# Sortledton: a Universal Graph Data Structure

Per Fuchs
TU Munich
per.fuchs@cs.tum.edu

Domagoj Margan
Imperial College London
d.margan15@imperial.ac.uk

Jana Giceva
TU Munich // MDSI
jana.giceva@in.tum.de

## ABSTRACT

Despite the wide adoption of graph processing across many different application domains, there is no underlying data structure that can serve a variety of graph workloads (analytics, traversals, and pattern matching) on dynamic graphs with single edge updates updates.

In this paper, we present Sortledton, a universal graph data structure that addresses the open problem by carefully optimizing for the most relevant data access patterns used by graph computation kernels. It can support millions of updates per second, while providing competitive performance (1.22x on average) for the most common graph workloads to the best-known baseline for static graphs – CSR. With this, we improve the ingestion throughput over state-of-the-art dynamic graph data structures, while supporting a wider range of graph computations, with a much simpler design and significantly smaller memory footprint (2.1x that of CSR).

## 1 INTRODUCTION

Graph processing on dynamic datasets is an increasingly important problem for many application domains, from recommender systems to fraud and threat detections [23, 13, 26, 25]. Today many scenarios need to perform a wide range of graph computations – analytics, graph pattern matching (GPM), and traversals – on diverse datasets, which can be highly dynamic and entail millions of edge insertions per second [25]. For example, Alibaba uses a combination of graph analytics and interactive graph traversals for fraud detection [8], while the Twitter recommendation service is based on GPM and traversals [13, 26]. Both need to perform the above-mentioned analysis while ingesting many updates per second [26, 8].

Building a system that can efficiently process such a diverse set of graph algorithms over a dynamic graph dataset is still an open problem. This is largely because such a system needs an underlying data structure that can absorb a high rate of updates, while efficiently processing the wide range of heterogeneous graph workloads. Designing such a universal data structure is a non-trivial challenge, that we discuss and address in this paper.

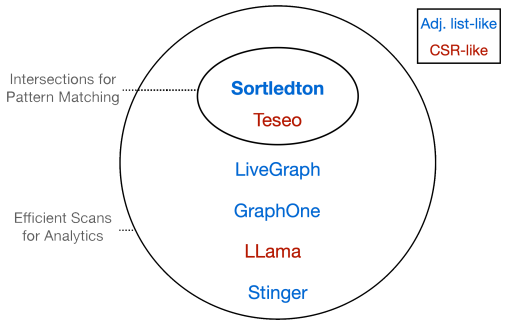



Figure 1: Supporting scans, updates and intersections for a universal graph data structure is challenging.

Figure 1 depicts the related work landscape in the context of supporting such requirements. Most of the existing related work has not succeeded at supporting all of them. Let's focus on the data structure that stores the neighborhood of vertices. First, to achieve a competitive performance on graph analytical workloads (*e.g.*, page rank) or traversals (*e.g.*, single-source shortest path) the data structure needs to support fast scans [28]. Second, to efficiently support GPM (*e.g.*, triangle counting), the data needs to be sorted to enable fast intersections [20]. Third, to support the ingestion of high update rates, the data structure needs to be dynamically adjustable. An avid reader will notice that satisfying any 2 out of the 3 requirements is relatively simple but the combination of all three is challenging.

As a result, most prior work has focused on static graphs [14, 27] or foregoes support for GPM when operating on dynamic graphs [28, 15, 10, 19]. Hence, many of the graph algorithms that have been developed for static graphs can only be run on bulk loaded data. To the best of our knowledge, only Teseo [17] provides support for all three requirements, depicted in Figure 1. However, Teseo imposes a significantly more complex design than us (*CSR-like*), without demonstrating advantages in performance.

We first present a systematic analysis of memory access patterns to support optimal performance, *e.g.*, *sequential vertex access* or *sequential neighborhood access*. Utilizing a framework composed of different basic data structures, we

isolate and quantify the effects of access patterns.

Based on our insights, we propose Sortledton – an *adjacency list-like* universal data structure that can ingest millions of updates per second while supporting the high performance execution of *heterogeneous* graph workloads. Sortledton uses a sorted set data structure to store neighborhoods, which can be scanned as fast as a contiguous region of memory, while supporting fast intersections for GPM computations and up to 5 million edge updates per second. We show that Sortledton can outperform all competitors for update throughput. At the same time, we are either faster or on par even with the highly specialized systems for all graph workloads on dynamic datasets and on average only 1.2x slower than running the computations on a static graph stored in CSR format.

# 2 BACKGROUND AND MOTIVATION

To better motivate the problem, we start by detailing what we mean by heterogeneous graph workloads. Three graph workloads categories exist in the literature: graph analytics, graph pattern matching (GPM), and graph traversals [4]. Notable example algorithms for each category are PageRank, triangle count, and single-source-shortest path. The survey by Sahu et al. finds that all three workload categories are frequent use-cases in graph databases computations [25]. For example, Alibaba uses a combination of graph analytics (finding big bicliques) and interactive graph traversals for fraud detection [8] and Twitter recommends tweets based on GPM as well as traversals [13, 26]. Both companies describe these workloads as highly dynamic [26, 8]. Despite the increasing necessity for efficient support of diverse dynamic graph workloads, most graph processing systems target only a single workload category or static graph computations [14, 27, 9, 28].

