

Programmierpraktikum: Datensysteme

02 Background Index Structures

Prof. Dr. Matthias Boehm

Technische Universität Berlin

Berlin Institute for the Foundations of Learning and Data

Big Data Engineering (DAMS Lab)

Announcements / Administrative Items

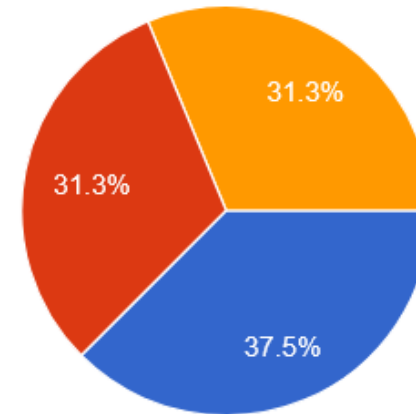


■ #1 Video Recording

- Hybrid lectures: in-person H 0111, zoom live streaming, video recording
- <https://tu-berlin.zoom.us/j/9529634787?pwd=R1ZsN1M3SC9BOU1OcFdmem9zT202UT09>

■ #2 Team Selection

- **Oct 26** team building/selection, afterwards assignment → 7 (5) DAMS
- Assignment: grouping of self-assignments
(2 **Java** groups a 2/3 students, 1 preassembled)



- #1 Transactional In-memory Indexing (DAMS)
- #2 Duplicate Detection (D2IP)
- #3 Provenance Tracking in ML Pipelines (DEEM)

Agenda



- Overview Access Methods
- Index Structures
- Learned Indexing

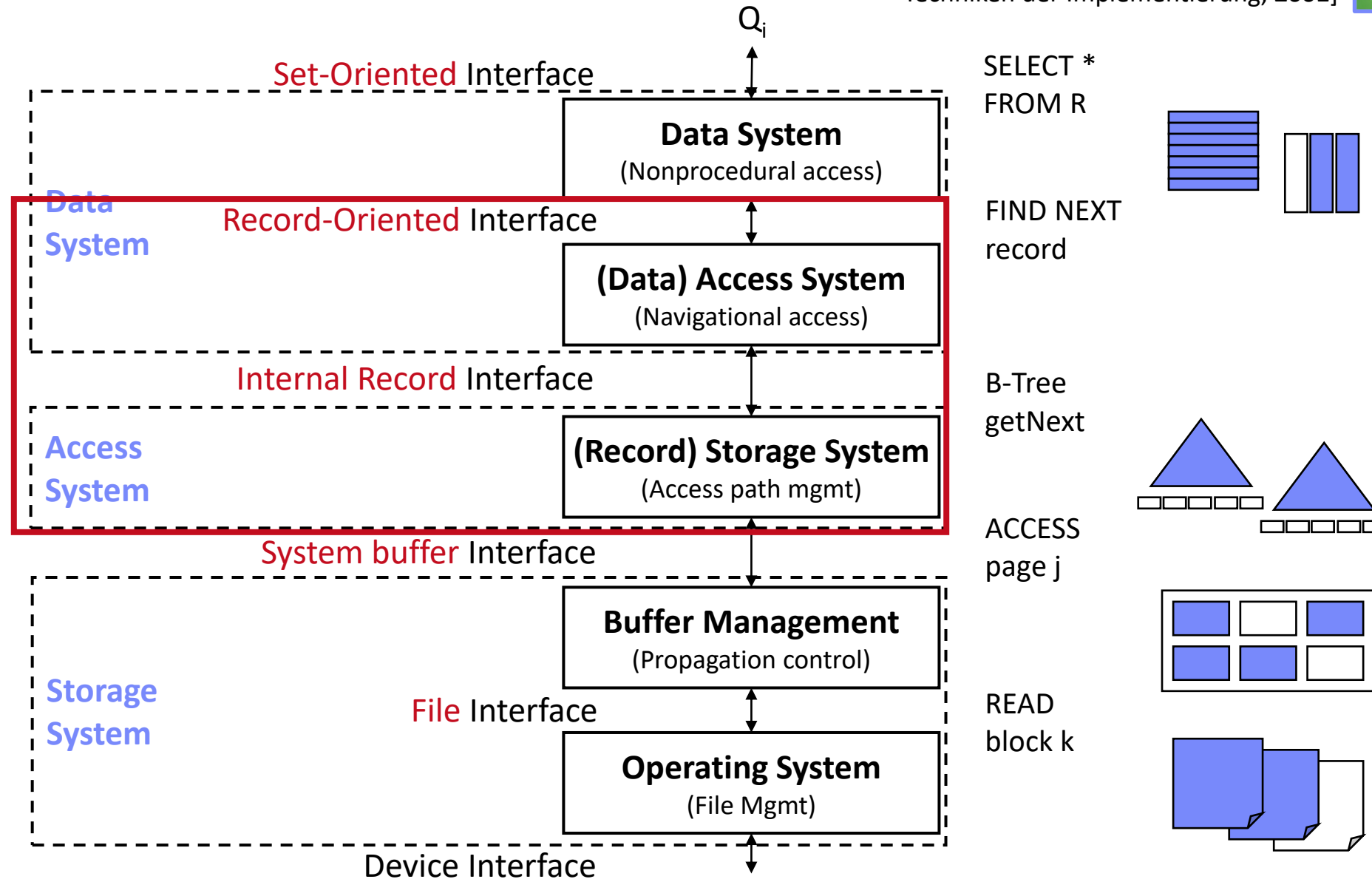


**How to implement your own
custom Index Structure**

Overview Access Methods

DBMS System Architecture

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

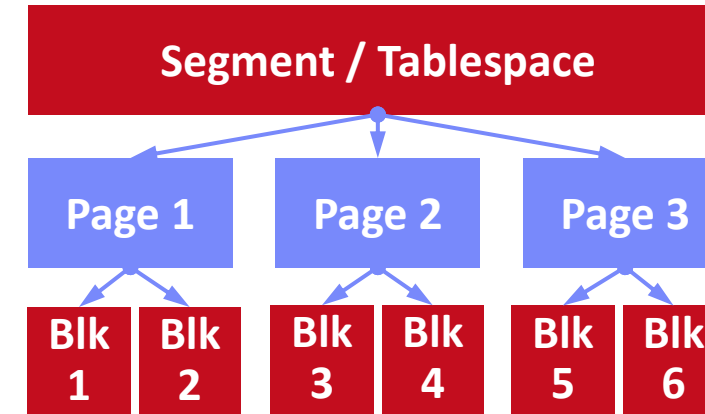


Background Storage System



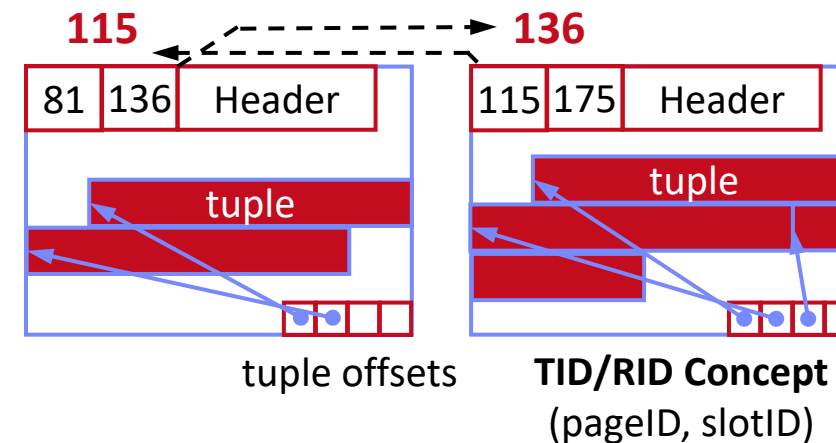
■ Segments, Pages, Blocks

- **Segments:** storage unit of DB objects like relations (heap) and indexes
- **Page:** fixed-size memory region
- **Block:** smallest addressable unit on disk (e.g., POSIX block devices)



