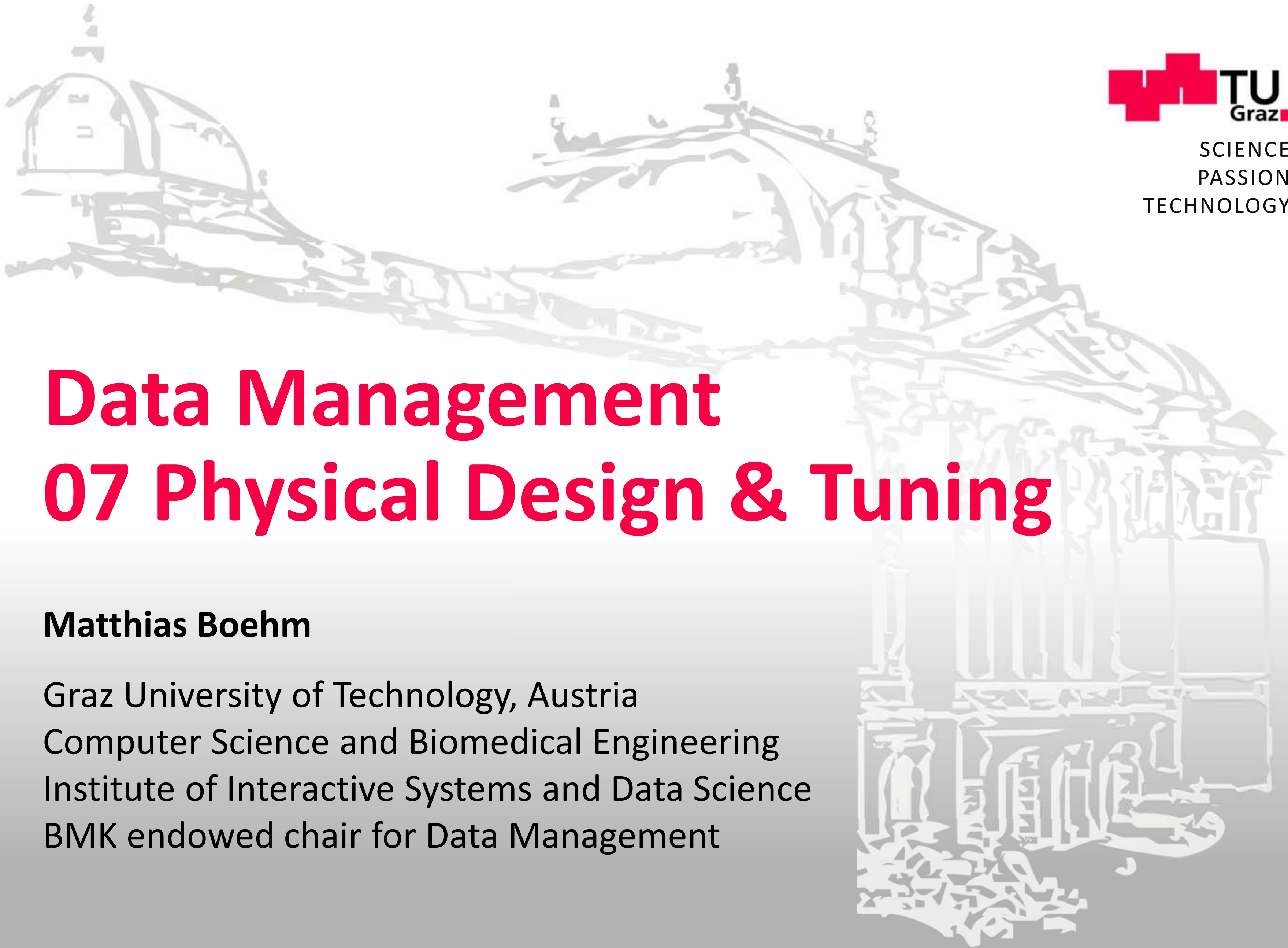


SCIENCE
PASSION
TECHNOLOGY

Data Management
07 Physical Design & Tuning

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMK endowed chair for Data Management

Last update: Nov 23, 2020

Announcements/Org

- **#1 Video Recording**
  - Link in TeachCenter & TUbe (lectures will be public)
  - Due to second lockdown: webex recording

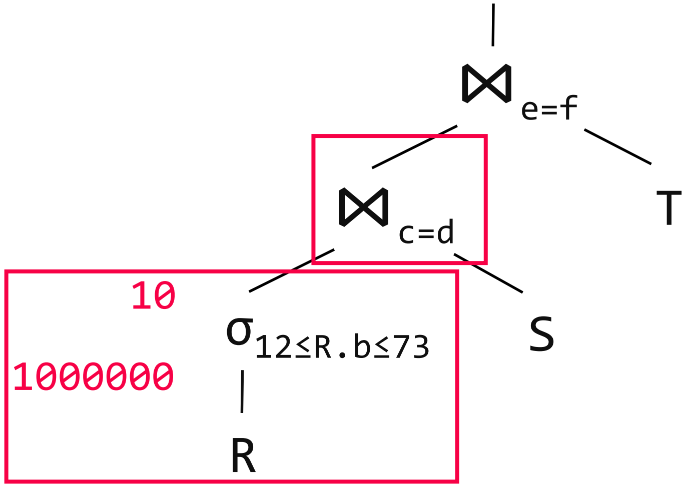
- **#2 Exercise Submissions**
  - **Exercise 1:** grading by Dec 01
  - **Exercise 2:** deadline Dec 01, TC submission open
  - Issues dataset (Nov 19) / expected results (Nov 16)
    → two approaches of handling actors

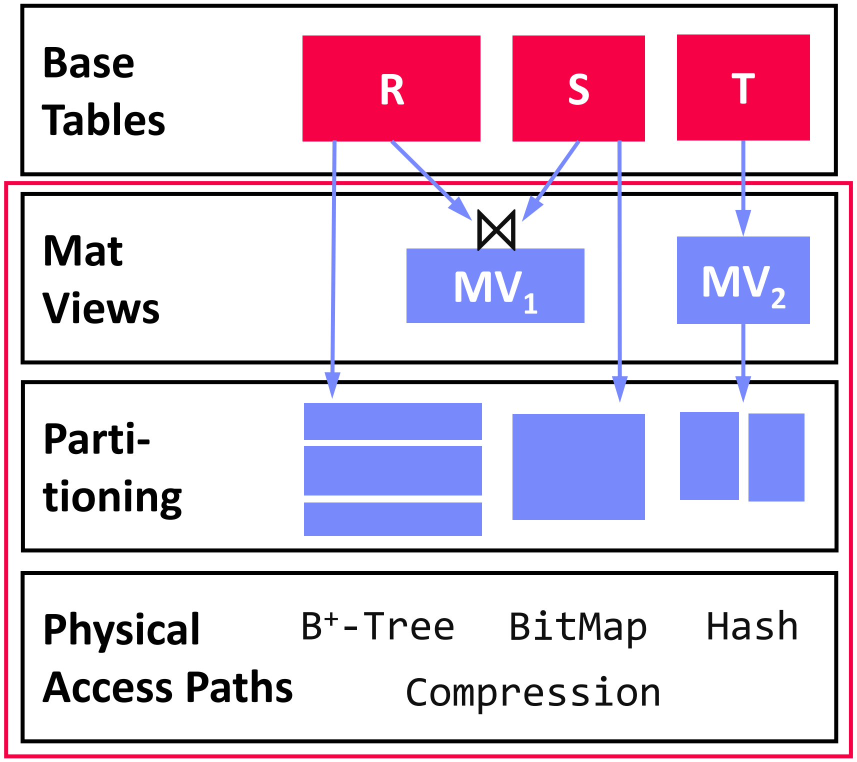
Physical Design, and why should I care?

