WHAT IF WE COULD REASON ABOUT THE DESIGN SPACE OF DATA STRUCTURES?
1. no perfect structure
1. no perfect structure
1. no perfect structure
2. apps & h/w evolve
GET $N$ SMART PEOPLE
GIVE THEM $T$ TIME
HOPE FOR THE BEST
GET $N$ SMART PEOPLE
GIVE THEM $T$ TIME
HOPE FOR THE BEST
AUTO DESIGN
“Is there a calculus of data structures by which one can choose the appropriate representation and techniques for a given problem?” (SIAM, 1978)
HOW MANY AND WHICH?

COMPUTE PERFORMANCE?
DESIGN SPACE OF POSSIBLE STORAGE LAYOUTS

- Workload
- Layout design
- H/w
DESIGN SPACE OF POSSIBLE STORAGE LAYOUTS

without coding or accessing the h/w
WHAT-IF DESIGN

AUTO-DESIGN

SELF-DESIGNING SYSTEMS
physical layout, e.g., partitioning
DATA
INDEX
DESIGN SPACE

metadata, model, function, filters

physical layout, e.g., partitioning
DATA
INDEX
DESIGN SPACE

physical layout, e.g., partitioning

metadata, model, function, filters

puts, get, delete update, range
EACH DESIGN: A SET OF CONCEPTS
{partitioning, links, fence pointers,...}
2 DESIGN

{ COMBINATION
  of existing concepts
}

TUNING
  of existing concepts

NEW concept
(ALMOST) ALL DESIGNS ARE A COMBINATION/TUNING OF EXISTING CONCEPTS
I hope for nothing.
I fear nothing.
I am free.

Nikos Kazantzakis
if we know the **fundamental** building blocks,
if we know the **fundamental** building blocks, how they combine and their properties,
if we know the **fundamental** building blocks, how they combine and their properties,

then we can **automate** the discovery of novel combinations and tunings
if we know the fundamental building blocks, how they combine and their properties, then we can automate the discovery of novel combinations and tunings
structures elements based on atomic number, electron configuration, and recurring chemical properties
Kosuke Morita

nihonium
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.
trial and error
FIRST PRINCIPLE: DESIGN CONCEPT THAT IS NOT POSSIBLE OR MEANINGFUL TO BREAK FURTHER
MAP LAYOUT FIRST

1. KNOWN DESIGNS
2. OPEN QUESTIONS
{arrays, logs, lsm-trees, b-trees}, filters, bitmaps, compression, stats

- e.g., 1000x NoSQL k-v: bloom filter bits, merging policy
- e.g., access path selection: scans vs b-tree depends on concurrency
- e.g., robust scans with value by value lossy compression
- e.g., updatable bitmap indexes
- e.g., fast statistics/ML

...
EXAMPLE: The design space of NoSQL Key-value Stores
insert (key-value)
insert (key-value)
insert (key-value)

buffer

Level 1

Level 2

MEMORY

DISK
insert (key-value)

buffer

Level 1

Level 2

Level 3

MEMORY

DISK
insert (key-value)
insert (key-value)

buffer

Level 1

Level 2

Level 3

...

Level N

MEMORY

DISK

tiered

leveled

sorted
insert (key-value)
bloom filters

hash fun.

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

…

Level 1

Level 2

Level 3

…

Level N

buffer

MEMORY

DISK

SSTables

pages

tiered

leveled

sorted

hash fun.
bloom filters

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

fence pointers

[min-max] [min-max]

buffer

Level 1

Level 2

Level 3

... ... ...

Level N

MEMORY

DISK

pages

SSTables

leveled
tiered

sorted

hash fun.

fence pointers

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

get (key)

buffer

Level 1

Level 2

Level 3

... 

Level N

MEMORY

DISK

SSTables

pages

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

[fence, pointers]

[min-max]
get (key)

buffer

Level 1

Level 2

Level 3

...  

Level N

MEMORY

DISK

pages

SSTables

leveled  
tiered  
sorted

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

bloom filters

fence pointers

[min-max]

hash fun.

fence pointers

[min-max]
get (key)

- **buffer**

**Memory**

- **Level 1**
- **Level 2**
- **Level 3**
- **Level N**

**Disk**

- SSTables

**Leveled, Tiered, Sorted**

- Bloom filters
- Fence pointers

- [1,0,0,1,1,1] hash fun.
- [min-max]
get (key)

buffer

Level 1

Level 2

Level 3

... 

