Technical Deep-Dive in a Column-Oriented In-Memory Database

Martin Faust
Martin.Faust@hpi.uni-potsdam.de

Research Group of Prof. Hasso Plattner
Hasso Plattner Institute for Software Engineering
University of Potsdam
Goals

Deep technical understanding of a column-oriented, dictionary-encoded in-memory database and its application in enterprise computing

Chapters

- The future of enterprise computing
- Foundations of database storage techniques
- In-memory database operators
- Advanced database storage techniques
- Implications on Application Development
Learning Map of our Online Lecture @ openHPI.de

The Future of Enterprise Computing

Introduction → New Requirements for Enterprise Computing

Enterprise Application Characteristics → Changes in Hardware

Blueprint SanssouciDB

Foundations of Database Storage Techniques

Partitioning

In-Memory Database Operators

Dictionary Encoding → Compression

Data Layout Row, Column, Hybrid

Foundations for a New Enterprise Application Development Era

Advanced Database Storage Techniques

DELETE INSERT UPDATE

Tuple Reconstruction

Scan Performance

Parallel Data Processing

Workload Management

Application Development

In-Memory Database Operators

Differential Buffer → Merge

Insert-Only Time Travel

On-The-Fly Database Reorganization

Logging → Recovery

Parallel JOIN → Aggregate Functions

Parallel Aggregate Functions

Implications → Dunning

Views

Handling Business Objects

ByPass Solution
Chapter 1:

The Future of Enterprise Computing
New Requirements for Enterprise Computing

- Sensors
- Events
- Structured + unstructured data
- Social networks
- ...

...
Enterprise Application Characteristics
Modern enterprise resource planning (ERP) systems are challenged by mixed workloads, including OLAP-style queries. For example:

- OLTP-style: create sales order, invoice, accounting documents, display customer master data or sales order
- OLAP-style: dunning, available-to-promise, cross selling, operational reporting (list open sales orders)

But: Today’s data management systems are optimized either for daily transactional or analytical workloads storing their data along rows or columns
Drawbacks of the Separation

- OLAP systems do not have the latest data
- OLAP systems only have predefined subset of the data
- Cost-intensive ETL processes have to sync both systems
- Separation introduces data redundancy
- Different data schemas introduce complexity for applications combining sources
Enterprise Workloads are Read-Mostly

- Workload in enterprise applications constitutes of
  - Mainly read queries (OLTP 83%, OLAP 94%)
  - Many queries access large sets of data
Few Updates in OLTP

<table>
<thead>
<tr>
<th>Percentage of rows updated</th>
<th>CPG</th>
<th>High Tech</th>
<th>Logistics</th>
<th>Discrete M.</th>
<th>Banking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6%</td>
<td>14%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Financial Accounting
Vision

Combine OLTP and OLAP data using modern hardware and database systems to create a single source of truth, enable real-time analytics and simplify applications and database structures.

Additionally,

- Extraction, transformation, and loading (ETL) processes
- Pre-computed aggregates and materialized views

become (almost) obsolete.
Enterprise Data Characteristics

- Many columns are not used even once
- Many columns have a low cardinality of values
- NULL values/default values are dominant
- Sparse distribution facilitates high compression

Standard enterprise software data is **sparse and wide.**
Many Columns are not Used Even Once

55% unused columns per company in average
40% unused columns across all analyzed companies
Changes in Hardware
Changes in Hardware...

... give an opportunity to re-think the assumptions of yesterday because of what is possible today.

- Multi-Core Architecture (96 cores per server)
- Large main memories: 2TB/blade
- One blade ~$50.000 = 1 enterprise class server
- Parallel scaling across blades

- Main Memory becomes cheaper and larger
- 64 bit address space
- 2TB in current servers
- Cost-performance ratio rapidly declining
- Memory hierarchies
In the Meantime
Research has come up with...

... several advances in software for processing data

- Column-oriented data organization
  (the column store)
  - **Sequential** scans allow best bandwidth utilization between CPU cores and memory
  - **Independence** of tuples allows easy partitioning and therefore parallel processing

- Lightweight Compression
  - Reducing data amount, while...
  - Increasing processing speed through late materialization

- And more, e.g., parallel scan/join/aggregation
A Blueprint of SanssouciDB
SanssouciDB: An In-Memory Database for Enterprise Applications

In-Memory Database (IMDB)
- Data resides **permanently** in main memory
- Main Memory is the **primary** “persistence”
- Still: logging and recovery to/from **flash**
- Main memory access is the new **bottleneck**
- Cache-conscious algorithms/data structures are **crucial** (locality is king)
Chapter 2:
Foundations of Database Storage Techniques
Learning Map of our Online Lecture @ openHPI.de

The Future of Enterprise Computing

Introduction → New Requirements for Enterprise Computing

Enterprise Application Characteristics → Changes in Hardware

Blueprint SanssouciDB → Partitioning

Advanced Database Storage Techniques

In-Memory Database Operators

Dictionary Encoding → Compression

Data Layout Row, Column, Hybrid

Partitioning

Foundations for a New Enterprise Application Development Era

Foundations of Database Storage Techniques

DELETE INSERT UPDATE

Tuple Reconstruction

Scan Performance

Parallel Data Processing

Workload Management

Application Development

Logging → Recovery

Differential Buffer

Merge

Insert-Only Time Travel

On-The-Fly Database Reorganization

Indices

JOIN

Parallel JOIN

Aggregate Functions

Parallel Aggregate Functions

Implications

Dunning

Views

Handling Business Objects

ByPass Solution
Data Layout in Main Memory
Memory in today's computers has a linear address layout: addresses start at 0x0 and go to 0xFFFFFFFFFFFFFFFF for 64-bit systems.

