Improving Cassandra's performance with byte order and tries

Branimir Lambov, DataStax
ApacheCon 2022

Motivation/Background

Cassandra relies heavily on comparison-based structures and algorithms:

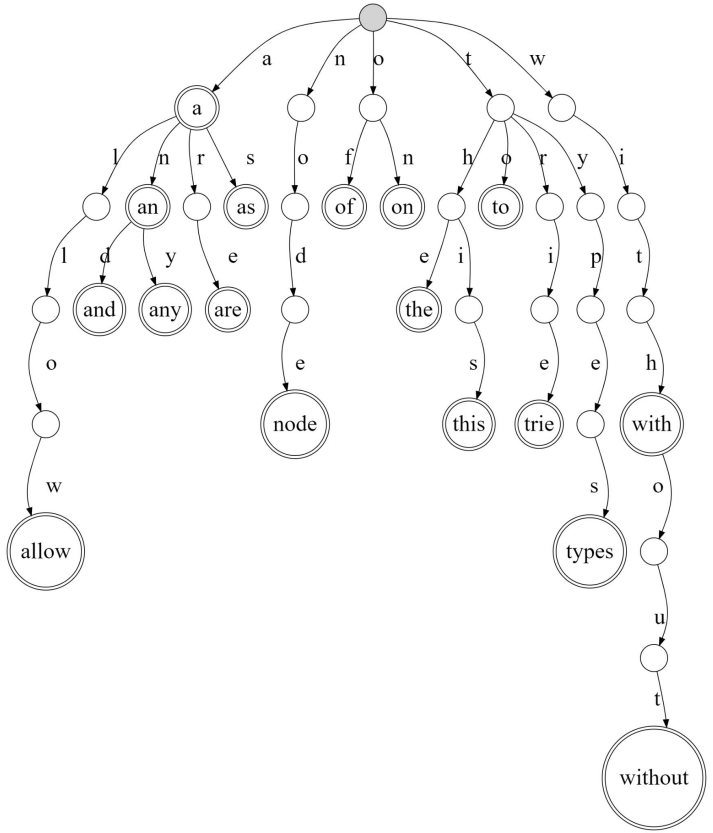
- SkipList + BTree memtables
- Search in summary + primary index
- MergeIterator for compaction and constructing results

Comparisons are inefficient

- Comparisons require full keys in comparable forms
  - Deserialization / pointer hop
- Keys can be long and their length varies
  - Difficult to package / inefficient to cache
- Repeated prefixes vs. map hierarchies
  - More work/space vs. more complex code
- O(k) multiplier on all operations
  - O($k \cdot \log n$) insertion/lookup
  - O($k \cdot n \cdot \log m$) compaction

Byte-comparable keys and tries

- Lexicographically comparable
- Prefix differences define order
- Data structures can take advantage:
  - Prefix sharing
  - Limited node size
  - O(*k*) lookup/insertion
  - O(*N*·log *m*) compaction



Byte-comparable representations (CASSANDRA-6936)

- Typed value <-> byte-comparable representation
- Sequence of bytes which lexicographically compare like the typed value
- For example:
  - Unsigned integers: as is
  - Signed integers: flip sign bit
  - IEEE floating point: flip sign bit, flip all other bits if negative
  - UUIDs: move bits around
- Flat multi-component keys

Byte-comparability applications

- Trie memtable
- BTI (“Big Trie-Indexed”) SSTable format:
  - Trie-based partition index
  - Trie-based row index

Legacy Memtables

- A hierarchy of comparison-based maps
  - Partition map is a concurrent skip list
  - Row maps and below are B-Trees

- Data may be stored off heap
- All maps (organizing structures) are on-heap
  - On-heap size dominates
  - Complex on-heap structure
  - Churn of objects with medium-term lifecycle

New memtable solution (CEP-19/CASSANDRA-17240)

- Byte-comparable key translation
- Partition map using a custom byte trie
  - Typed nodes with pointer tagging
  - Internal memory management with fixed block size
  - Traversal, merging and slicing using a “Cursor” paradigm
  - Single writer, multiple concurrent readers
- Sharding

