SCIENCE
PASSION
TECHNOLOGY

Database Systems
07 Physical Design & Tuning

Matthias Boehm

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

Last update: Apr 29, 2019

Announcements/Org

- **#1 Video Recording**
  - Since lecture 03, video/audio recording
  - Link in TeachCenter & TUbe (**video recorder fixed?**)

TUbe

- **#2 Statistics Exercise 1**
  - **All submissions accepted** (submitted/draft)
  - In progress of **grading**, but understaffed

65.3% → 77.4%

- **#3 New Exercise Rules**
  - **Still mandatory exercises** (final exam admission)
  - Relaxed 50% successful completion rule:
    1 exercise 2%-50% allowed

Ex.1 Sample / **Feedback**

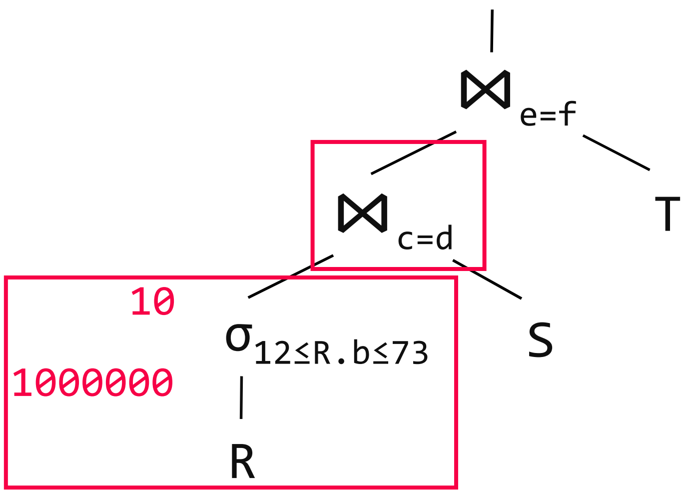
- **#4 Exercise 2**
  - Various issues (data cleaning, schema, etc), latest update Apr 22
  - Modified deadline: Apr 30 → **May 07 11.59pm**

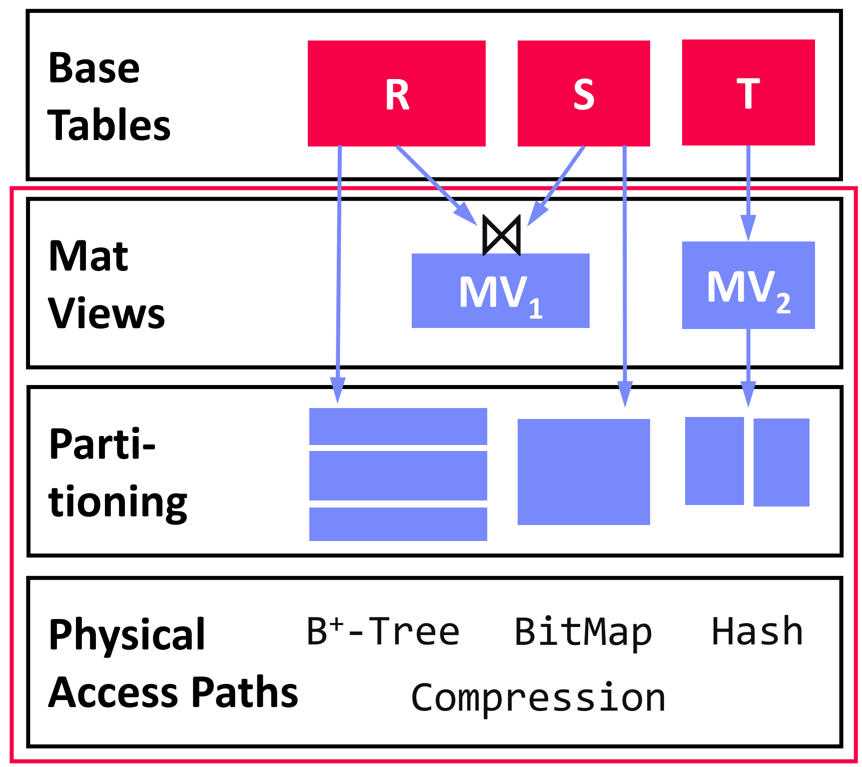
Physical Design, and why should I care?

- **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
```





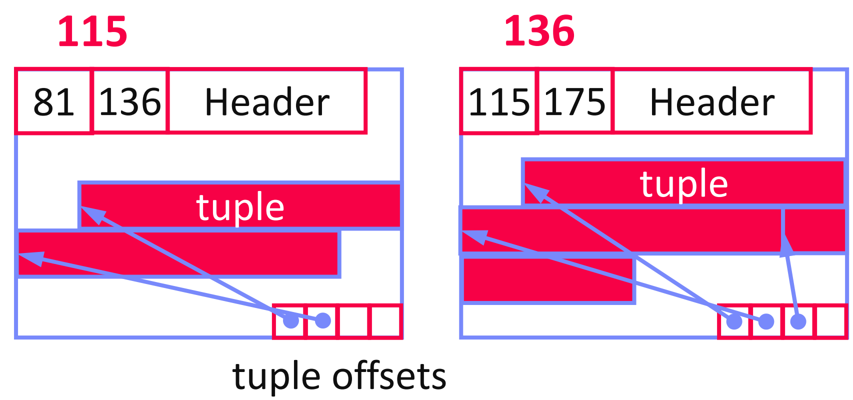
Agenda

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

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)



- **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...
```

