Streaming Computation

Data Analytics Lifecycle

[EMC Corporation 2014]

Data Analytics Lifecycle

[EMC Corporation 2014]

Data Analytics Lifecycle

Volume
Velocity
Variety
Veracity
…

[EMC Corporation 2014]

Common Big Data Challenges

- **Volume:**
  - ***Too much data to store/process*** — reduce processing time by using distributed & parallel computing
    - Handling social media/IoT data
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*
- **Velocity** — ***data comes in real-time*** → cannot store → process **on the fly**!
    - Detecting network failures from inspecting billions of packets per hour

Scaling: dealing with Big Data Volume and Velocity

- **Scale up** — *make the process more scalable on a* **single** *computer*
  - Some distinguishes “scale up” & “scale down” (scaling is a relative term) [Phillip B. Gibbons — IPDPS 2015]
  - ***Scale down*** the amount of data processed or the resources needed to perform the processing
  - ***Scale up*** the computing resources on a node, via parallel processing & faster memory/storage
    - or *scale up* the efficiency/performance of the process!
- **Scale out** — *add more computing resources using* **multiple** *computers*

Common Big Data Challenges

- **Volume:**
  - ***Too much data to process*** — reduce processing time by using distributed & parallel computing (next class)
    - Handling social media/IoT data
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*
- **Velocity** — ***data comes in real-time*** → cannot store → process **on the fly**!
    - Detecting network failures from inspecting billions of packets per hour

Common Big Data Challenges

- **Volume:**
  - ***Too much data to process*** — reduce processing time by using distributed & parallel computing (next class)
    - Handling social media/IoT data
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given :

Scale up with Streaming!

- **Velocity** — ***data comes in real-time*** → cannot store → process **on the fly**!
    - Detecting network failures from inspecting billions of packets per hour

Streaming Algorithms

- An active area of *theory* computation:
  - Study complexity (big O notation), error bounds, correctness, etc.
- Started in the 1970s, but getting more popular thanks to its applications in massive data processing
- Big data world adopts the stream processing model
  - Process data as soon as it arrives (asynchronous updates)
  - Continuous and incremental processing

Streaming Computation

- Given a data series of **n** elements: $<a_1, a_2, \ldots, a_n>$ (**n** is usually known) that *can only be examined in* ***a limited number of passes*** (usually one).
- Compute a function of stream, e.g. average, median, distribution (histograms), distinct elements, etc.

Streaming Computation

- Given a data series of **n** elements: $<a_1, a_2, \ldots, a_n>$ (**n** is usually known) that *can only be examined in* ***a limited number of passes*** (usually one).
- Compute a function of stream, e.g. average, median, distribution (histograms), distinct elements, etc.
- Constraints:
  - **Limited working memory** of size **m** (**m** << **n**)
  - Elements are accessed sequentially (**no random access**)
  - **Fast processing** time per element

Example: find the missing number

- There are 11 soccer players, whose shirt numbers are from 1 to 11, walking from the tunnel to the field. But only 10 of them arrives.
- How to determine the missing number in the “stream” of players?

Example: find the missing number

- There are 11 soccer players, whose shirt numbers are from 1 to 11, walking from the tunnel to the field. But only 10 of them arrives.
- How to determine the missing number in the “stream” of players?
- Constraints:
  - We can only *look at a number* at a time
  - We can only *“store” a number* in our head

8 2 6 1 10 3 5 11 9 7

Example: find the missing number

- It's **4**

Example: find the missing number

- It's **4**
- The sum of all numbers is fixed: (1+11)*11/2 = **66**

8 2 6 1 10 3 5 11 9 7

Example: find the missing number

- It's **4**
- The sum of all numbers is fixed: (1+11)*11/2 = **66**
- Record only the sum of the numbers as we scan through the stream

Example: find the missing number

- It's **4**
- The sum of all numbers is fixed: (1+11)*11/2 = **66**
- Record only the sum of the numbers as we scan through the stream
- 8,10,16,17,27,30,35,46,55,**62**

Example: find the missing number

- It's **4**
- The sum of all numbers is fixed: (1+11)*11/2 = **66**
- Record only the sum of the numbers as we scan through the stream
- 8,10,16,17,27,30,35,46,55,**62**
- Our missing number is 66 - 62 = **4**!

