CS265 Research Projects

Theme: Learned Data Systems

Subarna Chatterjee
efficient storage

faster navigation
State of The Art

Suboptimal System: Static Designs for Diverse Applications
State of The Art

Suboptimal System: Static Designs for Diverse Applications

Viber migrates from MongoDB to Couchbase halves number of AWS servers

Why we choose Badger over RocksDB in Dgraph - Dgraph Blog

Why I am Tempted to Replace Cassandra With DynamoDB
Suboptimal System: Static Designs for Diverse Applications

Blow Up Cloud Costs: Pinning Navigational Data Structures In Memory

State of The Art

Viber migrates from MongoDB to Couchbase halves number of AWS servers

Why we choose Badger over RocksDB in Dgraph - Dgraph Blog

Why I am Tempted to Replace Cassandra With DynamoDB
State of The Art

**Suboptimal System:** Static Designs for Diverse Applications

**Blow Up Cloud Costs:** Pinning Navigational Data Structures In Memory

---

Viber migrates from MongoDB to Couchbase halves number of AWS servers

Why we choose Badger over RocksDB in Dgraph - Dgraph Blog

Why I am Tempted to Replace Cassandra With DynamoDB
How can we create tailored systems that scale with data?
How can we create **tailored** systems that **scale** with data?
How can we create tailored systems that scale with data?

**Cosine**: A Self-Designing Storage Engine

Tailored Design For Each Context
How can we create **tailored** systems that **scale** with data?

**Cosine**: A Self-Designing Storage Engine

**Limousine**: A Self-Designing Learned Data System

Tailored Design For Each Context

Using learned data structures for efficient navigation
Cloud-provider and hardware space

VMs of different capacity

different storage type (SSD, HDD, EBS)
Cloud-provider and hardware space

Storage-engine design space

VMs of different capacity

different storage type (SSD, HDD, EBS)
Cloud-provider and hardware space

Storage-engine design space

LSM

B-Trees

LSH

VMs of different capacity (SSD, HDD, EBS)

Different storage type (SSD, HDD, EBS)
Cloud-provider and hardware space

Storage-engine design space

Performance space

massive number of unexplored designs

write-optimized

LSM

write

memory

LSH

B-Trees

read

write-optimized

FASTER

WiredTiger

RocksDB

read

VMs of different capacity

different storage type

(SSD, HDD, EBS)

massive number of unexplored designs

WiredTiger

RocksDB

FASTER

write-optimized
Cloud-provider and hardware space

Workload space

Performance space

Storage-engine design space

Cloud-provider and hardware space

- LSM
- B-Trees
- FASTER
- RocksDB
- WiredTiger
- LSH

- empty range + insert
- non-empty range + get
- range + blind update + get
- blind update + insert
- rmw + get
- write-optimized
- read-optimized

- massive number of unexplored designs

- memory
- write
- read

- VMs of different capacity
- different storage type (SSD, HDD, EBS)
Cloud-provider and hardware space

Workload space
- rmw + get
- range + get
- non-empty range + get
- empty range + insert
- blind update + insert

Performance space
- FASTER
- WiredTiger
- write-optimized
- read-optimized

Storage-engine design space
- LSM
- B-Trees
- LSH

Exhaustive Decision Space
- 10^35 possibilities

Cloud-provider and hardware space
- VMs of different capacity
- different storage type (SSD, HDD, EBS)

Performance
- memory

[Diagram showing various storage systems and their performance levels]
workload $\rightarrow$ budget $\rightarrow$ optimal configuration
workload → Cosine → optimal configuration
$ → Cosine → SE design, h/w, cloud provider
budget
Storage Engine Template

Layout Primitives
- buffer?
- filters?
- cache?
- indexes?
- hot-cold partition?
- growth factor?
- level size?
- greediness of merge?

Algorithmic Abstractions
- restructuring strategy
- filter design
- indexing algorithm
- file picking strategy
- block
- file
- run
- full
- partial
- hybrid
- fence
- pointer
- hash
- oldest
- merged
- flushed
- oldest
- oldest
- flushed

Storage Engine Template

DAStlab
@ Harvard SEAS
<table>
<thead>
<tr>
<th>MEMORY</th>
<th>DISK</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer?</td>
<td></td>
</tr>
<tr>
<td>filters?</td>
<td></td>
</tr>
<tr>
<td>indexes?</td>
<td></td>
</tr>
<tr>
<td>growth factor?</td>
<td></td>
</tr>
<tr>
<td>greediness of merge?</td>
<td></td>
</tr>
<tr>
<td>level size?</td>
<td></td>
</tr>
<tr>
<td>hot-cold partition?</td>
<td></td>
</tr>
<tr>
<td>cache?</td>
<td></td>
</tr>
</tbody>
</table>

