ZipG: A Memory-efficient Graph Store for Interactive Queries

Anurag Khandelwal
UC Berkeley

Zongheng Yang
UC Berkeley

Evan Ye
UC Berkeley

Rachit Agarwal
Cornell University

Ion Stoica
UC Berkeley

ABSTRACT

We present ZipG, a distributed memory-efficient graph store for serving interactive graph queries. ZipG achieves memory efficiency by storing the input graph data using a compressed representation. What differentiates ZipG from other graph stores is its ability to execute a wide range of graph queries directly on this compressed representation. ZipG can thus execute a larger fraction of queries in main memory, achieving query interactivity.

ZipG exposes a minimal API that is functionally rich enough to implement published functionalities from several industrial graph stores. We demonstrate this by implementing and evaluating graph queries from Facebook TAO, LinkBench, Graph Search and several other workloads on top of ZipG. On a single server with 244GB memory, ZipG executes tens of thousands of queries from these workloads for raw graph data over half a TB; this leads to an order of magnitude (sometimes as much as 23×) higher throughput than Neo4j and Titan. We get similar gains in distributed settings compared to Titan.

1. INTRODUCTION

Large graphs are becoming increasingly prevalent across a wide range of applications including social networks, biological networks, knowledge graphs and cryptocurrency. Many of these applications store, in addition to the graph structure (nodes and edges), a set of *attributes* or *properties* associated with each node and edge in the graph [11, 15, 18, 24, 29]. Many recent industrial studies [12, 24, 29] report that the overall size of these graphs (including both the structure and the properties) could easily lead to terabytes or even petabytes of graph data. Consequently, it is becoming increasingly hard to fit the entire graph data into the memory of a single server [11, 29, 54].

How does one operate on graph data distributed between memory and secondary storage, potentially across multiple servers? This question has attracted a lot of attention in recent years for offline graph analytics, *e.g.*, recent systems like GraphLab [48], GraphX [38], GraphChi [43] and Trinity [54]. These systems now enable efficient "batch processing" of graph data for applications that often run at the scale of minutes.

Achieving high performance for interactive user-facing queries on large graphs, however, remains a challenging problem. For such interactive queries, the goal is not only to achieve millisecond-level latency, but also high throughput [24, 29, 35]. When the graph data is distributed between memory and secondary storage, potentially across multiple servers, achieving these goals is particularly hard. For instance, consider the query: "Find friends of Alice who live in Ithaca". One possible way to execute this query is to execute two sub-queries — find friends of Alice, and, find people who live in Ithaca — and compute the final result using a join or an intersection of results from two sub-queries. Such joins may be complex[1], and may incur high computational and bandwidth[2] overheads [3].

Another interesting possibility to execute the above query is to first find friends of Alice, and then for each friend, check whether or not the friend lives in Ithaca. Executing the query in this manner alleviates the overheads of join operation, but requires *random access* into the "location" property of each friend of Alice. The well-known problem here is that 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. Unless the data for each of Alice's friends is stored in memory, the query latency and system throughput suffers. Thus, intuitively, to achieve high system performance, graph stores should store as large a fraction of graph data as possible in memory.

One way to store a larger fraction of graph data in memory is to use compression. However, traditional block compression techniques (*e.g.*, gzip) are inefficient for graph queries precisely due to lack of locality — since queries may touch arbitrary parts of the graph, each query may require decompressing a large number of blocks (*e.g.*, all blocks that contain Alice's friends in the above example). Thus, designing compression techniques specialized to graphs has been an active area of research for the better part of last two decades [27, 28, 32, 37, 39, 40, 49, 52, 56]. Many of these techniques even support executing queries on compressed graphs [28, 32, 37, 39, 40, 49, 52]. However, existing techniques 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). Contemporary applications require executing far more complex queries [2, 3, 8, 29], often involving node and edge properties.

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.

ISBN 978-1-4503-4197-4/17/05...$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064012

[1] Graph search queries (such as the ones that we evaluate later in §5) when implemented using traditional RDBMS may require complex joins are referred to as "Join Bomb" by one of the state-of-the-art graph serving companies [3].

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

We present ZipG — a memory-efficient, distributed graph store for efficiently serving interactive graph queries. ZipG achieves memory efficiency by storing the input graph data (nodes, edges and the associated properties) using a compressed representation, and consequently stores a larger fraction of graph data in memory when compared to existing graph stores. What differentiates ZipG from existing graph stores is its ability to execute a wide range of queries directly on this compressed representation — ZipG exposes a minimal API that is rich enough to implement functionalities from several graph stores including those from Facebook [29], LinkedIn [58] and Twitter [7]. We demonstrate this by implementing and evaluating the published graph queries from Facebook TAO, LinkBench, Graph Search and several other workloads on top of ZipG. Using a single server with 244GB memory, ZipG executes tens of thousands of TAO, LinkBench and graph search queries for raw graph data over half a Terabyte.

ZipG builds upon Succinct [21], a distributed data store that supports random access and arbitrary substring search directly on compressed unstructured data and key-value pairs. ZipG uses a new simple and intuitive graph layout that transforms the input graph data into a flat unstructured file (§3). This layout admits memory-efficient representation using compression techniques from Succinct [21]. In addition, this layout carefully stores small amount of metadata along with the original input graph data in a manner that the two primitives of Succinct (random access and substring search) can be extended to efficiently implement interactive graph queries as expressive as those in Facebook TAO, LinkBench and Graph Search workloads directly on compressed representation of ZipG.

There are two challenges associated with storing data using a compressed representation. The first challenge is to support high write rates. The traditional approach to resolving this challenge is to use a log-structured approach [21, 42] — updates are appended into a log; these logs are periodically compressed into an immutable representation, after which the new updates are written into a new log. However, naïvely using a log-structured approach in graph stores results in nodes and edges having their data "fragmented" across multiple logs; without any additional data structures, each query now requires touching all the logs resulting in reduced throughput (§3.5). ZipG uses log-structured approach, but avoids touching all logs using the idea of fanned updates. Specifically, 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) that are necessary for query execution. The second challenge with compressed representation is in providing strong consistency guarantees and transactions. ZipG currently does not attempt to resolve this challenge. While several graph stores used in production [7, 29, 58] make a similar design choice, extending ZipG to provide such guarantees is an interesting future direction.

We evaluate ZipG against Neo4j [11] and Titan [18], two popular open-source graph stores. All our experiments run on a set of commodity Amazon EC2 machines, and use five workloads — Facebook TAO [29], LinkBench [24], Graph Search [8], Regular Path Queries [25] and simple graph traversals. We use graphs from the real-world (annotated with node and edge properties using TAO distribution) as well as LinkBench generated graphs containing millions of nodes and billions of edges. Our evaluation shows that ZipG significantly outperforms Neo4j and Titan in terms of system throughput, usually by an order of magnitude but sometimes by as much as 23×.

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 [11, 15, 18, 29], with graph data comprising of nodes, edges, and their associated properties.

Nodes and Edges. ZipG uses usual definitions of nodes and edges. Edges in ZipG could be directed or undirected. To model applications where graphs may have different "types" of edges (*e.g.*, comments, likes, relationships) [29], ZipG represents each edge using a 3-tuple comprising of `sourceID`, `destinationID` and an `EdgeType`, where the latter identifies the type of the edge. Each edge may potentially have a different EdgeType and may optionally have a `Timestamp`. The precise representation is described in §3.

Node and Edge Properties. Each node and edge in ZipG may have multiple properties, represented by `PropertyList`. Each PropertyList is a collection of (PropertyID, PropertyValue) pairs; *e.g.*, the PropertyList for a node may be {(age, 20), (location, Ithaca), (zipcode, 14853)}. Each PropertyList in ZipG may have arbitrarily many properties.

2.2 ZipG Interface

ZipG exposes a minimal, yet functionally rich, interface that abstracts away the internal graph data representation details (*e.g.*, compression). Applications interact with ZipG as if they were working on original graph data. 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 edges. Many queries, however, also require a notion of time (*e.g.*, find all comments since last login). To efficiently execute such queries, ZipG uses TimeOrder — for each node, the incident edges of the same type are logically sorted using timestamps. TimeOrder of an edge represents the order (*e.g.* i-th) of the edge within this sorted list.
- `EdgeData`: Given the TimeOrder within an EdgeRecord, the EdgeData stores the triplet (destinationID, timestamp, PropertyList) for the corresponding edge.

