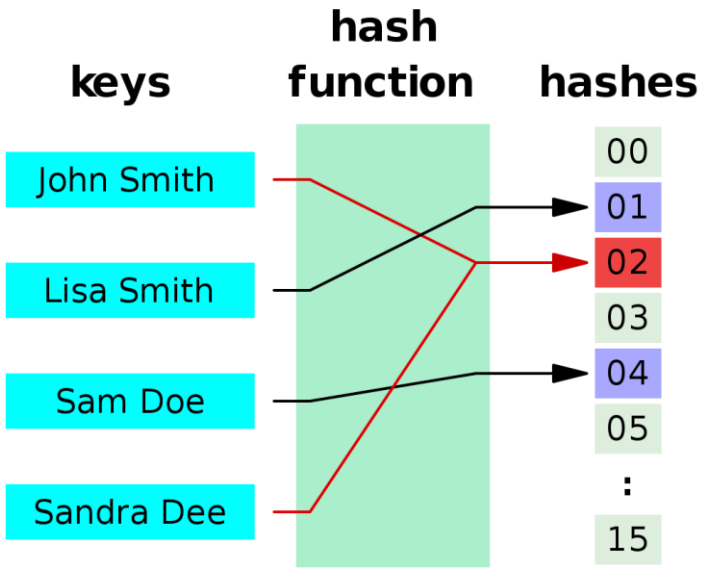
Effizienz verschiedener Hashverfahren



von
Team IV: Ivan Kalinin, Christopher Sann, Niklas Warmuth

Gliederung

- Einführung
- Ausgangssituation
- Ziel
- Vorgehen und Implementation
- Vorstellung einzelner Hashverfahren
- Zugriffszeit
- Kollisionen
- Fazit
- Literatur + Quellen

Experimente
< >
Algorithmik

AUSGANGSSITUATION

Projekt

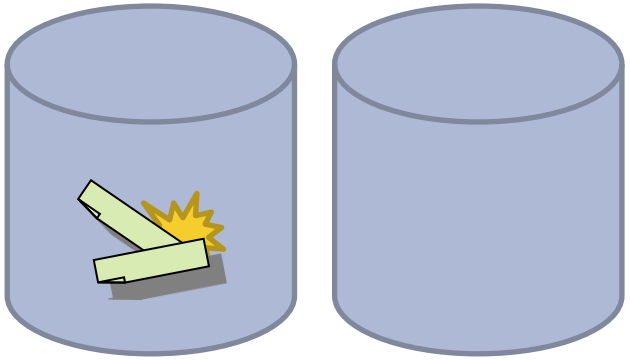
Aufgabenstellung

- Machen Sie ‚irgendetwas Sinnvolles' zum Thema „Suchmaschinen"
- Schwerpunkt natürlich im Bereich „Experimentelle Algorithmik"
- Programmiersprache freigestellt

Hashing!

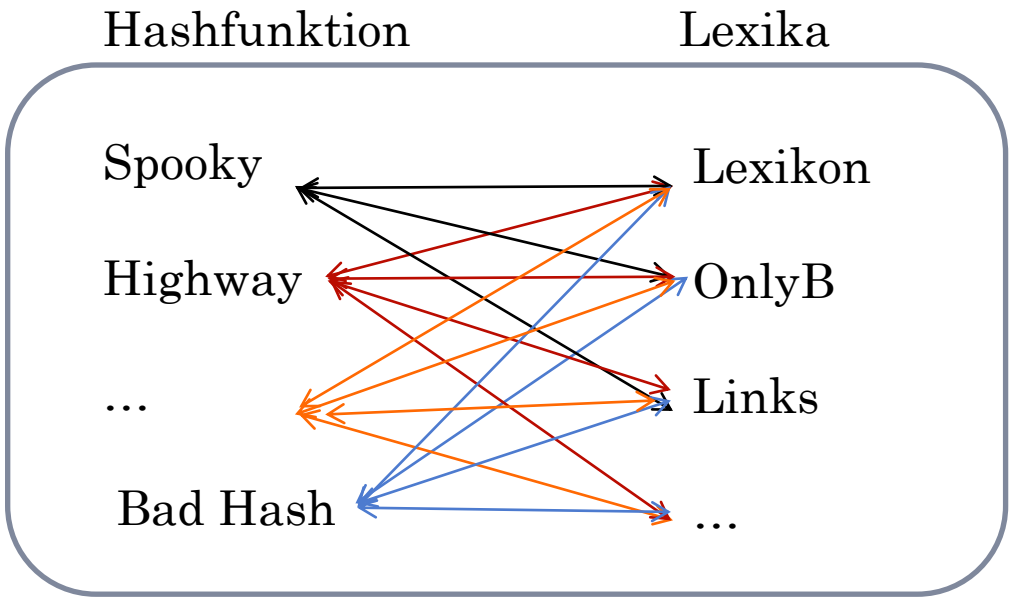
Ziel

- Testen verschiedener Hashverfahren
  - Wie schneller Zugriff auf Informationen?
  - Wie effizient ist das Kollisionsverhalten?



