Lecture 7: Storage and Buffer Management

Logistics

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

Recap

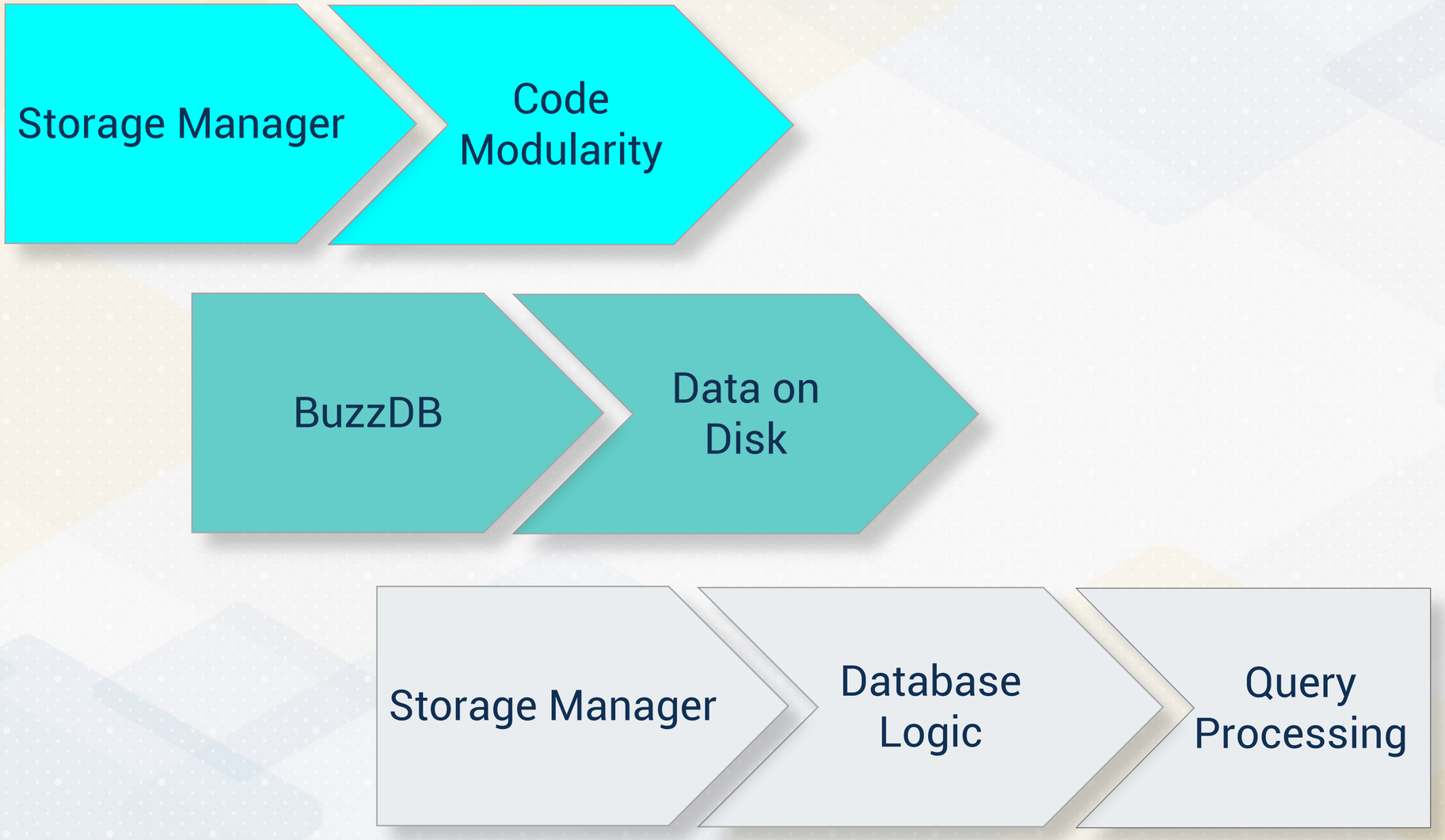
- File Management
- Index Construction
- Casting in C++
- Streams in C++

