Projektseminar

Experimentelle Untersuchung des “Split-and-Share”-Verfahrens

Chris Köcher Philipp Schlag

Wintersemester 2015/2016

Letzte Änderung: 29. März 2016

TECHNISCHE UNIVERSITÄT
ILMENAU

Inhaltsverzeichnis

Kapitel I
Konstruktionsverfahren

In dieser Arbeit untersuchen wir das "Split-and-Share"-Verfahren als mögliche Realisierung perfekter Hashfunktionen experimentell. Zunächst möchten wir in diesem Kapitel die von uns verwendeten Algorithmen zur Konstruktion dieser Hashfunktionen beschreiben. Im Allgemeinen basieren diese auf den in [Die07] und [BPZ13] dargestellten Methoden und theoretischen Betrachtungen.

1.1 Überblick

Für eine gegebene Menge $S \subseteq U = \{0,1\}^{64}$ von 64-Bit-Schlüsseln konstruieren wir unter Verwendung des "Split-and-Share"-Verfahrens eine perfekte Hashfunktion in Form einer Datenstruktur `PerfectHashFunction`, die anschließend genutzt werden kann, um für jeden Schlüssel $x \in S$ mittels der Methode `evaluate` den eineindeutigen Hashwert $h_{\text{perf}}(x)$ zu berechnen.

Das in der Programmiersprache C++ implementierte Kommandozeilen-Programm für Linux `Hashing` (für den Quellcode siehe beigefügte CD oder https://github.com/ckoecher/Hashing) wird dabei in der Form

```
./Hashing <configuration file> <data file> <statistics file>
```

aufgerufen. In der Konfigurationsdatei können diverse Parameter zur Erzeugung der Hashfunktion festgelegt werden. Dadurch kann z.B. die Anzahl der erzeugten Buckets variiert oder der Startwert des Zufallszahlengenerators gesetzt werden. Aus der Datendatei werden die Schlüssel als 64-Bit-Einheiten im Binärmodus gelesen. Zum Schluss werden statistische Kenngrößen der Konstruktion, z.B. die benötigten Zeiten der einzelnen Konstruktionsphasen, der Speicherplatz der erzeugten Datenstruktur und die Anzahl wiederholter Versuche, in der Statistikdatei gespeichert.

Im Folgenden verwenden wir Standardwerte, wie sie in [Die07] und [BPZ13] beschrieben werden.

1.2 Erzeugung von Zufallszahlen

Als Zufallszahlengenerator zur Erzeugung der notwendigen (Pseudo-)Zufallszahlen, natürliche Zahlen innerhalb variierender Wertebereiche, verwenden wir die in der C++-Standardbibliothek (seit C++11) enthaltene Implementierung `mt19937` des Mersenne-Twisters.

1.3 1-universelle Hashfunktionen

Unsere elementaren Grundbausteine zur Konstruktion der perfekten Hashfunktion sind 1-universelle Hashfunktionen $h\colon U \to [m'] = \{0, 1, \ldots, m' - 1\}$. Dabei wird jeder Schlüssel $x \in S$ in l Teilschlüssel bestehend aus k Bit aufgeteilt, d.h. $x = x_0 \ldots x_{l-1}$ mit $x_0, \ldots, x_{l-1} \in \{0,1\}^k$.

Um eine solche 1-universelle Hashfunktionen zufällig zu wählen, werden $l+1$ Koeffizienten $a_0, a_1, \ldots, a_l$ aus $\left[2^{k+\lceil \log m' \rceil + 6}\right]$ zufällig gewählt. Der Hashwert berechnet sich dann wie folgt:

$$h(x) = h(x_0 \ldots x_{l-1}) = \underbrace{\left(\left(\left(a_l + \sum_{i=0}^{l-1} a_i x_i\right) \bmod 2^{k+\lceil \log m' \rceil + 6}\right) \operatorname{div} 2^k\right)}_{=h'(x)} \bmod m'.$$

Da der innere Teil der Funktion (alles bis auf $\bmod m'$) eine 2-fach unabhängige Hashfunktion $h'\colon U \to [2^{\lceil \log m' \rceil + 6}] \approx [m' \cdot 2^6]$ ist, ist h selbst (annähernd) 1-universell.

1.4 Split in kleine Buckets

Zur Wahl einer geeigneten Anzahl von Buckets m, die von der zufällig gewählten Split-Funktion erzeugt werden, haben wir zwei mögliche Verfahren implementiert, welche im Folgenden kurz erläutert werden sollen.

1.4.1 Parametrisiertes Verfahren

Zunächst werden die $n = |S|$ 64-Bit-Schlüssel in $m = \min\left\{\lceil 2n^{2/3} \rceil, \lceil \frac{n}{20} \rceil\right\}$ Buckets aufgeteilt. Sowohl der Koeffizient als auch der Exponent zur Bestimmung der Anzahl von Buckets sind dabei durch Konfigurationsparameter variierbar. (In diesen beiden liegt auch besonders viel Potenzial für eine Verbesserung in der Konstruktion bzgl. Konstruktionsdauer und Größe der erzeugten Datenstruktur.) Die zusätzliche Einschränkung, dass höchstens $\lceil \frac{n}{20} \rceil$ Buckets erzeugt werden, soll für kleine n die Wahrscheinlichkeit verringern, dass zu viele zu kleine Buckets entstehen.

In der Split-Phase werden sämtliche Schlüssel aus der Datendatei mehrfach eingelesen. Zunächst wird für die zufällig gewählte 1-universelle Hashfunktion $h_{\text{split}}\colon U \to [m]$ getestet, ob alle m Buckets höchstens $\max\{\lfloor\sqrt{n}\rfloor, 40\}$ Schlüssel enthalten, d.h. nicht zu groß werden (Counting). Für $i \in [m]$ bezeichne n_i die Anzahl der Schlüssel, die in Bucket i enthalten sind. Dann ist $\hat{n} = \max\{n_i \mid i \in [m]\}$ die maximale Bucketgröße und nach [Die07, Lemma 1] gilt $\Pr(\hat{n} \leq \sqrt{n}) \geq \frac{3}{4}$. Im positiven Fall werden die Schlüssel noch einmal eingelesen und in einer temporären Datei derart gespeichert, dass alle Schlüssel desselben Buckets direkt hintereinander in der Datei stehen (Splitting)[1].

Im weiteren Verlauf wird für jeden Bucket i eine perfekte Hashfunktion mit Wertebereich $[m_i]$ konstruiert, wobei $m_i = \lceil 1{,}25 \cdot n_i \rceil$ ist. (Auch der Koeffizient $1{,}25$ kann in der Konfigurationsdatei variiert werden.) Allerdings muss zusätzlich sichergestellt werden (ggf. durch einfache Vergrößerung), dass $m_i \geq 10$ gilt, falls $m_i > 0$ ist. Das Konstruktionsverfahren alleine benötigt $m_i \geq 3$, da als Zwischenschritt für jeden Schlüssel des Buckets i drei verschiedene Hashwerte in $[m_i]$ berechnet werden müssen. Um weitere Schwierigkeiten bei der Erzeugung der perfekten Hashfunktion für Bucket i mit besonders wenigen Schlüsseln (in Tests war dies besonders für $2 \leq n_i \leq 5$ der Fall, weil dann m_i mit $3 \leq m_i \leq 7$ zu klein ist) zu vermeiden, hat sich $m_i \geq 10$ als ausreichender Wert herausgestellt.

