Group Programming Project – Database System Technology CSC443/CSC2525H

Summary

In this project, you will build a key-value store from scratch. A key-value store (KV-store) is a kind of database system that stores key-value pairs and allows retrieval of a value based on its key. KV-stores exhibit the following simple API:

Open("database name")	opens your database and prepares it to run
Put(key, value)	stores a key associated with a value
Value = Get(key)	retrieves a value associated with a given key
KV-pairs= Scan(Key1, Key2)	retrieves all KV-pairs in a key range in key order (key1 < key2)
Close()	closes your database

This API also reflects common data structures you have likely seen before (e.g., binary trees). Unlike simple binary trees, however, your KV-store must be able to store data greater than the amount of memory your system has available. In other words, it will have to spill data to storage (disk or SSD) as the size of the data grows.

You will implement this key-value store using various data structures that we will cover in class. Specifically, you will implement an LSM-tree in storage complemented by in-memory Bloom filters. This architecture follows that of many popular modern KV-stores, including RocksDB, Cassandra, HBase, etc.

The Use of KV-Stores

KV-stores are widely used in industry. Note that they have a far simpler API than traditional relational database systems, which expose SQL as an API to the user. There are many applications for which a simple KV API is sufficient. Note, however, that KV-stores can also be used as the backbone for relational database management systems. For example, MyRocks by Meta is an example of a relational database utilizing as its backbone a key-value store very similar to the one we will build, namely RocksDB.

Timeline

This project will be done in small steps for digestibility. There will be three steps overall, and we will release each step every two or so weeks as we cover more of the course content. The final deadline for the project is December 1st. At this point, we will ask you to submit your entire code base for revision. There will be no grading of smaller steps before the final hand-in.

Office Hours

Therefore, it is your group's responsibility to stay on track. Make sure to use the office hours if you feel you are falling behind. We will allocate as many office hours as we see there is demand for to make sure you are well-supported.

Language

You will be working with C++ or Rust (your pick). These languages are widely used in industry for system-level programming, so if you have not already encountered them, now is a good time to learn one of them. The first step of the project is an opportunity for you to brush up on your programming skills as it will only require algorithmic knowledge that you should already have from previous courses. We will then gradually build up.

Groups

This project will be done in groups of 3. It is your responsibility to form the groups. We will create a space in Piazza to help you find teammates. If you cannot find a group by the end of the second week of the course, email the course instructor.

How to Work Together

In any group scenario, different teammates arrive with different levels of experience. This is ok, and it will continue to happen throughout your careers. For example, at the start of your career, as you join development teams, you will be junior and your challenge will be to fit in and become productive as soon as possible. Later in your career, you will likely need to train new people coming into your group. We will inevitably be simulating this kind of environment in the course. If you come with more experience than your teammates, help them get up-to-speed and avoid taking most of the work on your shoulders. If you have less experience, it is your responsibility to proactively take initiative and tasks, ask for help, and ultimately become a productive and valuable team member. We also encourage you to program in groups. Pair programming is a wonderful way to learn from each other and also identify mistakes in each other's code. We also encourage you to review each other's code. Code reviews are a common part of any company's development workflow.

Keep in mind that any career is as much a people's game as it is about skill. Your classmates now will become your professional network in the future. Therefore, be kind to each other, and keep in mind that in group scenarios, one's success is everyone's success. Making friends is as important as your final grade.

Version Control System

To collaborate on this assignment, please use the version control system Git. This is a widely used version control system, and you will likely encounter it during your careers, so it is important to become conversant in it. Git would allow you and your teammates to work collaboratively by merging changes into the same code base without stepping on each other's toes (e.g., undoing each other's edits). You can create private git repositories on github.com or bitbucket.com. Please make sure your repositories are private so that others can't copy your code. Please also keep your repositories private once the code ends so that future generations of students are unable to copy your code.

Coding Practices

Be sure to write clear modular code and comment on the purpose of every class and method. This is important for you and your teammates to understand each other's code. It is also important for us to ultimately be able to understand your code so that we can grade it.

