Architecture of DB Systems
04 Index Structures and Partitioning

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

SCIENCE
PASSION
TECHNOLOGY

Last update: Oct 28, 2020

Announcements/Org

- **#1 Video Recording**
  - Link in TeachCenter & TUbe (lectures will be public)
  - Optional attendance (independent of COVID)
- **#2 COVID-19 Restrictions** (HS i5)
  - Corona Traffic Light: **Orange**
  - Max 25% room capacity (TC registrations)

max 18/74

- **#3 Programming Projects**
  - Updated `Project_Setup.zip`, news group will be set up
  - **Requirements**
    - Test suite must pass, no test cheating and gaming
    - Min performance target: `T(SUT) < 4 * T(ref_impl)` tested on 32 indexes/threads, different benchmarks and sizes
    - **Deadline:** **Thu Jan 21 11.59pm**

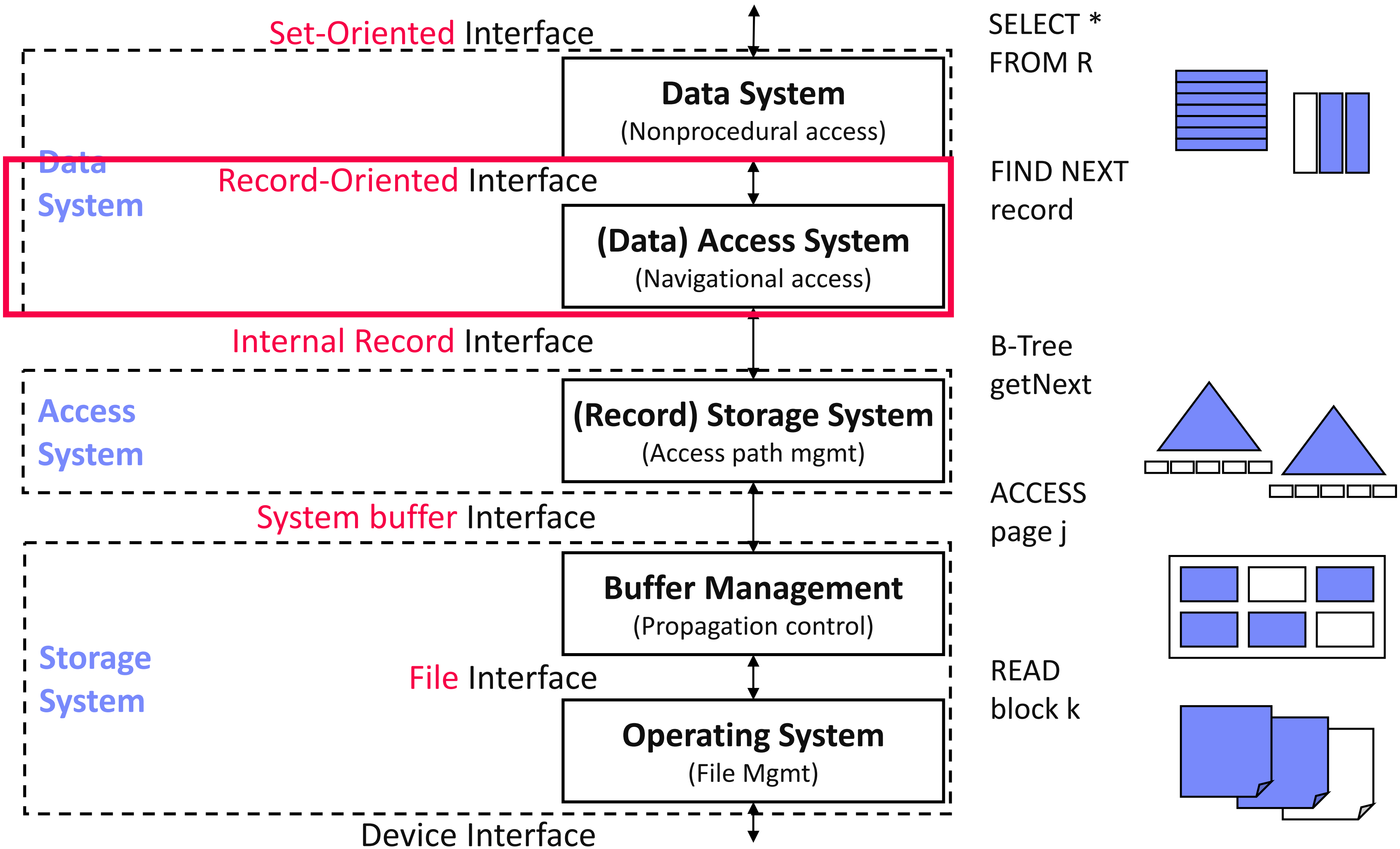
Agenda

- **Overview Access Methods**
- **Index Structures**
- **Partitioning and Pruning**
- **Adaptive and Learned Access Methods**

Overview Access Methods

DBMS Architecture, cont.

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

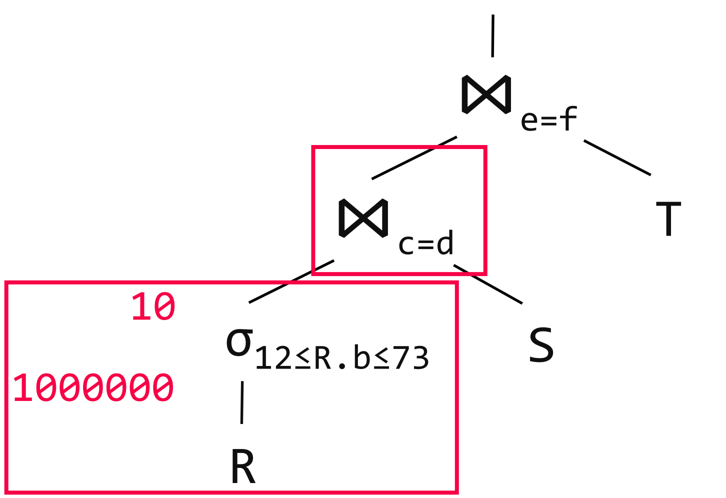


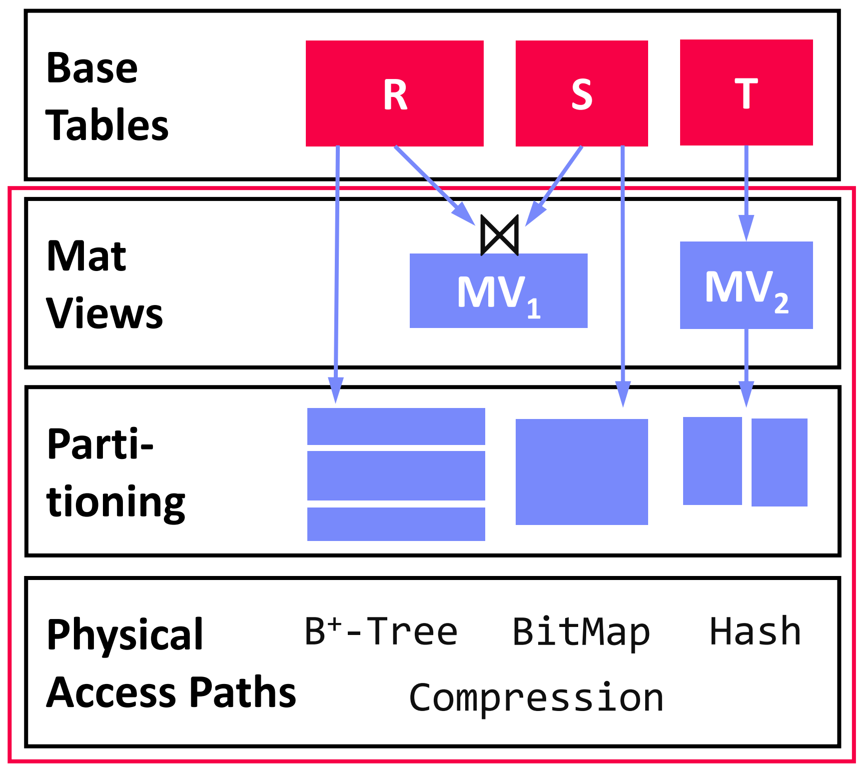
Access Methods and Physical Design

- **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**

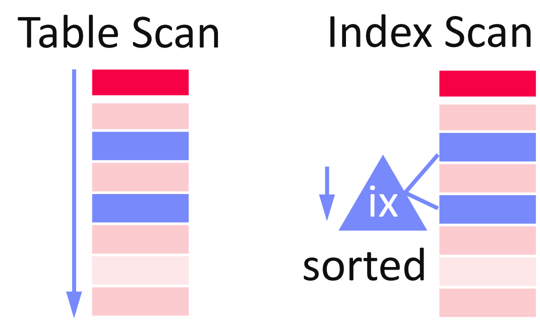
```
SELECT * FROM R, S, T
  WHERE R.c = S.d AND S.e = T.f
    AND R.b BETWEEN 12 AND 73
```





