

CS848 Fall 2026: Indexing for Modern Data Processing

B+ Trees

Structure, Operations, and Database Use

Xiao Hu

September 15, 2026

Announcement

- Please sign up the course presentation soon!
- An example of benchmark-type project:
 - <https://www.vldb.org/pvldb/vol14/p1640-wang.pdf>

Roadmap

- Relational data model and storage hierarchy
- Index design choices
- B+ tree
- Performance analysis
- How DBMSs use B+ trees and modern variants

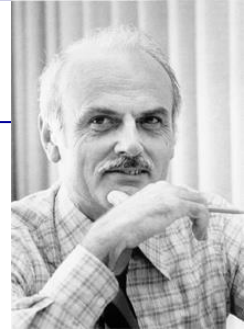
Learning goals:

Explain how a B+ tree is structured

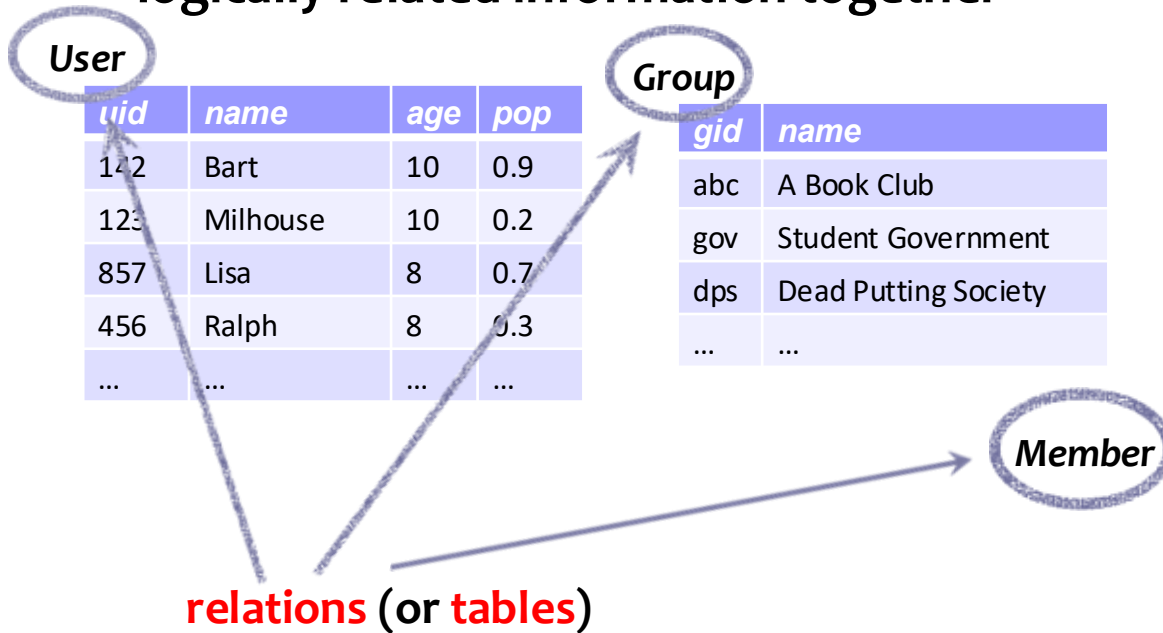
Trace lookup, range search, insertion and deletion procedures.

Connect textbook invariants to production database systems.

Data Organization in Traditional Data Systems



Modeling data as **relations** or **tables**, each storing logically related information together



Edgar F. Codd (1923-2003)

- Inventor of the relational model and algebra while at IBM; Turing Award, 1981

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
...	...

Data Organization in Traditional Data Systems

attributes (or columns)

User

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3
...	...	...	...

String

Int

Float

domain (or type)

Group

<i>gid</i>	<i>Description</i>
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
...	...

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
...	...

Member

tuples (or rows)

Duplicates (all attr. have same val) are not allowed

Ordering of rows doesn't matter
(even though output can be ordered)

Data Organization in Traditional Data Systems

(Candidate) key: A minimal subset of attributes that uniquely identify tuples

User

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3
...	...	...	...

uid

Group

<i>gid</i>	<i>Description</i>
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
...	...

gid

Member

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
...	...

uid, gid

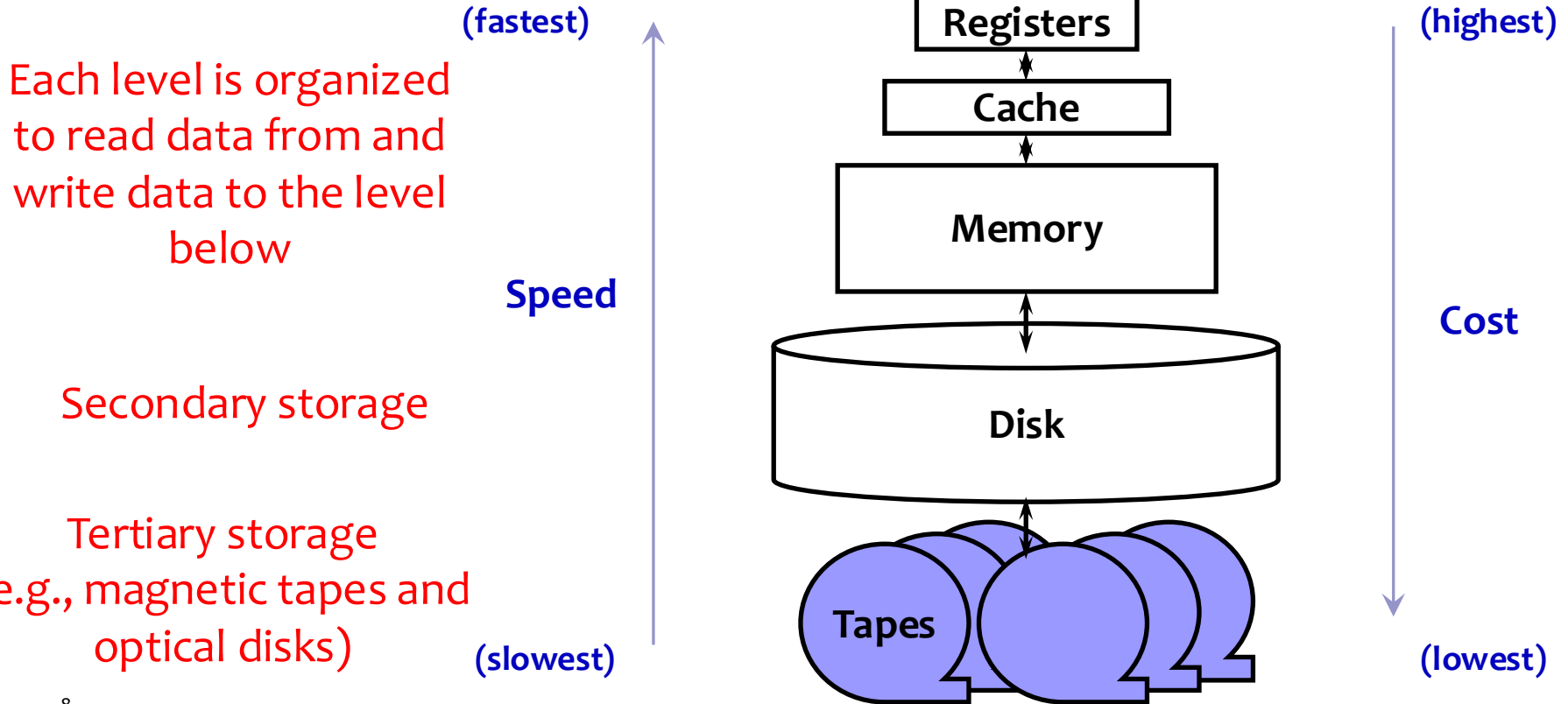
Summary of Relational data model

- A database is a collection of **relations** (or **tables**)
- Each relation has a set of **attributes** (or **columns**)
- Each attribute has a unique name and a **domain** (or **type**)
 - The domains are required to be **atomic**
- Each relation contains a set of **tuples** (or **rows**)
 - Each tuple has a value for each attribute of the relation
 - **Duplicate tuples are not allowed**
- Each relation contains a (candidate) **key**
- ☞ **Simplicity is a virtue!**



Single, indivisible
piece of information

Data Storage in Traditional Data Systems



I/O Cost Model

#blocks/#pages
transferred between main
memory and disk

Memory

<i>uid</i>	<i>gid</i>
142	dps
123	gov

Computation is done
in main memory
without cost



Disk

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3

<i>gid</i>	<i>name</i>
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
...	...

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov

PostgreSQL: 8 KB
SQL Server: 8 KB
MySQL InnoDB: 16 KB
SQLite: commonly 4 KB

What Is an Index For?

