Just-in-Time Data Structures

Oliver Kennedy & Lukasz Ziarek
SUNY Buffalo
What is best in life?
What is best in life?
(for organizing your data)
Storing & Organizing Data

API: Insert & Range Scan
Storing & Organizing Data

API: Insert & Range Scan

BTree

Heap

Sorted Array

Which should you use?
You guessed wrong.

(Unless you asked me what the workload was)
Each data structure makes a fixed set of tradeoffs
Workloads

Which structure is best can even change at runtime
Which structure is best can even change at runtime.
Traditional Data Structures

Physical Layout & Logic

Manipulation Logic  Access Logic
Just-in-Time Data Structures

Physical Layout & Logic

Abstraction Layer

Manipulation Logic  Access Logic
Abstractions
Abstractions

Black Box
Abstractions

My Data

Black Box

(A set of integer records)
Insertions

Let’s say I want to add a 3?

My Data

Black Box

3
Insertions

Let’s say I want to add a 3?

My Data → U

3

Black Box
Insertions

Let’s say I want to add a 3?

My Data → U

Black Box

This is **correct**, but probably **not efficient**
Insertions

Insertion creates a temporary representation…
Insertions
Insertions

... that we can eventually **rewrite** into a form that is correct and **efficient**

(once we know what ‘efficient’ means)
Building Blocks

Array (Unsorted)

Array (Sorted)

Concatenate

BTree Node
BTree Insertions

Let’s try something more complex: A BTree
BTree Insertions

A rewrite pushes the inserted object down into the tree
BTree Insertions

A rewrite pushes the inserted object down into the tree
BTree Insertions

The rewrites are **local**.
The rest of the data structure doesn’t matter!
Synergy

Let's form proactive synergy restructuring teams.
Hybrid Insertions
Hybrid Insertions

BTree Rewrite
Hybrid Insertions

BTree Rewrite

SArray Rewrite
Synergy

BTree Rewrite

BTree Leaf Rewrite
Which rewrite gets used depends on workload-specific policies.
Experiments

Cracker Index vs Adaptive Merge Tree vs JITDs

**API**
- RangeScan(low, high)
- Insert(Array)

**Gimmick**
- Insert is Free.
- RangeScan uses work done to answer the query to also organize the data.
Experiments

Cracker Index vs Adaptive Merge Tree vs JITDs

Less organization per-read
More organization per-read
100 M records (1.6 GB)

10,000 reads for 2-3 k records each

10M additional records written after 5,000 reads
Cracker Index

Time (s)

Iteration

Reads

Cracker Index

Time (s)

Iteration

Reads

Adaptive Merge Tree

Slow Convergence

Super-High Initial Costs

Bimodal Distribution

33s (not shown)
Policy 1: Swap  (Crack for 2k reads after write, then merge)
Policy 1: Swap   (Crack for 2k reads after write, then merge)

Switchover from Crack to Merge
Policy 1: Swap  (Crack for 2k reads after write, then merge)

Synergy from Cracking (lower upfront cost)
Policy 2: Transition (Gradient from Crack to Merge at 1k)
Policy 2: Transition (Gradient from Crack to Merge at 1k)

Gradient Period (% chance of Crack or Merge)
Policy 2: Transition (Gradient from Crack to Merge at 1k)

Tri-modal distribution: Cracking and Merging on a per-operation basis
• Separate **logic** and **structure/semantics**

• Composable Building Blocks

• Local Rewrite Rules

• Result: Flexible, hybrid data structures.

• Result: Graceful transitions between different behaviors.

• [https://github.com/okennedy/jitd](https://github.com/okennedy/jitd)

**Questions?**
Bonus Slides
Overall Throughput

Throughput (ops/s)

Iteration

Cracking

Swap

Merge

Transition
JITDs allow fine-grained control over DS behavior