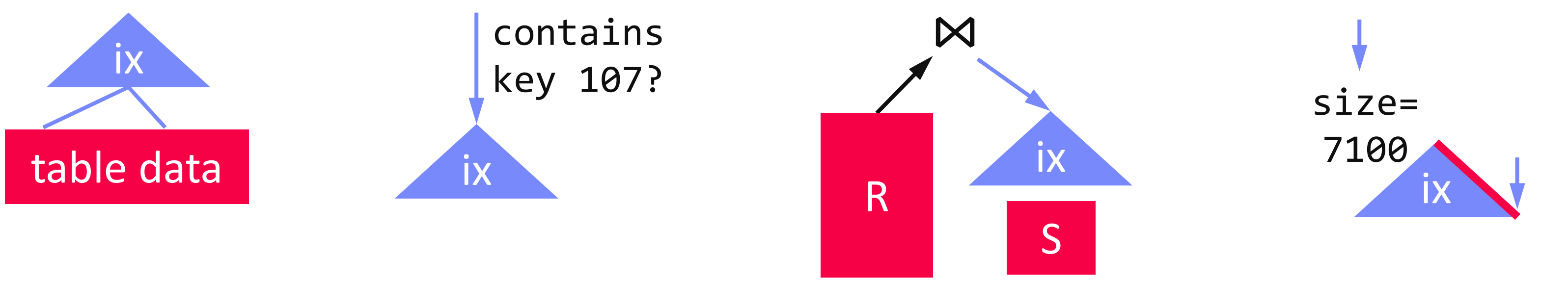
Overview Index Structures

- **Table Scan vs Index Scan**
  - For highly selective predicates, index scan **asymptotically much better** than table scan
  - Index scan **higher overhead** (~5% break even)
    - IXScan → TID-Sort → TID-Fetch
    - Multi-column predicates: TID-list intersection



- **Use Cases for Indexes**

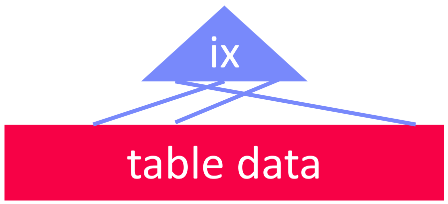
Lookups / Range Scans	Unique Constraints	Index Nested Loop Joins	Aggregates (count, min/max)



