ZipG: Serving Queries on Compressed Graphs

Paper #261, 15 Pages

Abstract

We present ZipG, a distributed graph store that serves queries on compressed graphs. ZipG exposes a minimal API that is functionally rich enough to implement all published functionalities from several industrial graph stores. As an example, we implement Facebook TAO graph store and Facebook graph search on top of ZipG. On a single server with 244GB main memory, ZipG executes all TAO and graph search queries for raw graph data over half a TB, achieving 3–10× higher throughput than Neo4j, the state-of-the-art single machine graph store. We get up to 5× gains in distributed settings compared to Titan, a distributed graph store used in several production clusters.

1 Introduction

Serving graph queries is challenging. Unlike batch processing jobs on graphs [27,28,33,35,40,41,49,50] that run at the scale of minutes or even hours, *user-facing* graph serving queries require millisecond level latency, as well as high throughput [47]. Achieving these goals is challenging because of two reasons.

The first reason is that graphs used for serving queries are massive, not only because of their sheer size but also because these graphs typically associate several *attributes* or *properties* with each node and/or edge [6, 15, 47]. For instance, Facebook stores a number of attributes (*e.g.*, age, location, birthplace) for each user node [47]. Thus, while the entire graph topology may fit on a single beefy server [35, 44], these properties result in graph serving systems operating on tens of terabytes or even petabytes of data [4, 47]. Indexes to serve queries further add to the overall graph size [2, 3, 6, 8–10, 13, 14, 16, 31, 42, 47].

The second reason is that graph queries are typically *complex and exhibit little locality*. For instance, consider the following query: "Find friends of Alice who live in Berkeley". One possible way to execute this query is to first execute two sub-queries — find friends of Alice, and, find people who live in Berkeley — and perform a join or an intersection of the corresponding results. Such joins are complex[1], incurring large computational and/or bandwidth[2] overheads [1]. Another possibility is to find friends of Alice, and check for each such friend whether or not the friend lives in Berkeley. Executing query in this manner alleviates the overheads of the join operation, but requires *random access* for the "location" attribute of each friend of Alice. Efficient partitioning of social graphs is a hard problem; thus, the query may touch arbitrary parts of the graph data potentially across multiple servers. Unless the attributes for each of Alice's friends is cached on these servers, the query latency and system throughput suffers. Thus, to avoid bandwidth-intensive joins and to maintain system performance, graph serving systems must cache as much data as possible.

One way to cache more data is to use compression. However, traditional block compression techniques (*e.g.*, gzip) are inefficient for graphs precisely due to the lack of locality problem discussed above — each query may require decompressing many blocks (*e.g.*, all blocks that contain Alice's friends in the above example). Due to these limitations, designing compression techniques that are specialized to graph queries has been an active area of research for the better part of last two decades [21–23, 26, 29, 30, 43, 50]. Several of these techniques even support queries directly on compressed graphs [22, 23, 26, 29, 30, 43]. However, all existing techniques are limited to simple queries like extracting the list of incident edges on a node, or identifying dense subgraphs. Graph serving requires executing far more complex queries [1, 4, 5, 18, 47, 53], often involving node and edge attributes.

This paper presents ZipG, the first distributed graph store that serves a wide range of graph queries directly on a compressed representation of the input graph data. ZipG exposes a minimal API that is functionally rich enough to implement published functionalities from several industrial graph stores including those from Facebook [47], LinkedIn [53] and Twitter [11]. We demonstrate this by implementing the entire Facebook TAO graph store [47] and Facebook graph search [18] on top of ZipG. Using a *single* server with 244GB memory, this implementation serves *tens of thousands of TAO and graph search queries for input graph data over half a TB.*

[1]Joins over graphs are referred to as "Join Bomb" by one of the state-of-the-art graph serving companies [1].

[2]Note that the cardinality of the results for the two sub-queries may be orders of magnitude larger than the final result cardinality.

ZipG builds upon Succinct [19], a system that supports random access and search queries on compressed data. We describe, in §3, several challenges that ZipG has to resolve to build an efficient graph serving system. The most fundamental of these is *to design a graph layout for efficiently storing and serving graph data*. Existing systems use layouts that expose a hard tradeoff between flexibility and scalability. On the one hand, systems like Neo4j [13] heavily use pointers to store neighbors and properties for each node and edge. While flexible in data representation, these systems suffer from scalability issues when the entire graph data does not fit into the memory of a single server[3]. Systems like Titan [16], on the other hand, scale well by using a layout that can be mapped to a key-value (KV) store abstraction. However, as identified in previous work [47], KV stores are inflexible for graph queries since these stores do not support random access *into* the "values". ZipG uses a new data layout that, while simple and intuitive, provides both scalability and flexibility by operating on flat unstructured files (§3).

ZipG makes two design choices to achieve above benefits. First, motivated by real-world workloads [47] where less than 0.05% of the queries perform in-place updates, ZipG trades off in-place update efficiency for faster reads, appends and deletes. Second, ZipG currently does not provide strong consistency and transaction guarantees. While several graph stores used in production [4,11,47,53] make a similar design choice, extending ZipG to provide such guarantees is an interesting future direction.

We evaluate ZipG against Neo4j [13] and Titan [16], two graph stores used in several production clusters. All our experiments run in the wild — a set of Amazon EC2 machines running Facebook-reported workloads [47] on real-world graphs containing millions of nodes and billions of edges. Our evaluation shows that ZipG significantly outperforms these systems both in terms of storage overhead and system throughput (sometimes by as much as an order of magnitude).

In summary, ZipG makes three main contributions:

- Design and implementation of a distributed graph store that serves queries directly on compressed graph data. Building on top of Succinct [19], ZipG achieves this using a new graph layout that is both scalable, and flexible to support graphs from various application domains.
- A minimal, yet functionally rich API for serving graph queries (§2). We demonstrate the expressiveness of ZipG's API by implementing the entire Facebook TAO graph store and Facebook graph search on top of ZipG.
- Evaluation (§6) against two popular state-of-the-art graph serving systems — Neo4j [13] and Titan [16] — for a wide variety of real-world graphs and workloads.

[3]Serving queries requires following pointers across secondary storage and/or different servers, leading to undesired bottlenecks.

2 Data model and Interface

We start by outlining ZipG graph data model (§2.1) and the interface exposed to the applications (§2.2).

2.1 ZipG Data Model

ZipG uses the property graph model [13, 15, 16, 47] that supports a flexible representation of graphs, consisting of nodes, edges, and a number of properties (or, attributes) associated with nodes and edges.

Nodes and Edges. ZipG uses usual definitions of nodes and edges. However, to model applications where edges may represent different kind of interactions (*e.g.*, comments, likes, relationships) [47], ZipG represents each edge using a 3-tuple comprising of `sourceID`, `destinationID` and an `EdgeType`, where the latter identifies the type of the edge[4]. Each edge can potentially have a different EdgeType and may optionally have a `Timestamp`. The precise representation of nodes, edges, EdgeType and Timestamps is described in §3.

Node and Edge Properties. Each Node and edge may have multiple properties, represented by `PropertyList`. A propertyList is a collection of (PropertyID, PropertyValue) pairs. For instance, the PropertyList for a node may be {(age, 20), (location, Berkeley), (nickname, cool)}. The PropertyLists in ZipG are schema flexible, and may have arbitrarily many properties.