Vorgehen – 1)

- Wir erstellen für jede Hashfunktion eine Map und füllen sie dann mit einer der Zeilen aus einer der Lexika



Implementation

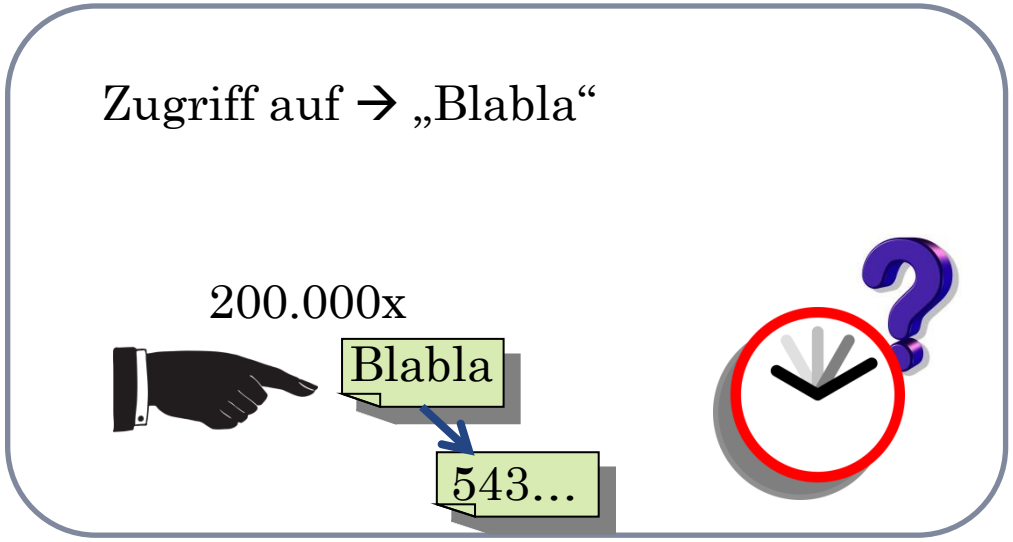
- Betrachtung drei verschiedener Datensätze:
  - Deutsches Lexikon (113.439 Zeilen) 1407 kB
    Bsp: Jonglierer, Logarithmus
  - Alle Wörter mit B (8167 Zeilen) 99 kB
    Bsp: Baustelle, Beamter
  - Links (17395 Zeilen) 1181 kB
    Bsp: https://www.thm.de, https://www.giessen.de

Programm: Standard Bibliotheken von C++

CPU: Intel Core i5-3450 Quadcore @ 3.10 GHz; RAM: 16 GB
Windows 10; Microsoft Visual Studio - Kompiliert auf Release x64

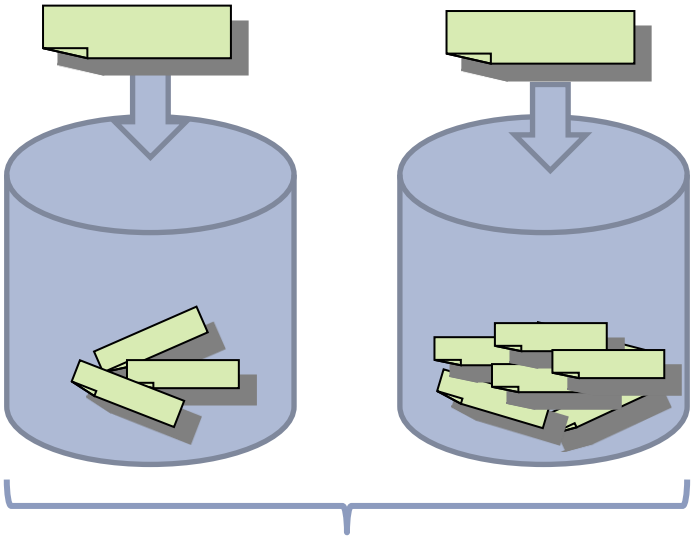
VORGEHEN – 2)

- Dann wählen wir 4 zufällige Keys aus, für die jeweils die Zeit gemessen wird, die benötigt wird, um 200.000 mal auf diesen Key zuzugreifen



