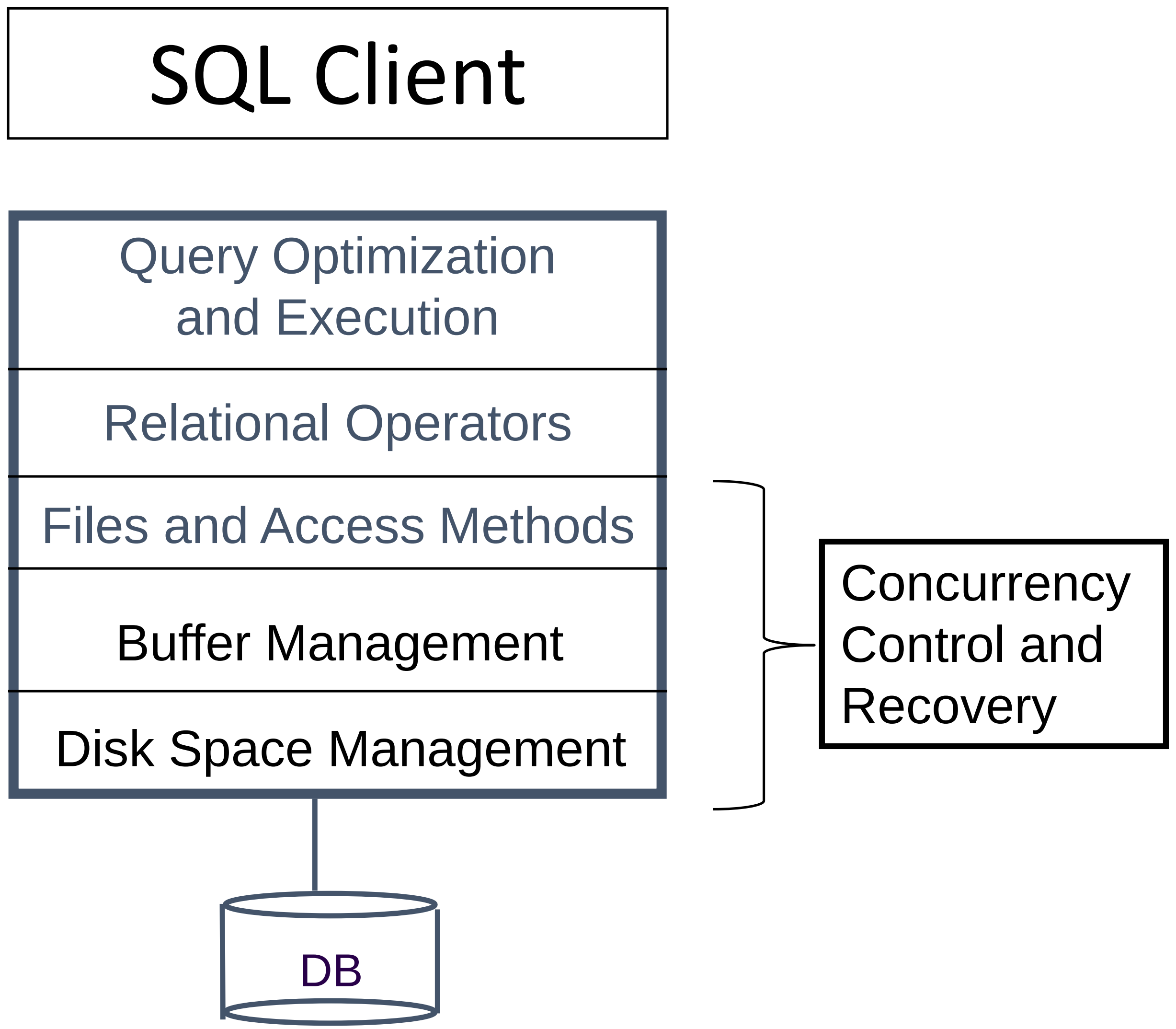
CS150: Database & Datamining
Lecture 9: The External Sorting & Files

ShanghaiTech-SIST

Spring 2019

Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis.

Block diagram of a DBMS



Today's Lecture

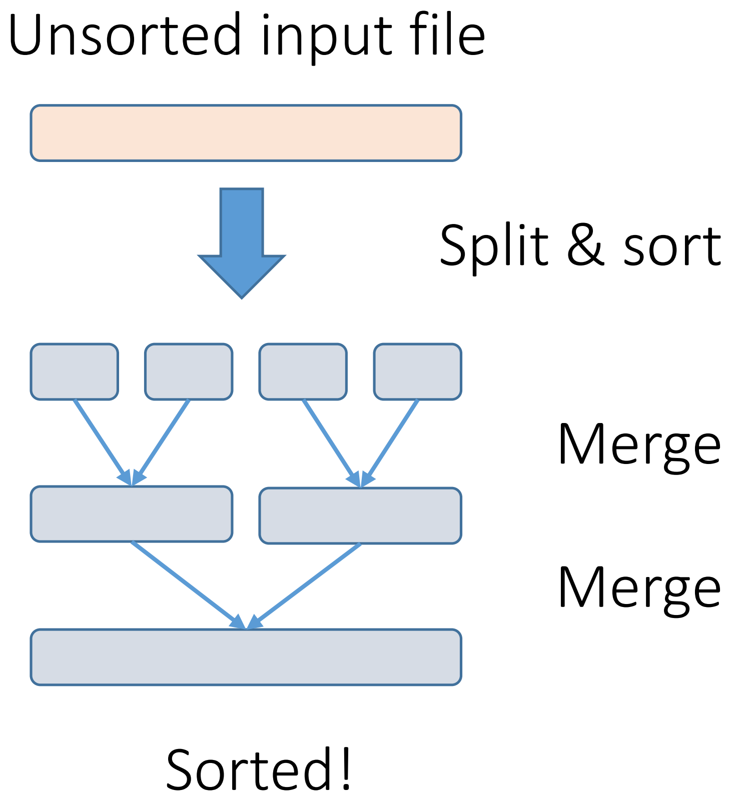
1. External Merge Sort
2. File Organizations

1. External Merge Sort

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 $\lceil \boldsymbol{log_2 N} \rceil$ passes
  - +1 for the initial split & sort
- Each pass involves reading in & writing out all the pages = ***2N IO***



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

External Merge Sort: Optimizations

Now assume we have **B+1 buffer pages**; three optimizations:

1. Increase the length of initial runs
2. B-way merges
3. Repacking

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

<u>IO Cost:</u>

$$2N(\lceil \log_2 N \rceil + 1) \rightarrow 2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

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:

$$2N(\lceil \log_2 N \rceil + 1) \rightarrow 2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right) \rightarrow 2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Starting with runs of length 1

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

Performing ***B***-way merges

Repacking for even longer initial runs

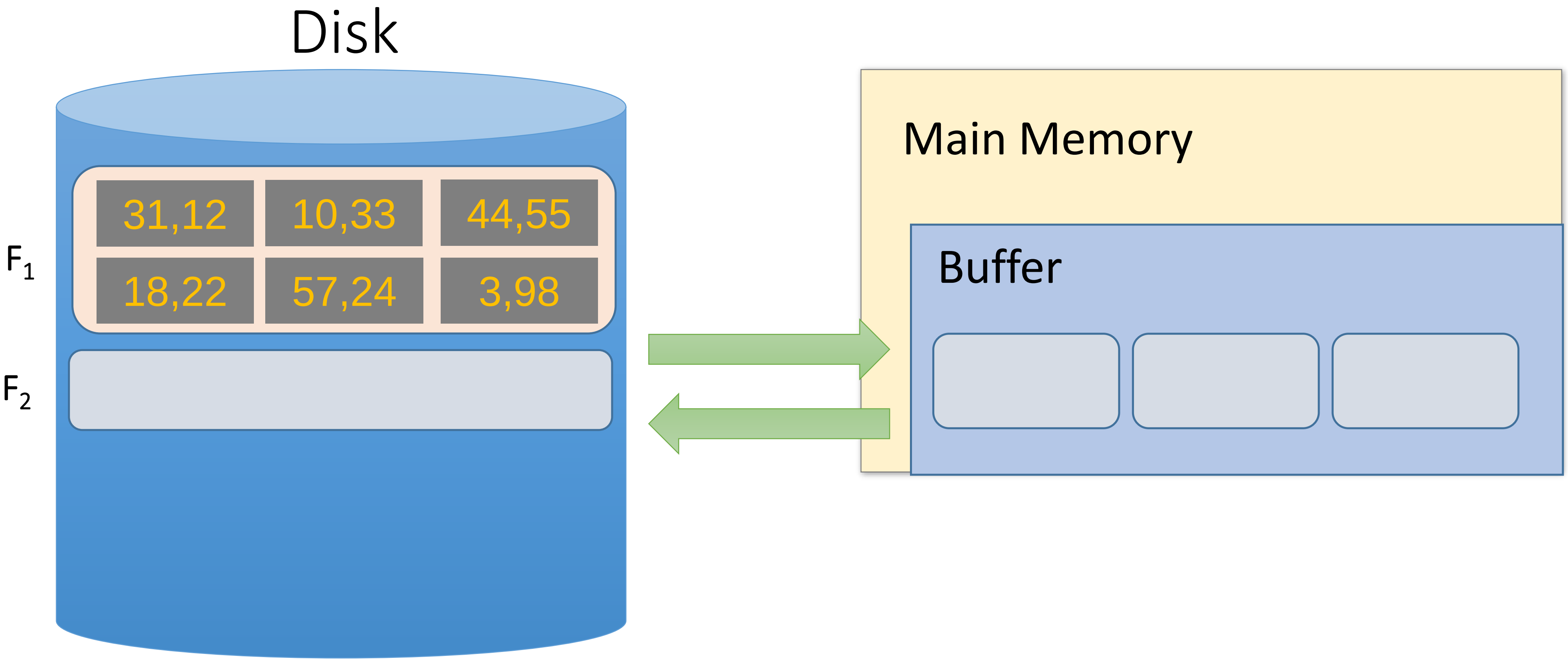
- With B+1 buffer pages, we can now start with ***B+1-length initial runs*** (and use ***B-way merges***) to get $2N(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1)$ IO cost…

- 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

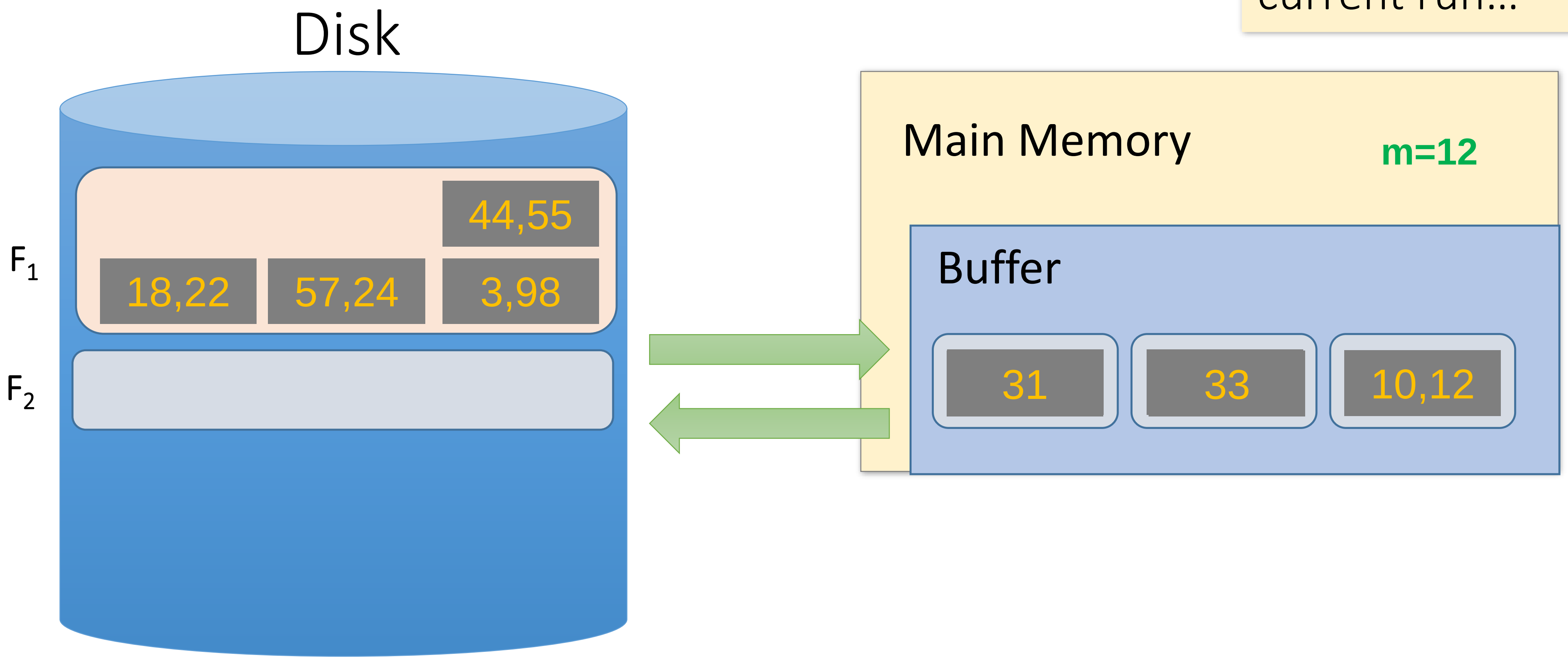
- Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

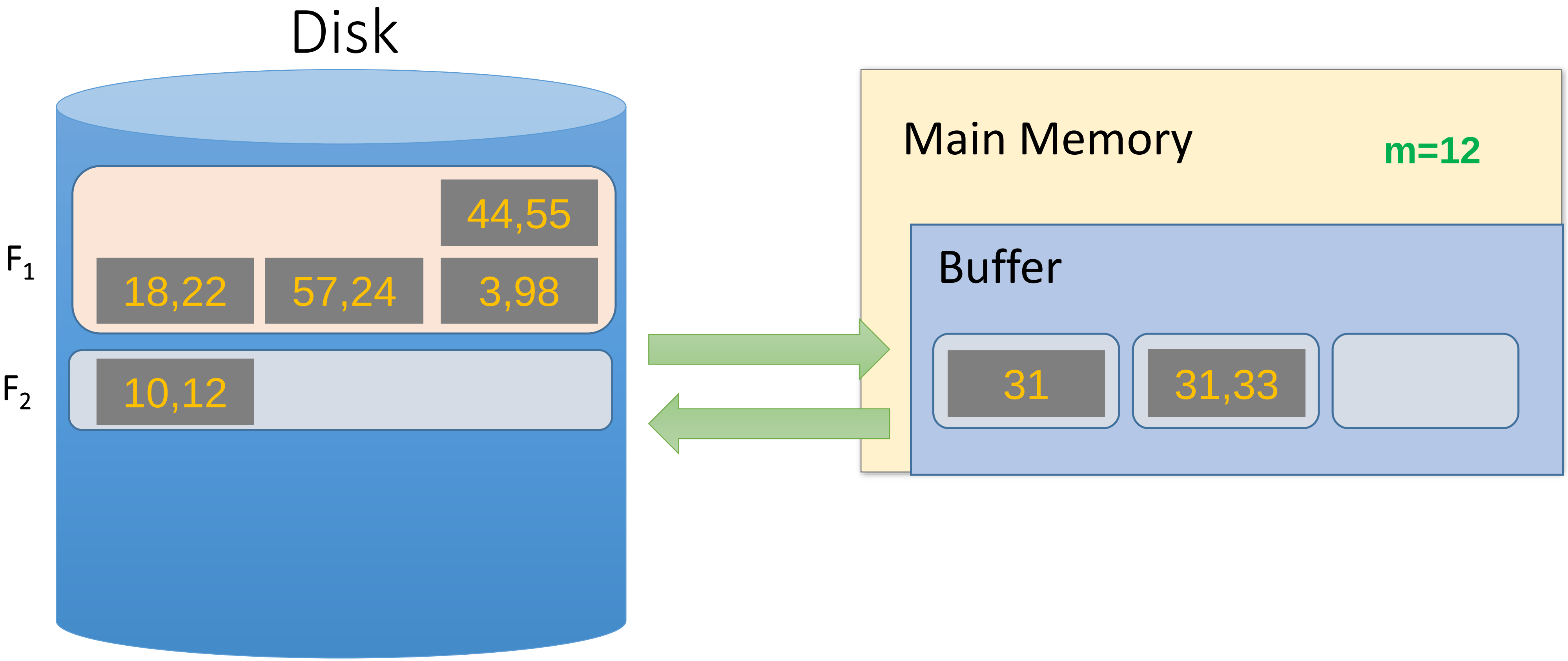
- Take the minimum two values, and put in output page

