ASGraph: A Mutable Multi-Versioned Graph Container with High Analytical Performance

Michael Haubenschild†⋆
michael.m.haubenschild@oracle.com

Manuel Then⋆
then@in.tum.de

Sungpack Hong†
sungpack.hong@oracle.com

Hassan Chafi†
hassan.chafi@oracle.com

†Oracle Labs ⋆Technical University of Munich

ABSTRACT

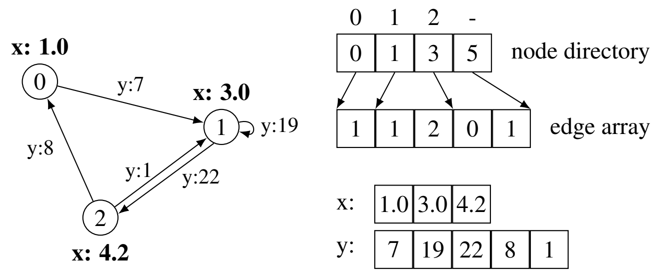
In the last years researchers and industry have become interested in the analysis of graphs to gain insights into social networks, road networks, and other data that is naturally organized as a set of connected entities. Many of these graphs are very large, some containing hundreds of billions of edges. Usually, graphs are stored in static or immutable representations. We propose ASGraph. ASGraph is a graph container that supports updates and multi-versioning while still providing high analytical performance in the order of magnitude of the predominant CSR. ASGraph stores temporal graphs with arbitrarily fine granularity. Additionally, it can optimize its internal layout for analytical queries at specific snapshots. We show that it has moderate runtime overhead between 7% - 98% for PageRank compared to CSR. Meanwhile it outperforms CSR both in runtime and memory consumption in scenarios where a graph is repeatedly updated between analysis. We designed ASGraph to support an update stream that can be applied concurrently to all analytical operations without blocking. In contrast to existing solutions for storing versioned graphs, its performance is independent of the number of stored snapshots.

CCS Concepts

•Information systems → Graph-based database models; Temporal data; Data structures; *Data streams;* **•Mathematics of computing →** *Graph algorithms;* **•Networks →** Online social networks;

1. INTRODUCTION

Graph analytics has become more and more popular over the last couple of years. Graphs are used to represent social networks, road maps, physical simulations as well as biomedical applications such as DNA splicing. The graphs encountered in these domains can become very large, e.g. a recent Facebook graph contains over 1.3 billion vertices representing users and well over 400 billion edges representing their friendships, likes, posts, etc.[5]. To analyze these large graphs one needs a data structure to physically store the graph



(a) Visual Representation **(b)** Corresponding CSR.

Figure 1: Small sample graph with three nodes, a floating point node property *x* and an integer edge property *y*

topology as well as associated properties of both vertices and edges. Furthermore, it should exploit modern hardware trends, e.g. parallel execution units and caches. Also, with today's servers having multiple terabytes of main memory, it is possible even for large graphs to fit in the main memory of a single machine. While most analysis today still focuses on static graphs, we predict that in the future there will be demand to gain insights on temporal graphs as most networks are not static[9]. People join a social network and new connections both between them as well as between the old members are established while some are removed again. Only an abstraction that models this fact can represent the underlying real-world application accurately. In the past the general trend when handling large amounts of data has been to use separate systems that are either optimized for updates or analytics. As a consequence, data analysts never operated on live data. But relational databases recently proved that both OLTP and OLAP on the same data can be a reality when the workload fits in main memory[8]. We promote the same approach for graph structured data by presenting a data structure that can handle updates efficiently while still providing high analytical performance. The state-of-the-art representation for in-memory graph analysis is the *Compressed Sparse Row* (CSR) format. Figure 1 shows the visual representation (1a) of a sample graph and the corresponding CSR (1b). The node directory stores offsets into one large edge array. For each node, the number of neighbors for a node `n` can be calculated as `nodeDir[n+1]-nodeDir[n]`.