Lecture Overview

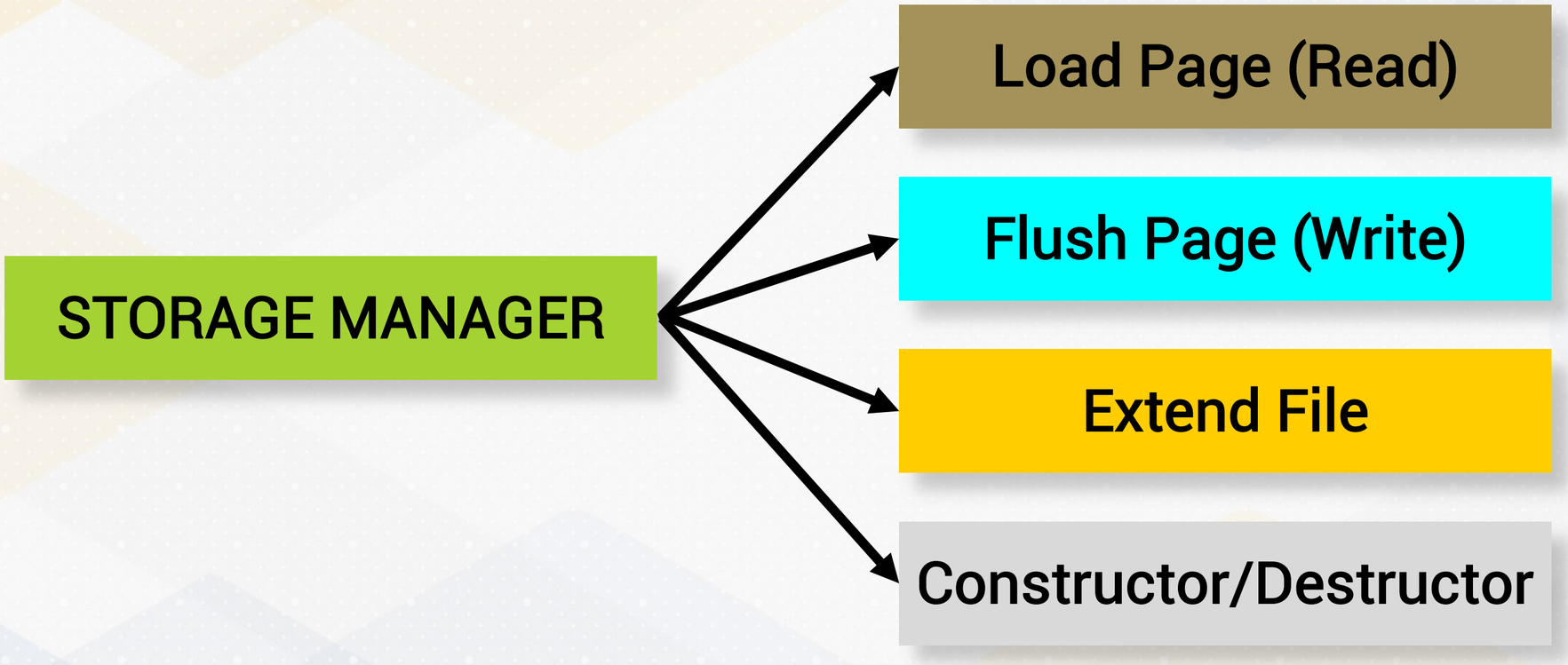
- Storage Manager
- Buffer Manager
- Cache Replacement Policy

