CS 348 Lecture 15

Indices

Semih Salihoğlu

Mar 5th 2025

Outline For Today

Database Indices

- 5 Index Designs in Increasing Level of Robustness
- Using Indices In Practice

Outline For Today

Database Indices

- 5 Index Designs in Increasing Level of Robustness
- Using Indices In Practice

Functionality of Indices (1)

- Indices are the primary mechanism to:
  1. *retrieve records quickly*
  2. *search records in sort order*

```
SELECT * FROM Students WHERE ID = 912;
SELECT * FROM Students WHERE ID > 100;
```

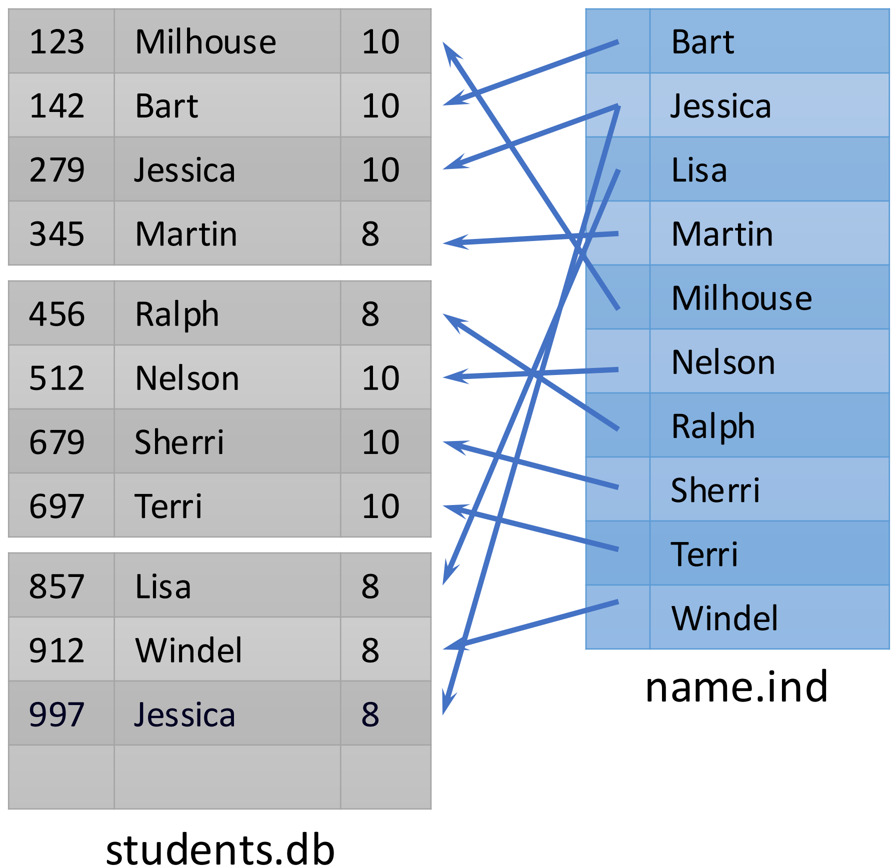
- Default way to find records: sequential scans
  - Read each page and each record
  - Can be very slow for large tables
  - If a file is sorted on some columns:
    - Can now do binary search
    - Key Question: How to sort a file efficiently?

	ID	name	mark
page 1	123	Milhouse	10
	142	Bart	10
	279	Jessica	10
	345	Martin	8
page 2	456	Ralph	8
	512	Nelson	10
	679	Sherri	10
	697	Terri	10
page 3	857	Lisa	8
	912	Windel	8
	997	Jessica	8
…			
page k	…	…	…
	…	…	…
	…	…	…

students.db

Functionality of Indices (2)

- Indices: are persistent data structures that are stored along with table files that allow fast search.
- An example of a simple index: can be much smaller than the original table.



Naïve Approach for Keeping a Table Sorted

- Sorting: primary technique to find things & data quickly!
- Once a file is sorted, we can do binary search on the pages of the file.
- How to keep a relation file in sorted order (e.g., students.db)?
- Assume a sequence of insertions. 2 records/page. Sort on ID
- Simple/naïve approach: Shift-based (index-less) sorting

Naïve Approach for Keeping a Table Sorted

Next Insertion

857	Lisa	8

PAGES

Naïve Approach for Keeping a Table Sorted

Next Insertion

512	Nelson	10

PAGES

857	Lisa	8

pg 1

Naïve Approach for Keeping a Table Sorted

Next Insertion

279	Jessica	10

PAGES

512	Nelson	10
857	Lisa	8

pg 1

Naïve Approach for Keeping a Table Sorted

Next Insertion

912	Windel	8

PAGES

pg 1

279	Jessica	10
512	Nelson	10

pg 2

857	Lisa	8

Naïve Approach for Keeping a Table Sorted

Next Insertion

345	Martin	8

PAGES

279	Jessica	10	pg 1
512	Nelson	10	
857	Lisa	8	pg 2
912	Windel	8	

Naïve Approach for Keeping a Table Sorted

Next Insertion

697	Terri	10

PAGES

279	Jessica	10	pg 1
345	Martin	8	

512	Nelson	10	pg 2
857	Lisa	8	

912	Windel	8	pg 3

Naïve Approach for Keeping a Table Sorted

Next Insertion

123	Milhouse	10

PAGES

279	Jessica	10	pg 1
345	Martin	8	
512	Nelson	10	pg 2
697	Terri	10	
857	Lisa	8	pg 3
912	Windel	8	

Naïve Approach for Keeping a Table Sorted

Next Insertion

PAGES

123	Milhouse	10	pg 1
279	Jessica	10	
345	Martin	8	pg 2
512	Nelson	10	
697	Terri	10	pg 3
857	Lisa	8	
912	Windel	8	pg 4

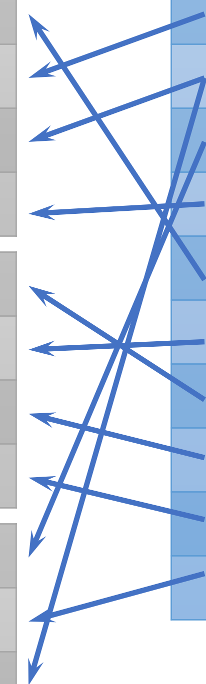
- Pro: Very simple to implement
- Con: Each insertion could require up to 2xb many I/Os (to read and right pages) if the table has b pages.
  - Will not scale. Not practical.

2nd Approach: Single-level Dense Index

- Lookup: find the record in index & follow pointer (page & offset)
- If index file is disk-based:
  - Con: Same problem as naïve solution
  - Pro: But at a smaller scale b/c the index is smaller (a projection).
- If index is in memory:
  - Pro: Optimal I/O. Only store the record in the relation file but no sorting (called unclustered index)
  - Con: Index cannot get very large.

123	Milhouse	10
142	Bart	10
279	Jessica	10
345	Martin	8
456	Ralph	8
512	Nelson	10
679	Sherri	10
697	Terri	10
857	Lisa	8
912	Windel	8
997	Jessica	8

students.db

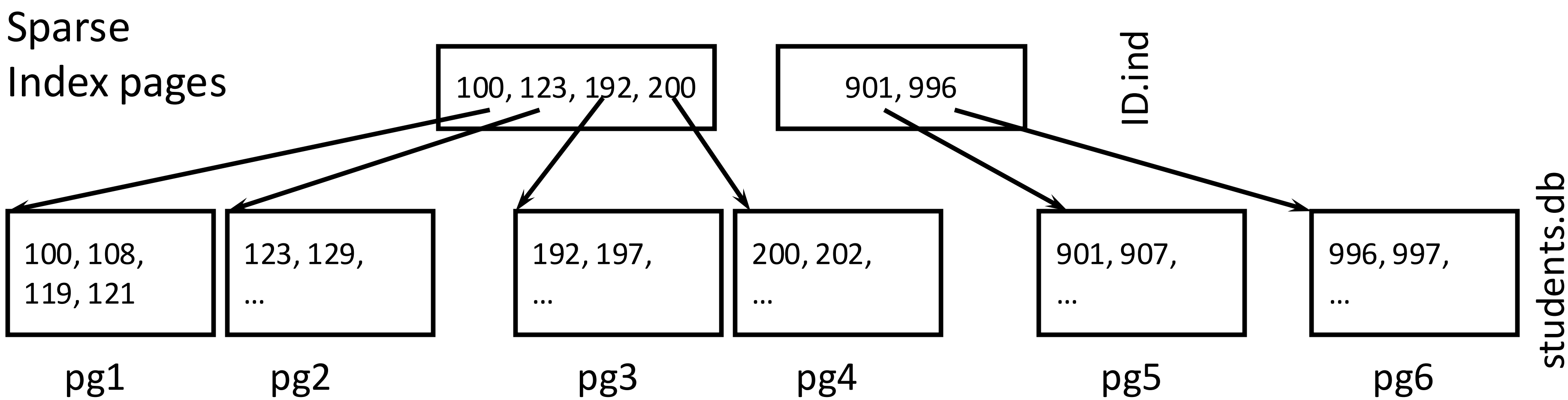


	Bart
	Jessica
	Lisa
	Martin
	Milhouse
	Nelson
	Ralph
	Sherri
	Terri
	Windel

name.idx

3rd Approach: Single-level Sparse Index w/ Overflows

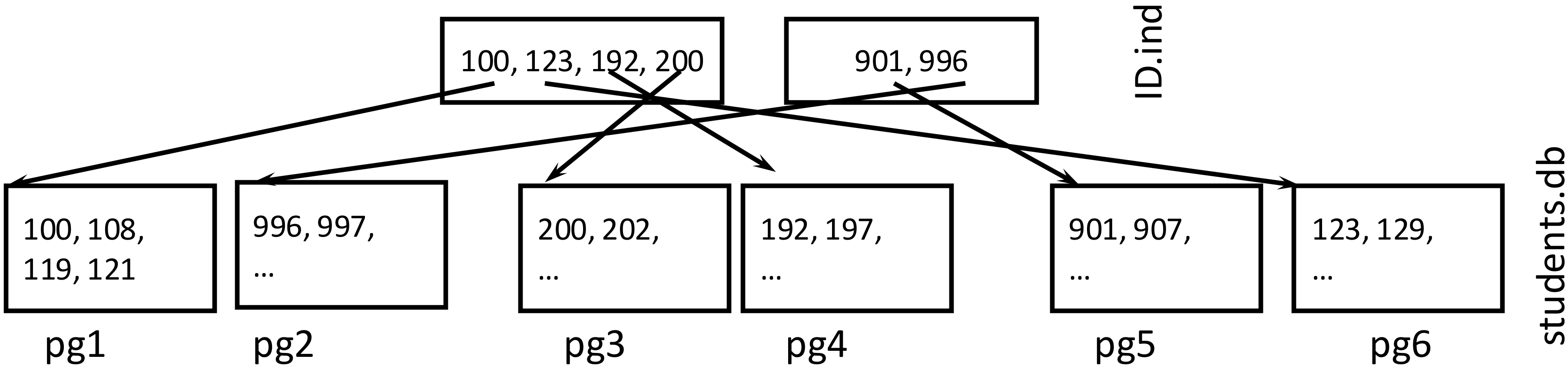
- Suppose the sort column keys have a relatively stable domain and the table is not expected to grow significantly.
- Can do an initial sort upon data ingestion and keep a sparse-index
- *Below: Just showing the sort column not entire rows*
- Lookup: Find page, follow pointer, scan page (& overflow pages (soon))



