TRINITY: A Fast Compressed Multi-attribute Data Store

Ziming Mao*
UC Berkeley

Kiran Srinivasan
NetApp

Anurag Khandelwal
Yale

Abstract

With the proliferation of attribute-rich machine-generated data, emerging real-time monitoring, diagnosis, and visualization tools ingest and analyze such data across multiple attributes simultaneously. Due to the sheer volume of the data, applications need storage-efficient and performant data representations to analyze them efficiently.

We present TRINITY, a system that simultaneously facilitates query and storage efficiency across large volumes of multi-attribute records. TRINITY accomplishes this through a new dynamic, succinct, multi-dimensional data structure, MDTRIE. MDTRIE employs a combination of novel Morton code generalization, a multi-attribute query algorithm, and a self-indexed trie structure to achieve the above goals. Our evaluation of TRINITY for real-world use-cases shows that compared to state-of-the-art systems, it supports (1) 7.2–59.6× faster multi-attribute searches, (2) storage footprint comparable to OLAP columnar stores and 4.8–15.1× lower than NoSQL stores and OLTP databases, and (3) point query throughput comparable to NoSQL stores and 1.7–52.5× higher than OLTP databases and OLAP columnar stores.

ACM Reference Format:
Ziming Mao, Kiran Srinivasan, and Anurag Khandelwal. 2024. TRINITY: A Fast Compressed Multi-attribute Data Store. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3650072

1 Introduction

Recent years have witnessed the proliferation of attribute-rich data generated from automated sources, or "machine-generated data" [1]. IoT devices already stream as many as 1.4 million data points per second per backing server [2–5]. Facebook's monitoring systems used for operational metrics like CPU, memory, and disk utilization generate around 12 million data points per second per server [6–9]; and network traffic logs for 100Gbps links can generate as many as 160 million data points per second at each server [10–12]. Such data is a rich source of information, offering utility to a wide range of monitoring, analysis, and visualization applications. However, the timeliness of such analysis is crucial; the value of the analysis is often highest shortly after the data is ingested, making the ability to query massive volumes of attribute-rich data in near real-time of paramount importance.

As a concrete example, consider a taxi service that collects information about rides made in a busy city like New York [13–16]. A subset of metadata recorded for each ride includes pickup and drop-off locations, times, trip durations, and fares. With up to 30 million rides per day [17], careful time-sensitive planning of car availability and fare pricing is required, particularly in locations with high demand. For instance, Uber accumulates hundreds of petabytes of analytical data for forecasting rider demand and addressing driver-client matching, with target data latency on the scale of minutes [14]. Facilitating this demands complex spatiotemporal analysis of the ride data along multiple attributes simultaneously, *e.g.*, tracking the number of trips with pickup and drop-off locations at popular locations at regular time intervals.

The unique characteristics of analyzing machine-generated data impose challenging requirements on the underlying data stores. On one hand, these systems should support low-latency, high-throughput queries across attribute-rich data records. These queries can be quite diverse; point queries involve fast record insertions and lookups using a unique primary attribute, while more complex analyses require simultaneous searches across multiple secondary attributes. On the other hand, supporting performant analysis over massive data scales requires holding large volumes of data in memory, making minimizing the storage overheads crucial.

Unfortunately, existing systems compromise on one or both of these requirements. Systems that support efficient query semantics across multi-attribute records leverage additional multi-attribute index structures with large storage overheads [18–24]. While such systems perform well for small datasets, they cannot scale to larger datasets as the underlying data structure easily outgrows the memory capacity — *e.g.*, indexes alone can consume 55% of the total memory in real-world in-memory databases [25]. Another emerging class of systems leverages queries on compressed in-memory data structures [26–29] to scale to larger datasets. However, they do so by compromising functionality, *e.g.*, they lack support for either efficient point queries [29, 30], fine-grained in-place updates [26–28], or multi-attribute queries [26–29].

*Work done while at Yale.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04
https://doi.org/10.1145/3627703.3650072

Maintaining multiple secondary indexes in addition to the compressed data can support the missing functionality but incurs additional storage overheads, which ultimately limits performance scalability (§2).

We present TRINITY, an in-memory data store that simultaneously supports fast point queries and multi-attribute range searches on a storage-efficient data representation. To achieve this, TRINITY argues for a fundamentally new approach to multi-attribute data representation. This is motivated by our observation that with the drastic growth in machine-generated data, it is increasingly infeasible to store and analyze data in memory, even in its native form, let alone with additional indexes for real-time analysis. As such, unlike prior approaches, TRINITY natively supports rich queries on a dynamic *succinct*[1] data structure, MDTRIE, with no additional indexes. MDTRIE encodes different attributes of a record using a *space-filling curve* [32] — specifically, a novel generalized form of Morton code [33] — and stores them succinctly in a trie.

Moreover, MDTRIE is *self-indexed*: our indexes are not separate data structures, but instead embedded within the same succinct trie that encodes the data. This approach effectively allows us to perform multi-attribute queries directly on a *compressed* data representation. We also present several optimizations that exploit the structure of data representation in MDTRIE, making it far more performant and space-efficient than prior multi-attribute indexes (§5).

We incorporate MDTRIE into TRINITY, a distributed, fault-tolerant data store (§4). We evaluate TRINITY for three real-world use-cases; compared to state-of-the-art systems, TRINITY supports (1) 7.2–59.6× faster multi-attribute searches, (2) storage footprint comparable to columnar stores, and 4.8–15.1× lower than NoSQL and OLTP databases, and (3) point query throughput comparable to NoSQL stores and 1.7–52.5× higher than OLTP databases and columnar stores (§5).

In summary, this paper makes three main contributions:

- Design of MDTRIE, a dynamic, succinct, self-indexed multi-dimensional data structure that supports multi-attribute range searches as well as efficient point queries.
- Design and implementation of TRINITY, a distributed data store that leverages MDTRIE to store and query massive multi-attribute datasets using a data-parallel architecture.
- Evaluation of MDTRIE and TRINITY against state-of-the-art data structures and data stores on real-world workloads.

We note that our data representation makes some trade-offs to achieve storage and query efficiency; specifically, TRINITY's Morton code-based representation requires queried attributes to have fixed bit-width representations. Fortunately, this is acceptable for our target applications for machine-generated data since they tend to work mainly with fixed-width attributes (§5). Extending our representation to support variable-length attributes is an exciting future work (§6). The code for TRINITY and the datasets and workloads used in our evaluation are available at `https://github.com/Trinity-data-store/Trinity`.

[1]Formally, if the information-theoretic lower bound of a given dataset is n bits, then a succinct data structure can represent it in $n + o(n)$ bits [31].

2 Background and Motivation

This section describes the need for storing and analyzing multi-attribute data (§2.1), the shortcomings of existing techniques in meeting this need (§2.2), and the approach that TRINITY employs to overcome these shortcomings (§2.3).

2.1 Multi-Attribute Data Storage

Machine-generated data is typically composed of a collection of *structured* records, with the records themselves composed of multiple well-defined attributes. One of these attributes is a unique identifier for the record — the *primary* key — while the remaining attributes are secondary. This data is often stored in NoSQL stores [34–37] or in-memory databases [38–41], which use the primary key to support point queries like lookups, insertions, and removals (via `get()`, `put()`, and `delete()` operations, respectively). Secondary attributes, on the other hand, are employed to support range and filter queries, *e.g.*, isolating all the events that may have occurred in a specific time window for an IoT application. Increasingly, applications that use such data for monitoring, analysis, and visualization require querying multiple attributes *simultaneously*, *e.g.*, spatiotemporal queries on vehicular data for real-time traffic applications (§1).

We identify three key requirements for storage systems imposed by applications operating on multi-attribute data:

- Support for **low-latency multi-attribute queries** (on the order of hundreds of milliseconds [42–45]) to enable rich monitoring, analysis, and visualization queries. For instance, §5 outlines three emerging use cases where multi-attribute queries are crucial to different applications.
- **Space-efficient representation of data** — ideally comparable to or lower than the original dataset size — to maximize the amount of data that can be stored in memory for low-latency queries at scale [26–29].
- Support for **high-throughput point queries**, i.e., record lookups, insertions, and removals based on their primary keys (on the order of thousands of operations per second per core [46, 47]). These operations enable high data ingestion rates and efficient retrieval for records of interest [2–12].

2.2 Need for TRINITY

Unfortunately, we find that existing approaches fail to meet one or more of the requirements outlined above:

Lack of support for low-latency multi-attribute queries. Multi-attribute queries are challenging to support efficiently; while scan-based approaches do not scale to large datasets, building indexes for multiple attributes incurs high storage overheads while querying and aggregating results from multiple indexes is slow. In-memory databases [38–41] typically

decompose collections of multi-attribute records into normalized relational tables to reduce data redundancy and employ *joins* to support multi-attribute queries. Although they are performant for small datasets, joins can be quite inefficient for larger datasets due to their quadratic computational complexity [48]. This inefficiency is exacerbated in distributed databases, where the joins require frequent cross-server communications [49]. Additional index data structures like B-Trees or hash-indexes can accelerate queries on secondary attributes. While these indexes perform well for single-attribute queries [50], their performance for multi-attribute queries degrades with the number of attributes queried simultaneously and the volume of data being queried. In TRINITY, we target a new approach that maintains low query latencies while significantly lowering storage overheads.

High storage overheads. The use of traditional indexes for secondary attributes in addition to the raw data incurs poor performance for multi-attribute queries and requires significant additional storage. Prior work has shown that systems that use indexes to support such queries can increase the required storage for an input dataset by as much as 8× [26]. There have also been efforts to design tree-based data structures for supporting *multi-dimensional*[2] queries such as K-D trees [22], R-trees [23, 51], UB-trees [21], and Qd-tree[52]. Unfortunately, these structures are not space-efficient, requiring far more storage relative to the performance improvements they offer (§5). Moreover, the storage overheads often result in the data structure spilling over to secondary storage, which results in further degradation in query performance.

