Monkey: Optimal Navigable Key-Value Store

By
Efstathios Karatsiolis
Monkey: **Optimal Navigable Key-Value Store**

- Monkey maps the design of LSM trees and is able to tune critical design knobs.
- Monkey is able to navigate the curve and find the best trade-off for a given application.

The Pareto curve, beyond which lookup cost cannot be improved without harming update cost, and vice versa.
LSM Trees Background
Key Words

✧ **Runs:**
  ◦ A block of data as stored in the LSM Tree

✧ **BloomFilter:**
  ◦ A probabilistic structure that can answer maybe/no. Has a False Positive Rate (FPR), which describes how often the bloom filter was wrong in its “Yes” prediction.

✧ **FencePointer:**
  ◦ Metadata presenting the minimum and the maximum value of a page

✧ **Tier/Level:**
  ◦ A level of the LSM tree. Levels start at 0

✧ **Lookup:**
  ◦ Either a single value lookup or a ranged query

✧ **Update:**
  ◦ Either an insert or update or delete
Overview of an LSM-tree and list of terms used throughout the paper.

<table>
<thead>
<tr>
<th>Level</th>
<th>Main Memory</th>
<th>Secondary Storage</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lookup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>filters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>fence pointers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$M = M_{buffer} + M_{filters} + M_{pointers}$$

$$P \cdot B \cdot T_i = N \cdot \frac{T_i - 1}{T}$$

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Total number of entries</td>
<td>entries</td>
</tr>
<tr>
<td>$L$</td>
<td>Number of levels</td>
<td>levels</td>
</tr>
<tr>
<td>$B$</td>
<td>Number of entries that fit into a disk page</td>
<td>entries</td>
</tr>
<tr>
<td>$E$</td>
<td>Size of an entry</td>
<td>bits</td>
</tr>
<tr>
<td>$P$</td>
<td>Size of the buffer in disk pages</td>
<td>pages</td>
</tr>
<tr>
<td>$T$</td>
<td>Size ratio between adjacent levels</td>
<td></td>
</tr>
<tr>
<td>$T_{lim}$</td>
<td>Size ratio value at which point $L$ converges to 1</td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>Total amount of main memory in system</td>
<td>bits</td>
</tr>
<tr>
<td>$M_{buffer}$</td>
<td>Main memory allocated to the buffer</td>
<td>bits</td>
</tr>
<tr>
<td>$M_{filters}$</td>
<td>Main memory allocated to the Bloom filters</td>
<td>bits</td>
</tr>
<tr>
<td>$M_{pointers}$</td>
<td>Main memory allocated to the fence pointers</td>
<td>bits</td>
</tr>
</tbody>
</table>
LSM Trees Background

Key Concepts

- **Buffering Updates**:
  - A buffer in memory receives all the updates.
  - The buffer is in Level 0.
  - When buffer fills $\rightarrow$ empty contents to disk.
  - Disk levels start at 0.
  - The size of the buffer $M_{buffer} = P \times B \times E$
  - The overall number of levels: $L = \left\lfloor \log_f \left( \frac{N \cdot E}{M_{buffer}} \cdot \frac{T - 1}{T} \right) \right\rfloor$

- **Merge Operations**:
  - To bound the number of runs that a lookup has to probe, an LSM-tree organizes runs among the different levels based on their sizes, and it merges runs of similar sizes.
  - Two merging policies: leveling and tiering
Tier vs Level

Tiered LSM-Tree

Leveled LSM-Tree

Levels and steps:

- Step 1:
  - Level 0: Insert 13
  - Level 1: Flush
  - Level 2: Move & Merge

- Step 2:
  - Level 0:
  - Level 1: Merge & Move
  - Level 2:

- Step 3:
  - Level 0:
  - Level 1:
  - Level 2: Merge
LSM Trees Background

Key Concepts

◊ **Lookups:**
  ◊ A lookup starts from the memory buffer and traverses the other levels. After finding the first matching entry it terminates. Range lookups require merging of runs on different levels.

◊ **Probing a Run:**
  ◊ Fence pointers allow to read almost exactly as many pages as we need

◊ **Bloom Filters:**
  ◊ Each run has a Bloomfilter to facilitate lookups. False positive rate is derived from the number of bits allocated to each entry and the total number of entries

\[
FPR = e^{-\frac{\text{bits}}{\text{entries}} \cdot \ln(2)^2}
\]
## Tier vs Level – Round 2

