

Sortledton: a Universal, Transactional 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 transactional updates.

In this paper, we present Sortledton, a universal graph data structure that addresses the open problem by being carefully optimizing for the most relevant data access patterns used by graph computation kernels. It can support millions of transactional 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 – . With this, we improve the ingestion throughput over state-of-the-art dynamic graph data structures, while supporting a wider range of graph computations under transactional guarantees, with a much simpler design and significantly smaller memory footprint (2.1x that of).

PVLDB Reference Format:

Per Fuchs, Domagoj Margan, and Jana Giceva. Sortledton: a Universal, Transactional Graph Data Structure. PVLDB, 15(6): 1173 - 1186, 2022.
doi:10.14778/3514061.3514065

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/PerFuchs/gfe_driver.

1 INTRODUCTION

Graph processing on dynamic datasets is an increasingly important problem for many application domains, from recommender systems to fraud and threat detections [25, 46, 50, 51]. Today many scenarios need to perform a wide range of graph computations – analytics, graph pattern matching (), and traversals – on diverse datasets, which can be highly dynamic and entail millions of edge insertions per second [50]. For example, Alibaba uses a combination of graph analytics and interactive graph traversals for fraud detection [17], while the Twitter recommendation service is based on and traversals [25, 51]. Both need to perform the above-mentioned analysis while ingesting many updates per second [17, 51].

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 transactional updates, while efficiently processing the wide range of heterogeneous graph

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

workloads. Designing such a universal data structure is a non-trivial challenge, that we discuss and address in this paper.

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 [57]. Second, to efficiently support (e.g., triangle counting), the data needs to be sorted to enable fast intersections [1, 22, 41]. Third, to support the ingestion of high update rates, the data structure needs to be dynamically adjustable. Finally, to ensure correct results for the concurrently executing analytical queries in the presence of updates [39], the data structure needs to support versioning (e.g., MVCC [43]). 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 [18, 22, 29, 52] or foregoes support for when operating on dynamic graphs [19, 30, 36, 57]. Furthermore, most dynamic graph data structures do not provide transactional guarantees. Hence, many of the graph algorithms that have been developed for static graphs cannot be run concurrently with updates [19, 30, 36]. Livegraph was the first system to propose a multi-versioned, transactional data structure, by storing two timestamps per edge [57]. However, this triples the memory consumption and hurts the scan performance. To the best of our knowledge, only Teseo [34] provides concurrent support for all three requirements, depicted in Figure 1. However, Teseo imposes a significantly more complex design than us (*CSRLike*), 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.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514065

Based on our insights, we propose Sortledton – an *adjacency list-like* universal data structure that can ingest millions of transactional 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 $\text{O}(\sqrt{\#})$ computations and up to 5 million transactional edge updates. 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 [49] format.

2 BACKGROUND AND MOTIVATION

To better motivate the problem, we start by detailing what we mean by heterogeneous graph workloads. Three graph workload categories exist in the literature: graph analytics, graph pattern matching (GPM), and graph traversals [8]. 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 [50]. For example, Alibaba uses a combination of graph analytics (finding big bicliques) and interactive graph traversals for fraud detection [17] and Twitter recommends tweets based on as well as traversals [25, 51]. Both companies describe these workloads as highly dynamic [17, 51]. 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 [18, 22, 29, 52, 57].

2.1 Understanding the Problem

There are two key challenges when designing a universal graph data structure for all aforementioned workload categories on dynamic graphs. The first one 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) $\text{O}(\sqrt{\#})$ intersections [12, 22, 41]. 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 $\text{O}(\sqrt{\#} \cdot \#)$ with $\#$ and $\#$ 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.

The second challenge is to run updates concurrently alongside computations while maintaining the correct semantics [39, 54]. Most graph algorithms are developed for static graphs. Therefore, they require a static view of the graphs. In dynamic environments, this can be provided by concurrency control systems. In particular, multi-version concurrency control allows us to write to the newest version of the graph and read older static versions. We address

this by first identifying the specific transactional requirements in Section 3.5 before describing a graph-optimized concurrency control system in Section 5.

2.2 Memory Access Patterns

Graph workloads are known to be memory access bound [7, 20]. 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
- (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 [11]. 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 (LP), 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 $\text{O}(\sqrt{\#})$ algorithm. All algorithms exhibit the *sequential neighborhood access* and *algorithm-specific property access patterns*. The analytical algorithms show *sequential vertex access*, while BFS and SSSP access the neighborhoods in random order. The direction-optimized WCC exhibits both vertex access patterns, but is dominated by *sequential vertex access*.

For WCC, PR, LP, and BFS, we used the reference implementation of the *Graph Algorithm Platform Benchmark Suite* (GAPBS) [6]. SSSP and LCC are implemented as in Teso [34].

