ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Stream Analytics

Yannis Kotidis

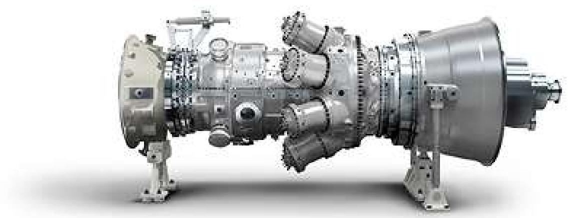
http://pages.cs.aueb.gr/~kotidis/

Stream Data Challenges

- Conventional (static) algorithms assume that data is available when we want it
- In a (pure) stream processing scenario, data arrives in streams and if not processed immediately or stored, then it is lost forever
- Main challenges: number of streams * velocity
  - Data arrives so rapidly that it is not feasible to store it all in memory or in a database to query it in real time
  - Even if a single stream is slow, there can be thousands of such steams in a large-scale application

Example: Gas Turbines Monitoring
[Optique FP7]

- 950 power generating turbines located across the globe
  - 100K sensors installed
  - Hundreds of TB worth of readings
- Detect in real-time undesirable patterns
  - Single-stream processing
  - Multi-stream processing
  - Live stream + archived stream correlation

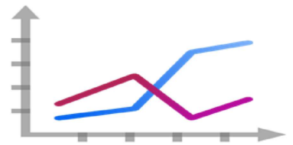


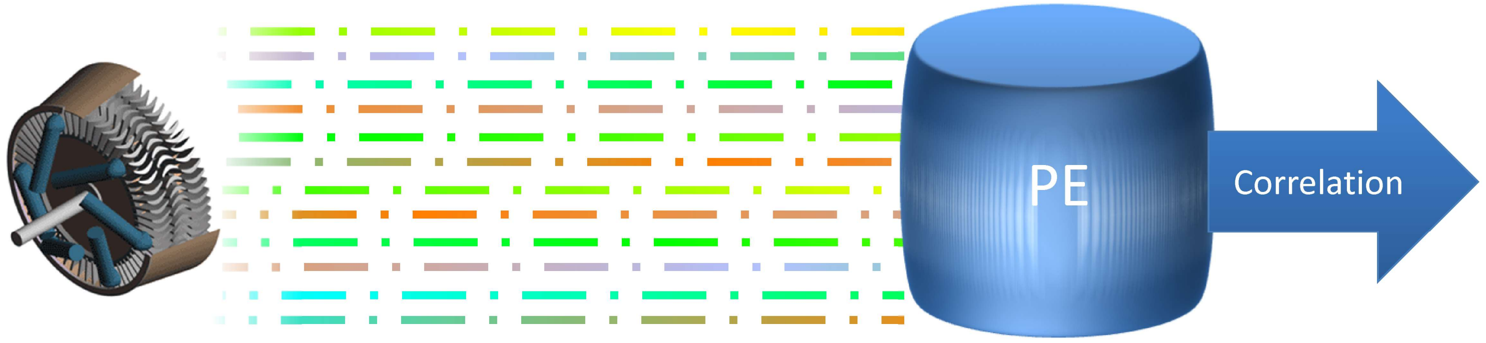
Optique

Turbine monitoring

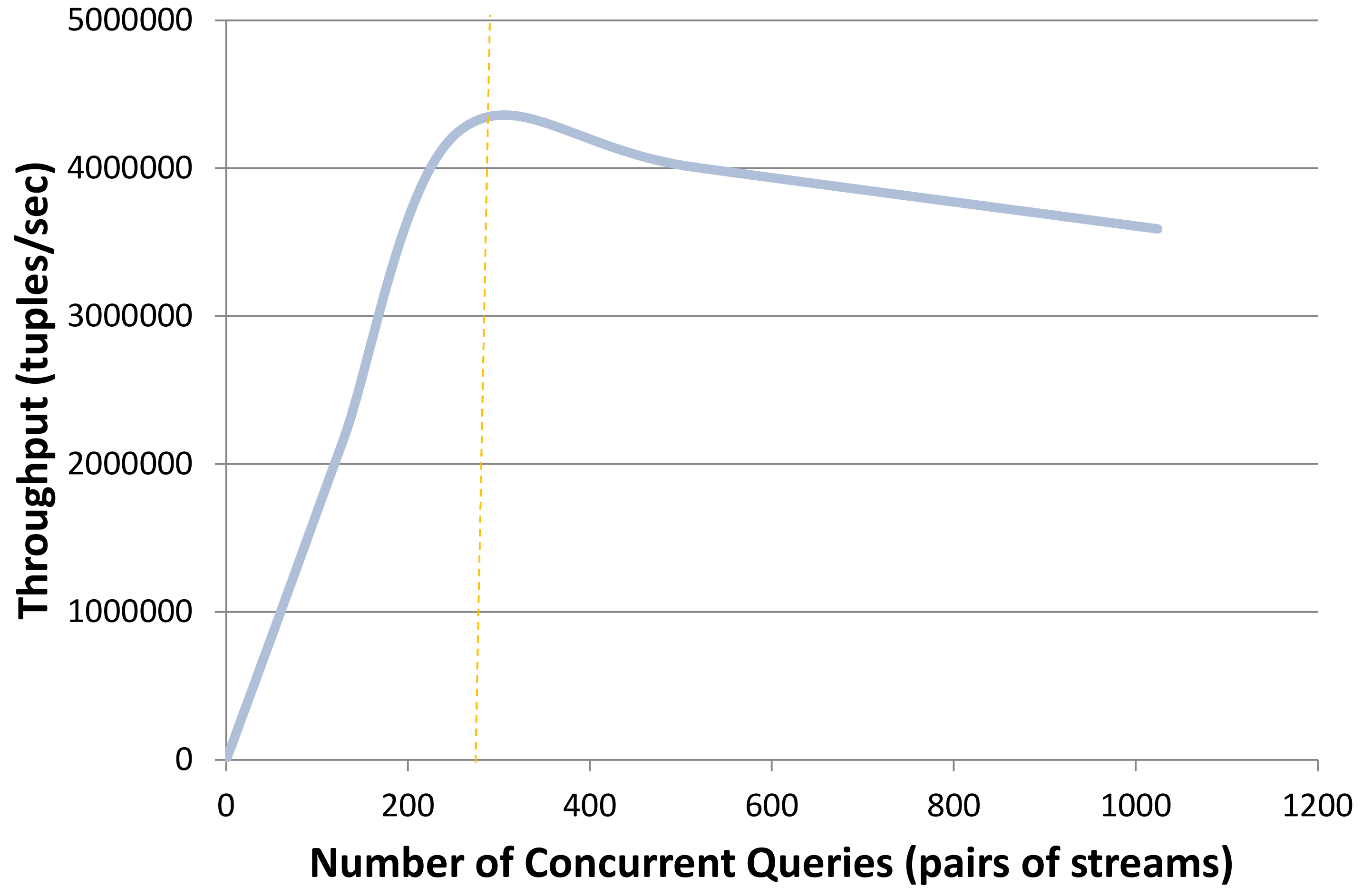
- Each Correlation query:
  - Intercepts two streams
  - Groups measurements over specified windows
  - Joins streams, computes Pearson coefficient:

$$\text{Pearson}(u_i, u_j) = cov(u_i, u_j) / (\sigma_{u_i} * \sigma_{u_j})$$





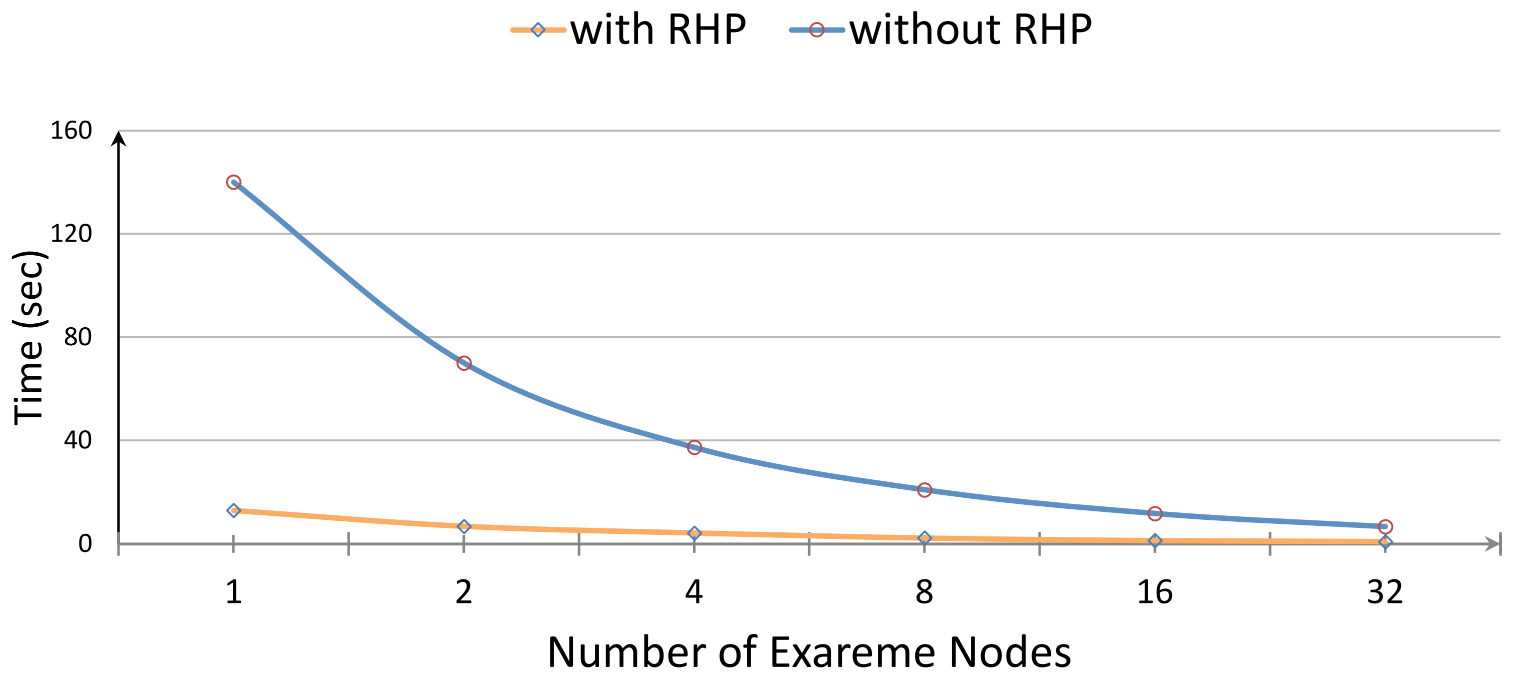
Throughput on a 256-core Exareme* cluster



