UNIT - 4

UNIT – 4
Representing Data Elements & Index Structures

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

Tapes: Can only read pages in sequence

- Cheaper than disks; used for archival storage.

File organization and Indexing:

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.

Primary and secondary Indexes:

Primary vs. secondary: If search key contains primary key, then called primary index.

Unique index: Search key contains a candidate key.

Clustered and unclustered:

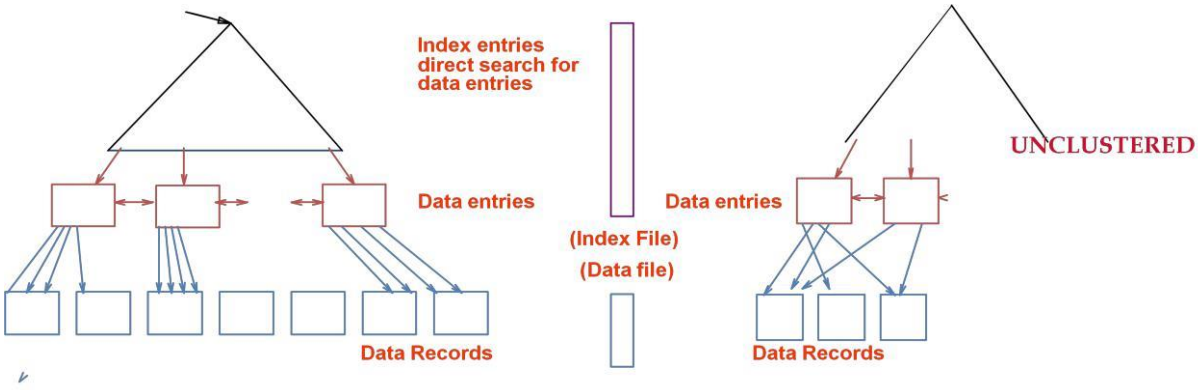
Clustered vs. unclustered: If order of data records is the same as, or `close to', order of data entries, then called clustered index.

- 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).

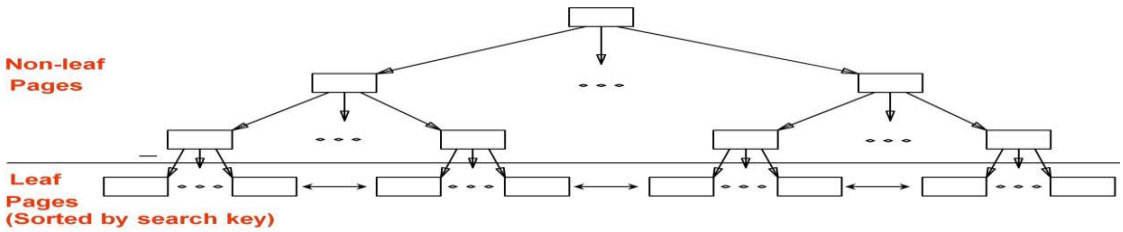


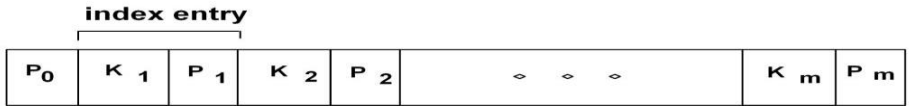
Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)

Index Data Structures:

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).





