U LISBOA | UNIVERSIDADE DE LISBOA

Ciências ULisboa

LASIGE

Introducing (Probabilistic) Data Structures for Big Data: Bloom Filter and LSH

Vinicius Vielmo Cogo
http://lasige.di.fc.ul.pt/~vielmo
vielmo@lasige.di.fc.ul.pt

UFSM – June 21, 2017

Outline

Big Data

Bloom Filter

Locality-Sensitive Hashing

Outline

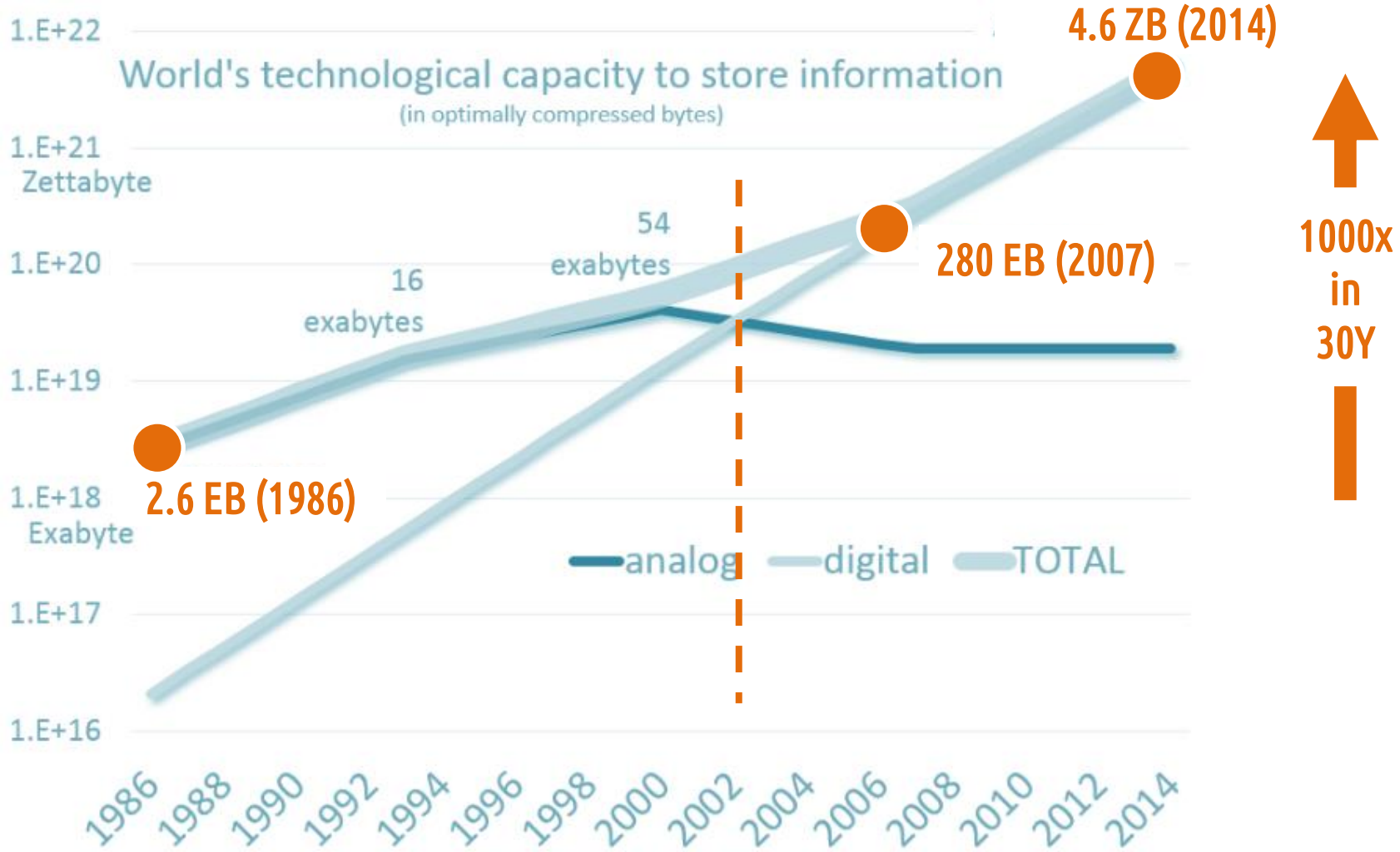
Big Data

Bloom Filter

Locality-Sensitive Hashing

World's Data Capacity

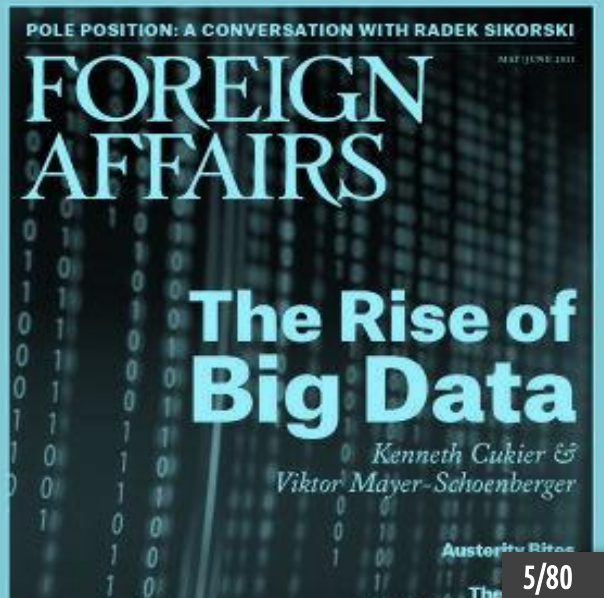
BD



Source: www.martinhilbert.net/worldinfocapacity

Trending Topic

BD



Characteristics

5 Vs

- **Volume** (quantity or size)
- **Variety** (type and nature)
- **Velocity** (speed it is generated)
- **Variability** (inconsistency in datasets)
- **Veracity** (quality)

Big Data …

BD

- … is **relative**
- … is **not defined** by a specific number of TB, PB, EB
- … is when it becomes **big for you**
- … is when **your solutions** become **inefficient**
- … is when **traditional processing** becomes **impractical**

Data Structures and Big Data

- **Traditional DSs** are subject to the **same problems**
  ↳ e.g., lists, trees

- Requires …

  … **distributing your data** (e.g., YARN, Spark)
  or
  … **using auxiliary data structures** (e.g., index, metadata)
  or
  … **trading precision for feasibility and utility**
  (e.g., probabilistic approach)

Probabilistic Data Structures

… trading precision for feasibility and utility

Precision:

- **Rounded values** (e.g., float with reduced precision)
- **Ranges** instead of **values** (e.g., location, ages)
- **False positives** (e.g., system incorrectly triggers something)

Recall:

- **False negatives** (e.g., system fails to find any solution)
- **Similar objects** instead of the **nearest** (e.g., approximate solutions)
- **Good path** instead of the **optimal** (e.g., suboptimal)

Outline

Big Data

Bloom Filter

Locality-Sensitive Hashing

~~Gentle~~ Introduction

SPOILER ALERT!

Bloom Filter

BF is an efficient data structure
to **test if an object belongs to a collection**

Membership Test

BF

Does my collection contain this element?

Algorithm

BF

- The collection …

City
Coimbra
Leiria

Algorithm

- The Bloom filter …

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
bf[i]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

http://billmill.org/bloomfilter-tutorial/

Algorithm – Insert

- Adding elements to the Bloom filter …

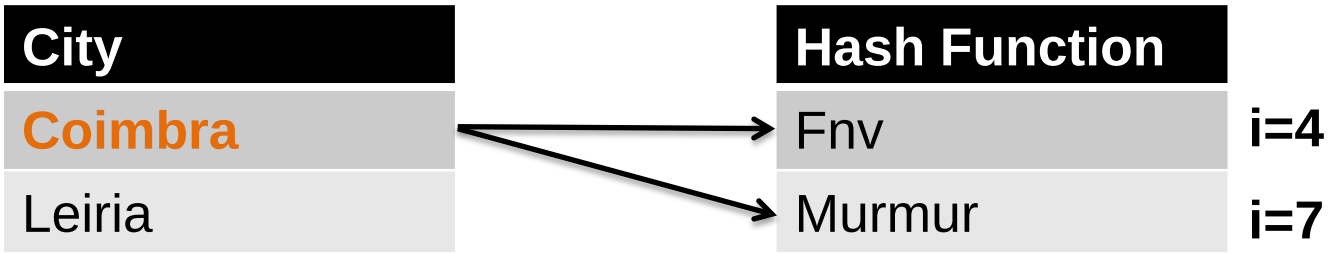
City
Coimbra
Leiria

Hash Function
Fnv
Murmur

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
bf[i]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Algorithm – Insert