2.2 ZipG Interface

ZipG exposes a minimal, yet functionally rich, interface that abstracts away the storage details (*e.g.*, compression). In this section, we outline this interface. We start with some definitions:

- `EdgeRecord`: An EdgeRecord holds a reference to all the edges of a particular EdgeType incident on a node and to the data corresponding to these edges (timestamps, destinationID, PropertyList, etc.).
- `TimeOrder`: EdgeRecord can be used to efficiently implement queries on a subset of edges. Many queries, however, are not edge-based but rather time-based (*e.g.*, find all comments since last login). For such queries, ZipG uses TimeOrder — for each node, the incident edges of the same type are logically sorted using timestamps. TimeOrder represents the order (*e.g.* *i*-th) of an edge within this sorted list.

[4]In the directed graph case, these are outgoing edges.

Table 1: ZipG's API and an example for each API. `EdgeType` can depict friendship, acquaintance, etc.. See §2.2 for description.

API	Example
`g = compress(graph)`	Compress **graph**.
`List<String> g.get_node_property(nodeID, propertyIDs)`	Get **Alice's age** and **location**.
`List<NodeID> g.get_node_ids(propertyList)`	Find people in **Berkeley** who like **Music**.
`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**.
`Pair<TimeOrder> g.get_edge_range(edgeRecord, tLo, tHi)`	§2.2
`EdgeData g.get_edge_data(edgeRecord, timeOrder)`	Find **Alice's most recent friend**.
`g.append(nodeID, PropertyList)`	Append new node for **Alice**.
`g.append(nodeID, edgeType, edgeRecord)`	Append new edges for **Alice**.
`g.delete(nodeID)`	Delete **Alice** from the graph.
`g.delete(nodeID, edgeType, destinationID)`	Delete **Bob** from **Alice's** friends list.

- `EdgeData`: Given the TimeOrder within an EdgeRecord, the EdgeData stores the triplet (destinationID, timestamp, PropertyList) corresponding to the edge.

Table 1 lists the interface exposed by ZipG, and an example for each individual query. The application submits the graphs (represented using the data model in §2.1) to ZipG, and compresses the graph using `compress(graph)`. Once compressed, the application can invoke a number of powerful primitives (Table 1). What makes ZipG unique is that while the applications execute these queries as if the graph was uncompressed, ZipG internally executes queries efficiently directly on the compressed representation. Most of the queries in Table 1 are self-explanatory; we discuss some of the interesting aspects below, and return to implementation of these queries in §4.

Wildcards. ZipG queries admit *wildcard* as an argument for `PropertyID`, `edgeType`, `tLo`, `tHi` and `timeOrder`. ZipG interprets wildcards as admitting any possible value. For instance, `get_node_property(nodeID,*)` returns all properties for the node, and `get_edge_record(nodeID,*)` returns all edgeRecords for the node (and not just of a particular edgeType).

Node-based queries. Consider again the query "Find friends of Alice who live in Berkeley". Assuming Alice is NodeID and friends has edgeType `0`, the query `get_neighbor_ids(Alice,0,{Location, Berkeley})` returns the desired results. Internally, ZipG implements this query by first finding Alice's friends, and then checking for each of the friends, whether or not the friend lives in Berkeley (that is, ZipG avoids joins). Note that if the application has knowledge about the structure of the graph and/or queries, it can also perform a join-based implementation of the same query — using `get_neighbor_ids(Alice,0,`$\star$`)` $\cap$ `g.get_node_ids(Location, Berkeley)`, where the former returns all friends of Alice and the latter returns all people who live in Berkeley.

Edge-based queries and Updates. ZipG allows applications to get random access to any EdgeRecord using `get_edge_record`, and, into the data for any specific edge in EdgeRecord using `get_edge_data`. If edges contain timestamps, ZipG also allows applications to access edges based on timestamps. Finally, the application can append new EdgeRecords using `append` or delete existing EdgeRecords using `delete`. Note that in-place updates within an EdgeRecord can be implemented using a `delete` followed by an `append`.

3 ZipG Layout

We describe the first of the two core contributions of ZipG: a graph layout that, while simple and intuitive, provides the scalability and flexibility for graph queries. We start by briefly outlining the necessary details from Succinct [19] that ZipG builds upon; this allows us to discuss the various design tradeoffs in ZipG layout (§3.1). We then provide a high-level overview of ZipG layout (§3.2), and then dive deep into the layout (§3.3, §3.4, §3.5).

3.1 Succinct Background

Succinct [19] is a distributed data store that supports queries (discussed below) on a compressed representation of the input data. Succinct exposes several interfaces to the applications: a flat file interface for executing queries on unstructured data, a key-value store and a NoSQL store interface for queries on semi-structured data, and a table interface for queries on more structured data.

- **Random access** via `extract(offset, len)` query on flat files, that returns `len` many characters starting at arbitrary `offset`; for KV, NoSQL and table interface, the standard `get(recordID)` interface for random access returns the corresponding record.
- **Search** via `search(val)` query on flat files that returns offsets where the file contains string "val"; for KV, NoSQL and table interface, recordIDs whose record contain string "val" are returned.

The description of data structures and query algorithms used in Succinct is not required to keep the paper self-contained; we refer the reader to [19].

3.2 ZipG overview

It is a priori conceivable that most of the "heavy lifting" in building a distributed compressed graph store could be delegated to Succinct [19]. However, ZipG has to resolve a number of fundamental challenges to be able to achieve the desired flexibility, scalability and performance. We outline some of these challenges below, and provide a brief overview of how ZipG resolves these challenges.

Storing graphs. Perhaps the most fundamental challenge that ZipG has to resolve relates to using the underlying data store in the most efficient manner. This is akin to, for example, GraphChi [35] and Ligra [49] that use new graph layouts for efficiently utilizing the underlying storage systems for batch processing of graphs. In the particular case of serving queries, previous work [47] has already argued that KV and NoSQL interfaces are inefficient since these typically provide no "random access" into the records; the case for ZipG is no different.

Rather non-intuitively, we show that a flat (unstructured) file interface could be used in an efficient manner while providing a flexible graph representation. To that end, ZipG layout for graphs uses two flat files:

- **NodeFile** stores all the NodeIDs and corresponding node properties. NodeFile adds a small amount of metadata to the list of (NodeID, nodeProperties) before invoking compression. This metadata allows ZipG to tradeoff storage (in uncompressed representation) for efficient random access into node properties. We discuss NodeFile design and associated tradeoffs in §3.3.
- **EdgeFile** stores all the EdgeRecords. The design of EdgeFile is crucial to ZipG performance — by adding metadata and by converting variable length data into fixed length data before invoking compression, ZipG EdgeFile trades off storage (in uncompressed representation) to optimize random access into EdgeRecords and more complex operations like binary search over timestamps and ordered access over the set of edges. We provide an in-depth discussion on EdgeFile design and associated tradeoffs in §3.4.