Storage Manager

Storage Manager



Storage Manager



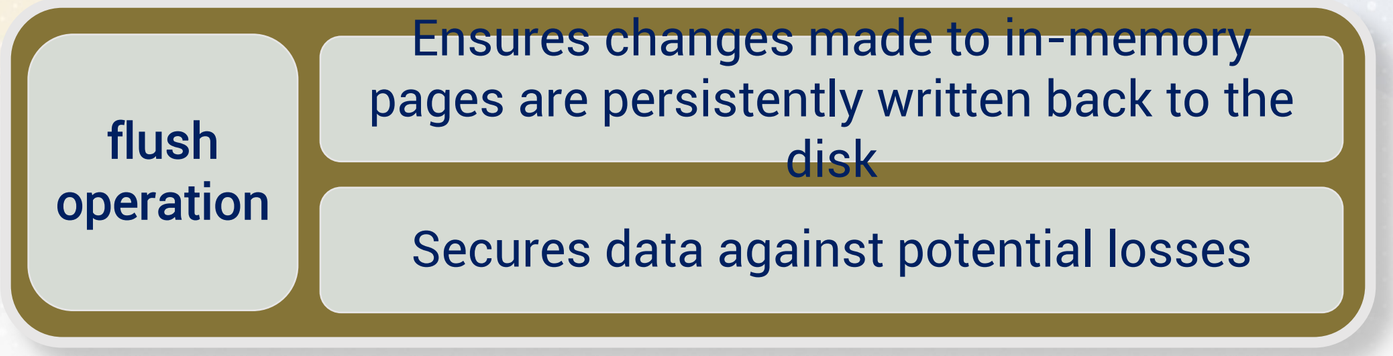
Storage Manager

```
StorageManager::StorageManager() {
    fileStream.open(database_filename,
std::ios::in | std::ios::out);
    if (!fileStream) {
        fileStream.clear(); // Reset the state
        fileStream.open(database_filename,
std::ios::out);
        fileStream.close();
    }
    fileStream.open(database_filename,
std::ios::in | std::ios::out);
    // Calculate number of pages
    fileStream.seekg(0, std::ios::end);
    num_pages = fileStream.tellg() / PAGE_SIZE;
}
```

Dynamic File Extension


```
void StorageManager::extend() {
    auto empty_slotted_page = std::make_unique<SlottedPage>();
    fileStream.seekp(0, std::ios::end);
    fileStream.write(empty_slotted_page->page_data.get(), PAGE_SIZE);
    fileStream.flush();
    num_pages += 1;
}
```

Data Persistence




```
void StorageManager::flush(uint16_t page_id) {
    size_t page_offset = page_id * PAGE_SIZE;
    fileStream.seekp(page_offset, std::ios::beg);
    fileStream.write(pages[page_id]->page_data.get(), PAGE_SIZE);
    fileStream.flush();
}
```

Data Loading


```
std::unique_ptr<SlottedPage> load(uint16_t page_id) {
    fileStream.seekg(page_id * PAGE_SIZE, std::ios::beg);
    auto page = std::make_unique<SlottedPage>();
    // Read the content of the file into the page
    if(fileStream.read(page->page_data.get(), PAGE_SIZE)){
        //std::cout << "Page read successfully from file." << std::endl;
    }
    else{
        std::cerr << "Error: Unable to read data from the file. \n";
        exit(-1);
    }
    return page;
}
```

