By Marina Barsky

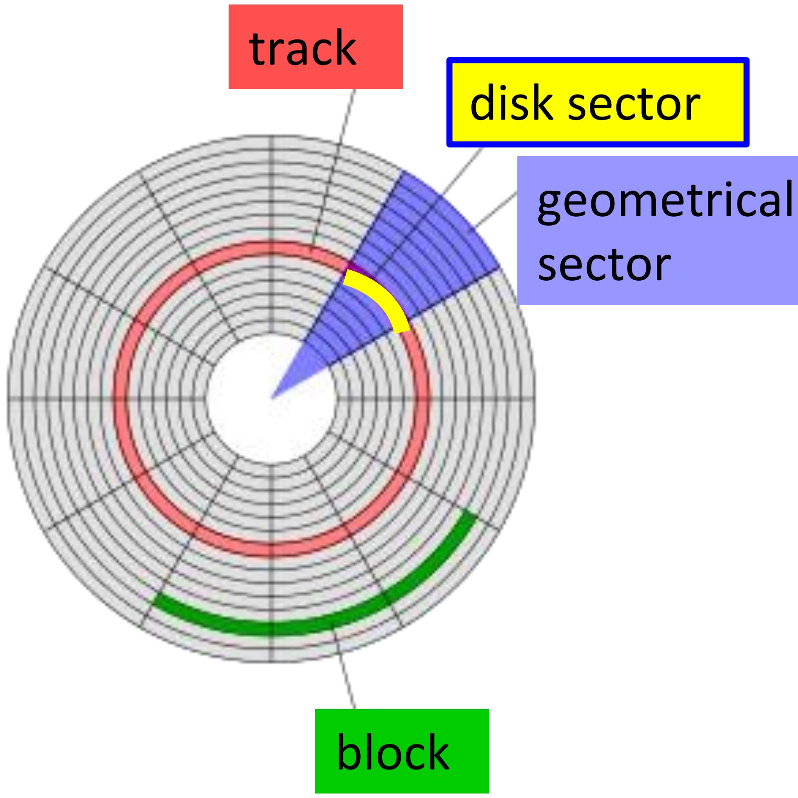
Advanced features of relational databases

Lecture 13

INDEXES

Disk blocks

- The ***Block*** (transferred as a ***Page*** to RAM) is a fixed-size portion of secondary storage corresponding to the amount of data transferred in a single access and physically occupies one or more consecutive sectors
    - Typical block size: 1, 4, 8, 16, or 32 KB.
    - Has to be set before creating database
- The data is read and written in blocks



Auxiliary data structures for efficient search: indexes

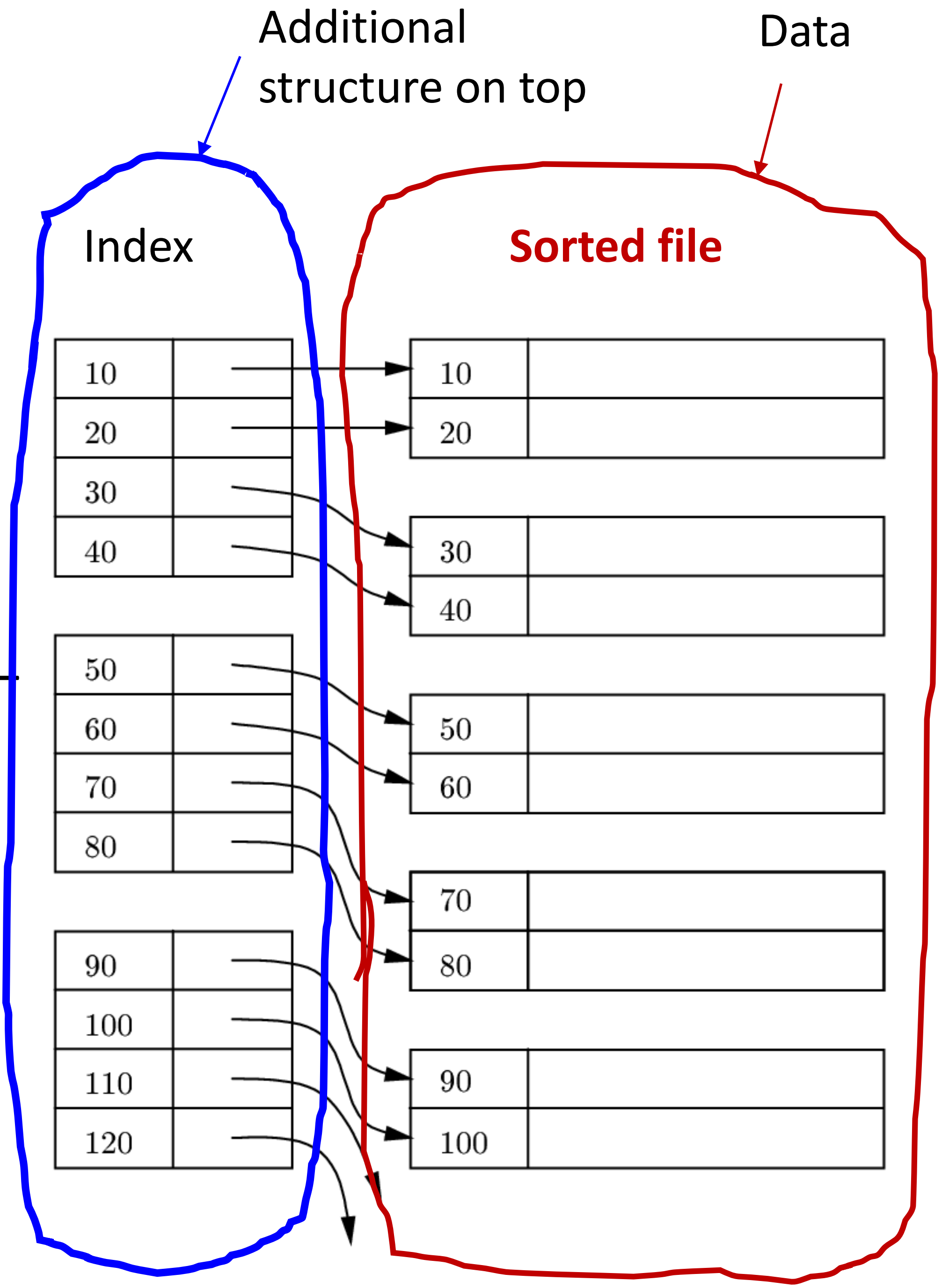
- Goal: quickly locate the record given a key
- Idea 1:
  - The records are mapped to the disk blocks in specific ways: we deduce the disk location from a key, because records either sorted by key or the block is a hash of a key
- Idea 2:
  - Store records in a pile
  - Provide auxiliary data structures guiding the search (think library index/catalogue)

