

# MASTERARBEIT

## **Organisation von Bloom-Filters zur effizienten $k$ -nächste-Nachbarn-Suche in kontextzentrischen sozialen Netzen**

Judith Greif

Entwurf vom 8. Juli 2016





# MASTERARBEIT

## **Organisation von Bloom-Filtern zur effizienten $k$ -nächste-Nachbarn-Suche in kontextzentrischen sozialen Netzen**

Judith Greif

Aufgabenstellerin: Prof. Dr. Claudia Linnhoff-Popien

Betreuer:  
Mirco Schönfeld  
Dr. Martin Werner

Abgabetermin: 26. Juli 2016





Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 26. Juli 2016

.....  
*(Unterschrift der Kandidatin)*



# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
<b>2. Hintergrund</b>	<b>3</b>
2.1. Kontextzentrische soziale Netze . . . . .	3
2.2. Bloom-Filter . . . . .	4
2.2.1. Distanzmaße . . . . .	5
2.2.2. Teil- und Obermengenbeziehung . . . . .	6
2.2.3. Hashfunktionen . . . . .	7
2.2.4. Bloom-Filter-Varianten und Anwendungen . . . . .	8
2.3. Indexstrukturen . . . . .	8
2.3.1. B-Bäume . . . . .	9
2.3.2. B <sup>+</sup> -Bäume . . . . .	10
<b>3. Verwandte Themen</b>	<b>13</b>
3.1. Implementierung von Bloom-Filtern . . . . .	13
3.2. Netzwerkanwendungen mit Bloom-Filtern . . . . .	15
3.3. Indexstrukturen für Bloom-Filter . . . . .	16
3.4. <i>k</i> -nächste-Nachbarn-Suche . . . . .	18
<b>4. Implementierung</b>	<b>21</b>
4.1. BloomFilterTree . . . . .	21
4.1.1. Aufbau . . . . .	21
4.1.2. Umsetzung . . . . .	22
4.1.3. Einfügen . . . . .	23
4.1.4. <i>k</i> -nächste-Nachbarn-Suche . . . . .	25
4.2. Alternative Ansätze . . . . .	27
<b>5. Evaluation</b>	<b>29</b>
5.1. Datensatz . . . . .	29
5.2. Versuchsaufbau . . . . .	30
5.3. Ergebnisse . . . . .	31
5.4. Interpretation . . . . .	40
<b>6. Zusammenfassung und Ausblick</b>	<b>43</b>
<b>A. Anhang</b>	<b>45</b>
<b>Abbildungsverzeichnis</b>	<b>53</b>
<b>Literaturverzeichnis</b>	<b>55</b>



# 1. Einleitung

Die digitale Kommunikation hat im letzten Jahrzehnt einen rapiden Wandel erlebt. Soziale Online-Netzwerke<sup>1</sup> haben an Bedeutung gewonnen und zu neuen Kommunikationsmustern im Internet geführt. Sofortnachrichtendienste und Instant-Messenger ersetzen zunehmend Kommunikationsformen wie SMS und Telefonie. Das Smartphone hat das Mobiltelefon als mobiles Endgerät fast vollständig abgelöst. Feststehende Desktop-Rechner mit einem gleich bleibenden Netzzugang sind außerhalb von Firmen und Bildungseinrichtungen rückläufig. Dagegen sind Notebooks und Tablets weiterhin auf Erfolgsszug<sup>2</sup>.

Die Kommunikation im Internet ist jedoch nach wie vor Ende-zu-Ende- beziehungsweise adressbasiert. Das spiegelt sich im Aufbau der bestehenden sozialen Online-Netzwerke wider: Kommunikation basiert darin auf Online-Freundschaft. Mobilität und spezifischer Kontext der Mitglieder werden kaum berücksichtigt. In der Realität verlieren die Webbrowser-Schnittstellen der sozialen Netzwerke jedoch an Bedeutung. So veröffentlichte Facebook 2014 eine Studie zum Nutzerverhalten von US-Bürgern in einer Multigeräte-Welt<sup>3</sup>. Danach nutzten 60% der Erwachsenen in den USA täglich mindestens zwei Endgeräte, knapp 25% sogar drei Geräte. Mehr als 40% begannen den Tag mit einem Gerät und beendeten ihn mit einem anderen. Das Smartphone ist dabei das Gerät, das am häufigsten mitgenommen wird und eine zentrale Rolle in der Kommunikation per E-Mail und in sozialen Netzen einnimmt. Das Szenario, in dem Alice vor ihrem Desktop-Rechner zu Hause oder im Büro sitzt und über die Webbrowser-Schnittstelle Nachrichten an ihren Facebook-Freund Bob schreibt, gehört demnach der Vergangenheit an. Stattdessen verwendet Alice wohl eher ein Tablet, ein Notebook und ein Smartphone und kommuniziert mit Bob je nach Aufenthaltsort und Kontext ganz unterschiedlich.

So stellt sich die Frage nach einer Neuorientierung der sozialen Online-Netze: Nicht nur in der praktischen Umsetzung, also durch Schaffung unterschiedlicher Schnittstellen und neuer Funktionalitäten, sondern im Sinne eines tatsächlichen Paradigmenwechsels. Ein kontextzentrisches soziales Netz basiert in seiner Struktur nicht auf Ende-zu-Ende-Kommunikation, adressbasiertem Routing und einem gleich bleibenden Netzzugang. Kommunikation beruht allein auf Kontext-Ähnlichkeit statt auf virtueller Freundschaft. Diese Überlegungen sind z.B. in das soziale Online-Netz AMBIENCE eingeflossen. Nachrichten werden darin auf Grund von zeitlicher und räumlicher Ähnlichkeit ausgetauscht, Sender und Empfänger bleiben weitgehend anonym. Ein solches Netz erfordert eine neue Kommunikationsstruktur. Nachrichten werden nicht aktiv von einem Sender für einen spezifischen Empfänger verfasst und an ihn verschickt. Stattdessen kann ein Sender eine Nachricht verfassen und z.B. an einem WiFi-Access Point hinterlegen. Mitglieder des Netzwerks, die sich in der Nähe des Access Points aufhalten, können die dort vorhan-

<sup>1</sup>Der englische Begriff hierfür lautet *Online Social Network (OSN)*. Die Begriffe *Soziales Netz*, *Soziales Online-Netz* und *Soziales Online-Netzwerk* werden im Folgenden synonym verwendet.

<sup>2</sup>Laut Analysen der Marktforschungsinstitute Gartner und IDC, vgl. z.B. [http://www.golem.de/news\\_pc-markt-absatz-von-pcs-geht-weiter-erheblich-zurueck-1601-118505.html](http://www.golem.de/news_pc-markt-absatz-von-pcs-geht-weiter-erheblich-zurueck-1601-118505.html).

<sup>3</sup>Vgl. <https://www.facebook.com/business/news/Finding-simplicity-in-a-multi-device-world>.

## 1. Einleitung

denen Nachrichten mit gezielten Anfragen durchsuchen und die zum jeweiligen Kontext ähnlichsten Nachrichten abrufen.

Damit stellt sich die Frage: Wie lassen sich die Nachrichten an einem Host, also z.B. an einem WiFi-Access Point, so organisieren, dass die  $k$  ähnlichsten Nachrichten möglichst schnell und effizient gefunden werden? Wenn das soziale Netz wachsen und über den Status eines Prototypen hinaus erfolgreich sein soll, ist das von entscheidender Bedeutung. Mengentheoretisch betrachtet handelt es sich dabei um das Problem der  $k$ -nächste-Nachbarn-Suche, die zu einer Anfrage die  $k$  ähnlichsten Elemente einer Menge, hier bestehend aus den Nachrichten an einem Host, finden soll.

Damit verknüpft ist die Frage nach der Nachrichtenform. Wie können Multimedia-Nachrichten wie Bilder, Textdateien oder Links effizient, einheitlich und sicher vor unbefugtem Zugriff hinterlegt und verschickt werden? Zudem muss die Ähnlichkeit oder Unähnlichkeit von Nachrichten ermittelt werden können, d.h. der  $k$ -nächste-Nachbarn-Suche muss ein Ähnlichkeitsmaß zu Grunde liegen. AMBIENCE verwendet dazu eine Bloom-Filter-Konstruktion. Eine Nachricht wird als Menge von Zeichenketten aufgefasst, die mit geeigneten Hashfunktionen in ein Bit-Array fester Länge eingefügt werden. Ähnlichkeit zwischen Nachrichten ist damit als Ähnlichkeit zwischen Bloom-Filtern definiert. Nachrichten werden in Form von Bloom-Filtern kodiert, gespeichert und verglichen. Als Ähnlichkeitsmaß dient die Jaccard-Distanz, mit der sich die Ähnlichkeit von Mengen beschreiben lässt.

Die folgende Arbeit behandelt daher die Organisation von Bloom-Filtern zur effizienten  $k$ -nächste-Nachbarn-Suche in kontextzentrischen sozialen Netzen. Das folgende Kapitel 2 gibt einen Überblick über mengentheoretische und probabilistische Grundlagen, verwendete Datenstrukturen und ihre Nutzung in AMBIENCE. Kapitel 3 gibt einen Überblick über verwandte Arbeiten und Fragestellungen. Das entwickelte Verfahren wird anschließend in Kapitel 4 dargestellt. Kapitel 5 vergleicht die Implementierung mit dem bisherigen, nicht optimierten Ansatz. Im abschließenden Kapitel 6 wird ein Fazit gezogen und auf mögliche zukünftige Arbeiten eingegangen.

## 2. Hintergrund

Im Folgenden wird das Konzept des *kontextzentrischen sozialen Netzes* erläutert, das dieser Arbeit zu Grunde liegt. Anschließend werden mengentheoretische und probabilistische Grundlagen und Verfahren sowie Daten- und Indexstrukturen dargestellt, die Eingang in diese Arbeit gefunden haben. Es wird erläutert, inwiefern sie für das soziale Online-Netz AMBIENCE relevant sind oder darin verwendet werden.

### 2.1. Kontextzentrische soziale Netze

Mit Zunahme der mobilen Endgeräte und dem Erfolgszug des Smartphones lässt sich eine Tendenz beobachten, die von bestehenden sozialen Online-Netzwerken wenig abgebildet wird: Weg von adressbasiertem Routing und Ende-zu-Ende-Kommunikation hin zu *context-awareness* und *information-centric networking*.

Der Begriff context-awareness wurde von Schilit et al. 1994 geprägt und bezeichnet die Nutzung von Kontextinformationen als Informationsquelle für Anwendungen und Netzwerke<sup>1</sup>. Information-centric networking (ICN) bezeichnet ein neuartiges Konzept für Netzwerke, die nicht auf Ende-zu-Ende- oder Sender/Empfänger-Kommunikation basieren, sondern auf den im Netzwerk vorhandenen Informationen<sup>2</sup>. Der Begriff *Kontext* im Zusammenhang mit interaktiven Anwendungen wurde bereits in den 90er Jahren geprägt. Die klassische Definition von Dey und Abowd lautet:

Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application including the user and application themselves<sup>3</sup>.

Diese Definition lässt sich für soziale Online-Netze eingrenzen auf

[...] any information that can be used to infer aspects of the surroundings of an entity in a way, in which some applications may have interest. Surroundings include all information that could possibly impact the behaviour of the entity<sup>4</sup>.

Ein kontextzentrisches soziales Netz ist also ein soziales Online-Netz, das auf Kontext-Variablen wie Ort und Zeit als Informationsquellen basiert und in der Regel dezentral organisiert ist, z.B. durch eine Peer-to-Peer- statt einer Client/Server-Architektur. Kommunikation beruht darin allein auf Kontext-Ähnlichkeit, nicht auf Online-Freundschaft:

---

<sup>1</sup>Vgl. [SAW94]: 85.

<sup>2</sup>Vgl. [ADI<sup>+</sup>12] für eine ausführliche Darstellung.

<sup>3</sup>[DA99]: 306f..

<sup>4</sup>[WDS15]: 2.

## 2. Hintergrund

A context-centric online social network is an online social network in which the edges of the social graph are defined from context information and context matching algorithms. An edge between two profiles exists for a fixed information object if and only if the two profiles share the relevant context as defined by the publisher<sup>5</sup>.

**AMBIENCE** AMBIENCE ist ein soziales Online-Netzwerk, das 2015 als Prototyp implementiert wurde und sich an diesem neuen Paradigma orientiert. Die vorliegende Arbeit hat das Ziel, einen spezifischen Aspekt des Netzwerks zu optimieren, nämlich die Organisation der Nachrichten für  $k$ -nächste-Nachbarn-Anfragen an einen Host. Nachrichten und Anfragen werden in AMBIENCE als Bloom-Filter kodiert. Bei einer Anfrage wird also ein Anfrage-Filter mit einer Menge von Bloom-Filtern verglichen, die an einem Host gespeichert sind. Aktuell werden die Filter dort einfach als unsortierte Liste gespeichert. Die Laufzeit für  $k$ -nächste-Nachbarn-Anfragen an einen Host mit  $n$  Filtern liegt damit in  $O(n^2)$ . Dieses Laufzeitverhalten gilt es zu optimieren, wenn das Netzwerk wachsen und über den Status eines Prototypen hinaus erfolgreich sein soll.

### 2.2. Bloom-Filter

Ein Bloom-Filter ist eine probabilistische Datenstruktur zur Beschreibung von Mengen und wurde in der ursprünglichen From 1970 von Burton H. Bloom eingeführt<sup>6</sup>. Er besteht aus einem Bit-Array der festen Länge  $m$ , dessen Elemente zunächst alle auf 0 gesetzt sind. Das Einfügen von Informationsobjekten basiert auf der Berechnung einer festen Anzahl unabhängiger Hashfunktionen  $k$ , die positive Werte kleiner als  $m$  annehmen. Soll ein Objekt in den Filter eingefügt werden, werden seine Hashwerte berechnet und die entsprechenden Bits im Filter gesetzt<sup>7</sup>.

Die Hashwerte werden für Anfragen an den Filter verwendet. Ist ein Objekt im Filter enthalten, sind seine charakteristischen Bits gesetzt worden, d.h. man kann eine Anfrage nach seinen Hashwerten durchführen und mit großer Wahrscheinlichkeit ermitteln, ob es im Filter vorhanden ist. Sind ein oder mehrere Bits des Anfrageobjekts nicht gesetzt, ist es mit Sicherheit nicht im Filter vorhanden. Es gibt also keine falsch negativen Antworten. Allerdings kann es sein, dass ein Element nicht in den Filter eingefügt wurde, obwohl alle seine Bits gesetzt sind (falsch positive Antworten). Grund dafür ist die Kollisions-eigenschaft von Hashfunktionen, die ein großes Universum von Objekten auf einen sehr viel kleineren Wertebereich, hier die Länge des Bloom-Filters, abbilden. Es kann somit zu Kollisionen zwischen unterschiedlichen Informationsobjekten bzw. ihren charakteristischen Hashwerten kommen. Die Falsch-Positiv-Rate eines Bloom-Filters ist abhängig von  $m$ ,  $k$  und der Anzahl der eingefügten Elemente  $n$  und lässt sich berechnen als<sup>8</sup>

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

---

<sup>5</sup>[WDS15]: 4.

<sup>6</sup>Vgl. [Blo70].

<sup>7</sup>Vgl. [BM04]: 487.

<sup>8</sup>Vgl. ebd. für eine umfassende Darstellung.

Aus der Kollisionseigenschaft folgt auch, dass ein einmal eingefügtes Objekt nicht mehr aus einem Bloom-Filter entfernt werden kann. Das könnte offensichtlich zu falsch negativen Ergebnissen für andere Objekte führen.

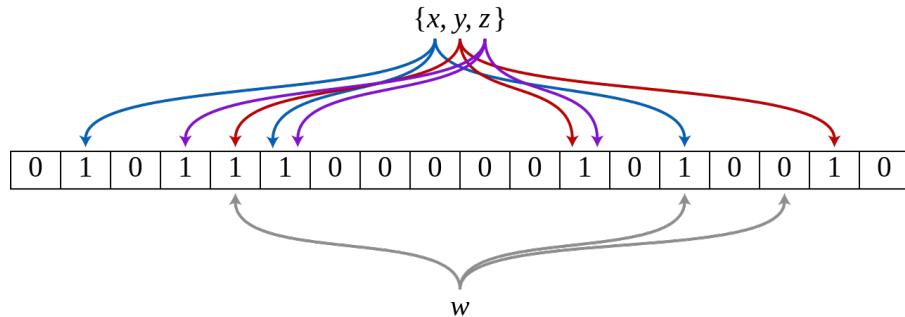


Abbildung 2.1.: Bloom-Filter, in den die Objekte  $x$ ,  $y$  und  $z$  eingefügt wurden. Das Objekt  $w$  ist nicht im Filter vorhanden.

### 2.2.1. Distanzmaße

Um die Ähnlichkeit zweier Mengen zu ermitteln, werden unterschiedliche Distanzmetriken oder Ähnlichkeitsmaße verwendet. Für den Vergleich von Bloom-Filtern und insbesondere für die  $k$ -nächste-Nachbarn-Suche muss ein geeignetes Ähnlichkeitsmaß zur Anwendung kommen. Bayardo et al. verwenden dazu die *Kosinus-Ähnlichkeit*<sup>9</sup>, Sakuma und Sato definieren die Ähnlichkeit von Bloom-Filtern als die Anzahl gleicher 1-Bits<sup>10</sup>. Ist diese für zwei Filter identisch, werden die Bit-Arrays negiert und die Anzahl gleicher 0-Bits ermittelt.

In AMBIENCE wird eine Abschätzung der *Jaccard-Distanz* zur Ermittlung der Ähnlichkeit von Bloom-Filtern verwendet. Die Jaccard-Distanz zwischen zwei Mengen  $A$  und  $B$  ist definiert als

$$J_\delta(A, B) = 1 - J(A, B) = \frac{|A \cap B| - |A \cup B|}{|A \cup B|},$$

wobei

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

den *Jaccard-Koeffizienten* bezeichnet. Die Jaccard-Distanz nimmt Werte im Bereich  $[0, 1]$  an. Identische Mengen haben eine Jaccard-Distanz von 0, Mengen ohne gemeinsame Elemente haben eine Jaccard-Distanz von 1. Für Bloom-Filter lässt sich die Jaccard-Distanz analog berechnen. Die Vereinigungsmenge zweier Bloom-Filter  $F$  und  $G$  lässt sich als bitweises logisches Oder, die Schnittmenge als bitweises logisches Und repräsentieren. Auch hier nimmt die Jaccard-Distanz offensichtlich Werte zwischen 0 und 1 an. Je ähnlicher die Filter sind, desto kleiner ist ihre Jaccard-Distanz. Die Jaccard-Distanz zwischen Bloom-Filtern ist, anders als die von Sakuma und Sato verwendete Distanzmetrik, nicht transitiv in dem Sinne, dass zwei Filter, die beide eine geringe Jaccard-Distanz zu einem dritten

<sup>9</sup>Vgl. [BMS07]: 131.

<sup>10</sup>Vgl. [SS11]: 321.

## 2. Hintergrund

Filter aufweisen, untereinander nicht ähnlich sein müssen. Man betrachte z.B. die Filter  $F_1$ ,  $F_2$  und  $F_3$  mit folgenden Werten: Die Jaccard-Distanzen zwischen  $F_1$ ,  $F_2$  und  $F_3$  betragen somit:

$$J_\delta(F_1, F_3) = 1 - J(F_1, F_3) = 1 - \frac{|F_1 \cap F_3|}{|F_1 \cup F_3|} = 1 - \frac{5}{10} = 0.5$$

$$J_\delta(F_2, F_3) = 1 - J(F_2, F_3) = 1 - \frac{|F_2 \cap F_3|}{|F_2 \cup F_3|} = 1 - \frac{5}{10} = 0.5$$

$$J_\delta(F_1, F_2) = 1 - J(F_1, F_2) = 1 - \frac{|F_1 \cap F_2|}{|F_1 \cup F_2|} = 1 - 0 = 1$$

Daran wird deutlich: Obwohl  $F_1$  und  $F_2$  jeweils die Hälfte der Elemente mit  $F_3$  gemeinsam haben, lässt sich daraus kein Wert für die Ähnlichkeit zwischen  $F_1$  und  $F_2$  ableiten. Sie sind sich im Gegenteil maximal unähnlich.

