



HAL
open science

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

Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis,
Alexander Weld, Dalila Chiadmi, Sungpack Hong, Hassan Chafi

► **To cite this version:**

Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, et al.. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure. 24th International Conference on Principles of Distributed Systems (OPODIS '20), Dec 2020, Strasbourg (on line), France. 10.4230/LIPIcs.OPODIS.2020.17 . hal-03060095

HAL Id: hal-03060095

<https://hal.science/hal-03060095>

Submitted on 13 Dec 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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, Morocco

Vasileios Trigonakis*

Oracle Labs, Switzerland

Jean-Pierre Lozi*

Oracle Labs, Switzerland

Iraklis Psaroudakis*

Oracle Labs, Switzerland

Alexander Weld*

Oracle Labs, Switzerland

Dalila Chiadmi†

Mohammed V University in Rabat, Ecole Mohammadia d'Ingénieurs, SIP Research Team, Morocco

Sungpack Hong*

Oracle Labs, USA

Hassan Chafi*

Oracle Labs, USA

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

* firstname.lastname@oracle.com

† chiadmi@emi.ac.ma



© Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong and Hassan Chafi; licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 17; pp. 17:1–17:16



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

41 **1 Introduction**

42 Graph processing is an invaluable tool for data analytics, as illustrated by the plethora of
43 relatively recent work aiming at achieving high performance for graph algorithms [12, 17, 23,
44 34, 35, 41], such as PageRank [29], or graph querying/mining [13, 21, 26, 27, 31, 33, 38], e.g.,
45 using PGQL [5]. At the heart of each graph system lies the graph data structure, responsible
46 for holding the vertices and the edges comprising the graph, and whose performance largely
47 contributes to the general performance of the system. The ideal graph structure should offer
48 excellent read-only performance, fast mutations (i.e., vertex or edge insertions and deletions),
49 and low memory consumption with or without mutations.

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

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

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

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

87 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 $2\times$ faster in applying batched updates, and consumes $4\times$ less memory when 100 update batches are applied on a base graph.

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]).

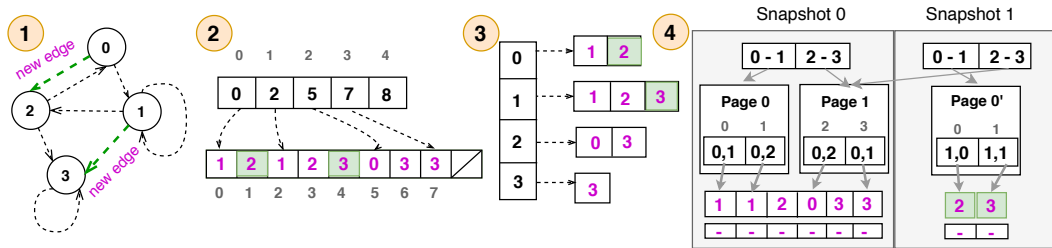
Graphs can be represented with different models and data representations. A popular model is the RDF (Resource Description Framework) graph data model [8], which became popular with the rise of the semantic web [10]. RDF regularizes the graph representation as a set of triples. RDF adds links for all 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



■ **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.

133 directed edge from a source vertex v_s to a destination vertex v_d . An adjacency matrix is not
 134 preferred for sparse graphs, i.e., graphs where the number of edges $E \ll V^2$, due to increased
 135 memory footprint and decreased performance in analytics.

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

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

154 2.2 Graph Mutations

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

159 **In-Place Updates.** Techniques that use in-place updates employ the aforementioned static
 160 data structures in a way that allows for in-place digestion of sets with insertions and deletions
 161 of vertices and edges, without requiring the expensive rebuild of the data structure. For
 162 instance, Dense [20] is a concurrent graph adjacency matrix that supports mutations and
 163 partial traversals through a coordination protocol, but does not handle graph properties.
 164 NetworKit [36], in order to perform edge insertions, stores adjacencies vectors that double the
 165 size of the initial array to reserve enough space for new incoming edges. Madduri et al. [24]

166 use the same underlying technique but define a configurable size of the new edge array instead
167 of using factor 2. Ediger et al. [14] implement blocked adjacency lists and allow insertions
168 by appending new blocks and updating pointers. Wheatman et al. [39] implement a variant
169 of CSR that leaves space at the end of each adjacency list to allow efficient single-threaded
170 mutations. We employ similar techniques in CSR++ to ingest mutations, but in a parallel
171 manner while also handling graph property mutations.