Flat indexes

- Have a catalog of search keys which is smaller than the entire table and can be searched more efficiently (in RAM or with less disk I/Os)
- Inside the index each value of a search key is associated with a unique, system-generated physical address of a corresponding tuple on disk: RID (file number, block number, slot within the data block)

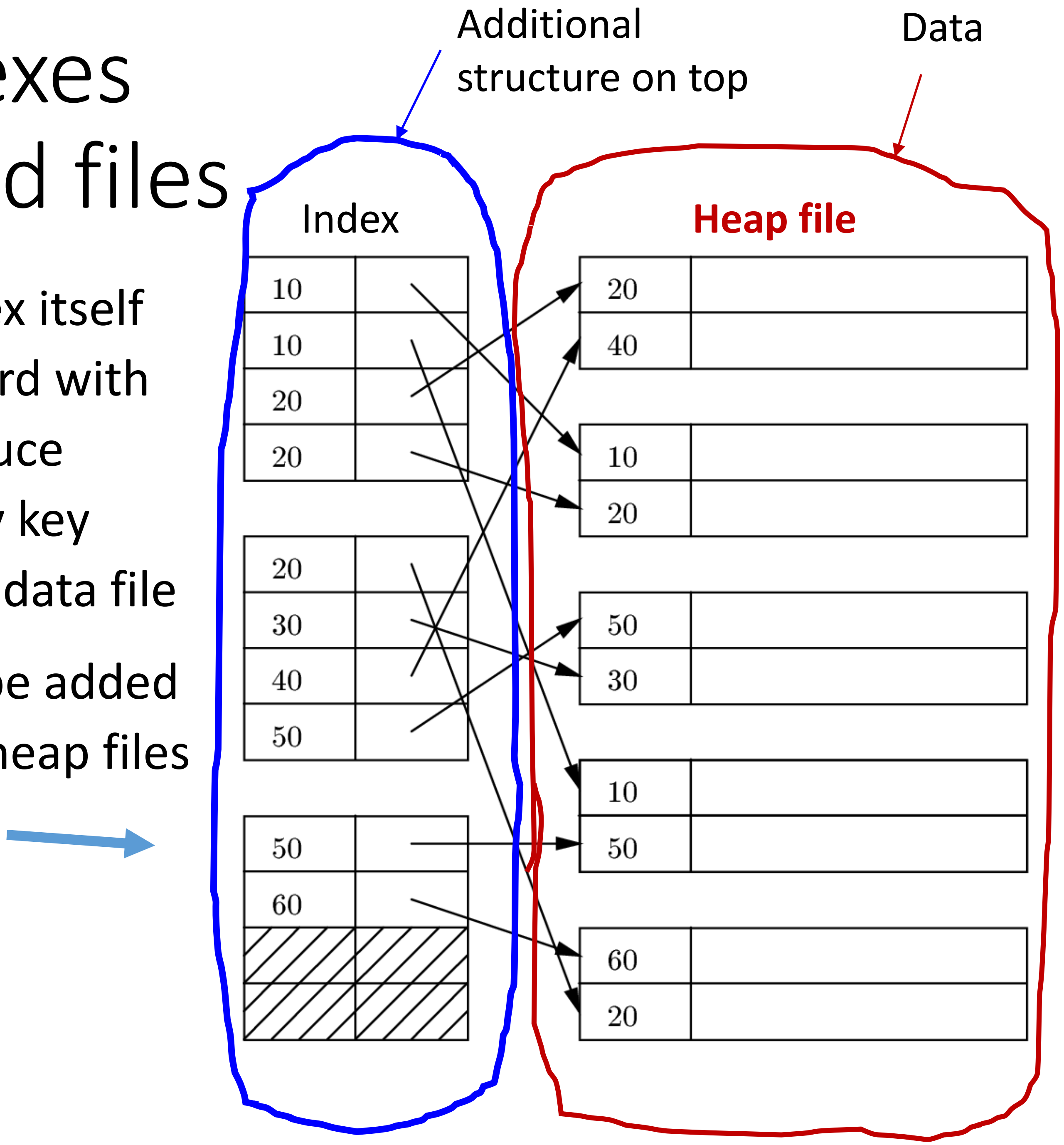
Dense indexes

- ***Dense index*** – each record has its representative inside an index