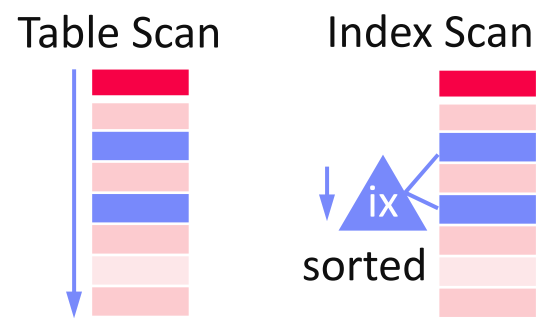
- **Frame-of-Reference Encoding**
  - Compress values by storing **delta to reference value** (outlier handling)

```
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

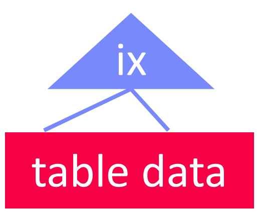
Overview Index Structures

- **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

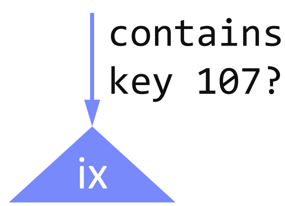


- **Uses of Indexes**

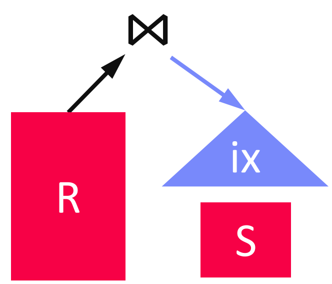
Lookups / Range Scans



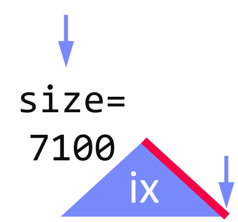
Unique Constraints



Index Nested Loop Joins



Aggregates (count, min/max)



Classification of Index Structures

[Theo Härder, Erhard Rahm: Datenbanksysteme – Konzepte und Techniken der Imple-mentierung, Springer, **2001**]

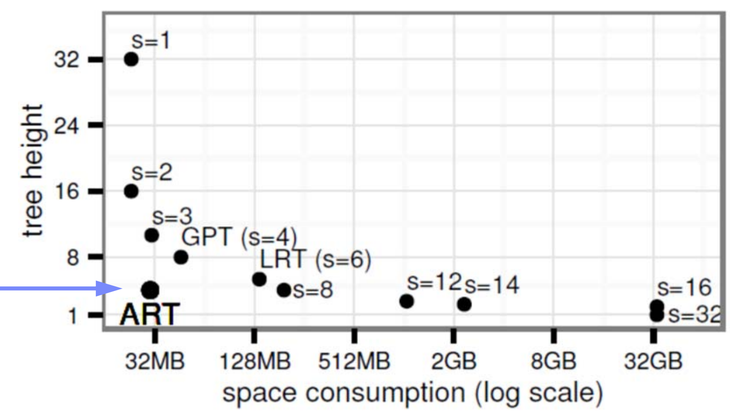
- **Traditional Classification**

- **1D Access Methods**
  - **Key Comparison**
    - **Sequential**: Sequential Lists, Linked Lists
    - **Tree-Based**: Binary Search Trees, **Multiway Trees** (**B-Tree**), Prefix Trees (Tries)
    - **Sort-Based**
  - **Key Transformation**
    - **Hash-Based**: Static, Dynamic

- **Prefix Trees for in-memory DBs**

[Matthias Boehm et al: Efficient In-Memory Indexing with Generalized **Prefix Trees**. **BTW 2011**]

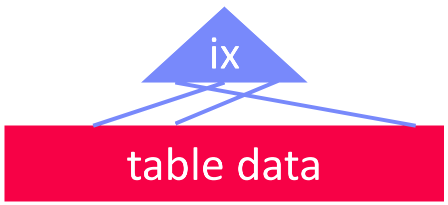
[Viktor Leis, Alfons Kemper, Thomas Neumann: The adaptive radix tree: ARTful Indexing for Main-Memory Databases. **ICDE 2013**]



Recap: Index Creation/Deletion via SQL

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

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



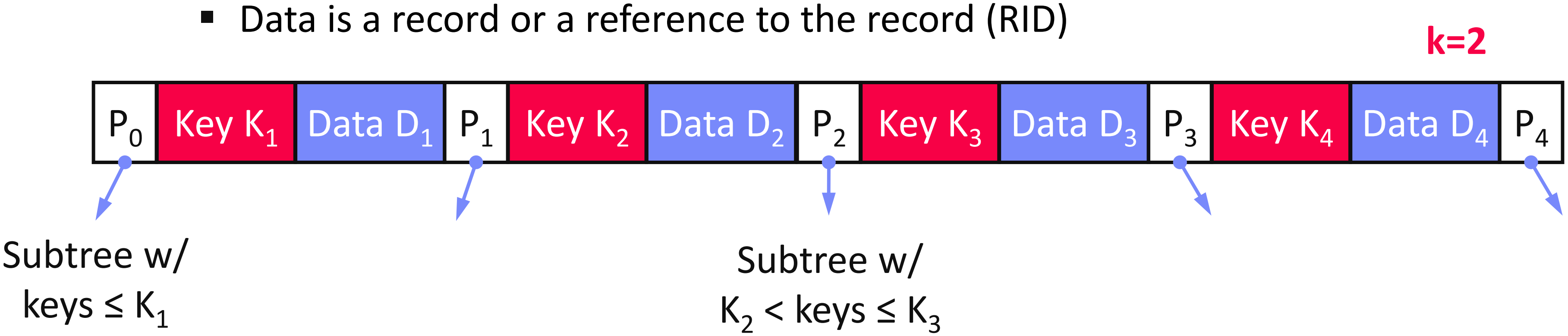
- **Delete Index**
  - Drop indexes by name