- Need the data pages sorted (called clustered index)
- Advantage over dense index: much smaller (can be a few orders of magn.)
- Insertions require chaining & deletions can lead to empty pages (soon)

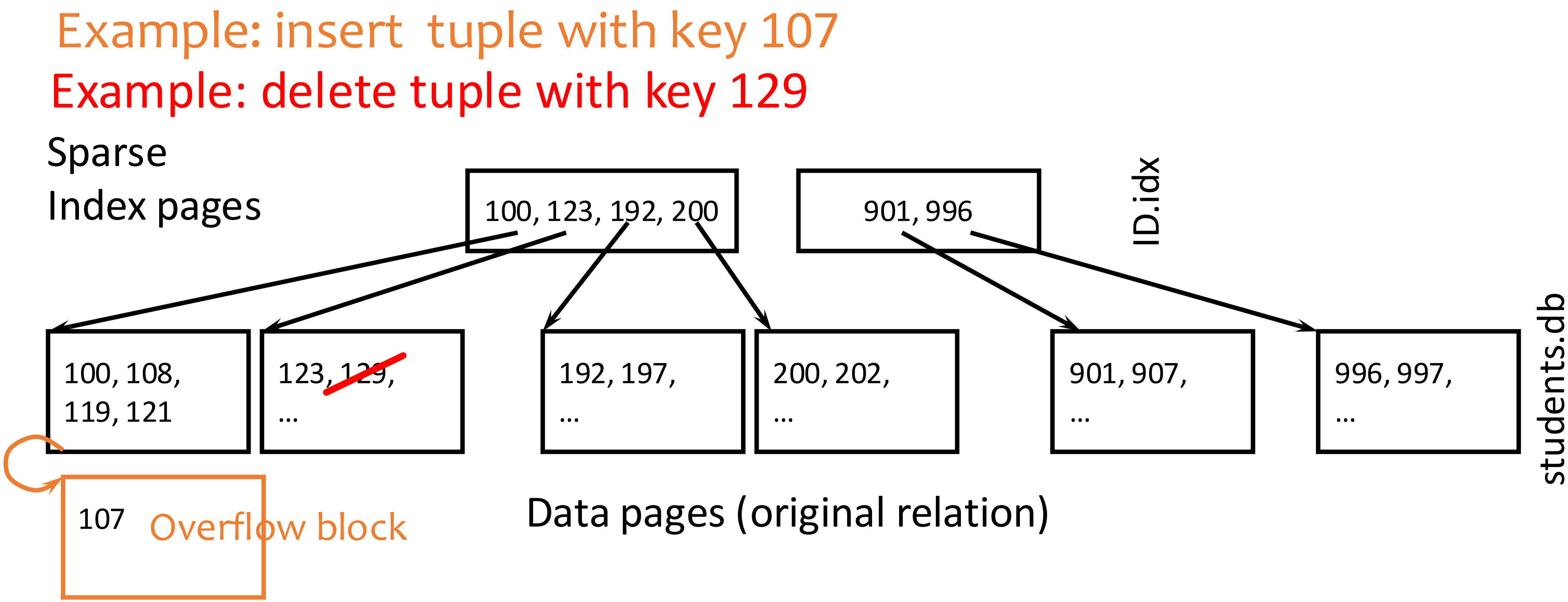
Note on Clustered Indices

- When a relation file has a clustered index, i.e., when pages are sorted, the pages themselves do not necessarily need the pages to be stored sequentially on disk in sort order.
- It is not practical to store pages on disk sequentially in sort order (and this does not decrease I/O, though can make I/Os more “sequential”)



3rd Approach: Single-level Sparse Index w/ Overflows

- Handling Insertions



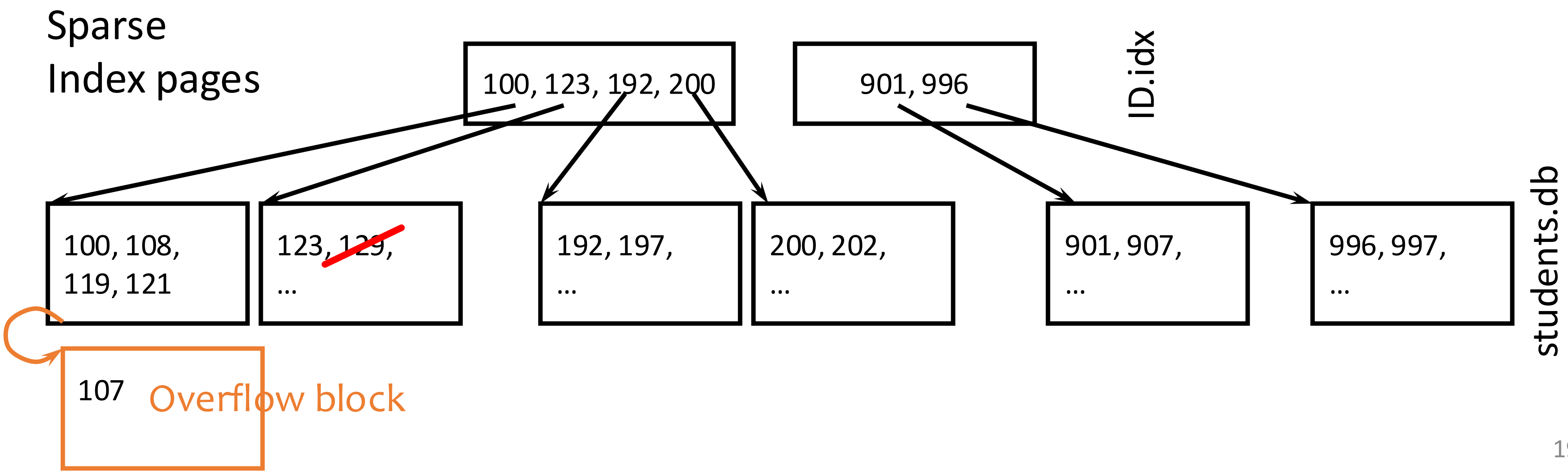
- Overflow chains and empty data blocks degrade performance
  - If there is significant *data distribution skew*: records can go into one long chain, so lookups require scanning all data in worst-case.

3rd Approach: Single-level Sparse Index w/ Overflows

- Pros: Index size smaller than dense index (1 key/ptr in index per page), so can be larger
- Cons:
  - Can still become very large (GBs) for large tables.
  - Need overflows, which is not robust, if table grows significantly over time (e.g., most pages can become overflows, leading to large scans)
  - Can lead to empty pages (but less of an issue in practice)

