Cassandra under the hood

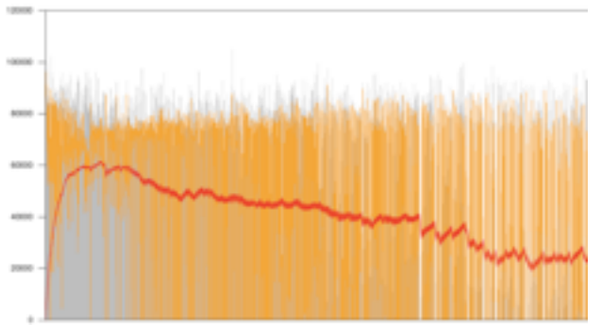
Richard Low
rlow@acunu.com

Outline

- What happens when you write?
  - Commit logs
  - Memtables
  - SSTables
- What happens when you read?
  - Point queries
  - Range queries
- Repair and snapshots

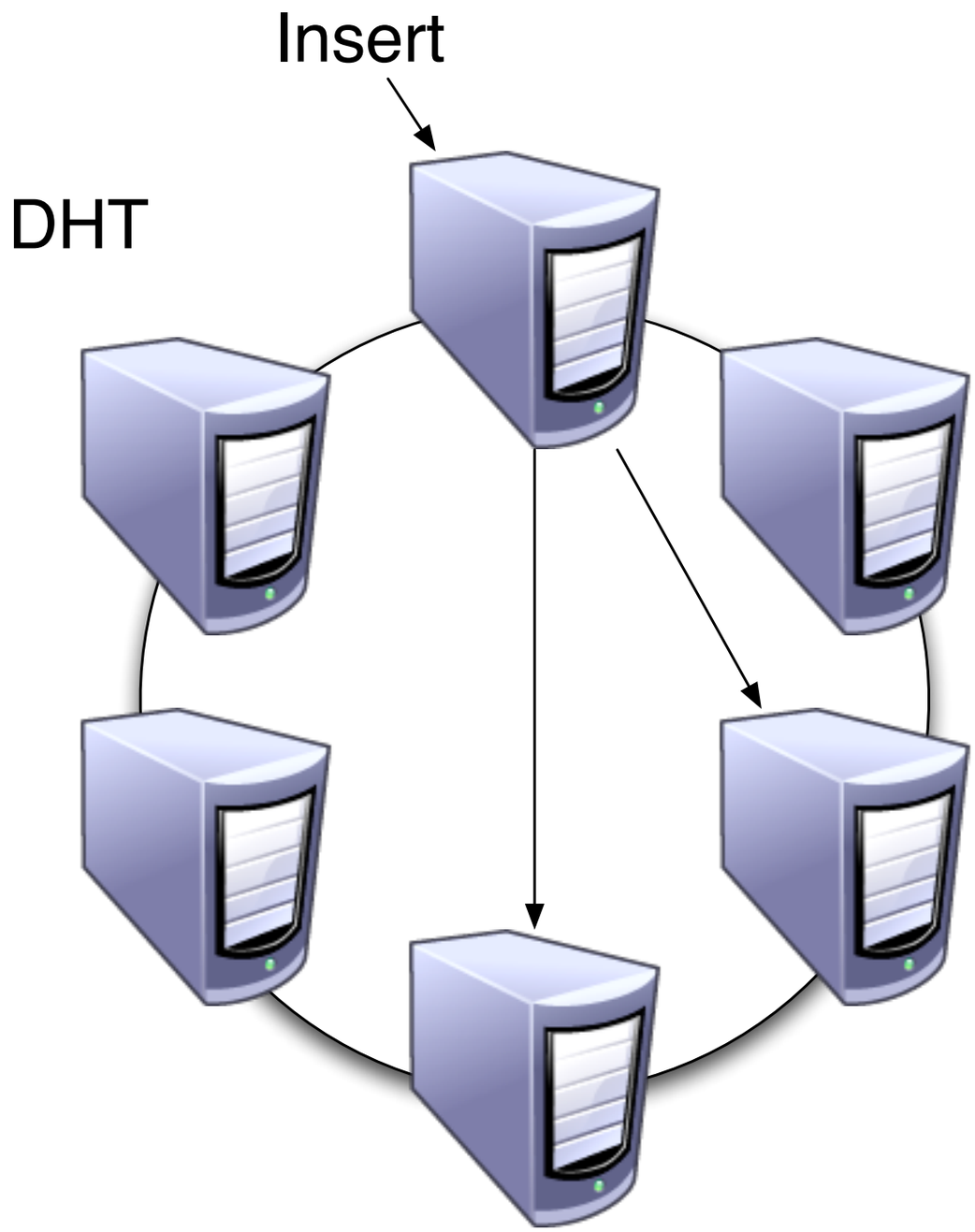
```
"richard":{
  "email":"rlow@acunu.com"
}
```

