are in the last N bits?**
- A simple solution that does not really solve our problem: **Uniformity assumption**



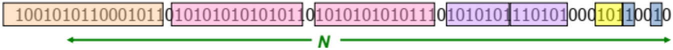
- **Maintain 2 counters:**
  - S: number of 1s from the beginning of the stream
  - Z: number of 0s from the beginning of the stream
- How many 1s are in the last N bits? $N \cdot \dfrac{S}{S+Z}$
- **But, what if stream is non-uniform?**
  - What if distribution changes over time?

DGIM Method (Datar, Gionis, Indyk, Motwani)

- **DGIM solution that does not assume uniformity**
- We store $O(\log^2 N)$ bits per stream
- **Solution gives approximate answer, never off by more than 50%**
  - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits.
- Read more here: `https://medium.com/fnplus/dgim-algorithm-169af6bb3b0c`

DGIM Method Idea

- **Idea:** Summarize blocks with specific number of 1s:
  - Let the block *sizes* (number of 1s) increase exponentially
- **When there are few 1s in the window, block sizes stay small, so errors are small**



Quiz

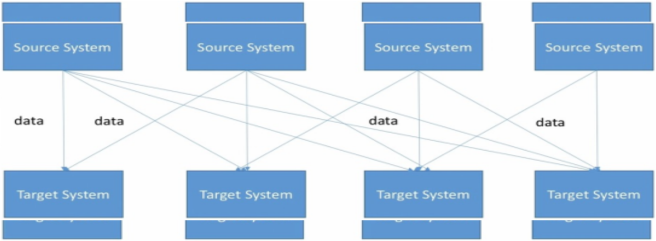
Question

- The question is so simple, calculate the AVERAGE of an input infinite stream of data!

Apache Kafka

Overview

- Kafka is a distributed event store and stream-processing platform.
- Fast
- Scalable
- Durable
- Distributed



Kafka Adoption and Use Cases

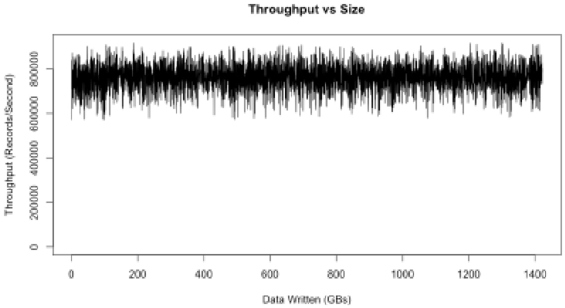
- **LinkedIn:** activity streams, operational metrics, data bus
  - 400 nodes, 18k topics, 220B msg/day (peak 3.2M msg/s), May 2014
- **Netflix:** real-time monitoring and event processing
- **Twitter:** as part of their Storm real-time data pipelines
- **Spotify:** log delivery (from 4h down to 10s),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

History

- Originally developed by Jay Kreps, Neha Narkhede and Jun Rao at LinkedIn and open sourced in 2011.
- Became a top level Apache project in 2012.
- Named after *Franz Kafka*, because it's a "a system optimized for writing."

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
- **Sustained throughput as stored data grows**
  - Slightly different test config than 2M writes/sec above.



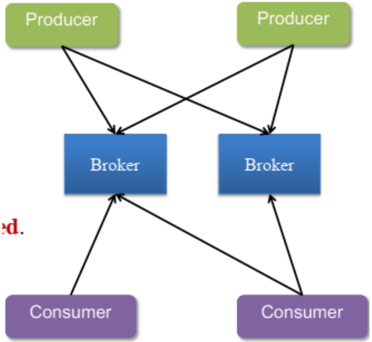
Why is Kafka so fast?

- **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
- **Combination of the two = fast Kafka!**
  - 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.

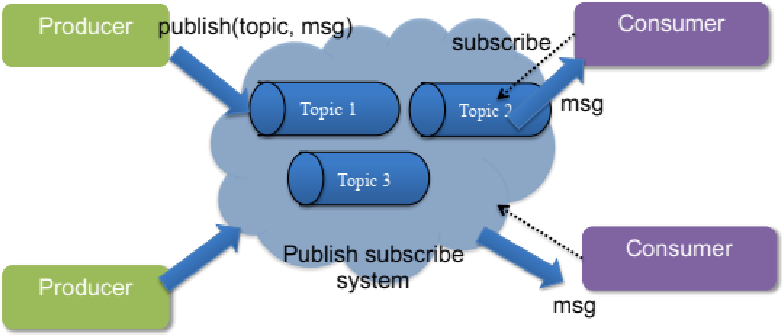
A First Look

- **The who is who**
  - Producers write data to brokers.
  - Consumers read data from brokers.
  - All this is distributed and load balanced.

