(8) [10%] Continue with the above question. Please rewrite the above C code into the ***blocked*** version with the calculated M.

2. [50%] Consider a *key-value* (*KV*) storage. A KV pair stored in the storage consists of a constant length of an alphabetical string as a key, and its associated value is simply a variable-length bit string. The KV storage provides `PUT(x,v)` and `v=GET(x)` APIs to applications, where `PUT(x,v)` stores a key-value pair in the storage with the designated key `x` and value `v`, and `GET(x)` returns the value `v`, given the queried key `x`, previously stored in the KV store. Additionally, `SCAN(x1,x2)` offered by the KV storage returns a list of KV pairs such that the keys of the retrieved data items are no less than `x1` and no greater than `x2`. Let the KV storage be operated in a computer system equipped with sizes of x volatile memory space and of y persistent storage, where $x \ll y$.

Notably, applications may introduce two types of workload patterns to the KV storage. The *random access* pattern denotes keys generated by `GET(.)`, `PUT(.)` and `SCAN(.)` are random, while the *sequential access* represents the scenario that data items are addressed with increasing (or decreasing) keys. Secondly, `GET(.)`, `PUT(.)` and `SCAN(.)` operations can be performed concurrently on the fly by the KV store. Thirdly, the total storage size of KV pairs stored in the system may be greater than x, the available volatile memory space as aforementioned. Finally, the KV storage may fail anytime and support the recovery for committed `PUT(.)` operations.

Please discuss designs and implementations of operating systems to provide such a KV storage service, considering to optimize the performance metrics in terms of delays and throughputs in manipulating `GET(.)`, `PUT(.)` and `SCAN(.)` operations. You shall state clearly subject to the following techniques in order.

(1) [5%] In-memory indexing
(2) [5%] In-disk indexing
(3) [5%] Caching
(4) [5%] Paging
(5) [5%] Batched I/O
(6) [5%] Multithreading and thread scheduling
(7) [5%] Shared memory and consistency
(8) [5%] Encoding/Decoding
(9) [5%] File system block layout
(10) [5%] File system compaction

名詞解釋：

1. In-memory indexing:

In-memory indexing is usually maintained as a B-tree in-memory.

---

2. In-disk indexing:

The database table does not reorder itself every time the query conditions change in order to optimize the query performance: that would be unrealistic. In actuality, the index causes the database to create a data structure. The data structure type is very likely a B-Tree. While the advantage of the B-Tree are numerous, the main advantage for our purpose is that it is sortable. When the data structure is sorted in order it makes our search more efficient.

---

3. Caching:

An in-memory cache is a data storage layer that sits between applications and databases to deliver responses with high speeds by sorting data from earlier requests or copied directly from databases. An in-memory cache removes the performance delays when an application built on a disk-based database must retrieve data from a disk before processing. For the Get() and Scan() operation, If data exists in the cache (when a "cache hit" happens), the app retrieves information from the cache. In order to balance from consistency and delay, Put() will write data to the buffer first, and write to disk later.

Paging:

One possible option is to use a disk-based database architecture that optimizes for the case data is on secondary storage. Traditional database management system pack data in fixed-size pages, use logical pointers (e.g. page IDs and record offsets) that allows the buffer pool abstraction to move data between memory and storage transparently, and have specialized page replacement algorithms to maximize hit rates and reduce I/O latency. Another straightforward way of extending available memory is to keep main-memory databases unchanged and simply use the default OS paging.

Batched I/O:

Batch program are used when you have a large number of database updates to do or a report to print. We can sort the batch I/O job with ascending or descending order to get better throughput on all of the operations.

Multithreading and threadscheduling:

Throughput and response time of database server can be improved by running task scheduler with task queues and worker threads. Such computation task on row data and data transfer can parallel execute. The important thing is that task scheduler need to offers synchronization primitives, e.g., barriers, to avoid synchronization issues.

Shared memory and consistency:

We can use strict consistency model to perform Shared memory consistency. Global clock is present in which every write e.g. put(), should be reflected in all processor caches by the end of that clock period. The next operation must happen only in the next clock period.

