Bloom Filter

Anna Jambura, Kathrin Pürstinger
Franziska Schnappauf, Coco Thiele

Was sind Bloom Filter?

- probabilistische Datenstruktur
- Prüfung der Mengenzugehörigkeit
- Ursprung: Burton H. Bloom, 1970

Membership Problem

- "Ist ein Element x Teil einer Menge S?"

$x \in S?$

- klassische Datenstrukturen -
  hoher Ressourcenbedarf

Lösungsansatz - Bloom Filter

- probabilistisch - Wahrscheinlichkeit statt absoluter Sicherheit

False Positives vs. False Negatives

Definition False Positive

Ein Element wird als vorhanden gemeldet, obwohl es nicht Teil der gegebenen Menge ist.

Definition False Negative

Ein Element wird als nicht vorhanden gemeldet, obwohl es Teil der gegebenen Menge ist. Ein Bloom Filter kann **kein** False Negative produzieren. Wird ein Element als „nicht vorhanden“ gemeldet, ist dies immer korrekt.

Trade-off

- Faktoren:

Reject Time	Speicherbedarf	Fehlerrate

Trade-off

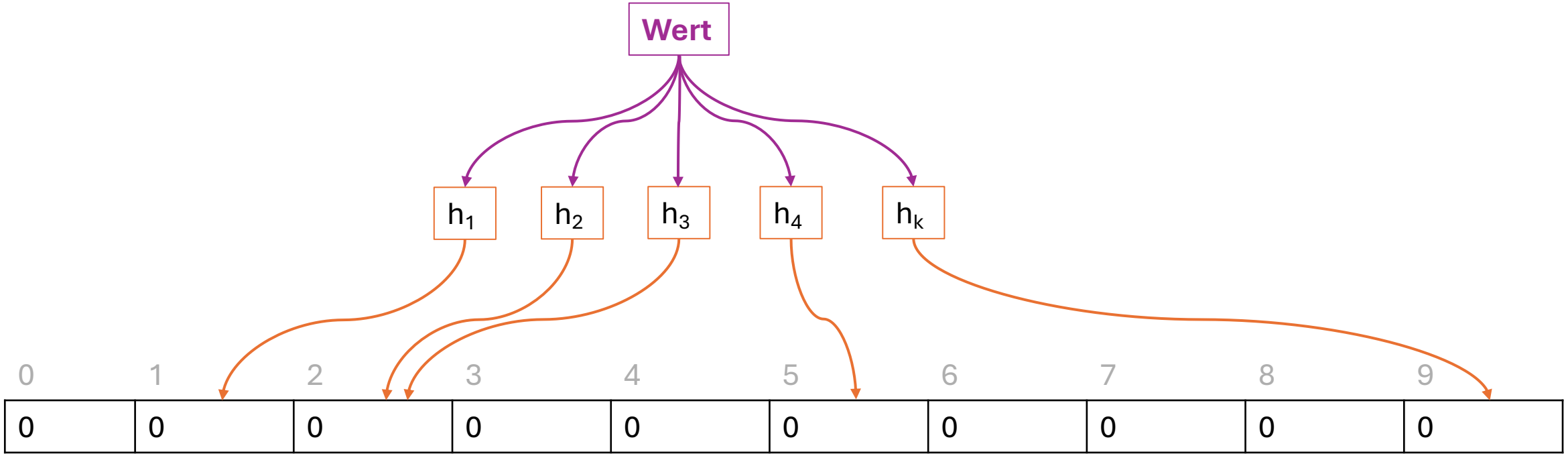
- Faktoren:

Reject Time	Speicherbedarf	Fehlerrate

- Genauigkeit vs. Speichereffizienz

Aufbau

- m-stelliges Bit-Array mit n Werten einer beliebigen Menge S
- k-unabhängige, uniforme Hashfunktionen



Einfügen der Werte

Folgende Schritte benötigt es zum Einfügen von Elementen:

1. berechne die k Hash-Werte: $h_1(x)$, $h_2(x)$, ..., $h_k(x)$
2. setze die Bits an allen k berechneten Positionen im Array auf 1
3. falls ein Bit bereits 1 ist, bleibt es 1

Beispiel

S: { 2, 4, 9 }

beispielhafte Hash-Funktionen:

$\mathbf{h_1(x)} = (x + 3) \bmod 10$

$\mathbf{h_2(x)} = (3 \cdot x + 1) \bmod 10$

$\mathbf{h_3(x)} = (x^2 + 2) \bmod 10$

Bit-Array

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

Beispiel

S: { 2, 4, 9 }

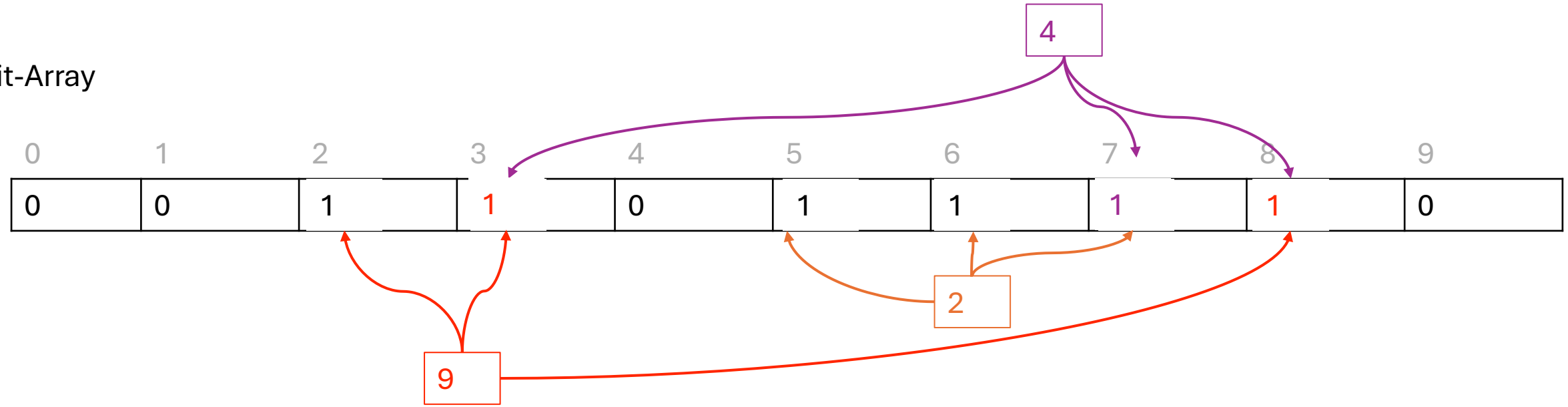
Hash-Werte:

$h_1(2)$ = (2 + 3) mod 10 = 5
$h_2(2)$ = (3 · 2 + 1) mod 10 = 7
$h_3(2)$ = (2^2 + 2) mod 10 = 6

$h_1(4)$ = (4 + 3) mod 10 = 7
$h_2(4)$ = (3 · 4 + 1) mod 10 = 3
$h_3(4)$ = (4^2 + 2) mod 10 = 8

$h_1(9)$ = (9 + 3) mod 10 = 2
$h_2(9)$ = (3 · 9 + 1) mod 10 = 8
$h_3(9)$ = (9^2 + 2) mod 10 = 3

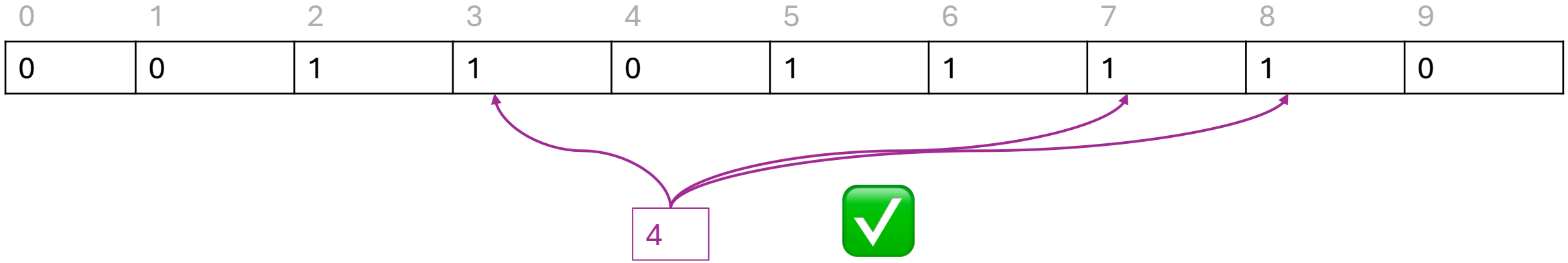
Bit-Array



Suchen von Elementen

Hash-Funktionen: $\mathbf{h_1(x)} = (x + 3) \bmod 10$
$\mathbf{h_2(x)} = (3 \cdot x + 1) \bmod 10$
$\mathbf{h_3(x)} = (x^2 + 2) \bmod 10$