- Each process has its own virtual address space.
- Virtual memory allocated by the program can distribute over multiple physical memory locations.
- Address translation is done in hardware by the CPU.
Memory Basics (2)

- Memory layout is **only linear**
- Every higher-dimensional access (like two-dimensional database tables) is mapped to this linear band
Memory Hierarchy
Physical Data Representation

- **Row store:**
  - Rows are stored consecutively
  - Optimal for row-wise access (e.g. SELECT *)

- **Column store:**
  - Columns are stored consecutively
  - Optimal for attribute focused access (e.g. SUM, GROUP BY)

- Note: concept is **independent** from storage type

![Diagram of Row-Store and Column-store](image-url)
Row Data Layout

- Data is stored tuple-wise
- Leverage co-location of attributes for a single tuple
- Low cost for reconstruction, but higher cost for sequential scan of a single attribute
Columnar Data Layout

- Data is stored attribute-wise
- Leverage sequential scan-speed in main memory for predicate evaluation
- Tuple reconstruction is more expensive
Dictionary Encoding
Motivation

- Main memory access is the new bottleneck
- Idea: **Trade** CPU time to compress and decompress data
- Compression reduces number of memory accesses
- Leads to **less** cache misses due to more information on a cache line
- Operation directly on compressed data
- Offsetting with bit-encoded fixed-length data types
- Based on limited value domain
Dictionary Encoding Example

8 billion humans

- Attributes
  - first name
  - last name
  - gender
  - country
  - city
  - birthday
  - → 200 byte

- Each attribute is stored dictionary encoded
## Sample Data

<table>
<thead>
<tr>
<th>rec ID</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>city</th>
<th>country</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>39</td>
<td>John</td>
<td>Smith</td>
<td>m</td>
<td>Chicago</td>
<td>USA</td>
<td>12.03.1964</td>
</tr>
<tr>
<td>40</td>
<td>Mary</td>
<td>Brown</td>
<td>f</td>
<td>London</td>
<td>UK</td>
<td>12.05.1964</td>
</tr>
<tr>
<td>41</td>
<td>Jane</td>
<td>Doe</td>
<td>f</td>
<td>Palo Alto</td>
<td>USA</td>
<td>23.04.1976</td>
</tr>
<tr>
<td>42</td>
<td>John</td>
<td>Doe</td>
<td>m</td>
<td>Palo Alto</td>
<td>USA</td>
<td>17.06.1952</td>
</tr>
<tr>
<td>43</td>
<td>Peter</td>
<td>Schmidt</td>
<td>m</td>
<td>Potsdam</td>
<td>GER</td>
<td>11.11.1975</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
## Dictionary Encoding a Column

- A column is split into a dictionary and an attribute vector.
- Dictionary stores all distinct values with implicit value ID.
- Attribute vector stores value IDs for all entries in the column.
- Position is implicit, not stored explicitly.
- Enables offsetting with fixed-length data types.

### Table: Dictionary and Attribute Vector for "fname"

<table>
<thead>
<tr>
<th>Rec ID</th>
<th>fname</th>
<th>Dictionary for “fname”</th>
<th>Attribute Vector for “fname”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Value ID</td>
<td>Value</td>
</tr>
<tr>
<td>39</td>
<td>John</td>
<td>23</td>
<td>John</td>
</tr>
<tr>
<td>40</td>
<td>Mary</td>
<td>24</td>
<td>Mary</td>
</tr>
<tr>
<td>41</td>
<td>Jane</td>
<td>25</td>
<td>Jane</td>
</tr>
<tr>
<td>42</td>
<td>John</td>
<td>26</td>
<td>Peter</td>
</tr>
<tr>
<td>43</td>
<td>Peter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The dictionary stores all distinct values with implicit value IDs.
- The attribute vector stores value IDs for all entries in the column.
- Position is implicit, not stored explicitly.
- Enables offsetting with fixed-length data types.
Sorted Dictionary

- Dictionary entries are sorted either by their numeric value or lexicographically
  - Dictionary lookup complexity: $O(\log(n))$ instead of $O(n)$

- Dictionary entries can be compressed to reduce the amount of required storage

- Selection criteria with ranges are less expensive (order-preserving dictionary)
# Data Size Examples

<table>
<thead>
<tr>
<th>Column</th>
<th>Cardinality</th>
<th>Bits Needed</th>
<th>Item Size</th>
<th>Plain Size</th>
<th>Size with Dictionary (Dictionary + Column)</th>
<th>Compression Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>First names</td>
<td>5 million</td>
<td>23 bit</td>
<td>50 Byte</td>
<td>373GB</td>
<td>238.4MB + 21.4GB</td>
<td>≈17</td>
</tr>
<tr>
<td>Last names</td>
<td>8 million</td>
<td>23 bit</td>
<td>50 Byte</td>
<td>373GB</td>
<td>381.5MB + 21.4GB</td>
<td>≈17</td>
</tr>
<tr>
<td>Gender</td>
<td>2</td>
<td>1 bit</td>
<td>1 Byte</td>
<td>7GB</td>
<td>2.0b + 953.7MB</td>
<td>≈8</td>
</tr>
<tr>
<td>City</td>
<td>1 million</td>
<td>20 bit</td>
<td>50 Byte</td>
<td>373GB</td>
<td>47.7MB + 18.6GB</td>
<td>≈20</td>
</tr>
<tr>
<td>Country</td>
<td>200</td>
<td>8 bit</td>
<td>47 Byte</td>
<td>350GB</td>
<td>9.2KB + 7.5GB</td>
<td>≈47</td>
</tr>
<tr>
<td>Birthday</td>
<td>40000</td>
<td>16 bit</td>
<td>2 Byte</td>
<td>15GB</td>
<td>78.1KB + 14.9GB</td>
<td>≈1</td>
</tr>
<tr>
<td>Totals</td>
<td></td>
<td></td>
<td>200 Byte</td>
<td>≈ 1.6TB</td>
<td>≈ 92GB</td>
<td>≈17</td>
</tr>
</tbody>
</table>
Chapter 3:

In-Memory Database Operators
Learning Map of our Online Lecture @ openHPI.de

The Future of Enterprise Computing

Introduction → New Requirements for Enterprise Computing

Enterprise Application Characteristics → Changes in Hardware

Blueprint SanssouciDB

Foundations of Database Storage Techniques

Data Layout → Row, Column, Hybrid

Partitioning

In-Memory Database Operators

Dictionary Encoding → Compression

Advanced Database Storage Techniques

DELETE, INSERT, UPDATE, Tuple Reconstruction → Scan, Performance

Parallel Data Processing

Parallel Aggregate Functions → Implications, Dunning, Views, Handling Business Objects, ByPass Solution

