CrimsonDB

A Self-Designing Key-Value Store

Niv Dayan, Wilson Qin, Manos Athanassoulis, Stratos Idreos

http://daslab.seas.harvard.edu/crimsondb/
storage is cheaper

\[
\text{price per GB} \downarrow \quad \text{time} \uparrow
\]

\[
\text{inserts & updates} \quad \frac{\text{workload}}{\text{workload}}
\]
Price per GB decreases over time, making storage cheaper. Inserts and updates to workload increase as well. Therefore, there is a need for write-optimized database structures.
need for write-optimized database structures

LSM-tree invented

1996 now

time
need for write-optimized database structures

LSM-tree invented

1996

time

Key-Value Stores

now
Key-Value Stores
Suboptimal Design

workload → performance

hardware → performance
Map Trade-Offs

Navigate

What-if?

http://daslab.seas.harvard.edu/crimsondb/
LSM-tree
Key-Value Stores

What are they really?
updates → buffer

memory | storage
updates → buffer → memory → storage → sort & flush runs
Updates $\rightarrow$ Buffer $\rightarrow$ Memory $\rightarrow$ Storage

- Sort & Flush
- Runs
- Sort-Merge
memory \quad storage

buffer

exponentially increasing capacities

$O(\log(N))$ levels
lookup key $X$

buffer

memory

storage

fence pointers

one I/O per run

$X$
lookup key $X$

buffer

memory

storage

fence

pointers

one I/O per run

$X$
lookup key $X$

buffer

Bloom filters

memory

fence pointers

storage

$X$
lookup key $X$

buffer

Bloom filters

true negative

fence pointers

memory

storage

$X$
lookup key $X$

buffer

Bloom filters

true negative
false positive

memory

fence

pointers

storage

false positive
true negative

Bloom filters

$X$
Performance & Cost Tradeoffs

lookup key $X$

buffer

Bloom filters

memory

fence pointers

true negative

false positive

ture positive

storage

false positive

true negative

true positive

$X$
Performance & Cost Tradeoffs

lookup key \(X\) → buffer

Bloom filters

- true negative
- false positive
- true positive

memory

fence pointers

storage

bigger filters → fewer false positives
Performance & Cost Tradeoffs

lookup key $X$

- buffer
- Bloom filters
- true negative
- false positive
- true positive

memory

- fence pointers
- true
- false positive
- true positive

storage

bigger filters $\rightarrow$ fewer false positives

memory vs. lookups
Performance & Cost Tradeoffs

lookup key $X$ → buffer → Bloom filters → fence pointers → memory → storage

memory vs. lookups

- bigger filters → fewer false positives
- more merging → fewer runs

false positive → true positive
true negative → true negative

false positive
true positive
true negative

false positive
true positive
true negative
Performance & Cost Tradeoffs

lookup key $X$

buffer

Bloom filters
true negative
false positive
true positive

memory

fence pointers

storage

more merging $\rightarrow$ fewer runs

lookups vs. updates

bigger filters $\rightarrow$ fewer false positives

memory vs. lookups
main memory

lookup cost

update cost
lookup cost

update cost

main memory

lookup cost

fixed memory

existing systems

update cost
lookup cost

update cost

main memory

lookup cost

fixed memory

merge less

existing systems

merge more

update cost
Problem 1:

Problem 2:

update cost

lookup cost

existing systems
Problem 1: **suboptimal design**

Problem 2:
Problem 1: **suboptimal design**

Problem 2:
Problem 1: suboptimal design

Problem 2: **hard to tune**
Problem 1: suboptimal design
Problem 2: **hard to tune**
Problem 1: suboptimal design
Problem 2: hard to tune