- **Performance Tuning via Physical Design**
  - Select physical data structures for relational schema and query workload
  - **#1:** User-level, manual physical design by DBA (database administrator)
  - **#2:** User/system-level automatic physical design via advisor tools

- **Example**

```
SELECT * FROM R, S, T
  WHERE R.c = S.d AND S.e = T.f
    AND R.b BETWEEN 12 AND 73
```





Base Tables

R S T

Mat Views

$\bowtie$ MV_1 MV_2

Partitioning

Physical Access Paths

B+-Tree BitMap Hash

Compression

Agenda

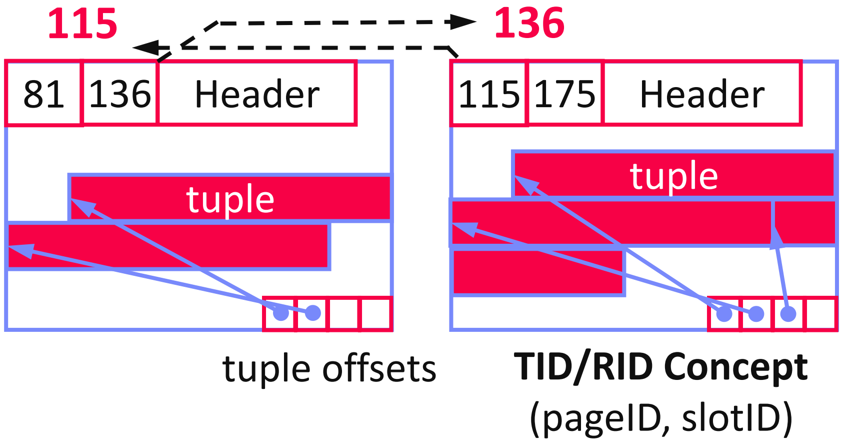
- **Compression Techniques**
- **Index Structures**
- **Table Partitioning**
- **Materialized Views**

} ➡ More details in **706.543 ADBS**

Compression Techniques

Overview Database Compression

- **Background: Storage System**
  - Buffer and storage management (incl. I/O) at granularity of **pages**
  - PostgreSQL default: **8KB**
  - Different table/page layouts (e.g., NSM, DSM, PAX, column)



TID/RID Concept (pageID, slotID)

- **Compression Overview**
  - **Fit larger datasets in memory**, less I/O, better cache utilization
  - Some allow query processing directly **on the compressed data**
  - **#1** Page-level compression (general-purpose GZIP, Snappy, LZ4)
  - **#2** Row-level heavyweight/lightweight compression
  - **#3 Column-level lightweight compression**
  - **#4** Specialized log and index compression

[Patrick Damme et al: Lightweight Data Compression Algorithms: An Experimental Survey. **EDBT 2017**]

Lightweight Database Compression Schemes

- **Null Suppression**
  - Compress integers by **omitting leading zero** bytes/bits (e.g., NS, gamma)

```
                                   106
00000000 00000000 00000000 01101010
                        11 01101010
```

- **Run-Length Encoding**
  - Compress sequences of equal values by **runs** of (value, start, run length)

```
1 1 1 1 7 7 7 7 7 3 3 3 3 3 3 ...
1,1,4   7,5,5     3,10,6
```

- **Dictionary Encoding**
  - Compress column w/ few distinct values as **pos in dictionary** (→ code size)

```
1 7 7 3 1 7 1 3 3 7 1 3 3 7 3 ...
1,3,7  dictionary (code size 2 bit)
1 3 3 2 1 3 1 2 2 3 1 2 2 3 2 ...
```

- **Delta Encoding**
  - Compress sequence w/ small changes by storing **deltas to previous value**

```
20 21 22 20 19 18 19 20 21 20 ...
 0  1  1 -2 -1 -1  1  1  1 -1...
```

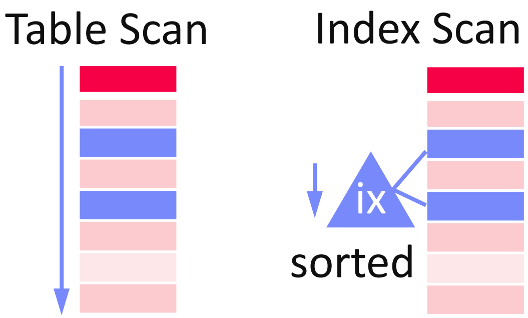
- **Frame-of-Reference Encoding**
  - Compress values by storing **delta to reference value** (outlier handling)

```
20 21 22 20 71 70 71 69 70 21 ...
21          70             22
-1  0  1 -1  1  0  1 -1  0 -1 ...
```

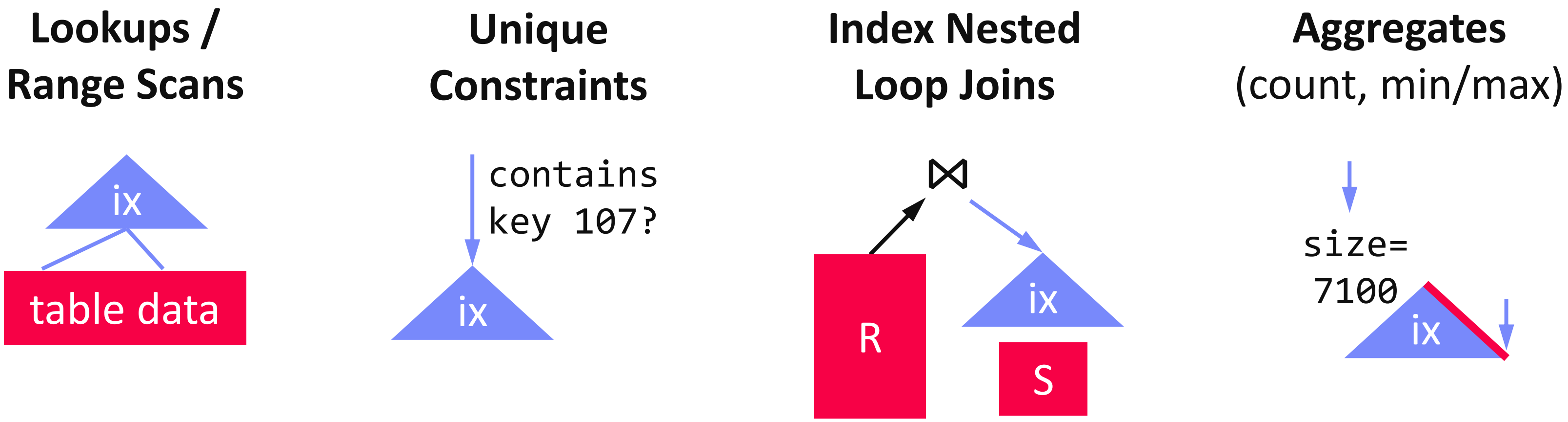
Index Structures

Overview Index Structures

- **Table Scan vs Index Scan**
  - For highly selective predicates, index scan **asymptotically much better** than table scan
  - Index scan **higher per tuple overhead** (break even ~5% output ratio)
  - Multi-column predicates: fetch/RID-list intersection



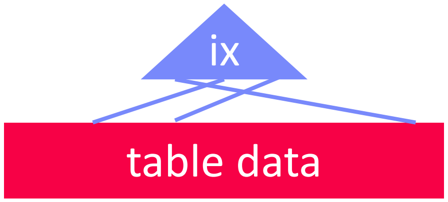
- **Use Cases for Indexes**



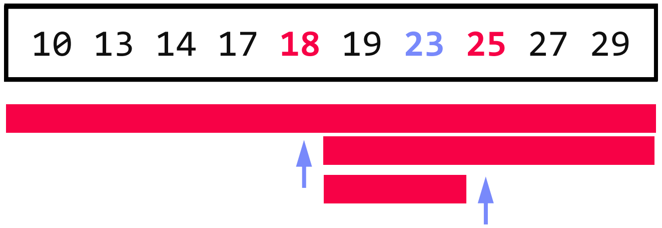
Additional Terminology

- **Create Index**
  - Create a secondary (nonclustered) index on a set of attributes
  - **Clustered:** tuples sorted by index
  - **Non-clustered:** sorted attribute with tuple references
  - Can specify uniqueness, order, and indexing method
  - **PostgreSQL methods:** btree, hash, gist, and gin

```
CREATE INDEX ixStudLname
  ON Students USING btree
  (Lname ASC NULLS FIRST);

DROP INDEX ixStudLname;
```



