CD216: Technical Deep-Dive in a Column-Oriented In-Memory Database

Dr. Jan Schaffner
Research Group of Prof. Hasso Plattner

Hasso Plattner Institute for Software Engineering
University of Potsdam

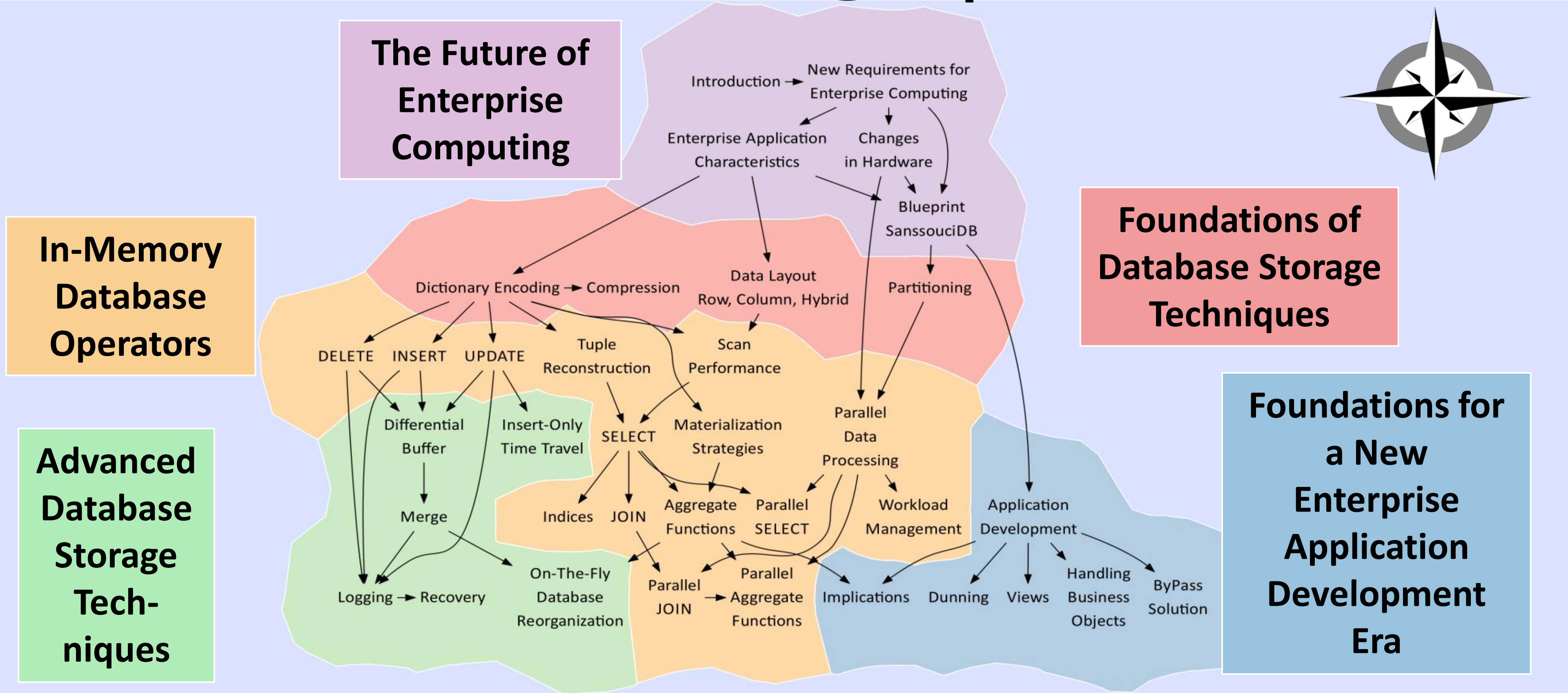
Goals

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

Chapters

- The status quo in 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



Chapter 1:

The Status Quo in Enterprise Computing

OLTP vs. OLAP

On**l**ine **T**ransaction **P**rocessing

On**l**ine **A**nalytical **P**rocessing

- 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)

OLTP vs. OLAP

On**l**ine **T**ransaction **P**rocessing

On**l**ine **A**nalytical **P**rocessing

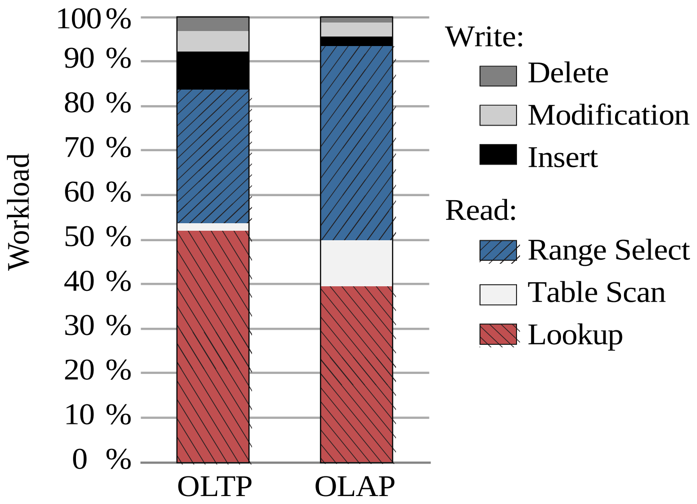
But: Today's data management systems are optimized **either** for daily **transactional** or **analytical** workloads storing their data along rows or columns

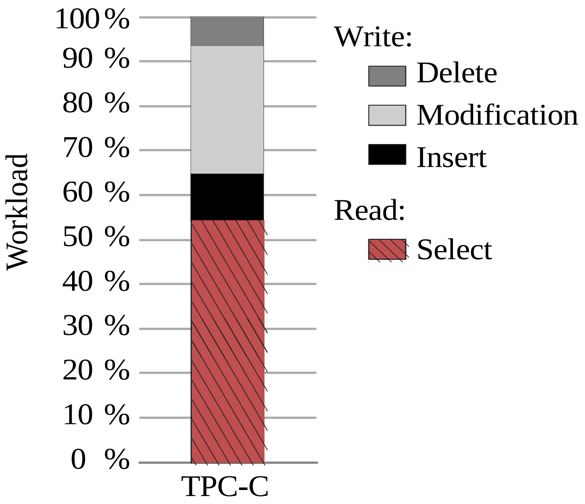
Drawbacks of the Separation

- 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

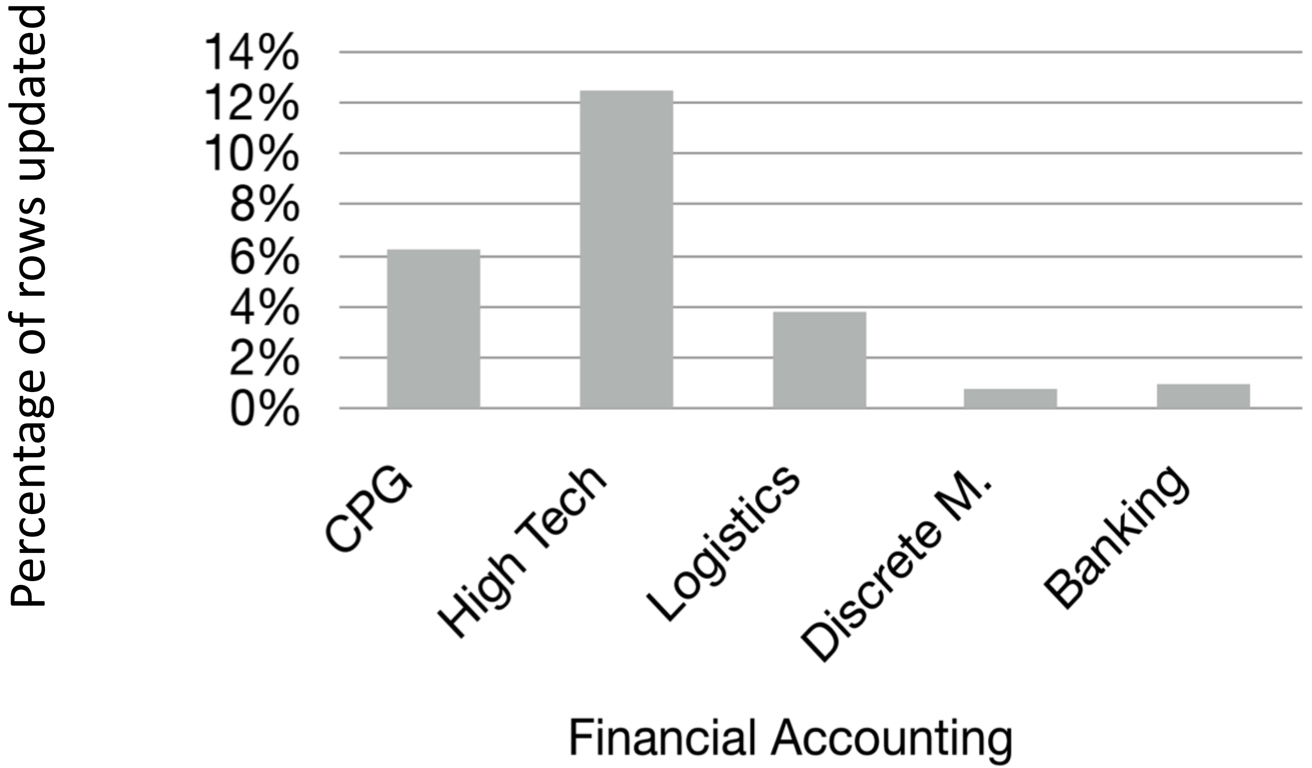
Enterprise Workloads are Read Dominated

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





Few Updates in OLTP



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.

Vision

Additionally,

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

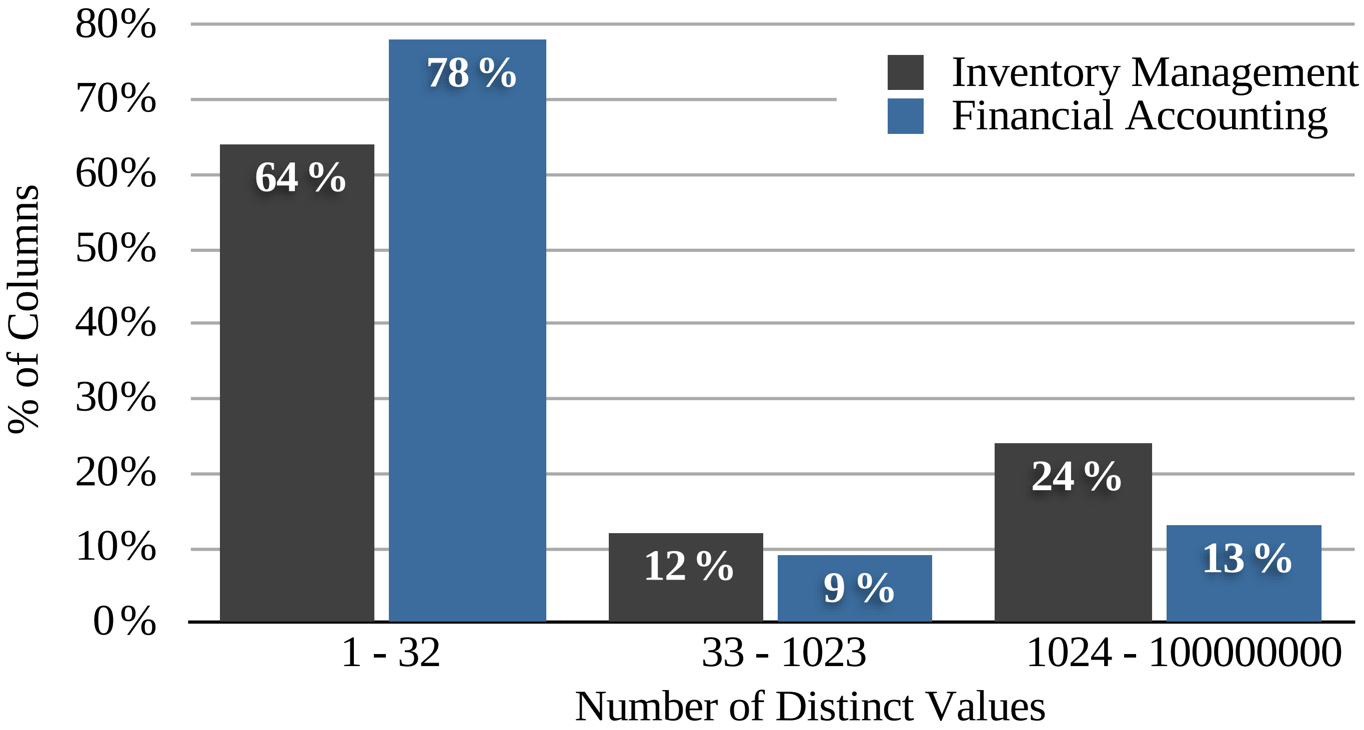
become (almost) obsolete.

