NoSQL/LevelDB Intro

NoSQL=No SQL or Not Only SQL

- database cannot support big data！ What? Are you kidding me?
- for a cluster with 1000 nodes, the CAP cannot be satisfied together?
- CAP=Consistency Availability Partitioning
- How to guarantee that data are 100% available and 100% accessible even hardware fails?
- Conventional database is not designed for this
- NoSQL is a compromise. It sacrifices Consistency for Availability by applying the MVC rule(multi-version concurrency) and eventual consistency
- only support key-based lookup

Popular NoSQL System

Type	Notable examples of this type
Key-Value Cache	Apache Ignite, Coherence, eXtreme Scale, Hazelcast, Infinispan, Memcached, Velocity
Key-Value Store	ArangoDB, Aerospike
Key-Value Store (Eventually-Consistent)	Oracle NoSQL Database, Dynamo, Riak, Voldemort
Key-Value Store (Ordered)	FoundationDB, InfinityDB, LMDB, MemcacheDB
Data-Structures Server	Redis
Tuple Store	Apache River, GigaSpaces
Object Database	Objectivity/DB, Perst, ZopeDB
Document Store	ArangoDB, BaseX, Clusterpoint, Couchbase, CouchDB, DocumentDB, IBM Domino, MarkLogic, MongoDB, Qizx, RethinkDB
Wide Column Store	Amazon DynamoDB, Bigtable, Cassandra, Druid, HBase, Hypertable
Native Multi-model Database	ArangoDB, Cosmos DB, OrientDB

But, We come to the origin: LevelDB

1 What is LevelDB? An open sourced implementation of BigTable by Jeff Dean

2

- open sourced implementation of bigtable: https://github.com/google/leveldb
-

3 LevelDb is designed to process billions of key-value pairs

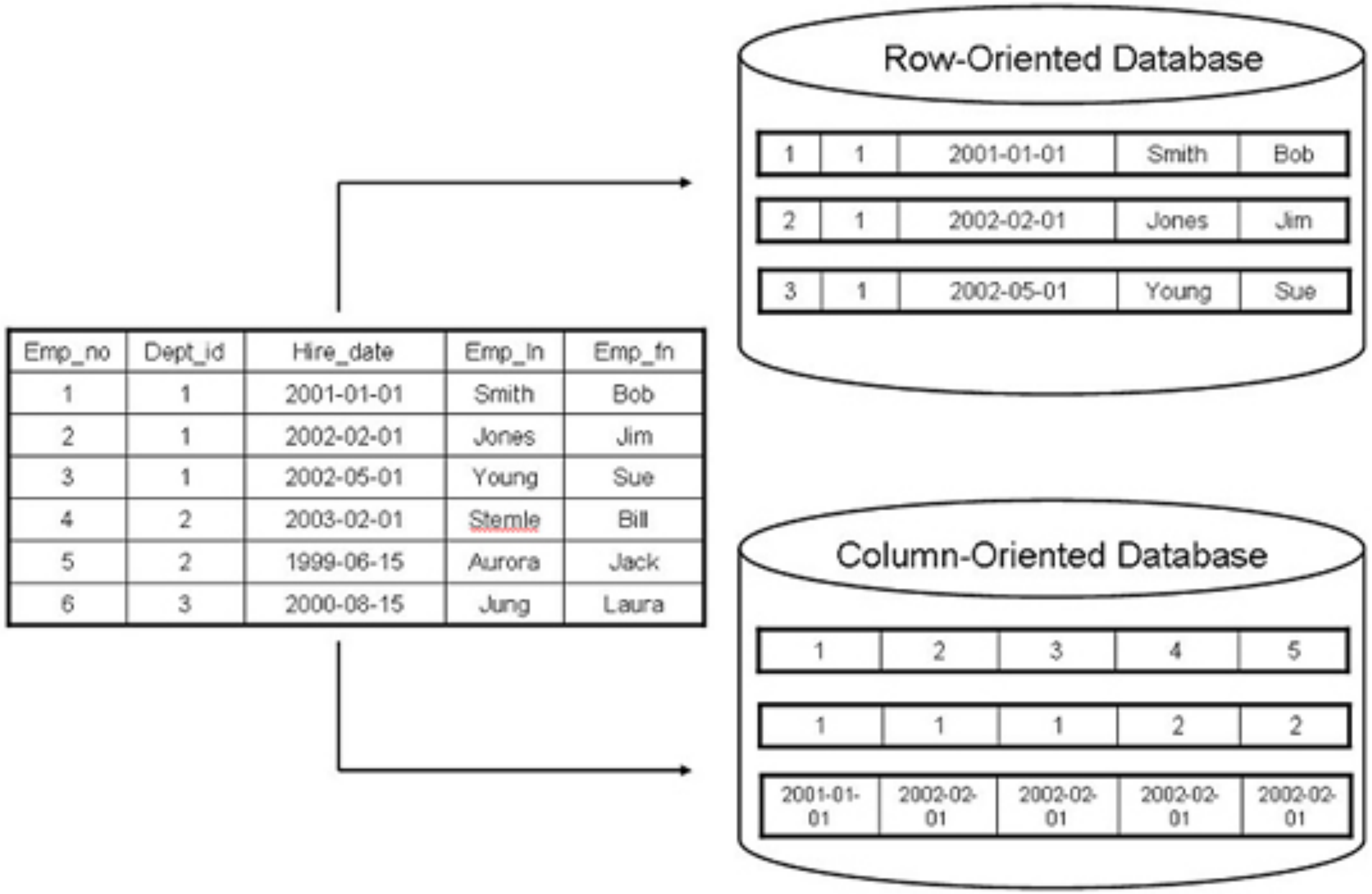
4

- The NoSQL era starts from LevelDB
-

Features of LevelDB

- Very high speed write: random writes=400,000/s, random reads=60,000/s
- LevelDb is different from Redis as it persists all data in disks。
- LevelDB stores data based on the values of keys. Therefore, close keys are stored together
- LevelDB only supports simple read/write operations.
- LevelDb support snapshots. So you can keep a version of data.

