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

Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica
University of California, Berkeley

Abstract

Immutability dramatically simplifies fault tolerance, straggler mitigation, and data consistency and is an essential part of widely-used distributed batch analytics systems including MapReduce, Dryad, and Spark. However, these systems are increasingly being used for new applications like stream processing and incremental analytics, which often demand fine-grained updates and are seemingly at odds with the essential assumption of immutability. In this paper we introduce the persistent adaptive radix tree (PART), a map data structure designed for in-memory analytics that supports efficient fine-grained updates without compromising immutability. PART (1) allows applications to trade off latency for throughput using batching, (2) supports efficient scans using an optimized memory layout and periodic compaction, and (3) achieves efficient fault recovery using incremental checkpoints.

PART outperforms existing persistent data structures in lookup, update, scan, and memory usage microbenchmarks. Additionally, we evaluate PART in a variety of distributed applications and show that it improves upon the performance of state-of-the-art immutable and even mutable systems by anywhere from 50x to 4 orders of magnitude.

1 Introduction

The benefits of immutability as a software design principle are well established: reduced vulnerability to concurrency bugs, simplified state management [2, 10], and referential transparency.

Immutability is particularly beneficial for large-scale distributed systems, where coordination is costly and components often fail. One of the primary innovations of contemporary batch analytics systems, including MapReduce [7], Dryad [12], and Spark [24], is their use of immutability to greatly simplify and improve fault tolerance and straggler mitigation. These systems divide queries into deterministic tasks that execute independently in parallel on partitions of immutable input data. In case of failure, these systems rebuild the lost output by simply re-executing the tasks on their inputs. Since inputs are immutable and tasks are independent and deterministic, re-execution is guaranteed to generate the same result. To bound recovery time for long-running queries, they periodically checkpoint the intermediate results. Since datasets are never modified, checkpoints can be constructed asynchronously without delaying the query [14]. Immutability thus enables both efficient task recovery and asynchronous checkpointing.

However, distributed batch analytics tasks are increasingly transitioning to streaming applications in which data and their derived statistics are both large and rapidly changing. In contrast to batch analytics, these new applications require the ability to efficiently perform sparse fine-grained updates to very large distributed statistics in real-time. For example, to analyze trending topics on Twitter we might want to track the frequency of words and phrases across all tweets. Each tweet only affects a very small subset of the frequency statistics. However, to provide real-time trend predictions we would need to update this large collection of statistics as tweets arrive.

Systems like Spark Streaming [25] have attempted to directly support these dynamic applications while preserving immutability. However, by relying on expensive full data copy-on-write mechanisms these systems have largely failed to deliver real-time performance. Others have abandoned the assumption of immutability in favor of more sophisticated solutions to address the challenges of consistency and fault tolerance. For example, systems like Trident [16] and Naiad [17] have explored the use of durable transaction processing and fine-grained logging and checkpointing to achieve consistency and fault tolerance without assuming immutability. However, these solutions complicate parallel recovery and sacrifice the ability to perform multiple unrelated transformations on the same base dataset.

In this paper we revisit the design of immutable batch analytics frameworks for real-time stream processing and challenge the assumption that fine-grained updates are

incompatible with the simplifying requirement of immutability. We build on classic ideas in persistent data structures [8] which efficiently preserve immutability in the presence of updates by leveraging fine-grained copy-on-write to enable substantial data sharing across versions. In addition to providing efficient updates, persistent data structures allow any version to be further modified, yielding a branching tree of versions. Persistent data structures are thus a natural fit for the programming model of batch analytics systems, particularly those which allow applications to access previous versions of the dataset.

However, advanced big data use cases add several new requirements which go beyond the capabilities of existing persistent data structures. First, these systems need to support not only low-latency updates but also fast query processing with high-throughput updates. Consider the trending tweets application in which users can interactively query the latest trending terms and phrases. Such an application must sustain a high volume of updates as the data comes in, while minimizing latency so users have access to the freshest state. Second, the system must support efficient scans (e.g., most popular phrases beginning with "I'm with *") while continuously updating the data, as such scans are common in data analytics workloads. Since some queries are long-running it is not feasible to simply delay updates until a query completes. Third, since failure is common in large systems, we must minimize recovery time; otherwise, the user experience may suffer, or, worse, the application may not be able to keep up with incoming data. Providing fast recovery time requires frequent checkpointing, which is challenging. Fourth, as these systems often deal with spatiotemporal data, they must support efficient range queries.

In this paper we propose the *persistent adaptive radix tree* (PART), an immutable in-memory map designed to address these requirements. PART builds on the adaptive radix tree [13], a state-of-the-art in-memory tree with a high branching factor and adaptively-sized nodes originally developed for database indexing. To support persistence PART uses copy-on-write updates, which are natural for trees since updating an internal node requires copying only the node and its ancestors. To support fast updates we introduce lightweight batching with limited intra-batch mutability, allowing applications to trade off latency (for which hundreds of milliseconds are often acceptable) for throughput by increasing the batch size. To support efficient scans, we add to PART an LSM [19] inspired scan-optimized memory layout with periodic compaction. Finally, to support frequent checkpoints we leverage the design of PART to introduce an incremental checkpointing scheme that reduces checkpoint size while minimizing runtime overhead and recovery cost.

We compare the design of PART to highly optimized mutable data structures with support for point queries and range queries. While persistent data structures are unlikely to achieve throughput parity with the most optimized mutable data structures, by applying low-latency batching PART can achieve update throughput within a factor of 2 of a mutable hash table, while retaining all the advantages of immutability. Its optimized memory layout brings PART within a factor of 2 of B-tree scan performance and within a factor of 2 of optimal space usage. In single-threaded performance and space usage, it outperforms the Ctrie [20], a competing persistent data structure. Additionally, PART exploits the radix tree structure to offer efficient union, intersection, and range scan operations.

We used the PART data structure to add support for efficient fine-grained updates, key-based lookup, and range scans in the Apache Spark immutable batch analytics system [24]. Our implementation of PART in Apache Spark automatically garbage collects unreachable versions, and leverages the design of PART to enable fast incremental checkpointing with reduced storage overhead.

We evaluate the performance improvements of Apache Spark with PART relative to widely used distributed analytics frameworks. PART enables the Apache Spark Streaming API to run over 4 orders of magnitude faster while preserving all the benefits of immutability. Compared to Cassandra, which is designed for fine-grained updates and exploits mutation, PART is 50× faster. Finally, for machine learning applications we find that Apache Spark with PART is able to outperform the standard parameter server [15] by exploiting the short feature ids and skewed update distribution common in real-world data.

In summary, the contributions of this paper are:

1. The persistent adaptive radix tree (PART), an immutable in-memory map designed to support efficient fine-grained updates within analytics systems.

2. A set of techniques that allow PART to (1) trade off latency for throughput, (2) support efficient scans, and (3) achieve low-overhead fault recovery.