Given a search key, **avoid scanning the entire relation.**

```
SELECT * FROM R  
WHERE A = value;
```

```
SELECT * FROM R  
WHERE A > value;
```

```
SELECT * FROM R, S  
WHERE R.A = S.B;
```

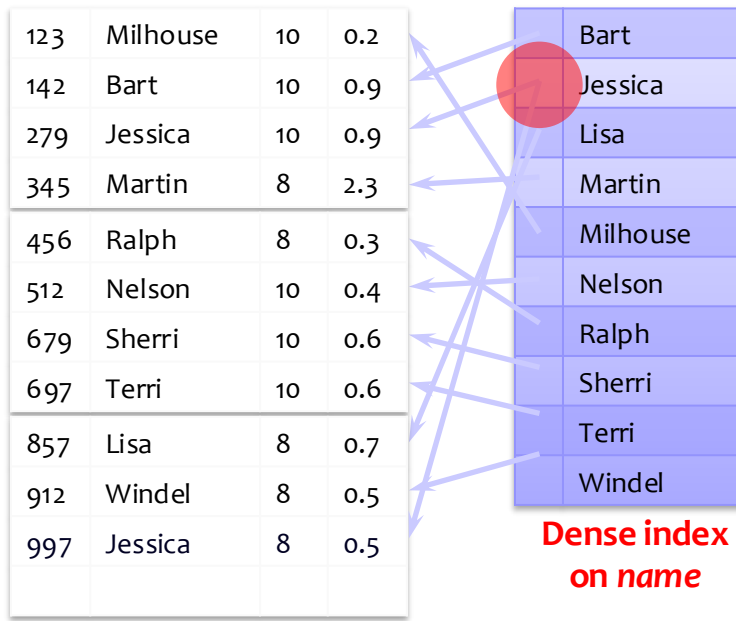
```
SELECT * FROM R  
ORDER BY A LIMIT 10;
```

CREATE INDEX RAIndex **ON** R(A)

One ordered index can support equality, ranges, and order.

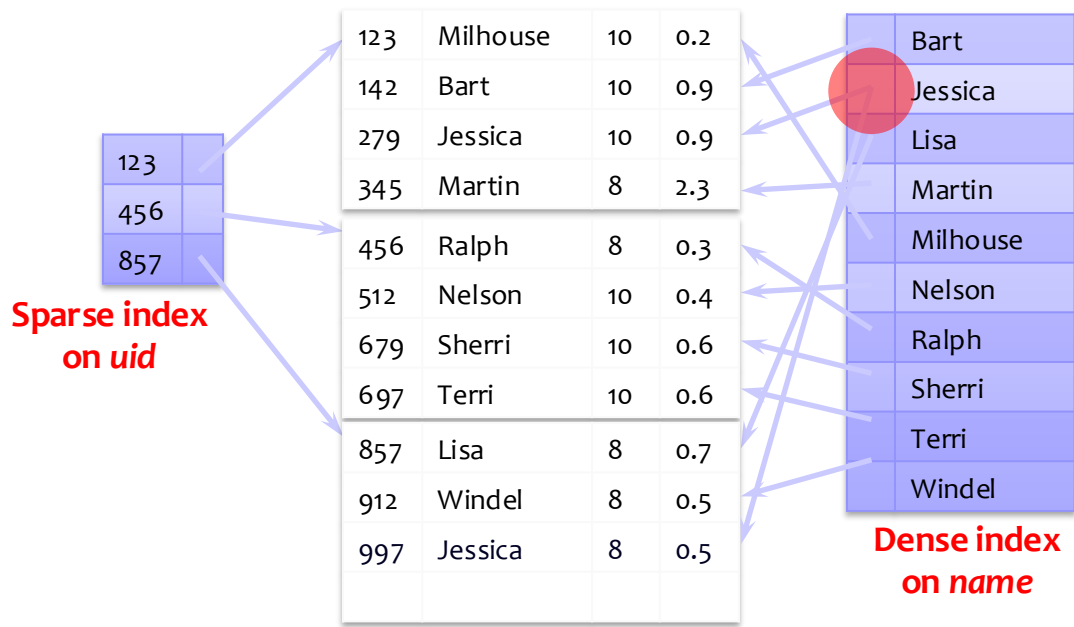
Dense v.s. Sparse indexes

- **Dense:** one index entry for each search key value
 - One entry may “point” to multiple records (e.g., two users named Jessica)



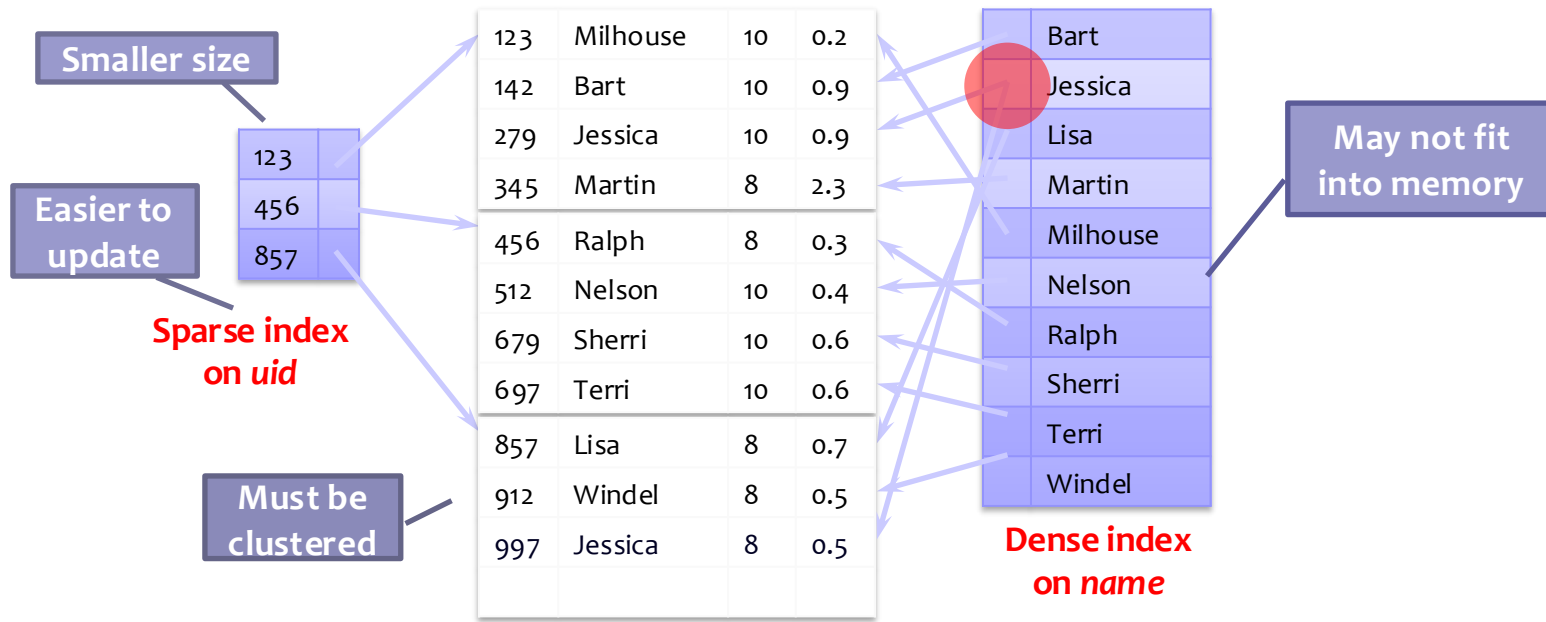
Dense v.s. Sparse indexes

- **Sparse**: one index entry for each block
 - Records must be **clustered** according to the search key on the disk