Parallel JOIN → Aggregate Functions, SELECT

JOIN

SELECT

Aggregate Functions, Parallels

Materialization Strategies

Workload Management

Parallel Processing

In-Memory Database Operators

Differential Buffer, Insert-Only Time Travel

Merge, On-The-Fly Database Reorganization

Indices, JOIN

Logging → Recovery

Advanced Database Storage Techniques

In-Memory Database Operators

Differential Buffer, Insert-Only Time Travel

Merge, On-The-Fly Database Reorganization

Indices, JOIN

Logging → Recovery
Scan Performance (1)

8 billion humans

- Attributes
  - First Name
  - Last Name
  - Gender
  - Country
  - City
  - Birthday

- 200 byte

- Question: How many men/women?

- Assumed scan speed: 2MB/ms/core
Scan Performance (2)

Row Store – Layout

Table: humans

<table>
<thead>
<tr>
<th></th>
<th>Last Name</th>
<th>Country</th>
<th>Birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 8 x 10^9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Table size = 8 billion tuples x 200 bytes per tuple
  → ≈ 1.6 TB

- Scan through all rows with 2MB/ms/core
  → ≈ 800 seconds
  with 1 core
Scan Performance (3)

Row Store – Full Table Scan

<table>
<thead>
<tr>
<th>Table: humans</th>
<th>Last Name</th>
<th>Country</th>
<th>Birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First Name</td>
<td>Gender</td>
<td>City</td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row $8 \times 10^9$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Table size = 8 billion tuples $\times$ 200 bytes per tuple $\rightarrow \approx 1.6$ TB
- Scan through all rows with 2MB/ms/core $\rightarrow \approx 800$ seconds with 1 core
Scan Performance (4)

Row Store – Stride Access “Gender”

<table>
<thead>
<tr>
<th>Table: humans</th>
<th>Last Name</th>
<th>Country</th>
<th>Birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td>Gender</td>
<td>City</td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row $8 \times 10^9$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 8 billion cache accesses à 64 byte → $\approx 512$ GB
- Read with 2MB/ms/core → $\approx 256$ seconds with 1 core
Scan Performance (5)

Column Store – Layout

<table>
<thead>
<tr>
<th>Table: humans</th>
<th>First Name</th>
<th>Last Name</th>
<th>Gender</th>
<th>Country</th>
<th>City</th>
<th>Birthday</th>
</tr>
</thead>
</table>

- Table size
  - Attribute vectors: \(\approx 91 \text{ GB}\)
  - Dictionaries: \(\approx 700 \text{ MB}\)
  \(\Rightarrow\) Total: \(\approx 92 \text{ GB}\)
- Compression factor: \(\approx 17\)
Scan Performance (6)

Column Store – Full Column Scan on “Gender”

Table: humans

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Gender</th>
<th>Country</th>
<th>City</th>
<th>Birthday</th>
</tr>
</thead>
</table>

- Size of attribute vector “gender” = 8 billion tuples x 1 bit per tuple → $\approx 1$ GB
- Scan through attribute vector with 2MB/ms/core → $\approx 0.5$ seconds with 1 core
Scan Performance (7)

Column Store – Full Column Scan on “Birthday”

Table: humans

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Gender</th>
<th>Country</th>
<th>City</th>
<th>Birthday</th>
</tr>
</thead>
</table>

- Size of attribute vector “birthday” = 8 billion tuples x 2 byte per tuple = \(\approx 16 \text{ GB}\)
- Scan through column with 2MB/ms/core → \(\approx 8 \text{ seconds}\) with 1 core
Scan Performance – Summary

- How many women, how many men?

<table>
<thead>
<tr>
<th></th>
<th>Column Store</th>
<th>Row Store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Full table scan</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>0.5</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td>1,600x slower</td>
<td>512x slower</td>
</tr>
</tbody>
</table>
TUPLE RECONSTRUCTION
## Tuple Reconstruction (I)

Accessing a record in a row store

<table>
<thead>
<tr>
<th>Table: world_population</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
</tr>
<tr>
<td>Row 1</td>
</tr>
<tr>
<td>Row 2</td>
</tr>
<tr>
<td>Row 3</td>
</tr>
<tr>
<td>Row 4</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Row $8 \times 10^9$</td>
</tr>
</tbody>
</table>

- All attributes are stored consecutively
- 200 byte $\Rightarrow$ 4 cache accesses $\Rightarrow$ 64 byte $\Rightarrow$ 256 byte
- Read with 2MB/ms/core $\Rightarrow$ $\approx 0.128 \mu$s with 1 core
Tuple Reconstruction (2)

Virtual record IDs

- All attributes are stored in separate columns
- Implicit record IDs are used to reconstruct rows

Table: world_population

<table>
<thead>
<tr>
<th>First Name</th>
<th>Gender</th>
<th>Country</th>
<th>City</th>
<th>Birthday</th>
</tr>
</thead>
</table>

- Attributes are stored in separate columns.
- Implicit record IDs are used to reconstruct rows.
## Tuple Reconstruction (3)

**Virtual record IDs**

<table>
<thead>
<tr>
<th>Table: world_population</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First Name</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>36</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>39</td>
</tr>
<tr>
<td>40</td>
</tr>
</tbody>
</table>

- 1 cache access for each attribute
- 6 cache accesses à 64 byte
  → 384 byte
- Read with 2MB/ms/core
  → ≈ 0.192 μs with 1 core
Select
SELECT Example

SELECT fname, lname FROM world_population
WHERE country="Italy" and gender="m"

