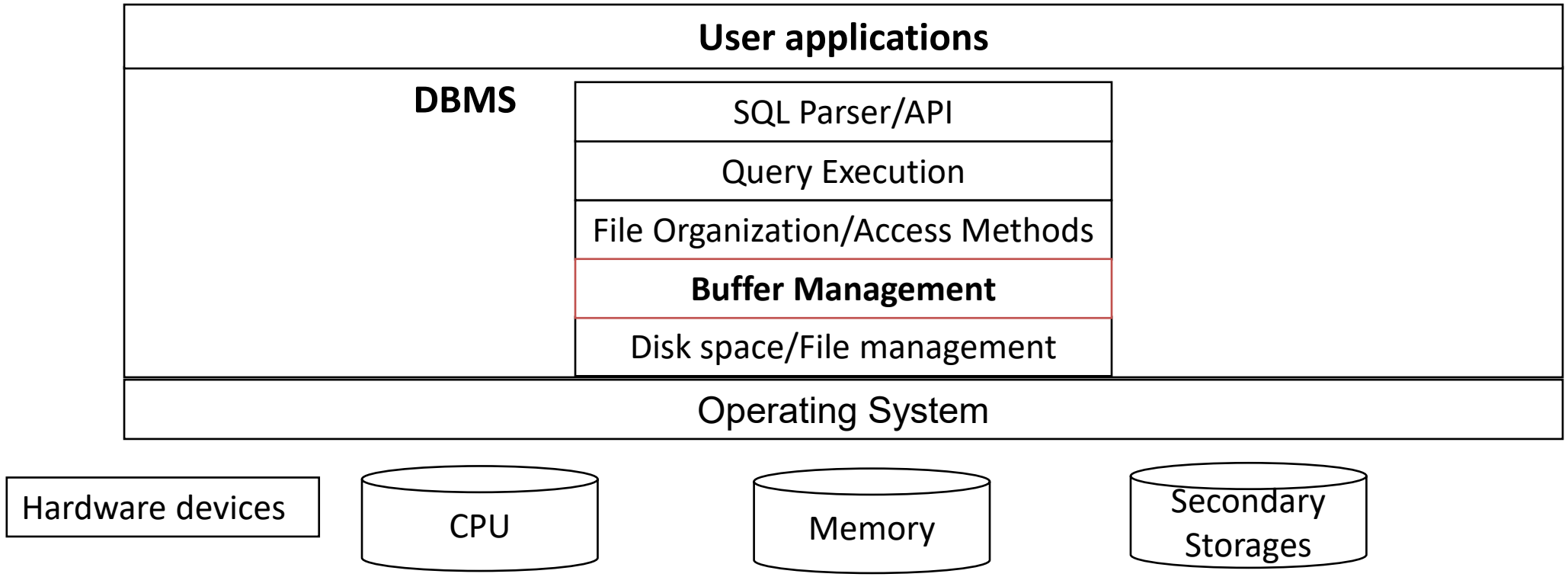
CSE462/562: Database Systems (Spring 24)

Lecture 2: Buffer Management
Data Storage Layout
2/5/2024

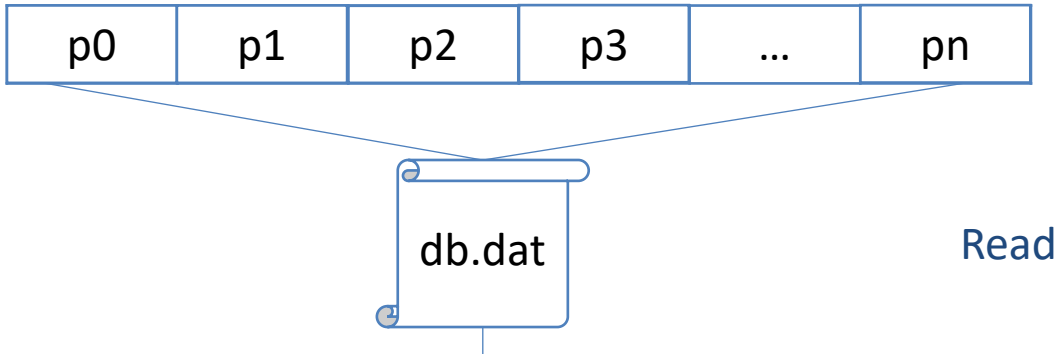
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Big Picture


User applications
DBMS
SQL Parser/API
Query Execution
File Organization/Access Methods
Buffer Management
Disk space/File management
Operating System
Hardware devices
CPU
Memory
Secondary Storages

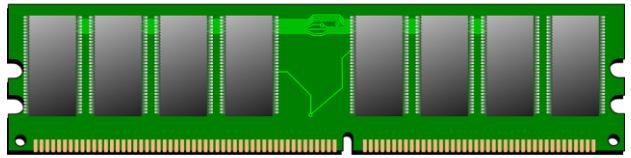
How does database access data pages?

- Data pages are stored in disk file
  - suppose we want to count how many rows there are in a database table
    - need to scan all pages
  - page must be loaded into memory before any computation happens



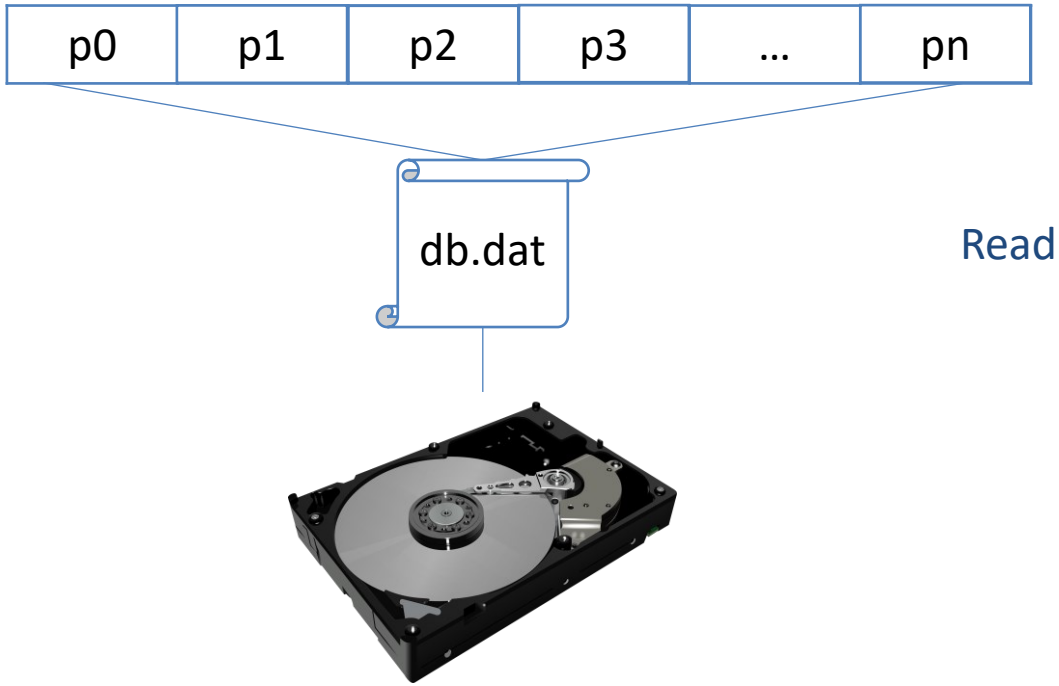
Read: ~10 ms

Computation: < 1 μs



How does database access data pages?

- Data pages are stored in disk file
  - suppose we want to count how many rows there are in a database table
    - need to scan all pages
  - page must be loaded into memory before any computation happens

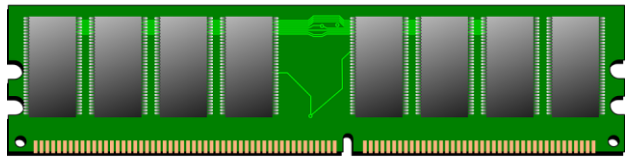


Read: ~10 ms

Computation: < 1 μs

- Repeat for all the n pages
- Execution time dominated by I/O

p0



How does database access data pages?

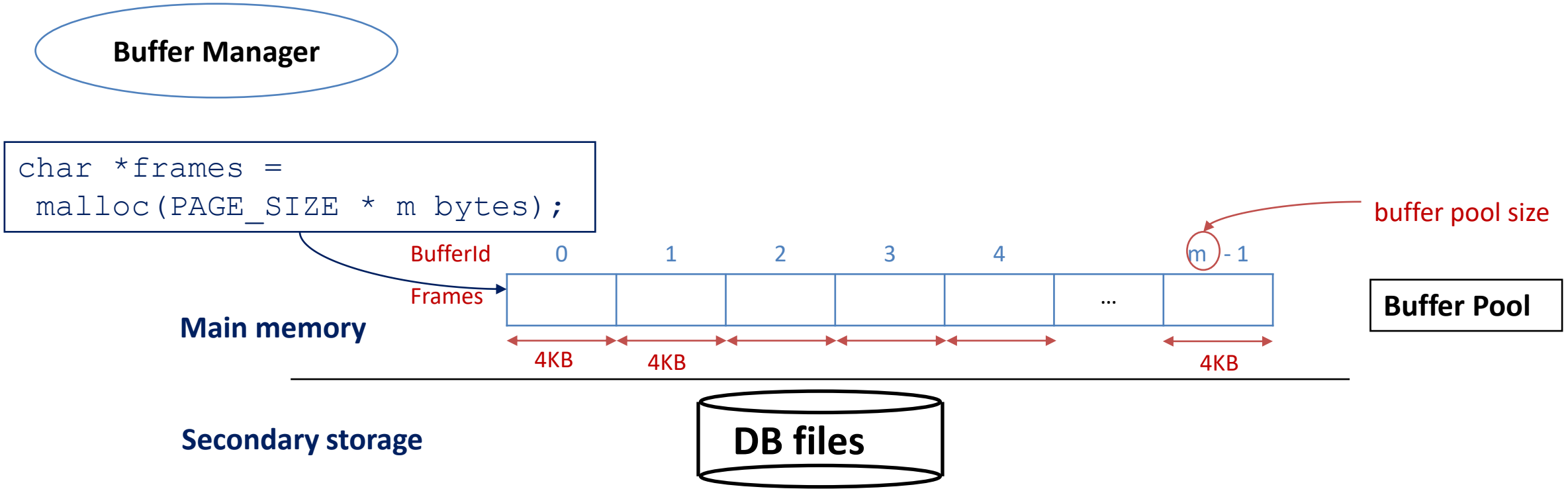
- Data pages are stored in disk file
  - suppose we want to count how many rows there are in a database table
    - need to scan all pages
  - page must be loaded into memory before any computation happens

- What if we want to scan the data file for multiple passes?
  - Option 1: read/write the entire page on demand before reading/writing the integer <- very slow
  - Option 2: read all data pages into memory at the beginning <- not scalable
    - May not fit in memory
    - What to do on modify?
      - Immediately write back? Or Flush when program shutsdown?
      - Data persistence?

- Solution: buffer pool

Buffer management in DBMS

- Buffer manager manages a fixed-size pool of in-memory page frames which
  - are of the same size as the data pages (e.g., 4KB)



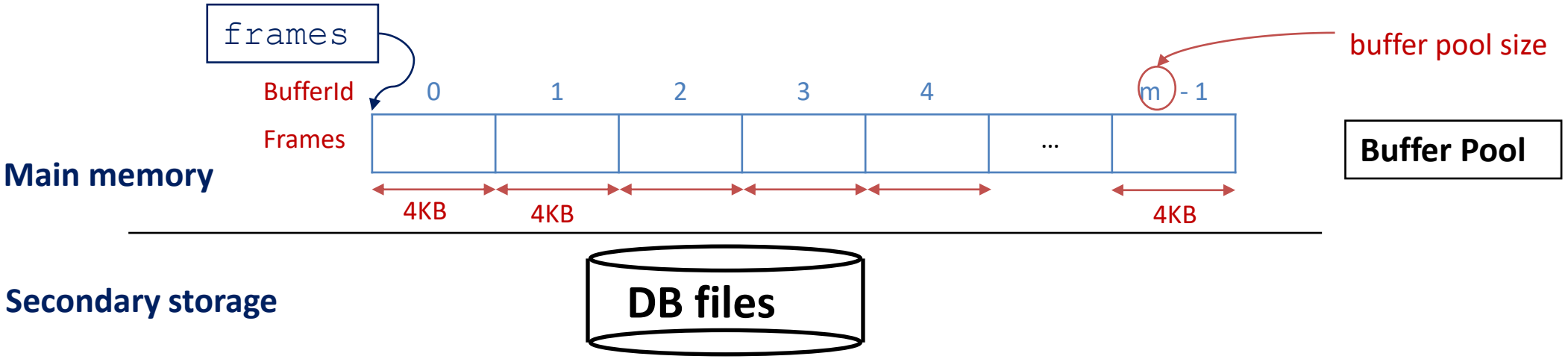
Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

Upper level components

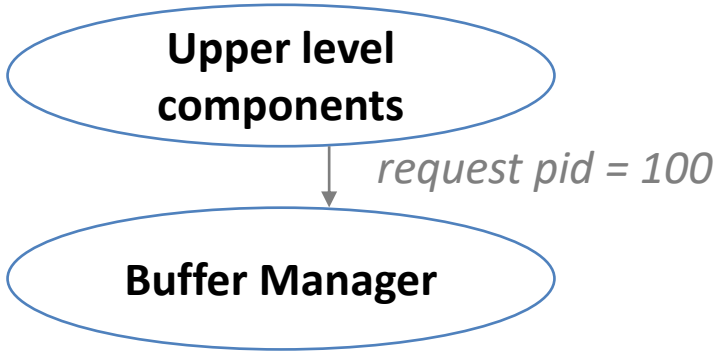
Buffer Manager

```
1 HandlePageRequest(pid):
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

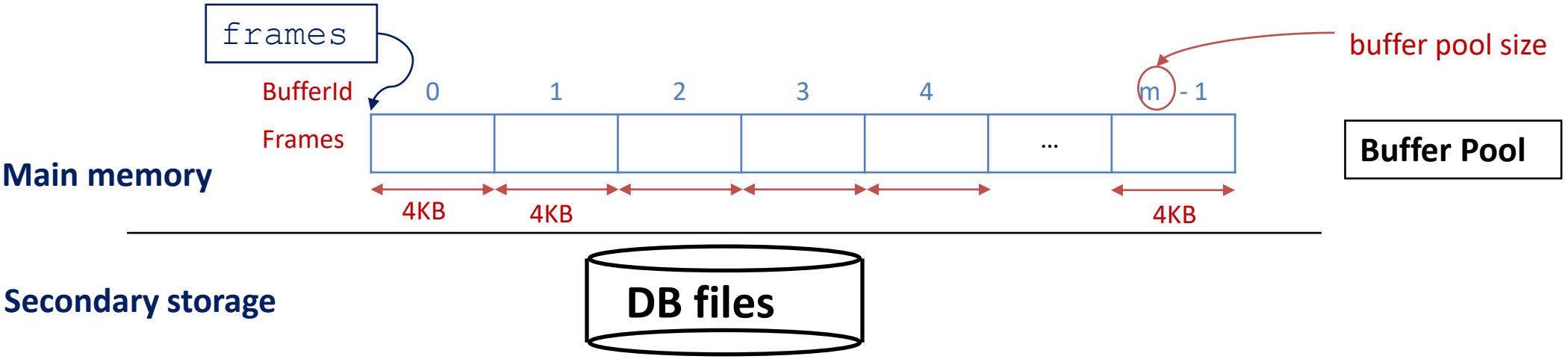


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

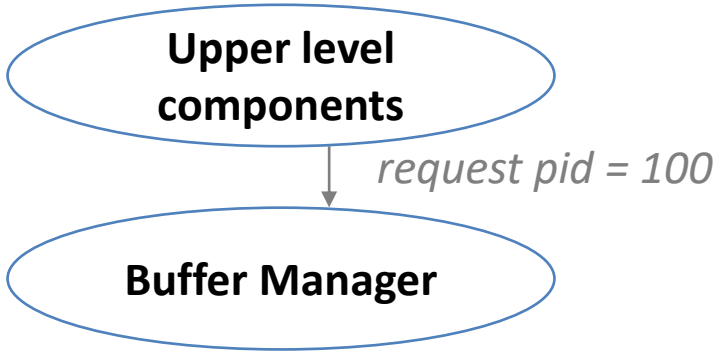