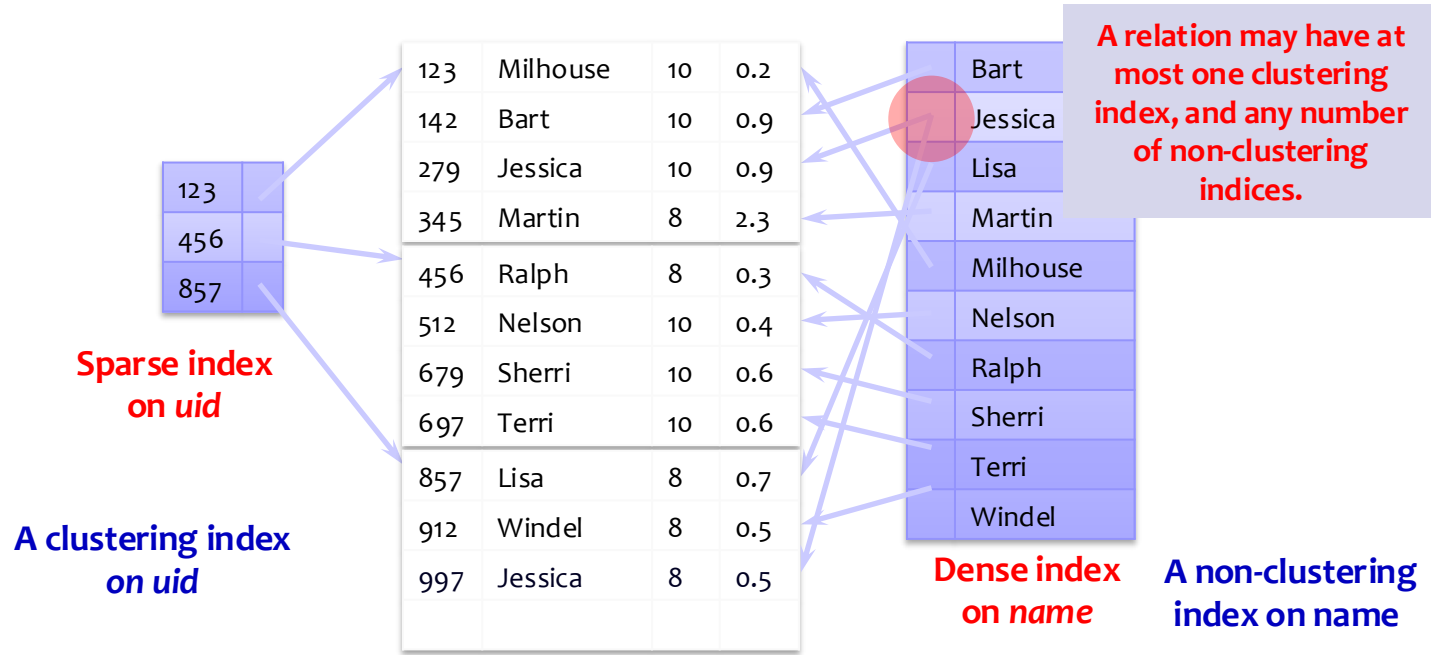
Dense v.s. Sparse indexes

- **Dense**: one index entry for each search key value
- **Sparse**: one index entry for each block
 - Records must be **clustered** according to the search key



Clustering v.s. Non-Clustering indexes

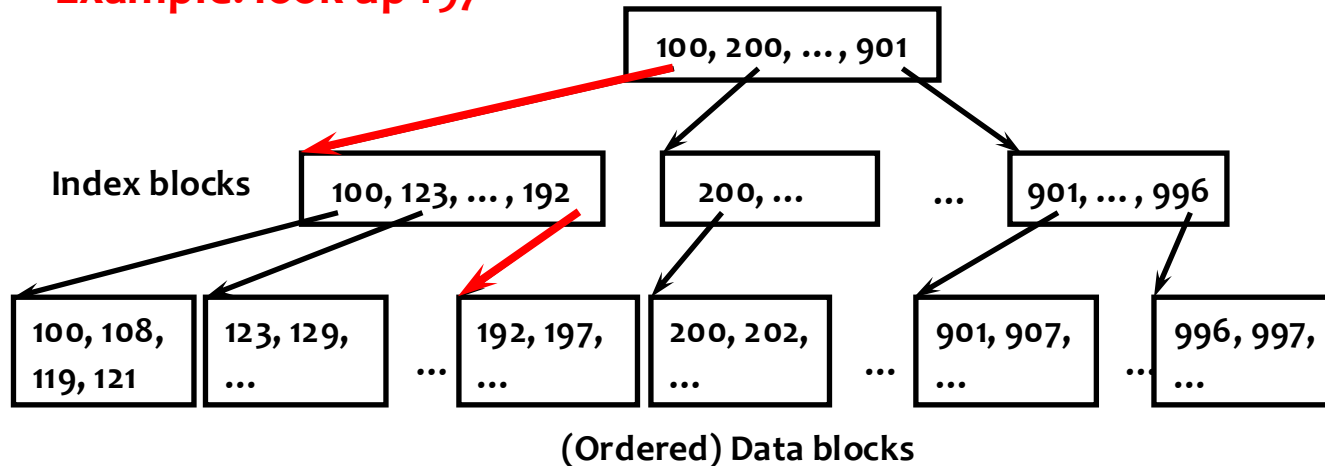
- An index on attribute A is a **clustering** index if tuples with similar A-values are stored together in the same block, and **non-clustering** otherwise.



First Attempt

- ISAM (Index Sequential Access Method)
- What if an index is still too big?
 - Put a another (sparse) index on top of that!

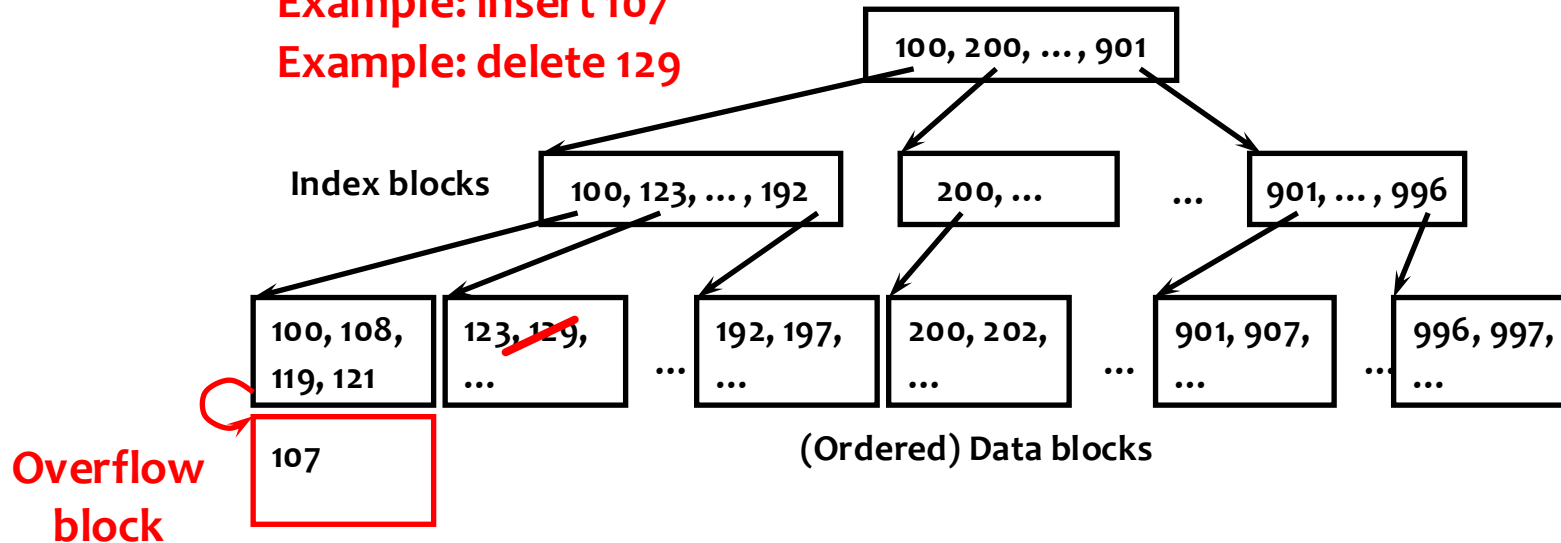
Example: look up 197



Updates with ISAM

Example: insert 107

Example: delete 129



- Overflow chains and empty data blocks degrade performance
 - Worst case: most records go into one long chain, so lookups require scanning all data!

Design Target: A Dynamic Ordered Index

Balanced	Every leaf remains at roughly the same depth.
Block-oriented	One node fits one storage block
High fan-out	Hundreds of children keep height as small as possible
Local repair	Lookup, insertion and deletion touch one search path.
Ordered leaves	Range scans continue sequentially.

B-tree Family

- There is a specific data structure called B-tree
- People also use the term to generally refer to a class of balanced tree structures:
 - ➔ B-tree (1970)
 - B+ tree (1973)
 - B* tree (1977)
 - ➔ Blink tree (1981)
 - B ϵ tree (2003)
 - Bw tree (2013)
 - ...