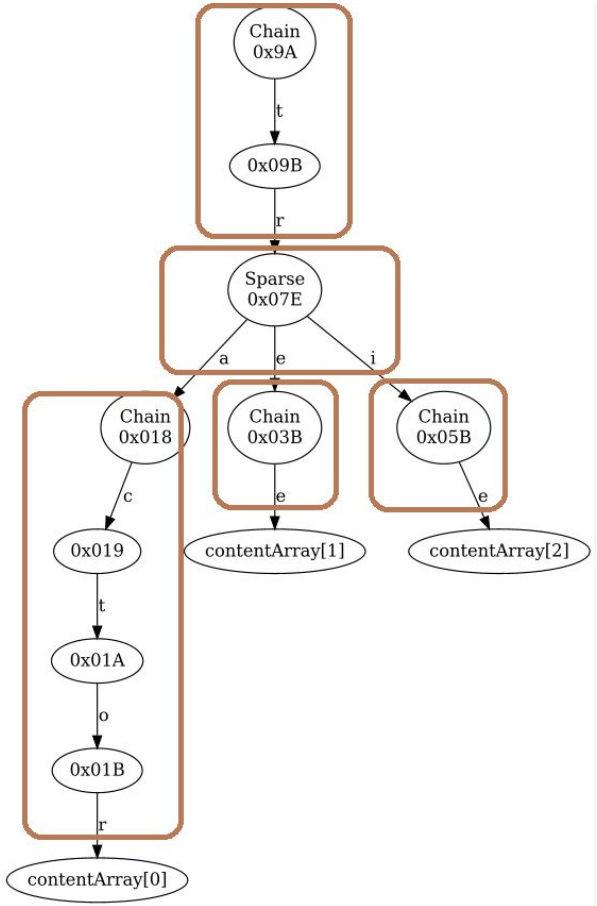
Typed Trie Nodes

- Dense
  - Lots of children, often consulted, pointer arithmetic
- Sparse
  - Few children, search in children list
- Chain
  - Single child sequences, comparison
- Leaf
  - No child, data pointer
- Prefix
  - Add data to node with child

Fixed Block Size

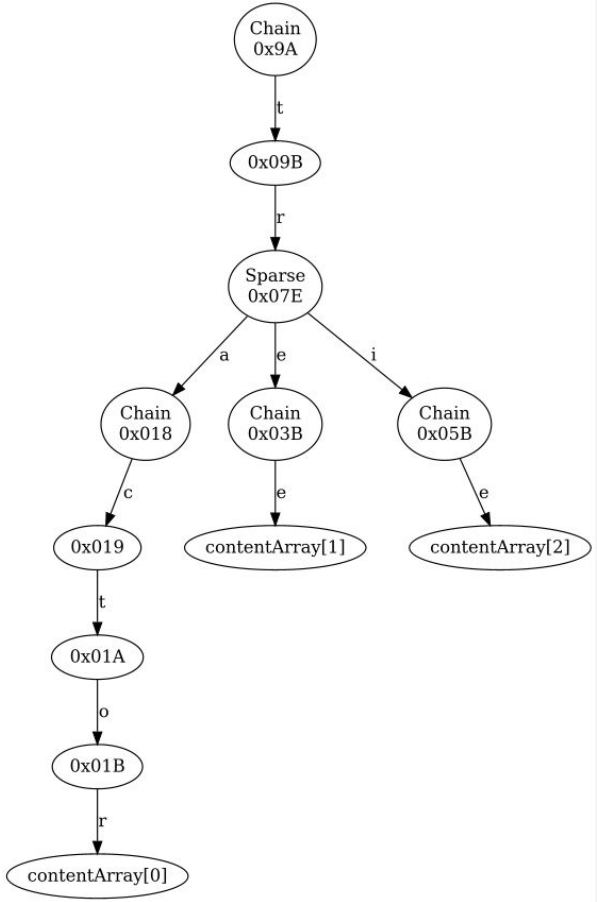
- Split dense nodes into 2-3-3 bit subtransitions
- Combine up to 28 chain nodes into one block
- Put content prefixes in unused space

