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

Slides by Martin Grund
Presented today by Alberto Lerner

eXascale Infolab
Department of Informatics
University of Fribourg Switzerland

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

2

Enterprise Application Characteristics

OLTP vs. OLAP

Onl**i**ne **T**ransaction **P**rocessing

Onl**i**ne **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)

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

4

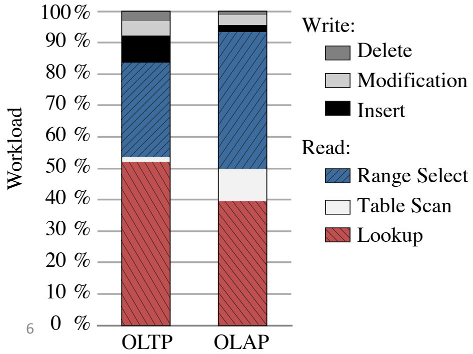
Drawbacks of the Separation

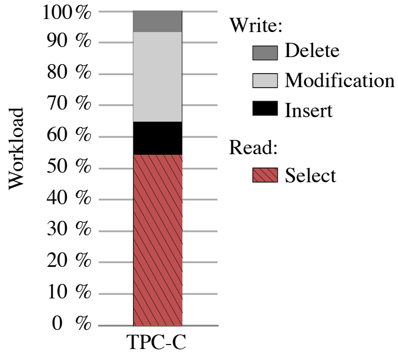
- OLAP systems do not have the **latest** data
- OLAP systems only have **predefined subset** of the data
- **Cost-intensive ETL** processes have to sync both systems
- Separation introduces data **redundancy**
- **Different data schemas** introduce complexity for applications combining sources

5

Enterprise Workloads are Read-Mostly

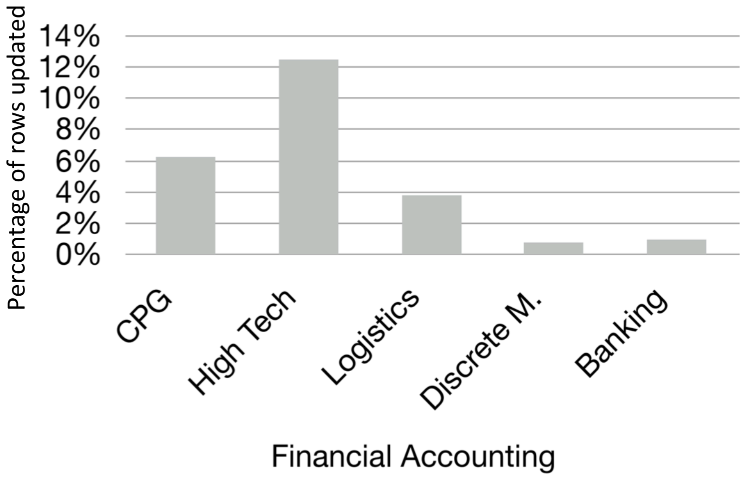
- Workload in enterprise applications constitutes of
  - Mainly read queries (OLTP 83%, OLAP 94%)
  - Many queries access large sets of data





6

Few Updates in OLTP



7

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.

8

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

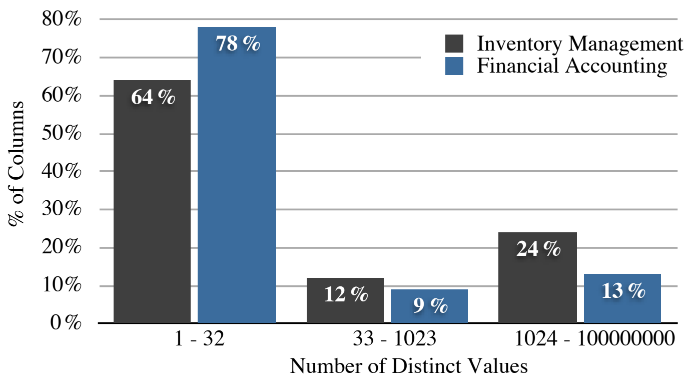
Standard enterprise software data is **sparse and wide.**

9

Many Columns are not Used Even Once

55% unused columns per company in average

40% unused columns across all analyzed companies



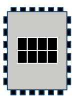
10

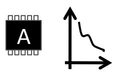
Changes in Hardware

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





- 64 bit address space
- 2TB in current servers
- Cost-performance ratio rapidly declining
- Memory hierarchies

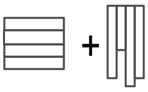
- Main Memory becomes **cheaper and larger**

12

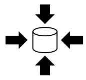
In the Meantime Research has come up with…

… 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

13

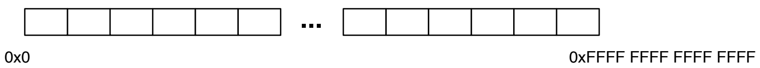
Foundations of Database Storage Techniques

Data Layout in Main Memory

Memory Basics (1)