```
DROP INDEX ixStudLname;
```

- **Tradeoffs**
  - Indexes often automatically created for **primary keys** / **unique** attributes
  - **Lookup/scan/join performance** vs **insert performance**
  - Analyze usage statistics: `pg_stat_user_indexes, pg_stat_all_indexes`

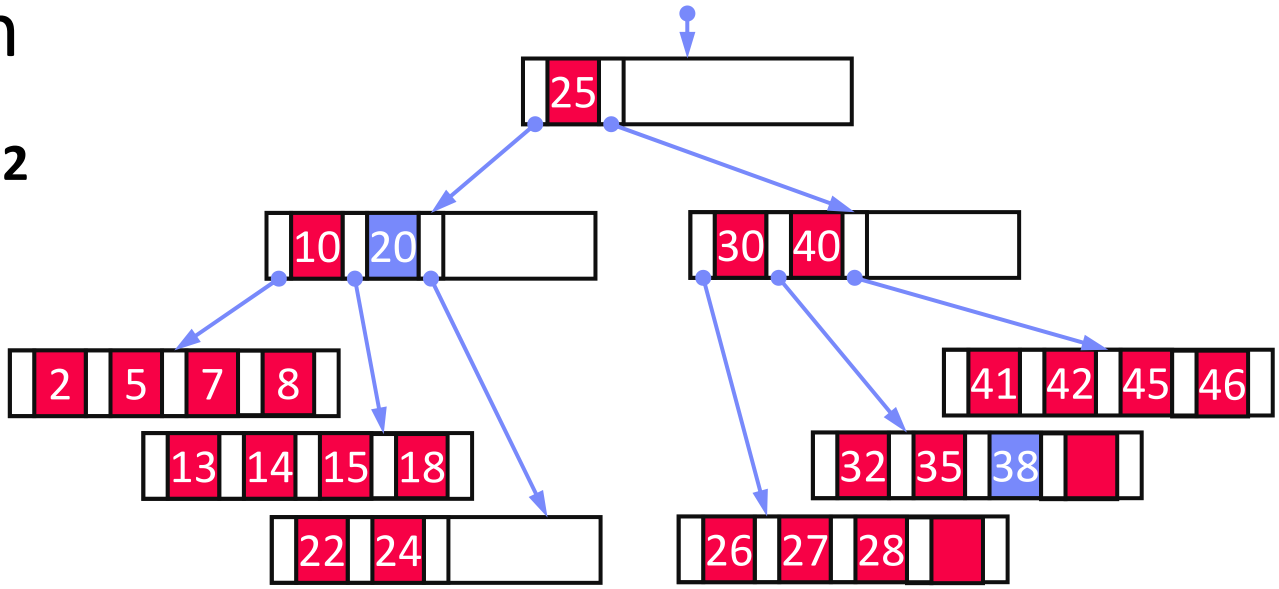
B-Tree Overview

- **History B-Tree**
  - **B**ayer and McCreight 1972 (multiple papers), **B**lock-based, **B**alanced, **B**oeing
  - 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 $\lceil \log_{2k+1}(n+1) \rceil \le h \le \left\lceil \log_{k+1}\left(\frac{n+1}{2}\right) \right\rceil + 1$
  - All nodes (except root=leaf) 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)



B-Tree Search

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



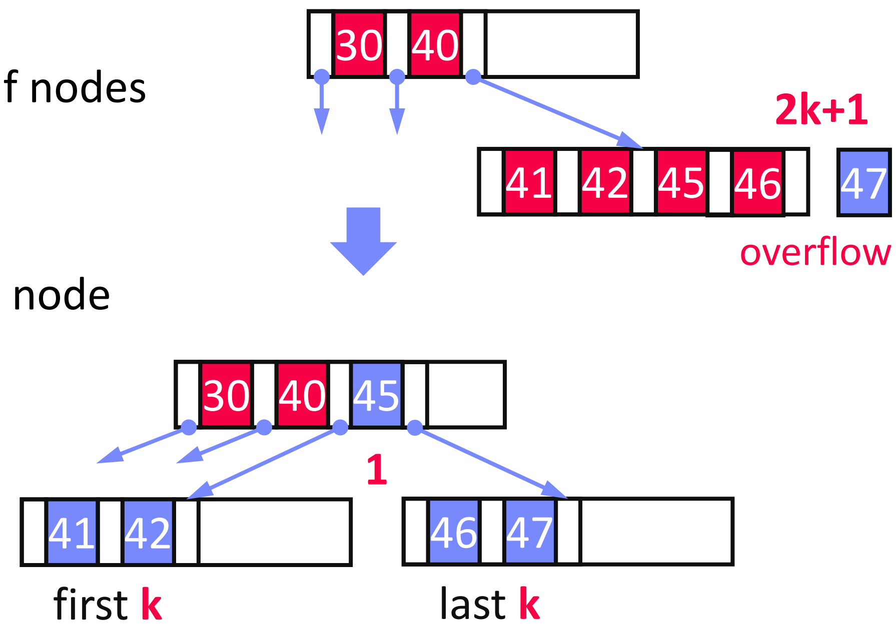
- **Lookup Q_K within a node**
  - Scan / binary search keys for Q_K, if $K_i=Q_K$, return D_i
  - If node does not contain key
    - If leaf node, abort search w/ NULL (not found), otherwise
    - Decent into subtree Pi with $K_i < Q_K \leq K_{i+1}$
- **Range Scan $Q_{L<K<U}$**