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.**

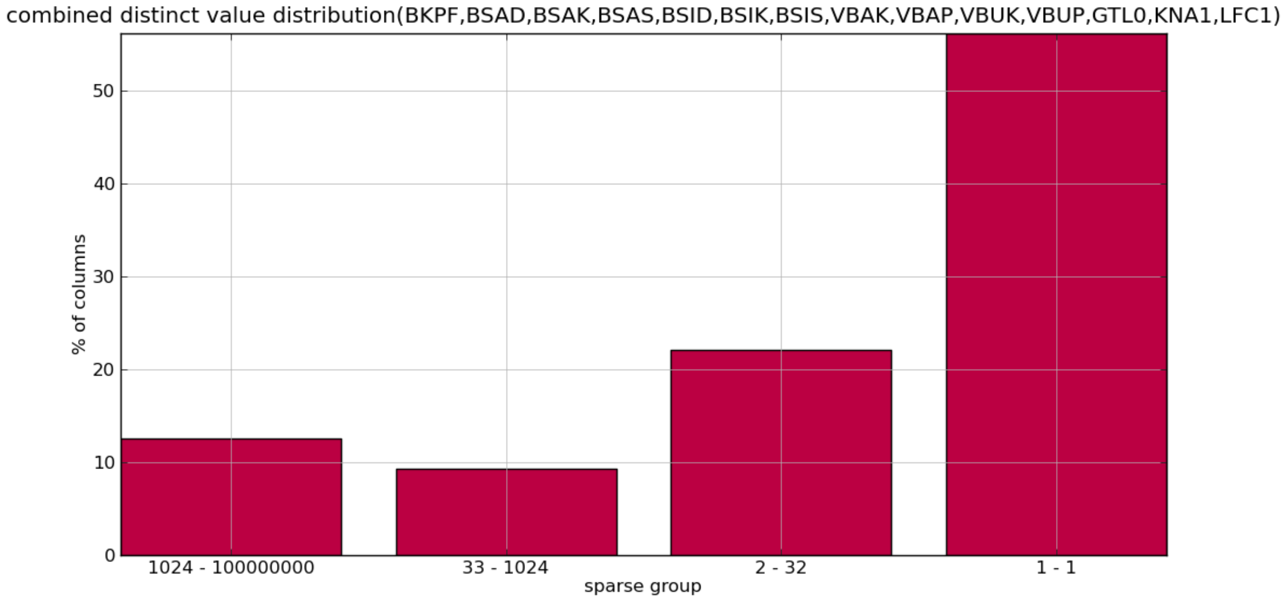
Sparse Tables



Sparse Tables

55% unused columns per company in average

40% unused columns across all analyzed companies

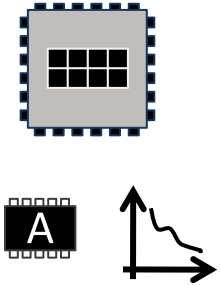


Changes in Hardware

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: 4TB /blade
- One enterprise class server ~$50.000
- Parallel scaling across blades



- 64-bit address space
- 4TB in current servers
- Cost-performance ratio rapidly declining
- Memory hierarchies

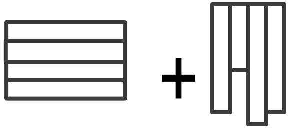
- Main Memory becomes **cheaper and larger**

In the Meantime

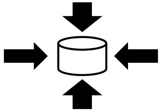
Research has come up with…

… 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



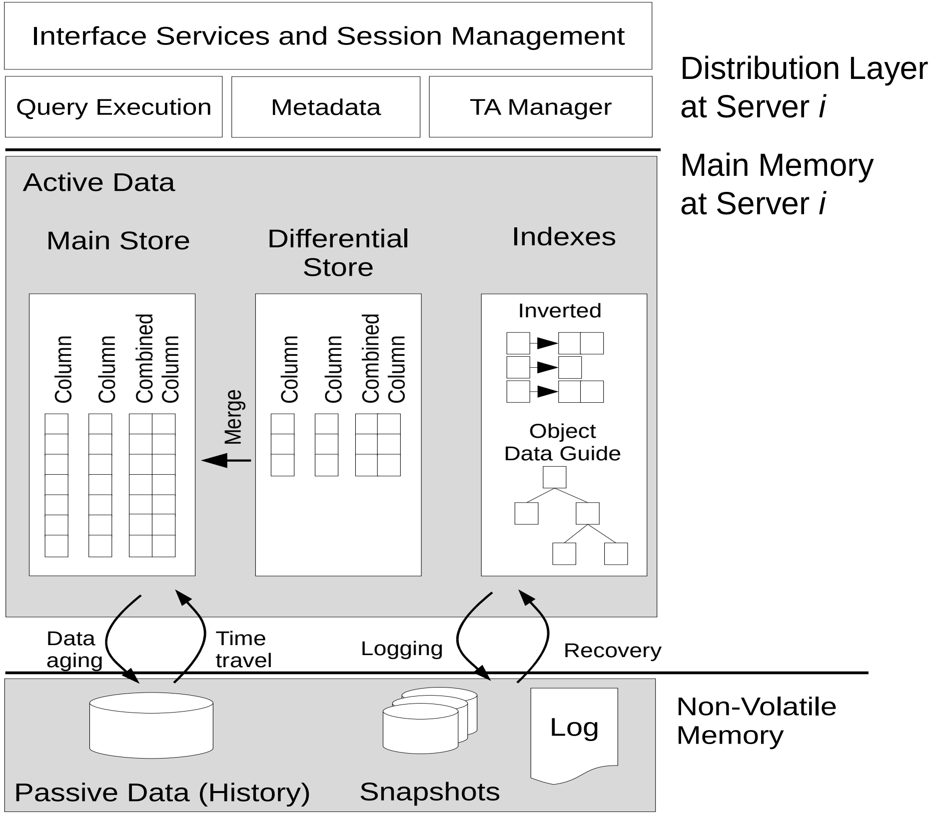
- Lightweight Compression
  - Reducing data amount
  - Increasing processing speed through late materialization



- And more, e.g., parallel scan/join/aggregation

A Blueprint of SanssouciDB

SanssouciDB: An In-Memory Database for Enterprise Applications



SanssouciDB: An In-Memory Database for Enterprise Applications

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

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)

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Chapter 2:

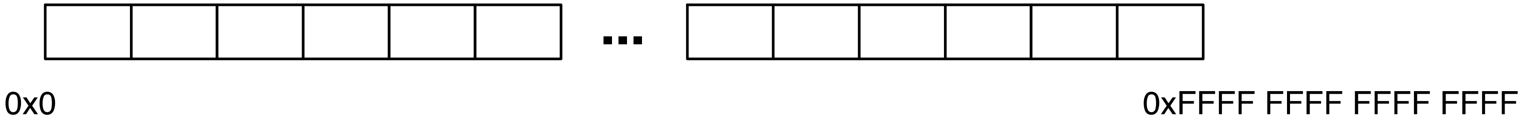
Foundations of Database Storage Techniques

Data Layout in Main Memory

Memory Basics (I)

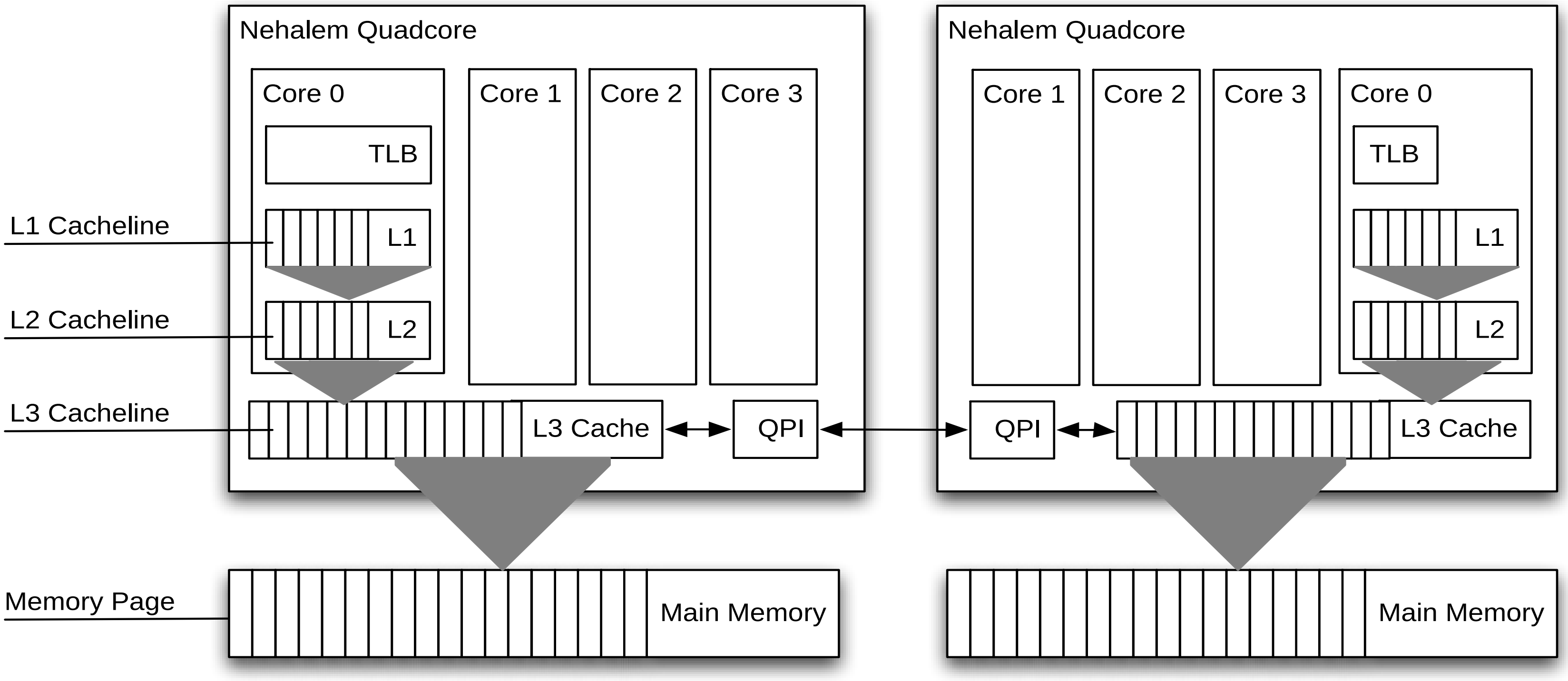
- Memory in today's computers has a linear address layout: addresses start at 0x0 and go to 0xFFFFFFFFFFFFFFFF for 64bit
- Each process has its own virtual address space
- Virtual memory allocated by the program can distribute over multiple physical memory locations
- Address translation is done in hardware by the CPU

Memory Basics (2)



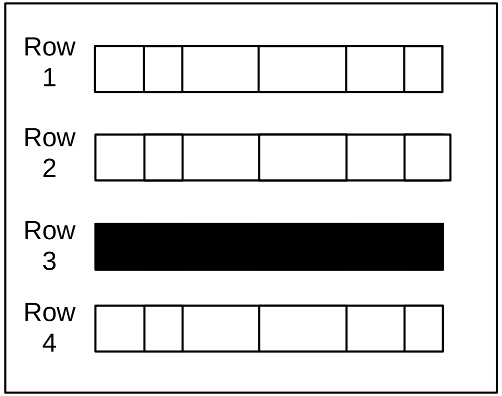
- Memory layout is **only linear**
- Every higher-dimensional access (like two-dimensional database tables) is mapped to this linear band

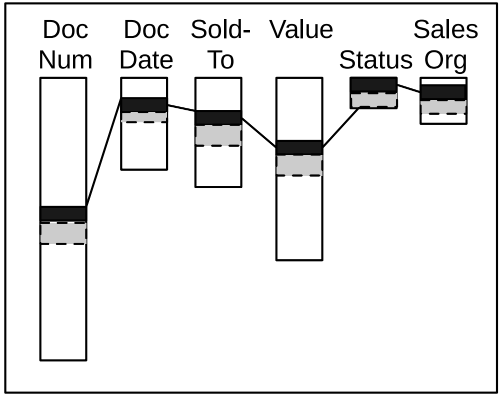
Memory Hierarchy



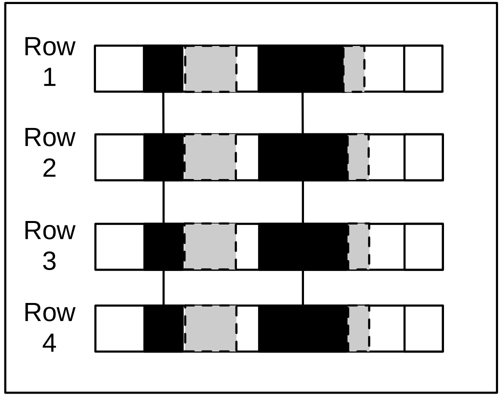
Rows vs. Columns

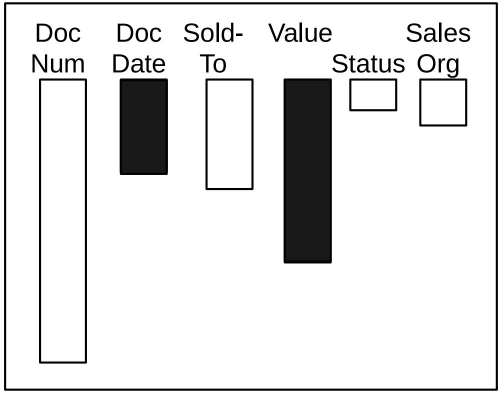
SELECT *
FROM Sales Orders
WHERE Document Number = '95779216'
(OLTP-style query)



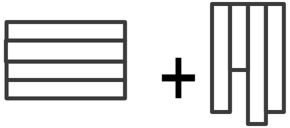


SELECT SUM(Value)
FROM Sales Orders
WHERE Document Date > 2011-08-28
(OLAP-style query)

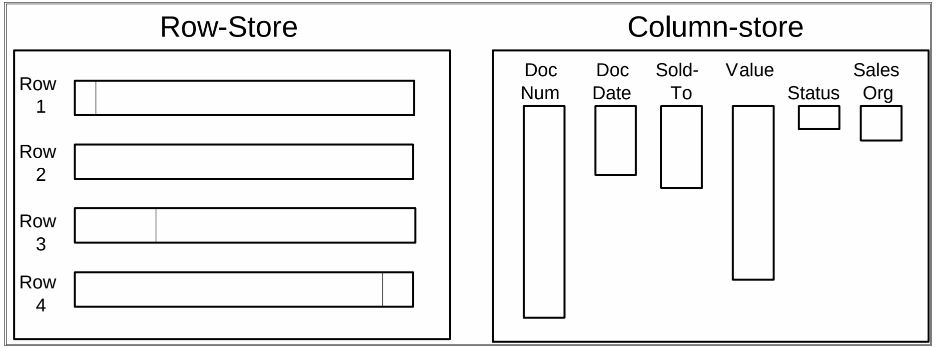




Physical Data Representation

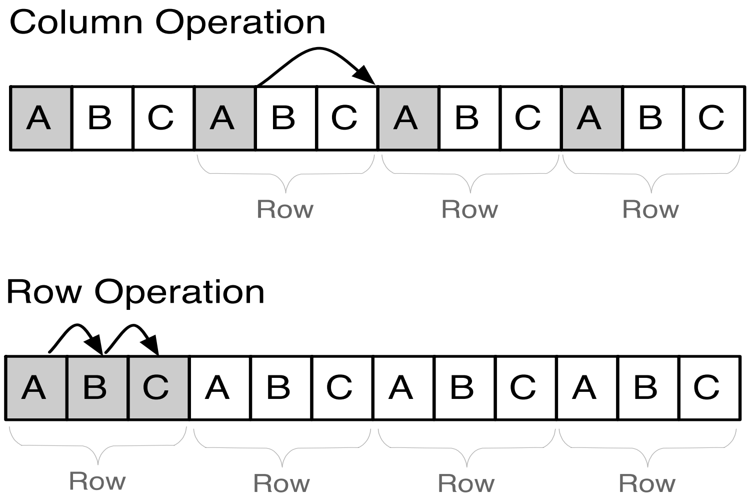


- **Row** store:
  - Rows are stored consecutively
  - Optimal for row-wise access (e.g. SELECT *)
- **Column** store:
  - Columns are stored consecutively
  - Optimal for attribute focused access (e.g. SUM, GROUP BY)
- Note: concept is **independent** from storage type



