TUM

Sortledton: a universal, transactional graph data structure

Submitted for VLDB 2022:

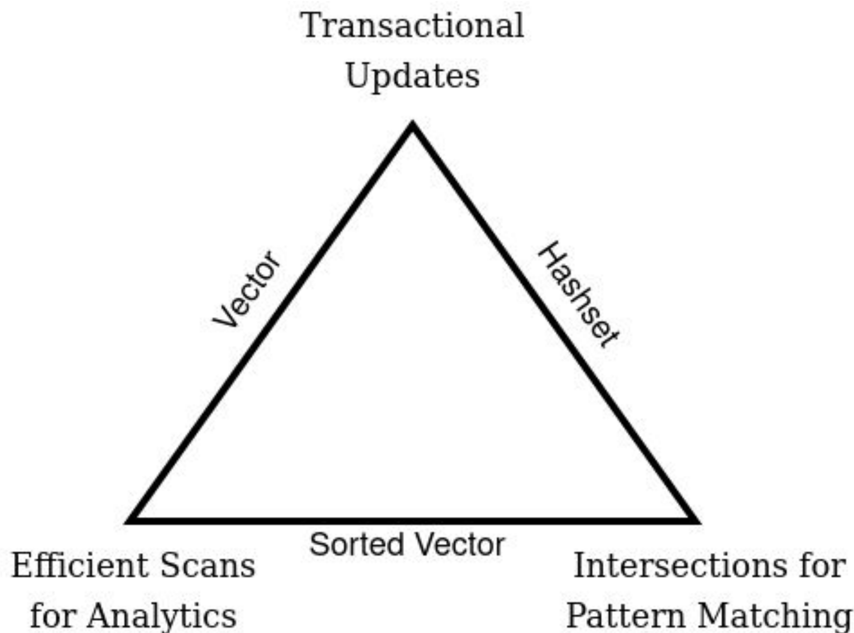
Per Fuchs, Domagoj Margan and Jana Giceva

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

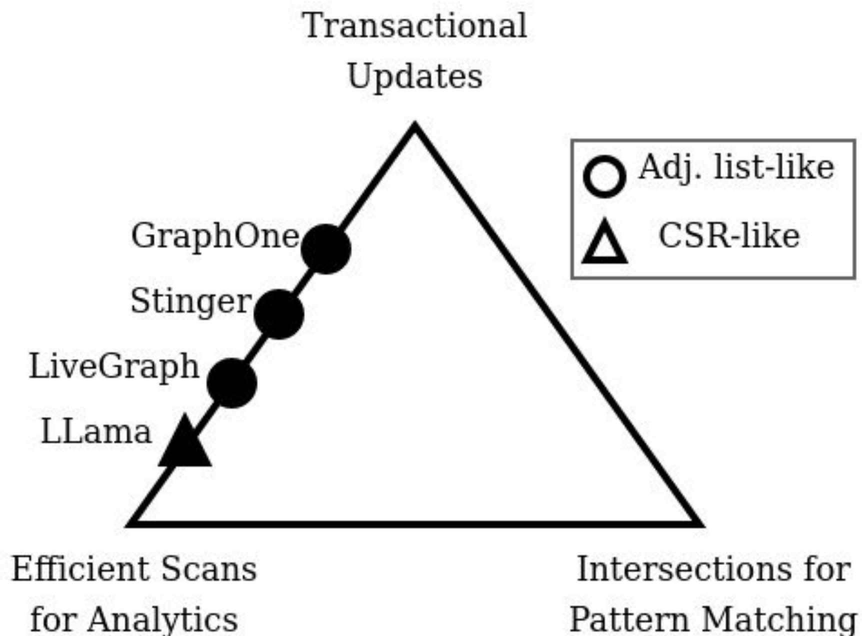
- Examples:
  - Alibaba: analytics and traversals in anti-fraud-pipelines [VLDB’20]
  - Twitter: use of traversals and GPM for recommendations [VLDB’14, ’15]
  - Both require up to 2 million edge insertions per second

New challenges in graph data structures: striking a good trade-off for all workloads



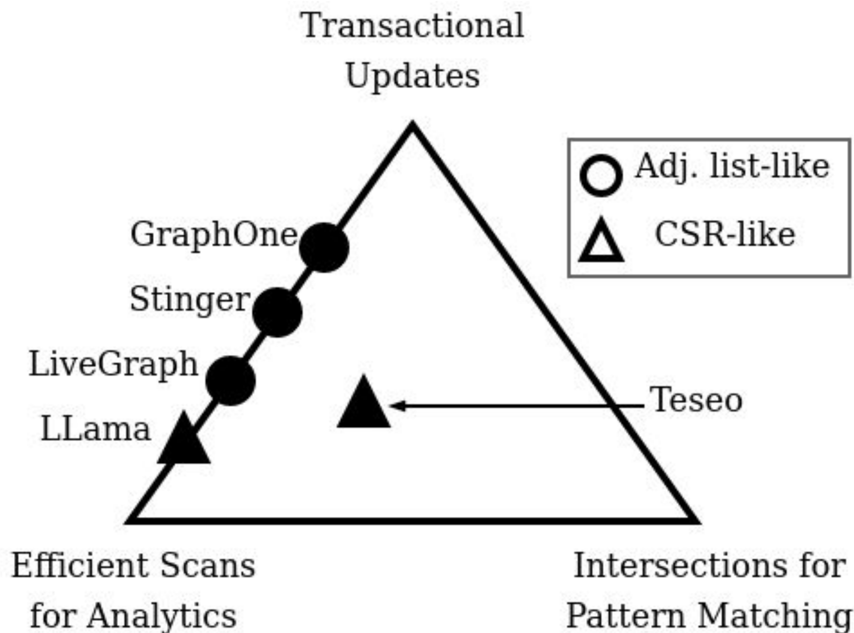
References: Livegraph [VLDB’20], Graphone [FAST’19], LLama [ICDE’15], Stinger, Teseo [VLDB’21]

New challenges in graph data structures: striking a good trade-off for all workloads