Level N

SSTables

pages

DISK

MEMORY

bloom filters

fence pointers

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

[min-max]

hash fun.

fence pointers

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

[min-max]

hash fun.

fence pointers

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

[min-max]
get (key)

buffer

bloom filters

disk

fence pointers

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

[min-max]

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.

fence pointers

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

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

hash fun.
get (key)

buffer

Level 1
Level 2
Level 3
Level N

DIGITAL SYSTEMS LABORATORY
@ Harvard SEAS
get (key)

buffer

Level 1

Level 2

Level 3

Level N

bloom filters

fence pointers

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

[min-max]

Level 1

Level 2

Level 3

Level N

MEMORY

DISK

pages

SSTables

leveled

tiered

sorted

hash fun.

fence pointers

[min-max]
Instant writes, but overtime have to be merged >1

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

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
hash fun.
```

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

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

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
```

```
buffer
```

```
Level 1
```

```
Level 2
```

```
Level 3
```

```
Level N
```

```
SSTables
```

```
MEMORY
```

```
MEMORY
```

```
DISK
```

```
pages
```

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

```
fence pointers
```

```
bloom filters
```

```
[min-max]
Instant writes, but overtime have to be merged >1
Temporal partitioning = recent data to the top

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

buffer

MEMORY
DISK

Level 1
Level 2
Level 3
... Level N

SSTables
guaranteed leveled and sorted

pages
Instant writes, but overtime have to be merged >1
Temporal partitioning = recent data to the top
But duplicates across levels (memory amplification)
Instant writes, but overtime have to be merged >1
Temporal partitioning = recent data to the top
But duplicates across levels (memory amplification)
Reads can get away with 0-1 disk page read per level
Instant writes, but overtime have to be merged >1
Temporal partitioning = recent data to the top
But duplicates across levels (memory amplification)
Reads can get away with 0-1 disk page read per level
Run layout affect read/write costs

MEMORY
DISK

levels

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

[min-max]

bloom filters

fence pointers

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

Level 1

Level 2

Level 3

Level N

Temporal partitioning = recent data to the top
But duplicates across levels (memory amplification)
Reads can get away with 0-1 disk page read per level
Run layout affect read/write costs

DISK
MEMORY

SSTables

pages

Level N

leveled

tiered

sorted

hash fun.

fence pointers

bloom filters

[1,0,0,1,1,1]
Instant writes, but overtime have to be merged >1
Temporal partitioning = recent data to the top
But duplicates across levels (memory amplification)
Reads can get away with 0-1 disk page read per level
Run layout affect read/write costs
Size ratio affects everything
[1,0,0,1,1,1]
hash fun.

bloom filters

[min-max]

fence pointers

buffer

Level 1

Level 2

Level 3

Level N

size ratio
merge policy
filters bits per entry
size of (page, buffer, ..)
internal k-v layout

MEMORY

DISK

pages

SSTables

leveled
tiered
sorted

hash fun.

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

bloom filters

[min-max]

fence pointers

buffer

Level 1

Level 2

Level 3

Level N

size ratio
merge policy
filters bits per entry
size of (page, buffer, ..)
internal k-v layout

MEMORY

DISK

pages

SSTables

leveled
tiered
sorted

hash fun.

[1,0,0,1,1,1]
For every design principle X:

1. Which are X’s meaningful values?
2. How does X affect read, update and memory amplification?
3. Should X be a design principle or can it be optimized out?
bits per entry in filters

monkey@SIGMOD2017

Monkey: Optimal Navigable Key-Value Store
standard design: fixed per run

at most one I/O per level

worst case I/O: sum of false positive rates

buffer

Level 1

Level 2

...  

Level N

bits per entry in filters

monkey@SIGMOD2017

Monkey: Optimal Navigable Key-Value Store
minimize sum of FPRs

"move" filters memory budget
from big to small levels

buffer

Level 1

Level 2

... 

Level N

bits per entry in filters

monkey@SIGMOD2017
Monkey: Optimal Navigable Key-Value Store
minimize sum of FPRs

"move" filters memory budget from big to small levels

bits per entry in filters

monkey@SIGMOD2017

Monkey: Optimal Navigable Key-Value Store

Bits per entry in bloom filter

buffer

Level 1

Level 2

...

Level N
minimize sum of FPRs

"move" filters memory budget from big to small levels

buffer

Level 1

Level 2

…

Level N

Bits per entry in bloom filter

update cost

lookup cost

WiredTiger

Cassandra, HBase

RocksDB, LevelDB

monkey

Monkey: Optimal Navigable Key-Value Store

monkey@SIGMOD2017
minimize sum of FPRs

"move" filters memory budget from big to small levels

---

buffer

Level 1

Level 2

…

Level N

---

Bits per entry in bloom filter

---

monkey@SIGMOD2017

Monkey: Optimal Navigable Key-Value Store

---

![Graph showing lookup latency](image)

- LevelDB
- Monkey

uniform, zero result, point queries, entry size=1KB

---

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 \( \approx 0.2 \) I/Os per lookup.
merge policy
dostoevsky@SIGMOD2018
Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store
standard design: fixed across levels

Leveled helps reads; tiered helps writes

buffer

Level 1

Level 2

Level N

Bits per entry in bloom filter

merge policy
dostoevsky@SIGMOD2018
Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store
hybrid merge policy
tiered for small levels

merge policy
dostoevsky@SIGMOD2018
Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store

buffer

Level 1

Level 2

...

Level N

Bits per entry in bloom filter
hybrid merge policy
tiered for small levels

buffer

Level 1

Level 2

... Level N

Bits per entry in bloom filter
hybrid merge policy
tiered for small levels

buffer

Level 1

Level 2

... Level N

Bits per entry in bloom filter

update cost

lookup cost

WiredTiger

Cassandra, HBase

RocksDB, LevelDB

monkey

dosto

Dostoevsky: Space-Time Optimized Evolvable Scalable Key-Value Store
**hybrid merge policy**

tiered for small levels

- buffer
- Level 1
- Level 2
- ... Level N

- Bits per entry in bloom filter

---

**Dostoevsky**

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

---

**merge policy**

**dostoevsky@SIGMOD2018**

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

---

**Normalized throughput (ops/s)**

- 
  - Dostoevsky
  - Monkey
  - RocksDB well tuned
  - RocksDB

---

**writes/reads (5-95% writes)**

- 
  - $10^{-2}$
  - $10^{-1}$
- Arrays
- B-trees
- Size ratio
- Merge policy
- Filters bits per entry
- Size of (page, buffer, ..)
- Internal k-v layout
- ...

(DASlab @ Harvard SEAS)
DESCRIBE ONE DATA BLOCK AT A TIME
AS A SET OF CONCEPTS
physical layout and domain partitioning

key-value
(read, bulk load)
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%)

<table>
<thead>
<tr>
<th>Key partitioning</th>
<th>Columnar</th>
<th>Row-wise</th>
<th>Row-Groups</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
<td>K</td>
<td>K</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>None</td>
</tr>
</tbody>
</table>

**Key and value layout**

### Fanout (fixed/functional | unlimited | terminal |)
### 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)
SETS OF CONCEPTS

- bloom
- filter bits
- sorted
- zone map
- link
- children
- layout
SETS OF CONCEPTS  POSSIBLE NODE DESIGNS

bloom filter bits  sorted  zone map
link
children layout

DASlab
© Harvard SEAS
SETS OF CONCEPTS  POSSIBLE NODE DESIGNS  POSSIBLE STRUCTURES

bloom filter bits  sorted  zone map

link

children layout

array  linked-list  hash-table  queue  b-tree  skip-list  binary-tree  masstree  csb-tree  fast trie
### The 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 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></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>

**Classes of Primitives**

- **DONE**: Completed
- **RUM**: Requires Update

@IEEE.EngBul18
# The Periodic Table of Data Structures

<table>
<thead>
<tr>
<th>Classes of Designs</th>
<th>Partitioning</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></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
</tr>
<tr>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
</tr>
<tr>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
</tr>
<tr>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
</tr>
<tr>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>DONE</strong></td>
<td><strong>DONE</strong></td>
</tr>
</tbody>
</table>

---

**Paper Machine**

[@IEEE.EngBul18](https://www.ieee.org)
PEOPLE ON EARTH

STARS IN THE SKY

POSSIBLE DATA STRUCTURES

$(10^9)$

$(10^{24})$

$(10^{32}, 2$-node$)$

$(10^{48}, 3$-node$)$
TIGRIS specifications are orders of magnitude shorter than workload.

We present a new programming model for container data structures, called TIGRIS, and this separation by introducing a declarative domain specific functional feature descriptions. We demonstrate the power of this separation by introducing a declarative domain specific functional feature descriptions. We demonstrate the power of this separation by introducing a declarative domain specific functional feature descriptions. We demonstrate the power of this separation by introducing a declarative domain specific functional feature descriptions.

Container data structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code. While essential, high-performance container structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code. Performance. While essential, high-performance container structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code.

Data-intensive programs, from scientific simulations to querying databases, heavily rely on high-performance container 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 be closely tuned to both the underlying hardware and those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations. Hash tables and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Large collections of base and intermediate data can be quickly answered using compact bit-vectors and secondary indexes, intermediate data structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code. Performance. While essential, high-performance container structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code.

