Lecture 30: The IO Model 1 External Sorting

Professor Xiannong Meng
Spring 2018
Lecture and activity contents are based on what Prof Chris Ré of Stanford used in his CS 145 in the fall 2016 term with permission

Today's Lecture

1. The Buffer
2. External Merge Sort

1. The Buffer

Transition to **Mechanisms**

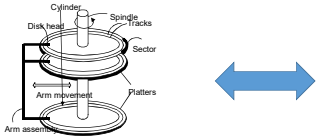
1. So you can **understand** what the database is doing!
   1. Understand the CS challenges of a database and how to use it.
   2. Understand how to optimize a query

2. Many **mechanisms** have become **stand-alone systems**
   - **Indexing** to Key-value stores
   - Embedded join processing
   - SQL-like languages take some aspect of what we discuss (PIG, Hive)

What you will learn about in this section

1. RECAP: Storage and memory model
2. Buffer primer

High-level: Disk vs. Main Memory



Disk:

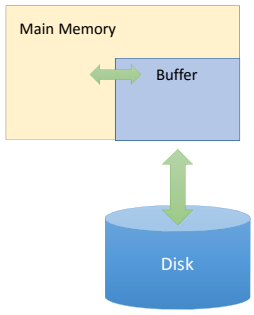
- ***Slow:*** Sequential *block* access
  - Read a blocks (not byte) at a time, so sequential access is cheaper than random
  - **Disk read / writes are expensive!**
- ***Durable:*** We will assume that once on disk, data is safe!
- ***Cheap***

Random Access Memory (RAM) or Main Memory:

- ***Fast:*** Random access, byte addressable
  - ~10x faster for sequential access
  - ~100,000x faster for random access!
- ***Volatile:*** Data can be lost if e.g. crash occurs, power goes out, etc!
- ***Expensive:*** For $100, get 16GB of RAM vs. 2TB of disk!

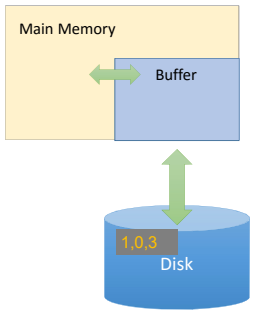
The Buffer

- A **buffer** is a region of physical memory used to store *temporary data*
  - *In this lecture:* a region in main memory used to store **intermediate data between disk and processes**
- *Key idea:* Reading / writing to disk is slow- need to cache data!



The (Simplified) Buffer

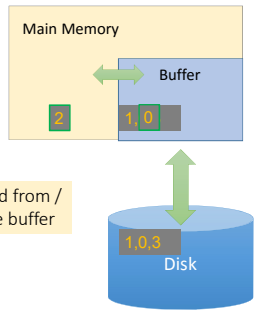
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*



The (Simplified) Buffer

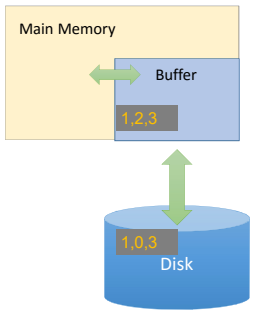
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*

Processes can then read from / write to the page in the buffer



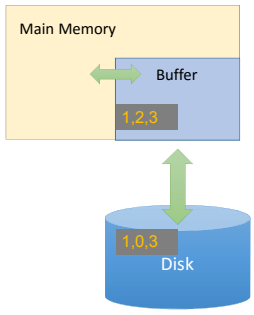
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*
  - **Flush(page):** Evict page from buffer & write to disk



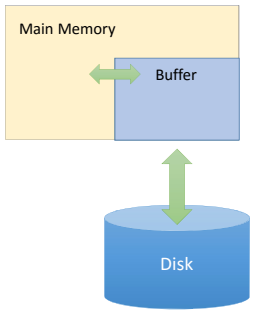
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*
  - **Flush(page):** Evict page from buffer & write to disk
  - **Release(page):** Evict page from buffer *without* writing to disk



Managing Disk: The DBMS Buffer

- Database maintains its own buffer
  - Why? The OS already does this…
  - DB knows more about access patterns.
    - Watch for how this shows up! (cf. *Sequential Flooding*)
- Recovery and logging require ability to **flush** to disk.