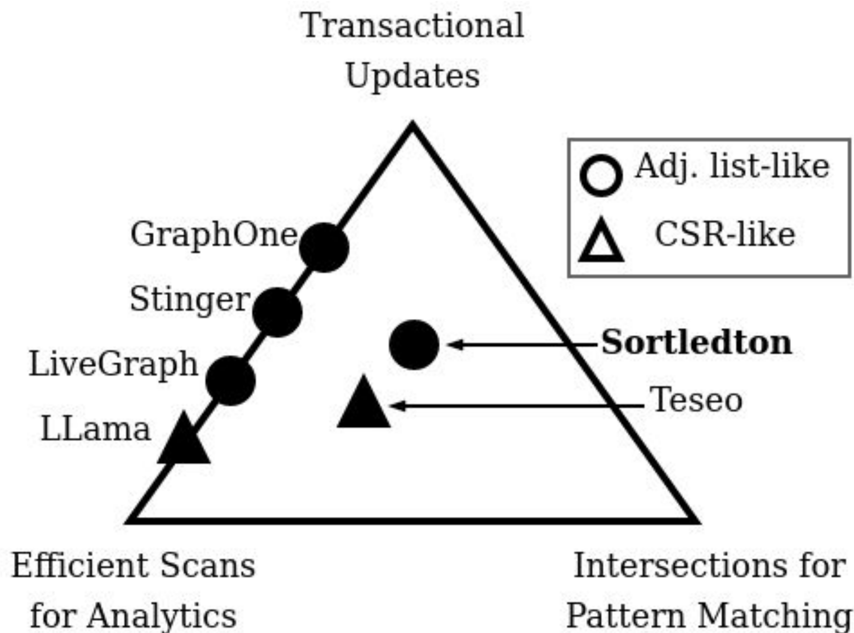
References: Livegraph [VLDB’20], Graphone [FAST’19], LLama [ICDE’15], Stinger, Teseo [VLDB’21]

New challenges in graph data structures: striking a good trade-off for all workloads



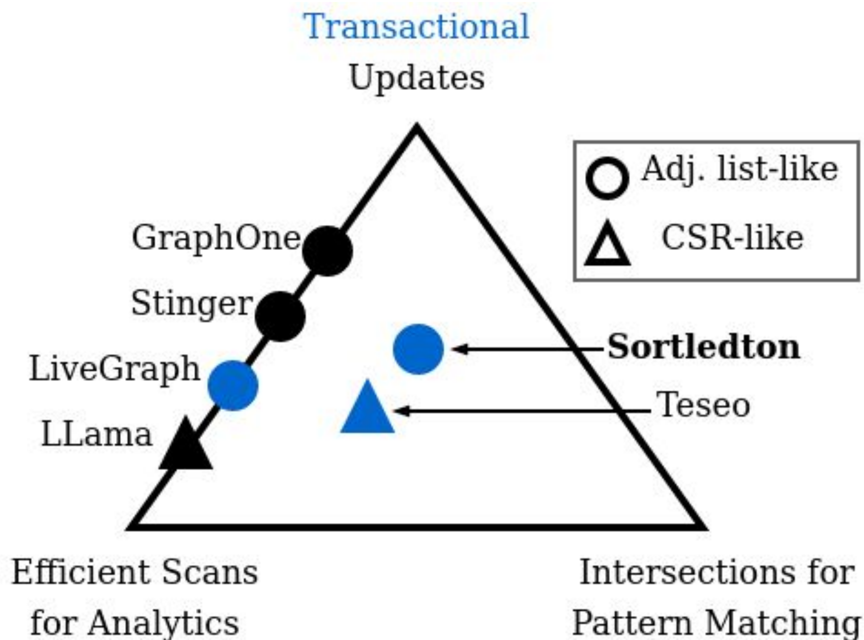
References: Livegraph [VLDB'20], Graphone [FAST'19], LLama [ICDE'15], Stinger, Teseo [VLDB'21]

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



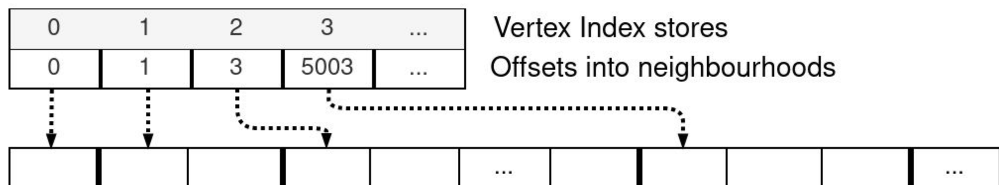
References: Livegraph [VLDB’20], Graphone [FAST’19], LLama [ICDE’15], Stinger, Teseo [VLDB’21]

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)

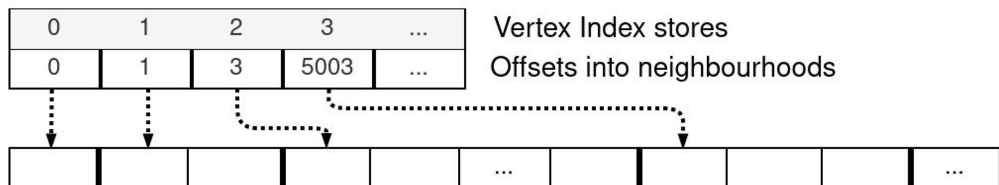
4. Design of a graph specialized, serializable MVCC system (in the paper)

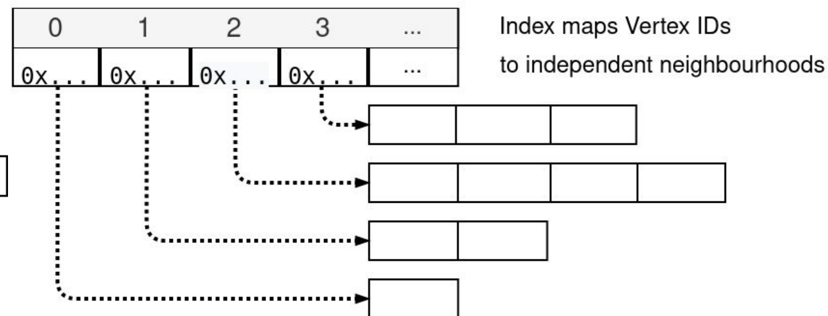
Comparison of fundamental graph data structures



