Datenbanken: Tabellenrepräsentation im Speicher

Stefan Halfpap, Ralf Teusner, Werner Sinzig

25. Mai 2020

Vorlesungsinhalte/-aufbau

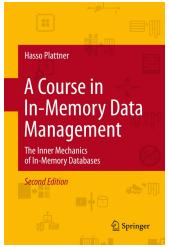
- Einführung zu Unternehmensanwendungen
- Grundlagen des IT-gestützten Rechnungswesens und der Planung
- Einführung zu relationalen Datenbanken und Anfrageverarbeitung
- **Grundlagen von (spaltenorientierten) Hauptspeicherdatenbanken**
- Trends in Hauptspeicherdatenbanken
- Klausur

Überblick

- Datentyprepräsentation
- Speicherhierarchie
- Tabellenrepräsentation
  - Zeilenorientiertes vs. spaltenorientiertes vs. hybrides Layout
- Festplattenbasierte zeilenorientierte Datenbanken
- Spaltenbasierte Hauptspeicherdatenbanken
- Zusammenfassung

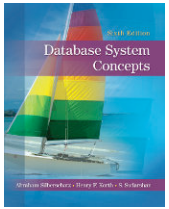
Literatur und Empfehlung

- Hasso Plattner „A Course in In-Memory Data Management”



- Avi Silberschatz „Database System Concepts“

  http://db-book.com/



- Andy Pavlo „Database Systems“

  https://15445.courses.cs.cmu.edu/fall2018/

- Andy Pavlo „Advanced Database Systems“

  https://15721.courses.cs.cmu.edu/spring2019/schedule.html

Tabellenrepräsentation im Speicher

- Relationales Modell als konzeptionelles/logisches Datenmodell mit abstrakten Operationen

First Name	Last Name	Country	Year of Birth
Paul	Smith	Australia	1986
Lena	Jones	USA	1990
Hanna	Schulze	Germany	1942
Hanna	Schulze	USA	2000

- Wie können Daten im Speicher repräsentiert werden?
- Wie können Anfragen **effizient** verarbeitet werden? (in der Regel enge Kopplung)

Datentypprepräsentation

- SQL-Datentypen:
  - Zeichenketten/Strings mit fester und variabler Größe
  - Zahlen mit exaktem Wert
  - Zahlen mit approximiertem Wert
  - Zeitpunkte und –intervalle
  - Große Objekte
  - Benutzerdefinierte Typen

Datentypprepräsentation

- Einzelner Attributwert ist irgendwann eine Bitsequenz im Speicher
  - Bitsequenz ist implementierungsabhängig, meist basierend auf „nativen“ C/C++ Typen
  - Komprimierung möglich
- Datenbank ist eine Aneinanderreihung von Attributwerten (Bitsequenzen); zusätzlich Schema und Metainformationen (welche Komprimierung, welches Datenlayout); + Alignment, Padding
- Datentypen mit fester vs. variabler Größe
  - Felder fester Länge ermöglichen direkten Zugriff im Falle einer kontinuierlichen Speicherung
  - Felder variabler Größe ermöglichen Speicherersparnis

Datentypprepräsentation
Repräsentation von NULL

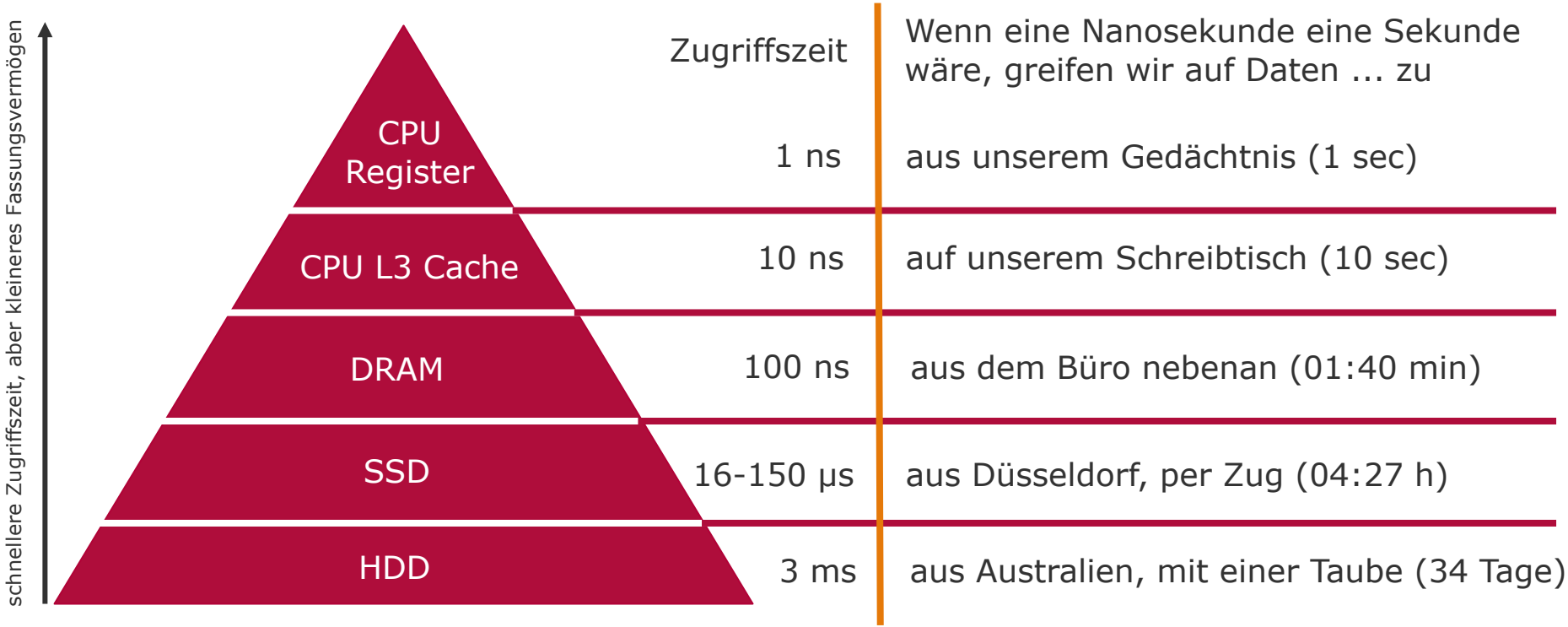
- Variante 1: Spezieller Wert
  - Festlegung eines speziellen Wertes für einzelne Datentypen (z.B. INT32_MIN)

