Grundproblem: Effiziente Suche nach Schlüsseln

Grundproblem: Effiziente Suche nach Schlüsseln (Im folgenden natürlichzahlige Schlüssel)

Motivation: Datenbank mit einer Milliarde Tupel, mit ganzzahligem Schlüssel (ID).

Bei 1kB pro Tupel: 10^9kB = 10^6MB = 10^3 GB = 1 TB

Grundproblem: Effiziente Suche nach Schlüsseln (Im folgenden natürlichzahlige Schlüssel)

Motivation: Datenbank mit einer Milliarde Tupel, mit ganzzahligem Schlüssel (ID).

Bei 1kB pro Tupel: 10^9kB = 10^6MB = 10^3 GB = 1 TB

Naiver Ansatz: Linearer Scan

- Lade alle Daten in den Hauptspeicher, und terminiere wenn Schlüssel gefunden

Transferrate Festplatte: Seagate Desktop SSHD 2000GB SATA (CHIP-Testsieger. 13.11.2013): 234MB/s

Gesamtzeit: 1.000.000MB / (234MB/s) = 1.000.000/234 s = 4273s > 1h

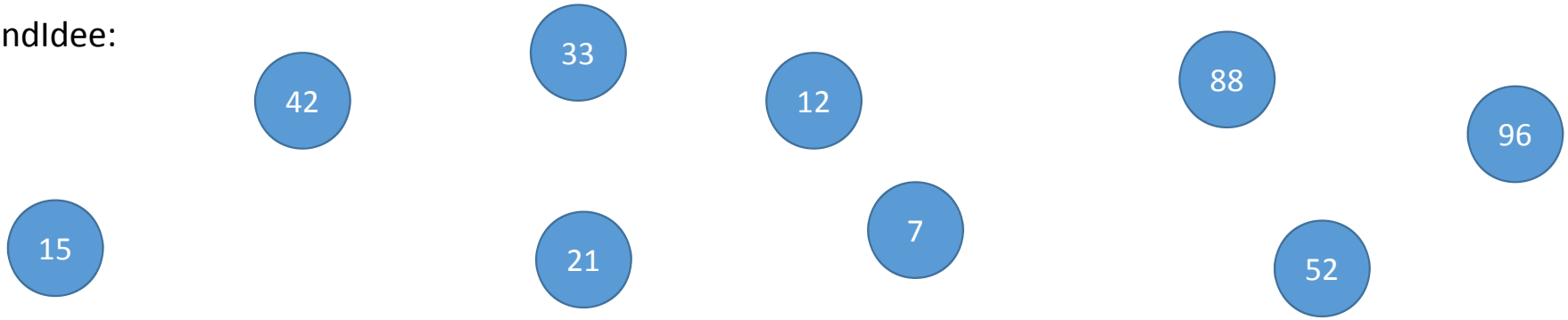
Grundidee: Verwenden eines Index zum effizienten Zugreifen auf Datensätze.

Beispiel: B-Baum mit 6 Levels

5 Directoryseitenzugriffe. 1 Datenseitenzugriff. 4kB pro Seite.
6*Latenzzeit+Transferzeit für 24kb =6*17ms + 24kb / (234MB/s)=104ms+0.1ms ~ 0.1s