- + 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 for v ∈ V do
2     incoming ← 0
3     for e ∈ v.neighbours do
4         incoming ← incoming + contrib[e]
5     scores[v] ← incoming
```

Graph access patterns

Inner PageRank loop

1. Sequential Vertex Access

```
1 for v ∈ V do
2 │   incoming ← 0
3 │   for e ∈ v.neighbours do
4 │   │   incoming ← incoming + contrib[e]
5 │   scores[v] ← incoming
```

Graph access patterns

Inner PageRank loop

1. Sequential Vertex Access

2. Sequential Neighbourhood Access

```
1 for v ∈ V do
2 |   incoming ← 0
3 |   for e ∈ v.neighbours do
4 |   |   incoming ← incoming + contrib[e]
5 |   scores[v] ← incoming
```

Graph access patterns

Inner PageRank loop

1. **Sequential Vertex Access**
2. **Sequential Neighbourhood Access**
3. **Random Algorithmic-Specific Access**

```
1 for v ∈ V do
2     incoming ← 0
3     for e ∈ v.neighbours do
4         incoming ← incoming + contrib[e]
5     scores[v] ← incoming
```

Graph access patterns

Inner PageRank loop

1. Sequential Vertex Access

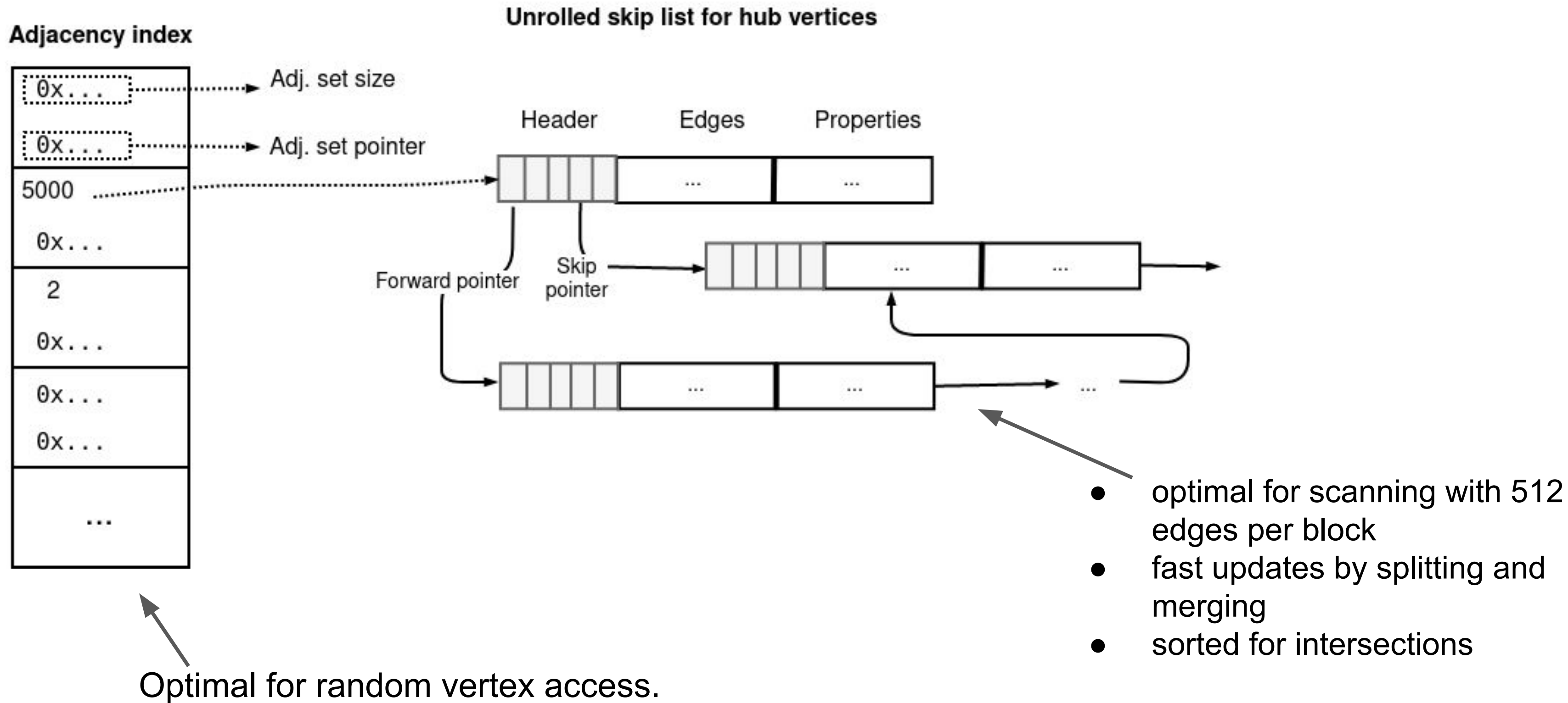
2. Sequential Neighbourhood Access

3. Random Algorithmic-Specific Access

