ALWAYS LEARNING

Approximate methods for scalable data mining

Andrew Clegg
Data Analytics & Visualization Team
Pearson Technology

Twitter: @andrew_clegg

Outline

1. Intro
2. What are approximate methods and why are they cool?
3. Set membership (finding non-unique items)
4. Cardinality estimation (counting unique items)
5. Frequency estimation (counting occurrences of items)
6. Locality-sensitive hashing (finding similar items)
7. Further reading and sample code

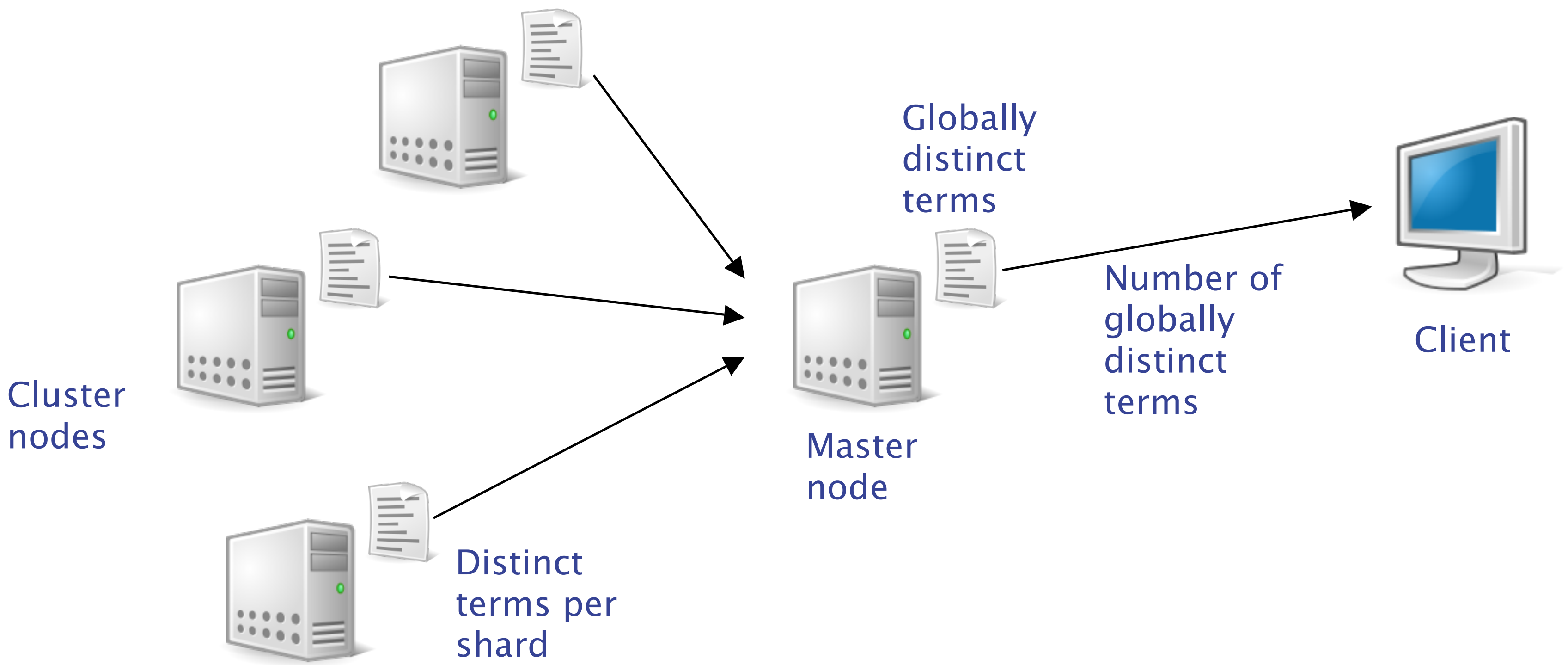
Intro

Me and the team

London

Intro
Motivation for getting into approximate methods

Counting unique terms across ElasticSearch shards

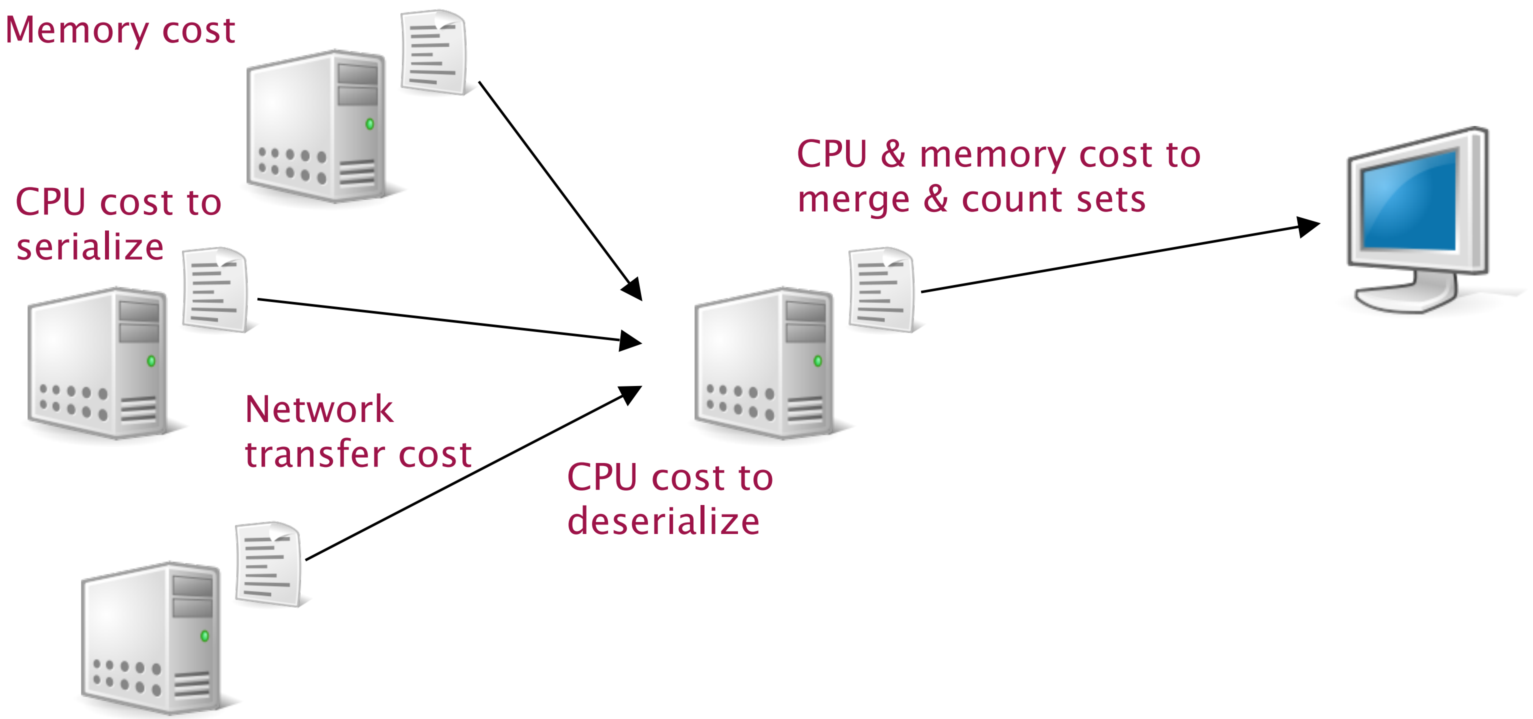


Icons from *Dropline Neu!* http://findicons.com/pack/1714/dropline_neu

Intro

Motivation for getting into approximate methods

But what if each term-set is BIG?



… and what if they're too big to fit in memory?

But we'll come back to that later.

What are approximate methods?
Trading accuracy for scalability

- Often use probabilistic data structures
  - a.k.a. “Sketches”

- Mostly stream-friendly
  - Allow you to query data you haven’t even kept!

- Generally simple to parallelize

- Predictable error rate (can be tuned)

What are approximate methods?
Trading accuracy for scalability

- Represent *characteristics* or *summary* of data

- Use much less space than full dataset (often via hashing)
  - Can alleviate disk, memory, network bottlenecks

- Generally incur more CPU load than exact methods
  - This may not be true in a **distributed** system, overall: [de]serialization for example
  - Many data-centric systems have CPU to spare anyway

Set membership

Have I seen this item before?