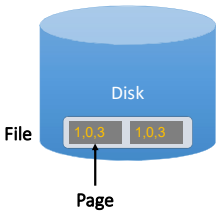


The Buffer Manager

- A **buffer manager** handles supporting operations for the buffer:
  - Primarily, handles & executes the "replacement policy"
    - i.e. finds a page in buffer to flush/release if buffer is full and a new page needs to be read in
  - DBMSs typically implement their own buffer management routines

A Simplified Filesystem Model

- For us, a **page** is a ***fixed-sized array*** of memory
  - Think: One or more disk blocks
  - Interface:
    - write to an entry (called a **slot**) or set to "None"
  - DBMS also needs to handle variable length fields
    - Page layout is important for good hardware utilization as well (see 346)
- And a **file** is a *variable-length list* of pages
  - Interface: create / open / close; next_page(); etc.



2. External Merge & Sort

15

What you will learn about in this section

1. External Merge- Basics
2. External Merge- Extensions
3. External Sort

16

External Merge

Challenge: Merging Big Files with Small Memory

How do we *efficiently* merge two sorted files when both are much larger than our main memory buffer?

External Merge Algorithm

- **Input**: 2 sorted lists of length M and N
- **Output:** 1 sorted list of length M + N
- **Required:** At least 3 Buffer Pages
- **IOs**: 2(M+N)

Key (Simple) Idea

To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

If:

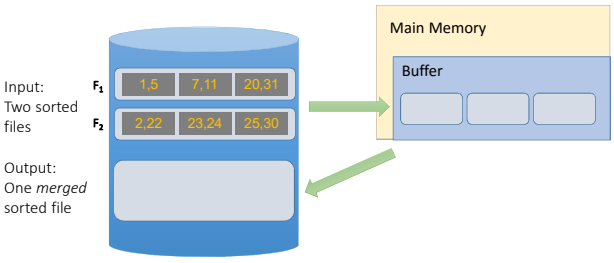
$$A_1 \leq A_2 \leq \cdots \leq A_N$$
$$B_1 \leq B_2 \leq \cdots \leq B_M$$

Then:

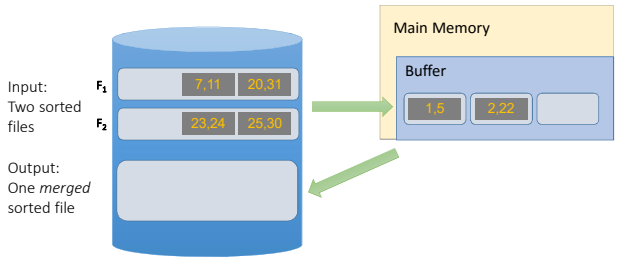
$$Min(A_1, B_1) \leq A_i$$
$$Min(A_1, B_1) \leq B_j$$

