Sortledton: a universal, transactional graph data structure

Submitted for VLDB 2022:
Per Fuchs, Domagoj Margan and Jana Giceva
Graph use-cases are diverse and dynamic

- **diverse**: use multiple graph workload categories, e.g. analytics, traversals and graph pattern matching (GPM)
- **dynamic**: requires insertions and deletions

**Examples:**
- Alibaba: analytics and traversals in anti-fraud-pipelines [VLDB’20]
- Twitter: use of traversals and GPM for recommendations [VLDB’14, ’15]
- Both require up to 2 million edge insertions per second
New challenges in graph data structures: striking a good trade-off for all workloads

References: Livegraph [VLDB’20], Graphone [FAST’19], LLama [ICDE’15], Stinger, Teseo [VLDB’21]
New challenges in graph data structures: striking a good trade-off for all workloads

References: Livegraph [VLDB’20], Graphone [FAST’19], LLama [ICDE’15], Stinger, Teseo [VLDB’21]
New challenges in graph data structures: striking a good trade-off for all workloads

References: Livegraph [VLDB’20], GraphOne [FAST’19], LLama [ICDE’15], Stinger, Teseo [VLDB’21]
New challenges in graph data structures: striking a good trade-off for all workloads

References: Livegraph [VLDB’20], GraphOne [FAST’19], LLama [ICDE’15], Stinger, Teseo [VLDB’21]
New challenges in graph data structures: striking a good trade-off for all workloads

References: Livegraph [VLDB’20], GraphOne [FAST’19], LLama [ICDE’15], Stinger, Teseo [VLDB’21]
Contributions

1. Comparison of fundamental graph data structure designs
2. Analysis of access patterns in graph workloads
3. A simple dynamic data structure design with memory consumption (2x CSR) and analytical performance (1.2x CSR)

4. Design of a graph specialized, serializable MVCC system (in the paper)
Comparison of fundamental graph data structures

CSR-like

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5003</td>
<td>...</td>
</tr>
</tbody>
</table>

Vertex Index stores
Offsets into neighbourhoods

+ sequential vertex access
Comparison of fundamental graph data structures

- sequential vertex access
- independent neighbourhoods
- cheap index maintenance
- reuse of existing data structures
Graph access patterns

Inner PageRank loop

1. \textbf{for } v \in V \textbf{ do}
2. \hspace{1em} incoming $\leftarrow$ 0
3. \hspace{1em} \textbf{for } e \in v.\text{neighbours} \textbf{ do}
4. \hspace{2em} \hspace{1em} incoming $\leftarrow$ incoming + contrib[e]
5. \hspace{1em} scores[v] $\leftarrow$ incoming
Graph access patterns

Inner PageRank loop

1. Sequential Vertex Access

1 for \( v \in V \) do
2 \hspace{1em} \text{incoming} \leftarrow 0
3 \hspace{1em} \text{for} \ e \in v.\text{neighbours} \ \text{do}
4 \hspace{2em} \text{incoming} \leftarrow \text{incoming} + \text{contrib}[e]
5 \hspace{2em} \text{scores}[v] \leftarrow \text{incoming}
Graph access patterns

Inner PageRank loop

1. Sequential Vertex Access

2. Sequential Neighbourhood Access

1. for \( v \in V \) do
2.     incoming \( \leftarrow 0 \)
3. for \( e \in v.\text{neighbours} \) do
4.     incoming \( \leftarrow \) incoming + contrib\([e]\)
5.     scores\([v]\) \( \leftarrow \) incoming
Graph access patterns

Inner PageRank loop

1. Sequential Vertex Access

2. Sequential Neighbourhood Access

3. Random Algorithmic-Specific Access

1. for \( v \in V \) do
2. incoming \( \leftarrow 0 \)
3. for \( e \in v\.neighbours \) do
4. incoming \( \leftarrow \) incoming + contrib[\( e \)]
5. scores[\( v \)] \( \leftarrow \) incoming
Graph access patterns

Inner PageRank loop

1. Sequential Vertex Access
   1. for $v \in V$ do
   2. incoming $\leftarrow 0$
   3. for $e \in v$.neighbours do
   4. incoming $\leftarrow$ incoming + contrib[$e$]
   5. scores[$v$] $\leftarrow$ incoming

2. Sequential Neighbourhood Access

3. Random Algorithmic-Specific Access

Optimizing for 2 and 3 has higher impact because mostly $|E| / |V| > 30$. 
Sortledton: simple and sorted

- Optimal for scanning with 512 edges per block
- Fast updates by splitting and merging
- Sorted for intersections

Optimal for random vertex access.

Adjacency index

Unrolled skip list for hub vertices

- 0x...
- 0x...
- 5000
- 0x...
- 0x...
- 0x...
- 0x...
- ...

- Adj. set size
- Adj. set pointer
- Forward pointer
- Skip pointer
Sortledton: simple and sorted

- Optimal for scanning with 512 edges per block
- Fast updates by splitting and merging
- Sorted for intersections

Optimal for random vertex access.
Evaluation

- Update performance: how many updates can the data structures process?
  - Challenge is to find the existing edges
- Graphalytics Benchmark: what is the slowdown for different workload categories compared to a CSR?
- Not in the presentation:
  - Mixing updates and deletions to expose aging effects
  - Memory consumption over aging
Update performance - power law graphs
Update performance - uniform graphs

![Graph showing update performance for different datasets and algorithms.](image)

- **x-axis**: Dataset (friendster, dota, g500-22, g500-24, g500-26, uni-24, uni-26)
- **y-axis**: Million edges per second
- **Legend**: Stinger, Llama, Livegraph, Teseo, Sortledton
Graphalytics benchmark performance