172 **Batching.** Regarding the sources of changes, they can be continuous streams of updates [11,
173 14] or single changes applied as “local” mutations. Generally, when applying a batch of
174 updates, frameworks perform pre-processing to re-arrange the batches in ways that can
175 speed-up the mutations. For instance, Madduri et al. [24] apply techniques on the list of new
176 edges, such as sorting, re-ordering, and partitioning, in order to exploit parallelism at the
177 time of the changes application. Similarly, CSR++ groups updates by their source vertices,
178 and uses multiple threads to perform fast edge insertions (see Section 3.2).

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

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

198 **3 CSR++: Design and Implementation**

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

205 **3.1 Graph Topology and Properties**

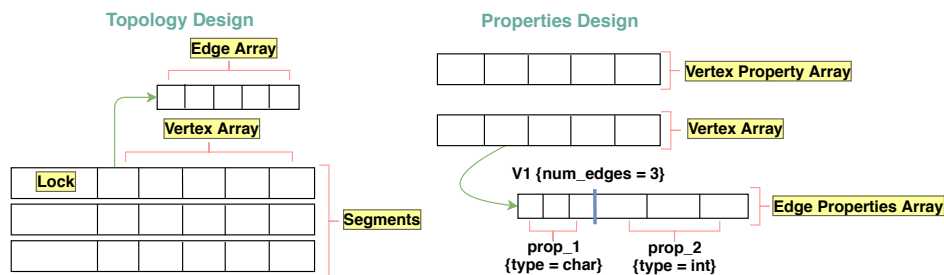
206 CSR++ is a concurrent structure that stores the graph in memory using segmentation techniques.
207 It allows in-place insertions by allocating additional space for new incoming edges and supports
208 logical deletions of vertices and edges. Figure 2 shows the building blocks of CSR++.

17:6 CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure

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

218 **Vertices.** Each vertex stores its degree, a pointer to its list of neighbors, and optionally
 219 a pointer to the property values of its edges. This design resembles a mix of CSR and
 220 adjacency lists, however, adding a new vertex in CSR++ is faster (see Section 3.2) considering
 221 that the vertex array is segmented, i.e., we do not need to copy the whole vertex array to
 222 add or remove entries. CSR++ does not store explicit IDs for vertices nor edges, but since
 223 all segments store a fixed number `NUM_V_SEG` of vertices, we can compute implicit IDs for
 224 vertices using the segment ID and the index of the vertex in the segment: $\text{global_v_id} =$
 225 $(\text{seg_id} * \text{NUM_V_SEG}) + \text{v_id}$. Overall, the vertex structure consists of the following fields:
 226 ■ **length** (4 bytes): The vertex degree. A length of -1 indicates a deleted vertex.
 227 ■ **neighbors** (8 bytes): A pointer to the set of neighbors. As a space optimization, if
 228 **length** = 1, this field directly contains the neighbor's vertex ID.
 229 ■ **edge_properties** (8 bytes): A pointer to the set of edge properties. As a space
 230 optimization, this field can be disabled in case the graph does not define edge properties.

231 **Edges.** CSR++ represents the neighbors list of a vertex by an array of edges, where every
 232 entry stores the coordinates (i.e., the vertex ID and the segment ID) of the corresponding
 233 neighbor. At loading time, the edges are sorted; as with CSR and LLAMA, keeping the
 234 edges sorted allows for better cache performance. Moreover, this semi-sorting is necessary for
 235 CSR++ in a deletion-frequent context, as we use binary search to locate edges. Additionally,
 236 as an optimization for update-friendliness, CSR++ can be configured to create extra empty
 237 space for new incoming edges during graph loading (see Section 3.2). The edge structure
 238 consists of the following fields:
 239 ■ **deleted_flag** (2 bytes): For logical deletion of edges.
 240 ■ **vertex_id** (2 bytes): The index of the neighbor in the segment; using 16 bits allows
 241 for segments with a capacity `NUM_V_SEG` of up to 65536 entries.
 242 ■ **segment_id** (4 bytes): The segment ID where the neighbor is stored.



