2 Segment- und Seitenverwaltung

Overall goal of this layer: Mapping of pages and segments to blocks

External Storage

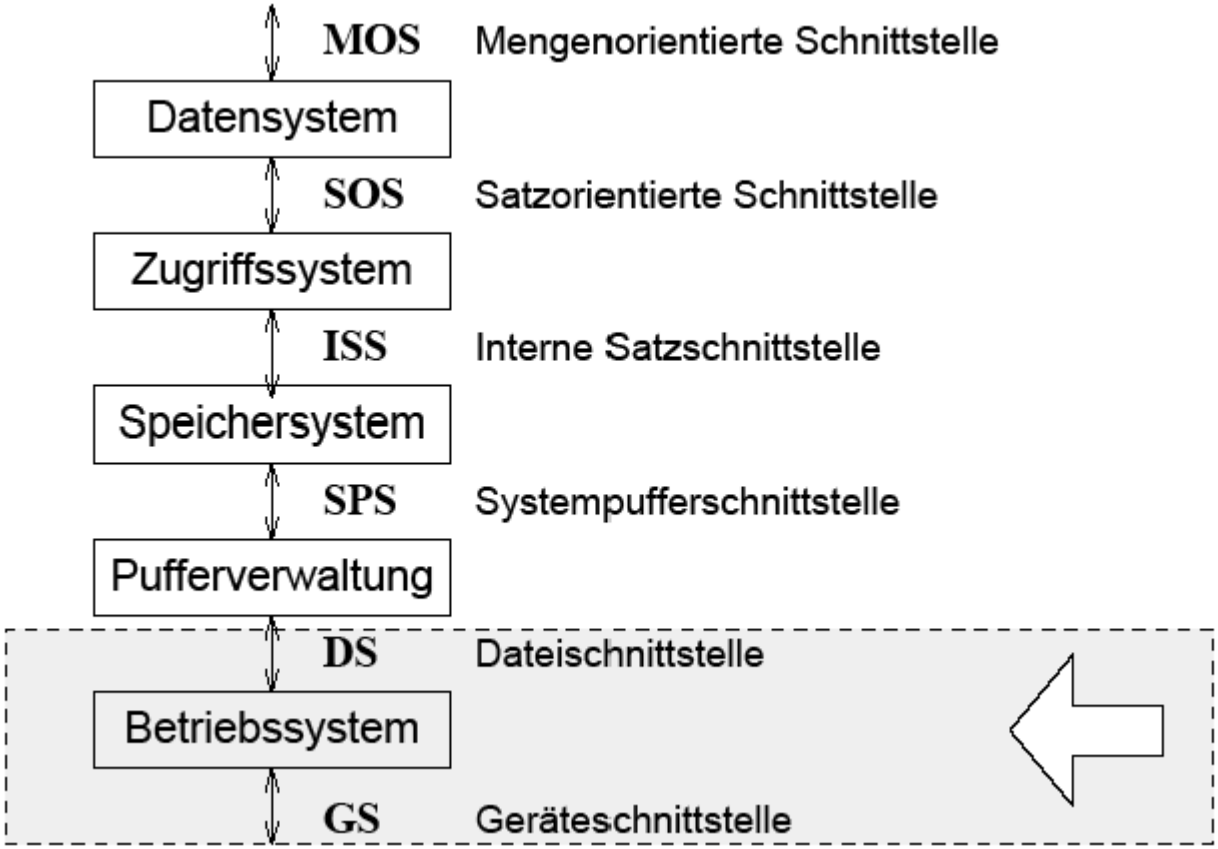
- storage hierarchy
- device interface

Page Adressing Schemes

- direct page adressing
- indirect page adressing

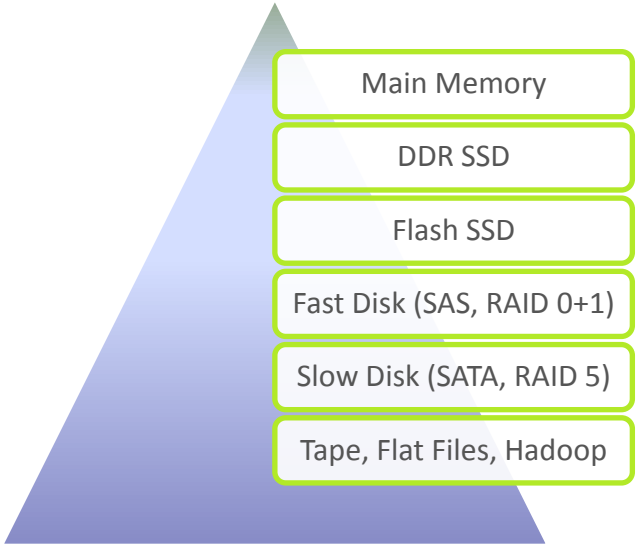
Data distribution

- partitioning and range fragmentation


MOS
Mengenorientierte Schnittstelle
Datensystem
SOS
Satzorientierte Schnittstelle
Zugriffssystem
ISS
Interne Satzschnittstelle
Speichersystem
SPS
Systempufferschnittstelle
Pufferverwaltung
DS
Dateischnittstelle
Betriebssystem
GS
Geräteschnittstelle

Anforderungen

- Verwaltung externer Speichermedien / Unterstützung von Speicherhierarchien
- Adressierung physischer Blöcke
- Kontrolle des Datentransportes vom/zum Hauptspeicher
- Fehlertoleranzmaßnahmen (RAID-Systeme, ...)



Speicherhierarchie

- extrem schneller Prozessor mit Registern
- sehr schneller Cache-Speicher
- schneller Hauptspeicher
- langsamer Sekundärspeicher mit wahlfreiem Zugriff sehr langsamer Nearline-Tertiärspeicher bei dem die Speichermedien automatisch bereitgestellt werden
- extrem langsamer Offline-Tertiärspeicher, bei dem die Speichermedien per Hand bereitgestellt werden (CD-R, CD-RW, DVD Magnetbänder etwa DLT (Digital Linear Tape))

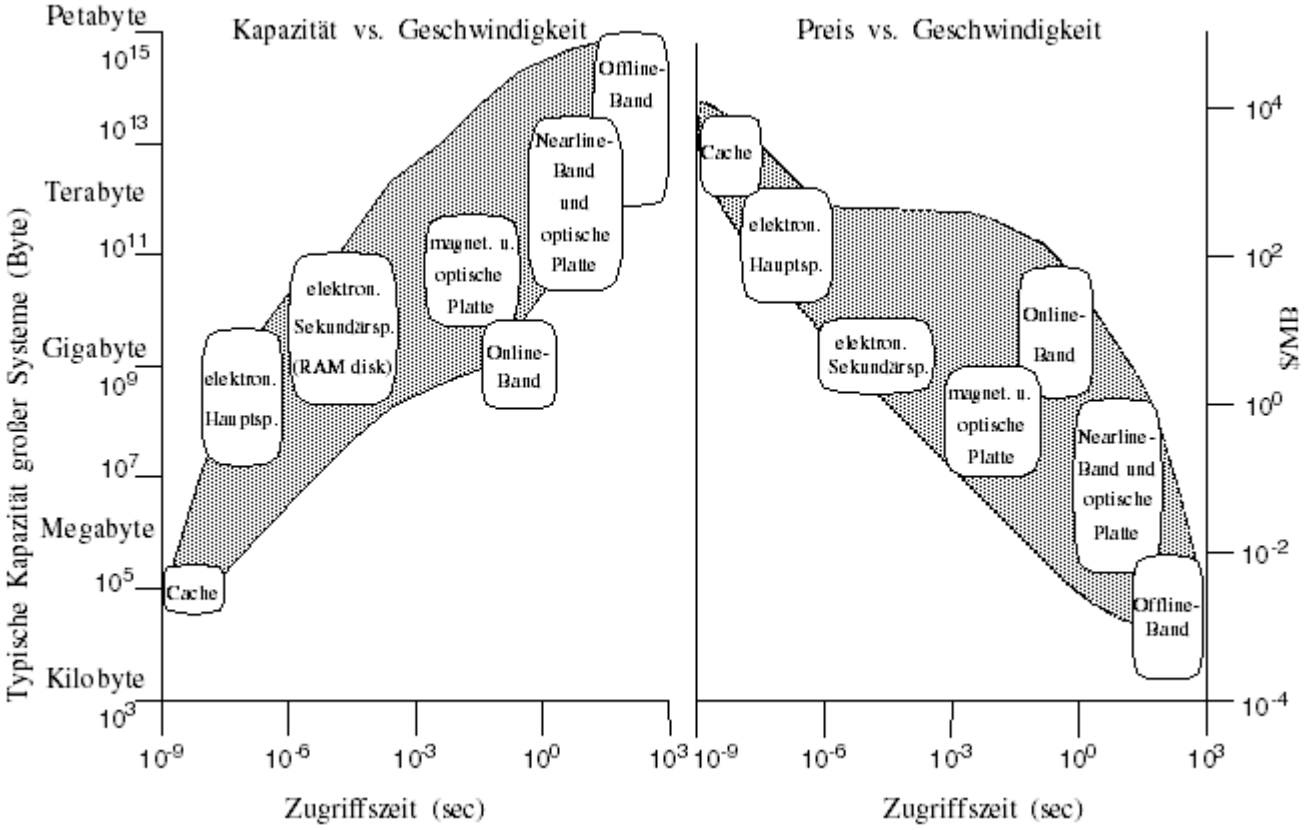
Zugriffslücke

- Unterschiede zwischen den Zugriffsgeschwindigkeiten auf die Daten vermindern
- Cache-Speicher speichern auf Ebene x Daten von Ebene x+1 zwischen:
  - Hauptspeicher-Cache: schnellere Halbleiterspeicher-Technologie für die Bereitstellung von Daten an Prozessor (Ebene 2 in der Speicherhierarchie)
  - Plattenspeicher-Cache im Hauptspeicher: Systempuffer
- Eigenschaften
  - Cache-Speicher: 6ns 4MB - 128MB
  - Hauptspeicher: 60ns 2GB - 64GB
  - Disk: 10ms 200GB – 1TB
  - Platten-Array: 10ms im TB-Bereich

Datenbankperspektive

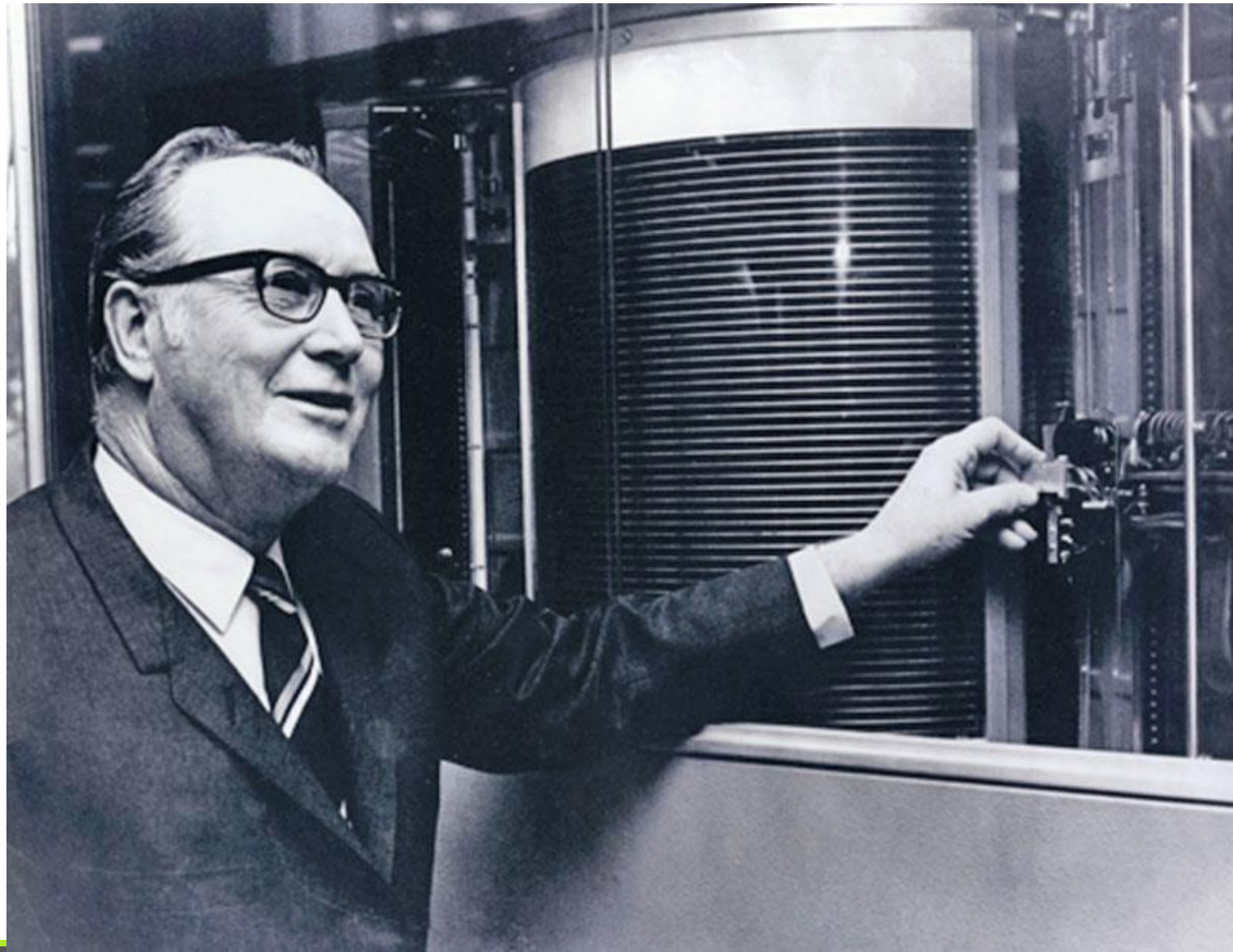
- Betriebssysteme realisieren einen Verzeichnisdienst zur Verwaltung von Dateien mit unstrukturiertem Inhalt
- Realisieren blockorientierte Zugriffe zum Lesen und Schreiben von Blöcken

Leistung versus Kosten unterschiedlicher Speichersysteme

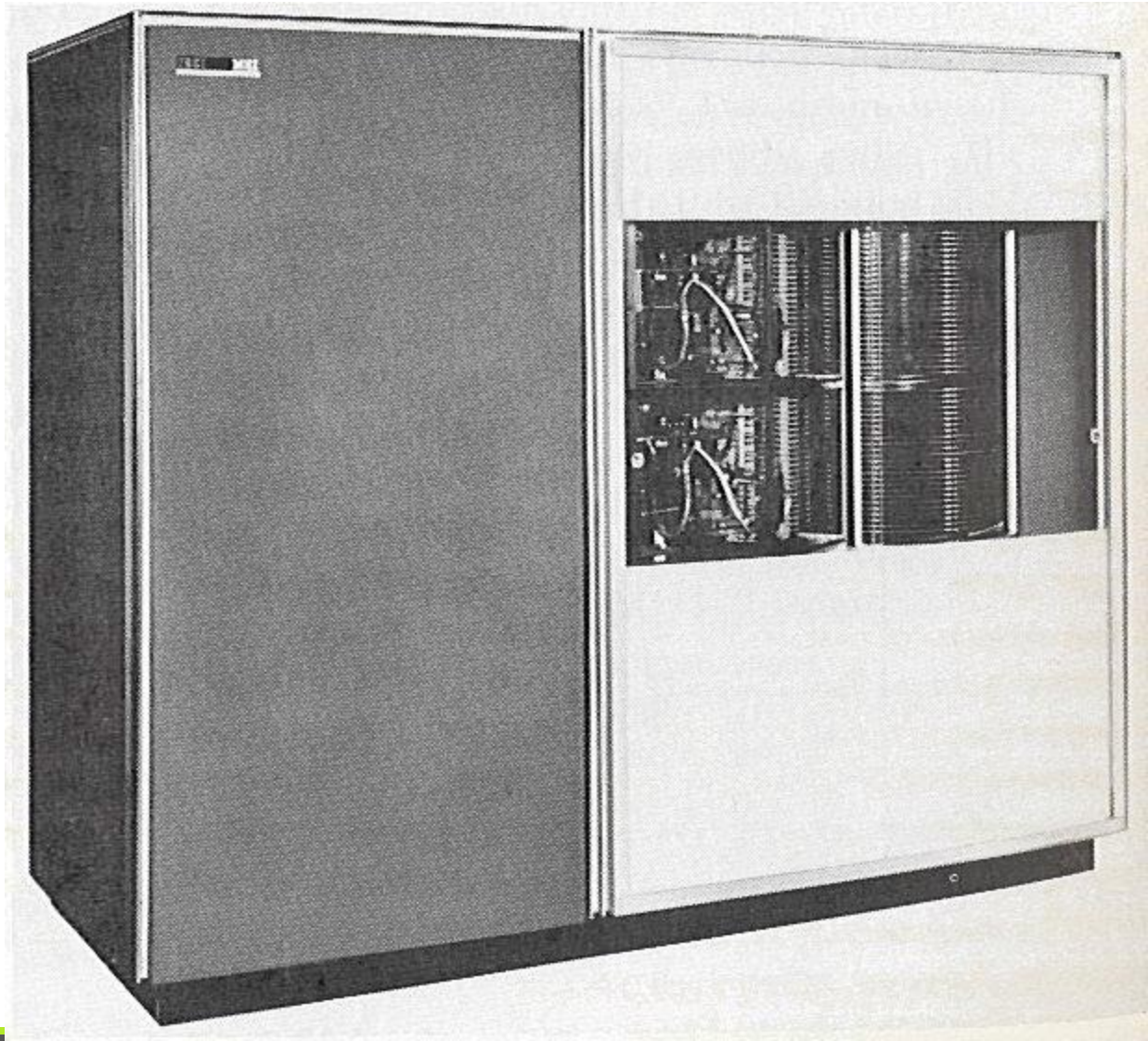


HDD Development

5MB HDD circa 1956



28MB HDD - 1961



The more that things change....