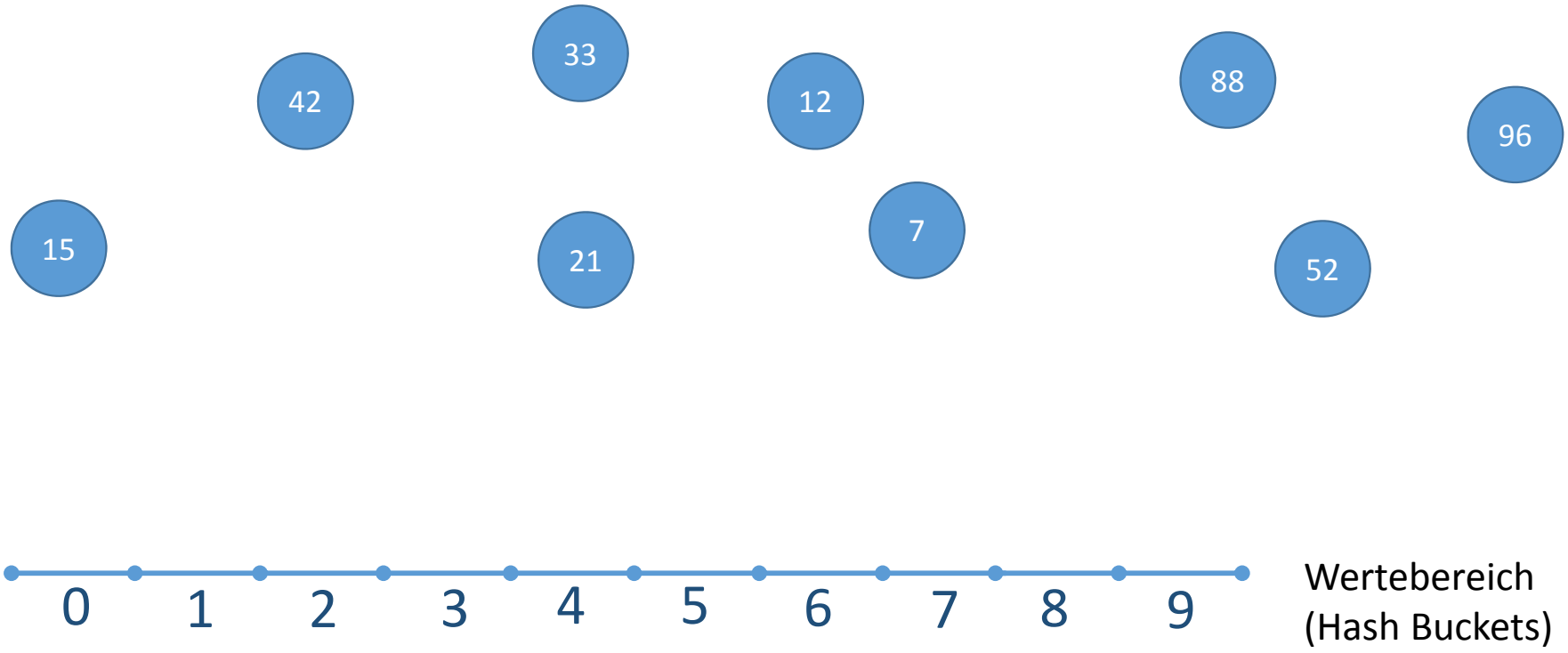
Hashfile

GrundIdee:



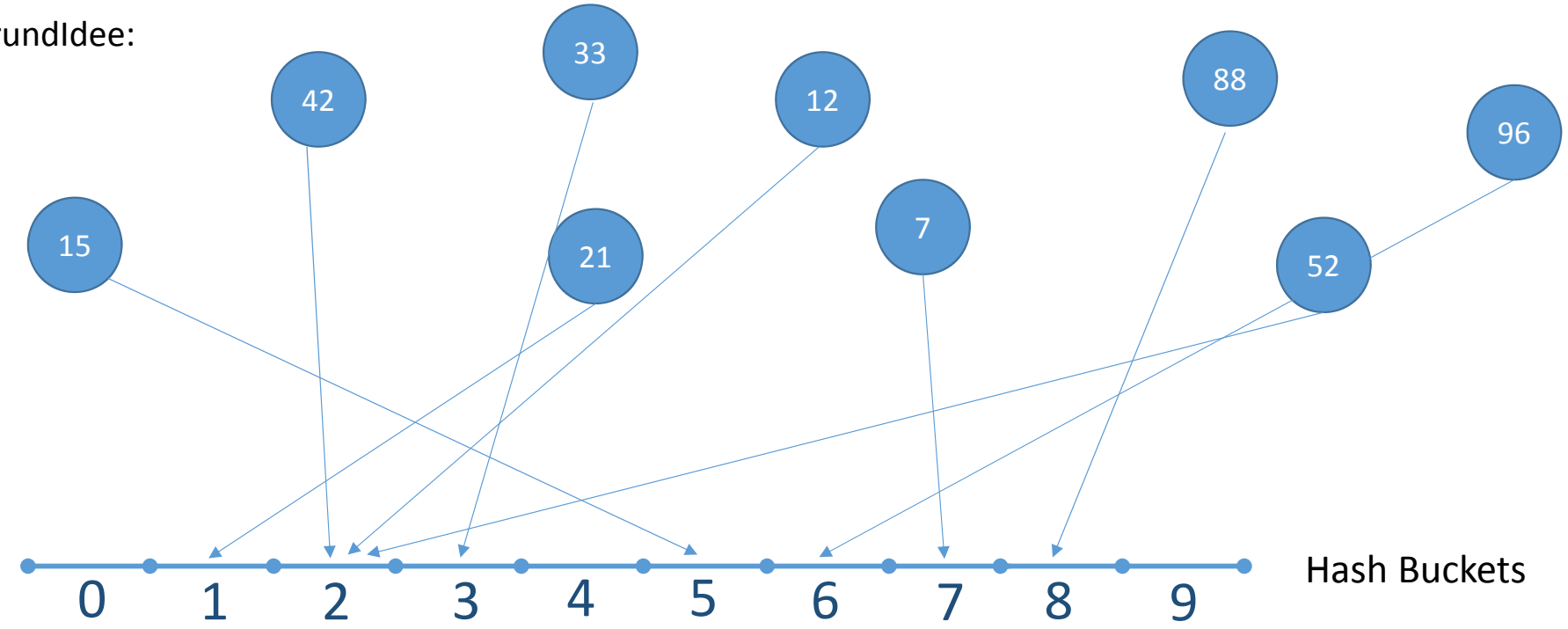
Hashfile

GrundIdee: Hashfunktion h(x) zur Abbildung von Schlüsseln x auf einen kleineren Wertebereich.
Bsp: h(x)= x mod 10. Wertebereich [0,9].



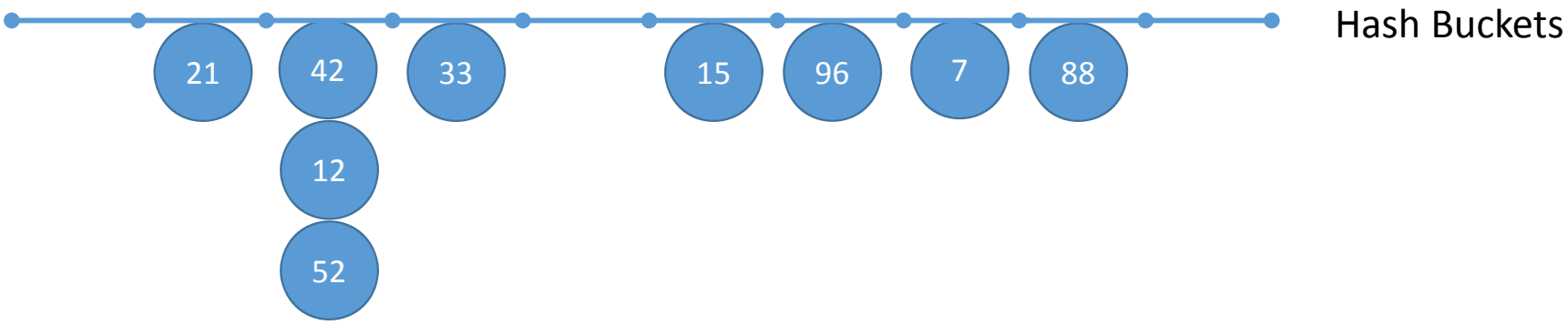
Hashfile

GrundIdee:



Hashfile

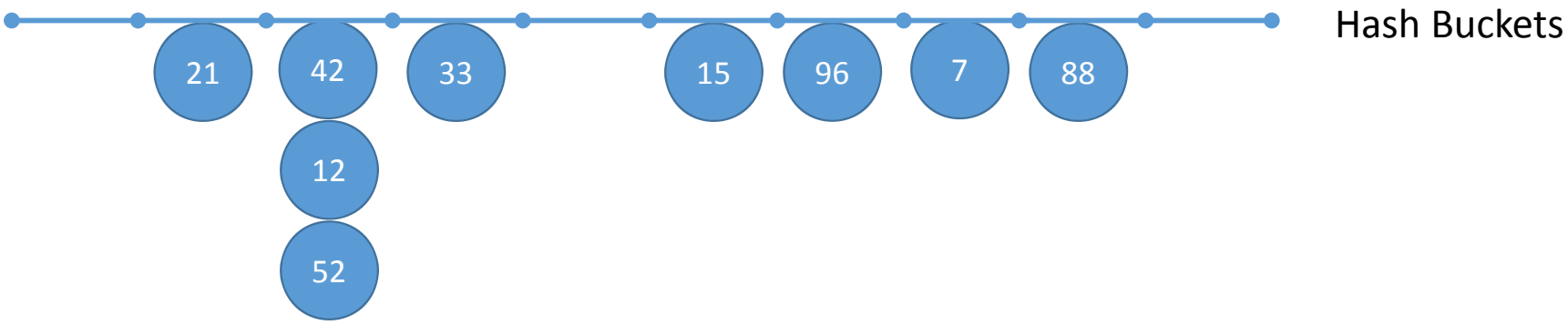
GrundIdee:



Hashfile

GrundIdee: Zur Suche nach einem Schlüssel x, muss nur der Bereich h(x) betrachtet werden.

Jeder Schlüssel x muss (falls vorhanden) in Bucket h(x) zu finden sein – aber nicht jeder Schlüssel in Bucket h(x) ist der gesuchte Schlüssel x.



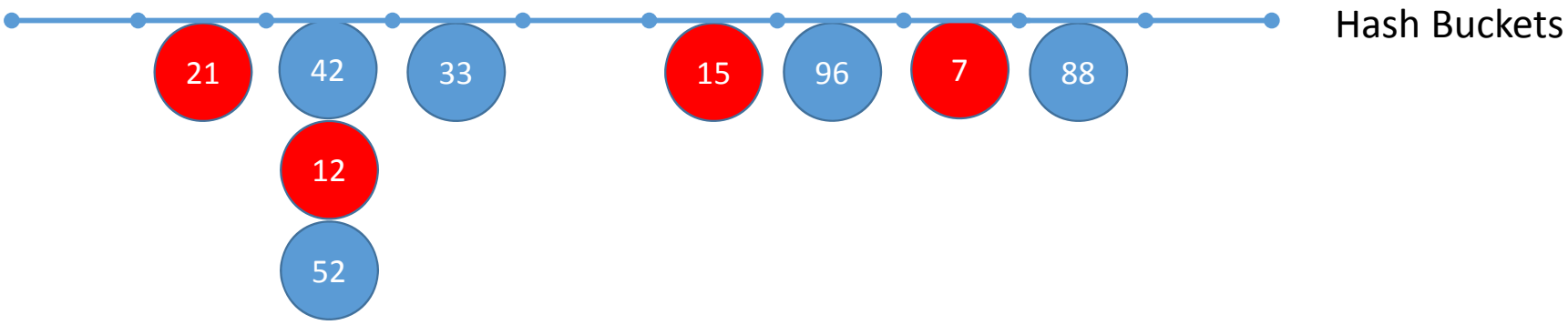
Anzahl der Buckets:

- Zu wenige Buckets
  - Viele Tupel im selben Bucket
  - Viele Überlaufseiten
  - Im schlimmsten Fall linearer Scan
- Zu viele Buckets
  - In den meisten Buckets nur höchstens ein Tupel
  - Speicherplatzverschwendung – eine Seite pro Tupel

Anzahl der Buckets:

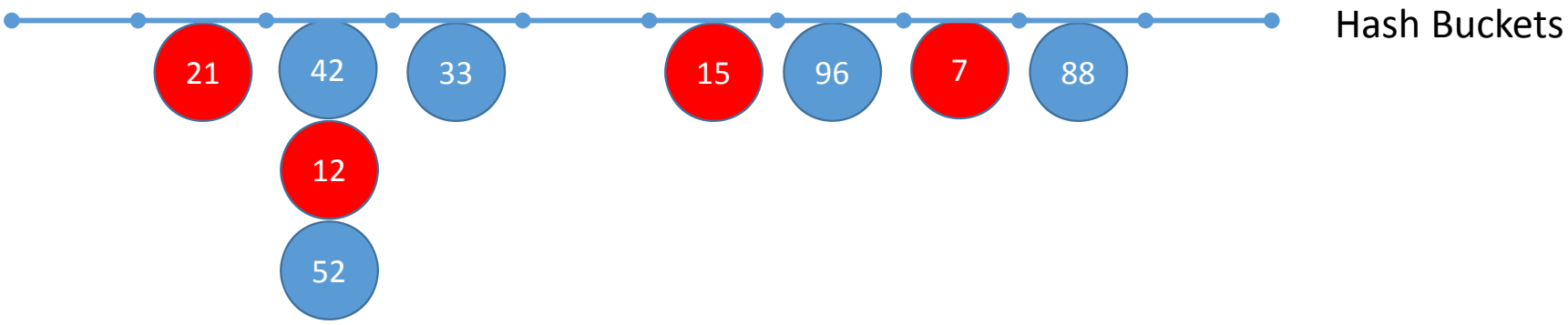
- Zu wenige Buckets
  - Viele Tupel im selben Bucket
  - Viele Überlaufseiten
  - Im schlimmsten Fall linearer Scan
- Zu viele Buckets
  - In den meisten Buckets nur höchstens ein Tupel
  - Speicherplatzverschwendung – eine Seite pro Tupel
- Adaptive Anzahl Buckets
  - desto mehr Daten, desto mehr Buckets
  - Optimalen Belegungsfaktor erreichen
  - Änderung der Hashfunktion falls Belegungsfaktor stark abweicht