```
1 for v ∈ V do
2 |   incoming ← 0
3 |   for e ∈ v.neighbours do
4 |   |   incoming ← incoming + contrib[e]
5 |   scores[v] ← incoming
```

Optimizing for 2 and 3 has higher impact because mostly |E| / |V| > 30.

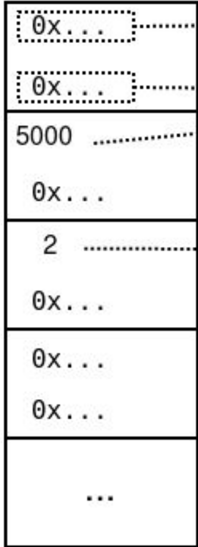
Sortledton: simple and sorted

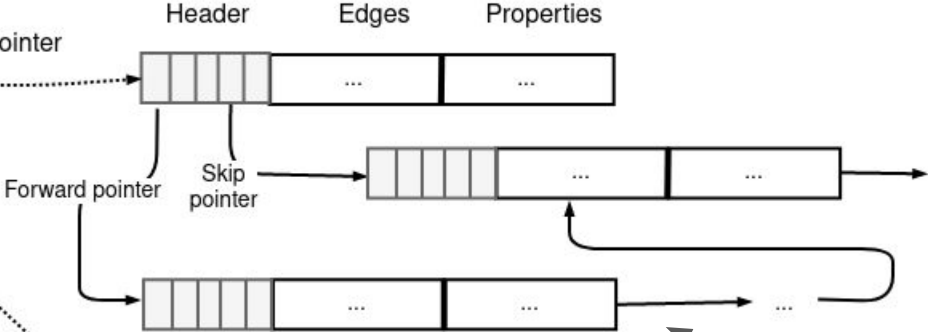


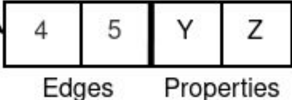
Optimal for random vertex access.

- optimal for scanning with 512 edges per block
- fast updates by splitting and merging
- sorted for intersections

Sortledton: simple and sorted