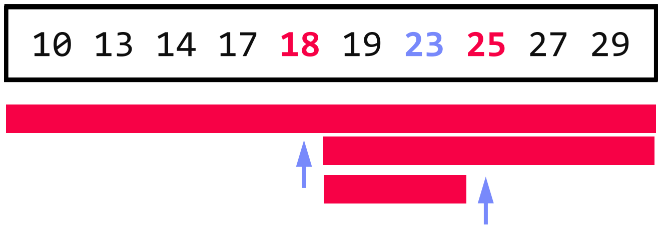
Additional Terminology

- **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);
```



- **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)`)

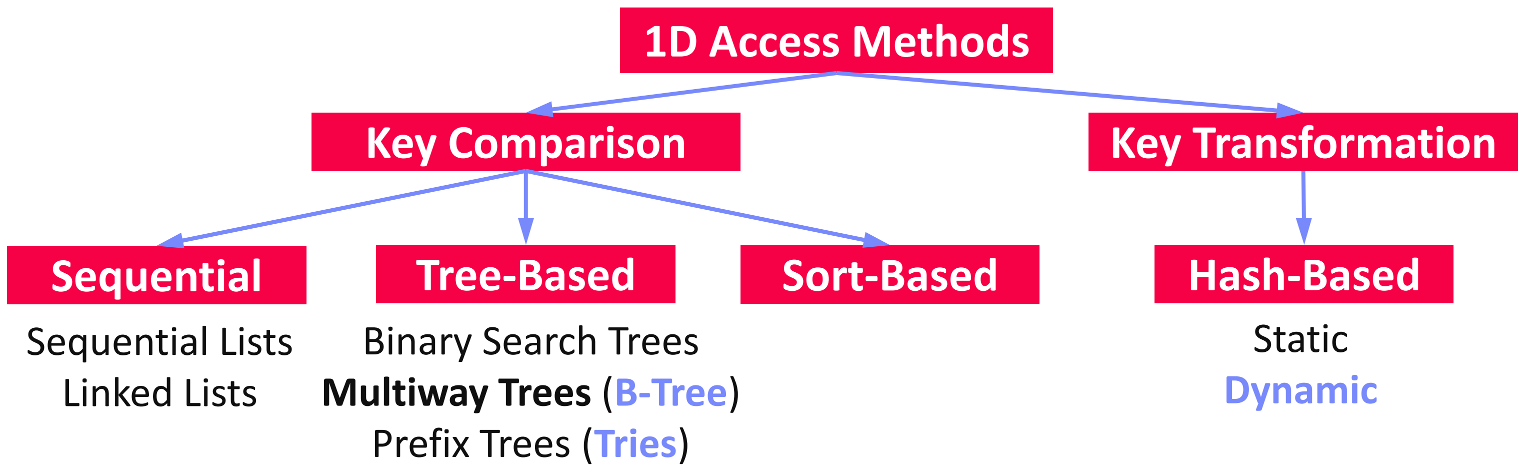


Index Structures

Classification of Index Structures

- **1D Access Methods**

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



- **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

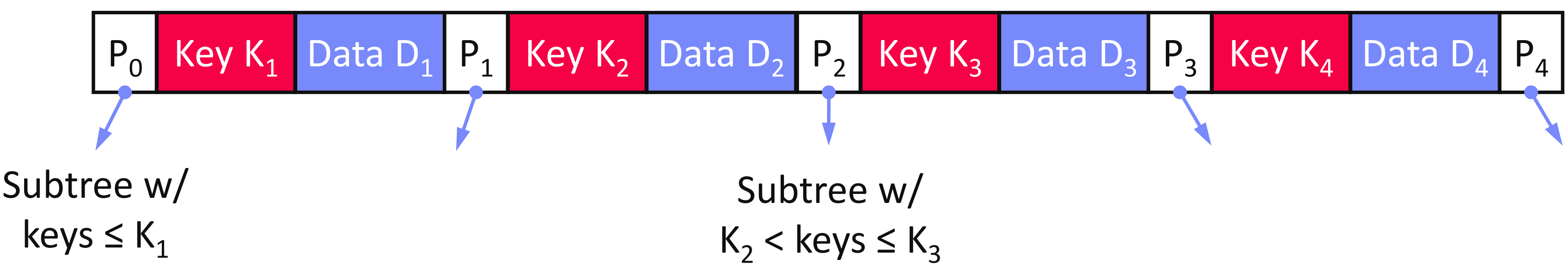
[Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. **Acta Inf. (1) 1972**]

- **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)

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

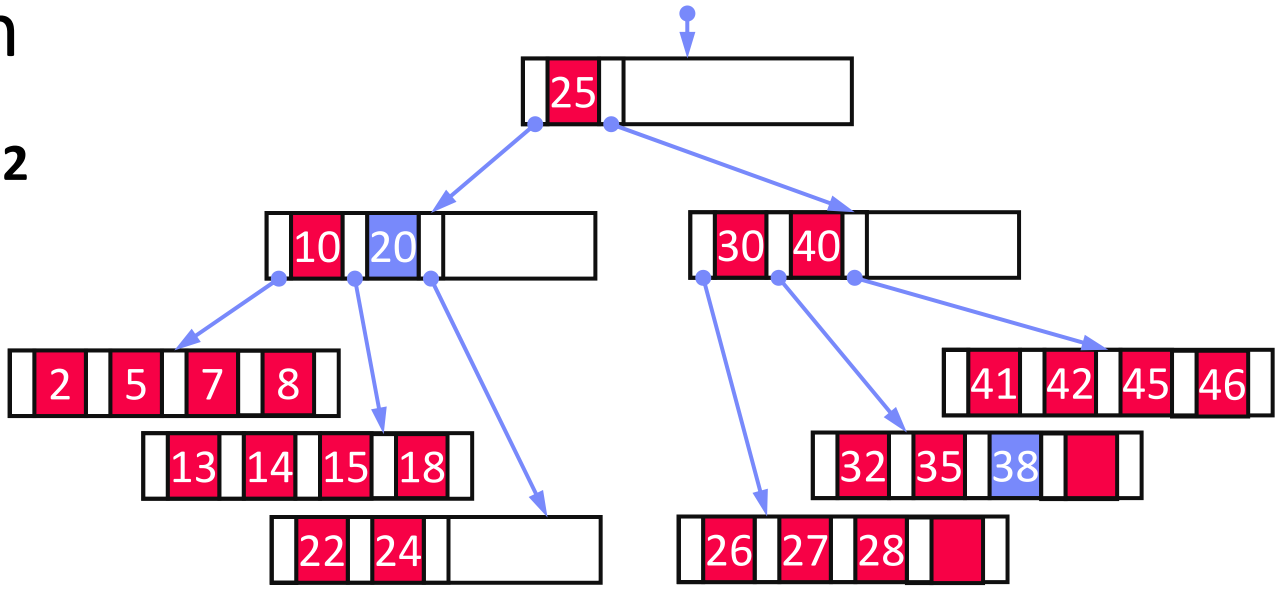
All nodes adhere to max constraints

k=2



B-Tree Search

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



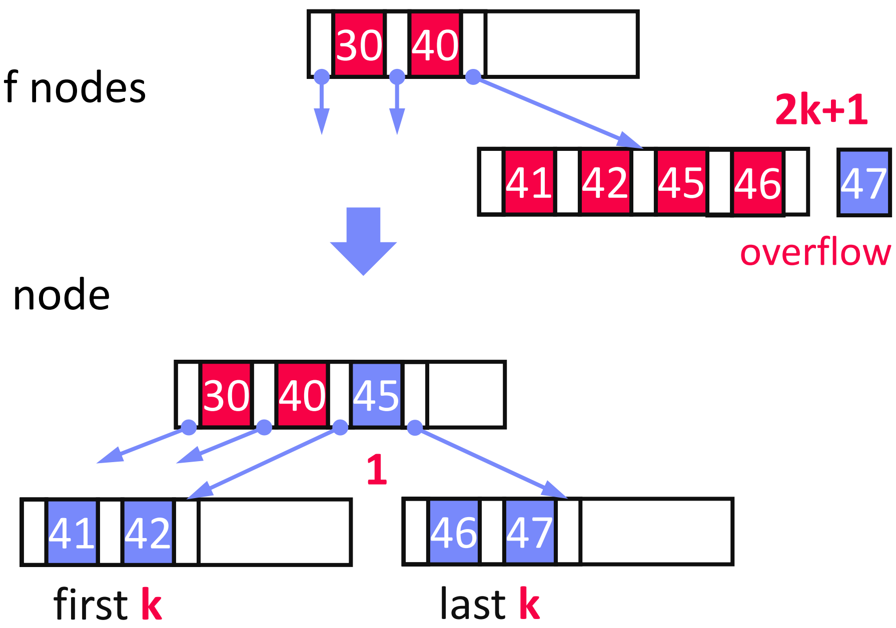
- **Lookup Q_K within a node**
  - Scan / binary search keys for Q_K, if K_i=Q_K, return D_i
  - If node does not contain key
    - If leaf node, abort search w/ NULL (not found), otherwise
    - Decent into subtree Pi with $K_i < Q_K \leq K_{i+1}$

- **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**

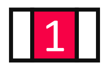
- **Node Splitting Approach**
  - Split the **2k+1** entries into two leaf nodes
  - **Left node:** first k entries
  - **Right node:** last k entries
  - (k+1)th entry inserted into parent node
    ➔ can cause **recursive splitting**
  - Special case: root split (h++)



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

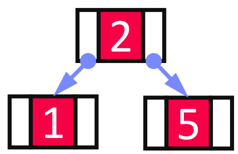
B-Tree Insert, cont. (Example w/ k=1)

- **Insert 1**

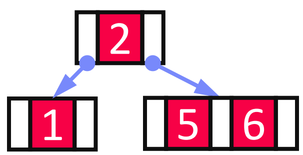


- **Insert 5**

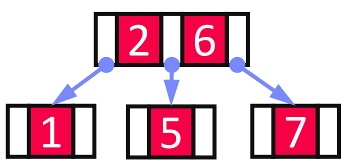
- **Insert 2 (split)**



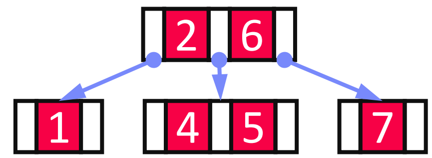
- **Insert 6**



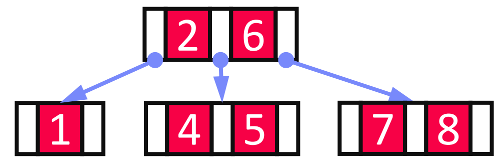
- **Insert 7 (split)**



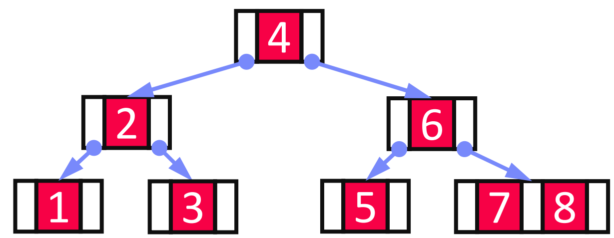
- **Insert 4**



- **Insert 8**



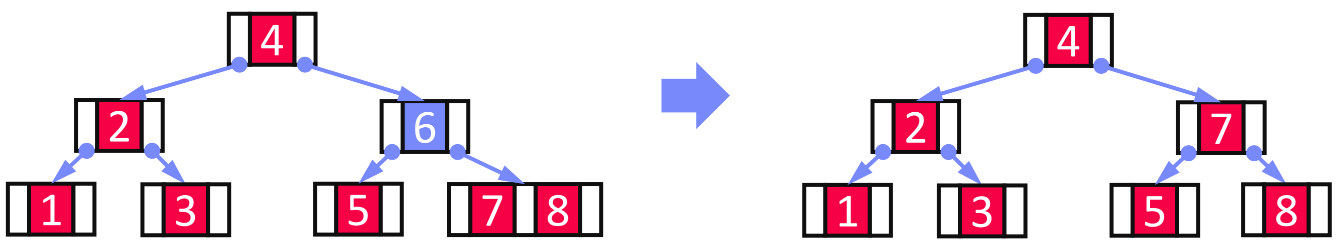
- **Insert 3 (2x split)**



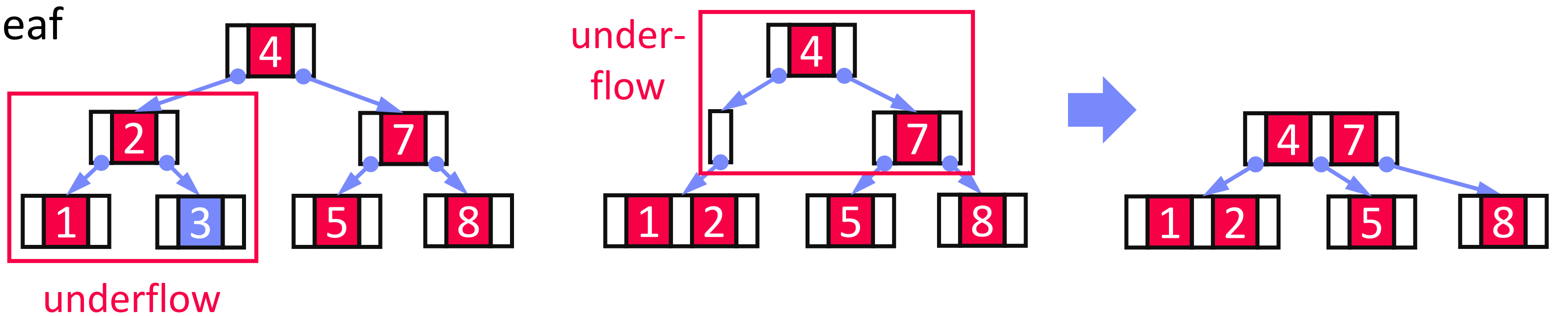
B-Tree Delete

- **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

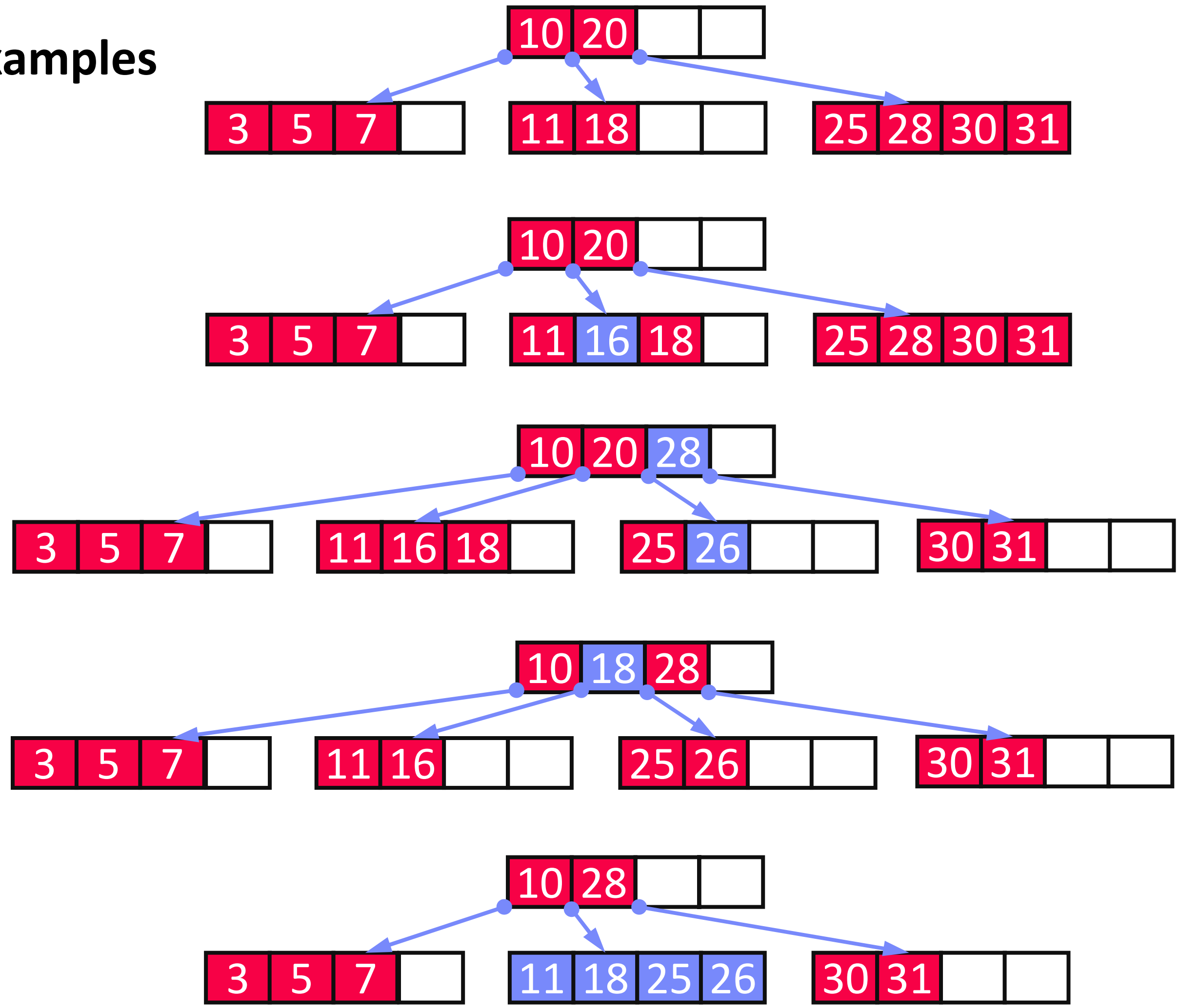


  - Case leaf



B-Tree Insert and Delete w/ k=2

- **Insert/Delete Examples**
  - Original
  - Insert 16
  - Insert 26
  - Delete 20
  - Delete 16



