BIG DATA SYSTEMS

Stratos Idreos

CS 265

NoSQL | Neural Networks | SQL | Graph | Data Science
Logistics:

Research projects available on website
Four lectures Week 4 and 5 for background/project intro
Project registration: end of Week 5

Systems project should start as of next week
Next week Tuesday, we will do intro to systems project
Daily Labs are active
NoSQL Key-value Stores

- RocksDB
- Google BigTable
- MongoDB
- SQLite
- LinkedIn
- Amazon DynamoDB
- Cassandra
- Apache HBase
- b-tree
- lsm-tree
- log+index

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

Design space for NoSQL KV-stores

Next level of technical detail in design: merging/levels
---

**Left pane:**

- Bloom filters
- [1,0,0,1,1,1] hash fun.
- Fence pointers
- [min-max] /page

---

**Central pane:**

- Buffer
  - Level 1
  - Level 2
  - Level 3
  - Level 4
  - ... Level N

---

**Right pane:**

- MEMORY
  - SSTables
  - Pages
  - Disk

---

**Footnote:**

DASlab © Harvard SEAS
Level 1

Level 2

Level 3

...  

Level N

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

hash fun.

[min-max]

/fpage

bloom filters

fence pointers

buffer

MEMORY

DISK

SSTables

pages

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

hash fun.

[min-max]

/fpage

bloom filters

fence pointers

buffer

MEMORY

DISK

SSTables

pages

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

levelled

tiered

sorted

size ratio

merge policy

filters bits per entry

size of buffer/cache

internal k-v layout

levelled

tiered

sorted
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
DOMINO?

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

AMPLIFICATION?

Read

Update

Memory
LSM-trees

- size ratio
- merge policy
- filters bits per entry
- size of buffer/cache
- internal k-v layout
key retention
value retention
partitioning
sub-block links
fanout
unified design space
utilization 50%

sorted

bloom filters off

no key retention

no value retention

sorted

bloom filters off

utilization 50%
POSSIBLE NODE DESIGNS
POSSIBLE NODE DESIGNS   POSSIBLE STRUCTURES
Data layout primitives

- UNSORTED DATAPAGE (TERMINAL ELEMENT)
- SORTED DATAPAGE (TERMINAL ELEMENT)
- COMPRESSED DATAPAGE (TERMINAL ELEMENT)
- LINKED LIST (NON-TERMINAL ELEMENT)

- RANGE PARTITIONING (NON-TERMINAL ELEMENT)
- HASH PARTITIONING (NON-TERMINAL ELEMENT)
- B+TREE ELEMENT (NON-TERMINAL ELEMENT)
- B-TREE ELEMENT (NON-TERMINAL ELEMENT)

- SKIP LIST (NON-TERMINAL ELEMENT)
- TRIE ELEMENT (NON-TERMINAL ELEMENT)
- LSM ELEMENT (NON-TERMINAL ELEMENT)
- LSM DATAPAGE (TERMINAL ELEMENT)

TRIE ELEMENT

SKIP LIST ELEMENT

B+TREE ELEMENT

B-TREE ELEMENT

SORTED DATAPAGE ELEMENT

DASlab
Harvard SEAS
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

**Fanout** (fixed/functional | unlimited | terminal |)
**Key partitioning** (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed/functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))

Intra node access (direct | head_link | tail_link | link_function(func))
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed/functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))

Intra node access (direct | head_link | tail_link | link_function(func))
Sub block links (next | previous | both | none)
Sub block skip links (perfect | randomized(prob: double) | function(func) | none)
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed/functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…) |)

Intra node access (direct | head_link | tail_link | link_function(func))

Sub block links (next | previous | both | none)
Sub block skip links (perfect | randomized(prob: double) | function(func) | none)

Zone Maps (min | max | both | exact | off)
Bloom filters (off | on(num_hashes: int, num_bits: int))
Filters layout (consolidate | scatter)
Links layout (consolidate | scatter)
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed/functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))

Intra node access (direct | head_link | tail_link | link_function(func))

Sub block links (next | previous | both | none)
Sub block skip links (perfect | randomized(prob: double) | function(func) | none)

Zone Maps (min | max | both | exact | off)
Bloom filters (off | on(num_hashes: int, num_bits: int))

Filters layout (consolidate | scatter)
Links layout (consolidate | scatter)

Physical location (inline | pointed | double- pointed)
Physical layout (BFS | scatter)
UNORDERED ARRAY

no order (fw-append)

