THE UNIVERSITY of EDINBURGH

Advanced Databases

Spring 2020

Lecture #04:

Introduction to Indexing

Milos Nikolic

File Organization

Method of arranging a file of records on external storage

Heap files

store records in no particular order

Sorted files

store records in sorted order, based on the value of the search key fields

B+ tree files

support ordered access to record even if there are inserts/deletes/updates

Hashing files

place records based on a hash function computed on record attribute(s)

Comparing File Organizations

Simplistic, but effective **I/O only cost model**

P = The number of data pages

D = (Average) time to read or write disk page

Considered operations

Scan all records in a given file

```
SELECT * FROM R
```

Search with equality test

```
SELECT * FROM R WHERE C = 42
```

Search with range selection

```
SELECT * FROM R WHERE A > 0 AND A < 100
```

Insert a given record in the file

Delete a record by rid

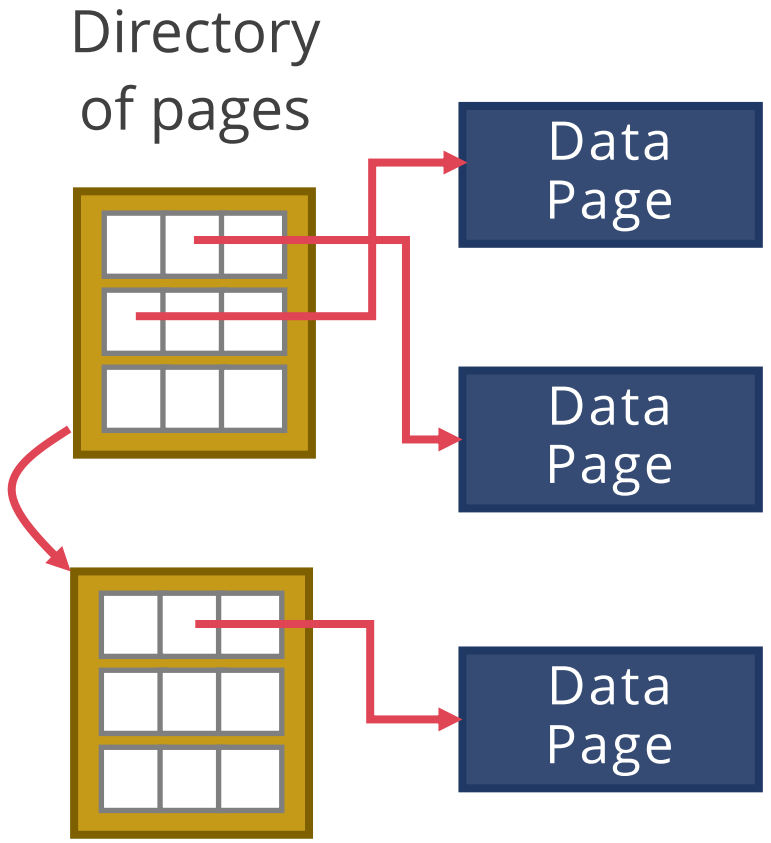
Heap Files

A heap file provides just enough structure to maintain a collection of records of a relation

The heap file only supports sequential scans, no specific support for further DB operations

SQL query efficiently executable using sequential scan

```
SELECT A, B FROM R
```