- If the table has multiple fields, the index – which stores only key-RID pair - is much smaller – may fit into RAM
- The keys in the index are sorted: use binary search, buffer guiding pointers at 1/2N, 1/4N, 3/4N, 1/8N, 3/8N, 5/8N, 7/8N –th positions to save disk I/Os



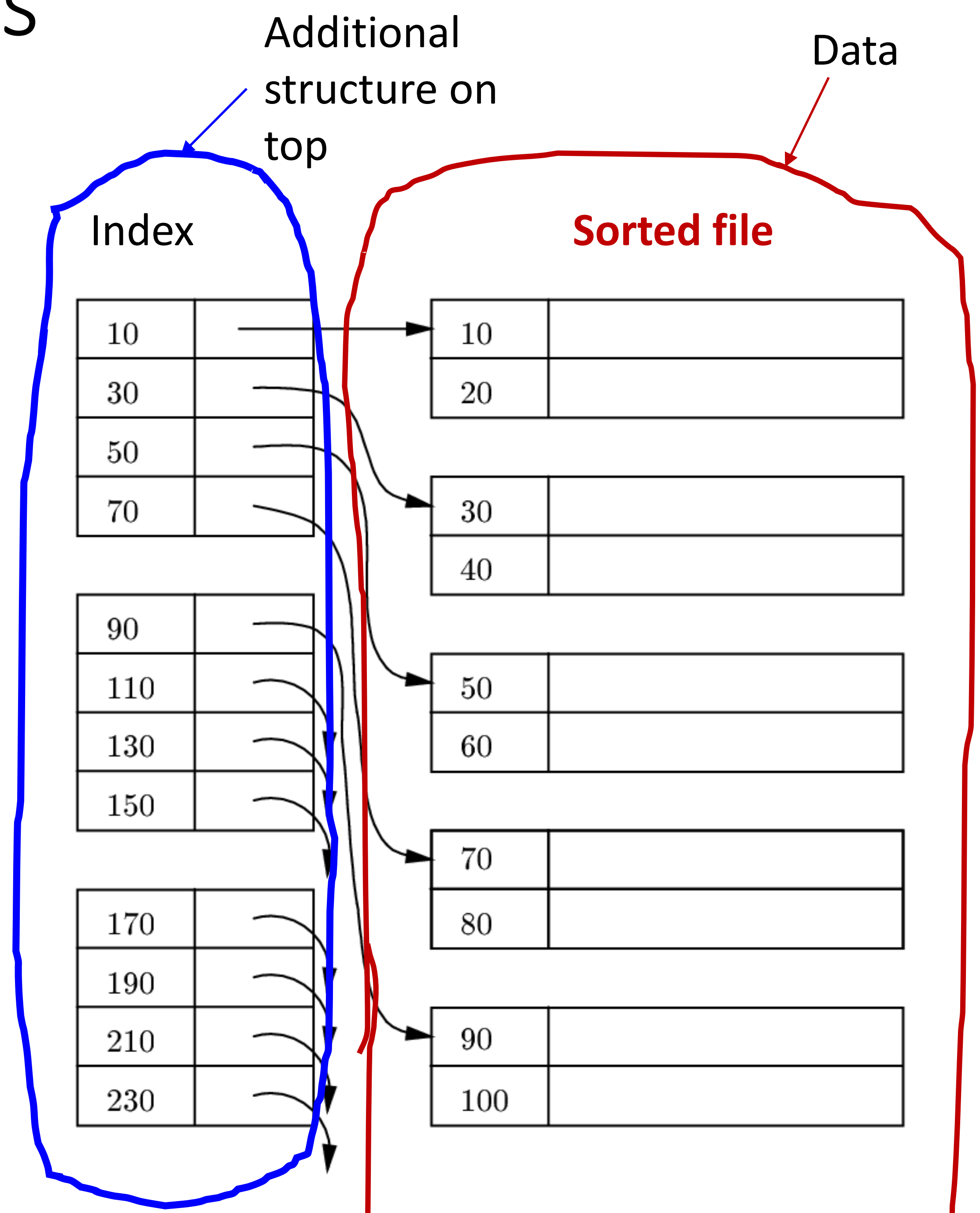
Dense indexes on unsorted files

- Search through index itself can answer if a record with key *A* exists or produce counts of records by key without accessing a data file
- Dense indexes can be added even to unordered heap files



