CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure

Soukaina Firmli
Mohammed V University in Rabat, Ecole Mohammadia d'Ingénieurs, SIP Research Team, Morocco
Oracle Labs, Casablanca, Morocco
soukaina.firmli@oracle.com

Vasileios Trigonakis
Oracle Labs, Zürich, Switzerland
vasileios.trigonakis@oracle.com

Jean-Pierre Lozi
Oracle Labs, Zürich, Switzerland
jean-pierre.lozi@oracle.com

Iraklis Psaroudakis
Oracle Labs, Zürich, Switzerland
iraklis.psaroudakis@oracle.com

Alexander Weld
Oracle Labs, Zürich, Switzerland
alexander.weld@oracle.com

Dalila Chiadmi
Mohammed V University in Rabat, Ecole Mohammadia d'Ingénieurs, SIP Research Team, Morocco
chiadmi@emi.ac.ma

Sungpack Hong
Oracle Labs, Palo Alto, CA, USA
sungpack.hong@oracle.com

Hassan Chafi
Oracle Labs, Palo Alto, CA, USA
hassan.chafi@oracle.com

Abstract

The graph model enables a broad range of analysis, thus graph processing is an invaluable tool in data analytics. At the heart of every graph-processing system lies a concurrent graph data structure storing the graph. Such a data structure needs to be highly efficient for both graph algorithms and queries. Due to the continuous evolution, the sparsity, and the scale-free nature of real-world graphs, graph-processing systems face the challenge of providing an appropriate graph data structure that enables both fast analytical workloads and low-memory graph mutations. Existing graph structures offer a hard trade-off between read-only performance, update friendliness, and memory consumption upon updates. In this paper, we introduce CSR++, a new graph data structure that removes these trade-offs and enables both fast read-only analytics and quick and memory-friendly mutations. CSR++ combines ideas from CSR, the fastest read-only data structure, and adjacency lists to achieve the best of both worlds. We compare CSR++ to CSR, adjacency lists from the Boost Graph Library, and LLAMA, a state-of-the-art update-friendly graph structure. In our evaluation, which is based on popular graph-processing algorithms executed over real-world graphs, we show that CSR++ remains close to CSR in read-only concurrent performance (within 10% on average), while significantly outperforming CSR (by an order of magnitude) and LLAMA (by almost 2×) with frequent updates.

2012 ACM Subject Classification Information systems → Data structures; Theory of computation → Concurrency; Theory of computation → Graph algorithms analysis; Computing methodologies → Concurrent algorithms

Keywords and phrases Data Structures, Concurrency, Graph Processing, Graph Mutations

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.17

1 Introduction

Graph processing is an invaluable tool for data analytics, as illustrated by the plethora of relatively recent work aiming at achieving high performance for graph algorithms [12, 17, 23, 34, 35, 41], such as PageRank [29], or graph querying/mining [13, 21, 26, 27, 31, 33, 38], e.g., using PGQL [5]. At the heart of each graph system lies the graph data structure, responsible for holding the vertices and the edges comprising the graph, and whose performance largely contributes to the general performance of the system. The ideal graph structure should offer excellent read-only performance, fast mutations (i.e., vertex or edge insertions and deletions), and low memory consumption with or without mutations.

Classic graph data structures typically trade some characteristics for others (see Section 2.1). Adjacency lists enable quick graph updates and consume relatively little memory, but sacrifice performance, as they lead to expensive pointer chasing. Adjacency matrices enable quick edge updates, but sacrifice vertex insertions and consume a lot of memory. Finally, the Compressed Sparse Row (CSR) representation offers a good memory footprint with excellent read-only performance by completely sacrificing mutability: even a single vertex or edge insertion requires complete reallocation of the underlying structures. There have been efforts to improve the update-friendliness of CSR (see Section 2.2). These include in-place update techniques [36, 39], batching techniques [11, 24], and changeset-based updates with delta maps [22] and multi-versioning [23]. A multi-versioning solution is used in LLAMA [23], a state-of-the-art update-friendly graph structure that enables mutability on top of CSR by appending delta snapshots. However, having a frequent flow of graph updates – which is the common case in real-life scenarios, such as financial transactions – results in a large number of delta logs, and thus high memory utilization and decreased performance. Compaction operations on these data structures are often expensive, hindering the benefits of fast mutability (see Section 4 for a performance analysis). Very often, users simply need to operate on the most up-to-date version of the graph data, thus asking for fast in-place graph updates.

In this paper, we introduce CSR++, a concurrent graph data structure with performance comparable to CSR, efficient updates, and memory consumption proportional to the number of mutations. CSR++ maintains the array-continuity that makes CSR very fast. In particular, vertices are stored in arrays, segmented for better update-friendliness. As in CSR, vertex IDs in CSR++ are implicitly determined by the location of the vertex, but include both the segment ID and where in the segment the vertex lies. Accordingly, the 64 bits of vertex IDs are split into `{int segment_id; int in_segment_id}`, making vertices directly addressable. Due to segmentation, inserting a new vertex is as simple as (i) if needed, appending a new segment to the array of segments, and (ii) appending the vertex to that segment.

In contrast to CSR, and like adjacency lists, CSR++ can independently manage the edges of each vertex. If a vertex has two or more edges, CSR++ holds a pointer to an array storing the edges. To reduce memory usage, for single-edge vertices, the target vertex of the edge is inlined in lieu of the array pointer. All in all, CSR++ maintains the array-oriented structures of CSR for performance, while enabling per-vertex edge-list modifications to enable fast updates as with adjacency lists.

Apart from vertices and edges, graph structures also need to store vertex and edge properties, which are a prominent feature of property graphs. CSR++ includes segmentation techniques to enable fast property updates when new vertices or edges are inserted. Vertex properties are stored in segmented arrays, and each vertex holds a pointer to an array of edge property values, allowing for fast per-segment or per-vertex reallocation of property arrays.

We evaluate CSR++ with both read and update workloads, with various graphs and graph algorithms, and compare it against CSR, adjacency lists, and LLAMA. Our results indicate that CSR++ is much faster than adjacency lists, is almost as fast as CSR on read-only workloads, and has faster updates and lower memory consumption than LLAMA. In particular, CSR++ performs on average within 10% of the read-only performance of CSR with 36 threads and is an order of magnitude faster for updates. Furthermore, CSR++ is faster than LLAMA for most read-only workloads, is almost 2× faster in applying batched updates, and consumes 4× less memory when 100 update batches are applied on a base graph.