Heap File Analysis

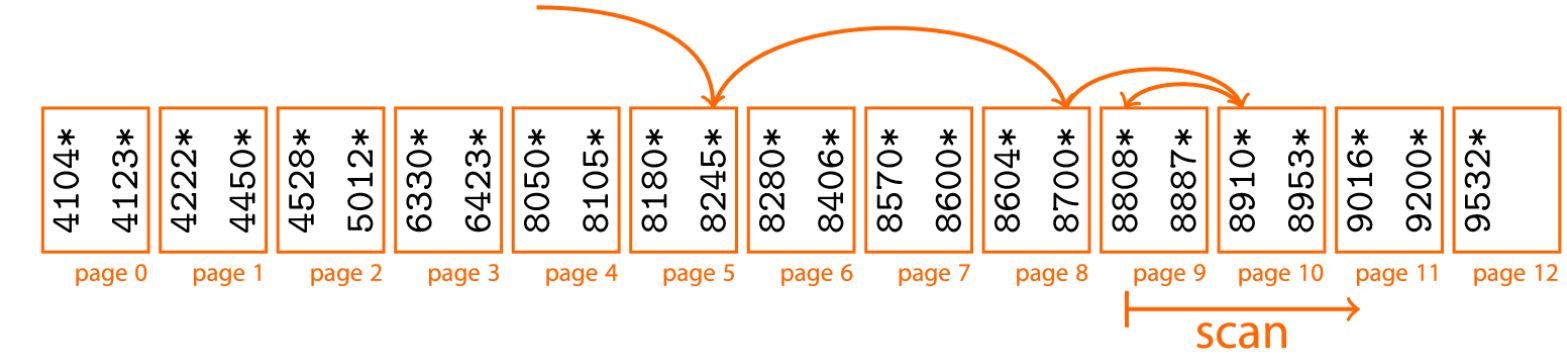
scan		iterate over all pages (linked or via directory)	$P \cdot D$
search	on key	scan the file until found	$0.5P \cdot D$
	on non-key	scan the file until end	$P \cdot D$
range query		same as scan	
delete		search, delete from page, and write page	$0.5P \cdot D + D$
insert		"just stick it at the end" (read last page, add, write)	$2D$

P = The number of data pages

D = (Average) time to read or write disk page

Sorted Files

Store records sorted by look-up attributes, no gaps



Fast sequential scan + records in sorted order (can be big plus)

Use binary search if the search condition matches the sort order

Insertion and deletions are slow: shift all subsequent records

Rarely used: unpredictable jumps defeat the purpose of prefetching

Sorted File Analysis

scan		iterate over all pages	$P \cdot D$
search	on key	binary search	$\log_2 P \cdot D$
	on non-key	binary search and search all matching records	$D \cdot (\log_2 P + \#\ pgs\ with\ match\ recs)$
range query		similar as search on non-key	
delete		search, delete, shift	search + $P \cdot D$
insert		search, insert, shift	search + $P \cdot D$

P = The number of data pages

D = (Average) time to read or write disk page

INDEXES

An **index** is a data structure that organizes data records to efficiently retrieve all records matching a given **search condition**

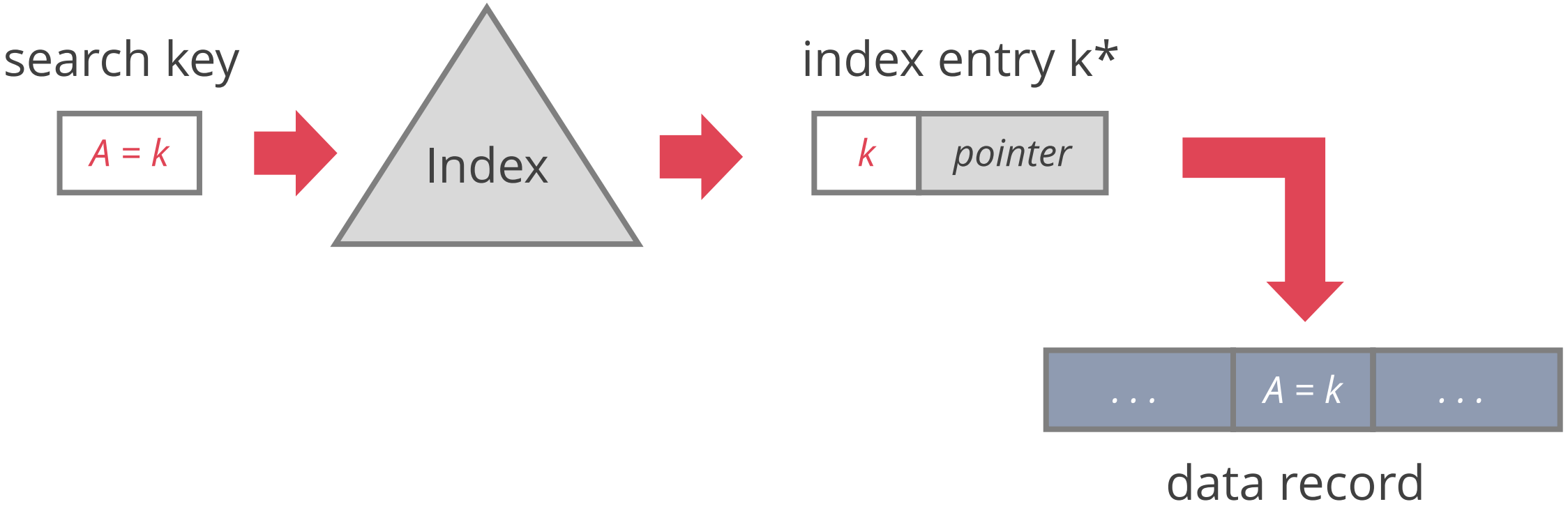
Search key = fields used to look up records in a relation

