COMP6237 – Mining Data Streams

Shoaib Ehsan

s.ehsan@soton.ac.uk

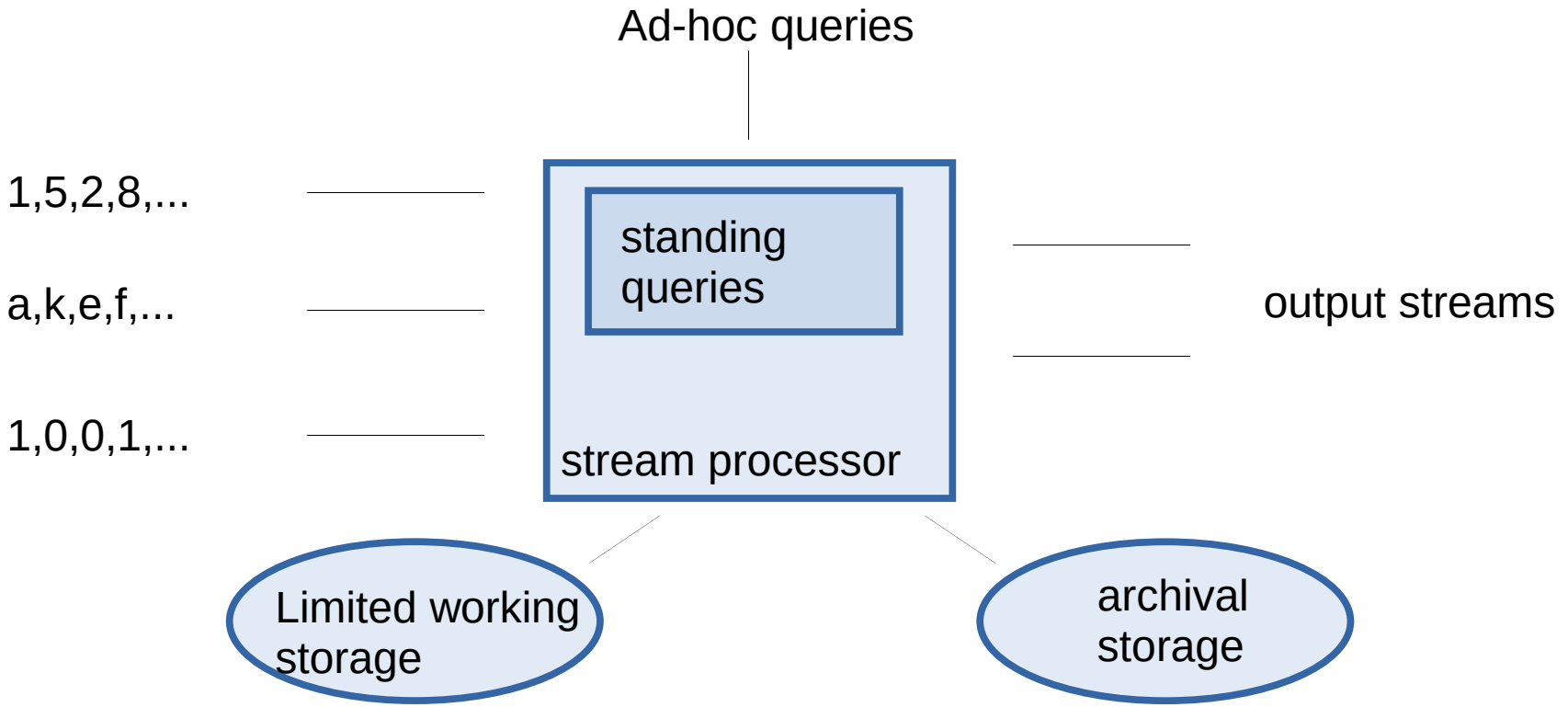
(Lecture based on chapter 4 of Leskovec et al. “Mining Massive Datasets”, 3rd edition)

COMP6237: Mining Data Streams

- Outline:
  - Motivation + Applications
  - Why is mining streams difficult?
  - Sampling data from a stream
  - Filtering streams – Bloom filters
  - Counting distinct elements in a stream: Flajolet-Martin algorithm
  - Summary

Motivation

- So far: most algorithms covered assume that data can be accessed in a data base
- Here: assume that data arrive in a stream and is lost forever if not processed immediately (and cannot all be stored in active storage)



Examples

- Sensor data
  - Ocean sensor + GPS to report surface height
    - E.g. reading of 4 bytes every 1/10 sec → 3.5MB/day → manageable
  - 1 million of such ocean sensors
    - 3.5 TB/day → ?
- Image data
  - Satellite data
  - Surveillance cameras (6 million in London alone!)
- Internet traffic
  - Switches … traffic routing, detecting attacks
  - Search engines … data mining on search queries? E.g. spread of the flu.

