

<sup>1</sup>

# Bloom Filter

<sup>2</sup>

Jambura Anna, Pürstinger Kathrin,  
Schnappauf Franziska, Thiele Coco

<sup>3</sup>

23. Februar 2026

<sup>4</sup>

## **Zusammenfassung**

<sup>5</sup>

passend auszufüllen

## <sup>6</sup> 1 Grundlagen und Motivation

### <sup>7</sup> 1.1 Das Membership-Problem

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

### <sup>14</sup> 1.2 Lösungsansatz

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

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

### <sup>23</sup> 1.3 Trade-off

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

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

## 36 2 Funktionsweise und Mathematische Grundlagen

### 37 2.1 Aufbau

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

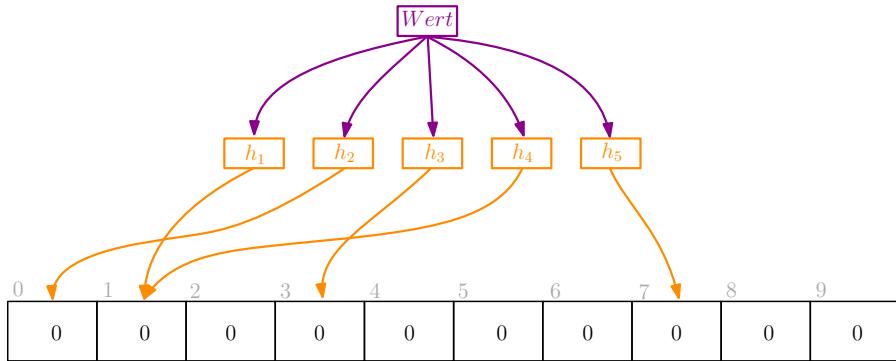


Abbildung 1: Visualisierung eines Bloomfilters

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

### 51 2.2 Einfügen/Suchen

#### 52 Einfügen

53 Eine Menge  $S$  wird nun wie folgt in einem Bloomfilter eingefügt:

54 Für jedes Element  $x \in S$  werden die Hash Werte aller  $k$  Hash Funktionen berechnet.  
55 Nun wird an diesen Positionen im Array die 0 auf eine 1 gesetzt. Sollte an einer dieser  
56 Positionen bereits eine 1 stehen, wird dies ignoriert. Dieser Vorgang wird für alle  $n$   
57 Elemente der Menge  $S$  wiederholt.

<sup>58</sup> **2.2.1 Beispiel Einfügen**

- <sup>59</sup> Betrachte folgende Menge  $S = \{2, 4, 9\}$  und einen Bloomfilter der Länge  $m = 10$  mit  
<sup>60</sup>  $k = 3$  Hashfunktionen.  
<sup>61</sup> Als beispielhafte Hashfunktionen verwenden wir:  $h_1(x) = x \bmod 10$   $h_2(x) = (2x+3) \bmod 10$  und  
<sup>62</sup>  $h_3(x) = (3x+7) \bmod 10$ .  
<sup>63</sup> Nun berechnen wir die Hashwerte für jedes Element der Menge  $S$ :

- <sup>64</sup> • Für  $x = 2$ :

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

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

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

- <sup>65</sup> • Für  $x = 4$ :

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

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

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

- <sup>66</sup> • Für  $x = 9$ :

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

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

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

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

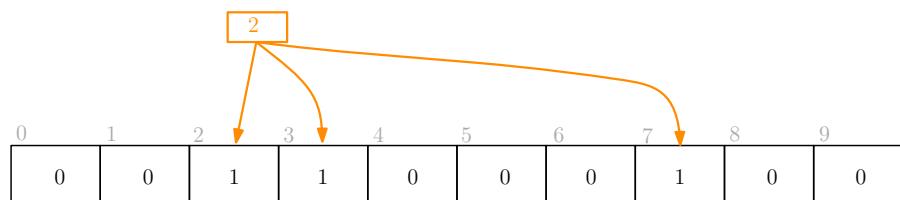


Abbildung 2: Bloomfilter nach Einfügen des Elements 2

<sup>69</sup>

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

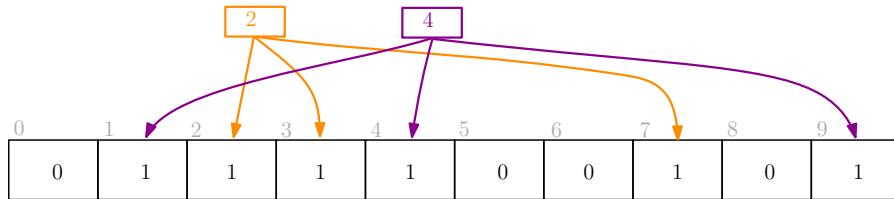


Abbildung 3: Bloomfilter nach Einfügen des Elements 4

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.

