Introduction to Database Systems
CSE 414

Lecture 6:
Basic Query Evaluation and Indexes

Announcements

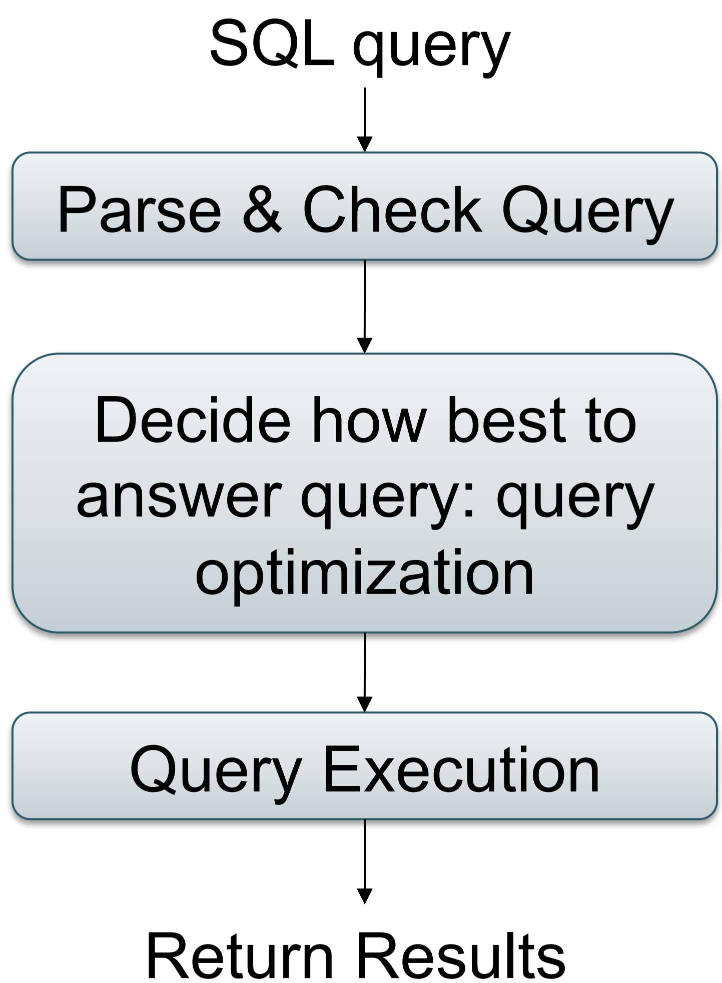
- Webquiz due tonight
- Homework 2 is posted, due next Wednesday

- Today: query execution, indexes
- Reading: 14.1, 8.4
- Monday: nested queries (6.3)

Where We Are

- We learned importance and benefits of DBMSs
- We learned how to use a DBMS
  - How to specify what our data will look like: schema
  - How to load data into the DBMS
  - How to ask SQL queries
- Today:
  - How the DBMS executes a query
  - How we can help it run faster

Query Evaluation Steps



Example

Student

ID	fName	lName
195428	Tom	Hanks
645947	Amy	Hanks
. . .		

Takes

studentID	courseID
195428	414
. . .	

Course

courseID	name
414	Databases
. . .	

```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
```

How can the DBMS answer this query?

```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
```

Possible Query Plan 1

```
for y in Takes
   if courseID > 300 then
     for x in Student
          if x.ID=y.studentID
            output *
```

Nested-loop join

SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300

Possible Query Plan 2

sort Student on ID
sort Takes on studentID (and filter on coursesID > 300)
merge join Student, Takes **on** ID = studentID
for (x,y) in merged_result **output** *

Merge join

```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
```

Possible Query Plan 3

Hash-join

create a hash-table
for x **in** Student
    insert x in the hash-table on x.ID

for y **in** Takes
    if courseID > 300
        then probe y.studentID in hash-table
            if match found
            then output *

Discussion

Which plan is best? Choose one:

- Nested loop join

```
for y in Takes
  if courseID > 300 then
   for x in Student
       if x.ID=y.studentID
        output *
```

- Merge join

```
sort Student on ID
sort Takes on studentID (and filter on coursesID > 300)
merge join Student, Takes on ID = studentID
return results
```