Queries

- Standing queries
  - Predetermined queries, can customize stream processor for it
  - E.g.: ocean sensor to report temperatures
    - Alert whenever temperature > 25 degrees → easy, only last stream element involved
    - All time average? → easy, only need to keep track of number of readings and sum.
    - Max. temperature ever recorded? → store current max and compare new readings.

Queries

- Ad-hoc queries
  - Questions asked once about current status of stream, may not be known beforehand.
  - Often use sliding window approach (e.g. store n most recent elements or elements that were streamed after time t, etc.)
  - Often better to have approximate answer than exact answer
  - → most of these techniques involve clever use of hashing

Sampling from Streams

Sampling from a Stream

- Problem:
  - Select subset of stream which is statistically representative of stream and allow to answer ad-hoc queries
- Example:
  - Search engine receives stream of queries of the form (user, query, time); can store 1/10 of stream
  - “What fraction of typical user queries were repeated over the last month?”
  - Idea: Sample 1/10 of stream ...

Sampling from a Stream (2)

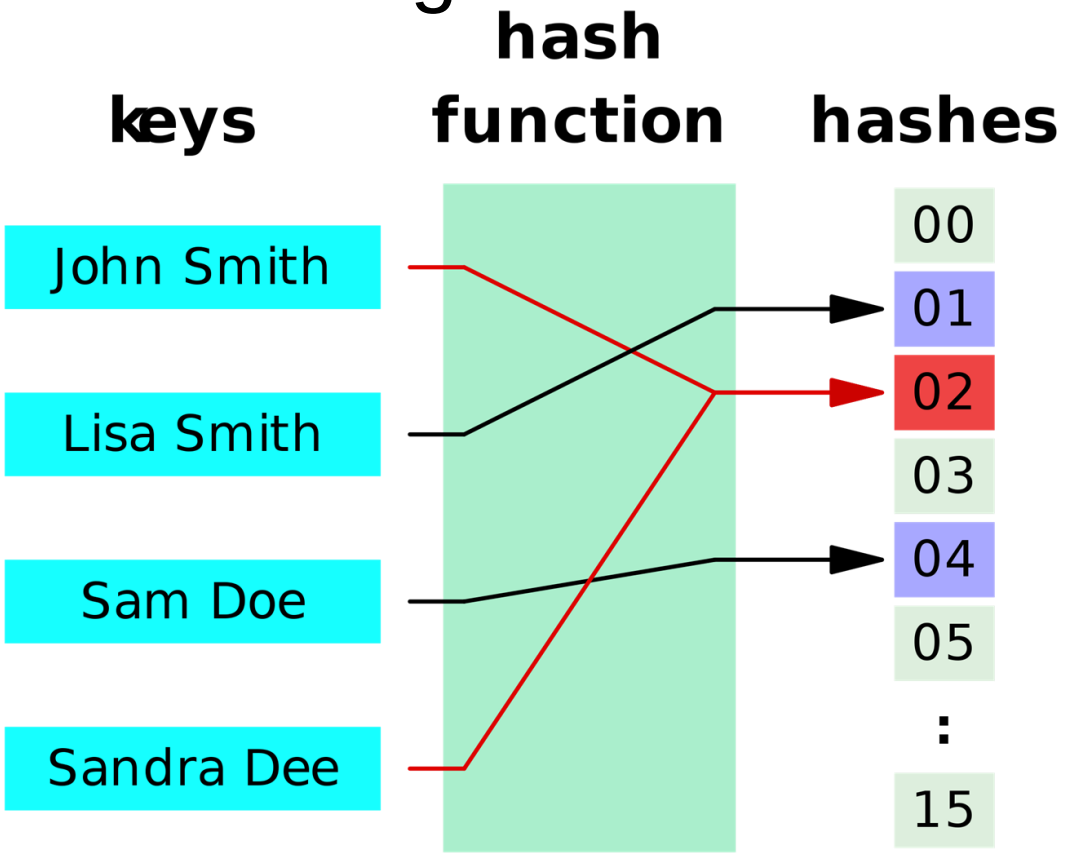
- Suppose typical user has issued s queries + d queries twice (and none more than twice).
  - Answer is that fraction d/(s+d) has appeared twice.
- What are we likely to find in our 1/10 sample?
  - s/10 queries issued once
  - d/100 queries issued twice
  - Of queries issued twice in full stream d*2*1/10*9/10=18d/100 will appear once in sample
  - Sample gives us answer: d/(10s+19d) of queries have appeared twice.
- → This type of query cannot be answered by randomly sampling queries from the stream!