### 2.2.2. Teil- und Obermengenbeziehung

Will man Bloom-Filter z.B. nach Ähnlichkeit gruppieren, kann man stattdessen Teil- und Obermengenbeziehungen zwischen ihnen betrachten. Die Teilmengenbeziehung zwischen zwei Bloom-Filtern sei hier wie folgt definiert:

Ein Bloom-Filter  $F$  ist *Teilmenge* eines Bloom-Filters  $G$ , wenn darin mindestens die gleichen 0-Bits gesetzt sind wie in  $G$  (und möglicherweise weitere, zusätzliche 0-Bits).

Die Obermengenbeziehung zwischen zwei Bloom-Filtern sei hier wie folgt definiert:

Ein Bloom-Filter  $F$  ist *Obermenge* eines Bloom-Filters  $G$ , wenn darin mindestens die gleichen 1-Bits gesetzt sind wie in  $G$  (und möglicherweise weitere, zusätzliche 1-Bits).

Der maximal gefüllte Filter, in dem alle Bits gesetzt sind, ist damit Obermenge aller Filter derselben Länge (auch von sich selbst). Der leere Filter ist die (triviale) Teilmenge aller

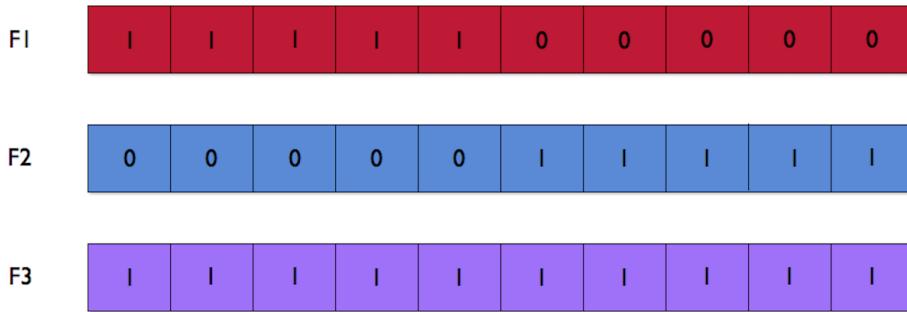


Abbildung 2.2.: Jaccard-Distanzen zwischen Bloom-Filtern.

Filter derselben Länge (auch von sich selbst). Teil- und Obermengen sind Umkehrungen voneinander, d.h. wenn  $F$  Teilmenge von  $G$  ist, folgt daraus, dass  $G$  Obermenge von  $F$  ist, und umgekehrt.

Zwischen den Bloom-Filtern in Abb. 2.2 bestehen folgende Teil- und Obermengenbeziehungen:  $F1$  und  $F2$  sind Teilmengen von  $F3$ .  $F3$  ist Obermenge von  $F1$  und  $F2$ . Zwischen den maximal unähnlichen Filtern  $F1$  und  $F2$  bestehen keine Teil- und Obermengenbeziehungen.

$F1$	0	0	0	0	0	0	1	0	0	0
$F2$	0	0	0	0	0	1	1	0	0	0
$F3$	1	1	1	1	1	1	1	0	0	0

Abbildung 2.3.: Teil- und Obermengenbeziehungen zwischen Bloom-Filtern.

Teil- und Obermengenbeziehung sind außerdem transitiv, was an Abb. 2.3 deutlich wird:  $F1$  ist Teilmenge von  $F2$ , damit auch Teilmenge von  $F3$ .  $F3$  ist Obermenge von  $F2$ , damit auch Obermenge von  $F1$ .

Teil- und Obermengenbeziehungen sind also im Gegensatz zur Jaccard-Distanz dazu geeignet, transitive Ähnlichkeitsbeziehungen zwischen Bloom-Filtern abzubilden. Diese Eigenschaft spielt eine zentrale Rolle im hier entwickelten Verfahren, das in Kap. 4 ausführlich dargestellt wird.

### 2.2.3. Hashfunktionen

Die Frage nach den idealen Hashfunktionen für einen Bloom-Filter ist nicht eindeutig zu beantworten<sup>11</sup>. Grundsätzlich muss zwischen kryptografischen Hashfunktionen wie MD5 und SHA und gewöhnlichen Hashfunktionen wie Murmur- oder Jenkins-Hashfunktionen unterschieden werden. Die Berechnung von kryptografischen Hashfunktionen dauert in der Regel länger, dafür haben sie bestimmte Eigenschaften wie eine hohe Kollisionsresistenz und Gleichverteilung der Ergebniswerte.

Werden Bloom-Filter z.B. zum schnellen Nachschlagen in großen, verteilten Datenbanken eingesetzt, wird auf die kryptografischen Eigenschaften zu Gunsten des verminderten Rechenaufwandes verzichtet. Das NoSQL-Datenbanksystem Cassandra und das Hadoop-Framework für skalierende, verteilte arbeitende Software verwenden beispielsweise Bloom-Filter in Kombination mit Murmur- und Jenkins-Hashfunktionen. Darüber hinaus ist MD5 für Bloom-Filter weit verbreitet. Murmur-Hashfunktionen haben gute Verteilungseigenschaften und lassen sich vergleichsweise schnell berechnen, weswegen sie generell

---

<sup>11</sup>Vgl. [BM04]: 487.

## 2. Hintergrund

für den Einsatz in Bloom-Filtern empfohlen werden<sup>12</sup>. AMBIENCE verwendet Murmur-Hashfunktionen zur Generierung der Bloom-Filter. Für die eigene Implementierung wurde der Murmur2-Hash verwendet<sup>13</sup>.

### 2.2.4. Bloom-Filter-Varianten und Anwendungen

Wegen ihres geringen Speicherbedarfs und einfachen Implementierung erfreuen sich Bloom-Filter in unterschiedlichsten Versionen großer Beliebtheit. Wichtige Varianten sind z.B. *Attenuated Bloom Filter*, *Counting Bloom-Filter* und *Compressed Bloom Filter*. Ein Counting Bloom-Filter benötigt mehr Speicherplatz als ein klassischer Bloom-Filter, dafür können Objekte wieder daraus entfernt werden<sup>14</sup>. Komprimierte Bloom-Filter werden eingesetzt, wenn Bloom-Filter als Nachrichten mit begrenzter Länge versendet werden oder die übertragene Datenmenge minimiert werden soll<sup>15</sup>. Attenuated Bloom-Filter<sup>16</sup> können als Array von Bloom-Filtern betrachtet werden und können z.B. in einem Netzwerk Informationen darüber enthalten, welche Dienste an einem anderen Knotenverfügbar sind. Besonders häufig kommen Bloom-Filter in verteilten Anwendungen und Netzwerkdiensten zum Einsatz<sup>17</sup>.

Neben Hadoop und Cassandra werden Bloom-Filter in unzähligen, zum Teil hoch skalierenden Anwendungen eingesetzt. Weitere Beispiele sind der quelloffene Webproxy Squid und der Chrome-Browser, wo Bloom-Filter zum schnellen Nachschlagen als bösartig eingestufter Webseiten verwendet werden. Broder und Mitzenmacher formulieren das Bloom-Filter-Prinzip wie folgt:

Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated<sup>18</sup>.

## 2.3. Indexstrukturen

Zur effizienten Bearbeitung von Anfragen und Operationen kommen in der internen Schicht von Datenbanksystemen spezielle Datenstrukturen und Speicherverfahren zum Einsatz. Sie werden *Indexstrukturen* genannt und organisieren die Daten an Hand von Indizes, um die gewünschten Operationen zu unterstützen<sup>19</sup>.

Ein *Index* oder *Verzeichnis* einer Datei enthält Informationen über ihre Struktur, wobei mit „Datei“ in diesem Zusammenhang eine komplette Datenstruktur gemeint ist, also z.B. ein Suchbaum oder ein Array. Indexstrukturen lassen sich danach klassifizieren, wie sie die Daten organisieren:

---

<sup>12</sup>Vgl. <http://spyced.blogspot.de/2009/01/all-you-ever-wanted-to-know-about.html>.

<sup>13</sup>Vgl. <https://sites.google.com/site/murmurhash/MurmurHash2.cpp> für den Quellcode.

<sup>14</sup>Vgl. [FCAB00].

<sup>15</sup>Vgl. [Mit02].

<sup>16</sup>Vgl. [SS11]: 316 und 318.

<sup>17</sup>Vgl. [BM04] für eine ausführliche Darstellung.

<sup>18</sup>[BM04]: 486.

<sup>19</sup>Trotz der großen Verbreitung von Indexstrukturen in Datenbanksystemen scheinen in der Literatur keine überblicksartigen Darstellungen zu existieren. Der aktuelle Abschnitt stützt sich daher im Wesentlichen auf das Skript zur Vorlesung *Anfragebearbeitung und Indexstrukturen in Datenbanksystemen* im Wintersemester 2013/2014 an der Ludwig-Maximilians-Universität München (vgl. [Kri14]).

1. *Daten-organisierende Indexstrukturen* werden zur Organisation der tatsächlich anfallenden Daten eingesetzt – meist in Form von *Suchbäumen*.
2. *Raum-organisierende Indexstrukturen* werden zur Organisation des Speichers eingesetzt, in dem die Daten gehalten werden. Sie verwenden vor allem *dynamische Hashverfahren*.
3. *Hybride Indexstrukturen* sind eine Kombination der vorgenannten Klassen und basieren auf *Hashbäumen*.

Eine gute bzw. effektive Indexstruktur sollte folgenden Anforderungen genügen:

1. *Effiziente Suche*: Eine Suchanfrage auf der Indexstruktur soll in optimaler Zeit ein Ergebnis liefern. D.h. die Anfrage soll in möglichst wenig Schritten an die Seite oder Seiten weiter geleitet werden, die die angefragten Daten enthalten.
2. *Dynamisches Einfügen, Modifizieren und Löschen von Datensätzen*: Die zu organisierende Datenmenge verändert sich möglicherweise über die Zeit, was durch die Indexstruktur widergespiegelt und unterstützt werden muss.
3. *Erhalt der lokalen Ordnung*: Falls es Datensätze gibt, deren Schlüssel in der angewandten Ordnungsrelation (z.B. die Kleiner-Gleich-Ordnung) aufeinander folgen, sollte die Indexstruktur diese Ordnung übernehmen. Suchbäumen erfüllen diese Eigenschaft, nicht aber lineare Hashverfahren. Die Wahl bzw. Implementierung der Indexstruktur muss also zum Anwendungsfall passen.
4. *Spechereffizienz*: Effiziente Speichernutzung ist für real existierende und hoch skalierende Anwendungen von zentraler Bedeutung.

Weitere mögliche Anforderungen sind *Machbarkeit* und *Implementierungskosten*. Sie sind für die vorgestellte Implementierung nachrangig in dem Sinne, dass der Nachweis der Machbarkeit durch die Implementierung selbst erfolgt. Da AMBIENCE ein Prototyp ist und im akademischen Umfeld entwickelt wurde, kann die wirtschaftliche Kalkulation der Kosten, wie sie ein Unternehmen vornehmen würde, außer Acht gelassen werden.

Für Implementierung (vgl. Kap. 4) und Evaluation (vgl. 5) stehen daher die Anforderungen 1–4 im Mittelpunkt. Als weiteres Kriterium, das nicht zu den allgemeinen Anforderungen an Indexstrukturen zählt, wurden die Aufbaukosten der Indexstruktur betrachtet.

### 2.3.1. B-Bäume

B-Bäume sind eine weit verbreitete Form der Suchbäume und wurden zuerst 1972 von Rudolf Bayer und Edward M. McCreight vorgestellt. Sie erfüllen unter anderem folgende Eigenschaften:

1. *Aufbau*: B- und B<sup>+</sup>-Bäume wachsen und schrumpfen von der Wurzel ausgehend.
2. *Balanciertheit*: Alle Blätter sind auf demselben Level.
3. *Minimaler Grad/Ordnung*: B- und B<sup>+</sup>-Bäume sind definiert durch die Ordnung oder den minimalen Grad  $t$ , d.h. jeder Knoten außer der Wurzel enthält mindestens  $t$  Schlüssel.
4. *Suchbaumeigenschaft*: Schlüssel sind aufsteigend sortiert.
5. *Verzweigungsgrad*: Ein innerer Knoten mit  $k$  Schlüsseln hat genau  $k+1$  Kinder<sup>20</sup>.

---

<sup>20</sup>Vgl. [OW12]: 339–348. Den Grad bzw. die minimale Ordnung betreffend ist die Nomenklatur in der Literatur uneinheitlich. [OW12] sprechen in Anlehnung an [Knu99] von B-Bäumen der Ordnung  $m$  und fordern, dass jeder Knoten mit Ausnahme der Wurzel und der Blätter mindestens  $\lceil \frac{m}{2} \rceil$  enthalte (vgl. ebd.: 342f.). Stein et al. definieren den minimalen Grad wie oben beschrieben (vgl. [SCLR09]: 489). Kriegel verwendet dasselbe Kriterium, bezeichnet es jedoch als Grad  $m$  (vgl. [Kri14]: 9).

## 2. Hintergrund

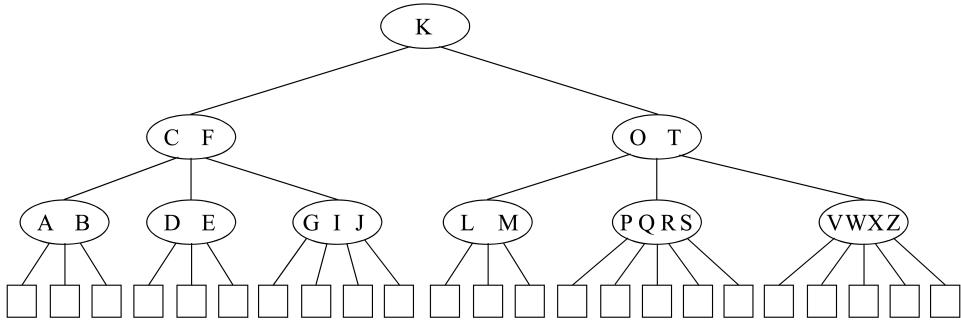


Abbildung 2.4.: Ein B-Baum der Ordnung 2.

Unterstützte Operationen sind in der Regel Einfügen, Suchen und Löschen von Datensätzen. Dabei ist die Löschoperation am aufwendigsten zu implementieren, da auch das Löschen eines Schlüssels aus einem inneren Knoten betrachtet werden muss. Ein Schlüssel aus einem inneren Knoten kann nicht direkt gelöscht werden, da er zusätzlich als Separator für seine Kindknoten dient. Es muss daher ein neuer Separator aus dem rechten oder linken Kindknoten entnommen und in den Knoten eingefügt werden. Falls damit die B-Baum-Eigenschaften verletzt werden, d.h. der Kindknoten anschließend zu wenig Schlüssel enthält, muss eine Underflow-Behandlung eingeleitet werden, bei dem der Kindknoten mit einem Geschwisterknoten verschmolzen wird. Der Underflow kann sich rekursiv bis zur Wurzel fortsetzen und dafür sorgen, dass der Baum eine Ebene verliert und eine neue Wurzel erhält.

### 2.3.2. $B^+$ -Bäume

Das entwickelte Verfahren stützt sich stark auf  $B^+$ -Bäume, eine Erweiterung der B-Bäume. Sie unterscheiden sich von ihnen in folgenden Eigenschaften:

1. *Minimale Schlüsselanzahl*: Jeder innere Knoten enthält mindestens einen Schlüssel.
2. *Ordnungsrelation*: Der Kindknoten zwischen den Schlüsseln  $k1$  und  $k2$  enthält alle Schlüssel  $\geq k1$  und  $< k2$ .
3. *Speicherung der Datensätze*: Alle Schlüssel werden auch in den Blättern gespeichert. Die tatsächlichen Datensätze sind mit dem entsprechenden Schlüssel im Blatt verknüpft.
4. *Sequentielle Verkettung*: Alle Blätter sind gemäß der Ordnung auf den Primärschlüsseln verkettet.

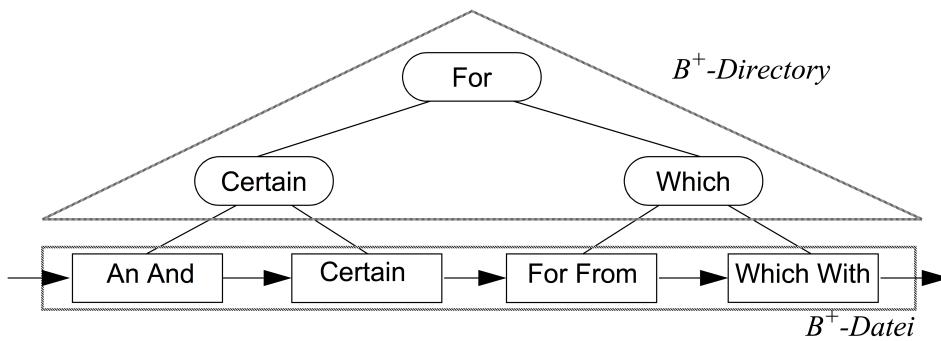


Abbildung 2.5.: Ein B<sup>+</sup>-Baum der Ordnung 2.

Die wesentlichen Vorteile gegenüber dem B-Baum sind der höhere Verzweigungsgrad und die Bereichssuche. Tab. 2.1 vergleicht die durchschnittlichen Laufzeiten für Einfügen, Suchen und Löschen in B-Bäumen und  $B^+$ -Bäumen:

	Einfügen	Suchen	Löschen	Bereichssuche
<b>B-Baum</b>	$O(\log(2t - 1))$	$O(\log(2t - 1))$	$O(\log(2t - 1))$	$\times$
<b>B<sup>+</sup>-Baum</b>	$O(\log(2t - 1))$	$O(\log(2t - 1))$	$O(\log(2t - 1))$	$O(\log 2t - 1)$

Tabelle 2.1.: Laufzeiten von Operationen auf Suchbäumen.

Daran wird deutlich, dass die Kosten für Einfügen, Suchen und Löschen in beiden Varianten vom Parameter  $t$  abhängen. Da die inneren Knoten im B<sup>+</sup>-Baum mehr Schlüssel enthalten als im B-Baum, wird der Baum breiter und flacher. Geht man davon aus, dass ein Zugriff auf einen inneren Knoten einem Plattenzugriff entspricht, wird deutlich, dass sich damit die Kosten für Einfügen, Suchen und Löschen gegenüber dem B-Baum reduzieren lassen.

Die sequentielle Verkettung der Blätter im B<sup>+</sup>-Baum ermöglicht zudem eine Bereichssuche. Das ist ein immenser Vorteil gegenüber dem B-Baum, bei dem jeder für eine Bereichssuche jeder einzelne Datensatz mit Kosten von  $O(\log(2t - 1))$  gesucht werden muss. Diese Kosten fallen beim B<sup>+</sup>-Baum nur für den ersten Datensatz mit dem kleinsten Primärschlüssel an. Anschließend kann eine doppelt verkettete Liste traversiert werden.



### 3. Verwandte Themen

Bloom-Filter spielen eine wichtige Rolle in AMBIENCE. Somit stellte sich weder die Frage, ob überhaupt Bloom-Filter eingesetzt werden sollen, noch nach ihrer optimalen Implementierung. AMBIENCE diente vielmehr als Schnittstelle für den eigenen Entwurf.

Dennoch wurden die Bloom-Filter auch selbst implementiert, um die eigenen Berechnungen zu überprüfen und mit realistischen Werten arbeiten zu können. Das folgende Kapitel stellt in Abschnitt 3.1 zunächst dar, auf welche Arbeiten und bewährten Techniken dabei Bezug genommen wurde.

Ziel dieser Arbeit war eine optimale Lösung für das Anwendungsszenario von AMBIENCE. Eine solche wird vom aktuellen Stand der Forschung nicht abgedeckt. Die Darstellung des aktuellen Forschungsstandes konzentriert sich wegen der Fülle der Einsatzmöglichkeiten auf Bloom-Filter in Netzwerkanwendungen und Indexstrukturen sowie die  $k$ -nächste-Nachbarn-Suche. Überlegungen und Arbeiten, die Eingang in die eigene Implementierung gefunden haben, werden in den Abschnitten 3.2, 3.3 und 3.4 vorgestellt.