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



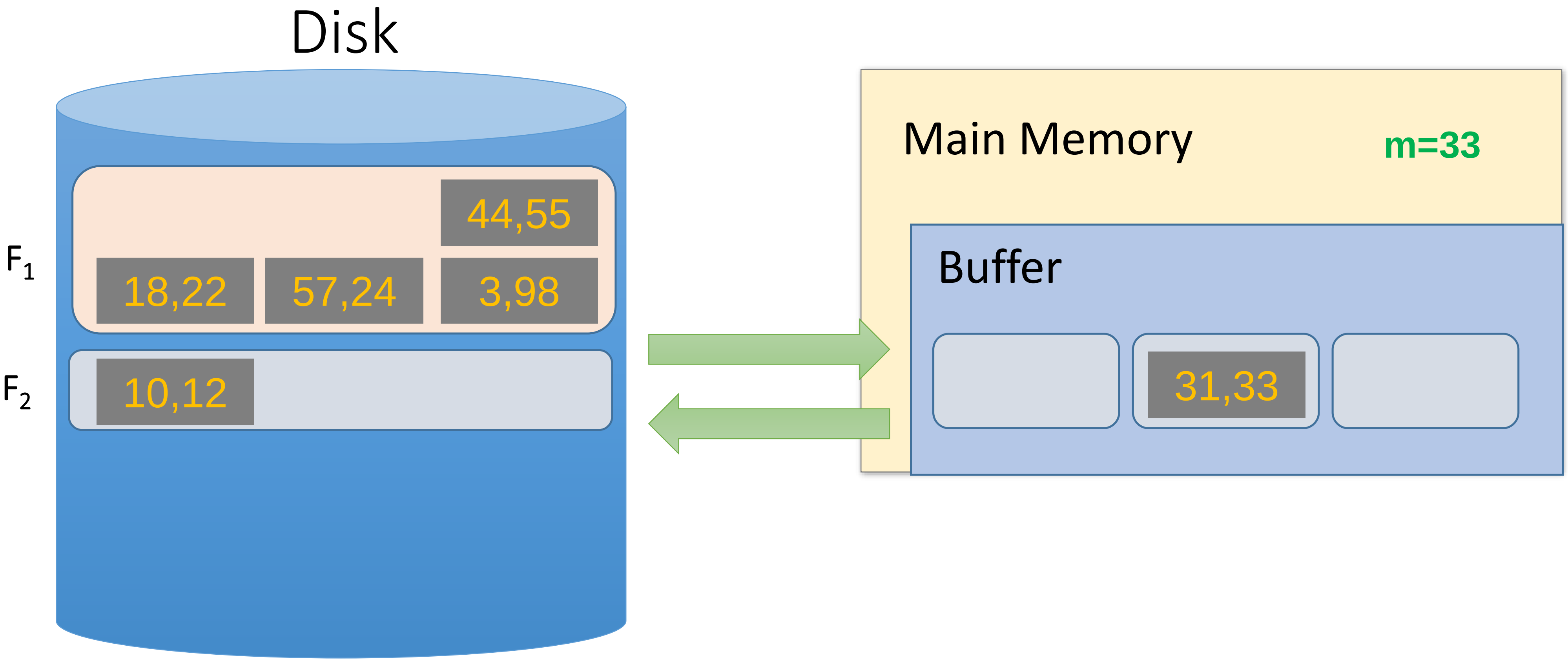
Repacking Example: 3 page buffer

- Next, ***repack***



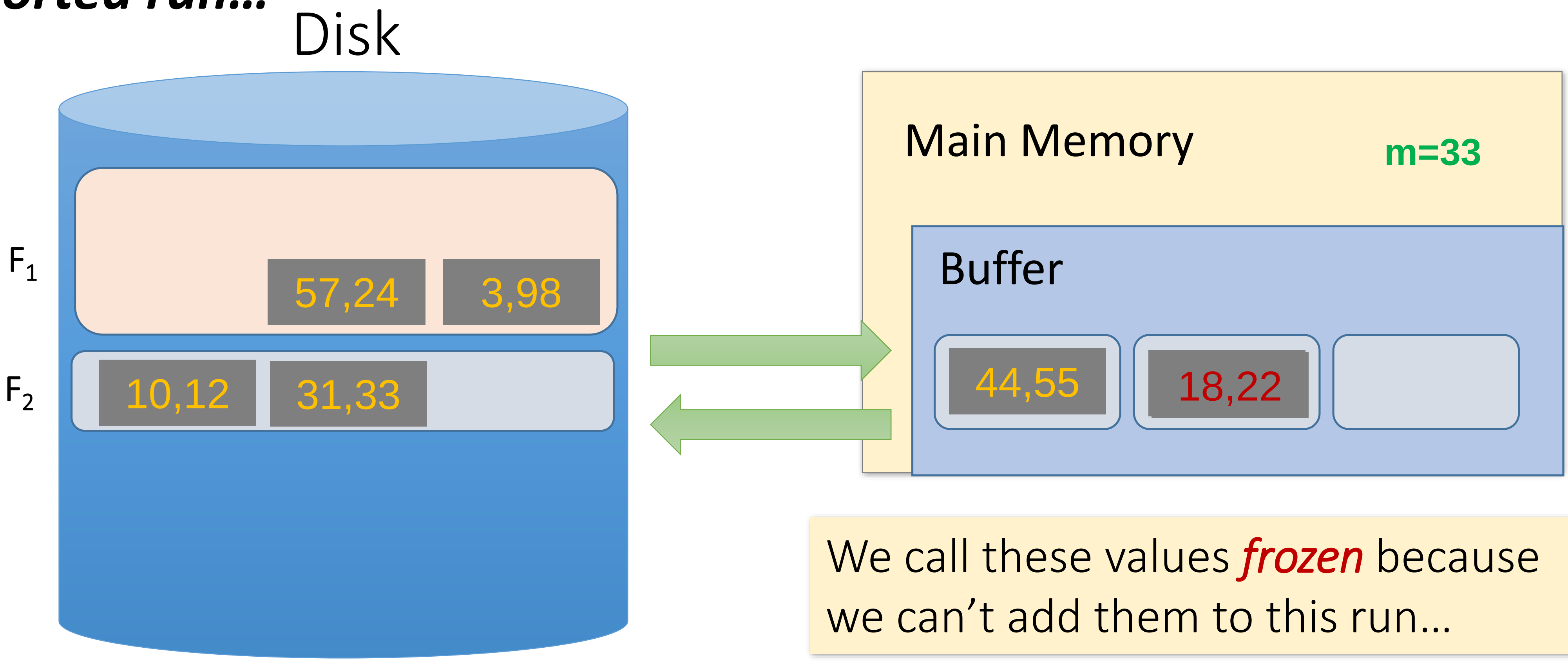
Repacking Example: 3 page buffer

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



Repacking Example: 3 page buffer

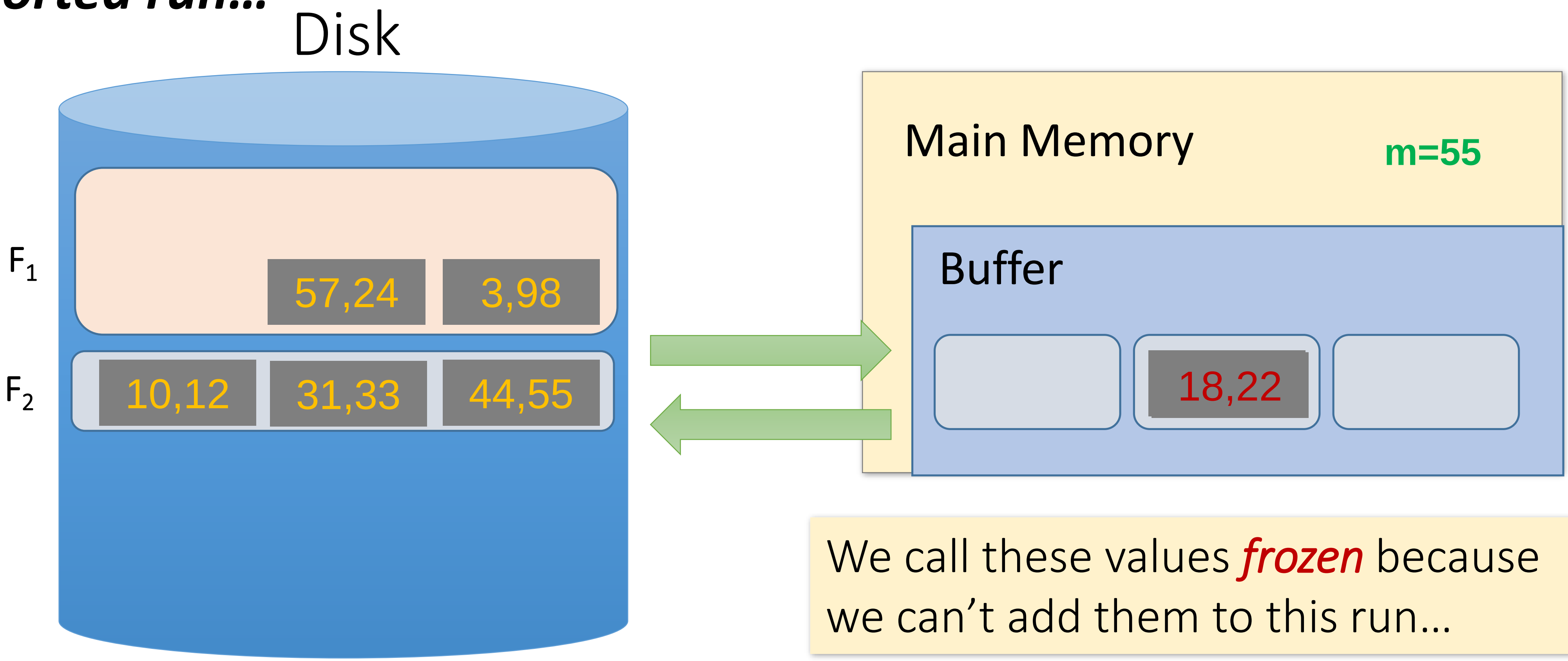
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



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

Repacking Example: 3 page buffer

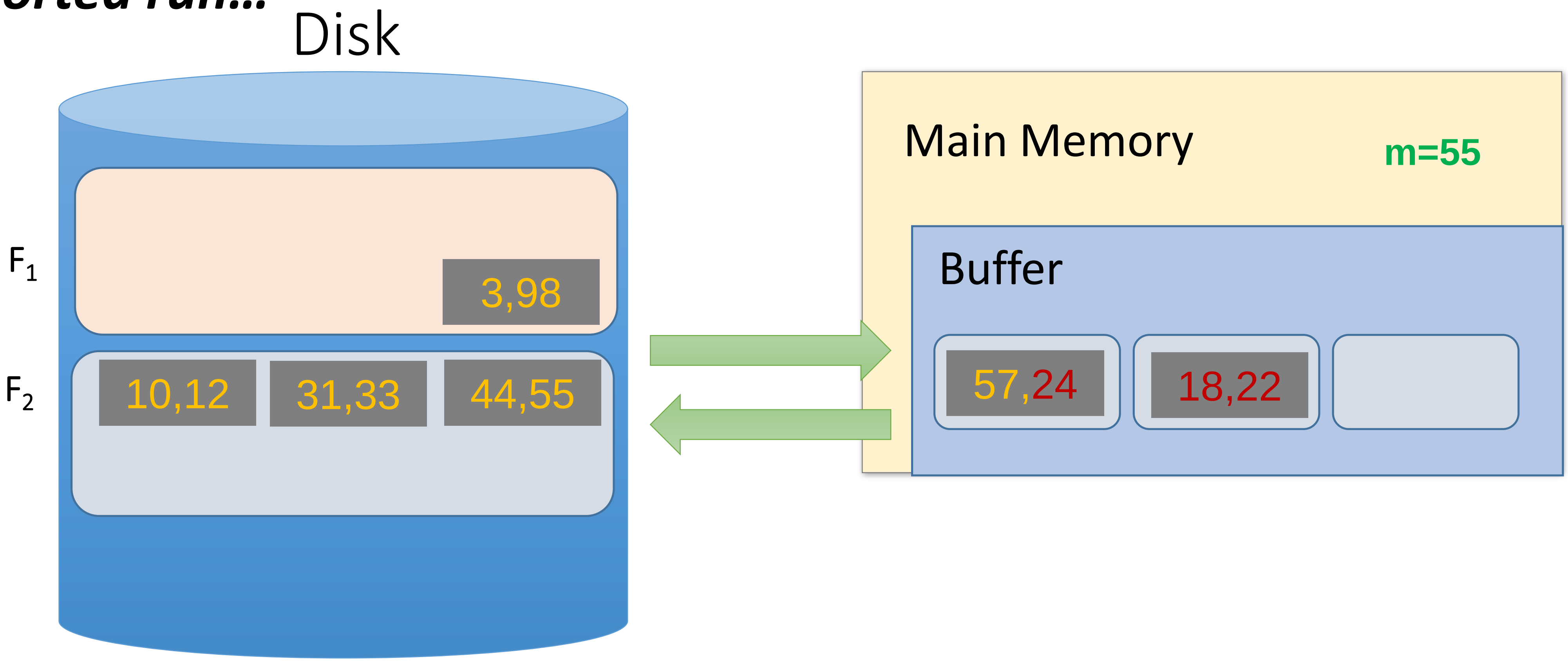
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