[256]

UNORDERED LIST OF ARRAYS

No KV retention
No filters

Scattered next links

Variable fanout

Head Addressing

Sub-block physical layout: scatter
Sub-block consolidation: false
Sub-block instantiation: lazy

K V

no order (fw-append)

[256]
UNORDERED ARRAY

B+Tree

No KV retention

Consolidated Filters (Fences only)
Balanced
Fixed fanout
Direct Addressing
Sub-block location: pointed

Recursion: log(n)
Sub-blocks homogeneous: true
Sub-block physical layout: scatter
Sub-block consolidation: false
Sub-block instantiation: lazy

K
V
sorted

[256]
## Primitives and Instances

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Domain</th>
<th>Hash Table</th>
<th>B+Tree/CSB+Tree/FAST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node organization</strong></td>
<td></td>
<td></td>
<td>LPL</td>
<td>B+</td>
</tr>
<tr>
<td>1 Key retention. <strong>No</strong>: node contains no real key data, e.g., intermediate nodes of b+trees and linked lists. <strong>Yes</strong>: contains complete key data, e.g., nodes of b-trees, and arrays. <strong>Function</strong>: contains only a subset of the key, i.e., as in tries.</td>
<td>yes</td>
<td>no</td>
<td>function(func)</td>
<td>3</td>
</tr>
<tr>
<td>2 Value retention. <strong>No</strong>: node contains no real value data, e.g., intermediate nodes of b+trees, and linked lists. <strong>Yes</strong>: contains complete value data, e.g., nodes of b-trees, and arrays. <strong>Function</strong>: contains only a subset of the values.</td>
<td>yes</td>
<td>no</td>
<td>function(func)</td>
<td>3</td>
</tr>
<tr>
<td>3 Key order. Determines the order of keys in a node or the order of fences if real keys are not retained.</td>
<td>none</td>
<td>sorted</td>
<td>k-ary (k: int)</td>
<td>12</td>
</tr>
<tr>
<td>4 Key-value layout. Determines the physical layout of key-value pairs. <strong>Rules</strong>: requires key retention != no or value retention != no.</td>
<td>row-wise</td>
<td>columnar</td>
<td>col-row-groups(size: int)</td>
<td>12</td>
</tr>
<tr>
<td>5 Intra-node access. Determines how sub-blocks (one or more keys of this node) can be addressed and retrieved within a node, e.g., with direct links, a link only to the first or last block, etc.</td>
<td>direct</td>
<td>head_link</td>
<td>tail_link</td>
<td>function(func)</td>
</tr>
<tr>
<td>6 Utilization. Utilization constraints in regards to capacity. For example, &gt;= 50% denotes that utilization has to be greater than or equal to half the capacity.</td>
<td>= (%X)</td>
<td>function(func)</td>
<td>none</td>
<td>(we currently only consider X=50)</td>
</tr>
<tr>
<td>7 Bloom filters. A node's sub-block can be filtered using bloom filters. Bloom filters get as parameters the number of hash functions and number of bits.</td>
<td>off</td>
<td>on(num_hashes: int, num_bits: int)</td>
<td>(up to 10 num_hashes considered)</td>
<td>0-1</td>
</tr>
<tr>
<td>8 Zone map filters. A node's sub-block can be filtered using zone maps, e.g., they can filter based on mix/max keys in each sub-block.</td>
<td>min</td>
<td>max</td>
<td>both</td>
<td>exact</td>
</tr>
<tr>
<td>9 Filters memory layout. Filters are stored contiguously in a single area of the node or scattered across the sub-blocks. <strong>Rules</strong>: requires bloom filter != off or zone map filters != off.</td>
<td>consolidate</td>
<td>scatter</td>
<td>2</td>
<td>scatter</td>
</tr>
<tr>
<td>10 Fanout/Radix. Fanout of current node in terms of sub-blocks. This can either be unlimited (i.e., no restriction on the number of sub-blocks), fixed to a number, decided by a function or the node is terminal and thus has a fixed capacity.</td>
<td>fixed(value: int)</td>
<td>function(func)</td>
<td>hited</td>
<td>terminal(cap: int)</td>
</tr>
</tbody>
</table>
| 11 Key partitioning. Set if there is a pre-defined key partitioning imposed, e.g., the | }
<table>
<thead>
<tr>
<th>Partitions</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Immediate node links. Whether and how sub-blocks are connected.</td>
<td>next</td>
</tr>
<tr>
<td>14</td>
<td>Skip node links. Each sub-block can be connected to another sub-block (not only the next or previous) with skip-links. They can be perfect, randomized or custom.</td>
<td>perfect</td>
</tr>
<tr>
<td>15</td>
<td>Area-links. Each sub-tree can be connected with another sub-tree at the leaf level through area links. Examples include the linked leaves of a B+Tree.</td>
<td>forward</td>
</tr>
<tr>
<td>16</td>
<td>Sub-block physical location. This represents the physical location of the sub-blocks. Pointed: in heap. Inline: block physically contained in parent. Double pointed: in heap but with pointers back to the parent.</td>
<td>inline</td>
</tr>
<tr>
<td>17</td>
<td>Sub-block physical layout. This represents the physical layout of sub-blocks. Scatter: random placement in memory. BFS: laid out in a breadth-first layout. BFS layer list: hierarchical level nesting of BFS layouts.</td>
<td>BFS</td>
</tr>
<tr>
<td>18</td>
<td>Sub-blocks homogeneous. Set to true if all sub-blocks are of the same type.</td>
<td>boolean</td>
</tr>
<tr>
<td>19</td>
<td>Sub-block consolidation. Single children are merged with their parents.</td>
<td>boolean</td>
</tr>
<tr>
<td>20</td>
<td>Sub-block instantiation. If it is set to eager, all sub-blocks are initialized, otherwise they are initialized only when data are available (lazy).</td>
<td>lazy</td>
</tr>
<tr>
<td>21</td>
<td>Sub-block links layout. If there exist links, are they all stored in a single array (consolidate) or spread at a per partition level (scatter).</td>
<td>consolidate</td>
</tr>
</tbody>
</table>
| 22 | Recursion allowed. If set to yes, sub-blocks will be subsequently inserted into a node of the same type until a maximum depth (expressed as a function) is reached. Then the terminal node type of this data structure will be used. | yes(func) | no | yes(func) | no | yes(func) | no | yes(func) | no | yes(func) | no | yes(func) | no