3. A distributed key-value store based on PART and Spark that outperforms widely-used distributed streaming systems.

2 Immutability vs. Efficient Small Updates

Parallel computing systems that use a dataflow programming model (MapReduce [7], Dryad [12], Spark [24], Hyracks [5], Pig [18], FlumeJava [6], and others), allow users to build and manipulate parallel collections of records, which we refer to as *datasets*. Users manipulate these datasets through data-parallel abstractions composed of basic operators such as the *map*, *reduce*, and *filter*. These operators are then executed broken into

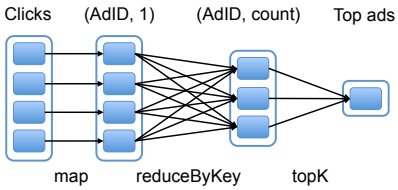


Figure 1: **Example task dependency graph.** Tall boxes represent datasets, while filled boxes represent partitions of a dataset, which are computed in parallel by different tasks.

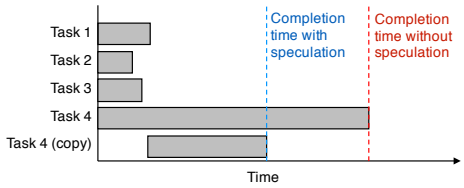


Figure 2: **Straggler mitigation.** Most tasks complete quickly, but Task 4 is slow (*e.g.,* due to server overload). After a delay, Spark schedules a duplicate copy of the slow task on a different machine. If it finishes sooner than the original task, its result is used.

tasks that execute concurrently across the cluster in *batch analytics systems*. Each task operates independently on its input records. For example, a map task would run a user function on each element of its input to produce a corresponding output record. Figure 1 shows a dependency graph for a simple batch program and its constituent tasks. In the remainder of this section we will use Spark as a motivating example of a batch analytics system.

Like other distributed batch analytics systems, Spark provides mechanisms for fault recovery and straggler mitigation. In fault recovery, Spark uses the dependency graph between tasks to recompute data partitions lost after a failure. If multiple partitions were lost, they can be recomputed in parallel across the cluster. Spark refers to the distributed dataset along with its dependency graph as a *resilient distributed dataset (RDD)*. With straggler mitigation, the system speculatively launches a duplicate copy of a slow-running task and then uses the output of the task that finishes first, while killing the other one. Figure 2 shows an example of straggler mitigation.

In order to provide these features, batch analytics systems require tasks to be idempotent: repeated executions of the same task must lead to the same result. To ensure a task's idempotence, these frameworks assume (1) the task's inputs are *immutable*, and (2) the task's execution is deterministic. To achieve deterministic execution, tasks are typically single threaded, and to preserve immutability of datasets, tasks must respect copy-on-write semantics by copying (cloning) a dataset before updating it.

Cloning a dataset works well for updates that involve a large fraction of the dataset, which is the case for many analytics workloads today. In those cases, the fixed cost of copy is amortized across the many updates.

However, batch analytics systems are increasingly being used for applications that require incremental behavior such as streaming aggregation and incremental algorithms. This broader adoption is due to the flexibility, scalability, and fault tolerance that these systems provide along with the opportunity to apply existing analytics code to new applications. Unifying batch and incremental analytics within the same system simplifies development and deployment, avoids overheads due to data transfer and re-encoding, and allows mixing multiple programming models to provide increased expressivity.

Unfortunately, because incremental applications require sparse, fine-grained updates, they are often difficult to implement and perform poorly in distributed batch analytics systems. For example, many web services today want to update their user table every few seconds to maintain up-to-date statistics. However, the set of users that are active over a time interval is typically much lower than the total number of users, and thus only a small fraction of entries in the user table are modified. For these cases the overhead of cloning the entire dataset can be prohibitive, limiting the scalability of systems like Spark Streaming [25] that try to support these workloads while using cloning to ensure immutability.

While there are other techniques that support efficient fine-grained updates, such as using mutable data structures or database systems, they come with significant drawbacks. In-place updating a mutable dataset is fast but violates the requirements of the fault tolerance model. Task failures may trivially corrupt the state, preserving the old values for some keys but updating values for others.

Alternatively, one might rely on a transactional database that supports atomic batched writes, whether local and checkpointed (*e.g.,* LevelDB) or external and replicated (*e.g.,* Cassandra). Storm's Trident layer [16] uses this approach, and streaming systems including Naiad [17] use logging and checkpointing to implement similar recovery mechanisms. However, this restricts the ability to express complex tasks that perform multiple unrelated transformations on a snapshot of the data. This ability can be recovered by creating a consistent database snapshot before each such transformation, but such snapshots incur high overhead in modern databases and are not optimized for bulk scans. Additionally, parallel recovery is greatly complicated in the presence of mutable state.

As a result, existing systems present a seemingly inescapable tension between the assumption of immutability—essential for fault tolerance, straggler mitigation, and simplicity—and the desire to support the high-throughput fine-grained updates demanded by many new streaming and incremental applications. However, in

the next section we demonstrate that by leveraging small tradeoffs in latency and storage overhead and introducing the right data structures we can achieve high-throughput fine-grained updates that preserve immutability.

3 Persistent Adaptive Radix Trees (PART)

We now introduce our persistent map data structure, the persistent adaptive radix tree (PART). We first describe the adaptive radix tree, the ephemeral data structure it is based on. We then describe how we added efficient persistence to the adaptive radix tree by exploiting its tree structure and using batched updates, while retaining good scan performance using an optimized memory layout and automatic compaction. Finally, we demonstrate PART's efficiency using microbenchmarks.

3.1 Background on Adaptive Radix Trees

Radix trees (also known as tries) represent key-value data by storing the value for a particular key at the position in the tree corresponding to the sequence of bits in the key. A radix tree with branching factor 2^s is implemented by storing an array of 2^s pointers at each node, enabling traversal by following the pointer at the offset corresponding to the relevant s bits of the key. Additionally, radix trees typically use path compression, in which chains (sequences of nodes with only one child) are compressed into a single node to avoid unnecessary indirection.

Radix trees offer several benefits compared to other data structures such as hash tables and binary search trees.

1. Unlike hash tables, radix trees store keys in sorted order, enabling range scans using in-order traversal.
2. Compared to binary search trees, radix trees offer superior asymptotic performance by exploiting the key structure. Each comparison in a binary search tree provides only one bit of information and reduces the key space by up to a factor of 2, while a radix tree with $s > 1$ can eliminate far more keys at each node. As a result, for keys of length k, operations in a binary search tree have complexity $O(k \lg n)$, while operations in a radix tree have complexity $O(k)$.
3. Radix trees support efficient union and intersection operations, because they enable large numbers of keys to be eliminated based on their prefix structure.
4. Radix trees require no rehashing or rebalancing, so insertion performance is much more predictable than for a hash table or a self-balancing binary search tree.