Graph Pattern Matching

Graph Analytics

Graph Traversals

Dataset: Graph500-24
Graphalytics benchmark performance

Graph Pattern Matching
Graph Analytics
Graph Traversals

Algorithm and CSR runtime

Dataset: Graph500-24
Conclusions

- adjacency list-like designs are simpler than CSR-like designs while showing equal performance
- we need to store neighbourhoods as sets to support GPM, updates, deletions and consistency
Memory Footprint

Graph500-24

Memory Consumption [GB]

Progress

18.3 GB = 2.2x CSR
Graphalytics Benchmark Performance

![Graphalytics Benchmark Performance Chart]

- **LCC 5.6m**
- **PR 10.62s**
- **WCC 3.00s**
- **CDLP 2m**
- **SSSP 26.43s**
- **BFS 0.48s**

Legend:
- GraphOne
- Livegraph
- Teseo sparse
- Teseo dense
- Sortedton

Slowdown vs. Algorithm and CSR runtime.
Dense vertex identifier for algorithmic specific access

- we translate arbitrary vertex identifier \{0, 5, 1000, \ldots\} on insertion into the dense domain \([0, 1, 2, \ldots]\) for better analytical performance
Sorted Blocks for Sequential Edge Access with intersections

Sorted blocks for neighbourhoods

normalized against

Sorted Vector for neighbourhoods (optimal, static)

Graphalytics Algorithms

Slowdown

Block Size [edges]
Optimizing for Sequential Vertex Access

Vector-based Adjacency List

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x</td>
<td>6x</td>
<td>6x</td>
<td>6x</td>
<td>...</td>
</tr>
</tbody>
</table>

Vertex IDs
Pointer to neighbourhoods

normalized against

CSR

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5003</td>
<td>...</td>
</tr>
</tbody>
</table>

Vertex IDs
Offsets into neighbourhoods

Graphalytics Algorithms

- BFS
- PageRank
- SSSP
- WCC

Slowdown

Dataset

- friendster
- dota
- g500-24
- uni-24

Normalized against
Which Vertex ID Domain to Store for Random Algorithmic-Specific Access?

sparse domain: \{0, 3, 1000, 1001, \ldots\}

normalized against

dense domain: [0, 1, 2, 3, \ldots]
Transactions in Graphs

Fall into two categories (mostly):

1. long-running, read-only transactions
   a. between seconds and multiple minutes
   b. e.g. PageRank (analytics), SSSP (traversals), triangle counting (GPM)

2. simple write-only transactions,
   a. with a-priori known read- and write-sets
   b. e.g. edge insertions
Transactions on Graphs (cont.)

- versioned records are 8 Bytes or less
- requires low overhead per version
  - we expect mostly 1 or 2 versions for each record
- our overhead is 0 for single versions, 8 Bytes for 2 versions and 16 Byte per additional version
Requirements for Concurrency Control

1. decouple reads from writes → use MVCC
2. high throughput for simple writes with known write-set → conservative two phase locking with fixed locking order
Transactions

- Example: inserting the undirected edge \((a, b)\) with \(b < a\)
  1. Acquire locks for vertex \(b\) then \(a\)
  2. Check if \(a\) and \(b\) exists, ensure neither \((a, b)\) nor \((b, a)\) exist
  3. Draw commit timestamp
  4. Insert \((a, b)\) and \((b, a)\)
  5. Release locks

Avoids overheads of other protocols, e.g. drawing two timestamps, deadlock detection, and rollback handling.
Real world graph workloads are diverse

Top 5 most common graph workloads according to a survey [VLDB, 2017]

<table>
<thead>
<tr>
<th>Computation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding Connected Components</td>
<td>55</td>
</tr>
<tr>
<td>Neighborhood Queries (e.g., finding 2-degree neighbors of a vertex)</td>
<td>51</td>
</tr>
<tr>
<td>Finding Short / Shortest Paths</td>
<td>43</td>
</tr>
<tr>
<td>Subgraph Matching (e.g., finding all diamond patterns, SPARQL)</td>
<td>33</td>
</tr>
<tr>
<td>Ranking &amp; Centrality Scores (e.g., PageRank, Betweenness Centrality)</td>
<td>32</td>
</tr>
</tbody>
</table>

Workload categories [arxiv, 2019]

- analytical
- neighborhood
- traversals
- graph pattern matching
  - used in analytical and transactional settings
Aging throughput over time

- Teseo and Sortledton provide high, stable throughput
- Livegraph has low, stable throughput
- LLLama throughput diminishes over time
- Graphone has severe issues with edge removals
## Properties of existing approaches

<table>
<thead>
<tr>
<th></th>
<th>GraphOne</th>
<th>LLama</th>
<th>Stinger</th>
<th>Livegraph</th>
<th>Teseo</th>
<th>Us</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intersections</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Sorted</td>
<td>Sorted</td>
</tr>
<tr>
<td><strong>Sequential Scans</strong></td>
<td>blocks</td>
<td>blocks</td>
<td>blocks</td>
<td>vector</td>
<td>blocks</td>
<td>blocks</td>
</tr>
<tr>
<td><strong>Skewed insertions</strong></td>
<td>O(D)</td>
<td>N/A</td>
<td>O(D)</td>
<td>O(D)</td>
<td>O(log D)</td>
<td>O(log D)</td>
</tr>
<tr>
<td><strong>Vertex identifiers</strong></td>
<td>dense</td>
<td>user needs to provide dense vertices</td>
<td>dense (no deletions)</td>
<td>user needs to provide dense vertices</td>
<td>sparse</td>
<td>concurrent sparse to dense translation</td>
</tr>
<tr>
<td><strong>Edge Contiguous</strong></td>
<td>no</td>
<td>partially</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>