Transaction Management & Indexing

Doris Xin, **Abdu** Alawini
University of Illinois at Urbana-Champaign

CS411: Database Systems

October 22, 2018

Announcements

- HW 3: Due by Friday 10/26 (23:59)
- Sign up for PT1 midterm demos: Due by Friday 10/26 (23:59)

  https://wiki.illinois.edu/wiki/display/cs411sfa18/Project+Track+1+Midterm+Demo+Signup
- Midterm review session: Friday 10/26 (4:00-4:50) SC 1404
  - To suggest topics to discuss in the review session, please fill this form: **https://goo.gl/forms/5fDcm80cDjmtMJ0H3**
- Please fill the early course feedback form:

  https://goo.gl/forms/SC4BYcrDy8dai8PE2
- Midterm: 10/29 in class 11-12:15 pm

Today's lecture

- TM: theory of serializability

- Storage
- Indexing

  - What is an index? Why do we need it?
  - B+ Trees

Locking and Serializability

- We said that a transaction must hold all locks until it terminates (a condition called strict locking)

- It turns out that this is crucial to guarantee serializability

  - Note that the first (bad) example could have been produced if transactions acquired and immediately released locks.

Well-Formed, Two-Phased Transactions

- A transaction is well-formed if it acquires at least a shared lock on Q before reading Q or an exclusive lock on Q before writing Q and doesn't release the lock until the action is performed
  - Locks are also released by the end of the transaction
- A transaction is two-phased if it never acquires a lock after unlocking one
  - i.e., there are two phases: a *growing phase* in which the transaction acquires locks, and a *shrinking phase* in which locks are released

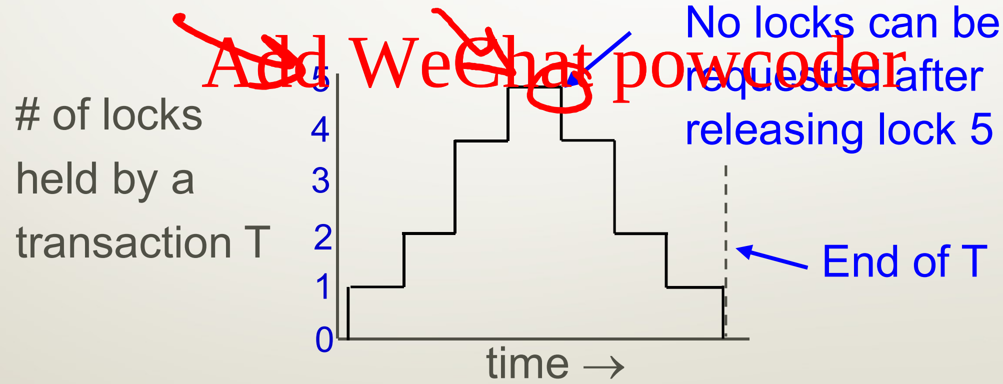
Two-Phased Locking Theorem

- If all transactions are well-formed and two-phase, then any schedule in which conflicting locks are never granted ensures serializability
  - i.e., there is a very simple scheduler!
- However, if some transaction is not well-formed or two-phase, then there is some schedule in which conflicting locks are never granted but which fails to be serializable
  - i.e., one bad apple spoils the bunch

Two Phase Locking Protocol (2PL)

2PL is a way of managing locks during a transaction T

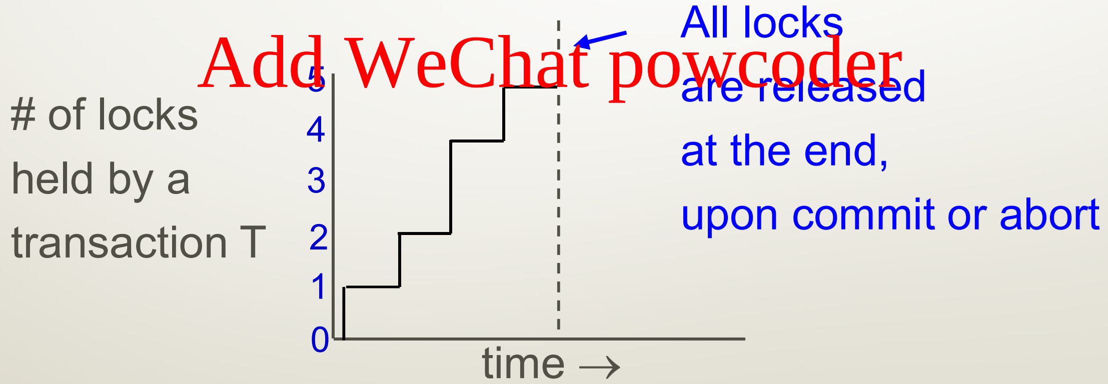
- T gets (S and X) locks gradually, as needed
- T cannot request any additional locks once it releases any locks



Strict Two Phase Locking Protocol (S2PL)

Strict 2PL is a way of managing locks during a transaction T

- T gets (S and X) locks gradually, as needed
- T holds all locks until end of transaction (commit/abort)



Which of these schedules can arise under strict 2PL

S(A)

T1:	R(A),		W(A)
T2: R(A),		W(A), R(B)	

S(A) X(A)

will not arise under S2PL

S(B) X(A) X(B) Rel(A,B)

T1:	R(B),		W(A), W(B)
T2: R(A),		W(A), R(B)	[illegible]

Rel(A,B)

S(A) X(A) S(B)

S(A)

T1: R(A),		R(B),	W(B)	
T2:	W(A)			
T3:		W(A)		R(B)

X(A)

$S_2(A)\ R_2(A)\ S_1(A)\ X_2(A)$ <wait> $X_1(A)$ <wait>

Deadlock

Strict 2PL guarantees serializability

- 2PL vs. strict 2PL
  - ~~s2PL avoids cascading abort~~
  - https://www.cs.colostate.edu/~cs430dl/yr2016sp/more_examples/Ch14/Locking.pdf
- Can prove that a Strict 2PL schedule is equivalent to the serial schedule in which each transaction runs instantaneously at the time that it commits
- This is huge: A property of each transaction (2PL) implies a property of any set of transactions (serializability)

  No need to check serializability of specific schedules
- Most DBMSs use 2PL to enforce serializability

Summary of TM

- Transactions are all-or-nothing units of work guaranteed despite concurrency or failures in the system.
- Theoretically, the “correct” execution of transactions is serializable (i.e. equivalent to some serial execution).
- Practically, this may adversely affect throughput ⇒ isolation levels.
- With isolation levels, users can specify the level of “incorrectness” they are willing to tolerate.

