Teseo and the Analysis of Structural Dynamic Graphs

Dean De Leo
CWI
dleo@cwi.nl

Peter Boncz
CWI
boncz@cwi.nl

ABSTRACT

We present Teseo, a new system for the storage and analysis of dynamic structural graphs in main-memory and the addition of transactional support. Teseo introduces a novel design based on sparse arrays, large arrays interleaved with gaps, and a fat tree, where the graph is ultimately stored. Our design contrasts with early systems for the analysis of dynamic graphs, which often lack transactional support and are anchored to a vertex table as a primary index. We claim that the vertex table implies several constraints, often neglected, that can actually impair the generality, the robustness and extension opportunities of these systems. We compare Teseo with other dynamic graph systems, showing a high resilience to workload and input changes, while achieving comparable, if not superior, throughputs in updates and latencies in raw scans.

PVLDB Reference Format:
Dean De Leo and Peter Boncz. Teseo and the Analysis of Structural Dynamic Graphs. PVLDB, 14(6): 1053 - 1066, 2021.
doi:10.14778/3447689.3447708

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

1 INTRODUCTION

Graph analysis involves the execution of computationally expensive and relatively long running algorithms over *structural graphs*. These are homogeneous graphs, simply composed of vertices, edges and, possibly, a weight attached to their elements. For sizable problems, the established approach is to run these algorithms in one among the many systems for *static graphs* already existing [62]. They are static, in the sense that, the studied graphs need to be first extracted from other primary data sources, preprocessed and finally loaded into the final tool for the analysis. In case of any change, the whole cycle needs to be repeated, propagating the changes from their primary data sources. As such, this pipeline remains suitable only when allowed to work with somewhat stagnant data.

The alternative is the analysis of *dynamic graphs*. These are graphs whose constituent structure and properties can continuously change, opening the opportunity of working constantly up-to-date information. For this scenario, there have been studies aimed at doing away with the ETL pipeline and running the graph algorithms on transactional primary data sources. On the one hand, this can be accomplished on a native and feature-rich graph DBMS, with Neo4j arguably representing the most compared system to day. On the other hand, there have been attempts to adapt existing Relational DBMSes (RDBMS) for graph analysis [22, 33].

Upon inspection, these approaches have been shown to come short in terms of performance [48, 50], compared to systems for static graphs, while offering a somewhat more restricted abstraction model. Nowadays, single machines can process relatively large graphs [51], and, recently, for this architecture, several libraries to tackle dynamic graphs have been published [20, 35, 37, 46, 63]. These systems, regardless of their physical implementation, all expose a logical view based on *adjacency lists*, one of the most conventional abstractions presented in algorithmic textbooks [14]. In this model, the user operates on the graph by selecting one specific vertex at-a-time and iterating over its edges and/or its properties. Published works for the above libraries report comparable latencies, in graph analysis, w.r.t. systems for static graphs. Furthermore, they show to be capable of very high throughput in updates, effectively suggesting the analysis of real-time dynamic graphs.

Regrettably, most of these systems forgo transactional support [20, 35, 43, 46] or incur overhead [63]. Analysis of dynamic graphs is increasingly relevant in sectors such as Finance, Insurance, Logistics, Media and Infrastructure and include applications like Fraud Detection [4], Threat Detection [34], Information Diffusion [24], Risk Analysis, Compliance as well as Supply Chain Optimization, and more [55]. Without the isolation and consistency of transactions, in presence of concurrent changes, queries and computations will yield incorrect results. For instance, in fraud detection, the wrong customer could be flagged, or, in a computer network, a suspicious authentication could pass unnoticed.

In this paper, we present Teseo, a library for the analysis of dynamic structural graphs with the addition of full transactional support. Similarly to the above mentioned libraries, Teseo also targets main-memory standalone machines and exposes to the end users the same abstraction of adjacency lists. Teseo guarantees snapshot isolation [56], with a protocol modelled after HyPer [52]. Furthermore, it introduces a novel design based on *sparse arrays*[1], large dynamic arrays interspersed with gaps to maintain a sorted order. Sparse arrays are the building block that form the leaves of *fat trees*, where the graph is finally stored.

This design presents several advantages. Large leaves, together with additional indexing, contribute to reducing random memory accesses, already compelling in graph algorithms [44] and a limiting factor of traditional B^+ trees [17, 63]. The sorted order enables long sequential scans, essential in analytical workloads [2]. The free space, arranged as a sparse array, guarantees a low cost per update while maintaining the sorted order. In-place updates and aggressive pruning curtail the overhead of transactional support. Through these techniques, Teseo aims to exploit the access patterns of graph analysis, while retaining sustainable throughput in updates.

[1] Also known as Packed Memory Arrays (PMA) in the literature [10].

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447708

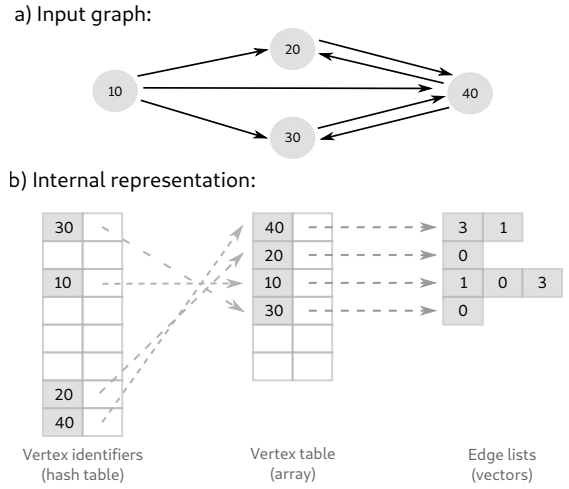

Figure 1: Systems for dynamic graphs typically rely on a vertex table, an array where all vertices are stored. Figure b) depicts the content of a vertex table for the input graph a). Vertex identifiers are relabelled by a hash table to the indices in the vertex table: $10 \rightarrow 2$**,** $20 \rightarrow 1$**,** $30 \rightarrow 3$ **and** $40 \rightarrow 0$**.**

Designs based on trees, and B^+-trees in particular, are predominant in RDBMSes [56], but they are a novelty with respect to the high-throughput and low-latency systems for dynamic graphs we examined [20, 35, 37, 46, 63]. These systems share a common scheme, outlined in Figure 1, ultimately anchored to a *vertex table*, a container where all vertices are stored together and whose entries refer to lists of their adjacent edges. The vertex table is usually implemented as a pre-allocated array and, often, with a fixed capacity. This solution ensures quick vertex look-ups, in $O(1)$.

We claim that a vertex table presents a number of limitations that are frequently overlooked. Because it is based on a static array, most of the examined systems do not allow vertex deletions, whereas insertions are usually bound up to the pre-allocated capacity. To support arbitrary vertex identifiers, the vertex table needs to be associated to a hash table, to map the identifiers to their indices in the array. As none of the examined systems cover the hash table by a suitable form of concurrency isolation, changes to vertices cannot be overlapped with concurrent reads. The edges associated to the same source are stored in an unindexed sequence, typically, unsorted. The cost of single edge look-ups, and updates as consequence, depends on the length of the sequence and can be expensive in vertices with a high degree. These systems can also become fragile in presence of skew in updates, as they tend to partition critical sections at the granularity of the single vertices. Finally, as described in Section 3, most algorithms for graph analysis tend to manifest one of two common patterns, which we name *sequential* and *random patterns*, where data accesses are predominantly sequential and random, respectively. While all systems perform similarly on random access, the updatable sparse array storage of Teseo is optimized for sequential access, unlike typical vertex table designs where updates disturb the access pattern.

In our experimental results, Teseo can compete favourably, in terms of both updates and raw scans, against these systems. Our results complement and, at first glance, can conflict with the experimental results reported in previous work. As we detail in Section 8, in 3/4 of our competitors [20, 35, 46] updates have only been evaluated as bulk loads, while, generally, different, but, occasionally incompatible implementations of the same graph algorithms have been compared. Our experiments, instead, target fine-grained updates, whereas all systems are evaluated in the LDBC Graphalytics standard benchmark [29], under the same implementation for the graph algorithms. This eases a direct comparison of the involved systems as algorithmic merits are detached. Our contributions are:

- We present Teseo, a new system for the analysis of dynamic structural graphs with full transactional support. It relies on a novel design based on sparse arrays and fat trees.
- We compare Teseo against some of most recent, prominent and compatible systems of the last decade, reporting similar or superior performance in terms of updates and raw scans.
- We uncover a different picture than what was provided by our competitors in their works. Our experiments tend to be more complex and comprehensive, and we often measured a lower performance than the results previously published.

The rest of the paper is organised as follows. In Section 2, we summarise sparse arrays and other components prerequisites of Teseo. In Section 3, we motivate our design rationale. Section 4 details the layout and the operations of the fat tree. In Section 5, we describe how Teseo handles concurrency, while Section 6 delves with a few remaining key features. In Section 7, we evaluate Teseo against other comparable systems, analysing the throughput of both insertions and general updates and their performance in graph analysis. In Section 8, we contrast our results with those published by our contenders. We review related work in Section 9 and conclude in Section 10.

2 PREREQUISITES

This section summarises sparse arrays [16] and, then, it reviews hybrid latches [12, 40] and the *Multi-Version Concurrency Control* (MVCC) protocol of HyPer [52], all central components of Teseo.

2.1 Sparse arrays

A *sparse array* is an array where elements are stored according to a key order, interspersed with *gaps*. These are empty slots, provisioned to accommodate future potential insertions. The supported operations are insertions, deletions, point look-ups and range scans. Point look-ups and range scans are akin to sorted dense arrays, with encountered gaps ignored or skipped.

Given its capacity C_A, the array is split into $\frac{C_A}{C_S}$ contiguous chunks, named *segments*, of a predetermined size C_S. Figure 2a depicts an instance of a sparse array. Insertions can be performed into a segment until it becomes full, that is, there are no more gaps. Analogously, deletions from a segment replace an element with a gap and can be performed until the segment becomes half full. At that point, a *rebalance* operation is carried out. The intuition is to visit the adjacent segments and share their elements. The sequence of segments involved in a rebalance is called a *window*.