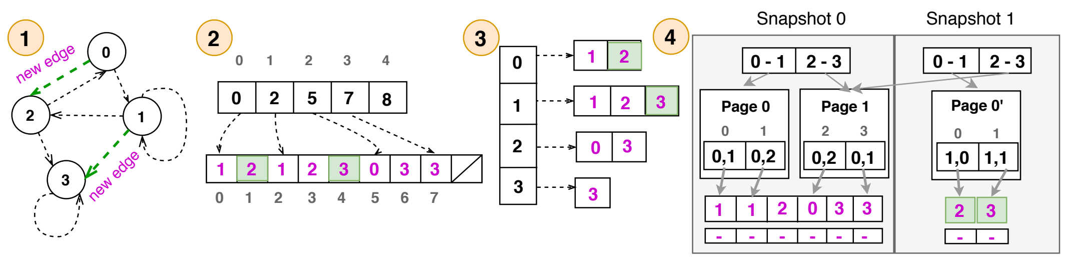

Figure 1 (1) An example graph with newly inserted edges in green, represented in different graph structures: (2) CSR, (3) Adjacency list, and (4) LLAMA with implicit linking and deletion vectors.

The main contributions of this paper are as follows:

- CSR++, a new graph data structure that supports fast in-place updates, without sacrificing read-only performance or memory consumption; and
- Our thorough evaluation that shows that CSR++ achieves the best of both read-only and update-friendly worlds.

2 Background & Related Work

Graphs are already a prominent data model, especially in the current era of big data and data deluge [16]. The advantage over the traditional relational model is that graphs can inherently model entities and their relationships. While a relational model needs to join tabular data in order to process foreign-key relationships, graph-processing engines have built-in ways to efficiently iterate over the graph [37], e.g., over the neighbors of vertices, and support a plethora of expressive graph algorithms (such as Green-Marl [19, 34]) and graph pattern-matching queries (such as PGQL [5], SPARQL [8], and Gremlin [9]).

Graphs can be represented with different models and data representations. A popular model is the RDF (Resource Description Framework) graph data model [8], which became popular with the rise of the semantic web [10]. RDF regularizes the graph representation as a set of triples. RDF adds links for all data, including constant literals, and it does not explicitly store vertices, edges, or properties separately. As the graph is not stored in its native format, it results in reduced performance [40], as RDF engines are forced to process and join a large number of intermediate results.

Our paper focuses on a more recent model, the Property Graph (PG) model [6, 38], which is widely adopted by various graph databases and processing systems (such as Neo4J [27] and PGX [28, 31]). PG represents the topology of a graph natively as vertices and edges, and stores properties separately in the form of key-value pairs. This separation allows for quick traversals over the graph structure. Classic graph algorithms, such as PageRank [29] and Connected Components, are very naturally expressed on top of property graphs [34].

In order for graph-processing engines to provide efficient solutions for large-scale graphs, they rely on efficient data structures, potentially resident on main memory [23, 41, 18, 15], to store and process vertices and their relationships. One of the key challenges for in-memory graph-processing engines is to design data structures with reasonable memory footprint [23] that can support fast graph algorithm execution [19] and query pattern matching [32], whilst supporting topological modifications (like additions or removals of vertices and edges), either in batches or in a streaming fashion [25, 11]. In the following, we discuss the most prominent data structures in related work [15], and motivate the necessity of the novel CSR++. We show in Figure 1 an example of a graph and how it is represented in different formats.

2.1 Graph Representations

Adjacency Matrices and Lists. An adjacency matrix represents a graph with a V^2 matrix M, where V is the number of vertices in the graph. A non-zero cell $M[v_s][v_d]$ represents the directed edge from a source vertex v_s to a destination vertex v_d. An adjacency matrix is not preferred for sparse graphs, i.e., graphs where the number of edges $E \ll V^2$, due to increased memory footprint and decreased performance in analytics.

Adjacency lists represent the graph with a set of vertices, where each vertex is associated with a list of neighbors, as shown in Figure 1(3). An adjacency list typically consumes less memory than an adjacency matrix, since for a given vertex only the existing edges need to be stored. The typical format for the adjacency list uses linked lists, with extra pointers, but more cache-friendly variants exist, such as Blocked Adjacency Lists, where adjacencies are represented by simple arrays [41] or with linked lists of buckets containing a fixed size of edges [14, 17]. As an example, the popular Boost C++ Library [1] implements adjacency lists and the edge structures can be configured to either be vectors, lists, or sets. Although adjacency lists can be efficient in terms of mutations, they struggle in read-only workloads, as we show in Section 4.

Compressed Sparse Row (CSR). CSR [19] is a commonly used data structure for sparse graphs, because it compacts adjacencies into two arrays: The vertex array and the edge array. In the vertex array, each vertex is identified by its array index. The vertex cell stores the begin offset in the edge array (the end offset is implicit, as it is equal to the begin offset of the next vertex cell), where the list of the destination neighbors of this vertex is stored, as shown in Figure 1(2). In terms of graph mutations, CSR is very inefficient. For example, to add an edge, the whole edge array needs to be reallocated with the newly-added edge and the subsequent edges shifted by one place.

2.2 Graph Mutations

Graph mutations, or updates, mostly refer to vertex or edge insertions and deletions. Although CSR is one of the most popular data structures for representing a graph, it is, as mentioned above, very limiting for graph mutations. This has prompted a lot of related work on mutable data structures to represent graphs that can efficiently digest sets of updates.