for i=1….N and j=1….M

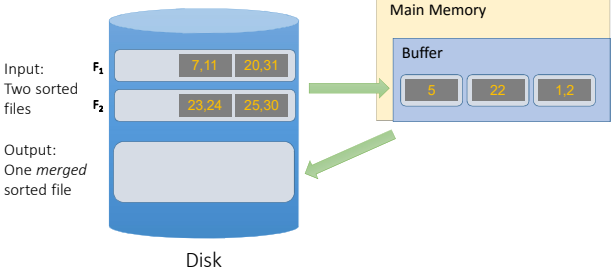
External Merge Algorithm



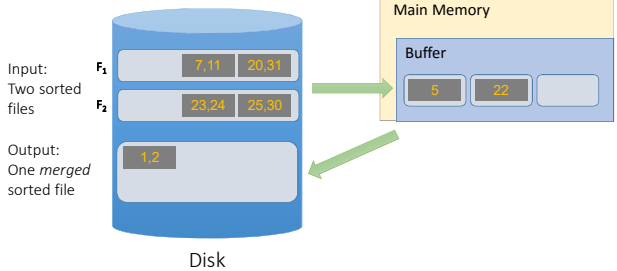
External Merge Algorithm



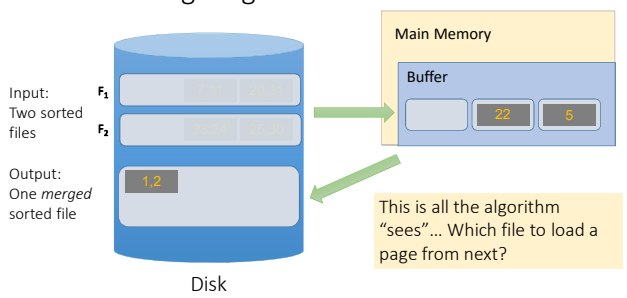
External Merge Algorithm



External Merge Algorithm

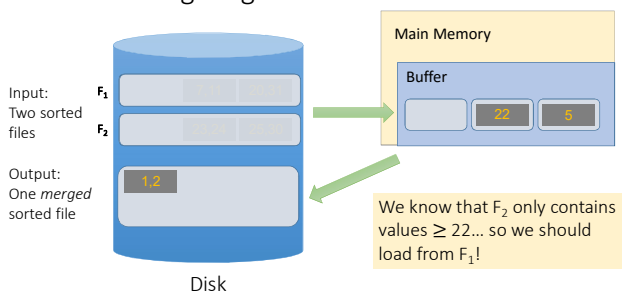


External Merge Algorithm



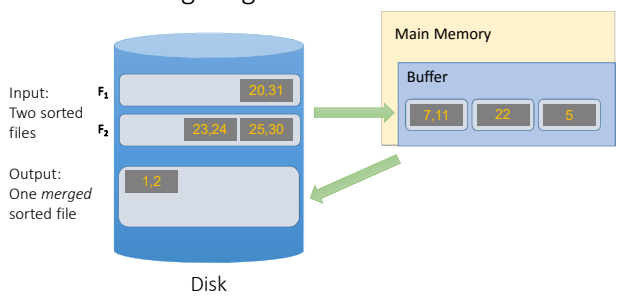
This is all the algorithm "sees"… Which file to load a page from next?

External Merge Algorithm

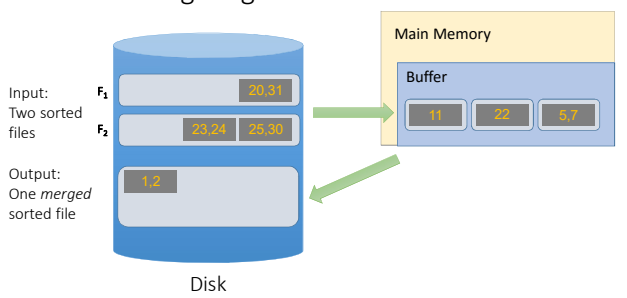


We know that F_2 only contains values ≥ 22… so we should load from F_1!

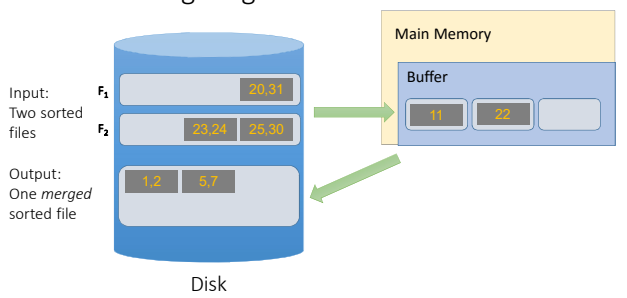
External Merge Algorithm



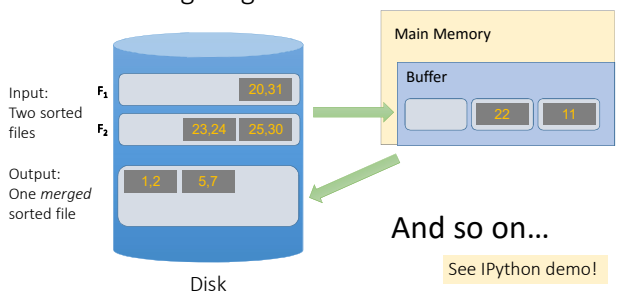
External Merge Algorithm



External Merge Algorithm



External Merge Algorithm



And so on…

See IPython demo!

We can merge lists of **arbitrary length** with *only* 3 buffer pages.

If lists of size M and N, then
Cost: 2(M+N) IOs
Each page is read once, written once

With B+1 buffer pages, can merge B lists. How?