Table 1 outlines the interface exposed by ZipG, along with some examples. Applications submit the `graph`, represented using the property model in §2.1, to ZipG and generate a memory-efficient representation using `compress(graph)`. Applications can then invoke powerful primitives (Table 1) as if the input graph was stored in an uncompressed manner; ZipG internally executes queries efficiently directly on the compressed representation. Most queries in Table 1 are self-explanatory; we discuss some of the interesting aspects below, and return to details on query implementation in §4.2.

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).

Table 1: ZipG's API and an example for each API. See §2.2 for definitions and detailed discussion.

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 **Ithaca** 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; `tLo` and `tHi` are `timestamps` (to define a time range).
`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.

Node-based queries. Consider again the query "Find friends of Alice who live in Ithaca". Letting Alice to be the NodeID and assuming friends have edgeType `0`, the query `get_neighbor_ids(Alice,0,{Location, Ithaca})` 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 Ithaca (that is, ZipG avoids joins to whatever extent possible). Applications that have knowledge about the structure of the graph and/or queries can also execute the same query using a join-based implementation — using `get_neighbor_ids(Alice,0,⋆)` ∩ `g.get_node_ids(Location, Ithaca)`, where the former returns all friends of Alice and the latter returns all people who live in Ithaca. We compare the performance of executing queries with and without joins in Appendix B.3.

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 using `get_edge_range`. Finally, applications can insert EdgeRecords using `append`, delete existing EdgeRecords using `delete`, and update an EdgeRecord using a `delete` followed by an `append`.

3. ZipG DESIGN

In this section, we present ZipG's design and implementation. We start with a brief description of Succinct [21].

3.1 Succinct Background

Succinct [21] is a data store that supports a wide range of queries directly 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, and a key-value (KV) interface for queries on semi-structured data. Two queries from Succinct that will be of use for ZipG are:

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

Internally, Succinct implements the above two queries using three main data structures. The first two data structures are a suffix array [17,50] and an inverse-suffix array [21], that enable the search and the random access functionalities, respectively. As such, both of these data structures have size $n\log(n)$ for a flat file with n characters. Succinct achieves compression using *sampling* — only a few sampled values (*e.g.*, for sampling rate α, value at indexes $0,\alpha,2\alpha,..$) from the two data structures are stored. The third data structure allows computing the unsampled values during query execution. This third data structure has an interesting structure and admits an extremely small memory footprint. 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.

More details on data structures and query algorithms used in Succinct are not required to keep the paper self-contained; we refer the reader to [21].

3.2 ZipG overview

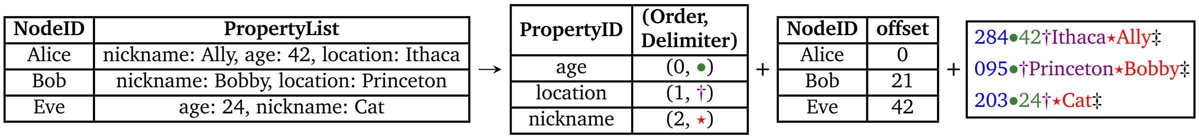
ZipG builds on top of Succinct, but has to resolve a number of challenges to achieve the desired expressivity, scalability and performance. We outline some of these challenges below, and provide a brief overview of how ZipG resolves these challenges.

Storing graphs. One of the fundamental challenges that ZipG has to resolve is to design an efficient layout for storing graph data using the underlying data store. This is akin to GraphChi [43] and Ligra [55], that design new layouts for using the underlying storage system for efficient batch processing of graphs. ZipG's layout should not only admit a memory-efficient representation when using Succinct, but should also enable efficient implementation of interactive graph queries using the random access and search primitives of Succinct.

ZipG's graph layout uses two flat unstructured files:

- **NodeFile** stores all NodeIDs and corresponding properties. ZipG NodeFile adds a small amount of metadata to the list of (NodeID, nodeProperties) before invoking compression; this allows ZipG to tradeoff storage (in uncompressed representation) for efficient random access into node properties.
- **EdgeFile** stores all the EdgeRecords. 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.

We discuss design of ZipG NodeFile and EdgeFile, and associated tradeoffs in §3.3.



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, ⋆)

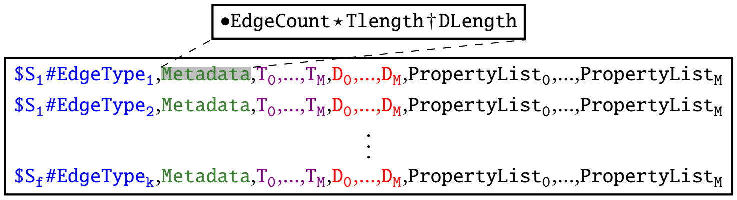
+

NodeID	offset
Alice	0
Bob	21
Eve	42

+

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

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



•EdgeCount⋆Tlength†DLength

$\$S_1$#EdgeType$_1$,Metadata,T$_0$,...,T$_M$,D$_0$,...,D$_M$,PropertyList$_0$,...,PropertyList$_M$
$\$S_1$#EdgeType$_2$,Metadata,T$_0$,...,T$_M$,D$_0$,...,D$_M$,PropertyList$_0$,...,PropertyList$_M$
⋮
$\$S_f$#EdgeType$_k$,Metadata,T$_0$,...,T$_M$,D$_0$,...,D$_M$,PropertyList$_0$,...,PropertyList$_M$

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.

Updating graphs. Another challenge for ZipG is to support high write rates over compressed graphs. Traditional data stores typically resolve this challenge using log-structured storage [21, 42] — updates are appended into a log; these logs are periodically compressed into an immutable representation, after which the new updates are written into a new log. However, naïvely 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, resulting in reduced throughput. ZipG avoids this problem using the idea of fanned updates — 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) that are necessary for query execution. We describe fanned updates in §3.5.

3.3 Graph Representation

Existing graph stores use layouts that expose a hard tradeoff between flexibility and scalability. On the one hand, systems like Neo4j [11] heavily use pointers to store both the structure of the graph and the properties for nodes and edges. While flexible in representation, a pointer-based approach suffers from scalability issues when the entire graph data does not fit into the memory of a single server[3]. Systems like Titan [18], on the other hand, scale well by using a layout that can be mapped to a key-value (KV) store abstraction. However, KV abstraction is not very well suited for interactive graph queries [29] — by enforcing values to be stored as a "single opaque object", KV abstraction limits the flexibility of graph stores. Specifically, storing all the node properties (or, set of incident edges) as a single opaque object precludes these systems from fine-grained access into individual node properties (or, individual edges).

ZipG uses a new graph layout that, while simple and intuitive, provides both the scalability and flexibility by operating on flat unstructured files. ZipG uses two flat unstructured files, which we describe next.

NodeFile

NodeFile stores all the NodeIDs and associated properties, and is optimized for two kind of queries on nodes: (1) given a (NodeID, List<propertyID>) pair, extract the corresponding propertyValues; and (2) given a PropertyList, find all NodeIDs whose properties match the propertyList.

NodeFile consists of three data structures (see Figure 1). First, each propertyID in the graph is assigned a unique delimiter[4] and stored as a **PropertyID → (order, delimiter) map**, where order is lexicographic ranking of the propertyID among all propertyIDs.

The second data structure is a flat unstructured file that stores PropertyLists along with some metadata as described next. The propertyValues are prepended by their propertyID's delimiter and then written in the flat file in sorted order of propertyIDs; if a propertyID has a null propertyValue, we 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 propertyList in Figure 1 is serialized into •42†Ithaca⋆Ally‡, where ‡ is the end-of-record delimiter.

The metadata in the second data structure exposes a space-latency tradeoff. Specifically, the size of propertyValues within a node's propertyList vary significantly in real-world datasets (*e.g.*, Alice's age 42 and location Ithaca in above example) [29]. Using the largest size of PropertyValues (8bytes for Alice) as a fixed size representation for each propertyValue enables efficient random access but at the cost of space inefficiency. On the other hand, naïvely using a space-efficient variable size representation (2bytes for age, 8bytes for location, etc.) without any additional information leads to inefficient random access — 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†Ithaca⋆Ally‡.