In-Place Updates. Techniques that use in-place updates employ the aforementioned static data structures in a way that allows for in-place digestion of sets with insertions and deletions of vertices and edges, without requiring the expensive rebuild of the data structure. For instance, Dense [20] is a concurrent graph adjacency matrix that supports mutations and partial traversals through a coordination protocol, but does not handle graph properties. NetworKit [36], in order to perform edge insertions, stores adjacencies vectors that double the size of the initial array to reserve enough space for new incoming edges. Madduri et al. [24] use the same underlying technique but define a configurable size of the new edge array instead of using factor 2. Ediger et al. [14] implement blocked adjacency lists and allow insertions by appending new blocks and updating pointers. Wheatman et al. [39] implement a variant of CSR that leaves space at the end of each adjacency list to allow efficient single-threaded mutations. We employ similar techniques in CSR++ to ingest mutations, but in a parallel manner while also handling graph property mutations.

Batching. Regarding the sources of changes, they can be continuous streams of updates [11, 14] or single changes applied as "local" mutations. Generally, when applying a batch of updates, frameworks perform pre-processing to re-arrange the batches in ways that can speed-up the mutations. For instance, Madduri et al. [24] apply techniques on the list of new edges, such as sorting, re-ordering, and partitioning, in order to exploit parallelism at the time of the changes application. Similarly, CSR++ groups updates by their source vertices, and uses multiple threads to perform fast edge insertions (see Section 3.2).

Multi-Versioning & Deltas. One way to extend CSR to support fast updates is by allocating a separate structure to store only the new changes [22] in delta maps. Furthermore, by using deltas, the following systems can run analytical workloads on different static versions (snapshots) of the changing graph over time. LLAMA [23] is a state-of-the-art snapshot-based graph system that supports multi-versioning by storing deltas as separate snapshots and supports concurrent access to those snapshots (see Figure 1(4)). ASGraph [17] limits its read-access to one snapshot at a time but still ensures high performance by extending its underlying data structure [14] with temporal attributes. Graphite [30] is an in-memory relational column-store that employs also multi-versioning snapshots using deltas.

The downside of the above approaches is two-fold. First, maintaining separate snapshots increases the memory requirements of the system, as a frequent flow of graph updates results in a large number of deltas. Second, the performance of analytics is degraded because they need to read from both the original structure and the deltas and reconcile them. A solution to the potential performance degradation is to periodically merge the delta maps into CSR, an operation called compaction. Compaction, however, can become very expensive, often zeroing the mutability performance benefits of these structures. For users that wish to operate on the most up-to-date version of the graph data, we show that CSR++, which is designed for in-place graph mutations, achieves better analytics and update performance than LLAMA [23], with up to an order of magnitude lower memory requirements (see Section 4).

3 CSR++: Design and Implementation

With CSR++, our goal is to design a data structure that stores graphs and allows fast in-place mutations with analytics performance comparable to CSR. In order to allow for fast algorithms, CSR++ enables fast concurrent accesses to the main graph data (vertex and edge tables) and stores additional graph data, such as reverse edges, user-defined keys, and vertex and edge properties. CSR++ does not aim to support versioning, but instead fast in-place updates, allowing to withstand frequent small updates without the overhead of snapshots.

3.1 Graph Topology and Properties

CSR++ is a concurrent structure that stores the graph in memory using segmentation techniques. It allows in-place insertions by allocating additional space for new incoming edges and supports logical deletions of vertices and edges. Figure 2 shows the building blocks of CSR++.

Segments. CSR++ stores vertices in arrays called *segments*. The graph is represented as an array of segments, each storing a fixed number of vertices defined by a global configurable parameter `NUM_V_SEG`. Segments give flexibility to CSR++ in three ways: (i) memory allocations and reallocations use segment granularity, (ii) vertex properties are allocated per segment, and (iii) synchronization for concurrency uses segment granularity. As with CSR, CSR++ packs the vertices in arrays to reduce the memory footprint when storing sparse graphs, which also results in better cache locality. The entry point to CSR++ is an array that stores all segments; this also enables quick segment additions. Finally, each segment stores a vector of pointers to the vertex property arrays.

Vertices. Each vertex stores its degree, a pointer to its list of neighbors, and optionally a pointer to the property values of its edges. This design resembles a mix of CSR and adjacency lists, however, adding a new vertex in CSR++ is faster (see Section 3.2) considering that the vertex array is segmented, i.e., we do not need to copy the whole vertex array to add or remove entries. CSR++ does not store explicit IDs for vertices nor edges, but since all segments store a fixed number `NUM_V_SEG` of vertices, we can compute implicit IDs for vertices using the segment ID and the index of the vertex in the segment: `global_v_id = (seg_id * NUM_V_SEG) + v_id` . Overall, the vertex structure consists of the following fields:

- `length (4 bytes)`: The vertex degree. A length of -1 indicates a deleted vertex.
- `neighbors (8 bytes)`: A pointer to the set of neighbors. As a space optimization, if `length = 1`, this field directly contains the neighbor's vertex ID.
- `edge_properties (8 bytes)`: A pointer to the set of edge properties. As a space optimization, this field can be disabled in case the graph does not define edge properties.

Edges. CSR++ represents the neighbors list of a vertex by an array of edges, where every entry stores the coordinates (i.e., the vertex ID and the segment ID) of the corresponding neighbor. At loading time, the edges are sorted; as with CSR and LLAMA, keeping the edges sorted allows for better cache performance. Moreover, this semi-sorting is necessary for CSR++ in a deletion-frequent context, as we use binary search to locate edges. Additionally, as an optimization for update-friendliness, CSR++ can be configured to create extra empty space for new incoming edges during graph loading (see Section 3.2). The edge structure consists of the following fields:

- `deleted_flag (2 bytes)`: For logical deletion of edges.
- `vertex_id (2 bytes)`: The index of the neighbor in the segment; using 16 bits allows for segments with a capacity `NUM_V_SEG` of up to 65536 entries.
- `segment_id (4 bytes)`: The segment ID where the neighbor is stored.

For better cache utilization when scanning over vertices and better load balancing when using multiple threads, the number of vertices that a segment stores should neither be very small nor too large, in order to avoid copying large amounts of data when the graph is updated. By default we use `NUM_V_SEG = 4096` vertices per segment.