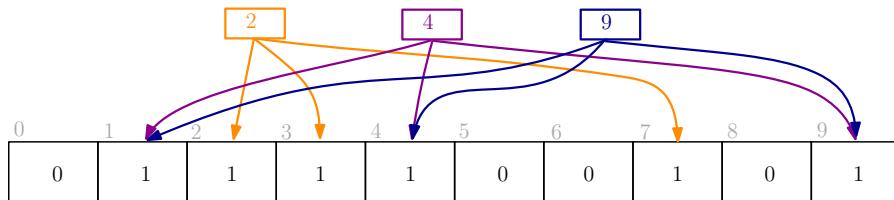


Abbildung 4: Bloomfilter nach Einfügen des Elements 9

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:

$$\begin{aligned} h_1(5) &= 5 \bmod 10 = 5 \\ h_2(5) &= (2 \cdot 5 + 3) \bmod 10 = 3 \\ h_3(5) &= (3 \cdot 5 + 7) \bmod 10 = 2 \end{aligned}$$

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:

$$\begin{aligned} h_1(12) &= 12 \bmod 10 = 2 \\ h_2(12) &= (2 \cdot 12 + 3) \bmod 10 = 7 \\ h_3(12) &= (3 \cdot 12 + 7) \bmod 10 = 3 \end{aligned}$$

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.

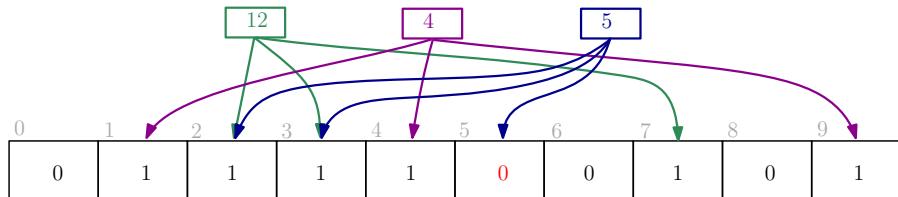


Abbildung 5: Suchen nach den Elementen 4, 5 und 12 im Bloomfilter

## 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

## <sup>122</sup> 3 Anwendungsbeispiele

### <sup>123</sup> 3.1 Web-Proxy-Caching

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

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

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

### <sup>141</sup> 3.2 Google Bigtable

<sup>142</sup> Bloomfilter werden oft in Datenbanksystemen verwendet, wobei Google Bigtable ein  
<sup>143</sup> bekanntes Beispiel hierfür ist. Bigtable speichert die Daten auf der Festplatte in Sorted-  
<sup>144</sup> String-Tables (SSTables). Wenn eine Leseoperation durchgeführt werden soll, müssen  
<sup>145</sup> potenziell mehrere dieser Tables durchsucht werden, bis die gewünschten Daten gefun-  
<sup>146</sup> den werden. Da jede Table auf der Festplatte liegt, verursacht jeder Zugriff auf eine  
<sup>147</sup> SSTable auch einen teuren Festplattenzugriff. Besonders problematisch im Bezug auf  
<sup>148</sup> die benötigten Ressourcen wird dies bei Abfragen nach nicht-existenten Daten.

<sup>149</sup> Kommen jetzt die Bloomfilter zum Einsatz, ändert sich dies drastisch. Für jede SSTable  
<sup>150</sup> wird ein Bloomfilter im Hauptspeicher gehalten, der Auskunft über deren Inhalt gibt.  
<sup>151</sup> Vor einem Festplattenzugriff wird also der Filter befragt, ob die gesuchten Daten in  
<sup>152</sup> der Table enthalten sind. Bei einem positiven Ergebnis wird der Zugriff durchgeführt,  
<sup>153</sup> 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 liegen drei SSTables. Ohne Verwendung von Bloomfiltern müssten alle drei Tables abgerufen und durchsucht werden, also drei Festplattenzugriffe durchgeführt werden.  
156 Unter Einsatz von Bloomfiltern werden jedoch zuerst diese konsultiert. Filter 1 könnte  
157 beispielsweise melden, dass Schlüssel  $X$  definitiv nicht in SSTable 1 vorhanden ist, dann  
158 kann diese übersprungen werden. Filter 2 meldet jetzt das gleiche für SSTable 2, also  
159 wird diese auch übersprungen. Filter 3 sagt jetzt, dass sich  $X$  in Table 3 befinden könnte  
160 – der Zugriff wird durchgeführt. Demzufolge wurde nur ein Festplattenzugriff durchge-  
161 führt, bis der gesuchte Schlüssel  $X$  gefunden wurde, das bedeutet eine Ersparnis von  
162 zwei Zugriffen durch die Verwendung von Bloomfiltern.  
163  
164

165 **3.3 Weitere Anwendungen**

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

## <sup>173</sup> Literatur

- <sup>174</sup> [TRL12] Sasu Tarkome, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory  
<sup>175</sup> and practice of bloom filters for distributed systems. *IEEE Communications*  
<sup>176</sup> *Surveys & Tutorials*, 14(1):131–155, 2012.