Implementation of LSM-Tree Key Value Store

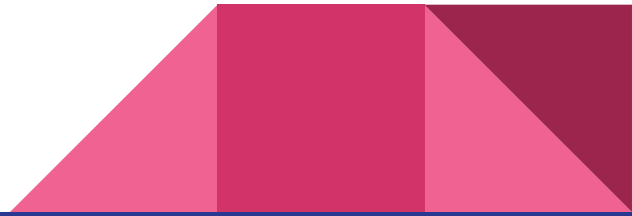
Greeshma Yaluru

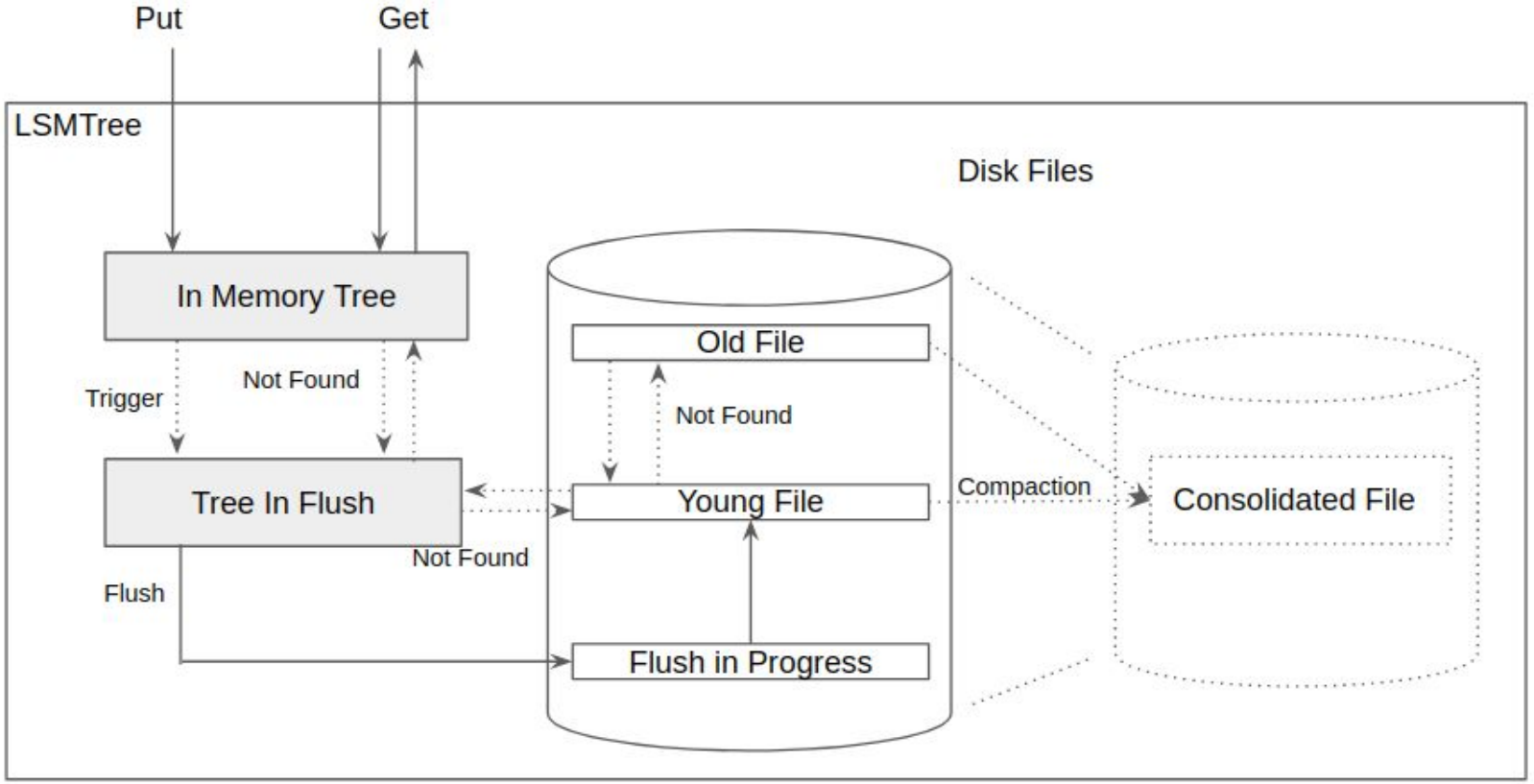
Zhenhuan Wu

What is an LSM Tree?

What are some main components in the Tree

Name a few partitioning strategies?