### Layout Primitives
- Restructuring strategy
- Indexing algorithm
- File picking strategy
- Full, partial, hybrid
- Oldest merged
- Oldest flushed

### Algorithmic Abstractions
- Storage Engine Template
- Fence pointer
- Hash
<table>
<thead>
<tr>
<th>MEMORY</th>
<th>DISK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>storage pattern?</td>
</tr>
<tr>
<td></td>
<td>growth factor?</td>
</tr>
<tr>
<td></td>
<td>level size?</td>
</tr>
<tr>
<td></td>
<td>greediness of merge?</td>
</tr>
<tr>
<td></td>
<td>file size?</td>
</tr>
<tr>
<td></td>
<td>hot-cold partition?</td>
</tr>
<tr>
<td>MEMORY</td>
<td>DISK</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td><strong>storage pattern?</strong>&lt;br&gt;{flat logs, hierarchical}</td>
</tr>
<tr>
<td></td>
<td><strong>growth factor?</strong>&lt;br&gt;[1, .., multiples of block size]</td>
</tr>
<tr>
<td></td>
<td><strong>level size?</strong>&lt;br&gt;[1, .., L]</td>
</tr>
<tr>
<td></td>
<td><strong>greediness of merge?</strong>&lt;br&gt;[1 (high), .., T (low)]</td>
</tr>
<tr>
<td></td>
<td><strong>file size?</strong>&lt;br&gt;Mb ... GB</td>
</tr>
<tr>
<td></td>
<td><strong>hot-cold partition?</strong></td>
</tr>
<tr>
<td>MEMORY</td>
<td>DISK</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>buffer?</td>
<td>storage pattern?</td>
</tr>
<tr>
<td>[1..M]</td>
<td>{flat logs, hierarchical}</td>
</tr>
<tr>
<td>filters?</td>
<td>growth factor?</td>
</tr>
<tr>
<td>Bloom? Cuckoo?</td>
<td>[1, .., multiples of block size]</td>
</tr>
<tr>
<td>indexes?</td>
<td>level size?</td>
</tr>
<tr>
<td>hash table? zone map?</td>
<td>[1, .., L]</td>
</tr>
<tr>
<td>cache?</td>
<td>greediness of merge?</td>
</tr>
<tr>
<td>[1..M]</td>
<td>[1 (high), .., T (low)]</td>
</tr>
<tr>
<td></td>
<td>file size?</td>
</tr>
<tr>
<td></td>
<td>MB ... GB</td>
</tr>
<tr>
<td></td>
<td>hot-cold partition?</td>
</tr>
</tbody>
</table>
hot-cold partition?
storage pattern?
growth factor?
level size?
greediness of merge?
thick file size?
Leveled LSM  
Tiered LSM  
BTree  
New layout

.storage pattern?  
growth factor?  
level size?  
greediness of merge?  
hot-cold partition?  
file size?
Storage Engine Template

Layout Primitives

- buffer?
- filters?
- cache?
- indexes?
- hot-cold partition?
- growth factor?
- level size?
- greediness of merge?
Storage Engine Template

Layout Primitives
- buffer?
- filters?
- cache?
- indexes?
- hot-cold partition?
- growth factor?
- level size?
- greediness of merge?

Algorithmic Abstractions
- restructuring strategy
- filter design
- indexing algorithm
- file picking strategy

Hot-cold partition:
- block
- file
- run
- full
- partial
- hybrid
- hash
- pointer
- fence
- oldest
- flushed
- merged
- oldest

Greediness of merge:
- buffer
- filters
- cache
- indexes
- hot-cold partition
- growth factor
- level size
- greediness of merge