```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

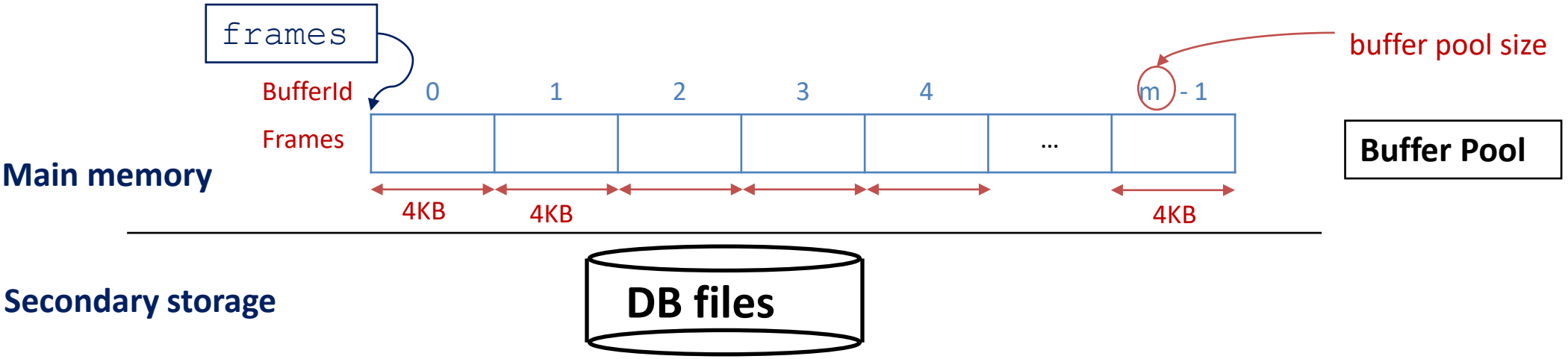


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

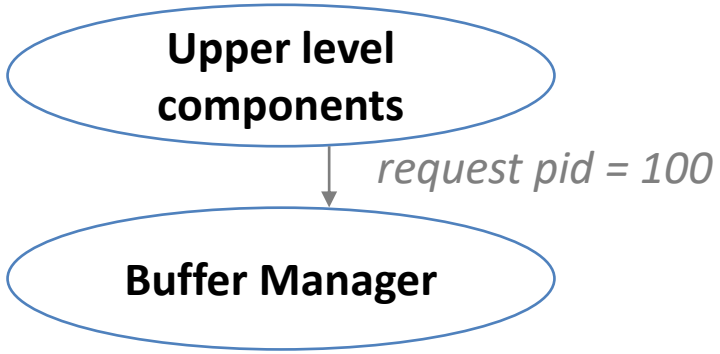



```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

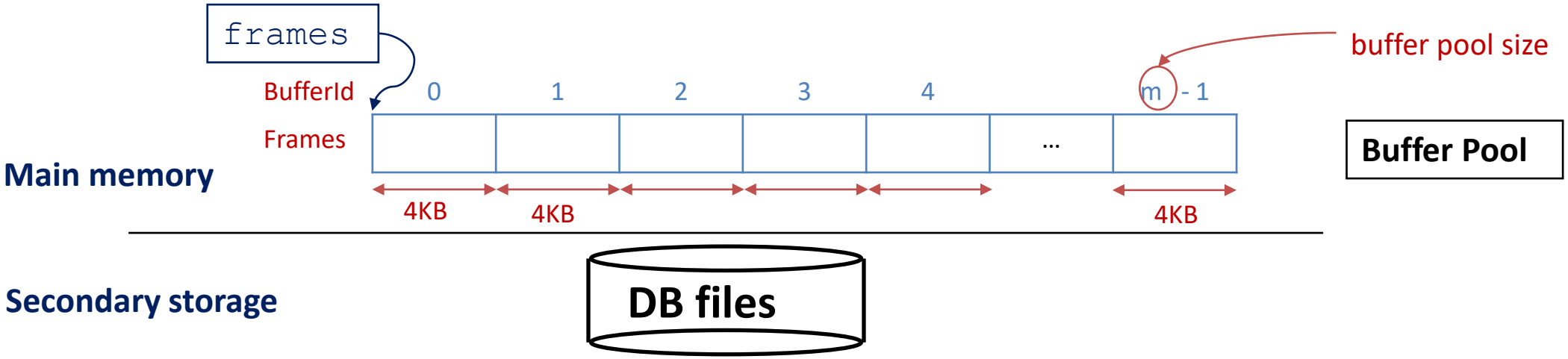


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

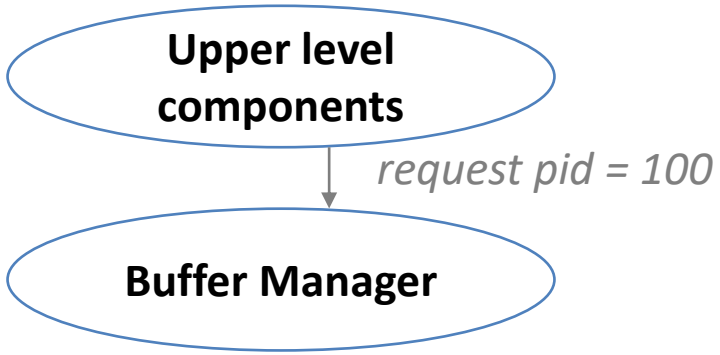



```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i     // i = 0
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

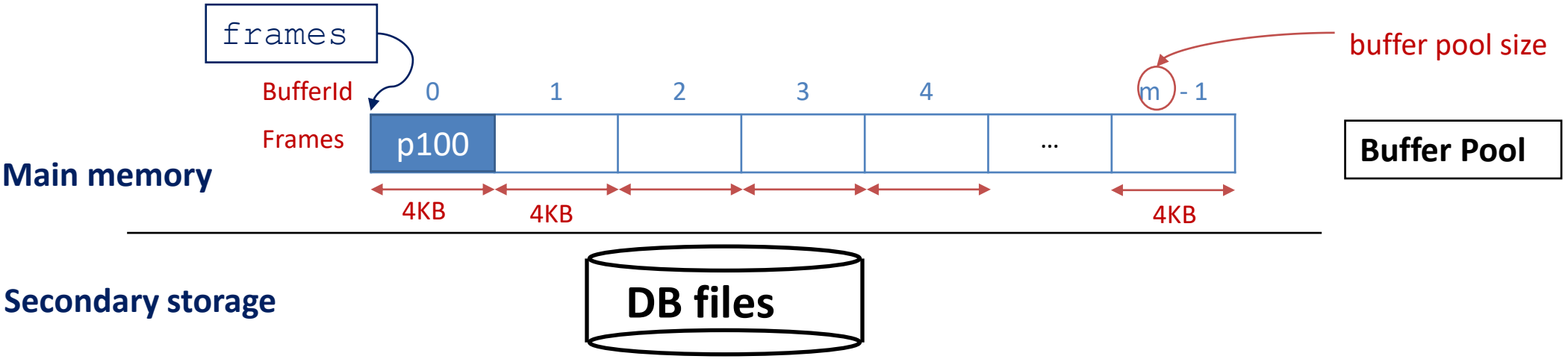


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100



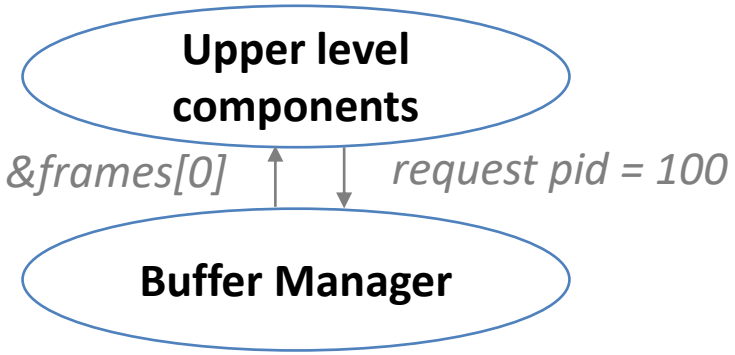

```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i     // i = 0
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```



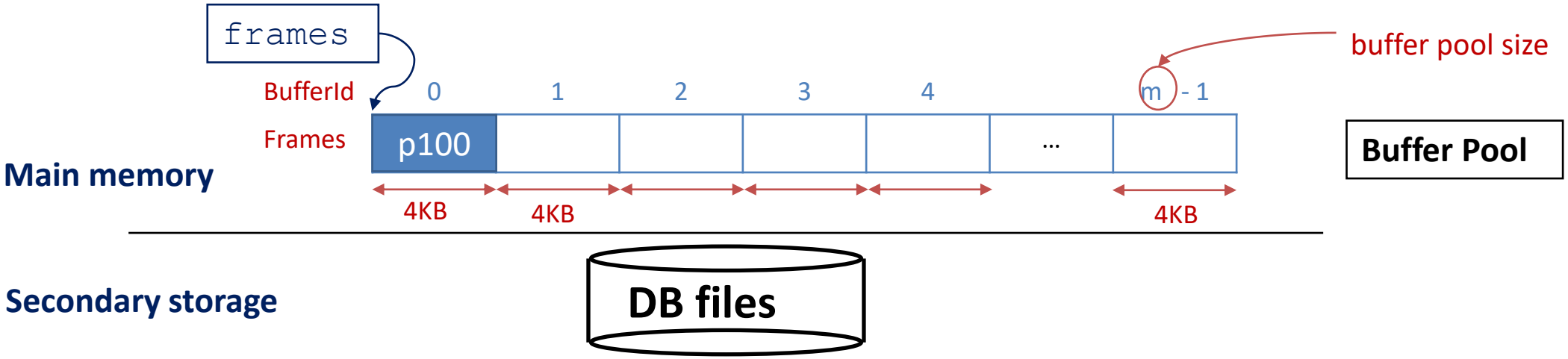
Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

Cost: 1 I/O




```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i     // i = 0
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```



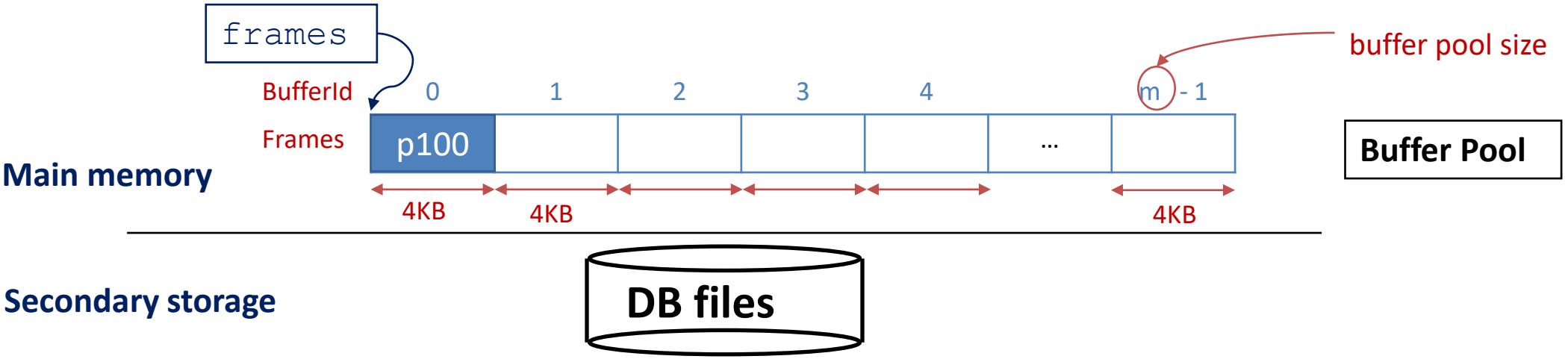
Handling a page request (buffer hit)

- Handling page request
  - Suppose we want to read the same page again (pid = 100)

Upper level components

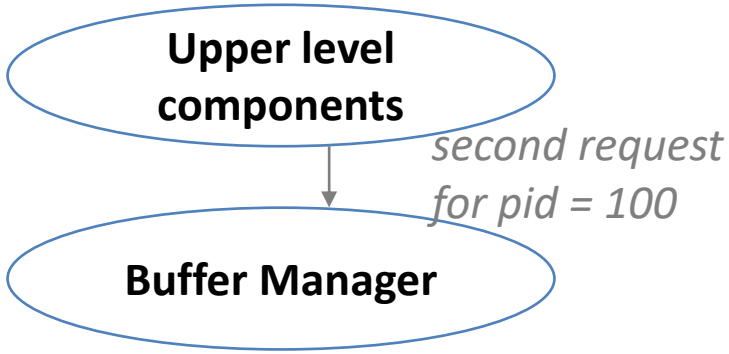
Buffer Manager

```
1 HandlePageRequest(pid):
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

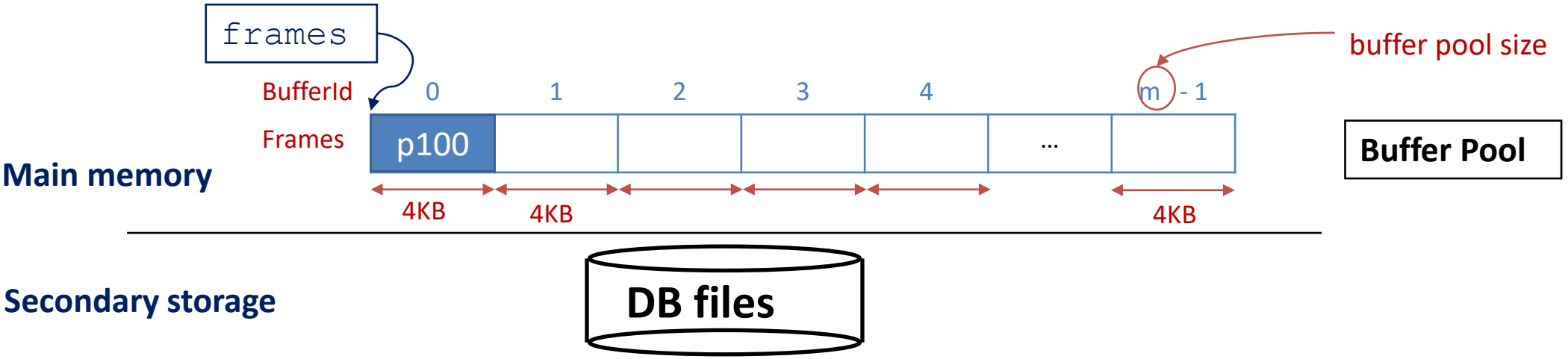


Handling a page request (buffer hit)

- Handling page request
  - Suppose we want to read the same page again (pid = 100)



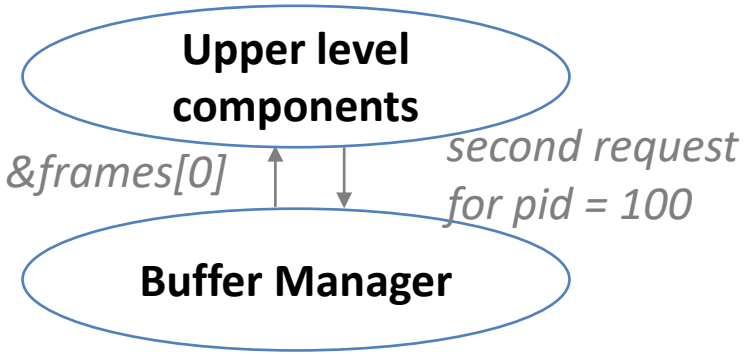

```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```



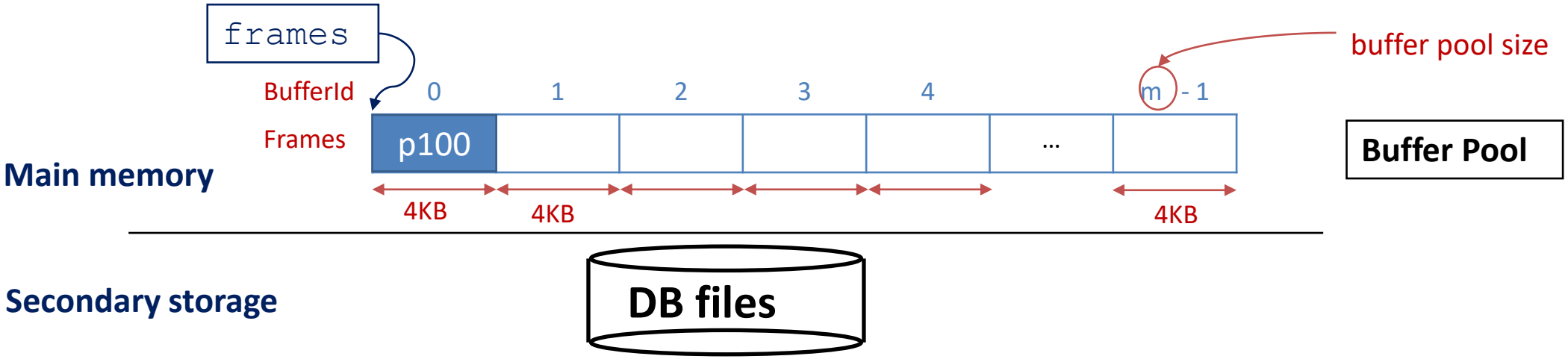
Handling a page request (buffer hit)

- Handling page request
  - Suppose we want to read the same page again (pid = 100)

Cost: 0 I/O




```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]  // i = 0
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

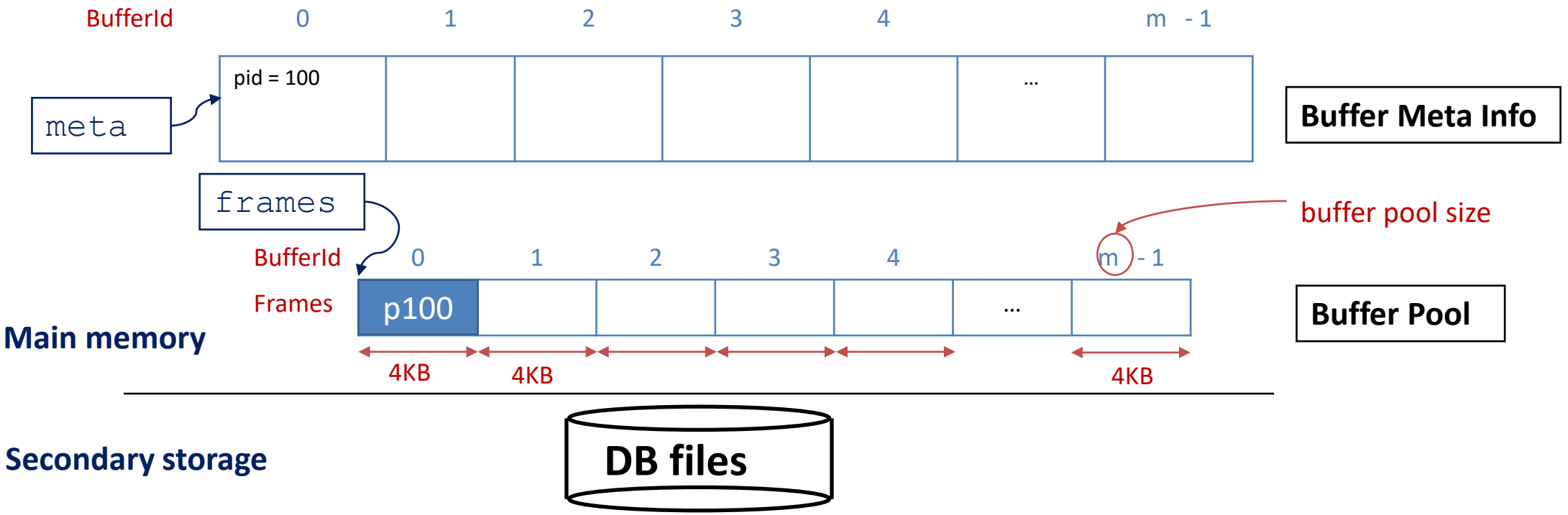


Map page numbers to buffer frames

- How to implement line 2?