Today's Lecture

1. *External Merge Sort & Sorting Optimizations*
2. Indexes: Motivations & Basics

32

1. External Merge Sort

33

What you will learn about in this section

1. External merge sort
2. External merge sort on larger files
3. Optimizations for sorting

34

Recap: External Merge Algorithm

- Suppose we want to merge two **sorted** files both much larger than main memory (i.e. the buffer)
- We can use the **external merge algorithm** to merge files of ***arbitrary length*** in **2*(N+M) IO** operations with only **3 buffer pages**!

Our first example of an "IO aware" algorithm / cost model

External Merge Sort

Why are Sorting Algorithms Important?

- Data requested from DB in sorted order is **extremely common**
  - e.g., find students in increasing GPA order
- **Why not just use quicksort in main memory??**
  - What about if we need to sort 1TB of data with 1GB of RAM…

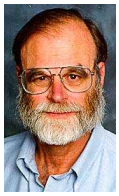
A classic problem in computer science!

More reasons to sort…

- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
- Sorting is first step in *bulk loading* B+ tree index. — *Coming up…*
- *Sort-merge* join algorithm involves sorting — *Next lecture*

Do people care?

http://sortbenchmark.org

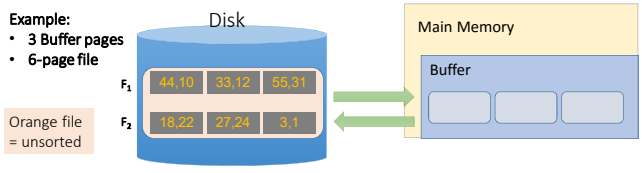


Sort benchmark bears his name

So how do we sort big files?

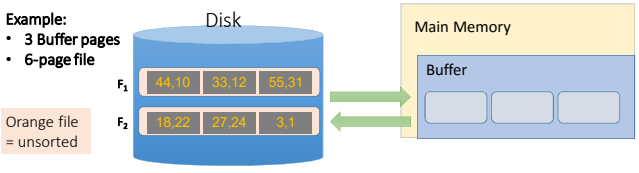
1. Split into chunks small enough to **sort in memory** ***("runs")***
2. **Merge** pairs (or groups) of runs ***using the external merge algorithm***
3. **Keep merging** the resulting runs ***(each time = a "pass")*** until left with one sorted file!

External Merge Sort Algorithm



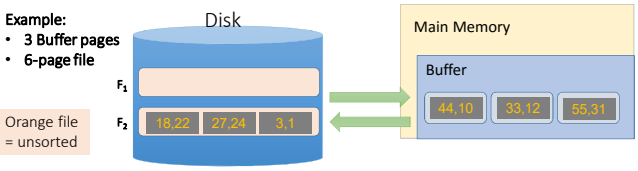
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



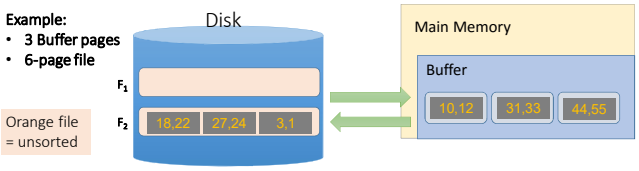
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



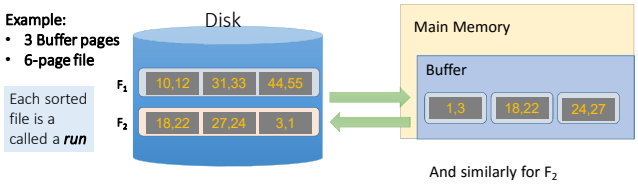
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



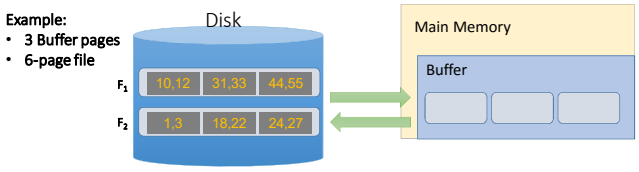
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



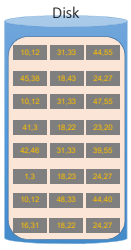
2. Now just run the **external merge** algorithm & we're done!

