ECS 165B: Database System Implementation
Lecture 3

UC Davis
April 2, 2010

Acknowledgements: design of course project for this class borrowed from CS 346 @ Stanford's RedBase project, developed by Jennifer Widom, and used with permission. Slides based on earlier ones by Raghu Ramakrishnan, Johannes Gehrke, Jennifer Widom, Bertram Ludaescher, and Michael Gertz.

Class Agenda

- Last time:
  - Finish file and buffer management review
  - File and buffer management in DavisDB
- Today:
  - Quick look at DavisDB Record Manager component
  - Start review of indexing
- Reading:
  - Chapter 8 of Ramarkrishnan & Gehkre
  - (or Chapter 12 of Silberschatz et al.)

Announcements

Project Part I has been posted; due Sunday, 4/11 at 11:59pm

http://www.cs.ucdavis.edu/~green/courses/ecs165b/recordManager.html

Please read all documentation carefully, and start early!

Teams have been finalized; still waiting on support@ for subversion repositories

Code distribution available from /home/cs165b/DavisDB

NOTE: some updates made there this afternoon

DavisDB Extra Credit Opportunity

- We've already found and fixed a bug in the page file component since posting the code last night
- There will undoubtedly be more (immature codebase…)
- EXTRA CREDIT OPPORTUNITY: 5% boost to your team's score for Part 1 for each new bug in the page file component you discover and **fix** yourself!

  (Up to 10% boost / team; first team to the bug gets the credit)

  Email your bug reports and fixes to the class mailing list

Quick Tour of Record Manager Component

- Provides classes and methods for managing *files of records* (aka *heap files*)
- Built on top of Page File component, described last time
- You have to implement four main classes, supplied with the code distribution:
  - RecordManager
  - RecordFileHandle
  - Record
  - RecordFileScan
- Don't change any methods in the interface we've given (you can add new methods)
  - Changing the interface will break automated tests

Coding Tip: Don't Forget to Mark Pages Dirty!

- Be diligent about getting this right from the beginning, else you risk introducing tough-to-track-down bugs

```
FileHandle* file;
PageHandle page;

ReturnCode code = file->getFirstPage(&page);
if (code == RC_OK) {
    // … modify contents of page …
    file->markDirty(page.pageNo);
    file->unpinPage(page.pageNo);
}
```

Coding Tip: Assertions are Very Useful

```
#include <assert.h>

void computeSomething() {
    assert(… preconditions …);

    … do some work …

    assert(… postconditions …);
}
```

`assert` will cause a crash if the condition is not satisfied.

This is exactly what you want to happen!

Coding Tip: Assertions in Page File Manager

- Page file manager makes heavy use of runtime assertions; some of these will catch **your** bugs!

```
ReturnCode PageFileManager::allocateBlock(FileHandle*
fileHandle, int pageNo, char** data) {
    // first, look for a free block, while also computing
    // the LRU unpinned block to use as backup
    int iLru = -1;
    long epochLru = LONG_MAX;
    for (uint i = 0; i < PF_BUFFER_SIZE; i++) {
        assert(pageBlocks_[i].isConsistent());
    …
    }
    …
}
```

If this assertion fires, it means your code wrote past the end of a page block!

Coding Tip: Assertions in Page File Manager


```
assert(pageBlocks_[i].isConsistent());
```

`isConsistent`() checks for modification of the *guard bytes* following the page block

Review: Indexing

Reading: Chapter 8 of Ramarkrishnan & Gehkre
(or Chapter 12 of Silberschatz et al.)

Alternative File Organizations

- Many alternatives exist, each ideal for some situations, and not so good for others
- **Unordered heap files** (aka *record files* in DavisDB): 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
- **Indices**: 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

Indices

- 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*

Alternatives for Data Entry *k** in Index

- Three alternatives:
  1. The actual data record with key value *k*
  2. <*k*, id of record with search key value *k*>
  3. <*k*, list of ids of records with search key value *k*>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value *k*
  - Examples of indexing techniques: B+ trees (DavisDB, part 2), hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)

- Alternative 1: the record itself
  - If used, index structure is really 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.

Alternatives for Data Entries (Contd.)