#### 3.1. Implementierung von Bloom-Filtern

Gemäß dem Bloom-Filter-Prinzip bietet sich die Verwendung von Bloom-Filtern an, wenn Speicherplatz effektiv genutzt werden soll und gleichzeitig die Auswirkungen von falsch Positiven abgemildert werden können (vgl. Abschnitt 2.2.4). Die mathematischen Grundlagen wie Minimierung der Falsch-Positiv-Rate, Abschätzung der Anzahl eingefügter Elemente, Abschätzung der Jaccard-Distanz etc. werden von Bloom<sup>1</sup>, Broder, Mitzenmacher<sup>2</sup> sowie Werner et al.<sup>3</sup> ausführlich dargestellt, um nur einige zu nennen.

Wie in den Abschnitten 2.2.3 und 2.2.4 erwähnt, werden Bloom-Filter im Apache-Projekt Cassandra eingesetzt. Sie dienen dort zum schnellen Nachschlagen in Tabellen, den so genannten *SSTables*. Die Cassandra-Entwickler, namentlich Jonathan Ellis, haben sich eingehend mit der Implementierung von Bloom-Filtern und optimalen Hashfunktionen beschäftigt. Diese Überlegungen haben keinen Eingang in wissenschaftliche Veröffentlichungen gefunden, sind jedoch sehr praxisrelevant<sup>4</sup>. Zur Wahl der Hashfunktionen schreibt Ellis:

---

<sup>1</sup>Vgl. [Blo70].

<sup>2</sup>Vgl. [BM04] und [Mit02].

<sup>3</sup>Vgl. [WDS15].

<sup>4</sup><http://cassandra.apache.org/> ist die Hauptseite des Cassandra-Projekts. Der Cassandra-Quellcode ist frei verfügbar unter <https://github.com/apache/cassandra>. Datenmodell und Architektur werden z.B. unter <http://wiki.apache.org/cassandra/DataModel>, <http://wiki.apache.org/cassandra/ArchitectureOverview> und <http://prettyprint.me/prettyprint.me/2010/05/02/understanding-cassandra-code-base/index.html> beschrieben. Für diese Arbeit wurden auch eine programmatische Rede (vgl. <https://youtu.be/WD1v6jr5fKY>) und ein Blog (vgl. <http://spyced.blogspot.de/>) von Ellis berücksichtigt.

### 3. Verwandte Themen

[I]t turns out that it's surprisingly hard to find good information on one part of the implementation: how do you generate an indefinite number of hashes? Even small filters will use three or four; a dozen or more is not unheard of<sup>5</sup>.

Die mathematischen Grundlagen der Generierung von  $i$  Hashfunktionen mit möglichst gleich verteilten Ergebnissen sind bei Kirsch und Mitzenmacher<sup>6</sup> zu finden. Viele Bloom-Filter-Implementierungen wie *PyBloom* verwenden jedoch Hashfunktionen, deren Ergebnisse nicht gleich verteilt sind. Das Problem ist, dass damit häufig eine deutlich höhere Falsch-Positiv-Rate im Bloom-Filter erzielt wird als rein rechnerisch zu erwarten wäre.

Der Grund für die Verwendung minderwertiger Hashfunktionen ist Ellis zu Folge, dass die meisten Implementierungen auf schnelle Berechnung der Hashwerte abzielten statt auf Gleichverteilung der Ergebnisse. Das ist aber für einen Bloom-Filter essentiell, wenn z.B. wie in Cassandra teure Eingabe/Ausgabe-Operationen durch den Einsatz von Bloom-Filters reduziert werden sollen. Weist der Bloom-Filter eine erhöhte Falsch-Positiv-Rate auf, beispielsweise von 140% gegenüber dem erwarteten Wert, reduziert das die positiven Effekte des Bloom-Filters drastisch.

Ellis präsentiert zwei Lösungsansätze: Entweder kryptografische Hashfunktionen oder Murmur- und Jenkins-Hashfunktionen mit guter Gleichverteilung der Ergebnisse. Kryptografische Hashfunktionen wurden bereits in Abschnitt 2.2.3 dargestellt. Der Nachteil daran ist, dass sie in der Regel aufwändiger zu berechnen sind als gewöhnliche Hashfunktionen. Das spielt z.B. bei der einmaligen Berechnung eines Fingerabdrucks keine große Rolle. Die Performanz von Bloom-Filters wird dadurch aber beeinträchtigt.

Mit Murmur- oder Jenkins-Hashfunktionen gibt es zwei Möglichkeiten, eine beliebige Anzahl von Hashfunktionen mit guter Gleichverteilung der Ergebnisse zu erzeugen. Entweder berechnet man den  $i$ -ten Hashwert als  $\text{hash}_0 + i * \text{hash}_1$  wie von Kirsch und Mitzenmacher beschrieben und in Cassandra angewendet. Alternativ nimmt man den  $i$ -ten Hashwert als Startwert für die Berechnung des  $i+1$ -ten Hashwerts. Dieser Ansatz wurde in Hadoop gewählt und wird auch hier verwendet.

Zur Organisation der Bloom-Filter äußert sich Ellis ebenfalls, jedoch nur auf seinem Twitter-Account und ohne auf Details einzugehen:

Rather than naively checking every Bloom Filter for the element, organize the BF in a hierarchy akin to B+ tree<sup>7</sup>.

Zwar ist hier offensichtlich von der Suche nach einem Element in einem Bloom-Filter die Rede, nicht nach einem Bloom-Filter selbst wie in AMBIENCE. Dennoch zeichnet sich ab, dass ein B<sup>+</sup>-Baum als Indexstruktur für Bloom-Filter in Erwägung gezogen werden sollte.

Damit sind die Gemeinsamkeiten zwischen Cassandra und AMBIENCE erschöpft. Es war zunächst überlegt worden, Cassandra zur Evaluation der eigenen Implementierung zu nutzen, d.h. die eigenen Datensätze in eine Cassandra-Installation zu überführen und z.B. die Laufzeiten der  $k$ -nächste-Nachbarn-Suche zu vergleichen. Der entscheidende Unterschied zu Cassandra ist jedoch, dass in AMBIENCE die Bloom-Filter selbst die Datensätze sind, während sie in Cassandra zum Nachschlagen der Datensätze verwendet werden. Damit erwies sich die Evaluation mit Cassandra als nicht praktikabel. Die Publikationen

---

<sup>5</sup>Vgl. <http://spyced.blogspot.de/2009/01/all-you-ever-wanted-to-know-about.html>.

<sup>6</sup>Vgl. [KM06].

<sup>7</sup>Vgl. <https://twitter.com/spyced/status/707266703751651328>.

aus dem Cassandra-Umfeld wurden daher ausschließlich bezüglich Implementierung der Bloom-Filter, Wahl und Berechnung der Hashfunktionen berücksichtigt. Der Hinweis auf  $B^+$ -Bäume in Zusammenhang mit Bloom-Filtern unterstützte zudem die Entscheidung für diese Indexstruktur.

### 3.2. Netzwerkanwendungen mit Bloom-Filtern

Bloom-Filter werden in zahlreichen Netzwerk-Anwendungen eingesetzt. Das erscheint auf den ersten Blick interessant für AMBIENCE als soziales Online-Netz. Ein guter Überblick über Netzwerk-Anwendungen mit Bloom-Filtern findet sich bei Broder und Mitzenmacher:

The aim of this paper is to survey the ways in which Bloom filters have been used and modified in a variety of network problems, with the aim of providing a unified mathematical and practical framework for understanding them and stimulating their use in future applications<sup>8</sup>.

Die Autoren beschreiben unter anderem die Verwendung von Bloom-Filtern zur Kollaboration in Overlay- und Peer-to-Peer-Netzwerken<sup>9</sup>, jedoch mit einem entscheidenden Unterschied zur Fragestellung: Zwar werden die Anfragen in AMBIENCE von einem mobilen Endgerät an einen Host gesendet, also über das Netzwerk übertragen. Die  $k$ -nächste-Nachbarn-Suche bezieht sich jedoch auf die Menge der an einem Host gespeicherten Bloom-Filter. Unter diesen sollen die ähnlichsten möglichst schnell und effizient gefunden werden. Ab dem Zeitpunkt, zu dem die Anfrage am Host eingegangen ist, werden aber keine Nachrichten mehr im Netzwerk übertragen. Die  $k$ -nächste-Nachbarn-Suche findet nur auf dem Host selbst statt und erst das Ergebnis wird an das anfragende Gerät übermittelt.

Ähnlich verhält es sich mit der Arbeit von Agarwal und Trachtenberg. Dort werden Counting Bloom-Filter verwendet, um die Abschätzung von Mengendifferenzen auf unterschiedlichen Hosts zu optimieren<sup>10</sup>. Daran werden zwar die Vorteile von Bloom-Filtern bei der blinden Kalkulation von Mengenunterschieden deutlich, ohne dass ihre Elemente bzw. die Objekte in den Bloom-Filtern bekannt sein müssen. Wie bei Werner et al. beschrieben, kann diese Eigenschaft den Schutz der Privatsphäre in einem sozialen Online-Netz verbessern<sup>11</sup>. Auch hier findet die Abschätzung der Mengendifferenzen jedoch zwischen zwei unterschiedlichen Hosts statt, weswegen dieser Ansatz für AMBIENCE nicht genutzt werden kann. Gleches gilt für die Arbeiten von Byers et al.<sup>12</sup>, Shiraki et al.<sup>13</sup>, Zhang<sup>14</sup> und Zhu et al.<sup>15</sup>. Überall dort werden Bloom-Filter im Zusammenhang mit Netzwerken eingesetzt, doch keiner der Ansätze lässt sich auf AMBIENCE anwenden.

---

<sup>8</sup>[BM04]: 485.

<sup>9</sup>Vgl. ebd.: 486.

<sup>10</sup>Vgl. [AT06].

<sup>11</sup>Vgl. [WDS15]: 2 und 5.

<sup>12</sup>Vgl. [BCM02].

<sup>13</sup>Vgl. [STT<sup>+</sup>09].

<sup>14</sup>Vgl. [Zha12].

<sup>15</sup>Vgl. [ZJW04].

### 3.3. Indexstrukturen für Bloom-Filter

Ein Überblick über Indexstrukturen für Datenbank-Managementsysteme auf Hauptspeicherbasis findet sich bei Lehman und Carey<sup>16</sup>. Die Publikation ist nicht ganz aktuell, kann aber zu einem besseren Verständnis der Indexstrukturen aus Abschnitt 2.3 beitragen. Wichtig ist vor allem die Erkenntnis, dass der Einsatz einer Hauptspeicher-Datenstruktur sinnvoll sein kann, um die Performanz von Anfragen auf größeren Datenmengen zu verbessern. B- und B<sup>+</sup>-Bäume werden allgemein als Hauptspeicher-Datenstrukturen verwendet. Ein Vorteil davon ist die Reduzierung der Plattenzugriffe bei der Suche nach Datensätzen. In der Regel benötigen sie viel weniger Speicherplatz als das tatsächliche Datenaufkommen. Außerdem kann in der Indexstruktur häufig mit Zeigern gearbeitet werden, ohne alle vorhandenen Daten tatsächlich in der Datenstruktur halten zu müssen.

Nachdem sich abzweichnete, dass die Organisation der Bloom-Filter durch eine geeignete Indexstruktur erfolgen könnte, wurde mit der Suche begonnen. Adaptierungen der kanonischen Formen für spezifische Anwendungen finden sich in der Literatur zu Hauf. Wenig überraschend ist darunter keine, die das spezifische Szenario von AMBIENCE abbildet.

Hellerstein und Pfeffers<sup>17</sup> Publikation von 1994 stellt den *RD-Tree* als Indexstruktur für objektrelationale Geo-Datenbanken vor. Abgesehen davon, dass der objektrelationale Ansatz mittlerweile kaum noch verfolgt wird, ist die Indexstruktur optimiert für Geodaten. Diese haben spezielle Eigenschaften wie z.B. Inklusionsbeziehungen zwischen Punkten und Flächen oder Flächen untereinander, die in der Indexstruktur abgebildet und von Suchanfragen unterstützt werden müssen. Spezielle Indexstrukturen für Geodaten erschienen daher ungeeignet, da sie einerseits einen unnötigen Implementierungsaufwand erfordern, der für die vorhandenen Daten nicht erforderlich wäre, andererseits die *k*-nächste-Nachbarn-Suche unnötig erschweren. Zu diesen speziellen Indexstrukturen zählt auch der *R-Baum* zur Organisation von minimalen umgebenden Rechtecken. Eine Variante davon ist der *bichromatic Rdnn-Tree* von der Yang und Lin<sup>18</sup>. Diese Arbeit erschien insofern interessant, als die Autoren nicht nur Punkt-Anfragen, sondern *k*-nächste-Paare und *k*-nächste-Nachbarn-Paare betrachten und ihre Indexstruktur dafür optimiert haben. Mangels Ähnlichkeit von Bloom-Filtern und spatialen Datensätzen wurde jedoch auch dieser Ansatz verworfen.

Am stärksten wurde schließlich die Arbeit „Evaluation of the Structured Bloom Filters Based on Similarity“ von Hiroshi Sakuma und Fumiaki Sato berücksichtigt<sup>19</sup>. Sie behandelt zwar ein Netzwerk-Problem, doch ließen sich Teile der Indexstruktur für AMBIENCE adaptieren. Kern der Arbeit ist ein Baum aus Bloom-Filtern zur Minimierung der Weiterleitungen von Anfragen in einem Verteilten System. Die Indexstruktur ist ein Baum aus Bloom-Filtern, an dem jeder im Netzwerk beteiligte Knoten einen Teil hält (vgl. Abb. 3.1). Er ist nach Ähnlichkeit zwischen Bloom-Filtern aufgebaut. Wie in Abschnitt 2.2.1 beschrieben, dient als Ähnlichkeitsmaß jedoch nicht die Jaccard-Distanz, sondern die Anzahl gleicher 1-Bits. Die Datensätze werden nicht im Baum selbst gehalten, sondern die Knoten des Baums dienen dem Management bzw. der Organisation. Da es sich um ein Verteiltes System handelt, aus dem Knoten ausscheiden oder neu hinzukommen können,

---

<sup>16</sup>Vgl. [LC86].

<sup>17</sup>Vgl. [HP94].

<sup>18</sup>Vgl. [YL02]

<sup>19</sup>Vgl. [SS11].

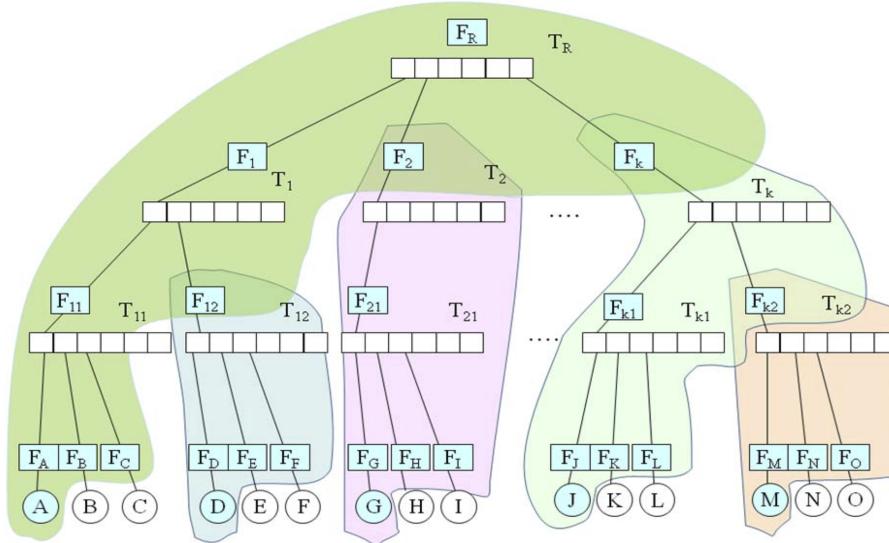


Abbildung 3.1.: Bloom-Filter-Baum bei Sakuma und Sato ([SS11]: 320).

müssen regelmäßige Updates per Broadcast-Nachrichten an alle Knoten im Baum propa-  
giert werden. Ein Knoten ist darin für den unter ihm liegenden Teilbaum verantwortlich.  
D.h. in Abb. 3.1 hält der Wurzelknoten  $F_R$  Informationen über das gesamte Netzwerk,  
Knoten  $T_{k2}$  hingegen hält nur Informationen über den orangefarbenen Teilbaum.

Im Zentrum stehen zwei Aspekte, die von den Autoren evaluiert und mit bestehenden Methoden verglichen werden: Anzahl der Schritte bei der Anfrage-Weiterleitung und Restrukturierung des Baumes beim Ausscheiden und Beitreten von Knoten<sup>20</sup>. Die Anfrage-Weiterleitung basiert dabei auf der Baumstruktur, die Restrukturierung auf der Anordnung der Bloom-Filter nach Ähnlichkeit. Beide Aspekte sind für AMBIENCE weniger relevant: Erstens findet die  $k$ -nächste-Nachbarn-Suche auf ein und demselben Host statt. Anfragen müssen also nicht an andere Hosts weiter geleitet werden. Daher wurde darauf verzichtet, in inneren Knoten Informationen über die Geschwisterknoten vorzuhalten. Zweitens ist davon auszugehen, dass die Indexstruktur an einem Host nach dem Aufbau relativ stabil bleibt. Insbesondere können Ausscheiden, Beitreten und Ausfall von Knoten in einem Verteilten System vernachlässigt werden. Die Restrukturierungskosten für die Indexstruktur wurden daher nicht näher betrachtet.

Bezüglich der Management-Methoden treffen Sakuma und Sato in naheliegender Weise eine Unterscheidung zwischen physischen und logischen Knoten. Physische Knoten sind die tatsächlich im Netzwerk vorhandenen Hosts. Logische Knoten sind die Baumkno-  
ten. Sie sind für die Weiterleitung von Anfragen und die Aktualisierung des Baumes bei Restrukturierung des Netzwerkes zuständig. Diese Aufteilung wurde nicht in die Imple-  
mentierung übernommen, sondern es gibt nur eine Art von Knoten. Grund hierfür ist wiederum, dass die tatsächlichen Datensätze bei Sakuma und Sato keine Bloom-Filter sind. Die Bloom-Filter dienen lediglich der Strukturierung des Netzwerks und der effi-

<sup>20</sup>Vgl. [SS11]: 316.

### 3. Verwandte Themen

zienten Weiterleitung von Anfragen. Da die Bloom-Filter in AMBIENCE selbst die Datensätze sind, musste diese Unterscheidung nicht getroffen werden. Vielmehr wurden die Bloom-Filter selbst bzw. ihre IDs als Primärschlüssel verwendet (vgl. Kap. 4 für Details der Implementierung).

Zur Verwaltung der Bloom-Filter im Baum wurde ein wichtiges Element übernommen: Jeder Knoten besitzt einen konsolidierten Bloom-Filter zur Verwaltung der Index-Informationen<sup>21</sup>. Konkret bedeutet das, dass beim Einfügen eines Objekts in den Baum das bitweise binäre Oder des eingefügten Filters und des konsolidierten Filters jedes Teilbaums gebildet wird. Der konsolidierte Filter jedes Knotens besteht damit aus dem bitweisen binären Oder aller Filter, die in diesen Teilbaum eingefügt wurden<sup>22</sup>. Dieser Faktor ist entscheidend für die Beschleunigung der  $k$ -nächste-Nachbarn-Suche, wie in den Kapiteln 4 und 5 dargestellt wird.

In Abschnitt 2.2.1 wurde erläutert, dass die Jaccard-Distanz im Unterschied zum Ähnlichkeitsmaß bei Sakuma und Sato nicht transitiv ist. Die Organisation der Bloom-Filter im Baum nach Ähnlichkeit konnte daher für AMBIENCE nicht angewendet werden. Des Weiteren stellte sich heraus, dass die Autoren zwar beständig von einem B-Baum sprechen, in Wirklichkeit aber einen  $B^+$ -Baum implementiert haben. Auch ohne Implementierungsdetails lässt sich das z.B. daran erkennen, dass alle Primärschlüssel auch in den Blättern zu finden sind<sup>23</sup>. Die eigene Implementierung verwendet einen  $B^+$ -Baum.

Im Unterschied zum B-Baum existieren für den  $B^+$ -Baum wenig kanonische Implementierungen. An dieser Stelle sei die Arbeit von Jan Jannink genannt, der insbesondere die Löschoperation im  $B^+$ -Baum sowohl theoretisch als auch mit Codebeispielen erläutert<sup>24</sup>. Die eigene Implementierung folgt nicht der von Jannink, doch wird die Löschoperation von ihm sehr gut veranschaulicht<sup>25</sup>.