    -> verkleinert den Wertebereich

- **Variante 2: Bitmap pro Spalte/Zeile**

- Variante 3: Markierung direkt am Attributwert

Speicherhierarchie
„Latency Numbers Every Programmer Should Know“



http://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html (Zahlen vom April 2018)

http://www.montana.edu/cpa/news/wwwpb-archives/yuth/pigeon.html

"All programming is an exercise in caching."

Terje Mathisen

Speicherhierarchie
„What Every Programmer Should Know About Memory“

- Ulrich Drepper „What Every Programmer Should Know About Memory“ (2007)
  - https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

Tabellenrepräsentation

- Wie sind zwei-dimensionale Tabellen im linearen (ein-dimensionalen) Adressraum abgebildet (um möglichst gut Caches zu nutzen)?
- Alternativ: physische Umsetzung des konzeptionellen relationalen Datenmodells

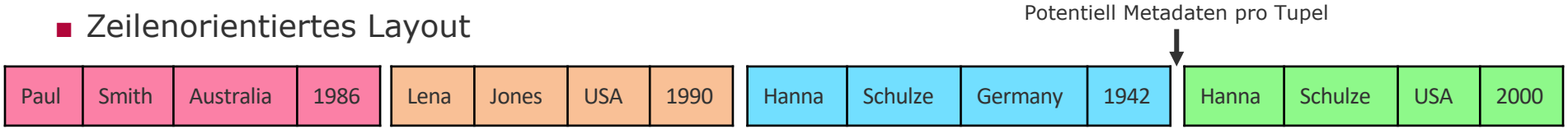
- 3 Varianten:
  - Zeilenorientiert
  - Spaltenorientiert
  - Hybrid (in Attributgruppen)

Tabellenrepräsentation

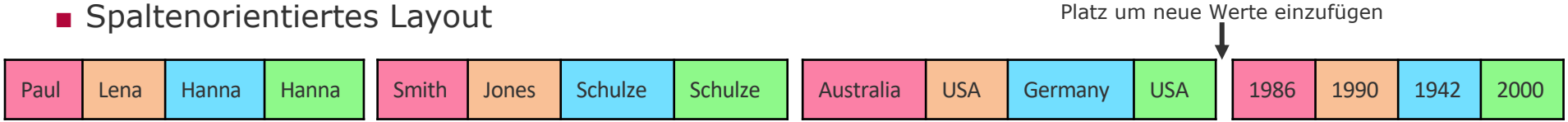
First Name	Last Name	Country	Year of Birth
Paul	Smith	Australia	1986
Lena	Jones	USA	1990
Hanna	Schulze	Germany	1942
Hanna	Schulze	USA	2000

Tabellenrepräsentation

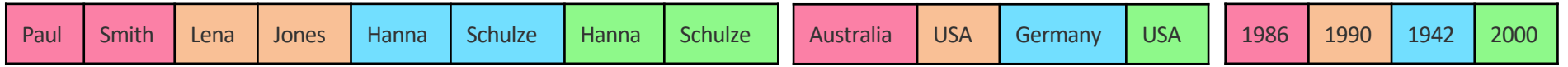
- Zeilenorientiertes Layout



- Spaltenorientiertes Layout



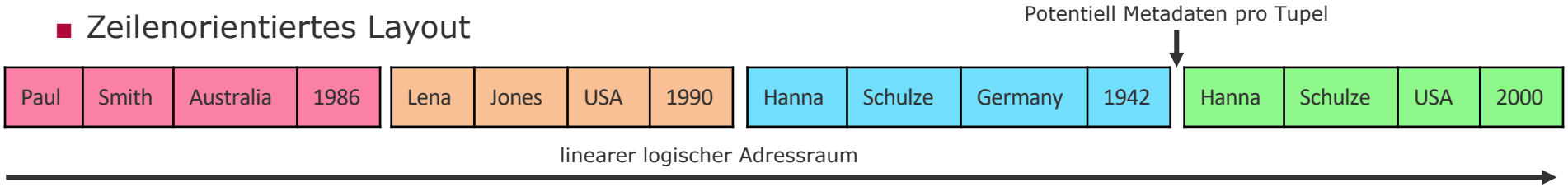
- Hybrides Layout (in Attributgruppen)