Lack of support for high-throughput point queries. To enable space efficiency, a recent class of systems focuses on enabling queries on compressed data structures [26–29, 45]. For example, Succinct [26] supports point queries and searches on secondary attributes while keeping the data compressed. Columnar stores like ClickHouse [53] support compression on data blocks (*e.g.*, with LZ4) and employ fast decompression during query execution. Although these approaches provide space efficiency and query efficiency for limited types of search, they compromise either on efficient in-place point updates [26–28, 42, 54] or on point lookups [29, 53].

2.3 The TRINITY Approach

To address the issues above, we designed TRINITY, a distributed data store that supports low-latency multi-dimensional queries and high-throughput point queries on space-efficient data representation. TRINITY achieves this through a new data structure — MDTRIE. MDTRIE combines advances across multiple data structure techniques with novel data layout and query execution algorithms to meet the needs of applications that leverage multi-attribute data (§3):

[2]We use the terms dimension and attribute interchangeably: the two are standard terms in theory and storage communities, respectively.

Encoding attributes with Morton codes. To efficiently support range queries across multiple attributes simultaneously, we collapse multiple attributes into a single attribute using an encoding scheme called Morton codes [33]. The benefit of using Morton codes is twofold. First, they preserve *spatial locality*, i.e., contiguity in ordering across the multiple component attributes even when translated to a single attribute. In other words, points closer in multi-dimensional space are mapped to numerically closer Morton codes. This readily permits efficient adaptations of range query algorithms designed for single attributes to multiple attributes. Second, Morton codes can encode any attribute with a fixed-width binary representation, making them applicable to data types seen in machine-generated data sources. Although traditional Morton codes require all attributes to have the same width, we provide a novel generalization for Morton codes to support attributes of different widths (§3.2). While there are other space-filling curves with similar properties [55], MDTRIE employs Morton encoding since it permits efficient pruning of multi-dimensional search spaces.

Encoding records in a succinct trie. A naïve approach to support range queries on the Morton-encoded values is to store them in a tree-based data structure [30, 56, 57], *e.g.*, represent each bit in the Morton-encoded value as one level in the tree. However, trees typically incur large storage overheads due to their heavy use of pointers — even more so when applied naively to Morton-encoded data, making them impractical for large datasets. Our second contribution is to leverage succinct trie structures [58–60], which exploit prefix redundancy across data points along with bit vector representations of tries to reduce the storage overheads significantly. The use of such tries to store Morton codes, however, introduces subtle issues for supporting multi-dimensional range queries, point queries on primary keys, and maintaining storage efficiency, especially when the number of attributes in the dataset grows large; to the best of our knowledge, none of the prior approaches [60, 61] address these issues. We make several algorithmic and data structure innovations to improve query execution and storage efficiency in our succinct MDTRIE.

Self-indexing for point queries. Since Morton codes interleave bits of multiple attributes of the original record, it makes *in-place* support for point insertions or lookups based on the primary key more challenging. This is further exacerbated when these values are compressed in a succinct trie representation, where nodes in MDTRIE cannot be directly referenced with pointers. Using additional indexes to support point queries adds to the storage overhead and requires accessing and updating additional data structures, incurring performance overheads. Instead, MDTRIE embeds metadata *within* the trie structure itself and reconstructs data points through hardware-accelerated bitwise traversals up the trie to support point queries efficiently with minimal storage overheads. Interestingly, this also ensures that MDTRIE remains a

self-index — a data structure that combines both the raw data and index within the same succinct representation.

We incorporate MDTRIE into TRINITY's distributed system architecture by using well-studied sharding techniques (§4). We also incorporate techniques for efficient distributed range query execution and providing persistence guarantees.

3 MDTRIE: Encoding Multi-Attribute Data

We now describe MDTRIE data structure for storing and querying multi-attribute data. We first outline prior techniques that MDTRIE builds on and their shortcomings in meeting TRINITY's goals (§3.1). We then describe salient components of MDTRIE and how they systematically overcome these shortcomings, including a generalization of Morton encoding to variable bit-width attributes (§3.2), a novel multi-dimensional range search algorithm (§3.3) and efficient support for point queries (§3.4).

3.1 Building Blocks

MDTRIE builds on a rich body of theoretical work on data structure design; we summarize relevant techniques here.

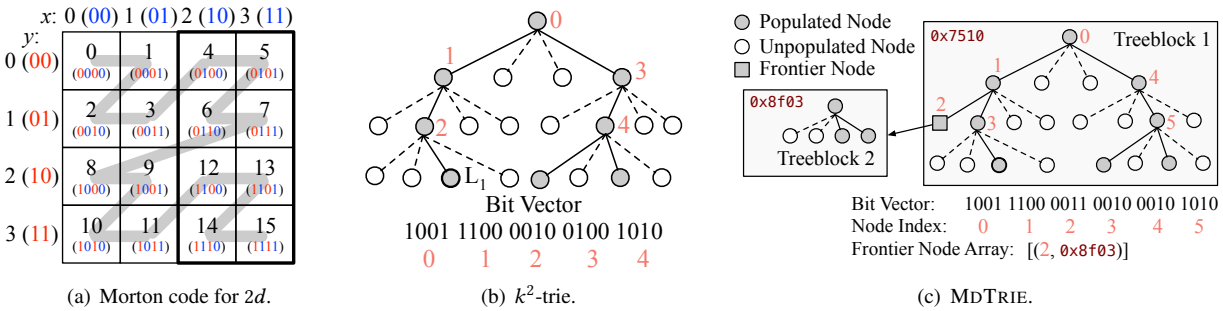
Morton code is an encoding scheme used to map n dimensional data to a single dimension while preserving the spatial locality of the encoded data points. This is achieved by ordering the data along a Z-order space-filling curve in n dimensions. Figure 1(a) shows an example of 2D space, where points with integer coordinates can be represented on a square grid: x and y correspond to column and row indices, respectively. The Morton code (4-bit code in each cell) for a point (depicted as cells) is obtained by interleaving the bits of the y and x dimensions, *e.g.*, the Morton code for $y = 1$, $x = 2$, or `01`, `10` in binary, is 6 (`0110` in binary). The Z-order space-filling curve (gray line) tracks the numerical order of Morton code values across points; it possesses a characteristic recursive Z-shape. This approach can be generalized to n dimensions by interleaving each of their bits.

Reducing the n-dimensional data to a single dimension using Morton codes permits using traditional tree structures to support efficient range searches across multiple dimensions simultaneously. For instance, the range $2 \le x \le 3, 0 \le y \le 3$ maps to two contiguous Morton code ranges (4–7 and 12–15), with the codes preserving much of the spatial locality. Algorithms for identifying these contiguous Morton codes corresponding to the multi-dimensional range queries [62] have been adapted to tree-based data structures [21, 30] for fast searches. Unfortunately, current approaches suffer from two main issues. First, these data structures either do not exploit the redundancy in the encoded data, aggressively employ pointers to support efficient traversals, or both. Consequently, such approaches are quite space-inefficient. Second, although these data structures are optimized for multi-dimensional range queries, they often cannot efficiently support queries on a small subset of attributes or point queries on primary keys; supporting point queries requires additional indexes, exacerbating their space inefficiency.

Succinct tries. Although tries can exploit redundancy across key prefixes to reduce storage overhead in the tree-based data structures outlined above, they still aggressively use per-node pointers to facilitate efficient traversals, limiting their space efficiency. Succinct tries [60] permit overcoming this hurdle: although a general tree of n nodes represented in pointer form requires $O(n \log n)$ bits, succinct representations typically require just $2n + o(n)$ bits [31]. At a high level, succinct trie representations reduce storage overheads through a bit-encoded representation for nodes and edges rather than using pointers to link nodes. More concretely, a node of degree d is represented as a d-bit vector, each bit indicating whether the corresponding child exists. To avoid storing pointers to child nodes, the bit vectors for different nodes are laid out one after the other in a well-defined order, *e.g.*, depth-first ordering (Figure 1(b)). While there are alternate layouts (§ 6), we use depth-first ordering for its cache- and update-efficiency [61].

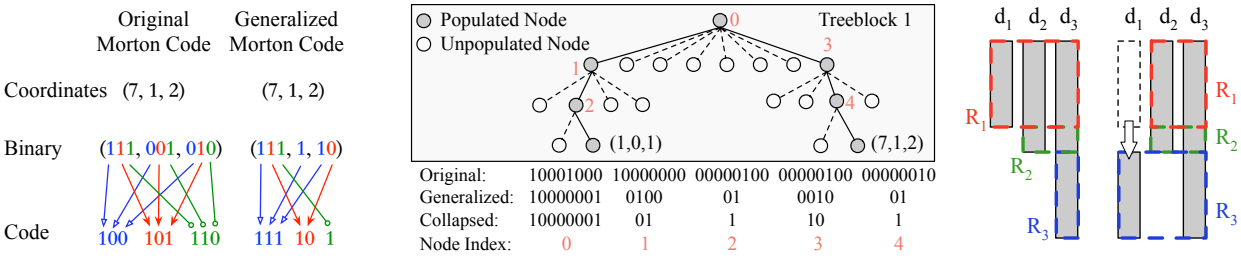
k^2-trie. Although much of the earlier work has focused on single-dimensional tries, recent work [61] explores the design of a dynamic succinct trie of Morton codes to represent two-dimensional data. Their succinct representation, k^2-trie is a 4-ary tree, where each node has four possible children encoded as four bits; node 0 in Figure 1(b) is encoded as `1001` as only its first and fourth children exist. Each edge to a child node tracks two bits of Morton code, one per dimension (`00`, `01`, `10`, and 11). Put together, k^2-trie tracks a contiguous sequence of nodes in depth-first order in a bit vector. The path from the root to a leaf represents the full Morton code of the corresponding data point. In Figure 1(b), the Morton code for leaf L_1 (`000110`) is the concatenation of Morton code bits tracked by the edges (`00`, `01`, `10`) along the path. The authors suggest k^2-trie can be generalized to n dimensions by letting each edge track n bits of Morton code, one bit per dimension. This would require storing 2^n bits per node to track the 2^n possible values that n bits of Morton code can assume.