- Memory in todays 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

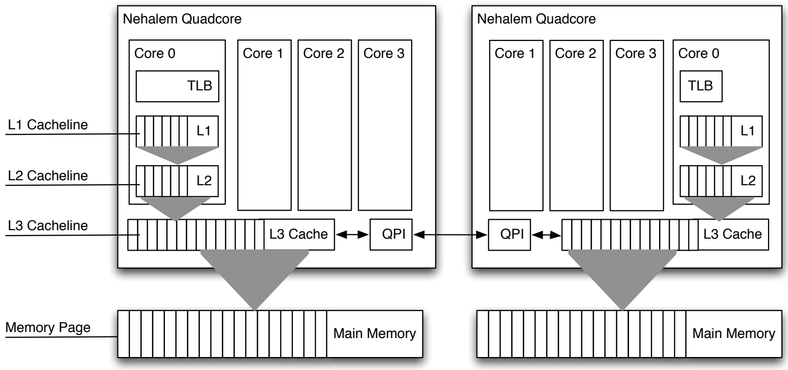
Memory Basics (2)



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

17

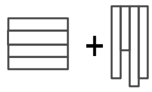
Memory Hierarchy

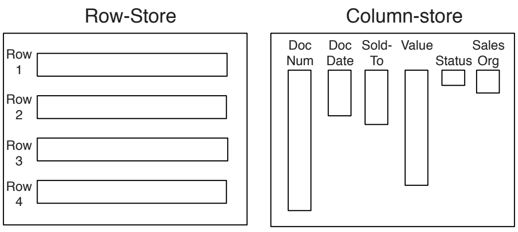


18

Physical Data Representation

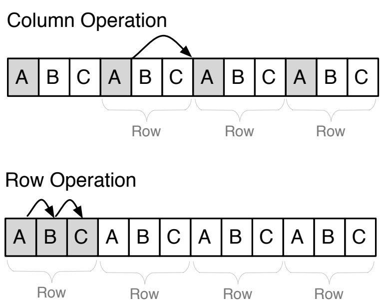
- **Row** store:
  - Rows are stored consecutively
  - Optimal for row-wise access (e.g. SELECT *)
- **Column** store:
  - Columns are stored consecutively
  - Optimal for attribute focused access (e.g. SUM, GROUP BY)
- Note: concept is **independent** from storage type





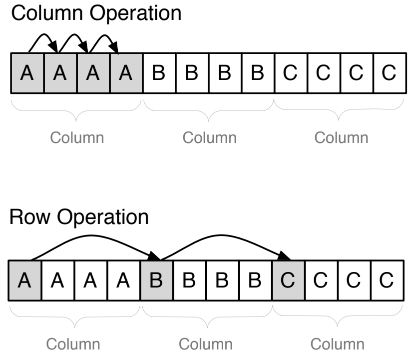
Row Data Layout

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

23

Dictionary Encoding Example

8 billion humans

- Attributes
  - first name
  - last name
  - gender
  - country
  - city
  - birthday
  - → 200 byte
- Each attribute is stored dictionary encoded

24

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
...	...	...	...		...	...

25

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

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

→

Dictionary for "fname"

Value ID	Value
...	...
23	John
24	Mary
25	Jane
26	Peter
...	...

Attribute Vector for "fname"

position	Value ID
...	...
39	23
40	24
41	25
42	23
43	26

26

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)

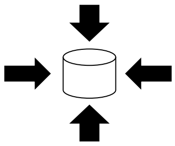
27

Data Size Examples

Column	Cardi-nality	Bits Needed	Item Size	Plain Size	Size with Dictionary (Dictionary + Column)	Compression Factor
First names	5 million	23 bit	50 Byte	373GB	238.4MB + 21.4GB	≈17
Last names	8million	23 bit	50 Byte	373GB	381.5MB + 21.4GB	≈17
Gender	2	1 bit	1 Byte	7GB	2.0b + 953.7MB	≈8
City	1million	20 bit	50 Byte	373GB	47.7MB + 18.6GB	≈20
Country	200	8 bit	47 Byte	350GB	9.2KB + 7.5GB	≈47
Birthday	40000	16 bit	2 Byte	15GB	78.1KB + 14.9GB	≈1
Totals			**200 Byte**	**≈ 1.6TB**	**≈ 92GB**	**≈17**

28

Compression



Compression Techniques

- Heavy weight vs. light weight techniques
- Focus on light weight techniques for databases

- For attribute vector
  - Prefix encoding
  - Run length encoding
  - Cluster encoding
  - Sparse encoding
  - Indirect encoding

- For dictionary
  - Delta compression for strings
  - Other data types are stored as sorted arrays

30

Example Table

recID	fname	lname	gender	country	city	birthday	2nd_nationality
0	Martin	Albrecht	m	GER	Berlin	08-05-1955	n/a
1	Michael	Berg	m	GER	Berlin	03-05-1970	n/a
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968	n/a
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992	US
4	Ulrike	Schulze	f	GER	Potsdam	09-03-1977	n/a
5	Martin	Schulz	m	GER	Mainz	06-04-1980	GER
6	Sushi	Pao	f	CN	Peking	09-12-1954	n/a
7	Chen	Su Wong	m	CN	Shanghai	27-06-1999	n/a
…	…	…	…	…	…	…	…

