Informatik
Hauptcampus

HOCHSCHULE TRIER

HyperLogLog

Analyse eines Algorithmus zur Schätzung von Kardinalitäten

Marius Müller

Seminararbeit Big-Data-Technologien

Betreuer: Prof. Dr. Christoph Schmitz

Trier, 09.02.2023

Kurzfassung

In dieser Seminararbeit wird der Algorithmus HyperLogLog zur Lösung des Count-Distinct-Problems vorgestellt, bei dem die Kardinalität einer Datenmenge, also die Anzahl der unterscheidbaren Elemente dieser Menge, bestimmt werden sollen. HyperLogLog kann Kardinalitäten bestimmen, ohne dass dabei die gesamte Menge im Speicher gehalten werden muss, indem die Eingabewerte mit Hilfe einer Hashfunktion randomisiert und anschließend auf bestimmte Bitmuster überprüft werden. Der Algorithmus gibt eine Datenstruktur zurück, die als HyperLogLog Sketch bezeichnet wird und die sich durch ihren geringen Speicherplatzbedarf und einfache Additivität auszeichnet. Im Rahmen dieser Arbeit wird gezeigt, dass mit dem Algorithmus HyperLogLog Kardinalitäten über 10^9 mit einer Genauigkeit von 2% bestimmt werden können, wobei nur $1,5kB$ an zusätzlichem Speicher benötigt wird. Außerdem wird am Beispiel des Open Source Datenstrukturspeichers *Redis* und der SQL Abfrage-Engine *Presto* gezeigt, wie HyperLogLog in realen Anwendungen implementiert werden kann.

Abstract

This seminar paper presents the algorithm HyperLogLog for solving the count-distinct problem, in which the cardinality of a data set, i. e. the number of distinguishable elements of that set, is to be determined. HyperLogLog can determine cardinality without having to keep the entire set in memory by randomizing the input values using a hash function and then checking for specific bit patterns. The algorithm returns a data structure called HyperLogLog Sketch, which is characterized by its low space requirement and simple additivity. This work shows that the HyperLogLog algorithm can determine cardinalities over 10^9 with an accuracy of 2% and requires only $1.5kB$ of additional memory. In addition, the open source, in-memory datastore Redis and the SQL query engine Presto demonstrate how HyperLogLog can be implemented in real applications.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

Laut einer Prognose von Statista, soll im Jahr 2025 der Umfang der weltweit erzeugten, erfassten, kopierten und verbrauchten Daten 180 Zettabytes - also eine Milliarde Terrabytes - betragen [Sta]. Um eine solche Menge an Daten verarbeiten zu können, werden Jahr für Jahr innovative Algorithmen und Datenstrukturen im Bereich von Big-Data Applikationen entwickelt. Eine Problemstellung die dabei immer wieder auftritt, ist die Bestimmung der Kardinalität einer Datenmenge, also die Berechnung der Anzahl der unterscheidbaren Elemente. Dieses Problem wird auch *Count-Distinct Problem* genannt.

Der triviale Ansatz, um dieses Problem zu lösen, ist alle Elemente der Menge zu durchlaufen und eine Liste aller bereits gefundenen Elemente zu speichern. Mit Hilfe von Strukturen wie Hashtabellen oder Suchbäumen, kann das Hinzufügen von Elementen bzw. die Überprüfung ob ein bestimmtes Element bereits gefunden wurde effizient durchgeführt werden [JL14]. Diese Vorgehensweise hat den Vorteil, dass die Kardinalität exakt bestimmt wird - in vielen Anwendungen kann die Platzkomplexität jedoch zum limitierenden Faktor werden [FM85]. Soll beispielsweise die Anzahl der Nutzer bestimmt werden, die in der vergangenen Woche das soziale Netzwerk Facebook besucht haben, würde die Berechnung mit dieser Methode Tage in Anspruch nehmen und Terabytes an Speicher verbrauchen [HT18].

