Just-in-Time Data Structures

Languages and Runtimes for Big Data
Updates

- Slack Channel
  - #cse662-fall2017 @ http://ubodin.slack.com

- Reading for Monday: MCDB
  - Exactly one piece of feedback (see next slide)
Don’t parrot the paper back

• Find something that the paper says is good and figure out a set of circumstances where it's bad.

• What else does something similar, why is the paper better, and under what circumstances?

• Think of circumstances and real-world settings where the proposed system is good.

• Evaluation: How would you evaluate their solution in a way that they didn’t.
What is best in life?
(for organizing your data)
Storing & Organizing Data

Which should you use?

... and many more.

Which should you use?
You guessed wrong.

(Unless you didn’t)
Which data structure is best depends on the workload.

- Sorted Array
- BTree
- Heap

Each data structure makes a fixed set of tradeoffs.
Which structure is best can even change at runtime.
Workloads

We want to gracefully transition between different DSes

Current Workload
- Many Reads
- Some Writes
- No Reads
- Many Reads
Traditional Data Structures

Physical Layout & Logic

Manipulation Logic  Access Logic
Just-in-Time Data Structures

Physical Layout & Logic

Abstraction Layer

Manipulation Logic

Access Logic
Picking The Right Abstraction

Accessing and Manipulating a JITD

Case Study: Adaptive Indexes

Experimental Results

Demo
Abstractions

My Data

Black Box

(A set of integer records)
Insertions

Let's say I want to add a 3?

This is correct, but probably not efficient
Insertions

Insertion creates a temporary representation...
Insertions

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

(once we know what ‘efficient’ means)
Traditional Data Structure Design

Binary Tree

Inner Nodes

Leaf Nodes

( Maybe In a Linked List)
Traditional Data Structure Design

- Binary Tree
- Heap
- Contiguous Array of Records
- Sorted Array
Building Blocks

Structural Properties

Concatenate

Array (Unsorted)

Semantic Properties

BinTree Node

Array (Sorted)
Picking The Right Abstraction

- Accessing and Manipulating a JITD

Case Study: Adaptive Indexes

Experimental Results

Demo
Binary Tree Insertions

Let’s try something more complex: A Binary Tree

...
Binary Tree Insertions

A rewrite pushes the inserted object down into the tree
Binary Tree Insertions

The rewrites are \textit{local}.
The rest of the data structure doesn’t matter!
Binary Tree Insertions

Terminate recursion at the leaves
Range Scan(low, high)

[Recur into A]  
UNION [Recur into B]

IF(sep > high)  
{ [Recur into A]  }
ELSIF(sep ≤ low)  
{ [Recur into B]  }
ELSE  
{ [Recur into A]
   UNION [Recur into B]  }

Full Scan

2x Binary Search
Synergy

LET'S FORM PROACTIVE SYNERGY RESTRUCTURING TEAMS.
Hybrid Insertions
Hybrid Insertions

BinTree
Rewrite
Hybrid Insertions

Binary Tree Rewrite

Sorted Array Rewrite
Synergy

Binary Tree Rewrite

Which rewrite gets used depends on workload-specific policies.
Picking The Right Abstraction
Accessing and Manipulating a JITD

- Case Study: Adaptive Indexes

Experimental Results

Demo
Adaptive Indexes

Your Index

Your Workload
Adaptive Indexes

Your Index

Your Workload

Time
Adaptive Indexes

Your Index

Your Workload

↓ Time
Range-Scan Adaptive Indexes

Start with an Unsorted List of Records

Converge to a Binary Tree or Sorted Array

• Cracker Index
  • Converge by emulating quick-sort

• Adaptive Merge Trees
  • Converge by emulating merge-sort
Cracker Indexes

Read [2, 4)
Cracker Indexes

Radix Partition on Query Boundaries (Don’t Sort)
Cracker Indexes

Each query does less and less work
Rewrite-Based Cracking

Read [2,4)
Rewrite-Based Cracking

In-Place Sort as Before
Rewrite-Based Cracking

Fragment and Organize
Rewrite-Based Cracking

Continue fragmenting as queries arrive.
(Can use Splay Tree For Balance)
Adaptive Merge Trees

Before the first query, partition data...
Adaptive Merge Trees

...and build fixed-size sorted runs
Adaptive Merge Trees

Read [2,4)

Merge only relevant records into target array
Adaptive Merge Trees

Merge only relevant records into target array
Adaptive Merge Trees

Continue merging as new queries arrive

Read [1,3)
Rewrite-Based Merging
Adaptive Merge Trees

Rewrite any unsorted array into a union of sorted runs
Adaptive Merge Trees

Method 1: Merge Relevant Records into LHS Run
(Sub-Partition LHS Runs to Keep Merges Fast)

Read [2,4)
Adaptive Merge Trees

or...
Adaptive Merge Trees

Method 2: Partition Records into High/Mid/Low (Union Back High & Low Records)

Read [2,4)
Synergy

• Cracking creates smaller unsorted arrays, so fewer runs are needed for adaptive merge

• Sorted arrays don’t need to be cracked!

• Insertions naturally transformed into sorted runs.

• (not shown) Partial crack transform pushes newly inserted arrays down through merge tree.
Picking The Right Abstraction

Accessing and Manipulating a JITD

Case Study: Adaptive Indexes

→ Experimental Results

Demo
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 Less organization per-read

Adaptive Merge Tree vs More organization per-read

vs JITDs
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

- Slow Convergence
- Super-High Initial Costs
- Bimodal Distribution

Adaptive Merge Tree

- Time (s)
- Iteration
- Reads

Bimodal Distribution

Initial Costs

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>33s</td>
<td>(not shown)</td>
</tr>
</tbody>
</table>
Policy 1: Swap  (Crack for 2k reads after write, then merge)
Policy 1: Swap  (Crack for 2k reads after write, then 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
Overall Throughput

![Graph showing throughput over iterations for different operations: Cracking, Merge, Swap, Transition.]

JITDs allow fine-grained control over DS behavior.
Just-in-Time Data Structures

- 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/UBOdin/jitd](https://github.com/UBOdin/jitd)

**Questions?**