CAI: Cerca i Anàlisi d'Informació
Grau en Ciència i Enginyeria de Dades, UPC

5. Scaling up

November 1, 2019

Slides by Marta Arias, José Luis Balcázar, Ramon Ferrer-i-Cancho, Ricard Gavaldà, Department of Computer Science, UPC

Contents

Google 1998. Some figures

- 24 million pages
- 259 million anchors
- 147 Gb of text
- 256 Mb main memory per machine
- 14 million terms in lexicon
- 3 crawlers, 300 connection per crawler
- 100 webpages crawled / second, 600 Kb/second
- 41 Gb inverted index
- 55 Gb info to answer queries; 7Gb if doc index compressed
- Anticipate hitting O.S. limits at about 100 million pages

Google today?

- Current figures = $\times$ 1,000 to $\times$ 10,000

- 100s petabytes transferred per day?
- 100s exabytes of storage?
- Several 10s of copies of the accessible web
- many million machines

Google in 2003

- More applications, not just web search
- Many machines, many data centers, many programmers
- Huge & complex data
- Need for abstraction layers

Three influential proposals:

- Hardware abstraction: The Google Cluster
- Data abstraction: The Google File System

  BigFile (2003), BigTable (2006)
- Programming model: MapReduce

Google cluster, 2003: Design criteria

Use more cheap machines, not expensive servers

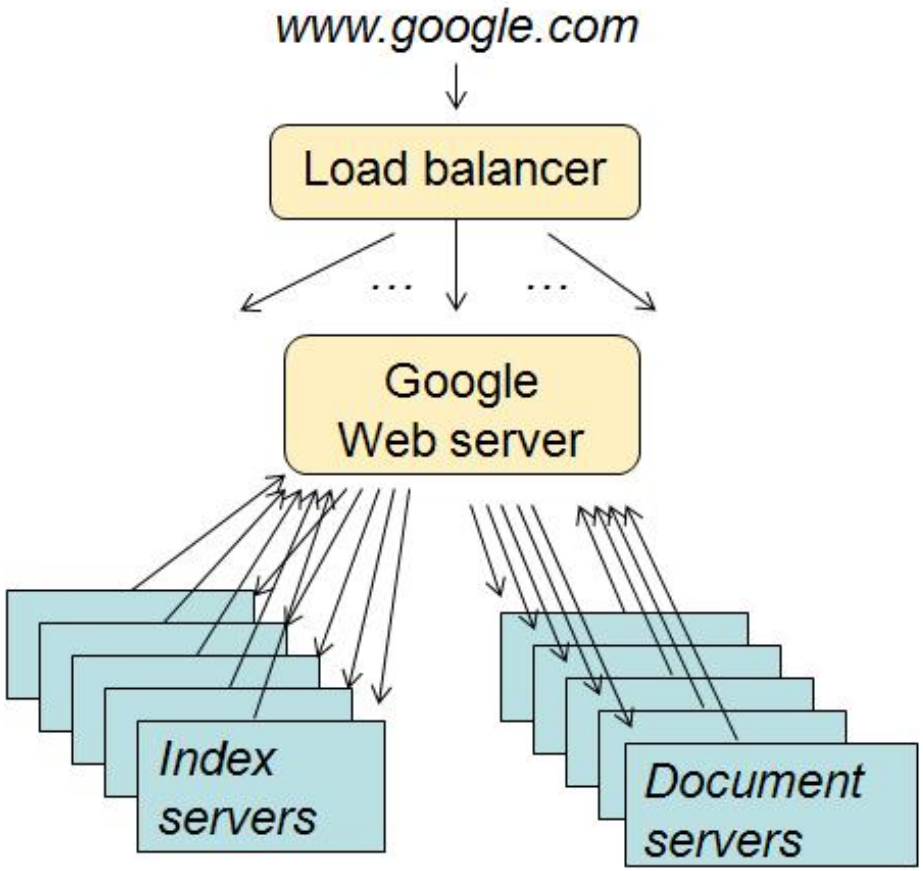
- High task parallelism; Little instruction parallelism

  (e.g., process posting lists, summarize docs)
- Peak processor performance less important than price/performance

  price is superlinear in performance!
- Commodity-class PCs. Cheap, easy to make redundant
- Redundancy for high throughput
- Reliability for free given redundancy. Managed by soft
- Short-lived anyway (< 3 years)

L.A. Barroso, J. Dean, U. Hölzle: “Web Search for a Planet: The Google Cluster Architecture”, 2003

Google cluster for web search



- Load balancer chooses freest / closest GWS
- GWS asks several index servers
- They compute hit lists for query terms, intersect them, and rank them
- Answer (docid list) returned to GWS
- GWS then asks several document servers
- They compute query-specific summary, url, etc.
- GWS formats an html page & returns to user

Index “shards”

- Documents randomly distributed into “index shards”
- Several replicas (index servers) for each indexshard
- Queries routed through local load balancer
- For speed & fault tolerance
- Updates are infrequent, unlike traditional DB’s
- Server can be temporally disconnected while updated

The Google File System, 2003

- System made of cheap PC's that fail often
- Must constantly monitor itself and recover from failures transparently and routinely
- Modest number of large files (GB's and more)
- Supports small files but not optimized for it
- Mix of large streaming reads + small random reads
- Occasionally large continuous writes
- Extremely high concurrency (on same files)

S. Ghemawat, H. Gobioff, Sh.-T. Leung: "The Google File System", 2003

The Google File System, 2003

- One GFS cluster = 1 master process + several chunkservers
- BigFile broken up in chunks
- Each chunk replicated (in different racks, for safety)
- Master knows mapping chunks $\rightarrow$ chunkservers
- Each chunk unique 64-bit identifier
- Master does not serve data: points clients to right chunkserver
- Chunkservers are stateless; master state replicated
- Heartbeat algorithm: detect & put aside failed chunkservers

MapReduce and descendance

- Mapreduce: Large-scale programming model developed at Google (2004)
  - Proprietary implementation
  - Implements old ideas from functional programming, distributed systems, DB's ...
- Hadoop: Open source (Apache) implementation at Yahoo! (2006 and on)
  - HDFS, Pig, Hive...
  - ...
- Spark, Kafka, etc. to address shortcomings of Hadoop.