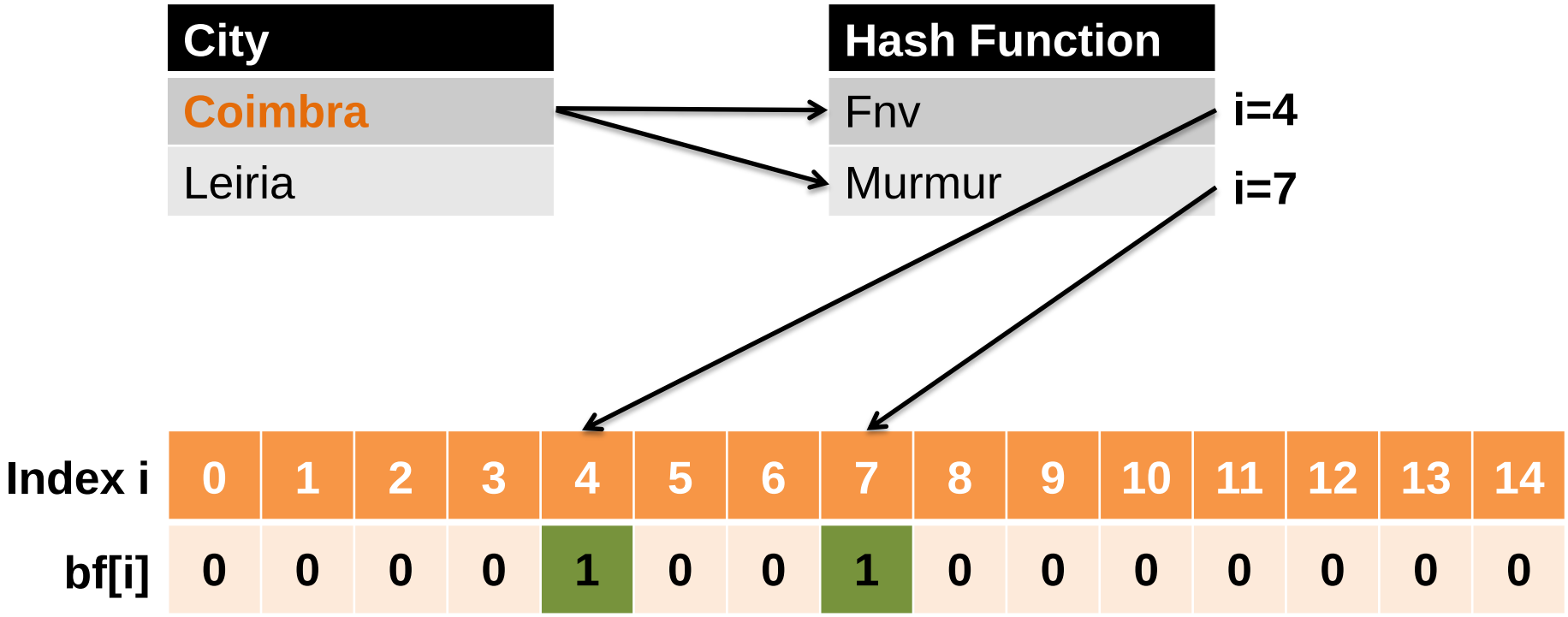
- Adding elements to the Bloom filter …



Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
bf[i]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Algorithm – Insert

- Adding elements to the Bloom filter …



Algorithm – Insert

- Adding elements to the Bloom filter …

City
Coimbra
Leiria

Hash Function
Fnv
Murmur

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
bf[i]	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0

Algorithm – Insert

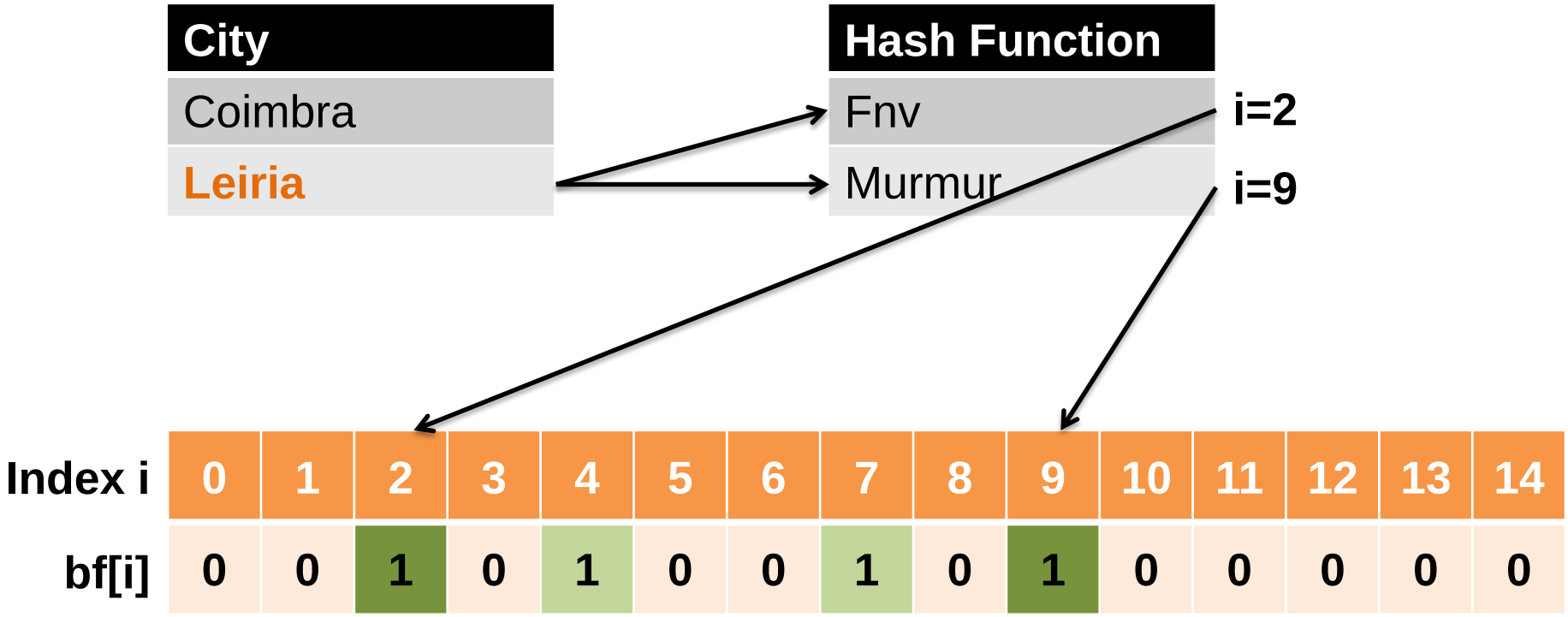
- Adding elements to the Bloom filter …

City	Hash Function	
Coimbra	Fnv	i=2
Leiria	Murmur	i=9

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
bf[i]	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0

Algorithm – Insert

- Adding elements to the Bloom filter …



Algorithm – Insert

- Adding elements to the Bloom filter …

City
Coimbra
Leiria

Hash Function
Fnv
Murmur

Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
bf[i]	0	0	1	0	1	0	0	1	0	1	0	0	0	0	0

