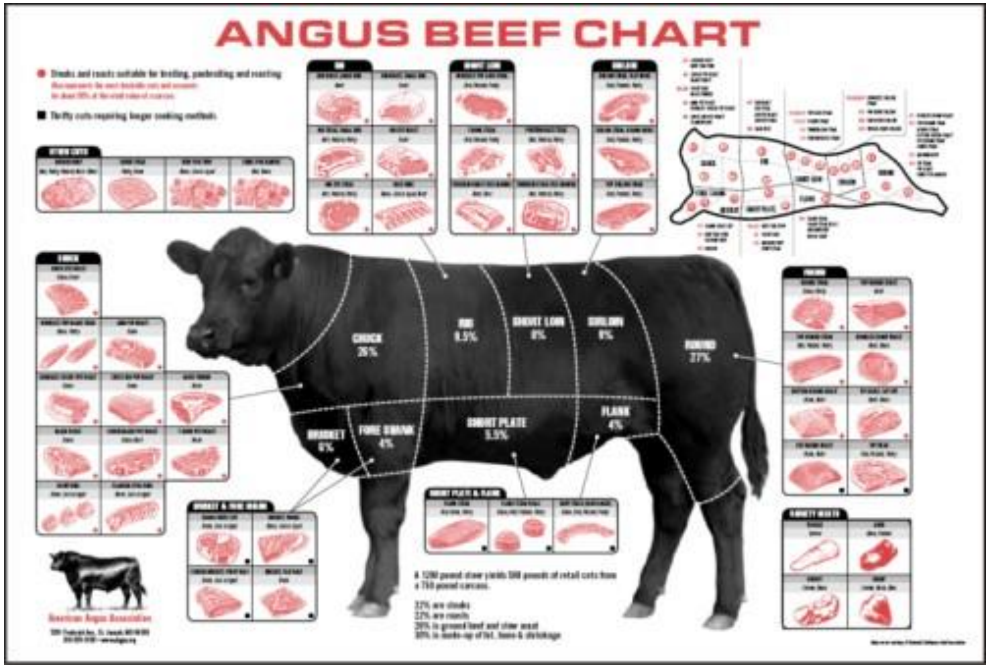
Overview of Storage and Indexing

Chapter 8



Data on External Storage

- Solid State Disks, Secure Digital (SD) non-volatile memory:
  - Block addressable storage device, relatively symmetric R/W speeds, Access latency
- Disks: Can retrieve random *page* at fixed cost
  - But *reading consecutive pages is much cheaper* than reading them in random order
- Tapes: Can only read pages sequentially
  - Cheaper than disks; used for archival storage
- 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:*

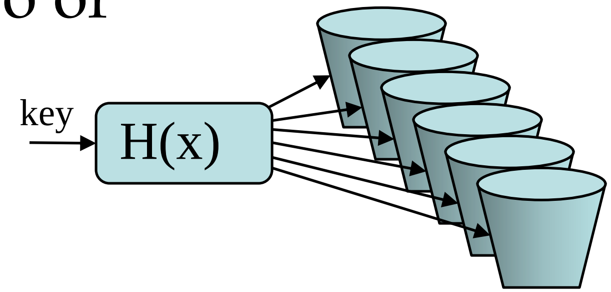
- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed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

Indexes

- An *index* is an axillary data structure that speeds up selections on the *search key fields* of the index.
  - Any subset of attributes from a relation can be a search key.
  - *Search key* is not necessarily a relation *key* (a set of fields that uniquely identify a tuple 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 …)