## 3.4. $k$ -nächste-Nachbarn-Suche

Einen anderen Ansatz verfolgen Bayardo et al., die das mit der  $k$ -nächsten-Nachbarn-Suche verwandte Problem der  $k$ -nächsten-Paare untersuchen<sup>26</sup>. Die Autoren haben einen viel zitierten und frei verfügbaren Algorithmus<sup>27</sup> entwickelt, um die  $k$ -nächste-Paare-Suche auf dünn besetzten Vektoren zu optimieren:

Given a large collection of sparse vector data in a high dimensional space, we investigate the problem of finding all pairs of vectors whose similarity score (as determined by a function such as cosine distance) is above a given threshold. We propose a simple algorithm based on novel indexing and optimization strategies that solves this problem without relying on approximation methods or extensive parameter tuning<sup>28</sup>.

---

<sup>21</sup>Vgl. [SS11]: 319

<sup>22</sup>Vgl. ebd.: 320.

<sup>23</sup>Vgl. ebd.: 319/Abb. 4.

<sup>24</sup>Vgl. [Jan95].

<sup>25</sup>Vgl. z.B. ebd.: 36/Abb. 1.

<sup>26</sup>Vgl. [BMS07].

<sup>27</sup>Vgl. <https://code.google.com/archive/p/google-all-pairs-similarity-search/> für den Quellcode.

<sup>28</sup>[BMS07]: 131.

Diese Verallgemeinerung der *k*-nächsten-Nachbarn-Suche wird auch als „similarity join problem“<sup>29</sup> bezeichnet, der entwickelte Algorithmus ist unter dem Namen „all pairs“ oder „all pairs similarity search“<sup>30</sup> bekannt.

Für diese Arbeit wurde der All-Pairs-Algorithmus nur theoretisch betrachtet. Grund hierfür ist neben dem abweichenden Distanzmaß vor allem der abweichende Ansatz, die Suchfunktion selbst und nicht die Indexstruktur zu optimieren. Zwar werden auch für All-Pairs Indexstrukturen auf Kandidatenmengen an Hand von Schranken aufgebaut<sup>31</sup>. Dies geschieht jedoch dynamisch und generiert keine persistente Datenstruktur, was das Ziel dieser Arbeit war. Zukünftige Arbeiten könnten jedoch prüfen, ob All-Pairs für AM-BIENCE eingesetzt werden kann (vgl. hierzu Kap. 6).

---

<sup>29</sup>[BMS07]: 132.

<sup>30</sup>Vgl. z.B. <http://mlwave.com/tutorial-google-all-pairs-similarity-search/>.

<sup>31</sup>Vgl. [BMS07]: 132.



# 4. Implementierung

Das folgende Kapitel beschreibt die Entwicklung einer Indexstruktur zur Optimierung der  $k$ -nächste-Nachbarn-Suche in AMBIENCE. Sie heißt *BloomFilterTree*. Abschnitt 4.1.1 beschreibt den konzeptionellen Aufbau. Die praktische Umsetzung wird in Abschnitt 4.1.2 dargestellt. Die zentralen Operationen Einfügen von Bloom-Filtern und  $k$ -nächste-Nachbarn-Suche werden in den Abschnitten 4.1.3 und 4.1.4 erläutert. Abschließend wird dargestellt, welche alternativen Ansätze während der Implementierung verfolgt und weshalb sie letztlich verworfen wurden (Abschnitt 4.2). Evaluation des Verfahrens und Gegenüberstellung mit der naiven Implementierung finden sich im folgenden Kapitel 5.

## 4.1. BloomFilterTree

Um möglichst plattformunabhängig zu bleiben, wurde die Implementierung in C++ realisiert. Ausgewählte Codebeispiele finden sich im Anhang (vgl. Kapitel A). Zunächst war überlegt worden, teilweise auf bestehende Bibliotheken für Bloom-Filter und Baumstrukturen zurückzugreifen wie die bekannte *Open Bloom Filter*-Bibliothek von Arash Partow<sup>1</sup>. Darauf wurde schließlich aus zwei Gründen verzichtet: Einerseits liegt der Fokus der Arbeit nicht auf der Implementierung von Bloom-Filtern. Andererseits stünde durch die Verwendung bestehender Bibliotheken zu befürchten, dass Messergebnisse durch den Rechnenaufwand für nicht benötigte Operationen oder durch in AMBIENCE nicht vorhandene Optimierungen verfälscht würden.

Eine Bibliothek für B<sup>+</sup>-Bäume ist z.B. das *STX B+ Tree package* von Timo Bingmann<sup>2</sup>. Wie in Abschnitt 3.3 dargestellt, sind keine Varianten von Indexstrukturen bekannt, die sich unverändert für AMBIENCE übernehmen ließen. Auch eine bestehende Bibliothek müsste also stark abgewandelt werden. Gleichzeitig könnten dieselben Verfälschungen auftreten wie bei Verwendung einer Bloom-Filter-Bibliothek. So wurde auch davon abgesehen.

### 4.1.1. Aufbau

Am vielversprechendsten erschien es, wie Sakuma und Sato in ihrer Arbeit über „Structured Bloom Filters Based on Similarity“<sup>3</sup> zunächst von einem B<sup>+</sup>-Baum auszugehen (vgl. Abschnitt 3.3). Dieser wurde zuerst implementiert mit allen Eigenschaften wie in Abschnitt 2.3 beschrieben. Anschließend wurde er schrittweise zur Organisation der Bloom-Filter erweitert:

1. Die Datensätze sind Bloom-Filter, d.h. jedes Blatt hält  $n$  Zeiger auf die  $n$  Bloom-Filter-Objekte, die darin eingefügt wurden. Bloom-Filter-Objekte werden über ihre ID als Primärschlüssel identifiziert.

<sup>1</sup>Vgl. <https://github.com/ArashPartow/bloom> für den Quellcode.

<sup>2</sup>Vgl. <https://github.com/bingmann/stx-btree> für den Quellcode.

<sup>3</sup>Vgl. [SS11].

#### 4. Implementierung

2. Jeder Baumknoten hat einen Vereinigungs-Bloom-Filter. Er wird aus dem bitweisen logischen Oder aller Filter gebildet, die in den darunter liegenden Teilbaum eingefügt wurden (vgl. Abschnitt 3.3).

Abb. 4.1 veranschaulicht den Aufbau eines BloomFilterTree. Die Baumknoten sind blau markiert. Sie enthalten die Primärschlüssel der Bloom-Filter sowie Zeiger auf die Kind- bzw. Nachbarknoten. Jeder Knoten hat einen weiß markierten Vereinigungsfilter. Die violetten markierten Filter repräsentieren die tatsächlichen Datensätze. Die Blätter verweisen jeweils auf die darin eingefügten Filter.

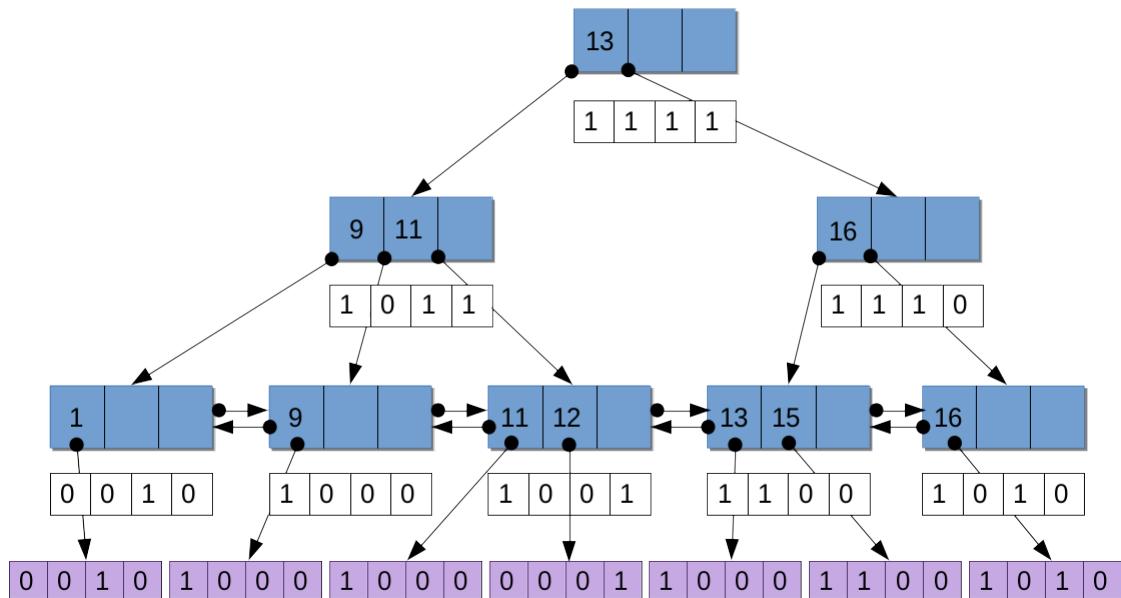


Abbildung 4.1.: Aufbau eines BloomFilterTree mit Bloom-Filtern als Datensätzen und Vereinigungsfiltern an allen Knoten.

#### 4.1.2. Umsetzung

Die Klasse `BloomFilterTree` enthält schließlich alle notwendigen Parameter und Operationen zur Organisation der Bloom-Filter. Dazu gehören die  $B^+$ -Baum-Operationen wie Einfügen und Suchen von Schlüsseln, Traversieren der Blätter, boolesche Abfrage nach Enthaltensein eines Schlüssel im Baum, Zählen der Blätter etc.. Darüber hinaus gibt es viele weitere Operationen:

1. *Management-Operationen* zum Berechnen der Jaccard-Distanzen zu allen Filtern im Baum, naive Version der  $k$ -nächste-Nachbarn-Suche, Traversieren der Datensätze etc..

2. Die *zentralen Operationen* des Verfahrens: Einfügen der Bloom-Filter nach Ähnlichkeit und  $k$ -nächste-Nachbarn-Suche.
3. *Mess- und Vergleichsoperationen*, um z.B. Varianten der  $k$ -nächste-Nachbarn-Suche, Aufbaukosten und Speicherbedarf der Datenstrukturen zu vergleichen.

Die Header-Datei der Klasse `BloomFilterTree` ist im Anhang abgedruckt (vgl. Kapitel A). In der Klasse `BloomFilter` wurden alle Parameter und Operationen auf Bloom-Filttern realisiert, die sich im Laufe der Arbeit als wichtig erwiesen. Dazu gehören:

1. *Typische Bloom-Filter-Parameter und -Operationen*: Anzahl der Hashfunktionen, Daten-Array, Setzen von Bitpositionen etc..
2. *Mathematische und Vergleichsoperationen* wie Berechnung von Teil- und Obermengen, bitweises logisches Und und Oder, Berechnung und Abschätzung von Jaccard-Distanzen.
3. *Operationen zum Bloom-Filter-Management* wie Einfügen von zufälligen Elementen aus einem Wörterbuch oder zufällige Initialisierung mit Werten aus  $\{0, 1\}$ .

Die Header-Datei der Klasse `BloomFilter` findet sich ebenfalls im Anhang.

#### 4.1.3. Einfügen

Die Einfüge-Operation zählt zu den wichtigsten Funktionen der Indexstruktur. Sie ist entscheidend für die Optimierung von Laufzeit und CPU-Zeit der  $k$ -nächste-Nachbarn-Suche. Der Algorithmus basiert auf den Teil- und Obermengenbeziehungen zwischen Bloom-Filttern. Er verwendet die Vereinigungsfilter der bereits existierenden Knoten, um die optimale Position für den neu einzufügenden Filter zu finden. Falls der Baum noch leer ist, wird ein neuer Blattknoten erstellt und der Filter als erstes Datenobjekt dort eingefügt. Der neue Knoten wird zur Wurzel des `BloomFilterTree`. Andernfalls wird ausgehend vom Wurzelknoten rekursiv die optimale Position im Baum gesucht. Dazu werden optimale Teil- und Obermengen-IDs des Filters berechnet. Dem Filter wird die neue Teilmengen-ID zugewiesen und er wird gemäß der  $B^+$ -Baum-Regeln in den Baum eingefügt. Sind Teilmengen- und Obermengen-IDs unterschiedlich, wird ein zweites Datenobjekt mit der Obermengen-ID erstellt und ebenfalls in den Baum eingefügt. Falls der Baum während des Einfügens eine neue Ebene erhalten hat, wird der Elternknoten des alten Wurzelknoten zur neuen Wurzel. Abbildung 5.1 verdeutlicht den Ablauf: Die Hauptarbeit des Einfügens findet in den Blattknoten statt, wo die Teilmengen- und Obermengen-IDs der Filter berechnet werden. Zur Berechnung der Teilmengen-ID werden folgende Schritte ausgeführt:

1. Einsammeln aller Bloom-Filter-Objekte im Baum, von denen der einzufügende Filter eine Teilmenge ist.
2. Sortieren der gesammelten Filter nach Jaccard-Distanz in aufsteigender Reihenfolge.
3. Einsammeln aller freien IDs im Baum zwischen der kleinsten und größten ID.
4. Sortieren der freien IDs in aufsteigender Reihenfolge.
5. Falls der Filter von keinem Objekt im Baum eine Teilmenge ist, wird die kleinste freie ID zurückgegeben.
6. Andernfalls wird die optimale Teilmengen-ID bestimmt. Dazu werden zu allen in Schritt 2 gesammelten Filtern jeweils die nächstgrößere und nächstkleinere freie ID bestimmt. Diese „guten“ IDs werden in einem Vektor gesammelt.

#### 4. Implementierung

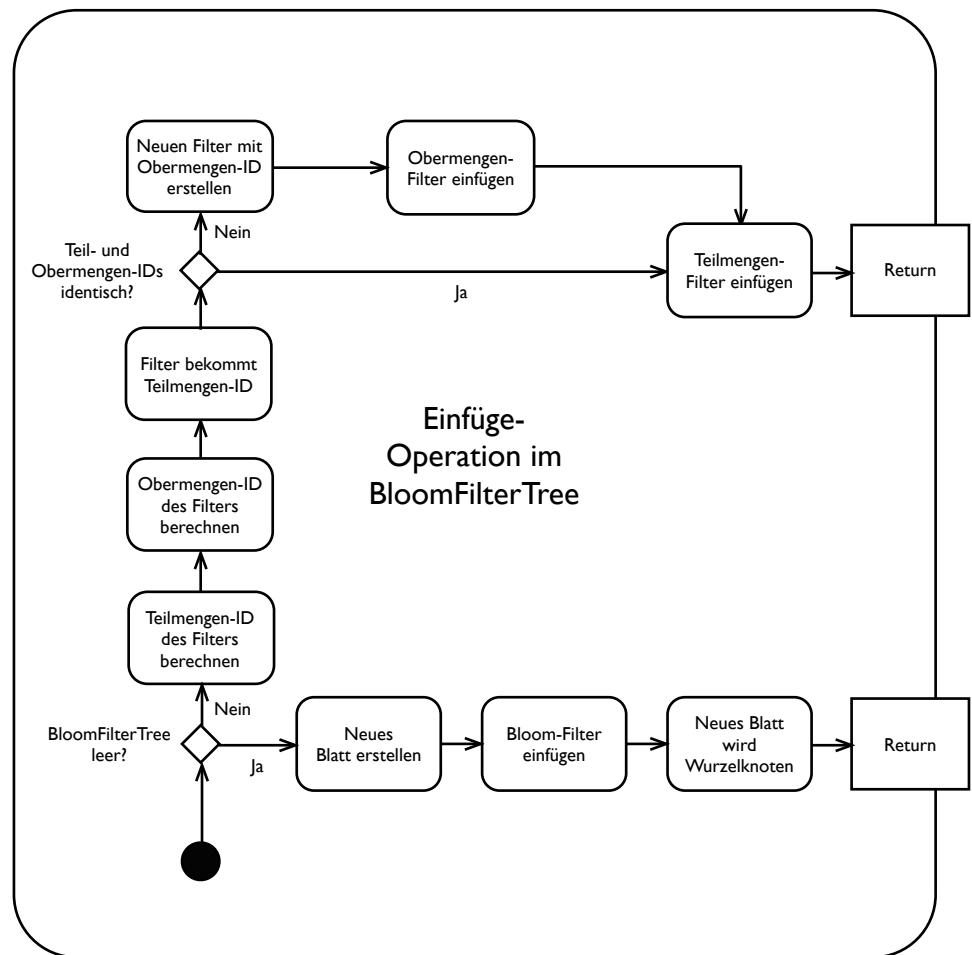


Abbildung 4.2.: Einfüge-Operation im BloomFilterTree.

7. Die „guten“ IDs werden nach Distanz zu einem Teilmengen-Filter des Anfragefilters sortiert.
8. Die ID mit der kleinsten Distanz zu einem Filter, von der der neu einzufügende Filter eine Teilmenge ist, wird als Ergebnis zurückgegeben.

Die Funktion `computeSubsetId()` der Klasse `BloomFilterLeaf` ist beispielhaft im Anhang abgedruckt (vgl. Kapitel A). Die Berechnung der optimalen Obermengen-ID geschieht analog dazu in der Funktion `computeSupersetId()`, nur dass dazu die Obermengen des neuen Filters betrachtet werden.

#### 4.1.4. *k*-nächste-Nachbarn-Suche

Das eben beschriebene Verfahren zum Einfügen von Objekten ist wesentlich aufwändiger als normalerweise beim  $B^+$ -Baum. Die Berechnung der optimalen Teil- und Obermengen-IDs für einen neuen Filter dient dazu, Filter mit bestehenden Teil- und Obermengenbeziehungen nahe beieinander im Baum abzuspeichern. Die *k*-nächste-Nachbarn-Suche musste natürlich darauf abgestimmt werden. Sie vergleicht nicht wie in der naiven Implementierung *k*-mal die Distanzen aller Bloom-Filter im Baum zum Anfragefilter und gibt die Filter mit den *k* kleinsten Distanzen zurück. Stattdessen werden die Vereinigungsfilter der Baumknoten danach untersucht, ob sie zum Anfragefilter in Teil- oder Obermengenbeziehung stehen. Ziel ist es, bei einer Anfrage nur den besten Zweig im Baum zu verfolgen.

Wie im vorigen Abschnitt dargestellt, werden die Filter beim Einfügen nach Teil- und Obermengenbeziehungen angeordnet. Der Vereinigungsfilter jedes Baumknotens enthält die Vereinigungsmenge aller Filter in seinem Teilbaum. Damit lässt sich die *k*-nächste-Nachbarn-Suche deutlich verkürzen.

Da die *k*-nächste-Nachbarn-Suche teurer und aufwändiger ist als eine Punktanfrage nach einem Objekt, wurden dafür zwei verschiedene Operationen implementiert. Die Punktanfrage geschieht in folgenden Schritten:

1. Falls der Baum leer ist, wird der Zeiger auf den Anfragefilter selbst zurückgegeben.
2. Andernfalls wird geprüft, ob der Wurzelknoten Teil- oder Obermenge des Anfragefilters ist.
3. Ist das nicht der Fall, wird eine normale nächste-Nachbarn-Suche auf dem Baum ausgeführt.
4. Andernfalls wird rekursiv beim Wurzelknoten beginnend geprüft, welche Vereinigungsfilter der Kindknoten Teil- oder Obermengen des Anfragefilters sind.
5. Gibt es mehrere davon, wird der Kindknoten mit der kleinsten Jaccard-Distanz des Vereinigungsfilters zum Anfragefilter bestimmt. Nur dieser Pfad wird weiter verfolgt.
6. Gibt es keine Teil- oder Obermengenbeziehungen zu den Vereinigungsfiltern der Kindknoten, wird eine nächste-Nachbarn-Suche auf dem restlichen Teilbaum durchgeführt.
7. Ist die Anfrage auf der Blattebene angekommen, wird der Filter mit der kleinsten Distanz im Blatt bestimmt und ein Zeiger darauf zurückgegeben.

Auch wenn ab einem bestimmten Punkt keine Teil- oder Obermengenbeziehungen mehr zwischen Anfragefilter und den Vereinigungsfiltern der Kindknoten des aktuellen Kno-

#### 4. Implementierung

ten bestehen, wird die Anfrage deutlich abgekürzt. Dann muss nur noch eine nächste-Nachbarn-Suche auf einem Teilbaum durchgeführt werden, nicht auf der gesamten Indexstruktur.