Put
Get
LSMTree
Disk Files
In Memory Tree
Trigger
Not Found
Tree In Flush
Not Found
Flush
Old File
Not Found
Young File
Flush in Progress
Compaction
Consolidated File

Buffer

1. Insert the key-value pair ("a", 1) into the write buffer.
2. Insert the key-value pair ("b", 2) into the write buffer.
3. Insert the key-value pair ("c", 3) into the write buffer.
4. Insert the key-value pair ("d", 4) into the write buffer.

Flush into disk

Level 0 (L0)
a	b	c	d

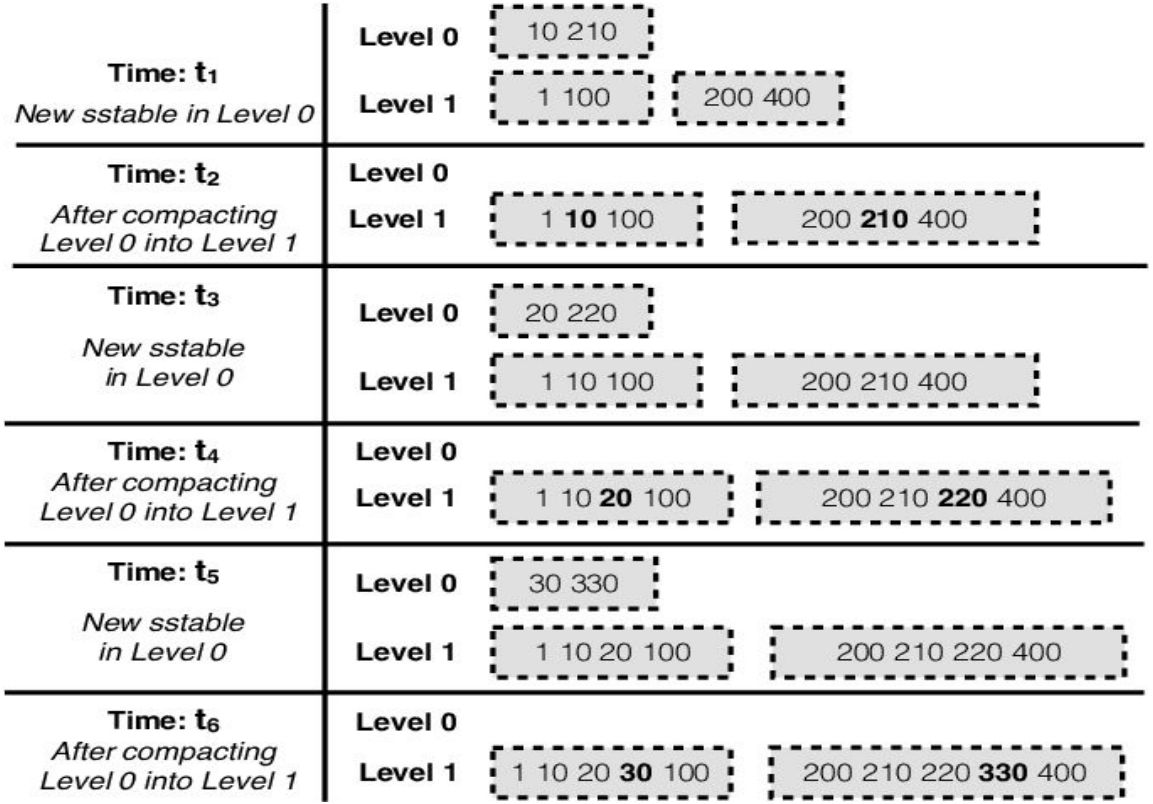
Insert “e”

Level 0 (L0)	Level 1 (L1)
a	b	c	d
e			

Insert “f”