Sampling from a Stream (3)

- How to do better?
  - Sample 1/10 of users and store all their queries
  - Could do:
    - Have a list of users in sample.
    - New query: check user in sample? Yes → store. No: with prob. 1/10 add user to sample and store query.
    - Works if we can afford to store list of all users + pay cost of looking up users in list!
  - Use hashing!
    - Devise hash function which maps user names to 10 buckets and keep query if user name maps to (e.g.) bucket 0.

An Aside: Hashing

- A hash function:
  - takes an input as a key, which is associated with a datum or record and used to identify it to the data storage and retrieval application.



Pros:
Very fast access
Cons:
Collisions, so not accurate.

(from wiki page on hashing)

Hashing to Construct Samples

- This is an example of a general problem
  - Say stream consists of elements with n components, part of the component is the **key** on which selection of the sample is to be based
    - In example (user, query, time) user was the “key”
  - Make a sample of size a/b by hashing key to b buckets and accept element into sample if hash is smaller than a.
- What about the sample size?
  - 1/10 of the stream might be very large, would rather want to bound by size of active storage (say t) → fraction of accepted key value would have to vary
  - Idea: hash to very large set of values, e.g.: 0,1,…,B-1
  - Start with some threshold t (could initially be B)
  - At all times sample consists of elements with key h(key)<=t
  - If number of elements in sample too large, decrease t and remove elements which no longer meet h(key)<=t from sample

Filtering Streams

Filtering Streams

- Want to only accept tuples of a stream that meet a certain criterion and pass it on to another stream

Incoming stream
(e.g. emails)

outgoing stream
(e.g. non-spam emails)

(key, value)

(e.g. email address, content)

Filtering Streams

- Want to only accept tuples of a stream that meet a certain criterion and pass it on to another stream
  - If selection criterion is property of the tuple that can be calculated → easy to do
  - Difficult if property is membership in a set (in particular if that set is large!) → "**Bloom filtering**"
- Example: Spam filters for emails
  - Stream: (email address, email content)
  - Suppose we have 1G of main memory
  - Say we have a set S of allowed email addresses; say 1 billion email addresses (cannot be stored in main memory)

Bloom Filtering for Spam Detection

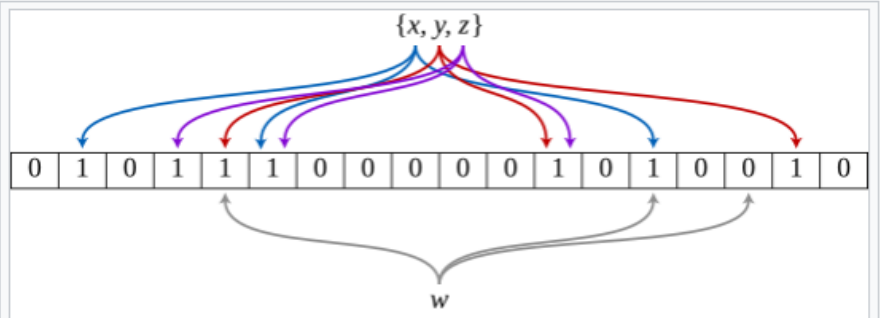
- Preparation:
  - Use main memory as bit array of 8 billion bits; initialize bit array to zero
  - Devise a hash function h that maps email addresses to 8 billion buckets
  - Set value of bit array to 1 for all hash values of the email addresses in S.
- Processing:
  - If query comes along, check if array[h(email address)]=1 → accept, otherwise reject.