## 2.1 Understanding the Problem

The key challenge, when designing a universal graph data structure for all aforementioned workload categories on dynamic graphs, is to support a wide range of operations: (1) all workloads require fast scanning of neighborhoods, (2) high throughput of new edges requires fast insertions, and (3) GPM needs intersection [20]. It is relatively easy to achieve a combination of any two of these operations. Scans and inserts can be supported by a vector – with an amortized *push_back* operation. However, intersections are slow because they run in $\mathcal{O}(N \times M)$, with $N$ and $M$ being the cardinalities of the participating vectors. Scans and intersections can be supported by a sorted array but this is a static data structure and individual insertions are very slow. Fast intersections and inserts can be supported with a hash set. However, hash sets have empty slots which require the evaluation of a predicate for each scanned element. Hence, supporting all three operations requires a trade-off and we address this with a systematic study in Section 3 and a data structure design in Section 4.

## 2.2 Memory Access Patterns

Graph workloads are known to be memory access bound [3]. Hence, optimizing for their memory access patterns is most important. We identify four common memory access patterns:

1. sequential access to the neighborhoods of all vertices
2. sequential access to the edges within a neighborhood
3. random access to algorithm-specific properties, *e.g.*, scores for PageRank or distances for BFS
4. random access to the neighborhoods of all vertices

The PageRank (PR) algorithm in Listing 1 exhibits all access patterns except the second. It accesses all neighborhoods and edges sequentially in the order of the vertices (line 3) and reads the *contributions* array at random locations (line 4). The *random vertex access pattern* typically arises within traversal algorithms.

```
Data: contrib: V-sized array, contributions per
      vertex this round, scores: V-sized array, scores
      next round
1 for v ∈ V do
2 |   incoming ← 0
3 |   for e ∈ v.neighbors do
4 |   |   incoming ← incoming + contrib[e]
5 |   scores[v] ← incoming
```

**Algorithm 1: An example for *sequential vertex access*: the main loop of a PageRank algorithm.**

## 2.3 Graphalytics Benchmark

To quantify the effects of optimizing for different memory access patterns, we use the kernels specified by the LDBC Graphalytics Benchmark [5]. These cover all three graph workload categories and all four access patterns. The benchmark includes 5 kernels: weakly connected component (WCC), PageRank (PR), community detection via label propagation (CDLP), breadth-first search (BFS), weighted single-source shortest path (SSSP), and local clustering coefficient (LCC). The first three are examples of analytical algorithms. The next two are graph traversals and the last one is dominated by triangle counting – a typical GPM algorithm. All algorithms exhibit the *sequential neighborhood access* and *algorithm-specific property access patterns*. The analytical algorithms show *sequential vertex access*, while SSSP and LCC access the neighborhoods in random order. The direction-optimized BFS exhibits both vertex access patterns, but is dominated by *sequential vertex access*.

For WCC, PR, BFS, and SSSP, we used the reference implementation of the *Graph Algorithm Platform Benchmark Suite* (GAP BS) [2]. LCC and CDLP are implemented as in Teseo [17].

The Graphalytics Benchmark also provides the graph dataset (Table 1). The Graph500-x datasets are synthetic power-law

Table 1: Graph datasets with number of edges and vertices, average degree and size in memory when stored as an undirected graph with 8 Bytes per vertex and 16 Bytes per weighted edge.

| Graph | #V | #E | $\overline{D}$ | Size [$GB$] |
|---|---|---|---|---|
| dota-league | 61K | 51K | 836 | 1.6 |
| graph500-22, uniform-22 | 2M | 64M | 26 | 2.0 |
| graph500-24, uniform-24 | 9M | 260M | 29 | 8.3 |
| graph500-26, uniform-26 | 33M | 1B | 33 | 33.9 |
| com-friendster | 29M | 2B | 72 | 67.0 |

Table 2: Operations for a universal graph data structure.

| Operation | Complexity | Required for |
|---|---|---|
| **Basic** | | |
| *get_neighbors* | $\mathcal{O}(1)$ | all workloads |
| *scan_neighbors* | $\mathcal{O}(D)$ | all workloads |
| *insert_edge* | $\mathcal{O}(\log D)$ | all dynamic workloads |
| *insert_vertex* | $\mathcal{O}(\log V)$ | all dynamic workloads |
| *delete_edge* | $\mathcal{O}(\log D)$ | dynamic workloads |
| *delete_vertex* | $\mathcal{O}(D)$ | dynamic workloads |
| **Set functionality** | | |
| *find_edge* | $\mathcal{O}(\log D)$ | updates, consistency checks |
| *intersect_neighbors* | $\mathcal{O}(D)$ | GPM |

graphs. The scale factor x describes the number of edges and vertices in the graph; each increment doubles the number of vertices and edges. Uniform-x datasets are like Graph500-x but they have a uniform degree distribution. *dota-league* and *com-friendster* are real-world graphs.

# 3 REQUIREMENTS AND DESIGN GOALS

Now that we understand the key challenge for building a universal graph data structure for dynamic graphs, we discuss our systematic approach to designing one. To address the heterogeneity challenge of Section 2.1, we begin by outlining the requirements for a graph data structure in general, *i.e.*, the necessary operations. They are listed in Table 2. We categorize them into basic and set functionality. The first category is supported by most former graph data structures [10, 15, 28, 19]. However, the second category is not captured by them as they store neighborhoods in list data structures. Hence, their *intersect_neighbors* operation has the complexity of $\mathcal{O}(N \times M)$ with $N$ and $M$ being the size of the participating neighborhoods. Similarly, their *find_edge* operation completes in $\mathcal{O}(degree)$. Efficient support of these operations is critical for GPM and dynamic workloads that update or delete edges. Hence, a universal graph data structure should store neighborhoods in set data structures. In the next subsections, we show how the memory access patterns from Section 2.2 influence the design of graph data structures by microbenchmarks.