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

EdgeFile

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

[3] Pointer chasing during query execution requires multiple accesses to secondary storage and/or different servers, leading to undesired bottlenecks.

[4] Graphs usually have a small number of propertyIDs across all nodes and edges; ZipG uses one byte non-printable characters as delimiters (for up to 25 propertyIDs) and two byte delimiters (for up to 625 propertyIDs).

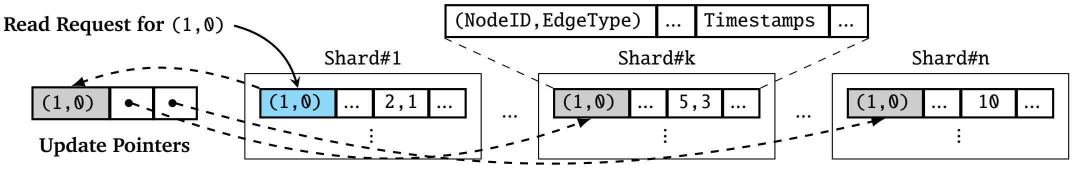


Figure 3: Update Pointers for the EdgeFile (§3.5).

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 (*e.g.*, return results sorted by timestamps, or find new comments since last login time [29]). Efficiently executing such queries requires performing binary search on timestamps. ZipG stores timestamps in each EdgeRecord in sorted order. To aid binary search, ZipG also stores the number of edges in the EdgeRecord within the metadata (denoted by `EdgeCount` in Figure 2).

There are several approaches for storing individual timestamp values. At one extreme are variable length encoding and delta encoding. 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 complicates random access on timestamps since extracting a timestamp requires extracting all the timestamps before it. Storing timestamps using delta encoding [53] also leads to a similar tradeoff. The other extreme is fixed length representation for 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 edges within an EdgeRecord. Since this length varies across EdgeRecords, ZipG stores the fixed length for each EdgeRecord in the corresponding metadata (`TLength` in Figure 2).

DestinationIDs. A natural choice for storing 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 corresponding destinationIDs, enabling efficient random access. ZipG uses a fixed length representation similar to timestamps for destinationIDs and stores this length in the metadata (`DLength` in Figure 2).

Edge Properties. As with destinationIDs, edge propertyLists are ordered such that ith timestamp and ith propertyList correspond to the same edge. The edge properties 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 (similar to NodeFile). ZipG currently does not support search on edge propertyLists, but can be trivially extended to do so using ideas similar to NodeFile.

3.4 ZipG Query Execution

We now describe how ZipG uses the graph layout from §3.3 to efficiently execute queries from Table 1.

Implementing node-based queries. It is easy to see that the NodeFile design allows implementing `get_node_property` query using two array lookups (one for property delimiter and one for ProperList offset), and one extra byte (for accessing propertyValue length) in addition to extracting the PropertyValue itself (using random access primitive from §3.1). Implementing `get_node_ids` is more interesting; 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.

Implementing edge-based queries. The `get_edge_record` operation returns the `EdgeRecord` for a given (sourceID, edgeType) pair and is implemented using `search($sourceID#edgeType)` on the EdgeFile. This returns the offset for the EdgeRecord within the EdgeFile. 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`).

3.5 Fanned Updates

As outlined in the introduction (§1), while storing data in a compressed form leads to performance benefits when the uncompressed data does not fit in faster storage, it also leads to the challenge of handling high write rates. Specifically, the overheads of decompressing and re-compressing data upon new writes need to be amortized over time, while maintaining low memory and computational overheads and while minimizing interference with ongoing queries on existing data. The traditional approach to achieve this is to use log-structured storage; within this high-level approach, there are two possible techniques that expose different tradeoffs.

The first technique is to maintain a log-structured store (LogStore), per shard or for all shards on a server, and periodically merge the LogStore data with the compressed data. To avoid scanning the entire LogStore during query execution (to locate the data needed to answer the query), additional `nodeID` → `LogStore-offset` pointers can be stored that allow random access for each node's data. The benefit of this approach is that all the data for any graph node remains “local”. The problem however is that such an approach requires over-provisioning of memory for LogStore at each server, which reduces the overall memory efficiency of the system. Moreover, periodically merging the LogStore data with compressed data interferes with ongoing

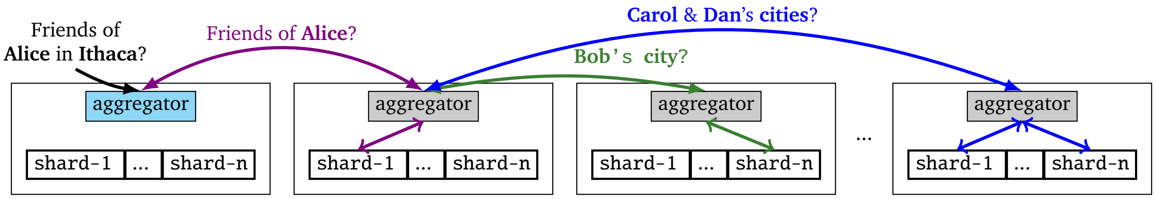


Figure 4: Function Shipping in ZipG. See §4.1 for discussion.

queries (as does copying the data to background processes for merging process). Finally, this approach requires using concurrent data structures to resolve read-write conflicts at *each* server (for concurrent reads and writes over LogStore data).

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. A single LogStore in ZipG offers three advantages. First, ZipG does not require decompressing and re-compressing the previously compressed data, and thus observes minimal interference on queries being executed on previously compressed data. Second, a single LogStore in ZipG also achieves higher memory efficiency than the first technique discussed above — rather than over-provisioning each server for a LogStore, ZipG requires just one dedicated LogStore server which can be optimized for query efficiency rather than memory efficiency. Finally, a single LogStore in ZipG avoids complicated data structures for concurrent reads and writes at each server, making the entire system simpler.

For unstructured and semi-structured data, where records are usually modeled as disjoint entities, a single LogStore leads to benefits compared to per-server LogStores [21,23,31,42]. However, for graph structured data, using a single LogStore leads to a new challenge — *fragmented* storage. In the absence of decompression and re-compression, as new edges are added on a node, the EdgeRecord of the node will be fragmented across multiple shards. For instance, suppose at time $t = 0$ we upload the graph data to ZipG and create a single LogStore ℓ. Consider a node u that has some data in the originally uploaded data to ZipG. Suppose that multiple updates happen between time $t = 0$ and $t = t_1$ such that: (1) the size of ℓ crosses the threshold at time t_1; and (2) some of these updates are on node u. Then, at time t_1, we convert ℓ into ZipG's memory-efficient representation and create a new LogStore ℓ'. Then, for all updates on node u after time t_1, the data belonging to node u will now be fragmented across at least three shards: the original one that had the data for node u in pre-loaded graph data, the shard corresponding to the old LogStore ℓ and the new LogStore ℓ'. Depending on the update rates and on the skew in update queries, any node in the graph may thus have data fragmented across multiple shards over time (we evaluate the fragmentation over time and across nodes in Appendix A). We thus need some additional techniques to efficiently exploit all the benefits of having a single LogStore.

One way to handle fragmented data is to send each query to all shards, and retrieve the corresponding results. This is extremely inefficient — as our results in Appendix A suggest, most queries can be answered by touching an extremely small fraction of the shards (less than 10% for 99.9% of the nodes); executing each query at all shards would thus have high unnecessary CPU overhead. ZipG instead uses the idea of `Fanned Updates`. 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 §4.1) ensures that most ZipG queries are first forwarded to a single shard[5]. At a high-level, Fanned Updates avoid touching all shards using a set of update pointers that logically *chain* together data correspond to the same node or edge. As shown in Figure 3, these pointers store a reference to offsets of NodeFile and/or EdgeFile at other shards that store the updated data. ZipG stores these update pointers only at the shard where the node/edge first occurs; that is, in our example above, 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. We describe below how ZipG uses fanned updates to optimize query execution. As the graph is updated or is fragmented over time, these update pointers are updated as well. For workloads where updates form a small fraction of all queries [8,24,29], the overhead of storing and updating these pointers is minimal. ZipG, thus, keeps these pointers uncompressed.

ZipG query execution for 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 by following the update pointers, and collect the additional query results while avoiding touching all servers. Note that since most nodes and edges are unlikely to be updated frequently in real-world graph workloads, a majority of read queries would be confined to a single server. ZipG implements deletes as lazy deletes with a bitmap indicating whether or not a node or an edge has been deleted; finally, updating a previously written record in ZipG is implemented as deletes followed by an append.