Outline

- ✓ TM: theory of serializability

- Storage
- Indexing

  - What is an index? Why do we need it?
  - B+ Trees

So far, we've been talking about databases in abstract terms

How are they implemented?

- How are relations stored?
- How are queries run?

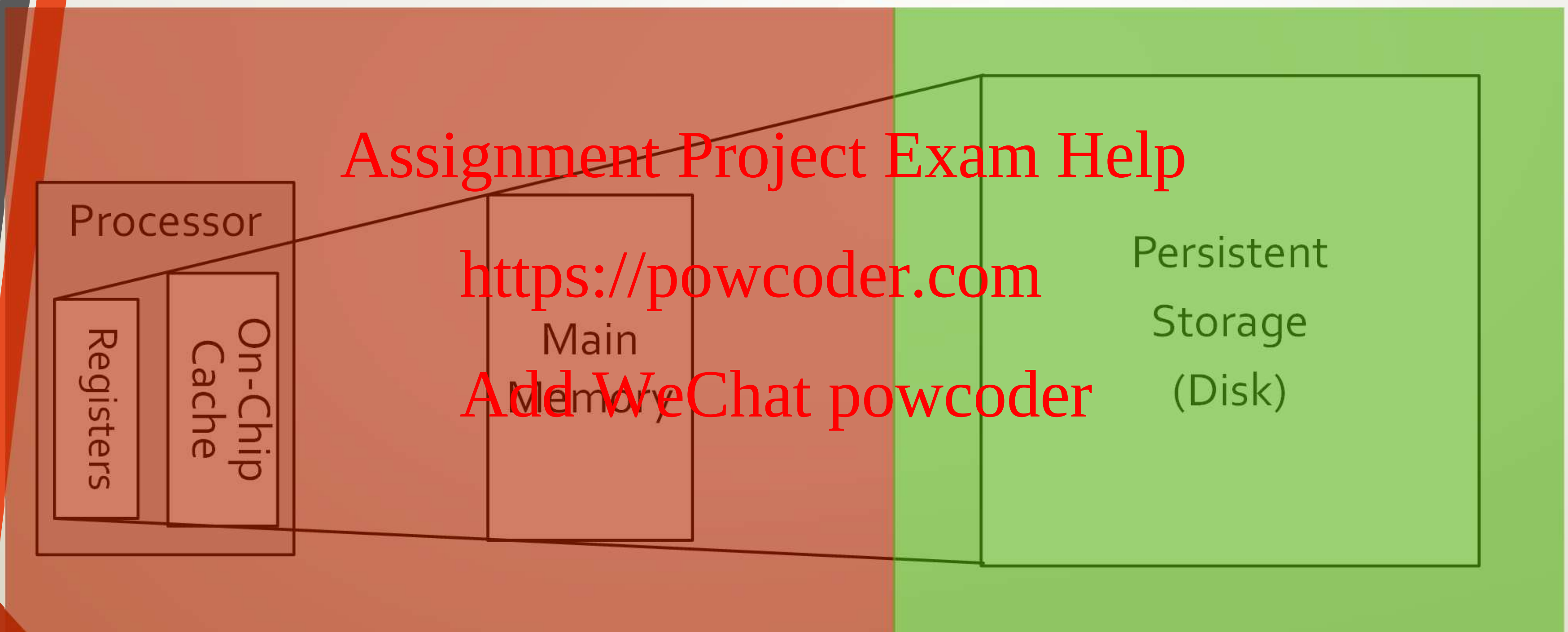
Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Storage

Simplified Computer Architecture



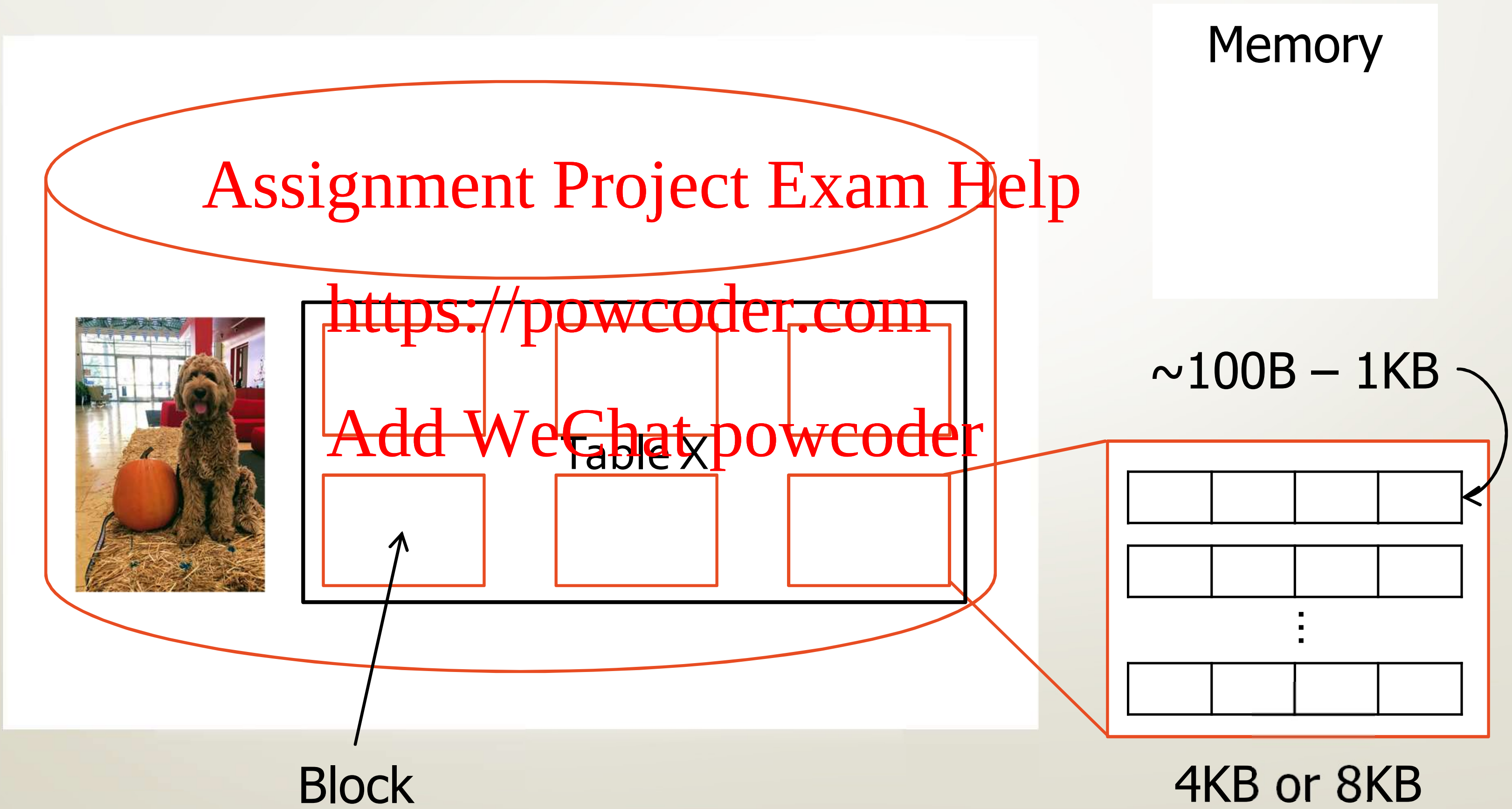
Speed (ns)	10 ns	10^2 ns	10^7 ns
Size	KB	GB	TB