Unfortunately, k^2-trie fails to meet several of our requirements. First, k^2-trie only supports insertion and checking for a point's existence, with no support for either multi-attribute range queries or point queries on primary keys. Second, k^2-trie's storage grows exponentially with more attributes since it must encode each node using 2^n bits, where n is the number of dimensions — a single 10-dimensional data point with 32-bit attributes would require 4KB ($2^{10} \times 32$ bits) of storage! Third, k^2-trie requires *all* attributes to have the same bit width, limiting its applicability to datasets with attributes of different types. We use k^2-trie as a starting point in our MDTRIE design and address its shortcomings, as described in the following sub-sections.



(a) Morton code for $2d$. (b) k^2-trie. (c) MDTRIE.

Fig. 1. Using Morton codes in k^2-trie and MDTRIE (a) Morton code (§3.1, values in cells) is a bitwise interleaving of the y (red) and x (blue) values. Z-order curve (gray line) tracks contiguous Morton codes; (b) Nodes in k^2-trie (§3.1) use 4 bits to encode their children and are stored in depth-first order in a bit vector. The four edges of each node track two bits of Morton code (`00`, `01`, `10`, `11`). Labels on nodes identify their node index in depth-first order; (c) MDTRIE (§3.2) partitions the trie into treeblocks, each tracking a contiguous chunk of nodes in depth-first order. The frontier node array in each treeblock tracks (node index, treeblock pointer) tuples; pointer addresses are shortened for illustration.



(a) Original vs. Generalized Morton encoding (b) MDTRIE w/ Generalized Morton encoding + Collapsed nodes (c) Staggered Morton encoding

Fig. 2. Generalizing Morton codes. (a) Contrasting Morton encoding with padding against Generalized Morton encoding; (b) MDTRIE storage representation for original, Generalized Morton encoding and with collapsed node optimization for points `(1,0,1)` and `(7,1,2)`. Generalized Morton encoding reduces the storage required for the treeblock bit vector from 40 to 20 bits, while collapsed nodes further reduce the storage to 18 bits; (c) Staggered Morton encoding for three attributes of different widths. See §3.2 for details.

3.2 MDTRIE Layout

Inserting a new data point in a succinct trie (*e.g.* k^2-trie) requires moving many bits to make space for the new nodes in the bit vector. To scale to larger datasets, MDTRIE (Figure 1(c)) partitions the trie into *treeblocks*, with each treeblock tracking a contiguous chunk of nodes in depth-first order in a bit vector. MDTRIE bounds the cost of bit shifts within a treeblock by ensuring that no treeblock has more than a certain number of nodes (512 in our realization). When this threshold is exceeded, it creates a new treeblock and adds a *frontier node*, or a pointer to a new treeblock, (*e.g.*, node 2 in Figure 1(c)) in the parent treeblock to point to it. The treeblock tracks these pointers in a frontier node array comprising tuples of frontier node index (i.e., its position in the treeblock in depth-first order) and the corresponding treeblock pointer.

We now outline the key components of MDTRIE layout that allow it to scale to a large number of dimensions.

Generalizing Morton Encoding. Real-world datasets contain multiple attributes with different bit widths, depending on their data types. Traditional Morton encoding requires all dimensions to have the same bit-width. A naive approach of padding every dimension to the same width can make the trie quite space-inefficient, especially as the number of dimensions grows large. As an example, consider three attributes with bit-widths of 3, 1, and 2 bits, respectively, encoded using a traditional 3D Morton (Figure 2(a)). If we pad the coordinates along each dimension to 3 bits, we would need to store 8 bits per node for every level in the trie. In particular, the padded zero bits in the second and third attributes only generate Morton codes with redundant zero bits.

Instead, we propose a generalization of Morton codes for dimensions of different widths that *avoids* padding: it simply skips a dimension's bits if all its bits are exhausted. Figure 2(a) shows this approach. In the first interleaving, we extract bits from all three dimensions; in the second interleaving, we extract bits from only the first and third dimensions; and in the third interleaving, we extract only the remaining bit from the first dimension. This approach significantly improves space efficiency for our MDTRIE as shown in Figure 2(b). The first level of MDTRIE in the above example still requires 8 bits per node, but the second and third levels only require 4 and 2 bits per node, respectively. Note that because MDTRIE nodes at different levels track different numbers of dimensions under

our generalized Morton encoding, we refer to the dimensions used at any level of the trie as *active dimensions*.

We introduce two optimizations to the generalized Morton encoding to further reduce the storage overheads for MDTRIE: *collapsed nodes* and *staggered Morton encoding*.

Collapsed nodes. During our preliminary analysis of storage overheads, we found that most MDTRIE nodes only had a single child, especially for datasets with many attributes. Intuitively, this is because the Morton code-space grows exponentially with additional dimensions, while the actual data points in the dataset occupy only a small fraction of the Morton code-space, i.e., real-world datasets are *sparse*. In MDTRIE, if a node with n active dimensions has only a single child, storing 2^n bits to track all possible n-bit Morton code sequences can be quite wasteful as n grows.

As an optimization, we collapse the representation for such nodes from 2^n bits to n bits, directly encoding the Morton code representation of the valid child *by value*. Whenever a new child needs to be added, we simply expand the node's representation back to 2^n bits. We store a separate bit vector per treeblock to track whether or not a node is collapsed.

To illustrate, consider an example where a node originally in its collapsed node representation directly stores the 3-bit Morton code representation of the child (`010`). When a new child (`101`) is to be added, the node is first expanded to 8 bits (`00100000`), with a bit set at child position 2 for the original child (`010`) and a new bit is set for the new child, yielding the final Morton code `00100100`. Figure 2(b) shows the reduction in storage for a complete treeblock using collapsed nodes.

Staggered encoding. An interesting corollary to our Morton code generalization is that we do not need to start interleaving bits for all dimensions from their first bit; Figure 2(c) shows that it is possible to *delay interleaving bits* along selected dimensions (*e.g.*, d_1). In our trie representation, we do not even need to prefix the delayed dimensions with zeros; instead, we can simply calculate the set of *active dimensions* at every trie level (*e.g.*, $\{d_2, d_3\}$ at level 1) during initialization.

Curiously, such staggering can actually improve storage footprint, depending on the dataset and its attribute widths. To see why, consider the example in Figure 2(c). Let a *region* R be a contiguous set of levels with the same number of active dimensions and w_R be the number of active dimensions in R. By staggering the first dimension to the end, we find that w_{R_1} decreases from 3 to 2, but w_{R_3} increases from 1 to 2. Because the number of bits needed per node is 2^w, balancing the number of active dimensions across different levels can improve storage efficiency relative to a more skewed distribution. Our approach to realizing this insight adapts a greedy bin-packing algorithm to minimize the maximum number of active dimensions in any region. We find that datasets with a wide range of attribute widths benefit most from this optimization since there is more room to stagger attribute bits to minimize the number of active dimensions (§5.2).

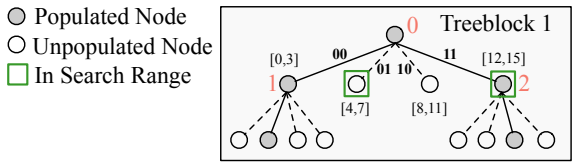


Fig. 3. Two-dimensional range search. See §3.3 for details.

3.3 Multi-dimensional Range Query

While Morton codes simplify multi-dimensional range queries by mapping them to range queries on a single dimension, subtle issues still remain. Despite their tendency to preserve spatial locality, the Z-order curves generated by Morton codes can still have discontinuities within a multi-dimensional search range. For example, the 2D search range $2 \le x \le 3, 0 \le y \le 3$ in Figure 1(a) has one discontinuity at Morton code 7, after which it jumps to Morton code 12. Using a naive range lookup on the Morton codes may require visiting nodes in the MDTRIE corresponding to irrelevant Morton code ranges, *e.g.*, 8–11. Even when the Morton codes for a search range are continuous, there may not be any valid points in the range, *e.g.*, if the range 4–7 does not have any actual points in it, visiting the corresponding node in MDTRIE would be wasteful. Both issues are exacerbated with more dimensions as the search space grows much sparser, incurring significant inefficiencies for multi-dimensional range queries.

Fortunately, the succinct node representation in MDTRIE offers a means to sidestep both inefficiencies. Since MDTRIE tracks how many dimensions n are active and which of the possible 2^n following n-bit Morton code prefixes (i.e., children) are valid at each node, we can quickly eliminate irrelevant and empty Morton code ranges during a search. Revisiting the above example, the MDTRIE root node would track four children corresponding to Morton code bits `00`, `01`, `10` and `11` (Figure 3). Since the search query corresponds to Morton code ranges 4–7 and 12–15, Morton code prefixes `00` and `10` (corresponding to Morton code ranges 0–3 and 8–11, respectively) are irrelevant for our range search query, so we can simply skip traversing the corresponding sub-trees with simple bitwise comparisons. Also, because the Morton code prefix `01` (corresponding to Morton code range 4–7) does not contain any points in the trie (since the second bit of `1001` stored at node 0 is `0`), we can eliminate it easily through a simple bitwise comparison as well. This leaves only the sub-tree under node 2 (prefix `11`), where the process can be repeated recursively. Note that the efficiency of the algorithm stems from aggressive pruning of traversed nodes based on both the data distribution (i.e., skipping sub-trees that do not contain any data points) and the search space distribution itself (i.e., skipping sub-trees that do not intersect with the search space).