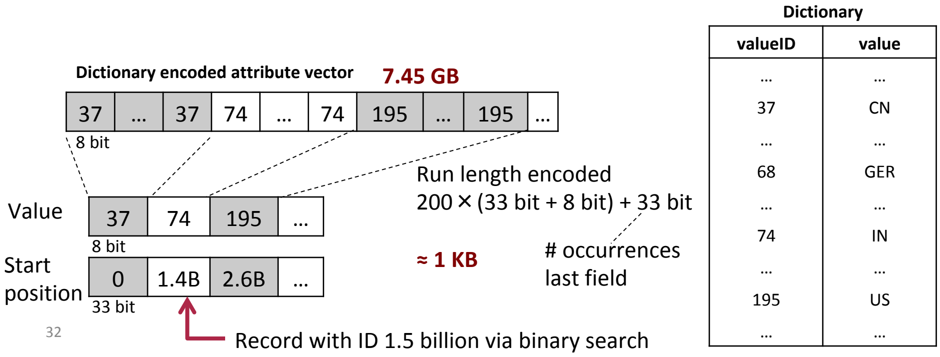
- 200 countries = 8 bit
- 1 million cities = 20 bit
- 100 different 2nd nationalities = 7 bit
- 5 million first names = 23 bit

31

Run Length Encoding

- Replace sequence of the same value with a single instance of the value and
  a) Its number of occurrences
  b) Its start position (shown below) **Direct access!**
- Variant b) speeds up access compared to a)

Example: country column, table sorted by population of country



32

Sparse Encoding

- Remove the value v that appears most often
- A bit vector indicates at which positions v was removed from the original sequence

Example: 2nd nationality column, regardless of sorting order of table

Dictionary

valueID	value
0	n/a
…	…
4	CO
…	…
9	GER
…	…
95	US
…	…

Dictionary encoded attribute vector **6.5 GB**

…	0	0	0	9	0	0	0	0	9	0	0	0	0	4	0	95	0	0	0	…

7 bit

v: 0 (7 bit)

Sparse encoded: … | 9 | 9 | 4 | 95 | … (7 bit)

Bit vector: …1110111101111010111…

Assumption: 99 % of people do not have a 2nd nationality

Bit vector 8 billion × 1 bit 954 MB

Sparse encoded attribute vector

80 mio × 7 bit + 67 MB

value + 7 bit ≈ **1 GB**

No direct access! Compute position via bit vector.

33

Keep in Mind

- Most compression techniques require sorted sets, but a table can only be sorted by one column or cascading
- No direct access to rows in some cases, but offset has to be computed

34

In-Memory Database Operators

Scan Performance (I)

8 billion humans

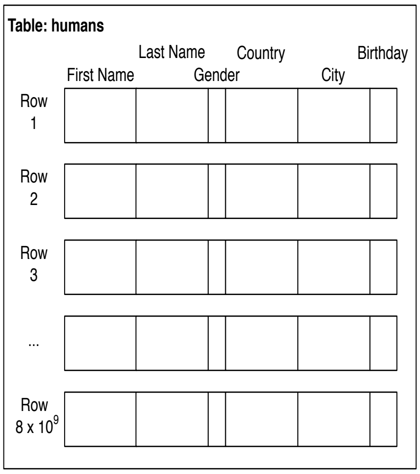
- Attributes
  - First Name
  - Last Name
  - Gender
  - Country
  - City
  - Birthday

  ➔ 200 byte
- Question: How many men/women?
- Assumed scan speed: 2MB/ms/core

36

Scan Performance (2)

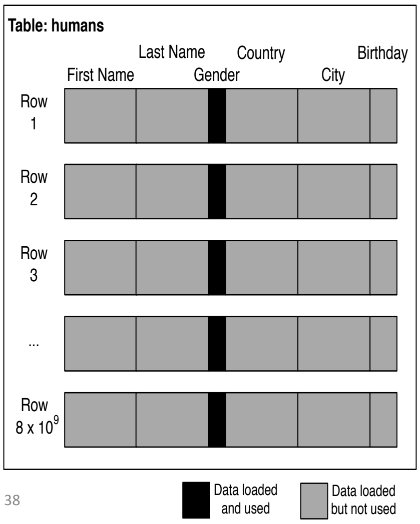
Row Store – Layout



- 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)

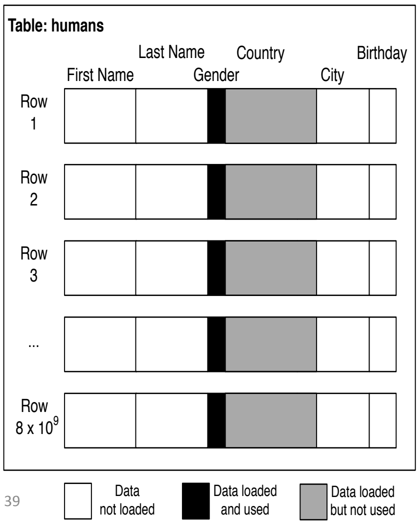
Row Store – Full Table Scan



- Table 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 (4)

Row Store – Stride Access "Gender"

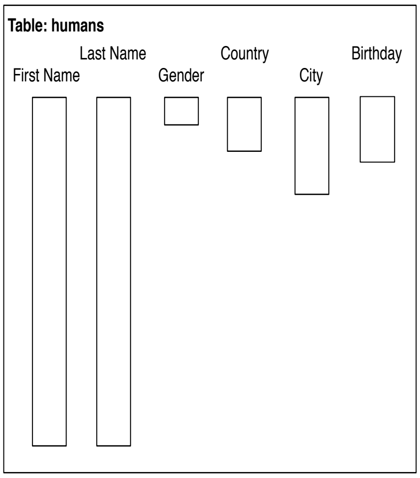


- 8 billion cache accesses à 64 byte → ≈**512 GB**
- Read with 2MB/ms/core → ≈**256 seconds** with 1 core