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

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

247 **Properties.** Vertex property values are stored in arrays parallel to the vertices array. CSR++
 248 keeps a vector of pointers to each vertex property array within the segment. The size of
 249 each array is therefore `NUM_V_SEG * sizeof(Property_Type)`. For edge properties, we use the
 250 same segmentation approach as vertices. If the user enables edge properties, each vertex
 251 structure stores a pointer to an array of edge property values, as shown in Figure 2. In
 252 case of multiple properties, we allocate an array that stores the values for different edge
 253 properties in a cache-aligned manner. In order to locate a specific edge property p , we use
 254 offsets and the position of its values can be calculated given the type of that property T_p ,
 255 the index i of the edge in the neighbor list, and the degree d of the vertex v . For example,
 256 suppose the user registers n edge properties, then the total size of the edge properties of
 257 a vertex v is $\sum_{p=1}^n (\text{sizeof}(Type_p) * d)$. Similarly, the values of the x th property begin at
 258 $Values(x) = \sum_{p=1}^x (\text{sizeof}(Type_p) * d)$. Accordingly, the property value for the x th property
 259 of the edge i is $Value(x, i) = Values(x) + (i * \text{sizeof}(Type_x))$.

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

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

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

279 3.2 Update Protocols

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

282 **Vertex and Edge Insertion.** For vertex insertions, as described in the previous section, the
 283 `length` field in the vertex structure stores the degree of the vertex. Lengths ≥ 0 indicate a
 284 valid vertex. New vertex insertions land in the last segment. To add a vertex: in case there

285 is enough space in the last segment, CSR++ finds the first non-valid vertex, and then sets the
 286 vertex accordingly. Setting the vertex also indicates that there is a reserved space for the
 287 corresponding vertex property entries. Otherwise, if the last segment is full, the insertion
 288 operation allocates a new one, along with new arrays for each registered vertex property.

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

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

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

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

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

326 3.3 Algorithms on Top of CSR++

327 CSR++ is written in C++ and is simple to use when writing graph algorithms. To iterate over
 328 vertices, CSR++ requires a nested loop to iterate over the segments then over the vertices per
 329 segment. Using parallelism APIs, such as OpenMP [4], the nested loops can be automatically

330 collapsed and optimized. For algorithms requiring access to edges, the vertex structure
 331 implements a `get_neighbors()` method that returns its edge list.

332 4 Evaluation

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

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

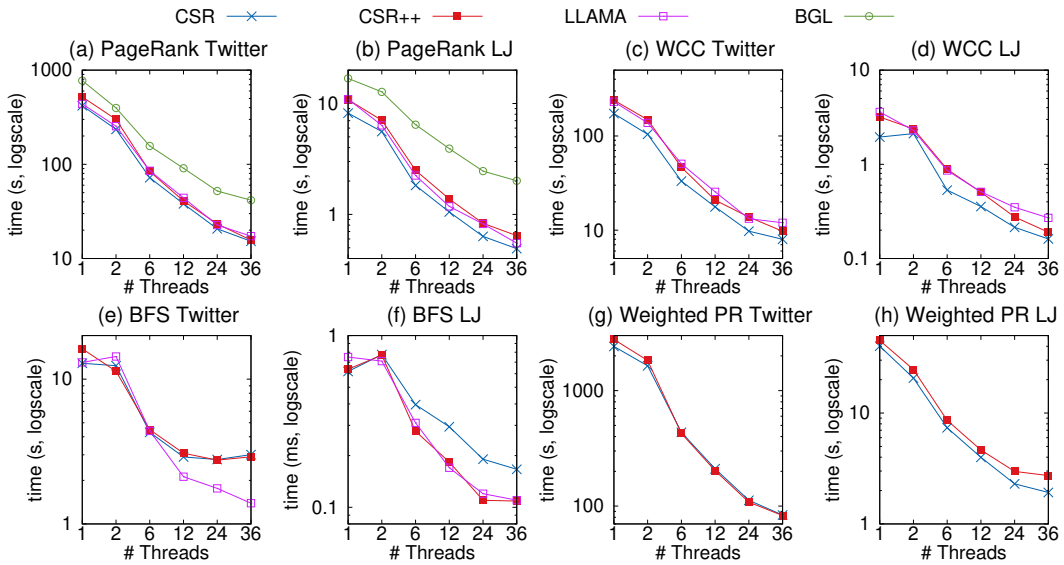
■ **Table 1** Graph structures and the configurations that we use in our evaluation.