1.4.2 Automatisiertes Verfahren

Werden die Parameter derart gewählt, dass für die Anzahl zu erzeugender Buckets $m = 0$ gelten würde, so wird im Programm eine Anzahl von Buckets berechnet, die den Gesamtspeicherplatz der am Ende erzeugten Datenstruktur zu minimieren versucht. Dieser ist im Wesentlichen von der Anzahl der Buckets m und der maximalen Bucketgröße $\hat{n}$ abhängig. Basierend auf [RS98, Theorem 1] und experimenteller Untersuchungen, die in Abschnitt 2.4 erläutert werden, hat sich gezeigt, dass

$$\frac{n}{m} + \sqrt{\frac{2 \cdot n \cdot \log m}{m}}$$

eine sinnvolle Abschätzung für $\hat{n}$ darstellt. Mit dieser ist es möglich, diejenige Anzahl von Buckets m zu bestimmen, sodass der Gesamtspeicherplatz (unter der Annahme, dass $\hat{n}$ tatsächlich den Wert der Abschätzung annimmt) minimiert wird.

1.5 Geteilte Datenstruktur

Um für jeden einzelnen Bucket eine perfekte Hashfunktion zu konstruieren, werden insbesondere sechs Tabellen $T_0^0, T_0^1, T_0^2, T_1^0, T_1^1, T_1^2$ gemeinsam genutzt. Jede Tabel-

[1] Wir haben uns für diese zeitintensivere Variante entschieden, damit auch für größere Schlüsselmengen das Konstruktionsverfahren noch auf einem gewöhnlichen PC ohne übermäßig viel Arbeitsspeicher ausgeführt werden kann.

le enthält $r = \lceil 2\hat{n}^{3/2} \rceil$ Einträge zufälliger Bitzahlen der Länge $b = \lceil \log \hat{n} \rceil + 6$. Um die Versagenswahrscheinlichkeit weiter zu minimieren, wird für jeden Bucket i noch eine weitere zufällige Bitzahl $s_i \in [2^b]$ genutzt. Indem einzelne s_i oder sämtliche Tabellen mit neuen zufälligen Zahlen gefüllt werden, können die folgenden Konstruktionsschritte im Falle eines Versagens wiederholt werden. Die maximale Anzahl solcher Wiederholungen kann in der Konfigurationsdatei angegeben werden.

1.6 Gute Paare 1-universeller Hashfunktionen pro Bucket

Für jeden Bucket $i \in [m]$ mit Schlüsselmenge S_i wird zunächst ein gutes Paar (h_0^i, h_1^i) 1-universeller Hashfunktionen $h_0^i, h_1^i \colon U \to [r]$ zur Ansteuerung der Tabelleneinträge erzeugt. Dabei heißt das Paar (h_0^i, h_1^i) **gut**, wenn für jeden Schlüssel $x \in S_i$ einer der Hashwerte eineindeutig ist, d.h. es gibt ein $j \in \{0, 1\}$ derart, dass $h_j^i(x) \neq h_j^i(y)$ für alle $y \in S_i \setminus \{x\}$ gilt. Nach [Die07, Lemma 2] gilt $\Pr\left((h_0^i, h_1^i) \text{ ist gut}\right) \geq \frac{3}{4}$.

1.7 Uniforme Hashfunktionen pro Bucket

Ist (h_0^i, h_1^i) ein gutes Paar 1-universeller Hashfunktionen, so sind die Funktionen $f_0^i \colon U \to [m_i]$, $f_1^i \colon U \to [m_i - 1]$ und $f_2^i \colon U \to [m_i - 2]$ mit

$$\begin{aligned} f_0^i(x) &= (T_0^0[h_0^i(x)] \cdot s_i \oplus T_1^0[h_1^i(x)]) \bmod m_i \\ f_1^i(x) &= (T_0^1[h_0^i(x)] \cdot s_i \oplus T_1^1[h_1^i(x)]) \bmod (m_i - 1) \\ f_2^i(x) &= (T_0^2[h_0^i(x)] \cdot s_i \oplus T_1^2[h_1^i(x)]) \bmod (m_i - 2) \end{aligned}$$

für $x \in S_i$ rein zufällig auf ihren jeweiligen Wertebereichen. Dabei beschreibt $\oplus$ das bitweise XOR (auf Bitzahlen der Länge b).

Daraus können leicht drei Funktion $g_0^i, g_1^i, g_2^i \colon U \to [m_i]$ konstruiert werden, wobei die Menge $\{g_0^i(x), g_1^i(x), g_2^i(x)\}$ eine rein zufällige dreielementige Teilmenge von $[m_i]$ ist.

1.8 Perfekte Hashfunktionen pro Bucket

Damit können wir für jeden Bucket $i \in [m]$ einen 3-Graphen $G_3^i = (V, E)$ konstruieren mit

$$\begin{aligned} V &= [m_i] \text{ und} \\ E &= \left\{ \{g_0^i(x), g_1^i(x), g_2^i(x)\} \mid x \in S_i \right\}. \end{aligned}$$

Es gilt:

G_3^i ist azyklisch.
$\Leftrightarrow$ Aus G_3^i können durch wiederholtes Löschen von Kanten mit inzidenten Knoten vom Grad 1 alle Kanten entfernt werden.
$\Rightarrow$ Jeder Kante $e \in E$ kann eineindeutig eine inzidente Ecke zugeordnet werden.
$\Leftrightarrow$ Jedem Schlüssel $x \in S_i$ kann eineindeutig ein Hashwert $h \in [m_i]$ zugeordnet werden, wobei $h \in \{g_0^i(x), g_1^i(x), g_2^i(x)\}$.
$\Rightarrow$ Es gibt eine perfekte Hashfunktion $h_i \colon U \to [m_i]$ für S_i, die "leicht" aus G_3^i berechnet werden kann.

Für die Berechnung dieser perfekten Hashfunktion wird wie in [CHM97, Theorem 6.4] beschrieben eine Funktion $g_i \colon [m_i] \to \{0, 1, 2\}$ aus G_3^i konstruiert, mit deren Hilfe schließlich der passende Hashwert $h_i(x) \in \{g_0^i(x), g_1^i(x), g_2^i(x)\}$ für $x \in S_i$ bestimmt werden kann.

Nach [BPZ13, Theorem 3.5] gilt

$$\lim_{m_i \to \infty} \Pr(G_3^i \text{ ist azyklisch}) = 1.$$

Tatsächlich haben unsere Tests mit rein zufälligen Schlüsselmengen untermauert, dass die Wahrscheinlichkeit auch für kleinere m_i zufriedenstellend ist.

1.9 Berechnung des Hashwertes

Sei $x \in S$ beliebig. Dann gehört x zum Bucket $i = h_{\text{split}}(x)$. Der Offset des Wertebereiches $[m_i]$ dieses Buckets im Gesamtwertebereich ist dann

$$\text{offset}_i = \sum_{j=0}^{i-1} m_j.$$

Aus den drei Werten $g_0^i(x)$, $g_1^i(x)$, und $g_2^i(x)$ wird der Index

$$j = (g_i[g_0^i(x)] + g_i[g_1^i(x)] + g_i[g_2^i(x)]) \bmod 3$$

bestimmt. Dann berechnet sich der tatsächliche Hashwert von x wie folgt:

$$h_{\text{perf}}(x) = \text{offset}_i + g_j^i(x).$$

Kapitel II
Untersuchungen und Ergebnisse

2.1 Testumgebung

Die in den weiteren Abschnitten angegebenen Tests wurden allesamt auf einem Rechner mit folgenden Spezifikationen durchgeführt:

- *Betriebssystem:* Linux Mint 17.1 Cinnamon 64-bit (Kernel: 3.13)
- *Prozessor:* Intel® Core™ i7-2600 CPU (3.40GHz) mit 128 kB L1-Cache, 1024 kB L2-Cache, 8192 kB L3-Cache
- *Hauptspeicher:* 11.7 GiB
- *Compiler:* GCC 4.8.4

2.2 Ablauf der experimentellen Untersuchungen

Im Folgenden werden die Ergebnisse zu Testläufen mit Schlüsselmengen der Größe 10^4, 10^5, 10^6, 10^7 und 10^8 präsentiert, wobei wir uns insbesondere für die letzten drei interessieren. Pro Größenordnung wurden fünf verschiedene zufällige Schlüsselmengen mit den in der `config.ini` Datei enthaltenen Parametern untersucht, wobei pro Schlüsselmenge fünf Durchläufe mit jeweils anderem Startwert für den Zufallszahlengenerator durchgeführt wurden. Konkret wurden die Seeds “1”, “42”, “1337”, “123456” und “9892002” verwendet. Damit wurden pro Größenordnung jeweils 25 Testläufe durchgeführt.

