



# Datenbanken: Weitere Datenbankoptimierungen

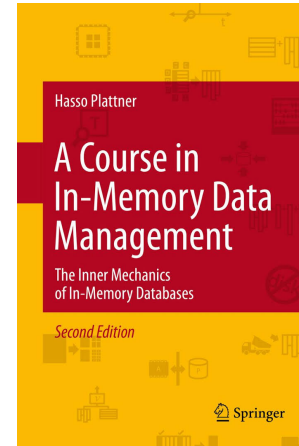
Dr. Matthias Uflacker, Stefan Halfpap, Dr. Werner Sinzig

5. Juni 2019

- Einführung zu Unternehmensanwendungen (2 Vorlesungen)
- Einführung zu relationalen Datenbanken und Anfrageverarbeitung (2 Vorlesungen)
- Grundlagen des IT-gestützten Rechnungswesens und der Planung (3 Vorlesungen)
- **Grundlagen von (spaltenorientierten) Hauptspeicherdatenbanken (5 Vorlesungen)**
- Trends in Hauptspeicherdatenbanken (4 Vorlesungen)
- Klausur

- Architektur einer spaltenbasierten HauptspeicherDB
- Datenbankoptimierungen
  - Insert-Only-Ansatz
  - Main-Delta- und Chunk-Architektur
  - History-Partition
  - Hot-Cold Datenpartitionierung
  - Replikation
- Zusammenfassung

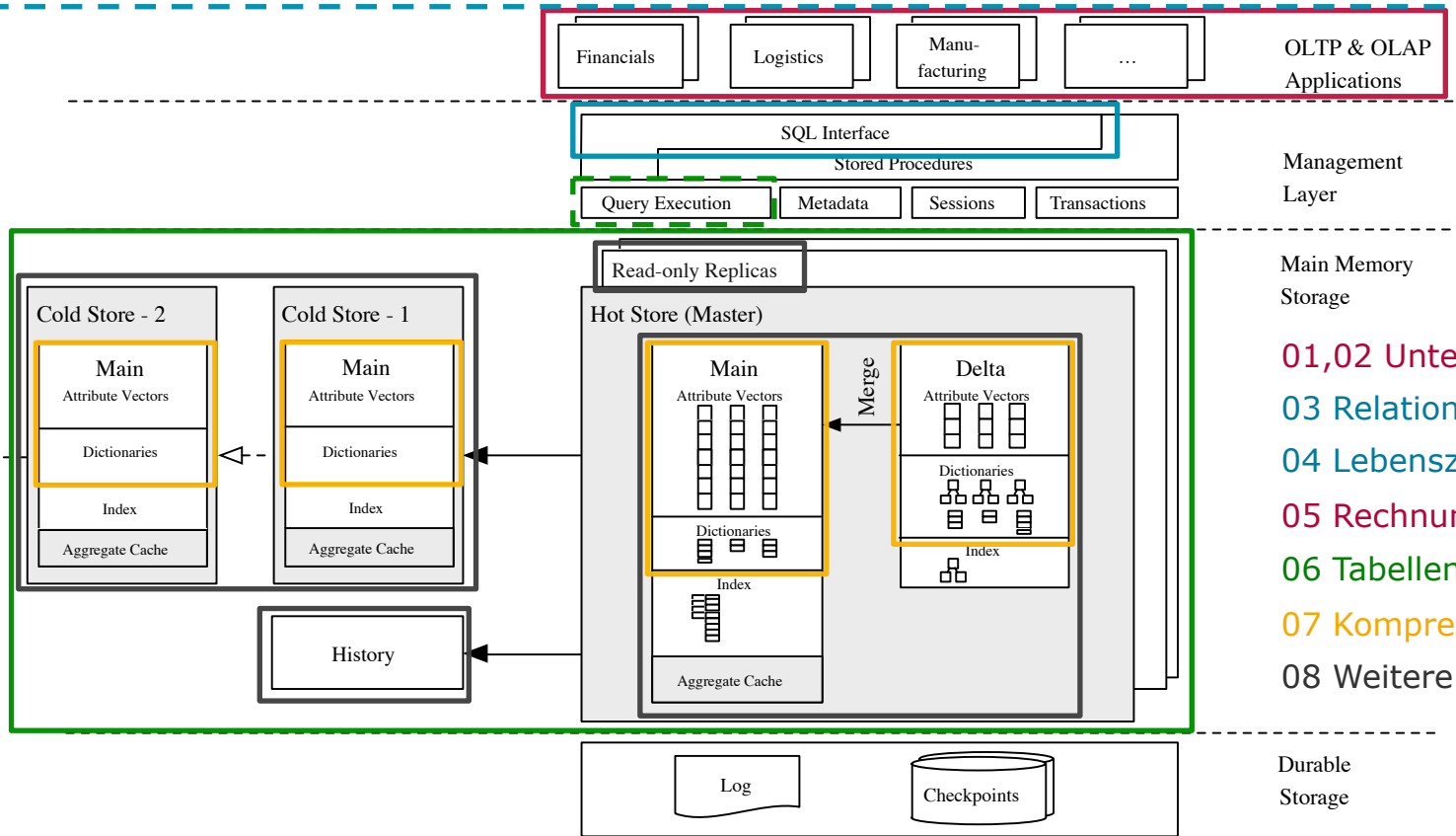
- Hasso Plattner „A Course in In-Memory Data Management“



