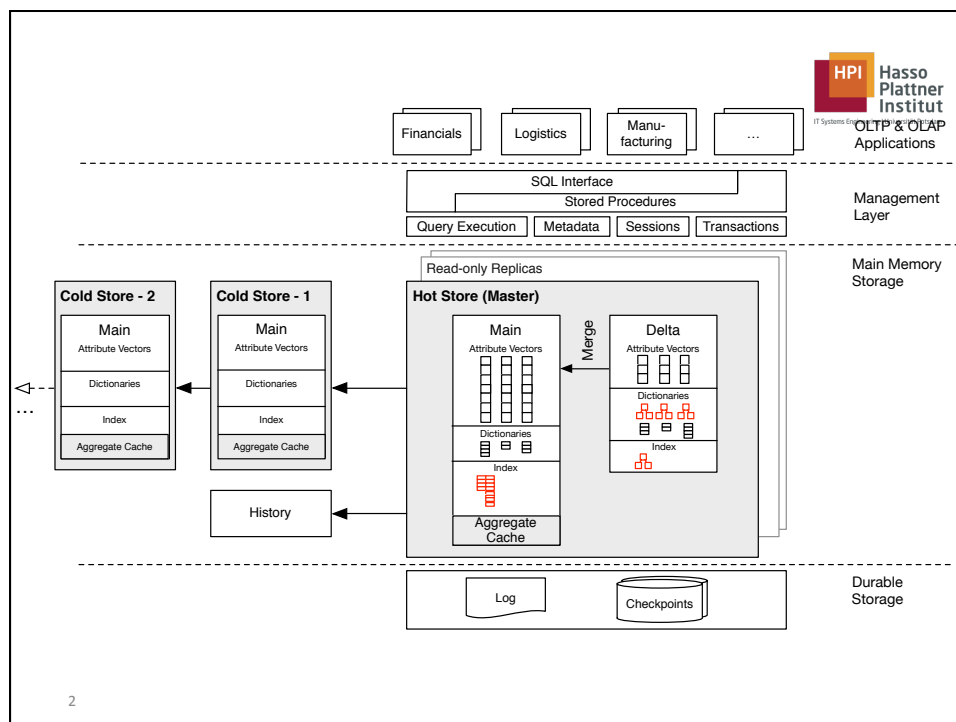


Indices

Prof. Hasso Plattner



Indices: Overview

□ Database Indices

- Clustered Index: The sort order of tuples in the primary storage
 - Primary Key: Enforced via Index on Primary Key Attribute, often also the Clustered Index
- Secondary Index: Additional sort order according to a non-key attribute
- Tree vs. Hash Index: Tree Indices allow for efficient range queries
- Inverted Index: Originated in Text Retrieval, but general concept

□ Application maintained indices

- Tables containing subsets of data
- Application maintains content: e.g. "Open Items" table in ERP

3

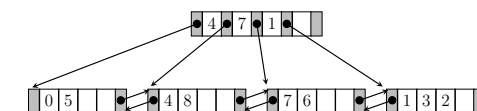
Delta Index

- The delta is small, but needs to perform quick inserts.
- **Problem:** Find Value ID of existing Values
- Main-memory optimized B+Trees,
- Tree contains ValueIDs
- While traversing the tree the respective dictionary entries are compared: Allows to efficiently translate Values to Value IDs
- Allows access in hundreds to thousands of CPU cycles
- Can be extended with positions(rowIDs) in attribute vector (inverted index)

Attribute Vector (AV) Unsorted Dictionary (UD)

0	1	San Francisco	0	Bakersfield
1	7	Palo Alto	1	San Francisco
2	2	Santa Clara	2	Santa Clara
3	3	San Jose	3	San Jose
4	1	San Francisco	4	Fresno
5	0	Bakersfield	5	Berkeley
6	8	Oakland	6	Sacramento
7	2	Santa Clara	7	Palo Alto
...	...	...	8	Oakland

Delta Storage Index



All Dictionary Entries in Sort Order

4

Main Index

Recap:

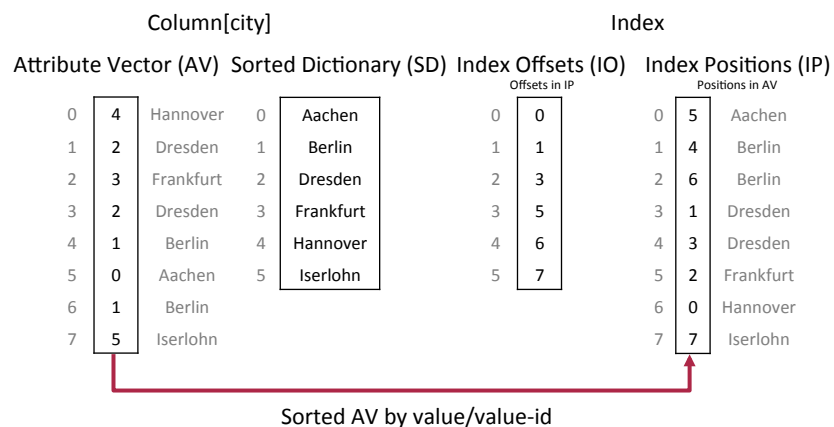
- The main partition is read-only during regular processing
- Index on the main partition does not need to handle updates
- Column is a mapping of (implicit) positions to values.
- Inverted Index maps Values to positions

