BIG DATA SYSTEMS

NoSQL | Neural Networks | SQL | Graph | Data Science
piazza is active now  video access active
what you should be doing?

READING
Readings for this week (and systems project)


Optimal Bloom Filters and Adaptive Merging for LSM-Trees. Niv Dayan, Manos Athanassoulis, Stratos Idreos. ACM Transactions on Database Systems, Dec 2018

how to prepare slides

no bullets  2 colors  big text  images  animation for examples
how to prepare slides

no bullets   2 colors   big text   images   animation for examples

story

one message per slide   connection from slide to slide
how to prepare slides

no bullets  2 colors  big text  images  animation for examples

story

one message per slide  connection from slide to slide

1) prepare slides, 2) meet with a TF and Stratos the week before
How many and which structures are possible?

Can we compute performance w/o coding?
more in depth discussion on NoSQL storage + design optimizations
(systems project and related research projects)

first steps in constructing a design space
NoSQL Key-value Stores

Machine learning  Social media
Smart homes  Web browsers
Phones  Web-based apps
Security  Health devices
Graphs  Analytics

RocksDB  Google BigTable  MongoDB  SQLite  LinkedIn
Amazon DynamoDB  Cassandra  Apache HBase

Memory  Read  Update

Ism-tree  B-tree  Log+Index
insert (key-value)
insert (key-value)

buffer

Level 1

MEMORY

DISK
DISK

MEMORY

Level 1
Level 1
insert (key-value)
insert (key-value)

buffer

Level 1

Level 2

Level 3

...

Level N
insert (key-value)

buffer

Level 1

Level 2

Level 3

...

Level N
insert (key-value)

buffer

Level 1

Level 2

Level 3

... 

Level N

MEMORY

DISK

pages

SSTables

tiered

leveled

sorted
Level 1

Level 2

Level 3

...  

Level N

SSTables

pages

[1,0,0,1,1,1] hash fun.
bloom filters

[min-max] /page

fence pointers

MEMORY

DISK

[1,0,0,1,1,1] hash fun.
bloom filters

[min-max] /page

fence pointers

MEMORY

DISK

[1,0,0,1,1,1] hash fun.
bloom filters

[min-max] /page

fence pointers

MEMORY

DISK
[1,0,0,1,1,1] hash fun.  
[min-max] /page

bloom filters  
fence pointers

[1,0,0,1,1,1] hash fun.  
[min-max] /page

bloom filters  
fence pointers

get (key)

buffer

Level 1

Level 2

Level 3

...  

Level N

MEMORY

DISK

pages

SSTables

tiered

leveled

sorted
get (key)

**buffer**

- Level 1
- Level 2
- Level 3
- ... (levels indented)
- Level N

**MEMORY DISK**

- SSTables
- Pages

**bloom filters**

- [1,0,0,1,1,1] hash fun.
- [min-max] /page

**fence pointers**

- ...
get (key)

buffer

[1,0,0,1,1,1] hash fun.

bloom filters

[min-max] /page

fence pointers

Level 1

Level 2

Level 3

... ...

Level N

MEMORY

DISK

SSTables

pages

tiered

leveled

sorted

hash fun.

/get (key)
get (key)

buffer

Level 1

Level 2

Level 3

... (brackets)

Level N

bloom filters

fence pointers

[1,0,0,1,1,1] hash fun.

[min-max] /page

[1,0,0,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.

网友评论：

[1,0,0,1,1,1,1,1]

hash fun.
get (key)

buffer

Level 1

Level 2

Level 3

... ...

Level N

MEMORY

DISK

SSTables

pages

tiered

leveled

sorted

bloom filters

fence pointers

[1,0,0,1,1,1]
hash fun.

[min-max]/page

[...]

hash fun.

fence pointers

[...]

pages

SSTables

MEMORY

DISK
bloom filters  
[min-max] /page  
fence pointers

[1,0,0,1,1,1] hash fun.

get (key)

buffer

Level 1

Level 2

Level 3

Level N

MEMORY  

DISK

pages

SSTables

leveled  
tiered  
sorted
leveled

buffer

Level 1

Level 2

Level 3

Level N

bloom filters

fence pointers

[1,0,0,1,1,1]
hash fun.

[min-max]
pages

hash fun.

fence pointers

\[\text{[min-max]} / \text{page}\]

MEMORY

DISK

SSTables

pages

tiered

leveled

sorted

Leveled sorted tiered SSTables
[1,0,0,1,1,1] hash fun.
bloom filters

[min-max]
/fpage
fence pointers

buffer

Level 1

Level 2

Level 3

... ... ...

