Data Mining and Analysis

Algorithmic Toolbox

CSCE 676 :: Fall 2019
Texas A&M University
Department of Computer Science & Engineering
Prof. James Caverlee

Amazon Dynamo (Key-Value Stores)

Dynamo: Distributed Storage System

Scale

Simple data model (name, value)

key-value store (big hashed table)

Highly available (sacrifice consistency)

Guarantee Service Level Agreements (SLA)

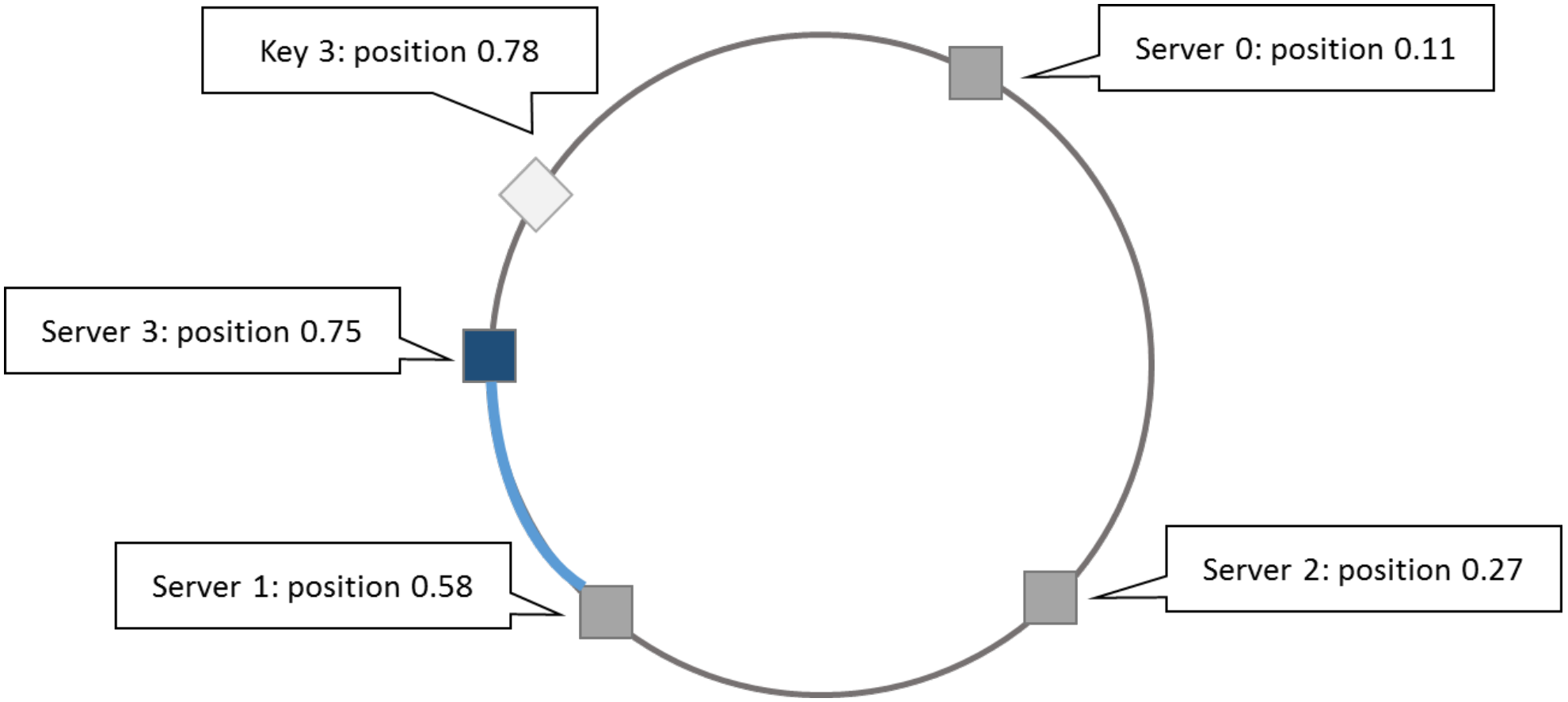
Key-Value Stores

Key–value-based database stores data as (key, value) pairs

- Keys are unique
- Hash map, or hash table or dictionary

NoSQL databases are built with horizontal scalability support in mind —> Distribute hash table over different locations!!

