CS 591 K1: Data Stream Processing and Analytics

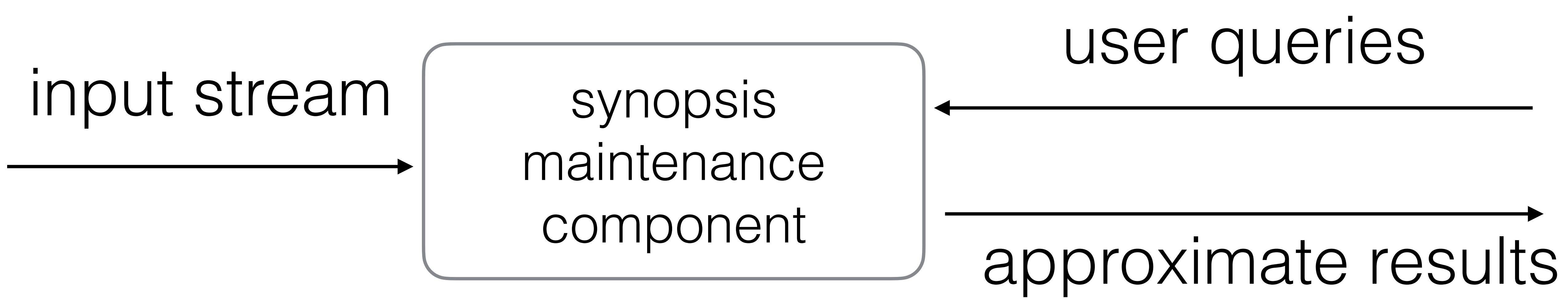
Spring 2020

4/21: Sampling and filtering streams

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Synopses for massive data streams

Synopsis: a lossy, compact summary of the input stream



- Maintaining synopses is often the only means of providing interactive response times when exploring massive datasets or high speed data streams.
- Queries are executed against the synopsis rather than the entire dataset.

A simple and efficient synopsis

Suppose that our data consists of a large numeric time series.

What summary would let us compute the **statistical variance** of this series?

$$var = \frac{\sum (x_i - \mu)^2}{N}$$

A simple and efficient synopsis

Suppose that our data consists of a large numeric time series.

What summary would let us compute the **statistical variance** of this series?

- the sum of all the values
- the sum of the squares of the values
- the number of observations

$$var = \frac{\sum (x_i - \mu)^2}{N}$$

A simple and efficient synopsis

Suppose that our data consists of a large numeric time series.

What summary would let us compute the **statistical variance** of this series?

- the sum of all the values
- the sum of the squares of the values
- the number of observations

$$var = \frac{\sum (x_i - \mu)^2}{N}$$

Then

- μ = sum / count
- var = (sum of squares / count) - μ^2

A simple and efficient synopsis

Suppose that our data consists of a large numeric time series.

What summary would let us compute the **statistical variance** of this series?

- the sum of all the values
- the sum of the squares of the values
- the number of observations

$$var = \frac{\sum (x_i - \mu)^2}{N}$$

Then

- μ = sum / count
- var = (sum of squares / count) - μ^2

We can compute the three summary values in a single pass through the data.

A simple and efficient synopsis

Suppose that our data consists of a large numeric time series.

What summary would let us compute the **statistical variance** of this series?

- the sum of all the values
- the sum of the squares of the values
- the number of observations

$$var = \frac{\sum (x_i - \mu)^2}{N}$$

Then

- μ = sum / count
- var = (sum of squares / count) - μ^2

We can compute the three summary values in a single pass through the data.

Can this synopsis be used to answer general queries?

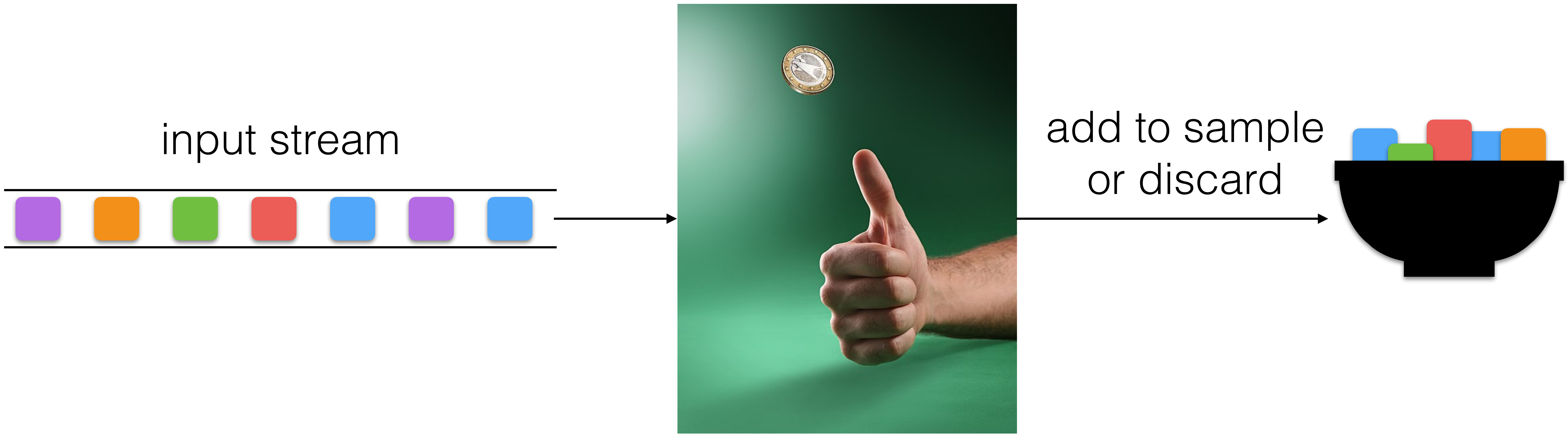
Synopses provide accurate estimations

- For many queries, an exact answer would require storing and analyzing the entire dataset
- Instead, we can relax this requirement and provide a good enough approximation
- A small synopsis can provide very accurate approximations using very little space:
    - It might suffice to know that the true answer is roughly $5 million without knowing that the exact answer is $5,001,482.76.

Sampling streams

Samples: the most fundamental synopses

A *sample* is a set of data elements selected via some random process



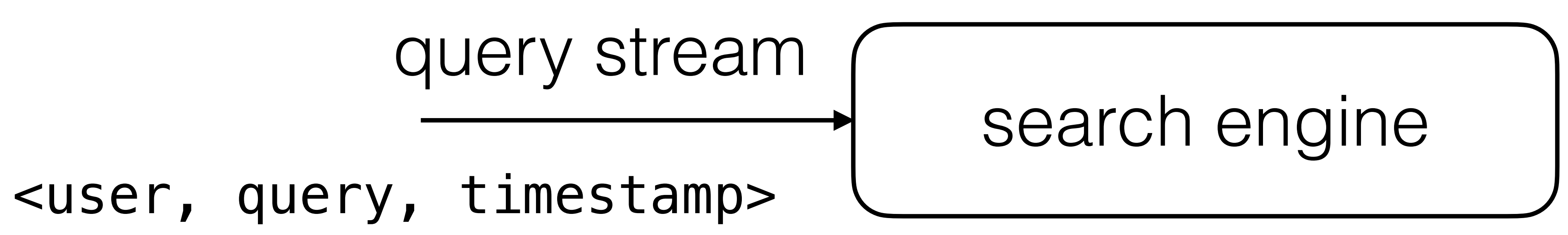
How can we select a **representative** sample of an unbounded stream?