Properties. Vertex property values are stored in arrays parallel to the vertices array. CSR++ keeps a vector of pointers to each vertex property array within the segment. The size of each array is therefore `NUM_V_SEG * sizeof(Property_Type)`. For edge properties, we use the

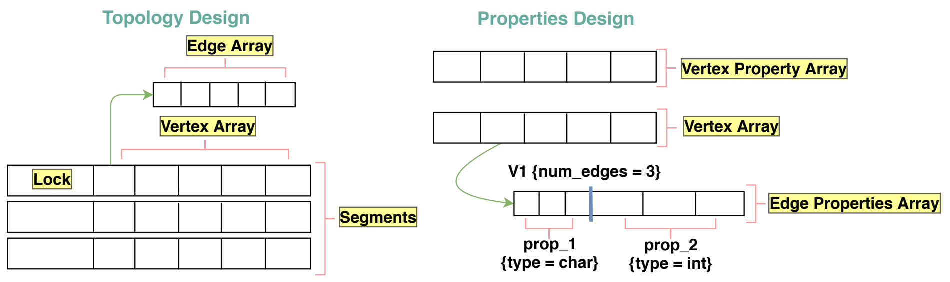

Figure 2 The building blocks of CSR++: Graph topology (left) and graph properties (right).

same segmentation approach as vertices. If the user enables edge properties, each vertex structure stores a pointer to an array of edge property values, as shown in Figure 2. In case of multiple properties, we allocate an array that stores the values for different edge properties in an cache-aligned manner. In order to locate a specific edge property p, we use offsets and the position of its values can be calculated given the type of that property T_p, the index i of the edge in the neighbor list, and the degree d of the vertex v. For example, suppose the user registers n edge properties, then the total size of the edge properties of a vertex v is $\sum_{p=1}^{n}(\mathit{sizeof}(\mathit{Type}_p) * d)$. Similarly, the values of the xth property begin at $\mathit{Values}(x) = \sum_{p=1}^{x}(\mathit{sizeof}(\mathit{Type}_p) * d)$. Accordingly, the property value for the xth property of the edge i is $\mathit{Value}(x, i) = \mathit{Values}(x) + (i * \mathit{sizeof}(\mathit{Type}_x))$.

The reason for this choice is that having the edge properties stored in parallel to the edge arrays allows to copy-on-write the edge property arrays of the updated vertices only, unlike with CSR where there is a need to rebuild edge properties for the entire graph. In addition, this design makes it easier to keep the property values in the same order as the edges in case we have to sort them after an update operation. As we show in Section 4.6, this design adds a moderate memory overhead. Naturally, if the to-be-loaded graph configuration does not include edge properties, edge property support can be disabled to save memory.

Additional Structures. Most real-life graphs include user-provided vertex IDs, e.g., a full-name string. CSR++ supports mapping of user vertex keys to internal IDs by storing them in a map and, inversely, internal IDs are mapped directly inside the segments of CSR++ using one ID mapping array per segment. For directed graphs, some algorithms, e.g., PageRank, require access to reverse edges and sometimes mappings from reverse to their corresponding forward edges (e.g., Weighted PageRank; see Section 4). To ensure fast lookup over the reverse edges and their mapping, similar to most representations, such as CSR in Green-Marl [2] and LLAMA, CSR++ reserves additional structures to store the reverse edges corresponding to each forward edge, as well as the mapping between their indices stored as an edge property. These increase the memory footprint but contribute to higher performance.

Synchronization. Synchronization in CSR++ is implemented at the segment level, using spinlocks to protect data writes. CSR++ does not support scans concurrent to updates.

3.2 Update Protocols

CSR++ supports efficient concurrent in-place mutations by allowing both single local updates (e.g., inserting edge by edge) and batch update operations.

Vertex and Edge Insertion. For vertex insertions, as described in the previous section, the `length` field in the vertex structure stores the degree of the vertex. Lengths ≥ 0 indicate a valid vertex. New vertex insertions land in the last segment. To add a vertex: in case there is enough space in the last segment, CSR++ finds the first non-valid vertex, and then sets the vertex accordingly. Setting the vertex also indicates that there is a reserved space for the corresponding vertex property entries. Otherwise, if the last segment is full, the insertion operation allocates a new one, along with new arrays for each registered vertex property.

Inserting a new segment in CSR++ is as simple as appending a new pointer to the segment array. Extending this array is lightweight, given that even for large graphs such as Twitter, CSR++ only needs to copy $\approx$ 3MB worth of pointers.

As for edge insertions, the per-vertex edge arrays use classic allocation amortization techniques for efficient edge insertions. If there is no space left to add edges, we double the size of the array through reallocation. This way we keep the size of the allocated array as a power of two, which helps amortize the allocation costs upon possible future insertions. Naturally, CSR++ can support different growing factors than 2× to enable tuning edge insertion and memory consumption performance.

Although CSR++ efficiently supports single vertex and/or edge insertions, in practice, insertions happen in batches, e.g., inserting a set of new transactions in a financial graph. Batch insertion enables CSR++ to leverage multi-threading and reduces the cost of maintaining per-vertex edge sorting. Batch insertions are implemented with the following steps:

1. Collect an input of edges grouped by their source vertices and convert both source and destination user keys to internal keys. New vertices are inserted in CSR++ and each acquires a new internal ID. We keep this step sequential in CSR++, as it is very lightweight (see Section 4.2).
2. Sort new edges (parallel for each source vertex) then insert them in direct and reverse maps (parallel for each source vertex).
3. Sort the final edge arrays using a technique that merges two sorted arrays (i.e., the old edges and the new ones) and reallocate edge properties (parallel for each modified segment) according to the new order of edges.

Vertex and Edge Deletion. Deletions are not very frequent in real-life workloads. Accordingly, we develop a very lightweight protocol of logical deletions. As presented above, for vertices, setting the `length` to a negative value indicates an invalid/deleted vertex. For edges, the separate `delete_flag` indicates deletion. Of course, vertex and edge iterators are adapted to take these flags into account and disregard deleted entities. Optionally, when deleting a vertex, the list of neighbors can be destroyed. Currently, CSR++ instead restricts access to the edges if the vertex length is negative. Since CSR++ does not store explicit edge keys, deleting an edge requires to translate the source and destination vertex keys to internal IDs and scan over the neighbor list to locate the edge to be deleted. As already mentioned, for fast scans, CSR++ keeps the per-vertex edges sorted and performs binary searches. In case storage becomes very fragmented due to many deletions, a rather heavyweight compaction operation needs to be invoked to physically remove logically deleted entities. The cost of this operation is proportional to the cost of populating the same graph from scratch. However, we expect that this operation seldom happens in real-life deployments. Additionally, segments with no deletions can be reused as-is in the compacted graph.