- Can be any subset of the fields of a relation
- **Not** the same as **key** (minimal set of fields that uniquely identify a record)

```
CREATE INDEX idx1 ON users USING btree(sid)
CREATE INDEX idx2 ON users USING hash(sid)
CREATE INDEX idx3 ON users USING btree(age,name)
```

Index Usage

An index contains a collection of **index entries** and supports efficient retrieval of all index entries **k*** with a given key value **k**



Index Entries

We can design the index entries (**k***) in various ways

Variant A

Variant B

Variant C

A: no need to store the data records in addition to the index

to avoid redundant storage of records at most one index on a file can use **A**

B and **C** use rids to point into the actual data file

Index Entries

Variant choice is orthogonal to the type of index (B+ trees, hash)

Variant A

Variant B

Variant C

B and **C** have index entries typically much smaller than data records

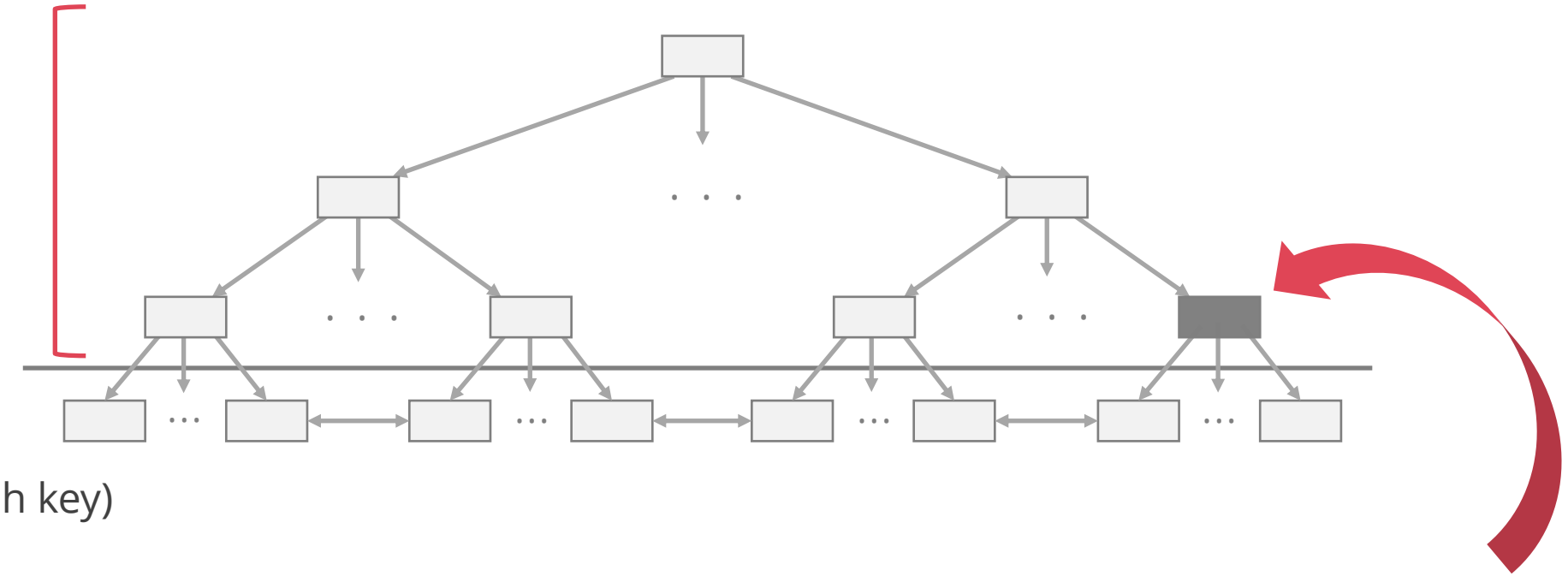
C leads to less index entries if multiple records match a search key *k*, but index entries are of variable length