8 2 6 1 10 3 5 11 9 7

Approximation and Randomization

- Many things are hard to compute exactly over a stream
  - Requires linear space to compute exactly
- **Approximation:** find an answer correct within some factor (e.g. 10% correct)
  - More generally, a $(1 \pm \varepsilon)$ factor approximation
- **Randomization:** allow a small probability of failure (e.g. probability 1 in 10,000)
  - More generally, success probability $(1-\delta)$
- Approximation and Randomization: (ε, δ)-approximations
  - Example: Actual answer is 5 ± 1 with prob ≥ 0.9

[M. Garofalakis, Yahoo! Research]

Data Stream Pseudo-Pipeline

- Potential queries: most frequent items, membership query, cardinality estimation, etc.

[M. Garofalakis, Yahoo! Research]

How to handle data streams?

- Many representations of the streaming model:
  - Time-Series (most popular in urban data, **storing temporal values**)
  - Cash-Register (mini histogram on data arrivals, **storing counts**)
  - Turnstile (arrival-departure summarization, **storing deltas**)
  - Sliding-window (keep a continuous subset of the input, **storing a buffer**)
- Many classes of techniques to process data elements, aka. “Stream Processor”:
  - Sampling (reduce data inputs)
  - Sketching (data aggregation)
  - Dimensionality Reduction (SVD, PCA, etc.) — *not discussed*
  - Summarization (Wavelets, Fourier Transform, etc.) — *not discussed*

Sampling Techniques

- Motivation: a small random sample of the data can be a good representative of all the data
- Action: sample the input data based on some probability model
  - E.g. sample **m** items **uniformly** from the stream (sample size=**m**)
- A challenge for data streams of unknown size

Why Sampling?

- Sampling has an intuitive semantics
  - Same data structure as the original data stream
- Sampling is general and agnostic to the analysis to be done
  - Many summarization methods only work for certain computations
- Sampling is (usually) easy to understand
  - So prevalent that we have an intuition about sampling
- Small enough to experiment with various models

[Sampling for Big Data, KDD 2014]

Example

- **n**=12, naively take one for every 3 items (**m**=4)

Stream	**8**	**2**	**6**	**1**	**4**	**3**	**5**	**8**	**9**	**4**	**6**	**2**
Sample	**8**			**1**			**5**			**4**		

- To compute the mean of **odd** items (index starting at 1)
  - Stream: (8+6+4+5+9+6)/6 ~ 6.333…
  - Sample: (8+5)/2 = 6.5
- Works okay but not uniformly selected: some with 100%, some with 0%
  - If each item correspond to a day of the week, certain days will be ignored

Bernoulli Sampling

- The easiest possible case of sampling: all weights are 1
  - **n** objects, and want to sample **m** from them uniformly
  - Each possible subset of **m** should be equally selected
- Uniformly sample an index from **n**, **m** times
- Issues:
  - Must know **n** beforehand
  - Assume that random number generators are good enough
  - Cannot be done progressively (i.e. showing results mid-stream)

Reservoir Sampling

- **Uniformly** sample a stream of size **n** with sample size **m** (**m**<<n)
- Steps:
  - For the first **m** elements, put on the sample/buffer
  - For any subsequent **i**'th element, with probability **m / i**, place it on the sample buffer **randomly** (at position 1 to **m**)
- Probability **i**'th element is the sample of the stream is: **m / n**
  - (probability **i** is sampled on arrival) **x** (probability **i** survives to end)