Scope

Typically, KV-stores can handle variable-sized keys and values and can operate concurrently with multiple threads. To simplify the project and only focus on core learning outcomes, however, it is ok for your implementation to be single-threaded and handle only primitive data types (e.g., keys and values can both be 8-byte integers).

On the use of external libraries

It is ok to use standard external C++ libraries, but you are not allowed to use them as a replacement for the material you will be asked to program yourself.

Integrated Development Environment

I recommend working with an integrated programming environment like Eclipse. This will provide you with visual debugging tools, which will make the programming process easier.

Makefiles

Please use a makefile to compile your code. This will simplify your life considerably. Makefiles simplify the compilation of your project by declaring the compilation workflow and executing it all with one command. This will require some fiddling effort to figure out, but it will pay off. This is also important for us when grading your project. Ideally, we should just be able to type "make" in the command line to compile your project.

Platform

This project should be done in a Linux environment for the best support. If you do not have Linux installed, you can use a virtual machine.

Report

Your final hand-in will include a report that describes your implementation. The goal of this is to draw the attention of the reader to how you implemented the requirements for the project, as well as to any “cool” or non-trivial features or optimizations that you added beyond the scope of the project. The foremost goal of this is to practice technical writing, something you will also do during your career. Another goal is to make it easier for us to grade your project by telling us where to direct our attention. You are welcome to get feedback on the state of your report during office hours.

Experiments

Your implementation should not only meet the required functionality but also perform well. To verify this, we will ask you to run experiments that benchmark the capabilities of your system. Please include those in your report. Each experiment should lead to one figure that shows how some performance metric changes as you vary some parameter of the system or the workload. Each figure should also include one paragraph to describe the experiment, summarize the take-away message, and provide intuition. Make sure your graphs are always clearly labeled. It should be crystal clear what the x-axis and y-axis represent and what the unit of measurement is. For each step in this project, we will provide guidelines about the kind of experiments you should run.

Reproducibility

Reproducibility in science is emerging as an important concern across many domains. In the database community, there are recent initiatives to ensure that results reported in academic papers can actually be reproduced by others in the community. You are welcome to browse here for more information: https://reproducibility.sigmod.org/. The goal is to check ourselves and ensure robust progress for our field as a whole. Reproducibility is also crucial in industry: as database vendors add new functionality into their products, we must understand their performance side-effects to prevent them from manifesting unexpectedly for the client while running in production.

To help you get used to this mindset, we will ask you to employ some basic reproducibility methodology for your experiments. For each experiment that you design, please generate one executable file from the Makefile that runs the experiment and generates a CSV output to the command line. We should then be able to copy this CSV into excel and easily regenerate your experimental figure.

Unit Tests

Unit tests are an important software engineering method to prevent regression as you build your system. Regression means accidentally creating new bugs in older functionality as you build new functionality. At a minimum, you should create some unit tests that make sure you can insert some data into your system and get sensible results when you query it. At least three tests should be included in this project that test your put, get and scan functionality. You should run these frequently as you develop to identify bugs as they emerge in real time.

While there are many frameworks out there to facilitate unit testing in C++ or Rust, we will not ask you to use them to keep things simple. We will only ask you to create one executable that we can run to test this basic functionality. Your Makefile should create this executable, and it should be called tests. It should generate some output to the command line indicating which tests are running, which succeeded, and which failed. By the time you hand in the project, all tests should succeed.