Updating graphs. Another important challenge that ZipG has to resolve relates to updating the underlying graph data. In particular, for sharding schemes typically used in KV and NoSQL stores, Succinct queries (*e.g.*, "find all documents that contain Berkeley") may touch all the servers in the worst-case. Thus, new data updates could be sent to any server since this does not impact the query latency and/or system throughput. In contrast, graph queries (*e.g.*, "find friends of Alice who live in Berkeley") have more structure; indeed, most queries are node-based and in the absence of updates, the queries need to touch only those servers where the data for the queried node and its neighbors resides. Presumably, this is a very small subset of servers for most queries. ZipG uses the idea of `Fanned Updates` to ensure that the queries touch minimal number of servers even in presence of graph updates (§3.5).

Querying graphs. Graph queries are typically much more complex than those encountered in KV and NoSQL stores (*e.g.*, `random access` and `search` in Succinct [19]). ZipG uses the idea of `Function Shipping` to efficiently implement graph queries (§4, §5).

3.3 NodeFile

NodeFile stores all the NodeIDs and the associated properties, and is optimized for two kind of queries on node data: (1) given a (NodeID, List<propertyID>) pair, extract the corresponding propertyValues; and (2) given a PropertyList, find all NodeIDs whose properties match the propertyList. The NodeFile consists of three data structures (see Figure 1 for a simple example).

First, each propertyID in the graph is assigned a unique delimiter[5] and stored as a **PropertyID $\rightarrow$ (order, delimiter) map**, where `order` is the ranking of the propertyID in the lexicographically sorted list of all propertyIDs. This map is stored in memory.

The second data structure is a flat (unstructured) file that stores PropertyLists as follows. In sorted order of propertyIDs, the corresponding propertyValue is first prepended by the propertyID's delimiter and then written in the flat file; if a propertyID has a null propertyValue, we

[5]Graphs usually have a small fixed number of propertyIDs; ZipG currently uses one byte non-printable characters as delimiters that are enough to represent tens of propertyIDs, but could be easily extended to use two byte delimiters to represent thousands of propertyIDs.

NodeID	PropertyList
Alice	nickname: Ally, age: 42, location: Berkeley
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†Berkeley⋆Ally‡
095•†Princeton⋆Bobby‡
203•24†⋆Cat‡

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

simply write down the delimiter. An end-of-record delimiter is appended to the end of the serialized propertyList of each node. For instance, Alice's propertyValues in Figure 1 are serialized into •42†Berkeley⋆Ally‡, where ‡ is the end-of-record delimiter.

Note that the size of propertyValues within the propertyList of a node may vary significantly (*e.g.*, Alice's age and job description). Thus, having a fixed size representation for propertyValues (efficient for forward mapping) leads to space inefficiency. On the other hand, naïvely using a space-efficient variable size representation leads to inefficient forward mapping — Alice may put her age, name, nickname, location, status, workplace, etc. and accessing status may require extracting many more bytes than necessary. To that end, ZipG uses variable size representation for propertyValues but also explicitly stores the length of each propertyValue into the metadata for each propertyList. The lengths of propertyValues are encoded using a global fixed size `len`, since they tend to be short and of nearly similar size. In the example of Figure 1, the propertyList for Alice is thus encoded as 284•42†Berkeley⋆Ally‡.

The third data structure is a two-dimensional array that stores a sorted list of NodeIDs and the offset of node's PropertyList and the metadata in NodeFile.

Implementing `get_node_property` from Table 1. Suppose the order of a given PropertyID is `idx` and denote the value of i-th `len` by $\texttt{len}_\texttt{i}$. Then, using the `random access` functionality from §3.1, the corresponding PropertyValue can be accessed directly by extracting $\texttt{len}_\texttt{idx}$ characters starting at offset $\sum_{i<idx} \texttt{len}_\texttt{i}$ within the PropertyList. This requires storing and extracting a few extra bytes for $\texttt{len}_\texttt{i}$, but offers a sweet spot between storage and efficiency of random access into PropertyValues in ZipG NodeFile.

Implementing `get_node_ids` from Table 1. We explain this using an example. Suppose the query specifies `{"nickname" = "Ally"}` as the propertyList. Then, ZipG first finds the delimiter of the specified PropertyID (⋆, for `nickname`) and the next lexicographically larger PropertyID (in this case, ‡ for end-of-record delimiter). It then prepends and appends `Ally` by ⋆ and ‡, respectively and uses the search primitive from §3.1. This returns the offsets into the flat file where this string occurs, which are then translated into NodeIDs using binary search over the offsets in the two-dimensional array.

3.4 EdgeFile

EdgeFile stores the set of edges and their associated properties. Recall from §2 that each edge is uniquely identified by the 3-tuple `(sourceNodeID,EdgeType,destinationNodeID)` and may have an associated timestamp and a list of properties. See Figure 2 for an illustration.

Each EdgeRecord in the EdgeFile corresponds to the set of edges of a particular EdgeType incident on a NodeID. The EdgeRecord for (NodeID, EdgeType) pair starts with `$NodeID#EdgeType`, where $ and # are two delimiters. Next, the EdgeFile stores certain metadata that we describe below. Following the metadata, the EdgeFile stores the TimeStamps for all edges, followed by destinationIDs for all edges, finally followed by the PropertyLists of all edges. We describe below the design decisions made for each of these individually.

Edge Timestamps. Edge timestamps are often used to impose ordering among the edges (*e.g.*, return results sorted by timestamps, or find new comments since last login time [47]). Efficiently executing such queries requires the ability to perform binary search on timestamps. To that end, ZipG stores the timestamps in each EdgeRecord in sorted order. ZipG also stores the number of edges (and hence, the timestamps) in the EdgeRecord within the metadata (denoted by `EdgeCount` in Figure 2).

There are several approaches for storing individual timestamp values, each presenting a different tradeoff. At one extreme are variable length representation and delta encoded representation. In the former, each timestamp can be stored using minimum number of bytes required to represent that timestamp along with some additional bytes (delimiters and/or length) to mark boundaries of timestamp values. While space-efficient, this representation makes random access on timestamps complex since extracting a timestamp requires extracting all the timestamps before it. Delta encoding [45] typically used for storage-efficient representation of sorted integers also leads to a similar tradeoff. At the other extreme is fixed

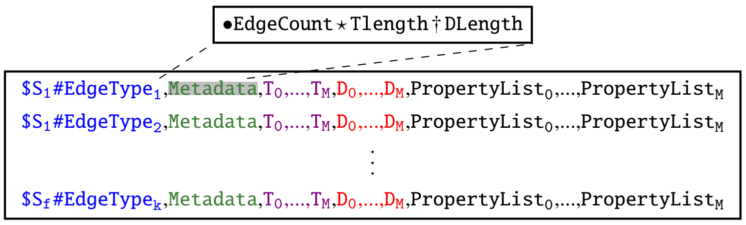

•EdgeCount ⋆Tlength †DLength
$S₁#EdgeType₁,Metadata,T₀,...,T_M,D₀,...,D_M,PropertyList₀,...,PropertyList_M
$S₁#EdgeType₂,Metadata,T₀,...,T_M,D₀,...,D_M,PropertyList₀,...,PropertyList_M
$S_f#EdgeType_k,Metadata,T₀,...,T_M,D₀,...,D_M,PropertyList₀,...,PropertyList_M

Figure 2: EdgeFile Layout in ZipG (§3.4). Each row is an EdgeRecord for a `(nodeID, 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.

length representation across all edge timestamps (*e.g.*, 64 bits) that enables efficient random access at the cost of increased storage.