Why should we care?

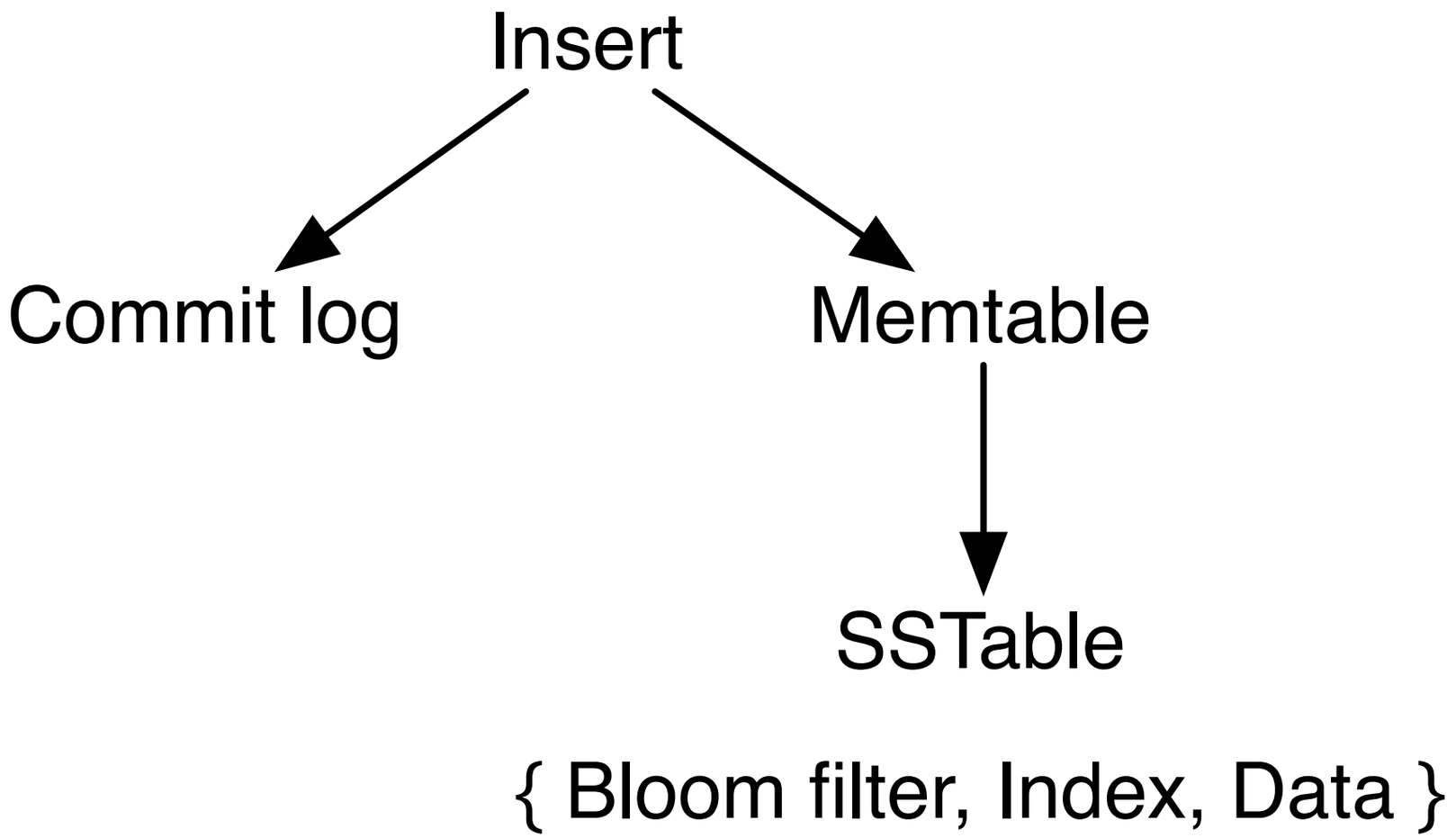


- Help understand performance
- Understand performance implications of data model
- Helps to fix it if something goes wrong
- Interesting!