The Graphalytics Benchmark also provides the graph dataset (Table 1). The Graph500-x datasets are synthetic power-law 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*, *com-friendster*, *yahoo-songs*, and *edit-wiki* are real-world graphs. The latter two are from the

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	\bar{d}	Size $\gg \frac{1}{2}$
dota-league	61K	51K	836	1.6
graph500-22, uniform-22	2M	64M	26	2.0
yahoo-songs	1.6M	256M	315	7.6
edit-wiki	51M	255M	22	7.6
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		
<i>get_neighbors</i>	$O(\frac{1}{2})$	all workloads
<i>scan_neighbors</i>	$O(\frac{1}{2} \frac{1}{2})$	all workloads
<i>insert_edge</i>	$O(\log \frac{1}{2})$	all dynamic workloads
<i>insert_vertex</i>	$O(\log + \frac{1}{2})$	all dynamic workloads
<i>delete_edge</i>	$O(\log \frac{1}{2})$	some dynamic workloads
<i>delete_vertex</i>	$O(\frac{1}{2} \frac{1}{2})$	some dynamic workloads
Set functionality		
<i>nd_edge</i>	$O(\log \frac{1}{2})$	updates, consistency checks
<i>intersect_neighbors</i>	$O(\frac{1}{2} \frac{1}{2})$	

KONECT project [31] and are bipartite networks with edge creation timestamps.

3 REQUIREMENTS AND DESIGN GOALS

Now that we understand the key challenges 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 [19, 30, 36, 57]. 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 $O(\frac{1}{2} \frac{1}{2})$ with $\#$ and $\frac{1}{2}$ being the size of the participating neighborhoods. Similarly, their *nd_edge* operation completes in $O(\frac{1}{2} \frac{1}{2})$ (defined as the average degree of the graph). Efficient support of these operations is critical for static 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 running multiple microbenchmarks.

Figure 2: Example graph with hub vertex, v , and vertices (u, w, x, y, z) with their neighbors.

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

3.1 Sequential Vertex Access

The first memory access pattern is (1) *sequential vertex access*, *e.g.*, the outer loop of `scan_neighbors` (see Listing 1). Ideally, one should store the neighborhoods of all vertices contiguously in memory. The data structure, depicted in Figure 3, 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 3. 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.

¹The *delete_vertex* operations cannot be supported in $O(\log V)$ due to the fact that all edges which reference this vertex need to be deleted.

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

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

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

Figure 4: Effects of optimizing for different access patterns (cf. Section 2.2.)

We do so only for the algorithm by adding a single prefetch instruction to fetch the neighborhood three vertices ahead.

The normalized runtimes are shown in Figure 4a, where a ratio above 1 means the is faster. We observe that even though and heavily rely on sequential vertex access, an adjacency list mostly stays within 40% of the runtime of a . Only on *com-friendster* has a slowdown of over 1.4. and show speedups of 20% for *graph500-24*. They have a dominant *random vertex access* pattern and we suspect that 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 $j+j$ while patterns (3) and (4) can occur in the order of $j-j$ accesses, where can be at least 10x larger than $+$ [31, 35].

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 [57]. However, no dynamic set data structure with such properties supports intersections. Therefore, we check how well the access pattern can be supported by sorted sets that maintain blocks of elements (e.g., B-trees [13] or unrolled skip lists [45]).

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 4b. 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., 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 ($\{0, \dots, j\}$) [19, 30, 36, 57], 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 [11, 21]. Therefore, we explore storing the *sparse* domain.

Since, our framework for microbenchmarks does not support *sparse* vertex identifier domains, we evaluate the effects of a *sparse* domain vs *dense* domain using Teseo [34]. We show normalized runtimes in Figure 4c. 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., . 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 [19, 30, 57]. 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 hashmap² instead

²<https://github.com/martinus/robin-hood-hashing>

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.

3.5 Transaction on Graphs

The second challenge we address is the support for running updates and analytics concurrently and in isolation using `atomic` to reuse static analytical and `atomic` solutions in dynamic settings. For this, we identify the transactional requirements of graph workloads.

Concurrency Control: Graph workloads are read-heavy and common read-write transactions are very simple [2, 4, 6, 11, 25, 46, 51]. While queries can take up to tens of seconds for analytics or multiple minutes for `analytics`, the majority of write transactions add a single edge with corresponding vertices and properties, which are very fast operations. Furthermore, many of the read-write requests are actually writes with a priori known write set [4, 21].

The design of the upcoming `graph` standard extension for graph transactions (`graph` and `graph` [23, 28]) reveals further insights into the type of graph transactions that need to be supported. While `graph` is read-only, `graph` supports writes as described earlier as well as *graph construction* [3]. The latter involves complex read-write transactions that can touch large parts of the graph. We summarize that graph transactions fall in the following categories with a descending frequency of occurrence:

- (1) long-running and complex read-only queries
- (2) short and simple writes with known write set
- (3) complex read-write requests with arbitrary large, unpredictable read- and write sets

Hence, an ideal graph concurrency control system should:

- (1) decouple read-only queries from write requests
- (2) support high throughput writes with a known write set
- (3) provide support for read-write transactions with large read/write sets

In this paper, we focus on the first two points. Small read/write sets can be supported by `atomic` protocols [43] but large read/write sets combined with highly frequent updates require novel concurrency control mechanisms and are out of scope for this paper.