Ergänzt werden diese Ergebnisse durch Daten über einen einzelnen Testlauf mit 10^9 Schlüsseln. Da ein solcher Durchlauf (insbesondere aufgrund des hohen Zeitaufwandes für das Lesen aus und das Schreiben in Dateien) bereits mehrere Stunden tatsächliche Zeit (nicht nur CPU-Zeit) benötigt, und wir uns im Rahmen der Experimente nur für “kleine” Schlüsselmengen interessiert haben, soll dies genügen.

Sämtliche erhobenen Daten (sowohl die hier präsentierten als auch weitere) können in der beigefügten Datei `stats.csv` im Detail betrachtet werden. Diese beinhalten zudem Ergebnisse zu noch kleineren Schlüsselmengen.

Es sei angemerkt, dass in allen Testläufen erfolgreich eine perfekte Hashfunktion konstruiert werden konnte.

2.3 Benötigter Speicherplatz

In weiteren Abschnitten wird der benötigte Speicherplatz (in Bits) der Datenstruktur betrachtet. In der `stats.csv` werden hier zwei verschiedene Gruppen von Werten festgehalten:

- der tatsächlich verwendete Speicherplatz der Komponenten und der gesamten Datenstruktur sowie
- der theoretisch benötigte Speicherplatz, wenn wir sämtliche Komponenten durch geeignete Kodierung platz-optimal abgespeichert hätten. (Dies haben wir aus Gründen der besseren Laufzeit und Komplexität des Quellcodes nicht umgesetzt.)

Da die theoretischen Größen weitaus interessanter sind als die Größen, die unsere konkrete Implementierung benötigt, werden wir ab jetzt nur noch diese betrachten.

Die einzelnen Komponenten mit ihren Größen lassen sich hierbei wie folgt berechnen:

- *Globale Konstanten:* $\text{size}_{\text{general}} = 24$,
- *Split-Funktion:* $\text{size}_{\text{splitUHF}} = (l+1) \cdot \log m + 128 + (l+1) \cdot (k+6)$,
- *Offsets der Wertebereiche:* $\text{size}_{\text{offsets}} = 64 \cdot (m+1)$,
- *Gute Funktionen-Paare:*

$$\text{size}_{\text{goodPairs}} = 3 \cdot (l+1) \cdot m \cdot \log \hat{n} + 2 \cdot (l+1) \cdot (k+7) \cdot m + 128,$$

- *Geteilte Tabellen:* $\text{size}_{\text{tables}} = \hat{n}^{\frac{3}{2}} \cdot (12 \cdot \log \hat{n} + 72)$,
- *Zufällige Faktoren:* $\text{size}_{\text{factors}} = m \cdot (\log \hat{n} + 6)$ und
- g_i*-Arrays:* $\text{size}_{\text{arrays}} = 2,5 \cdot n$.

Der Gesamtspeicherplatz ergibt sich dann per Addition dieser einzelnen Komponenten.

2.4 Geschätzte und tatsächliche maximale Bucketgröße

Wie in Abschnitt 1.4.2 erwähnt, haben wir für eine gegebene Schlüsselmenge mit n Schlüsseln sowie m Buckets die maximale Bucketgröße $\hat{n}$ nach [RS98, Theorem 1] durch

$$\frac{n}{m} + \sqrt{\frac{2 \cdot n \cdot \log m}{m}}$$

abgeschätzt.

In Abb. 2.1 wird für gegebene n unter Verwendung dieser Abschätzung die geschätzte mit den in den Testläufen tatsächlich aufgetretenen maximalen Bucketgrößen verglichen. Während die absolute Abweichung der geschätzten von der durchschnittlichen maximalen Bucketgröße für größere Schlüsselmengen zunimmt, sinkt die prozentuale Abweichung jedoch kontinuierlich:

Anzahl Schlüssel	10^4	10^5	10^6	10^7	10^8	10^9
Absolute Abweichung	$7,68$	$11,56$	$19,64$	$29,76$	$49,52$	84
Prozentuale Abweichung	$12,32\%$	$8,12\%$	$5,94\%$	$3,80\%$	$2,65\%$	$1,86\%$

Tabelle 2.1

Für $n \geq 10^6$ ist der geschätzte Wert also tatsächlich relativ nah am tatsächlichen Wert.

Damit können wir die Abschätzung zur Bestimmung der Anzahl von Buckets, die den Gesamtspeicherplatz zu minimieren versucht, verwenden.

In Abb. 2.2 wird der Anteil des Gesamtspeicherplatzes dargestellt, der von m und $\hat{n}$ (mit der Abschätzung also von n und m) abhängt. Wie leicht zu sehen ist, ist die gesuchte Bucketanzahl m, die den Gesamtspeicherplatz minimiert, für $1 \leq m \leq n$ eindeutig bestimmt.

Um diese zu finden, betrachten wir die erste Ableitung der dargestellten Funktionen (siehe Abb. 2.3). Mittels Bisektion (beginnend bei $m_{\min} = 1$ mit negativem Funktionswert und $m_{\max} = n$ mit positivem Funktionswert der ersten Ableitung) kann somit in $\lceil \log n \rceil$ Schritten dasjenige m gefunden werden, das der Nullstelle der Ableitung und somit dem Minimum der ursprünglichen Funktion am nächsten kommt. Auch für $n = 10^9$ sind dies nur 30 Iterationen, in denen insgesamt 31 Funktionswerte der ersten Ableitung berechnet werden müssen.