Calculating IO Cost

For 3 buffer pages, 6 page file:

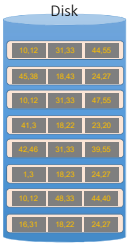
1. Split into **<u>two 3-page files</u>** and **sort in memory**
   1. **= 1 R + 1 W for each file = 2*(3 + 3) = 12 IO operations**
2. **Merge** each pair of sorted chunks ***using the external merge algorithm***
   1. **= 2*(3 + 3) = 12 IO operations**
3. **Total cost = 24 IO**

Running External Merge Sort on Larger Files



Assume we still only have 3 buffer pages *(Buffer not pictured)*

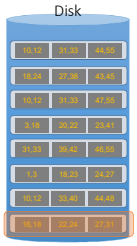
Running External Merge Sort on Larger Files



1. Split into files small enough to sort in buffer…

Assume we still only have *3* buffer pages *(Buffer not pictured)*

Running External Merge Sort on Larger Files

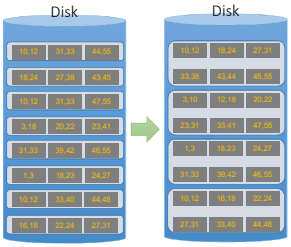


1. Split into files small enough to sort in buffer… and sort

Assume we still only have *3* buffer pages *(Buffer not pictured)*

Call each of these sorted files a ***run***

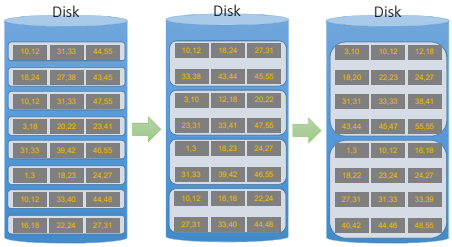
Running External Merge Sort on Larger Files



Assume we still only have *3* buffer pages *(Buffer not pictured)*

2. Now merge pairs of (sorted) files… **the resulting files will be sorted!**

Running External Merge Sort on Larger Files

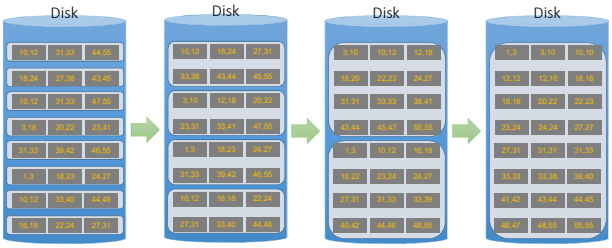


Assume we still only have *3* buffer pages *(Buffer not pictured)*

3. And repeat…

Call each of these steps a ***pass***

Running External Merge Sort on Larger Files

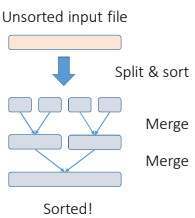


4. And repeat!

Simplified 3-page Buffer Version

Assume for simplicity that we split an N-page file into N single-page ***runs*** and sort these; then:

- First pass: Merge **N/2 *pairs* of runs** each of length **1 page**
- Second pass: Merge **N/4 *pairs* of runs** each of length **2 pages**
- In general, for **N** pages, we do $\lceil log_2 N \rceil$ passes
  - +1 for the initial split & sort
- Each pass involves reading in & writing out all the pages = ***2N IO***



→ **2N*(**$\lceil log_2 N \rceil$**+1)** total IO cost!

Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

1. Increase length of initial runs. Sort B+1 at a time!

At the beginning, we can split the N pages into runs of length B+1 and sort these in memory

IO Cost:

$2N(\lceil \log_2 N \rceil + 1)$ ➡ $2N(\lceil \log_2 \frac{N}{B+1} \rceil + 1)$

Starting with runs of length 1

Starting with runs of length ***B+1***

Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

2. Perform a B-way merge.

On each pass, we can merge groups of ***B*** runs at a time (vs. merging pairs of runs)!