Ein Ansatz zur Lösung dieses Problems ist der Algorithmus HyperLogLog , der in dieser Seminararbeit vorgestellt und analysiert wird. HyperLogLog kann die Kardinalität einer Datenmenge bestimmen, ohne dass dabei die gesamte Menge im Speicher gehalten werden muss. Um die Arbeitsweise von HyperLogLog zu verstehen, werden in Kapitel 2 die theoretischen Grundlagen bereitgestellt und die zugrundeliegende Idee von LogLog vorgestellt. Aus diesen Erkenntnissen wird der Algorithmus HyperLogLog in Kapitel 3 hergeleitet und analysiert. Dabei werden Aussagen über Platz- und Zeitkomplexität getroffen und Optimierungsstrategien aufgezeigt. Anschließend wird in Kapitel 4 auf Anwendungen von HyperLogLog eingangen und Implementierungen werden am Beispiel des In-Memory Datenstrukturspeichers *Redis* und der SQL Abfrage-Engine *Presto* vorgestellt.

2

Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen für den Algorithmus HyperLogLog beschrieben. Dabei wird zuerst die zugrundeliegende Problemstellung in Form des *Count-Distinct Problems* spezifiziert und ein trivialer Lösungsansatz aufgezeigt. Anschließend wird das Prinzip *Probabilistic Counting* vorgestellt.

2.1 Das Count-Distinc-Problem

Das *Count-Distinct Problem* befasst sich mit der Frage, die Kardinalität einer Datenmenge, also die Anzahl der unterscheidbaren Elemente, zu bestimmen. Die Problemstellung dieser Seminararbeit kann also folgendermaßen spezifiziert werden [DF03]:

Problemstellung: Count-Distinct

Sei $\mathbb{G} = 0, 1^*$ die Grundmenge mit einheitlicher Wahrscheinlichkeitsverteilung. Eine ideale Multimenge $\mathbb{M}$ mit der Kardinalität n wird erzeugt indem

1. eine Sequenz der Länge n aus zufälligen Elementen aus $\mathbb{G}$ gebildet wird,
2. Elemente auf beliebige Weise dupliziert werden,
3. eine zufällige Permutation der Elemente gewählt wird.

Das Ziel eines Algorithmus ist es bei gegebener Multimenge $\mathbb{M}$ (die im Normalfall sehr groß ist), den (ihm unbekannten) Wert von n mit möglichst geringer Zeit- und Platzkomplexität zu berechnen. Wichtig ist dabei, dass der Algorithmus vollkommen unabhängig von der Anzahl der Replikationen und der Anordnung der Elemente ist, über die keine Informationen vorliegen.

2.2 Count Distinct in SQL

Eine triviale Lösung für das Count-Distinct Problem bietet die *COUNT*-Funktion in der Datenbanksprache SQL, die die Anzahl der Zeilen einer Datenbankabfrage zurückgibt. Dabei kann das Schlüsselwort *DISTINCT* eingesetzt werden, um explizit zu fordern, dass mehrfach auftretene Werte nur einmal gezählt werden [Kle11]. Soll etwa das in der Einleitung vorgestellte Beispiel fortgeführt werden und die Anzahl der Benutzer eines sozialen Netzwerks am 01.01.2023 bestimmt werden, könnte die SQL-Abfrage folgendermaßen aussehen:

COUNT DISTINCT

SELECT COUNT (***DISTINCT*** user_id) ***AS*** distinct_users
FROM users
WHERE date='2023-01-01'

Listing 2.1: SQL-Abfrage angelehnt an [Kle11].

2.3 Probabilistic Counting

In vielen Anwendungsbereichen bei denen mit großen Datenmengen gearbeitet wird, genügt eine Schätzung der Kardinalität, sofern diese nur eine angemessene Abweichung vom wahren Wert aufweist. Aus diesem Grund wurde 1985 von Philippe Flajolet und Nigel G. Martin in [FM85] das *Probabilistic Counting* vorgestellt, bei dem die Kardinalität einer Multimenge mit Hilfe von Bit-Mustern geschätzt wird.