Algorithm 1 shows our range search algorithm, performed via the recursive SEARCH-LEVEL procedure. Its inputs are the search range S identified by the start (P_s) and end (P_e) points, the current level l of MDTRIE being traversed, and the current

Algorithm 1 Multi-dimensional Range Search

```
1: procedure SEARCH-LEVEL(S, l, i) ▷ Search at node index i at level l of
   MDTRIE for the search range S = (P_s, P_e)
2:     if level l is at the leaf level then
3:         Add i's primary keys to search result
4:     else
5:         (s, e) ← Morton bits for the current level in (P_s, P_e)
6:         childIdx ← s
7:         while childIdx ≤ e do
8:             if childIdx is populated ∧ childIdx ∈ S then
9:                 S' ← Search range for sub-tree at childIdx
10:                SEARCH-LEVEL(S', l + 1, childIdx)
11:            end if
12:            childIdx ← next set child bit position at node i
13:        end while
14:    end if
15: end procedure
```

node index i; both l and i are initialized to zero. If SEARCH-LEVEL reaches a leaf, we add the corresponding point to our search result. Otherwise, we compute the start and end Morton codes s and e from P_s and P_e for the current level l. We then iterate over valid and non-empty Morton codes between s and e, as outlined in the example in Figure 3. In practice, this can be sped up significantly by setting `childIdx` to the next set bit position in a single bitwise operation rather than checking every bit between s and e. If we find a Morton code matching a child that contains points and is within $[s, e]$, we update the current node index i to the corresponding child node index. We also update the search range S every time we descend down a sub-tree (i.e., at level $l + 1$) since the Morton code prefix for the path traversed so far can be eliminated from the search. We then recursively call SEARCH-LEVEL at level $l + 1$ with the updated parameters. Once the recursion completes, we move on to the next valid, non-empty child node at level l. As noted above, we aggressively leverage bitwise instructions [63] like `__builtin_ctzll`, `__builtin_popcountll` and bitmasks to accelerate range search and trie traversal.

3.4 Point Queries with Self-indexing

To the best of our knowledge, no prior multi-dimensional data structure [21, 30, 61] natively supports point queries — namely, lookups, insertions, and removals of points via a *unique primary key*. There are two main ways such queries could still be supported on such data structures: using an additional index mapping primary keys to the data points or by including the primary key as an attribute within each data point. Unfortunately, while the former is space-inefficient, the latter requires expensive range searches on the primary key attribute to locate the associated data point. The latter also reduces the spatial locality of the encoded data since primary keys are typically unique identifiers not correlated with secondary attributes.

We support point queries on MDTRIE by embedding a primary key index *within* the trie structure, ensuring space efficiency. There are two key components to our approach: storing the primary keys themselves and maintaining additional metadata to permit retrieval, insertion, and removal of a data point given its primary key.

Storing primary keys. Since a leaf node completely identifies a data point, we maintain an array of primary keys per leaf node in each treeblock; the array can store multiple primary keys for the same data point to handle duplicates. At the treeblock granularity, we store pointers to these per-leaf arrays in another array in the order the corresponding leaves appear in the treeblock. We employ two optimizations to reduce the storage footprint of this nested array. First, if a leaf has a single primary key, we inline its primary key in the top-level array instead of nesting an array with a single key. Second, for a leaf with multiple primary keys, the corresponding array is sorted and compressed using delta-encoding [64].

Retrievals. We support lookups on primary keys by first locating the leaf corresponding to the primary key and then reconstructing the data point (i.e., its Morton code) from leaf-to-root. Since multiple nodes (including leaf nodes) are encoded together in a single treeblock bit vector for space efficiency, a node's offset in the bit vector can frequently change over time as new nodes are inserted. This makes it difficult to identify a leaf node uniquely using its ever-changing offset within the corresponding bit vector. This also makes maintaining a primary key to leaf node mapping infeasible since, unlike pointer-based tries, nodes in MDTRIE cannot be directly referenced with pointers. Instead, we store a hash-based mapping from primary keys to the *treeblock* that contains the corresponding leaf node. We *compress* the treeblock pointers in our mapping, exploiting the observation that there are only a small number of treeblocks in MDTRIE— far fewer than the number of leaf nodes, permitting the use of a small number of bits to represent these pointers. Intuitively, this also results in a coarse treeblock-level mapping from primary keys to treeblocks, requiring a dynamic reconstruction of the precise location of the leaf in the treeblock. We locate the corresponding leaf node within the treeblock via traversals through the treeblock (accelerated using bitwise instructions). To reconstruct the data point, we simply traverse up the treeblock from the leaf to the root. Reconstructing them across multiple treeblocks requires storing two pieces of metadata at each treeblock t: a reverse pointer to t's parent treeblock p_t and the frontier node index of t in p_t so that the traversal can continue from the correct position in p_t.

Figure 4 illustrates how point lookups proceed using a concrete example. Given a primary key, we first find the treeblock that stores the corresponding leaf by using the primary key to treeblock pointer mapping (①). We then start at the root of that treeblock (node 0 in treeblock 2) and traverse nodes in the treeblock in depth-first order (node $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$),

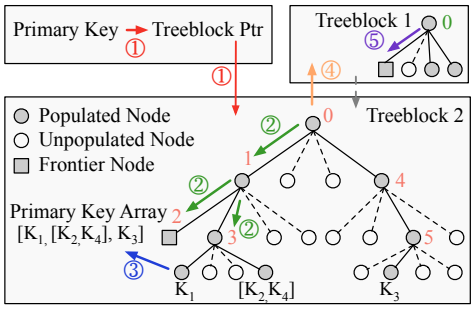


Fig. 4. Point queries in MDTRIE. See §3.4 for details.

API	Description
`t = addTable(schema)`	Initialize a table with given schema.
`s = t.search(`P_s`,`P_e`)`	Multi-attribute range search.
`r = t.get(key)`	Lookup record by primary key.
`t.insert(key,r)`	Insert new record given primary key.
`t.remove(key)`	Remove record given primary key.

Table 1. TRINITY API. See §4.1 for details.

tracking the trie path to the current node along the way until we reach a leaf node (②). At each leaf node, we search for the primary key in the primary key array. If found, we terminate our traversal and note the Morton code for the trie path until that leaf (③). The treeblock may have been a frontier node, so we may still need to find the point's remaining Morton code prefix by traversing further up the trie. That is, we follow the treeblock's parent pointer and its frontier node index within the parent (④), traversing up MDTRIE and reconstructing the Morton code of the path until we reach the root node (⑤).

Insertions and Removals. Inserting new points in MDTRIE has two components: inserting the Morton encoding for the data point and updating the metadata associated with the primary key index. The former requires traversing down MDTRIE until we hit a node where a new child must be inserted, then shifting bits in the treeblock to make room for the children nodes in depth-first order, creating frontier nodes as necessary. For the latter, we store parent pointers with frontier node indexes whenever a new frontier node is created. Also, at the leaf node, we insert the new primary key into the corresponding primary key array and add a corresponding entry to the primary key to treeblock mapping.

To remove a data point for a given primary key, we first locate its leaf node (similar to insertion) and then remove the primary key from the corresponding primary key array in the treeblock. If the removed data point is the last one associated with the leaf node, MDTRIE clears the corresponding bit in its parent node. If the parent node has no more children, it is removed, and the process is recursively repeated up the trie.

4 TRINITY: Design & Implementation

We now describe salient features of TRINITY's interface (§4.1) and system implementation (§4.2).

4.1 System Interface

TRINITY API (Table 1) is similar to NoSQL stores. A table abstracts data stored across a distributed (sharded) MDTRIE, with one unique aspect: it supports efficient multi-attribute range searches (§3). Towards supporting a flexible schema, our generalized Morton code (§3.2) can encode any attribute with a fixed-width binary representation. Supporting range queries additionally requires the attribute's binary representation to be *order-preserving*, *e.g.*, a date converted to binary POSIX timestamp preserves order across attribute values, but a binary-encoded `DDMMYY` representation does not. TRINITY can still support a wide range of data types since most (integers, timestamps, fixed-length strings, etc.) have straightforward order-preserving fixed-width binary representations. TRINITY requires additional processing to encode variable-length strings and high-precision floating point numbers, *e.g.*, by rounding values to a fixed number of fractional places for a fixed-width representation. We discuss future work on encoding variable-length data types in §6.

4.2 System Implementation

TRINITY is implemented in 4.4k lines of C++ code as a space-efficient in-memory data store for real-time multi-attribute queries. Similar to prior storage systems [34, 35, 47, 65], TRINITY's architecture is quite straightforward, with two key components: coordinator (§4.2.1) and storage servers (§4.2.2). At a high level, the storage servers store and serve queries on multiple MDTRIE shards (potentially from multiple tables). The coordinator tracks how MDTRIE shards are placed across the servers and tracks their health.

4.2.1 Coordinator performs two main tasks. First, TRINITY partitions the data for each table across multiple MDTRIE shards; the coordinator tracks the storage servers in TRINITY and the identity of the MDTRIE shards that they manage in a (`TableID`, `HashBucket`) to (`ServerID`, `ShardID`) mapping. When a client first initializes a table with a fixed schema, they contact the coordinator, which in turn allocates a fixed number of MDTRIE shards across storage servers for the table. To perform other operations on the table, clients obtain this mapping from the coordinator, cache it, and interact directly with the storage servers. If the mapping changes and the client contacts an incorrect storage server, the server asks the client to refresh its mapping from the coordinator. Second, the coordinator tracks the health of storage servers: the coordinator monitors if servers are alive using heartbeat messages. When a server fails, the coordinator initializes a replacement storage server using prior checkpoints and write-ahead logs (§4.2.2).

TRINITY scales coordinators by partitioning the table-to-shard mappings across multiple machines. In practice, we find that a single machine suffices as the amount of metadata storage and processing for even hundreds of storage servers is nominal.