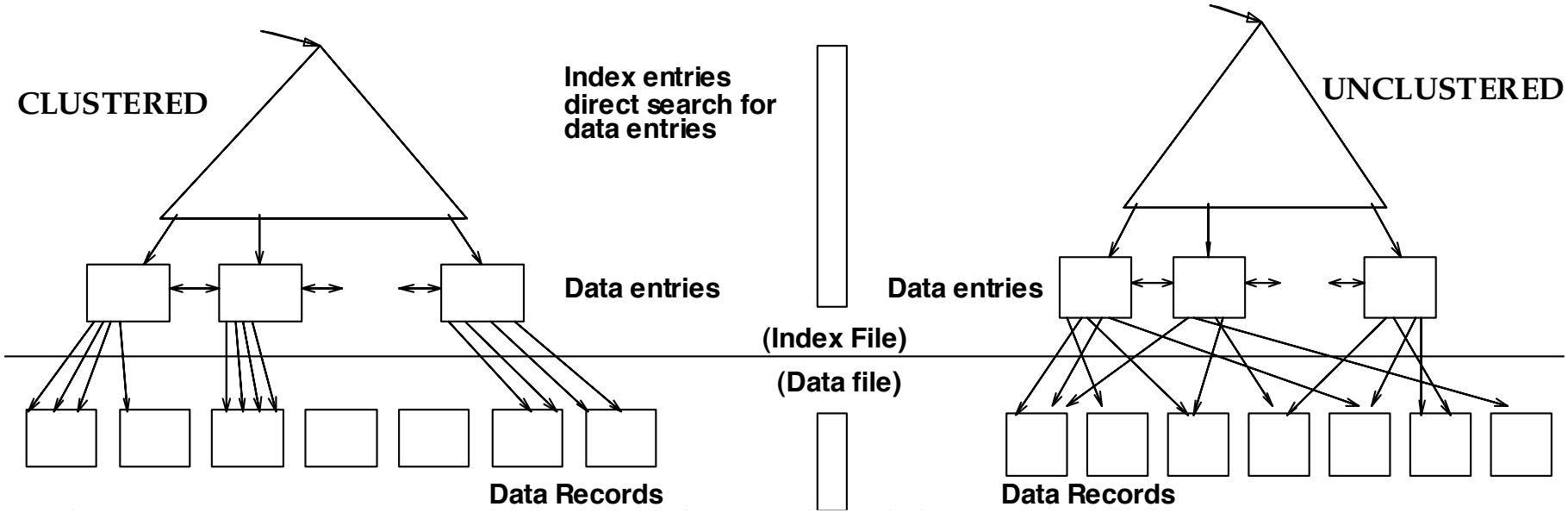
- Alternatives 2 and 3 (record id / list of record ids)
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small.
  - Alternative 3 more compact than Alternative 3, but leads to variable-sized data entries even if search keys are of fixed length

Index Classification

- *Primary* vs. *secondary*: if search key contains primary key, then called *primary index*
- *Clustered* vs. *unclustered*: if order of data records is the same as (or "close to") the order of data entries, then index is called *clustered*
  - Alternative 1 is always a clustered index; in practice, converse usually holds too (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

