Lecture 8:
2Q Policy

Logistics

- Point Solutions App
  - Session ID: **database**
- Programming assignment 2 due on **Sep 24** (Gradescope)
- Exercise sheet 1 due on **Sep 24** (Gradescope)

Recap

- Storage Manager
- Buffer Manager

Lecture Overview

- Cache Replacement Policy
- Buffer Pool Flooding
- 2Q Policy

Cache Replacement Policy

Policy Interface

Touch and **Evict** functions encapsulate the essence of any eviction policy

```
class Policy {
public:
    virtual void touch(PageID page_id) = 0;
    virtual PageID evict() = 0;
    virtual ~Policy() = default;
};
```

policy interface	Allows for different interchangeable policies

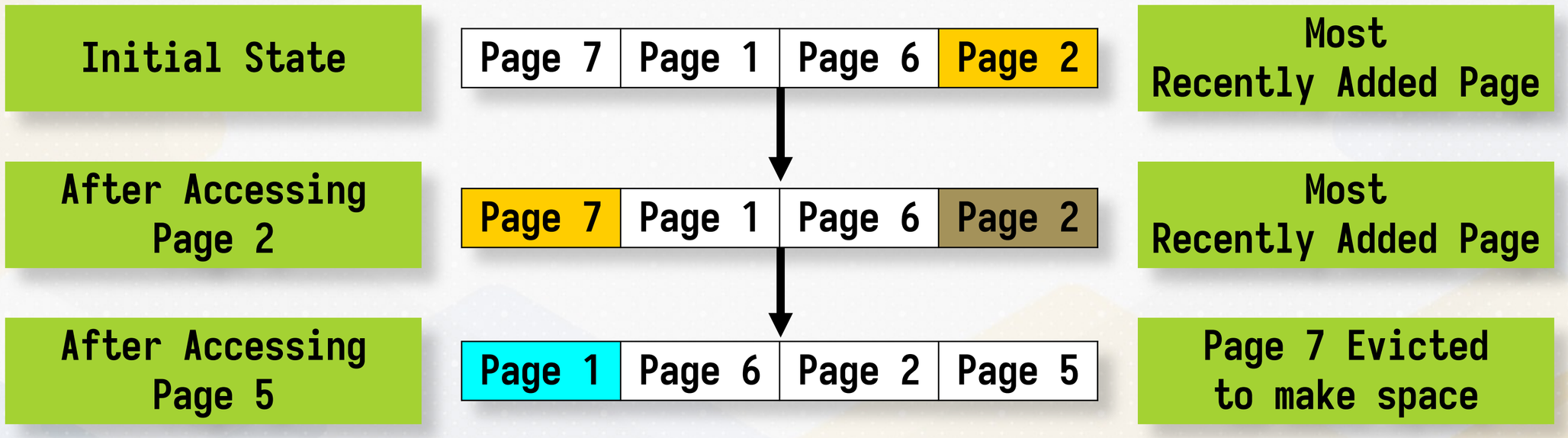
FIFO (First-In-First-Out) Policy

FIFO policy	Evicts the oldest loaded page based on its arrival time

```
class FifoPolicy : public Policy {
    std::queue<PageID> queue;
public:
    void touch(PageID page_id) override {
         auto it = std::find(queue.begin(), queue.end(),
page_id);
         if (it == queue.end()) { queue.push(page_id); }
    }
    PageID evict() override {
        PageID evictedPageId = queue.front(); queue.pop();
        return evictedPageId;
    }
};
```

FIFO (First-In-First-Out) Policy

FIFO policy May evict frequently or recently accessed pages



LRU (Least Recently Used) Policy

LRU policy	Evicts pages that have not been accessed recently

```
class LruPolicy : public Policy {
    std::list<PageID> lruList;
    std::unordered_map<PageID, std::list<PageID>::iterator> map;
public:
    void touch(PageID page_id) override; // Detailed implementation
omitted for brevity
    PageID evict() override; // Evicts the least recently used page
};
```

Integration into Buffer Management

Buffer Manager	Contains a pointer to the **Policy**

```
class BufferManager {
    std::unique_ptr<Policy> policy;
    // Other members omitted for brevity
public:
    BufferManager(std::unique_ptr<Policy> policy) :
policy(std::move(policy)) {}
    // Interface methods omitted for brevity
};
```

