Lecture 2: Sets & Bloom filters

(a) Sets
(b) Bloom filters

(a) Sets

Slides from Eric Roberts, CS 106B, Stanford

Outline

1. A brief overview of mathematical set theory
2. Extending the **`set.h`** interface
3. Comparison functions
4. Implementation strategies for sets
5. Mention characteristic vectors

Sets in Mathematics

- A ***set*** is an unordered collection of distinct values.

  digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
  evens = { 0, 2, 4, 6, 8 }
  odds = { 1, 3, 5, 7, 9 }
  primes = { 2, 3, 5, 7 }
  squares = { 0, 1, 4, 9 }

  colors = { *red*, *yellow*, *green*, *cyan*, *blue*, *magenta* }
  primary = { *red*, *green*, *blue* }
  secondary = { *yellow*, *cyan*, *magenta* }

  R = { x | x is a real number }
  Z = { x | x is an integer }
  N = { x | x is an integer and $x \geq 0$ }

- The set with no elements is called the ***empty set*** ($\varnothing$).

Set Operations

- The fundamental set operation is ***membership*** ($\in$).

$3 \in$ **primes**	$3 \notin$ **evens**
red $\in$ **primary**	*red* $\notin$ **secondary**
$-1 \in$ **Z**	$-1 \notin$ **N**

- The ***union*** of two sets A and B ($A \cup B$) consists of all elements in either A or B or both.
- The ***intersection*** of A and B ($A \cap B$) consists of all elements in both A or B.
- The ***set difference*** of A and B ($A - B$) consists of all elements in A but not in B.
- Set A is a ***subset*** of B ($A \subseteq B$) if all elements in A are also in B.
- Sets A and B are ***equal*** ($A = B$) if they have the same elements.

Exercise: Set Operations

Suppose that you have the following sets:

digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
evens = { 0, 2, 4, 6, 8 }
odds = { 1, 3, 5, 7, 9 }
primes = { 2, 3, 5, 7 }
squares = { 0, 1, 4, 9 }

What is the value of each of the following expressions:

a) **evens** $\cup$ **squares** { 0, 1, 2, 4, 6, 8, 9 }

b) **odds** $\cap$ **squares** { 1, 9 }

c) **squares** $\cap$ **primes** $\varnothing$

d) **primes** – **evens** { 3, 5, 7 }

Given only these sets, can you produce the set { 1, 2, 9 }?

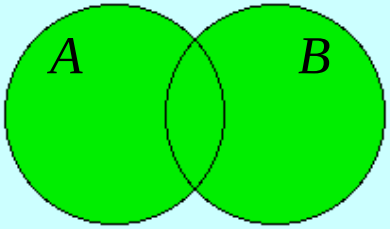
(**primes** $\cap$ **evens**) $\cup$ (**odds** $\cap$ **squares**)

Fundamental Set Identities

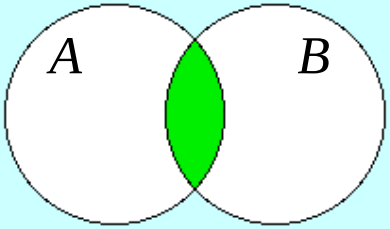
$S \cup S \equiv S$ $S \cap S \equiv S$	*Idempotence*
$A \cap (A \cup B) \equiv A$ $A \cup (A \cap B) \equiv A$	*Absorption*
$A \cup B \equiv B \cup A$ $A \cap B \equiv B \cap A$	*Commutative laws*
$A \cup (B \cup C) \equiv (A \cup B) \cup C$ $A \cap (B \cap C) \equiv (A \cap B) \cap C$	*Associative laws*
$A \cap (B \cup C) \equiv (A \cap B) \cup (A \cap C)$ $A \cup (B \cap C) \equiv (A \cup B) \cap (A \cup C)$	*Distributive laws*
$A - (B \cup C) \equiv (A - B) \cap (A - C)$ $A - (B \cap C) \equiv (A - B) \cup (A - C)$	*DeMorgan's laws*

Venn Diagrams

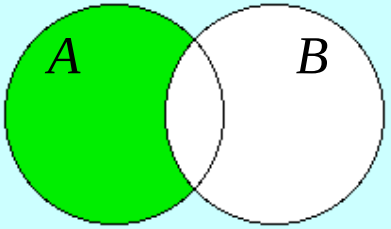
- A ***Venn diagram*** is a graphical representation of a set in that indicates common elements as overlapping areas.
- The following Venn diagrams illustrate the effect of the union, intersection, and set-difference operators:



$A \cup B$



$A \cap B$



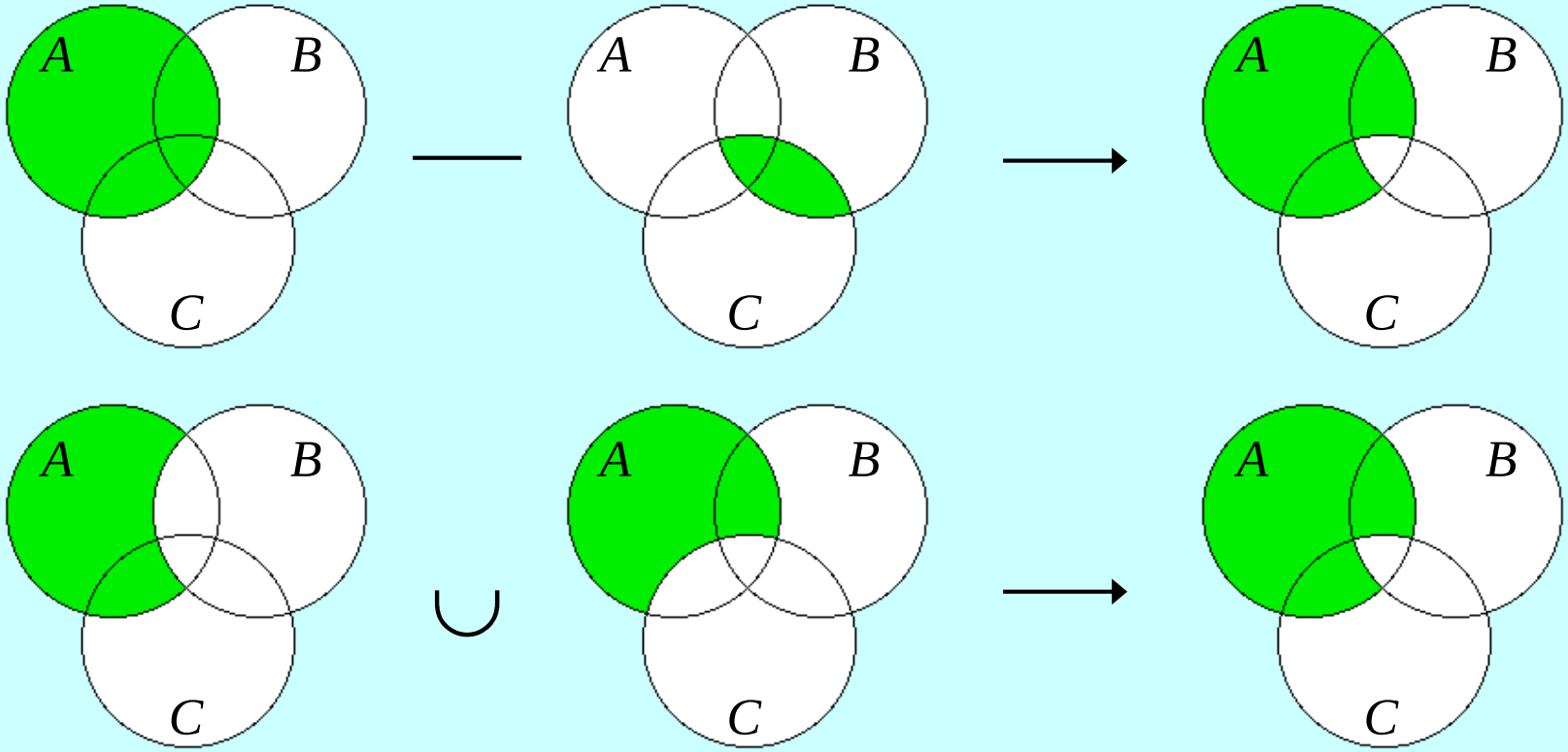
$A - B$

- If $A \subseteq B$, then the circle for A in the Venn diagram lies entirely within (or possibly coincident with) the circle for B.
- If $A = B$, then the circles for A and B are the same.

Venn Diagrams as Informal Proofs

- You can also use Venn diagrams to justify set identities. Suppose, for example, that you wanted to prove

$$A - (B \cap C) \equiv (A - B) \cup (A - C)$$



Extending the `set.h` Interface

- As described in Chapter 5, the **`Set`** class supports only the low-level operations of adding, removing, and testing the presence of an element. To simplify the development of set algorithms, it needs to support high-level operations of *union*, *intersection*, *set difference*, *subset*, and *equality* as well.
- The **`Set`** implementation must be able to compare elements of the specified value type. For most types, the built-in **`==`** and **`<`** operators are sufficient. Given that user-defined compound types will not define necessarily these operators, it is useful to allow clients to specify a *comparison function* for the value type as part of the **`Set`** constructor.

High-Level Set Operators

```
/*
 * Operator: ==
 * Usage: set1 == set2
 * -------------------
 * Returns true if set1 and set2 contain the same elements.
 */

   bool operator==(const Set & set2) const;

/*
 * Operator: !=
 * Usage: set1 != set2
 * -------------------
 * Returns true if set1 and set2 are different.
 */

   bool operator!=(const Set & set2) const;
```

Indicates that **`set1`** will not be changed.

High-Level Set Operators

```
/*
 * Operator: +
 * Usage: set1 + set2
 *        set1 + element
 * ----------------------
 * Returns the union of sets set1 and set2, which is the set of elements
 * that appear in at least one of the two sets.  The right hand set can be
 * replaced by an element of the value type, in which case the operator
 * returns a new set formed by adding that element.
 */

   Set operator+(const Set & set2) const;
   Set operator+(const ValueType & element) const;

/*
 * Operator: +=
 * Usage: set1 += set2;
 *        set1 += value;
 * ----------------------
 * Adds all elements from set2 (or the single specified value) to set1.
 */

   Set & operator+=(const Set & set2);
   Set & operator+=(const ValueType & value);
```

High-Level Set Operators

```
/*
 * Operator: *
 * Usage: set1 * set2
 * ------------------
 * Returns the intersection of sets set1 and set2, which is the set of all
 * elements that appear in both.
 */

   Set operator*(const Set & set2) const;

/*
 * Operator: *=
 * Usage: set1 *= set2;
 * --------------------
 * Removes any elements from set1 that are not present in set2.
 */

   Set & operator*=(const Set & set2);
```

High-Level Set Operators

```
/*
 * Operator: -
 * Usage: set1 - set2
 *        set1 - element
 * ----------------------
 * Returns the difference of sets set1 and set2, which is all of the
 * elements that appear in set1 but not set2.  The right hand set can be
 * replaced by an element of the value type, in which case the operator
 * returns a new set formed by removing that element.
 */

   Set operator-(const Set & set2) const;
   Set operator-(const ValueType & element) const;

/*
 * Operator: -=
 * Usage: set1 -= set2;
 *        set1 -= value;
 * ----------------------
 * Removes all elements from set2 (or a single value) from set1.
 */

   Set & operator-=(const Set & set2);
```

A Template-based `sort` Function

- Templates can be used to create a generic functions that can be applied to arguments of various types.
- The following code, for example, creates a template version of the **`sort`** function that works with vectors of any element type (or at least for those that define the **`<`** comparison operator):

```
template <typename ValueType>
void sort(Vector<ValueType> & vec) {
   for (int lh = 0; lh < vec.size() - 1; lh++) {
      int rh = lh;
      for (int i = lh + 1; i < vec.size(); i++) {
         if (vec[i] < vec[rh]) rh = i;
      }
      ValueType temp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = temp;
   }
}
```

Specifying a Comparison Function

- In some cases, it may be useful to specify your own function for comparing elements in the vector. In this case, the **`sort`** function must take a second argument, as follows:

```
template <typename ValueType>
void sort(Vector<ValueType> & vec, int (*cmp)(ValueType,ValueType)) {
   for (int lh = 0; lh < vec.size() - 1; lh++) {
      int rh = lh;
      for (int i = lh + 1; i < vec.size(); i++) {
         if (cmp(vec[i], vec[rh]) < 0) rh = i;
      }
      ValueType temp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = temp;
   }
}
```

- The argument **`cmp`** is a ***comparison function***, which takes two values of the element type and returns an integer that is negative if the first value is smaller than the second, zero if the values are equal, and positive if the first is larger.