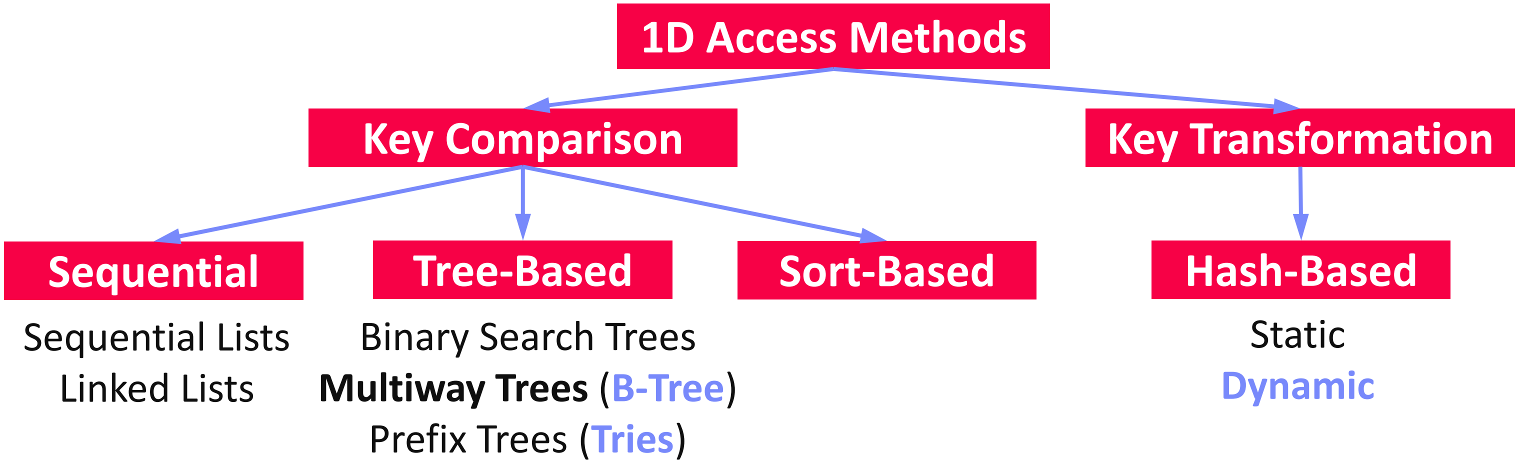
- **Binary Search**
  - `pos = binarySearch(data,key=23)`
  - Given **sorted data**, find key position (insert position if non-existing)
  - **k-ary search** for SIMD data-parallelism
  - **Interpolation search:** probe expected pos in key range (e.g., `search([1:10000], 9700)`)



Classification of Index Structures

- **1D Access Methods**

[Theo Härder, Erhard Rahm: Datenbanksysteme: Konzepte und Techniken der Implementierung, **2001**]



1D Access Methods

Key Comparison

Key Transformation

Sequential

Sequential Lists
Linked Lists

Tree-Based

Binary Search Trees
Multiway Trees (**B-Tree**)
Prefix Trees (**Tries**)

Sort-Based

Hash-Based

Static
Dynamic

- **ND Access Methods**
  - Linearization of ND key space + 1D indexing (Z order, Gray code, Hilbert curve)
  - Multi-dimensional trees and hashing (e.g., UB tree, k-d tree, gridfile)
  - Spatial index structures (e.g., R tree)

B-Tree Overview

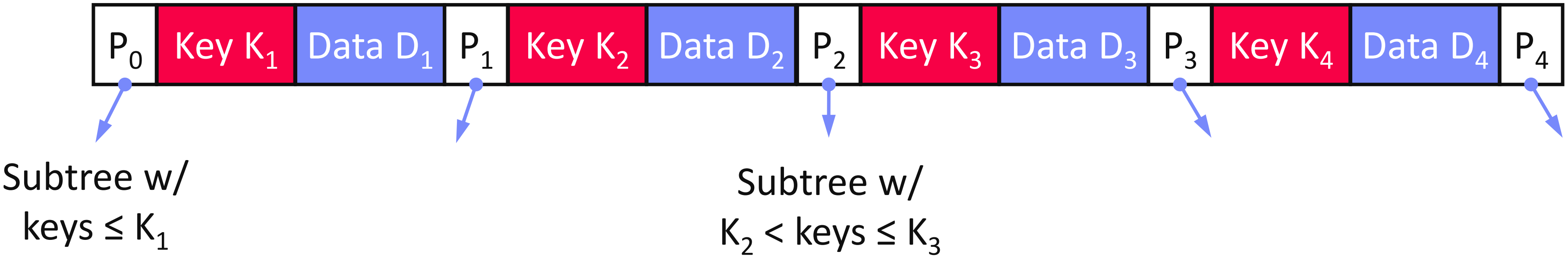
[Rudolf Bayer, Edward M. McCreight:
Organization and Maintenance of Large
Ordered Indices. **Acta Inf. (1) 1972**]

- **History B-Tree**
  - **B**ayer and McCreight 1972, **B**lock-based, **B**alanced, **B**oeing Labs
  - **Multiway tree** (node size = page size); designed for DBMS
  - Extensions: **B+-Tree/B*-Tree** (data only in leafs, double-linked leaf nodes)
- **Definition B-Tree (k, h)**
  - All paths from root to leafs have equal length h
  - All nodes (except root) have **[k, 2k]** key entries
  - All nodes (except root, leafs) have **[k+1, 2k+1]** successors
  - Data is a record or a reference to the record (RID)

$$\lceil \log_{2k+1}(n+1) \rceil \le h \le \left\lceil \log_{k+1}\left(\frac{n+1}{2}\right) \right\rceil + 1$$

All nodes adhere
to max constraints

k=2

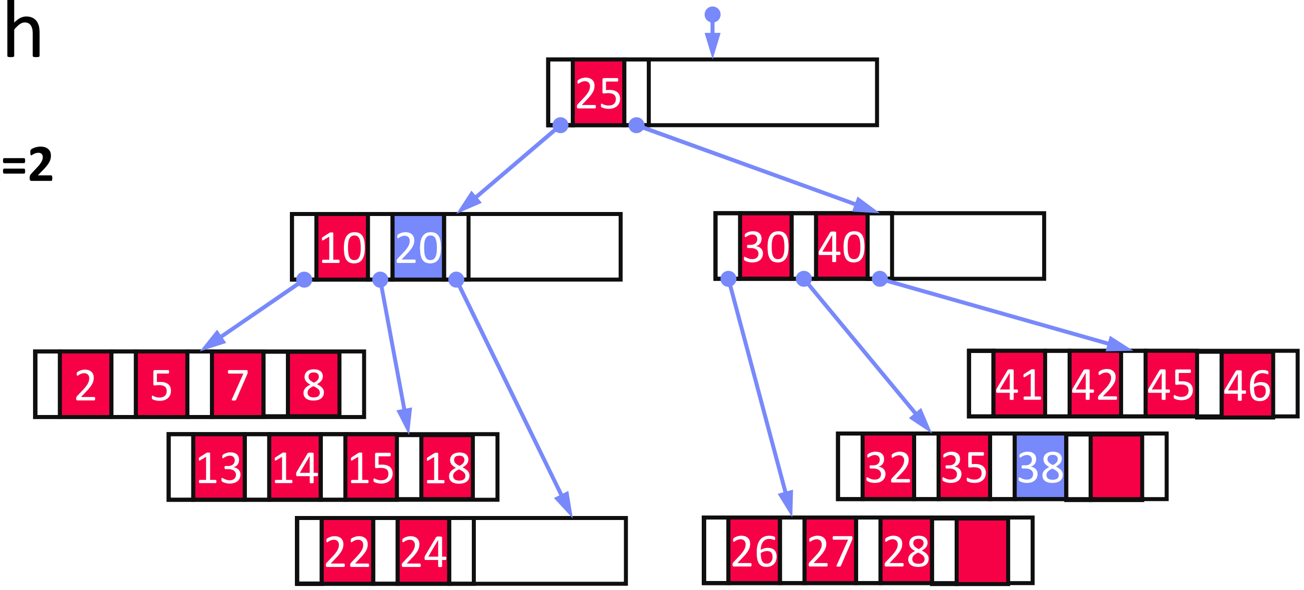


P_0 | Key K_1 | Data D_1 | P_1 | Key K_2 | Data D_2 | P_2 | Key K_3 | Data D_3 | P_3 | Key K_4 | Data D_4 | P_4