In a sparse array, there is a specific formula to determine the window W. Given $|W|$ the number of segments in W, $cardinality(W)$ as the cumulative number of elements in the segments of W, $h =$

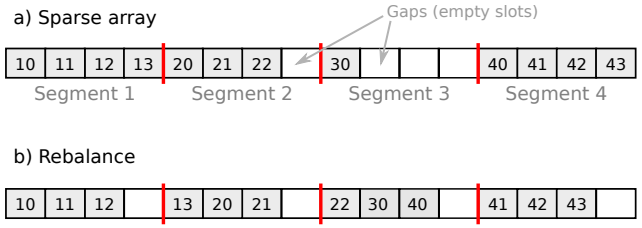

Figure 2: a) A sparse array of capacity $C_A = 12$, with 4 segments of size $C_S = 4$. b) The outcome of a rebalance for the whole array a).

$\lceil \log_2 \frac{C_A}{C_S} \rceil$ and a given set of constants $0 < \rho_1 < \rho_h \leq \tau_h < \tau_1 \leq 1$, then the objective is to find the smallest window W such that:

$$low(W) \leq cardinality(W) \leq high(W)$$
$$low(W) = \left\lfloor \rho_h - (\rho_h - \rho_1) \cdot \left(\frac{h - \log_2 |W|}{h} \right) \right\rfloor \cdot C_S \cdot |W| \quad (1)$$
$$high(W) = \left\lceil \tau_h + (\tau_h + \tau_1) \cdot \left(\frac{h - \log_2 |W|}{h} \right) \right\rceil \cdot C_S \cdot |W|$$

In practice, starting from an unbalanced segment, we create a window from it and iteratively add its adjacent segments until (1) is satisfied. Once a window is found, all contained elements are equally spread among its segments, as in Figure 2b. If the formula (1) cannot be satisfied even for the largest window, the array is resized to a new capacity $C'_A = 2N/(\rho_h + \tau_h)$.

The constants ρ_1, ρ_h, τ_h, τ_1 are named *density thresholds* and are an input parameter defined by the implementation. Here, we simply set $\rho_1 = 0.5$, $\tau_1 = 1$, $\rho_h = \tau_h = 0.75$, a choice that ensures that the fill factor of a resize will be $\approx 75\%$, whereas the overall array is always filled at 50% [31]. The segment size C_S is also an implementation parameter, conceptually analogous to the leaf size of a B$^+$ tree. In this paper, we assume $C_S = 4$ KB as it was experimentally shown in previous work [17] to provide, for in-memory sparse arrays, a good balance between the cost of updates and that of scans.

In the RAM model, the worst-case complexity of an update is $O(N)$, due to a resize or a large rebalance. The formula (1) guarantees a tighter bound of $O(log_2^2 N)$, per update, in the amortised worst-case analysis [10, 32]. In the I/O model [3], assuming $C_S = O(B)$, the complexity turns to $O((log_2^2 N)/B)$ per update[2] [9, 10]. The worst-case occurs in presence of skew, when elements are continuously either inserted into or deleted from the same segment. If the keys in the updates follow a uniform distribution, then the amortised average-case becomes $O(\log_B N)$ [8, 32]. Finally, range scans match the optimal worst-case $O(R/B)$ with sequential accesses [9, 10], where R is the number of elements in the range.

2.2 Hybrid latches

A *hybrid latch* combines a conventional latch with an *optimistic latch* [40]. Like a conventional latch, it can be acquired by either multiple readers or a single writer in mutual exclusion. The latch is also augmented with a *version*, a counter incremented every time a writer passes through it. A reader can alternatively acquire the latch *optimistically*, by checking the version both before and after accessing the content of the critical section. If the values are equal, it guarantees that, in the meantime, the content in the critical section was not altered. Otherwise, the whole read operation needs to be repeated. In read-intensive workloads, optimistic reads avoid modifying the internal state of a latch, a potential cause both of contention and of additional traffic in the CPU cache hierarchy.

2.3 MVCC

In HyPer, every data item has associated a pointer, potentially `null`, with the head of a linked list of *versions*. The chain contains the history of alterations applied by different transactions to the related data item during its lifetime. A reader can always access the corresponding visible version by traversing the history. A writer *locks* the data item until its transaction terminates, by commit or roll back, and prepends a new version to the data item's history. Upon conflict, a change is rejected and an error is thrown to the user. Finally, versions are marked with the commit time of the transaction that created them, so that other transactions can identify which change in the history is visible to them.

Periodically, inaccessible versions are pruned by a Garbage Collector (GC). A data item without a history is implicitly visible to all transactions. This enables an alternative "fast code path", where transactions do not incur the overhead of checking the versions.

3 OVERVIEW

3.1 Opportunities

Most algorithms for graph analysis, including those in Graphalytics, follow either a sequential or a random pattern, where a significant part, if not most, of the completion time is spent. A goal of Teseo is to handle both patterns as efficiently as possible. In the sequential pattern, the bulk of an algorithm has the following structure:

```
1: while condition do
2:     for all v in V do
3:         for all e in edges(v) do
4:             result[v] = f(result[v], v, e, ...)
5:         end for
6:     end for
7: end while
```

Here, an algorithm accesses all vertices and edges of the graph in a strict sequential order, repeating the same computation until a certain condition (line 1) is satisfied. An example of this pattern is PageRank, where the score of all vertices is computed as a function of all their neighbours and their score at the previous iteration.

The random pattern is similar to the sequential pattern, with the exception that the vertex fetched at line 2 cannot be pre-established statically, but it is determined at run-time. For instance, in a shortest path algorithm, the vertex examined at each iteration is typically extracted from a minimum priority queue.

In both patterns, once a vertex is accessed, then all of its outgoing or, in some cases, incoming edges are also retrieved.

We note that both patterns are effectively captured by the popular *Compressed Sparse Row* (CSR) format [25, 61][3], a straightforward graph layout employed by frameworks for static graphs. In the CSR, all vertices and edges are stored in two contiguous arrays, with the edges logically sorted by the pair ⟨source, destination⟩. For the sequential pattern, accessing vertices and edges is very efficient

[2]The parameter B is the conventional block size of the I/O model [3].

[3]Sometimes equivalently referred as *Compressed Row Storage* (CRS) in the literature.

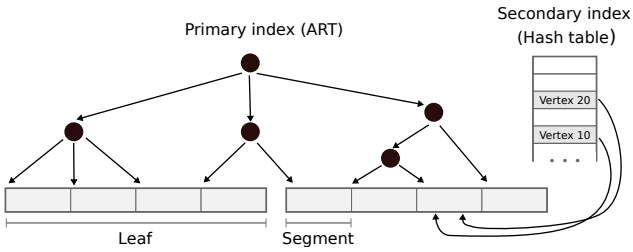

Figure 3: Sketch of a fat tree with two leaves.

because the data retrieval becomes a single sequential scan of these two arrays. Whereas in the random pattern, a vertex and its first neighbours can be still fetched with only two memory accesses. Finally, the usage of implicit pointers and contiguous arrays in the CSR also leads to a narrow memory footprint, further decreasing the cost of random memory look-ups [19].

Vertex tables can also be effective in the random pattern. Although their working set is larger than the CSR, vertex look-ups and the retrieval of their first adjacent edges can exhibit a constant cost. On the other hand, in the sequential pattern they are less efficient than the CSR, because logically consecutive vertices refer to unrelated memory regions, causing expensive random accesses, particularly detrimental on vertices with a low degree.

3.2 Graph storage

For a dynamic graph representation, our starting point for Teseo is to store the whole graph in a single in-memory B$^+$ tree. Vertices and edges are stored together. We further assume that vertex identifiers are derived from an ordered universe, e.g. natural integers. Then, we preserve a sorted order in the tree: vertices are stored according to their natural order, while directed edges are stored immediately after their source vertex, and, when sharing the same source, according to their destination vertex. We represent undirected edges $a - b$ as two directed edges: $a \rightarrow b$ and $b \rightarrow a$.

While this initial representation is effective for updates, it does not capture the patterns of graph analysis. The sequential pattern is dominated by sequential scans. Conventional B$^+$ trees are typically composed of leaves of one memory page (4KB) or a few more [23]. This causes expensive random memory jumps in scans, due to frequent leaf traversals [17, 63]. On the other hand, on vertices with a small degree, the random pattern will be instead dominated by point look-ups, which have a logarithmic cost in B$^+$ trees.

In Teseo, we mitigate these issues with a new variant of the B$^+$ tree, which we name *fat tree*, broadly sketched in Figure 3. In a fat tree, we extend the size of the leaves to the order of MBs. Scans will now traverse less leaves, improving the latency of the sequential pattern. Furthermore, we introduce a secondary index, a hash table, that maps each vertex to its physical position in the tree. The hash table guarantees that point look-ups, a dominating factor of the random pattern, now feature a cost, on average, of $O(1)$.

But naively extending the size of a leaf also proportionally impairs the latency of updates [17], as generally more elements need to be moved to maintain the sorted order. To regain the efficiency in updates, we organise the leaves of the tree as sparse arrays. This originates in a hybrid design, where full segments inside a leaf are rebalanced as in sparse arrays, while array resizes are substituted by leaf splits/merges as in B$^+$ trees. Finally, for efficient point look-ups,

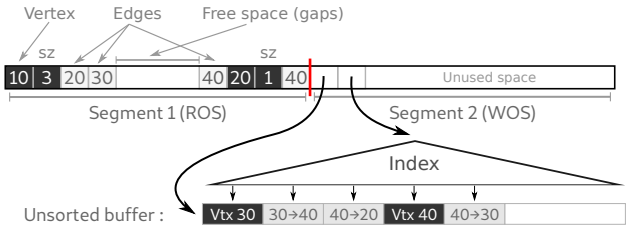

Figure 4: The storage of the graph of Figure 1a) over two segments. The first segment is in ROS format and the second one is in WOS format.

demanded by updates, a fat tree uses ART [41, 42], a form of trie, as primary index, with one search key per segment.

3.3 Asynchronous operations

Structural operations in a fat tree are more expensive than in a traditional B$^+$ trees, because its leaves are three orders of magnitude larger. In a fat tree, the structural operations are segment rebalances and leaf splits/merges. Moreover, proper maintenance of the indices can further aggravate the peak latency of updates.

