Extreme Computing

Joins and Fault Tolerance

Cluster

Theoretically working
Tasks fail occasionally, but (correct) jobs should run

Lab Sizes

Mon 09:00-09:50: 2 showed up, cancelled in future weeks
Mon 10:00-10:50: 29
Tue 14:10-15:00: 34
Wed 10:00-10:50: 12
Wed 14:10-15:00: 22
Thu 09:00-09:50: 9
Thu 11:10-12:00: 11
Fri 11:10-12:00: 13

Joins

How do we combine data sets in MapReduce?

It depends on how big each is...

Old Exam Question

You are provided with a set of interesting words and a large text file. The task is to count how many times each interesting word appears in the text file.

Hash Join

Load set of interesting words into RAM on each mapper:

```python
#!/usr/bin/python3
import sys
interesting = set()
for word in open("interesting.txt"):
  interesting.add(word.strip())

for line in sys.stdin:
  for word in line.split():
    if word in interesting:
      print(word + "\t1")
```

Hash Join Efficiency

- ✓ Limits traffic to reducers
- ✓ Fast
- ✗ Table needs to fit in RAM
- ✗ Table copied to all mappers

Good plan for joining small data with large data

Hash Join Efficiency

- ✓ Limits traffic to reducers
- ✓ Fast
- ✗ Table needs to fit in RAM
- ✗ Table copied to all mappers

Good plan for joining small data with large data

Can also query over the network. . . more later

Sorted Join in Mapper

Which words are used more in YouTube comments than Yahoo answers?

We already ran word count on each *with the same sorting and partitioning*.

YouTube	
a	10
video	3

Yahoo	
a	15
what	4

Mapper

YouTube	
cats	90
lolz	3

Yahoo	
cats	2
lolz	9

Mapper

Sorted Join in Mapper: Efficiency

- ✓ Fast (faster than hash join)
- ✓ Large data
- ✓ Limits traffic to reducers (or no reducers)
- ✗ Input must be sorted the same way
- ✗ Input must be partitioned the same way

Best plan if the data is already sorted and partitioned this way.
$\Longrightarrow$ Plan ahead!

Reducer Join

Already ran word count on YouTube and Yahoo answers.
But partitioned it differently $\rightarrow$ reduce join

Map: (word, count) $\mapsto$ (word, corpus, count)
Partition: word
Sort: (word, corpus)
Reduce: Divide counts

(We've seen this before with Alice and Bob)

Reducer Join: Efficiency

- ✗ Slow
- ✗ Data copied over network
- ✓ Large data
- ✓ General

Three Join Strategies

Sorted and partitioned same way?	$\rightarrow$Sorted Join in Map
Is one side small?	$\rightarrow$Hash Join in Map
General problem	$\rightarrow$Reducer Join (which is sorted)

Bloom Filters

A Problem

Interesting words do not fit in RAM, still want to do a hash join.

In General

Efficiently represent a set with some false positives.

Bloom Filter

Represent a set, probabilistically.

`insert(key)` Add `key` to the set.

`query(key)` If `key` is in the set, return `maybe`.
If `key` is not in the set, return `no` or `maybe`.

Bloom Filter

Represent a set, probabilistically.

`insert(key)` Add `key` to the set.

`query(key)` If `key` is in the set, return `maybe`.
If `key` is not in the set, return `no` or `maybe`.

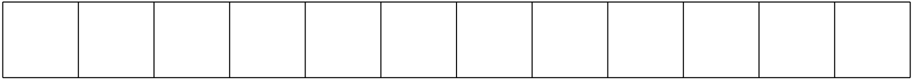
Usage

Ask a Bloom filter locally.

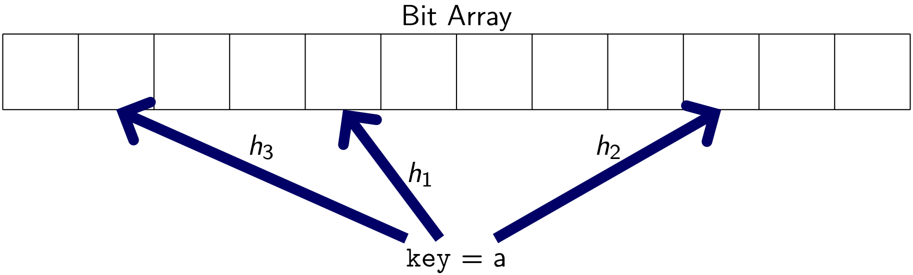
`no` Key is definitely not found → avoid network.