(**m / i**) **x** (**i / n**)*

** can be proved by induction*

Sample Python Code

```
import random

### S has items to sample, m is the sample size
def ReservoirSample(S, m):
    n = len(S)
    R = []

    # Fill the first m elements
    for i in range(m):
      R.append(S[i])

    ## replace elements with gradually decreasing probability
    for i in range(m,n):
        j = random.randint(0, i-1)
        if j<m:
            R[j] = S[i]

    return R
```

Min-Hashing Reservoir Sampling

- Assign each element a random value from **0** to **1**, called *rank*
- Select the **m** elements with the smallest *ranks*
- Each element has the same chance of getting the smallest rank, so uniformly selected
- Implementation:
  - Can put everything into a list, and sort, but inefficient
  - Use a priority queue, or Python's *heapq* — Heap Queue algorithm

Sample Python Code

```
import heapq
import random
heapq.nsmallest(m, zip(iter(random.random,1), range(n)))
```

Further on Sampling Techniques

- Book: *Sampling Algorithms*, by Yves Tille, 2006

Sampling Techniques

- Mostly acceptable for showing patterns

Sampling Techniques

- But challenging for detecting anomalies!

Sketching

- Not every problem can be solved with sampling:
  - Compute the cardinality (e.g. the number of distinct elements)
- Motivation: we don't want to store everything, but it's okay to inspect all data elements once, and keep a few!
- Sketch: project data into a "sketch" space, progressively build the function of stream, e.g. accumulation/aggregation
- The sketch size should be << the stream size

Sketching Examples

Sketching Examples

- Compute a stream average: sketch the (sum, count)
  - For each element, add its value to the sum and increase the count
- Compute a stream histogram: sketch the count per category
  - For each element, add to the count of its category
- Compute a stream heatmap: sketch the “image”
  - For each element, add a KDE to its pixel location

Example: find the median value

- How can we find the median of a large array with streaming?

Example: find the median value

- How can we find the median of a large array with streaming?
  - Why is it a challenge to do in a streaming fashion?

Example: find the median value

- How can we find the median of a large array with streaming?
  - Why is it a challenge to do in a streaming fashion?
- In data science projects, sometime, we’re only interested in the median range, e.g what is the median salary (range) of non-founder executives?

Example: find the median **range**

- Build a histogram of the values based on interested value ranges
- Select the 50 percentile bucket
- It's the first bucket that passes the 50% mark!
- It is **$210,000**

[*Nature Biotechnology* **32**, 393–395 (2014)]

[Katsov, I. 2012]

Raw Data

- 10^7 elements, each is a 32-bit number (~4 bytes each)
  - The amount of memory needed to store the data is `4 x` 10^7 `= 40MB`
- 10^6 distinct values, e.g. numbers from 1 to 1 million

Exact Frequency Query

- Determine how often a value has been seen in a data stream
- In Python, we could use a `dict()`
- Memory consumption:
  - `freq` needs to store 10^6 *pairs* (`value`,`counter`) = `7 x` 10^6 = `7MB`
    - value is 32-bit or 4 bytes
    - counter is 24-bit or 3 bytes

```
freq = dict()
for i in S:
    freq[i] = freq.get(i,0)+1

print(freq.get(i,0))
```

Exact Membership Query

- Determine if an element has been seen in a data stream
- Naively, in Python, we could use a `set()` or we could use a sorted array
- The memory consumption would include:
  - A list of 10^6 unique values:
    - $4 \times 10^6 = 4MB$

Exact Membership Query

- Determine if an element has been seen in a data stream
- Naively, in Python, we could use a `set()` or we could use a sorted array
- The memory consumption would include:
  - A list of 10^6 unique values:
    - $4 \times 10^6 = 4MB$

```
seen = set()
for i in S:
    seen.add(i)

print(my_value in seen)
```

Use hash-function to reduce space

- Instead of storing all values into the set **`seen`**, we will store a reduced amount of information of the values (e.g. certain digits of the values)
  - These are called *hashed values*
- For example:
  - Using only the last digits
  - The size of **`seen`** is 10

```
seen = set()
for i in S:
    seen.add(i%10)

print((my_value%10) in seen)
```

Use hash-function to reduce space

- Instead of storing all values into the set **`seen`**, we will store a reduced amount of information of the values (e.g. certain digits of the values)
  - These are called *hashed values*
- For example:
  - Using only the last digits
  - The size of **`seen`** is 10

```
seen = set()
for i in S:
    seen.add(i%10)

print((my_value%10) in seen)
```

What's wrong with this approach?

