

# Bloom Filter

Jambura Anna, Pürstinger Kathrin,  
Schnappauf Franziska, Thiele Coco

24. Februar 2026

## **Zusammenfassung**

passend auszufüllen

# 6 1 Grundlagen und Motivation

## 7 1.1 Das Membership-Problem

8 Seien ein beliebiges Element  $x$  und eine Menge  $S$  gegeben. Das Membership-Problem ist  
9 eine Bezeichnung für die Fragestellung: „Ist das Element  $x$  Teil der Menge  $S$ ?“ Diese Frage  
10 tritt in vielen verschiedenen Bereichen und Anwendungen auf. Einige Beispiele dafür  
11 sind Datenbankenabfragen und URL-Caching in Web-Browsern. Klassische Ansätze, wie  
12 Listen, Hashtabellen oder Suchbäume liefern eine exakte Antwort auf die Frage, jedoch  
13 benötigen sie alle entsprechend viel Zeit und Speicherplatz.

## 14 1.2 Lösungsansatz - Bloomfilter

15 Bloomfilter wurden 1970 von Burton H. Bloom entwickelt, um den hohen Ressourcen-  
16 bedarf zu umgehen. Sie sind probabilistische Datenstrukturen, das bedeutet sie arbeiten  
17 mit Wahrscheinlichkeiten anstatt absoluter Sicherheit.

18 Dabei erlauben sie False-Positives in einem begrenzten Ausmaß. Ein Filter kann also  
19 fälschlicherweise melden, das Element  $x$  sei Teil der Menge  $S$ , auch wenn dies nicht der  
20 Fall ist. Umgekehrt sind False-Negatives jedoch ausgeschlossen. Wenn  $x$  tatsächlich ein  
21 Element von  $S$  ist, wird das der Filter immer korrekt erkennen. Mit anderen Worten:  
22 Ein vorhandenes Element wird nie als „nicht vorhanden“ gemeldet.

## 23 1.3 Trade-off

24 Bloomfilter balancieren drei zentrale Faktoren. Neben der Reject-Time (Zeit zur Ab-  
25 lehnung von Nicht-Mitgliedern) und dem benötigten Speicherplatz, die auch in konven-  
26 tionellen Hashing-Methoden berücksichtigt werden müssen, wird hier auch die erlaubte  
27 Fehlerrate betrachtet. Der zentrale Trade-off ist dabei zwischen dem akzeptablen Anteil  
28 an False-Positives und der Speichereffizienz. Dieser ist bei der Implementierung eines  
29 Bloomfilters individuell konfigurierbar.

30 Durch die kontrollierte Fehlerwahrscheinlichkeit wird der Speicherbedarf bedeutend re-  
31 duziert, da er nicht von der Länge der Daten abhängt, sondern immer gleich viele Bits  
32 pro Element beträgt. Je niedriger die Fehlerrate gewählt ist, desto mehr Bits pro Element  
33 werden benötigt. Bloomfilter sind besonders hilfreich, wenn die Mehrheit der Anfragen  
34 nicht-existente Elemente betrifft – hier liefern sie schnell ein definitives „Nein“ auf die  
35 Membership-Frage.

## 2 Funktionsweise und Mathematische Grundlagen

### 2.1 Aufbau

Der Bloomfilter besteht auf einem  $m$ -stelligen Bitarray, welches initial mit Nullen befüllt wird. Weiters werden  $k$  unabhängige Hashfunktionen definiert. Diese verwendet man um die Elemente der gewünschten Menge zu hashen. Abhängig von ihrem Hashwert werden die Elemente dann an der entsprechenden Position im Array eingefügt. Um also jedes Element erfolgreich einzufügen, muss die Hashfunktion  $\text{mod } m$  angewandt werden. Somit erreicht man die Indizes 0 bis  $m - 1$ . [TRL12]