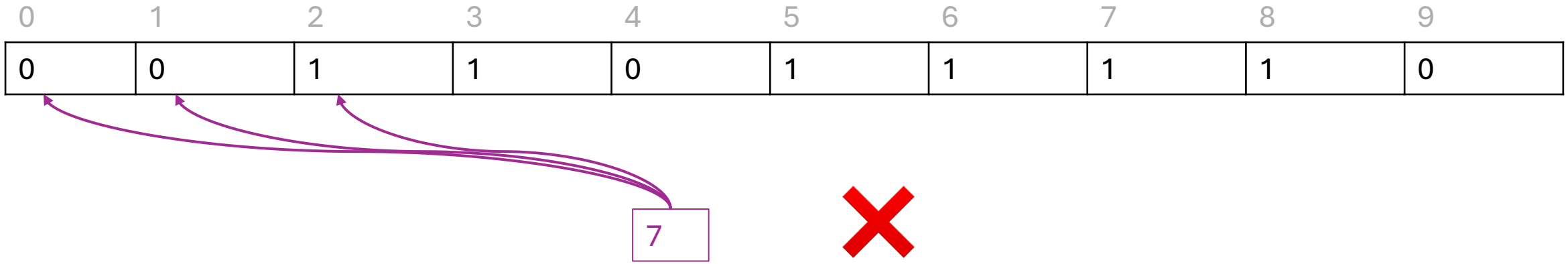
zu suchendes Element 4 : $\mathbf{h_1(4)} = (4 + 3) \bmod 10 = 7$
$\mathbf{h_2(4)} = (3 \cdot 4 + 1) \bmod 10 = 3$
$\mathbf{h_3(4)} = (4^2 + 2) \bmod 10 = 8$



Suchen von Elementen

Hash-Funktionen: $\mathbf{h_1(x)} = (x + 3) \bmod 10$
$\mathbf{h_2(x)} = (3 \cdot x + 1) \bmod 10$
$\mathbf{h_3(x)} = (x^2 + 2) \bmod 10$

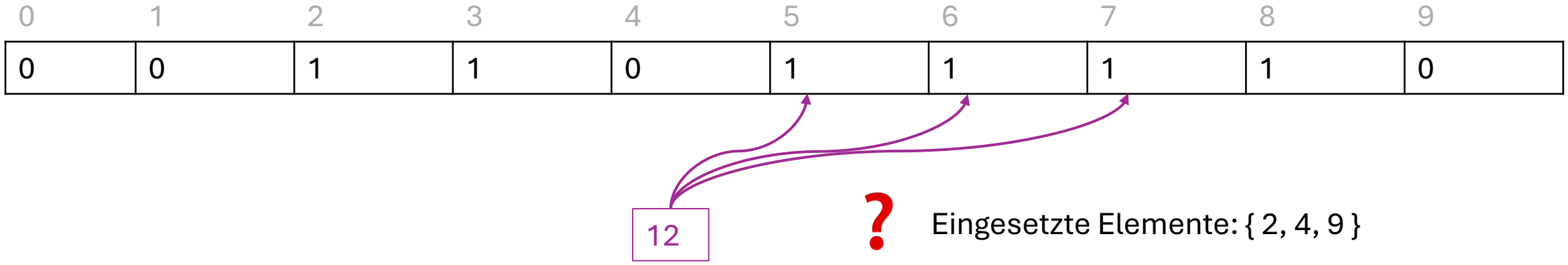
zu suchendes Element 7 : $\mathbf{h_1(7)} = (7 + 3) \bmod 10 = 0$
$\mathbf{h_2(7)} = (3 \cdot 7 + 1) \bmod 10 = 2$
$\mathbf{h_3(7)} = (7^2 + 2) \bmod 10 = 1$



Suchen von Elementen

Hash-Funktionen: $\mathbf{h_1(x)} = (x + 3) \bmod 10$
$\mathbf{h_2(x)} = (3 \cdot x + 1) \bmod 10$
$\mathbf{h_3(x)} = (x^2 + 2) \bmod 10$

zu suchendes Element 12 : $\mathbf{h_1(12)} = (12 + 3) \bmod 10 = 5$
$\mathbf{h_2(12)} = (3 \cdot 12 + 1) \bmod 10 = 7$
$\mathbf{h_3(12)} = (12^2 + 2) \bmod 10 = 6$



Formeln zur Evaluierung

- False Positive Wahrscheinlichkeit (False Positive Probability)

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

von “Theory and Practice of Bloom Filters for Distributed Systems“, S. Tarkoma, C. E. Rothenberg, E. Lagerspetz publ. 2012

Formeln zur Evaluierung

- optimale Anzahl Hashfunktionen

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}.$$

von “Theory and Practice of Bloom Filters for Distributed Systems“, S. Tarkoma, C. E. Rothenberg, E. Lagerspetz publ. 2012

Formeln zur Evaluierung - Beispiel

Szenario

Wir wollen 1.000 E-Mail-Adressen in einem Bloom Filter speichern und haben ein Bit-Array mit 8.000 Bits zur Verfügung.

n	1000
m	8000
k	???
False Positive Probability	???