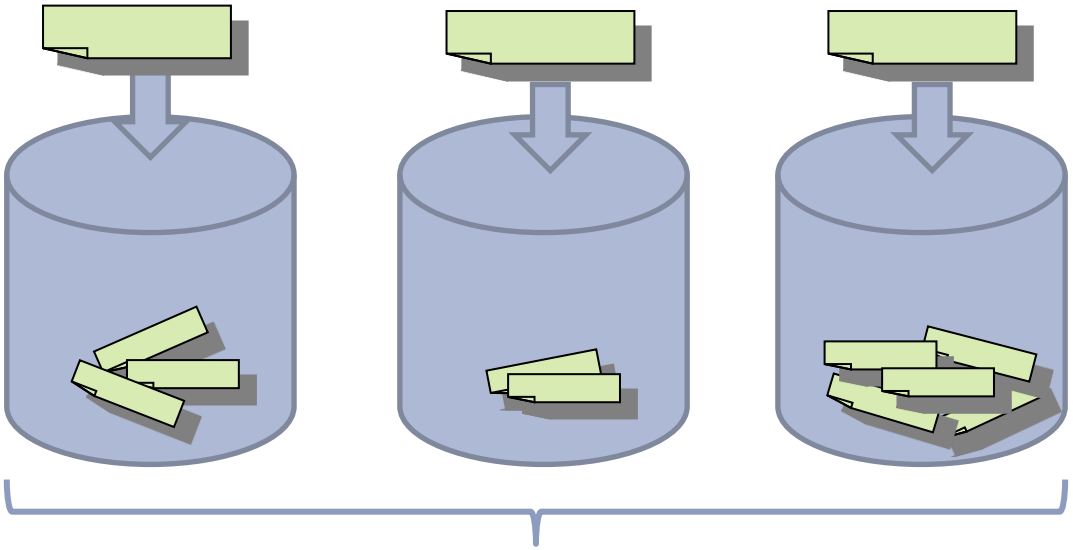
Vorgehen – 3)

- Danach lassen wir die Keys auf Grundlage ihres Hashes auf n Buckets verteilen und berechnen die Standardabweichung von der Durchschitts-verteilung bei n Buckets
- Das machen wir für n von 2 bis zur Länge der Map



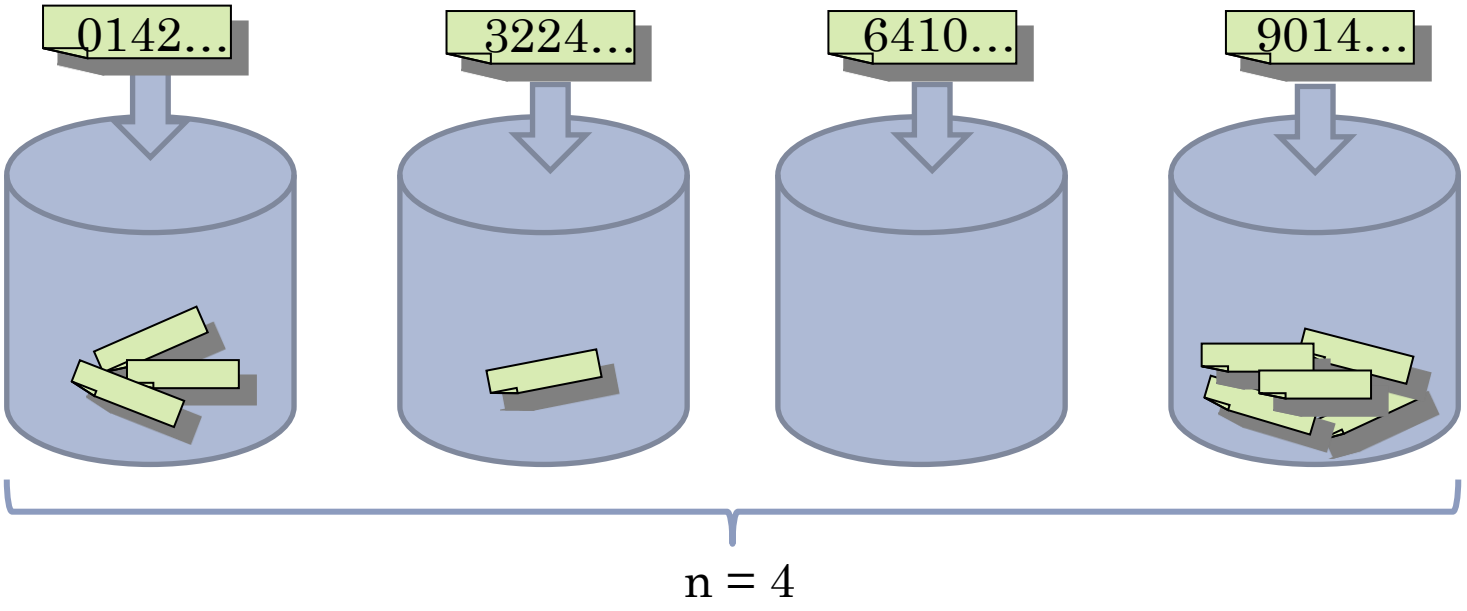
Vorgehen – 3)