The adaptive radix tree [13] is a radix tree with a branching factor of 256. In addition to reducing the tree height, this choice of branching factor simplifies tree operations, because each node may consider the key one byte at a time without need for bit shifting or masking. However, such a large branching factor would naively result in excessive space usage because, for a sparse tree, most child pointers at each node would be null.

The key idea of the adaptive radix tree is to reduce space usage by compressing nodes based on their sparsity, while keeping traversal time to a minimum by carefully choosing the representation of compressed nodes. To this end, the adaptive radix tree introduces four node types corresponding to different levels of sparsity. The number of node types is chosen to balance space usage against the frequency of node replacements when insertions or deletions change the sparsity. Note that nodes with only one non-null child out of 256 possible children are eliminated by path compression.

1. Nodes with up to 4 non-null children are stored using an array of 4 key fragments, each key fragment occupying a byte, and a parallel array of 4 child pointers. Key lookup is implemented by linearly searching the first array for the relevant byte of the key, then retrieving the child pointer from the corresponding position in the second array.
2. Nodes with 5 to 16 non-null children are stored using two 16-element parallel arrays as before. Because the 16-byte key fragment array fits into a 128-bit SIMD register, vector instructions can be used for key lookup by comparing the relevant byte of the key to all 16 stored key fragments in parallel rather than incurring the cost of a linear or binary search.
3. Nodes with 17 to 48 non-null children are stored using a 256-element key fragment array and an array of 48 child pointers. Unlike the smaller node types, these arrays are not parallel; instead, the key fragment array is addressed by the key fragment and stores a 6-bit index into the child pointer array. Key lookup is implemented by accessing the key fragment array at the offset corresponding to the relevant byte of the key, then using the stored value to index into the child pointer array. This avoids an expensive scan in favor of two array accesses.
4. Nodes with 49 to 256 non-null children are simply stored using an array of 256 child pointers indexed by the relevant byte of the key.

3.2 Persistence

Although adaptive radix trees provide indexing, range scans, and efficient fine-grained updates, they do so through in-place *mutation*. To preserve *immutability* we turn to a central technique in the design of persistent

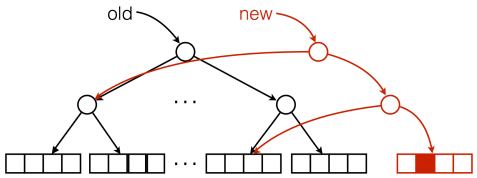


Figure 3: **Path copying.** Updates to an immutable tree involve copying a subset of the nodes and returning a new root reflecting the change. The old root remains valid until garbage-collected.

data structures [8]. With each update, a persistent data structure returns a new version which reflects any changes while sharing substantial portions of the internal representation with the unmodified previous version. Both the original and updated versions may go on to be further modified, yielding a branching history of versions.

Because modifications to nodes in adaptive radix trees only affect the $O(\log_{256} n)$ ancestors of that node, they can be efficiently transformed into persistent data structures using a technique known as *path copying*. Path copying (also referred to as shadowing), as the name implies, copies the path between the modified node and the root while preserving the remainder of the tree. More specifically, when an element is modified, we allocate a copy of the node containing the element to be modified, make the modification in the new copy, and repeated the procedure for all ancestors. Figure 3 shows how path copying may be used to update a leaf node without in-place modification. We call this efficient persistent data structure the persistent adaptive radix tree (PART).

We implemented a highly optimized version of PART using just 950 lines of C++. In addition to path copying, each tree node maintains a reference count so expired versions may be removed even while new versions reference parts of their data. We chose reference counting over tracing garbage collection to provide predictable performance; Figure 4 shows the impact of garbage collection pauses on a Java reimplementation of PART. Additionally, because internal nodes have just four possible sizes, we use object pools to reduce allocation overhead and control memory fragmentation.

We next discuss an optimization to PART when not all previous versions are needed. In Section 3.6 we will see that this optimization mitigates the overhead of path copying and in some cases allows PART to match the performance of in-place mutation.

3.3 Batched Updates

Compared to in-place mutation, path copying in PART incurs overhead for modifications because each element modification additionally requires allocating new nodes proportional to the tree height. However, this overhead can be reduced when not all previous versions are needed.

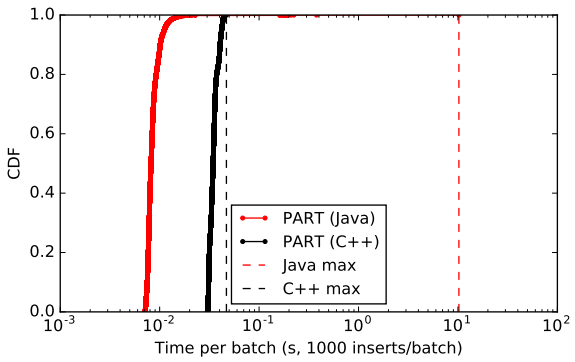


Figure 4: **Impact of garbage collection.** The C++ implementation of PART has predictable update latency (black line), while the Java implementation has lower average latency but a much larger variance due to garbage collection pauses (red line).

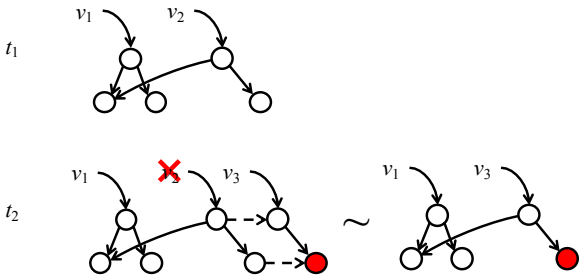


Figure 5: **In-place optimization for updates.** At time t_1, a dataset has two versions v_1 and v_2 that share some structure. At time t_2, version v_2 is updated, creating v_3. Since the affected node is referenced only by v_2, which will be discarded after the update, path copying (lower left) is equivalent to an in-place update (lower right).

In particular, updates to a version that will immediately be discarded can be made in place when they would not affect any other versions. For example, suppose an application has two versions of a dictionary v_1 and v_2, and it applies an update to v_2 forming v_3. If v_2 is subsequently never referenced, and the update applies only to tree nodes referenced by v_2, not those shared with v_1, then the update can be implemented as an in-place mutation to the nodes of v_2. Figure 5 illustrates this scenario.

We expose this optimization to users of PART by providing a *snapshot* operation as well as an in-place *update* operation. Updates that must preserve the existing version are performed by calling *snapshot* followed by *update*, while updates that may overwrite the existing version are performed simply by calling *update*.

This optimization is implemented using the existing reference counts for each tree node. The *snapshot* operation clones the root and increments the reference counts of each of its children. A call to *update* traverses the tree to find the appropriate leaf and examines the reference counts of each node on the path from the root to the leaf.