- **The data**
  - Data is stored in topics.
  - Topics are split into partitions, which are replicated.



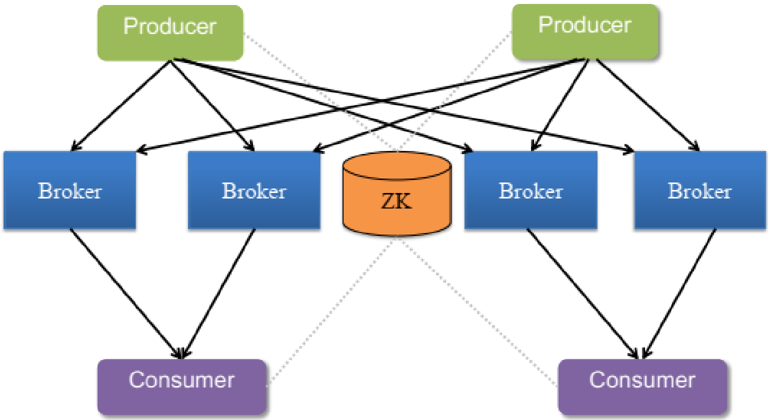
Kafka Implements a Pub/Sub



Apache ZooKeeper and Apache Kafka

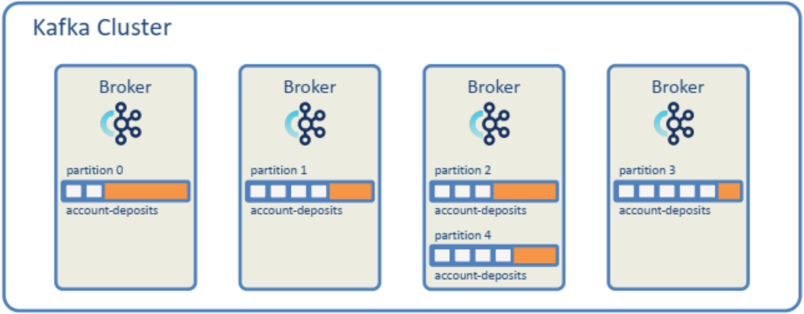
- **Apache ZooKeeper** is used in distributed systems for service synchronization and as a naming registry.
  - Apache Kafka depends on Apache ZooKeeper to run.
- When working with Apache Kafka, ZooKeeper is primarily used to track the status of nodes in the Kafka cluster and maintain a list of Kafka topics and messages.
- There's an experimental feature where you can run Apache Kafka without ZooKeeper.

Kafka Architeture



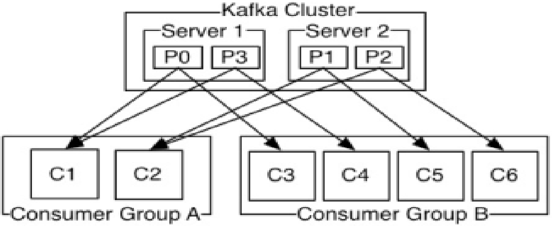
Topic Partitions

- Partition is the unit of distribution among topics across the cluster.
- Each partition's data is stored on a single broker.
- Partition is also the unit and parallelism for better scalability.



Partitions

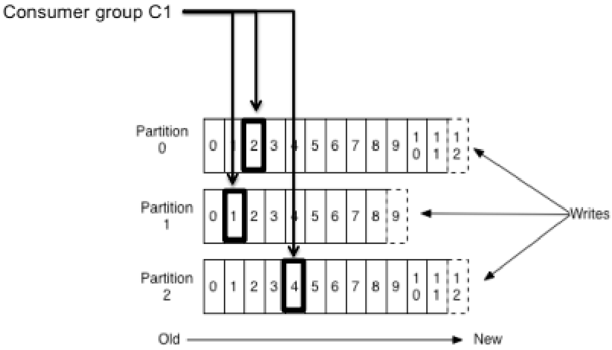
- Number of partitions of a topic is configurable.
- Number of partitions determines max consumer (group) parallelism.



- **Consumer group A**, with 2 consumers, reads from a 4-partition topic.
- **Consumer group B**, with 4 consumers, reads from the same topic.

Partition Offsets

- **Offset:** messages in the partitions are each assigned a unique (per partition) and sequential ID called the offset.
  - Consumers track their pointers via (offset, partition, topic) tuples.



Replicas of a Partition

- **Replicas:** "backups" of a partition
  - They exist solely to prevent data loss.
  - Replicas are never read from, never written to.
    - They do **NOT** help to increase producer or consumer parallelism!
  - Kafka tolerates (`numReplicas - 1`) dead brokers before losing data
    - `numReplicas == 2` $\rightarrow$ 1 broker can die