Cost of Accessing Data on Disk

Main Memory

Speed	10^7 ns	10^5 ns	10^2 ns

Block size vs. record size



OK. So how do we do simple stuff?

Lookups, Insertions, Deletions

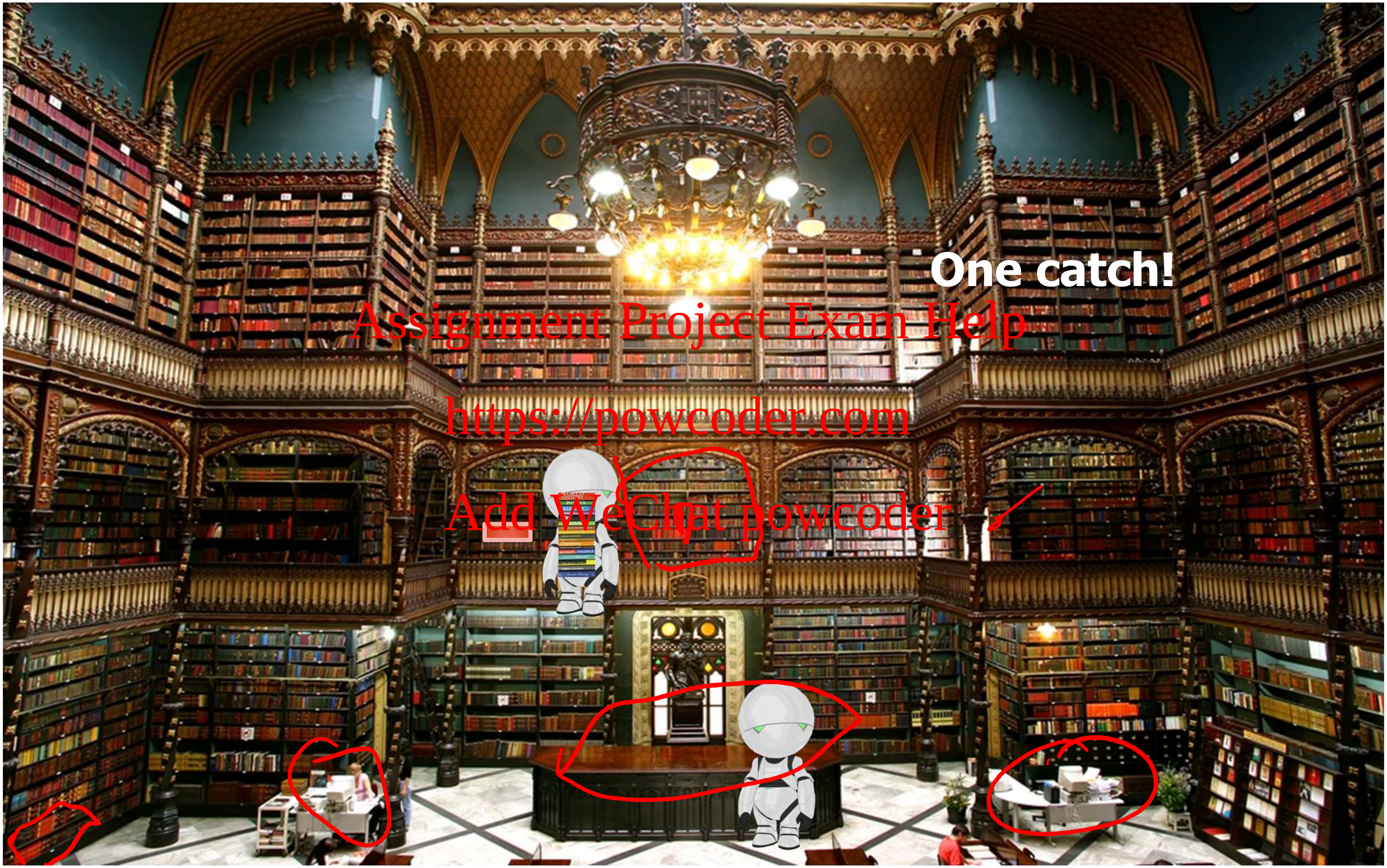
Outline

- ✓ TM: theory of serializability

- ✓ Storage
- Indexing

  - What is an index? Why do we need it?
  - B+ Trees

Indexing


One catch!