3.3 Algorithms on Top of CSR++

CSR++ is written in C++ and is simple to use when writing graph algorithms. To iterate over vertices, CSR++ requires a nested loop to iterate over the segments then over the vertices per segment. Using parallelism APIs, such as OpenMP [4], the nested loops can be automatically collapsed and optimized. For algorithms requiring access to edges, the vertex structure implements a `get_neighbors()` method that returns its edge list.

4 Evaluation

In this section, we answer to the following questions regarding the performance of CSR++: How does CSR++ perform on read-only and on update workloads? How much memory does CSR++ consume on these workloads? How does CSR++ perform in comparison to other read-friendly (i.e., CSR) and update-friendly graph structures (i.e., adjacency lists and LLAMA [23])?

To this end, we compare the graph-structure configurations in Table 1, using two real-world graphs [7], LiveJournal (4.8 million vertices and 68 million edges) and Twitter (41 million vertices and 1.4 billion edges), as well as the four algorithms in Table 2 in various workload configurations. Before we present the experimental results, we describe our configuration.

Table 1 Graph structures and the configurations that we use in our evaluation.

Name	**Type**	**Configuration**
CSR++	Segmentation based	Pre-allocating extra space for new edges. Deletion support enabled only on deletion workloads, in order to have fair comparison to LLAMA that does not support deletions by the default.
BGL [1]	Adjacency list	Bidirectional with default parameters.
CSR [2]	CSR	Implementation in the Green-Marl library [2].
LLAMA [3]	CSR with delta logs	Read- and space-optimized with explicit linking. The fastest overall variant of LLAMA. Deletion support enabled only on deletion workloads.

Table 2 Algorithms used in our evaluation.

Algorithm	**Description**
PageRank	Computes ranking scores for vertices based on their incoming edges.
Weakly Connected Components (WCC)	Computes affinity of vertices within a network.
Breadth-First Search (BFS)	Traverses the graph, starting from a root vertex, visits neighbors and stores distance of vertices from the root vertex, as well as parents.
Weighted PageRank	Computes ranking scores like the original PageRank and allows a weight associated with every edge. It requires access to edge properties.

Experimental Methodology. For every result point, we perform five iterations and plot the median. We report the execution time as a function of the number of threads. For most analytics workloads we use CSR as a baseline. We run our benchmarks on a two-socket, 36-core machine with 384GB of RAM. Its two 2.30Ghz Intel Xeon E5-2699 v3 CPUs have 18 cores (36 hardware threads) and 32KB, 256KB and 46MB L1, L2, and LLC caches, respectively. We disable Intel TurboBoost and do not use Intel Hyper-Threading in all experiments. Both CSR++ and the other evaluated systems are implemented in C++ and compiled using GCC 4.8.2, with optimization level `-O3` and `-fopenmp` on Oracle Linux 7.3. We use the implementation of graph algorithms from Green-Marl [2].

We use the evaluated graphs as follows. For the read-only and deletion workloads, we initially load the whole graph structures. For workloads with insertions, we initially load 80% of the graph and then insert batches of different sizes, generated using the graph-split techniques used for loading and testing in the original LLAMA paper [23]. The first split contains the 80% of the graph (≈1.1 billion edges for Twitter) that is loaded as a base graph. Then, the remaining 20% is split using a random uniform distribution over N files; we refer to N as the number of batches. Depending on the workload, we refer in figures to either the batch size (e.g., 1% corresponds to splitting the 20% in 20 batches, hence 1% of the overall graph), or the number of batches.

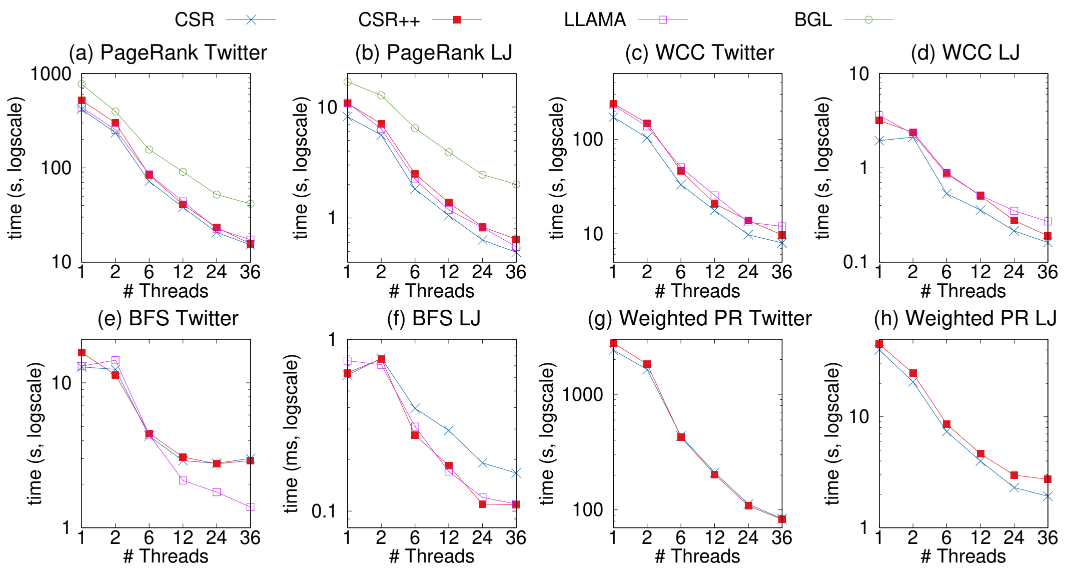

Figure 3 Read-only performance of CSR, CSR++, LLAMA, and adjacency lists (BGL).

4.1 Read-Only Workloads

We load the graph in memory and execute the evaluated algorithms. We report the execution time taken to complete each algorithm and examine how it scales with multiple threads.