<table>
<thead>
<tr>
<th>fname</th>
<th>lname</th>
<th>country</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gianluigi</td>
<td>Buffon</td>
<td>Italy</td>
<td>m</td>
</tr>
<tr>
<td>Lena</td>
<td>Gercke</td>
<td>Germany</td>
<td>f</td>
</tr>
<tr>
<td>Mario</td>
<td>Balotelli</td>
<td>Italy</td>
<td>m</td>
</tr>
<tr>
<td>Manuel</td>
<td>Neuer</td>
<td>Germany</td>
<td>m</td>
</tr>
<tr>
<td>Lukas</td>
<td>Podolski</td>
<td>Germany</td>
<td>m</td>
</tr>
<tr>
<td>Klaas-Jan</td>
<td>Huntelaar</td>
<td>Netherlands</td>
<td>m</td>
</tr>
</tbody>
</table>
Query Plan

- Multiple plans exist to execute query
  - Query Optimizer decides which is executed
  - Based on cost model, statistics and other parameters

- Alternatives
  - Scan “country” and “gender”, positional AND
  - Scan over “country” and probe into “gender”
  - Indices might be used
  - Decision depends on data and query parameters like e.g. selectivity

SELECT fname, lname FROM world_population
WHERE country=“Italy” and gender=“m”
Query Plan (i)

Positional AND:
- Predicates are evaluated and generate position lists
- Intermediate position lists are logically combined
- Final position list is used for materialization
### Query Execution (i)

#### Value ID  Dictionary for “country”
- 0: Algeria
- 1: France
- 2: Germany
- 3: Italy
- 4: Netherlands
- ...

#### Dictionary for “gender”
- 0: f
- 1: m

#### Return Values
- country = 3 (“Italy”)
- gender = 1 (“m”)

### Table
<table>
<thead>
<tr>
<th>Position</th>
<th>Value ID</th>
<th>Dictionary for “country”</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Algeria</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Netherlands</td>
</tr>
</tbody>
</table>

### Dictionary for “gender”
<table>
<thead>
<tr>
<th>Position</th>
<th>Value ID</th>
<th>Dictionary for “gender”</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>f</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>m</td>
</tr>
</tbody>
</table>

### AND Condition

#### Query Results
- Gianluigi Buffon
- Lena Gercke
- Mario Balotelli
- Manuel Neuer
- Lukas Podolski
- Klaas-Jan Huntelaar
Query Plan (ii)

Based on position list produced by first selection, *gender* column is probed.
Insert

- Insert is the dominant modification operation (Delete/Update can be modeled as Inserts as well)
- Inserting into a compressed in-memory persistence can be expensive
  - Updating sorted sequences (e.g. dictionaries) is a challenge
  - Inserting into columnar storages is generally more expensive than inserting into row storages
Insert Example

<table>
<thead>
<tr>
<th>rowID</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>4</td>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

INSERT INTO world_population
VALUES (Karen, Schulze, f, GER, Rostock, 11-15-2012)
INSERT (1) w/o new Dictionary entry

INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 11-15-2012)

<table>
<thead>
<tr>
<th>AV</th>
<th>D</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Albrecht</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Berg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Meyer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Schulze</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>fname</th>
<th>Iname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>4</td>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Attribute Vector (AV)
Dictionary (D)
**INSERT (1) w/o new Dictionary entry**

**INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, , 11-15-2012)**

1. **Look-up on D → entry found**

<table>
<thead>
<tr>
<th>AV</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

**AV**

- **0**
  - Albrecht
  - Berg
- **1**
  - Meyer
- **2**
  - Schulze

**Dictionary (D)**

<table>
<thead>
<tr>
<th>fname</th>
<th>Iname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Attribute Vector (AV)**

**Dictionary (D)**
**INSERT (I) w/o new Dictionary entry**

**INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, , 11-15-2012)**

1. Look-up on D → entry found
2. Append ValueID to AV
INSERT (2) with new Dictionary Entry I/II

INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, , 11-15-2012)

<table>
<thead>
<tr>
<th>AV</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>4</td>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Attribute Vector (AV)
Dictionary (D)
INSERT (2) with new Dictionary Entry I/II

INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 11-15-2012)

1. Look-up on D \(\rightarrow\) no entry found
INSERT (2) with new Dictionary Entry I/II

1. Look-up on D → no entry found
2. Append new value to D (no re-sorting necessary)
INSERT (2) with new Dictionary Entry I/II

1. Look-up on D → no entry found
2. Append new value to D (no re-sorting necessary)
3. Append ValueID to AV
INSERT (2) with new Dictionary Entry I/II

INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, , 11-15-2012)
1. Look-up on D → no entry found
1. **Look-up on D** → **no entry found**
2. **Insert new value to D**
1. Look-up on D \(\rightarrow\) no entry found
2. Insert new value to D

**INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 11-15-2012)**

---

**Attribute Vector (AV)**

<table>
<thead>
<tr>
<th>AV</th>
<th>D (old)</th>
<th>D (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Dictionary (D)**

<table>
<thead>
<tr>
<th>fname</th>
<th>Iname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anton</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>Hanna</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>Martin</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>Michael</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td></td>
<td>Schulze</td>
<td></td>
<td></td>
<td>Rostock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
INSERT (2) with new Dictionary Entry II/II

INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 11-15-2012)

1. Look-up on D → no entry found
2. Insert new value to D
3. Change ValuelIDs in AV

Attribute Vector (AV)
Dictionary (D)
1. Look-up on D → no entry found
2. Insert new value to D
3. Change ValueIDs in AV
4. Append new ValueID to AV

**INSERT (2) with new Dictionary Entry II/II**

**INSERT INTO world_population VALUES** (Karen, Schulze, f, GER, Rostock, 11-15-2012)

<table>
<thead>
<tr>
<th>AV</th>
<th>DICTIONARY (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 Anton</td>
</tr>
<tr>
<td>1</td>
<td>4 Hanna</td>
</tr>
<tr>
<td>2</td>
<td>1 Karen</td>
</tr>
<tr>
<td>3</td>
<td>0 Martin</td>
</tr>
<tr>
<td>4</td>
<td>5 Michael</td>
</tr>
<tr>
<td>5</td>
<td>2 Sophie</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>AV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 Anton</td>
</tr>
<tr>
<td>1</td>
<td>4 Hanna</td>
</tr>
<tr>
<td>2</td>
<td>1 Karen</td>
</tr>
<tr>
<td>3</td>
<td>0 Martin</td>
</tr>
<tr>
<td>4</td>
<td>5 Michael</td>
</tr>
<tr>
<td>5</td>
<td>2 Sophie</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACOL</th>
<th>CNAME</th>
<th>Gender</th>
<th>Country</th>
<th>City</th>
<th>Birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>4</td>
<td>Sophie</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td>5</td>
<td>Karen</td>
<td>f</td>
<td>Rostock</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Attribute Vector (AV)**