A crucial point of Teseo is to delay and perform all the above operations asynchronously by *service threads*. These are background threads spawned and managed by the system, responsible for maintenance tasks and GC. In particular, when a segment becomes full, a writer can always keep adding new elements into a write-optimised side buffer. The run-time will eventually rebalance the segment, clearing the buffer and physically redistributing the elements.

4 FAT TREE

In this section, we detail the layout and the operations in a fat tree. Hereafter, we assume that vertex identifiers are represented by 8-byte integers, although our description can be reasonably generalised to any arbitrary domain with a sorted order.

4.1 Leaf layout

All vertices and edges are stored in the leaves of a fat tree. A leaf is organised as a sparse array. It is composed by a contiguous sequence of page-sized segments. A segment consists of some metadata and can be alternatively arranged in two different layouts: *read-optimised segment* (ROS), where elements are physically stored sorted in the segment, or *write-optimised segment* (WOS), where elements are instead stored unsorted in an outside buffer. Figure 4 depicts the representation of the graph of Figure 1a) over two segments.

In Teseo, the metadata is formed by a hybrid latch, for concurrency purposes, and a pair of fence keys. Fence keys define the [min, max) values for the interval covered by a segment. The max fence key of a segment is equal to the min fence key of the following segment. Fence keys are only altered during rebalances. Note that, in a leaf, there are no pointers towards its siblings. Instead, a thread can find the next leaf through a look-up in the index for the max fence key of the last segment in the current leaf.

In a ROS, data items are physically packed together towards the start and the end of the segment, while the free space resides in the middle. A data item can be either a vertex or an edge. A vertex is represented with 16 bytes: 8 bytes for its identifier and 8 bytes for the field sz, the number of edges following the vertex in the segment. An edge simply consists of 8 bytes for its destination.

When the edges adjacent to the same source span over multiple segments, we duplicate the source vertex in each involved segment.

A new element can be inserted into a ROS in either the left or right area of the segment, by reclaiming some free space from the middle. To maintain the sorted order, the existing elements are physically shifted accordingly. When the free space exhausts, a writer changes the segment into a WOS, which consists of a pointer to a buffer, where elements can only be appended at the end, accompanied by a secondary dense index for quick point look-ups. In the buffer, an edge is explicitly stored as a 16 bytes pair ⟨source, destination⟩. The index is a sequential ART trie.

The ROS format favours sequential scans while the WOS format favours updates. Most of the time, a segment is expected to remain in ROS format, where scans are analogous to dense arrays, but skipping the middle area of free space, and a limited amount of updates can be still accommodated. The WOS is instead a temporary state. It is eventually transformed back into a ROS during a rebalance.

4.2 Structural operations

In Teseo, structural operations are performed by service threads. When the free space in a segment is starting to deplete, a writer *proactively* notifies the run-time. In turn, the run-time forwards the request, but only after a predefined delay D, to a service thread. At that point, the service thread follows the same procedure of Section 2.1 to determine the window to rebalance. Eventually, the service thread redistributes the elements among the involved segments, and when present, switches back their format from WOS to ROS.

We implement sparse array resizes, to a new capacity C'_A, as follows. The leaves of the fat tree are variable-sized, with a size between C_L and $C_L/2$, with C_L an implementation parameter. In a resize, if $C'_A \leq C_L$, we recreate the leaf with a new capacity C'_A. Otherwise, we split, as in a traditional B$^+$ tree, the leaf in two, with each one having capacity $C'_A/2$. With the density thresholds of Section 2.1, this procedure always produces leaves with a fill factor of $\approx 75\%$, while the capacity of the leaves is at least $C_L/2$. Finally, occasionally, due to the extra capacity of the WOS segments, a resize could similarly split a leaf in more than two leaves.

The purpose of the delay D is to mitigate the worst-case complexity of sparse arrays, which occurs in presence of update skew. With skew, a given segment is constantly accessed by writers. By filling a WOS and delaying the rebalance, many more elements can be redistributed in a single rebalance. This resembles the effect of a batch insertion into a sparse array, which has only a logarithmic cost [11]. And, while a large WOS is detrimental to readers, these are already impaired, in the first place, by the concurrent writers.

Finally, in Teseo, contrary to traditional sparse arrays, we never rebalance a segment when it becomes underfilled due to deletions. Rather, a service thread periodically visits the fat tree and merges together neighbour leaves. A merge occurs when the fill factor of the resulting leaf is less or equal than $\rho_h = 0.75$.

4.3 Indices

As described in Section 3.2, Teseo relies on two indices for the leaves: an ART trie and a hash table. The trie is a clustering sparse index, used for general point look-ups and updates. Its search keys are the min fence keys of the segments. The hash table maps the vertices to their physical position inside a segment. It is used to initialise scans, as they always start from a source vertex. In a NUMA machine, the hash table is duplicated in all sockets. In our implementation, the hash table is loosely based on a custom lock-free variant of [47].

Both indices are only updated asynchronously, by the service threads, during a structural operation. Therefore, occasionally, a concurrent thread t can retrieve an outdated entry. For the primary index, t detects an incorrect segment by checking whether the key searched belongs to the interval defined by the fence keys. If not, as rebalances tend to spread elements nearby, t can simply continue its search in the proximate segments, checking again the fence keys.

In the secondary index, each entry consists of a tuple ⟨`segment`, `version`, `offset`⟩, where the `segment` is a pointer to the related segment, `version` refers to the segment's latch version when the entry was last updated and `offset` to the relative position of the data item in the segment. Similarly to the primary index, a reader detects if a segment is invalid through the mechanism of the fence keys. Furthermore, if the segment is in ROS format, a reader checks whether the `version` retrieved matches the one in the latch. If so, it implies that no writes took place in the meanwhile and the reader can directly jump to the source vertex through the `offset`. Otherwise, the reader falls back to a linear search, potentially skipping unrelated edges via the field `sz` of prior vertices. For segments in the WOS format, the `version` and the `offset` are ignored, and a reader retrieves the searched element through the WOS index.

4.4 Weights

Weights are simply stored *in-line*, together with its associated element, in a WOS segment. Whereas in the ROS format, weights are stored *out-of-place*. For fixed-length values, the idea is to imagine the sequence of segments in a leaf as a single logical array K. Then, for each weight w, we append an array V_w of equal capacity at the end of the leaf. If the segment is in ROS format, given an edge at position i in K, its associated weight w will be stored at the same position i in V_w. In presence of updates, we shift the weights in V_w by the same amount the elements are moved in K.

The layout of ROS resembles *vertical partitioning*, a characteristic of column stores [2]. It is geared more towards analytical workloads. Computations that only visit the graph topology are advantaged, since they operate on a smaller working set. Conversely, updates are more expensive, as they now affect multiple memory locations.

Our prototype does not yet support variable-length values, as they are uncommon in structural graphs. Still, in principle, they could be enabled with an extra layer of indirection [2].

5 CONCURRENCY

There are two orthogonal dimensions to concurrency. Section 5.1 first describes how multiple logical threads can operate concurrently on Teseo data structures. Section 5.2, conversely, describes how Teseo enables multiple transactions to operate concurrently while respecting the isolation boundaries.

5.1 Parallelism

Every segment is protected by a hybrid latch. Writers acquire it in exclusive mode, whereas point look-ups, a fine granularity operation, optimistically. For scans, it depends on the kind of transaction. For optimisation purposes, a transaction can be optionally created, by an end user, as read-only. In a scan, a read-only (RO) transaction acquires the latch in conventional shared mode, whereas a

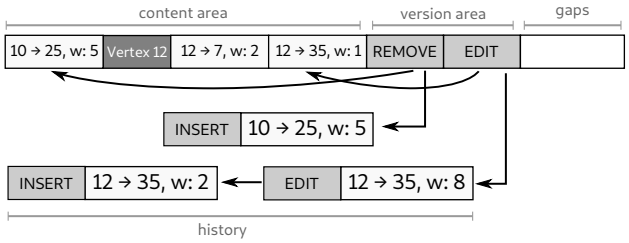

Figure 5: Sample layout of a ROS segment with one vertex 12, three edges: $10 \longrightarrow 25$, $12 \longrightarrow 7$ and $12 \longrightarrow 35$ plus an attribute w, and two chains of versions. Note that, here, for simplicity reasons, the attribute is sketched in-line with the data item, while, in a ROS, attributes are actually always stored out-of-place and data items serialised as described in Section 4.

read-write (RW) transaction can only acquire it optimistically. The reason is that our latches are non-reentrant. If we had relied on shared locks for RW transactions, a user could hit a deadlock when altering the content of a segment currently visited by a cursor. ART only relies on optimistic latches [42], one per node of the trie.

During a rebalance, a thread needs to hold all latches for the related segments in exclusive mode. When computing the window to rebalance, competing threads may try to include the same segments and acquire the same latches in different order. To avoid the risk of deadlocks, every leaf contains an additional latch, only respected by rebalancers. This latch needs to be acquired before computing the window to rebalance, effectively serialising this operation inside a leaf. Once the window has been identified, the actual relocation of the data items inside the segments can be concurrent to the other rebalancers and the leaf's latch can be released.

As the search of the window to rebalance is serial, a thread t_1 can also encompass (or steal) a neighbour window computed by another thread t_2, freeing t_2 by the task of rebalancing. For this purpose, segments contain an extra pointer *ptr* in their metadata. When t_1 visits a segment with an empty value for *ptr*, it sets it to its internal state. Otherwise, it steals the window W computed by the competing thread, which must be blocked on a latch, and replaces *ptr* in all segments of W with its state. Finally, after a window has been computed, t_1 resets the value *ptr* in all acquired segments.

5.2 Transactions

Users can operate on the graph only by transactions. Teseo follows the same scheme of HyPer [§2.3]. As in HyPer, in a WOS, each data item includes a pointer to its chain of versions. In a ROS, to save space, we do not preallocate a side array for the history, because most entries will be empty. Rather, we only store the pointers that are initialised. Versions also change more frequently than data items. For instance, an insertion of a new vertex logically creates both a new data item and a new version, while version pruning only removes a version.

