ZipG: A Memory-efficient Graph Store for interactive Queries

Cheng Zhao

August 27, 2017

1 Motivation

- Distributed graph computing system is particularly hard to reduce queries like *"Find friends of Alice who live in Ithaca"*. It may incur high computational and bandwith overheads.
  Typical graph queries exhibit *little or no locality* — the query may touch arbitrary parts of the graph, potentially across multiple servers, some of which may be in memory and some of which may be on secondary storage.

- Traditional block compression techniques (*e.g.*, gzip) are inefficient for graph queries precisely due to **lack of locality**.

- Some techniques and systems executing queries on compressed graphs, but they often **ignore node and edge properties** and **are limited to a small subset of queries on graph structure** (*e.g.*, extracting edges incident on a node, or subgraph matching).

2 Summarize

What differentiates ZipG from existing graph stores is **its ability to execute a wide range of queries directly** on this compressed representation. Succinct: a distributed data store that supports **random access and arbitrary substring search** directly on compressed unstructured data and key-value pairs. Two challanges:

- support high write rates — using log-structured approach,but avoids touching all logs using the idea of fanned updated;

- compressed representation is in providing strong consistency guarantees and transactions — does not attempt to resolve this challenge.

3 ZipG Design

3.1 Succinct Background

Succinct interfaces: a flat file interface for executing queries on unstructured data, and a kv interface for queries on semi-structured data. In ZipG:

- **Random access** (on flat files `extract(offset, len)` ; on KV `get(id)`)
- **Search** (on flat files, returens offset)

Succinct achieves compression using sampling —– only a few sampled values (e.g., for sampling rate α value at indexes 0, α, 2α, ..) from the two data structures are stored.Overall, for a sampling rate of α, Succinct's storage requirement are roughly $2n\lceil \log n \rceil / \alpha$ and the latency for computing each unsampled value is roughly α.

See detail in Succinct Note.

3.2 Graph Representation

Note that each edge may potentially have a different EdgeType and may optionally have a `Timestamp` .

ZipG's graph layout uses two **flat unstructured** files:

- **NodeFile** stores all NodeIDs and correspinding properties. (add a small amount of metadata to the list of (NodeID, properties) before compressing)
  - It is optimized for two kind of queries on nodes: a) given a `(NodeID, List<propertyID>)` pair, extract the corresponding `propertyValues`; and b) given a `PropertyList`, find all `NodeID`s whose properties match the `propertyList`.
  - The lengths of `propertyValues` are encoded using a global fixed size `len`.
- **EdgeFile** stores all the EdgeRecords. (add metadata and convert variable length data into fixed length data before compressing).
  - Each edge is uniquely identified by the 3-tuple `(sourceNodeID, destinationNodeID, EdgeType)`. Following the metadata, the EdgeFile stores the `TimeStamps` for all edges.
  - Efficiently executing such queries requires performing binary search on timestamps. Storing timestamps using delta encoding.

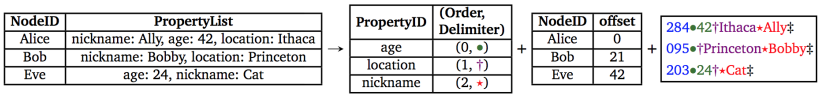

Figure 1: An example for describing the layout of NodeFile. See description in §3.3.

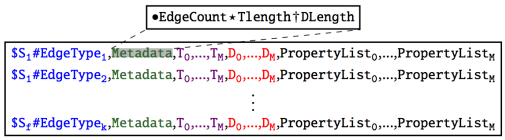

Figure 2: EdgeFile Layout in ZipG (§3.3). Each row is an EdgeRecord for a `(sourceID, edgeType)` pair. Each EdgeRecord contains, from left to right, metadata such as edge count and width of different edge data fields, sorted timestamps, destination IDs, and edge PropertyLists.

3.3 Fanned Updates

Another challenge for ZipG is to **support high write rates over compressed graphs**. Naively using a log-structured approach in graph stores results in nodes and edges having their data "fragmented" across multiple logs and each query will now need to touch all the data, thus reduced throughput.

ZipG use the idea of fanned updated — each server in ZipG stores a set of update pointers that ensure that during query execution, ZipG touches exactly those logs (and bytes within the logs).

ZipG instead uses a single LogStore for the entire system — all write queries are directed to a query-optimized (rather than memory-optimized) LogStore. Once the size of the LogStore crosses a certain threshold, the LogStore is compressed into a memory-efficient representation and a new LogStore is instantiated.

对于非结构化和半结构化的数据，往往以邻接模型存储，这样单LogStore就比多机的LogStore要更好。但是对于图结构数据，使用单LogStore会导致 fragmented storage。比如，假设在时间 $t = 0$ 时，我们上传图数据并创建一个单LogStore l。考虑一个包含原始数据的节点 u，在一次更新时（从 $t = 0$ 时刻开始，在 $t = t_1$ 时刻结束）使得 l 达到了临界值，那么在 t_1 之后，节点 u 的数据会被分成三片：节点 u 的原始数据，旧的LogStore l，新的LogStore l'。

Fanned Updates ZipG stores these update pointers only at the shard where the node/edge first occurs. Only the shard containing the data for node u in pre-loaded graph will store the update pointers for all occurrences in shards corresponding to l, l' and any other future shards. ZipG keeps these pointers uncompressed.

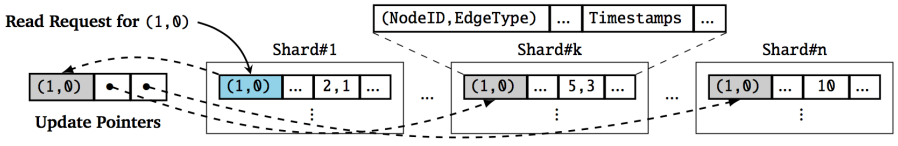

Figure 3: Update Pointers for the EdgeFile (§3.5).