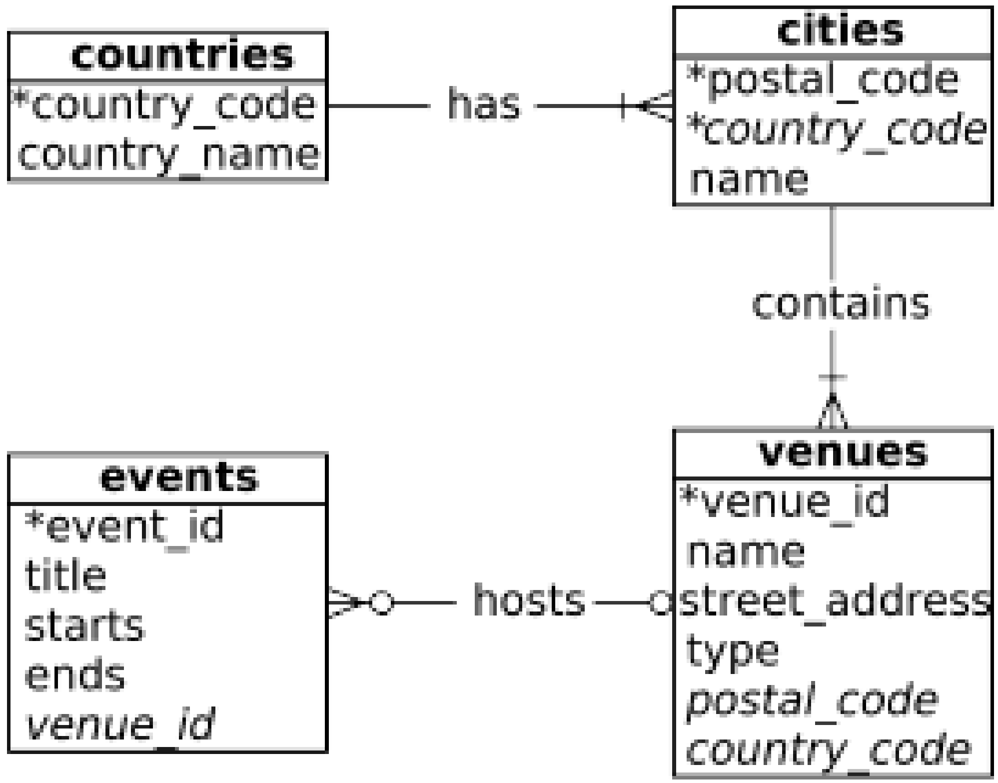
Our working example



Let’s add more data to countries and cities.

Indexes: B-TREES and Hashing

Files, pages, and records

The raw disk space is organized into **files**.
Files are made up of **pages**, and pages contain **records**

Data is allocated and deallocated in increments of pages.

- File: A collection of pages
  Page: a collection of records.
- File operations:
  - insert/delete/modify record
  - read a particular record (specified using the *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, the system must:
  - keep track of the *pages* in a file: page id (pid)
  - keep track of *free space* on pages
  - keep track of the *records* on a page: record id (rid)
  - Many alternatives for keeping track of this information
- Operations: create/destroy file, insert/delete record, fetch a record with a specified rid, scan all records

Indexes

- A Heap file allows us to retrieve records:
  - by specifying the *rid,* or
  - by scanning all records sequentially
- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
  - Find all students in the "ECE" department
  - Find all students with a gpa > 3
- Indexes are file structures that enable us to answer such *value-based queries* efficiently.

Motivation

Consider the following table:

```
CREATE TABLE Tweets (
   uniqueMsgID INTEGER,       -- unique message id
   tstamp      TIMESTAMP,     -- when was the tweet posted
   uid         INTEGER,       -- unique id of the user
   msg         VARCHAR (140), -- the actual message
   zip         INTEGER        -- zipcode when posted
   );
```

Consider the following query, Q1:

```
SELECT * FROM Tweets
WHERE uid = 145;
```

And, the following query, Q2:

```
SELECT * FROM Tweets
WHERE zip BETWEEN 53000 AND 54999
```

Ways to evaluate the queries, efficiently?

1. Store the table as a heapfile, scan the file. I/O Cost?
2. Store the table as a **sorted file**, binary search the file. I/O Cost?
3. Store the table as a heapfile, build an **index**, and search using the index.
4. Store the table in an **index** file. The entire tuple is stored in the index!