Indexes in databases

- An *index* speeds up selections on the *search key field(s)*

- Search key = any subset of the fields of a relation
  - *Search key* is not necessarily the same as a *key*

- Entries in an index: (k, r), where:
  - k = the search key
  - r = the record OR record id OR record ids OR pointers

Some terminology

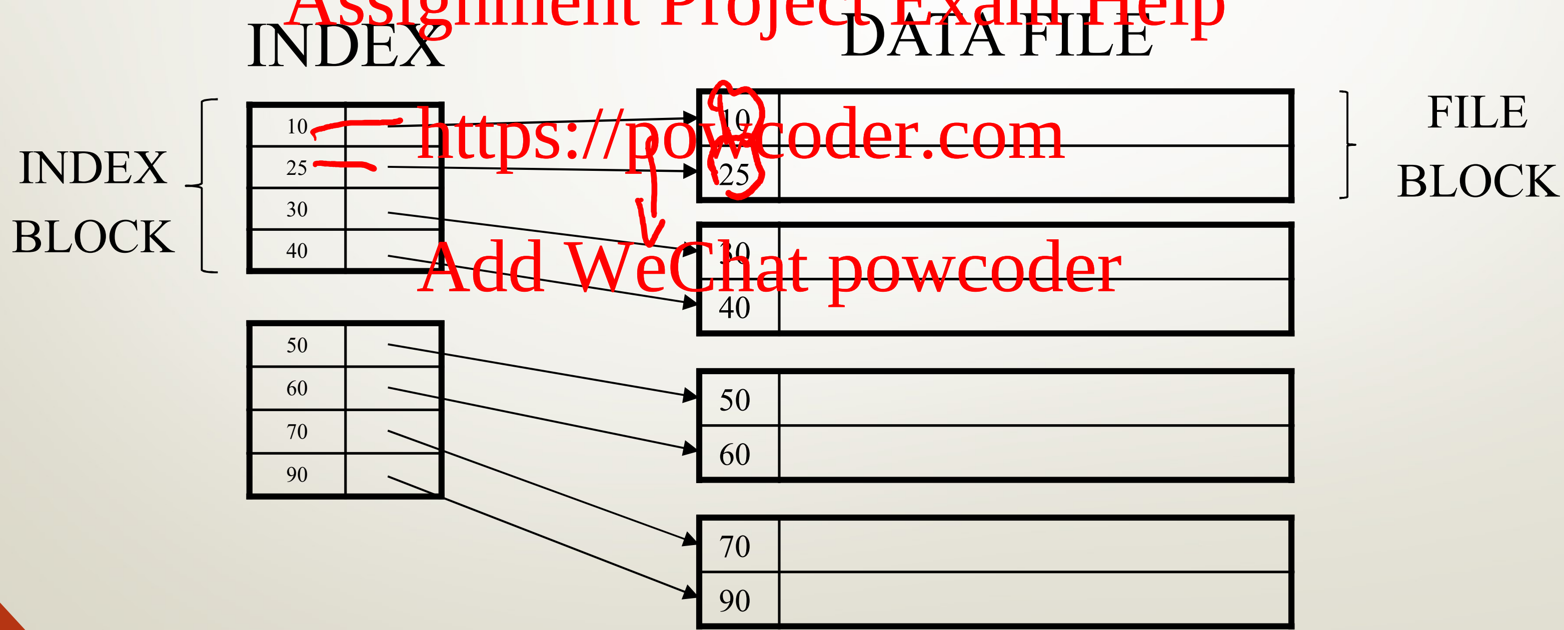
- ***Data file***: has the data corresponding to a relation
- ***Index file***: has the index
- File consists of smaller units called **blocks** (e.g. of size 4 KB or 8 KB)
- # index blocks < # data blocks. Index may even fit into main memory.

Characteristics of Indexes

- Clustered/unclustered
  - Clustered: keys sorted
  - Unclustered: keys unsorted
- Dense/sparse
  - Dense = each record has an entry in the index
  - Sparse = only some records have
- Primary/secondary
  - Primary = on the primary key
  - Secondary = on any attribute

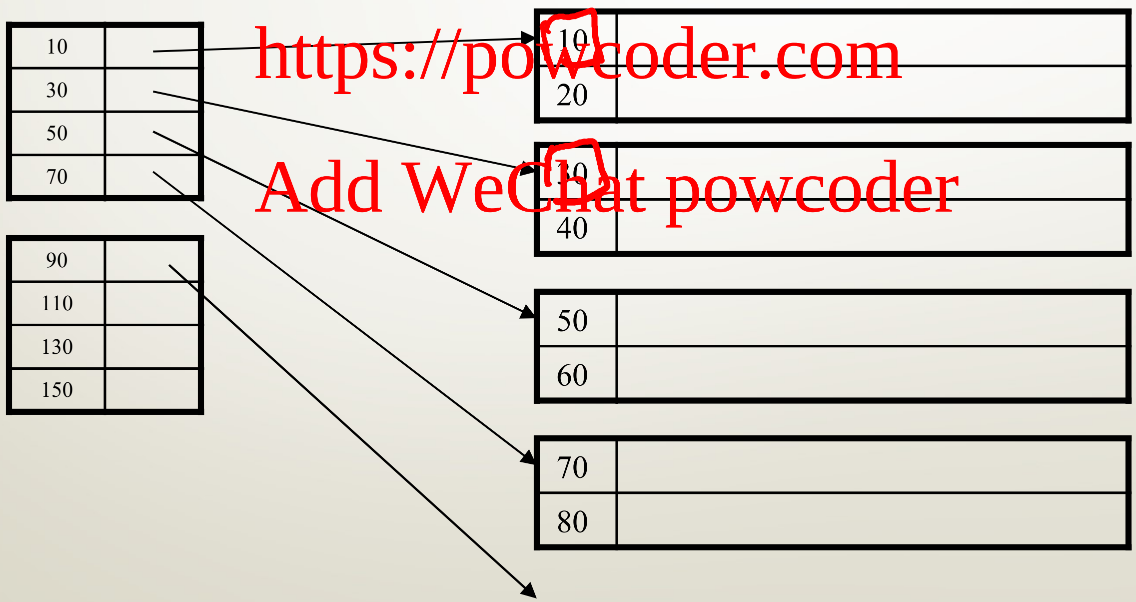
Ex: Clustered, Dense Index

- Clustered: File is sorted on the index attribute
- *Dense*: sequence of (key,pointer) pairs



Clustered, Sparse Index

- *Sparse* index: one key per data block, corresponding to the lowest search key in that block