- we want to ask queries and get statistically meaningful answers about the entire stream
- we don't necessarily know the queries in advance
- we can store a **fixed proportion** of the stream, e.g. 1/10th

How can we select a **representative** sample of an unbounded stream?

- we want to ask queries and get statistically meaningful answers about the entire stream
- we don’t necessarily know the queries in advance
- we can store a **fixed proportion** of the stream, e.g. 1/10th

Example use-case: **Web search user behavior study**



Q: How many queries did users repeat last month?

Solution #1: uniform sampling

Q: How many queries did users repeat last month?

- Since we can store 1/10th of the stream, we select a stream element *i* with probability 10%.
- We can use a random generator that produces an integer r_i between 0 and 9. We then select an input element *i* if $r_i=0$.

Solution #1: uniform sampling

Q: How many queries did users repeat last month?

- Since we can store 1/10th of the stream, we select a stream element *i* with probability 10%.
- We can use a random generator that produces an integer r_i between 0 and 9. We then select an input element *i* if $r_i = 0$.

Will this approach provide the right answer?

Ted has issued *n* queries in the last month:

- *s* of those are unique
- *d* of those are duplicates
- no query was issued more than twice

How many of Ted's queries will be in the 1/10th sample, *S*?

Each of the *s* unique queries has a probability $P_s = 1/10$ to be selected:

- an expected number of *s/10* of those queries will be in *S*.

How many of Ted's queries will be in the 1/10th sample, *S*?

What about the duplicates, *d* ?

How many of Ted's queries will be in the 1/10th sample, *S*?

What about the duplicates, *d* ?

Probability that both occurrences are in *S*:

P_a = 1/10 * 1/10 = 1/100 => ***d/100*** will appear in *S* twice.

How many of Ted's queries will be in the 1/10th sample, *S*?

What about the duplicates, *d* ?

Probability that both occurrences are in *S*:

P_a = 1/10 * 1/10 = 1/100 => ***d/100*** will appear in *S* twice.

Probability that only one occurrence is in *S*:

P_b = 1/10 * 9/10 + 9/10 * 1/10 = 18/100 =>

18*d/100 will appear in *S* once.

How many of Ted's queries will be in the 1/10th sample, *S*?

What about the duplicates, *d* ?

Probability that both occurrences are in *S*:

P_a = 1/10 * 1/10 = 1/100 => ***d/100*** will appear in *S* twice.

one is selected

the other is not

Probability that only one occurrence is in *S*:

P_b = 1/10 * 9/10 + 9/10 * 1/10 = 18/100 =>

18*d/100 will appear in *S* once.

Q: How many queries did Ted repeat last month?

$$\frac{\frac{d}{100}}{\frac{s}{10} + \frac{18d}{100} + \frac{d}{100}}$$

Q: How many queries did Ted repeat last month?

queries appearing in S twice

$$\frac{\frac{d}{100}}{\frac{s}{10} + \frac{18d}{100} + \frac{d}{100}}$$

Q: How many queries did Ted repeat last month?

$$\frac{\frac{d}{100}}{\frac{s}{10} + \frac{18d}{100} + \frac{d}{100}}$$

queries appearing in S twice

all of Ted's queries in S

Q: How many queries did Ted repeat last month?

queries appearing in S twice

$$\frac{\frac{d}{100}}{\frac{s}{10} + \frac{18d}{100} + \frac{d}{100}} = \frac{d}{10s + 19d}$$

all of Ted's queries in S

Q: How many queries did Ted repeat last month?

queries appearing in S twice

$$\frac{\frac{d}{100}}{\frac{s}{10}+\frac{18d}{100}+\frac{d}{100}} = \frac{d}{10s+19d}$$

all of Ted's queries in S

…instead of $\frac{d}{s+d}$

Solution #2: sampling users

Sample 1/10th of the *users* instead

Solution #2: sampling users

Sample 1/10th of the *users* instead

- Maintain a list of all users seen so far and a *flag* indicating whether they belong to the sample or not
- When a query arrives:
  - if the user is sampled: add the query to S
  - if we haven't seen the user before: generate a random integer r_u between 0 and 9 and add the user to the sample if $r_u = 0$.

Solution #2: sampling users

Sample 1/10th of the *users* instead

- Maintain a list of all users seen so far and a *flag* indicating whether they belong to the sample or not
- When a query arrives:
  - if the user is sampled: add the query to S
  - if we haven't seen the user before: generate a random integer r_u between 0 and 9 and add the user to the sample if $r_u = 0$.

Do we need to keep all users in memory?

We can use a **hash function** *h* to hash the user name (or IP) and select queries only when $h(user) = 0$.

In general:

We can obtain a sample of any a/b fraction of users by hashing usernames to b buckets and selecting the query if $h(user) < a$.

For example, to get a 30% sample:

- use 10 buckets, b_0, b_1, …, b_9.
- select the query if the user hash value is in b_0, b_1, or b_2.

We can use a **hash function** *h* to hash the user name (or IP) and select queries only when $h(user) = 0$.

<u>In general</u>:

We can obtain a sample of any a/b fraction of users by hashing usernames to b buckets and selecting the query if $h(user) < a$.

For example, to get a 30% sample:

- use 10 buckets, b_0, b_1, …, b_9.
- select the query if the user hash value is in b_0, b_1, or b_2.

How can we limit the sample size from growing indefinitely?

Instead of a fixed proportion, assume we can only store a sample *S* of **fixed size**, e.g. *s* elements.

Instead of a fixed proportion, assume we can only store a sample *S* of **fixed size**, e.g. *s* elements.

How can we continuously maintain a representative fixed-size sample of the stream so far?

Instead of a fixed proportion, assume we can only store a sample *S* of **fixed size**, e.g. *s* elements.

How can we continuously maintain a representative fixed-size sample of the stream so far?

At all times, we want the following property to hold:

an element is in S with probability s/n, where n is the total number of stream elements seen so far.

Instead of a fixed proportion, assume we can only store a sample *S* of **fixed size**, e.g. *s* elements.

How can we continuously maintain a representative fixed-size sample of the stream so far?

At all times, we want the following property to hold:

an element is in S with probability s/n, where n is the total number of stream elements seen so far.

As if we could keep all *n* elements and at any time pick *s* of those with equal probability.

Reservoir sampling

- Add the first s elements to S.
- When the n_{th} element arrives, e_n, $n > s$, keep it with probability $p=s/n$.
- If e_n is selected, then pick an existing element in S at random and replace it with e_n.

Claim: at time t_n, the probability that an element appears in S is s/n.

Claim: at time t_n, the probability that an element appears in *S* is *s/n*.

We'll use induction to prove this, i.e. we need to prove that the claim is true for *n+1*:

at time t_{n+1}, elements are in *S* with probability *s/(n+1)*.

Claim: at time t_n, the probability that an element appears in S is s/n.

We'll use induction to prove this, i.e. we need to prove that the claim is true for $n+1$:

at time t_{n+1}, elements are in S with probability $s/(n+1)$.

Base case

At time t_s, S has exactly s elements and each one appears in S with probability $s/s = 1$ => **true**.