■ Buffer & Storage Management

- Buffer management at granularity of **pages**
- PostgreSQL default: **8KB**
- Different table/page layouts (e.g., **NSM**, DSM, PAX, column)
- **TID/RID Concept** (pageID, slotID)
→ stable ID, even if **records reorganized**



■ TID Concept (p, s)

- TID := (page number, slot index)
- Page slot directory holds tuple offsets (byte position) within page

■ Example PostgreSQL

- Papers(PKey, Title, Pages, CKey, JKey)
- Hidden CTID system column
(not shown on *, but usable)

```
SELECT CTID, PKey,  
       Title, Pages  
FROM Papers
```

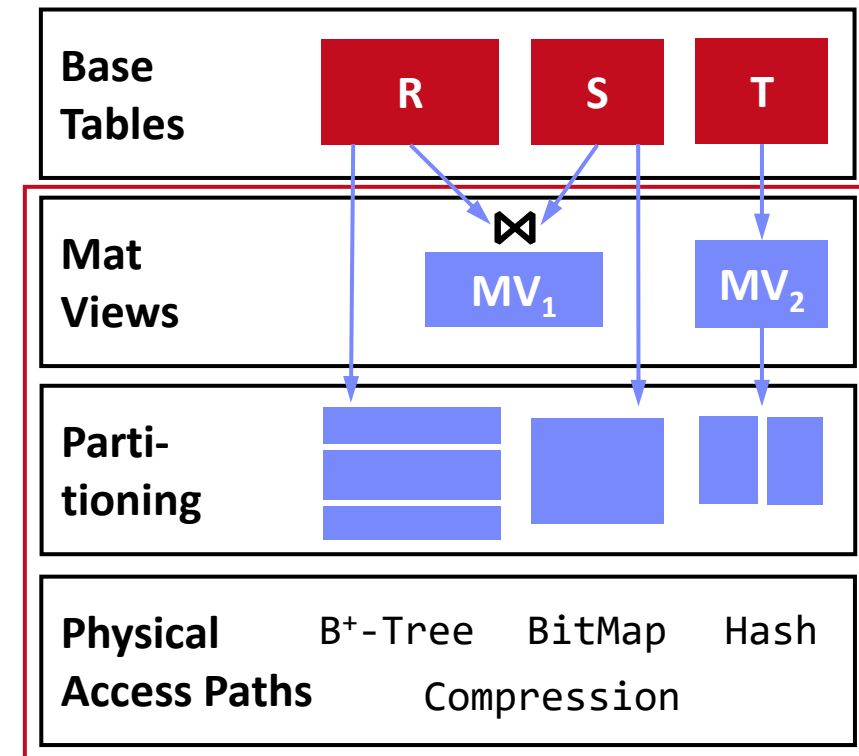
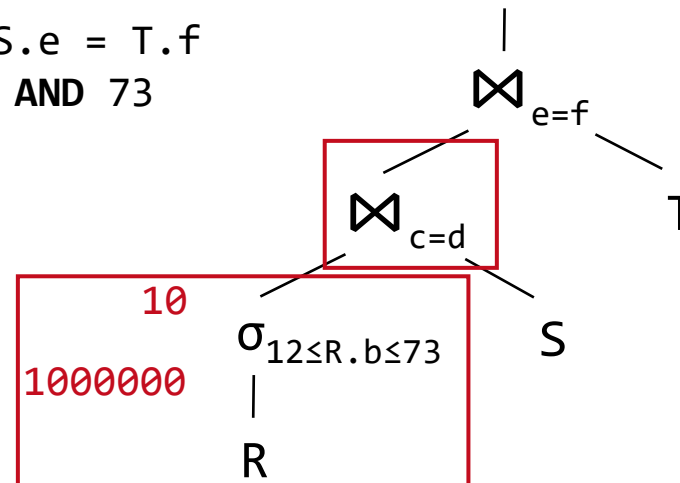
	ctid tid	pkey integer	title character varying (512)	pages character va
5681	(78,21)	731118	MV-IDX: Multi-Version Index in Action	671-674
5682	(78,22)	731121	Hochperformante Analyse von Graph-Dat...	311-330
5683	(78,23)	731122	SPARQling Pig - Processing Linked Data wi...	279-298
5684	(78,24)	731123	RelaX: A Webbased Execution and Learnin...	503-506
5685	(78,25)	731129	Efficient In-Memory Indexing with General...	227-246
5686	(78,26)	731130	Datensicherheit in mandantenfähigen Clo...	477-489
5687	(78,27)	731131	In-Database Machine Learning: Gradient ...	247-266
5688	(78,28)	731133	FlexY: Flexible; datengetriebene Prozessm...	503-506
5689	(78,29)	731134	Extending the MPSM Join	57-71
5690	(78,30)	731137	Orthogonal key-value locking	237-256

■ Performance Tuning via Physical Design

- Select physical data structures for relational schema and query workload
- #1: User-level, **manual physical design** by DBA
- #2: User/system-level **automatic physical design**

■ Example

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

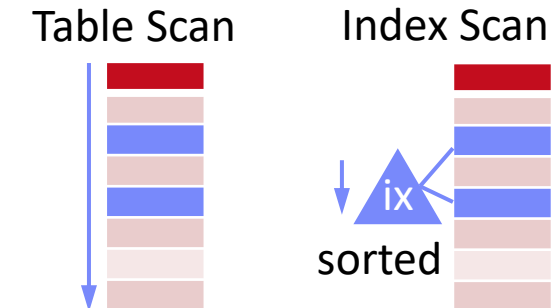


Overview Index Structures



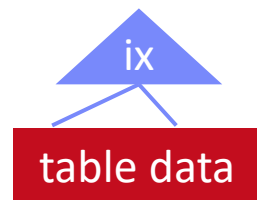
Table Scan vs Index Scan

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

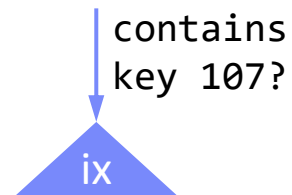


Use Cases for Indexes

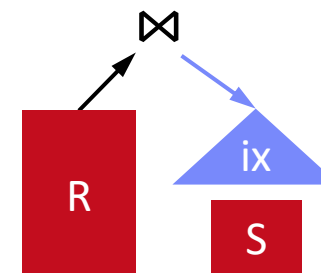
Lookups / Range Scans



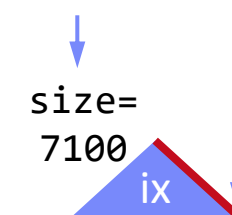
Unique Constraints



Index Nested Loop Joins



Aggregates (count, min/max)



Additional Terminology



■ Create Index

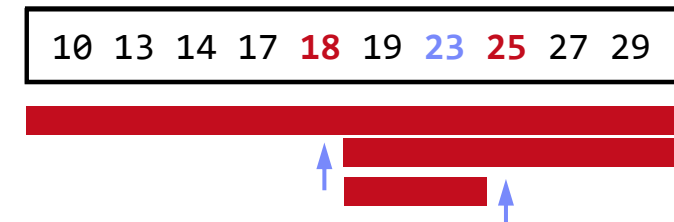
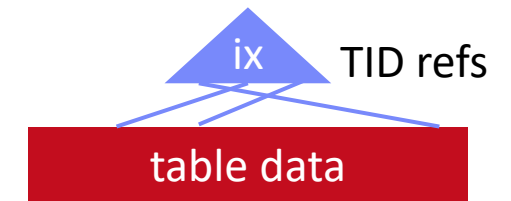
- Create a secondary (non-clustered) index on a set of attributes
- **Clustered**: tuples sorted by index
- **Non-clustered**: sorted attribute with tuple references
- Can specify uniqueness, order, and indexing method
- **PostgreSQL methods**: btree, hash, gist, and gin