For these reasons, we logically split the filled space of a ROS in a *content area* followed by a *version area*, as per Figure 5. The content area only contains the data items, while the version area the pointers to the versions. Pointers in the version area follow the same sorted order of the data items and are tagged both with a back reference to the data item they refer and with the kind of change performed: insert, edit or remove. A data item without an associated version is visible to all transactions.

The rest of the protocol works like HyPer. An update alters an element *in place* and prepends the previous value in the chain of versions. Versions are only pruned by service threads, either during a rebalance or a periodic visit of a background thread. When the version area is empty, a scan switches to the "fast code path", fetching the elements directly.

6 FURTHER ASPECTS

6.1 Sequential scans

In the adjacency list model, a scan always starts from a vertex and fetches its edges. In Teseo, scans are accomplished by a cursor. When a cursor is opened, the system executes a point look-up in the secondary index for a given vertex v_i, acquires a read latch for the segment where v_i is located and searches the element inside the segment. To retrieve its edges, the system reads the elements in the segment one after another, unless the same data item must be discriminated among multiple versions due to transaction isolation.

Teseo further applies an additional optimisation for the sequential pattern. When all edges of v_i have been fetched, the cursor is not implicitly closed, but it holds the last retained latch and the position of the next data item, vertex v_{i+1}. If a user eventually requests a range scan for v_{i+1}, the cursor will resume its last position, avoiding the opening cost. Otherwise, the cursor will be cleared and the scan will restart from scratch. This optimisation makes the sequential pattern more effective, because it is implicitly transformed into a single sequential scan over the fat tree.

6.2 Logical vertex identifiers

A large collection of graph algorithms is described or already exists for static graphs in the public domain, assuming that all vertices can be identified as integers in $[0, |V|)$, where $|V|$ is the number vertices in the graph. Frequently, these algorithms employ static arrays or bitmaps as side data structures, leveraging the association between an index in the array and a vertex. In a dynamic graph, vertices change over time and their identifiers are arbitrary values.

To ease the porting of existing algorithms, transactions expose an abstraction where vertices can be interchangeably referred by their real identifiers or their rank in the sorted order. This functionality is implemented by a materialised view, transparently computed upon first request. In our current implementation, computing the materialised view still requires a pass of the whole graph.

The materialised view is implemented differently depending on the kind of transaction. In RO transactions, we employ a static hash table to translate a real vertex identifier into its logical counterpart, and by a static array for the other direction. In NUMA architectures, the data structures are duplicated in all sockets. In RW transactions, the materialised view is implemented by a single counting B^+ tree, synchronously updated as the user alters the graph.

7 EVALUATION

We present an evaluation of Teseo in terms of both throughput for updates and latency for common graph algorithms. For our evaluation, we rely on the LDBC Graphalytics Benchmark [29, 30]. The benchmark specifies six algorithms or kernels: BFS, weighted

Graphs	Vertices $\|V\|$	Edges $\|E\|$
dota-league	$\approx 61 \times 10^3$	$\approx 51 \times 10^6$
graph500-22, uniform-22	$\approx 2.4 \times 10^6$	$\approx 64 \times 10^6$
graph500-24, uniform-24	$\approx 8.8 \times 10^6$	$\approx 260 \times 10^6$
graph500-26, uniform-26	$\approx 32.8 \times 10^6$	$\approx 1.05 \times 10^9$

Table 1: Number of vertices $|V|$ and edges $|E|$ in the input graphs evaluated.

shortest paths from a single source (SSSP), weakly connected components (WCC), PageRank (PR), local triangle counting (LCC) and a variation of the community detection via label propagation (CDLP) algorithm [54]. In our implementation, BFS, PR and CDLP exhibit a sequential pattern, LCC and SSSP a random pattern, whereas WCC somewhere in between[4]. The outputs of the algorithms are deterministic, our results were validated with those expected.

Graphalytics already ships with a collection of data sets ready for evaluation, from which we selected *DOTA League* and the Graph500 graphs. DOTA League is one of the few weighted graphs available, while Graph500 is a common synthetic graph also considered by other benchmarks and comes in multiple scale factors *SF*. DOTA League exhibits a mild exponential trend in the node degree distribution, whereas Graph500 can be characterised by a power law in the tail. As both graphs are heavily skewed, we augmented our evaluations with some additional synthetic graphs, created with a uniform distribution and the same size of the Graph500 graphs. Table 1 summarises the size of all considered graphs. All graphs are *simple*, i.e. there cannot exist multiple edges between two given endpoints, and undirected. The weights are double floating point scalars associated to the edges. In the unweighted graphs, we still randomly associated a weight in $(0, 1]$ to each edge.

As Graphalytics only describes a static scenario, we extended our evaluation to cover insertions and updates in two ways. In the first case, we inserted all the vertices and all the edges, in a random permutation, of the original graph G. In the second case, we simulated general updates in G by inserting and deleting *temporary edges*, generated following the same node degree distribution of G.

We prototyped Teseo in C++, specifically for weighted and undirected graphs, as almost all graphs in Graphalytics are undirected. However, we believe that supporting directed and unweighted graphs should be a straightforward extension. For the implementation parameters of the fat tree, we set the density thresholds as in Section 2.1. We fixed the delay D of asynchronous rebalances to $200ms$, a value we inherited from previous work [16], where it proved experimentally to be a suitable trade-off between the latency of concurrent updates and scans in the processing of batch updates in sparse arrays. We finally set the maximum leaf capacity C_L to 1MB. In our internal profiling, this size bounded the latency of splits involving two leaves to less than $60ms$, a value we deemed acceptable, while still keeping the capacity of leaves large.

We conducted our experiments on a set of dual socket machines, each equipped with an Intel Xeon Gold 5115 @ 2.4GHz. Each CPU features 10 cores and 20 physical threads in SMT mode. Each machine has 384 GB of memory in total. The source code was compiled with GCC v10.2, with the optimisation flags `-O3 -march=native -mtune=native`. Each experiment has been repeated 5 times. The reported results refer to the median.

[4]The first iteration of WCC has a sequential pattern, but in the following iterations fewer and fewer vertices become active and participate in the computation.

7.1 Competitors

We considered the following competitors, all implemented in C++:

- **Stinger** [20]: a library with parallel support for both insertion and removal of edges. Edges sharing the same source vertex are stored in a linked list of blocks. The vertices are instead stored in a pre-allocated vertex table. The library only guarantees that single writes and single reads are thread safe, while scans cannot be safely executed when updates are performed concurrently.
- **LLAMA** [46]: an implementation of a *multi-version CSR*. The graph is split into a read store, consisting of static CSR arrays and multiple ordered levels (or deltas), and a write store, implemented by a custom key-value store, where all single updates are propagated. New levels need to be explicitly created through the API, having the effect of flushing all the changes from the write store into a new delta. Read operations can only access the read store, while updates can be performed concurrently in the write store. In our experiments, we explicitly issued a new level every 10 seconds, a value suggested in the original paper [46].
- **GraphOne** [35, 37]: a library also inspired by a delta design with multiple levels. The read store is logically arranged as an adjacency list, with the change sets maintained at the node granularity. The write store is implemented as a circular buffer. New deltas are periodically created asynchronously by the library, when the buffer becomes full, or they can be explicitly issued by the user. Read operations can either only access the read store, or, at the price of a performance penalty, obtain a *static view*, where all the changes currently in the write store, as present at the time of creation of the view, are incorporated. Updates can always concurrently be performed in the write store.
- **LiveGraph** [63]: a recent library targeted to HTAP workloads with full support of transactions via MVCC. The library aims to retain sequential accesses, by storing all edges adjacent to a given vertex in a vector, timestamped with their commit time. Deletions only set the end timestamp in an edge, while insertions append a new item in the vector. We evaluated LiveGraph v2020.08.29 [64]. Note that the source code is not publicly available, limiting our possibility of analysis compared to the other systems.

All four of these systems for dynamic graphs depend on a vertex table. In Graphalytics, vertex identifiers are non-contiguous non-negative 8-byte integers. As depicted in Figure 1, we need a way to map these external identifiers into the dense indices of the vertex table. Stinger and GraphOne expose to end users an auxiliary data structure for the purpose. LLAMA and LiveGraph do not offer it natively, and we augmented them with an external hash map, from Intel TBB [28], as it is also ultimately used by GraphOne for the same task. For fairness, we also disabled the disk logging in systems supporting it, that is, GraphOne and LiveGraph.

All dynamic systems conceptually expose a similar API based on adjacency lists. The API simply consists of methods to insert, get and, when supported, remove vertices and edges in the graph, iterate over the edges of a given vertex and fetch their weights. In Teseo and LiveGraph, these methods are only exposed from a local transaction object, created by the user, while in the other systems from the global graph instance. We detail the API of Teseo in [1].

As baseline for LDBC Graphalytics, we implemented the graph algorithms on top of a custom static CSR data structure. We compared

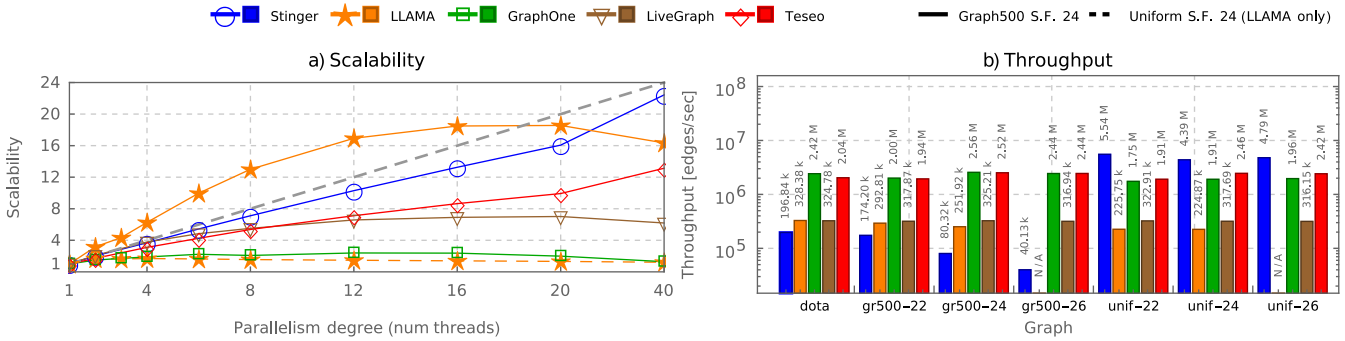

Figure 6: Plot a) shows the scalability measured in the insertions for graph500-24. For LLAMA, the plot also reports the scalability in uniform-24. Plot b) reports the average throughput measured in the insertions among various graphs.