**Dictionary (D)**
RESULT

<table>
<thead>
<tr>
<th>rowID</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>4</td>
<td>Ulrike</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td>5</td>
<td>Karen</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Rostock</td>
<td>11-15-2012</td>
</tr>
</tbody>
</table>

INSERT INTO world_population
VALUES (Karen, Schulze, f, GER, Rostock, 11-15-2012)
Chapter 4:

Advanced Database Storage Techniques
Learning Map of our
Online Lecture @ openHPI.de

The Future of Enterprise Computing

Introduction
New Requirements for Enterprise Computing

Enterprise Application Characteristics
Changes in Hardware

Blueprint SanssouciDB

Partitioning

Data Layout
Row, Column, Hybrid

Foundations of Database Storage Techniques

In-Memory Database Operators

Dictionary Encoding ➔ Compression

Advanced Database Storage Techniques

DELETE ➔ INSERT ➔ UPDATE

Tuple Reconstruction
Scan Performance

Parallel Data Processing
Workload Management

Parallel JOIN ➔ Aggregate Functions

Parallel Aggregate Functions

Database Reorganization

On-The-Fly

Materialization Strategies

Joining Strategies

Implications
Dunning
Views
Handing Business Objects
ByPass Solution

Logging ➔ Recovery

Differential Buffer
Merge

Insert-Only Time Travel

Indices

Aggregate Functions

Application Development

Parallel SELECT

Workload Management

ByPass Solution
Differential Buffer
Motivation

- Inserting new tuples directly into a compressed structure can be expensive
  - Especially when using sorted structures
  - New values can require reorganizing the dictionary
  - Number of bits required to encode all dictionary values can change, attribute vector has to be reorganized
Differential Buffer

- New values are written to a dedicated differential buffer (Delta)
- Cache Sensitive B+ Tree (CSB+) used for fast search on Delta

```
world_population

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

Main Store

- 8 Billion entries