39

Scan Performance (5)

Column Store – Layout

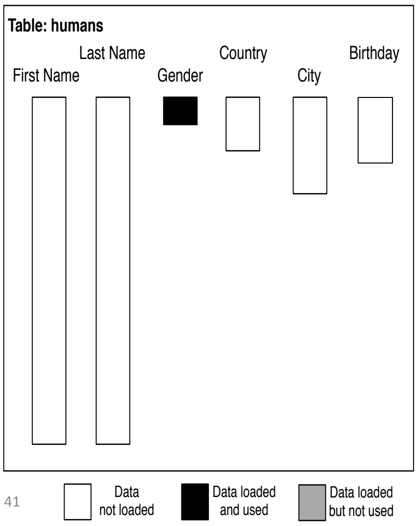


- Table size
  - Attribute vectors: ≈**91 GB**
  - Dictionaries: ≈**700 MB**
  - → Total: ≈**92 GB**
- Compression factor: ≈**17**

40

Scan Performance (6)

Column Store – Full Column Scan on "Gender"

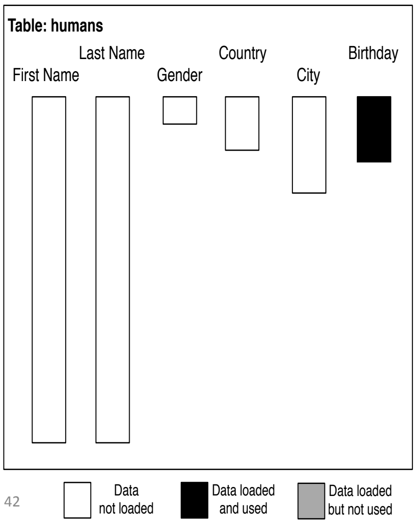


- Size of attribute vector "gender" = 8 billion tuples x 1 bit per tuple → ≈**1 GB**
- Scan through attribute vector with 2MB/ms/core → ≈**0.5 seconds** with 1 core

41

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 2MB/ms/core → ≈**8 seconds** with 1 core

42

Scan Performance – Summary

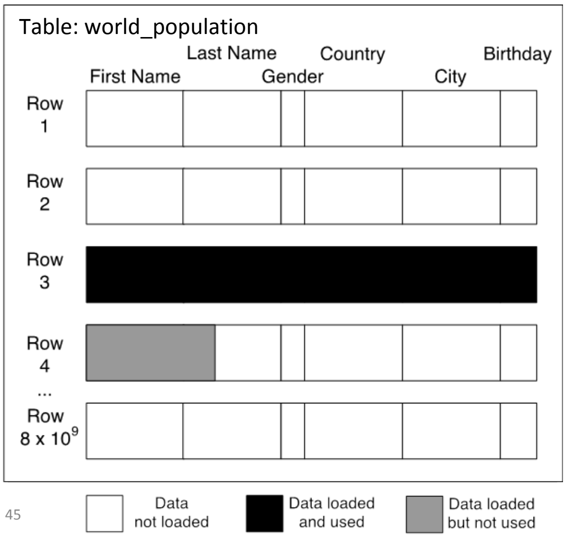
- How many women, how many men?

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

Tuple Reconstruction

Tuple Reconstruction (1)

Accessing a record in a row store

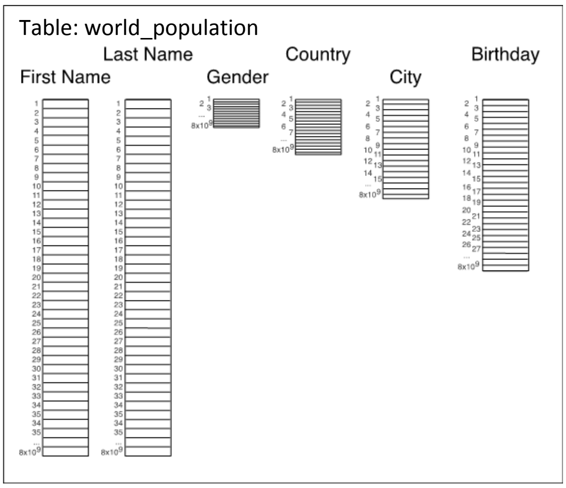


- All attributes are stored consecutively
- 200 byte → 4 cache accesses à 64 byte **→ 256 byte**
- Read with 2MB/ms/core **→ ≈ 0.128 µs** with 1 core

45

Tuple Reconstruction (2)

Virtual record IDs

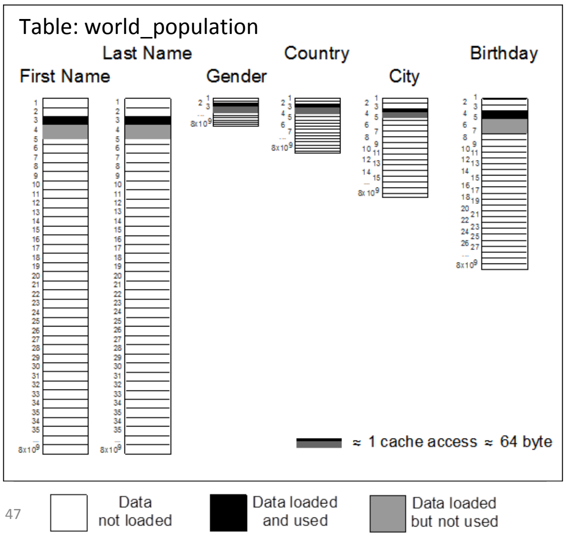


- All attributes are stored in separate columns
- Implicit record IDs are used to reconstruct rows