Storage Engine Template

DASlab @ Harvard SEAS
<table>
<thead>
<tr>
<th>Algorithmic Abstractions</th>
<th>Layout Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallellism</strong></td>
<td><strong>Data access</strong></td>
</tr>
<tr>
<td>derived with empirically verified rules</td>
<td>initialized by search through engine design space</td>
</tr>
<tr>
<td>1. No. of threads: Denotes how many threads are used to process the workload.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>2. Memory size (M): Denotes how much memory is allocated to indexes (fence pointers/hashtables).</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>3. Logical block size (B): Number of consecutive disk blocks.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>4. Number of available cores to use in a VM.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>5. Rule should be less than size ratio.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>6. Merger threshold: If a level is more than x% full, a compaction is triggered.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>7. At what capacity hot levels are compacted.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>8. Merge per level (Z): At what capacity hot levels are compacted.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>9. Bloom Filter run strategy: Denotes the run which can be picked for compaction (only for partial/full compaction).</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>11. Hybrid compaction: Hybrid block-to-block and file-to-file compaction.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>12. Filter placement strategy: Denotes which level to be picked for compaction (only for partial/full compaction).</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>13. Filter placement strategy: Denotes which level to be picked for compaction (only for partial/full compaction).</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>15. Full compaction/Restructuring strategy: Full block-to-block and file-to-file compaction.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>16. Compaction/Restructuring algorithm: Full block-to-block compaction.</td>
<td>128-bit Floating Point</td>
</tr>
<tr>
<td>17. Use threads per CPU core:</td>
<td>128-bit Floating Point</td>
</tr>
</tbody>
</table>

**Example templates for diverse data structures**

**Template Variants**
- Example templates for diverse data structures
- Auto-configured from the sample workload
- Variables: Design Abstractions of Template
- Variants: WiredTiger
- Features: Faster
- Adjustments: N/A

<table>
<thead>
<tr>
<th>Type/Domain</th>
<th>Use</th>
<th>Use</th>
<th>Memory size (M)</th>
<th>Logical block size (B)</th>
<th>No. of CPUs</th>
<th>No. of threads</th>
<th>Merge per cold level (Z)</th>
<th>Size ratio (T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSM trees</td>
<td>0.5</td>
<td>0.5</td>
<td>128-bit Floating Point</td>
<td>128-bit Floating Point</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B-trees</td>
<td>0.5</td>
<td>0.5</td>
<td>128-bit Floating Point</td>
<td>128-bit Floating Point</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Hash tables</td>
<td>0.5</td>
<td>0.5</td>
<td>128-bit Floating Point</td>
<td>128-bit Floating Point</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Evaluate and rank configurations

Cost
Latency

Cloud Provider and Hardware Space

Optimized configuration for a single provider

Pick optimal

Navigation through search space

Fault tolerance
Reliability

DB migration
Backup

Search Algorithms

Latency
Cost
Pareto frontier
Cost-Performance Continuum

Repeat for all configurations

Storage Engine Design Space

Co-optimize hardware and SLA

Search space of configurations

KV OPERATIONS (get, put, range, update, rmw)

Memory (layout primitive)
Buffer
Fences
Filters
LSM-Tree
B+-Tree

New Design

On-Disk Run (layout primitive)
LSM Run
B-tree Run
Fanout
Cascading
Fence Pointers
Restructuring Module (algorithm)

hybrid
full
partial

Cosine Engine
LSHTable

Shape of SE
Per operation IO cost

Empirically learn proportion of parallelizable component

Get speedup coefficient
Get actual end-to-end performance

Concurrency-Aware CPU Model

Amdahl’s Law
KV OPERATIONS (get, put, range, update, rmw)

Memory (layout primitive)
- Buffer
- Fence
- Filters
- LSM-Tree
- B+-Tree

On-Disk Run (layout primitive)
- LSM Run
- B-tree Run
- Fanout
- Cascading
- Fence Pointers

Restructuring Module (algorithm)
- Hybrid
- Full
- Partial

Cosine Engine
- LSHTable

Layout Primitives
- Buffer?
- Filters?
- Cache?
- Indexes?
- Hot-cold partition?
- Level size?
- Greediness of merge?

Algorithmic Abstractions
- Restructuring strategy
- Indexing algorithm
- File picking strategy

Storage Engine Template

Distribution-Aware I/O Model
- Shape of SE
- Per operation IO cost

Concurreny-Aware CPU Model
- Empirically learn proportion of parallelizable component
- Get speedup coefficient
- Get actual end-to-end performance

Empirically learn proportion of parallelizable component

Concurreny-Aware CPU Model
- Empirically learn proportion of parallelizable component
- Get speedup coefficient
- Get actual end-to-end performance

Distribution-Aware I/O Model
- Shape of SE
- Per operation IO cost

Concurreny-Aware CPU Model
- Empirically learn proportion of parallelizable component
- Get speedup coefficient
- Get actual end-to-end performance

Distribution-Aware I/O Model
- Shape of SE
- Per operation IO cost

Concurreny-Aware CPU Model
- Empirically learn proportion of parallelizable component
- Get speedup coefficient
- Get actual end-to-end performance

Distribution-Aware I/O Model
- Shape of SE
- Per operation IO cost

Concurreny-Aware CPU Model
- Empirically learn proportion of parallelizable component
- Get speedup coefficient
- Get actual end-to-end performance
KV OPERATIONS (get, put, range, update, rmw)

Memory (layout primitive)
- Buffer
- Fences
- Filters
- LSM-Tree
- B+-Tree
- New Design...