All edges are stored densely in memory. When iterating over all neighbors of all nodes—a common task in graph algorithms—memory accesses are sequential in both the node directory and the edge array. This is a beneficial memory access pattern because it can easily be predicted and automatically prefetched by the CPU, thus, nearly all memory accesses hit the cache. In cases where nodes are accessed in a non-sequential order such as in a breadth-first search, CSR does not show cache-friendly behavior, but so will no other data structure. This is an inherent problem of graph analysis that comes down to efficient graph partitioning and is not further covered here. As for most data structures, there are trade-offs that need to be considered. For CSR the antagonistic goals are read-optimized performance and update friendliness. It provides very good performance for static graphs but inherently does not support efficient updates. While single neighbors could theoretically be exchanged in the edge array, arbitrary insertions or deletions are not possible. A common solution for this is to gather a number of updates in a delta store and apply them all in one batch[13]. However, this is still by no means optimal. The eventual update of the CSR requires copying the whole edge array, which takes $\mathcal{O}(n)$ time in the number of edges. Even if this operation is spread over multiple updates, it is still very expensive for large graphs. Other desired features such as support for streaming or versioning likewise cannot be implemented in CSR.

Our proposed data structure ASGraph (Analytical Snapshot Graph) solves this challenge by breaking up the edge list of CSR into multiple chunks which support efficient mutation and employing an append-only scheme for updates. We combine these two well-known concepts with a novel `createSnapshot` operation that rearranges the edge list fragments for high analytical performance while concurrent updates can still be applied without blocking.
Among other things, this allows for the following scenario: An initial graph is loaded into memory. Then, a PageRank is calculated on it. Simultaneously, updates are applied to the graph. When the user wishes to include the latest changes, they can request a new consistent snapshot and start the next query. Note that creating this snapshot naively would involve building up a whole new CSR from the old one and the delta, while for ASGraph this is an efficient operation that can naturally be parallelized.

In the following, we compare existing approaches for graph containers which all support a different subset of our desired features in Section 2. Then we go into the high level design choices of ASGraph in Section 3, followed by implementation details (Section 4) and an evaluation (Section 5). We conclude our work with directions for future work in Section 6.

2. RELATED WORK

STINGER[6] is a mutable graph container that can handle streaming updates which can be inserted in parallel. It uses fixed-size chained buckets to store edges. STINGER does not provide the concepts of consistent snapshots or multi-versioning. In fact, due to its loosely synchronized parallel updates that only maintain physical consistency of the data structure, a query might see the graph in a state that never existed. STINGER tightly integrates the concept of different edge types as each bucket contains only edges of one type. STINGER has a higher memory footprint than our approach as it stores more additional information for each edge such as an edge weight, a creation and a modification timestamp.
LLAMA[11] is a recent effort to extend CSR with version support. It provides consistent views on the graph and allows concurrent access to multiple snapshots. The graph including all snapshots can be stored to disk. Single updates are first buffered in a changeset and periodically applied as a new snapshot. The major drawback is that LLAMA's performance is deteriorating with the number of existing snapshots. More specifically, the access to newer snapshots gets more expensive. This is particularly severe as the most recent snapshot is usually queried most often. Therefore, an expensive compaction is regularly necessary. It merges old snapshots and basically comes down to applying a delta to a CSR as described above. Snapshot boundaries must be known at graph creation time. Also, too many snapshots push up memory consumption since a vertex indirection array is forked for each of them.

3. DESIGN CHOICES

Our goal is to design a graph container that provides analytical performance close to CSR while offering additional functionality. We want to be able to apply a continuous stream of updates to the graph which, for example, comes from an RDBMS or from a sensor network. The problem of synchronization is thereby reduced to a single update writer, which enables us to run it completely lock-free as we show later. Our intention is not to propose a whole new graph analytics platform, but one low-level building block of a hypothetic future system that allows for efficient temporal analysis of graphs. In contrast to LLAMA, we want to provide higher analytical performance while sacrificing the possibility to run queries on multiple snapshots in parallel. The reason for this choice is that we found that creating CSRs for a series of snapshots and running a PageRank sequentially on all of them is still faster than running a PageRank on each LLAMA snapshot in parallel. Furthermore, our graph algorithms are already parallelized, so inter-query parallelization most likely will not give better utilization, but rather hurts performance because of worse cache locality and TLB use since we encountered that most of the common graph algorithms we ran are not CPU-bound but limited by memory.
Note that we explicitly support node delete operations, which STINGER and LLAMA lack to do. We do not incorporate the concept of different edge types in the graph representation as STINGER does. If a user wants to distinguish between different edge types they can use an edge property in ASGraph.