46

Tuple Reconstruction (3)

Virtual record IDs



- 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

47

Materialization Strategies

Tuple Reconstruction

Strategies:

- ☐ **Early** materialization
  - Decompress and decode data early and operate on strings
  - Create a row-wise data representation early on
- ☐ **Late** materialization
  - Operate on integers as long as possible
  - Operate on columns as long as possible

49

Example

Query:
List the number of Male inhabitants per city in Germany

SELECT city, COUNT(*)
FROM world_population
WHERE gender = “m“
AND country= “GER“
GROUP BY city

Table “world_population”

fname	lname	gender	country	city	birthday
Martin	Albrecht	m	GER	Berlin	08-05-1955
Michael	Berg	m	GER	Berlin	03-05-1970
Hanna	Schulze	f	GER	Bonn	04-04-1968
Ulrich	Schulze	m	GER	Bonn	10-20-1992
…	…	…	…	…	…

Dictionary encoded attribute vectors

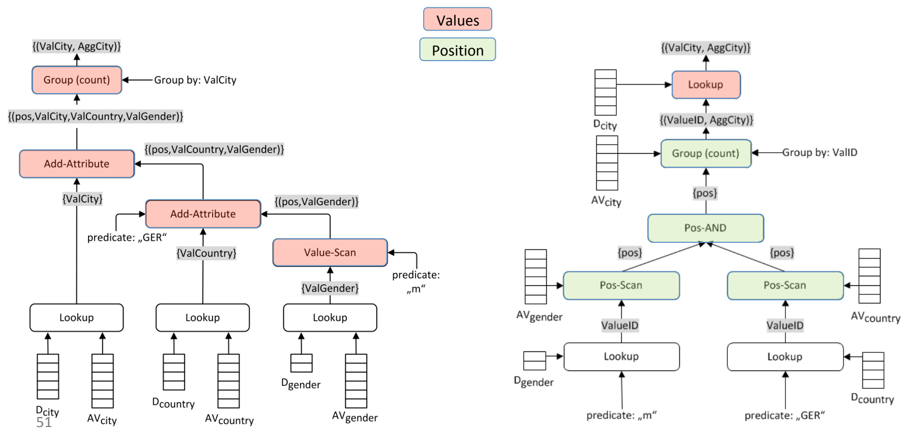
fname	lname	gender	country	city	birthday
53946	10435	0	68	357	15556
54368	25063	0	68	357	20882
30145	99645	1	68	443	20182
99312	99645	0	68	443	29147

50

Comparison

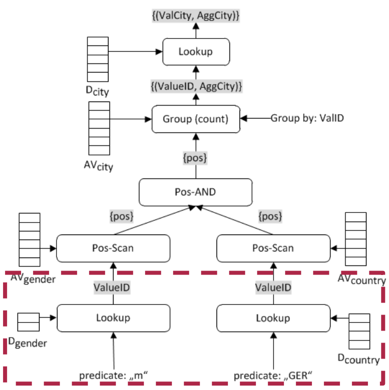
Early Materialization
Early dictionary lookup → working on materialized values

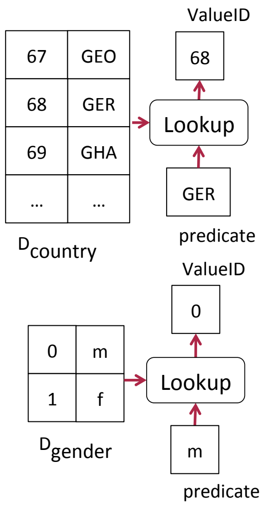
Late Materialization
Late dictionary lookup → working with positional information



Late Materialization (I)

Look up constraint values in dictionary



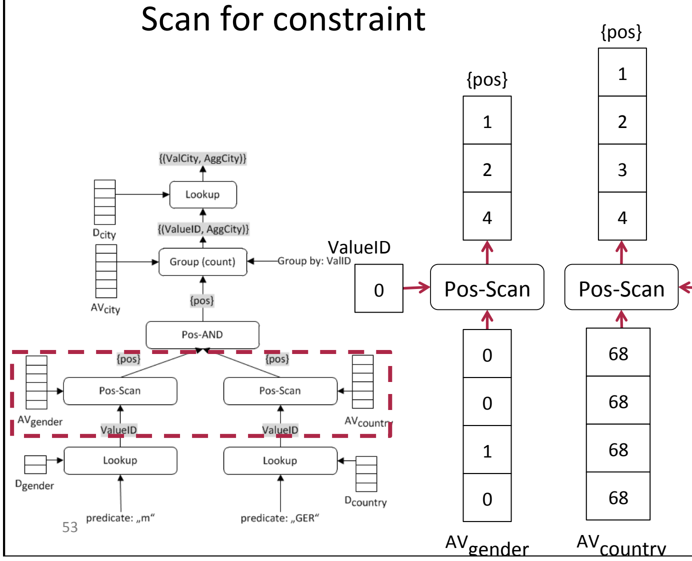


Query:

SELECT city, COUNT(*)
FROM world_population
WHERE gender = "m"
AND country= "GER"
GROUP BY city