On-Disk Run (layout primitive)
- LSM Run
- B-tree Run
- Fanout
- Cascading
- Fence Pointers
- Restructuring Module (algorithm)
  - hybrid
  - full
  - partial

Cosine Engine
- LSHTable
- Evaluate and rank configurations
- Cloud Provider and Hardware Space
  - Cost
  - Latency
  - optimized configuration for a single provider

Search Algorithms

Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Navigate through search space
- Repeat for all configurations
  - Co-optimize hardware and SLA
    - reliability
    - fault tolerance
    - backup
    - DB migration
  - Evaluate and rank configurations
  - Pick optimal

Cost-Performance Continuum
- Pareto frontier

Storage Engine Template
- Layout Primitives
  - buffer?
  - filters?
  - cache?
  - hot-cold partition?
  - growth factor?
  - level size?
  - greediness of merge?

Algorithmic Abstractions
- Filter design
- Restructuring strategy
- Indexing strategy
- File picking strategy
  - full
  - partial
  - hybrid

Concurrency-Aware CPU Model
- Empirically learn proportion of parallelizable component
- Get speedup coefficient
- Get actual end-to-end performance

Distribution-Aware I/O Model
- Shape of SE
- Per operation IO cost
- Probabilistic existence in disk
  - key1
  - L1
  - L2
  - L3
  - L4
  - L5

Storage Engine Template
- Buffer
- Fences
- Filters
- LSM-Tree
- B+-Tree
- New Design...

On-Disk Run (layout primitive)
- LSM Run
- B-tree Run
- Fanout
- Cascading
- Fence Pointers
- Restructuring Module (algorithm)
  - hybrid
  - full
  - partial

Cosine Engine
- LSHTable
- Evaluate and rank configurations
- Cloud Provider and Hardware Space
  - Cost
  - Latency
  - optimized configuration for a single provider

Search Algorithms

Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Navigate through search space
- Repeat for all configurations
  - Co-optimize hardware and SLA
    - reliability
    - fault tolerance
    - backup
    - DB migration
  - Evaluate and rank configurations
  - Pick optimal

Cost-Performance Continuum
- Pareto frontier

Storage Engine Template
- Layout Primitives
  - buffer?
  - filters?
  - cache?
  - hot-cold partition?
  - growth factor?
  - level size?
  - greediness of merge?

Algorithmic Abstractions
- Filter design
- Restructuring strategy
- Indexing strategy
- File picking strategy
  - full
  - partial
  - hybrid

Concurrency-Aware CPU Model
- Empirically learn proportion of parallelizable component
- Get speedup coefficient
- Get actual end-to-end performance

Distribution-Aware I/O Model
- Shape of SE
- Per operation IO cost
- Probabilistic existence in disk
  - key1
  - L1
  - L2
  - L3
  - L4
  - L5

Storage Engine Template
- Buffer
- Fences
- Filters
- LSM-Tree
- B+-Tree
- New Design...

On-Disk Run (layout primitive)
- LSM Run
- B-tree Run
- Fanout
- Cascading
- Fence Pointers
- Restructuring Module (algorithm)
  - hybrid
  - full
  - partial

Cosine Engine
- LSHTable
- Evaluate and rank configurations
- Cloud Provider and Hardware Space
  - Cost
  - Latency
  - optimized configuration for a single provider

Search Algorithms

Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Navigate through search space
- Repeat for all configurations
  - Co-optimize hardware and SLA
    - reliability
    - fault tolerance
    - backup
    - DB migration
  - Evaluate and rank configurations
  - Pick optimal

Cost-Performance Continuum
- Pareto frontier

Storage Engine Template
- Layout Primitives
  - buffer?
  - filters?
  - cache?
  - hot-cold partition?
  - growth factor?
  - level size?
  - greediness of merge?

Algorithmic Abstractions
- Filter design
- Restructuring strategy
- Indexing strategy
- File picking strategy
  - full
  - partial
  - hybrid

Concurrency-Aware CPU Model
- Empirically learn proportion of parallelizable component
- Get speedup coefficient
- Get actual end-to-end performance

Distribution-Aware I/O Model
- Shape of SE
- Per operation IO cost
- Probabilistic existence in disk
  - key1
  - L1
  - L2
  - L3
  - L4
  - L5
KV OPERATIONS
(get, put, range, update, rmw)

Memory (layout primitive)
Buffer
Fences
Filters
LSM-Tree
B+-Tree
New Design...