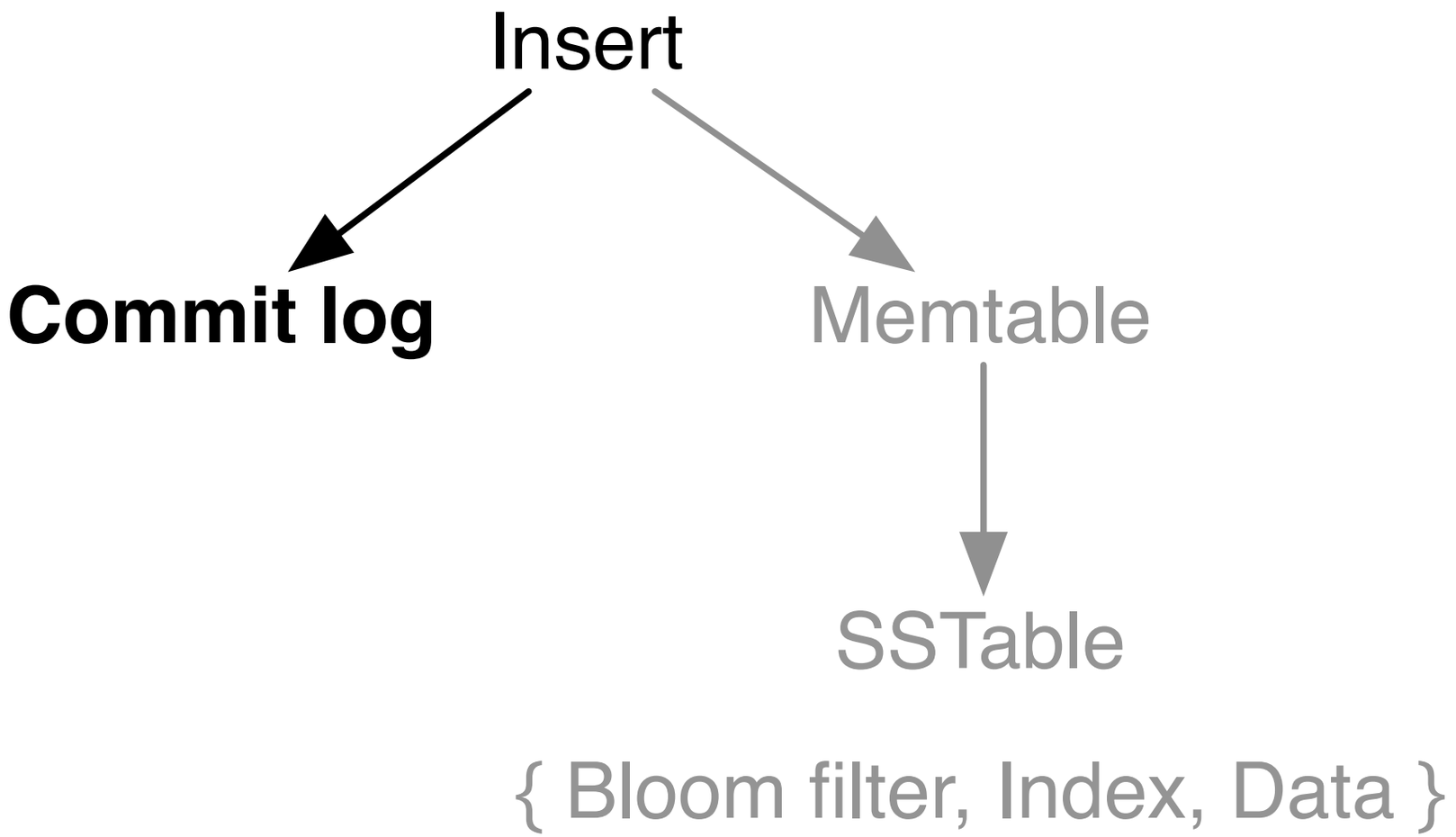
Writes



Writes (2)