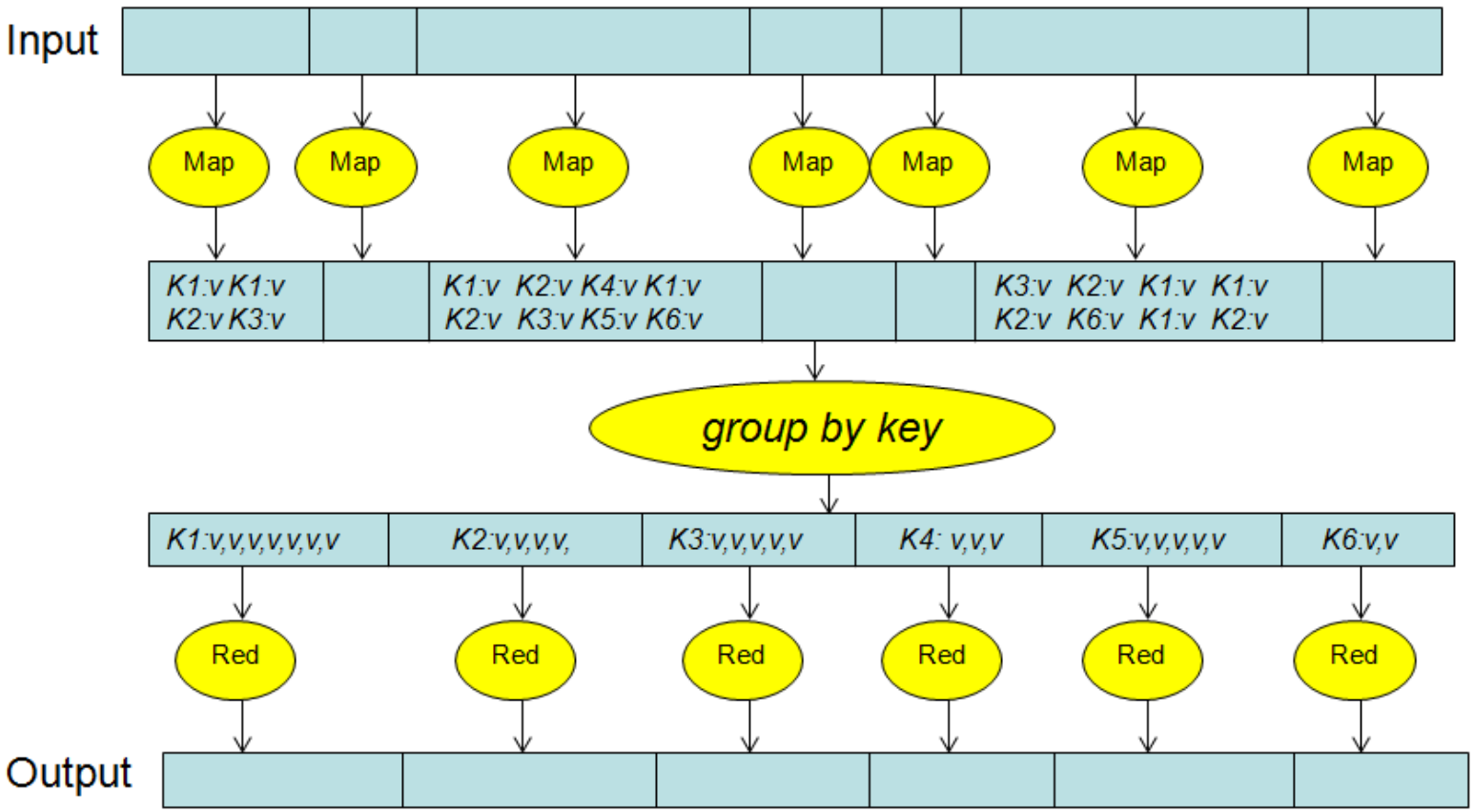
MapReduce

Design goals:

- Scalability to large data volumes and number of machines
  - 1000's of machines, 10,000's disks
  - Abstract hardware & distribution (compare MPI: explicit flow)
  - Easy to use: good learning curve for programmers
- Cost-efficiency:
  - Commodity machines: cheap, but unreliable
  - Commodity network
  - Automatic fault-tolerance and tuning. Fewer administrators

Semantics

Key step, handled by the platform: group by or shuffle by key



Example: Inverted Index

(Replacement for all the low-level, barrel, RAM vs. disk stuff)

Input: A set of text files
Output: For each word, the list of files that contain it

```
map(filename):
    foreach word in the file text do
        output (word, filename)

combine(word,L):
    remove duplicates in L;
    output (word,L)

reduce(word,L):
    //want sorted posting lists
    output (word,sort(L))
```

Big Data

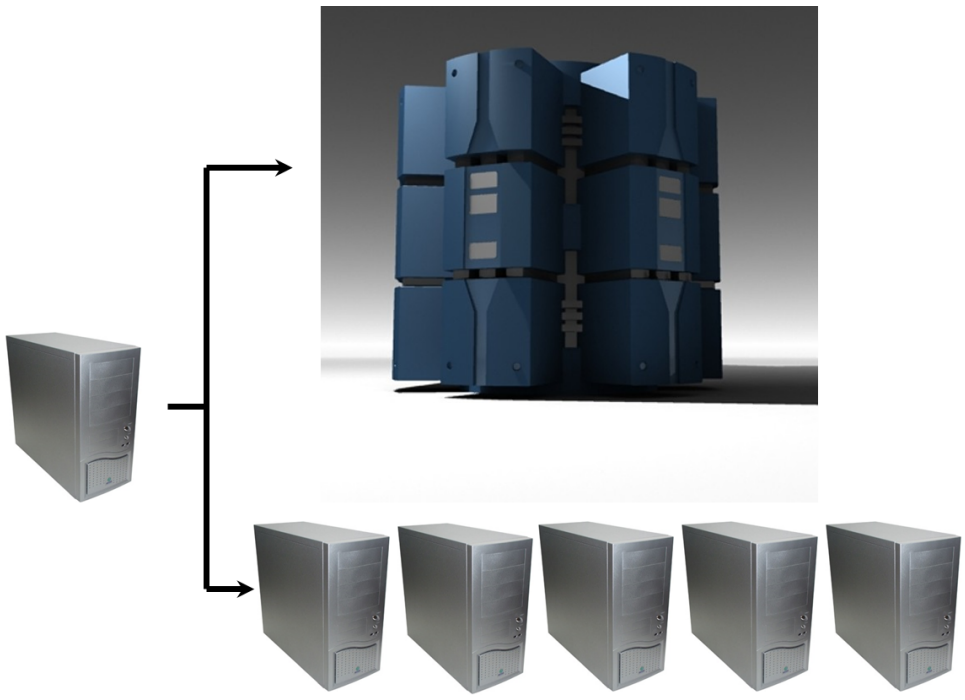
- Sets of data whose size surpasses what data storage tools can typically handle
- The 3 V's: Volume, Velocity, Variety, etc.
- Figure that grows concurrently with technology
- The problem has always existed and driven innovation

Big Data

- Technological problem: how to store, use & analyze?
- Or business problem?
  - what to look for in the data?
  - what questions to ask?
  - how to model the data?
  - where to start?

The problem with Relational DBs

- The relational DB has ruled for 2-3 decades
- Superb capabilities, superb implementations
- One of the ingredients of the web revolution
  - LAMP = Linux + Apache HTTP server + MySQL + PHP
- Main problem: scalability



Scaling UP

- Price superlinear in performance & power
- Performance ceiling

Scaling OUT

- No performance ceiling, but
- More complex management
- More complex programming
- Problems keeping ACID properties

The problem with Relational DBs

- RDBMS scale *up* well (single node). Don't scale *out* well
- Vertical partitioning: Different tables in different servers
- Horizontal partitioning: Rows of same table in different servers

Apparent solution: Replication and caches

- Good for fault-tolerance, for sure
- OK for many concurrent reads
- Not much help with writes, if we want to keep ACID

There's a reason: The CAP theorem

Three desirable properties:

- Consistency: After an update to the object, every access to the object will return the updated value
- Availability: At all times, all DB clients are able to access *some* version of the data. Equivalently, every request receives an answer
- Partition tolerance: The DB is split over multiple servers communicating over a network. Messages among nodes may be lost arbitrarily

The CAP theorem [Brewer 00, Gilbert-Lynch 02] says:

> No distributed system can have these three properties

In other words: In a system made up of nonreliable nodes and network, it is impossible to implement atomic reads & writes and ensure that every request has an answer.