Wir nehmen an, dass eine Hashfunktion h vorliegt, die Elemente aus der Grundmenge $\mathbb{G}$ in Binärstrings umwandelt, sodass die Bits des Hashwertes zufälligen, unabhängigen Bits entsprechen [DF03]. Auch wenn es in der Theorie nicht möglich ist, eine Hashfunktion zu definieren, die zufällige Daten aus nicht-zufälligen Daten erzeugt, kann in der Praxis leicht eine Hashfunktion gefunden werden, die in solches Verhalten mit ausreichender Genauigkeit imitiert [Knu98]. Mit Hilfe dieser Hashfunktion h, werden die Elemente $m \in \mathbb{M}$ sukzessive in Binärstrings b umgewandelt und anschließend die Position i des Most Significant 1-Bit in b bestimmt. Wurde die Position i bestimmt, wird dies in einer Bitmap B durch $B[i] = 1$ festgehalten. Am Ende dieses Vorgangs gilt $B[k] = 1$ sofern unter den gehashten Elementen aus $\mathbb{M}$ ein Wert auftritt, der mit einem Bitmuster der Form $0^k1...$ beginnt [FM85]. Unter der Annahme, dass die Hashwerte $h(x)$ gleichmäßig verteilt sind, tritt das Bitmuster $0^k1...$ mit einer Wahrscheinlichkeit von 2^{-k-1} auf. $B[i]$ wird also sehr wahrscheinlich $Null$ sein, wenn $i \gg log_2 n$ und $Eins$ sein, wenn $i \ll log_2 n$. Aus diesem Grund ist der Index I der ersten $Null$ in B (von $B[0]$ aus gezählt) ein Indikator für $log_2 n$ und kann somit zur Abschätzung der Kardinalität von $\mathbb{M}$ verwendet werden [FM85]. Flajolet und Martin konnten zeigen, dass für den Erwartungswert des Index I

$$E(I) \approx log_2(\phi \cdot n) \tag{2.1}$$

mit dem Korrekturfaktor $\phi = 0,77351$ gilt [FM85]. Dabei beläuft sich die Standardabweichung von I auf ungefähr

$$\sigma(I) \approx 1,12 \tag{2.2}$$

sodass der geschätzte Wert für die Kardinalität im Durchschnitt bis zu einer binäre Größenordnung vom wahren Wert entfernt ist [FM85].

3

Der Algorithmus HyperLogLog

In diesem Kapitel wird beschrieben, wie auf Basis der im Abschnitt *Probabilistic Counting* beschriebenen Konzepte, ein Algorithmus zur Schätzung der Kardinalität einer Multimenge $\mathbb{M}$ über einer diskreten Grundmenge $\mathbb{G}$ erstellt werden kann.

3.1 Grundidee LogLog

Ein Algorithmus zur Lösung der in Kapitel 2 definierten *Problemstellung* ist der im Jahr 2003 von Durand und Flajolet vorgestellte LogLog Algorithmus [DF03]. Hierbei handelt es sich um einen probabilistischen Algorithmus, der die Anzahl der distinkten Elemente einer beliebigen Multimenge $\mathbb{M}$ ausgibt. Die Multimenge wird mit Hilfe einer Hashfunktion randomisert, sodass die Elemente als einheitliche, unabhänge Binärdaten vorliegen. Wie beim *Probabilistic Counting* besteht die grundlegende Idee des Algorithmus darin, Bitmuster der Form 0^*1 am Anfang der gehashten Werte zu finden. Dazu wird die Funktion $p(x)$ definiert, die für einen Binärstring $x \in \{0,1\}^\infty$ die Position des ersten 1-Bit bestimmt, sodass $p(1...) = 1$, $p(001...) = 3$ usw. gilt . Es ist zu erwarten, dass für $n/2^k$ Elemente aus $\mathbb{M}$ $p(x) = k$ gilt, wobei n die Kardinalität von $\mathbb{M}$ ist. Der Parameter

$$R(\mathbb{M}) = \max_{x \in \mathbb{M}} p(x) \tag{3.1}$$

stellt also einen Schätzwert für $\log_2 n$ dar.[1]