linearer logischer Adressraum

Tabellenrepräsentation
Zeilenorientiertes Layout

- Zeilenorientiertes Layout



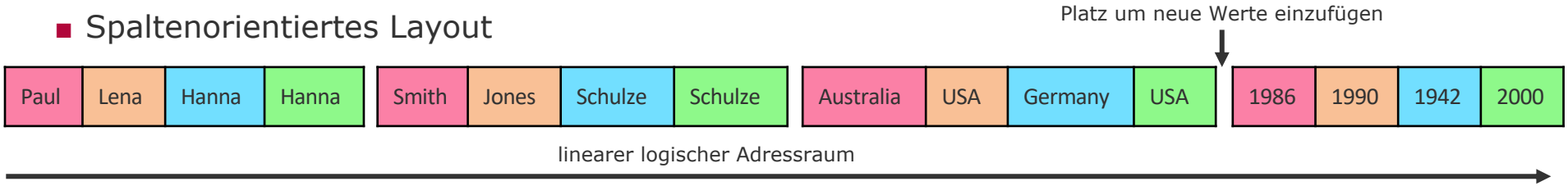
- Speichert alle Attributwerte eines Tupels zusammenhängend

  (bei variabel langen Werten oder sehr großen Objekten zum Teil Verweise/Zeiger)

- Gut geeignet für OLTP-Lasten, in denen Transaktionen auf einzelnen Tupeln arbeiten (Zum effizienten Suchen wird ein Index benötigt!) oder viele neue Tupel einfügen
- Nutzen Tupel-At-A-Time-Iteratormodell

Tabellenrepräsentation
Spaltenorientiertes Layout

- Spaltenorientiertes Layout



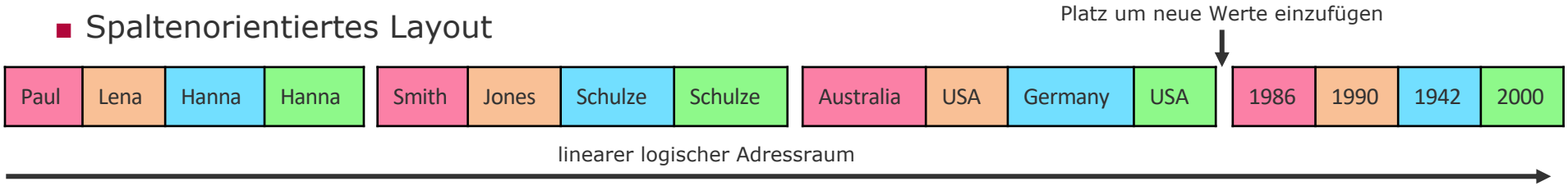
- Speichert alle Attributwerte eines Attributs zusammenhängend

  (bei variabel langen Werten oder sehr großen Objekten zum Teil Verweise/Zeiger)

- Gut geeignet für OLAP-Lasten, in denen Leseanfragen große Teile weniger Attribute durchsuchen
- Nutzen Vector/Block-At-A-Time-Iteratormodell

Tabellenrepräsentation
Spaltenorientiertes Layout

- Spaltenorientiertes Layout



- Tupelrekonstruktion
  - **Variante 1: Offset fester Länge**
  - Variante 2: Explizit gespeichert ID

- **Bessere Komprimierung als zeilenorientiertes Layout** (nächste Vorlesung)

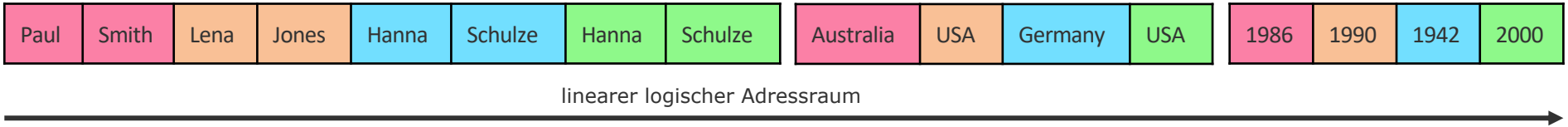
Tabellenrepräsentation
Spaltenorientiertes Layout - Geschichte

- 1970er: Cantor DBMS
- 1980er: Setrag Khoshafian „A Query Processing Strategy for the Decomposed Storage Model“
- 1990er: SybaseIQ (Heute: SAP IQ)
- 2000er: Vertica, MonetDB
- 2010er: **Hyrise** (neue Version), SAP HANA, Cloudera Impala, Amazon Redshift, MemSQL

Andy Pavlo: „Advanced Database Systems: Storage Models & Data Layout (Spring 2019)“
https://15721.courses.cs.cmu.edu/spring2019/schedule.html

Tabellenrepräsentation
Hybrides Layout

- Hybrides Layout

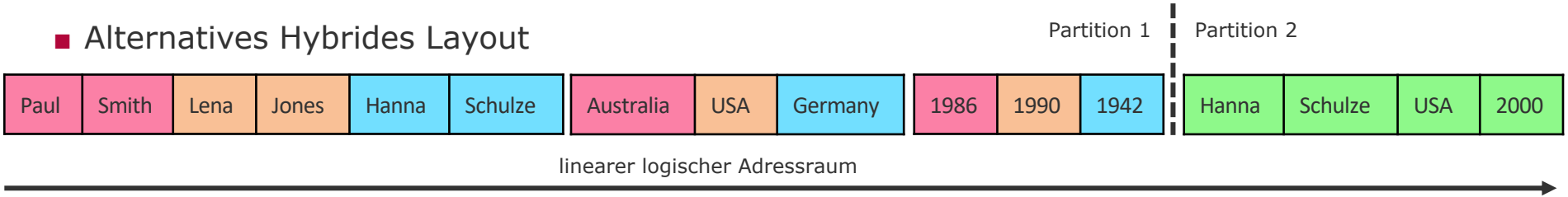


- Speichert alle Attributwerte von Attributgruppen zusammenhängend
  (bei variabel langen Werten oder sehr großen Objekten zum Teil Verweise/Zeiger)