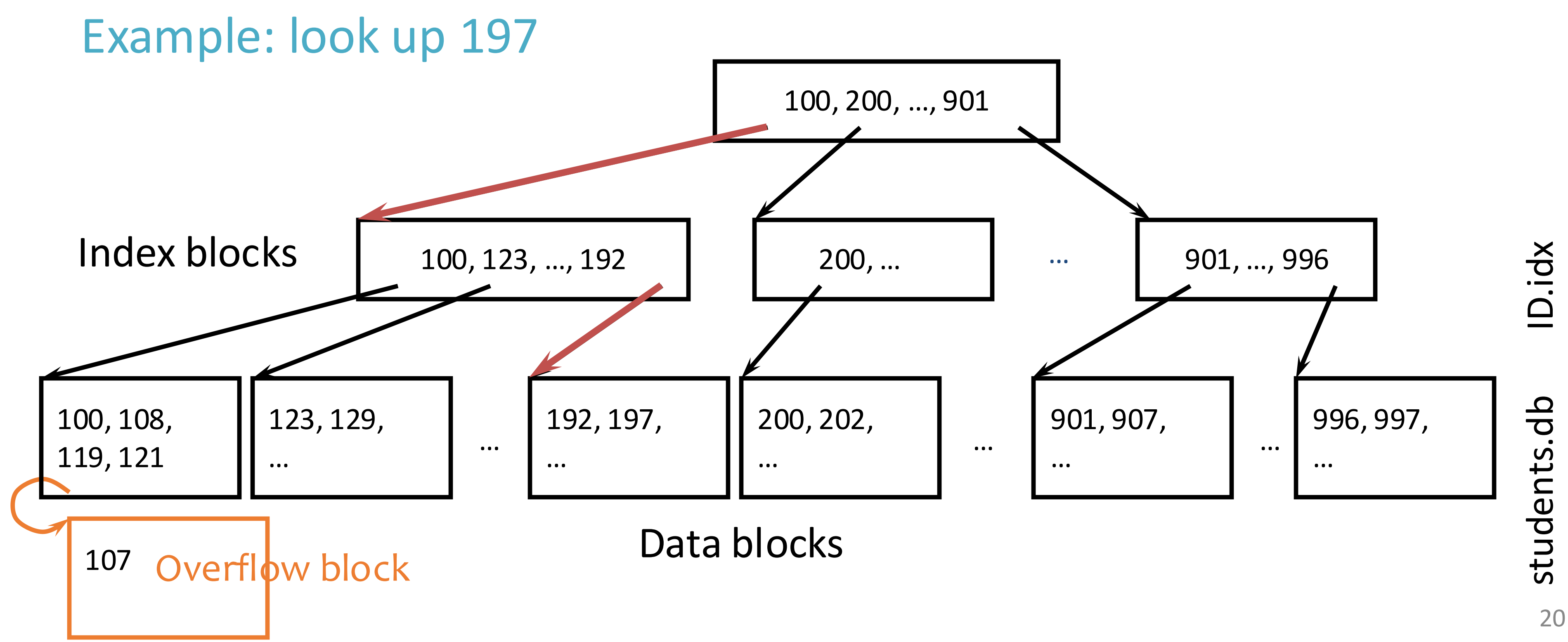
Address w/ multi-level indices

Address w/ splitting/merging



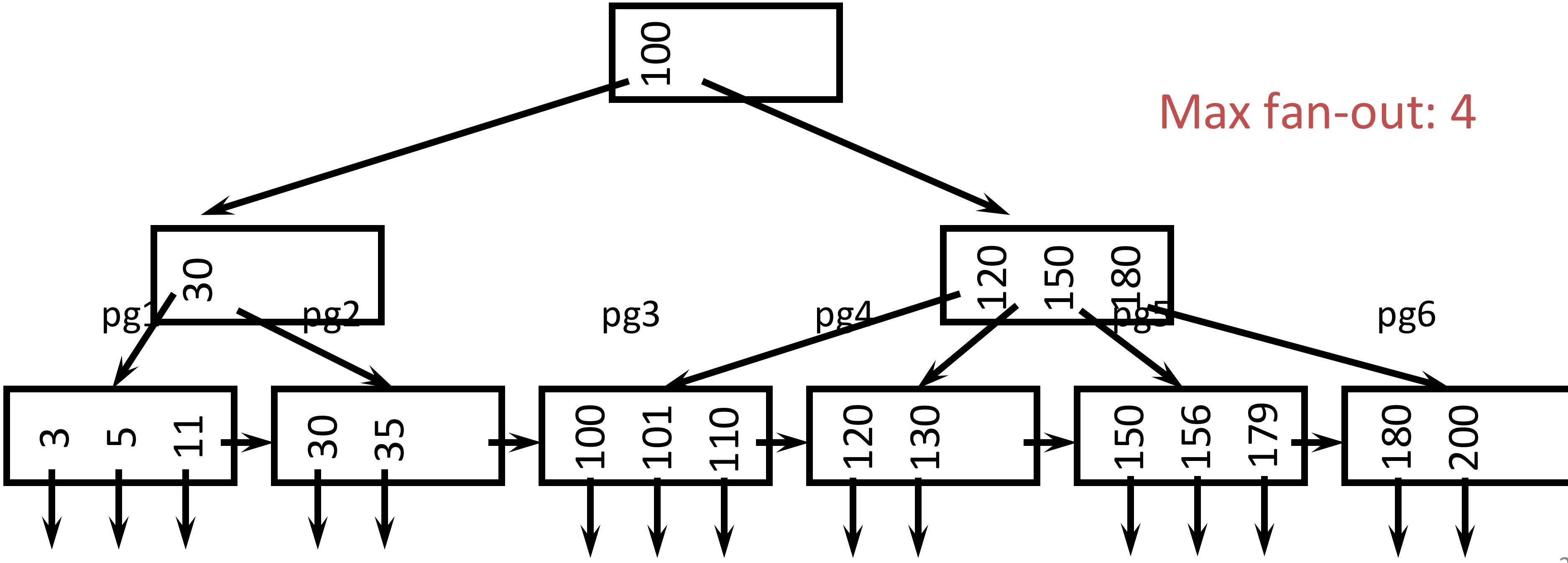
4th Approach: Multi-level Indices

- If an index is too large, can put other layers of sparse indices on the index
- Forms a tree & the system can keep higher-levels of the index in memory
  - Cand do depth-1 many I/Os in lookups (ignoring overflows)

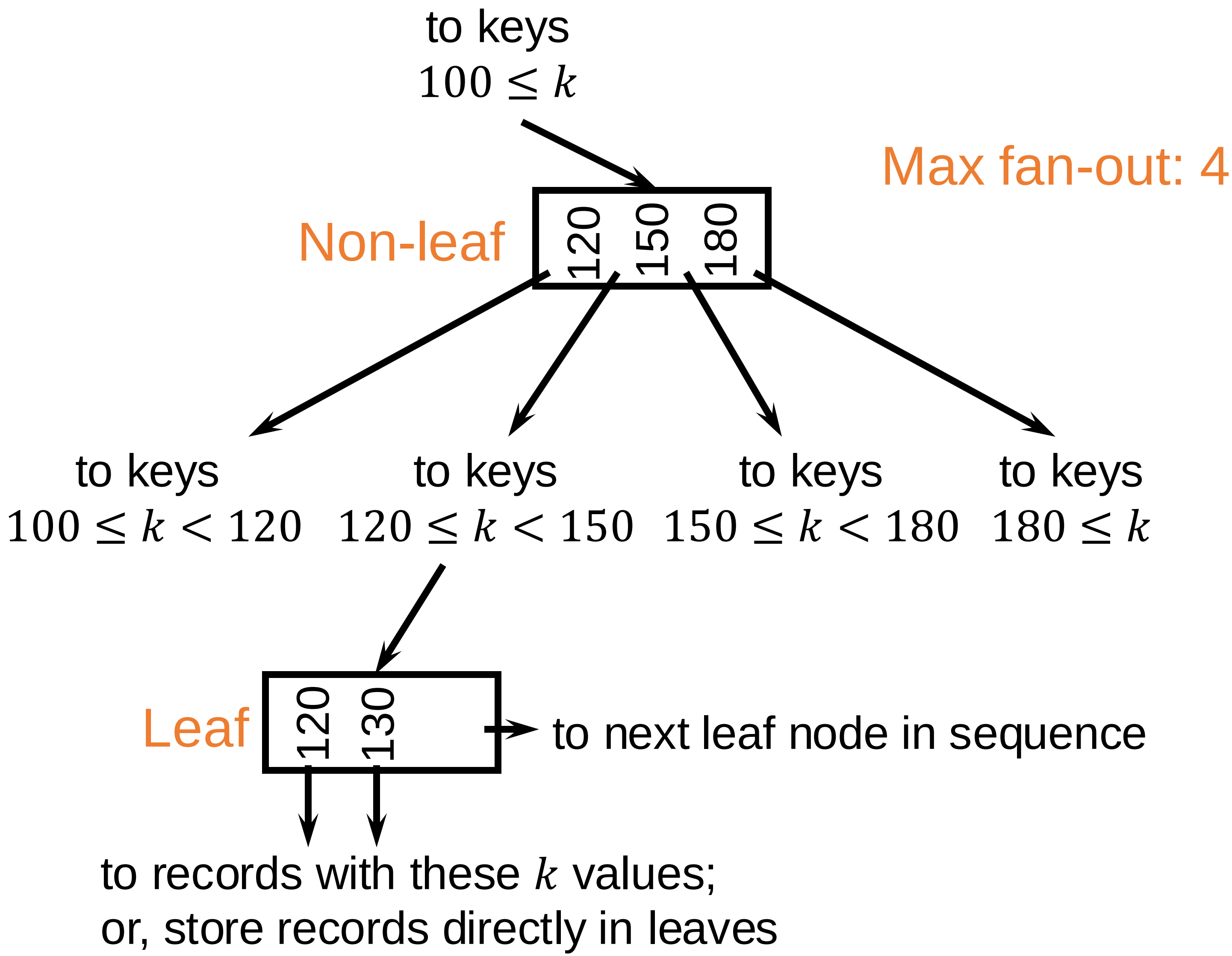


5th Approach: B/B+ Tree Indices

- Multi-level sparse indices on a first level of pages that is either:
  - actual relation pages (if clustered)
  - dense index on the relation pages (if unclustered)
- First level consists of *chained pages*
- Forms a k-ary balanced tree
- Instead of overflow pages uses splitting and merging of pages at any layer



Sample B$^+$-tree Nodes



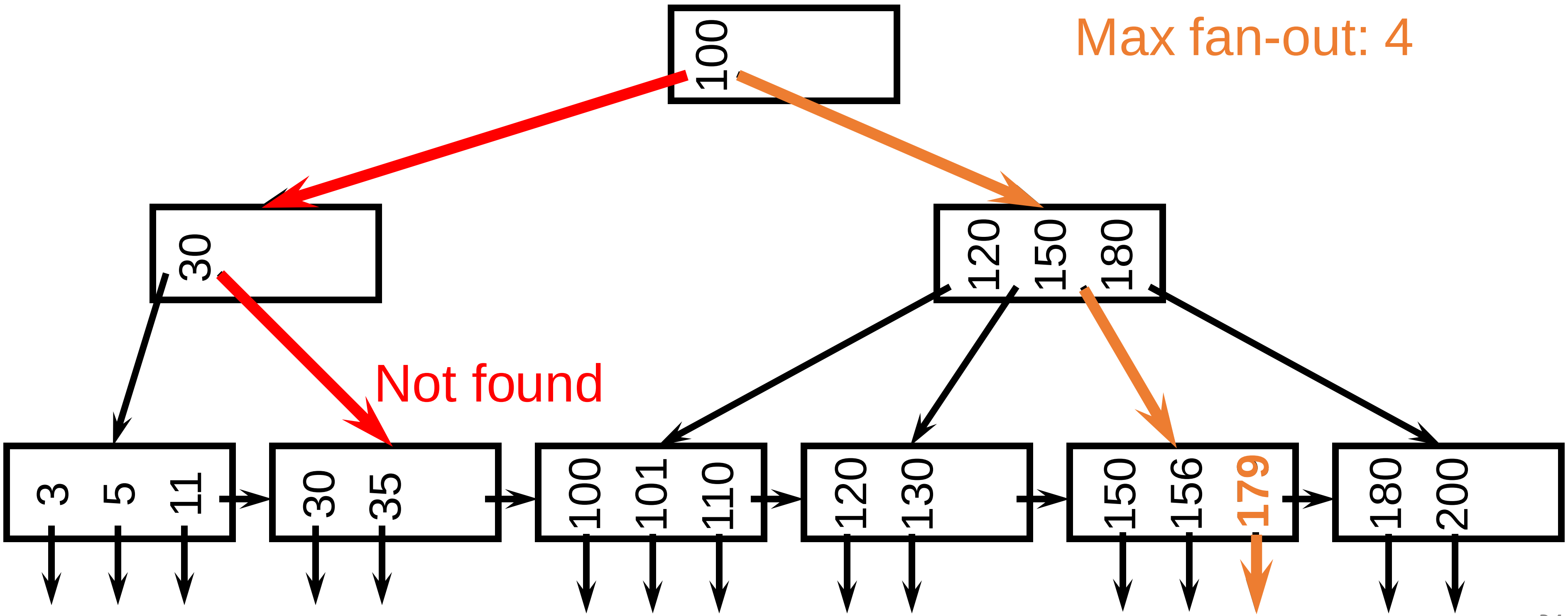
B^+-tree Balancing Properties

- 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	$f-1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