4. ZipG IMPLEMENTATION

We have implemented ZipG on top of Succinct using roughly 4000 lines of C++ code, as well as a package running atop Apache Spark in Scala. We start this section by outlining some of the system implementation details (§4.1). We then show how ZipG API can be used to implement published functionalities from a variety of graph stores (§4.2).

4.1 System Implementation

We now outline the key aspects of ZipG implementation.

Graph Partitioning (Sharding). Several previous studies have established that efficient partitioning of social graphs is a hard problem [22,45,46]. Similar to existing graph stores [18], ZipG uses a simple hash-partitioning scheme — it creates a number of shards, default being one per core, and hash partitions

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

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 node is co-located on the same shard, enabling efficient execution of neighborhood queries. Finally, each of the shards is transformed into the ZipG layout (§3).

Fault Tolerance and Load Balancing. ZipG currently uses 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 current implementation to incorporate storage-efficient fault tolerance and skew-tolerant load balancing techniques [34, 42] is an interesting 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 Ithaca"). To minimize network roundtrips and bandwidth utilization in a distributed setting, ZipG pushes computation closer to the data via *function shipping* [5, 57]. 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 requires executing subqueries 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. Indeed, ZipG also supports multi-level function shipping; that is, a subquery may be further decomposed into sub-subqueries and forwarded to respective servers.

Concurrency Control. Having a log-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.

4.2 ZipG Expressiveness

ZipG design and interface is rich enough to implement published functionalities from several industrial graph stores. To demonstrate this, we have implemented all the published queries from Facebook TAO [29], LinkBench [24], Graph Search [8] as well as more complex graph queries such as regular path queries [26] and graph traversal queries [1] on top of ZipG. We now discuss these implementations and associated tradeoffs. Table 2 outlines the implementation for TAO and LinkBench queries[6], and Table 3 outlines the implementation of Graph Search queries using ZipG API.

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.

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,*)`

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`.

```
rec $\leftarrow$ get_edge_record(id, atype)
results $\leftarrow \emptyset$
for i $\leftarrow$ idx to idx+limit do
    edgeEntry $\leftarrow$ get_edge_data(rec, i)
    Add edgeEntry to results
end for
return results
```

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`.

```
rec $\leftarrow$ get_edge_record(id1, atype)
(beg, end) $\leftarrow$ get_time_range(rec, hi, lo)
results $\leftarrow \emptyset$
for i $\leftarrow$ beg to end do
    edgeEntry $\leftarrow$ get_edge_data(rec, i)
    Add edgeEntry to results if destination $\in$ id2set
end for
return results
```

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)`.

```
rec $\leftarrow$ get_edge_record(id, atype)
(beg, end) $\leftarrow$ get_time_range(rec, hi, lo)
results $\leftarrow \emptyset$
for i $\leftarrow$ beg to min(beg+limit, end) do
    edgeEntry $\leftarrow$ get_edge_data(rec, i)
    Add edgeEntry to results
end for
return results
```

[6] The nodes and edges in ZipG are equivalent to *objects* and *associations* in TAO and LinkBench.

The second type of queries are based on Timestamps. For instance, consider the following query: "find all comments from Alice between SIGMOD abstract and paper submission deadlines". ZipG is particularly efficient for such queries due to its ability to efficiently perform binary search on timestamps (§3) and return corresponding edges and their properties. Algorithms 1, 2 and 3 show that these fairly complicated Facebook TAO queries can be implemented in ZipG using less than 10 lines of code.

LinkBench models Facebook's database workload for social graph queries. Note that TAO and LinkBench have the same set of queries, but vary significantly in terms of query distribution (LinkBench is much more write-heavy). Thus, ZipG implements LinkBench queries similar to TAO queries, as outlined above.

Facebook Graph Search originally supported interesting, and complex, queries on graphs [8]. Implementing graph search queries is even simpler in ZipG since most queries directly map to ZipG API, as shown in Table 3.

Regular path queries and graph traversals differ significantly from the above queries. In particular, while the queries discussed above usually require information from the immediate neighborhood of a single node, path queries and traversals examine the structure for larger subgraphs. However, both of these classes of queries can be implemented in a recursive manner where each step requires access to the set of edges incident on a subset of nodes (and the corresponding neighbor nodes). In ZipG, this translates to sequences of `get_neighbor_ids`, `get_edge_record` and `get_edge_data` operations.

5. EVALUATION

We evaluate ZipG against popular open-source graph stores across graphs of varying sizes, real-world and benchmark query workloads, and varying cluster sizes.

Compared Systems. We compare ZipG against two open-source graph stores. Neo4j [11] is a single machine graph store and does not support distributed implementations. 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 distributed graph store that requires a separate storage backend. We use Titan version 0.5.4 [18] with Cassandra 2.2.1 [44] as the storage backend. We also experimented with DynamoDB 0.5.4 for Titan but found it to be performing worse. Titan supports compression. We present results for both uncompressed (Titan) and compressed (Titan-compressed) representations.

Experimental Setup. All our experiments run on Amazon EC2. 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. To make results consistent (Neo4j does not support graph partitioning across servers), we configured all systems to run without replication.

Workloads. Our evaluation employs a wide variety of graph workloads. We use TAO and Linkbench workloads from Facebook (with original query distributions, Table 2), and a synthetic Graph Search workload (Table 3). We also evaluate more complex workloads such as regular path queries [6] and graph traversal queries [1]. In addition, we evaluate the performance for each workload's component queries in isolation to build in-depth insights on the performance of the three systems. We discuss the results for TAO, Linkbench and Graph Search workloads in this section, and the results for other workloads in Appendix B.

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

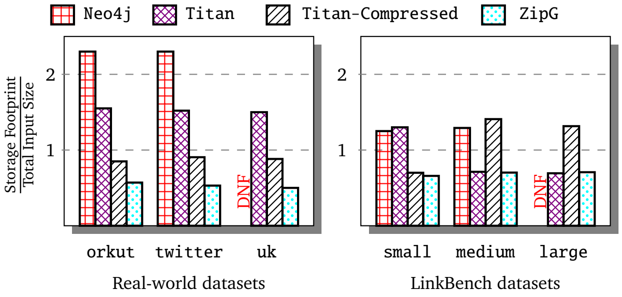


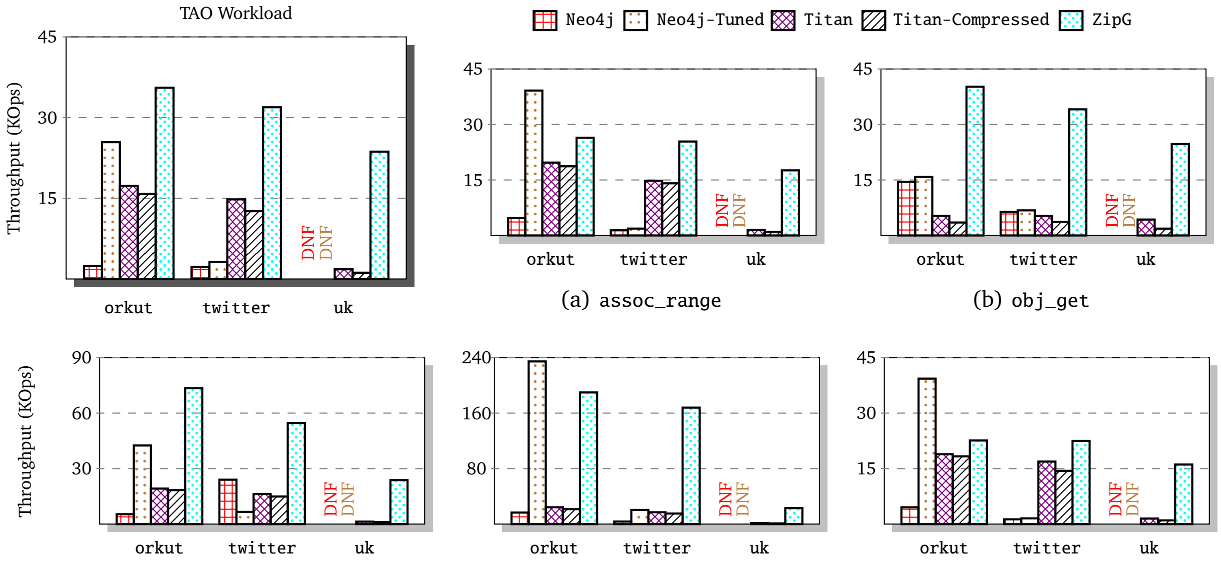
Figure 5: ZipG's storage footprint (§5.1) is 1.8−4× lower than Neo4j and 1.8−2× lower than Titan. `DNF` denotes that the experiment did not finish after 48 hours of data loading.