Formeln zur Evaluierung - Beispiel

optimale Anzahl Hashfunktionen

Gegeben: $n = 1.000$, $m = 8.000$

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}.$$

$$k_{\text{opt}} = \frac{8.000}{1.000} \times \ln 2$$

$$k_{\text{opt}} \approx 5{,}5$$

$k_{\text{opt}} = 6$ **Hashfunktionen**

Formeln zur Evaluierung - Beispiel

False Positive Probability

Gegeben: $k = 6, n = 1.000, m = 8.000$

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

$$= (1 - e^{-\frac{6 \times 1.000}{8.000}})^6$$

$$= (1 - e^{-0,75})^6$$

$$= (1 - 0,472)^6$$

$$= 0,528^6$$

$$\approx 0,0219$$

$$\approx 2,19\% \approx 2,2\%$$

Pseudocode

Initialisierung

Einfügen

Abfragen

```
1     BloomFilterInit(m,k):
2         bitArray B[0..m-1] ← 0
3         hashFunctions h_1, h_2, …, h_k
4
5     Insert(x):
6         for i = 1 to k:
7             index ← h_i(x) mod m
8             B[index] ← 1
9
10    Contains(x):
11        for i = 1 to k:
12            index ← h_i(x) mod m
13            if B[index] ==  0:
14                return FALSE
15        return TRUE
```

O(k)

O(k)

Komplexitätsanalyse

Zeitkomplexität

- Einfügen + Abfragen: O(k)
- unabhängig von Anzahl der Elemente, die bereits gespeichert sind
- speichert keine Elemente, sondern nur Bit-Array

Speicherkomplexität

- O(m)
- Fehlerrate für Reduzierung des Speicherbedarfs
- abhängig von Anzahl Elemente und False-Positive-Wahrscheinlichkeit

Speichereffizienz

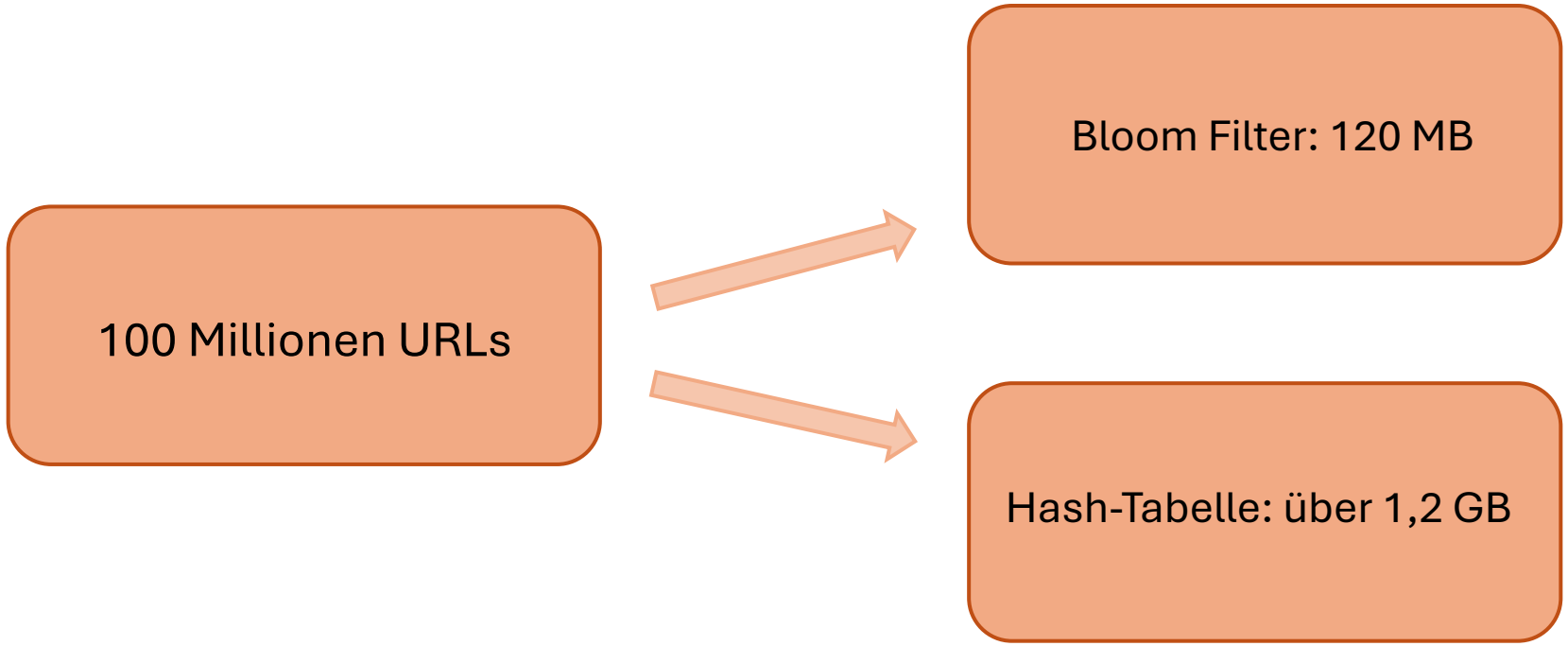
Fehlerrate von 1% → 9,6 Bits/Element unabhängig von der Größe des Elements

