CSE 410: Midterm Review

March 1, 2024

Exam Day

- **Do** have...
  - Writing implement (pen or pencil)
  - One note sheet (up to $8\frac{1}{2} \times 11$ inches, double-sided)
- You will **not** need...
  - Computer/Calculator/Watch/etc...

Abstract Disk API

- **Disk** : A collection of **Files**
- **File** : A list of pages, each of size P ($\sim 4K$)
  - `file.read_page(page)`: Get the data on page page of the file.
  - `file.write_page(page, data)`: Write data to page page of the file.

Complexity

```
const RECORDS_PER_PAGE = sizeof::<Record>() / PAGE_SIZE;

fn get_element(file: File, position: u32) -> Record
{
  let page = position / RECORDS_PER_PAGE;
  let data = file.read_page(page);
  return get_records(data)[position % RECORDS_PER_PAGE];
}
```

Complexity

```
fn find_element(file: File, key: u32) -> Record
{
  let mut records: Vec<Record> = Vec::new()
  for page in (0..N)
  {
    let data = file.read_page(idx);
    for record in get_records(data)
    {
      records.push(record);
    }
  }
  return records.binary_search(key)
}
```

Streaming Reads/Writes

```
struct BufferedFile {
  file: File,
  buffer: Page,
  page_idx: u32,
  record_idx: u16,
}
impl BufferedFile {
  fn append(&mut self, record: Record) {
    self.buffer[self.record_idx] = record;
    self.record_idx ++;
    if self.record_idx >= RECORDS_PER_PAGE {
      self.file.write_page(self.page_idx, self.buffer);
      self.record_idx = 0; self.page_idx ++;
    }
  }
}
```

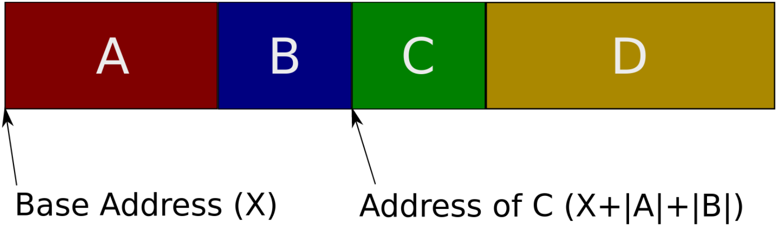
Streaming Reads/Writes

```
struct BufferedFile {
  file: File,
  buffer: Page,
  page_idx: u32,
  record_idx: u16,
}
impl BufferedFile {
  fn next(&mut self) -> Record {
    if self.record_idx >= RECORDS_PER_PAGE {
      self.file.read_page(self.page_idx)
      self.page_idx += 1; self.record_idx = 0
    }
    self.record_idx += 1
    return self.buffer[self.record_idx - 1];
  }
  ...
}
```

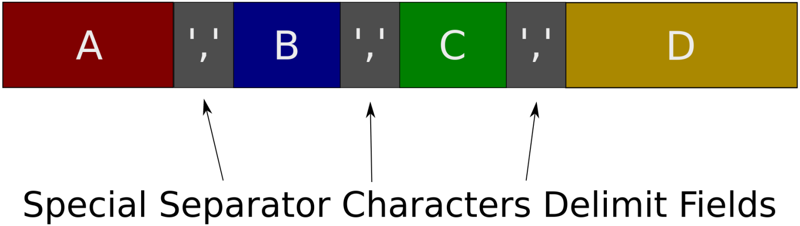
Complexity

```
fn group_by_sum(input: BufferedFile, output: BufferedFile) {
  let mut buffers: Vec<BufferedFile> = Vec::new();
  for _i in (0..B)  { buffers.push(BufferedFile::new()); }
  while !input.done() {
    let record = input.next();
    let i = HASH(record.key) % B;
    buffers[i].append(record)
  }
  for i in (0..B) {
    let local_sums: Map<String,f32> = Map::new()
    buffer[i].reset()
    while !buffer[i].done() {
      let record = buffer[i].next();
      local_sums[record.key] += record.value;
    }
    for key, value in local_sums {
      output.append( Record { key, value } )
} } }
```