Version Storage: All entities are small (e.g., an edge takes only 4 or 8 bytes). Thus, versioned storage should induce no overhead for unversioned records and little overhead for versioned edges. Second, vertices and edges have only two states: present or not. Multiple versions can only occur if vertices/edges are inserted, updated, or deleted multiple times before a version can be garbage collected which is rare. Hence, an efficient system should optimize for the case of one and two versions.

Consistency: Most graph systems assume the following consistency guarantees:

no dangling edge: $\forall e \in E, \exists v \in V, w \in V, e = (v, w)$

no duplicate edge

reverse edge exists (for undirected graphs): $\forall e = (v, w) \in E, (w, v) \in E$

Enforcing these rules requires an implicit read set in write-only transactions (e.g., an edge insertion also reads the edge and its vertices to ensure non-existence of the edge and existence of the vertices). These read sets can be determined from the write sets. Concurrency control protocols used in relational databases are a natural way to enforce that these consistency guarantees hold atomically. Hence, transaction handling is also relevant for graph processing and graph streaming systems. The outlined requirements are addressed in Section 5.

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.

Finally, 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 3). The *adjacency list-like* design has one *adjacency index* and an *adjacency list* for each neighborhood. The neighborhoods are sets $\text{Set}\langle \text{ID} \rangle$ of destinations, and the index is a map $\text{Map}\langle \text{ID}, \text{Set}\langle \text{ID} \rangle \rangle$. 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 $\text{Map}\langle \text{ID}, \text{offset} \rangle$. A strawman of the neighborhood data structure is a set of edges: $\text{Set}\langle \text{pair}\langle \text{ID}, \text{ID} \rangle \rangle$. 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 induces 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 4), 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 [6, 37]. Second, the maintenance of the index is simple and cheap because it is independent of changes to the neighborhoods. This is opposed to

Figure 5: Two-level vector.

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 [34, 36]. Finally, the *adjacency list-like* design allows reusing well-studied map and set data structures from prior research [27]. The *CSR-like* design requires a novel data structure that incorporates the factorization of edges into existing set designs [34]. 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 [14].

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 [13] and unrolled skip lists [45]. We choose the second because it does not need global rebalancing [26]. In contrast to the original unrolled skip list, we keep edges within blocks sorted. We show this structure in Figure 6 3. 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 $O(\frac{\log V}{\log B} \cdot \frac{1}{\log 2} \cdot B)$.

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 2). This is space-efficient and follows the power-law distribution [57]. Insertions and deletions respect the sorted order and complete in $O(\frac{\log V}{\log 2} \cdot B)$.

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 as detailed in Section 6. Figure 6 1 shows these translations as logical-to-physical and physical-to-logical indices (*lp-index/pl-index*). To store the translation, we use a concurrent hash set [47] from sparse to dense, and a concurrent two-level vector from dense to sparse domains.

Edge Properties Edge properties are common in graph analytics, joins, and traversals. Their storage should be governed by their access patterns as detailed by Gupta et al. [24]. 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 [24]. 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 1). 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 [45]. Additionally, we propose optimistic latches [9] for all read operations, but scans. However, we choose the simple locking model because it has better scalability than all competitors (Section 7.2), and leave the concurrent skip list implementation for future work.

5 CONCURRENCY CONTROL

Now, we describe the design of our graph-optimized transaction support that addresses the second challenge of running updates concurrently with computations (Section 3). In relational systems, this is achieved by two-phase protocols [16, 43, 48]. However, in graph transactions, the writes are simpler, and the versioned items (edges and vertices) are at most 8 bytes. Therefore, we adapt prior work from relational systems [43, 48] to these new characteristics with the goal to minimize memory usage and computational overhead. For

Figure 6: Sortledton’s data structure for vertices v , w and u of the graph in Figure 2: 1 Translation and Adjacency Index, 2 Vectors for small neighborhoods, 3 unrolled skip list for hub vertices, 4 size and neighborhoods versioning.

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

durability, we propose using group commits, command *write-ahead logging*, and snapshotting as described thoroughly for relational workloads [38, 42, 43, 53], which can be implemented with low overhead [55]. However, we do not implement it because it is orthogonal to the other aspects of our design.