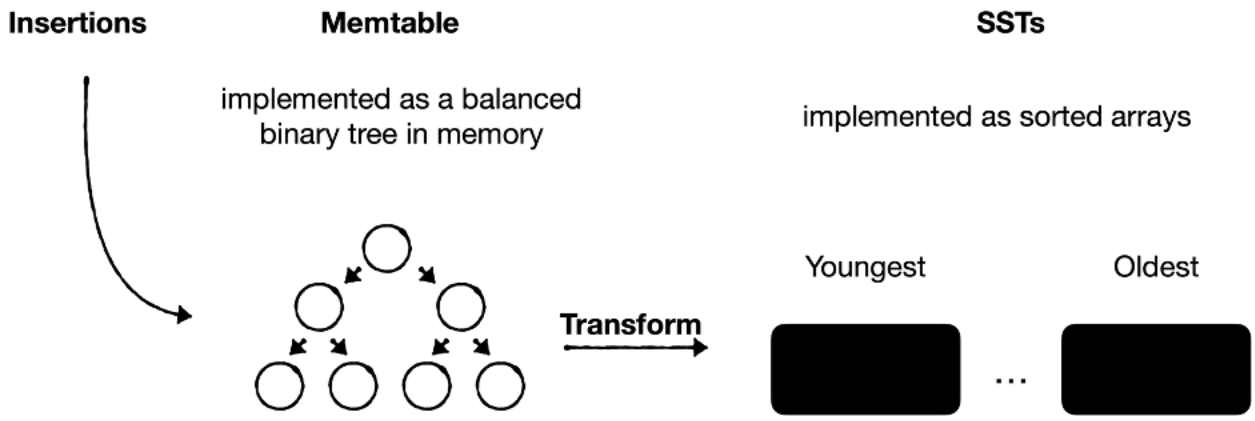
Step 1 – Creating a Memtable and SSTs

In the first step of the project, you will create a balanced binary tree that complies with the API above. You may choose any binary tree implementation you know and like (e.g., red-black tree or AVL tree). This can be implemented as a C++ or Rust class. You can also create a class to represent nodes in your tree. We will refer to this tree henceforth as the memtable (i.e., memory table).

Your implementation should take a parameter called memtable_size. This controls the maximum size of your memtable. As data is added to your memtable, it will eventually reach this threshold. At this point, you should implement a traversal of the memtable based on the order of keys, and output all the key-value pairs to a sorted file, which you will save in your storage device. You can employ the same code as your scan operator to perform this traversal. You will need to investigate and employ C++/Rust libraries to perform an efficient sequential write to a file.

Once the file is created and stored in storage, you will need to allocate a new memtable to be able to ingest more data. When this newer memtable fills up, you will again transform it into a sorted file using the same workflow as above. We will refer to these files from now on as SSTs (Sorted String Tables). This is how real systems like RocksDB refer to them, so we may as well use terminology from the real world. The SSTs should be assigned unique names.

After many insertions of data entries, your overall system should look as follows.



Your get method should search both the memtable and all SSTs in the order of youngest to oldest. It should return the first value with a matching key that it finds. To do this, you will need to implement a binary search over each SST. Please investigate the pread linux commands to only read one page of an SST for each hop of your binary search.

Your scan method should also search the memtable and the SSTs. To search the SSTs, you can use a binary search to find the start of the range and continue with a scan until you encounter a key outside of the range. Note that the scan command is more complex to implement than the put or get commands because it is not known in advance how many key-value pairs are in the target range. Therefore, this command should return a pointer to an array of results, along with the size of this array.

The Open command in the API should create a directory with the name of the database you are creating and store all SSTs and other data relating to your database within this directory. If the open command is called with the name of an existing directory, it should prepare that database for operation. The close command should transform whatever is in the current Memtable into an SST and gracefully shut down the database without losing any data in memory. When you subsequently open the same database, all data that had previously been inserted should still be available.

For now, let's focus on only inserting unique keys into the tree (i.e., insertions). We will later learn how to also handle updates when we learn about LSM-trees.

This step of the project can be further decomposed into mini-steps, which you can assign to each of the three members of your group.

1. Implement the memtable as a balanced binary tree that supports put and get.
2. Implement scans over the memtable, and use this mechanism to transform the memtable into an SST when it reaches capacity
3. Extend your get and scan functionalities to search the memtable and all SSTs with binary search.

For this step of the project, please generate experiments that measure the performance of your three operators, put, get and scan, as you insert more data into the system. The x-axis should report the data volume that had been inserted, while the y-axes should report throughput. Three figures should be produced, one for each operation. These should be included in your report under the title “Experiments for Step 1”. Bandwidth can be measured by dividing time into consecutive windows (e.g., 10 seconds each), recording how many operations were performed in each window, and diving by the window length in seconds to get the average number of operations per second.