Insert
Commit log
Memtable
SSTable
{ Bloom filter, Index, Data }

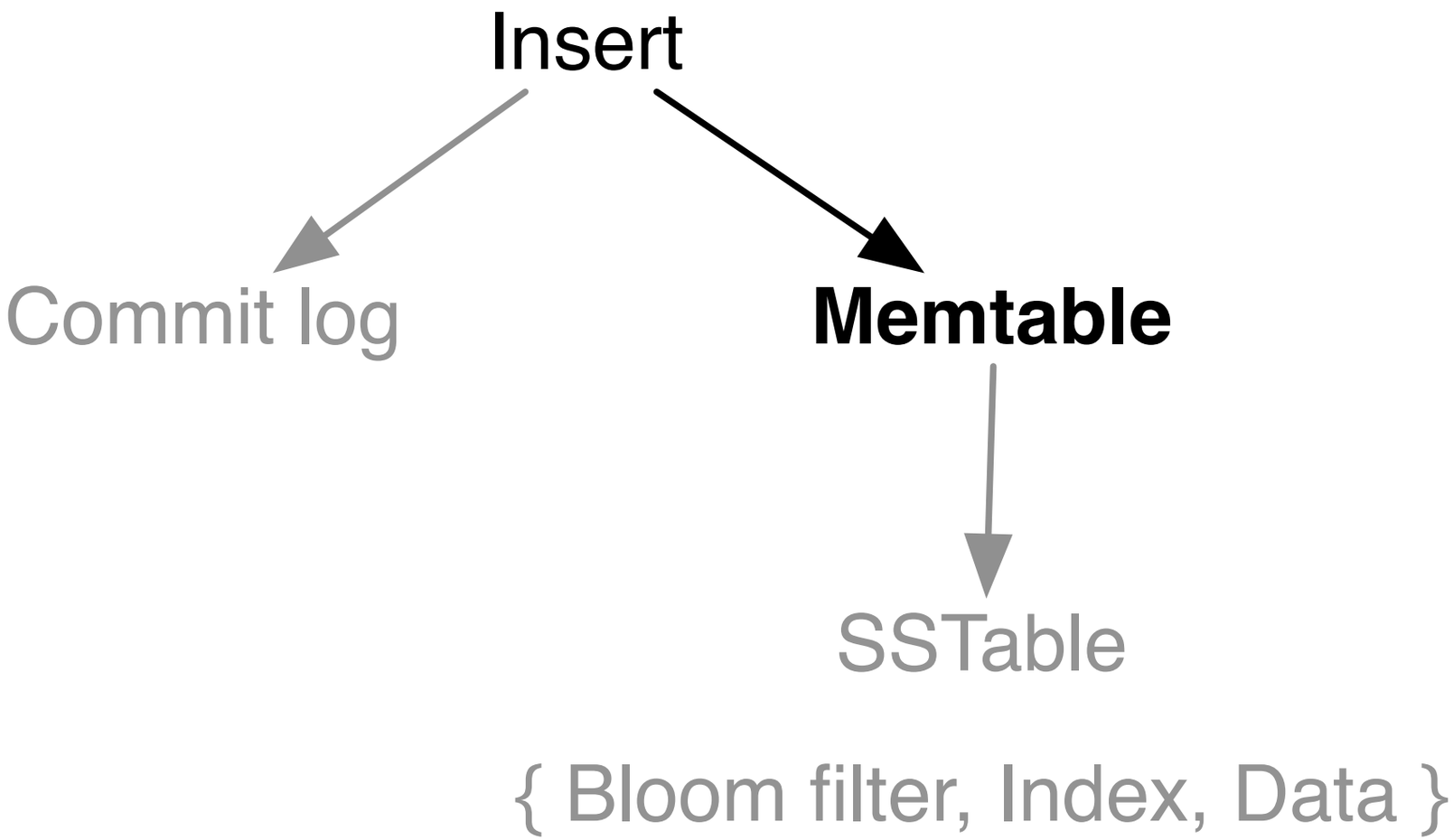
Commit log


Insert
Commit log
Memtable
SSTable
{ Bloom filter, Index, Data }

Commit log

- Each insert written to commit log first
- Stored in insertion order
- Inserts not acknowledged until written to commit log
- Batch vs periodic
- In case of crash, can replay