Use hash-function to reduce space

- Instead of storing all values into the set **`seen`**, we will store a reduced amount of information of the values (e.g. certain digits of the values)
  - These are called *hashed values*
- For example:
  - Using only the last digits
  - The size of **`seen`** is 10

```
seen = set()
for i in S:
    seen.add(i%10)

print((my_value%10) in seen)
```

What's wrong with this approach?

False Positives

We can mitigate the issue with additional hash functions!

- Use the second to last digit as a secondary hash value
  - There are two "**`seen`**"
  - The False Positive rate is reduced since each value must pass two hashed checks.

```
seen = [set(),set()]
for i in S:
    seen[0].add(i%10)
    seen[1].add((i//10)%10)

hv0 = my_value%10
hv1 = (my_value//10)%10
print ((hv0 in seen[0]) and
       (hv1 in seen[1]))
```

Membership Query: Bloom Filter

- A Bloom filter is an array of **m** bits/items, together with a number of hash functions
- The argument of each hash function **h(x)** is a stream element, and it returns a position in the array
- Initially, all bits are **0**
- When input x arrives, we set to **1** the bits **h(x)**, for each hash function **h**

[J. Ullman, 2014]

Bloom Filter: Example

- Our filter has 11 items, **m=11**
- Stream elements = integers.
- Use two hash functions:
  - $\mathbf{h_1(x)}$ =
    - Take odd-numbered bits in the binary representation of **x**
    - Treat it as an integer **i**
    - Result is **i** modulo 11.
  - $\mathbf{h_2(x)}$ = same, but take even-numbered bits.

[J. Ullman, 2014]

Bloom Filter: Example

Stream element	h_1	h_2	Filter contents
			00000000000
25 = 11001	5	2	00100100000
159 = 10011111	7	0	10100101000
585 = 1001001001	9	7	10100101010

Note: bit 7 was already 1.

[J. Ullman, 2014]

Bloom Filter: Lookup

- Suppose element **y** appears in the stream, and we want to know if we have seen **y** before
- Compute **h(y)** for each hash function **h**
- If **all** the resulting bit positions are **1**, say we have seen **y** before
  - We could be wrong
    - Different inputs could have set each of these bits
- If **any** of these positions is **0**, say we have not seen **y** before
  - We are certainly right

[J. Ullman, 2014]

Bloom Filter: Example

- Suppose we have the same Bloom filter as before, and we have set the filter to 10100101010
- Lookup element y = 118 = 1110110 (binary)
- **$h_1(y)$** = 14 modulo 11 = 3
- **$h_2(y)$** = 5 modulo 11 = 5
- Bit 5 is 1, but bit 3 is 0, so we are sure y is not in the set

[J. Ullman, 2014]

Bloom Filter: Performance

- Probability of a false positive depends on the density of 1's in the array and the number of hash functions

  = (fraction of 1's)$^{\text{\# of hash functions}}$

- The number of 1's is approximately the number of elements inserted times the number of hash functions
  - But collisions lower that number slightly
  - The probability after inserting **n** elements using **k** hash functions:

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

Bloom Filter: Performance

- Probability of a false positive depends on the density of 1's in the array and the number of hash functions

  = (fraction of 1's)$^{\text{\# of hash functions}}$

- The number of 1's is approximately the number of elements inserted times the number of hash functions
  - But collisions lower that number slightly
  - The probability after inserting **n** elements using **k** hash functions:

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

"optimal" **k**

$$k = \frac{m}{n} \ln 2$$

Bloom Filter in our example

- $m = 0.8MB \times 8 = 6.4 \times 10^6$
- $n = 10^6$ (distinct values)
- $k = (m/n) \times \ln 2 \sim 4$
- Error rate = $(1 - e^{-kn/m})^k = (1 - e^{-4/6.4})^4 \sim 4.6\%$

Given a Bloom Filter with 10 items (m=10) that operates on two hash functions:

```
h1(x) = (x*7+4) % m
h2(x) = (x*13+9) % m
```

After inserting some elements, our filter has the following bit contents:

Bit Index	0	1	2	3	4	5	6	7	8	9
Bit Value	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**1**

Have we seen **16**, **23**, and **34**?

Membership Query for 16

- h1 = (16*7+4) % 10 = 6
- h2 = (16*13+9) % 10 = 7
- Both bit values are 0
- The answer is NO

Membership Query for 23

- h1 = (23*7+4) % 10 = 5
- h2 = (23*13+9) % 10 = 8
- Bit value 8 is 0
- The answer is NO

Membership Query for 34

- h1 = (34*7+4) % 10 = 2
- h2 = (34*13+9) % 10 = 1
- Both bit values are 1
- The answer is MAY BE (cannot say YES for sure)

Exact Frequency Query

- Determine how often a value has been seen in a data stream
- In Python, we could use a `dict()`
- Memory consumption:
  - `freq` needs to store 10^6 *pairs* (`value`,`counter`) = `7 x` 10^6 = `7MB`
    - value is 32-bit or 4 bytes
    - counter is 24-bit or 3 bytes

```
freq = dict()
for i in S:
    freq[i] = freq.get(i,0)+1

print(freq.get(i,0))
```

Frequency Query: Count-Min Sketch

- Creates a small summary as an array **CM** of **w** x **d** in size
- Use **d** hash functions to map vector entries to **[1..w]**
- Each entry in input vector **S[]** is mapped to one bucket per row
  - **h()**'s are pairwise independent
- Merge two sketches by entry-wise summation
- Estimate **S[j]** by taking $\mathbf{min_k\{CM[k,h_k(S[j])]\}}$

Count-Min Sketch: Guarantees

- *[Cormode, Muthukrishnan'04]* CM sketch guarantees approximation error on point queries less than $\varepsilon||S||$ in space $O(1/\varepsilon \log 1/\delta)$

- Probability of more error is less than $1-\delta$

Count-Min Sketch

- 192 counters
- 10,000 elements
- 8,500 distinct values
- Find 100 most frequently used words
- Not so good estimation

[Katsov, I. 2012]

Count-Min Sketch

- 192 counters
- 80,000 elements
- 8,500 distinct values
- Find 100 most frequently used words
- Much better for highly skewed data

[Katsov, I. 2012]

Count Min Sketch in our example

- delta = `0.04`, so the number of hash functions is `ln(1/0.04) ~ 3`
- The width is `4000`, epsilon is `0.0005`
- The total size is `4000x4x3 = 48KB`

[Katsov, I. 2012]

Linear Counter — projecting the counts

```
class LinearCounter:
    ### Over-simplified LinearCounter
    def __init__(self, m):
        self.mask = set()
        self.m = m

    def add(self, value):
        position = id(value) % self.m
        self.mask.add(position)

    def cardinality(self):
        w = len(self.mask)
        if w==self.m:
            return w # could be infinite
        return -self.m*math.log((self.m-w)/self.m)
```

[Katsov, I. 2012]

Linear Counting

$\hat{n} = -m \ln \frac{m-w}{m}$	$\hat{n}$ - cardinality estimation w - mask weight (a number of 1's) m - mask size
$bias(\frac{\hat{n}}{n}) = E(\frac{\hat{n}}{n}) - 1 = \frac{e^t - t - 1}{2n}$	This equation expresses a bias of the estimation (the ratio between estimation and true cardinality) as a function of the load factor and expected cardinality (or upper bound). t - load factor, n/m E(.) - mathematical expectation n - maximum cardinality (or upper bound, or capacity)
$m > \max(5, 1/(\varepsilon t)^2) \cdot (e^t - t - 1)$	A practical formula that allow one to choose m by the standard error of the estimation. m - mask size ε - standard error of the estimation t - load factor, n/m

Assume equal probability for new distinct values!