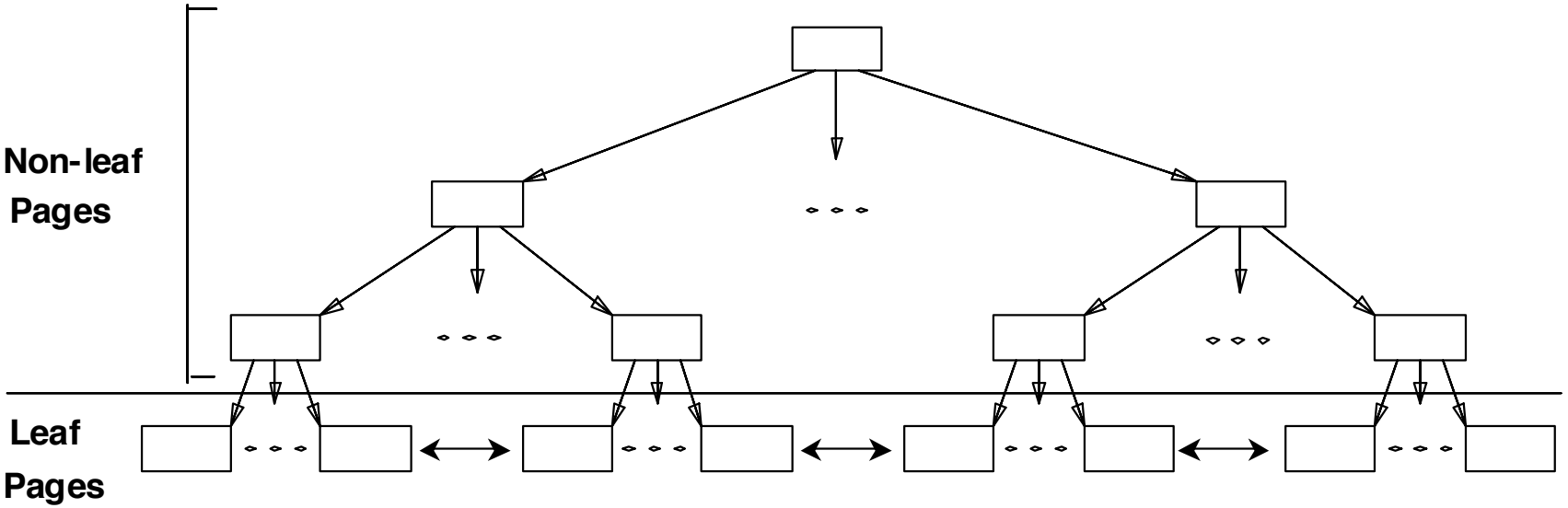
- Suppose Alternative 2 is used for data entries, and that the data records are stored in a heap file
  - To build clustered index, first sort the heap file (with some free space on each page for future insertions)
  - Overflow pages may be needed for insertions. (Thus, order of data records is "close to", but not identical to, the sort order.)



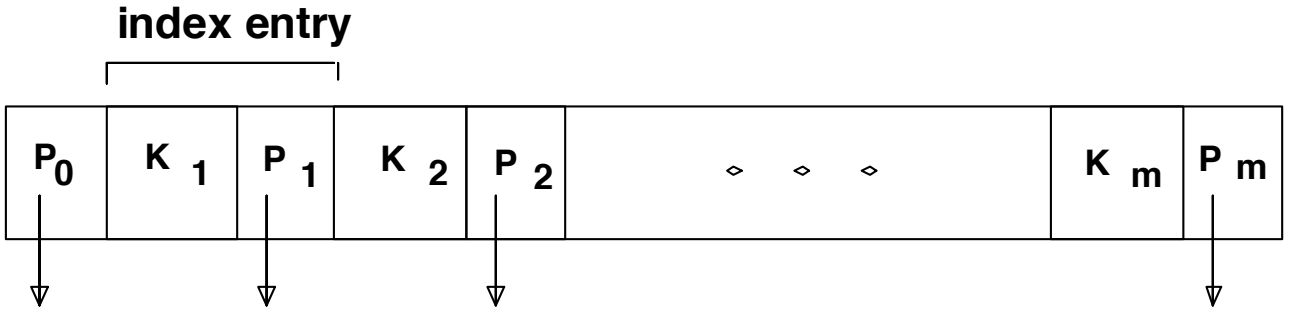
Hash-Based Indices

- Good for equality selections
  - Index is a collection of *buckets*. Bucket = *primary* page plus zero or more *overflow* pages
  - *Hash function h*: *h*(*r*) = bucket in which record *r* belongs. *h* looks at the *search key* fields of *r*.
- If Alternative 1 is used, the buckets contain the data records themselves; otherwise, they contain <key, record id> or <key, record id list> pairs