Consistent Hashing



Issues with Consistent Hashing

If two servers end up being mapped close to one another, one of these nodes will end up with few keys to store

In the case that a server is added, all of the keys moved to this new node originate from just one other server

Instead of mapping a server s to a single point on our ring, we map it to multiple positions, called replicas

For each physical server s, we hence end up with r (the number of replicas) points on the ring

Note: each of the replicas still represents the same physical instance ($\leftrightarrow$ redundancy)

Virtual nodes

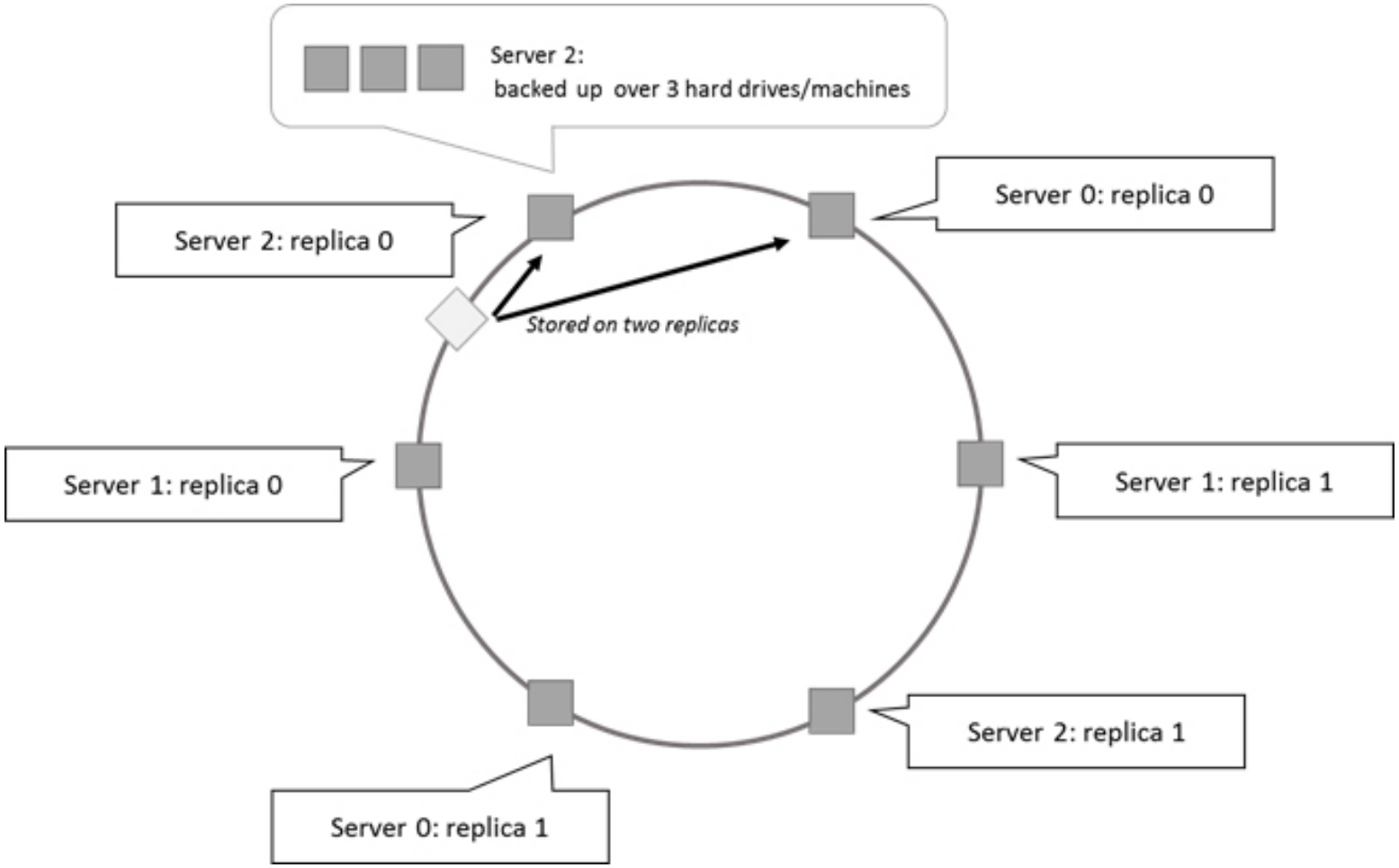
Replication and Redudancy

To handle data replication or redundancy, many vendors extend the consistent hashing mechanism so that key–value pairs are duplicated across multiple nodes

e.g., by storing the key–value pair on two or more nodes clockwise from the key’s position on the ring

Replication and Redundancy

It is also possible to set up a full redundancy scheme in which each node itself corresponds to multiple physical machines, each storing a fully redundant copy of the data



Bloom Filters

Bloom filter: “Space-efficient probabilistic data structure that is used to test whether an element is a member of a set.”

Motivation

Suppose you build a web crawler that collects URLs.

You'd like to store URLs in a space-efficient way so that we can check membership very efficiently

- e.g., Have I seen this URL before? Yes or No.

Ideal case?

- Use O(n) space to store n keys (i.e., URLs)
- From a universe of size U (i.e., all possible URLs), where U >> n
- Check membership in time O(1)
- The data structure maintains a subset S out of the U keys

Requirements: all we need is Insert and IsMember

- IsMember(k) returns true/false
- No need for Delete, Find, Successor

Email Spam Filtering

We know 1 billion “good” email addresses

If an email comes from one of these, it is NOT spam

Pub-sub Systems

You are collecting lots of messages (news articles)