Data partitioning. TRINITY partitions data across MDTRIE shards based on the hashes of their primary key. Despite its simplicity, hash-based partitioning works well for MDTRIE in practice – search (§3.3) terminates early if a shard does not contain relevant data points, and such shards can then serve other queries. More shards lead to a minimal increase in shard metadata but generally faster search queries and higher lookup throughput due to increased parallelism. It is possible for value-based partitioning (*e.g.*, by partitioning the multi-dimensional value space itself [66]) to reduce the number of shards contacted per search query, permitting higher throughput. However, spreading the query over fewer shards would reduce per-query parallelism and lead to potentially longer search latencies. In TRINITY, we opted for lower search latency over higher search throughput since all our evaluated real-world search workloads are latency-bound (§5).

4.2.2 Storage Servers store MDTRIE shards and support:

Query execution. Each shard has an associated query handler process, which accepts and executes client queries on the shard. For point queries, the client issues a request to the query handler for the shard containing the relevant primary key. For multi-attribute searches, the client issues the request to shards that belong to the queried MDTRIE and the corresponding query handlers execute the search in parallel. The query results are streamed back to the client, which aggregates them before returning them to the user. This is similar to query execution in prior search-based systems [26, 27, 36] since it enables high throughput for point queries while minimizing the latency for range queries via parallel execution.

Concurrency. TRINITY supports concurrency for its queries with atomicity guarantees by using read-write locks at treeblock granularity, rather than at node or shard granularity. Since nodes are encoded as bits, maintaining locks at node granularity would incur exorbitant storage overheads. On the other extreme, one lock per shard introduces too much contention when multiple queries contact the same shard. We empirically found treeblock granularity locks to offer a sweet spot in minimizing both storage and contention.

Persistence. Similar to other in-memory data stores [26, 39, 47, 65], TRINITY provides persistence by maintaining a bounded write-ahead log per shard, where every `insert()` and `remove()` request to a MDTRIE shard is first logged to persistent storage before being applied in-memory. TRINITY also periodically checkpoints the in-memory MDTRIE by serializing it to persistent storage, truncating the write-ahead log after a successful checkpoint for storage efficiency. To recover the in-memory MDTRIE, we reload the last checkpoint and sequentially apply all operations in the write-ahead log.

Sizing MDTRIE treeblocks. Treeblock size, measured in terms of the maximum number of nodes allowed in a MDTRIE treeblock, is an important parameter at TRINITY storage servers since it impacts both insertion performance and metadata overhead. In particular, larger treeblocks reduce the amount of per-treeblock metadata that must be maintained at the storage servers. At the same time, with larger treeblocks, inserting new data points (§3.4) requires shifting more bits in the treeblock's bitvector representation to make room for the new nodes. Our empirical analysis of real-world datasets (§5) shows that insertion and lookup latency increase slightly with larger treeblocks while the metadata overhead decreases. We use a treeblock capacity of 512 nodes by default since it offers a reasonable tradeoff between performance and storage overheads in practice.

Enabling large data dimensionality. Compared to k^2-trie, TRINITY can encode a much larger number of dimensions with a much smaller footprint. A naive generalization of k^2-trie to d dimensions requires 2^d bits for every node. As we saw in §3, MDTRIE representation significantly reduces storage footprint for d-dimensional data in two key ways. First, Generalized Morton encoding and Staggered Morton encoding (§3.2) reduce the number of active dimensions (d') to be much smaller than d in practice. Second, the collapsed node optimization permits the majority of the MdTrie nodes to use d' bits rather than $2^{d'}$ bits.

Even with these optimizations, an uncompressed MDTRIE node may still require 2^d bits in the worst case. As the number of dimensions grows large (*e.g.* > 25), this can cause MDTRIE's storage footprint and performance to increase non-linearly. While such high dimensionality for queried attributes is uncommon in practice (§5), TRINITY combats non-linear scaling at a larger number of dimensions by *slicing* dimensions across multiple MDTRIE instances, with each MDTRIE slice encoding dimensions that are often queried together. These slices handle both point queries and range queries in parallel; while point queries simply require combining the extracted dimensions, range queries require performing a set intersection across the sets of records returned by each MDTRIE slice. We show in §5.2 that the storage overhead and latency increase linearly with the number of dimensions (up to 128).

5 Evaluation

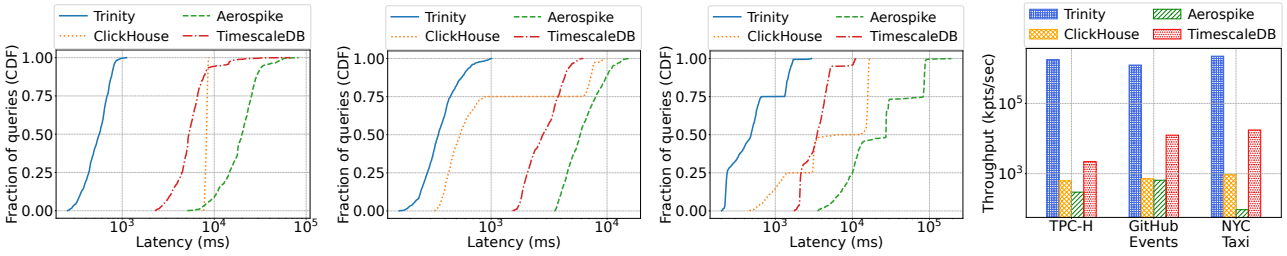
We now evaluate TRINITY to show that it meets the goals for applications that leverage multi-attribute datasets (§2).

Datasets and workloads. Table 2 summarizes datasets and multi-attribute queries used in our experiments.

- **Real-time business analytics.** We use the synthetic TPC-H dataset [67] with 1 billion records. We coalesce its `lineitem` and `orders` tables on the `orderkey` attribute so that each record has 9 attributes. Since the TPC-H benchmark lacks multi-attribute queries, we generate $1k$ synthetic range queries over multiple attributes, similar to prior work [42].
- **Statistical analysis of GitHub repositories.** We analyze the GitHub Events dataset [68] comprising user events on

Dataset	Query Templates
TPC-H (1B records, 9 attr.)	Synthetic range queries over shipdate, receiptdate, quantity, discount, orderkey and supplierkey attributes with 0.1% query selectivity.
GitHub Events (828M records, 10 attr.)	**GQ1:** Repositories with # stars $\geq g_1$ and # forks $\geq g_2$.
	GQ2: Repositories with total # events $\leq g_3$, # issues $\geq g_4$, and # stars $\geq g_5$.
	GQ3: Repositories with earliest modified date $\leq g_6$, latest modified data $\geq g_7$, and # stars $\geq g_8$
	GQ4: Repositories with earliest modified date in range (g_9, g_{10}), latest modified date in range (g_{11}, g_{12}).
NYC Taxi (675M records, 15 attr.)	**NQ1:** Trips with distance $\leq n_1$km and fare amount between \$$n_2$ and \$$n_3$.
	NQ2: Trips with pick-up and drop-off years between n_4 and n_5.
	NQ3: Trips with pick-up dates earlier than n_6 and with n_7 passenger.
	NQ4: Trips with pickup locations between latitudes (n_8, n_9) and longitudes (n_{10}, n_{11}).

Table 2. Datasets and query templates. We generate $1k$ multi-attribute queries for the TPC-H dataset based on prior work [42]. For GitHub Events and NYC Taxi datasets, we use real-world query templates to generate $1k$ queries with 0.1% average query selectivity.



(a) Business Analytics (b) GitHub Repository Analytics (c) Taxi Dataset Analysis (d) Search Throughput

Fig. 5. Performance for multi-attribute search (§5.1). Note that y-axis is in log-scale for (d).

GitHub. We group the events by repository, resulting in 828 million entries, each with 10 attributes. Our queries are adapted from real-world use cases outlined in [68].

- **Interactive taxi traffic analytics.** We use the NYC Taxi dataset [13] with logs of taxi trips in New York City. After pruning attributes with missing values, the dataset had 675 million records with 15 attributes each. We adapt multi-attribute queries from several real-world sources [69–71].

The adapted real-world templates correspond to range queries over multiple attributes with widely varying data distributions; we generate the template variable values (g_1–g_{12} for GitHub Events, n_1–n_{11} for NYC Taxi) to ensure the average query selectivity is 0.1%. We generate $1k$ queries for each use-case with an equal number of queries per template to nullify the effects of query selectivity on query performance. For point queries, we generate accesses across primary keys using a Zipf distribution representative of real-world workloads [72]. We report throughput for search and point queries as the number of points retrieved or inserted per second.

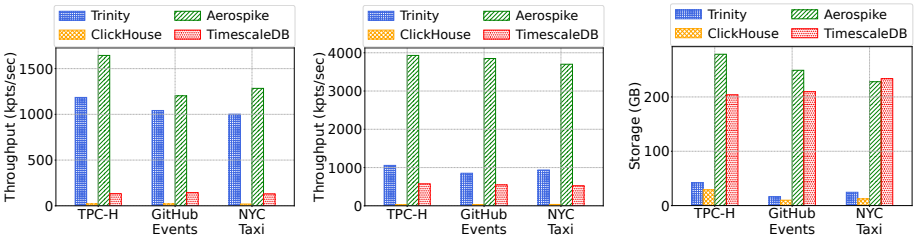
Compared Approaches. We benchmark TRINITY against three classes of systems, and MDTRIE against state-of-the-art data structures. For systems (§5.1), we compare TRINITY against Aerospike [47], a representative in-memory NoSQL data store; TimescaleDB [46], an OLTP time-series database based on PostgreSQL for fast analytics; and ClickHouse [53], a column-oriented OLAP database. We create indexes on queried attributes for systems that support them. For data structures (§5.2), we compare MDTRIE against R*-Tree [73], a variant of R-Tree for indexing spatial information; Patricia-Hypercube Tree (PH-tree) [30], a tree structure for indexing multidimensional data; and BB-Tree [43, 74], a k-ary search tree for multidimensional range queries. Since PH-Tree, R*-Tree, and BB-Tree do not *natively* support data point lookups given primary key (PH-Tree and BB-Tree instead support primary key lookups given data point), we perform a single-attribute search for the primary key attribute.