Redundant Array of Inexpensive Disks

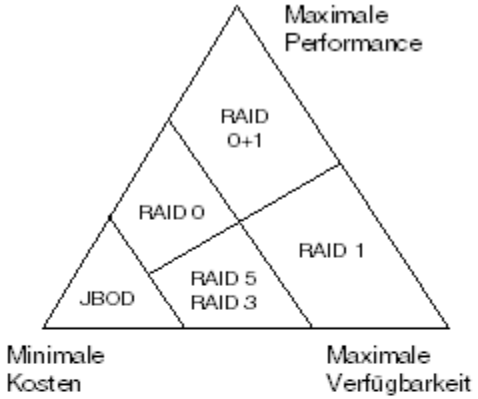
- Durchsatz: mehrere Disks in Parallelbetrieb -> Performanz
- Fehlertoleranz: redundante Daten auf mehreren Disks -> Redundanz

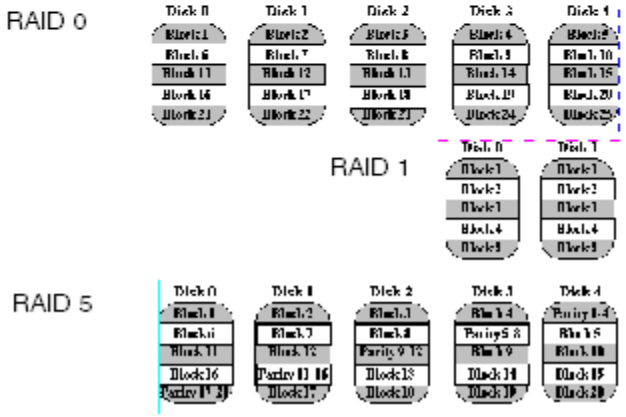
Entwurfsziele bei der Benutzung

- Maximierung der Anzahl von Disks, die parallel benutzt werden
- Minimierung der redundanten Datenmenge
- Minimierung des Overheads

Unterschiedliche Stufen (Auswahl)

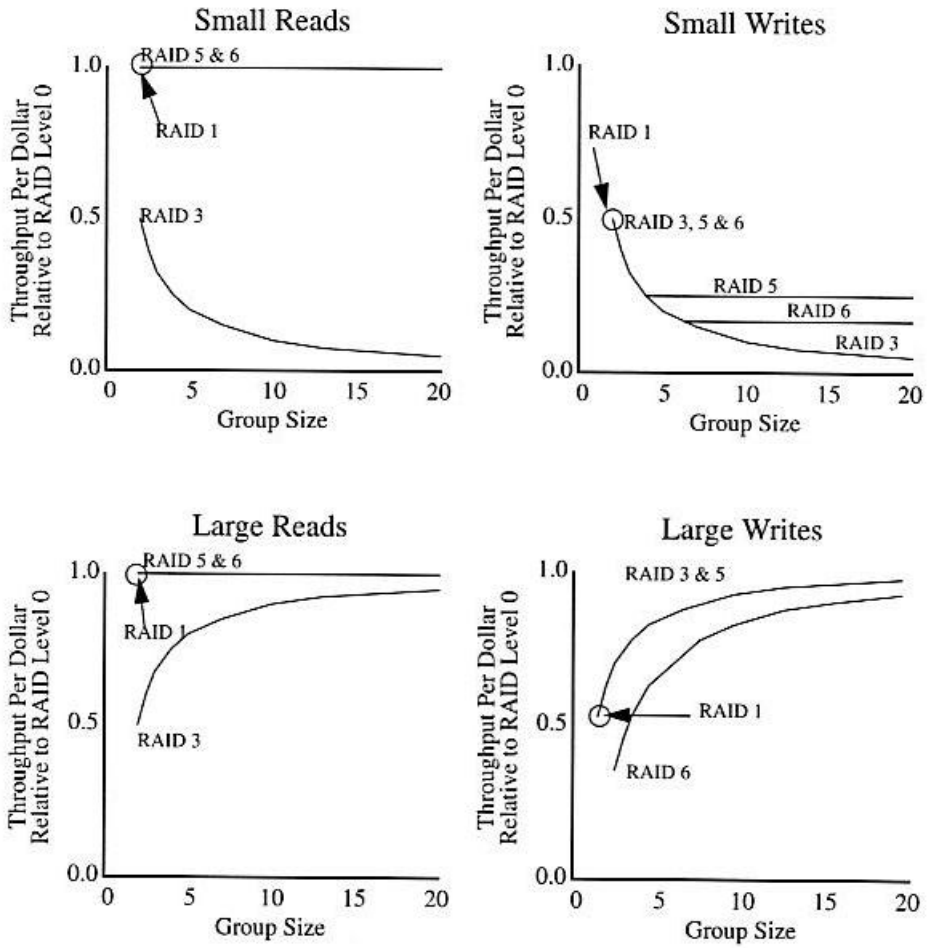
- RAID 0: Verteilung: nicht-redundant
- RAID 1: Spiegelung: keine Performanzverbesserung
- RAID 5: block-interleaved distributed parity

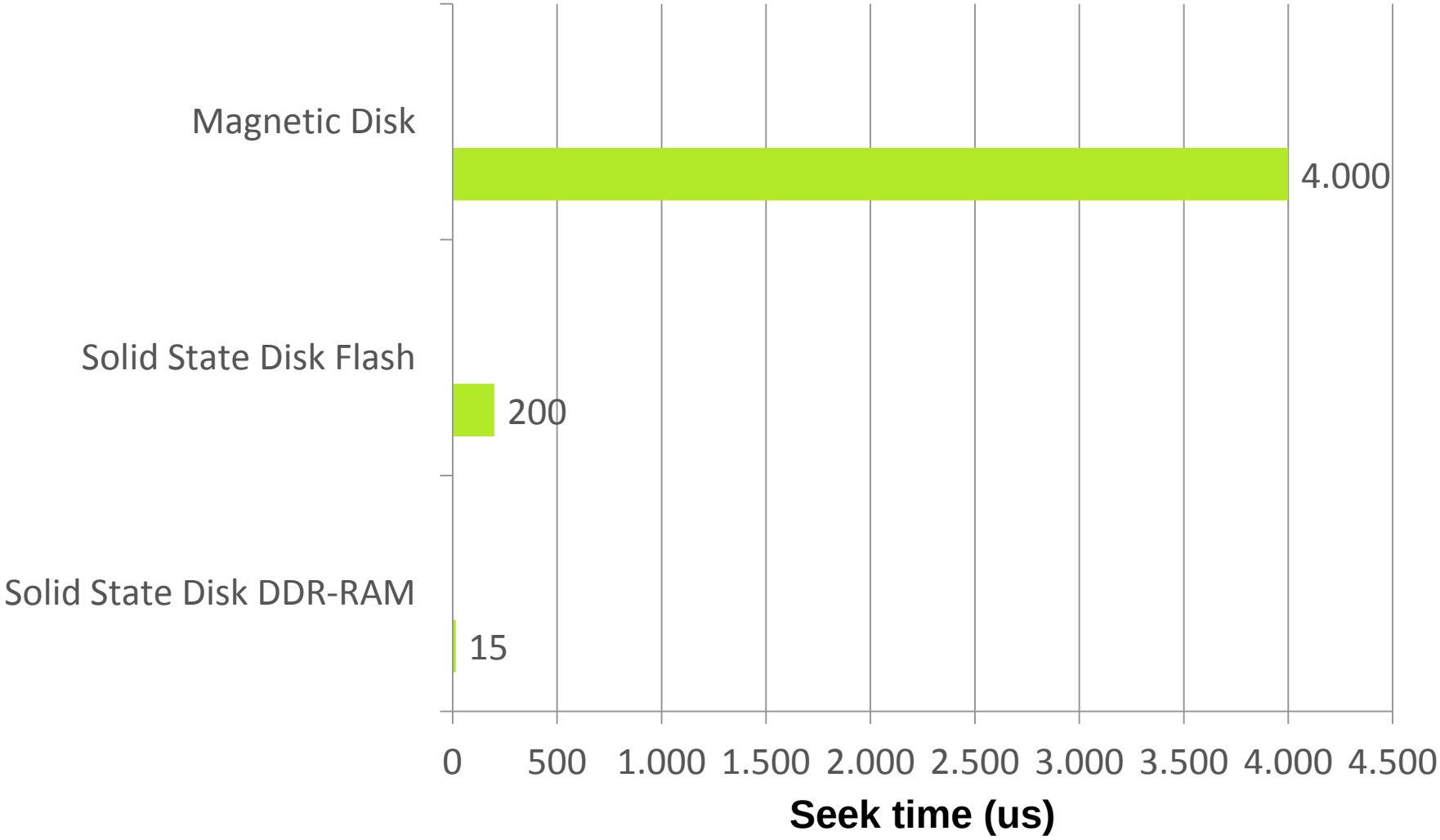


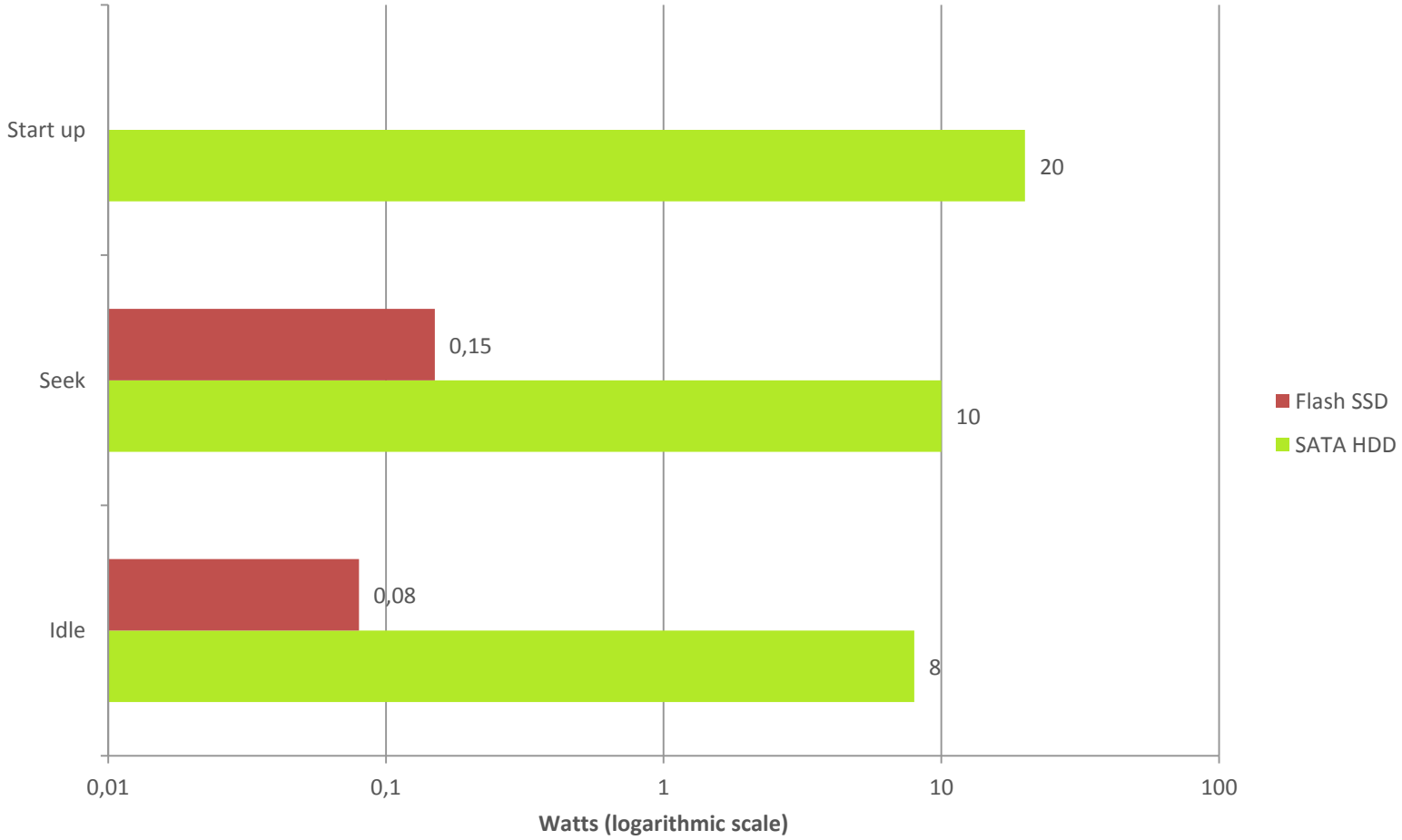


... aus Sicht des Datenbanksystems

- Realisierung einer virtuellen Disk
- keine Kenntnis/Kontrolle über Verteilungsstrategie
- Gefahr
  - Gegenläufige Optimierungskriterien (TablesSpaces)
  - Performanz- / Redundanzverhältnis beachten




Magnetic Disk
Solid State Disk Flash
Solid State Disk DDR-RAM
4.000
200
15
0
500
1.000
1.500
2.000
2.500
3.000
3.500
4.000
4.500
Seek time (us)


Start up
Seek
Idle
20
0,15
10
0,08
8
0,01
0,1
1
10
100
Watts (logarithmic scale)
Flash SSD
SATA HDD

	Virident TachIOn	FC HDDs
Sustained 4K R|W IOPS/Drive	200,000	200
Number of Drives for 200,000 IOPS	1	1000
Usable storage (%)	100%	10%
Virident TachIOn: 1000X IOPS for +50% price increment		
Racks required to support drives	0	2-4 Datacenter Racks
Power consumed	25W	18,000 W
IOPS/Watt	8000	11
Virident TachIOn: 1000X IOPS/TCO $		

www.virident.com

Storage Hierarchy:

- Cell: One (SLC) or Two (MLC) bits
- Page: Typically 4K
- Block: Typically 128-512K