Any node with a reference count greater than 1 is unsafe to update in place, so it and its descendants must be copied to perform the update. However, nodes on the path from the root to the first such node are only referenced by the current version and can be safely modified.

As a result, two updates to the same node without a snapshot in between only require copying the node once. When snapshots are infrequent this allows update performance to approach the performance of in-place mutation despite maintaining two versions, because many updated nodes have already been updated since the last snapshot. Also, for non-uniform access distributions, frequently-updated elements will only be copied once and then can be updated in place.

3.4 Node Allocation and Compaction

Implementing updates using path copying and batching enables efficient updates while maintaining immutability, but seems to come at the expense of scan performance because of three challenges. First, a naive tree construction strategy will likely allocate nodes in a suboptimal order and with high fragmentation, especially under memory pressure. Second, as versions are created and deleted, fragmentation will worsen because shared nodes will become increasingly scattered across memory. Third, PART's tree structure inherently introduces overhead compared to a sequential array scan due to the space overhead of the internal tree nodes and the traversal overhead of following pointers from one node to the next.

In this section we address the first and second challenges by optimizing PART's node allocation strategy and introducing periodic node compaction.

Node allocation. We use a custom memory pool allocator for PART that allocates new nodes of each type contiguously and in traversal order when possible. In particular, leaves are allocated in sorted order to improve scan performance. (However, due to the possibility of updates we must still traverse internal nodes.) When creating a derived version, we allocate its new nodes contiguously but maintain backward references to unchanged nodes from previous versions.

Compaction. When a version is deleted, there is an opportunity to improve scan performance for the remaining versions by regaining a contiguous layout using compaction. In compaction, nodes are rearranged to optimize scan performance and referring pointers are updated.

Compaction is triggered automatically during idle time. It operates incrementally and is interrupted if a task arrives to avoid delaying latency-sensitive jobs. Each PART instance maintains metadata used to determine whether or not to run compaction and what layout compaction should create. The metadata contains a version history graph where each connected component represents an independent set of PART versions. For each connected component in the history graph, versions with more elements are prioritized for contiguous storage over versions with fewer elements. For example, consider the following versions, illustrated in Figure 6:

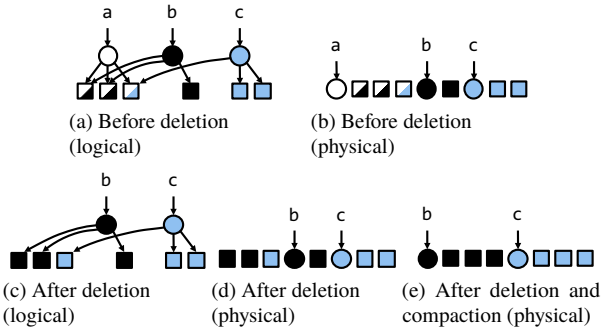


Figure 6: **Compaction.** (a) The nodes of versions `a`, `b`, and `c` are colored by version, with shared nodes in mixed color. Root nodes are shown as circles and leaves as squares. (b) The nodes are laid out in memory in creation order. (c) Version `a` is deleted but shared nodes are preserved. (d) However, this results in a fragmented memory layout. (e) Compaction reorders `b` and `c` to be contiguous.

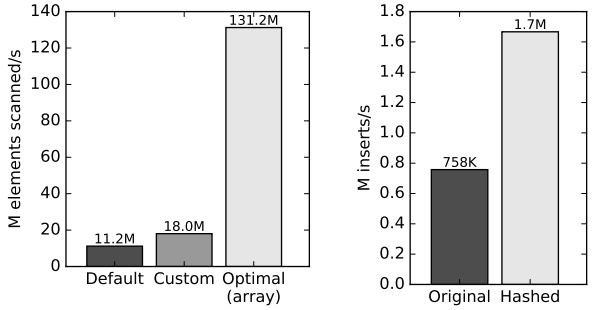
```
val a = PART(3 keys)
val b = a.update(1 key)
val c = a.update(2 keys)
// Later:
a.destroy()
```

Initially the versions are laid out in creation order, with `a` fully contiguous and `b` and `c` containing backward references to `a`. However, the last line destroys `a`. Afterwards, when compaction occurs it will reorganize both `b` and `c` to be contiguous.

Impact on scan performance. Figure 7a shows the results of custom node allocation on scan performance in a PART instance with 10 million pairs of 4-byte keys and 4-byte values. Compared to using the default C++ allocator, our allocator results in 60% better scan performance. The remaining overhead compared to a sequential array scan is due to the space and traversal overhead from PART's internal nodes, which are necessary to support updates.

3.5 Key Space Transformations

A crucial weakness of radix trees is their vulnerability to long keys, which can degrade performance severely by increasing tree depth and forcing deep traversals. As discussed in Section 3.1, PART largely mitigates this problem using path compression to eliminate unnecessary indirection (in addition to the constant-factor reduction in tree depth from choosing a high branching factor). However, certain key distributions seen when events cluster together can render path compression ineffective.



(a) Compaction (b) Key space transformation

Figure 7: **PART optimizations.** (a) Compared to the default C++ allocator, our custom allocator improves scan performance by 60%. (b) Hashing transforms highly skewed keys into nearly uniform keys, providing a 2.2× speedup by reducing the worst-case tree depth.

Inspired by the HAMT [3], we provide the option of applying a randomly selected hash function in this case to minimize the probability of such a distribution, though this comes at the expense of in-order traversals. Figure 7b shows that this optimization provides a 2.2× speedup when evaluated with 32-byte keys chosen from a highly skewed distribution.

3.6 Microbenchmarks

The optimizations to PART in Sections 3.3 to 3.5 aimed to mitigate its weaknesses in scan and update performance. Figure 8 summarizes the performance impact of these optimizations.

In this section we now demonstrate that, with these optimizations, PART's performance and memory efficiency are competitive with standard mutable data structures. We compare PART against a mutable hash table implemented using chaining (STL's `unordered_map`), a mutable red-black tree (STL's `map`), and an in-memory B-tree (Google's `cpp-btree`). While faster implementations exist for some of these data structures, the implementations we chose

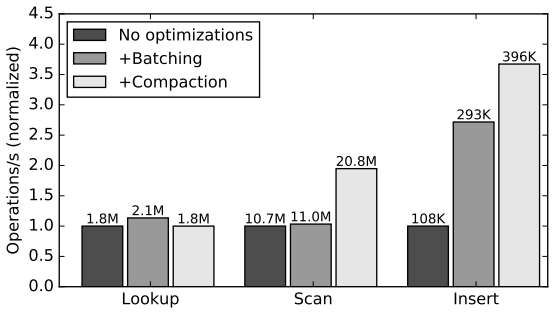


Figure 8: **Summary of optimizations.**

are widely used and well-understood. We aim for performance comparable to these while providing persistence rather than attempting to match the speed of the fastest available mutable data structures.