Those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations. Hash tables and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Modern data systems use a large variety of containers 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 be closely tuned to both the underlying hardware and those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Data-intensive programs, from scientific simulations to querying databases, heavily rely on high-performance container 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 be closely tuned to both the underlying hardware and those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Large collections of base and intermediate data can be quickly answered using compact bit-vectors and secondary indexes, intermediate data structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code. Performance. While essential, high-performance container structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code.

Those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations. Hash tables and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Modern data systems use a large variety of containers 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 be closely tuned to both the underlying hardware and those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Data-intensive programs, from scientific simulations to querying databases, heavily rely on high-performance container 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 be closely tuned to both the underlying hardware and those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Large collections of base and intermediate data can be quickly answered using compact bit-vectors and secondary indexes, intermediate data structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code. Performance. While essential, high-performance container structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code.

Those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations. Hash tables and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Modern data systems use a large variety of containers 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 be closely tuned to both the underlying hardware and those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Data-intensive programs, from scientific simulations to querying databases, heavily rely on high-performance container 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 be closely tuned to both the underlying hardware and those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.

Large collections of base and intermediate data can be quickly answered using compact bit-vectors and secondary indexes, intermediate data structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code. Performance. While essential, high-performance container structures are notoriously difficult to write and typically consist of hundreds of lines of hand-tuned non-trivial code.

Those structures have been proposed, including compression, hashing, and Bloom filters. Hash tables, and ordered tree structures provide the foundation of many data caching and algorithmical optimizations. Hash tables and ordered tree structures provide the foundation of many data caching and algorithmical optimizations.
DESIGN SPACE

if we know the fundamental building blocks, how they combine and their properties,

then we can automate the discovery of novel combinations and tunings
HOW TO JUDGE A DESIGN?

1. COMPLEXITY ANALYSIS
2. IMPLEMENTATION & TESTING
3. GENERALIZED MODELS
Access path selection @SIGMOD2017

**Equation 13** is the combination of five logical parts: tree traversal, leaf traversal, result writing, sorting, and base scan. The optimizer in a primary experimental server using secondary index tuning parameters, is based on our experimental data from four machines. For each set of experiments, we used a multidimensional unconstrained nonlinear minimization technique (Nelder-Mead) to fit the model. More details about the verification process are available in Appendix C.

Let us now examine step by step the two parts of the numerator that are defined as follows:

\[
APS(q,S_{tot}) = \frac{q \cdot \frac{1}{N} \cdot \log_b(N) \cdot (BW_S \cdot CM + \frac{b \cdot BW_S \cdot CA}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2})}{\max(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}
\]

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

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

**Workload**

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

**Dataset**

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

**Hardware**

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

**Scan & Index**

<table>
<thead>
<tr>
<th>$rw$</th>
<th>result width (bytes per output tuple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>tree fanout</td>
</tr>
<tr>
<td>$aw$</td>
<td>attribute width (bytes of the indexed column)</td>
</tr>
<tr>
<td>$ow$</td>
<td>offset width (bytes of the index column offset)</td>
</tr>
</tbody>
</table>
DESIGN SPACE OF POSSIBLE STORAGE LAYOUTS
DESIGN SPACE OF POSSIBLE STORAGE LAYOUTS
ALGORITHM & COST SYNTHESIS
decide access pattern based on the data block’s physical organization.
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
Rules: access principles

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
...
Learning of fine-grained access patterns

Rules: access principles

sorted keys
columnar layout

RULES

binary search
interpolation search
using new SIMD instruction X

code, model
code, model
code, model

columnar layout

sorted search

batched write
BF probe
scan
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

e.g., binary search

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

1 11 17 37 51 66 80 94
SYNTHESES 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 vs data size](image)
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