postgres / src / backend / access / nbtree / README



petergeoghegan Use fake LSNs to improve nbtree dropPin behavior. ✓

Code

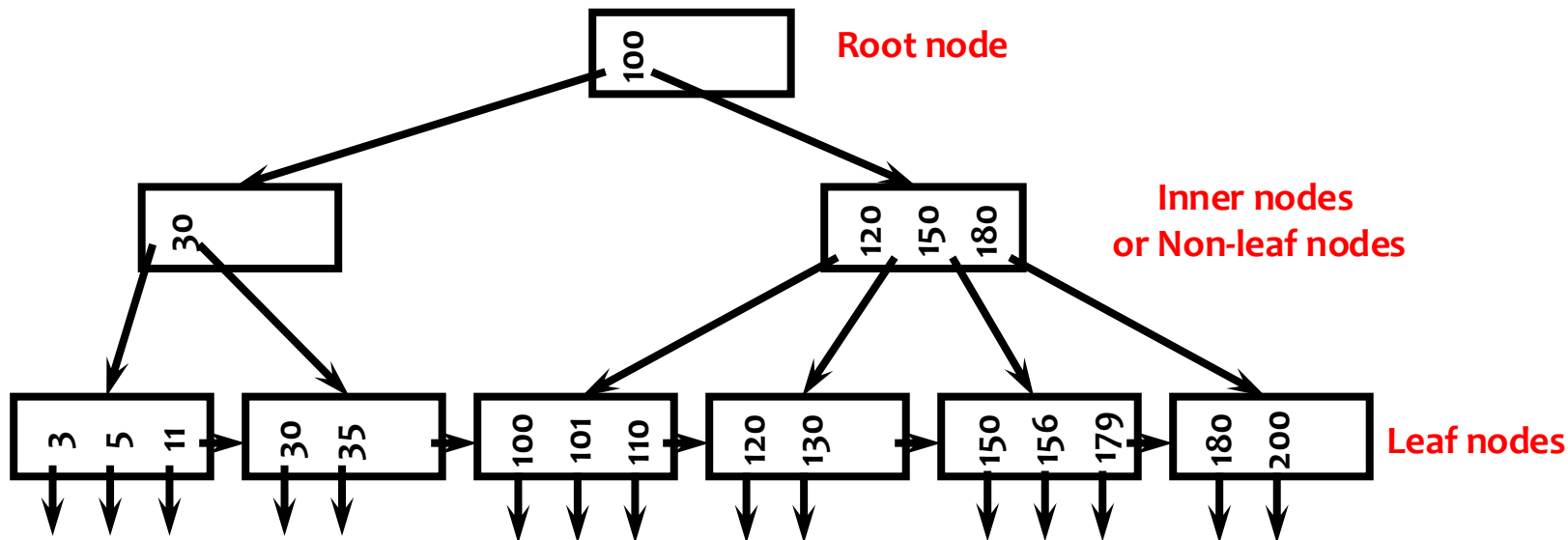
Blame

1082 lines (958 loc) · 62.8 KB

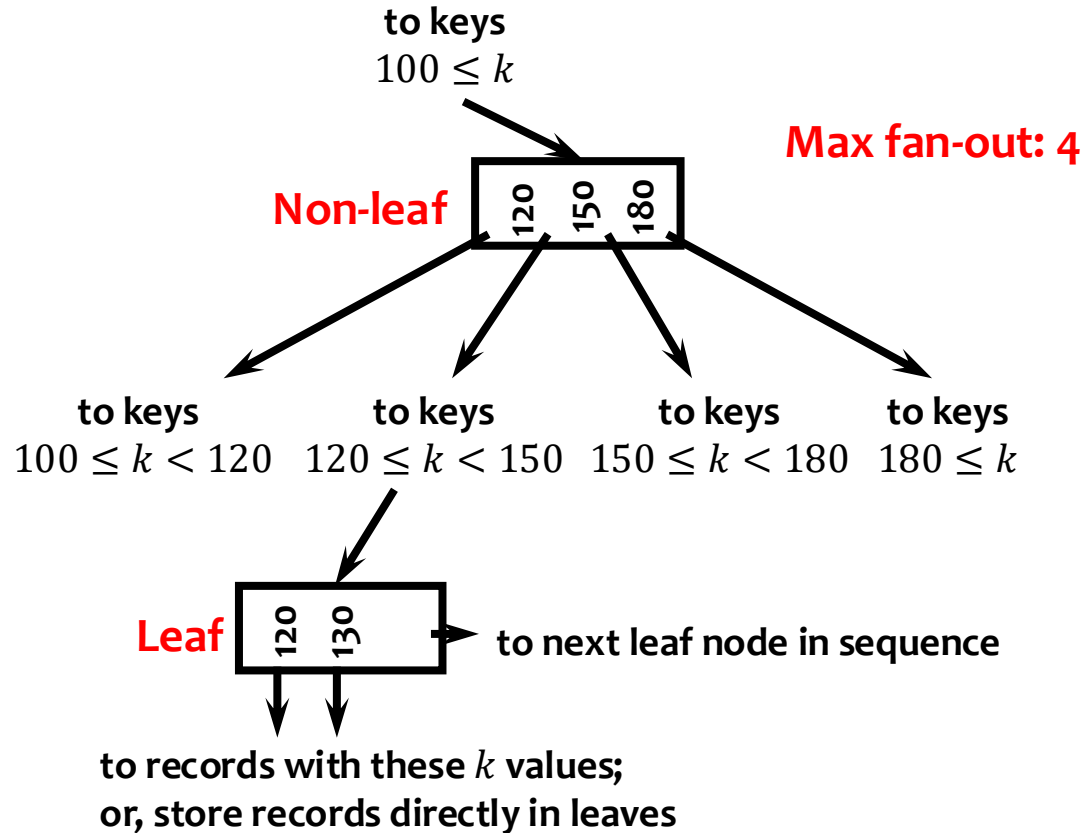
```
1  src/backend/access/nbtree/README
2
3  Btree Indexing
4  =====
5
6  This directory contains a correct implementation of Lehman and Yao's
7  high-concurrency B-tree management algorithm (P. Lehman and S. Yao,
8  Efficient Locking for Concurrent Operations on B-Trees, ACM Transactions
9  on Database Systems, Vol 6, No. 4, December 1981, pp 650-670). We also
10 use a simplified version of the deletion logic described in Lanin and
11 Shasha (V. Lanin and D. Shasha, A Symmetric Concurrent B-Tree Algorithm,
12 Proceedings of 1986 Fall Joint Computer Conference, pp 380-389).
13
```

B+ Tree

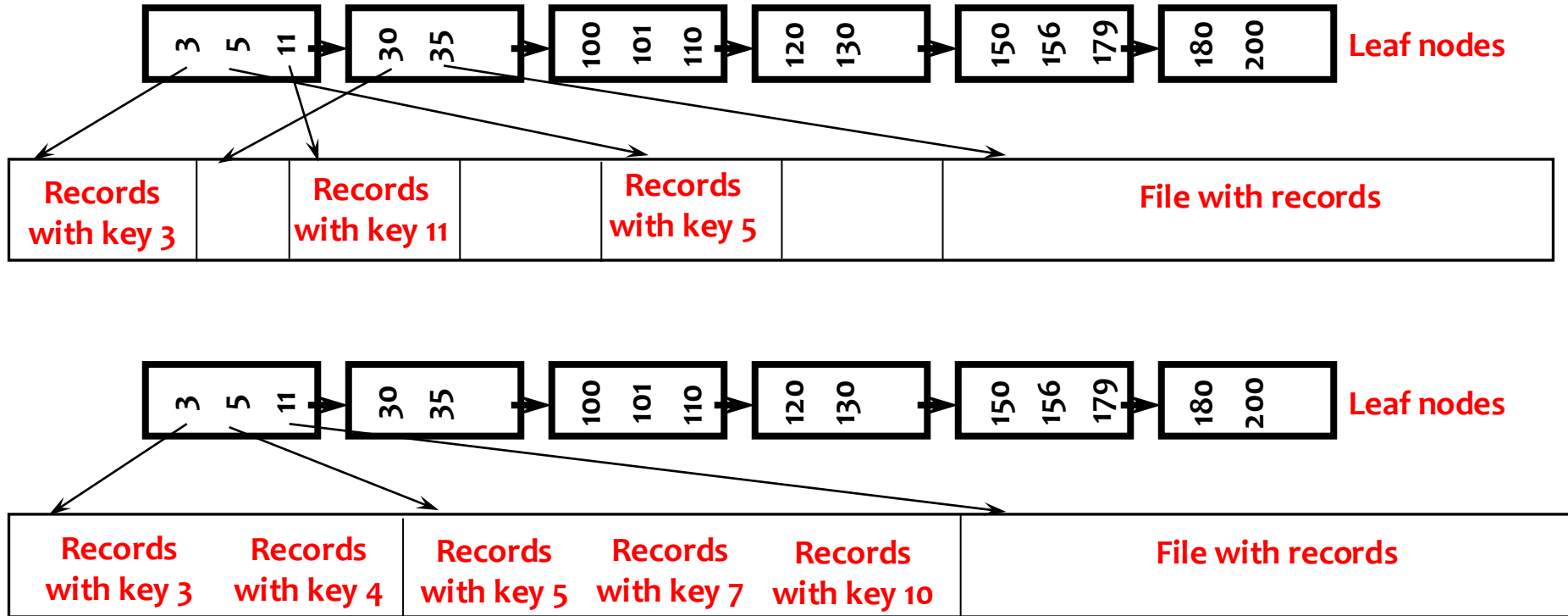
- A hierarchy of nodes with intervals
- **Balanced**: good performance guarantee
- **Disk-based**: one node per block; large fan-out