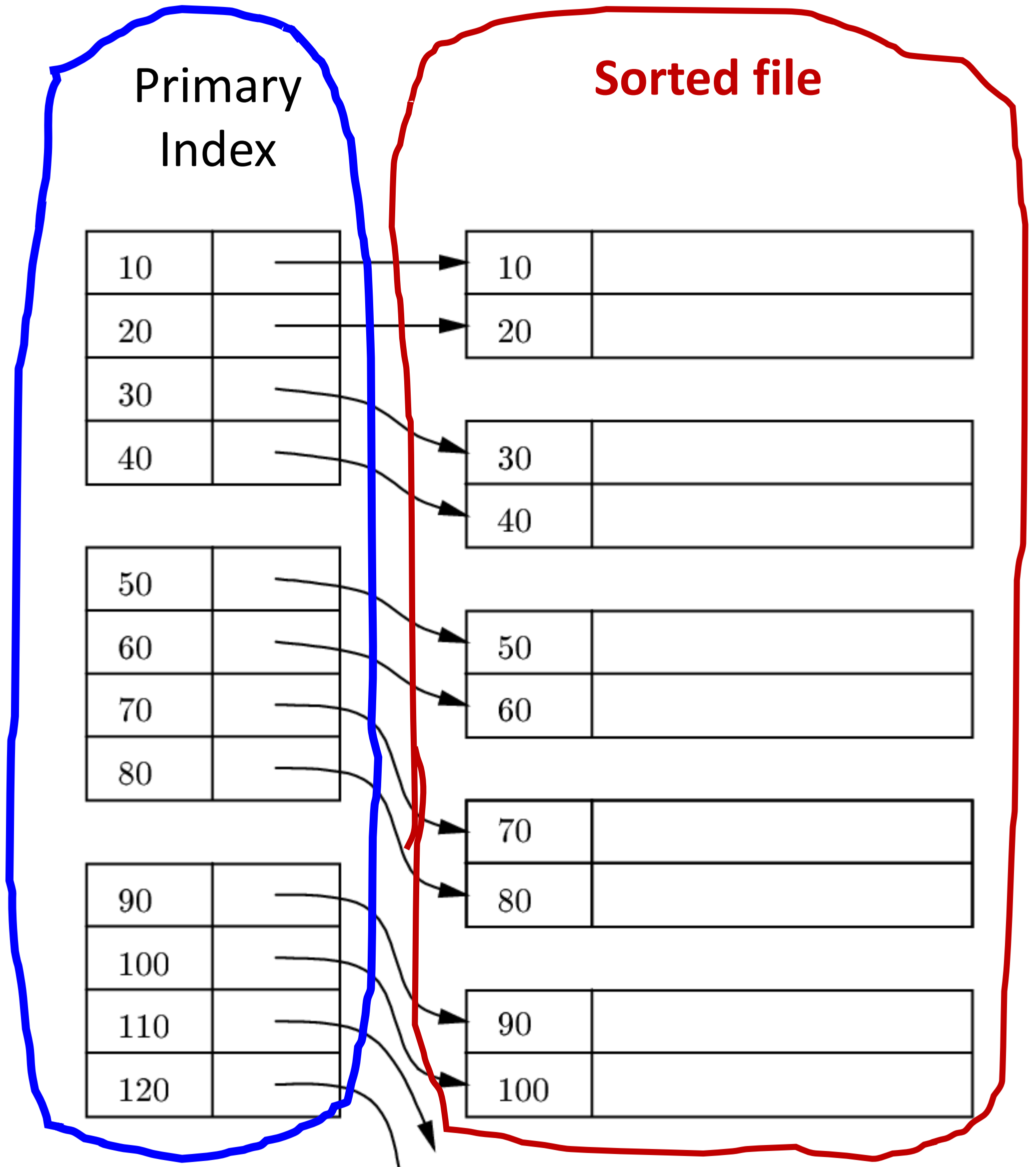
Sparse indexes

- ***Sparse index*** – contains key-RID pairs for only a subset of records, typically first in each block.
- Works only with sorted files – why?
- Allows for very small indexes - better chance of fitting in memory
- Tradeoff: *must* access the relation file even if the record is not present



Primary indexes

- ***Primary index*** – indexes on a sorted file for the sorting attribute
- Only one primary index per relation – otherwise needs to maintain several sorted copies of the same data



