CSEP 521: Applied Algorithms
Lecture 11
Stream Algorithms: Hyperloglog

Richard Anderson

February 9, 2021

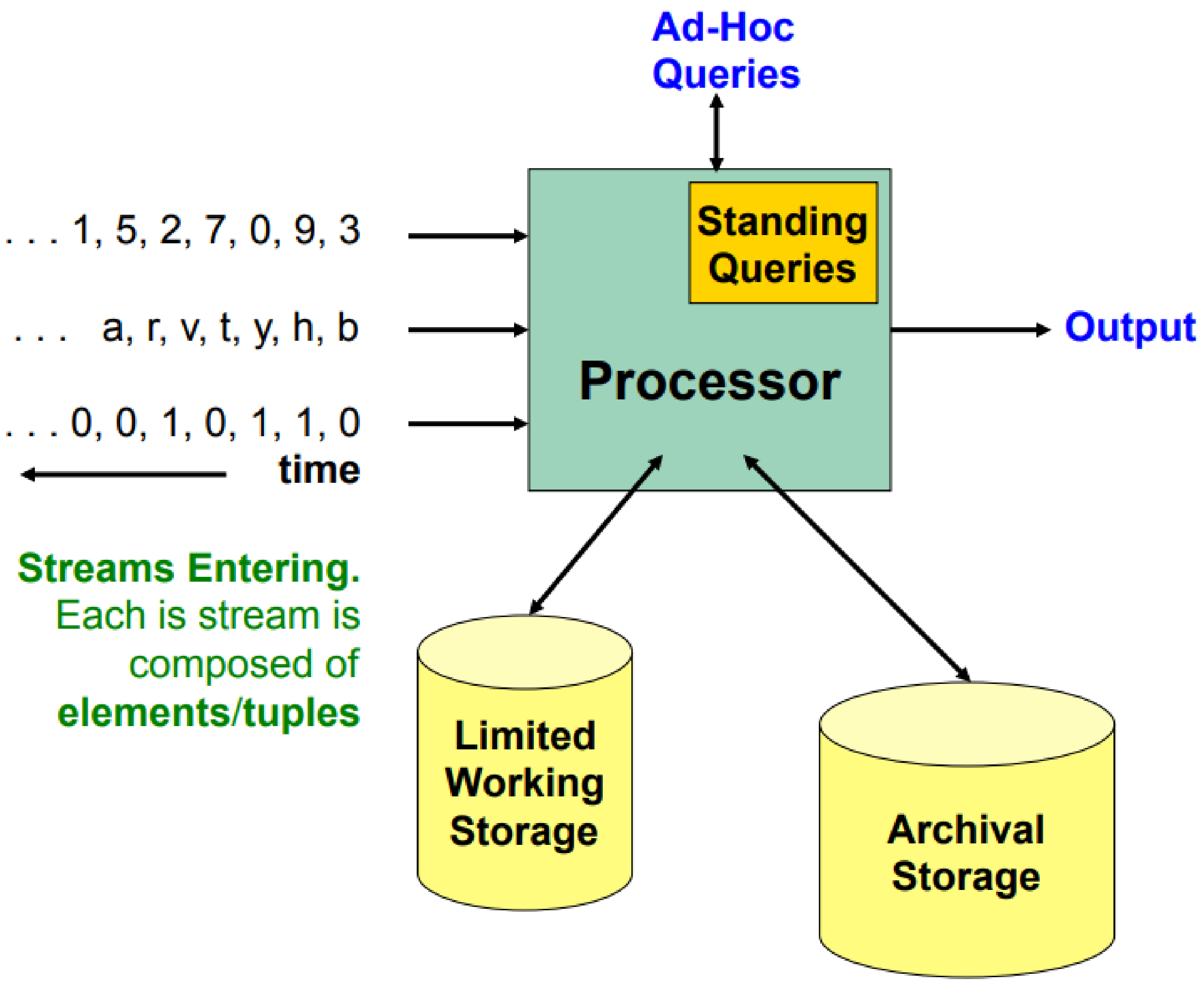
Announcements

- HW6
  - Implement count min and test on provided dataset

- Course material – streaming algorithms
  - Today: Count number of distinct elements: Hyperloglog
  - Thursday: Determine the second moment
    - $\sum f_x^2$
    - Alon, Matias, Szegedy

Algorithms for data streams

- Data items received one at a time, N is number of items received
- Computation performed on each data item
- Memory is limited to being much less than N
  - Memory in thousands
  - Data in millions
- Motivation
  - Large scale computation
  - Estimates are often good enough



Results so far

- Easy things that require little memory
  - Computing the maximum, sample a random element
- Impossible things (provably require $\Omega(n)$ memory)
  - Is there an element with frequency $\geq n/3$, find the median
- Identification and approximate estimates of high frequency items

Warmup

- Suppose we want to track j smallest items in a stream
  - Space complexity
  - Time complexity

Count distinct problem

- FB Distinct visitors per week
- How many distinct IP addresses accesses a website

Solutions

- Sort, remove duplicates
  - sort –u foo.txt | wc
- Hashing
- Bloom Filter

```
public int CountUnique(List<string> strings){
    Dictionary<string, int> dict
                  = new Dictionary<string, int>();

    foreach (string str in strings)
        dict.TryAdd(str, 1);

    return dict.Count;
}
```

