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 graph store for serving interactive queries on graphs. ZipG builds upon two key ideas. First, it uses a simple and intuitive layout for storing the input graph as a flat (unstructured) file. This layout admits efficient graph compression, and yet enables efficient query execution directly on compressed graph data. ZipG thus executes a significantly larger fraction of queries in main memory than existing graph stores, achieving performance gains. Second, ZipG uses the idea of fanned updates to efficiently overcome the challenges of updating compressed flat files; a set of update pointers allow ZipG to handle high write rates using log-structured storage while touching partitions necessary for query execution.

We evaluate the performance of ZipG against several popular open-source graph stores for Facebook TAO, LinkBench and graph search workloads. On a single server with 244GB memory, ZipG executes tens of thousands of queries from all these workloads for raw graph data over half a TB. This leads to an order of magnitude (sometimes as much as $23 \times$) higher throughput than Neo4j and Titan. We get similar gains in distributed settings compared to Titan.

1. INTRODUCTION

Serving graph queries is challenging. Unlike batch processing jobs on graphs [30, 31, 37, 41, 50] that run at the scale of minutes, user-facing graph serving queries require millisecond level latency, as well as high throughput [23]. Achieving these goals is challenging because of two reasons.

First, the graph data used for serving queries is massive, not only because of the sheer number of graph nodes and edges but also because many applications associate attributes or properties with each node and edge in the graph [2, 11, 23]. For instance, social networks store a number of attributes for each user node (e.g., age, location, birthplace) and edge (e.g., timestamp, number of exchanged messages, etc.) [23]. Thus, while the entire topology may fit in memory of a single beefy server [37, 46], the associated attributes result in graph data sizes much larger than memory of a typical server [23].

Indexes to serve queries further add to the overall graph size [2, 8–10, 12, 14, 23, 27, 44, 49].

The second reason is that graph serving queries typically exhibit little or no locality. For instance, consider the query: “Find friends of Alice who live in Berkeley”. One possible way to execute this query is to execute two sub-queries — find friends of Alice, and, find people who live in Berkeley — and perform a join or an intersection of results from two sub-queries. Such joins are complex, incurring high computational and bandwidth emissions [5]. Another possibility is to find friends of Alice, and check for each friend whether or not the friend lives in Berkeley. Executing query in this manner alleviates the overheads of join operation, but requires random access into the “location” attribute of each friend of Alice, which may touch arbitrary parts of the graph (potentially across multiple servers). Unless the attributes for each of Alice’s friends is cached, the query latency and system throughput suffers. Thus, to avoid join overheads and to maintain performance, graph serving systems must cache as much data as possible.

ZipG is a new, memory-efficient, distributed graph store for serving interactive queries on large graphs. ZipG achieves low memory footprint using a new, yet simple and intuitive, graph layout that is amenable to efficient compression of graph data. ZipG, thus, caches a large fraction of graph data in-memory precisely solving the problem outlined in the above example. However, what truly differentiates ZipG from existing graph stores is that this low memory footprint does not come at the cost of expressiveness — ZipG allows executing queries on the compressed graph for all published queries from several industrial graph stores including those from Facebook [23], LinkedIn [54] and Twitter [4]. We demonstrate this by implementing and evaluating the entire Facebook TAO [23], LinkBench [19], and graph search [5] functionality on top of ZipG. For instance, using a single server with 244GB of main memory, this implementation serves tens of thousands of TAO, LinkBench and graph search queries for raw graph data as large as half a TB.

Executing queries on compressed graphs is not a novel problem. In fact, designing compression techniques specialized to graphs has been an active area of research for the better part of last two decades [21, 22, 25, 29, 32, 33, 42, 45, 51]. Many existing techniques even support executing queries directly on compressed graphs [22, 25, 29, 32, 33, 42, 45]. However, each of

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

2The cardinality of results for the two sub-queries may be orders of magnitude larger than the final result cardinality.
these techniques is tailored towards a small subset of queries (e.g., extracting edges incident on a node, or subgraph matching), and often ignore queries on node and edge attributes. We compare and contrast ZipG against prior work in Section 6, but note that the main contribution of ZipG is the ability to execute queries as expressive as TAO [23], LinkBench [19] and graph search [5] using a unified memory-efficient representation.