- Hasso Plattner „The Impact of Columnar In-Memory Databases on Enterprise Systems“ (2014)

# Architektur einer spaltenbasierten HauptspeicherDB

Hasso Plattner „The Impact of Columnar In-Memory Databases on Enterprise Systems“ (2014)



01,02 Unternehmensanwendungen

03 Relationales Modell, SQL

04 Lebenszyklus einer Query

05 Rechnungswesen, Planung

06 Tabellenrepräsentation

07 Kompression

08 Weitere Optimierungen

# Architektur einer spaltenbasierten HauptspeicherDB

## Datenbankoptimierungen

---

- Insert-Only-Ansatz
- Main-Delta- und Chunk-Architektur
- History-Partition
- Hot-Cold Datenpartitionierung
- Replikation

- Gelöschte Daten erzeugen Lücken in den Datenstrukturen
- Reorganisation/Lückenschluss ist in spaltenorientierten Datenbanken teuer, da alle Spalten umorganisiert werden müssen und Datenkompression die Umorganisation zusätzlich erschweren kann

## Insert-Only-Ansatz

- Gelöschte Daten werden nicht überschrieben, sondern als ungültig markiert
- UPDATES sind als INSERT und DELETE umgesetzt
- Verschiedene Implementierungen:
  - Bitmap pro Tabelle, die Gültigkeit einzelner Tupel angibt
  - **Gültigkeitsbereich pro Tupel** → ermöglicht Multi Version Concurrency Control (MVCC) (Details in 10\_DB\_Nebenläufigkeitskontrolle)

- Dictionary-Kompression: Sortiertes vs. unsortiertes Dictionary (Wiederholung)
  - Pro Sortierung:
    - Suche der Wert-ID im Dictionary hat Komplexität  $O(\log(n))$  statt  $O(n)$
    - Bereichsabfragen (range queries) beschleunigen
    - Dictionary kann besser komprimiert werden
  - Contra Sortierung:
    - Einfügen neuer Werte kann eine Reorganisation der Datenstrukturen benötigen (um Sortierung zu bewahren oder wenn sich die Anzahl der benötigten Bits pro Wert-ID verändert)
- Löschen von Tupeln führt zu „Lücken“



