Buffer Pools
Project #1 is due Sunday, Sept 26\textsuperscript{th} @11:59pm

Q&A Session about the project on Thursday, Sept 16\textsuperscript{th} @4:00pm
→ Zoom link posted on Piazza
DATABASE STORAGE

Problem #1: How the DBMS represents the database in files on disk.

Problem #2: How the DBMS manages its memory and move data back-and-forth from disk.
DATABASE STORAGE

Spatial Control:
→ Where to write pages on disk.
→ The goal is to keep pages that are used together often as physically close together as possible on disk.

Temporal Control:
→ When to read pages into memory, and when to write them to disk.
→ The goal is to minimize the number of stalls from having to read data from disk.
# ACCESS TIMES

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache Ref</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>L2 Cache Ref</td>
<td>7 ns</td>
</tr>
<tr>
<td>DRAM</td>
<td>100 ns</td>
</tr>
<tr>
<td>SSD</td>
<td>150,000 ns</td>
</tr>
<tr>
<td>HDD</td>
<td>10,000,000 ns</td>
</tr>
<tr>
<td>Network Storage</td>
<td>~30,000,000 ns</td>
</tr>
<tr>
<td>Tape Archives</td>
<td>1,000,000,000 ns</td>
</tr>
</tbody>
</table>

- 0.5 sec
- 7 sec
- 100 sec
- 1.7 days
- 16.5 weeks
- 11.4 months
- 31.7 years
DISK-ORIENTED DBMS

Get page #2

Memory

Buffer Pool

Execution Engine

Disk

Database File

Pages
DISK-ORIENTED DBMS

Memory

Disk

Buffer Pool

Database File

Directory

Header

1

2

3

4

5

Pages

Get page #2
DISK-ORIENTED DBMS

Get page #2

Disk → Memory → Buffer Pool

Directory
Header

Pages

Execution Engine

CMU-DB
15-445/645 (Fall 2021)
**DISK-ORIENTED DBMS**

- **Memory**
  - Buffer Pool
  - **Directory**
    - Page 1
    - Page 2
    - ... (More pages)

- **Disk**
  - Database File
  - **Directory**
    - Page 1
    - Page 2
    - Page 3
    - ... (More pages)

- **Execution Engine**
  - Get page #2
  - Pointer to page #2
TODAY'S AGENDA

Buffer Pool Manager
Replacement Policies
Other Memory Pools
BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages. An array entry is called a frame.

When the DBMS requests a page, an exact copy is placed into one of these frames.
BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages. An array entry is called a frame.

When the DBMS requests a page, an exact copy is placed into one of these frames.

Buffer Pool

page1
frame2
frame3
frame4

On-Disk File

page1 page2 page3 page4
BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages. An array entry is called a **frame**.

When the DBMS requests a page, an exact copy is placed into one of these frames.
The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**
The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:
- Dirty Flag
- Pin/Reference Counter
The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**
The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:

→ **Dirty Flag**

→ **Pin/Reference Counter**
The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:

→ **Dirty Flag**
→ **Pin/Reference Counter**
The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:
- **Dirty Flag**
- **Pin/Reference Counter**
The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:
- **Dirty Flag**
- **Pin/Reference Counter**
LOCKS VS. LATCHES

**Locks:**

→ Protects the database's logical contents from other transactions.
→ Held for transaction duration.
→ Need to be able to rollback changes.

**Latches:**

→ Protects the critical sections of the DBMS's internal data structure from other threads.
→ Held for operation duration.
→ Do not need to be able to rollback changes.
PAGE TABLE VS. PAGE DIRECTORY

The **page directory** is the mapping from page ids to page locations in the database files.
→ All changes must be recorded on disk to allow the DBMS to find on restart.

The **page table** is the mapping from page ids to a copy of the page in buffer pool frames.
→ This is an in-memory data structure that does not need to be stored on disk.
Global Policies:
→ Make decisions for all active txns.

Local Policies:
→ Allocate frames to a specific txn without considering the behavior of concurrent txns.
→ Still need to support sharing pages.
BUFFER POOL OPTIMIZATIONS

Multiple Buffer Pools
Pre-Fetching
Scan Sharing
Buffer Pool Bypass
The DBMS does not always have a single buffer pool for the entire system.