Set membership
Naïve approach

- Put all items in a hash table in memory
  - e.g. HashSet in Java, set in Python
- Checking whether item exists is very cheap

- Not so good when items don't fit in memory any more

- Merging big sets (to increase query speed) can be expensive
  - Especially if they are on different cluster nodes

Set membership
Bloom filter

A probabilistic data structure for testing set membership

Real-life example:

BigTable and HBase use these to avoid wasted lookups for non-existent row and column IDs.

Set membership
Bloom filter: creating and populating

- Bitfield of size *n* (can be quite large but << total data size)
- *k* independent hash functions with integer output in [0, *n*–1]

- For each input item:
  - For each hash:
    - Hash item to get an index into the bitfield
    - Set that bit to 1

i.e. Each item yields a unique pattern of *k* bits.

These are ORed onto the bitfield when the item is added.

Set membership
Bloom filter: querying

- Hash the query item with all *k* hash functions
- Are **all** of the corresponding bits set?
  - No = we have never seen this item before
  - Yes = **we have *probably* seen this item before**

- Probability of false positive depends on:
  - *n* (bitfield size)
  - number of items added
- *k* has an optimum value also based on these
  - Must be picked in advance based on what you expect, roughly

Set membership
Bloom filter

Example (3 elements, 3 hash functions, 18 bits)

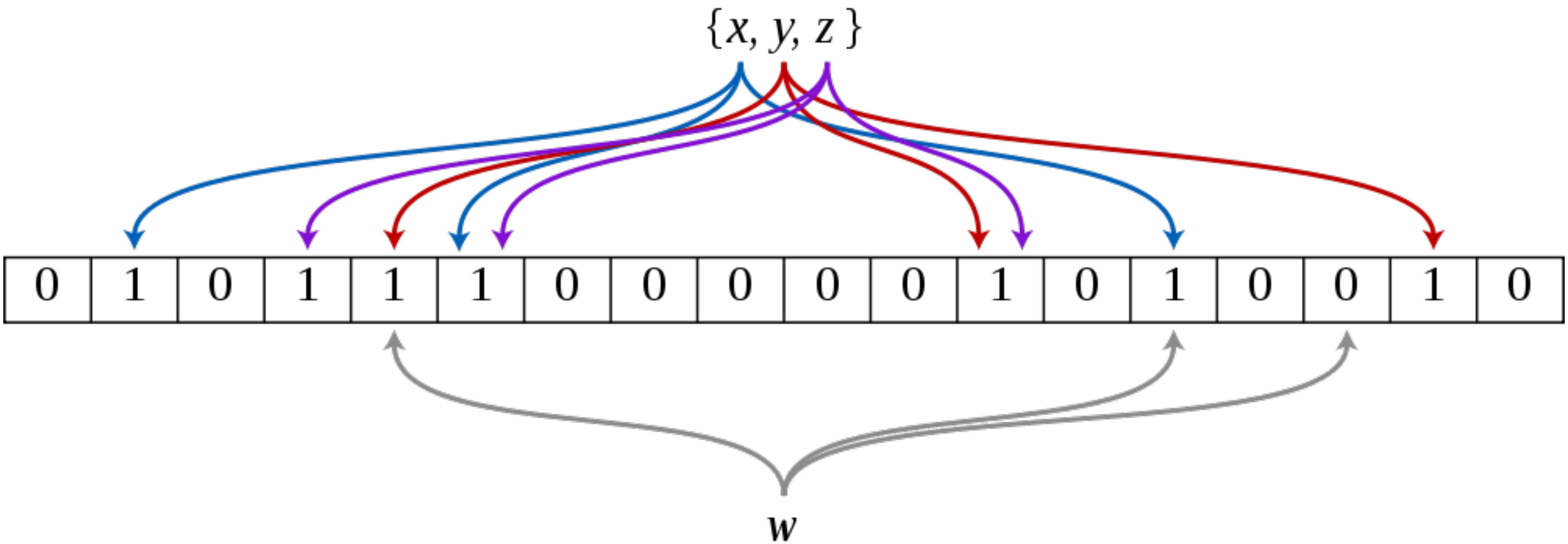


Image from Wikipedia http://en.wikipedia.org/wiki/File:Bloom_filter.svg

PEARSON

Set membership
Bloom filter

Cool properties

- Union/intersection = bitwise OR/AND
- Add/query operations stay at O(*k*) time (and they're fast)
- Filter takes up constant space
  - Can be rebuilt bigger once saturated, if you still have the data

Extensions

- BFs supporting "remove", scalable (growing) BFs, stable BFs, …

Cardinality estimation

How many distinct items have I seen?