Storage Manager Destructor

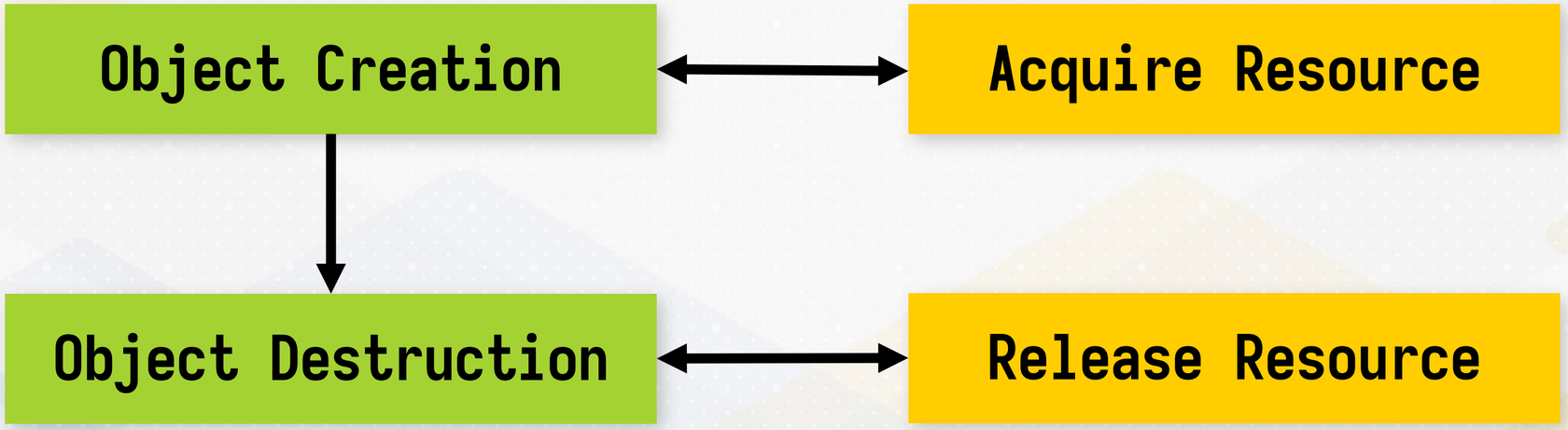
Destructor ensures that the open file stream is correctly closed and cleaned up

```
StorageManager::~StorageManager() {
    if (fileStream.is_open()) {
        fileStream.close();
    }
}
```

RALL in C++

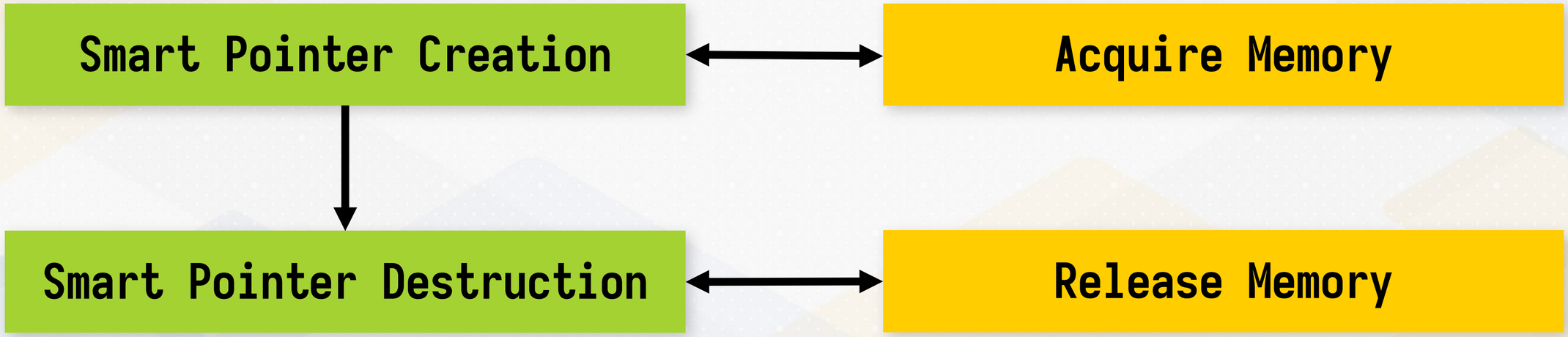
RAII (Resource Acquisition Is Initialization)

a programming idiom that binds the lifecycle of resources (i.e., file handles and memory) to object lifetimes



