ZipG

A Memory-efficient Graph Store for Interactive Queries

Cheng Zhao

September 6, 2017

Table of contents

Motivation

Motivation

- 分布式图计算系统往往较难处理 *“Find friends of Alice who live in Ithaca”* 这类查询，因为典型的图查询几乎没有“**局部性**”（它可能会访问图的任意部分；查找所有包括 *Alice’s friends* 的块）
- 传统的块压缩技术 (*e.g.*, gzip) 也因为缺少“局部性”而在处理图查询时效率很低
- 一些系统在压缩后的图上进行查询，但他们往往忽略点和边的属性，并且只能进行有限的操作 (*e.g.*, 查找与点相连的边，子图匹配) [1, 2, 3, 4, 5, 6, 7]

Preliminary: Succinct

ZipG Design

Graph Representation

- **NodeFile**
  - 记录每个属性的字典序和使用的分隔符；
  - 记录每个 NodeID 对于的offset；
  - 在每条 PropertyList 的开始记录每列属性占用的空间。

主要处理两类请求：

1. 给定(`NodeID, List<propertyID>`)找出该节点的对应的属性值；
2. 给定一个属性列表，找到所有与之匹配的节点。

NodeID	PropertyList
Alice	nickname: Ally, age: 42, location: Ithaca
Bob	nickname: Bobby, location: Princeton
Eve	age: 24, nickname: Cat

→

PropertyID	(Order, Delimiter)
age	(0, •)
location	(1, †)
nickname	(2, ⋆)

+

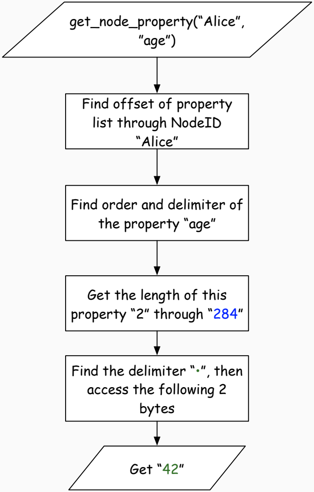
NodeID	offset
Alice	0
Bob	21
Eve	42

+

284•42†Ithaca⋆Ally‡
095•†Princeton⋆Bobby‡
203•24†⋆Cat‡

Query Execution

- `get_node_property(NodeID, propertyID)`



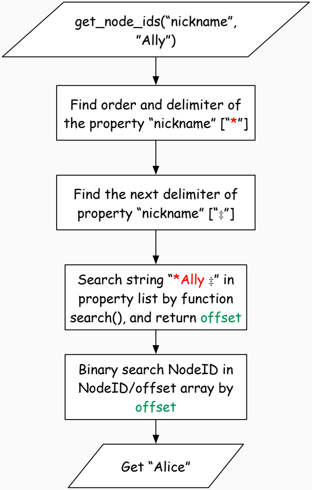
NodeID	offset
Alice	0
Bob	21
Eve	42

PropertyID	(Order, Delimiter)
age	(0, •)
location	(1, †)
nickname	(2, ⋆)

284•42†Ithaca⋆Ally‡
095•†Princeton⋆Bobby‡
203•24†⋆Cat‡

Query Execution

- `get_node_ids(propertyID, propertyValue)`



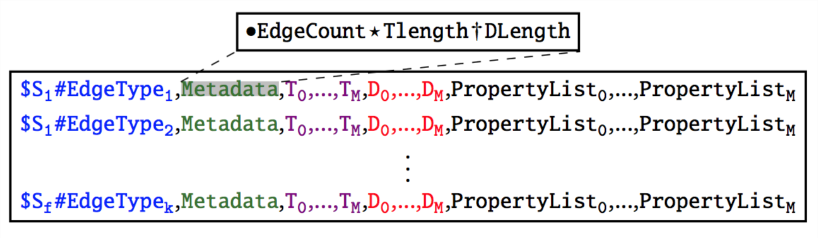
PropertyID	(Order, Delimiter)
age	(0, •)
location	(1, †)
nickname	(2, ⋆)