Datasets. Table 4 shows the datasets used in our evaluation. For real-world datasets, we used the node and edge property distribution from Facebook TAO paper [29]. 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. For LinkBench datasets, we directly use the LinkBench benchmark tools [9] to generate three datasets: `small`, `medium` and `large`. 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.

5.1 Storage Footprint

Figure 5 shows the ratio of total data representation size and raw input size for each system. We note that ZipG can put 1.7−4× larger graphs in main memory compared to Neo4j and Titan uncompressed (which, as we show later, leads to degraded performance[7]). 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 a memory-efficient representation of the input graph.

[7] Intuitively, Titan uses delta encoding for edge destinationNodeIDs, and variable length encoding for node and edge attributes [19] 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.



(a) assoc_range

(b) obj_get

(c) assoc_get

(d) assoc_count

(e) assoc_time_range

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.

Table 5: Summary of which datasets fit completely in memory.

Dataset	Neo4j	Titan-C	Titan	ZipG
orkut / linkbench-small	✓	✓	✓	✓
twitter / linkbench-medium		✓		✓
uk / linkbench-large				

Since LinkBench assigns synthetically generated properties to nodes and edges, LinkBench datasets have lower compressibility compared to the real-world datasets. Accordingly, ZipG's compression factor is roughly 15% lower for the LinkBench datasets than the corresponding real-world datasets. The storage overheads for Neo4j and Titan, on the other hand, are lower for the LinkBench datasets, since they have to maintain much smaller secondary indexes for a single node property. As such, ZipG's storage footprint is 1.8−2× smaller than Neo4j and Titan uncompressed, while being comparable to Titan-Compressed.

Table 5 summarizes which datasets fit *completely* in memory for different systems our experiments.

5.2 Single Machine (Figure 6, 7, 8)

We now analyze the performance of different graph stores on a single server with 244GB of RAM and 32 CPU cores. We note that across all experiments, Neo4j-Tuned achieves strictly better performance than Neo4j. Similarly, Titan uncompressed achieves strictly better performance Titan compressed (for reasons discussed in Footnote 7). The discussion thus focuses on Neo4j-Tuned, Titan uncompressed and ZipG.

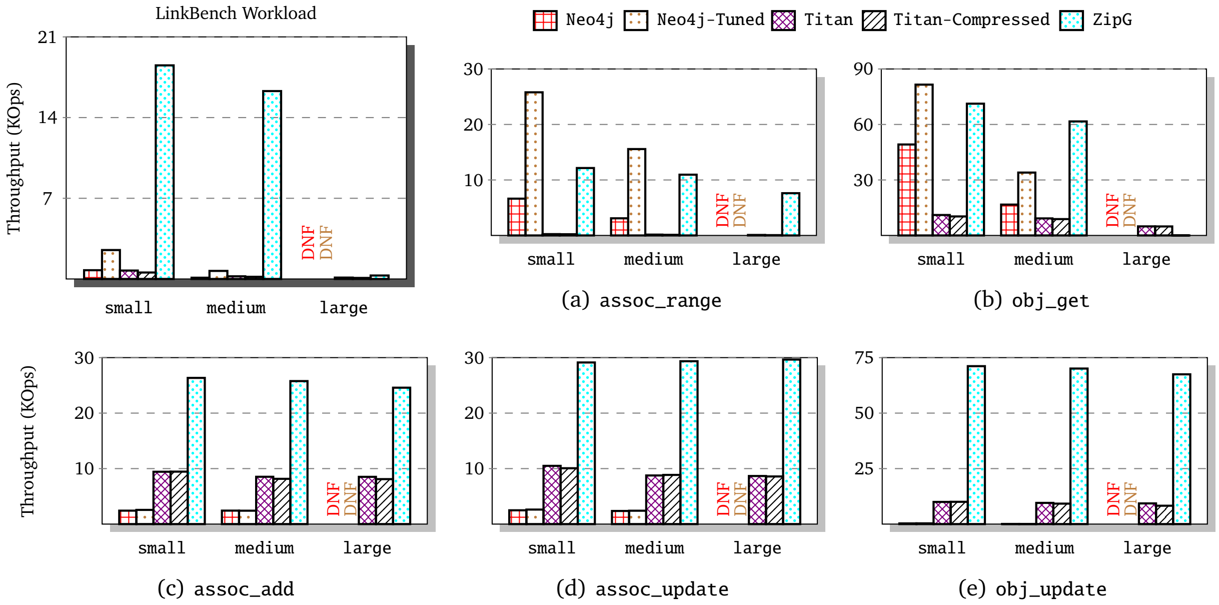
TAO Workload (Figure 6)

We start by observing that when the 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 while Neo4j and Titan are not — Neo4j requires following a set of pointers on NodeTable, while Titan needs to first extract the corresponding (key, value) pair from Cassandra and then scan the value to extract node properties. 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.

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. Titan, on the other hand, maintains its throughput for all queries. This 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.

Individual TAO queries (Figure 6(a)-6(e)). Analyzing the performance of the top 5 TAO queries for the Orkut dataset, node-based queries involving random access (obj_get, Figure 6(b)) perform better for ZipG than Neo4j and Titan due to reasons cited earlier. Similarly, for the edge-based queries (*e.g.*, assoc_get, Figure 6(c)), ZipG achieves higher



(a) assoc_range

(b) obj_get

(c) assoc_add

(d) assoc_update

(e) obj_update

Figure 7: Single server throughput for the LinkBench workload, and its top 5 component queries in isolation. DNF denotes that the experiment did not finish after 48 hours of data loading. Note that the figures have different y-scales.

throughput by avoiding the overheads of scans employed by other systems. When queries have a `limit` on the result cardinality, other systems can stop scanning earlier and thus achieve relatively improved performance, *e.g.*, `assoc_range` and `assoc_time_range` in Figures 6(a) and 6(e) respectively. For larger datasets, while compared systems fail to keep their data in memory, ZipG achieves considerably higher throughput for all the individual queries, since its lower storage footprint permits execution of most queries in memory.

LinkBench Workload (Figure 7)

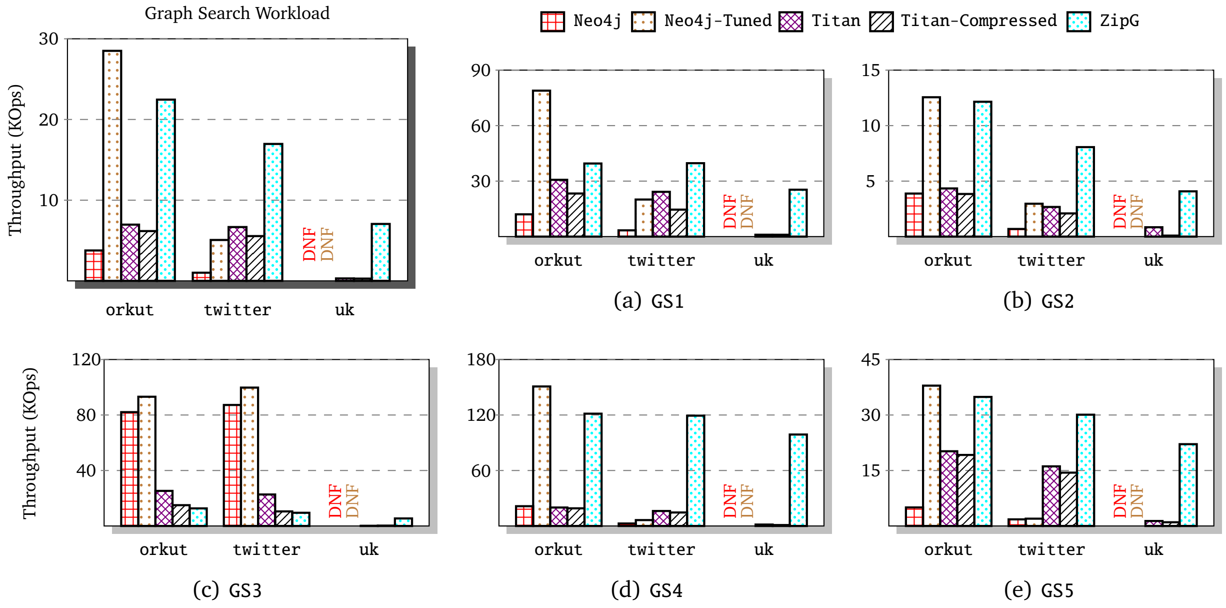
Despite having the same set of queries as TAO, the absolute throughput for the LinkBench workload is distinctly lower for all systems. This is due to two main reasons: first, a much larger fraction of the queries (see Table 2) are either write, update or delete operations, requiring modification of graph elements. This leads to overheads due to data persistence, as well as lock-based synchronization for atomicity and correctness of graph mutations in all compared systems. Second, most of the queries perform filters on node neighborhoods, with their accesses being skewed towards nodes with more neighbors [24] — as a result, the average number of edges accessed per query is much larger than in the TAO workload, leading to lower query throughput.