- Achieves:
  - Cache efficiency
  - Memory management simplicity



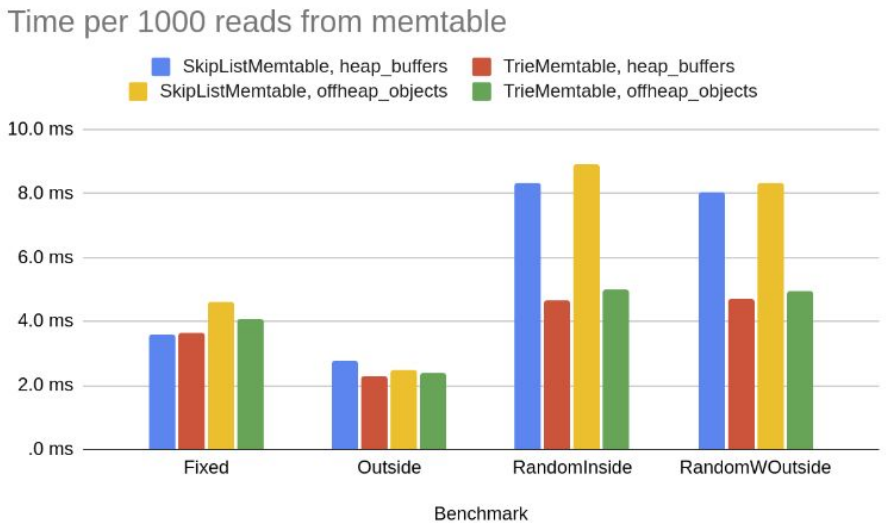
Pointer tagging

- Use pointer bits to identify:
  - Type of node
  - Exact node in a chain
- No space needed for leaf nodes



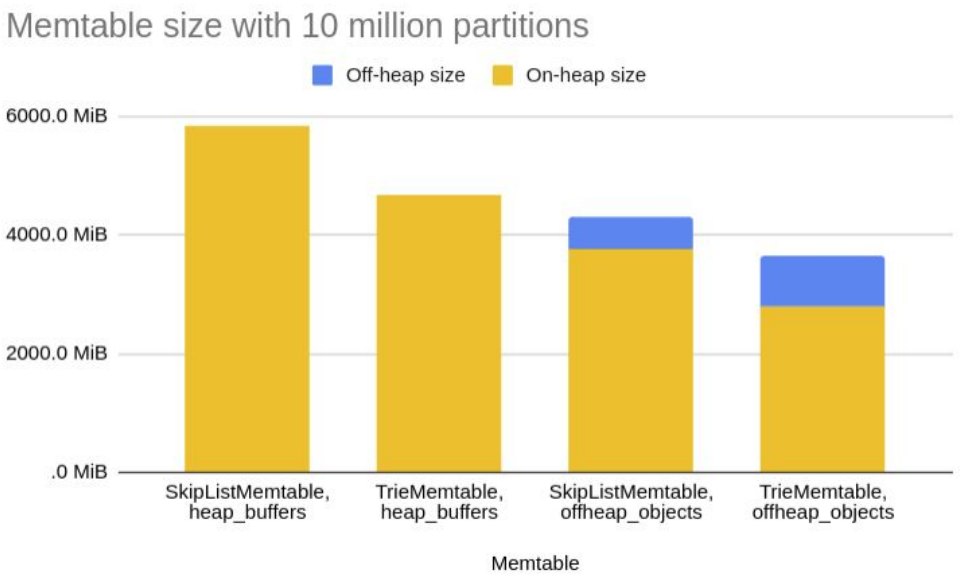
Microbenchmark summary

- Trie vs. skip-list on 10,000,000-entry key-value memtable:
  - 1.8x times faster random reads
  - 2.5x faster single-threaded insertion