ZipG uses a middle-ground: it uses a fixed length representation but rather than using a *globally* fixed length, it uses the maximum length required across all edges corresponding to a (sourceID, EdgeType) pair. Since this length varies across (sourceID, EdgeType) pairs, ZipG stores the fixed length for each such pair in the corresponding metadata; `TLength` in Figure 2.

DestinationIDs. A natural choice for the layout for the destinationIDs is to order them according to edge timestamps, such that the ith timestamp and ith destinationID correspond to the same edge. Such an ordering avoids the need to maintain an explicit mapping between edge timestamps and the corresponding destinationIDs, enabling efficient random access. ZipG uses a fixed length representation similar to timestamps for destinationIDs; this width may be different from the timestamp width and is stored in the metadata as `DLength` (see Figure 2).

Edge Properties. As with the destinationIDs, edge propertyLists are ordered such that ith timestamp and ith propertyList correspond to the same edge. The edge properties within a propertyList are encoded similar to node properties, since the layout design criteria and tradeoffs for both are identical. Specifically, the lengths of all the propertyLists are stored, followed by delimiter separated propertyValues as in §3.3. Note that ZipG currently does not support search on edge propertyLists, but can be trivially extended to do so using ideas similar to NodeFile at the cost of some additional storage overhead.

Implementing Edge-based queries (Table 1). An `EdgeRecord` for a given sourceID and edgeType is obtained by performing `search($sourceID#edgeType)` on the EdgeFile to get the offset for the record, and extracting the metadata at the offset (i.e., `get_edge_record`). Using the metadata in the EdgeRecord, ZipG can efficiently perform binary searches on the timestamps (`get_time_range`) and random access into the destination IDs and edge properties (`get_edge_data`) as described above.

3.5 Fanned Updates

ZipG uses a log structured storage approach, similar to SILT [39] and Succinct [19], to efficiently implement node and edge updates. Specifically, new nodes and edges are appended into a query-optimized LogStore, which is periodically merged into a memory-optimized data store. Deletes are implemented as lazy deletes with a bitmap indicating whether or not a node or an edge has been deleted; finally, in-place updates are implemented as deletes followed by an append. While several modern KV and NoSQL stores use the above design for updates, as outlined in §3.2, ZipG has to resolve a number of challenges to efficiently use such a design for graph stores.

The main challenge arises due to fragmentation — as LogStore data is merged into memory-optimized data store, the data corresponding to a node and its incident edges will be fragmented across multiple servers. ZipG queries, that could previously be restricted to a small subset of servers, would now require touching *all* servers leading to significant reduction in system throughput. ZipG resolves this challenge using `Fanned Updates`, which we describe next.

Fanned Updates (Figure 3). Consider a static graph, that is, a graph that has never been updated since the initial upload to ZipG. The sharding scheme used in ZipG (described in §5) ensures that most ZipG queries are first forwarded to a single server[6]. At a high-level, Fanned Updates avoid touching all servers using a set of update pointers that logically *chain* together data correspond to the same node or edge. Specifically, update pointers store a reference to the offsets of NodeFile and/or EdgeFile at other servers that store the updated data. As the graph is updated or is defragmented over time (§5), these update pointers are updated as well. Since updates form a small fraction of real-world graph workloads [47], the overhead of storing and updating these pointers is minimal. ZipG, thus, keeps these pointers uncompressed.

[6]Most ZipG queries in Table 1 are node-based and are first forwarded to the server that store queried node's data. Some of these queries may then be forwarded to servers that store node's neighbors's data. The only exception is `get_node_ids` that requires touching all servers.

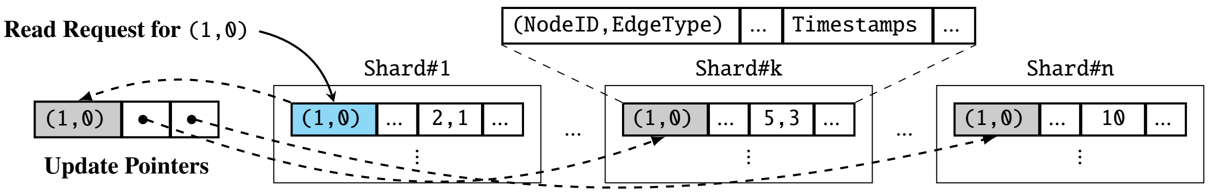


Figure 3: Update Pointers for the EdgeFile (§3.5).

Table 2: Read and write queries that have $> 0.1\%$ frequency in Facebook's TAO workload [47].

TAO Query	Execution in ZipG	%
`assoc_range`	Algorithm 1	40.8
`obj_get`	`get_node_property`	28.8
`assoc_get`	Algorithm 2	15.7
`assoc_count`	`get_edge_record`	11.7
`assoc_time_range`	Algorithm 3	2.8
`assoc_add`	`append`	0.10

Algorithm 1 `assoc_range(id, atype, idx, limit)`
Obtain at most `limit` edges with source node `id` and edge type `atype` ordered by timestamps, starting at index `idx`.

```
1: rec ← get_edge_record(id, atype)
2: results ← ∅
3: for i ← idx to idx+limit do
4:     edgeEntry ← get_edge_data(rec, i)
5:     Add edgeEntry to results
6: end for
7: return results
```

Read Queries with Fanned Updates. Fanned Updates require minimal extension to ZipG query execution for static graphs. In addition to executing query as in a static graph, the ZipG servers also forward the query to the precise servers that store updated data (using update pointers), and collect the additional query results while avoiding touching all servers.

4 Facebook TAO and Graph Search

ZipG design, abstractions and interface are rich enough to implement functionalities from all published industrial graph stores. In fact, we have implemented the entire Facebook TAO [47] functionality as well as Facebook Graph Search [12] functionality on top of ZipG. In this section, we discuss this implementation and associated tradeoffs. Table 2 maps the TAO read operations to their execution in ZipG.[7]

[7]ZipG's definitions of nodes and edges are equivalent to TAO's *objects* and *associations*; for instance, `EdgeType` in ZipG translates to `atype` (or association type) in TAO.

Algorithm 2 `assoc_get(id1, atype, id2set, hi, lo)`
Obtain all edges with source node `id1`, edge type `atype`, timestamp in the range `[hi,lo)`, and destination ∈ `id2set`.

```
1: rec ← get_edge_record(id1, atype)
2: (beg, end) ← get_time_range(rec, hi, lo)
3: results ← ∅
4: for i ← beg to end do
5:     edgeEntry ← get_edge_data(rec, i)
6:     Add edgeEntry to results if destination ∈ id2set
7: end for
8: return results
```

Facebook TAO queries are of two types. First, those that do not operate on Timestamps (`obj_get` and `assoc_count` in Table 2). These queries translate to obtaining all properties for a NodeID and counting edges of a particular type incident on a given NodeID. These are easily mapped to ZipG API — `get_node_property(id, *)` and the `EdgeCount` metadata using `get_edge_record`, respectively.

The second kind of queries are based on Timestamps. For instance, the query "find all comments posted by Alice between OSDI abstract and paper submission deadlines that must be visible to Bob" essentially translates to `assoc_get` query in TAO. ZipG is particularly efficient for these kind of queries, since as discussed in §3, ZipG can efficiently perform binary search on timestamps and return corresponding edges and their properties. Algorithms 1, 2 and 3 show that these fairly complicated TAO queries can be implemented in ZipG using less than 10 lines of code.