Ein Problem bei diesem Vorgehen ist die hohe Variabilität der Ergebnisse, da bereits *ein* Ausreißer das Ergebnis stark verfälschen kann. Daher wäre es naheliegend, die Kardinalitätsschätzung mehrfach mit verschiedenen Hashfunktionen durchzuführen und den Mittelwert der Ergebnisse zu bilden. Für dieses Vorgehen würden jedoch eine große Mengen an unabhängigen Hashfunktionen benötigt werden und die Laufzeit würde signifikant ansteigen, da für jedes Element mehrere Hashwerte bestimmt werden müssten [FFGM07]. Dieses Problem wird im LogLog Algorithmus adressiert, indem die Elemente der Eingabemenge in $m = 2^k$ Gruppen, die sogenannten "*Buckets*" eingeteilt werden, was einfach umgesetzt werden kann, indem die ersten k Bits des gehashten Wertes jedes Elements den binärer Index eines Buckets repräsentieren.

[1] Die Herleitung dieses Zusammenhangs wurde im Kapitel Probabilistic Counting erklärt.

Anschließend wird für jeden Bucket der Parameter R nach Formel Formel 3.2 bestimmt und der arithmetische Mittelwert $\frac{1}{m} \cdot \sum_{j=1}^{m} R^{(j)}$ berechnet, der den Wert $log_2(n/m)$ approximiert.[2] Der Schätzwert für die Kardinalität n, den der Algorithmus LogLog zurückgibt, ergibt sich dementsprechend zu

$$E = \alpha_m m \cdot 2^{\frac{1}{m} \cdot \sum R^{(j)}} \tag{3.2}$$

wobei α_m einen Korrekturfaktor darstellt, um den systematischen Fehler bei der Schätzung auszugleichen. Durand und Flajolet konnten in [DF03] zeigen, dass sich der Standardfehler des Algorithmus abhängig von der Anzahl der Buckets zu

$$Standardfehler \approx \frac{1,30}{\sqrt{m}} \tag{3.3}$$

berechnet, was für $m = 256$ und $m = 1024$ einen Standardfehler von 8% bzw. 4% ergibt.

3.2 HyperLogLog

Der in Kapitel 3.1 vorgestellte Algorithmus kann durch kleinere Anpassungen weiter optimiert werden. Durand und Flajolet konnten feststellen, dass einzelne Ausreißer nach oben in den Buckets den Schätzwert stark verzerren können, da bei der Berechnung im Algorithmus mit dem Mittelwert der Buckets exponentiert wird [DF03]. Um dieser Tatsache entgegenzuwirken, kann die sog. Trunkierungsregel angewendet werden, bei der nur die $m_0 = \lfloor \sigma m \rfloor$ kleinsten Werte der Buckets betrachtet werden. Für $\sigma = 0,7$ produziert der Algorithmus Ergebnisse mit einem signifikant verbesserten Standardfehler von $\frac{1,05}{\sqrt{m}}$.[3] Dieses Vorgehen hat jedoch den Nachteil, dass sich der Algorithmus nicht mehr ohne weiteres hinsichtlich des Standardfehlers und des Biases analysieren lässt. Daher wird beim HyperLogLog Algorithmus statt der Trunkierungsregel der Evaluierungsparameter R mithilfe des harmonischen Mittelwerts bestimmt um den Einfluss von Ausreißer zu reduzieren [FFGM07]:

$$x_{harmonisch} = \frac{n}{\frac{1}{x_1} + ... + \frac{1}{x_n}} \tag{3.4}$$

Da bei der Bestimmung des harmonischen Mittelwertes mit dem Kehrwert der Einzelwerte gerechnet wird, fallen starke Ausreißer nach oben nur sehr wenig ins Gewicht. Mit Hilfe dieser Erkenntnisse kann der HyperLogLog Algorithmus wiefolgt definiert werden:

[2] Dieses Vorgehen wird auch Stochastic Averaging genannt.
[3] Ein Algorithmus mit dieser Erweiterung wurde in [FFGM07] unter dem Namen SuperLogLog vorgestellt

HyperLogLog

Sei $\boldsymbol{p(x)}$ *eine Funktion, die für einen Binärstring* $x \in \{0,1\}^\infty$ *die Position des ersten 1-Bit bestimmt*
Sei $\boldsymbol{h(x)}$ *eine Hashfunktion, die Elemente aus* $x \in \mathbb{M}$ *in Binaerstrings umwandelt*

Algorithmus HyperLogLog($\mathbb{M}, m \equiv 2^k$):

Initialisiere m Buckets $M[1], ..., M[m]$ mit $-\infty$

for $v \in \mathbb{M}$ **do**:

// Hashwert eines Elements aus $\mathbb{M}$
$x = h(v)$

// Binaere Adresse, die durch die ersten k Bits von x festegelegt wird
$j = 1 + < x_1 x_2 ... x_b >_2$

// Hashwerte eines Elements ohne die ersten k Adress-Bits
$w = x_{b+1} x_{b+2}$

// Bucket aktualisieren, wenn neuer Hoechstwert gefunden wurde
$M[j] = max(M[j], p(w))$

// Berechne harmonischen Mittelwert Z
$Z = m \cdot (\sum_{i=1}^{m} 2^{-M[j]})^{-1}$

return $E = \alpha_m m Z$

Listing 3.1: Algorithmus HyperLogLog angelehnt an [FFGM07].

Die Eingabe ist eine Multimenge $\mathbb{M}$, die als Datenstrom repräsentiert wird, dessen Elemente sequentiell gelesen werden und die Anzahl der zu verwendenden Buckets m. Wie beim LogLog Algorithmus werden die Hashwerte der Elemente des Datenstroms basierend auf den ersten b-Bits in $m = 2^b$ Buckets aufgeteilt und mit Hilfe der Funktion $p(x)$ die maxmiale Position des ersten 1-Bit aller Elemente eines Buckets bestimmt. Diese Werte werden verwendet, um die Indikatorfunktion Z zu bestimmen. Der harmonische Mittelwert der $2^{R^{(j)}}$-Werte (mZ) in der Größenordnung n/m liegt, stellt $m^2 Z$ einen Schätzwert für n dar, der multipliziert mit dem Korrekturfaktor α_m vom Algorithmus HyperLogLog zurückgegeben wird.

$$E = \alpha_m m^2 Z = \frac{\alpha_m m^2}{\sum_{j=1}^{m} 2^{-R^{(j)}}} \tag{3.5}$$

Auf diese Weise wird mit dem HyperLogLog Algorithmus die Kardinalität des Eingabestroms mit einem Standardfehler von $1,04/\sqrt{m}$ berechnet.

3.3 Analyse und Vergleich zu Count-Distinct

Der Algorithmus HyperLogLog unterscheidet sich elementar von dem konventionellen Count Distinct. Besonders bei Big-Data Anwendungen werden drei fundamentale Vorteile des HyperLogLog Algorithmus deutlich:

1. Speicherplatzbedarf

Flajolet und Durand konnten in [DF03] zeigen, dass der der LogLog Algorithmus lediglich $O(loglog\ n)$-Bits an Speicher benötigt um Kardinalitäten bis n zu berechnen (diese Tatsache ist namensgebend für den LogLog- sowie den HyperLogLog Algorithmus). Der benötigte Speicher entspricht im wesentlich dem Speicherbedarf der m Buckets. Da in diesen Buckets lediglich der maximale $p(x)$-Wert bereitgehalten wird und dieser bei L-Bit Hashfunktionen im Intervall $[0, L + 1 - log_2 m]$ liegt, reichen 5 Bits ('Short Bytes') aus, um die Bucketwerte unter Verwendung einer 32-Bit Hashfunktion zu speichern - der zusätzliche Speicherbedarf beläuft sich also auf m-Short Bytes [FFGM07].
Mit $m = 2048$ Buckets, einer 32-Bit Hashfunktion können also Kardinalitäten über $N = 10^9$ mit einer Genauigkeit von 2% bestimmt werden, wobei nur $1,5kB$ an zusätzlichem Speicher benötigt wird [FFGM07].