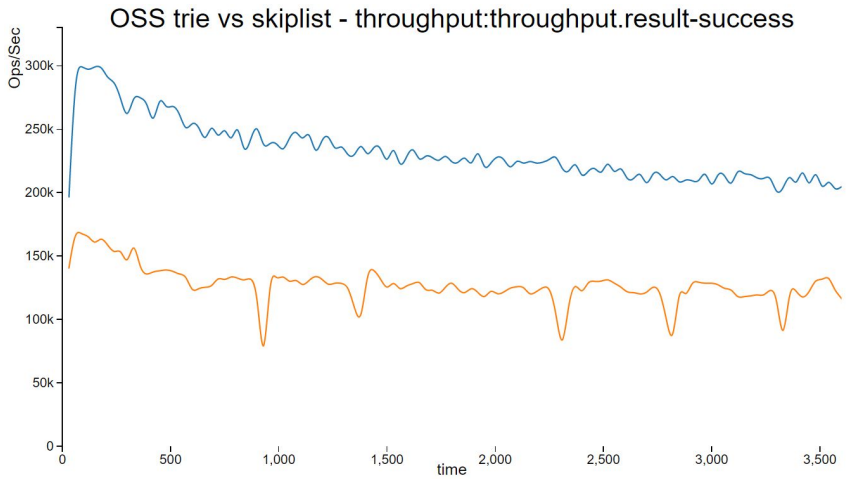
Microbenchmark summary

- Trie vs. skip-list on 10,000,000-entry key-value memtable:
  - 25-35% more data for the same on-heap size



Short-term throughput in Apache Cassandra

10% reads, key-value workload using NoSQLBench / fallout / i3.4xlarge

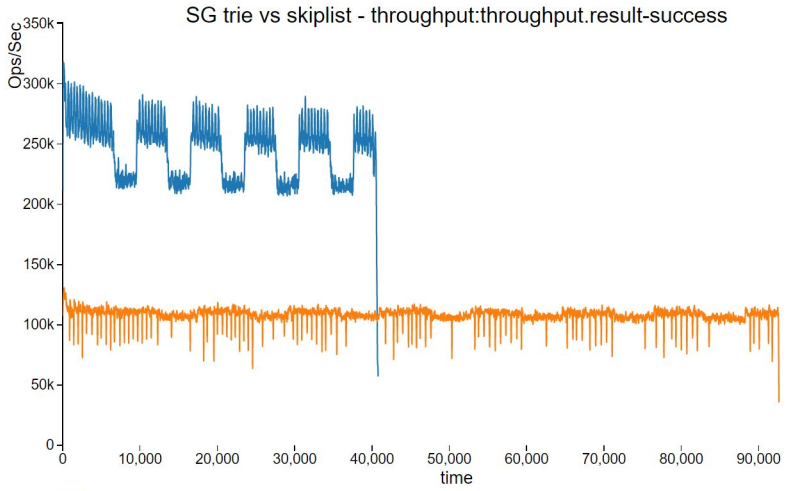


- 1.8x higher burst throughput

Sustained performance with improved compaction

Using datastax/cassandra : ds-trunk

- >2x higher sustained throughput
- ~30% latency reduction at fixed rate (10 and 50% read workload @110k ops/s)
- ~30% more data per memtable flush
- 2.5x less total garbage collection time



Legacy partition index (BIG format)

- Binary search in in-memory index summary
- Linear search in sorted index file
  - Includes deserialization and decoration
  - Has to skip over row index
- Needs key cache