- Hash join

```
create a hash-table
for x in Student
    insert x in the hash-table on x.ID

for y in Takes
    if courseID > 300
      then probe y.studentID in hash-table
           if match found and additional conditions
           then return match
```

Discussion

Which plan is best? Choose one:

- Nested loop join: $O(N^2)$
  - Could be $O(N)$ when few courses>300
- Merge join: $O(N \log N)$
  - Could be $O(N)$ if tables already sorted
- Hash join: $O(N)$ expectation

```
for y in Takes
   if courseID > 300 then
    for x in Student
          if x.ID=y.studentID
            output *
```

```
sort Student on ID
sort Takes on studentID (and filter on coursesID > 300)
merge join Student, Takes on ID = studentID
return results
```

```
create a hash-table
for x in Student
      insert x in the hash-table on x.ID

for y in Takes
      if courseID > 300
          then probe y.studentID in hash-table
                  if match found and additional conditions
                  then return match
```

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Data Storage

- DBMSs store data in **files**
- Most common organization is row-wise storage
- On disk, a file is split into blocks
- Each block contains a set of tuples

10	Tom	Hanks	block 1
20	Amy	Hanks	
50	…	…	block 2
200	…		
220			block 3
240			
420			
800			

In the example, we have 4 blocks with 2 tuples each

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Data File Types

The data file can be one of:

- Heap file
  - Unsorted
- Sequential file
  - Sorted according to some attribute(s) called *key*

Note: *key* here means something different from primary key:
it just means that we order the file according to that attribute.
In our example we ordered by **ID**. Might as well order by **fName,**
if that seems a better idea for the applications running on
our database.

Index

- An **additional** file, that allows fast access to records in the data file given a search key
- The index contains (key, value) pairs:
  - The key = an attribute value (e.g., student ID or name)
  - The value = a pointer to the record
- Could have many indexes for one table

Key = means here search key

This  Is Not A Key

Different keys:

- Primary key – uniquely identifies a tuple
- Key of the sequential file – how the datafile is sorted, if at all
- Index key – how the index is organized