Hier findet offensichtlich die hauptsächliche Arbeit in den inneren Knoten des BloomFilterTree statt, den so genannten Index- oder Directoryknoten. Bei der  $k$ -nächste-Nachbarn-Suche findet hingegen wie zuvor ein großer Teil der Sortierarbeit auf Blattebene statt. Abbildung 5.2 zeigt den Ablauf im BloomFilterTree: Der Quellcode der Funktion `simQueryVec()`

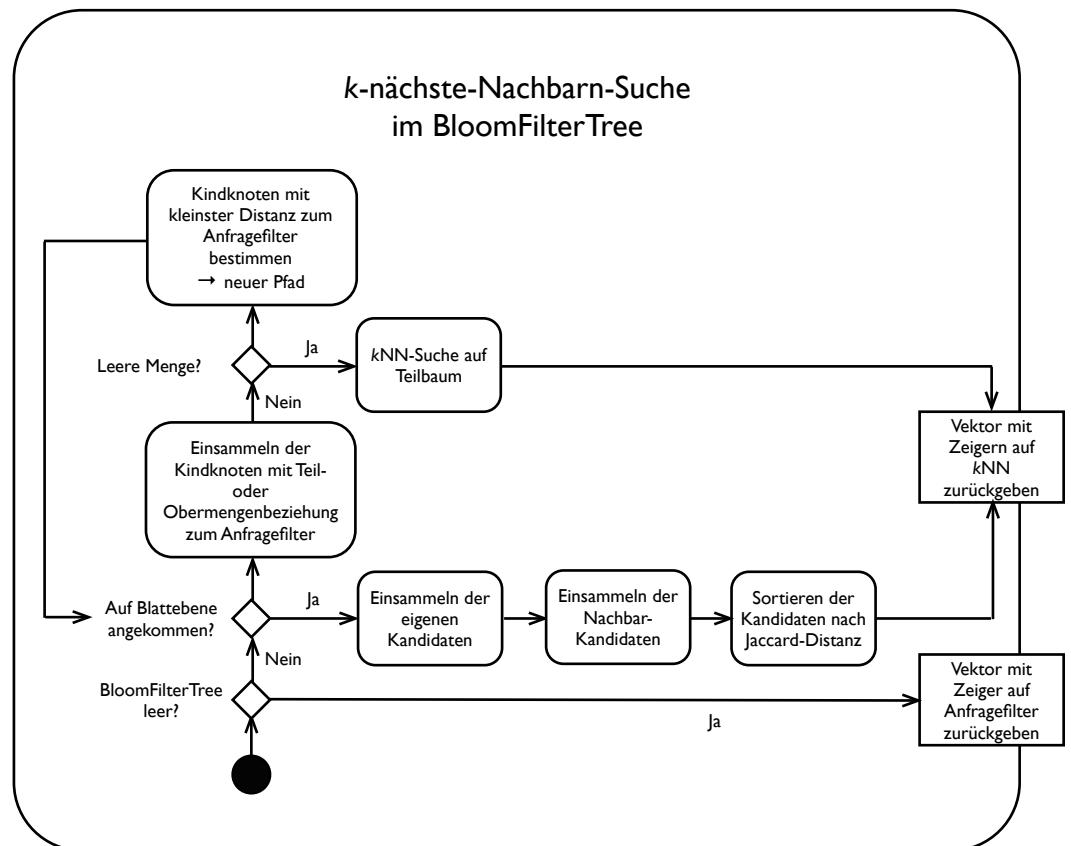


Abbildung 4.3.:  $k$ -nächste-Nachbarn-Suche im BloomFilterTree.

der Klasse `BloomFilterTree` ist beispielhaft im Anhang abgedruckt (vgl. Kapitel A). Wie die Evaluation in Kapitel 5 zeigt, lassen sich mit den vorgestellten Methoden Laufzeit und CPU-Zeit der  $k$ -nächste-Nachbarn-Suche deutlich reduzieren.

Es sei angemerkt, dass die  $B^+$ -Baum-spezifische Bereichssuche bei allen vorgestellten Funktionen von Nutzen ist. Die Einfüge-Operation wird z.B. dadurch erleichtert, dass zum Einsammeln aller freien IDs nur einmal die Blattebene traversiert werden muss. Andere Baumstrukturen benötigen dazu in der Regel eine deutlich komplexere Breiten- oder Tiefensuche. Bei der  $k$ -nächste-Nachbarn-Suche in einem Teilbaum muss ebenfalls

nur der Teilbereich zwischen einem Start- und Endwert traversiert werden. Das liegt daran, dass die Blätter eine doppelt verkettete Liste bilden und die Indexstruktur die Suchbaumeigenschaft auf den Primärschlüsseln, d.h. den Bloom-Filter-IDs, erfüllt.

## 4.2. Alternative Ansätze

Zum Abschluss des Kapitels werden zwei Ansätze vorgestellt, die alternativ zum Bloom-FilterTree bzw. zur Organisation nach Teil- und Obermengenbeziehungen verfolgt wurden. Letztlich erwies sich jedoch die Kombination aus Baumstruktur und Mengenbeziehungen am erfolgreichsten.

**Einfügen gemäß Jaccard-Distanzen** Anfangs wurde der Ansatz verfolgt, die Bloom-Filter wie bei Sakuma und Sato an Hand ihrer Distanzen im Baum zu organisieren (vgl. Abschnitt 3.3). Dabei stellte sich heraus, dass das dort verwendete Distanzmaß transitiv ist im Sinne von Abschnitt 2.2.1. Das trifft auf die Jaccard-Distanz nicht zu. Die Suche nach Alternativen ergab die Kombination aus Vereinigungsfilter und Transitivität der Teil- und Obermengenbeziehungen, die letztlich umgesetzt wurde.

**Doppelt verkettete Liste** Eine andere Idee war die Gruppierung der Filter nach Gleichheit von Teilsegmenten. Als Datenstruktur sollte eine doppelt verkettete Liste dienen mit derselben Anzahl an Listenknoten wie Positionen im Bloom-Filter. Die einzufügenden Bloom-Filter wurden in Segmente unterteilt. Ein Segment entsprach dabei einem Element im Daten-Array. Ein neuer Filter wurde wie folgt in die Datenstruktur eingefügt:

1. Jeder Listenknoten hat eine 0- und eine 1-Bit-Liste. Darin sind Zeiger auf die bereits eingefügten Filter gespeichert, die an derselben Bitposition eine 0 bzw. eine 1 enthalten.
2. Wird ein neues Objekt in die Liste eingefügt, wird an jedem Listenknoten die entsprechende Liste aktualisiert, je nachdem, ob der einzufügende Filter an der Position 0 oder 1 enthält.

Das Ziel war, bei der  $k$ -nächsten-Nachbarn-Suche die 0- bzw. 1-Bit-Listen der Listenknoten mit gleichem Segmentinhalt abzuprüfen. Alle darin enthaltenen Zeiger wurden in einem Vektor gesammelt, der nach Häufigkeit der Zeiger sortiert wurde. Es bestand die Hoffnung, dass sich die am häufigsten vorkommenden Zeigern und die nächsten Nachbarn des Anfragefilters proportional zueinander verhielten.

Eine solche Datenstruktur ist in Implementierung und Pflege weit weniger aufwändig als eine Baumstruktur. Es zeigte sich jedoch, dass damit nicht die gewünschten Ergebnisse erzielt werden konnten, da bei der  $k$ -nächste-Nachbarn-Suche zu viele Ergebnisse mit gleich vielen Zeigern gefunden wurden. Die Suche scheiterte somit an mangelnder Ergebnisqualität und die doppelt verkettete Liste wurde als Datenstruktur verworfen.



# 5. Evaluation

Das folgende Kapitel dient dem Vergleich zwischen der entwickelten Datenstruktur und der bisherigen Organisation der Bloom-Filter in AMBIENCE. Abschnitt 5.1 beschreibt zunächst den für die Evaluation verwendeten Datensatz. Dieser stammt nicht aus AMBIENCE, sondern die Bloom-Filter wurden wie in Abschnitt 4.1.2 beschrieben selbst implementiert. Der Versuchsaufbau, d.h. welche Aspekte der Indexstruktur untersucht und verglichen wurden, findet sich in Abschnitt 5.2. Die erzielten Ergebnisse werden in Abschnitt 5.3 vorgestellt und in Abschnitt 5.4 ausgewertet.

## 5.1. Datensatz

Obgleich nicht mit echten AMBIENCE-Daten gearbeitet wurde, wurde ein möglichst realistisches Szenario erstellt mit folgenden Parametern:

Filtergröße	256 Bit	512 Bit
Anzahl Bloom-Filter (m)	100	100
Anzahl eingefügte Objekte (n)	50	50
Anzahl Hashfunktionen (d)	4	8

Tabelle 5.1.: Datensatz für den Versuchsaufbau.

Als Wörterbuch wurde die Unix-Datei `words` verwendet, aus der jeweils 50 zufällige Einträge in die Bloom-Filter eingefügt wurden. Wie in Abschnitt 2.2.3 erläutert, wurden zum Einfügen Murmur-Hashfunktionen verwendet. Die Anzahl der zu verwendenden Hashfunktionen lässt sich berechnen als

$$d = \left\lceil \frac{m}{n} * \ln(2) \right\rceil.$$

Der  $i+1$ -te Hashwert wurde dabei jeweils aus dem  $i$ -ten Hashwert berechnet wie in Abschnitt 3.1 beschrieben. Den Bloom-Filtern wurden zunächst zufällige IDs zugewiesen. Sie wurden jeweils in einen BloomFilterTree mit 256 bzw. 512 Bit Filtergröße und in einen Bloom-Filter-Vektor<sup>1</sup> eingefügt, der die unsortierte Liste repräsentiert.

---

<sup>1</sup>Objekte vom Typ `std::vector<BloomFilter>`.

## 5.2. Versuchsaufbau

Der Versuchsaufbau vergleicht die Indexstruktur BloomFilterTree mit einer unsortierten Liste von Bloom-Filttern, die der bisherigen Implementierung in AMBIENCE entspricht. Für beide Filtergrößen wurden fünf Experimente durchgeführt:

1. Ergebnisqualität
2. CPU-Zeit
3. Speicherbedarf
4. Komplexität
5. Aufbaukosten

**Ergebnisqualität** Zur Ermittlung der Ergebnisqualität wurde eine nächste-Nachbarn-Suche und eine 3-nächste-Nachbarn-Suche auf dem BloomFilterTree durchgeführt. Die erzielten Ergebnisse wurden mit den Sollwerten einer regulären  $k$ -nächste-Nachbarn-Suche verglichen.

**CPU-Zeit** Zur Messung der CPU-Zeit wurde die C++-Bibliothek `chrono` verwendet. Es wurden jeweils die Ausführungszeiten der Funktionen `simQuery()` und `simQueryVec()` für einen bzw. 3 nächste Nachbarn (vgl. Abschnitt 4.1.4) ermittelt und mit den Ausführungszeiten einer regulären  $k$ -nächste-Nachbarn-Suche verglichen.

**Speicherbedarf** Da der BloomFilterTree nur Zeiger auf Bloom-Filter-Objekte enthält und nicht die Datenstrukturen selbst, ist sein Speicherbedarf sehr gering. Zur Ermittlung des Speicherbedarfs wurde daher von den tatsächlich allokierten Instanzen der Klasse `BloomFilter` ausgegangen. Das sind bei einer unsortierten Liste alle eingefügten Bloom-Filter, beim BloomFilterTree zusätzlich die Vereinigungsfilter aller Knoten. Diese Speicherbedarfe wurden für BloomFilterTree und unsortierte Liste ermittelt und gegenüber gestellt.

**Komplexität** Zur Ermittlung der Komplexität der unterschiedlichen Suchalgorithmen wurde die Anzahl Vergleiche, die zur Anfragebeantwortung notwendig sind, als Maß vorausgesetzt. Diese wurden für BloomFilterTree und unsortierte Liste ermittelt und verglichen. Für den BloomFilterTree wurden dazu Varianten der Funktionen `simQuery()` und `simQueryVec()` implementiert, die die Anzahl an durchgeföhrten Vergleichen exakt mitprotokollieren.

**Aufbaukosten** In diesem Experiment wurden die Kosten für den Aufbau der Daten- bzw. Indexstruktur. Als Maß hierfür wurde die Zeitkomplexität des Einfügens aller Elemente bestimmt. Diese liegt für Objekte vom Typ `/texttstd::vector` mit  $n$  Elementen durchschnittlich in  $O(1)$  für die verwendete Funktion `std::vector::push_back()`<sup>2</sup>. Beim BloomFilterTree setzt sich die Komplexität der Einfüge-Operation aus der Berechnung der optimalen Teilmengen- und Obermengen-IDs und dem tatsächlichen Einfügen des Elements zusammen. Wie in Abschnitt 4.1.3 dargestellt, ist die Berechnung der Teilmengen-IDs relativ rechenintensiv und deutlich teurer als das kanonische Einfügen im

---

<sup>2</sup>Vgl. [http://www.cplusplus.com/reference/vector/vector/push\\_back/](http://www.cplusplus.com/reference/vector/vector/push_back/).

$B^+$ -Baum. Die teuersten Operationen sind hierbei das Sortieren der freien und „guten“ IDs. Die Kosten für den Aufbau der Indexstruktur liegen damit insgesamt in  $O(n * \log(n))$  für einen BloomFilterTree mit  $n$  Elementen.

### 5.3. Ergebnisse

Im Folgenden werden die Ergebnisse der fünf Experimente mit den eingangs beschriebenen Parametern vorgestellt.

**Ergebnisqualität** Abbildungen 5.2 und 5.3 stellen die Ergebnisqualität der nächste-Nachbarn-Suche dar. Die Sollwerte sind darin grün, die Messwerte blau markiert. Stimmt der Messwert mit dem Sollwert überein, ist nur die blaue Markierung vorhanden. Der quadratische Fehler von Messwert gegenüber Sollwert ist in Rot angegeben. Abbildungen 5.3 – 5.6 stellen die Ergebnisqualität der 3-nächste-Nachbarn-Suche dar. Die Sollwerte sind darin in drei Grünstufen markiert, die Messwerte sind in drei Blaustufen markiert. Der mittlere quadratische Fehler der Messwerte gegenüber den Sollwerten ist in Rot angegeben.

**CPU-Zeit** Abbildungen 5.7 und 5.8 stellen die CPU-Zeit zur Ausführung der nächste-Nachbarn-Suche dar. Die Ergebnisse für den BloomFilterTree sind darin blau, die Ergebnisse für die unsortierte Liste grün markiert. Abbildungen 5.9 und 5.10 stellen die CPU-Zeit zur Ausführung der 3-nächste-Nachbarn-Suche dar. Die Ergebnisse für den BloomFilterTree sind darin blau, die Ergebnisse für die unsortierte Liste grün markiert.

**Speicherbedarf** Abbildung 5.11 stellt den Speicherbedarf der angelegten Objekte dar. Der Speicherbedarf für Objekte vom Typ BloomFilterTree ist darin blau markiert. Der Speicherbedarf für Objekte vom Typ `std::vector<BloomFilter>` ist grün markiert.

**Komplexität** Abbildung 5.12 stellt die Komplexität der k-nächste-Nachbarn-Suche wie in Abschnitt 5.2 beschrieben als Anzahl der zur Anfragebearbeitung nötigen Vergleiche dar. Die Ergebnisse für den BloomFilterTree sind darin blau, die Ergebnisse für die unsortierte Liste grün markiert.

**Aufbaukosten** Abbildung 5.13 stellt die Aufbaukosten der angelegten Objekte dar. Die Aufbaukosten für Objekte vom Typ BloomFilterTree sind darin blau markiert. Die Aufbaukosten für Objekte vom Typ `std::vector<BloomFilter>` sind grün markiert.

## 5. Evaluation

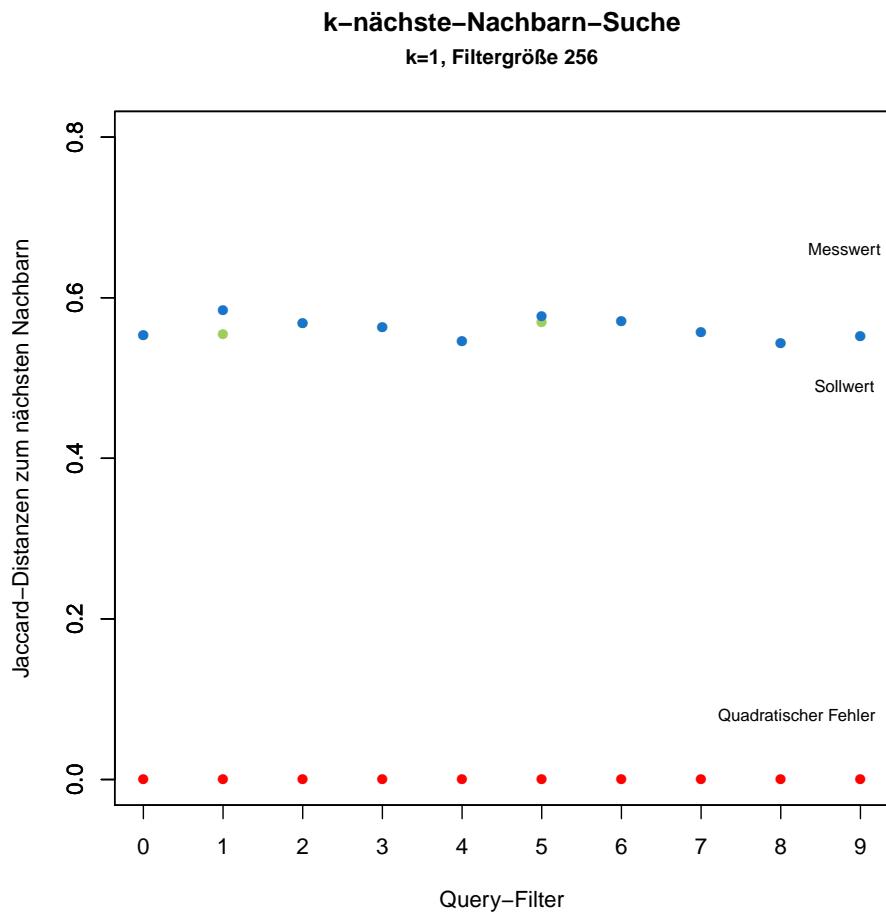


Abbildung 5.1.: Ergebnisqualität der nächste-Nachbarn-Suche für 256 Bit-Bloom-Filter.

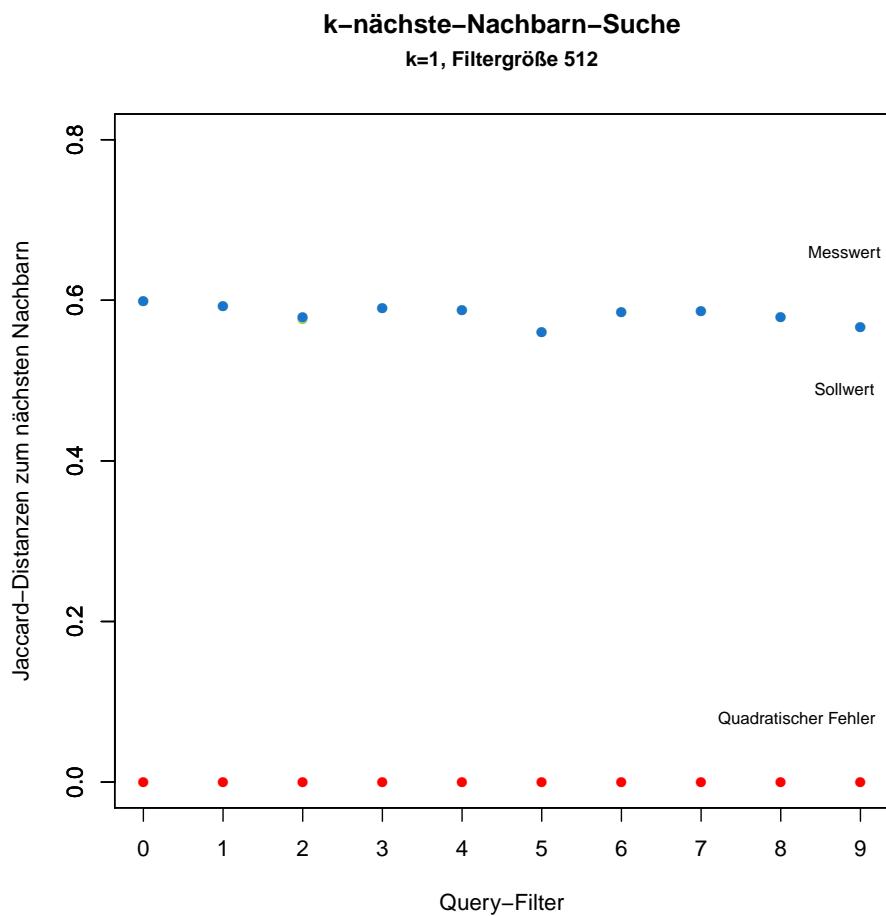


Abbildung 5.2.: Ergebnisqualität der nächste-Nachbarn-Suche für 512 Bit-Bloom-Filter.