4. ASGRAPH

Our approach is coarsely based on the physical layout of STINGER [6]. We use fixed-size buckets that can store a certain number of entries and which are chained in a single-linked list if a bucket overflows. There are four kinds of operations: node inserts (`NINS`), node deletions (`NDEL`), edge insertions (`EINS`) and edge deletions (`EDEL`). An operation consists of the triple

$$operation := (destination, timestamp, opType)$$

Note that we do not store the source of an edge explicitly, since a chain of buckets contains only operations with the same source node. New operations are always appended, since we need to preserve all information in order to create a snapshot at an arbitrary point in time. When a bucket is full, we allocate a new bucket and add it at the end of the chain. Using this append-only scheme and atomic primitives, we support multiple concurrent update operations even for the same source node in a lock-free manner. Figure 2 shows how the graph from Figure 1 is stored as an ASGraph. The bucket size of two is only used for illustration purposes. In our tests we found a bucket size of 12 to provide a good trade-off between *a)* memory overhead from unused slots and *b)* runtime overhead introduced by iterating over the list of buckets. Each bucket contains several header fields (see Figure 3). There is a pointer to the next bucket in the list and two counters `currentSlot` and `endOfSnapshot` which store the current number of slots used in the bucket and the last valid entry of the currently materialized snapshot. The latter is explained in more detail in Section 4.1. The rest of the bucket stores the edge destinations, timestamps and operation types. Physically, they are stored in a columnar manner. This allows for good cache utilization and thereby fast scans over neighbors during analytics.

In our implementation we combined two instances of ASGraph together, one storing the outgoing edges and one storing the incom-

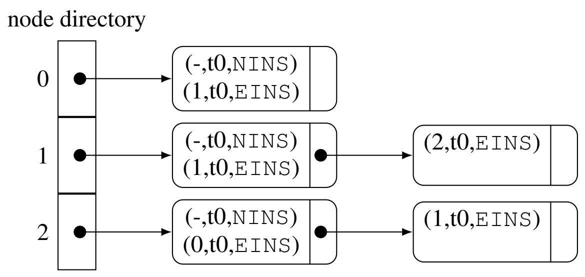


Figure 2: Overview of ASGraph with `bucketSize=2`. For the initial graph, we assume all timestamps are zero.

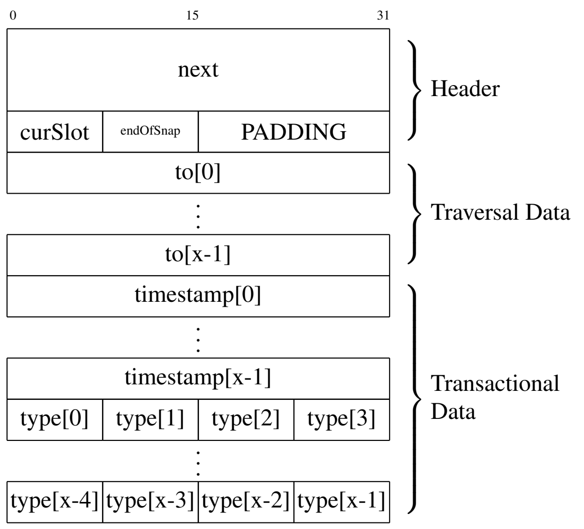


Figure 3: Physical memory layout of one bucket with `x` entries

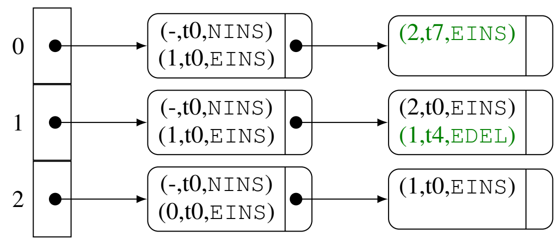
ing edges. For some algorithms and undirected graphs the latter can be omitted, cutting the memory consumption in half.