Level N

size ratio
merge policy
filters bits per entry
size of buffer/cache
internal k-v layout

DISK

MEMORY

pages

SSTables

tiered

leveled

sorted

hash fun. /page
DOMAIN?

- size ratio
- merge policy
- filters bits per entry
- size of buffer/cache
- internal k-v layout
DOMAIN?

- size ratio
- merge policy
- filters bits per entry
- size of buffer/cache
- internal k-v layout

AMPLIFICATION?

- Read
- Update
- Memory

(DASlab @ Harvard SEAS)
merging

writes

reads
when we do more

merging

writes

reads
when we do more

merging

writes  

reads
merging

Tiering
write-optimized

cassandra

Leveling
read-optimized

RocksDB
Tiering
write-optimized

Leveling
read-optimized
Tiering
write-optimized

gather

Leveling
read-optimized
Tiering
write-optimized

gather

merge & flush

Leveling
read-optimized
Tiering
write-optimized

gather

Leveling
read-optimized
Tiering
write-optimized

gather

Leveling
read-optimized

merge
Tiering
write-optimized

Leveling
read-optimized

gather

merge
Tiering
write-optimized

gather

Leveling
read-optimized

merge

flush↓
Tiering
write-optimized

gather

Leveling
read-optimized

merge
\log_R(N) \begin{align*}
\text{Tiering} & \quad \text{Leveling} \\
\text{write-optimized} & \quad \text{read-optimized}
\end{align*}
Tiering
write-optimized

Leveling
read-optimized

$\log_R(N)$

size ratio
Leveling
read-optimized

Tiering
write-optimized

$\log_R(N)$

size ratio

$R$ runs per level

1 run per level
Tiering
write-optimized

Leveling
read-optimized

size ratio $R$
Tiering
write-optimized

Leveling
read-optimized

1 run per level

size ratio $R \gg$
Tiering
write-optimized

Leveling
read-optimized

T runs per level

size ratio $R$

1 run per level
Tiering
write-optimized

$O(N)$ runs per level

Leveling
read-optimized

1 run per level

size ratio $R$ \uparrow
Tiering
write-optimized

$O(N)$ runs per level

Leveling
read-optimized

1 run per level

size ratio $R \uparrow$
Tiering
write-optimized

$O(N)$ runs per level

log

Leveling
read-optimized

1 run per level

sorted array

size ratio $R \uparrow$
log

Tiering

Leveling

sorted array
Tiering

Leveling

log

size ratio $R$

Leveling

sorted

array
Tiering

Leveling

size ratio $R$

log

sorted array
Tiering

Leveling

size ratio $R$

$R$ \uparrow \log

sorted array
what happens as we collect more data?
what happens as we collect more data?
both reads and writes get worse!

what happens as we collect more data?
what happens as we collect more data?
what happens as we collect more data?
BITS PER ENTRY IN FILTERS: OPTIMIZED OUT

Monkey: Optimal Navigable Key-Value Store
BITS PER ENTRY IN FILTERS: OPTIMIZED OUT

Monkey: Optimal Navigable Key-Value Store

@SIGMOD2017
BITS PER ENTRY IN FILTERS: OPTIMIZED OUT

Monkey: Optimal Navigable Key-Value Store

@SIGMOD2017

bits per entry: fixed per run

buffer

Level 1

Level 2

…

Level N
BITS PER ENTRY IN FILTERS: OPTIMIZED OUT

Monkey: Optimal Navigable Key-Value Store

@SIGMOD2017

worst lookup cost: sum of false positive rates

bits per entry: fixed per run

buffer

Level 1

Level 2

... Level N
The same memory budget is more impactful at smaller levels.
BITS PER ENTRY IN FILTERS: OPTIMIZED OUT

Monkey: Optimal Navigable Key-Value Store

@SIGMOD2017

the same memory budget is more impactful at smaller levels

buffer

Level 1

Level 2

…

Level N
BITS PER ENTRY IN FILTERS: OPTIMIZED OUT

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

*the same memory budget is more impactful at smaller levels*

- Buffer
  - Level 1
  - Level 2
  - …
  - Level N

![Graph showing lookup cost and update cost for different systems](image)

- WiredTiger
- Cassandra, HBase
- RocksDB, LevelDB

@SIGMOD2017
In this experiment, we show that Monkey significantly improves lookup latency as a function of the number of entries. The key observation is that for any configuration, Monkey achieves significantly lower lookup cost than LevelDB due to the tuning of its Bloom filters, as predicted by our analysis in Section 4.3. Hence, Monkey reaches the Pareto frontier and is therefore able to navigate a better trade-off continuum between update cost and zero-result lookup cost.

