ZipG: A Memory-efficient Graph Store for Interactive Queries

Authors: Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal*, Ion Stoica

Presented by: Tuhin Tiwari

CS 848: Big Data Management Platforms
Content

• Introduction
• ZipG API
• ZipG Implementation
• Evaluation
• ZipG Pros & Cons
• Conclusion
• Discussion
INTRODUCTION
YOU TALK ABOUT DATA ANALYTICS, PEOPLE LOVE YOU.

YOU TALK ABOUT "BIG DATA" AND EVERYONE LOSES THEIR MINDS
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?
Existing Graph Systems

Flexibility Vs Scalibility

Offline Graph Analytics
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.

Join Bomb

Random access & locality
## ZipG

### Compressed representation of graph

<table>
<thead>
<tr>
<th>E</th>
<th>S</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>P</td>
<td>A</td>
<td>R</td>
</tr>
<tr>
<td>R</td>
<td>L</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>I</td>
<td>O</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>R</td>
</tr>
<tr>
<td>SI</td>
<td>I</td>
<td>M</td>
</tr>
<tr>
<td>V</td>
<td>L</td>
<td>A</td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>N</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>E</td>
</tr>
</tbody>
</table>

**ZIPG**

**SUCCINCT**
Succinct’s Data Compression

SUCCINCT STORE

SUFFIX STORE

LOG STORE

Data Appends
ZIPG API
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

Application

Query Graph

Interacts with interface just like operating on raw graph data

ZipG

Compressed Graph

EdgeData

EdgeRecord

TimeOrder
ZipG API

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

- Node-based

  ```java
  List<NodeID> g.get_neighbor_ids(nodeID, edgeType, propertyList)
  ```

  Find Alice's friends who live in Boston.

- Edge-based

  ```java
  EdgeRecord g.get_edge_record(nodeID, edgeType)
  ```

  Get all information on Alice's friends.

- Delete, Append and Update
Graph Representation - NodeFile

- NodeIDs and NodeProperties.
- Small metadata - length of PropertyValues
- 3 Data structures: PropertyID, Flat file, 2D array
Graph Representation - NodeFile

- NodeIDs and NodeProperties.
- Small metadata - length of PropertyValues
- 3 Data structures: PropertyID, Flat file, 2D array

Latency b/n Storage and Random Access
Graph Representation - EdgeFile

- Edges and EdgeProperties.
- 3 tuples – {sourceNodeID, destinationNodeID, EdgeType}
- Small metadata - EdgeCount
- Edge record contains (from left to right):
  
  $NodeID#EdgeType

- Timestamps, DestinationIDs, PropertyList
- Doesn’t support search on Edge PropertyList.
ZIPG
IMPLEMENTATION
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

- Challenge: High write rates over compressed graphs
- Minimal interference on ongoing query-execution.
- Memory efficiency – over per-server LogStores
- Avoids complicated data structure for concurrency
Fanned Updates

- Challenge: Fragmentation, new edges added to a node.
- 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
Evaluation

- **Storage Footprint**: ZipG is 2x smaller than Neo4j but comparable to Titan - compressed.
- **TAO**: ZipG sees 2.5x increase in throughput proportional to the number of cores – in distributed setting.
- **LinkBench**: Write-based queries outperform in ZipG. No change in performance in distributed setting.
- **Graph Search**: Neo4j-Tuned achieves better performance when the uncompressed graph fits entirely in memory- ZipG overheads. No change in performance in distributed setting.
- ZipG outperforms Neo4j for branched or long linear path traversals. Graph traversal – Neo4j outperforms because of aggregation overheads in ZipG.
ZIPG PROS & CONS
<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Dedicated LogStore help avoid decompression of data as well as achieve higher memory efficiency.</td>
<td>1. Fault-tolerance is traditional. But not described in detail for LogStore since that is the central system.</td>
</tr>
<tr>
<td>2. Data Persistence is achieved by performing all write operations on LogStore and update pointers in sec. store.</td>
<td>2. Overhead if the uncompressed data fits entirely in memory.</td>
</tr>
<tr>
<td>3. Concurrency control is also achieved through LogStore. Locks only required for uncompressed update pointers and deletion bitmaps.</td>
<td>3. Search-based queries like “Find Musicians in Ithaca” are an overhead as ZipG touches all partitions.</td>
</tr>
<tr>
<td>4. ZipG is rich enough to implement almost all complex queries</td>
<td>4. No guarantees of consistency and transactions</td>
</tr>
</tbody>
</table>
CONCLUSION
KEY TAKEAWAY

- ZipG, a memory-efficient graph store
- Executes queries directly on compressed graph representation (no decompression & scans)
- Rich functionality
  - Queries from industrial workloads: Regular path queries & graph traversals
- New log-structured graph storage
  - Efficient read & write queries
DISCUSSION
Could the Authors have chosen a better system than Succinct as the underlying layer of ZipG?

Will it be a good idea to include bitmap representation(from previous paper) for all operations instead of using suffix and inverse-suffix data structures?

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

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

Do the pointers really not create an overhead, even when the number of edges increase at a high rate?

Update queries are implemented as deletes followed by an append. With no proper fault tolerance in place, the system might lose some edge properties if the system shuts down.

Conflict between input data and delimiters chosen for PropertyList might be a problem?
THANK YOU