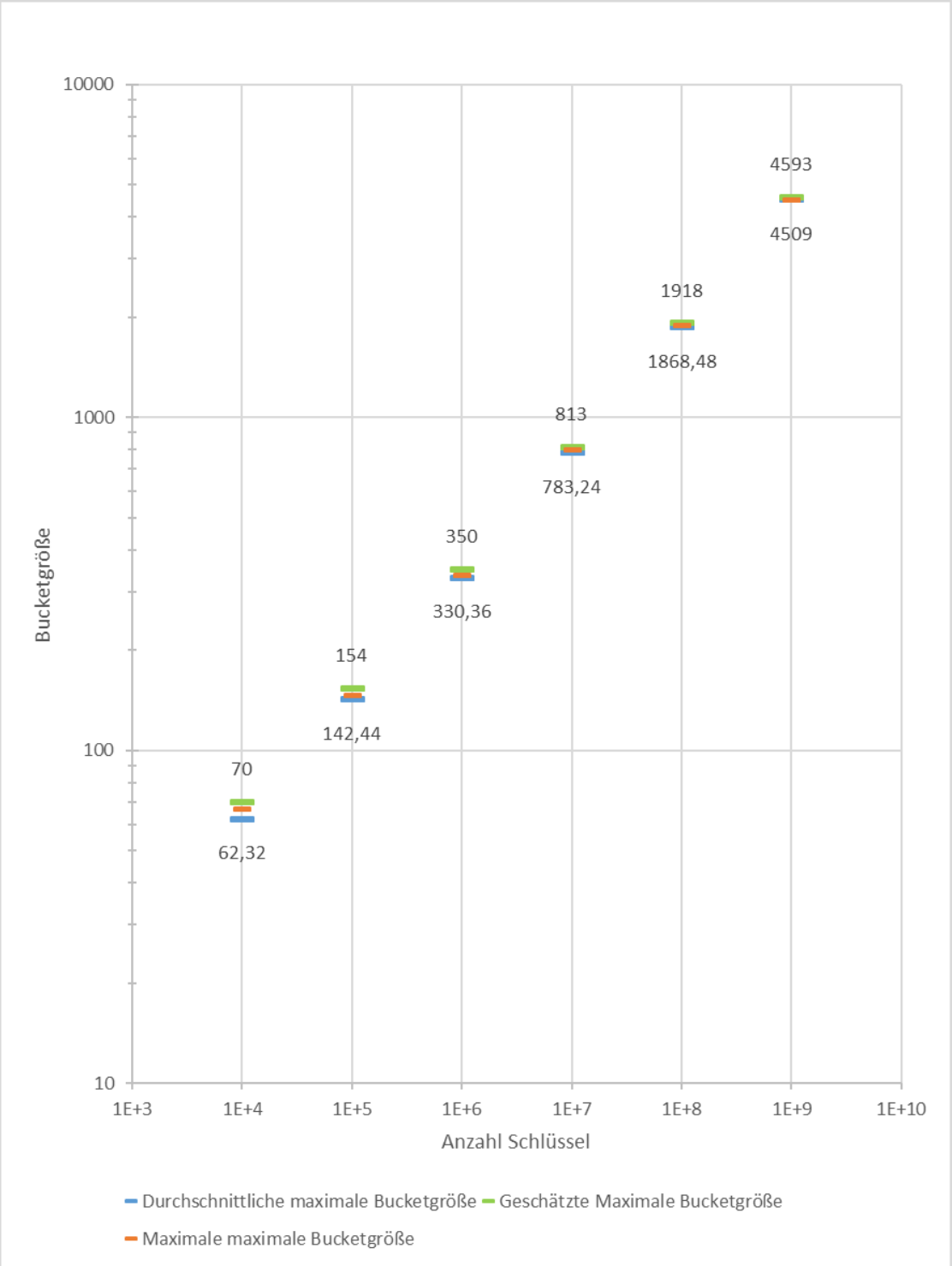

Abbildung 2.1. Tatsächliche und geschätzte maximale Bucketgröße in Abhängigkeit von n

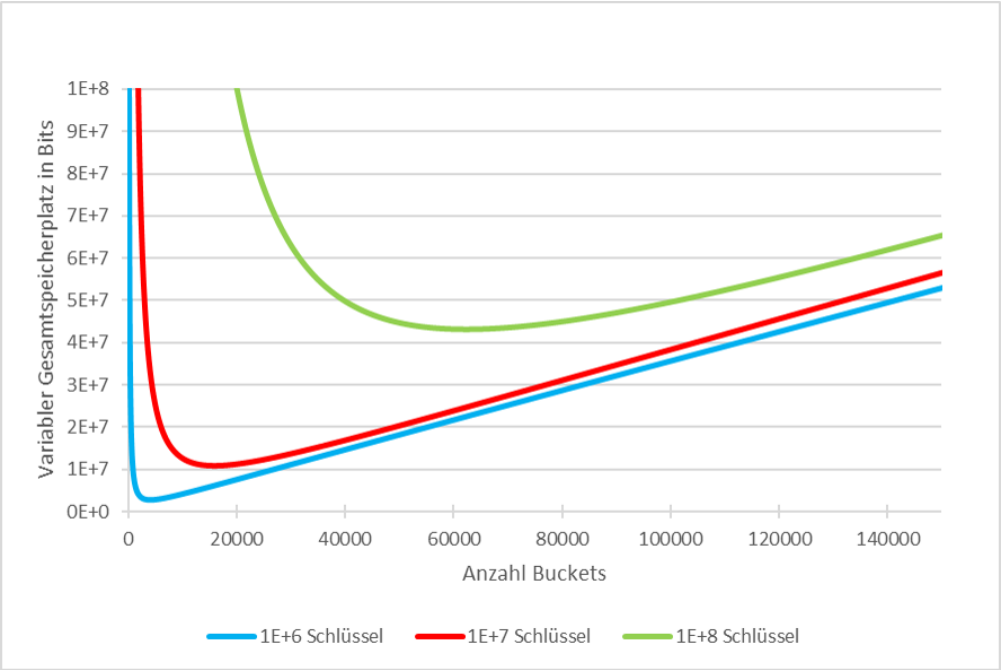

Abbildung 2.2. Die zu minimierende Funktion (siehe Abschnitt 2.3)