  - Lookup Q_L and call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

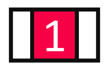
- **Node Splitting Approach**
  - Split the **2k+1** entries into two leaf nodes
  - **Left node:** first k entries
  - **Right node:** last k entries
  - (k+1)th entry inserted into parent node
    ➔ can cause **recursive splitting**
  - Special case: root split (h++)

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



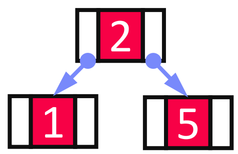
B-Tree Insert, cont. (Example w/ k=1)

- **Insert 1**

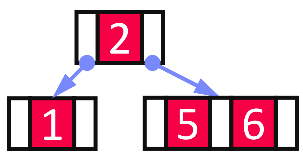


- **Insert 5**

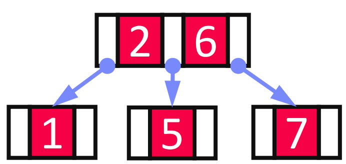
- **Insert 2 (split)**



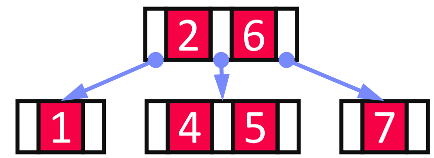
- **Insert 6**



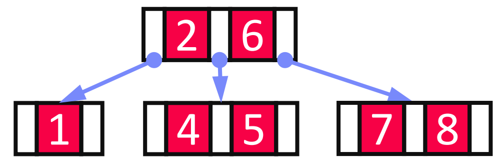
- **Insert 7 (split)**



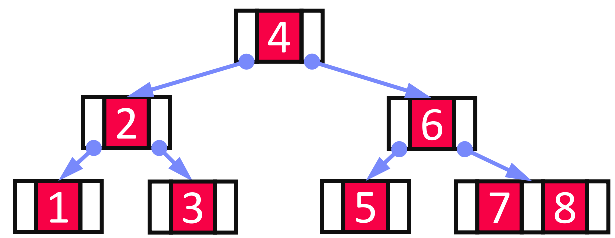
- **Insert 4**



- **Insert 8**



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

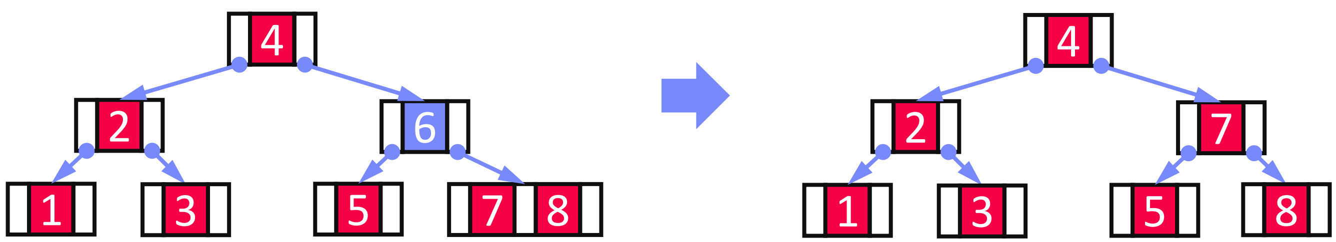


- **Note: Exercise 03, Task 3.2 (B-tree insertion and deletion)**

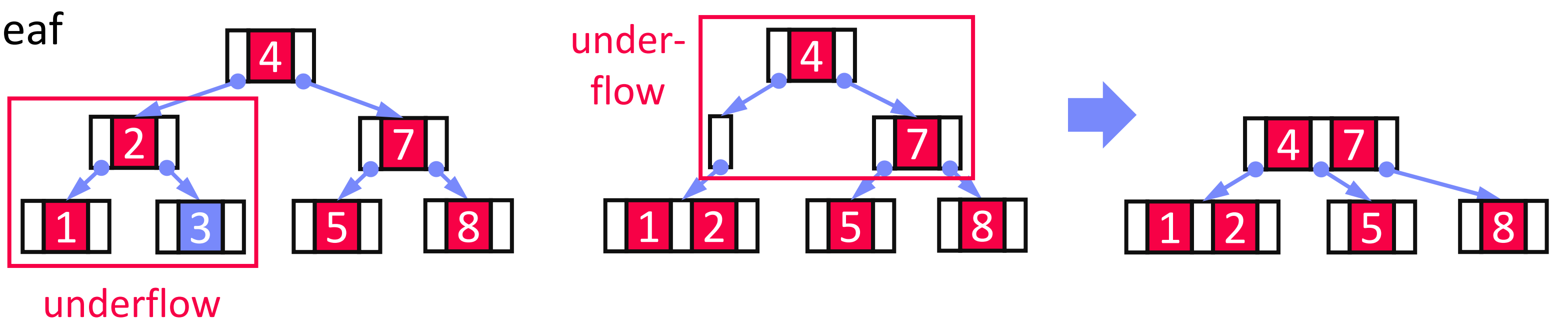
B-Tree Delete