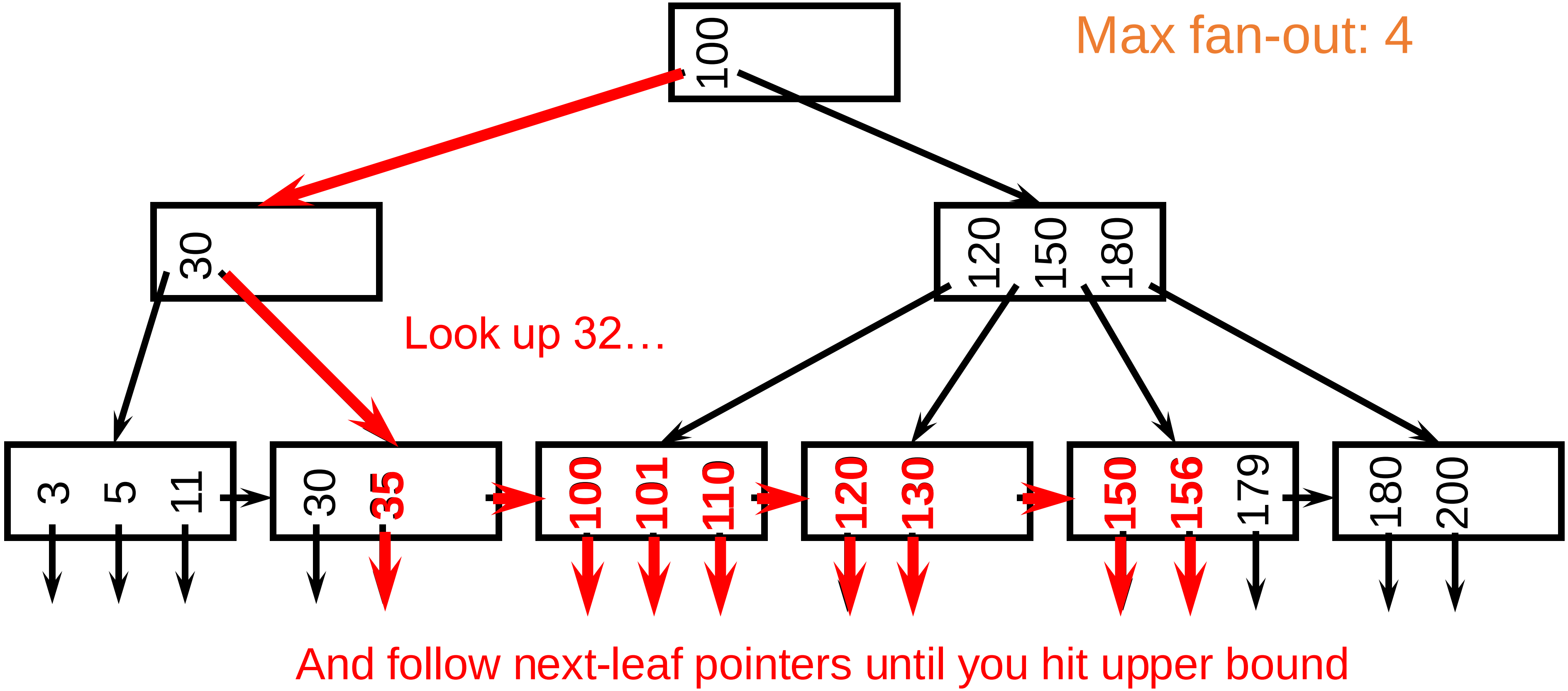
Lookups

```
SELECT * FROM R WHERE k = 179;
SELECT * FROM R WHERE k = 32;
```