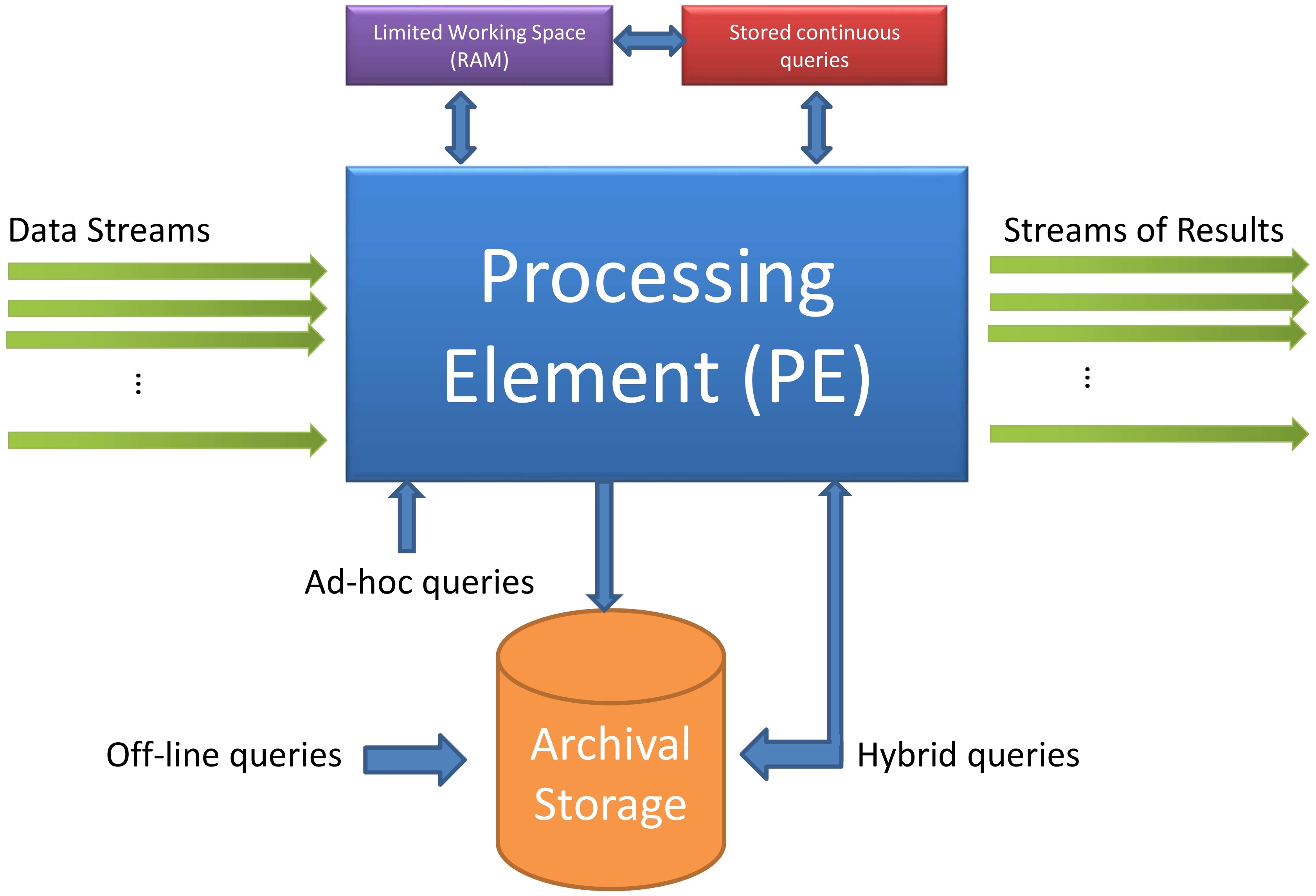
**http://madgik.github.io/exareme/*

Speed-up via LSH

- Corr. between current window and 100K archived ones [ISWC 2016, BigData 2016]

