Overview of Storage and Indexing

Iztok Savnik, FAMNIT

Slides & Textbook

- Textbook:
  - Raghu Ramakrishnan, Johannes Gehrke, *Database Management Systems, McGraw-Hill, 3rd ed., 2007.*
- *Slides:*
  - *From „Cow Book“: R.Ramakrishnan, http://pages.cs.wisc.edu/~dbbook/*

Data on External Storage

- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order (HD 300-600 MB/s, SSD 0.5-12 GB/s)
- Tapes: Can only read pages in sequence
  - Cheaper than disks; used for archival storage (100-150 MB/s)
- File organization: Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate record
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- Architecture: Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

Alternative File Organizations

Many alternatives exist, *each ideal for some situations, and not so good in others:*

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.
- Complexity of operations on files?

Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.
  - Given data entry k*, we can find record with key k in at most one disk I/O. (Details soon …)

Alternatives for Data Entry **k*** in Index

- In a data entry k* we can store:
  - Data record with key value **k,** or
  - <**k**, rid of data record with search key value **k**>, or
  - <**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

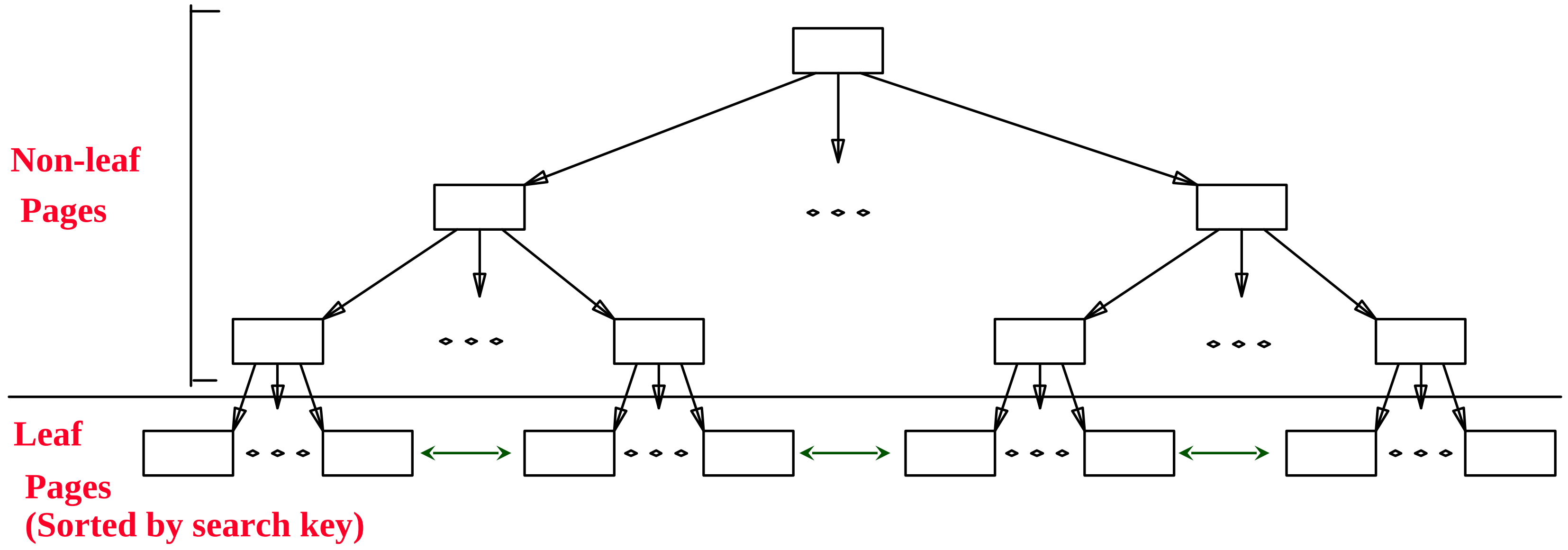
Alternatives for Data Entries (Contd.)