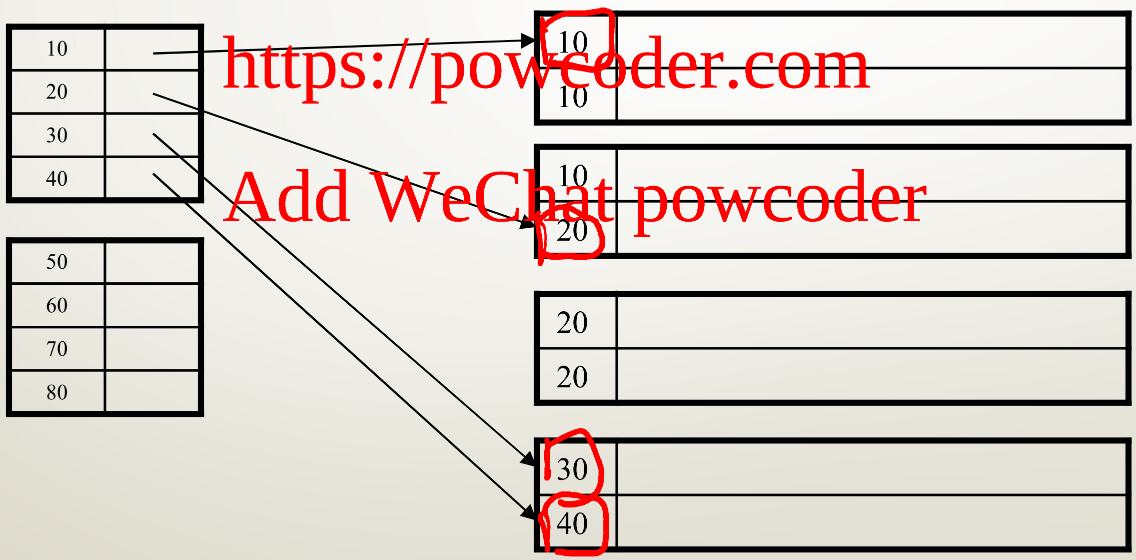


What if there are duplicate keys?

Clustered Index with Duplicate Keys

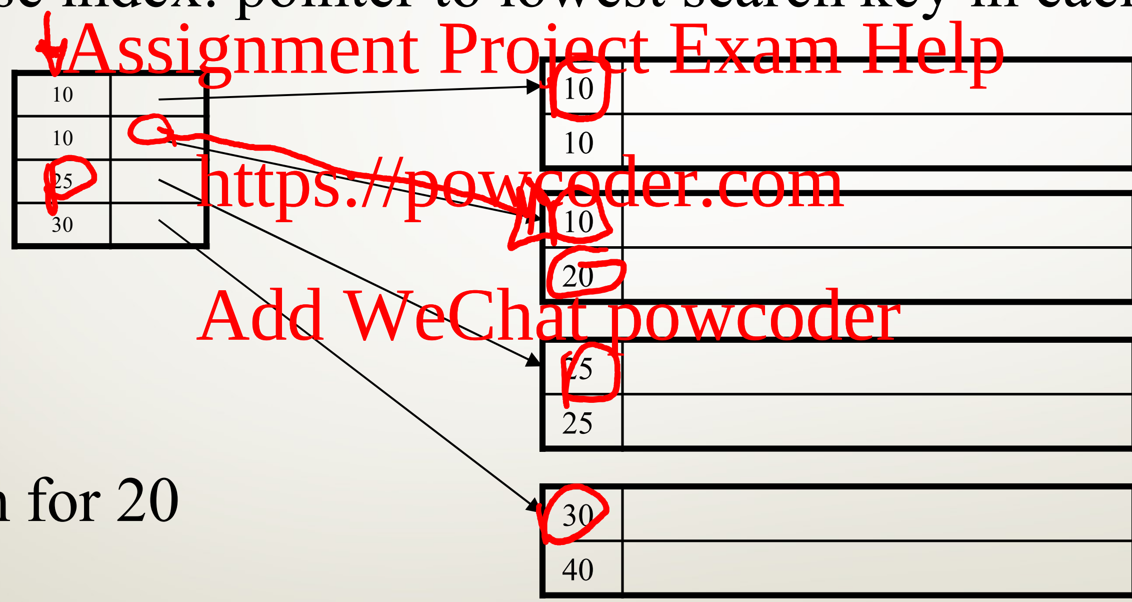
Dense index: point to the first record with that key

(must have a pointer for each new key)



Clustered Index with Duplicate Keys

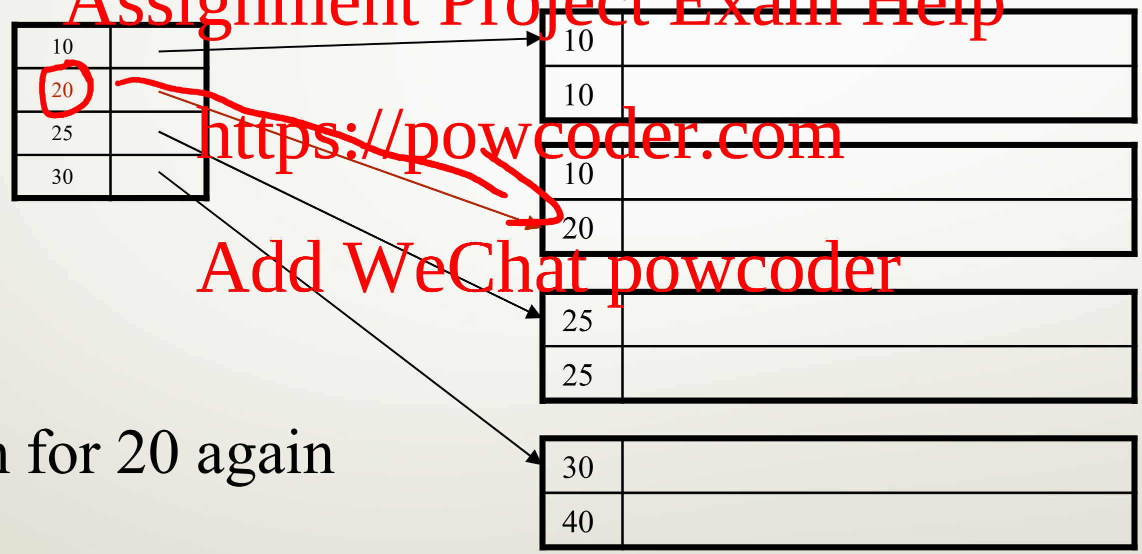
- Sparse index: pointer to lowest search key in each block