- **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



Prefix Trees (Radix Trees, Tries)

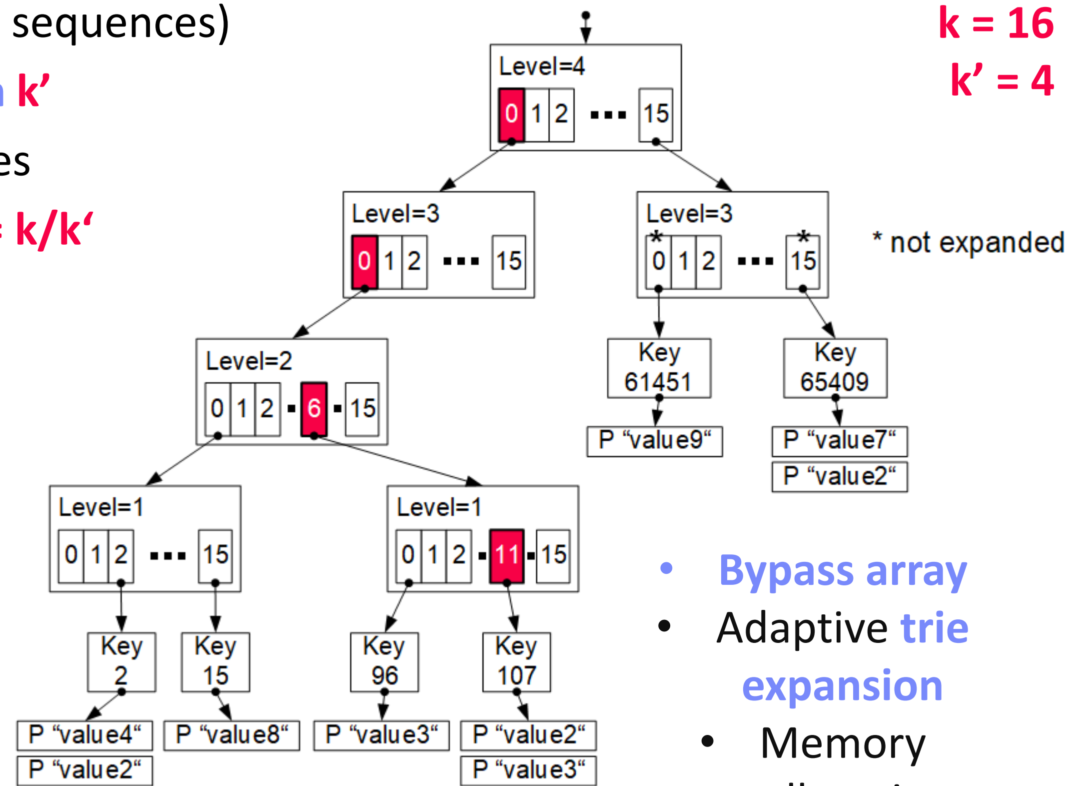
insert (107,value4)

0000 0000 0110 1011

k = 16

k' = 4

- **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)

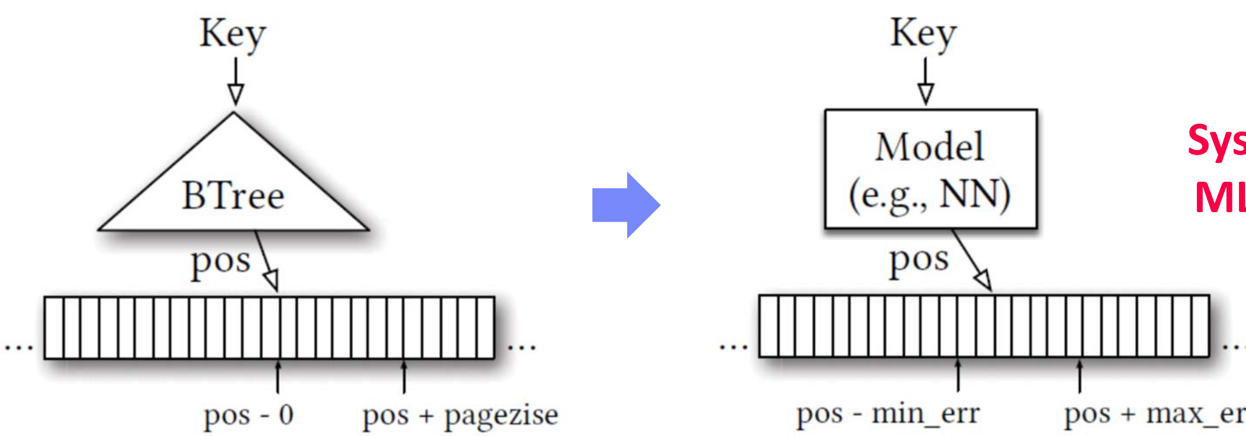


- Bypass array
- Adaptive trie expansion
- Memory preallocation + reduced pointers

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**

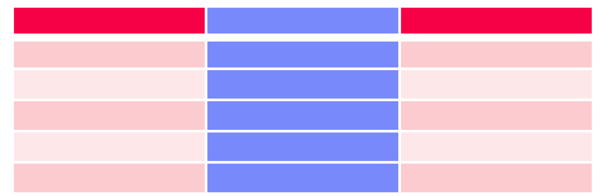
[Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, Vikram Nathan: SageDB: A Learned Database System. **CIDR 2019**]

Table Partitioning

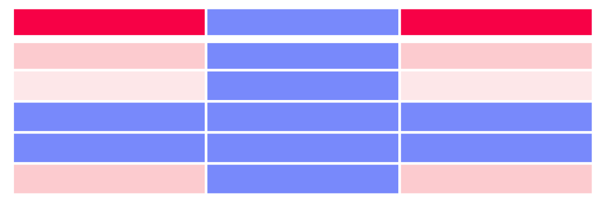
Overview Partitioning Strategies

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

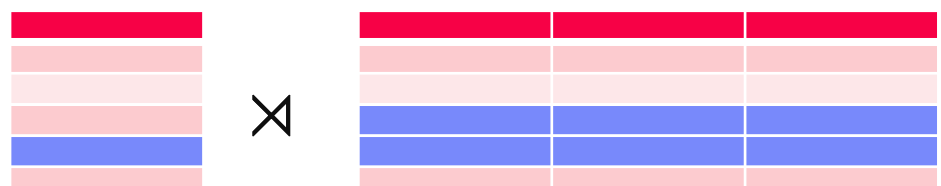
- **Vertical Partitioning**
  - Partitioning of attributes with similar access pattern



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



- **Derived Horizontal Partitioning**



Correctness Properties

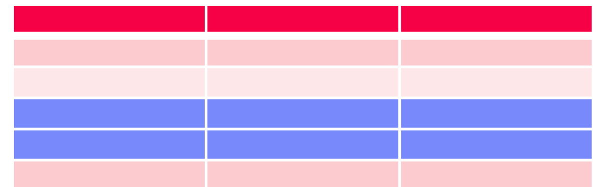
- **#1 Completeness**