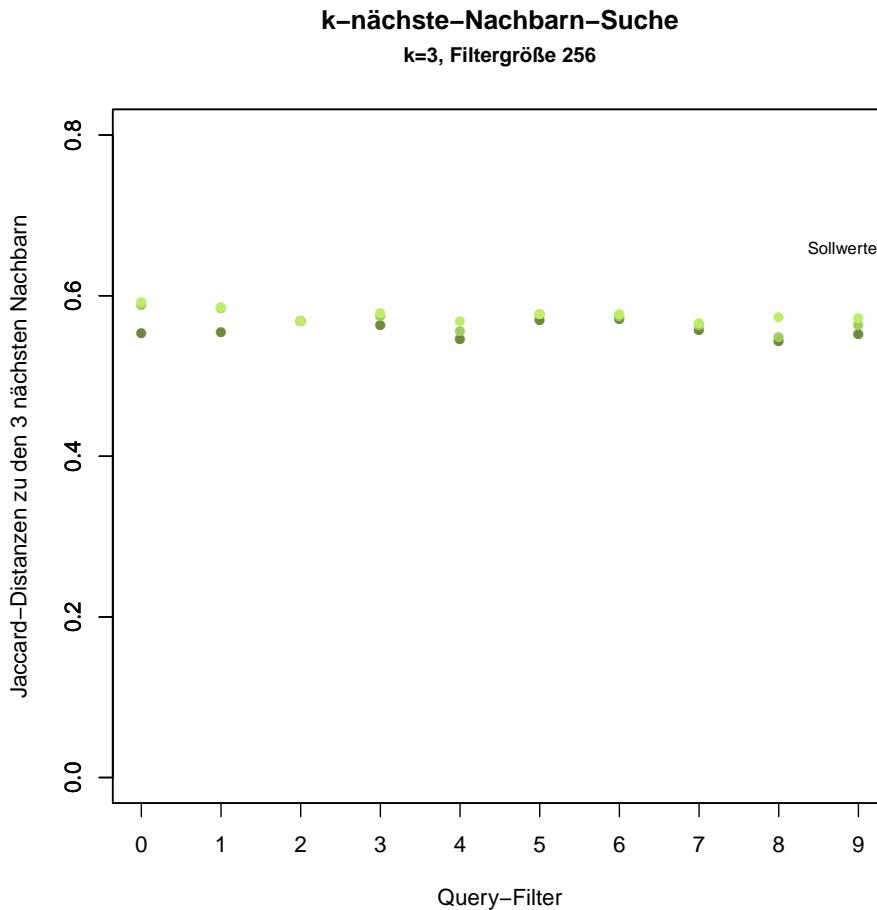


Abbildung 5.3.: Sollwerte der 3-nächste-Nachbarn-Suche für 256 Bit-Bloom-Filter.

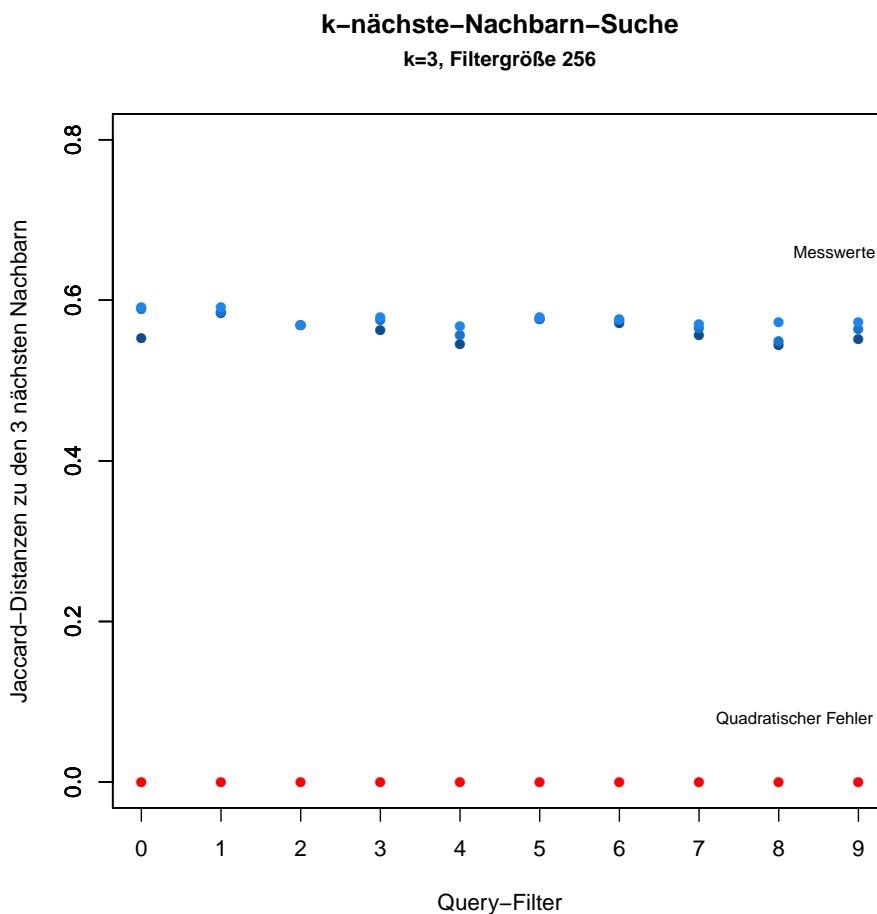


Abbildung 5.4.: Messwerte der 3-nächste-Nachbarn-Suche für 512 Bit-Bloom-Filter.

## 5. Evaluation

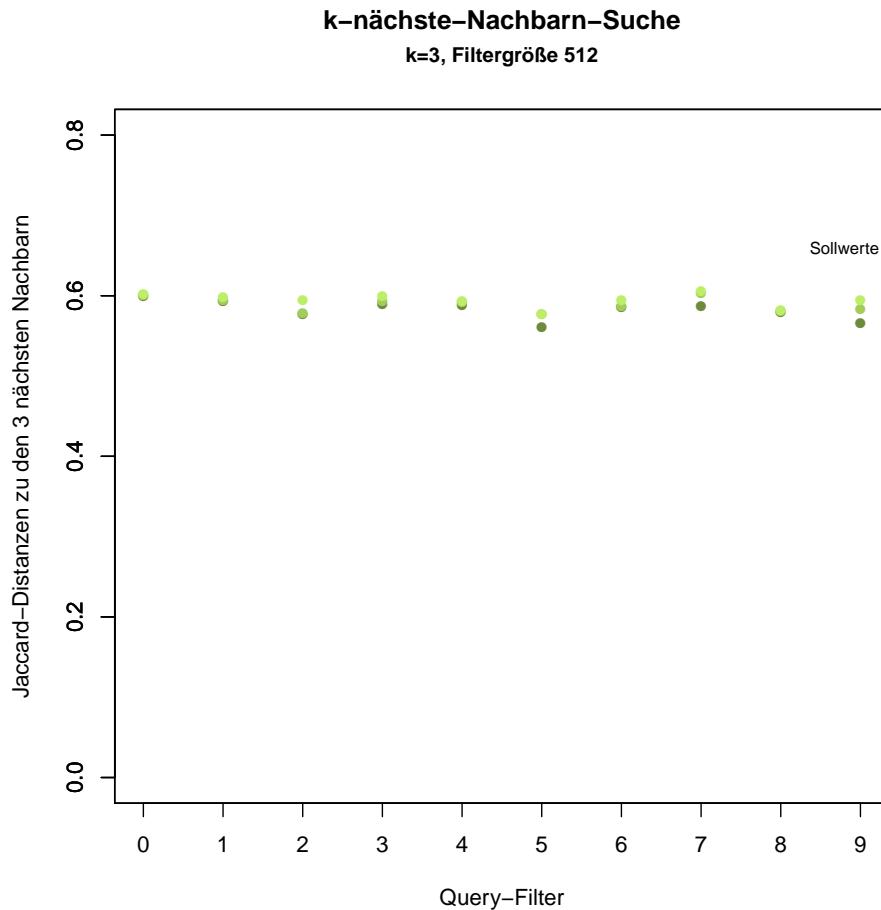


Abbildung 5.5.: Sollwerte der 3-nächste-Nachbarn-Suche für 512 Bit-Bloom-Filter.

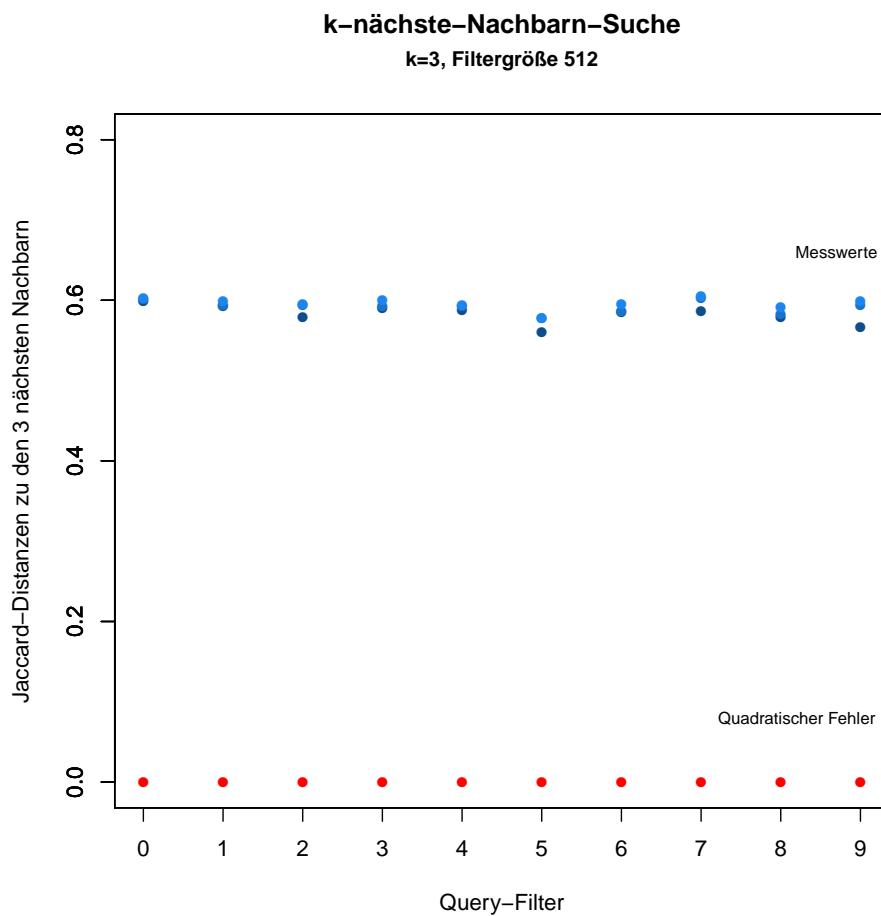


Abbildung 5.6.: Messwerte der 3-nächste-Nachbarn-Suche für 512 Bit-Bloom-Filter.

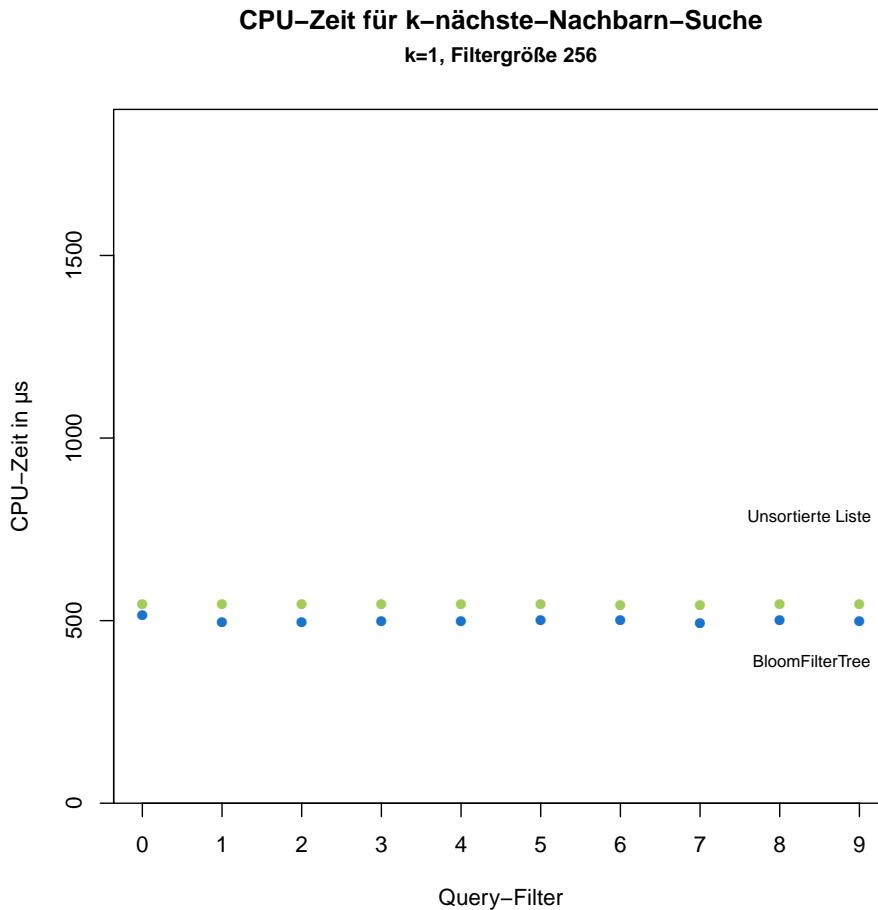


Abbildung 5.7.: CPU-Zeit für  $k$ -nächste-Nachbarn-Suche mit 256 Bit-Bloom-Filtern.

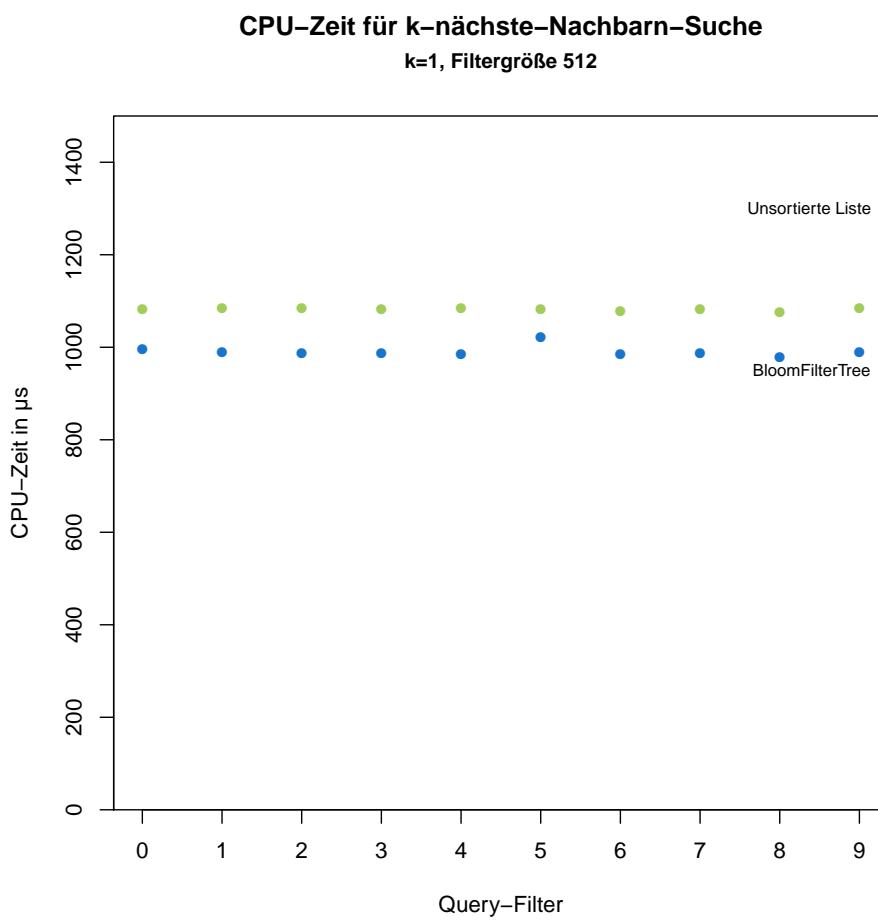


Abbildung 5.8.: CPU-Zeit für  $k$ -nächste-Nachbarn-Suche mit 512 Bit-Bloom-Filtern.

## 5. Evaluation

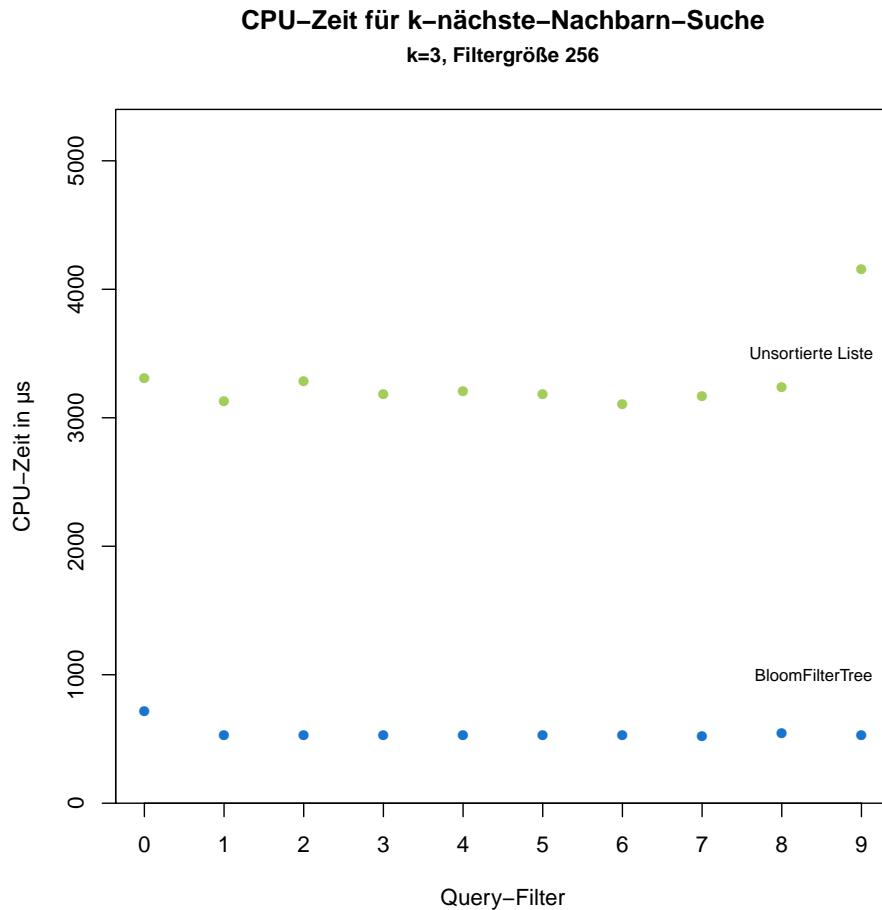


Abbildung 5.9.: CPU-Zeit für nächste-Nachbarn-Suche mit 256 Bit-Bloom-Filtern.