Inductive step

At time t_{n+1}, we need to compute the probability that an element x in S remains:

Inductive step

At time t_{n+1}, we need to compute the probability that an element x in S remains:

Probability that element $n+1$ is not selected

Inductive step

At time t_{n+1}, we need to compute the probability that an element x in S remains:

Probability that element $n+1$ is not selected

OR

Inductive step

At time t_{n+1}, we need to compute the probability that an element x in S remains:

Probability that element $n+1$ is not selected

OR

Probability that $n+1$ is selected but it doesn't replace x

Inductive step

At time t_{n+1}, we need to compute the probability that an element x in S remains:

Probability that element $n+1$ is not selected

OR

Probability that $n+1$ is selected but it doesn't replace x

$$P_1 = 1 - \frac{s}{n+1}$$

Inductive step

At time t_{n+1}, we need to compute the probability that an element x in S remains:

Probability that element $n+1$ is not selected

OR

Probability that $n+1$ is selected but it doesn't replace x

$$P_1 = 1 - \frac{s}{n+1}$$

$$P_2 = \frac{s}{n+1} * (1 - \frac{1}{s}) = \frac{s-1}{n+1}$$

Inductive step

At time t_{n+1}, we need to compute the probability that an element x in S remains:

Probability that element $n+1$ is not selected

$$P_1 = 1 - \frac{s}{n+1}$$

OR

Probability that $n+1$ is selected but it doesn't replace x

$$P_2 = \frac{s}{n+1} * \left(1 - \frac{1}{s}\right) = \frac{s-1}{n+1}$$

selected

Inductive step

At time t_{n+1}, we need to compute the probability that an element *x* in *S* remains:

Probability that element *n+1* is not selected

OR

Probability that *n+1* is selected but it doesn't replace *x*

$$P_1 = 1 - \frac{s}{n+1}$$

$$P_2 = \frac{s}{n+1} * \left(1 - \frac{1}{s}\right) = \frac{s-1}{n+1}$$

selected

chosen to replace

Inductive step

At time t_{n+1}, we need to compute the probability that an element x in S remains:

Probability that element $n+1$ is not selected

OR

Probability that $n+1$ is selected but it doesn't replace x

$$P_1 = 1 - \frac{s}{n+1}$$

$$P_2 = \frac{s}{n+1} * (1 - \frac{1}{s}) = \frac{s-1}{n+1}$$

selected: $\frac{s}{n+1}$

chosen to replace: $\frac{1}{s}$

$$P_1 + P_2 = \frac{n}{n+1}$$

So, at time t_{n+1}, the probability that an element is in S is equal to:

So, at time t_{n+1}, the probability that an element is in S is equal to:

$$\frac{s}{n} * \frac{n}{n+1} = \frac{s}{n+1}$$

So, at time t_{n+1}, the probability that an element is in S is equal to:

it was in S at t_n → $$\frac{s}{n} * \frac{n}{n+1} = \frac{s}{n+1}$$

So, at time t_{n+1}, the probability that an element is in S is equal to:

$$\frac{s}{n} * \frac{n}{n+1} = \frac{s}{n+1}$$

it was in S at t_n

it is kept in S at t_{n+1}

Advantages of sampling

- Simple to understand and implement.
- Almost 100 years of prior research in sampling we can apply.
- The sample can often be constructed after a query has been issued and it can be adapted according to query needs:
  - if a small sample does not provide enough accuracy for a specific query, then more tuples can be sampled to provide for more accuracy, in an online fashion.
- It is a general-purpose synopsis and can be used to answer a wide variety of arbitrary queries.

Drawbacks of sampling

- It might be unsuitable for highly selective queries:
  - queries that depend only upon a few tuples from the dataset
- Providing an estimate via a sample can be much more expensive than estimation via other methods:
  - Evaluating a query over a 5% sample of a dataset may take 5% of the time that it takes to evaluate the query over the entire dataset. A 20× speedup may be significant, but other, more compact synopses such as histograms can provide much faster estimates.
- Sampling is generally sensitive to skew and outliers.
- It is difficult to find a good estimator for some queries:
  - How can we scale the answer for NOT IN, DISTINCT, anti-joins, outer-joins

Filtering streams

The membership problem

What data structure would you use to:

- Filter out all emails that are sent from a suspected spam address?
- Filter out all URLs that contain malware?
- Filter out all compromised passwords?
- Remove duplicate tuples on recovery when using upstream backup?

The membership problem

What data structure would you use to:

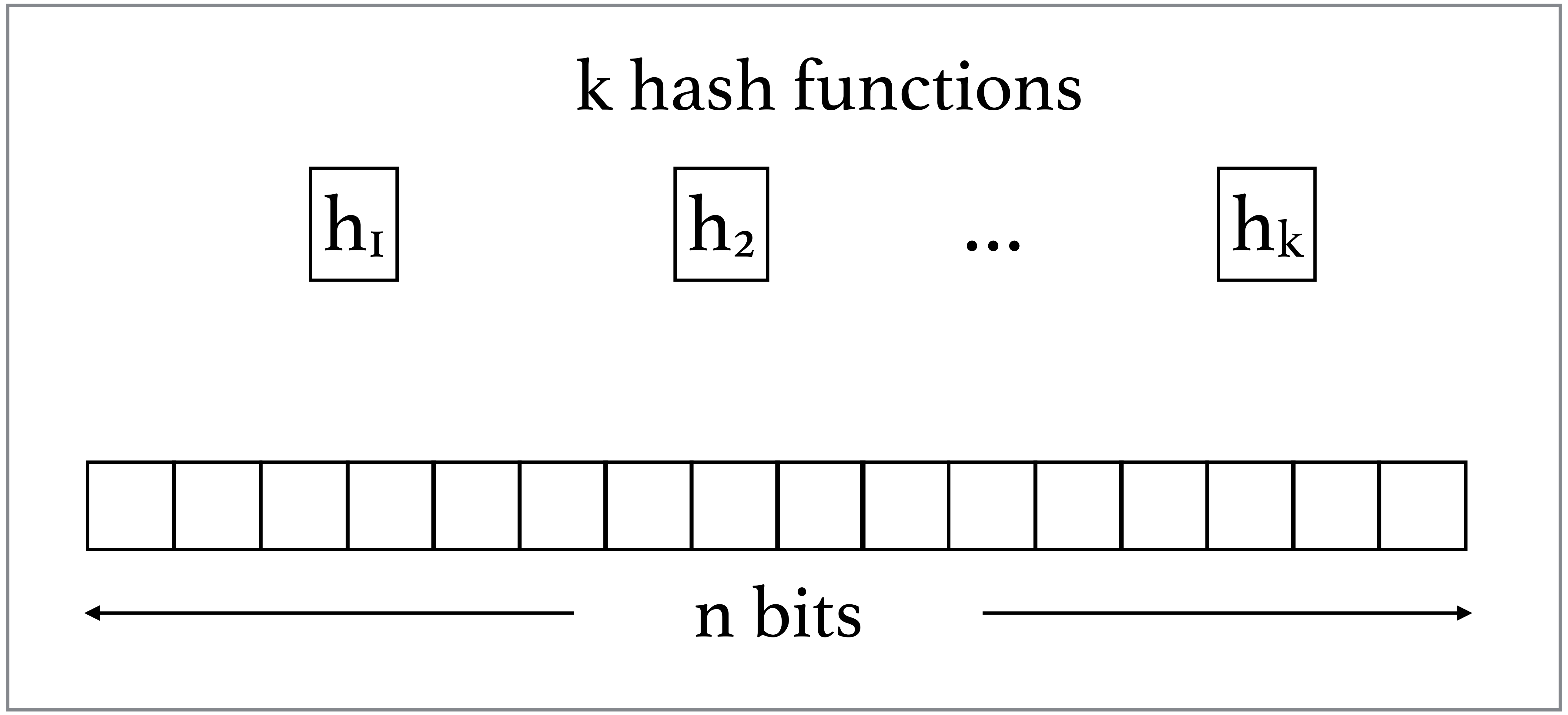
- Filter out all emails that are sent from a suspected spam address?
- Filter out all URLs that contain malware?
- Filter out all compromised passwords?
- Remove duplicate tuples on recovery when using upstream backup?