Specifying a Comparison Function

- To use this generalized version of **`sort`**, your must first define a comparison function. For example, if you want to sort strings by length, you could define the following comparison function, which returns an integer of the appropriate sign:

```
int lengthCompare(string s1, string s2) {
   return s1.length() - s2.length();
}
```

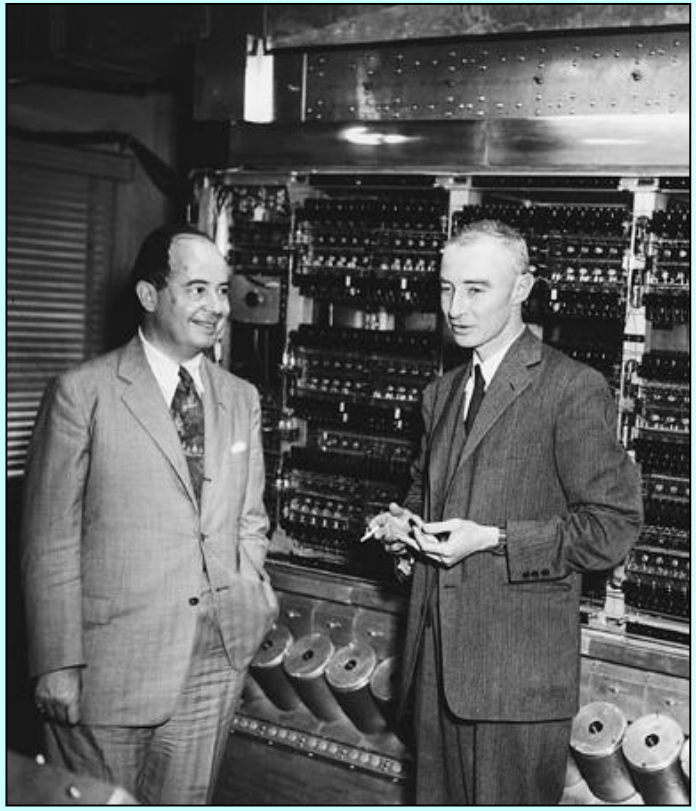
- In C++, understanding how to *use* a function argument is easy; all you have to do is supply its name. Thus, to sort a string vector by length, you would write

```
sort(strvec, lengthCompare);
```

The hard part is understanding how to *declare* a function argument, which in C++ must be a ***pointer to a function***.

The von Neumann Architecture

- One of the fundamental ideas of modern computing—traditionally attributed to John von Neumann although others can make valid claims to the idea—is that code is stored in the same memory as data. This concept is called the ***stored programming model***.
- If you go on to take CSAPP, you will learn more about how code is represented inside the computer. For now, the important idea is that the code for every C++ function is stored somewhere in memory and therefore has an address.



John von Neumann and J. Robert Oppenheimer

Function Pointers in C++

- As C did before it, C++ makes it possible for programmers to use function pointers explicitly.
- The syntax for declaring function pointers is consistent with the syntax for other pointer declarations, although it takes some getting used to. Consider the following declarations:

`int n;`	*Declares* **`n`** *as an* **`int`**.
`int *pn;`	*Declares* **`pn`** *as a pointer to an* **`int`**.
`int f();`	*Declares* **`f`** *as a function returning an* **`int`**.
`int *g();`	*Declares* **`g`** *as a function returning a pointer to an* **`int`**.
`int (*fn)();`	*Declares* **`fn`** *as a pointer to a function that takes no arguments and returns an* **`int`**.
`int (*cmp)(int, int);`	*Declares* **`cmp`** *as a pointer to a function that takes two* **`int`**s *and returns an* **`int`**.

The `cmpfn.h` Interface

```
/*
 * File: cmpfn.h
 * -------------
 * This interface exports a template function for comparing values of an
 * unspecified type.  Most clients will have no need to use this interface
 * explicitly.  Its primary purpose is to provide a default comparison
 * function that allows maps and sets to use the standard operators defined
 * for their base type.
 */

#ifndef _cmpfn_h
#define _cmpfn_h

/*
 * Function: operatorCmp
 * Usage: int sign = operatorCmp(v1, v2);
 * --------------------------------------
 * This template function is a generic function that compares two values
 * using the built-in == and < operators */

template <typename Type>
int operatorCmp(Type v1, Type v2) {
   if (v1 == v2) return 0;
   if (v1 < v2) return -1;
   return 1;
}

#endif
```

Implementing Sets

- Modern library systems adopt either of two strategies for implementing sets:
  - ***Hash tables***. Sets implemented as hash tables are extremely efficient, offering average *O*(1) performance for adding a new element or testing for membership. The primary disadvantage is that hash tables do not support iteration in the order imposed by the value type.
  - ***Balanced binary trees***. Sets implemented using balanced binary trees offer *O*(log *N*) performance on the fundamental operations, but do make it possible to write an ordered iterator.
- As with the **`Map`** class, C++ uses the latter approach.

The Easy Implementation

- As is so often the case, the easy way to implement the **`Set`** class is to build it out of data structures that you already have. In this case, it make sense to build **`Set`** on top of the **`Map`** class.
- The private section looks like this:

```
/*
 * File: setpriv.h
 * ---------------
 * This file contains the private section for the set.h interface.
 */

/* Instance variables */

   Map<ValueType,char> map;          /* The char is unused */
```

The `setimpl.cpp` Implementation

```
template <typename ValueType>
Set<ValueType>::Set(int (*cmp)(ValueType, ValueType)) : map(cmp) {
   /* Empty */
}

template <typename ValueType>
Set<ValueType>::~Set() {
   /* Empty */
}

template <typename ValueType>
int Set<ValueType>::size() const {
   return map.size();
}

template <typename ValueType>
bool Set<ValueType>::isEmpty() const {
   return map.isEmpty();
}

template <typename ValueType>
void Set<ValueType>::add(ValueType element) {
   map.add(element);
}
```

. . . and so on . . .