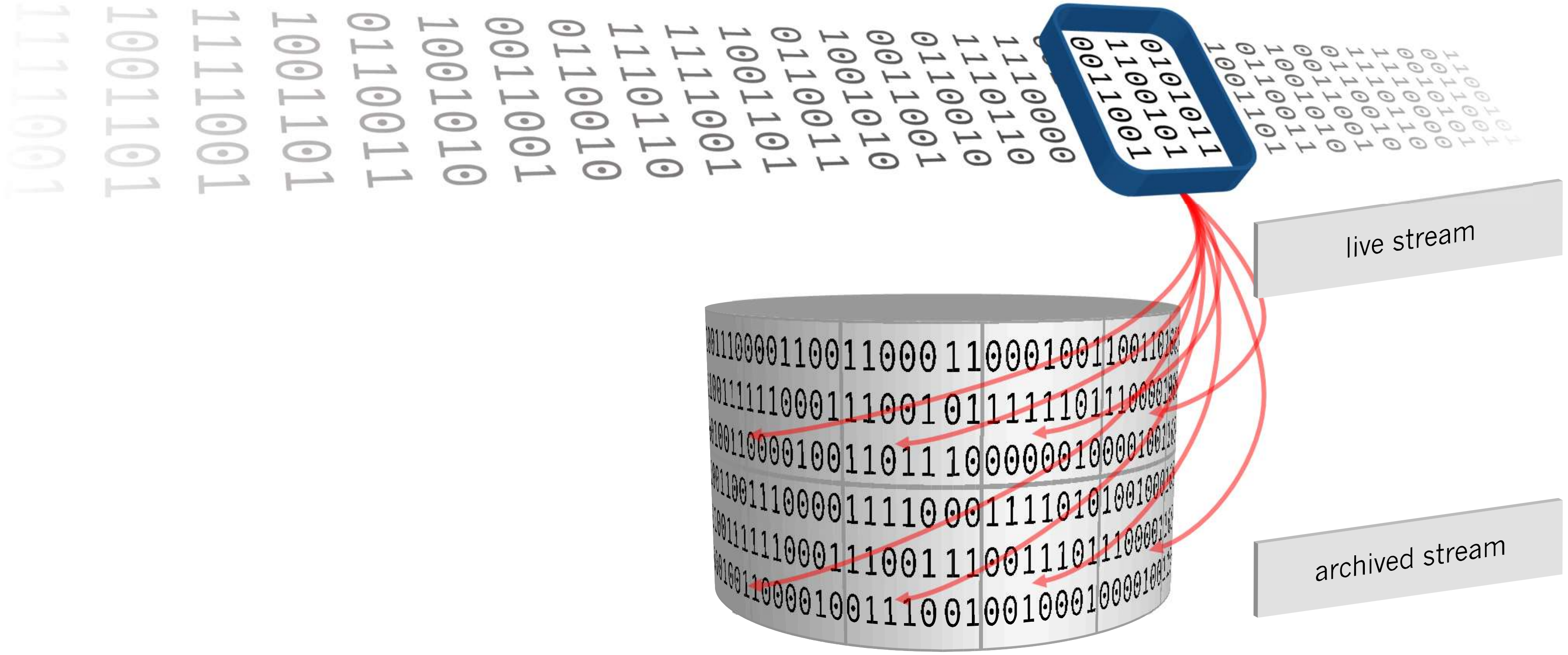


Data Stream Processing

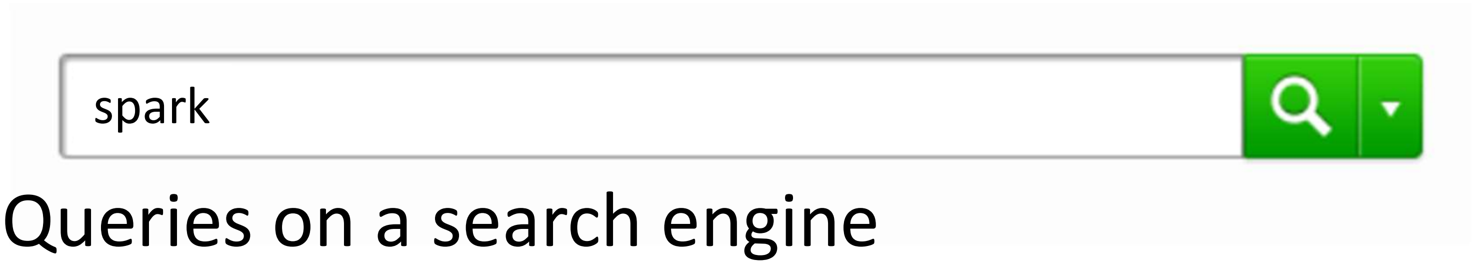


Static and stream data processing

- E.g. compute correlation between the *current state of a stream* and its *past states stored in archive storage*



Ad-hoc query example



- Queries on a search engine
  - Stream of tuples <user, term, timestamp>
- Simplification (for the shake of this running example): a user may ask the same query (term) once or twice
- Want to compute the fraction of duplicate queries issued by a typical user

Query Stream

(showing one user for simplicity)

- User makes 6 searches
    - 5 unique terms
- One duplicate term ("Real")
- fraction of duplicate queries = 1/5 =20%

User	Term	Timestamp
user1	Barca	t1
user1	Real	t2
user1	Liverpool	t3
user1	Porto	t4
user1	Real	t5
user1	Panathinaikos	t6

Sampling from a data stream

- Keep a 10% sample of the stream
  - E.g. draw a random integer x in range (0..9). Then keep tuple if x = 0

- For a typical user, we want to compute the fraction of duplicate queries from the sample

- Assume a user make s one-time searches and d duplicate searches
  - Correct answer is d/(s+d)

Using the sample

- Look at the sample to determine duplicates
  - Let s’ be the number of unique queries, for a user
  - Let d’ be the number of duplicates found, for a user
  - Report d’/(s’+d’)

- Is this correct?

Sample 50% of user searches (rows in green color)

- Real fraction of duplicate queries = 1/5 =20%
- Estimate = ?

User	Term	Timestamp
user1	Barca	t1
user1	Real	t2
user1	Liverpool	t3
user1	Porto	t4
user1	Real	t5
user1	Panathinaikos	t6

Sampling unique queries

- Let s be the number of unique searches a user makes