Late Materialization (2)

Scan for constraint



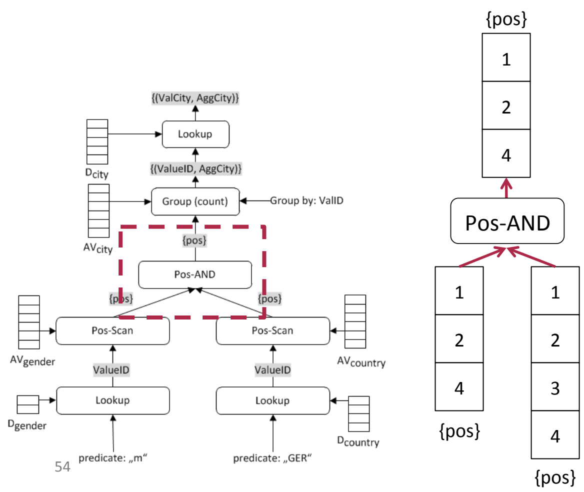
Query:

SELECT city, COUNT(*)
FROM world_population
WHERE gender = “m“
AND country= “GER“
GROUP BY city

53

Late Materialization (3)

Logical AND



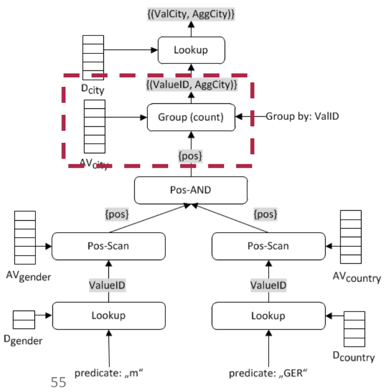
Query:

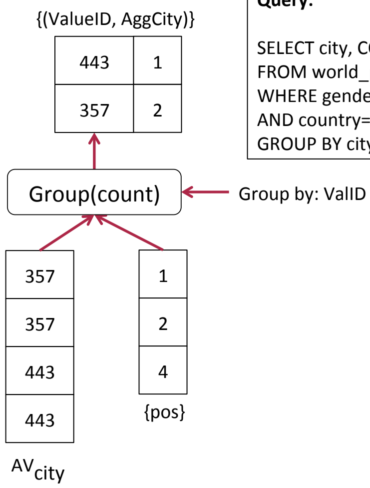
SELECT city, COUNT(*)
FROM world_population
WHERE gender = “m“
AND country= “GER“
GROUP BY city

54

Late Materialization (4)

Filter attribute vector





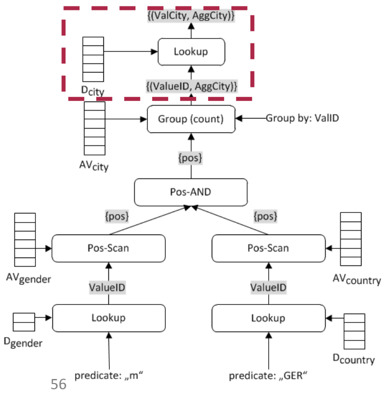
Query:

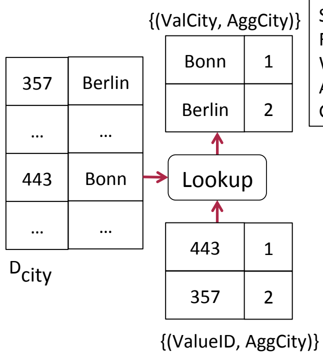
SELECT city, COUNT(*)
FROM world_population
WHERE gender = "m"
AND country= "GER"
GROUP BY city

55

Late Materialization (5)

Lookup values from the dictionary





Query:

SELECT city, COUNT(*)
FROM world_population
WHERE gender = "m"
AND country= "GER"
GROUP BY city