- Danach lassen wir die Keys auf Grundlage ihres Hashes auf n Buckets verteilen und berechnen die Standardabweichung von der Durchschitts-verteilung bei n Buckets
- Das machen wir für n von 2 bis zur Länge der Map



Vorgehen – 3)

- Danach lassen wir die Keys auf Grundlage ihres Hashes auf n Buckets verteilen und berechnen die Standardabweichung von der Durchschitts-verteilung bei n Buckets
- Das machen wir für n von 2 bis zur Länge der Map



Hashes – Bad Hash

- Ein schlechter Hash, der immer denselben Wert zurückgibt.
- *Was passiert, wenn der „Teufel“ mit eurem Hash spielt?*

```
unsigned long badHash(string key) {
    return 0;
}
```

Hashes – Low Quality Hash

- Erzeugt Hash aus den ersten vier Buchstaben aus dem Key (es sei denn, der Key ist kürzer)
- Steht und fällt mit der Variablität der ersten vier Zeichen.

```
unsigned long lowQualityHash(string key) {
    switch (key.length()) {
    case 0: return 0;
    case 1: return key[0];
    case 2: return key[0] | (key[1] << 8);
    case 3: return key[0] | (key[1] << 8) | (key[2] << 16);
    default: return key[0] | (key[1] << 8) | (key[2] << 16) | (key[2] << 24);
    }
}
```

Hashes – Jenkins Hash

- Hash für Strings von Bob Jenkins
- Vorläufer von lookup, SpookyHash

```
unsigned long jenkins_one_at_a_time_hash(const string key) {


    unsigned long hash = 0;
    for (int i = 0; i<key.size(); i++) {
        hash += key[i];
        hash += hash << 10;
        hash ^= hash >> 6;
    }
    hash += hash << 3;
    hash ^= hash >> 11;
    hash += hash << 15;
    return hash;
}
```

Hashes – Spooky

- Erstellt von Bob Jenkins 2012
- All Rounder mit 64, 128 and 256 bit
- Erweiterung des lookup3 und Jenkins at a time

Hashes – MurMur 3

- All Rounder mit 32, 64 oder 128 Bit
- War lange Zeit ein Standard
- Fand Verwendung in Java, Ruby, C#, …
- Erweiterung um Hashflooding zu verhindern