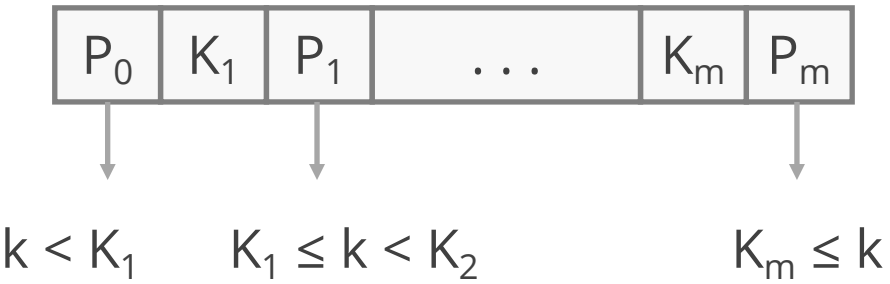
B+ Tree Indexes

Non-leaf pages

Leaf pages

(sorted by search key)

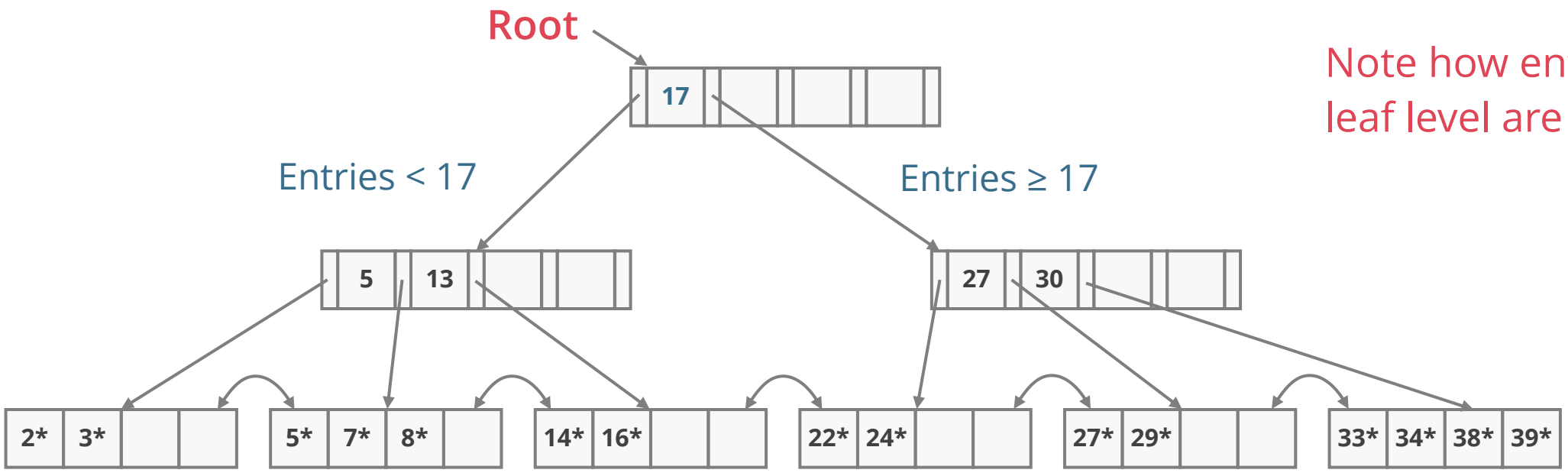




Non-leaf pages only direct searches

Leaf pages are doubly linked

Example B+ Tree



Note how entries in leaf level are sorted

Find 28*? Find 29*? Find all entries > 15* and < 30*

Insert/delete: Find data entry in leaf, then change it

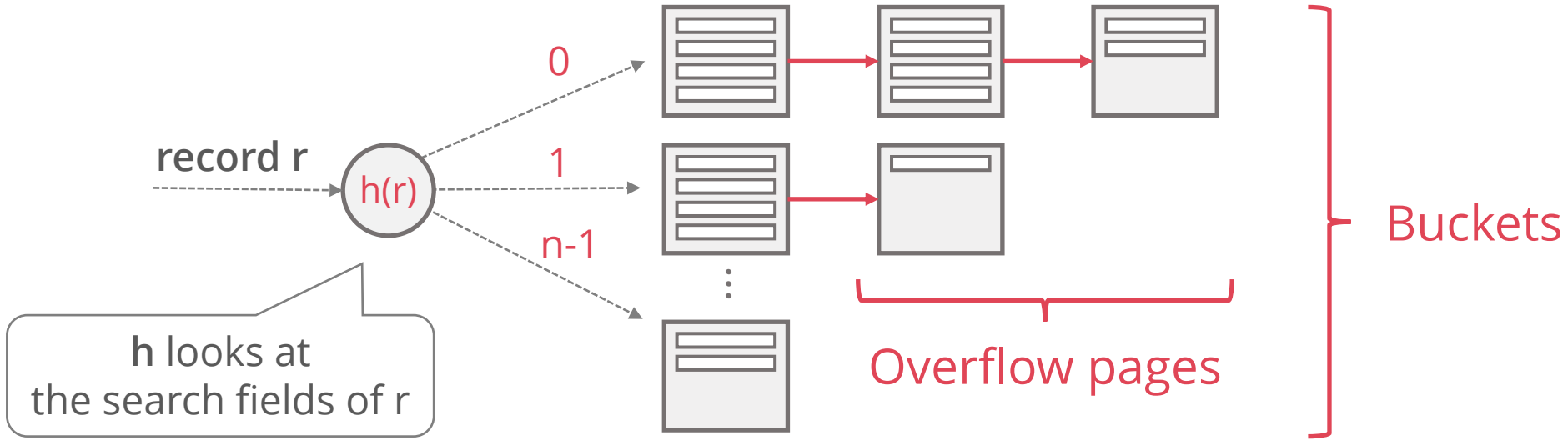
Need to adjust parent sometimes, and change sometimes bubbles up the tree

Hash-Based Index

Index is a collection of **buckets**

- Bucket = **primary page** plus zero or more **overflow pages**
- Buckets contain index entries