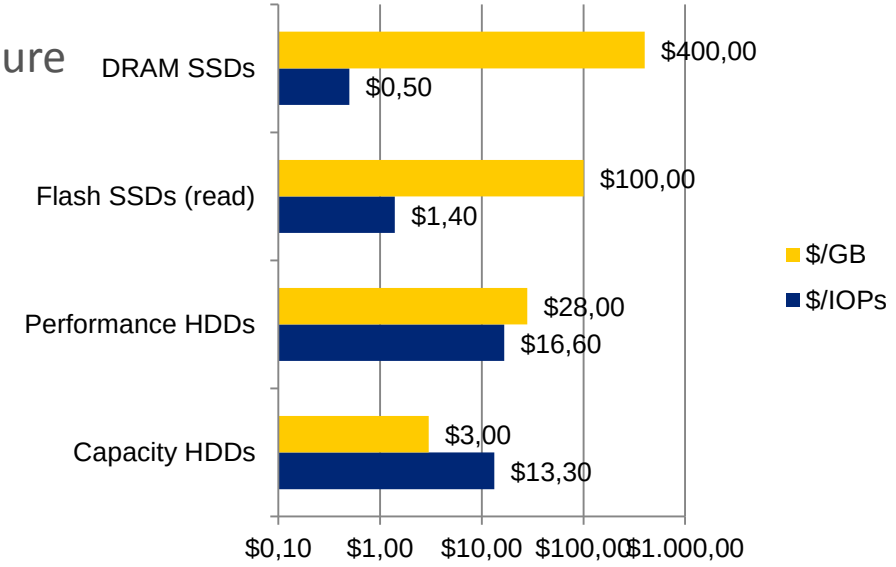
Writes:

- Read and first write require single page IO
- Overwriting a page requires an erase & overwrite of the block

Write endurance:

- 100,000 erase cycles for SLC before failure
- 5,000 – 10,000 erase cycles for MLC

Different Characteristcs:



I/O Ops per sec

- HDD: several 100 SSD: up to 785.000[1] and increasing

SSD I/O Queue

- SSDs (could) allow many parallel hardware queues[1]
  - Each thread could write concurrently
  - Not yet supported by OSes
- A read is (usually) more expensive than a write[2]
  - A write can be buffered in the SSD's internal RAM…
  - …while internal capacitors allow to program Flash chips later
  - A read (usually) requires to actually fetch data from the Flash chips

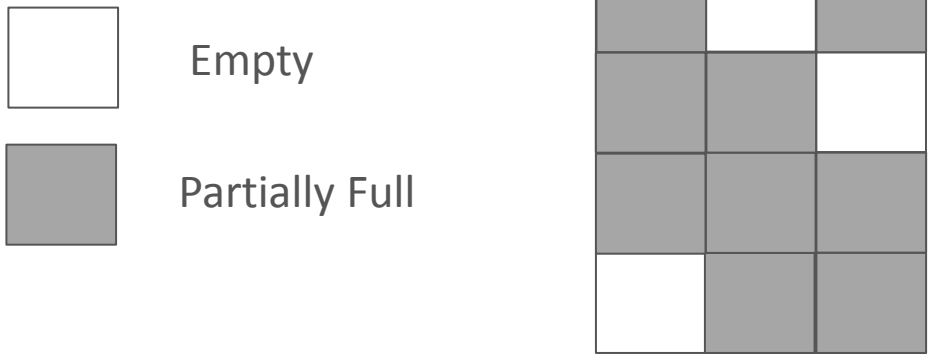
Direct I/O:

- Read from underlying **block device** block-wise (today typically 4K)
- Requires aligned memory
- (Should be) faster than Buffered I/O

Buffered I/O

- Memory alignment and block-wise I/O hidden by OS (system calls)
- → Additional OS calls required
- may be re-used for byte-addressable persistent storage like PCM! (e.g. PMFS[3])

[1] "Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems" (SYSTOR'13)
[2] "The necessary Death of the Block Device Interface" (CIDR'13)
[3] linux-pmfs (persistent memory file system) https://github.com/linux-pmfs



All Blocks empty

- Write time=250 us

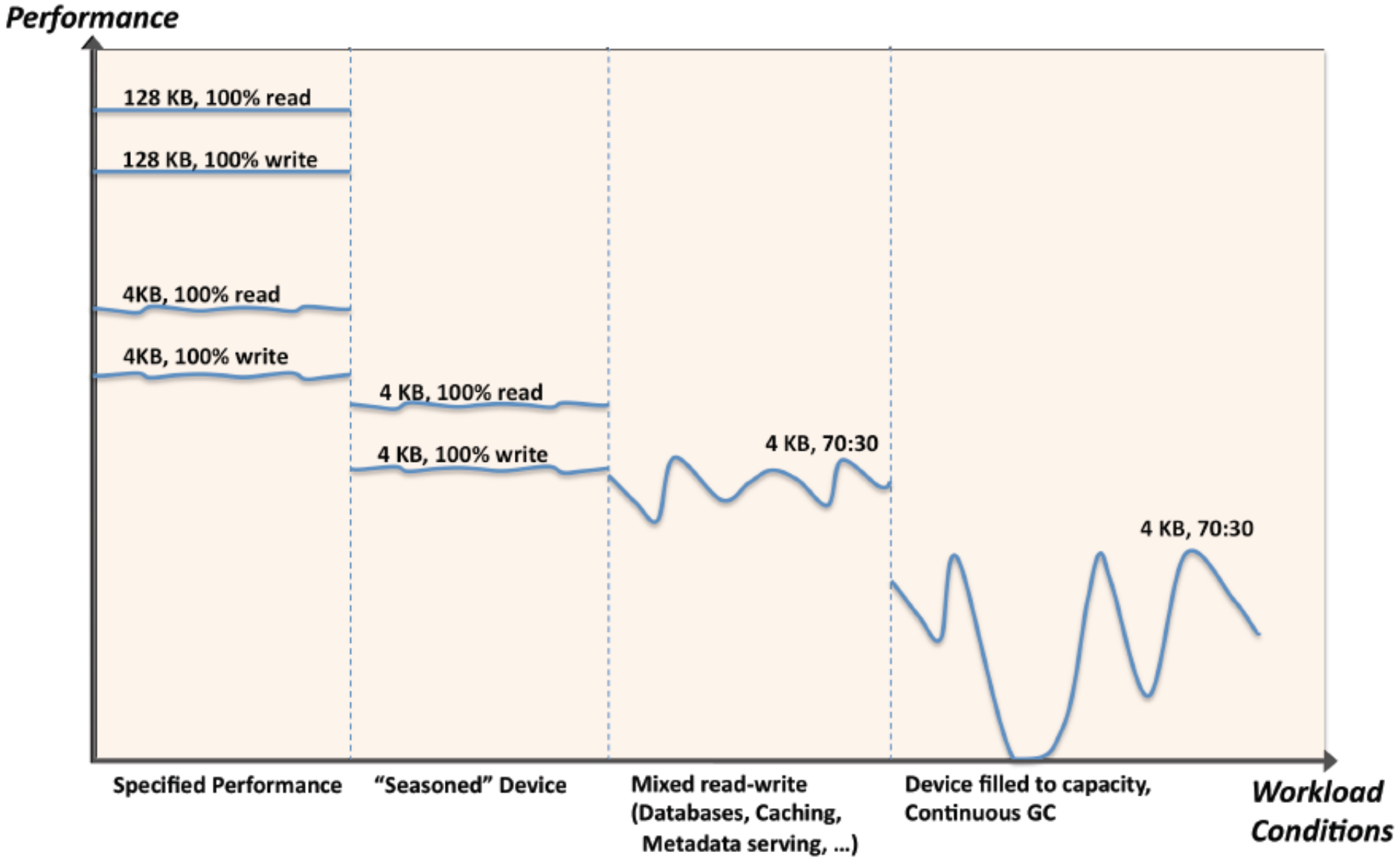
25% part full

- Write time= (¾ * 250 us + 1/4 * 2000 us) = 687 us

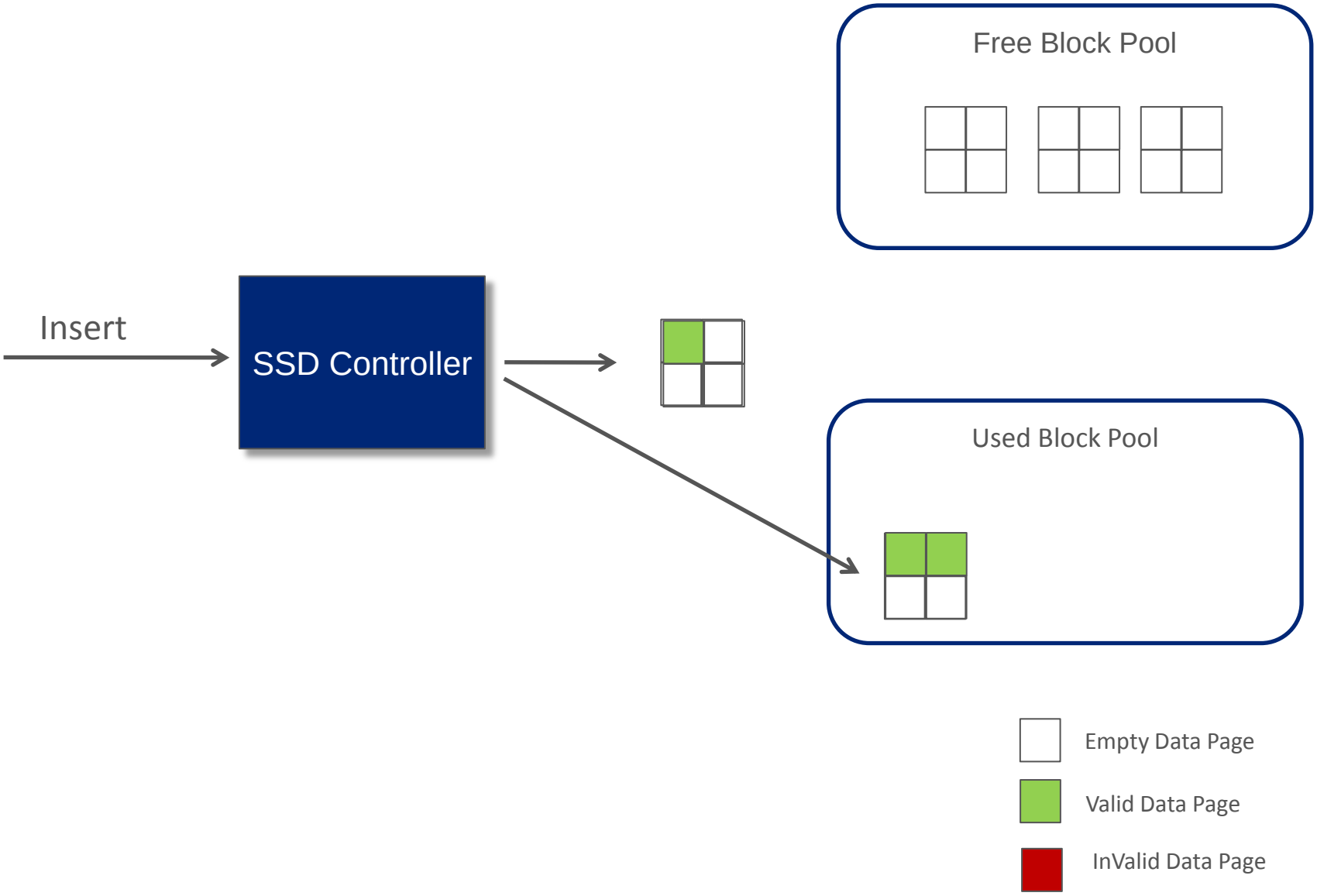
75% part full

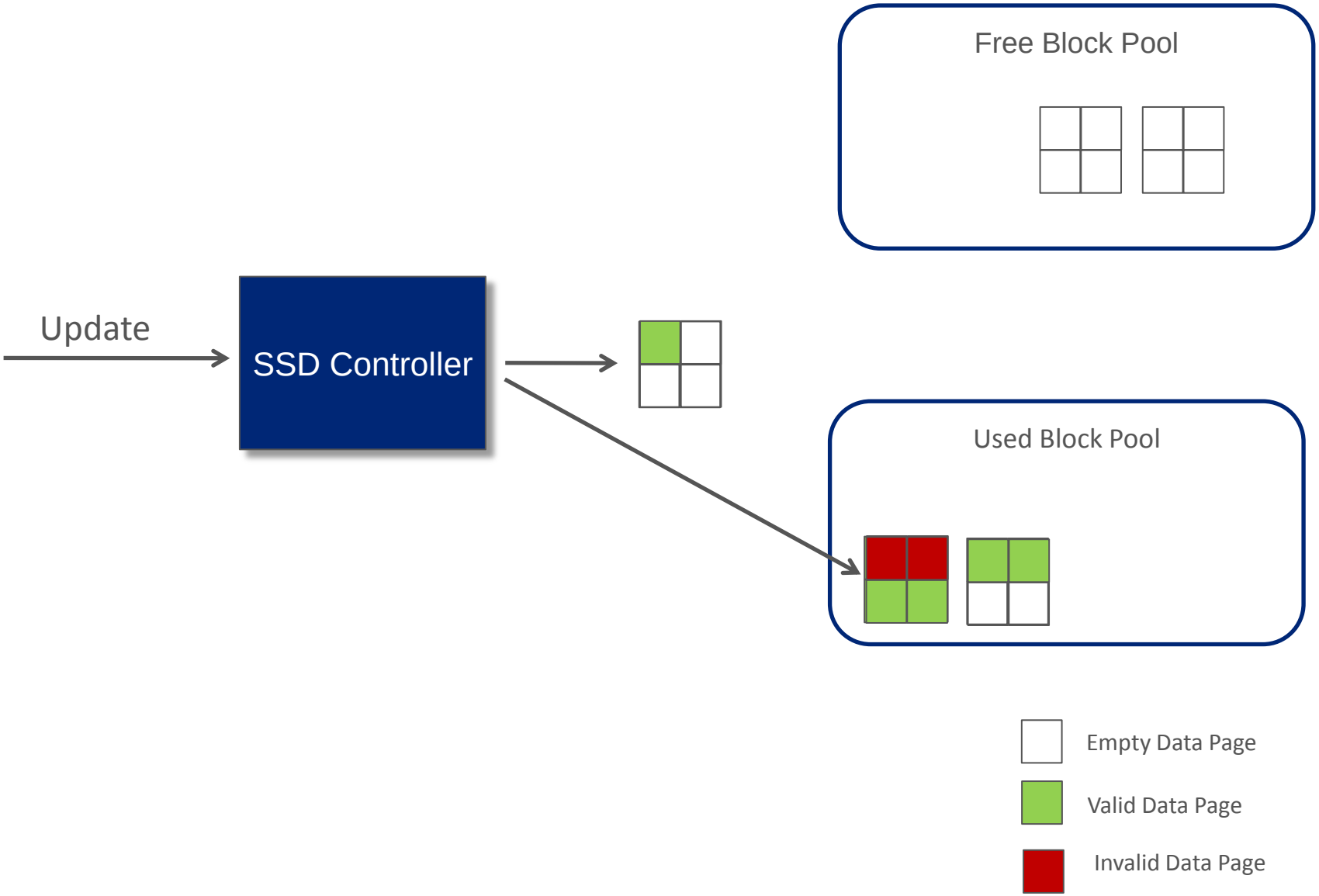
- Write time = (¼ * 250 us + ¾ * 2000 us) = 1562 us