- These appear s/10 times in the sample

Sampling duplicate queries

- Let d be the number of duplicate searches a user makes

- A duplicate search appears twice in the sample with probability 1/10 * 1/10 = 1/100

Sampling duplicate queries

- A duplicate search appears once in the sample with probability $\underbrace{1/10 * 9/10}_{\text{Sample only 1st occurrence}} + \underbrace{9/10*1/10}_{\text{Sample only 2nd occurrence}}$

- A duplicate search does not appear in the sample with probability 9/10 * 9/10

In conclusion

- One-time queries in the sample
  - $s' = s/10 + 18d/100 = (10s+18d)/100$
- Duplicate queries in the sample
  - $d' = d/100$
- Our estimate is $d'/(s'+d') = d/(10s+18d)$
- Notice that this is different that $d/(s+d)$

Under-estimation

s	d	Fraction d/(s+d)	Estimate d/(10s+18d)
95	5	5%	0.5%
90	10	10%	0.9%
85	15	15%	1.3%
80	20	20%	1.7%
75	25	25%	2.1%
⋮	⋮	⋮	⋮
5	95	95%	5.4%

Obtaining a Representative Sample

- As shown a random sample from all users is not representative of the average behavior

- Alternative idea: select 10% of the users and keep all their queries
  - Select these users at random
  - Do not store searches from users not in the sample

User selection

- Incoming stream tuple <user, term, time>
- Let h(x) be a hash function returning values in the range (0..9)
- Keep tuple if h(user) = 0

Maintaining fixed sample size

- In the previous example we keep about 10% of the searches
- Recall that stream is (in theory) infinite
  - Thus, the sample keeps growing
  - Also recall that we do not have control over the input stream. System may exhibit bursts of heavy usage
- How to keep the sample size memory bound?