Example for Lookup of "Berlin":

0	4	Hannover	0	Aachen	0	0	0	5	Aachen
1	2	Dresden	1	Berlin	1	1	1	4	Berlin
2	3	Frankfurt	2	Dresden	2	3	2	6	Berlin
3	2	Dresden	3	Frankfurt	3	5	3	1	Dresden
4	1	Berlin	4	Hannover	4	6	4	3	Dresden
5	0	Aachen	5	Iserlohn	5	7	5	2	Frankfurt
6	1	Berlin					6	0	Hannover
7	5	Iserlohn					7	7	Iserlohn

5

Improving SELECT Performance



6

Improving SELECT Performance

QUERY: SELECT * FROM table WHERE city = "Berlin"

Column[city]			Index		
Attribute Vector (AV)	Sorted Dictionary (SD)		Index Offsets (IO)	Index Positions (IP)	
			Offsets in IP	Positions in AV	
0	4	Hannover	0	0	0
1	2	Dresden	1	1	1
2	3	Frankfurt	2	3	2
3	2	Dresden	3	5	3
4	1	Berlin	4	6	4
5	0	Aachen	5	7	5
6	1	Berlin			6
7	5	Iserlohn			7

7

Improving SELECT Performance

QUERY: SELECT * FROM table WHERE city = "Berlin"

Column[city]			Index		
Attribute Vector (AV)	Sorted Dictionary (SD)		Index Offsets (IO)	Index Positions (IP)	
			Offsets in IP	Positions in AV	
0	4	Hannover	0	0	0
1	2	Dresden	1	1	1
2	3	Frankfurt	2	3	2
3	2	Dresden	3	5	3
4	1	Berlin	4	6	4
5	0	Aachen	5	7	5
6	1	Berlin			6
7	5	Iserlohn			7

□ Find value in D (search in sorted array) → rowID(IO) = [1]

8

Improving SELECT Performance

QUERY: SELECT * FROM table WHERE city = "Berlin"

Column[city]			Index		
Attribute Vector (AV)	Sorted Dictionary (SD)		Index Offsets (IO)	Offsets in IP	Index Positions (IP)
0 4 Hannover	0 Aachen		0 0	0 0	0 5 Aachen
1 2 Dresden	1 Berlin	→	1 1	1 1	1 4 Berlin
2 3 Frankfurt	2 Dresden		2 3	2 3	2 6 Berlin
3 2 Dresden	3 Frankfurt		3 5	3 5	3 1 Dresden
4 1 Berlin	4 Hannover		4 6	4 6	4 3 Dresden
5 0 Aachen	5 Iserlohn		5 7	5 7	5 2 Frankfurt
6 1 Berlin					6 0 Hannover
7 5 Iserlohn					7 7 Iserlohn

- Find offset in IO (direct access via rowID): 1

9

Improving SELECT Performance

QUERY: SELECT * FROM table WHERE city = "Berlin"

Column[city]			Index		
Attribute Vector (AV)	Sorted Dictionary (SD)		Index Offsets (IO)	Offsets in IP	Index Positions (IP)
0 4 Hannover	0 Aachen		0 0	0 0	0 5 Aachen
1 2 Dresden	1 Berlin		1 1	1 1	1 4 Berlin
2 3 Frankfurt	2 Dresden		2 3	2 3	2 6 Berlin
3 2 Dresden	3 Frankfurt		3 5	3 5	3 1 Dresden
4 1 Berlin	4 Hannover		4 6	4 6	4 3 Dresden
5 0 Aachen	5 Iserlohn		5 7	5 7	5 2 Frankfurt
6 1 Berlin					6 0 Hannover
7 5 Iserlohn					7 7 Iserlohn

- Find offset in IO (direct access via rowID): 1
- Use offset of next higher value to determine search interval in IP
→ interval = [1,3)

10

Improving SELECT Performance

QUERY: SELECT * FROM table WHERE city = "Berlin"