■ 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)`)

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

```
DROP INDEX ixStudLname;
```



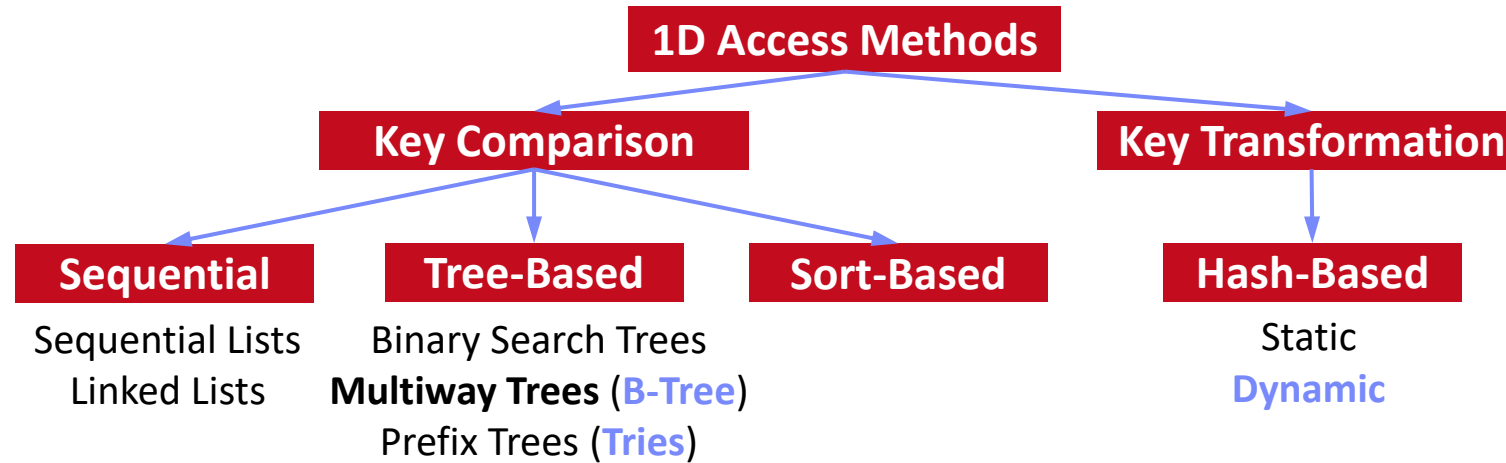
Index Structures

Classification of Index Structures

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



■ 1D Access Methods



■ 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)

History B-Tree

- Bayer 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)

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

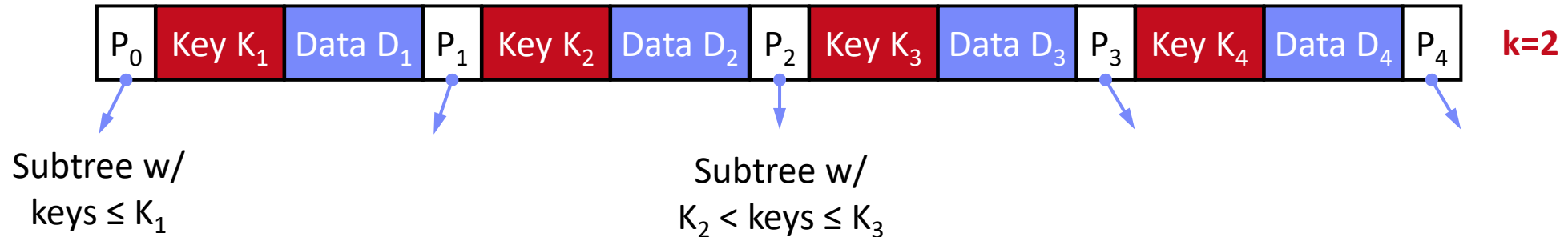


Definition B-Tree (k, h)

- 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)

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

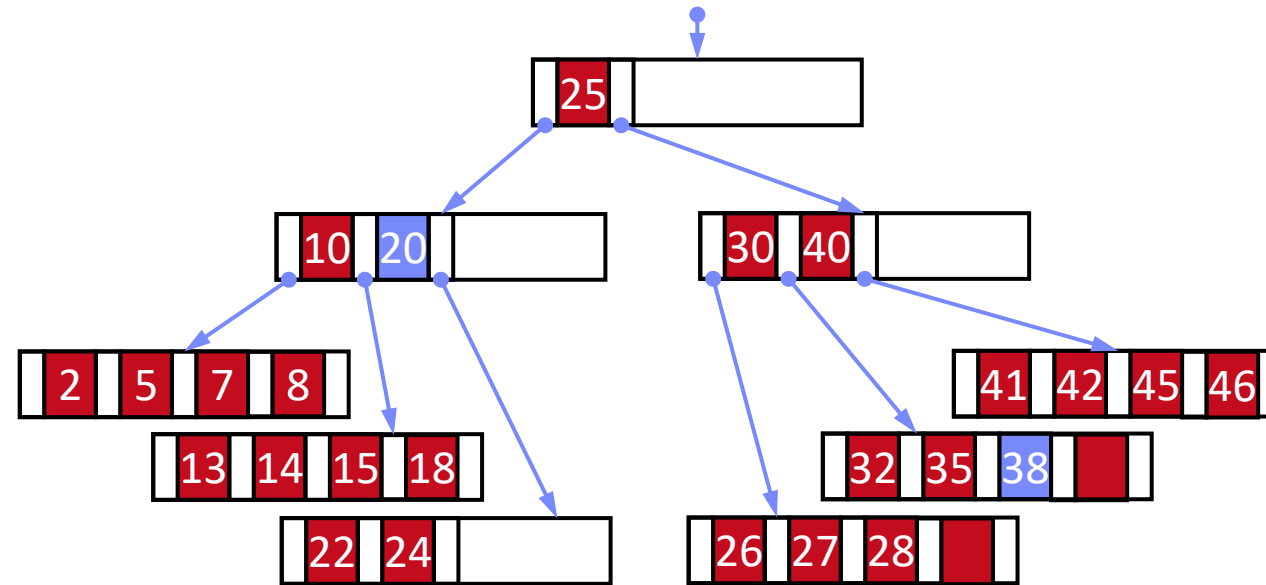
} All nodes adhere
to max constraints



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 P_i with $K_i < Q_k \leq K_{i+1}$

Range Scan $Q_L < K < Q_U$

- Lookup Q_L and call next K while $K < Q_U$ (keep current position and node stack)

