Files, Indexes, Query Optimization (very briefly)

Query Optimization … Overview

- Which query plan is the fastest?
- How many query plans are there?
  (Can we enumerate them?)
- How can we estimate the cost of a plan?
  What units do we use – for cost? (Time? Space? ???)
- But wait, how are queries (query operators) implemented?
- But wait, how are the files stored?

Query Optimization

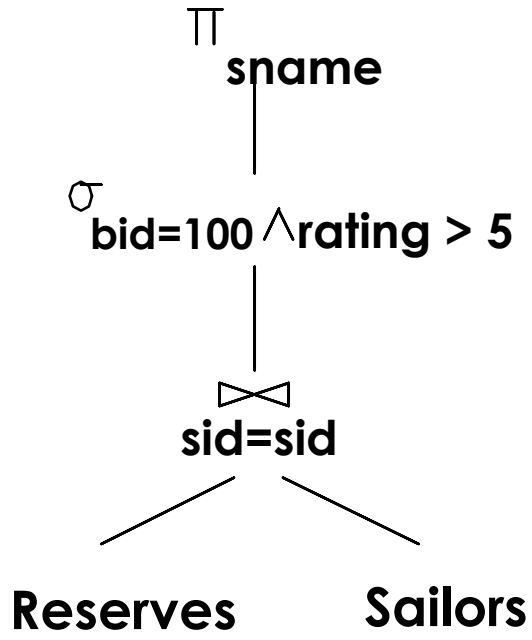
- Translate SQL query into a query tree
  (operators: relational algebra plus a few other ones)
- Generate other, equivalent query trees
  (e.g., using relational algebra equivalences)
- For each possible query tree:
  - select an algorithm for each operator (producing a query *plan*)
  - estimate the cost of the plan
- Choose the plan with lowest estimated cost - of the plans considered (which is not necessarily all possible plans)

Initial Query Tree - Equivalent to SQL
(without any algorithms selected)

SQL Query:

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid AND
        R.bid = 100 AND
        S.rating > 5;
```

Relational Algebra Tree:



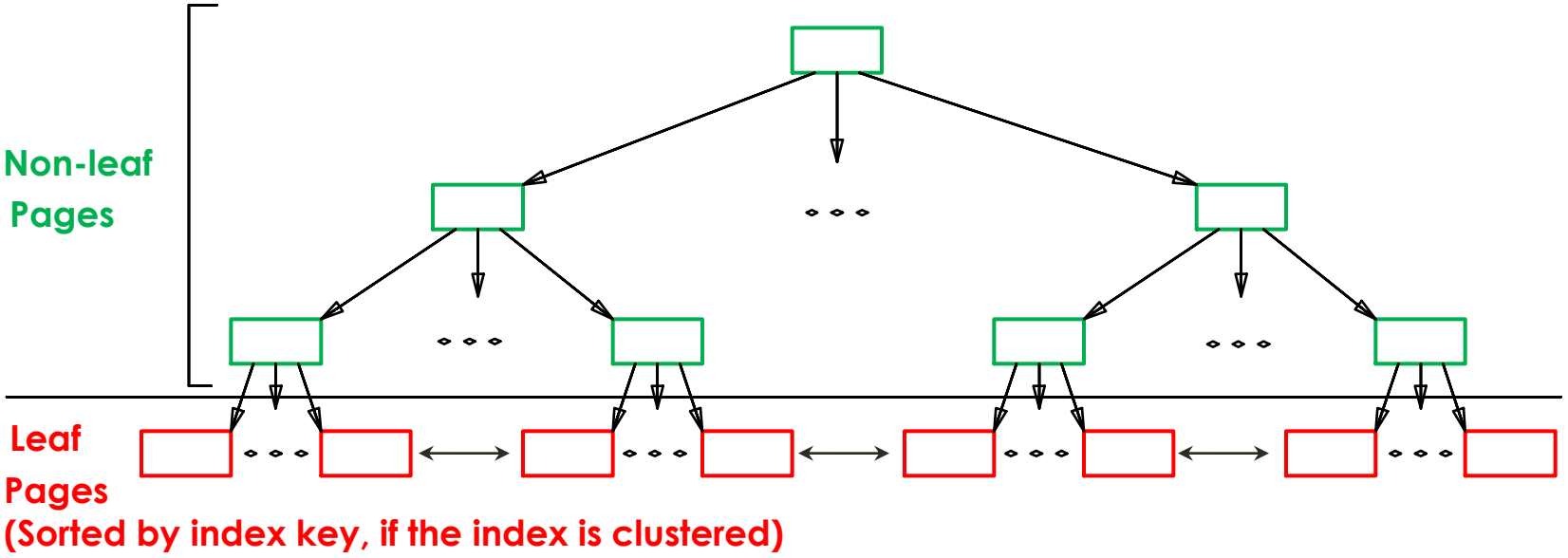
Relational Algebra Equivalence: Cascade of (and Uncascade of) Selects

- $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$
- This symbol means equivalence.
