Logistics

Systems projects and Labs are active.

Research projects to be released tomorrow:
  Design space NN & Image AI
Logistics
Systems projects and Labs are active. Research projects to be released tomorrow: Design space NN & Image AI

Discussion phase timeline
We need ~2 more weeks for lectures + at least 2 classes for new areas. Discussion phase papers and timeline: end of next week
Logistics
Systems projects and Labs are active.
Research projects to be released tomorrow:
   Design space NN & Image AI

Discussion phase timeline
We need ~2 more weeks for lectures + at least 2 classes for new areas
Discussion phase papers and timeline: end of next week

Action Items for now
Finish up the readings. Get familiar with projects.
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

worst lookup cost:
sum of false positive rates
BITS PER ENTRY IN FILTERS: OPTIMIZED OUT

Monkey: Optimal Navigable Key-Value Store

the same memory budget is more impactful at smaller levels

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

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

@SIGMOD2017

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

```
buffer

Level 1

Level 2

...  

Level N
```

![Diagram showing lookup cost and update cost with various databases like WiredTiger, Cassandra, HBase, RocksDB, and LevelDB. The diagram emphasizes the impact of memory budget at different levels.]
the same memory budget is more impactful at smaller levels

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 this source of latency using the dotted gray line, which represents \( \approx 0.2 \text{ I/Os per lookup} \). The curve for LevelDB slightly decreases as temporal locality increases because a lookup traverses fewer levels on average and so fewer false positives are evicted from the buffer. As temporal locality increases, the low FPRs at these lower levels mean that false positives are rare, and so they contribute very modestly to latency. The reason is that all but the last level have traversed one tier of filters, and so they are only sensitive to temporal locality. The reason is that in an LSM-tree the most recently updated entries are at the shallower levels, which have exponentially lower latency. The curve for LevelDB is largely insensitive to temporal locality because recent updates and updates to the last level are both less frequent.

In this experiment, we show that 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 lookup latencies of lookups and updates and plot them against each other.

Figure 11 (B) depicts results for a similar experiment, with the try set to 0, both Monkey and LevelDB degenerate into an LSM-tree with no filters, and so lookup cost is the same. As we increase the entry size, this has the same impact on the approximate time to perform one seek on our hard disk. Any increase is set to 0.5, the workload is uniformly randomly distributed.

Figure 11 (E) shows that Monkey reaches the Pareto frontier and is therefore able to navigate the design space to find the design that maximizes throughput (F). 

Figure 11: Monkey improves lookup cost under any (A) number of entries, (B) entry size, (C) amount of memory, (D) lookup locality, (E) merge policy and size ratio. It navigates the design space to find the design that maximizes throughput (F).

The key observation is that for any configuration, Monkey achieves significantly lower lookup cost than LevelDB due to the tuning of its data entries. The reason is that in an LSM-tree the most recently updated entries receive most of the lookups. When we define a coefficient \( r \) for non-zero-result lookups across a wide range of temporal locality, the update rate drops to nearly 0, at which point the latency. Eventually, the filters for both systems become so accurate that the latency is set to 0 and (F) merge policy and size ratio. It navigates the design space to find the design that maximizes throughput (F).
MERGE POLICY: SHOULD BE TUNED

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

@SIGMOD2018
MERGE POLICY: SHOULD BE TUNED

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

Do stoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store

merge policy: fixed across levels

helps reads

helps writes

Level 2

... 

Level N
Do\textsuperscript{stoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store}

MERGE POLICY: SHOULD BE TUNED

merge policy: fixed across levels

buffer

Level 1

Level 2

... 

Level N
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

buffer

Level 1

Level 2

...

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

buffer

Level 1

Level 2

...

Level N
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

@SIGMOD2018
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)
25% monthly savings $$ vs Google LevelDB

Data Size

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

Data Size

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

Data Size

20TB 200TB 2PB 20PB

$10K $500K

SSD SSD

Amazon Cloud (North America)
Summary

Once you understand the design, you can think of new ideas. Just keep asking “why”.
Summary

Once you understand the design, you can think of new ideas. Just keep asking “why”.