Step 2 - Extendible Buffer Pool & Static B-trees

In the second step of the project, we will add a buffer pool to our KV-store, and we will make our SSTs more query-efficient using B-tree techniques.

Buffer Pool. Your system should implement a buffer pool to cache frequently accessed pages in memory and thereby save storage I/Os. Before issuing any read to storage, your system will first check if the given page is in the buffer pool. If so, it will skip issuing the I/O. The buffer pool should be implemented as a hash table mapping from a unique ID for each page to the 4KB page itself. The buffer pool should maintain O(1) expected performance.

Identifying Pages. Any page in your system can be identified uniquely by concatenating the name of the file that contains it with the offset of the page within that file. You can hash this ID to a random bucket in your buffer pool. You will need to store this ID alongside the actual 4KB page to make sure you return the correct page in case of hash collisions.

Collision Resolution. Being a hash table, the buffer pool will need to deal with the case that multiple pages map to the same bucket. You must deal with this by implementing some kind of collision resolution strategy. Chaining or Robin Hood hashing are good choices.

Eviction Policy. We would like you to implement either the LRU or the Clock eviction policy. Both of them are good choices, and it is up to you to choose.

Hash function. You should investigate some hash functions and employ a reasonable one (you may want to investigate murmur hash and/or xxhash).

Query Path. You should integrate access to the buffer pool as a part of your get (point query) workflow. A query (get or scan) command should first attempt to find the target

DB page in memory before retrieving it from storage. Once a page from storage is retrieved, it should be placed in the buffer pool.

Bonus 1: Handling Sequential Flooding. We encourage you to implement some logic in the buffer pool to safeguard against sequential flooding**.** If a range query is short, it likely makes sense to keep the constituent pages in the buffer pool. If the range is long, though, we may want to evict the constituent pages immediately or with higher priority.

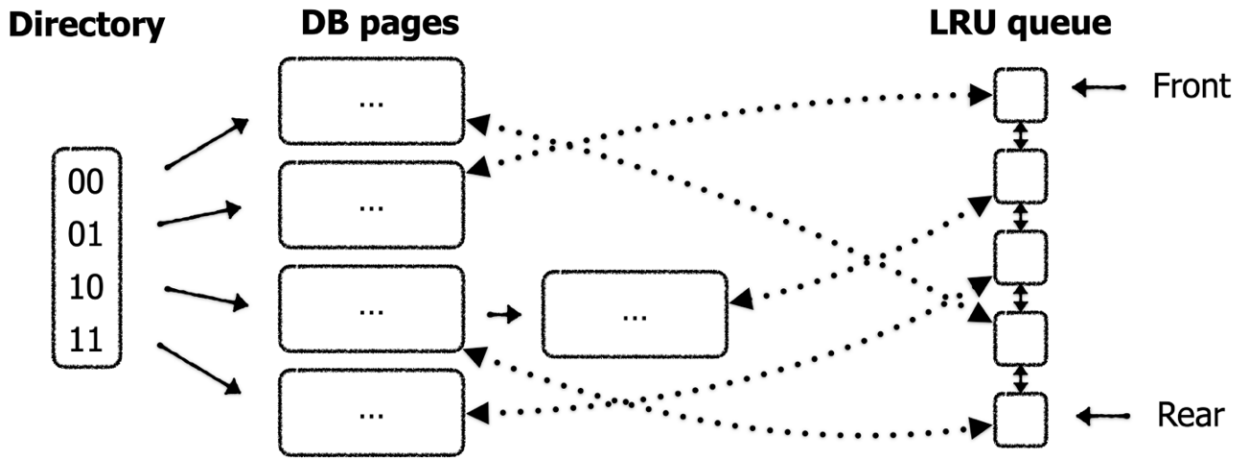
Bonus 2: an Expandable Buffer Pool. In many database applications, it is common to allocate the hash table for the buffer pool as a fixed number of contiguous buckets in memory. The problem with such a design is that it's not very flexible for tuning: shrinking or enlarging the buffer pool requires rewriting the entire hash table from scratch. To achieve a more tunable design, we encourage you to implement the buffer pool using extendible hashing: there should be a directory of hash prefixes where each prefix maps to a linked list of frames, each of which caches one 4KB database page.

