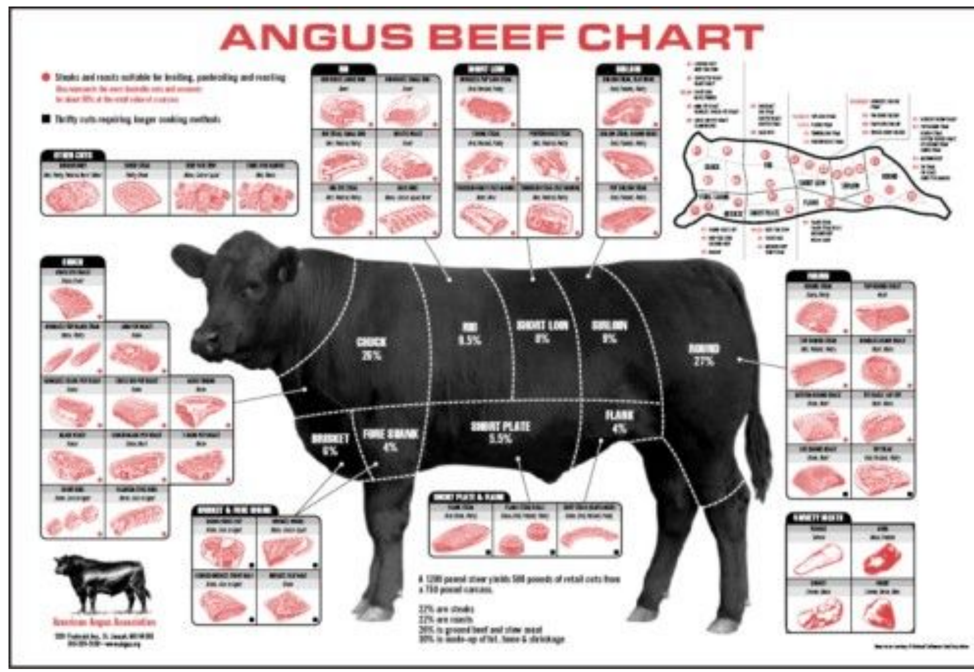
Overview of Storage and Indexing

Due date of Problem Set #2 is now midnight next Tuesday (9/22).
Problem Set #3 will be issued the same day.

I have modified my office hours to W, Th from 11am-noon. Still open door, and group programming oriented.



"Physical" Storage

- Solid State Disks, Secure Digital (SD) non-volatile memory:
  - Block addressable storage device, relatively symmetric R/W speeds, low latency, but number of write cycles is limited.
- Disks: Can retrieve random *page* at fixed cost
  - Also block oriented with high latency, but *reading consecutive blocks is much cheaper* than reading them in random order
- Tapes: Can only read pages sequentially
  - Block oriented, even higher latency. Practically limited to sequential access. But, cheaper than disks; used for archival storage

- Key Observations: The attributes of physical storage vary both within and between types. The data-life cycle is longer than the distance between technology and performance changes. There is a constant need to retrofit new technolgies into existing storage systems.
- Our goal: Create an abstraction over physical storage that is both effcient and future proof.

Pages and Blocks

- Pages: Pages are the *minimally addressable unit* of data access. Generally, a page is composed one or more "device" blocks.

- File organization: Method of arranging *records* in pages.
  - Multiple records are placed in *pages* whose size is a good match to a block.
  - A page that spans multiple sequential blocks has certain advantages.
  - Record id (rid) is an identifier that sufficient to locate record's page
  - Indexes are data structures that allow us to find the record ids of records with given values using "search key" fields.
  - Indexes are also kept on phyical storage, just like data records.

- Architecture: Buffer manager stages pages from external storage to main memory in a datastructure called a "buffer pool". File and index layers both make calls to the buffer manager.

Alternative File Organizations

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

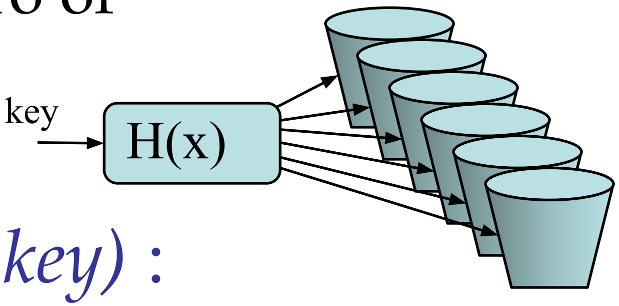
- Heap files (random record order) : Suitable when typical access is a file scan retrieving all records.
- Sorted Files (records sorted by some ordered set of attributes) : Best if records are commonly retrieved in some order, or only a `range' of records is needed. A file can only be sorted by a single set of attributes.
- Indexes: Data structures that organize *record IDs* to simplify searches and queries. Can have many.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates can be faster than in sorted files.

