KvStor

Design

Key-value storage is a way to use key to quickly access datas. Datas on the storage system are generally located in two areas, memory(cache) and storage. Memory is used for quick access while storage for backup or persistency. To achieve high performance, general ways to implement key-value storage are to use B-Tree, Rb-Tree etc.. However, B-Tree and Rb-Tree are complex and are not easy to implement. Hence, I use skip list, a simple but efficient data structure to implement key-value storage.

To insure safety and persistence, the datas must be stored on persistent storage like hard disk driver, ssd etc.. Skip list in memory is easy to build and access. The data can be quickly accessed by searching the key in the skip list. In memory, the address of the data is easy to obtain. But how can we get the address of the data in file? To implement a key-value storage system, we must find the map from key to address(offset) in file.

The KvStor storage system is described as blow.

- The skip list is in memory as cache. Each node of the skip list contains the key, the value and the address of the data in file. The cache size may be limited.
- All the datas are stored in the data file. They are located by the offsets of datas in file.
- The offsets (address) of datas are stored in the index file. The data format of the index file is:
  key address size
  key is the key. *address* represents the offset of the data in data file. *size* is the size of the data. I call (*key address size)* index.
  Every line is the same. We can use it to access every data item in data file.
- The datas are appended in the data file instead of updated in-place. Thus the old datas are abandoned. This is for fast writes.
- Thanks to the appended writing, consistency is subtlety ensured. Because the address of the data is modified after the data is written to file. If a crash happens during the writing of data or after the writing of data but before the modifying of the address, the old data is not be ruined. And the address is still rightly pointing to the data. Similarly, if a crash happens during the writing of index, the old index is also remained. Thus the old data will not be lost.
- Because the cache is limited, the system maintains a daemon to flush data in cache to storage when the remaining cache volume is small.
- There are four basic operations for client side:
  set(key, value) get(key) del(key) exit()

Implementation

Data structure

```
/* node in skip list*/
typedef struct node
{
	keyType key;
	int64 addr;	/*address of value in data file*/
	bool inmem;	/*value in memory or not*/
	valueType value;
	int64 size;
	bool dirty;
	struct node **next;
} Node;
/*skip list*/
typedef struct skip_list
{
	unsigned level; /*the highest node's hight */
	Node *head;	/*point to first node*/
} skip_list;
```

Balance among performance, consistency and complexity

A simple idea of implement key-value storage system is to implement a interface that let the file storage space can be allocated and reclaimed like memory. It maintains the space in a range of blocks. They are divided into two types, free blocks and old blocks. Each type contains a range of lists. Each list contains a range of blocks with the same size. When allocating space, it choosees the block with proper size. When reclaiming space, the old block is inserted into the corresponding list.

However this method is inefficient. The management of the storage space takes plenty of time. The KvStor storage system, although wastes much space, it gains a high access speed. The appended writing can greatly enhance the sequence of IOs, thus can greatly reduce the access time.

The KvStor storage system contains a mechanism that reclaims the cache memory. I suppose that the KvStor has only a limited amount of memory. A daemon is created to periodically check the usage of memory. If the remaining memory has reached to a threshold, it begins to reclaim space, flushing some nodes to file. Considering the complexity and the limited time, this function has not been implemented.

Files

Skip_list.hpp defines the skip list class, it contains functions that operate the skip list like create, insert, delete etc.
Skip_list.cpp implements the skip list class
KvStor.hpp define the kvstor class, it contains the operations that offered to client side, like set, get, del etc.
KvStor.cpp implements the KvStor class
Server.cpp implements the server side, uses socket, can handle concurrent reads and writes.
client.cpp implements the client side

Results

Platform: Vmware + CentOs 7

Handle multiple clients

Three clients send requests to server

```
[rao@volcano srcj$ ./server

-------Waiting for clients's request-------

client 2 is connecting.
IP: 127.0.0.1  Port: 47096
client 2 connected success
get a
set b B
get name
set e E
get e
del e
get e
lock end
client 3 is connecting.
IP: 127.0.0.1  Port: 47098
client 3 connected success
get a
get school
lock end
set school cqu
get school
client 4 is connecting.
IP: 127.0.0.1  Port: 47100
client 4 connected success
set f FF
get f
del f
get f
lock end
```

```
[rao@volcano srcj$ ./client

-------KvStor Usage:-------

command         function                example
set             set key's value         eg.`set name Tom`
get             get key's value         eg.`get name`
del             delete the key          eg.`del name`
exit            exit KvStor             eg.`exit`

KvStor> get a
b
KvStor> set b
bad command..
KvStor> set b B
OK!
KvStor> get name
rao
KvStor> set e E
OK!
KvStor> get e
E
KvStor> del e
OK!
KvStor> get e

KvStor>
KvStor>
KvStor> dlsd
command not found..
KvStor> get a b
bad command..
KvStor>
```

Problem

In the process of implement the KvStor storage system, I got in trouble when sync the data and index into files. When set a key, the data is surely in memory, but no matter write how many times, even when I use fsync to force syncing the data to file, it can not write into file. Sometimes a few datas appear in file, however, most of time, the datas seems be lost.

After thinking and debugging a very long time, I find the problem is caused by the fstream object. I didn’t close fstream object. Now the reason is clear. I think, it might be that the datas are still remained in memory until the fstream object is closed.

Source code see: https://github.com/yonggangrao/KvStor