# COMP336 — Big Data

## Week 8 Lecture 1: Mining Data Streams

Diego Mollá

Department of Computer Science
Macquarie University

COMP348 2018H1

# Programme



## Reading

- Leskovec, Rajaraman, Ullman (2014): Mining of Massive Datasets, Chapter 4. `http://www.mmds.org/`

# Programme

# Characteristics of Data Streams

**Velocity:** Data may arrive faster than we can process it.

**Volume:** Accumulated data might not fit in the available storage space. We can think of data as infinite.

**Variety:** Data may change in time. Data that happened some time ago might not be relevant any more. We can think of data as non-stationary.

- (This is not the standard meaning of variety...)

We still need to handle the "classic" issue of variety: we may need to handle multiple streams at once.

# Examples of Data Streams

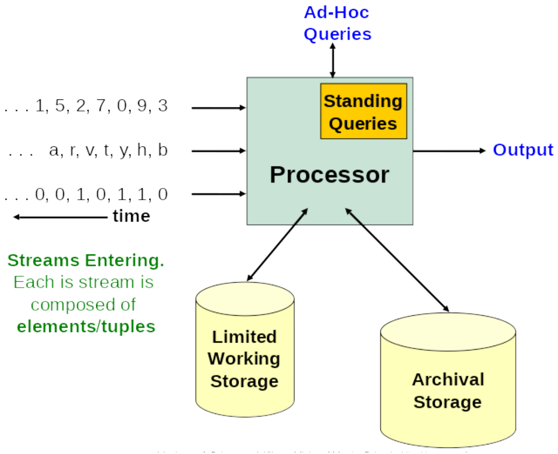
Image Data: Surveillance cameras, satellite imaginery, ...

Sensor data: Temperature, GPS coordinates, heart rate, ...

Internet and Web Traffic :

- Search queries;
- Posts from Twitter, Facebook, ...
- IP packets;
- Clicks.

# The Stream Model




http://www.mmds.org/

# Storage in the Stream Model

## Archival Storage

- Large storage for archival purposes.
- We assume it is not possible to answer queries from the archival store.
- Can be used only under special circumstances using time-consuming retrieval processes.

## Working Store

- Holds summaries or parts of streams.
- Can be used for answering queries.
- Might be in disk or in main memory.
- Cannot store all the data from all the streams.

# Storage in the Stream Model

## Archival Storage

- Large storage for archival purposes.
- We assume it is not possible to answer queries from the archival store.
- Can be used only under special circumstances using time-consuming retrieval processes.

## Working Store

- Holds summaries or parts of streams.
- Can be used for answering queries.
- Might be in disk or in main memory.
- Cannot store all the data from all the streams.

# Types of Queries

## Standing Queries

- Queries that are always performed on the data.
- In a sense, these are queries that are permanently executing.
- Since these queries are known in advance, it is fairly easy to design efficient storage and query processes to handle them.

## Ad-Hoc Queries

- Queries that are not known in advance.
- These queries are created, for example, by a user or operator.
- We need to find a way to query the current state of the stream.

# Types of Queries

## Standing Queries

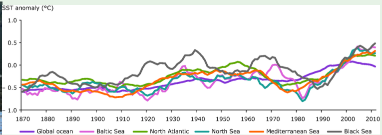
- Queries that are always performed on the data.
- In a sense, these are queries that are permanently executing.
- Since these queries are known in advance, it is fairly easy to design efficient storage and query processes to handle them.

## Ad-Hoc Queries

- Queries that are not known in advance.
- These queries are created, for example, by a user or operator.
- We need to find a way to query the current state of the stream.

# Examples of Standing Queries

## Example: Ocean Surface Temperature Sensor




1. Alert when the temperature exceeds 25 degrees centigrade.
2. Average the 24 most recent readings.
3. Maximum temperature ever recorded.
4. Average temperature.

## Question

What information do we need to keep in the working storage to answer each of these standing queries?

# Examples of Working Storage Needs

**Q1: Alert when the temperature exceeds 25°C**

- No information required (unless we want to allow a threshold input by the user)

**Q2: Average the 24 most recent readings**

- 24 variables, one per reading.

**Q3: Maximum temperature ever recorded**

- 1 variable with the value of the maximum so far.

**Q4: Average temperature**

- 1 variable with the value of the sum of readings so far.
- 1 variable that counts the number of readings so far.

# Examples of Working Storage Needs

**Q1: Alert when the temperature exceeds 25°C**

- No information required (unless we want to allow a threshold input by the user)

**Q2: Average the 24 most recent readings**

- 24 variables, one per reading.

**Q3: Maximum temperature ever recorded**

- 1 variable with the value of the maximum so far.