Setup. We used a Cloudlab [75] cluster of 15 machines — 5 for hosting the storage system, and 10 hosting clients. Our setup ensures that the *entirety* of the dataset and the secondary indexes can fit in memory among the storage servers for all compared systems. Each machines has 10-core 2.4GHz Intel E5-2640v4 processors, 64GB of memory and 10Gbps NIC.

5.1 Performance for Real-world Applications

We evaluate TRINITY for real-world use cases in multi-attribute search, point queries, and storage footprint. We also conducted evaluations with mixed query workloads and provided them in the Appendix for completeness.

5.1.1 Multi-attribute Search As a system that is specifically designed to support efficient multi-attribute search, TRINITY significantly outperforms compared systems (Figure 5), consistently achieving sub-second latency for queries of varying nature (Table 2). Across three datasets, TRINITY achieves on average 7.2 – 15.8× lower latency than ClickHouse, 7.3 – 10.9× lower latency than TimescaleDB, and 17.8 – 59.6× lower latency than Aerospike. Figure 5(d) shows



(a) Inserts (b) Lookups (c) Storage footprint

Fig. 6. Point query throughput (§5.1.2) and Storage footprint (§5.1.3).

that TRINITY's search throughput is 2 – 5 orders of magnitude higher than compared systems. TRINITY's query efficiency stems from its multi-dimensional MDTRIE index design that can query all attributes simultaneously. TRINITY leverages both the query search boundary and data distribution within the data shard to speed up the query using Algorithm 1.

The multi-attribute query performance for compared systems offers insights into how alternative designs handle such queries. As an OLAP columnar store, ClickHouse stores values for each attribute in a separate sorted and compressed column. It supports multi-attribute queries via vectorized scans across the compressed columns to filter out irrelevant results. However, ClickHouse must scan through many irrelevant results to eliminate them. As with all columnar storage formats, it must reconstruct records corresponding to search results via expensive scans through multiple columns, further exacerbating query latencies. On the other hand, Aerospike is a NoSQL store that maintains an index for every queried attribute. However, to perform multi-attribute queries, it only leverages a single, most selective user-specified attribute index to retain matching records for that attribute and scans through these records to eliminate irrelevant results for other attributes. Although retrieving records is more efficient in Aerospike, without the optimized vectorization offered by columnar representation, expensive scans in Aerospike result in significantly higher query latencies. Finally, TimescaleDB, a time-series database based on PostgreSQL, also maintains an index for every queried attribute and performs multi-attribute queries by finding the intersection across individual attribute-index searches. This approach still requires eliminating many irrelevant results, resulting in high search latencies.

We also make a couple of interesting application-specific observations. First, for the first three query templates (**GQ1**, **GQ2**, **GQ3**) in the GitHub analysis use-case, ClickHouse observes significantly better performance than Aerospike and TimescaleDB. This is because the queried attributes in these queries (*e.g.*, number of stars, forks, and issues) assume values heavily skewed towards a small set of unique numbers (typically zero or one), which ClickHouse can represent in its sorted and compressed columnar representation efficiently. Moreover, its vectorized scans are particularly well suited to such skewed value distributions and can significantly outperform index-based scans/joins.

TimescaleDB, on the other hand, outperforms ClickHouse on the fourth query template (**GQ4**) that accesses uniformly-distributed timestamp attributes — TimescaleDB's *hypertables* provide automatic data partitioning across timestamp fields for more performant time-series-based queries. Second, the query performance for all compared systems tends to vary significantly across templates in the taxi traffic analytics use case. This is because the NYC Taxi dataset's attributes have significant variance in value distribution and selectivity, and often, the dataset is quite sparse.

5.1.2 Point Queries We next focus on the performance for point queries; Figures 6(a) and 6(b) show insertion and lookup throughput across datasets. TRINITY outperforms ClickHouse (∼ 52.5× for inserts, ∼ 30.2× for lookups) and TimescaleDB (∼ 7.9× for inserts, ∼ 1.7× for lookups), while is slower than Aerospike (within 1.27× for inserts, 4.1× for lookups) — a NoSQL store optimized for point queries.

While inserts require ClickHouse to update multiple columns and relevant metadata, lookups require reconstructing records via multiple random accesses across these columns. ClickHouse's low point query throughput issue is well known and noted by ClickHouse maintainers [76]; our results confirm this issue. TimescaleDB observes lower insert throughput since each insert requires updating multiple attribute indexes along with the raw data; its lookup throughput, on the other hand, is comparable to TRINITY's since it can leverage its primary key index to facilitate efficient lookups. TRINITY achieves a desirably high point query throughput due to its space- and performance-efficient primary key index embedded within the MDTRIE structure (§3.4).

Aerospike's relatively higher throughput is expected: as a NoSQL store, it is optimized to support point lookups and inserts. In particular, its raw record data representations and primary key index fit completely in DRAM, and both inserts and lookups require only a couple of memory accesses to the primary key index and the raw data itself. In contrast, TRINITY's succinct trie representation requires additional non-trivial work in traversing the tree and decompressing the path to retrieve or insert the relevant record. We believe that a point query throughput lower than optimized NoSQL stores (but higher than others) while achieving a significantly better storage footprint (§5.1.3) and better performance for multi-attribute queries is a reasonable tradeoff for TRINITY.

5.1.3 Storage Footprint TRINITY observes low storage footprint (Figure 6(c)): $6.6 - 15.1\times$ lower than Aerospike and $4.8 - 12.7\times$ than TimescaleDB across the various datasets. These storage improvements are due to MDTRIE's succinct representation enabled by various optimizations outlined in §3.2. TimescaleDB and Aerospike must store both the dataset and large indexes to accelerate query execution, which can incur overhead multiple times the size of the *raw* dataset, e.g., 35GB → 249GB for Aerospike on GitHub Events dataset.

ClickHouse has a slightly lower ($\sim 1.4\times$) storage footprint than TRINITY, owing to ClickHouse's lz4-compressed columnar representation. Moreover, ClickHouse does not support secondary range indexes, relying solely on vectorized scans for query execution, further reducing its storage overheads. This, however, comes at the cost of performance for multi-attribute queries (discussed in §5.1.1) and point queries (discussed in §5.1.2). We believe that a slight increase in storage overhead for significantly improved query performance is a reasonable tradeoff for TRINITY's target applications.

5.2 Evaluating MDTRIE

We evaluate MDTRIE against state-of-the-art multi-attribute data structures, namely R*-Tree [73, 77], Patricia-Hypercube Tree (PH-Tree) [30], and BB-Tree [43, 74]. To evaluate them, we replace MDTRIE with the corresponding data structure in TRINITY. We defer a sensitivity analysis to the Appendix for brevity.

Multi-attribute search (Figure 7(a)). To ensure that our results depend only on the data structure (and not the query type), we generate $1k$ multi-attribute queries spanning all attributes with 0.1% query selectivity. Across various datasets, MDTRIE observes $1.8 - 17.7\times$, $5.0 - 12.8\times$, $1.1 - 11.6\times$ lower average latency compared to PH-Tree, R*-Tree and BB-Tree respectively.

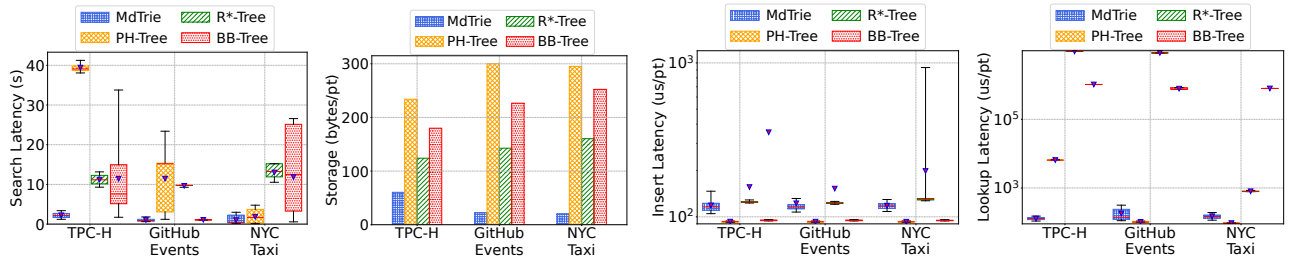
Similar to MDTRIE, PH-Tree encodes attributes with original Morton codes for each record in a tree structure. However, unlike MDTRIE, it does not employ a succinct data representation, generalized Morton codes, or collapsed nodes for sparse datasets. Additionally, its query algorithm elides optimizations we outlined in §3.3, relying on much slower pointer-based traversals rather than bitwise instructions that skip unpopulated nodes. As such, MDTRIE achieves significantly more efficient query execution compared to the PH-Tree.

R*-Trees group n-dimensional points in close proximity into *mimimum bounding rectangles*, and organizes these rectangles hierarchically in a tree. At the leaf, these rectangles encapsulate a single point, while nodes at higher layers represent aggregates of multiple points in larger bounding rectangles. The search algorithm traverses down the tree and finds rectangles that intersect with the search space. The tree structure itself heavily employs pointer-based links. R*-Tree suffers from similar issues: the search algorithm tends to visit more nodes than MDTRIE, is less cache efficient due to its larger storage footprint, and does not enjoy fast traversals via bitwise instructions, resulting in relatively slower searches.