A hash table requires $O(\log n)$ bits per element which might still be infeasible in practice…

The Bloom filter

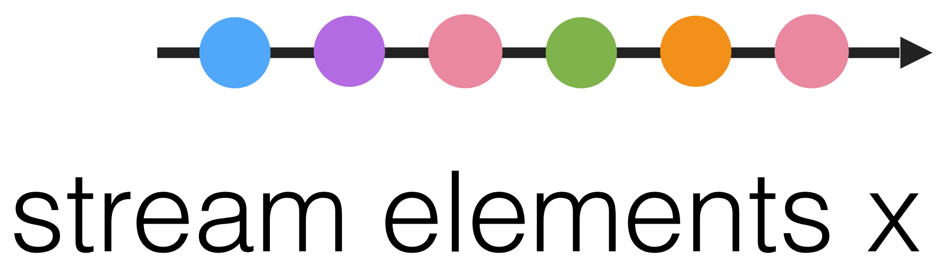
- Introduced by Burton Bloom in 1970.
- A probabilistic data structure for representing a (possibly growing) dataset of elements that supports:
  - adding an element to the set
  - checking if an element belongs to the set
- False positives are possible: *an element is not a member of the set but the filter says it is.*
- No false negatives: *if the filter says an element is not in the set, then it definitely isn't.*

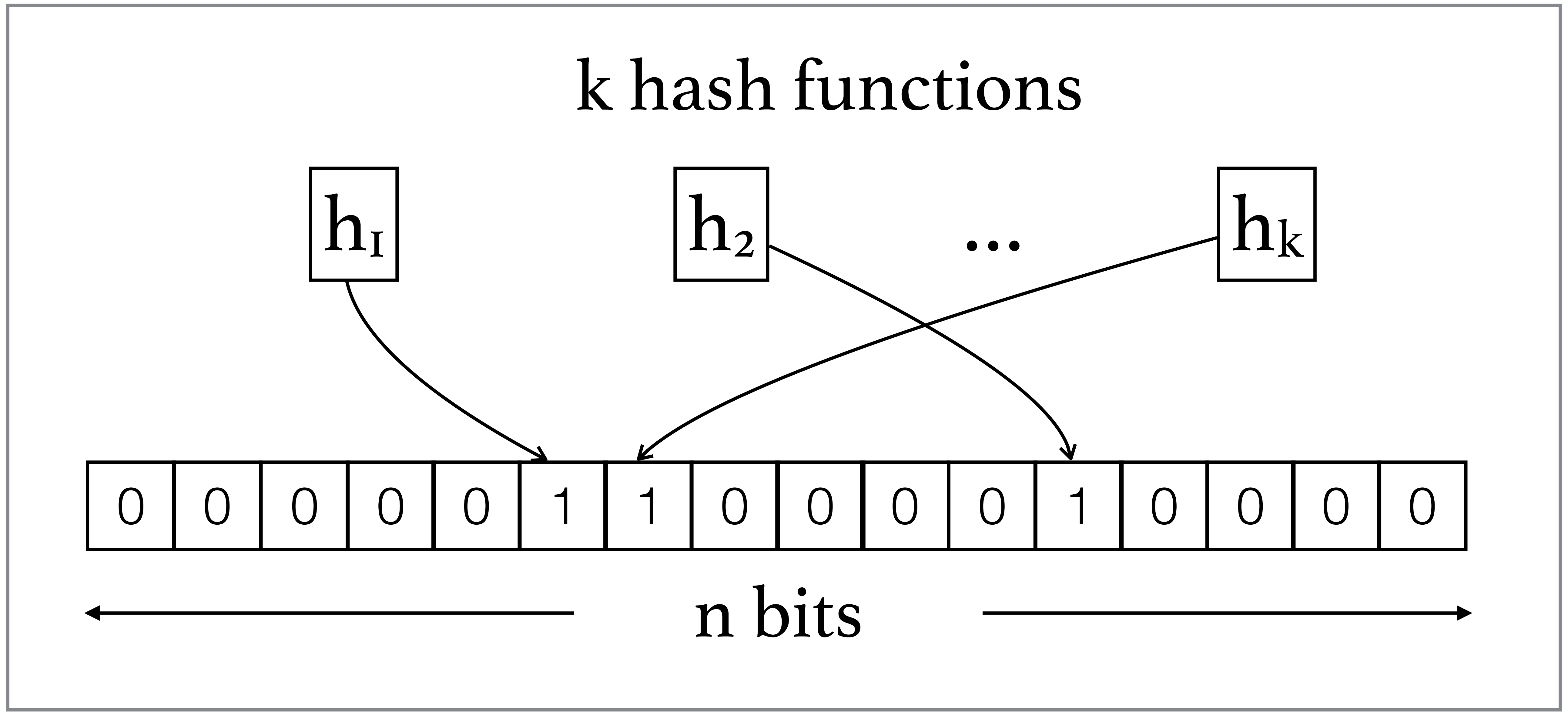
The Bloom filter



- A bit array of size n, where n is generally higher than the expected number of elements in the input
- k independent and uniformly distributed hash functions, where $k << n$

Adding an element to the filter

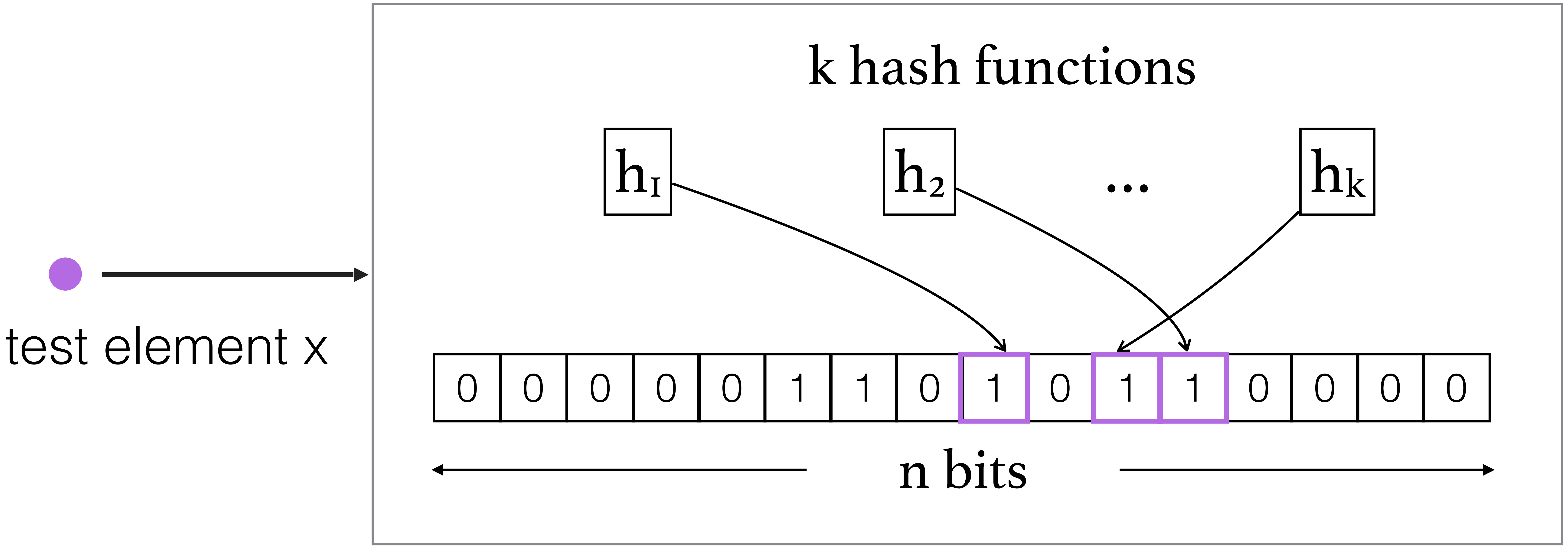




The empty filter is initialized to all 0s

```
for i=1 to k do
  j = hi(x)
  set jth bit in the filter
```

Testing if an element is in the filter



If all bits are set, the element may exist in the set.