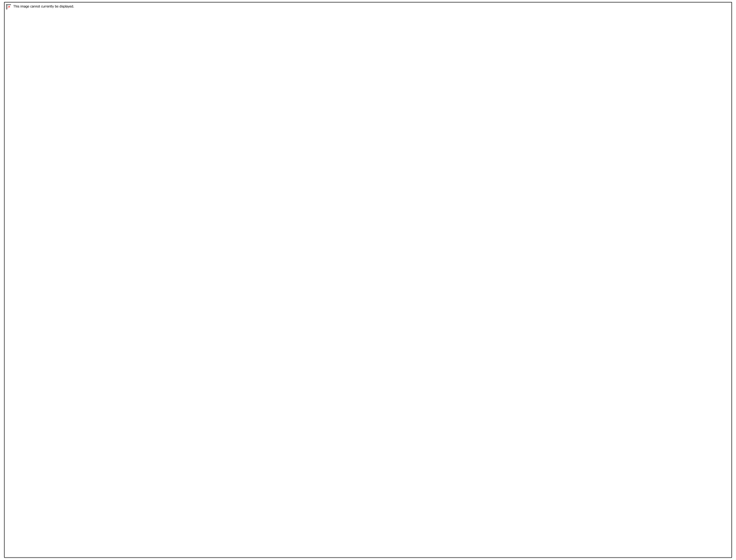
Row Data Layout

- Data is stored tuple-wise
- Leverage co-location of attributes for a single tuple
- Low cost for tuple 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



Row-oriented storage

A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4

Row-oriented storage

A1	B1	C1

A2	B2	C2
A3	B3	C3
A4	B4	C4

Row-oriented storage

A1 B1 C1 A2 B2 C2

A3 B3 C3

A4 B4 C4

Row-oriented storage

A1	B1	C1	A2	B2	C2	A3	B3	C3

A4	B4	C4

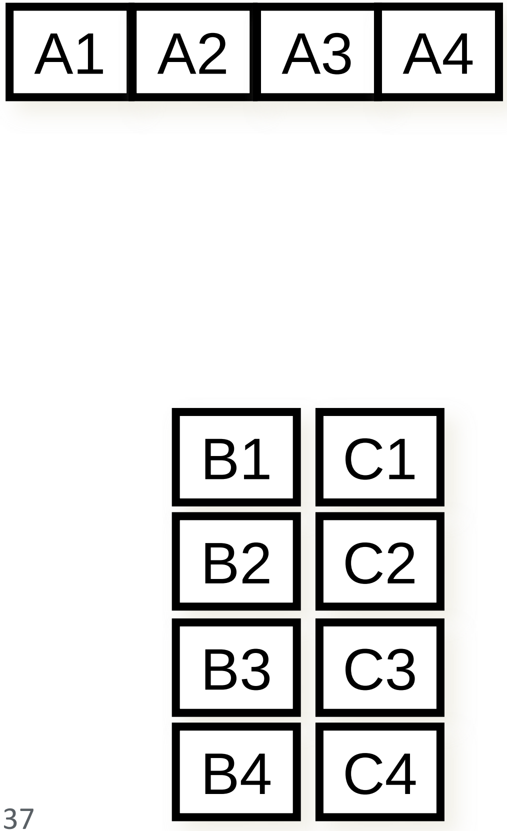
Row-oriented storage

A1	B1	C1	A2	B2	C2	A3	B3	C3	A4	B4	C4

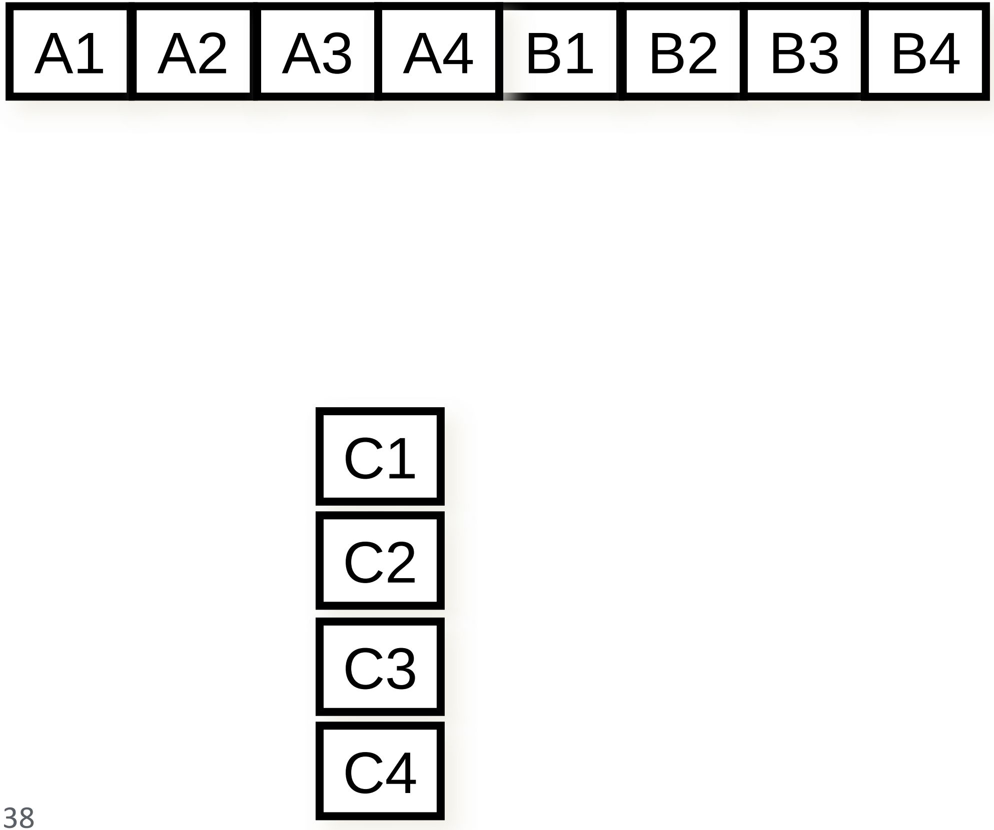
Column-oriented storage

A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4

Column-oriented storage



Column-oriented storage



Column-oriented storage

A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	C4

Dictionary Encoding

Motivation

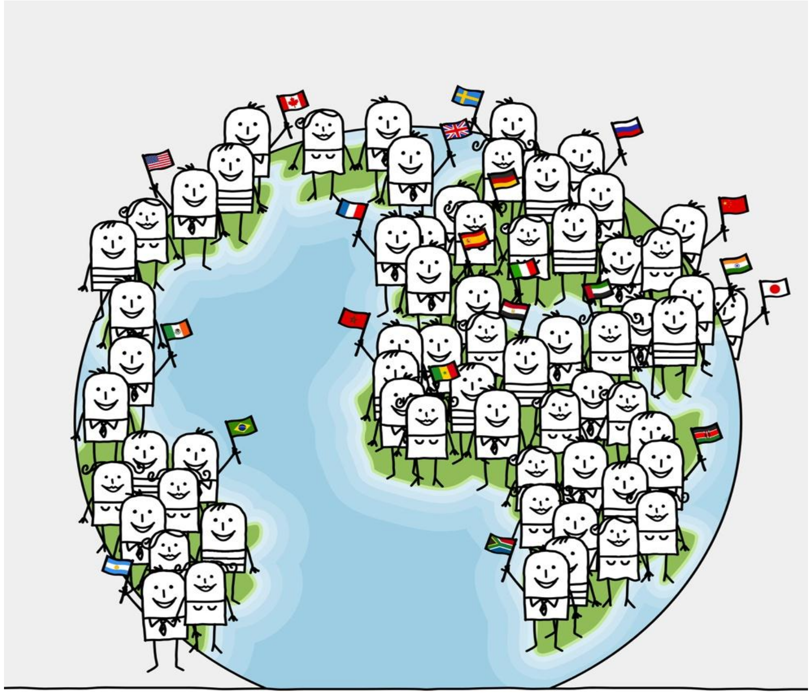
- Main memory access is the new bottleneck
- Idea: **Trade** CPU time to compress and decompress data
- Compression reduces number of memory accesses
- Leads to **less** cache misses due to more information on a cache line
- Operation directly on compressed data possible
- Offsetting with bit-encoded 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

rec ID	fname	lname	gender	city	country	birthday
…	…	…	…	…	…	…
39	John	Smith	m	Chicago	USA	12.03.1964
40	Mary	Brown	f	London	UK	12.05.1964
41	Jane	Doe	f	Palo Alto	USA	23.04.1976
42	John	Doe	m	Palo Alto	USA	17.06.1952
43	Peter	Schmidt	m	Potsdam	GER	11.11.1975
…	…	…	…		…	…

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

Dictionary Encoding a Column

Rec ID	fname
…	…
39	John
40	Mary
41	Jane
42	John
43	Peter
…	…

Dictionary for "fname"

Value ID	Value
…	…
23	Jane
24	John
…	…
28	Mary
29	Peter

Attribute Vector for "fname"

position	Value ID
…	…
39	24
40	28
41	23
42	24
43	29
…	…

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

Column	Cardi-nality	Bits Needed	Item Size	Plain Size	Size with Dictionary (Dictionary + Column)	Compression Factor
First name	5 million	23 bit	50 Byte	373 GB	238.4 MB + 21.4 GB	≈ 17
Last name	8 million	23 bit	50 Byte	373 GB	381.5 MB + 21.4 GB	≈ 17
Gender	2	1 bit	1 Byte	8 GB	2 bit + 1 GB	≈ 8
City	1 million	20 bit	50 Byte	373 GB	47.7 MB + 18.6 GB	≈ 20
Country	200	8 bit	47 Byte	350 GB	9.2 KB + 7.5 GB	≈ 47
Birthday	40,000	16 bit	2 Byte	15 GB	78.1 KB + 14.9 GB	≈ 1
Totals			**200 Byte**	**≈ 1.6 TB**	**≈ 92 GB**	**≈ 17**

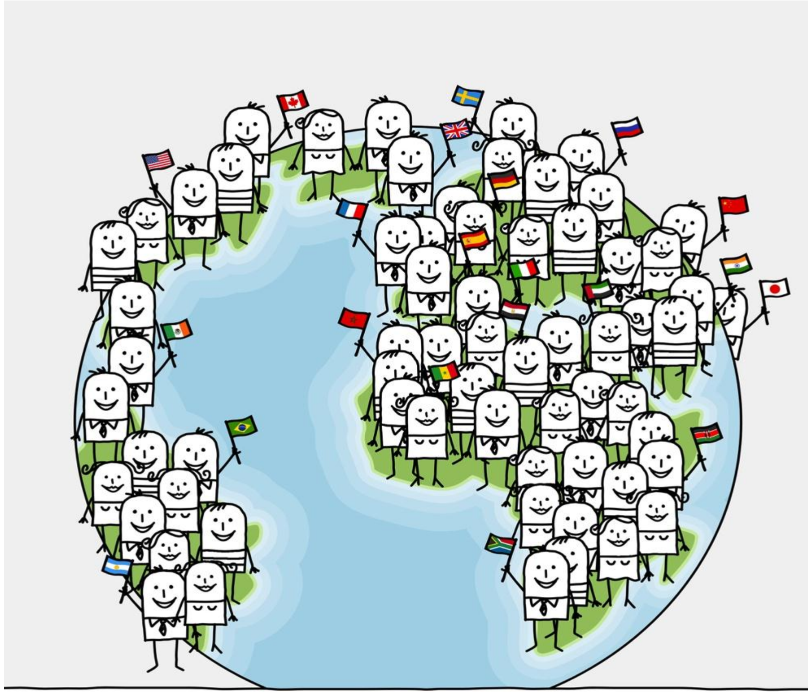
Chapter 3:

In-Memory Database Operators

Scan Performance (I)

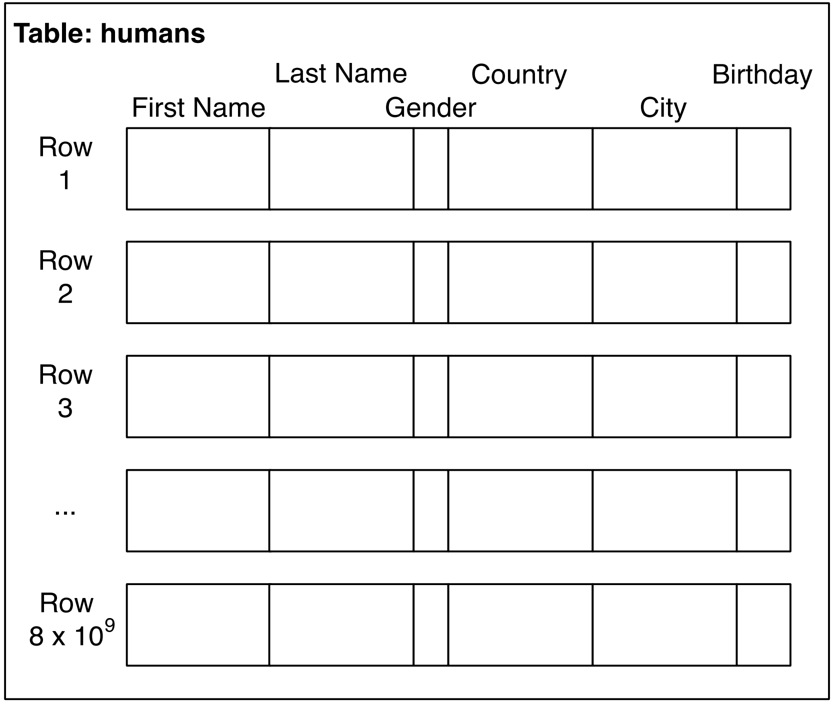
8 billion humans

- Attributes
  - First Name
  - Last Name
  - Gender
  - Country
  - City
  - Birthday
  - ➔ 200 byte
- Question: How many men/women?
- Assumed scan speed: 2 MB/ms/core



Scan Performance (2)

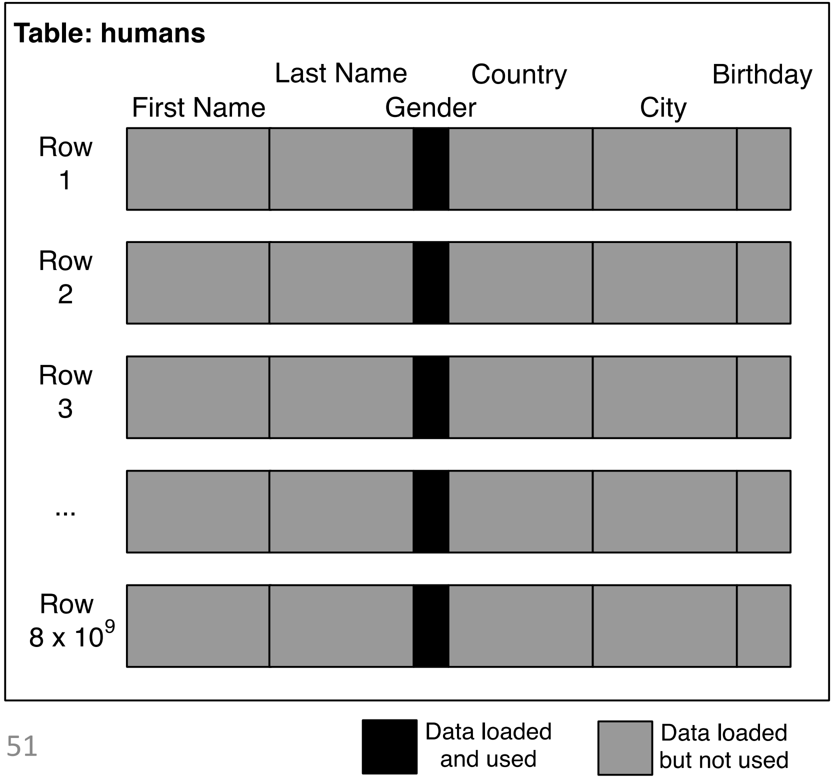
Row Store – Layout



- Table size = 8 billion tuples x 200 bytes per tuple → ~**1.6 TB**
- Scan through all rows with 2 MB/ms/core → ~**800 seconds** with 1 core