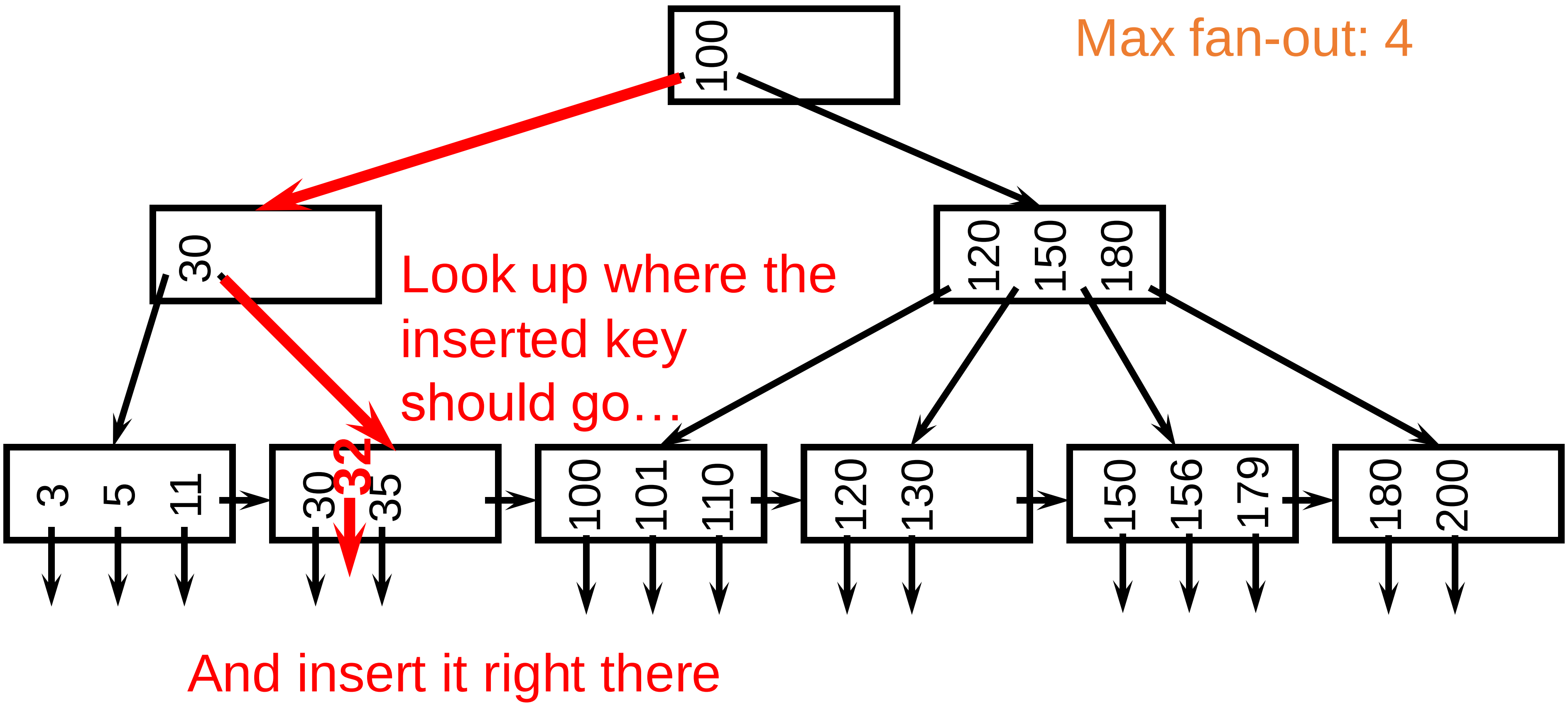
Range Query

- `SELECT * FROM` *R* `WHERE` *k* `> 32 AND` *k* `< 179;`



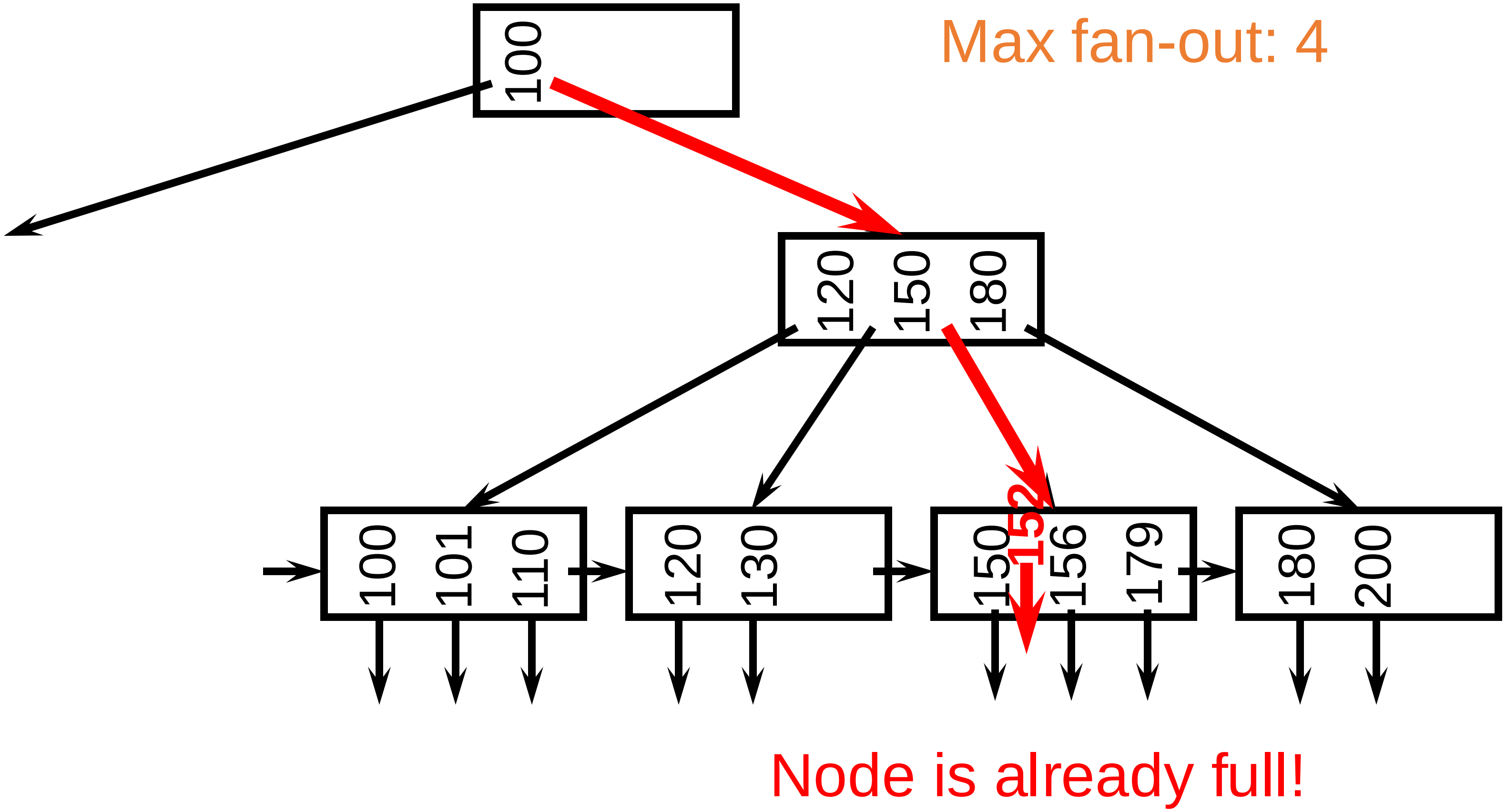
Insertion

- Insert a record with search key value 32

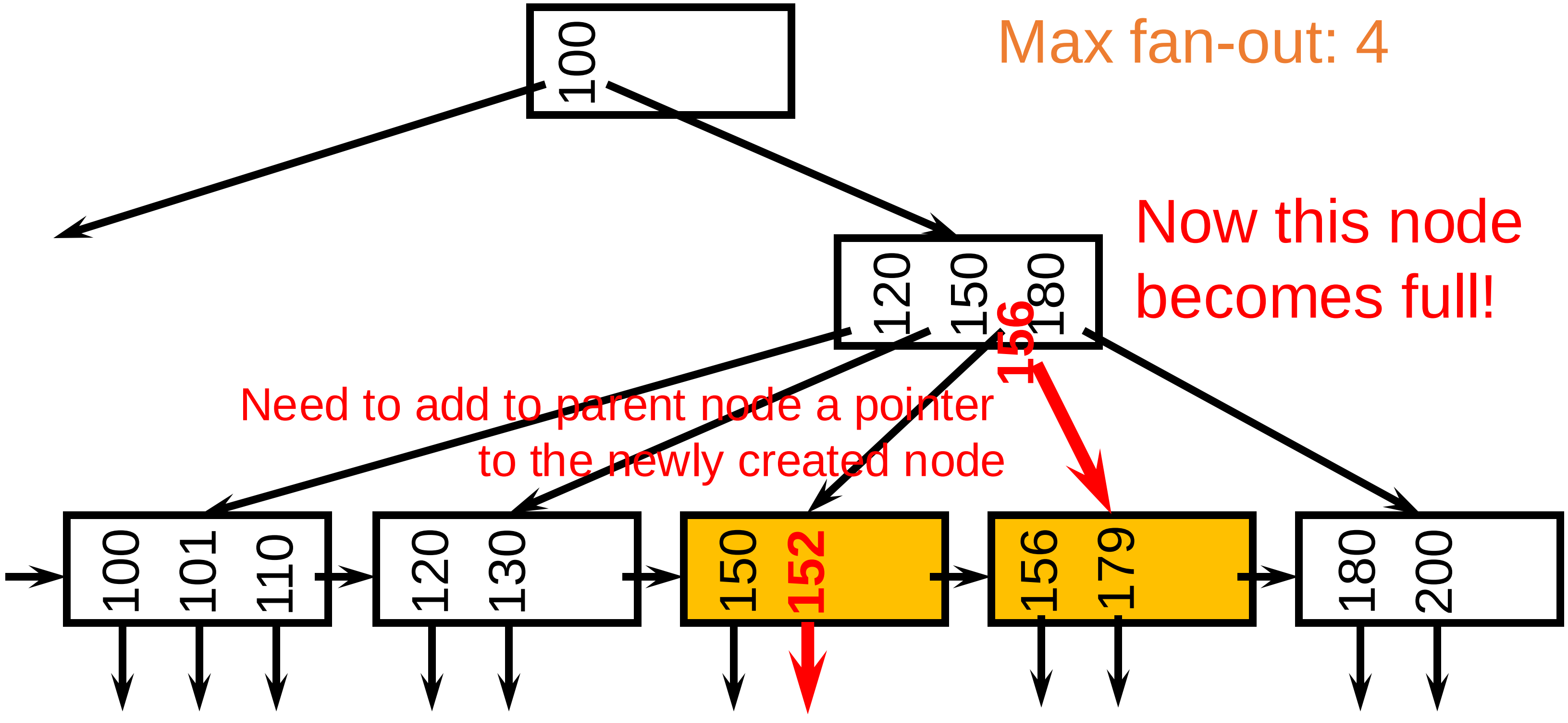


Another Insertion Example

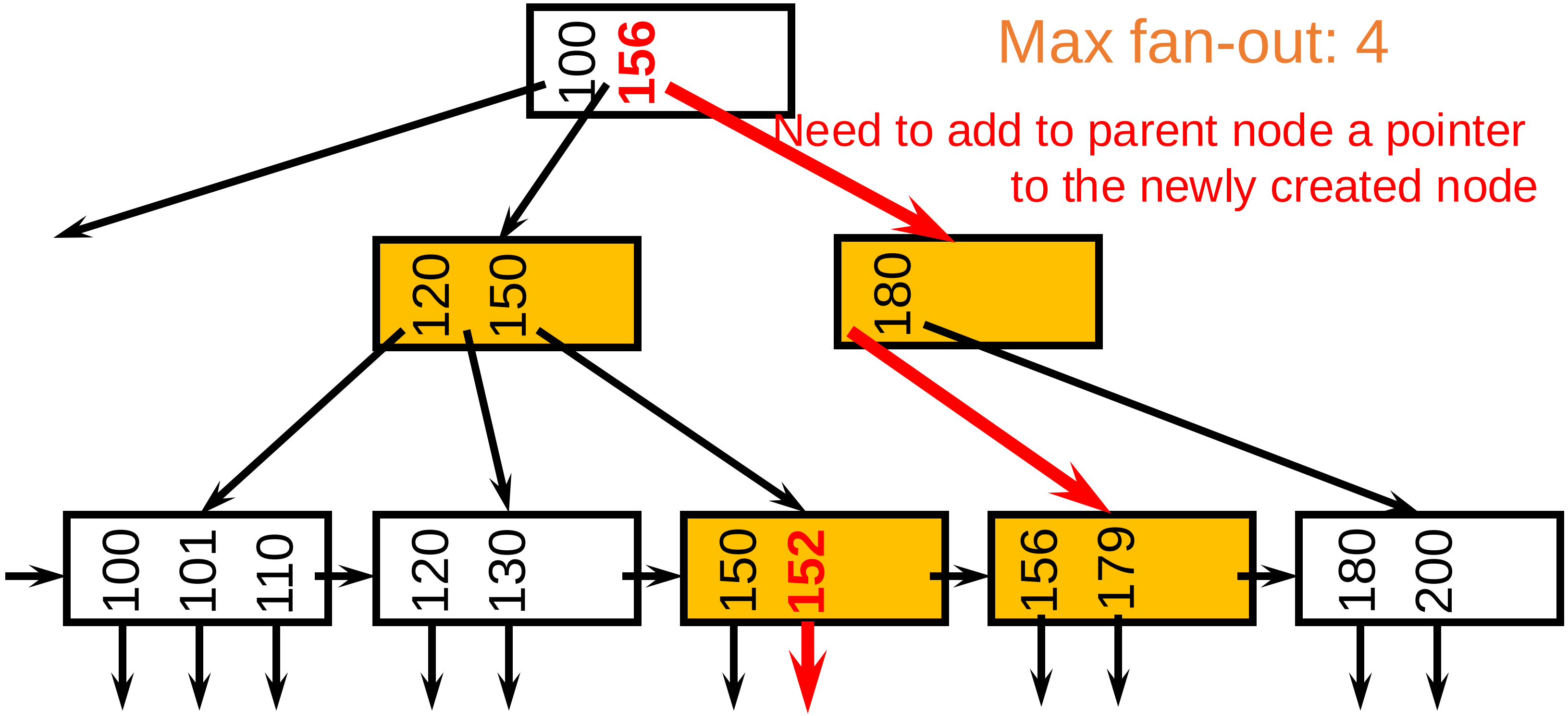
- Insert a record with search key value 152