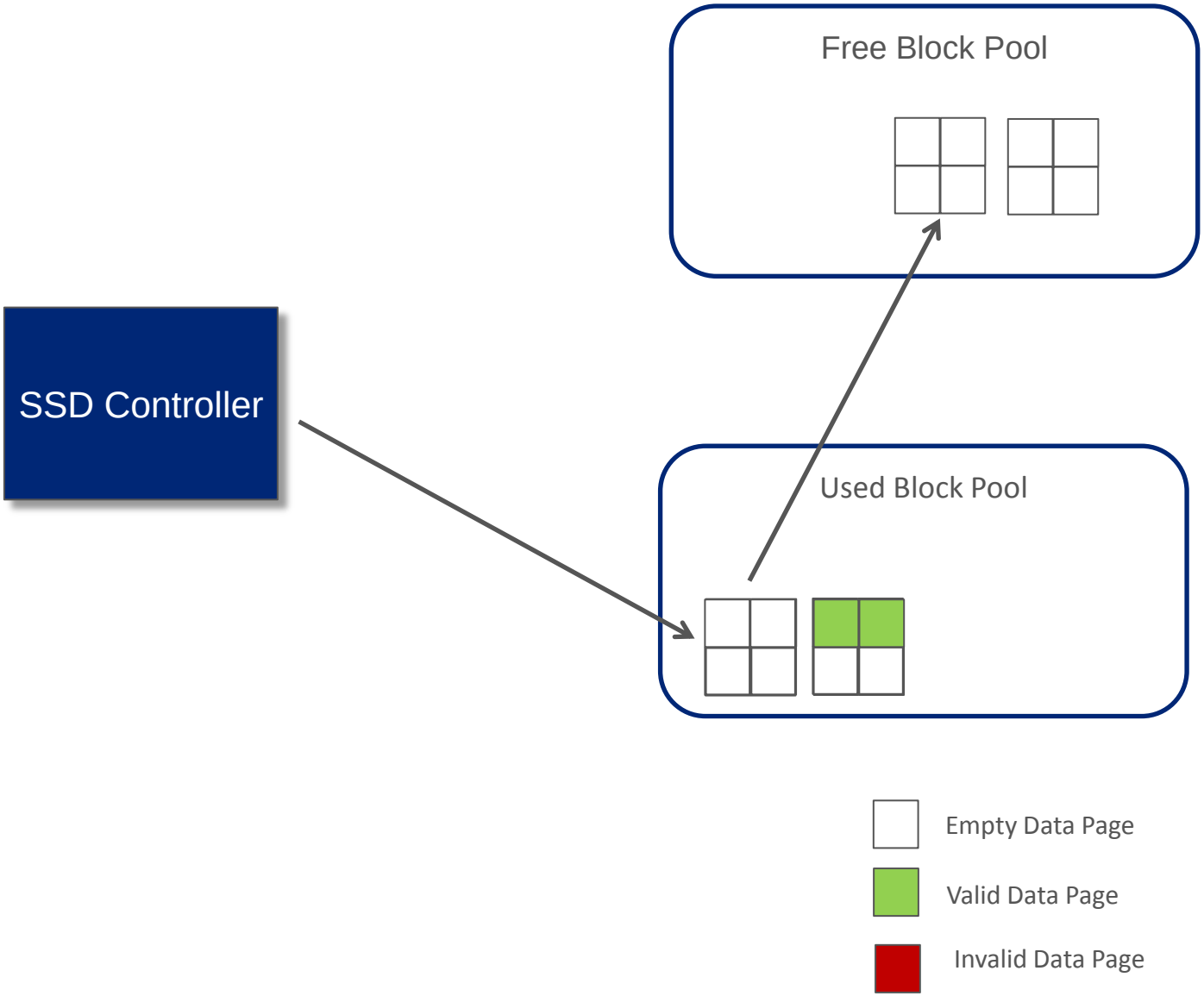
Garbage collection works to avoid this degradation



www.virident.com


Free Block Pool
Insert
SSD Controller
Used Block Pool
Empty Data Page
Valid Data Page
InValid Data Page


Free Block Pool
Update
SSD Controller
Used Block Pool
Empty Data Page
Valid Data Page
Invalid Data Page

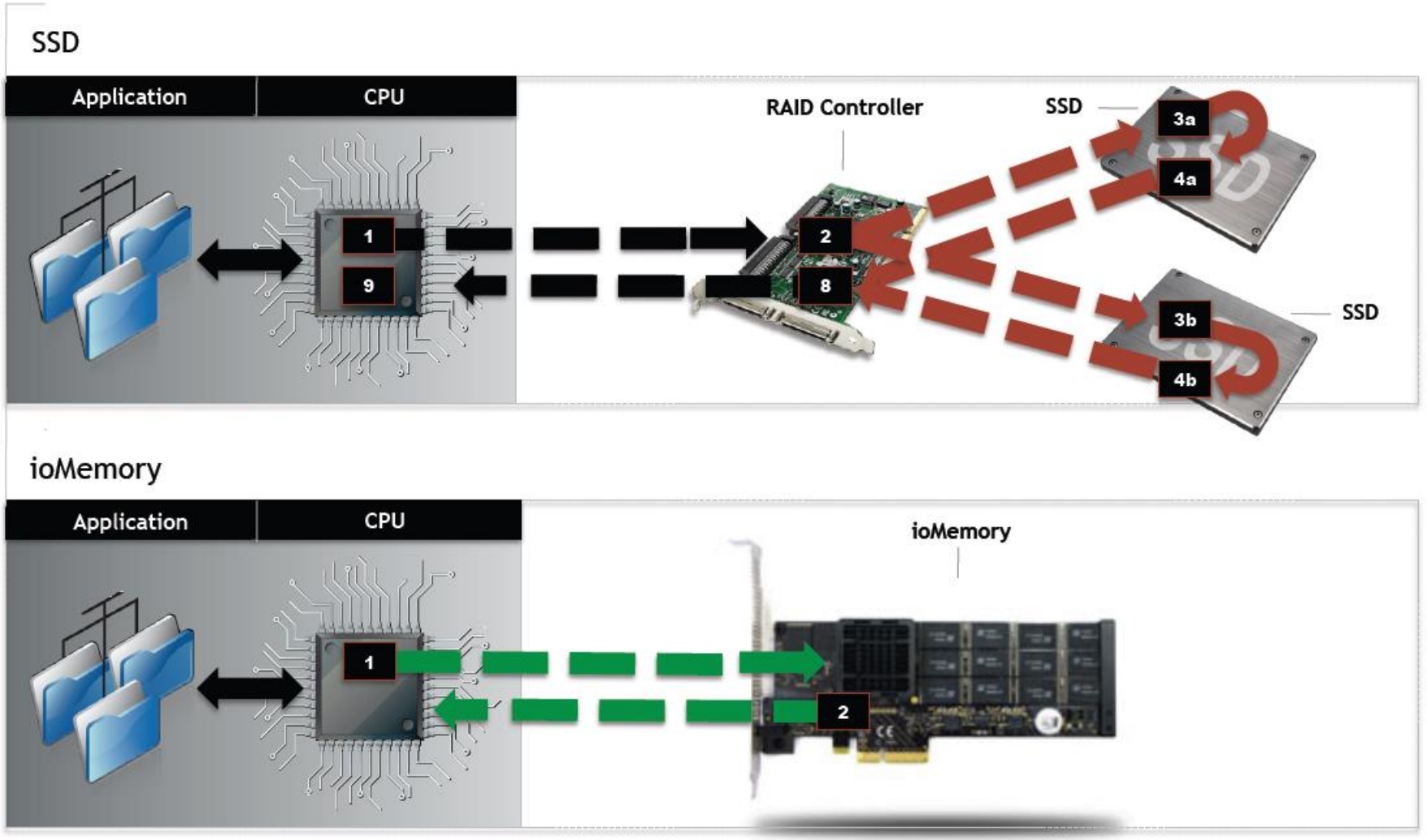

Free Block Pool
SSD Controller
Used Block Pool
Empty Data Page
Valid Data Page
Invalid Data Page

Flash write mitigation algorithms

- Garbage collection (GC) maintains free blocks for writes.
- “Write Amplification” measures the efficiency of GC
- Wear Levelling” ensures that writes are evenly spread across the drive
- Some SSD have extra blocks for write optimization
- Databases want a SSD with SLC, advanced garbage collection and low write amplification

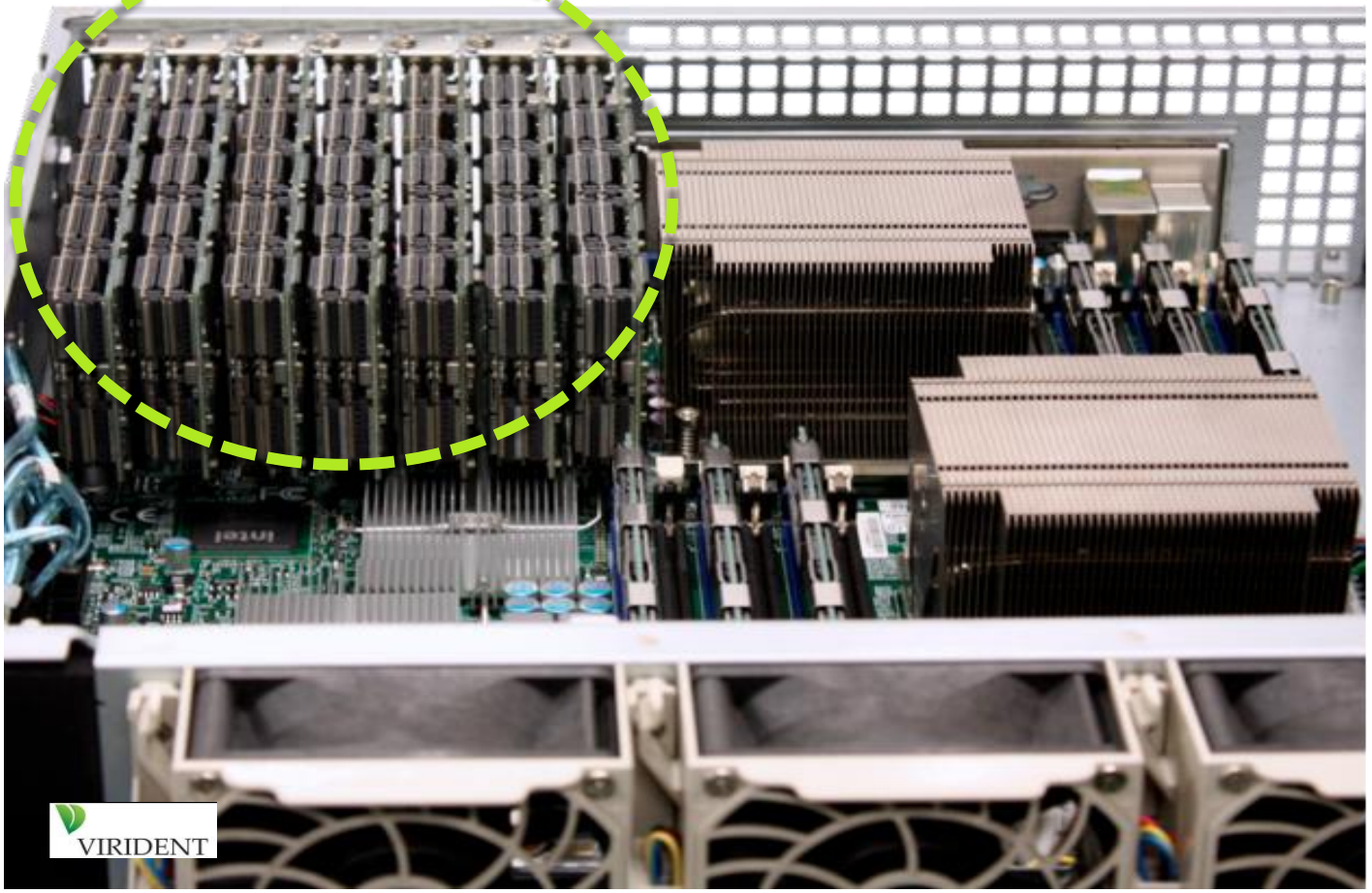
Other SSD topics

- PCI vs SATA
  - SATA was designed for traditional disk drives with high latencies
  - PCI is designed for high speed devices
  - PCI SSD provides higher throughput than SATA
- Network and CPU bandwidth
  - Implementing SSD inverts the typical performance profile
  - Network links and CPU capacities may be the new bottleneck
- Beware the write cache
  - Intel X-25E write cache is volatile
  - PCI such as Virident, RamSan, FusionIO are capacitor backed


SSD
Application
CPU
RAID Controller
SSD
1
9
2
8
3a
4a
3b
4b
SSD
ioMemory
Application
CPU
ioMemory
1
2

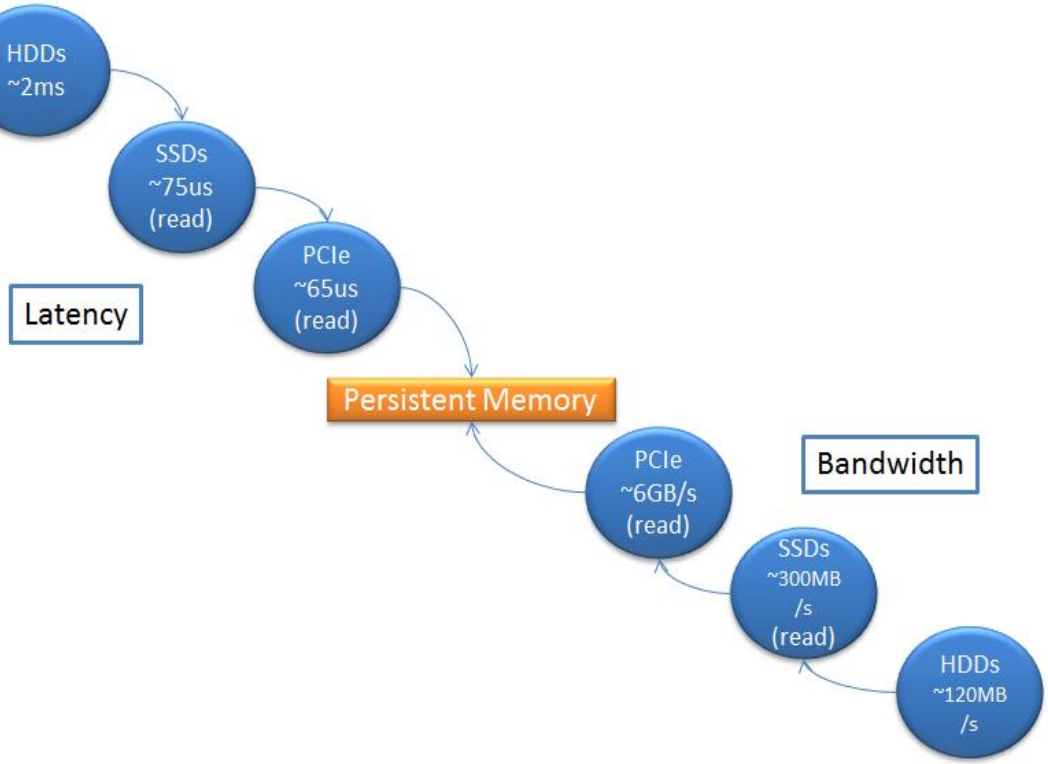
> PCI SSD

7 low profile, PCIe *tach*IOn™ drives
Each drive = 200GB – 800GB, 200K IOPS (4KB, 75% reads)
7 cards = 1.4TB – 5.6TB



www.viridient.com

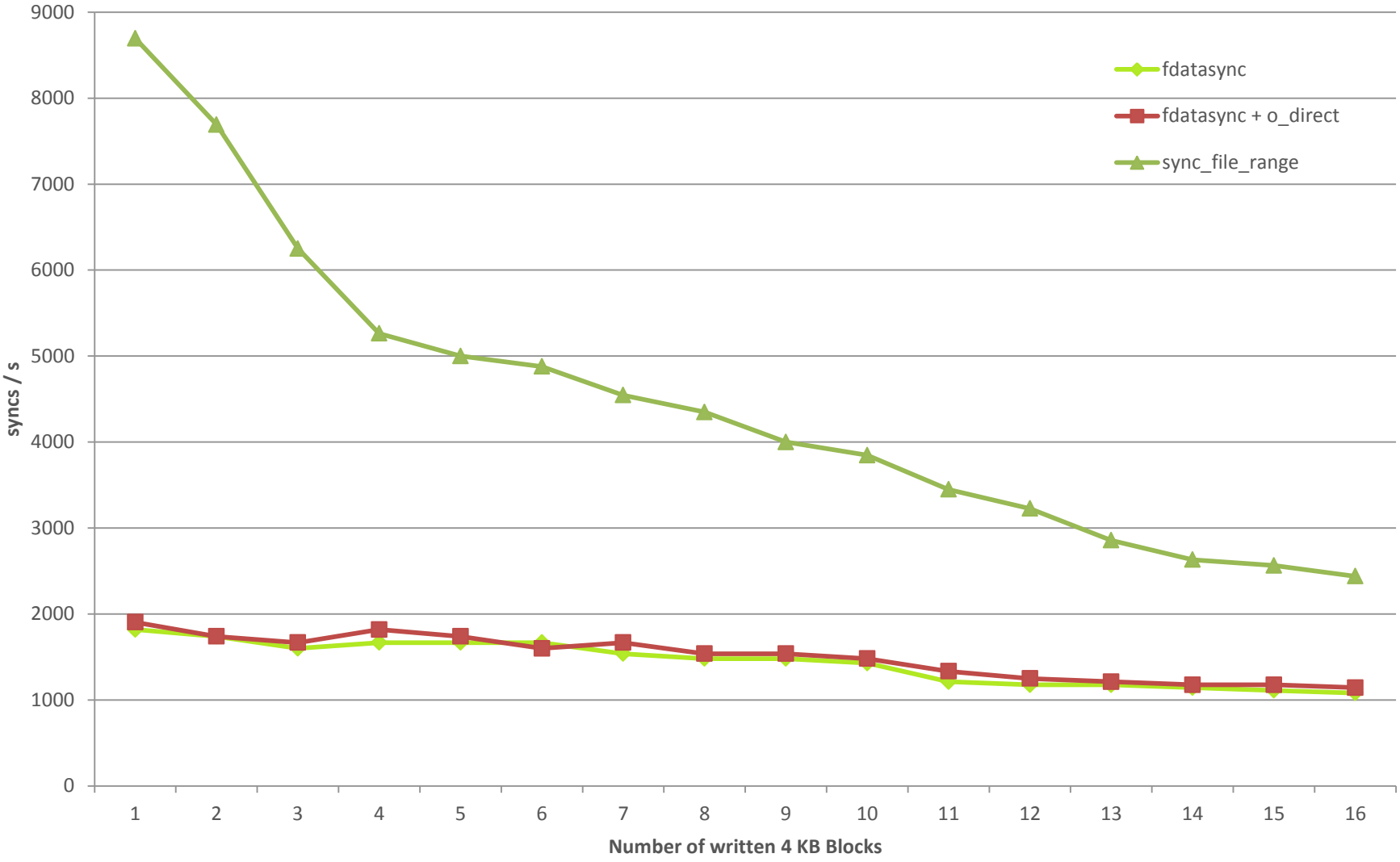
- Today, widely used as auxiliary storage or a long-term memory.
- Emerging byte addressable NVM. Examples: MRAM(IBM), FRAM (Ramtron), PCM(Samsung).
- Merging Point between storage and memory.
- Need of an evolved NVM programming model.