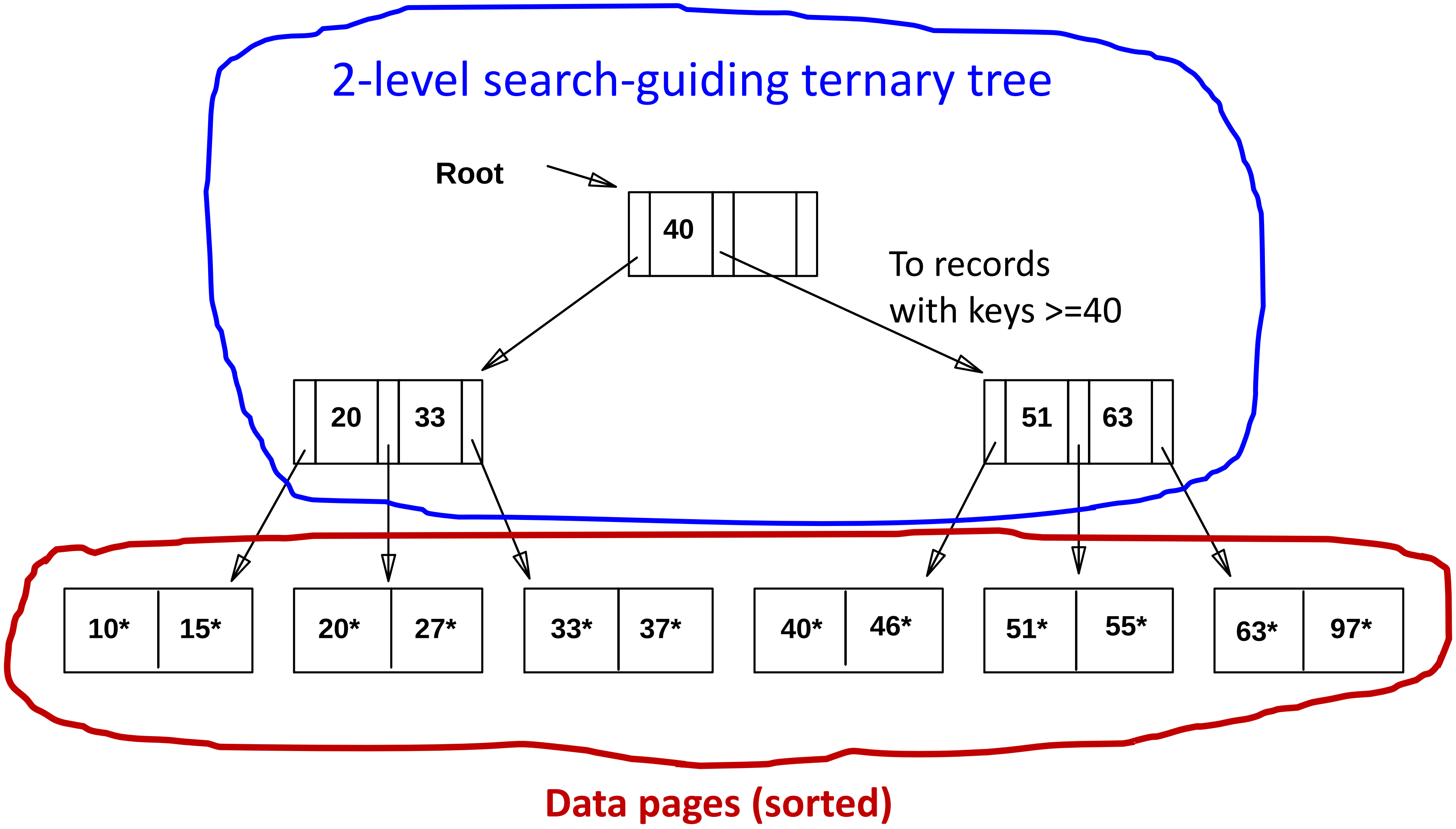
What if a flat index is too big?

Example:

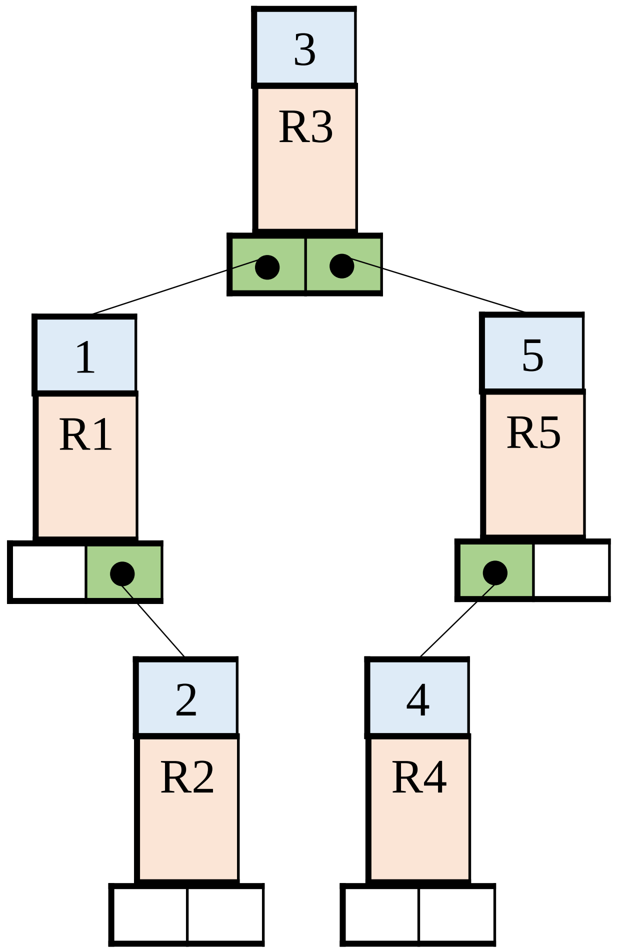
- Relation of size: N = 500 GB = $5*10^{11}$ bytes
- 100 tuples per block: $5*10^9$ blocks to index
- Each key-blockID pair is at least 16 bytes
- So, even keeping one entry per page (sparse index) takes too much space - 8 GB

Solution: build an index on the index itself!