`maybe` Ask the network.

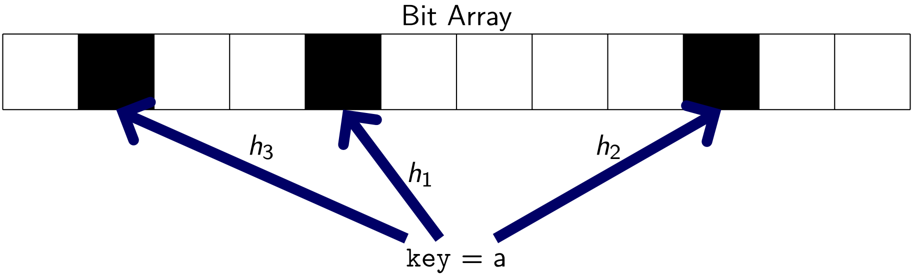
Bit Array



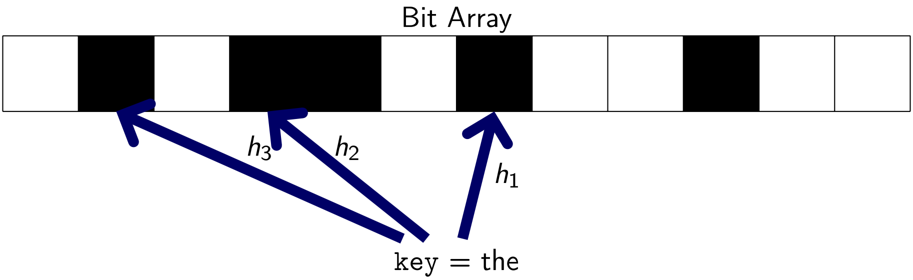
1. Initially the array is all 0s.



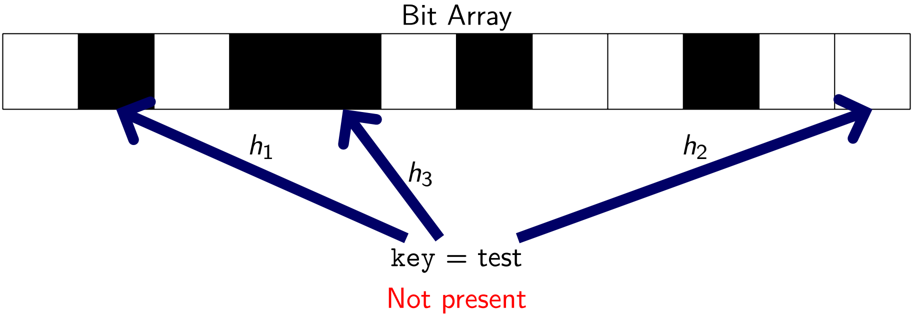
1. Initially the array is all 0s.
2. Hash functions assign bit positions to keys.



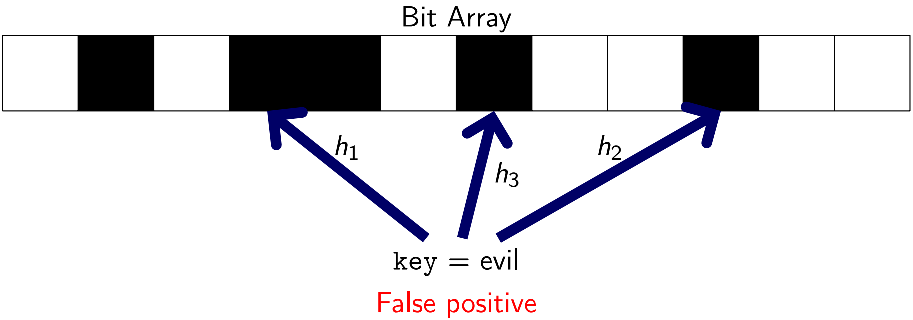
1. Initially the array is all 0s.
2. Hash functions assign bit positions to keys.
3. Insertion sets the corresponding bits to 1.