# Main-Delta-Architektur

Idee: Zwei verschiedene horizontale Partitionen

- Leseoptimierte **Main-Partition** mit sortiertem Dictionary

- Schreiboptimierte **Delta-Partition** mit unsortiertem Dictionary

Lesen

Schreiben

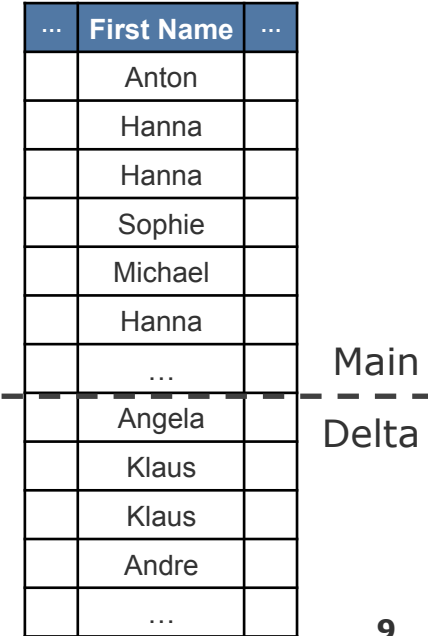
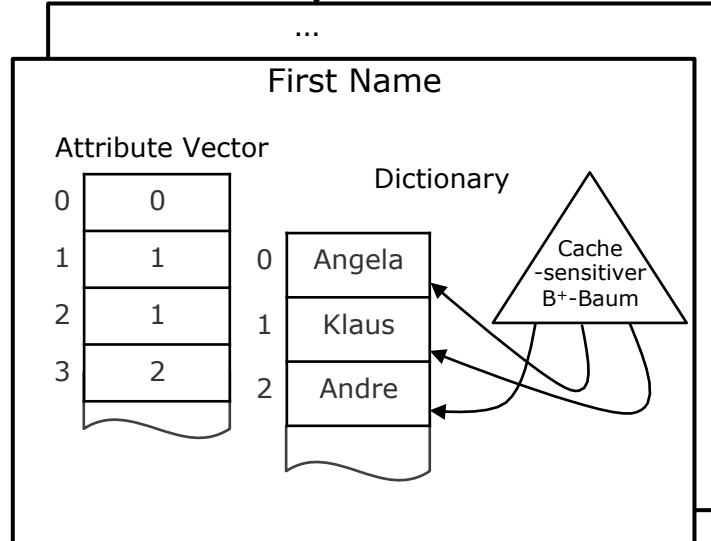
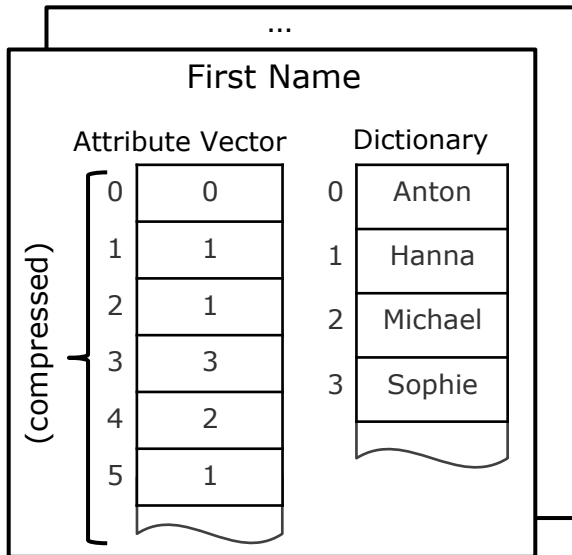


Diagram illustrating the logical view of the data, showing a table with columns 'First Name' and '...'. The table is divided into two horizontal partitions: 'Main' and 'Delta'. The Main partition contains sorted data, and the Delta partition contains unsorted data. The data is as follows:

...	First Name	...
	Anton	
	Hanna	
	Hanna	
	Sophie	
	Michael	
	Hanna	
	...	
	Angela	
	Klaus	
	Klaus	
	Andre	
	...	

Datenrepräsentation im Speicher

Logische Sicht

- Gute Leseperformanz und günstigeres Einfügen
  - Gute Leseperformanz, da Daten komprimiert sind
  - Günstigeres Einfügen, da Dictionary und Attributvektor nicht aufwendig umorganisiert werden müssen
  
- Benötigt mehr Speicher
  - zusätzlicher Baum fürs Dictionary
  - i.d.R. keine Attributvektorkomprimierung

- INSERTs werden im Delta gespeichert
- DELETES erzeugen ungültige Tupel (im Main und Delta)
- UPDATES sind als INSERT und DELETE umgesetzt (Insert-Only-Ansatz)
  -
- Delta wächst kontinuierlich
- Main und Delta bekommen „Lücken“
  -
- Datenreorganisation / Merge / Tuple-Mover notwendig  
(Anfrageoptimierung für zukünftige Queries)

- Asynchroner Prozess in Bezug auf Anfragebearbeitung  
(speicherintensiv, aber optimierbar)
- Angestoßen durch Anzahl der Tupel im Delta oder durch Kostenmodell (oder manuell)
- Schritte:
  - „Merge“ Main- und Delta-Dictionary (Optional: Löschen nicht benötigter Werte)
  - Erstellen der Abbildung: alte Wert-ID → neue Wert-ID (für Main und Delta)
  - (Main-)Attributvektor neu schreiben
  - (Delta-Partition ist nach dem Merge-Prozess leer)

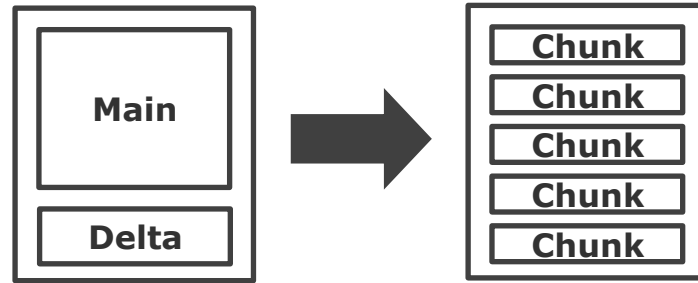
# Main-Delta-Architektur

## Alternative: Chunks

- Motivation: Kosten für Merge-Prozess steigen mit der Zeit

- Idee:

- Mehrere Chunks/Blöcke fester Größe  
(horizontale Partitionen)



- Drei Chunktypen (Phasen werden sequentiell durchlaufen)  
→ komprimierte Chunks sind (im Vergleich zum Main) „stabil“ (kein Merge)



# Main-Delta-Architektur

## Alternative: Chunks - Anmerkungen

---

- Tupel aus komprimierten Chunks können gelöscht/ungültig werden  
→ Merge mehrerer „löchriger“ Chunks
- Potenziell höherer Speicherverbrauch (mehrere gleiche Dictionary-Einträge für unterschiedliche Chunks vs. weniger Bits pro Wert-ID)

- Implementiert in Hyrise



<https://github.com/hyrise/hyrise>

- Insert-Only-Ansatz: Gelöschte Daten werden nicht überschrieben, sondern ungültig
  - + MVCC/Snapshot-Isolation (und Zeitreise-Queries)
  - + Möglicherweise gesetzlich vorgeschrieben
  - Speicherverbrauch
  - Höhere Scan-Kosten
- Ungültige Daten werden (z.B. beim Merge) in die History-Partition geschoben und bei der (normalen) Anfragebearbeitung nicht mehr gelesen

- Fast jede Anwendung folgt dem „working set model“
  - “there is a subset of (pages) that are accessed distinctively more frequently”

Peter Denning “The working set model for program behaviour” (1968)
- In Unternehmensanwendungen: aktuelles vs. erstes Geschäftsjahr
- Hot-Cold-Datenpartitionierung ist die Ausnutzung dieses Wissens bei der Datenspeicherung
- Idee:
  - Partitioniere Daten in Blöcke und speichere Metainformationen zu den Daten pro Block
  - Prüfe bei der Anfragebearbeitung anhand der Metainformationen, ob die Partition relevante Daten enthält und gelesen werden muss



# Current-Historical Datenpartitionierung

## Ein Extremfall von Hot-Cold Datenpartitionierung

### Aktuelle (current) Daten

- **Durch den Anwendungsentwickler** als relevante Daten **definiert**
- Alle veränderbaren Daten
- Häufig zugriffene Daten (hot)
- Auf schnellen Speicher

### Historische/ältere (historical) Daten (≠ gelöschte Daten)

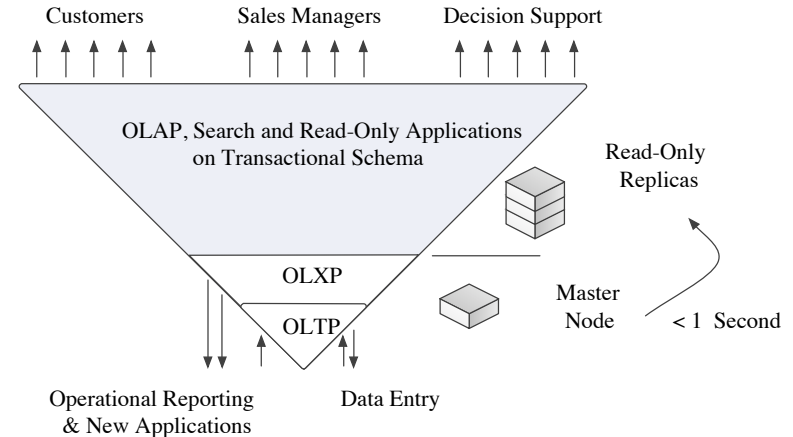
- Die restlichen Daten sind „historisch“
- Daten werden nur noch gelesen
- Selten zugriffene Daten (cold)
- Teilweise auf günstigeren (langsameren) Speicher ausgelagert

### ■ Vorteile:

- Performanzsteigerung durch das Weglassen des Scans der „historischen“ Datenpartition
- Geringere HW-Kosten durch günstigeren Speicher (und effizientere Scanoperation)

- Wie oft wird auf historische Partitionen zugegriffen?
  - Für tägliche Geschäftsprozesse fast nie
  - Die meisten analytischen Anfragen greifen auch nur auf aktuelle Daten zu
- Um unnötige Zugriff auf nicht relevante (historische) Partitionen zu vermeiden (z.B. bei defensiver Programmierung oder Anwendungen, die die Partitionierung nicht kennen), können spezielle Datenstrukturen (**Pruning-Filter**) verwendet werden, die Metainformationen zu den Daten der Partition speichern
  - Platz-effiziente probabilistische Datenstrukturen
  - Fassen die Daten (Werte, Wertebereiche und Kombinationen) von Partitionen zusammen
  - Erkennen unnötigen Zugriff