CAP theorem: Proof

- Two nodes, A, B
- A gets request “read(x)”
- To be consistent, A must check whether some “write(x,value)” performed on B
- . . . so sends a message to B
- If A doesn’t hear from B, either A answers (inconsistently)
- or else A does not answer (not available)

The problem with RDBMS

- A truly distributed, truly relational DBMS should have Consistency, Availability, and Partition Tolerance
- . . . which is impossible

- Relational is full C+A, at the cost of P

- NoSQL obtains scalability by going for A+P or for C+P
- . . . and as much of the third one as possible

NoSQL: Generalities

Properties of most NoSQL DB's:

1. BASE instead of ACID
2. Simple queries. No joins
3. No schema
4. Decentralized, partitioned (even multi data center)
5. Linearly scalable using commodity hardware
6. Fault tolerance
7. Not for online (complex) transaction processing
8. Not for datawarehousing

BASE, eventual consistency

- Basically Available, Soft state, Eventual consistency
- Eventual consistency: If no new updates are made to an object, eventually all accesses will return the last updated value.
- ACID is pessimistic. BASE is optimistic. Accepts that DB consistency will be in a state of flux
- Surprisingly, OK with many applications
- And allows *far* more scalability than ACID

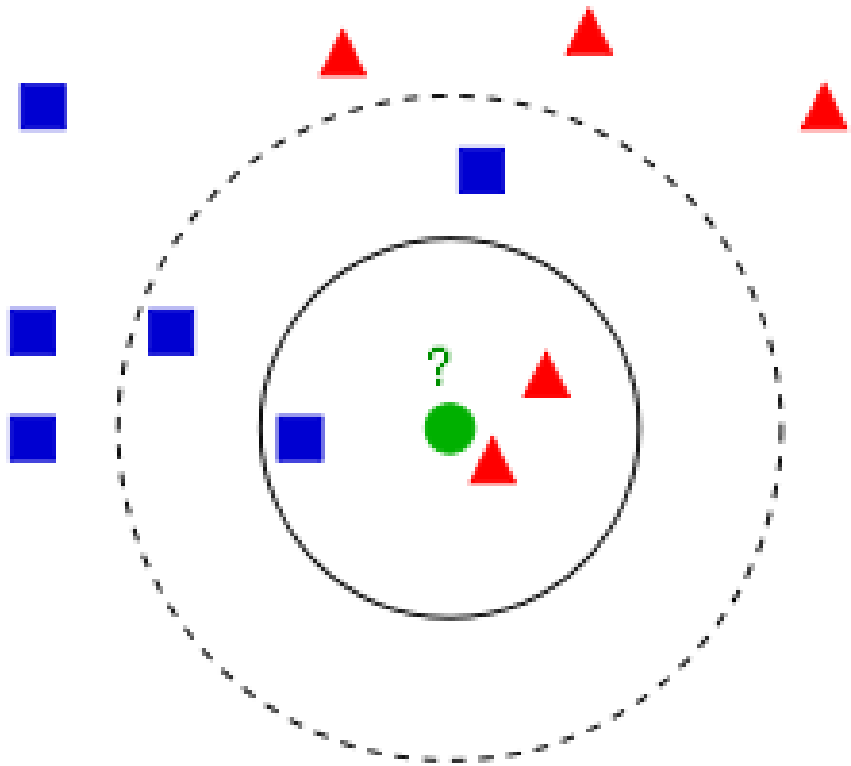
Two useful algorithms

- Finding near-duplicate items: Locality Sensitive Hashing
- Distributed data: Consistent hashing

Locality Sensitive Hashing: Motivation

Find similar items in high dimensions, quickly

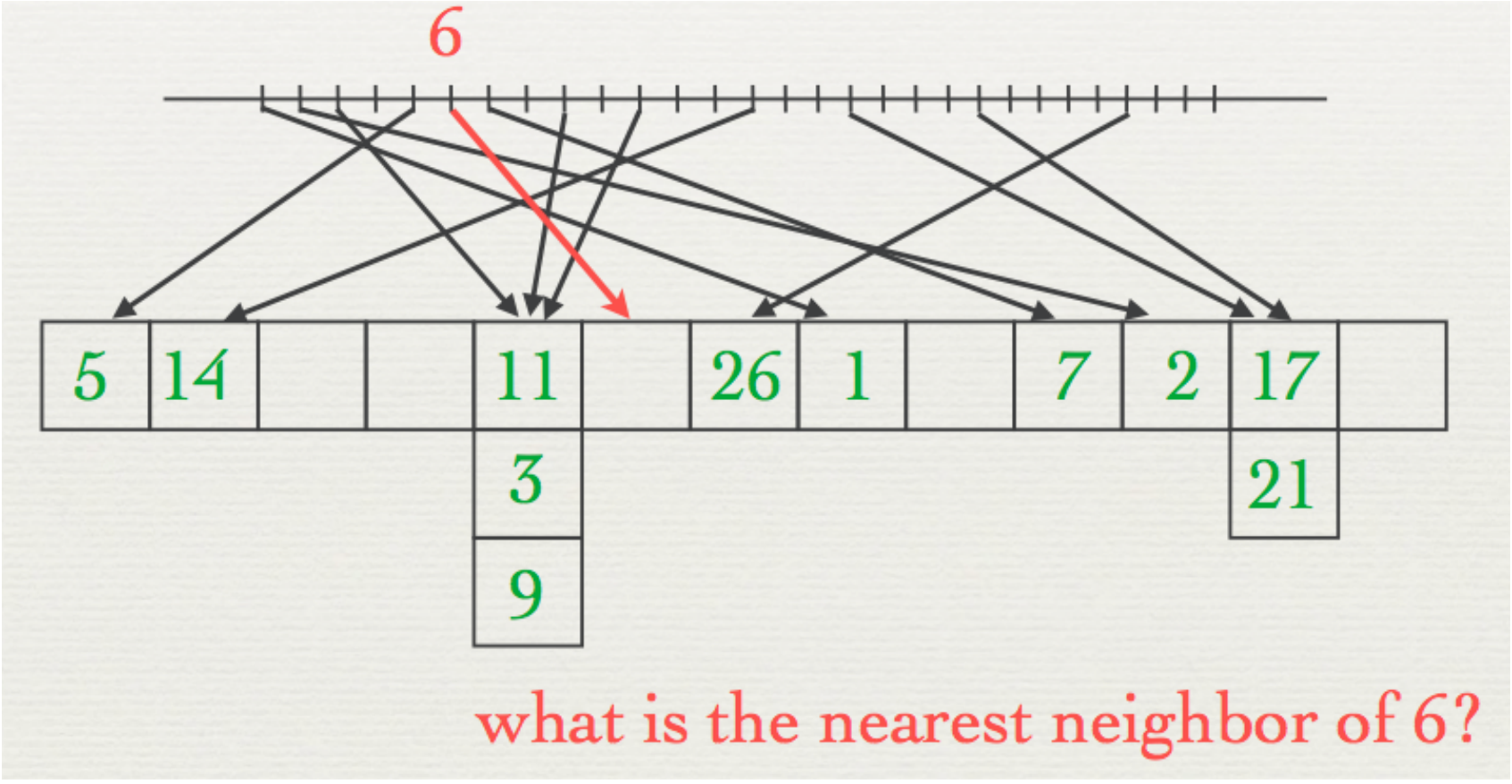
Could be useful, for example, in nearest neighbor algorithm..
but in a large, high dimensional dataset this may be difficult!



Very similar documents, images, audios, genomes. . .