Algorithm – Test

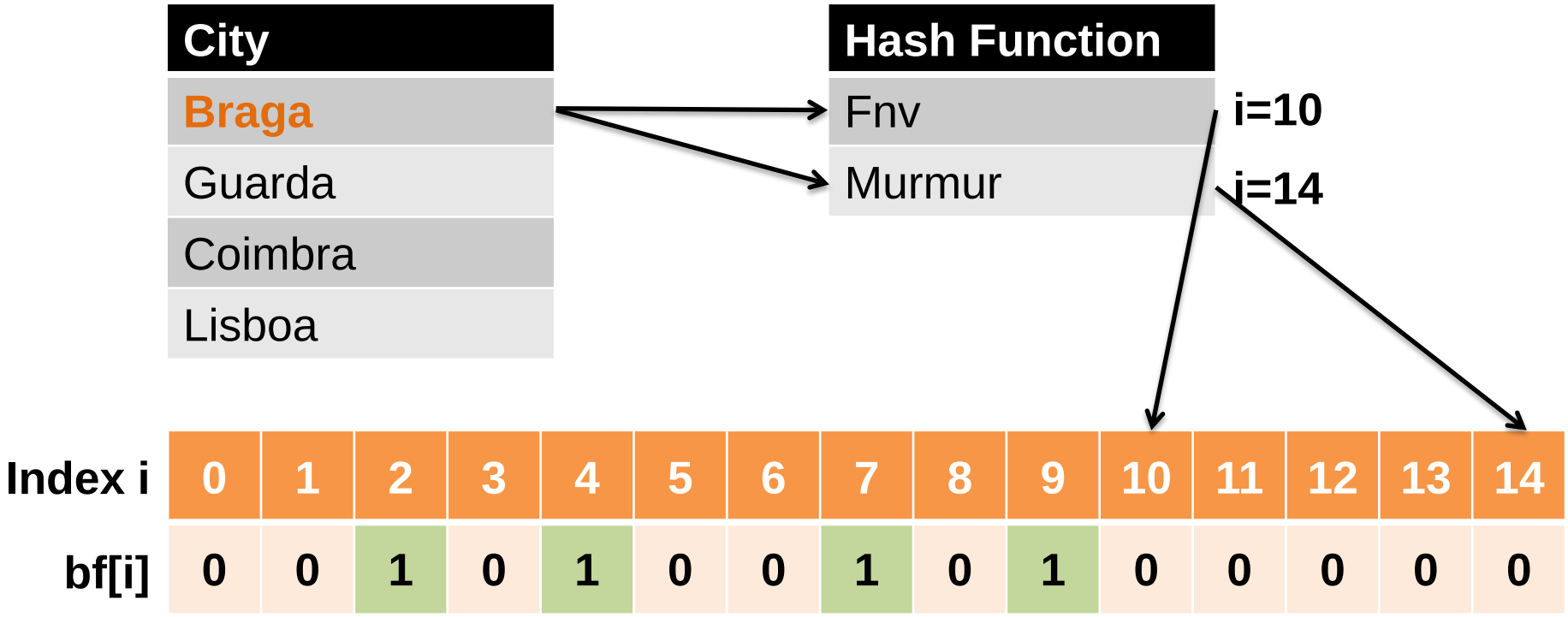
BF

- Testing if my collection contains some elements …

City
Braga
Guarda
Coimbra
Lisboa

Algorithm – Test

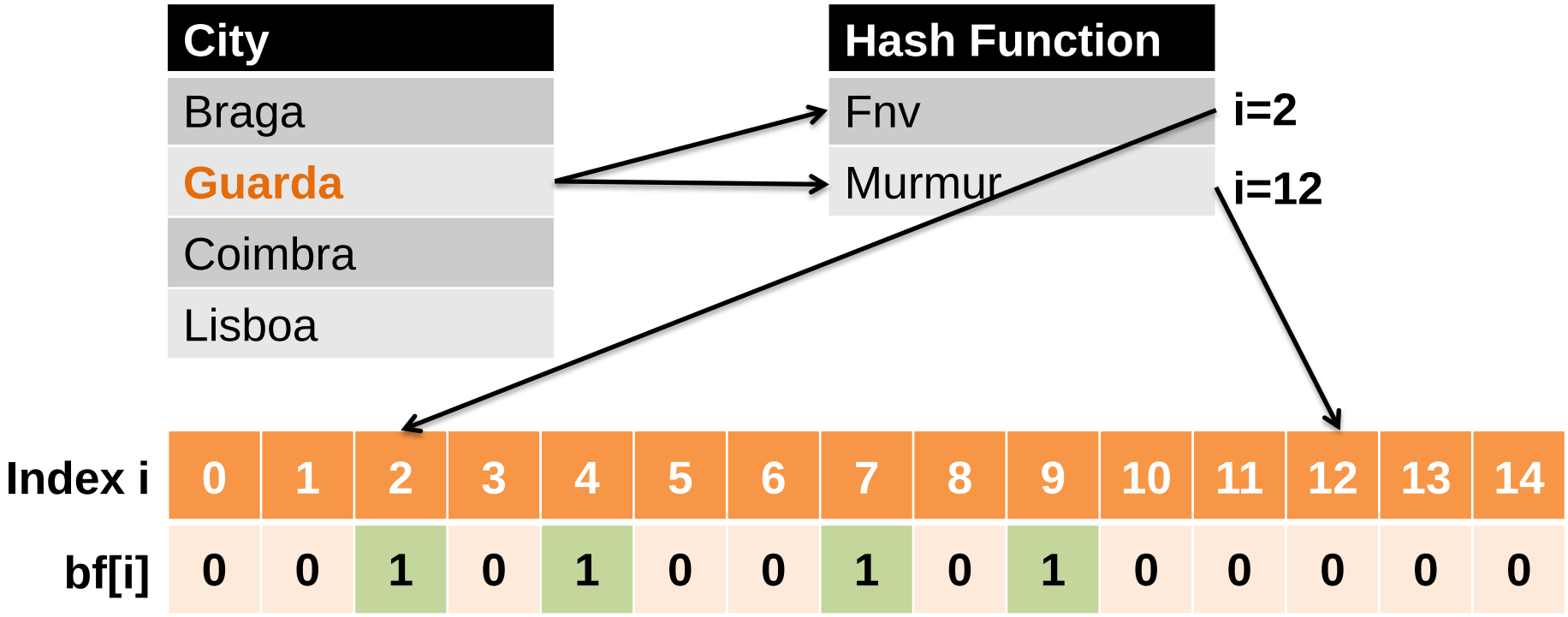
- Testing if my collection contains some elements …



Result: false

Algorithm – Test

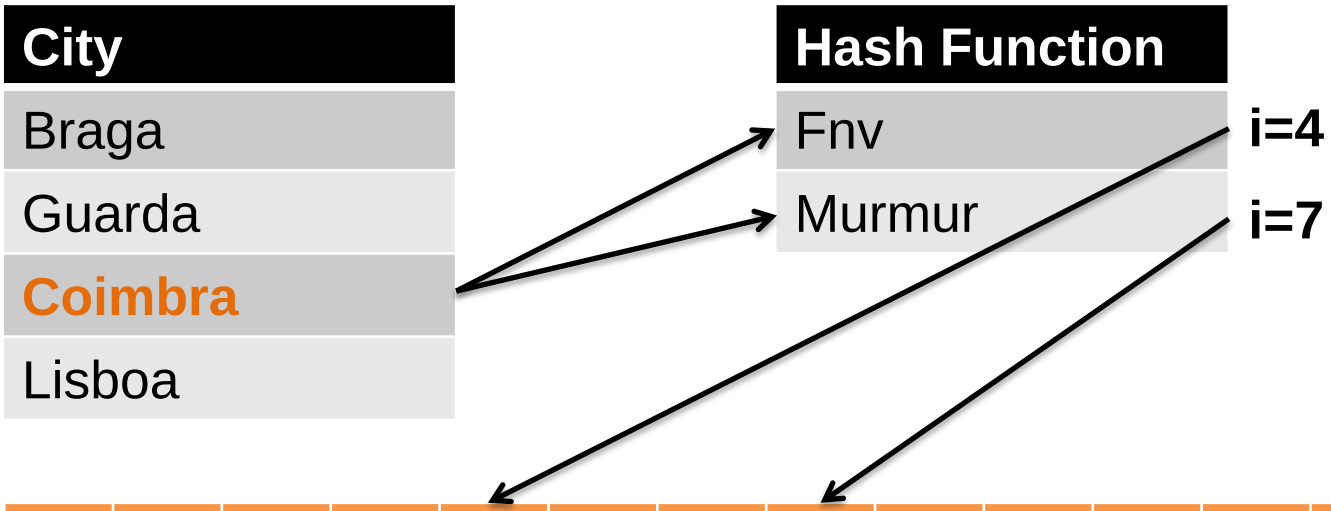
- Testing if my collection contains some elements …