Note that in the lecture slides, we have seen Extendible Hashing used as a possible solution for indexing data in storage. However, extendible hashing is a general technique that can be applied whenever we have a hash table that we would like to dynamically grow or shrink. We can therefore also apply it to the context of a buffer pool.

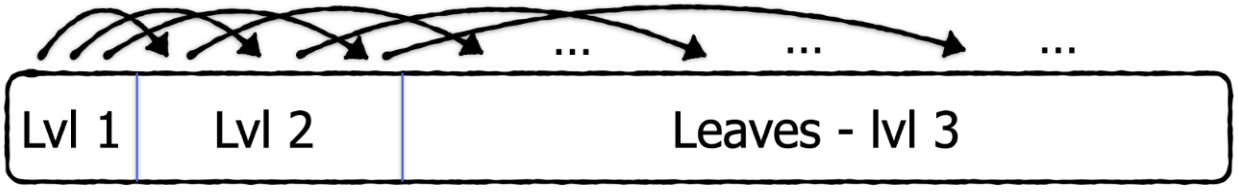
Your design should take as parameters the initial and maximal sizes of the directory. The directory should be initialized to the minimum size and grow gradually as more data is inserted on a need basis. Once it reaches the maximum size, the eviction policy should kick in.

You should expand the API of your kv-store to allow users to change the maximum size of the directory. If a user chooses to shrink it to save memory, the eviction policy should first kick out as many pages as needed. For completeness, you can also shrink the directory size if the buffer pool size shrinks by so much that the directory size becomes a memory bottleneck.

We would like you to implement either the LRU or the Clock eviction policy. Both of them are good choices, and it is up to you to choose. With LRU, for instance, the design may be visualized as follows:



Structuring an SST as a B-Tree. In addition to implementing a buffer pool, we will also optimize our SST structure to make it more query efficient. Your current SST structure is just a sorted list of key-value pairs, which you can binary search. As we saw in class, such a design performs worst-case $O(\log_2(N/B))$ I/O per query. In contrast, a B-tree can perform a query in worst-case $O(\log_B(N))$ I/O. We would like to achieve these superior performance properties for querying an SST by transforming the SST structure into a static B-tree. It is static in the sense that it should not handle insertions, updates, or deletes. This makes it significantly easier to implement. Your existing sorted data can be thought of as contiguously allocated 4kb leaf nodes with no padding (i.e., there is no spare space in each leaf node to accommodate insertions). You will need to add $O(\log_B(N))$ layers of internal nodes. The root of the B-tree should be at a known place near the beginning of the SST so that queries know where to begin the search. This may be visualized broadly as follow. Each page should be 4KB aligned within the SST file.



SST static B-tree structure

Note that an additional benefit of using a B-tree structure is that each query creates less pressure on the buffer pool: at most O(log_B(N)) page evictions are needed rather than O(log_2(N/B)).

Your system should support both doing a binary search over the leaves or doing a B-tree search from root to leaf. It should take a parameter to decide which of these algorithms to use to process queries.

Experiment. Design an experiment comparing your binary search to B-tree search in terms of query throughput (on the y-axis) as you increase the data size (on the x-axis). This experiment should be done with uniformly randomly distributed point queries and data. The buffer pool should be enabled in this experiment, and the data should grow beyond the maximum buffer pool size so that evictions kick in. Explain your findings.

This step of the project can be broken down into multiple mini-steps, which you can use to divide the work among your team members.

1. One member can implement the core hash table and collusion resolution strategy
2. One member can implement the eviction policy
3. One member can implement the b-tree structure for SSTs

All experiments you conduct from Step 2 and onwards should employ direct I/O for reading and writing to storage while bypassing the operating system's cache.

Remember to document all of your design choices in the report.

