1) Slotted Page Implementation:

- Each Slotted Page is equal to a disk block. The size of the disk block is initialized on startup (pass in as a command line argument).
- A disk block is simulated as a file, named `block_{block_number}.img` . A table is a directory `storage/storage_{block_size}_SEQ/{table_name}`, whose contents are these disk block files (the storage is split by block size so we can run experiments on different block sizes without them interfering with each other)
- The records inside the slotted pages are not ordered. According to the actual implementation of slotted pages, they aren't supposed to be ordered. A linear search is required to search for a record (equality search). However in slotted pages, we also have the pointers pointing to each record and thus this marginally speeds up linear searching. Each record can only show up in one slotted page block.
- The size of mem is the size of our cache, since our memory (the database memory buffer) is a cache to storage. Cache will only be able to fit `mem_size/block_size` number of disk blocks. The disk blocks will be evicted based on a timestamp (LRU scheme).
    - When you are attempting to search for a record on equality search, you will first look through the cache to see if what you are looking for is in there.
        - In the case of ID retrieval, you will search the cache first. If it is in the cache, return the record. If it is not in the cache, search through all the blocks on disk. If no record is found, it will return an empty list.
        - In the case of an area-code retrieval, you will search both the cache *and* the storage.
- For write operations, we designed it to first look through the cache and storage for matching IDs (if the ID matches, then it is an update operation)
    - If a record is found, then we update the record.
    - If a record is not found, then we will insert the record into an available slotted page.
        - We will attempt to find an available slotted page in cache first. The first open slot in the slotted page is used.
        - If all slotted pages in cache are full (cannot contain any more records) then we will look through storage to find an available slotted page to bring into cache. If no slotted pages on disk are available for a record insertion, then a new slotted page will be created and kept in cache.
    - To distinguish between update and write, we must first walk all blocks in storage in addition to the blocks in cache. This means that even if cache has space to simply write the record, we must walk all blocks in storage to make sure the record with the same ID doesn't already exist. In order to walk the storage, we have to bring in every block to cache, so if the table is bigger than memory, then we'll have to evict (ie. swap out) many pages to do the walk.
- We designed our cache to use a write-back policy. What this means is that all slotted pages in cache are the most up-to-date. The slotted page on disk will only be updated upon a page eviction (also on program completion).

    - What this means is if there is a slotted page in cache that is also on disk, the slotted page in cache is the up-to-date version and the slotted page on disk might be outdated.
        - The slotted page on disk would only be updated if the cached version is be evicted.
        - Upon eviction, the slotted page on disk would be overwritten (if the file exists)
- For record deletion, we first search through the cache to see if there is a record that matches the target ID. If the record exists in cache, we will delete the record in that slotted page. If the record is not found in any of the slotted pages in cache, we will search through the disk storage to see if the record exists there. If it is, we will bring that block into memory and delete the record in that slotted page.
- For table deletion, we just delete the entire directory.
- We have automated unit tests for single slotted page for all operations (see “Running unit tests” in README.md). We also have test scripts in test_scripts/ to run the whole Data Manager (see “Running the program” in README.md).

2) LSM Implementation:

• Description about the record placement, file organization, and memory management strategies.

File storage has the following scheme: Each table gets a directory in the storage directory. Each table directory has three directories, one for each of L0, L1, L2. These directories hold a subdirectory for each SSTable in that level. The counts of SSTables per level and the ranges of each SSTable are kept as metadata in dictionaries in LSMStorage. Each SSTable directory holds files that each are a max of one block size and hold sorted records that make up that SSTable. The max number of these files are limited by the amounts of blocks backing the SStables, as specified by input. The files are laid out one record after another. The file name format is num-in-sorted-order_num-records-in-block.

The memory divides the meta data into tables, meaning that meta data allocates a memtable to each schema name. The memory always looks for the memtable that belongs to a schema name and writes the record to that memtable. Memory may flush the memtable if necessary. The memtables of memory contains the most recent write changes of the database, meaning all adds, updates and deletes of records. Thus, the memtables will be the first thing that memory inspects upon the fetching of record(s). The read cache consists of both memtable and LRU will be explained in the read cache section.

• The design and components of the row key (i.e. the key for each record).

Each key-value node in the memtable consists of (pk, rec), where the pk is the primary key of the record rec. But once the memtable is flushed to L0 only actual records are written to disk blocks.

The key is ensured to be unique within an LSM level, but older versions may have the same key in lower levels of storage. These older versions will be removed in compaction to ensure the uniqueness of the key (see For operation R).

• The design of your read cache, and the methods to keep it consistent.

The majority of the read cache consists of the LRU dictionary and a small part of it will be the memtables. As explained in the memory management, the most recent write records are always in the memtable and thus the memory will first inspect memtables for a requested record. Then the memory will inspect LRU dictionary.