Index

- Two main types of indices
  - **Hash** index: good for equality search (e.g. Q1)
  - **B-tree** index: good for both range search (e.g. Q2) and equality search (e.g. Q1)
    - Generally a hash index is faster than a B-tree index for equality search
- Hash indices aim to get O(1) I/O and CPU performance for search and insert
- B-Trees have $O(\log_F N)$ I/O and CPU cost for search, insert and delete.

What is in the index

- Two things: **index key** and **some value**
  - Insert(indexKey, value)
  - Search (indexKey) -> value (s)
- What is the index key for Q1 and Q2?
- Consider Q3:

```
SELECT * FROM Tweets
WHERE uid = 145 AND
 zip BETWEEN 53000 AND 54999
```

- Value:
  - Record id
  - List of record id
  - The entire tuple!

Hash indexes

- *Hash-based indexes are best for equality selections*
  - *Cannot* support range searches, except by generating all values
  - Static and dynamic hashing techniques exist

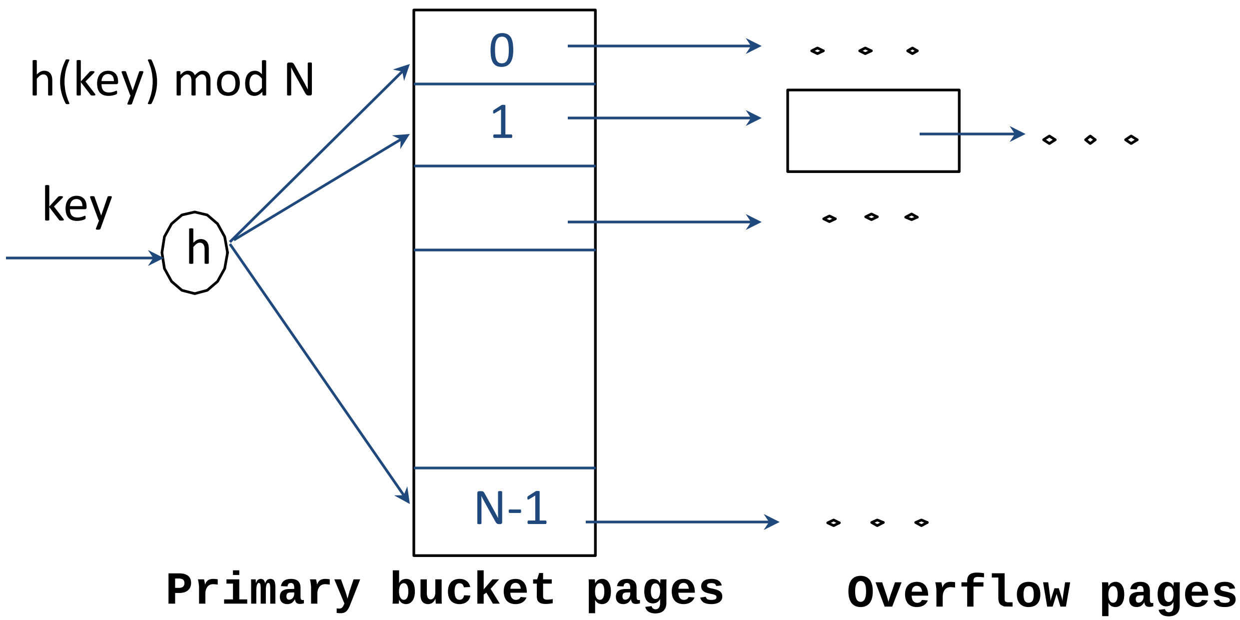
- *Hash indexes not as widespread as B+-Trees*
  - Some DBMS do not provide hash indexes (Postgres does)
  - But hashing still useful in query optimizers (DB Internals)
  - E.g., in case of equality joins