<table>
<thead>
<tr>
<th>Technique</th>
<th>Point Lookup Cost</th>
<th>Update Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Log</td>
<td>$O \left( \frac{N \cdot E}{M_{buffer}} \cdot e^{\frac{M_{filters}}{N}} \right)$</td>
<td>$O \left( \frac{1}{B} \cdot \log_2 \left( \frac{N \cdot E}{M_{buffer}} \right) \right)$</td>
</tr>
<tr>
<td>(2) Tiering</td>
<td>$O(T \cdot \log_2 \left( \frac{N \cdot E}{M_{buffer}} \right) \cdot e^{\frac{M_{filters}}{N}})$</td>
<td>$O \left( \frac{T}{B} \cdot \log_2 \left( \frac{N \cdot E}{M_{buffer}} \right) \right)$</td>
</tr>
<tr>
<td>(3) Leveling</td>
<td>$O \left( \log_2 \left( \frac{N \cdot E}{M_{buffer}} \right) \cdot e^{\frac{M_{filters}}{N}} \right)$</td>
<td>$O \left( \frac{T}{B} \cdot \log_2 \left( \frac{N \cdot E}{M_{buffer}} \right) \right)$</td>
</tr>
<tr>
<td>(4) Sorted Array</td>
<td>$O \left( e^{\frac{M_{filters}}{N}} \right)$</td>
<td>$O \left( \frac{1}{B} \cdot \frac{N \cdot E}{M_{buffer}} \right)$</td>
</tr>
</tbody>
</table>

**Graph:**
- **Log**: $O \left( \frac{N \cdot E}{M_{buffer}} \cdot e^{\frac{M_{filters}}{N}} \right)$
- **Tiering**: $O \left( \frac{T}{B} \cdot \log_2 \left( \frac{N \cdot E}{M_{buffer}} \right) \right)$
- **Leveling**: $O \left( \frac{1}{B} \cdot \frac{N \cdot E}{M_{buffer}} \right)$
- **Sorted Array**: $O \left( e^{\frac{M_{filters}}{N}} \right)$

**Merge Policy**
- **Tiering**: $T_{lim}$
- **Leveling**: $2$
- **Sorted Array**: $8$

**Size Ratio $T$**
- $2$
- $4$
- $8$

**Update cost (I/Os)**
- $O \left( \frac{1}{B} \cdot \frac{N \cdot E}{M_{buffer}} \right)$

**Lookup cost (I/Os)**
- $O \left( e^{\frac{M_{filters}}{N}} \right)$
Cost Analysis
Tier vs Level – Round 3

Tier Lookup Cost
\[ O(L \cdot T \cdot e^{-\frac{M_{\text{filters}}}{N}}) \text{ I/Os} \]

Tier Update Cost
\[ O\left(\frac{L}{B}\right) \text{ I/Os} \]

Level Lookup Cost
\[ O(L \cdot e^{-\frac{M_{\text{filters}}}{N}}) \text{ I/Os} \]

Level Update Cost
\[ O\left(\frac{T \cdot L}{B}\right) \]
Design space contentions

✧ Contention 1:
  How do we allocate a given amount of main memory of M-filters among the different Bloom filters?
  → Reallocating memory from one filter to another reduces/increases false positive rate, what is the optimal?

✧ Contention 2:
  How to allocate the available main memory between the buffer and the filters?
  → Larger buffer == faster lookup AND smaller update cost
    BUT smaller Bloom Filters == slower lookup

✧ Contention 3:
  How to tune the size ratio and merge policy?
  → Decreasing/increasing size ratio on tiering/leveling improves lookup but degrades update.
    With different workloads, how do we optimize?
That’s where MONKEY comes in
Monkey

- Monkey is an LSM tree based key-value store
- Is able to reach the Pareto curve and find the best possible balance between lookups and updates
- Maximized throughput for uniformly random workloads

**Key features:**
- Design Knobs
- Minimizing Lookup Cost
- Performance Prediction
- Autotuning
Figure 6: Overview of how Monkey allocates false positive rates $p_1, p_2 ... p_L$ to Bloom filters in proportion to the number of key-value pairs in a given level, and a further list of terms used to describe Monkey.
Minimizing Lookup Cost

Average worst case lookup cost:

\[ R = \begin{cases} \frac{\sum_{i=1}^{L} p_i}{(T-1) \cdot \sum_{i=1}^{L} p_i}, & \text{with leveling} \\ \sum_{i=1}^{L} p_i, & \text{with tiering} \end{cases} \]

where 0 < p_i \leq 1

Main Memory Footprint

\[ M_{\text{filters}} = \frac{N}{\ln(2)} \cdot \frac{T - 1}{T} \cdot \sum_{i=1}^{L} \frac{\ln(p_i)}{T^{L-i}} \]

Leveling

\[ p_i = \begin{cases} 1, & \text{if } i > L_{\text{filtered}} \\ \frac{(R-L_{\text{unfiltered}})(T-1)}{T_{\text{filtered}}^{T-1-i}}, & \text{else} \end{cases} \]

for 0 < R \leq L

and 1 \leq i \leq L

and \( L_{\text{filtered}} = L - \max(0, \lfloor R - 1 \rfloor) \)  

(5)