Facebook Graph Search originally supported interesting, and complex, queries on graphs [1,18]. Implementing graph search queries is even simpler in ZipG since most queries directly map to ZipG API, as shown in Table 4.

5 System Implementation

We outline the key aspects of ZipG implementation, which is done in C++ using roughly 4000 lines of code.

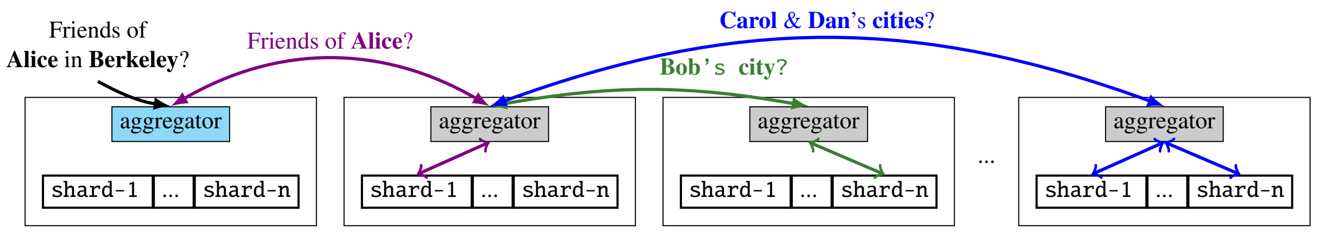


Figure 4: Function Shipping in ZipG. See §5 for discussion.

Algorithm 3 `assoc_time_range(id, atype, hi, lo, limit)`
Obtain at most `limit` edges with source node `id`, edge type `atype` and timestamps in the range `[hi,lo)`.

```
1: rec ← get_edge_record(id, atype)
2: (beg, end) ← get_time_range(rec, hi, lo)
3: results ← ∅
4: for i ← beg to min(beg+limit, end) do
5:     edgeEntry ← get_edge_data(rec, i)
6:     Add edgeEntry to results
7: end for
8: return results
```

Graph Partitioning (Sharding). Several previous studies have established that efficient partitioning of social graphs is a hard problem [20, 37, 38]. Similar to existing graph stores [16], ZipG uses a simple hash-partitioning scheme — it creates a number of shards, default being one per core, and hash partitions the NodeIDs on to these shards. All the data corresponding to NodeID (PropertyList and edge information of edges incident on NodeID) are then stored in that shard. This ensures that all node and edge data associated with a particular node's neighborhood are co-located on the same shard, enabling execution of complex neighborhood queries *locally*, without requiring any communication. Finally, each of the shards is transformed into the ZipG layout (§3).

Fault Tolerance and Load Balancing. ZipG currently implements traditional replication-based techniques for fault tolerance; an application can specify the desired number of replicas per shard. Queries are load balanced evenly across multiple replicas. While orthogonal to ZipG design, extending the current implementation to incorporate more storage-efficient fault tolerance and skew-tolerant load balancing techniques [24, 34, 48] is an interesting future direction.

Data Persistence and Caching. To achieve data persistence, ZipG stores NodeFiles, EdgeFiles, newly added data on LogStore and the update pointers on secondary storage as serialized flat files. ZipG maps these files to virtual memory using the `mmap` system call, and all writes to them are propagated to the secondary storage before the operation is considered complete.

Query Execution via Function Shipping (Figure 4). Graph queries often require exploring the neighborhood of the queried node (*e.g.*, "friends of Alice who live in Berkeley"). To minimize network roundtrips and bandwidth in a distributed setting, ZipG pushes computation closer to the data via *function shipping* [7, 51, 52]. Each ZipG server hosts an *aggregator* process that maintains a pool of local threads for executing queries on the server. When an aggregator receives a query that comprises of subqueries to be executed on other servers, it *ships* the subqueries to the corresponding servers, each of which execute the subquery locally. Once all the subquery results are returned, the aggregator computes the final result.

Concurrency Control. Having a log structured store for data updates significantly simplifies concurrency control in ZipG. The compressed data structures are immutable (except periodic garbage collection) and see only read queries; locks are only required at uncompressed update pointers and deletion bitmaps (§3.5), that are fast enough and do not become system bottleneck at write rates encountered in real-world workloads.

Data consistency and Transactions (Limitation). Storing the data in a compressed form presents new challenges in terms of data consistency and transactional workloads. ZipG currently does not provide strong consistency guarantees and does not support transactional workloads. We note that some recent systems have explored efficient transactional workloads for graph queries(*e.g.*, Weaver [25]) and achieve extremely high system throughput compared to state-of-the-art. It would be interesting to incorporate ZipG techniques into Weaver [25] and explore whether graph stores can achieve both the benefits of compression and support for transactional workloads.

6 Evaluation

We now evaluate ZipG against popular open-source graph stores across a variety of real-world graphs, real-world and synthetic query workloads, and cluster sizes.

Compared Systems. We compare ZipG against two popular open source graph stores. Neo4j [13] is a single machine graph store and does not support distributed implementations. We use version 2.2.2. Our preliminary results for Neo4j were not satisfactory. We worked with Neo4j engineers for over a month to tune Neo4j and made several improvements in Neo4j query execution engine. Along with the original version (Neo4j), we also present the results for this improved version (Neo4j-Tuned).

We also compare ZipG against Titan, a popular open-source distributed graph store that requires a separate storage backend. We use Titan version 0.5.4 [16] with Cassandra 2.2.1 [36] as the storage backend. We also experimented with DynamoDB for Titan but found it to be performing worse. Titan supports compression. We present results for both uncompressed (Titan) and compressed (Titan-compressed) representations.

We configure all systems to run without replication.

Experimental Setup. We run all our experiments in the wild — on an Amazon EC2 cluster. To compare against Neo4j, we perform single machine experiments over an r3.8xlarge instance with 244GB of RAM and 32 virtual cores. Our distributed experiments use 10 m3.2xlarge instances each with 30GB of RAM and 8 virtual cores. Note that all instances were backed by local SSDs and *not* hard drives. We warm up each system for 15 minutes prior to running experiments to cache as much data as possible.

Table 3: Datasets used in our evaluation; sizes correspond to on-disk footprint for the augmented graphs.

Dataset	#nodes & #edges	Type	Size
`orkut` [32]	$\sim$ 3M & $\sim$ 117M	social	20 GB
`twitter-2010` [22]	$\sim$ 41M & $\sim$ 1.5B	social	250 GB
`uk-2007-05` [22]	$\sim$ 105M & $\sim$ 3.7B	web	636 GB

Datasets. Table 3 shows the three real-world datasets used in our evaluation. Unfortunately, we did not have access to datasets tagged with attributes. To that end, we used the attribute distribution from Facebook TAO paper [47] to add attributes to our datasets. Each node has an average propertyList of 640 bytes distributed across 40 propertyIDs. Each edge is randomly assigned one of 5 distinct `EdgeType`s, a POSIX timestamp drawn from a span of 50 days, and a 128-byte long edge property.