Tons of opportunities in big data as everything is new and changing.
Summary

Once you understand the design, you can think of new ideas. Just keep asking “why”.

Tons of opportunities in big data as everything is new and changing.

Once you think of a new idea, then it is just about following good research practices = requires technical skills but easier.
DESIGN SPACE

COST SYNTHESIS

WHAT-IF
DESIGN SPACE

COST SYNTHESIS

WHAT-IF
STARS IN THE SKY

10^{24}

POSSIBLE DATA STRUCTURES

10^{32}, 2-node
10^{48}, 3-node
COST?
What would the performance be if we were to implement that design in a specific programming language and test a specific workload on a specific hardware?
What would the performance be if we were to implement that design in a specific programming language and test a specific workload on a specific hardware?

If we have the cost for 2 designs we can compare them, and we can build search algos.
HOW TO JUDGE A DESIGN?
HOW TO JUDGE A DESIGN?

1

COMPLEXITY ANALYSIS
HOW TO JUDGE A DESIGN?

1. COMPLEXITY ANALYSIS
2. IMPLEMENTATION & TESTING
HOW TO JUDGE A DESIGN?

1. COMPLEXITY ANALYSIS
2. IMPLEMENTATION & TESTING
3. GENERALIZED MODELS
HOW TO JUDGE A DESIGN?

1. COMPLEXITY ANALYSIS
2. IMPLEMENTATION & TESTING
3. GENERALIZED MODELS

This sounds ideal: is it possible?
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

data* system
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

@SIGMOD 2017
ACCESS PATH SELECTION
scan vs secondary index selection

algorithms/operators
data structure 1
data structure 2
data structure 3

Pat Selinger
ACCESS PATH SELECTION
scan vs secondary index selection

Scan is best

Index is best

P. Selinger et. all, 1979
DO WE STILL NEED INDEXING? (AND IF YES HOW DO WE CHOOSE)
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

P. Selinger et. all, 1979

Scan is best

Index is best

Pat Selinger
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

Pat Selinger
P. Selinger et. all, 1979

selectivity

Scan is best

Index is best

selectivity

Scan is best

Index is best

multi-core, SIMD, compression, columnar/hybrid, scan sharing, …

# of concurrent queries

selectivity

Scan is best
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

@SIGMOD 2017

P. Selinger et al., 1979

selectivity

Index is best
Scan is best

multi-core, SIMD, compression, columnar/hybrid, scan sharing, …

# of concurrent queries

selectivity threshold

Hardware Improvements

Dawn of time

2000

Column Stores

Main Memory

2010

2017

latency

bandwidth

Future

2017

2010

2000

Dawn of time

0%

1%

10%

selectivity threshold
scan vs secondary index selection @SIGMOD 2017

![Diagram showing scan vs secondary index selection]

**Equation 16**

\[
\text{APS}(q, S_{\text{tot}}) = \frac{q \cdot \left[1 + \left\lfloor \log_b(N) \right\rfloor \right] \cdot \left( BW_S \cdot C_M + \frac{b \cdot BW_S \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2} \right)}{\max (ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S) + S_{\text{tot}} \cdot rw \cdot \frac{BW_S}{BW_R}}
\]

\[
S_{\text{tot}} \left( \frac{BW_S \cdot C_M}{b} + (aw + ow) \cdot \frac{BW_S}{BW_I} + rw \cdot \frac{BW_S}{BW_R} \right)
\]

\[
+ \frac{S_{\text{tot}} \cdot \log_2 (S_{\text{tot}} \cdot N) \cdot BW_S \cdot C_A}{\max (ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S) + S_{\text{tot}} \cdot rw \cdot \frac{BW_S}{BW_R}}
\]

**Workload**

<table>
<thead>
<tr>
<th>Workload</th>
<th>$q$</th>
<th>$s_i$</th>
<th>$S_{\text{tot}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of queries</td>
<td>selectivity of query $i$</td>
<td>total selectivity of the workload</td>
</tr>
</tbody>
</table>

**Dataset**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$N$</th>
<th>$t_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>data size (tuples per column)</td>
<td>tuple size (bytes per tuple)</td>
</tr>
</tbody>
</table>