**Q4: Average temperature**

- 1 variable with the value of the sum of readings so far.
- 1 variable that counts the number of readings so far.

# Examples of Working Storage Needs

**Q1: Alert when the temperature exceeds 25°C**

- No information required (unless we want to allow a threshold input by the user)

**Q2: Average the 24 most recent readings**

- 24 variables, one per reading.

**Q3: Maximum temperature ever recorded**

- 1 variable with the value of the maximum so far.

**Q4: Average temperature**

- 1 variable with the value of the sum of readings so far.
- 1 variable that counts the number of readings so far.

# Examples of Working Storage Needs

**Q1: Alert when the temperature exceeds 25°C**

- No information required (unless we want to allow a threshold input by the user)

**Q2: Average the 24 most recent readings**

- 24 variables, one per reading.

**Q3: Maximum temperature ever recorded**

- 1 variable with the value of the maximum so far.

**Q4: Average temperature**

- 1 variable with the value of the sum of readings so far.
- 1 variable that counts the number of readings so far.

# Question: An effective way to compute the average temperature

## Q4: Average temperature

If we keep the sum of readings so far we may have problems with data overflow (the sum may exceed the capacity of storage)

1. How serious is this problem?
2. How could we fix this problem?

# Examples of Ad-hoc Queries

## Example: Web Site

1. What were the unique users in the past month?
2. What were the users from Australia?
3. What were the users with generated most traffic?

## Note

- If the above were questions were known beforehand they would be standing queries.
- Given an application we can optimise it to enable the processing of some kinds of ad-hoc queries.
- In general, it is impossible to be able to accurately answer any possible ad-hoc query.

# Issues in Stream Processing

### Issues

- Velocity: We may need to give up on processing all data.
- Volume: We may need to build summaries.
  - Not all ad-hoc questions can be answerable.

### Possible Solution

- Obtain an approximate answer to the question rather than an exact answer.

$\Rightarrow$ Techniques related to hashing can be very useful.

# Programme

# Programme

# Example: Stream of user queries

Suppose we store 1/10th of all user queries in order to save space.

- If the ad-hoc query is "how many queries did user $u$ ask?", the (approximate) answer is easy: $10 \times$ the recorded queries of user $u$.
- If the ad-hoc query is "how many of $u$'s queries were repeated?" the answer is more complicated. Imagine:
  - $s$ is the number of queries recorded once.
  - $d$ is the number of queries recorded twice.
  - No queries were issued more than twice (to simplify this problem).

  Then, the (approximate) answer $10 \times d$ is wrong; why?

# Working Through the Example

- If the user has made $n$ queries once, $n/10$ will be recorded.
- If the user has made $m$ queries twice, only $m/100$ will be recorded twice (on average).
- Thus, the correct answer is $100 \times d$.

**Note**

If the ad-hoc query is "what fraction of the user's queries were repeated?" the answer is more complex; see textbook section 4.2.1, pages 134–135 for a detailed explanation.

# Using a Hash Function to Obtain a Representative Sample

- We need to be able to obtain a representative sample of fast stream data.
- Suppose we want to keep the information of 1/10th of the users.
- Sometimes, even checking whether a user of a search string is in the list of previous users is too time-consuming.
- By using a hash function we can avoid keeping a list of past users.

# Keeping 1/10 of the users — brute force approach

1. Keep a set of past users $u$ initialised to empty $\emptyset$.
2. Keep a set of past selected users $s$ initialised to $\emptyset$.
3. If the user from the incoming stream item is not in $u$:
    1. Add the user to $u$.
    2. Generate a random number between 0 and 10.
    3. If the random number is 0, add the user to $s$ and store the query.
4. If the user is in $u$:
    1. If the user is also in $s$, store the query.

**But**

- The lists $u$ and $s$ could become too large and difficult to maintain!
- Checking whether a user is in $u$ or in $s$ could become too time-consuming!

# Keeping 1/10th of the users — using a hash

1. Hash user using hash function that maps users to 10 buckets.
2. If resulting bucket is 0, store the query.

**Notes**

- We use hash functions to approximate sampling.
- The resulting approach is much simpler.
- The resulting approach is much faster!

# The General Sampling Problem

- The stream consists of tuples with $n$ components.
- A subset of $n$ are the key components.
- We want to make a selection of the key components.
- We can use hash functions to obtain a sample consisting of any rational function $a/b$.
  1. Hash the key into $b$ buckets.
  2. If the hash value is less than $a$, record the tuple.
- If the key has more than 1 component, the hash needs to obtain a value for the combined key components.

# The General Sampling Problem: Example

## The Problem

- A stream of social media posts issues pairs of two components:
  1. User ID.
  2. Text of the social media post.
- How do we keep samples from two thirds of the users?