B-tree – Advanced Aspects

[Goetz Graefe: Modern B-Tree Techniques. **Found. Trends Databases** 3(4): 203-402, **2011**]

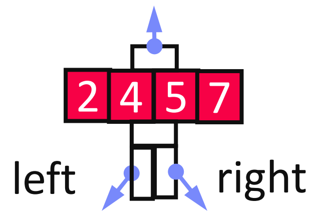
- **Variable-Length Fields**
  - In-page slot-array to variable length fields → direct lookup
  - With fixed page size, **no guarantees on min/max entries**
  - **Various approaches:** overflow pages, pick separators during bulk loading

- **Concurrent Access**
  - DB locks: only leaf nodes for B+ tree in practice at **value/value ranges**
  - Concurrent threads require page latching (parent-child)

- **Duplicate Keys**
  - **#1** use **prefix truncation** for compression → store common prefix once)
  - **#2** **concatenate key-TID** for unique lockups w/ O(log N)
  - Duplicate records as replicates or once w/ counter

Other In-Memory Trees

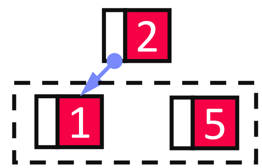
- **Balanced Binary Trees**
  - Red-Black Tree, AVL Tree (left/right height diff 1)
  - T tree (combines pros of AVL and B trees)



[G. M. Adel'son-Vel'skii and E. M. Landis: An algorithm for the organization of information, Soviet Mathematics Doklady, 3, **1962**]

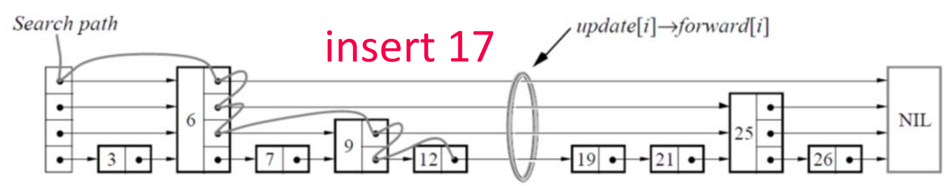
[Tobin J. Lehman, Michael J. Carey: A Study of Index Structures for Main Memory Database Management Systems. **VLDB 1986**]

- **CSB⁺-Tree**
  - Align node size to cache line (64B)
  - Reduce pointers via node groups
  - More keys, higher fan-out, at cost of slower insert



[Jun Rao, Kenneth A. Ross: Making B+-Trees Cache Conscious in Main Memory. **SIGMOD 2000**]

- **Skip Lists**
  - Linked list with multiple levels
  - Fraction p w/ level i pointers

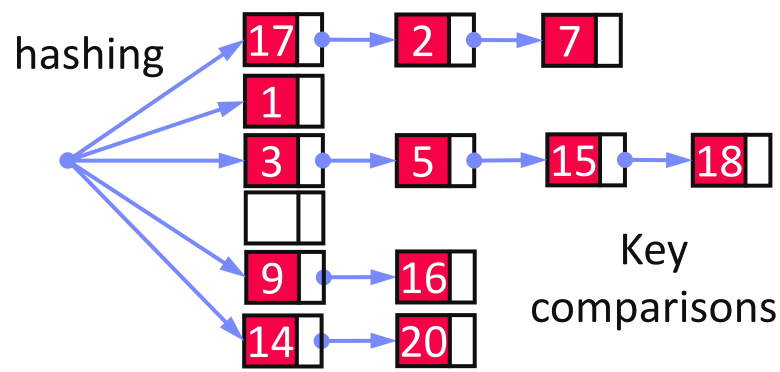


[William Pugh: Skip Lists: A Probabilistic Alternative to Balanced Trees. **CACM 1990**]

Hashing Overview

- **Static vs Dynamic Hashing**
  - Hash table of buckets B, compute h=hash(key), find bucket B[h mod |B|]
  - **Static:** pre-allocation of buckets, **over- and under-provisioning** (open addressing: linear probe, robin hood, cuckoo)
  - **Dynamic:** extend as needed (chained bucket, extendible, linear hashing)

- **Chained Bucket Hashing**
  - Handle hash collisions via overflow list of linked buckets
  - Reorganization if fill factor reached
  - **On disk:** buckets are pages



- **Common Hash Functions**
  - MurmurHash 2, MurmurHash 3, Jenkins, CRC