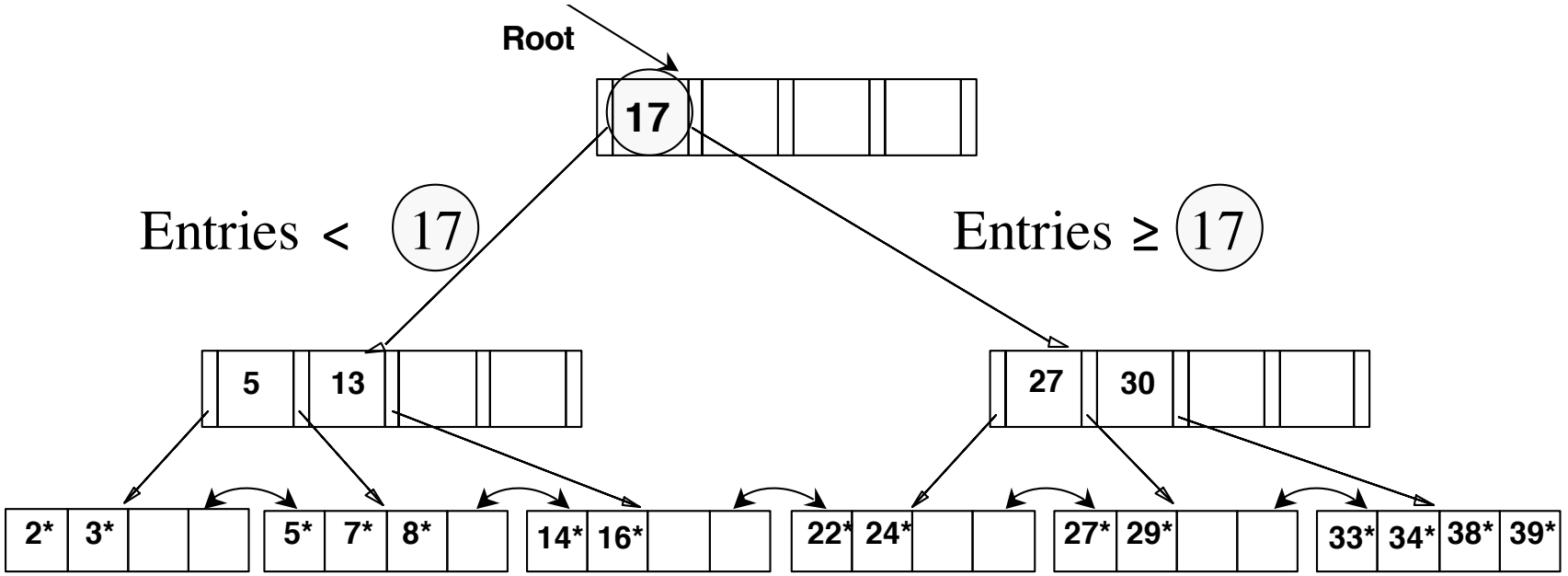
B+ Tree Indices



- Leaf pages contain *data entries*, and are *chained* (prev + next)
- Non-leaf pages contain *index entries* and direct searches



Example of a B+ Tree



- Find 28*? 29*? (> 15 and < 30)*?
- Insert/delete: find data entry in leaf, then change it.
  - Need to adjust parent sometimes
  - Change sometimes bubbles up the tree

Costs and Benefits of Different Schemes

- **Cost model**: ignore CPU costs, for simplicity
  - *B*: number of data pages
  - *R*: number of records per page
  - *D*: (average) time to read or write a disk page
- Measuring number of page I/Os 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
  - But, good enough to show the overall trends!

Comparing File Organizations

- Heap files (random order; insert at end-of-file)
- Sorted files, sorted on <*age*, *sal*>
- Clustered B+ tree file, Alternative 1, 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

Assumptions in Our Analysis

- Heap files:
  - Equality selection on key; exactly one match
- Sorted files:
  - Files compacted after deletions
- Indices:
  - Alternatives 2, 3: data entry size = 10% size of record
  - Hash: no overflow buckets
  - B+ tree: 67% occupancy (this is typical)
    - Implies file size = 1.5 data size

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap					
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog $_2$B	Dlog $_2$ B + # matches	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog $_F$ 1.5B	Dlog $_F$ 1.5B + # matches	Search + D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log $_F$ 0.15B)	Dlog $_F$ 0.15B + # matches	D(3 + log $_F$ 0.15B)	Search + 2D
(5) Unclustered Hash index	BD(R+0.125)	2D	BD	4D	Search + 2D

- *B* = # data pages; *R* = # of records per page; *D* = (average) time to read or write a disk page