B-Tree Insert



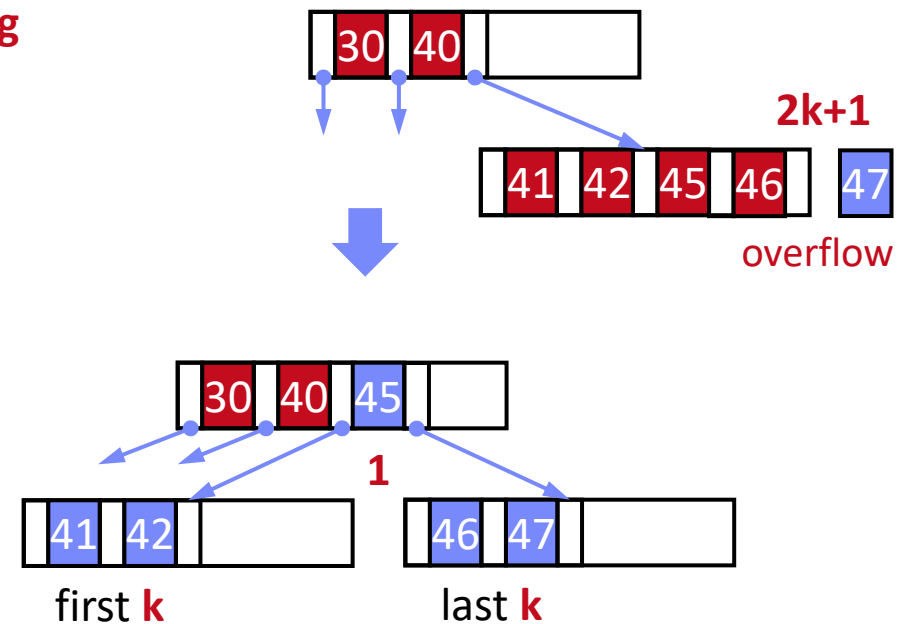
Basic Insertion Approach

- **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/ $k=1$)



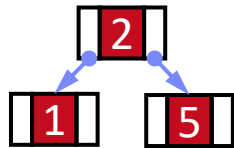
- Insert 1



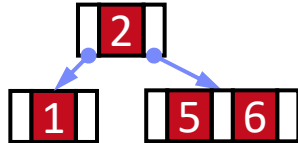
- Insert 5



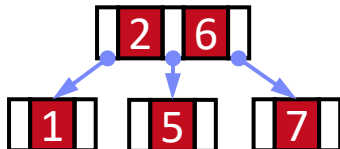
- Insert 2
(split)



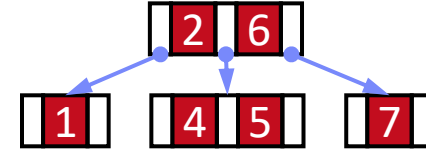
- Insert 6



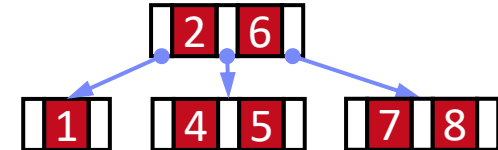
- Insert 7
(split)



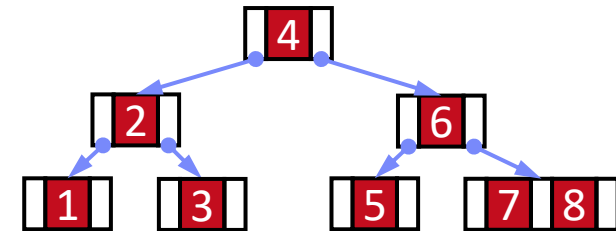
- Insert 4



- Insert 8



- Insert 3
(2x split)



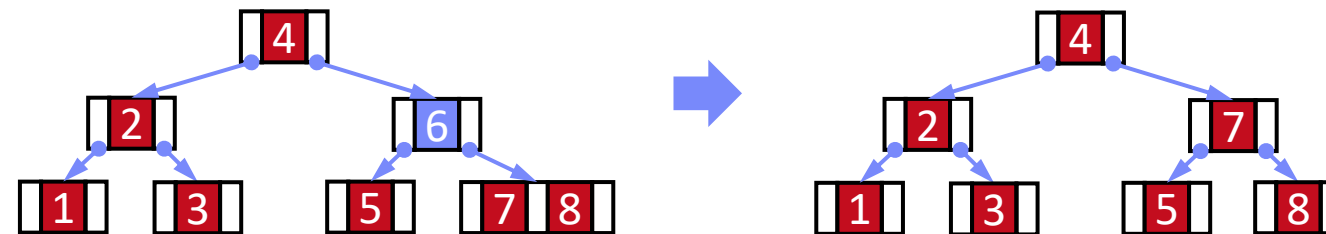
B-Tree Delete

Basic Deletion Approach

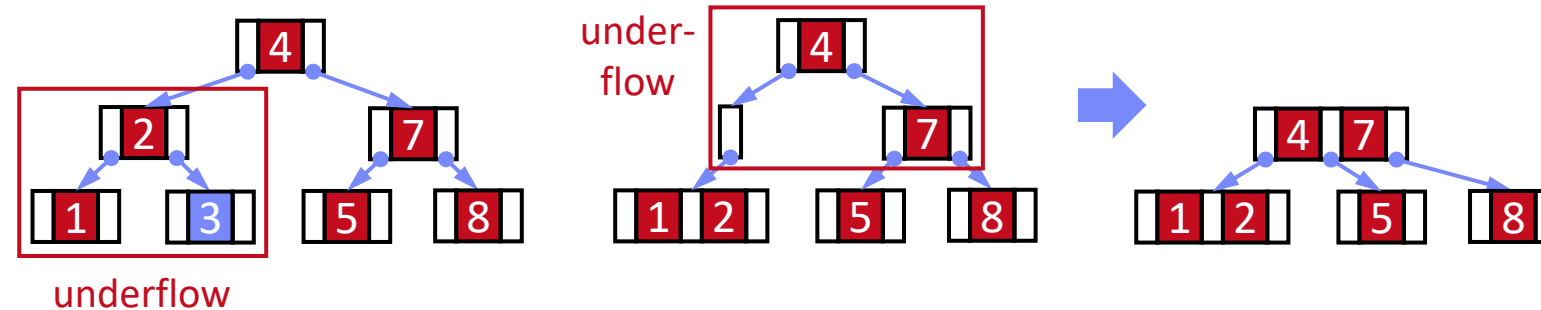
- Lookup deletion key, abort if non-existing
- Case inner node: **move entry** from fullest successor node into position
- Case leaf node: if underflows ($< k$ entries) **merge w/ sibling**

Example

- Case inner



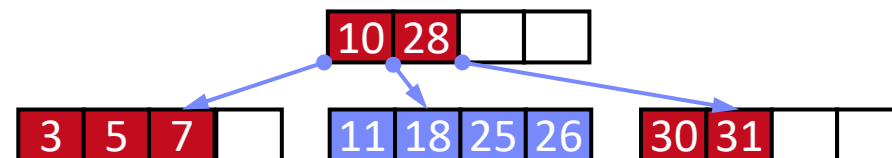
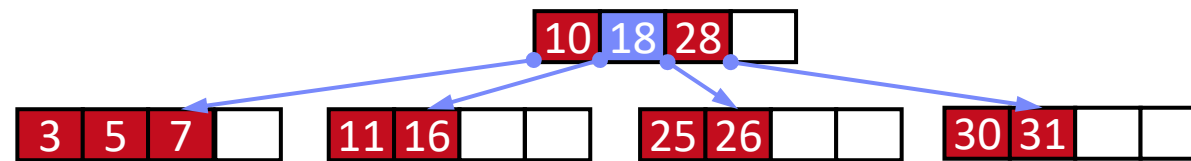
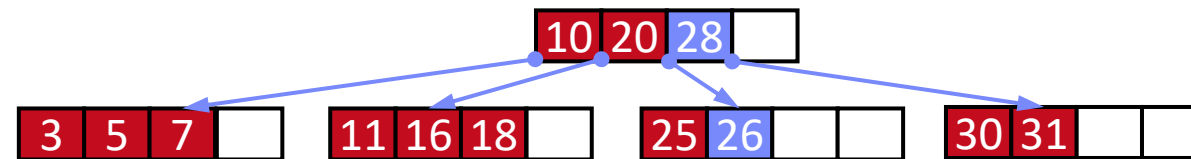
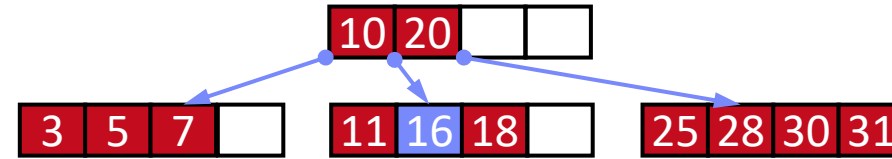
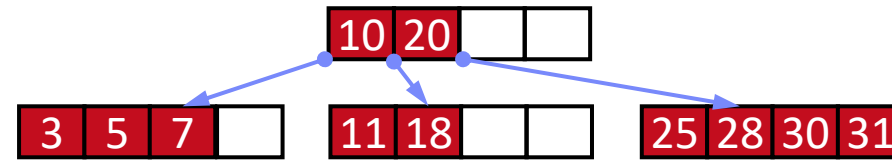
- Case leaf