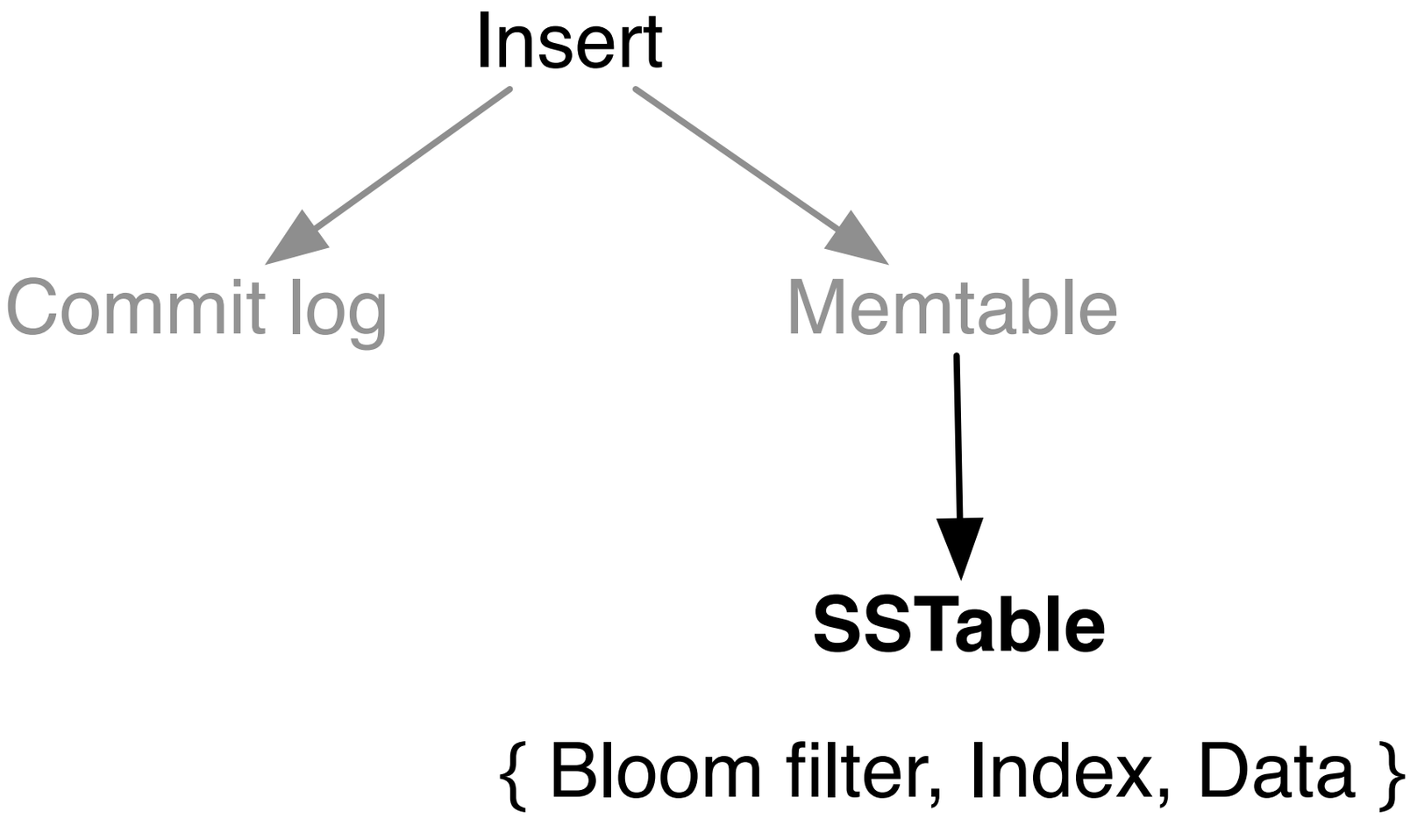
Memtable


Insert
Commit log
Memtable
SSTable
{ Bloom filter, Index, Data }

Memtable

- In memory store of insertions
- ConcurrentSkipListMap
- When too large, flushed to disk
- Ensures all writes to disk are sequential