On-Disk Run (layout primitive)
LSM Run
B-tree Run
Fanout
Cascading
Fence Pointers
Restructuring Module (algorithm)
hybrid
full
partial
Cosine Engine
LSHTable

Evaluate and rank configurations
Cloud Provider and Hardware Space
Cost
Latency
optimized configuration for a single provider

Storage Engine and Algorithmic Abstractions

Distribution-Aware I/O Model
Shape of SE
Per operation IO cost

Concurrency-Aware CPU Model
Empirically learn proportion of parallelizable component
Get speedup coefficient
Get actual end-to-end performance

Concurrency-Aware CPU Model
Empirically learn proportion of parallelizable component
Get speedup coefficient
Get actual end-to-end performance

Search Algorithms
Search space of configurations
Pareto frontier

Navigation through search space
Co-optimize hardware and SLA

Evaluate and rank configurations
Pick optimal

Repeat for all configurations

Cost-Performance Continuum
Latency

Cost

Pareto frontier

Optimized configuration for a single provider
Empirically learn proportion of parallelizable component

Get speedup coefficient

Amdahl’s Law

Concurrency-Aware CPU Model

Probabilistic existence in disk

L1

L2

L3

L4

L5

Shape of SE

Per operation IO cost

Concurrency-Aware Disk Model

Co-optimize hardware and SLA

Cloud Provider and Hardware Space

Storage Engine Design Space

Search space of configurations

Search Algorithms

Cloud Provider and Hardware Space

Storage Engine Design Space

Cost-Performance Continuum

B-tree Run

LSM Run

Fate

Cascading

Fence Pointers

Restructuring Module (algorithm)

filter design

filter

restructing strategy

indexing algorithm

file picking strategy

Algorithmic Abstractions

Layout Primitives

buffer?

cache?

hot-cold partition?

level size?

greediness of merge?

indexes?

growth factor?

Cosine Engine

Memory (layout primitive)

Cloud Provider and Hardware Space

Storage Engine Design Space

Search Algorithms

Cloud Provider and Hardware Space

Storage Engine Design Space

Cost-Performance Continuum

B-tree Run

LSM Run

Fate

Cascading

Fence Pointers

Restructuring Module (algorithm)

full partial hybrid

KV OPERATIONS (get, put, range, update, rmw)
Empirically learn proportion of parallelizable component.

Get speedup coefficient.

Amdahl's Law.

Get actual end-to-end performance.

Concurrency-Aware CPU Model:
- Empirically learn proportion of parallelizable component.
- Get speedup coefficient.
- Get actual end-to-end performance.

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost
- probabilistic existence in disk

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost
- probabilistic existence in disk

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost

Distribution-Aware I/O Model:
- Shape of SE
- Per operation IO cost

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Search Algorithms:
- Search space of configurations
- Cloud Provider and Hardware Space
- Storage Engine Design Space

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Storage Engine Template:
- Layout Primitives
- Algorithmic Abstractions

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)

Cosine Engine:
- Memory (layout primitive)
YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)
YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)
Throughput (kops/s)

YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)

Throughput (kops/s)

WiredTiger

RocksDB

FASTER (no support for range queries)

Budget ($/month)

20K 60K 100K
YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)

Throughput (kops/s)

Existing Systems

- - - RocksDB
- - - WiredTiger
- - - FASTER

FASTER (no support for range queries)

20K 60K 100K
Budget ($/month)
YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)

Throughput (kops/s) vs. Budget ($/month)

- RocksDB
- WiredTiger
- FASTER (no support for range queries)

Existing Systems
YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)

Throughput (kops/s)

<table>
<thead>
<tr>
<th>Budget ($/month)</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>20K</td>
<td>Cosine</td>
</tr>
<tr>
<td>60K</td>
<td>WiredTiger</td>
</tr>
<tr>
<td>100K</td>
<td>FASTER</td>
</tr>
</tbody>
</table>

Existing Systems

- RocksDB
- WiredTiger
- FASTER (no support for range queries)
YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)

Throughput (kops/s)

- RocksDB
- WiredTiger
- FASTER (no support for range queries)

Existing Systems

- Leveled LSM + AWS + 128 GB mem + 8 CPUs
- Btree + Azure + 64 GB mem + 4 CPUs
- Hybrid + GCP + 128 GB mem + 4 CPUs
- Cosine
- WiredTiger
- FASTER

Budget ($/month)

20K 60K 100K
YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)