Hashing to the rescue

- Let h(x) return values in the range (0..B-1) for some very large value B
- Keep <user,term,time> in the sample if h(user) ≤ k, for some constant k≤B,
  - Store <h(user),user,term,time> in memory
  - Possibly index by h(user)
- If memory is full, reduce value of k
  - discard samples with h(user)>k

STREAM FILTERING

Applying filters on streams

- Often the selection criterion can be calculated from the stream tuple
  - Does the query term contain > 5 characters?
    - Easy to compute: length(term) > 5
- In other cases the selection criterion involves lookup for membership in a set
  - Problem becomes hard when this set is very large
  - Is the query term a “bad” word

Membership Test: Motivational Example

- Have 1 billion bad URLs you would like to block ($n=10^9$)
  - each URL is ~50 characters long
  - Need >50GB to keep all in main memory
- Would like to block a URL request in real time if it belongs to the black list

Membership test: Bloom Filters

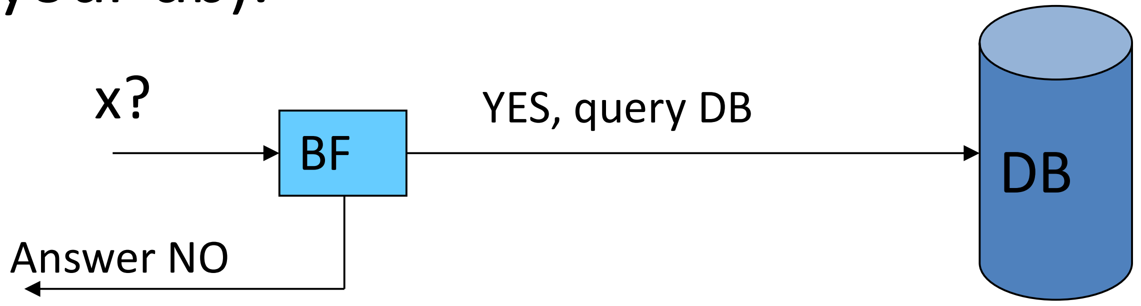
- Be able to quickly test where key value x is part of a set S
- Application: spam filtering
  - Have a set S of one billion valid email addresses (white list) for spam filtering
  - Assume 20 bytes per email address. S does not fit in memory
  - Want a memory resident data structure that will tell us whether an incoming email is spam or not

Spam Filtering

- Bloom filter will check whether an incoming email is from a valid email address in the white list
- If the answer is no then the email is guaranteed to be spam and is thus rejected
- If the answer is yes, the email is with high probability in the list
  - Cases where the filter says "yes" while the true answer is "no" are termed false positives

More applications of Bloom Filters

- Web-crawler: avoid visiting same page twice
- High-traffic on-line music store with millions of titles
  - only fetch song information when you know the song exists in your collection (minimize #queries to your db).



Another Application: Spell Checker

- Have a long list S of all English words
- Want to spell-check a document D
- Semi-naïve implementation:
  - keep list sorted
  - For every word in document D do a binary-search over S
  - Run-time: $O(|D| * \log_2 |S|)$

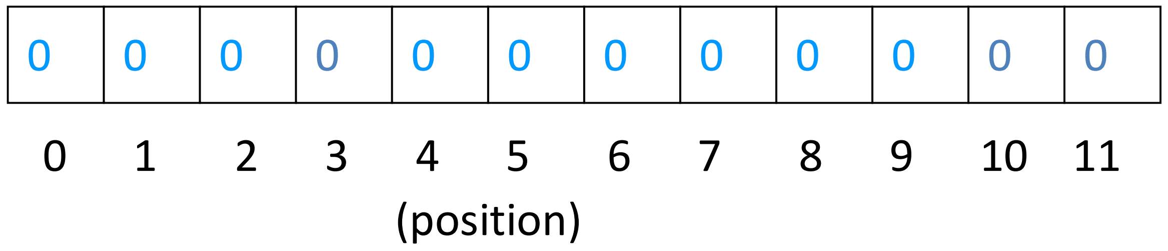
Problem Statement

- Have a very large set S
- Membership test: is x part of S?
- Want a data structure that
  - Is small (can fit in memory, when S cannot)
  - Requires a (small) constant time for look-ups
  - Guarantees no false negatives
  - Introduces a limited number of false positives
    - For those cases you can optionally look up x in S in a second step
      - This works only if answering “yes” happens infrequently

Bloom Filter

- Use bitmap of length m and k hash functions
  - Each $h_i(x)$ maps x to [0..m-1]
- Initially, all bits are zero

Initially Empty Bloom Filter (m=12)



Training (using 3 hash functions)