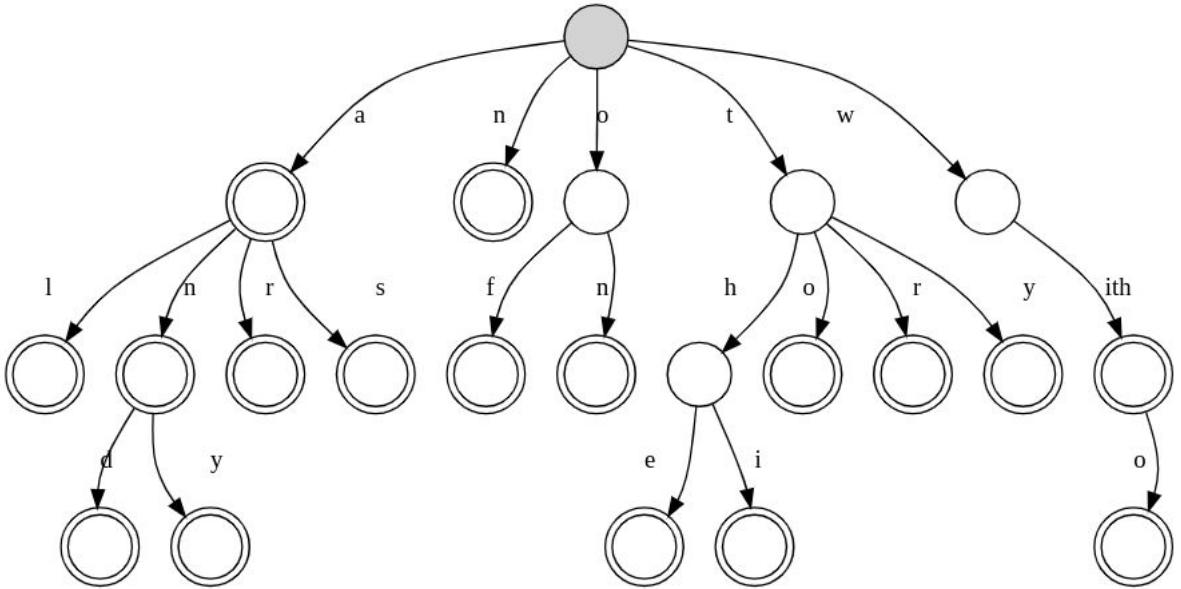
Trie-based partition index (BTI format)

- Token, partition key encoded as byte-ordered sequence
- Trie stores unique prefixes
  - Typed nodes, sized pointers
  - Written page-packed
- Points to data or row index file
- Includes hash bits

Unique prefix

Only store up to unique prefix and rely on full key in data file to check full match.

Store and check hash bits to minimize reading data file on mismatch.

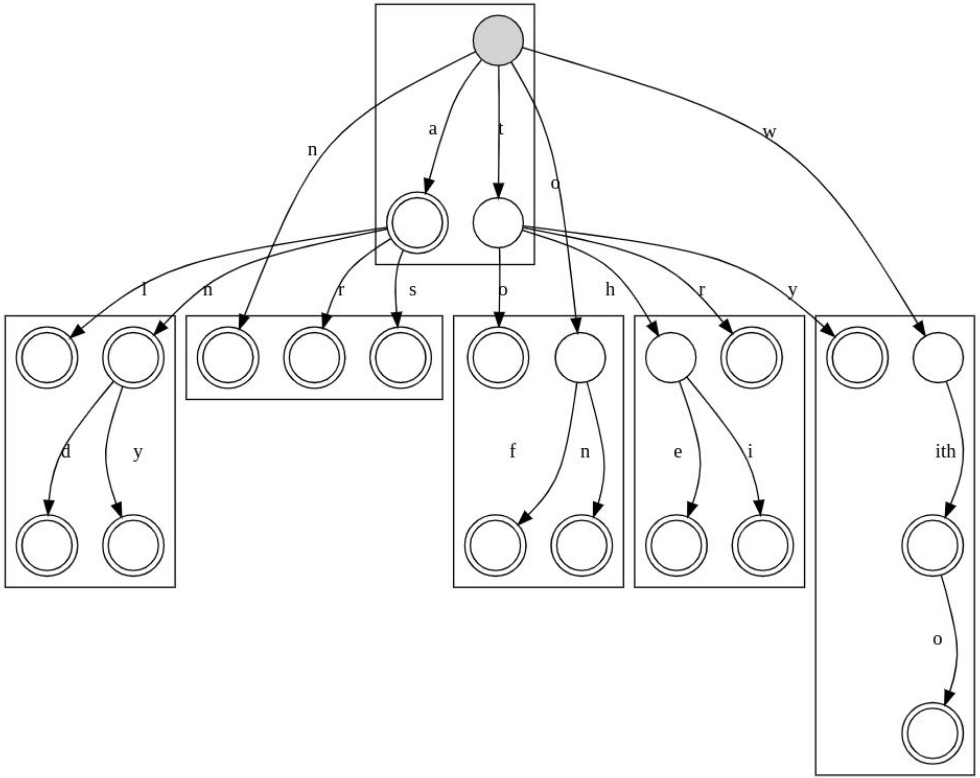


Page packing

Include as much substructure as possible inside disk page.