We additionally aim to show that PART is better-suited for distributed analytics than existing persistent data structures. To this end we compare against the Ctrie [20], a JVM-based variant of the HAMT supporting concurrent read-only snapshots.

We loaded each data structure with 10 million uniform-random 4-byte keys. We conducted two sets of experiments with different value sizes: Figure 9 uses 4-byte values, while Figure 10 uses 1024-byte values. Error bars denote standard deviations and are shown on every point, though they may be too small to see.

Figures 9a and 10a show that PART matches the hash table and outperforms the other data structures in random lookup performance. Note that all lookups in this microbenchmark result in success, which is the worst case for PART because each lookup must traverse the whole tree. For lookups resulting in failure, PART's radix tree structure enables early termination, and the first few levels of the tree are likely to be present in cache. As a result, for lookups resulting in failure, PART is 40% faster than the hash table (not shown). Figures 9b and 10b show that PART is comparable to the hash table and outperforms the Ctrie and red-black tree in scan performance. PART benefits from node compaction, which places many leaf nodes in sequential order. The B-tree is 2x faster than PART for scans because its large, densely-filled leaf nodes minimize pointer lookups when scanning.

We measure the performance of random inserts in Figures 9c and 10c. As an upper bound, we consider in-place inserts in the chart on the right. When performing all updates in place, PART is effectively a reimplementation of ART. In this case it is able to outperform the mutable hash table by a factor of 2, largely due to the hash table's open addressing and periodic need for rehashing.

On the left side of Figures 9c and 10c we plot the performance of each data structure with persistence for varying snapshot intervals. For the hash table, red-black tree, and B-tree, we implemented persistence naively by cloning the data structure before applying updates, while for PART we used the snapshot-based persistence of Sections 3.2 and 3.3, and for the Ctrie we used its read-only snapshot capability. With 4-byte values and a snapshot interval of 1, corresponding to creating a new version for every insert, PART provides 127,000 inserts per second, the Ctrie provides 83,000 inserts per second, and the other data structures provide less than 1 insert per second due to the overhead of cloning. At a more realistic snapshot interval of 1000, PART provides 256,000 inserts per second, the Ctrie provides 136,000 inserts per second,

and the other data structures provide less than 300. As the snapshot interval approaches the data structure size, later inserts can be performed in place and PART's performance approaches the mutable case. For 1024-byte values, the cost of operating on the values begins to dominate. As in Figure 4, the Ctrie performs better on average than the C++-based data structures at the cost of high variance due to unpredictable garbage collection.

Finally, we measured memory usage in Figures 9d and 10d. For data structures implemented in C++, we report the process resident set size after loading each data structure with 10 million pairs, while for the JVM-based Ctrie, we calculate the data structure size by traversing its in-memory representation using reflection. As a lower bound, we also measured the memory usage of two parallel arrays for keys and values.

4 System

In this section we describe our implementation of a distributed dictionary in Spark using PART. Our dictionary uses the standard hash and range partitioning techniques for distributed storage; elements are assigned to partitions by hash or range partitioning their keys.

At each partition, elements are stored off-heap and managed by an instance of our C++ implementation of PART. When an RDD is no longer used at the driver, Spark notifies the PART process at each partition using JNI, allowing it to recursively free nodes not referenced by other PART data structures within the same partition.

We reimplement many standard Spark operations such as *filter*, and *join* to take advantage of the capabilities of PART. For example, *filter* can avoid copying the leaf nodes containing the elements, instead filtering the tree structure but reusing the pointers to the remaining leaves. Other Spark operations such as *map*, *foreach*, and *group-by* are implemented by falling back to the standard Spark implementation, which only requires PART to expose an iterator to traverse the elements sequentially through JNI and leverages the optimizations for bulk scan support introduced in PART.

We augment the Spark RDD interface by adding support for efficient key lookups, updates, insertions, and deletions, as well as unions, intersections, and joins. These operations were previously emulated using full dataset scans; we use PART to implement them more efficiently. Lookups, updates, insertions, and deletions are implemented as described in Section 3. When two RDDs are co-partitioned (*i.e.,* they share a partitioning function), unions, intersections, and joins can be implemented using local operations at each partition. Unions and intersections exploit the radix tree structure, making them particularly efficient for disjoint key sets. Joins are implemented as coordinated scans over the two data structures at each partition.

4.1 Incremental Checkpointing

Spark supports fault recovery for batch and streaming applications by logging the input file or stream as well as any operations needed to construct the application's datasets from the input. Using this scheme, log space and recovery time for streaming and iterative applications would grow unboundedly as the input stream or number of iterations grows over time. Spark therefore periodically checkpoints all active datasets, bounding recovery time by allowing it to use the checkpoint rather than the log for all changes up to the time of the checkpoint. Checkpointing is generally a very expensive operation because it involves writing the entirety of each active dataset to stable storage, which is usually replicated for fault tolerance.

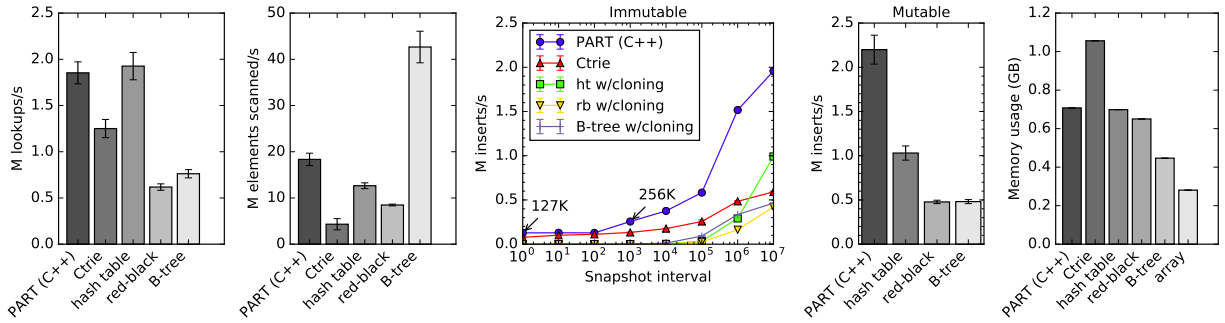
Checkpointing each dataset in full is required for applications where most records change from one checkpoint to the next. However, in workloads where records change infrequently, this can be inefficient. We use PART to reduce the checkpoint data size when parts of the key space have not changed since the last checkpoint. If an entire subtree is the same since the last checkpoint, which can occur for skewed update distributions, we simply refer to it in the old checkpoint rather than writing it out again.

Checkpointing with hard links. We implement tree checkpointing by (1) taking a snapshot for consistency, (2) partitioning the tree coarsely into subtrees, and (3) storing each subtree as a different file, representing inter-page pointers as file identifiers. The root node of each subtree in memory holds the file identifier for its subtree on disk, and in-place updates clear these file identifiers. This way, any subtree with a file identifier is guaranteed to be checkpointed to its latest version, allowing future checkpoints to avoid duplicating these subtrees.

