ZipG: A Memory Efficient Graph Store

Presented by: Tuhin Tiwari

Big Data Management Platforms
What’s the problem?

- How do you achieve execution of large fraction of queries (possibly in a distributed setting)?
- Which is stored in-memory?
- to achieve query interactivity?
Query Interactivity

- High performance for interactive user-facing queries
- Millisecond-latency & High throughput
- Query: Find friends of Alice who live in Ithaca.

2 sub-queries. Compute final result using join or intersection.

Find friends of Alice. Then for each friend, check whether or not friend lives in Ithaca.
Query Interactivity

- High performance for interactive user-facing queries
- Millisecond-latency & High throughput
- Query: Find friends of Alice who live in Ithaca.

2 sub-queries. Compute final result using join or intersection.

Find friends of Alice. Then for each friend, check whether or not friend lives in Ithaca.
ZipG

- Distributed, memory-efficient representation of Graph Data using compression techniques.
- Applications can operate on the compressed representation, avoiding compression & decompression.
- Effective implementation of interactive graph queries.
- Resolves the challenges of expressivity, scalability, and performance.
Succinct

Succinct: Enabling Queries on Compressed Data
Rachit Agarwal, Anurag Khandelwal, and Ion Stoica, University of California, Berkeley
https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agarwal

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’15).
May 4–6, 2015 • Oakland, CA, USA
ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’15)

ZipG’s Data Model

Nodes

Edges

Nodes

Types: comments, likes, relationships
3 tuple : {sourceID, destinationID, EdgeType}

Associated Properties

PropertyList
Tuple : {PropertyID, PropertyValue}
ZipG’s Interface

Application

Query Graph

Interacts with interface just like operating on raw graph data

ZipG

Compressed Graph
ZipG’s Interface

Interacts with interface just like operating on raw graph data

ZipG

Compressed Graph

Application

Query Graph

EdgeData

EdgeRecord

TimeOrder
ZipG API

- Exposes a functionally rich API to implement all queries from TAO, LinkBench, and GraphSearch.

```
List<NodeID> g.get_neighbor_ids(nodeID, edgeType, propertyList)  # Find Alice's friends who live in Boston.

EdgeRecord g.get_edge_record(nodeID, edgeType)  # Get all information on Alice's friends.
```

- Delete, Append and Update
- Node-based and edge-based
Graph Representation - NodeFile

• Unstructured file that stores NodeIDs and NodeProperties.
• Small metadata - length of PropertyValues
• 3 Data structures:
  o PropertyID is assigned a delimiter and is mapped to order.
  o Flat file contains PropertyLists (prepended and appended with delimiters) and metadata (space-latency tradeoff)
  o Simple 2D array with sorted list of NodeIds and its offset.

<table>
<thead>
<tr>
<th>PropertyID</th>
<th>(Order, Delimiter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>(0, *)</td>
</tr>
<tr>
<td>location</td>
<td>(1, †)</td>
</tr>
<tr>
<td>nickname</td>
<td>(2, +)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NodeID</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>0</td>
</tr>
<tr>
<td>Bob</td>
<td>21</td>
</tr>
<tr>
<td>Eve</td>
<td>42</td>
</tr>
</tbody>
</table>

284*42†Ithaca•Ally‡
095*†Princeton•Bobby‡
203*24†•Cat‡
Graph Representation - EdgeFile

- Unstructured file that stores Edges and EdgeProperties.
- 3 tuples – \{sourceNodeID, destinationNodeID, EdgeType\}
- Small metadata - EdgeCount
- Edge record contains(from left to right):
  - EdgeRecord for (NodeID, EdgeType) pair - $NodeID#EdgeType$
  - Metadata - Edgecount
  - Timestamps – impose ordering & efficient query execution(binary search on timestamps) – fixed length but maximum length is stored as metadata.
  - DestinationIDs – Sort DestinationIDs according to Timestamps. Efficient random access.
  - PropertyList – Sorted according to edges. Encoded similar to Node PropertyList. Doesn’t support search on Edge PropertyList.
Graph Partitioning (Sharding)

- Hash-partitioning scheme – NodeID maps to shards
- Default: 1 per core
- Data corresponding to NodeID (PropertyList and EdgeInformation) is stored in each shard.
- All node and edge-data associated with a node is co-located on the same shard.

Enabling execution of neighbor queries
Log Store Server

- A logstore per shard or for all shards on a server.
- Periodically merge Logstore data with compressed data.
- Once size of LogStore crosses threshold, compressed into memory-efficient representation & new LogStore initiated.
- Additional nodeID to Logstore-offset pointers can be stored for random access to avoid scanning entire Logstore
  - All data local
  - Resolves concurrent read/write conflicts at each server
Fanned Updates

- Challenge: High write rates over compressed graphs
- Avoid touching all shards
- Update pointers logically chain data corresponding to same node or edge.
- Pointers - uncompressed
Function Shipping

- Computation pushed closer to data via function shipping
- Supports multi-level i.e., a subquery may be further decomposed into sub-subqueries & forwarded to respective servers.
Evaluation

- Compared ZipG with two open-source graph stores: single machine (Neo4j) and distributed implementation (Titan)
- Datasets: TAO, LinkBench, and GraphSearch
- By avoiding overheads of scans, ZipG achieves higher throughput because of random-access for TAO
- Write-based queries outperform in ZipG in LinkBench
- Graph Search: Neo4j-Tuned achieves better performance when the uncompressed graph fits entirely in memory. ZipG overheads
- With increasing graph size, ZipG achieves 3x higher throughput than Neo4j-Tuned
- ZipG sees 2.5x increase in throughput proportional to the number of cores – in distributed setting – for TAO
- LinkBench and GraphSearch – no change in performance in distributed settings because of search
## PROS AND CONS OF ZIPG

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Minimum interference on queries being executed on previously computed data.</td>
<td>1. Fault-tolerance is traditional. But not described in detail for LogStore.</td>
</tr>
<tr>
<td>2. Can achieve higher memory efficiency with dedicated Log Server.</td>
<td>2. Overhead if the uncompressed data fits entirely in memory.</td>
</tr>
<tr>
<td>3. Single Log Server can resolve read/write conflicts by avoiding complicated data structure.</td>
<td>3. Search-based queries are an overhead as ZipG touches all partitions.</td>
</tr>
</tbody>
</table>
Using an API interface, the applications are presented with a choice of graph database which allows working on the compressed representation of large graph data, in-memory and enables interactive query-execution.
Since search is one of the most often used queries for Graph databases, is ZipG really a better choice than Titan and Neo4j? Knowing the workload, why couldn’t they extend this query on edge properties?

Could the Authors have chosen a better system than Succinct as the underlying layer of ZipG?

What can be the alternative of LogStore being an overhead for data that completely fits in-memory?

Deletion is mentioned using bitmap but what about their Garbage Collection Policy?