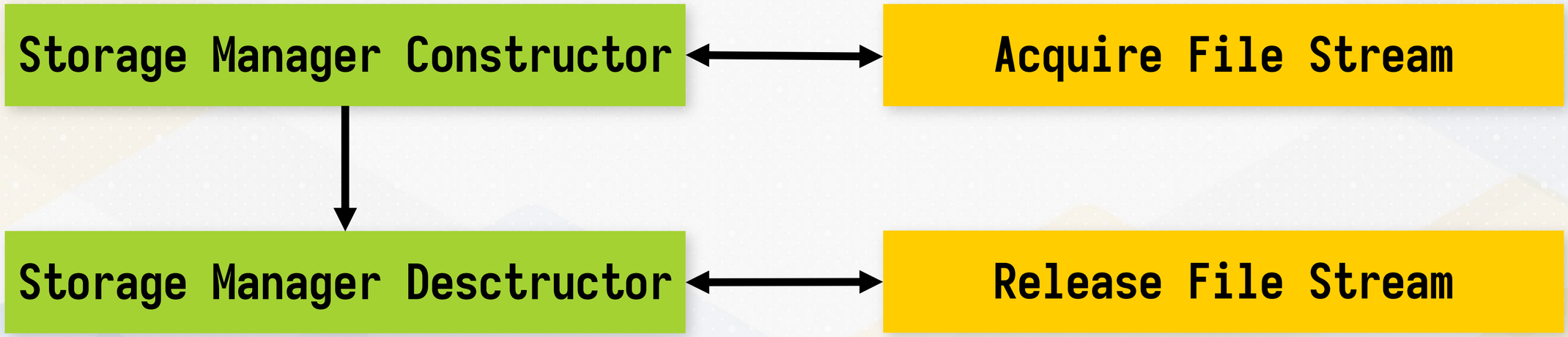
RALL in C++: Smart Pointers

- **Smart pointers** illustrate the RAII principle
- Resource lifetime (memory) is tied to the scope of the smart pointer, ensuring **deterministic** and **automatic** resource cleanup



RALL in C++: File Handles

The **Storage Manager** also illustrates the RAII principle.



Integration into BuzzDB

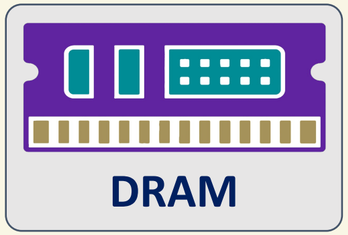
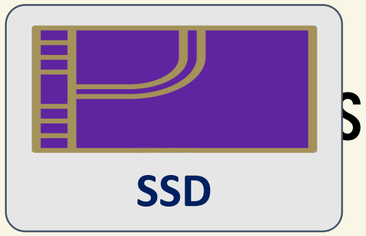
buzzDB

Utilizes the **Storage Manager** to streamline the process of inserting tuples into the database

```
bool BuzzDB::try_to_insert(int key, int value) {
    for (size_t page_id = 0; page_id < sm.num_pages; ++page_id) {
        auto& page = sm.pages[page_id]; // Access page via
Storage Manager
        if (page->addTuple(/* tuple data */)) {
            sm.flush(page_id); // Persist changes to disk
            return true; // Successful insertion
        }
    }
    return false; // Indicate failure to insert due to full pages
}
```