merge policy greed
lookups vs. updates

max
throughput

lookup cost

update cost
Monkey: **Optimal Navigable Key-Value Store**

Niv Dayan, Manos Athanasollis, Stratos Idreos

SIGMOD 2017

“Best of SIGMOD”
<table>
<thead>
<tr>
<th>Monkey:</th>
<th>Optimal</th>
<th>Navigable</th>
<th>Key-Value Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>observations:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insights:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>steps:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monkey:</td>
<td>Optimal</td>
<td>Navigable</td>
<td>Key-Value Store</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------</td>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>observations:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fixed false positive rates</td>
<td>filters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insights:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lookup cost = ( \sum p_i )</td>
<td>suboptimal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>steps:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>optimize allocation</td>
<td>asymptotically better</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>memory vs. lookups</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monkey:</td>
<td><strong>Optimal</strong></td>
<td><strong>Navigable</strong></td>
<td><strong>Key-Value Store</strong></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td><strong>observations:</strong></td>
<td>filters</td>
<td>LSM-tree</td>
<td>merge policy</td>
</tr>
<tr>
<td></td>
<td>fixed false positive rates</td>
<td></td>
<td>↓? performance</td>
</tr>
<tr>
<td><strong>insights:</strong></td>
<td></td>
<td>log</td>
<td>sorted array</td>
</tr>
<tr>
<td></td>
<td>lookup cost = $\sum p_i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>suboptimal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>steps:</strong></td>
<td>optimize allocation asymptotically better memory vs. lookups</td>
<td>updates vs. lookups navigate</td>
<td></td>
</tr>
<tr>
<td>Monkey:</td>
<td>Optimal</td>
<td>Navigable</td>
<td>Key-Value Store</td>
</tr>
<tr>
<td>---------</td>
<td>--------------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td><strong>observations:</strong></td>
<td>filters</td>
<td>LSM-tree</td>
<td>ad-hoc trade-offs</td>
</tr>
<tr>
<td>fixed false positive rates</td>
<td>merge policy</td>
<td>navigation</td>
<td>memory</td>
</tr>
<tr>
<td><strong>insights:</strong></td>
<td>log sorted array</td>
<td></td>
<td>lookups updates</td>
</tr>
<tr>
<td>lookup cost = ( \sum p_i )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>suboptimal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>steps:</strong></td>
<td>optimize allocation</td>
<td>updates vs. lookups</td>
<td>answer what-if design questions</td>
</tr>
<tr>
<td>asymptotically better</td>
<td>navigate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
update cost

lookup cost

for fixed memory

WiredTiger

Cassandra, HBase

RocksDB, LevelDB

Monkey

Pareto frontier

update cost
For fixed memory, the Pareto frontier is shown for different storage systems. The diagram plots lookup cost on the y-axis and update cost on the x-axis. The maximum throughput is marked as 'max throughput' with an arrow pointing towards the Monkey storage system. Other systems include WiredTiger, Cassandra, HBase, RocksDB, and LevelDB.
**Monkey:** Optimal | Navigable | Key-Value Store
---|---|---
**Observations:**
- Fixed false positive rates
- LSM-tree merge policy
- Ad-hoc trade-offs

**Insights:**
- Lookup cost $= \sum p_i$
- Sorted array
- Memory

**Steps:**
- Optimize allocation
- Updates vs. lookups
- Answer what-if design questions
- Asymptotically better memory vs. lookups
### Monkey:

<table>
<thead>
<tr>
<th><strong>Optimal</strong></th>
<th><strong>Navigable</strong></th>
<th><strong>Key-Value Store</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>filters</td>
<td>LSM-tree</td>
<td>ad-hoc trade-offs</td>
</tr>
<tr>
<td>fixed false positive rates</td>
<td>merge policy</td>
<td>memory</td>
</tr>
<tr>
<td>suboptimal</td>
<td>log</td>
<td>lookups updates</td>
</tr>
<tr>
<td>lookup cost = $\sum p_i$</td>
<td>sorted array</td>
<td></td>
</tr>
<tr>
<td>optimize allocation</td>
<td>updates vs. lookups</td>
<td>answer what-if design questions</td>
</tr>
</tbody>
</table>

**Observations:**
- fixed false positive rates

**Insights:**
- lookup cost = $\sum p_i$
- suboptimal

**Steps:**
- optimize allocation
- asymptotically better memory vs. lookups

**Ad-hoc trade-offs:**
- memory
- lookups updates
buffer

Bloom filters
data

fence pointers
Bloom filters

$X$ bits per entry

memory

storage

data
Bloom filters

X bits per entry

memory

storage

data
Bloom filters

memory

$X$ bits per entry

Bloom filters

\[
false \text{ positive rate } p = e^{\frac{-\text{bits } M}{\text{entries } N} \ln(2)^2}
\]
Bloom filters

memory

$X$ bits per entry

false positive rate $p = e^{-\frac{\text{bits } M}{\text{entries } N} \ln(2)^2}$
worst-case I/O overhead:

\[ p \text{ bits per entry} \]

\[ X \text{ bits per entry} \]

\[ \text{Bloom filters} \]

\[ p \]

\[ p \]

\[ p \]

\[ \frac{\text{bits } M}{\text{entries } N} \ln(2)^2 \]

\[ = e \]

false positive rate \( p \)
worst-case I/O overhead:

\[ O(\sum p) \]

\[ \text{false positive rate } p = e^{- \frac{\text{bits } M}{\text{entries } N} \ln(2)^2} \]
worst-case I/O overhead:

\[ O(\sum p) \]

false positive rate \( p = e^{-\frac{\text{bits } M}{\text{entries } N} \ln(2)^2} \)
worst-case I/O overhead:

\[ O\left( \sum e^{-M/N} \right) \]

false positive rate \( p = e \)
Bloom filters

memory

$X$ bits per entry

Bloom filters

$p$

$p$

$p$

$p$

worst-case I/O overhead:

$O(\sum e^{-M/N})$
worst-case I/O overhead:

\[ O( \log(N) \cdot e^{-M/N} ) \]
Can we do better?

worst-case I/O overhead:

\[ O( \log(N) \cdot e^{-M/N} ) \]
lookup
key X

Bloom filters

fence pointers

data runs

…
lookup key X

Bloom filters

false positive
false positive
false positive
false positive
false positive

most memory

fence pointers

false positive
false positive
false positive
false positive
false positive

data runs

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

I/O
I/O
I/O
I/O
I/O

...
most memory saves at most 1 I/O
Bloom filters

false positive rates

reallocate some

most memory
same memory, fewer lookup I/Os

false positive rates

reallocate some

most memory
relax

false positive rates

$p_0$

$p_1$

$p_2$
relax

false positive rates

\[ 0 < p_2 < 1 \]
\[ 0 < p_1 < 1 \]
\[ 0 < p_0 < 1 \]
false positive rates

\[0 < p_2 < 1\]
\[0 < p_1 < 1\]
\[0 < p_0 < 1\]

relax

model

lookup cost \(= f(p_0, p_1 \ldots)\)

memory footprint \(= f(p_0, p_1 \ldots)\)
false positive rates

\[ 0 < p_2 < 1 \]
\[ 0 < p_1 < 1 \]
\[ 0 < p_0 < 1 \]

model

lookup cost = f(p_0, p_1 ...)

memory footprint = f(p_0, p_1 ...)

optimize

in terms of \( p_0, p_1 \)
Bloom filters

false positive rates

lookup cost $= \sum p_i$
Bloom filters

false positive rates

\[ p_2 \]
\[ p_1 \]
\[ p_0 \]

lookup cost \( = \sum p_i \)

memory footprint

\[ \text{false positive rate} = e^{\frac{\text{bits}}{\text{entries}}} \ln(2)^2 \]
Bloom filters

\[ : \]

\[
p_2
\]

false positive rates

\[
p_1
\]

\[
p_0
\]

lookup cost \( = \sum p_i \)

memory footprint

\[
\text{bits} \:= \:- \frac{\ln\left( \frac{\text{false positive rate}}{\ln(2)^2} \right)}{\text{entries}}
\]
Bloom filters

false positive rates

lookup cost  =  \sum p_i

memory footprint

bits(p_0, N)

bits(p_1, N/T)

bits(p_2, N/T^2)
Bloom filters

\[ \begin{align*}
&: \\
&p_0 \\
&p_1 \\
&p_2
\end{align*} \]

false positive rates

lookup cost  \( = \sum p_i \)

memory footprint

\[ \begin{align*}
&: \\
&\text{bits}(p_2, N/T^2) \\
&\text{bits}(p_1, N/T) \\
&\text{bits}(p_0, N)
\end{align*} \]

false positive rates

memory  \( = -c \cdot N \cdot \sum \frac{\ln(p_i)}{T^i} \)

size ratio

constant

entries

lookup cost  \( = \sum p_i \)
false positive rates

Bloom filters

\[ p_0 \quad p_1 \quad p_2 \quad \ldots \]

\[ \text{lookup cost} = \sum p_i \]

\[ \text{memory} = -c \cdot N \cdot \sum \frac{\ln(p_i)}{T_i} \]

optimize
false positive rates

\[ \frac{p_0}{T^2} \]

\[ \frac{p_0}{T} \]

\[ p_0 \]

exponential decrease
false positive rates

\[ p_0/T^2 \]
\[ p_0/T \]
\[ p_0 \]

exponential decrease

State-of-the-Art Bloom filters

\[ p \]
\[ p \]
\[ p \]

same
Monkey Bloom filters

false positive rates

lookup cost $= \sum p_i$  
$<  \sum p$

State-of-the-Art Bloom filters

$p_0/T^2 < p$

$p_0/T < p$

$p_0 > p$
State-of-the-Art Bloom filters

... < ...