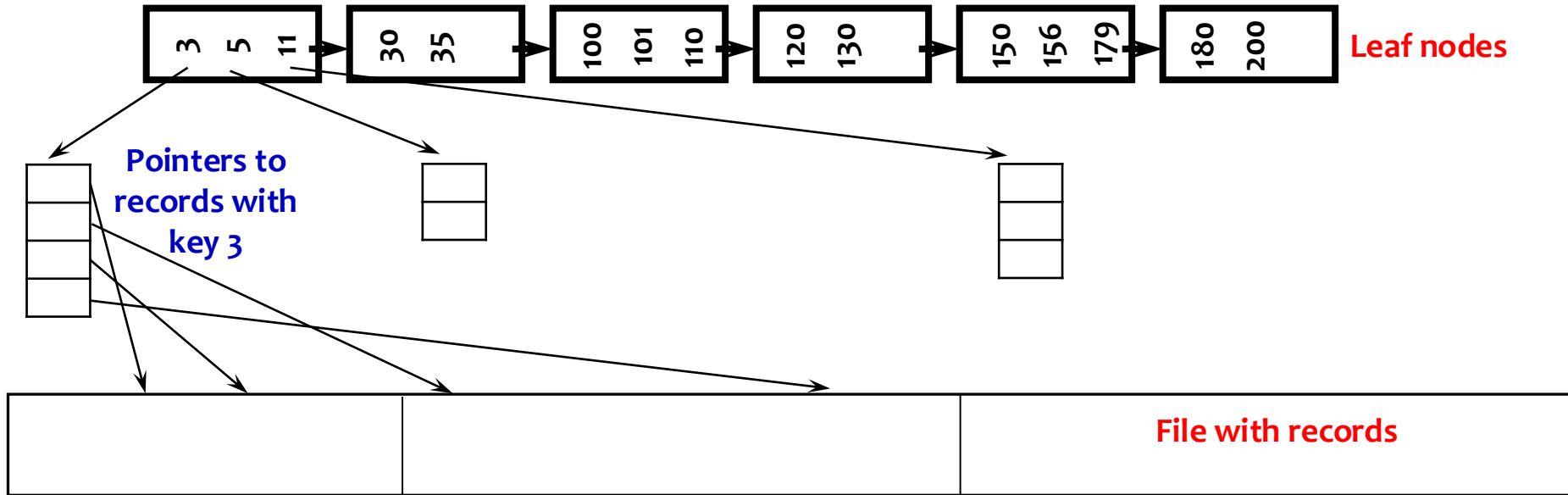
B+ tree nodes



Dense/Sparse and Clustering/Non-clustering B+ tree



Dense/Sparse and Clustering/Non-clustering B+ tree



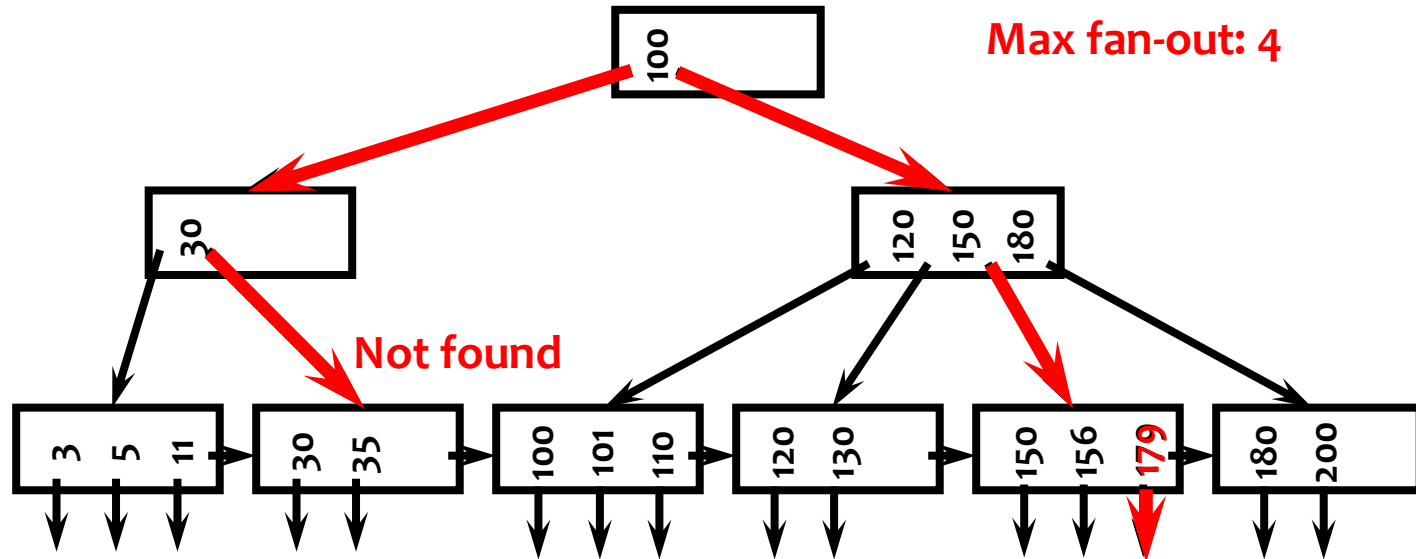
B+ tree balancing properties

- Height constraint: all leaves at the same lowest level
- Fan-out constraint: all nodes at least half full (except root)

	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	f	$f - 1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f - 1$	2	1
Leaf	f	$f - 1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

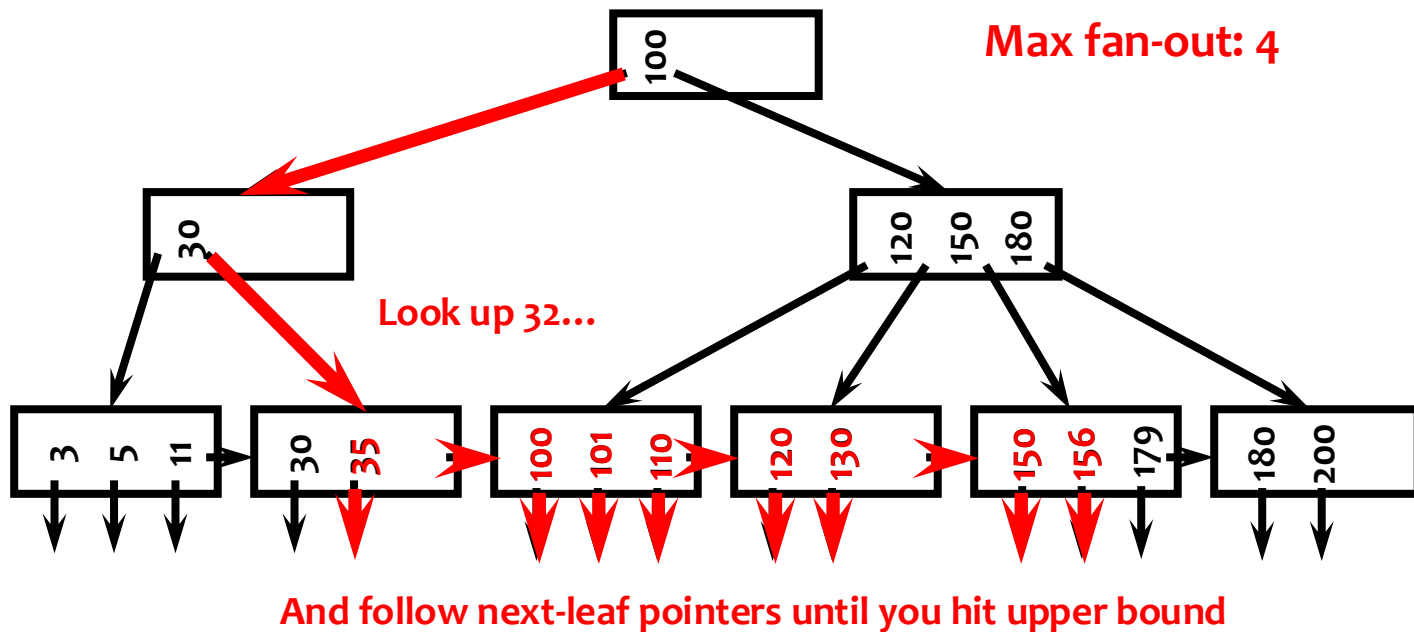
Lookups

- SELECT * FROM R WHERE A = 179;
- SELECT * FROM R WHERE A = 32;