People express interest in certain sets of keywords

Determine whether each message matches user’s interest

Apache Cassandra

Open source distributed database

> “designed to handle large amounts of data across many commodity servers”
>
> “offers robust support for clusters spanning multiple datacenters”

Used by: Facebook, Netflix, Reddit, Soundcloud, Twitter

Cassandra uses bloom filters to save IO when performing a key lookup: each SSTable has a bloom filter associated with it that Cassandra checks before doing any disk seeks, making queries for keys that don't exist almost free

http://wiki.apache.org/cassandra/ArchitectureOverview

Google Chrome

To determine if a particular website is malicious or safe

In essence, check locally in a compact data structure (the bloom filter) rather than issue a (relatively expensive) query to Google servers

If the bloom filter says this is malicious, it probably is!

- For URLs that are called malicious (including “false positives” — in this case, a URL that is legitimate is labeled as malicious), hit the Google service for additional info

http://blog.alexyakunin.com/2010/03/nice-bloom-filter-application.html

Bitly

Check for malicious URLs

http://word.bitly.com/post/28558800777/dablooms-an-open-source-scalable-counting-bloom

to verify payments

Recall: Motivation

Suppose you build a web crawler that collects URLs.

You'd like to store URLs in a space-efficient way so that we can check membership very efficiently

- e.g., Have I seen this URL before? Yes or No.

Ideal case?

- Use O(n) space to store n keys (i.e., URLs)
- From a universe of size U (i.e., all possible URLs), where U >> n
- Check membership in time O(1)
- The data structure maintains a subset S out of the U keys

Requirements: all we need is Insert and IsMember

- IsMember(k) returns true/false
- No need for Delete, Find, Successor

Ideas?

Binary search tree?

Trie?

Linked list?

…

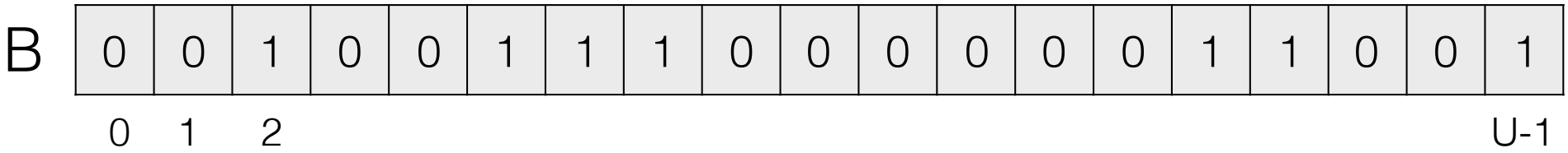
(typically, these will store the data items themselves, meaning they are not very space efficient)