Workloads. We use the real `TAO` workload (Table 2) and a synthetic `Graph Search` workload (Table 4) for our evaluation. The query distribution for the TAO workload is from the TAO paper [47]; for the graph search workload, we simply use a uniform distribution. In addition, we also evaluate the performance of each query in isolation to build more in-depth insights on the performance of the three systems.

Table 4: The `Graph Search` Workload. Individual queries are mapped to their implementation using ZipG API (Table 1); `p1` and `p2` correspond to node properties, while `id` and `type` correspond to `NodeID` and `EdgeType` respectively. All queries occur in equal proportion in the workload.

QID	Example	Implementation in ZipG
GS1	All friends of **Alice**	`get_neighbor_ids(id,*,*)`
GS2	**Alice**'s friends in **Berkeley**	`get_neighbor_ids(id,*,{p1})`
GS3	**Graduate students** in **Berkeley**	`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,*)`

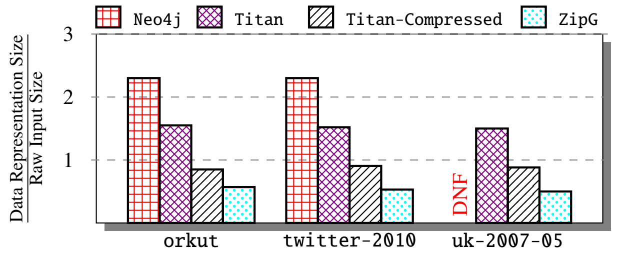


Figure 5: Comparison of storage footprint (§6.1). ZipG's storage footprint is 4× lower than Neo4j and 1.8× lower than Titan. `DNF` denotes that the experiment did not finish after 48 hours of data loading.

6.1 Storage Footprint

Figure 5 shows the ratio of total data representation size and the raw input size for each system. We note that ZipG can put $2.7-4\times$ larger graph sizes in main memory compared to Neo4j and Titan, and $1.7\times$ larger graph sizes compared to Titan compressed (which, as we show later, leads to degraded performance[8]). The main reason is the secondary indexes stored by Neo4j and Titan to support various queries efficiently; ZipG, on the other hand, executes queries efficiently directly on compressed graphs as discussed in §3.1.

6.2 TAO Workload

Figure 6 and Figure 7 compare the performance of the three systems for single machine and distributed setups for the TAO workload from Table 2 and for individual

[8] Intuitively, Titan uses delta encoding for edge destinationNodeIDs, and variable length encoding for node and edge attributes [17] which leads to high CPU overhead during query execution. Moreover, enabling LZ4 compression for Cassandra's SSTables reduces the storage footprint for Titan, but required data decompression for query execution.

Table 5: Summary of which graph datasets fit completely in memory for different systems.

In memory?	`Neo4j`	`Titan-C`	`Titan`	`ZipG`
`orkut`	✓	✓	✓	✓
`twitter`		✓		✓
`uk`				

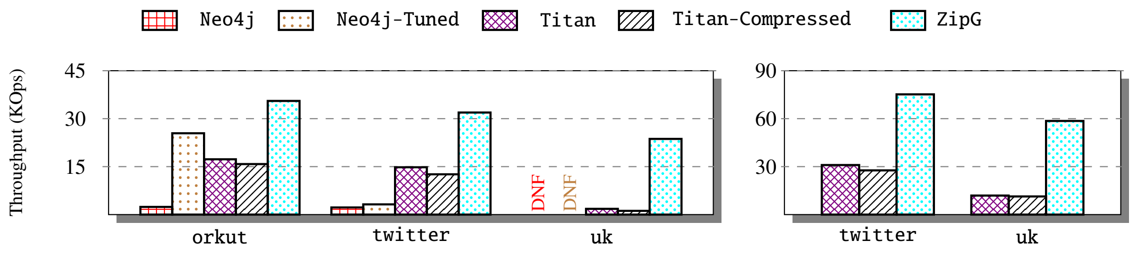
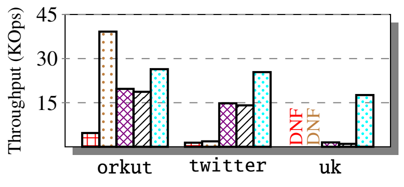
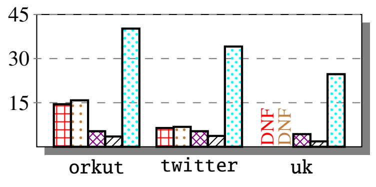


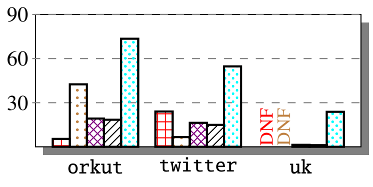
Figure 6: Throughput for TAO workload (§6.2). DNF denotes that the experiment did not finish after 48 hours of data loading. Note that the figures have different scales for the y-axis.



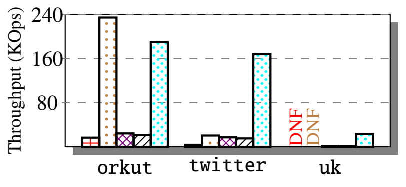
(a) assoc_range



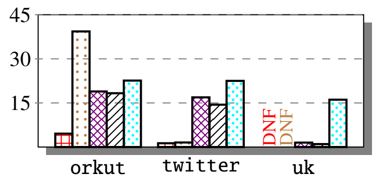
(b) obj_get



(c) assoc_get



(d) assoc_count



(e) assoc_time_range

Figure 7: Throughput for isolated TAO queries (§6.2). DNF indicates that that the experiment did not finish after 48 hours of data loading. Note that the figures have different y-scales.

queries in TAO workload, respectively. We start by noting that, across all results, Neo4j-Tuned achieves strictly better performance than Neo4j. Similarly, Titan uncompressed achieves strictly better performance Titan compressed (for reasons discussed in Footnote 8). We hence focus on Neo4j-Tuned, Titan uncompressed and ZipG in the following discussion.

Single machine (Figure 6(left)). We start by observing that when the entire uncompressed dataset fits in memory (*e.g.*, Orkut), all systems achieve comparable performance. There are two reasons for ZipG achieving slightly better performance than Neo4j and Titan. First, ZipG is optimized for random access on node PropertyList (§3) while Neo4j and Titan have to perform quite a bit of extra work to extract node properties — Neo4j requires following a set of pointers on Node Table, while Titan needs to first extract the corresponding (key, value) pair from Cassandra and then scan the value to extract node properties. This leads to ZipG achieving significantly high throughput for the obj_get query; see Figure 7(b). The second reason ZipG performance is slightly better is that ZipG extracts all edges of a particular edgeType directly, while other systems have to scan the entire set of edges and filter out the relevant results. This leads to improved ZipG throughput for the assoc_get query; see Figure 7(c). When queries have a limit on the result cardinality, other systems can stop scanning earlier and thus achieve relatively improved performance, *e.g.*, queries assoc_range and assoc_time_range in Figure 7(a) and 7(e).