## The Solution

1. Key: User ID
2. a: 2
3. b: 3

We hash the user ID into $b = 3$ buckets (0, 1, 2). If the hash value is less than $a = 2$, we record the sample.

# The General Sampling Problem: Example

## The Problem

- A stream of social media posts issues pairs of two components:
  1. User ID.
  2. Text of the social media post.
- How do we keep samples from two thirds of the users?

## The Solution

1. Key: User ID
2. a: 2
3. b: 3

We hash the user ID into $b = 3$ buckets (0, 1, 2). If the hash value is less than $a = 2$, we record the sample.

# Programme

1. Data Streams
2. Sampling and Filtering
   - Sampling Data in a Stream
   - Filtering Streams
3. Counting Elements in a Stream

# Example

- We want to allow incoming email only from a whitelist $S$ of authorised users.
- But the list $S$ has 1 billion email addresses!
- We can use hash functions to solve this (again).

## Using a hash function

1. Define a hash function that maps email addresses into buckets (as many buckets as practical).
2. Keep an array with as many bits as hash buckets.
3. For every email address in whitelist $S$, store 1 in the array indexed by the email hash bucket.
4. Then, any incoming email address that hashes to a bucket with value 1 stored in the array, is deemed as belonging to whitelist $S$.

# Filtering a Stream, Algorithm

## Building the filter

```
hash_filter = [False for i in range(nbuckets)]
for s in S:
    hash_filter[hash(s, nbuckets)] = True
```

## Testing the filter

```
def in_filter(item):
    return hash_filter[hash(item, nbuckets)]
```

# Questions about Our Example

## Questions

1. Would this approach generate false positives? (email not in the whitelist is accepted)
2. Would this approach generate false negatives? (email in the whitelist is filtered out)
3. Which problem is worse? 1 or 2?

# The Bloom Filter

- A Bloom filter consists of:
  1. An array of $n$ bits, initially all 0's.
  2. A collection of $k$ hash functions $h_1, h_2, \ldots, h_k$.
     - Each hash function maps key values to $n$ buckets.
  3. A set $S$ of $m$ key values.
- To initialise the bit array:
  1. Begin with all bits 0.
  2. For each key value $v$ in $S$ and for each hash function $h_i$:
     1. Set 1 to array bit indexed by $h_i(v)$.
- To test a key value $w$ that arrives in the stream:
  1. If all $h_1(w), h_2(w), \ldots h_k(w)$ are 1's in the bit array, let the stream element through.
  2. If one or more of these bits are 0, reject the stream element.

# Bloom Filter, Algorithm

## Building the filter

```
hash_filter = [False for i in range(nbuckets)]
for s in S:
   for k in range(K):
      hash_filter[hash((s, k), nbuckets)] = True
```

## Testing the filter

```
def in_filter(item):
    result = True
    for k in range(K):
        result = result and hash_filter[hash((item, k),
                                              nbuckets)]
    return result
```

# Analysis of Bloom Filtering

- If a key value is in $S$, then the element will pass through the Bloom filter.
- If the key value is not in $S$, it might still pass (a false positive).

What is the probability of a false positive?

(see text book, section 4.3.3, pages 138-139, for an explanation)

- The probability of a false positive is $(1 - e^{(-km/n)})^k$.

# Programme

1. Data Streams
2. Sampling and Filtering
3. Counting Elements in a Stream
   - Counting Distinct Elements
   - Exponentially Decaying Windows

# Programme

1. Data Streams
2. Sampling and Filtering
3. Counting Elements in a Stream
   - Counting Distinct Elements
   - Exponentially Decaying Windows

# The Count-Distinct Problem

## The Problem

Suppose you want to count the number of distinct items in a stream, either from the beginning or from some known time in the past.

## Issues

1. We cannot store all distinct items so far.
2. Even using a hash table, the size of the table is limited.

## Question

How would you count distinct items using a hash table?

# The Flajolet-Martin Algorithm

- The Flajolet-Martin algorithm estimates the count of distinct items without keeping track of all past distinct items.
- The count is estimated in an unbiased way.
- The algorithm limits the probability that the estimation error is large.
- The algorithm uses hash functions but it does not need to keep hash tables.

# The Flajolet-Martin Algorithm

1. Use hash functions that map each of the $N$ elements to at least $\log_2 N$ bits.
    - For example, $2^{64}$ buckets (64 bits) are enough to hash URL's.
2. Define the tail length of hash $h$ and item $a$ as the number of trailing zeroes in $h(a)$.
    - If $h(x) = 11010$, then the tail length is 1.
    - If $h(x) = 01000$, then the tail length is 3.
3. Let $R$ be the largest tail length observed so far.
4. The estimated number of distinct elements is $2^R$.