One possible solution

Maintain a bitstring B of U bits in an array

each bit B[k] is either 0 or 1

1 means k is in S



Problems?

space depends on U (which can be BIG!!) — think of all possible URLs

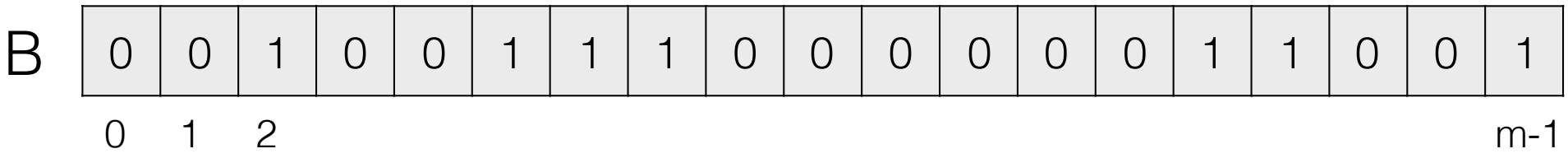
Second solution: Hashing!

Maintain an m-bit string B (where m << U)

We have a hash function that maps a k to an integer h(k) in [0,m-1)

Set bit number h(k) of B to 1 when key k is inserted

IsMember? Just check whether the bit of B at position h(k) is equal to 1



Example: Suppose m = 3

hash(www.tamu.edu) = 0,

hash(www.ut.edu) = 2,

hash(www.lame.com) = 2

Start	0	0	0
Insert(www.tamu.edu)	1	0	0
Insert(www.ut.edu)	1	0	1

Member(www.lame.com) ??

Problems?

If m < U, we will have some keys that hash to the same position (collisions)

If we call IsMember(k) for some key not in S, if h(k) = h(k') for some k' in S, then we will incorrectly output True

False positives!

Hopefully:

Our hash function is uniformly random

That is, the probability that h(k) = j is equal to 1/m for all j between 1 and m

Example

|S| = 1 billion email addresses

|B|= 1GB = 8 billion bits

If the email address is in S, then it surely hashes to a bucket that has the bit set to 1, so it always gets through (no false negatives)

Approximately 1/8 of the bits are set to 1, so about 1/8th of the addresses not in S get through to the output (false positives)

Actually, less than 1/8th, because more than one address might hash to the same bit

Analysis: Throwing Darts

Consider: If we throw m darts into n equally likely targets, what is the probability that a target gets at least one dart?

In our case:

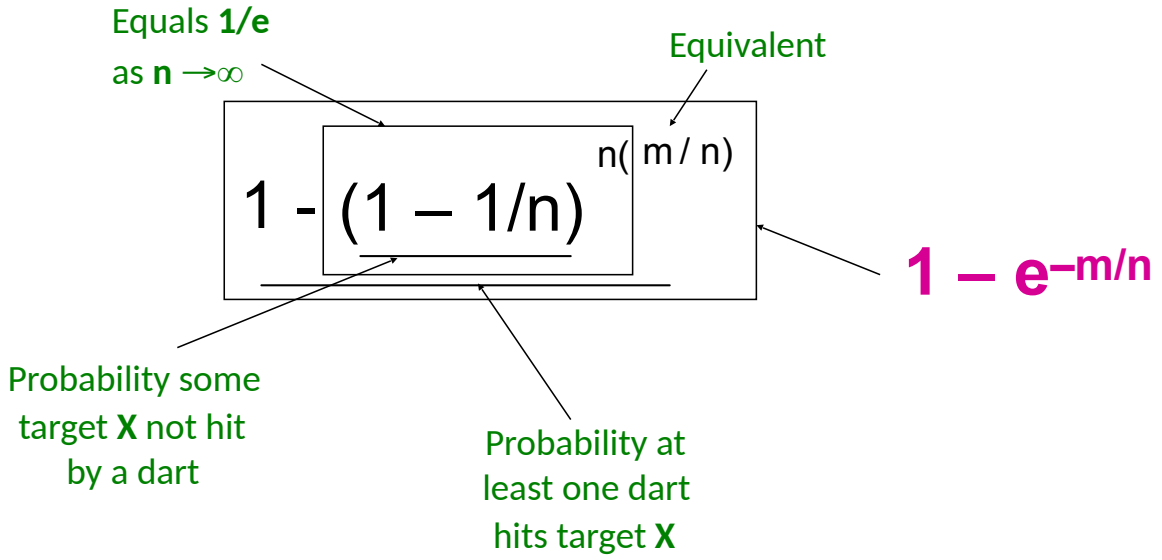
Targets = bits/buckets

Darts = hash values of items

Analysis: Throwing Darts

We have m darts, n targets

What is the probability that a target gets at least one dart?



Analysis: Throwing Darts

Fraction of 1s in the array B =
= probability of false positive = $1 - e^{-m/n}$

Example: 10^9 darts, $8 \cdot 10^9$ targets

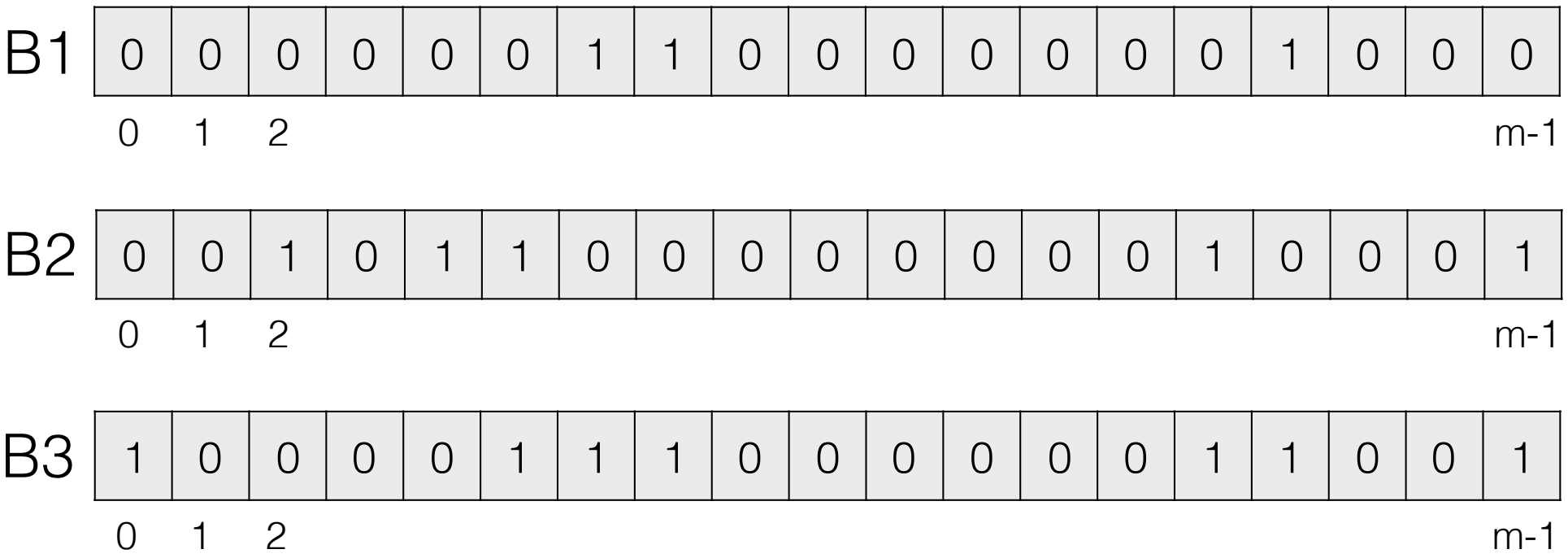
Fraction of 1s in B = $1 - e^{-1/8} = 0.1175$

Compare with our earlier estimate: 1/8 = 0.125

Doing better? Third solution ...