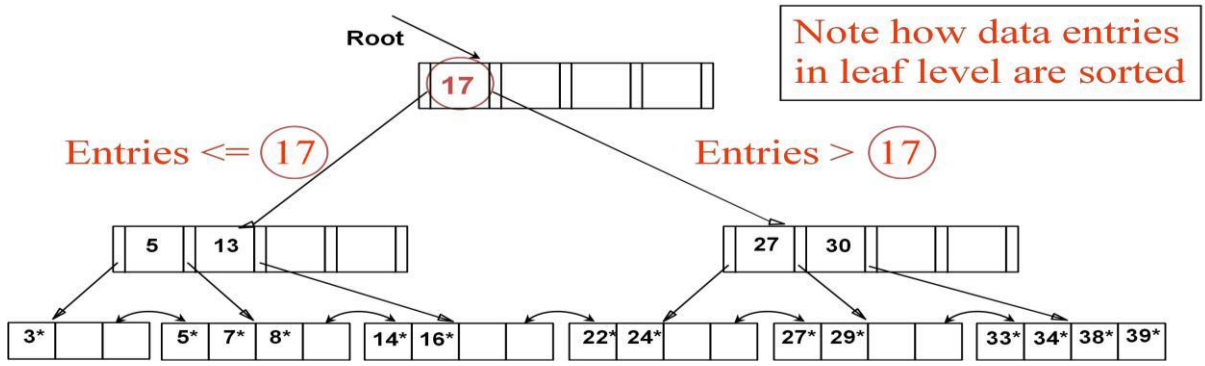
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

- Given data entry k*, we can find record with key k in at most one disk I/O.

(Details soon …)

B+ Tree Indexes

Example B+ Tree



1. Find 28*? 29*? All > 15* and < 30*

2. Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.

   - And change sometimes bubbles up the tree

Hash-Based Indexing:

- 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 atthe *search key* fields of *r*.
- *No need for "index entries" in this scheme.*

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**.

Tree Based Indexing:

– Examples of indexing techniques: B+ trees, hash-based structures

– Typically, index contains auxiliary information that directs searches to the desired data entries

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.

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

– **B:** The number of data pages

– **R:** Number of records per page

– **D:** (Average) time to read or write disk page

– Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.

– Average-case analysis; based on several simplistic assumptions.

Choice of Indexes

1. What indexes should we create?

– Which relations should have indexes? What field(s) should be the search key?

Should we build several indexes?

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

Clustered? Hash/tree?

1. One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index.

If so, create it.

– Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!

– For now, we discuss simple 1-table queries.

Before creating an index, must also consider the impact on updates in the workload!

– Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines

Attributes in WHERE clause are candidates for index keys.

– Exact match condition suggests hash index.

– Range query suggests tree index.

Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.

Multi-attribute search keys should be considered when a WHERE clause contains several conditions.

– Order of attributes is important for range queries.

– Such indexes can sometimes enable index-only strategies for important queries.

For index-only strategies, clustering is not important!

B+ Tree:

B+ Tree: Most Widely Used Index. Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages) Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree. Supports equality and range-searches efficiently.

Example B+ Tree

1. Search begins at root, and key comparisons direct it to a leaf (as in ISAM).

2. Search for 5*, 15*, all data entries >= 24* ...

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

Inserting a Data Entry into a B+ Tree

Find correct leaf *L*.

Put data entry onto *L*.

– If *L* has enough space, *done*!

– Else, must *split* *L (into L and a new node L2)*

• Redistribute entries evenly, **copy up** middle key.

• Insert index entry pointing to *L2* into parent of *L*.

This can happen recursively

– To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.) Splits "grow" tree; root split increases height.

– Tree growth: gets *wider* or *one level taller at top.*

Inserting 8* into Example B+ Tree

Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.

Example B+ Tree After Inserting 8*

1. Deleting a Data Entry from a B+ Tree

2. Start at root, find leaf *L* where entry belongs.

3. Remove the entry.

– If L is at least half-full, *done!*

– If L has only **d-1** entries,

- Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.
- If re-distribution fails, *merge L* and sibling.

If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*. Merge could propagate to root, decreasing height.

Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

Deleting 19* is easy.

Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.... And Then Deleting 24*

Must merge.

Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).

Hash Based Indexing:

Bucket: Hash file stores data in bucket format. Bucket is considered a unit of storage. Bucket typically stores one complete disk block, which in turn can store one or more records.

Hash Function: A hash function h, is a mapping function that maps all set of search-keys K to the address where actual records are placed. It is a function from search keyto bucket addresses.