- Multiple buffer pool instances
- Per-database buffer pool
- Per-page type buffer pool

Helps reduce latch contention and improve locality.
MULTIPLE BUFFER POOLS

Approach #1: Object Id
→ Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.

Approach #2: Hashing
→ Hash the page id to select which buffer pool to access.

Buffer Pool #1

Buffer Pool #2

Q1 GET RECORD 123
MULTIPLE BUFFER POOLS

**Approach #1: Object Id**
→ Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.

**Approach #2: Hashing**
→ Hash the page id to select which buffer pool to access.
MULTIPLE BUFFER POOLS

Approach #1: Object Id
→ Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.

Approach #2: Hashing
→ Hash the page id to select which buffer pool to access.
The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans

→ Index Scans
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans
The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans
→ Index Scans
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans
The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans
→ Index Scans
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans
The DBMS can also prefetch pages based on a query plan.

- Sequential Scans
- Index Scans
The DBMS can also prefetch pages based on a query plan.  
→ Sequential Scans  
→ Index Scans
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans
Q1

SELECT * FROM A
WHERE val BETWEEN 100 AND 250

PRE-FETCHING

Disk Pages

index-page0
index-page1
index-page2
index-page3
index-page4
index-page5

Buffer Pool
PRE-FETCHING

Disk Pages
- index-page0
- index-page1
- index-page2
- index-page3
- index-page4
- index-page5

Buffer Pool
- index-page0
- index-page1
- index-page2
- index-page3
- index-page4
- index-page5
PRE-FETCHING

Buffer Pool

Disk Pages

index-page0
index-page1
index-page2
index-page3
index-page4
index-page5
index-page6
PRE-FETCHING

Buffer Pool

Disk Pages

Q1

index-page0
index-page1
index-page2
index-page3
index-page4
index-page5
index-page6
PRE-FETCHING

Buffer Pool

index-page0

Disk Pages

index-page0
index-page1
index-page2
index-page3
index-page4
index-page5

Q1
PRE-FETCHING

Buffer Pool

index-page0

index-page1

Disk Pages

index-page0

index-page1

index-page2

index-page3

index-page4

index-page5

index-page6

Q1

0----99 100----199 200----299 300----399
PRE-FETCHING

Buffer Pool

index-page0
index-page1

Disk Pages

index-page0
index-page1
index-page2
index-page3
index-page4
index-page5
PRE-FETCHING

Buffer Pool
- index-page0
- index-page1

Disk Pages
- index-page0
- index-page1
- index-page2
- index-page3
- index-page4
- index-page5
- index-page6

Q1
- index-page1
- index-page3
- index-page5
Queries can reuse data retrieved from storage or operator computations.
→ Also called *synchronized scans*.
→ This is different from result caching.

Allow multiple queries to attach to a single cursor that scans a table.
→ Queries do not have to be the same.
→ Can also share intermediate results.
SCAN SHARING

If a query wants to scan a table and another query is already doing this, then the DBMS will attach the second query's cursor to the existing cursor.

Examples:
→ Fully supported in IBM DB2, MSSQL, and Postgres.
→ Oracle only supports cursor sharing for identical queries.
If a query wants to scan a table and another query is already doing this, then the DBMS will attach the second query's cursor to the existing cursor. Examples:

→ Fully supported in IBM DB2, MSSQL, and Postgres.
→ Oracle only supports cursor sharing for identical queries.

If it cannot be implemented, what about the Postgres design prevents it from doing so? How can this be circumvented? If it is possible, what is preventing the implementation today?

Also, do you know why everyone says that Andy smells so bad? I've heard that he smells like old Arby's beef-and-cheddar sandwiches that have been left out in the sun for too long.
If a query wants to scan a table and another query is already doing this, then the DBMS will attach the second query's cursor to the existing cursor.

Examples:
- Fully supported in IBM DB2, MSSQL, and Postgres.
- Oracle only supports cursor sharing for identical queries.

`synchronize_seqscans (boolean)`

This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload. When this is enabled, a scan might start in the middle of the table and then "wrap around" the end to cover all rows, so as to synchronize with the activity of scans already in progress. This can result in unpredictable changes in row ordering returned by queries that have no ORDER BY clause. Setting this parameter to off ensures the pre-8.3 behavior in which a sequential scan always starts from the beginning of the table.