Subtree w/
keys ≤ K_1

Subtree w/
K_2 < keys ≤ K_3

B-Tree Search



- **Example B-Tree k=2**
  - Get 38 → D38
  - Get 20 → D20
  - Get 6 → NULL

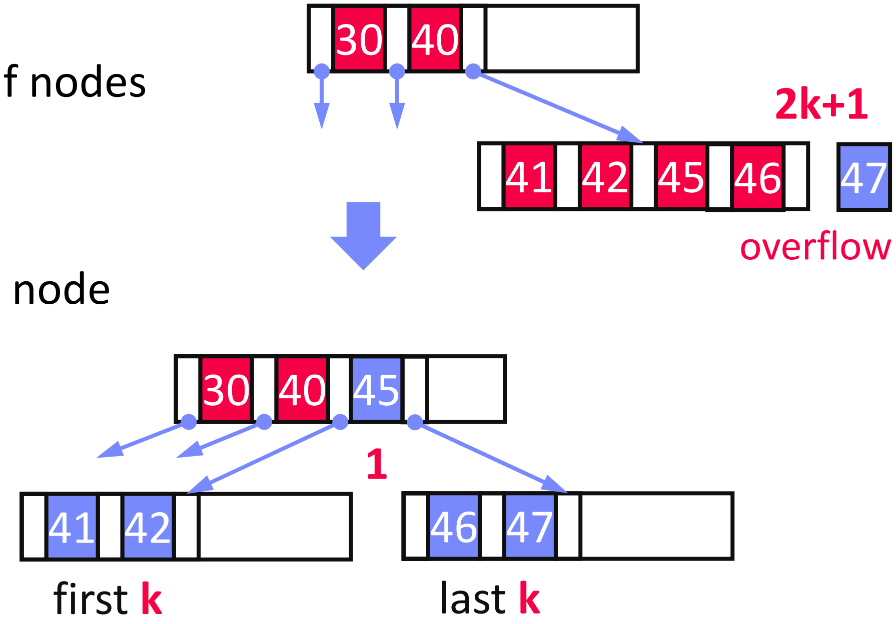
- **Lookup Q_K within a node**
  - Scan / binary search keys for Q_K, if $K_i = Q_K$, return D_i
  - If node does not contain key
    - If leaf node, abort search w/ NULL (not found), otherwise
    - Decent into subtree Pi with $K_i < Q_K \leq K_{i+1}$

- **Range Scan $Q_{L<K<U}$**
  - Lookup Q_L and call next K while $K<Q_U$ (keep current position and node stack)

B-Tree Insert

- **Basic Insertion Approach**
  - **Always insert into leaf nodes!**
  - Find position similar to lookup, insert and maintain sorted order
  - If node overflows (exceeds 2k entries) ➔ **node splitting**

- **Node Splitting Approach**
  - Split the **2k+1** entries into two leaf nodes
  - **Left node:** first k entries
  - **Right node:** last k entries
  - (k+1)th entry inserted into parent node
    ➔ can cause **recursive splitting**
  - Special case: root split (h++)



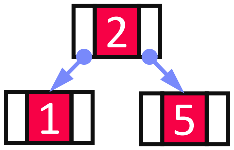
- **B-Tree is self-balancing**

B-Tree Insert, cont. (Example w/ k=1)

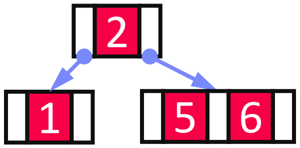
- **Insert 1**

- **Insert 5**

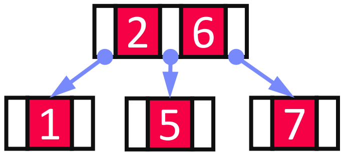
- **Insert 2 (split)**



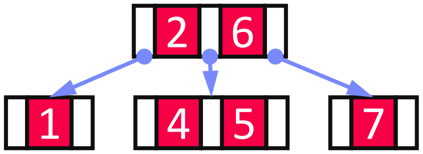
- **Insert 6**



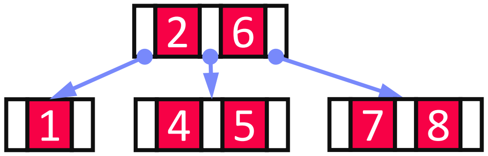
- **Insert 7 (split)**



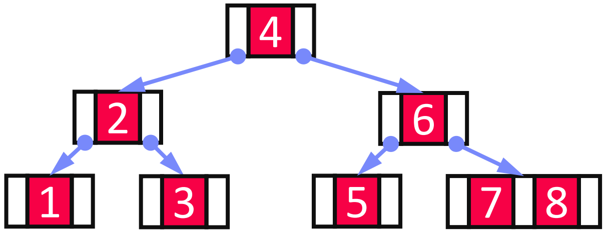
- **Insert 4**



- **Insert 8**



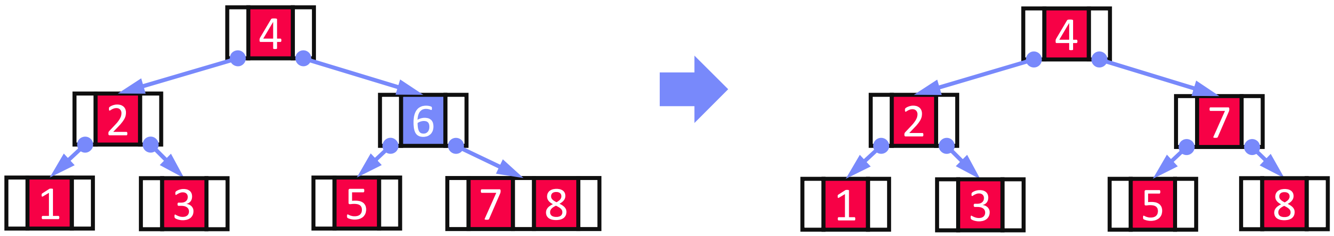
- **Insert 3 (2x split)**



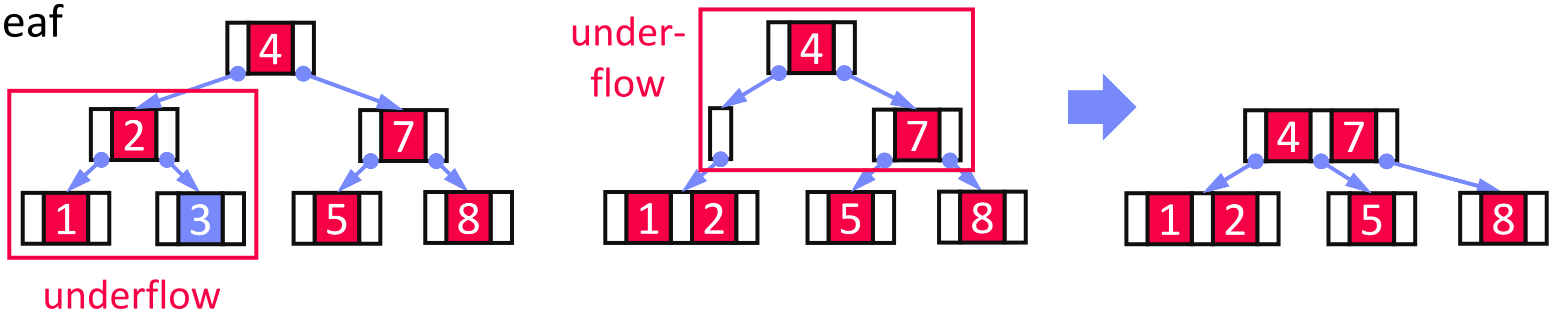
- **Note: Exercise 03?**
  (B-tree insertion and deletion)

B-Tree Delete

- **Basic Deletion Approach**
  - Lookup deletion key, abort if non-existing
  - Case inner node: **move entry** from fullest successor node into position
  - Case leaf node: if underflows (<k entries) ➔ **merge w/ sibling**
- **Example**
  - Case inner

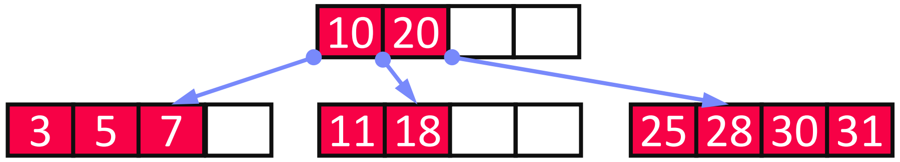


  - Case leaf

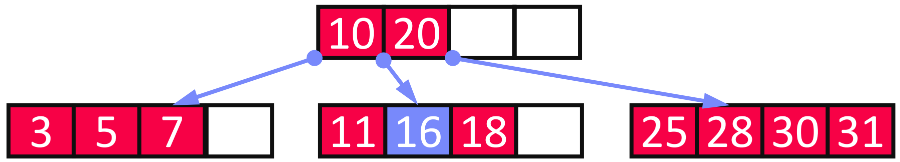


B-Tree Insert and Delete w/ k=2

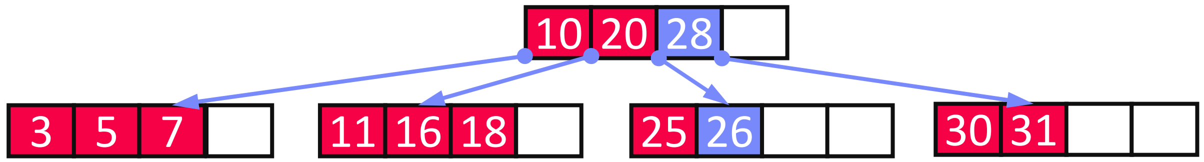
- **Insert/Delete Examples**
  - Original



