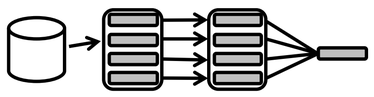
Persistent Adaptive Radix Trees: Efficient Fine-Grained Updates to Immutable Data

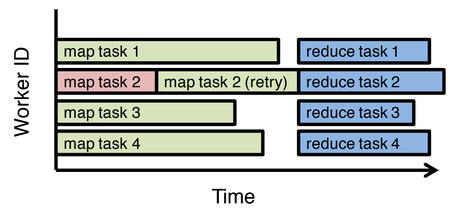
Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

MOTIVATION

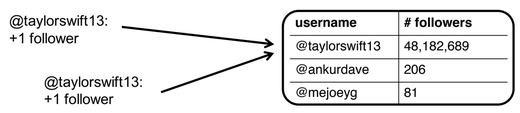
- Spark's core abstraction is bulk transformations of *immutable* datasets (RDDs)



- For batch analytics, immutability enables automatic mid-query fault tolerance and straggler mitigation



- But applications like streaming aggregation depend on efficient fine-grained updates



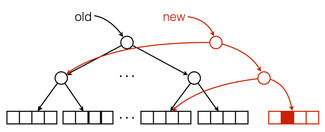
- Existing solutions involving mutable state sacrifice the advantages of Spark

1. Direct mutation: task failures corrupt state
2. Atomic batched database updates: incurs high overhead for long-lived snapshots
3. Full copy: wasteful for small updates

- Instead, we propose PART, a data structure that enables *efficient updates without sacrificing immutable semantics*

PERSISTENT DATA STRUCTURES

- Enable efficient updates without modifying the existing version in any way
- *Immutable semantics*: Updates return a new copy of the data that internally shares structure with existing copy (similar to copy-on-write snapshot)
- Nodes from old versions are GC'd



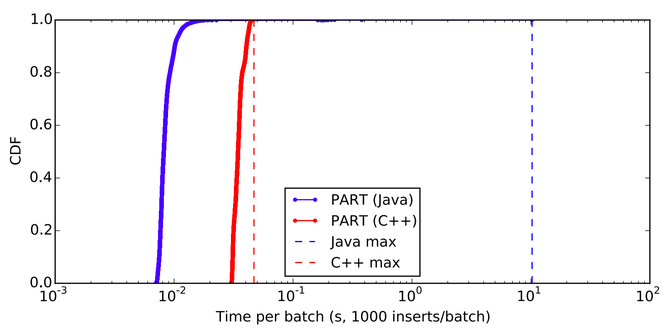
ADAPTIVE RADIX TREE

- 256-ary radix tree (trie) with node compression
- Radix trees provide:
  1. Sorted order traversals (unlike hash tables)
  2. Better asymptotic performance than binary search trees for long keys (O(k log n) vs O(k))
  3. Very efficient union and intersection operations
  4. Predictable performance: no rehashing or rebalancing
- Handles sparsity using node compression

V. Leis, A. Kemper, and T. Neumann. *The adaptive radix tree: ARTful indexing for main-memory databases*. In ICDE 2013.

PERSISTENT ADAPTIVE RADIX TREE

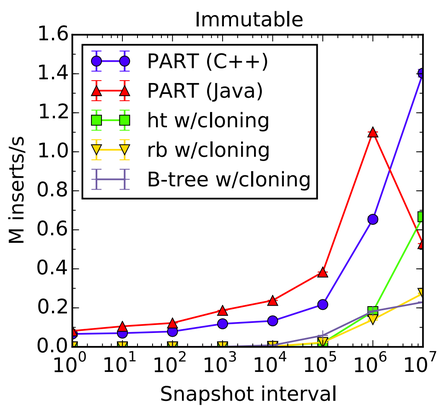
- Adds persistence to the adaptive radix tree using path copying (shadowing) and reference counting
- 950 lines of C++, 1100 lines of Java
- C++ (reference-counted) provides lower average throughput but better predictability than Java (tracing GC)