Result: false

Algorithm – Test

- Testing if my collection contains some elements …

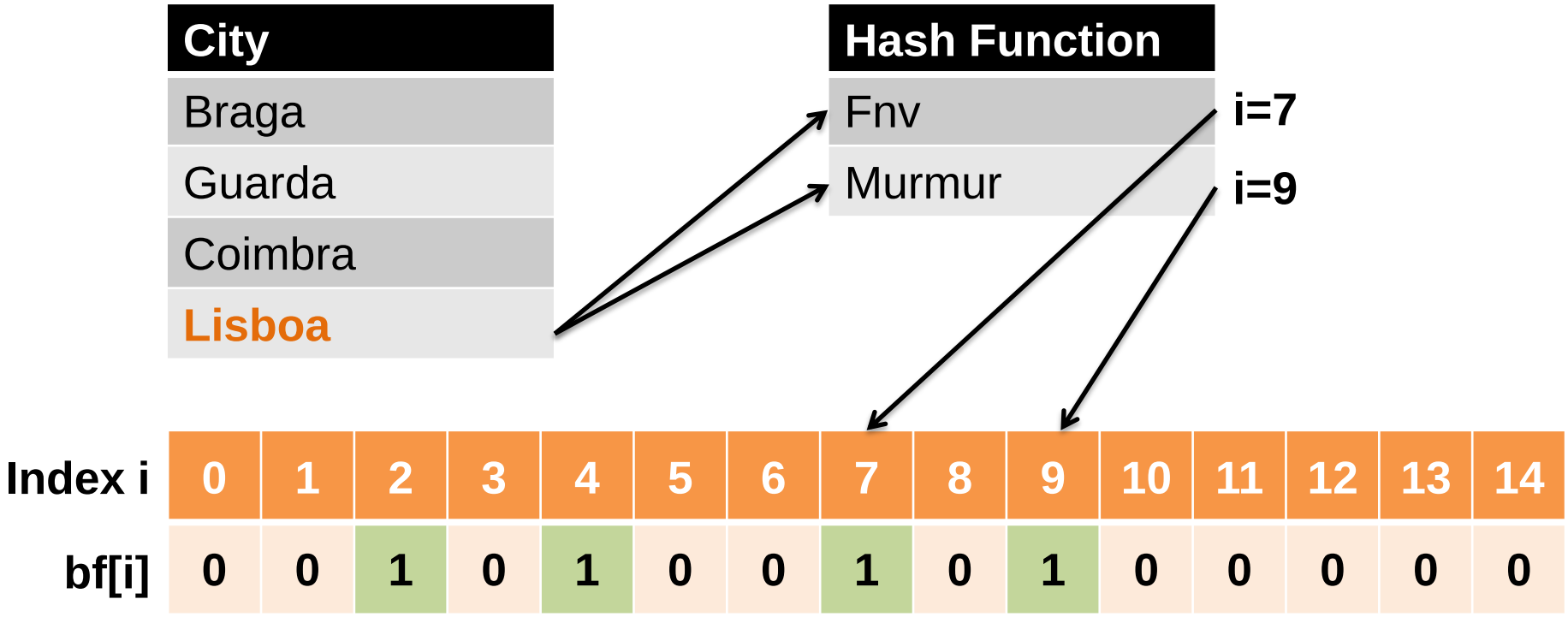


Index i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
bf[i]	0	0	1	0	1	0	0	1	0	1	0	0	0	0	0

Result: true

Algorithm – Test

- Testing if my collection contains some elements …



Result: true (but it is a false positive)

Bloom Filter

BF

- Proposed by Burton Howard **Bloom** in **1970**
- **Design principles**
  - **Space-efficient**
    - Smaller than the original dataset
    - Doesn't depend on object sizes
  - **Time-efficient**
    - Low latency R/W
    - $O(k)$, which is much smaller than $O(n)$
  - **High throughput**
  - **Probabilistic**
    - E.g., *myCollection.**might**Contain(myObject)*
    - **False positives** happen (but in a **configurable** way)
    - **False negatives don't** happen

Bloom Filter

BF

- Important variables

n = Expected collection size →

City
Coimbra
Leiria

p = False positive rate (e.g., 0.0001% or 1 in 1M)

m = Bitmap size →

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

k = Optimal number of hash functions →

Hash Function
Fnv
Murmur

Bloom Filter

- Important variables

$$n$$

$$p = \left(1 - e^{-kn/m}\right)^k$$

$$m = -\left(\frac{n \ln p}{(\ln 2)^2}\right)$$

$$k = \frac{m}{n} \ln 2$$

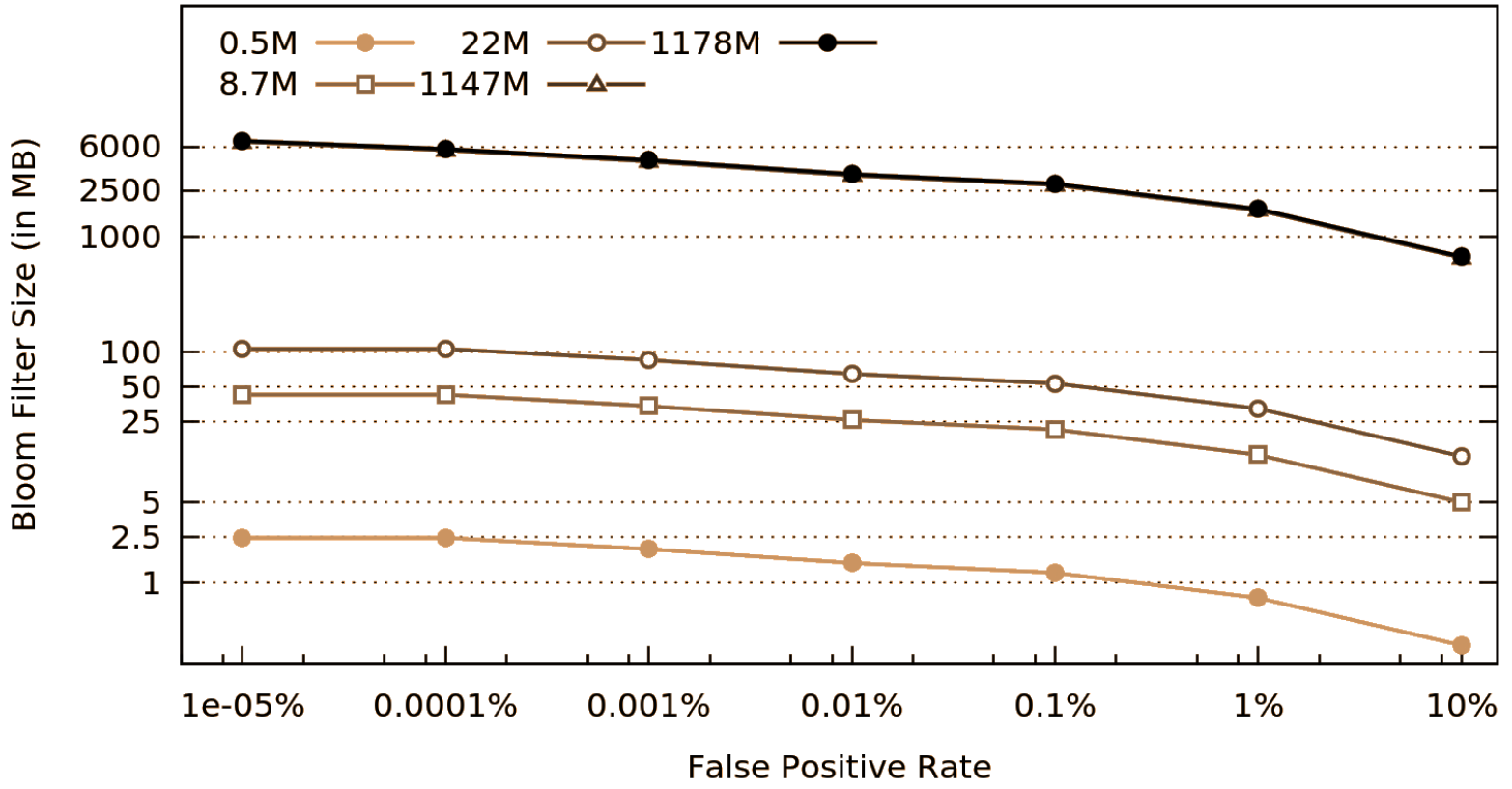
Bloom Filter

- **Users define two** of them (normally ***n*** and any other)
- The **other two are calculated** with those equations
- Interesting relations:
  - Bigger collection **(↑n)** → Larger bitmap **(↑m)**
  - Bigger collection **(↑n)** → More false positives **(↑p)**
  - Larger bitmap **(↑m)** → Less false positives **(↓p)**
  - Larger bitmap **(↑m)** → Less hash functions **(↓k)**
  - Less hash functions **(↓k)** → Faster operations