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)

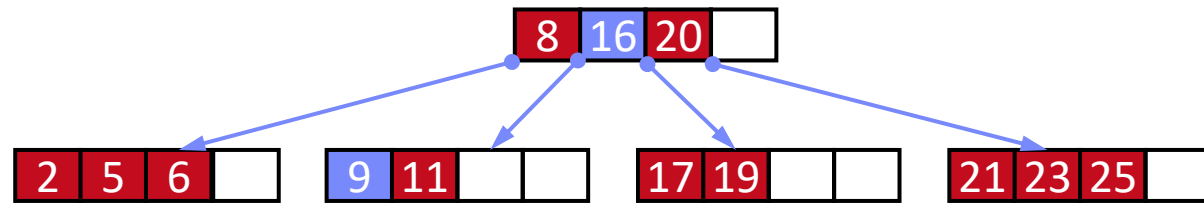
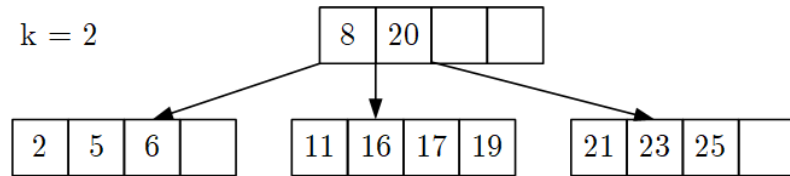
■ Duplicate Keys

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

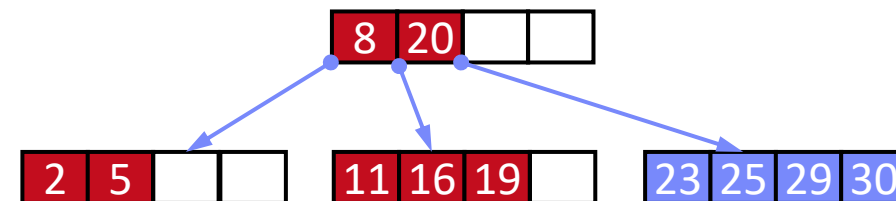
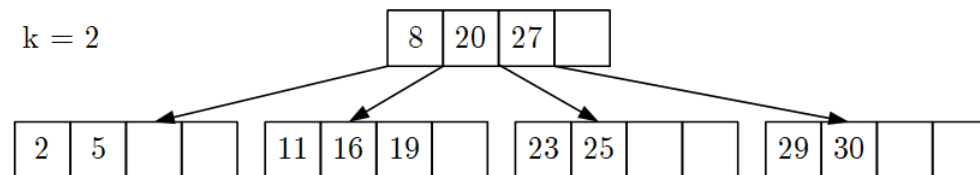
TEST YOURSELF: B-tree Understanding



- Given B-tree below, **insert key 9** and draw resulting B-tree (5/100 points)



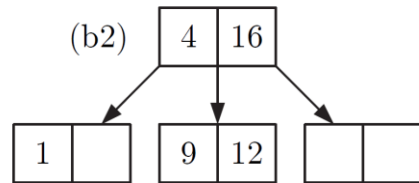
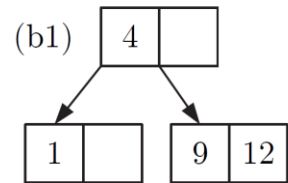
- Given B-tree below, **delete key 27**, and draw resulting B-tree (5/100 points)



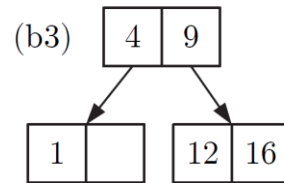
TEST YOURSELF: B-tree Understanding, cont.



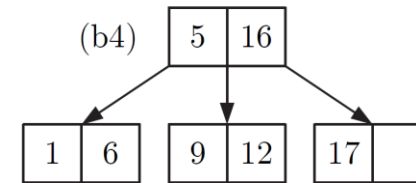
- Which of the following trees are valid – i.e., satisfy the constraints of – B-trees with $k=1$? Mark each tree as valid or invalid and name the violations (4/100 points)



(empty leaf node,
underflow)



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



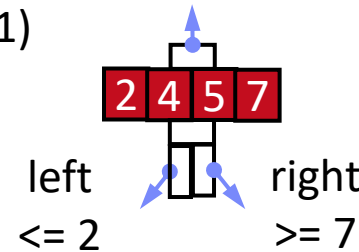
(**invalid ordering** of
data items, $6 > 5$ but
in left subtree)

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]

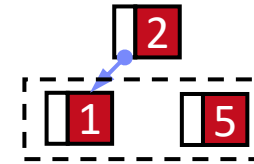


[Tobin J. Lehman, Michael J. Carey: A Study of Index Structures for Main Memory Database Management Systems. VLDB 1986]



CSB⁺-Tree

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

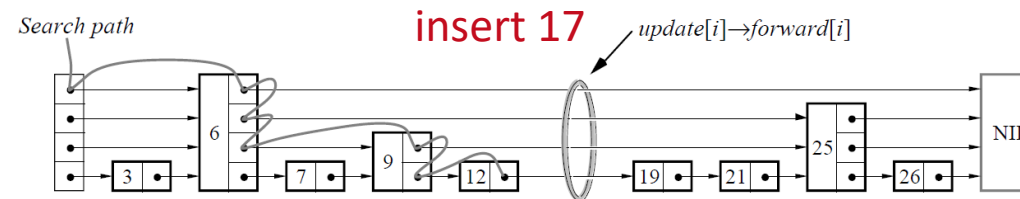


[Jun Rao, Kenneth A. Ross: Making B+-Trees Cache Conscious in Main Memory. SIGMOD 2000]



Skip Lists

- Linked list w/ multiple levels
- Fractions p w/ pointers



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

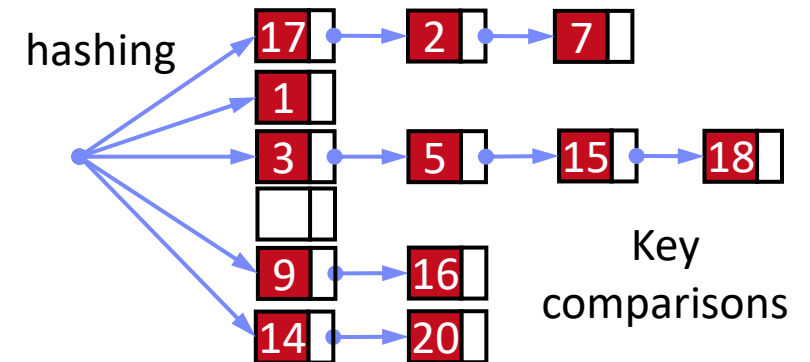


■ Static vs Dynamic Hashing

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

■ 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

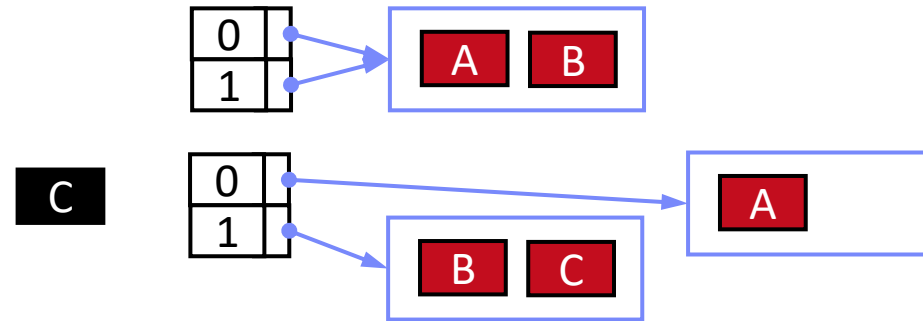
[Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, H. Raymond Strong: Extendible Hashing - A Fast Access Method for Dynamic Files. **TODS 4(3), 1979**]