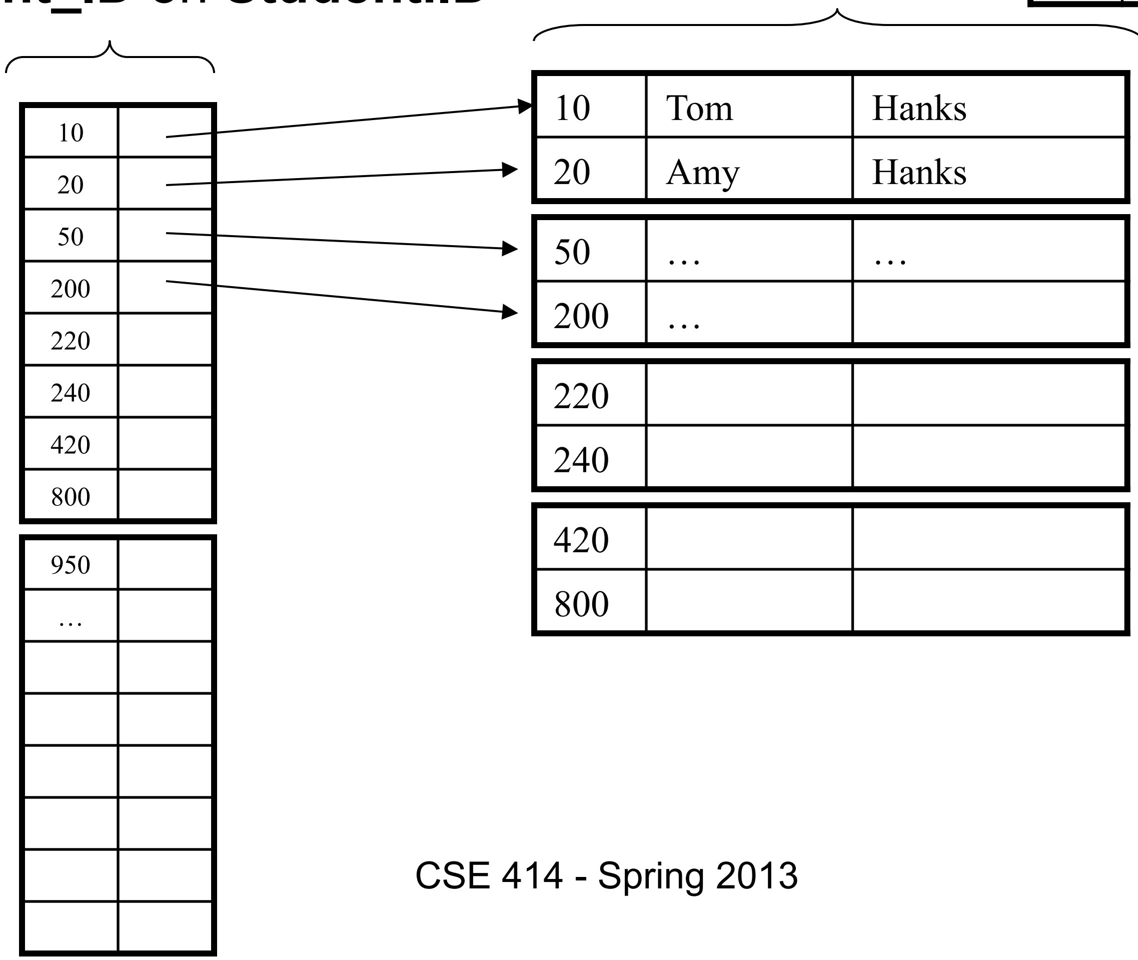
Example 1: Index on ID

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Index **Student_ID** on **Student.ID**