Ways of making writes durable:

- Only commit transaction when the (logging-) data was successfully persisted on disk
- “fdatasync” to flush I/O buffers to disk

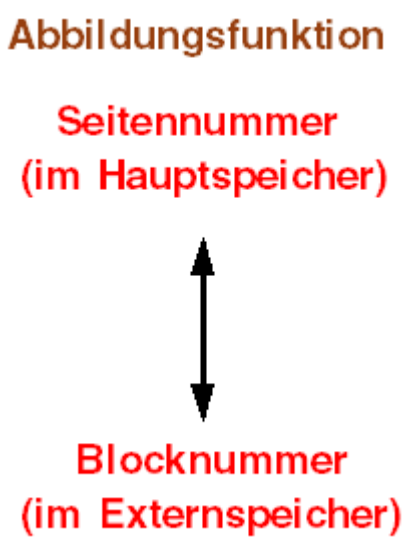
fsync	Write-back data and meta data. Issues a CACHE FLUSH to disk
fdatasync	Write-back only data. Issues a CACHE FLUSH to disk
msync	Effectively a call to **fsync**
open(… | O_DSYNC)	Automatically calls **fdatasync** after every write operation
sync_file_range	syncs the given range of the file. Ambiguous documentation. Man pages claim it to be durable. Test revealed, that it doesn’t flush disks write cache. Therefore its is more than twice as fast as **fdatasync**
fdatasync + open(… | O_DIRECT)	O_DIRECT flag lets writes bypass the operating systems file cache. Collateral damage: read cache disabled, no read ahead.

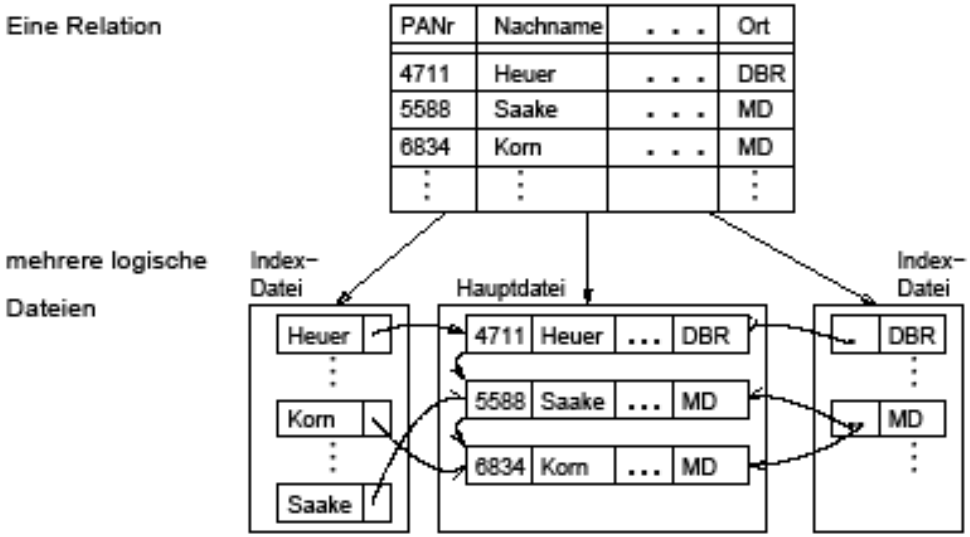

9000
8000
7000
6000
5000
4000
3000
2000
1000
0
syncs / s
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Number of written 4 KB Blocks
fdatasync
fdatasync + o_direct
sync_file_range

Storage Layer

Aufgaben

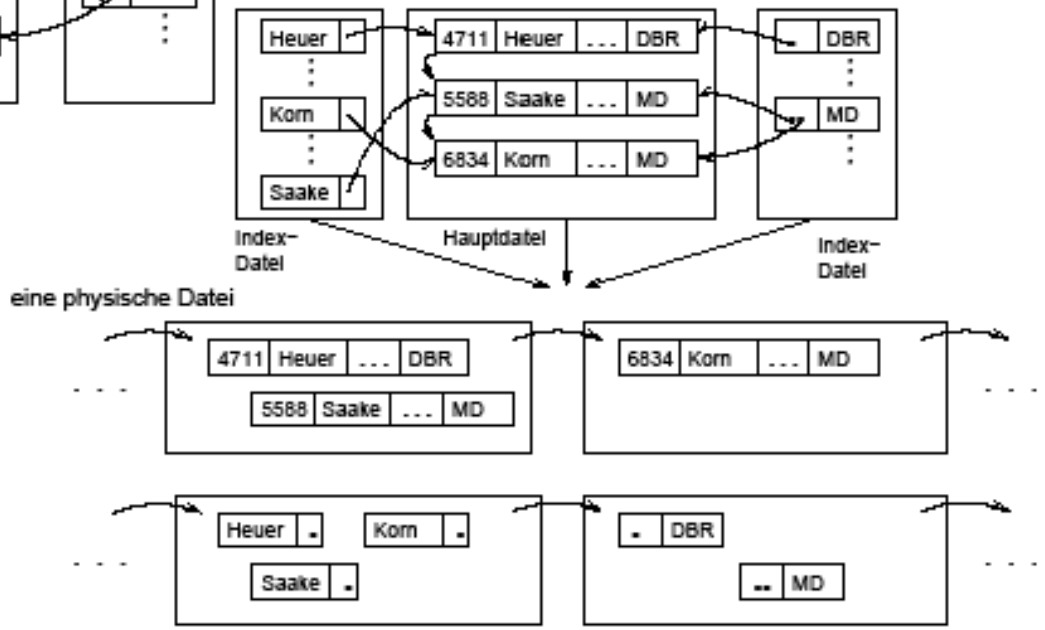
- Bereitstellen eines linearen, logischen, potentiell unendlichen, aktual aber endlichen Adressraumes (Segment) mit sichtbaren Seitengrenzen.
- Freie Adressierbarkeit innerhalb eines Segmentes wie in einem virtuellem Adressraum
- Abbildung von Seiten aus Segmenten auf physische Blöcke von Dateien
- Pufferverwaltung i.e.S., d.h. Auswahl einer zu ersetzenden Seite, falls Puffer voll, und Anforderung der gewünschten Seite (Seitenwechselmechanismus).

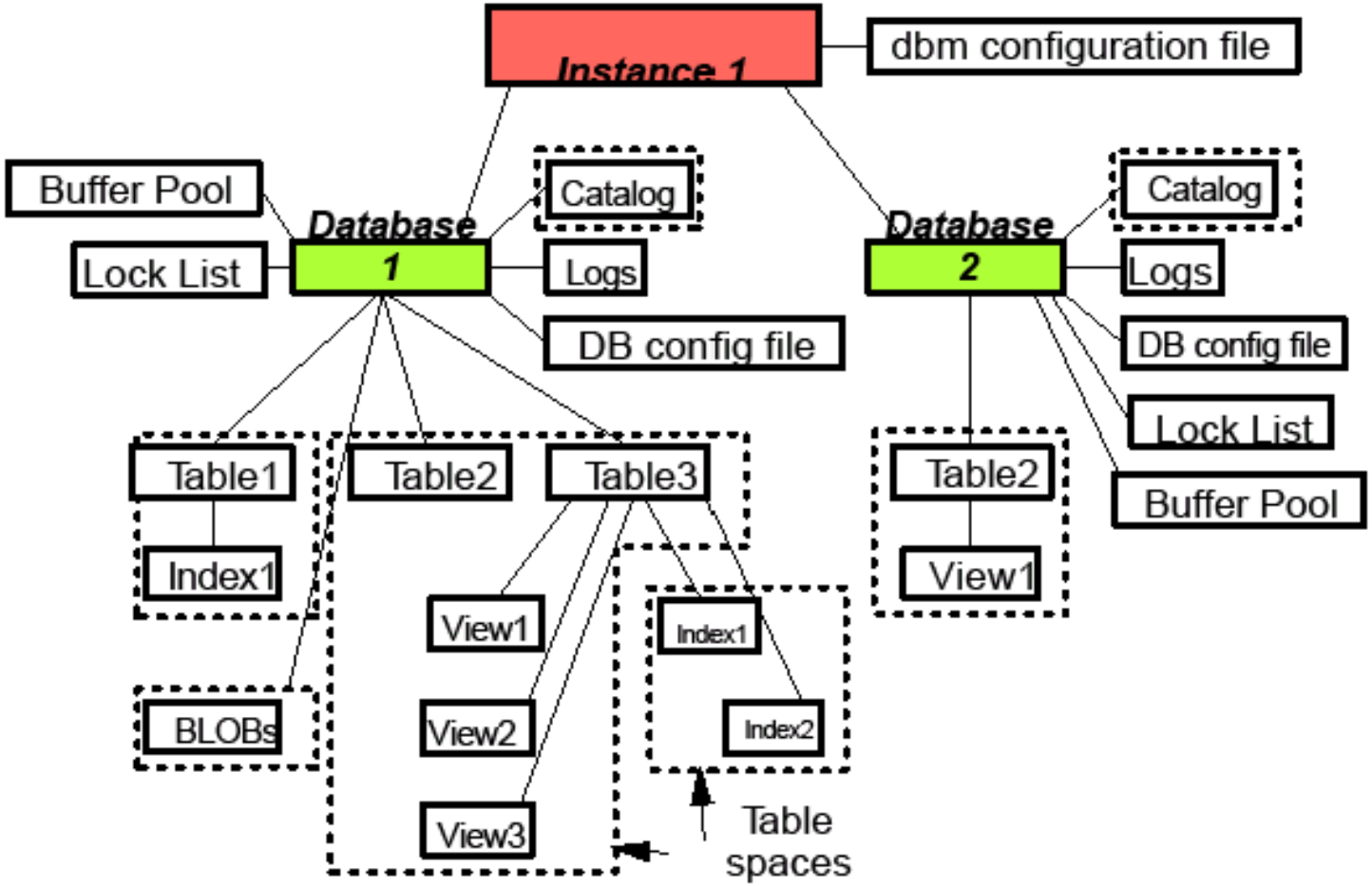




❑ Fazit

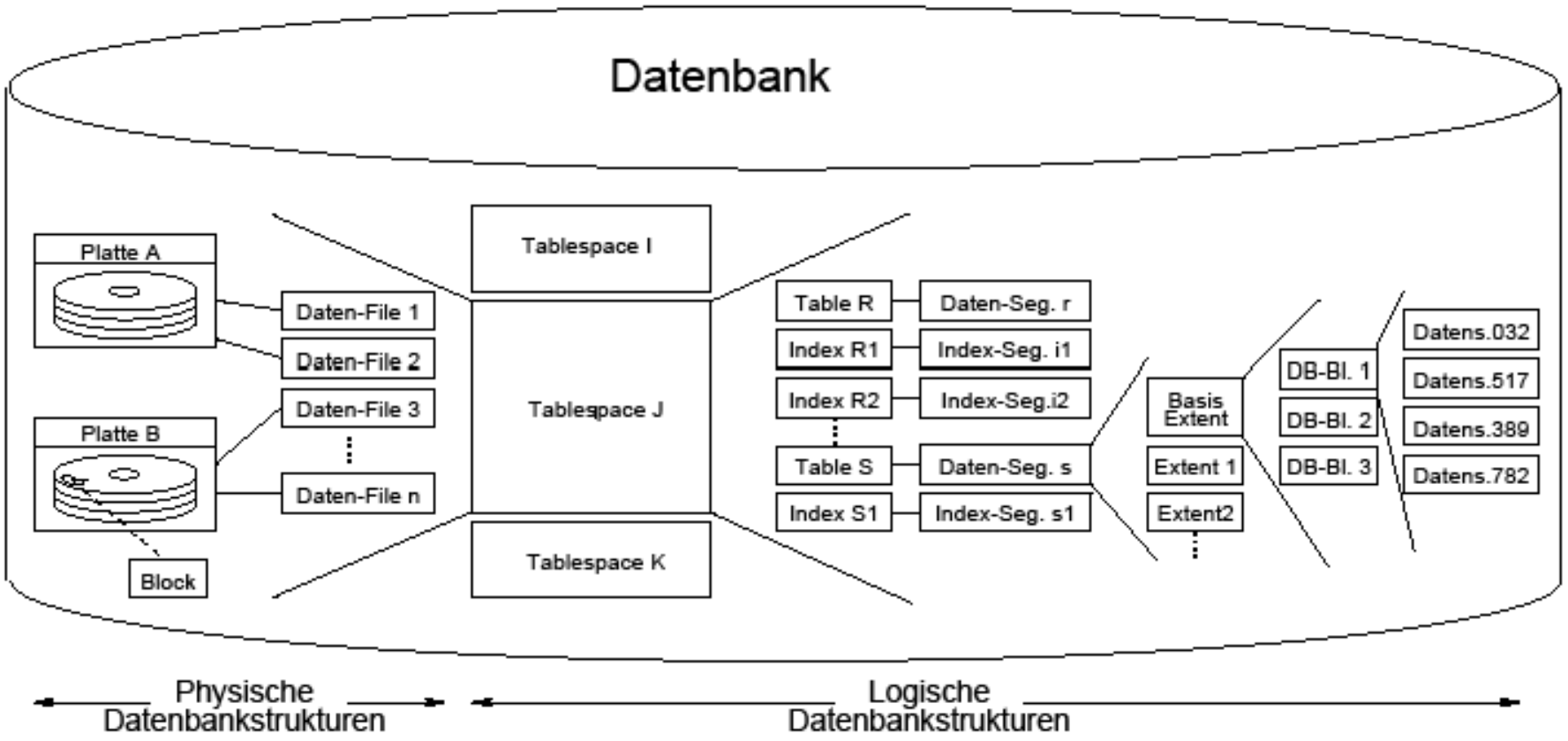
- Entkopplung logischer und physischer Einheiten
- Mehrstufiges Konzept -> Segmente





DB2 Data Management Software

IBM


Datenbank
Platte A
Platte B
Block
Daten-File 1
Daten-File 2
Daten-File 3
Daten-File n
Tablespace I
Tablespace J
Tablespace K
Table R
Index R1
Index R2
Table S
Index S1
Daten-Seg. r
Index-Seg. i1
Index-Seg.i2
Daten-Seg. s
Index-Seg. s1
Basis Extent
Extent 1
Extent2
DB-Bl. 1
DB-Bl. 2
DB-Bl. 3
Datens.032
Datens.517
Datens.389
Datens.782
Physische Datenbankstrukturen
Logische Datenbankstrukturen

Segmente

- Einheiten des Sperrens, der Recovery und der Zugriffskontrolle
- Verhältnis von Segment zu Seite
- Verhältnis Segment zu Seite: **1:N**
- Verhältnis Seite zu Block: **1:1 (typischerweise, aber auch 1:N in manchen Systemen)**
- Verhältnis Segment zu Datei: **sehr systemabhängig**
  - **1:1** alle Seiten eines Segmentes werden in Blöcken einer Datei gespeichert
  - **N:1** Seite mehrerer Segmente werden in Blöcken einer Datei gespeichert
  - **1:N** Seiten eines Segmentes werden in Blöcken unterschiedlicher Dateien gespeichert.