56

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

58

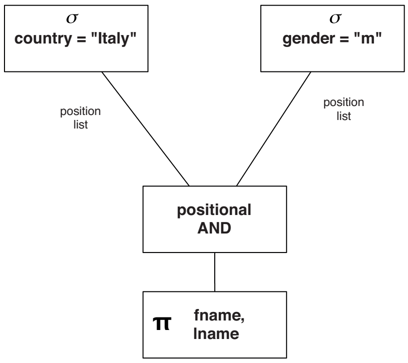
Query Plan

- 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"
```

59

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

60

UNIVERSITAS FRIBURGENSIS

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

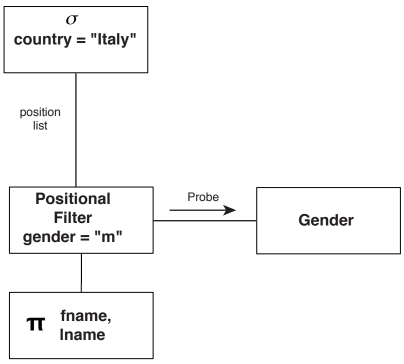
AND

Position
0
2

61

UNIVERSITAS FRIBURGENSIS

Query Plan (ii)



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

62

UNIVERSITAS FRIBURGENSIS

Insert

UNIVERSITAS FRIBURGENSIS

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

64

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)

65

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)

66

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	$\boxed{\text{Schulze}}$

	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

67

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
5	$\boxed{3}$

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
5		Schulze				
…	…	…	…	…	…	…

Attribute Vector (AV)
Dictionary (D)

1. Look-up on D → entry found
2. Append ValueID to AV

68

INSERT (2) with new Dictionary Entry

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

69

INSERT (2) with new Dictionary Entry

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	Karen
3	0	3	Martin
4	4	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

70

INSERT (2) with new Dictionary Entry

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

	AV		D (old)	→	D (new)
0	2	0	Anton	0	Anton
1	3	1	Hanna	1	Hanna
2	1	2	Martin	2	Karen
3	0	3	Michael	3	Martin
4	4	4	Sophie	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

71

INSERT (2) with new Dictionary Entry

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

	AV (old)	→	AV (new)		D (new)
0	2	0	3	0	Anton
1	3	1	4	1	Hanna
2	1	2	1	2	Karen
3	0	3	0	3	Martin
4	4	4	5	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

72

INSERT (2) with new Dictionary Entry

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	$\boxed{2}$	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

73

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)

74

UNIVERSITAS FRIBURGENSIS

Advanced Database Storage Techniques

UNIVERSITAS FRIBURGENSIS

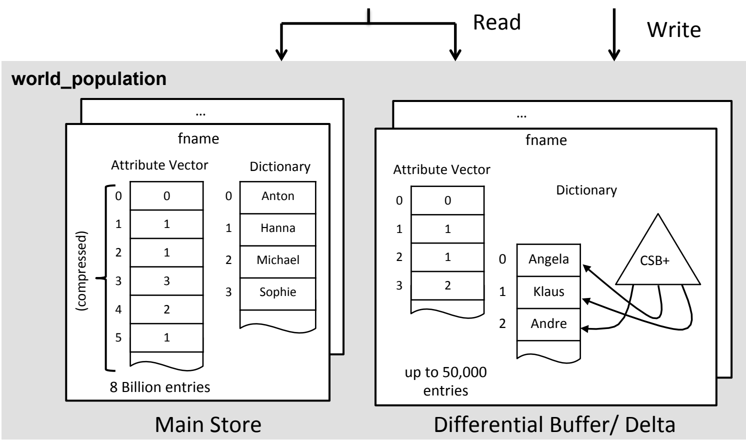
Differential Buffer

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 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

79

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
4	Ulrike	Schulze	f	GER	Potsdam	09-03-1977
5	Sophie	Schulze	f	GER	Rostock	06-20-2012
...	...	...	...	...	...	...
$8 * 10^9$	Zacharias	Perdopolus	m	GRE	Athen	03-12-1979

Main Store

Differential Buffer

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

80

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
4	Ulrike	Schulze	f	GER	Potsdam	09-03-1977
5	Sophie	Schulze	f	GER	Rostock	06-20-2012
...	...	...	...	...	...	...
$8 * 10^9$	Zacharias	Perdopolus	m	GRE	Athen	03-12-1979

Differential Buffer

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

81

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
4	Ulrike	Schulze	f	GER	Potsdam	09-03-1977
5	Sophie	Schulze	f	GER	Rostock	06-20-2012
...	...	...	...	...	...	...
$8 * 10^9$	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"*

82

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

83

Tuple Lifetime

Michael moves from Berlin to Potsdam

Main Table: world_population

recId	fname	lname	gender	country	city	birthday	valid	
0	Martin	Albrecht	m	GER	Berlin	08-05-1955	1	Main Store
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	
4	Ulrike	Schulze	f	GER	Potsdam	09-03-1977	1	
5	Sophie	Schulze	f	GER	Rostock	06-20-2012	1	
...	...	...	...	...	...	...		
$8 * 10^9$	Zacharias	Perdopolus	m	GRE	Athen	03-12-1979	1	
0	**Michael**	**Berg**	**m**	**GER**	**Potsdam**	**03-05-1970**	**1**	Differential Buffer

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

84

UNIVERSITAS FRIBURGENSIS

Tuple Lifetime

Michael moves from Berlin to Potsdam

Main Table: world_population

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
4	Ulrike	Schulze	f	GER	Potsdam	09-03-1977	1
5	Sophie	Schulze	f	GER	Rostock	06-20-2012	1
...	...	...	...	...	...	...	
$8 * 10^9$	Zacharias	Perdopolus	m	GRE	Athen	03-12-1979	1
0	**Michael**	**Berg**	**m**	**GER**	**Potsdam**	**03-05-1970**	**1**

Main Store

Differential Buffer

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

85

UNIVERSITAS FRIBURGENSIS

Merge

Handling Write Operations

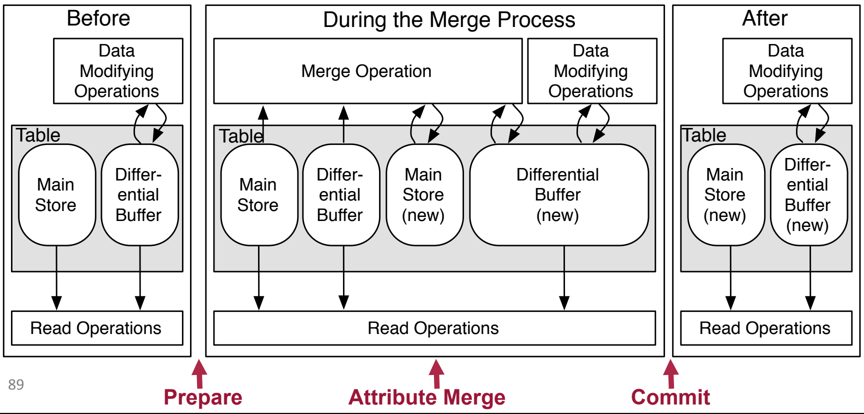