If at least one element is 0, the element is definitely not a member.

```
for i=1 to k do
  j = hi(x)
  if the jth bit is not set then
     return FALSE
return TRUE
```

Probability of a false positive

- The probability of false positives depends on the choice of $\mathbf{k}$ and $\mathbf{n}$:

$$P_{fp} \approx (1 - e^{\frac{km}{n}})^k$$ *

- Let $\mathbf{m}$ be the number of expected elements:
  - If the allocated bits per element, $\mathbf{n/m}$, is too small, the filter will fill up too quickly
  - All lookups will yield a false positive

- For a given $\mathbf{n/m}$, the false positive probability can be tuned by choosing the number of hash functions:

$$k = \frac{n}{m} ln2$$

*: see slide 31

Parameter tuning example

Assume we expect around 1 billion elements and we have a fixed memory budget of 512MB

- How many hash functions to use?
- What would be the false positive rate?

Parameter tuning example

Assume we expect around 1 billion elements and we have a fixed memory budget of 512MB

- How many hash functions to use? $k \approx 3$
- What would be the false positive rate? $P_{fp} \approx 0.14$

Parameter tuning example

Assume we expect around 1 billion elements and we have a fixed memory budget of 512MB

- How many hash functions to use? $k \approx 3$
- What would be the false positive rate? $P_{fp} \approx 0.14$

What if we had 1GB of memory instead?

Parameter tuning example

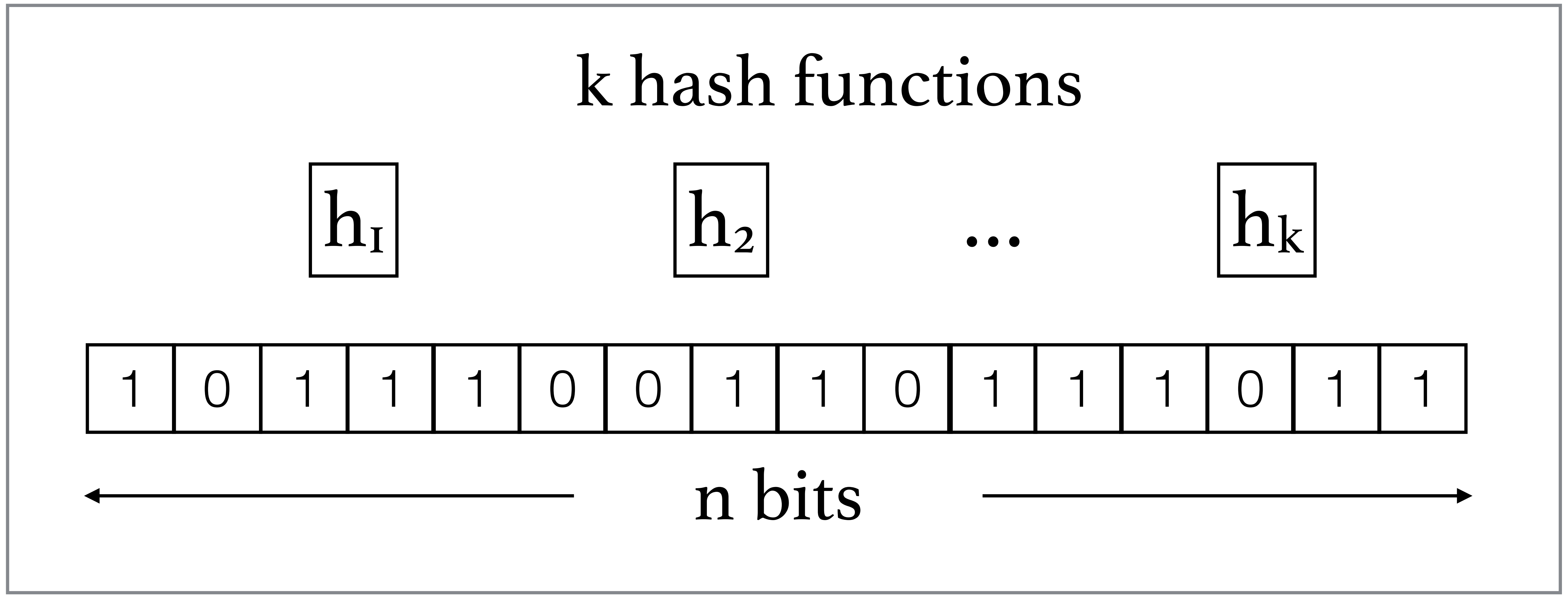
Assume we expect around 1 billion elements and we have a fixed memory budget of 512MB

- How many hash functions to use? $k \approx 3$
- What would be the false positive rate? $P_{fp} \approx 0.14$

What if we had 1GB of memory instead? $k \approx 5$ $P_{fp} \approx 0.02$

Optimal number of hash functions

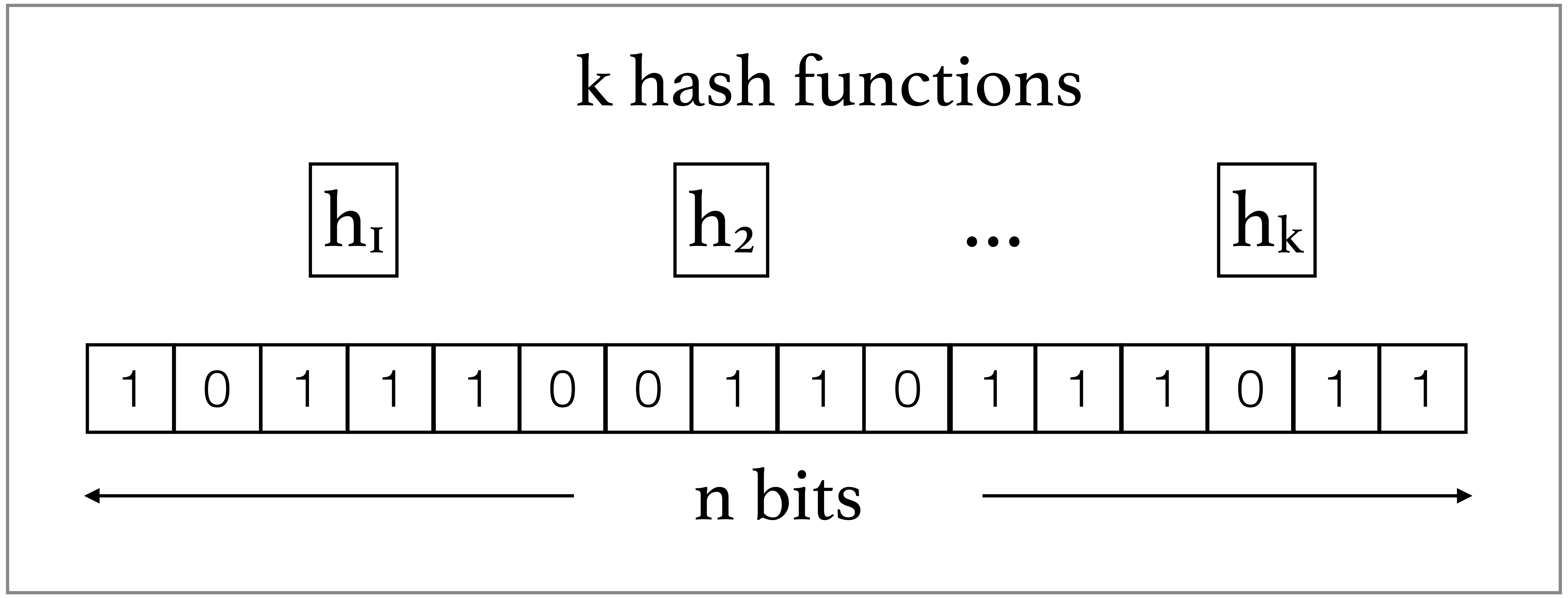
Given an expected number of elements and a fixed memory budget, how many hash functions do we need in order to minimize P_{fp}?