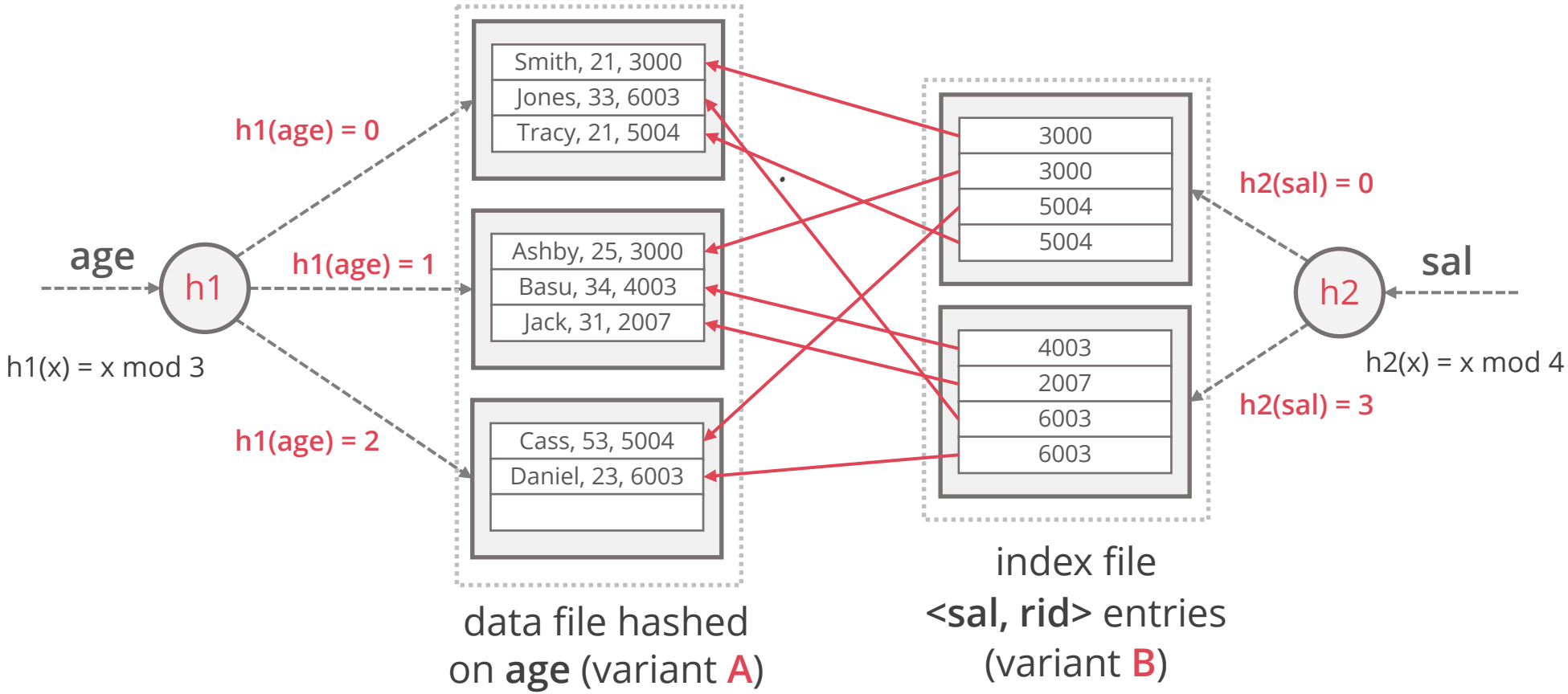
Hashing function h: h(r) = bucket in which record r belongs



Example Hash-Based Index

Employee(name, age, sal)

Efficient equality searches on the **age** and **sal** keys



Hash-Based Index

Good for equality searches

- Hash the search key to its bucket and scan it
- Inserts/deletes are generalization of lookups

```
SELECT * FROM R WHERE A = 10
```

Range queries are not supported

```
SELECT * FROM R WHERE A > 10
```

Hashing File Analysis

scan		iterate over all pages	$P \cdot D$
search	on key	use hash to retrieve page, find record in page	0.5 · # pages in bucket · D
	on non-key	use hash to retrieve page, find record in page	# pages in bucket · D
range query		as heap file, no help from hash (scatters records)	
delete		search, delete, write	search + D
insert		search, insert, write	search + D

P = The number of data pages

D = (Average) time to read or write disk page

Index Classification

Clustered vs. unclustered

Clustered = order of data records is same as, or ‘close to’, order of index entries