ZipG Techniques. ZipG builds upon Succinct [17], a system that supports random access and search queries on compressed unstructured data and key-value pairs. Section 3 outlines challenges that ZipG has to resolve to build an interactive graph serving system. The most fundamental of these challenges 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 [9] heavily use pointers to store neighbors and properties for each node and edge. While flexible in 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 [14], 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 [23], KV abstraction is not suited for graph queries and leads to degraded performance even for simple queries.

ZipG uses a new data layout that, while simple and intuitive, provides both scalability and flexibility by operating on flat unstructured files (Section 3). This layout admits efficient graph compression; perhaps more interestingly, this layout allows using the two simple primitives of Succinct (random access and search) to implement queries as expressive as TAO, LinkBench and graph search directly on a memory-efficient representation of ZipG’s layout.

There are two widely acknowledged challenges associated with storing data using a compressed representation, that are not necessarily specific to graph data. The first challenge is in supporting high write rates. ZipG resolves this challenge using the idea of fanned updates — each server in ZipG stores a carefully designed set of update pointers that allow achieving the efficiency of log-structured storage (in terms of handling high write rates) with the performance of executing queries on a memory-efficient representation. In essence, these update pointers ensure that during query execution, ZipG touches exactly those partitions (and bytes within the partition) that are necessary for query execution. Second, storing data using a compressed representation also introduces new challenges in providing strong consistency guarantees and transactions. ZipG currently does not attempt to resolve this challenge. While several graph stores used in production [4, 23, 54] make a similar design choice, extending ZipG to provide such guarantees is an interesting future direction.

ZipG Performance. We evaluate ZipG against Neo4j [9] and Titan [14], two popular open-source graph stores used in several production clusters. All our experiments run in the wild on a set of Amazon EC2 machines, and use three workloads — Facebook TAO [23], LinkBench for interactive graph queries from SIGMOD'13 [19] and graph search queries [5]. 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 outperforms Neo4j and Titan in terms of system throughput by an order of magnitude (sometimes by as much as 23×).

2. DATA MODEL AND INTERFACE

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

2.1 ZipG Data Model

ZipG uses the property graph model [9, 11, 14, 23], with graph data comprising of nodes, edges, and a (potentially varying) number of properties associated with nodes and edges.

Nodes and Edges. ZipG uses usual definitions of nodes and edges; edges could be directly or undirected. However, to model applications where edges may represent different types (e.g., comments, likes, relationships) [23], 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 may potentially have a different EdgeType and may optionally have a Timestamp. The precise representation is described in Section 3.

Node and Edge Properties. Each Node and edge 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, Berkeley), (nickname, cool)}. ZipG PropertyLists 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 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 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.

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

\(^3\)Pointer chasing during query execution requires multiple accesses to secondary storage and/or different servers, leading to undesired bottlenecks.

\(^4\)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 Section 2.2 for description.

<table>
<thead>
<tr>
<th>API</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>g = compress(graph)</td>
<td>Compress graph.</td>
</tr>
<tr>
<td>List&lt;String&gt; g.get_node_property(nodeID, propertyIDs)</td>
<td>Get Alice’s age and location.</td>
</tr>
<tr>
<td>List&lt;NodeID&gt; g.get_node_ids(propertyList)</td>
<td>Find people in Berkeley who like Music.</td>
</tr>
<tr>
<td>List&lt;NodeID&gt; g.get_neighbor_ids(nodeID, edgeType, propertyList)</td>
<td>Find Alice’s friends who live in Boston.</td>
</tr>
<tr>
<td>EdgeRecord g.get_edge_record(nodeID, edgeType)</td>
<td>Get all information on Alice’s friends.</td>
</tr>
<tr>
<td>Pair&lt;TimeOrder&gt; g.get_edge_range(edgeRecord, tlo, thI)</td>
<td>Section 2.2</td>
</tr>
<tr>
<td>EdgeData g.get_edge_data(edgeRecord, timeOrder)</td>
<td>Find Alice’s most recent friend.</td>
</tr>
<tr>
<td>g.append(nodeID, PropertyList)</td>
<td>Append new node for Alice.</td>
</tr>
<tr>
<td>g.append(nodeID, edgeType, edgeRecord)</td>
<td>Append new edges for Alice.</td>
</tr>
<tr>
<td>g.delete(nodeID)</td>
<td>Delete Alice from the graph.</td>
</tr>
<tr>
<td>g.delete(nodeID, edgeType, destinationID)</td>
<td>Delete Bob from Alice’s friends list.</td>
</tr>
</tbody>
</table>

