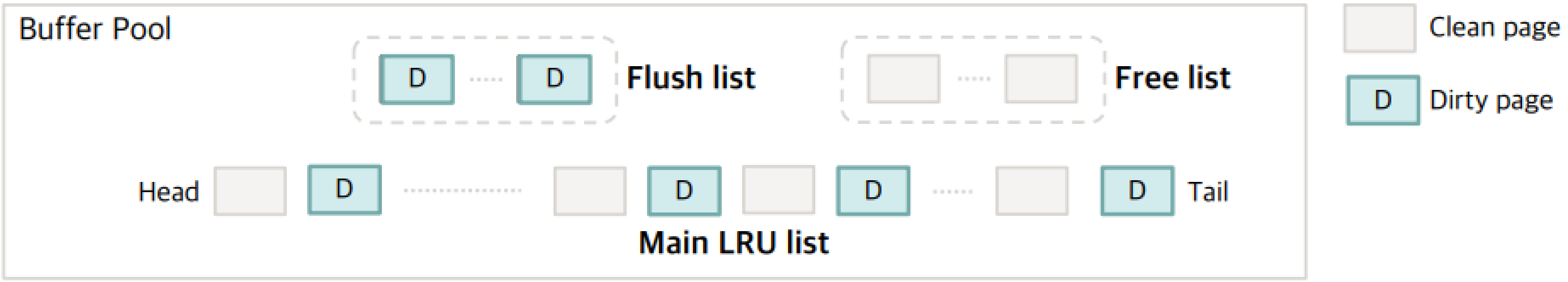
Buffer Management of MySQL/InnoDB (2) Write Requests Part 2

Bo-Hyun Lee
lia323@skku.edu

Index

- Three Types of Disk Write
- InnoDB Redo Log, LSN, and Checkpoint
- Flush List Flush
- Innodb parameters
- PA1

Lists of Buffer Blocks



- **Free list**
  - Contains free (empty) buffer frames
- **LRU list**
  - Contains all the blocks holding a file page
- **Flush list**
  - Contains the blocks holding file pages that have been modified in the memory but not written to disk yet (i.e., **dirty**)

Three Types of Disk Writes

- **Single Page Flush:**
  - A single write request issued by the foreground user process
  - Used as a victim for replacement
- **LRU Tail Flush:**
  - Asynchronous write requests issued by the background process
  - For cold page eviction
- **Flush List Flush (i.e., fuzzy checkpoint):**
  - Asynchronous write requests issued by the background process
  - For database recovery upon failure

Disk write type 3:

Flush List Flush

What is the Role of REDO Log?

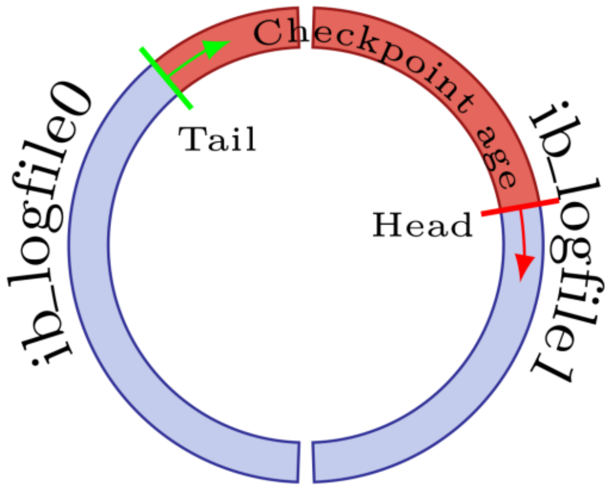
- Each time data is changed, the page containing the data is modified in memory

- The page is denoted as ***dirty***

- In case of an unexpected failure, we cannot lose all the changes
  - *But the data in memory is gone!*

- This is why diff data of the page is also written (and by default flushed to disk) to **REDO logs**
  - The data in REDO logs will be read only in case of recovery
  - During recovery, the modified pages will be reconstructed with the modified data

InnoDB Redo Log



- What people mean by InnoDB log (e.g., in `my.cnf`)
  - `Innodb_log_file_size=1G`
  - `Innodb_log_files_in_group=2`
- Two or more redo logs pre-allocated and used in a circular fashion
  - `ib_logfile1 ib_logfile2` files in `/path/to/test-data`
- Information necessary to redo (or re-apply) changes to data stored in InnoDB
- Used to reconstruct changes if necessary (i.e., crash recovery)