IO Cost:

$2N(\lceil \log_2 N \rceil + 1)$ ➡ $2N(\lceil \log_2 \frac{N}{B+1} \rceil + 1)$ ➡ $2N(\lceil \log_B \frac{N}{B+1} \rceil + 1)$

Starting with runs of length 1

Starting with runs of length ***B+1***

Performing ***B***-way merges

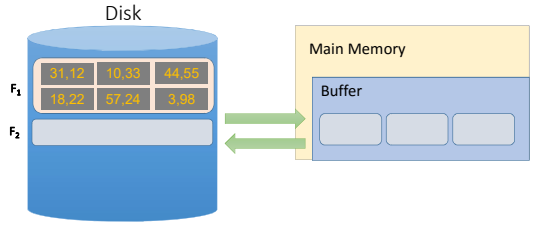
Repacking

Repacking for even longer initial runs

- With B+1 buffer pages, we can now start with ***B+1-length initial runs*** (and use ***B-way merges***) to get $2N(\lceil \log_B \frac{N}{B+1} \rceil + 1)$ IO cost…
- Can we reduce this cost more by getting even longer initial runs?
- Use **repacking**- produce longer initial runs by "merging" in buffer as we sort at initial stage

Repacking Example: 3 page buffer

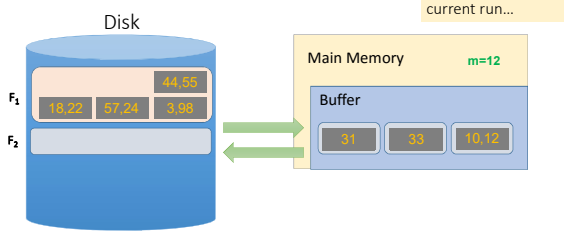
- Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

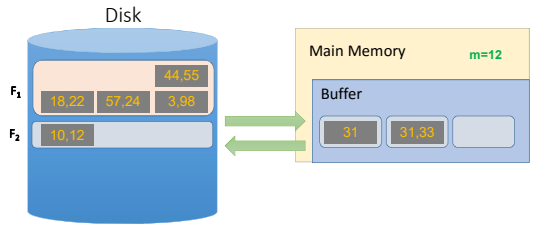
- Take the minimum two values, and put in output page

Also keep track of max (last) value in current run…



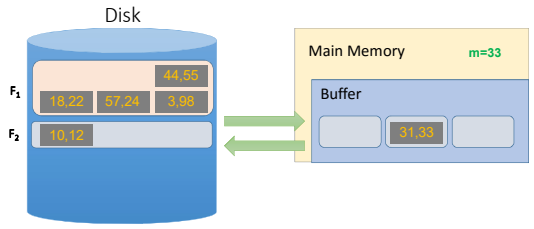
Repacking Example: 3 page buffer

- Next, ***repack***



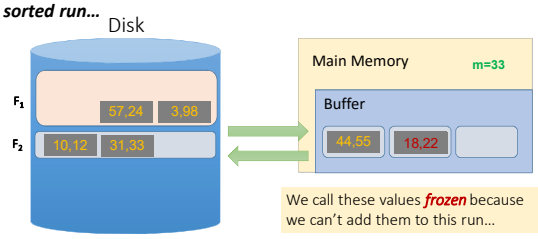
Repacking Example: 3 page buffer

- Next, ***repack***, then load another page and continue!



Repacking Example: 3 page buffer

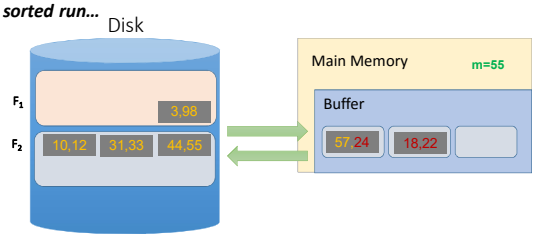
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



We call these values ***frozen*** because we can't add them to this run…

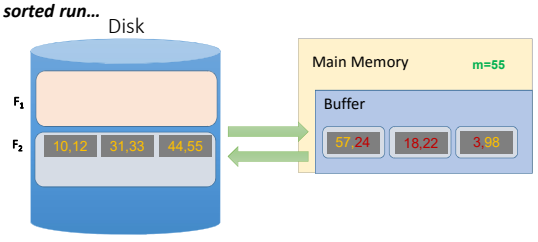
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



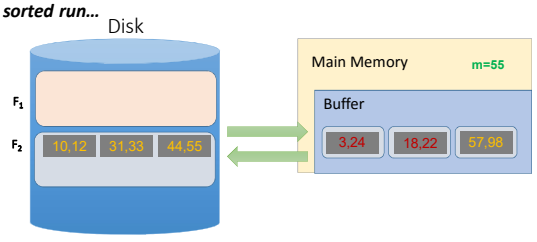
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