Bloom Filtering for Spam Detection (2)

- Pro's:
  - Very fast! (cost of query is essentially cost of calculating hash function)
  - If element is not in S → array[h(key)]=0, so definitely spam (no false negatives)
- Con's:
  - Cannot remove elements from filter
  - Around 1/8th of the values of array will be 1's
  - Suppose array[h(key)]=1.
    - Can be in S
    - Could not be in S and just accidentally hash to a bit 1 and pass (allows for false positives)
    - Approx. 1/8 of email addresses not in S will get through …
    - Could improve this fraction by a cascade of filters!

Bloom Filters

- A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set.
- False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set".



An example of a Bloom filter, representing the set $\{x, y, z\}$. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set $\{x, y, z\}$, because it hashes to one bit-array position containing 0. For this figure, $m = 18$ and $k = 3$.

Source: https://en.wikipedia.org/wiki/Bloom_filter

Bloom Filters

- A Bloom filter consists of:
  - An array of n bits, initially all set to zero
  - A collection of k hash functions h1,h2,…,hk. Each hash function maps keys to n buckets corresponding to the n bits of the bit array
  - A set S of m key values
- Initialization:
  - Set all bits to zero.
  - Take each element of S and set elements of bit array to one for all hash functions hi(key).
- Processing:
  - Test whether a key that arrives is in S by checking if all h1(key), h2(key),…,hk(key)=1. Otherwise not in S.

Counting the number of distinct elements in a stream

Counting Distinct Elements in a Stream

- Suppose stream elements chosen from some set S
- Q: How many different elements have appeared in stream?
  - Example: Web site gathering stats on number of unique users over some time frame – maybe amazon (users log in with unique user name) or google (search queries identified with IP addresses)
- Obvious solution:
  - Store all elements in main memory and count
  - Problem if number of distinct elements is too great!
- Idea: Only estimate this number using hashing ...

Flajolet-Martin Algorithm

- Hash elements of S to sufficiently long bit string
  - More possible results of hash function than elements in S
- Idea:
  - More different elements of stream → more different hash values.
  - As we see more different hash values, it becomes more likely that one of these values becomes unusual. Here we focus on “tail length”, number of zeros at the end of the hash
  - Let R be the max. tail length seen so far → Estimate that there are 2^R distinct elements in the stream
  - Combine estimates of many hash functions.

Flajolet-Martin Algorithm (2)

- Why does this work?
  - Prob. that a particular stream element h(a) ends in at least r zero's is 2^{-r}.
  - Suppose we have m distinct elements in stream. Prob. that at least one of them has tail length of at least r is $(1-2^{-r})^m \sim \exp(-m\,2^{-r})$.
    - Suppose m much larger than 2^r → prob to find tail of at least length r approaches 1
    - Suppose m much smaller than 2^r→ prob to find tail of at least length r approaches 0
  - → 2^R is unlikely to be either much to high or much too low.

Flajolet-Martin Algorithm (3)

- How to combine estimates from different hash functions?
  - Take expectation of estimates of different hash functions?
    - Will not work, typically an overestimate
    - Consider r such that $2^r >> m$. Some prob. p that we discover r to be the largest number of 0's at the end of any hash value
    - Prob of finding r+1 is at least p/2, but 2^R has doubled ...
  - Use median?
    - Not affected by occasionally large 2^R but always power of 2 → not possible to find a close estimate.
  - Combine both!
    - Form small groups of hash functions, calculate expectation over group
    - Calculate median over all groups

Flajolet-Martin Algorithm (4)

- Good:
  - Do not need to store any stream element! Storage requirement is one integer per hash function.
  - More hash functions → better estimate. Practical limitation is cost of evaluating the hash functions.
- Bad:
  - Only approximate solution.

Flajolet-Martin Algorithm (5)

- Example:
  - Consider stream 1,2,1,3
  - Hash function: h(x)=2x+2 mod 4 (treated as 3-bit integer)

1 → 0 → 000
2 → 2 → 010
3 → 0 → 000

Should ignore these, since when mapping to 0 they provide no information

max tail length is 1 → 2 distinct elements for this hash function

  - This is not a very good estimate … would need to calculate for more hash functions to improve.

Summary

- Stream processing model
- Sampling from a stream
  - Issues
  - Sliding window technique
  - Use of hashing
- Filtering streams → Bloom filters
- Counting distinct elements → Flajolet-Martin algorithm