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

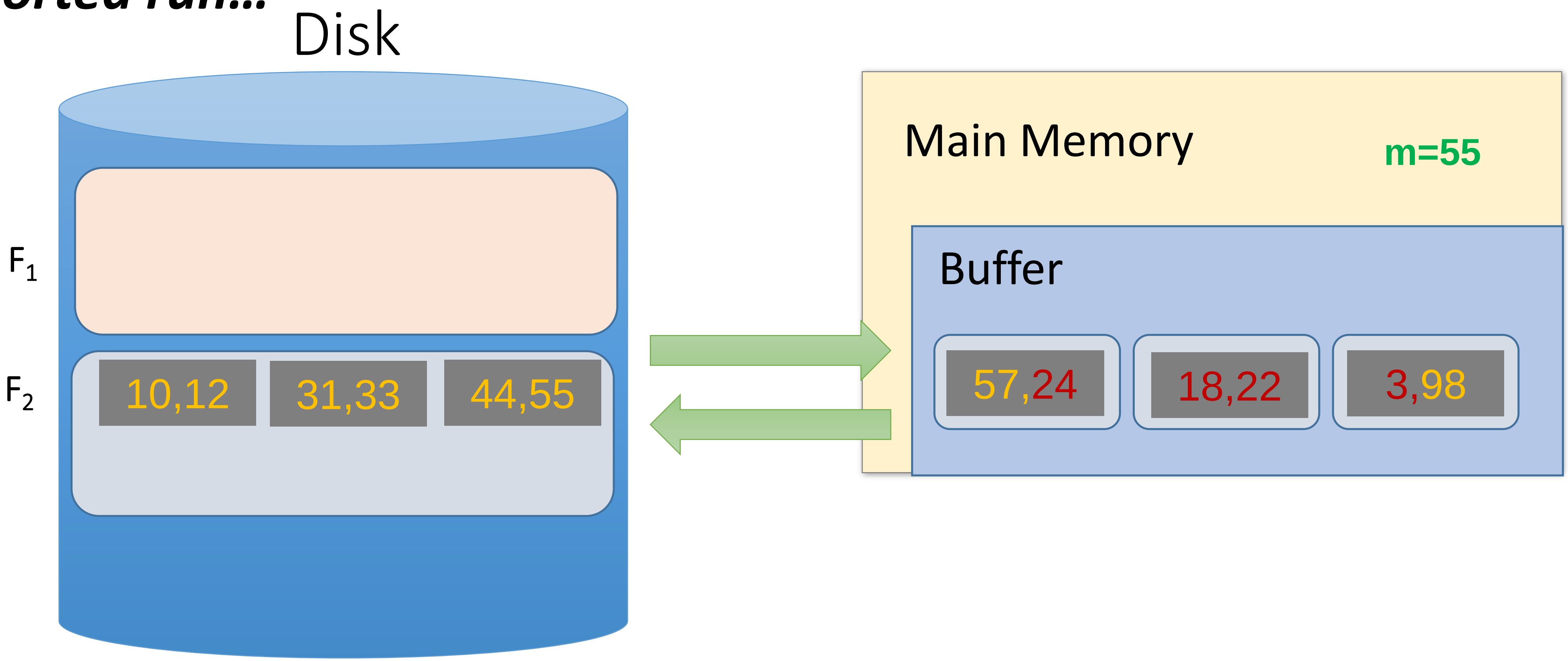
Repacking Example: 3 page buffer

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



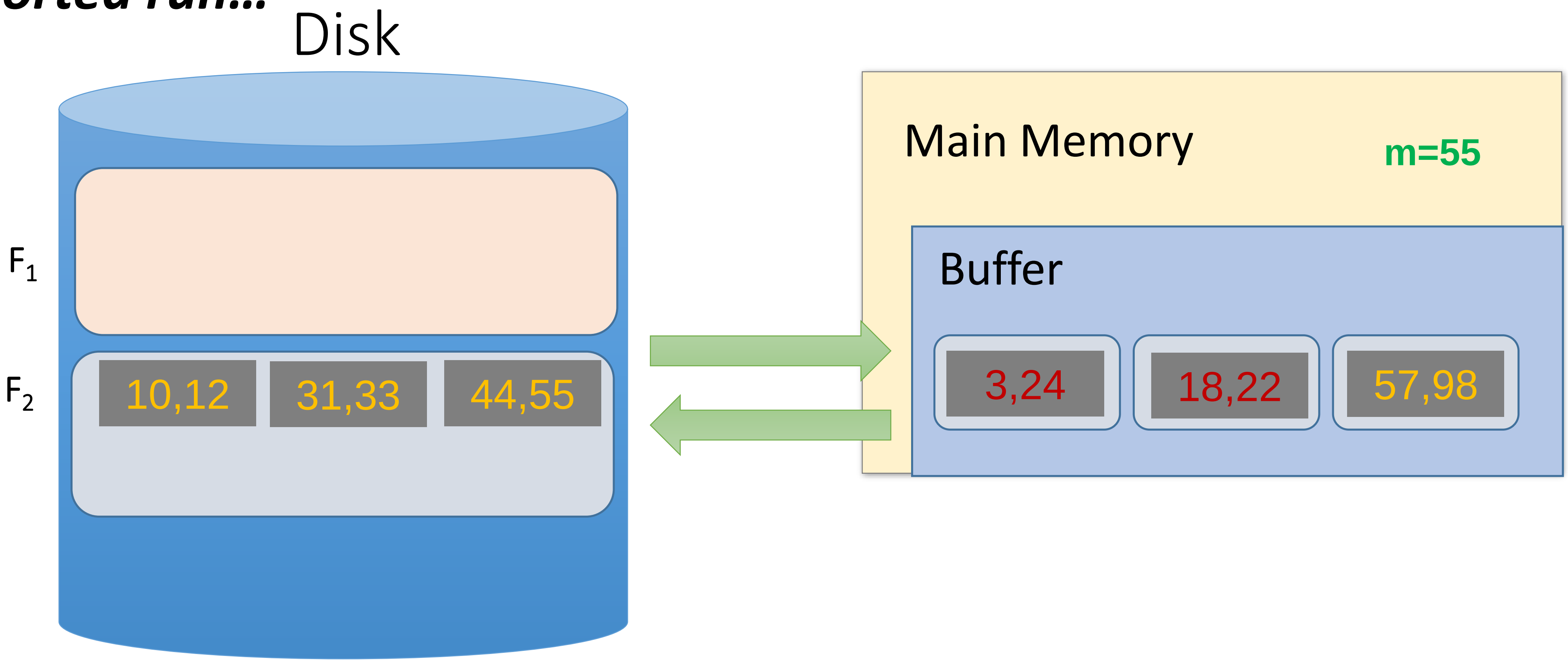
Repacking Example: 3 page buffer

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



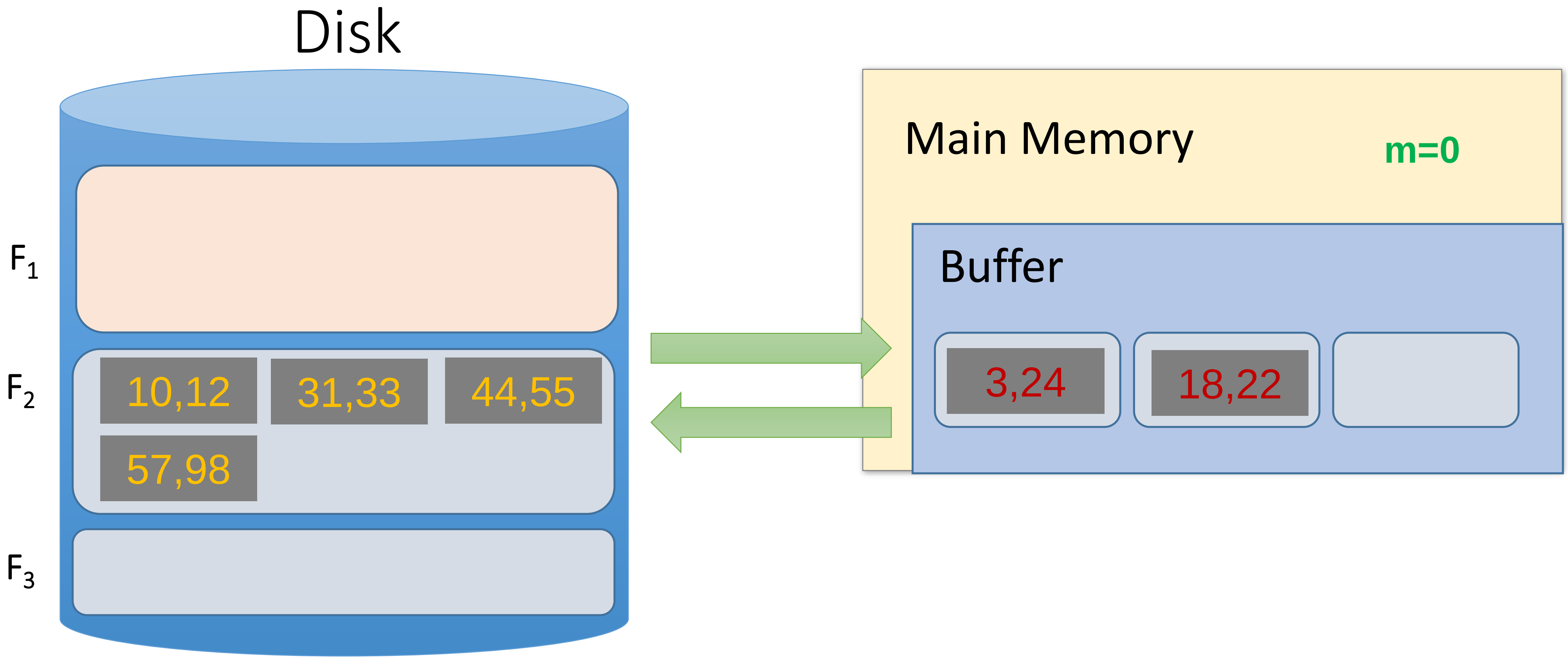
Repacking Example: 3 page buffer

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



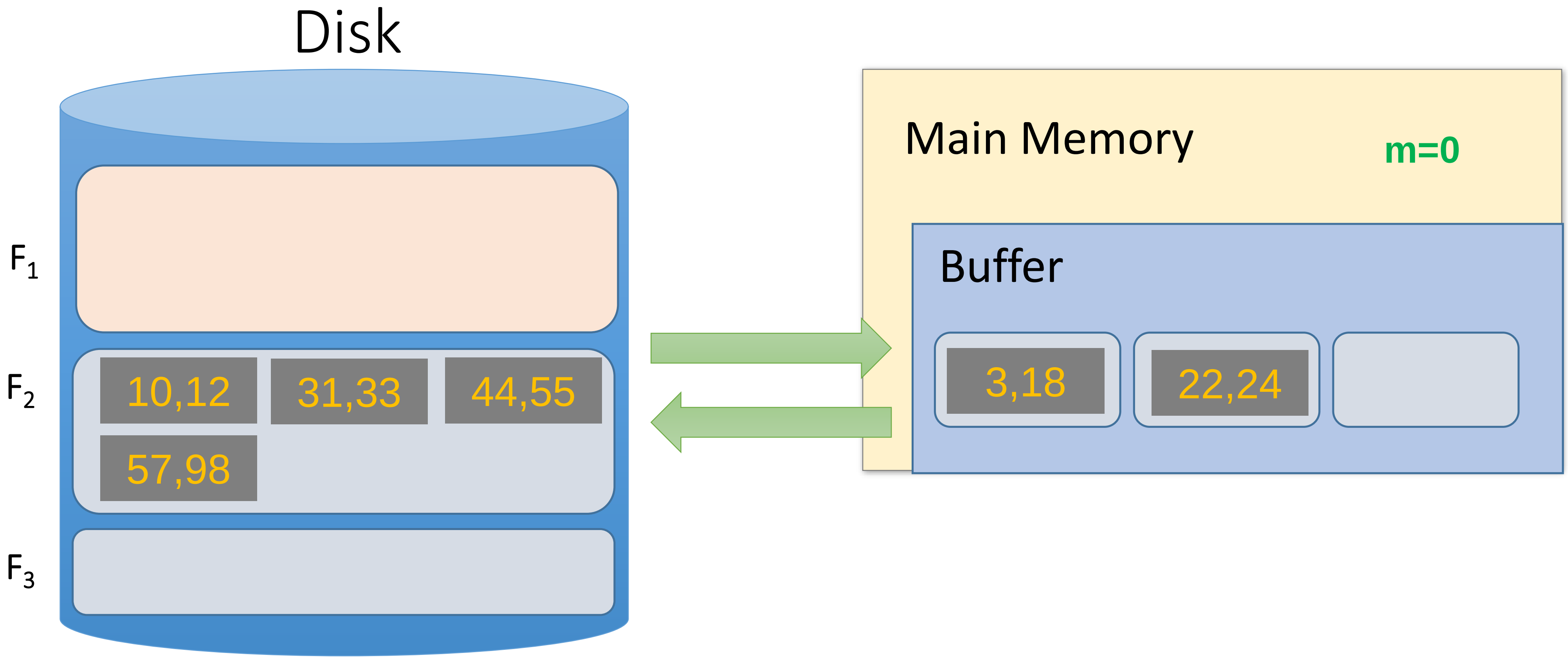
Repacking Example: 3 page buffer

- 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:
  - **Best case:** If input file is sorted → nothing is frozen → we get **a single** run!
  - **Worst case:** If input file is reverse sorted → everything is frozen → we get runs of length **B+1**

- In general, with repacking we do **no worse** than without it!

- Engineer's approximation: runs will have **~2(B+1)** length

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

Summary

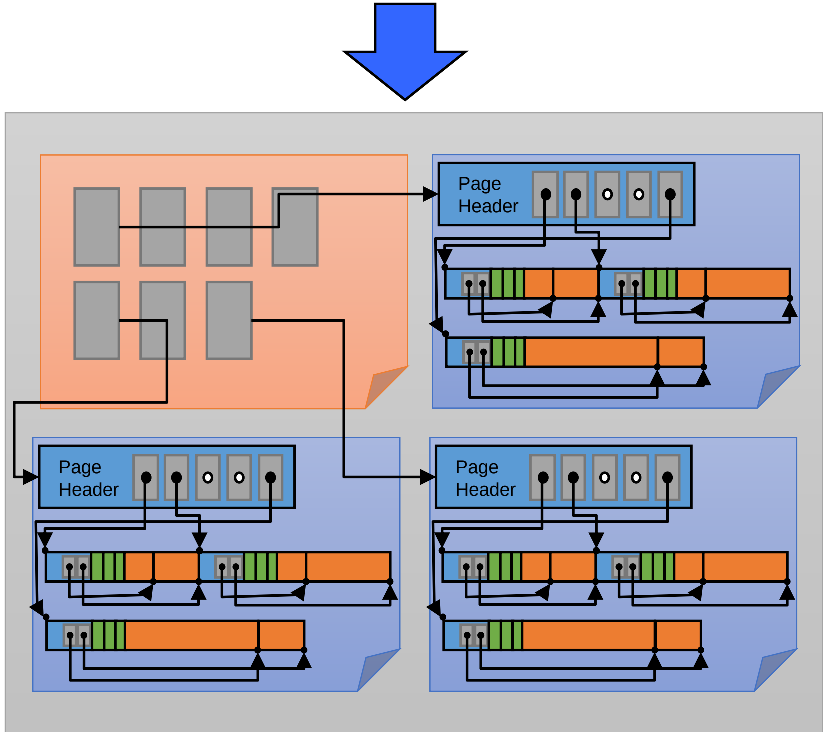
- We introduced the IO cost model using **sorting**.
- Described a few optimizations for sorting

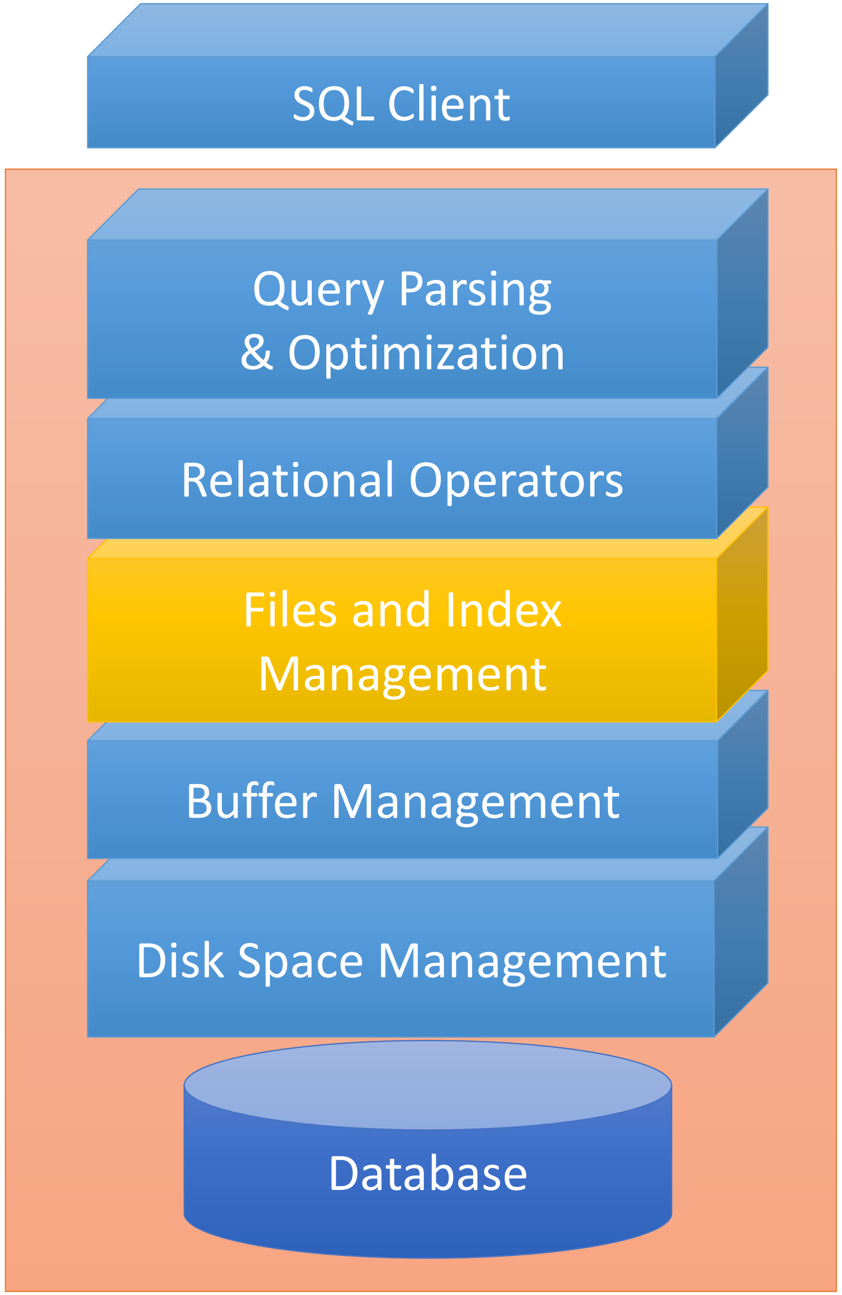
2. File and Page

Architecture of a DBMS

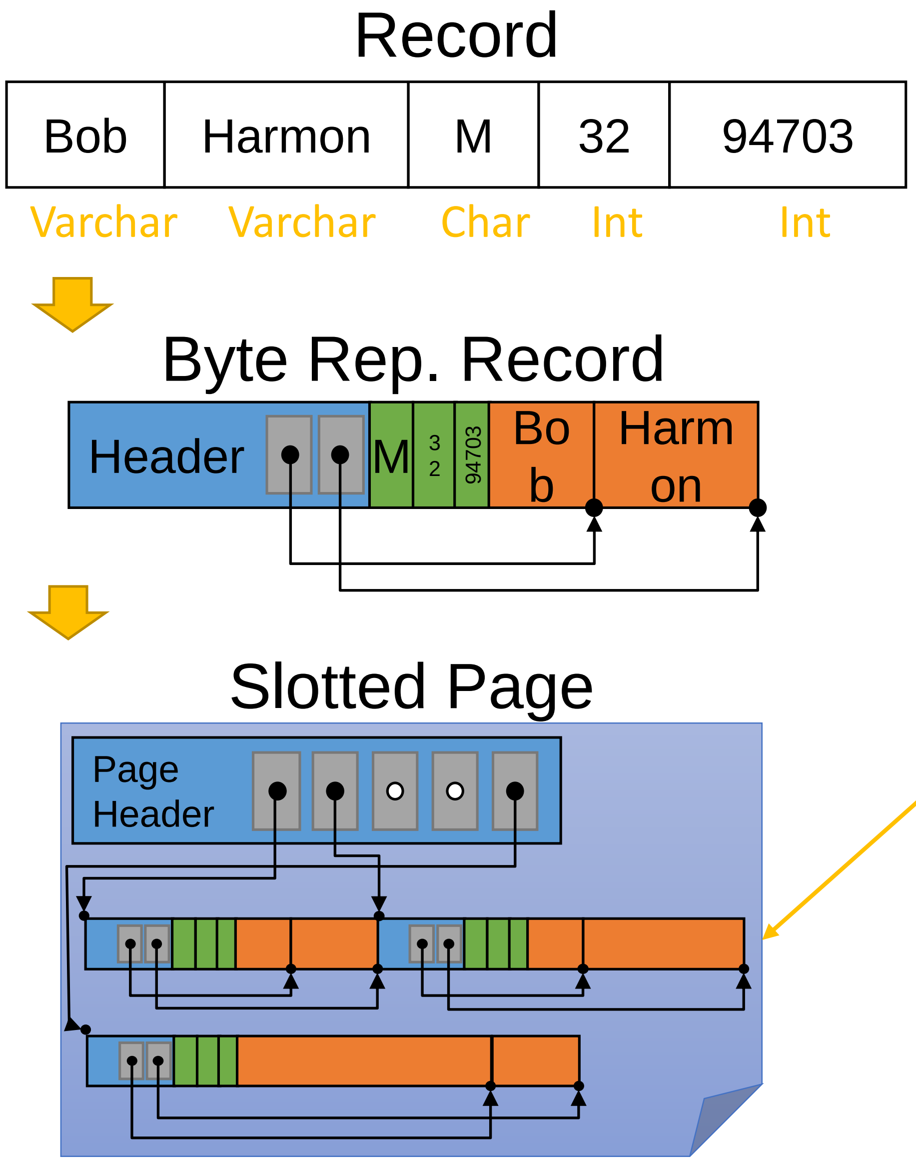
Organizing tables and records as groups of pages in a logical file.