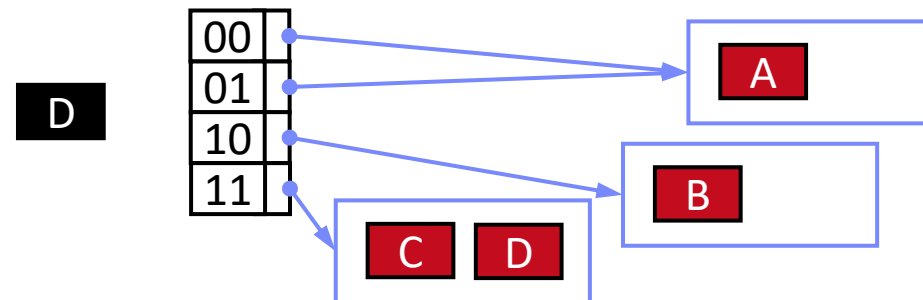
Overview

- Dynamic resizing on demand, w/o rehashing/reassigning tuples to pages
- $h = \text{hash}(\text{key})$, 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**]

Overview Prefix Trees (Tries)



■ Overview

- From information retrieval, 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

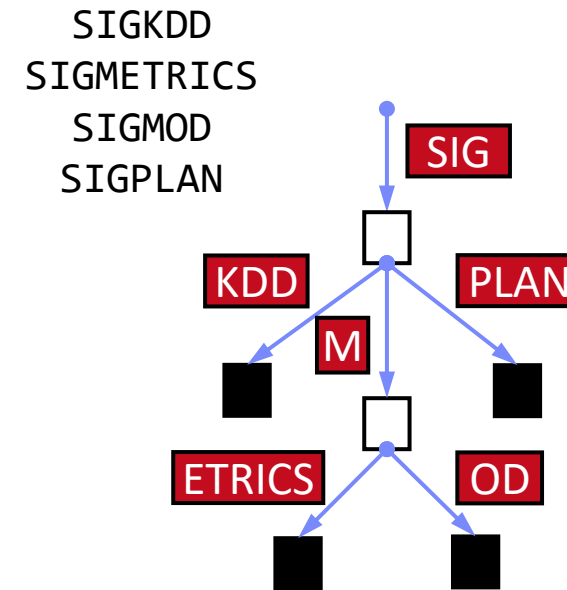
- Extended binary (character-level) trie, with compressed substrings



[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

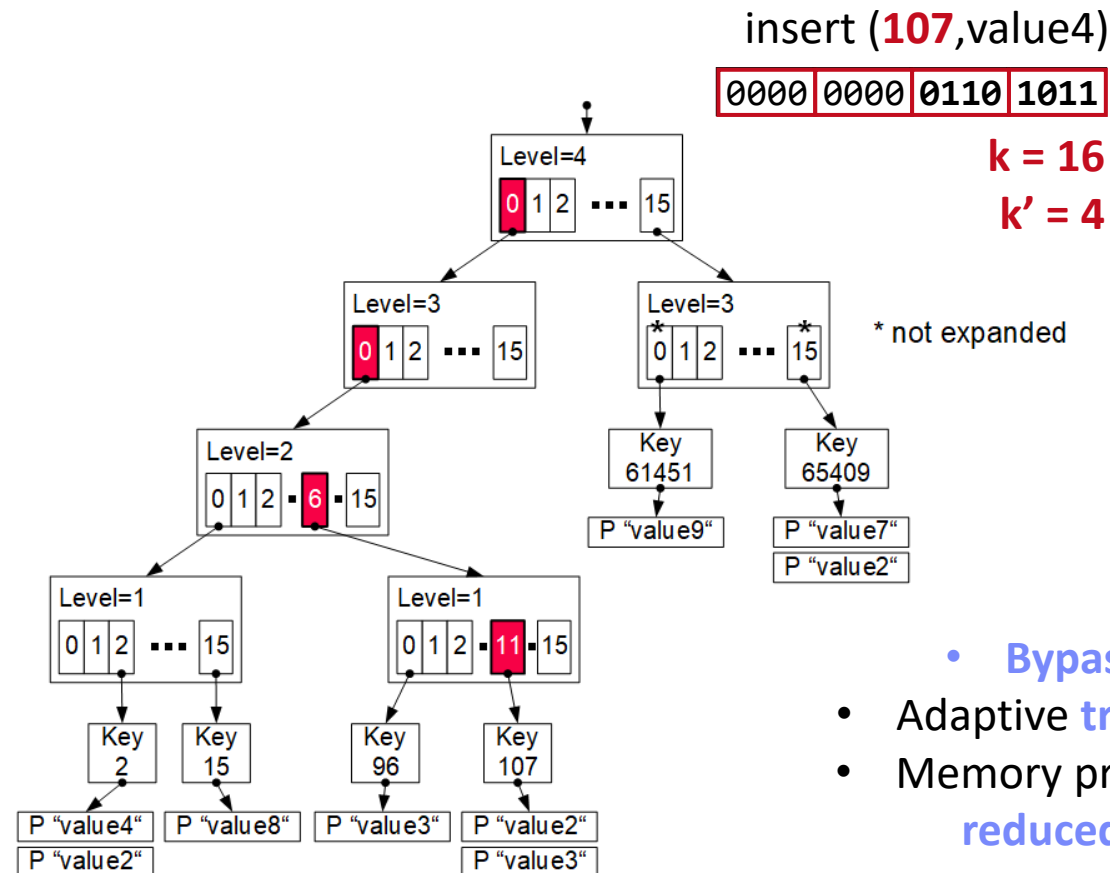
- Arbitrary data types (byte sequences)
- Configurable prefix length k'
- Node size: $s = 2^{k'}$ references
- Fixed maximum height $h = k/k'$
- Secondary index structure

Characteristics

- Partitioned data structure
- Order-preserving (for range scans)
- Update-friendly