Row Based vs Column Based



Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura

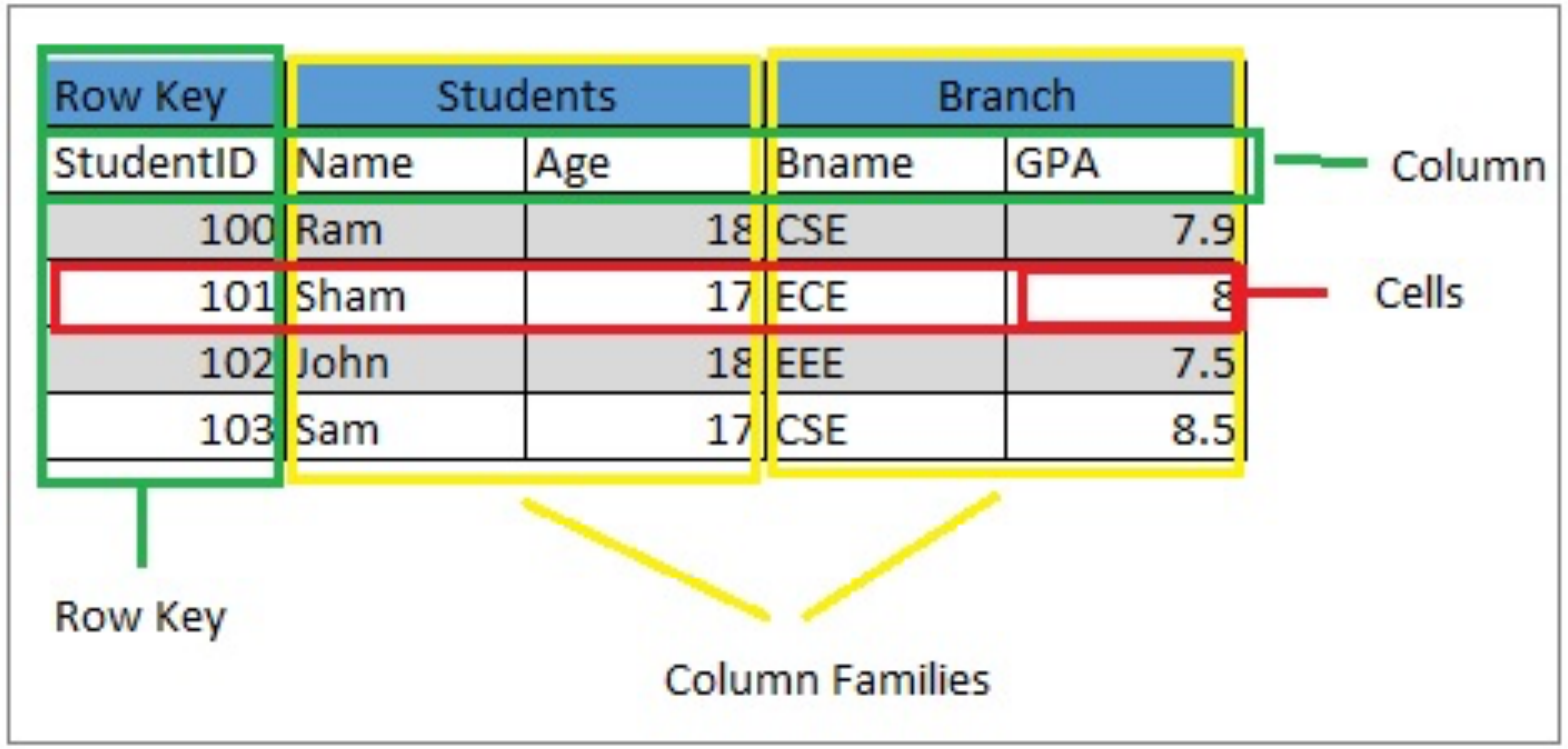
Row-Oriented Database

1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue

Column-Oriented Database

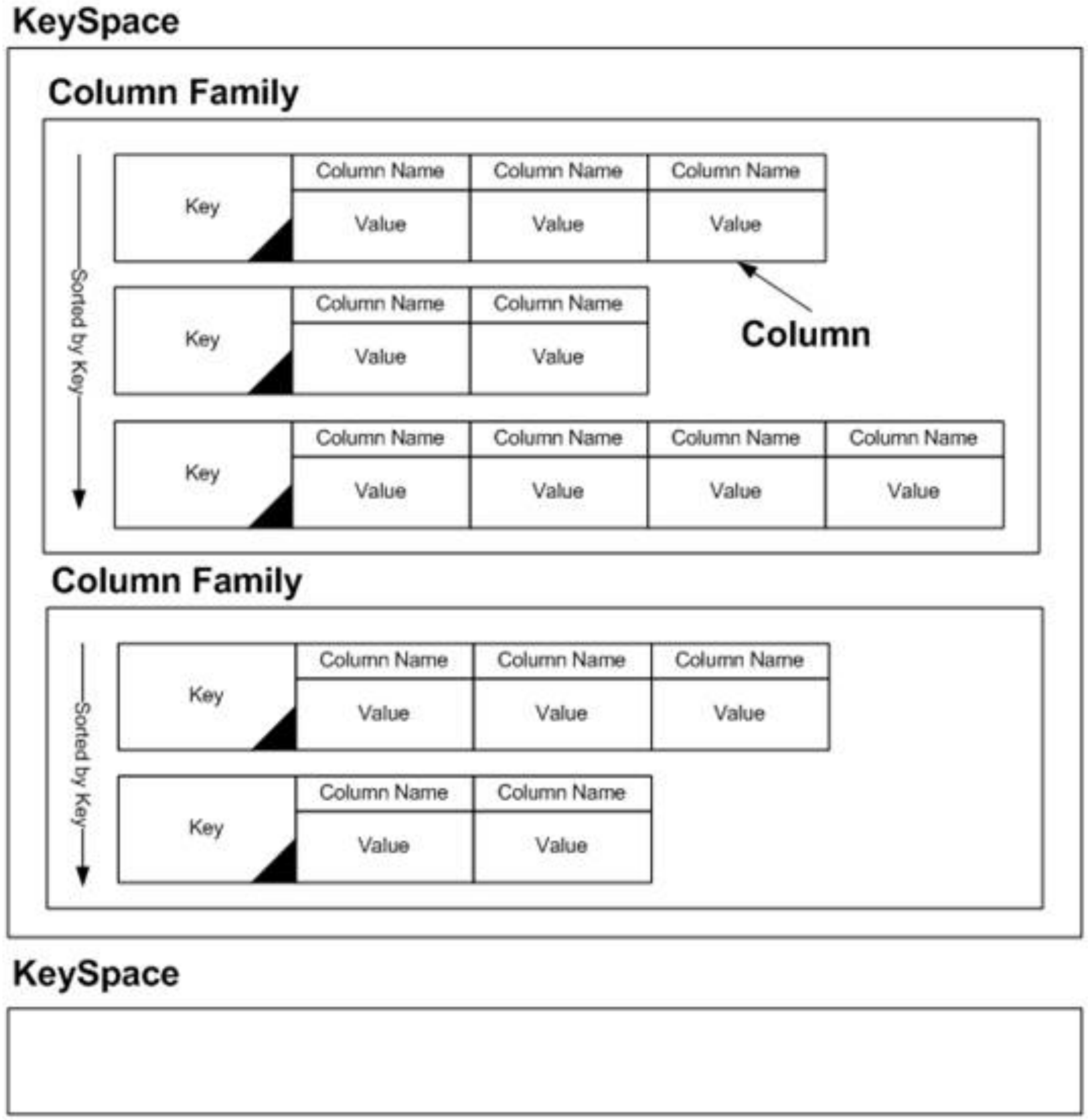
1	2	3	4	5
1	1	1	2	2
2001-01-01	2002-02-01	2002-02-01	2002-02-01	2002-02-01

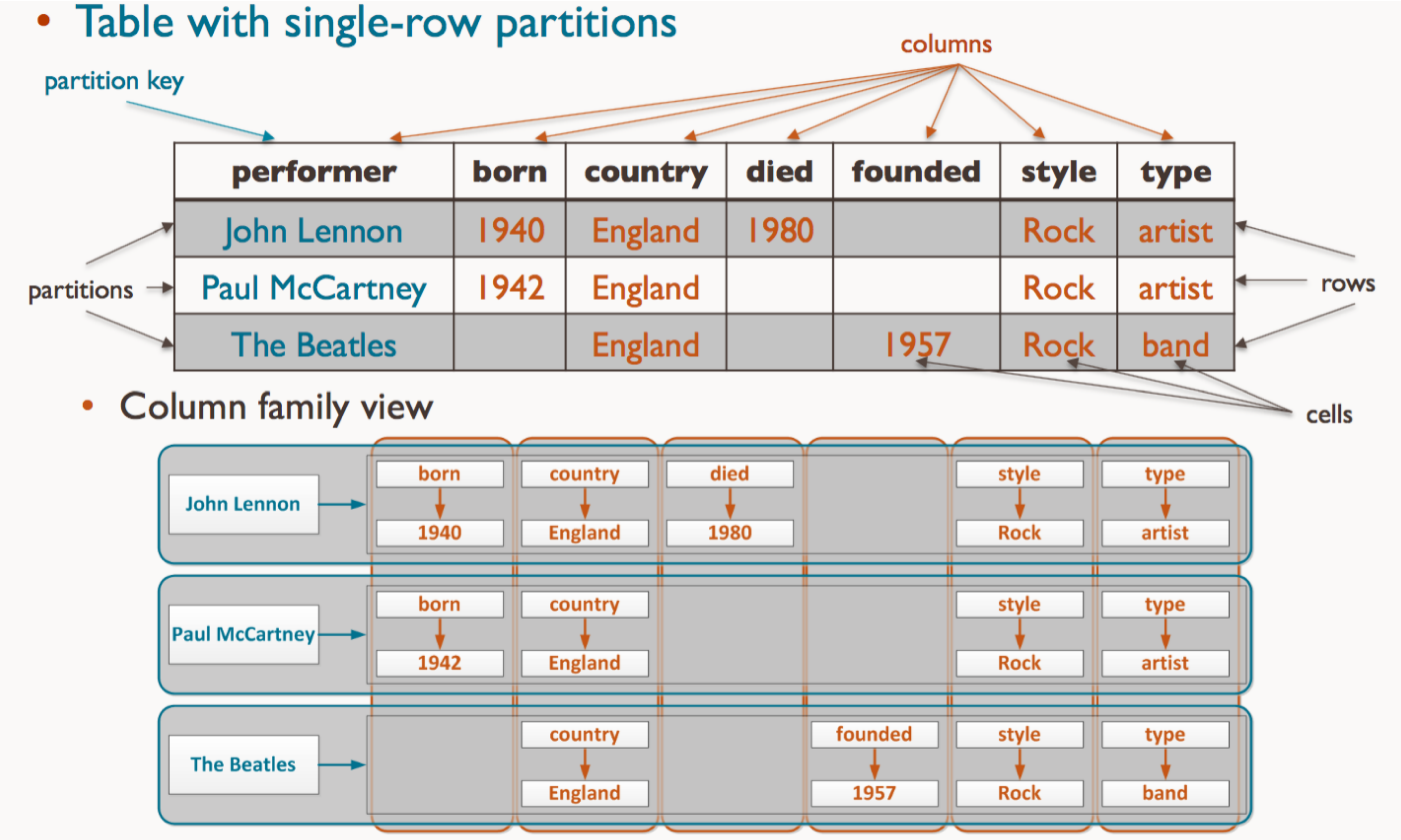
Storage of Column Family



Row Key	Students		Branch	
StudentID	Name	Age	Bname	GPA
100	Ram	18	CSE	7.9
101	Sham	17	ECE	8
102	John	18	EEE	7.5
103	Sam	17	CSE	8.5

Column
Cells
Row Key
Column Families





- Table with single-row partitions

performer	born	country	died	founded	style	type
John Lennon	1940	England	1980		Rock	artist
Paul McCartney	1942	England			Rock	artist
The Beatles		England		1957	Rock	band

- Column family view

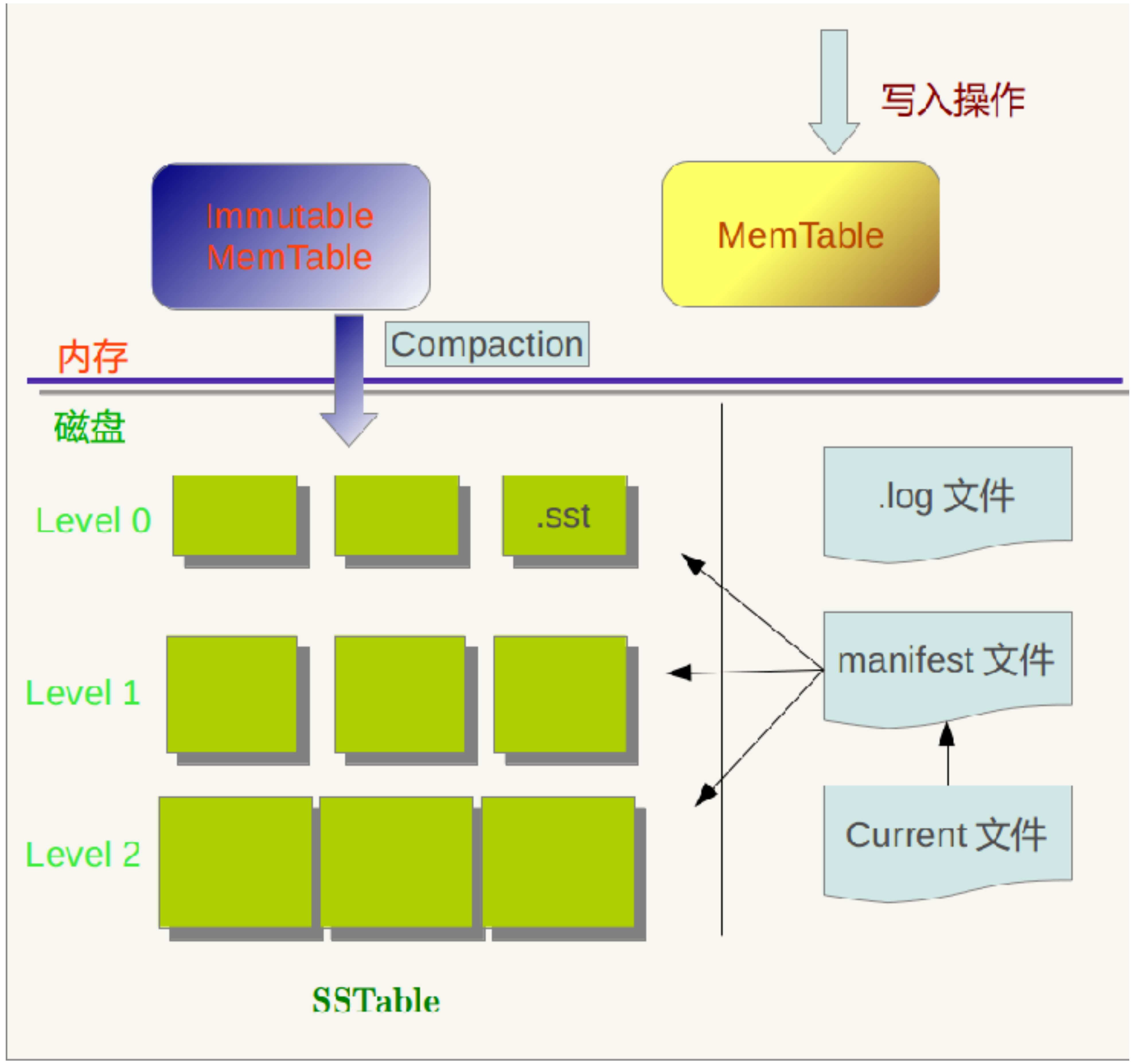
John Lennon →	born 1940	country England	died 1980		style Rock	type artist
Paul McCartney →	born 1942	country England			style Rock	type artist
The Beatles →		country England		founded 1957	style Rock	type band