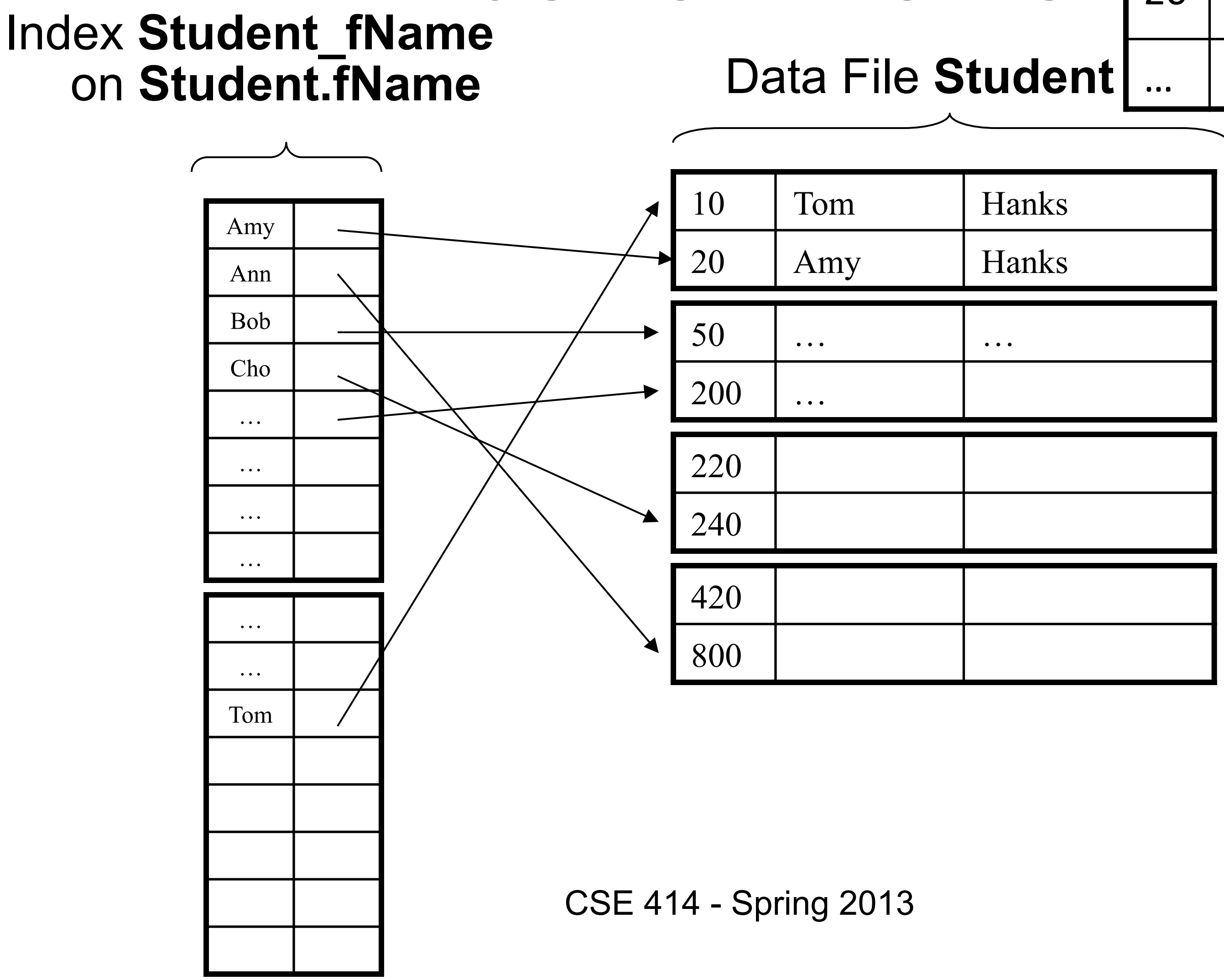
Data File **Student**



Example 2: Index on fName

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		



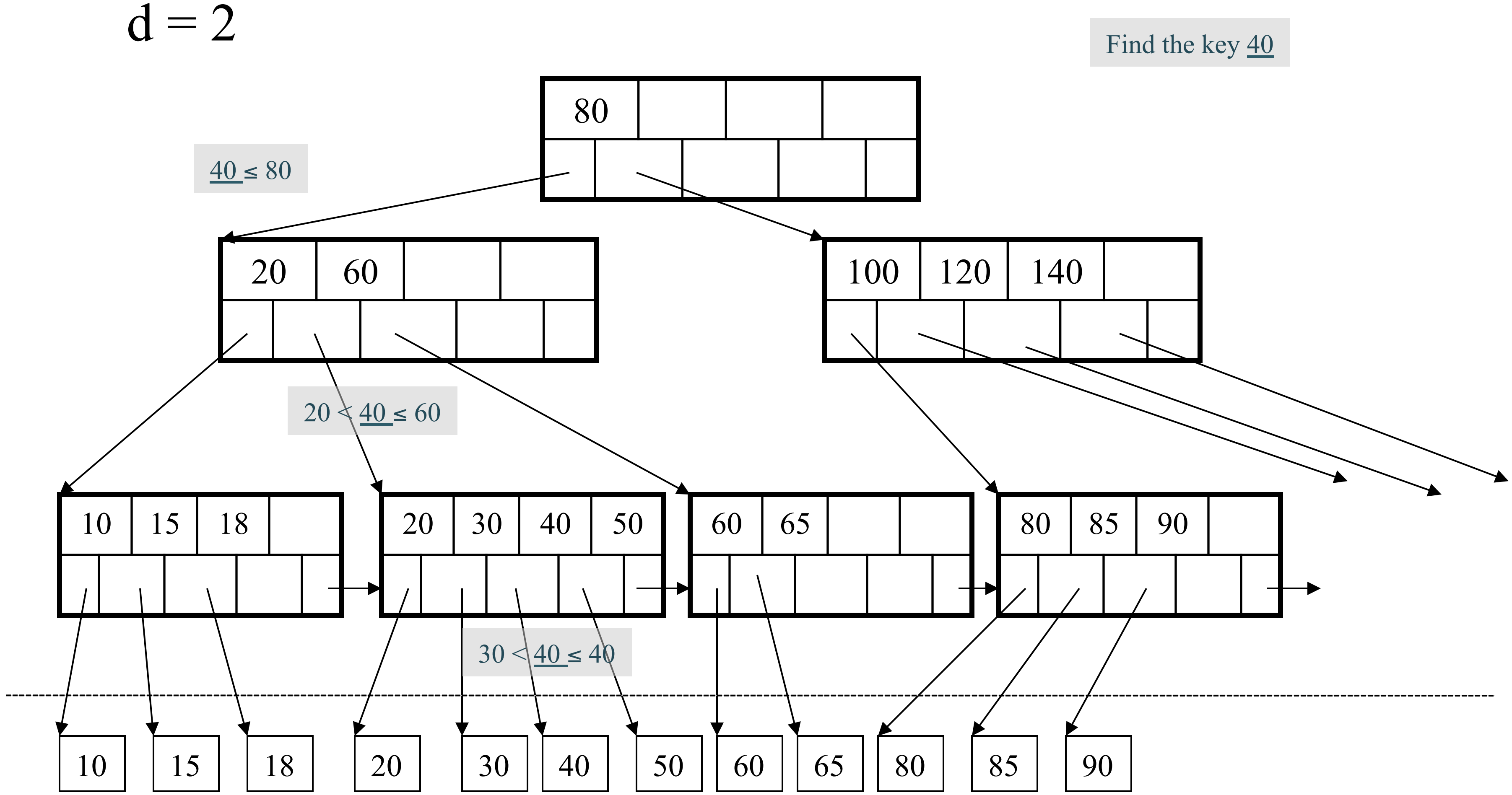
Index Organization

Several index organizations:

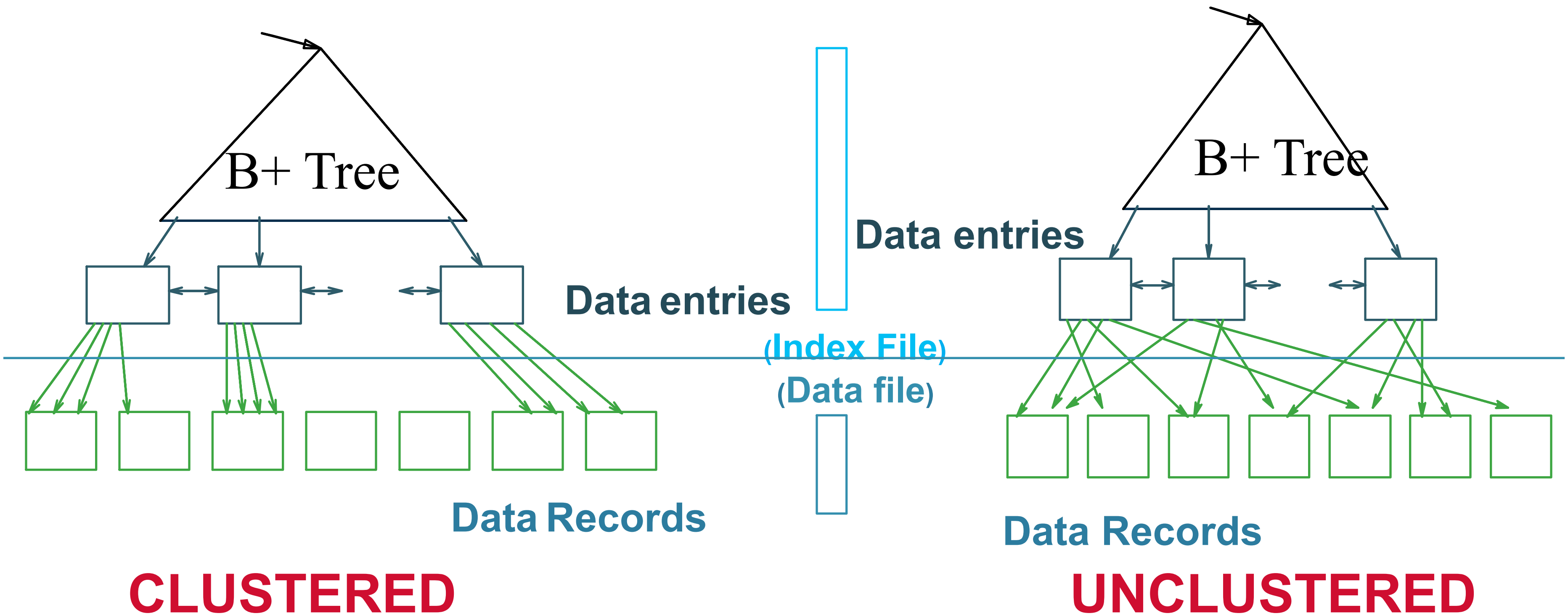
- Hash table
- B+ trees – most popular
  - They are search trees, but they are not binary instead have higher fanout
  - will discuss them briefly next
- Specialized indexes: bit maps, R-trees, inverted index