Cardinality calculation
Naïve approach

- Put all items in a hash table in memory
  - e.g. HashSet in Java, set in Python
  - Duplicates are ignored
- Count the number remaining at the end
  - Implementations typically track this -- fast to check
- Not so good when items don't fit in memory any more
- Merging big sets can be expensive
  - Especially if they are on different cluster nodes

Cardinality estimation
Probabilistic counting

An approximate method for counting unique items

Real-life example:

Implementation of parallelizable distinct counts in ElasticSearch.
https://github.com/ptdavteam/elasticsearch-approx-plugin

Cardinality estimation
Probabilistic counting

Intuitive explanation

Long runs of trailing 0s in random bit strings are rare.

But the more bit strings you look at, the more likely you are to see a long one.

So **"longest run of trailing 0s seen"** can be used as an estimator of **"number of unique bit strings seen"**.

01110001
11101010
00100101
11001100
11110100
11101100
00010100
00000001
00000010
10001110
01110100
01101010
01111111
00100010
00110000
00001010
01000100
01111010
01011101
00000100

PEARSON

Cardinality estimation
Probabilistic counting: basic algorithm

- Let $n = 0$
- For each input item:
  - Hash item into bit string
  - Count trailing zeroes in bit string
  - If this count > n:
    - Let n = count

Cardinality estimation
Probabilistic counting: calculating the estimate

- n = longest run of trailing 0s seen
- Estimated cardinality ("count distinct") = 2^n … that's it!

This is an estimate, but not actually a great one.

Improvements

- Various "fudge factors", corrections for extreme values, etc.
- Multiple hashes in parallel, average over results (LogLog algorithm)
- Harmonic mean instead of geometric (HyperLogLog algorithm)

Cardinality estimation
Probabilistic counting and friends

Cool properties

- Error rates are predictable
  - And tunable, for multi-hash methods
- Can be merged easily
  - max(longest run counters from all shards)
- Add/query operations are constant time (and fast too)
- Data structure is just counter[s]

Frequency estimation

How many occurences of each item have I seen?

Frequency calculation
Naïve approach

- Maintain a key-value hash table from item -> counter
  - e.g. HashMap in Java, dict in Python

- Not so good when items don't fit in memory any more

- Merging big maps can be expensive
  - Especially if they are on different cluster nodes

Frequency estimation
Count-min sketch

A probabilistic data structure for counting occurences of items

Real-life example:

Keeping track of traffic volume by IP address in a firewall, to detect anomalies.