Insert “apples”

$$\left.\begin{array}{l} -\ h_1(\text{"apples"}) = 3 \\ -\ h_2(\text{"apples"}) = 11 \\ -\ h_3(\text{"apples"}) = 10 \end{array}\right\}\ \text{set corresponding bits}$$

BITMAP (after insertion of “apples”)

0	0	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Train with more data

BITMAP (apples)

0	0	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Now insert “oranges”

- h_1("oranges ") = 10
- h_2("oranges ") = 1
- h_3("oranges ") = 5

BITMAP (apples+oranges)

collision

0	1	0	1	0	1	0	0	0	0	**1**	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Querying: Membership test

- All bits indicated by $h_i(x)$ must be set
  - – h_1(“bananas") = 10
  - – h_2(“bananas") = 5
  - – h_3(“bananas") = 7

Is “bananas” part of my data?

BITMAP

0	1	0	1	0	1	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

What can we guarantee?

- No false negatives (why?)
- Small probability of false positives

$$(1-(1-1/m)^{kn})^k$$

- False positive when all k bits are set for an item we have not seen
  - A bit is set with probability $1/m$ assuming ideal hash function
  - $(1-1/m)^k$ = probability a bit is not set after one insertion
  - $(1-1/m)^{kn}$ = probability that a bit is not set after n insertions

Running Example

- Have 1 billion bad URLs you would like to block ($n=10^9$)
  - each URL is ~50 characters long
  - Need >50GB to keep all in main memory
- Use a bitmap of 8 billion entries ($m=8*10^9$)
  - hash table takes 1GB of memory
- For k=6, probability of false positives = $(1-(1-1/(8*10^9))^{6*10^9})^6$ =2.1%

Dependency on k

more hash functions

k	False positives Probability
1	12%
2	5%
3	3%
4	2.4%
5	2.2%
6	2.1%
7	2.3%
8	2.5%
9	3%

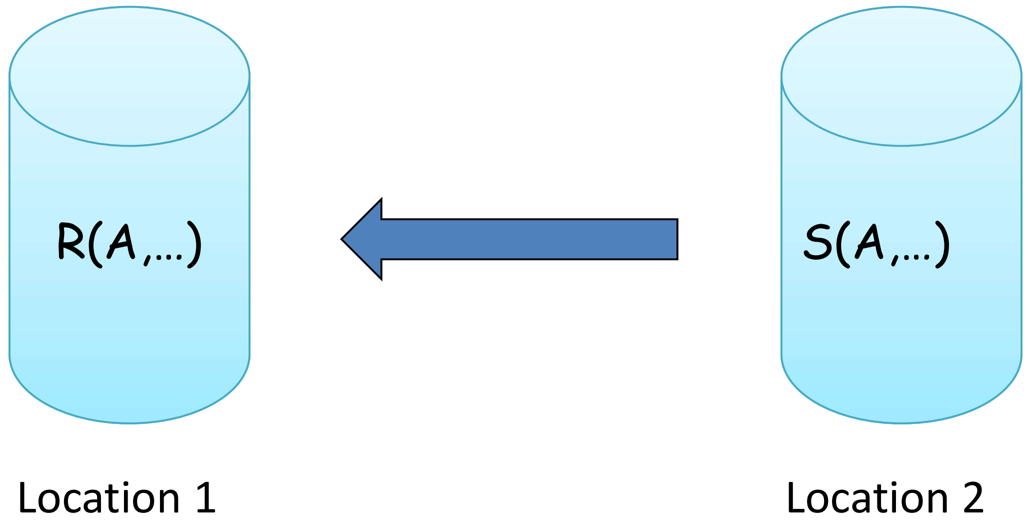
Bloom Filters in Distributed Databases

- Suppose we want to join two tables R(A,...) and S(A,...) that reside on two distant locations
  - Join result can be computed at either location