B+ Tree Index by Example

d = 2



Clustered vs Unclustered



Every table can have **only one** clustered and **many** unclustered indexes

Index Classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
    - Option 1: Data inside data file is sorted on disk
    - Option 2: Store data directly inside the index (no separate files)
  - Unclustered = records close in index may be far in data
- **Primary/secondary**
  - Meaning 1:
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
  - Meaning 2: means the same as clustered/unclustered
- **Organization** B+ tree or Hash table

Scanning a Data File

- Disks are mechanical devices!
  - Technology from the 60s; density much higher now
- We read only at the rotation speed!
- Consequence:
  Sequential scan is MUCH FASTER than random reads
  - Good: read blocks 1,2,3,4,5,…
  - Bad: read blocks 2342, 11, 321,9, …
- Rule of thumb:
  - Random reading 1-2% of the file ≈ sequential scanning the entire file; this is decreasing over time (because of increased density of disks)
- Solid state (SSD): $$$ expensive; put indexes, other "hot" data there, not enough room for everything

```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
```

Query Plan 1 Revisited

```
for y in Takes
   if courseID > 300 then
     for x in Student
         if x.ID=y.studentID
           output *
```

Assume the database has indexes on these attributes:

- **index_takes_courseID** = index on Takes.courseID
- **index_student_ID** = index on Student.ID

Index selection

```
for y in index_Takes_courseID where y.courseID > 300
   for x in Takes where x.ID = y.studentID
          output *
```

Index join

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N varchar(20), P int);
```

```
CREATE INDEX V1 ON V(N)
```

```
CREATE INDEX V2 ON V(P, M)
```

```
CREATE INDEX V3 ON V(M, N)
```

```
CREATE UNIQUE INDEX V4 ON V(N)
```

```
CREATE CLUSTERED INDEX V5 ON V(N)
```

Not supported in SQLite

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Which Indexes?

- How many indexes could we create?
- Which indexes should we create?

In general this is a very hard problem

Which Indexes?

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

- The *index selection problem*
  - Given a table, and a "workload" (big Java application with lots of SQL queries), decide which indexes to create (and which ones NOT to create!)
- Who does index selection:
  - The database administrator DBA

  - Semi-automatically, using a database administration tool

Index Selection: Which Search Key

- Make some attribute K a search key if the WHERE clause contains:
  - An exact match on K
  - A range predicate on K
  - A join on K

The Index Selection Problem 1

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

What indexes ?

The Index Selection Problem 1

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

A: V(N) and V(P) (hash tables or B-trees)

The Index Selection Problem 2

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

What indexes ?

The Index Selection Problem 2

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

A: definitely V(N) (must B-tree); unsure about V(P)

The Index Selection Problem 3

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

1000000 queries:

```
SELECT *
FROM V
WHERE N=? and P>?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

What indexes ?

The Index Selection Problem 3

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

1000000 queries:

```
SELECT *
FROM V
WHERE N=? and P>?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

A: V(N, P)

The Index Selection Problem 4

```
V(M, N, P);
```

Your workload is this

1000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100000 queries:

```
SELECT *
FROM V
WHERE P>? and P<?
```

What indexes ?

The Index Selection Problem 4

```
V(M, N, P);
```

Your workload is this

1000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100000 queries:

```
SELECT *
FROM V
WHERE P>? and P<?
```

A: V(N) secondary, V(P) primary index

Basic Index Selection Guidelines

- Consider queries in workload in order of importance
- Consider relations accessed by query
  - No point indexing other relations
- Look at WHERE clause for possible search key
- Try to choose indexes that speed-up multiple queries
- And then consider the following…

Index Selection: Multi-attribute Keys

Consider creating a multi-attribute key on K1, K2, … if

- WHERE clause has matches on K1, K2, …
  - But also consider separate indexes
- SELECT clause contains only K1, K2, ..
  - A *covering index* is one that can be used exclusively to answer a query, e.g. index R(K1,K2) covers the query:

```
SELECT K2 FROM R WHERE K1=55
```

To Cluster or Not

- Range queries benefit mostly from clustering
- Covering indexes do *not* need to be clustered: they work equally well unclustered

```
SELECT *
FROM R
WHERE K>? and K<?
```