```
fname

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Dictionary

- Anton
- Hanna
- Michael
- Sophie

Differential Buffer/

- Delta

```
fname

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Dictionary

- Angela
- Klaus
- Andre

- up to 50,000 entries

CSB+
Differential Buffer

- Inserts of new values are fast, because dictionary and attribute vector do not need to be resorted
- Range selects on differential buffer are expensive
  - Unsorted dictionary allows no direct comparison of valueIds
  - Scans with range selection need to lookup values in dictionary for comparisons
- Differential Buffer requires more memory:
  - No attribute vector compression
  - Additional CSB+ Tree for dictionary
### Tuple Lifetime

**Michael moves from Berlin to Potsdam**

Main Table: `world_population`

<table>
<thead>
<tr>
<th>recId</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>4</td>
<td>Ulrike</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td>5</td>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Rostock</td>
<td>06-20-2012</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8 * 10^9</td>
<td>Zacharias</td>
<td>Perdopolus</td>
<td>m</td>
<td>GRE</td>
<td>Athen</td>
<td>03-12-1979</td>
</tr>
</tbody>
</table>

UPDATE `world_population`
SET `city`='Potsdam'
WHERE `fname`="Michael" AND `lname`="Berg"
Tuple Lifetime

Michael moves from Berlin to Potsdam

Main Table: world_population

<table>
<thead>
<tr>
<th>reклд</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>4</td>
<td>Ulrike</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td>5</td>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Rostock</td>
<td>06-20-2012</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8 * 10^9</td>
<td>Zacharias</td>
<td>Perdopolus</td>
<td>m</td>
<td>GRE</td>
<td>Athen</td>
<td>03-12-1979</td>
</tr>
</tbody>
</table>

UPDATE ‘world_population’
SET  city='Potsdam'
WHERE  fname=“Michael” AND lname=“Berg”
### Tuple Lifetime

#### Michael moves from Berlin to Potsdam

<table>
<thead>
<tr>
<th>recId</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
</tr>
<tr>
<td>4</td>
<td>Ulrike</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
</tr>
<tr>
<td>5</td>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Rostock</td>
<td>06-20-2012</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8 * 10^9</td>
<td>Zacharias</td>
<td>Perdopolus</td>
<td>m</td>
<td>GRE</td>
<td>Athen</td>
<td>03-12-1979</td>
</tr>
</tbody>
</table>

**UPDATE** ‘world_population’

**SET**  *city*='Potsdam'

**WHERE**  *fname*='Michael' AND *lname*='Berg'
Tuple Lifetime

- Tuples are now available in Main Store and Differential Buffer
- Tuples of a table are marked by a validity vector to reduce the required amount of reorganization steps
  - Like an attribute vector for validity
  - 1 bit required per database tuple
- Invalidated tuples stay in the database table, until the next reorganization takes place
- Query results
  - Main and delta have to be queried
  - Results are filtered using the validity vector
## Tuple Lifetime

### Michael moves from Berlin to Potsdam

#### Main Table: world_population

<table>
<thead>
<tr>
<th>recld</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Ulrike</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Rostock</td>
<td>06-20-2012</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8 * 10^9</td>
<td>Zacharias</td>
<td>Perdopolus</td>
<td>m</td>
<td>GRE</td>
<td>Athen</td>
<td>03-12-1979</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Potsdam</td>
<td>03-05-1970</td>
<td>1</td>
</tr>
</tbody>
</table>

**UPDATE** ‘world_population’
**SET** `city='Potsdam'`
**WHERE** `fname=“Michael” AND lname=“Berg”`
Tuple Lifetime

Michael moves from Berlin to Potsdam

Main Table: world_population

<table>
<thead>
<tr>
<th>reclid</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>country</th>
<th>city</th>
<th>birthday</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Martin</td>
<td>Albrecht</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>08-05-1955</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Berlin</td>
<td>03-05-1970</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Hanna</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Hamburg</td>
<td>04-04-1968</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Anton</td>
<td>Meyer</td>
<td>m</td>
<td>AUT</td>
<td>Innsbruck</td>
<td>10-20-1992</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Ulrike</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Potsdam</td>
<td>09-03-1977</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Sophie</td>
<td>Schulze</td>
<td>f</td>
<td>GER</td>
<td>Rostock</td>
<td>06-20-2012</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8 * 10^9</td>
<td>Zacharias</td>
<td>Perdopolus</td>
<td>m</td>
<td>GRE</td>
<td>Athen</td>
<td>03-12-1979</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>Michael</td>
<td>Berg</td>
<td>m</td>
<td>GER</td>
<td>Potsdam</td>
<td>03-05-1970</td>
<td>1</td>
</tr>
</tbody>
</table>

UPDATE ‘world_population’
SET city='Potsdam'
WHERE fname=“Michael” AND lname=“Berg”
Merge
Handling Write Operations

- All Write operations (INSERT, UPDATE) are stored within a differential buffer (delta) first
- Read-operations on differential buffer are more expensive than on main store
- Differential buffer is merged periodically with the main store
  - To avoid performance degradation based on large delta
  - Merge is performed asynchronously
Merge Overview I/II

- The merge process is triggered for single tables

- Is triggered by:
  - Amount of tuples in buffer
  - Cost model to
    - Schedule
    - Take query cost into account
  - Manually
Merge Overview II/II

- Working on data copies allows asynchronous merge
- Very limited interruption due to short lock
- At least twice the memory of the table needed!

Before

<table>
<thead>
<tr>
<th>Table</th>
<th>Main Store</th>
<th>Differential Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Modifying Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operations</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

During the Merge Process

<table>
<thead>
<tr>
<th>Table</th>
<th>Main Store</th>
<th>Differential Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Modifying Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operations</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Table</th>
<th>Main Store (new)</th>
<th>Differential Buffer (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Modifying Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operations</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prepare  
Attribute Merge  
Commit
**Attribute Merge**

**Step 1: Dictionary Merge**
- Merge main and delta dictionary
  - Optionally remove unused values
  - Merge of two sorted arrays
- Create mapping if valueIDs changed

**Step 2: Update Attribute Vector**
- Create new merged main partition
- Update valueIDs reflecting changed dictionary
**Example**

**Main Store**

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
<th>recID</th>
<th>fname</th>
<th>city</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Albert</td>
<td>Berlin</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>London</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Nadja</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Delta**

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
<th>recID</th>
<th>fname</th>
<th>city</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>recID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Dictionaries**

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
<th>recID</th>
<th>fname</th>
<th>city</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>recID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Michael moves from London to Berlin

<table>
<thead>
<tr>
<th>Dictionaries</th>
<th>Attribute Vectors</th>
<th>Validity Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueID</td>
<td>recID</td>
<td>fname</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Albert</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Michael</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Nadja</td>
</tr>
</tbody>
</table>

Delta

<table>
<thead>
<tr>
<th>Dictionaries</th>
<th>Attribute Vectors</th>
<th>Validity Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueID</td>
<td>recID</td>
<td>fname</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Michael</td>