Each schema name is mapped to one LRU structure, which consists of three lists L0, L1 and L2. If a read operation requests records from storage, all the records fetched from storage will be placed in their corresponding LRU structure and in corresponding level list. If a read operation can find its record in LRU, then that record will be moved to the end of the list. Therefore, the first record in LRU list is the least recently used record. When we read from LRU we always start searching at the beginning of the list.
The multi-level design in LRU is crucial in consistency control. Since all add, update and delete records are inserted in memtables which will be flushed to the storage levels. Once those records are brought back to the LRU, they would contain records with duplicate record keys. Among these are the old and new write records with the same key which can only be merged during LSM storage compaction. Thus, we must separate them in LRU and put them in separate lists, in which case their order of use time will be arranged only within their levels. When we inspect the LRU structure, we search in the same sequence as we do in stoarge: L0, L1 and L2.

If the record is not found in the cache, memory will request the record from the storage. Upon read operations by area code, the memory will always request records form the storage.

• For operation W: Describe all processes that could be triggered by an operation W. Detailed description of the compaction strategies (e.g., how many and which SSTable(s) to pick for compaction in each level?).

Writes are always done to the Memtable object. If there is not a Memtable for that table, one is created. If the Memtable is full, it will be pushed to storage and a new one is created. This pushing of a full table possibly triggers compaction of L0 if there is not room for another SSTable in that level. This compaction possibly triggers compaction of L1 as SSTables are being pushed to L1 and there may not be room. These compactions are also done routinely in the background via some background threads. We decided to compact all of them so as to amortize the cost of this operation and reduce the time between compactions.

• For operation E: Describe all processes that could be triggered by an operation E, which are different than those by an operation W.

Erase does a write, but instead uses the negative version of the primary key and -1 as data to indicate that the record is a delete record (see For operation R for why). It is then treated as a normal record by storage and will filter down into the lower levels, eventually overwriting

• For operation D: Describe all processes that could be triggered by an operation D.

When a table is deleted, any Memtable is found and discarded, then the compaction threads for that table are killed. The file storage for that table is removed and so is all metadata being kept in the program.

• For operation R: How to guarantee the correctness of the read under different conditions (e.g., multiple updates, deleted records, deleted tables).

Read will always check higher levels first, starting with the Memtable in memory if there is one. As such, it will always find the newest version. Multiple updates will overwrite the same record if in the memtable. When the memtable is pushed, the older duplicates are invalidated. There is a special indicator for a record: it’s primary key is turned to the negative of that primary key and the data set to -1. This allows binary search to still work while keeping the same record format and minimizing the length of records. These records are still considered valid records and as such will be found if they are the newest, but they are treated as nothing. If the table has been deleted, this is checked first and the record will not be searched for. Ranges for SSTables are kept in metadata so that only the SSTables that could possibly contain the records are checked.

• For operation M: Have you implemented any methods other than the full table scan to do it?

We have not. We discussed some sort of clustered index to make it easier to find blocks with those records, but decided it would not be worth it since the maintenance of this would incur significant overhead and as a worst case it would still be linear in the number of blocks.

• Any other implemented optimization methods, e.g., additional index files, auxiliary memory objects, parallel compaction, etc.

The storage component was implemented with multiple threads to allow for parallel compaction. We determined that the overhead from the locking schemes needed was minimal in comparison to the benefit that we got from doing this.

• Test cases indicating the correctness of the four operations under different conditions.

Background methods for the storage were tested with lsm_test.py. The memory side was tested with a test option for mem_lsm.py. Additionally, we made various scripts for the database to run and empirically verified their correctness and used the errors to track down bugs.

3) Comparison between the two file organization strategies:

Read Only (Calculated from Read+Write Time - Write Time)		
	Sizes (bytes)	Time (seconds)
Slotted Page	Mem size - 4,096,000 block size - 4,096	5.270
	Mem size - 34 block size - 34	459.382
LSM	Mem size - 4194304 block size - 4096 blocks per sstable - 16	327.258
	Mem size - 34 block size - 34 blocks per sstable - 1	17.714

Writes Only		
	Sizes (bytes)	Time (seconds)
Slotted Page	Mem size - 4,096,000 block size - 4,096	3.670
	Mem size - 34 block size - 34	984.098
LSM	Mem size - 4194304 block size - 4096 blocks per sstable - 16	0.285
	Mem size - 34 block size - 34 blocks per sstable - 1	535.237

Read + Writes		
	Sizes (bytes)	Time (seconds)
Slotted Page	Mem size - 4,096,000 block size - 4,096	8.940
	Mem size - 34 block size - 34	1443.480
LSM	Mem size - 4194304 block size - 4096 blocks per sstable - 16	327.544
	Mem size - 34 block size - 34 blocks per sstable - 1	552.950

4) Comparison with and without optimizations:
The optimizations were discussed before implementation. Due to this, there is no version without optimizations to compare with.

Contribution of each team member:

- We practiced pair-programming. Since there are two file organization strategies, we splitted into pairs of two.
    - Core (main control, parser, logger, etc..), API and program architecture design - everyone
    - Slotted Pages - Bin (mem) and Paul (storage)
    - LSM - Nate (storage) and Shibo (mem)