2. Additivität

Ein weiterer Vorteil ist, dass *HLL*-Datenstrukturen im Gegensatz zum Count Distinct additiv sind. Ein gutes Beispiel dafür ist das Einführungsbeispiel bei dem die Anzahl der Benutzer eines sozialen Netzwerks mit Hilfe der SQL-Funktion *COUNT DISTINCT* bestimmt wurden. Dieses Ergebnis ist nicht additiv. Wird in einer anderen SQL-Abfrage die Anzahl der Benutzer am Folgetag bestimmt, lässt sich keinerlei Aussage über die Anzahl der distinkten Benutzer an beiden Tagen machen. Zwischenergebnisse aus *HLL*-Datenstrukturen können dahingehend auf einfach Weise addiert werden, indem der maximale Wert von korrespondierenden Buckets in die vereinigte Datenstruktur übernommen wird [HT18].

3. Einfache Parallelisierbarkeit

Aus der Additivität der *HLL*-Datenstrukturen folgt, dass sich die Berechnung von Kardinalitäten mit dem HyperLogLog Algorithmus einfach auf unabhängigen Rechnern parallelisieren lässt. Dazu muss lediglich der Eingangsdatenstrom auf die Rechner aufgeteilt werden und anschließend die berechneten *HLL*-Datenstrukturen vereinigt werden [HT18].

3.4 Optimierungsstrategien

Bereits in der ersten Veröffentlichung des HyperLogLog Algorithmus in [FFGM07] wurden Möglichkeiten zur Optimierung vorgestellt. Folgende Anpassungen wurden vorgenommen, die den Algorithmus HLL_{opt}

1. Initialisierung der Register

Der in Listing 3.1 vorgestellte Algorithmus initialisiert die m-Buckets mit $-\infty$. Das hat den Vorteil, dass die Analyse des Algorithmus wesentlich einfacherer ausfällt und zu übersichtlicheren Ausdrücken führt. Dennoch führt dies dazu, dass für den Schätzwert E der Kardinalität immer 0 zurückgegeben wird, sobald einer der Buckets unberührt bleibt. Dies bedeutet, dass $E = 0$ erwartet wird, sobald $n \ll mlog_m$. Aus diesem Grund werden die Buckets in HLL_{opt} mit 0 initialisiert.[4] Dadurch können brauchbare Schätzwerte berechnet werden, auch wenn n ein kleines Vielfaches von m ist.

2. Korrektur für kleine Kardinalitäten

Ausführliche Simulationen haben ergeben, dass für Kardinalitäten $\leq \frac{5}{2}m$ nichtlineare Verzerrungen auftreten, die den von HLL_{opt} berechneten Schätzwert E stark verfälschen. Daher wird für Kardinalitäten $\leq \frac{5}{2}m$ der in [WVZT90] vorgestellte Algorithmus Hit Counting verwendet.

3. Korrektur für große Kardinalitäten

Für Kardinalitäten bis zur Größenordnung 10^9 sollten mindestens $L = 32$-Bit Hashfunktionen verwendet werden. Wenn sich die Kardinalität jedoch $2L$ annähert (oder diesen Wert sogar überschreitet), werden Hashing-Kollisionen immer wahrscheinlicher. In diesem Fall wird in HLL_{opt} der Schätzwert korrigiert zu:[5]