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

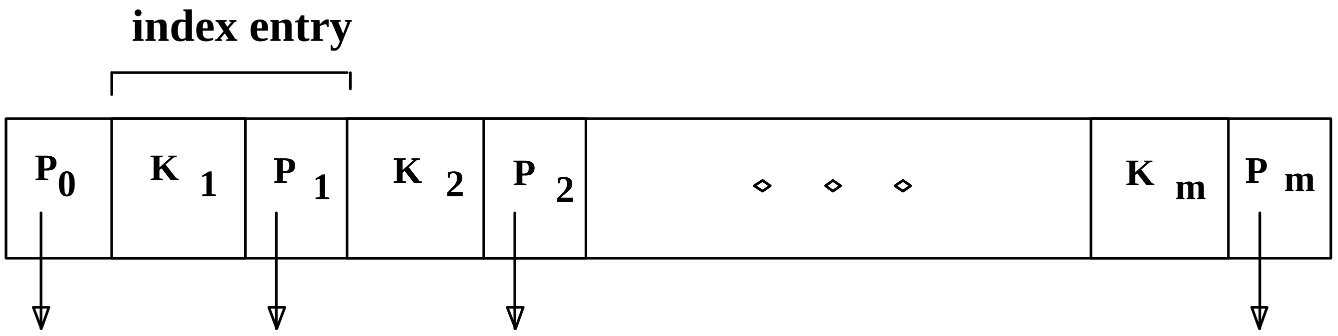
Alternatives for Data Entries (Contd.)

- 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, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

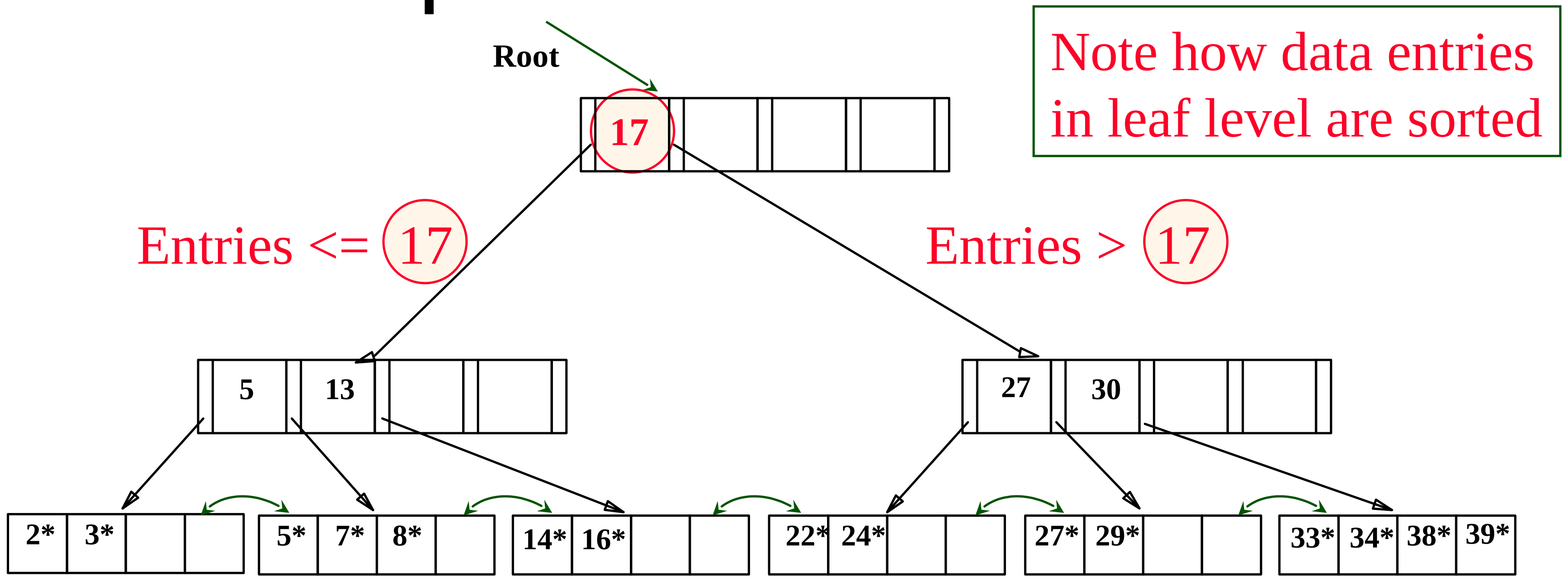
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries;* only used to direct searches:



Example B+ Tree



- Find 28*? 29*? All > 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 Indexes

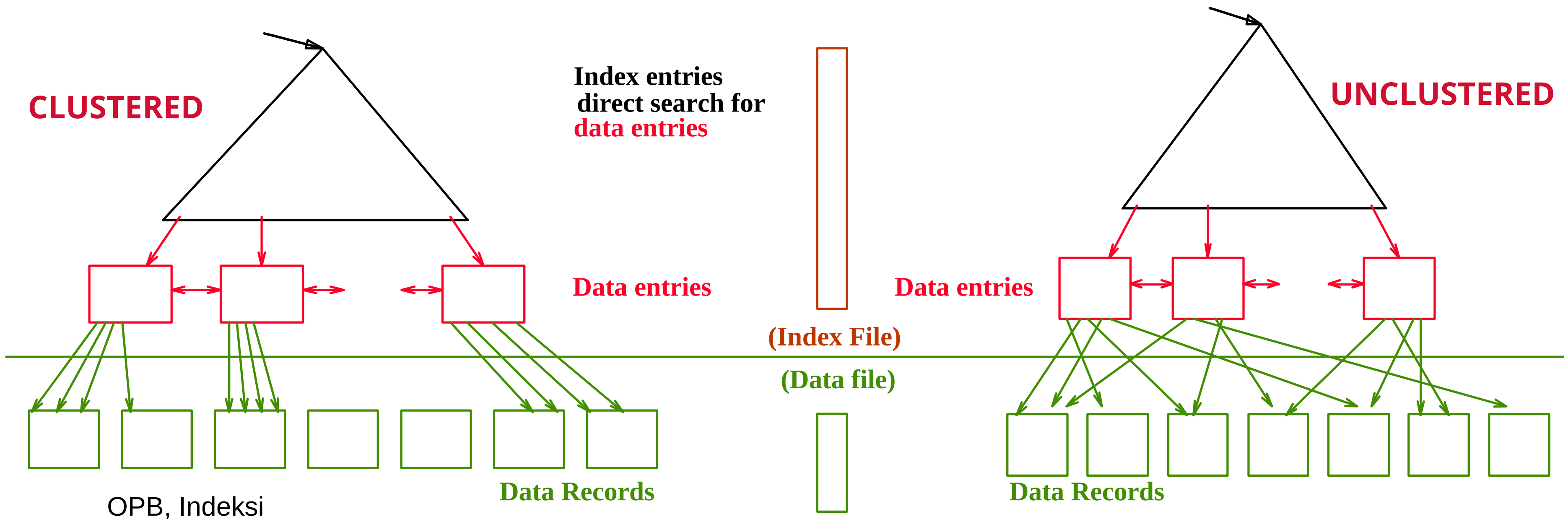
- Good for equality selections.
- Index is a collection of *buckets.*
  - Bucket = *primary* page plus zero or more *overflow* pages.
  - Buckets contain data entries.
- *Hashing function* **h**: **h**(*r*) = bucket in which (data entry for) record *r* belongs. **h** looks at the *search key* fields of *r.*
  - *No need for "index entries" in this scheme.*

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; in practice, clustered also implies Alternative 1 (since sorted files are rare).
  - 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.)


CLUSTERED
Index entries
direct search for
data entries
UNCLUSTERED
Data entries
Data entries
(Index File)
(Data file)
Data Records
Data Records