Optimal number of hash functions

Given an expected number of elements and a fixed memory budget, how many hash functions do we need in order to minimize P_{fp}?

After m elements have been inserted to the filter, what is the probability P_0 that a bit is still 0?

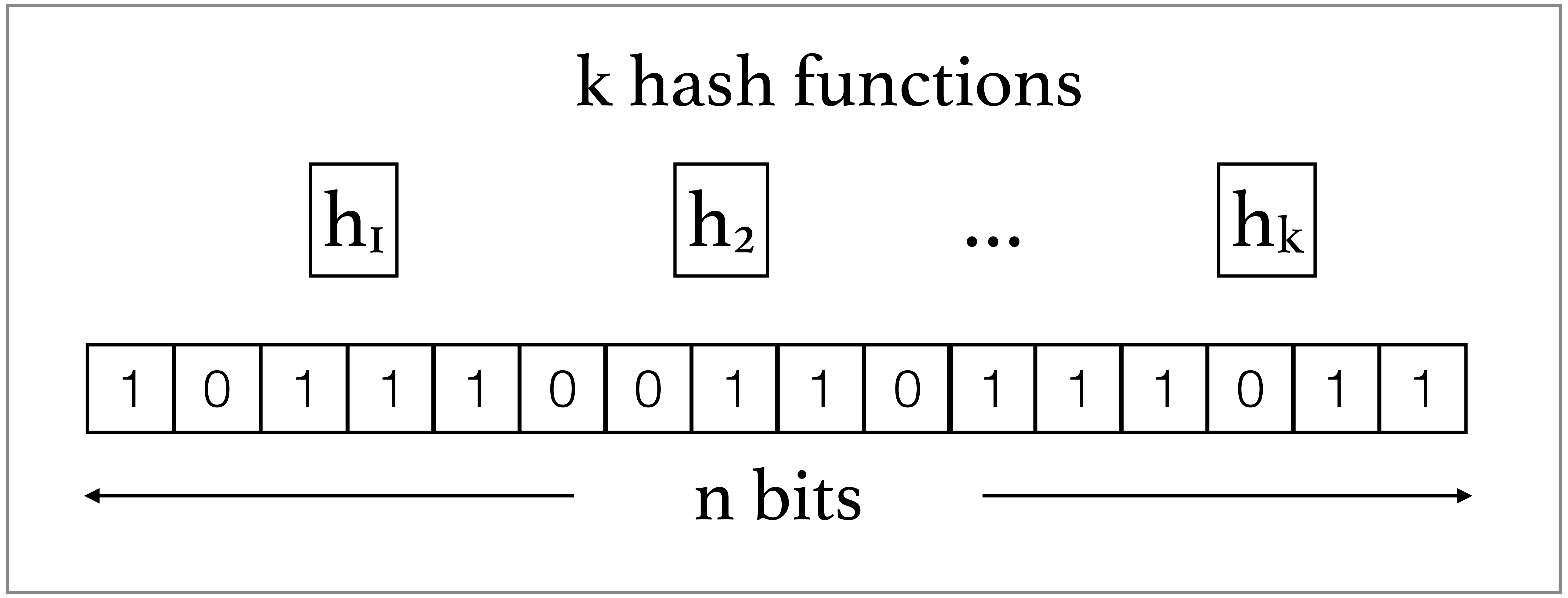


Optimal number of hash functions

Given an expected number of elements and a fixed memory budget, how many hash functions do we need in order to minimize P_{fp}?

After m elements have been inserted to the filter, what is the probability P_0 that a bit is still 0?

1. The probability that h_1 sets bit j is $\frac{1}{n}$

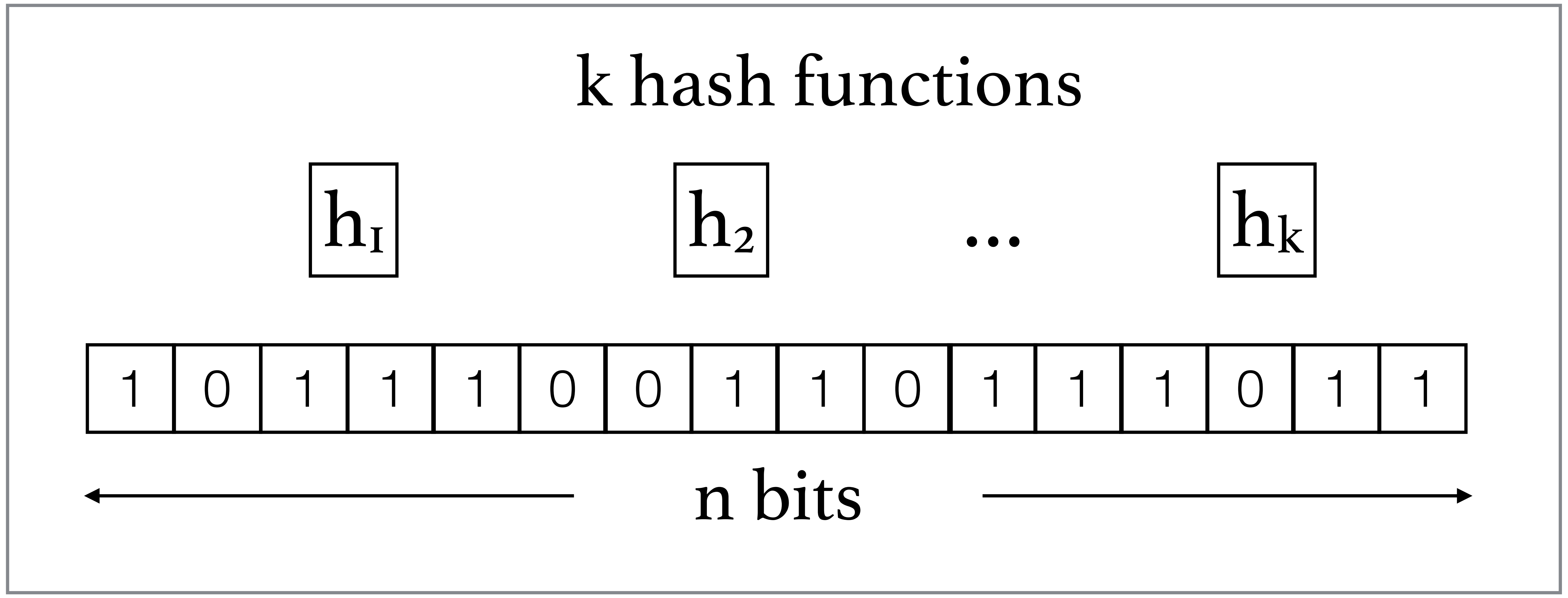


Optimal number of hash functions

Given an expected number of elements and a fixed memory budget, how many hash functions do we need in order to minimize P_{fp}?