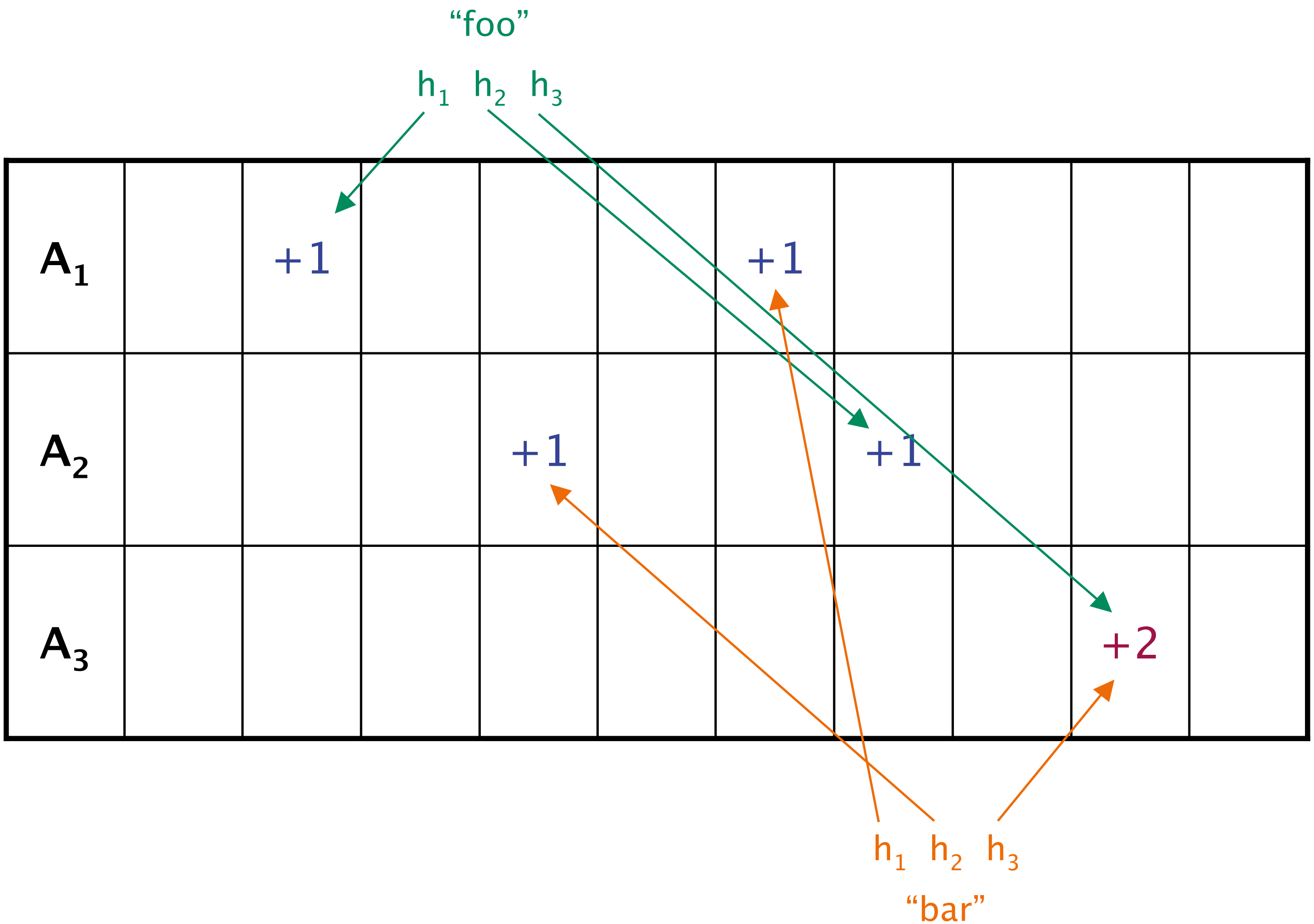
Frequency estimation
Count–min sketch: creating and populating

- *k* integer arrays, each of length *n*
- *k* hash functions yielding values in [0, *n*–1]
  - These values act as indexes into the arrays

- For each input item:
  - For each hash:
    - Hash item to get index into corresponding array
    - Increment the value at that position by 1

Frequency estimation

Count–min sketch: creating and populating



PEARSON

Frequency estimation
Count-min sketch: querying

- For each hash function:
  - Hash query item to get index into corresponding array
  - Get the count at that position
- Return the *lowest* of these counts

This minimizes the effect of hash collisions.

(Collisions can only cause over-counting, not under-counting)

Frequency estimation

Count-min sketch: querying

“foo”

h_1 h_2 h_3

min(1, 1, 2) = 1

A_1	0	1	0	0	0	1	0	0	0	0
A_2	0	0	0	1	0	0	1	0	0	0
A_3	0	0	0	0	0	0	0	0	2	0

Caveat: You can’t iterate through the items, they’re not stored at all.

Frequency estimation
Count-min sketch

Cool properties

- Fast adding and querying in O(k) time
- As with Bloom filter: more hashes = lower error
- Mergeable by cellwise addition
- Better accuracy for higher-frequency items (“heavy hitters”)
- Can also be used to find quantiles

Similarity search

Which items are most similar to this one?

Similarity search
Naïve approach

Nearest-neighbour search

- For each stored item:
  - Compare to query item via appropriate distance metric*
  - Keep if closer than previous closest match

- Distance metric calculation can be expensive
  - Especially if items are many-dimensional records

- Can be slow **even if data small enough to fit in memory**

*Cosine distance, Hamming distance, Jaccard distance etc.

Similarity search
Locality-sensitive hashing

A probabilistic method for nearest-neighbour search

Real-life example:

Finding users with similar music tastes in an online radio service.