Tiering

\[ p_i = \begin{cases} 1, & \text{if } i > L_{\text{filtered}} \\ \frac{(R-L_{\text{unfiltered}})(T-1)}{T_{\text{filtered}}^{T-1-i}}, & \text{else} \end{cases} \]

for 0 < R \leq L \cdot (T - 1)

and 1 \leq i \leq L

and \( L_{\text{filtered}} = L - \max(0, \left\lfloor \frac{R - 1}{T} \right\rfloor) \)  

(6)
Bloomfilters in Monkey

- Monkey does not implement BloomFilters to all the levels. Why?
  - Because of the FPR
  - Because of the fence pointer, probe cost is the same no matter the level
  - Bloomfilters are T times bigger in every level compared to the i-1 level
  - Maintaining low FPR requires much more space

- Monkey finds optimal number of levels to have a BloomFilter
- Monkey sets the FPR proportional to their size
Bloomfilters everywhere

Bloomfilters where it is useful
Predicting Lookup and Update Cost

- Modeling zero result lookup cost ($R$)

\[
R = R_{\text{filtered}} + R_{\text{unfiltered}}
\]

\[
R_{\text{filtered}} = \begin{cases} 
\frac{T}{T-1} \cdot e^{-\frac{M_{\text{filters}}}{N}} \cdot \ln(2)^2 \cdot T^{L_{\text{unfiltered}}} & \text{with leveling} \\
\frac{T}{T-1} \cdot e^{-\frac{M_{\text{filters}}}{N}} \cdot \ln(2)^2 \cdot T^{L_{\text{unfiltered}}} & \text{with tiering}
\end{cases}
\]

\[
R_{\text{unfiltered}} = \begin{cases} 
L_{\text{unfiltered}}, & \text{with leveling} \\
L_{\text{unfiltered}} \cdot (T - 1), & \text{with tiering}
\end{cases}
\]

(7)
Predicting Lookup and Update Cost

- Number of deeper levels that have no filter

\[ L_{\text{unfiltered}} = \begin{cases} 
0, & \text{if } M_{\text{threshold}} \leq M_{\text{filters}} \\
\left\lceil \log_T \left( \frac{M_{\text{threshold}}}{M_{\text{filters}}} \right) \right\rceil, & \frac{M_{\text{threshold}}}{TL} \leq M_{\text{filters}} \leq M_{\text{threshold}} \\
L, & 0 \leq M_{\text{filters}} \leq \frac{M_{\text{threshold}}}{TL} 
\end{cases} \]

\[ M_{\text{threshold}} = \frac{N}{\ln(2)^2} \cdot \frac{\ln(T)}{(T - 1)} \]
Modeling Worst-Case Non-Zero-Result Lookup Cost ($V$)

\[ V = R - p_L + 1 \]

Modeling Worst-Case Update Cost ($W$)

\[
W = \begin{cases} 
\frac{L}{B} \cdot \frac{(T-1)}{2} \cdot (1 + \phi), & \text{with leveling} \\
\frac{L}{B} \cdot \frac{(T-1)}{T} \cdot (1 + \phi), & \text{with tiering}
\end{cases} \quad (10)
\]
Modeling Worst-Case Range Lookup Cost \( (Q) \).

\[
Q = \begin{cases} 
  s \cdot \frac{N}{B} + L, & \text{with leveling} \\
  s \cdot \frac{N}{B} + L \cdot (T - 1), & \text{with tiering}
\end{cases}
\]

(11)
### Time to compare

<table>
<thead>
<tr>
<th>Merge policy</th>
<th>Update Cost (W)</th>
<th>State of the Art Lookup Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( O\left(\frac{1}{B}\right) )</td>
<td>( O\left(\frac{N-E}{M_{buffer}}\right) )</td>
</tr>
<tr>
<td>(1) Tiering ((T = T_{lim}))</td>
<td>( O\left(\frac{1}{B} \cdot \log_T\left(\frac{N-E}{M_{buffer}}\right)\right) )</td>
<td>( O\left(T \cdot \log_T\left(\frac{N-E}{M_{buffer}}\right)\right) )</td>
</tr>
<tr>
<td>(2) Tiering ((2 \leq T &lt; T_{lim}))</td>
<td>( O\left(\frac{T}{B} \cdot \log_T\left(\frac{N-E}{M_{buffer}}\right)\right) )</td>
<td>( O\left(\log_T\left(\frac{N-E}{M_{buffer}}\right)\right) )</td>
</tr>
<tr>
<td>(3) Leveling ((2 \leq T &lt; T_{lim}))</td>
<td>( O\left(\frac{1}{B} \cdot \frac{N-E}{M_{buffer}}\right) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>(4) Leveling ((T = T_{lim}))</td>
<td>( O\left(\frac{N-E}{M_{buffer}} \cdot e^{\frac{M_{filters}}{N}}\right) )</td>
<td>( O\left(T \cdot \log_T\left(\frac{N-E}{M_{buffer}} \cdot e^{\frac{M_{filters}}{N}}\right)\right) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Merge policy</th>
<th>Monkey Lookup Cost (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{M_{threshold}}{T_L} \leq M_{filters} \leq M_{threshold} )</td>
<td>( M_{threshold} \leq M_{filters} )</td>
</tr>
<tr>
<td>( (d) )</td>
<td>( (e) )</td>
</tr>
<tr>
<td>(1) Tiering ((T = T_{lim}))</td>
<td>( O\left(\frac{N-E}{M_{buffer}}\right) )</td>
</tr>
<tr>
<td>(2) Tiering ((2 \leq T &lt; T_{lim}))</td>
<td>( O(T \cdot \log_T\left(\frac{N}{M_{filters}}\right)) )</td>
</tr>
<tr>
<td>(3) Leveling ((2 \leq T &lt; T_{lim}))</td>
<td>( O(T \cdot \log_T\left(\frac{N}{M_{filters}}\right)) )</td>
</tr>
<tr>
<td>(4) Leveling ((T = T_{lim}))</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>
Time to compare