Integration into Buffer Management

touch method | Called to update the **policy state**

```
std::unique_ptr<SlottedPage>& BufferManager::getPage(int page_id) {
   if (it == pageMap.end()) { // Page not in memory
        if (pageMap.size() >= MAX_PAGES_IN_MEMORY) {
            PageID evictedPageId = policy->evict(); // Evict page
            ...
        }
    }
    policy->touch(page_id); // Update policy with recent access
    return pageMap[page_id];
}
```

Evict method invoked if **buffer page limit** is reached

Policy vs. Mechanism

policy	The “what”: strategy that decides behavior

POLICY (What)	MECHANISM (How)
Cache Eviction Policy (LRU or FIFO etc.)	Cache Eviction Mechanism (Flush page to disk)

mechanism	The “how”: infrastructure implementing policy

Policy vs. Mechanism: Index Maintenance

POLICY (What)	MECHANISM (How)
Lazy vs Eager Updating	index[key] = value

Buffer Pool Flooding

Sequential Scan

sequential scan	when a database reads every table page in response to a query

```
// Sequential scan across the sales_data table
// Assumption: no index on sale_date
SELECT * FROM sales_data WHERE sale_date BETWEEN ‘2030-01-01' AND ‘2030-01-31';
```

Buffer Pool Flooding

Sequential Scans flood the buffer with pages that may not be accessed again soon

FIFO Policies

Evicts the Oldest Pages from Cache

LRU Policies

Evicts Hot Pages to Access Cold Pages

FIFO: Buffer Pool Flooding

Page Access Trace

1	2	1	2	3	4	1	2	5	6	1	2

Buffer Pool Capacity: 3 Pages

FIFO Evicts **Hot** Pages 1&2

Access Page	1	2	1	2	3	4	1	2	5	6	1	2
FIFO	1	2	2	2	3	4	1	2	5	6	1	2
Queue	-	1	1	1	2	3	4	1	2	5	6	1
	-	-	-	-	1	2	3	4	1	2	5	6
Evict Page						1	2	3	4	1	2	5
Cache Misses	1	2	2	2	3	4	5	6	7	8	9	10

LRU: Buffer Pool Flooding

Page Access Trace

1	2	1	2	3	4	1	2	5	6	1	2

Buffer Pool Capacity: 3 Pages

LRO Also Evicts **Hot** Pages 1&2

Access Page	1	2	1	2	3	4	1	2	5	6	1	2
LRU	1	2	2	2	3	4	1	2	5	6	1	2
Queue	-	1	1	1	2	3	4	1	2	5	6	1
	-	-	-	-	1	2	3	4	1	2	5	6
Evict Page						1	2	3	4	1	2	5
Cache Misses	1	2	2	2	3	4	5	6	7	8	9	10

2Q Policy

TwoQ Policy

page transition logic	• Page initially enters **FIFO** queue • Page moves to **LRU** queue upon second access

FIFO Queue

Read-Once Pages Accessed During Sequential Scan

LRU Queue

Frequently-Accessed **Hot** Pages Promoted to LRU Queue

TwoQ Policy: touch

Pages Enter **FIFO** Queue

Pages Promoted To **LRU**

Page Position Updated When Accessed

```
void touch(PageID page_id) {
    if (pageMap.find(page_id) != pageMap.end()) {
        auto it = pageMap[page_id];
        // If in FIFO and accessed again, move to LRU
        // Otherwise, adjust position within LRU
        // New pages are added to FIFO unless cache
is full, then evict
    }
}
```

TwoQ Policy: touch

Check **FIFO** Queue When Page Accessed

Page in FIFO Queue Relocated to LRO Queue

Page in LRO Queue Reflects Higher Reuse Probability

```
void touch(PageID page_id) {
    // If the page is already in the cache
    if (pageMap.find(page_id) != pageMap.end()) {
        auto it = pageMap[page_id];
        // If it's in FIFO, move to LRU
        if (std::find(FIFO.begin(), FIFO.end(),
page_id) != FIFO.end()) {
            FIFO.erase(it);
            LRU.push_front(page_id);
            pageMap[page_id] = LRU.begin();
        }
```

TwoQ Policy: touch

Pages Accessed Repeatedly in **LRU** Queue Evicted Last

```
void touch(PageID page_id) {
    // If the page is already in the cache
    if (pageMap.find(page_id) != pageMap.end()) {
        ...
        else {
            // If it's in LRU, move to the front
            LRU.erase(it);
            LRU.push_front(page_id);
            pageMap[page_id] = LRU.begin();
        }
    }
```