The `setimpl.cpp` Implementation

```
/*
 * Implementation notes: ==
 * ------------------------
 * Two sets are equal if they are subsets of each other.
 */

template <typename ValueType>
bool Set<ValueType>::operator==(const Set & s2) const {
   return isSubsetOf(s2) && s2.isSubsetOf(*this);
}

/*
 * Implementation notes: isSubsetOf
 * --------------------------------
 * The implementation of the high-level functions does not require knowledge
 * of the underlying representation
 */

template <typename ValueType>
bool Set<ValueType>::isSubsetOf(Set & s2) {
   foreach (ValueType value in *this) {
      if (!s2.contains(value)) return false;
   }
   return true;
}
```

Exercise: Implementing Set Methods

```
template <typename ValueType>
Set<ValueType> Set<ValueType>::operator+(const Set & s2) const {









}

template <typename ValueType>
ValueType Set<ValueType>::first() {









}
```

Exercise: Implementing Set Methods

```
template <typename ValueType>
Set<ValueType> Set<ValueType>::operator+(const Set & s2) const {

        if (cmpFn != set2.cmpFn) {
                error("Sets have different comparison functions");
        }
        Set<ValueType> set = *this;
        foreach (ValueType value in set2) {
                set.add(value);
        }
        return set;

}

template <typename ValueType>
ValueType Set<ValueType>::first() {




}
```

Exercise: Implementing Set Methods

```
template <typename ValueType>
Set<ValueType> Set<ValueType>::operator+(const Set & s2) const {

        if (cmpFn != set2.cmpFn) {
                error("Sets have different comparison functions");
        }
        Set<ValueType> set = *this;
        foreach (ValueType value in set2) {
                set.add(value);
        }
        return set;

}

template <typename ValueType>
ValueType Set<ValueType>::first() {

        if (isEmpty()) error("first: set is empty");
                return *begin();

}
```

Initial Versions Should Be Simple

Premature optimization is the root of all evil.
—Don Knuth

- When you are developing an implementation of a public interface, it is best to begin with the simplest possible code that satisfies the requirements of the interface.
- This approach has several advantages:
  - You can get the package out to clients much more quickly.
  - Simple implementations are much easier to get right.
  - You often won't have any idea what optimizations are needed until you have actual data from clients of that interface. In terms of overall efficiency, some optimizations are much more important than others.

Sets and Efficiency

- After you release the set package, you might discover that clients use them often for particular types for which there are much more efficient data structures than binary trees.
- One thing you could do easily is check to see whether the element type was **`string`** and then use a **`Lexicon`** instead of a binary search tree. The resulting implementation would be far more efficient. This change, however, would be valuable only if clients used **`Set<string>`** often enough to make it worth adding the complexity.
- One type of sets that do tend to occur in certain types of programming is **`Set<char>`**, which comes up, for example, if you want to specify a set of delimiter characters for a scanner. These sets can be made astonishingly efficient as described on the next few slides, which we won't have time to go over.

Character Sets

- The key insight needed to make efficient character sets (or, equivalently, sets of small integers) is that you can represent the inclusion or exclusion of a character using a single bit. If the bit is a 1, then that element is in the set; if it is a 0, it is not in the set.
- You can tell what character value you're talking about by creating what is essentially an array of bits, with one bit for each of the ASCII codes. That array is called a ***characteristic vector***.
- What makes this representation so efficient is that you can pack the bits for a characteristic vector into a small number of words inside the machine and then operate on the bits in large chunks.
- The efficiency gain is enormous. Using this strategy, most set operations can be implemented in just a few instructions.

Bit Vectors and Character Sets

- This picture shows a characteristic vector representation for the set containing the upper- and lowercase letters:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
3	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bitwise Operators

- If you know your client is working with sets of characters, you can implement the set operators extremely efficiently by storing the set as an array of bits and then manipulating the bits all at once using C++'s ***bitwise operators***.
- The bitwise operators are summarized in the following table and then described in more detail on the next few slides:

x **&** *y*	Bitwise AND. The result has a 1 bit wherever both *x* and *y* have 1s.
x **|** *y*	Bitwise OR. The result has a 1 bit wherever either *x* or *y* have 1s.
x **^** *y*	Exclusive OR. The result has a 1 bit wherever *x* and *y* differ.
~*x*	Bitwise NOT. The result has a 1 bit wherever *x* has a 0.
x **<<** *n*	Left shift. Shift the bits in *x* left *n* positions, shifting in 0s.
x **>>** *n*	Right shift. Shift *x* right *n* bits (logical shift if *x* is unsigned).

The Bitwise AND Operator

- The bitwise AND operator (`&`) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position in which both *x* and *y* have 1 bits. A table for the `&` operator appears to the right.

	0	1
0	0	0
1	0	1

- The primary application of the `&` operator is to select certain bits in an integer, clearing the unwanted bits to 0. This operation is called ***masking***.
- In the context of sets, the `&` operator performs an intersection operation, as in the following calculation of **odds** ∩ **squares**:

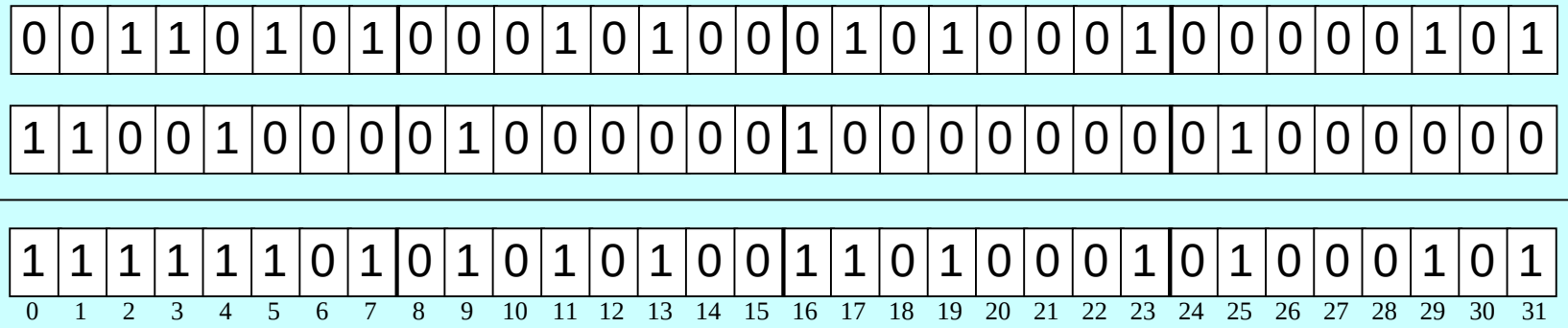
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
&	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

The Bitwise OR Operator

- The bitwise OR operator (`|`) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position which either *x* or *y* has a 1 bit (or if both do), as shown in the table on the right.