4.1 Snapshot Creation

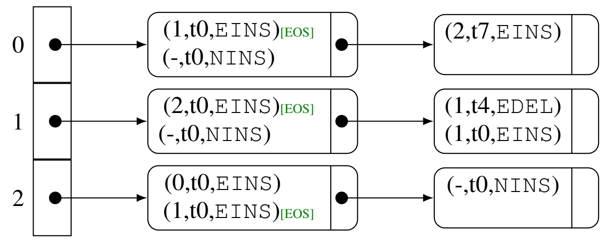
When a user requests to create a snapshot at a given timestamp t, for every node in the node directory the edge entries are reordered in a way that provides optimal performance for scans. We logically partition all entries in two categories. For all valid edges in the snapshot we put the corresponding `EINS` entry in the first partition. All other entries, particularly the `NINS`,`NDEL` and `EDEL` as well as the `EINS` entries that have a later timestamp or have been removed by `EDEL` are put in the second partition. The partitions are separated by a special `[EOS]` marker. Analytical queries never read past it. Concretely, a call to `createSnapshot(t)` performs the following steps for each neighbor list:

1. Scan through all operations and copy `EINS` operations to a list `CAND`, all other operations to another list `REST`. When an `EDEL` entry with a timestamp $t' \leq t$ is encountered, set `DELETIONS[dest] =` t' where `DEL` is a map from node-ids to timestamps (Lines 6-11).[1]
2. Iterate over each entry `e` in `CAND`. If `DEL[e.to] > e.timestamp`, this edge is already deleted at t and is moved to `REST`.
3. Rewrite the bucket entries with the concatenated lists `CAND` and `REST`. Put a `[EOS]` marker after the entries from `CAND`.

[1]Since this process never changes the relative order of edge deletion entries, after the scan the map will contain the highest deletion timestamp for that node.



(a) Before any snapshot was ever applied, timestamps in each bucket are in increasing order. Analytics is not possible until the first snapshot is created



(b) After the reordering, only operations up to the [EOS] marker can be seen by analytics.

Figure 4: Snapshot creation in ASGraph for timestamp t5.

During these steps a flag keeps track if the node is deleted at t. If so, steps 2 and 3 are skipped and a node deletion marker `[ND]` is put at the beginning of the first bucket in the chain. Note that steps 1 and 2 operate on temporary data structures while step 3 never writes operations to memory that is touched by the update stream. Thus, snapshot creation can run concurrently while operations are applied to ASGraph. We exploit parallelism in the snapshot creation phase by processing multiple entries in the node directory at once. An additional flag per node tracks if there have been changes since the last call to `createSnapshot` which enables us to skip these nodes. The complete procedure is shown (without the node deletion logic) in Listing 1.

```
1   for n : nodes
2     var CAND := List[operation] //edge candidates
3     var REST := List[operation] //remaining operations
4     var DEL := Map[nodeId->timestamp]
5     for op : n.operations
6       if(op.type == EINS && op.timestamp <= t)
7         CAND.append(op)
8       else
9         REST.append(op)
10        if(op.type == EDEL)
11          DEL[op.to] = op.timestamp
12    for c : CAND
13      if(c.timestamp <= DEL[c.to])
14        //move c to REST
15        var tmp = CAND.remove(c)
16        REST.append(tmp)
17    //OPTIONALLY: sort CAND at this point
18    //Replace bucket content with reordered operations
19    n.operations <- concat(CAND,[EOS],REST)
```

Listing 1: The `createSnapshot(t)` method

To illustrate snapshot creation with an example, we first apply the following two operations to our example graph: The edge from node 1 to itself is removed at t4 and an edge from node 0 to 2 is inserted at t7. This gives us the ASGraph shown in Figure 4a. Calling `createSnapshot(t5)` transforms it to the state shown in Figure 4b which now can be used to run analytical queries.