Timestamp
µseconds
Column key
string = family:qualifier
Row key
string in lexicographic order
Column Family example with only one column key
Column Family example with variable multiple column keys
Column Family contents
Column Family language
Column Family anchor
Column Family name
contents:
language:
anchor:cnnsi.com
anchor:my.look.ca
com.cnn.www
com.google.www
com.lego.com
org.apache.hadoop
<html>
<html>
<html>...
← t3
← t5
← t6
EN
← t6
CNN
← t9
CNN.com
← t8
A cell is an un-interpreted array of bytes
Cell versions are custom garbage-collected by column family:
• Keep the last N versions
Tablet 1

The Idea of Key-Value Format

Architecture of LevelDB

- LevelDB has many layers
- In memory, there is an LSM (Log-Structured Merge tree) to maintain index. If the LSM is full, it will be locked as a Memtable.
- Disk part is partitioned into Level 0 - Level K. Each level is N (N=10 by default) times larger than previous one.
- If one level is full, it needs to be flushed and merged with the next level. The process is called compaction



API of LevelDB

```
DB() { };
virtual ~DB();
static Status Open(const Options& options,
                   const std::string& name,
                   DB** dbptr);
virtual Status Put(const WriteOptions& options,
                   const Slice& key,
                   const Slice& value) = 0;
virtual Status Delete(const WriteOptions& options, const
Slice& key) = 0;
virtual Status Write(const WriteOptions& options, WriteBatch*
updates) = 0;
```

```
virtual Status Get(const ReadOptions& options,
                   const Slice& key, std::string* value) = 0;
virtual Iterator* NewIterator(const ReadOptions& options) = 0;
virtual const Snapshot* GetSnapshot() = 0;
virtual void ReleaseSnapshot(const Snapshot* snapshot) = 0;
```

In Memory Table： MemTable

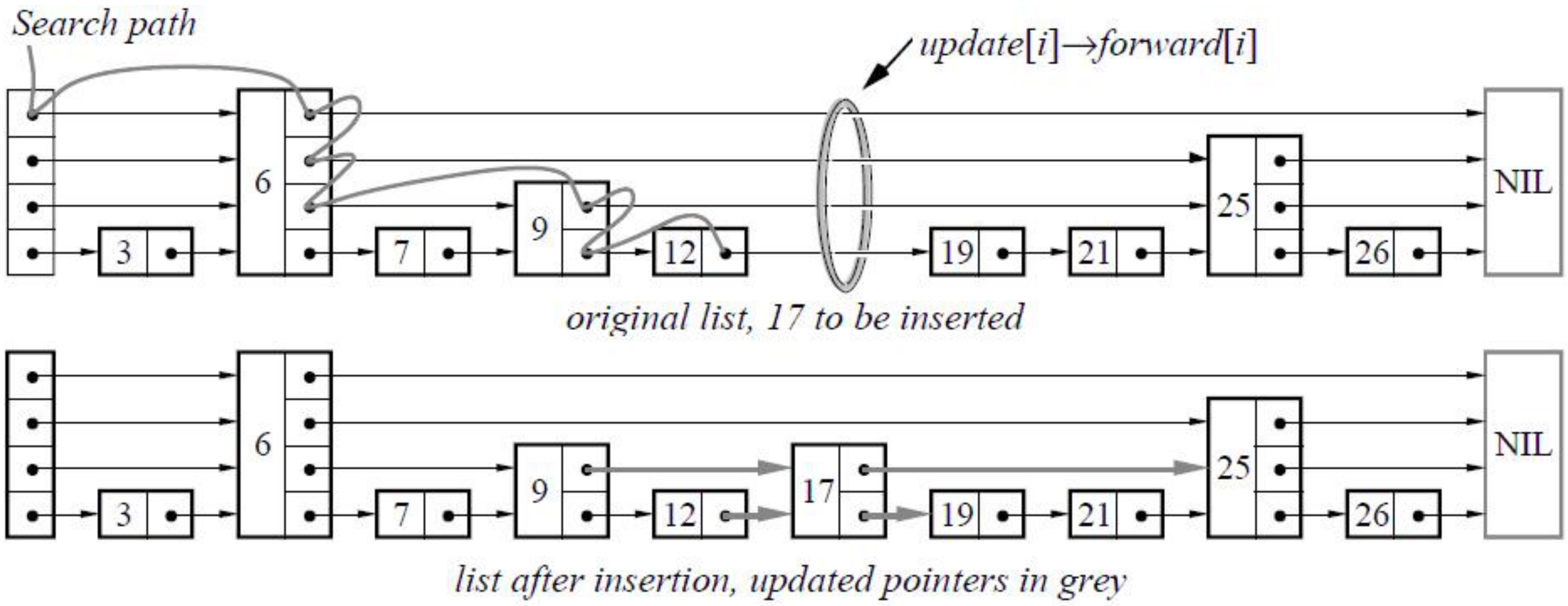
- All read/writes are first buffered in the MemTable
- By default， the size of a MemTable is 4M
- When a MemTable is full, the MemTable is locked. New updates will be sent to a newly created MemTable. The old one waits for compaction.

one question： Given the MemTable, how to efficiently locate the data?

We need an Index!

MemTable=SkipList

- Similar to the Binary Tree
- No need rotations in the tree to keep balanced
- Keep balanced with a high probability

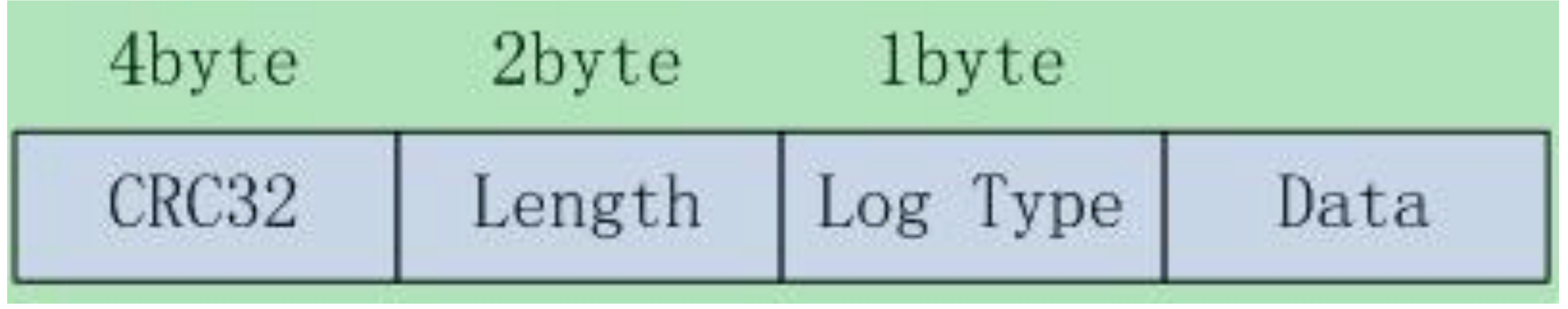


original list, 17 to be inserted

list after insertion, updated pointers in grey

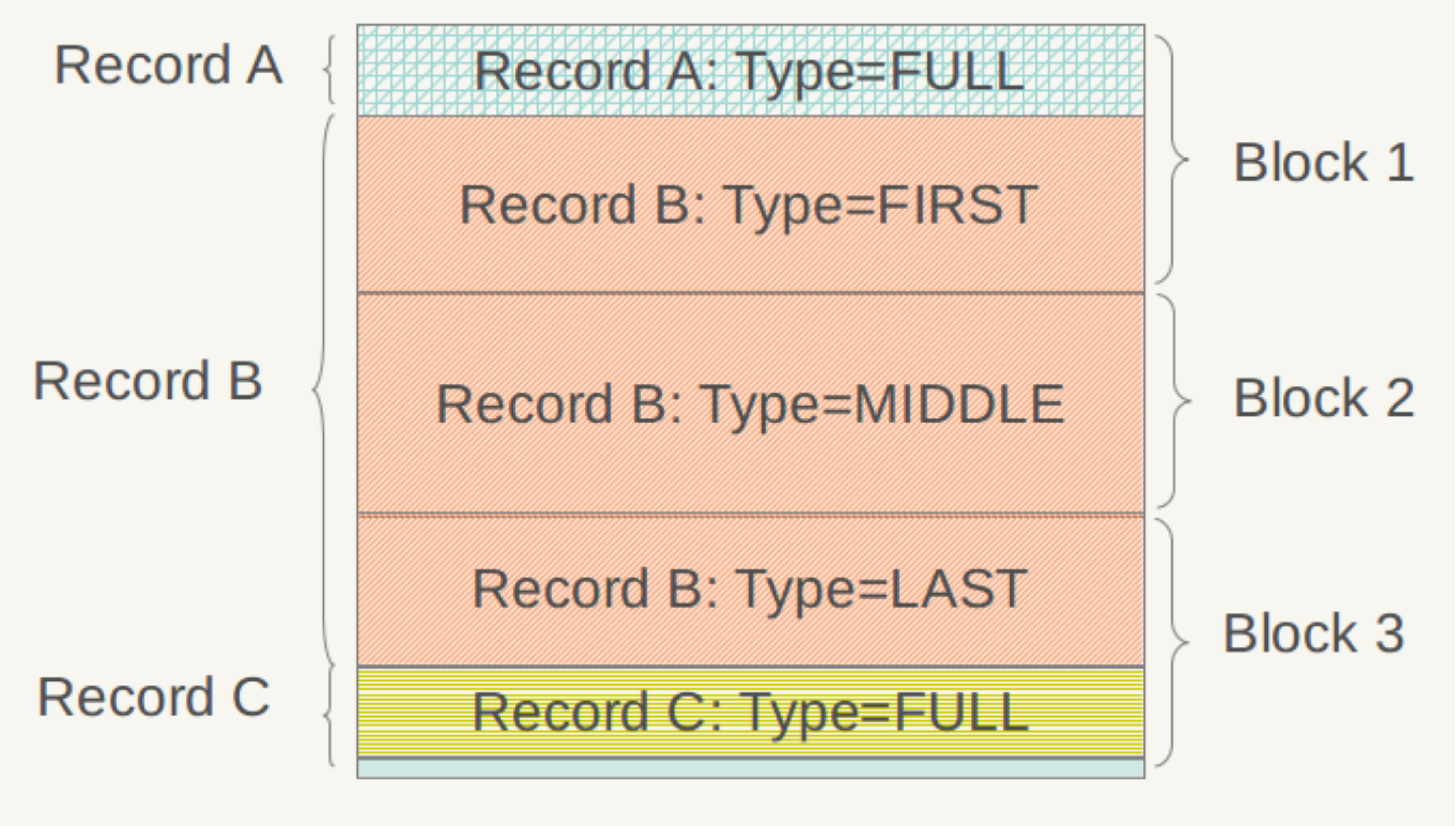
Do not Forget the Log

- To avoid data loss, before update data in memory, we write create a log record.
- In LevelDB，the format of log is：