Also observe that Neo4j and Titan observe much lower throughput than ZipG for all datasets. While Neo4j is relatively efficient in executing read-only queries, write queries become a significant performance bottleneck, since they require modifications at multiple random locations due to Neo4j's pointer based-approach. Titan, on the other hand, is able to support write and update operations at relatively higher throughput due to Cassandra's write-optimized design. However, the throughput for edge-based operations is significantly lower because Cassandra is not optimized for range queries.

ZipG avoids both of the above issues for the `small` and `medium` datasets. In particular, all graph mutations are isolated to a write-optimized LogStore through Fanned Updates (§3.5), while edge-based operations do not need to scan the entire neighborhood to filter the required edges (§3.3). However, ZipG's throughput drops for the `large` dataset; this is due to the relatively lower compressibility of LinkBench generated graphs, which prevents crucial data-structures in the underlying Succinct representation for the NodeFile from fitting in memory.

Individual LinkBench Queries (Figures 7(a)-7(e)). Note that the top 5 queries in the LinkBench workload differ from the TAO workload. The performance trends for the `assoc_range` (Figure 7(a)) and `obj_get` (Figire 7(b)) queries are similar to the corresponding TAO queries, except for a few key differences. First, Titan observes significantly worse performance for `assoc_get` query in the LinkBench workload. This is because the average number of neighbors for each node in the LinkBench dataset is much larger than the TAO workload, and is heavily skewed, i.e., some nodes have very large neighborhoods, while most others have relatively few neighbors. Titan's performance drops significantly due to range queries over large neighborhoods, since Cassandra is not optimized for such queries, resulting in reduced throughput. Second, Neo4j observes better performance for `obj_get` query in the LinkBench workload, because of the workload's *query* skew. Since the accesses to the nodes for `obj_get` query are heavily skewed, Neo4j is able to cache the most frequently accessed nodes, leading to higher throughput. Finally, for the `large` dataset, ZipG is unable to keep one of Succinct's component data structures in memory that is responsible for answering node-based queries, leading to a reduced throughput for the `obj_get` query.

Finally, we note that ZipG outperforms compared systems for write-based queries including `assoc_add` (Figure 7(c)), `assoc_update` (Figure 7(d)) and `obj_update` (Figure 7(e)). As discussed above, Neo4j's write performance suffers since each write incurs updates at multiple random locations in its graph representation. Titan achieves a relatively better performance due to Cassandra's write-optimized design. ZipG, on the other hand, is able to maintain a high write throughput due to its write optimized LogStore and Fanned Updates approach.



(a) GS1

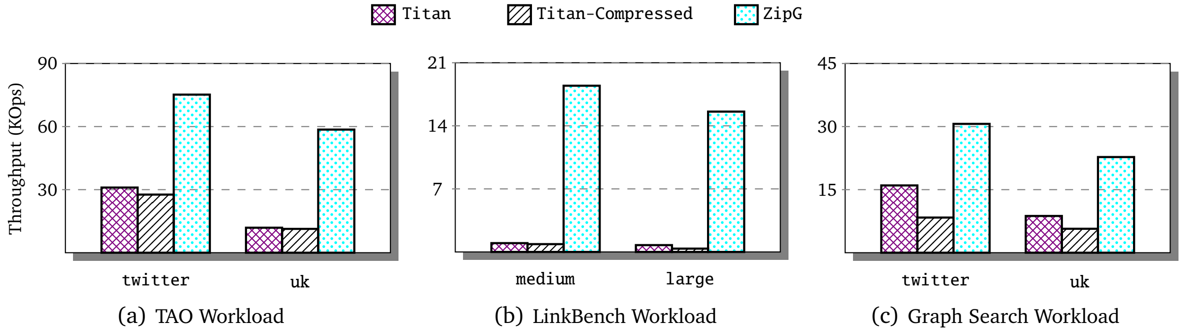
(b) GS2

(c) GS3

(d) GS4

(e) GS5

Figure 8: Single server throughput for the Graph Search workload, and its 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.



(a) TAO Workload

(b) LinkBench Workload

(c) Graph Search Workload

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

Graph Search Workload (Figure 8)

We designed the graph search workload for two reasons. First, while TAO and LinkBench workloads are 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 3, ZipG's powerful API enables simple implementation of queries that are far more complex than the TAO and LinkBench 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.

The results for the graph search workload follow a very similar pattern as for TAO workload (Figure 6, left), with two main differences. First, as with the LinkBench workload, 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. Second, when the uncompressed graph fits entirely in memory, Neo4j-Tuned achieves better performance than ZipG. The latter highlights ZipG overheads. 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 for search-based queries like "Find musicians in Ithaca". For such a query, ZipG's partitioning scheme requires ZipG touching all partitions. Neo4j and Titan, on the other hand, use global indexes and thus require touching no more than two partitions. 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.

Individual Graph Search Queries (Figure 8(a)-8(e)). Most

Graph Search queries follow trends similar to the TAO workload since they are random-access intensive. In particular, queries GS1 (Figure 8(a)), GS4 (Figure 8(d)) and GS5 (Figure 8(e)) perform random-access on edge data, while query GS2 (Figure 8(b)) performs random access on both edge and node data. For all such queries, Neo4j-Tuned achieves high throughput for the orkut dataset since all the data fits in memory, but as the datasets no longer fit in memory the performance drops drastically due to Neo4j's pointer-based approach, similar to the TAO workload. Titan, on the other hand, extracts the data corresponding to nodes and edges as key value pairs and scans the value component to obtain the relevant data (node or edge properties), resulting in lower throughput, which drops even lower when the datasets no longer fit in memory. ZipG, on the other hand, exploits its random-access friendly layout to achieve high throughput for all such queries, with little degradation on increasing the dataset size due to its memory-efficient graph representation.

Finally, query GS3 (Figure 8(c)) is unique, in that, it performs search queries on node attributes. As discussed before, ZipG's performance for this query is comparable or worse than the compared systems for the orkut dataset, since it touches multiple partitions to evaluate search results while other systems use global indexes. However, as the dataset sizes grow larger, the compared systems observe much lower throughput since these indexes no longer fit in memory, resulting in expensive accesses to secondary-storage.

We note that while ZipG supports joins for executing graph queries if required, it avoids joins in executing queries GS2 and GS3 due to performance considerations. We provide performance comparisons for these queries executed with and without joins in ZipG in Appendix B.3.

5.3 Distributed Cluster (Figure 9)

Neo4j does not have a distributed implementation. We therefore restrict our discussion to the performance of ZipG and Titan on a distributed cluster of 10 servers, with a total of 300GB of RAM and 80 CPU cores across the cluster.

TAO Workload (Figure 9(a))

We make two observations. First, Titan can now fit the entire Twitter dataset in memory leading to 2× higher throughput compared to single server setting, despite the increased overhead of inter-server communication (similar remarks for UK dataset). The second observation is that ZipG achieves roughly 2.5× higher throughput in distributed settings compared to single server setting. Note that our distributed servers have 10×8 cores, 2.5× 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.