  - $R \rightarrow R_1, R_2, \ldots, R_n$ (Relation R is partitioned into *n* fragments)
  - Each item from R must be included **in at least one fragment**

- **#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)
  - $\boldsymbol{R_i \cap R_j = \emptyset}\ (1 \leq i,j \leq n;\ i \neq j)$

Horizontal Partitioning

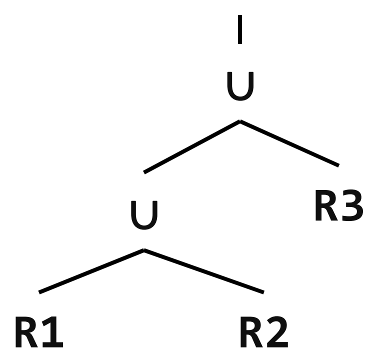
- **Row Partitioning into n Fragments R_i**
  - **Complete**, **disjoint**, **reconstructable**
  - **Schema of fragments is equivalent** to schema of base relation



- **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

PK	A1	A2

PK	A1

PK	A2

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

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

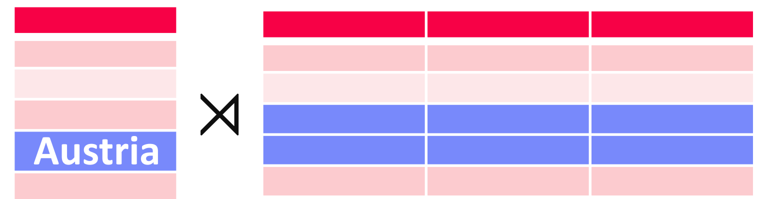
- **Reconstruction**
  - **Natural join** over primary key

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

- **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)**

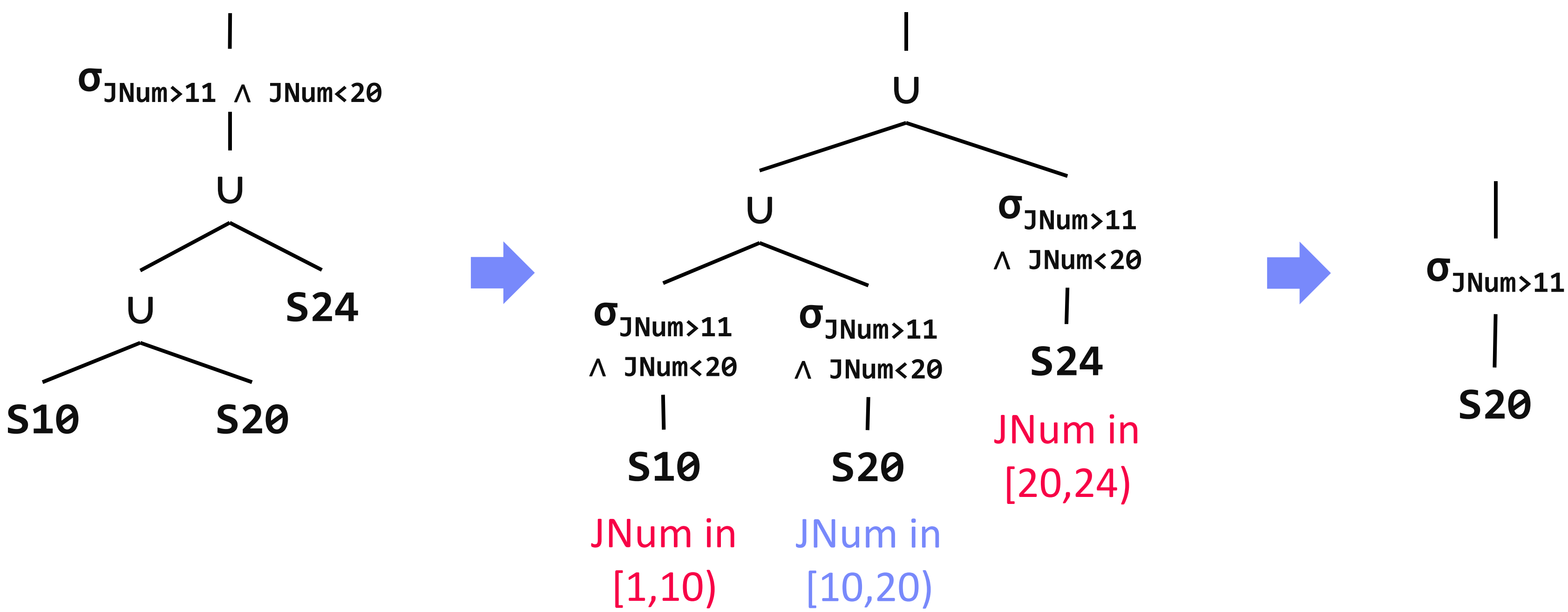
```
CREATE TABLE Squad(
    JNum INT PRIMARY KEY,
    Pos CHAR(2) NOT NULL,
    Name VARCHAR(256)
) PARTITION BY RANGE(JNum);