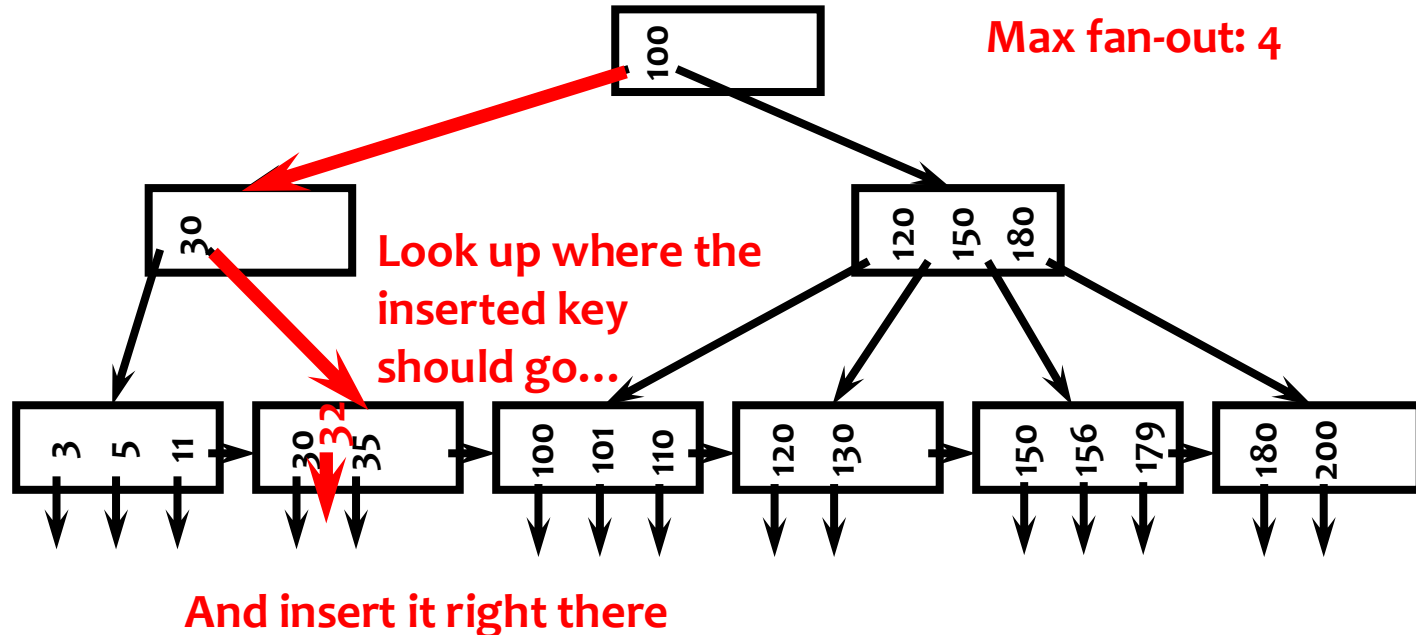
Range Query

■ SELECT * FROM R WHERE $A > 32$ AND $A < 179$;



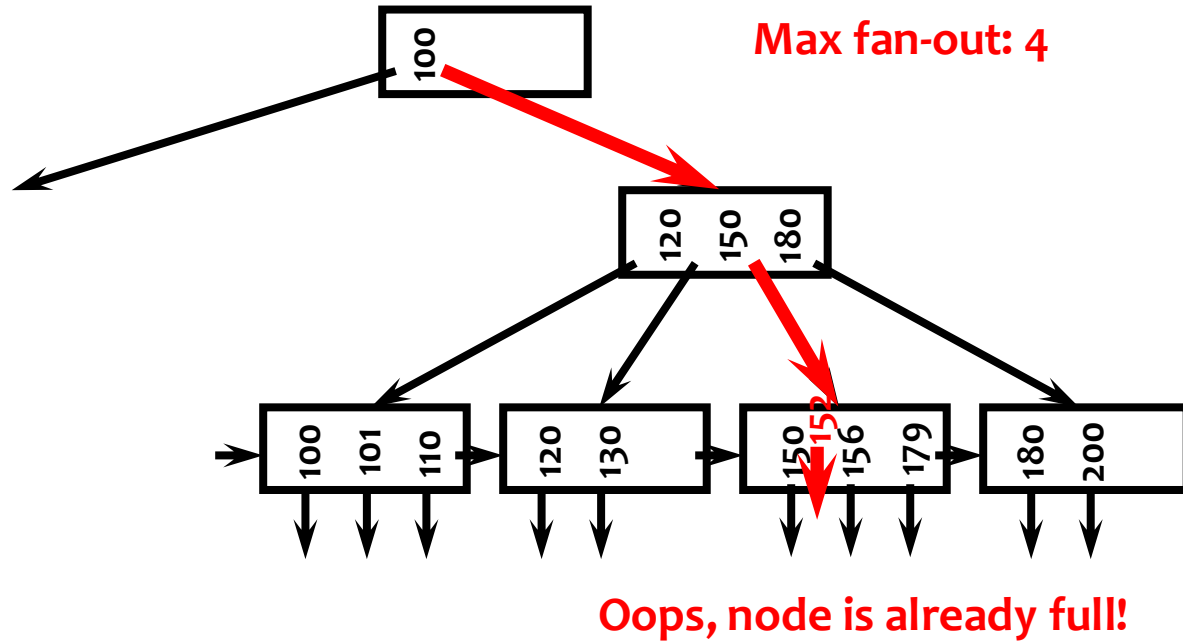
Insertion

- Insert a record with search key value 32

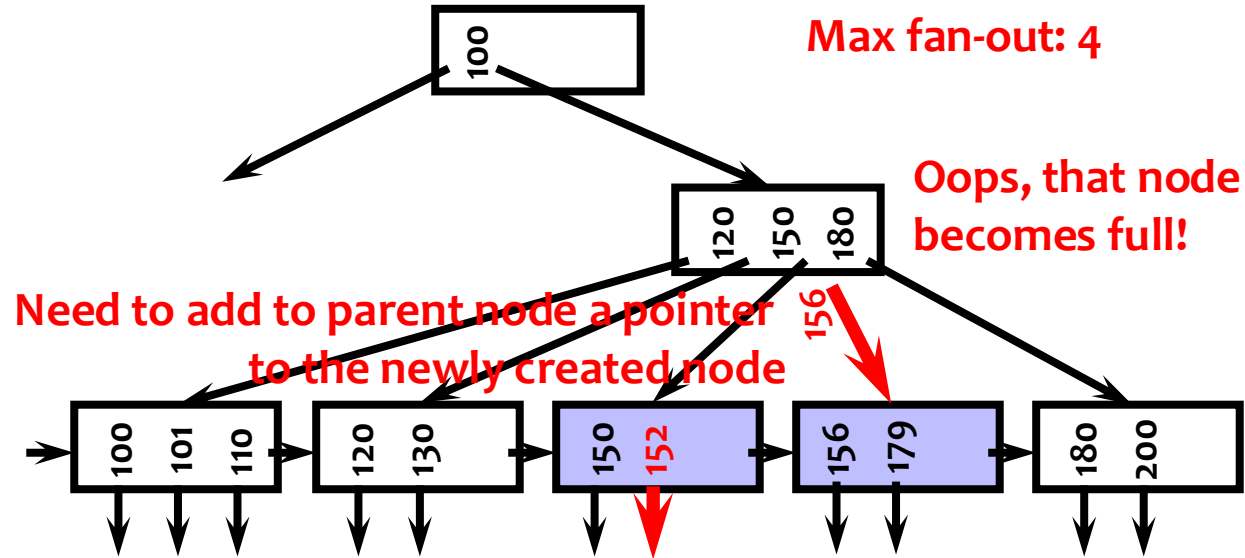


Another insertion example

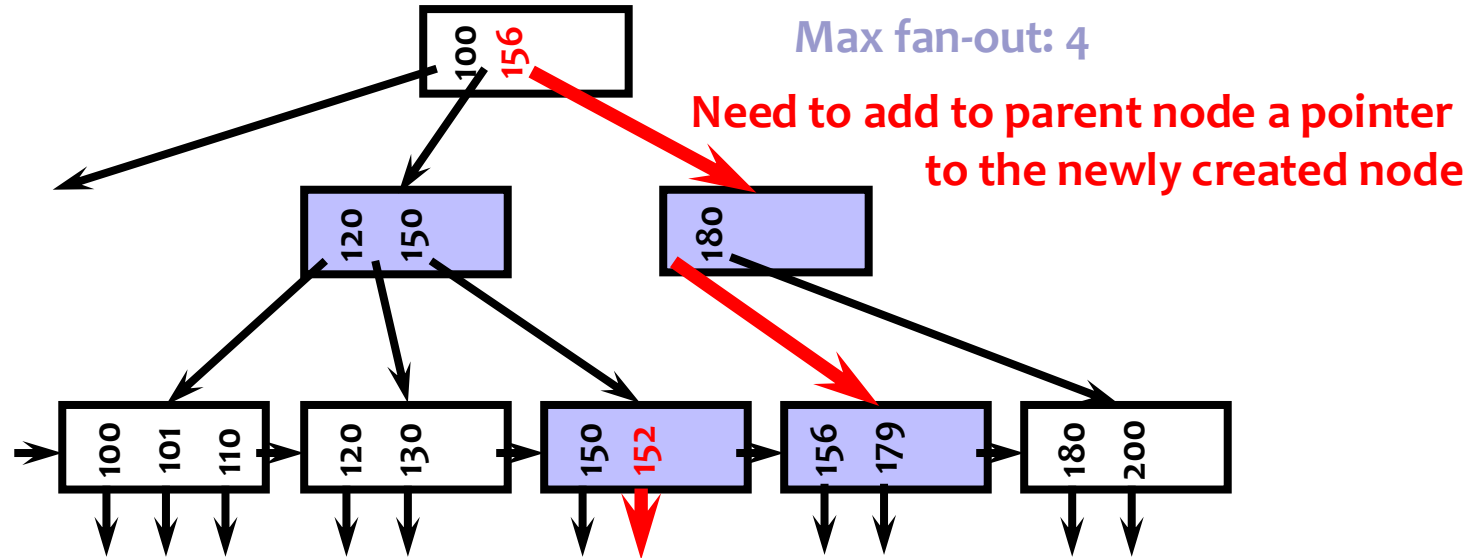
- Insert a record with search key value 152