Log Sequence Number (LSN)

- A 64-bit unsigned integer representing a point in time in the redo log system
  - **The total number of log data that have been generated since log initialization**

- An ever-increasing number similar to the number of bytes logged

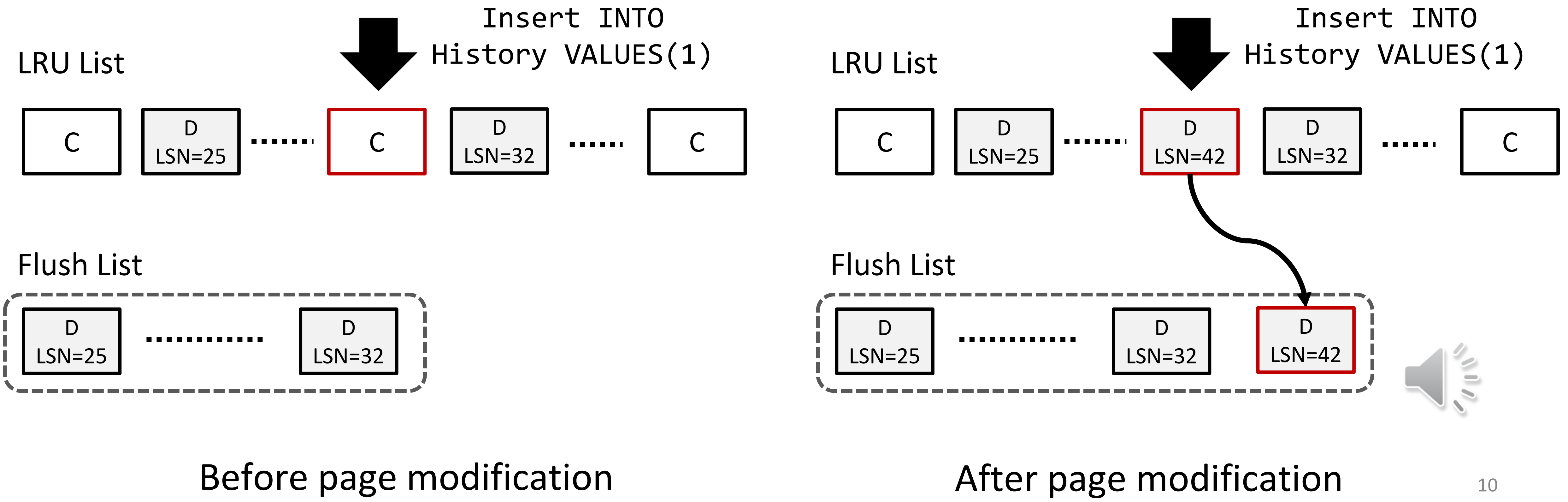
- You can leverage the **LSN** to locate a log data in a log file
  - LSN: *`lsn_t`* *type variable*

Checkpoint

- The LSN value of the latest changes written to the data files
- Checkpoint means that changes made prior to the checkpoint LSN have been flushed
- Once a checkpoint has been completed, redo logs prior to the checkpoint are no longer needed