**Hardware**

<table>
<thead>
<tr>
<th>Hardware</th>
<th>$C_A$</th>
<th>$C_M$</th>
<th>$BW_S$</th>
<th>$BW_R$</th>
<th>$BW_I$</th>
<th>$p$</th>
<th>$f_p$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 cache access (sec)</td>
<td>LLC miss: memory access (sec)</td>
<td>scanning bandwidth (GB/s)</td>
<td>result writing bandwidth (GB/s)</td>
<td>leaf traversal bandwidth (GB/s)</td>
<td>The inverse of CPU frequency</td>
<td>Factor accounting for pipelining</td>
</tr>
</tbody>
</table>

**Scan & Index**

<table>
<thead>
<tr>
<th>Scan &amp; Index</th>
<th>$rw$</th>
<th>$b$</th>
<th>$aw$</th>
<th>$ow$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>result width (bytes per output tuple)</td>
<td>tree fanout</td>
<td>attribute width (bytes of the indexed column)</td>
<td>offset width (bytes of the index column offset)</td>
</tr>
</tbody>
</table>
scan vs secondary index selection @SIGMOD 2017

\[
\text{APS}(q, S_{\text{tot}}) = \frac{q \cdot \left(1 + \left\lceil \log_b(N) \right\rceil \cdot (BW_S \cdot C_M + \frac{b \cdot BW_S \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2})}{\max \left( ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S \right) + S_{\text{tot}} \cdot rw \cdot \frac{BW_S}{BW_R}}
\]

\[
S_{\text{tot}} \left( \frac{BW_S \cdot C_M}{b} + (aw + ow) \cdot \frac{BW_S}{BW_I} + rw \cdot \frac{BW_S}{BW_R} \right) + \max \left( ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S \right) + S_{\text{tot}} \cdot rw \cdot \frac{BW_S}{BW_R} + \frac{S_{\text{tot}} \cdot \log_2 (S_{\text{tot}} \cdot N) \cdot BW_S \cdot C_A}{\max \left( ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S \right) + S_{\text{tot}} \cdot rw \cdot \frac{BW_S}{BW_R}}
\]

<table>
<thead>
<tr>
<th>Workload</th>
<th>( q )</th>
<th>number of queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( s_i )</td>
<td>selectivity of query ( i )</td>
</tr>
<tr>
<td></td>
<td>( S_{\text{tot}} )</td>
<td>total selectivity of the workload</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>( N )</th>
<th>data size (tuples per column)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( ts )</td>
<td>tuple size (bytes per tuple)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hardware</th>
<th>( C_A )</th>
<th>L1 cache access (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( C_M )</td>
<td>LLC miss: memory access (sec)</td>
</tr>
<tr>
<td></td>
<td>( BW_S )</td>
<td>scanning bandwidth (GB/s)</td>
</tr>
<tr>
<td></td>
<td>( BW_R )</td>
<td>result writing bandwidth (GB/s)</td>
</tr>
<tr>
<td></td>
<td>( BW_I )</td>
<td>leaf traversal bandwidth (GB/s)</td>
</tr>
<tr>
<td></td>
<td>( p )</td>
<td>The inverse of CPU frequency</td>
</tr>
<tr>
<td></td>
<td>( f_p )</td>
<td>Factor accounting for pipelining</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scan &amp; Index</th>
<th>( rw )</th>
<th>result width (bytes per output tuple)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( b )</td>
<td>tree fanout</td>
</tr>
<tr>
<td></td>
<td>( aw )</td>
<td>attribute width (bytes of the indexed column)</td>
</tr>
<tr>
<td></td>
<td>( ow )</td>
<td>offset width (bytes of the index column offset)</td>
</tr>
</tbody>
</table>
We need something else: Something more scalable & robust!
POSSIBLE DATA LAYOUTS

read

write

memory

operation
COST SYNTHESIS

ALGORITHM &

POSSIBLE DATA
LAYOUTS

operation

read

write

memory

ALGORITHM &

COST SYNTHESIS

DASlab
@ Harvard SEAS
synthesize access pattern

POSSIBLE DATA LAYOUTS

read

write

memory

operation

RULES

If ..., then ..., else

synthesize access pattern
sorted keys
columnar layout
sorted keys
columnar layout

RULES

sorted search
DEPENS ON
HARDWARE
ENGINEERING

sorted keys
columnar layout

RULES

sorted search

binary search1
binary search2
interpolation search1
interpolation search2
using new SIMD
instruction X
...