- Attributgruppen einer Tabelle können sich pro Partition unterscheiden
- Idee: optimierte Speicherung der Attribute in Abhängigkeit des Zugriffsmusters
  (Aber: Erhöhte Komplexität bei Query-Optimierung und Ausführungs-Engine)

Tabellenrepräsentation
Hybrides Layout – verschiedene Attributgruppen

- Alternatives Hybrides Layout



- Idee:
  - Sich nicht mehr ändernde Tupel werden für Analysen optimiert gespeichert (Partition 1)
  - Neue Tupel werden zeilenorientiert gespeichert (Partition 2)

- Änderung des Layouts im Laufe der Zeit in Abhängigkeit des Zugriffsmusters (Adaptive/Autonomous/„Self-Tuning"/„Self-Driving" DBs)

Tabellenrepräsentation
Hybrides Layout – Hyrise

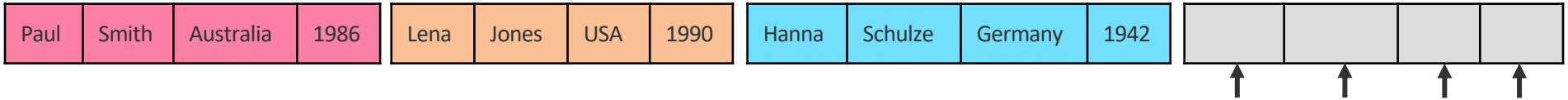
- Martin Grund „HYRISE - A Main Memory Hybrid Storage Engine“ (2010)
  - http://www.vldb.org/pvldb/vol4/p105-grund.pdf
  - Am HPI für Forschungszwecke entwickelte Hauptspeicherdatenbank mit hybridem Speicherformat https://github.com/hyrise/hyrise-v1