Table 1 lists the interface exposed by ZipG, and an example for each individual query. Applications submit the graph data (represented using the property model in Section 2.1) to ZipG, and generate a memory-efficient representation using compress(graph). Applications can then invoke powerful primitives (Table 1) as if the graph was uncompressed; 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 Section 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 θ, the query get_neighbor_ids(Alice, θ, {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, θ, *) ∩ 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 DESIGN

In this section, we present ZipG’s design, implementation, and associated tradeoffs. We start by briefly outlining the necessary details from Succinct [17] that ZipG builds upon; this will allow us to discuss the design tradeoffs in ZipG layout.

3.1 Succinct Background

Succinct [17] is a distributed data store that supports random access and search queries on a compressed representation of the input data, with several interfaces to the applications: a flat file interface for executing queries on unstructured data, and key-value (KV) interface for queries on semi-structured data.

- **Random access** via extract(offset, 1len) query on flat files, that returns 1len many characters starting at arbitrary offset; for KV 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 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 one is suffix array [13,43] that enables the search functionality, and the second one is inverse-suffix array that enables the random access functionality. As such, both of these data structures have size $n \log n$ for a flat file with $n$ characters. Succinct achieves a memory-efficient representation for these arrays using sampling — only a few sampled values (e.g., for sampling rate $\alpha$, value at indexes $0, \alpha, 2\alpha, \ldots$) from these two arrays are stored. A third array allows computing the required unsampled values during query execution. This third array has an interesting structure and admits extremely small storage overhead. The tradeoff is that for a sampling rate of $\alpha$, the storage requirement for the first two arrays is $2n/\alpha + \log n$. The number of operations required for computing each unsampled value is $O(1)$. 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 [17]. The rest of the paper focuses on the ZipG data structures and query algorithms that use the random access and search functionality above in an efficient manner to enable a wide range of interactive queries on a memory-efficient representation of graph data.
3.2 ZipG overview

It is a priori conceivable that most of the “heavy lifting” in building a memory-efficient distributed graph store could be delegated to Succinct [17]. However, ZipG has to resolve a number of fundamental challenges 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 [37] and Ligra [50] 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 [23] 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.

ZipG’s graph layout uses two flat unstructured 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 Section 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 Section 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 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 (Section 3.5).

3.3 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 and stored as a PropertyID → (order, delimiter) map, where order is the lexicographic ranking of the propertyID among all propertyIDs. This map is stored in memory.

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 •42BerkeleyAlly, 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 Berkeley in above example) [23]. 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 1en, 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•42BerkeleyAlly.

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

Implementing queries from Table 1. It is easy to see that the NodeFile design allows implementing get_node_property

---

**Figure 1:** An example for describing the layout of NodeFile. See description in Section 3.3. Swap keys and flat-file
query using two array lookups (one for property delimiter and one for ProperList offset), and one extra byte (for accessing property Value length) in addition to extracting the Property Value itself (using random access primitive from Section 3.1). Implementing get_node_\_ids is more interesting; we explain this using an example. Suppose the query specifies \{"\text{nickname}\" = \text{"Ally\"}\} as the propertyList. Then, Zip\_G first finds the delimiter of the specified PropertyID (\(*\), for \text{nickname} and the next lexicographically larger PropertyID (in this case, \(\dagger\) for end-of-record delimiter). It then prepends and appends Ally by \(*\) and \(\dagger\), respectively and uses the search primitive from Section 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 properties. Recall from Section 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 $\text{NodeID}\#\text{EdgeType}$, 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 post time [23]). Efficiently executing such queries requires the ability to perform binary search on timestamps. Zip\_G stores the timestamps in each EdgeRecord in sorted order. To aid binary search, Zip\_G 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, 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 complicates random access on timestamps since extracting a timestamp requires extracting all the timestamps before it. Storing timestamps using delta encoding [47] 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.