</tr>
</tbody>
</table>
Example

Nadja moves from Berlin to Potsdam

### Main Store

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
<th>recID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Albert</td>
<td>Berlin</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>London</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Nadja</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Delta

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
<th>recID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Michael</td>
<td>Berlin</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Nadja</td>
<td>Potsdam</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Validity Vector

<table>
<thead>
<tr>
<th>recID</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Michael moves from Berlin to Potsdam

**Main Store**

<table>
<thead>
<tr>
<th>valueID</th>
<th>Dictionaries</th>
<th>Attribute Vectors</th>
<th>Validity Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fname</td>
<td>city</td>
<td>recID</td>
</tr>
<tr>
<td>0</td>
<td>Albert</td>
<td>Berlin</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>London</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Nadja</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

**Delta**

<table>
<thead>
<tr>
<th>valueID</th>
<th>Dictionaries</th>
<th>Attribute Vectors</th>
<th>Validity Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fname</td>
<td>city</td>
<td>recID</td>
</tr>
<tr>
<td>0</td>
<td>Michael</td>
<td>Berlin</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Nadja</td>
<td>Potsdam</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
# First Step: Validity Detection

## Main Store

<table>
<thead>
<tr>
<th>valueID</th>
<th>Dictionaries</th>
<th>Attribute Vectors</th>
<th>Validity Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>recID</td>
<td>fname</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dictionaries</th>
<th>Attribute Vectors</th>
<th>Validity Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueID</td>
<td>recID</td>
<td>fname</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

## Delta

<table>
<thead>
<tr>
<th>valueID</th>
<th>Dictionaries</th>
<th>Attribute Vectors</th>
<th>Validity Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>recID</td>
<td>fname</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Will be deleted / moved into history partition
First Step: Dictionary Merge

**Main Store**

<table>
<thead>
<tr>
<th>ValueID</th>
<th>Fname</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Albert</td>
<td>Berlin</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>London</td>
</tr>
<tr>
<td>2</td>
<td>Nadja</td>
<td></td>
</tr>
</tbody>
</table>

**Delta**

<table>
<thead>
<tr>
<th>ValueID</th>
<th>Fname</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Michael</td>
<td>Berlin</td>
</tr>
<tr>
<td>1</td>
<td>Nadja</td>
<td>Potsdam</td>
</tr>
</tbody>
</table>

**Mappings**

<table>
<thead>
<tr>
<th>ValueID (old)</th>
<th>ValueID (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2</td>
</tr>
<tr>
<td>ValueID (Delta)</td>
<td>ValueID (Main)</td>
</tr>
<tr>
<td>0</td>
<td>1 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ValueID (old)</th>
<th>ValueID (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>ValueID (Delta)</td>
<td>ValueID (Main)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Second Step: New Main Column

Main Store

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Albert</td>
<td>Berlin</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>London</td>
</tr>
<tr>
<td>2</td>
<td>Nadja</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>recID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Validity Vector

<table>
<thead>
<tr>
<th>recID</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Delta

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Michael</td>
<td>Berlin</td>
</tr>
<tr>
<td>1</td>
<td>Nadja</td>
<td>Potsdam</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>recID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Validity Vector

<table>
<thead>
<tr>
<th>recID</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Mappings

<table>
<thead>
<tr>
<th>fname: old to new Main</th>
<th>fname: Delta to Main</th>
<th>city: old to new Main</th>
<th>city: Delta to Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueID (old)</td>
<td>valueID (Delta)</td>
<td>valueID (old)</td>
<td>valueID (Delta)</td>
</tr>
<tr>
<td>valueID (new)</td>
<td>valueID (Main)</td>
<td>valueID (new)</td>
<td>valueID (Main)</td>
</tr>
<tr>
<td>0  1  2</td>
<td>0  1</td>
<td>0  1</td>
<td>0  1</td>
</tr>
<tr>
<td>0  1  2</td>
<td>1  2</td>
<td>0 NA</td>
<td>0  1</td>
</tr>
</tbody>
</table>
Second Step: New Main Column

Main Store

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
<th>recID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Albert</td>
<td>Berlin</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Michael</td>
<td>Potsdam</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Nadja</td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>valueID</th>
<th>recID</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Delta

<table>
<thead>
<tr>
<th>valueID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>recID</th>
<th>fname</th>
<th>city</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>recID</th>
<th>valid</th>
</tr>
</thead>
</table>
Chapter 5:

Implications on Application Development
Learning Map of our Online Lecture @ openHPI.de

The Future of Enterprise Computing

Foundations of Database Storage Techniques

In-Memory Database Operators

Advanced Database Storage Techniques
How does it all come together?

1. Mixed Workload combining OLTP and analytic-style queries
   - **Column-Stores** are best suited for analytic-style queries
   - **In-memory** databases enable fast tuple re-construction
   - In-memory column store allows aggregation on-the-fly

2. Sparse enterprise data
   - Lightweight **compression** schemes are optimal
   - Increases query execution
   - Improves feasibility of in-memory databases

3. Mostly read workload
   - Read-optimized stores provide best throughput
     - i.e. compressed in-memory column-store
   - Write-optimized store as delta partition to handle data changes is sufficient
An In-Memory Database for Enterprise Applications

- **In-Memory Database (IMDB)**
  - Data resides **permanently** in main memory
  - Main Memory is the **primary “persistence”**
  - Still: logging and recovery from/to **flash**
  - Main memory access is the new **bottleneck**
  - Cache-conscious algorithms/data structures are **crucial** (locality is king)
## Simplified Application Development

<table>
<thead>
<tr>
<th></th>
<th>Traditional</th>
<th>Column-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application cache</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Database cache</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Prebuilt aggregates</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Raw data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Fewer caches necessary
- No redundant data (OLAP/OLTP, LiveCache)
- No maintenance of materialized views or aggregates
- Minimal index maintenance
Examples for Implications on Enterprise Applications
SAP ERP Financials on In-Memory Technology

In-memory column database for an ERP system

- Combined workload (parallel OLTP/OLAP queries)
- Leverage in-memory capabilities to
  - Reduce amount of data
  - Aggregate on-the-fly
  - Run analytic-style queries (to replace materialized views)
  - Execute stored procedures

- Use Case: **SAP ERP Financials** solution
  - Post and change documents
  - Display open items
  - Run dunning job
  - Analytical queries, such as balance sheet
Current Financials Solutions

Base Tables

- Accounting Document Header
- Accounting Document Items

Change History

- General Ledger
  - Accounts Payable
  - Accounts Receivable
  - Material Ledger

- Sales Ledger
  - Tax Ledger
  - Fixed Asset
  - Cash Ledger

Materialized Aggregates

Reporting Cubes

Indices

- General Ledger Items
  - Accounts Payable Items
  - Accounts Receivable Items
- Sales Ledger Items
  - Tax Ledger Items
  - Fixed Asset Items
- Material Ledger Items
  - Dunning
  - Cash Ledger Items
- Payments

Materialized Views
The Target Financials Solution

Only base tables, algorithms, and some indices