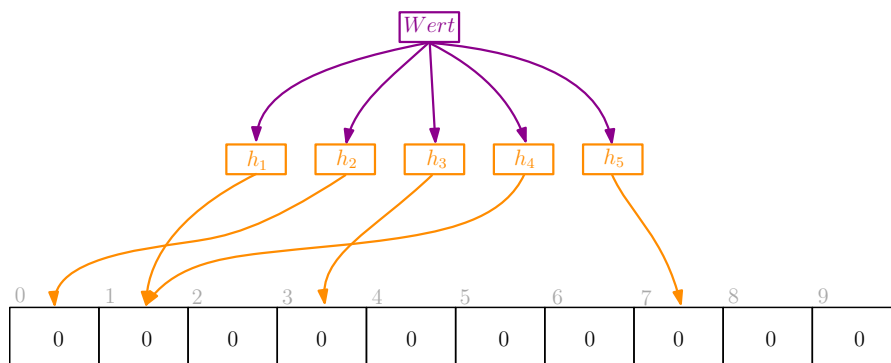


Abbildung 1: Visualisierung eines Bloomfilters

Da die Hashfunktionen keinem Sicherheitsstandard entsprechen, müssen keine kryptographischen Eigenschaften gelten. Kryptographische Eigenschaften bedeutet, minimale Eingabeänderungen müssen zu einer maximalen Änderung des Hashwerts führen. Die Eingabe darf nicht mittels der Hashfunktion wiederhergestellt werden können und zwei Eingaben haben fast unmöglich den selben Hashwert. Für Bloomfilter verwendet man schnelle und einfache Hashfunktionen, da die Effizienz im Vordergrund steht.

### 2.2 Einfügen/Suchen

#### Einfügen

Eine Menge  $S$  wird nun wie folgt in einem Bloomfilter eingefügt:  
Für jedes Element  $x \in S$  werden die Hash Werte aller  $k$  Hash Funktionen berechnet. Nun wird an diesen Positionen im Array die 0 auf eine 1 gesetzt. Sollte an einer dieser Positionen bereits eine 1 stehen, wird dies ignoriert. Dieser Vorgang wird für alle  $n$  Elemente der Menge  $S$  wiederholt.

## 58 2.2.1 Beispiel Einfügen

59 Betrachte folgende Menge  $S = \{2, 4, 9\}$  und einen Bloomfilter der Länge  $m = 10$  mit  
 60  $k = 3$  Hashfunktionen.

61 Als beispielhafte Hashfunktionen verwenden wir:  $h_1(x) = x \bmod 10$   $h_2(x) = (2x+3) \bmod$   
 62  $10$  und  $h_3(x) = (3x + 7) \bmod 10$ .

63 Nun berechnen wir die Hashwerte für jedes Element der Menge  $S$ :

- 64 • Für  $x = 2$ :

$$h_1(2) = 2 \bmod 10 = 2$$

$$h_2(2) = (2 \cdot 2 + 3) \bmod 10 = 7$$

$$h_3(2) = (3 \cdot 2 + 7) \bmod 10 = 3$$

- 65 • Für  $x = 4$ :

$$h_1(4) = 4 \bmod 10 = 4$$

$$h_2(4) = (2 \cdot 4 + 3) \bmod 10 = 1$$

$$h_3(4) = (3 \cdot 4 + 7) \bmod 10 = 9$$

- 66 • Für  $x = 9$ :

$$h_1(9) = 9 \bmod 10 = 9$$

$$h_2(9) = (2 \cdot 9 + 3) \bmod 10 = 1$$

$$h_3(9) = (3 \cdot 9 + 7) \bmod 10 = 4$$

67 Nun fügt man die Elemente in den Bloomfilter ein. Für das erste Element 2 werden die  
 68 Positionen 2, 7 und 3 auf 1 gesetzt. Daraus resultiert der folgende Bloomfilter:

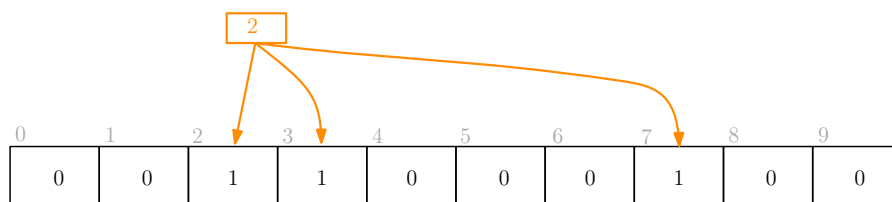


Abbildung 2: Bloomfilter nach Einfügen des Elements 2

69

70 Für das zweite Element 4 werden die Positionen 4, 1 und 9 auf 1 gesetzt.

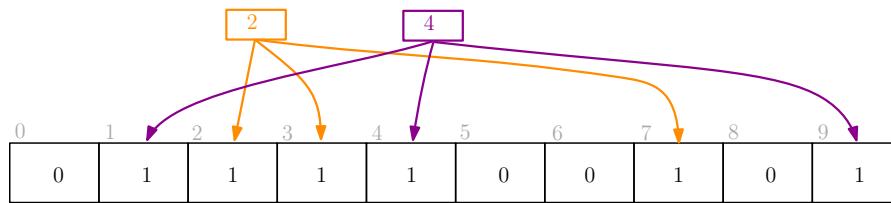


Abbildung 3: Bloomfilter nach Einfügen des Elements 4

71 Für das dritte Element 9 werden die Positionen 9, 1 und 4 auf 1 gesetzt. Da die Positionen  
 72 1, 4 und 9 bereits auf 1 gesetzt wurden, ändert sich der Bloomfilter nicht weiter.