System	Graph	BFS	CDLP	LCC	PageRank	SSSP	WCC
GraphMat	graph500-24	5.07x	35.62x	DNF	25.30x	5.48x	56.77x
	uniform-24	17.34x	40.95x	130x	53.70x	8.13x	70.02x
SuiteSparse	graph500-24	0.74x	3.01x	2.95x	5.35x	1.56x	1.33x
	uniform-24	0.51x	1.21x	3.04x	6.56x	2.17x	2.04x

Table 2: Speed-up achieved by our CSR baseline over GraphMat [53] and SuiteSparse:GraphBLAS [26]. The results refer to the *processing time* [30], that is, the completion time without the preprocessing and loading time, from the median out of 5 runs.

our baseline against GraphMat [58] and SuiteSparse:GraphBLAS v3.1.1 [6, 15], two libraries for static graphs for which a public driver for Graphalytics exists [26, 53]. At the time of writing, GraphMat is the best performing implementation for SMP machines among those evaluated by the LDBC authors [39], but despite their assistance and numerous attempts, we were never able to obtain satisfactory results in our setup. Moreover, with the exception of BFS, on SF 24, our baseline also outperformed SuiteSparse in a range between 1.2x and 6.5x. On the other hand, their BFS implementation favours smaller graphs and, on SF 26, our implementation also resulted $\approx$ 4x faster. Table 2 reports the detailed speed-ups of our baseline for graph500-24 and uniform-24. In Graphalytics, we mark as *Did Not Finish* (DNF) kernels that do not complete within one hour.

7.2 Insertions

In this experiment, we measured the throughput, in terms of insertions of edges, for each of the evaluated systems. The experiment starts with an empty data structure, and inserts, one by one and in random order, all the edges of the input graphs. We first assessed the *(strong) scalability*[5] of each system, to determine the optimal number of user threads to employ. Figure 6a shows the measured scalability on graph500-24. We note that, with the notable exception of LLAMA, we also obtained comparable results on uniform-24. Figure 6b reports the measured throughput when using the optimal number of threads, those that yielded the best results in Figure 6a.

In the discussion of the scalability, all systems, regardless of the parallelism degree, always use some form of service threads. Teseo spawns one service thread per core to execute the rebalances and the periodic garbage collection (GC). Both LLAMA and GraphOne use all the available physical threads to compact the write store, when needed. Stinger internally also depends on OpenMP.

Teseo reaches a scalability of 9 with $p = 20$ and around 13 with 40 user threads in hyper-threading. At $p = 40$, the system is oversubscribed as, when also accounting the service threads, there are more than 60 logical threads in execution. On the other hand, the service threads are only occasionally active, and when rebalances do occur, the involved segments are locked in mutual exclusion. In this context, having more user threads available increases the number of writers proceeding, while other writers are blocked for a segment being rebalanced. In terms of throughput, with $p = 40$, Teseo can insert $\approx 2.4M$ edges/sec, and about 20% less for smaller graphs due to contention. To emulate the behaviour of the other systems, we wrap each update into its own transaction. Therefore, Teseo also exhibits a throughput of roughly 2.4M transactions/sec.

Stinger can scale to a value of 22 with $p = 40$. While this translates to a throughput of up to 5.5M edges/sec with uniform graphs, it only manages 80k edges/sec on graph500-24. Before inserting an edge, Stinger executes a linear search in the edge list, to check if the edge already exists. This procedure is expensive in presence of skew, as the degree of the most referred vertices can be very large. For comparison, when implementing a sequential adjacency list in C++, on top of the STL containers `unordered_map` and `vector`, in our machine we almost reached the same throughput of Stinger with 70k edges/sec for the skewed graphs, while achieving 736k edges/sec for the uniform graphs. In contrast to Stinger, in Teseo, edges are indexed per segment of fixed size, rather than per vertex, avoiding the issue of these expensive linear searches.

LLAMA achieves a throughput in the range of $225k \sim 330k$ edges/sec. Creating new levels every 10 seconds, the value proposed in [46], we were able to perform insertions for up to about $4 \sim 5$ hours before the library exhausted the available memory in the machine. As consequence, we were not able to process the graphs with SF 26. LLAMA seems to present optimal scalability and even hyper-scalability in graph500-24. This is a mere artifact as the actual throughput also depends on the number of delta levels present. With $p = 1$, more levels are created, and therefore it takes longer to perform each single insertion.

As in Stinger, the majority of the time in LLAMA is spent in a linear search to check if the edges inserted already exist. With $p = 1$, this operation accounts for 90% of the total time on graph500-24,

[5] Formally, the strong scalability [49] is the ratio T(1)/T(p), where T(p) is the completion time of the system with *parallelism degree* p. In this context, the parallelism degree is always the number of concurrent user threads.

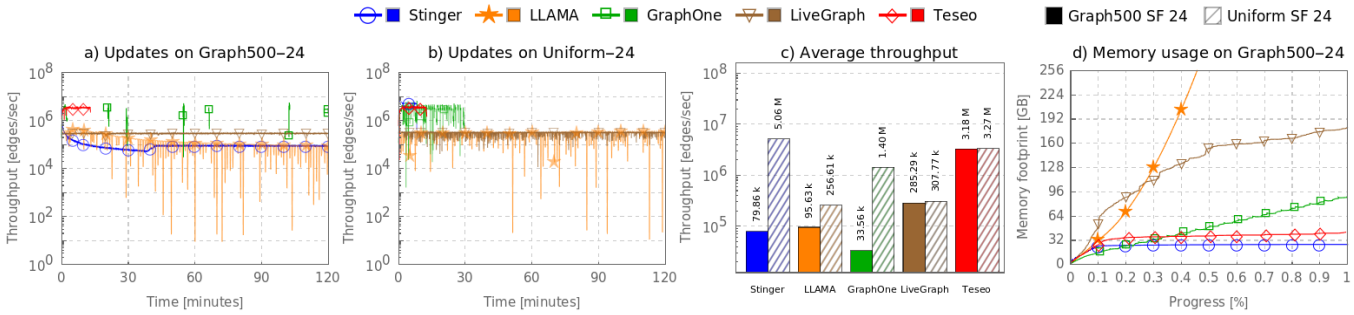

Figure 7: Plots a) and b) show the throughput measured each second, for the first two hours, of the workload with updates on graph500-24 and uniform-24. Plot c) reports the average throughput measured on both graphs (solid vs. dashed bars). Plot d) shows the memory footprint of the systems while the experiment on graph500-24 progressed.

and 64% for the uniform graph. In terms of scalability, LLAMA presents a number of critical points. As it acquires a latch at vertex granularity, edge insertions can be executed in parallel only as far as the vertices being referred do not overlap. Moreover, the creation of new delta levels is not thread safe, it must occur in mutual exclusion with the threads involved in the insertions. Finally, the creation of new vertices is serial, while the code base makes an extensive usage of latches. All these factors impair its scalability.

GraphOne achieves a throughput of up to $1.7M \sim 2.5M$ edges/sec, depending on the graph. But the semantics is different. By design, it also assumes that all provided updates are always correct and any consistency requirement is guaranteed externally, by other means. If we were to match the semantics of the other systems, that is, creating a new snapshot before an insertion, just to check whether an edge already exists, the throughput would only be 5 edges/sec, as the creation of snapshots is a heavy and rather expensive operation.

In terms of scalability, GraphOne achieved a value of $2.2x$ with $p = 6$ in graph500-24, showing only minor increments at higher parallelism degrees. The major limitation is the creation of new vertices, a procedure that it is not thread safe and must occur in mutual exclusion. When disabling the vertex dictionary and relabelling in advance the vertices of the input graphs into a dense domain, the throughput improved by $4x \sim 8x$ with $p = 1$, depending on the graph. However, already with $p = 2$, the throughput decreased by more than 20%, due to the additional contention in the write buffer.

LiveGraph executed up to $325k$ edges/sec $\approx 325k$ transactions/sec. The system was also able to scale up to $p = 8$ and, after that, it only showed meager improvements. We speculate these limitations are actually due to implementation details of the transaction manager [27], rather than design choices of the system.

7.3 Updates

In this experiment, we considered the impact of deletions in the overall throughput. The experiment proceeded as follows. We first inserted $|E|$ edges as in the original graphs. We then performed a mix of edge insertions and deletions, while still maintaining the total number of edges in the database close to $|E|$. The final number of updates carried in the experiment was roughly $10 \cdot |E|$. New edges were created following the same degree distribution of the original graphs. Figure 7 depicts the throughput measured each second and the final average throughput for both graph500-24 and uniform-24, when using the optimal parallelism degree.

Teseo achieved a throughput of $\approx 3.20M$ updates per second. This is 25% $\sim$ 30% higher than the scenario with only insertions. In general, updates tend to favour in-place data structures as deletions make room for new insertions. Specifically, in Teseo, less rebalances and leaf splits are triggered. The throughput per second is very stable and close to the average throughput, with little difference between the uniform and the skewed scenario. Finally, although not depicted for reasons of space, the scalability measured is similar to Figure 6a, and the throughput reported refers to $p = 40$.

On the other hand, deletions are detrimental for the overall performance of both LLAMA and GraphOne. They pay for the lack of an index to quickly locate the edges to replace. This situation is very pronounced in GraphOne for the experiment based on graph500-24. In Figure 7a, its throughput remains stable at the start of the experiment, the leftmost part of the plot, where only insertions take place. Once deletions appear, the buffer of the write store gets quickly filled. Creating new deltas becomes highly inefficient, as, in deletions, GraphOne needs to locate the previously inserted edges to set a *tombstone*. This scenario gives rise to short peaks where updates are performed, followed by long intervals devoted to the creation of new deltas, where the system becomes unresponsive.