Figure 3 includes the results for CSR++, CSR, BGL adjacency lists, and LLAMA. As expected, the read-only CSR provides the best performance in this workload, since with CSR, any graph access, for vertices, edges, and properties is as simple and efficient as an indexed array access. Still, CSR++ delivers performance comparable to CSR, especially in the presence of multi-threading. Over all datapoints, CSR++ is on average 15% slower than CSR, while with 36 threads, CSR++ is on average less than 10% slower than CSR.

As shown in Figure 3, we evaluate BGL adjacency lists only with PageRank. The reason for this is that the other algorithms require reverse-to-forward edge mapping, which is not supported out of the box in BGL. Still, the results of PageRank are conclusive: plain adjacency lists cannot deliver performance comparable to read-friendly structures such as CSR and CSR++. Based on these results, and for simplicity of presentation, we omit adjacency lists from the experiments in the rest of the paper.

Compared to LLAMA, CSR++ is faster for four out of the six configurations by 16% on average with 36 threads. Overall, the two systems perform within 1% of each other on average. LLAMA is faster than CSR++ for Pagerank with Livejournal and for BFS on Twitter.

For Weighted PageRank, we only evaluate CSR and CSR++ and omit LLAMA because it does not support edge properties out of the box. CSR++ still performs close to CSR as shown in Figures 3h and 3i. With 36 threads, CSR++ is 1% faster than CSR on Twitter and 42% slower for Livejournal. The slowdown in Livejournal is due to the small size of the graph: with CSR's representation, all data is served from the last-level cache, while CSR++ needs to slightly spill to main memory. These results show that the representation of edge properties in CSR++ performs comparably to CSR, especially on large graphs.

Overall, CSR++ is very fast on read-only workloads, especially in the presence of concurrency, which is the intended use case of graph analytics.

4.2 Updates: Vertex Insertions

Vertex insertions in CSR++ are very lightweight, mainly due to segmentation (see Section 3.2). Table 3 shows the time to insert different number of vertices on a fully loaded Twitter graph (the choice of the graph has little impact on the performance of vertex insertions in CSR++), when the graph contains either no vertex properties or 50 vertex properties. Vertex insertions are fast: With 10M insertions, inserting a vertex takes an average of 118 and 126 nanoseconds per-vertex with no and 50 properties, respectively. Vertex properties are lightweight in CSR++, as they require just one memory allocation per property per segment.

Table 3 Time to add new vertices to Twitter graph in milliseconds.

#Vertices	10K	100K	1M	10M
Time (ms) – 0 vertex properties	1.6	11	120	1188
Time (ms) – 50 vertex properties	10	32	181	1259

4.3 Updates: Edge Insertions

First, we evaluate the time to insert all edges of one batch in both forward and reverse structures (plus edge semi-sorting). Figure 4a shows the results. CSR++ completes this full batch insertion one order of magnitude faster than CSR. As expected, CSR completes all batch insertions in the same amount of time, regardless of the batch size. In contrast, CSR++ performs localized graph updates and thus delivers fast performance that is proportional to the batch size.

Next, we examine the scalability of edge insertions with CSR++ using the same workloads and exploiting multi-threading. The results are shown in Figure 4b. We isolate insertions by removing the edge semi-sorting that takes a significant amount of overall insertion time. CSR++ achieves good scalability for up to 12 threads. For more threads, performance does not improve, in part because of the effects of memory contention and NUMA, but mainly because of actual vertex contention: Twitter is a very skewed graph, hence many of the edge insertions land in the same high-degree vertices, hindering parallelism. Note that for these workloads, due to limited space, we only show the results with Twitter; we reach very similar conclusions with the smaller LiveJournal graph.

We further compare CSR++ to LLAMA for graph insertions. In Figure 5, we compare the edge-insertion latency and memory consumption. We apply 1000 batches of insertions (equivalent to the 0.02% workload in Figure 4), and print the memory usage and timestamp

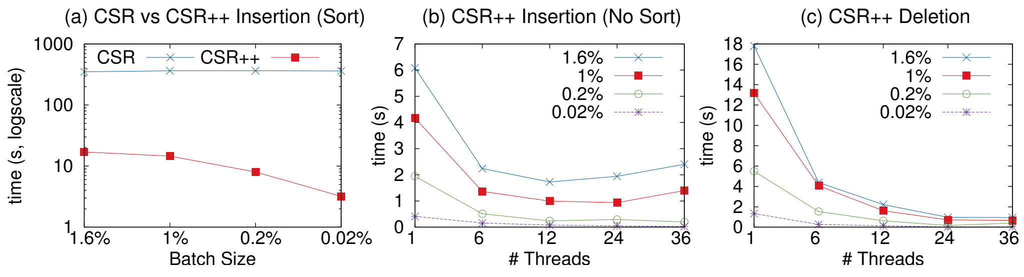

Figure 4 Graph mutations on Twitter. (a) Time to insert batches of edges of different sizes in CSR and CSR++ and sorting the edge arrays using 36 threads; (b) Time to insert different batch sizes in CSR++ without sorting using 1 to 36 threads; (c) Time to delete different batch sizes in CSR++.

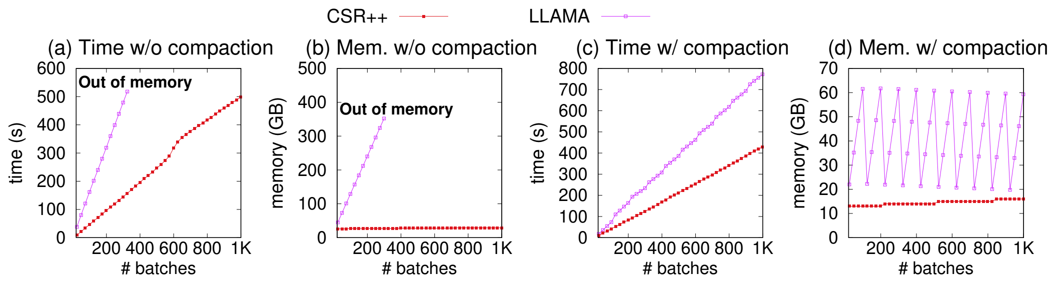

Figure 5 Memory consumption and batch-insertion latency of update workloads with 36 threads. (a) & (b): Comparing CSR++ and LLAMA without compaction. (c) & (d): Comparing CSR++ and LLAMA with compaction after every 100th batch insertion.

after inserting each batch. As shown in Figures 5a and 5b, the memory usage of LLAMA explodes after applying 370 batches, causing the system to run out of memory. In contrast, CSR++ consumes memory proportional to the actual graph size. Additionally, CSR++ is up to 2.7× faster in performing the insertions.