```
2  if pid exists in some buffer frame i:
```

  - Need to store the page numbers, but where?
    - For each buffer frame, we maintain a metadata structure which includes **pid**.



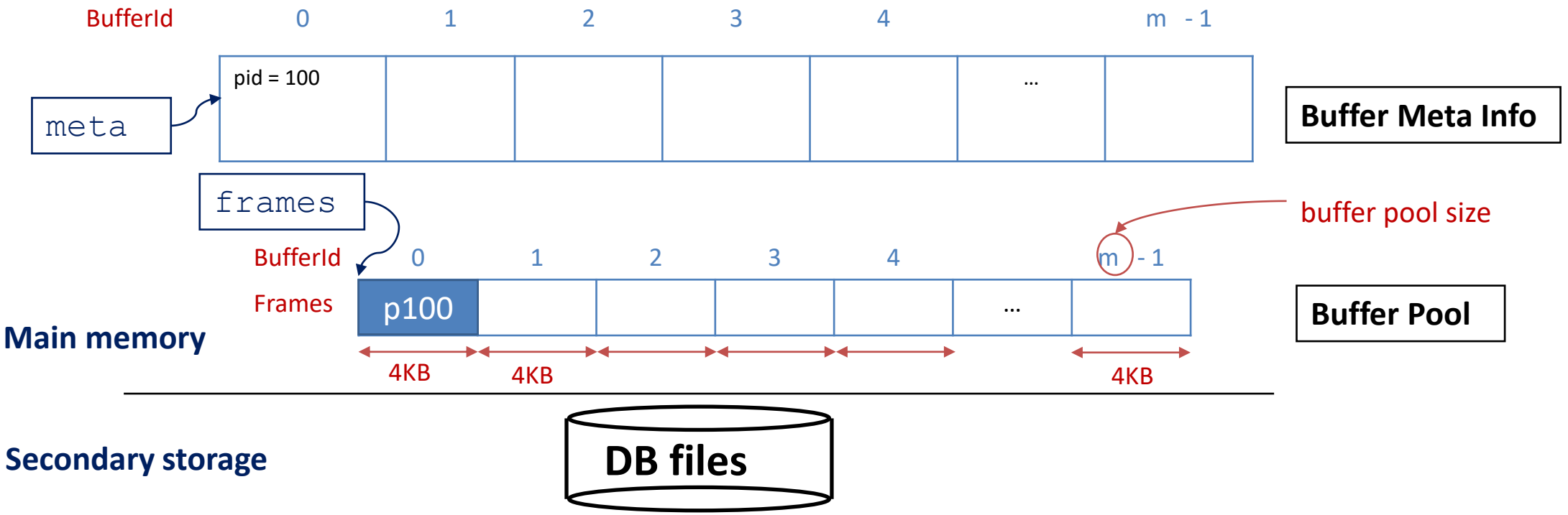
Map page numbers to buffer frames

- How to implement line 2?

```
2  if pid exists in some buffer frame i:
```

```
for (BufferId i = 0; i < m; ++i) {
  if (meta[i].pid == 100)
    return i;
}
return InvalidBufferId;
```

O(m) time -- slow!



Map page numbers to buffer frames

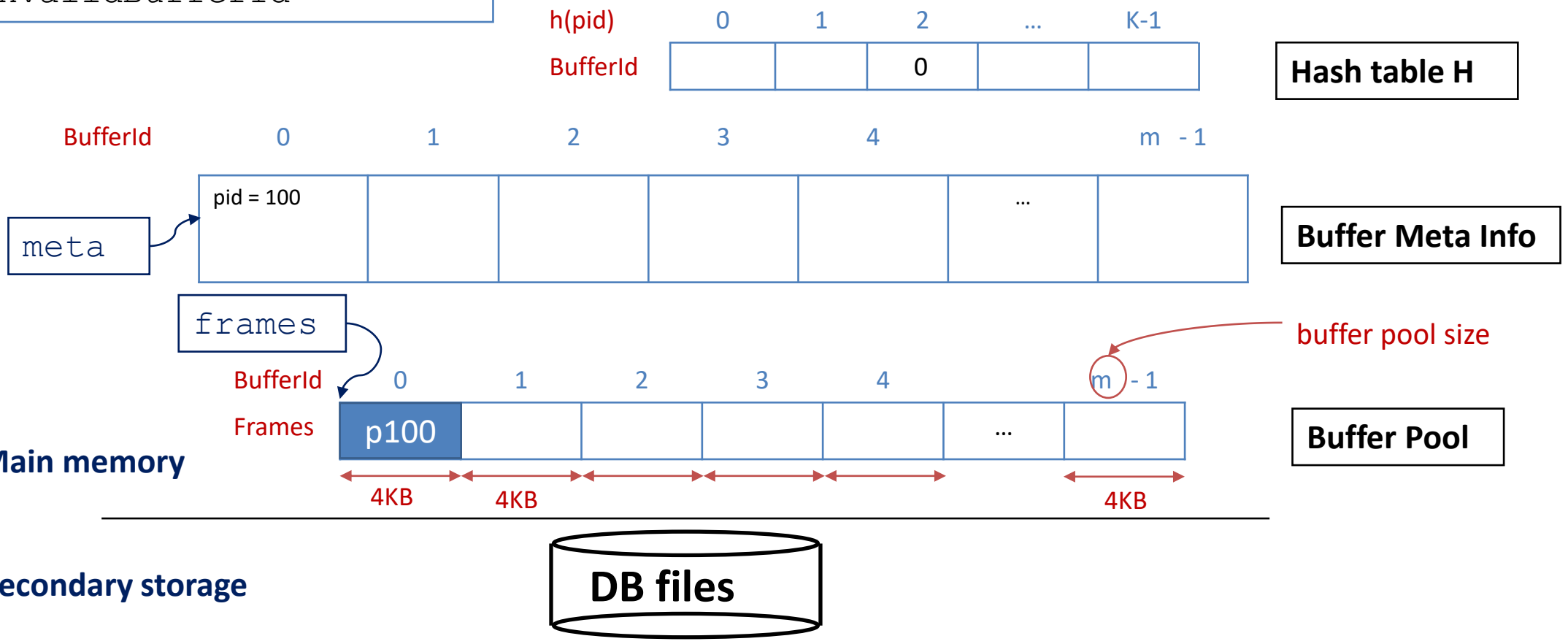
- How to implement line 2?