Vergleich

- Hashtabelle: mind. Größe des Elements selbst + Pointer = 64+ Bits pro Element

Speichereffizienz

konkret:



Vergleich mit anderen Datenstrukturen

Eigenschaft	Bloom Filter	Hash-Tabelle mit Chaining	Balancierter Baum
Zeitkomplexität	O(k) (konstant)	ø O(1), worst O(n)	O(log n)
Speicherkomplexität	O(m) (sehr gering)	O(n)	O(n)
Genauigkeit	Probabilistisch (False Positives)	Exakt (keine Fehler)	Exakt (keine Fehler)
Anwendungsfall	Membership-Test bei großen Datenmengen	Exakte Mitgliedschaft	Geordnete Speicherung, Bereichsabfragen

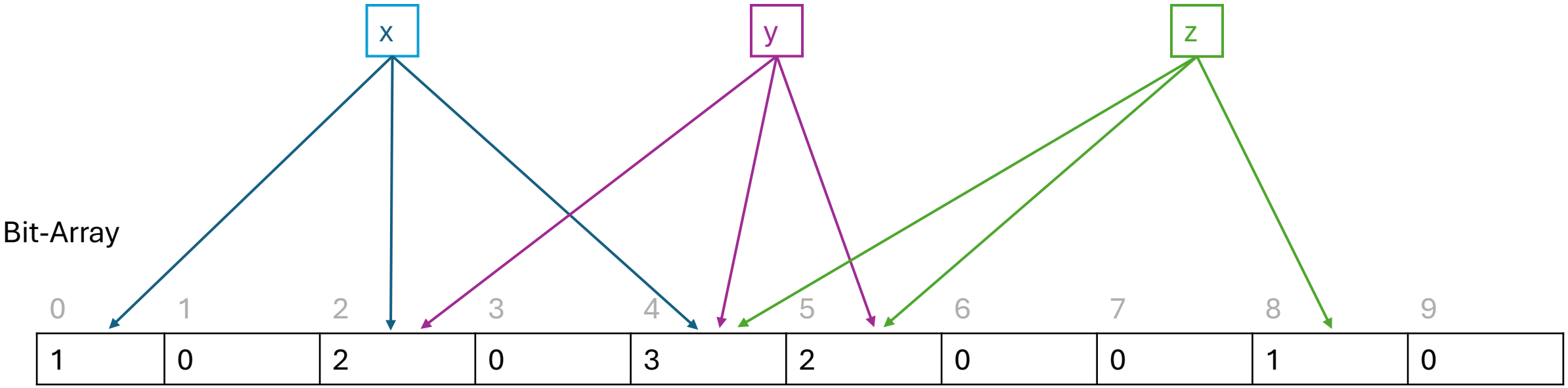
Bloom Filter Probleme

klassischer Bloom Filter unterstützt kein Löschen

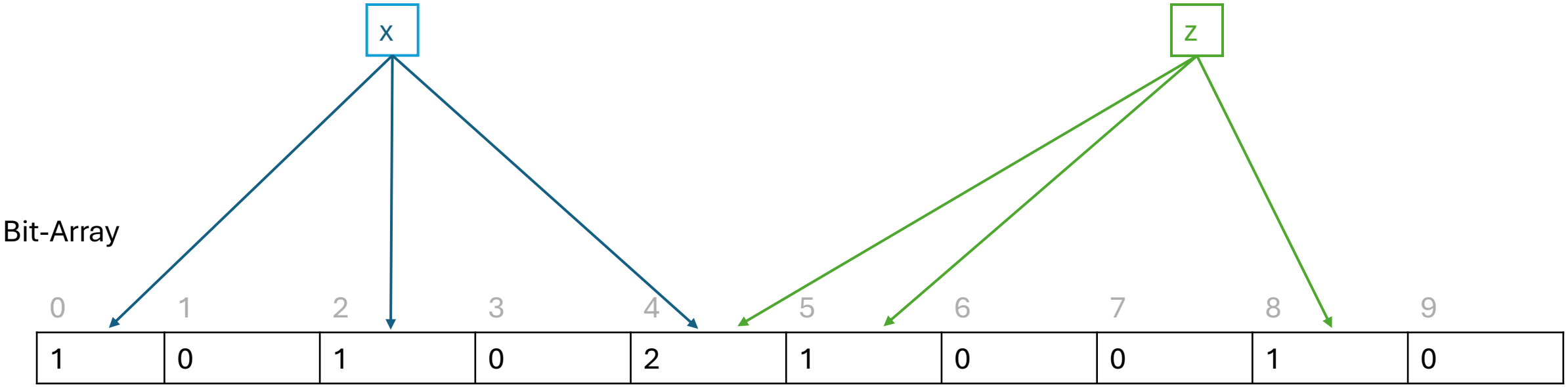
- ➢ Elemente können verloren gehen

Die Lösung: Counting Bloom Filter

Counting Bloom Filter



Counting Bloom Filter



Counting Bloom Filter

Löschproblem zwar gelöst, aber:

- Speicherverbrauch massiv höher
  - ➢ benötigt 4mal mehr Speicher
- Zähler können überlaufen
  - ➢ größerer Zähler möglich, aber mehr Speicher nötig

Bloom Filter Probleme

Größenplanung

- Größe vorher festlegen

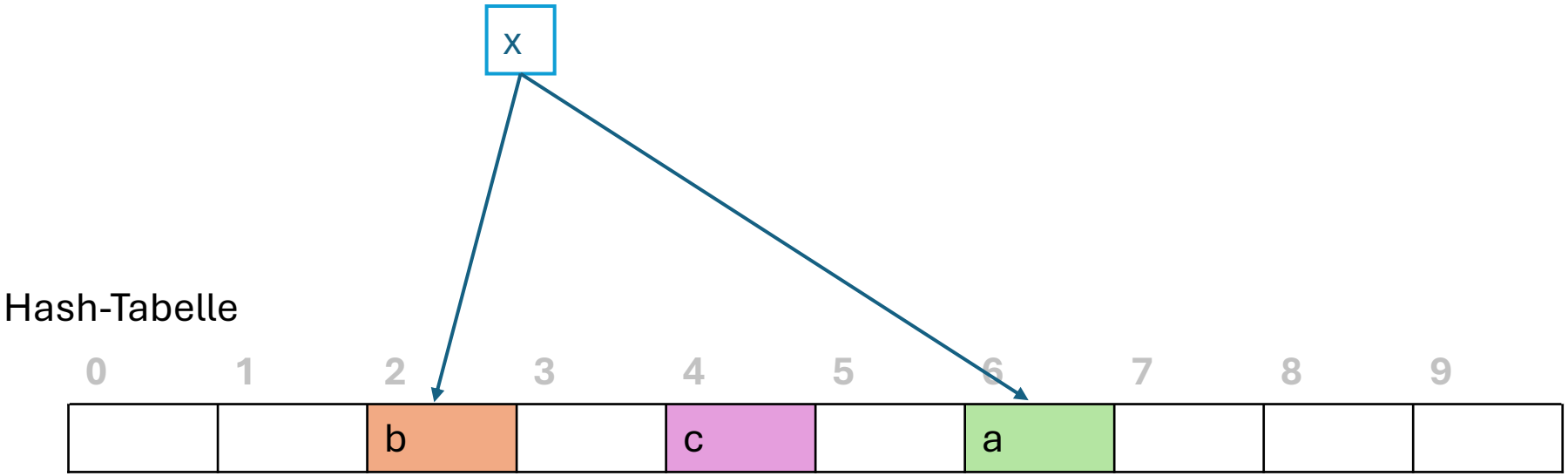
Die Lösung: Scalable Bloom Filter

Scalable Bloom Filter

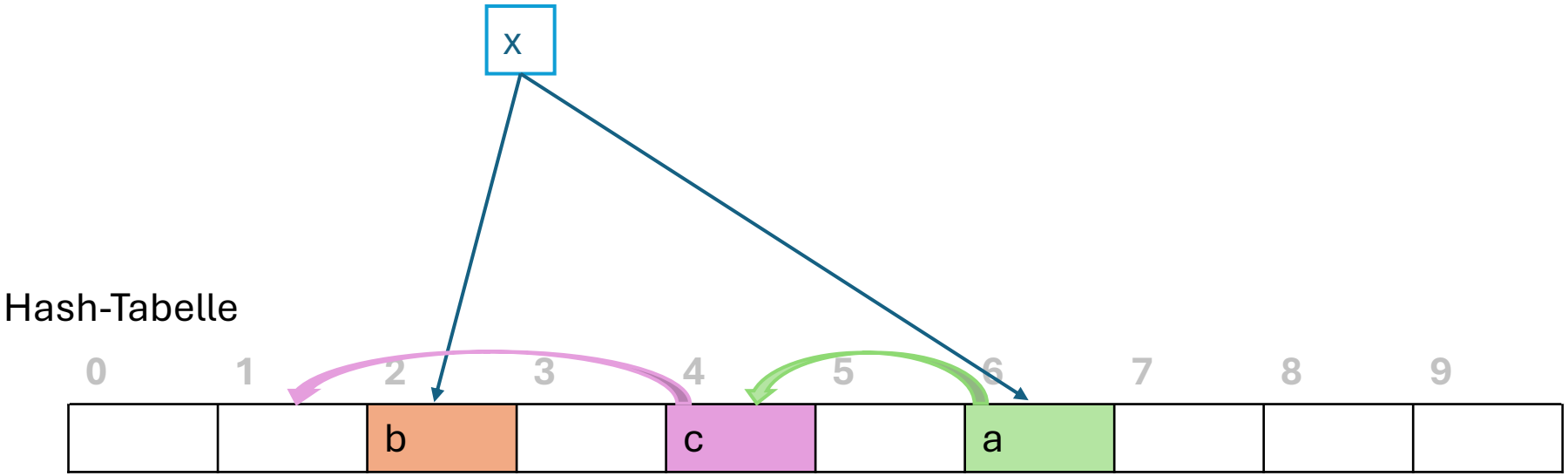
- mehrere normale Bloom Filter nacheinander