Version Storage: The version store aims to keep the memory overhead per edge as low as possible. An edge version records the type of the operation (insertion, update, or deletion), a commit timestamp, and the associated property. We store the version records as a linked list directly behind the edge in the block. The list is ordered from new to old versions. Versions of adjacency set sizes and vertices are stored similarly. Figure 6 4 shows an example: the user inserted $v_3 \cdot 0$ with property \cdot , updated the property to \cdot and deleted it at timestamp γ_G and γ_{\cdot} , respectively. The neighborhood size 4 (a) of 3 has two versions: 4 at timestamp γ_0 before $v_3 \cdot 0$ was deleted and 3 after the insertion at γ_{\cdot} . A linked list stores the edge versions for the deletion and the two property updates, the latest property value is also stored directly in the edge block.

We optimize for the case of two edge versions or less (Section 3.5). For a single version-edge, no version record is stored and the edge

is assigned the implicit timestamp $\text{rst version}(\gamma_0)$ for all operations. For two version-edges, we store the version record inline.

Concurrency Protocol: Our concurrency protocol handles the first and second requirements that were outlined in Section 3.5. It decouples reads from writes and optimizes for high throughput on writes with a priori known write set. It adapts with read-only optimization () and two-phase locking. The read-write transaction protocol has five steps:

- (1) claim all locks in a global order
- (2) complete all reads with the newest versions and abort if the consistency guarantees are not fulfilled (Section 3.5)
- (3) get a commit timestamp
- (4) complete all writes using the commit timestamp
- (5) release all locks

We leverage the a priori known read and write-sets to claim all locks in a global order at the beginning of the transaction [48]. Furthermore, as the protocol cannot abort after finishing the read step, no rollback logging is necessary.

For read-only transactions, we draw a commit timestamp. When reading a value, we first acquire a read lock, read the latest version before our commit-timestamp, and then release the read lock. Despite being pessimistic, the read locks are practically harmless as they are only held for the duration of one read operation. Furthermore, using locks for read-only transactions, allows more scalable implementations of read-write transactions. That is, it guarantees atomic commit without the overheads of other protocols (e.g., an atomic commit and a validation phase, or drawing two timestamps for read-write transactions [34, 57]).

We give a proof sketch for the correctness of the protocol. Let \mathcal{C}_1 and \mathcal{C}_2 be transactions with commit timestamps G_1 and G_2 . If both are read-write transactions, they are serializable by 2PL. Let

ℓ_1 be read-only and ℓ_2 be read-write. If $G_1 \prec G_2$, ℓ_1 cannot read any values written by ℓ_2 because ℓ_2 versions are written with timestamp G_2 . If $G_2 \prec G_1$, we can reason that ℓ_2 already holds all locks before ℓ_1 starts because the locking phase is completed before ℓ_2 gets its timestamp. Since ℓ_2 releases its locks only after completing, ℓ_1 needs to wait for the commit or abort of ℓ_2 , when it tries to read any value written by ℓ_2 . Hence, it reads all values of ℓ_2 after a commit or none after an abort.

Garbage Collection: We address garbage collection in two steps: (1) when can a version be collected, and (2) who collects them? We collect a version once no transaction can access it anymore. That is, we track the timestamps of all active transactions and collect all versions which are invisible to all active transactions as done in Hyper [10]. This is an important optimization to avoid long version chains in the presence of long-running transactions. Garbage is collected by the threads that execute write transactions. This leads to good data locality, requires no background threads and memory is freed where and when it is needed [16, 43].

6 IMPLEMENTATION

We implemented Sortledton by composing existing data structures. The *adjacency index* and the indices for translation use the ConcurrentVector and ConcurrentHashMap from Intel’s *Threading Building Blocks* [47]. For our *adjacency sets*, we implemented the unrolled skip list from Platz et al. [45] with slight modifications as indicated in Section 4.2 and 4.3.

Our interface is similar to prior work [19, 34] with two exceptions. First, we allow the user to access both the sparse and the dense vertex identifiers. We run graph computations by translating inputs into the dense domain, then running the analytics, and finally translating the output back to the original sparse domain. We show this process in Listing 2. The translation is fast because the input to most computations is small while the output translation is cheap and easy to parallelize. For example, a vertex receives its start ID as input and outputs one distance value per vertex. So, we need \sqrt{V} parallel translations, which take only a fraction of the runtime.

Data: tx: *ReadOnlyTransaction*, start_vertex: *logical_id*
Result: result: *array of size V containing the hop distance from start_vertex*

```
1 start_vertex_dense ← tx.dense_id(start_vertex)
2 distances ← bfs(start_vertex_dense)
3 for i ← 0 to V do
4   | result[i] ← pair(tx.sparse_id(i), distances[i])
```

Algorithm 2: Running a query on a dense domain using sparse identifiers externally.

Second, we optimize the interface that scans neighborhoods. Prior works offer three different methods to access their neighborhoods: (1) via an iterator interface [19, 57], (2) by providing a lambda function executed for each edge [34, 52], or (3) by direct memory access (e.g., for the CSR). However, using an iterator or a lambda function executes two or one function(s) per edge, respectively. To avoid this, we allow direct memory access to blocks without any versions. In Figure 8, we present a comparison of graph computations with and without this optimization. It can lead to speedups of up to 2.3x.

