

## OVERVIEW

During the semester, you will be building a new disk-oriented storage manager for the [SQLite](#) DBMS. Such a storage manager assumes that the primary storage location of the database is on disk. You will be using SQLite's [Virtual Table](#) interface; this will allow you use your storage manager in SQLite without changing application-level code. You do not need to know exactly how SQLite works to complete these assignments.

The first programming project is to implement a buffer pool in your storage manager. The buffer pool is responsible for moving physical pages back and forth from main memory to disk. It allows a DBMS to support databases that are larger than the amount of memory that is available to the system. Its operations are transparent to other parts in the system. For example, the system asks the buffer pool for a page using its unique identifier ( [page\\_id](#) ) and it does not know whether that page is already in memory or whether the system has to go retrieve it from disk.

Your implementation will need to be thread-safe. Multiple threads will be accessing the internal data structures at the same and thus you need to make sure that their critical sections are protected with [latches](#) (these are called "locks" in operating systems). You will need to implement the following three components in your storage manager:

- [Extendible Hash Table](#)
- [LRU Page Replacement Policy](#)
- [Buffer Pool Manager](#)

Although SQLite is written in C, all of the code in this programming assignment must be written in C++ (specifically C++11). We are providing you with the scaffolding to have your C++ code build and link with SQLite's virtual table API.

**This is a single-person project that will be completed individually (i.e., no groups).**

## PROJECT SPECIFICATION

For each of the following components, we are providing you with stub classes that contain the API that you need to implement. You should not modify the signatures for the pre-defined functions in these classes. If you do this, then it will break the test code that we will use to grade your assignment you end up getting no credit for the project. You should also not add additional classes in the source code for these components. That is, they should be entirely self-contained. Again, this is necessary to ensure that your implementation is compatible with our testing infrastructure.

If a class already contains data members, you should not remove them. For example, the `BufferPoolManager` contains `DiskManager` and `Replacer` objects. These are required to implement the functionality that is needed by the rest of the system. You may need to add data members to these classes in order to correctly implement the required functionality. You can also add additional helper functions to these classes. This choice is up to you.

You are allowed to use any built-in C++11 containers in your project unless specified otherwise. It is up to you to decide which ones you want to use. Note that these containers are not thread-safe and that you will need to include latches in your implementation to protect them. You may not bring in additional third-party dependencies (e.g., boost).

## TASK #1 - EXTENDIBLE HASH TABLE

For the first part of this project, you will build a general purpose hash table that uses unordered buckets to store unique key/value pairs. Your hash table must support the ability to insert/delete key/value entries without specifying the max size of the table. Your table needs to automatically grow in size as needed but you do not need shrink it. That is, you do not need to implement support for shrinking or compacting the hash table. You will also need to support checking to see whether a key exists in the hash table and return its corresponding value.

You must implement your hash table in the designated files in the project source code. You are only allowed to modify the hash table header file (`src/include/hash/extendible_hash.h`) and its corresponding implementation file (`src/hash/extendible_hash.cpp`). You do not need to modify any other files. You may not use another built-in hash table internally in your implementation. You must implement the following functions in the `ExtendibleHashTable` class:

- `HashKey(K)` : For the given key `K`, return the offset of the Bucket where it should be stored.
- `Find(K, V)` : For the given key `K`, check to see whether it exists in the hash table. If it does, then store the pointer to its corresponding value in `V` and return true. If the key does not exist, then return false.
- `Insert(K, V)` : Insert the key/value pair into the hash table. If the key `K` already exists, overwrite its value with the new value `V` and return true.
- `Remove(K)` : For the given key `K`, remove its corresponding key/value pair from the hash table and return true. If the key `K` does not exist in the hash table, then return false.
- `GetGlobalDepth()` : Return the current global depth of the entire hash table.
- `GetLocalDepth(bucket_id)` : Return the current local depth for the bucket at the given offset.
- `GetNumBuckets()` : Return the total number of buckets allocated in the hash table.

You need to make sure that all operations in the hash table are thread-safe using `std::mutex`. It is up to you to decide how you want to protect the data structure.

## TASK #2 - LRU PAGE REPLACEMENT POLICY

This component is responsible for tracking the usage of pages in the buffer pool. You will implement a new sub-class called `LRUReplacer` in `src/include/buffer/lru_replacer.h` and its corresponding implementation file in `src/buffer/lru_replacer.cpp`. This is a generic class that is used to keep track of when elements that it is tracking are used. You will need to implement the least-recently used policy discussed in the class and the textbook.

Your new class will extend the abstract `Replacer` class (`src/include/buffer/replacer.h`). You will need to implement the following functions:

- `Insert(T)` : Mark the element `T` as having been accessed in the database. This means that the element is now the most frequently accessed and should not be selected as the victim for removal from the buffer pool (assuming there exists more than one element).
- `Victim(T)` : Remove the object that was accessed the least recently compared to all the elements being tracked by the Replacer, store its contents in the value `T`, and then return `true`. If there is only one element in the Replacer, then that is always considered to be the least recently used. If there are zero elements in the Replacer, then this function should return `false`.
- `Erase(T)` : Completely remove the element `T` from the Replacer's internal tracking data structure regardless of where it appears in the LRU replacer. This should delete all tracking data from the element. If the element `T` exists and it was removed, then the function should return `true`. Otherwise, return `false`.
- `Size()` : Return the number of elements that this Replacer is tracking.