Instead, future checkpoints represent unchanged subtrees using *hard links* to their original files. Since checkpoint files from different trees are stored in different directories, checkpoint cleanup is a simple matter of deleting the appropriate directory. The hard links ensure that subtrees still referenced by new checkpoints are not deleted.

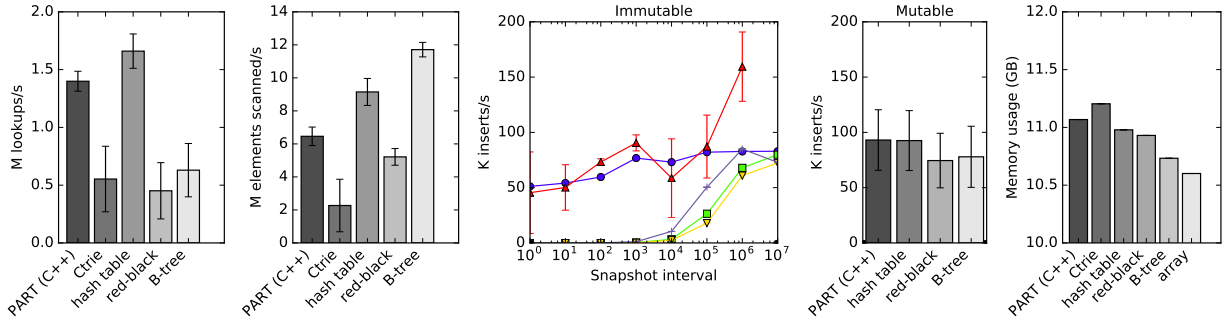
Tree partitioning. Due to path copying, each update to a PART node creates new versions of all of the node's ancestors up to the root. Nodes higher up in the tree are thus more likely to change from one snapshot to the next, and the root always changes between versions. However, in the above coarse file-based scheme, a file containing even a single changed node must be copied in full. We therefore desire a tree partitioning scheme that minimizes the number of changed files per update.

Our solution is a tree partitioning that concentrates the most frequently-changed nodes (those close to the root) into a single file, while dividing infrequently-changed



(a) Lookup throughput (b) Scan throughput (c) Insert throughput (d) Memory usage

Figure 9: **Data structure comparison for 10 million pairs of 4-byte string keys and 4-byte integer values.**



(a) Lookup throughput (b) Scan throughput (c) Insert throughput (see Figure 9 for legend) (d) Memory usage

Figure 10: **Data structure comparison for 10 million pairs of 4-byte string keys and 1024-byte string values.**

nodes (the subtrees closest to the leaves) into multiple files. We use a depth threshold as a heuristic to separate frequently-changed nodes from infrequently-changed nodes. Figure 11 illustrates this partitioning scheme.

Efficiency experiment. We measured the effectiveness of this coarse-grained incremental checkpointing scheme by loading a PART instance with 10 million key-value pairs and creating an initial checkpoint, then measuring the size of an incremental checkpoint after a number of random writes.

Figure 12 plots the size of the incremental checkpoint for varying numbers of uniform and Zipf-distributed writes. For small numbers of uniform writes, incremental checkpointing reduces checkpoint size by up to 3 orders of magnitude. It provides space savings up to 1 write for approximately 100 existing elements (*i.e.,* 1% sparsity). Due to their skewed update pattern, Zipf-distributed writes continue to see space savings from incremental checkpointing even when the number of writes exceeds the number of existing elements.

Recovery from checkpoint. Finally, recovering a dataset from checkpoint is straightforward: we read each checkpoint file starting at the root, recursively load each node's children by following a link within the same file or opening a new file, and set the child pointers to the memory addresses of the newly-loaded children.

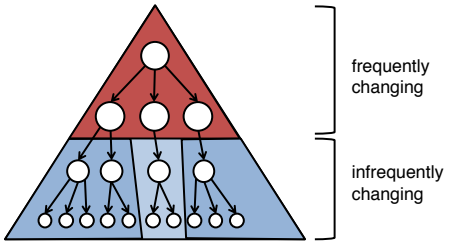


Figure 11: **Tree partitioning:** minimize number of changed files by segregating frequently-changed nodes from infrequently-changed nodes.

4.2 Concurrency

Concurrent operations on a single partition occur in Spark in two situations. First, multiple jobs may be launched against a single RDD using concurrency at the driver. This typically occurs when multiple users share a driver using a *job server* to access shared in-memory datasets. Second, RDDs may be checkpointed concurrently in

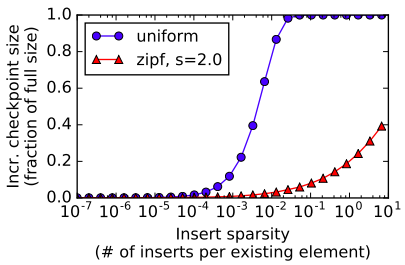


Figure 12: **Incremental checkpointing:** provides space savings for uniformly-distributed updates of sufficient sparsity (up to 1%) and for skewed update distributions almost regardless of sparsity.

the background, avoiding unavailability or a backlog of streamed input.

A key observation is that these situations only require support for concurrent snapshots and concurrent reads to an existing snapshot, not concurrent writes to a new snapshot. PART therefore supports concurrency simply by using atomic increment and decrement operations when updating node reference counts. This ensures concurrent snapshots will perform copy-on-write updates for conflicting nodes.

5 Evaluation

In this section we demonstrate that our PART-based distributed dictionary offers both stronger versioned semantics and superior performance compared to existing analytics systems. We implement four distributed applications using PART and compare them to existing alternatives.

Cluster-based experiments in this section were conducted on Amazon EC2 using 8 r3.2xlarge worker nodes in March 2015–April 2017. Each node had 8 virtual cores, 61 GB of memory, and a 160 GB SSD. Single-node experiments were conducted on a machine with a 2 GHz Intel Core i7 processor and 16 GB of memory.

5.1 Streaming Word Count

Given a stream of 26-character alphanumeric strings uniformly distributed over the key space, in streaming word count we tally the number of occurrences of each key seen so far using a 64-bit counter. For this experiment we loaded the 8-node cluster with an initial set of 1 billion keys, then measured each system's average throughput over a stream of 100 million keys.

This workload exhibits a sparse update pattern where only a small fraction of keys are updated in each timestep. The sparse workload is intended to model applications such as a social network where each user maintains profile statistics (*e.g.,* the user's tweet count), but where during any interval, only a small fraction of users interact with the service (*e.g.,* by sending a tweet) and need to be accessed. We performed the comparison across the following systems:

1. Our PART-based distributed dictionary, dividing the input into 100 batches of 1 million tuples each.
2. A similar implementation using a Ctrie at each partition instead of PART.
3. Spark Streaming 1.3.0, a fault-tolerant stream processing system, also with 100 batches of 1 million tuples each.
4. Cassandra 2.1.3, a distributed database, configured with a replication factor of 1.
5. As an upper bound, a per-partition C++ mutable hash table (STL's `unordered_map`). Unlike the other systems, this does not provide fault tolerance or access to previous versions.