Figure 8: Slowdown when using an iterator interface instead of direct memory access.

7 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 GiB of memory. We compiled all systems with v10.2 and the O3 parameter. Further, we disabled Linux’s 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 4b and Figure 7). We add software prefetching to Sortledton as described in Section 3.2.

7.1 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 [19], GraphOne [30], LLama [36], Livegraph [57], and Teseo [34]. 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. Hence, they achieve good isolation of writing and reading threads for concurrency control of read-only queries. However, this 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 [36]. 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 for LLama.

Teseo stores its vertices and edges in a B+ tree with 2MB-sized leaves containing packed memory arrays [33]. Packed memory

Figure 9: Edge insertion throughput with random edge ordering. GraphOne upholds lower consistency guarantees.

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.

7.2 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 9 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 [34]. Sortledton’s processes up to 1.6 million edges per second more than Teseo without using background threads while using less memory (see Section 7.3) and versioning adjacency set sizes.

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.

Resilience to real-world update patterns: Until now, we load all edges in random order to be comparable with past work [30, 34, 36]. However, the *yahoo-songs* and *edit-wiki* graphs show a

Figure 10: Edge insertion throughput with creation order.

strong temporal locality between updates to the same vertex. In this experiment, we load these graphs in the order of their edge creation timestamps with no sleep time as opposed to the actual update frequency of a few edges per minute.

Figure 10 shows the throughput for both creation and random order of applying the updates. Sortledton, Livegraph, and LLama, have lower throughput when the graph is loaded in creation order because they use one lock per vertex leading to higher contention in bursty workloads. Although Teseo can split large neighborhoods over multiple blocks, it suffers from higher contention when loading in sorted order. GraphOne and Stinger are not affected because they batch updates or use lock-free synchronization, respectively. Stinger and LLama cannot load *edit-wiki* in either order. It is a bipartite graph with 5 times more vertices in one partition, a high edge to vertex ratio of 5:1 and a high maximum degree of 5M edges.

We conclude that vertex-centric locking is a weak point for bursty workloads and should be replaced by lock-free synchronization. Furthermore, benchmarks should specify the loading order of edges, because it can significantly influence the system performance and they should cover the complete space of graph types, *i.e.*, Graphalytics and GAPBS do not include a bipartite graph [6, 11].

7.3 Updates

We next evaluate how the data structures behave when running a balanced mix of insertions and deletions with the same setup as in Teseo [34] on an already large graph. The experiment has two phases. The first 10% of all operations load *Graph500-24*. After, we run a balanced mix 9x the operations as insertions and deletions while keeping the graph size stable. We discuss throughput over time, average throughput, and memory usage.

Average throughput (Figure 11a) allows us to compare how the different data structures react to deletions by comparing their throughput to the insertion-only experiment (Section 7.2). Livegraph and Stinger show no significant difference because they treat insertions and deletions in the same way. Sortledton’s throughput is slightly lower due to the creation and maintenance of version chains. Teseo profits from deletions as they free space and lead to fewer rebalances. LLama’s throughput halves compared to insertions-only and GraphOne’s is 64 times lower.

Throughput over time (Figure 11b) shows how the systems react to a growing workload and the effects of snapshotting in LLama and GraphOne. The legend shows the completion time per system. Teseo and Sortledton finish within 14 minutes while all other systems take more than 2 hours.

(a) Average throughput.

(b) Average throughput per second for *g500-24*.(c) Memory consumption for *g500-24*.**Figure 11: Throughput and memory consumptions on a mixture of insertions and deletions. Total execution time in the legend.**

For Sortledton, Teseo, and Livegraph, we observe smooth lines as they perform updates individually and in place. They do not depend on the current size of the dataset. Stinger loses performance while the data structure grows, but has a stable throughput afterwards. LLama’s throughput decreases over time because of neighborhood fragmentation and the memory pressure by storing multiple snapshots. Furthermore, their throughput oscillates due to the need to digest edges from write to read-store. GraphOne shows throughput between 0 and 1 million edges per second. The system becomes unresponsive when applying edges to its read store because it struggles to locate edges to delete.

Memory consumption in Figure 11c allows us to categorize the systems into two classes. Livegraph, LLama, and GraphOne show increasing memory consumption due to partial or missing garbage collection implementations. Teseo’s, Stinger’s, and Sortledton’s memory usage grow until the graph reaches its final size at 10% of all operations. Then they show stable memory consumption. They use 48 GB (Teseo), 27 GB (Stinger), and 18 GB (Sortledton), respectively. For reference, storing *graph500-24* statically in a requires 8.3 GB. So, Sortledton’s overhead is 2.1x because our blocks are 75% full on average, and we store the vertex 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.

This experiment leads us to three conclusions. First, list-based designs struggle with deletions. Second, batching updates in a write-optimized store leads to high average throughput but comes at the cost of unstable throughput and can lead to unresponsiveness. Third, it is possible to store a dynamic graph in twice the memory needed for a static graph.