Column[city]			Index		
Attribute Vector (AV)	Sorted Dictionary (SD)		Index Offsets (IO)	Index Positions (IP)	
0 4 Hannover	0 Aachen	0 0	0 0	0 5 Aachen	
1 2 Dresden	1 Berlin	1 1	1 1	1 4 Berlin	
2 3 Frankfurt	2 Dresden	2 3	2 3	2 6 Berlin	
3 2 Dresden	3 Frankfurt	3 5	3 5	3 1 Dresden	
4 1 Berlin	4 Hannover	4 6	4 6	4 3 Dresden	
5 0 Aachen	5 Iserlohn	5 7	5 7	5 2 Frankfurt	
6 1 Berlin				6 0 Hannover	
7 5 Iserlohn				7 7 Iserlohn	

□ Scan all values ≥ 1 and < 3 to determine rowIDs of AV $\rightarrow$ rowID(AV) = (4,6)

Improving SELECT Performance

QUERY: SELECT * FROM table WHERE city = "Berlin"

Column[city]			Index		
Attribute Vector (AV)	Sorted Dictionary (SD)		Index Offsets (IO)	Index Positions (IP)	
0 4 Hannover	0 Aachen	0 0	0 0	0 5 Aachen	
1 2 Dresden	1 Berlin	1 1	1 1	1 4 Berlin	
2 3 Frankfurt	2 Dresden	2 3	2 3	2 6 Berlin	
3 2 Dresden	3 Frankfurt	3 5	3 5	3 1 Dresden	
4 1 Berlin	4 Hannover	4 6	4 6	4 3 Dresden	
5 0 Aachen	5 Iserlohn	5 7	5 7	5 2 Frankfurt	
6 1 Berlin				6 0 Hannover	
7 5 Iserlohn				7 7 Iserlohn	

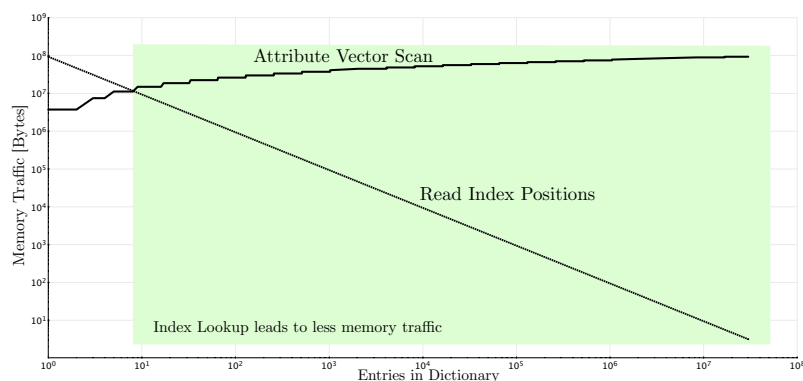
Result: AV positions (4,6)

When to use Main Index: A Theoretical Approach

- **Assumption:** Memory transfer as cost measure
 - **Disclaimer:** Real world is more complicated: cache/instruction misses, order of intermediate result sets, result caching ...
- **Single Lookups:** For single tuples retrieval through index is always faster than a column scan.
- **Ranges:** When is the index access more expensive than the column scan?
 - Depends on size of the selected range
 - Depends on distinct values: Column is compressed with $\log_2(\text{distinct values})$ bits, while index positions are stored with $\log_2(\text{tablesize})$ bits
 - Simplified: If $\text{selected_tuple} * \log_2(\text{tablesize}) < \text{tablesize} * \log_2(\text{distinct values})$
Inverted Index leads to less memory traffic

13

Memory Traffic Comparison



Theoretical Memory Traffic Comparison for a single dictionary selection
Attribute Vector Scan vs. Index Position List Read
for the Main Part of a Column with 30 Million Entries and a varying Distinct Value Count
(Note the Log-Log Scale)

**Index not viable for columns with few distinct values.
Best if predicate is a "strong filter".**

14

Data Access Matrix

Workload	OLTP Transactional	OLXP Mixed	OLAP Analytical
Typical Query	Get Salesorder with ID=X	- Get all open Items of one Customer - Calculate Balance of Account - Calculate discounts/promotions for customer on phone	Aggregate all Sales of the last year by product group and region
Speed Requirements	Microseconds to few Milliseconds (Thousands of CPU Cycles)	Tens to hundreds of Milliseconds	Sub-second "Human Think Time"
Parallelization	Many concurrent queries expected: Inter-query-parallelism	Normally inter-query, sometimes intra-query to meet response time goals	Intra-query-parallelism necessary, e.g. use more cores to meet response time goals.
Access Path	Primary Key Access via an Index for maximum throughput.	"It depends" Highly selective predicates served with help of Index. If large percentage of tuples qualify: Full Column Scans	No index, all via Full column scans (FCS)