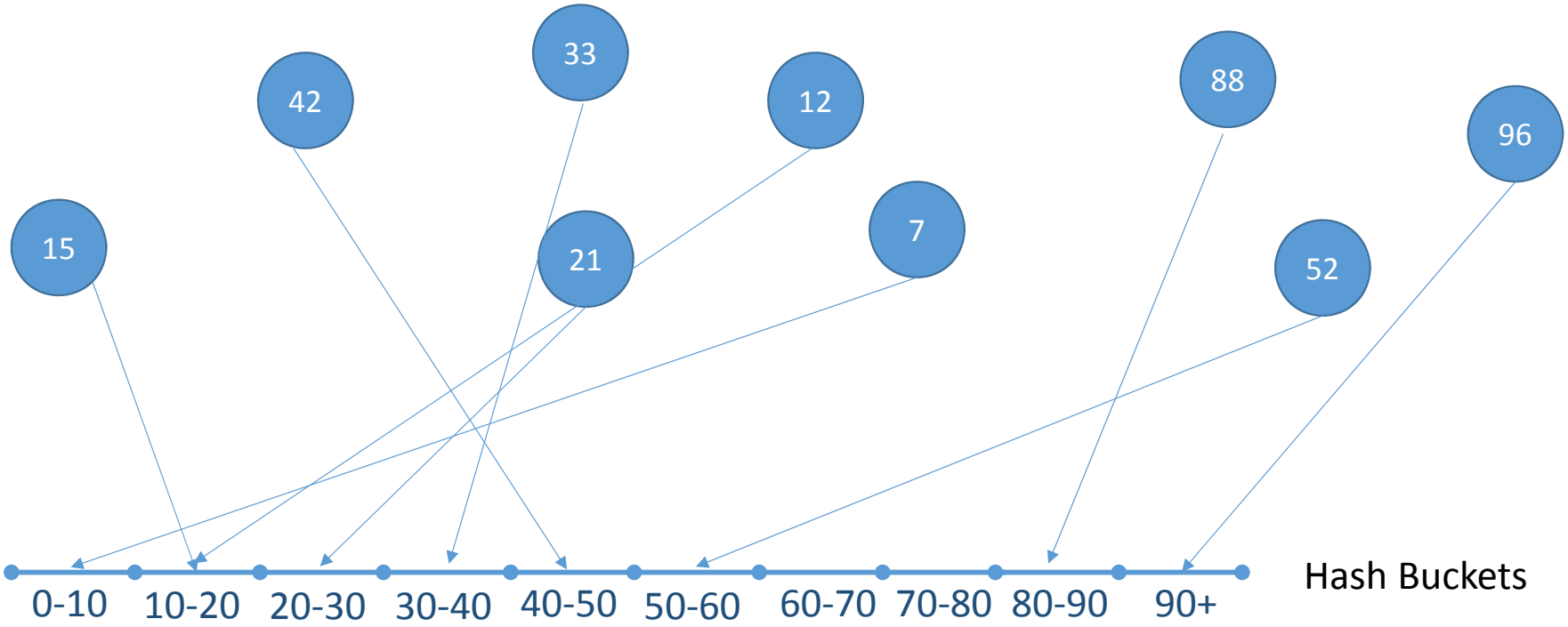
Ordnungserhaltendes Hashing - Grundidee:
Effiziente Unterstützung von Bereichsanfragen.
Gib mir alle Tupel mit Schlüssel im Interval [5,25].



Ordnungserhaltendes Hashing - Grundidee:
Effiziente Unterstützung von Bereichsanfragen.
Gib mir alle Tupel mit Schlüssel im Interval [5,25].

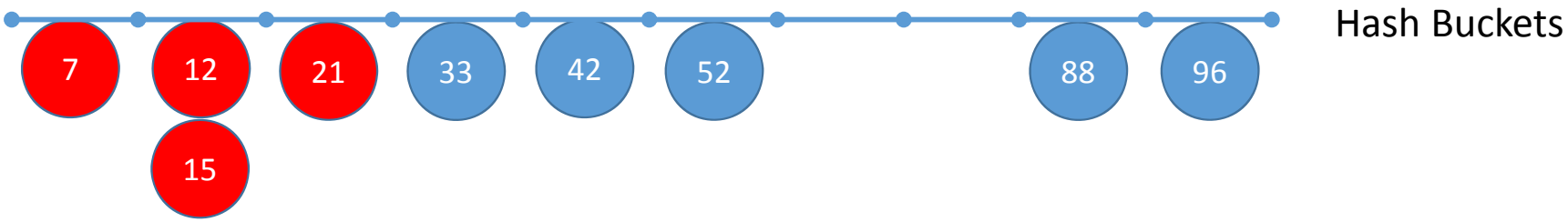
-Alle Buckets müssen durchsucht werden.
Selbe Performance wie linearer Scan




33
42
12
88
96
15
21
7
52
0-10
10-20
20-30
30-40
40-50
50-60
60-70
70-80
80-90
90+
Hash Buckets

Ordnungserhaltendes Hashing - Grundidee:
Effiziente Unterstützung von Bereichsanfragen.
Gib mir alle Tupel mit Schlüssel im Interval [5,25].