LLAMA provides a function to compact all snapshots into a single one. Figures 5c and 5d show the performance of CSR++ and LLAMA with compaction. After every 100 batches, we compact all 100 snapshots. LLAMA's memory usage increases until compaction is invoked, but it is still higher than CSR++, even immediately after compaction. Note that the compaction method in LLAMA does not provide instructions for building the reverse edges, hence these figures show the performance of inserting only forward edges. In principle, building the reverse edges is quite more expensive than building forward edges, i.e., if the reverse operation was included, the cost of compacting would be significantly higher. Compacting 100 snapshots with only direct edges in LLAMA takes up to 40 seconds with a single thread and 5-7 seconds with 36 threads. As shown in Figure 5c, CSR++ is still consistently faster than LLAMA by a factor of approximately 1.8×.

4.4 Updates: Edge Deletions

To support edge deletions, we modify our vertex and edge iterators in CSR++ to check whether a vertex or an edge is deleted, using the embedded flag in their respective structures. For LLAMA, we enable the deletion vector which similarly adds the cost of checking whether edges are deleted. Figure 4c shows the time CSR++ takes to perform edge deletions. Each data point represents the time to delete a whole batch of edges. The scalability is almost linear relative to the number of threads, and, as we increase the batch size, the effect of multithreading is more noticeable.

With a single thread, deletions are more computationally heavy, and therefore slower than insertions, as can be seen in Figures 4b and 4c. As we mention earlier, CSR++ does not store edge indices (it would be very memory consuming), which means that for every edge that is deleted, the thread needs to perform a (binary) search to find the target edge to delete logically. Note that CSR++ can "easily" also support physical deletion of edges, at the additional cost of having to reshuffle edge properties to match the new edge array.

4.5 Analytics After Graph Updates

To evaluate CSR++ in a mutation context, we first load the initial 80% of the graph and simulate the insertion of a stream of updates (batches of new edges and new vertices), then we evaluate PageRank. For insertions, this workload evaluates the impact of updates on

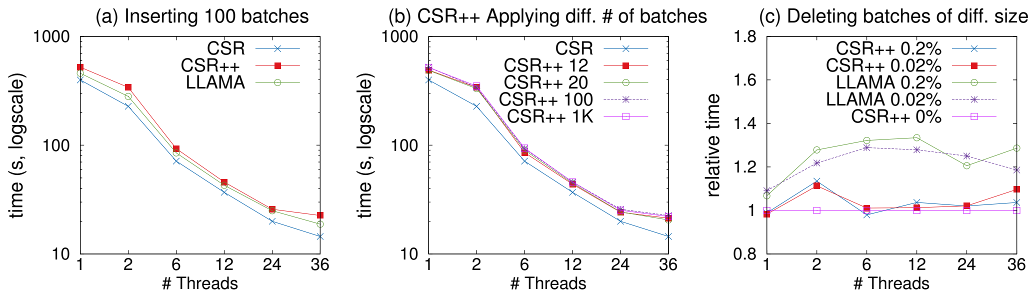

Figure 6 PageRank performance after graph updates. (a) Comparing performance of CSR, CSR++, and LLAMA after applying 100 (of size 0.2%) batches of new edges; (b) Performance of CSR++ after applying 12 (1.6%), 20 (2%), 100 (0.2%) and 1000 (0.02%) batches of new edges, CSR is used as a baseline; (c) Performance of CSR++ and LLAMA after deleting one batch of edges of different sizes.

the performance of the graph structures, e.g., for CSR++ reallocations of edge arrays and the added pointers to newly-allocated segments. For deletions, we examine the overhead of the extra conditional branch to check the deleted flags and the cost of virtual deletions.

Figure 6 shows the performance of PagerRank with CSR++ and LLAMA after applying mutations to the graph. In Figure 6a, we observe that, after inserting 100 batches of new edges, the performance of CSR++ only decreases by a factor of less than 1.25× as compared to CSR, which shows the moderate overhead that is caused by the continuous reallocations of edge arrays and the copy-on-write of the indirection layer. Additionally, LLAMA is faster than CSR++ by a factor of 1.12× but consumes ≈5× more memory than CSR++ (see also Table 4). This is due to the 100 snapshots LLAMA stores as multi-versioning support. If we need to perform analytics on the latest version of the graph (which is the case of most real-world scenarios), the significant memory overhead of these snapshots may not be worth the minimal performance improvement. Figure 6b shows a breakdown of the performance of CSR++ when inserting different numbers of new batches: increasing the number of batches results in more reallocations and copy-on-write operations. CSR++ scales well in all cases and keeps the moderate overhead of ≈1.25× over CSR even after inserting 1000 batches.

Finally, Figure 6c shows the performance of CSR++ and LLAMA after deleting one batch of edges of different sizes, relative to CSR++'s performance without deletions. As we mention earlier, we modified the iterators in CSR++ to check for deletion flags in vertices and edges. We delete up to 23 million edges from the 1.47 billion total edges of Twitter, and as expected, the performance is similar to that of the baseline (i.e., without deletions). The extra conditional branches in CSR++ do not introduce considerable overhead. In case there are only few deletions, branch prediction makes sure that these deletion checks have minimal overhead, resulting in performance close to the original implementation (i.e., without deletion checks). In contrast, LLAMA's performance significantly suffers when enabling support for deletions and makes LLAMA ≈30% slower than CSR++.

4.6 Memory Footprint

We calculate the memory footprint of Twitter and LiveJournal graphs stored in CSR, CSR++, and LLAMA (read-optimized), both just after loading them in memory and after applying different numbers of batch insertions on Twitter (Table 4).

As shown in Table 4, CSR is the most compact representation and consumes the least memory – at the cost of mutability. The memory overhead of LLAMA is small when storing one snapshot (i.e., before applying mutations), but as can be seen in the same Table 4, this

Table 4 Memory footprint of different graph structures in GB in read-only workloads and after inserting a different number of batches (Twitter-x, where x is the number of batches).