  - Google CityHash, Google FarmHash, Facebook XXHash3 (http://cyan4973.github.io/xxHash/)

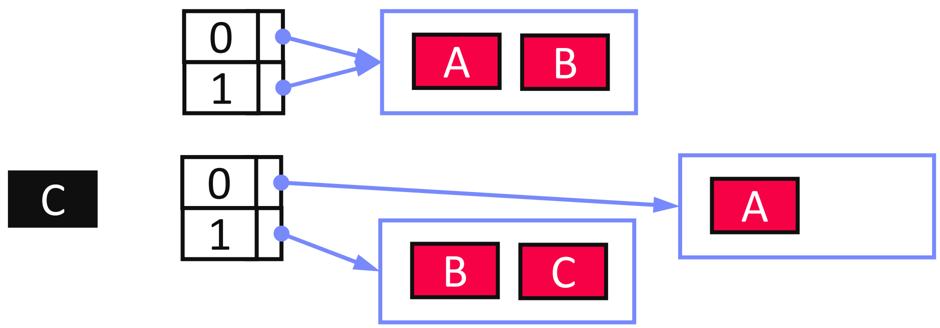
[Andy Palvo: Database Systems – Hash Tables, CMU Lecture, **2019**]

Extendible Hashing

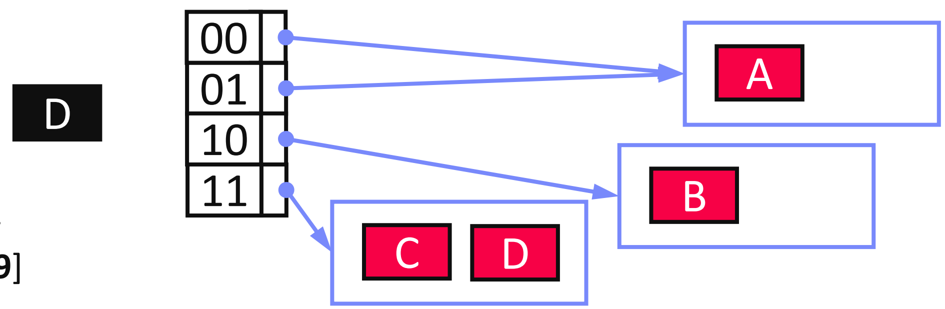
[Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, H. Raymond Strong: Extendible Hashing - A Fast Access Method for Dynamic Files. **TODS 4(3), 1979**]

- **Overview**
  - Dynamic resizing on demand, w/o rehashing/reassigning tuples to pages
  - h=hash(key), use **d bits** and **directory of 2^d entries** (with max table size, then bucket chaining)
  - Directory entries point to buckets, multiple refs to one bucket possible

- **Example d = 1**



- **Example d = 2**

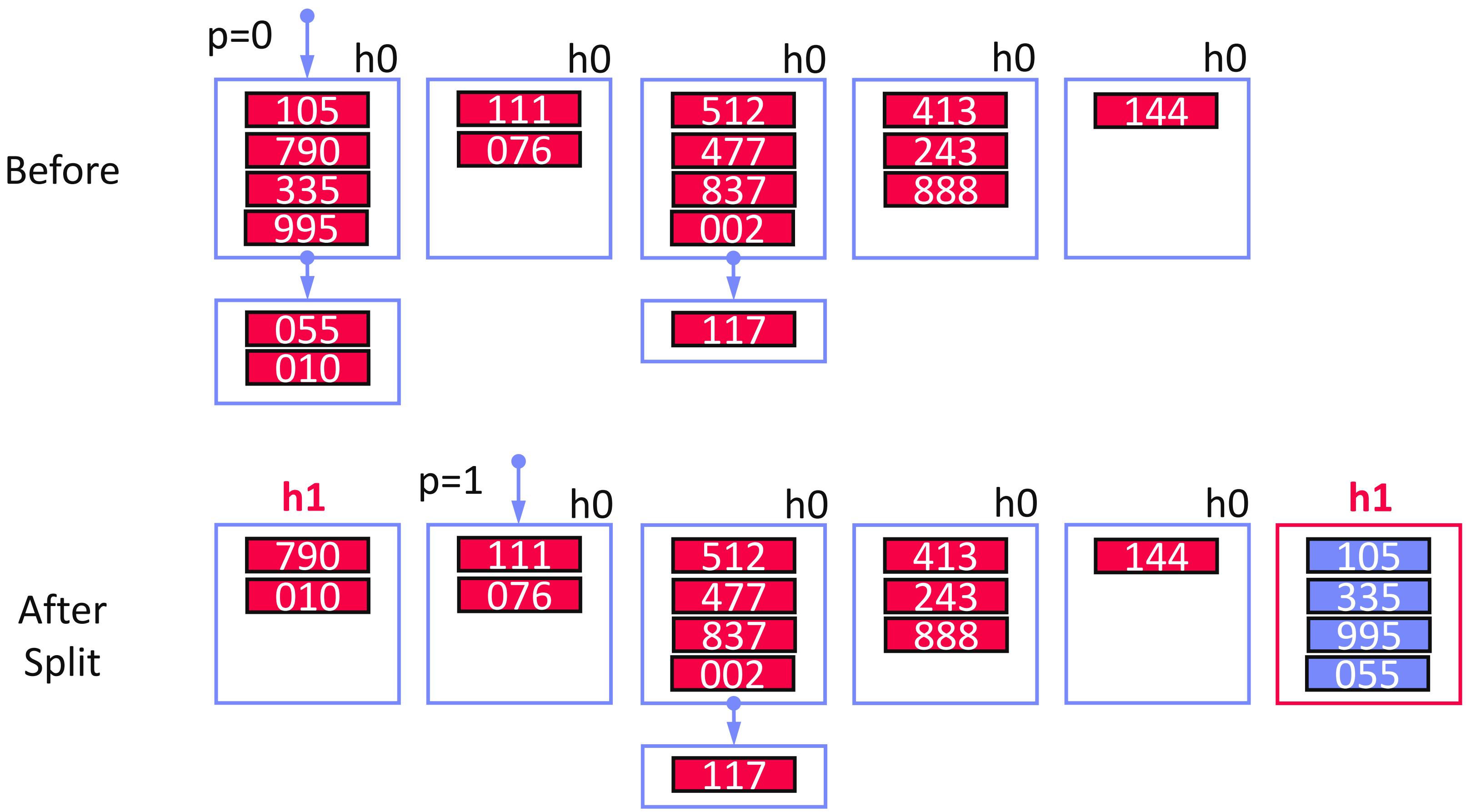


[Thomas Neumann: Datenbanksysteme und moderne CPU-Architekturen – Access Paths, TU Munich, **2019**]

Linear Hashing

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

- **Overview**
  - Improved Extensible Hashing scheme, w/o exponential directory growth
  - First start chaining, then incrementally **split individual buckets** (in order)

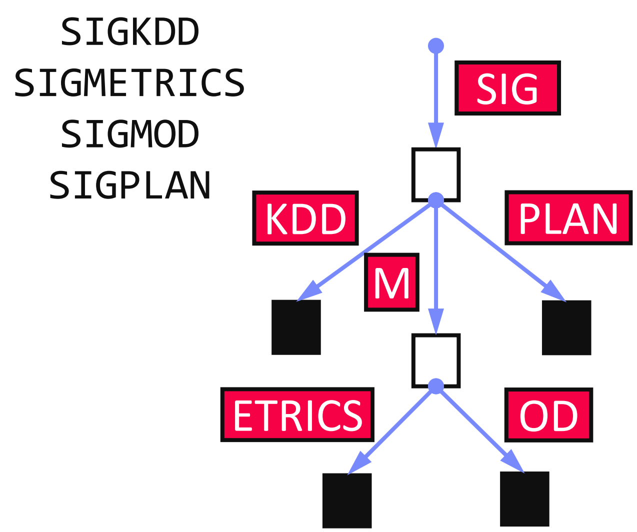


Overview Prefix Trees (Tries)

- **Overview**
  - From information re**trie**val, mostly for string indexing
  - Trie: “A tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes.” (NIST DADS)

- **PATRICIA Trie**
  - Extended binary (character-level) trie, with compressed substrings

[Donald R. Morrison: PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. **J. ACM 15(4) 1968**]



- **Variants**
  - Radix Tree, key alteration radix tree (Kart), digital search trees

Generalized Prefix Tree

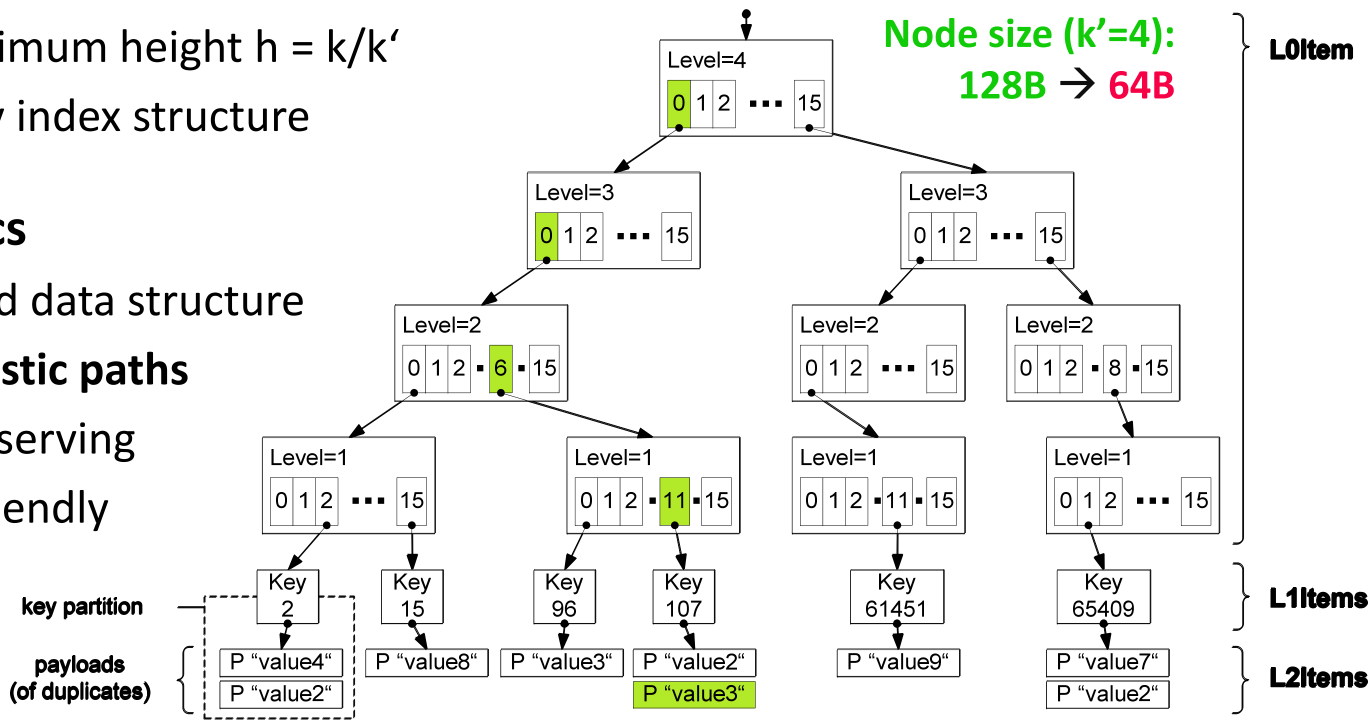
[Matthias Boehm et al: Efficient In-Memory Indexing with Generalized Prefix Trees. **BTW 2011**]

- **Generalized Prefix Tree (IXByte)**
  - **Arbitrary data types (byte sequences)**
  - **Variable prefix length k'**
  - Node size: s = $2^{k'}$ references
  - Fixed maximum height h = k/k'
  - Secondary index structure
- **Characteristics**
  - Partitioned data structure
  - **Deterministic paths**
  - Order-preserving
  - Update-friendly
- **Trie Expansion & Bypass**

INSERT key=107, payload="value3"

key = 107

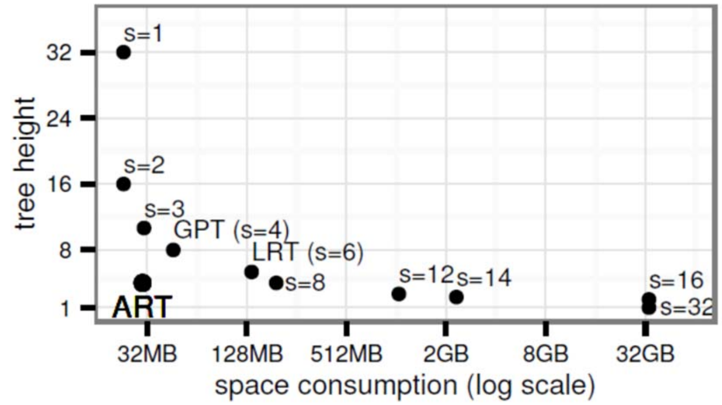
0000	0000	0110	1011
0	**0**	**6**	**11**



Adaptive Radix Trees

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