Zeilenoperation
Beispiel – Neues Tupel

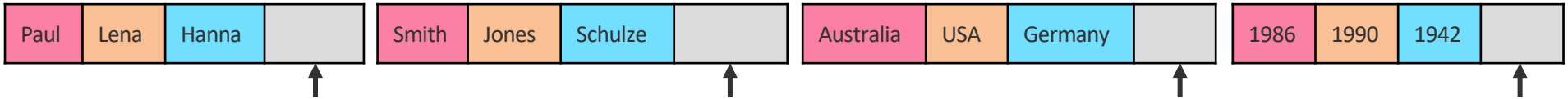
Hanna	Schulze	USA	2000

einfügen

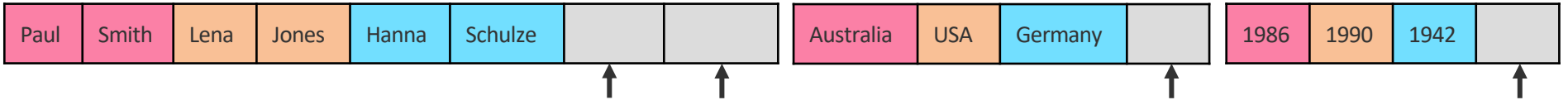
- Zeilenorientiertes Layout



- Spaltenorientiertes Layout

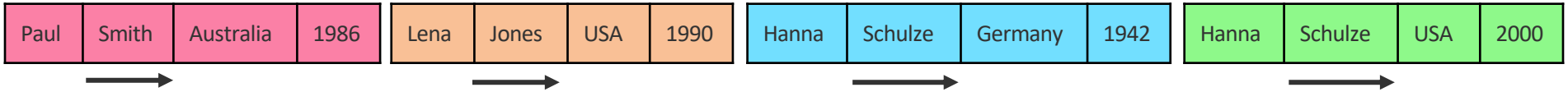


- Hybrides Layout

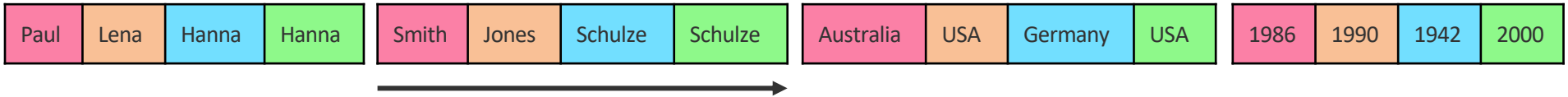


Spaltenoperation
Beispiel – Tabelle nach Nachname filtern

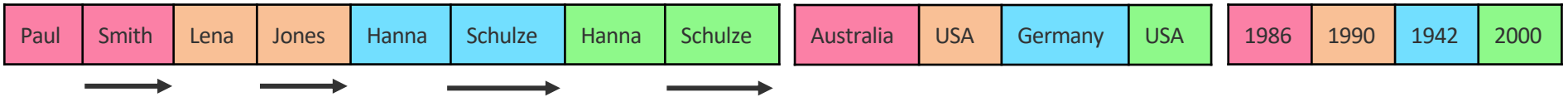
- Zeilenorientiertes Layout



- Spaltenorientiertes Layout

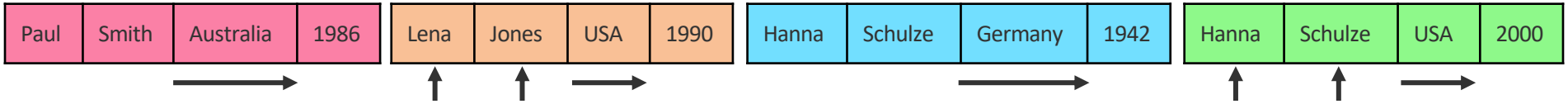


- Hybrides Layout

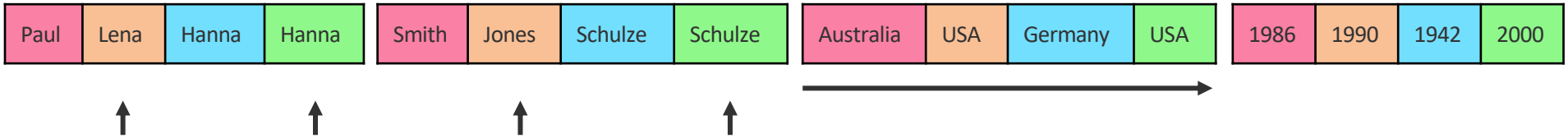


Kombinierte Operationen
Beispiel – Namen aller Personen aus den USA

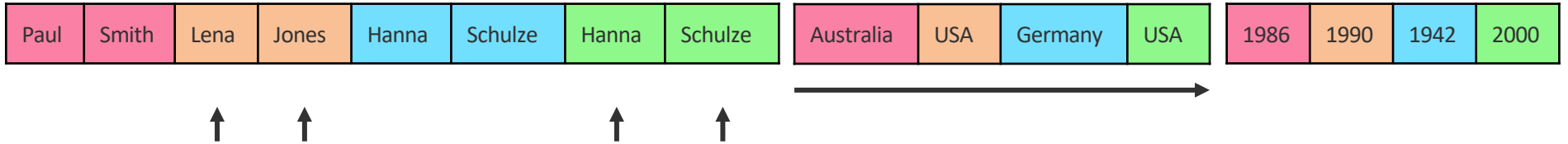
- Zeilenorientiertes Layout



- Spaltenorientiertes Layout



- Hybrides Layout



Festplattenbasierte zeilenorientierte Datenbanken
Motivation

- Festplattenbasierung und Zeilenorientierung sind historisch gewachsen
  - Historische Entwicklung von Datenbanksystemen ist durch den Umstand begrenzter Hardwareressourcen geprägt z.B. Uniprozessoren, kleiner Hauptspeicher

    → Datenbanken mussten auf Festplatte gespeichert werden
  - Zeilenorientierung besser für OLTP als historisch wichtigere Anfrageklasse

Festplattenbasierte zeilenorientierte Datenbanken
Tabellenrepräsentation

- Primärer Speicherplatz der Daten ist die Festplatte
- Daten werden zur Anfragebearbeitung in den Hauptspeicher geladen
- Daten sind in **Blöcke** fester Größe (meist 1 – 16 KiB) unterteilt

Festplattenbasierte zeilenorientierte Datenbanken
Blocklayout

- Einfacher Fall: Datentypen fester Länge
- Direkter Zugriff innerhalb des Blocks, aber ungenutzte Bytes möglich
- Zwei Herausforderungen:
  - Sollen Einträge über Blockgrenzen hinausgehen?
  - Wie löscht man Einträge?

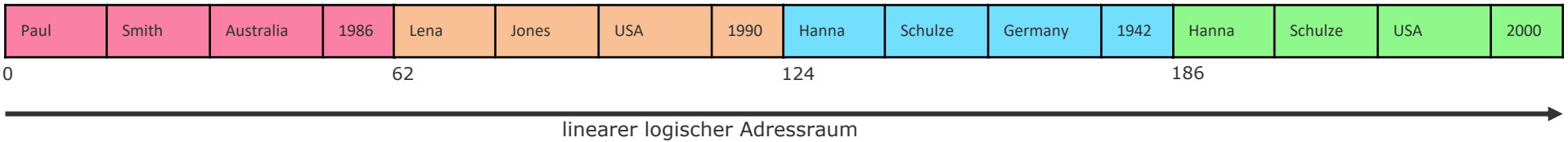
Festplattenbasierte zeilenorientierte Datenbanken
Blocklayout - Datentypen fester Länge

- Sollen Einträge über Blockgrenzen hinausgehen? In der Regel nicht.
- Größe eines Eintrags:

First Name	20 Bytes
Last Name	20 Bytes
Country	20 Bytes
Year Of Birth	2 Bytes

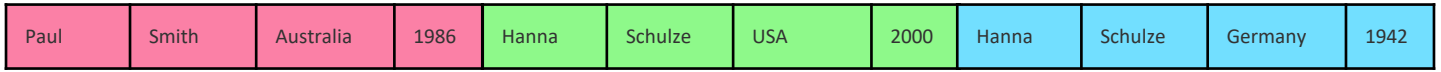
Gesamtgröße	62 Bytes

→ 132 Einträge pro 8 KiB Block



Festplattenbasierte zeilenorientierte Datenbanken
Blocklayout - Datentypen fester Länge

- Wie löscht man Einträge?
  - Andere Einträge in die Lücken kopieren



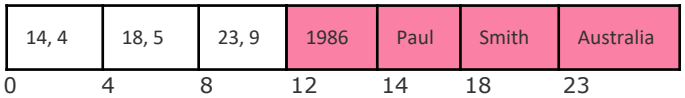
  - **Zukünftig eingefügte Einträge füllen Lücken**
    Blockheader speichert erste Lücke (erste Lücke verweist auf Zweite, ...)