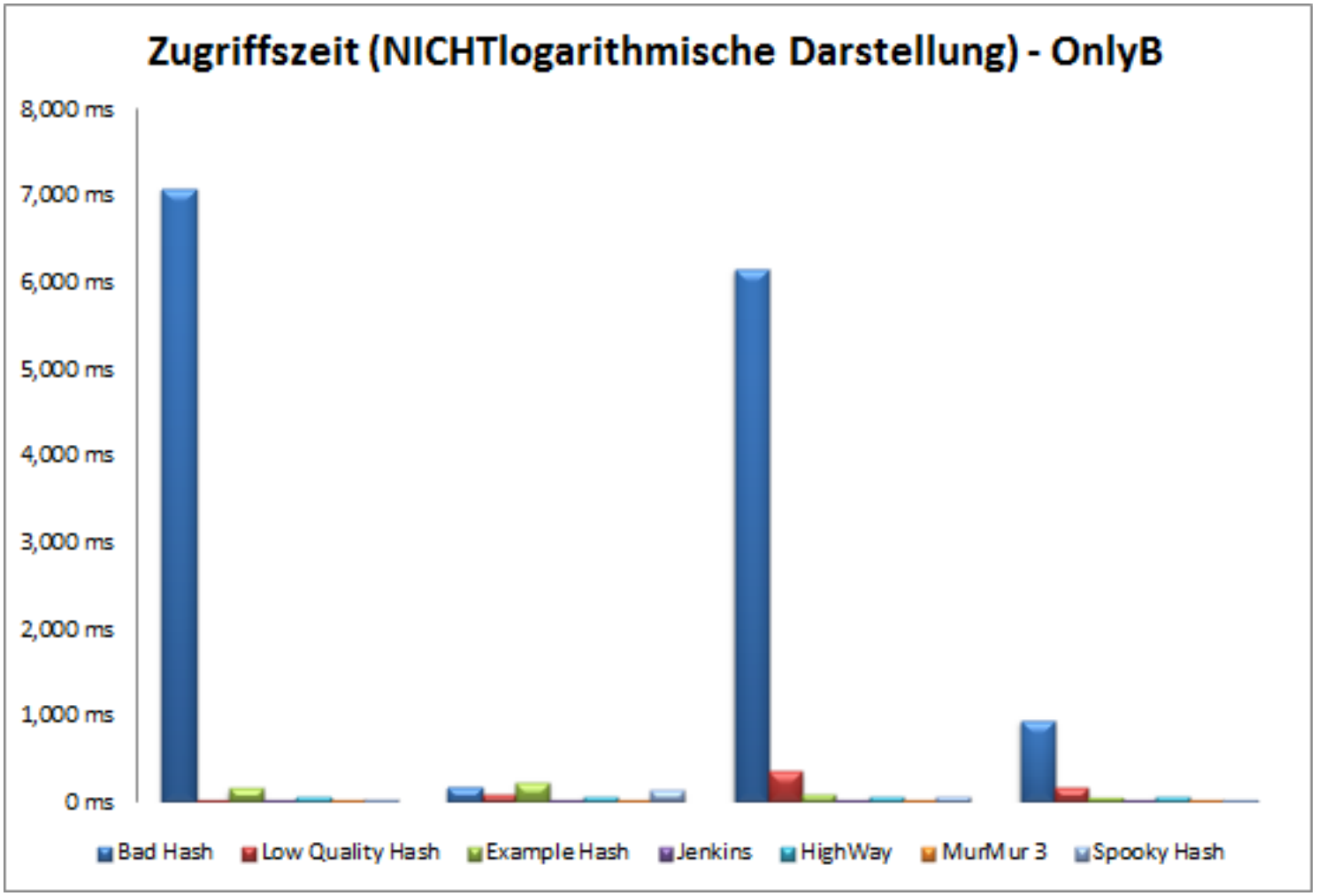
Hashes - SIP

- Kryptografisch sicherer All-Rounder mit 64 bit
- Schwachstelle in Ruby – Murmur – 2012
- Verbreiteter Standard
- Verwendet in C++, Python, C#, Rust, GO, Swift, …
- Leider nicht getestet