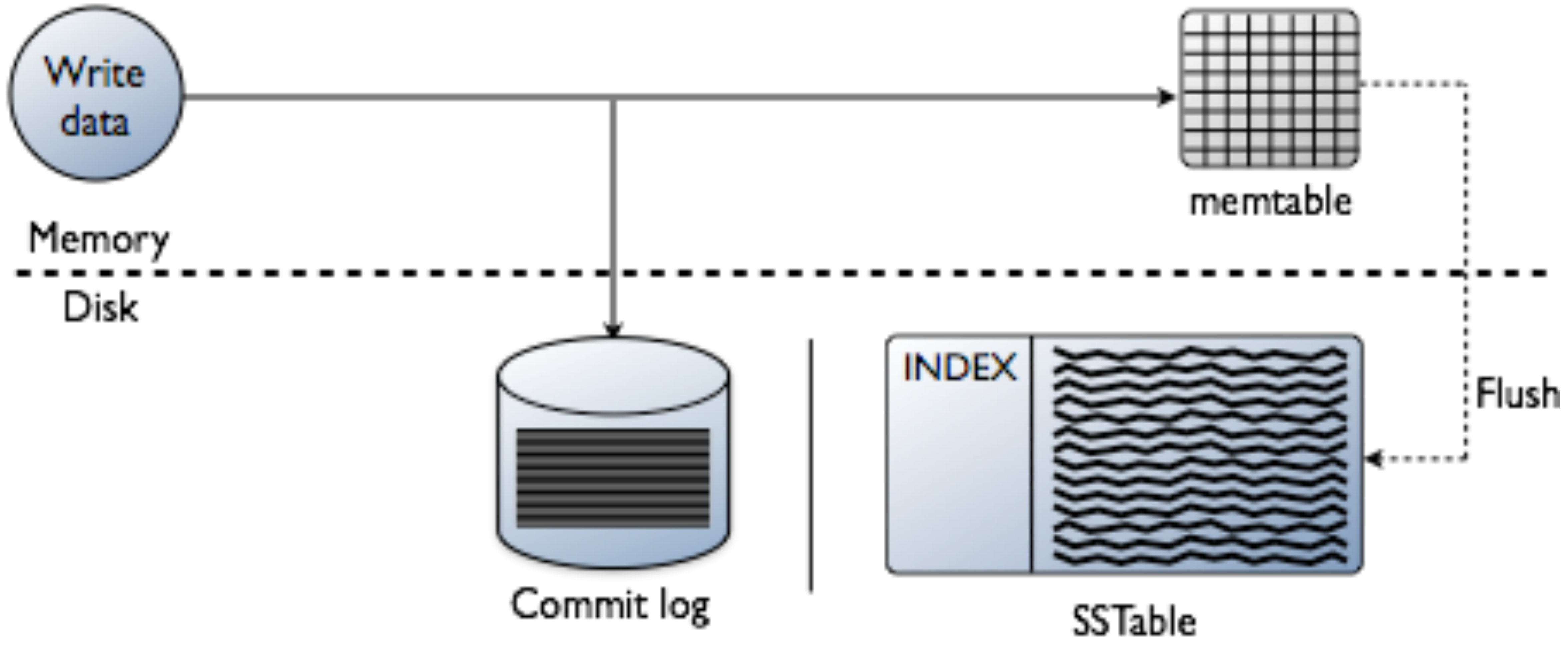
4byte	2byte	1byte	
CRC32	Length	Log Type	Data

- CRC32 is the verification code, length= record length，type=full, first, middle, last



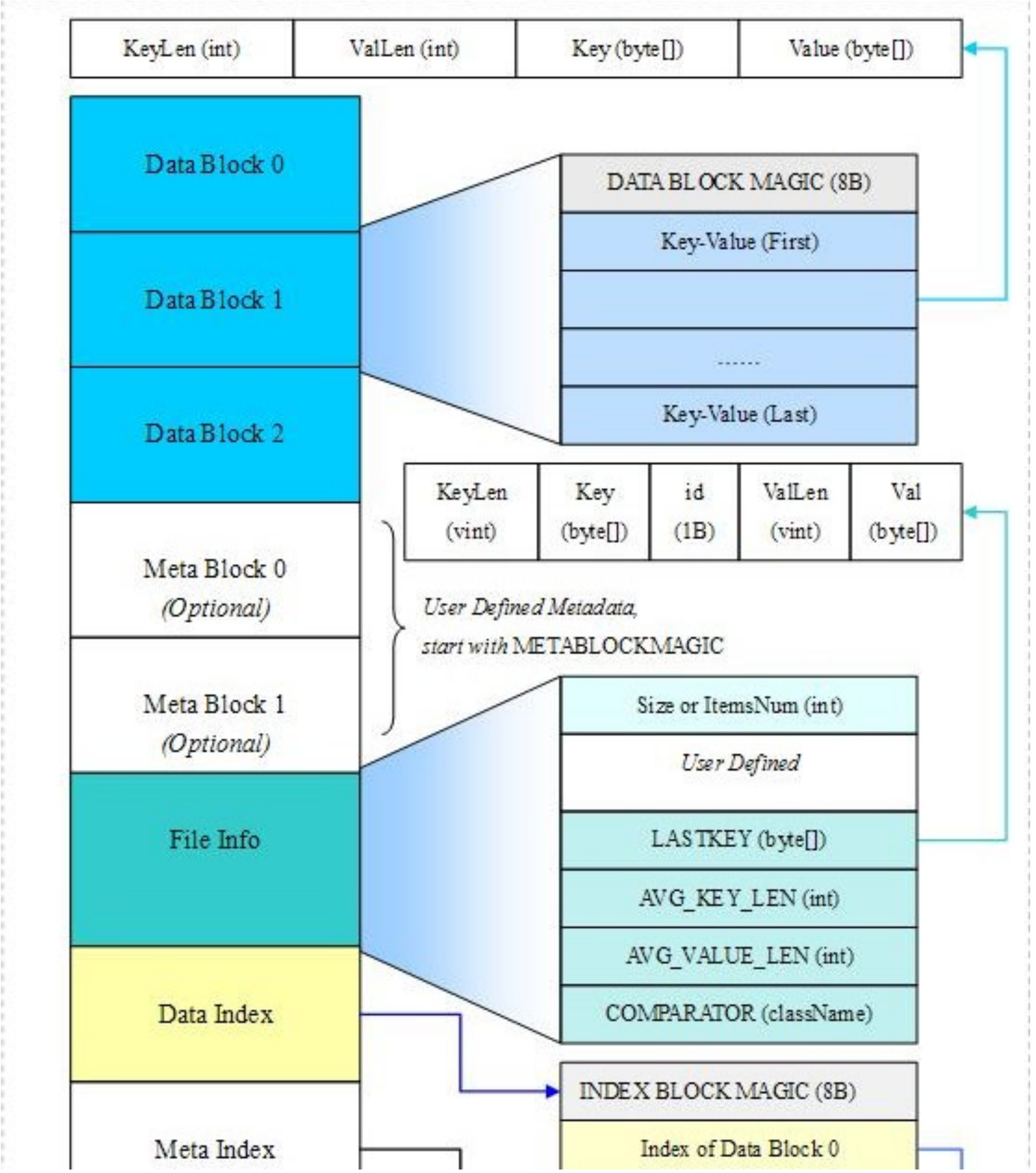
MemTable Updates

- If MemTable is full, we need to flush it to the disk.
- LevelDb will create a new Memtable and the corresponding log file. The old one is called the Immutable Memtable, which will be written as a SSTable in the disk.



Disk File： SSTable

- SSTable is the disk part of LevelDB. Each sstable is 2MB and belongs to one specific level：
  - level 0： at most 4 sstables
  - level 1： the total size of sstable less than 10M
  - level 2： the total size of sstable less than 1000M



KeyLen (int) | ValLen (int) | Key (byte[]) | Value (byte[])

Data Block 0

Data Block 1

Data Block 2

DATA BLOCK MAGIC (8B)

Key-Value (First)

......