Buffer Management

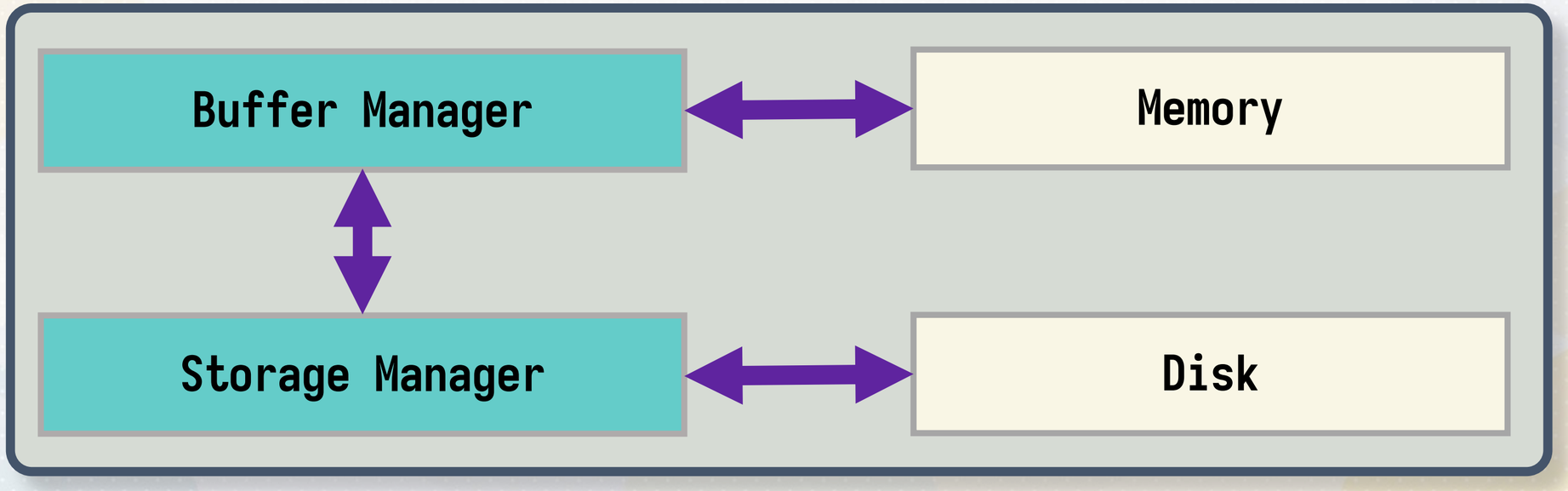
DRAM vs SSD vs HDD

Device	Latency (ns)	Price per GB ($)
 Dynamic RAM (DRAM)	50	10
 Solid State Disk (SSD)	50 * 1000	0.1