Node Splitting



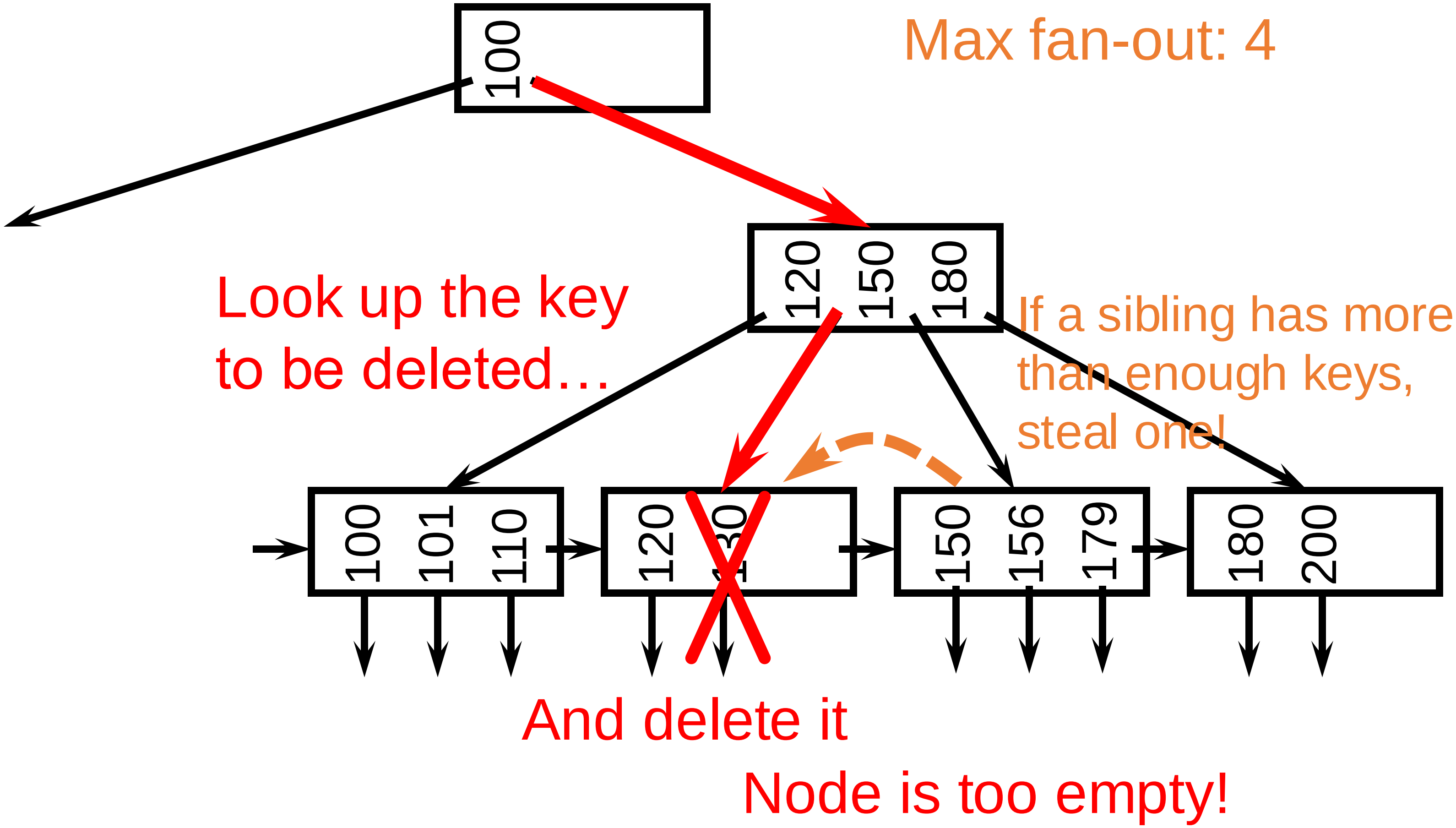
More Node Splitting



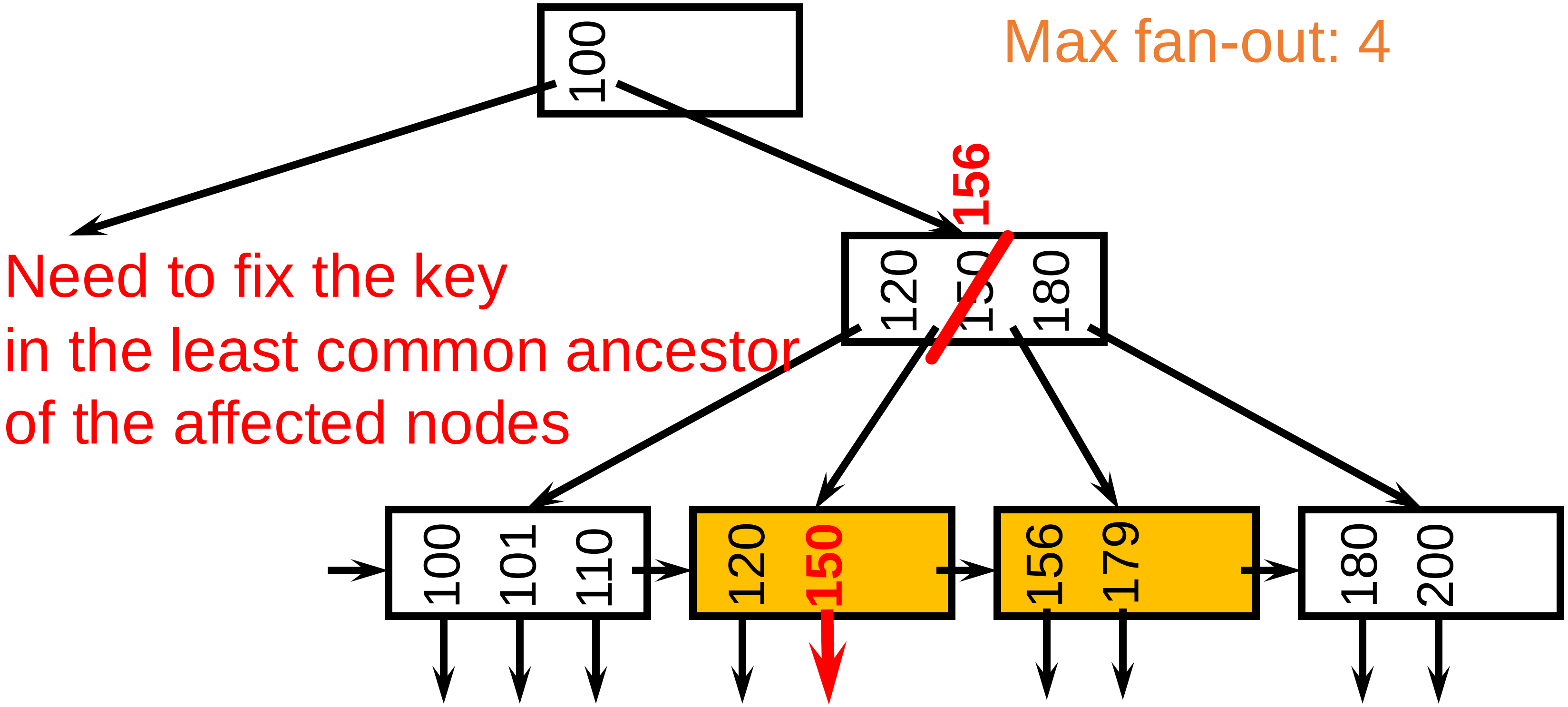
- In the worst case, node splitting can “propagate” all the way up to the root of the tree (not illustrated here)
  - Splitting the root introduces a new root of fan-out 2 and causes the tree to grow “up” by one level (this is why roots can have < f/2-1 keys)