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110



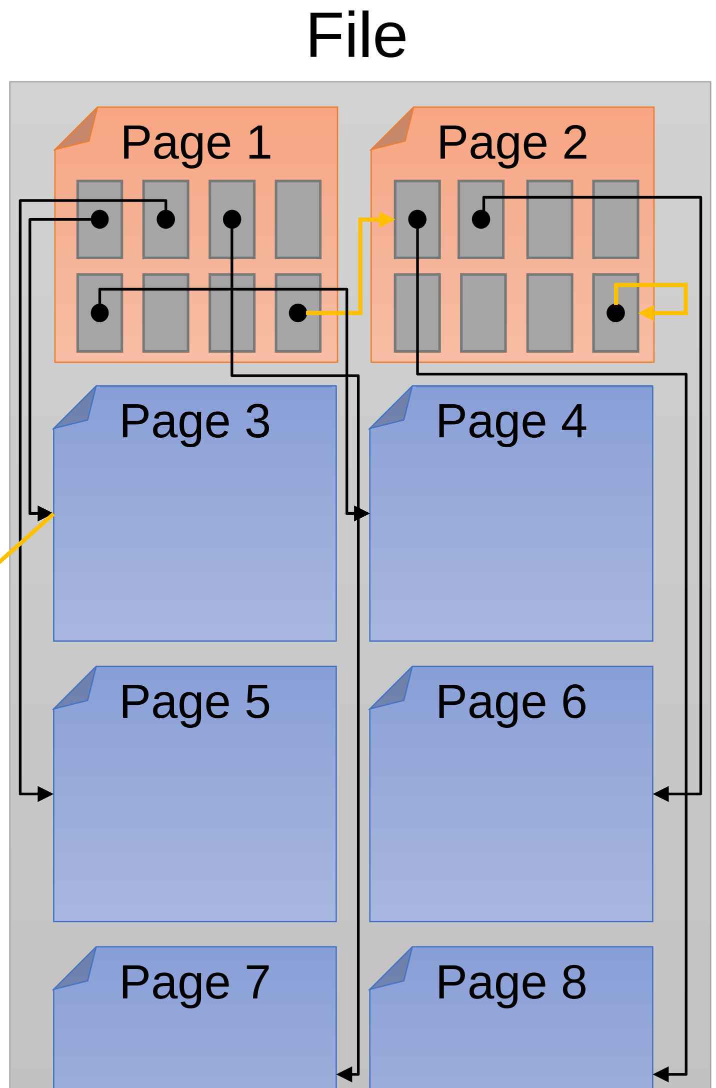


Files



Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110



Files of Pages of Records

- Higher levels of DBMS operate on *pages of records* and *files of pages*.

- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - fetch a particular record by ***record id*** ...
    - *Think: pointer encoding Page_ID and location on page.*
  - scan all records (possibly with some conditions on the records to be retrieved)

- Could span multiple OS files and even machines
  - Or "raw" disk devices

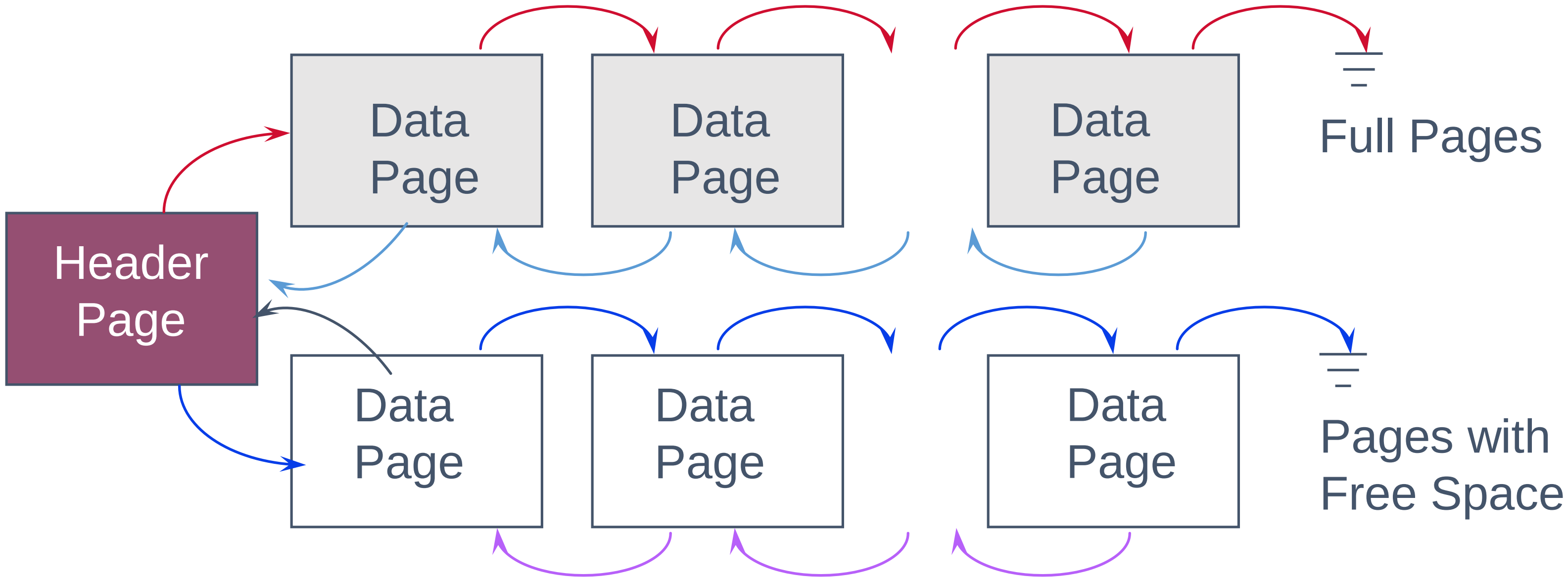
Many Kinds of Database Files

- Unordered Heap Files
  - Records placed arbitrarily across pages
- Clustered Heap File and Hash Files
  - Records and pages are grouped
- Sorted Files
  - Page and records are in sorted order
- Index Files
  - B+ Trees, Hash Tables, …
  - May contain records or point to records in other files

Basic Unordered Heap Files

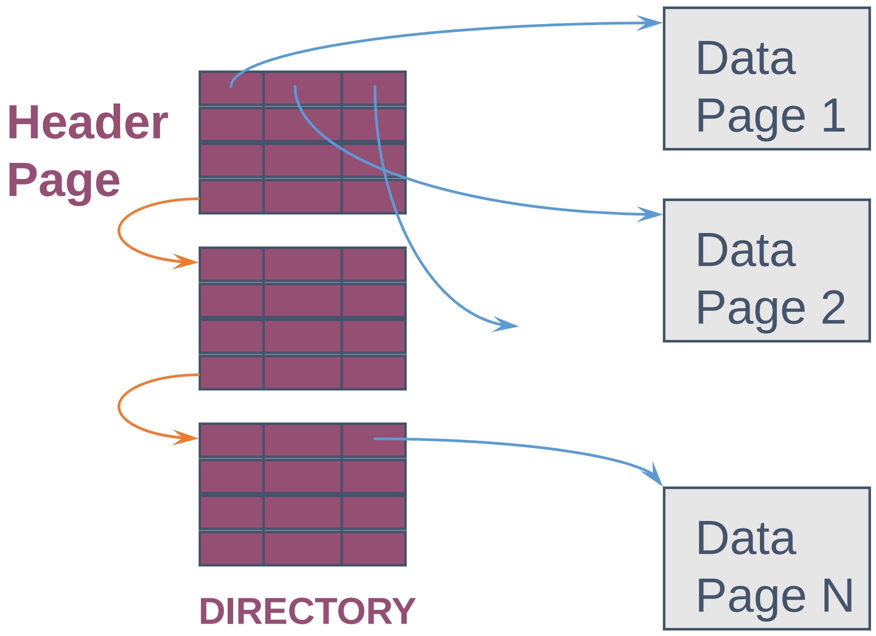
- Collection of records in no particular order
  - Not to be confused with ”heap data-structure”
- As file shrinks/grows, pages (de)allocated
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this, we’ll consider two in this class

Heap File Implemented as a List



- Header page ID and Heap file name stored elsewhere
  - Database “catalog”
- Each page contains 2 “pointers” plus **free-space** and **data**.
- What is wrong with this?
  - How do I find a page with enough space for a 20 byte record?

Better: Use a Page Directory



- Directory entries include **#free bytes** on the page.
- Header pages accessed often → likely in cache
  - What eviction policy is best here?
- Finding a page to fit a record required far fewer page loads than linked list (Why?)
  - One header page load reveals free space of many pages.

Indexes (sneak preview)

- A Heap file allows us to retrieve records:
  - by specifying the *record id (page id + slot)*
  - by scanning all records sequentially

- Would like to fetch records *by value*, e.g.,
  - Find all students in the "CS" department
  - Find all students with a gpa > 3 AND blue hair

- Indexes: file structures for efficient value-based queries

Overview

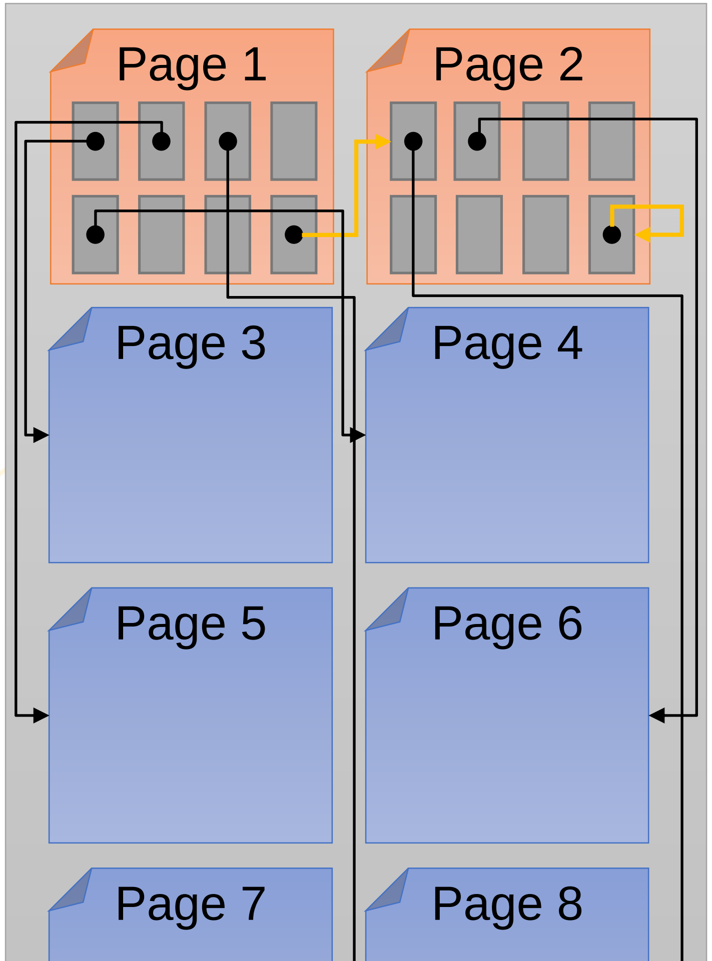
Summary

Table encoded as files which are collections of pages.

Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110

Heap File



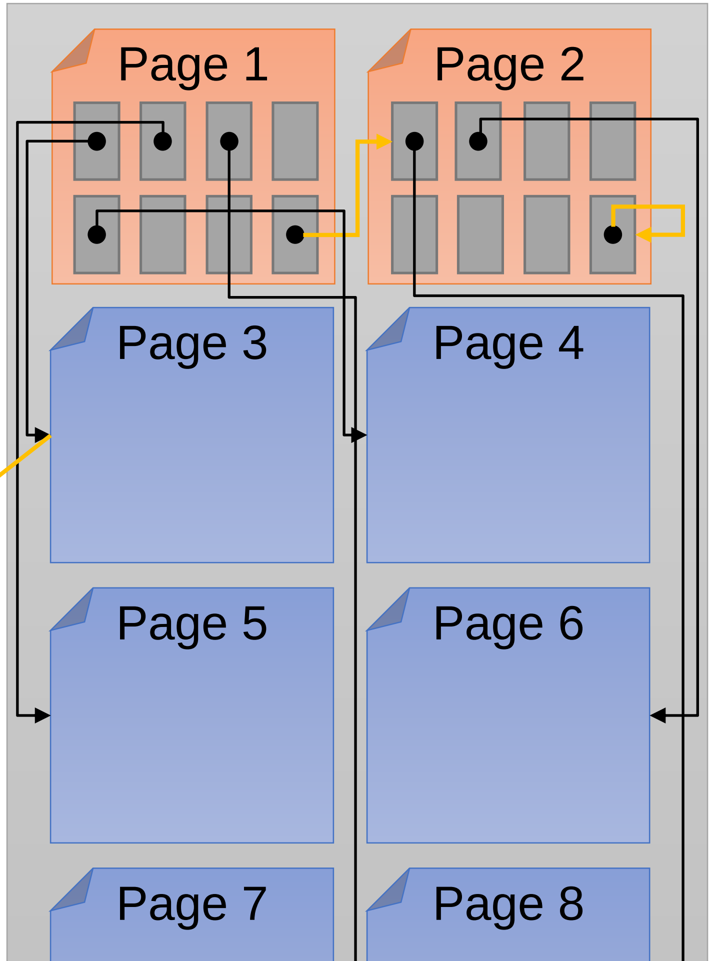
Overview

How do we store records on a page?