Static Hashing

Number of primary pages N *fixed, allocated sequentially*

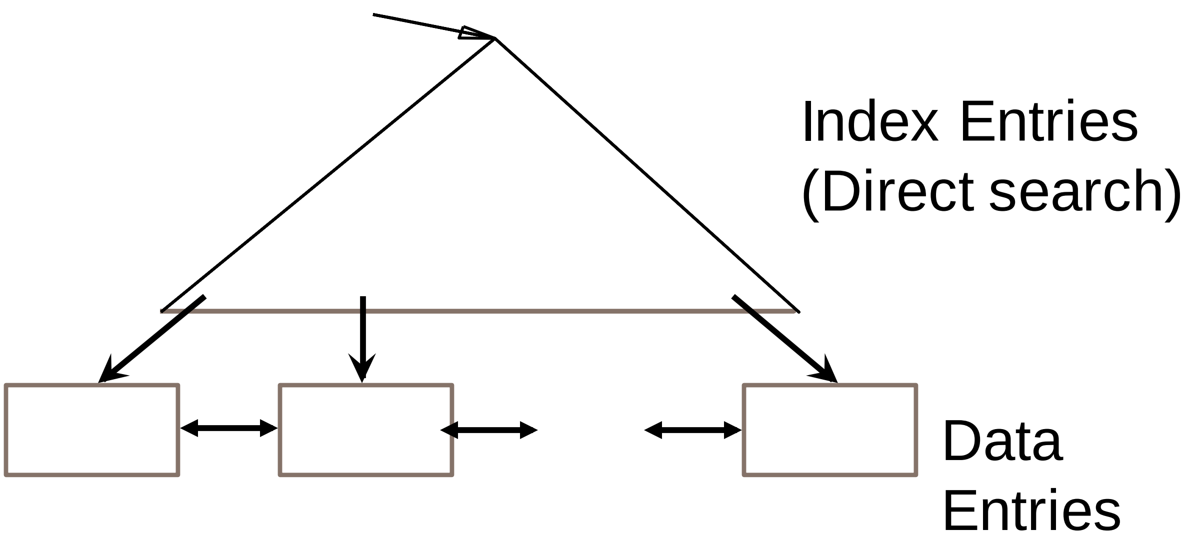
- overflow pages may be needed when file grows
- Buckets contain data entries

***h**(k) mod N = bucket for data entry with key k*



(Ubiquitous) B+ Tree

- Height-balanced (dynamic) tree structure
- Insert/delete at $\log_F N$ cost (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= $\underline{m}$ <= 2**d** entries. The parameter **d** is called the **order** of the tree.
- Supports equality and range-searches efficiently.

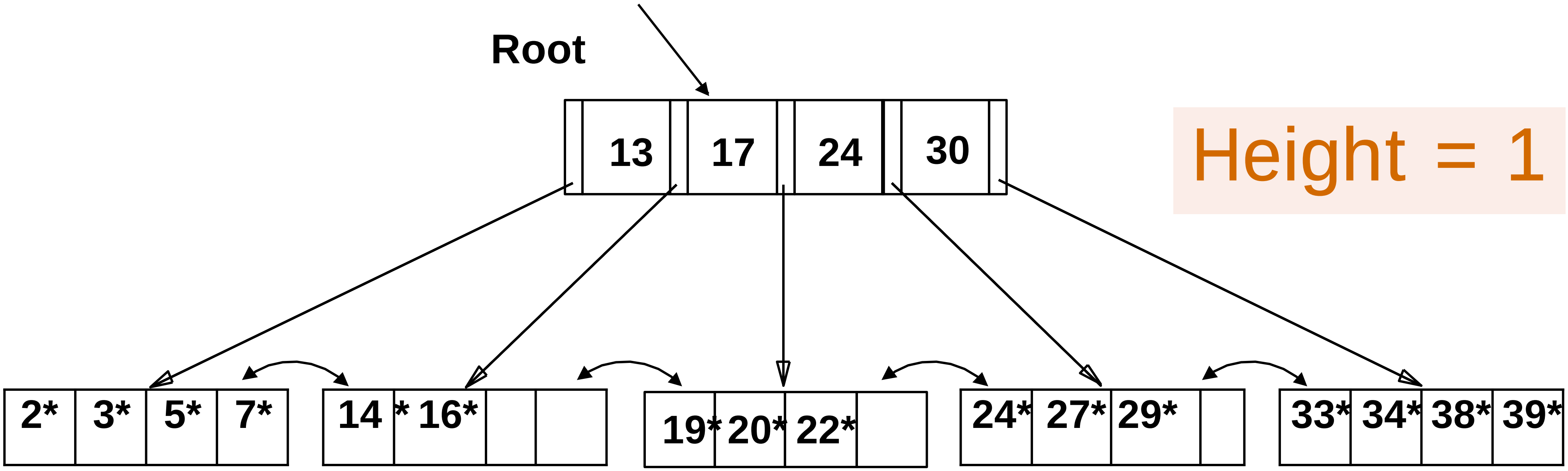


Index Entries
Entries in the index
(i.e. non-leaf) pages:
(search key value, pageid)