Scan Performance (3)

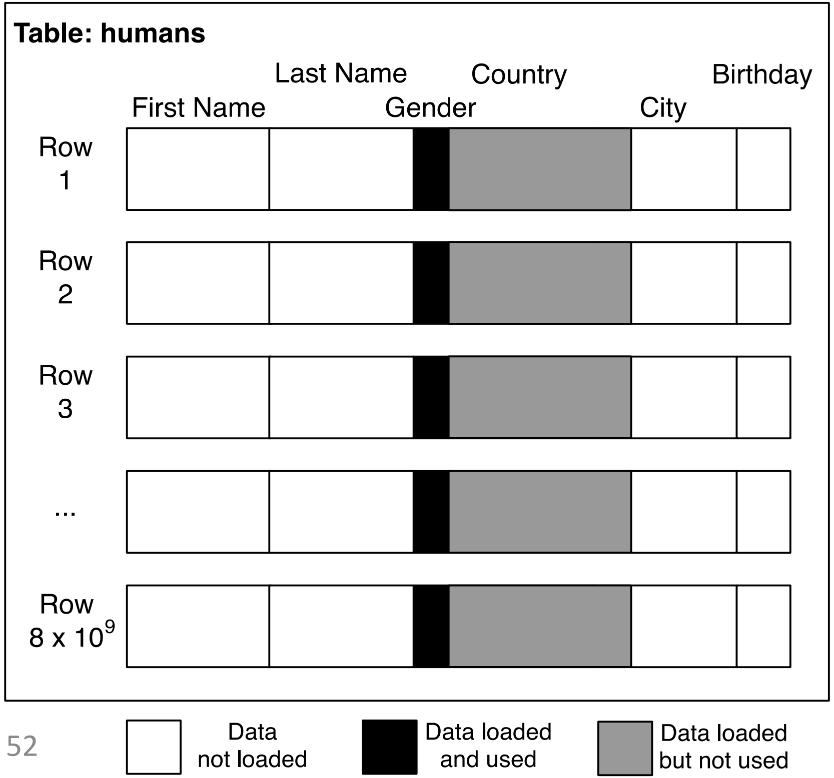
Row Store – Full Table Scan



- Table size = 8 billion tuples x 200 bytes per tuple → ~**1.6 TB**
- Scan through all rows with 2 MB/ms/core → ~**800 seconds** with 1 core

Scan Performance (4)

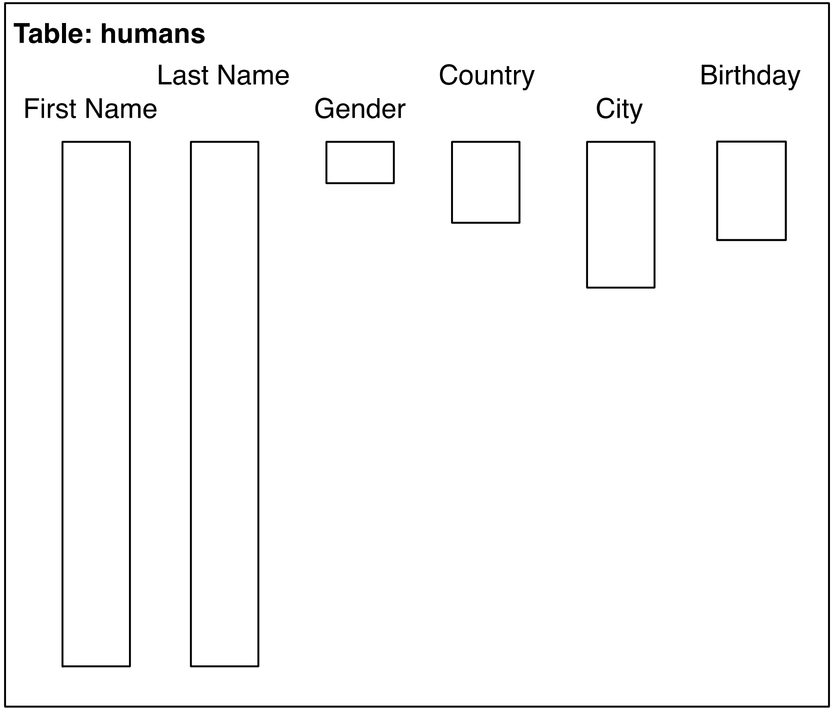
Row Store – Stride Access "Gender"



- 8 billion cache accesses à 64 byte
  → **~512 GB**
- Read with 2 MB/ms/core
  → **~256 seconds** with 1 core

Scan Performance (5)

Column Store – Layout

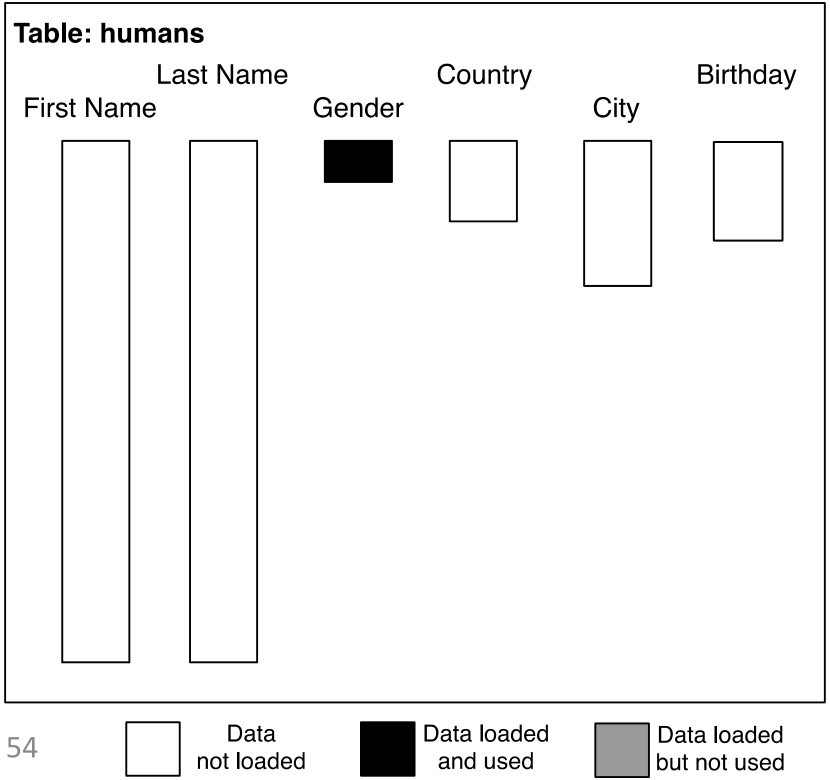


- Table size
  - Attribute vectors: ~**91 GB**
  - Dictionaries: ~**700 MB**

  → Total: ~**92 GB**
- Compression factor: ~**17**

Scan Performance (6)

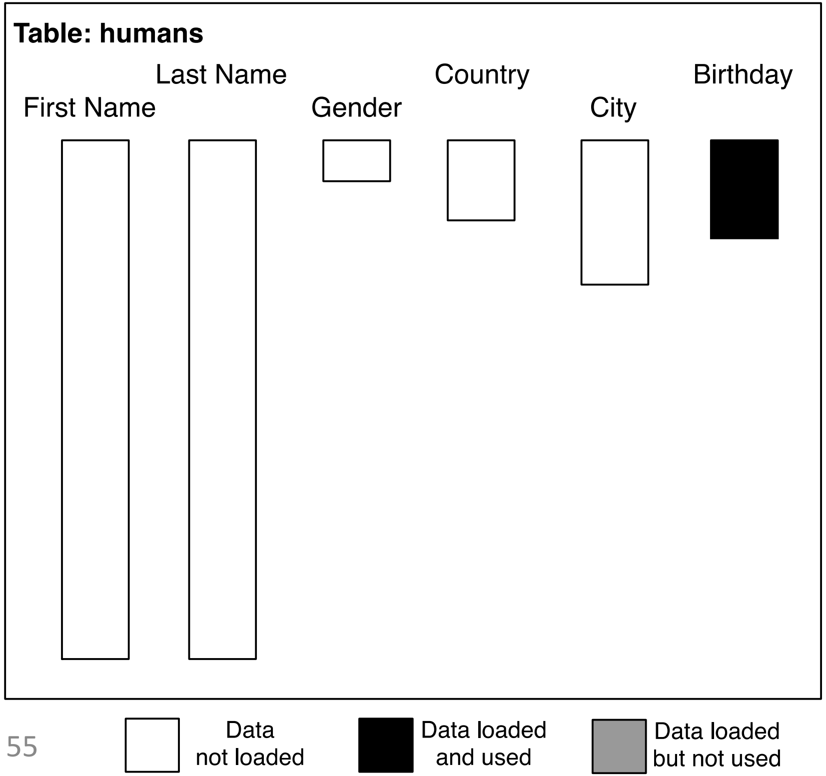
Column Store – Full Column Scan on "Gender"



- Size of attribute vector "gender" = 8 billion tuples x 1 bit per tuple → **~1 GB**
- Scan through attribute vector with 2 MB/ms/core → **~0.5 seconds** with 1 core

Scan Performance (7)

Column Store – Full Column Scan on “Birthday”



- Size of attribute vector “birthday” = 8 billion tuples x 2 Byte per tuple → ~**16 GB**
- Scan through column with 2 MB/ms/core → ~**8 seconds** with 1 core

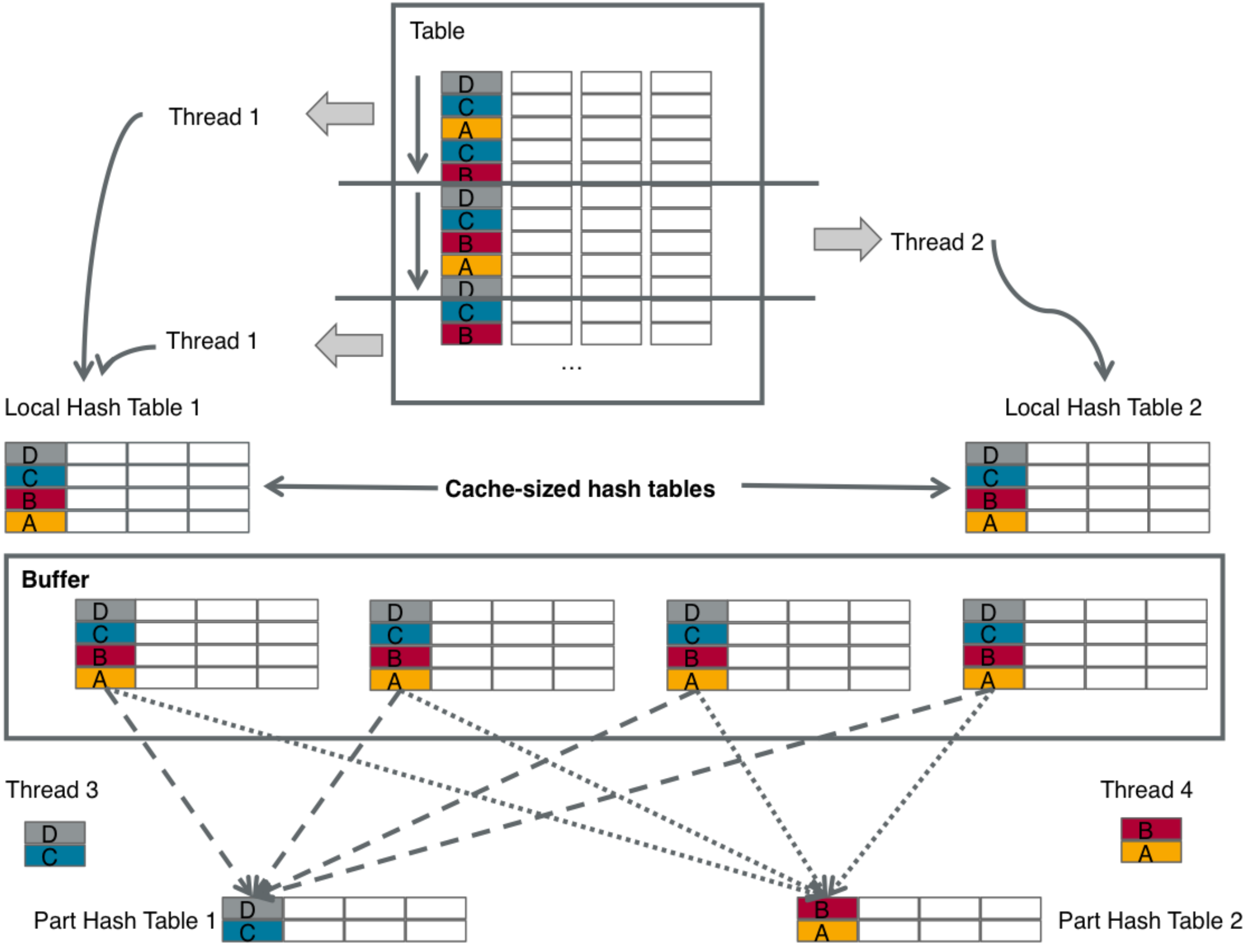
Scan Performance – Summary

- How many women, how many men?