284•42†Ithaca⋆Ally‡
095•†Princeton⋆Bobby‡
203•24†⋆Cat‡

NodeID	offset
Alice	0
Bob	21
Eve	42

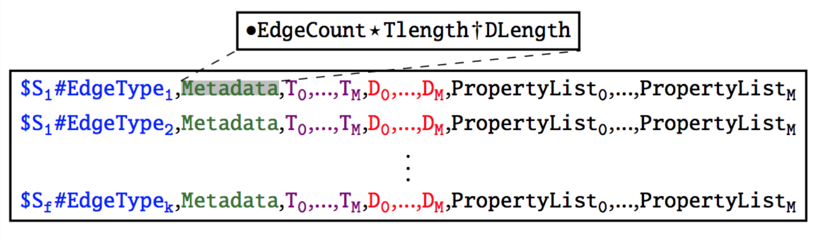
Graph Representation

- **EdgeFile**
  - 一条 EdgeRecord 记录一组同一源点相同类型的边
  - 时间戳存储需要支持二分查找，存储的两个极端
    1. **Space-efficient** 用最少的空间表示每个时间戳，但是需要用分隔符或者存储长度来标识每个时间戳；或者使用 *delta encoding* 存储
    2. **Latency-efficient** 固定一个全局时间戳长度，每条 EdgeRecord 存储相同长度的时间戳



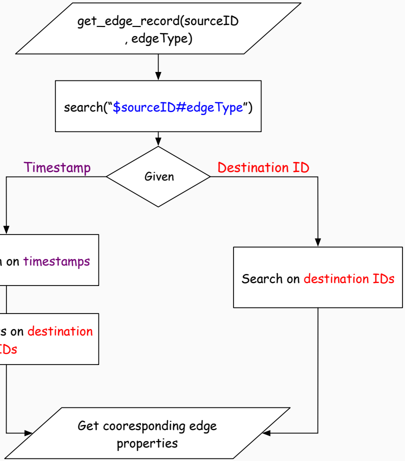
Graph Representation

- **EdgeFile**
  - 一条 EdgeRecord 记录一组同一源点相同类型的边
  - Metadata 存储边的个数，时间戳的长度，目的节点的长度
  - 以此条 EdgeRecord 内需要存储的时间戳最大长度作为每个时间戳的长度（每条 EdgeRecord 的时间戳长度可能不同），并在Metadata中记录
  - 以相同方法存储对应的目的节点，并在Metadata中记录其长度
  - 边属性的存储方法类似于节点属性，但是目前ZipG不支持在此上做查询



Query Execution

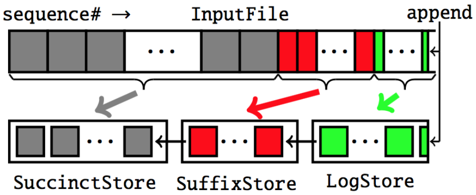
- `get_edge_record(sourceID, edgeType)`



Sharding

根据CPU的核心数划分数据（1 shard per core），根据 NodeID 将节点哈希到各个 shard 上，并存储所有与节点的出边信息。

Succinct Store



	Succinct Store	Suffix Store	Log Store
Stores	Comp. Data (§3.1)	Data + AoS2Input	Data + Inv. Index
Appends	-	Bulk	Fine
Queries	§3.2	Index	Scans+ Inv. Index
#Machines	$n-2$	1	1
%Data(est.)	$>99.98\%$	$<0.016\%$	$<0.001\%$
Memory	$\approx 0.4\times$	$\approx 5\times$	$\approx 9\times$

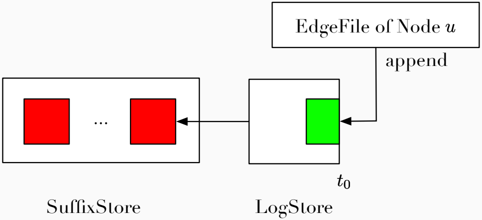
LogStore

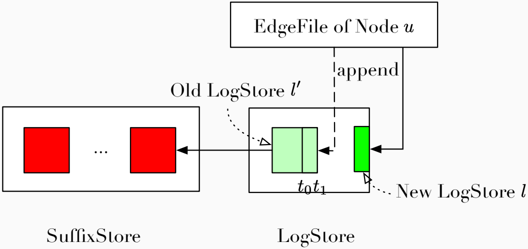
ZipG 在整个系统中只用一个 LogStore (query-optimized)，一旦其大小超过阈值即压缩为 memory-optimized 存储方式并创建一个新的 LogStore. 其优势在于:

1. 不需要对数据进行解压和再压缩;
2. 单 LogStore 比之前的方法（每个Worker上一个LogStore）更省内存;
3. 避免了多 LogStore 时同步读写的麻烦。

Fanned Updates

- 对于非结构化和半结构化的数据，单LogStore就比多机的LogStore更好；
- 对于图结构数据，使用单LogStore会导致 fragmented storage





Fanned Updates

Fanned Updates 在节点/边第一次出现的shard上设置 update pointers，从而避免了访问所有的 shard 。

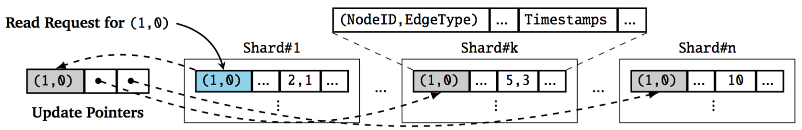


Figure 3: Update Pointers for the EdgeFile (§3.5).

Evaluation

Test Functions

- `assoc_range(id, atype, idx, limit)`
  从位置 idx 开始，获取不超过 limit 条源点为 id，类型为atype 的边
- `assoc_get(id1, atype, id2set, hi, lo)`
  获取所有的源点为 id1，类型为 atype，时间戳在 [hi, lo) 内，汇点在 id2set 中的边
- `assoc_time_range(id, atype, hi, lo, limit)`
  获取不超过源点为 id1，类型为 atype，时间戳在 [hi, lo) 内的边

Queries

Table 2: Queries in TAO [29] and LinkBench [24] workloads.

Query	Execution in ZipG	TAO %	LinkBench %
`assoc_range`	Algorithm 1	40.8	50.6
`obj_get`	`get_node_property`	28.8	12.9
`assoc_get`	Algorithm 2	15.7	0.52
`assoc_count`	`get_edge_record`	11.7	4.9
`assoc_time_range`	Algorithm 3	2.8	0.15
`assoc_add`	`append`	0.1	9.0
`obj_update`	`delete`, `append`	0.04	7.4
`obj_add`	`append`	0.03	2.6
`assoc_del`	`delete`	0.02	3.0
`obj_del`	`delete`	< 0.01	1.0
`assoc_update`	`delete`, `append`	< 0.01	8.0

Table 3: The `Graph Search` **Workload** and implementation using ZipG API; `p1` and `p2` are node properties, `id` and `type` are `NodeID` and `EdgeType`. All queries occur in equal proportion in the workload.

QID	Example	Execution in ZipG
GS1	All friends of **Alice**	`get_neighbor_ids(id,*,*)`
GS2	**Alice**'s friends in **Ithaca**	`get_neighbor_ids(id,*,{p1})`
GS3	**Musicians** in **Ithaca**	`get_node_ids({p1,p2})`
GS4	**Close** friends of **Alice**	`get_neighbor_ids(id,type,*)`
GS5	All **data** on **Alice**'s friends	`assoc_range(id,type,0,*)`

Datasets

Table 4: Datasets used in our evaluation.

	Dataset	#nodes & #edges	Type	On-disk Size
Real-world	orkut [41]	∼3M & ∼117M	social	20 GB
	twitter [28]	∼41M & ∼1.5B	social	250 GB
	uk [28]	∼105M & ∼3.7B	web	636 GB
LinkBench	small	∼32.3M & ∼141.7M	social	20 GB
	medium	∼403.6M & ∼1.76B	social	250 GB
	large	∼1.02B & ∼4.48B	social	636 GB

- For real-world datasets, Each node has an average propertyList of 640 bytes distributed across 40 propertyIDs. Each edge is randomly assigned one of 5 distinct EdgeTypes, a POSIX timestamp drawn from a span of 50 days, and a 128- byte long edge property.
- LinkBench models Facebook's database workload for social graph queries. For LinkBench datasets, these datasets mimic the Orkut, Twitter and UK graphs in terms of their total on-disk size. LinkBench assigns a single property to each node and edge in the graph, with the properties having a median size of 128 bytes.