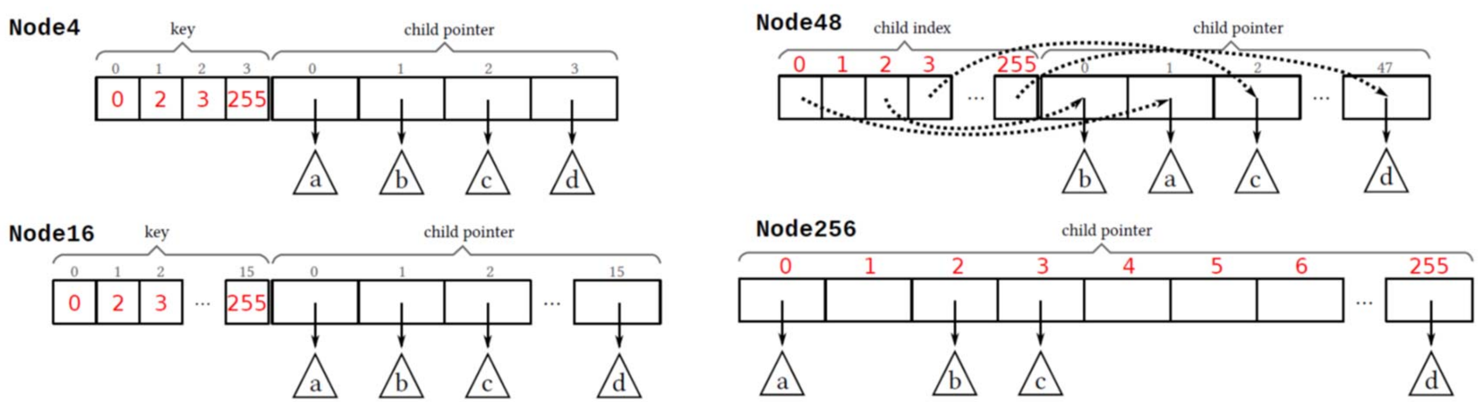
- **Motivation and Overview**
  - Small trie height/high fan-out, but with low space overhead
  - Prefix k'=8 → 256 children
  - Adaptive nodes 4, 16, 48, 256 entries
  - Lazy expansion and path compression



- **Node Types**

Linear/binary search for keys

256 element arrays of indexes / child pointers



Hybrid Prefix Trees

	Binary	B-Tree	CSB-Tree	Hash	T-Tree	Trie
Prefix Hash Tree ’70				X		X
Prefix B-Tree ’77		X				X
Ternary Search Tree ’97	X					X
Partial Keys ’01		X			X	X
Burst-Trie ’02	X	X	X	X	X	X
HAT-Trie ’07				X		X
J^+-Tree ’09		X			X	X
CS-Prefix Tree ’09			X			X
SuRF ’18				X		X

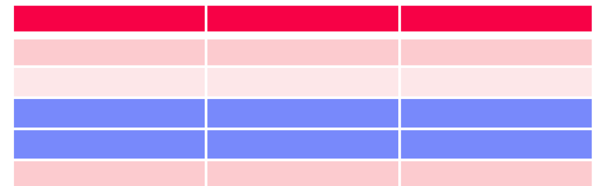
Partitioning and Pruning

Coarse-grained Table Partitioning

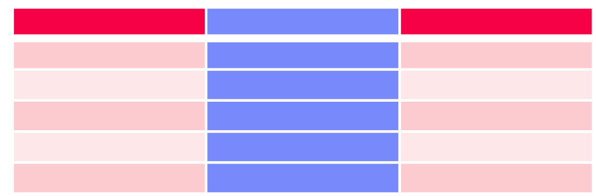
Fine-grained Physical Partitioning and Sketching

Overview Partitioning Strategies

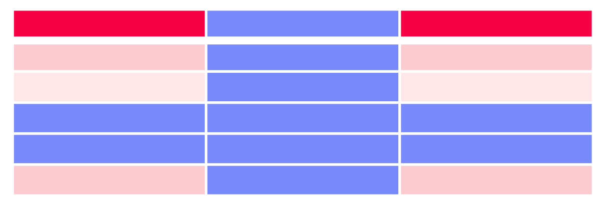
- **Horizontal Partitioning**
  - Relation partitioning into disjoint subsets



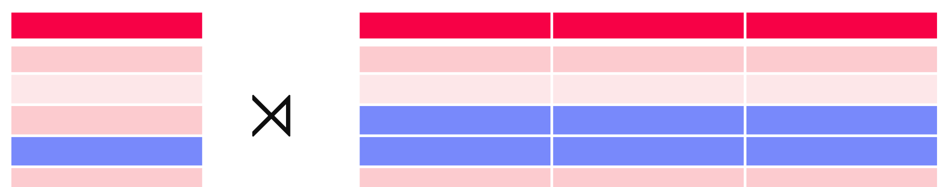
- **Vertical Partitioning**
  - Partitioning of attributes with similar access pattern



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



- **Derived Horizontal Partitioning**



- **Physical Partitioning Schemes**
  - Hash Partitioning, Round-Robin, Radix Partitioning, etc

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

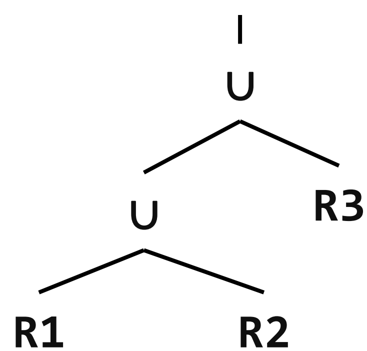
Horizontal Partitioning

- **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

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

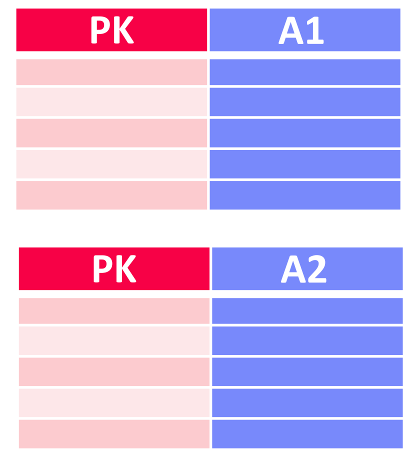
- **Reconstruction**
  - **Union** of all fragments
  - Bag semantics, but no duplicates across partitions



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

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

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

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

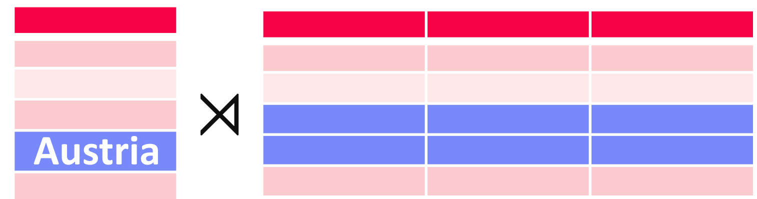
$$R = R_1 \bowtie R_i \bowtie R_n \quad (1 \le i \le n)$$

- **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

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

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)**

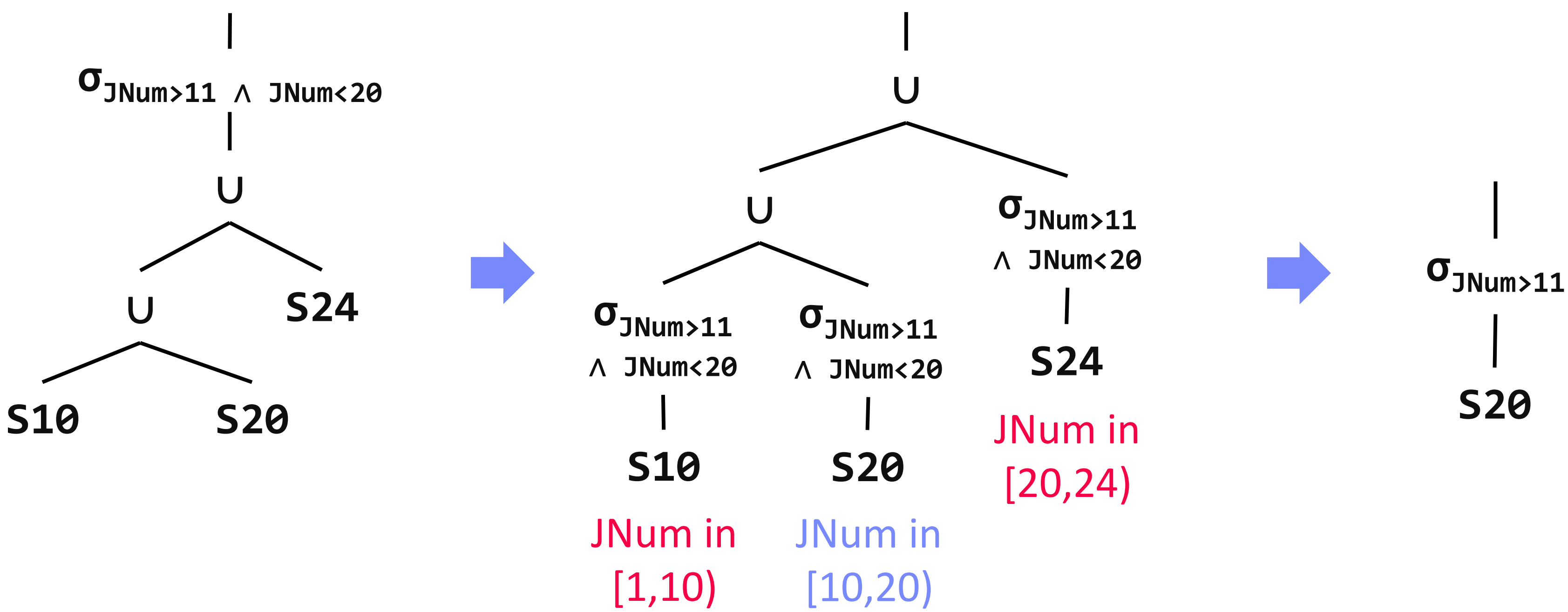
```
CREATE TABLE Squad(
   JNum INT PRIMARY KEY,
   Pos CHAR(2) NOT NULL,
   Name VARCHAR(256)
) PARTITION BY RANGE(JNum);