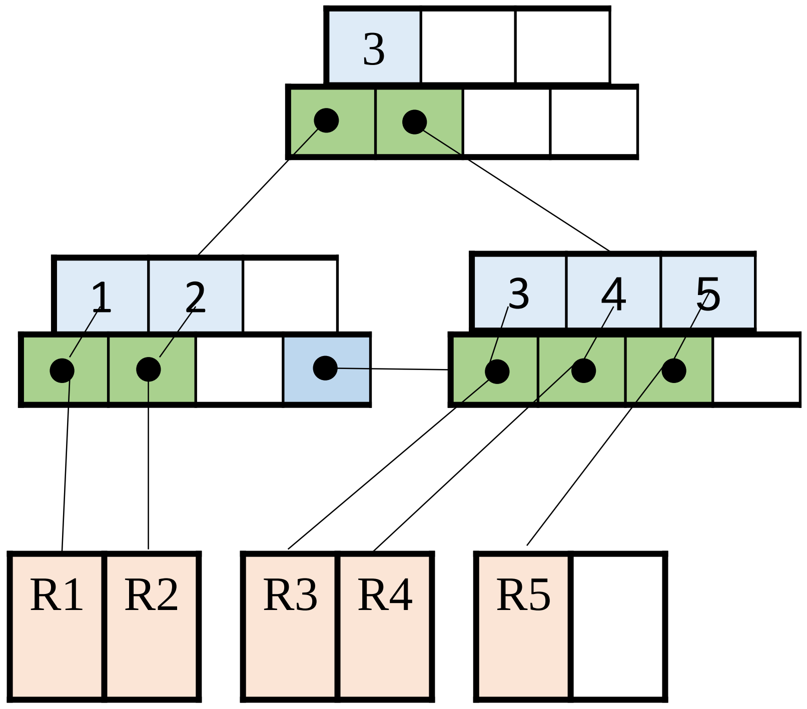
Multi-level indexes - static trees



Dynamic indexes: B+ trees



Binary search tree



Data stored separately

B+ tree

Motivation for Indexes

Consider:

```
SELECT title
FROM Movies
WHERE studioName = 'Disney' AND year =1995;
```

- There might be 10,000 Movies tuples, of which only 200 were made in 1995.
  - Naive way to implement this query is to get all 10,000 tuples and test the condition of the WHERE clause on each.
  - Much more efficient if we had some way of getting only the 200 tuples from the year 1995 and testing each of them to see if the studio was Disney.
  - Even more efficient if we could obtain directly only the 10 or so tuples that satisfied both the conditions of the WHERE clause.