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.**

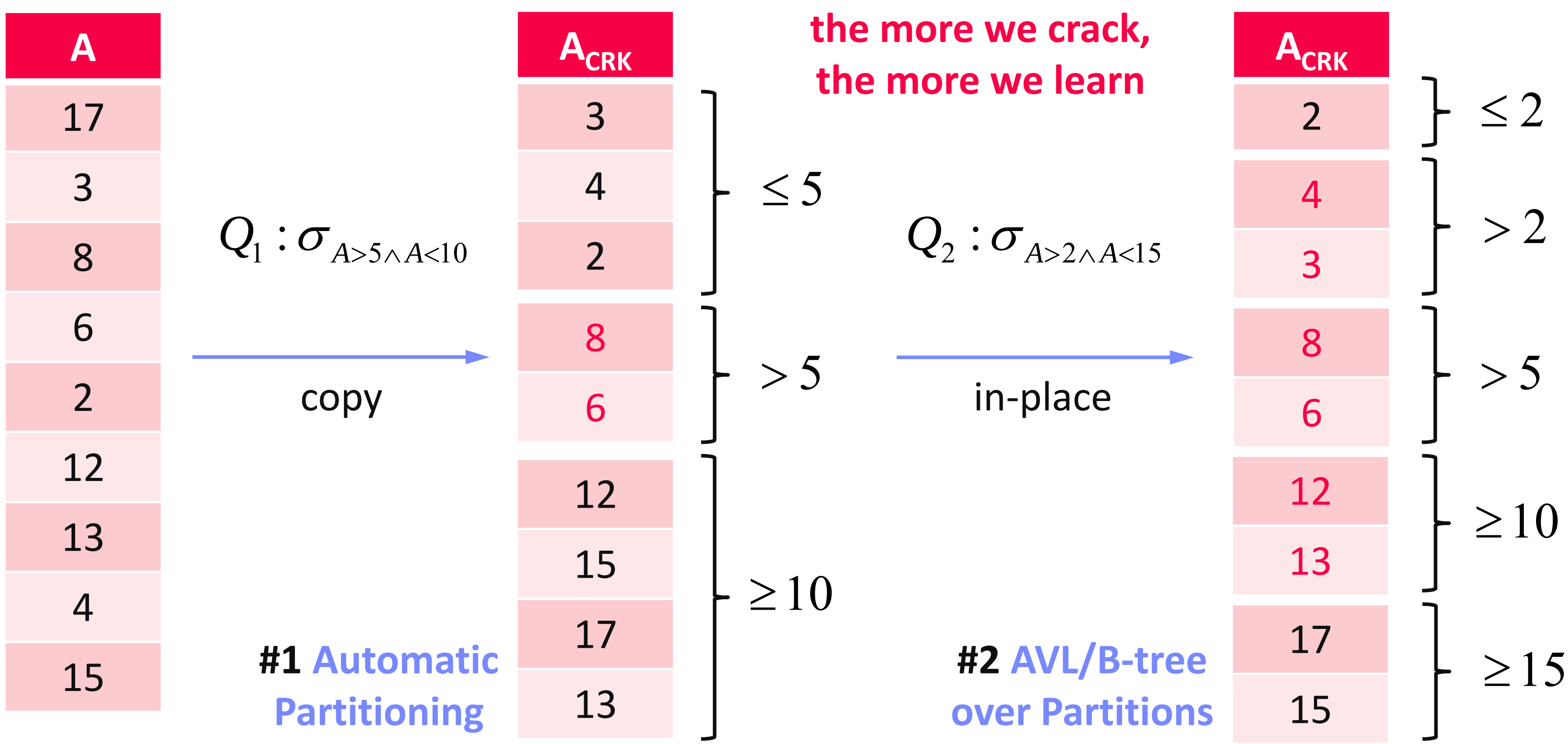
```
SELECT * FROM Squad
   WHERE JNum > 11 AND JNum < 20
```