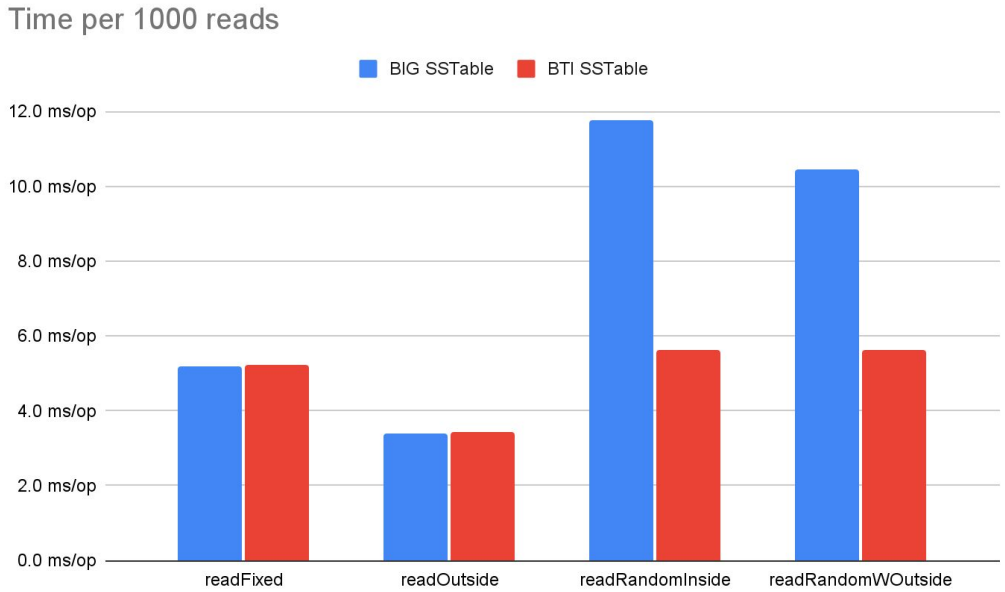
Only write page once branch is greater than page.

Treat pointers to placed pages like leaves.



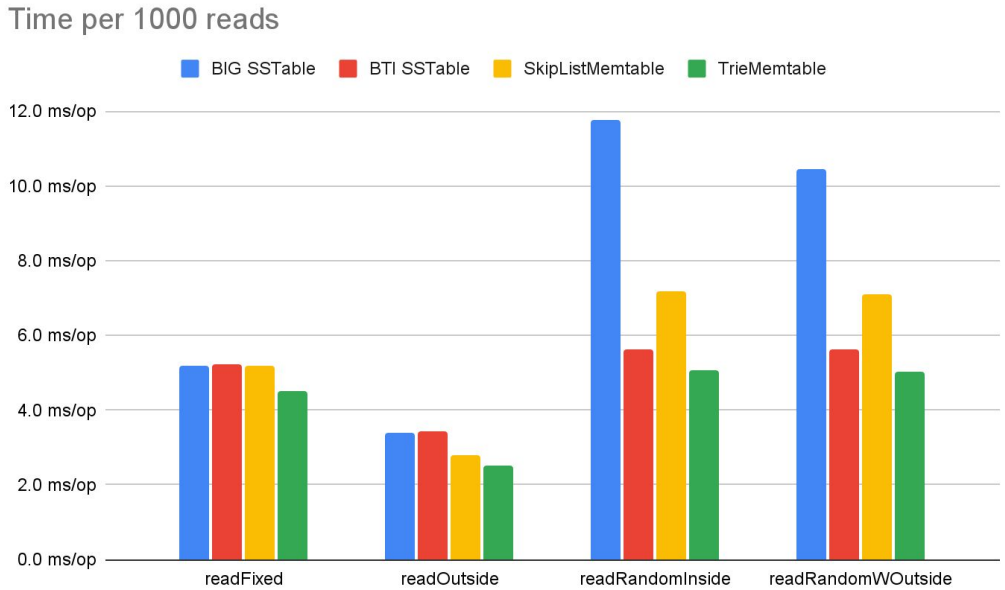
Key-value microbenchmark

- ~2x faster random reads in microbenchmarks
- ~30% smaller index size
- ~10% latency reduction on 1TB fixed-throughput test