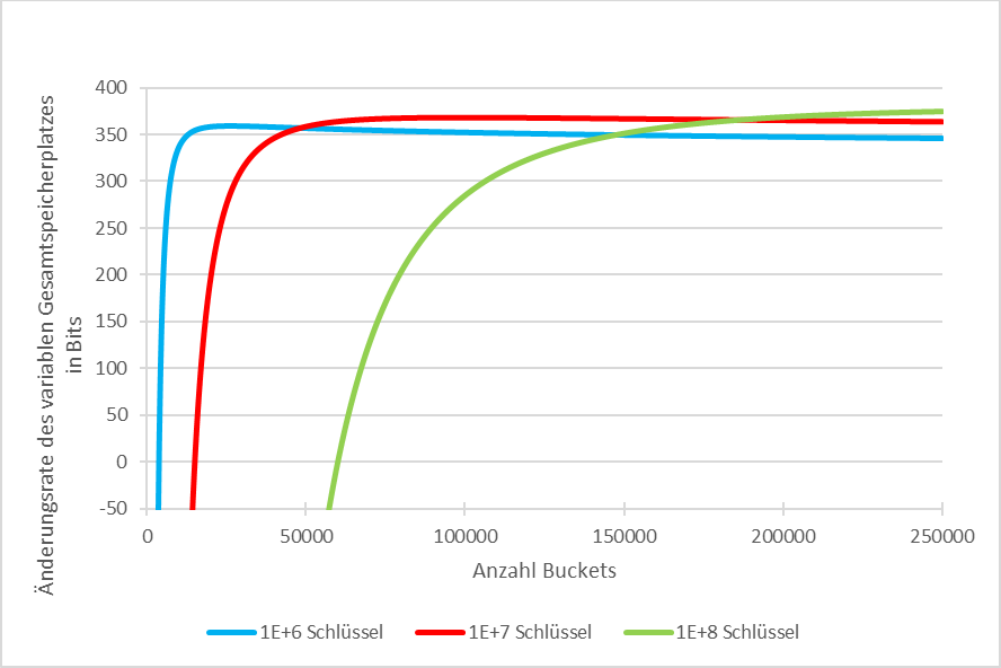

Abbildung 2.3. Die erste Ableitung der zu minimierenden Funktion

2.5 Anzahl Buckets und durchschnittliche maximale Bucketgröße

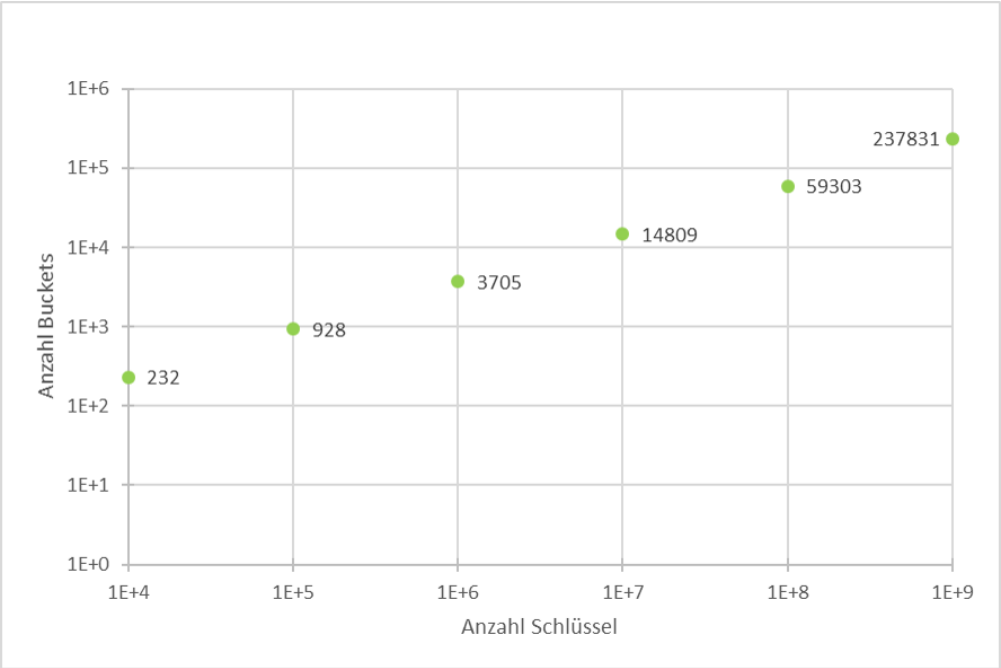

Abbildung 2.4. Die berechneten Bucketanzahlen in Abhängigkeit von n

Unter Verwendung der in Abschnitt 2.4 dargestellten Abschätzung ergaben sich in den Testläufen die in Abb. 2.4 dargestellten Bucketanzahlen und die in Abb. 2.5 dargestellten zugehörigen tatsächlich aufgetretenen maximalen Bucketgrößen. Während sowohl die Anzahl der Buckets als auch die durchschnittliche maximale Bucketgröße in etwa linear anwachsen (was ausschlaggebend für das Wachstum des Gesamtspeicherplatzes ist, siehe Abschnitt 2.6), verringert sich die relative Abweichung von der durchschnittlichen maximalen Bucketgröße bei steigender Schlüsselzahl kontinuierlich.

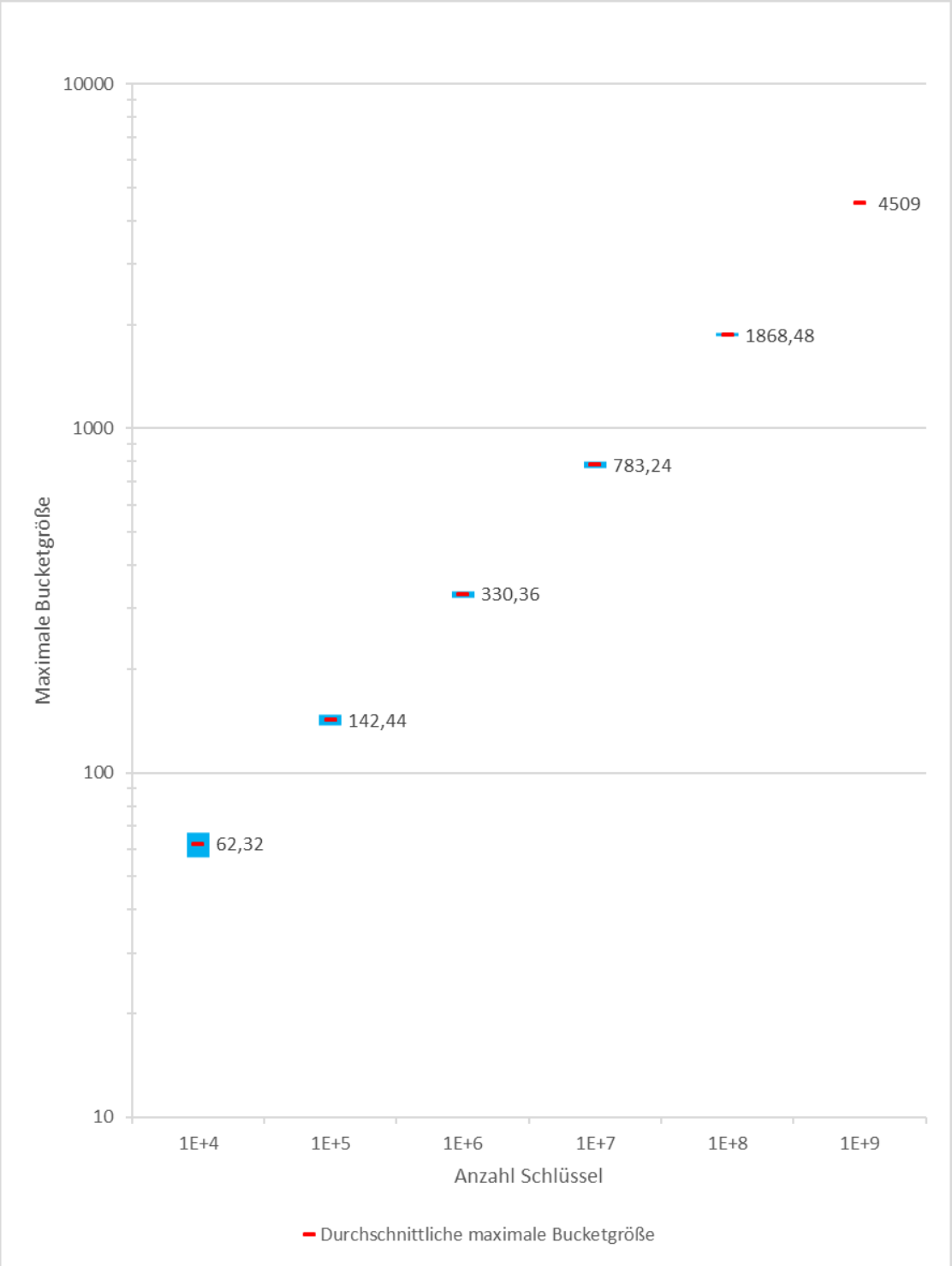

Abbildung 2.5. Die tatsächlich verwendeten maximalen Bucketgrößen in Abhängigkeit von n

2.6 Durchschnittlicher Gesamtspeicherplatz und Split-and-Share-Overhead

In Abb. 2.6 erkennen wir, dass der Gesamtspeicherplatz in etwa linear in der Größe der Schlüsselmenge wächst. Dies liegt insbesondere an dem ebenfalls linearen Wachstum der Anzahl der Buckets und der durchschnittlichen maximalen Bucketgröße, was in Abschnitt 2.5 beobachtet wurde. Ebenso deckt sich dieses Ergebnis mit den theoretischen Überlegungen in [BPZ13]. Andererseits sinkt der relative Speicherplatz pro Schlüssel antiproportional zur Größe der Schlüsselmenge. In Abb. 2.7 sehen wir insbesondere, dass der zusätzlich zu den (auch ohne Verwendung des Split-and-Share-Verfahrens notwendigen) $2,5$ Bit pro Schlüssel entstehende Overhead ab 10^7 Schlüsseln mit fast 40% und insbesondere bereits für 10^8 Schlüsseln mit $16,5\%$ erfreulich gering ist.