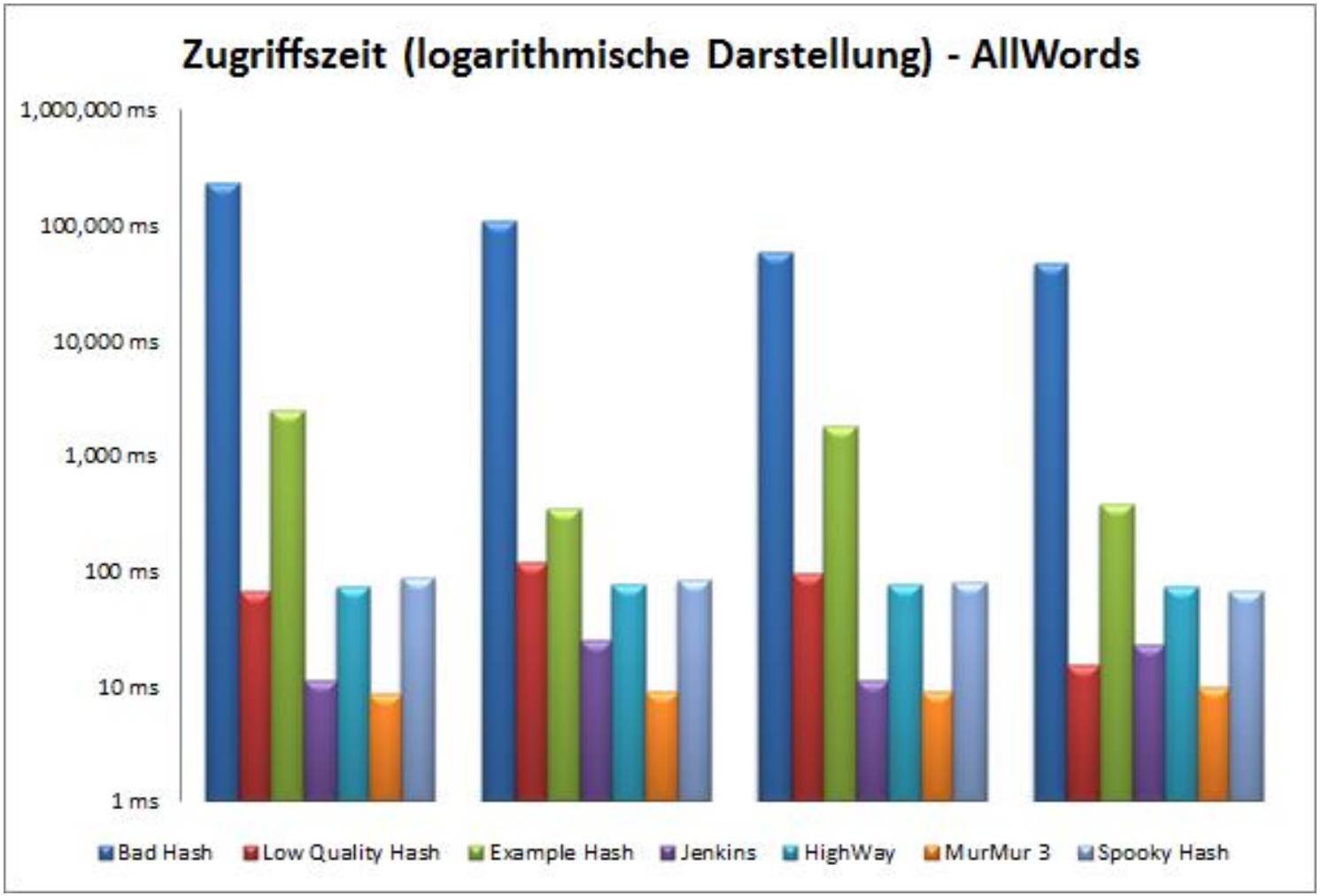
Hashes – HighWay Hash

- Kryptografischer All Rounder mit 64, 128, 256 Bit
- Öffentlich implementiert in 2016

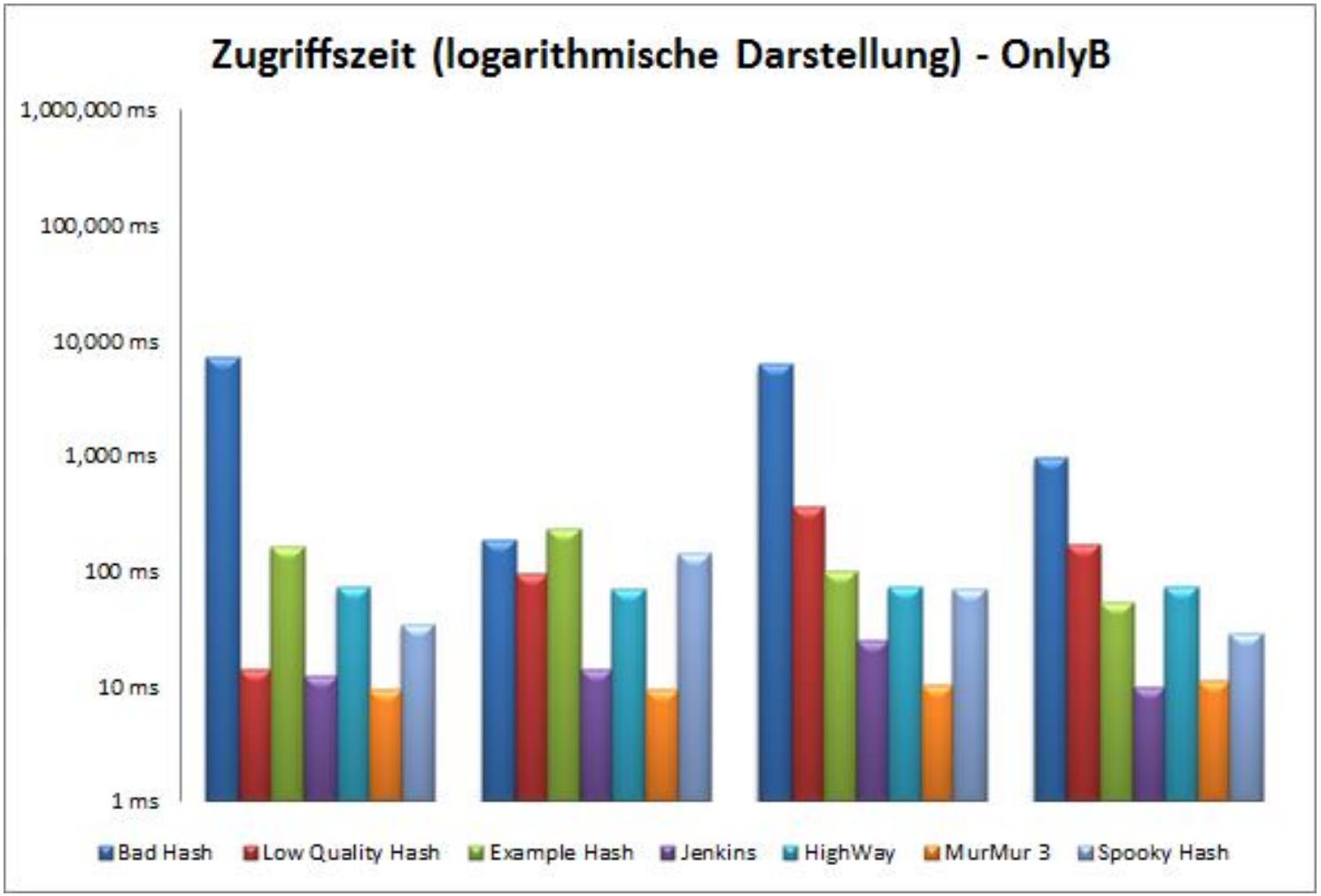
Zugriffszeit – OnlyB Nichtlogarithmisch Ein Beispiel