- So you can replace $\sigma_{c1}(\ldots \sigma_{cn}(R))$ with $\sigma_{c1 \wedge \ldots \wedge cn}(R)$
- And you can replace $\sigma_{c1 \wedge \ldots \wedge cn}(R)$ with $\sigma_{c1}(\ldots \sigma_{cn}(R))$
- If you have several conditions connected by "AND" in a select operator, then you can apply them one at a time.

Disk: 10^5 to 10^6 times slower than memory (we'll use 10^6 in this example)

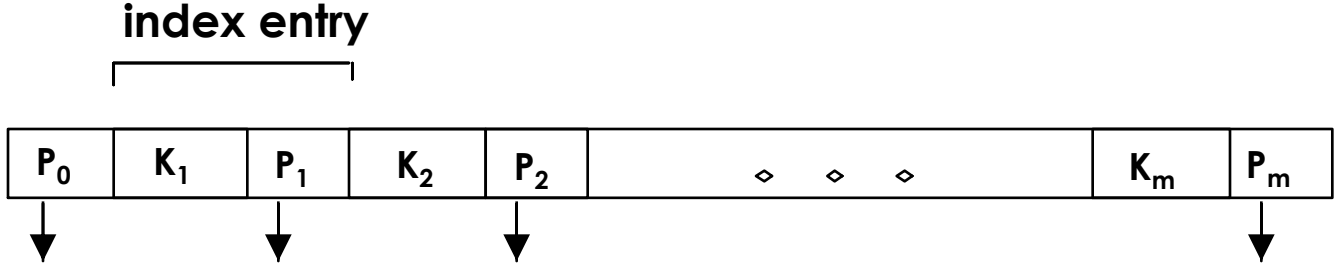
- Disk access time is about 2 to 7 milliseconds
- Memory access time: 50 to 70 nanoseconds
- 7 milliseconds vs. 70 nanoseconds
  therefore disk access is 100,000 times slower than memory access
- Contrast 1 second (pick up a piece of paper)
  vs. 100,000 seconds (drive to SF and back – about 28 hours)
- Thus, cost of a query is the # of input/output (I/O) operations required, e.g., for a query.

B+ Tree Indexes

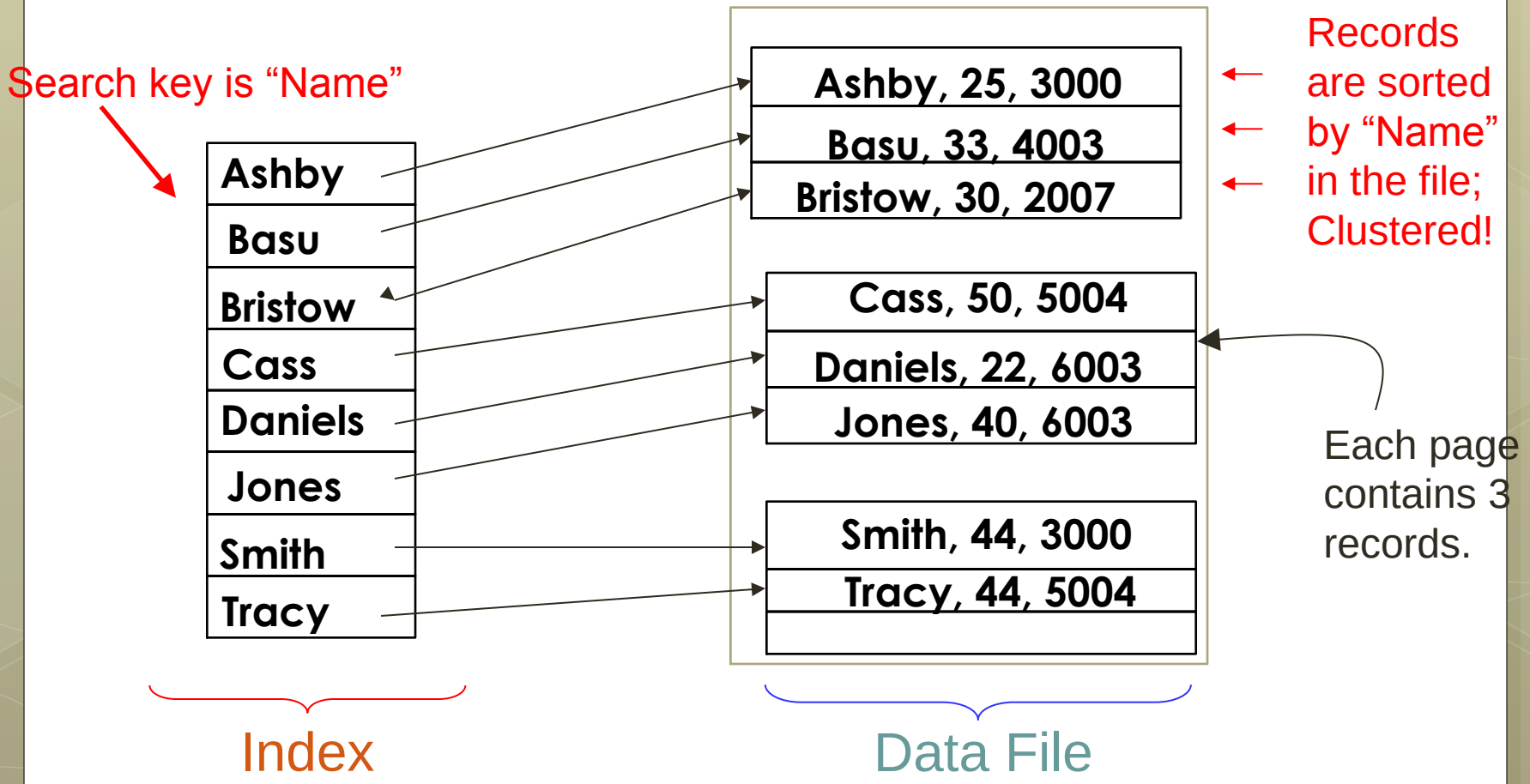


Leaf pages contain *rows*, and are chained (prev & next)

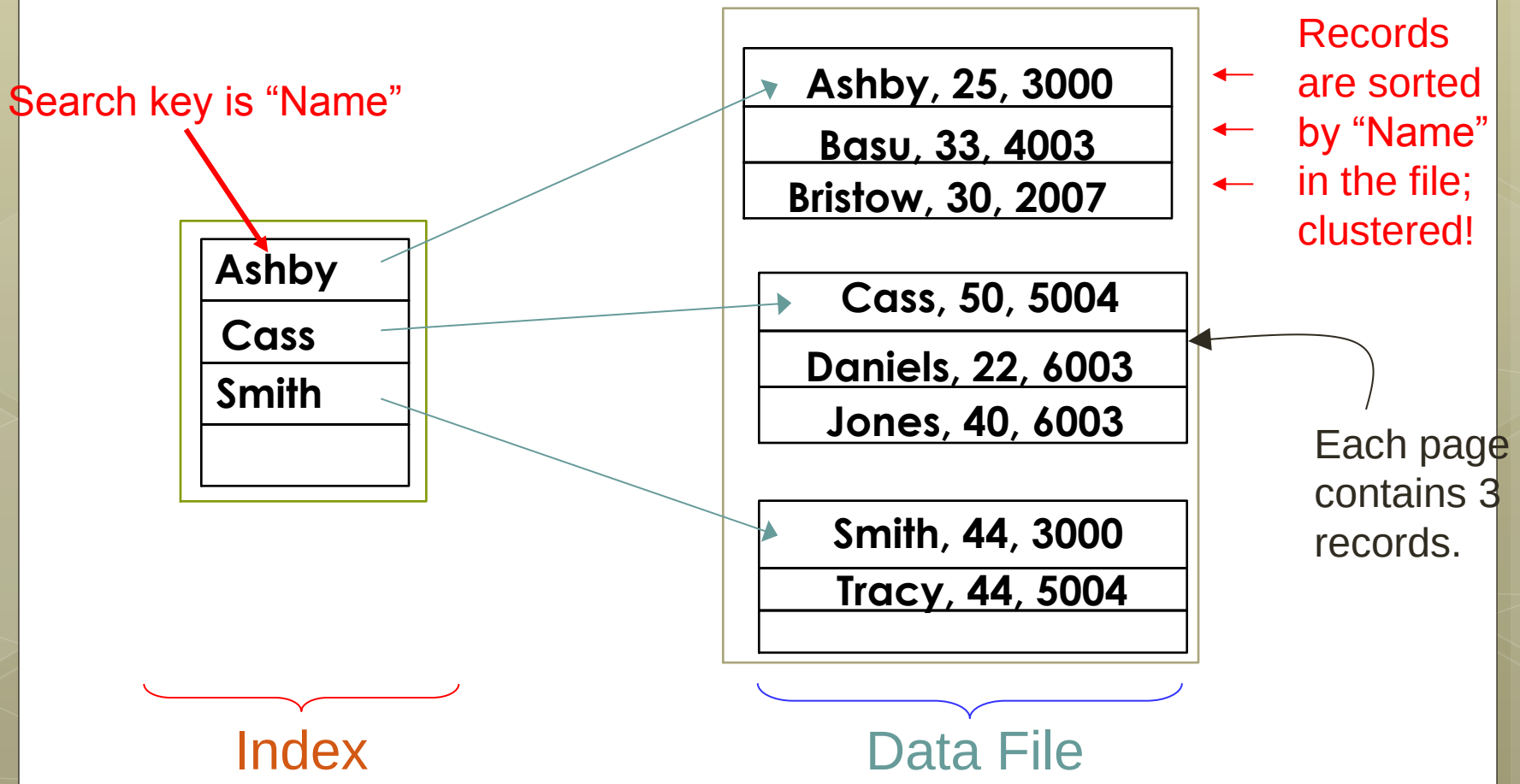
Non-leaf pages have *index entries;* only used to direct searches:



dense index (one index entry for each data record) and clustered index (data are sorted based on search key)



sparse index (one index entry per PAGE of data)
clustered index(data is sorted on search key)

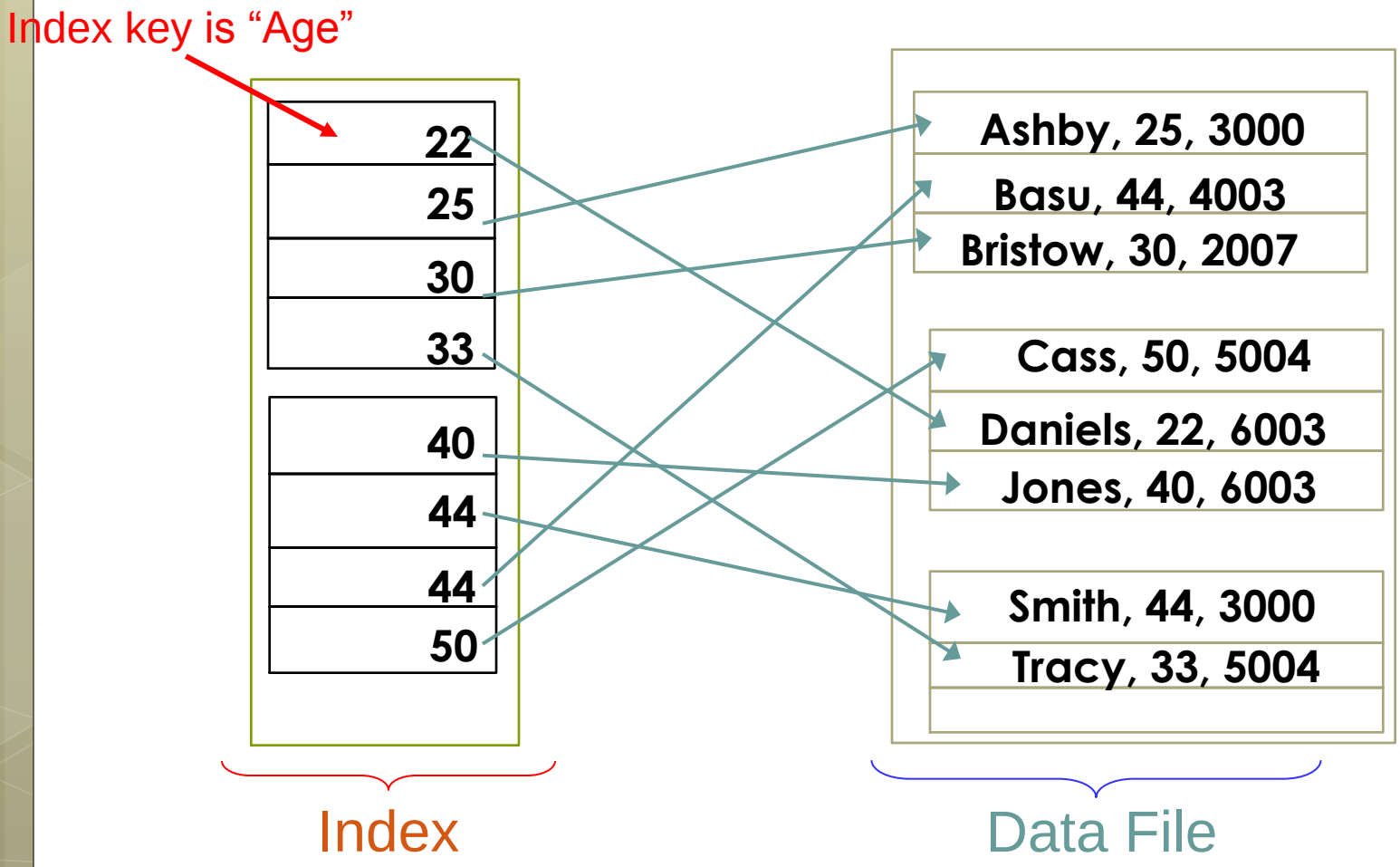


Sparse, clustered index
good for range (e.g., a_id > 400 and a_id < 500)

- The data is sorted according to the index key. There is one entry for each page of data records.
- Consider a phone book – with the heading on each page such as “Mcinroy – Mckee” or “Lowe – Lozano”. This tells us that all names that fall between Lowe and Lozano – will be on this page.
- On disk, one I/O operation gets us a whole page of entries – sorted by name. (~350 names)

unclustered index (records NOT Sorted on Search Key)

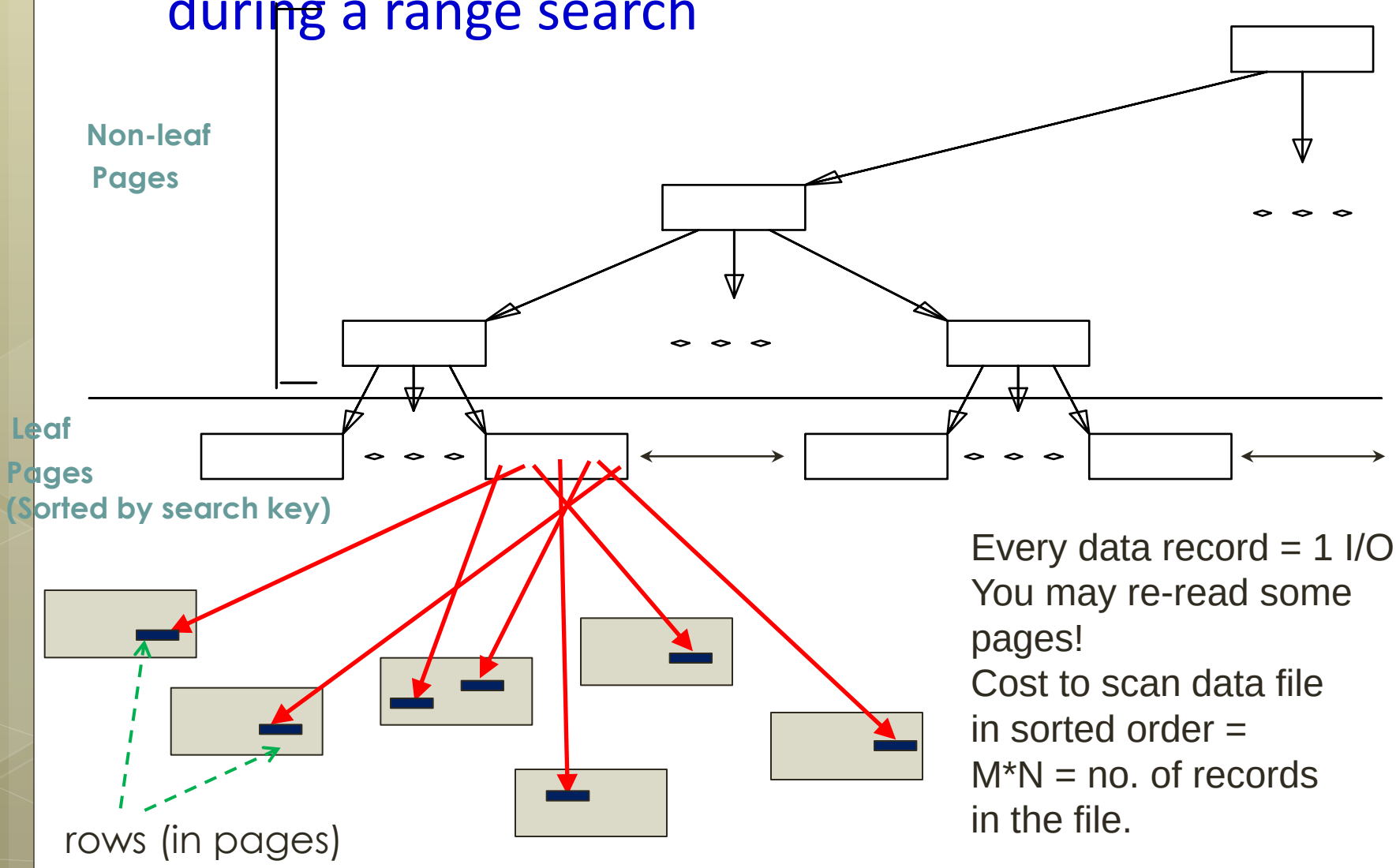
dense index (one entry for every data record)



Unclustered indexes

- In an unclustered index (also called a secondary index) the underlying data is NOT sorted according to the search key.
- Example: imagine building an index on a phone book based on phone number. You MUST put an entry in the index for every single person in the phone book.