Node splitting



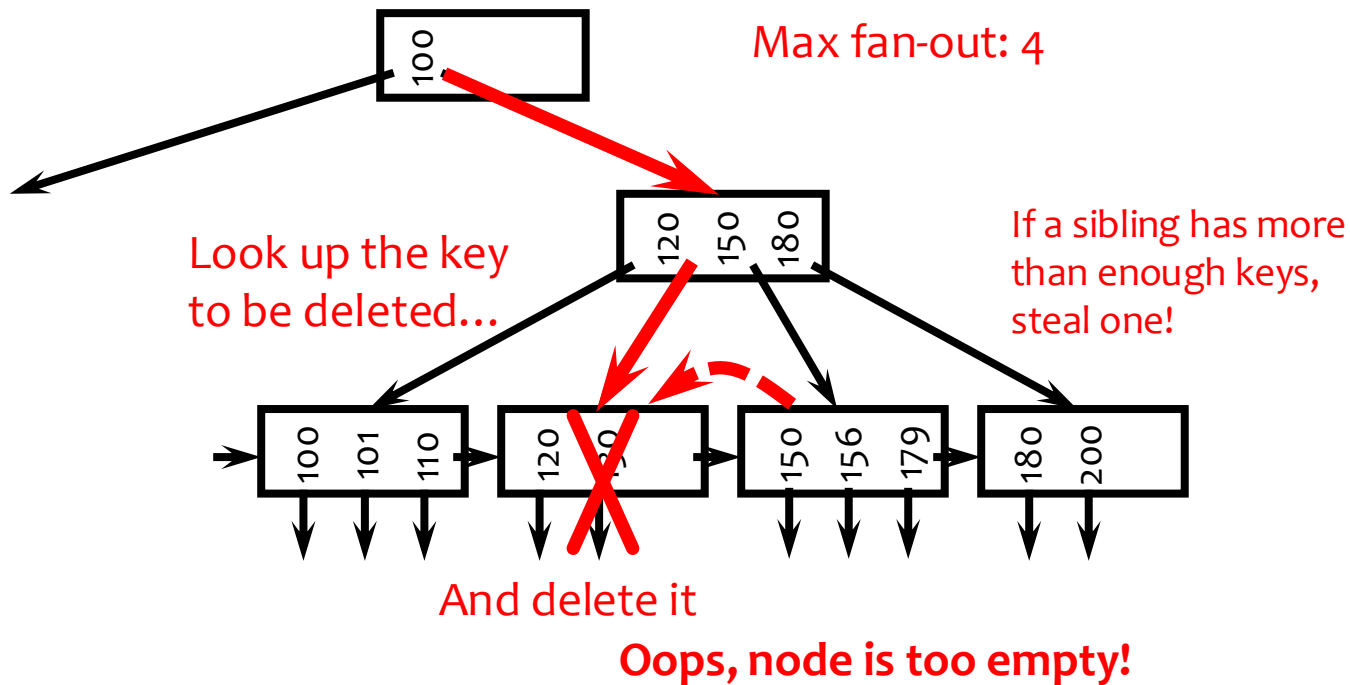
More node splitting



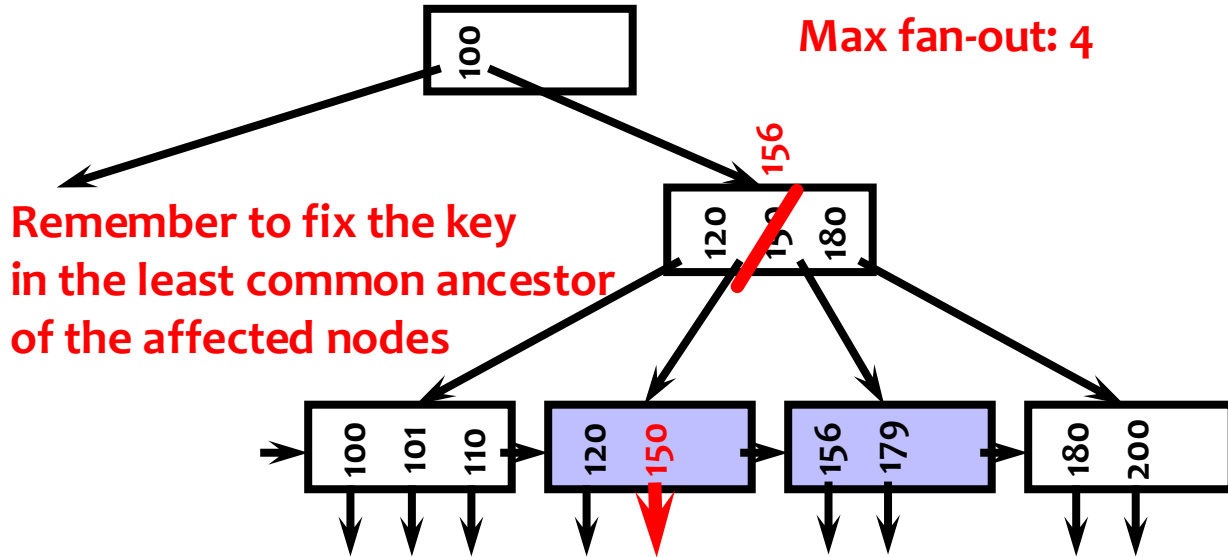
- In the worst case, node splitting can “propagate” all the way up to the root of the tree (not illustrated here)
 - Splitting the root introduces a new root of fan-out 2 and causes the tree to grow “up” by one level

Deletion

- Delete a record with search key value 130

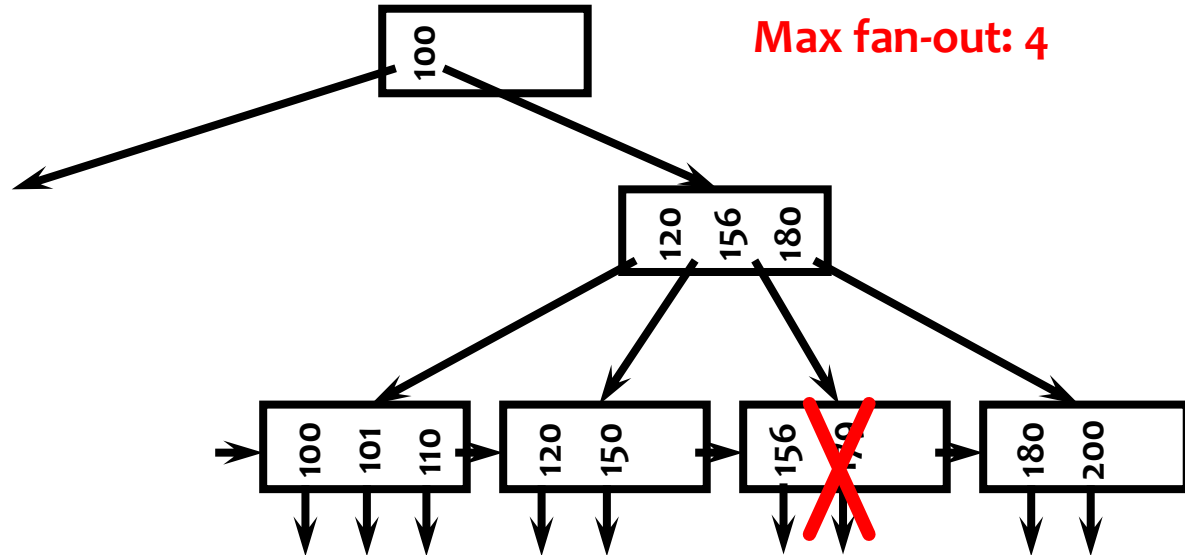


Stealing from a sibling



Another deletion example

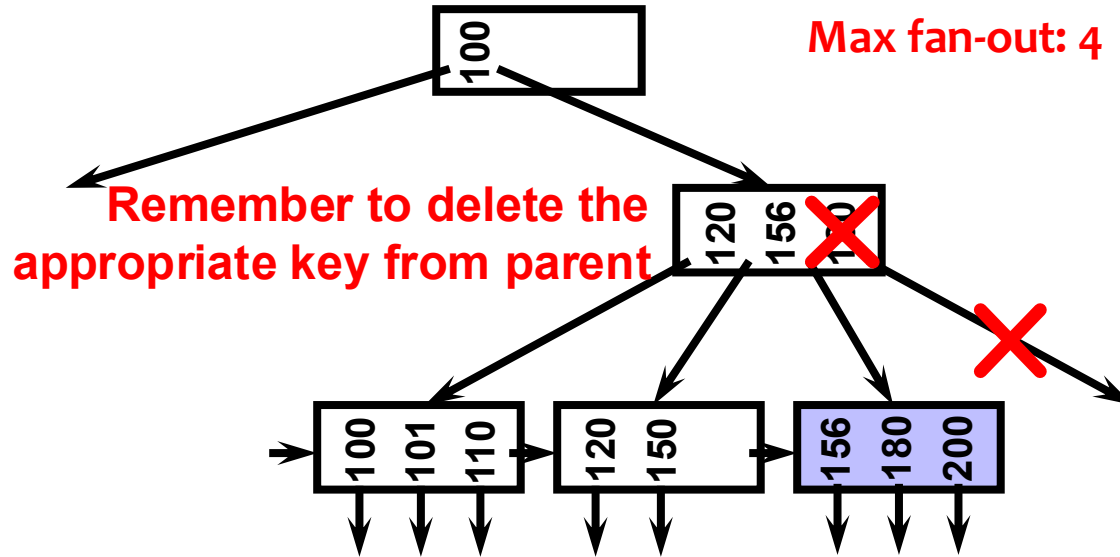
- Delete a record with search key value 179