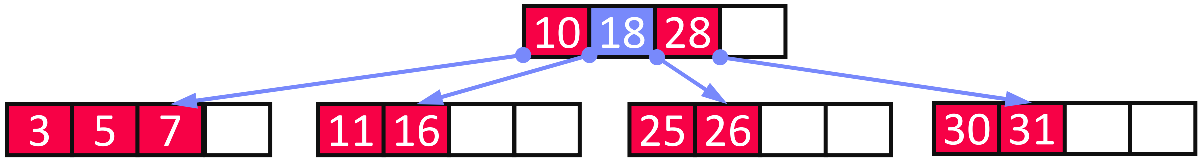
  - Insert 16



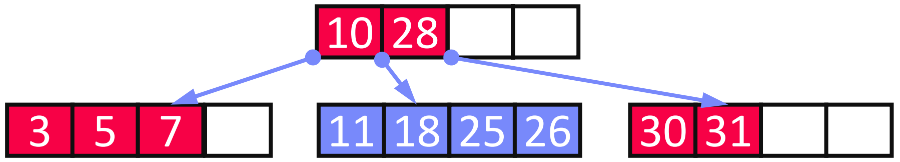
  - Insert 26



  - Delete 20



  - Delete 16



Excursus: Prefix Trees (Radix Trees, Tries)

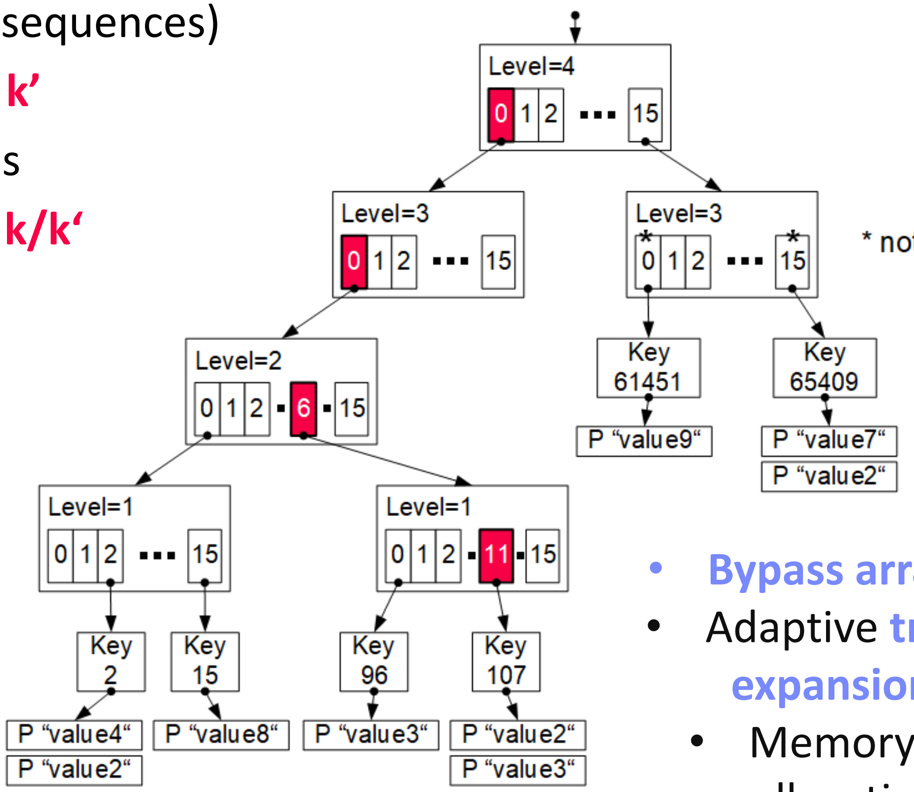
- **Generalized Prefix Tree**
  - **Arbitrary data types** (byte sequences)
  - **Configurable prefix length k'**
  - Node size: **$s = 2^{k'}$** references
  - Fixed maximum height **h = k/k'**
  - Secondary index structure
- **Characteristics**
  - Partitioned data structure
  - **Order-preserving** (for range scans)
  - **Update-friendly**
- **Properties**
  - **Deterministic paths**
  - Worst-case complexity O(h)

insert (**107**,value4)

`0000 0000 0110 1011`

k = 16

k' = 4



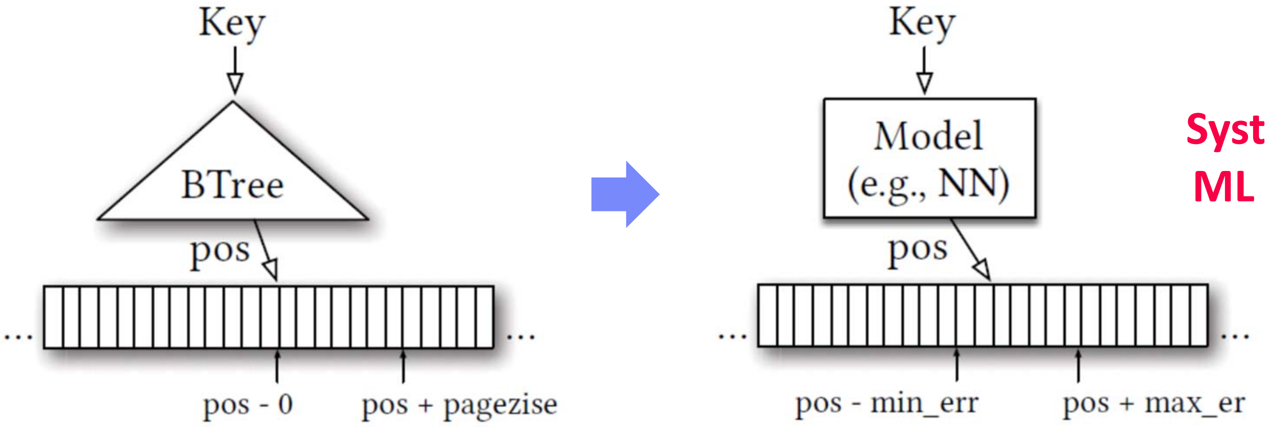
* not expanded

- **Bypass array**
- Adaptive **trie expansion**
- Memory preallocation + **reduced pointers**

Excursus: Learned Index Structures

- **A Case For Learned Index Structures**
  - Sorted data array, predict position of key
  - **Hierarchy of simple models** (stages models)
  - Tries to **approximate the CDF** similar to interpolation search (uniform data)

[Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis: The Case for **Learned Index Structures**. **SIGMOD 2018**]



Systems for ML, ML for Systems

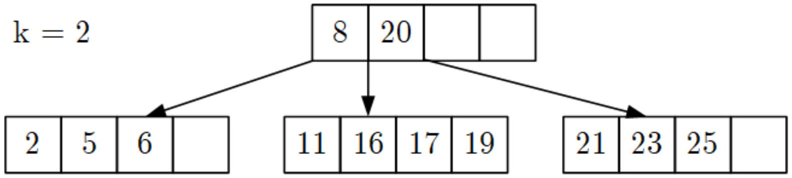
- **Follow-up Work on SageDBMS**

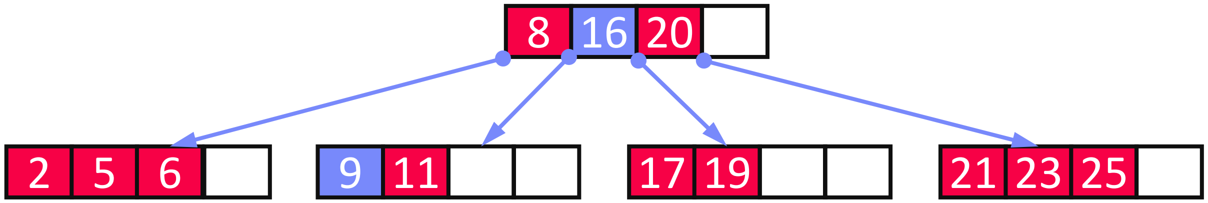
[Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, Vikram Nathan: **SageDB: A Learned Database System**. **CIDR 2019**]