# Why it Works: Intuition

- $h(a)$ hashes with equal probability to any of $N$ values. Therefore:
- About 50% (1 out of 2) of $a$'s hash to ***0.
- About 25% (1 out of $2^2$) of $a$'s hash to **00.
- About 12.5% (1 out of $2^3$) of $a$'s hash to *000.
- So, if we saw the longest tail end = i, then we have probably seen about $2^i$ distinct items so far.

# Combining Several Hash Functions

- Perhaps, by bad luck, a seen object hashes to a bucket with a very high tail end.
- To limit the probability of error, we can combine the tail ends of multiple independent hashes.
  - The average is not reliable since it is sensitive to very large numbers.
    - That's why, for example, real estate agents use the median to compare house prices among districts.
  - The median would only produce numbers that are powers of 2.
  - The solution is to take the median of the averages.
    - Partition the $K$ hashes into groups of $G$ hashes each.
    - Compute the average of the values in each group.
    - Compute the median of the averages.

# Programme

1. Data Streams
2. Sampling and Filtering
3. Counting Elements in a Stream
   - Counting Distinct Elements
   - Exponentially Decaying Windows

# The Problem of Most-Common Elements

- Suppose we have a stream whose elements are the movie name and the number of tickets sold in the current week.
- We want to find out what are the most popular movies "currently".
- Which of these movies is more popular "currently"?
  - Movie 1 sold *n* tickets each week for the last 5 weeks.
  - Movie 2 sold 2*n* tickets last week but none in the previous weeks.
  - Movie 2 sold 10*n* tickets last year but none in the last weeks.
- Depending on what we mean by "currently" we may have a different answer.
- With exponentially decaying windows, we give more importance to the most recent items.

# Definition of a Decaying Window

- A decaying window keeps a smooth aggregation of all the counts from the beggining of the stream.
- More recent counts are given more importance.
- If the stream is $a_1, a_2, \ldots$, then the sum of all values with an exponentially decaying window at time $t$ is:

$$\sum_{i=1}^{t} a_i(1-c)^{t-i}$$

where $c$ is a fixed constant, e.g. $10^{-6}$ or $10^{-5}$.

# Update when a New Item Arrives

- If we already know $s = a_t + a_{t-1}(1-c) + a_{t-2}(1-c)^2 + \ldots$
- Then we can easily update $s$ when a new $a_{t+1}$ arrives.
- We simply compute $s \leftarrow s(1-c) + a_{t+1}$.

**Proof**

$$\begin{aligned}
a_{t+1} + s(1-c) &= \\
a_{t+1} + (a_t + a_{t-1}(1-c) + a_{t-2}(1-c)^2 + \ldots)(1-c) &= \\
a_{t+1} + a_t(1-c) + a_{t-1}(1-c)^2 + a_{t-2}(1-c)^3 + \ldots
\end{aligned}$$

# Example: Finding the Most Popular Elements

- Suppose we want to keep counts of the tickets of the most popular movies currently.
- We will generate a stream per movie with 1 each time a ticket for that movie appears in the stream, and 0 otherwise.
- We set $c$ and a threshold of "importance", say 0.5.
- We will keep counts of tickets for those movies whose score is greater than 0.5.
- When a new ticket arrives to the stream:
    1. For each movie whose score we are maintaining, multiply its score by $(1 - c)$.
    2. If the new ticket is for a movie $M$:
        1. If are maintaining the score for $M$, add 1 to that score.
        2. If there is no score for $M$, create one and add 1 to that score.
    3. If any score falls below the threshold 0.5, drop that score.

# How many movies are we maintaining?

$$s = \sum_{i=1}^{t} a_i (1-c)^{t-i}$$

- The sum over all weights in an infinite stream is $\sum_{t=0}^{\infty}(1-c)^t = 1/c$.
- Thus, there cannot be more than $2/c$ movies with score $1/2$ or more.
- In practice, the number of movies with score $1/2$ or more is much less than $2/c$.
- So we can adjust $c$ and the threshold to determine the maximum number of movies we want to keep track at any time.

# Sliding versus Decaying Windows

## Sliding Window

- Keeps summaries of the last $N$ elements.
- All elements have the same weight.
- We need to worry about the element that falls out of the window when we update the summaries.

## Decaying Window

- Keeps summaries of the last $N$ elements.
- More recent elements have higher weight.
- Easy to update the summaries.

# Take-home Messages

- The Stream Data Model.
- Sampling of streams.
- Bloom filters.
- Counting distinct elements.
- Counting in the last N elements.

# What's Next

### Week 9

- Catch-up tutorial on Monday (see announcement in iLearn)
- Assignment 3 ready
- Link Analysis