Indexes

- An *index* is an auxillary data structure that accererates queries using 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 …)

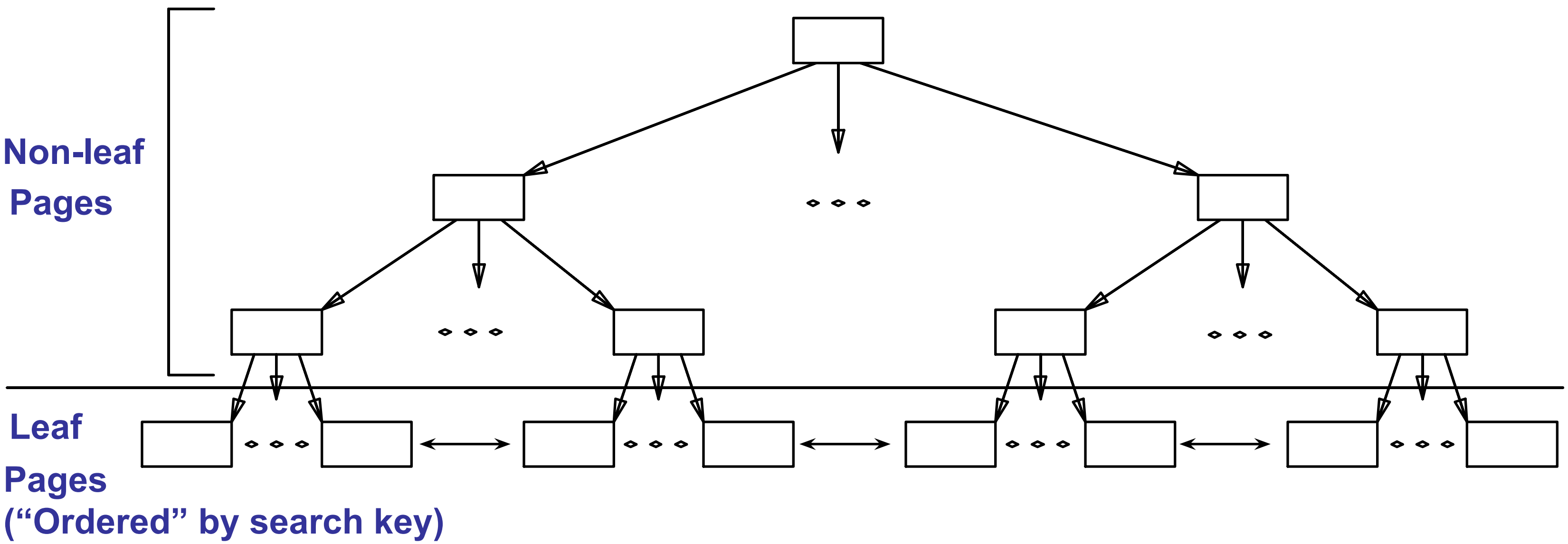
Hash-Based Index

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

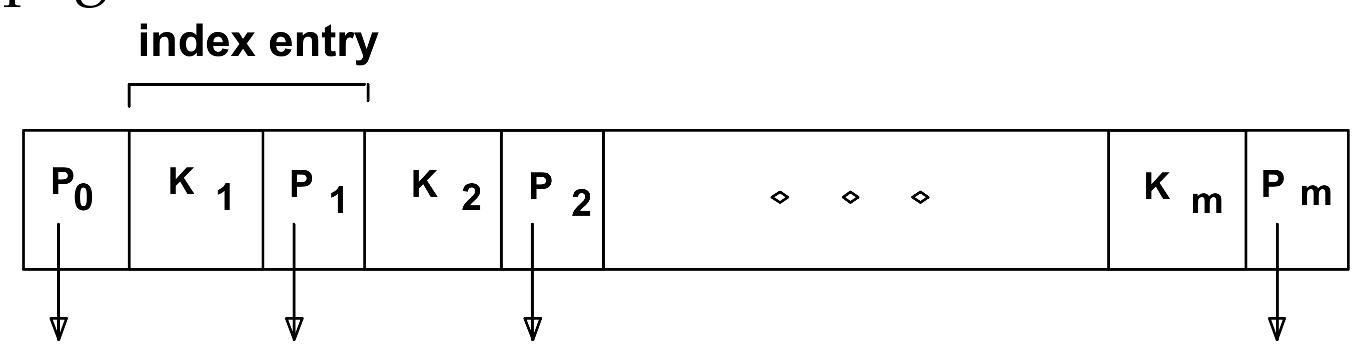


- *Hashing function, r = h(search key) :* Mapping from the index's *search key* to a bucket in which the (data entry for) record *r* belongs.

Tree-Based Index



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