An alternative to this approach would be to materialize valid neighbors on-the-fly during each neighbor iteration. However, since most algorithms iterate multiple times over a neighbor list and more than one analysis might run on a snapshot this is potentially slower. However, slow path access is possible as outlined at the end of the next section.

4.2 Analytics

Analytical algorithms access ASGraph through a simple API. It is very similar to that of CSR and is oblivious to the temporal character of the stored graph as it always sees a consistent view of the graph at a certain timestamp. There are the usual operations to get the total number of nodes and edges for the current snapshot, getting the number of outgoing edges for an individual node and getting the neighbors of a node. The last one is where algorithms usually spend most of their time and thus it needs special attention. While CSR supports random access to all neighbors of a node, ASGraph cannot implement this efficiently, because for each access it needs to traverse the whole bucket list. We found however that most algorithms do not need random access. Thus, we implemented an `iterateNeighbors()` method that takes a callback which is called for each neighbor. Note that due to the columnar layout of the bucket entries this iterator never needs to load memory into cache that contains the timestamps or `opTypes`. Furthermore, we can prefetch the next bucket in the chain before processing the current one. Both techniques combined bring us close to the cache efficiency of CSR which is crucial for ASGraph's performance.
For point queries it might be too expensive to create a snapshot for the whole graph in order to access only a few buckets lists. Therefore we propose a slow path that materializes valid neighbors on-the-fly in temporary data structures. This also allows fully concurrent analyses of different snapshots with the restriction that only one of them can be accessed with high performance. We leave the investigation of the slow path for future work.

4.3 Node Array

In our current implementation we use a fixed-size array of bucket pointers, one for each node. This has the drawback that at creation time of the ASGraph data structure, the user has to specify the maximum number of nodes that can be stored. In many scenarios the growth rate of the graph can be estimated, thus allowing to choose a proper upper bound of nodes for the required timeframe. However if this cannot be anticipated, ASGraph can be extended to use a dynamic array that can grow. For large graphs one might not want to copy the whole array on resizing, so an alternative is to use an extendable array, either with fixed-size or exponentially growing segments. This introduces the cost of an additional level of indirection for each node access which results in additional cache misses. However since the first level indirection array is generally small, it should fit into the cache of modern CPUs.

4.4 Properties

ASGraph supports mutable node and edge properties. While for immutable graphs both can be implemented as arrays, in our multi-versioned use case they behave differently, as described below. Our basic approach for both is the same as for storing the graph topology, which is having a node directory where each entry points to a chain of buckets. Node property buckets contain entries of the type $nodePropOp := (value, timestamp)$. The `createSnapshot(t)` operation for a node property consists of looking for the entry with the highest timestamp less or equal to `t` and copying that into a backing array at the index that corresponds to this vertex in the node directory. That way, entries in the buckets never have to be reordered and the access time for a property is $\mathcal{O}(1)$. We again exploit the advantages of columnar layout to maximize cache utilization during snapshot creation. While scanning for the correct timestamp, we do not have to load memory that corresponds to property values. In the CSR representation, the layout of an edge property can mirror that of the edge list. In the mutable case, property updates can occur independent of topology updates, so the entries for a property do in general not correspond to the entries for the edges. Therefore, we must associate an edge property update with its corresponding edge. We do so by including the destination edge in the edge property entry, as follows: $edgePropOp := (destination, value, timestamp)$. To access a property for a given edge `e :=(from,to)`, we skip through the bucket list of `from` until we find the correct bucket and search it for an entry with destination `to`. This leads to worst case access times in the order of the number of neighbors a node has, which can become very large, especially for graphs with skewed degree distribution. Therefore we adapt our algorithms to use an optimization if possible: Instead of iterating the neighbors in the ASGraph itself and access the property for each neighbor it can iterate the property directly if the algorithm only needs to access one property. If it needs to access multiple properties, at least one of them can be accessed with this optimization. In Section 5.1 we compare the performance for running the Bellman-Ford algorithm once with this optimization and once without it.

