# CS 44800: Project 1

**Aryan Wadhwani**

# Task 1: Deciding Between MRU and LRU

I believe Most Recently Used will perform better, because we can expect that viewing a certain block of memory makes that block of memory less like to be viewed later. I believe having the unmodified buffers first will give us a valuable runtime difference as well, since we otherwise may select a modified buffer, and would hence lose time by having to flush the modified contents to disk.

# Task 2: Design and Implement

I’ve selected Most Recently Used. For specific details about the implementation, see *README.md*.

# Task 3: Measurement and Evaluation

**Measuring hits and misses**

Since we want to move past wall-clock times, which vary across systems and are subject to changes due to file system caching and background processes, we will look at the portion of the code that will generally take the longest time: accessing the disk. Accessing the disk would be the longest operation, in comparison to the relatively quicker operations for data in the memory. Hence, the biggest determinant for the performance of our buffer would be the number of times we end up going to disk.

When we attempt to pin a block to the buffer, what determines whether we end up going to disk is whether the buffer exists in our buffer pool or not. If we have a buffer with the requested block in our pool this is considered a **hit**, since we don’t have to read data from the disc. If we have to read from disc, that is considered a **miss.**

**Implementing MRU**

With the provided specification, we need to construct two lists, one for unmodified-unpinned buffers and modified-pinned buffers. When performing MRU, priority is given to unmodified-unpinned buffers. Hence, we have three problems to solve:

1. **Storing unmodified-unpinned and modified-unpinned buffers:** We have two lists *unmodifiedPool and modifiedPool* to store the two buffers. When the Buffer Manager is created, we begin with all of the buffers in *unmodifiedPool*.
2. **Selecting the correct Buffer when pin() is called:** If we try to pin a block and the buffer pool doesn’t contain said block, we need to select a buffer pin the aforementioned block to. If a buffer is already pinned though, that buffer cannot be used since other transactions require that buffer. Hence, we have a method to select from unpinned buffers. As described above, we will first select the last buffer in *unmodifiedPool*, but if it is empty, we will select the last buffer in *modifiedPool*.
3. **Adding a buffer to the correct list when unpin() is called:** When unpinning a buffer leads it to be completely unpinned, which is to say that there are no transactions using the block inside this buffer, we can add it to one of the free buffer pools. To determine which pool to add this buffer to, we need to figure out if the buffer was modified or not. Looking at *Buffer.java*, we can observe in the *setModified* method:

public void setModified(int txnum, int lsn) {  
 this.txnum = txnum;  
 if (lsn >= 0)  
 this.lsn = lsn;  
}

When the contents of the buffer are modified, we attach a positive transaction number to it. Hence, we can determine whether the contents are modified by not by checking if *txnum* is positive or not. For further evidence of this, we can observe that while flushing:

void flush() {  
 if (txnum >= 0) {  
 lm.flush(lsn);

BufferMgr.*discWrites*++;  
 fm.write(blk, contents);  
 txnum = -1;  
 }  
}

We only flush the contents of the block if the *txnum* is positive. In other cases, doing so would be pointless, since if the transaction has not modified the contents, there is no need to rewrite the same information. Additionally, to observe differences in the number of writes due to our optimization, we can keep increment the number of *discWrites* here.

**Analyzing Results**

Text, whiteboard

Description automatically generatedA screenshot of a computer

Description automatically generated with low confidenceA picture containing shape

Description automatically generatedFor comparing between MRU and the default implementation, I analyzed the data for hits, misses, disc writes and the hits to misses ratio for various SQL queries. Here are the resulting graphs:

A picture containing text, screenshot

Description automatically generatedA picture containing shape

Description automatically generatedIcon

Description automatically generated with medium confidenceA picture containing text, screenshot

Description automatically generated

We have several takeaways from these graphs:

1. MRU makes great improvements for Join operations, which is visible by the massive four-fold jump in the hits/misses ratio,
2. MRU has more decent smaller improvements for Larger Joins and Delete operations, which can be seen by the MRU hits/misses ratio being larger, but decreasingly larger as *n* grows.
3. MRU slightly underperforms for Select, Update and Create operations, which is visible by the default implementation having a slightly larger hits/misses ratio.
4. In the average case, MRU greatly underperforms the default case. This can be seen by the difference in the hits/misses ratio being in the factor 3.

**Task 4:**