Primary vs. secondary

Primary index = search key contains primary key

Unique index = search key contains a candidate key

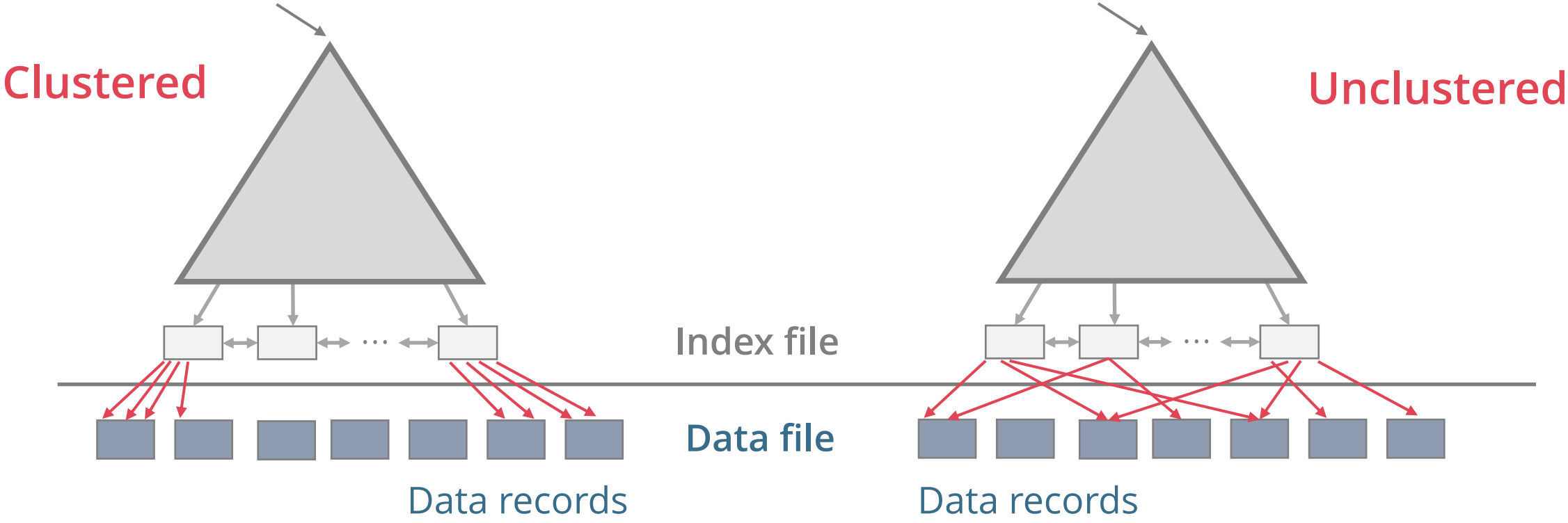
Secondary index = search key may match multiple records

Dense vs. sparse

Dense index = index entry appears for every search-key value

Sparse index = contains index entries for only some search-key values

Clustered vs. Unclustered Index



Variant A implies clustered; in practice, clustered also implies A (since sorted files are rare)

Clustered vs. Unclustered Index

A file can be clustered on at **most one** search key