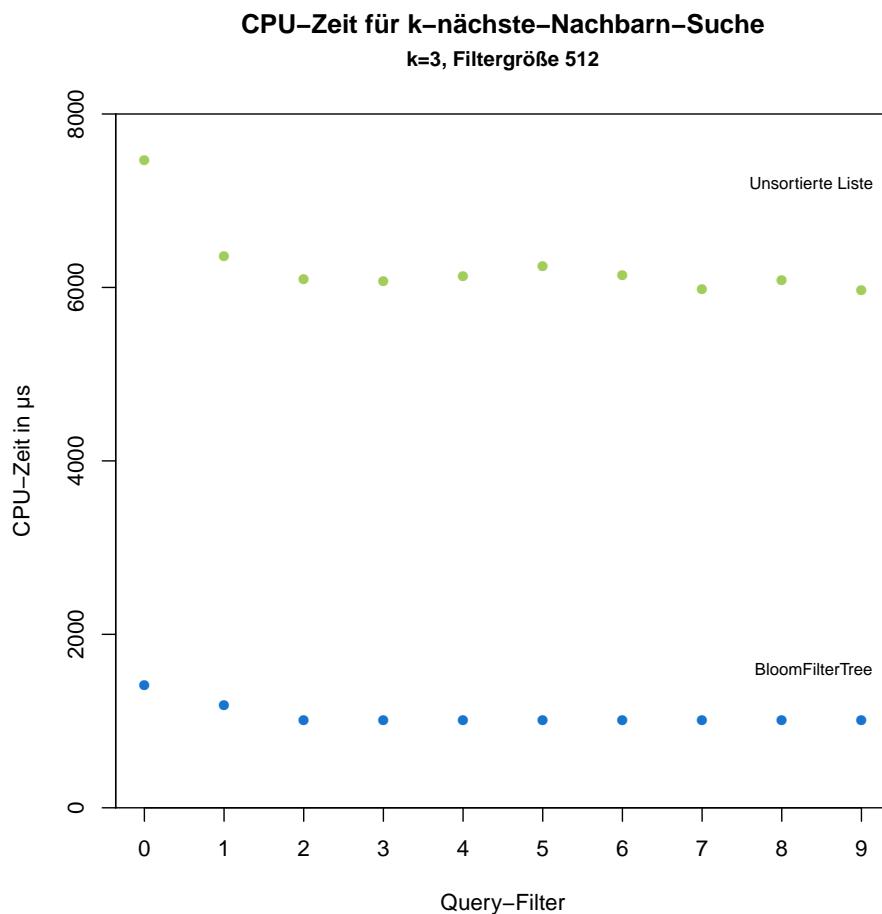


Abbildung 5.10.: CPU-Zeit für nächste-Nachbarn-Suche mit 512 Bit-Bloom-Filtern.

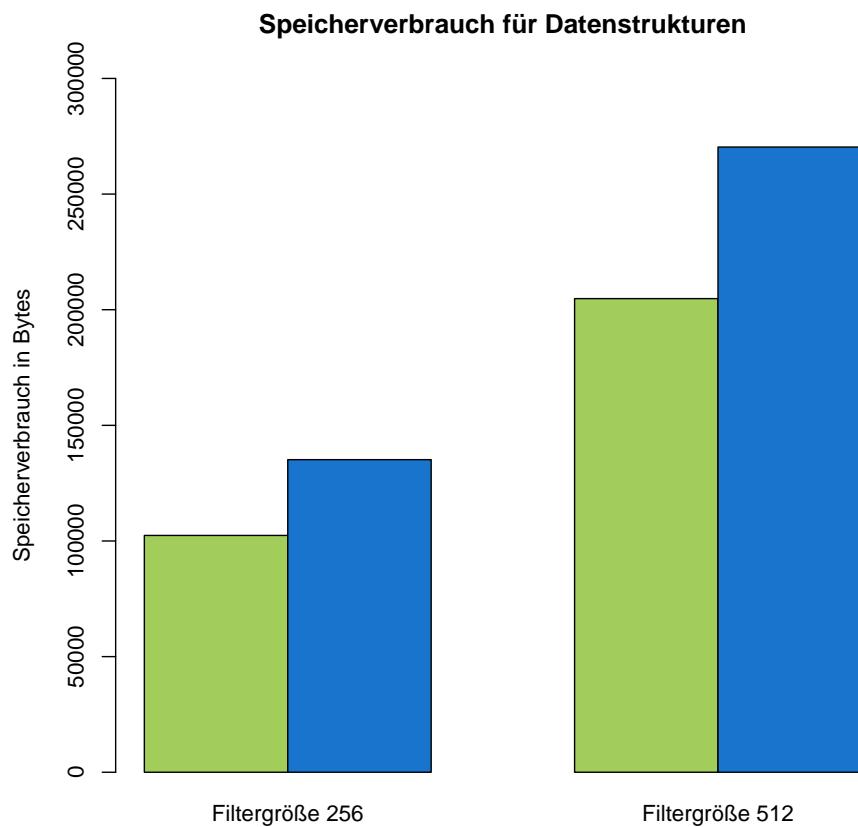


Abbildung 5.11.: Speicherbedarf für BloomFilterTree und unsortierte Liste.

## 5. Evaluation

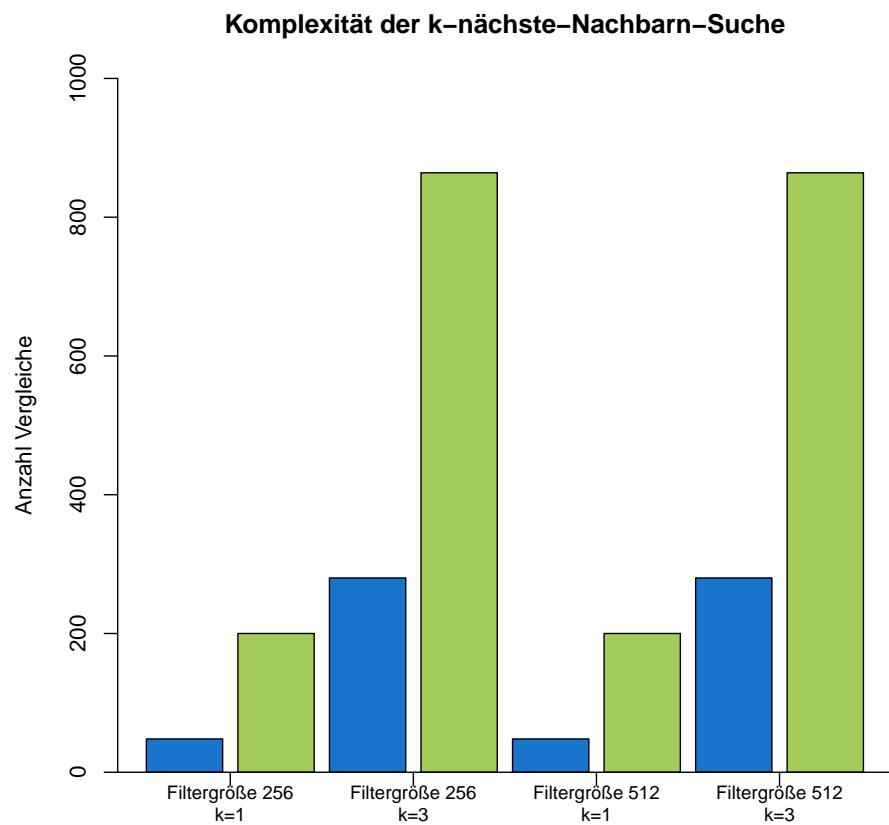


Abbildung 5.12.: Anzahl zur  $k$ -nächste-Nachbarn-Suche benötigter Vergleiche.

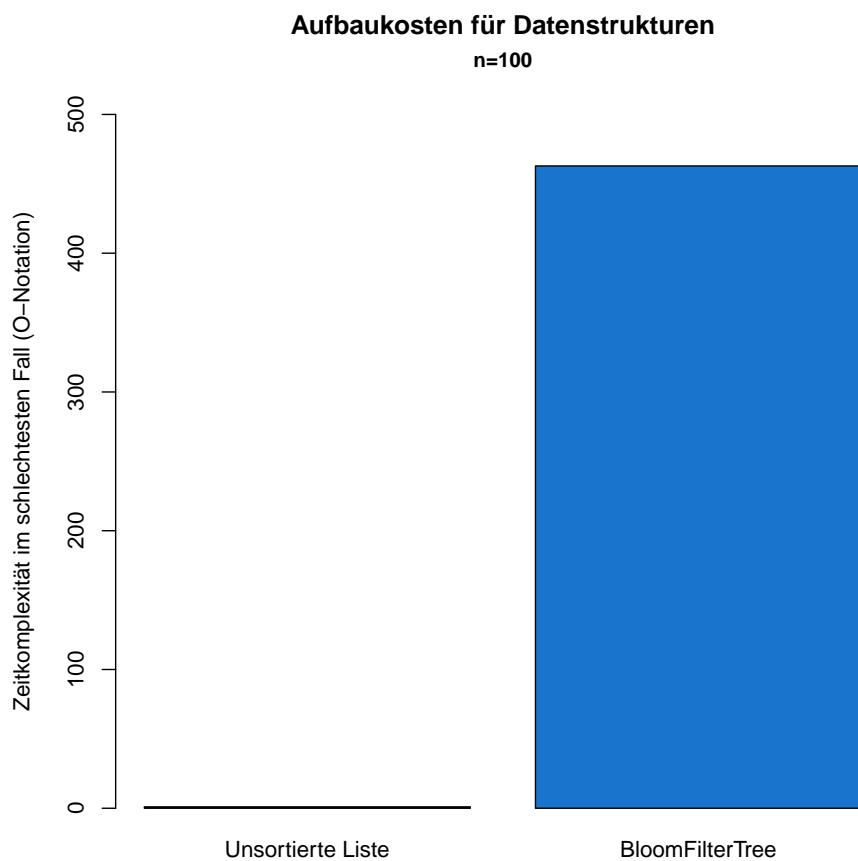


Abbildung 5.13.: Aufbaukosten für BloomFilterTree und unsortierte Liste.

## 5.4. Interpretation

Wie in Abschnitt 5.1 dargestellt, wurde die Evaluation mit zehn Anfragefiltern pro Experiment durchgeführt. Die soeben präsentierten Ergebnisse sind also dazu geeignet, den Mittelwert über einem Anfragevektor zu bilden und etwaige Ausreißer zu erkennen. Das gilt insbesondere für die Ergebnisse der CPU-Zeitmessung, die in der Regel wegen schwankender Nutzlast der verwendeten Maschine und nicht exakt vorhersehbaren CPU-Schedulings gewissen Schwankungen unterliegen. Demnach kann hierfür das Minimum der erzielten Werte als Benchmark für die  $k$ -nächste-Nachbarn-Suche angesehen werden.

**Ergebnisqualität** Die Ergebnisse der  $k$ -nächste-Nachbarn-Suche stimmen zum größten Teil mit den Sollwerten überein. Bei der nächsten-Nachbarn-Suche mit 256 Bit-Bloom-Filtern treten zwei abweichende Ergebnisse auf (vgl. Abbildung 5.1). Bei der 3-nächste-Nachbarn-Suche mit 256-Bit-Filtern treten bei 30 Ergebnissen fünf abweichende Einzelergebnisse auf. Bei der 3-nächste-Nachbarn-Suche mit 512-Bit-Filtern treten bei 30 Ergebnissen sechs abweichende Einzelergebnisse auf.

Der quadratische Fehler für diese Fälle beträgt bei der nächste-Nachbarn-Suche mit 256 Bit-Bloom-Filtern maximal 0.0008548022, andernfalls 0. Bei der 3-nächste-Nachbarn-Suche mit 256 Bit-Bloom-Filtern beträgt der mittlere quadratische Fehler bei abweichen den Messwerten maximal 0.0002956207, andernfalls 0. Bei der 3-nächste-Nachbarn-Suche mit 512 Bit-Bloom-Filtern beträgt er maximal 0.00008640455, andernfalls 0.

Das bedeutet: Nächste-Nachbarn-Anfragen werden mit dem entwickelten Verfahren in den meisten Fällen korrekt beantwortet. Mit einigen wenigen Fällen werden suboptimale Antworten zurückgegeben. Diese sind dennoch als „gute“ Antworten bezüglich des verwendeten Distanzmaßes und des maximalen quadratischen Fehlers zu bezeichnen. Dieses Resultat ist essentiell für die Bewertung des entwickelten Verfahrens. Es kann nur dann zuverlässig eingesetzt werden, wenn es in einem Großteil der Fälle zufriedenstellende Ergebnisse bzw. optimale Antworten liefert.

**CPU-Zeit** Die drastisch reduzierte CPU-Zeit ist als entscheidender Vorteil des entwickelten Verfahrens zu betrachten. In der verwendeten Versuchsanordnung kommt sie vor allem bei der 3-nächste-Nachbarn-Suche zum Tragen. Abbildung 5.14 stellt die jeweils erreichte CPU-Zeitersparnis gegenüber: Daran wird deutlich: Die Zeitersparnis wächst mit dem Parameter  $k$  und der Filtergröße, doch auch bei  $k = 1$  wird bereits CPU-Zeit eingespart. Es ist zu erwarten, dass bei größeren BloomFilterTree-Objekten, höheren Anzahlen an Filtern und ggf. größeren Filtern die Zeitersparnis noch deutlich gesteigert werden kann:

Beim hier verwendeten Versuchsaufbau werden i.d.R Bäume mit drei Ebenen aufgebaut, d.h. bei einer Punktanfrage wird auch bei bestehenden Teil- und Obermengenbeziehungen ein großer Teilbaum durchsucht bzw. ein großer Bereich der Blattebene traversiert. Bei größerer Höhe des Baums durch mehr eingefügte Filter kommen damit die Vorteile des entwickelten Verfahrens deutlicher zum Tragen. Wie in Abbildung 5.14 erkennbar, gilt ebenso für höhrere Filtergrößen. Die CPU-Zeitersparnis zeigt sich bei Anfragen mit  $k > 1$  besonders deutlich. Auch hier ist zu erwarten, dass sich die Vorteile des entwickelten Verfahrens bei höheren Bäumen und Filteranzahlen sowie größeren Filtern noch deutlich stärker zeigen.

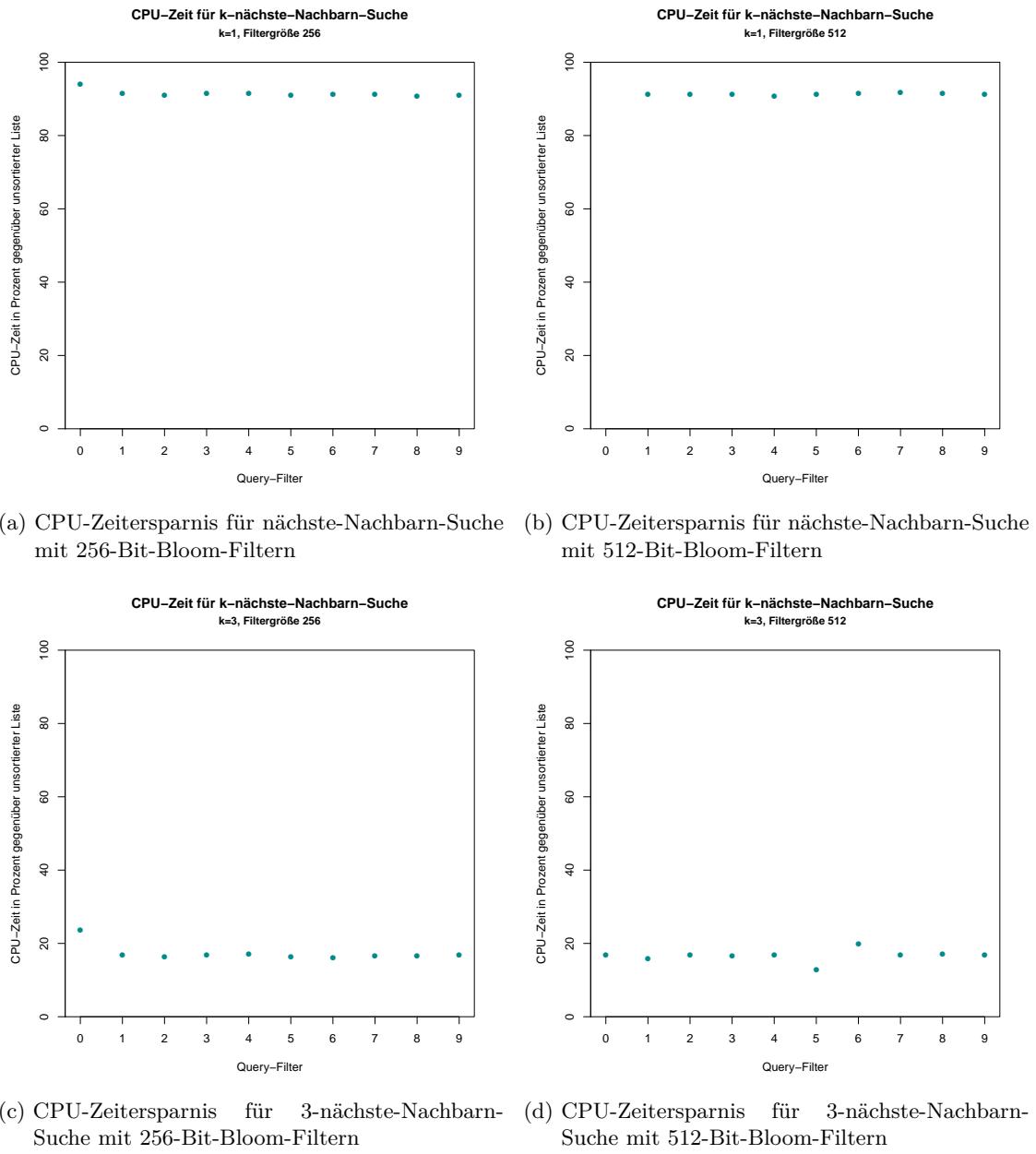


Abbildung 5.14.: CPU-Zeitersparnis für k-nächste-Nachbarn-Suche im BloomFilterTree.

## 5. Evaluation

Wie in Abschnitt 2.3.2 dargestellt, ist die Komplexität der Suchoperation im  $B^+$ -Baum abhängig vom Parameter  $t$  ab, d.h. dem Grad des Baumes. Die hier verwendeten BloomFilterTree-Objekte haben Grad 3, d.h. der maximale Fan-out beträgt 7. Dieser Parameter kann beim Anlegen eines BloomFilterTree-Objekts in der eigenen Implementierung frei gewählt werden. In der Praxis werden deutlich höhere Verzweigungsgrade verwendet, z.B. mit 10.000 Elementen pro innerem Knoten beim Einsatz in Datenbank-Management-Systemen. Es ist daher anzunehmen, dass die Anwendung für ein Szenario mit deutlich höheren Parameterwerten gut geeignet wäre.

**Speicherbedarf** Der zusätzliche Speicherbedarf beim BloomFilterTree gegenüber einem Bloom-Filter-Vektor ergibt sich durch die Allokierung eines Vereinigungsfilters für jeden Knoten. Er ist somit abhängig von der Anzahl der Knoten im Baum. Mit dem verwendeten Versuchsaufbau liegt er um 32% für den BloomFilter gegenüber einem Bloom-Filter-Vektor. Auch hier ließe sich durch eine geringere Knotenzahl, d.h. BloomFilterTree-Objekte mit höherem Verzweigungsgrad, noch Speicherplatz einsparen.

**Komplexität** Auch bezüglich der Zeitkomplexität, d.h. der Abschätzung der notwendigen Berechnungsschritte unabhängig von der gewählten Plattform, zeigt das entwickelte Verfahren erfreuliche Ergebnisse. Die Anzahl der nötigen Vergleiche bei der  $k$ -nächste-Nachbarn-Suche beträgt 24% bzw. 32% beim BloomFilterTree gegenüber der unsortierten Liste. Auch hier ist zu erwarten, dass sich die Ergebnisse bei höheren Bäumen noch verbessern lassen, da bei der  $k$ -nächste-Nachbarn-Suche nur der beste Pfad verfolgt wird. Bei höheren Bäumen werden somit mehr bzw. größere Teilbäume abgeschnitten und müssen nicht betrachtet werden.

**Aufbaukosten** Die Kosten für den Aufbau der Indexstruktur sind erwartungsgemäß ein Wermutstropfen. Auch wenn in der Praxis niedrigere Werte zu erwarten sind, da weniger als  $n$  freie und „gute“ IDs sortiert werden müssen, liegen die Aufbaukosten in  $O(n * \log(n))$ . Das ist offensichtlich ein starker Zuwachs gegenüber einer unsortierten Liste, die sich z.B. durch ein Objekt vom Typ `std::vector` realisieren lässt. Andererseits kann davon ausgegangen werden, dass die Aufbaukosten in AMBIENCE deutlich seltener anfallen als z.B. in einem Verteilten System mit häufigem Ausscheiden und Hinzukommen von Knoten wie in den Abschnitten 3.2 und 3.3 beschrieben.