Search for 20

Clustered Index with Duplicate Keys

- *Better: pointer to lowest new search key in each block*

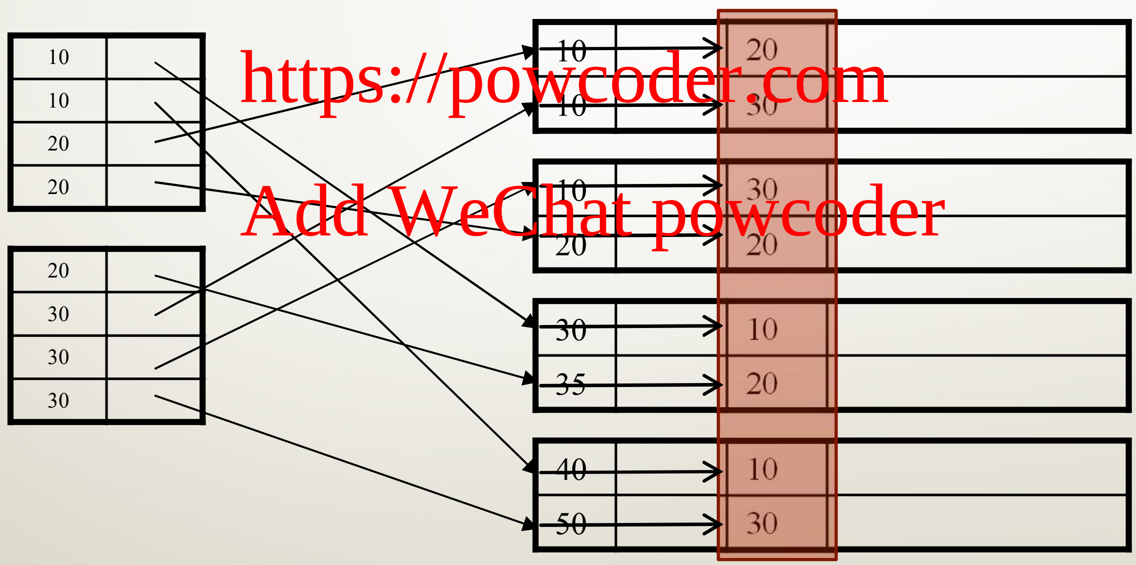


Search for 20 again

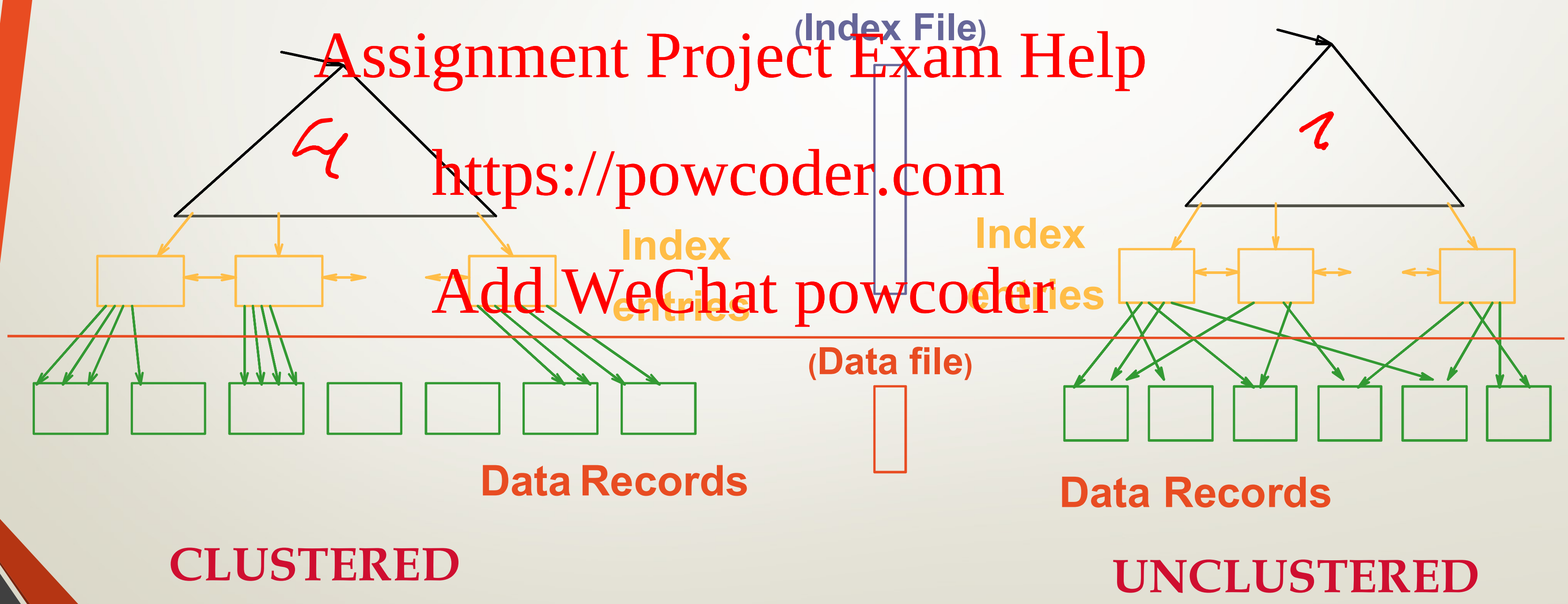
Unclustered Indexes

- Often for indexing other attributes than primary key
- Can it be sparse?