Existing Systems

- RocksDB
- WiredTiger
- FASTER

Leveled LSM + AWS + 128 GB mem + 8 CPUs
Btree + Azure + 64 GB mem + 4 CPUs
Hybrid + GCP + 128 GB mem + 4 CPUs

FASTER (no support for range queries)

Throughput (kops/s)

Budget ($/month)

size ratio (T): 12
hot merge thresh. (K): 4
cold merge thresh. (Z): 9
tot mem (M): 128 GB
#parallelism: 8
VM: r5d.4xlarge
Cloud: AWS

size ratio (T): 15
hot merge thresh. (K): 1
cold merge thresh. (Z): 8
tot mem (M): 64 GB
#parallelism: 4
VM: n1-highmem-16
Cloud: GCP
YCSB E variant (30% blind update, 20% non-empty range, 50% empty range)

Throughput (kops/s)

- RocksDB
- WiredTiger
- FASTER (no support for range queries)

Existing Systems

- AWS
- GCP
- Azure

Cosine

Budget ($/month)

20K  60K  100K
Can we do better?

Let’s dive deeper!

Mixed workload:
50-50 lookups-inserts
Analyzing IO Cost in Cosine

![Bar chart showing total IOs (10^6) vs. budget ($/month) for Hybrid, Btree, LSMs, and LSH.](chart.png)
Analyzing IO Cost in Cosine Btree, Hybrid, LSMs, LSH

Budget ($/month)

Total IOs (10^6)

IOs (navigation) IOs (data)
Navigational structures can have up to 50% IO overhead.
Navigational structures can have up to 50% IO overhead.

The overhead is maximum for designs involving Btrees - classical Btrees + hybrid designs.
Analyzing IO Cost in Cosine

Storing navigational structures on cloud can be expensive!!
Our intuition: We cannot skip reading base data, but we can skip reading navigation data if we navigate differently.
Brief Introduction to Learned Indexes
sorted array + binary search

sorted array + B tree index
sorted array + binary search

sorted array + B tree index
sorted array
+ binary search

position
key

$y = f(x) = \left(\frac{x}{2}\right) - 1$

B tree index

sorted array + binary search

sorted array + B tree index

\( y = f(x) = \left(\frac{x}{2}\right) - 1 \)

position key

model
Piecewise Linear Approximation (PLA)
Piecewise Linear Approximation (PLA)

\[ f_3(\text{key}) = [\text{pos} + k, \text{pos} - k] \]
Piecewise Linear Approximation (PLA)
Piecewise Linear Approximation (PLA)
Piecewise Linear Approximation (PLA)

$\text{CDF}$

$\text{key}$

$\text{PLA segments}$ (models)

$\text{binary search on start of segments}$
Piecewise Linear Approximation (PLA)

CDF

key

f1 f2 f3 f4 PLA segments (models)

f1 f2 f3 f4 PLA segments

+ classical index

binary search on start of segments

FIT-ing Tree, SIGMOD 2019
Radix Spline, aIDM 2020
Piecewise Linear Approximation (PLA)

CDF

key

f1 f2 f3 f4 PLA segments (models)

CDF

key

f1 f2 f3 f4 PLA segments

binary search on start of segments

PLA segments

models

+ classical index

PLA segments

+ recursive model index

FIT-ing Tree, Radix Spline, PGM, SIGMOD 2019, aiDM 2020, VLDB 2020
#Btree-nodes = 10 X #models (indexing the same data)

#Btree-nodes = 80 X #models (indexing the same data)
The memory footprint of Btrees can be up to two orders of magnitude higher than that of a learned index.
Can we self-design navigational structures intelligently so that the resulting SE achieves a better cost-performance balance for large data?
Can we self-design navigational structures intelligently so that the resulting SE achieves a better cost-performance balance for large data?
LIMOUSINE

Blending Learned and Traditional Indexes to Self-Design Larger-than-Memory Cloud Storage Engines
Cosine’s Design Space

Log-Structured Hash Tables

B-Trees

Log-Structured Merge Trees
Cosine’s Design Space

- Log-Structured Hash Tables
- B-Trees
- Log-Structured Merge Trees
Cosine’s Design Space