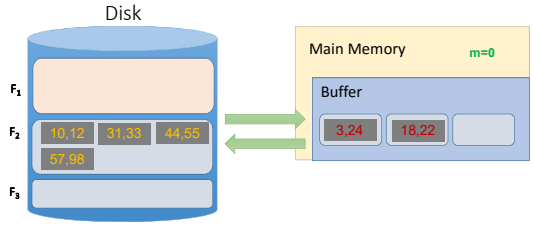
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



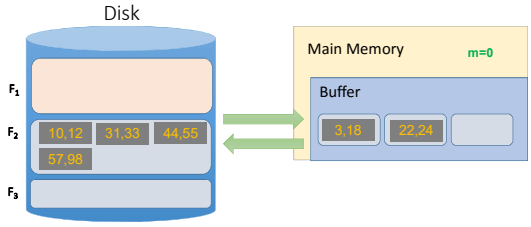
Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value,*** or input file is empty, start new run with the frozen values



Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value,*** or input file is empty, start new run with the frozen values



Repacking

- Note that, for buffer with B+1 pages:
  - If input file is sorted → nothing is frozen → we get **a single** run!
  - If input file is reverse sorted (worst case) → everything is frozen → we get runs of length **B+1**
- In general, with repacking we do **no worse** than without it!
- What if the file is already sorted?
- Engineer's approximation: runs will have **~2(B+1)** length

$$\sim 2N\left(\left\lceil \log_B \frac{N}{2(B+1)} \right\rceil + 1\right)$$

Summary

- Basics of IO and buffer management.
  - See notebook for more fun! (Learn about *sequential flooding*)
- We introduced the IO cost model using **sorting**.
  - Saw how to do merges with few IOs,
  - Works better than main-memory sort algorithms.
- Described a few optimizations for sorting

What you will learn about in this section

1. Indexes: Motivation
2. Indexes: Basics

71

Index Motivation

Person(name, age)

- Suppose we want to search for people of a specific age
- ***First idea:*** Sort the records by age… we know how to do this fast!
- How many IO operations to search over ***N sorted*** records?
  - Simple scan: ***O(N)***
  - Binary search: ***O($\log_2 N$)***

Could we get even cheaper search? E.g. go from $\log_2 N$ → $\log_{200} N$?

Index Motivation

- What about if we want to **insert** a new person, but keep the list sorted?

- We would have to potentially shift ***N*** records, requiring up to **~ 2*N/P** IO operations (where P = # of records per page)!
  - We could leave some "slack" in the pages…

Could we get faster insertions?

Index Motivation

- What about if we want to be able to search quickly along multiple attributes (e.g. not just age)?
  - We could keep multiple copies of the records, each sorted by one attribute set… this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

We'll create separate data structures called ***indexes*** to address all these points

Further Motivation for Indexes: NoSQL!

- NoSQL engines are (basically) ***just indexes!***
  - A lot more is left to the user in NoSQL… one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!
  - Sometimes use B+ Trees (covered next), sometimes hash indexes (not covered here)

Indexes are critical across all DBMS types

Indexes: High-level

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Search key properties
    - Any subset of fields
    - is **not** the same as *key of a relation*
- *Example:*

Product(name, maker, price)

On which attributes would you build indexes?

More precisely

- An *index* is a **data structure** mapping search keys to sets of rows in a database table
  - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table
- An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)
  - We'll mainly consider secondary indexes

Operations on an Index

- Search: Quickly find all records which meet some *condition on the search key attributes*
  - More sophisticated variants as well. Why?
- Insert / Remove entries
  - Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance

Conceptual Example

What if we want to return all books published after 1867? The above table might be very expensive to search over row-by-row…

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

```
SELECT *
FROM Russian_Novels
WHERE Published > 1867
```

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	002
Tolstoy	Anna Karenina	003
Tolstoy	War and Peace	001

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

Covering Indexes

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

We say that an index is **covering** *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

The "needed" attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```

High-level Categories of Index Types

- B-Trees *(covered next)*
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called* ***B+ Trees***
- Hash Tables *(not covered)*
  - There are variants of this basic structure to deal with IO
  - Called ***linear*** or ***extendible hashing***- IO aware!

The data structures we present here are "IO aware"

Real difference between structures: costs of ops *determines which index you pick and why*