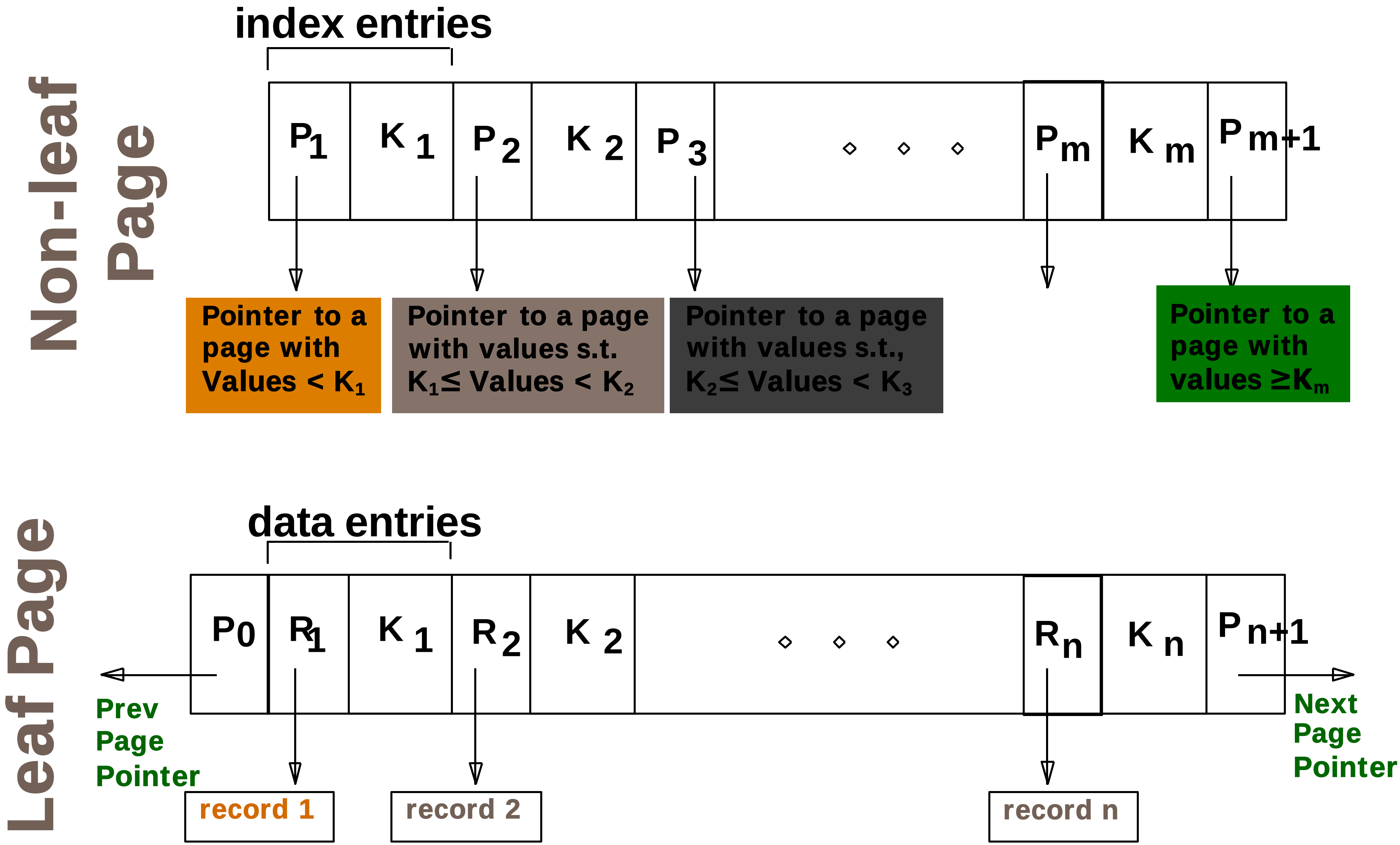
Data Entries
Entries in the leaf pages:
(search key value, recordid)

Example B+ Tree

- Search: Starting from root, examine index entries in non-leaf nodes, and traverse down the tree until a leaf node is reached
  - Non-leaf nodes can be searched using a binary or a linear search.
- Search for 5*, 15*, all data entries >=24*



B+-tree Page Format



B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

System Catalogs/Data dictionary

- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

  ☛ *Catalogs are themselves stored as relations*!