The default is on.
SCAN SHARING

Q1 \[\text{SELECT \text{SUM}(val) FROM A}\]

Buffer Pool

Disk Pages

- page0
- page1
- page2
- page3
- page4
- page5
SCAN SHARING

Q1: SELECT \text{SUM}(\text{val}) \text{ FROM } A

Buffer Pool

Disk Pages

page0
page1
page2
page3
page4
page5
Q1: \( \text{SELECT SUM(val)} \text{ FROM A} \)
SCAN SHARING

Q1: $\text{SELECT } \text{SUM(val)} \text{ FROM } A$

Buffer Pool
- page0
- page1
- page2

Disk Pages
- page0
- page1
- page2
- page3
- page4
- page5

Q1
SCAN SHARING

Q1: \[ \text{SELECT SUM(val) FROM A} \]
SCAN SHARING

Q1

SELECT SUM(val) FROM A

Buffer Pool

page0
page1
page2

Disk Pages

page0
page1
page2
page3
page4
page5
Q1: \( \text{SELECT SUM(val) FROM A} \)
SCAN SHARING

Q1: \[ \text{SELECT } \text{SUM}(\text{val}) \text{ FROM } A \]

Q2: \[ \text{SELECT } \text{AVG}(\text{val}) \text{ FROM } A \]

Buffer Pool
- page3
- page1
- page2

Disk Pages
- page0
- page1
- page2
- page3
- page4
- page5

Q1
Scan Sharing

Q1: \( \text{SELECT } \text{SUM}(\text{val}) \text{ FROM } A \)

Q2: \( \text{SELECT } \text{AVG}(\text{val}) \text{ FROM } A \)

Buffer Pool:
- page3
- page1
- page2

Disk Pages:
- page0
- page1
- page2
- page3
- page4
- page5
SCAN SHARING

Q1: SELECT SUM(val) FROM A

Q2: SELECT AVG(val) FROM A

Buffer Pool
- page3
- page1
- page2

Disk Pages
- page0
- page1
- page2
- page3
- page4
- page5
SCAN SHARING

Q1
SELECT SUM(val) FROM A

Q2
SELECT AVG(val) FROM A

Buffer Pool
- page3
- page4
- page5

Disk Pages
- page0
- page1
- page2
- page3
- page4
- page5

Q2 Q1
**SCAN SHARING**

Q1: `SELECT SUM(val) FROM A`

Q2: `SELECT AVG(val) FROM A`

Buffer Pool:
- page3
- page4
- page5

Disk Pages:
- page0
- page1
- page2
- page3
- page4
- page5
SCAN SHARING

Q1: SELECT SUM(val) FROM A

Q2: SELECT AVG(val) FROM A

Buffer Pool
- page0
- page1
- page2

Disk Pages
- page0
- page1
- page2
- page3
- page4
- page5
SCAN SHARING

Q1: \[ \text{SELECT} \ \text{SUM}(\text{val}) \ \text{FROM} \ A \]

Q2: \[ \text{SELECT} \ \text{AVG}(\text{val}) \ \text{FROM} \ A \ \text{LIMIT} \ 100 \]

Buffer Pool

Disk Pages

page0

page1

page2

page3

page4

page5
BUFFER POOL BYPASS

The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead.
→ Memory is local to running query.
→ Works well if operator needs to read a large sequence of pages that are contiguous on disk.
→ Can also be used for temporary data (sorting, joins).

Called "Light Scans" in Informix.
Most disk operations go through the OS API. Unless you tell it not to, the OS maintains its own filesystem cache (i.e., the page cache).

Most DBMSs use direct I/O (\texttt{O\_DIRECT}) to bypass the OS’s page cache.
→ Redundant copies of pages.
→ Different eviction policies.
→ Loss of control over file I/O.
BUFFER REPLACEMENT POLICIES

When the DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool.

Goals:
→ Correctness
→ Accuracy
→ Speed
→ Meta-data overhead
LEAST-RECENTLY USED

Maintain a single timestamp of when each page was last accessed.

When the DBMS needs to evict a page, select the one with the oldest timestamp.
→ Keep the pages in sorted order to reduce the search time on eviction.
**CLOCK**

Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
**CLOCK**

Approximation of LRU that does not need a separate timestamp per page.