- Thus … an unclustered index MUST be dense; there is no other choice.
- For a range search, we need one I/O for every data record!

I/Os using a dense, unclustered index during a range search



Every data record = 1 I/O!
You may re-read some pages!
Cost to scan data file in sorted order = M*N = no. of records in the file.

Quick Exercise

- Insert a large number of rows into one of your tables in your w13db?? database.
- Create an index on one of your attributes in your table.
- Run some queries and see if you can see the benefits of an index.
- Try clustering one of your indices; see if it makes a different.
- Use queries like this:
  select count(*) from agent where …
  to avoid the cost of presenting the query answer onto the web page.
- Try using explain … if you want to see the query plan.

14

Indexes

- DBMSs often create a clustered index on all primary keys. Note: primary keys are the values used in foreign keys.
- Only one clustered index per table! Why?
- You need to decide whether you want additional (unclustered/secondary) indexes.
- (All) indices make updates slower and (typically) make at least some queries faster. It's a classic tradeoff.
- Deciding on indexes is part of physical DB design

We Need to Understand the Workload to Choose Indices

- For each *query* in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be? (That is, how many matching records will there be?)
- For each *update* in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - What type of updates (i.e., INSERT/DELETE/UPDATE) are prominent, and which tables and attributes are affected?

Tips for index selection

- Don't use indexes on small tables (< 200 rows)
- Don't use indexes on columns with few values (T/F, gender, state)
- For most systems, indexes on primary keys and foreign keys are sufficient
- Don't forget to add indexes when the schema changes!

Physical DB design can change the schema!

- Denormalize – but the system must correctly manage the redundancy and prevent update anomalies
- Decompose a table horizontally or vertically

Vertical decomposition: split one table into two – choose which attributes go into each table

Take this table:

Course				
c#	cname	instructor	room	days

And replace it with these three tables:

Course1		
c#	room	days

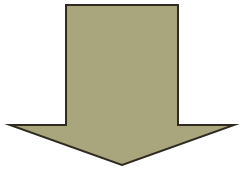
Course2	
c#	cname

Course3	
c#	instructor

Advantage: if queries use only a subset of the attributes, smaller rows (fewer attributes) means faster access.
Disadvantage: if queries need all attributes, then you have to join.

Horizontal partition/decomposition (split data into two identical tables – with two different names)

Course				
c#	cname	instructor	room	days



Undergraduate-Course				
c#	cname	instructor	room	days

Graduate-Course				
c#	cname	instructor	room	days

Define views to hide changes in physical DB

Undergraduate-Course (c#, cname, instructor, room, days)
Graduate-Course (c#, cname, instructor, room, days)

```
CREATE VIEW  Courses(cid, sid, jid, did, pid, qty, val)
        AS      SELECT  *
                FROM  Undergraduate-courses
        UNION
                SELECT  *
                FROM  Graduate-courses
```

- A query can use any of the three.
- Updates will need to know which table to use.