Compared to the experiments with only insertions, in graph500-24, the final throughput is 44% for LLAMA and 2% for GraphOne. The latency of single updates is rather unstable, due to the time consumed in the creation of new deltas. In Figure 7, we reported a sample every 10 seconds in case of LLAMA, otherwise the plots would be completely filled by an orange rectangle. Lastly, both systems exhibited limited scalability, as most of the time was spent in the creation of the deltas.

Figure 7d shows the amount of physical memory (RSS) traced by `/proc/self/statm` on Linux. The plot should be interpreted in terms of peak memory, because the C++ run-time does not consistently release the freed memory back to the OS [18]. After the initial phase of insertions (10%), the memory footprint in Stinger is constant, all updates are in place. In Teseo and LiveGraph, the memory usage depends on the parallelism degree p. Teseo shows comparable footprint w.r.t. Stinger with $p = 1$, and it gradually increases up to $1.6x$ with $p = 40$, due to the extra pressure in the internal GC. Stinger and LLAMA do not feature a GC, while in GraphOne it is only partially implemented. None of these three systems shows any variation of memory usage relative to p. For reasons of

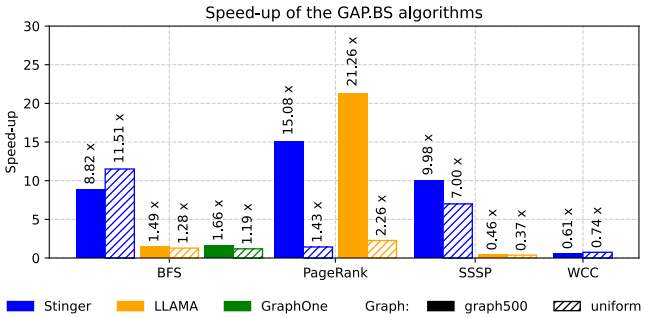

Figure 8: Speed-up of the graph algorithms from the GAP BS over the same algorithms provided by the library authors, for the graph500 and uniform graphs at SF 24.

space, we only report the memory footprint on graph500-24, but we note that we observed similar results on uniform-24.

7.4 Graphalytics

For the Graphalytics suite, we executed the same algorithm implementations on all systems evaluated. While some competitors ship a native implementation for some of the kernels in Graphalytics, when evaluating those, it becomes unclear whether improvements in the completion times are owed to more tuned implementations of the graph algorithms or, instead, to intrinsic merits of the systems themselves. On the other hand, as all dynamic systems share a similar API, any better algorithm implemented on top of one these should be straightforwardly portable to the others.

For the kernels BFS, PR, SSSP and WCC, we adapted the source code of the related algorithms from the *Graph Algorithm Platform Benchmark Suite* (GAP BS) [7]. For the kernels LCC and CDLP we derived a custom parallel implementation from the Graphalytics specification. For reference, in Figure 8 we compare the kernel implementations from GAP BS with the corresponding native implementations provided by the library authors, where available. Stinger includes its own implementation of the BFS, PageRank (PR), SSSP and WCC kernels, LLAMA of the BFS, PR and SSSP, while GraphOne only of the BFS kernel. In fact, GraphOne also ships with an implementation of the PageRank, but we deemed it too incompatible from both a computation and output standpoint with all the other implementations of the same algorithm. With a few exceptions, notably WCC for Stinger and SSSP for LLAMA, the implementations from the GAP BS generally improve the execution time of the kernels, in some cases even dramatically, achieving a speed-up of up to $15x$ for Stinger and $21x$ for LLAMA.

Table 3 reports the completion times measured for the kernels in the Graphalytics suite. All graphs were created as in Section 7.2, but the CSR, loaded in one pass. Figure 9 shows the completion time of a single scan of the whole graph. Systems based on a vertex table retrieve the edges as logical identifiers, the indices of the vertices in the vertex table, see Figure 1. This is convenient for the graph algorithms because, as noted in Section 6.2, they also expect the identifiers being part of a dense domain. In Teseo, scans retrieve the external (or real) vertex identifiers. There is an additional cost to translate them into logical identifiers. To analyse this cost, we report in Table 3 also the completion times for Teseo when evaluating the same algorithms without the logical identifiers, by relabelling in advance the graphs into a dense domain.

In Figure 9, Teseo ranks second, after the CSR baseline, in the raw scans when using the real vertex identifiers. In the sequential pattern, the difference with the CSR is $\approx$ 50%. This often translates to a divergence of up to 50% in Graphalytics, becoming more substantial in the kernels with more random accesses (LCC, SSSP). In Figure 9, scans with the logical identifiers incur an additional penalty of up to $7x$. In Graphalytics, with the logical identifiers, Teseo can still attain comparable completion times w.r.t. the other systems, but in SSSP and LCC. We attribute this behaviour both to the significant random accesses present in these kernels and to the relabelling, which also exhibits a random access per item and depletes more rapidly the amount of memory bandwidth available in the computation of the graph algorithms.

LiveGraph and Stinger rank third and fourth in terms of raw scans, respectively. Stinger stores the edges adjacent to a given vertex in a linked list of blocks of up to 14 edges. The blocks are relatively small and cause frequent memory jumps to iterate over them. LiveGraph avoids this issue by storing the edges in a per vertex vector, but it incurs an extra cost per edge due to the MVCC mechanism. Contrary to both CSR and Teseo, there is no significant diversity between the sequential and random patterns, as their layouts do not target the sequential pattern in the first place. Teseo further takes advantage of vertical partitioning [§4.4] and of the "fast code path" [§2.3] to skip the overhead of the MVCC mechanism.

GraphOne ranks fifth in terms of raw scans. It is the only system that does not use an iterator to retrieve the edges. Rather, a user provides a buffer to the system, where all edges adjacent to the vertex requested are copied. There are two main drawbacks with this approach. The first one is that it involves an additional copy, while with an iterator edges are immediately processed when retrieved from the storage. The second one is that, sometimes, an algorithm does not need to visit all edges of a vertex but it may request to terminate a scan prematurely due to some arisen condition, removing the necessity of copying all edges in advance. In particular, this can occur in the BFS and the WCC kernels, where GraphOne is significantly slower than the other systems.

In Figure 9, LLAMA exhibits a high variance in throughput, depending on both the graph and the access pattern. LLAMA favours locality inside a delta, edges adjacent to a given vertex are also packed together following a sorted order. On the other hand, when creating a delta level every 10 secs, edges end up in many different small deltas, breaking locality, while iterations over the outgoing edges of a vertex become alike traversing a linked list. This effect is reflected in the Graphalytics kernels, where LLAMA can outperform the other systems in some cases (BFS, WCC), but it can also be significantly slower (LCC, SSSP).

Finally, we note that an improved algorithm of the LCC kernel exists. The kernel aims to count the number of triangles passing on each vertex. We could devise a better implementation for the problem by noting that, once we have found a triangle among three vertices $v_1 - v_2 - v_3$, we already know that five other triangles exist by permuting these three vertices. This new implementation, referred in Table 3 as *LCC (opt)*, can be up to $30x$ faster than the standard LCC algorithm and allows to solve the kernel in all graphs well below the time limits. Teseo lags relatively behind the CSR as

Graph	System	BFS	CDLP	LCC	LCC (opt)	PageRank	SSSP	WCC	Graph	System	BFS	CDLP	LCC	LCC (opt)	PageRank	SSSP	WCC
DOTA League	CSR (baseline)	1.24*s*	2.72*s*	488*s*	70*s*	1.59*s*	1.73*s*	1.14*s*	Graph500 SF 22	CSR (baseline)	1.64*s*	13.17*s*	449*s*	19.49*s*	2.25*s*	2.99*s*	1.65*s*
	Stinger	1.10*x*	**0.74x**	3.20*x*	N/A	1.61*x*	1.50*x*	1.16*x*		Stinger	1.10*x*	**0.45x**	2.96*x*	N/A	1.60*x*	1.42*x*	1.37*x*
	LLAMA	1.00*x*	1.69*x*	DNF	N/A	1.08*x*	2.09*x*	1.03*x*		LLAMA	1.01*x*	0.83*x*	DNF	N/A	**1.08x**	2.14*x*	**0.99x**
	GraphOne	3.83*x*	2.34*x*	1.28*x*	N/A	3.19*x*	3.08*x*	5.52*x*		GraphOne	3.25*x*	0.92*x*	1.33*x*	N/A	2.87*x*	2.69*x*	9.26*x*
	LiveGraph	1.04*x*	1.09*x*	1.40*x*	N/A	1.26*x*	1.19*x*	1.06*x*		LiveGraph	1.27*x*	0.81*x*	1.45*x*	N/A	1.65*x*	1.41*x*	1.08*x*
	Teseo, log. vtx	1.00*x*	1.21*x*	1.55*x*	1.48*x*	1.18*x*	1.26*x*	1.07*x*		Teseo, log. vtx	1.03*x*	0.74*x*	2.18*x*	3.90*x*	1.62*x*	1.80*x*	1.49*x*
	Teseo, real vtx	**0.99x**	0.98*x*	**1.08x**	**1.28x**	**1.07x**	**1.04x**	**1.01x**		Teseo, real vtx	**1.01x**	0.49*x*	**1.18x**	**1.93x**	1.15*x*	**1.29x**	1.06*x*
Uniform SF 24	CSR (baseline)	2.40*s*	64*s*	34*s*	6.12*s*	6.60*s*	10.97*s*	3.63*s*	Graph500 SF 24	CSR (baseline)	2.51*s*	52*s*	3271*s*	108*s*	5.16*s*	9.47*s*	3.20*s*
	Stinger	1.13*x*	0.66*x*	2.23*x*	N/A	1.91*x*	**1.65x**	1.40*x*		Stinger	1.25*x*	0.51*x*	DNF	N/A	2.10*x*	**1.64x**	1.82*x*
	LLAMA	1.05*x*	1.40*x*	DNF	N/A	1.90*x*	6.65*x*	**1.21x**		LLAMA	1.09*x*	0.97*x*	DNF	N/A	**1.34x**	4.01*x*	**1.05x**
	GraphOne	2.98*x*	1.16*x*	3.04*x*	N/A	2.56*x*	2.67*x*	27.84*x*		GraphOne	2.93*x*	0.63*x*	DNF	N/A	2.65*x*	2.36*x*	21.92*x*
	LiveGraph	1.29*x*	0.64*x*	1.61*x*	N/A	2.28*x*	2.56*x*	1.33*x*		LiveGraph	1.77*x*	0.76*x*	DNF	N/A	2.34*x*	1.65*x*	1.25*x*
	Teseo, log. vtx	1.06*x*	0.99*x*	3.11*x*	3.52*x*	2.40*x*	2.85*x*	1.92*x*		Teseo, log. vtx	1.06*x*	0.82*x*	DNF	5.98*x*	2.54*x*	2.64*x*	2.21*x*
	Teseo, real vtx	**1.03x**	**0.61x**	**1.36x**	**1.94x**	**1.33x**	1.85*x*	1.21*x*		Teseo, real vtx	**1.02x**	**0.49x**	DNF	**2.02x**	1.42*x*	1.71*x*	1.19*x*
Uniform SF 26	CSR (baseline)	4.18*s*	175*s*	142*s*	22*s*	23*s*	44*s*	10.60*s*	Graph500 SF 26	CSR (baseline)	5.03*s*	110*s*	DNF	704*s*	19.20*s*	36*s*	9.66*s*
	Stinger	1.49*x*	1.12*x*	2.41*x*	N/A	2.10*x*	**1.77x**	1.58*x*		Stinger	1.18*x*	1.15*x*	DNF	N/A	2.33*x*	**1.97x**	2.08*x*
	GraphOne	3.68*x*	1.72*x*	3.41*x*	N/A	2.44*x*	2.80*x*	45.72*x*		GraphOne	2.64*x*	1.73*x*	DNF	N/A	2.55*x*	2.84*x*	36.61*x*
	LiveGraph	1.63*x*	1.62*x*	1.63*x*	N/A	2.62*x*	3.00*x*	1.54*x*		LiveGraph	2.76*x*	1.55*x*	DNF	N/A	2.73*x*	2.32*x*	1.32*x*
	Teseo, log. vtx	1.13*x*	1.62*x*	3.47*x*	4.24*x*	2.80*x*	3.26*x*	1.71*x*		Teseo, log. vtx	1.11*x*	1.80*x*	DNF	DNF	3.03*x*	3.16*x*	2.83*x*
	Teseo, real vtx	**1.04x**	**1.04x**	**1.40x**	**2.08x**	**1.43x**	2.05*x*	**1.30x**		Teseo, real vtx	**0.89x**	**1.05x**	DNF	**2.17x**	**1.52x**	2.27*x*	**1.21x**