Bloom Filter

BF

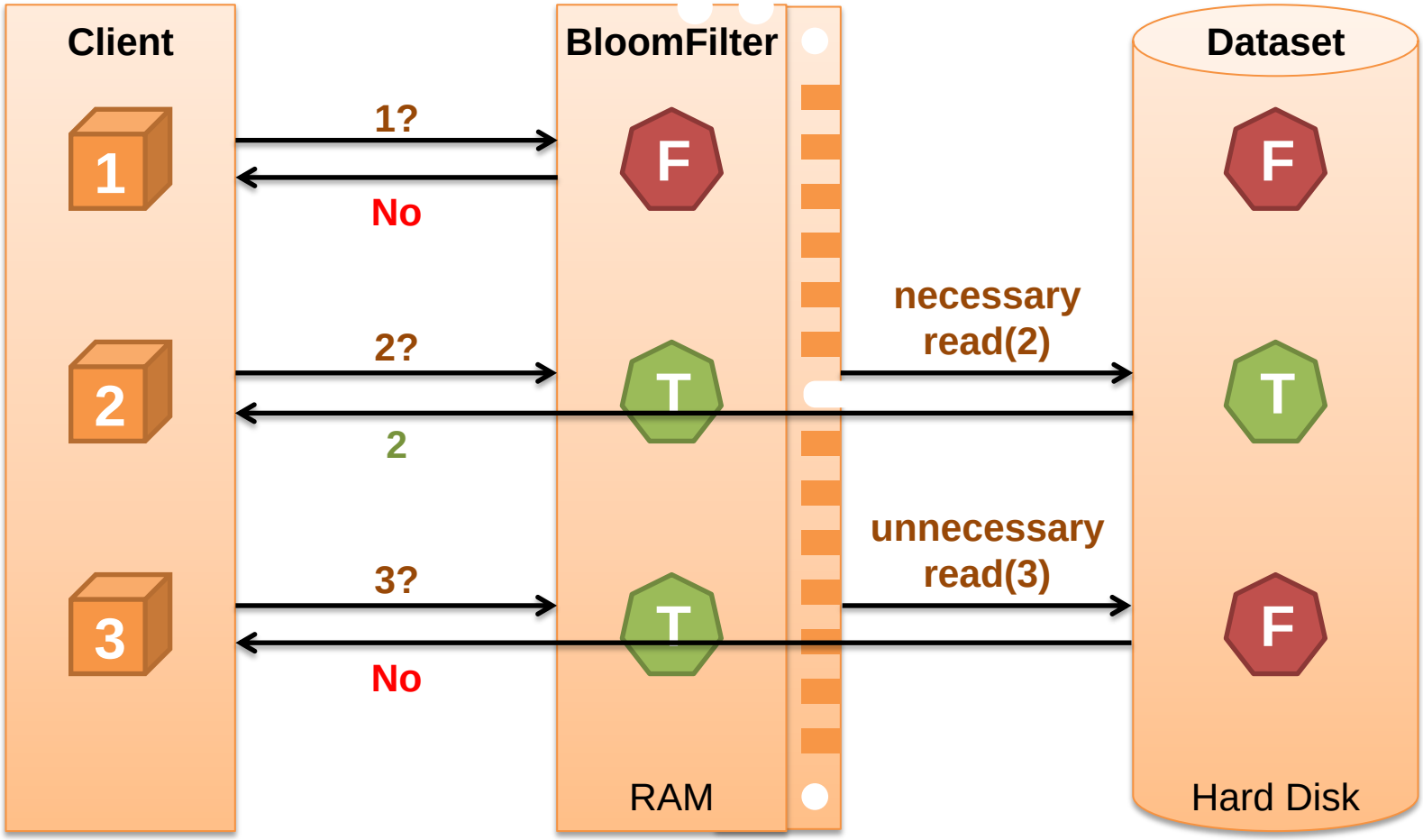
- Bloom filter size vs. False positive rate



Use Cases

BF

- Reducing unnecessary disk reads



Use Cases

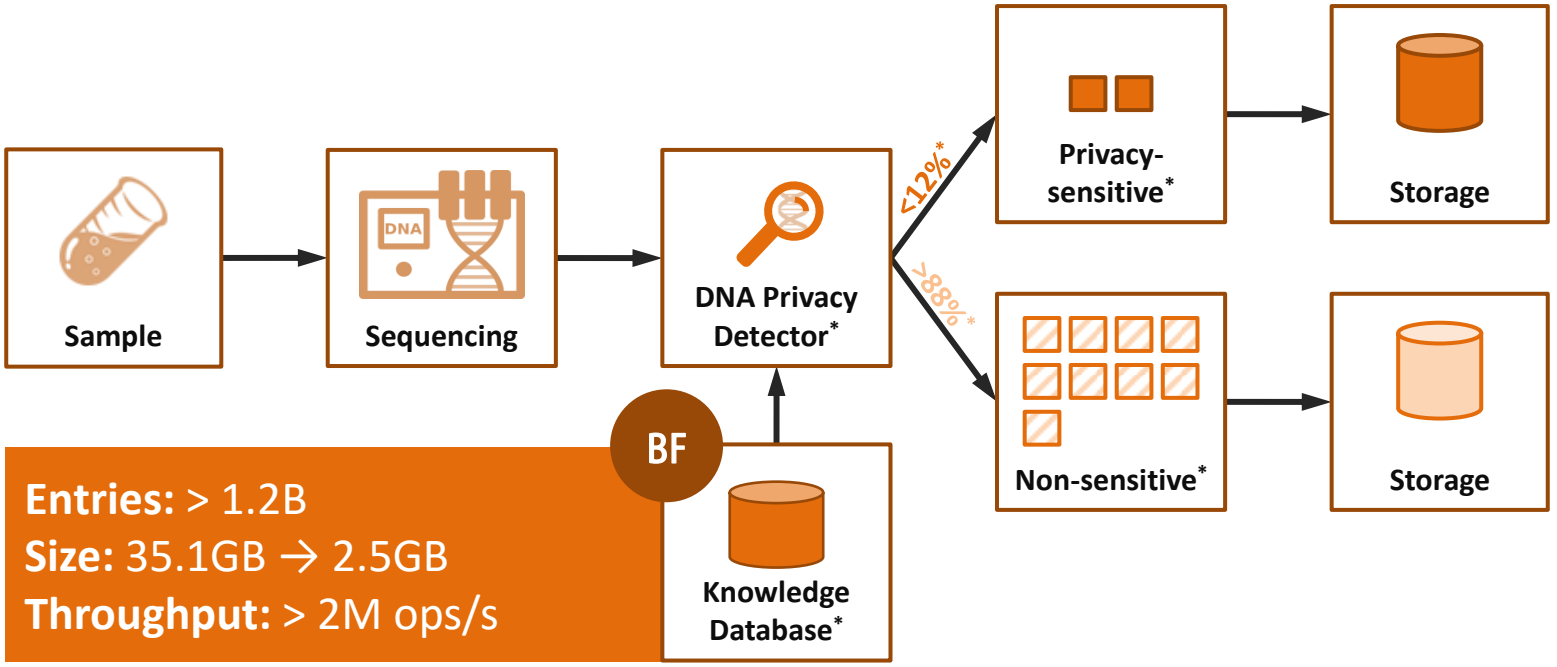
BF

- **Google BigTable, Apache Cassandra and HBase**
  - Reducing disk lookups
- **Google Chrome**
  - Lookup a list of known malicious URLs
- **Bitcoin**
  - Get only the transactions relevant to your wallet

BF in my PhD

BF

- Detecting privacy-sensitive genomic data



* A High-Throughput Method to Detect Privacy-Sensitive Human Genomic Data @ WPES 2015

Available Implementations

BF

- **Google's Guava-libraries**

https://code.google.com/p/guava-libraries/

- **Orestes-Bloomfilter**

https://github.com/Baqend/Orestes-Bloomfilter

- **java-bloomfilter**

https://github.com/magnuss/java-bloomfilter

- **java-longfastbloomfilter**

https://code.google.com/p/java-longfastbloomfilter/

Other Filters

- **Counting Bloom filters**

Allow deletions (use a 4-bit counter instead of 1 bit)

- **Buffered Bloom filters**

Sub-filters in SSD with buffered R/W exploring bit locality

- **Quotient and Cascade filters**

Uses an SSD, instead of the main memory, for scalability

Outline

Big Data

Bloom Filter

Locality-Sensitive Hashing

Locality **S**ensitive **H**ashing

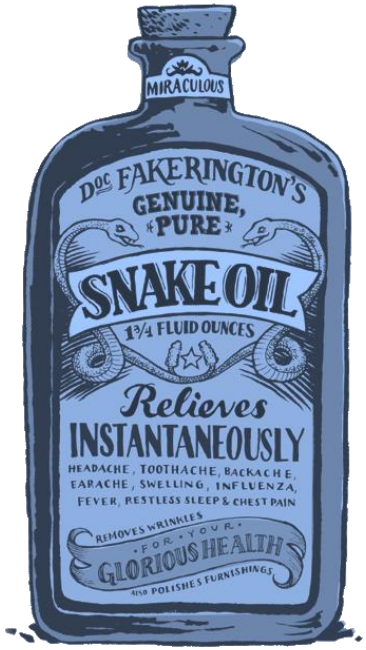
SPOILER ALERT!