Ansatz: selektive Einführung zusätzlicher Attribute

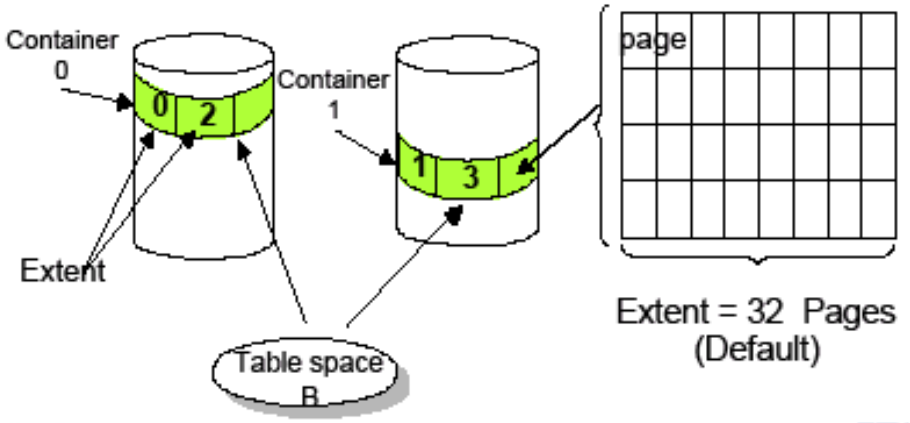
- Katalog, Schema-Informationen (SYSCATSPACE)
- Protokollinformationen, alle gemeinsam benutzbaren DB-Teile
- Teile der DB, die für bestimmte Benutzer oder Benutzergruppen reserviert sind
- private Kopien von Teilen der DB für einzelne Benutzer (Snapshots)
- Hilfsdateien für Benutzerprogramme
- Temporärer Speicher, z.B. für Sortierprogramme (TEMPSPACE1)

Unterschiedliche Typen (DB2)

- REGULAR: normale Datenbestände
- LARGE: LOB-Datentypen
- TEMPORARY
  - SYSTEM TEMPORARY: wird vom System benutzt für Sortierungen, Verbundoperationen ..
  - USER TEMPORARY: explizit ausgewiesene temporäre Daten (kein Logging, ...)
- ...muss mit einem Systempuffer mit der gleichen Seitengröße assoziiert sein

Containers und Extents

- Container (Segment in Oracle)
  - Teil einer (logischen) Disk / Datei, der zu einem TableSpace beiträgt
- Extent: Allokationseinheit innerhalb eines Containers



Unterschiedliche Speicherverwaltung

- System Managed Space (SMS)
  - alle Tabellendaten und Indexdaten teilen sich den gleichen TableSpace
  - einfach zu verwalten, speziell für TEMP TableSpace

```
CREATE TABLESPACE ts1
MANAGED BY SYSTEM USING ( '/mydir1', '/mydir2' )
EXTENTSIZE 4
```

  - Containers = Verzeichnisse in Dateisystemen
  - Datenverteilung: Striping über Verzeichnisse, Belegung nach Bedarf

- Database Managed Space (DMS)
  - Speicher wird zur Erzeugungszeit belegt (SMP: Space Map Pages über belegte/freie Extents)
  - automatische Verteilung der Daten über Container
  - Nutzung von direkten I/O-Vorgängen: RAW-devices möglich

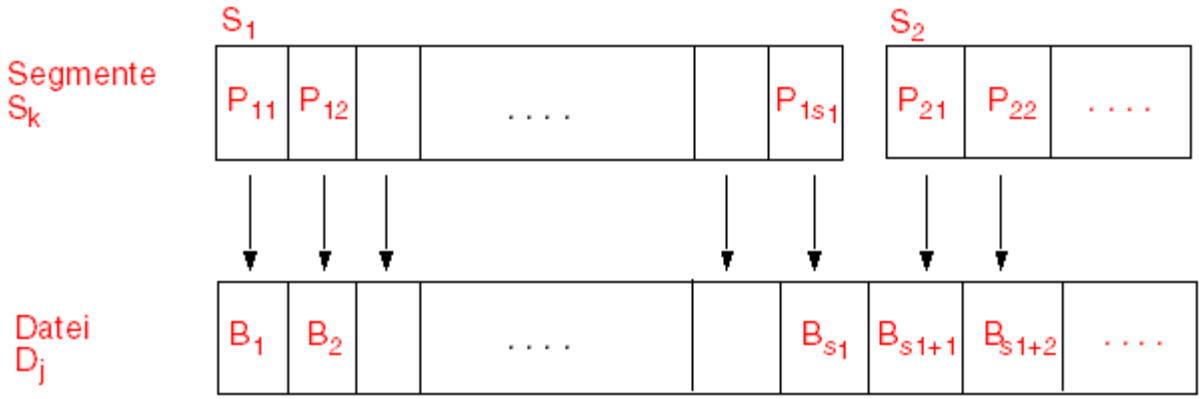
```
CREATE TABLESPACE ts2
MANAGED BY DATABASE USING (FILE '/myfile' 1024,DEVICE '/dev/rhd7'
2048)    EXTENTSIZE 4 PREFETCHSIZE 8 ;
```

Prinzip

- Zuordnung der (physischen) Blöcke zu Seiten
  - festes Größenverhältnis: 1,2,4 oder 8 Blöcke -> Seite
- höhere Schichten des DBS adressieren über Seitennummer

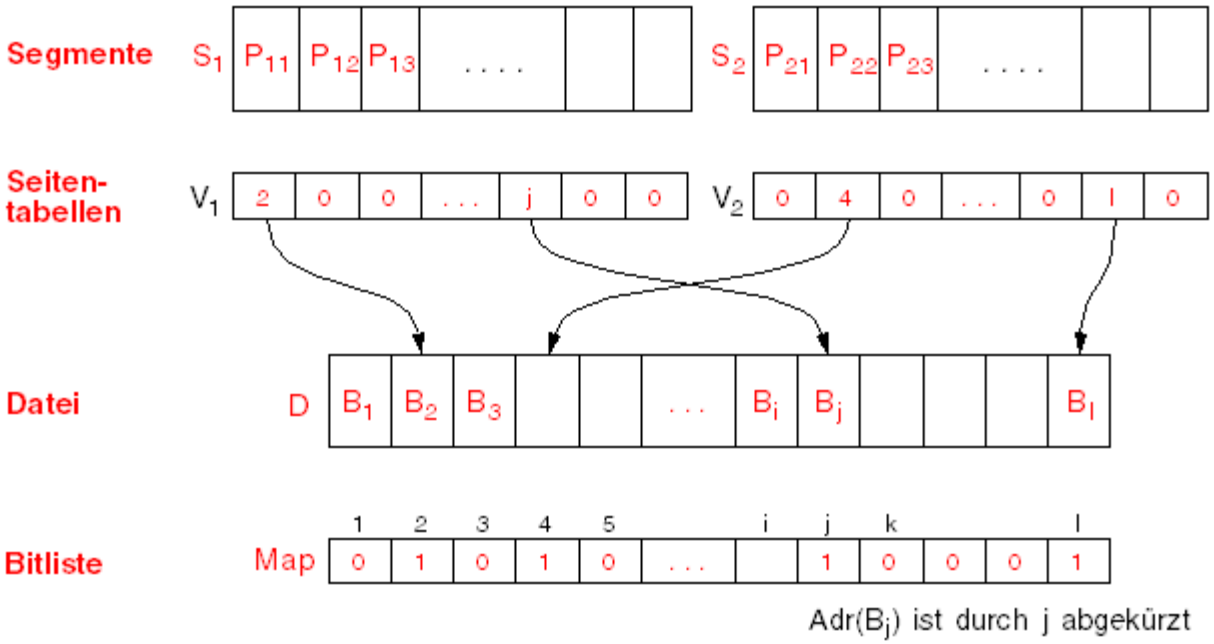
Möglichkeiten der Seitenadressierung

- Direkte Seitenadressierung
  - aufeinanderfolgende Seiten (bzgl. des virtuellen Adressraumes) werden auf aufeinanderfolgende Blöcke einer Datei abgebildet
- Indirekte Seitenadressierung
  - Maximum an Flexibilität bei der Blockzuordnung
  - benötigt zwei Hilfsstrukturen zur Seitenzuordnung



Eigenschaften

- keine Fragmentierungsprobleme wegen der geforderten Übereinstimmung von Seiten- und Blockgröße ($L_k = L_j$)
- jede Seite $P_{ki} \in S_k$ wird genau ein Block $B_{jl} \in D_j$ zugeordnet
- funktioniert natürlich nur für N:1 Verhältnis von Segment (Datenbanksystem) und Datei (Betriebssystem)



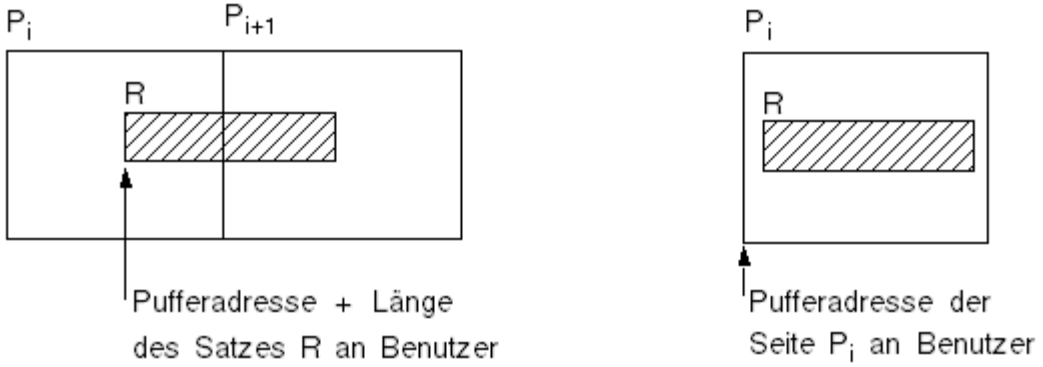
- Für jedes Segment S_k existiert eine Seitentabelle V_k, die für jede Seite einen Eintrag (4 Bytes) mit der aktuellen Blockzuordnung besitzt.
- Für die Datei D existiert eine Bitliste die ihre aktuelle Belegung beschreibt, d. h., die für jeden Block angibt, ob er momentan eine Seite enthält oder nicht.

Vergleich mit "Virtueller Hauptspeicher aus BS"

- Virt. HS: keine Kachelgrenzen zu den Anwendungsprogrammen hin sichtbar.
- Frage: Warum Erhaltung der Seitengrenzen für ein DBS?

"spanned record facility"

- benachbarte Seiten müssen im DB-Puffer benachbart angeordnet sein
- resultiert in erheblichen Abbildungs- und Ersetzungsproblemen



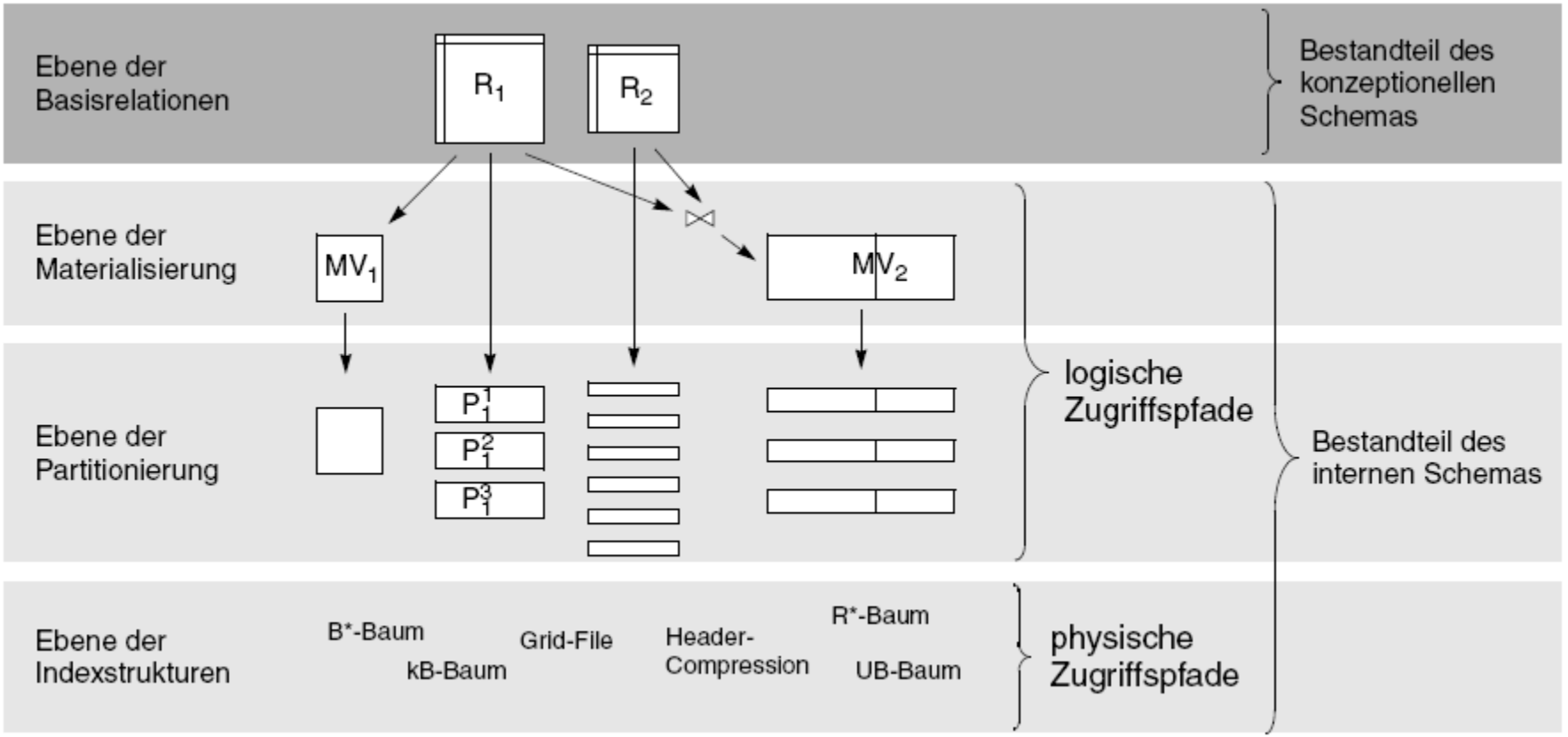
a) Satzreferenzierung mit relativer Byteadresse

b) Seitenreferenzierung

- deshalb: sichtbare Seitengrenzen an der DB-Speicherschnittstelle

Partitionierung

Auszug aus 3-Schema-Schichtenarchitektur



Horizontale Verteilung/Partitionierung von Relationen

- Nutzung von Datenparallelität
- Verbesserung von Bandbreite und E/A-Rate (E/A-Parallelität)
- Lastbalancierung