Use multiple (r) hash functions

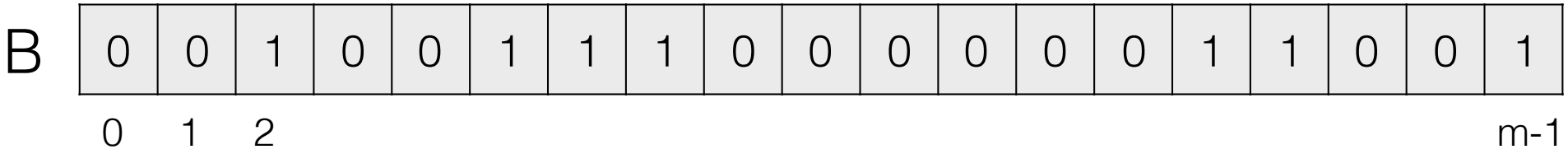
r bitstrings, one per hash function



Problems?

… Bloom filter!

Multiple hash functions! One bitstring!



A Bloom filter consists of a bitstring B of m bits and a set of r hash functions

Each hash function maps a key k to an integer between 1 and m

Need to choose the parameters: m and r

Bloom filter operations

```
Insert(k)
    for i from 1 to r
        B[h_i(k)] <— 1

IsMember(k)
    for i from 1 to r
        if B[h_i(k)] == 0
            return false
    return true
```

Example

Suppose m=4, r =2 (that is, we have two hash functions)

hash1(www.tamu.edu) = 0; hash2(www.tamu.edu) = 1

hash1(www.ut.edu) = 2; hash2(www.ut.edu) = 1

hash1(www.lame.com) = 2; hash2(www.lame.com) = 3

Start	0	0	0	0
Insert(www.tamu.edu)	1	1	0	0
Insert(www.ut.edu)	1	1	1	0
Member(www.lame.com)	??			

Bloom Filter Analysis

What fraction of the bit vector B are 1s?

Throwing $k \cdot m$ darts at n targets

So fraction of 1s is $(1 - e^{-km/n})$

But we have k independent hash functions and we only let the element x through if all k hash element x to a bucket of value 1

So, false positive probability = $(1 - e^{-km/n})^k$

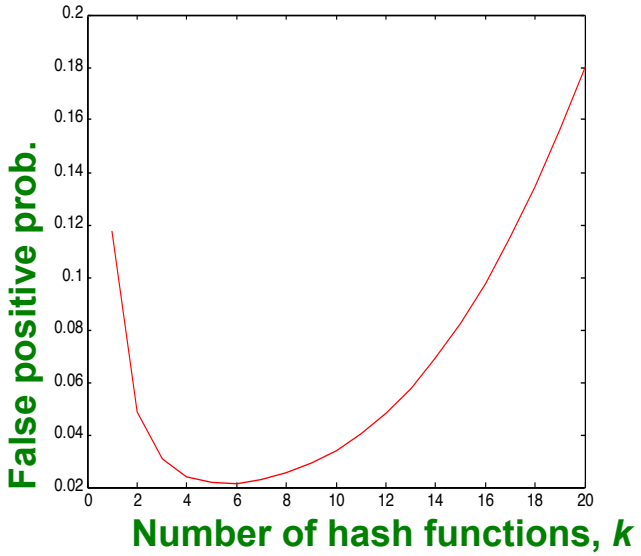
Bloom Filter Analysis

m = 1 billion, n = 8 billion

k = 1: $(1 - e^{-1/8}) = 0.1175$

k = 2: $(1 - e^{-1/4})^2 = 0.0493$

What happens as we keep increasing k?



"Optimal" value of k: n/m ln(2)

In our case: Optimal k = 8 ln(2) = 5.54 ≈ 6

Error at k = 6: $(1 - e^{-1/6})^2 = 0.0235$

Challenge time

Bloom filters are great for checking membership

What if we want to handle deletions as well?

Advanced: Counting Bloom Filter

- Can handle deletions
- Use counters instead of 0/1s
- When adding an element, increment the counters
- When deleting an element, decrement the counters
- Counters must be large enough to avoid overflow (4 bits)