Pointer Intersection example

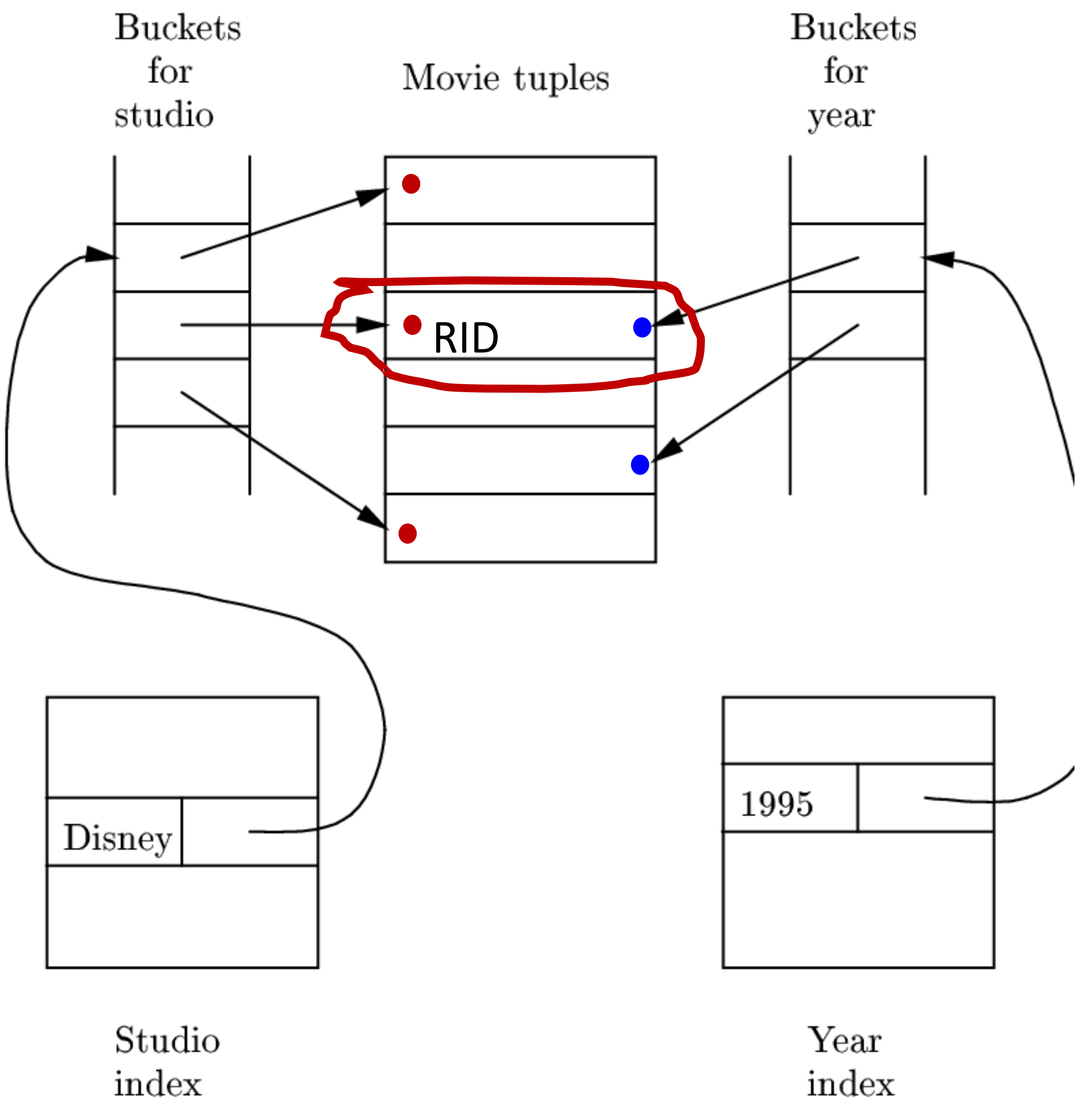
```
Movies(
  title,
  year,
  length,
  studioName);
```

Schema

Assume secondary indexes on **studioName** and **year**.

Query

```
SELECT title
FROM Movies
WHERE
  studioName='Disney'
  AND year = 1995;
```



Intersection of results from 2 indexes gives RID

Create Index

```
CREATE INDEX index_name ON table_name (column_name);
```

Created implicitly if you declare the column to be a PRIMARY KEY or UNIQUE.

Once an index exists, the user does not have to open or use it with a command, it is used automatically (by query optimizer)

All indexes are stored separately from the table on which they are based

Examples

1. CREATE INDEX YearIndex ON Movies(year);
2. CREATE INDEX KeyIndex ON Movies(title, year);
3. CREATE INDEX KeyIndex ON Movies (year, title);

When would it be beneficial to create the third vs. second?

Dropping an index:

DROP INDEX YearIndex;

Index on primary key

- Often, the most useful index we can put on a relation is an index on its key.
- Two reasons:
  - Queries in which a value for the key is specified are common.
  - Since there is at most one tuple with a given key value, the index returns either nothing or one location for a tuple.
    - Thus, at most one page of the relation must be retrieved to get that tuple into main memory

Example

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'Star Wars'
   AND year='1994'
   AND producerC =cert;
```

Index on primary key: comparison

Without Key Indexes

- Read each of the blocks of **Movies** and each of the blocks of **MovieExec** at least once.
    - In fact, since these blocks may be too numerous to fit in main memory at the same time, we may have to read each block from disk many times (to form a Cartesian product).

With Key Indexes

- Only two block reads.
    - Index on the key **(title, year)** for **Movies** helps us find the one Movie tuple for 'Star Wars' quickly.
        - Only one block - containing that tuple - is read from disk.
    - Then, after finding the producer-certificate number in that tuple, an index on the key **cert** for **MovieExec** helps us quickly find the one tuple for the producer in the MovieExec relation.
        - Only one block is read again.

Non-Beneficial Indexes

- When the index is not on a key, it may or may not be beneficial.

Example (of not being beneficial)

Suppose the only index we have on Movies is one on **year**, and we want to answer the query:

```
SELECT *
FROM Movie
WHERE year = 1990;
```

- Suppose the tuples of Movie are stored alphabetically by title.
- Then this query gains little from the index on year. If there are, say, 100 movies per page, there is a good chance that any given page has at least one movie made in 1990.

Beneficial Indexes

- There are two situations in which an index can be effective, even if it is not on a key.
    1. If the attribute is almost a key; that is, relatively few tuples have a given value for that attribute.

       Even if each of the tuples with a given value is on a different page, we shall not have to retrieve too many pages from disk.

Example

- Suppose Movies had an index on **title** rather than (title, year).

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'King Kong' AND producerC =cert;
```

Beneficial Indexes (cont.)