```
2  if pid exists in some buffer frame i:
```

```
if (H.find(100) != H.end())
  return H[100];
return InvalidBufferId
```

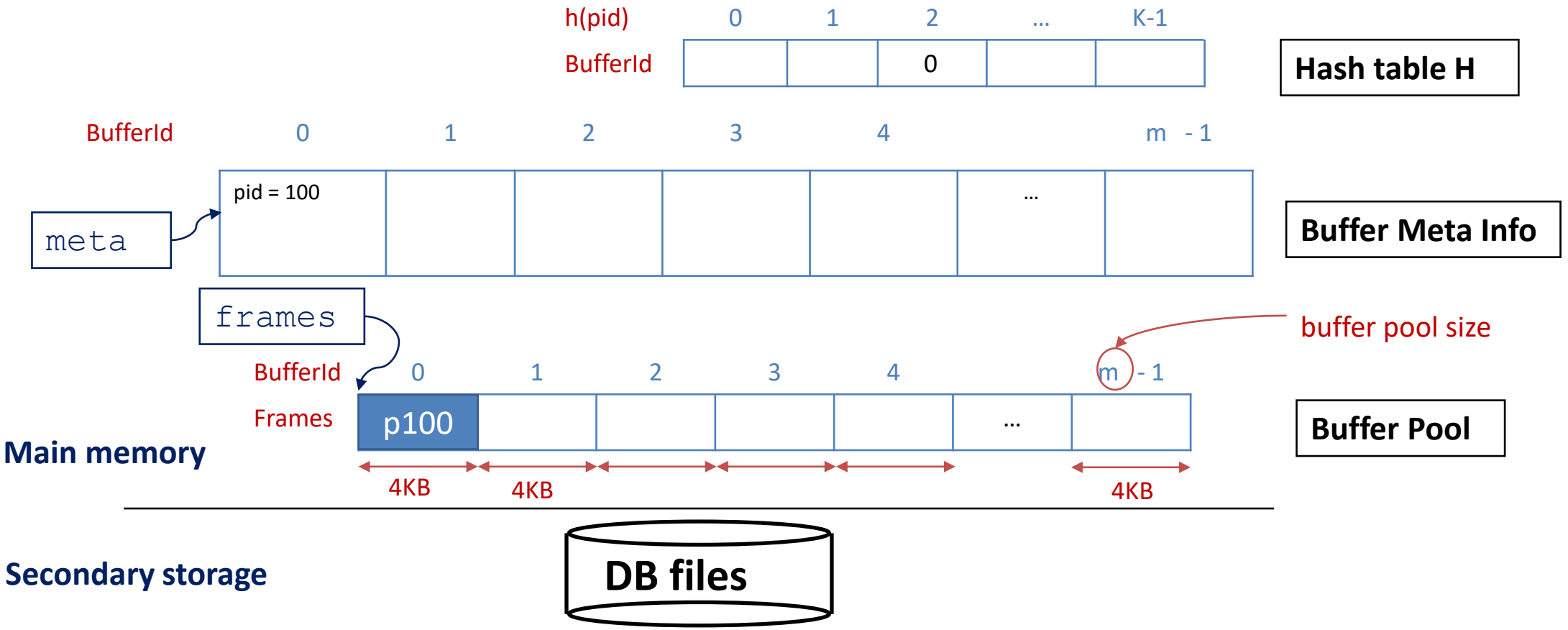
O(1) time in expectation

suppose h(100) == 2



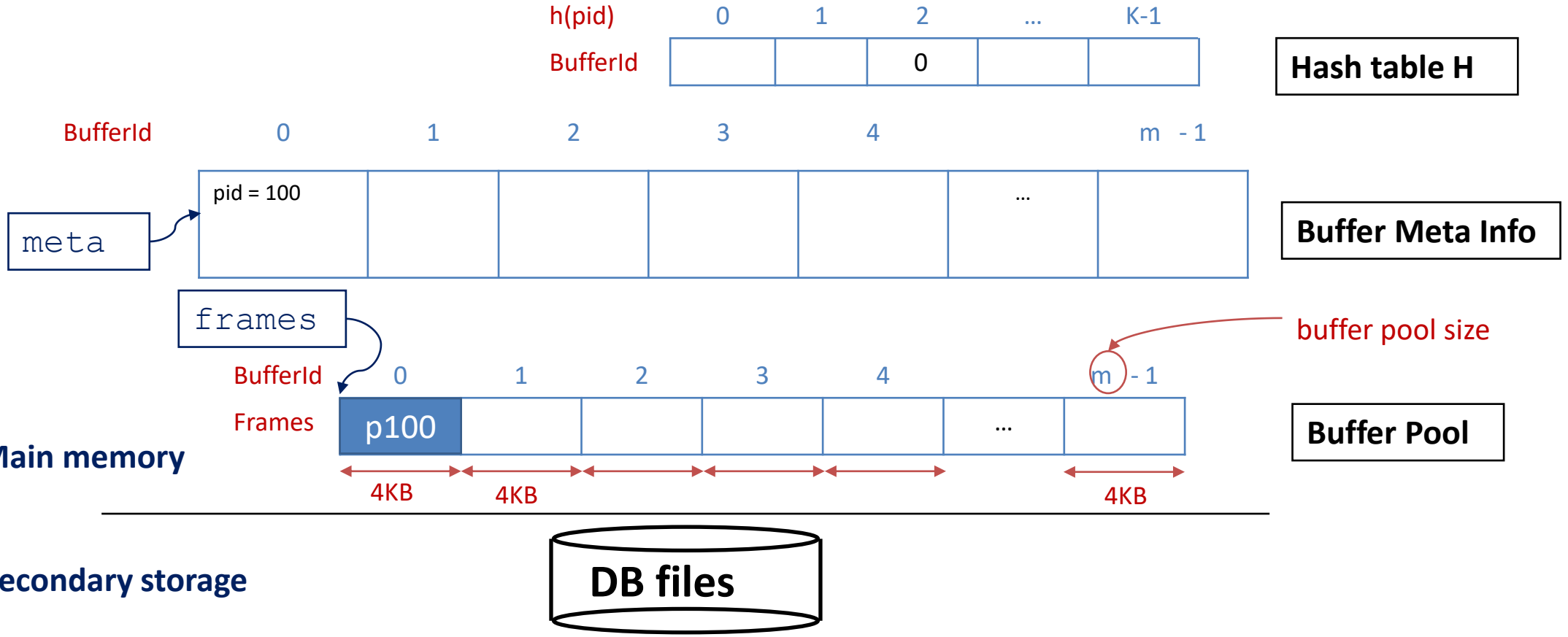
Map page numbers to buffer frames

- Practical consideration for hash tables
  - DBMS usually has its own hash tables implementation for buffer manager -- why?
    - memory constraints, efficiency, concurrency control, …



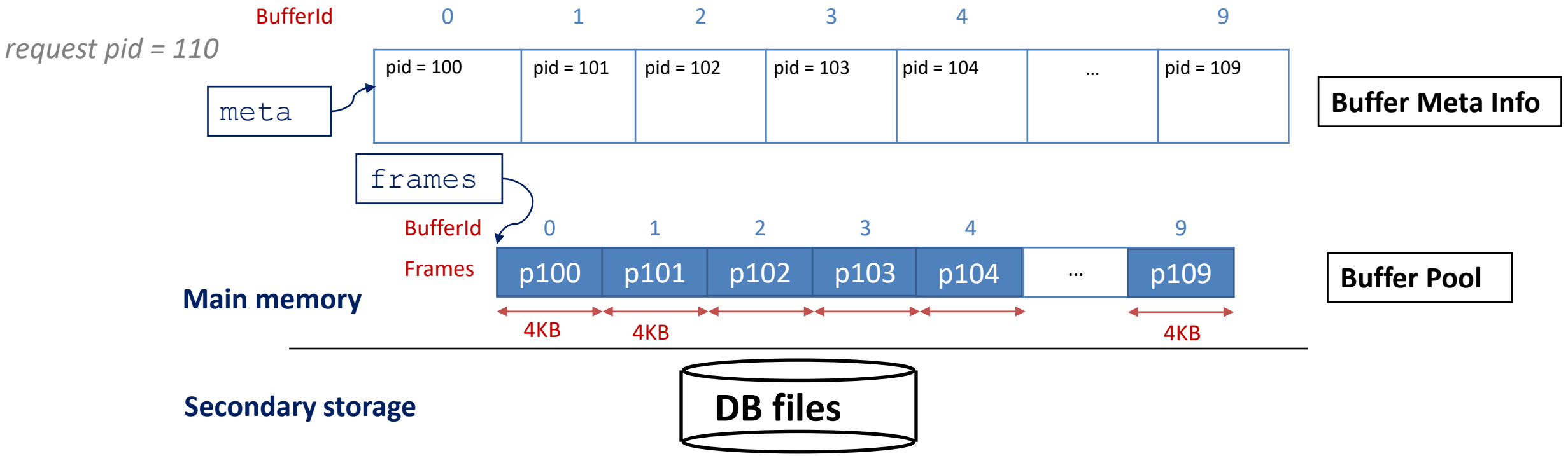
Map page numbers to buffer frames

- Practical consideration for hash tables
  - For Project 2: feel free to use libraries (e.g., `absl::flat_hash_map`)
    - Tips for time and memory efficiency: avoid rehashing
      - Set the initial bucket count `K >= m / max_load_factor()`



Buffer eviction

- What if we run out of buffer frames?
  - e.g., we are scanning a table with N = 100 pages, but buffer pool size m = 10

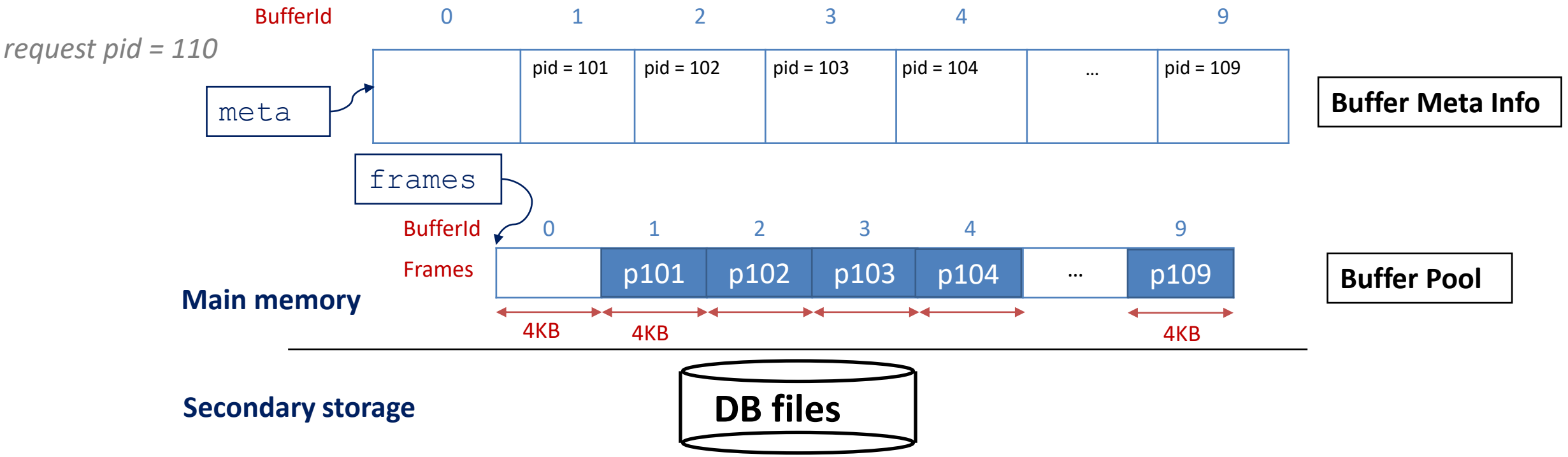


Buffer eviction

- What if we run out of buffer frames?
  - Buffer eviction: choose a victim to remove from the buffer pool
    - Several possible policies (more on this later)

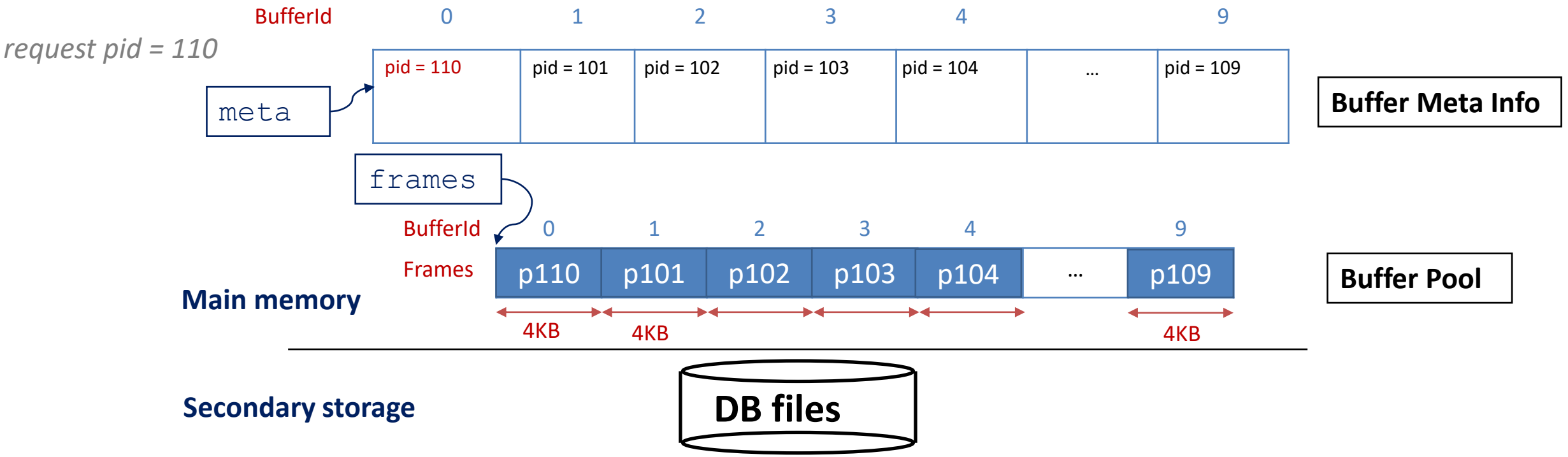
Buffer eviction

- What if we run out of buffer frames?
  - Buffer eviction: choose a victim to remove from the buffer pool
    - Several possible policies (more on this later)



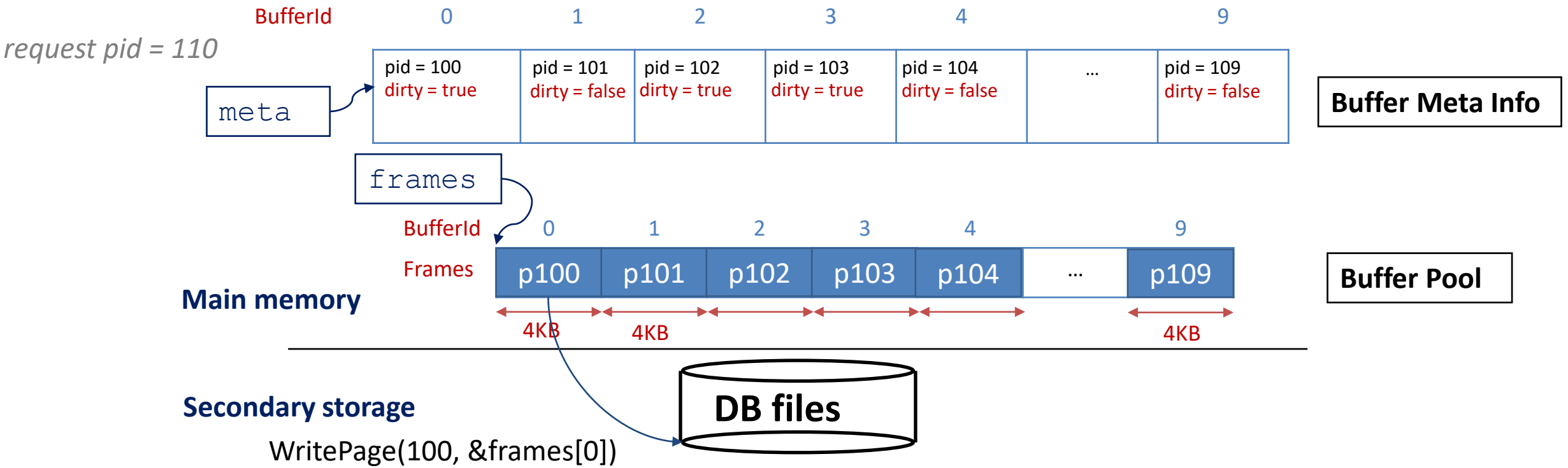
Buffer eviction

- What if we run out of buffer frames?
  - Buffer eviction: choose a victim to remove from the buffer pool
    - Several possible policies (more on this later)



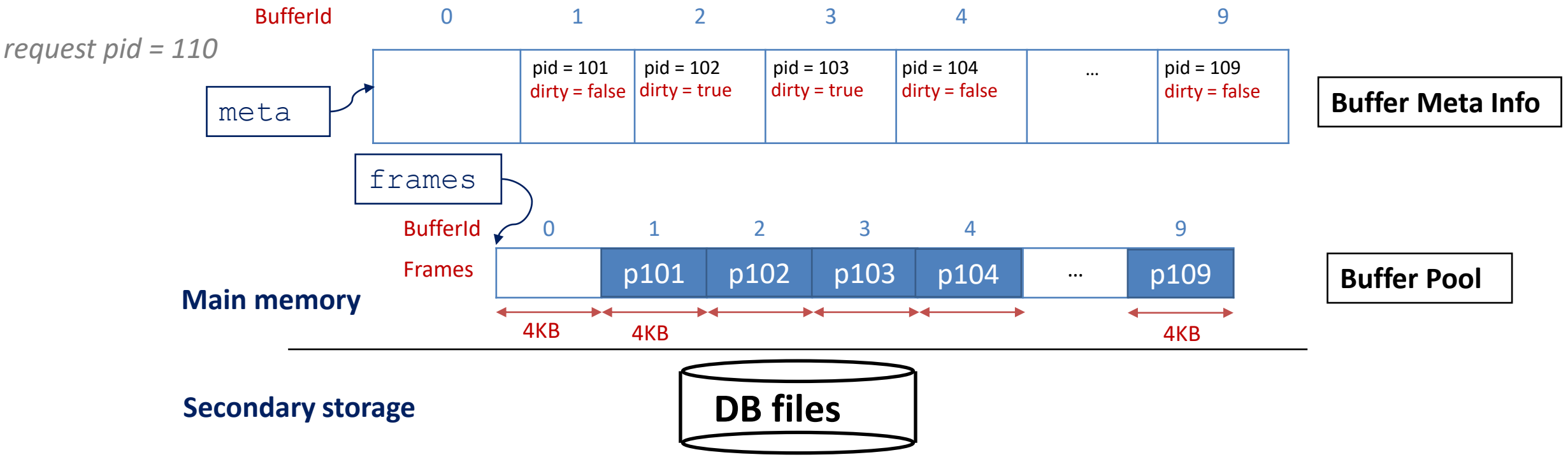
Page requested for writes

- Potential problem with page eviction?