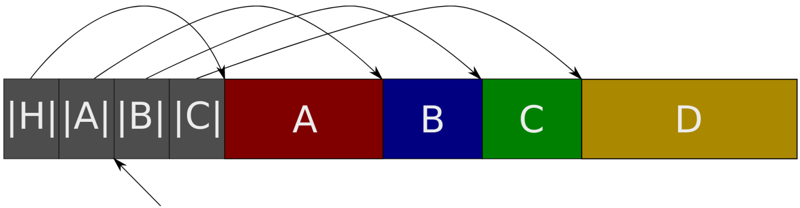
Record Layouts



Record Layouts



Record Layouts

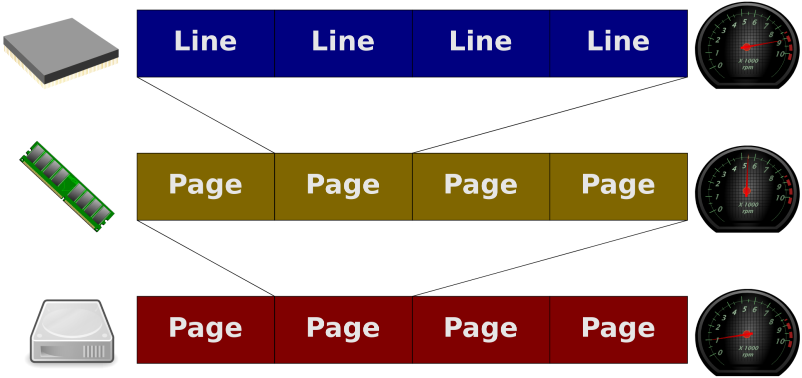


Record header points to each field

Record Layouts

- **Fixed**: Constant-size fields. Field i at byte $\sum_{j<i} |Field_j|$.
- **Delimited**: Special character or string (e.g., ,) between fields.
- **Indexed**: Fixed-size header points to start of each field.

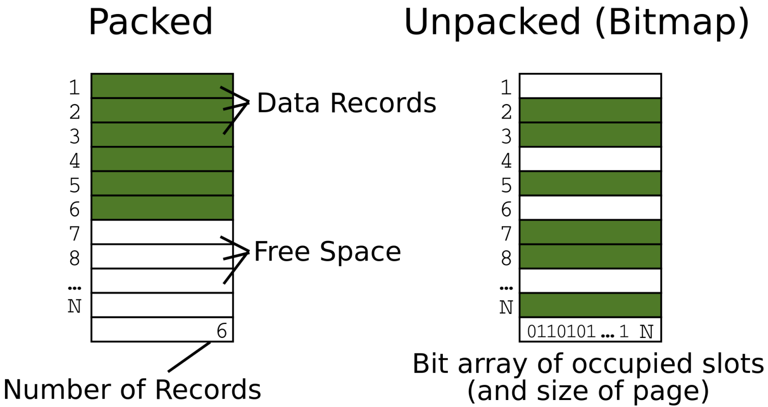
Page Layouts



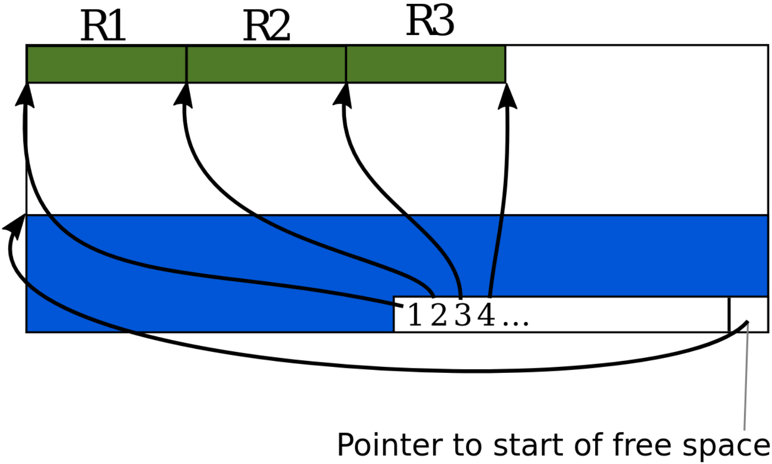
Page Layouts

- **Fixed**: Constant-size records. Record i at byte $i \cdot |Record|$.
- **Delimited**: Special character or string (e.g., `\n`) between records.
- **Indexed**: Fixed-size header points to start of each record.