## 3.1 Sequential Vertex Access

The first memory access pattern is (1) *sequential vertex access*, *e.g.*, the outer loop of PR (see Listing 1). Ideally, one should store the neighborhoods of all vertices contiguously in memory. The CSR data structure, depicted in Figure 4, is specifically optimized for this access pattern. It is a static structure and it stores all neighbors in a large array in the order of the vertices they belong to.

We analyze the effects of such memory layout optimizations with a simple microbenchmark. We compare the runtimes of the algorithms from the Graphalytics benchmark (c.f. Section 2.3) executed on CSR and a simple sorted vector-based adjacency list, as presented in Figure 4. Such an adjacency list implementation stores the neighborhoods in random memory places with no relationship to each other, thereby representing the other end of the spectrum. Even when using the adjacency list, one can add software prefetching instructions to optimize for the predictable vertex access pattern. We do so only for the BFS algorithm by adding a single prefetch instruction to fetch the neighborhood three vertices ahead.

The normalized runtimes are shown in Figure 2a, where a ratio above 1 means the CSR is faster. We observe that even though PR and WCC heavily rely on sequential vertex access, an adjacency list mostly stays within 40% of the runtime of a CSR. Only WCC on *com-friendster* has a slowdown of over 1.4. LCC and SSSP show speedups of 20% for *graph500-24*. They have a dominant *random vertex access* pattern and we suspect that CSR results in a higher false sharing of the cache lines.

We conclude that while optimizing for the *sequential vertex access* is beneficial, it is not strictly necessary for good analytical performance because optimizing for *sequential vertex access* only addresses a small fraction of all memory accesses, *i.e.*, the first memory access to a new neighborhood. These are in the order of $|V|$ while patterns (3) and (4) can occur in the order of $|E|$ accesses, where $E$ can be at least 10x larger than $V$ [18].

## 3.2 Sequential Neighbourhoods Access

Next, we analyze the effects of optimizing for the *sequential neighborhood access pattern*. Ideally, one would use a contiguous memory region for each neighborhood, as done by Livegraph. However, no dynamic set data structure with such properties. Therefore, we check how well the access pattern can be supported by sorted sets that maintain blocks of elements (*e.g.*, B+ trees or unrolled skip lists [22]).

To do that, we compare the runtimes of the Graphalytics algorithms on: (1) vector-based adjacency list where neighborhoods are completely continuous to (2) the adjacency sets in sorted blocks that are linked together via pointers. We vary the block size. Intuitively, a larger block size would lead to lower run times.

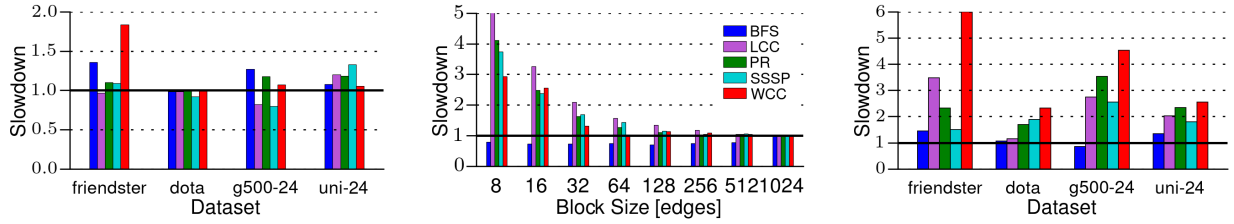
The normalized runtimes are shown in Figure 2b. We note that all algorithms show nearly equal performance on both data structures when (2) uses a block size larger than or equal to 256 edges. We conclude that optimal *sequential neighborhood access* can be supported by set data structures with at least 256 edges per block.

## 3.3 Random Access to Algorithm Properties

The third memory access pattern is reading the algorithm-specific properties. Thus, it is not influenced by the memory layout of the graph data structure itself. However, we can influence the data structure that holds algorithmic-specific properties by choosing the domain of the vertex identifiers. We hypothesize that it is the most important access pattern because it happens in the innermost loop of the computation and is random, *e.g.*, PR line 4 of Listing 1.

To explore the effects, we test two options for the vertex identifier domain: dense and sparse. Most data structures store the *dense* domain ($[0, \dots |V|]$) [19, 15, 10, 28], and many graph algorithms are implemented assuming this domain, *i.e.* they use arrays to store algorithmic-specific properties. However, storing a dense domain complicates the deletion of vertices and we cannot assume that the vertex identifiers provided by the user to a graph database are dense [5, 11]. Therefore, we explore storing the *sparse* domain.

Since, our framework for microbenchmarks does not support *sparse* vertex identifier domains, we evaluate the effects




(a) *sequential vertex access*: vector adjacency list vs CSR.

(b) *sequential adjacency access*: varying edge block sizes vs vectors.

(c) *algorithmic-specific properties*: Teseo sparse vs dense domain.

Figure 2: Effects of optimizing for different access patterns (cf. Section 2.2.)

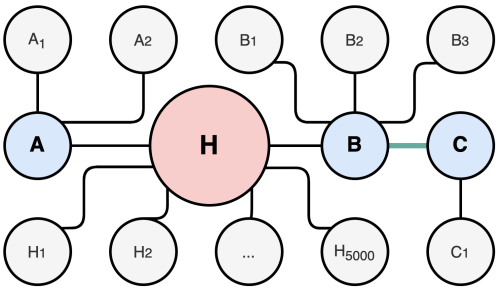



Figure 3: Example graph with hub vertex, $H$, and vertices ($A$, $B$, $C$) with their neighbors.