Similarity search
Locality-sensitive hashing

Intuitive explanation

Typical hash functions:

- Similar inputs yield very different outputs

Locality-sensitive hash functions:

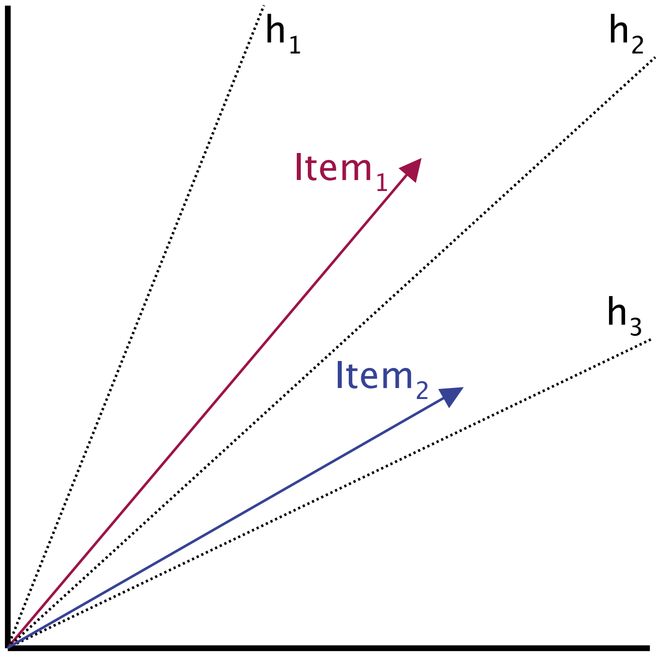
- Similar inputs yield similar or identical outputs

So: Hash each item, then just compare hashes.

- Can be used to pre-filter items *before* exact comparisons
- You can also index the hashes for quick lookup

Similarity search
Locality-sensitive hashing: random hyperplanes

Treat *n*-valued items as vectors in *n*-dimensional space.



In this example: n=2, k=3

Draw *k* random hyperplanes in that space.

For each hyperplane:

Is each vector above it (1) or below it (0)?

Hash(Item_1) = 011

Hash(Item_2) = 001

As the cosine distance decreases, the probability of a hash match increases.

Similarity search

Locality-sensitive hashing: random hyperplanes

Cool properties

- Hamming distance between hashes approximates cosine distance
- More hyperplanes (higher *k*) -> bigger hashes -> better estimates