SSTable


Insert
Commit log
Memtable
SSTable
{ Bloom filter, Index, Data }

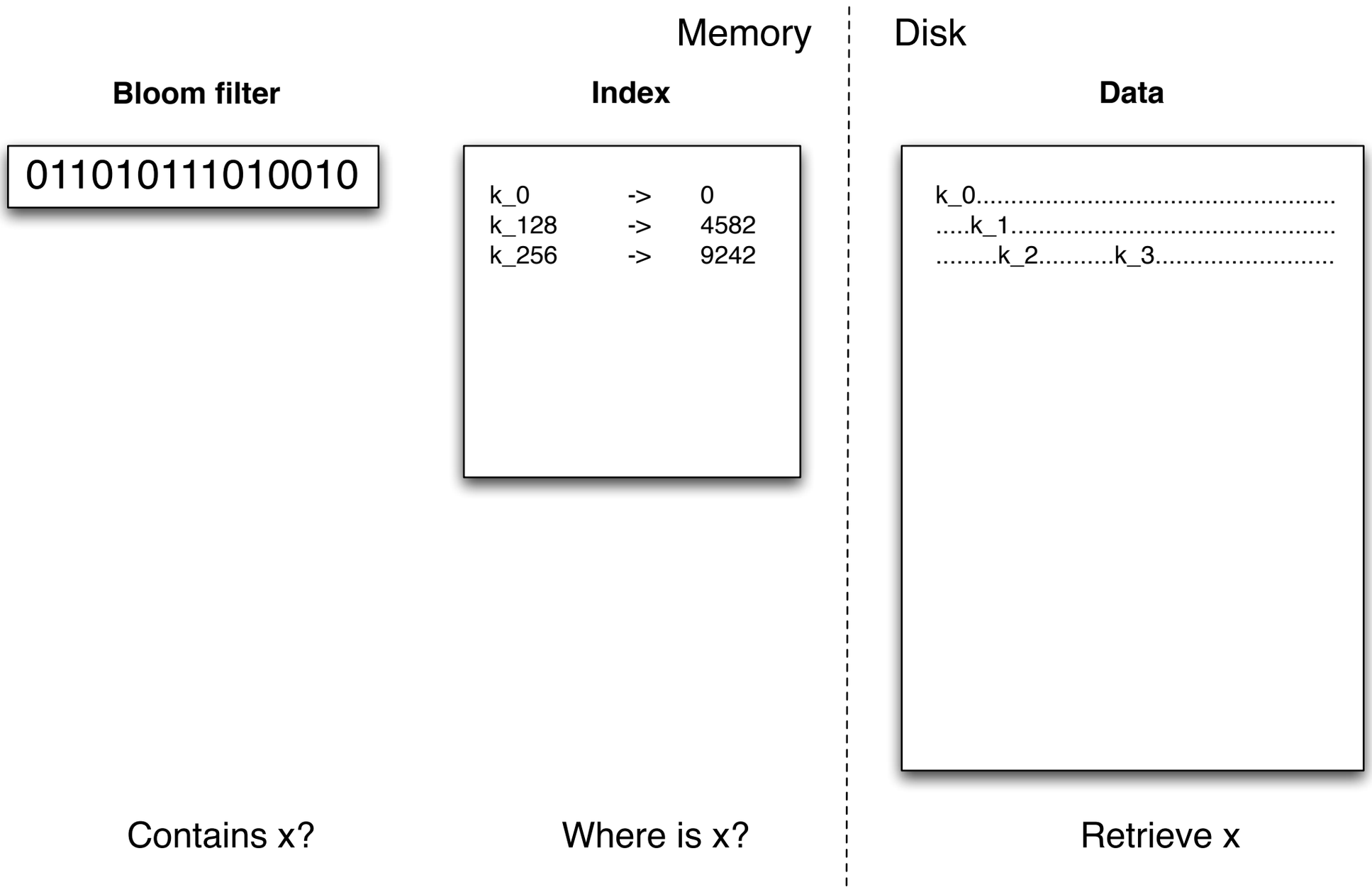
SSTables

- Stores actual data, sorted by key
- Contains a Bloom filter and index to help find keys
- Read only