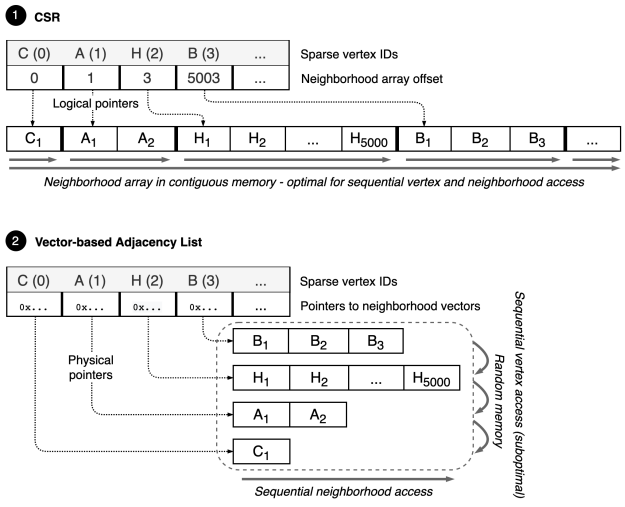



Figure 4: Classical graph data structure designs ① CSR, ② Vector-based adjacency list for example graph from Figure 3.

of a *sparse* domain vs *dense* domain using Teseo [17]. We show normalized runtimes in Figure 2c. Running on a *dense domain* is very beneficial for most algorithms, leading up to 6x performance improvements. In the case of Teseo, the main overhead is from using a hash map to translate edges from a sparse to a dense domain.

An alternative would be to rewrite all graph algorithms to use concurrent hash maps to store algorithmic-specific properties, *e.g.*, the *contrib* and *scores* array in Listing 1, so, they can run on the sparse domain directly. This, however, would incur similar overheads. Furthermore, it complicates the parallelization of the algorithm due to using concurrent hash maps instead of arrays.

In conclusion, any graph data structure for analytics needs to store a dense vertex identifier domain. In the dynamic setting with user-provided vertex IDs this requires translating a sparse domain into a dense domain when inserting new vertices.

## 3.4 Random Vertex Access

Finally, the *random vertex access pattern* happens when neighborhoods of vertices are accessed in an unpredictable order. This is typical for graph traversals, *e.g.*, SSSP. Ideally, one aims to minimize the latency for retrieving the neighborhood of a vertex. Given the need for dense vertex identifiers, we can use the IDs as offsets in a vector to store the mapping, as done in many existing data structures [10, 28, 15]. This minimizes lookup latency compared to other mapping data structures like trees and hash sets because vectors need exactly one memory access per lookup. We measure that using a `std::sorted_map` or a robin hood hashmap instead of a vector for the mapping, resulting in slowdowns between 1.1x and 3x, depending on the algorithm and graph.

In conclusion, we establish that a universal dynamic data structure for heterogeneous workloads needs to have:

1. a set data structure for neighborhoods to run intersections
2. a neighborhood data structure keeping large blocks of edges for *sequential neighborhood access* (3.2)
3. ability to expose a sparse and a dense vertex domain to the user (3.3)
4. a low-latency index for *random vertex access* (3.4)

Furthermore, we find that optimizing for *sequential vertex access* can be beneficial, but is not as critical as for other access patterns.

# 4 DATA STRUCTURE DESIGN

Our design implements the operations from Table 2 and is optimized for *random vertex access*, *sequential neighborhood access*, and *algorithm-specific property access*. Supporting only these three access patterns allows a simple design because neighborhoods can be independent data structures. Furthermore, our data structure has no hidden costs like amortized operations or background threads, runs analytics without the need for any precomputation, and ingests updates into read-optimized segments directly.

## 4.1 High-Level Design

There are two high-level designs for graph data structures. We name them *adjacency list-like* and *CSR-like* because their main characteristics stem from these classical data structures (Figure 4). The *adjacency list-like* design has one *adjacency index* and an *adjacency list* for each neighborhood. The neighborhoods are sets `Set<ID>` of destinations, and the index is a map `Map<ID, Set<ID>>`. The *CSR-like* design stores all neighborhoods in one data structure and maintains an index of offsets for it. The formalization of the index is `Map<ID, offset>`. A strawman of the neighborhood data structure is a set of edges: `Set<pair<ID, ID>>`. However, this is neither performant for computation nor space-efficient because it replicates the source of the edges many times. Ideally, we need a set that stores sources and destinations clustered by source.

We compare both designs in terms of the memory access patterns from Section 2.2. Both optimize for *random vertex access* with their indices and neither influences the *algorithm-specific access*. Furthermore, the *CSR-like* design optimizes for both *sequential vertex access* and *sequential neighborhood access*, while the *adjacency list-like* design only optimizes for *sequential neighborhood access*.

Given our insight that optimizing for *sequential vertex access* is less beneficial than for any other pattern (Figure 2), we choose the *adjacency list-like* design. This has three advantages.

First, an *adjacency list-like* data structure is embarrassingly parallel at the granularity of vertices because its neighborhoods are independent. Parallelization at this granularity is successfully utilised in the vertex-centric computation model and many algorithms [2]. Second, the maintenance of the index is simple and cheap because it is independent of changes to the neighborhoods. This is opposed to the expensive maintenance in the *CSR-like* design where one edge insertion leads to multiple index updates. This requires solutions that impede *random vertex access*, *e.g.* lazy or amortized index updates [19, 17]. Finally, the *adjacency list-like* design allows reusing well-studied map and set data structures from prior research. The *CSR-like* design requires a novel data structure that incorporates the factorization of edges into existing set designs [17]. The key insight is to decompose the problem of building a graph data structure into choosing a map and set type, and parallelizing them. Next, we pick a suitable map and set candidate.