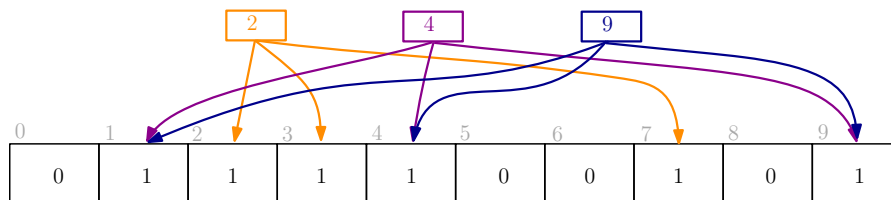


Abbildung 4: Bloomfilter nach Einfügen des Elements 9

73

## 74 Suchen

75 Um ein Element  $x$  in einem Bloomfilter zu suchen, werden dieselben Hashfunktionen wie  
 76 beim Einfügen verwendet. Die Hashwerte werden berechnet und an den entsprechenden  
 77 Positionen im Array geprüft. Wenn alle Positionen auf 1 gesetzt sind, so ist das Element  
 78 wahrscheinlich in der Menge enthalten. Wenn mindestens eine Position auf 0 gesetzt ist,  
 79 so ist das Element sicher nicht in der Menge enthalten. [TRL12]

### 80 2.2.2 Beispiel Suchen

81 Betrachten wir den zuvor erstellten Bloomfilter und suchen nach dem Element 4. Be-  
 82 rechnen wir die Hashwerte für 4:

$$h_1(4) = 4 \bmod 10 = 4$$

$$h_2(4) = (2 \cdot 4 + 3) \bmod 10 = 1$$

$$h_3(4) = (3 \cdot 4 + 7) \bmod 10 = 9$$

83 Nun prüfen wir die Positionen 4, 1 und 9 im Bloomfilter. Alle drei Positionen sind auf  
 84 1 gesetzt, daher ist das Element 4 wahrscheinlich in der Menge enthalten.

85 Betrachten wir nun das Element 5 und berechnen die Hashwerte:

$$h_1(5) = 5 \bmod 10 = 5$$

$$h_2(5) = (2 \cdot 5 + 3) \bmod 10 = 3$$

$$h_3(5) = (3 \cdot 5 + 7) \bmod 10 = 2$$

86 Nun prüfen wir die Positionen 5, 3 und 2 im Bloomfilter. Die Position 2 ist auf 0 gesetzt,  
87 daher ist das Element 5 sicher nicht in der Menge enthalten.

88 Ein wichtiger Aspekt des Bloomfilters ist, dass er fälschlicherweise angeben kann, dass  
89 ein Element in der Menge enthalten ist, obwohl es tatsächlich nicht vorhanden ist. Dies  
90 wird als *False Positive* bezeichnet. Wenn alle Positionen, die durch die Hashfunktionen  
91 eines Elements angegeben werden, auf 1 gesetzt sind, obwohl das Element nicht in der  
92 Menge enthalten ist, führt dies zu einem False Positive. Ein Beispiel hierfür wäre das  
93 Element 12:

$$h_1(12) = 12 \bmod 10 = 2$$

$$h_2(12) = (2 \cdot 12 + 3) \bmod 10 = 7$$

$$h_3(12) = (3 \cdot 12 + 7) \bmod 10 = 3$$

94 Die Positionen 2, 7 und 3 sind alle auf 1 gesetzt, obwohl das Element 12 nicht in der  
95 Menge enthalten ist. Daher würde der Bloomfilter fälschlicherweise angeben, dass 12 in  
96 der Menge enthalten ist.