Level 0 (L0)	Level 1 (L1)	Level 2 (L2)
a	b	c	d		
e					
f					

Leveling Strategy



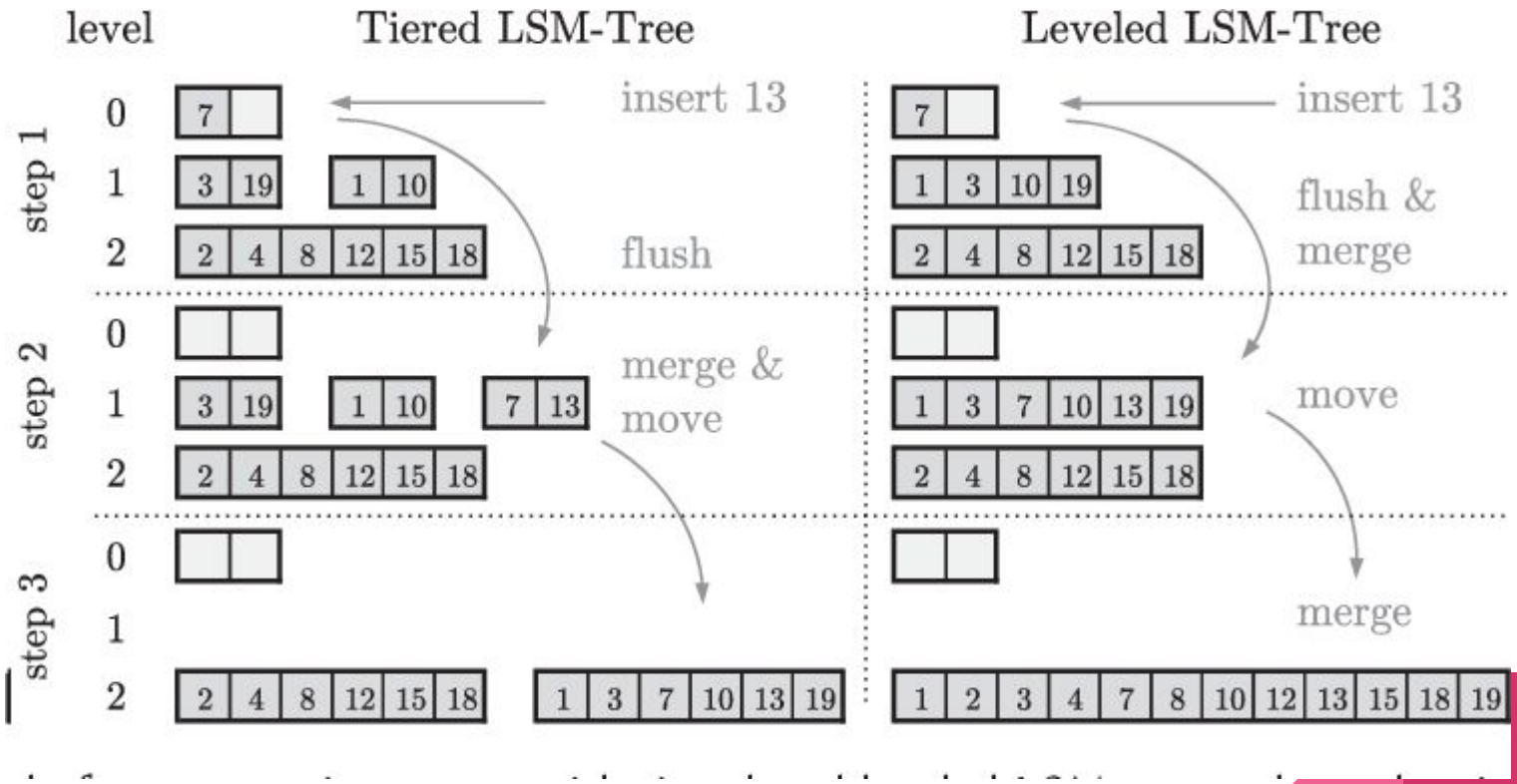
Implementation

```
#include "Buffer.h"

Value Buffer::get(int key) {
    if ((key < this->min) || (key > this->max)) return Value(false);
    for (auto &pair: this->pairs) {
        if ((pair.first == key) && (pair.second.visible)) {
            return pair.second;
        }
    }
    return Value(false);
}

void Buffer::put(int key, Value val) {
    if (this->size() == 0) {
        this->max = key;
        this->min = key;
    } else if (this->max < key) { this->max = key; }
```