BREAK (and Test Yourself)

- **Given B-tree below, insert key 9 and draw resulting B-tree** (7/100 points)

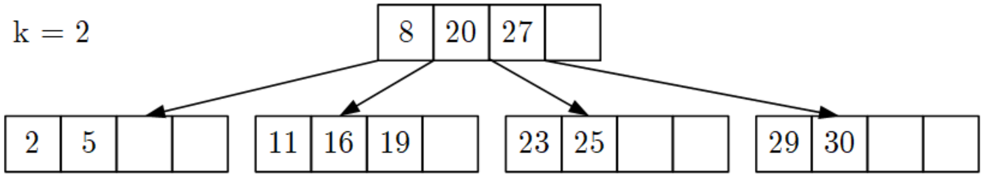
k = 2

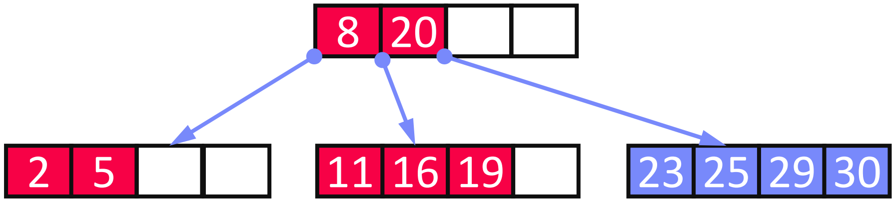




- **Given B-tree below, delete key 27, and draw resulting B-tree** (8/100 points)

k = 2

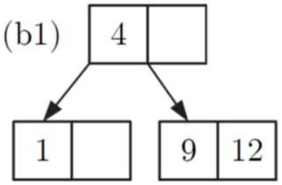




BREAK (and Test Yourself), cont.

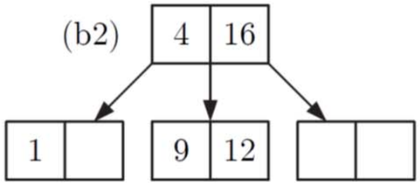
- **Which of the following trees are valid – i.e., satisfy the constraints of – B-trees with k=1? Mark each tree as valid or invalid and name the violations** (4/100 points)

(b1)



✓

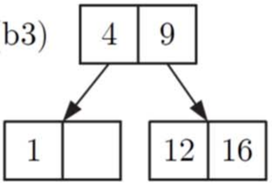
(b2)



✗

(empty leaf node, **underflow**)

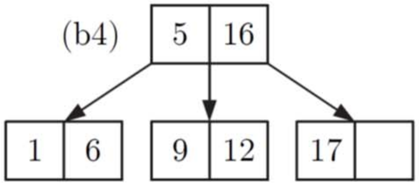
(b3)



✗

(**invalid #** of pointers and subtrees)

(b4)



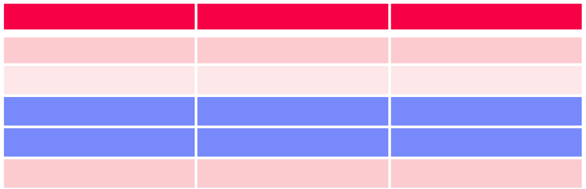
✗

(**invalid ordering** of data items, 6>5 but in left subtree)

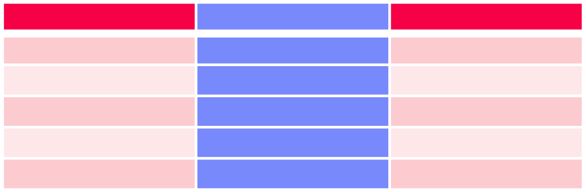
Table Partitioning

Overview Partitioning Strategies

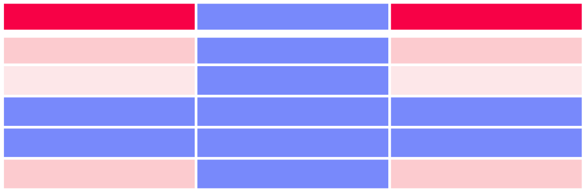
- **Horizontal Partitioning**
  - Relation partitioning into disjoint subsets



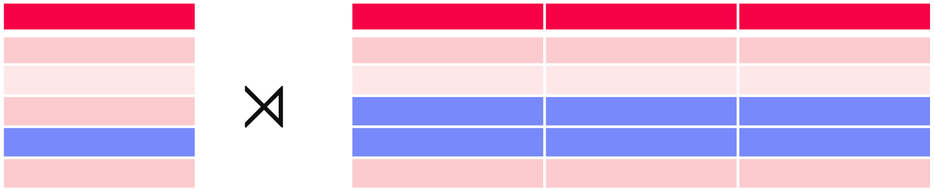
- **Vertical Partitioning**
  - Partitioning of attributes with similar access pattern



- **Hybrid Partitioning**
  - Combination of horizontal and vertical fragmentation (hierarchical partitioning)



- **Derived Horizontal Partitioning**

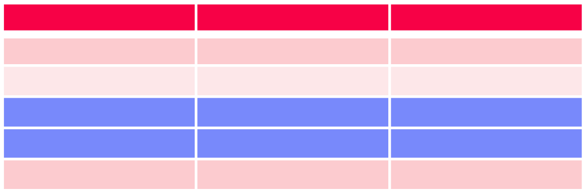


Correctness Properties

- **#1 Completeness**
  - $R \rightarrow R_1, R_2, \ldots, R_n$ (Relation R is partitioned into *n* fragments)
  - Each item from R must be included **in at least one fragment**
- **#2 Reconstruction**
  - $R \rightarrow R_1, R_2, \ldots, R_n$ (Relation R is partitioned into *n* fragments)
  - **Exact reconstruction** of fragments must be possible
- **#3 Disjointness**
  - $R \rightarrow R_1, R_2, \ldots, R_n$ (Relation R is partitioned into *n* fragments)
  - $\boldsymbol{R_i} \cap \boldsymbol{R_j} = \emptyset \;\; (1 \le i, j \le n;\; i \neq j)$

Horizontal Partitioning

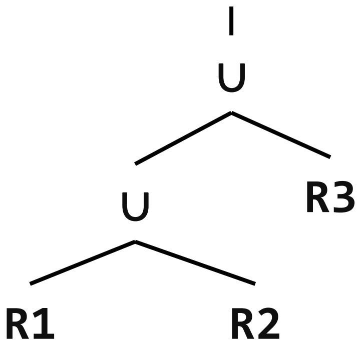
- **Row Partitioning into n Fragments R_i**
  - **Complete**, **disjoint**, **reconstructable**
  - **Schema of fragments is equivalent** to schema of base relation



- **Partitioning**
  - Split table by n **selection predicates** P_i (partitioning predicate) on attributes of R
  - Beware of attribute domain and skew

$$R_i = \sigma_{P_i}(R)$$
$$(1 \leq i \leq n)$$

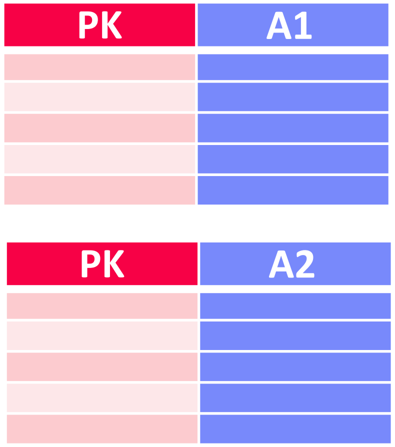
- **Reconstruction**
  - **Union** of all fragments
  - Bag semantics, but no duplicates across partitions



$$R = \bigcup_{1 \leq i \leq n} R_i$$

Vertical Fragmentation

- **Column Partitioning into n Fragments Ri**
  - **Complete**, **reconstructable**, but not disjoint (**primary key** for reconstruction via join)
  - Completeness: each attribute must be included in at least one fragment



- **Partitioning**
  - Partitioning via **projection**
  - Redundancy of primary key

$$R_i = \pi_{PK,A_i}(R)$$
$$(1 \le i \le n)$$

- **Reconstruction**
  - **Natural join** over primary key

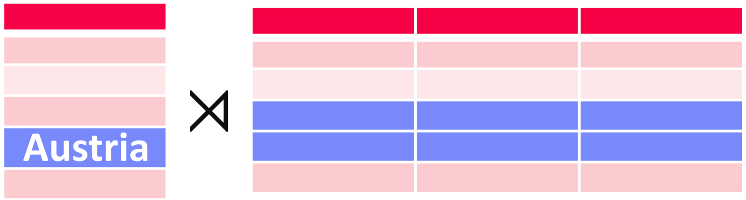
$$R = R_1 \bowtie R_i \bowtie R_n$$
$$(1 \le i \le n)$$