Festplattenbasierte zeilenorientierte Datenbanken
Blocklayout

- Datentypen variabler Länge und NULL-Werte → Einträge variabler Länge
- Zwei Herausforderungen/Probleme:
  - **Einzelne Attributwerte** eines Eintrags müssen effizient extrahierbar sein
  - **Einzelne Einträge** innerhalb des Blockes müssen effizient extrahierbar sein

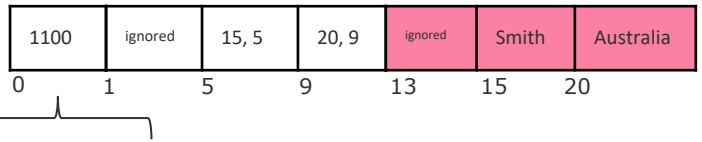
Festplattenbasierte zeilenorientierte Datenbanken
Blocklayout - Datentypen variabler Länge und NULL-Werte

- **Einzelne Attributwerte** eines Eintrags müssen effizient extrahierbar sein (verschiedene Implementierungsmöglichkeiten, hier Lehrbuchbeispiel)
  - Datentypen variabler Länge: Tupel (Offset, Länge) verweisen auf den Attributwert
  - (Datentypen fester Länge ohne zusätzliche Metadaten und am Anfang für direkten Zugriff)



In der Praxis Alignment möglich.

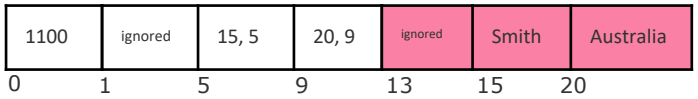
  - Verschiedene Kompromisse zwischen Effizienz und Speicherverbrauch für NULL-Bitmap:
    1. pro Speicherverbrauch: keine Daten bei gesetztem NULL-Bit
    2. pro Effizienz: Daten bei gesetztem NULL-Bit ignorieren



NULL-Bitmap in einem Byte gespeichert. (Reihenfolge entspricht hier der physischen Speicherung.)

Festplattenbasierte zeilenorientierte Datenbanken
Blocklayout - Datentypen variabler Länge und NULL-Werte

Beispiel einer Tupelrepräsentation, die effizienten Zugriff auf alle Attributwerte zulässt



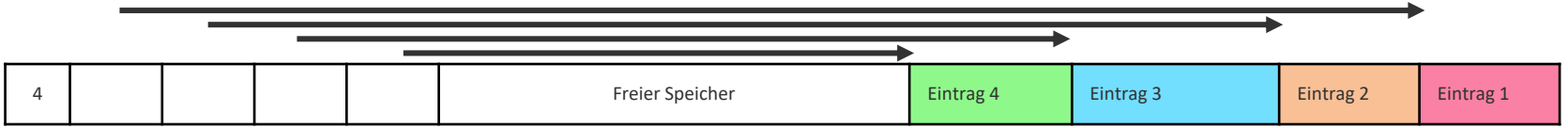
- Die vier NULL-Bits 1, 1, 0 und 0 stehen für die Attribute YearOfBirth, FirstName, LastName und Country
- Datentypen variabler Länge: Tupel (Offset, Länge) verweisen auf den Attributwert
  - gesetztes NULL-Bit für FirstName → Meta-Daten (hier: Bytes an Offset 1 - 4) werden ignoriert
    → erlaubt direkten Zugriff auf Meta-Daten für nachfolgende Attributwerte variabler Länge
  - Attributwert kann bei gesetztem NULL-Bit hingegen weggelassen werden
- Datentypen fester Länge (hier: YearOfBirth)
  - ohne zusätzliche Metadaten (Anzahl der Bytes (hier 2) durch Schema bekannt)
  - am Anfang für direkten Zugriff
  - gesetztes NULL-Bit → Daten (hier: Bytes an Offset 13 und 14) werden ignoriert
    → erlaubt direkten Zugriff für potentiell nachfolgende Attributwerte fester Länge

Festplattenbasierte zeilenorientierte Datenbanken
Blocklayout - Einträge variabler Länge

- **Einzelne Einträge** innerhalb des Blockes müssen effizient extrahierbar sein

  Slotted-Page Struktur: Block-Header speichert:

  - Anzahl der Einträge (oder Beginn des freien Speichers im Block)
  - (Das Ende des freien Speichers im Block)
  - Verweise auf alle Einträge
    - Indexe verweisen auf Verweise im Block-Header und nicht auf die Einträge selbst; diese Indirektion erlaubt es Einträge zu verschieben um Fragmentierung zu verhindern
    - Können auch gelöschte Einträge abbilden



Festplattenbasierte zeilenorientierte Datenbanken
Blocklayout - PostgreSQL

- The Internals of PostgreSQL
  http://www.interdb.jp/pg/pgsql01.html

- PostgreSQL 11 Documentation Chapter 68. Database Physical Storage
  https://www.postgresql.org/docs/11/storage.html

Festplattenbasierte zeilenorientierte Datenbanken
Buffermanagement

- Das Datenbanksystem nutzt (in der Regel) einen „Buffer"/Puffer um Blöcke im Hauptspeicher zu cachen
- Ziel: Anzahl der Festplattenzugriffe minimieren
- Alle Speicherzugriffe der Ausführungseinheit gehen über den Buffermanager
- Buffermanager prüft, ob Block bereits im Speicher ist
  - Falls nicht, liest der Buffermanager den Block in der Buffer
    - Falls kein Platz, muss ein anderer Block ersetzt werden
    - (Nur) falls der zu ersetzende Block verändert wurde, muss er zurück auf die Festplatte geschrieben werden