## 6. Zusammenfassung und Ausblick

Die vorliegende Arbeit hatte das Ziel, das mengentheoretische Problem der  $k$ -nächste-Nachbarn-Suche durch die Organisation von Bloom-Filtern für kontextzentrische soziale Netze zu optimieren. Als Schnittstelle diente das kontextzentrische soziale Netz AMBIENCE, dessen Nachrichten als Bloom-Filter kodiert, übertragen und gespeichert werden.

Dazu wurde eine speziell auf AMBIENCE zugeschnittene Indexstruktur entwickelt, der BloomFilterTree. Sie basiert auf einem  $B^+$ -Baum, in den Objekte an Hand ihrer Teil- und Obermengenbeziehungen eingefügt werden. Die implementierte Variante der  $k$ -nächste-Nachbarn-Suche nutzt diese Mengenbeziehungen zwischen dem Anfrageobjekt und den Baumknoten, um nur den besten Suchpfad zu verfolgen und Teilbäume möglichst früh abzuschneiden. Als Ähnlichkeitsmaß wurde wie in AMBIENCE die Jaccard-Distanz zu Grunde gelegt, die jedoch mangels Transitivität nicht zur Organisation der Bloom-Filter herangezogen werden konnte.

Das entwickelte Verfahren zeigte auf dem Testdatensatz sehr gute Ergebnisse bezüglich Zeitkomplexität und CPU-Zeit, d.h. die  $k$ -nächste-Nachbarn-Suche lässt sich gegenüber der bestehenden Implementierung deutlich beschleunigen. Die Stärken des Verfahrens zeigen sich umso mehr, je größer die Datenstrukturen und Bloom-Filter sind und je mehr Bloom-Filter eingefügt werden. Somit ist zu erwarten, dass das Verfahren gute Einsatzmöglichkeiten für ein Szenario der echten Welt bietet, wenn z.B. das soziale Netzwerk AMBIENCE über den Status eines Prototypen hinaus wachsen soll.

Dies wäre auch der erste Ansatzpunkt für zukünftige Arbeiten. In der vorliegenden Arbeit wurde zwar ein möglichst realistisches Szenario gewählt und implementiert. Dennoch wäre es wichtig und interessant, ein Szenario mit deutlich höheren Parameterwerten zu untersuchen. Insbesondere bliebe zu ermitteln, ob die Vorteile der  $B^+$ -Baum-Indexstruktur, die vor allem im Bereich großer Datenbank-Managementsysteme eingesetzt wird, dann noch stärker zum Tragen kommen.

Die Entwicklung des BloomFilterTree konzentrierte sich auf die Operationen Einfügen und Löschen. Für diese wurden vom kanonischen  $B^+$ -Baum abweichende, eigene Algorithmen entwickelt und implementiert. Nicht betrachtet wurden jedoch Löschoperation und Restrukturierungskosten, wenn Filter aus dem Baum entfernt werden. Auch hier wären Erweiterungen denkbar und sinnvoll. Einen zusätzlichen Ansatzpunkt bietet der von Bayardo et al. entwickelte All-Pairs-Algorithmus. Dieser optimiert ein mit der  $k$ -nächste-Nachbarn-Suche verwandtes Problem, beruht aber auf einem abweichenden Distanzmaß und einem dynamischen Aufbau der Indexstruktur. Die Frage, ob sich All-Pairs für AMBIENCE anpassen und dort einsetzen ließe, erscheint spannend und wäre sicherlich der Untersuchung wert.



# A. Anhang

## Die Klasse BloomFilterTree

```
// BloomFilterTree.hpp, Judith Greif
// Description: Header for class BloomFilterTree

#ifndef BloomFilterTree_hpp
#define BloomFilterTree_hpp

#include "BloomFilterNode.hpp"
#include "BloomFilterIndexNode.hpp"
#include "BloomFilterLeaf.hpp"

using namespace std;

class BloomFilterTree {

private:
    int t;                                // Order = minimum degree
    int filtersize;                         // Size of associated Bloom filters (# of bits)

public:
    BloomFilterNode *root;                  // Pointer to root node
    BloomFilterTree(int _t, int _s);
    ~BloomFilterTree();
    BloomFilterNode *getRoot();

    // Tree management
    void traverse();
    void traverseFilters();
    double computeMinJaccard(BloomFilter *filter);
    double computeMaxJaccard(BloomFilter *filter);
    int getMinJaccardKey(BloomFilter *filter);
    BloomFilter *getMinJaccardFilter(BloomFilter *filter);
    int getMinKey();
    int getMaxKey();
    vector<BloomFilter> collectAllFilters();
    int countFilters();
    int countUnionFilters();
    int computeSubsetId(BloomFilter *filter);
    int computeSupersetId(BloomFilter *filter);
    bool contains(int k);
    BloomFilterNode *search(int k);
    vector<pair<int, double>>computeAllDistances(BloomFilter *filter);
    vector<pair<int, double>>computekDistances(BloomFilter *filter, int k);
    int countLeaves();
}
```

## A. Anhang

```
// Measurement and comparison
vector<pair<BloomFilter, double>> compare(BloomFilter *filter, int k);
vector<int> compareMem();
vector<double> compareConstrCost();
vector<int> compareComplSimQuery(BloomFilter *filter);
vector<int> compareComplSimQueryVec(BloomFilter *filter, int k);

// Insertion
void insert(BloomFilter *filter);
void insertAsSets(BloomFilter *filter);

// Similarity queries
BloomFilter *simQuery(BloomFilter *filter);
vector<BloomFilter*> simQueryVec(BloomFilter *filter, int k);
};

#endif
```

## Die Klasse BloomFilter

```
// BloomFilter.hpp, Judith Greif
// Description: Header for class BloomFilter

#ifndef BloomFilter_hpp
#define BloomFilter_hpp

#include <iostream>
#include <vector>
#include <cstdlib>
#include <random>
#include <math.h>
#include <string>
#include <functional>

using namespace std;

const int NUM_FILTERS = 100;
const int NUM_ELEMENTS = 50;
const int NUM_QUERYFILTERS = 10;
const int seed = rand();

class BloomFilter {

private:
    int id;
    int count;           // # of elements inserted
    int size;            // # of bits
    int d;               // # of hash functions
    int *data;

public:
    BloomFilter();
    BloomFilter(const BloomFilter& fSource);
    BloomFilter(int _id, int _size);
    ~BloomFilter();
    BloomFilter & operator = (const BloomFilter &fSource);
    void setId(int value);
    int getId();
    int getSize();
    void setValue(int index, int value);
    int *getData();
    void printData();
    void printArr();
    void initRandom();
    double fractionOfZeros();
    double eSize();
    BloomFilter *logicalOr(BloomFilter *filter);
    BloomFilter *logicalAnd(BloomFilter *filter);
    bool isSubset(BloomFilter *filter);
    bool isSuperset(BloomFilter *filter);
    int mySupersetCount();
}
```

## A. Anhang

```
int mySubsetCount();
int binomialCoefficient(int n, int k);
int setOnes();
int setZeros();
int validOnes();
int possibleFreeZeros();
int possibleAddedOnes();
double setUnion(BloomFilter *filter) const;
double setIntersection(BloomFilter *filter) const;
double computeAmbienceJaccard(BloomFilter *filter);
double computeJaccard(BloomFilter *filter) const;
double eUnion(BloomFilter *filter);
double eIntersect(BloomFilter *filter);
void add(string &elem);
void increment();
int getNumHashes();
bool checkCorrectFillDegree();
};

#endif
```

## Die Funktion `computeSubsetId()` der Klasse `BloomFilterLeaf`

```
int BloomFilterLeaf::computeSubsetId(BloomFilter *filter) {
    vector<pair<int, double>> subsets;
    vector<int> freeIds;
    vector<pair<int, int>> goodIds;
    BloomFilterLeaf *tmp = this;
    double jacc;
    int minId = filters[0]->getId()-1;
    int maxId;
    int optId;
    bool no_subsets = true;
    while (tmp != NULL) {

        // Collect all filters that filter is subset of
        for (int i=0; i<tmp->getCount(); i++) {
            if ((tmp->filters[i])->isSubset(filter)) {
                jacc = computeJaccard(tmp->filters[i], filter);
                subsets.push_back(make_pair(tmp->filters[i]->getId(), jacc));
                no_subsets = false;
            }
        }
        maxId = tmp->filters[tmp->getCount()-1]->getId()+1;
        tmp = tmp->getNext();
    }

    // Sort subsets by jacc distances in ascending order
    sort(subsets.begin(), subsets.end(), [](const pair<int, double> &left,
    const pair<int, double> &right) {
        return left.second < right.second;
    });

    // Collect free ids
    tmp = this;
    freeIds.push_back(minId);
    freeIds.push_back(maxId);
    while (tmp != NULL) {
        for (int i=0; i<tmp->getCount()-2; i++) {
            for (int j=tmp->filters[i]->getId()+1; j<tmp->filters[i+1]->getId(); j++) {
                if (j<tmp->filters[i+1]->getId()) {
                    freeIds.push_back(j);
                }
            }
        }
        if (tmp->getCount() < tmp->getMax()) {
            if (tmp->getNext() != NULL) {
                int start = tmp->filters[tmp->getCount()-1]->getId()+1;
                int last = tmp->getNext()->filters[0]->getId();
                for (int j=start; j<last; j++) {
                    freeIds.push_back(j);
                }
            }
        }
    }
}
```

## A. Anhang

```

        }
        tmp = tmp->getNext();
    }
    sort(freeIds.begin(), freeIds.end(), less<int>());
}

// If there are no subsets, return smallest free id as pair with numerical distance 0
if (no_subsets == false) {

    // Determine optimal id
    // Check subsets in ascending order
    // Get next greater and smaller id
    int distNeg = subsets[0].first - minId;
    int distPos = maxId - subsets[0].first;
    for (int i=0; i<subsets.size(); i++) {
        optId = subsets[i].first;
        int j=0;
        while (freeIds[j] < subsets[i].first) {
            if (optId-freeIds[j] < optId-minId) {
                minId = freeIds[j];
                distNeg = optId-minId;
            }
            j++;
        }

        j=freeIds.size()-1;
        while (freeIds[j] > subsets[i].first) {
            if (freeIds[j]-optId < maxId-optId) {
                maxId = freeIds[j];
                distPos = maxId-optId;
            }
            j--;
        }
        goodIds.push_back(make_pair(minId, distNeg));
        goodIds.push_back(make_pair(maxId, distPos));
    }

    // Sort next smaller and greater ids by numerical distance in ascending order
    sort(goodIds.begin(), goodIds.end(), [](const pair<int, int> &left,
        const pair<int, int> &right) {
        return left.second < right.second;
    });
}
else {
    goodIds.push_back(make_pair(freeIds[0], 0));
}

// Return first element
return goodIds[0].first;
}

```

### Die Funktion simpleSimQueryVec() der Klasse BloomFilterLeaf

```
vector<BloomFilter*> BloomFilterLeaf::simQueryVec(BloomFilter *filter, int k) {
```

```

vector<BloomFilter*> results;
vector<pair<BloomFilter*, double>> distances;
double jacc;

// Collect all candidates
// Collect own candidates
for (int i=0; i<getCount(); i++) {
    jacc = computeJaccard(filters[i], filter);
    distances.push_back(make_pair(filters[i], jacc));
}

// Collect candidates from previous leaf
if (prev != NULL) {
    for (int i=0; i<prev->getCount(); i++) {
        jacc = computeJaccard(prev->filters[i], filter);
        distances.push_back(make_pair(prev->filters[i], jacc));
    }
}

// Collect candidates from next leaf
if (next != NULL) {
    for (int i=0; i<next->getCount(); i++) {
        jacc = computeJaccard(next->filters[i], filter);
        distances.push_back(make_pair(next->filters[i], jacc));
    }
}

// Sort candidates by jaccard distance in ascending order
sort(distances.begin(), distances.end(), [] (const pair<BloomFilter*, double> &left,
const pair<BloomFilter*, double> &right) {
    return left.second < right.second;
});

// If tree does not hold enough filters, return all results
if (distances.size() < k) {
    for (int i=0; i<distances.size(); i++) {
        results.push_back(distances[i].first);
    }
}
else {

    // Return first k Bloom filters from distances vector
    for (int i=0; i<k; i++) {
        results.push_back(distances[i].first);
    }
}
return results;
}

```



# Abbildungsverzeichnis

2.1.	Bloom-Filter (Bildnachweis: <a href="https://commons.wikimedia.org/wiki/File:Bloom_filter.svg">https://commons.wikimedia.org/wiki/File:Bloom_filter.svg</a> ) . . . . .	5
2.2.	Jaccard-Distanzen zwischen Bloom-Filtern . . . . .	6
2.3.	Teil- und Obermengenbeziehungen zwischen Bloom-Filtern . . . . .	7
2.4.	Ein B-Baum der Ordnung 2 (Bildnachweis: [Kri14]: 9) . . . . .	10
2.5.	Ein B <sup>+</sup> -Baum der Ordnung 2 (Bildnachweis: [Kri14]: 10) . . . . .	11
3.1.	Bloom-Filter-Baum bei Sakuma und Sato (Bildnachweis: [SS11]: 320) . . . . .	17
4.1.	Aufbau eines BloomFilterTree . . . . .	22
4.2.	Einfüge-Operation im BloomFilterTree . . . . .	24
4.3.	$k$ -nächste-Nachbarn-Suche im BloomFilterTree . . . . .	26
5.1.	Ergebnisqualität der nächste-Nachbarn-Suche für 256 Bit-Bloom-Filter . . . . .	32
5.2.	Ergebnisqualität der nächste-Nachbarn-Suche für 256 Bit-Bloom-Filter . . . . .	32
5.3.	Sollwerte der 3-nächste-Nachbarn-Suche für 256 Bit-Bloom-Filter . . . . .	33
5.4.	Messwerte der 3-nächste-Nachbarn-Suche für 512 Bit-Bloom-Filter . . . . .	33
5.5.	Sollwerte der 3-nächste-Nachbarn-Suche für 512 Bit-Bloom-Filter . . . . .	34
5.6.	Messwerte der 3-nächste-Nachbarn-Suche für 512 Bit-Bloom-Filter . . . . .	34
5.7.	CPU-Zeit für nächste-Nachbarn-Suche mit 256 Bit-Bloom-Filtern . . . . .	35
5.8.	CPU-Zeit für nächste-Nachbarn-Suche mit 512 Bit-Bloom-Filtern . . . . .	35
5.9.	CPU-Zeit für 3-nächste-Nachbarn-Suche mit 256 Bit-Bloom-Filtern . . . . .	36
5.10.	CPU-Zeit für 3-nächste-Nachbarn-Suche mit 512 Bit-Bloom-Filtern . . . . .	36
5.11.	Speicherbedarf für BloomFilterTree und unsortierte Liste . . . . .	37
5.12.	Anzahl zur $k$ -nächste-Nachbarn-Suche benötigter Vergleiche . . . . .	38
5.13.	Aufbaukosten für BloomFilterTree und unsortierte Liste . . . . .	39
5.14.	CPU-Zeitersparnis für $k$ -nächste-Nachbarn-Suche im BloomFilterTree . . . . .	41



# Literaturverzeichnis

- [ADI<sup>+</sup>12] AHLGREN, BENGT, CHRISTIAN DANNEWITZ, CLAUDIO IMBRENDA, DIRK KUTSCHER und BÖRJE OHLMAN: *A Survey of Information-Centric Networking*. Communications Magazine, IEEE, 50(7):26–36, 2012.
- [AT06] AGARWAL, SACHIN und ARI TRACHTENBERG: *Approximating the number of differences between remote sets*. In: *Information Theory Workshop, 2006. ITW '06 Punta del Este. IEEE*, Seiten 217–221, March 2006.
- [BCM02] BYERS, JOHN, JEFFREY CONSIDINE und MICHAEL MITZENMACHER: *Fast Approximate Reconciliation of Set Differences*. In: *BU Computer Science TR*, Seiten 2002–2019, 2002.
- [Blo70] BLOOM, BURTON H.: *Space/Time Trade-offs in Hash Coding with Allowable Errors*. Communications of the ACM, 13(7):422–426, 1970.
- [BM04] BRODER, ANDREI und MICHAEL MITZENMACHER: *Network Applications of Bloom Filters: A Survey*. Internet Mathematics, 1(4):485–509, 2004.
- [BMS07] BAYARDO, ROBERTO J., YIMING MA und RAMAKRISHNAN SRIKANT: *Scaling Up All Pairs Similarity Search*. In: *Proceedings of the 16th international conference on World Wide Web*, Seiten 131–140. ACM, 2007.
- [DA99] DEY, ANIND K. und GREGORY D. ABOWD: *Towards a Better Understanding of Context and Context-Awareness*. In: *Handheld and Ubiquitous Computing*, Seiten 304–307. Springer, 1999.
- [FCAB00] FAN, LI, PEI CAO, JUSSARA ALMEIDA und ANDREI BRODER: *Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol*. IEEE/ACM Transactions on Networking (TON), 8(3):281–293, 2000.
- [HP94] HELLERSTEIN, JOSEPH M. und AVI PFEFFER: *The RD-Tree: An Index Structure for Sets*. Technischer Bericht, University of Wisconsin-Madison, Computer Sciences Department, 1994.
- [Jan95] JANNINK, JAN: *Implementing Deletion in B+-Trees*. ACM Sigmod Record, 24(1):33–38, 1995.
- [KM06] KIRSCH, ADAM und MICHAEL MITZENMACHER: *Less Hashing, Same Performance: Building a Better Bloom Filter*. In: *European Symposium on Algorithms*, Seiten 456–467. Springer, 2006.
- [Knu99] KNUTH, DONALD E.: *The Art of Computer Programming*, Band 3. Addison Wesley Longman, 1999.
- [Kri14] KRIEGEL, HANS-PETER: *Skript zur Vorlesung Anfragebearbeitung und Indexstrukturen in Datenbanksystemen*. 1994–2014.
- [LC86] LEHMAN, TOBIN J. und MICHAEL J. CAREY: *A Study of Index Structures for Main Memory Database Management Systems*. In: *Proc. VLDB*, 1986.
- [Mit02] MITZENMACHER, MICHAEL: *Compressed Bloom Filters*. IEEE/ACM Transactions on Networking (TON), 10(5):604–612, 2002.
- [OW12] OTTMANN, THOMAS und PETER WIDMAYER: *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 5 Auflage, 2012.
- [SAW94] SCHILIT, BILL, NORMAN ADAMS und ROY WANT: *Context-aware Computing Applications*. In: *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, Seiten 85–90. IEEE, 1994.

## Literaturverzeichnis

- [SCLR09] STEIN, CLIFFORD, THOMAS H. CORMEN, CHARLES E. LEISERSON und RONALD L. RIVEST: *Introduction to Algorithms*. MIT Press, 3rd Auflage, 2009.
- [SS11] SAKUMA, HIROSHI und FUMIAKO SATO: *Evaluation of the Structured Bloom Filters Based on Similarity*. In: *Advanced Information Networking and Applications (AINA), 2011 IEEE International Conference on*, Seiten 316–323, März 2011.
- [STT<sup>+</sup>09] SHIRAKI, TORU, YUICHI TERANISHI, SUSUMU TAKEUCHI, KANAME HARUMOTO und SHOJIRO NISHIO: *A Bloom Filter-Based User Search Method Based on Movement Records for P2P Network*. In: *Applications and the Internet, 2009. SAINT '09. Ninth International Symposium on*, Seiten 177–180. IEEE, Juli 2009.
- [WDS15] WERNER, MARTIN, FLORIAN DORFMEISTER und MIRCO SCHÖNFELD: *AMBIENCE: A Context-Centric Online Social Network*. In: *12th IEEE Workshop on Positioning, Navigation and Communications (WPNC '15)*, 2015.
- [YL02] YANG, CONGJUN und KING-IP LIN: *An Index Structure for Improving Closest Pairs and Related Join Queries in Spatial Databases*. In: *Database Engineering and Applications Symposium, 2002. Proceedings. International*, Seiten 140–149. IEEE, 2002.
- [Zha12] ZHANG, ZHENGHAO: *Analog Bloom Filter: Efficient simultaneous query for wireless networks*. In: *Global Communications Conference (GLOBECOM), 2012 IEEE*, Seiten 3340–3346. IEEE, 2012.
- [ZJW04] ZHU, YIFENG, HONG JIANG und JUN WANG: *Hierarchical Bloom Filter Arrays (HBA): A Novel, Scalable Metadata Management System for Large Cluster-based Storage*. In: *Cluster Computing, 2004 IEEE International Conference on*, Seiten 165–174. IEEE, 2004.