- **Hybrid horizontal/vertical partitioning**

$R = R_1 \bowtie R_i \bowtie R_n$ w/ $R_i = \cup R_{ij}$

→ $R = \cup R_j$ w/ $R_j = R_{1j} \bowtie R_{ij} \bowtie R_{nj}$

Derived Horizontal Fragmentation



- **Row Partitioning R into n fragements R_i, with partitioning predicate on S**
  - Potentially complete (not guaranteed), **restructable**, **disjoint**
  - Foreign key / primary key relationship determines correctness
- **Partitioning**
  - **Selection** on independent relation S
  - **Semi-join** with dependent relation R to select partition R_i

$$R_i = R \ltimes S_i = R \ltimes \sigma_{P_i}(S)$$
$$= \pi_{R.*}\left(R \bowtie \sigma_{P_i}(S)\right)$$

- **Reconstruction**
  - Equivalent to horizontal partitioning
  - **Union** of all fragments

$$R = \bigcup_{1 \leq i \leq n} R_i$$

Exploiting Table Partitioning

- **Partitioning and query rewriting**
  - **#1 Manual partitioning and rewriting**
  - **#2 Automatic rewriting** (spec. partitioning)
  - **#3 Automatic partitioning and rewriting**
- **Example PostgreSQL (#2)**

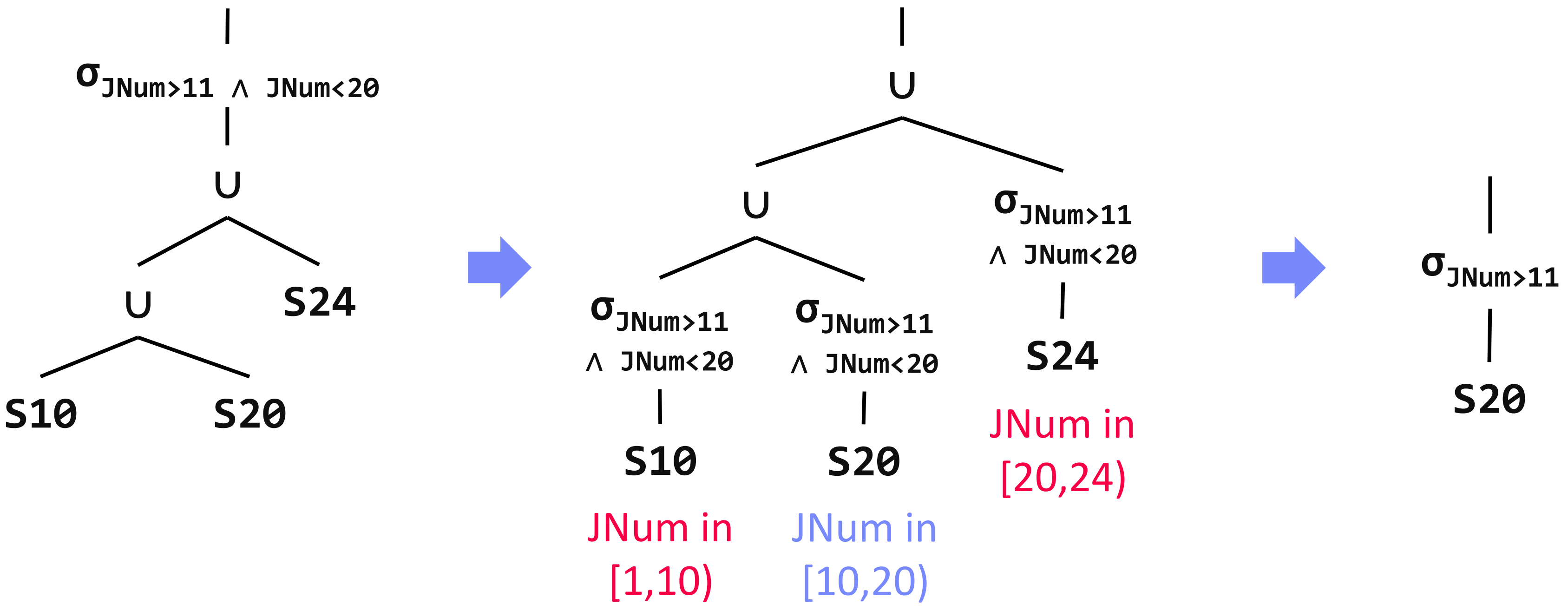
```
CREATE TABLE Squad(
   JNum INT PRIMARY KEY,
   Pos CHAR(2) NOT NULL,
   Name VARCHAR(256)
) PARTITION BY RANGE(JNum);


CREATE TABLE Squad10 PARTITION OF Squad
   FOR VALUES FROM (1) TO (10);

CREATE TABLE Squad20 PARTITION OF Squad
   FOR VALUES FROM (10) TO (20);

CREATE TABLE Squad24 PARTITION OF Squad
   FOR VALUES FROM (20) TO (24);
```

J#	Pos	Name
1	GK	Manuel Neuer
12	GK	Ron-Robert Zieler
22	GK	Roman Weidenfeller
2	DF	Kevin Großkreutz
4	DF	Benedikt Höwedes
5	DF	Mats Hummels
15	DF	Erik Durm
16	DF	Philipp Lahm
17	DF	Per Mertesacker
20	DF	Jérôme Boateng
3	MF	Matthias Ginter
6	MF	Sami Khedira
7	MF	Bastian Schweinsteiger
8	MF	Mesut Özil
9	MF	André Schürrle
13	MF	Thomas Müller
14	MF	Julian Draxler
18	MF	Toni Kroos
19	MF	Mario Götze
21	MF	Marco Reus
23	MF	Christoph Kramer
10	FW	Lukas Podolski
11	FW	Miroslav Klose

Exploiting Table Partitioning, cont.

- **Example, cont.**

```
SELECT * FROM Squad
   WHERE JNum > 11 AND JNum < 20
```