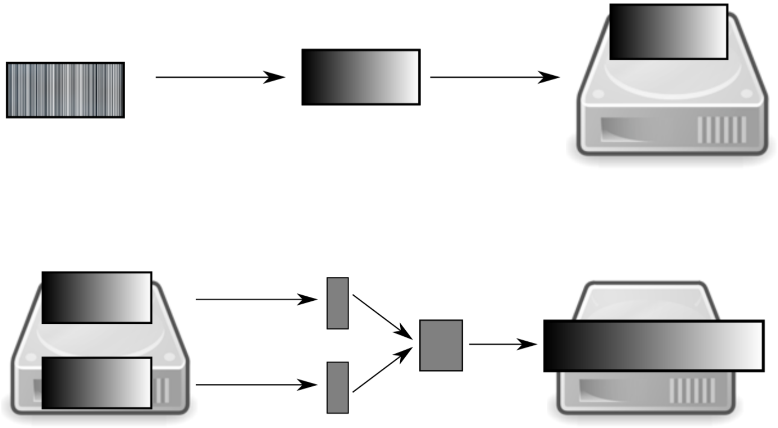
Page Layouts



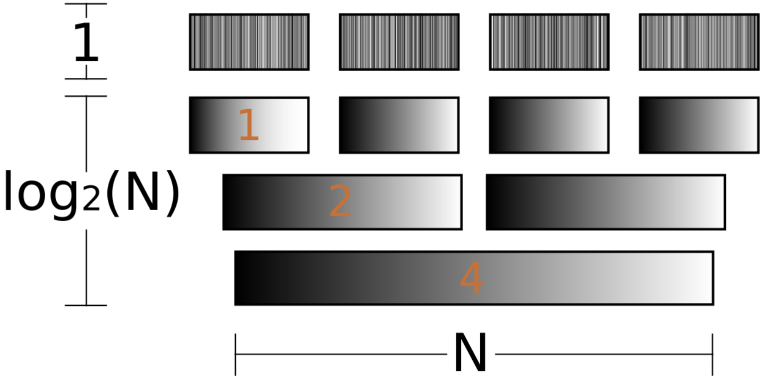
Page Layouts



2-Pass Sort



2-Pass Sort



2-Pass Sort

- **Pass 1**: Use $O(K)$ memory for the initial buffer
- **Pass 2**: Merge $O(K)$ buffers simultaneously

Aggregation

TREE_ID	SPC_COMMON	BORONAME	TREE_DBH

{}

180683	'red maple'	'Queens'	3

{ 'red maple' = 1 }

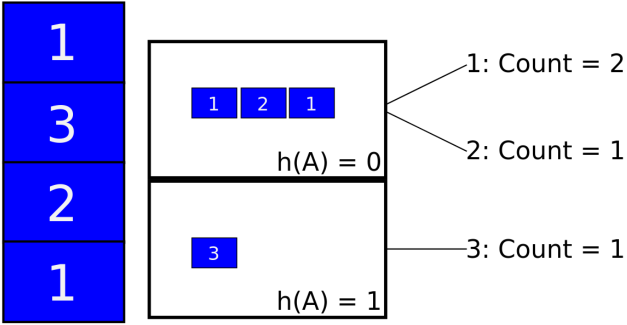
204337	'honeylocust'	'Brooklyn'	10

{ 'red maple' = 1, 'honeylocust' = 1 }

315986	'pin oak'	'Queens'	21

{ 'red maple' = 1, 'honeylocust' = 1, 'pin oak' = 1 }

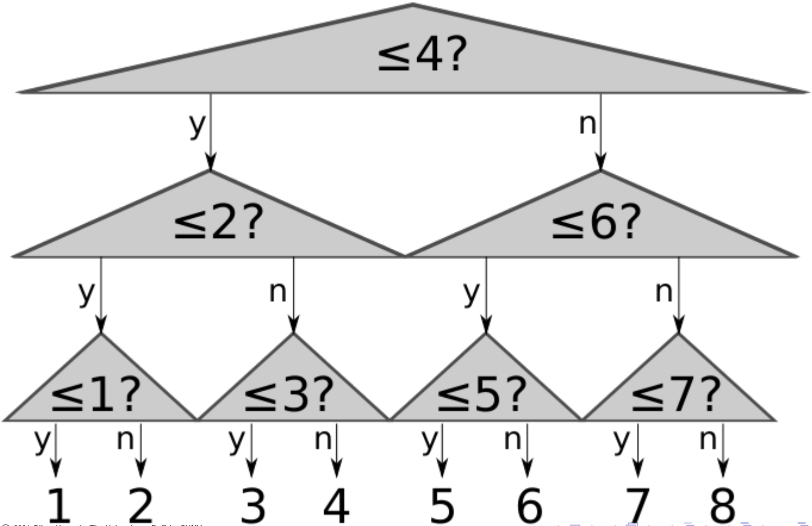
Aggregation



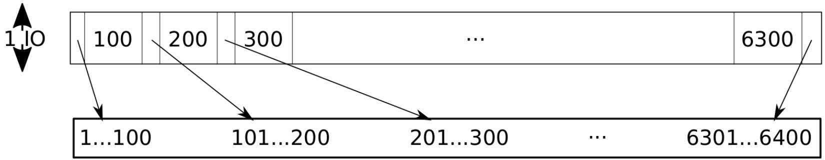
Aggregation

TREE_ID	SPC_COMMON	BORONAME	TREE_DBH
{}			
204337	'honeylocust'	'Brooklyn'	10
{ 'honeylocust' = 1 }			
204026	'honeylocust'	'Brooklyn'	3
{ 'honeylocust' = 2 }			
... and more			
315986	'pin oak'	'Queens'	21
{ 'honeylocust' = 3206, 'pin oak' = 1 }			