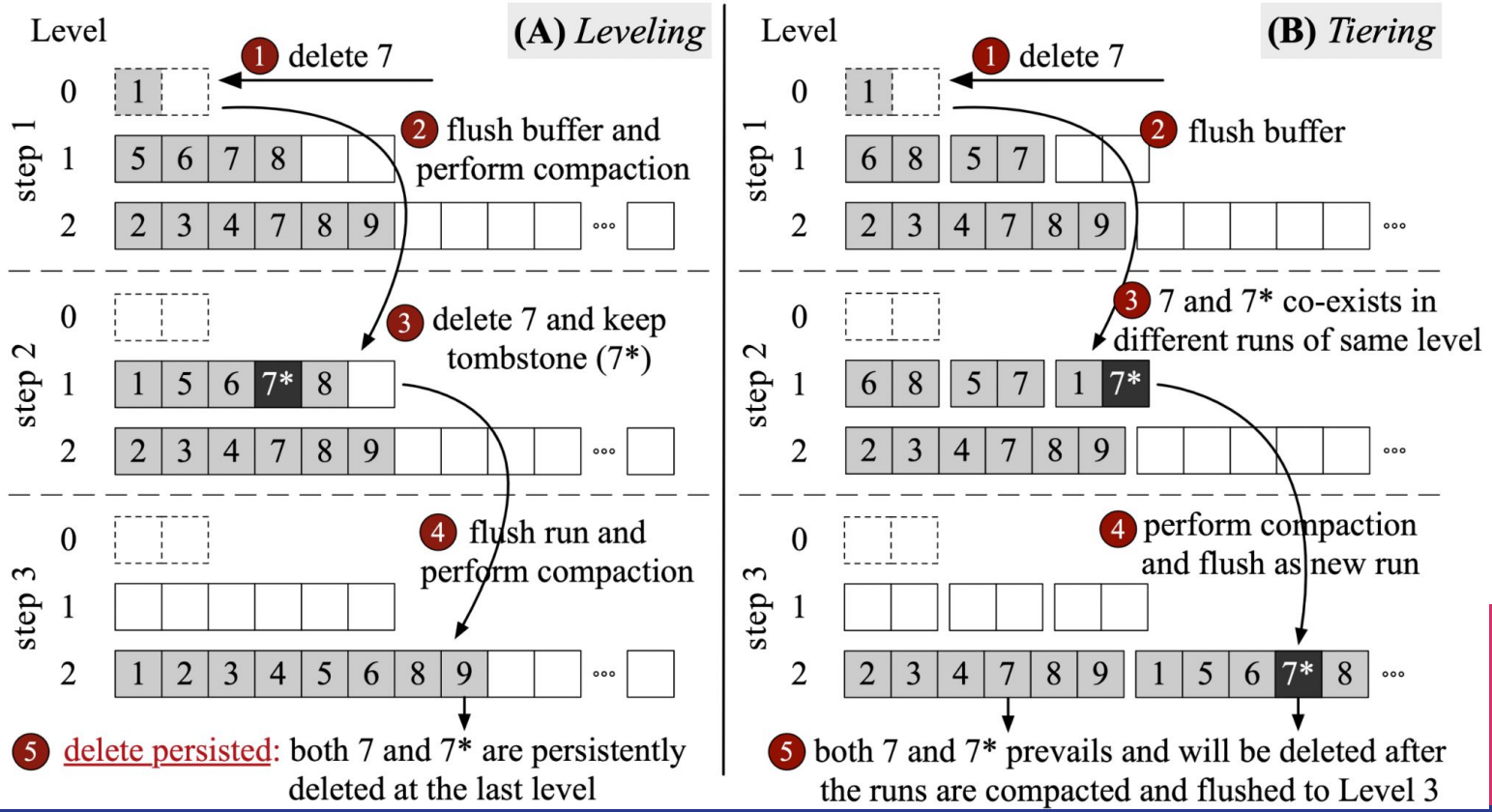
61_templatedb > src > templatedb > diskrun.hpp > DiskRun<K, V> > run_No

```cpp
#include <string>

using namespace std;

template<typename K, typename V>
class DiskRun : public Run<K, V> {
    typedef Pair<K, V> KV_pair;
    int capacity;
    int entries_in_page;
    int level;
    int run_No;
    int entries_num;
    int page_num;
    int page_size;
    bool doExist;
    K* fence_pointer;
    K MIN;
    K MAX;
    string dir;

public:
    DiskRun(int capacity, int pagesize, int level, int run_No) {
        MIN = 0;
        MAX = 0;
        dir = "./data/LSM_L" + to_string(level) + "_R" + to_string(run_No) + ".run";
        this->capacity = capacity;
        this->page_size = pagesize;
        this->level = level;
        this->run_No = run_No;
        doExist = false;
```

```cpp
#include <vector>
#include <cmath>
#include "buffer.hpp"
#include "diskrun.hpp"
#include "bloomfilter.hpp"

class LSM {
private:
    std::vector<Buffer*> buffers;
    std::vector<DiskRun*> runs;
    int maxBufferSize;
    int runSize;
    int level;
    int numLevels;
    double mergeThreshold;
    BloomFilter bloomFilter;

public:
    LSM(int bufferSize, int runSize, int numLevels, double mergeThreshold, int bloomFilterSize) :
        maxBufferSize(bufferSize),
        runSize(runSize),
        level(0),
        numLevels(numLevels),
        mergeThreshold(mergeThreshold),
        bloomFilter(bloomFilterSize)
    {
        // Initialize the buffers and runs
        for (int i = 0; i < numLevels; i++) {
            buffers.push_back(new Buffer(maxBufferSize));
```


level
Tiered LSM-Tree
Leveled LSM-Tree
step 1
step 2
step 3
0
1
2
insert 13
flush
merge &
move
flush &
merge
move
merge
7
3 19
1 10
2 4 8 12 15 18
7 13
1 3 10 19
1 3 7 10 13 19
1 2 3 4 7 8 10 12 13 15 18 19