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.

341 **Experimental Methodology.** For every result point, we perform five iterations and plot the
 342 median. We report the execution time as a function of the number of threads. For most
 343 analytics workloads we use CSR as a baseline. We run our benchmarks on a two-socket,
 344 36-core machine with 384GB of RAM. Its two 2.30Ghz Intel Xeon E5-2699 v3 CPUs have
 345 18 cores (36 hardware threads) and 32KB, 256KB and 46MB L1, L2, and LLC caches,
 346 respectively. We disable Intel TurboBoost and do not use Intel Hyper-Threading in all
 347 experiments. Both CSR++ and the other evaluated systems are implemented in C++ and
 348 compiled using GCC 4.8.2, with optimization level `-O3` and `-fopenmp` on Oracle Linux 7.3.
 349 We use the implementation of graph algorithms from Green-Marl [2].



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

350 We use the evaluated graphs as follows. For the read-only and deletion workloads, we
 351 initially load the whole graph structures. For workloads with insertions, we initially load
 352 80% of the graph and then insert batches of different sizes, generated using the graph-split
 353 techniques used for loading and testing in the original LLAMA paper [23]. The first split
 354 contains the 80% of the graph (≈ 1.1 billion edges for Twitter) that is loaded as a base graph.
 355 Then, the remaining 20% is split using a random uniform distribution over N files; we refer
 356 to N as the number of batches. Depending on the workload, we refer in figures to either the
 357 batch size (e.g., 1% corresponds to splitting the 20% in 20 batches, hence 1% of the overall
 358 graph), or the number of batches.

359 4.1 Read-Only Workloads

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

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

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

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

377 For Weighted PageRank, we only evaluate CSR and CSR++ and omit LLAMA because it

378 does not support edge properties out of the box. CSR++ still performs close to CSR as shown
 379 in Figures 3h and 3i. With 36 threads, CSR++ is 1% faster than CSR on Twitter and 42%
 380 slower for Livejournal. The slowdown in Livejournal is due to the small size of the graph:
 381 with CSR’s representation, all data is served from the last-level cache, while CSR++ needs to
 382 slightly spill to main memory. These results show that the representation of edge properties
 383 in CSR++ performs comparably to CSR, especially on large graphs.

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

386 4.2 Updates: Vertex Insertions

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

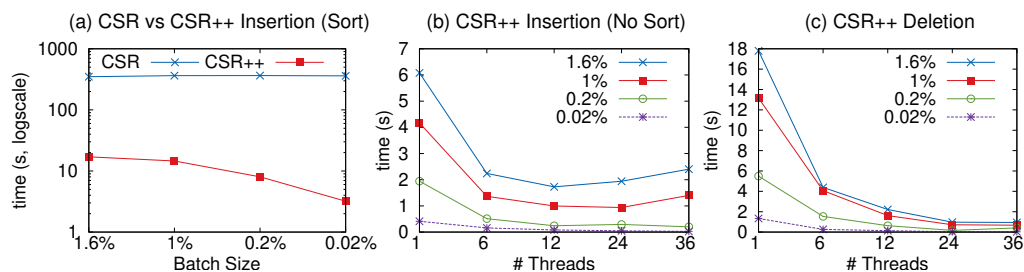
■ **Table 3** Time to add new vertices to Twitter graph in milliseconds.