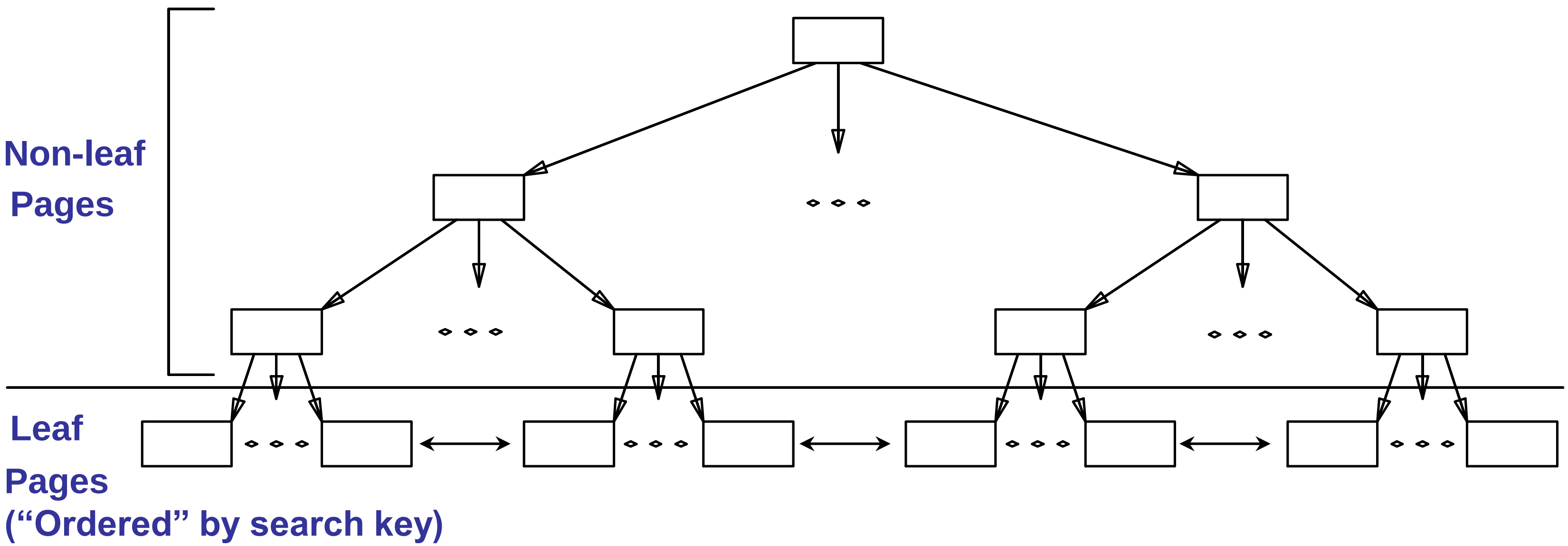
Hash-Based Index

- Place all records with a common attribute together.
- Index is a collection of *buckets*.
  - Bucket = *primary* page plus zero or more *overflow* pages.
  - Buckets contain data entries.

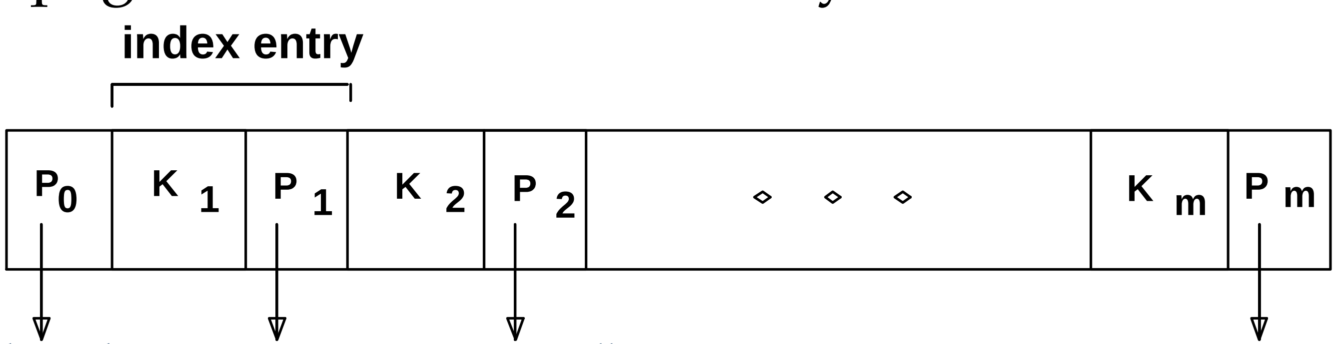


- *Hashing function, r = h(key) :* Mapping from the index's *search key* to a bucket in which the (data entry for) record *r* belongs.

Tree-Based Index



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



Alternative Data/Index Organizations

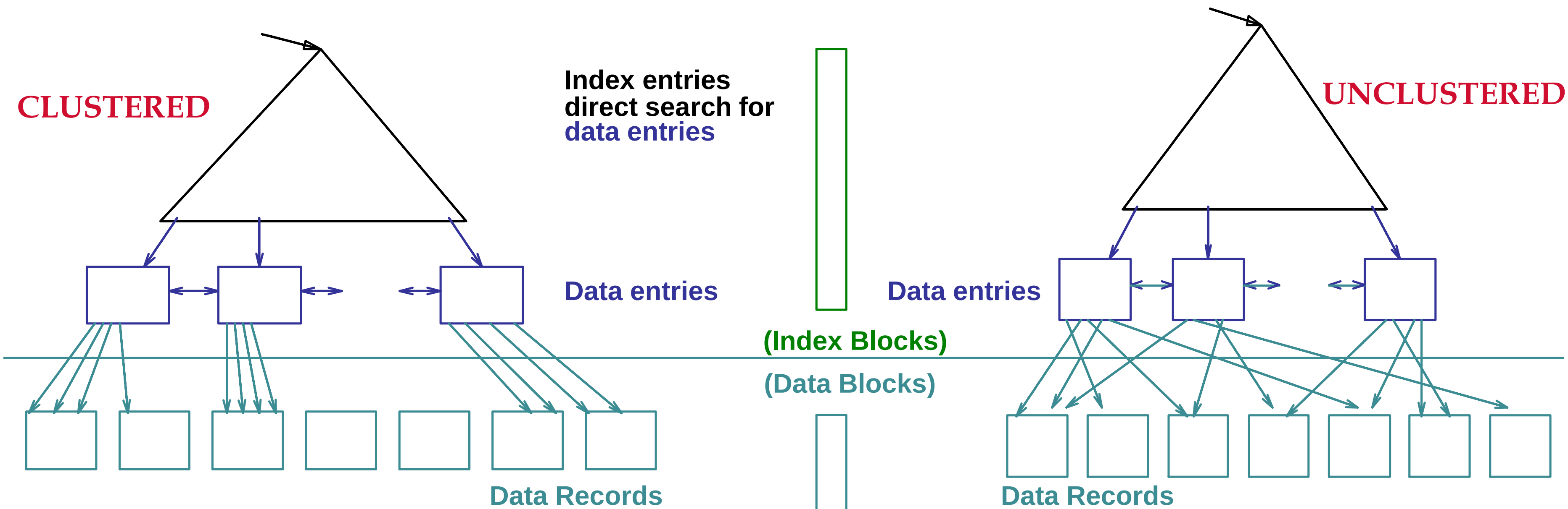
- In data entry k* we store one of the following:
  - The actual data record with its key **k** (clustered)
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Data organization choice is independent of the indexing method.
  - Clustered indices save on accesses, but you can only have 1 clustered index per relation
  - Unclustered alternatives tradeoff uniformity of index entries verses size considerations
  - Often, indices contains auxiliary information