## 4.2 Data Structure

**The adjacency index** maps vertex IDs to vertex records. A record contains multiple fields: a pointer to the neighborhood, its size and a read-write latch for parallelization.

As described in Section 3.4, to minimize the latency of a map lookup, we use a vector. To concurrently resize the vector without locking it for updates, we use two levels (Figure 5). The first level is small and has a fixed size. It holds pointers to the second-level segments that contain the vector's elements. When resizing the vector, we allocate exponentially growing second-level segments and add a corresponding pointer in the first level concurrently [6].

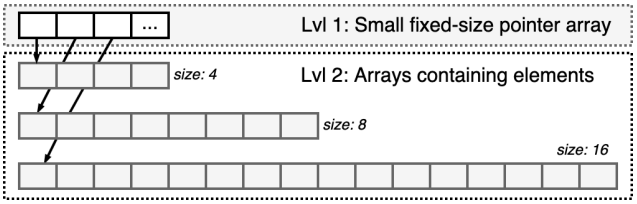



Figure 5: Two-level vector.

**The adjacency sets** store the neighborhood of each vertex. For a universal graph data structure, they should support intersections and *sequential neighborhood access* (Section 3) - sorted sets that store blocks of edges are well-suited. Typical implementations of such sets are B+ trees and unrolled skip lists [22]. We choose the second because it does not need global rebalancing. In contrast to the original unrolled skip list, we keep edges within blocks sorted. We show this structure in Figure 6 ❸. The elements of the unrolled skip list are blocks of edges combined with the header containing: the number of edges, the highest destination within the block, and pointers for each level of the skip list.

The data structure supports standard set operations by combining ordinary skip list algorithms to find the correct block and then a binary search within the block to find the correct position for reading or writing. Blocks split into two when they fill up, and merge into one when they are less than half full. Therefore, the fill ratio of our block is between 50% and 100%. Both insertions and deletions move at most *block size* elements. Hence, the operations complete in $\mathcal{O}(\max(\log D, blocksize))$.

With such properties, an unrolled skip list is a good choice for hub vertices. However, for vertices with neighborhoods smaller than the *block size*, we use headerless, power of two-sized vectors (Figure 6 ❷). This is space-efficient and follows the power-law distribution [28]. Insertions and deletions respect the sorted order and complete in $\mathcal{O}(blocksize)$.

The *block size* influences the performance of graph computations (cf. Section 3.2) and edge insertion throughput which we analyze in Figure 7 when loading *Graph500-24* edge-by-edge (meps stands for million edges per second). A *block size* of 128 leads to the highest throughput. With smaller blocks, insertions suffer from random memory access to find the correct block. For larger blocks, insertions need to shift a larger number of edges within the blocks. We expect similar results for other power-law graphs. For uniform graphs, block sizes above the average degree show no influence on performance because all neighborhoods are kept in single blocks.

**Vertex identifier translation** Analytical workloads require a dense vertex identifier domain but in dynamic settings users usually provide identifiers from sparse domains (Section 3.3). Therefore, we provide a simple binary translation between the domains, and an interface to access both. Performance critical computations should use the dense domain and translate the inputs and outputs. Figure 6 ❶ shows these translations as logical-to-physical and physical-

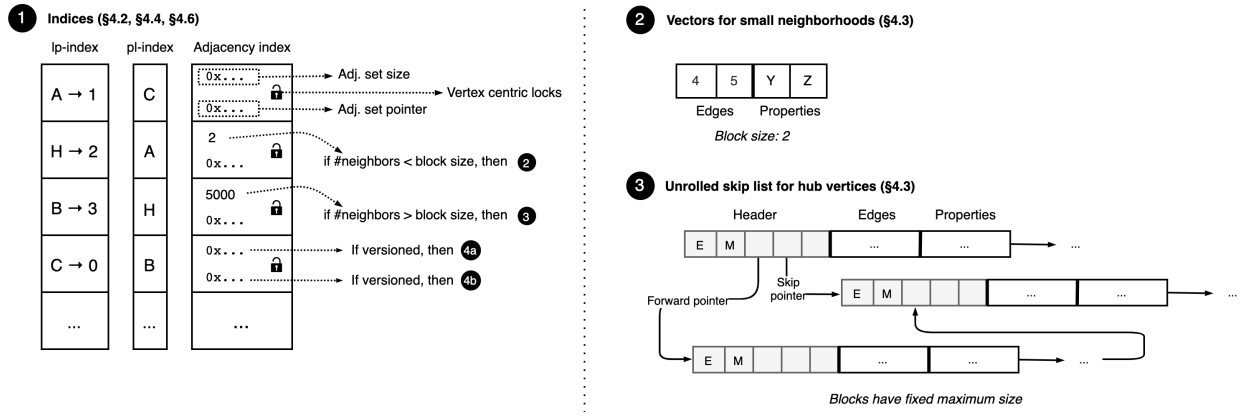



Figure 6: Sortledton's data structure for vertices $H$, $A$ and $B$ of the graph in Figure 3: ① Translation and *Adjacency Index*, ② Vectors for small neighborhoods, ③ unrolled skip list for hub vertices.

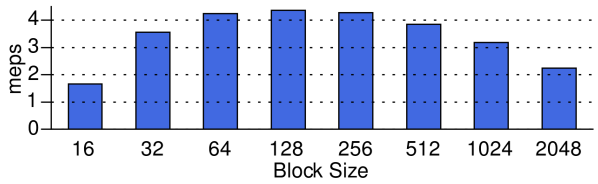



Figure 7: Sortledton's insertion throughput with varying block sizes.

to-logical indices (*lp-index*/*pl-index*). To store the translation, we use a concurrent hash set [24] from sparse to dense, and a concurrent two-level vector from dense to sparse domains.