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

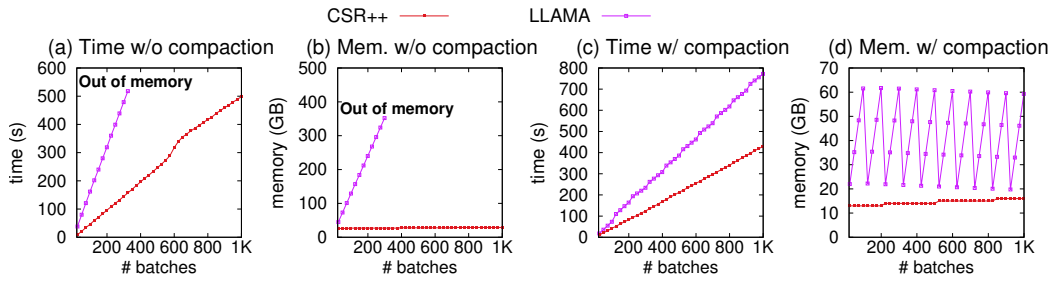
394 4.3 Updates: Edge Insertions

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

401 Next, we examine the scalability of edge insertions with CSR++ using the same workloads
 402 and exploiting multi-threading. The results are shown in Figure 4b. We isolate insertions



■ **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.

403 by removing the edge semi-sorting that takes a significant amount of overall insertion time.
 404 CSR++ achieves good scalability for up to 12 threads. For more threads, performance does
 405 not improve, in part because of the effects of memory contention and NUMA, but mainly
 406 because of actual vertex contention: Twitter is a very skewed graph, hence many of the edge
 407 insertions land in the same high-degree vertices, hindering parallelism. Note that for these
 408 workloads, due to limited space, we only show the results with Twitter; we reach very similar
 409 conclusions with the smaller LiveJournal graph.

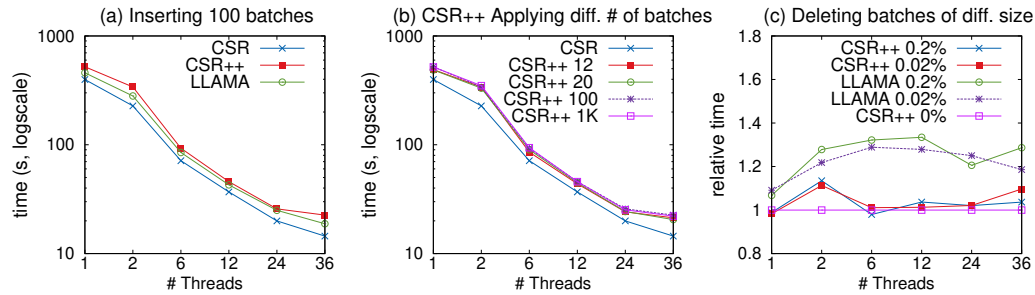
410 We further compare CSR++ to LLAMA for graph insertions. In Figure 5, we compare
 411 the edge-insertion latency and memory consumption. We apply 1000 batches of insertions
 412 (equivalent to the 0.02% workload in Figure 4), and print the memory usage and timestamp
 413 after inserting each batch. As shown in Figures 5a and 5b, the memory usage of LLAMA
 414 explodes after applying 370 batches, causing the system to run out of memory. In contrast,
 415 CSR++ consumes memory proportional to the actual graph size. Additionally, CSR++ is up to
 416 $2.7\times$ faster in performing the insertions.

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

428 4.4 Updates: Edge Deletions

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

436 With a single thread, deletions are more computationally heavy, and therefore slower



■ **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.