Bloom filters

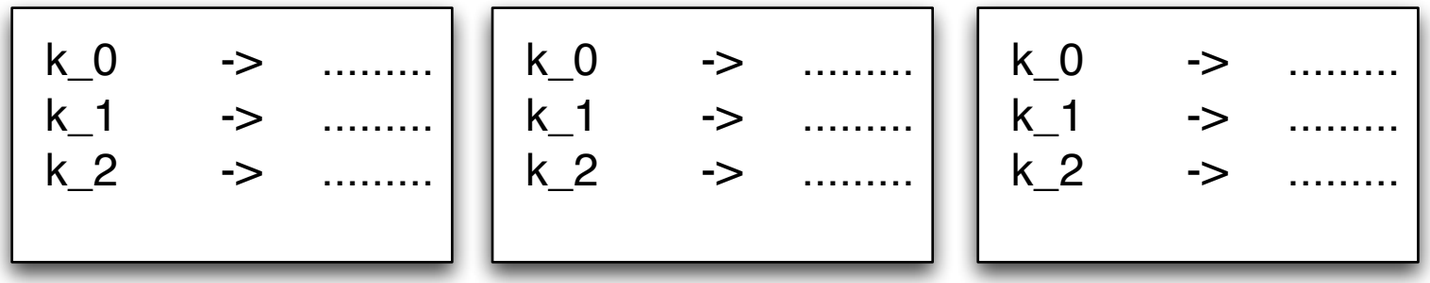
- Probabilistic data structure
- Answers membership queries:
  - *'Does the set contain x?'*
- Can give false positives, never false negatives
- Space efficient
- Typical size: 1 byte per key