LinkBench Workload (Figure 9(b))

These results allow us to make an interesting observation — unlike single machine setting, ZipG is able to cache a much larger fraction of crucial Succinct data-structures, leading to almost negligible performance degradation on going from the medium to the large dataset. Second, unlike the TAO workload, ZipG is unable to achieve throughput increase proportional to the number of cores. This is because the access pattern for edge-based queries in LinkBench is skewed towards nodes that have larger neighborhoods. As a consequence, a small set of servers that store nodes with large neighborhoods remain bottlenecked due to higher query volume and computational overheads.

Graph Search Workload (Figure 9(c))

Again, most performance trends for the Graph Search workload are similar for the distributed cluster and single server settings. We note that Titan's performance for the Graph Search workload scales better than ZipG's performance when the number of servers is increased. This is due to the contribution of search based queries, i.e., query GS3. As discussed in §5.2, the difference in performance for such queries lie in the choice of partitioning scheme for the two systems. While ZipG touches all partitions, and therefore all servers in the cluster for search-based queries, Titan's global index approach confines such queries to a single server for most situations, allowing the query performance to scale better. However, Titan's global index approach suffers in performance when the index grows too large to fit in memory.

6. RELATED WORK

Graph Stores. In contrast to graph batch processing systems [38, 43, 48, 55], graph stores [4, 10, 11, 14, 16, 18, 29, 35, 51, 54] usually focus on queries that are user-facing. Consequently, the goal in design of these stores is to achieve millisecond-level query latency and high throughput. We already compared the performance of ZipG against Neo4j and Titan, two popular open-sourced graph stores. Other systems, *e.g.*, Virtuoso [20], GraphView [10] and Sparksee [16] that use secondary indexes for efficiently executing graph traversals suffer from storage overhead problems similar or to Neo4j (high latency and low throughput due to queries executing off secondary storage).

Graph Compression. Traditional block compression techniques (*e.g.*, gzip) are inefficient for graphs due to lack of locality — each query may require decompressing many blocks. Several graph compression techniques have focused on supporting queries on compressed graphs [27, 28, 32, 37, 39, 40, 49, 52, 56]. However, these techniques are limited in expressiveness to queries like extracting adjacency list of a node, or matching subgraphs. Graph serving often requires executing much more complex queries [3, 8, 29, 58] involving node and edge attributes. ZipG achieves compression without compromising expressiveness, and is able to execute all published queries from Facebook TAO [29], LinkBench [24] and graph search [8].

7. CONCLUSION

We have presented ZipG, a distributed memory-efficient graph store that supports a wide range of interactive graph queries on compressed graphs. ZipG exposes a minimal but functionally rich API, which we have used to implement all the published queries from Facebook TAO, LinkBench, and Graph Search workloads, along with complex regular path queries and graph traversals. Our results show that ZipG can executes tens of thousands of queries from these workloads for a graph with over half a TB of data on a single 244GB server. This leads to as much as 23× higher throughput than Neo4j and Titan, with similar gains in distributed settings where ZipG achieves up to 20× higher throughput than Titan.

Acknowledgments

This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

8. REFERENCES

[1] Breadth First Search. https://en.wikipedia.org/wiki/Breadth-first_search.

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

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

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

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

[6] gMark Queries for LDBC Social Network Benchmark. https://github.com/graphMark/gmark/tree/master/demo/social/social-translated.

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

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

[9] LinkBench. https://github.com/facebookarchive/linkbench.

[10] Microsoft GraphView. https://github.com/facebookarchive/linkbench.

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

[12] Neo4j Pushes Graph DB Limits Past a Quadrillion Nodes. https://www.datanami.com/2016/04/26/neo4j-pushes-graph-db-limits-past-quadrillion-nodes/.

[13] openCypher. http://www.opencypher.org.

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

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

[16] Sparksee by Sparsity Technologies. http://www.sparsity-technologies.com.

[17] Suffix Array. http://en.wikipedia.org/wiki/Suffix_array.

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

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

[20] Virtuoso Universal Server. http://virtuoso.openlinksw.com.

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

[22] 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)*, 2013.

[23] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large cams for high performance data-intensive networked systems. In *NSDI*, 2010.

[24] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *ACM International Conference on Management of Data (SIGMOD)*, 2013.

[25] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. Generating flexible workloads for graph databases. *Proceedings of the VLDB Endowment (PVLDB)*, 9(13):1457–1460, 2016.

[26] P. Barceló Baeza. Querying graph databases. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 175–188, 2013.

[27] 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.

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

[29] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference (ATC)*, 2013.

[30] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 194–204, 1999.

[31] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[32] 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)*, 2009.

[33] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *ACM International Conference on Management of Data (SIGMOD)*, pages 323–330, 1987.

[34] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment*, 2010.

[35] 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.

[36] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *ACM International Conference on Management of Data (SIGMOD)*, pages 619–630, 2015.

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

[38] 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)*, 2014.

[39] 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.

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

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

[42] 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)*, 2016.

[43] 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)*, 2012.

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

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

[46] 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*, 2009.

[47] L. Libkin and D. Vrgoč. Regular path queries on graphs with data. In *ACM International Conference on Database Theory (ICDT)*, pages 74–85, 2012.

[48] 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.

[49] A. Maccioni and D. J. Abadi. Scalable pattern matching over compressed graphs via dedensification. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2016.

[50] U. Manber and G. Myers. Suffix Arrays: A new method for on-line string searches. *SIAM Journal on Computing (SICOMP)*, 1993.

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

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

[53] 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*, 1997.

[54] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *ACM International Conference on Management of Data (SIGMOD)*. ACM, 2013.

[55] 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)*, 2013.

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

[57] 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)*, 2009.

[58] 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.

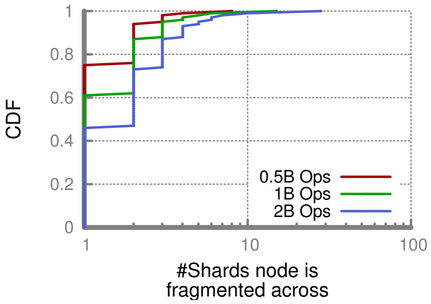


Figure 10: Most of the nodes have their data fragmented across a small number of shards. In addition, as expected, nodes have their data fragmented across more shards as more queries are executed by the system.

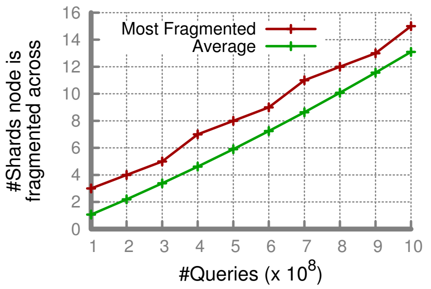


Figure 11: As more queries are executed, the data fragmentation gets worse for both: average fragmentation across all the nodes, and fragmentation for the most fragmented node (the one that has its data fragmented across the maximum number of shards).

APPENDIX

A. DATA FRAGMENTATION

We discussed the problem of data fragmentation in §3.5. We now provide more insights on how data fragments over time and across nodes. We start with the LinkBench large dataset (from Table 4) and partition is across 40 shards (with same settings as in §5). We then execute LinkBench queries for a varying amount of time over a running system — we start with a single LogStore and when the size of the LogStore crosses 8GB threshold, we compress the LogStore data into a new shard and create a new LogStore. We then take snapshots of the system every time the system has executed 100 Million LinkBench queries. Using these snapshots, we can evaluate the data fragmentation.

Figure 10 shows the CDF (across all nodes) for the fraction of shards (among all shards at the time of the snapshot) that a particular node's data is fragmented across for three snapshots — after the system has executed $0.5, 1$ and 2 Billion queries. We make two observations. First, for more than 99% of the nodes, their data is fragmented across a very small albeit non-trivial number of shards ($<10\%$ of the shards in the system). This is precisely the case where update pointers help — ZipG needs to touch more than a single server, but touching all servers for query execution has significantly higher overheads than using update pointers. Moreover, as more queries are executed, data gets fragmented across a larger number of shards (also shown in Figure 11).

B. PATH QUERIES, TRAVERSALS & JOINS

We now present evaluation results of ZipG and Neo4j-tuned for three set of queries that are significantly different from the ones evaluated in §5 — regular path queries from [25], graph traversal queries (breadth first search), and graph search queries [8] when implemented using joins[8].