DEPENDS ON HARDWARE ENGINEERING
COMPONENTS OF KEY-VALUE ALGORITHMS

RULES

sorted keys
columnar layout

sorted search

batched write

BF probe

scan

binary search1
binary search2
interpolation search1
interpolation search2
using new SIMD instruction X
...
COMPONENTS OF KEY-VALUE ALGORITHMS

RULES

- Sorted keys
- Columnar layout

LEARNING

- Binary search 1
- Binary search 2
- Interpolation search 1
- Interpolation search 2
- Using new SIMD instruction X

...
SYNTHESIS FROM LEARNED MODELS

coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

e.g., binary search

```cpp
if (data[middle] < search_val) {
    low = middle + 1;
} else {
    high = middle;
}
middle = (low + high)/2;
```

1 11 17 37 51 66 80 94
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

   e.g., binary search

   ```
   C++
   if (data[middle] < search_val) {
     low = middle + 1;
   } else {
     high = middle;
   }
   middle = (low + high)/2;
   ```

2. BENCHMARK
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

e.g., binary search

```cpp
if (data[middle] < search_val) {
    low = middle + 1;
} else {
    high = middle;
} middle = (low + high)/2;
```

2. BENCHMARK

![Graph showing time versus data size](image)

3. FIT MODEL

$$f(x) = ax + b \log x + c$$
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

```cpp
if (data[middle] < search_val) {
    low = middle + 1;
} else {
    high = middle;
} middle = (low + high)/2;
```

2. BENCHMARK

```
Time (s)
```

3. FIT MODEL

```
f(x) = ax + b \log x + c
```

FOLDING ALGORITHMIC, ENGINEERING, AND H/W, PROPERTIES INTO THE COEFFICIENTS
<table>
<thead>
<tr>
<th>Data Access Primitives Level 1</th>
<th>Model Parameters</th>
<th>Data Access Primitives Layer 2</th>
<th>Fitted Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan (Element Size, Comparison, Data Layout; None)</td>
<td>Data Size</td>
<td>Scalar Scan (RowStore, Equal)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scalar Scan (RowStore, Range)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scalar Scan (ColumnStore, Equal)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scalar Scan (ColumnStore, Range)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMD-AVX Scan (ColumnStore, Equal)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMD-AVX Scan (ColumnStore, Range)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td>Sorted Search (Element Size, Data Layout; )</td>
<td>Data Size</td>
<td>Binary Search (RowStore)</td>
<td>Log-Lineal Model (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Binary Search (ColumnStore)</td>
<td>Log-Lineal Model (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interpolation Search (RowStore)</td>
<td>Log + LogLog Model (3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interpolation Search (ColumnStore)</td>
<td>Log + LogLog Model (3)</td>
</tr>
<tr>
<td>Hash Probe (; Hash Family)</td>
<td>Structure Size</td>
<td>Linear Probing (Multiply-shift [29])</td>
<td>Sum of Sigmoid (5), Weighted Nearest Neighbors (7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linear Probing (k-wise independent, k=2,3,4,5)</td>
<td>Sum of Sigmoid (5), Weighted Nearest Neighbors (7)</td>
</tr>
<tr>
<td>Bloom Filter Probe (; Hash Family)</td>
<td>Structure Size, Number of Hash Functions</td>
<td>Bloom Filter Probe (Multiply-shift [29])</td>
<td>Sum of Sigmoid (6), Weighted Nearest Neighbors (7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bloom Filter Probe (k-wise independent, k=2,3,4,5)</td>
<td>Sum of Sigmoid (6), Weighted Nearest Neighbors (7)</td>
</tr>
<tr>
<td>Sort (Element Size; Algorithm)</td>
<td>Data Size</td>
<td>QuickSort</td>
<td>NLogN Model (4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MergeSort</td>
<td>NLogN Model (4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ExternalMergeSort</td>
<td>NLogN Model (4)</td>
</tr>
<tr>
<td>Random Memory Access</td>
<td>Region Size</td>
<td>Random Memory Access</td>
<td>Sum of Sigmoid (5), Weighted Nearest Neighbors (7)</td>
</tr>
<tr>
<td>Batched Random Memory Access</td>
<td>Region Size</td>
<td>Batched Random Memory Access</td>
<td>Sum of Sigmoid (5), Weighted Nearest Neighbors (7)</td>
</tr>
<tr>
<td>Unordered Batch Write (Layout: )</td>
<td>Write Data Size</td>
<td>Contiguous Write (RowStore)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contiguous Write (ColumnStore)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td>Ordered Batch Write (Layout: )</td>
<td>Write Data Size, Data Size</td>
<td>Batch Ordered Write (RowStore)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Batch Ordered Write (ColumnStore)</td>
<td>Linear Model (1)</td>
</tr>
<tr>
<td>Scattered Batch Write</td>
<td>Number of Elements, Region Size</td>
<td>ScatteredBatchWrite</td>
<td>Sum of Sigmoid (6), Weighted Nearest Neighbors (7)</td>
</tr>
</tbody>
</table>
TRAINING
TRAINING
QUEYRYING
FOR EACH OPERATION
FOR EACH OPERATION