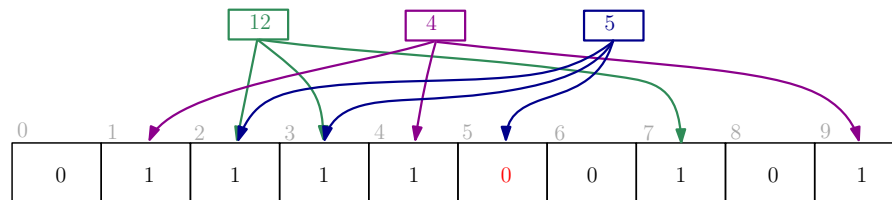


Abbildung 5: Suchen nach den Elementen 4, 5 und 12 im Bloomfilter

## 97 2.3 Formeln zur Evaluierung

### 98 2.3.1 False Positive Probability

99 Zur Erstellung des optimalen Bloomfilter ist es wichtig, die False Positive Probability  
100 (FPP) zu berechnen. Diese gibt an, wie wahrscheinlich es ist, dass der Bloomfilter fälsch-  
101 licherweise angibt, dass ein Element in der Menge enthalten ist, obwohl es tatsächlich  
102 nicht vorhanden ist. Laut [TRL12] entsteht die Formel zur Berechnung aus folgenden  
103 Komponenten:

104 Unter der Annahme, dass die Hashfunktionen unabhängig und gleichverteilt sind, ergibt  
 105 sich die Wahrscheinlichkeit dass ein bestimmtes der  $m$  Bits nicht gesetzt ist durch:

$$1 - \frac{1}{m} \quad (1)$$

106 Weiters werden nun die  $k$  Hashfunktionen mitbetrachtet, immer noch für den Fall, dass  
 107 ein bestimmtes Bit nicht gesetzt ist.

$$\left(1 - \frac{1}{m}\right)^k \quad (2)$$

108 Nun werden die  $n$  Elemente der Menge  $S$  betrachtet, welche in den Bloomfilter eingefügt  
 109 werden.

$$\left(1 - \frac{1}{m}\right)^{kn} \quad (3)$$

110 Die Wahrscheinlichkeit, dass ein bestimmtes Bit auf 1 gesetzt ist, ergibt sich aus der  
 111 Gegenwahrscheinlichkeit:

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \quad (4)$$

112 Da es bei Bloomfiltern um Membership-Tests geht, muss die Wahrscheinlichkeit berech-  
 113 net werden, dass alle  $k$  Positionen eines Elements auf 1 gesetzt sind, obwohl das Element  
 114 nicht in der Menge enthalten ist.

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \quad (5)$$

115 Aus der Formel lässt sich schließen, dass je **größer**  $m$  gewählt wird, desto **kleiner** wird  
 116 die False Positive Probability. Je **größer**  $n$  gewählt wird, desto **größer** wird die False  
 117 Positive Probability.

118 Da die False Positive Probability so klein wie möglich gehalten werden soll, ist auch  
 119 die Wahl der Anzahl Hashfunktionen von großer Bedeutung. Setzt man die Formel für  
 120 die False Positive Probability gleich 0 und löst sie nach  $k$  auf, erhält man die optimale  
 121 Anzahl an Hashfunktionen:

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n} \quad (6)$$

## 122 3 Anwendungsbeispiele

### 123 3.1 Web-Proxy-Caching

124 In verteilten Peer-to-Peer Netzwerken arbeiten mehrere Proxy-Server zusammen und  
125 tauschen sich untereinander aus. Bei einer Anfrage nach einer Webseite sucht ein Proxy  
126 zunächst im eigenen Cache, ob er diese bereits gespeichert hat. Wenn das nicht der Fall  
127 ist, spricht man von einem Cache-Miss und es wird geprüft, ob sich die Webseite im  
128 Cache eines anderen Proxys befindet. Wird sie hier gefunden, wird die Anfrage an den  
129 entsprechenden Proxy weitergeleitet, anstatt die Seite direkt aus dem Web zu laden.

130 Damit dieses System funktioniert, muss jeder Proxy über den Inhalt der Caches aller  
131 anderen Proxies Bescheid wissen. Um den enormen Netzwerkverkehr, der beim wieder-  
132 holten Austausch der kompletten URL-Listen entstehen würde, zu vermeiden, kommen  
133 hier Bloomfilter zum Einsatz. Im Summary Cache Protokoll tauschen Proxies periodisch  
134 Bloomfilter untereinander aus, die den Inhalt ihres Caches zusammenfassen. Wenn nun  
135 ein Cache-Miss auftritt, werden die Bloomfilter jener anderen Proxies konsultiert, die  
136 ein positives Ergebnis versprechen und die Anfrage wird entsprechend weitergeleitet.