B.1 Regular Path Queries

General regular path queries [30, 33, 47] identify paths in graphs through regular expressions over the edge labels (`edgeTypes` in ZipG terminology) of the graph. Their results are collections of paths, where the concatenation of consecutive edge labels in each path satisfy the regular expression.

Implementation. We implemented *unions of conjunctive regular path queries* (UCRPQs) [26] in ZipG by executing regular expressions over edge labels on ZipG layout, translating them into sequences of operations from ZipG's API. In fact, ZipG is able to execute all regular path queries generated by the gMark benchmark tool [25].

The execution of regular path queries in ZipG begins by obtaining all EdgeRecords corresponding to the first edge in the path expression using `get_edge_record(*,edgeType)`. Subsequently, ZipG identifies the neighbor nodes in these EdgeRecords using `get_edge_data`, and iteratively searches for non-empty EdgeRecords corresponding to the neighbor nodes and the subsequent edge labels in the path expression. To minimize communication overheads, ZipG *ships* the `get_edge_record` requests to the shards that hold the data for the particular node using function shipping (§4.1). ZipG supports recursion in path queries via the Kleene-star ("*") operator by computing the transitive closure of the set of paths identified by the path expression under the Kleene-star. Currently, the transitive closure computation requires collecting all the paths at an aggregator and employs a serial algorithm; this can be further optimized using careful modifications in the underlying data structures and query execution.

Performance. In order to evaluate the performance of regular path queries, we used the gMark [25] benchmark tool to generate both the graph datasets and the path queries. We used gMark's encoding of the schema provided with the LDBC Social Network Benchmark [36], which models user activity in a typical social network. gMark generates 50 queries of widely varying nature [6], ranging from linear path traversals, to branched traversals and highly recursive queries; these can be easily mapped to their Cypher representations [13]. We ran our benchmarks for datasets with varying number of nodes and edges for ZipG and Neo4j; Figure 12 shows results for graphs with 2 million nodes on a single machine setup[9].

Note that given the dataset size, both systems are able to fit their entire data completely in memory. For most queries, ZipG's performance is either comparable to or better than Neo4j. The queries where ZipG outperforms Neo4j by a large margin (*e.g.*, `q18`, `q25`, `q38`, `q48`, `q49`, etc) are typically branched or long linear path traversals, with little or no recursion in them. On the other hand, Neo4j outperforms ZipG for queries that are heavy

[8] ZipG does support joins, but favors executing many join queries using random access instead as discussed in the introduction (§1). The results in §5 present results for the non-join version; here we present results when the same queries are implemented using joins.

[9] Due to the complexity of the queries, both systems timed out (with a time limit of 10 minutes) in executing most queries on graphs with more than 2 million nodes on a single 8 core machine. Moreover, despite significant effort, we could not run these queries on Titan even for smaller graphs due to some bug in the Titan release that supports regular path queries.

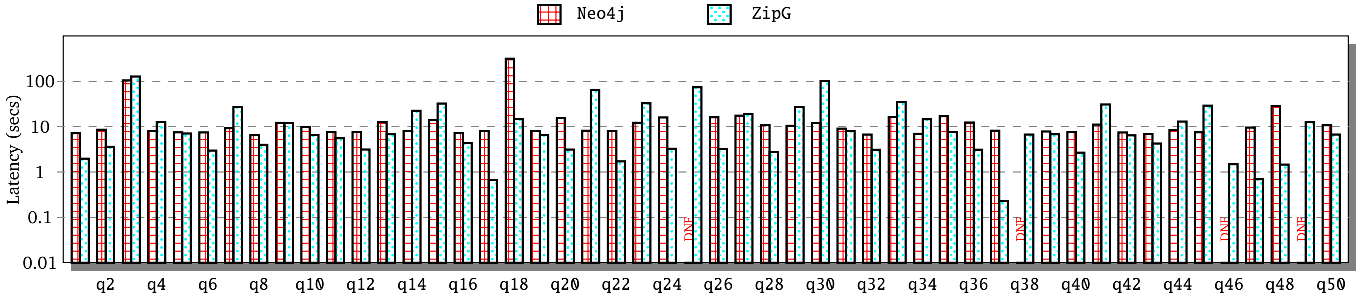
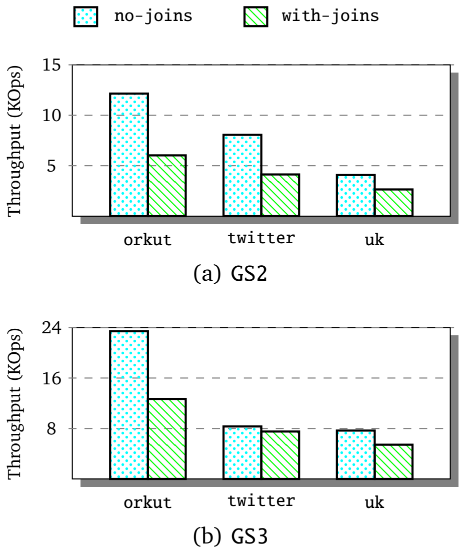


Figure 12: Latency for executing Regular Path Queries from gMark workload. Note that the y-axis has a `log` scale. For most of the queries, ZipG performs significantly better than Neo4j. However, for some queries, ZipG does perform worse than Neo4j. We discuss the properties of the queries that lead to such performance in Appendix B.1.



(a) GS2

(b) GS3

Figure 14: Executing queries with and without joins in ZipG..

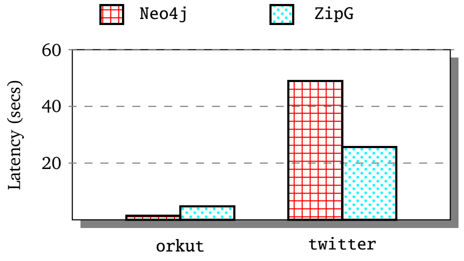


Figure 13: Breadth First Traversal Latency for Neo4j and for ZipG. When the entire graph data fits in memory (orkut), Neo4j performs better than ZipG. However, when the data does not fit in memory for Neo4j, ZipG outperforms Neo4j for graph traversal queries.

on recursion, or where computing the transitive closure becomes a bottleneck in ZipG (*e.g.*, q4, q15, q21, q29, q30, etc.). This is precisely due to the communication and serial bottlenecks in executing transitive closure in ZipG, as discussed above.

B.2 Graph Traversal

We now compare the performance for breadth first traversal of graphs for Neo4j and ZipG on a single machine. The traversals were performed starting at 100 randomly selected nodes, and the average traversal latencies for the two systems are shown in Figure 13. We bound the traversal depth to 5 for the datasets — unbounded depth led to timeouts for Neo4j for the larger dataset (that does not fit in memory).

For the orkut dataset, the graph fits completely in memory for both systems. For this case, Neo4j achieves lower latency. This is because ZipG has overheads to execute queries on the compressed representation; in addition, ZipG stores its graph data across multiple shards, and incurs some aggregation overheads in combining results from different shards. However, for the twitter dataset, Neo4j is no longer able to fit its data in memory, and incurs significantly higher latency for breadth first traversals. This is because even the data for a single node not fitting in memory requires Neo4j to access the slower storage, significantly slowing down the overall graph traversal query. ZipG, on the other hand, is able to maintain its data in memory, and thus achieves lower query latency compared to Neo4j.

B.3 Graph Queries with Joins

ZipG implementation for most queries outlined so far avoids joins due to performance considerations (as discussed in the introduction). However, ZipG does support joins and can be used to implement graph queries that necessitate joins. To illustrate this, we execute queries GS2 and GS3 from the Graph Search workload using joins. In particular, a GS2 query of the form "Find Alice's friends in Ithaca" (Table 3), can be executed both with joins (a join over all of Alice's friends and all people living in Ithaca) and without joins (find all of Alice's friends, and then checks if they live in Ithaca). The same holds true for a GS3 query.

Figure 14 shows the performance for the two execution alternatives in ZipG on a single machine. Clearly, the alternative that avoids joins yields higher throughput across different graph datasets. Intuitively, this is because Alice is likely to have much fewer friends than the people living in Ithaca, and checking if Alice's friends is more efficient than performing a join between her friends and the people living in Ithaca. Moreover, joins additionally have added communication overheads in distributed settings.