Graph	# vertices	# edges	Source
SanFrancisco	59.813	149.715	subset of [10]
LiveJournal	4.847.571	68.993.773	[1]
LDBC-300	1.253.978	136.219.368	[7, 4]
Twitter	41.652.230	1.468.365.182	[2, 3]
WebGraph	77.741.046	2.965.197.340	[2, 3]

Table 1: Datasets used in our benchmarks

5. EVALUATION

We ran our benchmarks on datasets with different characteristics and sizes (Table 1). Our test machine is a dual socket server computer equipped with Intel Xeon E5-2699 v3 18-core CPUs and 378 GB of main memory. Each core has two Hyper-Threads, resulting in a total of 72 hardware threads.

5.1 Algorithm Comparison

To get a feeling of the performance of ASGraph compared to CSR we evaluate four different algorithms that cover a broad spectrum of graph access patterns (see Figure 5).

PageRank

PageRank is a friendly algorithm in terms of memory access. On the other hand, its low computational complexity means that performance is mostly restrained by memory accesses. Depending on the dataset, ASGraph performs 7% to 98% slower than CSR which stems from the additional instructions in its iterator and the cache miss when a new bucket is accessed.

Bellman-Ford

Our Bellman-Ford experiments use an edge property of double type that is used as the distance weight. For all datasets we achieve equal or better performance than CSR. This is due to the optimization discussed in 4.4 where we iterate over the property directly, since it already stores the information about edge destinations alongside the actual property values. Since the algorithm always accesses the property together with the edge information it has high cache

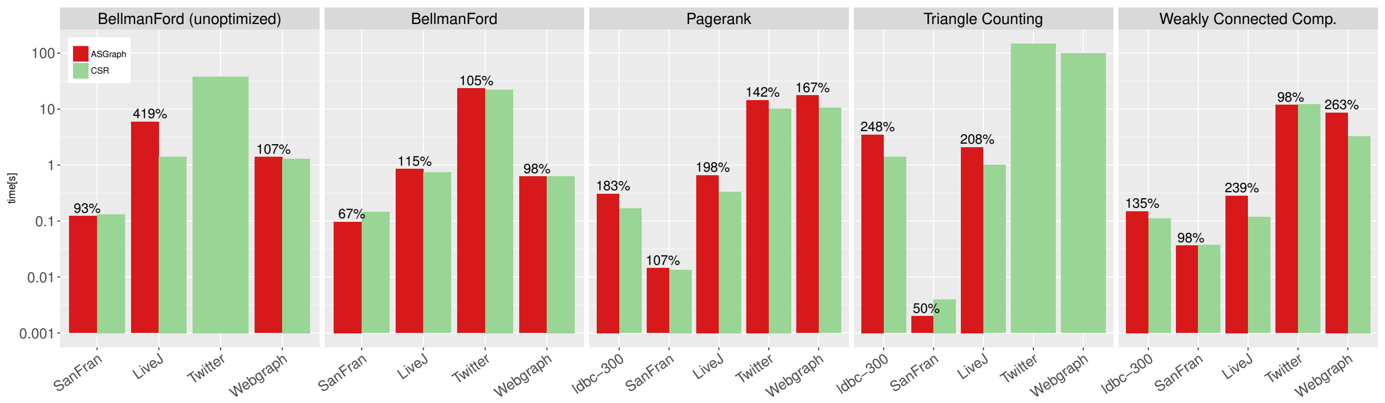


Figure 5: Comparison of different algorithms between ASGraph and CSR. Missing bars are runs that did not finish in a reasonable time

utilization. This optimization could also be applied to CSR by storing the property value next to each edge entry. Compared to the other algorithms, Bellman-Ford's inner loop is also more complex which alleviates the overhead of our iterator.