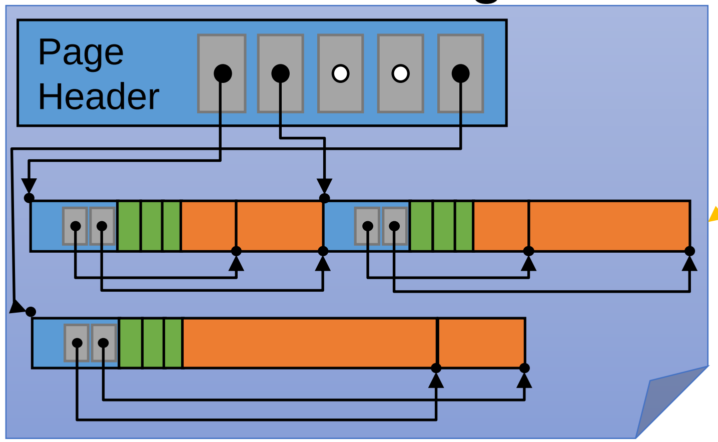
Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110

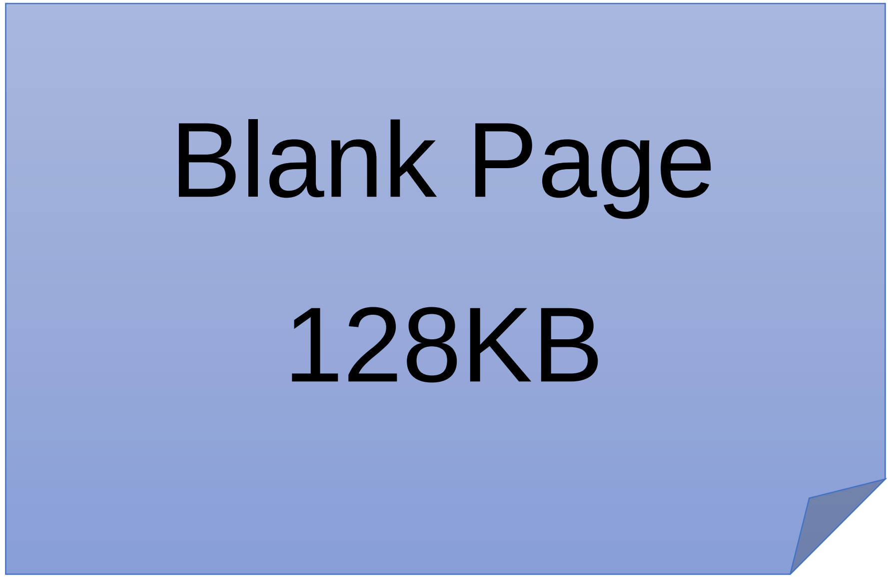
Heap File



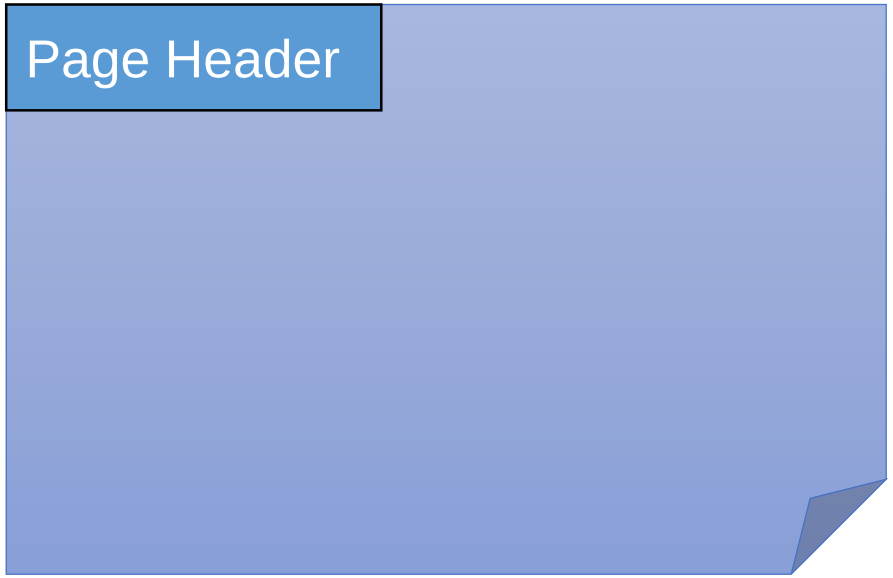
Slotted Page



Page Basics: The Header



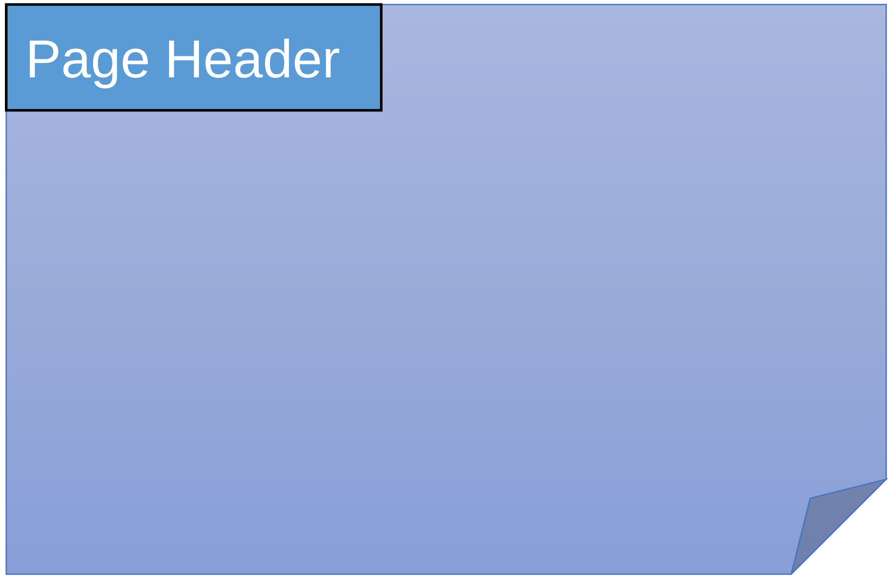
Page Basics: The Header



Header may contain:

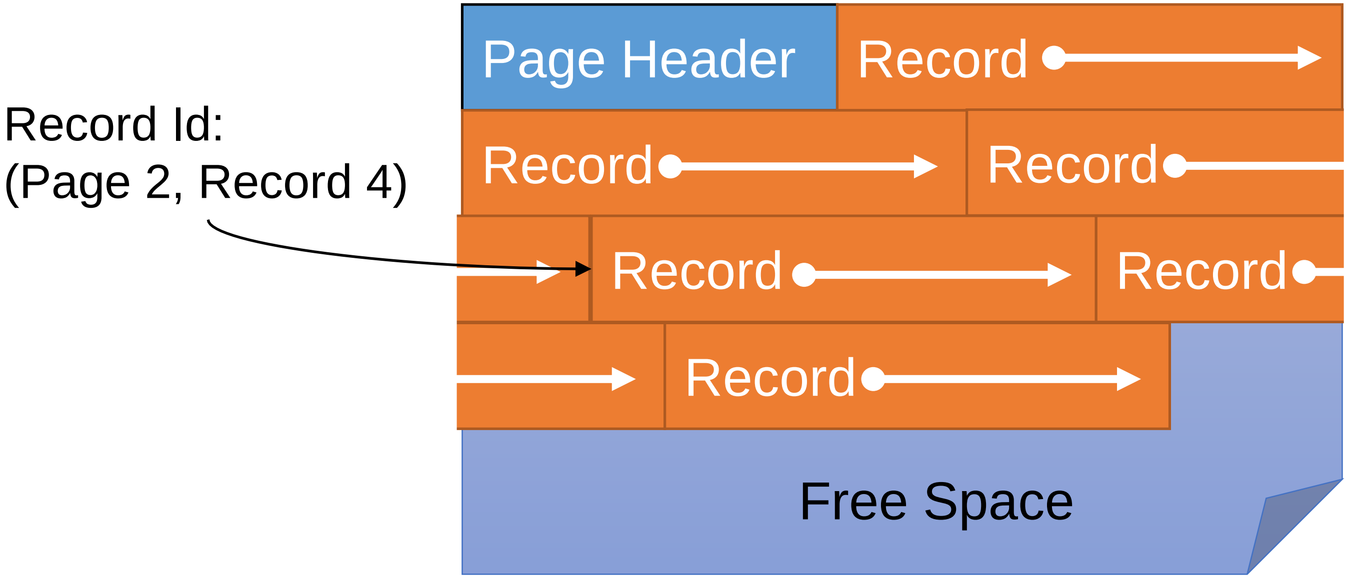
- Number of records
- Free space
- Maybe next/last pointer
- Slot Table … more soon

Things to Consider



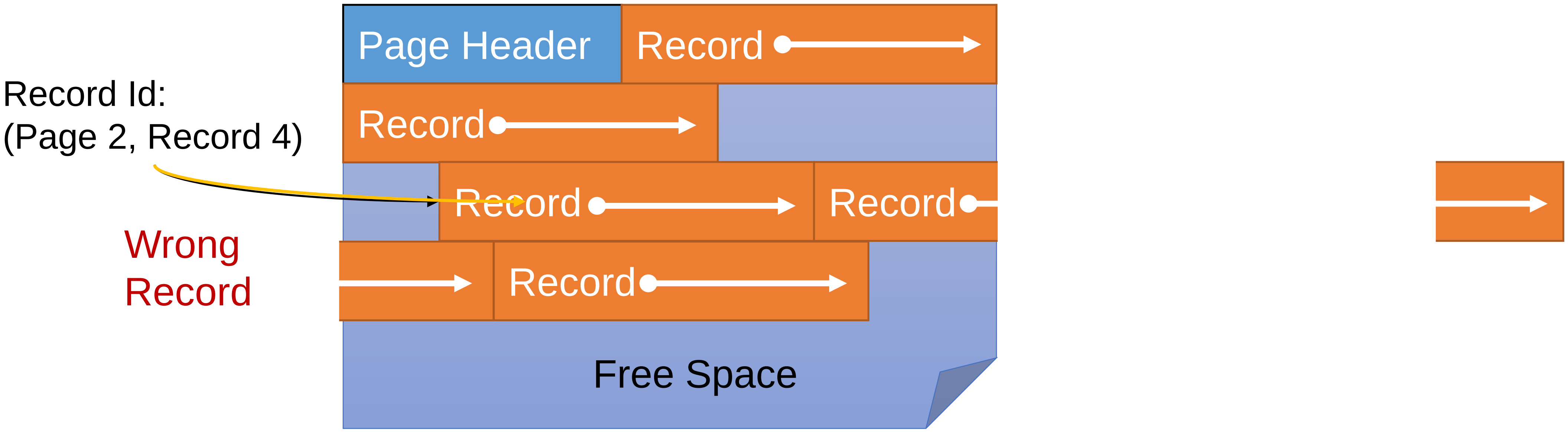
- Record length? *Fixed* or *Variable*
- Find records by record id? *Offset…*
- How do we add and delete records?
  - Bitmaps & Slot Tables

Fixed Length Records: Packed



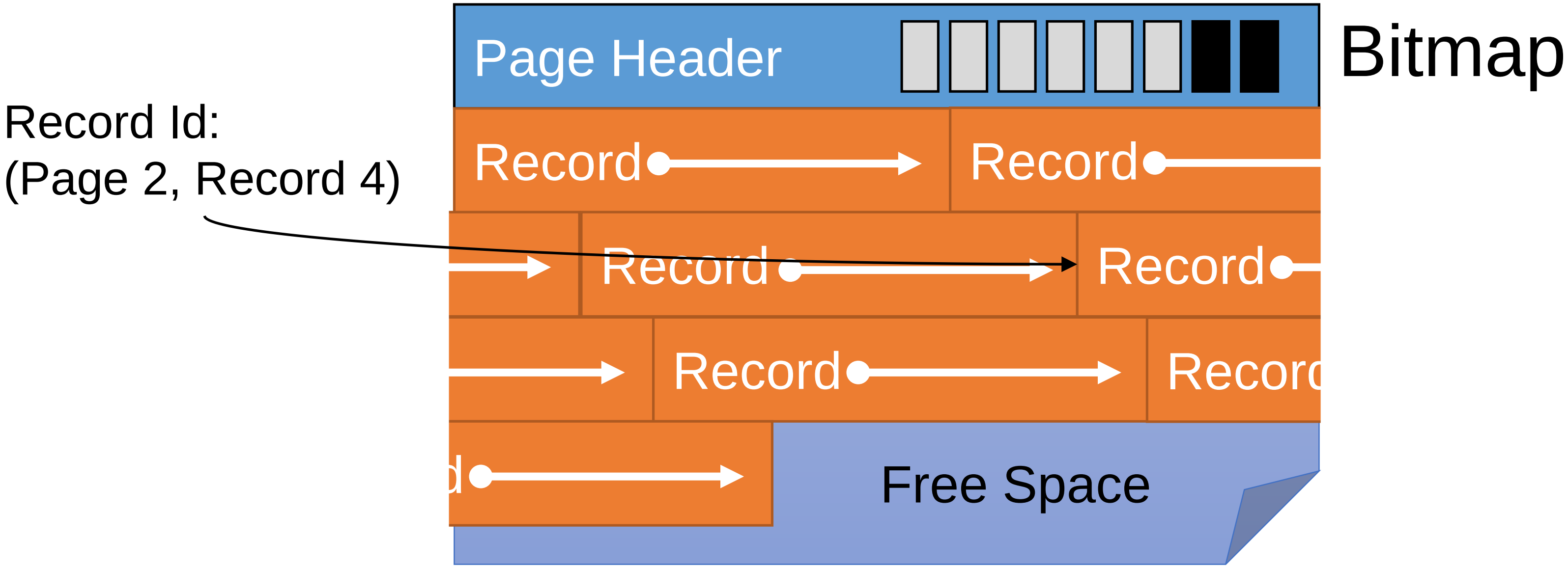
- Pack records densely
- Record id: record number in page
- Easy to add: just append
- Delete?

Fixed Length Records: Packed



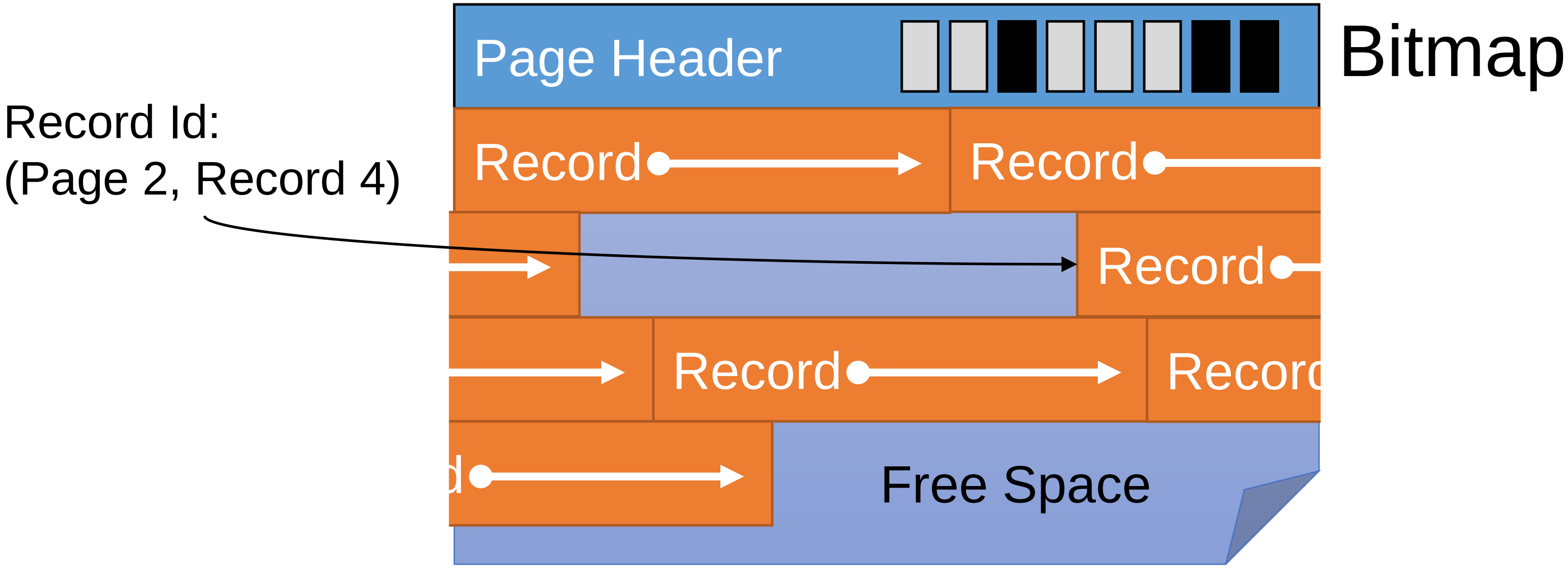
- Pack records densely
- Record id: record number in page
- Easy to add: just append
- Delete? Re-arrange …

Fixed Length Records: Unpacked



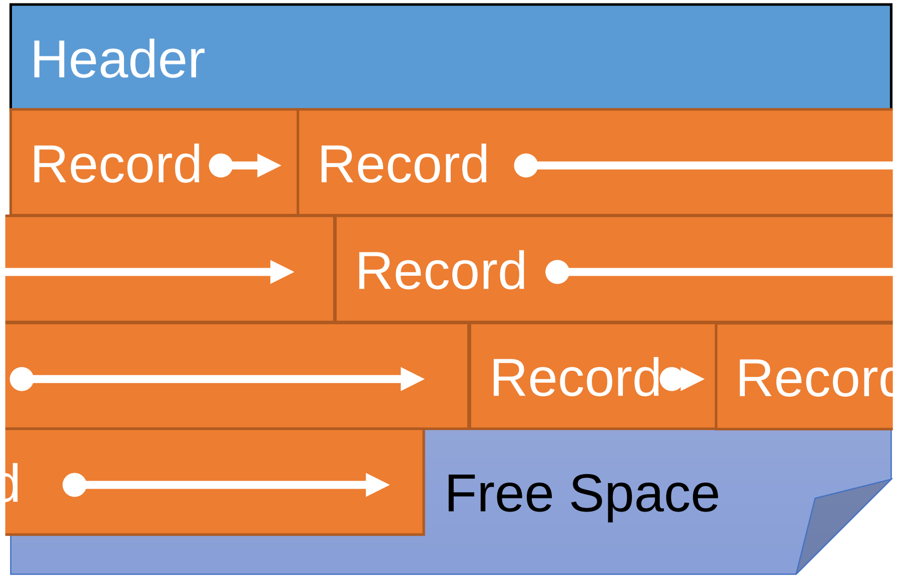
- Bitmap denotes “slots” with records
- Record id: record number in page
- **Insert:** find first empty slot
- **Delete?**