→ Each page has a **reference bit**.

→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

→ Upon sweeping, check if a page's bit is set to 1.

→ If yes, set to zero. If no, then evict.
CLOCK

Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.

→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.

→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
CLOCK

Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.

→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
CLOCK

Approximation of LRU that does not need a separate timestamp per page.
→ Each page has a reference bit.
→ When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.
PROBLEMS

LRU and CLOCK replacement policies are susceptible to *sequential flooding*.
→ A query performs a sequential scan that reads every page.
→ This pollutes the buffer pool with pages that are read once and then never again.

In some workloads the most recently used page is the most unneeded page.
**SEQUENTIAL FLOODING**

Q1: `SELECT * FROM A WHERE id = 1`

**Disk Pages**
- page0
- page1
- page2
- page3
- page4
- page5

(Buffer Pool diagram not shown)
SEQUENTIAL FLOODING

Q1: \( \text{SELECT } * \text{ FROM A WHERE id = 1} \)

Disk Pages:
- page0
- page1
- page2
- page3
- page4
- page5

Buffer Pool
SELECT * FROM A WHERE id = 1
SEQUENTIAL FLOODING

Q1: \( \text{SELECT} \ast \text{ FROM A WHERE id = 1} \)

Q2: \( \text{SELECT} \ \text{AVG(val)} \ \text{FROM A} \)

Buffer Pool

Disk Pages

page0

page1

page2

page3

page4

page5
**SEQUENTIAL FLOODING**

- **Q1**  
  `SELECT * FROM A WHERE id = 1`

- **Q2**  
  `SELECT AVG(val) FROM A`

**Buffer Pool**

- page0
- page1
- page2

**Disk Pages**

- page0
- page1
- page2
- page3
- page4
- page5
SEQUENTIAL FLOODING

Q1: SELECT * FROM A WHERE id = 1
Q2: SELECT AVG(val) FROM A

Buffer Pool
- page0
- page1
- page2

Disk Pages
- page0
- page1
- page2
- page3
- page4
- page5

Q2
SEQUENTIAL FLOODING

Q1: SELECT * FROM A WHERE id = 1
Q2: SELECT AVG(val) FROM A

Buffer Pool
- page3
- page1
- page2

Disk Pages
- page0
- page1
- page2
- page3
- page4
- page5

Q2
SEQUENTIAL FLOODING

**Q1**
SELECT * FROM A WHERE id = 1

**Q2**
SELECT AVG(val) FROM A

**Q3**
SELECT * FROM A WHERE id = 1

**Buffer Pool**
- page3
- page1
- page2

**Disk Pages**
- page0
- page3
- page1
- page2
- page4
- page5
### Sequential Flooding

**Q1**
```
SELECT * FROM A WHERE id = 1
```

**Q2**
```
SELECT AVG(val) FROM A
```

**Q3**
```
SELECT * FROM A WHERE id = 1
```

**Disk Pages**
- page0
- page1
- page2
- page3
- page4
- page5

**Buffer Pool**
- page3
- page1
- page2
BETTER POLICIES: LRU-K

Track the history of last $K$ references to each page as timestamps and compute the interval between subsequent accesses.

The DBMS then uses this history to estimate the next time that page is going to be accessed.
THE DBMS chooses which pages to evict on a per txn/query basis. This minimizes the pollution of the buffer pool from each query.

→ Keep track of the pages that a query has accessed.

Example: Postgres maintains a small ring buffer that is private to the query.
BETTER POLICIES: PRIORITY HINTS

The DBMS knows about the context of each page during query execution. It can provide hints to the buffer pool on whether a page is important or not.
BETTER POLICIES: PRIORITY HINTS

The DBMS knows about the context of each page during query execution.

It can provide hints to the buffer pool on whether a page is important or not.
The DBMS knows about the context of each page during query execution. It can provide hints to the buffer pool on whether a page is important or not.

Q1 \text{INSERT INTO } A \text{ VALUES } (id++)
BETTER POLICIES: PRIORITY HINTS

The DBMS knows about the context of each page during query execution. It can provide hints to the buffer pool on whether a page is important or not.