Graph Structure	LiveJournal	Twitter	Twitter-12	Twitter-20	Twitter-100
CSR	0.53	11.09	11.09	11.09	11.09
CSR++ read-only	0.57	11.54	-	-	-
CSR++	0.82	16.55	16.55	16.55	16.55
LLAMA	0.58	11.56	21.66	27.03	78.00
LLAMA implicit linking	0.58	11.56	19.02	23.99	73.64

overhead increases steeply when applying batches. It is primarily due to storing different delta-snapshots of the graph for versioning. As mentioned earlier, for realistic workloads such as applying updates at a high frequency and then running analytics on recent versions of the graph, this memory overhead may lead to out-of-memory errors. As a reference, we include a second variant of LLAMA with implicit linking across snapshot versions, which trades performance for memory. The memory savings of this variant are low, however, and its performance is significantly worse (hence why our performance figures do not include it).

The default version of CSR++ has a moderate memory overhead of 33% compared to CSR, due to the pre-allocation of extra space for edge arrays. When this optimization is disabled, memory is allocated in a tight manner and CSR++ consumes closely to CSR.

5 Concluding Remarks

We introduced CSR++, a new concurrent graph data structure that is as fast as the fastest existing read-only graph structure, namely CSR, while enabling fast and memory-efficient in-place graph mutations. CSR++ achieves this sweet spot by combining the array-based design of CSR with the mutability of adjacency lists. In practice, CSR++ is within 10% of the performance of CSR and delivers an order of magnitude faster updates.

Future work includes using smarter synchronization mechanisms in CSR++ (such as e.g., developing lock-free protocols to avoid per-segment locking) as well as improving scalability with concurrent updates. Furthermore, we intend to explore smarter, faster, and locality-preserving memory reallocations using different memory allocators that are better suited for multithreaded applications.

References

1 Boost Adjacency-List Documentation. `https://www.boost.org/doc/libs/1_67_0//libs/graph/doc/adjacency_list.html`.
2 Green-Marl Code. `https://github.com/stanford-ppl/Green-Marl`.
3 LLAMA Code. `https://github.com/goatdb/llama`.
4 OpenMP. `https://www.openmp.org`.
5 PGQL: Property Graph Query Language. `http://pgql-lang.org/`.
6 Property Graph Model. `https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model`.
7 SNAP (2014). Stanford Network Analysis Platform. `http://snap.stanford.edu/snap`.
8 SPARQL Query Language For RDF. `http://www.w3.org/TR/rdf-sparql-query/`.
9 Tinkerpop, Gremlin. `https://github.com/tinkerpop/gremlin/wiki`.
10 Tim Berners-Lee, James Hendler, Ora Lassila, et al. The Semantic Web. *Scientific american*, 284(5), 2001.

11 Raymond Cheng, Enhong Chen, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, and Feng Zhao. Kineograph: Taking The Pulse Of A Fast-changing And Connected World. In *EuroSys*, 2012.

12 Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A Framework For Parallel Graph Algorithms Using Work-efficient Bucketing. In *SPAA*, 2017.

13 Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. In *SIGMOD*, 2019.

14 David Ediger, Jason Riedy, David A. Bader, and Henning Meyerhenke. Tracking Structure of Streaming Social Networks. In *IPDPSW*, 2011.

15 Soukaina Firmli and Dalila Chiadmi. A Review Of Engines For Graph Storage And Mutations. In *Innovation In Information Systems And Technologies To Support Learning Research*, 2020.

16 Gartner. Gartner Top 10 Data And Analytics Trends For 2019. `https://www.gartner.com/smarterwithgartner/gartner-top-10-data-analytics-trends/`.

17 Michael Haubenschild, Manuel Then, Sungpack Hong, and Hassan Chafi. ASGraph: A Mutable Multi-versioned Graph Container With High Analytical Performance. In *GRADES*, 2016.

18 S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. PGX.D: A Fast Distributed Graph Processing Engine. In *SC*, 2015.

19 Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL For Easy And Efficient Graph Analysis. In *ASPLOS*, 2012.

20 Nikolaos D. Kallimanis and Eleni Kanellou. Wait-Free Concurrent Graph Objects With Dynamic Traversals. In *OPODIS*, 2016.

21 Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, 2017.

22 Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

23 P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In *ICDE*, 2015.

24 K. Madduri and D.A. Bader. Compact Graph Representations And Parallel Connectivity Algorithms For Massive Dynamic Network Analysis. In *IPDPS*, 2009.

25 Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *EuroSys*, 2019.

26 Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-level Abstraction And High Performance For Graph Mining. In *SOSP*, 2019.

27 Neo4j. Neo4j Graph Database. `http://www.neo4j.org`.

28 Oracle. Parallel Graph Analytics (PGX). `https://www.oracle.com/middleware/technologies/parallel-graph-analytix.html`.

29 Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The Pagerank Citation Ranking: Bringing Order To The Web. Technical report, Stanford InfoLab, 1999.

30 Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. GRAPHITE: An Extensible Graph Traversal Framework For Relational Database Management Systems. In *SSDBM*, 2015.

31 Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. PGX.ISO: Parallel And Efficient In-memory Engine For Subgraph Isomorphism. In *GRADES*, 2014.

32 Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *GRADES*, 2017.

33 Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-SPARQL: A Hybrid Engine For Querying Large Attributed Graphs. In *ACM CIKM*, 2012.

34 Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. Using Domain-specific Languages For Analytic Graph Databases. *PVLDB*, 9(13):1257–1268, September 2016.

35 Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework For Shared Memory. In *PPoPP*, 2013.

36 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A Tool Suite For Large-Scale Complex Network Analysis. *Network Science*, 4(4):508–530, 2016.

37 Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In *SIGMOD*, 2015.

38 Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A Property Graph Query Language. In *GRADES*, 2016.

39 Brian Wheatman and Helen Xu. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *HPEC*, 2018.

40 Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed Graph Engine For Web Scale RDF Data. *PVLDB*, 6(4), 2013.

41 Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-Aware Graph-Structured Analytics. In *PPoPP*, 2015.