TwoQ Policy: touch

```
void touch(PageID page_id) {
    // If the page is not in the
cache
    else {
        // If cache is full, evict
        if (FIFO.size() +
LRU.size() >= cacheSize) {
            evict();
        }
        // Add page to FIFO
        FIFO.push_back(page_id);
        pageMap[page_id] =
std::prev(FIFO.end());
    }
```

TwoQ Policy: evict

TwoQ evict

- Prioritizes Eviction from **FIFO** Queue
- Minimizes Cache Pollution from Sequential Scans

```
PageID evict() {
    // FIFO pages are evicted first to minimize cache
pollution,
    // followed by the least recently used pages in LRU
}
```

TwoQ Policy: evict

```
PageID evict() {
    PageID evictedPageId =
INVALID_VALUE;
    if (!FIFO.empty()) {
        evictedPageId = FIFO.front();
        FIFO.pop_front();
pageMap.erase(evictedPageId);
    } else if (!LRU.empty()) {
        evictedPageId = LRU.back();
        LRU.pop_back();
pageMap.erase(evictedPageId);
    }
    return evictedPageId;
}
```

TwoQ: No Buffer Pool Flooding

TwoQ retains **hot** pages like 1 and 2

Access Page	1	2	1	2	3	4	1	2	5	6	1	2
FIFO	1	2	2	_	3	4	4	4	5	6	6	6
Queue	-	1	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	-	-	-	-
LRU	-	-	1	2	2	2	1	2	2	2	1	2
Queue	-	-	-	1	1	1	2	1	1	1	2	1
	-	-	-	-	-	-	-	-	-	-	-	-
Evict Page						3			4	5		
Cache Misses	1	2	2	2	3	4	4	4	5	6	6	6

More Policies: LFU

LFU Policy Benefits

Keeps Frequently-Used Pages in Memory

LFU Policy Limitations

Does Not Consider Changes to Recent Access Patterns

More Policies: ARC

Arc Policy Benefits

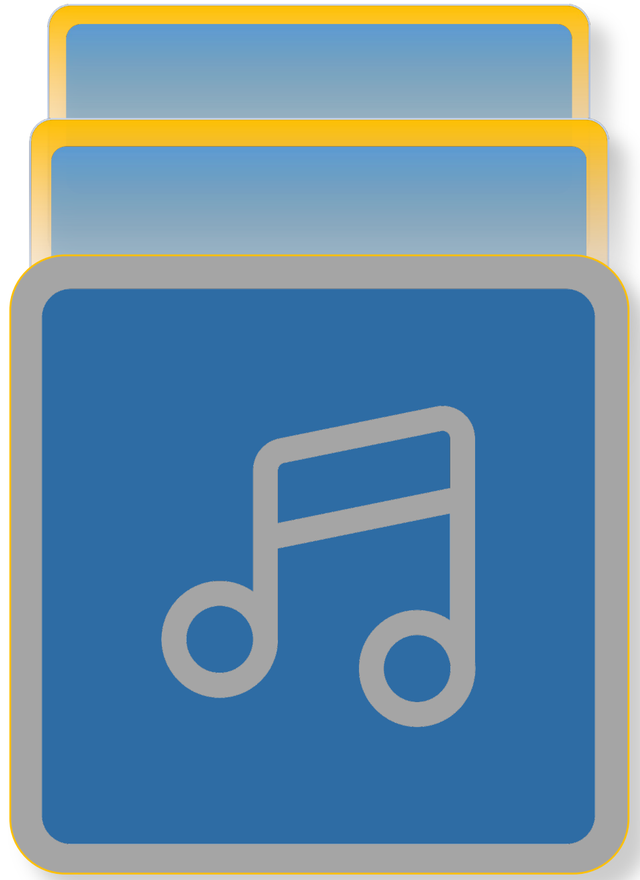
Dynamically Balances between LRU and LFU

ARC Policy Limitations

Requires Complex Implementation

Music Playlist + Cache Eviction Policies

LRU	Keeps Recently Played Songs
TwoQ	Keeps Mix of Favorites and New Items



Conclusion

- Cache Replacement Policy
- Buffer Pool Flooding
- 2Q Policy