Figure 7: Monkey dominates the state of the art in terms of lookup cost $R$ for all values of $M_{\text{filters}}$.

Figure 8: Monkey dominates the state of the art for any merge policy and size ratio.
Scalability and Tunability

- Scaling Bloom filters’ footprint with the number of data, lookup cost remains fixed whereas in the state of the art it increases at a logarithmic rate.

- Lookup cost is independent of the entry size, and so it does not increase for data sets with larger entry sizes.

- Lookup cost is independent of the buffer size. This simplifies tuning relative to the state of the art → no need to carefully balance main memory allocation between the buffer and filters to optimize lookup performance.
Scalability and Tunability

Leveling: $X = 1$
Tiering: $X = T$

State of the art
Monkey

Unfiltered LSM-tree
Sweet spot

Update cost $W$ ($1/O_s$)
Lookup cost $R$ ($1/O_s$)

$O(X \cdot \log_T(N \cdot \frac{E}{M})$)
$O(X \cdot e^{-\frac{M}{N}})$

$O(\frac{T}{B\cdot X} \cdot \log_T(N \cdot \frac{E}{M})$)
$O(0 \cdot \log_T(N \cdot \frac{E}{M})$)

Buffer size $M_{buffer}$ (log scale)

$\frac{M}{4}$ $\frac{M}{2}$ $M$
**Tuning the Size Ratio and Merge Policy**

- The merge policy and size ratio are complementary means of navigating the same tradeoff continuum.

- Monkey a divide and conquer algorithm that linearizes this continuum into a single dimension and finds the tuning that maximizes throughput.
Tuning Main Memory Allocation

- Three-step strategy
  - Allocate first $M$ bytes to buffer, since there are levels where bloomfilters might have no use.
  - Allocate 5% to the buffer and the rest 95% to the bloomfilters
  - Continue following step two until the I/O overhead due to false positives becomes negligible
  - If there is more main memory in our budget beyond step two → allocate it to the buffer to further reduce update cost.
Experiments

◊ Monkey Scales Better with Data Volume
  ◊ Figure 11 (A) shows how lookup latency corresponds to disk I/Os. The average number of I/Os per lookup is lower than one because the lookups target non-existing keys, and so they do not issue I/Os most of the time due to the filters.
  ◊ Figure 11 (B) depicts results for a similar experiment, with the difference that this time we keep the number of data entries fixed and we instead increase the entry size.

◊ Monkey Needs Less Main Memory
  ◊ Repeat the default experimental setup multiple times, each time using a different number of bits-per-entry ratio allocated to the filters. The results are shown in Figure 11 (C).

◊ Monkey Improves Lookup Cost for Different Workloads
  ◊ Experiment shows that Monkey significantly improves lookup latency for non-zero-result lookups across a wide range of temporal locality in the query workload. The results are shown in Figure 11 (D).
Figure 11: Monkey improves lookup cost under any (A) number of entries, (B) entry size, (C) amount of memory, (D) lookup locality, and (F) merge policy and size ratio. It navigates the design space to find the design that maximizes throughput (F).
Experiments

◊ Monkey Reaches the Pareto Curve
   ◇ Repeat the experimental setup multiple times, each time using a different configuration of size ratio and merge policy. Measure the average latencies of lookups and updates and plot them against each other for Monkey and LevelDB. The result is shown in Figure 11 (E).

◊ Monkey Navigates the Design Space to Maximize Throughput.
   ◇ Repeat the default experimental setup multiple times, with the difference that during the query processing phase we vary the ratio of zero-result lookups to updates from 10% to 90%. Results are shown in Figure 11 (F).
Discussion
Thank you