BB-Tree constructs an almost balanced k-ary search tree with efficient scans in main memory. Its inner search tree (IST) is linearized in breadth-first order and stored in a cache-optimized array. Data objects are stored in special leaf nodes ("Bubble Bucket" or BB) that can store multiple data points. While BB-Tree picks a delimiter dimension at every level, MDTRIE uses Morton code, where the traversal of each level prunes all dimensions simultaneously. As a result, BB-Tree observes higher range query latency.

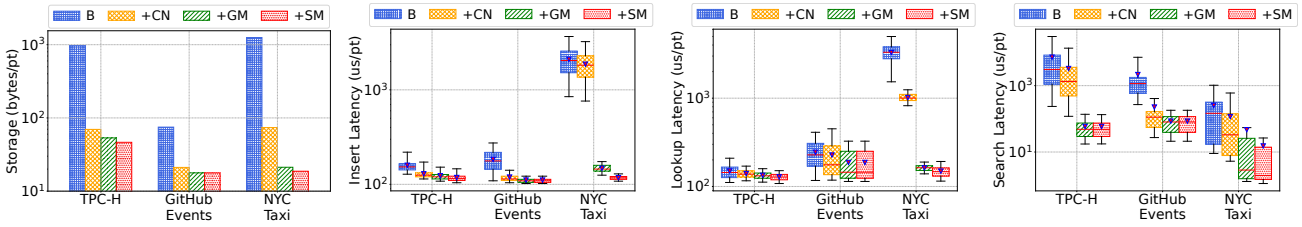
Point queries (Figure 7(c), 7(d)). MDTRIE's insertion latency is on average $1.28\times$ lower than R*-Tree's, albeit $1.19\times$ higher than PH-Tree's (Figure 7(c)). While MDTRIE's median latency is $1.22\times$ higher than BB-Tree's, BB-Tree's average latency is $3\times$ higher due to its need to rebuild the inner search tree after a series of insertions. R*-Tree employs a relatively sophisticated insertion algorithm that computes efficient bounding rectangles and often requires revising the splitting of rectangles during insertion for better spatial locality. This results in its relatively higher insertion overhead, with particularly high variance depending on the number of split revisions required; in contrast, the use of Morton codes directly affords spatial locality in both PH-Tree and MDTRIE without requiring such recomputations. PH-Tree's lower insertion latency relative to MDTRIE is expected: PH-Tree uses pointers to represent its tree structure, compared to MDTRIE's succinct representation. While this permits faster updates, it compromises storage efficiency. Similarly, BB-Tree observes faster median insertion latency as it links its inner search tree to leaf nodes (bubble buckets) with pointers. We note that MDTRIE can have slightly higher tail latencies for inserts since it occasionally requires splitting an overflowing treeblock (§3.2). Prioritizing large gains in space-efficiency over a slight reduction in insert latency is a conscious design choice in MDTRIE.

MDTRIE achieves good lookup performance (Figure 7(d)), with lower latency across datasets compared to R*-Tree and BB-Tree. While PH-Tree's performance is slightly better than MDTRIE for the GitHub Events and NYC Taxi datasets, it is $16\times$ worse for the TPC-H dataset, because tree-based encoding of the TPC-H dataset has much more balanced distribution of children nodes, i.e., all nodes have a moderate number of children (instead of mostly sparse sub-trees with some dense nodes). This forces PH-Tree to search many more children by iterating over their pointers at each node, resulting in higher lookup latency. As BB-Tree is optimized for full data point lookup (find the primary key given the data point) and does not natively support lookup given primary key, it uses slower range query on a single dimension for primary key lookup. MDTRIE, on the other hand, employs its efficient bitwise instructions and embedded primary key index to traverse up the trie and reconstruct the data point much faster.



(a) Multi-attribute search (b) Storage Footprint (c) Insert (d) Lookup

Fig. 7. Evaluating MDTRIE (§5.2). For box plots, the red line marks the median, the box marks 25^{th} and 75^{th} percentiles, the whiskers show 10^{th} and 90^{th} percentiles, and the inverted triangle marks the mean. The y-axis for (c) and (d) is in log scale.



(a) Storage (b) Insertion (c) Lookup (d) Range Query

Fig. 8. Contributions of optimizations on TRINITY performance. `B` denotes the baseline, `+CN` adds collapsed nodes, `+GM` adds generalized Morton code, `+SM` adds staggered Morton code. Note that the y-axis is in log scale.

Storage overheads (Figure 7(b)). MDTRIE observes 3.9 – 14.5×, 2.1 – 7.9×, and 3.0 – 12.7× lower storage footprint compared to PH-Tree, R*-Tree, and BB-Tree, respectively. This is due to MDTRIE's optimized succinct representation, in contrast to pointer-based representations in other data structures.

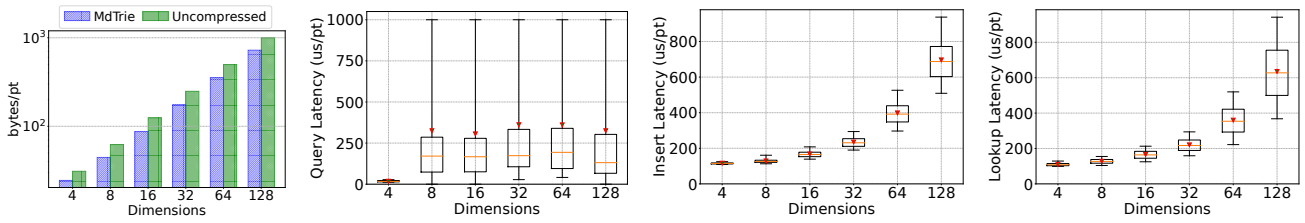
Contributions of optimizations (Figure 8(a) – 8(d)). Each of TRINITY's optimizations (§3.2) contribute to reducing its storage footprint. The collapsed node optimization yields 2.7–13.9× reduction, with further reductions due to Generalized Morton encoding (1.18 – 21.4×). As noted in §3.2, Staggered Morton encoding's improvements are dataset-dependent. In the GitHub Events dataset, all but one attribute has the same bit width, leaving no room for improvement with staggering. The NYC Taxi and TPC-H datasets have many attributes with varying bit-widths, yielding an additional 25.5% and 12.9% reduction, respectively. MDTRIE's storage optimizations also improve query performance, since storing fewer bits per trie node speeds up traversal and improves cache efficiency. The collapsed node optimization speeds up range queries by 2.2 – 9.5×, while generalized Morton encoding further adds 2.5 – 56× improvement across different datasets. Staggered Morton encoding reduces query latency even further by 65% on the NYC Taxi dataset. We observe similar improvements for point queries: collapsed nodes speed up point queries by 1.33–2.2×, while generalized Morton codes further speed it up by 6.6× on the NYC Taxi dataset. Staggered Morton encoding further improves point query latency by up to 21%.

Effect of number of attributes. We modify the number of dimensions in the TPC-H dataset by dropping or replicating some of its existing dimensions. We support large data dimensionality (> 16) using the dimension slicing approach outlined in §5.1.3, i.e., the set of dimensions is sliced and encoded across multiple MDTRIE instances. We configure each MDTRIE slice to cover 8 dimensions. Figure 9 shows that TRINITY's storage footprint and point latency increase linearly as the number of attributes is increased from 4 to 128. The large variances in query latency are due to the randomness in the query generation process. Specifically, queries over 4 dimensions are less selective (fewer dimensions to filter the dataset) and return, on average, significantly more points, leading to a much smaller amortized query latency per point.

Effect of Treeblock sizes. We study the impact of varying the treeblock size in MDTRIE from 128 to 1024 on query latency and storage overhead (Figure 10). We find that, as expected (§5.1.3), insertion and lookup latency gradually increases as the treeblock size increases, while the storage size decreases as we increase the treeblock size.

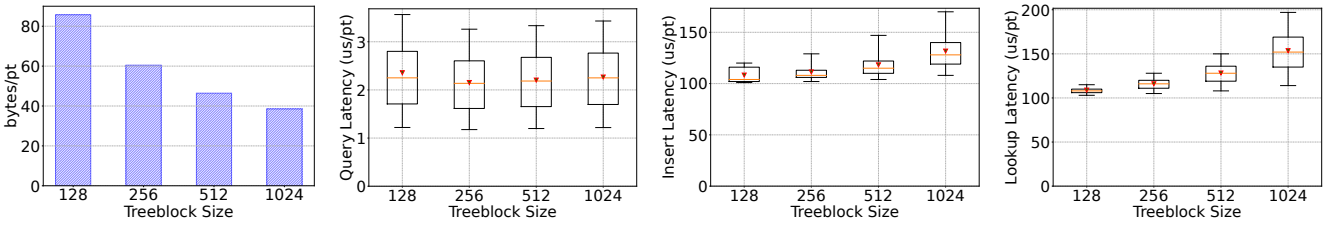
6 Discussion and Future Work

Variable-length data types. Since most applications for machine-generated data are satisfied with searches on floating points up to a few digits (*e.g.*, geo-data, timestamps) and fixed-length strings (*e.g.*, message type, log severity), fixed-length attributes do not limit our use cases. High-precision floating



(a) Storage vs #Dimensions (b) Search latency vs #Dimensions (c) Insertion latency vs. #Dimensions (d) Lookup latency vs. #Dimensions

Fig. 9. Effect of number of dimensions on TRINITY performance (5.2). The x-axis is in log-scale. The y-axis for (a) is in log-scale.



(a) Storage vs. Treeblock size (b) Query latency vs. Treeblock size (c) Insert latency vs. Treeblock size (d) Lookup latency vs. Treeblock size

Fig. 10. Effect of Treeblock size on TRINITY performance (5.2). The x-axis is in log-scale.

point numbers (*e.g.*, FP32) can be supported by encoding exponent and fractional components as two separate attributes. Encoding variable-length strings remains an open problem: while order-preserving string compression (*e.g.*, HOPE [78]) provides a fixed-width order-preserving encoding for variable-length strings, they require knowing the entire dataset a priori. TRINITY can also store the fixed-length prefix of variable-length strings in MDTRIE (for searchability) while storing the complete strings on persistent storage.