Teilaufgaben zur Bestimmung der Datenverteilung

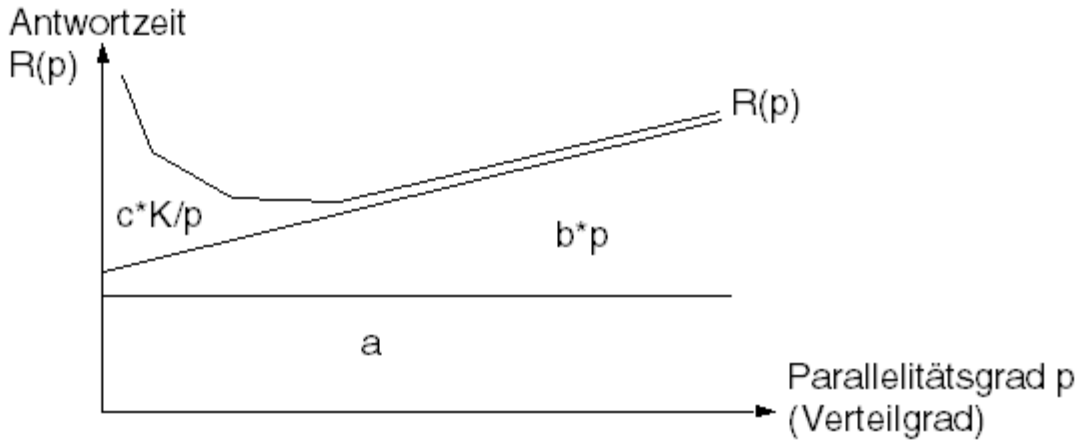
- Festlegung des Verteilgrads D einer Relation (Anzahl Partitionen)
- Fragmentierung
- Allokation

Bestimmung des "optimalen" Verteilgrads schwierig

- Kommunikations-Overhead steigt mit N (Anzahl der Rechner)
- Parallelisierungsgewinn sinkt mit wachsendem N
- "Full Declustering" (D = N) oft nicht sinnvoll
- einfache Anfragen zur Reduzierung des Kommunikationsaufwands auf möglichst wenige Partitionen beschränken
- datenintensive Anfragen (große Relationen) zur Nutzung von Parallelität / Lastbalancierung auf größere Anzahl von Partitionen verteilen

Mathematisches Modell

- p = Parallelitätsgrad
  a = nicht-parallelisierbare Anteile (z.B. Initialisierungskosten)
  b × p = Zeit zum Starten und Beenden der p Teiloperationen (steigt linear mit p)
  c × K = eigentliche Nutzarbeit (steigt linear mit Anzahl Tupel)
- Antwortzeit: R(p) = a + b × p + (c × K) / p
- optimaler Parallelitäts-/Verteilgrad p_{opt} = Ö((c × K) / b)



Optimaler Parallelitäts-/Verteilgrad: p_{opt}

- für einen Anfragetyp berechnet
- für verschiedene Anfragetypen durchaus unterschiedlich
- z.B. 64 für Relationen-Scan, 52 für Index-Scan mit 1 % Selektivität und 16 für Index-Scan mit 0,1 % Selektivität

Verteilgrad D

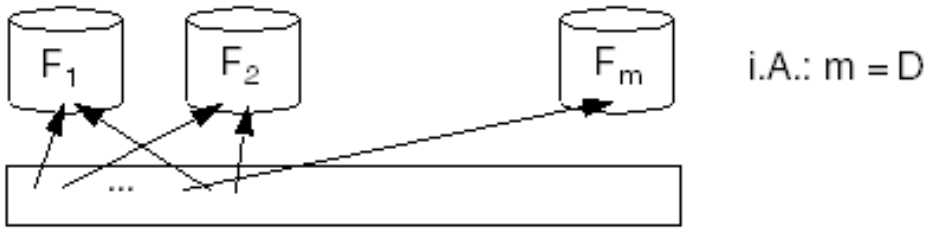
- Berechnung als gewichteten Mittelwert
- im Beispiel:
  $D = 0{,}2 \times 64 + 0{,}4 \times 52 + 0{,}4 \times 16 = 40$

Vorteile

- Einfachheit
- gleichmäßige Fragmentgrößen
- günstige Lastbalancierung (geringer "Skew")
- gleiche Tupelanzahl pro Rechner garantiert jedoch nicht immer gleichmäßige Zugriffshäufigkeit

Nachteil

- Verteilung von Attributwerten unbekannt
- Sämtliche Anfragen müssen an allen Rechnern bearbeitet werden.
- besonders effizient für Einzelsatzzugriffe sowie Exact-Match-Anfragen



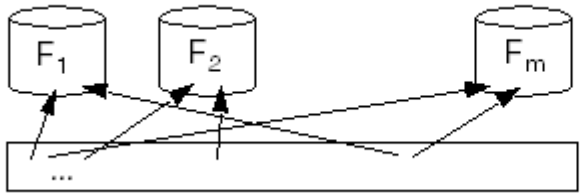
Verteilung der Tupel über Hash-Funktion auf Fragmentierungs- bzw. Verteilattribut VA (z.B. Primärschlüssel)

Vorteile

- Einfachheit
- Exact-Match-Queries auf Verteilattribut auf einem Fragment
- Equi-Joins über das Verteilattribut werden unterstützt

Nachteile

- keine Unterstützung für Bereichsanfragen (range queries)
- Gefahr ungleichmäßiger Partitionsgrößen (Skew) bei "schlechter" Hash-Funktion und ungünstiger Werteverteilung
- Hash-Fragmentierung



i.A.: m = D

Beispiel: Oracle

- Angabe von Partitionen
- Benennung optional…

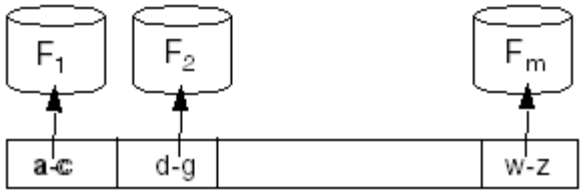
```
// Oracle-SQL-Dialekt
  CREATE TABLE TPCD.ORDERS (O_ORDERKEY       INTEGER NOT NULL,
                            O_CUSTKEY        INTEGER NOT NULL,
                            O_ORDERSTATUS    CHAR(1) NOT NULL,
                            O_TOTALPRICE     DECIMAL(15,2) NOT NULL,
                            O_ORDERDATE      DATE NOT NULL,
                            ...
  PARTITION BY HASH(O_ORDERKEY, O_CUSTKEY) PARTITIONS 4);
```

Merke

- Jede Partition kann individuelle Speicherrepräsentation besitzen

Fragmentierung durch Wertebereiche

- Spezifikation durch Prädikate auf einem Verteilattribut
- Analog zu: Fragmentierungsansatz verteilter DBS



Vorteile

- Exact-Match-Queries sowie Range-Queries auf Verteilattribut auf relevante Fragmente (Rechner) eingrenzbar
- Unterstützung für Equi-Joins über das Verteilattribut
- stabiler als Hash-Fragmentierung bei ungleicher Werteverteilung

Nachteile

- erhöhte Gefahr ungünstiger Lastbalancierung
- Bestimmung der einzelnen Wertebereiche relativ aufwändig

Ziel

- Verbesserung der Lastbalancierung durch größere Anzahl von Fragmenten (m > D)

Beobachtung

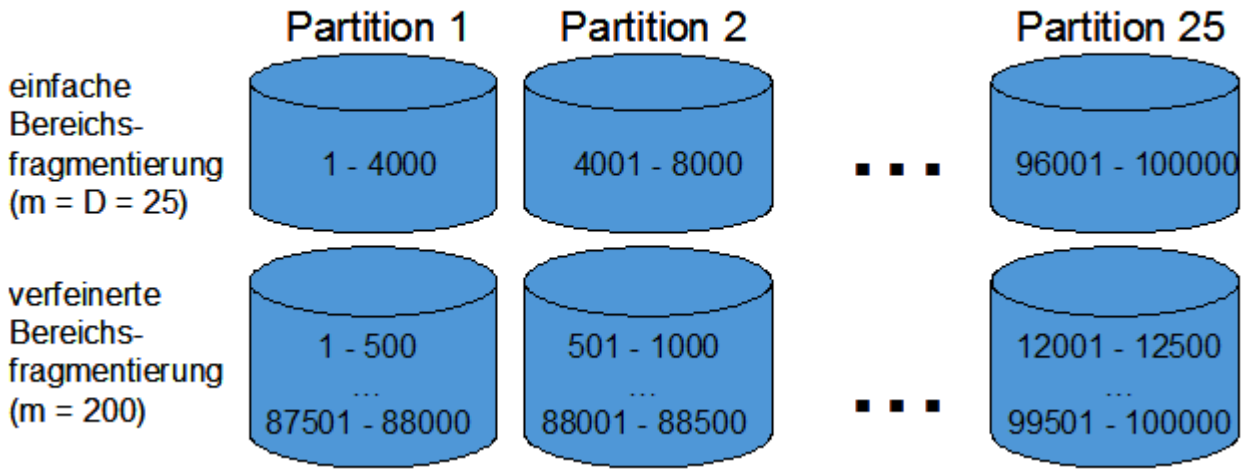
- einfache Bereichsfragmentierung beschränkt Anfragen über Verteilattribut auf minimale Anzahl von Partitionen, falls m = D (ein Fragment pro Rechner)

Ansatz

- erhöhte Fragmentanzahl, so dass relevanten Fragmenten im Mittel p_{opt} Rechner zugeordnet werden können:
- $m = p_{opt} / S$ (S = mittlere Selektivität der Bereichsanfragen, $0 \leq S \leq 1$)

Beispiel

- card (Konto) = 100.000
- Verteilattribut KtoNr
- S = 0,05
- p_{opt} = 10
- D = 25



Paritionierung bei Oracle

- Einstufig über RANGE-Anweisung; Sortierung ist maßgeblich!

```
// Oracle-SQL-Dialekt
  CREATE TABLE TPCD.LINEITEM (
          L_ORDERKEY                  INTEGER NOT NULL,
          L_PARTKEY                   INTEGER NOT NULL,
          L_SUPPKEY                   INTEGER NOT NULL,
          ...,
          L_SHIPDATE                  DATE NOT NULL,
          ...
  PARTITION BY RANGE(L_SHIPDATE)
  ( PARTITION LINEITEM1998 VALUES LESS THAN '1999-01-01' ),
  ( PARTITION LINEITEM1999 VALUES LESS THAN '2000-01-01' ),
  ( PARTITION LINEITEM2000 VALUES LESS THAN '2001-01-01' ),
  ( PARTITION LINEITEM2001 VALUES LESS THAN '2002-01-01' ) );
```

Ziel

- Unterstützung von Exact-Match- und Bereichsanfragen auf mehreren Attributen

Beispiel

- zwei Anfragetypen, 9 Rechner, 36 Fragmente

```
select * from Pers where Name = :Z
select * from Pers where Gehalt between [:X, :Y]
```

Hash- und Bereichsfragmentierung

- können nur einen Anfragetyp unterstützen
- für andere sind alle Rechner involviert

mehrdimensionale Bereichsfragmentierung

- erlaubt, beide Anfragetypen auf Teilmenge der Fragmente zu beschränken
- nur empfehlenswert, wenn bei mehreren Attributen etwa gleiche Zugriffshäufigkeit besteht

... Am Beispiel Oracle (Hash- und Bereichspartitionierung)

- Stufe 1: nach Bereichen (definiert über die Anwendung)
- Stufe 2: nach Hash (festgelegt durch physische Speicherorganisation)

```
// Oracle-SQL-Dialekt
  CREATE TABLE TPCD.LINEITEM (
          L_ORDERKEY       INTEGER NOT NULL,
          L_PARTKEY        INTEGER NOT NULL,
          L_SUPPKEY        INTEGER NOT NULL,
          ...,
          L_SHIPDATE       DATE NOT NULL,
          ...
  PARTITION BY RANGE(L_SHIPDATE)
          SUBPARTITION BY HASH(L_ORDERKEY, L_PARTKEY, L_SUPPKEY) ...,
  ( PARTITION LINEITEM2000 VALUES LESS THAN '2001-01-01' )
          SUBPARTITIONS 2,
  ( PARTITION LINEITEM2001 VALUES LESS THAN '2002-01-01' )
          SUBPARTITION LINEITEM2001_1 TABLESPACE DATA2001_PART1
          SUBPARTITION LINEITEM2001_2 TABLESPACE DATA2001_PART2
          SUBPARTITION LINEITEM2001_3 TABLESPACE DATA2001_PART3
          SUBPARTITION LINEITEM2001_4 TABLESPACE DATA2001_PART4);
```

Allokation: Rechnerzuordnung der Fragmente (Shared Nothing)

- Festlegung des Verteilgrads D
- "gleichmäßige" Aufteilung der m Fragmente unter D Partitionen
- Ziel: (statische) Lastbalancierung
- analoge Partitionierung von Indexstrukturen

Replikation

- Fehlertoleranz gegenüber Externspeicherfehlern
- Fehlertoleranz gegenüber Rechnerausfällen
- günstige Lastbalancierung im Normalbetrieb und Fehlerfall

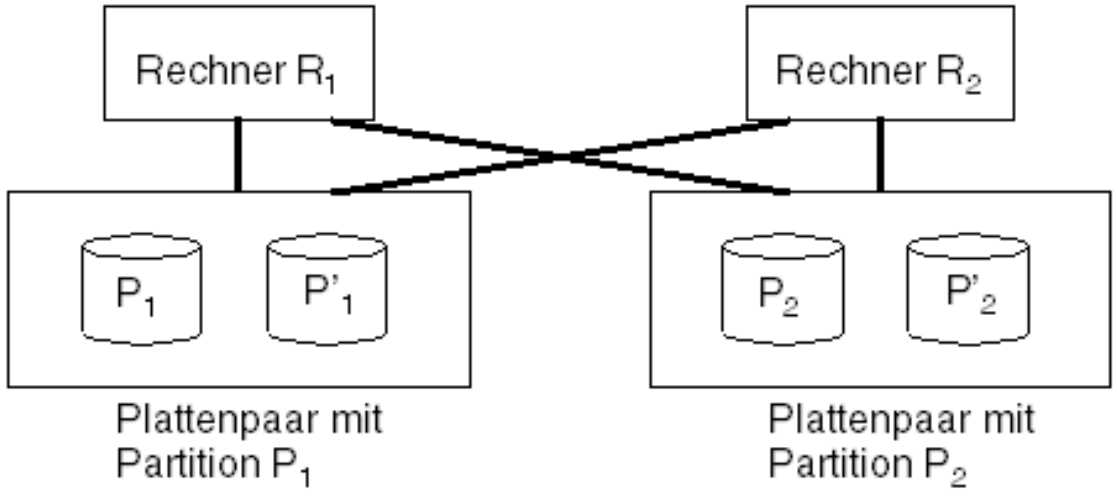
hier: Replikationsansätze mit doppelter Speicherung der Daten

Allgemein: k-safety

- Ausfall von k Replikaten kann verkraftet werden

Zielsetzung

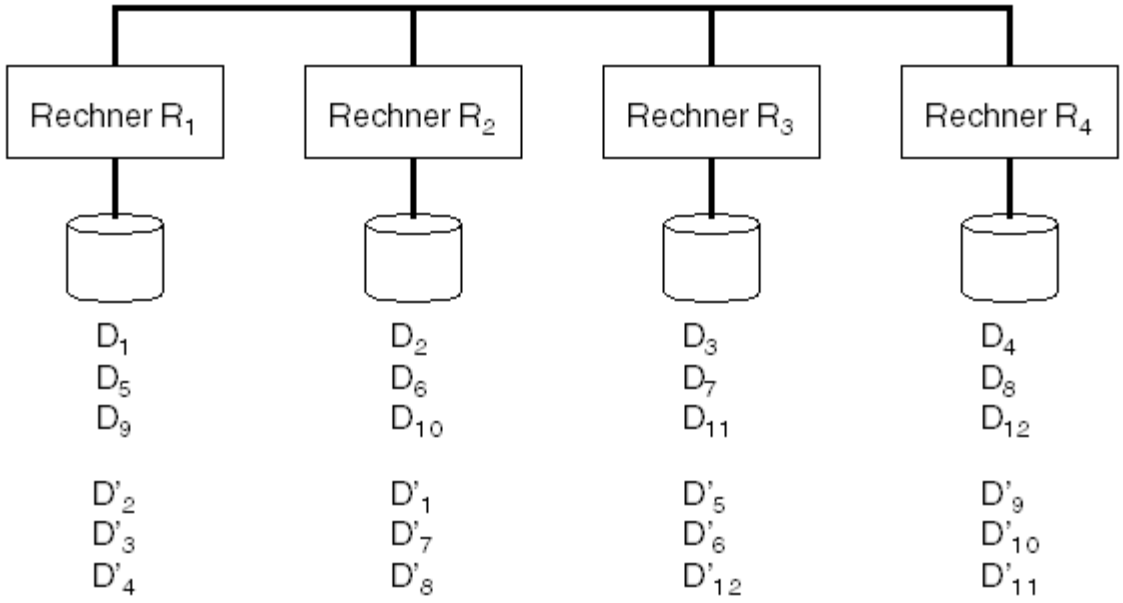
- verbesserte Lese-Leistung
- weit verbreitet zur Maskierung von Plattenfehlern
- Shared Nothing: Replikation auf einen Knoten beschränkt
- Rechnerausfall erfordert Übernahme der kompletten Partition durch zweiten Rechner (ungünstige Lastbalancierung im Fehlerfall)