2.7 Durchschnittlicher Overhead durch "schlechte" Zufallszahlen

An einigen Stellen bei der Konstruktion der perfekten Hashfunktion sind wir auf die Wahl zufälliger Zahlen angewiesen, die anschließend gute Eigenschaften erfüllen müssen.

Während die Wahl der Splitfunktion h_{split} in sämtlichen Testläufen sofort erfolgreich war (d.h. die einzelnen Buckets wurden nicht zu groß), kam es - wie in Abb. 2.8 zu sehen - gelegentlich vor, dass ein zufälliges Paar 1-universeller Hashfunktionen nicht gut war. Allerdings beschränkte sich dies selbst bei größeren Schlüsselmengen nur auf einen geringen (aber wie zu erwarten wachsenden) Anteil.

Ebenso erwies sich die Einführung der Zufallsfaktoren s_i als Ergänzung zu den Zufallstabellen als äußerst sinnvoll. Während in keinem der Durchläufe neue Tabellen erzeugt werden mussten, zeigte sich besonders für kleine Schlüsselmengen eine erhöhte Notwendigkeit, einen anderen Zufallsfaktor zu erzeugen, da die damit in Abschnitt 1.8 beschriebene erzeugte Hashfunktion nicht injektiv war. Da der Anteil wiederholter Erzeugungen von Zufallszahlen sogar noch im Bereich von 10^6 bis 10^8 Schlüsseln von knapp 90% auf $12,52\%$ fällt, hätte dies katastrophale Folgen auf die Laufzeit gehabt, wenn stattdessen immer wieder neue Zufallstabellen hätten erzeugt werden müssen, die wiederum für sämtliche Buckets eine gute Eigenschaft besitzen müssen.

