BIG DATA SYSTEMS

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

Second MLsys systems project ready.
Optimize data movement for neural network training
One vision and one LLM model.
Implementing M2 paper.
Available tomorrow on the class website.
Second more detailed version in 2 weeks.

μ-TWO: 3x Faster Multi-model Training with Orchestration and Memory Optimization
Logistics: Systems Projects

Lab hours ready (on the class website).
Labs start next week.
Three times a week. Two in person. One Zoom (link on class website).
Labs are for systems projects only. Research projects will have diff sessions.
Logistics: Systems Projects

**Lab hours ready** (on the class website).
Labs start next week.
Three times a week. Two in person. One Zoom (link on class website).
Labs are for systems projects only. Research projects will have diff sessions.

**Systems projects can “start” as of next week**
Go to lab to start understanding what is needed.
It will be good practice to have a 2-3 week warm up (coding/background)
Next week Tuesday

Raul Castro, University of Chicago

On Data Ecology, Data Markets, the Value of Data, and Dataflow Governance
fundamental building blocks properties when combined
fundamental building blocks

properties when combined
Three steps required

- DESIGN SPACE
- COST SYNTHESIS
- WHAT-IF
Today:

Building a design space in detail: Data structures

Next level of technical detail in KV-stores: merging/levels
bloom filters

fence pointers

hash fun.

[min-max]
/page

hash fun.

/filters

/bit
/entry

page

size ratio

merge policy

size of buffer/cache

internal k-v layout

Level 1

Level 2

Level 3

Level N

leveled

tiered

sorted

MEMORY

DISK

SSTables

pages
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
DOMAIN?

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

AMPLIFICATION?

Memory

Read

Update
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
utilization 50%
no key retention
no value retention
sorted
bloom filters off
internal k-v layout
size ratio
merge policy
fi filters bits per entry
size of buffer/cache
key retention
value retention
partitioning
sub-block links
fanout
uni uni
fi fi
design
design
space
space
sorted
zone map
bloom
bloom
filter bits
filter bits
link
link
layout
layout
50%
no key retention
no value retention
sorted
bloom filters off
POSSIBLE NODE DESIGNS

bloom filter bits
sorted
children
layout
link

10^6
POSSIBLE NODE DESIGNS   POSSIBLE STRUCTURES
POSSIBLE NODE DESIGNS

POSSIBLE STRUCTURES

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

Unknown 1
Unknown 2
Unknown N

Sorted Zone Map
Bloom Filter Bits
Link Children
Layout

@SIGMOD18
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

UNORDERED LIST OF ARRAYS
### Primitives and Instances

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Domain</th>
<th>Hash Table</th>
<th>B+Tree/CSB+Tree/FAST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key retention.</strong> <em>No:</em> node contains no real key data, e.g., intermediate nodes of b+trees and linked lists. <em>Yes:</em> contains complete key data, e.g., nodes of b+trees, and arrays. <em>Function:</em> contains only a subset of the key, i.e., as in tries.</td>
<td>yes</td>
<td>no</td>
<td>function (func)</td>
</tr>
<tr>
<td><strong>Value retention.</strong> <em>No:</em> node contains no real value data, e.g., intermediate nodes of b+trees, and linked lists. <em>Yes:</em> contains complete value data, e.g., nodes of b+trees, and arrays. <em>Function:</em> contains only a subset of the values.</td>
<td>yes</td>
<td>no</td>
<td>function (func)</td>
</tr>
<tr>
<td><strong>Key order.</strong> 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>
</tr>
<tr>
<td><strong>Key-value layout.</strong> Determines the physical layout of key-value pairs. <em>Rules:</em> requires key retention != no or value retention != no.</td>
<td>row-wise</td>
<td>columnar</td>
<td>col-row-groups (size: int)</td>
</tr>
<tr>
<td><strong>Intra-node access.</strong> 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</td>
<td>link</td>
</tr>
<tr>
<td><strong>Utilization.</strong> 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>(%)</td>
<td>function (func)</td>
<td>none (we currently only consider K=50)</td>
</tr>
<tr>
<td><strong>Bloom filters.</strong> 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</td>
<td>(num_hashes: int, num_bits: int)</td>
</tr>
<tr>
<td><strong>Zone map filters.</strong> 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>
</tr>
<tr>
<td><strong>Filters memory layout.</strong> Filters are stored contiguously in a single area of the node or scattered across the sub-blocks. <em>Rules:</em> requires bloom filter != off or zone map filters != off.</td>
<td>consolidate</td>
<td>scatter</td>
<td></td>
</tr>
<tr>
<td><strong>Fanout/Radix.</strong> 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</td>
<td>(value: int)</td>
<td>function (func)</td>
</tr>
<tr>
<td><strong>Key partitioning.</strong> Sets if there is a pre-defined key partitioning imposed, e.g., the sub-block where a key is located can be dictated by a radix or range partitioning node or scattered across the sub-blocks.</td>
<td>fanout/radix</td>
<td>!= terminal.</td>
<td>node</td>
</tr>
</tbody>
</table>