- Nur relevante Buckets müssen durchsucht werden.



Ordnungserhaltendes Hashing - Grundidee:
Effiziente Unterstützung von Bereichsanfragen.
Gib mir alle Tupel mit Schlüssel im Interval [5,25].

- Nur relevante Buckets müssen durchsucht werden.

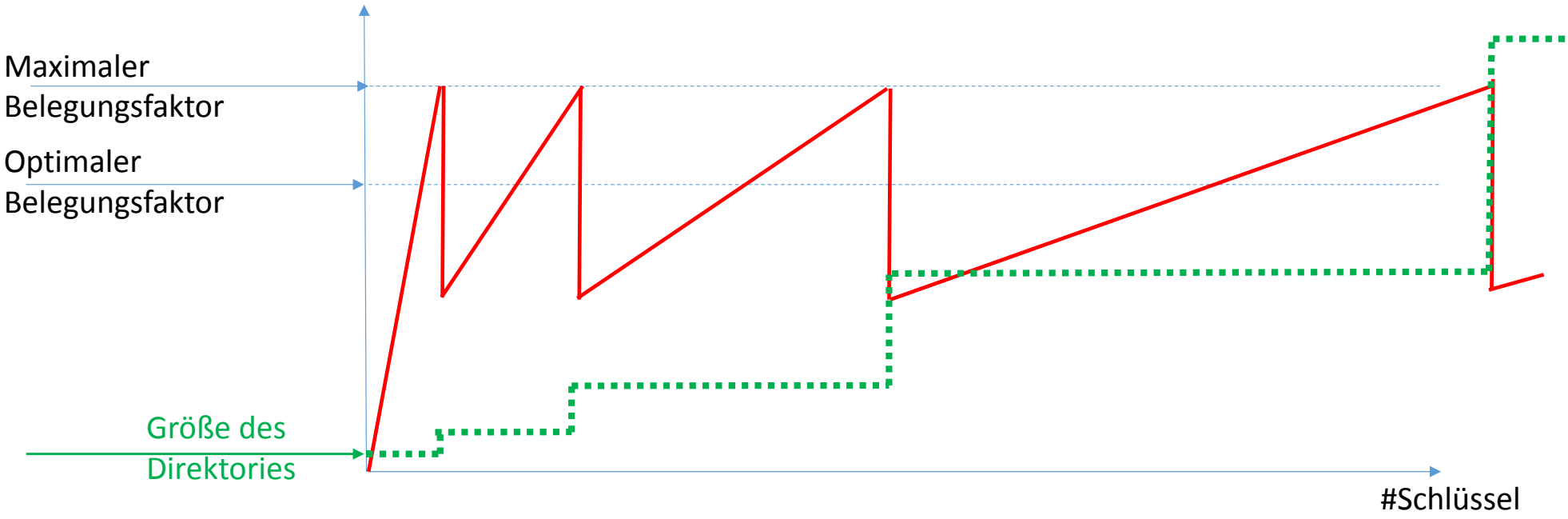
- Praxis: Transformation des Wertebereichs der Schlüssel auf [0,1].
  - Vorteil: Binärdarstellung gibt an, in welcher Hälfte, welchem Viertel, Achtel etc. des Datenraums sich der Schlüssel befindet.
  - Bsp: Wert 33 auf dem Bereich [0,100] wird transformiert auf den Wert $0.33 = 0.010\ldots_2$
    .010 zu interpretieren als
    „linke Hälfte“ ([0,0.5])
    davon „rechte Hälfte“ ([0.25,0.5])
    davon „linke Hälfte“ ([0.25,0.375])

Lineares Hashing - Motivation:

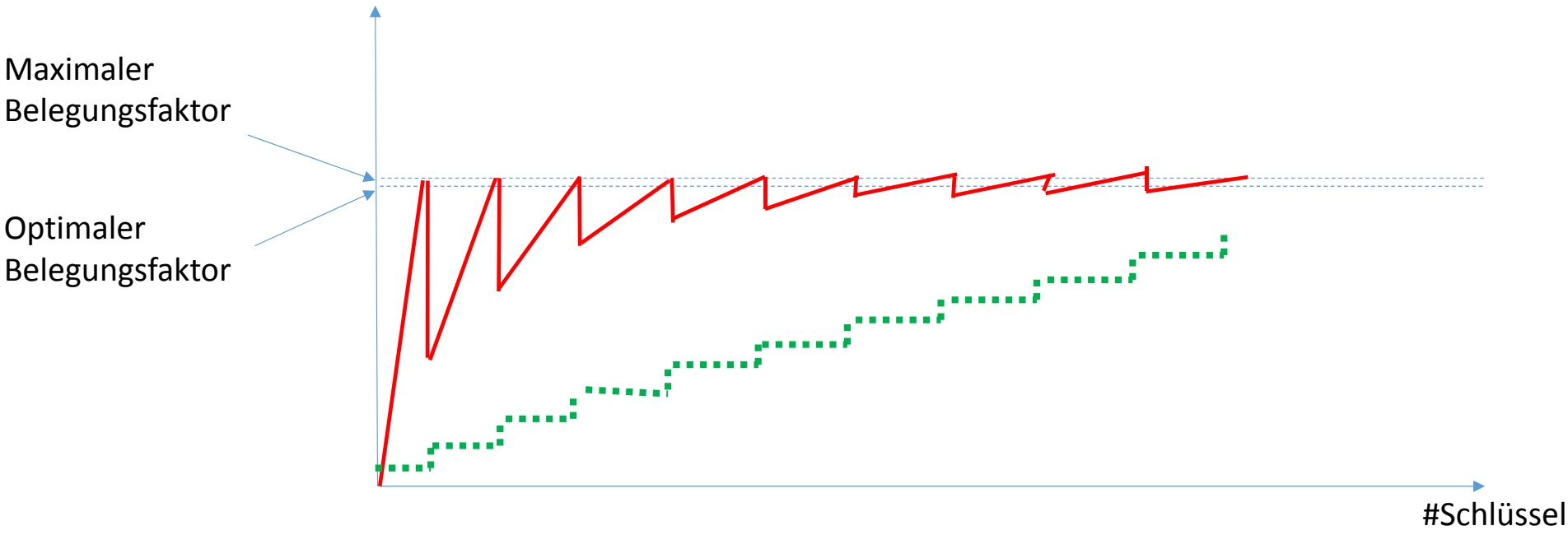
-Vorher erwähnt: Das Hash-Directory soll sich dynamisch an die Datenmenge anpassen.