Cost of retrieving records can **vary greatly** based on index type!

- Clustered: I/O cost = # pages in file with matching records
- Unclustered: I/O cost ≈ # of matching **leaf index entries**

What are the trade-offs?

- Clustered indexes are efficient for range searches
- BUT more expensive to maintain (overflow pages may be needed for inserts)

Dense vs. Sparse Index

We can design a clustered index to be more **space efficient**!

Sparse index: keep the size of the index small by

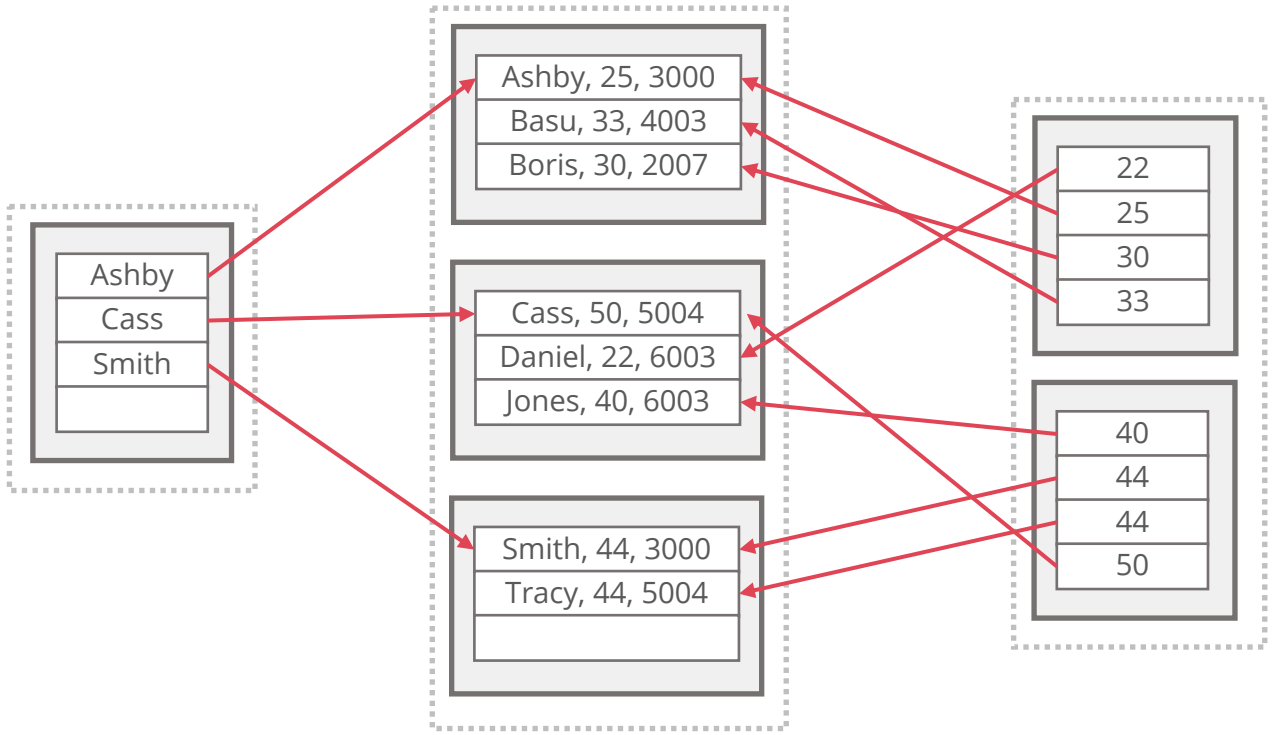
- Maintaining one index entry **k*** per data file page (not per data record)
- Key **k** is the smallest key on that page