Q1

\[
\text{INSERT INTO } A \text{ VALUES (} id++ \text{)}
\]
The DBMS knows about the context of each page during query execution. It can provide hints to the buffer pool on whether a page is important or not.

**Q1**

```
INSERT INTO A VALUES (id++)
```
The DBMS knows about the context of each page during query execution. It can provide hints to the buffer pool on whether a page is important or not.

Q1 \texttt{INSERT INTO A VALUES (id++)}
**BETTER POLICIES: PRIORITY HINTS**

The DBMS knows about the context of each page during query execution. It can provide hints to the buffer pool on whether a page is important or not.

**Q1**
```
INSERT INTO A VALUES (id++)
```

**Q2**
```
SELECT * FROM A WHERE id = ?
```
The DBMS knows about the context of each page during query execution. It can provide hints to the buffer pool on whether a page is important or not.

**Q1**
```
INSERT INTO A VALUES (id++)
```

**Q2**
```
SELECT * FROM A WHERE id = ?
```
DIRTY PAGES

**FAST**: If a page in the buffer pool is **not** dirty, then the DBMS can simply "drop" it.

**SLOW**: If a page is dirty, then the DBMS must write back to disk to ensure that its changes are persisted.

Trade-off between fast evictions versus dirty writing pages that will not be read again in the future.
BACKGROUND WRITING

The DBMS can periodically walk through the page table and write dirty pages to disk.

When a dirty page is safely written, the DBMS can either evict the page or just unset the dirty flag.

Need to be careful that we don’t write dirty pages before their log records have been written...
OTHER MEMORY POOLS

The DBMS needs memory for things other than just tuples and indexes.

These other memory pools may not always be backed by disk. Depends on implementation.

→ Sorting + Join Buffers
→ Query Caches
→ Maintenance Buffers
→ Log Buffers
→ Dictionary Caches
CONCLUSION

The DBMS can almost always manage memory better than the OS.

Leverage the semantics about the query plan to make better decisions:
→ Evictions
→ Allocations
→ Pre-fetching
Hash Tables
PROJECT #1

You will build the first component of your storage manager.
→ LRU Replacement Policy
→ Buffer Pool Manager Instance
→ Parallel Buffer Pool Manager

We will provide you with the disk manager and page layouts.

Due Date:
Sunday Sept 26th @ 11:59pm
**TASK #1 – LRU REPLACEMENT POLICY**

Build a data structure that tracks the usage of pages using the **LRU** policy.

General Hints:

→ Your **LRUReplacer** needs to check the "pinned" status of a **Page**.

→ If there are no pages touched since last sweep, then return the lowest page id.
Use your LRU replacer to manage the allocation of pages.

→ Need to maintain internal data structures to track allocated + free pages.
→ We will provide you components to read/write data from disk.
→ Use whatever data structure you want for the page table.

General Hints:
→ Make sure you get the order of operations correct when pinning.
Task #3 — Multiple Buffer Pools

Approach #1: Object Id
- Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.

Approach #2: Hashing
- Hash the page id to select which buffer pool to access.

Q1: Get Record 123

\[ \text{Buffer Pool #1} \]

\[ \text{Buffer Pool #2} \]

\[ \text{page_id mod num_instances} \]
THINGS TO NOTE

Do not change any file other than the six that you must hand in. Other changes will not be graded.

The projects are cumulative.

We will not be providing solutions.

Post any questions on Piazza or come to office hours, but we will not help you debug.
CODE QUALITY

We will automatically check whether you are writing good code.

→ Google C++ Style Guide
→ Doxygen Javadoc Style

You need to run these targets before you submit your implementation to Gradescope.

→ make format
→ make check-lint
→ make check-censored
→ make check-clang-tidy
EXTRA CREDIT

Gradescope Leaderboard runs your code with a specialized in-memory version of BusTub.

The top 20 fastest implementations in the class will receive extra credit for this assignment.

→ **#1**: 50% bonus points
→ **#2–10**: 25% bonus points
→ **#11–20**: 10% bonus points

Student with the most bonus points at the end of the semester will receive a BusTub shirt!
Your project implementation must be your own work.
→ You may not copy source code from other groups or the web.
→ Do not publish your implementation on GitHub.

Plagiarism will not be tolerated.
See CMU's Policy on Academic Integrity for additional information.