- All Write operations (INSERT, UPDATE) are stored within a differential buffer (delta) first
- Read-operations on differential buffer are more expensive than on main store
- Differential buffer is merged periodically with the main store
  - To avoid performance degradation based on large delta
  - Merge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



89

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

90

Example

Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
London

Attribute Vectors

recID	fname	city
0	2	0
1	1	1
2	0	0

Validity Vector

recID	valid
0	1
1	1
2	1

Delta

Dictionaries

valueID	fname	city

Attribute Vectors

recID	fname	city

Validity Vector

recID	valid

Example

Michael moves from London to Berlin

Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
London

Attribute Vectors

recID	fname	city
0	2	0
1	1	1
2	0	0

Validity Vector

recID	valid
0	1
1	0
2	1

Delta

Dictionaries

valueID	fname	city
0	Michael	Berlin

Attribute Vectors

recID	fname	city
0	0	0

Validity Vector

recID	valid
0	1

UNIVERSITAS FRIBURGENSIS

Example

Nadja moves from Berlin to Potsdam

Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
London

Attribute Vectors

recID	fname	city
0	2	0
1	1	1
2	0	0

Validity Vector

recID	valid
0	0
1	0
2	1

Delta

Dictionaries

valueID	fname	city
0	Michael	Berlin
1	Nadja	Potsdam

Attribute Vectors

recID	fname	city
0	0	0
1	1	1

Validity Vector

recID	valid
0	1
1	1

93

UNIVERSITAS FRIBURGENSIS

Example

Michael moves from Berlin to Potsdam

Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
London

Attribute Vectors

recID	fname	city
0	2	0
1	1	1
2	0	0

Validity Vector

recID	valid
0	0
1	0
2	1

Delta

Dictionaries

valueID	fname	city
0	Michael	Berlin
1	Nadja	Potsdam

Attribute Vectors

recID	fname	city
0	0	0
1	1	1
2	0	1

Validity Vector

recID	valid
0	0
1	1
2	1

94

First Step: Validity Detection

UNIVERSITAS FRIBURGENSIS

Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
London

Attribute Vectors

recID	fname	city
0	2	0
1	1	1
2	0	0

Validity Vector

recID	valid
0	0
1	0
2	1

Delta

Dictionaries

valueID	fname
0	Michael
1	Nadja

city
Berlin
Potsdam

Attribute Vectors

recID	fname	city
0	0	0
1	1	1
2	0	1

Validity Vector

recID	valid
0	0
1	1
2	1

Will be deleted / moved into history partition

95

First Step: Dictionary Merge

UNIVERSITAS FRIBURGENSIS

Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
London

Delta

Dictionaries

valueID	fname
0	Michael
1	Nadja

city
Berlin
Potsdam

New Combined Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
Potsdam

Mappings

fname: old to new Main

valueID (old)	0	1	2
valueID (new)	0	1	2

fname: Delta to Main

valueID (Delta)	0	1
valueID (Main)	1	2

city: old to new Main

valueID (old)	0	1
valueID (new)	0	NA

city: Delta to Main

valueID (Delta)	0	1
valueID (Main)	0	1

Second Step: New Main Column

Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
London

Attribute Vectors

recID	fname	city
0	0	0

Validity Vector

recID	valid
0	1

Delta

Dictionaries

valueID	fname
0	Michael
1	Nadja

city
Berlin
Potsdam

Attribute Vectors

recID	fname	city
0	1	1
1	0	1

Validity Vector

recID	valid
0	1
1	1

Mappings

fname: old to new Main

valueID (old)	0	1	2
valueID (new)	0	1	2

fname: Delta to Main

valueID (Delta)	0	1
valueID (Main)	1	2

city: old to new Main

valueID (old)	0	1
valueID (new)	0	NA

city: Delta to Main

valueID (Delta)	0	1
valueID (Main)	0	1

Second Step: New Main Column

Main Store

Dictionaries

valueID	fname
0	Albert
1	Michael
2	Nadja

city
Berlin
Potsdam

Attribute Vectors

recID	fname	city
0	0	0
1	2	1
2	1	1

Validity Vector

recID	valid
0	1
1	1
2	1

Delta

Dictionaries

valueID	fname	city

Attribute Vectors

recID	fname	city

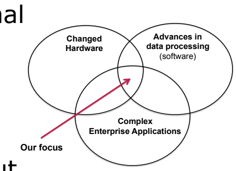
Validity Vector

recID	valid

98

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



99

Want More?

- Master Thesis Topics
  - **Distributed In-Memory Key-Value Store on Ultra-Low-Power Hardware**

Raspberry PI Cluster

  - **Graph Analytics and User Tracking in In-Memory Databases**
    - Data analysis + query execution + compression und huge servers
      - E.g. 32-cores, 412 GB RAM
    - Based on HYRISE: http://github.com/hyrise/hyrise

100

UNIVERSITAS FRIBURGENSIS

UNIVERSITAS FRIBURGENSIS

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

Slides by Martin Grund
Presented today by Alberto Lerner
alberto.lerner@unifr.ch

eXascale Infolab
Department of Informatics
University of Fribourg Switzerland