Buffer Pools
ADMINISTRIVIA

Homework #1 is due TODAY @ 11:59pm

Project #1 is due Fri Sept 26th @ 11:59pm
On-Line Transaction Processing (OLTP)
→ Fast operations that only read/update a small amount of data each time.

On-Line Analytical Processing (OLAP)
→ Complex queries that read a lot of data to compute aggregates.

Hybrid Transaction + Analytical Processing
→ OLTP + OLAP together on the same database instance
BIFURCATED ENVIRONMENT

Transactions

OLTP Data Silos

Extract
Transform
Load

Analytical Queries

OLAP Data Warehouse

CMU 15-445/645 (Fall 2019)
BIFURCATED ENVIRONMENT

Transactions
Analytical Queries

HTAP Database

Extract Transform Load

OLAP Data Warehouse
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.
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 minimize the number of stalls from having to read data from disk.
DISK-ORIENTED DBMS

Get page #2

Memory

Buffer Pool

Directory

Header

2

Directory

Header

1

Directory

Header

2

Directory

Header

3

Directory

Header

4

Directory

Header

5

Pages

Execution Engine

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.
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.
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**
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

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
MULTIPLE BUFFER POOLS

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.
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
PRE-FETCHING

Q1

SELECT * FROM A
WHERE val BETWEEN 100 AND 250

Buffer Pool

Disk Pages

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

index-page0

Disk Pages

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

Q1

index-page0

index-page1

index-page2

index-page3

index-page4

index-page5

index-page6
PRE-FETCHING

Buffer Pool

index-page0

index-page1

Disk Pages

index-page0

index-page1

index-page2

index-page3

index-page4

index-page5

index-page6

index-page0

index-page1

index-page2

index-page3

index-page4

index-page5

index-page6

Q1
PRE-FETCHING

Buffer Pool

index-page0
index-page1

Disk Pages

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

Q1
SCAN SHARING

Queries can reuse data retrieved from storage or operator computations.
→ 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 starts a scan and if there one already doing this, then the DBMS will attach to the second query's cursor.
→ The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure.

Fully supported in IBM DB2 and MSSQL. Oracle only supports cursor sharing for identical queries.
SCAN SHARING

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

Buffer Pool

Disk Pages

\[ \text{page0} \]
\[ \text{page1} \]
\[ \text{page2} \]
\[ \text{page3} \]
\[ \text{page4} \]
\[ \text{page5} \]
Q1: \texttt{SELECT SUM(val) FROM A}

**Scan Sharing**

**Buffer Pool**
- page0
- page1
- page2

**Disk Pages**
- page0
- page1
- page2
- page3
- page4
- page5
Q1

SELECT SUM(val) FROM A

Buffer Pool
- page0
- page1
- page2

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

SCAN SHARING
Q1: SELECT SUM(val) FROM A

Buffer Pool:
- page3
- page1
- page2

Disk Pages:
- page0
- page1
- page2
- page3
- page4
- page5
SELECT SUM(val) FROM A

SELECT AVG(val) FROM A

Buffer Pool

Disk Pages

Q1

Q2

page0

page1

page2

page3

page4

page5
SCAN SHARING

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

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

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

Buffer Pool:
- page3
- page1
- page2

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

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: SELECT SUM(val) FROM A

Q2: SELECT AVG(val) FROM A LIMIT 100

Buffer Pool:
- page0
- page1
- page2

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.
OS PAGE CACHE

Most disk operations go through the OS API. Unless you tell it not to, the OS maintains its own filesystem cache.

Most DBMSs use direct I/O (\texttt{O\_DIRECT}) to bypass the OS's cache.
→ Redundant copies of pages.
→ Different eviction policies.

Demo: Postgres
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 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.
Approximation of LRU without needing 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 without needing 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 without needing 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 without needing 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 without needing 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 without needing 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 without needing 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 without needing 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.
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.

The most recently used page is actually the most unneeded page.
SEQUENTIAL FLOODING

Q1: \texttt{SELECT * FROM A WHERE id = 1}

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

Buffer Pool:
- page0
Sequential Flooding

Q1: SELECT * FROM A WHERE id = 1

Q2: SELECT AVG(val) 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

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**

- page1
- page2
- page3

**Disk Pages**

- page0
- page1
- page2
- page3
- page4
- page5
BETTER POLICIES: LRU-K

Track the history of the last $K$ references 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 what 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 what 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 what 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 backed by disk. Depends on implementation.
→ Sorting + Join Buffers
→ Query Caches
→ Maintenance Buffers
→ Log Buffers
→ Dictionary Caches
CONCLUSION

The DBMS can manage that sweet, sweet memory better than the OS.

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

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

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

Due Date:
Friday Sept 27th @ 11:59pm
TASK #1 — CLOCK REPLACEMENT POLICY

Build a data structure that tracks the usage of frame_ids using the CLOCK policy.

General Hints:
→ Your ClockReplacer needs to check the "pinned" status of a Page.
→ If there are no pages touched since last sweep, then return the lowest page id.
**TASK #2 – BUFFER POOL MANAGER**

Use your CLOCK replacer to manage the allocation of pages.

→ Need to maintain an internal data structures of 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.
Download the source code from GitHub.

Make sure you can build it on your machine.
→ We've tested Ubuntu, OSX, and Windows (WSL2).
→ We are also providing a docker file to setup your environment.
→ It does not compile on the Andrews machines. Please contact me if this is a problem.
THINGS TO NOTE

Do **not** change any file other than the four that you must hand in.

The projects are cumulative.

We will **not** be providing solutions.

Post your questions on Piazza or come to our office hours. 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
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.
NEXT CLASS

HASH TABLES!