- kann beliebig wachsen, aber:
  - ➢Abfragen werden langsamer

Cuckoo Filter

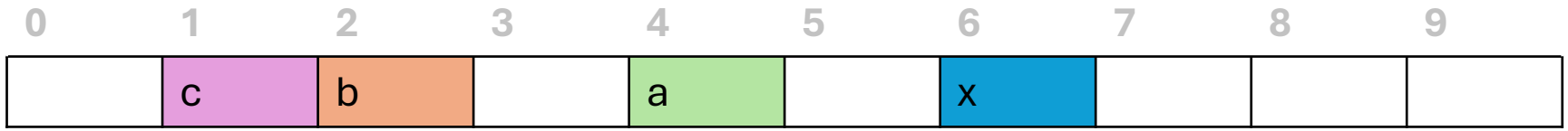


Cuckoo Filter



Cuckoo Filter

Hash-Tabelle



Cuckoo Filter

Es gibt Vor- und Nachteile:

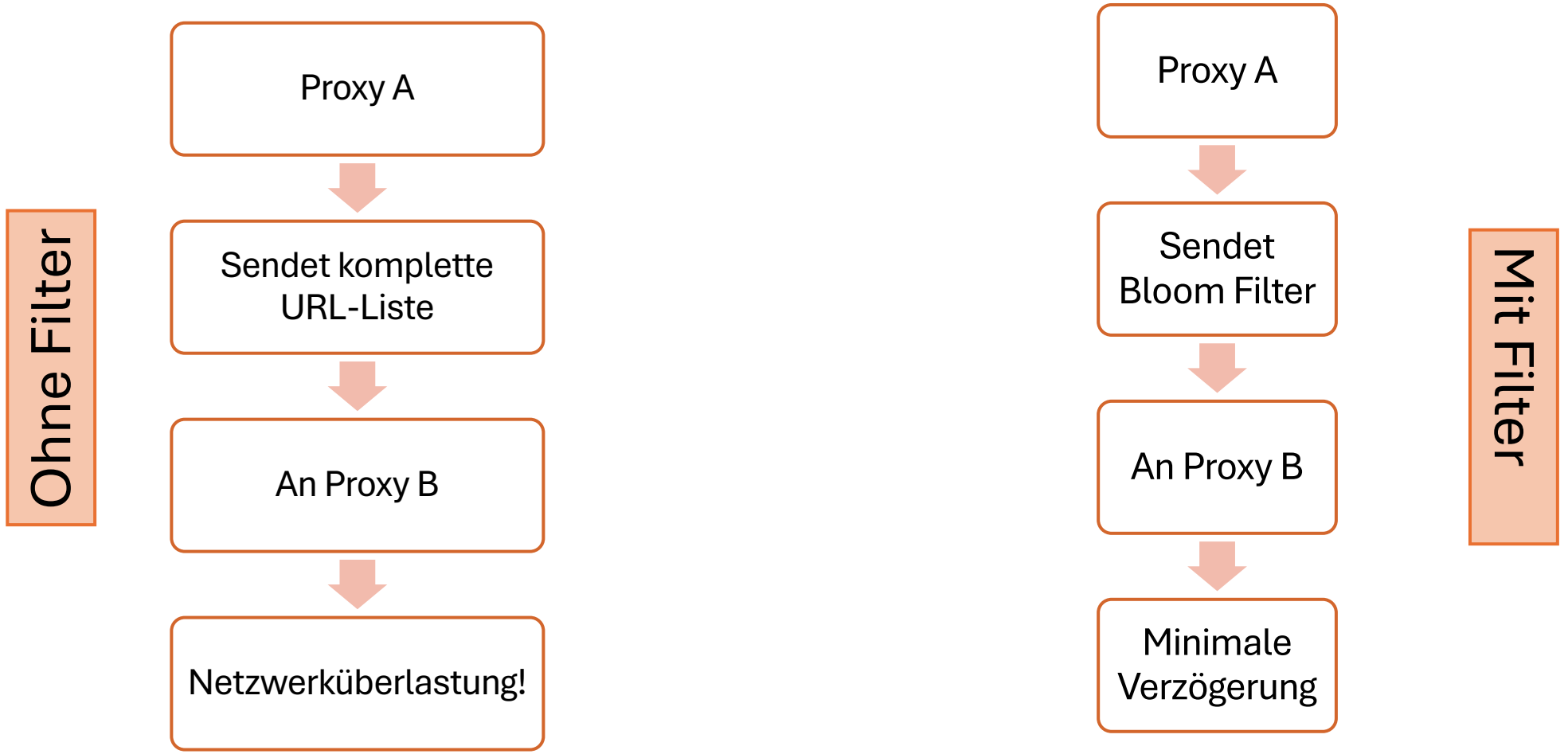
+ Elemente können problemlos gelöscht werden +schnelle Abfragen	- Bei hoher Auslastung lange Verdrängungskette