Rules: requires fanout/radix != terminal.
STARS IN THE SKY

10^24

POSSIBLE DATA STRUCTURES

10^32, 2-node
10^48, 3-node
The TIGRIS Container Description Language and Compiler present initial results that indicate that TIGRIS can express for sophisticated automatic optimization and tuning. We implementations written in traditional systems languages like C, while being retargetable across platforms and allowing functional feature descriptions. We demonstrate the power of concise descriptions and the data structures in a concise manner, without sacrificing per- performance. We also show how rapid exploration of the container design space.

Novel applications demand new designs to achieve optimal data access for both point and range queries, and over such as the omnipresent B-tree are used to provide efficient container implementatio cient container implementatio...
TIGRIS specifications are orders of magnitude shorter than
This not only makes the code hard to maintain and debug,
workload.

1. INTRODUCTION
Consist of hundreds of lines of hand-tuned non-trivial code.

While essential, high-performance container
structures are notoriously di-
icult to write and typically

The TIGRIS Container Description Language and Compiler
We present a new programming model for container data
structures in a concise manner, without sacrificing per-
formance. We also show how concise descriptions and the
reuse of optimizations can facilitate rapid exploration of the
container design space.

While some

specifications to highly-e
fficient container implementatio

can express
for sophisticated automatic optimization and tuning. We
C, while being retargetable across platforms and allowing
implementations written in traditional systems languages li

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.
Permission to make digital or hard copies of all or part of this work for
personal or classroom use is granted without fee provided that copies are
not made or distributed for profit or commercial advantage and that copies
republish, to post on servers or to redistribute to lists, requires prior specific
not made or distributed for profit or commercial advantage and that copies
personal or classroom use is granted without fee provided that copies are

Consequently, 

er pool and fast lookup 

10^24

10^32, 2-node
10^48, 3-node

10^{48}-5 \times 10^3 = 10^{48} \quad \text{zero progress?}
The Periodic Table of the Elements explains and predicts missing elements based on atomic number, electron configuration, and recurring chemical properties.