Festplattenbasierte zeilenorientierte Datenbanken
Buffermanager vs. Betriebssystem (virtuelle Speicherverwaltung)

Datenbank hat mehr Informationen als Betriebssystem

- Ersetzungsstrategien
  - LRU (Least recently used)
  - Wenn letzter Eintrag verarbeitet wurde (Toss-immediate)
  - MRU (Most recently used)

  } nützlich für Nested-Loop-Join
- Blockprefetching
- Blockpinning
- Forciertes Schreiben auf die Festplatte

Festplattenbasierte zeilenorientierte Datenbanken
Anfragebearbeitung

https://15445.courses.cs.cmu.edu/fall2018/slides/10-queryprocessing.pdf

- Geschwindigkeitsengpass ist Festplattenzugriff
  - Sequentielle Zugriffe verwenden
  - Große Zwischenergebnisse vermeiden, die den Hauptspeicher unnötig füllen
- Verwenden traditionell das Tupel-At-A-Time-Iteratormodell
  (auch Volcano- oder Pipeline-Modell)
  - Jeder Operator der Anfrageplans implementiert next()-Funktion
  - Bei jedem Aufruf gibt der Operator ein einziges Tupel zurück oder signalisiert, dass es keine weiteren Tupel gibt
  - Operatoren rufen next() auf ihren Kindern um deren Tupel abzufragen und zu verarbeiten
  - Top-Down-Planverarbeitung
  - Unterstützt Pipelining, aber einige Operatoren (Joins, Subqueries, Sortierung) blockieren bis alle Tupel der Kinder abgefragt wurden

(Spaltenbasierte) Hauptspeicherdatenbanken
Motivation

- Durch steigende DRAM-Kapazitäten passen die meisten Datenbanken in den Hauptspeicher
  - **Strukturierte (z.B. relationale) Datensätze** sind kleiner ← unser Fokus
  - Unstrukturierte oder nur teilweise strukturierte Datensätze sind größer

- Insbesondere analytische Anforderungen an Datenbanksystemen sind gewachsen
- Heute: spezialisierte Hauptspeicherdatenbanken
  - Zeilenorientierung besser für OLTP
  - **Spaltenorientierung** besser für OLAP und **Mixed Workloads** ← unser Fokus (und hybrides Layout)

(Spaltenbasierte) Hauptspeicherdatenbanken
Hauptspeicher- vs. Festplattendatenbanken

- Hauptspeicherzugriff ist schneller
- Hauptspeicher ist üblicherweise flüchtig (NVRAM natürlich nicht)

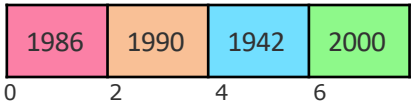
  (Hauptspeicherdatenbanken benötigen Festplatten für das Log und zur Wiederherstellung)
- Festplatten sind blockorientiert mit großen Fixkosten für Zugriffe

  (sequenzieller Zugriff ist im Hauptspeicher auch schneller als zufälliger Zugriff (Caching))
- **Datenrepräsentation und -zugriff für dauerhafte Speicherung im Hauptspeicher optimieren**, zum Beispiel keine Slotted-Page Struktur, kein Buffermanager, anderes Ausführungsmodell

Stavros Harizopoulos et al. „OLTP Through the Looking Glass, and What We Found There“ (2008)
Hector Garcia-Molina et al. „Main Memory Database Systems: An Overview“ (1992)

Spaltenbasierte Hauptspeicherdatenbanken
Tabellenrepräsentation

- Ziel: (schneller Scan und) **direkter/effizienter** (konstante Zeit) **Zugriff** auf Attributwerte
- Datenblöcke/Spalten speichern Attributwerte fester Länge direkt oder Verweise (mit fester Länge) auf Attributwerte bei variabler Länge
  - Spalte wird als Vektor/Array implementiert



  - Spalten variabler Länge speichern logische Verweise (Offset) oder Pointer

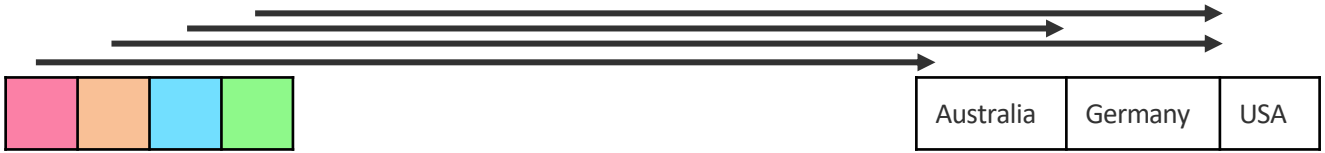
→ Indirektion verursacht zusätzlichen Speicherzugriff (Speicher-Effizienz-Kompromiss)

(Maximale Attributwertgröße und „Small/Short String Optimization“ (SSO) als Alternativen)

  - Verweise müssen nicht auf benachbarte Speicherregion (gleicher Block) zeigen

Spaltenbasierte Hauptspeicherdatenbanken
Tabellenrepräsentation

- Dictionary-Kodierung erzeugt Attributvektoren fester Länge (siehe Vorlesung „Datenkompression“)