$$E^* = -2^L \cdot log(1 - E/2^L) \tag{3.6}$$

Weitere Optimierungsstrategien wurden von Mitarbeitern der Google LLC in [HNH13] vorgestellt und unter dem Namen HyperLogLog_{++} zusammengefasst. Bei dem Algorithmus HyperLogLog_{++} wird im Gegensatz zu HyperLogLog mit einer 64-Bit Hashfunktion gearbeitet, da auf diese Weise Hashkollisionen wesentlich unwahrscheinliche werden und somit Kardinalitäten weiter über 10^9 geschätzt werden können. Des Weiteren wurde die Schätzung von kleinen Kardinalitäten verbessert, indem neue Schätzverfahren entwickelt wurden und eine Datenrepräsentation gewählt wurde, die auf die Speicherung von 0-Bits optimiert ist.

[4] Ein Beweis, dass die Analyse des Algorithmus auch mit dieser Änderung noch gültig ist, kann in [FFGM07] nachgelesen werden.

[5] Der korrigierte Wert wird hergeleitet aus Wahrscheinlichkeits-Betrachtungen in [WVZT90]: Werden n Bälle in m Körbe geworfen und man betrachtet die leeren Körbe V, ist der Erwartungswert für $E(n/m) = mlog(m/V)$

4

Anwendung von HyperLogLog

Auf Grund der Relevanz des Count-Distinct Problems ist der Algorithmus HyperLogLog, bzw. eine äquivalente HLL-Datenstruktur in vielen Datenbanksystemen vorhanden. In diesem Kapitel werden die Implementierungen im Datenstrukturspeicher *Redis* und in der SQL Abfrage-Engine *Presto* vorgestellt und ein beispielhafter Funktionsablauf dargestellt.

4.1 Redis

Redis ist ein Open Source Datenstrukturspeicher, der als Datenbank, Cache und Streaming-Engine verwendet werden. Weitere Informationen können in der offiziellen Dokumentation [Red] nachgelesen werden. *Redis* stellt eine Datenstruktur *HyperLogLog* mit einer festen Anzahl an Buckets zur Verfügung, die bis zu $12kB$ an Speicher verwendet und einen Standardfehler von $0,81\%$ aufweist. Auf die Datenstruktur können drei Basisoperationen angewendet werden:

1. ***PFADD***: fügt ein Element zu einem *HyperLogLog* Sketch hinzu. Laufzeit: $O(1)$
2. ***PFCOUNT***: gibt einen Schätzwert für die Kardinalität der *HyperLogLog* Sketch zurück. Laufzeit: $O(1)$
3. ***PFMERGE***: kombiniert zwei oder mehr *HyperLogLog* Sketches. Laufzeit: $O(k)$ wobei k die Anzahl der *HyperLogLog* Sketche ist.

4.2 Facebook: Presto

Presto ist eine SQL Abfrage-Engine, die 2012 von Facebook entwicklet wurde und 2013 als Open Source Software zur Verfügung gestellt wurde. Presto implementiert die Datenstruktur HyperLogLog als 32-Bit Buckets und stellt die folgenden Funktionen zur Verfügung, die denen von Redis stark ähneln [Pre]:

1. ***approx_set(x)***: gibt den *HyperLogLog* Sketch der Eingabemenge x zurück.
2. ***approx_set(x,e)***: gibt den *HyperLogLog* Sketch der Eingabemenge x mit maximalem Standardfehler e zurück.
3. ***cardinality(hll)***: gibt die Kardinalität des *HyperLogLog* Sketches *hll* als Big-Integer zurück.
4. ***merge_hll([hll])***: kombiniert die *HyperLogLog* Sketches die als Liste an die Funktion übergeben werden.

5

Zusammenfassung und Ausblick

In dieser Seminararbeit wurde eine vollständige Herleitung, Beschreibung und Analyse des Algorithmus HyperLogLog präsentiert. Dazu wurden die theoretischen Grundlagen in Form des Probabilistic Counting und die Vorgängerversionen von HyperLogLog vorgestellt, sodass der Leser ein tiefgehendes Verständnis für Algorithmen zur Kardinalitätsschätzung aufbauen konnte. Es wurden die Stärken und Limitationen des Algorithmus herausgearbeitet und zudem erste Optimierungsstrategien vorgestellt. An den Beispielen Redis und Presto wurde demonstriert, wie HyperLogLog in verschiedenen Datenbanksystemen implementiert wurde und mit welchen Funktionen auf HLL-Strukturen gearbeitet werden kann.