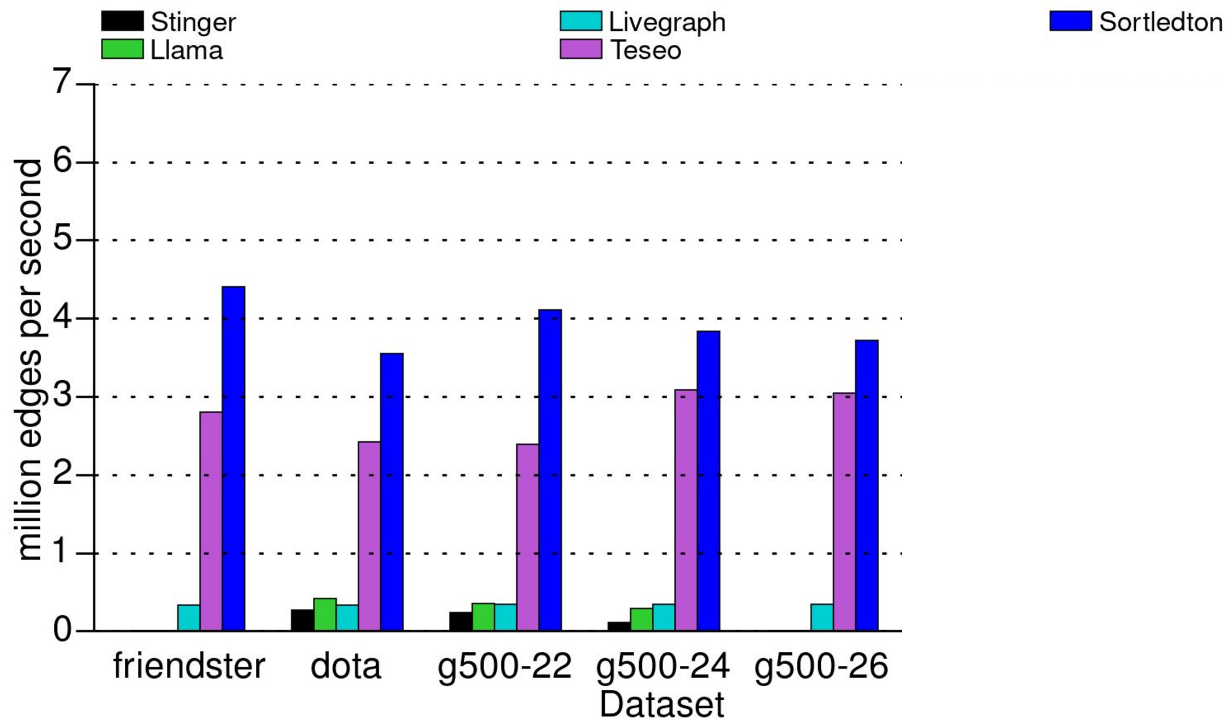
- optimal for scanning with 512 edges per block
- fast updates by splitting and merging
- sorted for intersections

Optimal for random vertex access.

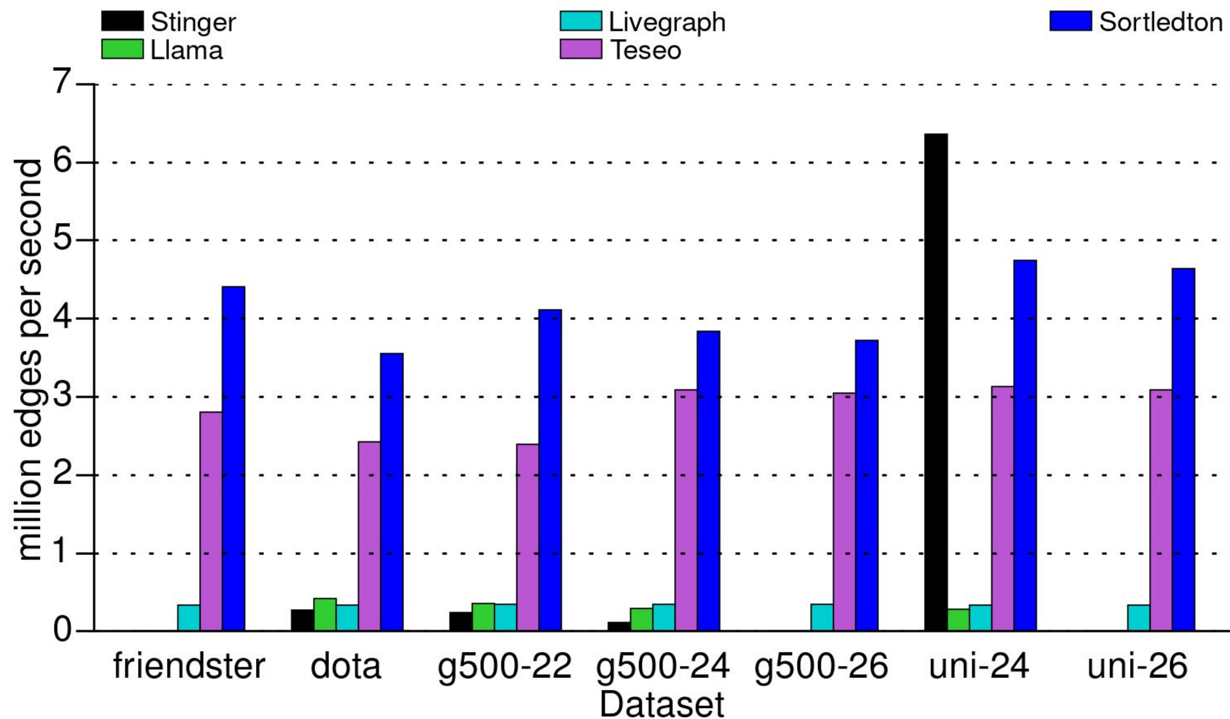
Evaluation

- 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

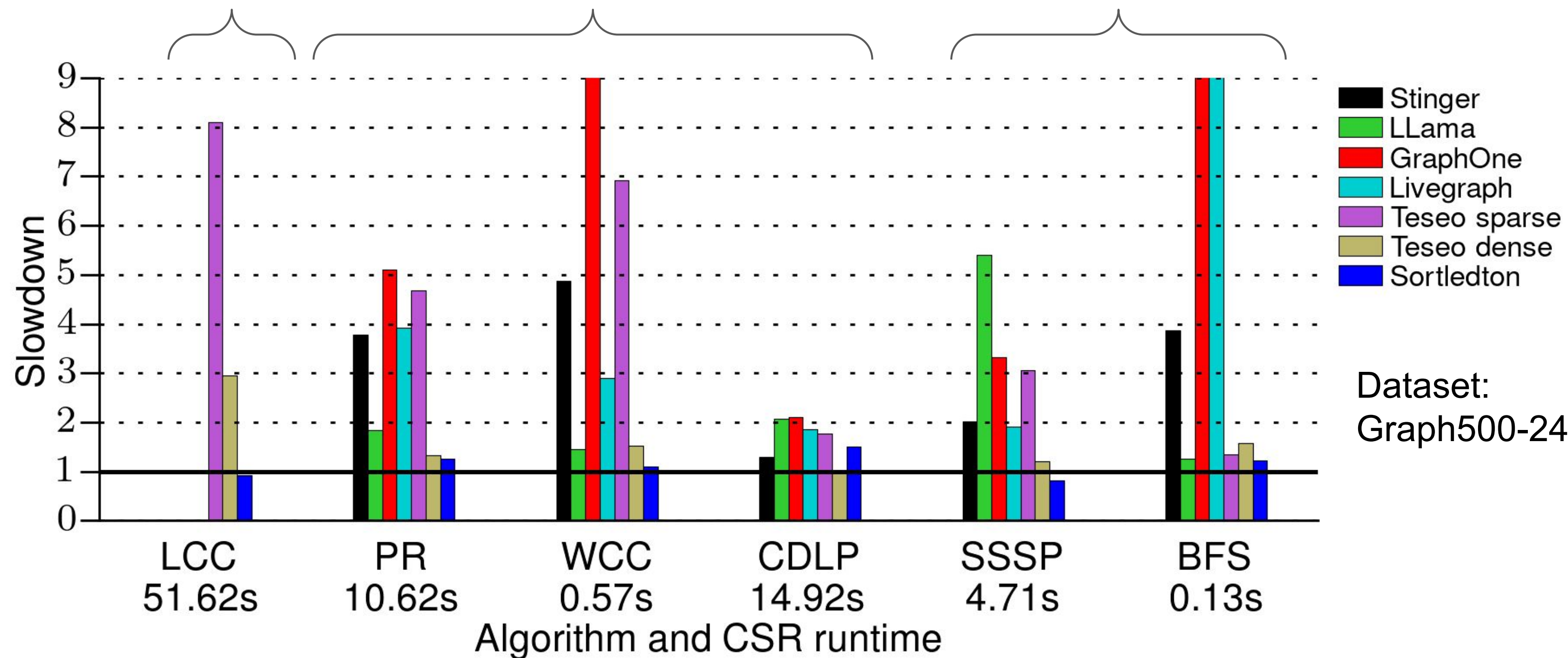
Update performance - power law graphs



Update performance - uniform graphs

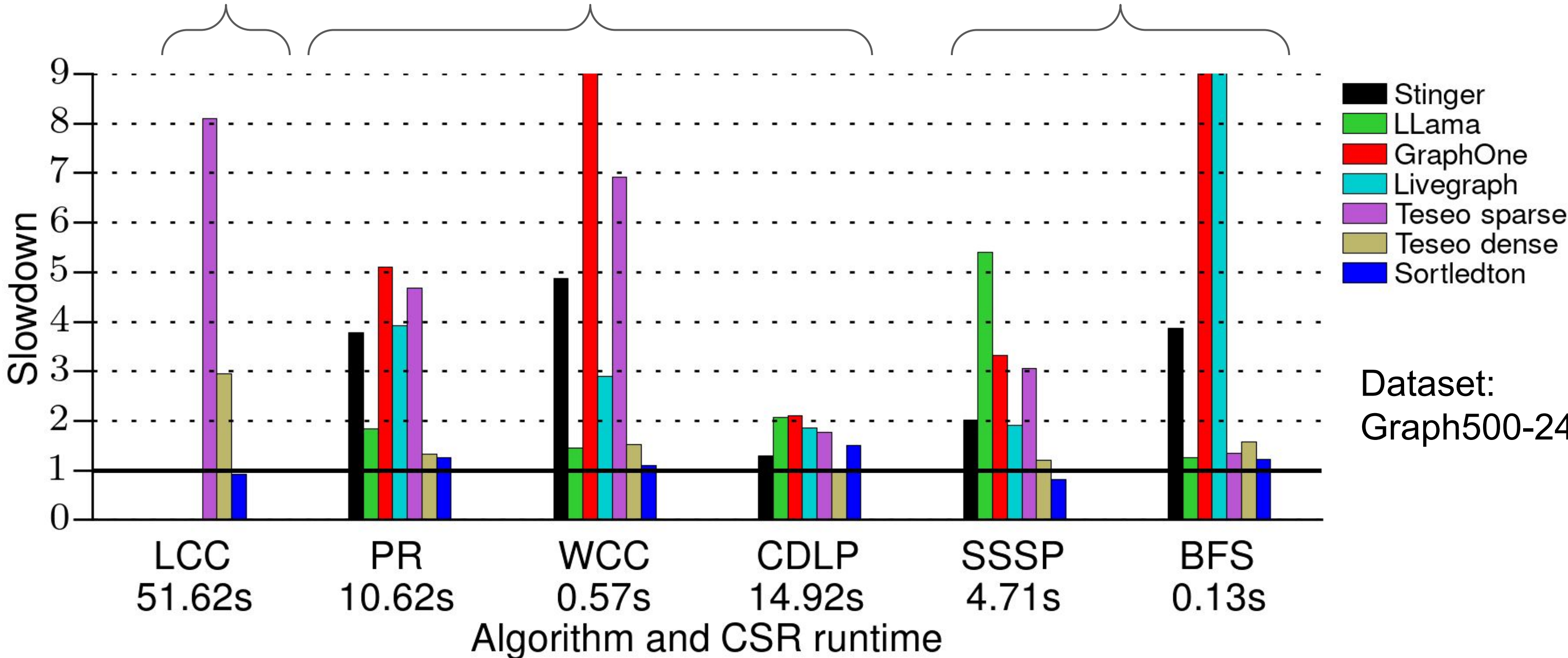


Graphalytics benchmark performance



Dataset:
Graph500-24

Graphalytics benchmark performance

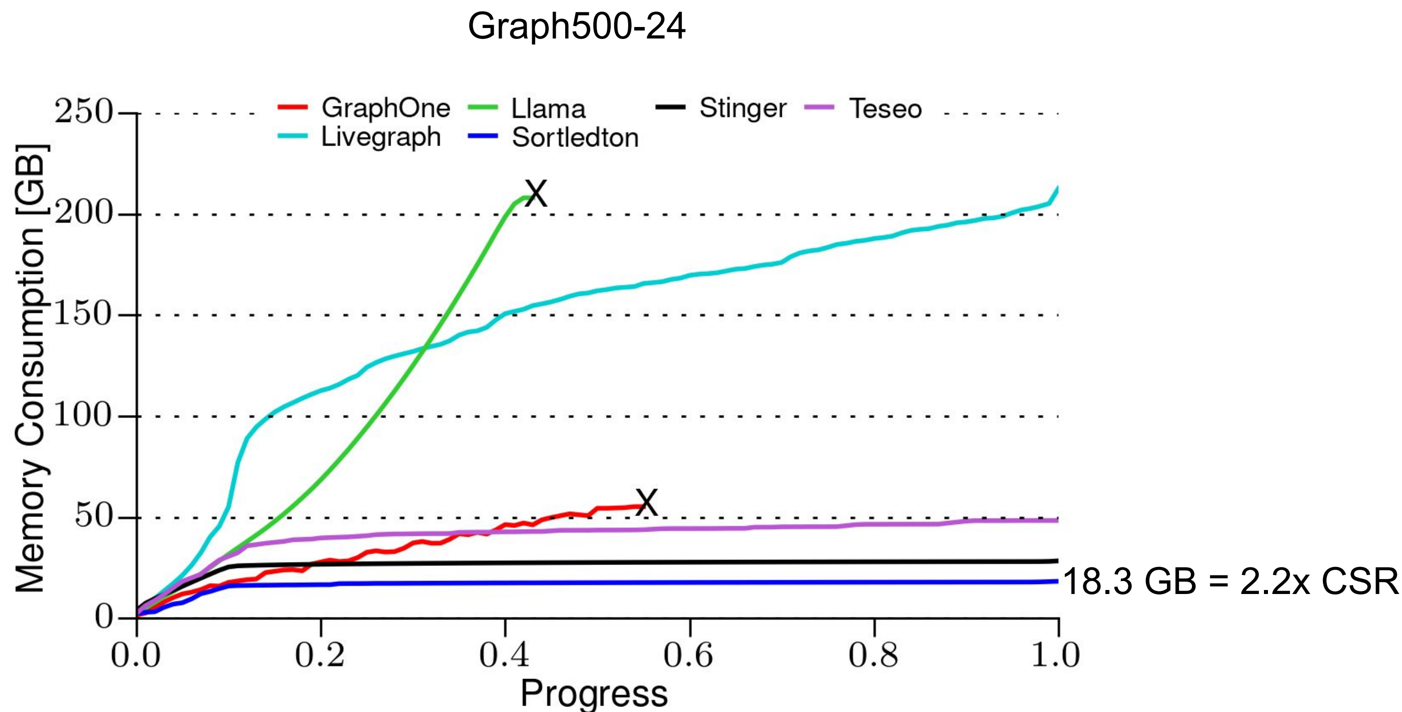