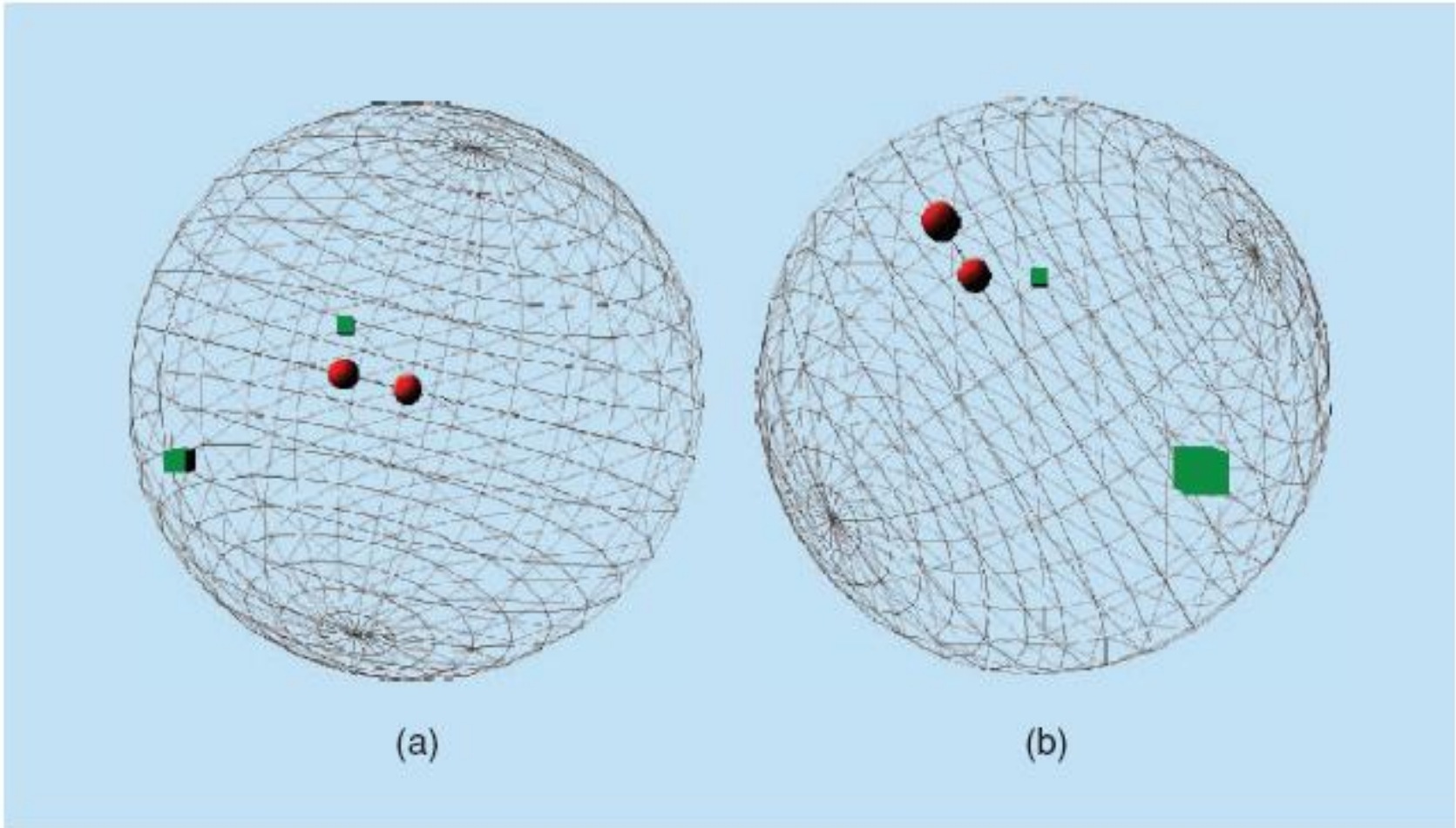
Motivation

Hashing is good for checking existence, not nearest neighbors



Motivation

Main idea: want hashing functions that map similar objects to nearby positions using *projections*



[FIG1] Two examples showing projections of two close (circles) and two distant (squares) points onto the printed page.

Hashing

(

Hashing

A hash function $h : X \to Y$ distributes elements Y randomly among elements in Y

At least for some subset $S \subseteq X$ that we want to hash (e.g. some set of strings of length at most 2000 characters)

But when we say randomly, we probably need to talk about probabilities. . .

Fact: For every fixed h there is a set $S \subseteq X$ of size $S \geq |X|/|Y|$ that gets mapped by h to a single value.

= If I know your h, I can sabotage your hashing.

Hashing

Let $\mathcal{H}$ be a set of functions $h : X \to Y$.

The randomness is in picking a function from $\mathcal{H}$ at random.

An incredibly good $\mathcal{H}$ looks like each value is randomly generated. For every $x \in X$, $y \in Y$

$$Pr_{h \in \mathcal{H}}[h(x) = y \mid \text{all other values of } h] = 1/|Y|.$$

but once we pick h, each time we ask $h(x)$ we must get the same answer.

Storing such an h needs $\Omega(\log |Y|^{|X|}) = \Omega(|X| \log |Y|)$ bits. Not practical.

I skip the $h \in \mathcal{H}$ subindex from now on.

Universal Hashing

A weaker condition

We say that $\mathcal{H}$ is a universal family of hash functions if for every two different $x, x' \in X$

$$Pr[h(x) = h(x')] \leq 1/|Y|.$$

Fact: if $\mathcal{H}$ is universal, then for each $y \in Y$

$$E[\text{number of } x \in X \text{ s.t. } h(x) = y] = |X|/|Y|$$

but still the value of $h(x)$ may tell something about the probability of y.

Pairwise independence (a.k.a. strong universality)

$\mathcal{H}$ is a pairwise independent family of hash functions if for every $x, x' \in X$ ($x \neq x'$), $y, y' \in Y$

$$Pr[h(x) = y \wedge h(x') = y'] = Pr[h(x) = y] \cdot Pr[h(x') = y'] = 1/|Y|^2.$$

Equivalently, for all $x, x' \in X$,

$$Pr[h(x) = y | h(x') = y'] = Pr[h(x) = y] = 1/|Y|.$$

Fact (check): pairwise independence implies universality.

Building pairwise independent functions

Let p be prime, $[p] = \{0..p-1\}$

Fact: $\{h(x) = (ax + b) \bmod p \mid a, b \in [p]\}$ is a family of p.i. functions from $[p]$ to $[p]$.

Proof: Fix $x \neq x'$, y, y'. The system of equations

$$ax + b = y, \quad ax' + b = y'$$

has exactly one solution for (a, b) in $[p] \times [p]$.

Note: If I tell you $h(x), h(x')$ then you solve for a, b and you know all values of h.

Building pairwise independent functions

One such h can be chosen and stored in $2 \log p$ bits
(pick $a, b \in [p]$ uniformly at random.)

Can be generalized to generate k-wise independent functions.

Building pairwise independent bits

Let $\mathcal{H}$ be a family of p.i. functions from X to $[n]$.

Then the family

$$\{h(x) \bmod 2 \mid h \in \mathcal{H}\}$$

is a family of p.i. functions from X to $\{0,1\}$. That is

$$Pr[h(x) = b \mid h(x') = b'] = 1/2.$$

Building pairwise independent bits

Another construction: $X = \{0,1\}^d$ to $Y = \{0,1\}$.

Then the family of functions

$$\{h(x) = \bigoplus_{i=1..d} (r_i \wedge x_i) \bmod 2 \mid r \in \{0,1\}^d\}$$

is a family of p.i. functions from $\{0,1\}^d$ to $\{0,1\}$.

(inner product in GF[2])

Hashing

)

Different types of hashing functions

Perfect hashing

- Provide 1-1 mapping of objects to bucket ids
- Any two different objects mapped to different buckets (no collisions)

Universal hashing

- Probability of collision for different objects is at most $1/n$

Locality sensitive hashing (lsh)

- Collision probability for *similar* objects is high enough
- Collision probability for *dissimilar* objects is low

Locality sensitive hashing functions

Definition

A family $\mathcal{F}$ is called $(s, c \cdot s, p_1, p_2)$-sensitive if for any two objects x and y we have:

- If $s(x, y) \geq s$, then $P[h(x) = h(y)] \geq p_1$
- If $s(x, y) \leq c \cdot s$, then $P[h(x) = h(y)] \leq p_2$

where the probability is taken over chosing h from $\mathcal{F}$, and $c < 1$, $p_1 > p_2$

How to use LSH to find nearest neighbor

The main idea

Pick a hashing function h from appropriate family $\mathcal{F}$

Preprocessing

- Compute $h(x)$ for all objects x in our available dataset

On arrival of query q

- Compute $h(q)$ for query object
- Sequentially check nearest neighbor in "bucket" $h(q)$

Locality sensitive hashing I

An example for bit vectors

- Objects are vectors in $\{0,1\}^d$
- Distances are measured using Hamming distance

$$d(x,y) = \sum_{i=1}^{d} |x_i - y_i|$$

- Similarity is measured as nr. of common bits divided by length of vector

$$s(x,y) = 1 - \frac{d(x,y)}{d}$$

- For example, if $x = 10010$ and $y = 11011$, then $d(x,y) = 2$ and $s(x,y) = 1 - 2/5 = 0.6$