Excursus: Database Cracking

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

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



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

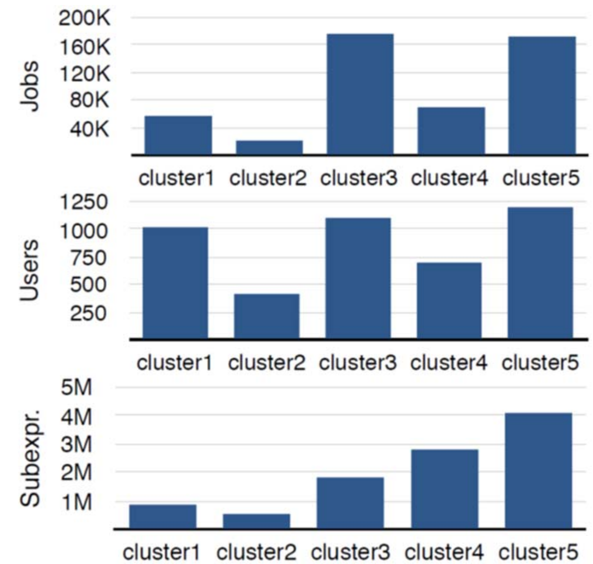
- **The MatView Lifecycle**

#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**

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

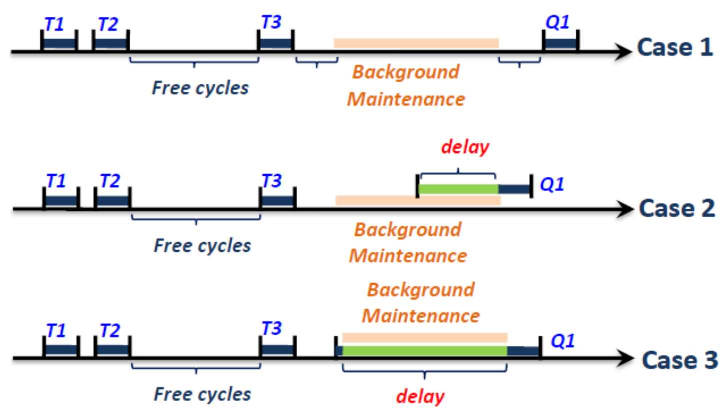
[Alekh Jindal, Konstantinos Karanasos, Sriram Rao, Hiren Patel: Selecting Subexpressions to Materialize at Datacenter Scale. **PVLDB 2018**]

[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** (Global Net Delta → Local View Delta)

Incremental **Apply** ([Global View Delta] → Super Delta → Apply Delta)

Materialized Views in PostgreSQL

- **View Selection**
  - **Manual definition** of materialized view only
  - With or without data

```
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;
```

- **View Usage**
  - **Manual use** of view
  - No automatic query rewriting

```
REFRESH MATERIALIZED VIEW TopScorer;
```

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

- **View Maintenance**
  - **Manual (deferred) refresh**
  - Complete, no incremental maintenance
  - Note: Community work on IVM

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

Conclusions and Q&A

- **Summary**
  - **Physical Access Paths:** Compression and Index Structures
  - **Logical Access Paths:** Table Partitioning and Materialized Views
- **Exercise 2 Reminder**
  - Submission deadline: Apr 30 → **May 07 11.59pm**
  - **Start early** (most time consuming of all four exercises)
- **Next Lectures**
  - May 6: **08 Query Processing**
  - May 13: **09 Transaction Processing and Concurrency**