- Can use to narrow search space before exact nearest-neighbours

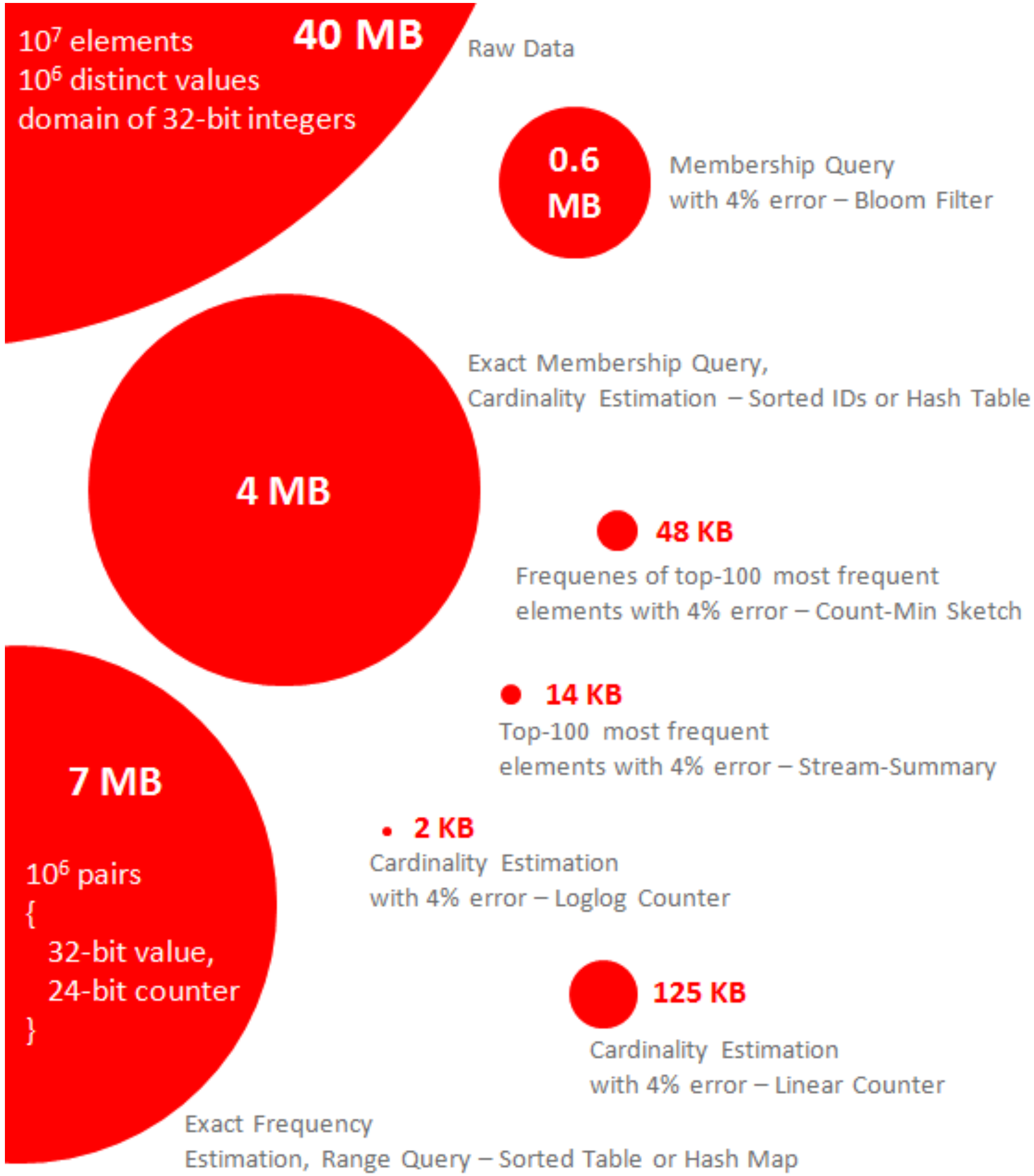
- Various ways to combine sketches for better separation

Similarity search
Locality-sensitive hashing

Other approaches

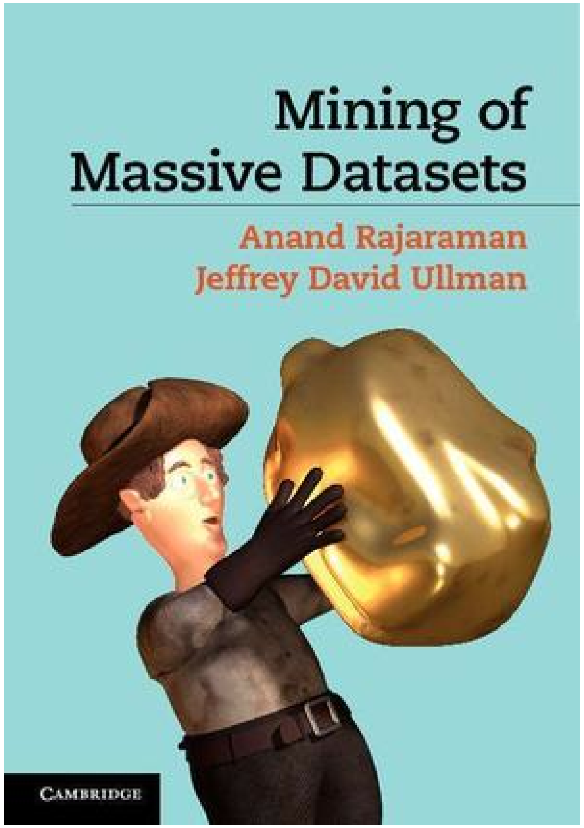
- Bit sampling: approximates Hamming distance
- MinHashing: approximates Jaccard distance
- Random projection: approximates Euclidean distance

Wrap-up



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

Resources



stream-lib:

https://github.com/clearspring/stream-lib

Java libraries for cardinality, set membership, frequency and top-N items (not covered here)

No canonical source of multiple LSH algorithms, but plenty of separate implementations

Wikipedia is pretty good on these topics too

Ebook available free from:

http://infolab.stanford.edu/~ullman/mmds.html