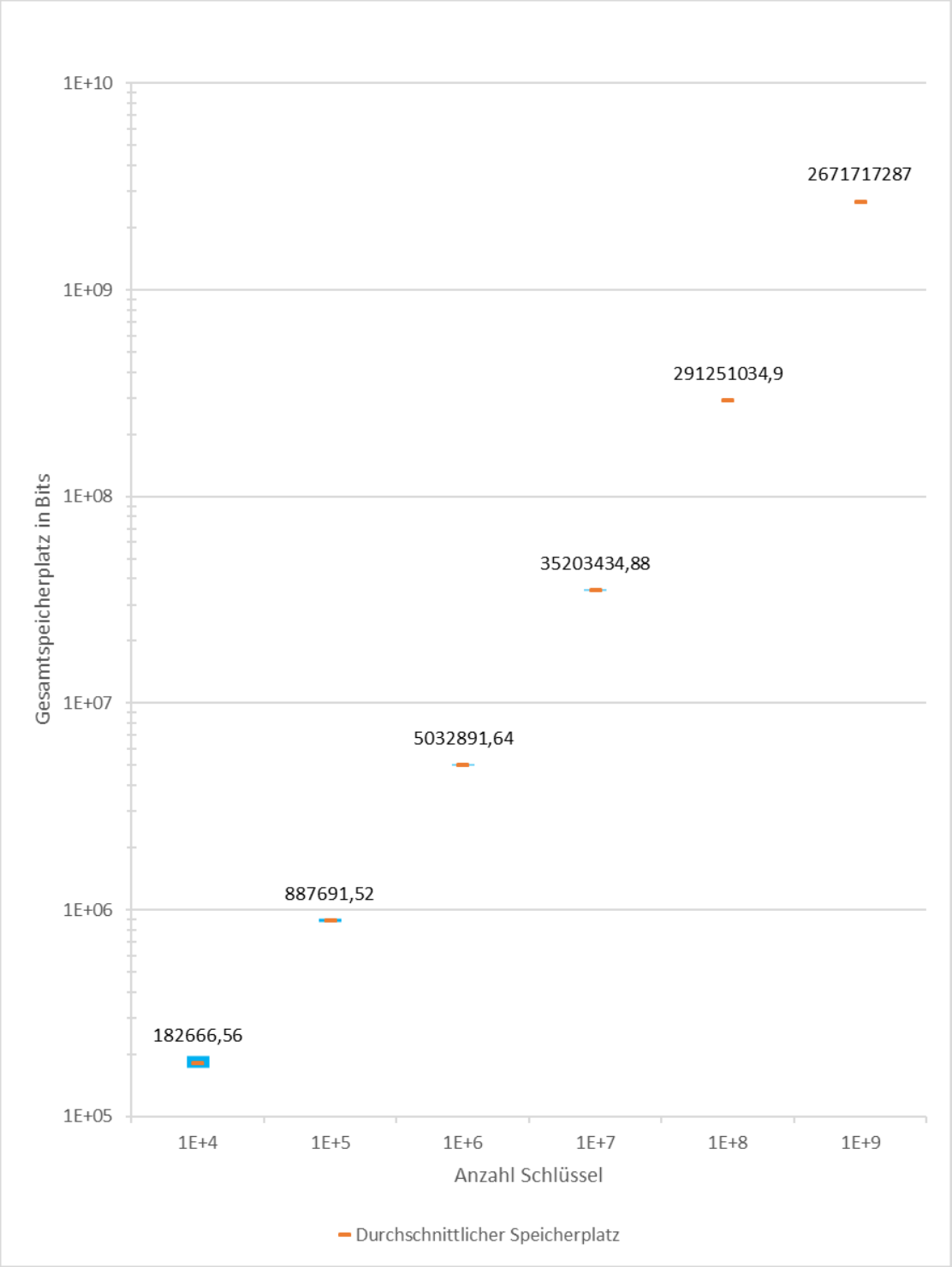

Abbildung 2.6. Der Gesamtspeicherplatz in Bits in Abhängigkeit von n

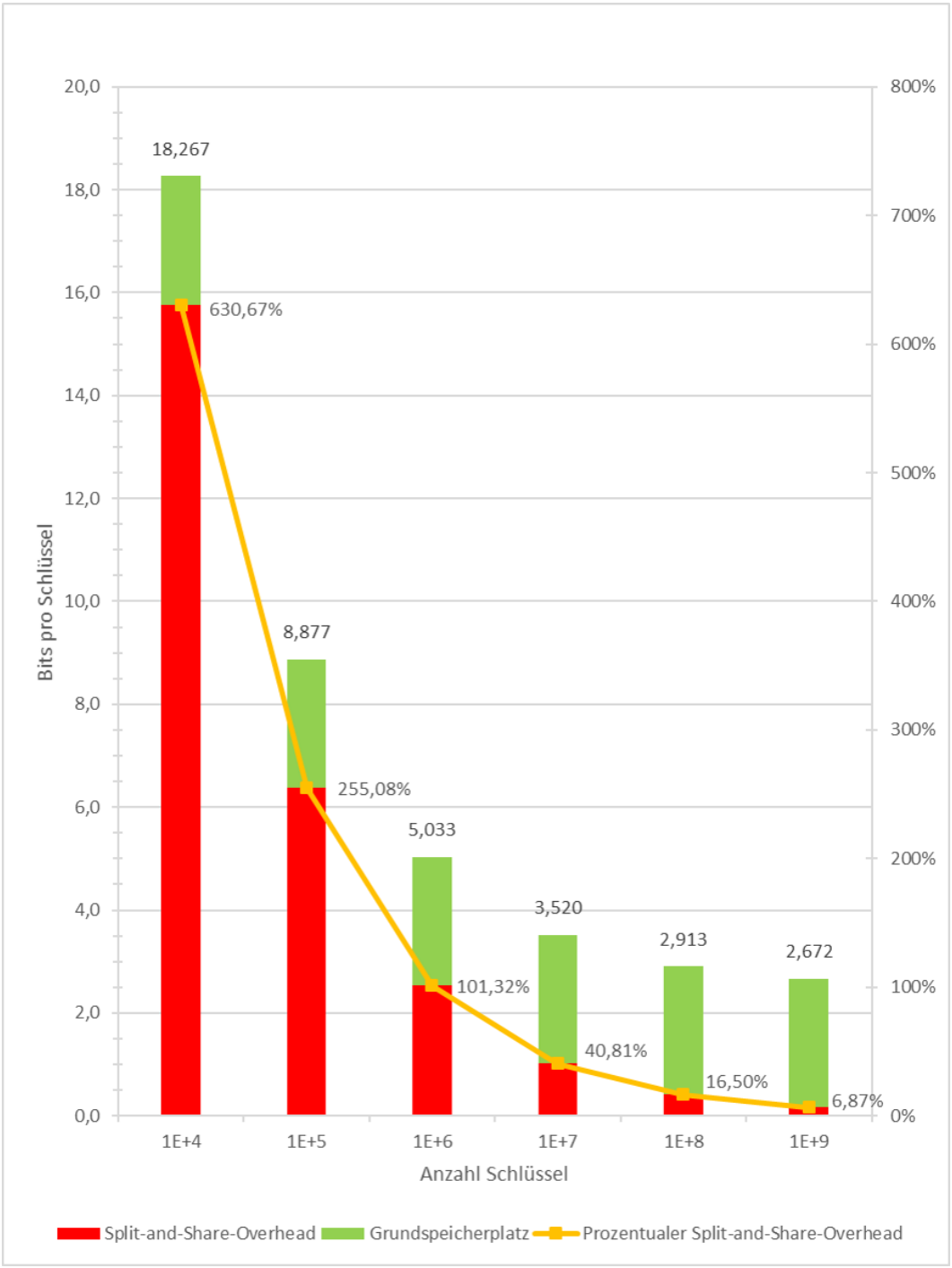

Abbildung 2.7. Die Anzahl der Bits pro Schlüssel sowie der prozentuale Anteil des Overheads in Abhängigkeit von n

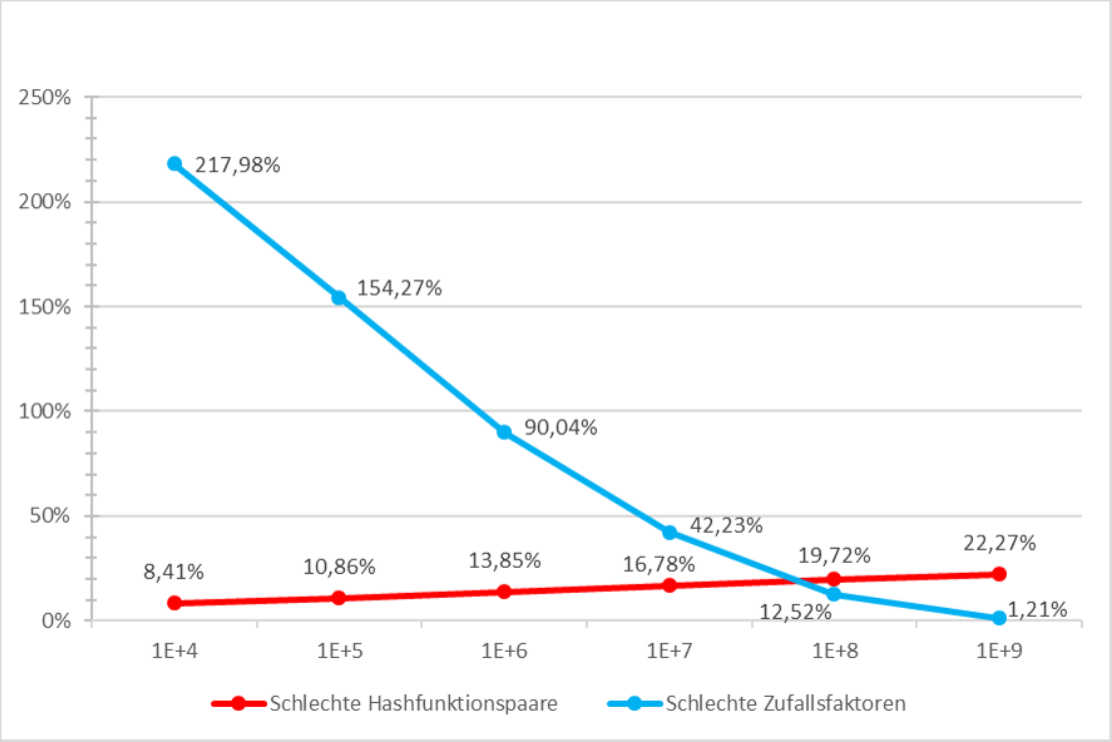

Abbildung 2.8. Der Overhead durch Wahl ungünstiger Zufallszahlen in Abhängigkeit von n

2.8 Durchschnittliche Konstruktionszeit

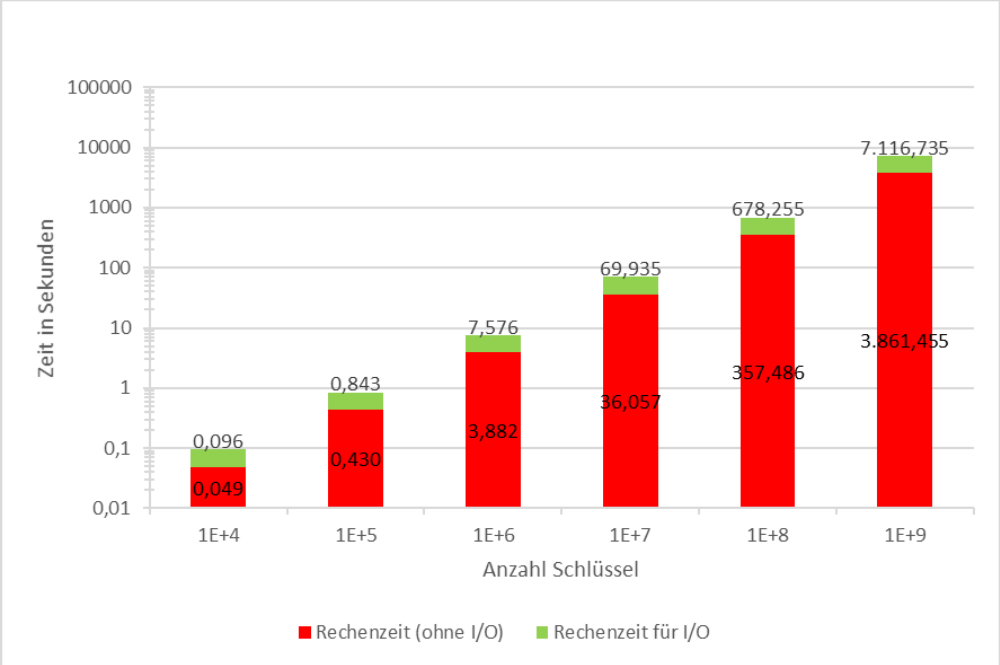

Abbildung 2.9. Die durchschnittliche Konstruktionszeit sowie der Anteil der I/O-Operationen in Abhängigkeit von n

Die auf dem Testrechner erreichten Konstruktionszeiten liegen (trotz der Verwendung temporärer Dateien) für $n \leq 10^8$ in einem akzeptablen Rahmen. Wie in Abb. 2.9 zu sehen, wächst die Konstruktionszeit in etwa proportional mit der Größe der Schlüsselmenge. Dies deckt sich also mit den theoretischen Überlegungen aus [BPZ13]. Ebenso kann man den doch relativ hohen Anteil der Zeit für die I/O-Operationen (fast 50%) erkennen. Auf entsprechender Hardware, die auch für größere Schlüsselmengen nicht auf das Arbeiten mit temporären Dateien angewiesen ist, sollte sich dieser Anteil entsprechend deutlich reduzieren lassen.