At the other end of the scale is the Bellman-Ford algorithm without that optimization. There ASGraph has to do a normal lookup of each property value for each edge, which involves traversing the property bucket chain and searching for the correct entry in the qualifying bucket. For skewed datasets with a long tail such as LiveJ and Twitter, this effect leads to an explosion in runtime (419% for LiveJ, Twitter did not finish in a reasonable timeframe).

Weakly Connected Components

Computing the number and affinity of weakly connected components shows no special access pattern that is worth mentioning. It behaves comparable to PageRank and likewise demonstrates the overhead of the iterator and the cache misses which results in runtimes up to 139% slower than CSR, while for some datasets it is as fast as CSR.

Triangle Counting

Our CSR implementation of triangle counting is highly tuned as described in [12]. Therefore we expect relative low performance for ASGraph compared to CSR as most of the optimizations are not applicable to ASGraph since they require random accesses to the neighbor list of a node.The benchmarks confirm this assumption. We leave tuning ASGraph's common neighbor iterator for future work, but believe that it inherently performs bad in this algorithm because ASGraph cannot do random access to neighbors due to its bucket layout, so we have a $\mathcal{O}(n)$ runtime compared to $\mathcal{O}(\log n)$ for finding the first eligible entry in a neighbor list. For this reason, triangle counting marks off another corner case for which ASGraph performs particularly bad. We didn't obtain runtimes for Twitter and Webgraph as, due to the long tail and the graph size, they did not finish in time.

5.2 Scaling

We found no significant difference between the scaling behavior between ASGraph and CSR, which is not surprising as both employ no synchronization for reading the graph.

5.3 Complex Scenario

While ASGraph is slower most of the time if we look at a single analytical query run in isolation (Section 5.1), it beats CSR for a scenario that is even a bit more complex and which comes much closer

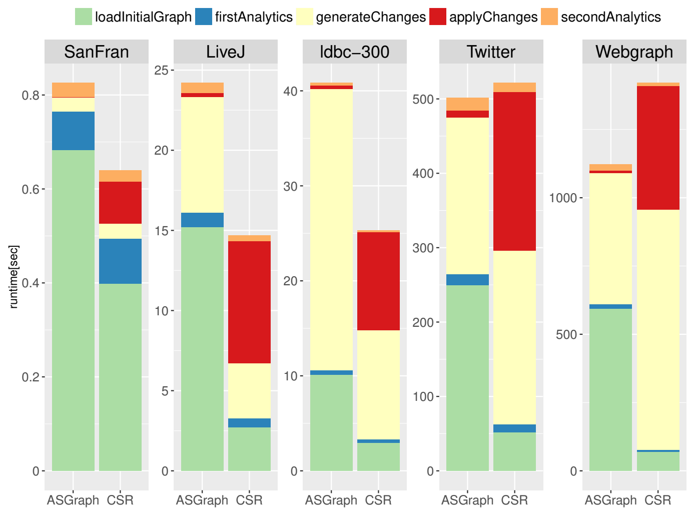


Figure 6: Runtime for a more complex scenario including breakdown in the different phases

to a real use-case. We look at the total runtime for 1) loading a graph, 2) running an analysis on it (PageRank), 3) generating changes, 4) applying those changes and finally 5) running a second analysis on the changed graph. The changeset we use in this benchmark randomly adds 10% of the original graph's nodes and 10% of the edges. Figure 6 shows our results for this scenario and also breaks up each run into its single steps. While for smaller graphs runtime is dominated by the loading and the time to run the algorithm, for the larger graphs generating and applying the changes dominates overall runtime. Generating the changes for CSR means collecting them in a delta structure while for ASGraph they are appended to the bucket list directly as explained in Section 4. Applying the changes for CSR involves generating a whole new CSR while for ASGraph it comes down to the `createSnapshot()` method explained in Section 4.1. As Figure 6 shows, ASGraph is faster than CSR for this scenario for large graphs with the size of Twitter or Webgraph. The lead increases if steps 3) to 5) are executed repeatedly which mitigates the role of the initial loading time and makes ASGraph also the faster alternative on the long run.