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.**

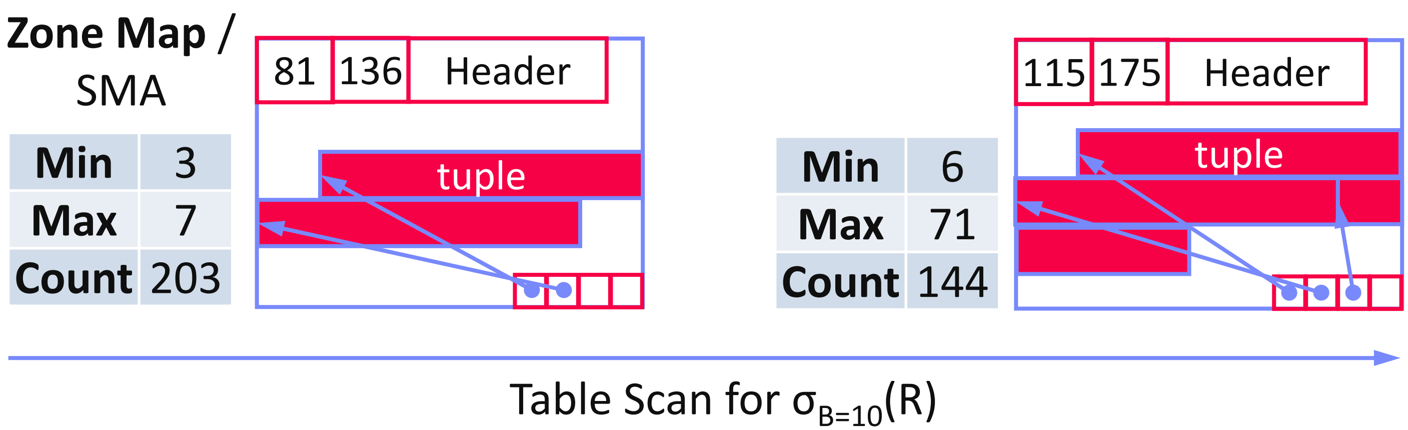
```
SELECT * FROM Squad
   WHERE JNum > 11 AND JNum < 20
```



Zone Maps

[Guido Moerkotte: Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. **VLDB 1998**]

- **Small Materialized Aggregates (SMA)**
  - Data stored in zones (pages, blocks, or partitions)
  - Maintain SMA (e.g., min, max, count, sum) as **summary per zone**
  - Global vs local storage, eager vs lazy maintenance on updates

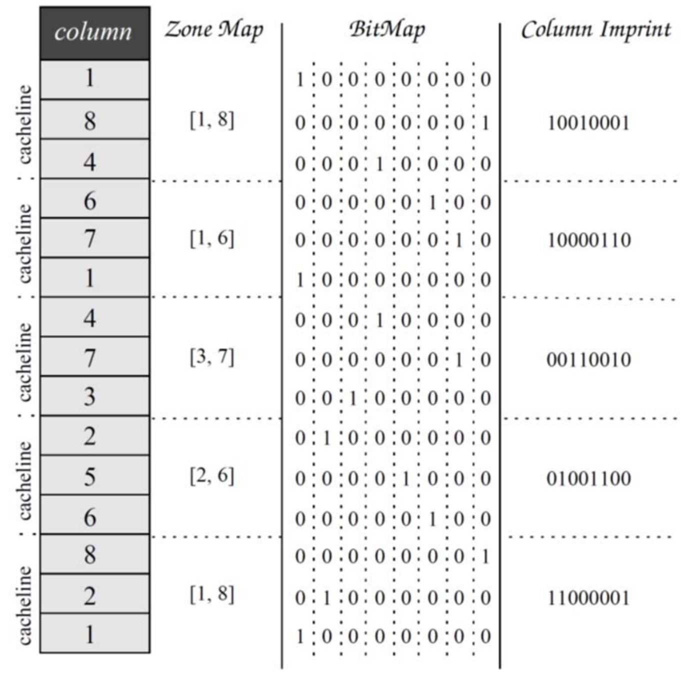


- **Query Processing**
  - Partition pruning for selection predicates
  - Precomputed partial aggregates (see materialized views)

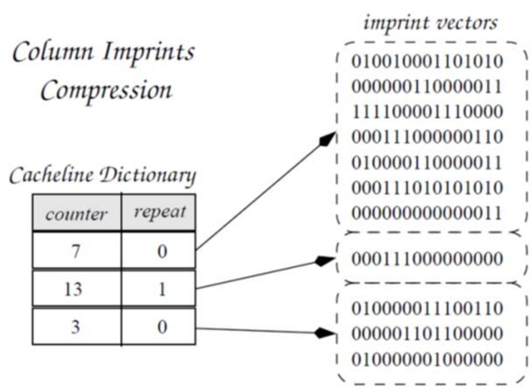
Column Imprints

[Lefteris Sidirourgos, Martin L. Kersten: Column imprints: a secondary index structure. **SIGMOD 2013**]

- **Column Imprints**
  - Zone = cache line (64 Byte)
  - Column imprint = union of one-hot vectors
  - Sampled histogram → bins (**max 64** bins)



- **Compression**
  - CL Dictionary (next x CLs, repeat flag)



- **Query Processing**
  - Cacheline pruning for selection predicates (point, range)
  - `imprint & predicate` (predicate w/ potentially many bits for ranges)