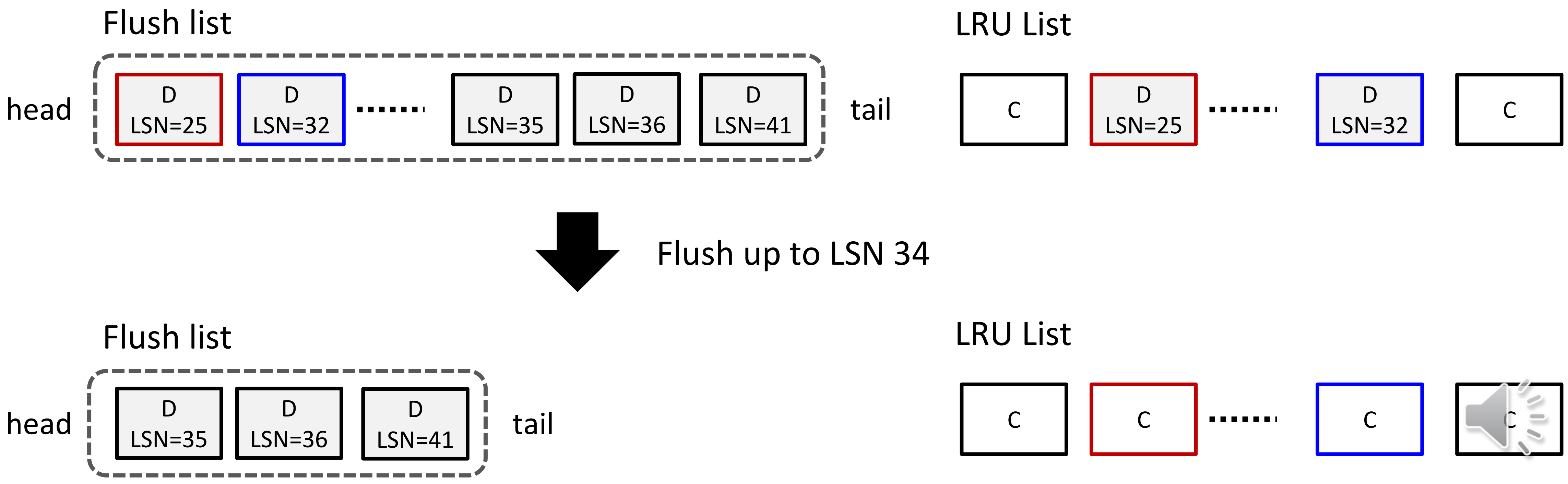
Flush List

- **List of dirty pages** that have been changed in the memory but not written to disk
  - Strictly ordered on the *oldest modification LSN*



Flush List Flush

- **Flushing to advance the oldest modified LSN**
  - Also known as fuzzy checkpoint



Types of Checkpoints

- *Sharp checkpoint (at shutdown):*
  - Flushes all modified pages for committed transactions to disk
  - Writes down the LSN of the most recently committed transaction
  - All flushed pages is consistent as of a single point in time (the checkpoint LSN) → "sharp"

- ***Fuzzy checkpoint (at normal time):***
  - Flushes pages as time passes (i.e., flush list flushing)
  - Flushed pages might not all be consistent with each other as of a single point in time → "fuzzy"

Page Cleaner Thread(s)

- `buf/buf0flu.cc: buf_flush_page_cleaner_coordinator()`

```
switch (recv_sys->flush_type) {
case BUF_FLUSH_LRU:
    /* Flush pages from end of LRU if required */
    pc_request(0, LSN_MAX);
    while (pc_flush_slot() > 0) {}
    pc_wait_finished(&n_flushed_lru, &n_flushed_list);
    break;

case BUF_FLUSH_LIST:
    /* Flush all pages */
    do {
        pc_request(ULINT_MAX, LSN_MAX);
        while (pc_flush_slot() > 0) {}
    } while (!pc_wait_finished(&n_flushed_lru,
                               &n_flushed_list));
    break;
```

LRU tail flush

Flush list flush

Flush List Flush

- buf/buf0flu.cc: buf_do_flush_list_batch()

```
static
ulint
buf_do_flush_list_batch(
    buf_pool_t*      buf_pool,
    ulint            min_n,
    lsn_t            lsn_limit)
{
    ulint       count = 0;
    ulint       scanned = 0;

    ut_ad(buf_pool_mutex_own(buf_pool));

    /* Start from the end of the list looking for a suitable
    block to be flushed. */
    buf_flush_list_mutex_enter(buf_pool);
    ulint len = UT_LIST_GET_LEN(buf_pool->flush_list);
```

Flush the dirty pages inside the flush list until lsn_limit

Acquire flush list mutex

Flush List Flush

- buf/buf0flu.cc: buf_do_flush_list_batch()

```
for (buf_page_t* bpage = UT_LIST_GET_LAST(buf_pool->flush_list);
     count < min_n && bpage != NULL && len > 0
     && bpage->oldest_modification < lsn_limit;
     bpage = buf_pool->flush_hp.get(),
     ++scanned) {

    buf_page_t* prev;

    ut_a(bpage->oldest_modification > 0);
    ut_ad(bpage->in_flush_list);

    prev = UT_LIST_GET_PREV(list, bpage);
    buf_pool->flush_hp.set(prev);
    buf_flush_list_mutex_exit(buf_pool);
```