**Edge Properties** Edge properties are common in graph analytics, GPM, and traversals. Their storage should be governed by their access patterns as detailed by Gupta et al. [12]. They are usually accessed during scans and should follow the same order as the edges. However, as many workloads do not access them, columnar storage is preferred. Hence, we store them at the end of each edge block, in the same order as the edges (Figure 6).

## 4.3 Parallelization

We have one read-write latch per vertex to allow multi-threaded access (see Figure 6 ❶). Before executing any operation on the vertex or its edges, the latch needs to be locked. This locking mechanism is simple. It scales because the number of latches in the system grows with the number of vertices. It is dead-lock-free because most operations require only a single latch, and we guarantee a global locking order for intersections and multiple open scans.

Our locking model can lead to scalability bottlenecks when processing hub vertices. To overcome this, one can parallelize the unrolled skip list with one latch per block. That way, we can implement all operations such that at most one block per skip list level needs to be locked at any time [22]. However, we choose the simple locking model because it has better scalability than all competitors (Section 5.2), and leave the concurrent skip list implementation for future work.

# 5 EVALUATION

We run our experiments on a dual-socket machine with Intel Xeon E5-2680v4 processors, which has 70 MiB of L3 cache, 14 hardware threads, and 256 GB of memory. We compiled all systems with GCC v10.2 and the O3 parameter. Further, we disabled Linux's NUMA aware page migration feature. Numbers reported are the median of 5 runs. For Graphalytics kernels, runtimes include translation costs for inputs and outputs for all systems but Teseo on sparse identifiers. We use a state-of-the-art kernel implementation (see Section 2.3) and we disable disk-logging for all systems. For Sortledton, we set the *block size* to 512 trading insertion for analytical performance (cf . Figure 2b and Figure 7). We add software prefetching to BFS.

## 5.1 Qualitative Comparison to Related Work

We compare our work to a diverse range of state-of-the-art dynamic graph data structures that support single edge updates: Stinger [10], GraphOne [15], LLama [19], Livegraph [28], and Teseo [17]. We relate the data structures used by all systems with the memory access patterns (Section 2.2) and the high-level designs (Section 4.1).

**Stinger, GraphOne, and Livegraph** are *adjacency list-like*. Hence, they have good support for *random vertex access* and are not optimized for *sequential vertex access*. Livegraph uses one vector per neighborhood for optimal *sequential neighborhood access*. Stinger and GraphOne use one (14 edges) and two fixed block sizes (8 or 512 edges) for their neighborhoods, respectively. All of them use neighborhood data structures that append the inserts which does not allow for efficient graph pattern matching.

**LLama and Teseo** have a *CSR-like* design and optimize for *sequential vertex access*. LLama's read-store holds multiple snapshots. A snapshot is a sorted array of new edges since the last snapshot. Writes are buffered in a key-value store. We create a new snapshot every 10 seconds as suggested by the authors. The snapshotting fragments neighborhoods. Hence, LLama does not optimize for *sequential neighborhood access*. However, this is hidden due to temporal locality when the computation follows the *sequential vertex access pattern*. The combination of *random vertex access* and *sequential neighborhood access* is most challenging

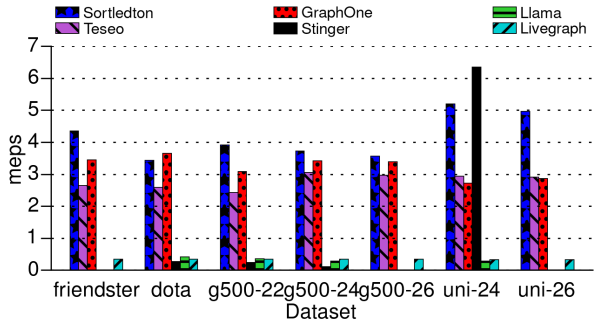



Figure 8: Edge insertion throughput with random edge ordering. GraphOne upholds lower consistency guarantees.

for LLama.

Teseo stores its vertices and edges in a B+ tree with 2MB-sized leaves containing packed memory arrays [16]. Packed memory arrays store blocks of elements interleaved with gaps to allow insertions. Blocks contain multiple neighborhoods and up to 512 edges. Thus, Teseo has good support for *sequential vertex access* and *sequential neighborhood access*. For *random vertex access*, Teseo uses a hash map that is lazily updated. Teseo is designed to store sparse vertex identifiers and needs to compute a dense mapping to interface with existing graph algorithms. They need to translate each edge read during analytics into the dense domain. Since this is expensive, the authors provide a specialized version that can load only graphs with dense identifiers. For graph computations, we measure both versions.

## 5.2 Insertions Performance

The experiment evaluates the throughput of single edge insertions in all systems. We add all edges of the input graph in random order, one-by-one as undirected edges with no sleep time on the user side. The addition of an edge checks if both vertices exist and inserts them if not. The next step asserts that the edge does not exist before inserting it. Livegraph, Teseo, and Sortledton perform all operations for each edge insertion encapsulated in a transaction, thereby ensuring atomicity and isolation. The other systems give no guarantees and GraphOne cannot check if an edge already exists.

Figure 8 shows the throughput in million edges per second (meps). Missing bars indicate that a system could not load the graph due to memory restrictions. For power-law graphs (first 6), Teseo and Sortledton are superior to the others because their neighborhood sets allow for efficient checks if an edge exists. GraphOne has similar performance, but as noted earlier does not perform the check if an edge exists. If we introduce this check, its throughput would drop to 5 edges per second [17]. Sorltedton's processes up to 1.6 million edges per second more than Teseo without using background threads while using less memory (see Section 5.4).