Deletions

- Delete a record with search key value 130



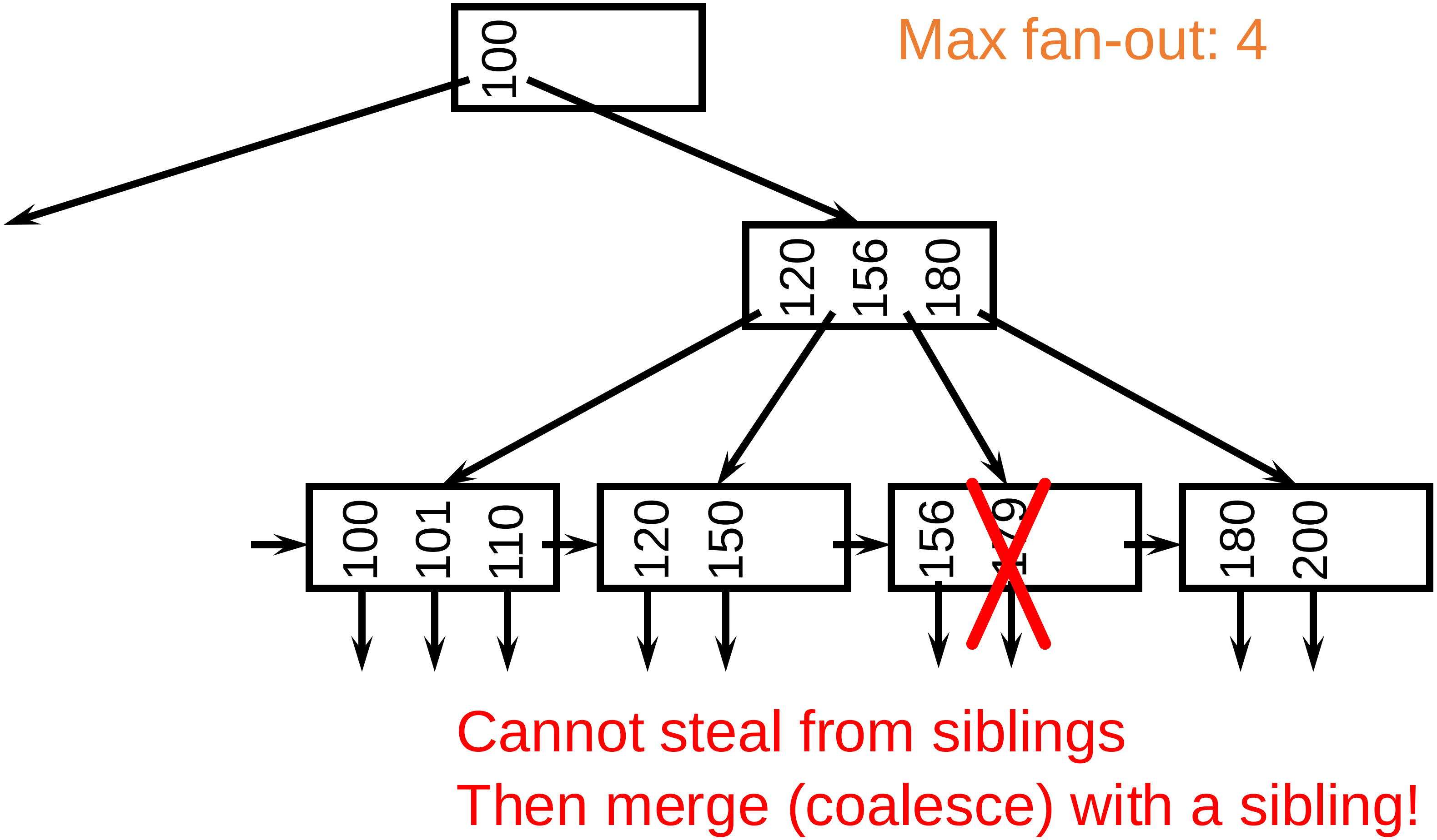
Stealing From a Sibling Node



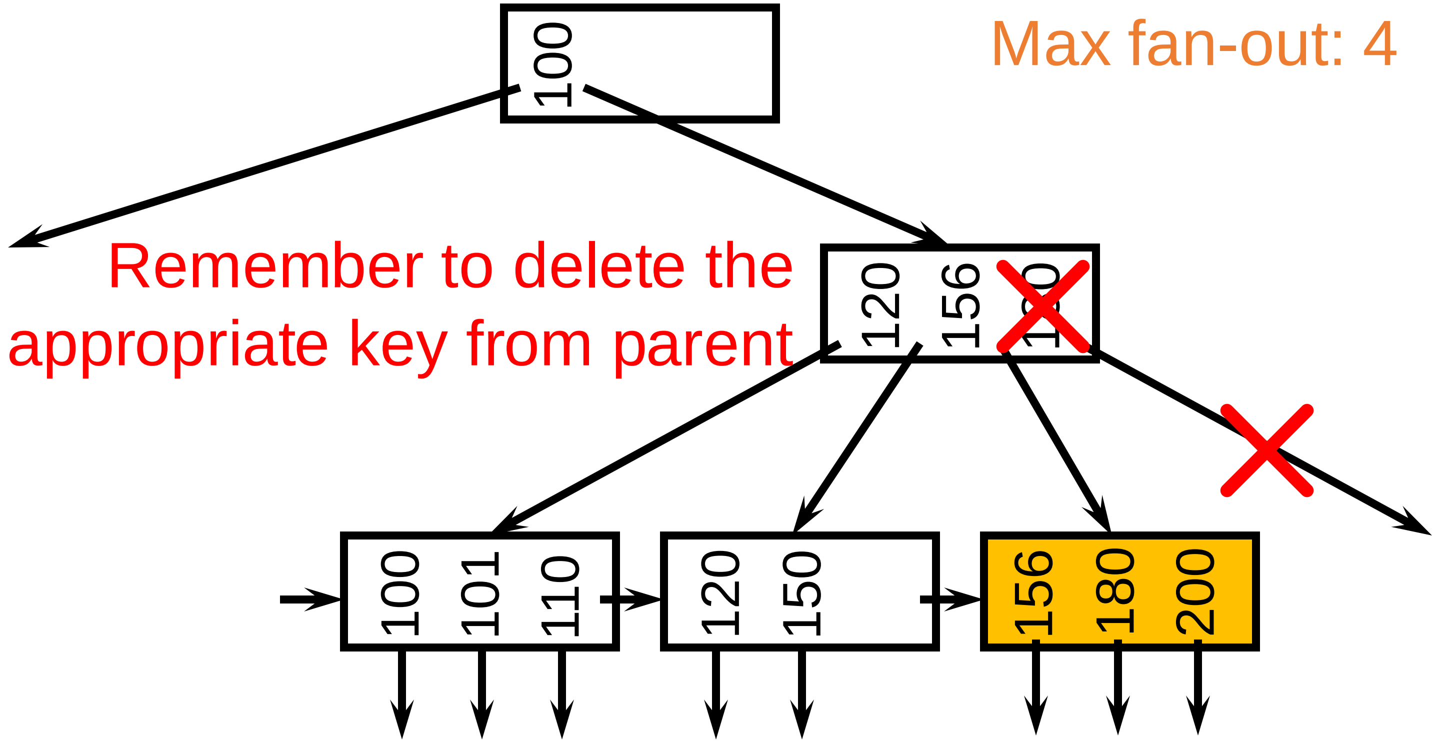
- If you are hacker, encourage you to implement the deletion subroutine of an external B+ tree. Quite challenging!

Another Deletion Example

- Delete a record with search key value 179



Merging



- 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

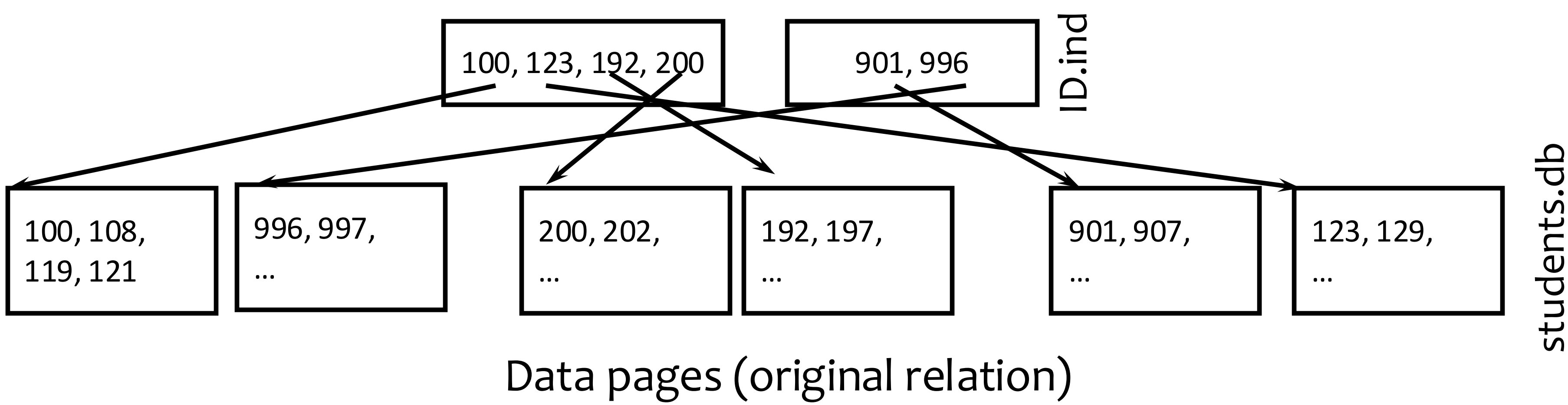
Performance analysis of B^+-tree

- 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 $\log_{\text{fanout}} N$, 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 many tables (e..g, if f=200, then you can accommodate 1.6B rows)

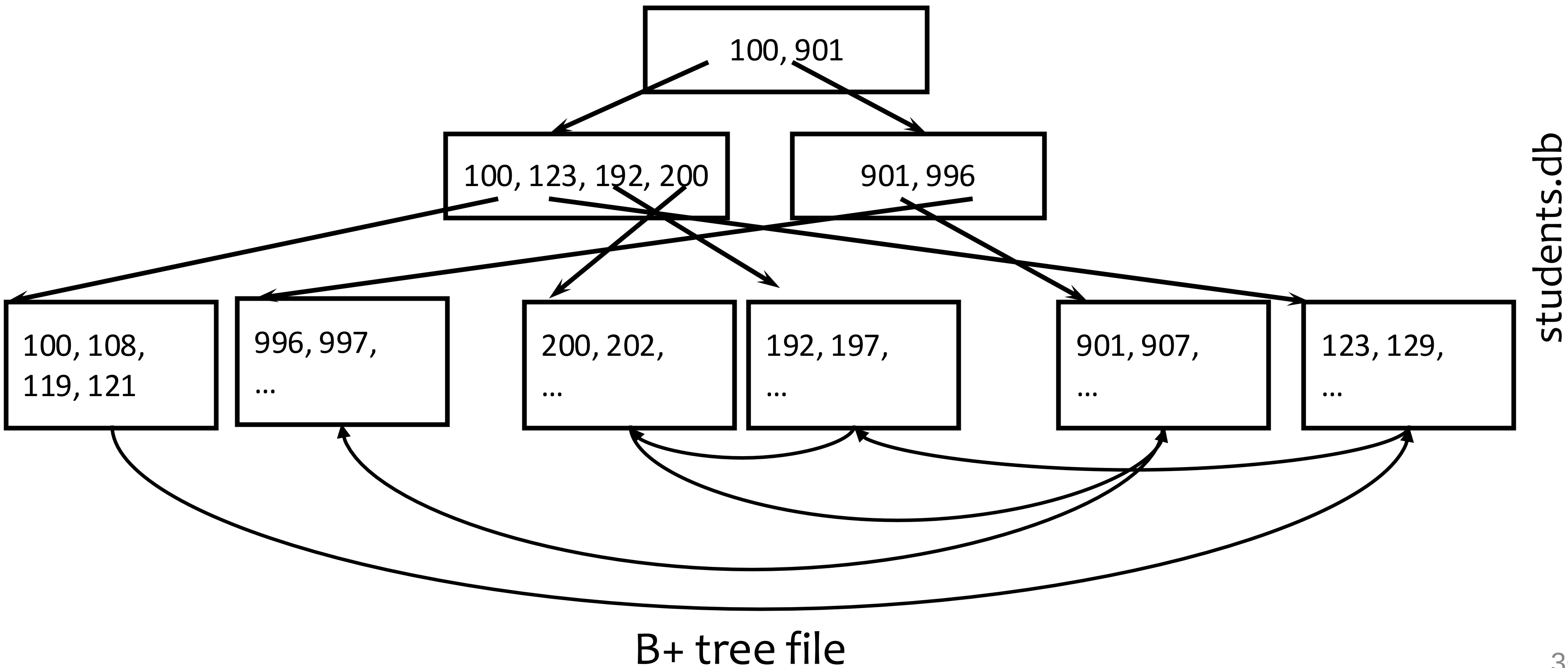
How to Keep A Table Sorted?