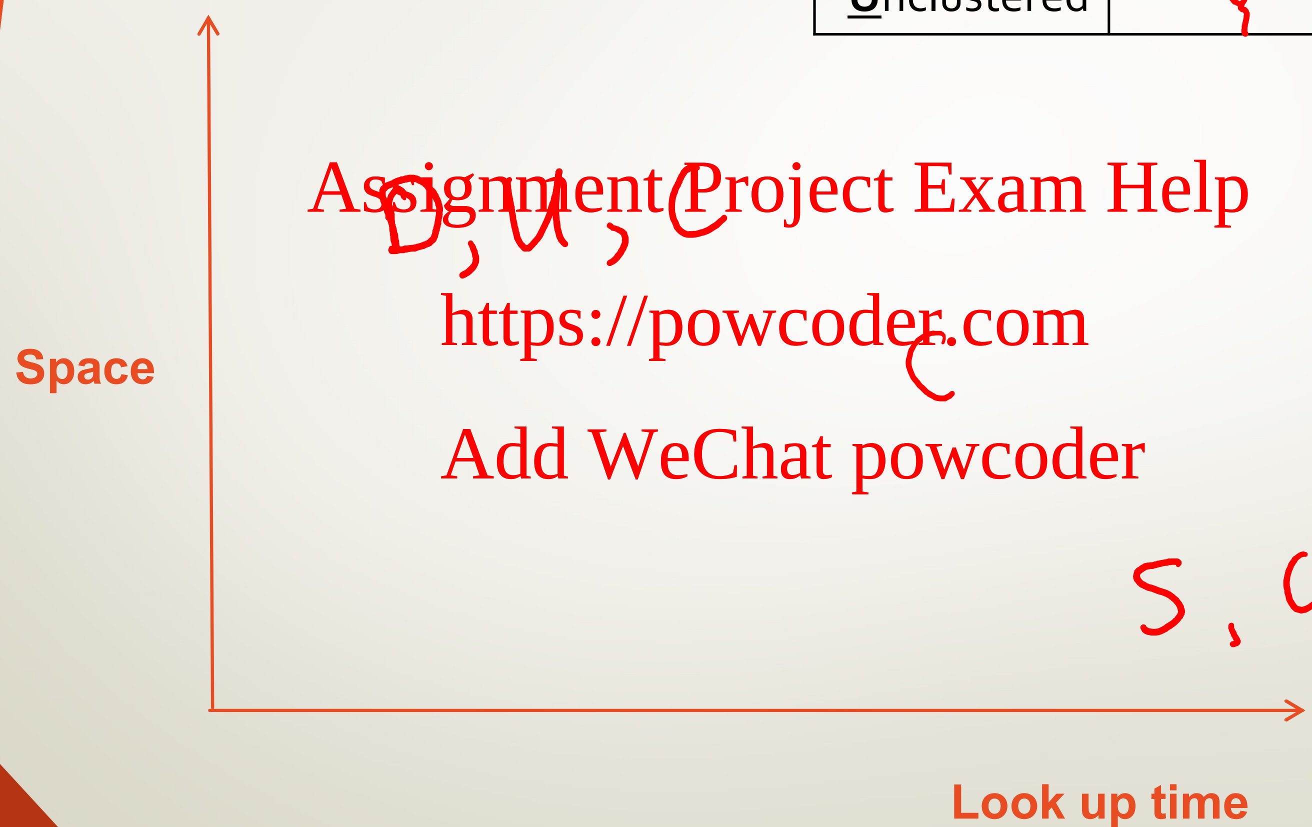
Secondary



Summary Clustered vs. Unclustered Index



	Dense	Sparse
Clustered	Y	Y
Unclustered	Y	N



An Index is a Function!

f(what: key) = where: file block

B+ Trees

B+ Trees

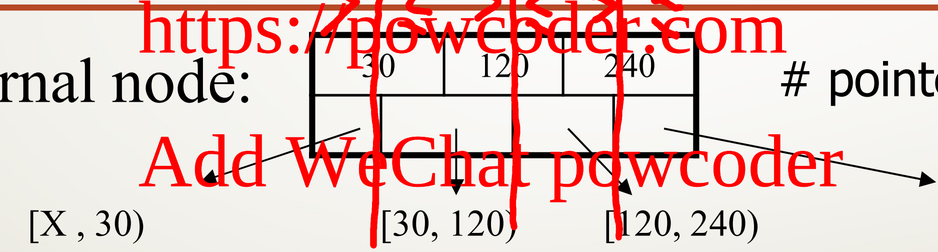
- Intuition:
  - The index can be very large.
  - Index of index?
  - Index of index of index?
  - How best to create such a multi-level index?
- B+ trees:
  - Textbook refers to B+ trees (a popular variant) as B-trees (as most people do)

Focus on the dense version:
applies to clustered and unclustered settings

B+ Trees Basics

- B+ Trees are trees with nodes with keys and pointers to:
  - Other nodes [if the node is an internal node]
  - Data Records [if the node is a leaf]

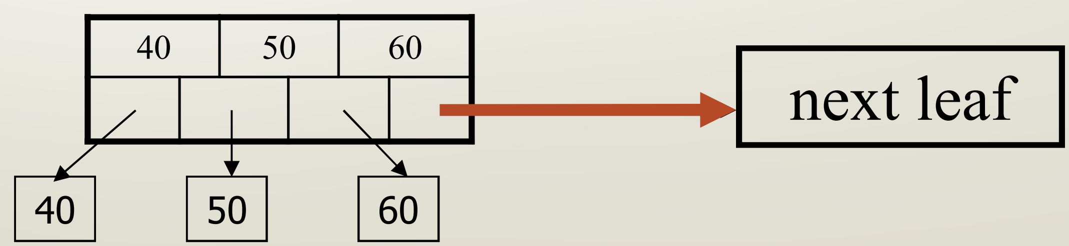
- Internal node:



pointers = # keys + 1

- Leaf:

Data records:



B+ Trees Basics

- Parameter d = the *degree* ; n = max keys
- When n is even [*this is our focus for simplicity*]
  - each node has [d, 2d] keys (except root); $n = 2d$
- At least half full at all times
  - d is the minimum amount it needs to be full.