Table 3: Results measured in the Graphalytics benchmark. For the CSR, we report the absolute completion time. For all the other systems, for ease of comparison, we show the speed-up of the CSR over the given system. The best result, outside the CSR, is marked in bold. The results refer to the median out of 15 runs.

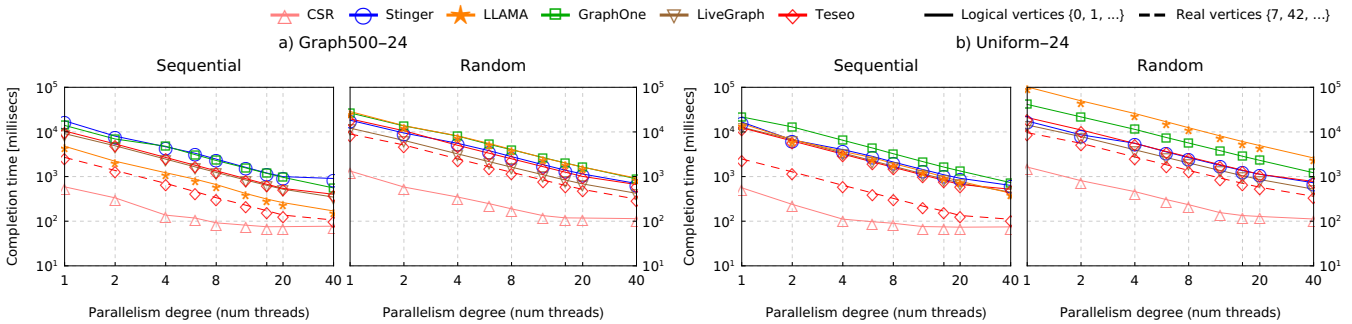

Figure 9: Completion time to perform a single scan in the whole graph in a) graph500-24 and b) uniform-24, depending on whether the vertices are selected sequentially and in order (v_0, v_1, v_2, ...) or randomly.

the impact of random accesses, in this variant, is more significant. Differently from the other kernels, we could not directly port this implementation to our contenders, because we exploited the sorted order of the edges, a property lacking in our competitors and fundamental in this implementation[6]. The sorted order is employed to prune in advance visiting adjacency lists and *sort-merge* them, to swiftly identify the neighbours shared between two given vertices.

8 REMARKS

Our results do not generally match those previously published from the authors of the compared systems. In the published papers, Stinger claims a throughput of 1.6M edges/sec [20], LLAMA 5.6M edges/sec [46], whereas GraphOne $40-50M$ edges/sec [35, 37]. Our measurements for the insertions on graph500-24 place Stinger at 80k edges/sec, LLAMA at 260k edges/sec, and GraphOne at 2.56M edges/sec, all at 5% w.r.t. above results. As noted in the introduction, our experiments are based on fine-grained updates and rather complement the results of earlier works. In the graph algorithms, our results also present a more levelled ground than what depicted in [37], where Stinger, LLAMA and GraphOne have been compared and differences of more than 10x reported. In the following, we scrutinise and address our discrepancies w.r.t. these papers.

8.1 Updates

The value published for Stinger refers to a batch mode, where edges are loaded using contiguous constant batches of 100k edges. On single updates, reference [20] also reports a value of 12k edges/sec, on a graph comparable to graph500-24[7], thus more in line with our measurements. Stinger claims a speed-up of the batch mode of 133x, but as noted before, this starts from a weak baseline, outperformed by a sequential implementation using the C++ STL containers. On the other hand, in the current public code base, the batch mode of Stinger does not support edge deletions anymore.

LLAMA goes further, as the throughput of 5.6M edges/sec refers to loading from scratch a snapshot of the Twitter graph in one pass. Their paper does not provide measurements for single updates. Still, the major drawback of LLAMA for a real-time workload remains

[6] The LCC results in Table 2 are compared against this variant, since the implementation in the driver of SuiteSparse also exploits the sorted order.

[7] Although the two graphs have the same number of edges, Stinger is evaluated on a graph with twice the number of vertices than graph500-24. Both graphs are also created by an RMAT generator, but with diverse parameters and a different characterisation of the power law distribution.

the large memory footprint caused by delta levels, providing only up to four hours of usage in our configuration. Compaction can only reclaim deltas from the most recent to the oldest, rather than the opposite, impairing known strategies based on *low watermarks* [57]. In practice, the authors suggest merging all deltas together and rebuilding the whole database from time to time.

The results reported by GraphOne are obtained with a sequential driver, with fully static vertices already inserted, vertex identifiers represented as 32-bit values from a dense domain, weightless graphs and no support for deletions (a compiler switch). Even with these settings and after explicitly subtracting the overhead of our driver, we were not able to reproduce their peak values, reaching a maximum throughput of $\approx 20M$ insertions/sec. We note that the public code base [36] of GraphOne matured from the time the related papers were published, adding new missing features and introducing a series of fundamental bug fixes, that impacted its performance.

8.2 Graph algorithms

Experiments in LLAMA [46] consider only the scenario when the graph is statically loaded or 10 deltas are present, an arbitrary choice. In our experiments, we create a delta every 10 secs because our target is a real-time workload. In LLAMA, readers cannot access the write store and are bound to the time when the last delta was created. With SF 24, our experiments produce ≈ 80 delta levels for graph500 and ≈ 90 for the uniform graph. We measured a difference of up to $9x$ for a complete scan of the graphs between the scenarios with no deltas and when the deltas were created as in our setting.

In our results for Graphalytics, LLAMA also does not perform well in the LCC kernel. In [46], LLAMA outperforms all the compared systems in the similar problem of global triangle counting. But there is a caveat, their algorithm only applies when there are no deltas, although the reason is not disclosed in the paper. By inspecting the released source code [45], its implementation turns out that it uses the sort-merge to find common neighbours. Therefore, as our second implementation of LCC, it exploits the sorted order, which, in LLAMA, only exists when there are no deltas.

GraphOne [37] reports significant performance superiority, w.r.t. both Stinger and LLAMA, in the BFS and PageRank algorithms. These results are not confirmed in our experiments. We believe there are two major causes for that. The first one is merely algorithmic. As noted in Figure 8, when comparing different implementations for the same kernel, we achieved substantial speed-ups just because the algorithms used were simply better. The second reason is that GraphOne ships their algorithms on a custom abstraction model. The idea is the following. For the read store, the user accesses the graph using the same model of the adjacency lists. For the write store, instead, the user accesses directly the write buffer and compensates for the single updates one by one.

For instance, consider the algorithm for a BFS. Each iteration scans the read store normally as the other systems. For the write store, the algorithm checks whether each edge present contains as source a vertex in the frontier, and in this case, it adds its destination to be processed at the next step. This works correctly as far as all updates are insertions. The case with deletions becomes significantly more complex, because the algorithm needs to check whether the destination of a removed edge was reached at the current iteration and can still be part of the vertices being processed at the next step. Currently, GraphOne ships a version of the BFS assuming that all edges in the write buffer are only insertions.

In general, we believe that exposing a custom abstraction model and demanding multiple implementations, depending on the scenario, of the same algorithms pose a high burden to end users. For these reasons and for fairness, in our experiments, we strove to compare all systems under the *same programming model*, *same algorithm implementation* and *same output*.

9 RELATED WORK

Sparse arrays were invented by [32]. The idea of asynchronously delaying rebalances in sparse arrays was firstly introduced in [16] as a mechanism to fight contention.