In the next step of the project, we will implement a compaction policy to transform your KV-store into an LSM-tree. We will also implement Bloom filters to improve query performance.

Step 3 – An LSM-Tree with Bloom Filters

In this step of the project, we will transform your implementation into a real LSM-tree. We will do so by implementing a compaction mechanism, adding a Bloom filter for each SST, and loading Bloom filters into the buffer pool when accessed. There are five bonus tasks in this step. You do not need to do any of them to get full marks on the project. They are there to provide an additional challenge to those who want it.

Compaction Policy. You should implement a basic LSM-tree with a fixed size ratio of 2 between any two levels. Whenever there are two SSTs at the same level, merge them.

Compaction Implementation. Files may be large, so it is important not to bring them fully into memory to compact them. A merge operation should take place by allocating three buffers: two input buffers for the two files being merged, and one output buffer for the new file being created. In each iteration, the minimum key in the input buffers should be appended to the output buffer. Whenever the output buffer is full, we append its contents to the new file being created and clear it.

Bonus 1: Dostoevsky. We encourage you to generalize the Basic LSM-tree implementation to the Dostoevsky compaction policy from the research lecture. This means you will have one sorted run at the largest level and use tiering at all the smaller levels.

Bonus 2: Min-Heap. If you implement Dostoevsky, you will be compacting data from across multiple runs in a given level at the same time. To do this more efficiently, we encourage you to use a min-heap. It is ok to use a library for the min-heap implementation.

Updates. In this phase of the project, we will also finally deal with updates. Whenever a compaction operation encounters two entries with the same key across two SSTs that are being merged, only the more recent version should be kept as the older one is superseded. You should also ensure that your get operation returns the most recent version of each key to the user and terminates after it finds it. Similarly, your scan operations should return the most recent version of each entry in the specified key range to the user.

Deletes. We will also add support for deletes. You should add a delete(key) operation to the API. This operation adds a tombstone to the memtable. You can represent a tombstone by reserving a specific value for it (e.g., the minimum 8-byte integer). If a get or scan operation encounters a tombstone as the first occurrence of a given key, they should return a negative to the user. When tombstones reach the largest level of the LSM-tree, they should be disposed of, as at this point there are no longer any older versions of the entry in exitance for them to mask.

Integration with the Buffer Pool. Once a file is disposed of after compaction, some pages belonging to this file may still be in the buffer pool. As these pages will no longer be in use, they will eventually be evicted by the buffer pool's eviction policy. However, you may also consider evicting these pages immediately to clear space in the buffer pool and meanwhile prevent other non-obsolete pages from getting evicted. This is not an obligatory part of this project, but how to accomplish this efficiently is an interesting question that I encourage you to think about.

Bloom Filters. You will need to implement a Bloom filter for each SST. This Bloom filter should be created for an SST during a merge operation while the file is being created. The Bloom filter should be integrated into the get workflow. If it returns a negative, we can skip accessing the file. However, if a positive is returned, we must search the file for the target key. A Bloom filter query should abort on the first zero that is encountered. The number of bits per entry assigned to the filters should be provided as a parameter. Please employ the same number of bits per entry for Bloom filters across all levels of the LSM-tree.

Bonus 3: Blocked Bloom Filters or Cuckoo filters. For an extra challenge, you may implement a blocked Bloom filter (as we saw in the midterm) or a Cuckoo filter (which we will study in the course soon) instead of a regular Bloom filter.

Bonus 4: Monkey. For another challenge, please implement Monkey to optimally assign false positive rates across the different Bloom filters.

Persisting the Filter. The filter should be persisted in storage as a part of its corresponding file. Some metadata should be added to the file with information about the offset at which the bloom filter begins and how large it is.

Putting Bloom Filters in the Buffer Pool. Once accessed by a query, the filter should be brought into the buffer pool. You may bring the whole Bloom filter into the buffer pool as one contiguous chunk of memory, or you may bring separate pages of the filter into the buffer pool as they are requested. You should ensure that the buffer pool as a whole is still able to keep track of the volume of contents it is storing and to ensure that eviction is triggered when we are at capacity.