Dataset:
Graph500-24

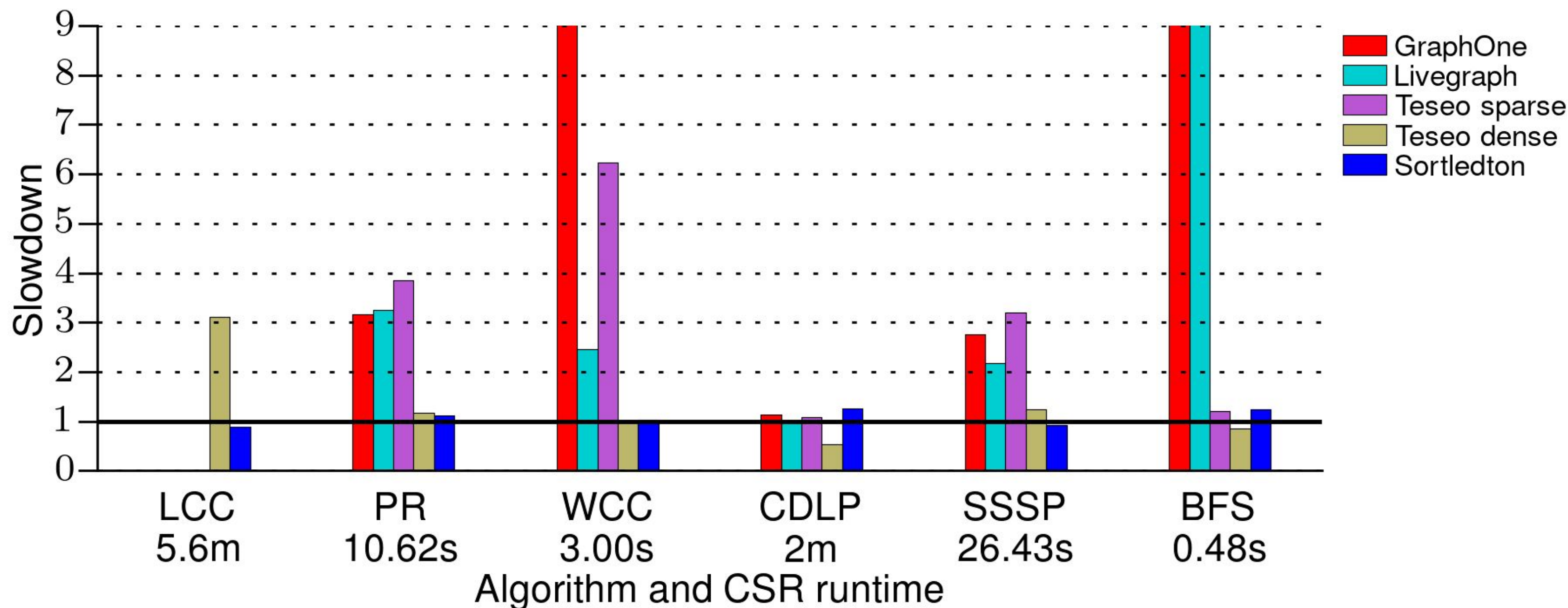
Conclusions

- 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

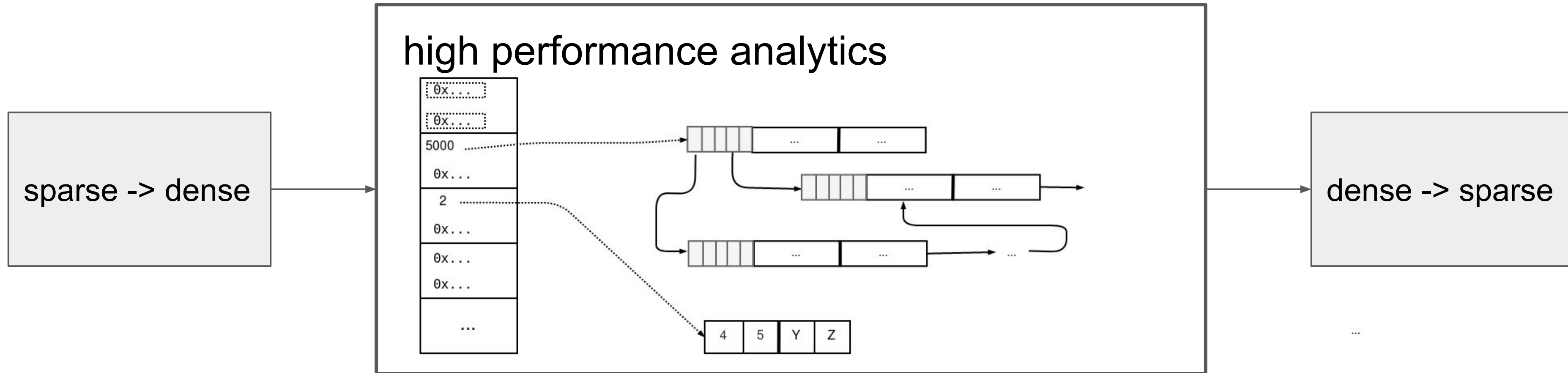


Graphalytics Benchmark Performance



Dense vertex identifier for algorithmic specific access

- we translate arbitrary vertex identifier {0, 5, 1000, …} on insertion into the dense domain [0, 1, 2, …] for better analytical performance



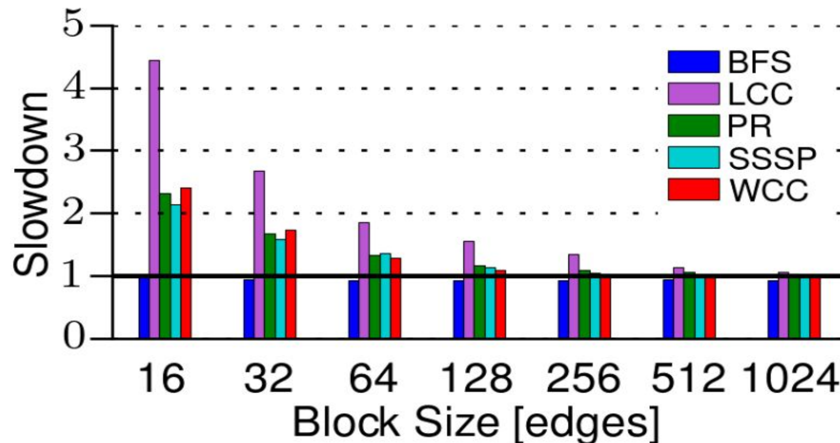
Sorted Blocks for Sequential Edge Access with intersections

Sorted blocks for neighbourhoods