Properties

- Deterministic paths
- Worst-case complexity $O(h)$



Node size ($k'=4$):
128B $\rightarrow$ 64B

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

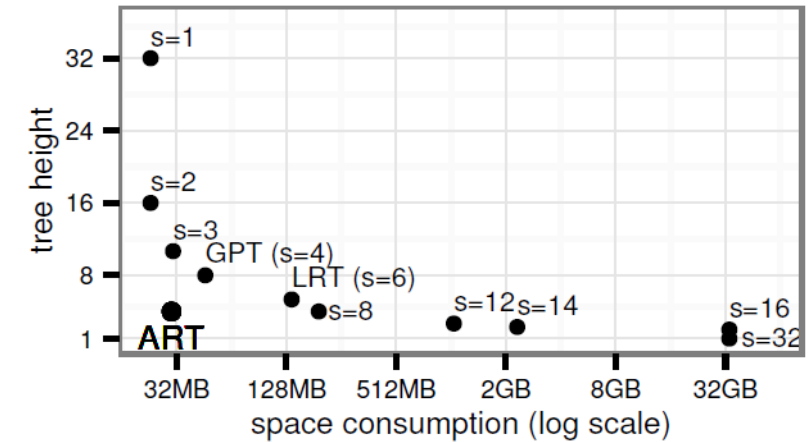
Adaptive Radix Trees

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



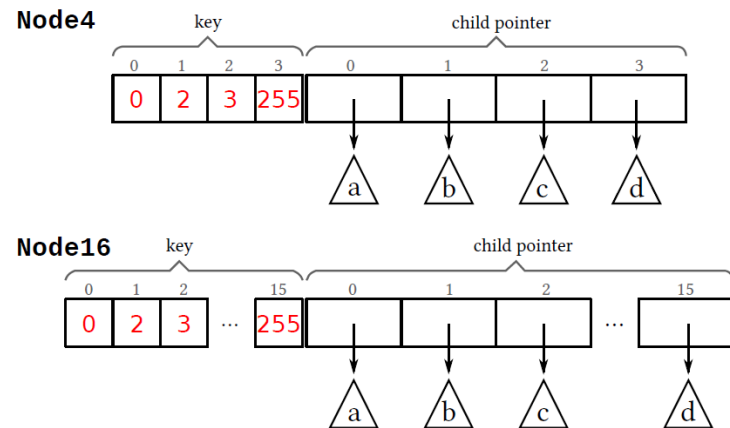
■ Motivation and Overview

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

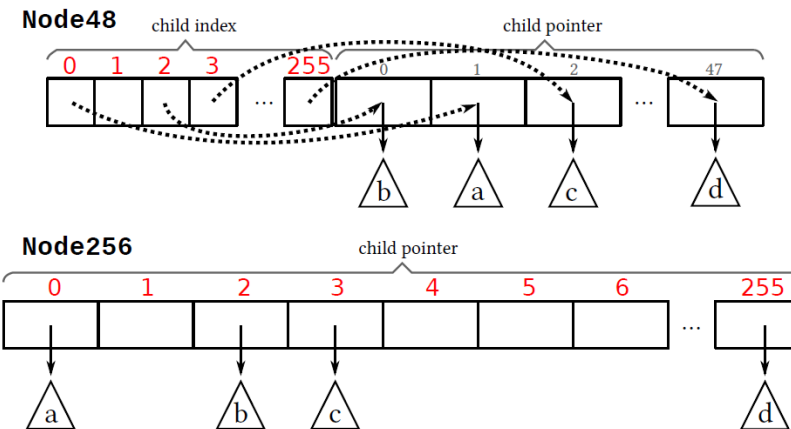


■ Node Types

Linear/binary
search for keys



256 element arrays of
indexes / child pointers



Hybrid Prefix Trees



	Binary	B-Tree	CSB-Tree	Hash	T-Tree	Skip List	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
SkipTrie '13						X	X
SuRF '18				X			X

Log-structured Merge Trees (LSM)

[Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O'Neil: The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 1996]

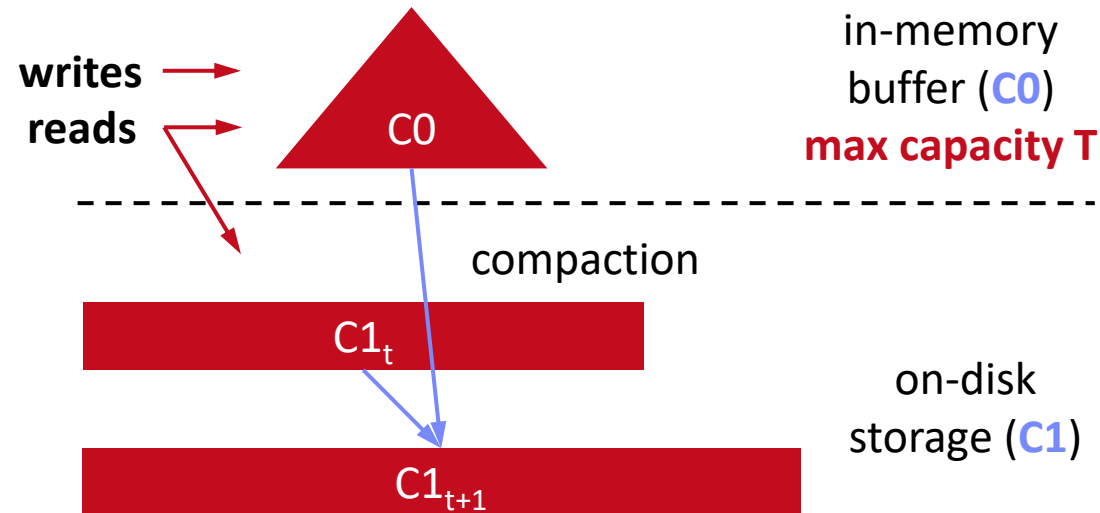


■ LSM Overview

- Many KV-stores rely on LSM-trees as their storage engine (e.g., [BigTable](#), [DynamoDB](#), [LevelDB](#), [Riak](#), [RocksDB](#), [Cassandra](#), [HBase](#))
- **Approach:** Buffers writes in memory, flushes data as sorted runs to storage, merges runs into larger runs of next level (compaction)

■ LSM System Architecture

- Writes in C0
- Reads against C0 and C1 (w/ buffer for C1)
- Compaction (rolling merge): sort, merge, incl **deduplication**



Probabilistic Set Containment

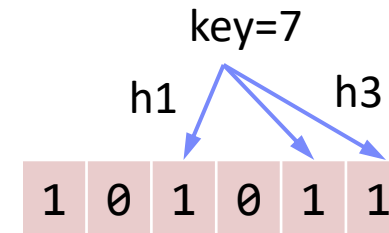


■ Motivation

- Many use cases for applying cheap pre-filters
- **Requirement: no false negatives**, small number of false positives (FP)

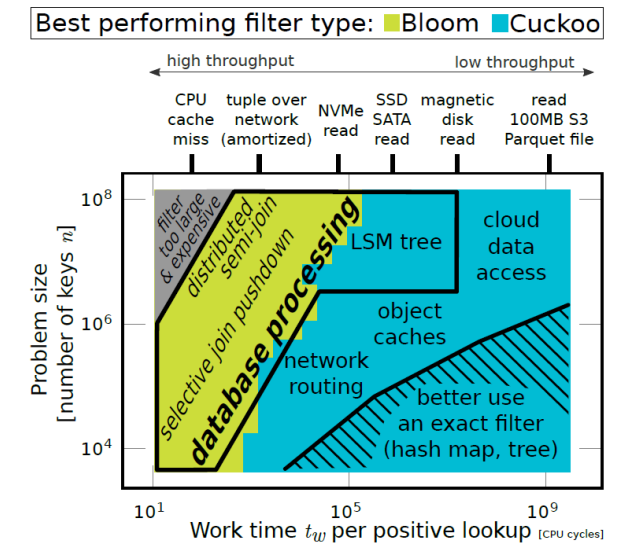
■ #1 Bloom Filter

- Array X of m bits, initialized w/ zeros
- k different hash functions applied on each key
- **Insert:** $k \times h_i(\text{key})$, set all hashed positions to 1
- **Query:** $k \times h_i(\text{key})$, return $(\sum(X[h_i(\text{key})])=k)$



■ #2 Cuckoo Filter

- Cuckoo hash table with key signatures
- 2 hash functions w/ displacements
- Allows deletes, duplicates, smaller FP rate



[Harald Lang, Thomas Neumann, Alfons Kemper, Peter A. Boncz: Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. **PVLDB 12(5), 2019**]