-Überlicherweise wird das Hash-Directory verdoppelt wenn ein vorgegebener Belegungsfaktor überschritten wird.

- Beispiel: Wechsel von Hashfunktion (x mod 5) auf (x mod 10). Dadurch 10 statt 5 Buckets.
- Problem: Grosse Schwankung um den optimalen Belegungsfaktor



Lineares Hashing - Idee:
- Anstatt das Hash-Directory in exponentiell größer werdenden Schritten exponentiell zu verdoppeln, das Directory linear wachsen zu lassen.



Partielle Erweiterungen – Idee

Ähnliche Idee wie beim Linearen Hashing.

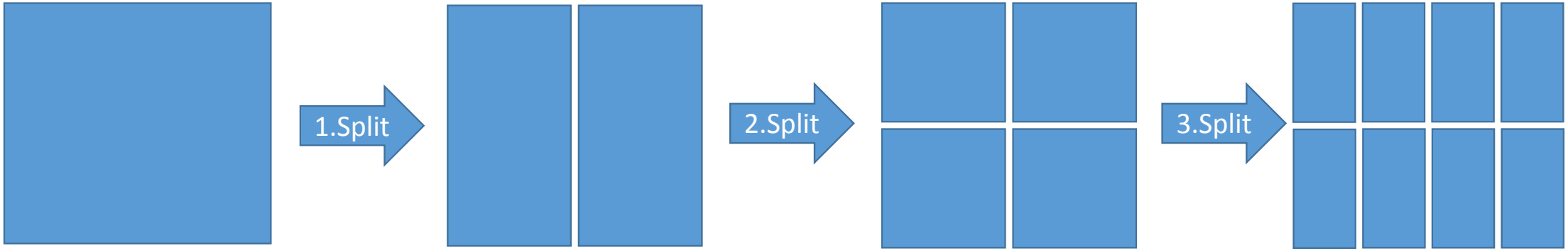
- Statt zu verhindern dass sich das ganze Directory verdoppelt, wird verhindet dass sich ganze Zellen verdoppeln.
- Statt ein Bucket auf zwei Buckets aufzuteilen, werden zwei Buckets auf drei Buckets aufgeteilt (1. Partielle Expansion), und beim nächsten mal diese drei Buckets auf vier Buckets aufgeteilt (2. Partielle Expansion).

Motivation: Vermindern des Effekts dass zwischen zwei Verdopplungen des Directories manche Zellen doppelt soviel Wertebereich abdecken wie andere.

Mehrdimensionales Hashing

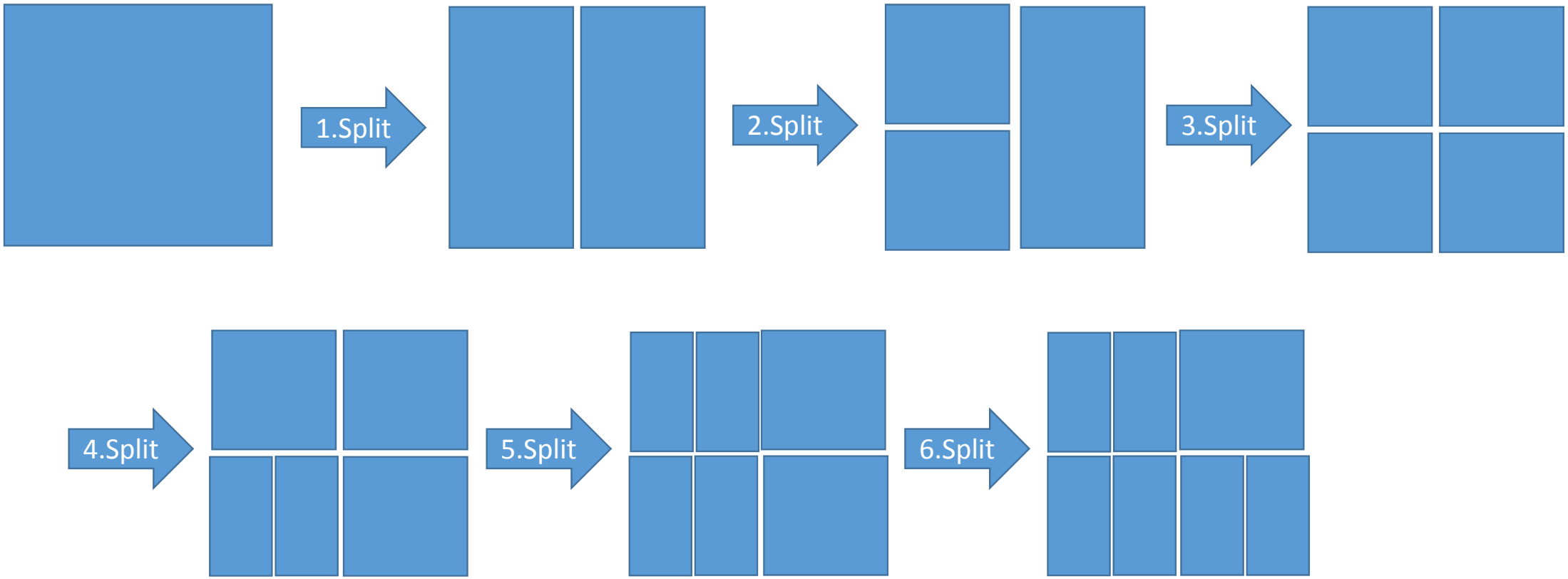
- Bisher: Eindimensionale (nicht-zusammengesetzte Schlüssel)
- Problem beim mehrdimensionalen (zusammengesetzten) Schlüssel:
  - Wie kann man mehrdimensionale Schlüssel sortieren? (für Ordnungserhaltung)
- Sortierung jeder Dimension einzeln → Mehrdimensionales Grid