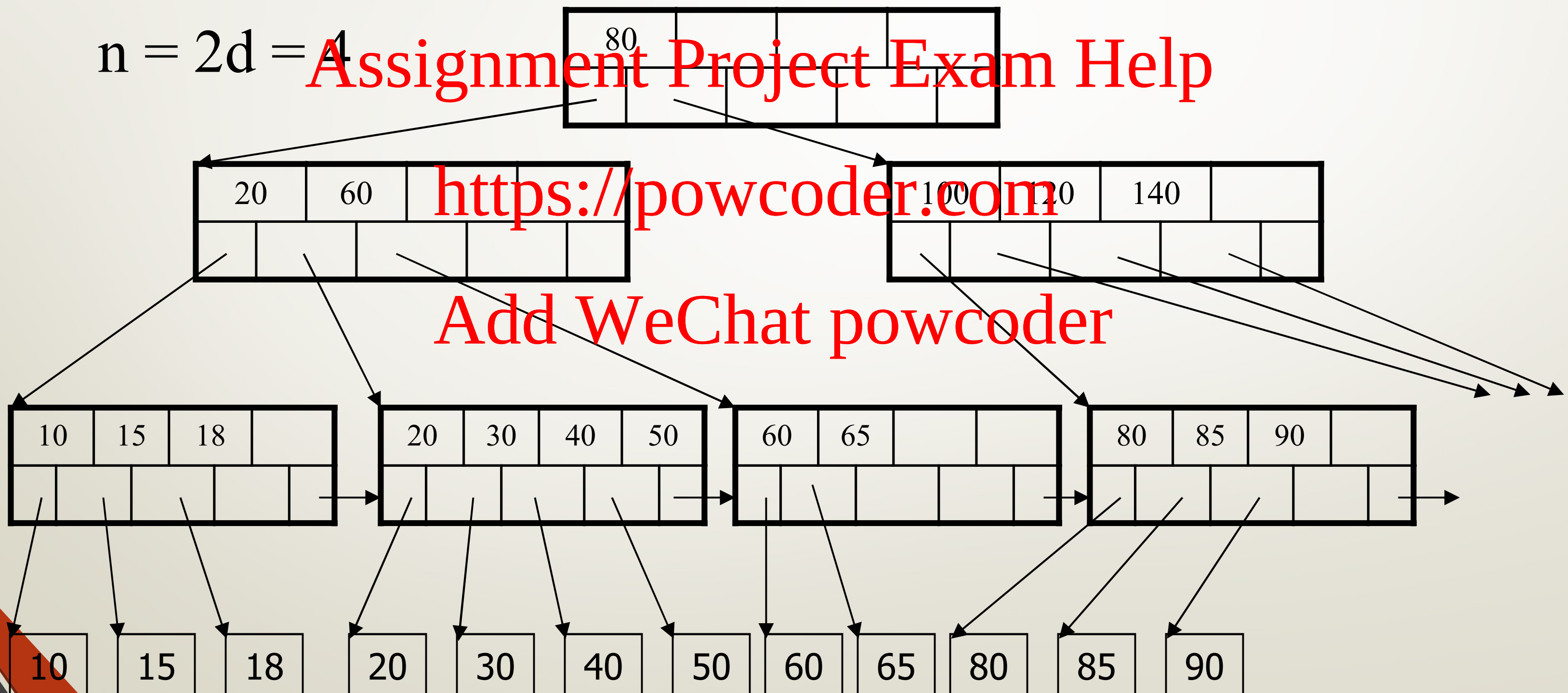
B+ Tree Example

Root can have 1 or more filled in keys

Rest have at least d

d = 2

n = 2d = 4



B+ Tree Design

- How large should *d* be?

4KB

- Example:

  - Key size = 4 bytes
  - Pointer size = 8 bytes

  - Block size = 4096 byes

- $\boxed{2d \times 4} + \boxed{(2d+1) \times 8} <= 4096$
- d = 170; 2d = 340

So up to 340 records in leaf blocks

B+ Trees in Practice

- Typical d: 100. Typical fill-factor: 66.5%.
  - average "fanout" = 66.5 * 2 =133

- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records

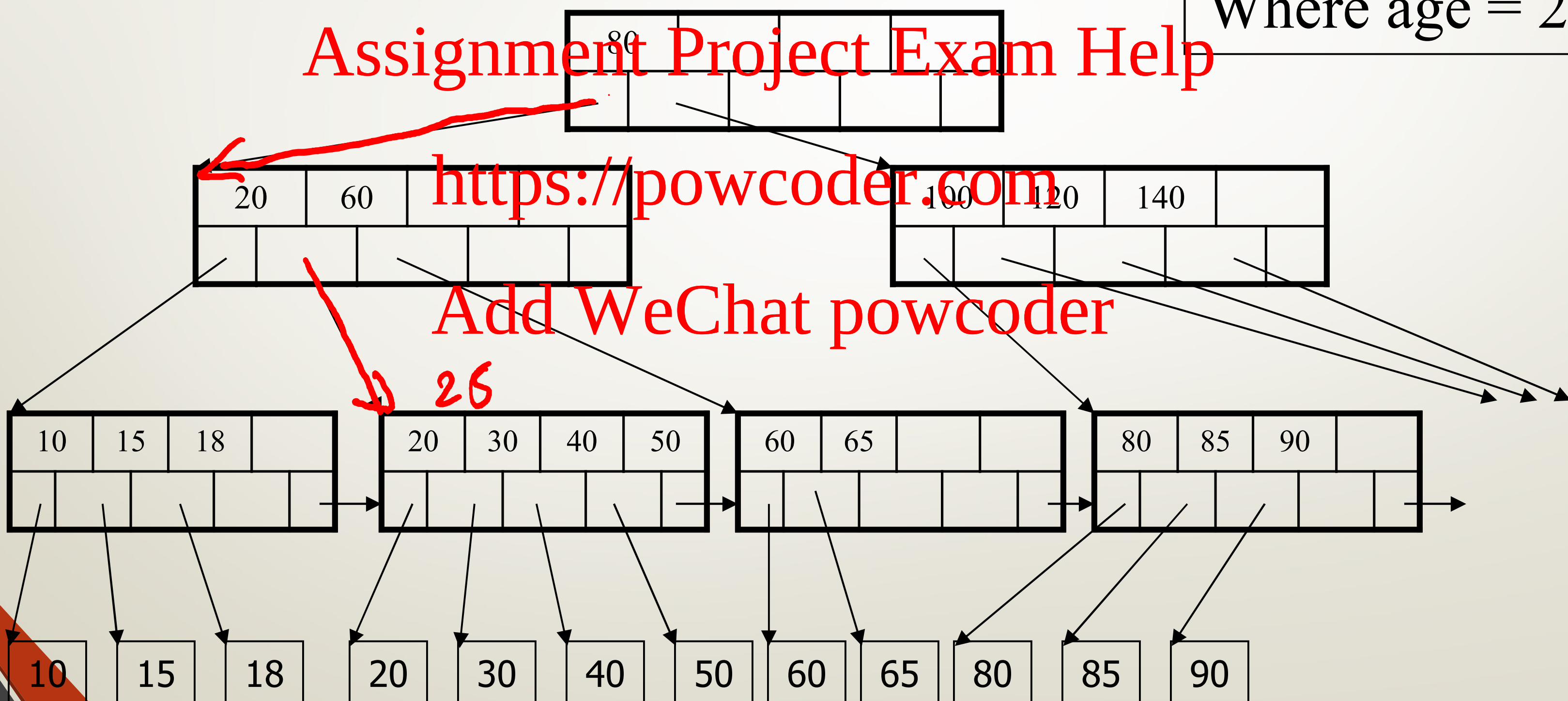
  - Height 3: 133^3 = 2,352,637 records

- Can often hold top levels in main memory:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Searching a B+ Tree

- Exact key values:
  - Start at the root;
  - Proceed down to the leaf

Select name
From people
Where age = 25



Searching a B+ Tree

- Range queries:
  - As above
  - Then sequential traversal using “next leaf” pointers

Select name
From people
Where 20 <= age
and age <= 70