Bonus 5: Bulk-Loading. We encourage you to implement a bulk-loading process using external sorting. This process takes a file that contains unsorted data. It sorts it using an external sorting procedure and places the resulting SST in the appropriate level given its size. You may use library functions for this (e.g., a sorting function and a min-heap). You will need to extend your API to support this function.

Experiment. Measure insertion, get, and scan throughput for your implementation over time as the data size grows. Describe your experimental setup and make sure all relevant variables are controlled. Please fix the buffer pool size to 10 MB, the Bloom filters to use 5 bits per entry, and the memtable to 1 MB. Run this experiment as you insert 1 GB of data. Measure get and scan throughput at regular intervals as you insert

this data. If you did any of the bonus tasks, please make sure to report how they are used in the experiment.

Grading Rubric and Instructions for Project Final Report

The following is a guideline for how we plan on grading your group project. The more clear and concise your report is, the easier it will be for us to give you points.

Submitting your project: Add instructors into your repository. Report file should also be added to the repository. Use the following email id's to send us an invite. Please do not email the TAs about anything else. Any communication about the project should be directly communicated to the course instructor.

Akshay Arun Bapat - bapataks@gmail.com
Pooyan Habibi - pooyan.habibi1989@gmail.com
Shikhar Jaiswal - jaiswalshikhar87@gmail.com
Ioannis Xarchakos - xarchakos@cs.toronto.edu

Marking scheme - Total 110 points + bonus (18 points)

1. Report quality 10 points
2. Core Implementation - 52 points
3. Software Engineering practices - 18 points
4. Experiments - 30 points
5. Bonus features - 18 points

List of report must-haves

1. Description of design elements - concisely describe the different elements of your design (i.e., memtable, B-tree creation, buffer pool, etc), and any interesting design decision you have made. This should correspond to the list of design elements below. Feel free to tell us about extra cool stuff you might have added. Please also tell us where to find the implementation of each design element in the code (i.e., file and function name). (2 points)
2. Project status - Give details on what works and what does not. If there are known bugs in your code, list them here. (2 points)
3. Experiments - please show experimental figures and explain your findings for each experiment. If you have some extra experiments that you did, put them here under a separate "Extra Experiments" heading. (2 points)
4. Testing - Testing is a very important part of any implementation, Mention how you tested your implementation, if you have several unit and integration tests, list them here. (2 points)
5. Compilation & running Instructions - detail out how to run your project (i.e., give makefile targets and describe what they are for). If you have an executable that we can use to run simple commands, give instructions on how to use that. (2 points)

List of coding practices to check

1. Code readability - consistent naming and indentation, meaningful identifiers, comments, no too long lines (3 points)
2. Code modularity - short and specific task functions, good code reuse (3 points)
3. Version control - use of version control to collaborate and making meaningful commits (2 points)
4. Makefile - makefile for experiments and executable build (doesn't have to be separate as long as instructions are in the report) - (4 points)
5. Tests - correctness and performance tests (6 points)

List of Design Elements

Step 1

1. KV-store get API - 1 points
2. KV-store put API - 1 point
3. KV-store scan API - 2 point
4. In-memory memtable as balanced binary tree - 4 points
5. SSTs in storage with binary search - 5 points
6. Database open and close API - 2 points

Step 2

7. Implementation of Buffer pool as hash table with collision resolution - 4 points
8. Integration buffer pool with queries - 4 points
9. Clock or LRU eviction policy - 4 points
10. Static B-tree for SSTs - 7 points

Step 3

11. Bloom filter for SST and integration with get - 5 points
12. Persisting Bloom filters in SSTs - 3 points
13. Compaction/Merge of two SSTs - 6 points
14. Support update - 2 points
15. Support delete - 2 points

Bonus Features

16. Handling sequential flooding (2) - 3 points
17. Extendible hash buffer pool (2) - 5 points
18. Dostoevsky (3) - 3 points
19. Min-heap (3) - 2 points
20. Blocked Bloom filters or Cuckoo filters (3) - 3 points
21. Monkey (3) - 2 points

