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 \(\implies \approx 1.5 \text{ petabytes}\)
- **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 \(f_.loc = \text{SF}\) 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
```

- 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

```
get_edge_record(nodeID, edgeType)
get_time_range(edgeRec, tLo, tHi)
get_edge_data(edgeRec, dataType, timeOrder)
get_node_property(nodeID, propertyKeys)
get_node_ids(nodeID)
get_neighbor_ids(nodeID)
```

Example

```
get_edge_record(nodeID, edgeType)
get_time_range(edgeRec, tLo, tHi)
get_edge_data(edgeRec, dataType, timeOrder)
get_node_property(nodeID, propertyKeys)
get_node_ids(nodeID)
get_neighbor_ids(nodeID)
```

Evaluation: Storage Footprint

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#nodes, #edges</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>orkut</td>
<td>3M, 117M</td>
<td>20 GB</td>
<td>social</td>
</tr>
<tr>
<td>twitter-2010</td>
<td>41M, 1.4B</td>
<td>250 GB</td>
<td>social</td>
</tr>
<tr>
<td>uk-2007-05</td>
<td>105M, 3.7B</td>
<td>636 GB</td>
<td>web</td>
</tr>
</tbody>
</table>

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

Evaluation: System Throughput

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Throughput (KOps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>orkut</td>
<td>15</td>
</tr>
<tr>
<td>twitter</td>
<td>30</td>
</tr>
<tr>
<td>uk</td>
<td>45</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Throughput (KOps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>twitter</td>
<td>70</td>
</tr>
<tr>
<td>uk</td>
<td>105</td>
</tr>
</tbody>
</table>

(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