For uniform graphs (first 2 from the right), Stinger demonstrates the best performance – although, it cannot load *uni-26* on our system due to its high memory consumption. This reveals that for uniform graphs, it is cheaper to linearly search for the existence of an edge than paying the price of keeping them sorted.

## 5.3 Analytics

We analyze the influence of the different data structures on the runtime of analytics, traversals, and GPM queries. We run the Graphalytics benchmark kernels as defined in Section 2.3 after loading the graph edge-by-edge. We normalize the runtimes against using a CSR, which is arguably the best general-purpose baseline for static graphs. When beneficial, we use a NUMA optimized CSR as baseline.

Figure 9 shows the slowdown of each system. We select *dota-league*, *Graph500-24*, and *com-friendster* as a representative set of graphs. Missing bars either indicate that the data structure could not load the graph due to the memory constraints, or did not complete the kernel computation within an hour.

LCC, a GPM algorithm, shows no slowdown for Sortledton, a 3x for Teseo and 11x to 106x or no completion for other systems. This is due to Teseo's and Sortledton's set-based neighborhoods.

WCC and PR exhibit the *sequential vertex access* and *sequential neighborhood patterns*. Sortledton, Teseo on dense vertices, and LLama have runtimes close to the CSR, confirming that support for either pattern is effective. Teseo on sparse vertices takes up to 14x longer because it translates each edge into the dense domain using a hashmap lookup (Section 2.2). Stinger's small, fixed-size neighborhood blocks lead to pointer chasing (cf. Figure 2b). Livegraph needs to scan 3x as much data and evaluates a predicate for each scanned edge. GraphOne implements access to its neighborhood by copying them into a user-provided vector.

SSSP combines *random vertex access* with *sequential neighborhoods accesses*. The runtime for all systems, apart from LLama, is similar to WCC and PR. LLama does not optimize for *sequential neighborhood access* in combination with *random vertex access*.

BFS shows the highest variance among all systems. We use the direction-optimized variant by Scott Beamer [1]. It exhibits the *sequential vertex* and the *sequential neighborhood access patterns*. It differs from all other kernels, as it stops scanning early after finding an edge that satisfies a predicate; often scanning less than 8 edges. Hence, Livegraph and GraphOne have decreased performance due to their overheads for accessing a few edges. On *graph500-24*, other systems show similar performance as for PR and WCC. However, slowdowns on *com-friendster* or *dota-league* show that BFS is hard to optimize for. CDLP is dominated by building histograms of IDs. Hence, it is not indicative for the graph data structure performance.

## 5.4 Memory Consumption

Figure 10 shows the final memory consumption of the systems after loading *g500-24* edge-by-edge combined with deletions. The finally stored graph contains all and only edges from *g500-24*. LLama, Livegraph and Graphone garbage collection subsystems have issues with deletions and their memory consumption grows. Although, the experiment allows for a stable memory consumption because there are never more edges than in the final graph concurrently stored. Teseo, Stinger and Sortledton show stable memory consumption once the graph reaches its final size. They use 48 GB (Teseo), 27 GB (Stinger), and 18 GB (Sortledton), respectively. For reference, storing *g500-24* statically in a CSR requires 8.3 GB. So, Sortledton's overhead is 2.1x because our blocks are 75% full on average, and we store the vertex

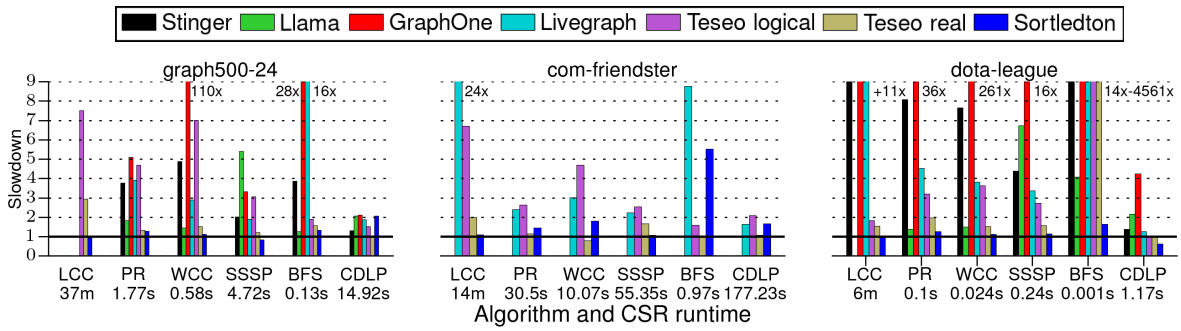



Figure 9: Graph kernel runtimes normalized to CSR.

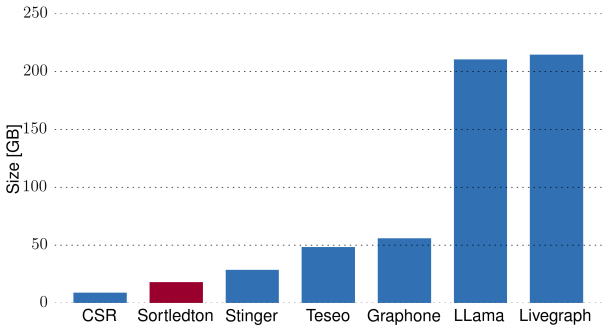



Figure 10: Memory consumption storing *g500-24*.

ID translations in $2 \times |V|$. The first overhead is inherent in many dynamic data structures (e.g., B-trees, vectors, or hash sets). The second is needed by systems that offer a sparse domain and a dense domain. To conclude, it is possible to store a dynamic graph in twice the memory needed for a static graph.