It is up to you to decide how you want to implement the data structures to store the meta-data about the elements inside of `LRUReplacer`. For example, you can use the `ExtendibleHashTable` that you built in the first task or use a built-in STL container. You do not need to worry about a maximum size of the data structures. You can assume that you will not run out of memory. Again, you need to make sure that the operations are thread-safe.

### TASK #3 - BUFFER POOL MANAGER

Lastly, you need to implement the buffer pool manager in your system ( `BufferPoolManager` ). It is responsible for fetching database pages from the `DiskManager` and storing them in memory. The `BufferPoolManager` can also write dirty pages out to disk when it is either explicitly instructed to do so or when it needs to evict a page to make space for a new page.

To make sure that your implementation works correctly with the rest of the system, we will provide you with some of the functions already filled in. You will also not need to implement the code that actually reads and writes data to disk (this is called the `DiskManager` in our implementation). We will provide that functionality for you.

All in-memory pages in the system are represented by `Page` objects. The `BufferPoolManager` does not need to understand the contents of these pages. But it is important for you as the system developer to understand that `Page` objects are just containers for memory in the buffer pool and thus are not specific to a unique page. That is, each `Page` object contains a block of memory that the `DiskManager` will use as a location to copy the contents of a physical page that it reads from disk. The `BufferPoolManager` will reuse the same `Page` object to store data as it moves back and forth to disk. This means that the same `Page` object may contain a different physical page throughout the life of the system. The `Page` object's identifier ( `page_id` ) keeps track of what physical page it contains; if a `Page` object does not contain a physical page, then its `page_id` must be set to `INVALID_PAGE_ID`.

Each `Page` object also maintains a counter for the number of threads that have "pinned" that page. Your `BufferPoolManager` is not allowed to free a `Page` that is pinned. Each `Page` object also keeps track of whether it is dirty or not. It is your job to record whether a page was modified when it is unpinned. Your `BufferPoolManager` must write the contents of a dirty `Page` back to disk before that object can be reused.

Your `BufferPoolManager` implementation will use the `ExtendibleHashTable` and `LRUReplacer` classes that you created in the previous steps of this assignment. It will use the `ExtendibleHashTable` for the table that maps `page_id` 's to `Page` objects. It will also use the `LRUReplacer` to keep track of when `Page` objects are accessed so that it can decide which one to evict when it must free a frame to make room for copying a new physical page from disk.

You will need to implement the following functions defined in the header file (`src/include/buffer/buffer_pool_manager.h`) in the source file (`src/buffer/buffer_pool_manager.cpp`):

- **FetchPage(page\_id)** : This returns a Page object that contains the contents of the given page\_id. The function should first check its internal page table to see whether there already exists a Page that is mapped to the page\_id. If it does, then it returns it. Otherwise it will retrieve the physical page from the DiskManager. To do this, the function needs to select a Page object to store the physical page's contents. If there are free frames in the page table, then the function will select a random one to use. Otherwise, it will use the LRUReplacer to select an unpinned Page that was least recently used as the "victim" page. If there are no free slots (i.e., all the pages are pinned), then return a null pointer (nullptr). If the selected victim page is dirty, then you will need to use the DiskManager to write its contents out to disk. You will then use the DiskManager to read the target physical page from disk and copy its contents into that Page object.  
IMPORTANT: This function must mark the Page as pinned and remove its entry from LRUReplacer before it is returned to the caller.
- **NewPage(page\_id)** : Allocate a new physical page in the DiskManager, store the new page id in the given page\_id and store the new page in the buffer pool. This should have the same functionality as FetchPage() in terms of selecting a victim page from LRUReplacer and initializing the Page's internal meta-data (including incrementing the pin count).
- **UnpinPage(page\_id, is\_dirty)** : Decrement the pin counter for the Page specified by the given page\_id. If the pin counter is zero, then the function will add the Page object into the LRUReplacer tracker. If the given is\_dirty flag is true, then mark the Page as dirty; otherwise, leave the Page's dirty flag unmodified. If there is no entry in the page table for the given page\_id, then return false.
- **FlushPage(page\_id)** : This will retrieve the Page object specified by the given page\_id and then use the DiskManager to write its contents out to disk. Upon successful completion of that write operation, the function will return true. This function should not remove the Page from the buffer pool. It also does not need to update the LRUReplacer for the Page. If there is no entry in the page table for the given page\_id, then return false.
- **FlushAllPages()** : For each Page object in the buffer pool, use the DiskManager to write their contents out to disk. This function should not remove the Page from the buffer pool. It also does not need to update the LRUReplacer for the Page.
- **DeletePage(page\_id)** : Instruct the DiskManager to deallocate the physical page identified by the given page\_id. You can only delete a page if it is currently unpinned.

## TESTING

You can test the individual components of this assignment using our testing framework. We use **GTest** for unit test cases. There are three separate files that contain tests for each component:

- **ExtendibleHashTable** : `test/hash/extendible_hash_test.cpp`
- **LRUReplacer** : `test/buffer/lru_replacer_test.cpp`
- **BufferPoolManager** : `test/buffer/buffer_pool_manager_test.cpp`

You can compile and run each test individually from the command-line:

```
$ mkdir build
$ cd build
$ make extendible_hash_test
$ ./test/extendible_hash_test
```