- Log-Structured Hash Tables
- Log-Structured Merge Trees
- B-Trees

Learned Data Structures
Cosine’s Design Space

- Log-Structured Hash Tables
- Log-Structured Merge Trees
- B-Trees

Learned Data Structures

- FIT-ing Trees
- PGM
- Radix Spline
Cosine’s Design Space

- Log-Structured Hash Tables
- B-Trees
- Log-Structured Merge Trees

Learned Data Structures

- FIT-ing Trees
- PGM
- Radix Spline

hybrid designs
Cosine’s Design Space

Log-Structured
Hash Tables

Log-Structured
Merge Trees

B-Trees

FIT-ing
Trees

PGM

Radix
Spline

Learned Data Structures
Cosine’s Design Space

Log-Structured Merge Trees

Log-Structured Hash Tables

B-Trees

FIT-ing Trees

PGM

Radix Spline

Learned Data Structures

Write cost

Read cost
Log-Structured Merge Trees

Log-Structured Hash Tables

FIT-ing Trees

PGM

Radix Spline

read cost

write cost

navigable

searchable

Pareto frontier
IO Model for Learned Data Layouts

Corollary 1. Under the assumptions of Theorem 1, the expected number of keys covered by a segment is:

- \( \frac{3(a+b)}{(b-a)^2} e^2 \) if the gaps are iid and uniformly distributed with minimum \( a \) and maximum \( b \).
- \( \alpha (\alpha - 2) e^2 \) if the gaps are iid and Pareto (power law) distributed with minimum value \( k > 0 \) and shape parameter \( \alpha > 2 \).
- \( e^2/(e\sigma^2 - 1) \) if the gaps are iid and lognormally distributed with mean \( \mu \) and variance \( \sigma^2 \).
- \( e^2 \) if the gaps are iid and exponentially distributed with rate \( \lambda > 0 \).
- \( ke^2 \) if the gaps are iid and gamma distributed with shape parameter \( k > 0 \) and scale parameter \( \theta > 0 \).

Read Cost

Corollary 1. Under the assumptions of Theorem 1, the expected number of keys covered by a segment is:

- \( \frac{3(a+b)^2}{(b-a)^2} \epsilon^2 \) if the gaps are iid and uniformly distributed with minimum \( a \) and maximum \( b \).
- \( \alpha (\alpha - 2) \epsilon^2 \) if the gaps are iid and Pareto (power law) distributed with minimum value \( k > 0 \) and shape parameter \( \alpha > 2 \).
- \( \epsilon^2/(e\sigma^2 - 1) \) if the gaps are iid and lognormally distributed with mean \( \mu \) and variance \( \sigma^2 \).
- \( \epsilon^2 \) if the gaps are iid and exponentially distributed with rate \( \lambda > 0 \).
- \( ke^2 \) if the gaps are iid and gamma distributed with shape parameter \( k > 0 \) and scale parameter \( \theta > 0 \).

Read Cost

\[
\frac{n}{3(a+b)^2} \epsilon^2
\]

#segments in last level:
\( (S_L) \)

Corollary 1. Under the assumptions of Theorem 1, the expected number of keys covered by a segment is:

- $3 \frac{(a+b)^2}{(b-a)^2} \epsilon^2$ if the gaps are iid and uniformly distributed with minimum $a$ and maximum $b$.
- $\alpha (\alpha - 2) \epsilon^2$ if the gaps are iid and Pareto (power law) distributed with minimum value $k > 0$ and shape parameter $\alpha > 2$.
- $\epsilon^2/(e^{\sigma^2} - 1)$ if the gaps are iid and lognormally distributed with mean $\mu$ and variance $\sigma^2$.
- $\epsilon^2$ if the gaps are iid and exponentially distributed with rate $\lambda > 0$.
- $ke^2$ if the gaps are iid and gamma distributed with shape parameter $k > 0$ and scale parameter $\theta > 0$.

Read Cost

<table>
<thead>
<tr>
<th>Segment Level</th>
<th>Formula</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Level</td>
<td>$\frac{S_L}{B}$</td>
<td>$3 \frac{(a'+b')^2}{(b'-a')^2} \epsilon^2$</td>
</tr>
<tr>
<td>Second Last</td>
<td>$\frac{n}{3 \frac{(a+b)^2}{(b-a)^2} \epsilon^2}$</td>
<td>$3 \frac{(a+b)^2}{(b-a)^2} \epsilon^2$</td>
</tr>
<tr>
<td>Next Last</td>
<td>$\frac{n}{3 \frac{(a+b)^2}{(b-a)^2} \epsilon^2}$</td>
<td>$3 \frac{(a+b)^2}{(b-a)^2} \epsilon^2$</td>
</tr>
<tr>
<td>First Last</td>
<td>$\frac{n}{3 \frac{(a+b)^2}{(b-a)^2} \epsilon^2}$</td>
<td>$3 \frac{(a+b)^2}{(b-a)^2} \epsilon^2$</td>
</tr>
</tbody>
</table>