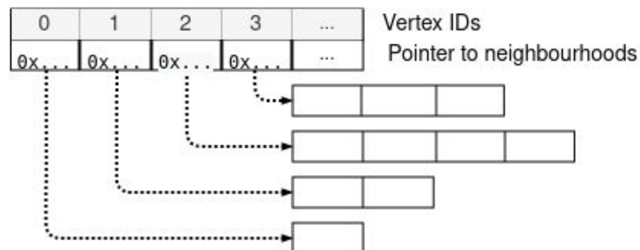
normalized against

Sorted Vector for neighbourhoods (optimal, static)

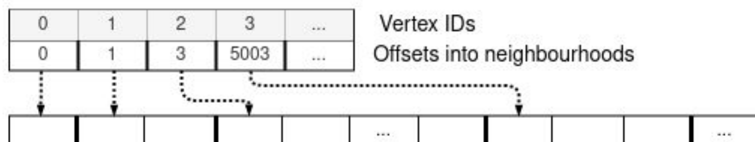
Graphalytics Algorithms

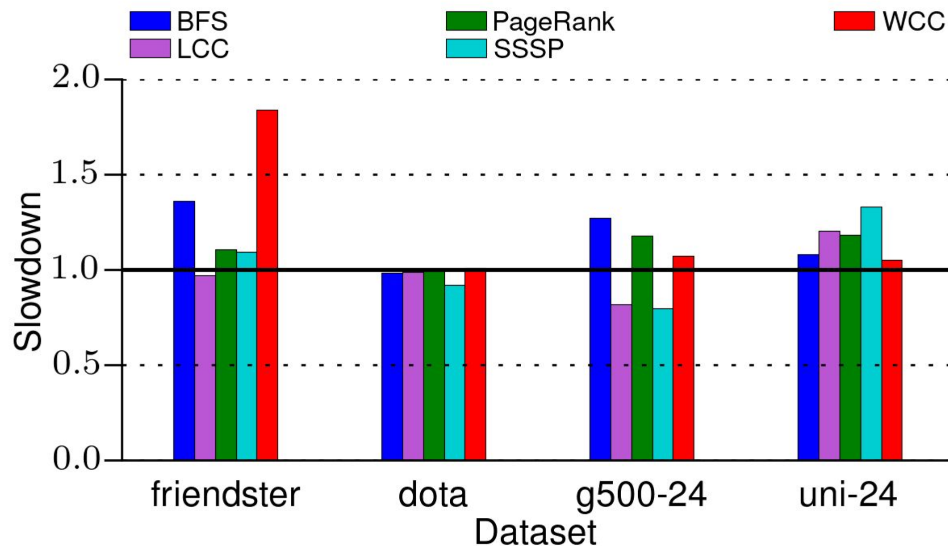


Optimizing for Sequential Vertex Access



normalized against





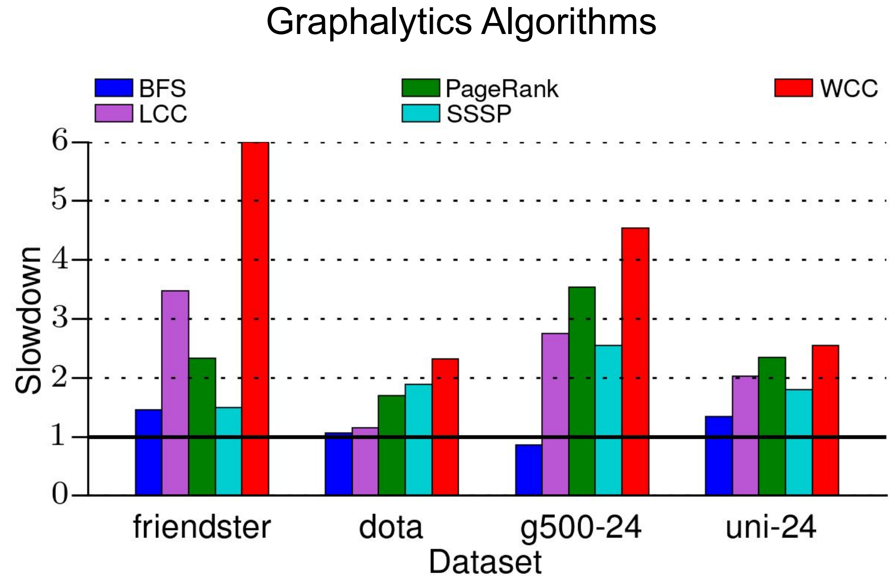
Which Vertex ID Domain to Store for Random Algorithmic-Specific Access?

sparse domain: {0, 3, 1000, 1001, ...}

normalized against

dense domain: [0, 1, 2, 3, …]

Graphalytics Algorithms



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]