LSH is an efficient algorithm
to **find similar objects using hashes**

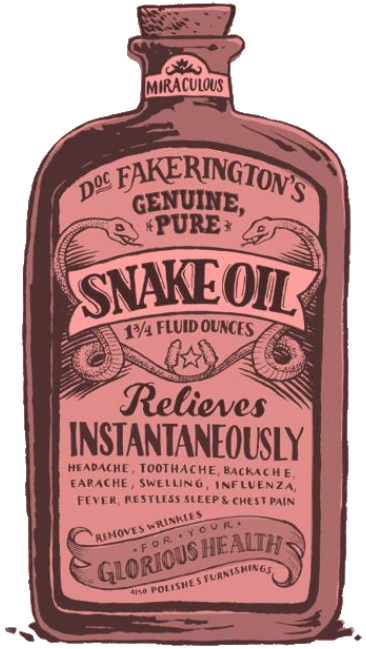
Recommendation Algorithms

Suggest something to users …

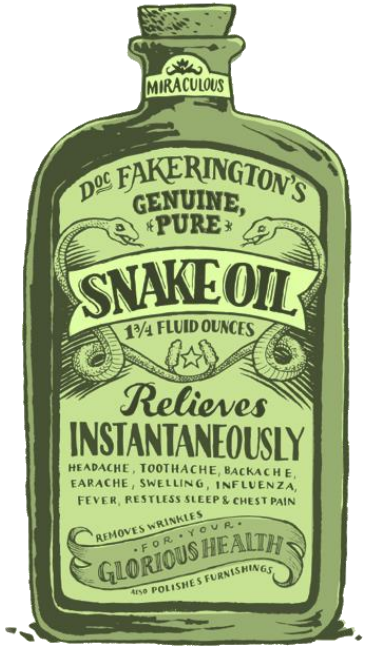
Customers who bought this object also bought …



★★★★★ 50€



★★★★ 35€



★★★ 20€

Recommendation Algorithms

LSH

→ … people you may know

→ … people you may like

→ … videos you may like

→ … movies you may like

→ … music you may like

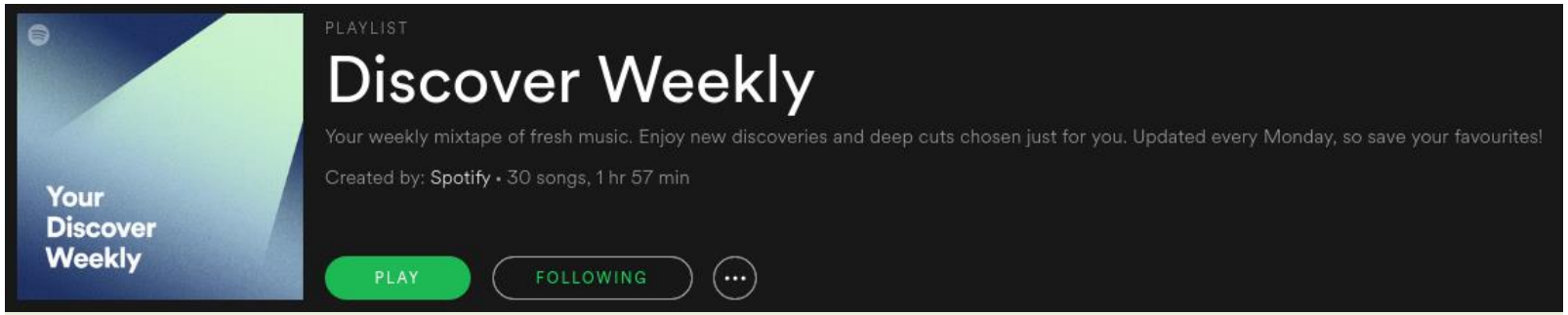
→ … products you may like

Recommendation Algorithms

How do they work?

Recommendation Algorithms

LSH



Task: Suggest 30 tracks per week for each user

Recommendation Algorithms

General
Recommendations
(e.g., popular songs)

≠

Personalized
recommendations

Example: Music Recommendation

1) Find **similar users** (with similar preferences)

Comparing the list of things they like

2) Suggest what one likes and the other doesn't know yet

Example: Music Recommendation

LSH

P1

P2

P3

Example: Music Recommendation

P1

P2

P3

Example: Music Recommendation

P1

P2

Example: Music Recommendation

P1

P2

Example: Music Recommendation

LSH

P1 

P2 

P3 

How to identify similar users?