Alternatives for Data Entry k* in Index

- Three alternatives:
  1. Data record with key value **k**
  2. <**k**, rid of data record with search key value **k**>
  3. <**k**, list of rids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)

- Alternative 1:
  - If this is used, index structure is a file organization for data records (like Heap files or sorted files).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records duplicated, leading to redundant storage and potential inconsistency.)
  - If data records very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives for Data Entries (Contd.)

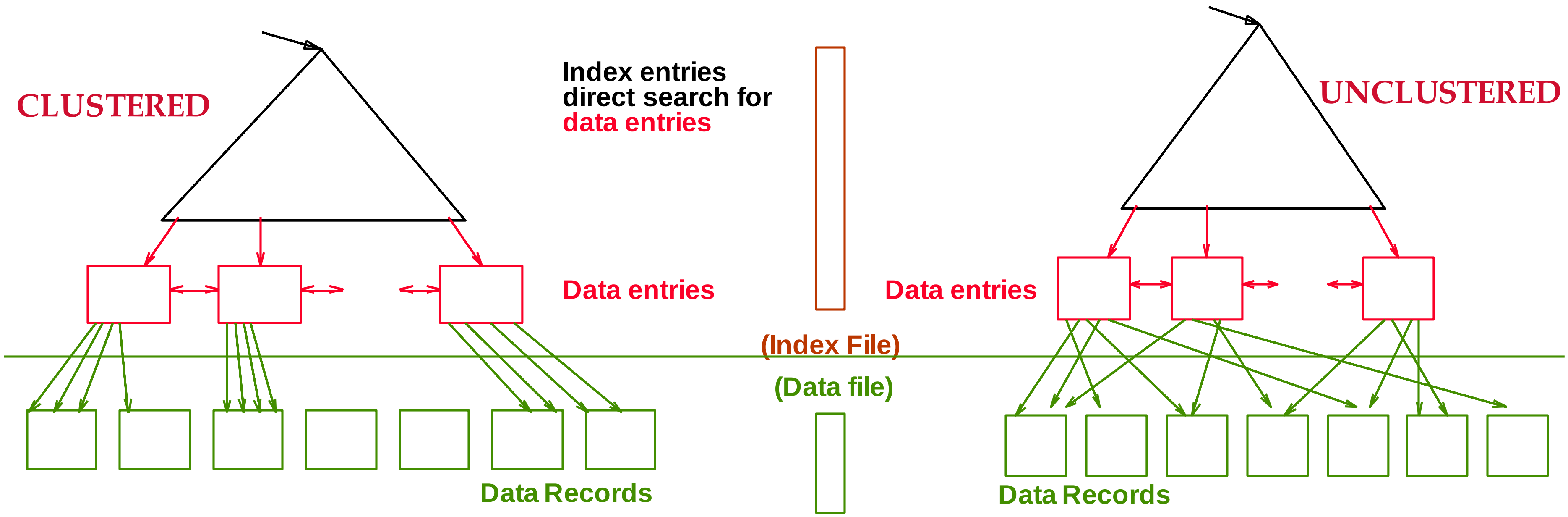
- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search is much smaller than with Alternative 1.)
  - If more than one index is required on a given file, at most one index can use Alternative 1; rest must use Alternatives 2 or 3.
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Index Classification

- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', order of data entries, then called clustered index.
  - Alternative 1 implies clustered, but not vice-versa.
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