- Neue Unternehmensanwendungen ..
  - .. locken zunehmend Nutzer an
  - .. stellen zunehmend komplexe Anfragen
  - .. unterstützen interaktive Datenexploration
- .. erfordern Skalierbarkeit



Hasso Plattner „The Impact of Columnar In-Memory Databases on Enterprise Systems“ (2014)

- Verwende Datenbankreplikate (zusätzlicher Computer als Kopien des Datenbanksystems) für Anfragebearbeitung (und zur höheren Verfügbarkeit)
- Replikationsverfahren können klassifiziert werden
- Aktive Master-Replikation: Replikatknoten verarbeiten ausschließlich lesende Anfragen auf Snapshots (konsistenten Kopien) des Masters

- **Aktive** vs. passive Replikation
  - Replikate werden für Anfrageverarbeitung (aktiv) und nicht nur als Backup (passiv) verwendet
- **Master-** vs. Multimaster-/Group-Replikation
  - Ein einzelner Computer (Master) ist für die transaktionale Verarbeitung zuständig (keine teuren verteilten Transaktionen)
- Weitere Spezialisierung/Klassifikation von Replikationsverfahren:
  - Sofortige vs. „träge“ Synchronisation der Replikate
  - Logische vs. physische Synchronisationsinformationen
  - Homogene vs. heterogene Replikate

- Sofortige vs. „träge“ Synchronisation der Replikate
  - Sofortig („eager“): Aktualisiere Replikate als Teil der Transaktion
  - „Träge“ („lazy“): Aktualisiere Replikate asynchron zur Transaktion
  
- Logische vs. physische Synchronisationsinformationen
  - Logisch: Operationen, z.B. SQL-Anfragen
  - Physisch: Datenänderungen
  
- Homogene vs. heterogene Replikate
  - Homogen: Replikate als genaues „Spiegelbild“ des Masters
  - Heterogen: Replikate speichern Teilmengen der Daten oder optimierte Datenstrukturen

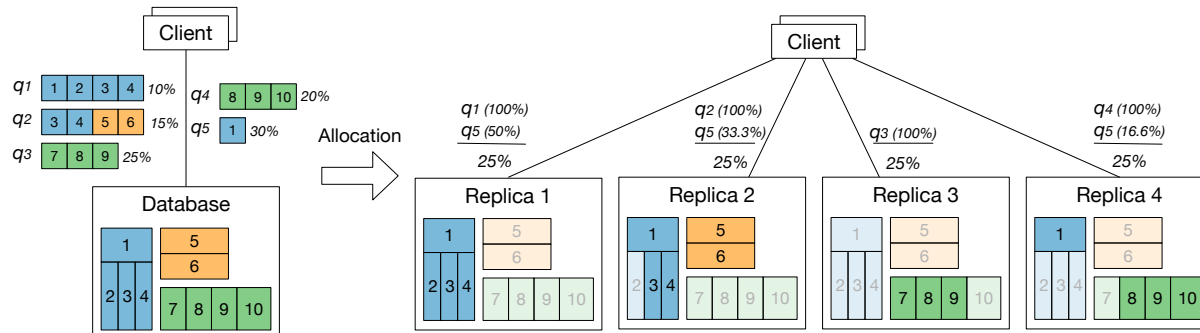
# Replikation

## Heterogene Replikation – Partielle Replikation

Replikation der Daten basierend auf Anfragen, die Teilmengen von Daten benötigen und für einen bestimmten Workload-Anteil verantwortlich sind

- Partielle Replikatknoten speichern nur Teilmengen der Daten und verbrauchen daher weniger Speicherplatz
- Partielle Replikatknoten können nur Teilmenge der Anfragen (für die alle relevanten Daten gespeichert sind) bearbeiten

→ Effiziente Skalierung



- Main-Delta-Architektur mit leseoptimierter Main- und schreiboptimierter Delta-Partition
- Chunk-Architektur als Alternative um steigende Merge-Kosten zu meiden
- History-Partition speichert gelöschte Daten
- Historische Partition speichert nicht mehr „relevante“ Daten, die für die meisten Anfragen nicht mehr gelesen werden müssen
- Replikation ist eine Möglichkeit Leseanfragen auf zusätzlichen Computern zu skalieren