Key-Value (Last)

KeyLen (vint) | Key (byte[]) | id (1B) | ValLen (vint) | Val (byte[])

Meta Block 0 *(Optional)*

Meta Block 1 *(Optional)*

User Defined Metadata, start with METABLOCKMAGIC

Size or ItemsNum (int)

User Defined

LASTKEY (byte[])

AVG_KEY_LEN (int)

AVG_VALUE_LEN (int)

COMPARATOR (className)

File Info

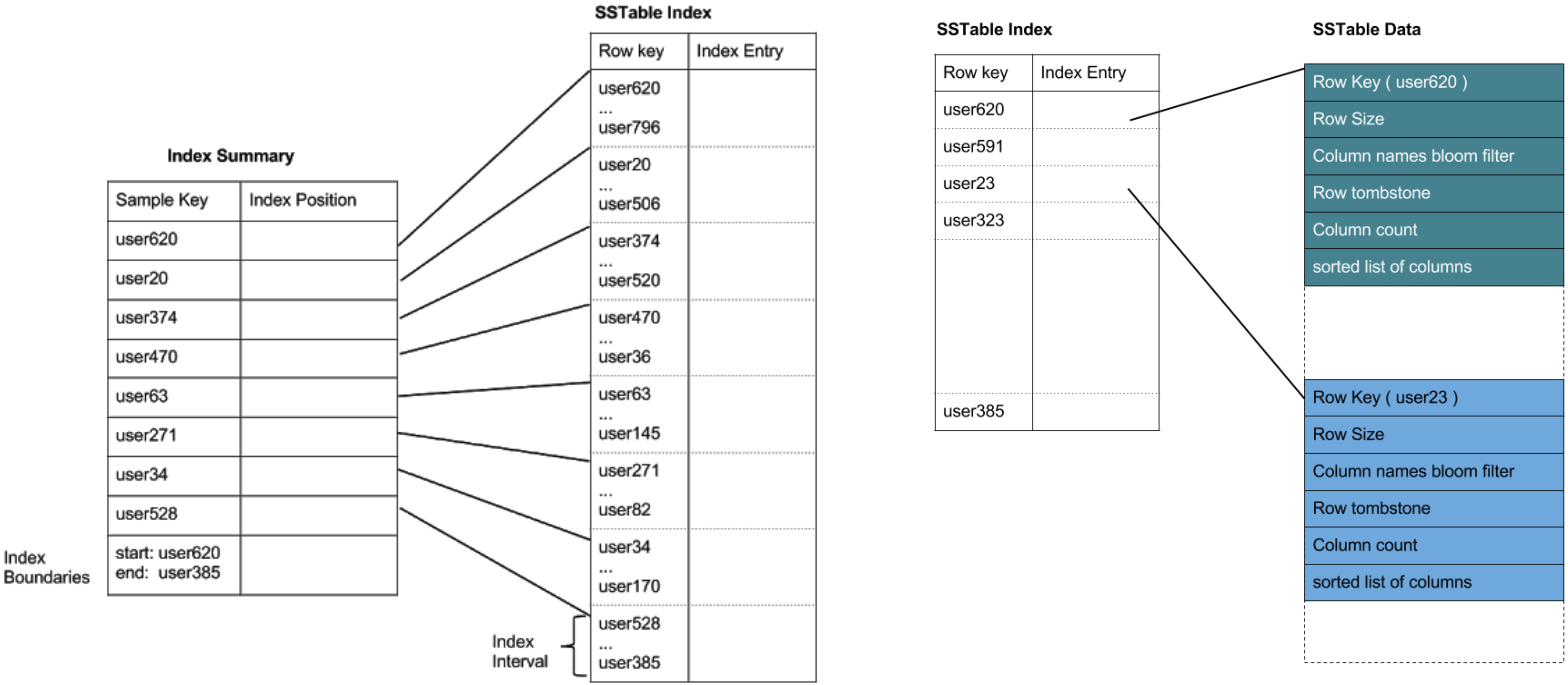
Data Index

INDEX BLOCK MAGIC (8B)

Index of Data Block 0

Meta Index

BlockIndex



Index Summary

Sample Key	Index Position
user620	
user20	
user374	
user470	
user63	
user271	
user34	
user528	
start: user620 end: user385	

SSTable Index

Row key	Index Entry
user620 ... user796	
user20 ... user506	
user374 ... user520	
user470 ... user36	
user63 ... user145	
user271 ... user82	
user34 ... user170	
user528 ... user385	

SSTable Index

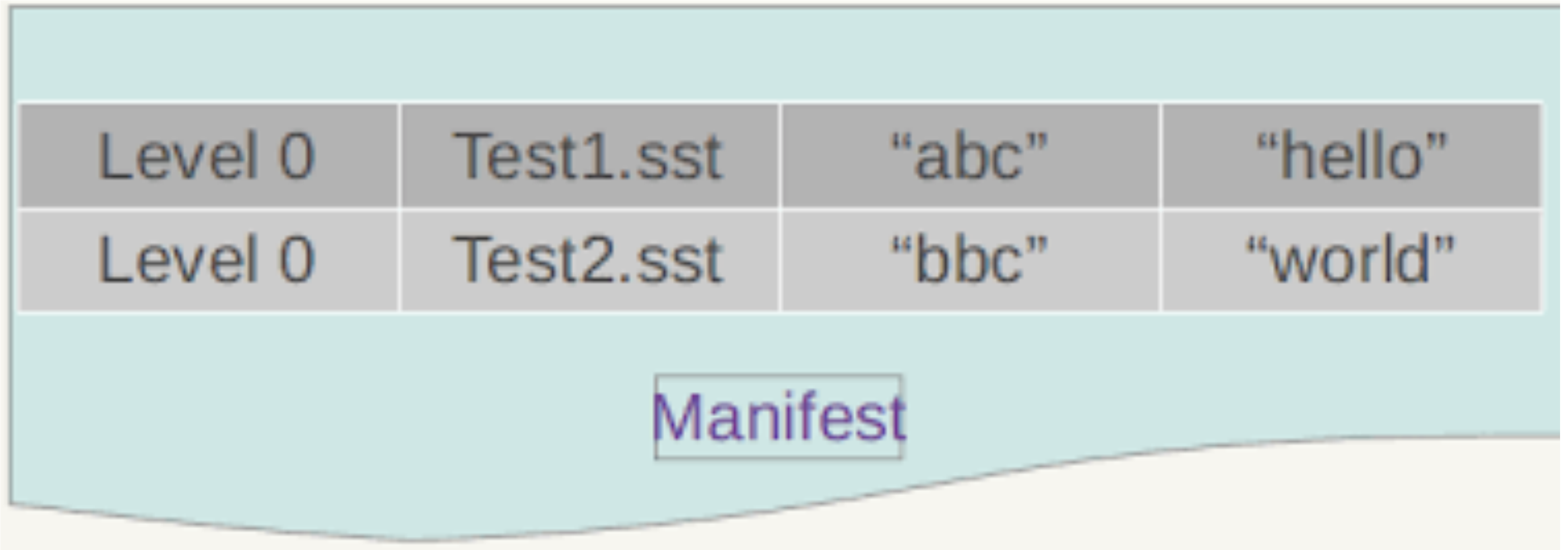
Row key	Index Entry
user620	
user591	
user23	
user323	
user385	

SSTable Data

Row Key (user620)
Row Size
Column names bloom filter
Row tombstone
Column count
sorted list of columns
Row Key (user23)
Row Size
Column names bloom filter
Row tombstone
Column count
sorted list of columns

Meta-Index

- The search problem remains. We cannot afford searching every SSTable for a single key.
- We need to know whether a key DOES exist in a block or not!
- One specific index: BloomFilter is introduced

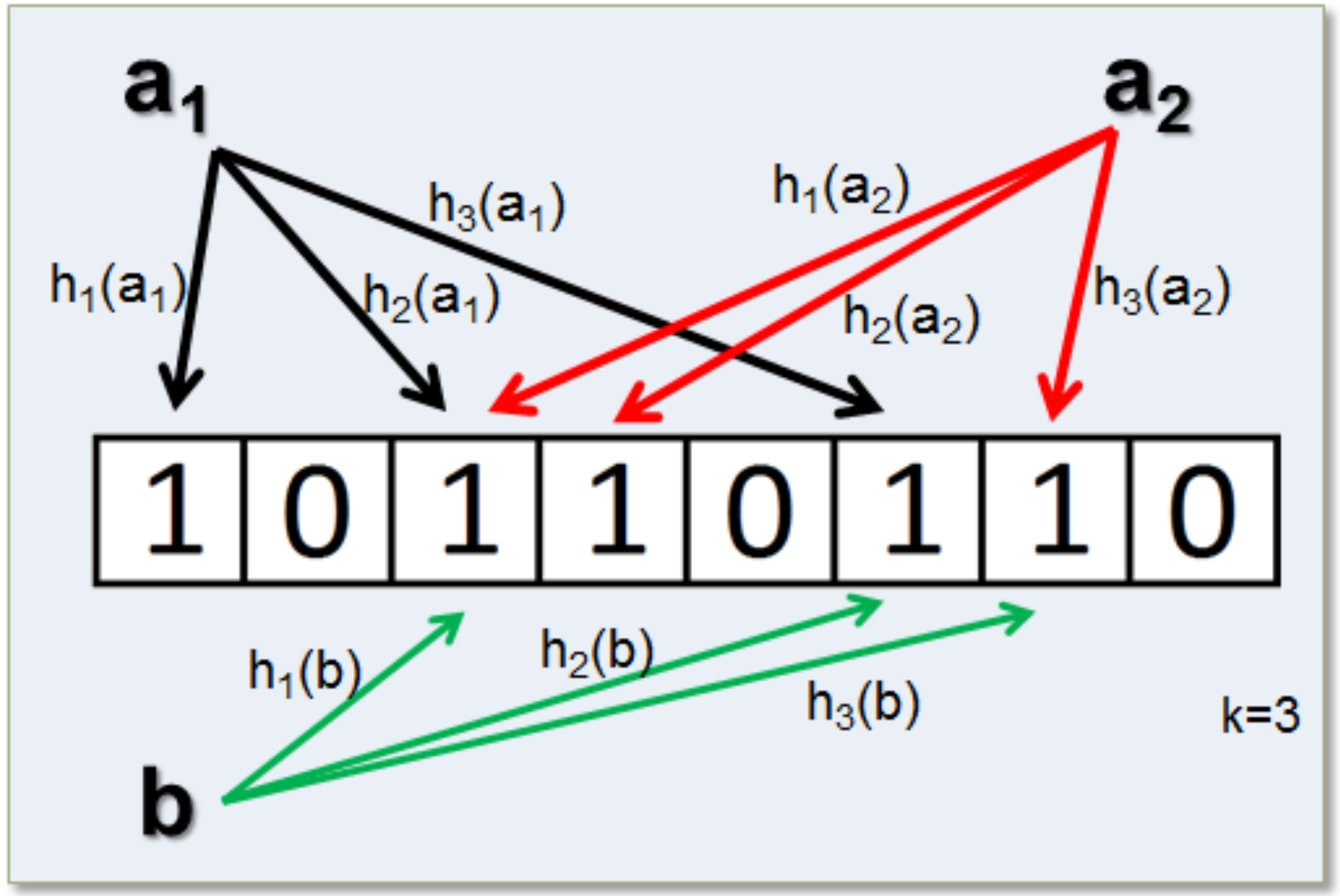


Level 0	Test1.sst	“abc”	“hello”
Level 0	Test2.sst	“bbc”	“world”

Manifest

Bloomfilter

- Bloomfilter=bitmap, Initialized all 0s. suppose we have total m bits.
- We need K independent Hash function. For every value v, we have h1(v), h2(v),…,hk(v) hash values, which mapped v into k positions from 0 to m.
- We set values in those K positions to 1