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

442 4.5 Analytics After Graph Updates

443 To evaluate CSR++ in a mutation context, we first load the initial 80% of the graph and
 444 simulate the insertion of a stream of updates (batches of new edges and new vertices), then
 445 we evaluate PageRank. For insertions, this workload evaluates the impact of updates on
 446 the performance of the graph structures, e.g., for CSR++ reallocations of edge arrays and the
 447 added pointers to newly-allocated segments. For deletions, we examine the overhead of the
 448 extra conditional branch to check the deleted flags and the cost of virtual deletions.

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

462 Finally, Figure 6c shows the performance of CSR++ and LLAMA after deleting one batch
 463 of edges of different sizes, relative to CSR++’s performance without deletions. As we mention
 464 earlier, we modified the iterators in CSR++ to check for deletion flags in vertices and edges. We
 465 delete up to 23 million edges from the 1.47 billion total edges of Twitter, and as expected, the
 466 performance is similar to that of the baseline (i.e., without deletions). The extra conditional
 467 branches in CSR++ do not introduce considerable overhead. In case there are only few
 468 deletions, branch prediction makes sure that these deletion checks have minimal overhead,

469 resulting in performance close to the original implementation (i.e., without deletion checks).
 470 In contrast, LLAMA’s performance significantly suffers when enabling support for deletions
 471 and makes LLAMA $\approx 30\%$ slower than CSR++.

472 4.6 Memory Footprint

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

■ **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	16.55	16.55	16.55
CSR++	0.82	16.55	21.66	27.03	78.00
LLAMA	0.58	11.56	21.66	27.03	78.00
LLAMA implicit linking	0.58	11.56	19.02	23.99	73.64

476 As shown in Table 4, CSR is the most compact representation and consumes the least
 477 memory—at the cost of mutability. The memory overhead of LLAMA is small when storing
 478 one snapshot (i.e., before applying mutations), but as can be seen in the same Table 4, this
 479 overhead increases steeply when applying batches. It is primarily due to storing different
 480 delta-snapshots of the graph for versioning. As mentioned earlier, for realistic workloads
 481 such as applying updates at a high frequency and then running analytics on recent versions
 482 of the graph, this memory overhead may lead to out-of-memory errors. As a reference, we
 483 include a second variant of LLAMA with implicit linking across snapshot versions, which
 484 trades performance for memory. The memory savings of this variant are low, however, and
 485 its performance is significantly worse (hence why our performance figures do not include it).