IO Model for Learned Data Layouts

Write Cost For Last Level

#segments at time t: 
\[ \frac{n}{3 \left( \frac{(a+b)^2}{(b-a)^2} \epsilon^2 \right)} \]

#inserts after which retraining is needed: 
\[ 3 \left( \frac{a+b}{b-a} \right)^2 \epsilon^2 \]

data movement cost for each training phase (D): 
\[ \frac{n}{B} \]

cost of 1 insert: 
\[ \frac{D}{3 \left( \frac{(a+b)^2}{(b-a)^2} \epsilon^2 \right)} \]
Project 1

Out-of-Place Writes for Learned Indexes
in-place insert in a learned index

insert(key)

sorted
in-place insert in a learned index
in-place insert in a learned index
in-place insert in a learned index

diagram showing sorted in-place insert
in-place insert in a learned index

![Diagram showing in-place insert and repeated retraining]

- repeated retraining and reconstruction of index
- reduced accuracy on lazy reconstruction
out-of-place insert in a learned index
out-of-place insert: key idea
out-of-place insert: key idea
out-of-place insert: key idea
out-of-place insert: key idea
out-of-place insert: key idea
Task 1: Design and implementation of the buffer data structure

Task 2: Design and implementation of the lazy merge algorithm

Task 3: Benchmarking and evaluation
Project Workflow

☑ Task 1: Design and implementation of the buffer data structure

☑ Task 2: Design and implementation of the lazy merge algorithm

☑ Task 3: Benchmarking and evaluation

arrays
skiplist
hash table
Project Workflow

☑ Task 1: Design and implementation of the buffer data structure

☑ Task 2: Design and implementation of the lazy merge algorithm

☑ Task 3: Benchmarking and evaluation

- arrays
- skiplist
- hash table
Project Workflow

☑ Task 1: Design and implementation of the buffer data structure

☑ Task 2: Design and implementation of the lazy merge algorithm

☑ Task 3: Benchmarking and evaluation

- arrays
- skiplist
- hash table
- trigger
- granularity
- background/foreground
Project Workflow

☑ Task 1: Design and implementation of the buffer data structure

☑ Task 2: Design and implementation of the lazy merge algorithm

☑ Task 3: Benchmarking and evaluation

- arrays
- skiplist
- hash table
- trigger
- granularity
- background/foreground
- integration with existing learned indexes
- comparing with in-place inserts
What is success?

- tradeoff between memory footprint and runtime - can we win in either?
- synthesize decision logic about when to use in-place and when to use out-of-place insert algorithms
Project 2

Btrees vs Learned Index: Bringing the Best of Both Worlds to Create Hybrid Indexes
Hybrid Indexes: Motivation

![Diagram showing the relationship between write cost, read cost, memory footprint, and hybrid index types (Learned, Btree)]
Hybrid Indexes: Motivation

Can we tap into this space?
Hybrid Indexes: Key Idea

learned models

Btree nodes
Hybrid Indexes: Key Idea

- separate design space for each level
- learned models
- Btree nodes
Hybrid Indexes: Key Idea

- Minimize memory footprint
- Optimize read-write tradeoff
Project Workflow

✅ Task 1: Mathematical formalization

✅ Task 2: Extending Cosine’s cost models

✅ Task 3: Extending Cosine’s search
Project Workflow

✔ Task 1: Mathematical formalization

✔ Task 2: Extending Cosine’s cost models

✔ Task 3: Extending Cosine’s search

flow

generate the optimal combination of designs

design space

optimization problem

new design primitive

new design primitive

layer-by-layer
Project Workflow

- Task 1: Mathematical formalization
- Task 2: Extending Cosine’s cost models
- Task 3: Extending Cosine’s search
Project Workflow

Task 1: Mathematical formalization

Task 2: Extending Cosine’s cost models

Task 3: Extending Cosine’s search

- Design space
- Optimization problem
- New design primitive
- Costing designs layer-by-layer
Project Workflow

☑ Task 1: Mathematical formalization

☑ Task 2: Extending Cosine’s cost models

☑ Task 3: Extending Cosine’s search

- design space
  - optimization problem
  - new design primitive
  - costing designs layer-by-layer
  - new design primitive
  - generate the optimal combination of designs
What is success?

- If hybrid designs show up on the Pareto frontier

- Can we identify the class of workloads that benefit from hybrid designs?
Skills Required for The Projects
Skills Required for The Projects

- Programming in C/C++
- Familiarization with benchmarking
- Familiarization with system profiling
- Reasoning with probability theory, classical distribution functions