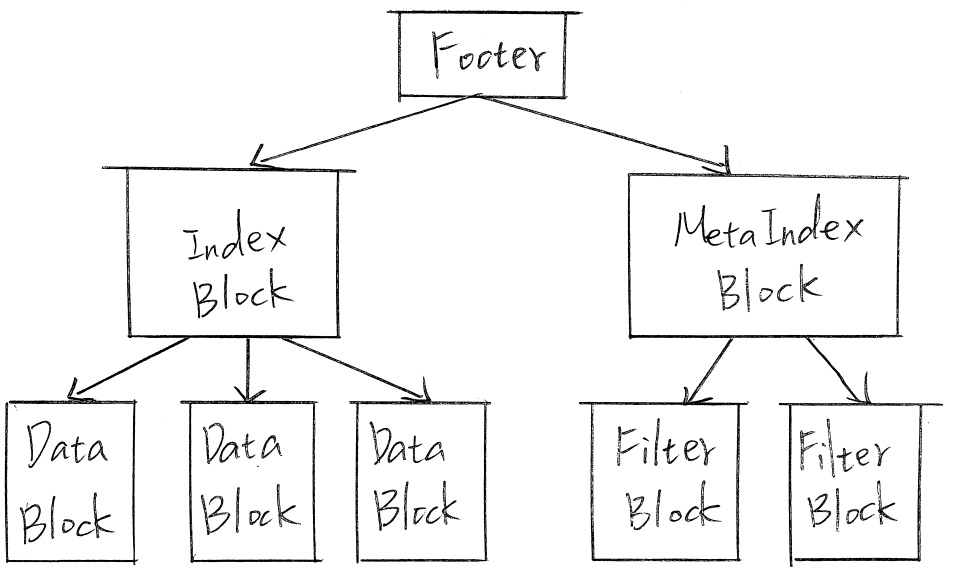
Encoding / Decoding:

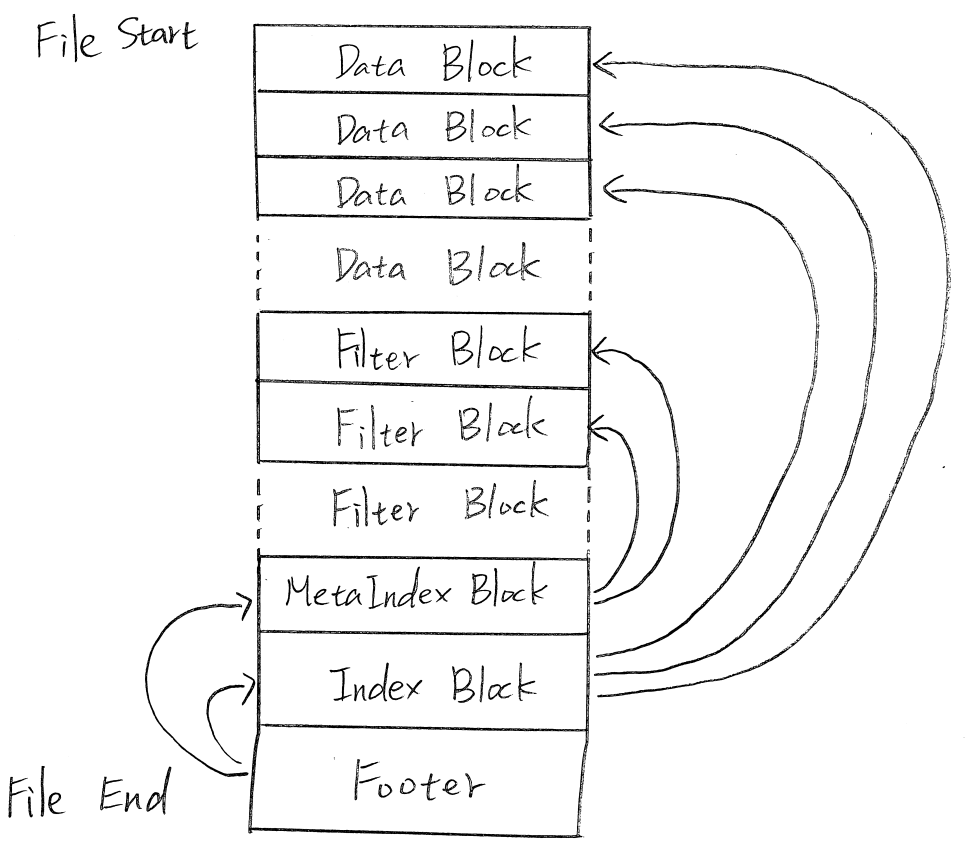
Encoding the data in database will conserve serve space and keep data secure, Here we can encode the value by UTF-8 which convert Unicode characters into binary numerals. Scan and Get will read the encode data from database then decode it, since the encode data size is less than the original ones, data transfer time can be reduced.

File system block layout:

The File system block layout is as follow, sorted string table is a sorted key-value structure. Since key is sorted, seek time of Scan() can be reduced, thus increase disk throughput.

SSTable





File system compaction:

Compaction is the process of taking all the current revisions in the file and writing a new file with just the active documents, effectively deleting all old revisions. Compaction can have dramatic (巨大) results in recovered disk space and performance (as seek times are reduced on smaller files).