After m elements have been inserted to the filter, what is the probability P_0 that a bit is still 0?

1. The probability that h_1 sets bit j is $\frac{1}{n}$
2. The probability that a bit was not set by any of the k hash functions is $(1 - \frac{1}{n})^k$

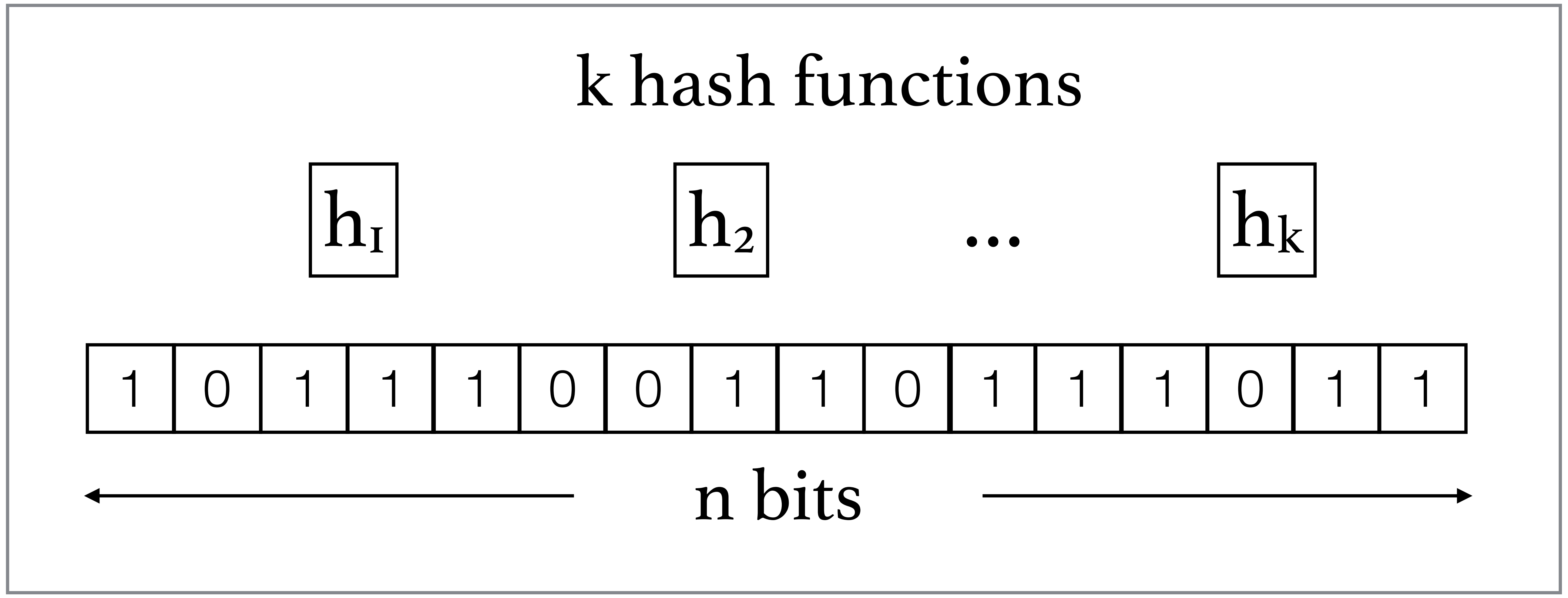


Optimal number of hash functions

Given an expected number of elements and a fixed memory budget, how many hash functions do we need in order to minimize P_{fp}?

After m elements have been inserted to the filter, what is the probability P_o that a bit is still 0?

1. The probability that h_1 sets bit j is $\frac{1}{n}$
2. The probability that a bit was not set by any of the k hash functions is $(1-\frac{1}{n})^k$
3. After all m elements have been inserted, $P_0 = (1-\frac{1}{n})^{km}$



Optimal number of hash functions

$$P_0 = (1 - \frac{1}{n})^{km}$$

We know that $(1 - \epsilon)^{\frac{1}{\epsilon}} \approx \frac{1}{e}$,

so for $\epsilon = \frac{1}{n} \rightarrow P_0 \approx e^{-\frac{km}{n}}$.

Optimal number of hash functions

$$P_0 = (1 - \frac{1}{n})^{km}$$

We know that $(1 - \epsilon)^{\frac{1}{\epsilon}} \approx \frac{1}{e}$,

so for $\epsilon = \frac{1}{n} \rightarrow P_0 \approx e^{-\frac{km}{n}}$.

The probability of a false positive is the probability that an element that was not inserted in the filter is mapped by all k hash functions to 1s:

$$P_{fp} = (1 - P_0)^k \rightarrow P_{fp} = (1 - e^{\frac{km}{n}})^k$$

Optimal number of hash functions

$$P_0 = (1 - \frac{1}{n})^{km}$$

We know that $(1 - \epsilon)^{\frac{1}{\epsilon}} \approx \frac{1}{e}$,

so for $\epsilon = \frac{1}{n} \rightarrow P_0 \approx e^{-\frac{km}{n}}$.

The probability of a false positive is the probability that an element that was not inserted in the filter is mapped by all k hash functions to 1s:

$$P_{fp} = (1 - P_0)^k \rightarrow P_{fp} = (1 - e^{\frac{km}{n}})^k$$

If we take the derivative, the value that minimizes $\mathrm{P_{fp}}$ is $k = \frac{n}{m} ln2$.