7.4 Multicore Scalability

Figure 12 shows multicore scalability from 1 to 56 threads for all systems on *graph500-24*. Teseo and Sortledton execute more than 3.1 and 4.8 million checked edge insertions per second, respectively. Most other systems achieve less than 500 thousand. GraphOne achieves 3.4 million unchecked insertions per second.

Both Teseo and Sortledton scale up to 56 threads. Sortledton scales better because its concurrency control protocol is more lightweight. One could achieve even better scalability with Sortledton by using contention-free hash sets for translation between the vertex domains or adding lock-free awareness.

Figure 12: Edge insertion throughput and multicore scalability on *graph500-24*.

GraphOne does not scale beyond 14 threads because vertex ID translation is sequential, and due to contention on its write buffer [34]. However, it is twice as efficient with 14 threads as Teseo and Sortledton. This is because (1) the different insertion semantics, and (2) GraphOne batches updates in a circular buffer, then partitions them per vertex and applies the partitions in parallel to the main data structure. While this design enables high throughput, it leads to higher update latencies and requires building a snapshot before analytics. Therefore, it is not suited for Alibaba’s fraud detection workload with strict latency requirements and computations per edge insertion [46, 56].

7.5 Analytics

We analyze the influence of the different data structures on the runtime of analytics, traversals, and 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 *graph500-24*, which is arguably the best general-purpose baseline for static graphs. When beneficial, we use a *graph500-24* optimized as baseline.

Figure 13 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.

Figure 13: Graph kernel runtimes normalized to

, a 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.

and exhibit the *sequential vertex access* and *sequential neighborhood patterns*. Sortledton, Teseo on dense vertices, and LLama have runtimes close to the , con rming 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, xed-size neighborhood blocks lead to pointer chasing (cf. Figure 4b). 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.

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

shows the highest variance among all systems. We use the direction-optimized variant by Scott Beamer [5]. It exhibits the *sequential vertex* and the *sequential neighborhood access patterns*. It differs from all other kernels, as it stops scanning early after nding an edge that satis es 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 and . However, slowdowns on *com-friendster* or *dota-league* show that is hard to optimize for. is dominated by building histograms of IDs. Hence, it is not indicative for the graph data structure performance.

7.6 Concurrent Read-Write Workload

We evaluate the in uence of concurrently executing updates from Section 7.3 and or from Section 7.5. Figure 14 shows the latency of analytics and the throughput of updates when combining 1 to 32 analytical threads with 16 and 48 writers compared to running them in isolation. The experiment can only be executed by transactional systems. Teseo cannot execute the workload due to a bug.

When is run concurrently with updates, the throughput is at most 12% (Sortledton) and 22% (Livegraph) lower than updates in isolation. When updates run alongside , the throughput drops signi cantly (Figure 14c). There are two reasons for this: 1) scans complete neighborhoods while scans only parts of most neighborhoods, 2) runs longer transactions. Consequently,

holds locks on neighborhoods longer and leads to a higher number of versions in the system. In particular, a long query with 1 analytical thread strains the system with many versions [10]. The latency of both and is higher than in isolation (Figures 14a and 14d), due to multi-versioned edges which disable our direct access optimization (Figure 8) and instead follow version chains.

For Livegraph, the concurrent workloads affect each other less. There are two reasons: 1) the use of a log-structured data structure that allows writers to append the inserts without affecting the readers. 2) Livegraph always pays the overhead of having all edges versioned which lowers efficiency. Despite the interference, Sortledton is ahead for updates/analytics in all/most cases.

8 RELATED WORK

We discuss graph data structures with support for single edge inserts [19, 30, 34, 36, 57]. Figure 1 relates them to the challenges outlined in Section 2.1. Only, Teseo and Sortledton solve the first challenge of supporting graph pattern matching by computing intersections in linear time. The second challenge of allowing for concurrent updates and computations is only addressed by Teseo, Livegraph, and Sortledton. Livegraph’s concurrency control is based on an -optimized protocol [32], that causes overheads on analytical workloads and memory consumption. Teseo’s protocol is an -optimized protocol [43] for general read-write transactions. This comes at the cost of having a higher overhead on small transactions with a known write set. In particular, they have a higher abort rate, do rollback logging, need to draw two timestamps per transaction, and have a sequential validation phase. As a result, only Teseo and Sortledton address both challenges, but 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. Only Sortledton and Livegraph could execute analytics and updates concurrently. 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. In the case of LLama and GraphOne, this will result in unstable throughput over time (cf. Figure 11b). 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 [34]). In particular, in combination with

(a) Sortedledton

(b) Livegraph

(c) Sortedledton update throughput

(d) Sortedledton

(e) Livegraph

(f) Livegraph update throughput

Figure 14: Mixed workload for Graph500-24.

compute-intensive this leads to overprovisioning. Finally, most systems run pre-computations before analytics after applying updates [30, 34, 36]. LLama and GraphOne ingest all buffered writes into read-optimized storage. Similarly, GraphOne and Teseo create a snapshot in $\frac{1}{4}$ 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 [15, 44]. 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.