- Im einfachsten Fall (nicht linear, nicht ordnungserhaltend, ohne part. Erw.):



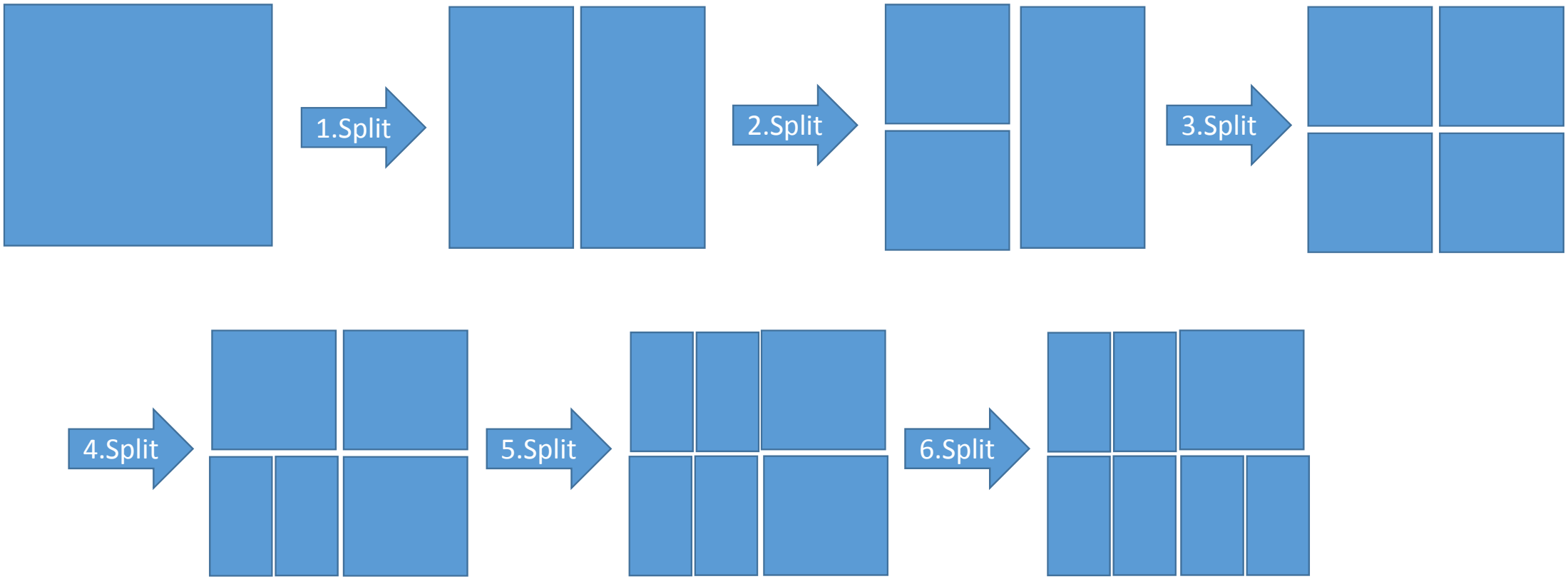
Mehrdimensionales Hashing

- Mehrdimensionales lineares hashing, analog zum eindimensionalen linearen hashing