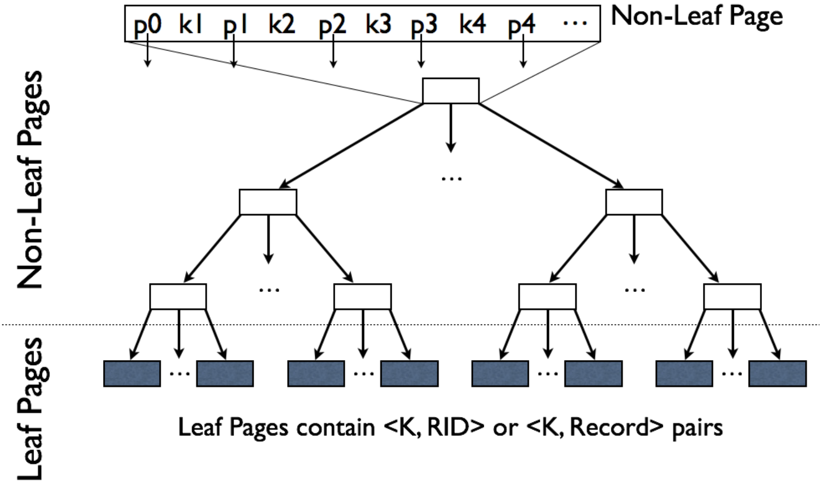
Binary Search



Fence Pointers



ISAM Index


p0 k1 p1 k2 p2 k3 p3 k4 p4 ...
Non-Leaf Page
Non-Leaf Pages
Leaf Pages
Leaf Pages contain <K, RID> or <K, Record> pairs

B+ Tree

Like an ISAM index, but not every page needs to be full, and...

Any page (except the root) must be at least half-full

- Splitting a full page creates a half-full page.
- On deleting the $\frac{P}{2}$th record, steal record from adjacent page.
- If no records can be stolen, must be able to merge with an adjacent page.

B+ Tree

With P records / key+pointer pairs per page:

get(k)

- $O(1)$ Memory complexity
- $O(\log_P(N))$ IO complexity
  - Contrast: $O(\log_2(N))$ in binary search

put(k, v)

- $O(1)$ Memory complexity
- $O(\log_P(N))$ IO complexity
  - $O(\log_P(N))$ reads
  - $O(\log_P(N))$ writes; $O(1)$ amortized writes

LSM Tree

Insight: Updating one record involves many redundant writes in a B+ Tree

Building Block: Sorted Run

- Originally: ISAM Index
- Now: Sorted Array + Fence Pointers (optional Bloom Filter)

LSM Tree

- In-Memory Buffer
- Level 1: B records
- Level 2: $2B$ records
- Level 3: $4B$ records
- Level i: $2^{i+1}B$ records

LSM Tree

put(k,v)

- Append to in-memory buffer.
- If buffer full, sort, and write sorted run to level 1.
- If level 1 already occupied, merge sorted runs and write result to level 2.
- If level 2 already occupied, merge sorted runs and write result to level 3.
- ...
- If level i already occupied, merge sorted runs and write result to level i+1.

LSM Tree

get(k,v)

- Linear scan for k over in-memory buffer.
- If not found, look up k in level 1.
- If not found, look up k in level 2.
- ...

LSM Tree

update(k,v)

- exactly as **put**
- ... but when merging sorted runs, if both input runs contain a key, only keep the newer copy of the record.

delete(k)

- exactly as **update**, but write a 'tombstone' value.
- If **get** encounters a tombstone value, return "not found".
- When merging into lowest level, can delete tombstone.

$\beta - \epsilon$ Trees

Like B+ Tree, but directory pages contain a buffer.

- Writes go to the root page buffer.
- When the root page buffer is full, move its buffered writes to level 2 buffers.
- When a level 2 buffer is full, move its buffered writes to level 3 buffers.
- ...
- When the last directory level buffer is full, apply the writes to the relevant leaves.

Set

- **add(k)**: Updates the set.
- **test(k)**: Returns true iff **add(k)** was called on the set.

Lossy Set

- **add(k)**: Updates the set.
- **test(k)**:
  - Always returns true if **add(k)** was called on the set.
  - Usually returns false if **add(k)** was not called on the set.

Bloom Filters

- A specific implementation of a lossy set.
- $O(N)$ memory to store N keys with a fixed false-positive rate.
  - ... but with a very small constant (1 byte per key $\approx 1 - 2\%$ false positive rate).

Bloom Filters

Before

- Read file
- Find and return record for key

After

- If in-memory bloom filter returns false, return not-found
- Read file
- Find and return record for key