[Katsov, I. 2012]

Linear Counter

[Katsov, I. 2012]

LogLog Counter

- Tracks the rarest element in the data set
- Estimate the cardinality based on this information (the rank — **r**)
- *Intuition: getting all heads in flipping coins, what are the odds?*
- The rank value **r** has high variance
  - Count multiple of them and take the mean value

[Katsov, I. 2012]

LogLog Counter

- H is the hashed length (in bits)
- 1024 estimators (m) ~ 4% std error
- 5-bit buckets (etype=5) can support a load factor of close to 1 billion
- 8-bit buckets (etype=8) can support extremely large cardinalities
- HyperLogLog = LogLog + Harmonic Mean (lower std error)

$\hat{n} = \alpha_m \cdot m \cdot 2^{1/m \sum_j estimators[j]}$ $\alpha_m = \Gamma(\frac{-1}{m})\frac{1-2^{\frac{1}{m}}}{\ln 2} \overset{m>64}{\approx} 0.39701$	$\hat{n}$ – cardinality estimation m – number of buckets (estimators) α_m – estimation factor, close to 0.39701 for m > 64, i.e. for most of practical applications
$\varepsilon \approx \frac{1.30}{\sqrt{m}}$	Dependency between the standard error of the estimation and the number of buckets (estimators). ε - standard error of the estimation m – number of buckets (estimators)
$H = \log_2 m + \lceil \log_2(n/m) + 3 \rceil$	A practical formula for length of the hash function. m – number of buckets (estimators) n – maximum cardinality (i.e. capacity)
$etype \Leftarrow \lceil \log_2 \lceil \log_2(n/m) + 3 \rceil \rceil$	A number of bits in etype is determined by the maximal possible rank. The rank is limited by H, so the length of etype is a log of H (except the part that is used for bucket ID computation).

[Katsov, I. 2012]

A LogLog algorithm with 5-bit bucket

- hash each item.
- use the first 5 bits of the hash to decide which of 32 buckets to use
- use the remaining 27 bits of the hash to update the max-leading-zeros count for that bucket
- average the max leading zeros seen in each bucket to get k
- return an estimate of $32\alpha \cdot 2^k$, where $\alpha \approx 0.7$ is a correction factor

[moderndescartes.com]

A HyperLogLog algorithm with 5-bit bucket

- hash each item.
- use the first 5 bits of the hash to decide which of 32 buckets to use
- use the remaining 27 bits of the hash to update the max-leading-zeros count for that bucket
- take the harmonic mean $HM = \frac{32}{\Sigma 2^{-k_i}}$, over each bucket's count k_i.
- let $E = 32\alpha \cdot HM$
- if $E < 2.5 \cdot 32$ and number of buckets with zero count V > 0: return $-32 \cdot \log(V/32)$
- else if $E > 2^{32}/30$: return $-2^{32} \log(1 - E/2^{32})$
- else: return E

[moderndescartes.com]

Good read on Sketching

- *Probabilistic Data Structures for Web Analytics and Data Mining*, Ilya Katsov, 2012.

  https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/

- *A practical introduction to the Count-Min Sketch*, Hannes Korte, 2013.

  http://hkorte.github.io/slides/cmsketch/

- *HyperLogLog Demo*

  http://content.research.neustar.biz/blog/hll.html

Further Readings on Streaming

- *Data streams: Algorithms and applications*, Muthukrishnan, S. ACM-SIAM Symposium on Discrete Algorithms, 2003. (Background)
- *Selection and sorting with limited storage*, Munro, J. and Paterson, M., Theoretical Computer Science (TCS) 12, 1980. (Median)
- *Probabilistic counting algorithms for data base applications*, Flajolet P. and Martin G., Elsevier JCSS, 31, 1985. (Distinct Element)
- *An Improved Data Stream Summary: The Count-Min Sketch and Its Applications,* Cormode, G. and Muthukrishnan, S., Theoretical Informatics 2004. (Frequency)
- *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm,* Flajolet et al., AofA 2007. (Count Distinct Values).