To ensure that we measured each system's maximum throughput, we performed load generation in parallel on each node for all systems. Each load generator's output was constrained to the key range of its partition, ensuring that none of the systems needed to communicate any data over the network.

Figure 14a shows the average throughput of these systems. The mutable hash table provided overall average throughput of 49 million keys per second on the 8-node cluster. Our PART-based distributed dictionary provided overall average throughput of 9.3 million keys per second, 18% the performance of the mutable hash table while providing fault tolerance and access to previous versions. The Ctrie-based approach processed 7.5 million keys per second, a result that was suprisingly close to PART's due to the key hashing used by the Ctrie that reduced its tree depth at the cost of support for range scans. Cassandra provided overall average throughput of 174K keys per second. Though we avoided the need for immediate write visibility (by enabling batching) and network communication (by disabling replication and using local load generators), Cassandra was still limited by disk I/O in the leveled compaction process.

Spark Streaming did not finish but averaged 8,799 keys per second. This was because it joins the input batch with the existing aggregates by rehashing both datasets to form a new dataset, using an approach similar to that described in Section 2 involving copying the aggregated state in full and then modifying the clone in place. This strategy is appropriate for dense aggregation workloads where each key is likely to be updated in every batch, but is very inefficient for the sparse workload tested in this benchmark. Including existing tuples in the throughput

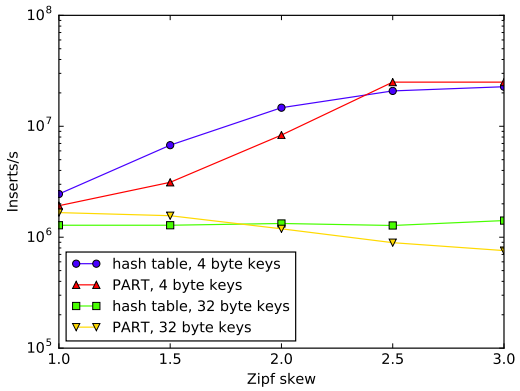


Figure 13: **Effect of key length and skew.** Skew disproportionately benefits PART for short keys but disproportionately harms it for long keys.

calculation shows that Spark Streaming was processing 2.7 million total records per second, but almost all of these were existing records.

Key length and skew. To explore the effect of key length and distribution skew on the streaming word count application, we conducted a single-node experiment comparing insert performance for PART and the mutable hash table under varying key lengths (4 byte keys to 32 byte keys) and Zipf skew parameters (1.0, representing a uniform distribution, up to 3.0, a highly skewed distribution). We did not use any key space transformations for this experiment. Figure 13 shows that for short keys, increased skew benefits both data structures due to improved cache hit rate, but PART benefits more than the hash table (in fact outperforming it at high levels of skew) because its sorted layout allows it to benefit from the spatial locality of reference created by the skew. These results replicate the trend reported for ART [13]; note that the absolute numbers are not comparable due to the large difference in hardware. However, for long keys the reverse trend occurs: increased skew still benefits the hash table slightly, but it decreases PART's performance by a factor of 2. As explained in Section 3.5, this is because skew for long keys increases the average tree height by reducing the opportunities for path compression that would otherwise occur for isolated keys.

5.2 Time Series Ingestion

In time series ingestion, we load a continuous dense stream of timestamped values and index them by timestamp. This is useful for log analysis, where a high volume of timestamped log entries arrives mostly in order with some probability of reordering, and most queries involve fixed or sliding time windows.

We performed this experiment on a single node, initializing each data structure with 1 million values and then measuring the ingestion performance for a further 1 million values in temporal order. We stored timestamps using the standard 64-bit `time_t` representation. As an upper bound, we compared PART to a mutable hash table, which does not support time windowing but does support out-of-order arrivals.

Figure 14b shows that, as the batch size increases, PART outperforms the Ctrie and approaches and exceeds the performance of the mutable hash table while providing the benefits of immutability. The fact that larger batch sizes have better performance makes PART well-suited to high-utilization streaming, where a failure could increase the backlog beyond the system's stability threshold and prevent the system from ever catching up. Due to batching, PART's efficiency scales with the size of the backlog, reducing the likelihood of a runaway backlog.

Recovery experiment. To demonstrate the impact of batching on fault recovery, we further simulated a machine failure in the time series ingestion application. Figure 15 shows that, when a failure occurs in batch 10 and the system must rerun all 10 previous batches for the affected partition, batching these updates together speeds up recovery by more than a factor of 5.

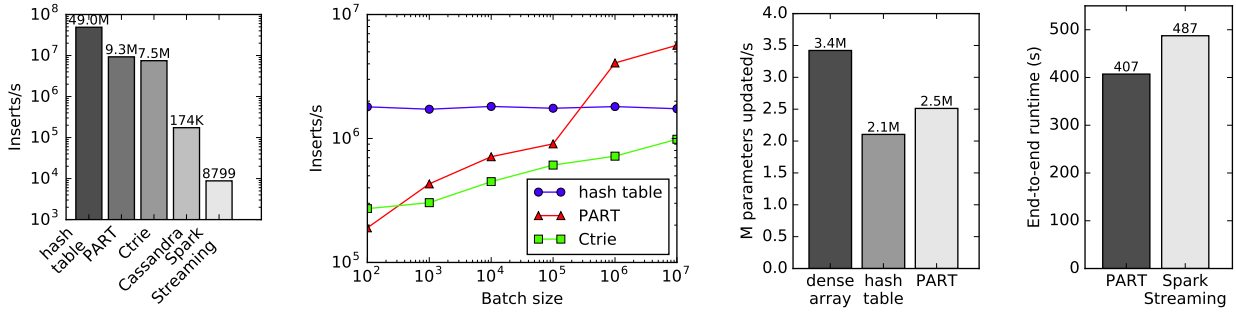
5.3 Parameter Server

We implemented Latent Dirichlet Analysis (LDA), a topic modeling algorithm, using a parameter server approach [15] where each word results in an update to a number of parameters stored in a sparse distributed map. We used PART to implement this map and we compared it against a mutable hash table on a synthetic corpus where words appear with a skewed distribution.

We performed this experiment on a single node, running 20 iterations of LDA over a corpus with 10 million words and 4-byte parameter ids. For PART, each iteration was performed using a single batched update; in the distributed setting this corresponds to a minibatch training approach. Figure 14c shows that PART performed well on this algorithm since this workload is well-suited for it due to the short keys and high update skew. PART outperformed the hash table and achieved 73% of the performance of a dense parameter array containing all 2^{32} parameter entries, in which a parameter update requires only a single array lookup. This dense approach would be infeasible for larger parameter spaces, but we include it as an upper bound in this comparison.