Space-filling curves. Prior surveys [79] find Morton and Hilbert codes to be the most suitable encoding schemes that preserve spatial locality. While Hilbert codes have impressive theoretical clustering properties [80], MDTRIE employs Morton codes mainly because knowing the search boundary provides no straightforward benefit in pruning the search space for Hilbert codes [55]. Realizing efficient algorithms for Hilbert code variant of MDTRIE is an exciting area of future work. Space-filling curves have also been used in other aspects of data storage, notably, HyperDex [81] maps multi-attribute objects to multi-dimensional hyperspace and assigns objects to servers that own partitions of the hyperspace. While HyperDex supports distributed multi-attribute search, it is primarily an approach for partitioning data across servers and is complementary to TRINITY.

Depth-first versus level ordering. Depth-first [59] and level ordering [43, 58] are two ways of laying out trie nodes onto a compact bit vector representation. Level-ordering stores nodes by level and supports edge traversals in constant time. While depth-first ordering does not support constant-time edge traversals, it does enable greater cache- and update-efficiency [61], which makes it the basis of our choice in MDTRIE. While prior works have explored level-ordered trie in a single dimension [29], we leave the adaptation of level-ordered tries to multi-dimensional data to future work.

7 Conclusion

We have presented TRINITY, a compressed in-memory data store that simultaneously facilitates query-efficiency across large volumes of multi-attribute records. TRINITY achieves this using a novel dynamic, succinct, multi-dimensional data structure MDTRIE. Our evaluation of TRINITY across real-world use cases shows that TRINITY supports significantly faster multi-attribute searches, while enabling comparable or lower storage footprint and comparable or higher point query throughput relative to state-of-the-art systems.

Acknowledgement

We thank our shepherd, André Brinkmann, and anonymous EuroSys reviewers for their valuable comments and insightful feedback. This work is supported in part by NSF Awards `2047220`, `2147946`, and a NetApp Faculty Fellowship, as well as gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

References

[1] Jianqing Fan, Fang Han, and Han Liu. Challenges of big data analysis. *National science review*, 1(2):293–314, 2014.

[2] Michael P Andersen and David E. Culler. Btrdb: Optimizing storage system design for timeseries processing. In *FAST*, pages 39–52, 2016.

[3] Emma M. Stewart, Anna Liao, and Ciaran Roberts. Open μpmu: A real world reference distribution micro-phasor measurement unit data set for research and application development. *IEEE*, 2016.

[4] Henggang Cui, Kimberly Keeton, Indrajit Roy, Krishnamurthy Viswanathan, and Gregory R. Ganger. Using data transformations

for low-latency time series analysis. In *ACM SoCC*, pages 395–407, 2015.

[5] Galen Reeves, Jie Liu, Suman Nath, and Feng Zhao. Managing massive time series streams with multi-scale compressed trickles. *VLDB*, 2(1):97–108, 2009.

[6] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. Scuba: Diving into data at facebook. *VLDB*, 6(11):1057–1067, 2013.

[7] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *VLDB*, 8(12):1816–1827, 2015.

[8] Google Stackdriver. `https://cloud.google.com/stackdriver/`.

[9] Amazon CloudWatch. `https://aws.amazon.com/cloudwatch/`.

[10] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *NSDI*, pages 421–436, 2019.

[11] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *SIGCOMM*, 2016.

[12] P. Tammana, R. Agarwal, and M. Lee. Simplifying Datacenter Network Debugging with PathDump. In *OSDI*, 2016.

[13] NYC Taxi Download. `https://tinyurl.com/bdk9k5uk`.

[14] Uber's Big Data Platform: 100+ Petabytes with Minute Latency. `https://www.uber.com/blog/uber-big-data-platform/`.

[15] Uber Freight Carrier Metrics with Near-Real-Time Analytics. `https://tinyurl.com/bdj68hd9`.

[16] Introducing AresDB: Uber's GPU-Powered Open Source, Real-time Analytics Engine. `https://www.uber.com/blog/aresdb/`.

[17] Haitao Yuan and Guoliang Li. A survey of traffic prediction: from spatio-temporal data to intelligent transportation. *Data Science and Engineering*, 6:63–85, 2021.

[18] Andreas Papadopoulos and Dimitrios Katsaros. A-tree: Distributed indexing of multidimensional data for cloud computing environments. In *IEEE*, pages 407–414, 2011.

[19] Yu Hua, Dan Feng, and Ting Xie. Multi-dimensional range query for data management using bloom filters. In *IEEE*, pages 428–433, 2007.

[20] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *SIGMOD*, pages 2071–2086, 2020.

[21] Rudolf Bayer and Volker Markl. The ub-tree: Performance of multidimensional range queries. Technical report, 1998.

[22] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[23] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. SIGMOD '84, page 47–57, New York, NY, USA, 1984.

[24] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An efficient multi-dimensional index for cloud data management. In *Proceedings of the first international workshop on Cloud data management*, pages 17–24, 2009.

[25] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1567–1581, 2016.

[26] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *NSDI*, pages 337–350, 2015.

[27] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, pages 485–500, 2016.

[28] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *SIGMOD*, pages 1149–1164, 2017.

[29] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. In *SIGMOD*, pages 323–336, 2018.

[30] Tilmann Zäschke, Christoph Zimmerli, and Moira C Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. In *SIGMOD*, pages 397–408, 2014.

[31] Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, CMU, 1988.

[32] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. PODS '84, page 181–190, New York, NY, USA, 1984.

[33] Steven M Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *PACMCGIT*, pages 110–116, 1980.

[34] MongoDB. `http://www.mongodb.org`.

[35] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS*, 44(2):35–40, 2010.

[36] Elasticsearch. `http://www.elasticsearch.org`.

[37] Swaminathan Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *SIGMOD*, 2012.

[38] Apache HBase. `https://hbase.apache.org/`.

[39] SingleStore: The Database for the Data-Intensive Era. `https://www.singlestore.com/`.

[40] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, 2006.

[41] SAP HANA. `http://www.saphana.com/`.

[42] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. SIGMOD '20, page 985–1000, New York, NY, USA, 2020.

[43] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. Bb-tree: A main-memory index structure for multidimensional range queries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1566–1569. IEEE, 2019.

[44] Songrui Wu, Qi Li, Guoliang Li, Dong Yuan, Xingliang Yuan, and Cong Wang. Servedb: Secure, verifiable, and efficient range queries on outsourced database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 626–637. IEEE, 2019.

[45] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168, 2014.

[46] TimescaleDB: SQL made scalable for time-series data. `https://tinyurl.com/e9r9an3y`.

[47] V Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms. *VLDB*, 9(13):1389–1400, 2016.

[48] Jin-Yi Cai, Venkatesan T. Chakaravarthy, Raghav Kaushik, and Jeffrey F. Naughton. On the complexity of join predicates. PODS '01, page 207–214, New York, NY, USA, 2001.

[49] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. SIGMOD '15, page 63–78, New York, NY, USA, 2015.

[50] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[51] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Transactions on Algorithms (TALG)*, 4(1):1–30, 2008.

[52] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 193–208, 2020.

[53] ClickHouse. `https://clickhouse.com/`.

[54] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *VLDB*, 2020.

[55] Jonathan K. Lawder and Peter J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *ACM Sigmod Record*, 30(1):19–24, 2001.

[56] Peter Kirschenhofer, Helmut Prodinger, and Wojciech Szpankowski. Multidimensional digital searching and some new parameters in tries. *International Journal of Foundations of Computer Science*, 4(01):69–84, 1993.

[57] Bradford G Nickerson and Qingxiu Shi. On k-d range search with patricia tries. *SIAM Journal on Computing*, 37(5):1373–1386, 2008.

[58] Naila Rahman, Rajeev Raman, et al. Engineering the louds succinct tree representation. In *International Workshop on Experimental and Efficient Algorithms*, pages 134–145. Springer, 2006.

[59] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[60] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *2010 ALENEX*, pages 84–97. SIAM, 2010.

[61] Diego Arroyuelo, Guillermo de Bernardo, Travis Gagie, and Gonzalo Navarro. Faster dynamic compressed d-ary relations. In *International Symposium on String Processing and Information Retrieval*, pages 419–433. Springer, 2019.

[62] David A White and Ramesh Jain. Similarity indexing with the ss-tree. In *IEEE*, pages 516–523, 1996.

[63] Intrinsics for Bitwise Logical Operations. `https://tinyurl.com/vjxcnh52`.

[64] Delta Encoding. `http://en.wikipedia.org/wiki/Delta_encoding`.

[65] Redis. `http://www.redis.io`.

[66] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[67] TPC-H Download. `http://www.tpc.org/tpch/`.

[68] Github Events Download. `https://tinyurl.com/yme6zp7r`.

[69] A ride through NYC: SQL queries visualization. `https://tinyurl.com/2s3j3ce9`.

[70] New York City Taxi and For-Hire Vehicle Data. `https://tinyurl.com/bdk9k5uk`.

[71] Introduction to IoT: New York City Taxicabs. `https://tinyurl.com/4fnbsx63`.

[72] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*, 2010.

[73] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[74] BB-Tree: C++ implementation. `https://github.com/flippingbits/bb-tree`.

[75] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *USENIX ATC 19*, pages 1–14, Renton, WA, July 2019.

[76] ClickHouse Low Throughput Github Issue. `https://tinyurl.com/2p9fyj3b`.

[77] R*-Tree: C++ implementation. `https://tinyurl.com/5f4n4njn`.

[78] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-preserving key compression for in-memory search trees. SIGMOD '20, page 1601–1615, New York, NY, USA, 2020.

[79] David J Abel and David M Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems*, 4(1):21–31, 1990.

[80] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE*, 13(1):124–141, 2001.

[81] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 25–36, 2012.