\[ \frac{p_0}{T^2} < p \]
\[ \frac{p_0}{T} < p \]
\[ p_0 > p \]

lookup cost

\[ = \sum p_i < \sum p \]
\[ = O(e^{-M/N}) \]
\[ = O(\log(N) \cdot e^{-M/N}) \]

\( N \) | number of entries
\( M \) | overall memory for Bloom filters
State-of-the-Art
Bloom filters

... < ...

false positive rates

\[ \frac{p_0}{T^2} < p \]

\[ \frac{p_0}{T} < p \]

\[ p_0 > p \]

lookup cost

\[ = \sum p_i < = \sum p \]

\[ = O\left( e^{-M/N} \right) = O\left( \log(N) \cdot e^{-M/N} \right) \]

asymptotic win

lookup cost increases at slower rate as data grows
false positive rates

\[ \frac{p_0}{T^2} \]

\[ \frac{p_0}{T} \]

\[ p_0 \]

convergent geometric series

\[ p_0/T^2 \]

\[ p_0/T \]

\[ p_0 \]
Bloom filters

- $p_0/T^2$
- $p_0/T$
- $p_0$

false positive rates

\[
\text{memory} = c \cdot \text{entries} \cdot \sum \frac{-\ln(p_i)}{T^i}
\]
false positive rates

\[ p_0/T^2 \]
\[ p_0/T \]
\[ p_0 \]

\[ \text{memory} = c \cdot \text{entries} \cdot -\ln(\text{lookup cost}) \]
false positive rates

\[
p_0 / T^2 \quad \frac{p_0}{T} \quad p_0
\]

memory = \( c \cdot entries \cdot -\ln(\text{lookup cost}) \)

model lookups vs. memory trade-off
Problem 1: suboptimal filters allocation
Problem 2: hard to tune
Problem 1: **suboptimal filters allocation**

Problem 2: hard to tune
Problem 1: suboptimal filters allocation
Problem 2: **hard to tune**
Problem 1: suboptimal filters allocation
Problem 2: hard to tune
<table>
<thead>
<tr>
<th>Monkey:</th>
<th>Optimal</th>
<th>Navigable</th>
<th>Key-Value Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>observations:</td>
<td>filters</td>
<td>LSM-tree</td>
<td>ad-hoc trade-offs</td>
</tr>
<tr>
<td>fixed false positive rates</td>
<td>merge policy</td>
<td>memory</td>
<td></td>
</tr>
<tr>
<td>lookup cost = $\sum p_i$</td>
<td>log</td>
<td>lookups</td>
<td></td>
</tr>
<tr>
<td>suboptimal</td>
<td>sorted array</td>
<td>updates</td>
<td></td>
</tr>
<tr>
<td>insights:</td>
<td>optimize allocation</td>
<td>updates vs. lookups</td>
<td></td>
</tr>
<tr>
<td>asymptotically better</td>
<td>navigate</td>
<td>answer what-if design questions</td>
<td></td>
</tr>
<tr>
<td>memory vs. lookups</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monkey:</td>
<td>Optimal</td>
<td>Navigable</td>
<td>Key-Value Store</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>observations:</td>
<td>filters</td>
<td>LSM-tree</td>
<td>ad-hoc trade-offs</td>
</tr>
<tr>
<td></td>
<td>fixed false positive rates</td>
<td>merge policy</td>
<td>memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>↓ ?</td>
<td>lookups</td>
</tr>
<tr>
<td></td>
<td></td>
<td>performance</td>
<td>updates</td>
</tr>
<tr>
<td>insights:</td>
<td>lookup cost = Σ pi</td>
<td>log sorted array</td>
<td>lookup cost</td>
</tr>
<tr>
<td></td>
<td>suboptimal</td>
<td></td>
<td>existing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Monkey</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>update cost</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>steps:</td>
<td>optimize allocation</td>
<td>updates vs. lookups</td>
<td>answer what-if design questions</td>
</tr>
<tr>
<td></td>
<td>asymptotically better</td>
<td>navigate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>memory vs. lookups</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Identify