Fixed Length Records: Unpacked



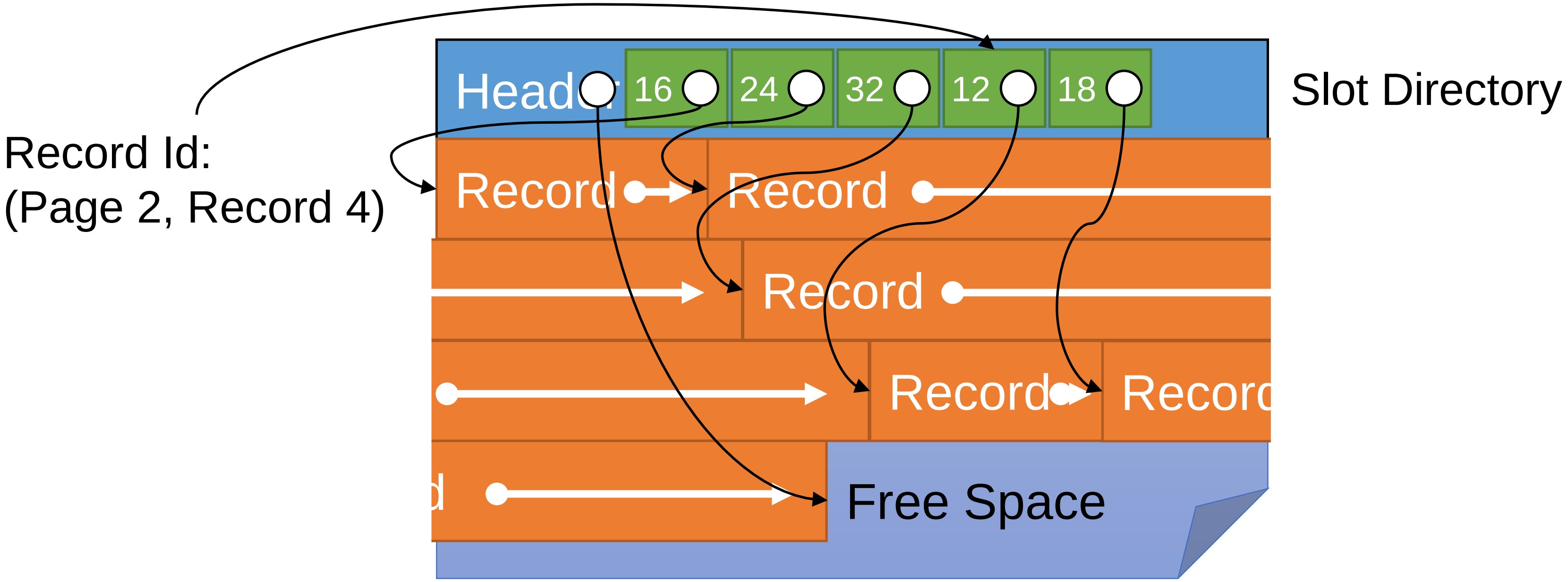
- Bitmap denotes "slots" with records
- Record id: record number in page
- **Insert:** find first empty slot
- **Delete:** Clear bit

Variable Length Records



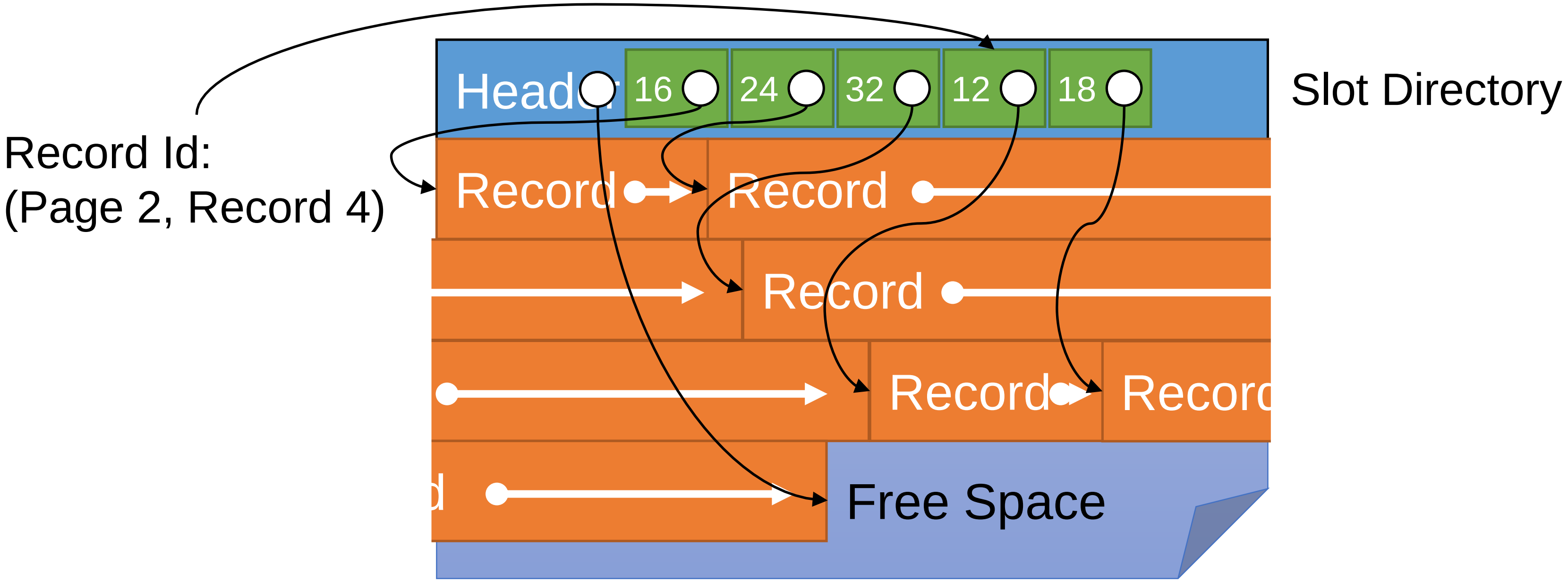
- How do we know where each record begins?
- What happens when we add and delete records?

Slotted Page



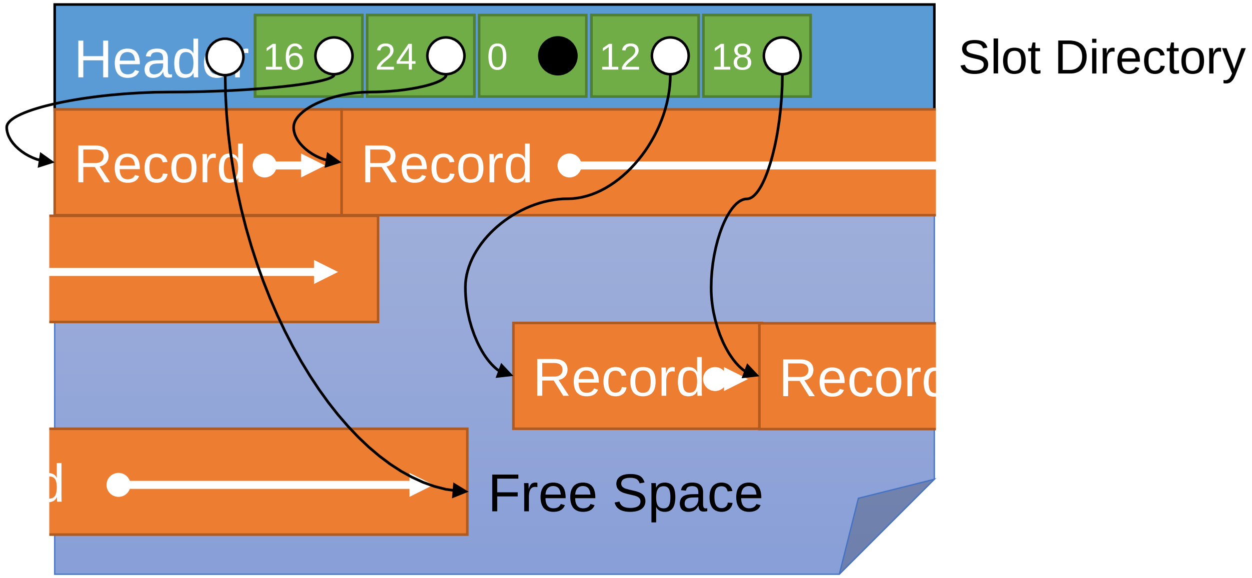
- Introduce **slot directory** in header
  - Length + Pointer to beginning of record
  - Pointer to free space
- Record ID = location in slot table
- ***Delete***? (e.g., 3rd record on the page)

Slotted Page



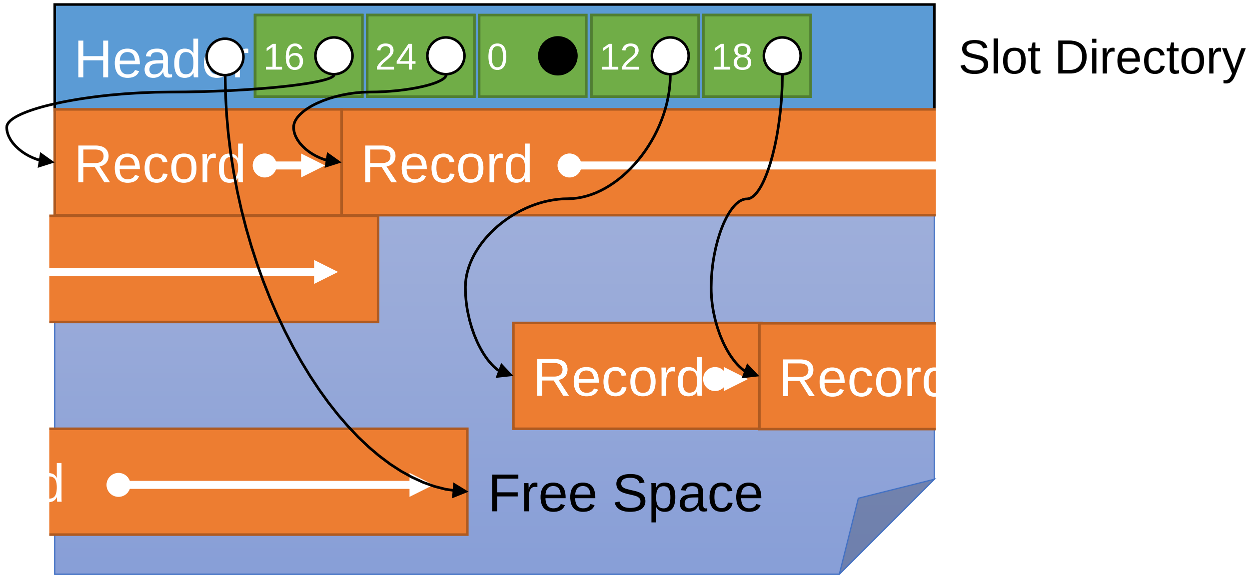
- **Delete**: Set pointer to null.
  - Doesn't affect pointers to other records
  - However, need to make sure we remove any references to **record_id** in indexes

Slotted Page



- ***Insert*?**

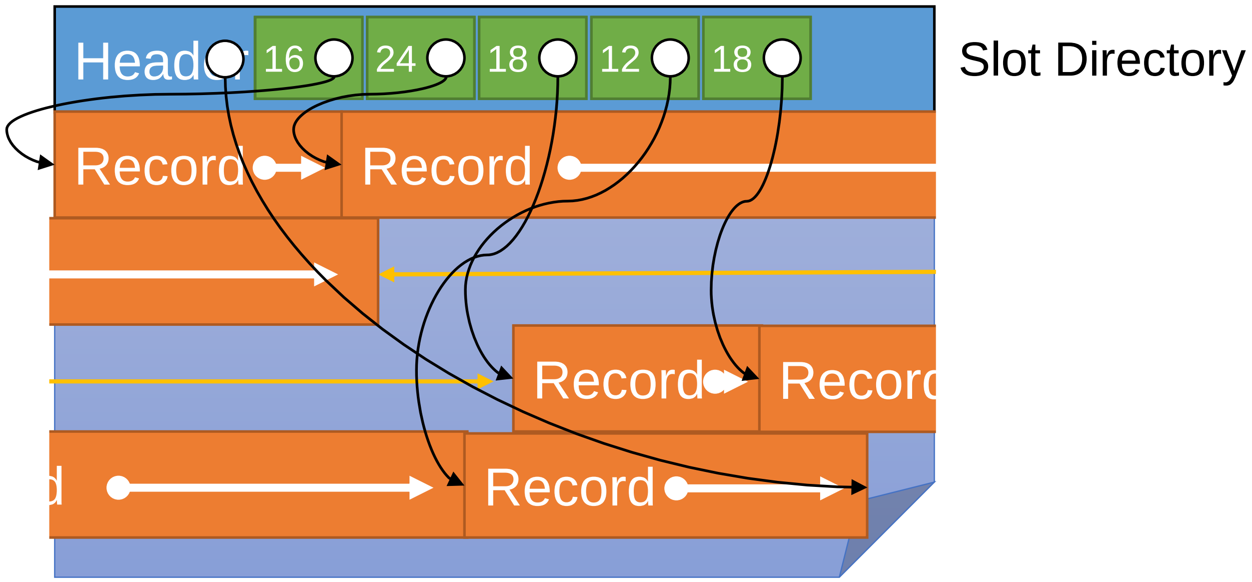
Slotted Page



- **Insert:**

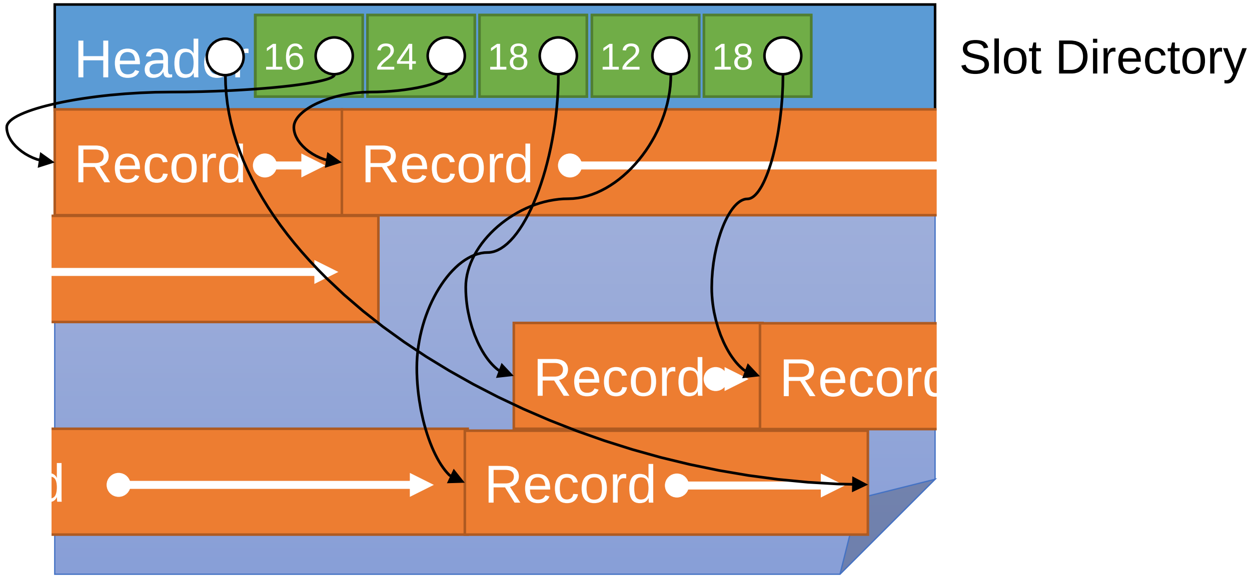
  – Place record in free space on page

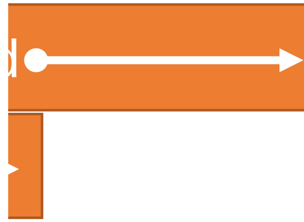
Slotted Page



- **Insert**:
  - Place record in free space on page
  - Create pointer in next open slot in slot directory
  - Fragmentation?

Slotted Page

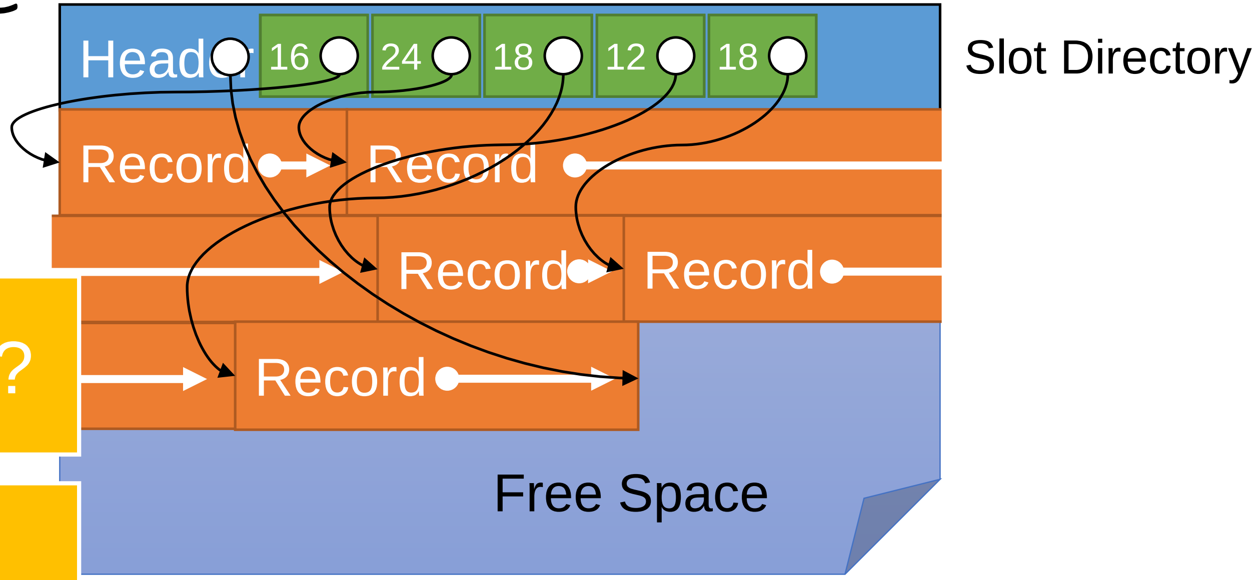




- **Insert**:
  - Place record in free space on page
  - Create pointer in next open slot in slot directory
  - Reorganize data on page.