Several techniques applied in Teseo derive from previous work in the domain of RDBMSes. Garbage collectors (GC) are customary in MVCC systems [13]. Our GC implements an algorithm analogous to Steam [13]. In particular, it can remove all inaccessible versions even in presence of long running transactions. Fence keys are a standard technique of *B*-trees [23], although employed for different purposes. Optimistic latches are a generalization of *Optimistic Lock Coupling* [42]. Teseo employs a trie as primary index for efficiency reasons. Reference [59] presents an experimental evaluation of different existing indices, showing the benefits of tries over the standard separator keys of B^+ trees. Read-only transactions are a well-known optimisation in RDBMSes [56].

While an extensive collection of systems for the analysis of static graphs exists [62], little attention has been dedicated to dynamic graphs. LDBC Graphalytics [29] and the GAP BS [7] are arguably the two most referred benchmarks for structural static graphs. The two standards share 2/3 of the graph algorithms. We are not aware of the existence of any standard analytical benchmark for dynamic structural graphs. The LDBC SNB Interactive [21, 38] is another prominent transactional benchmark with insertions, it targets an *online* workload on *property graphs* [5]. At the time of writing, the LDBC consortium is pursuing an ongoing effort [60] for a hybrid workload on property graphs, with the inclusion of general updates.

10 CONCLUSIONS

We presented Teseo, a library for the real-time analysis of dynamic graphs. Teseo is optimised for a hybrid workload with fine-grained updates and long running analytics. It is based on a novel variant of the B^+ tree, the fat tree. By leveraging the sorted order, leaves with a large size and additional indexing, Teseo strives to minimise the memory accesses typical of graph analysis, while remaining capable of sustaining high rates of updates. Moreover, Teseo can translate the vertex identifiers into a logical domain, easing the adoption of existing algorithms originally developed for static graphs.

Our approach contrasts with the design adopted by other prominent systems, based on a vertex table. This design can be fragile towards general updates and does not properly capture the access patterns common in graph analysis. Although competing systems can somewhat outperform Teseo, they do at the expense of a lack or reduced consistency and/or flexibility, a coarser granularity, and very limited, when not absent, support for dynamic vertices, becoming more specialised tools. On the other hand, in addition to the above, Teseo brings to the table full support for transactions.

REFERENCES

[1] 2021. Teseo public repository. https://github.com/cwida/teseo

[2] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems.* Now Publishers Inc., Hanover, MA, USA.

[3] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (Sept. 1988), 1116–1127. https://doi.org/10.1145/48529.48535

[4] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph Based Anomaly Detection and Description: A Survey. *Data Min. Knowl. Discov.* 29, 3 (May 2015), 626–688. https://doi.org/10.1007/s10618-014-0365-y

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages. https://doi.org/10.1145/3104031

[6] Mohsen Aznaveh, Jinhao Chen, Timothy A. Davis, Bálint Hegyi, Scott P. Kolodziej, Timothy G. Mattson, and Gábor Szárnyas. 2020. Parallel GraphBLAS with OpenMP. In *SIAM Workshop on Combinatorial Scientific Computing (CSC) co-located with the SIAM Conference on Parallel Processing for Scientific Computing (PP)*. SIAM.

[7] Scott Beamer, David Patterson, and Krste Asanović. 2019. Reference implementation of the GAP Benchmark Suite. https://github.com/sbeamer/gapbs

[8] Michael Bender, Martin Farach-Colton, and Miguel Mosteiro. 2006. Insertion Sort is O(N log N). *Theor. Comp. Sys.* 39, 3 (2006), 391–397.

[9] M. A. Bender, E. D. Demaine, and M. Farach-Colton. 2000. Cache-Oblivious B-Trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS '00)*. IEEE Computer Society, USA, 399.

[10] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. 2005. Cache-Oblivious B-Trees. *SIAM J. Comput.* 35, 2 (Aug. 2005), 341–358. https://doi.org/10.1137/S0097539701389956

[11] Michael A. Bender and Haodong Hu. 2007. An Adaptive Packed-Memory Array. *ACM Trans. Database Syst.* 32, 4 (Nov. 2007), 26–es. https://doi.org/10.1145/1292609.1292616

[12] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and Robust Latches for Database Systems. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/3399666.3399908

[13] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 128–141. https://doi.org/10.14778/3364324.3364328

[14] Thomas Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). MIT Press.

[15] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (Dec. 2019), 25 pages. https://doi.org/10.1145/3322125

[16] Dean De Leo and Peter 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

[17] Dean De Leo and Peter Boncz. 2019. Packed Memory Arrays – Rewired. In *Proceedings of the 2019 IEEE International Conference on Data Engineering (ICDE 2019) (ICDE '19)*. IEEE Computer Society, Washington, DC, USA, 830–841.

[18] GLIBC developers. 2019. The community wiki for GLIBC: Malloc internals. https://sourceware.org/glibc/wiki/MallocInternals

[19] Ulrich Drepper. 2007. *What Every Programmer Should Know About Memory*. Technical Report. Red Hat, Inc. https://lwn.net/Articles/250967

[20] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[21] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter 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) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2723372.2742786

[22] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper20.pdf

[23] Goetz Graefe. 2011. Modern B-Tree Techniques. 3, 4 (April 2011), 203–402. https://doi.org/10.1561/1900000028

[24] Adrien Guille, Hakim Hacid, Cecile Favre, and Djamel A. Zighed. 2013. Information Diffusion in Online Social Networks: A Survey. *SIGMOD Rec.* 42, 2 (July 2013), 17–28. https://doi.org/10.1145/2503792.2503797

[25] Gundolf Haase, Manfred Liebmann, and Gernot Plank. 2007. A Hilbert-order Multiplication Scheme for Unstructured Sparse Matrices. *Int. J. Parallel Emerg. Distrib. Syst.* 22, 4 (Jan. 2007), 213–220. https://doi.org/10.1080/17445760601122084

[26] Bálint Hegyi and Gábor Szárnyas. 2020. SuiteSparse:GraphBLAS driver for LDBC Graphalytics. https://github.com/ftsrg/ldbc-graphalytics-platforms-graphblas

[27] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. 13, 5 (Jan. 2020), 629–642. https://doi.org/10.14778/3377369.3377373

[28] Intel corporation. [n.d.]. Intel Threading Building Blocks (TBB). https://software.intel.com/en-us/tbb

[29] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[30] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2017. LDBC Graphalytics Benchmark specification, v0.9.0. http://github.com/ldbc/ldbc_graphalytics_docs

[31] Alon Itai and Irit Katriel. 2007. Canonical Density Control. *Inf. Process. Lett.* 104, 6 (2007), 200–204.

[32] Alon Itai, Alan G. Konheim, and Michael Rodeh. 1981. A Sparse Table Implementation of Priority Queues. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*. Springer-Verlag, Berlin, Heidelberg, 417–431.

[33] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. 2015. Graph analytics using vertica relational database. In *2015 IEEE International Conference on Big Data (Big Data)*. 1191–1200.

[34] Alexander D. Kent, Lorie M. Liebrock, and Joshua C. Neil. 2015. Authentication graphs: Analyzing user behavior within an enterprise network. *Comput. Secur.* 48, C (Feb. 2015), 150–166. https://doi.org/10.1016/j.cose.2014.09.001

[35] 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*. 249–263. https://www.usenix.org/conference/fast19/presentation/kumar

[36] Pradeep Kumar and H. Howie Huang. 2019. GraphOne public repository. https://github.com/the-data-lab/GraphOne

[37] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4, Article 29 (Jan. 2020), 40 pages. https://doi.org/10.1145/3364180

[38] Linked Data Benchmark Council (LDBC). 2015. Social Network Benchmark (SNB). http://ldbcouncil.org/benchmarks/snb

[39] LDBC Graphalytics Team. 2018. Graphalytics Spring Competition #1. https://graphalytics.org/competition

[40] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42 (2019), 73–84.

[41] V. Leis, A. Kemper, and T. Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49.

[42] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2933349.2933352

[43] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1, Article 1 (July 2016), 20 pages. https://doi.org/10.1145/2898361

[44] A. Lumsdaine, Douglas P. Gregor, B. Hendrickson, and Jonathan W. Berry. 2007. Challenges in Parallel Graph Processing. *Parallel Process. Lett.* 17 (2007), 5–20.

[45] Peter Macko. 2015. LLAMA public repository. https://github.com/goatdb/llama

[46] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering*. 363–374. https://doi.org/10.1109/ICDE.2015.7113298

[47] Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General(?)! *ACM Trans. Parallel Comput.* 5, 4, Article 16 (Feb. 2019), 32 pages. https://doi.org/10.1145/3309206

[48] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. 2014. A Performance Evaluation of Open Source Graph Databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications* (Orlando, Florida, USA) *(PPAA '14)*. Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/2567634.2567638

[49] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[50] Frank McSherry. 2017. COST in the land of databases. https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md

[51] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at What Cost?. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland) *(HOTOS'15)*. USENIX Association, USA, 14.

[52] 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) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 677–689. https://doi.org/10.1145/2723372.2749436

[53] Wing Lung Ngai, Stijn Heldens, Mihai Capotă, Ahmed Musaafir, Tim Hegeman, Narayanan Sundaram, and Michael J. Anderson. 2017. Graphalytics GraphMat platform driver. https://github.com/atlarge-research/graphalytics-platforms-graphmat

[54] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* 76 (Sep 2007), 036106. Issue 3. https://doi.org/10.1103/PhysRevE.76.036106

[55] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB Journal* 29, 2-3 (2020), 595–618.

[56] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. 2010. *Database System Concepts* (6 ed.). McGraw-Hill Higher Education.

[57] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564.

[58] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[59] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, New York, NY, USA, 473–488. https://doi.org/10.1145/3183713.3196895

[60] Jack Waudby, Benjamin A. Steer, Arnau Prat-Pérez, and Gábor Szárnyas. 2020. Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator. In *GRADES-NDA at SIGMOD/PODS*. ACM, ACM.

[61] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. 2013. Graph Analysis: Do We Have to Reinvent the Wheel?. In *First International Workshop on Graph Data Management Experiences and Systems* (New York, New York) *(GRADES '13)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2484425.2484432

[62] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Found. Trends databases* 7, 1-2 (Jan. 2017), 1–195. https://doi.org/10.1561/1900000056

[63] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (March 2020), 1020–1034. https://doi.org/10.14778/3384345.3384351

[64] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph, binary release. https://github.com/thu-pacman/LiveGraph-Binary