  - What if the evicted page is modified? (e.g., `UPDATE A SET x = x + 10 WHERE …`)
    - We must write modified page back before eviction



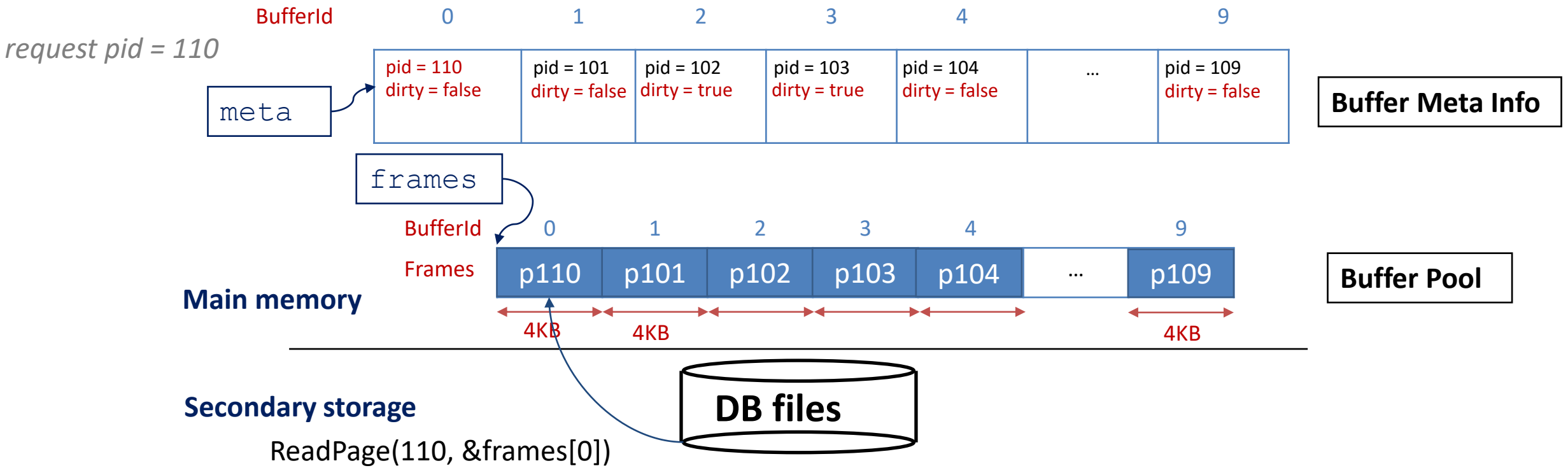
Page requested for writes

- Potential problem with page eviction?
  - What if the evicted page is modified? (e.g., `UPDATE A SET x = x + 10 WHERE …`)
    - We must write modified page back before eviction



Page requested for writes

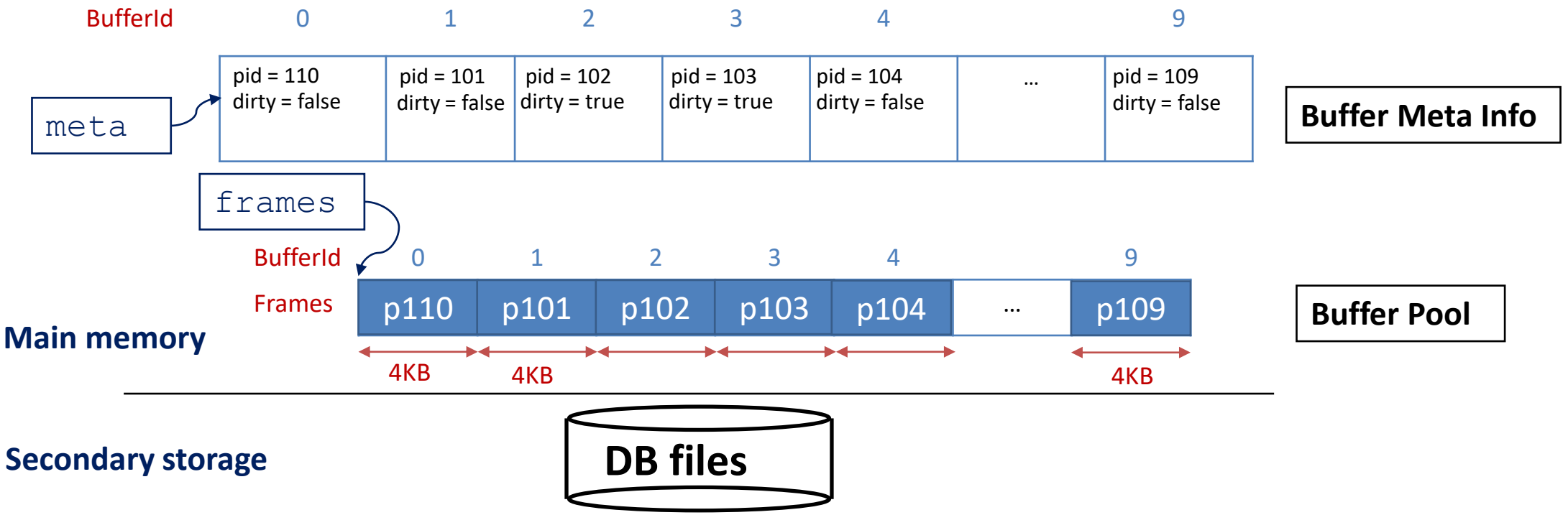
- Potential problem with page eviction?
  - What if the evicted page is modified? (e.g., `UPDATE A SET x = x + 10 WHERE …`)
    - We must write modified page back before eviction



Buffer pins

- Problems with concurrency
  - One thread reading a block while the other tries to evict it

T1: char * frame = BufMgr.HandlePageRequest(110) // &frames[0]

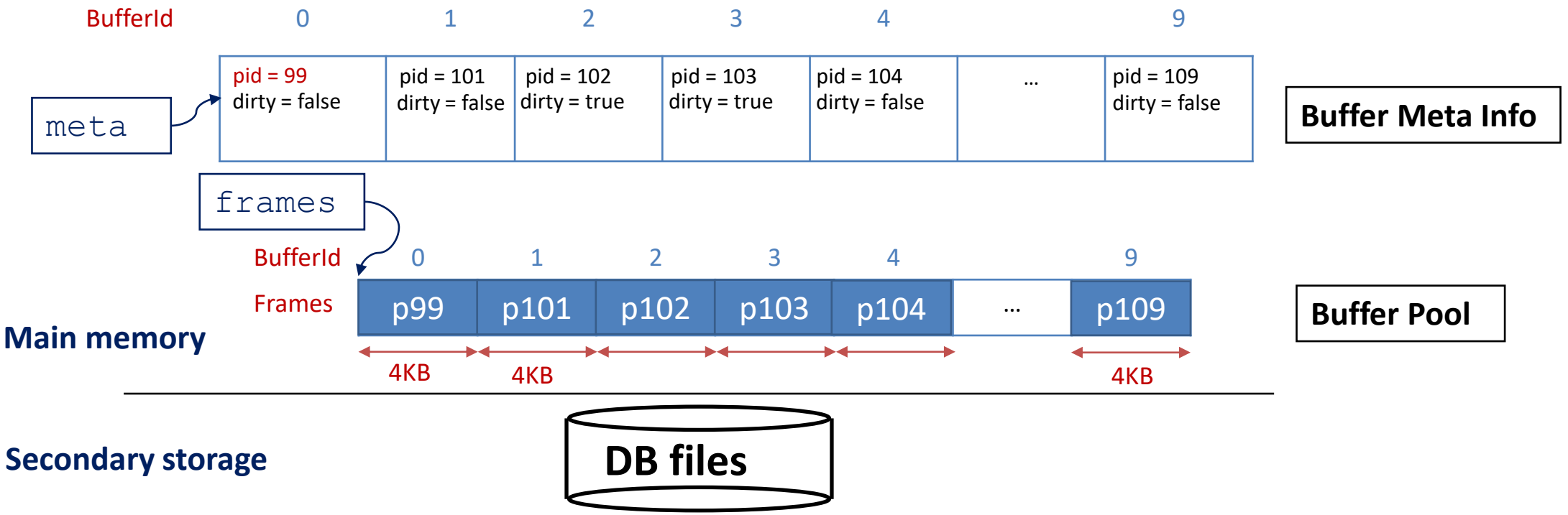


Buffer pins

- Problems with concurrency
  - One thread reading a block while the other tries to evict it

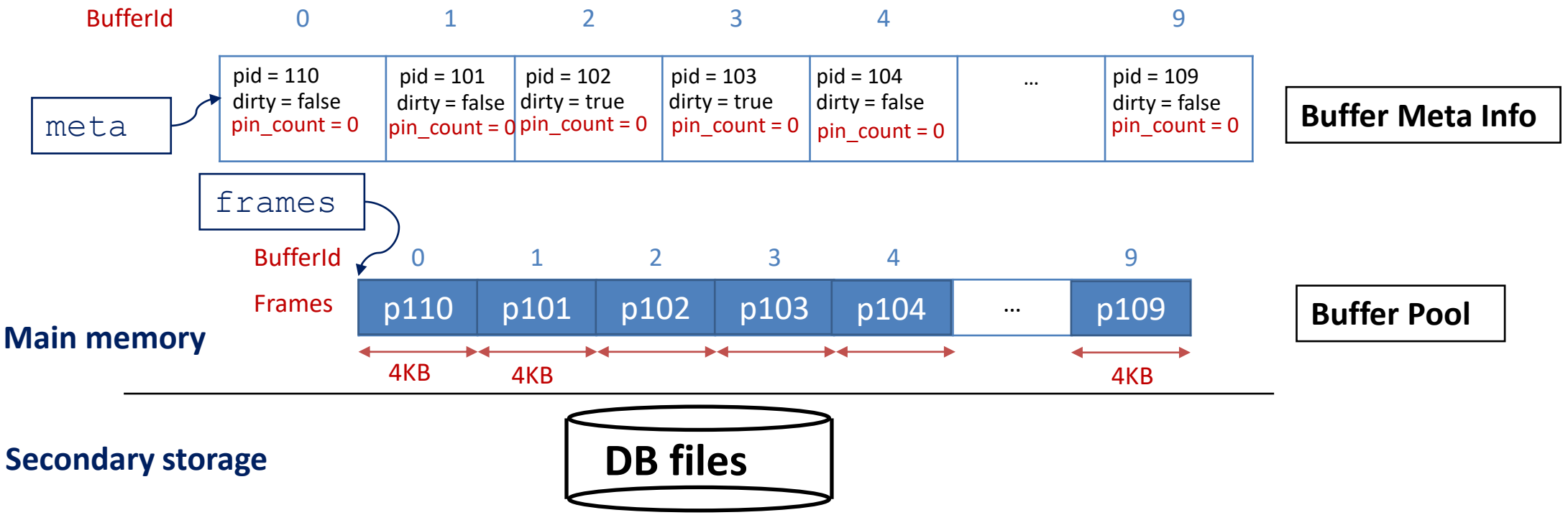
T1: char * f1 = BufMgr.HandlePageRequest(110) // &frames[0] *f1 now contains a wrong page for T1*

T2: char * f2 = BufMgr.HandlePageRequest(99) // &frames[0]



Buffer pins

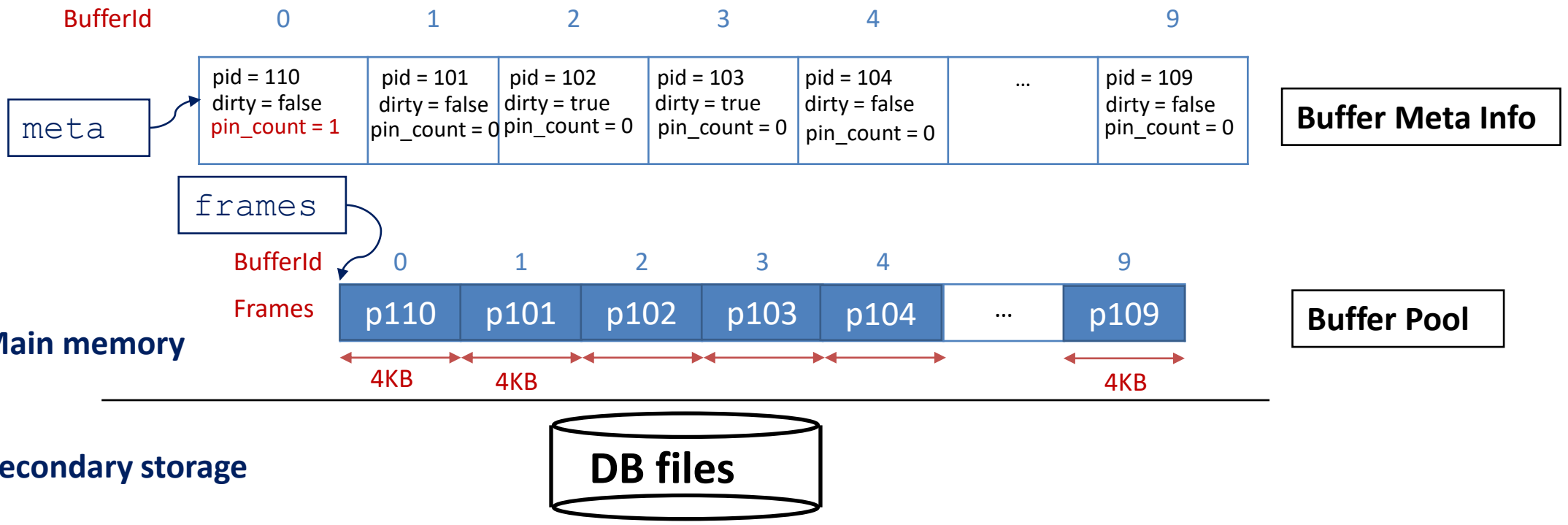
- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0



Buffer pins

- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0

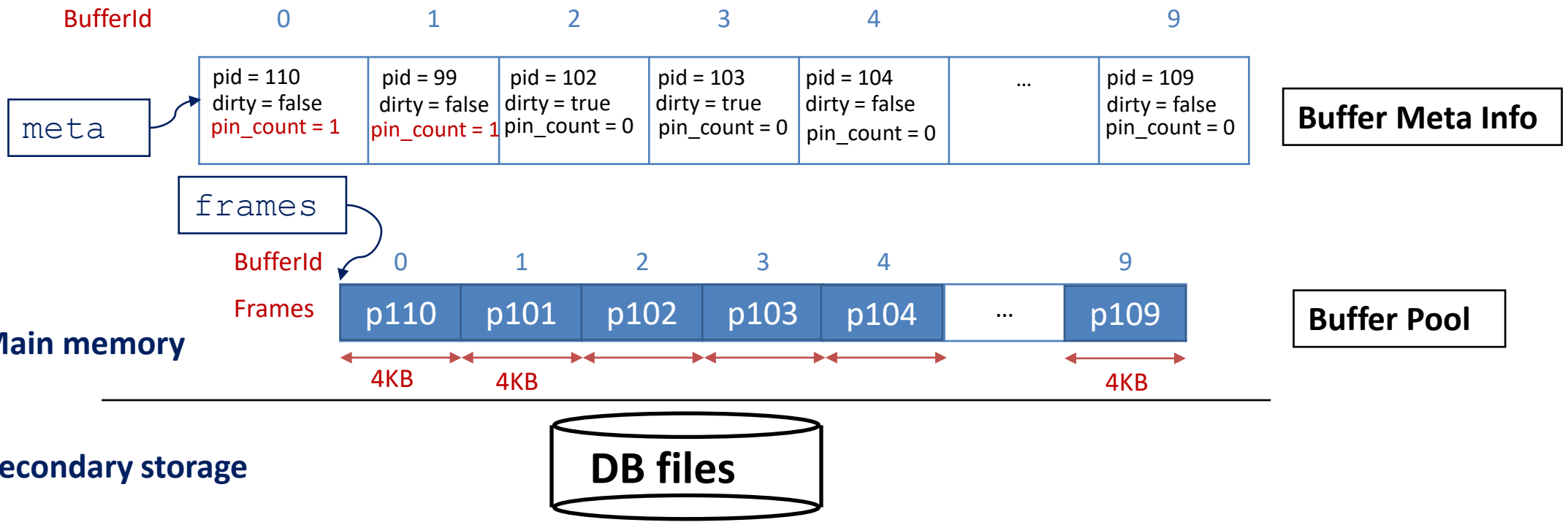
T1: BufferId b1 = BufMgr.PinPage(110, &f1) // b1 = 0



Buffer pins

- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0

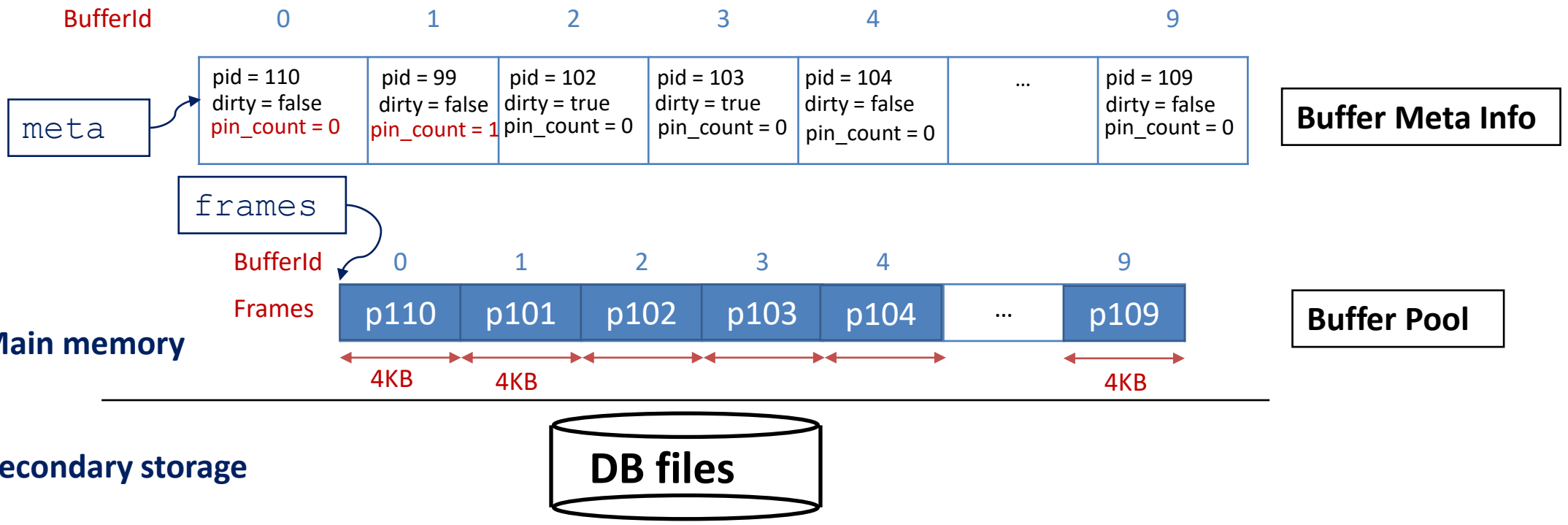
```
T1: BufferId b1 = BufMgr.PinPage(110, &f1)    // b1 = 0
T2: BufferId b2 = BufMgr.PinPage(99, &f2)     // b2 = 1
```



Buffer pins

- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0