	0	1
0	0	1
1	1	1

- The primary use of the `|` operator is to assemble a single integer value from other values, each of which contains a subset of the desired bits.
- In the context of sets, the `|` operator performs a union, as in the following calculation of **primes** ∪ **squares**:



The Exclusive OR Operator

- The exclusive OR or XOR operator (^) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position in which *x* and *y* have different bit values, as shown on the right.

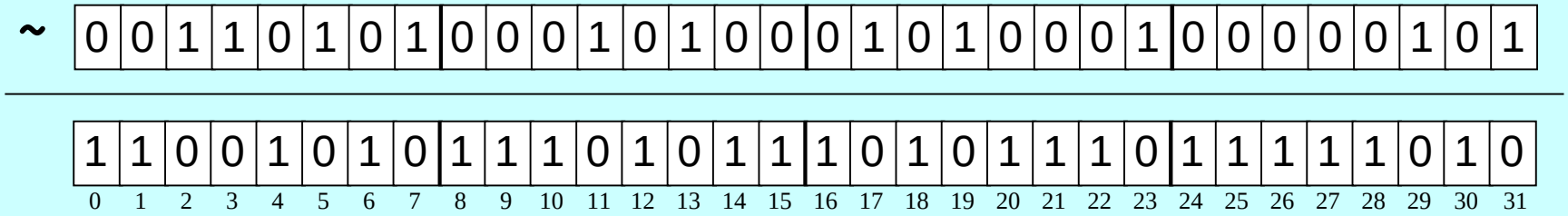
	0	1
0	0	1
1	1	0

- The XOR operator has many applications in programming, most of which are beyond the scope of this text.
- The following example flips all the bits in the rightmost three bytes of a word:

	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1
^	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The Bitwise NOT Operator

- The bitwise NOT operator (**~**) takes a single operand *x* and returns a value that has a 1 wherever *x* has a 0, and vice versa.
- You can use the bitwise NOT operator to create a mask in which you mark the bits you want to eliminate as opposed to the ones you want to preserve.
- The **~** operator creates the ***complement*** of a set, as shown with the following diagram of **~primes:**



- Question: How could you use the **~** operator to compute the set difference operation?

The Shift Operators

- C++ defines two operators that have the effect of shifting the bits in a word by a given number of bit positions.
- The expression *x* **<<** *n* shifts the bits in the integer *x* leftward *n* positions. Spaces appearing on the right are filled with 0s.
- The expression *x* **>>** *n* shifts the bits in the integer *x* rightward *n* positions. The question as to what bits are shifted in on the left depend on whether *x* is a signed or unsigned type:
  - If *x* is a signed type, the **>>** operator performs what computer scientists call an ***arithmetic shift*** in which the leading bit in the value of *x* never changes. Thus, if the first bit is a 1, the **>>** operator fills in 1s; if it is a 0, those spaces are filled with 0s.
  - If *x* is an unsigned type, the **>>** operator performs a ***logical shift*** in which missing digits are always filled with 0s.

(b) Bloom filters

Zhengwei QI

Most slides from
http://www.cs.jhu.edu/~fabian/courses/CS600.624/slides/bloomslides.pdf

Bloom Filter

Burton H. Bloom proposed the Bloom filter in 1970.

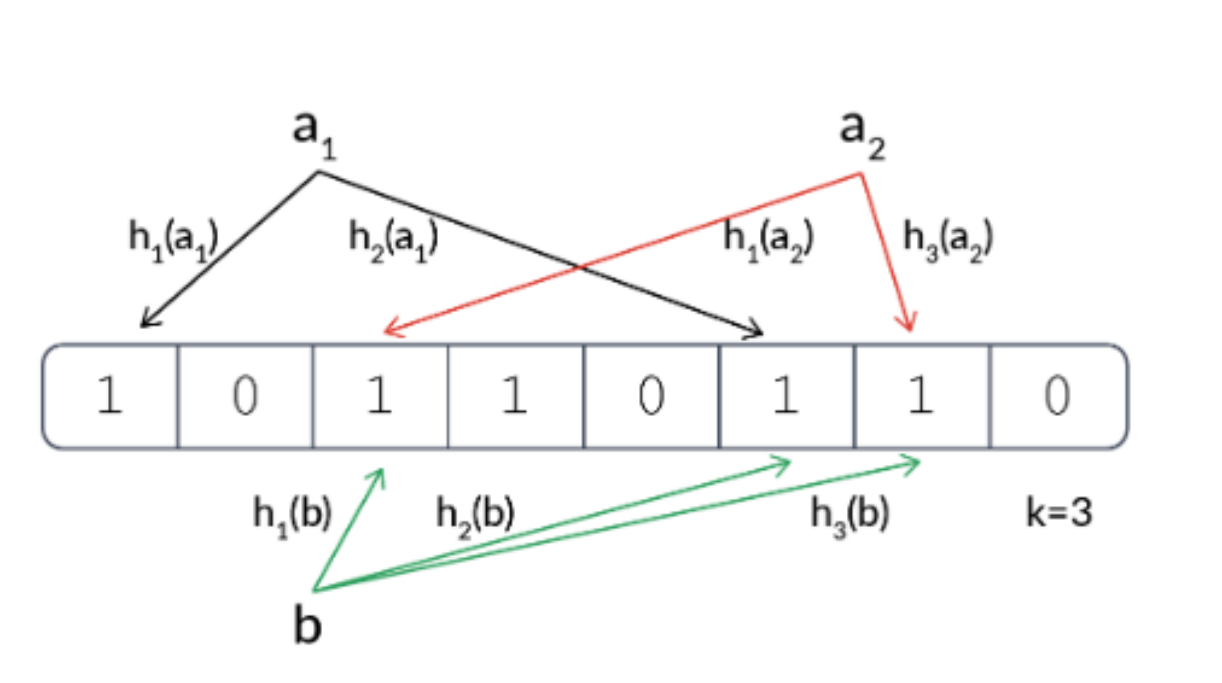
Space/Time Trade-offs in Hash Coding with Allowable Errors

BURTON H. BLOOM
Computer Usage Company, Newton Upper Falls, Mass.