- Recall this key question
- Recall further note on clustered indices and page order.



How to Keep A Table Sorted?

- Recall this key question
- Recall further note on clustered indices and page order.
- Again assume leaf nodes are tuples
- Many RDBMSs use "B+ tree files" to store the tables, i.e., entire file is a B+ tree index, with leaf nodes storing tuples (instead of pointers to tuples)



Difference Between B and B+ Tree

- B-tree stores pointers to records in non-leaf nodes too (and does not store these search keys in other non-leaf or leaf nodes)
- Pro: These records can be accessed with fewer I/O's
- Cons:
  - Storing more data in nodes decrease fan-out and increases h
  - Records in leaves require more I/O's to access
  - Vast majority of the records live in leaves!

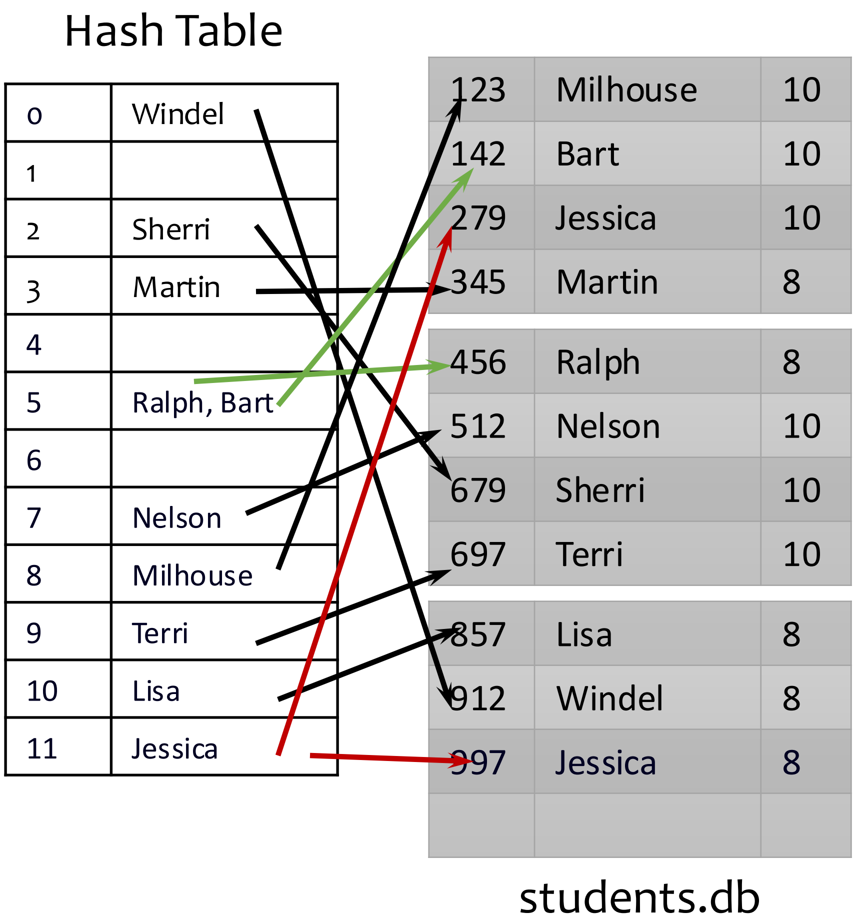
What Does B Stand For?

- No one really knows!
- But Edward M. McCreight, co-inventor with Rudolf Bayer, has a video that says:
  - They never resolved what B is but they had in mind:
  - Boeing, Bayer, and Balance

Other Common Indices

- 2 Classes of Indices Overall

1. Tree-based: can do both lookups and range queries
   - B/B+ Trees, R Trees, Radix Tree
2. Hash-based
   - Can only do look ups. Cannot do range queries.
   - In practice: handle collisions
3. Many other indices: bitmap indices, probabilistic indices, suffix arrays, GiST or Inverted Index for different applications and data types.



Outline For Today

Database Indices

- 5 Index Designs in Increasing Level of Robustness
- Using Indices In Practice

Using Indices In Practice (1)

- Indices can be defined on one ore more attributes:
  - CREATE INDEX NameIndex ON User(Lastname,Firstname);
  - I.e., B+'s keys are (Lastname, Firstname) pairs and tuples are sorted first by LastName and then Firstname.
  - This index would be useful for these queries:

```
select * from User where Lastname = 'Smith'
select * from User where Lastname = 'Smith' and Firstname='John'
```

  - But not this query:

```
select * from User where Firstname='John'
```

- Many systems use indices by default on the primary key
- Many systems use indices to implement UNIQUE constraints

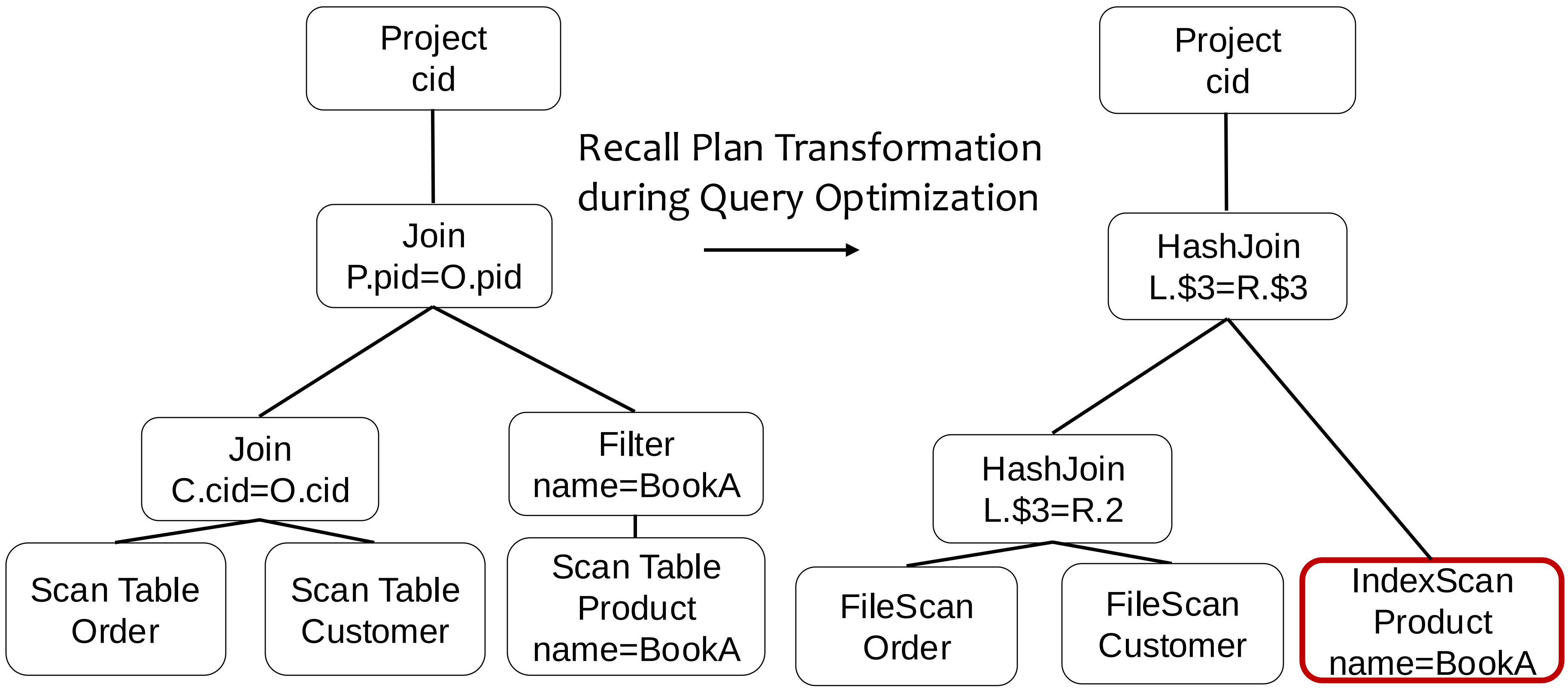
```
CREATE TABLE Students(
    studentID int,
    sinNumber varchar(16) UNIQUE
    PRIMARY KEY (studentID))
```

Will create 2 B+ indices:

1) on studentID;
2) 2) on sinNumber

Using Indices In Practice (2)

- Users only create indices. They do not refer to indices in queries.
- Pro: Some user queries will get much faster
  - B/c RDBMSs use indices during query evaluation
  - Ex: IndexScan operators, or IndexMergeJoin (in Oracle) or IndexNestedLoopJoin etc.



Using Indices In Practice (3)

- Con: Updates will get slower because indices need to be maintained
- Q: How should users pick indices given a workload W (i.e., the set of queries an application asks and their frequencies)
- General Guideline:
  - Profile slow queries. Check if they have =, <, ≤, >, ≥ predicates

```
SELECT * FROM R WHERE A = value;
SELECT * FROM R WHERE A = value AND B = 27;
SELECT * FROM R, S WHERE R.A = S.C;
SELECT * FROM S WHERE D > 50;
```

  - E.g., above indices on R.A, R.A and R.B multicolumn, S.C, S.D are possible indices that can speed queries
  - But one should weigh these benefits against slow downs due to updates

Using Indices In Practice (4)

- Many RDBMSs have “Physical Design Advisor” (PDA) tool
- Input: Database D (w/ existing indices), workload W
- Output: A set of recommended indices
- Internally PDA does a “what if” analysis:
  - Uses Query Optimizer & inspects the estimated runtimes/costs of plans the system would use for queries in W with & without additional indices