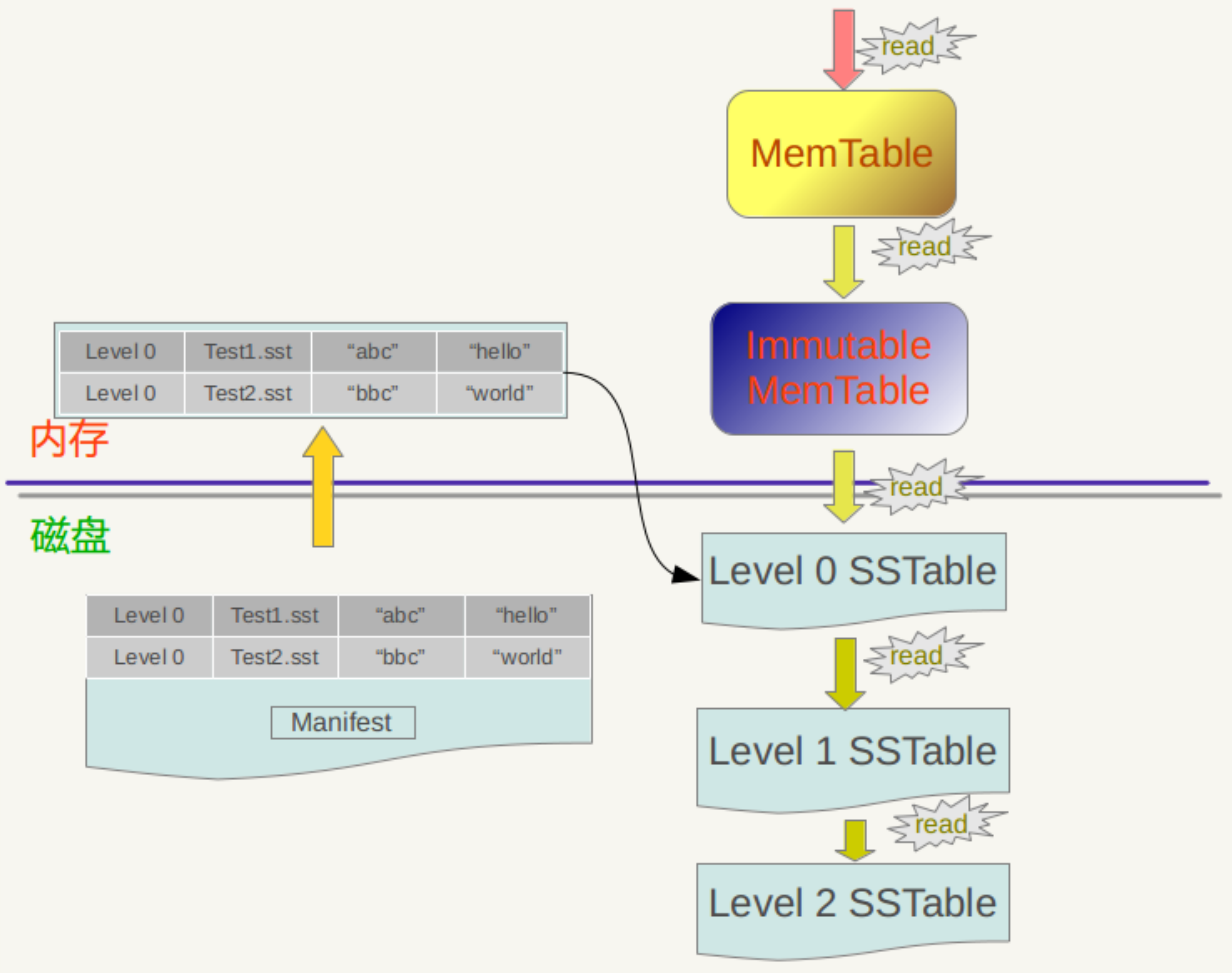


Some Fact of Bloomfilter

- The probability that a certain bit is not set to 1 by a certain hash function during the insertion of an element: 1 - 1/m
- k is the number of hash functions, the probability that the bit is not set to 1 by any of the hash functions is: (1-1/m)^k
- If we have inserted n elements, the probability that a certain bit is still 0： (1-1/m)^{nk}
- the probability that it is 1 is therefore: 1-(1-1/m)^{nk}
- The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is often given as

$$\left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}.$$

Read in LevelDB

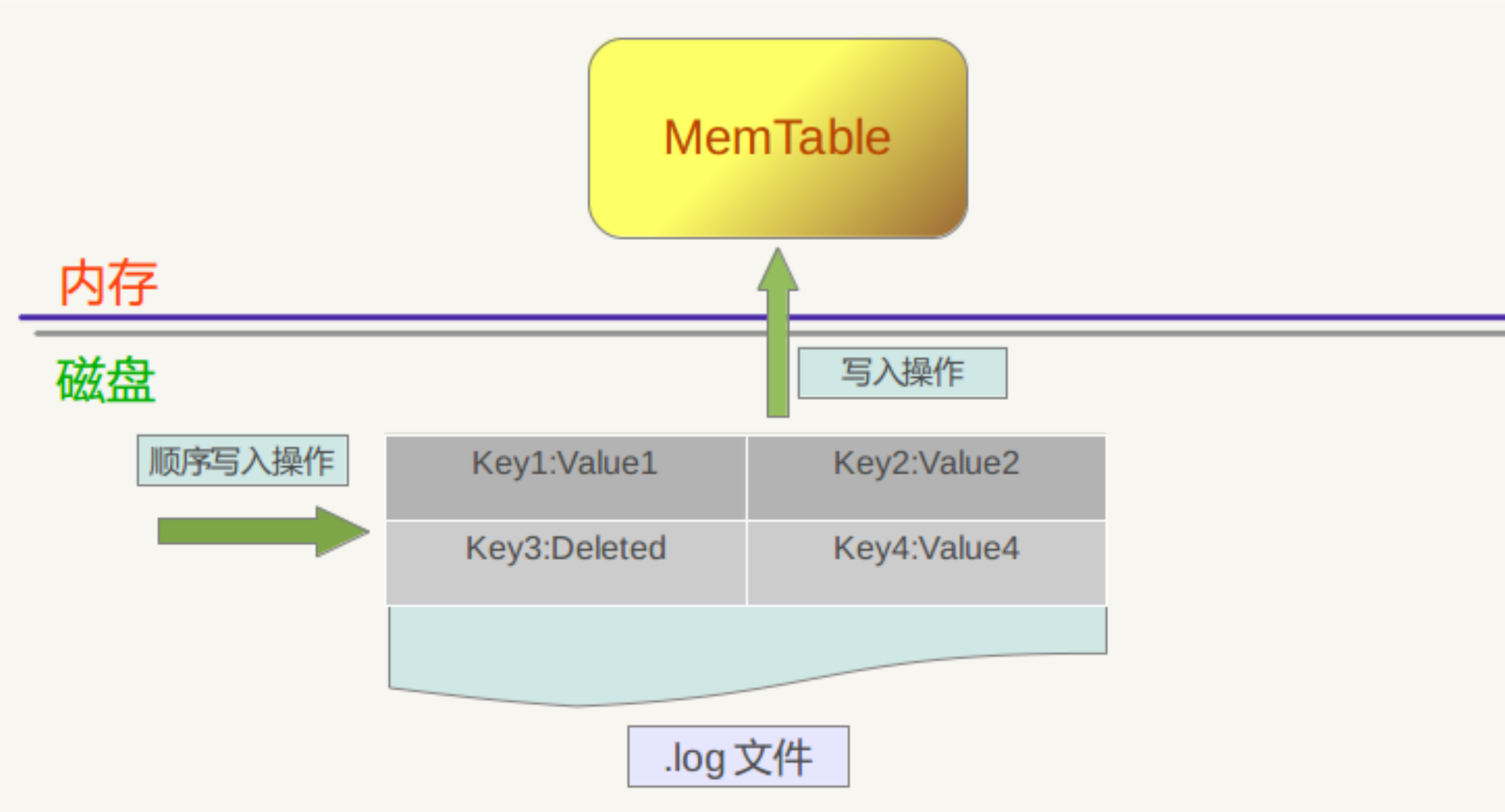


Level 0	Test1.sst	“abc”	“hello”
Level 0	Test2.sst	“bbc”	“world”

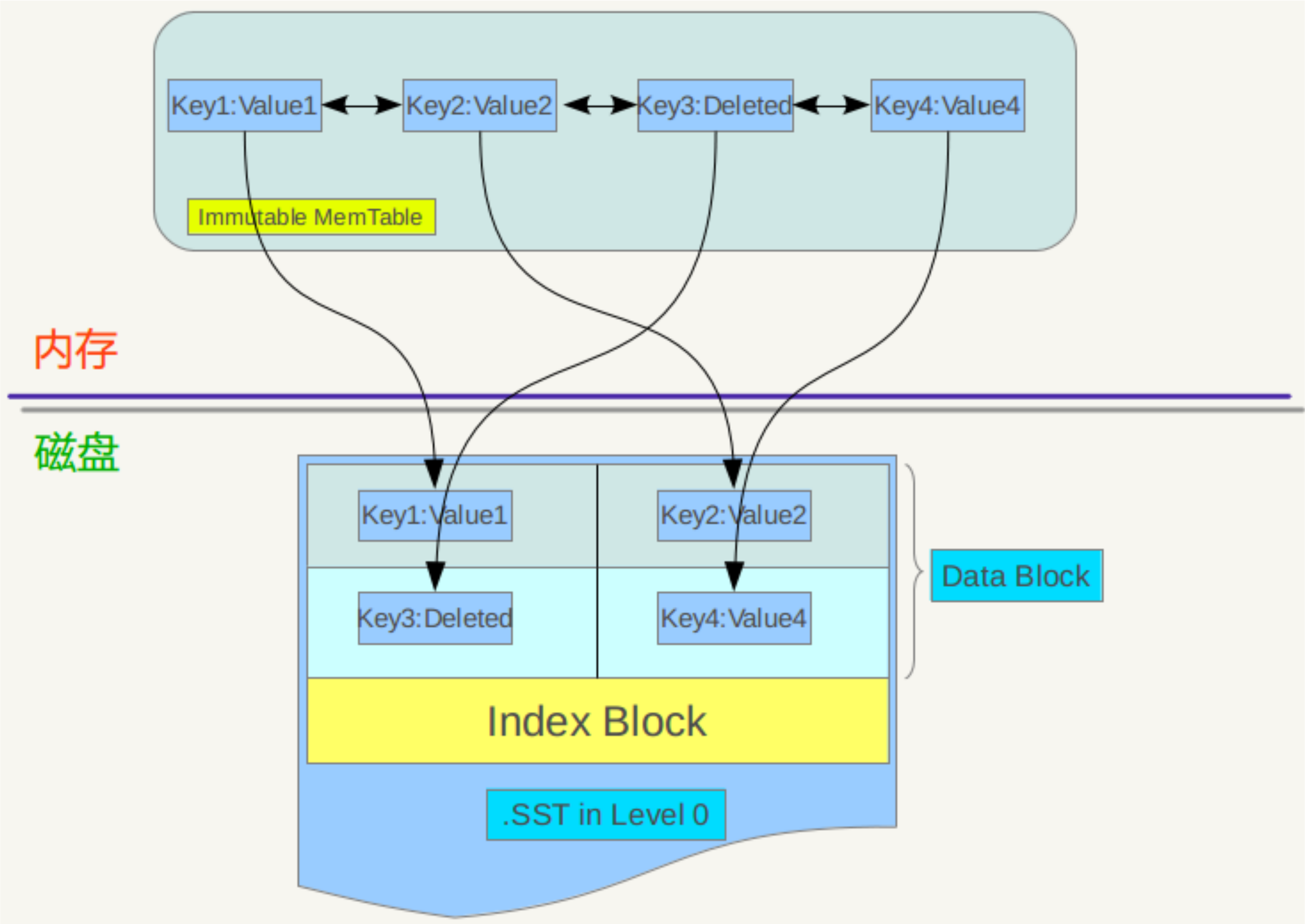
Level 0	Test1.sst	“abc”	“hello”
Level 0	Test2.sst	“bbc”	“world”