TABLE I. SUMMARY OF EXPECTED PERFORMANCE OF HYPHENATION APPLICATION OF HASH CODING USING METHOD 2 FOR VARIOUS VALUES OF ALLOWABLE FRACTION OF ERRORS

P = Allowable Fraction of Errors	*N = Size of Hash Area (Bits)*	*Disk Accesses Saved*
$\frac{1}{2}$	72,800	45.0%
$\frac{1}{4}$	145,600	67.5%
$\frac{1}{8}$	218,400	78.7%
$\frac{1}{16}$	291,200	84.4%
$\frac{1}{32}$	364,000	87.2%
$\frac{1}{64}$	509,800	88.5%

Bloom Filter



A data structure designed to rapidly determine whether an **element** is present in a **set**, in a **memory-efficient** manner.

https://redislabs.com/redis-enterprise/redis-bloom/

LSM

Bigtable: A Distributed Storage System for Structured Data

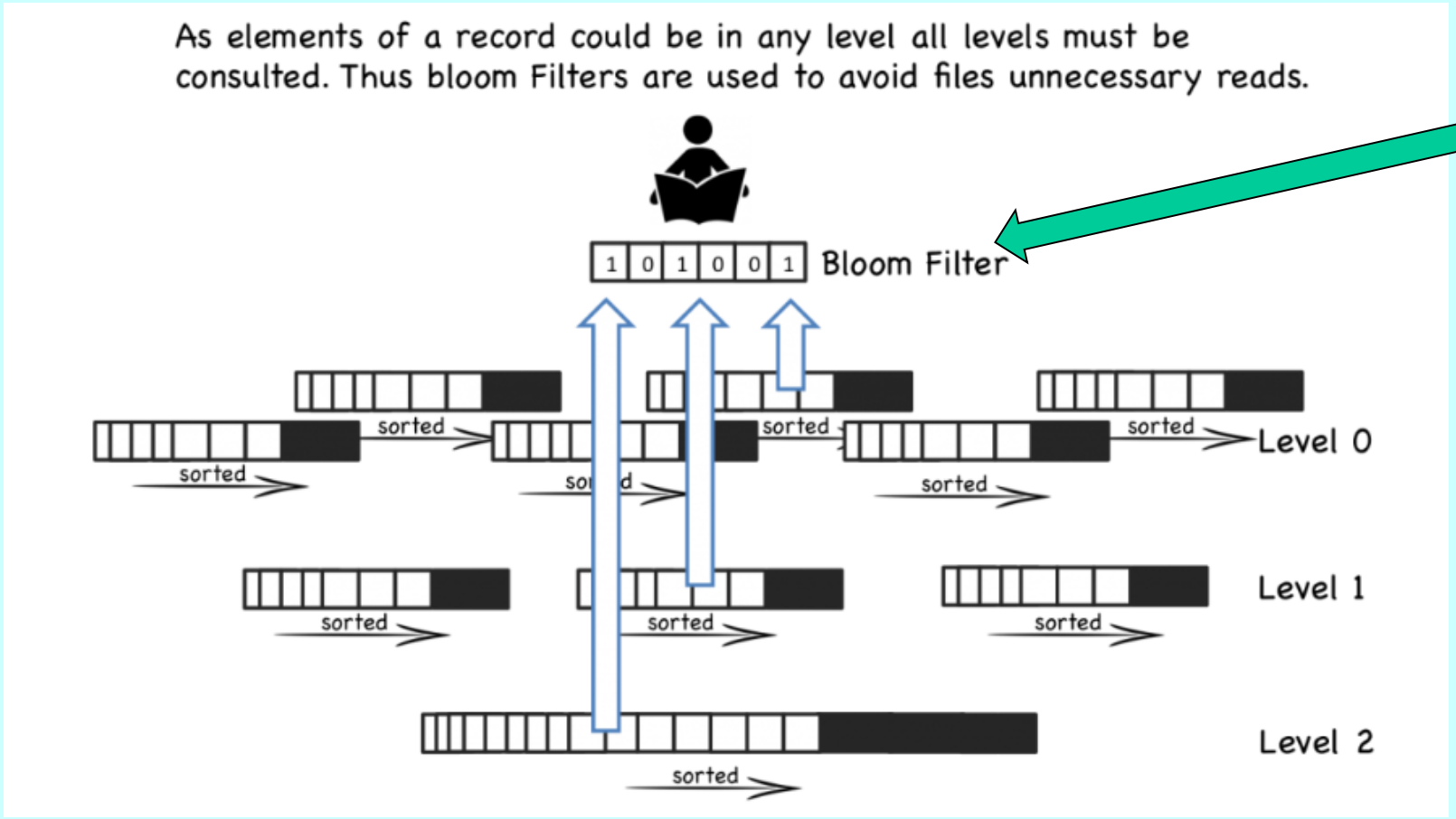
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
Google, Inc.

The manner in which Bigtable uses memtables and SSTables to store updates to tablets is analogous to the way that the Log-Structured Merge Tree [26] stores updates to index data. In both systems, sorted data is buffered in memory before being written to disk, and reads must merge data from memory and disk.

Bloom filters

As described in Section 5.3, a read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end up doing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom filters [7] should be created for SSTables in a particular locality group. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations. Our use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch disk.

Recap-应用：LSM算法



Notation

S is a set of n elements.

Set of k hash functions with range $\{1...m\}$ (or $\{0...m-1\}$).

m-long array of bits initialized to 0.

Families of Hash Functions

k hash functions h_1 ... h_k

We could use SHA1, MD5, etc.

How could we get a family of size k?

$h_i(x)$ = MD5(x + i) would work.

Example

We insert and query on a Bloom filter of size $m = 10$ and number of hash functions $k = 3$.

Let $H(x)$ denote the result of the three hash functions which we will write as a set of three values $\{h1(x), h2(x), h3(x)\}$

We start with an empty 10-bit long array:

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Example

Insert x_0: $H(x_0) = \{1,4,9\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

Insert x_1: $H(x_1) = \{4,5,8\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	**1**	1	0	0	1	1

Example

0	1	2	3	4	5	6	7	8	9
0	1	0	0	**1**	1	0	0	1	1

Query *y*0:

$H(y0) = \{0,4,8\} \rightarrow$ **No**

Query y1:

H(y1) = {1,5,8} → Yes (False Positive)

Note:

False Positive: 误报率：假阳性
False Negative：漏报率：假阴性

Question： 是否存在False Negative?