---

**Rules:**
- Requires fanout/radix != terminal.
- Requires key partitioning != none.

**Intra-node access:**
- 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.
- BFS layer list: hierarchical level nesting of BFS layouts.
- BFS | BFS layer (level-grouping: int, merge_policy: [tier | level])
- Utilization constraints in regards to capacity. For example, >= 50% Utilization.
- Domains: row-wise | columnar | col-row-groups (size: int)
- Direct | head_link | tail_link | link_function (func)
- Forw. | Backw. | Both | Exact | Off
- Fixed (value: int) | Function (func) | Balanced | None | Function (func)
- No: | Yes: |
- CSB+ | FAST | B+Tree/CSB+Tree/FAST
- Primitives and Instances | LPL | B+ | CSB+ | FAST | ODP

<table>
<thead>
<tr>
<th>Domain</th>
<th>H</th>
<th>LL</th>
<th>UDP</th>
<th>B+</th>
<th>CSB+</th>
<th>FAST</th>
<th>ODP</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>none</td>
<td>sorted</td>
<td>sorted</td>
<td>4-ary</td>
<td>sorted</td>
<td></td>
</tr>
<tr>
<td>direct</td>
<td>head</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td></td>
</tr>
<tr>
<td>direct</td>
<td>head</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td></td>
</tr>
<tr>
<td>direct</td>
<td>head</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td>direct</td>
<td></td>
</tr>
<tr>
<td>off</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td></td>
</tr>
<tr>
<td>off</td>
<td>off</td>
<td>off</td>
<td>min</td>
<td>min</td>
<td>min</td>
<td>off</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>max</td>
<td>both</td>
<td>exact</td>
<td>off</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>off</td>
<td>off</td>
<td>off</td>
<td>term (256)</td>
<td>fixed (256)</td>
<td>term (256)</td>
<td>term (256)</td>
<td></td>
</tr>
</tbody>
</table>
### Rules:
- **Immediate node links.** Whether and how sub-blocks are connected.
  - next | previous | both | none

- **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.
  - perfect | randomized(prob: double) | function(func)

- **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.
  - forward | backward | both | none

### 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.

- Rules: requires fanout/radix  != terminal.

### 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.

- Rules: requires fanout/radix  != terminal.

### Sub-block homogeneity.
Set to true if all sub-blocks are of the same type.

- Rules: requires fanout/radix  != terminal.

### Sub-block consolidation.
Single children are merged with their parents.

- Rules: requires fanout/radix  != terminal.

### Sub-block instantiation.
If it is set to eager, all sub-blocks are initialized, otherwise they are initialized only when data are available (lazy).

- Rules: requires fanout/radix  != terminal.

### 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).

- Rules: requires immediate node links != none or skip links != none.

### 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.

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

POSSIBLE DATA STRUCTURES

$10^{24}$

$10^{32}$, 2-node

$10^{48}$, 3-node
TIGRIS specifications are orders of magnitude shorter than
This not only makes the code hard to maintain and debug,
workload.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.
permission and/or a fee.
republish, to post on servers or to redistribute to lists, requires prior specific
bear this notice and the full citation on the first page. To copy otherwise, to
not made or distributed for profit or commercial advantage and that copies
personal or classroom use is granted without fee provided that copies are
implementations written in traditional systems languages li

Consequently,

while being retargetable across platforms and allowing
functional feature descriptions. We demonstrate the power of
a large variety of traditional and special-purpose container
structures in a concise manner, without sacrificing per-
quickly iterating between alternative designs.