Graph Databases. Mature graph databases exist, e.g., Neo4J and Virtuoso. They use a linked list of edges per vertex and a columnar relational layout for storage. Neo4J’s concurrency control uses the isolation level read-committed and Virtuoso uses single version locking. Hence, both systems could profit by changing to Sortedledton as underlying storage because of its cache-friendly layout and low neighborhood lookup latencies as well as the higher isolation level and/or better decoupling of readers and writers, respectively.

Further graph workloads. So far, we have discussed analytics, traversals, and graph pattern matching workloads because they drive our design. However, Besta et al. list three further workloads: local, neighborhood, and the interactive and business intelligence benchmarks. These can be efficiently supported by our

low-latency index and ability to find existing edges. For the workload, like other dynamic data structures, we do not support labels. For static use cases, the issue is addressed by Mhedhbi et al. [40].

9 CONCLUSIONS

Sortedledton is a sorted, simple, transactional graph data structure that executes up to 5 million edge updates per second, supports analytical, , and traversal workloads with runtimes within 1.2x on average of while needing only 2x the space of . Furthermore, it runs analytics and a high number of updates concurrently. We achieve this by reusing existing data structures.

We construct Sortedledton based on two key principles. First, a universal graph data structure needs to store neighborhoods in sets to support , 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.

ACKNOWLEDGMENTS

We thank Dean De Leo for taking a big step towards making graph data structure research comparable with his test driver. We thank Maximilian Bandle, Dominik Durner, and our reviewers for their valuable feedback.

References

- [1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Riz. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 431–446. <https://doi.org/10.1145/2882903.2915213>
- [2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martiñez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Piñez, Mirko Spasic, Benjamin A. Steer, Gábor Székely, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 <http://arxiv.org/abs/2001.02299>
- [3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindacker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 1421–1432. <https://doi.org/10.1145/3183713.3190654>
- [4] Timothy G. Armstrong, Vamsi Ponnepantla, Dhruva Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 1185–1196. <https://doi.org/10.1145/2463676.2465296>
- [5] Scott Beamer, Krste Asanovic, and David A. Patterson. 2013. Direction-optimizing breadth-first search. *Sci. Program.* 21, 3-4 (2013), 137–148. <https://doi.org/10.3233/SPR-130370>
- [6] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 <http://arxiv.org/abs/1508.03619>
- [7] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*. IEEE Computer Society, 56–65. <https://doi.org/10.1109/IISWC.2015.12>
- [8] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefer. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* abs/1910.09017 (2019). arXiv:1910.09017 <http://arxiv.org/abs/1910.09017>
- [9] Jan Bittacher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*. ACM, 2:1–2:8. <https://doi.org/10.1145/3399666.3399908>
- [10] Jan Bittacher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proc. VLDB Endow.* 13, 2 (2019), 128–141. <https://doi.org/10.14778/3364324.3364328>
- [11] Mihai Capota, Tim Hegeman, Alexandru Iosup, Arnau Prat-Piñez, Orri Erling, and Peter A. Boncz. 2015. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*. ACM, 7:1–7:6. <https://doi.org/10.1145/2764947.2764954>
- [12] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 63–78. <https://doi.org/10.1145/2723372.2750545>
- [13] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137. <https://doi.org/10.1145/356770.356776>
- [14] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2006. Lock-Free Dynamically Resizable Arrays. In *Principles of Distributed Systems, 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Proceedings (Lecture Notes in Computer Science)*, Vol. 4305. Springer, 142–156. https://doi.org/10.1007/11945529_11
- [15] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 918–934. <https://doi.org/10.1145/3314221.3314598>
- [16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-ikke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwillig. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 1243–1254. <https://doi.org/10.1145/2463676.2463710>
- [17] Bolin Ding, Kai Zeng, and Wenyan Yu. 2020. Alibaba Sponsor Talk at VLDB.
- [18] Vinícius Vitor dos Santos Dias, Carlos H. C. Teixeira, Dorgival O. Guedes, Wagner Meira Jr., and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1357–1374. <https://doi.org/10.1145/3299869.3319875>
- [19] David Ediger, Robert McCall, E. Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*. IEEE, 1–5. <https://doi.org/10.1109/HPEC.2012.6408680>
- [20] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. 2016. Parallel Graph Processing: Prejudice and State of the Art. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*. ACM, 85–90. <https://doi.org/10.1145/2851553.2851572>
- [21] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chahimi, Andrey Gubichev, Arnau Prat-Piñez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 619–630. <https://doi.org/10.1145/2723372.2742786>
- [22] Per Fuchs, Peter A. Boncz, and Bogdan Ghit. 2020. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. In *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, Portland, OR, USA, June 14, 2020. ACM, 4:1–4:11. <https://doi.org/10.1145/3398682.3399162>
- [23] Alastair Green. 2019. *THE GQL MANIFESTO*. <https://gql.today/>
- [24] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Integrating Column-Oriented Storage and Query Processing Techniques Into Graph Database Management Systems. *CoRR* abs/2103.02284 (2021). arXiv:2103.02284 <https://arxiv.org/abs/2103.02284>
- [25] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabuk, Quannan Li, and Jimmy J. Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 7, 13 (2014), 1379–1380. <https://doi.org/10.14778/2733004.2733010>
- [26] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *Structural Information and Communication Complexity, 14th International Colloquium, SIROCCO 2007, Castiglione, Italy, June 5-8, 2007, Proceedings (Lecture Notes in Computer Science)*, Vol. 4474. Springer, 124–138. https://doi.org/10.1007/978-3-540-72951-8_11
- [27] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyu Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.* 41, 3 (2018), 64–75. <http://sites.computer.org/debull/A18sept/p64.pdf>
- [28] SQL ISO. 2020. *ISO/IEC CD 9075-16*. <https://www.iso.org/standard/79473.html>
- [29] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2009. Optimistic parallelism requires abstractions. *Commun. ACM* 52, 9 (2009), 89–97. <https://doi.org/10.1145/1562164.1562188>
- [30] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. USENIX Association, 249–263. <https://www.usenix.org/conference/fast19/presentation/kumar>
- [31] János Kertész, and Kunegis. 2013. KONECT: the Koblenz network collection. *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*. International World Wide Web Conferences Steering Committee / ACM, 1343–1350. <https://doi.org/10.1145/2487788.2488173>
- [32] Per-ikke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwillig. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (2011), 298–309. <https://doi.org/10.14778/2095686.2095689>
- [33] Dean De Leo and Peter A. Boncz. 2019. Fast Concurrent Reads and Updates with PMAs. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, Amsterdam, The Netherlands, 30 June 2019. ACM, 8:1–8:8. <https://doi.org/10.1145/3327964.3328497>
- [34] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066. <http://www.vldb.org/pvldb/vol14/p1053-leo.pdf>
- [35] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>
- [36] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 363–374. <https://doi.org/10.1109/ICDE.2015.7113298>
- [37] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 135–146. <https://doi.org/10.1145/1807167.1807184>