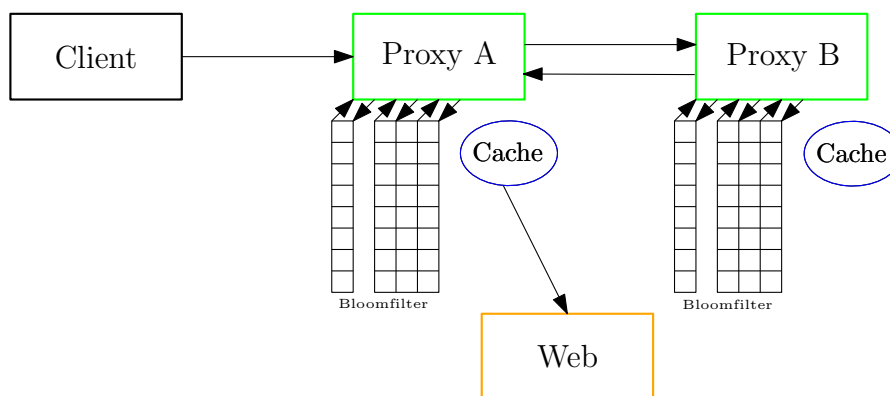


Abbildung 6: Web-Proxy-Caching mit Bloomfiltern

137 Hierbei können False-Positives auftreten, wobei es dann zu einer minimalen Verzögerung  
138 kommt. Die massive Reduktion des Netzwerkverkehrs durch den Bloomfilter überwiegt  
139 diesen Nachteil bei Weitem. Das Summary Cache Protokoll wird beispielsweise im Web-  
140 Proxy-Cache „Squid“ eingesetzt.

### 141 3.2 Google Bigtable

142 Bloomfilter werden oft in Datenbanksystemen verwendet, wobei Google Bigtable ein  
143 bekanntes Beispiel hierfür ist. Bigtable speichert die Daten auf der Festplatte in Sorted-  
144 String-Tables (SSTables). Wenn eine Leseoperation durchgeführt werden soll, müssen



145 potenziell mehrere dieser Tables durchsucht werden, bis die gewünschten Daten gefun-  
146 den werden. Da jede Table auf der Festplatte liegt, verursacht jeder Zugriff auf eine  
147 SSTable auch einen teuren Festplattenzugriff. Besonders problematisch im Bezug auf  
148 die benötigten Ressourcen wird dies bei Abfragen nach nicht-existenten Daten.

149 Kommen jetzt die Bloomfilter zum Einsatz, ändert sich dies drastisch. Für jede SSTable  
150 wird ein Bloomfilter im Hauptspeicher gehalten, der Auskunft über deren Inhalt gibt.  
151 Vor einem Festplattenzugriff wird also der Filter befragt, ob die gesuchten Daten in  
152 der Table enthalten sind. Bei einem positiven Ergebnis wird der Zugriff durchgeführt,  
153 ansonsten kann er eingespart werden.

### 154 3.2.1 Beispiel Anfrage

155 Angenommen es wird eine Anfrage auf den Schlüssel  $X$  gestellt und auf der Festplatte  
156 liegen drei SSTables. Ohne Verwendung von Bloomfiltern müssten alle drei Tables abge-  
157 rufen und durchsucht werden, also drei Festplattenzugriffe durchgeführt werden. Unter  
158 Einsatz von Bloomfiltern werden jedoch zuerst diese konsultiert. Die ersten beiden Fil-  
159 ter könnten melden, dass Schlüssel  $X$  jeweils nicht in SSTable 1 bzw. SSTable 2 liegt,  
160 sie können also beide übersprungen werden. Filter 3 sagt jetzt, dass sich  $X$  in Table 3  
161 befinden könnte – dieser Zugriff wird durchgeführt. Demzufolge wurde nur ein Festplat-  
162 tenzugriff durchgeführt, bis der gesuchte Schlüssel  $X$  gefunden wurde, das bedeutet eine  
163 Ersparnis von zwei Zugriffen durch die Verwendung von Bloomfiltern.

## 164 3.3 Weitere Anwendungen

165 Heute kommen Bloomfilter in zahlreichen Systemen zum Einsatz. Google Chrome nutzt  
166 sie für Safe-Browsing zur Malware-Erkennung. Sicherheitsdienste wie „Have I Been Pw-  
167 ned“, prüfen mit ihrer Hilfe, ob Passwörter kompromittiert wurden, ohne dabei die kom-  
168 plette Leak-Datenbank lokal speichern zu müssen. Neben Google Bigtable setzen auch  
169 weitere Datenbanksysteme, wie Apache Cassandra und LevelDB auf die Vorteile von  
170 Bloomfiltern, um unnötige Festplattenzugriffe zu vermeiden.

## 171 Literatur

- 172 [TRL12] Sasu Tarkome, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory  
173 and practice of bloom filters for distributed systems. *IEEE Communications*  
174 *Surveys & Tutorials*, 14(1):131–155, 2012.