Cannot steal from siblings

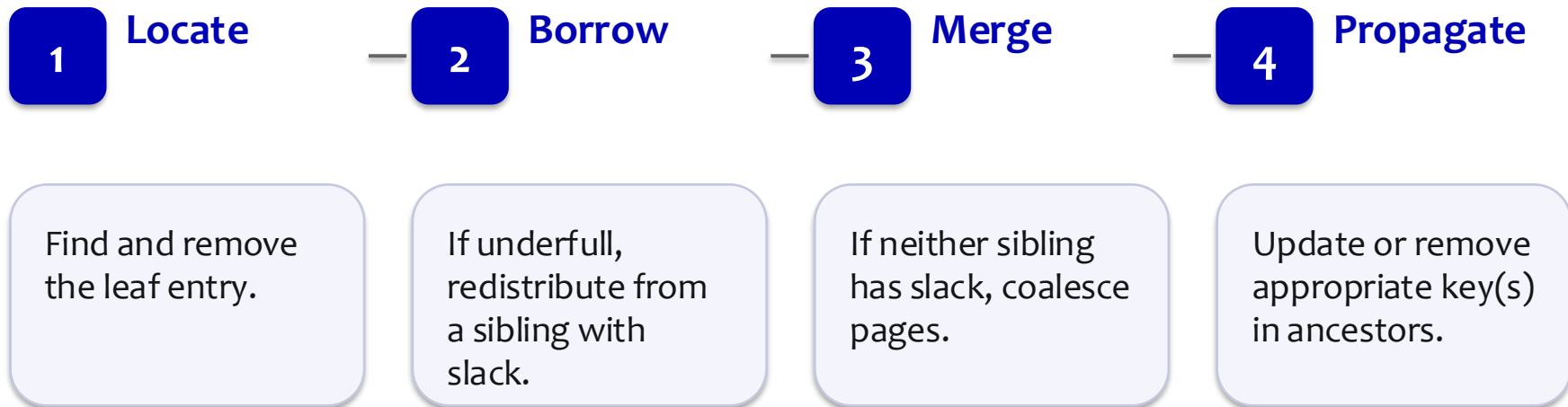
Then coalesce (merge) with a sibling!

Coalescing



- Deletion can “propagate” all the way up to the root of the tree (not illustrated here)
 - When the root becomes empty, the tree “shrinks” by one level

Deletion: Repair Only When Necessary



Deletion can decrease height only when the root becomes empty.

Deletion in Practice Is Often Lazy

Textbook policy

Maintain half-full pages after every deletion.

Borrow or merge immediately.

Strong occupancy invariant.

Production policy

Mark entries dead or removable.

Delay merging.

Reclaim space during vacuum, rebuild, or background work.

Why delay?

Immediate merges add latches, logging, and oscillation.

Underfull pages may still give acceptable search cost.

Production invariants balance space, concurrency, and maintenance cost.

Performance analysis of B+ tree

- How many I/O's are required for each operation?
 - h , the height of the tree
 - Plus one or two to manipulate actual records
 - Plus $O(h)$ for reorganization (rare if f is large)
 - Minus one if we cache the root in memory
- How big is h ?
 - Roughly $\log_f N$, where N is the number of records
 - Fan-out is typically large (in hundreds) -- many keys and pointers can fit into one block
 - A 4-level B+ tree is enough for “typical” tables

Why B+ Trees Instead of B-Trees?

B-tree

Records or row pointers may appear in internal nodes.

Potential benefit: some records need fewer page reads.

Costs: lower fan-out, harder deletion, and less uniform range scans.

B+ tree

Internal pages contain routing data only.

All records or row pointers live in leaves, so search must reach a leaf node.

Benefits: higher fan-out, uniform lookup, efficient leaf scans and easier implementation.

Database systems usually prefer predictable routing and range performance.

Applications: B+ Trees Also Support Indexed Joins

Question: How to use B+ tree to join User, Group and Member?

User

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3
...	...	...	...

Group

<i>gid</i>	<i>name</i>
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
...	...

Member

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
...	...

How Major DBMSs Use B+ Trees

PostgreSQL

Default ordered secondary index.

Heap tuple IDs at leaves.

MVCC cleanup and deduplication.

MySQL

Primary B+ tree stores rows.

Secondary leaves store primary keys.

Clustering key affects every index.

SQL Server

Clustered leaves store rows.

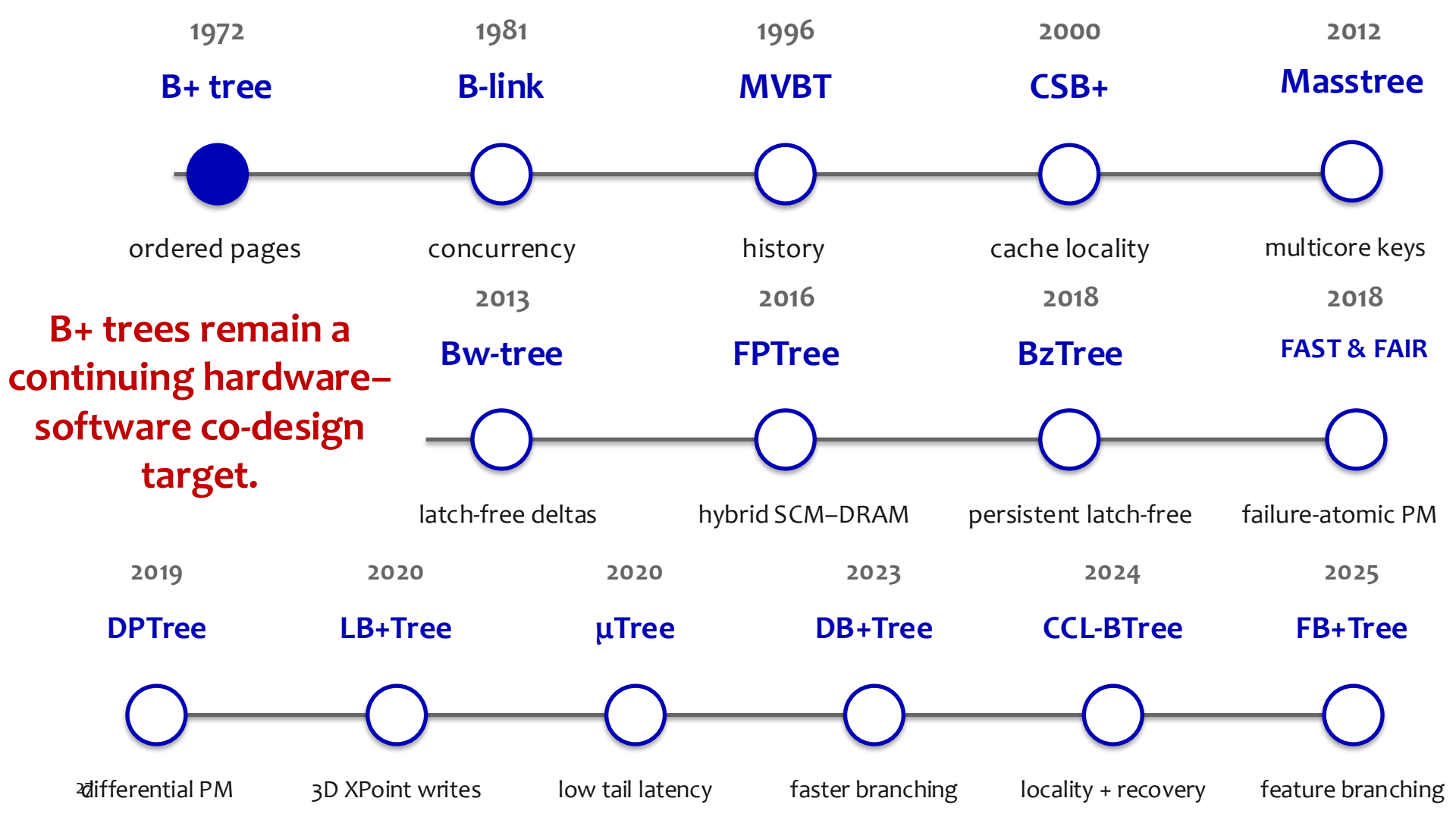
Nonclustered leaves store locators and included columns.

SQLite

Table B-trees for rowid tables.

Index B-trees for indexes and WITHOUT ROWID tables.

The same B+ tree shape supports different leaf and transaction semantics.



Takeaways

1

A B+ tree is a shallow, block-oriented hierarchy of key intervals.

2

Linked leaves make ranges and ordered operators efficient.

3

Splits and merges preserve balance through local repair.

4

Leaf payloads distinguish dense/sparse, and clustering/non-clustering indexes.

5

Real systems add concurrency, recovery, MVCC, compression, and lazy cleanup.