Example: Music Recommendation

P1 

P2 

P3 

Jaccard Distance*

$$\text{Similarity} = \frac{|A \cap B|}{|A \cup B|}$$

Example: Music Recommendation

P1

$$\frac{|P1 \cap P2|}{|P1 \cup P2|} = \frac{4}{6} = 0.667$$

P2

$$\frac{|P2 \cap P3|}{|P2 \cup P3|} = \frac{1}{8} = 0.125$$

P3

$$\frac{|P1 \cap P3|}{|P1 \cup P3|} = \frac{1}{8} = 0.125$$

Example: Music Recommendation

P1

$$\frac{|P1 \cap P2|}{|P1 \cup P2|} = \frac{4}{6} = \underline{0.667}$$

P2

$$\frac{|P2 \cap P3|}{|P2 \cup P3|} = \frac{1}{8} = 0.125$$

P3

$$\frac{|P1 \cap P3|}{|P1 \cup P3|} = \frac{1}{8} = 0.125$$

Problem

Millions of users that listen
thousands different **songs** each

Problem

Users
=
Objects

Songs
=
Dimensions*

Millions of users that listen
thousands different **songs** each

Problem

Comparing **60M** users

1/10 s per comparison

Naïve: **60M x 60M** = 3.6 * 10 ^14 s = **11M** years

Smart: **60M x log(60M)** = 46M s = **14** years

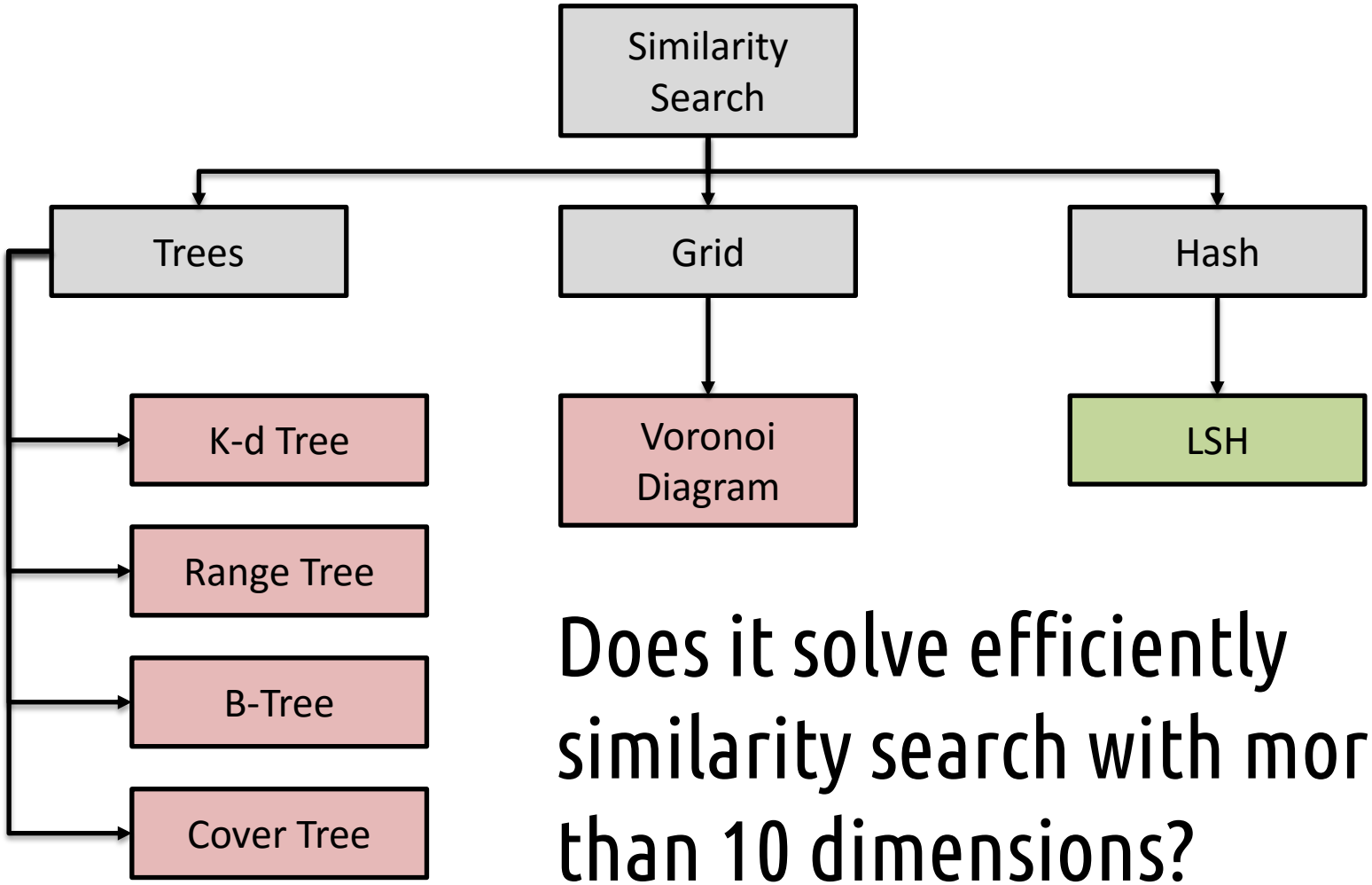
Problem

1000
machines
=
1 week

What if fails?

Being more efficient

Alternative Algorithms



Does it solve efficiently similarity search with more than 10 dimensions?

Locality Sensitive Hashing

NOT A SPOILER ANYMORE!

LSH is an efficient algorithm
to **find similar objects using hashes**

Algorithm

So, imagine **millions** of **buckets**

Algorithm – Insert

Given an object (user): **hash** it, obtain a **value**, and **place** the object in this bucket

Algorithm – Insert

In a way that objects in the **same bucket** have **bigger probability** of being similar

Algorithm – Query

Calculate the **distance** between objects **within** the same **bucket** only

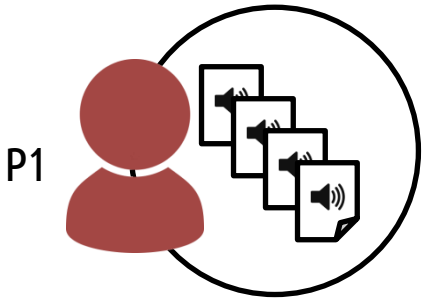
Components

LSH = Hash Function + Hash Tables

MinHash* + MultiMap[]

MinHash

- An array with the minimal hashes from all dimensions for each hash function



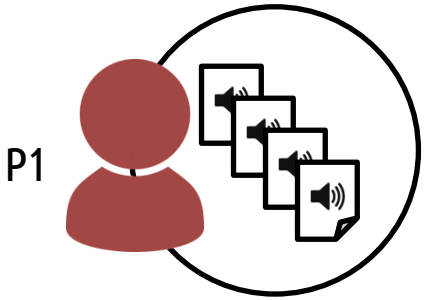
minHash(P1)

23	12	5	45	7	34	12	76	87	...

```
minHash(Object o):
     num_hashes <- 200 //defined based on a similarity threshold
     hashes <- new hash[num_hashes] //200 different hash functions
     minHash <- new int[num_hashes] //MinHash of the object
     for i in 0..hashes: // for each hash function
          for d in object.dimensions: //for each dimension (music)
                    hi <- hashes[i](d) //calculate the hash of d
                    minHash[i] <- min(minHash[i], hi) // store the min
```

- Converts **variable** number of dimensions **to** a **fixed** configurable number
- Using the **same order** of hash functions is important **to find similar** objects

Example



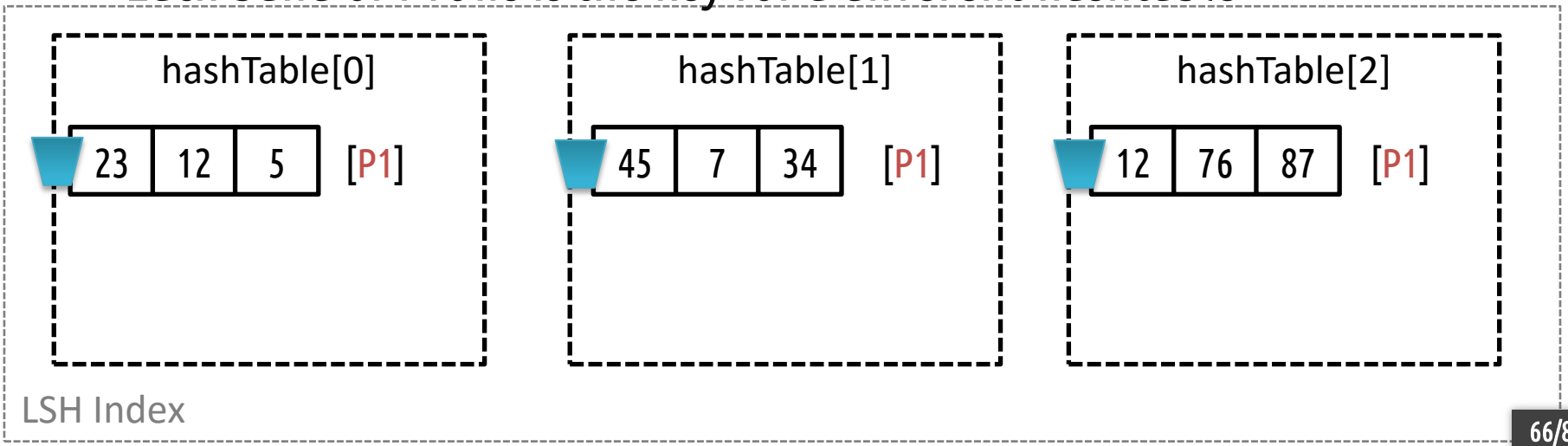
minHash(P1)

23	12	5	45	7	34	12	76	87	…

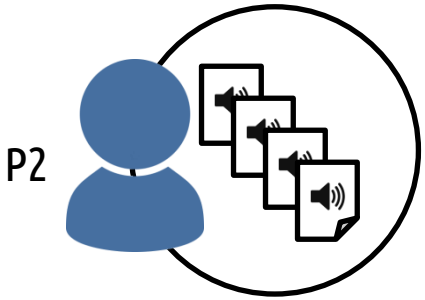
break it into b bands and r rows (based also on the desired similarity threshold)

23	12	5		45	7	34		12	76	87		…

Each band of r rows is the key for a different hashtable



Example



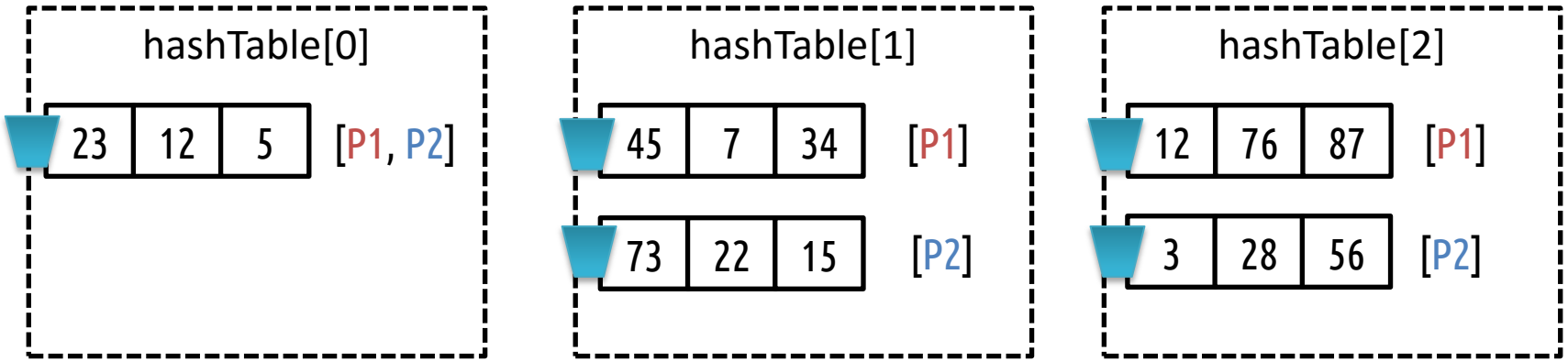
minHash(P2)

23	12	5	73	22	15	3	28	56	…

break it into b bands and r rows (based also on the desired similarity threshold)

23	12	5		73	22	15		3	28	56		…

Each band of r rows is the key for a different hashtable

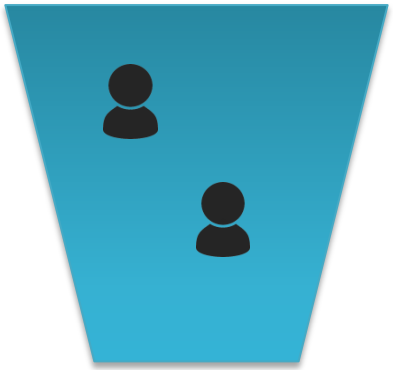


Interfaces


```
distance(Object o1, Object o2)
insert(Object o)
Queries (similarity search):
    ★  nearestNeighbor(Object o)
  ★★★  nearNeighbors(Object o, int maxNeighbors)
       clustering(Object o)
```

Nearest neighbor

Near neighbors



Clustering

Who uses LSH for what?