1. Decide access strategy (L1) based on node design
2. Decide exact access strategy implementation (L2) based on available models
3. Get cost for chosen model
K fences-pointer pairs, sorted

T key-value pairs, no order

STATE GENERATION
LAYOUT SPEC & INSERTS

# of nodes & # entries in each node

computed cost = average cost

DASlab @ Harvard SEAS
Accessing Level 3

random access

C++

```cpp
for(int i=0; i<size; i++)
    probe(array[pos[i]])
```

pos

1 7 6 2 3 5 4 0

array

12 56 9 37 1 45 11 20

random/sequential access

Run

Train

\[ f(x) = \sum_i \frac{c_i}{1 + e^{-k_i(x-x_i)}} \]
EASY EXTENSIBILITY OF LEVEL 2 ACCESS PRIMITIVES

just adding a new benchmark for a Level 1 primitive
can be used in any design!
CAN WE COMPUTE PERFORMANCE ACCURATELY?
CAN WE COMPUTE PERFORMANCE ACCURATELY?

layout spec $\rightarrow$ DC $\rightarrow$ cost $\quad$ VS $\quad$ C++ $\rightarrow$ cost

(same workload, hardware, data)
Response time (secs)

<table>
<thead>
<tr>
<th>Query Skew</th>
<th>CALCULATOR</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1</td>
<td>0.0004</td>
<td>0.0002</td>
</tr>
<tr>
<td>1.5</td>
<td>0.0006</td>
<td>0.0003</td>
</tr>
<tr>
<td>2</td>
<td>0.0008</td>
<td>0.0004</td>
</tr>
</tbody>
</table>

B+Tree

{10M (uniform) k-v pairs, 100 point queries (skewed)}
Response time (secs)

Query Skew

CALCULATOR IMPLEMENTATION

B+Tree

{10M (uniform) k-v pairs, 100 point queries (skewed)}
B+Tree

Response time (secs)

<table>
<thead>
<tr>
<th>Query Skew</th>
<th>CALCULATOR</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.0007</td>
<td>0.0006</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0005</td>
<td>0.0005</td>
</tr>
<tr>
<td>1.5</td>
<td>0.0004</td>
<td>0.0004</td>
</tr>
<tr>
<td>2.0</td>
<td>0.0003</td>
<td>0.0003</td>
</tr>
</tbody>
</table>

{10M (uniform) k-v pairs, 100 point queries (skewed)}
CALCULATOR IMPLEMENTATION

B+Tree

CSB+Tree

{10M (uniform) k-v pairs, 100 point queries (skewed)}
{10M (uniform) k-v pairs, 100 point queries (skewed)}
It works for numerous data structure classes and for diverse hardware and operations.

Training cost 50-100 secs

var h/w and op