How it works together



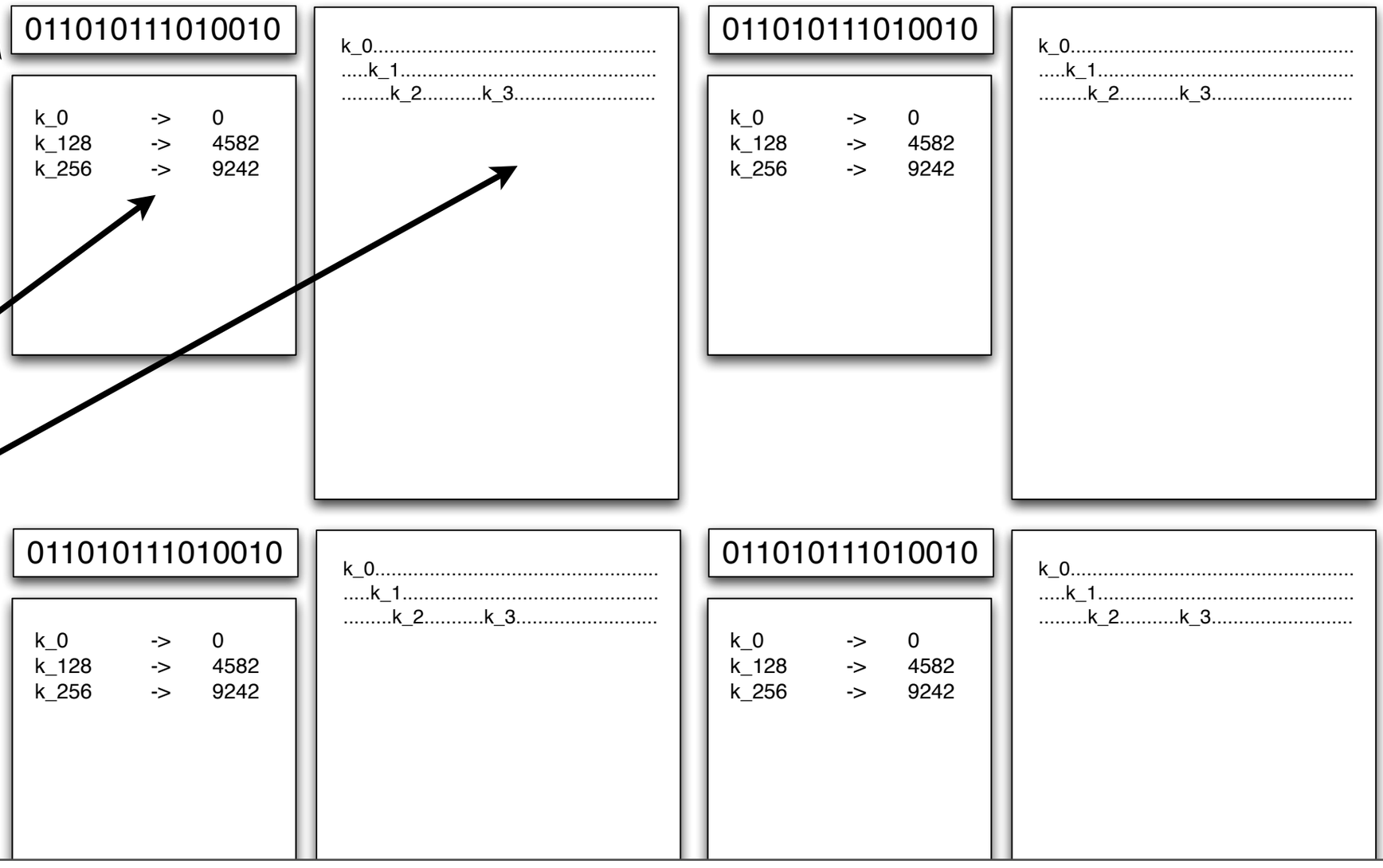
Point queries

Memtables



SSTables

1. Query filter
2. Find location
3. Read data



Range queries

- Bloom filters useless
- Use index to locate portion of SSTable
- Read data, merge results
- Necessary to lookup in every SSTable data file
- Disk I/O proportional to #SSTables

Compaction

- Merges SSTables
- Removes overwrites and obsolete tombstones
- Improves range query performance
- Major compaction creates one SSTable

Write optimised

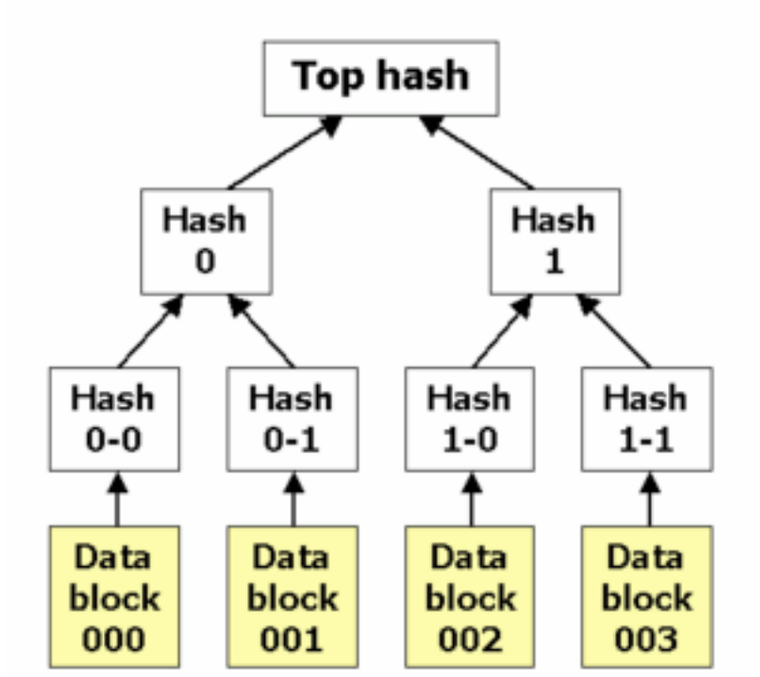
- All writes are sequential on disk
- Each write is written multiple times during compactions
- Bloom filters mean approx. one I/O per read
- Avoid a read-modify-write data model

Scaling

- In memory:
  - Buffers
  - Memtables
  - Bloom filters
  - Index
- If not enough memory, significant performance impact

Repair: Merkle Trees

- Repair builds a Merkle tree
- Compared with replicas
- Efficient
- If differences are found, portions of SSTables are streamed
- Requires full disk scan to build



Snapshot

- For backup, want consistent set of SSTables
- nodetool snapshot does this
- Creates hard links to existing SSTables
- Implies data will be copied after a few compactions

Summary

- How writes end up on disk
- How point queries and range queries find the data
- Implications
- Repair
- Snapshot