Footprint

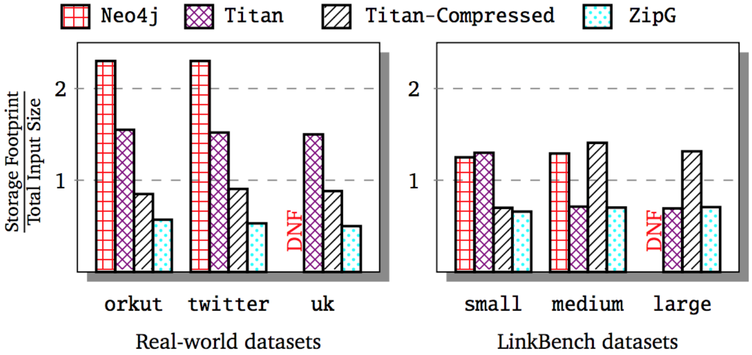


Figure 5: ZipG's storage footprint (§5.1) is 1.8−4× lower than Neo4j and 1.8−2× lower than Titan. DNF denotes that the experiment did not finish after 48 hours of data loading.

Single Server

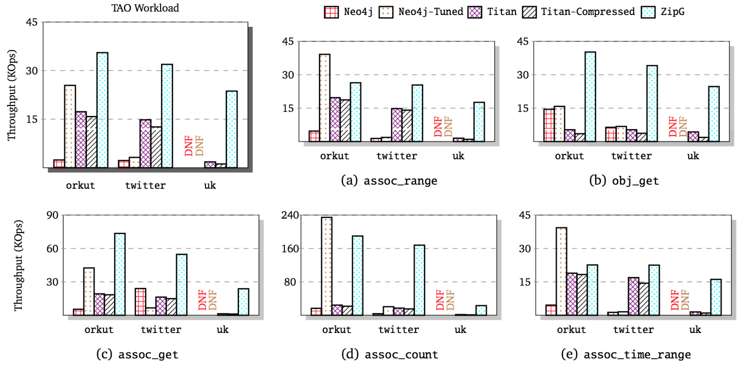


Figure 6: Single server throughput for the TAO workload, and its top 5 component queries in isolation. DNF indicates that that the experiment did not finish after 48 hours of data loading. Note that the figures have different y-scales.

Distributed Servers

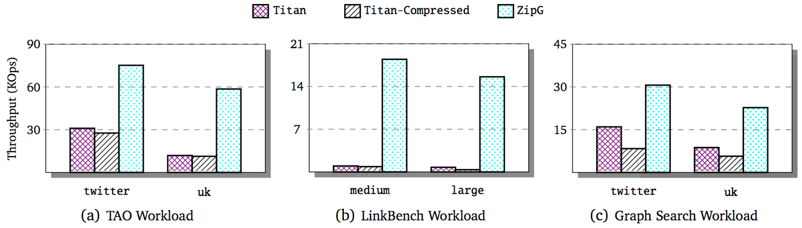


Figure 9: Throughput for TAO, LinkBench and Graph Search workloads for the distributed cluster.

Reference

Reference

- Boldi, Paolo and Vigna, Sebastiano *The webgraph framework I: compression techniques*. WWW, 2004
- Flavio SChierichetti, et al. *On compressing social networks.* SIGMOD, 2009
- Wenfei Fan, et al. *Query preserving graph compression.* SIGMOD, 2012
- Hernández C, et al. *Compression of web and social graphs supporting neighbor and community queries* SIGKDD, 2011
- Hernández C, et al. *Compressed representations for web and social graphs.* KAIS, 2014
- Maccioni A, Abadi D J. *Scalable pattern matching over compressed graphs via dedensification.* SIGKDD, 2016
- Maserrat H, Pei J. *Neighbor query friendly compression of social networks.* SIGKDD, 2010