Sparse indexes need 2-3 orders of magnitude less space than dense indexes (consider # of records / page)

Dense vs. Sparse Index

Employee(name, age, sal)

clustered sparse index on **name** (variant **B**)



unclustered dense index on **age** (variant **B**)

Dense vs. Sparse Index

To search a record with field **A = k** in a sparse A-index

- Locate the largest index entry **k'*** such that **k'** <= **k**
- Access the page pointed to by **k'***
- Scan this page (and the following pages, if needed) to find records **<..., A = k, ...>**
  - Since the file is clustered (i.e., sorted) on field **A**, we are guaranteed to find matching records in the proximity

We **cannot** build a sparse index that is unclustered

- There is at most one sparse index per file

When and How to Index?

Some indices are created automatically by the DBMS

- e.g. whenever you set a primary key

Trade-off: Indexes can make **queries faster**, but **updates slower**

- require **disk space**, too

For which attributes should we create an index?

For each index, what kind of an index should it be?

Which index should be **primary**?

- used to organise the data on disk
- **clustered** index if the data is organised in a meaningful way (e.g., sorted in every cluster)

Naïve Index Selection

```
get query workload
    group queries by type
    for each query type in order of importance
        calculate best cost using current indexes
        if new index IDX will further reduce cost
            create IDX
```

High-Level Guidelines

Check the WHERE clause

- attributes in WHERE are search/index keys
- equality predicates → hash index (or tree index)
- range predicates → (clustered) tree index

Multi-attribute search keys supported

- order of attributes matters for range queries
- may enable **index-only** queries that do not look at data pages

Index-Only Plans

Queries might be answered without retrieving any tuples from one or more of the relations if a suitable index is available

```
SELECT E.dno, COUNT(*)
  FROM Emp E
GROUP BY E.dno
```

Index on E.dno

```
SELECT E.dno, MIN(E.sal)
  FROM Emp E
GROUP BY E.dno
```

Tree index on <E.dno, E.sal>

```
SELECT AVG(E.sal)
  FROM Emp E
 WHERE E.age = 25
   AND E.sal BETWEEN 3000 AND 5000
```

Tree index on
<E.age, E.sal> or <E.sal, E.age>

Index-Only Plans

Index-only plans can also be found for queries involving more than one table

e.g. Index Nested Loop join (more on this later)

```
SELECT D.mgr
  FROM Dept D JOIN Emp E
    ON D.dno = E.dno
```

Index on E.dno

```
SELECT D.mgr, E.eid
  FROM Dept D JOIN Emp E
    ON D.dno = E.dno
```

Index on <E.dno, E.eid>

Conclusion

Index is a data structure used to speed up read queries

Tree vs. hash indexing

- Hash-based index only good for equality search
- Tree index is always a good choice

Clustered vs. unclustered index

- At most one clustered index per table
- Multiple unclustered indexes are possible

Index selection depends on query workload

- Build indexes to support index-only strategies