2. If the tuples are "clustered" on the indexed attribute. We cluster a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible.
   - Then, even if there are many tuples, we shall not have to retrieve nearly as many pages as there are tuples.

Example

- Suppose Movies had an index on **year** and tuples are clustered on year.

```
SELECT *
FROM Movie
WHERE year = 1990;
```

When to create indexes

Trade-off

- The existence of an index on an attribute may speed up greatly the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well.

- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

Cost Model

1. Tuples of a relation are stored in many pages (blocks) of a disk.
2. One block, which is typically several thousand bytes at least (e.g. 16K), will hold many tuples.
3. To examine even one tuple requires that the whole block be brought into main memory.

The *cost* of a query is dominated by **the number of block accesses**. Main memory accesses can be neglected.

Introduction to selection of indexes

StarsIn(movieTitle, movie Year , starName)

Q1:

```
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = s;
```

Q2:

```
SELECT starName
FROM StarsIn
WHERE movieTitle = t AND movieYear= y;
```

I:

```
INSERT INTO StarsIn VALUES(t, y, s);
```

Before we begin: answer

1. How many pages for StarsIn: 10 pages

   If we need to examine the entire relation the cost is 10.

2. Average number of movies per star: on the average, a star has appeared in 3 movies

3. Average number of stars per movie: a movie has 3 stars on average

Analysis

1. Since the tuples for a given star or a given movie are likely to be spread over the 10 pages of **StarsIn**, even if we have an index on starName or on the combination of movie title and movieYear, it will take 3 disk accesses to retrieve the 3 tuples for a star or movie. If we have no index on the star or movie, respectively, then 10 disk accesses are required.

2. One disk access is needed to read a page of the index every time we use that index to locate tuples with a given value for the indexed attribute(s). If an index page must be modified (in the case of an insertion), then another disk access is needed to write back the modified page.

3. Likewise, in the case of an insertion, one disk access is needed to read a page on which the new tuple will be placed, and another disk access is needed to write back this page. We assume that, even without an index, we can find some page on which an additional tuple will fit, without scanning the entire relation.

Average cost for NO INDEX

Action	NO INDEX
Q1	10
Q2	10
Ins	2
Average	$10p_1 + 10p_2 + 2(1 - p_1 – p_2) = 2 + 8p_1 +8p_2$

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

Average cost for Star INDEX

Action	NO INDEX	INDEX on Star attribute of StarsIn
Q1	10	4 (1 to read an index, 3 to load corresponding blocks
Q2	10	10
Ins	2	4 (read/write 1 page for the index and for the data)
Average	$2 + 8p_1 + 8p_2$	$4p_1 + 10p_2 + 4(1 - p_1 - p_2) = 4 + 6p_2$

Q1: SELECT movieTitle, movieYear FROM StarsIn WHERE starName = s;
Q2: SELECT starName FROM StarsIn WHERE movieTitle = t AND movieYear= y;

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

Average cost for Movie INDEX

Action	NO INDEX	Star INDEX	INDEX on Movies attributes only
Q1	10	4	10
Q2	10	10	4
Ins	2	4	4
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$

Q1: SELECT movieTitle, movieYear FROM StarsIn WHERE starName = s;
Q2: SELECT starName FROM StarsIn WHERE movieTitle = t AND movieYear= y;

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

Average cost for both indexes

Action	NO INDEX	Star INDEX	Movies INDEX	INDEX on Movies attributes AND Star
Q1	10	4	10	4 (1 page to read an index, 3 to read data)
Q2	10	10	4	4 (1 page to read an index, 3 to read data)
Ins	2	4	4	6 (read/write 1 page for each index and 1 for data)
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$4p_1 + 4p_2 + 6(1 – p_1 - p_2) = 6 - 2p_1 - 2p_2$

Q1: SELECT movieTitle, movieYear FROM StarsIn WHERE starName = s;
Q2: SELECT starName FROM StarsIn WHERE movieTitle = t AND movieYear= y;

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

What solution is the best?

Action	NO INDEX	Star INDEX	Movies INDEX	INDEX on both
Q1	10	4	10	4
Q2	10	10	4	4
Ins	2	4	4	6
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

If p1 = p2 = 0.1:
If p1 = p2 = 0.4:
If p1 = 0.5, p2 = 0.1:
If p1 = 0.1, p2 = 0.5:

p_1	is the fraction of times Q1 is executed
p_2	is the fraction of times Q2 is executed
$1-p_1-p_2$	is the fraction of times I is executed

Reasoning

- If $p_1 = p_2 = 0.1$, then the expression $2+ 8p_1 + 8p_2$ is the smallest, so we would prefer not to create any indexes.

- If $p_1 = p_2 = 0.4$, then the formula $6 - 2p_1 - 2p_2$ turns out to be the smallest, so we would prefer indexes on both starName and on the (movieTitle, movieYear) combination.

- If $p_1 = 0.5$ and $p_2 = 0.1$, then an index on stars only gives the best average value, because $4 + 6p_2$ is the formula with the smallest value.

- If $p_1 = 0.1$ and $p_2 = 0.5$, then create an index on only movies.