```
Base Tables

Accounting Document Header

Accounting Document Items

Indices
```
Feasibility of Financials on In-Memory Technology in 2009

- Modifications on SAP Financials
  - **Removed** secondary indices, sum tables and pre-calculated and materialized tables
  - **Reduce** code complexity and simplify locks
  - Insert Only to enable **history** (change document replacement)
  - Added **stored procedures** with business functionality

- European division of a retailer
  - ERP 2005 ECC 6.0 EhP3
  - 5.5 TB system database size
  - Financials:
    - 23 million headers / 1.5 GB in main memory
    - 252 million items / 50 GB in main memory
    (including inverted indices for join attributes and insert only extension)
In-Memory Financials on SAP ERP

- **Accounting documents**
  - BKPF
  - BSEG
  - Secondary indices:
    - BSAD
    - BSAK
    - BSAS
    - BSID
    - BSIK
    - BSIS

- **Dunning data**
  - Change documents:
    - CDHDR
    - CDPOS
  - Sum tables:
    - MHNK
    - MHND
    - GLT0
    - LFC1
    - KNC1

- In-Memory Financials on SAP ERP
In-Memory Financials on SAP ERP

accounting documents

BKPF

BSEG
Reduction by a Factor 10

<table>
<thead>
<tr>
<th></th>
<th>Classic Row-Store (w/o compr.)</th>
<th>IMDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKPF</td>
<td>8.7 GB</td>
<td>1.5 GB</td>
</tr>
<tr>
<td>BSEG</td>
<td>255 GB</td>
<td>50 GB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>263.7 GB</strong></td>
<td><strong>51.5 GB</strong></td>
</tr>
<tr>
<td>Secondary Indices</td>
<td>255 GB</td>
<td>-</td>
</tr>
<tr>
<td>Sum Tables</td>
<td>0.55 GB</td>
<td>-</td>
</tr>
<tr>
<td>Complete</td>
<td><strong>519.25 GB</strong></td>
<td><strong>51.5 GB</strong></td>
</tr>
</tbody>
</table>
Booking an accounting document

- Insert into BKPF and BSEG only
- Lack of updates reduces locks

![Time Saved Chart]

- BSID
- KNC1
- GLT0
- Enqueue other

Time Saved

- 15 ms
- 10 ms
- 5 ms
- 0 ms
Wrap Up (I)

- The future of enterprise computing
  - Big data challenges, changes in Hardware
  - OLTP and OLAP in one single system
- Foundations of database storage techniques
  - Data layout optimized for memory hierarchies
  - Light-weight compression techniques
- In-memory database operators
  - Operators on dictionary compressed data
  - Query execution: Scan, Insert, Tuple Reconstruction
Wrap Up (II)

- Advanced database storage techniques
  - Differential buffer accumulates changes
  - Merge combines changes periodically with main storage

- Implications on Application Development
  - Move data intensive operations closer to the data
  - New analytical applications on transactional data possible
  - Less data redundancy, more on the fly calculation
  - Reduced code complexity
References

- Plattner: A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases
  - [http://epic.hpi.uni-potsdam.de/Home/InMemoryBooks](http://epic.hpi.uni-potsdam.de/Home/InMemoryBooks)

- Publications of our Research Group:
  - Papers about the inner-workings of in-memory databases
  - [http://epic.hpi.uni-potsdam.de/Home/Publications](http://epic.hpi.uni-potsdam.de/Home/Publications)
Thank You!

Technical Deep-Dive in a Column-Oriented In-Memory Database

Martin Faust
Martin.Faust@hpi.uni-potsdam.de
Research Group of Prof. Hasso Plattner
Hasso Plattner Institute for Software Engineering
University of Potsdam
Dunning Run

- Dunning run determines all open and due invoices
- Customer defined queries on 250M records
- Current system: 20 min
- New logic: **1.5 sec**
  - In-memory column store
  - Parallelized stored procedures
  - Simplified Financials
Bring Application Logic Closer to the Storage Layer

- Select accounts to be dunned, for each:
  - Select open account items from BSID, for each:
    - Calculate due date
    - Select dunning procedure, level and area
    - Create MHNK entries
- Create and write dunning item tables
Bring Application Logic Closer to the Storage Layer

- Select accounts to be dunned, for each:
  - Select open account items from BSID, for each:
    - Calculate due date
    - Select dunning procedure, level and area
    - Create MHNK entries
- Create and write dunning item tables

1 SELECT
10000 SELECT’s
10000 SELECT’s
31000 Entries
Bring Application Logic Closer to the Storage Layer

- Select accounts to be dunned, for each:
  - Select open account items from BSID, for each:
    - Calculate due date
    - Select dunning procedure, level and area
    - Create MHNK entries

- Create and write dunning item tables

One single stored procedure executed within IMDB

10000 SELECT’s
31000 Entries
Bring Application Logic Closer to the Storage Layer

- Select accounts to be dunned, for each:
  - Select open account items from BSID, for each:
    - Calculate due date
    - Select dunning procedure, level and area
    - Create MHNK entries
- Create and write dunning item tables

One single stored procedure executed within IMDB

Calculated on-the-fly
Dunning Application
Dunning Application

Top Dunning Items

<table>
<thead>
<tr>
<th>Due Date</th>
<th>Days Overdue</th>
<th>Amount</th>
<th>Lost Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-09-19</td>
<td>240</td>
<td>$3,465.60</td>
<td>$113.94</td>
</tr>
<tr>
<td>2009-09-19</td>
<td>240</td>
<td>$3,184.72</td>
<td>$104.70</td>
</tr>
<tr>
<td>2009-09-19</td>
<td>240</td>
<td>$1,478.40</td>
<td>$48.60</td>
</tr>
<tr>
<td>2009-09-20</td>
<td>239</td>
<td>$3,806.06</td>
<td>$124.61</td>
</tr>
<tr>
<td>2009-09-20</td>
<td>239</td>
<td>$3,592.68</td>
<td>$117.62</td>
</tr>
<tr>
<td>2009-09-20</td>
<td>239</td>
<td>$1,478.40</td>
<td>$48.40</td>
</tr>
<tr>
<td>2009-09-21</td>
<td>238</td>
<td>$12,026.35</td>
<td>$392.09</td>
</tr>
</tbody>
</table>
Available-to-Promise Check

- Can I get enough quantities of a requested product on a desired delivery date?
- Goal: Analyze and validate the potential of in-memory and highly parallel data processing for Available-to-Promise (ATP)
- Challenges
  - Dynamic aggregation
  - Instant rescheduling in minutes vs. nightly batch runs
  - Real-time and historical analytics
- Outcome
  - Real-time ATP checks without materialized views
  - Ad-hoc rescheduling
  - No materialized aggregates
In-Memory Available-to-Promise
Demand Planning

- Flexible analysis of demand planning data
- Zooming to choose granularity
- Filter by certain products or customers
- Browse through time spans
- Combination of location-based geo data with planning data in an in-memory database
- External factors such as the temperature, or the level of cloudiness can be overlaid to incorporate them in planning decisions
GORFID

- HANA for Streaming Data Processing
- Use Case: In-Memory RFID Data Management
- Evaluation of SAP OER
- Prototypical implementation of:
  - RFID Read Event Repository on HANA
  - Discovery Service on HANA (*10 Billion data records* with ca. 3 seconds response time)
  - Frontends for iPhone, iPad2

- Key Findings:
  - HANA is suited for streaming data (using bulk inserts)
  - Analytics on streaming data is now possible
GORFID: “Near Real-Time” as a Concept

- **Read Event Repositories**: up to 8,000 read event notifications per second
- **Verification Services**: up to 2,000 requests per second
- **Discovery Service**

bulk load every 2-3 seconds: > 50,000 inserts/s

**SAP HANA**