2.9 Durchschnittliche Auswertezeit

Abschließend haben wir noch die benötigte Zeit zur Auswertung, d.h. Berechnung des Hashwertes für einen gegebenen Schlüssel, untersucht. Im Einklang mit unserem Ziel blieb diese auch für große Schlüsselmengen im Wesentlichen konstant. Abb. 2.10 zeigt die pro Testlauf gemessenen durchschnittlichen Auswertezeiten und deren Mittelwerte.

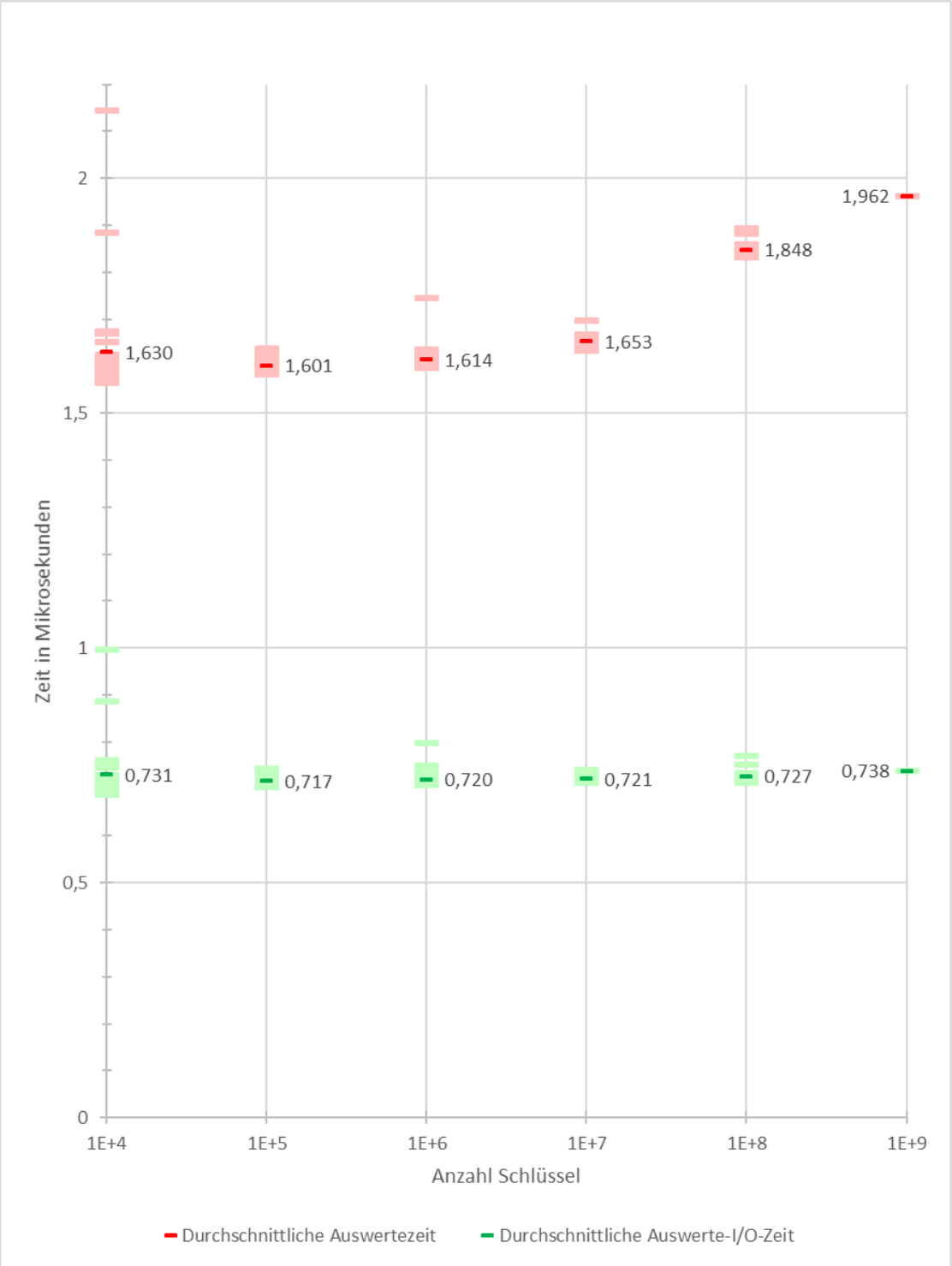

Abbildung 2.10. Die durchschnittliche Auswertezeit pro Schlüssel in Abhängigkeit von n

Kapitel III
Fazit

Unsere Untersuchungen haben gezeigt, dass das “Split-and-Share”-Verfahren zur Konstruktion perfekter Hashfunktionen auch für “kleine” Schlüsselmengen (zwischen 10^6 und 10^8 Schlüssel) sinnvoll sein kann. Einerseits der nicht mehr zu große Overhead bzgl. des Gesamtspeicherplatzes (Abschnitt 2.7) und andererseits die hohen Erfolgsquoten und geringen Anzahlen wiederholter Erzeugungen von Zufallszahlen (Abschnitt 2.8) weisen das “Split-and-Share”-Verfahren auch für diese Schlüsselmengen als geeignet aus.

Zum Schluss sei nochmals angemerkt, dass unsere Implementierung natürlich Spielraum zum Optimieren von Gesamtspeicherplatz (z.B. arithmetische Kodierung) und Konstruktions- sowie Auswertezeiten (z.B. effizientere Datenstrukturen und Algorithmen, z.B. für den Azyklizitätstest) lässt.

Literaturverzeichnis

[BPZ13] Fabiano C. Botelho, Rasmus Pagh und Nivio Ziviani: *Practical perfect hashing in nearly optimal space*. Information Systems, Vol. 38 (1): Seiten 108 – 131, 2013. doi:10.1016/j.is.2012.06.002.

[CHM97] Zbigniew J. Czech, George Havas und Bohdan S. Majewski: *Fundamental study perfect hashing*. Theoretical Computer Science, Vol. 182 (1): Seiten 1–143, 1997.

[Die07] Martin Dietzfelbinger: *Stochastic Algorithms: Foundations and Applications: 4th International Symposium, SAGA 2007, Zurich, Switzerland, September 13-14, 2007. Proceedings*, Kapitel Design Strategies for Minimal Perfect Hash Functions, Seiten 2–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74871-7_2.

[RS98] Martin Raab und Angelika Steger: *Randomization and Approximation Techniques in Computer Science: Second International Workshop, RANDOM'98 Barcelona, Spain, October 8–10, 1998 Proceedings*, Kapitel "Balls into Bins" — A Simple and Tight Analysis, Seiten 159–170. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. doi:10.1007/3-540-49543-6_13.