## 6 RELATED WORK

We discuss graph data structures with support for single edge inserts [19, 10, 15, 28, 17]. Figure 1 relates them to the challenge outlined in Section 2.1. Only, Teseo and Sortledton solve the challenge of supporting graph pattern matching by computing intersections in linear time. However, they follow fundamentally different designs. Sortledton has an *adjacency list-like* design, while Teseo follows the *csr-like* design. We compared these designs in Section 4.1. LLama, GraphOne and Livegraph optionally support disk-based storage. We discuss three other differentiators between Sortledton and related work. First, GraphOne, LLama, and Teseo use read and write-optimized segments to handle inserts. This leads to lower read performance or reduced freshness. Second, all competitors rely on background threads to perform data structure maintenance, *e.g.*, Teseo uses one thread per core for rebalancing and garbage collection. In particular, in combination with compute-intensive GPM this leads to overprovisioning. Finally, most systems run pre-computations before analytics after applying updates. LLama and GraphOne ingest all buffered writes into read-optimized storage. Similarly, GraphOne and Teseo create a snapshot in $O(V)$ steps before starting analytics. GraphOne stores the sizes of the neighborhoods to guarantee isolation from new updates. Teseo creates a translation from sparse to dense vertices.

**Batched update graph data structures** trade-off update latency for higher throughput. Aspen and Terrace support fast scans and intersections [7, 21]. Aspen is *adjacency list-like* and can run coarse-grained transactions per update batch by a single-writer copy-on-write scheme. It uses purely functional trees storing blocks of edges and a functional tree for the adjacency index. Terrace mixes a *adjacency list-like* and *CSR-like* design. It uses three different data structures depending on the size of the neighborhoods: they inline small neighborhoods in the index, use packed memory arrays for medium-sized neighborhoods, and B-Trees for hub vertices.

## 7 CONCLUSIONS

*Sortledton* is a sorted, simple graph data structure that executes up to 5 million edge updates per second, supports analytical, GPM, and traversal workloads with runtimes within 1.2x on average of CSR while needing only ∼2x the space of CSR. We achieve this by reusing existing data structures.

We construct Sortledton based on two key principles. First, a universal graph data structure needs to store neighborhoods in sets to support GPM, consistency, edge updates, and deletions. Second, we identify four memory access patterns in graph workloads: *sequential vertex*, *sequential neighborhood*, *algorithmic-specific property*, and *random vertex access patterns*. With a series of microbenchmarks, we show that it is more important to optimize for *sequential neighborhood access* and *algorithmic-specific property access* because they occur once per edge, rather than the other two access patterns that occur once per vertex. Therefore, *csr-like* designs lose their main advantage over *adjacency list-based* designs that are significantly simpler to build.

## 8 References

[1] S. Beamer, K. Asanovic, and D. A. Patterson. Direction-optimizing breadth-first search. *Sci. Program.*, 21(3-4):137–148, 2013.

[2] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[3] S. Beamer, K. Asanovic, and D. A. Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IEEE IISWC*, pages 56–65, 2015.

[4] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *CoRR*, abs/1910.09017, 2019.

[5] M. Capota, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. A. Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In *ACM GRADES*, pages 7:1–7:6, 2015.

[6] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *Springer OPODIS*, volume 4305, pages 142–156, 2006.

[7] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 918–934. ACM, 2019.

[8] B. Ding, K. Zeng, and W. Yu. Alibaba sponsor talk at VLDB, 2020.

[9] V. V. dos Santos Dias, C. H. C. Teixeira, D. O. Guedes, W. M. Jr., and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *ACM SIGMOD*, pages 1357–1374, 2019.

[10] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader. STINGER: high performance data structure for streaming graphs. In *IEEE HPEC*, pages 1–5, 2012.

[11] O. Erling, A. Averbuch, J. L. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz. The LDBC social network benchmark: Interactive workload. In *ACM SIGMOD*, pages 619–630, 2015.

[12] P. Gupta, A. Mhedhbi, and S. Salihoglu. Integrating column-oriented storage and query processing techniques into graph database management systems. *CoRR*, abs/2103.02284, 2021.

[13] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. J. Lin. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proc. VLDB Endow.*, 7(13):1379–1380, 2014.

[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *Commun. ACM*, 52(9):89–97, 2009.

[15] P. Kumar and H. H. Huang. Graphone: A data store for real-time analytics on evolving graphs. In *USENIX FAST*, pages 249–263, 2019.

[16] D. D. Leo and P. A. Boncz. Fast concurrent reads and updates with pmas. In *ACM GRADES-NDA*, pages 8:1–8:8, 2019.

[17] D. D. Leo and P. A. Boncz. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endow.*, 14(6):1053–1066, 2021.

[18] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[19] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: efficient graph analytics using large multiversioned arrays. In *IEEE ICDE*, pages 363–374, 2015.

[20] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[21] P. Pandey, B. Wheatman, H. Xu, and A. Buluç. Terrace: A hierarchical graph container for skewed dynamic graphs. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1372–1385. ACM, 2021.

[22] K. Platz, N. Mittal, and S. Venkatesan. Concurrent unrolled skiplist. In *IEEE ICDCS*, pages 1579–1589, 2019.

[23] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, 2018.

[24] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[25] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, 2017.

[26] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. J. Lin. Graphjet: Real-time content recommendations at twitter. *Proc. VLDB Endow.*, 9(13):1281–1292, 2016.

[27] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN PPoPP*, pages 135–146, 2013.

[28] X. Zhu, M. Serafini, X. Ma, A. Aboulnaga, W. Chen, and G. Feng. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *Proc. VLDB Endow.*, 13(7):1020–1034, 2020.