```
T1: BufferId b1 = BufMgr.PinPage(110, &f1)      // b1 = 0
T2: BufferId b2 = BufMgr.PinPage(99, &f2)       // b2 = 1
T1: BufMgr.UnpinPage(b1)
```



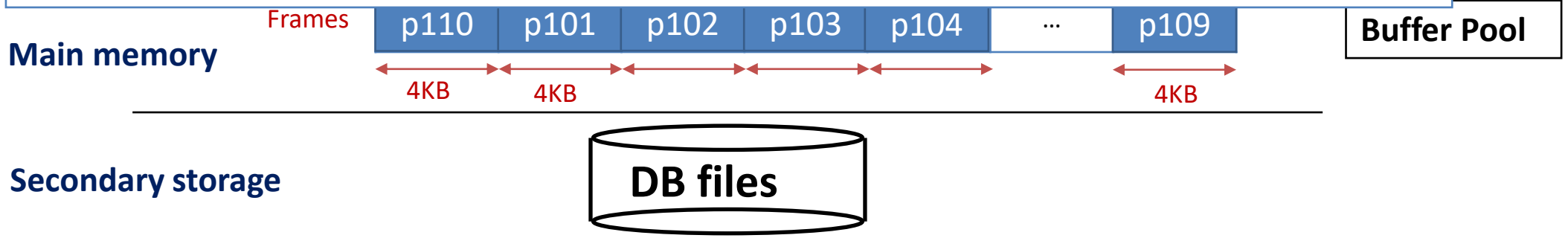
Buffer pins

- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0