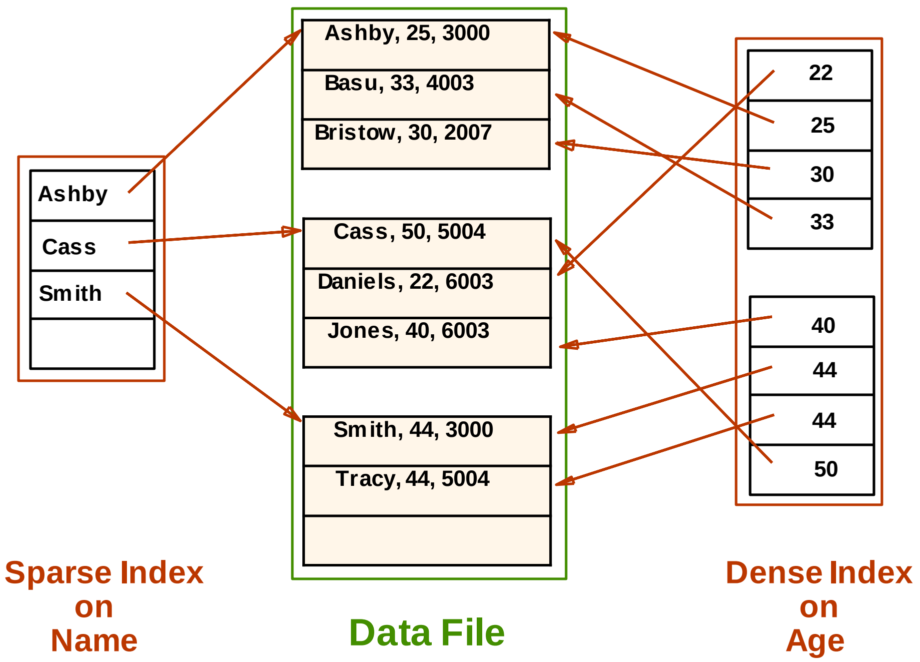
Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



Index Classification (Contd.)

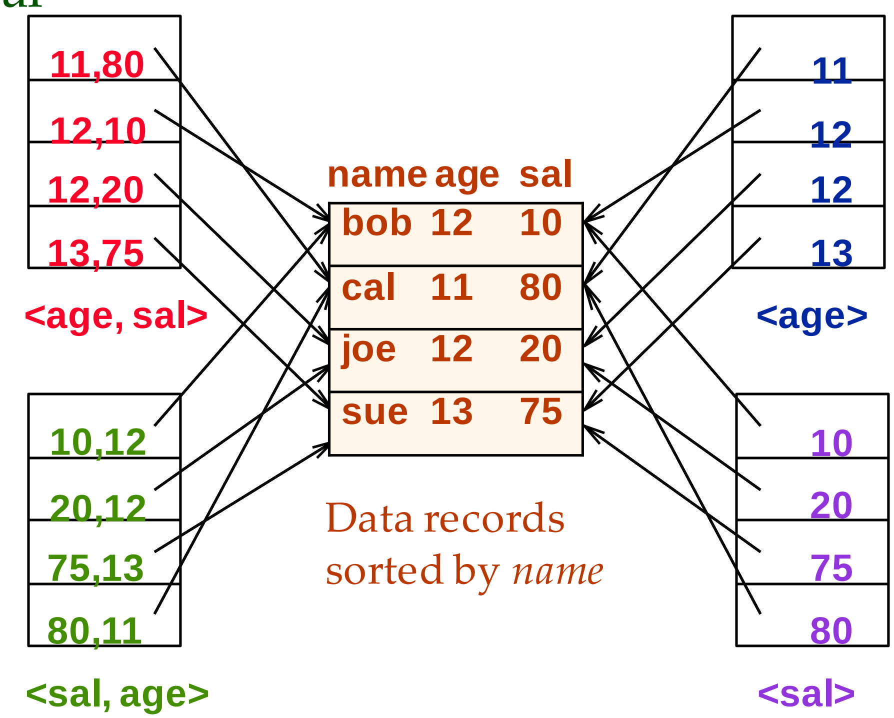
- *Dense vs. Sparse:* If there is at least one data entry per search key value (in some data record), then dense.
  - Alternative 1 always leads to dense index.
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.



Index Classification (Contd.)

- *Composite Search Keys*: *Search on a combination of fields.*
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10
- *Data entries in index sorted by search key to support range queries.*
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



Data entries in index sorted by *<sal,age>*

Data entries sorted by *<sal>*

For today…

1. Write a query that finds the country name of the Fight Club event. Store it as a view.

2. Alter the venues table such that it contains a Boolean column called active with a default value of TRUE.

3. Add a b-tree on table cities over country_code