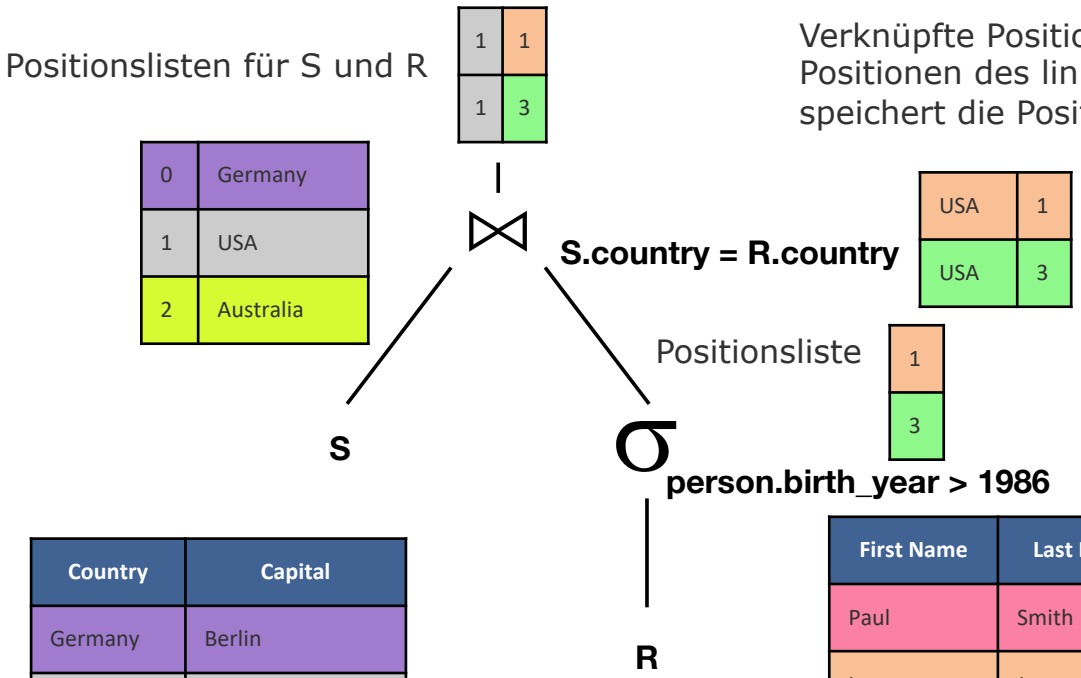
- Zusätzliche NULL-Bitmap pro Spalte, falls die Spalte NULL-Werte erlaubt

- Löschen von Einträgen in der Regel durch Invalidierung; zusätzlicher Vektor markiert, ob Eintrag gültig ist oder nicht (Details in Vorlesung „Weitere Datenbankoptimierungen“)

Spaltenbasierte Hauptspeicherdatenbanken
Anfragebearbeitung

- Traditionelles Tupel-At-A-Time-Iteratormodell verursacht zu viele Funktionsaufrufe (pro Operator pro Tupel) und ist nicht Cache-effizient bei Spaltenbasierung
- Verwenden das Vektor-At-A-Time-Iteratormodell und späte Materialisierung
  - Operatoren arbeiten auf Spalten/Vektoren
  - (Top-Down oder Bottom-Up möglich)
  - Verarbeitung kompletter Spalten kann Caches überfüllen
    → (Top-Down) Block-At-A-Time-Iteratormodell verarbeiten Spalten blockweise
- Späte Materialisierung
  - Zwischenergebnisse speichern Positionslisten (logische Tabellen)
  - Attributwerte werden erst geladen/materialisiert, wenn sie für die Verarbeitung durch Operatoren gebraucht werden

Spaltenbasierte Hauptspeicherdatenbanken
Anfragebearbeitung - späte Materialisierung: Beispiel



Verknüpfte Positionsliste: die linke Seite speichert die Positionen des linken Joinpartners, die rechte Seite speichert die Positionen des rechten Joinpartners

Unter Verwendung der Positionsliste kann auf die Joinattribute zugegriffen werden

Ergebnis der Selektion ist eine Positionsliste mit einem Verweis auf die Originaltabelle

Country	Capital
Germany	Berlin
USA	Washington
Australia	Canberra

First Name	Last Name	Country	Year of Birth
Paul	Smith	Australia	1986
Lena	Jones	USA	1990
Hanna	Schulze	Germany	1942
Hanna	Schulze	USA	2000

Vergleich

Spaltenbasierte Hauptspeicherdatenbanken

- Geschwindigkeitsengpass: Cacheeffizienz und Funktionsaufrufe (+ CPU-Kosten)
- Effizienter („direkter“) Zugriff auf Attributwerte
- Vektor/Block-At-A-Time-Iteratormodell

Festplattenbasierte zeilenorientierte DBs

- Geschwindigkeitsengpass: Festplattenzugriff
- Zugriff auf Attributwerte über Buffermanager und Slotted-Page Struktur
- Tupel-At-A-Time-Iteratormodell

Tabellenrepräsentation

- Zwei-dimensionale Tabellen müssen auf ein-dimensionalen Speicheradressraum abgebildet werden
- Ziel: Speicherhierarchie bestmöglich ausnutzen

- Das bestes Datenlayout hängt vom Workload (Menge aller Anfragen) ab
- Datenkompression ist ein zusätzlicher Einflussfaktor bei der Wahl des Datenlayouts

- Datenlayout und primärer Speicherplatz der Daten haben großen Einfluss auf Implementierung des Datenbanksystems