ABSTRACT
The TIGRIS Container Description Language and Compiler
We present a new programming model for container data
implementations written in traditional systems languages li
by describing an optimizing compiler that compiles TIGRIS
language for container data structures, called TIGRIS, and
this separation by introducing a declarative domain specific

1. INTRODUCTION

Conversely, over the last five decades, hundreds
of specialized container designs have been proposed and
those structures have been proposed, including compression,
the last decades many optimizations and specializations to
such as the omnipresent B-tree are used to provide e
queries can be quickly answered using compact bit-vectors
result between data processing stages, while fast membership
different forms are commonly used to store intermediate re-
tainer data structures to store large collections of base and
programs would store all their data in dictionaries,
ating on top of the data collection. Contrary,
fully loaded into memory and the program is directly oper-
collections). Especially in smaller programs, data is often
paged collections and column-major layouts (i.e., tuples of
collections). Base data is commonly stored in lists
update behavior for their respective use case.
storage and retrieval.
data structures

POSSIBLE DATA STRUCTURES

10^24

STARS IN THE SKY

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

10^24

~5K since the dawn of CS
This not only makes the code hard to maintain and debug, but also prevents data structure and system designers from rapidly iterating between alternative designs. Consequently, container designs are notorious for being hard to write and typically consist of hundreds of lines of hand-tuned non-trivial code. While essential, high-performance container performance needs to be closely tuned to both the underlying hardware and software. 

Integral to data system architectures, container structures need to rapidly iterate between alternative designs. We present initial results that indicate that TIGRIS can express different container design space.

We demonstrate the power of container design space.

The TIGRIS Container Description Language and Compiler

1. INTRODUCTION

Containers in Data Systems

Modern data systems use a large variety of container data structures to store large collections of base and intermediate data, and to provide auxiliary structures such as indexes and fast lookup tables. Because of their central role in data system architectures, container structures need to quickly iterate between alternative designs. 

While some applications would store all their data in dictionaries, this separation by introducing a declarative domain specific functional feature descriptions. We demonstrate the power of these container data structures to store large collections of base and intermediate data, and to provide auxiliary structures such as indexes and fast lookup tables. Because of their central role in data system architectures, container structures need to quickly iterate between alternative designs.

Conferences in major database conferences refer- encing indexes, trees or access methods in their title (as shown in Figure 1).

Data-intensive programs, from caching and algorithmical optimizations. Hash tables and ordered tree structures also provide the foundation of many data structures. Base data is commonly stored in lists or orderless collections, while secondary indexes, intermediate data, and meta-data are often stored in highly specialized containers.

10^24

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

~5K since the dawn of CS

10^{48} - 5 \times 10^3 = 10^{48} 

zero progress?
The Periodic Table of the Elements structures elements based on atomic number, electron configuration, and recurring chemical properties.
nihonium
# 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>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Logarithmic Design</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fractional Cascading 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>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td>DONE</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>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptivity</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

@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 Design</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>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></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></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>
### Periodic Table of Data Structures

<table>
<thead>
<tr>
<th></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><strong>Partitioning</strong></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><strong>Logarithmic</strong></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><strong>Differential Updates</strong></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sparse Indexing</strong></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Adaptivity</strong></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**

- **classes of designs**
- **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>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
</tbody>
</table>

**Paper Machine**

- **Updatable Bitmap Indexes**
  - @SIGMOD16
“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.”
Understanding the KV design space in more detail: size ratio and merging

- bloom filters
- fence pointers
- [1,0,0,1,1,1] hash fun.
- [min-max] /page

MEMORY

DISK

SSTables

pages

tiered

leveled

sorted

Level 1

Level 2

Level 3

......

Level N

hash fun.

fence pointers

buffer

[3,0,3,3,3]
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

Leveling
read-optimized

gather
Tiering
write-optimized

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
\( \log_R(N) \)

Tiering
write-optimized

Leveling
read-optimized
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

\( R \) runs per level

\( 1 \) run per level

size ratio \( R \)
Tiering
write-optimized

1 run per level

Leveling
read-optimized

1 run per level

size ratio $R \gg$
Tiering
write-optimized

size ratio $R$

Leveling
read-optimized

$T$ runs per level

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

log

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 $R \uparrow$
Tiering

Leveling

sorted array

log
Tiering

size ratio $R$

Leveling

log

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?
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?
Readings for this week (and systems project)