merge policy

size ratio
Identify

merge policy

size ratio

Map

lookups

updates
Identify

merge policy

Map

LSM-tree

log

updates

sorted array
Identify

- merge policy

- size ratio

Map

- log
- sorted array
- lookups
- updates

Navigate

- workload
- hardware
- maximum throughput
Merge Policies

**Tiering**
write-optimized

**Leveling**
read-optimized
Tiering
write-optimized

Leveling
read-optimized
Tiering
write-optimized

Leveling
read-optimized

$T$ runs per level
Tiering
write-optimized

$T$ runs per level

merge & flush

Leveling
read-optimized
Tiering
write-optimized

$T$ runs per level

Leveling
read-optimized
Tiering
write-optimized

$T$ runs per level

Leveling
read-optimized

merge
Tiering
write-optimized

$T$ runs per level

Leveling
read-optimized

$T$ times bigger

flush
Tiering
write-optimized

$T$ runs per level

Leveling
read-optimized

$T$ times bigger
Tiering
write-optimized

Leveling
read-optimized

$T$ runs per level

1 run per level
Tiering
write-optimized

Leveling
read-optimized

lookup cost:

\[ \mathcal{O}(T \cdot \log_\tau(N) \cdot e^{-M/N}) \]

- \( T \) runs per level
- levels
- false positive rate

\[ \mathcal{O}(\log_\tau(N) \cdot e^{-M/N}) \]

- levels
- false positive rate
Tiering
write-optimized

runs per level
$T$

levels

false positive rate

lookup cost:
$O(T \cdot \log \pi(N) \cdot e^{-M/N})$

Leveling
read-optimized

1 run per level

levels

false positive rate

lookup cost:
$O(\log \pi(N) \cdot e^{-M/N})$
Tiering
write-optimized

$O(T \cdot e^{-M/N})$

runs per level
false positive rate

Leveling
read-optimized

$O(e^{-M/N})$

false positive rate
Tiering
write-optimized

merges per level

lookup cost:
$O(T \cdot \log (N))$

update cost:
$O(T \cdot \log (N))$

levels

Leveling
read-optimized

1 run per level

lookup cost:
$O(e^{-M/N})$

update cost:
$O(T \cdot e^{-M/N})$

merges per level

levels
Tiering
write-optimized

Leveling
read-optimized

lookup cost:
$O(T \cdot e^{-M/N})$

update cost:
$O(\log T(N))$

size ratio $T \geq$
Tiering
write-optimized

1 run per level

lookup cost:

update cost:

Leveling
read-optimized

1 run per level

\( O(e^{-M/N}) = O(e^{-M/N}) \)

\( O(\log(N)) = O(\log(N)) \)

size ratio \( T \downarrow \)
Tiering
write-optimized

Leveling
read-optimized

lookup cost:

$O(T \cdot e^{-M/N})$

$O(e^{-M/N})$

update cost:

$O(\log T(N))$

$O(T \cdot \log T(N))$

size ratio $T \uparrow$
Tiering
write-optimized

$O(N)$ runs per level

lookup cost: $O(N \cdot e^{-M/N})$
update cost: $O(1)$

Leveling
read-optimized

1 run per level

size ratio $T$ $\uparrow$

lookup cost: $O(e^{-M/N})$
update cost: $O(N)$
Tiering
write-optimized

O( N) runs per level

log

lookup cost:
O(N \cdot e^{-M/N})

update cost:
O(1)

Leveling
read-optimized

1 run per level

sorted array

size ratio \( T \uparrow \)

\( O(1) \)

\( O(e^{-M/N}) \)

\( O(N) \)
lookup cost

Tiering

Leveling

update cost

sorted array

\( T \mid \text{size ratio} \)
lookup cost

log

Tiering

log | LSM-tree | sorted array

T | size ratio
lookup cost

update cost

log

Tiering

T=2

Leveling

sorted array

T | size ratio

workload

hardware

maximum throughout
Problem 1: suboptimal filters allocation

Problem 2: **hard to tune**

Diagram:
- Merge policy greed
- Lookups vs. updates
- Max throughput
- Update cost
- Lookup cost
better asymptotic scalability

![Graph showing comparison between LevelDB and Monkey]

- **Y-axis:** Lookup latency (ms)
- **X-axis:** Number of entries (log scale)

- **LevelDB**
- **Monkey**
better asymptotic scalability

workload adaptability

lookup latency (ms)

number of entries (log scale)

% lookups in workload

throughput

LevelDB

Monkey

LevelDB

navigable Monkey

fixed Monkey

LevelDB
Map Trade-Offs

Navigate

What-if?

http://daslab.seas.harvard.edu/crimsondb/
Crimson DB

Map Trade-Offs
Navigate
What-if?

http://daslab.seas.harvard.edu/crimsondb/

Thanks!