概率计算

After *n* elements inserted into bloom filter of size *m*, probability that a specific bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \approx \mathrm{e}^{-\frac{kn}{m}}$$

(The useful approximation comes from a well-known formula for calculating e):

$$\lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{-x} = \mathbf{e}$$

Thus the probability that a specific bit has been flipped to 1 is

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - \mathrm{e}^{-\frac{kn}{m}}$$

概率计算

A false positive on a query of element *x* occurs when all of the hash functions $h_1 ... h_k$ applied to *x* return a filter position that has a 1.

We assume hash functions to be independent.

Thus the probability of a false positive is

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - \mathrm{e}^{-\frac{kn}{m}}\right)^k$$

最小化误报率

We are given m and n, so we choose a k to minimize the false positive rate.

Let $p = \mathrm{e}^{-\frac{kn}{m}}$ **. Thus we have**

$$\begin{aligned} f &= \left(1 - \mathrm{e}^{-\frac{kn}{m}}\right)^k \\ &= (1-p)^k \\ &= \mathrm{e}^{k \ln (1-p)} \end{aligned}$$

最小化误报率

We are given m and n, so we choose a k to minimize the false positive rate.

Let $p = \mathrm{e}^{-\frac{kn}{m}}$ **. Thus we have**

$$\begin{aligned} f &= \left(1 - \mathrm{e}^{-\frac{kn}{m}}\right)^k \\ &= (1-p)^k \\ &= \mathrm{e}^{k \ln (1-p)} \end{aligned}$$

So we wish to minimize g = k ln(1 − p).

最小化误报率

We could use calculus. Less messy, we notice that since

$$\ln\left(e^{-\frac{kn}{m}}\right) = -\frac{kn}{m}$$

we have

$$\begin{aligned} g &= k\ln(1-p) \\ &= -\frac{m}{n}\ln(p)\ln(1-p) \end{aligned}$$

and by symmetry, we see that g is minimized when

$$p = \frac{1}{2}$$

最小化误报率

We could use calculus. Less messy, we notice that since

$$\ln\left(e^{-\frac{kn}{m}}\right) = -\frac{kn}{m}$$

we have

$$\begin{aligned} g &= k\ln(1-p) \\ &= -\frac{m}{n}\ln(p)\ln(1-p) \end{aligned}$$

and by symmetry, we see that g is minimized when

$$p = \frac{1}{2}$$

最小化误报率

Since $p = \mathrm{e}^{-\frac{kn}{m}}$ **, when**

$$p = \frac{1}{2}$$

we have

$$k = \ln 2 \cdot \left(\frac{m}{n}\right)$$

Plugging back into $f = (1 - p)^k$, we find the minimum false positive rate is

$$\left(\frac{1}{2}\right)^k \approx (.6185)^{\frac{m}{n}}$$

Caveat: k must be an integer.

m, n, k Examples

From http://www.cs.wisc.edu/~cao/papers/summary-cache/

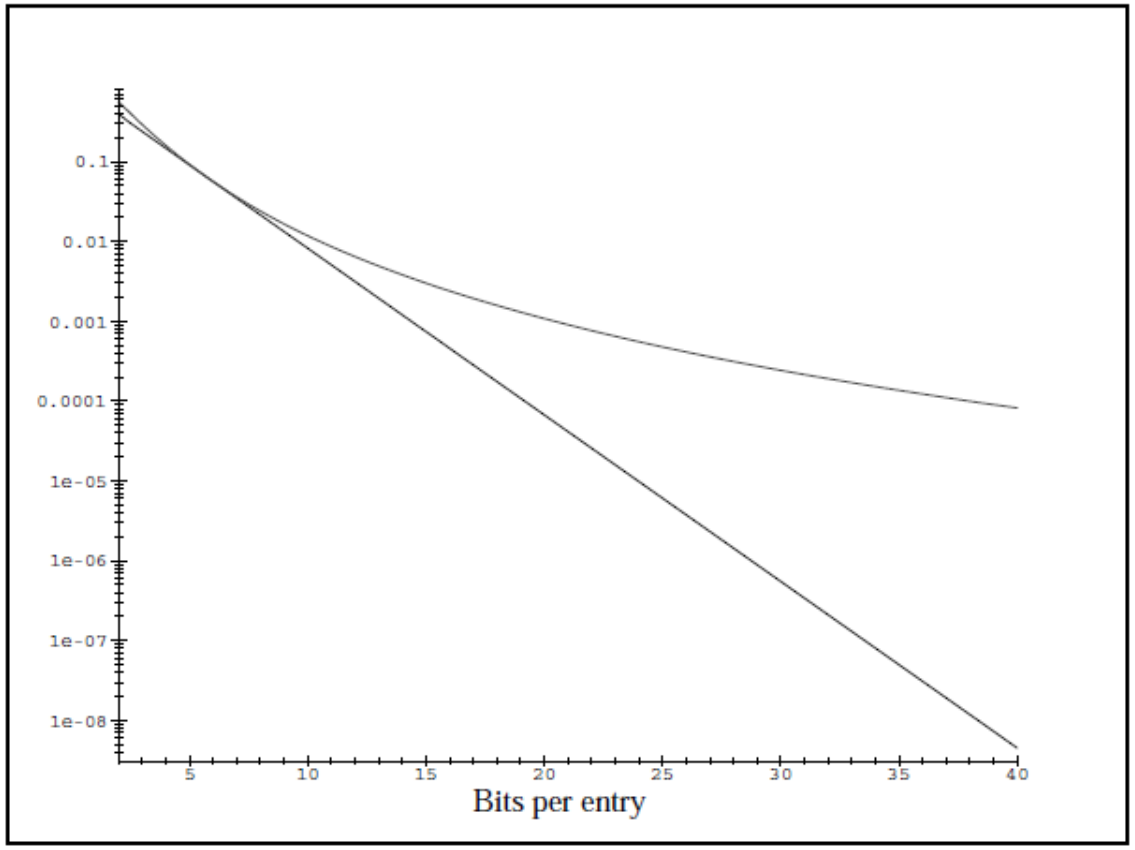
False positive rates for choices of *k* given *m/n*