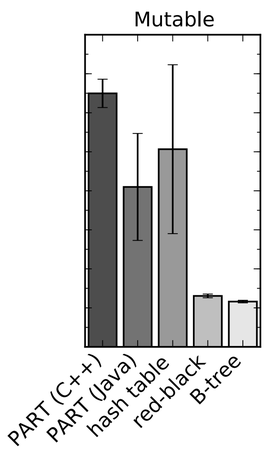


MICROBENCHMARKS

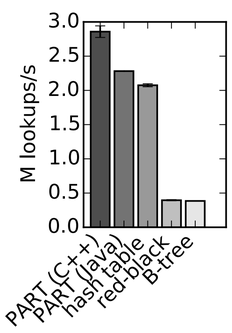
10 million uniform-random pairs of 4-byte keys and 4-byte values

Insert throughput

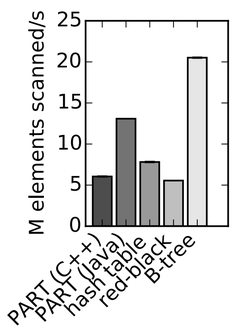




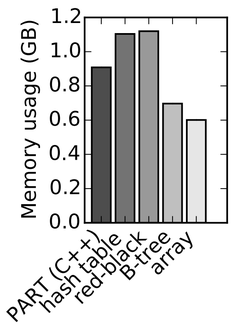
Lookup



Scan



Memory

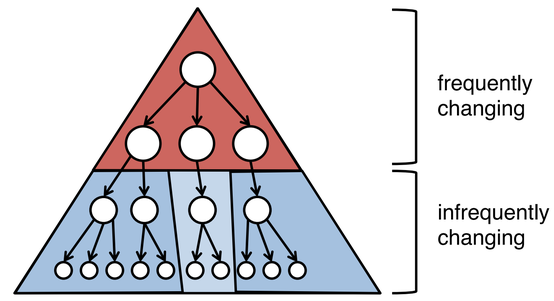


INDEXEDRDD

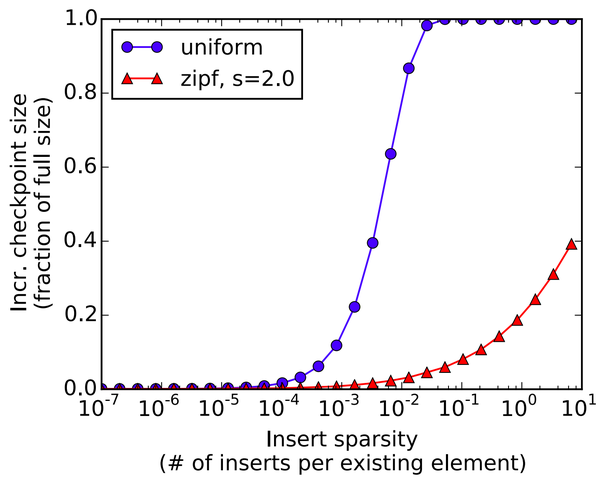
- Efficiently-updatable distributed dictionary in Spark
- Keys are hash or range partitioned into a PART instance in each partition
- Updates are batched, logged to RDD lineage (effectively a write-ahead log), and applied to create a new dictionary
- Available on Spark Packages: https://github.com/amplab/spark-indexedrdd

INCREMENTAL CHECKPOINTING

- Partition the tree into pages and reuse unchanged pages using hard links
- Minimize number of changed pages by segregating frequently-changed nodes from infrequently-changed nodes

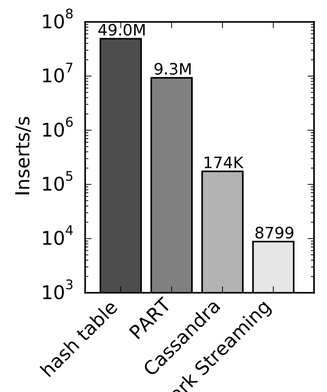


- Substantial space savings for frequent checkpoints or skewed write distributions



EVALUATION

- Streaming aggregation workload: count occurrences of random 26-character keys
- 1B existing keys, 100M-key stream, 8 r3.2xlarge machines



- PART provides:
- 50x more throughput than Cassandra
- 18% the throughput of mutable hash table while preserving versioning and fault tolerance

Intel Science & Technology Center for Cloud Computing
Carnegie Mellon University
Georgia Tech
intel
PRINCETON UNIVERSITY
UC Berkeley
UNIVERSITY of WASHINGTON