ZipG: Serving Queries on Compressed Graphs

Designing a memory-efficient distributed graph store

Zongheng Yang, Rachit Agarwal, Evan Ye, Anurag Khandelwal, Ion Stoica

amplab

User-Facing Graph Serving

- Online graph serving increasingly important in social networks: FB, Twitter, LinkedIn, Pinterest
- **Graphs are huge:** unlike graph processing, nodes/edges have attributes (location, name, etc.)
  - Facebook ∼ 1 billion nodes, ∼ 1 trillion edges $\Longrightarrow\sim$ 1.5 **petabytes**
- **Graph queries are complex:** non-trivial execution choices; consider "find friends who live in SF":
  1. *[my friends] joins [San Franciscans]*: second list can be huge, high computation cost
  2. *For friend f, check f.loc == SF in parallel*: touches arbitrary parts of graph, poor locality

For efficiency, caching more data is critical

ZipG's Approach

- **Compression helps caching!** ZipG thus queries directly off the compressed representation of an input graph, via Succinct
- Many ways to use Succinct! ZipG picks a graph layout that invokes as few primitive ops as possible (`search`, `extract`) to run graph ops

Existing Graph Stores

Native graph storage [Neo4j]
Adjacency list; uses pointers to link scattered per-node states (nhbr list, attributes) together

- flexible data access; fast traversal
- caching ineffective, due to scattered data

Relational/key-val storage [Titan on Cassandra; Facebook's TAO on MySQL; Pinterest's Zen on HBase]

- well-understood architecture; block compression
- small access may decompress large blocks

ZipG queries directly on compressed graphs: proves effective for caching, flexibility, scalability

ZipG Layout: Edge Table

•EdgeCount ⋆ TWidth † DWidth

$S_1#EdgeType_1	Metadata	$T_0, ..., T_M$	$D_0, ..., D_M$	PropertyList
$S_1#EdgeType_2	Metadata	$T_0, ..., T_N$	$D_0, ..., D_N$	PropertyList
⋮				
$S_f#EdgeType_k	Metadata	$T_0, ..., T_L$	$D_0, ..., D_L$	PropertyList

- Fields: `<Source NodeID, EdgeType, Metadata, Timestamps, Destination IDs, Properties>`
  - *"All comments on Golden Gate Bridge"*; locatable using one `search` operation
- **Flexibility:** random access into any field (timestamp, dst ID, prop.), only extracting a few more bytes (Metadata) than the absolute minimum
- **Range queries:** permits binary search on edges, since a sort order can be imposed
  - Ex: *get edges up to 7 days ago*, without inspecting all edges in the list

ZipG API

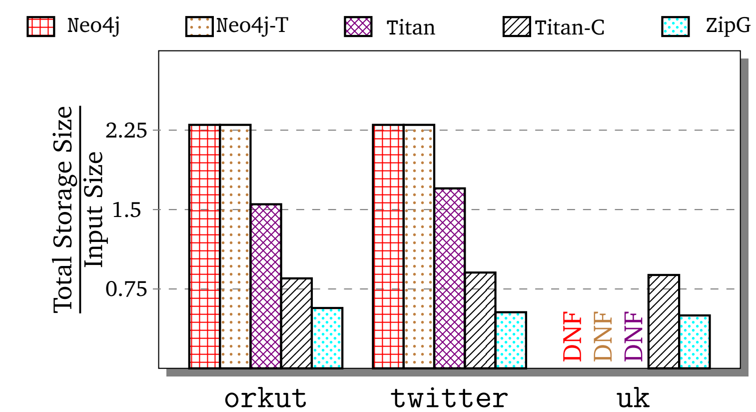
Query	Example
`get_edge_record(nodeID, edgeType)`	Get a handle to all information on **Alice's friends**.
`get_time_range(edgeRec, tLo, tHi)`	Find Alice's friends **between 2014 and 2015**.
`get_edge_data(edgeRec, dataType, timeOrder)`	Find Alice's **ten most recent friend**.
`get_node_property(nodeID, propertyKeys)`	Fetch **Alice's age** and **location**.
`get_node_ids(propertyList)`	Find people in **Berkeley** who like **Music**.
`get_neighbor_ids(nodeID)`	Find connections to **Alice**.

Extensibility: via convenient APIs, users can add their own queries against layout

- expressive enough to implement FB's entire TAO read APIs in tens of lines

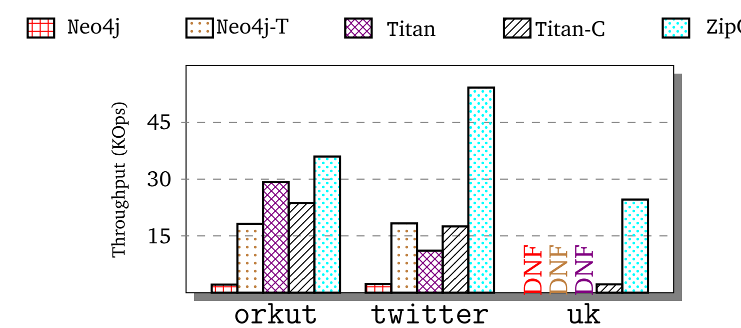
Evaluation: Storage Footprint

Dataset	#nodes, #edges	Size	Type
orkut	3M, 117M	20 GB	social
twitter-2010	41M, 1.4B	250 GB	social
uk-2007-05	105M, 3.7B	636 GB	web

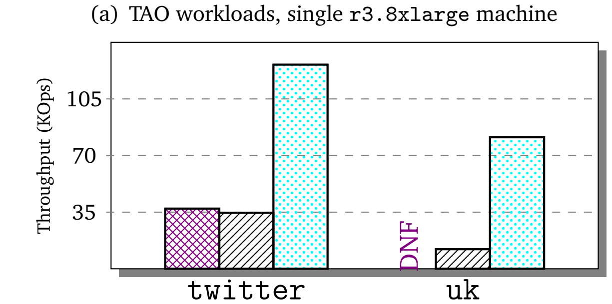


Takeaway: ZipG up to 4x lower storage footprint (i.e., can cache 4x larger graphs)

Evaluation: System Throughput



(a) TAO workloads, single `r3.8xlarge` machine



(b) TAO workloads, 10-node `m3.2xlarge` cluster

Takeaway: ZipG serves complex ops efficiently; performant even when graphs fit in memory (orkut)

We'd *love* your feedback!: {zhyang, ragarwal}@berkeley.edu