Iterate for loop starting from the last bpage of the flush list until bpage lsn(oldest modification) is smaller than lsn_limit

oldest_modification ==0 → clean
oldest_modification >0 → dirty

Flush List Flush

- `buf/buf0flu.cc: buf_do_flush_list_batch()`

```
#ifdef UNIV_DEBUG
        bool flushed =
#endif /* UNIV_DEBUG */
        buf_flush_page_and_try_neighbors(
            bpage, BUF_FLUSH_LIST, min_n, &count);

        buf_flush_list_mutex_enter(buf_pool);

        ut_ad(flushed || buf_pool->flush_hp.is_hp(prev));

        --len;
    }
```

For all flushable pages within the flush area, flush them asynchronously

Decrement the length of a flush list

Flush List Flush: Complete I/O

- After all the work for flushing is complete, the following function is called last to complete I/O
- `buf/buf0buf.cc: buf_page_io_complete()`

```
bool
buf_page_io_complete(
/*=================*/
	buf_page_t*	bpage,	/*!< in: pointer to the block in question */
	bool		evict)	/*!< in: whether or not to evict the page
				from LRU list. */

{
	enum buf_io_fix	io_type;
	buf_pool_t*	buf_pool = buf_pool_from_bpage(bpage);
	const ibool	uncompressed = (buf_page_get_state(bpage)
					== BUF_BLOCK_FILE_PAGE);

	ut_a(buf_page_in_file(bpage));
```

```
/* We do not need protect io_fix here by mutex to read
it because this is the only function where we can change the value
from BUF_IO_READ or BUF_IO_WRITE to some other value, and our code
ensures that this is the only thread that handles the i/o for this
block. */

io_type = buf_page_get_io_fix(bpage);
ut_ad(io_type == BUF_IO_READ || io_type == BUF_IO_WRITE);
```

io_type is either read or write

Flush List Flush: Complete I/O

- `buf/buf0buf.cc: buf_page_io_complete()`

```
case BUF_IO_WRITE:
    /* Write means a flush operation: call the completion
    routine in the flush system */

    buf_flush_write_complete(bpage);

    if (uncompressed) {
        rw_lock_sx_unlock_gen(&((buf_block_t*) bpage)->lock,
                              BUF_IO_WRITE);
    }

    buf_pool->stat.n_pages_written++;
```

io_type is write

Do not free the flushed page!
Keep it in LRU list as a clean page.

```
    /* We decide whether or not to evict the page from the
    LRU list based on the flush_type.
    * BUF_FLUSH_LIST: don't evict
    * BUF_FLUSH_LRU: always evict
    * BUF_FLUSH_SINGLE_PAGE: eviction preference is passed
    by the caller explicitly. */
    if (buf_page_get_flush_type(bpage) == BUF_FLUSH_LRU) {
        evict = true;
    }

    if (evict) {
        mutex_exit(buf_page_get_mutex(bpage));
        buf_LRU_free_page(bpage, true);
    } else {
        mutex_exit(buf_page_get_mutex(bpage));
    }

    break;
```

If LRU `list` `flush`, then free the page and return it to the free list

InnoDB Configuration Options (`my.cnf`)

- You can readjust innodb parameters according to your environment
  - E.g., `innodb_io_capacity`, `innodb_buffer_pool_size`…

- Optimizing InnoDB by just modifying these parameters can lead to significant performance improvement

- Refer to this link for details about innodb parameters: https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html

PA 1

- **PA1: Improve TpmC by changing IO related InnoDB parameters**
- For the assignment, your task is to change/add the I/O related InnoDB parameters in my.cnf to improve TpmC
- By looking at MySQL source code and MySQL document, investigate which parameter affects I/O operation process in InnoDB and how you can improve the TpmC by readjusting it/them.
- You can add/modify multiple InnoDB options.

PA 1

- After achieving performance gain, present an experiment result before and after changing `my.cnf`

- Then, elaborate the reason why it leads to performance improvement based on MySQL source code and document.

- The PA will be graded based on the following criteria: **TpmC improvement** (40%), **Result Analysis** (60%).

- You will get zero marks on both criteria if you forge your results.

- Refer to week5 for the PA 1 experiment guide
https://github.com/LeeBohyun/SWE3033-S2023

References

[1] MySQL document: https://dev.mysql.com
[2] Mijin An, MySQL Buffer Management, https://www.slideshare.net/meeeejin/mysql-buffer-management
[3] An et.al., “Avoiding Read Stalls on Flash Storage”, SIGMOD2022, https://dl.acm.org/doi/pdf/10.1145/3514221.3526126