Wann Bloom Filter sinnvoll sind

Eignen sich gut für:

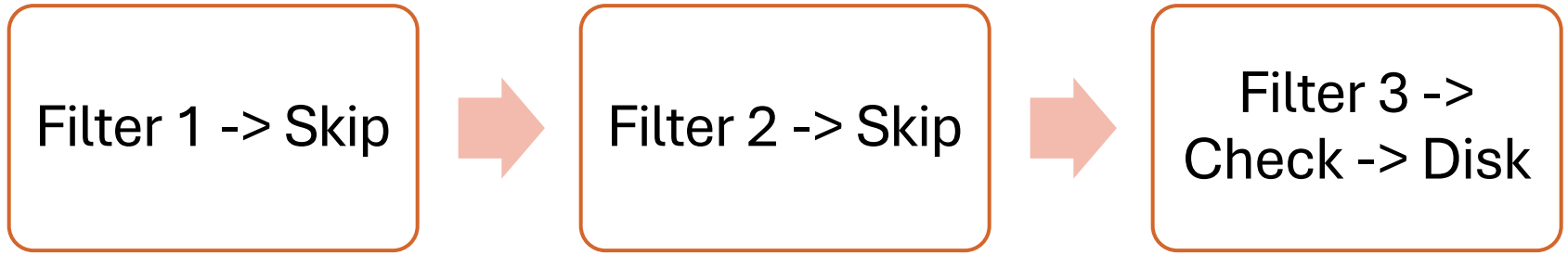
- Speicherplatz knapp oder teuer
- kleine Fehlerraten akzeptabel
- sehr große Datenmengen verarbeitet werden
- Vorfilter von teuren Operationen

Anwendungsbeispiel – Web-Proxy-Caching



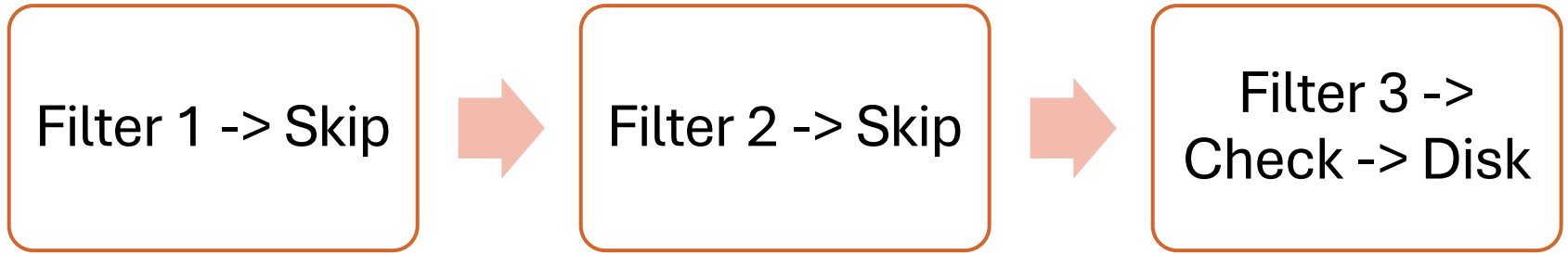
Anwendungsbeispiel – Google Bigtable

- Anfrage: "Schlüssel X?"



Anwendungsbeispiel – Google Bigtable

- Anfrage: "Schlüssel X?"



- Ersparnis: 2 Festplattenzugriffe

Weitere Anwendungen

- Web & Sicherheit
  - Google Chrome: Safe Browsing
  - "Have I been Pwned": Passwort-Checks
- Weitere Datenbanksysteme
  - Apache Cassandra
  - LevelDB