	Column Store	Row Store	
		Full table scan	Stride access
Time in seconds	0.5	800	256
		1,600x slower	512x slower

Parallel Aggregation

- **1.) *n* Aggregation Threads**
  - 1) each thread fetches a small part of the input relation
  - 2) aggregate part and write results into a small hash-table
  - **If the entries in a hash-table exceed a threshold, the hash-table is moved into a shared buffer**
- **2.) *m* Merger Threads**
  - 3) each merge thread operates on a partition of the hash function values and writes its result into a private part hash-table
  - 4) the final result is obtained by concatenating the part hash-tables

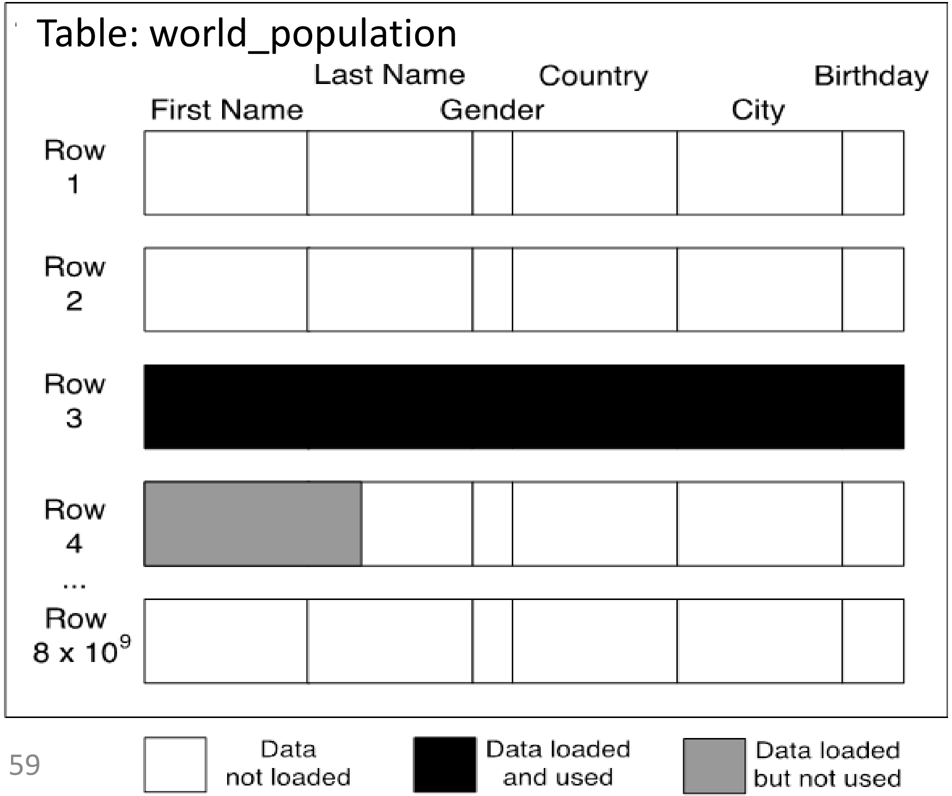


Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Tuple Reconstruction

Tuple Reconstruction (I)

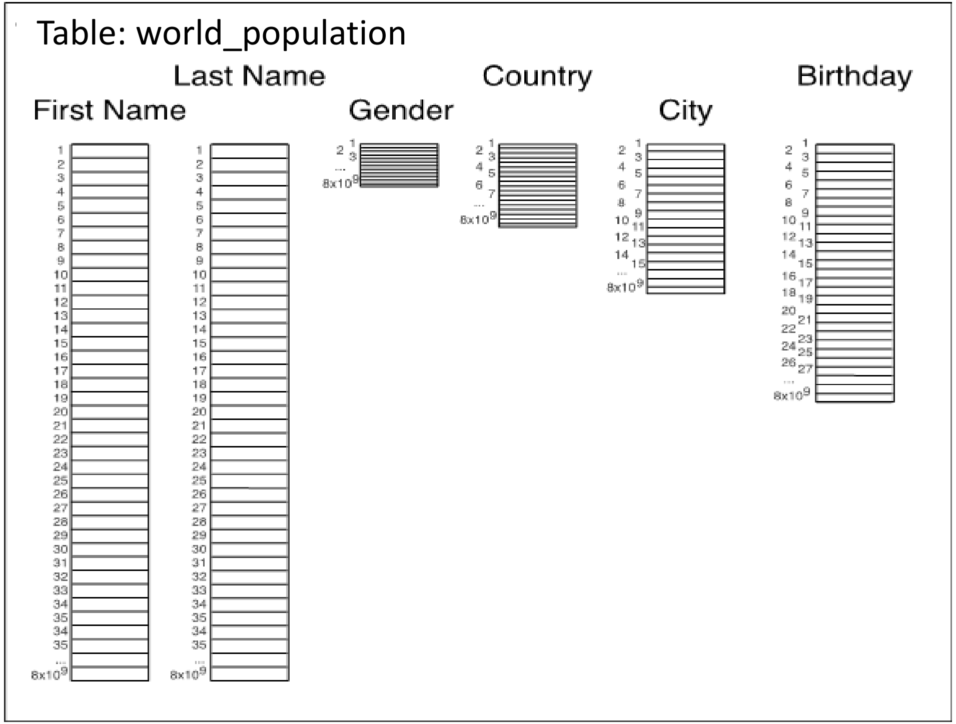
Accessing a record in a row store



- All attributes are stored consecutively
- 200 byte → 4 cache accesses à 64 byte → **256 byte**
- Read with 2 MB/ms/core **→ ~0.128 µs** with 1 core

Tuple Reconstruction (2)

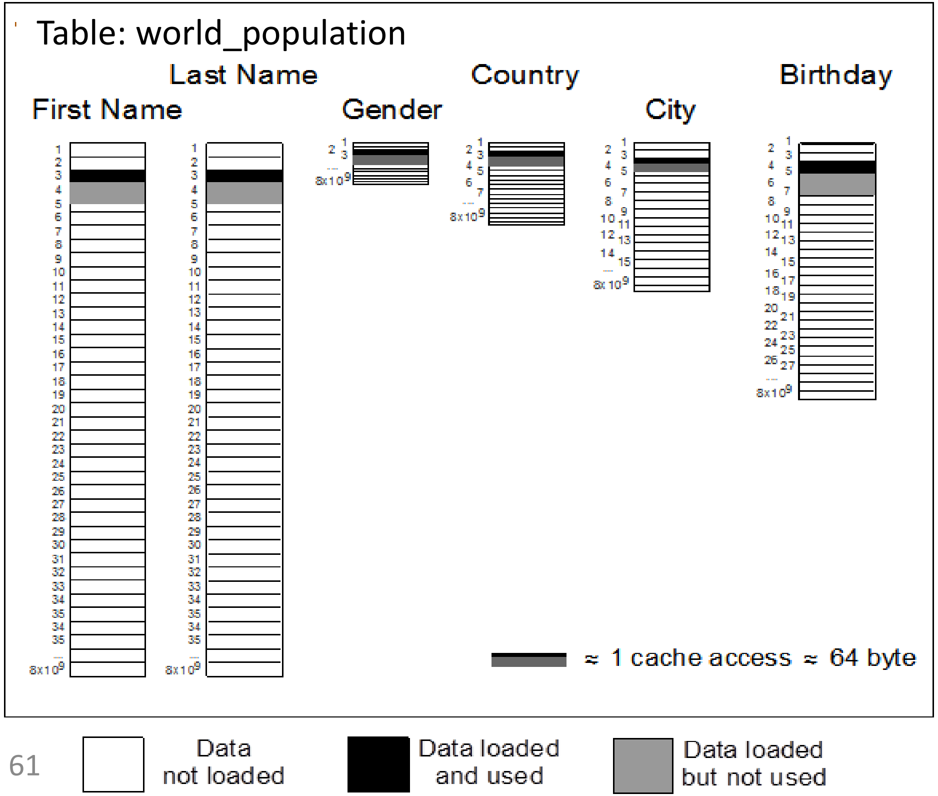
Column store



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

Tuple Reconstruction (3)

Column store



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

Select

SELECT Example

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

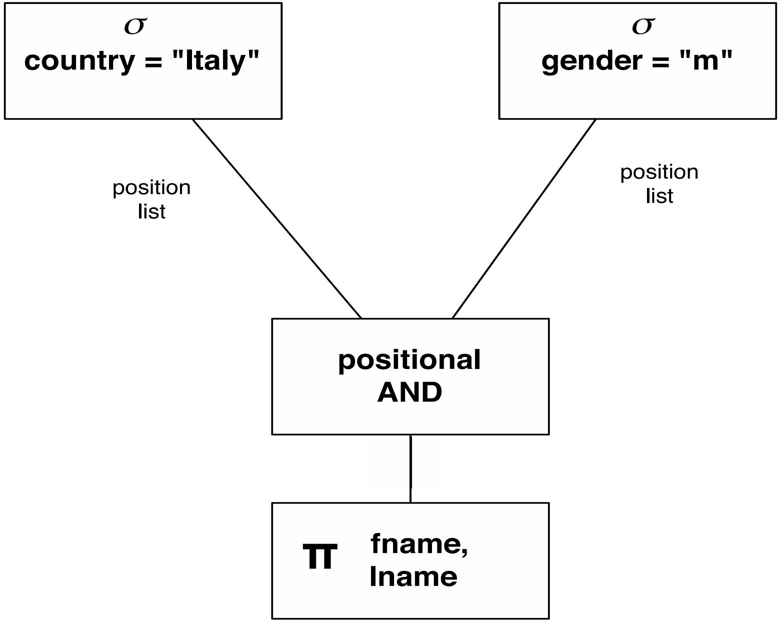
fname	lname	country	gender
Gianluigi	Buffon	Italy	m
Lena	Gercke	Germany	f
Mario	Balotelli	Italy	m
Manuel	Neuer	Germany	m
Lukas	Podolski	Germany	m
Klaas-Jan	Huntelaar	Netherlands	m

Query Plan

- Multiple plans possible to execute this 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
	…

fname	lname	country	gender
Gianluigi	Buffon	3	1
Lena	Gercke	2	0
Mario	Balotelli	3	1
Manuel	Neuer	2	1
Lukas	Podolski	2	1
Klaas-Jan	Huntelaar	4	1

Value ID	Dictionary for "gender"
0	f
1	m

country = 3 ("Italy")

Position
0
2

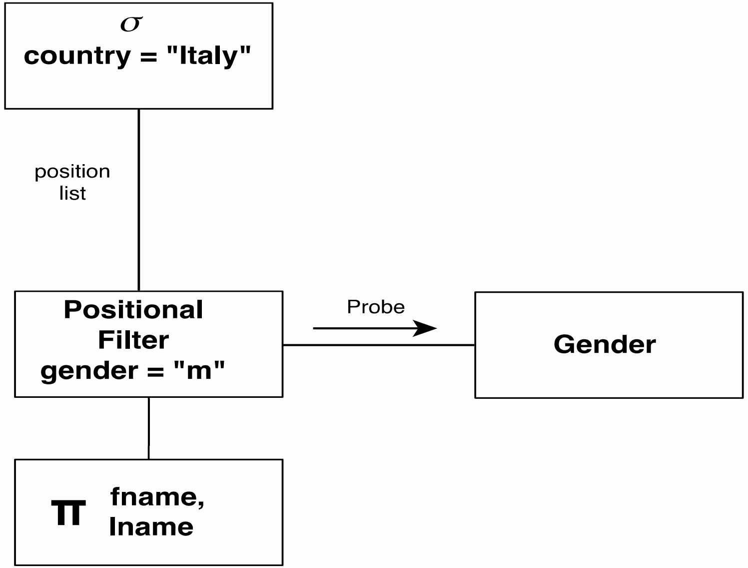
gender = 1 ("m")

Position
0
2
3
4
5

AND

Position
0
2

Query Plan (ii)



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

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Insert

Insert

- Insert is the dominant modification operation
  - Delete/Update can be modeled as Inserts as well (Insert-only approach)
- 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

world_population

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

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)

AV

0	0
1	1
2	3
3	2
4	3

D

0	Albrecht
1	Berg
2	Meyer
3	Schulze

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

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)

AV

0	0
1	1
2	3
3	2
4	3

D

0	Albrecht
1	Berg
2	Meyer
3	Schulze

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

Attribute Vector (AV)
Dictionary (D)

1. Look-up on D → entry found

INSERT (I) w/o new Dictionary entry

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

AV

	AV
0	0
1	1
2	3
3	2
4	3
5	3

D

	D
0	Albrecht
1	Berg
2	Meyer
3	Schulze

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

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)

AV

0	0
1	0
2	1
3	2
4	3

D

0	Berlin
1	Hamburg
2	Innsbruck
3	Potsdam

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

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)

AV

0	0
1	0
2	1
3	2
4	3

D

0	Berlin
1	Hamburg
2	Innsbruck
3	Potsdam

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

1. Look-up on D → **no** entry found

INSERT (2) with new Dictionary Entry I/II

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

	AV
0	0
1	0
2	1
3	2
4	3

	D
0	Berlin
1	Hamburg
2	Innsbruck
3	Potsdam
4	Rostock

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

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

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

	AV
0	0
1	0
2	1
3	2
4	3
5	$\boxed{4}$

	D
0	Berlin
1	Hamburg
2	Innsbruck
3	Potsdam
4	Rostock

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze			Rostock	
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

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)

AV

	AV
0	2
1	3
2	1
3	0
4	4

D

	D
0	Anton
1	Hanna
2	Martin
3	Michael
4	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze			Rostock	
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

INSERT (2) with new Dictionary Entry II/II

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

	AV		D
0	2	0	Anton
1	3	1	Hanna
2	1	2	Martin
3	0	3	Michael
4	4	4	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze			Rostock	
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

1. Look-up on D → **no** entry found

INSERT (2) with new Dictionary Entry II/II

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

AV

	AV
0	2
1	3
2	1
3	0
4	4

D

	D
0	Anton
1	Hanna
2	$\boxed{\text{Karen}}$
3	Martin
4	Michael
5	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze			Rostock	
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

1. Look-up on D → **no** entry found
2. Insert new value to D

INSERT (2) with new Dictionary Entry II/II

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

AV

	AV
0	2
1	3
2	1
3	0
4	4

D (old)

	D (old)
0	Anton
1	Hanna
2	Martin
3	Michael
4	Sophie

→

D (new)

	D (new)
0	Anton
1	Hanna
2	Karen
3	Martin
4	Michael
5	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze			Rostock	
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

1. Look-up on D → **no** entry found
2. Insert new value to D

INSERT (2) with new Dictionary Entry II/II

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

AV (old) → AV (new)

	AV (old)		AV (new)
0	2	0	3
1	3	1	4
2	1	2	1
3	0	3	0
4	4	4	5

D (new)

	D (new)
0	Anton
1	Hanna
2	Karen
3	Martin
4	Michael
5	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze			Rostock	
...	...	...	...	...	...	...

Attribute Vector (AV)
Dictionary (D)

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

INSERT (2) with new Dictionary Entry II/II

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

Changed Value IDs

	AV		D
0	3	0	Anton
1	4	1	Hanna
2	1	2	Karen
3	0	3	Martin
4	5	4	Michael
5	2	5	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5	Karen	Schulze			Rostock	
...	...	...	...	...	...	...