Zusammenfassend lässt sich sagen, dass HyperLogLog ein effizientes Verfahren zur Schätzung von Kardinalitäten von Multimengen darstellt. Sofern eine gewisse Abweichung vom tatsächlichen Wert erlaubt ist, ermöglicht HyperLogLog Bestimmung von Kardinalitäten, die mit herkömmlichen Verfahren wie dem COUNT DISTINCT Tage zur Berechnung benötigen würden. Besonders der stark verringerte zusätzliche Speicherplatzbedarf macht den Algorithmus sehr attraktiv für Big-Data Anwendungen. Die Additivität der Datenstruktur führt dazu, dass sich HyperLogLog Sketches einfach mit neuen Informationen erweitern lassen und dass sich Kardinalitätsberechnungen ohne Weiteres parallelisieren lassen, wodruch sich der Algorithmus ideal für die Verarbeitung von großen Datenmengen eignet. Jedes Jahr steigt die Menge der weltweit erfassten Daten und der Algorithmus HyperLogLog trägt seinen Teil dazu bei, den wachsenden Bedarf an skalierbaren und effizienten Lösungen zu decken.

Literaturverzeichnis

DF03. Durand, Marianne und Philippe Flajolet: *Loglog Counting of Large Cardinalities*. In: *Algorithms - ESA 2003*, Seiten 605–617. Springer Berlin Heidelberg, 2003.

FFGM07. Flajolet, Philippe, Éric Fusy, Olivier Gandouet und Frédéric Meunier: *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*. Discrete Mathematics and Theoretical Computer Science, DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), jan 2007.

FM85. Flajolet, Philippe und G. Nigel Martin: *Probabilistic counting algorithms for data base applications*. Journal of Computer and System Sciences, 31(2):182–209, oct 1985.

HNH13. Heule, Stefan, Marc Nunkesser und Alexander Hall: *HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm*. In: *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, mar 2013.

HT18. Honarkhah, M. und A. Talebzadeh: *HyperLogLog in Presto: A significantly faster way to handle cardinality estimation*, Dezember 2018. Abgerufen am 18.01.2022 von `https://engineering.fb.com/2018/12/13/data-infrastructure/hyperloglog/`.

JL14. J. Leskovec, A. Rajaraman, J.D. Ullman: *Mining Data Streams*. In: *Mining of Massive Datasets*, Seiten 123–153. Cambridge University Press, nov 2014.

Kle11. Kleuker, Stephan: *Grundkurs Datenbankentwicklung - von der Anforderungsanalyse zur komplexen Datenbankanfrage*. Vieweg+Teubner Verlag / Springer Fachmedien Wiesbaden, Wiesbaden, 2011.

Knu98. Knuth, Donald E.: *The Art of Computer Programming*. Addison Wesley, Second Auflage, 1998.

Pre. Presto: *Presto Documentation*. Abgerufen am 18.01.2022 von `https://prestodb.io/docs/current/`.

Red. Redis: *Redis Documentation*. Abgerufen am 18.01.2023 von `https://redis.io/docs/`.

Sta. Statista: *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to*

2025. Abgerufen am 27.01.2022 von `https://www.statista.com/statistics/871513/worldwide-data-created/`.

WVZT90. Whang, Kyu-Young, Brad T. Vander-Zanden und Howard M. Taylor: *A linear-time probabilistic counting algorithm for database applications*. ACM Transactions on Database Systems, 15(2):208–229, jun 1990.

A

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Seminararbeit ohne fremde Hilfe verfasst und nur die im Literaturverzeichnis angegebenen Quellen verwendet habe.

Datum	Unterschrift der Kandidatin/des Kandidaten