Ziel: bessere Lastbalancierung im Fehlerfall

- Datenknoten einer Relation werden in **Gruppen** von G Rechnern unterteilt
- Replikate zu Daten eines Rechners werden gleichmäßig unter den G–1 anderen Rechnern der Gruppe verteilt
- nach Crash erhöht sich Last jedes überlebenden Rechners der Gruppe gleichmäßig um Faktor G/G–1
- Ausfall mehrerer Rechner beeinträchtigt Datenverfügbarkeit nicht, sofern jeweils verschiedene Gruppen betroffen sind
- Wahl von G erlaubt Kompromiss zwischen Verfügbarkeit und Lastbalancierung
- Extremfälle: G = D (=> Verteilgrad der Relation) und G = 2
- Nutzung der Replikate zur Lastbalancierung im Normalbetrieb aufwändig (im Teradata-System nicht unterstützt)

Beispiel (D = G = 4)

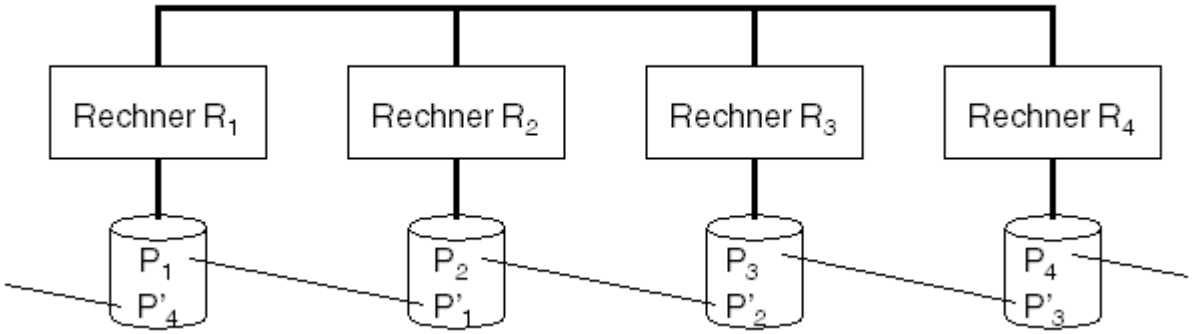


Ziel

- hohe Verfügbarkeit wie für Spiegelplatten und günstigere Lastbalancierung im Fehlerfall wie für verstreute Replikation

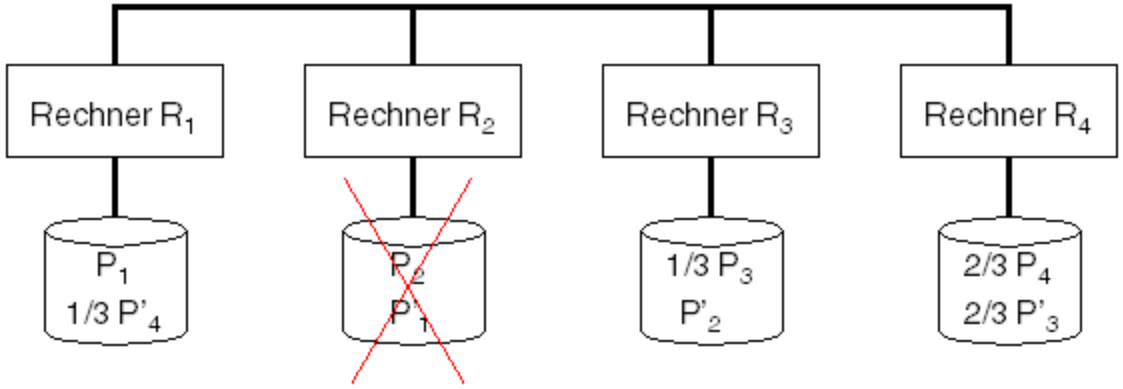
Ansatz

- Kopie der Partition von Rechner R_i wird vollständig am "nächsten" Rechner $R_{(i \text{ MOD } G) +1}$ der Gruppe gehalten
- Selbst Mehrfachausfälle in einer Gruppe erhalten die Verfügbarkeit aller Daten, solange nicht zwei benachbarte Rechner ausfallen.



Lastbalancierung im Fehlerfall

- Zugriffe auf betroffener Partition sind vollständig vom "nächsten" Rechner der Gruppe auszuführen
- Zugriffe auf den anderen Partitionen sind unter den beiden Kopien umzuverteilen, so dass Lastbalancierung erreicht wird
- keine Umverteilung der Datenbank, sondern nur Anpassung der Verteilungsinformationen



Komprimierung

Ziel

- Reduktion des Platzbedarfs auf dem Hintergrundspeicher
- Einsparung von E/A Operationen
- damit indirekt auch: Einsparung des Hauptspeicherbedarfs
  (Daten werden i.A. in komprimierter Form zwischen Hintergrundspeicher und Hauptspeicher transferiert)

Aktueller Stand

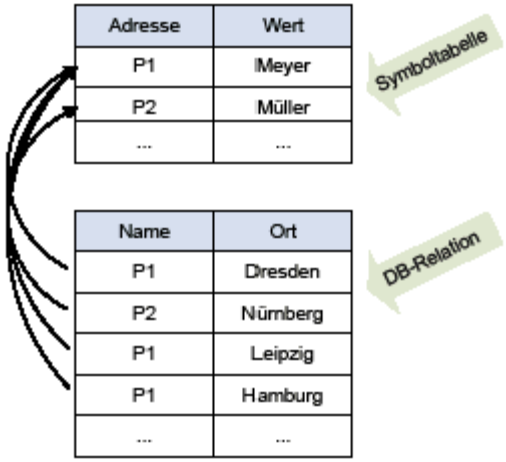
- kaum nennenswerte Rolle in der Forschung für klassische DB-Architekturen
- Bestandteil vieler kommerzieller Datenbanksysteme
  (z.B. Oracle, Sybase, Teradata ...)
- Wesentliches Paradigma für spaltenorientierte Datenorganisation

Prinzip

- Verfahren basieren auf der Nutzung einer Symboltabelle (Pointer, Wert)-Paaren

Symboltabelle

- global (ganze Relation)
- partiell (z.B. DB-Block)
- pro Spalte (Teradata, Sybase)
- spaltenübergreifend (Oracle)



Ersetzung von Duplikaten durch Referenz in die Symboltabelle

Vielzahl unterschiedlicher Ansätze

- spaltenbasiert und relationen-global (Teradata, Sybase IQ)
- blockbasiert (partiell) Oracle

Prinzip

- globale Symboltabelle pro Spalte => an den Datentyp der Spalte angepasste Komprimierung

Teradata

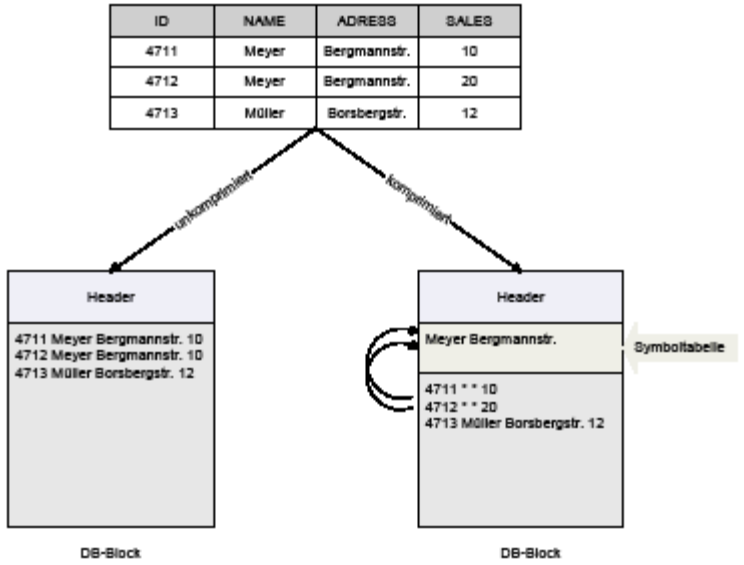
- Angabe des COMPRESS Schlüsselworts innerhalb des DDL Statements einer Relation für eine Spalte
- 8 Bit Pointer => maximal 255 Einträge (+ evtl. NULL-Werte) in der Symboltabelle
- Komprimierung der 255 häufigsten Attribut-Ausprägungen (ausreichend für viele Anwendungen => Zipf's Law)

Sybase IQ

- Standard Speicherstruktur: Fast Projection Index
- Angabe des IQ UNIQUE Schlüsselwortes für eine Spalte im DDL Statement einer Relation
- 8 und 16 Bit Adressierung in der Symboltabelle (256 bzw. 65536 verschiedene Werte)
- automatisches Roll-Forward bei steigenden Kardinalitäten, kein Rollback

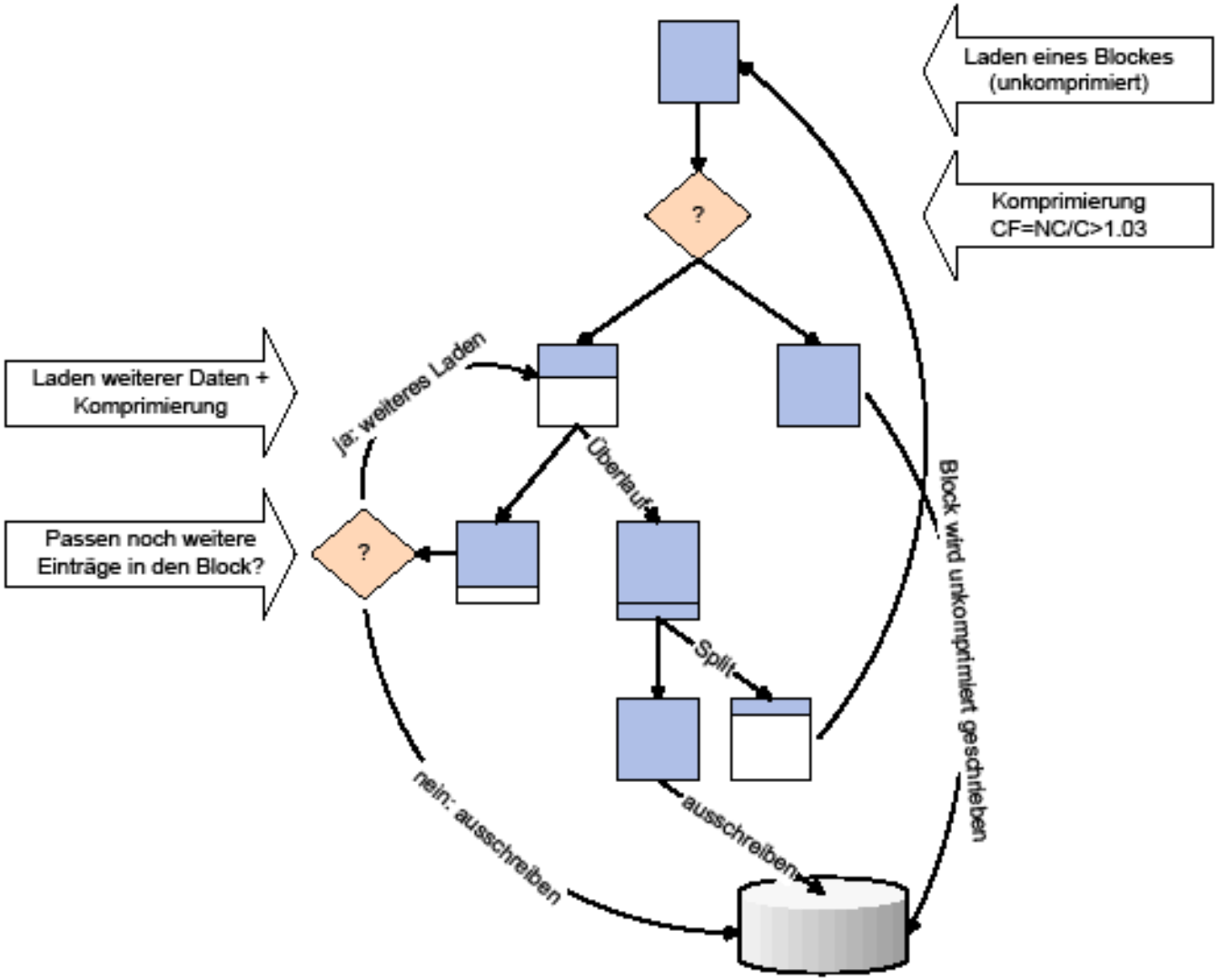
am Beispiel: Oracle

- eine Symboltabelle pro DB-Block => Referenzlokalität
- DB-Block kann unabhängig von anderen Blöcken dekomprimiert werden
- spaltenübergreifende Einträge in der Symboltabelle
- Eintrag in der Symboltabelle ist eine Ausprägung einer Spalte oder einer Sequenz von Spalten



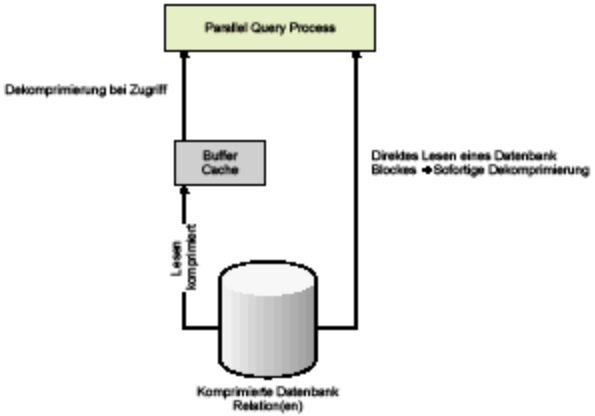
- Aufnahme eines Wertes in die Symboltabelle abhängig von der Häufigkeit des Auftretens des Wertes in einer Spalte bzw. Spaltensequenz
- Auswirkungen für Update und Delete Operationen => Wartung der Symboltabelle
- Reference-Counter für jeden Eintrag in der Symboltabelle (Inkrement / Dekrement)

> Komprimierungsprozess beim blockbasierten Ansatz



Lesender Zugriff

- Datenaustausch zwischen Hintergrundspeicher und Hauptspeicher in komprimierter Form
- Dekomprimierung erst bei Datenzugriff



Auswirkung auf DML-Operationen

- Delete
  - freigegebener Speicherplatz wird in komprimierten Blöc Speicherfragmentierung
- Update
  - Updates können nicht direkt auf komprimierten Blöcken erfolgen => Chained Rows
- Insert
  - Standard-Inserts werden nicht komprimiert, d.h. sie erzeugen oder erweitern ausschließlich nicht komprimierte Blöcke

Potenzial

- besonders bei lesenden Zugriff auf Relationen (z.B. Data Warehouse)
- vielfach verwendet für “kalte Replikate” (i.S. eines Backups)
- Reduktion des Platzbedarfs auf dem Hintergrundspeicher (i.A. 4:1, 2:1)
- weniger I/O, da Daten in komprimierter Form zwischen Hintergrundspeicher und Hauptspeicher ausgetauscht werden
  => Beschleunigung der Antwortzeiten bei Anfragen

nicht geeignet für OLTP-Anwendungen mit häufigen Updates

- Modifikation komprimierter Daten kann zu Fragmentierung und damit zu erhöhten Speicherbedarf führen

Speicherzuordnungsstrukturen erfordern effizientes Dateikonzept

- viele Dateien variierender, nicht statisch festgelegter Größe
- Wachstum und Schrumpfung erforderlich; permanente und temporäre Dateien

Unterscheidung in

- direkte Seitenadressierung (Seite im festen Block)
- indirekte Seitenadressierung (über eine Indirektionsstufe)

Datenverteilung

- mathematische Verteilung: schwierig
- Fragmentierung, Allokation, Replikation auf Satz-/Blockebene

Komprimierung

- tabellenbasiert
- blockbasiert