Index Classifications

- *Primary* vs. *Secondary*: If search key contains primary key, then it is called a *primary index*.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *Unclustered*:
  - *Clustered*: tuples are sorted by search key and stored sequentially in data blocks
  - A file can be clustered on at most one search key.
  - *Unclustered*: search keys are stored with *record ids* (rids) that identify the block containing the associated tuple

Clustered vs. Unclustered Index

- Index type (Hash or Tree) is independent of the data's organization (clustered or unclustered).
  - To build clustered index, we must first sort the records (perhaps allowing for some free space on each page for future inserts).
  - Later inserts might create overflow pages. Thus, eventual order of data records is "close to", but not identical to, the sort order.



Costs / Benefits of Indexing

- Adding an index incurs
  - Storage overhead
  - Maintenance overhead
- Without indexing, searching the records of a database for a particular record would require on average

Number of Records * Cost to read a Record * 0.5

(assumes records are in random order)

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 a block
- 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.

☞ *Good enough to show the overall trends!*

Comparing File Organizations

- Heap file (random record order; insert at eof)
- Sorted files, sorted on <*age, sal*>
- Clustered B+ tree file, clustered on search key <*age, sal*>
- Heap file with unclustered B+ tree index on search key <*age, sal*>
- Heap file with unclustered hash index on search key <*age, sal*>

Operations to Compare

- Scan: Fetch all records from disk
- Equality search
- Range selection
- Insert a record
- Delete a record

```
SELECT *
FROM Emp
```

```
SELECT *
FROM Emp
WHERE Age = 25
```

```
SELECT *
FROM Emp
WHERE Age > 30
```

```
INSERT
INTO Emp(Name, Age, Salary)
VALUES('Jordan', 49, 3000000)
```

```
DELETE
FROM Emp
WHERE Name ='Bristow'
```

Assumptions in Our Analysis

- Heap Files:
  - Equality selection is on key → exactly one match
- Sorted Files:
  - Files compacted after deletions.
- Indexes:
  - Search key overhead = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy => File size = 1.25 data size
  - Tree: 67% occupancy (this is typical).
    - Implies file size = 1.5 data size
    - Tree Fan-out = F

Assumptions (contd.)

- Scans:
  - Leaf levels of a tree-index are chained.
  - Index data-entries plus actual file scanned for unclustered indexes.
- Range searches:
  - We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

Cost of Operations

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	$D\log_2 B$	$D\log_2 B$ + #matches	Search + BD	Search + BD
Clustered	1.5BD	$D\log_F 1.5B$	$D\log_F 1.5B$ + #matches	Search + D	Search + D
Unclustered tree index	BD(R+0.15)	$D(1+\log_F 0.15B)$	$D(1+\log_F 0.15B+$ #matches$)$	$D(\log_F 0.15B)$	Search + 2D
Unclustered hash index	BD(R+0.125)	2D	BD	4D	Search + 2D

☞ *Several assumptions underlie these (rough) estimates! We'll cover them in the next few lectures.*

Indexes and *Workload*

- 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 the conditions applied likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Index-Only Plans

- Some queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

<E.dno>

Index stores a count of tuples with the same key

```
SELECT  E.dno, COUNT(*)
FROM  Emp E
GROUP BY  E.dno
```

A Tree index on *<E.dno,E.sal>* would give the anwser

```
SELECT  E.dno, MIN(E.sal)
FROM  Emp E
GROUP BY  E.dno
```

<E. age,E.sal>

or

<E.sal, E.age>

Average the index keys

```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

Summary

- Alternative file organizations, each suited for different situations.
- If selection queries are frequent, data organization and *indices* are important.
  - Hash-based indexes
  - Sorted files
  - Tree-based indexes
- An *index* maps search-keys to associated tuples.
- Understanding the *workload* of an application, and its performance goals, is essential for a good design.