Data Streaming in Python

- Given a data series of **n** elements: $<a_1, a_2, \ldots, a_n>$ (**n** is usually known) that *can be examined only a few passes* (typically just one).
- Compute a function of stream, e.g. average, median, distribution (histograms), distinct elements, etc.
- Constraints:
  - Limited working memory of size **m** (**m** << **n**)
  - Elements are accessed sequentially (no random access)
  - Fast processing time per element

Data Streaming in Python

Python's **list,** NumPy's **array,** panda's **Series, etc.**

- Compute a function of stream, e.g. average, median, distribution (histograms), distinct elements, etc.
- Constraints:
  - Limited working memory of size **m** (**m** << **n**)
  - Elements are accessed sequentially (no random access)
  - Fast processing time per element

Data Streaming in Python

Python's **list,** NumPy's **array,** panda's **Series, etc.**

- Compute a function of stream, e.g. average, median, distribution (histograms), distinct elements, etc.

Python's **generators** and **iterators** (on top of **iterables**)

iterable & iterator

- Anything that can be read sequentially in Python is a ***iterable***: dict, string, list, tuple, etc. This is practically anything that can be put inside a **for** loop so that we can sequentially *iterate* through every elements.

```
A = [i for i in range(5)] # or list(range(5))
print('OUTPUT:')
for i in A:
    print(i)
# OUTPUT: 0 1 2 3 4
```

- We can repeatedly iterate through an iterable as many times as we'd like since all the elements are already stored in memory (A is a list)

- Each time you call (aka applying a **for** loop) on an iterable object, it creates a new ***iterator*** automatically to feed the data from memory

generator

- ***generators*** are special classes for creating *iterators* that can only be traversed once.

```
A = (i for i in range(5)) # or range(5)
print('OUTPUT:')
for i in A:
    print(i)
print('OUTPUT:')
for i in A:
    print(i)
# OUTPUT: 0 1 2 3 4
# OUTPUT:
```

- generators can be used as data streams as it usually **doesn't store** everything in memory

Example generators

- Python's **file** object: we can only iterate through the file contents **ONCE**

```
with open('filename.csv', 'r') as f:
    for line in f:
        pass
    size = 0
    for line in f:
        size += len(line)
    print('OUTPUT:', size)
# OUTPUT: 0
```

- urllib.request.urlopen() returns object where we can only read **ONCE**

Example of non-generators

- Panda's DataFrame, DataSeries
- Numpy's array, matrix
- Python's dict, list, string, pickled objects, etc.

How to create a generator?

- generators are functions BUT use the keyword **yield** in place of **return**
- Example: create a generator of square numbers from 1 to 100
- The code in function body only runs each time the for uses the generator to “generate” data
- The generator is done when the function finishes

```
def squared_numbers():
    for i in range(10):
        yield i**2

numbers = squared_numbers()
print('FIRST:')
for i in numbers:
    print(i)
print('SECOND:')
for i in numbers:
    print(i)
# FIRST: 0 1 4 9 16 25 36 49 64 81
# SECOND:
```

Streaming Summary

- Benefits of Streaming:
  - Reduce the memory footprint
  - Enable pipelining, aka. pushing data out as soon as possible
- Challenges of Streaming:
  - Not every algorithm is streamable
  - Many are not trivial to approximate
- Handling data streams in Python:
  - Generators and Iterators

Reading for Next Class

- *Hadoop: The Definitive Guide — Storage and Analysis at Internet Scale*
  - Chapter 3: The Hadoop Distributed Filesystem