Is this safe?

Slotted Page



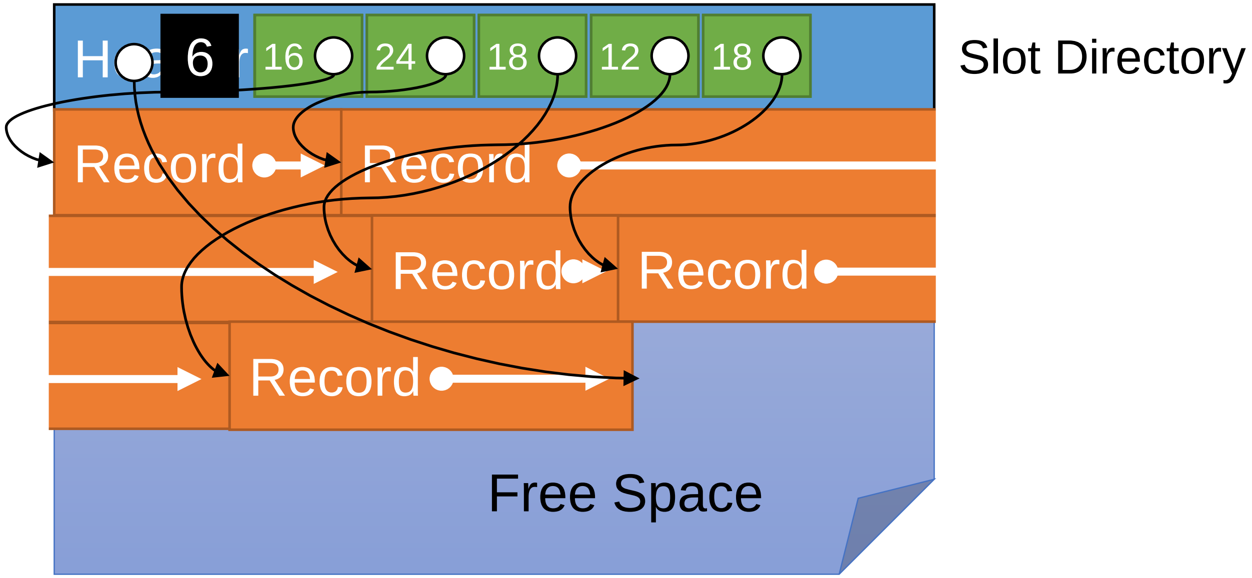
Is this safe?

When?

What if we need more slots?

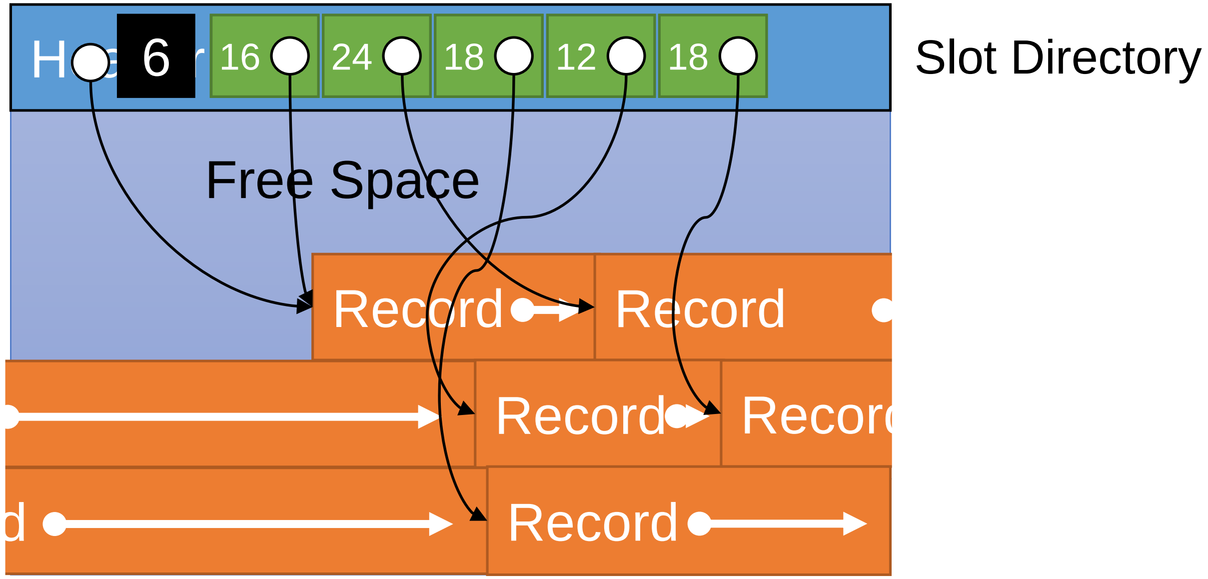
- **Insert**:
  - Place record in free space
  - Create pointer in next open slot in slot directory
  - Reorganize data on page.

Slotted Page: Growing Slots



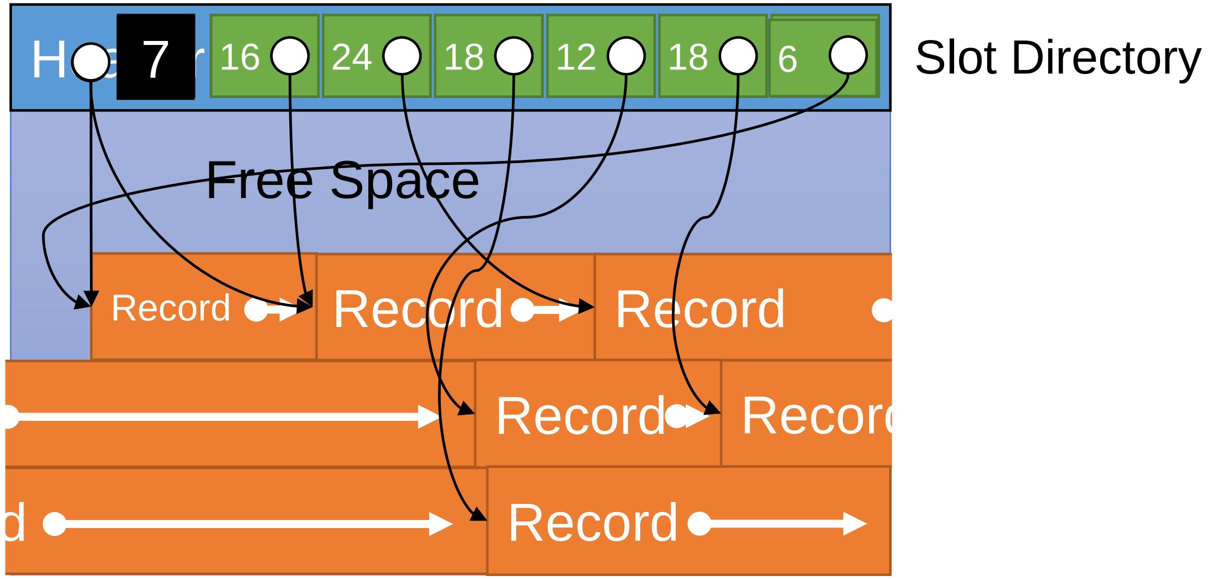
- Track number of slots in slot directory

Slotted Page: Growing Slots



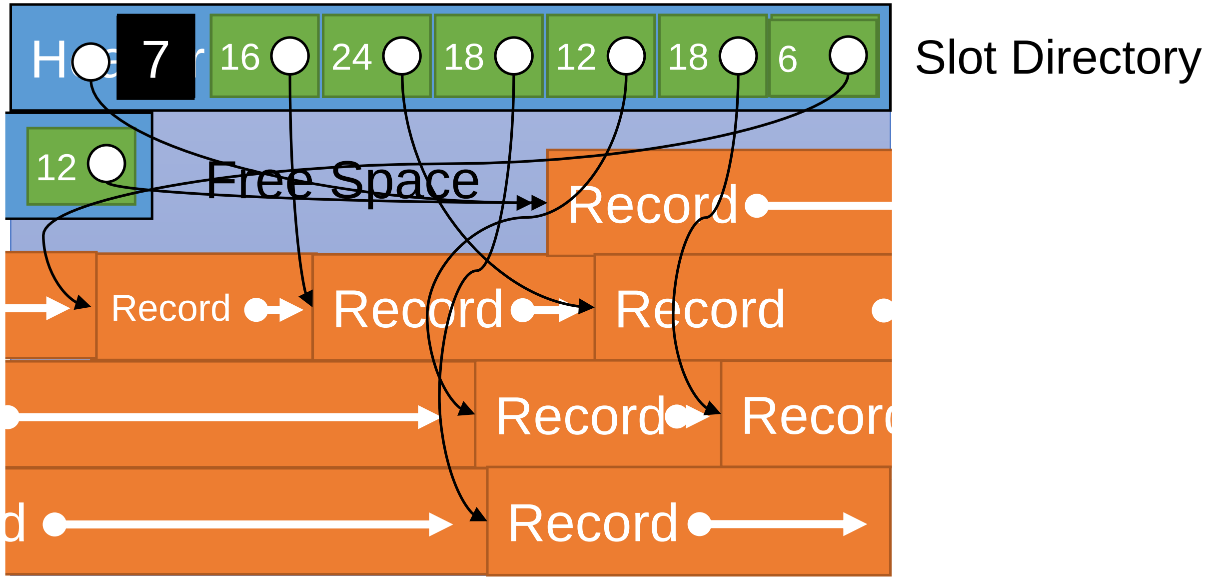
- Track number of slots in slot directory
- Grow records from other end of page
  - Why?

Slotted Page: Growing Slots



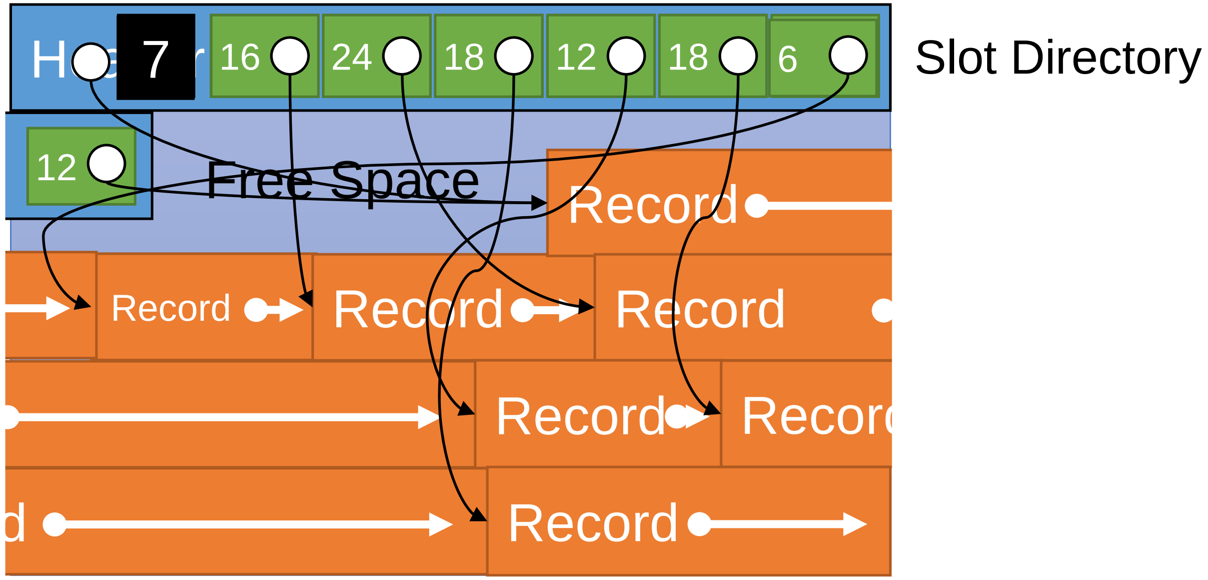
- Track number of slots in slot directory
- Grow records from other end of page
- Extend slot directory on insert
  - Add record in free space & update counter

Slotted Page: Growing Slots



- Track number of slots in slot directory
- Grow records from other end of page
- Extend slot directory on insert
  - Add record in free space & update counter

Slotted Page Summary

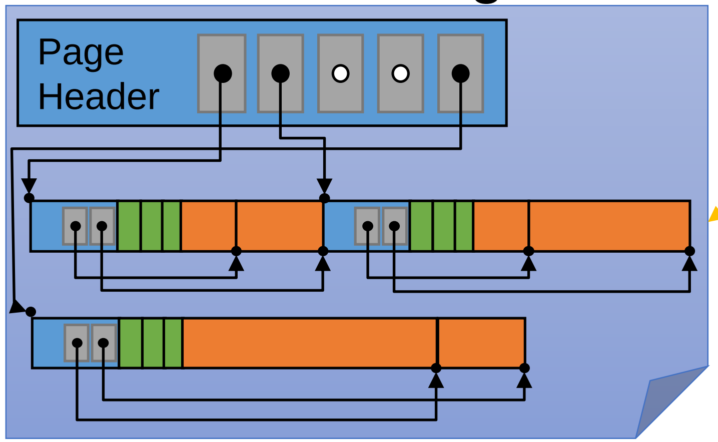


- Typically use slotted Page
  - Good for variable and fixed length records
- Good for fixed length records too. Why?
  - Re-arrange (e.g., sort) and squash null fields

Overview

Store records on slotted page

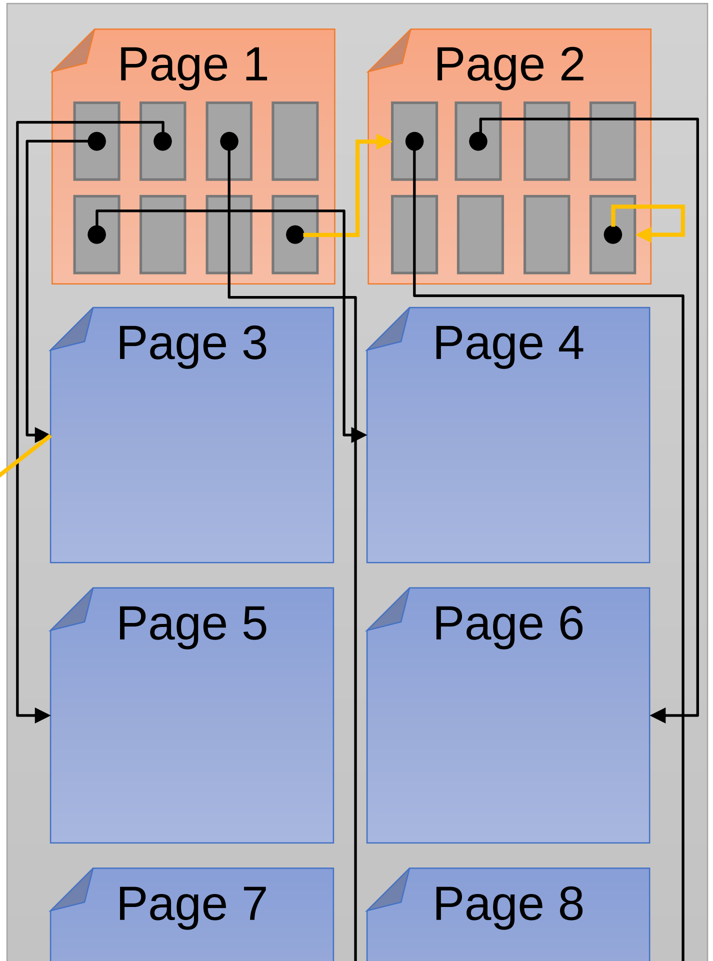
Slotted Page



Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110

Heap File

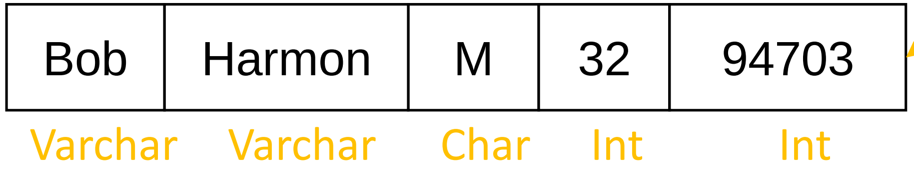


Overview

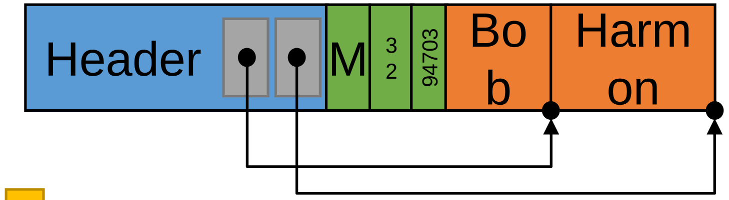
Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110