Buffer Manager

Buffer Manager Caches Frequently-Accessed Pages

Storage Manager: low-level operations

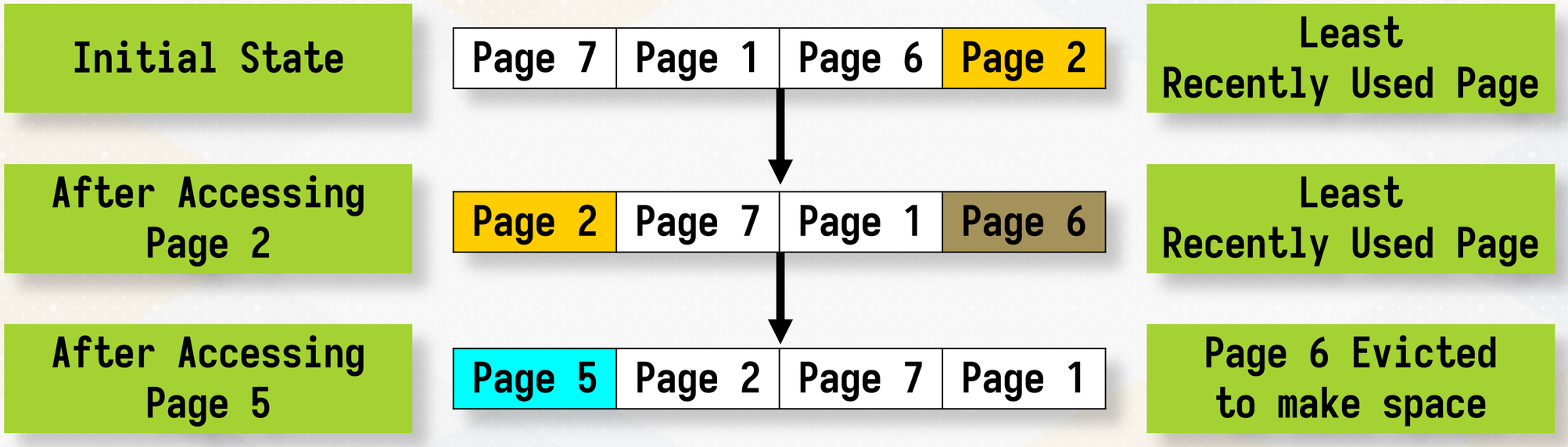


Buffer Manager

Buffer Manager: Maintains **Page List** in Memory, Mapped by IDs for Quick Access

```
class BufferManager {
private:
    StorageManager storage_manager;
    // LRU list to maintain the order of page usage
    std::list<std::pair<PageID, std::unique_ptr<SlottedPage>>>
lruList;
    // Map to quickly find pages in the list
    std::unordered_map<PageID, typename PageList::iterator>
pageMap;
public:
    // Fetches a page with given ID, applying LRU policy for
caching
    std::unique_ptr<SlottedPage>& getPage(int page_id);
};
```

Least Recently Used (LRU) Policy



Buffer Manager

PageID

Utilized for page identification

PageList

Stores currently loaded pages in LRU order

PageMap

mapping from PageID to iterators of the PageList

```
class BufferManager {
private:
    StorageManager storage_manager;
    // LRU list to maintain the order of page usage
    std::list<std::pair<PageID,
std::unique_ptr<SlottedPage>>> lruList;
    // Map to quickly find pages in the list
    std::unordered_map<PageID, typename
PageList::iterator> pageMap;
};
```

Buffer Manager & Storage Manager

StorageManager: Loads Page from Disk when Page Not Found in Buffer Cache

```
std::unique_ptr<SlottedPage>& BufferManager::getPage(int page_id) {
    auto it = pageMap.find(page_id);
    if (it == pageMap.end()) {
        // Page not in cache, load from disk
        auto page = storage_manager.load(page_id);
        ...
    }
    touch(pageMap[page_id]);
    return lruList.begin()->second;
}
```

touch

touch

- Updates a page's position in the **LRU list** to reflect its recent access
- The splice method in **std::list** is used to move the page to the front of the list

```
void touch(PageList::iterator it) {
    // Move page to the front of the list, indicating recent use
    lruList.splice(lruList.begin(), lruList, it);
}
```