```
T1: BufferId b1 = BufMgr.PinPage(110, &f1)      // b1 = 0
T2: BufferId b2 = BufMgr.PinPage(99, &f2)       // b2 = 1
T1: BufMgr.UnpinPage(b1)
```

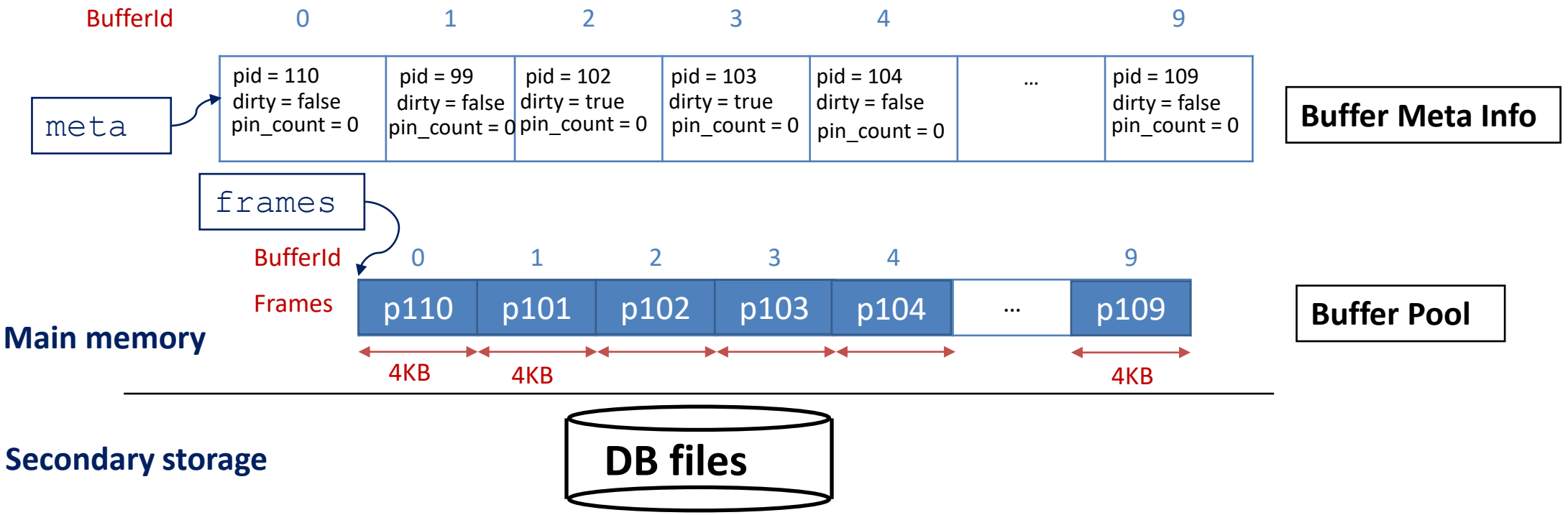
Question: are buffer pins necessary when the DBMS is single-threaded?

Yes. Think about why?



Eviction policy

- How do we choose a victim for eviction?
  - Randomly? The one with the lowest buffer ID that is not pinned? (Inefficient!)



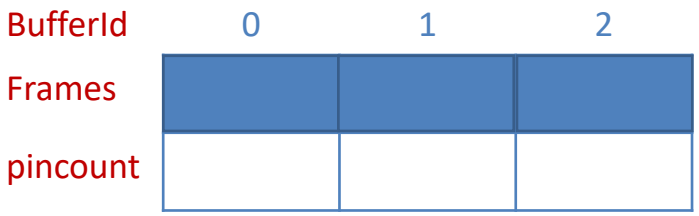
Eviction policy

- Eviction policy (aka replacement policy)
  - An algorithm for choosing unpinned frames when there's no free frame
    - It can have huge impacts on the # of I/Os, depending on the access pattern
  - Many common choices:
    - Least recently used (LRU)
    - Most recently used (MRU)
    - Clock
    - Database workload specific policies
    - …

Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)



Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)

BufferId	0	1	2
Frames	p1	p2	p3
pincount	1	1	1

Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)

BufferId	0	1	2
Frames	p1	p2	p3
pincount	1	0	1

LRU list:

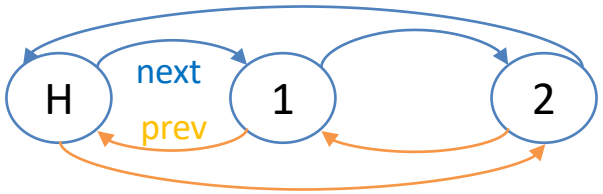
Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)

BufferId	0	1	2
Frames	p1	p2	p3
pincount	1	0	0

LRU list:



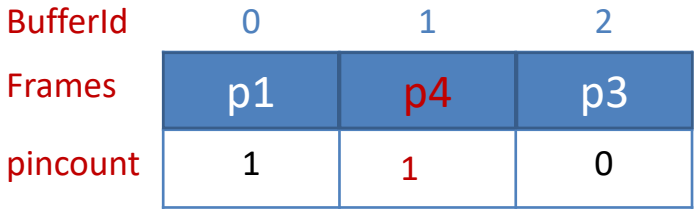
How to implement in practice?

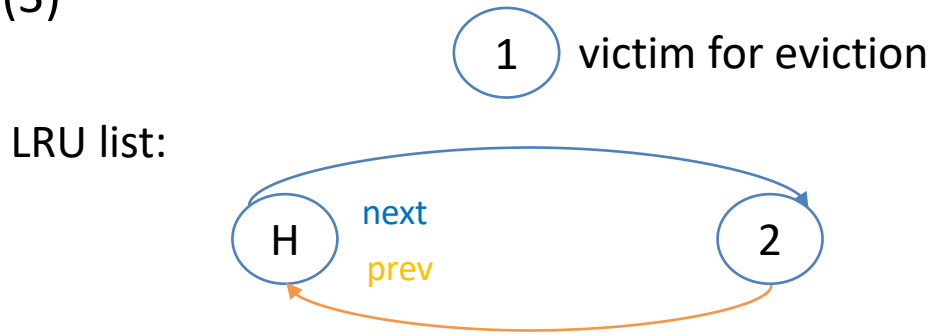
Exercise: how to remove a node in the middle of LRU list when there's a buffer hit?

Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)



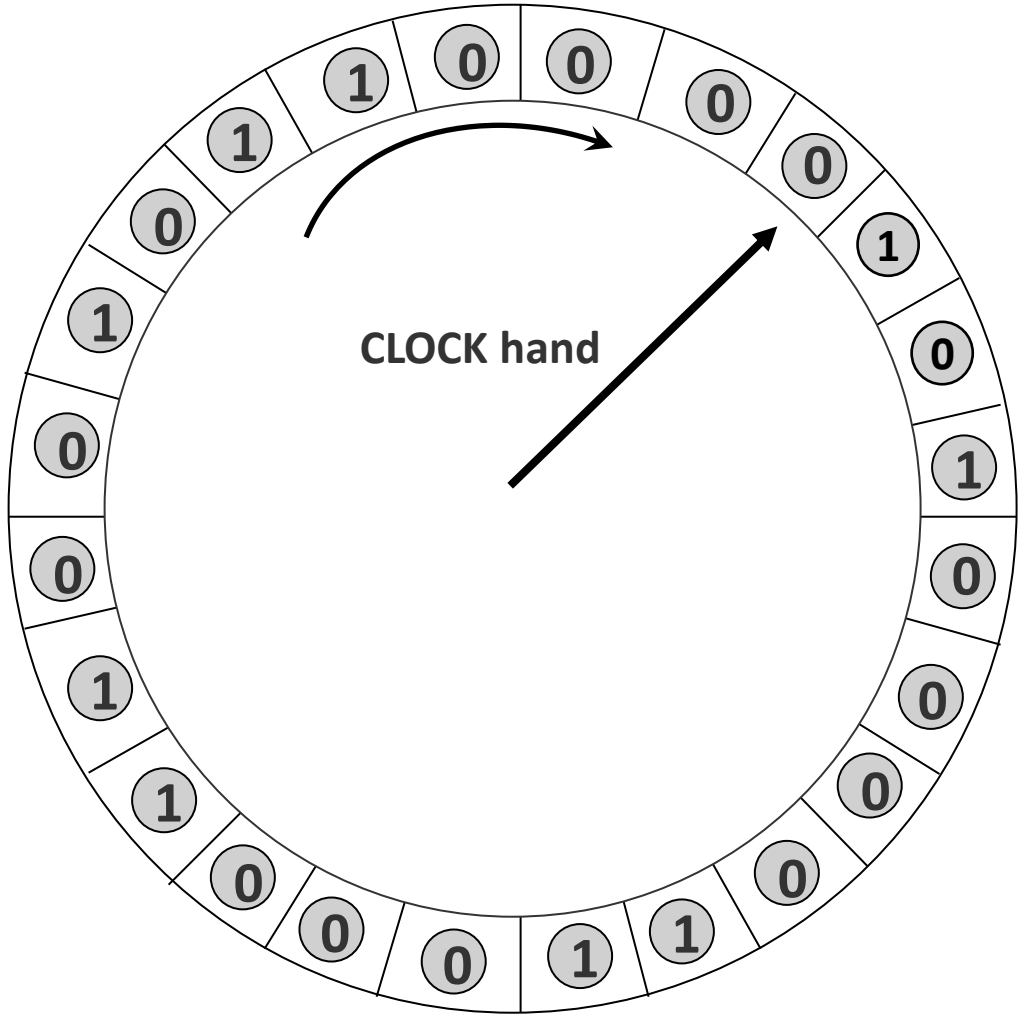


Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages that *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload
- Problems?
  - Sequential flooding:
    - # buffer frames < # pages in file means every existing page in the buffer gets evicted
    - Prevents buffer hit for other transactions working on other files
- DB may know the access pattern before hand so that it can adapt its replacement policies
  - e.g., using a small ring buffer for sequential scan to avoid flooding the entire buffer pool

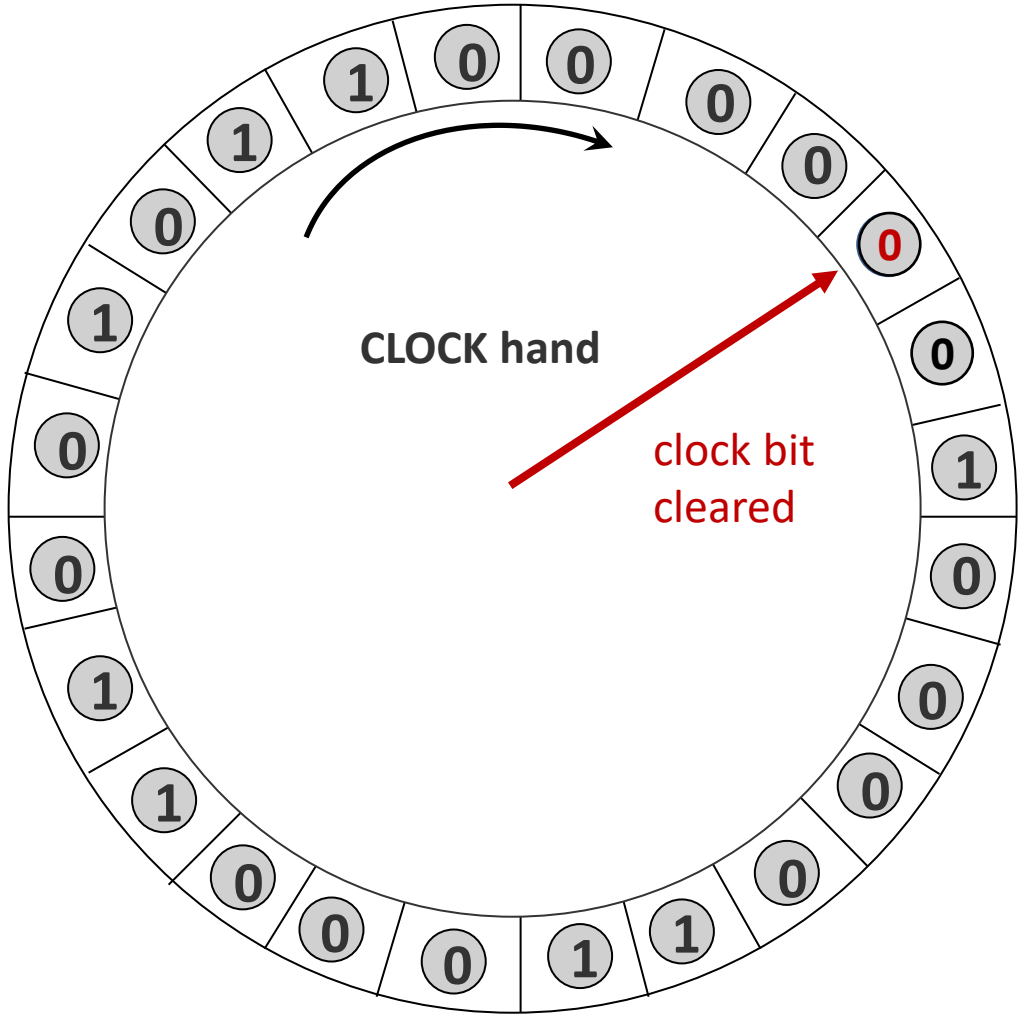
Clock policy

- Approximate LRU
- Each buffer frame has a clock bit
  - Set upon page pinned or unpinned (why?)
- When we need an eviction, move the clock hand
  - Ignore any page that is still pinned
  - Otherwise
    - If bit is set, clear it
    - If bit is clear, evict it
    - i.e., second chance



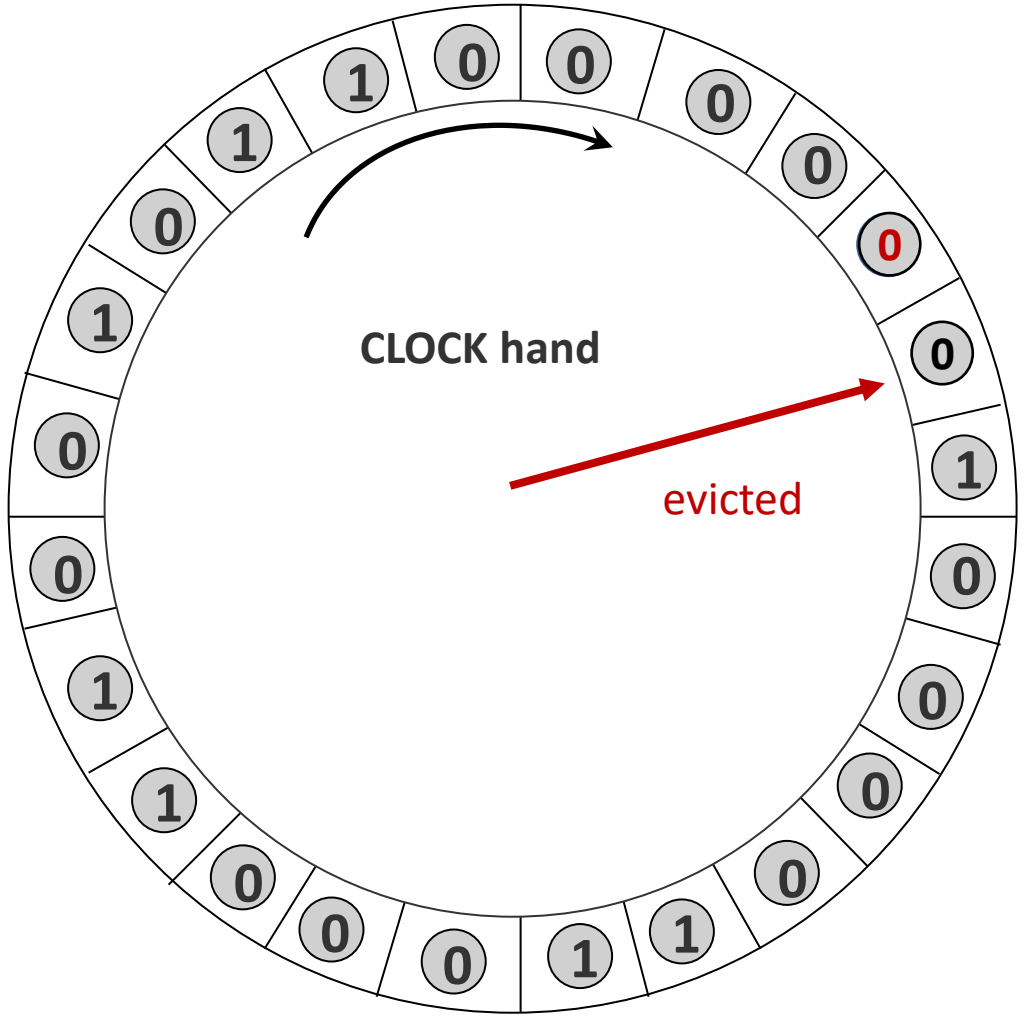
Clock policy

- Approximate LRU
- Each buffer frame has a clock bit
  - Set upon page pinned or unpinned (why?)
- When we need an eviction, move the clock hand
  - Ignore any page that is still pinned
  - Otherwise
    - If bit is set, clear it
    - If bit is clear, evict it
    - i.e., second chance



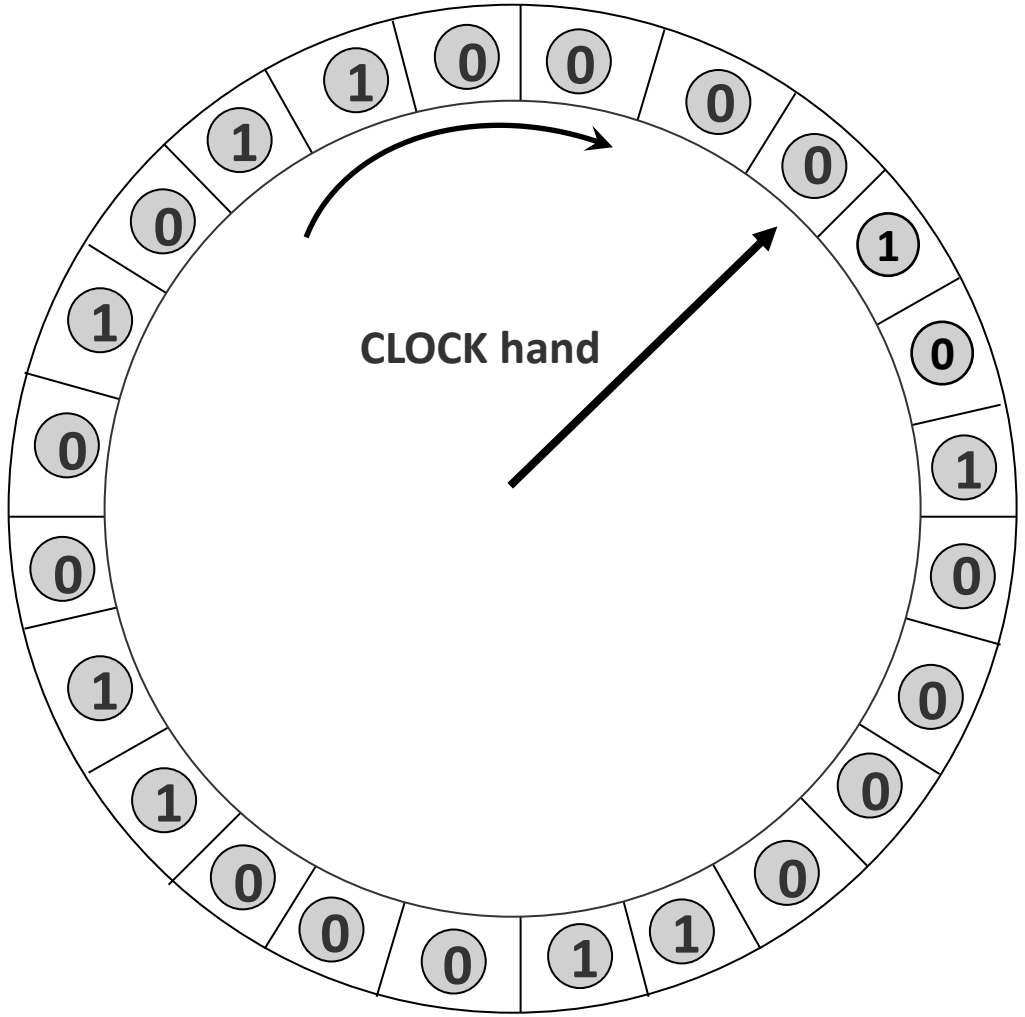
Clock policy

- Approximate LRU
- Each buffer frame has a clock bit
  - Set upon page pinned or unpinned (why?)
- When we need an eviction, move the clock hand
  - Ignore any page that is still pinned
  - Otherwise
    - If bit is set, clear it
    - If bit is clear, evict it
    - i.e., second chance
- Why this might be faster and easier to implement than LRU?
  - Hint: put the clock bit into the buffer meta structures
    - scan buffer meta structures instead



Clock policy

- Approximate LRU
- Each buffer frame has a clock bit
  - Set upon page pinned or unpinned (why?)
- When we need an eviction, move the clock hand
  - Ignore any page that is still pinned
  - Otherwise
    - If bit is set, clear it
    - If bit is clear, evict it
    - i.e., second chance
- Alternative: third/fourth/… chance
  - allowing clock counters up to 2/3/…



Buffer flush

- When are dirty pages written back to disk?
  - When evicted
  - During shutdown

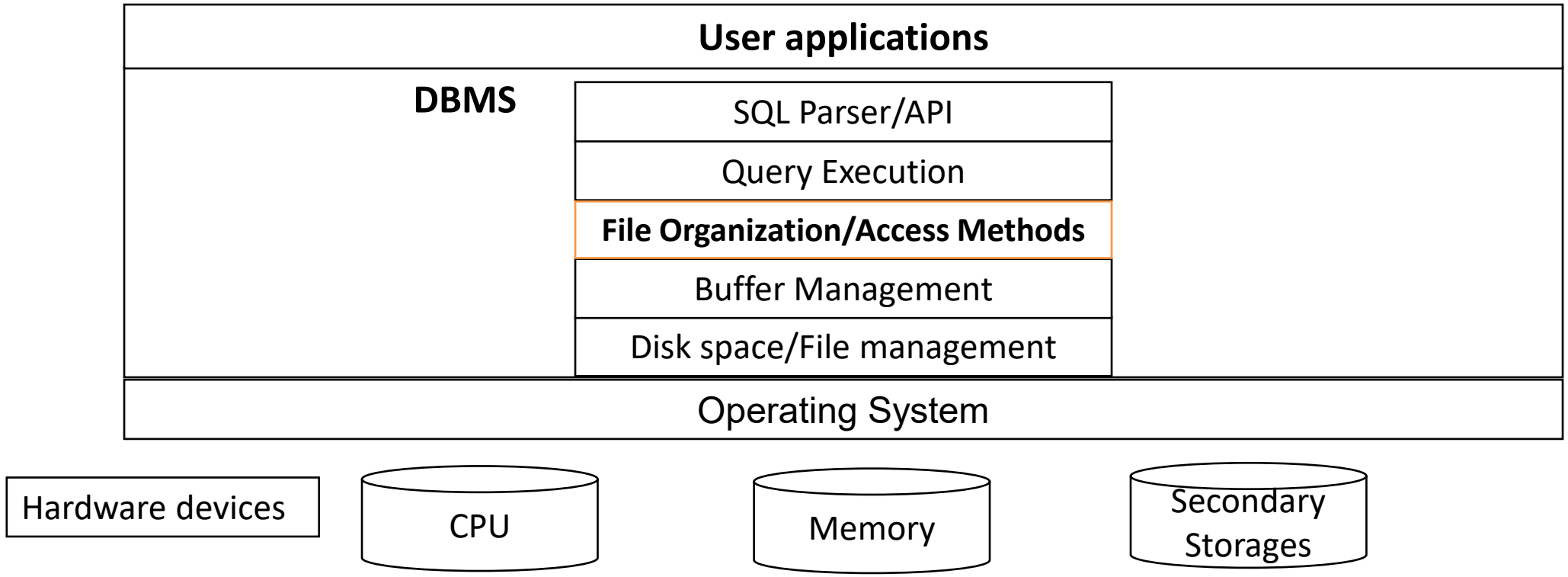
  - Forced flush: flushing certain dirty pages to disk
    - when data need to be persisted for data consistency
    - only unpinned page may be flushed
    - other constraints apply (discussed later this semester)

DBMS vs. OS File System

OS does disk space & buffer management as well: why not let OS manage these tasks?

- Some limitations, e.g., files can't span disks.
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk & order writes (important for implementing CC, concurrency control, & recovery)
  - adjust *eviction policy,* and prefetch pages based on access patterns in typical DB operations.

Big Picture


User applications
DBMS
SQL Parser/API
Query Execution
File Organization/Access Methods
Buffer Management
Disk space/File management
Operating System
Hardware devices
CPU
Memory
Secondary Storages

Relational database

- A relational Database is *logically* a collection of *tables* (aka *relations*)
- **Table schema**: each *table* has one or more *fields* (aka *columns*)
  - Each *field* has a type and (usually) a name
- **Table instance**: a *table* is a (multi-)set of *records* (aka *rows/tuples*)
  - Each *record* has one value or NULL for each *field* in the *table schema*
    - The field type dictates the set of valid values

student

sid	name	login
100	Alice	alicer34
101	Bob	bob5
102	Charlie	charlie7
103	David	davel

enrollment

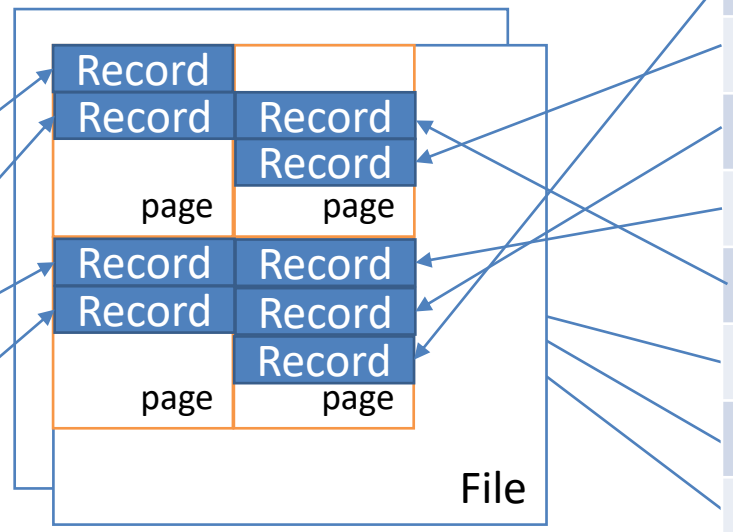
sid	semester	cno	grade
100	s22	562	2.0
102	s22	562	2.3
100	f21	560	3.7
101	s21	560	3.3
102	f21	560	4.0
103	s22	460	2.7
101	f21	560	3.3
103	f21	250	4.0

Database storage architecture

- Mapping from relational database to physical storage
  - Database -> files
  - Records -> contiguous bytes on fixed-size pages (e.g., 4KB)
    - Assumption: each record fits in a page
    - What if a record does not fit?

student

sid	name	login
100	Alice	alicer34
101	Bob	bob5
102	Charlie	charlie7
103	David	davel



enrollment

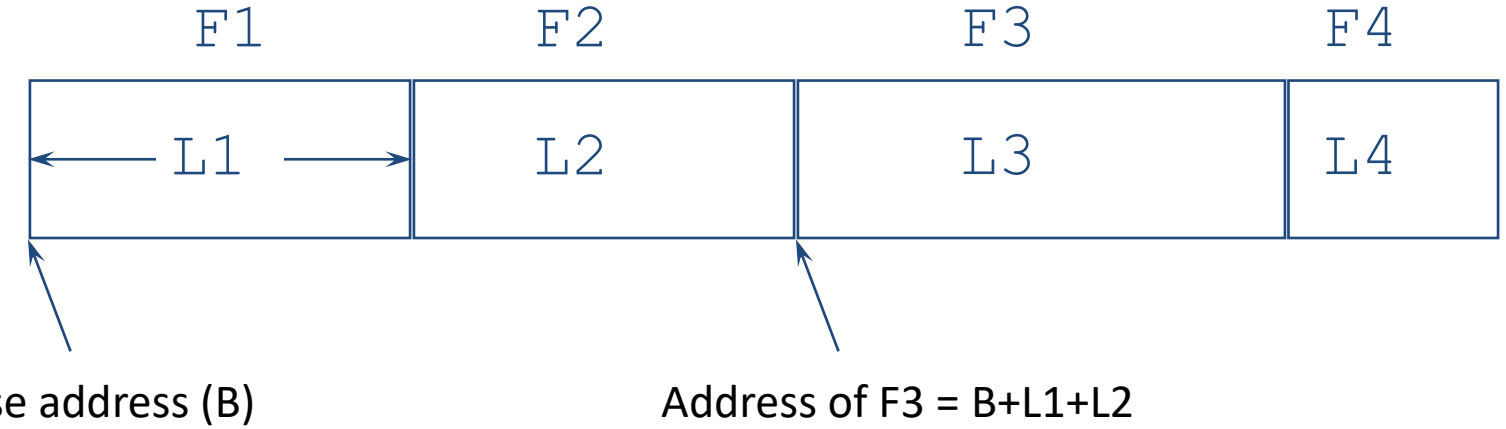
sid	semester	cno	grade
100	s22	562	2.0
102	s22	562	2.3
100	f21	560	3.7
101	s21	560	3.3
102	f21	560	4.0
103	s22	460	2.7
101	f21	560	3.3
103	f21	250	4.0

- What about relations?

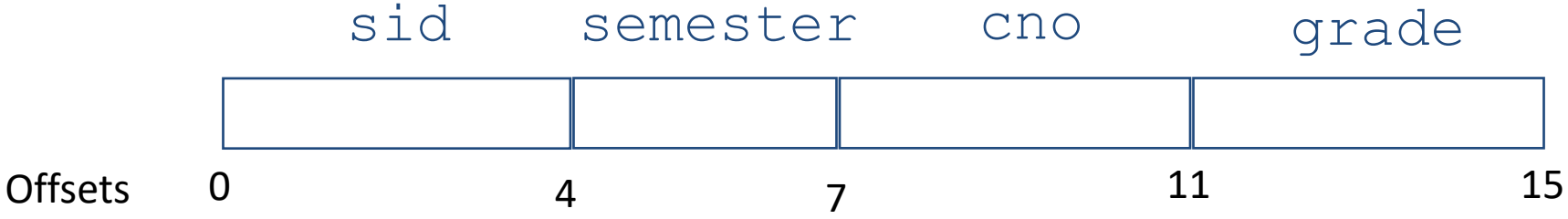
  One/several file(s) per relation? Mixing records from correlated relations in one/several file(s)?

Record format: fixed-length

- Fixed-length record
  - Assuming all fields F1, F2, F3, … have known (maximum) length
    - Denote the maximum lengths as L1, L2, L3, …
  - Base address B: may be a file offset or a memory address
  - Offset of field $Fi = \sum_{j=1}^{i-1} Li$

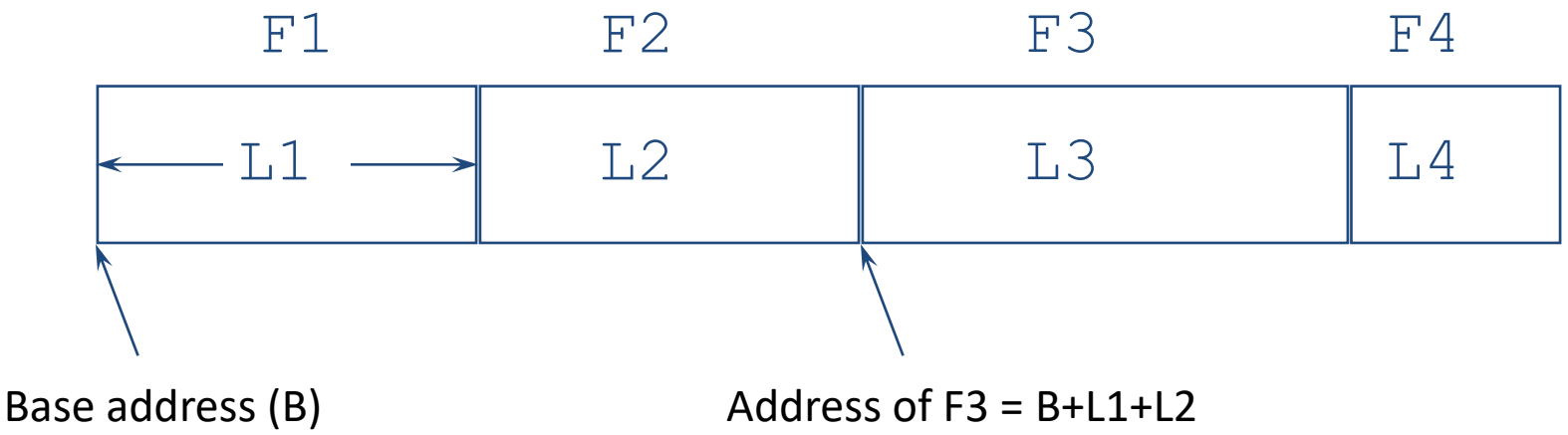


Example: consider the enrollment table *E(sid: INT4, semester: CHAR(3), cno: INT4, grade: FLOAT)*



Record format: fixed-length

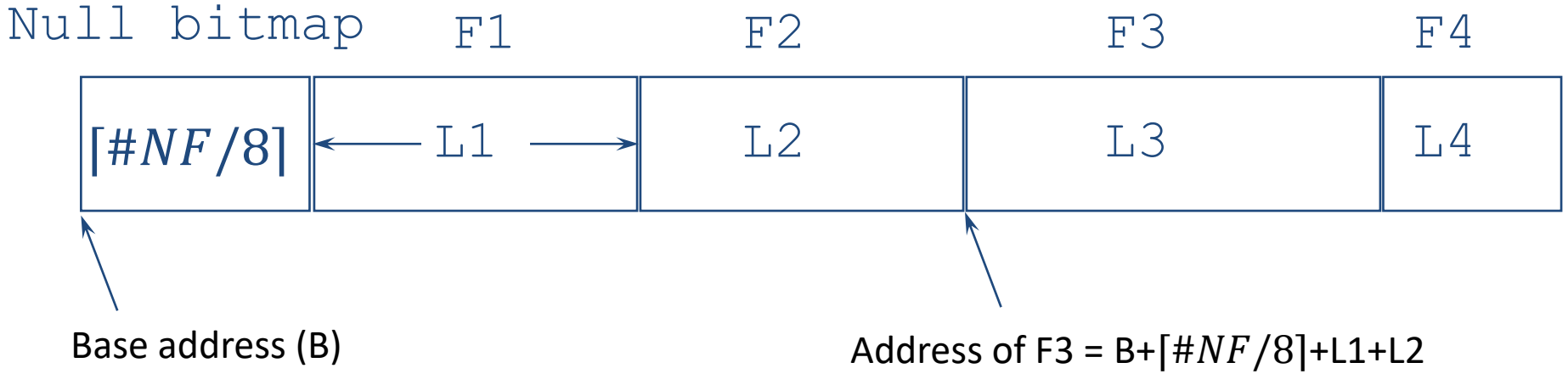
- Fixed-length record
  - How to handle NULLs?



Record format: fixed-length

- Fixed-length record
  - How to handle NULLs?
    - *Null bitmap*: set the i^{th} bit if Fi is NULL. Otherwise, clear the i^{th} bit.

#NF: number of nullable fields



Example: consider the enrollment table *E(sid: INT4, semester: CHAR(3), cno: INT4, grade: FLOAT), NF = 4*

Address alignment in records

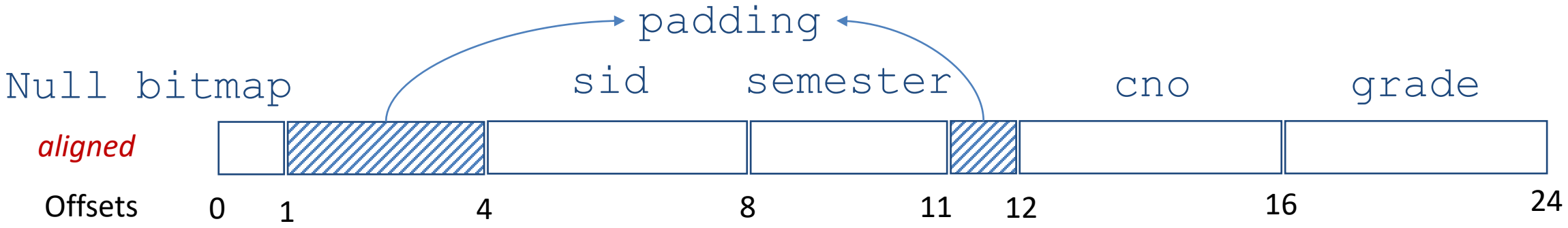
- Address alignment requirements?
  - Alignment example: to read/write a 32-bit integer *in memory*, *its address* $mod\ 4 == 0$
  - Most architecture has address alignment requirements
    - Some strictly enforces alignment (most RISC arch, e.g., ARM v5 or earlier)
    - Some don't but have restrictions/performance loss/atomicity issues (e.g., x86_64/newer ARM)
  - By default, compilers automatically align values properly
  - DB records? Two choices:
    - Pack everything, and memcpy the field before access
      - Less efficient, but save space
    - Align offsets manually
      - More efficient field access, but waste space