m/*n*	*k*	*k*=1	*k*=2	*k*=3	*k*=4	*k*=5	*k*=6	*k*=7	*k*=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846
11	7.62	0.0869	0.0276	0.0136	0.00864	0.0065	0.00552	0.00513	0.00509
12	8.32	0.08	0.0236	0.0108	0.00646	0.00459	0.00371	0.00329	0.00314
13	9.01	0.074	0.0203	0.00875	0.00492	0.00332	0.00255	0.00217	0.00199
14	9.7	0.0689	0.0177	0.00718	0.00381	0.00244	0.00179	0.00146	0.00129
15	10.4	0.0645	0.0156	0.00596	0.003	0.00183	0.00128	0.001	0.000852

m, n, k Examples

From http://www.cs.wisc.edu/~cao/papers/summary-cache/

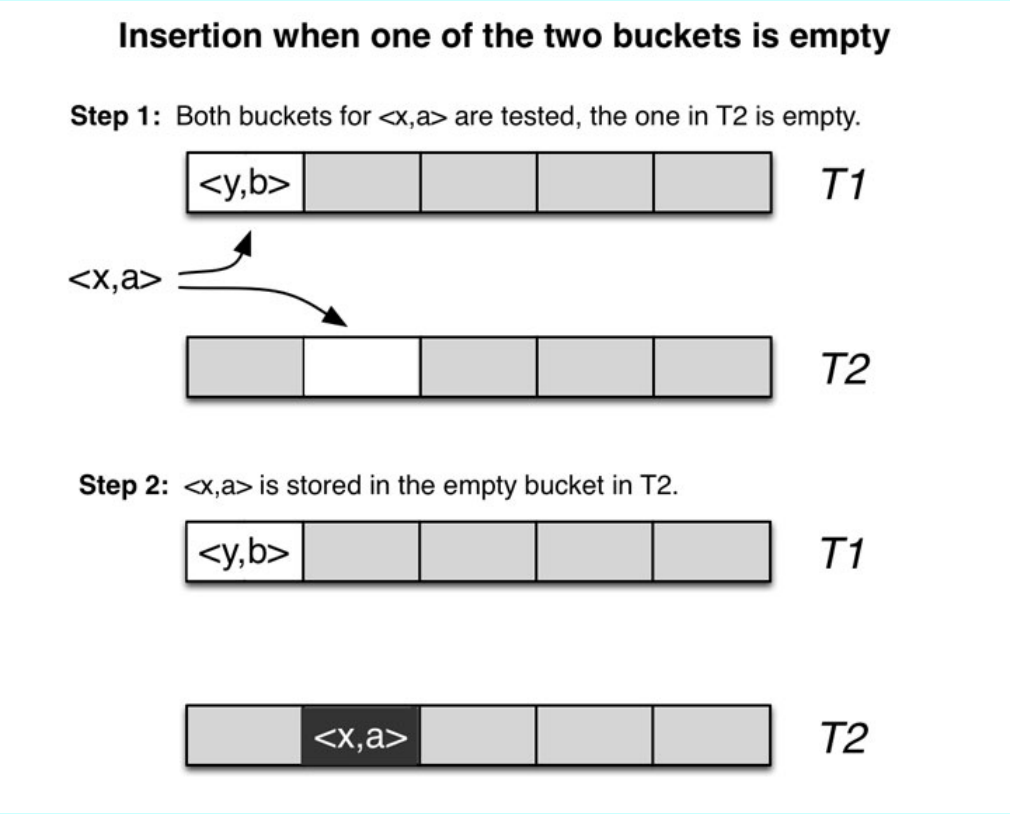
False positive rates for choices of k given m/n



Probability of false positives (log scale). The top curve is for 4 hash functions. The bottom curve is for the optimum (integral) number of hash functions.

How to support Delete?

Cuckoo filter
[http://www.cs.cmu.edu/~binfan/papers/conext14_cuckoofilter.pdf]
An alternative to Bloom filter with additional support for deletion of elements from a set.

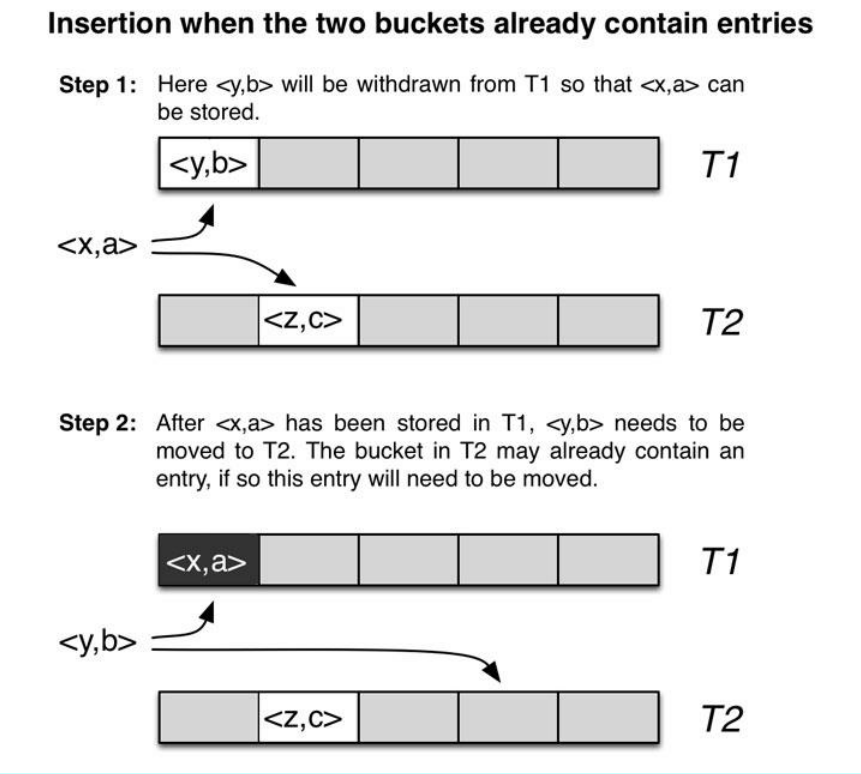


http://codecapsule.com/2013/07/20/cuckoo-hashing/

How to support Delete?

Cuckoo filter

An alternative to Bloom filter with additional support for deletion of elements from a set.



http://codecapsule.com/2013/07/20/cuckoo-hashing/

Reference

1. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert Gruber: Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). OSDI 2006: 205-218Log
2. B. Bloom, “Space/Time Trade-Offs in Hash Coding with Allowable Errors,” Communications of the ACM, Vol. 13 No. 7, 1970, pp. 422-426.

Next

- AVL tree

The End