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

Result

world_population

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

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

Chapter 4:

Advanced Database Storage Techniques

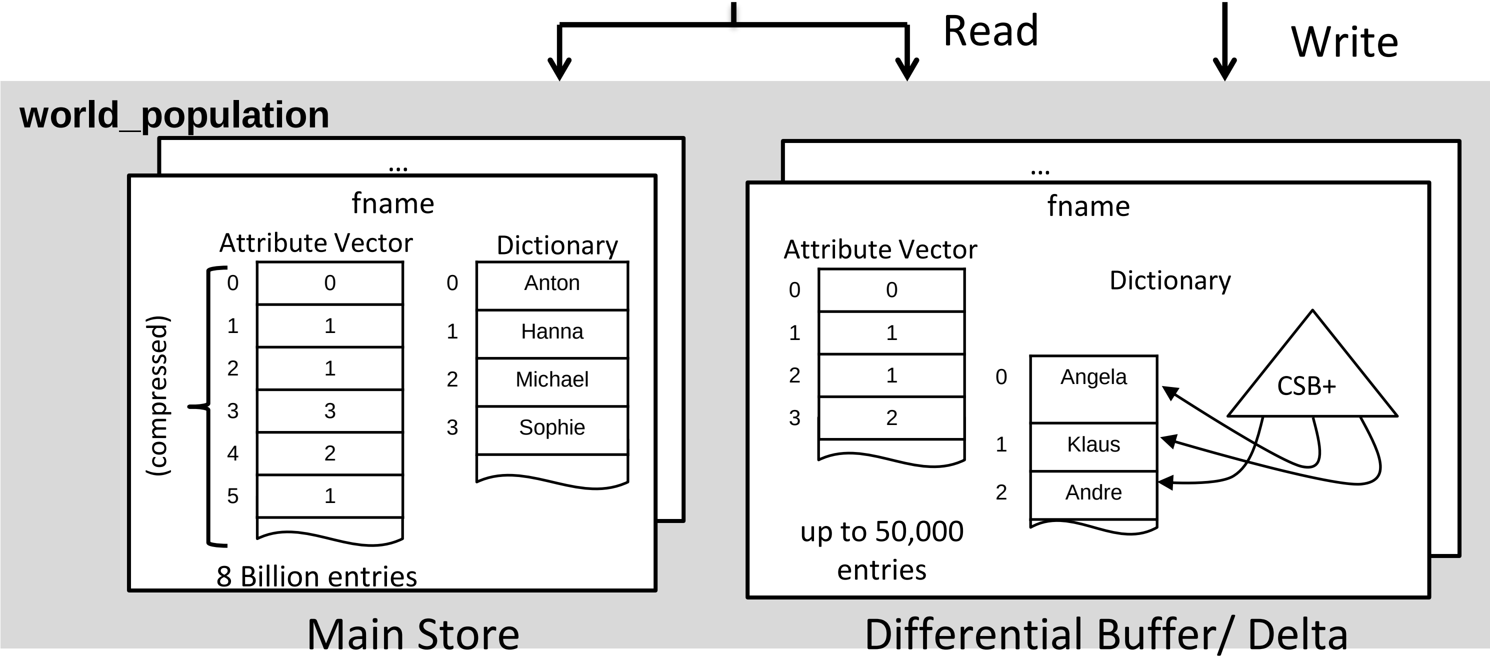
Differential Buffer

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



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 value IDs
  - Scans with range selection need to lookup values in dictionary for comparisons
- Differential Buffer requires more memory:
  - Attribute vector not bit-compressed
  - Additional CSB+ Tree for dictionary

Tuple Lifetime

Michael moves from Berlin to Potsdam

Main Table: world_population

recId	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
...	...	...	...	...	...	...
$8 * 10^9$	Zacharias	Perdopolus	m	GRE	Athen	03-12-1979

Main Store

Differential Buffer

UPDATE "world_population"
SET city="Potsdam"
WHERE fname="Michael" AND lname="Berg"

Tuple Lifetime

Michael moves from Berlin to Potsdam

Main Table: world_population

recId	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
...	...	...	...	...	...	...
$8 * 10^9$	Zacharias	Perdopolus	m	GRE	Athen	03-12-1979

Main Store

Differential Buffer

UPDATE "world_population"
SET city="Potsdam"
WHERE fname="Michael" AND lname="Berg"

Tuple Lifetime

Michael moves from Berlin to Potsdam

Main Table: world_population

Main Store

recId	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
...	...	...	...	...	...	...
$8 * 10^9$	Zacharias	Perdopolus	m	GRE	Athen	03-12-1979

Differential Buffer

0	**Michael**	**Berg**	**m**	**GER**	**Potsdam**	**03-05-1970**

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
  - Additional 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

Main Store

recId	fname	lname	gender	country	city	birthday	valid
0	Martin	Albrecht	m	GER	Berlin	08-05-1955	1
1	Michael	Berg	m	GER	Berlin	03-05-1970	0
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968	1
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992	1
...	...	...	...	...	...	...	
$8 * 10^9$	Zacharias	Perdopolus	m	GRE	Athen	03-12-1979	1

Differential Buffer

recId	fname	lname	gender	country	city	birthday	valid
0	**Michael**	**Berg**	**m**	**GER**	**Potsdam**	**03-05-1970**	**1**

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

Merge

Handling Write Operations

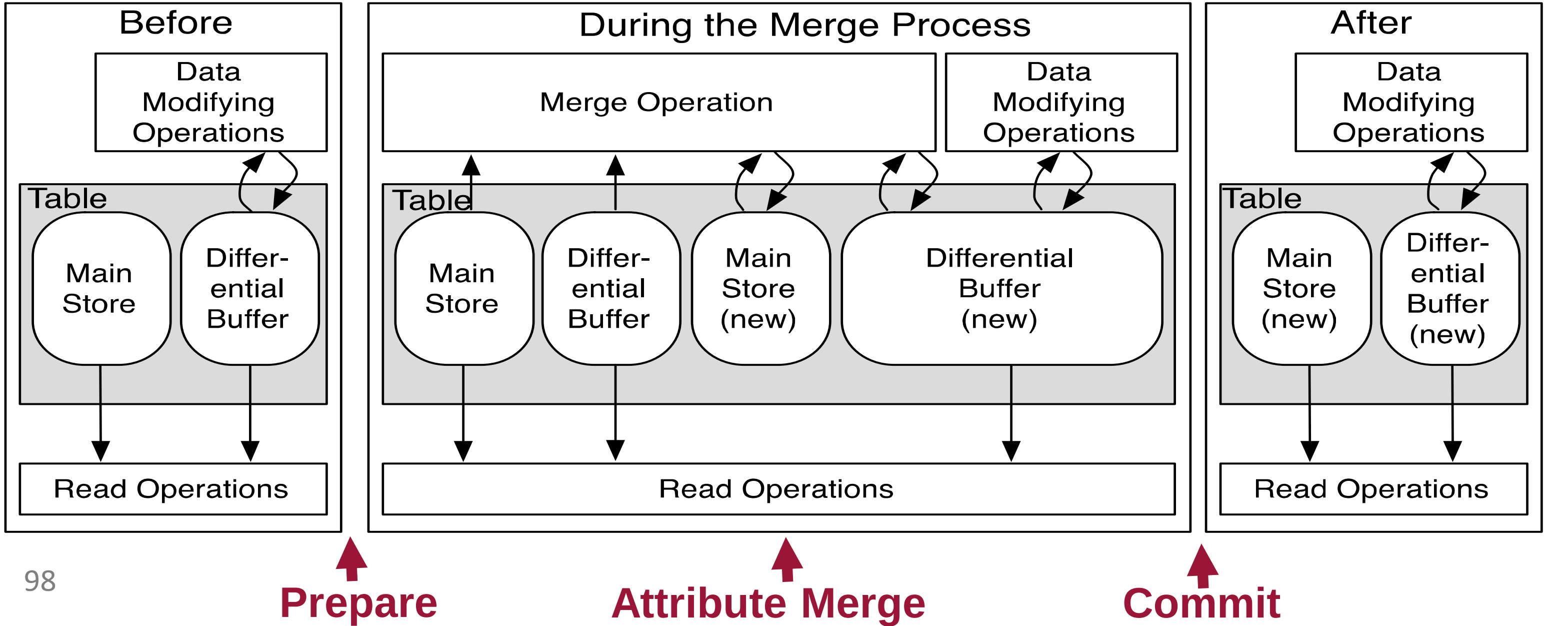
- All Write operations (INSERT, UPDATE) are stored within a differential buffer (delta) first
- Read-operations on differential buffer are more expensive than on main store
- Differential buffer is merged periodically with the main store
  - To avoid performance degradation based on large delta
  - Merge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



Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Chapter 5:

Implications on Application Development

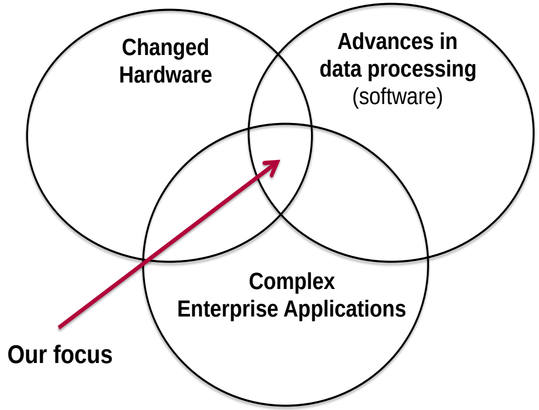
How does it all come together?

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

How does it all come together?