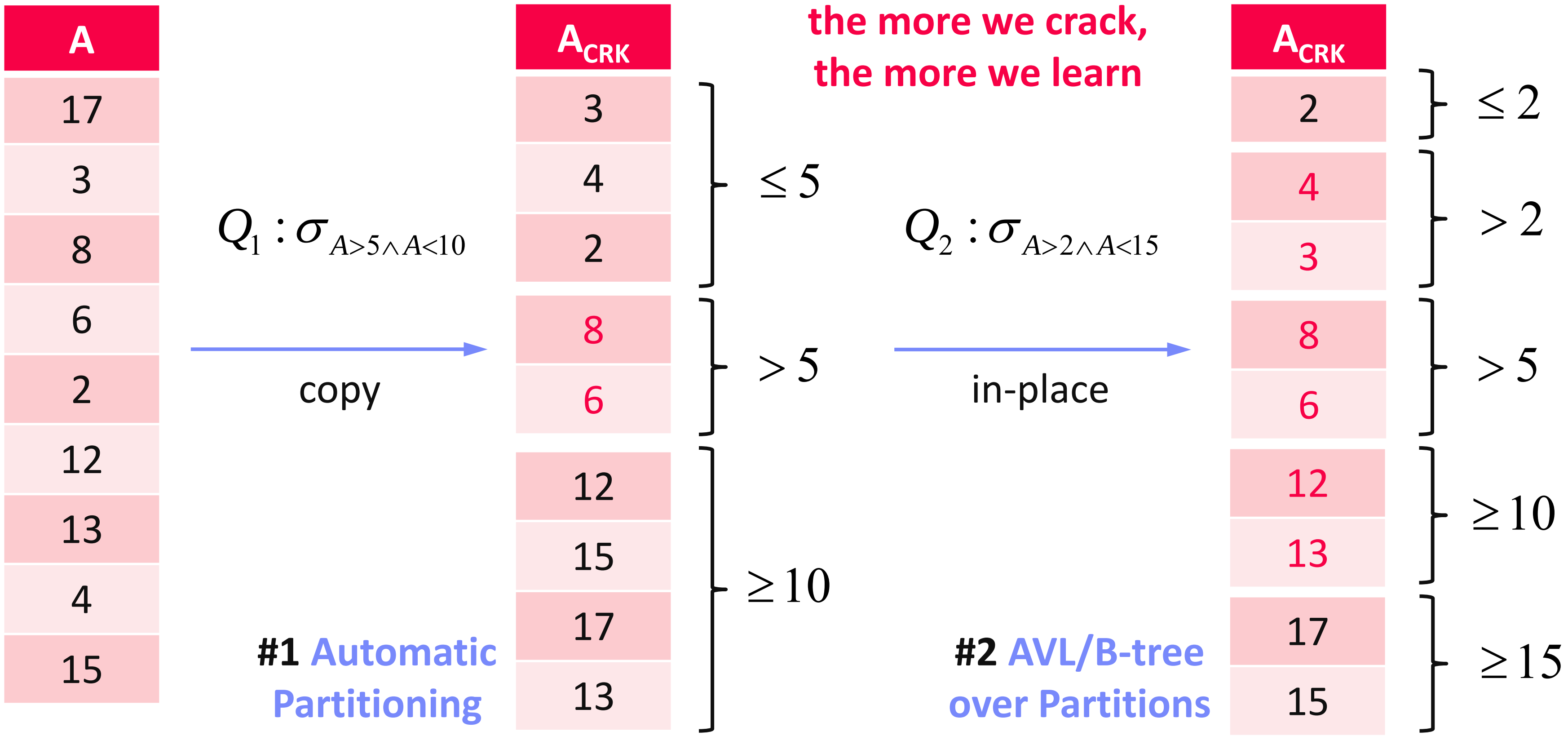


Excursus: Database Cracking

[Pedro Holanda et al: Progressive Indexes: Indexing for Interactive Data Analysis. **PVLDB 2019**]

[Stratos Idreos, Martin L. Kersten, Stefan Manegold: Database Cracking. **CIDR 2007**]

- **Core Idea:** Queries trigger physical reorganization (partitioning and indexing)



the more we crack, the more we learn

#1 Automatic Partitioning

#2 AVL/B-tree over Partitions

Materialized Views

Overview Materialized Views

- **Core Idea of Materialized Views**
  - Identification of frequently re-occuring queries (views)
  - Precompute subquery results once, store and reuse many times

- **The MatView Lifecycle**

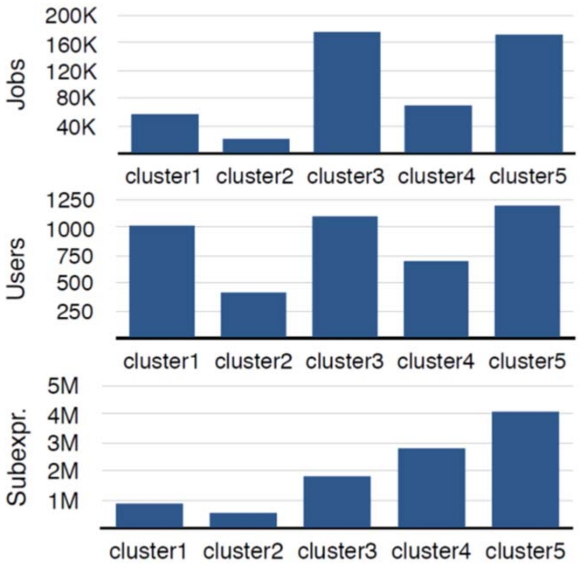
#1 View Selection
(automatic selection via advisor tools, approximate algorithms)

#3 View Maintenance
(maintenance time and strategy, when and how)

#2 View Usage
(transparent query rewrite for full/partial matches)

View Selection and Usage

- **Motivation**
  - Shared subexpressions very common in analytical workloads
  - Ex. **Microsoft's Analytics Clusters** (typical daily use -> 40% CSE saving)



- **#1 View Selection**
  - Exact view selection (query containment) is **NP-hard**
  - Heuristics, greedy and approximate algorithms

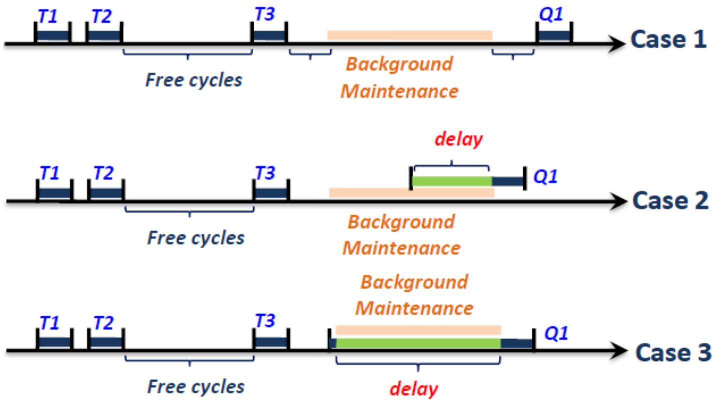
[Alekh Jindal, Konstantinos Karanasos, Sriram Rao, Hiren Patel: Selecting Subexpressions to Materialize at Datacenter Scale. **PVLDB 2018**]

[Leonardo Weiss Ferreira Chaves, Erik Buchmann, Fabian Hueske, Klemens Boehm: Towards materialized view selection for distributed databases. **EDBT 2009**]

- **#2 View Usage**
  - Given query and set of materialized view, decide which views to use and rewrite the query for produce correct results
  - Generation of compensation plans

View Maintenance – When?

- **Materialized view creates redundancy → Need for #3 View Maintenance**
- **Eager Maintenance (writer pays)**
  - Immediate refresh: updates are directly handled (consistent view)
  - On Commit refresh: updates are forwarded at end of successful TXs
- **Deferred Maintenance (reader pays)**
  - Maintenance on explicit user request
  - Potentially **inconsistent base tables and views**
- **Lazy Maintenance (async/reader pays)**
  - Same guarantees as eager maintenance
  - Defer maintenance until free cycles or view required (invisible for updates and queries)



[Jingren Zhou, Per-Åke Larson, Hicham G. Elmongui: Lazy Maintenance of Materialized Views. **VLDB 2007**]

View Maintenance – How?

- **Incremental Maintenance**
  - **Propagate:** Compute required updates
  - **Apply:** apply collected updates to the view

Example View:

```
SELECT A, SUM(B)
  FROM Sales
  GROUP BY CUBE(A)
```

A	SUM
NULL	107
X	30
Y	77

Global Net Delta ΔR

A	B
+ X	3
+ Z	9

Local View Delta ΔV_L

A	SUM
+ NULL	3
+ X	3
+ NULL	9
+ Z	9

[Global View Delta] ΔV_G

A	SUM
+ NULL	12
+ X	3
+ Z	9

Super Delta ΔV_S

A	SUM	SUM2
NULL	12	107
X	3	30
Z	9	NULL

Apply Delta ΔV_A

A	SUM
Update NULL	119
Update X	33
Insert Z	9

Incremental **Propagate**

Incremental **Apply**

Materialized Views in PostgreSQL

- **View Selection**
  - **Manual definition** of materialized view only
  - With or without data
- **View Usage**
  - **Manual use** of view
  - No automatic query rewriting
- **View Maintenance**
  - **Manual (deferred) refresh**
  - Complete, no incremental maintenance
  - Note: Community work on IVM

[Yugo Nagata: Implementing Incremental View Maintenance on PostgreSQL, **PGConf 2018**], patch in 2019

[Yugo Nagata: The Way for Updating Materialized Views Rapidly, **PGConf 2020,** https://www.pgcon.org/events/pgcon_2020/sessions/session/56/slides/47/pgcon2020_nagata_the_way_to_update_materialized_views_rapidly.pdf]

```
CREATE MATERIALIZED VIEW TopScorer AS
  SELECT P.Name, Count(*)
    FROM Players P, Goals G
    WHERE P.Pid=G.Pid AND G.GOwn=FALSE
    GROUP BY P.Name
    ORDER BY Count(*) DESC
WITH DATA;
```

```
REFRESH MATERIALIZED VIEW TopScorer;
```

Name	Count
James Rodríguez	6
Thomas Müller	5
Robin van Persie	4
Neymar	4
Lionel Messi	4
Arjen Robben	3

Conclusions and Q&A

- **Summary**
    - Physical Access Paths: Compression and Index Structures
    - Logical Access Paths: Table Partitioning and Materialized Views

- **Exercises**
    - Exercise grading by Dec 1
    - Exercise 2 deadline Dec 1
    - Exercise 3 published Dec 1, deadline Dec 23

- **Next Lectures** (Part A)
    - 08 Query Processing [Nov 30]
    - 09 Transaction Processing and Concurrency [Dec 07]