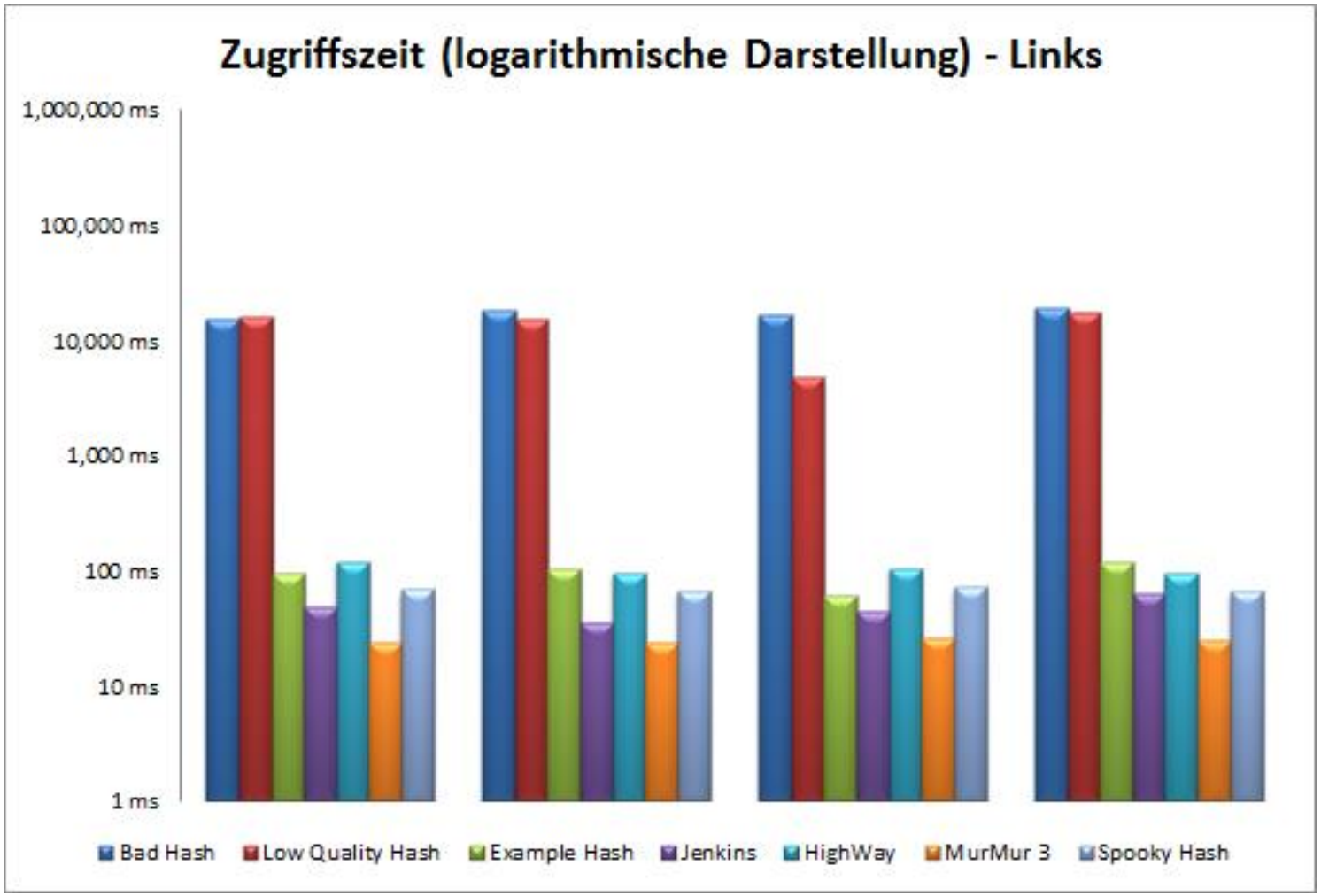
Zugriffszeit – AllWords



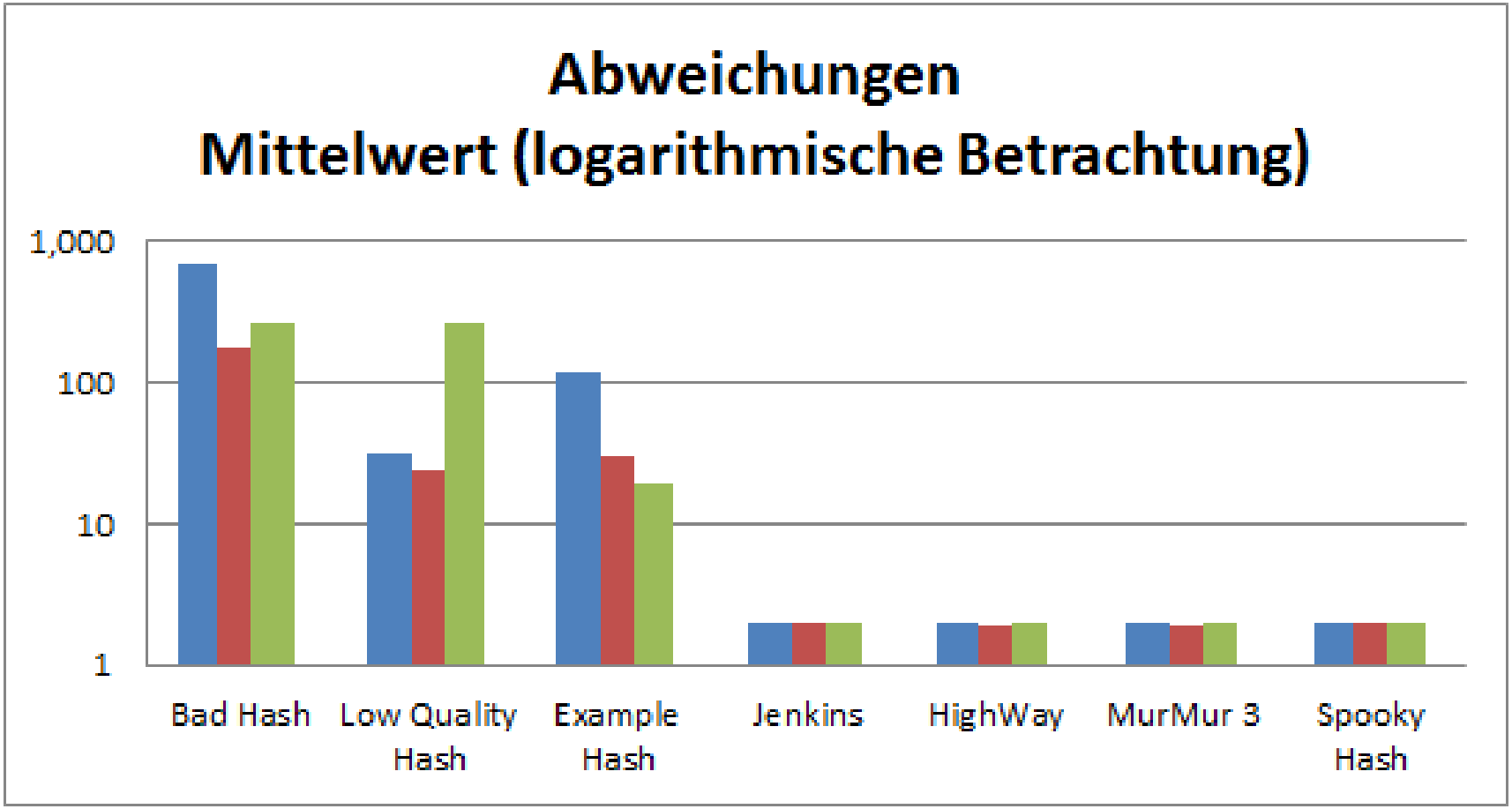
Zugriffszeit – OnlyB



Zugriffszeit – Links



KOLLISIONEN



Blau = AllWords | Rot = OnlyB | Grün = Links

Fazit

- Effizienter Hash wichtig für zügigen Zugriff
- MurMur 3 hat sich im Test als der schnellste aller getesteten Hashes herausgestellt
- Aber: je nach Hashmap & Daten ist ein Hashverfahren mehr oder weniger geeignet
  → Es gibt keinen „perfekten“ Hash

Fazit

- Effiziente Implementation ist die Voraussetzung für gute Programme – einzelne Algorithmen können helfen, sind aber kein Allheilmittel
- Aktuelle Informationen sind eventuell bedeutend – Anforderungen an Hashes ändern sich alle Jahre mal wieder
- Prozessoren und deren Instruktionsset beeinflussen neue Trends

Literatur

- Algorithmen und Datenstrukturen, Thomas Ottmann, Peter Widmayer, 5. Auflage

- Algorithms and Data Structures – The Basic Toolbox, Kurt Mehlhorn, Peter Sanders

Quellen

- https://en.wikipedia.org/wiki/List_of_hash_functions#Non-cryptographic_hash_functions
- http://www.burtleburtle.net/bob/hash/spooky.html
- https://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/

Quellen

- Github für den Quellcode von Highway, Murmur und Spooky Hash

- Stackoverflow und für allgemeine C++-Fragen sowie:
  - http://de.cppreference.com/w/cpp/container/unordered_map
  - http://www.cplusplus.com/reference/unordered_map/unordered_map/

Fragen, Wünsche, Kritik, Anmerkungen?