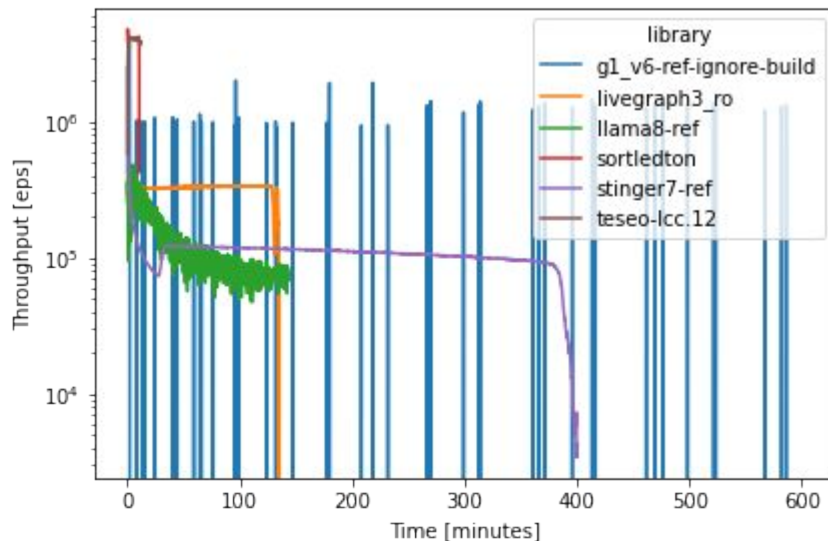
Computation	Total
Finding Connected Components	55
Neighborhood Queries (e.g., finding 2-degree neighbors of a vertex)	51
Finding Short / Shortest Paths	43
Subgraph Matching (e.g., finding all diamond patterns, SPARQL)	33
Ranking & Centrality Scores (e.g., PageRank, Betweenness Centrality)	32

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
- LLama throughput diminishes over time
- Graphone has severe issues with edge removals



Properties of existing approaches

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