List of Experiments

1. Data volume VS put performance (1) - 4 points
2. Data volume VS get performance (1) - 4 points
3. Data Volume VS scan performance (1) - 4 points
4. Binary search vs B-tree index with query throughput with changing data size (2) - 6 points
5. Put throughput with increasing data size (3) - 4 points
6. Get throughput with increasing data size (3) - 4 points
7. Scan throughput with increasing data size (3) - 4 points

A few additional notes:

1. Even if you get bonus points, the project is still capped at 110 points. Getting all 110 points would amount to 100% for your grade for the project.
2. You may claim bonus points for other cool implementation or design decisions you made that go beyond the project description and not already listed as bonus suggestions. Make sure to describe these to us in the report.
3. Extra features do not necessarily get you bonus unless they provide new insights (new experiments) or improve the design (new or changed implementation)

Appendix - Command Cheat Sheet

This appendix contains some useful system calls and C functions that might be helpful for your project implementation. Review them before you start.

Disclaimer - This is not an exhaustive list but just a set of generally required standard function calls. The information provided here is also limited to standard use and you should rely on proper documentation to verify corner cases and error codes.

Useful System Commands

1. open(const char *pathname, int flags) - Opens the file specified by <pathname>. Returns a file descriptor used by other commands to refer to this file. Following are some useful flags.
    a. O_RDONLY - open file with read only access
    b. O_WRONLY - open file with write only access
    c. O_RDWR - open file with both read and write access
    d. O_APPEND - opens file in append mode (affects pwrite)
    e. O_DIRECT - opens file with direct IO access
    f. O_SYNC - write operations finish with file integrity sync
2. close(int fd) - Removes the association of file descriptor <fd> with its associated file. Returns 0 on success and -1 on failure.
3. pread(int fd, void *buf, size_t count, off_t offset) - Reads <count> bytes from the file referenced by file descriptor <fd> at an offset of <offset> into the memory referenced by <buf>. Returns the number of size read or -1 if failed.

4. pwrite(int fd, const void *buf, size_t count, off_t offset) - Writes <count> bytes from the memory referenced by <buf> to the file referenced by file descriptor <fd> at an offset of <offset>. Returns the number of bytes written or -1 if failed.

Useful C commands

1. malloc(size_t size) - allocates memory of size <size>, returns a pointer to the starting address of the allocated memory
2. posix_memalign(void **memptr, size_t alignment, size_t size) - allocates memory of size <size> such that *memptr points to the starting address of the allocated memory. It differs from malloc in a way that the address of the allocated memory is a multiple of <alignment>. Returns 0 on success.
3. free(void *ptr) - deallocates any allocated memory referenced by <ptr>

Direct IO

Typically when any read call is issued, the data is fetched from the disk into OS cache (along with some readahead) and then the required information is copied from OS cache into the memory referenced by the read system call. This caching enables the OS to potentially skip costly data copy from disk to memory when any required information is found in the OS cache. Another advantage is that it avoids a few CPU cycles for each read/write.

We want to design our own buffer manager that will perform a very similar operation as OS cache but with some differences. If we keep using OS cache, we will essentially be double buffering. This means that all read operations will first copy data from disk to OS cache, then from OS cache to our buffers. We do not need to have this data present twice in the memory.

When applications want to perform their own buffering/caching, the OS kernel provides a way to skip OS caching. This is called “Direct IO”. When enabled, all read/write operations skip the OS caching and are directly read from or written into the disk from memory.

Linux allows enabling Direct IO per file and is done when opening the file for any operations. We use the O_DIRECT flag when the associated file will be used for Direct IO. One should avoid using Direct IO and normal IO on the same file as the result could be undefined especially when the file is also being written.

Memory Alignment

In some linux kernel versions Direct IO is restricted to aligned memory and only in batches of some size. Using posix_memalign instead of malloc helps to get the alignment correct. Malloc is also aligned but only to 1 byte. posix_memalign can be used to get alignment with larger sizes.

Aligned memory is also helpful/required in some cases where the hardware imposes some restriction on certain commands and can boost performance if used.