- [38] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 604–615. <https://doi.org/10.1109/ICDE.2014.6816685>
- [39] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*. ACM, 25:1–25:16. <https://doi.org/10.1145/3302424.3303974>
- [40] Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, and Semih Salihoglu. 2020. A+ Indexes: Lightweight and Highly Flexible Adjacency Lists for Graph Database Management Systems. *CoRR abs/2004.00130* (2020). arXiv:2004.00130 <https://arxiv.org/abs/2004.00130>
- [41] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704. <https://doi.org/10.14778/3342263.3342643>
- [42] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162. <https://doi.org/10.1145/128765.128770>
- [43] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 677–689. <https://doi.org/10.1145/2723372.2749436>
- [44] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluç. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhui Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1372–1385. <https://doi.org/10.1145/3448016.3457313>
- [45] Kenneth Platz, Neeraj Mittal, and S. Venkatesan. 2019. Concurrent Unrolled Skiplist. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. IEEE, 1579–1589. <https://doi.org/10.1109/ICDCS.2019.00157>
- [46] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888. <https://doi.org/10.14778/3229863.3229874>
- [47] James Reinders. 2007. *Intel threading building blocks - out of the box for multi-core processor parallelism*. O'Reilly. <http://www.oreilly.com/catalog/9780596514808/index.html>
- [48] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 1583–1598. <https://doi.org/10.1145/2882903.2882958>
- [49] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM. <https://doi.org/10.1137/1.9780898718003>
- [50] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2017), 420–431. <https://doi.org/10.1145/3186728.3164139>
- [51] Aneesh Sharma, Jerry Jiang, Praveen Bommanavar, Brian Larson, and Jimmy J. Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (2016), 1281–1292. <https://doi.org/10.14778/3007263.3007267>
- [52] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. ACM, 135–146. <https://doi.org/10.1145/2442516.2442530>
- [53] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 18–32. <https://doi.org/10.1145/2517349.2522713>
- [54] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 237–251. <https://doi.org/10.1145/3037697.3037748>
- [55] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 465–477. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting
- [56] Xiaowei Zhu, Zhisong Fu, Zhenxuan Pan, Jin Jiang, Chuntao Hong, Yongchao Liu, Yang Fang, Wenguang Chen, and Changhua He. 2021. Taking the Pulse of Financial Activities with Online Graph Processing. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 84–87. <https://doi.org/10.1145/3469379.3469389>
- [57] Xiaowei Zhu, Marco Sereni, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034. <https://doi.org/10.14778/3384345.3384351>