The results are shown in Figure 11 (D). For both Monkey and LevelDB, each lookup involves at least one I/O for the target key, and so lookup latency comprises at least one disk seek. We mark the approximate time to perform one seek on our hard disk. Any contribution to latency above this line arises due to false positives. The results show that for low FPRs at these lower levels, the number of false positives is significantly lower for Monkey than for LevelDB. Even though a lookup on any level is performed on average. The curve for LevelDB slightly decreases as temporal locality increases, the low FPRs at these lower levels mean that false positives are not significant in all cases. In this way, Monkey improves lookup latency by up to 60% smaller in this experiment, though the asymptotic improvement is set to 0.5, the workload is uniformly randomly ranging from 0 to 1 whereby the number of queries is set to 0.5, both Monkey and LevelDB degenerate into an LSM-tree with no filters, and so lookup cost is the same. As we increase the number of bits per entry, Monkey significantly reduces lookup cost.

The results are shown in Figure 11 (C). When the number of bits per entry in filters is optimized out, the performance of LevelDB with a smaller memory footprint (up to case of no memory budget for the filters, Monkey can match the performance of LevelDB using significantly less main memory. We set up this experiment by repeating the experimental setup multiple times, each time using a different configuration of size ratio and merge policy. We measure the average of the most recently updated entries receive most of the lookups, and when it is below 0.5 the least recently updated entries receive most of the lookups, and so they do not issue I/Os most of the time due to the filters. The key observation is that for any configuration, Monkey achieves a significantly lower lookup cost than LevelDB due to the tuning of its Bloom filters, as predicted by our analysis in Section 4.3. Hence, Monkey reaches the Pareto frontier and is therefore able to navigate a better trade-off continuum between update cost and zero-result lookup cost.

The results are shown in Figure 11 (E). When the number of bits per entry in filters is optimized out, the performance of LevelDB with a smaller memory footprint (up to case of no memory budget for the filters, Monkey can match the performance of LevelDB using significantly less main memory. We set up this experiment by repeating the experimental setup multiple times, each time using a different configuration of size ratio and merge policy. We measure the average of the most recently updated entries receive most of the lookups, and when it is below 0.5 the least recently updated entries receive most of the lookups, and so they do not issue I/Os most of the time due to the filters. The key observation is that for any configuration, Monkey achieves a significantly lower lookup cost than LevelDB due to the tuning of its Bloom filters, as predicted by our analysis in Section 4.3. Hence, Monkey reaches the Pareto frontier and is therefore able to navigate a better trade-off continuum between update cost and zero-result lookup cost.
MERGE POLICY: SHOULD BE TUNED

Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store
MERGE POLICY: SHOULD BE TUNED

Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store
MERGE POLICY: SHOULD BE TUNED

Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store

merge policy: fixed across levels

helps reads
Level 2
...
Level N

helps writes
MERGE POLICY: SHOULD BE TUNED

Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store
merging small levels does not help that much (point, range, space)

merge policy: fixed across levels

MERGE POLICY: SHOULD BE TUNED

Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store

buffer

Level 1

Level 2

... 

Level N
merging small levels does not help that much (point, range, space)
merging small levels does not help that much (point, range, space)

MERGE POLICY: SHOULD BE TUNED

Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store
merging small levels does not help that much (point, range, space)
Amazon Cloud (North America)

Data Size

20TB
200TB
2PB
20PB

monthly savings $\$

vs Google LevelDB

$10K
$500K
25% monthly savings $$ vs Google LevelDB

Data Size

$10K

20TB  200TB  2PB  20PB

Amazon Cloud (North America)
25% monthly savings $$
vs Google LevelDB

Data Size

Amazon Cloud (North America)
25% monthly savings $\$\$ $ vs Google LevelDB

Data Size

$10K 

20TB 200TB 2PB 20PB

Amazon Cloud (North America)
leveled

tiered

sorted

[1.0,0.1,1.1,1]
hash fun.

bloom filters

fence pointers

[min-max]
/page

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

buffer

Level 1

Level 2

Level 3

Level N

MEMORY

DISK

pages

SSTables

leveled

tiered

sorted

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.

[1,0,0,1,1,1]
hash fun.

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

hash fun.
LSM-trees

- size ratio
- merge policy
- filters bits per entry
- size of buffer/cache
- internal k-v layout
LSM-trees
B-trees
Logs
Arrays
Bitmaps

size ratio
merge policy
filters bits per entry
size of buffer/cache

key retention
value retention
partitioning
sub-block links
fanout
key retention
value retention
partitioning
sub-block links
fanout
unified design space
sorted zone map

POSSIBLE NODE DESIGNS
POSSIBLE NODE DESIGNS  POSSIBLE STRUCTURES

- Trie
- Array
- Skip-List
- Linked-List
- Sorted Array
- Hash-Table