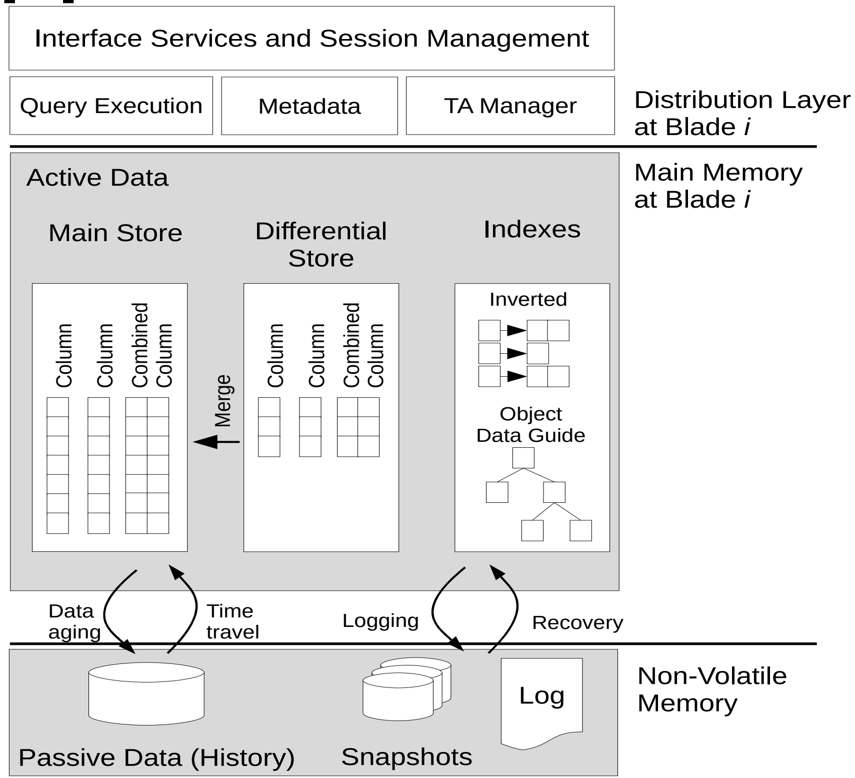
3. Mostly read workload

- 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

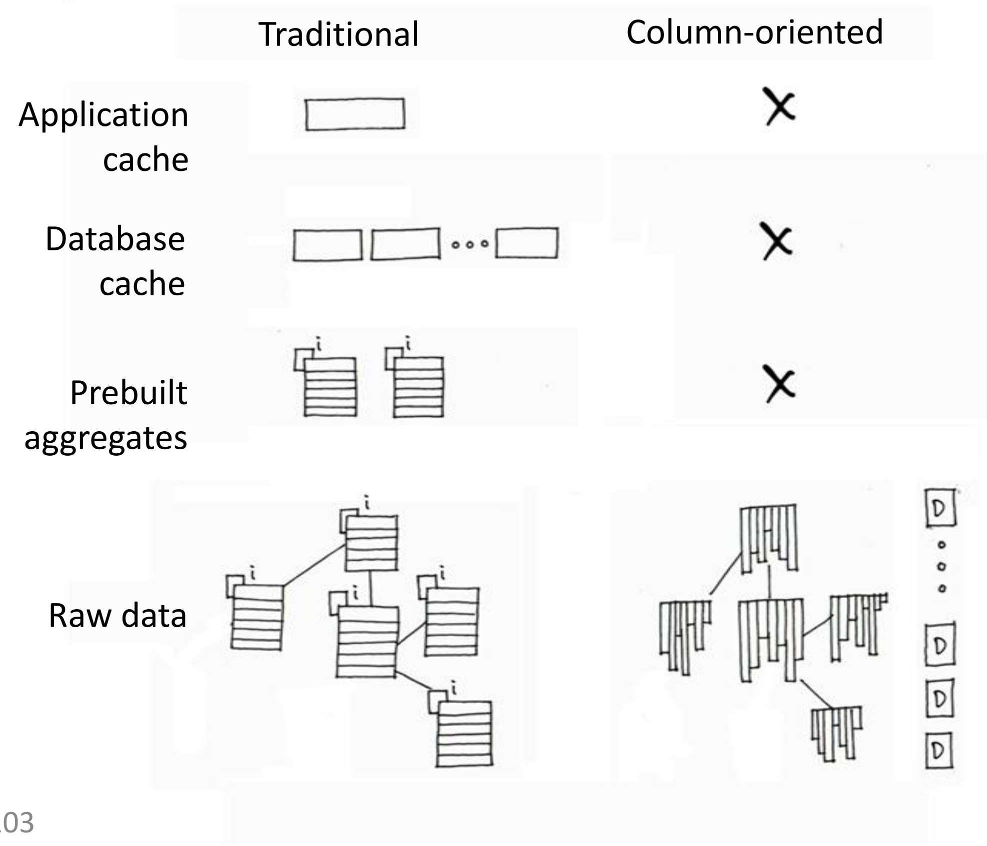


An In-Memory Database for Enterprise Applications

- **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



- 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

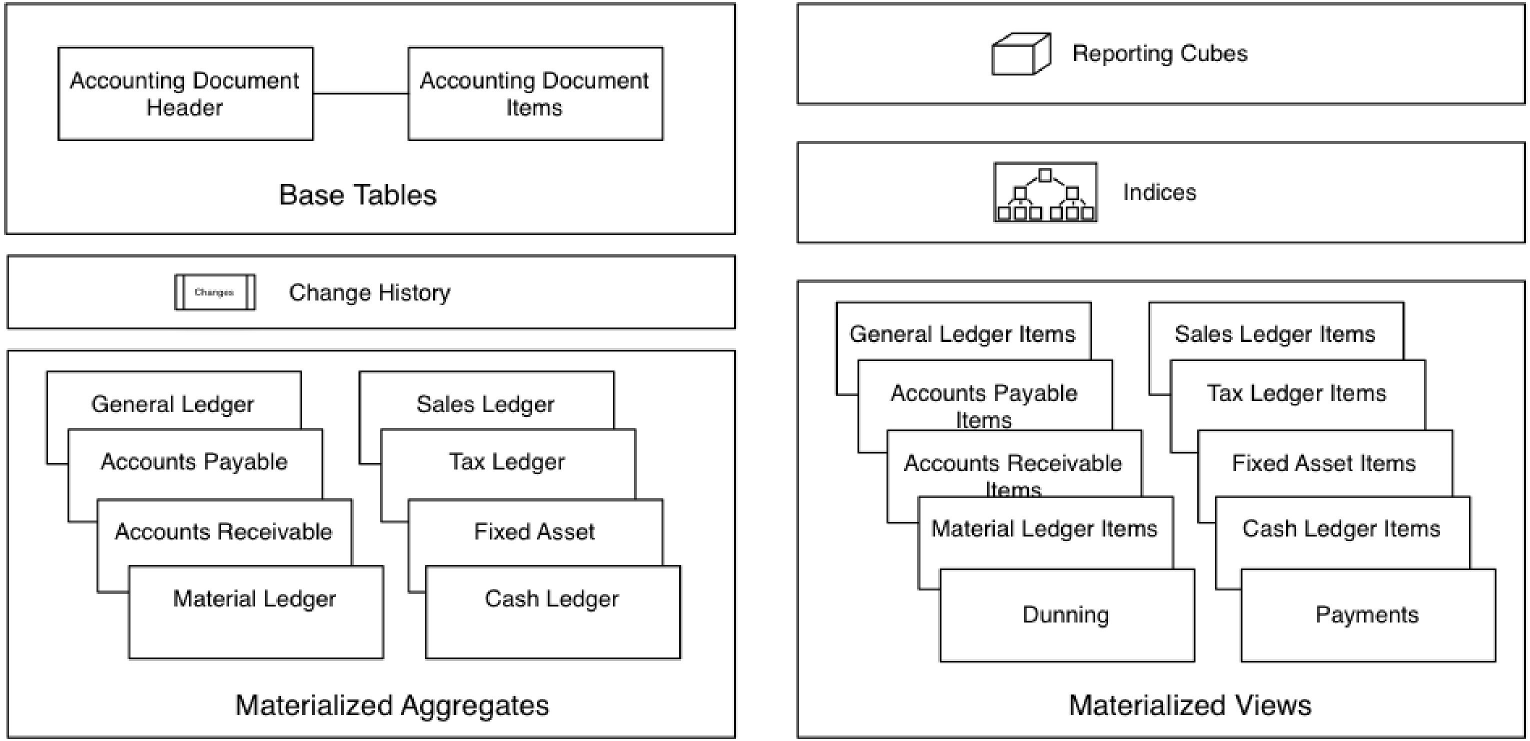
- Combined workload (parallel OLTP/OLAP queries)
- Leverage in-memory capabilities to
  - Reduce amount of data
  - Aggregate on-the-fly
  - Run analytic-style queries (to replace materialized views)
  - Execute stored procedures

SAP ERP Financials on In-Memory Technology

In-memory column database for an ERP system

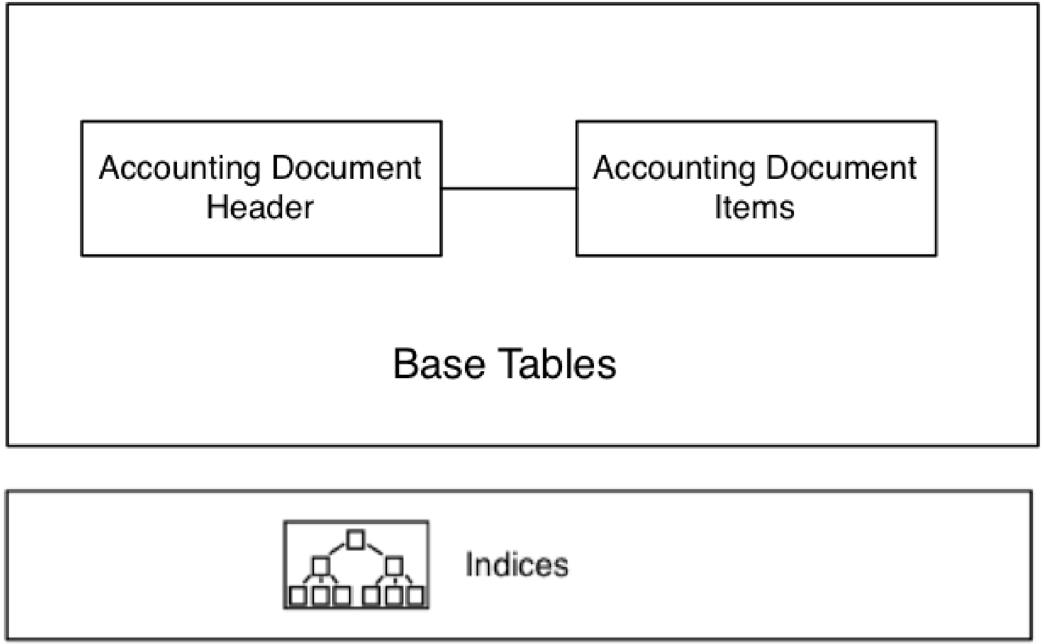
- Use Case: **SAP ERP Financials** solution
  - Post and change documents
  - Display open items
  - Run dunning job
  - Analytical queries, such as balance sheet

Current Financials Solutions



The Target Financials Solution

Only base tables, algorithms, and some indices



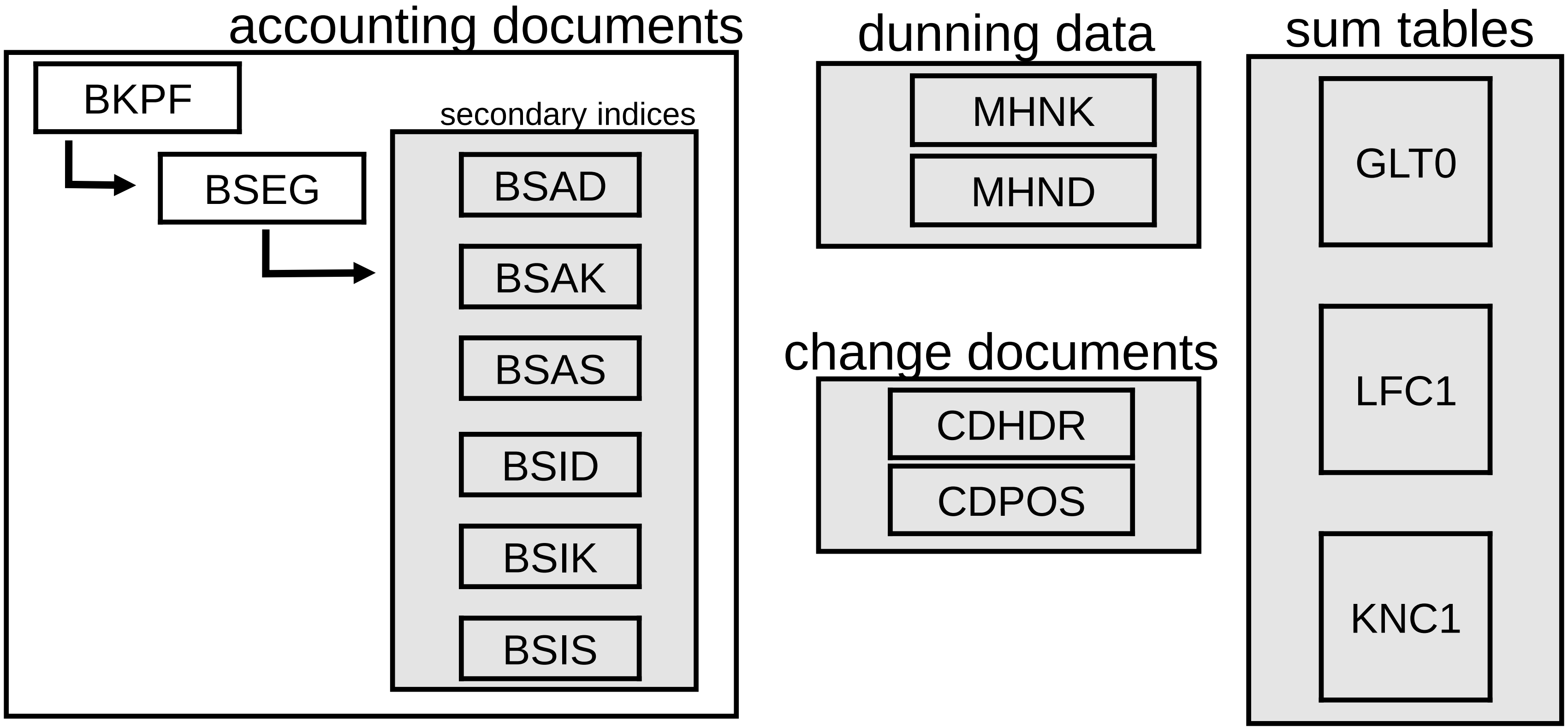
Feasibility of Financials on In-Memory Technology in 2009

- 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

Feasibility of Financials on In-Memory Technology in 2009

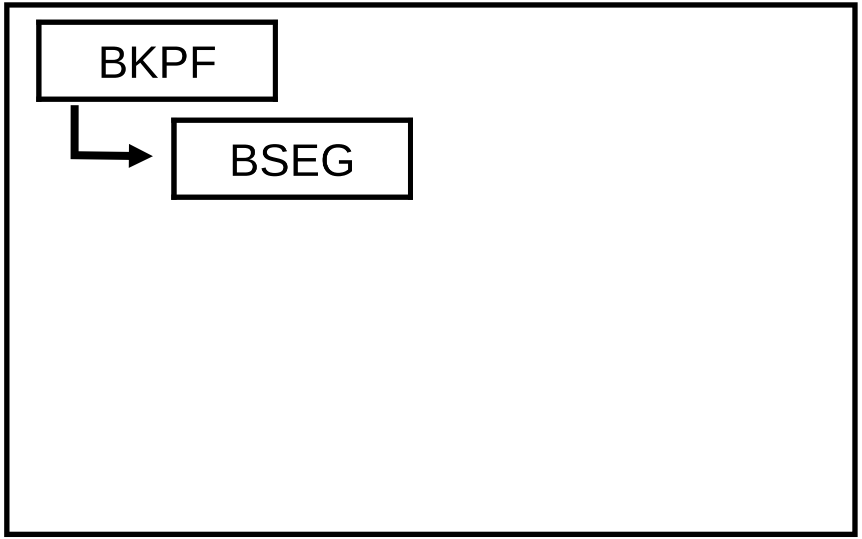
- European division of a retailer
  - ERP 2005 ECC 6.0 EhP3
  - 5.5 TB system database size
  - Financials:
    - **23** million headers / 1.5 GB in main memory
    - **252** million items / 50 GB in main memory
    (including inverted indices for join attributes and insert only extension)

In-Memory Financials on SAP ERP



In-Memory Financials on SAP ERP

accounting documents

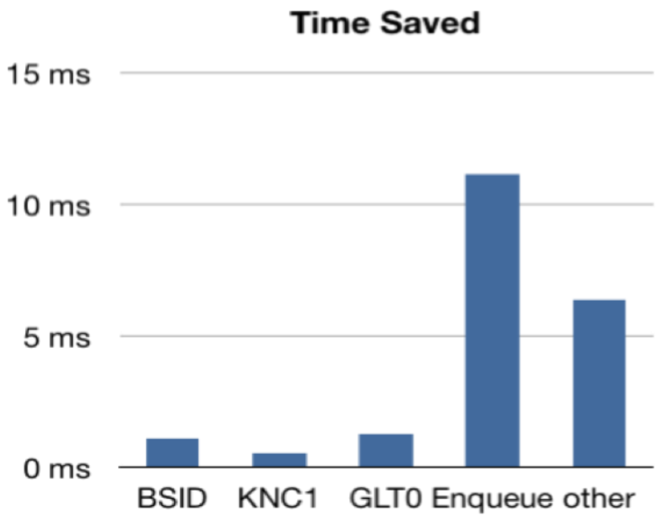


Reduction by a Factor 10

	Classic Row-Store (w/o compr.)	IMDB
BKPF	8.7 GB	1.5 GB
BSEG	255 GB	50 GB
	263.7 GB	**51.5 GB**
Secondary Indices	255 GB	-
Sum Tables	0.55 GB	-
Complete	**519.25 GB**	**51.5 GB**

Booking an accounting document

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



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: **1 SELECT**
  - Select open account items from BSID, for each: **10000 SELECTs**
    - Calculate due date
  - Select dunning procedure, level and area **10000 SELECTs**
    - Create MHNK entries **31000 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

One single stored procedure executed within IMDB

31000 Entries

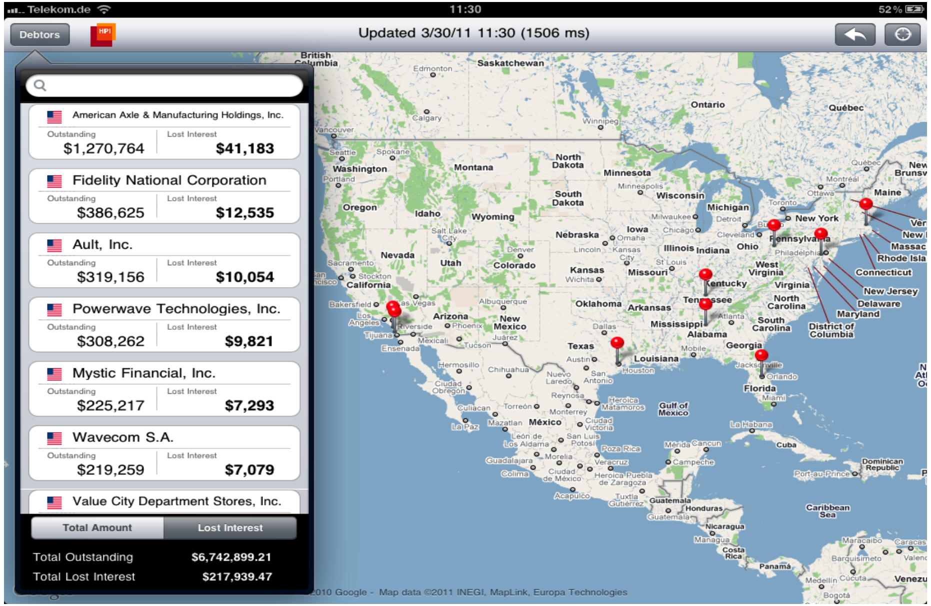
Bring Application Logic Closer to the Storage Layer