R(A,...)
S(A,...)
Location 1
Location 2

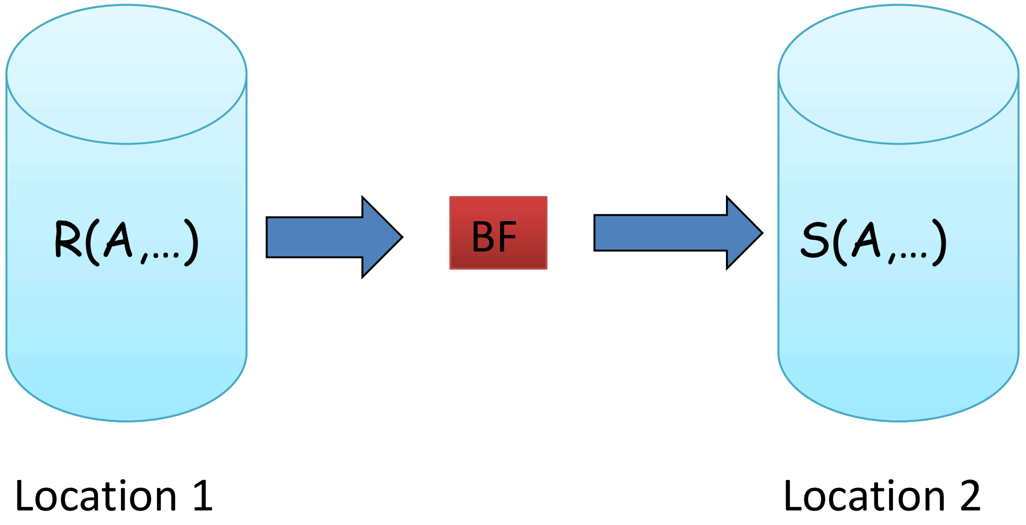
Idea 1: Ship smallest relation to the other side

- Suppose S is smaller
- Communication Cost = size(S)
- Can we do better?



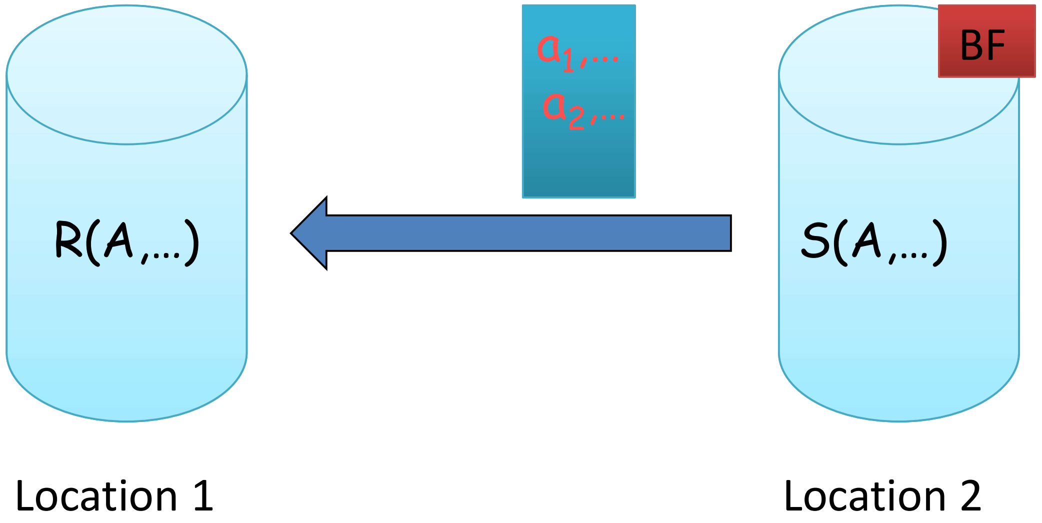
Idea 2: Step 1

- Build BF on the values of R.A
- Ship BF to location 2
  - Recall that size(BF) << size(R)



Idea 2: Step 2

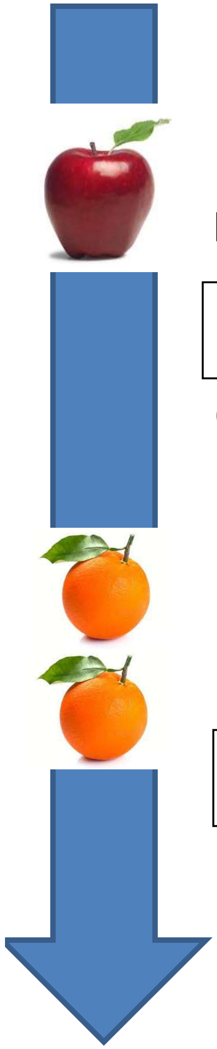
- For each S.A value a test using BF whether a exists in R.A column
- Ship to Location 1 those records that pass the BF test
  - If a value S.A does not pass the BF test, then S.A does not join for sure (why?)
  - But we may ship a few records that will not join (false positives)
  - Final result is always correct!



Extensions

- Support insertions/deletions/multi-set semantics
- Have a grocery store and the following list of transactions
  - Buy apple from supplier
  - Buy apple from supplier
  - Sell apple to buyer
  - Buy apple from supplier
  - Sell apple to buyer
- Do I have apples left in my store?

Intuition: maintain counters within buckets



BITMAP (after insertion of 1 apple)

0	0	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

BITMAP (after insertion of 2 oranges)

0	2	0	1	0	2	0	0	0	0	3	1
0	1	2	3	4	5	6	7	8	9	10	11

(position)

Neat Implementation: Count-Min sketch