```cpp
C++

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

2. BENCHMARK

![Time vs Data Size Graph]

3. FIT MODEL

![Log-Linear Model](f(x) = ax + b \log x + c)
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

3. FIT MODEL

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

FOLDING ALGORITHMIC, ENGINEERING, AND H/W, PROPERTIES INTO THE COEFFICIENTS
TRAINING
TRAINING

First Principles (access)

Binary Search

Scan

Hardware

Data Calculator

CPP

Probe

CPP

CPP

CPP

...
TRAINING
FOR EACH OPERATION

QUEYRYING

First Principles (access)

Learned Models

Data Calculator

Hardware Workload Specification

First Principles (layout)

Cost
**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

---

**QUERYING**

- **First Principles (access)**
- **Learned Models**
CAN WE COMPUTE PERFORMANCE ACCURATELY?

layout spec → DC → cost
VS
C++ → cost

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

CALCULATOR
IMPLEMENTATION

B+Tree

{10M (uniform) k-v pairs, 100 point queries (skewed)}
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></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B+Tree

{10M (uniform) k-v pairs, 100 point queries (skewed)}
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>0.0008</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.0006</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>0.0004</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.0002</td>
<td></td>
</tr>
</tbody>
</table>

B+Tree

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

CSB+Tree

Query Skew

{10M (uniform) k-v pairs, 100 point queries (skewed)}
DESIGN SPACE | COST SYNTHESIS

WHO AND HOW TO USE
What-if we add bloom filters in the hash-table buckets?

~20 SECONDS
(workload:10 Million entries, 100 queries)
What-if the workload changes to 90% writes?

~20 SECONDS
(workload: 10 Million entries, 100 queries)
What-if we **buy faster CPU X**?

~20 SECONDS

(workload: 10 Million entries, 100 queries)
design.continuums
UNIFY DESIGNS FOR FAST NAVIGATION
$log$
LSM-tree*

@SIGMOD17/18
The graph compares the lookup cost and update cost of different data structures: log, LSM-tree*, B-tree*, and sorted array.

- **Log**: High lookup cost and very low update cost.
- **LSM-tree**: Lower lookup cost than log and lower update cost than B-tree.
- **B-tree**: Lower lookup cost and lower update cost than LSM-tree.
- **Sorted Array**: Lowest lookup cost and update cost among the structures shown.

The graph indicates that sorted arrays are the most efficient in terms of both lookup and update costs, followed by B-trees, LSM-trees, and logs.
LSM-tree

B-tree*

sorted array

lookup cost

update cost

one model

log

@SIGMOD17/18
few continuous parameters & closed form formulas for metrics
few continuous parameters & closed form formulas for metrics

near instantaneous design space navigation
few continuous parameters & closed form formulas for metrics
near instant design space navigation

CrimsonDB
a self-designing key-value store
**design space** (updates, concurrency)

**cost synthesis** (accuracy, scalability)

**easy extensibility** (plug & play rules)

---

**calculator infrastructure**

**study structure & gaps**

**building more on top of**

**design continuums** (optimizations)

**go after gaps** (new design opport.)

**design guide** (static design rule sys)

**auto-search** (ML/algo hybrids, hints)

**self-designing** (log to sorted arrays)

**dsl & compilers** (productivity & perf)

---

daslab.seas.harvard.edu
THANKS!

calculator infrastructure
DESIGN SPACE (updates, concurrency)
COST SYNTHESIS (accuracy, scalability)
EASY EXTENSIBILITY (plug & play rules)

study structure & gaps
DESIGN CONTINUUMS (optimizations)
GO AFTER GAPS (new design opport.)
DESIGN GUIDE (static design rule sys)

building more on top of
AUTO-SEARCH (ML/algo hybrids, hints)
SELF-DESIGNING (log to sorted arrays)
DSL & COMPILERS (productivity & perf)