Locality sensitive hashing II

An example for bit vectors

- Consider the following "hashing family": sample the i-th bit of a vector, i.e. $\mathcal{F} = \{f_i | i \in [d]\}$ where $f_i(x) = x_i$
- Then, the probability of collision

$$P[h(x) = h(y)] = s(x, y)$$

  (the probability is taken over chosing a random $h \in \mathcal{F}$)
- Hence $\mathcal{F}$ is (s, cs, s, cs)-sensitive (with $c < 1$ so that $s > cs$ as required)

Locality sensitive hashing III

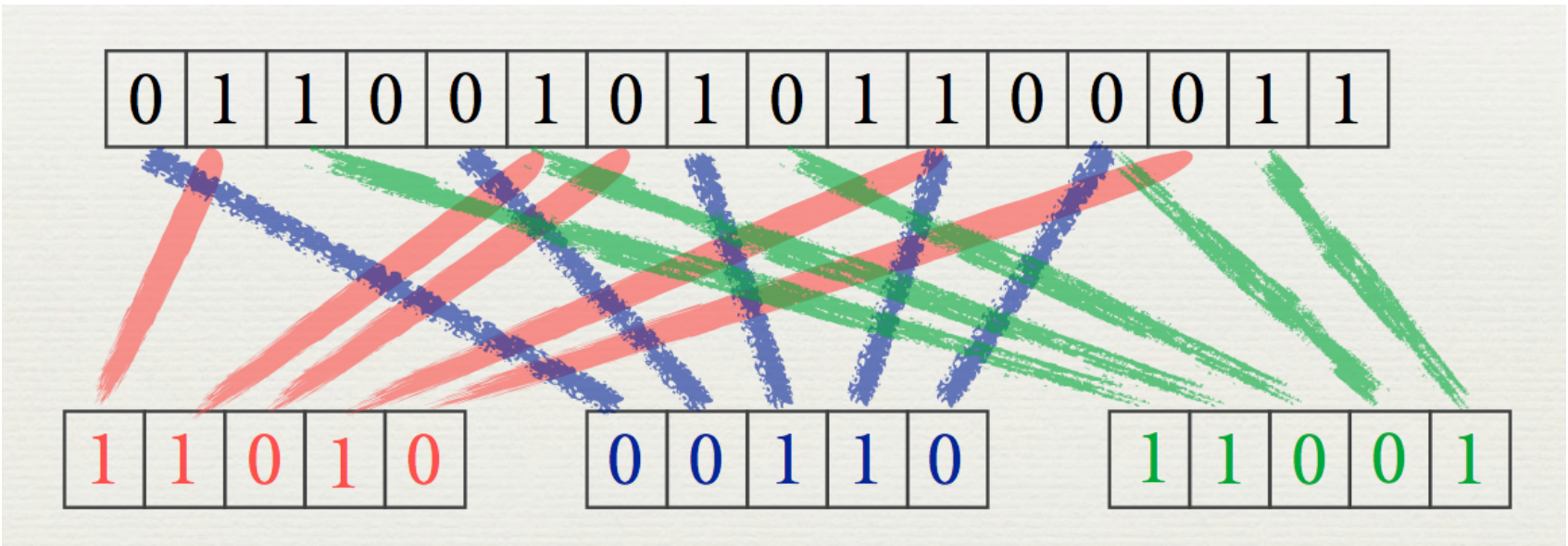
An example for bit vectors

- If gap between s and cs is too small (between p_1 and p_2), we can amplify it:
  - By stacking together k hash functions
    - $h(x) = (h_1(x), .., h_k(x))$ where $h_i \in \mathcal{F}$
    - Probability of collision of similar objects decreases to s^k
    - Probability of collision of dissimilar objects decreases even more to $(cs)^k$
  - By repeating the process m times
    - Probability of collision of similar objects increases to $1 - (1 - s)^m$
  - Choosing k and m appropriately, can achieve a family that is $(s, cs, 1 - (1 - s^k)^m, 1 - (1 - (cs)^k)^m)$-sensitive

Locality sensitive hashing IV

An example for bit vectors

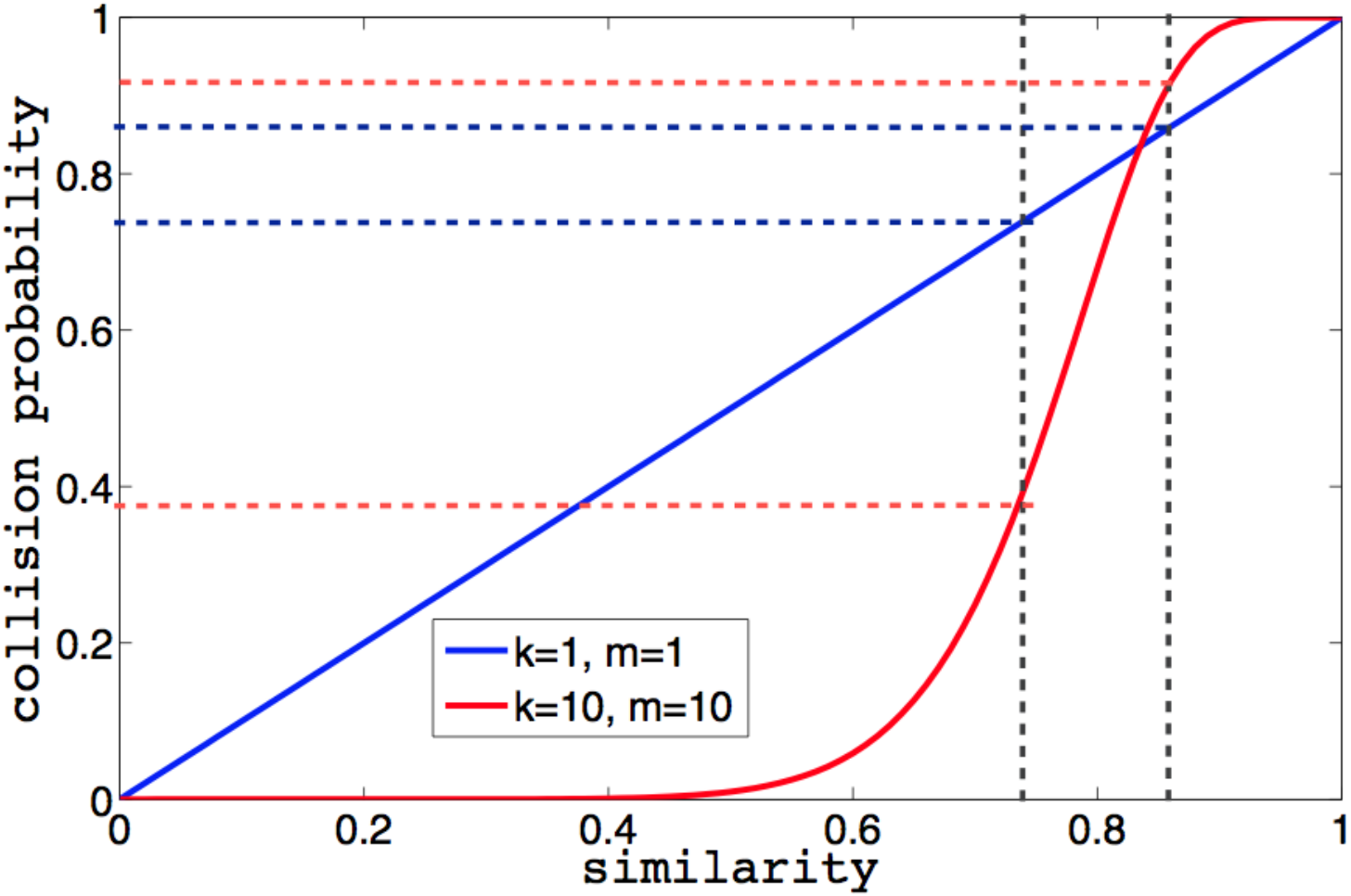
A string of d bits gets hashed to a string of km bits.
Here, $k = 5, m = 3$



Locality sensitive hashing V

An example for bit vectors

Collision probability is $1 - (1 - s^k)^m$



Similarity search becomes. . .