**PERIODIC TABLE OF ELEMENTS**

Dimitri Mendelev
## Periodic Table of Data Structures

<table>
<thead>
<tr>
<th>Classes of Designs</th>
<th>B-trees &amp; Variants</th>
<th>Tries &amp; Variants</th>
<th>LSM-Trees &amp; Variants</th>
<th>Differential Files</th>
<th>Membership Tests</th>
<th>Zone maps &amp; Variants</th>
<th>Bitmaps &amp; Variants</th>
<th>Hashing</th>
<th>Base Data &amp; Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Logarithmic Design</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fractional Cascading</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log-Structured</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffering</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
</tr>
<tr>
<td>Differential Updates</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse Indexing</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Adaptivity</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
</tr>
</tbody>
</table>

Classes of primitives

Partitioning: DONE
Logarithmic Design: DONE
Fractional Cascading: DONE
Log-Structured: DONE
Buffering: DONE
Differential Updates: DONE
Sparse Indexing: DONE
Adaptivity: DONE
# Periodic Table of Data Structures

<table>
<thead>
<tr>
<th>Classes of Designs</th>
<th>B-trees &amp; Variants</th>
<th>Tries &amp; Variants</th>
<th>LSM-Trees &amp; Variants</th>
<th>Differential Files</th>
<th>Membership Tests</th>
<th>Zone Maps &amp; Variants</th>
<th>Bitmaps &amp; Variants</th>
<th>Hashing</th>
<th>Base Data &amp; Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Logarithmic Design</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Fractional Cascading</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log-Structured</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffering</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Differential Updates</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse Indexing</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptivity</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
</tr>
</tbody>
</table>

Classes of Primitives

© IEEE.EngBul18
## Periodic Table of Data Structures

<table>
<thead>
<tr>
<th>Classes of Designs</th>
<th>B-trees &amp; Variants</th>
<th>Tries &amp; Variants</th>
<th>LSM-Trees &amp; Variants</th>
<th>Differential Files</th>
<th>Membership Tests</th>
<th>Zone Maps &amp; Variants</th>
<th>Bitmaps &amp; Variants</th>
<th>Hashing</th>
<th>Base Data &amp; Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Differential Updates</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse Indexing</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptivity</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PAPER MACHINE**
## Periodic Table of Data Structures

<table>
<thead>
<tr>
<th>Classes of Designs</th>
<th>B-trees &amp; Variants</th>
<th>Tries &amp; Variants</th>
<th>LSM-Trees &amp; Variants</th>
<th>Differential Updates</th>
<th>Membership Tests</th>
<th>Zone Maps &amp; Variants</th>
<th>Bitmaps &amp; Variants</th>
<th>Hashing</th>
<th>Base Data &amp; Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Differential</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Updates</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse Indexing</td>
<td>DONE</td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptness</td>
<td>DONE</td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Paper Machine**

- **Updatable Bitmap Indexes**
  - @SIGMOD16

---

![Logo](image)
“The taxonomy is used to shed light both on the nature of the design space and on the performance tradeoffs implied by many of the choices that exist in the design space.”
“The taxonomy is used to shed light both on the nature of the design space and on the performance tradeoffs implied by many of the choices that exist in the design space.”
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

Leveling
read-optimized

gather
Tiering
write-optimized

gather

Leveling
read-optimized

merge
Tiering
write-optimized

gather

Leveling
read-optimized

merge
Tiering
write-optimized

Leveling
read-optimized

gather

merge

flush
Tiering
write-optimized

gather

Leveling
read-optimized

merge
\[ \log_R(N) \]

Tiering
write-optimized

Leveling
read-optimized
Leveling
write-optimized

Tiering
read-optimized

\[ \log^R(N) \]

size ratio
\[ \log_R(N) \]

- **Tiering**
  - write-optimized
  - \( R \) runs per level

- **Leveling**
  - read-optimized
  - 1 run per level
Tiering  
write-optimized

Levelling  
read-optimized

$R$ runs per level

$1$ run per level

$\bowtie$ size ratio $R$
Tiering
write-optimized

Leveling
read-optimized

1 run per level

size ratio $R \gg$
Tiering
write-optimized

$T$ runs per level

Leveling
read-optimized

$1$ run per level

$\text{size ratio } R$
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

size ratio $R \uparrow$
Tiering
write-optimized

O( N ) runs per level

Leveling
read-optimized

1 run per level

log

sorted array

size ratio $R \uparrow$
Tiering

Leveling

log

sorted array
Tiering

log

size ratio $R$

Leveling

sorted array
Tiering

Leveling

size ratio $R$

log

sorted array
Tiering

Leveling

size ratio $R$

$log$

$R$

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?