5.4 Historical Experiments on the Netflix Watch Stream

To motivate immutability and show that PART provides an end-to-end speedup, we implemented an application inspired by a real-world problem in maintaining Netflix's



(a) Distributed streaming word count (b) Time series ingestion (c) LDA using a parameter server (d) Netflix watch stream

Figure 14: **End-to-end experiments.** (a) PART outperforms other systems and achieves 18% the performance of the mutable hash table without sacrificing immutability. (b) PART's performance scales with batch size, allowing it to recover quickly in case of failure. (c) PART performs well due to the short parameter id length and high skew in this machine learning algorithm. (d) PART's efficiency gain over Spark Streaming results in a faster end-to-end pipeline runtime.

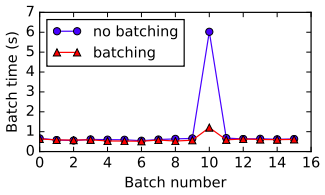


Figure 15: **Effect of batching on fault recovery.** Batching improves the speed of fault recovery by more than 5x in this benchmark.

movie recommendation system [22]. Netflix trains its recommendation algorithm on a *watch stream* recording each time a user watches or rates a movie. Any proposed change to the recommendation algorithm must demonstrate improved results in historical experiments comparing it to existing algorithms on past snapshots of the watch stream. The candidate algorithm is trained on an old snapshot, then tested on a newer snapshot where its predictions are compared with existing algorithms.

We reimplemented a similar pipeline using PART to update and maintain historical snapshots. We generate 10 million synthetic movie watch events, uniformly distributed over 1 hour and 100 million users and Zipf distributed over 100,000 movies. We maintain a PART instance mapping each 8-byte user id to a 100-element vector, and another PART instance mapping 4-byte movie ids to 100-element vectors. We update these feature collections by batching the watch events into 5-second intervals. For each watch event, we make a random update to the corresponding user and movie feature vectors. We snapshot the feature collections every 100 batches. Finally, we scan the first snapshot 100 times to simulate training a recommendation algorithm on it, then scan the last one to simulate comparing predictions with subsequent events.

We compare against taking periodic full snapshots of copied state (similar to Spark Streaming). Figure 14c compares the runtime of these two alternatives when run on these stages (ingesting and experimentation). PART is more efficient because it saves space compared to a full snapshot while providing access to historical versions.

6 Related Work

The capability of persistent data structures to perform updates without modifying previous versions is similar to work on versioning in databases. The multiversion concurrency control (MVCC) capabilities [4] in many popular databases rely on versioned mutable data structures such as the log-structured merge tree [19]. However, techniques used to implement MVCC are often not directly applicable for the analytics setting because they assume versions will be short-lived, lasting only for the duration of the transaction before being reconciled. On the other hand, versions in the analytics setting are closer to different views of the same data; versions may persist for the entire lifetime of an application and multiple versions may be joined together.

Work in versioned file systems and databases does aim to address the case of long-lived versions, which are directly used to provide immutable semantics. Copy-on-write B-trees are similar to our work and are used to provide snapshots in ZFS [11], btrfs [21], and CouchDB [1], among others. The Stratified Doubling Array [23] aims to improve the performance of copy-on-write B-trees. However, these solutions focus on data on disk, which is subject to very different performance tradeoffs than in-memory data. In particular, PART relies heavily on adaptive radix trees which leverage low-latency random access to exploit sparsity and aggressively compress low fan-out nodes. Additionally, the Stratified Doubling Array exposes a versioned interface but internally uses mutation to update data in memory and on disk, requiring locking to support checkpointing and concurrent reads.

An important motivation for PART is incrementally iterative algorithms, which make repeated fine-grained updates to algorithm state. For example, to compute connected components on a distributed graph, vertices repeatedly exchange component ids and update their own membership. Stratosphere [9] and Naiad [17] address these algorithms using a specialized programming model based on state mutation for maximum performance. However, this comes at the expense of simple fault tolerance. Naiad uses costly synchronous checkpoints for fault tolerance, while Stratosphere uses a complex distributed snapshot protocol. In contrast, PART retains fully asynchronous checkpoints thanks to immutability, and it additionally benefits from the space savings of incremental checkpointing.

7 Conclusion

In this paper we introduced persistent adaptive radix trees (PART) to enable batch analytics systems to efficiently support the fine-grained point updates commonly found in incremental and streaming computation while preserving immutability and all the benefits it confers. We leveraged developments in persistent data structure design to enable point queries and updates while preserving past versions with minimal memory overhead. We extended adaptive radix trees to support the batch ingest and bulk scans commonly found in batch analytics workloads. Finally, we enabled asynchronous low-overhead incremental checkpointing by exploiting the tree structure of PART. We demonstrated that PART substantially outperforms, often by orders of magnitude, existing widely adopted mechanisms to introduce point updates in batch analytics systems and performs comparably to mutable data structures while preserving the benefits of immutability.

More generally, we believe that immutability is a beneficial constraint in distributed systems because it improves programmability for the user and flexibility for the system. This will increasingly become the case as applications grow more complex and coordination becomes more expensive.

Finally, PART was developed in context of the recent movement towards increasing integration between batch and online analytics at the system level. This trend, which includes popular new systems such as Spark Streaming [25], Naiad [17], and Flink (formerly Stratosphere) [9], is driven by the promise of improved convenience, increased scalability, and reduced latency between data and resulting action.

References

[1] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide*, chapter The Power of B-trees. O'Reilly Media, Inc., 1st edition, 2010.

[2] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, 2015.

[3] P. Bagwell. Ideal hash trees, 2001.

[4] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, Dec. 1983.

[5] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC*, 1986.

[9] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *VLDB*, 5(11):1268–1279, July 2012.

[10] P. Helland. Immutability changes everything. In *CIDR*, 2015.

[11] V. Henson, M. Ahrens, and J. Bonwick. Automatic performance tuning in the Zettabyte File System. In *FAST*, 2003.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

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

[14] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC*, 2014.

[15] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[16] N. Marz. Trident: a high-level abstraction for real-time computation. https://blog.twitter.com/2012/trident-a-high-level-abstraction-for-realtime-computation.

[17] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[19] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[20] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *PPoPP*, 2012.

[21] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, Aug. 2013.

[22] H. Taghavi, P. Padmanabhan, D. Tsai, F. Z. Siddiqi, and J. Basilico. Distributed time travel for feature generation. http://techblog.netflix.com/2016/02/distributed-time-travel-for-feature.html, 2016.

[23] A. Twigg, A. Byde, G. Miloś, T. Moreton, J. Wilkes, and T. Wilkie. Stratified B-trees and versioned dictionaries. In *HotStorage*, 2011.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.