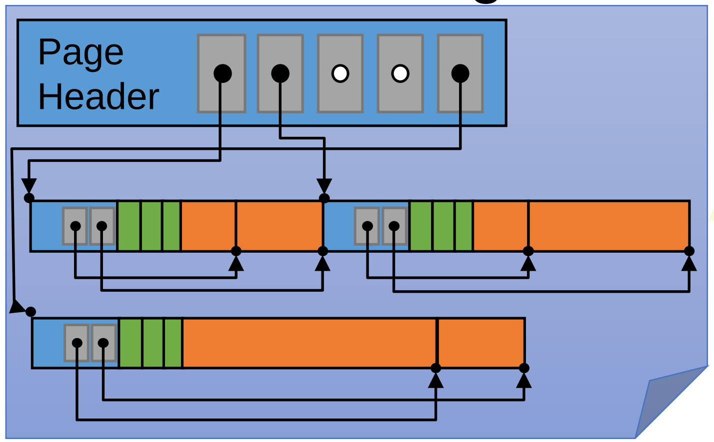
Record



Byte Rep. Record



Slotted Page



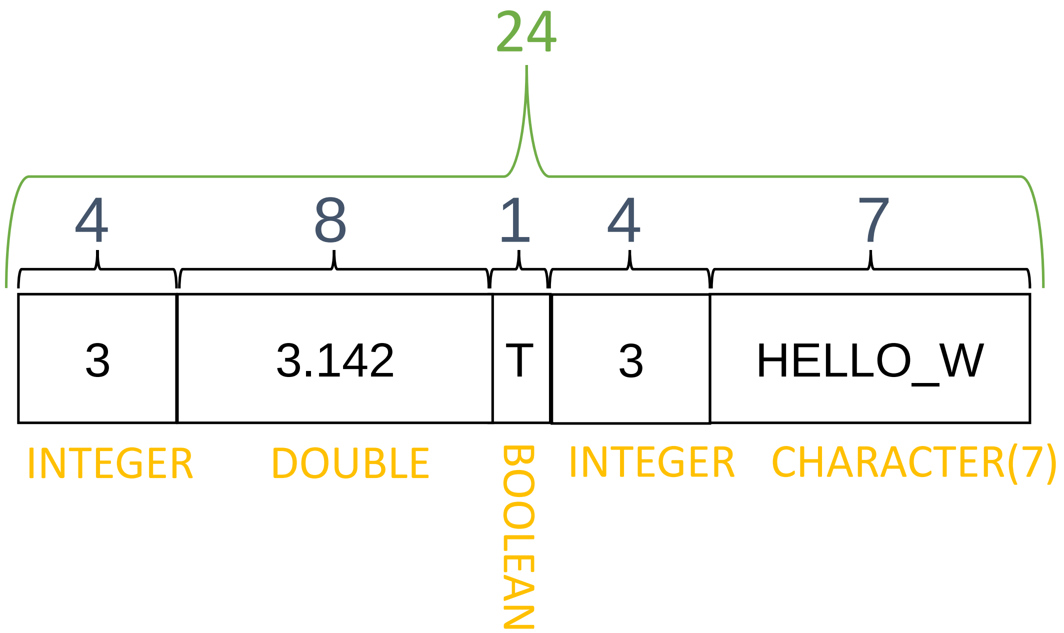
Heap File

Record Formats

- Relational Model →
  - Each record in table has same fixed type
- Assume *System Catalog* with *Schema*
  - No need to store type information (save space!)
  - This will be another table … (bootstraping)
- Goals:
  - Compact in memory & disk format
  - Fast access to fields (why?)

- Easy Case: *Fixed Length Fields*
- Interesting Case: *Variable Length Fields*

Record Formats: Fixed Length



- Field types same for all records in a file.
  - Type info stored separately in *system catalog*
- On disk byte representation same as in memory
- Finding *i'th* field?
  - done via arithmetic (fast)
- Compact? (Nulls?)

Record Formats: Variable Length

What happens if fields are variable length?

Record

Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

Could store with padding? (Fixed Length)

Wasted Space

Bob	Big, St.	M	32	94703
CHAR(20)	CHAR(18)	CHAR	INT	INT

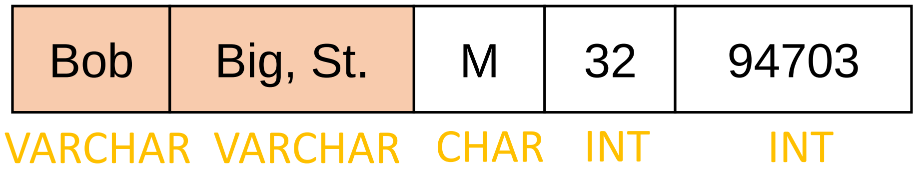
Field Not Big Enough

Alice	Boulevard of the Allies		32	94703
CHAR(20)	CHAR(18)	CHAR	INT	INT

Record Formats: Variable Length

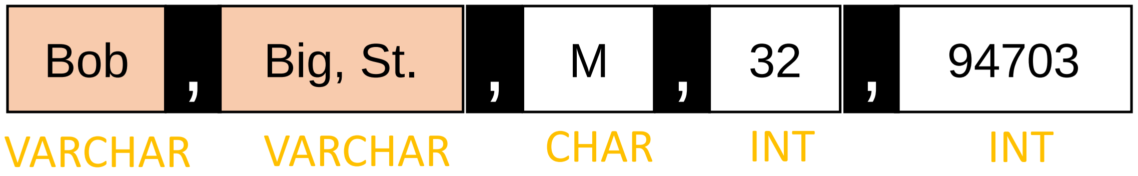
What happens if fields are variable length?

Record



Could use delimiters (i.e., CSV):

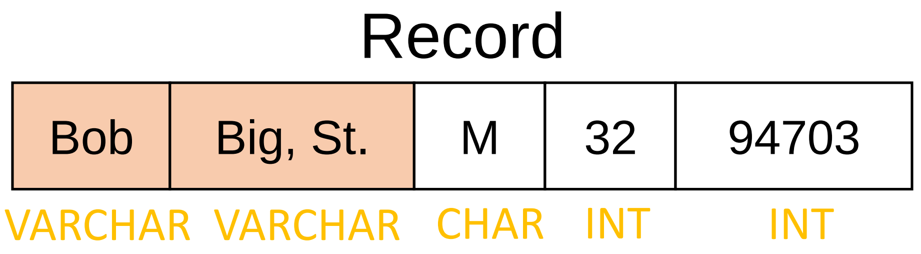
Comma Separated Values (CSV)



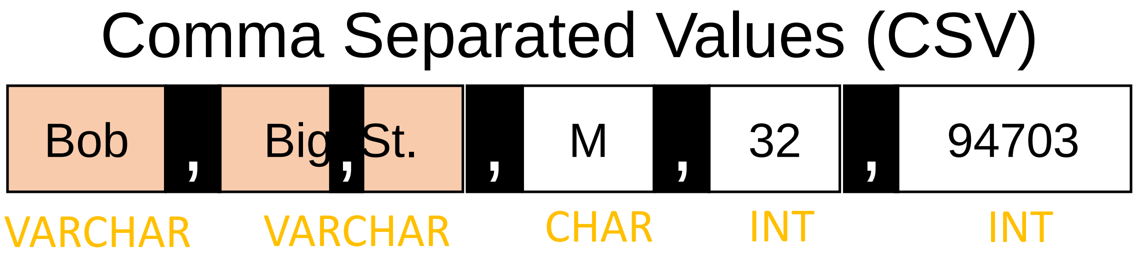
- Issues?

Record Formats: Variable Length

What happens if fields are variable length?



Could use delimiters (i.e., CSV):



- Requires scan to access field
- What if text contains commas?

Record Formats: Variable Length

What happens if fields are variable length?

Record

Bob	Big, St.	M	32	94703
VARCHAR	VARCHAR	CHAR	INT	INT

Store length information before fields:

Variable Length Fields with Offsets

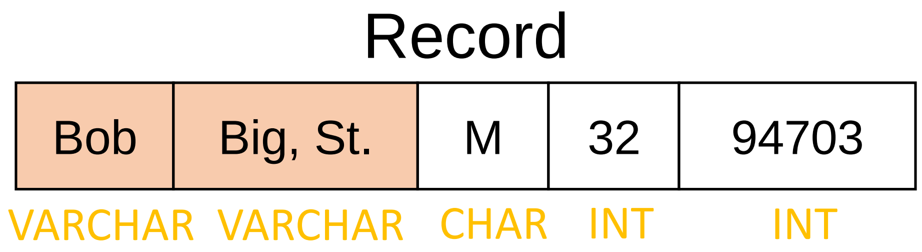
M	32	94703	3	Bob	8	Big, St.
CHAR	INT	INT		VARCHAR		VARCHAR

Move all variable length fields to end
→enable fast access

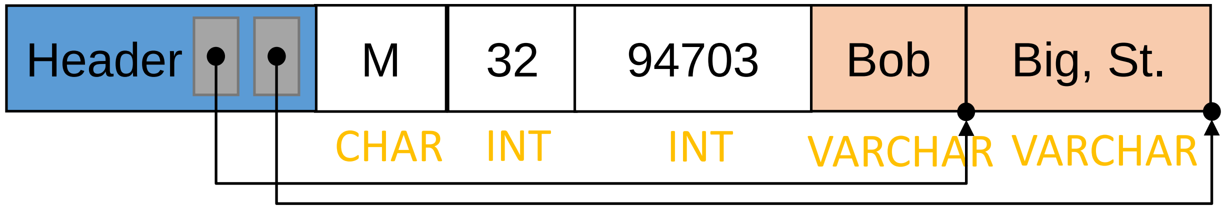
- Requires scan to access field
- ~~What if text contains commas?~~

Record Formats: Variable Length

What happens if fields are variable length?



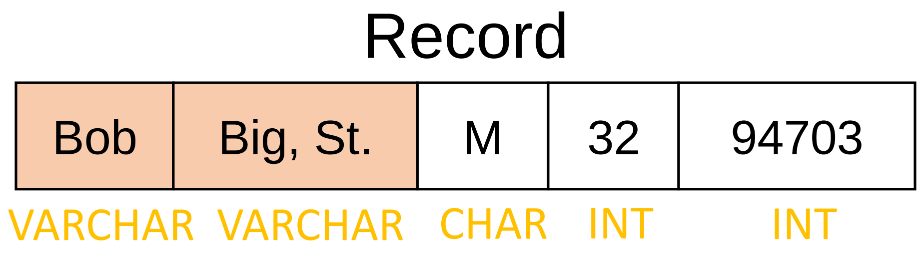
Introduce a record header:



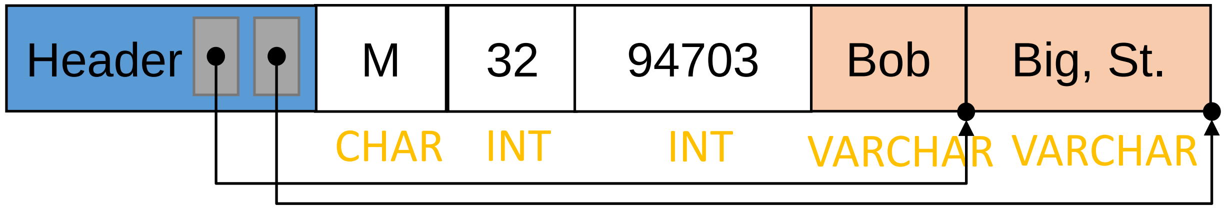
- ~~Requires scan to access field. Why?~~
- ~~What if text contains commas?~~

Record Formats: Variable Length

What happens if fields are variable length?



Introduce a record header:



- Direct access & no "escaping", other adv.?
  - Handle null fields → useful for fixed length

Overview

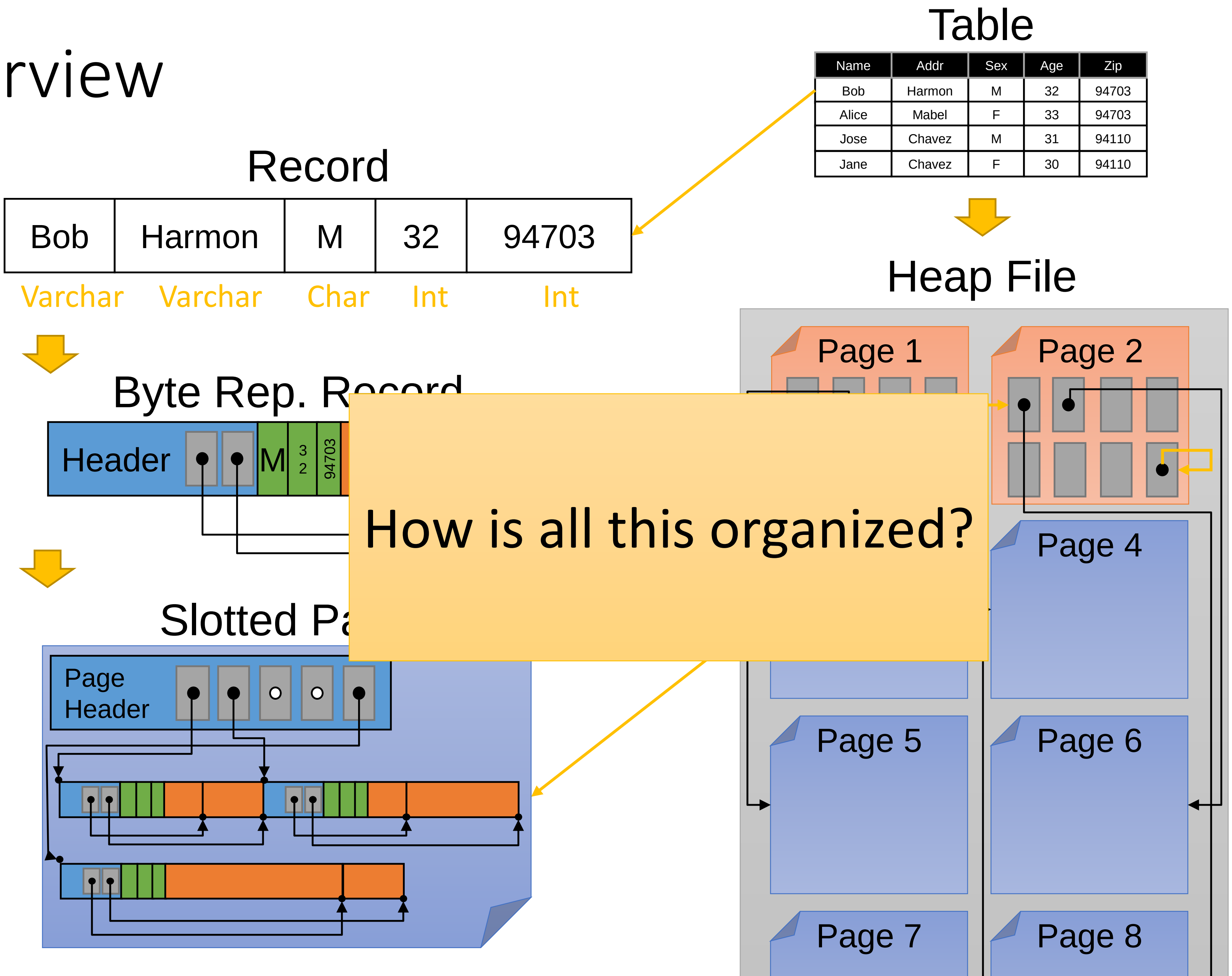


Table
Name
Addr
Sex
Age
Zip
Bob
Harmon
M
32
94703
Alice
Mabel
F
33
94703
Jose
Chavez
M
31
94110
Jane
Chavez
F
30
94110
Record
Bob
Harmon
M
32
94703
Varchar
Varchar
Char
Int
Int
Byte Rep. Record
Header
M
32
94703
Slotted Pa
Page Header
Heap File
Page 1
Page 2
Page 4
Page 5
Page 6
Page 7
Page 8
How is all this organized?

System Catalogs

- For each relation:
  - name, file location, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

⌧ *Catalogs are themselves stored as relations*!

sqlite_master

SELECT name, rootpage FROM **sqlite_master**
WHERE type='table'
ORDER BY name;

ChinookDatabase1 — sqlite3 Chinook_Sqlite.sqlite — sqlite3 — sqlite3 Chinook_Sqlite.sqlite — 71×21

```
sqlite>
sqlite>
sqlite>
sqlite>
sqlite> SELECT name, rootpage FROM sqlite_master
   ...> WHERE type='table'
   ...> ORDER BY name;
name        rootpage
----------  ----------
Album       2
Artist      3
Customer    4
Employee    7
Genre       9
Invoice     10
InvoiceLin  12
MediaType   14
Playlist    15
PlaylistTr  16
Track       19
sqlite>
```

Summary

- Disk manager loads and stores pages
  - Block level reasoning
  - Abstracts device and file system; provides fast next
- Buffer manager brings pages into RAM
  - page pinned while reading/writing
  - dirty pages written to disk
  - good *replacement policy* essential for performance
- DBMS "File" tracks collection of pages, records within each.
  - Heap-files: unordered records organized with directories

Summary (Contd.)

- Slotted page format
  - Variable length records and intra-page reorg

- Variable length record format
  - Direct access to i' th field and null values.

- Catalog relations store information about relations, indexes and views.