Write in LevelDB

- As mentioned before, if MemTable is full, we flushed it to disk as SSTable. But what if SSTable is full?
- We need Compaction
  - minor Compaction: merge memtable with level 0 SSTable
  - major compaction: merge SSTables in different levels
  - full compaction: merge all SSTables

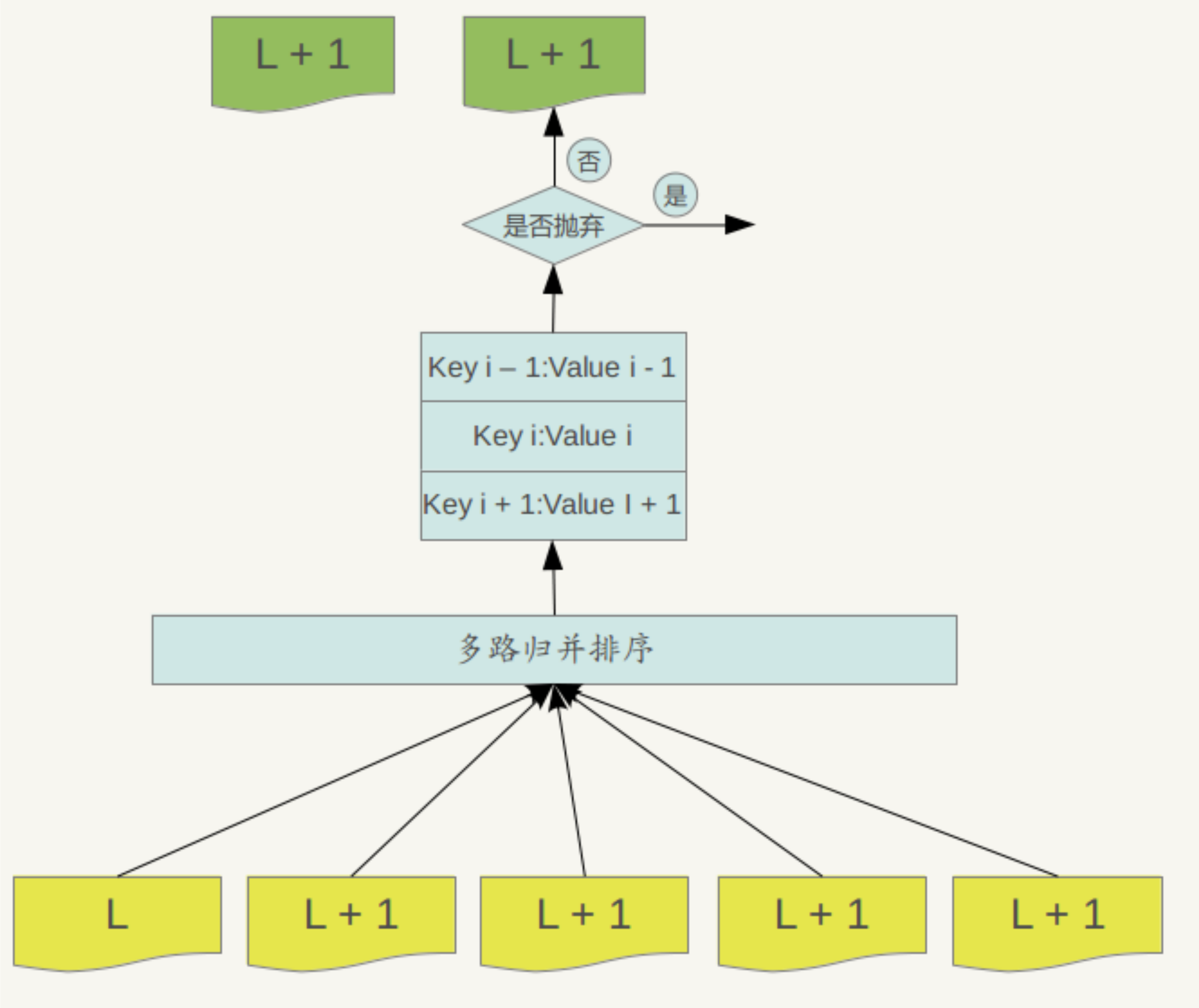


Minor Compaction



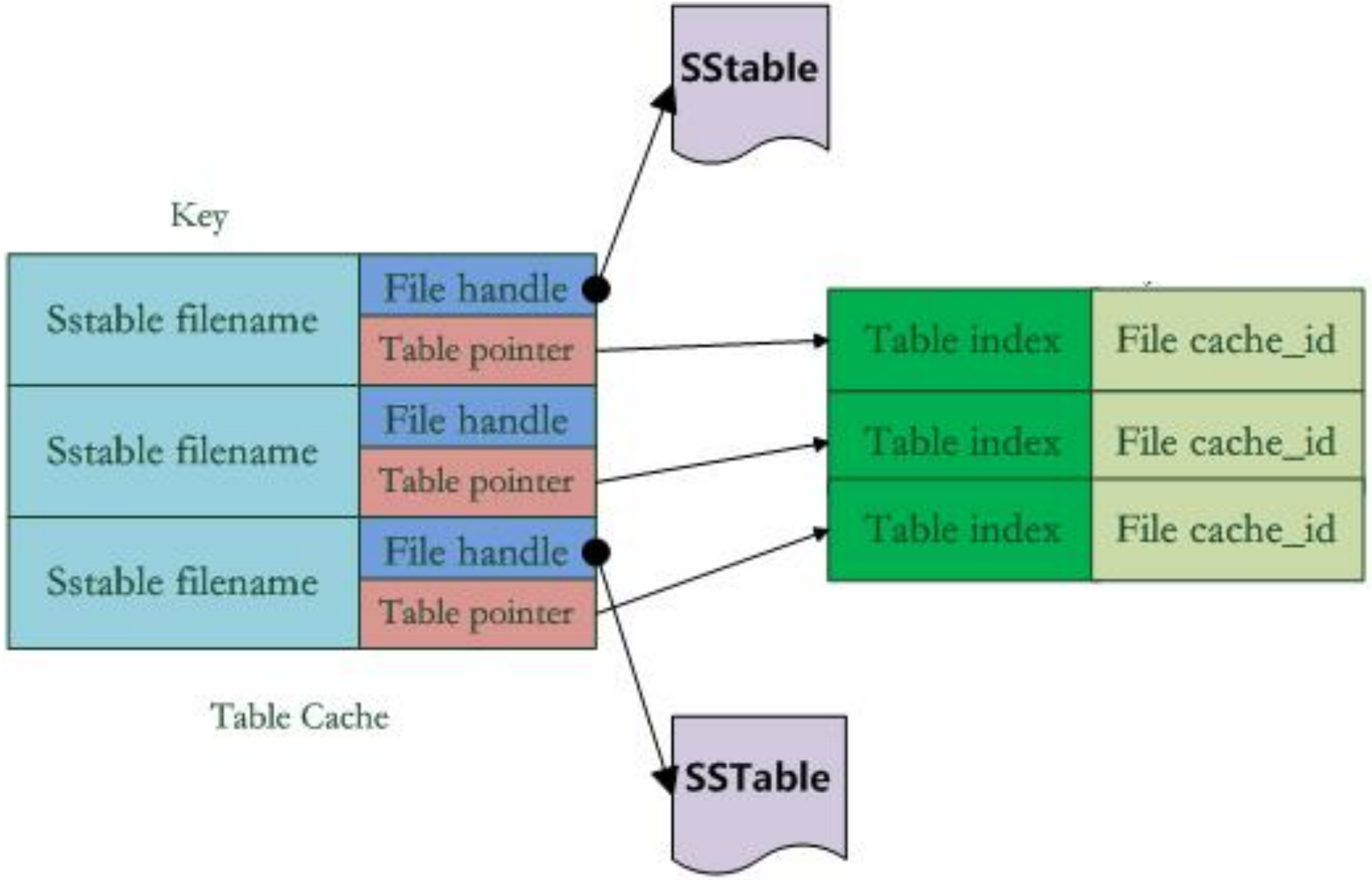
Major Compaction

- select a SSTable file in level L
- select SSTable files in level L+1 that have overlapped key ranges
- merge them together
- Note： For level 0, more than 1 SSTable files in level L are involved!