Alternative Data/Index Organizations

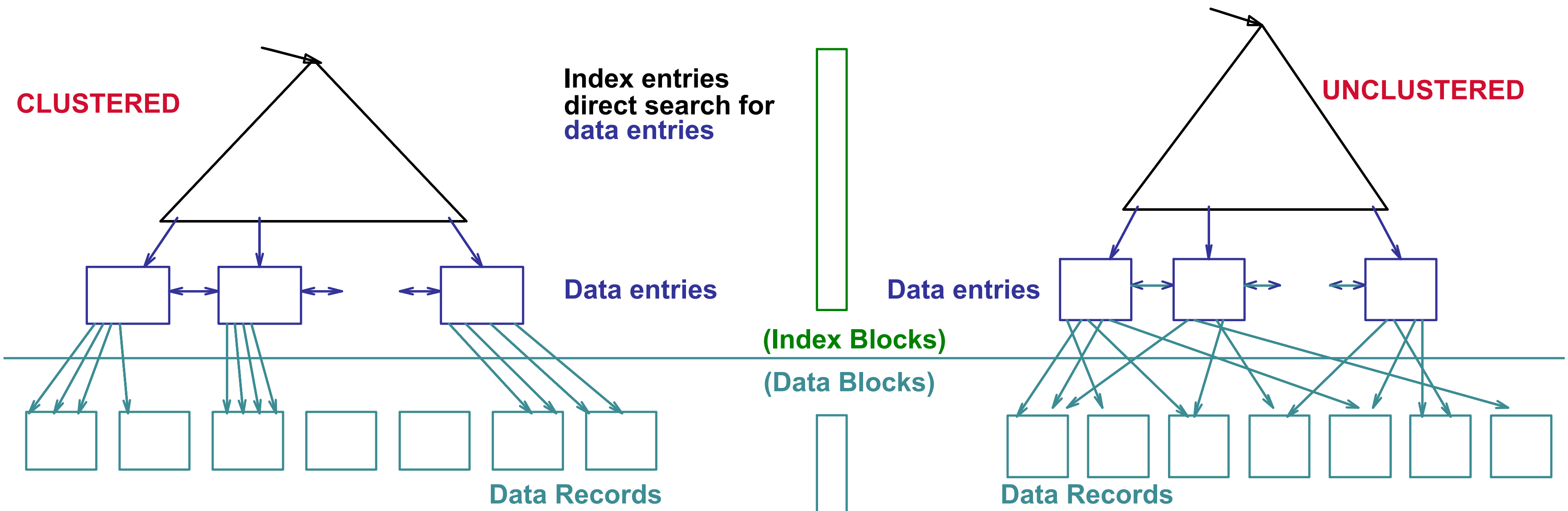
- In leaf pages we can store one of the following:
  - Actual data records with the key's value or range (clustered index), which implies a sorted file
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Data organization choice is independent of the indexing method.
  - Clustered indices save on accesses, but you can only have 1 clustered index per relation
  - Unclustered alternatives gather rids, and then plan how to access pages.