1. Initially the array is all 0s.
2. Hash functions assign bit positions to keys.
3. Insertion sets the corresponding bits to 1.



1. Initially the array is all 0s.
2. Hash functions assign bit positions to keys.
3. Insertion sets the corresponding bits to 1.
4. Queries check that the corresponding bits are 1.



1. Initially the array is all 0s.
2. Hash functions assign bit positions to keys.
3. Insertion sets the corresponding bits to 1.
4. Queries check that the corresponding bits are 1.

Bloom Filters: memory efficient

. . . but some probability of false positives.

Bloom Filters: memory efficient
. . . but some probability of false positives.

Not done yet:

- Need multiple hash functions.
- What is the false-positive probability?
- How many hash functions?

Multiple Hash Functions?

We need independent hash functions:

$$h_1(\texttt{the}), h_2(\texttt{the}), h_3(\texttt{the}), \ldots$$

Multiple Hash Functions?

We need independent hash functions:

$$h_1(\texttt{the}), h_2(\texttt{the}), h_3(\texttt{the}), \ldots$$

Just use one good hash function h and concatenate with key:

$$h(\texttt{1\_key}), h(\texttt{2\_key}), h(\texttt{3\_key}), \ldots$$

The optimal number of hashes is

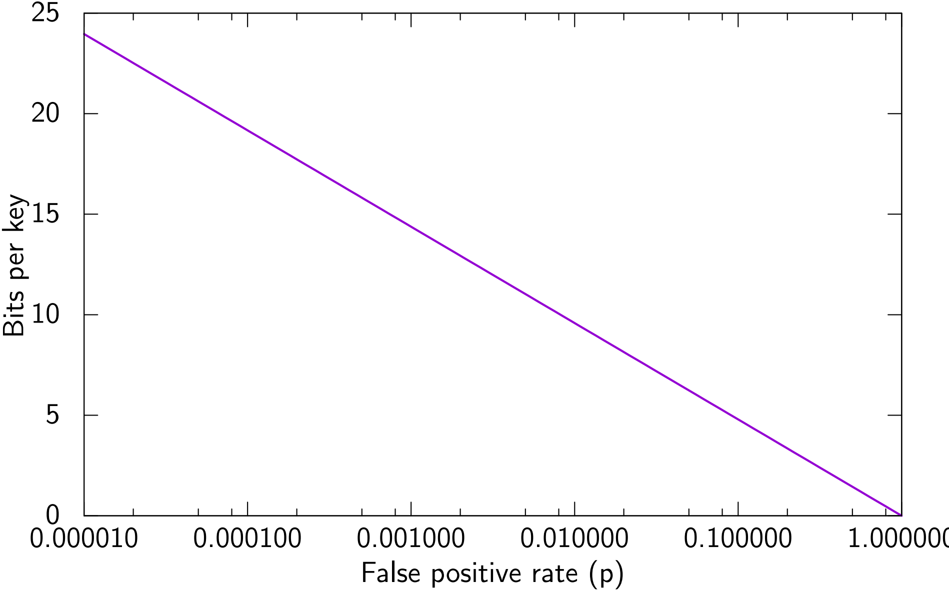
$$\text{hashes} \approx \frac{\text{bits}}{\text{entries}} \ln 2$$

To satisfy false-positive probability p, Bloom filters use

$$\approx \frac{-\log_2 p}{\ln 2}$$

bits per key.

Don't worry about the exact equations.
But deriving them is fun!


25
20
15
10
5
0
Bits per key
0.000010
0.000100
0.001000
0.010000
0.100000
1.000000
False positive rate (p)

Summary

Approximately represent a very large set in small memory.

Used to reduce expensive lookups in SSTable, BigTable, . . .

Also useful in isolation for error-tolerant tasks.