Still too slow： Cache

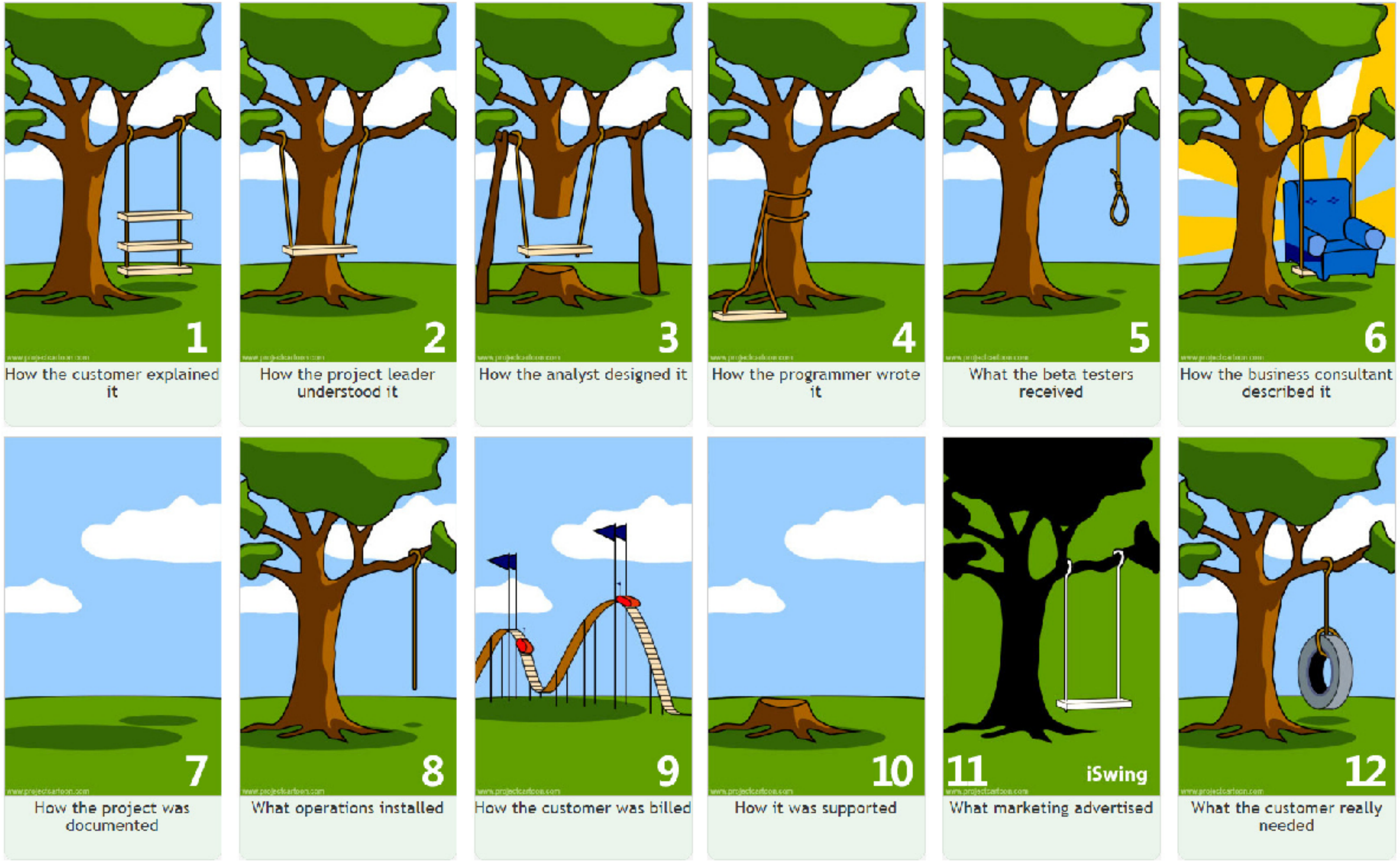
- Even if the data reside in level 0, we need to first check the Memtable and then the SSTable, which is slow.
- What if, the data are in level K?
- To avoid the case, we use a cache to maintain all hot data.



The Most Challenging Task: Multithreads

- It is OK for multi-threads to read together
- But if they write together, there may be problem.
- To guarantee the correctness, we need to lock the whole skiplist.
- That is too slow

What if you are the designer


1
How the customer explained it
2
How the project leader understood it
3
How the analyst designed it
4
How the programmer wrote it
5
What the beta testers received
6
How the business consultant described it
7
How the project was documented
8
What operations installed
9
How the customer was billed
10
How it was supported
11
iSwing
What marketing advertised
12
What the customer really needed
www.projectcartoon.com

One Solution （not the best)

- All writes are buffered in a queue and proceed one by one.
- The process who gets the access to the write operations also handles the read operations.

```
Status DBImpl::Write(const WriteOptions& options, WriteBatch* my_batch) {
  // A begin
  Writer w(&mutex_);
  w.batch = my_batch;
  w.sync = options.sync;
  w.done = false;
  // A end

  // B begin
  MutexLock l(&mutex_);
  writers_.push_back(&w);
  while (!w.done && &w != writers_.front()) {
    w.cv.Wait();
  }
  if (w.done) {
    return w.status;
  }
  // B end
```

Code Continues…

```
// May temporarily unlock and wait.
  Status status = MakeRoomForWrite(my_batch == NULL);
  uint64_t last_sequence = versions_->LastSequence();
  Writer* last_writer = &w;
  if (status.ok() && my_batch != NULL) {  // NULL batch is for compactions
    WriteBatch* updates = BuildBatchGroup(&last_writer);
    WriteBatchInternal::SetSequence(updates, last_sequence + 1);
    last_sequence += WriteBatchInternal::Count(updates);

    // Add to log and apply to memtable.  We can release the lock
    // during this phase since &w is currently responsible for logging
    // and protects against concurrent loggers and concurrent writes
    // into mem_.
    {
      mutex_.Unlock();
      status = log_->AddRecord(WriteBatchInternal::Contents(updates));
      bool sync_error = false;
      if (status.ok() && options.sync) {
        status = logfile_->Sync();
        if (!status.ok()) {
          sync_error = true;
        }
      }
      if (status.ok()) {
        status = WriteBatchInternal::InsertInto(updates, mem_);
      }
```

And More

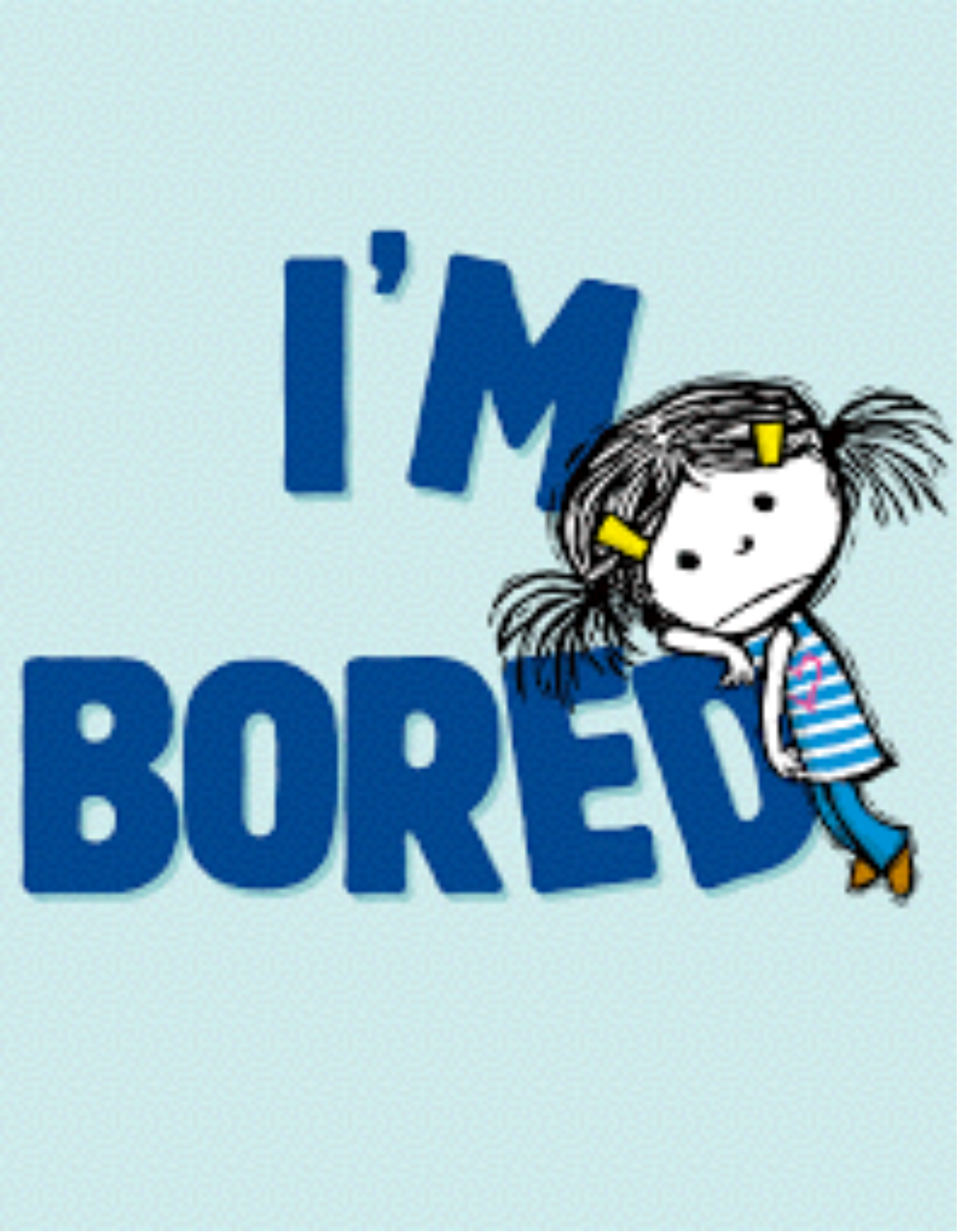
```
mutex_.Lock();
   if (sync_error) {
    RecordBackgroundError(status);
   }
  }
  if (updates == tmp_batch_) tmp_batch_->Clear();

  versions_->SetLastSequence(last_sequence);
 }

 while (true) {
  Writer* ready = writers_.front();
  writers_.pop_front();
  if (ready != &w) {
   ready->status = status;
   ready->done = true;
   ready->cv.Signal();
  }
  if (ready == last_writer) break;
 }

 // Notify new head of write queue
 if (!writers_.empty()) {
  writers_.front()->cv.Signal();
 }

 return status;
}
```



LevelDB Resources

- Website: http://leveldb.org/
- Source code: https://github.com/google/leveldb/releases
- papers:
  1. https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf
  2. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf
  3. http://citeseer.ist.psu.edu/viewdoc/download;jsessionid=6CA79DD1A90B3EFD3D62ACE5523B99E7?doi=10.1.1.127.9672&rep=rep1&type=pdf

LEVELDB Opensourced

- RocksDB: Facebook version, https://rocksdb.org
- HBase: https://hbase.apache.org

Homework: deploy Hbase in the Cloud and test