Optimal number of hash functions

$$P_{fp} \approx (1 - e^{\frac{km}{n}})^k$$ *

Unfortunately, this classic formula is wrong…

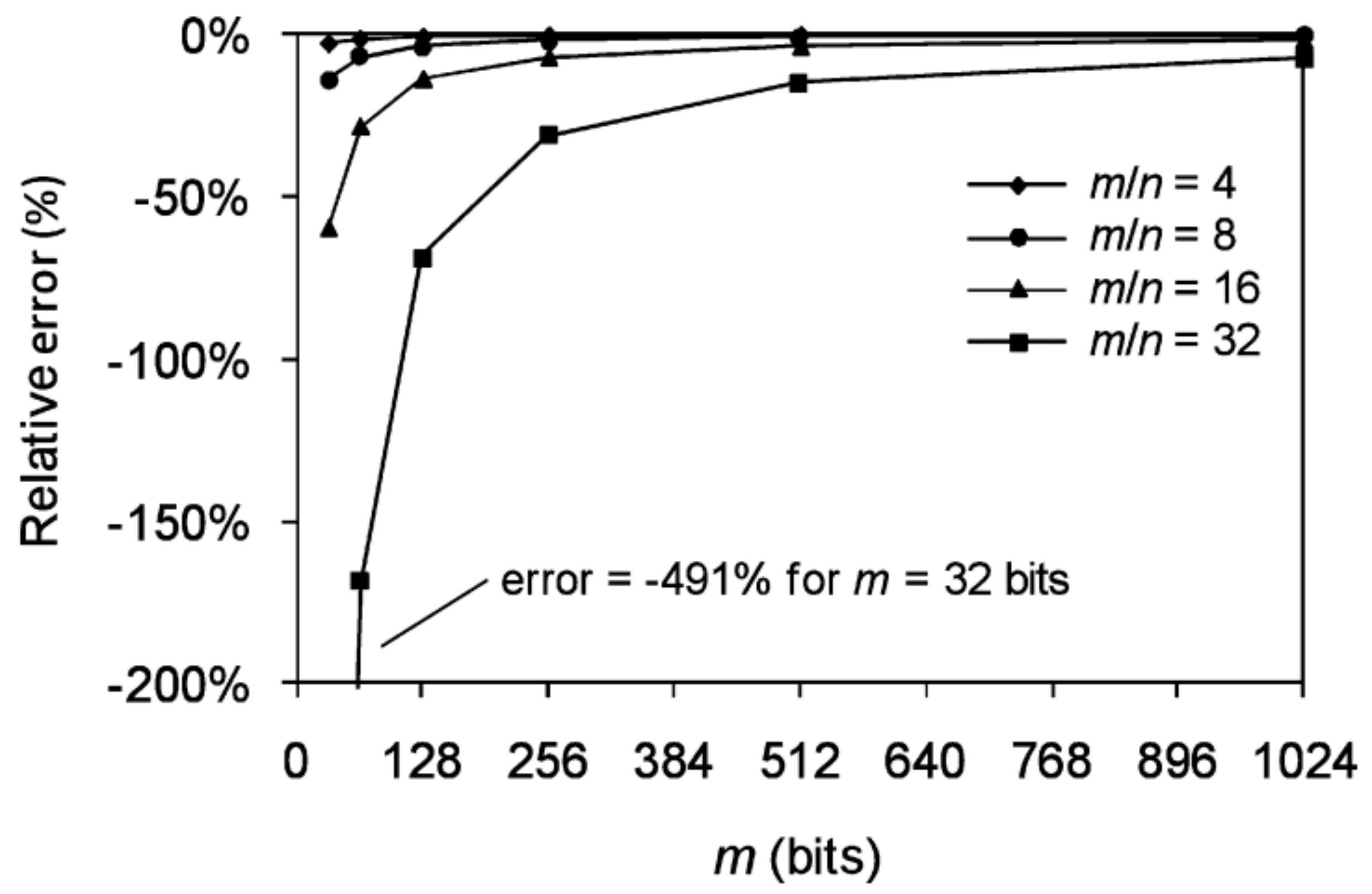


Fig. 2. Relative error of classic versus new formula.

Optimal number of hash functions

$$P_{fp} \approx (1 - e^{\frac{km}{n}})^k$$ *

Unfortunately, this classic formula is wrong…

If after m elements have been inserted to the filter, s bits are set, then:

$$P_{set} = \frac{s}{n} \text{ and } P_{fp} = (\frac{s}{n})^k,$$

which is at least as large as the classic formula.

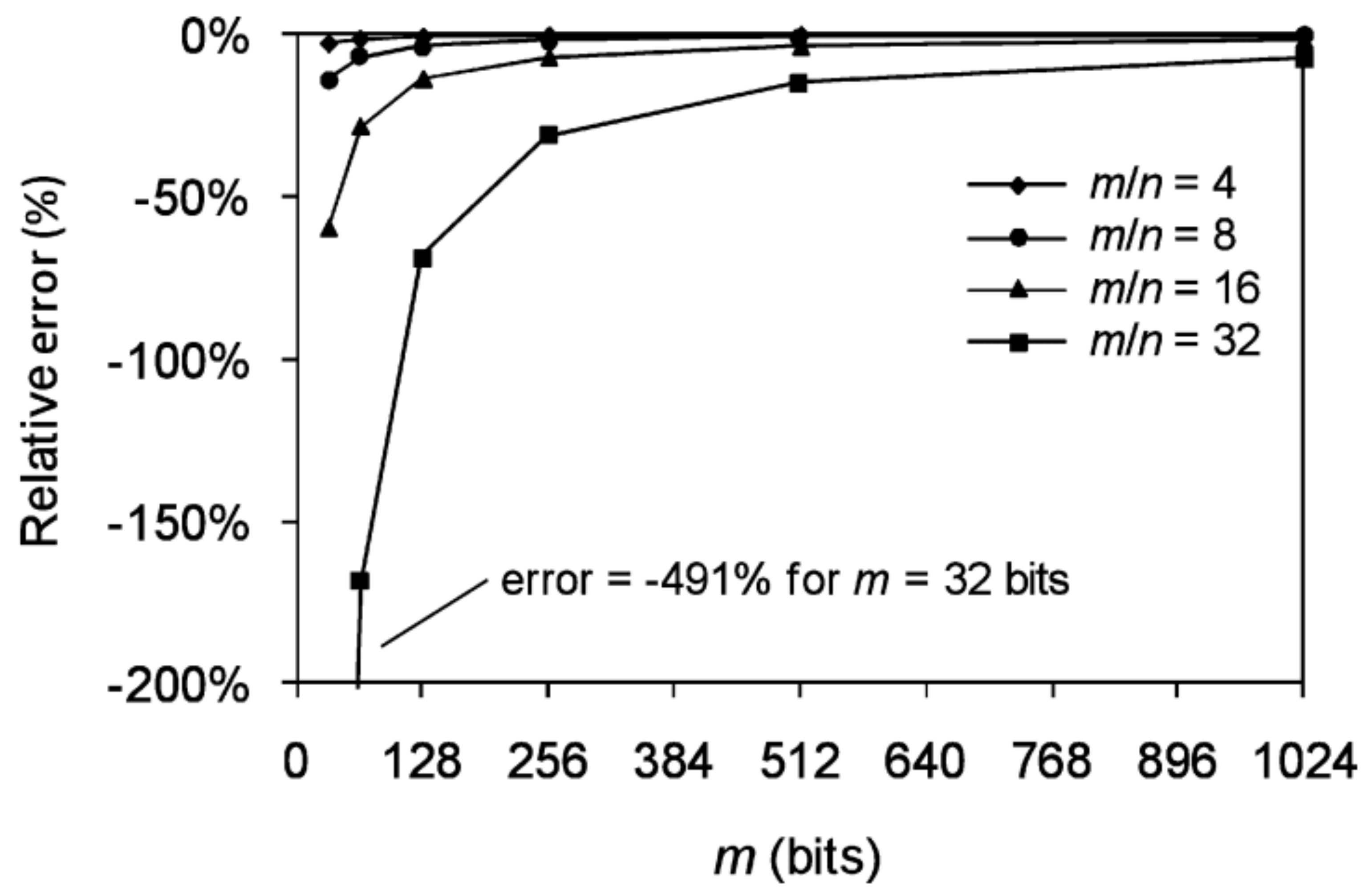


Fig. 2. Relative error of classic versus new formula.

Further reading

- Graham Cormode, Minos Garofalakis, Peter J. Haas and Chris Jermaine. **Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches.** https://dsf.berkeley.edu/cs286/papers/synopses-fntdb2012.pdf
- Jure Lescovec, Anand Rajaraman and Jeffrey David Ullman. **Mining of Massive Datasets**. http://infolab.stanford.edu/~ullman/mmds/book.pdf
- Ken Christensen, Allen Roginsky, Miguel Jimeno. **A new analysis of the false positive rate of a Bloom filter**. Information Processing Letters 110 (2010).