For the Twitter dataset, Neo4j can no longer keep the entire dataset in memory; Titan, however, retains most of the working set in memory due to its lower storage overhead than Neo4j and also because TAO queries do not operate on edge PropertyList. Neo4j observes significant impact in throughput for a reason that highlights the limitations of pointer-based data model of Neo4j — since pointer-based approaches are "sequential" by nature, a single application query leads to multiple SSD lookups leading to significantly degraded throughput compared to Orkut dataset for all queries (see Figure 7). Titan, on the other hand, maintains its throughput for all queries. This

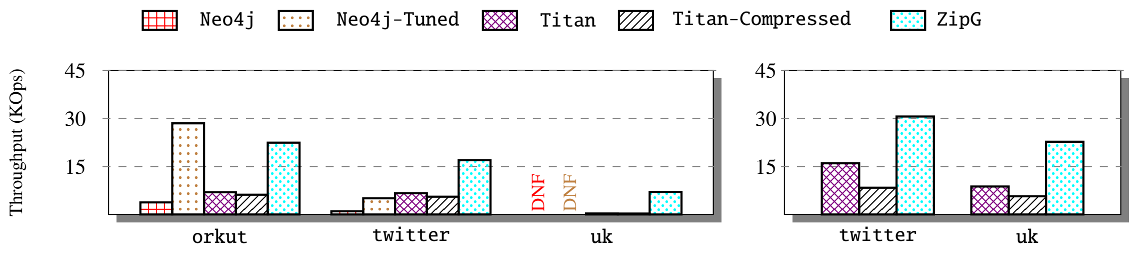


Figure 8: Throughput for `Graph Search` workload (§6.3). `DNF` denotes that the experiment did not finish after 48 hours of data loading.

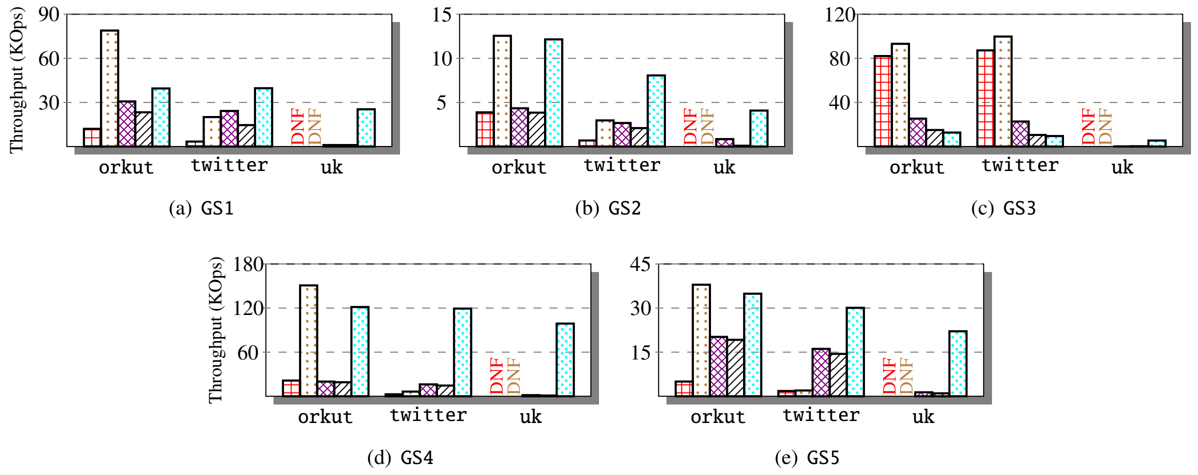


Figure 9: Throughput for isolated `Graph Search` queries on a single server (§6.3). `DNF` indicates that that the experiment did not finish after 48 hours of data loading. Note that the figures have different y-scales.

is both because Titan has to do fewer SSD lookups (once the key-value pair is extracted, it can be scanned in memory) and also because Titan essentially caches most of the working dataset in memory.

For the UK dataset, none of the systems can fit the data in memory (Neo4j cannot even scale to this dataset size). Titan now starts experiencing significant performance degradation due to a large fraction of queries being executed off secondary storage (similar to the performance degradation of Neo4j in Twitter dataset). ZipG also observes performance degradation but of much lesser magnitude than other systems because of two reasons. First, ZipG is able to execute a much larger fraction of queries in memory due to its lower storage overhead; and second, even when executing queries off secondary storage, ZipG has significantly lower I/O since it requires a single SSD lookup for all queries unlike Titan and Neo4j.

Distributed Cluster (Figure 6 (right)). Neo4j does not have a distributed implementation. We now discuss the performance of ZipG and Titan in a distributed setting. We make two observations. First, Titan in distributed settings can fit the entire Twitter dataset in memory. This leads to roughly $2\times$ higher throughput for Titan despite the increased overhead of inter-server communication. Moreover, Titan can fit more of UK dataset in memory again leading to improved throughput. The second observation is that ZipG achieves roughly $2.5\times$ higher throughput in distributed settings compared to single server setting. Note that our distributed servers have 10×8 cores, $2.5\times$ of the single beefy server that has 32 cores. ZipG thus achieves throughput increase proportional to the increase in number of cores in the system, an ideal scenario.

6.3 Graph Search Workload

We designed the graph search workload for two reasons. First, while TAO workload is mostly random access based, graph search workload mixes random access (`GS1, GS4, GS5`) and search (`GS2, GS3`) queries. Second, this workload highlights both the power and overheads of ZipG. In particular, as shown in Table 4, ZipG's powerful API enables simple implementation of queries that are far more complex than the TAO queries. Indeed, most of the graph search queries can be implemented using a couple of lines of code on top of ZipG API. On the flip side, the graph search workload also highlights the overheads of executing queries on compressed graphs. We discuss the latter below.

High-level discussion. The results for the graph search workload (Figure 8) follow a very similar pattern as for TAO workload (Figure 6). The patterns are similar in that (1) for single machine experiments, Neo4j-Tuned and ZipG consistently achieve higher performance than native Neo4j and Titan; (2) Titan uncompressed achieves higher performance than Titan compressed; (3) ZipG achieves similar performance improvements over Titan in single machine and distributed settings; and (4) as the graph size increases, Neo4j and Titan observe significantly higher performance degradation than ZipG.

However, there are two main differences in TAO and graph search results. First, the overall throughput reduces for all systems. This is rather intuitive — search queries are usually far more complex than random access queries, and hence have higher overheads. And second, when the uncompressed graph fits entirely in memory, Neo4j-Tuned achieves better performance than ZipG. We discuss this in more depth below.

Diving deeper. The main difference between TAO and graph search workload is that the latter helps us expose the overheads of executing queries on compressed data. In particular, for the Orkut dataset, both Neo4j-Tuned and ZipG fit the entire data in memory. However, in graph search workload, Neo4j could use its indexes to answer search queries (and avoid heavy-weight neighborhood scans). As a result, for the Orkut dataset, Neo4j starts observing roughly 1.23× higher throughput than ZipG as opposed to lower throughput for TAO queries which is attributed to ZipG executing queries on compressed graphs. Of course, as the graph size increases, the overhead of executing queries off secondary storage becomes higher than executing queries on compressed graphs leading to ZipG achieving 3× higher throughput than Neo4j-Tuned.

The second overhead of ZipG is highlighted in Figure 9(c) for queries like "Find graduate students in Berkeley". For such a query, ZipG's partitioning scheme requires ZipG touching all servers. Neo4j and Titan, on the other hand, use global indexes and thus require touching no more than two servers. Thus, for small datasets, ZipG observes significantly lower throughput for this query than Neo4j and Titan. As earlier, for larger graph sizes, this overhead becomes smaller than the overhead of executing queries off secondary storage and ZipG achieves higher throughput.