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

489 5 Concluding Remarks

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

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

500 — **References** —

- 501 **1** Boost Adjacency-List Documentation. https://www.boost.org/doc/libs/1_67_0/libs/graph/doc/adjacency_list.html.
502
- 503 **2** Green-Marl Code. <https://github.com/stanford-ppl/Green-Marl>.
- 504 **3** LLAMA Code. <https://github.com/goatdb/llama>.
- 505 **4** OpenMP. <https://www.openmp.org>.
- 506 **5** PGQL: Property Graph Query Language. <http://pgql-lang.org/>.
- 507 **6** Property Graph Model. <https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>.
508
- 509 **7** SNAP (2014). Stanford Network Analysis Platform. <http://snap.stanford.edu/snap>.
- 510 **8** SPARQL Query Language For RDF. <http://www.w3.org/TR/rdf-sparql-query/>.
- 511 **9** Tinkerpop, Gremlin. <https://github.com/tinkerpop/gremlin/wiki>.
- 512 **10** Tim Berners-Lee, James Hendler, Ora Lassila, et al. The Semantic Web. *Scientific american*,
513 284(5), 2001.
- 514 **11** Raymond Cheng, Enhong Chen, Ji Hong, Aapo Kyrola, Youshan Miao, Xuettian Weng,
515 Ming Wu, Fan Yang, Lidong Zhou, and Feng Zhao. Kineograph: Taking The Pulse Of A
516 Fast-changing And Connected World. In *EuroSys*, 2012.
- 517 **12** Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A Framework For Parallel
518 Graph Algorithms Using Work-efficient Bucketing. In *SPAA*, 2017.
- 519 **13** Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthas-
520 arathy. Fractal: A General-Purpose Graph Pattern Mining System. In *SIGMOD*, 2019.
- 521 **14** David Ediger, Jason Riedy, David A. Bader, and Henning Meyerhenke. Tracking Structure of
522 Streaming Social Networks. In *IPDPSW*, 2011.
- 523 **15** Soukaina Firmli and Dalila Chiadmi. A Review Of Engines For Graph Storage And Mutations.
524 In *Innovation In Information Systems And Technologies To Support Learning Research*, 2020.
- 525 **16** Gartner. Gartner Top 10 Data And Analytics Trends For 2019. <https://www.gartner.com/smarterwithgartner/gartner-top-10-data-analytics-trends/>.
526
- 527 **17** Michael Haubenschild, Manuel Then, Sungpack Hong, and Hassan Chafi. ASGraph: A
528 Mutable Multi-versioned Graph Container With High Analytical Performance. In *GRADES*,
529 2016.
- 530 **18** S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. PGX.D: A
531 Fast Distributed Graph Processing Engine. In *SC*, 2015.
- 532 **19** Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL For
533 Easy And Efficient Graph Analysis. In *ASPLOS*, 2012.
- 534 **20** Nikolaos D. Kallimanis and Eleni Kanellou. Wait-Free Concurrent Graph Objects With
535 Dynamic Traversals. In *OPODIS*, 2016.
- 536 **21** Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu.
537 Graphflow: An Active Graph Database. In *SIGMOD*, 2017.
- 538 **22** Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation
539 on Just a PC. In *OSDI*, 2012.
- 540 **23** P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics
541 Using Large Multiversioned Arrays. In *ICDE*, 2015.
- 542 **24** K. Madduri and D.A. Bader. Compact Graph Representations And Parallel Connectivity
543 Algorithms For Massive Dynamic Network Analysis. In *IPDPS*, 2009.

17:16 CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure

- 544 **25** Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing
545 of Streaming Graphs. In *EuroSys*, 2019.
- 546 **26** Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-level Abstraction And High
547 Performance For Graph Mining. In *SOSP*, 2019.
- 548 **27** Neo4j. Neo4j Graph Database. <http://www.neo4j.org>.
- 549 **28** Oracle. Parallel Graph Analytics (PGX). [https://www.oracle.com/middleware/
550 technologies/parallel-graph-analytix.html](https://www.oracle.com/middleware/technologies/parallel-graph-analytix.html).
- 551 **29** Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The Pagerank Citation
552 Ranking: Bringing Order To The Web. Technical report, Stanford InfoLab, 1999.
- 553 **30** Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. GRAPHITE: An Extensible
554 Graph Traversal Framework For Relational Database Management Systems. In *SSDBM*, 2015.
- 555 **31** Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee.
556 PGX.ISO: Parallel And Efficient In-memory Engine For Subgraph Isomorphism. In *GRADES*,
557 2014.
- 558 **32** Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris
559 Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching
560 Engine. In *GRADES*, 2017.
- 561 **33** Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-SPARQL: A Hybrid Engine For Querying
562 Large Attributed Graphs. In *ACM CIKM*, 2012.
- 563 **34** Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan
564 Chafi. Using Domain-specific Languages For Analytic Graph Databases. *PVLDB*, 9(13):1257–
565 1268, September 2016.
- 566 **35** Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework For
567 Shared Memory. In *PPoPP*, 2013.
- 568 **36** Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A Tool Suite
569 For Large-Scale Complex Network Analysis. *Network Science*, 4(4):508–530, 2016.
- 570 **37** Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and
571 Guo Tong Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In *SIGMOD*,
572 2015.
- 573 **38** Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A
574 Property Graph Query Language. In *GRADES*, 2016.
- 575 **39** Brian Wheatman and Helen Xu. Packed Compressed Sparse Row: A Dynamic Graph
576 Representation. In *HPEC*, 2018.
- 577 **40** Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed
578 Graph Engine For Web Scale RDF Data. *PVLDB*, 6(4), 2013.
- 579 **41** Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-Aware Graph-Structured Analytics. In
580 *PPoPP*, 2015.