Pseudocode

Preprocessing:

- Input: set of objects X
- for $i = 1..m$
  - for each $x \in X$
    - stack k hash functions and form $x_i = (h_1(x), .., h_k(x))$
    - store x in bucket given by x_i

At query time:

- Input: query object q
- $Z = \emptyset$
- for $i = 1..m$
  - stack k hash functions and form $q_i = (h_1(q), .., h_k(q))$
  - $Z_i = \{$ objects found in bucket $q_i \}$
  - $Z = Z \cup Z_i$
- Output all $z \in Z$ such that $s(q, z) \geq s$

For objects in $[1..M]^d$

The idea is to represent each coordinate in unary form

- For example, if $M = 10$ and $d = 2$, then $(5, 2)$ becomes $(1111100000, 1100000000)$
- In this case, we have that the L_1 distance of two points in $[1..M]^d$ is

$$d(x, y) = \sum_{i=1}^{d} |x_i - y_i| = \sum_{i=1}^{d} d_{Hamming}(u(x), u(y))$$

  so we can concatenate vectors in each coordinate into one single dM bit-vector
- In fact, one does not need to *store* these vectors, they can be computed on-the-fly

Note: all this is just one of the ideas for LSH. There are others (minhash, simhash, ...) but the course is finite.

Generalizing the idea...

- If we have a family of hash functions such that for all pairs of objects x, y

$$P[h(x) = h(y)] = s(x, y) \tag{1}$$

- We can then amplify the gap of probabilities by stacking k functions and repeating m times
- .. and so the core of the problem becomes to find a similarity function s and hash family satisfying (1)

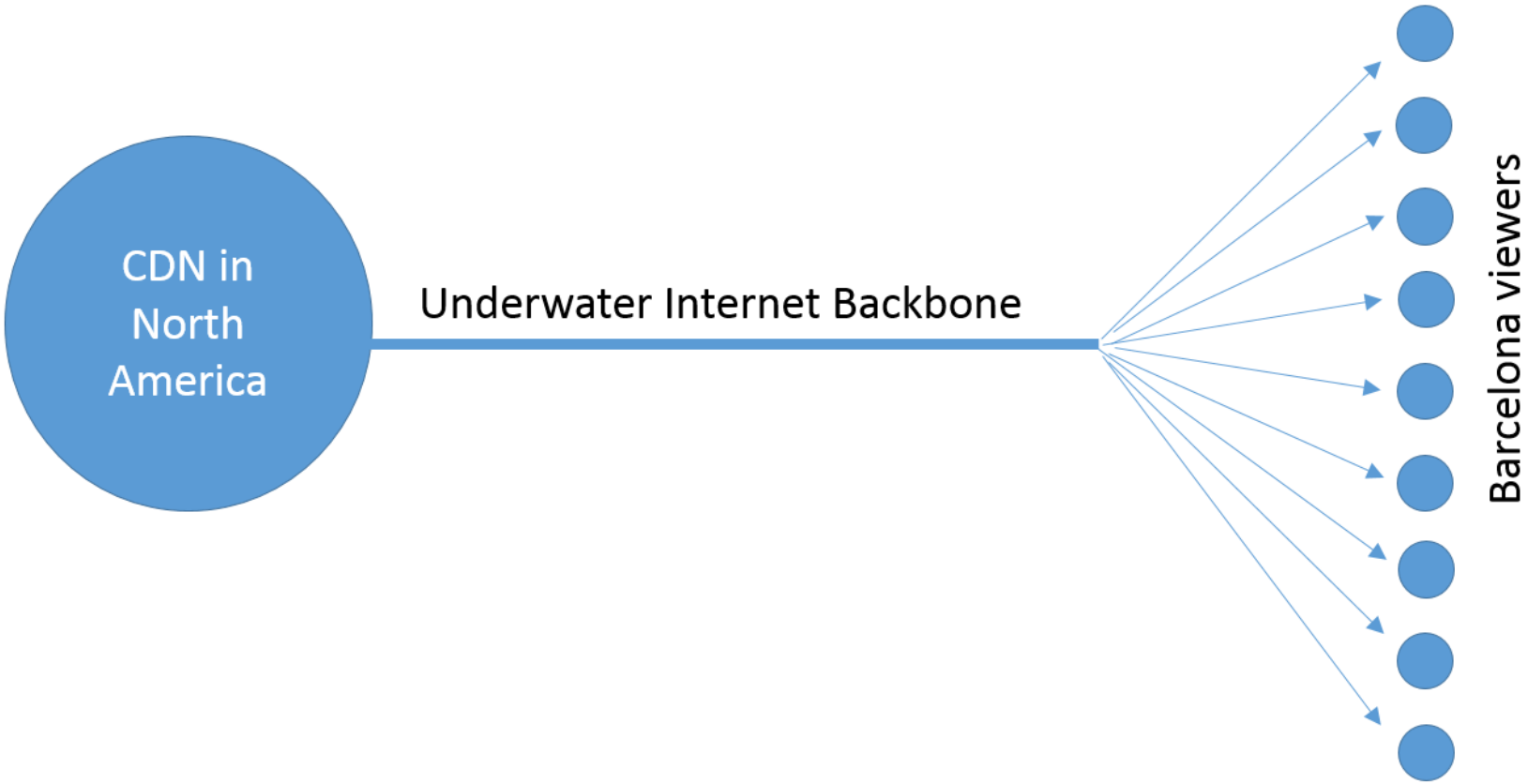
Consistent hashing

[Karger, Lehman, Leighton, Panigrahy, Levine, Lewin (1997)]

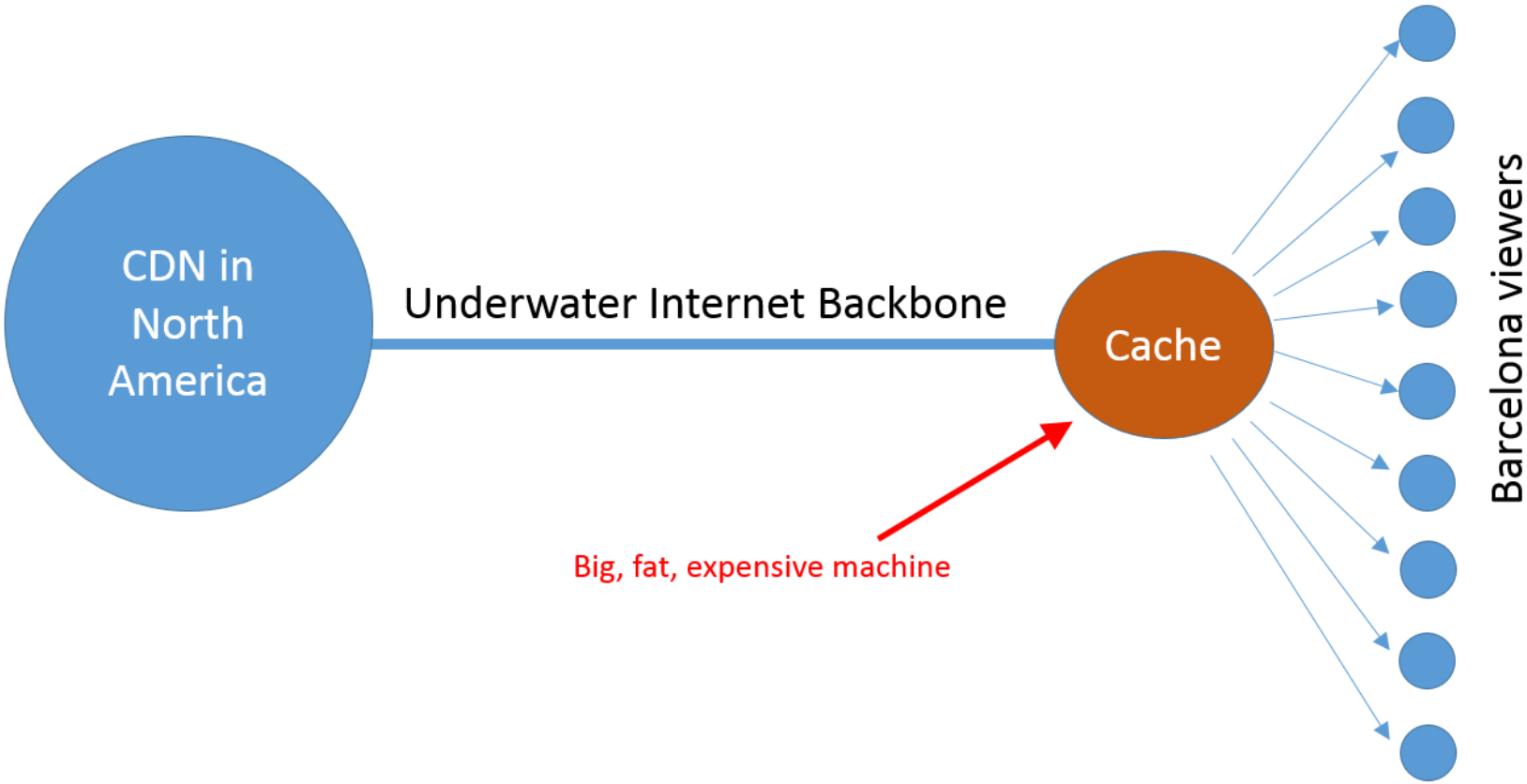
Motivation:

- Big Content Distributor
- Sends content to many users via network
- Needs caches near their users, for latency

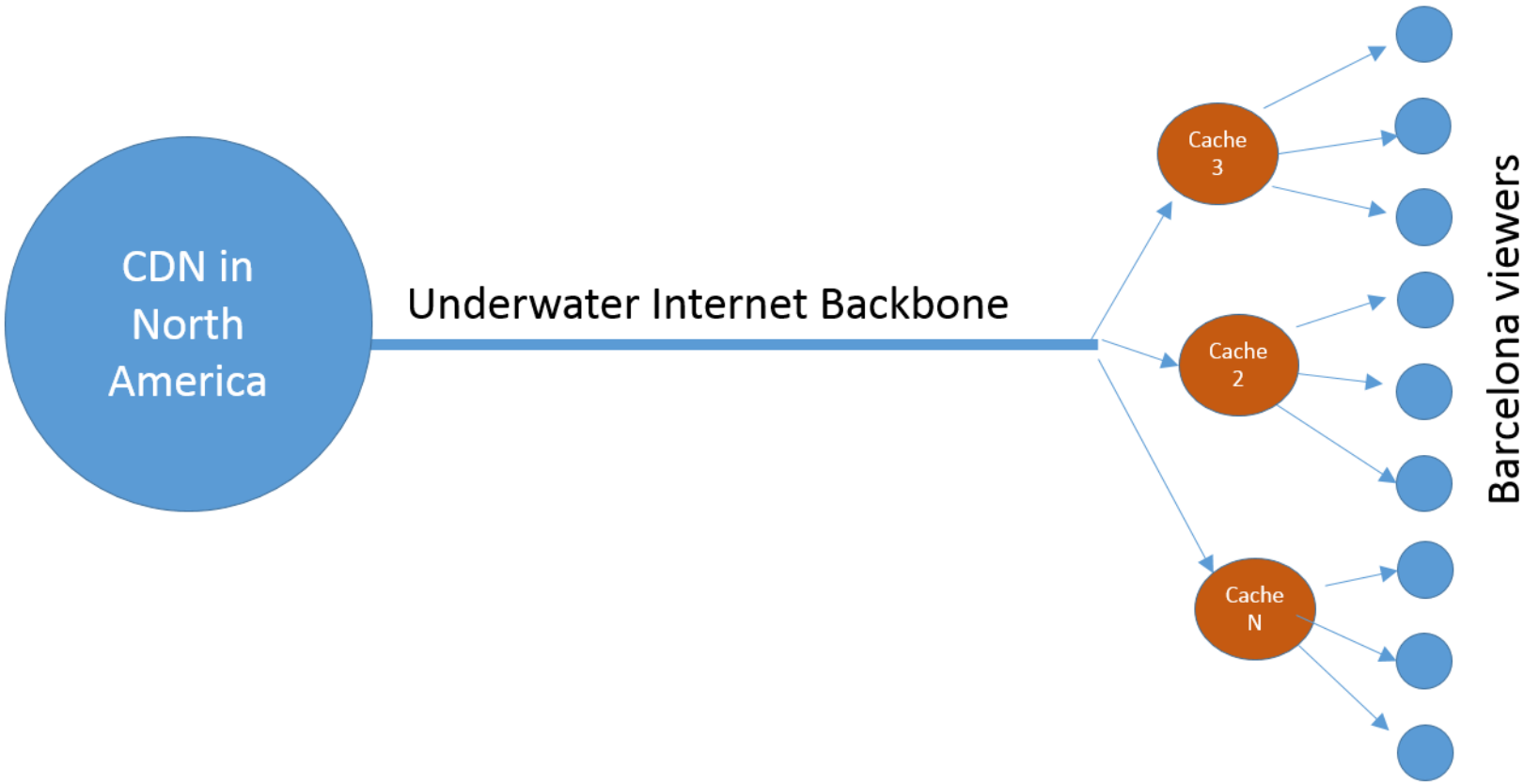
Consistent hashing



Consistent hashing



Consistent hashing



h that hashes URL's to $\{1 \ldots N\}$ ($N =$ number of caches)
Page w is stored in cache $h(w)$
User knows h

The problem: Adding caches

Traffic grows and you need to add new caches. Say 1 new.

Current hash function $h(x) = x \bmod N$
New hash function $h_{new}(x) = x \bmod (N + 1)$

Almost every page needs to be moved to another cache!!!

(plus all users need to be notified of new hash function)

Consistent hashing

Two hash functions:

- h_S: maps servers to the unit circle C
- h_I: maps items to the unit circle C

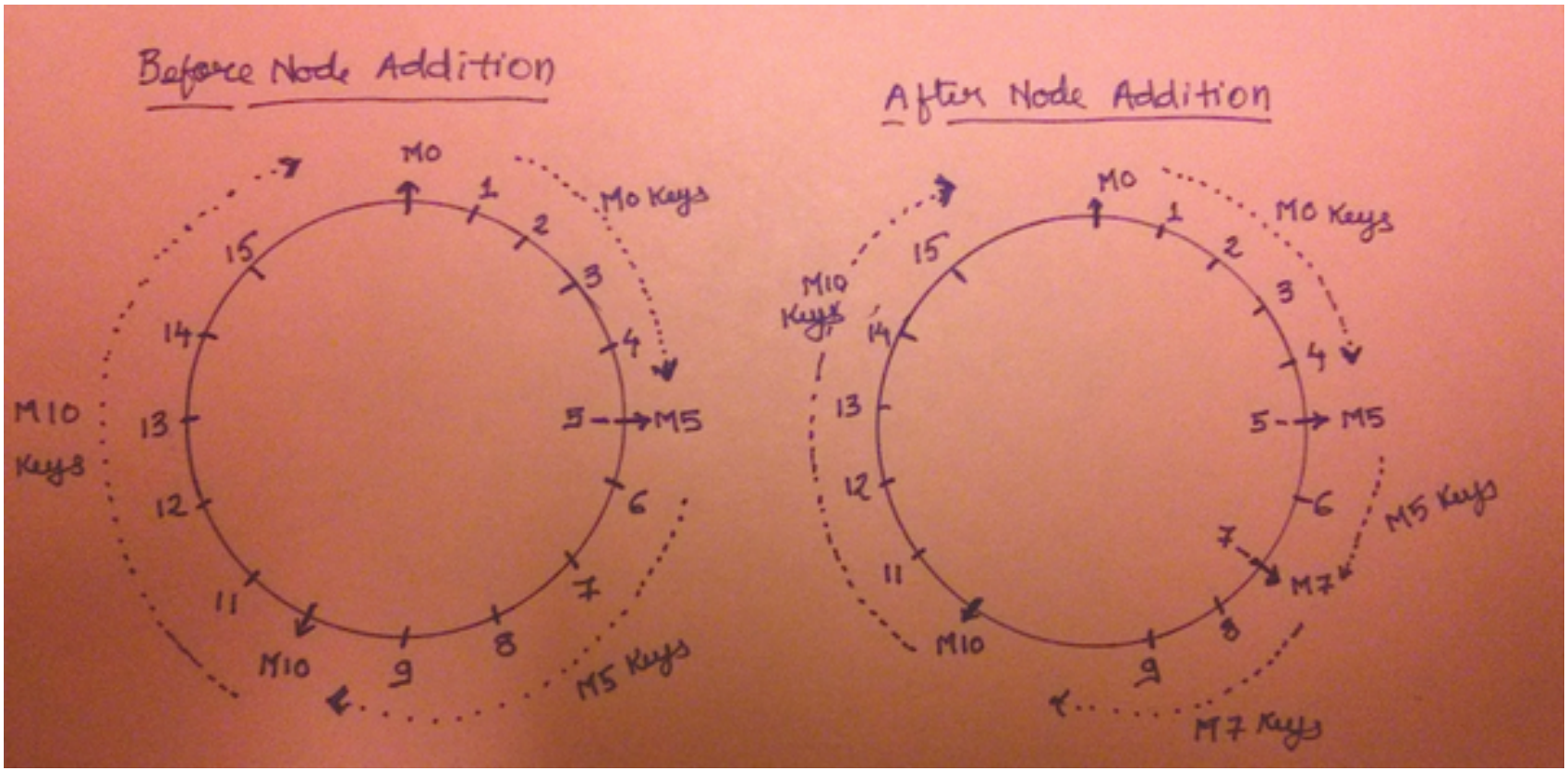
The Consistent hashing rule:

To find the server for item i:

- Jump to $h_I(i)$ in the circle
- Walk back in the circle till you find a value $h_S(s)$
- Item i is in server s

Consistent hashing

[From https://www.quora.com/q/ebzrgpkirtzpthyb/Distributed-Systems-Part-1-A-peek-into-consistent-hashing]



Consistent hashing: Details

- "Walk back" is a metaphor.
- N servers split the circle in N intervals.
- Balanced binary tree to find the interval where an item falls.
- Adding or removing nodes = operations on tree.
- Time = $O(depth)$ = $O(\log N)$.
- C can be discretized to reasonable precision (omitted)

Consistent hashing: Guarantees

Theorem.
Let L_s be the load of server $s \in 1 \ldots N$ = the number of pages assigned to s at any given moment. Then

1. $E[L_s] =$ #items$/N$.
2. Var$[L_s] \leq \ldots$
3. For any δ, $L_s \leq$ E$[L_s] + \ldots$ with prob $> 1 - \delta$

2 works assuming $O(N)$-wise independence of the hash functions
3 uses 2.
We will see more details on these kind of claims later on.

Reducing variance and unbalance

- Choose k hash functions independently $h_{S,1}, \ldots, h_{S,k}$.
- Map server s to $h_{S,1}(s), \ldots, h_{S,k}(s)$.
- Now there are kN virtual servers.
- Probability of too-large loads is smaller.

Replication: hotspots, safety & speed

This still does not solve a major problem

What if a page i becomes very very popular?

The server responsible for i becomes a hotspots

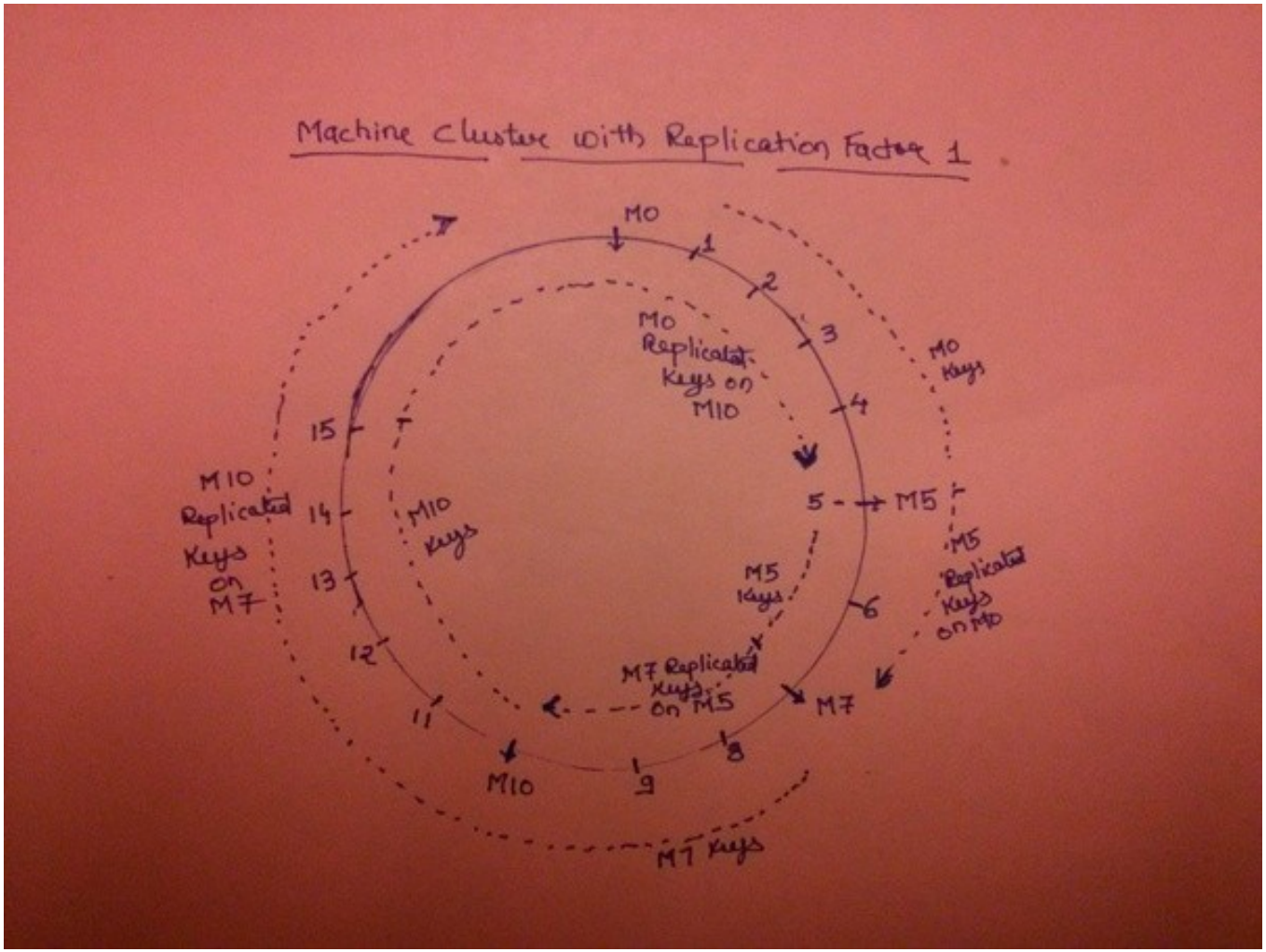
Help: Replicate

Impleemntation using Random Trees (omitted)

Replication: hotspots, safety & speed

[From `https://www.quora.com/q/ebzrgpkirtzpthyb/Distributed-Systems-Part-1-A-peek-into-consistent-hashing`]

Map i to the 2, 3, . . . , k, previous servers in the circle



Conclusion

- Secret of Akamai (late90's).
- Still routes 15% to 30% of all Web traffic.
- Used in Cassandra and many other distributed DB's.