Estimator

- Assign each distinct element a random value, in [0, 1)
  - By hashing, of course
  - Each copy of the same item has the same hash value
- Compute the minimum value: min_val
  - Only remember a single data item
- If the number of distinct items is K, the expected value of min_val is 1/K
- Report the estimate 1/min_val

Improve the estimator

- Track the J smallest values
  - J values need to be maintained
  - Compute min_val_J
- Expected value of min_val_J is J/K
- Report estimate of J/min_val_j

- Probabilistically, this is a far more robust estimator

Hyperloglog

- Estimate number of distinct elements
  - Estimate cardinalities of > 10^9 with accuracy 2% using 1.5 kB of memory
- Derived from a 1984 theoretical result
- Suggested practical applications in early 2000s
- Derivatives are now used as a practical tool by large internet companies
  - **`APPROX_COUNT_DISTINCT`** in BigQuery
  - Reddit, to count unique views of posts

- From this discussion
  - Key ideas
  - Basic algorithm
  - But the analysis still contains some magic

Log Log idea

- What is the probability that a random number has exactly k consecutive one's (in binary) in low order bits
  - 10100010100101001001001001111
  - Define $\rho(x)$ as consecutive ones at end of hash(x)

- Estimate the cardinality of $\{x_1, x_2, \ldots, x_M\}$ as 2^Q where $Q = \max\{\rho(x_1), \rho(x_2), \ldots, \rho(x_M)\}$
- $\rho(x) = k$ with probability $2^{-(k+1)}$
- Q is an estimate of log M, so Q can be stored in log log M bits

Intuition

- If you have one million items, one of them is going to have a hash that ends in: 01111111111111111111

Strengthening the estimate

- Risk of over estimating with a very unlikely hash
- If we have 2^k distinct items, we expect to have items of that have hashes that end with j consecutive 1's for $j \leq k$
- We will need to track all the ρ values, which we will do by keep a bitwise-or

Some bit hacking

- r(x) is the number of trailing 1s in the binary representation of x
- $R(x) = 2^{r(x)}$
- R(x) = ~x & (x+1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	r(x)	R(x)	$R(x)_2$
1	0	1	1	1	1	0	1	1	1	1	1	0	1	0	1	1	2	10
1	0	1	0	1	0	1	0	1	0	0	0	1	1	1	0	0	1	1
0	1	1	0	1	0	0	1	0	1	0	1	1	1	1	1	5	32	100000

0	1	1	0	1	0	0	1	0	1	0	1	1	1	1	1	*x*
1	0	0	1	0	1	1	0	1	0	1	0	0	0	0	0	*~x*
0	1	1	0	1	0	0	1	0	1	1	0	0	0	0	0	*x + 1*
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	*~x & (x + 1)*

Probabilistic Counting Trace

x	*r(x)*	*R(x)*	*sketch*
01100010011000111010011110111011	2	100	00000000000000000000000000000100
01100111001000110001111100000101	1	10	00000000000000000000000000000110
00010001000111000110110110110011	2	100	00000000000000000000000000000110
01000100011101110000000111011111	5	100000	00000000000000000000000000100110
01101000001011000101110001000100	0	1	00000000000000000000000000100111
00110111101100000000101001010101	1	10	00000000000000000000000000100111
00110100011000111010101111111100	0	1	00000000000000000000000000100111
00011000010000100001011100110111	3	1000	00000000000000000000000000101111
00011001100110011110010000111111	6	1000000	00000000000000000000000001101111
01000101110001001010110011111100	0	1	00000000000000000000000001101111

R(sketch) $= 10000_2$

$= 16$

Probabilistic Counting

```
public long R(long x) {
        return ~x & (x+1);
}

public long estimate (iterable<string> stream){
        long sketch;
        for (s : stream)
                sketch |= R(Hash(s));
        return R(sketch);
}
```

- Returns the smallest value not seen
- It can be shown that this off by a factor of 0.77351
  - Established mathematically, and verified experimentally
- Typically off by a binary order of magnitude

Next idea – M independent experiments

- M independent hash functions and average: work, but expensive
- Stochastic averaging
  - Divide stream into 2^m independent streams
  - Use probabilistic counting on each stream, yielding 2^m sketches
  - Compute mean = average number of trailing bits in each sketch
  - Return 2^{mean} / .77531

Constructing the independent experiments

- Assume we have a j bit has function (so hashing to $[0..2^j-1]$)
- Use the first m bits to divide into substreams
- Use the remaining j-m bits as a hash function (into $[0..2^{j-m}-1]$)

101010001010101010001010
010000101010101010011111
010101010101110111010011
100101010100110010010001
001001010100010010100010
110100010100010100101011
100100101110110111010101
101011011101100010010111

Probabilistic Counting Algorithms

- Flajolet-Martin, 1983
- Use of M words to achieve relative accuracy of 0.78/sqrt(M)
- Validated through experimentation
  - Theory doesn't answer questions such as performance with real hash functions or what are the implementational constants
- Many versions now available with modified techniques
  - E.g., different mechanisms for averaging estimates across substreams, harmonic means vs. geometric means