## Memory-Based Snapshots (§6.1)

**Apply entry:**
Mutate in-memory data structure

**Service read:**
Look up result in in-memory data structure

**Take snapshot:**
When Raft log size in bytes reaches 4x previous snapshot size:
1. Fork the state machine's memory
   - In parent, continue processing requests
   - In child, serialize in-memory data structure to new snapshot file on disk
2. Discard previous snapshot file on disk
3. Discard Raft log up through child's last applied index

**State to transfer to slow follower:**
Latest snapshot file (immutable)

## Log Cleaning (§6.3)

**Apply entry:**
1. Append entry to in-memory head segment
2. Update index with location of key

**Service read:**
1. Look up location of key in index
2. Read value from segment on disk

**Flush head segment:**
When in-memory log segment reaches 2 MB:
1. Write in-memory segment to disk as new head segment
2. Reset in-memory segment
3. Discard Raft log up through last applied index

**Compact segments:**
When free disk space drops below 5%:
1. Select 10 segments to clean with the largest value of:

$$\frac{benefit}{cost} = \frac{1-u}{1+u} \times segmentAge$$

   where $u$ is the fraction of live bytes in the segment
2. Copy live entries into new segments:
   - Look up key in index to determine if entry is live
   - Update index with new location of key
3. Discard original segments

**State to transfer to slow follower:**
All segments on disk (immutable)

## Raft State for Compaction

Persisted before discarding log entries. Also sent from leader to slow followers when transmitting state.

| | |
|---|---|
| **prevIndex** | index of last discarded entry (initialized to 0 on first boot) |
| **prevTerm** | term of last discarded entry (initialized to 0 on first boot) |
| **prevConfig** | latest cluster membership configuration up through prevIndex |

## Disk-Based Snapshots (§6.2)

**Apply entry:**
1. Mutate on-disk data structure
2. Discard Raft log up through last applied index

**Service read:**
Look up result in on-disk data structure

**State to transfer to slow follower:**
Copy-on-write snapshot of on-disk data structure

## Log-Structured Merge Trees (§6.3)

**Apply entry:**
Add entry to in-memory tree

**Service read:**
1. Search for key in in-memory tree
2. Search all level 0 runs (any might contain the key)
3. For each level counting up from 1 in order, search the single run that might contain the key

**Create new run:**
When in-memory tree reaches 1 MB:
1. Serialize in-memory tree into new sorted level 0 run on disk
2. Reset in-memory tree
3. Discard Raft log up through last applied index

**Compact runs:**
When there are 4 runs at level 0:
1. Merge all level 0 runs with all level 1 runs, producing new non-overlapping level 1 runs split at 2 MB boundaries
2. Discard merged runs

When the total size of all runs at level L exceeds $10^L$ MB:
1. Merge one level L run (chosen round-robin) with all overlapping level L+1 runs, producing new non-overlapping level L+1 runs split at 2 MB boundaries
2. Discard merged runs

**State to transfer to slow follower:**
All runs on disk (immutable)

## Very Small Leader-Based Snapshots (§6.4)

**Apply entry:**
Mutate in-memory data structure

**Service read:**
Look up result in in-memory data structure

**Take snapshot:**
When Raft log size in bytes reaches 1 MB:
1. Stop accepting client requests
2. Wait until last applied index reaches end of log
3. Serialize data structure, append to new snapshot entry in log
4. Resume processing client requests
5. As each server learns the snapshot entry is committed, it discards its Raft log entries up to that entry

**State to transfer to slow follower:**
Raft log (no additional state)