LSH

Detect near-duplicate web pages

Detecting Near-Duplicates for Web Crawling

Google News recommendations

Google News Personalization: Scalable Online Collaborative Filtering

Detect very similar routes

https://spark-summit.org/2016/events/locality-sensitive-hashing-by-spark/

Detect spam and malicious messages for event organizers

https://www.eventbrite.com/engineering/multi-index-locality-sensitive-hashing-for-fun-and-profit/

Clustering People

http://www.freepatentsonline.com/y2015/0213112.html

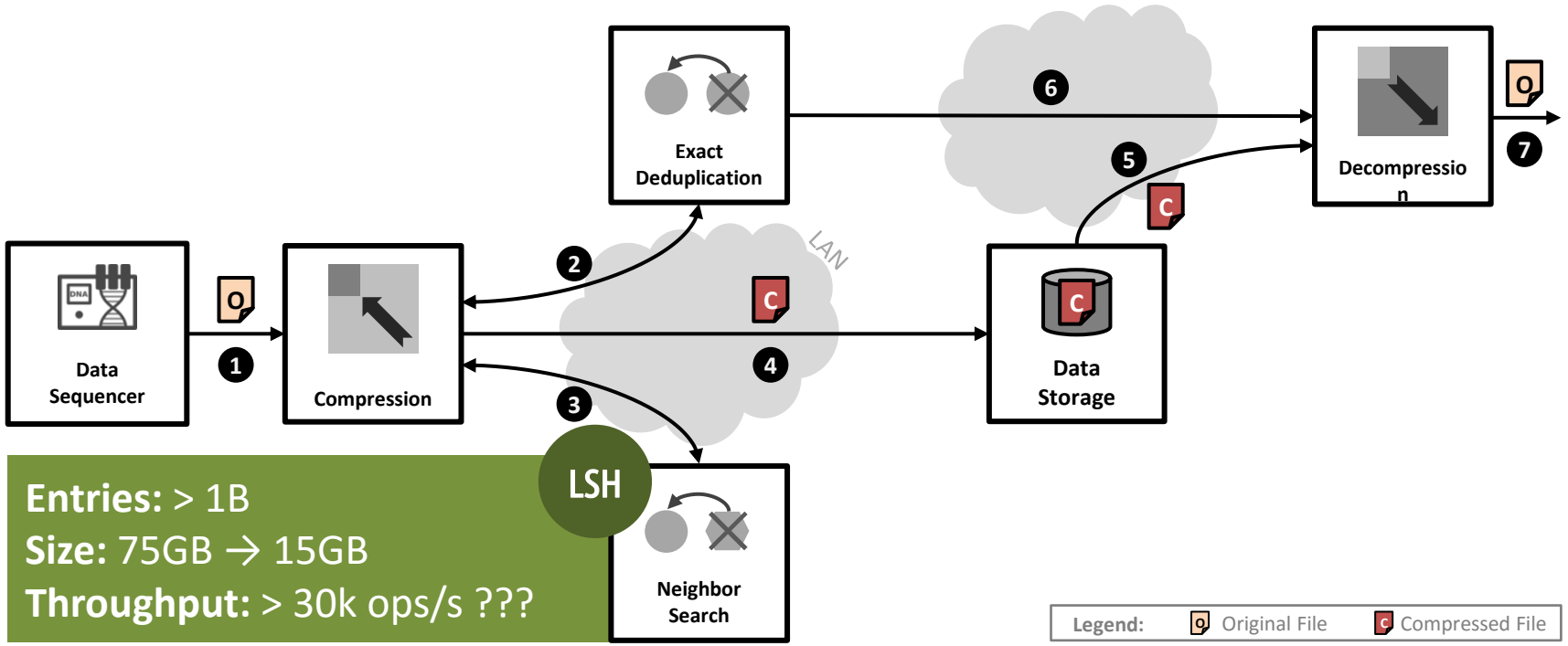
Spotify recommender system

LSH forest - ANNOY

LSH in my PhD

LSH

- Deduplicating similar genomic and quality portions to compress data
- Pointer + modifications



Available implementations

- OpenLSH (https://github.com/singhj/locality-sensitive-hashing)
- Datasketch (https://github.com/ekzhu/datasketch)
- TarsosLSH (https://github.com/JorenSix/TarsosLSH)
- E2LSH (https://github.com/JorenSix/TarsosLSH)

Challenges: Implementing

- Generic to any **object**
- Providing multiple hash function families
  (generic to all **distances**)
- Being **efficient** (space and time)
- **Durability**

Challenges: Scaling up

- **MultiMaps** (1:n)
- **Off-heap** implementation (avoid garbage collection)
- **Bigger** than memory (e.g., using RAM + SSD disk space)
- **Multi-threaded** (fine-grain locks or non-blocking)
- Using **primitives** (avoid space overhead)

Challenges: Scaling out

- Distributing hash tables in several machines
  - hashTable[0] -> s1
  - hashTable[1] -> s2
  - hashTable[2] -> s3
  - hashTable[3] -> s4
  - hashTable[4] -> s5

- Partitioning keys (require to inform hashTable number)
  - Keys [0 – 1,000,000] -> s1 (hashTable[0-4])
  - Keys [1,000,000–2,000,000] -> s2 (hashTable[0-4])
  - Keys [2,000,000–3,000,000] -> s3 (hashTable[0-4])
  - Keys [3,000,000–4,000,000] -> s4 (hashTable[0-4])

Final Remarks

- **Big Data** is **real** and **requires efficient** solutions
- **Probabilistic** algorithms are **feasible** and **useful**
- **Bloom Filter** → membership test
- **Locality-Sensitive Hashing** → similarity search

Final Remarks

- These algorithms are usually **a step** to something bigger
- **What** to do with them?
- **Where** are they **useful?**
- There are opportunities of **enhancements** and **applications** on them

U LISBOA | UNIVERSIDADE DE LISBOA

Ciências ULisboa

LASIGE

Thank You!

Vinicius Vielmo Cogo
http://lasige.di.fc.ul.pt/~vielmo
vielmo@lasige.di.fc.ul.pt

UFSM – June 21, 2017

Objects and Dimensions

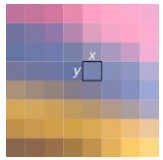
LSH

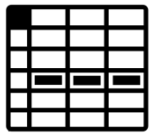
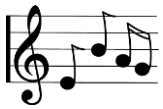
1 Dimension
- Binary values: 0 or 1
- Numbers: age, height, weight, etc.

2 Dimensions
- Cartesian coordinates: (x, y)
- Tuples: (k, v)

3 Dimensions
- 3D coordinates: (x, y, z)
- 2D Animation: (time, x, y)

N Dimensions
- Characters in a string: “abcdefgh”
- Substrings of a string: “abc”, “bcd”, “cde”…
- Bits in a Byte array: 0011 1101
- Words in a sentence: “Foo bar bar foo”
- Sentences in a document
- Pixels in an image
- Notes in a music
- Music in a playlist
- Properties in an object
- Columns in a DB row
- *Minutiae* of fingerprints




Distances and LSH families

Distance	Description	LSH family
Euclidean	Distance between two vectors	Random projections
Jaccard	len(intersection)/len(union)	MinHash
Cosine	Angular distance between vectors	SimHash
Hamming	Number of Substitutions	BitSampling
Levenshtein	Minimal number of substitutions, insertions and deletions	

Some LSH papers

- Similarity Search in High Dimensions via Hashing
- Locality-Preserving Hashing in Multidimensional Spaces
- Approximate Nearest Neighbors: Towards Removing the Curse of dimensionality
- Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions
- Fast Search in Hamming Space with Multi-Index Hashing
- b-Bit Minwise Hashing
- LSH forest: self-tuning indexes for similarity search