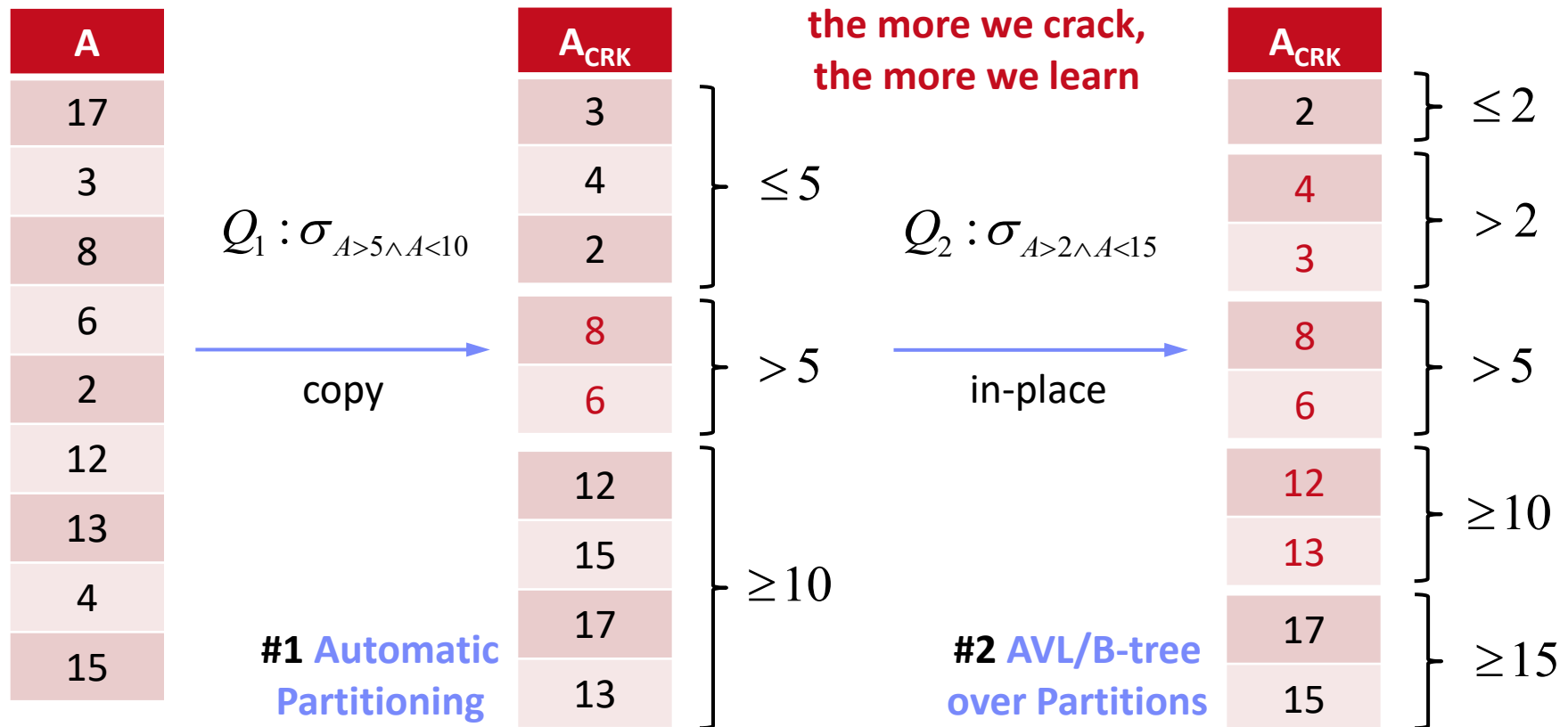
Learned Indexing

Excursus: 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**]



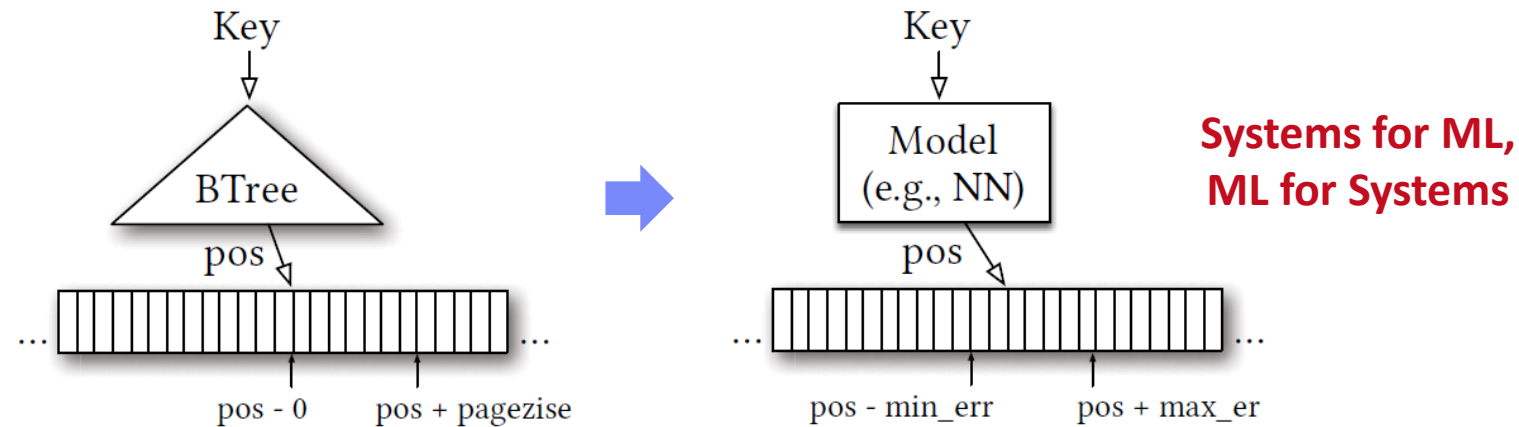
Learned Index Structures

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



■ 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)



■ Follow-up Work on SageDBMS



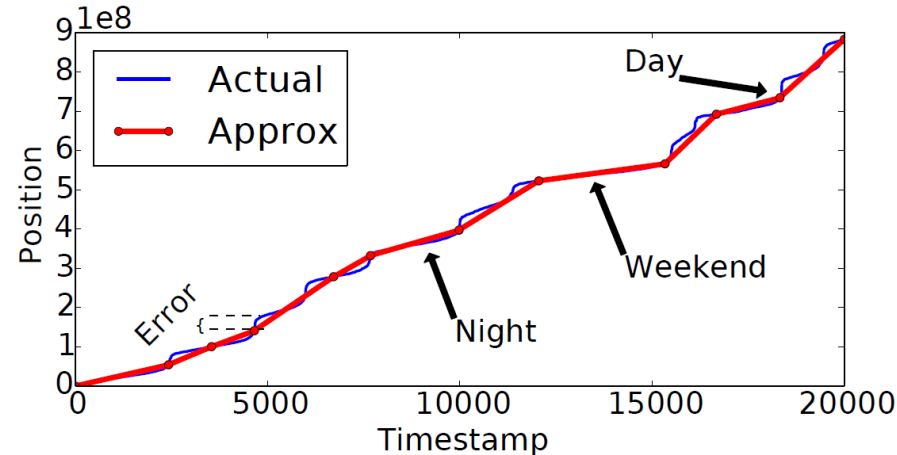
[Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, Vikram Nathan: [SageDB: A Learned Database System](#). CIDR 2019]

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**]



Thanks

- Overview Access Methods
- Index Structures
- Learned Indexing
- Next Lectures
 - Nov 10: [Background Transaction Processing](#)
 - Dec 08: [Experiments and Reproducibility](#)
 - Additional lectures / Q&A sessions on demand