Partition index performance

- ~2x faster random reads in microbenchmarks
- ~30% smaller index size
- ~10% latency reduction on 1TB fixed-throughput test



Operational benefits

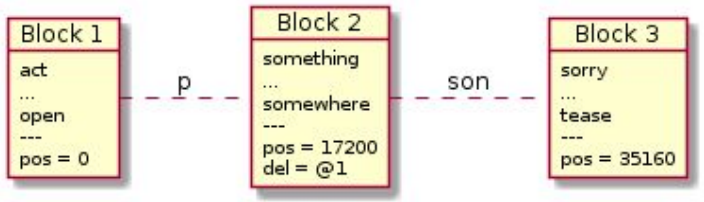
- No key cache
  - Cached by chunk cache / OS page cache
- No index summary
  - Small non-leaf page set stays cached
- No key deserialization
  - Much less object churn during queries
- No skipping over row index during search
  - Typically a single leaf page fetch

Legacy row index (BIG format)

- Binary-search in index of blocks of row keys
  - Page fetch per comparison or deserialization of the whole row index to memory
- Block is formed at given granularity (64kb by default)
  - Block needs to be large
- Linear search within block
- Full deserialization of block on reversed queries

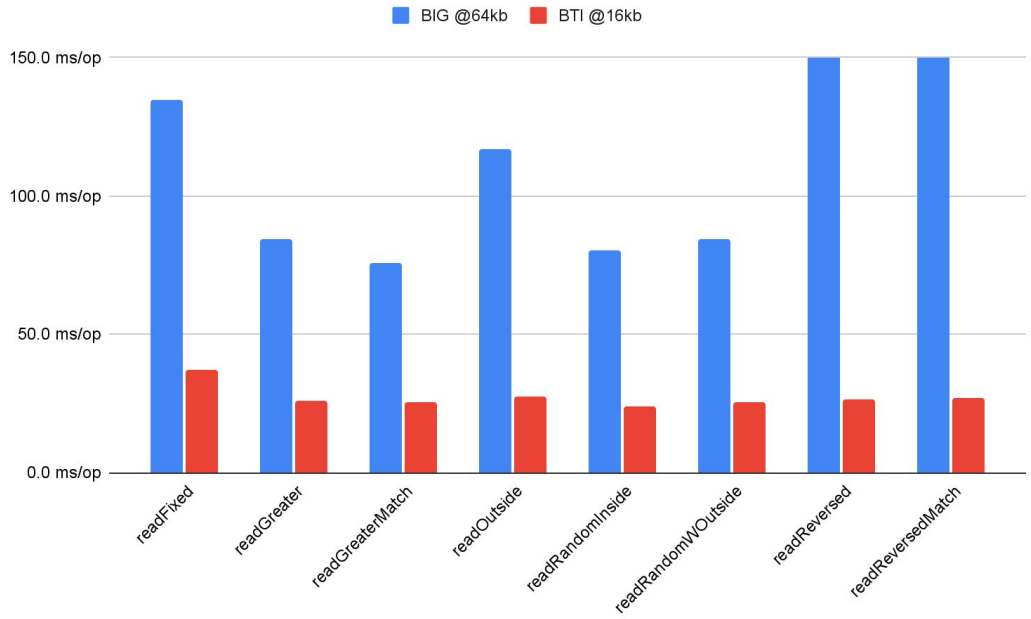
Trie-based row index (BTI format)

- Clustering key encoded as byte-ordered sequence
- Index blocks at given granularity
- Stores “separator” between index blocks
  - Key ≥ separator means no block before can contain entry
  - Cut off at first different byte
- Plus deletion information



Row index performance

- 3-4x more compact index → better index granularity → close to 4x reduction in latency
- Improved reversed queries
- 4/2/1 kb column_index_size is useable
- As well as 0kb (full indexing)



Future work

- Memtable trie to rows and cells
- Trie-based PartitionUpdate in commit log
- On-disk TrieTables
- Compaction and retrieval with trie cursors
- Partition-level segregation of tombstones from data

Thank you!