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.
  - Each search key leads to one record
- *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 page 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; inserts are placed at end-of-file)
- 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', 57, 16000000000)
```

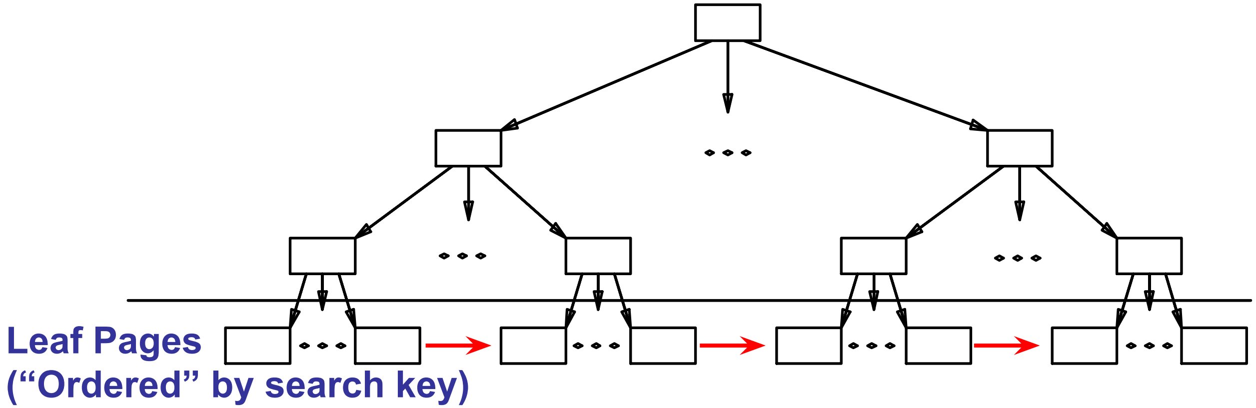
```
DELETE
FROM Emp
WHERE Name ='Bristow'
```

Assumptions in Our Analysis

- Heap Files:
  - Equality selection is on key □ exactly one match
- Sorted Files:
  - Files are 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)	$D(2+\log_F 0.15B)$	Search + 2D
Unclustered hash index	BD(R+0.125)	2D	BD	3D	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 a relation if a suitable index is available.

<E.dno>

Index on *dno* has rids of tuples with the same key. Just count them.

```
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 qualifying index keys

```
SELECT AVG(E.sal)
FROM   Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 30000 AND 50000
```

Example

```
import time
import sqlite3

Q = """SELECT C.name, SUM(count) AS females
       FROM County C, Demographics D
       WHERE C.fips=D.fips
         AND D.year = 2020 AND D.sex='female'
       GROUP BY C.name
       HAVING SUM(count) > 100000"""

db = sqlite3.connect("NCCOVID19.db")
db.row_factory = sqlite3.Row
cursor = db.cursor()


start = time.time()
cursor.execute(Q)
countyList = []
for row in cursor:
    countyList.append((row['name'],row['females']))
print("number of counties with more than 100,000 females = %d (%6.4f secs)" % (len(countyList),
time.time()-start))

for row in sorted(countyList, reverse=True):
    print(row)
```

```
number of counties with more than 100,000 females = 12 (0.0276 secs)
('Wake', 577859)
('Union', 124022)
('New Hanover', 124109)
('Mecklenburg', 577691)
('Johnston', 108371)
('Guilford', 287571)
('Gaston', 115435)
('Forsyth', 202098)
('Durham', 171391)
('Cumberland', 171159)
('Cabarrus', 111734)
('Buncombe', 138218)
```

Example

```
import time
import sqlite3

Q = """SELECT C.name, SUM(count) AS females
       FROM County C, Demographics D
       WHERE C.fips=D.fips
         AND D.year = 2020 AND D.sex='female'
       GROUP BY C.name
       HAVING SUM(count) > 100000"""

db = sqlite3.connect("NCCOVID19.db")
db.row_factory = sqlite3.Row
cursor = db.cursor()

cursor.execute("CREATE INDEX IF NOT EXISTS YearSex ON Demographics(year,sex)")
start = time.time()
cursor.execute(Q)
countyList = []
for row in cursor:
    countyList.append((row['name'],row['females']))
print("number of counties with more than 100,000 females = %d (%6.4f secs)" % (len(countyList),
time.time()-start))

for row in sorted(countyList, reverse=True):
    print(row)
```

```
number of counties with more than 100,000 females = 12 (0.0064 secs)
('Wake', 577859)
('Union', 124022)
('New Hanover', 124109)
('Mecklenburg', 577691)
('Johnston', 108371)
('Guilford', 287571)
('Gaston', 115435)
('Forsyth', 202098)
('Durham', 171391)
('Cumberland', 171159)
('Cabarrus', 111734)
('Buncombe', 138218)
```

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.