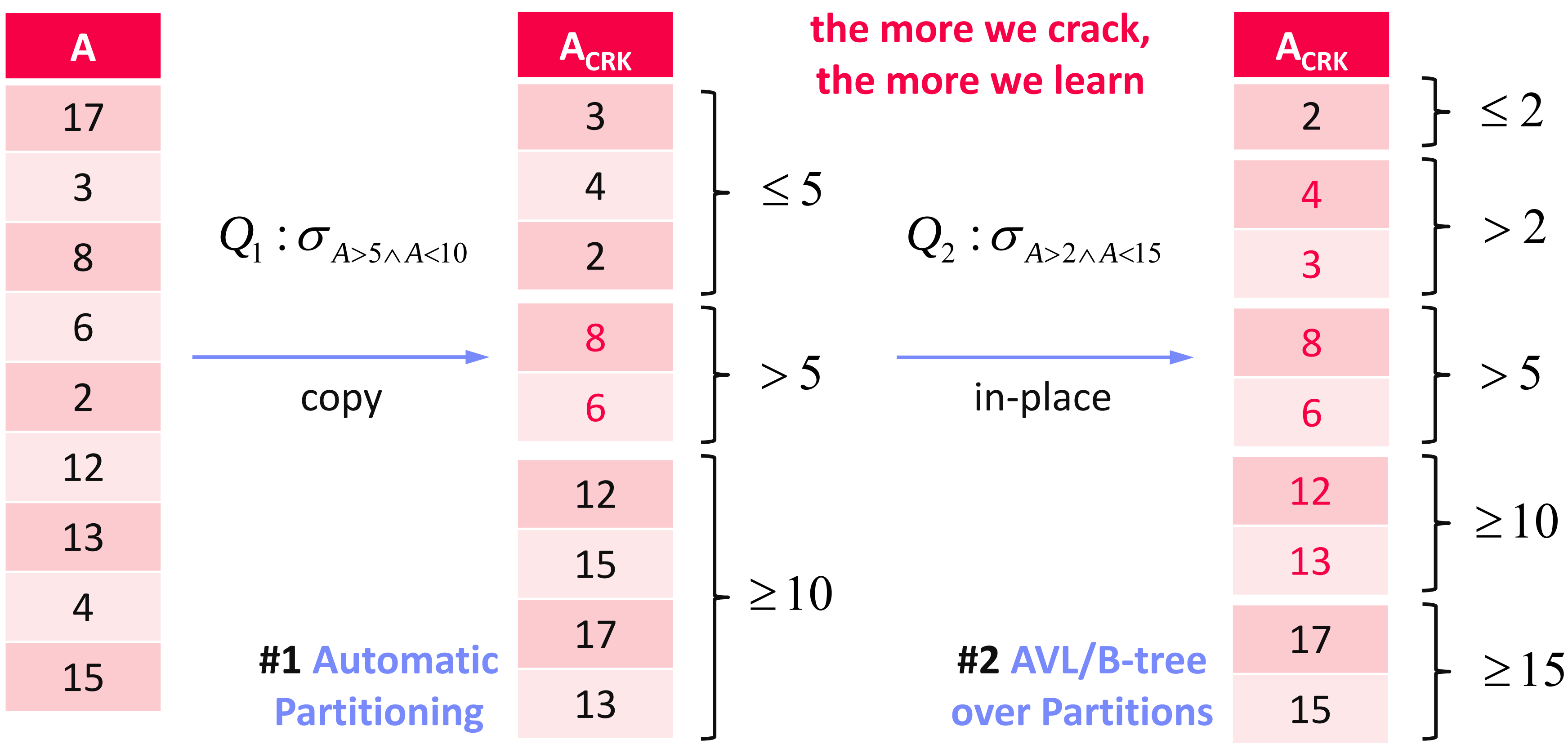
Adaptive and Learned Access Methods

Database Cracking

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

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

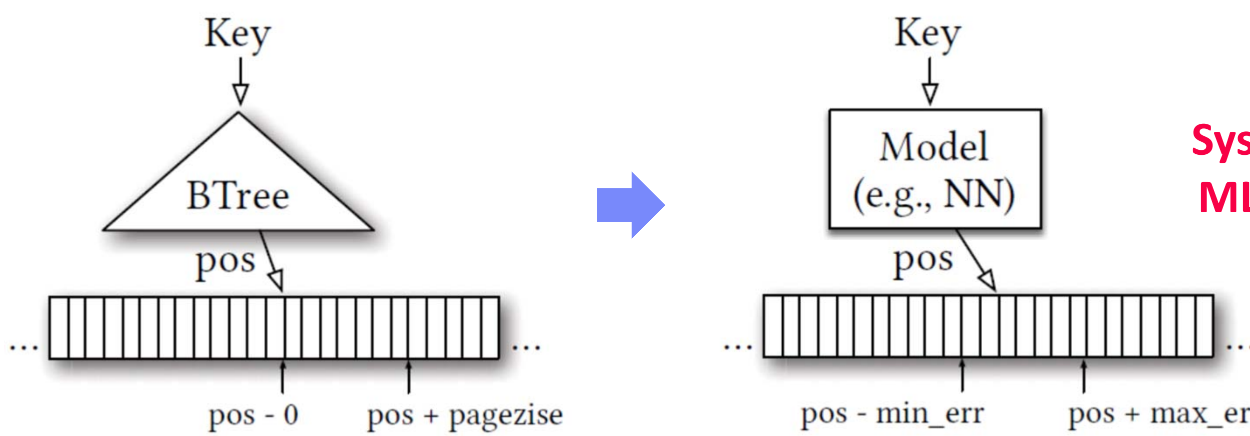
[Stratos Idreos, Martin L. Kersten, Stefan Manegold: Database Cracking. **CIDR 2007**]



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

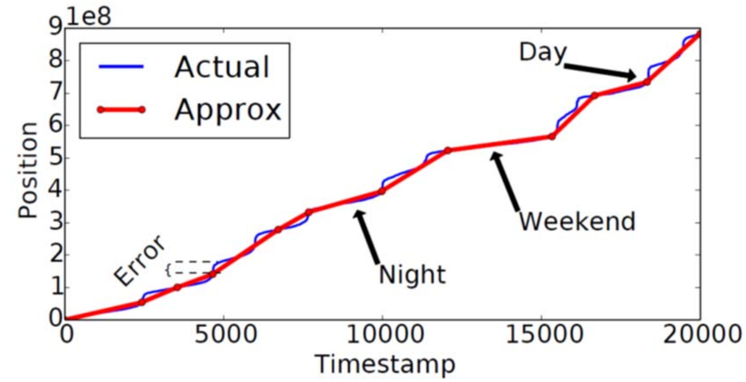
- **Follow-up Work on SageDBMS**

[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**]

Learned Index Structures, cont.

- **FITing-Tree**
  - Adapt to underlying data and patterns
  - Piecewise linear functions
  - Maximum pos error guarantees
  - Segment pages w/ free space

[Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, Tim Kraska: FITing-Tree: A Data-aware Index Structure. **SIGMOD 2019**]



- **PGM-index**
  - Piecewise geometric model index
  - Recursive, compressed segment tree

[Paolo Ferragina, Giorgio Vinciguerra: The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. **PVLDB 13(8) 2020**]

- **RadixSpline**
  - Lookup table to spline points, selected w/ max error guarantee

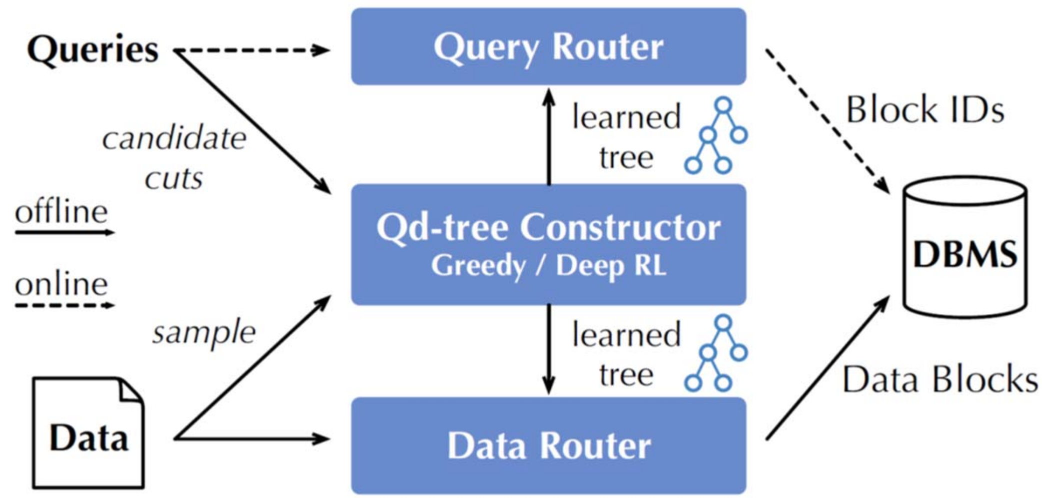
[Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, Thomas Neumann: RadixSpline: a single-pass learned index. **aiDM@SIGMOD 2020**]

Learned Partitioning Schemes

- **Query-Data Routing Tree (qd-Tree)**
  - Binary decision tree, with data blocks at leaf nodes (min size constraint)
  - Given dataset, and workload, find tree that minimized number of accessed tuples
  - Deep reinforcement learning

[Zongheng Yang el al: Qd-tree: Learning Data Layouts for Big Data Analytics. **SIGMOD 2020**]

- **Query Processing**
  - Get list of blocks that need to be evaluated



Summary and Q&A

- **Overview Access Methods**
- **Index Structures**
- **Partitioning and Pruning**
- **Adaptive and Learned Access Methods**

- **Programming Projects**
  - Initial test suite, benchmark, make file, and reference implementation
  - Start **your own implementation** in next weeks

- **Next Lectures** (Part A)
  - 05 **Compression Techniques** [Nov 04]