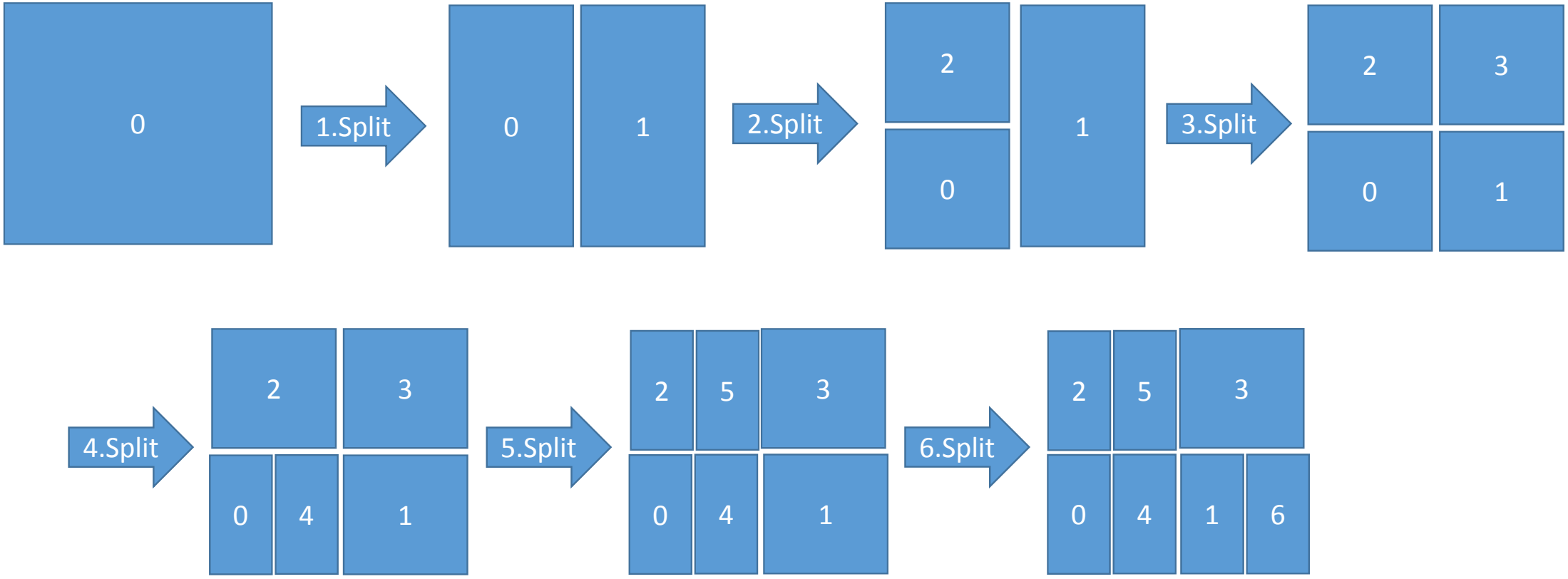
Mehrdimensionales Hashing

- Mehrdimensionales ordnungserhaltendes lineares hashing
- Jede Dimension Orndungserhalten.
- Im Beispiel bereits Ordngungserhaltend.



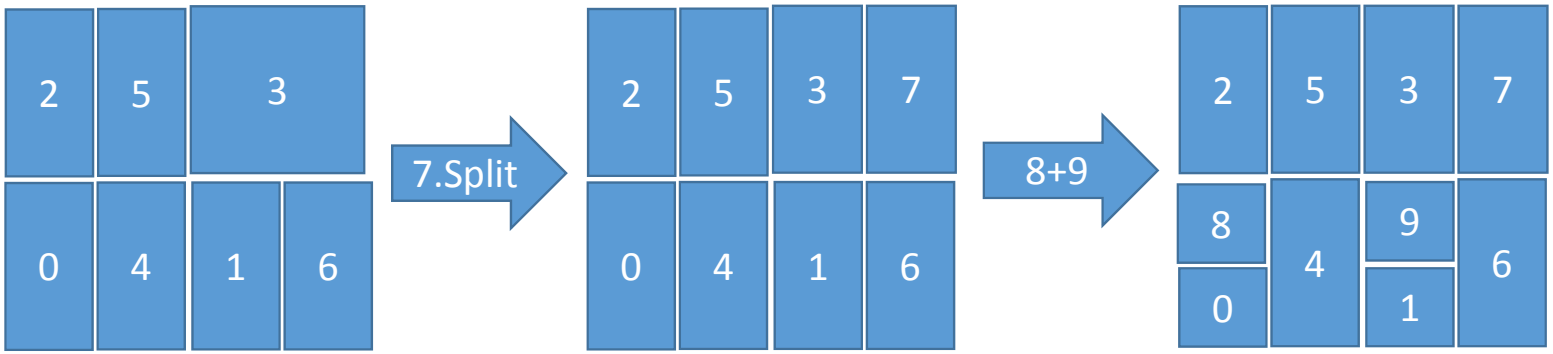
Mehrdimensionales Hashing:

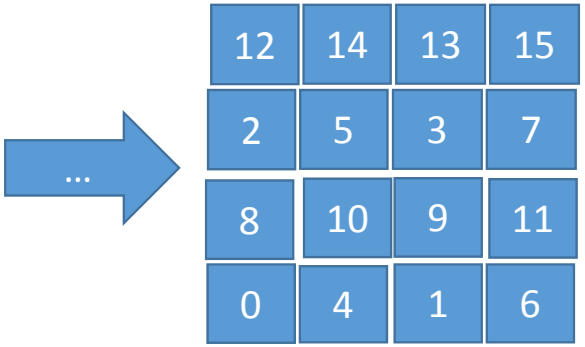
- Seitenaddressen ergeben sich natürlich durch die Splits
- Splitreihenfolge von „MOLPHE“
  - Dimensionen abwechselnd (abwechselnd „Zeilenweise“ / „Spaltenweise“)
  - Innerhalb einer Dimension Reihenfolge laut eindim. Ordn. Hashfkt.



Mehrdimensionales Hashing:

- Seitenaddressen ergeben sich natürlich durch die Splits
- Splitreihenfolge von „MOLPHE“
  - Dimensionen abwechselnd (abwechselnd „Zeilenweise“ / „Spaltenweise“)
  - Innerhalb einer Dimension Reihenfolge laut eindim. Ordn. Hashfkt.





(s. Skript)

Mehrdimensionales Hashing:

Punktanfrage: Mittels Addressfunktion

Bsp.: Addresse von Punkt (0.3,0.6)

Umwandlung der einzelnen Koordinaten mittels eindimensionaler ordnungserhaltender Hashfunktion. (0.3,0.6) -> (2,1)

Einsetzen in mehrdimensionale Addressfunktion G(2,1,9). (9=Anzahl Buckets)

Funktion G(.,.) prüft wie oft die einzelnen Dimensionen sich schon verdoppelt wurden – und prüft welche Zellen dann schon in welchen Dimensionen gesplittet wurden.

G(.,.) abhängig von:

- Dimensionalität
- Splitreihenfolge der Dimensionen
- Splitreihenfolge innerhalb einer Dimension