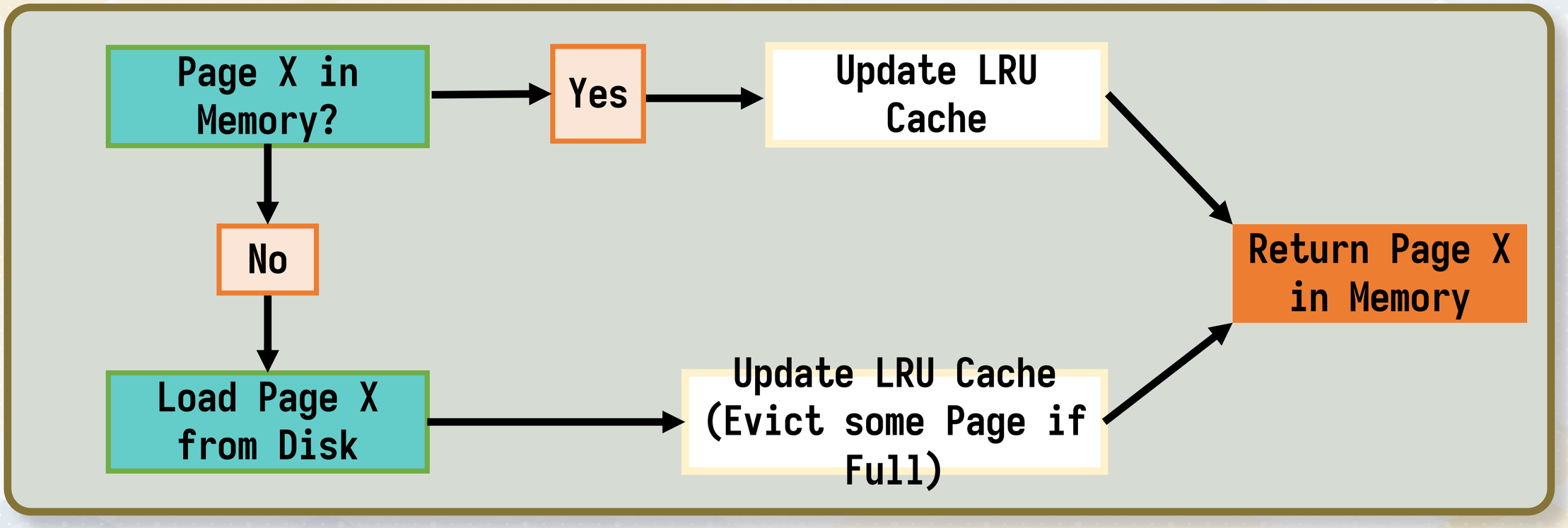
evict

evict

- Removes Least Recently Used Page from Buffer
- Changes Flushed to Ensure No Data Lost

```
void evict() {
    auto last = --lruList.end(); // Get iterator to the
last element
    int evictedPageId = last->first;
    // Flush changes to disk before eviction if necessary
    storage_manager.flush(evictedPageId, last->second);
    // Remove from page map and LRU list
    pageMap.erase(evictedPageId);
    lruList.pop_back();
}
```

Buffer Manager



Modularity

buzzDB

- Interacts Only with **BufferManager** for Page-Related Operations
- BuzzDB Never Accesses **StorageManager**

```
// BuzzDB now uses BufferManager for page operations
void BuzzDB::insert(int key, int value) {
    ...
    auto& page = buffer_manager.getPage(page_id); // Interacts through BufferManager
    page->addTuple(std::move(newTuple));
    ...
}
```

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';
```

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

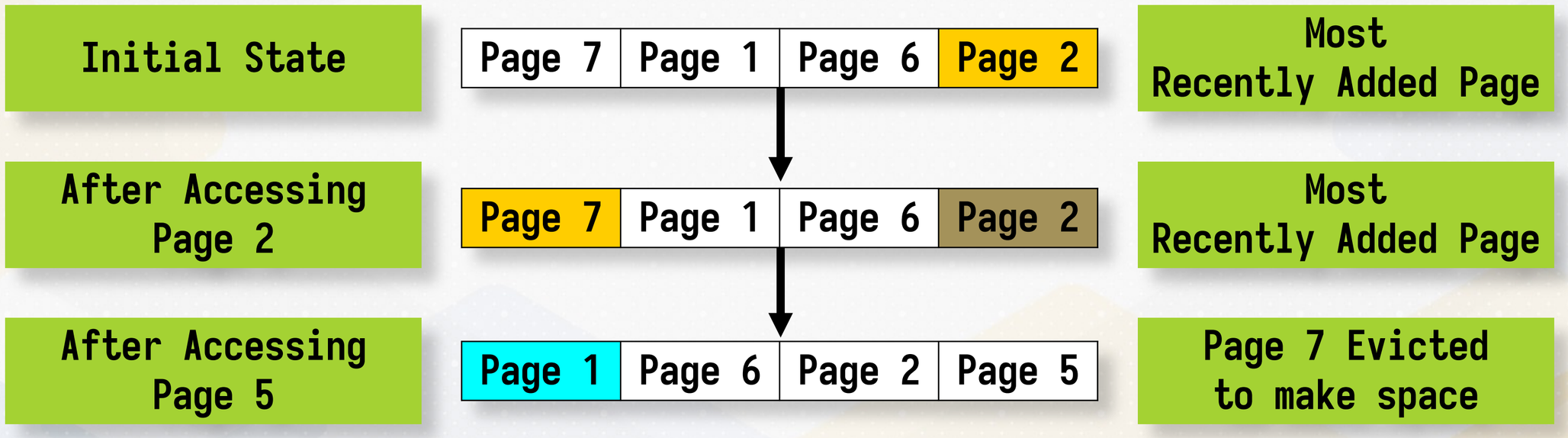
FIFO (First-In-First-Out) Policy

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

Conclusion

- Storage Manager
- Buffer Manager
- Cache Replacement Policy