5.4 Memory Consumption

Since ASGraph keeps track of the complete operation history, it has to store the timestamps and opTypes for each operation which

	Graph	Topology	Property	Total
ASGraph	Twitter	34 GB	29 GB	63 GB
CSR	Twitter	23 GB	11 GB	34 GB
ASGraph	Webgraph	68 GB	57 GB	125 GB
CSR	Webgraph	47 GB	22 GB	69 GB

Table 2: Memory consumption of ASGraph vs. CSR

introduces a non-negligible memory overhead. Table 2 quantifies this. But as soon as we look at a scenario like the one from Section 5.3, the tide turns in favor of ASGraph. For CSR, we need to materialize the whole graph multiple times, or at least twice during the merge of the old CSR with the change set. ASGraph on the other hand inherently stores all the necessary information for all different versions, thus it has a smaller memory consumption when the mutations to the graph do not exceed a certain percentage of the original graph.

6. CONCLUSIONS

We presented ASGraph, a mutable graph container that shows performance comparable to CSR for running single analytical queries and beats it in overall runtime for more complex scenarios. ASGraph's performance is independent of the number of stored snapshots, thus performance does not deteriorate even when ASGraph is used to store thousands of snapshots. Furthermore, it offers support for a concurrent update stream and a slow path access to arbitrary versions which do not have to be materialized. ASGraph supports node and edge properties. While node properties can be accessed without a performance penalty, lookup of edge properties can become expensive in the current scheme.

6.1 Future Work

Our next steps include evaluating the slow path access to neighbor lists. For a practical implementation we further need to investigate mapping of vertex keys to internal ids. This is a non-trivial task as most algorithm implementations expect a dense range of indices, but for ASGraph the internal representation can contain deleted nodes, so we need a second level of mapping to fill these holes. Our property design currently only allows fixed-width values, but there are use-cases where one would like to densely store variable-length properties. To store these in fixed-size buckets we need to adopt an approach similar to slotted pages from a RDBMS. While our current implementation can handle more than 1 million operations per second, eventually its performance will deteriorate because each operation has to traverse the whole bucket chain to find the correct slot. By reversing the chain we could solve that problem at the cost of a more complex iterator and snapshot logic. This would also introduce the possibility to optimize snapshot creation for the common case where a user wants to incorporate the latest changes. Another challenge is that the average length of the bucket chain and thereby the memory consumption grows with the number of updates. So a user may wish to compact the graph by removing older entries, sacrificing the possibility to restore the graph to older snapshots. Repeated insert and delete operations and property update operations up to that time are thereby collapsed into one.

Once our system is mature enough, we want to compare it against others in the Graphalytics Benchmark [4]. We claimed before, that for most workloads a single shared memory machine is sufficient, but there are graphs in the size of multiple terabytes for which we need to develop a distributed version of ASGraph. We expect the same challenges for this as there are for distributed CSR.

7. REFERENCES

[1] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 44–54, 2006.

[2] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 587–596.

[3] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 595–602, 2004.

[4] M. Capota, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. A. Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*, pages 7:1–7:6, 2015.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[6] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader. STINGER: high performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, pages 1–5, 2012.

[7] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 619–630, 2015.

[8] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the hybrid OLTP & OLAP main-memory database system hyper. *IEEE Data Eng. Bull.*, 36(2):41–47, 2013.

[9] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 177–187, 2005.

[10] F. Li. www.cs.utah.edu/~lifeifei/SpatialDataset.htm, 2005.

[11] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: efficient graph analytics using large multiversioned arrays. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 363–374, 2015.

[12] M. Sevenich, S. Hong, A. Welc, and H. Chafi. Fast in-memory triangle listing for large real-world graphs. In *Proceedings of the 8th Workshop on Social Network Mining and Analysis, New York, NY, USA, August 24, 2014*, pages 2:1–2:9, 2014.

[13] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 731–742, 2012.