- 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

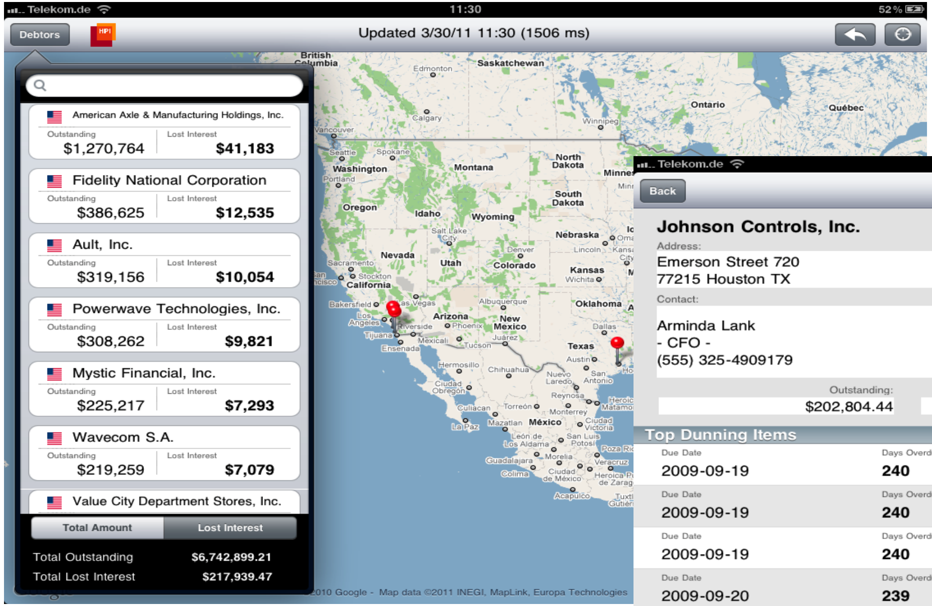
One single stored procedure executed within IMDB

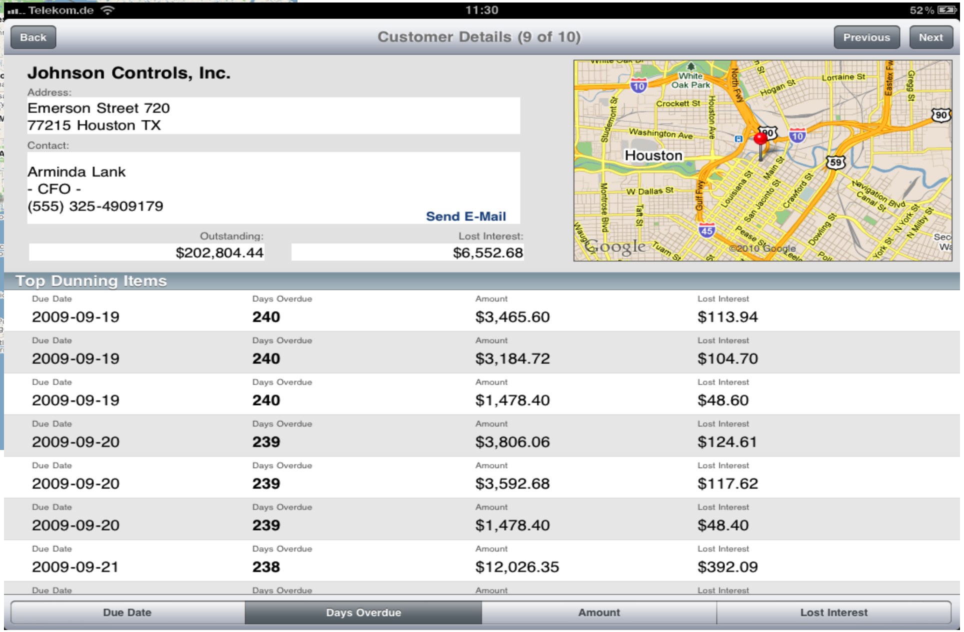
Calculated on-the-fly

Dunning Application



Dunning Application





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

Cooperation with Industry Partners

- Real use cases from industry partners
- Benefit for partners
  - Cutting edge soft- and hardware technologies
  - IT students work on new ideas from scratch or in addition to existing solutions
- Long track of success with partners
  - Multiple global enterprises from retail, engineering and other sectors
  - Improvement of modern ERP and analytical systems
  - New applications for mobile use cases

References

- A Course in In-Memory Data Management, H. Plattner
  http://epic.hpi.uni-potsdam.de/Home/InMemoryBook

- Publications of our Research Group:
  - Papers about the inner-workings of in-mmeory databases
  - http://epic.hpi.uni-potsdam.de/Home/Publications

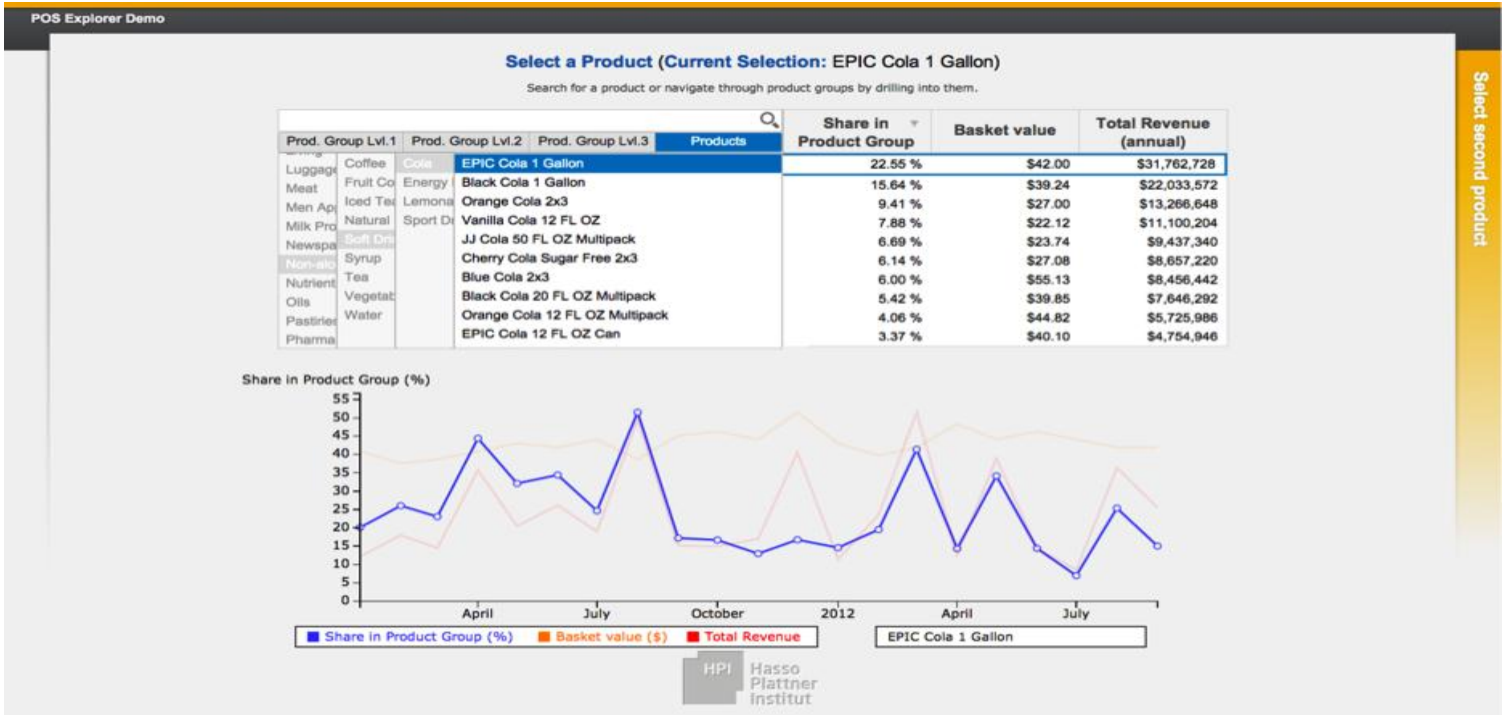
Thank You!

CD216:
Technical Deep-Dive in a Column-Oriented In-Memory Database

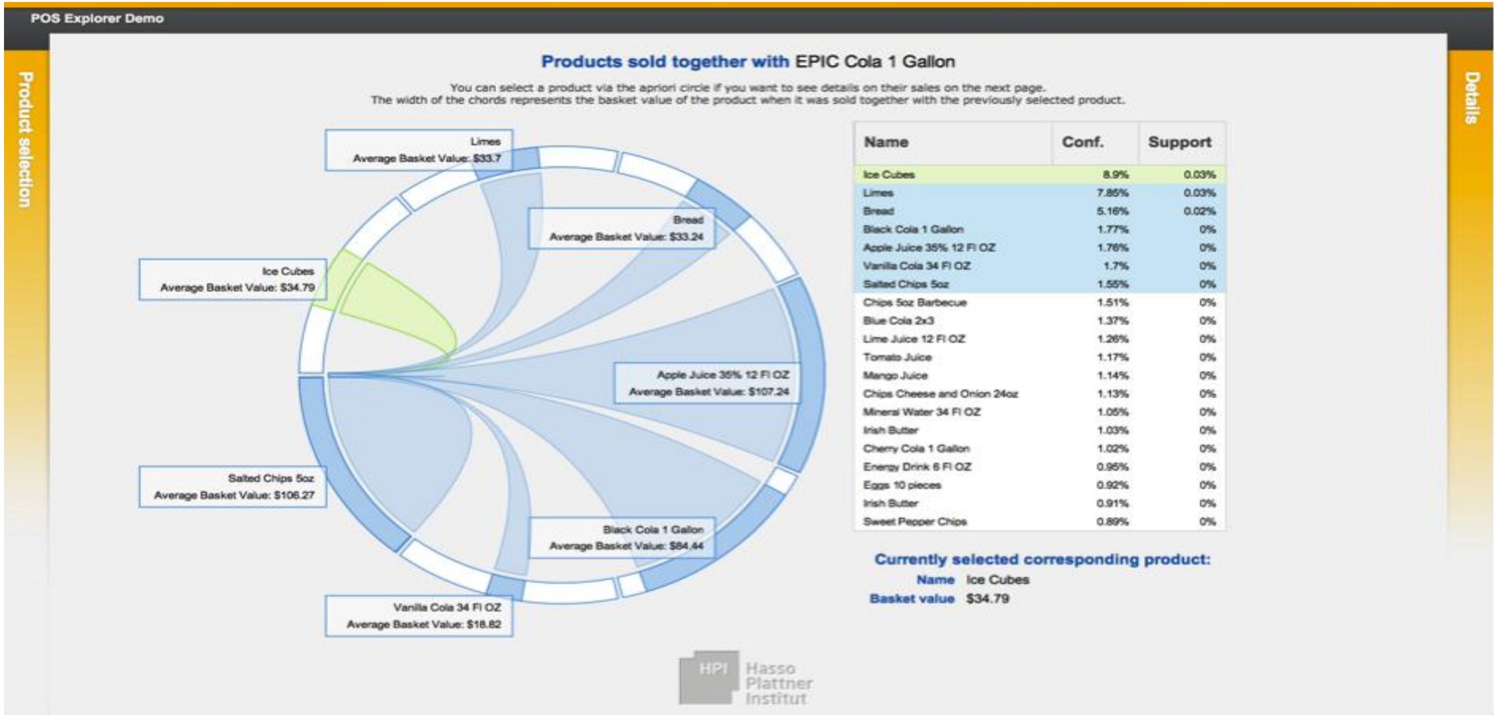
Dr. Jan Schaffner
Research Group of Prof. Hasso Plattner

Hasso Plattner Institute for Software Engineering
University of Potsdam

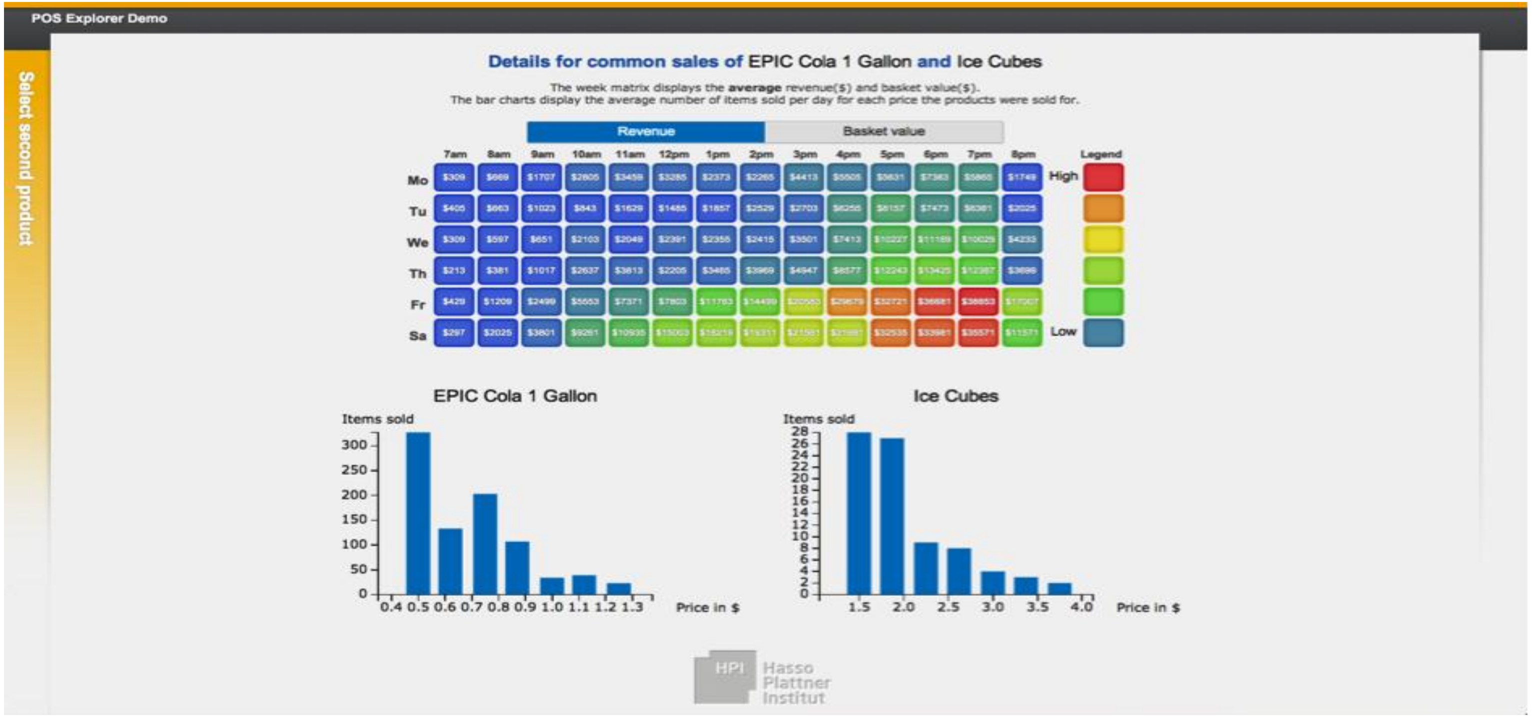
POS Explorer I



POS Explorer II



POS Explorer III



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

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

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 ~3 seconds response time)
  - Front ends for iPhone & iPad

- 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

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