Tiering Strategy



Tiering Strategy

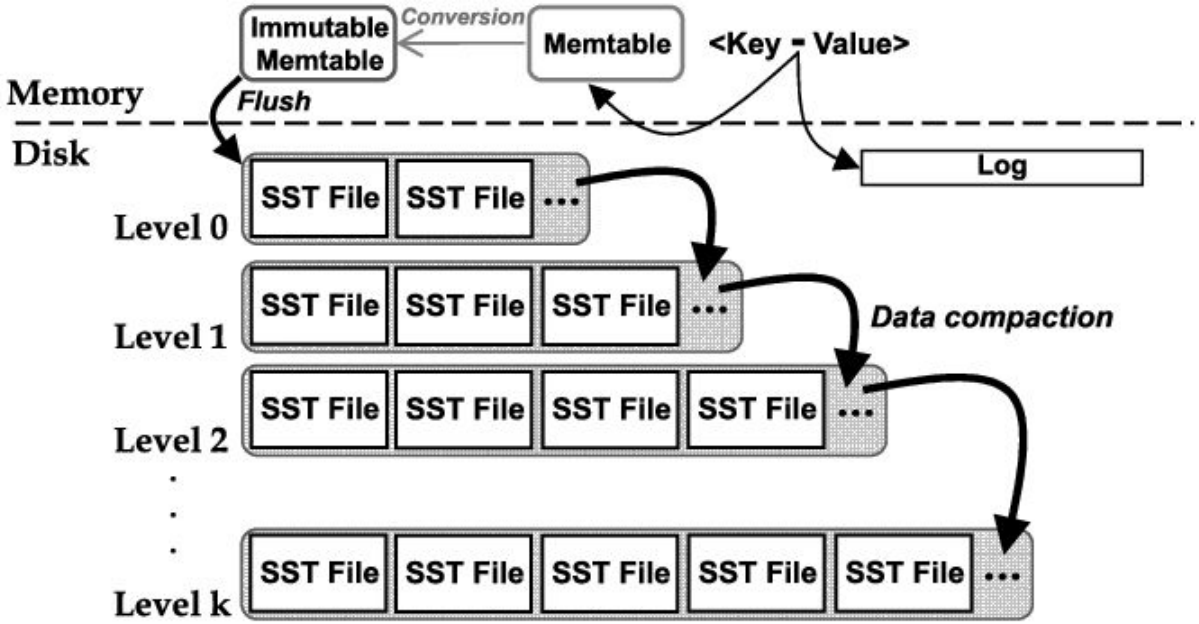
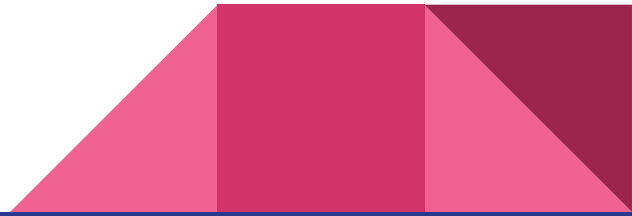


FIGURE 1. An overview of LSM-tree in LevelDB.

Solution Approach

- Understanding the framework
- Buffer & Memory
- Disk & Leveling
- Tiering (in-progress)



Solution Approach

- Understanding the framework

```
class Value
{
public:
    std::vector<int> items;
    bool visible = true
    int range = 0;
    Value() {}
    Value(bool _visible) {visible = _visible;}
    Value(bool _visible, int _range) {visible = _visible  range = _range;}
    Value(std::vector<int> _items) { items = _items;}

    bool operator ==(Value const & other) const
    {
        return (visible == other.visible) && (items == other.items);
    }
```

Solution Approach

- Understanding the framework

```
class Value
{
public:
    std::vector<int> items;
    bool visible = true;
    int range = 0;
    Value() {}
    Value(bool _visible) {visible = _visible;}
    Value(bool _visible, int _range) {visible = _visible; range = _range;}
    Value(std::vector<int> _items) { items = _items;}

    bool operator ==(Value const & other) const
    {
        return (visible == other.visible) && (items == other.items);
    }
```

Preliminary Results

Thank You

Questions?