```
struct A {
  int32_t x;
  int16_t y;
  int64_t z;
};

// alignof(A) == 8
// offsetof(A, x) == 0
// offsetof(A, y) == 4
// offsetof(A, z) == 8 (not 6!)
```

Address alignment in records

- Example: consider the enrollment table E(sid: INT4, semester: CHAR(3), cno: INT4, grade: FLOAT), NF = 4
  - alignment requirements
    - INT4: 4
    - CHAR: 1
    - FLOAT: 4



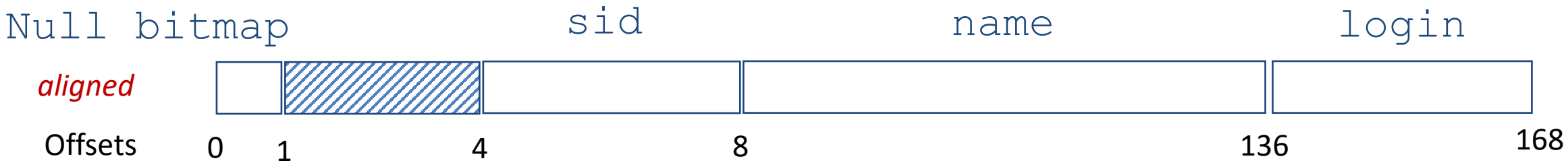
Is there any assumption for the fields in an aligned record to be really aligned?

Base address B must be aligned to the strictest alignment requirement. (depends on arch, OS and DB type system)

Record format: fixed-length

- Problem with fixed-length record?
  - What if we have a variable-length field whose maximum length >> average length
    - Wastes space

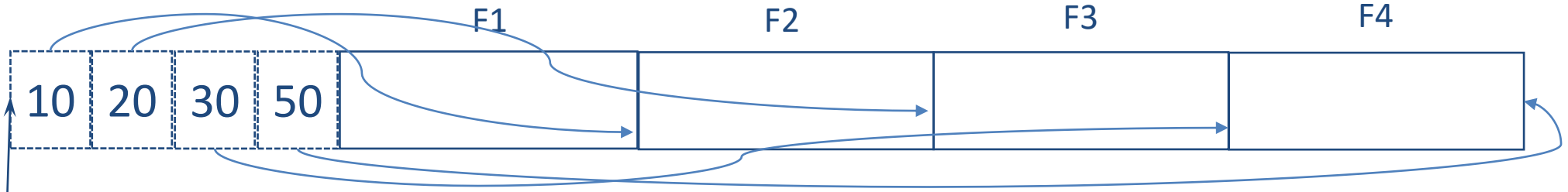
Example: consider the student table *S(sid: INT4, name: VARCHAR(128), login: VARCHAR(32))*



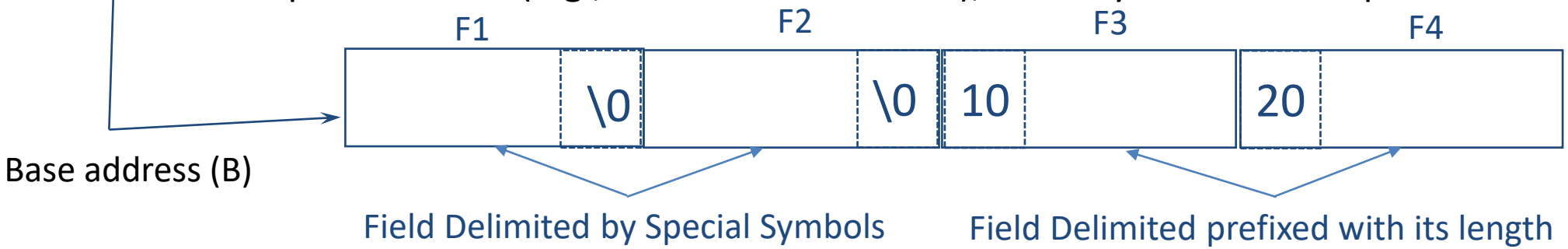
Solution: variable-length records

Record format: variable-length

- Variable-length record
  - Two approaches:
    - Encode field length in an offset array (e.g., stores the end offset of each field)
      - random access to fields given B, but takes more space

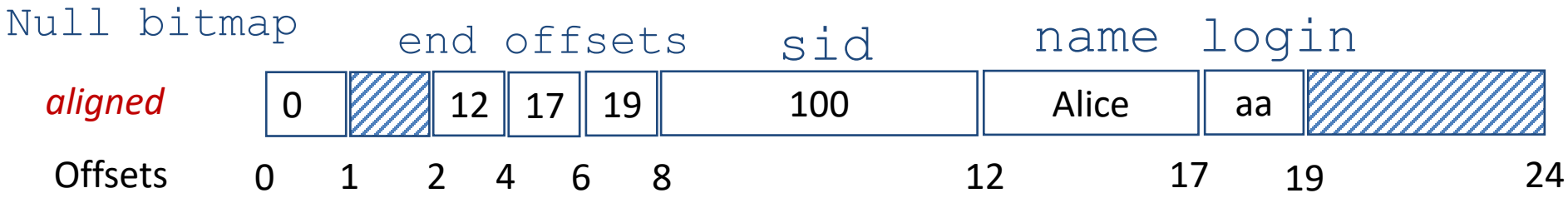


    - Using self-contained data field (with separator/encoded length)
      - Computed offsets (e.g., offset of F3 = L1 + L2); but may be more compact

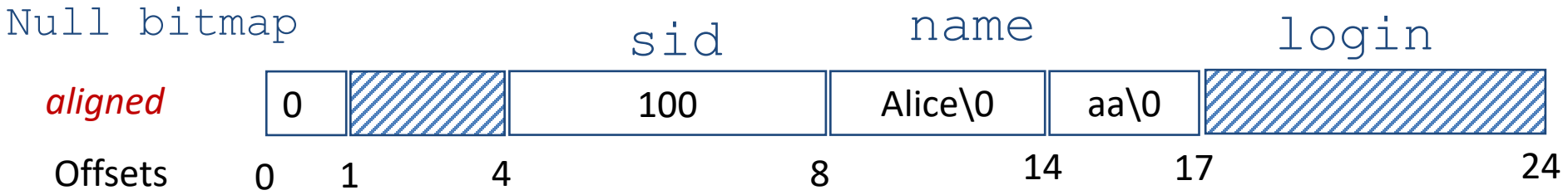


Record format: variable-length

- Example: consider a record in S with (sid = 100, name = ‘Alice’, login = ‘aa’), NF = 3
  - Two approaches:
    - Encode field length in an offset array (e.g., stores the end offset of each field)
      - assuming offsets are stored as `int16_t`

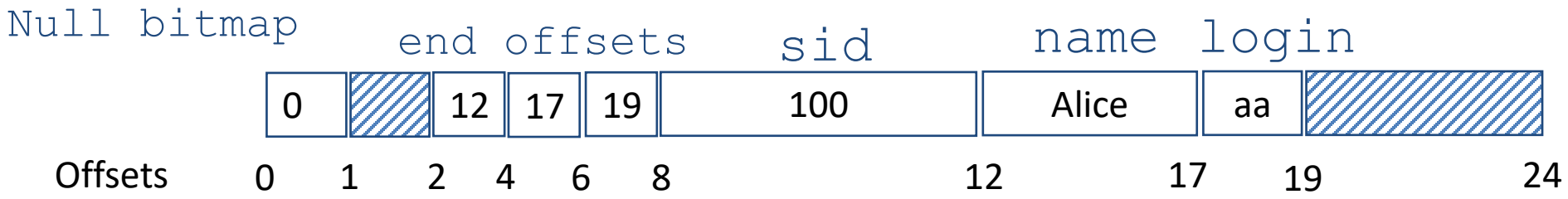


    - Using self-contained data field (with separator/encoded length)
      - Computed offsets (e.g., offset of F3 = L1 + L2); but may be more compact



Record format: variable-length

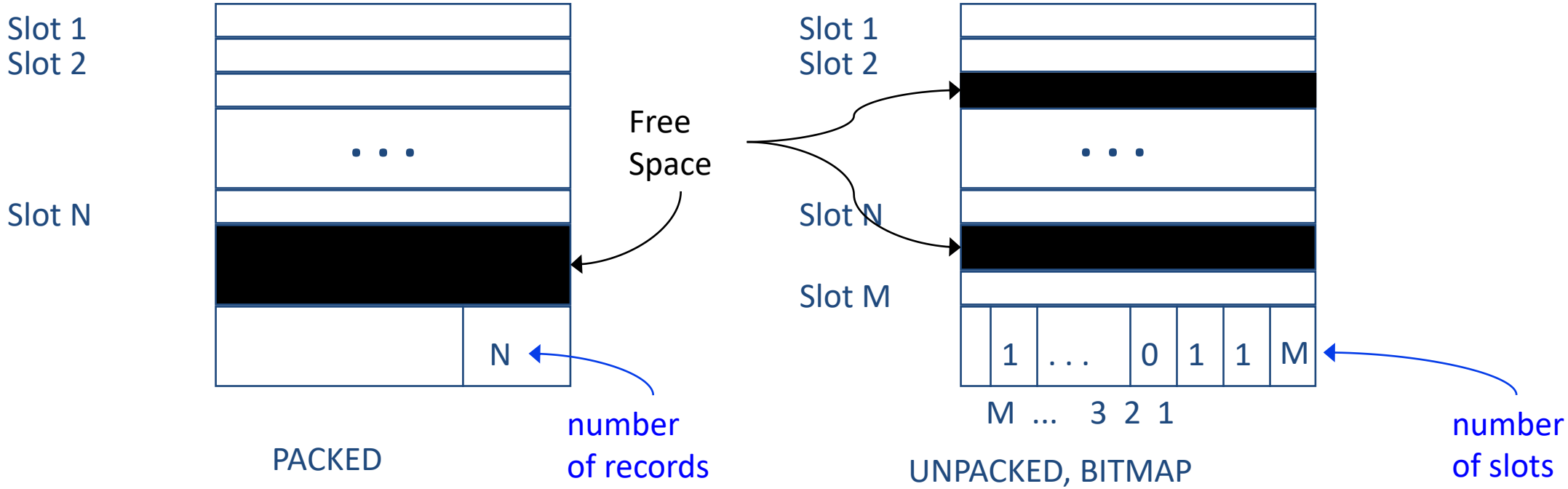
- Example: consider a record in S with (sid = 100, name = 'Alice', login = 'aa'), NF = 3
  - Many possible designs with minor tweaks for different space/time efficiency trade-offs
    - Can also combine both fixed-length and variable-length record formats
  - Encode field length in an offset array (e.g., stores the end offset of each field)
    - assuming offsets are stored as `int16_t`



  - *Example tweaks and assumption:*
    - *Fixed-length fields appear before variable-length fields => have fixed offsets*
    - *(Real) record length without the trailing padding stored somewhere else*

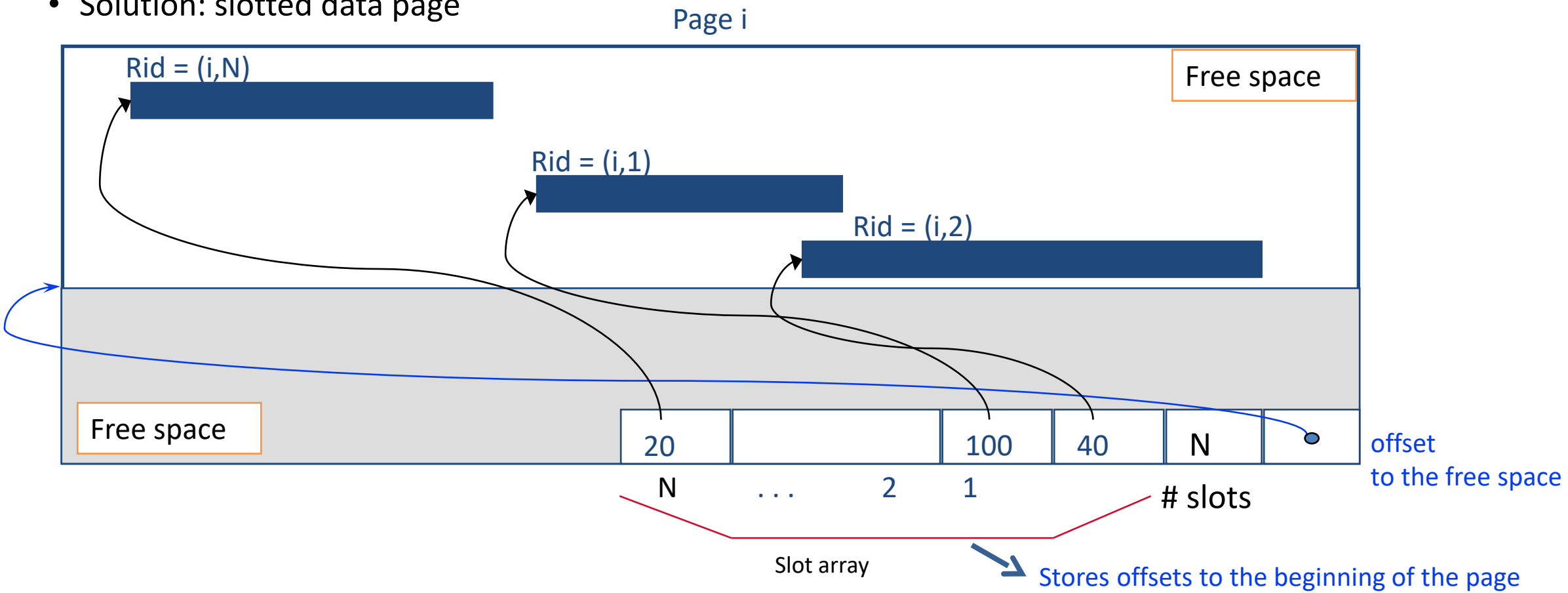
Page layout for fixed-length records

- Why not storing record consecutively in a file?
  - Linear time to update/delete!
- How do we store records in fixed-size pages?
  - Fixed-length record: easy (packed vs unpacked)
    - Not commonly used as it wastes space



Page layout for variable-length records

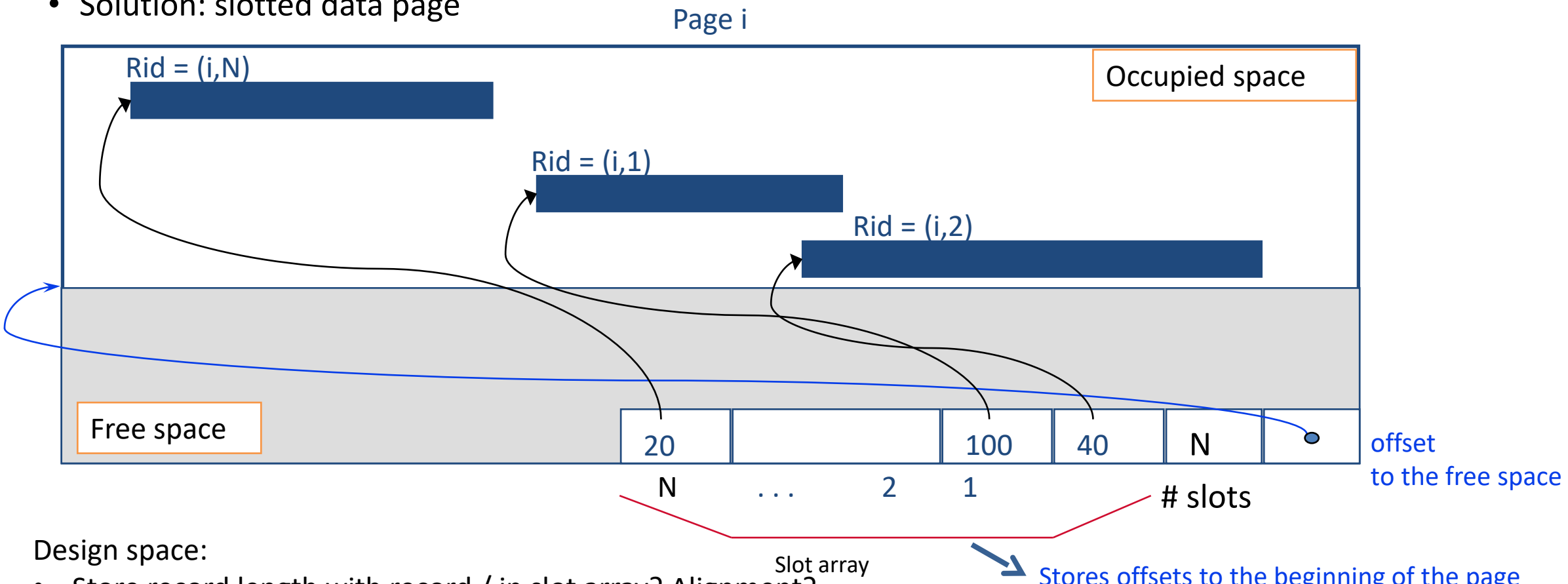
- What about variable-length records?
  - Solution: slotted data page



Can move records within a page without changing its record id.

Page layout for variable-length records

- What about variable-length records?
  - Solution: slotted data page



Design space:

- Store record length with record / in slot array? Alignment?
- Allow free space within the occupied space?
  - Eager vs lazy compaction?
- Optional page header?

Organizing pages in a heap file

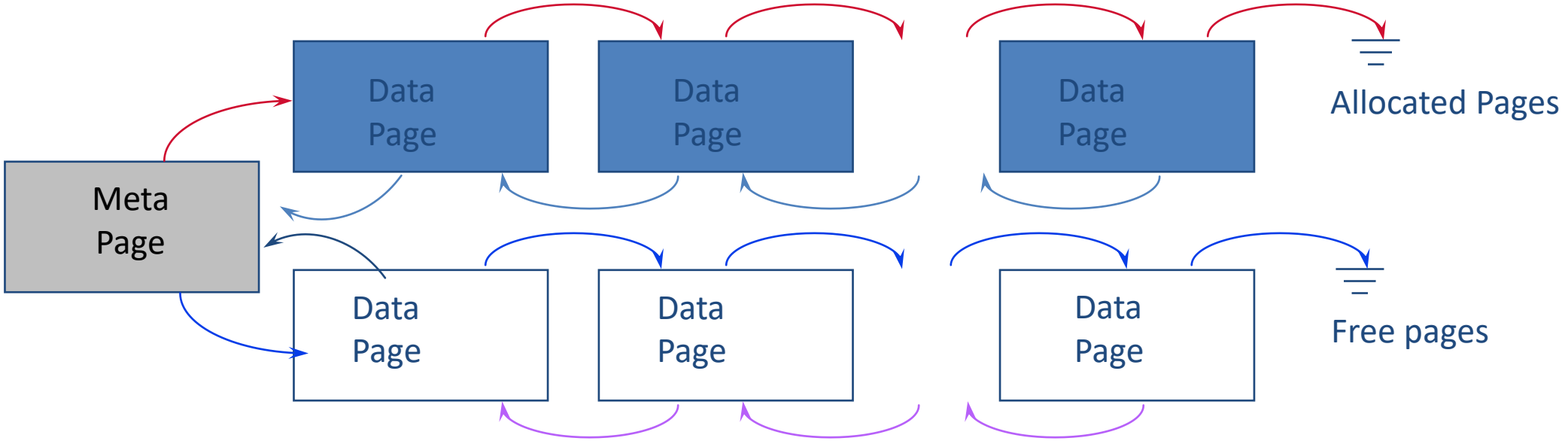
- Heap file is the most basic and common way of managing pages for a single relation
  - Consists of a collection of fixed-size pages
  - Pages/records are unordered
- Heap files must support
  - Efficient insertion/deletion/update of records
  - Efficient access of a record
  - Efficient enumeration of all the records
  - Management of free space (also managed by disk space manager/file system)

- Note
  - A heap file does not necessarily map to a single file on FS
    - A heap file can span multiple FS files (e.g., PostgreSQL)
  - A file on FS does not necessarily only store pages for a single heap file
    - All heap files are stored in a single FS File (i.e., single-file DBMS such as SQLite)
    - Our course project Taco-DB: stores pages of different heap files across a number of files on FS

Organizing pages in a heap file

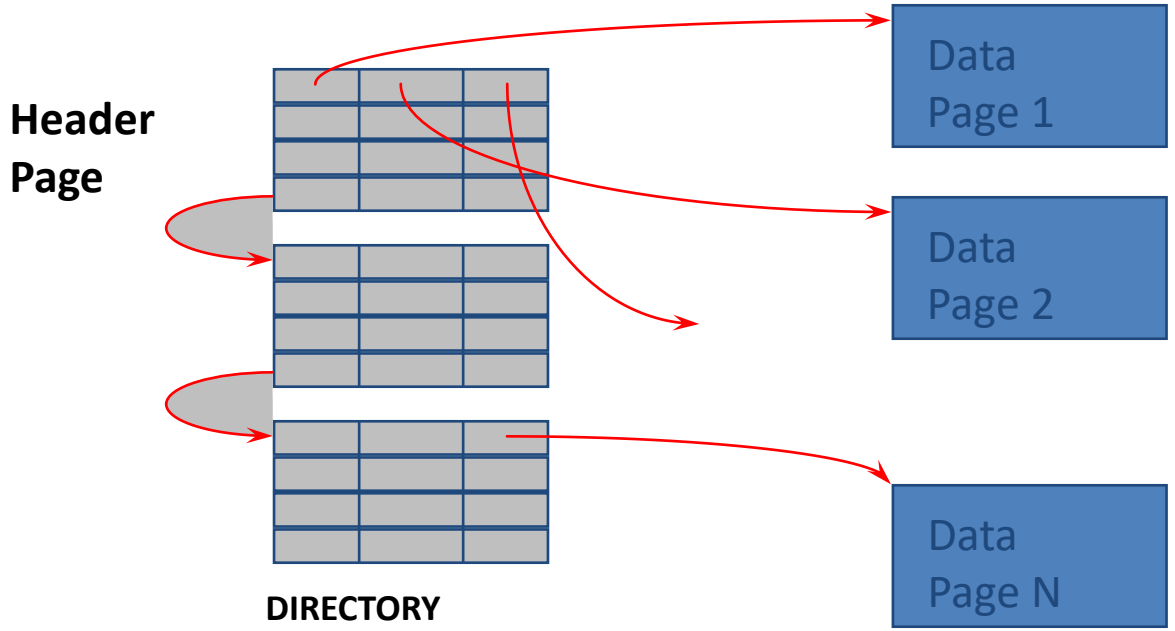
- Many possible alternatives and variants
  - We consider the most representative two of them

Heap file alternative 1: doubly-linked lists



- The header page id and Heap file name must be stored someplace.
  - Database catalog
- Each page contains 2 `pointers' plus data.
  - What are these pointers? Page Number and/or File ID?
- Supports sequential access
  - Random access? Only if you know the page number (and the underlying file system supports random seek)
- Does enumerating the pages through the next pointers always incur sequential I/O?
  - Not necessarily! Depending on how you allocate pages.

Heap file alternative 2: page directory



- The entry for a page can include the number of free bytes on the page.
  - Or use free space bitmap in a (separate) contiguous space.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - Can also allocate contiguous pages for page directory for faster random access and/or using hierarchical page directory
  - *PD is much smaller than the all data pages*!

Database catalog

- How does DBMS remember the layout?
- Catalogs are DBMS defined relations that
  - stores meta-information about
    - Relation schemas
    - Physical storage format and location
    - And many other important internal states
- *Can be implemented as regular relations*

Table

TABID	TABNAME	TABFPATH
1	TABLE	/dbdata/1
2	COLUMN	/dbdata/2
100	STUDENT	/dbdata/100
101	ENROLLMENT	/dbdata/101

Column

TABID	COLID	COLNAME	COLTYPNAME
1	0	TABID	OID
1	1	TABNAME	VARCHAR(64)
1	2	TABFPATH	VARCHAR(256)
2	0	TABID	OID
2	1	COLID	INT2
2	2	COLNAME	VARCHAR(64)
2	3	COLTYPNAME	VARCHAR(64)
100	0	SID	SERIAL
100	1	NAME	VARCHAR(32)
100	2	LOGIN	VARCHAR(40)
101	0	SID	INTEGER
101	1	SEMESTER	CHAR(3)
101	2	CNO	INTEGER
101	3	GRADE	DOUBLE

Summary

- This lecture
  - Buffer management (Buffer Pool, Eviction Policies)
  - Data storage layout (Heap file, Data Page, Record)

- Next time:
  - Access methods and Index files
  - Hashing techniques

- Reminders
  - Homework assignment 1 released today; due 2/18 23:59 pm.
    - Use Piazza or office hours to seek clarification if needed.
  - Project 2 will be released on Wednesday, 2/7, due 2/25 23:59 pm.