Overall, we conclude that ZipG achieves higher throughput than existing open-source graph stores for both TAO and graph search workloads even for moderate size graph data.

7 Related Work

Graph Stores. In the recent past, there has been significant effort towards the design of Graph Stores [2, 3, 6, 8–10, 13, 14, 16, 25, 31, 42, 46, 47] for serving queries on data naturally modelled as graphs. While a majority of them adopt the Property Graph Model [15], they vary widely in system design and data layout to acheive different tradeoffs. Graph Stores like Neo4j [13] acheive flexibility in graph representation by storing different graph entities (nodes, edges, properties) separately, using pointers to associate them; however, such flexibility comes at the cost of scalability – query performance scales poorly when the graph size grows beyond availabile system memory. On the other hand, systems like Titan [16] adopt a key-value store layout for graphs to acheive scalability at the cost of flexibility. ZipG introduces a new data layout to reconcile flexibility and scalability in graph representation, and supports complex queries *directly on compressed graphs*.

Graph Compression Techniques. Over the past decade, a number of graph compression techniques have focused on supporting queries on compressed graphs [21–23, 26, 29, 30, 43, 50]; however, these techniques are limited to neighborhood queries and dense subgraph identification. Graph serving often demands much richer functionality [1, 4, 5, 18, 47, 53]. ZipG uses Succinct [19] as an underlying store to support queries on property as well as neighborhood queries on compressed graphs.

8 Conclusion

We have presented ZipG, a distributed graph store that supports queries on compressed graphs. ZipG exposes a minimal but functionally rich API, which we have used to implement Facebook's TAO and Graph Search functionalities. ZipG employs a *flexible* and *scalable* layout for graphs that supports complex graph queries efficiently. As a consequence, ZipG is able to run TAO and Graph Search workloads for a graph with over half a TB of data on a single 244GB server, achieving 3-10× higher throughput than compared systems. These gains carry over to distributed settings, where ZipG achieves as much as 5× higher throughput.

References

[1] Actual Facebook Graph Searches. http://actualfacebookgraphsearches.tumblr.com.

[2] AllegroGraph. http://franz.com/agraph/allegrograph/.

[3] ArangoDB. https://www.arangodb.com.

[4] Building a follower model from scratch. https://engineering.pinterest.com/blog/building-follower-model-scratch.

[5] Demining the "Join Bomb" with graph queries. http://neo4j.com/blog/demining-the-join-bomb-with-graph-queries/.

[6] FlockDB. https://github.com/twitter/flockdb.

[7] Function Shipping: Separating Logical and Physical Tiers. https://docs.oracle.com/cd/A87860_01/doc/appdev.817/a86030/adx16nt4.htm.

[8] GraphBase. http://graphbase.net.

[9] InfiniteGraph. http://www.objectivity.com/products/infinitegraph/.

[10] InfoGrid, The Web Graph Database. http://infogrid.org/.

[11] Introducing FlockDB. https://blog.twitter.com/2010/introducing-flockdb.

[12] Introducing Graph Search Beta. http://newsroom.fb.com/news/2013/01/introducing-graph-search-beta/.

[13] Neo4j. http://neo4j.com/.

[14] OrientDB. http://orientdb.com/.

[15] Property Graph Model. https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model.

[16] Titan. http://thinkaurelius.github.io/titan/.

[17] Titan Data Model. http://s3.thinkaurelius.com/docs/titan/current/data-model.html.

[18] Under the Hood: Building Graph Search Beta. https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-graph-search-beta/10151240856103920.

[19] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling Queries on Compressed Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 337–350, 2015.

[20] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed Large-scale Natural Graph Factorization. In *ACM International Conference on World Wide Web (WWW)*, pages 37–48, 2013.

[21] Bharat, Krishna and Broder, Andrei and Henzinger, Monika and Kumar, Puneet and Venkatasubramanian, Suresh. The connectivity server: Fast access to linkage information on the web. *Computer networks and ISDN Systems*, 30, 1998.

[22] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *International Conference on World Wide Web (WWW)*, pages 595–602, 2004.

[23] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On Compressing Social Networks. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 219–228, 2009.

[24] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[25] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: A High-Performance, Transactional Graph Store Based on Refinable Timestamps. *CoRR*, abs/1509.08443, 2015.

[26] W. Fan, J. Li, X. Wang, and Y. Wu. Query Preserving Graph Compression. In *ACM International Conference on Management of Data (SIGMOD)*, pages 157–168, 2012.

[27] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[28] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

[29] C. Hernández and G. Navarro. Compression of Web and Social Graphs supporting Neighbor and Community Queries. In *ACM Workshop on Social Network mining and Analysis (SNAKDD)*, 2011.

[30] C. Hernández and G. Navarro. Compressed Representations for Web and Social Graphs. *Knowledge and Information Systems*, 40(2):279–313, 2014.

[31] B. Iordanov. HypergraphDB: A Generalized Graph Database. In *Web-Age Information Management*. Springer, 2010.

[32] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, 2014.

[33] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *IEEE International Conference on Data Mining (ICDM)*, pages 229–238, 2009.

[34] A. Khandelwal, R. Agarwal, and I. Stoica. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 485–500, 2016.

[35] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[36] Lakshman, Avinash and Malik, Prashant. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44, 2010.

[37] K. Lang. Finding good nearly balanced cuts in power law graphs. *Preprint*, 2004.

[38] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[39] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13, 2011.

[40] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. *arXiv preprint arXiv:1408.2041*, 2014.

[41] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *ACM International Conference on Management of Data (SIGMOD)*, pages 135–146. ACM, 2010.

[42] N. Martinez-Bazan, S. Gómez-Villamor, and F. Escalé-Claveras. DEX: A high-performance graph database management system. In *IEEE Data Engineering Workshops (ICDEW)*, pages 124–127, 2011.

[43] H. Maserrat and J. Pei. Neighbor Query Friendly Compression of Social Networks. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 533–542, 2010.

[44] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[45] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *ACM SIGCOMM Computer Communication Review*, pages 181–194, 1997.

[46] Najork, Marc. The Scalable Hyperlink Store. In *ACM Conference on Hypertext and Hypermedia (HT)*, pages 89–98, 2009.

[47] Nathan Bronson and Zach Amsden and George Cabrera and Prasad Chakka and Peter Dimov and Hui Ding and Jack Ferris and Anthony Giardullo and Sachin Kulkarni and Harry Li and Mark Marchukov and Dmitri Petrov and Lovro Puzar and Yee Jiun Song and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[48] A. Pavlo, C. Curino, and S. Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72. ACM, 2012.

[49] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoP)*, pages 135–146, 2013.

[50] J. Shun, L. Dhulipala, and G. Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *IEEE Data Compression Conference (DCC)*, pages 403–412, 2015.

[51] Y. J. Song, M. K. Aguilera, R. Kotla, and D. Malkhi. RPC Chains: Efficient Client-server Communication in Geodistributed Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 277–290, 2009.

[52] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. Mercury: Enabling remote procedure call for high-performance computing. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2013.

[53] R. Wang, C. Conrad, and S. Shah. Using Set Cover to Optimize a Large-Scale Low Latency Distributed Graph. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.