Zip\_G 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, Zip\_G stores the fixed length for each EdgeRecord 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 $i^{th}$ timestamp and $i^{th}$ 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. Zip\_G uses a fixed length representation similar to timestamps for destinationIDs and accordingly, stores this length in corresponding metadata (DLength in Figure 2).

**Edge Properties.** As with destinationIDs, edge propertyLists are ordered such that $i^{th}$ timestamp and $i^{th}$ 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 as in Section 3.3. Note that Zip\_G 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).** The get_\_edge_\_record function returns the EdgeRecord for a given (sourceID, edgeType) pair and is implemented using search($\text{sourceID}\#\text{edgeType}$) on the EdgeFile. This returns the offset for the EdgeRecord. Using the metadata in the EdgeRecord, Zip\_G 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

Zip\_G uses a log structured approach to efficiently handle node and edge updates. Specifically, new data is appended into a query-optimized LogStore, which is periodically merged into above memory-optimized representation. 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 stores use the above design for updates, as outlined in Section 3.2, Zip\_G has to resolve several challenges to efficiently use such a design for graph stores.
The main challenge arises due to fragmentation — before recently written data is merged into memory-optimized representation, node and/or edge data may be fragmented across multiple partitions. Without any additional data structures, this would lead to ZipG queries touching all partitions resulting in degraded throughput.

To achieve write efficiency of log structured storage along with query efficiency of touching necessary partitions only, ZipG uses Fanned Updates (Figure 3). Consider a static graph, that is, a graph that has never been updated since the initial upload to ZipG. The partitioning scheme used in ZipG (Section 3.6) ensures that most queries are first forwarded to a single partition. At a high level, Fanned Updates avoid touching all partitions 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 updated data. As the graph is updated or is defragmented over time, these update pointers are updated as well. Our evaluation suggests that even for workloads with high write rates [19], the overhead of storing these pointers is minimal. ZipG, thus, keeps these pointers uncompressed which in turn also allows for efficiently updating these pointers.

3.6 System Implementation

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

Graph Partitioning, Fault tolerance and Load Balancing. ZipG design is orthogonal to the underlying graph partitioning and load balancing schemes, as well as fault tolerance mechanisms. Several previous studies have established that efficient partitioning of graphs is a hard problem [18, 39, 40], and several heuristics exist [14, 30, 34]. ZipG currently uses a simple hash-partitioning scheme — it creates a number of partitions, default being one per core, and hash partitions the NodeIDs on to these shards. All the data corresponding to NodeID (PropertyList and edges incident on NodeID) are stored in that partition. This ensures that ZipG executes as many queries as possible locally, that is, without any inter-partition communication. Finally, each of the partitions is transformed into the ZipG layout (Section 3). Highly efficient partitioning schemes have been proposed recently [34], which can further improve the performance of ZipG.

ZipG currently uses replication for fault tolerance. When enabled, queries are load balanced across the replicas. While orthogonal to ZipG design, extending ZipG implementation to incorporate more efficient fault tolerance and skew-tolerant techniques [26, 36, 48] is an interesting future direction.

Data Persistence and Caching. To achieve data persistence, ZipG stores NodeFile, EdgeFile, LogStore data and update pointers on secondary storage as serialized flat files. ZipG maps these files to virtual memory using the mmap system call, and all writes 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 [3, 52, 53]. 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 (Section 3.5), that are fast enough and do not become
system bottleneck.

4. ZipG EXPRESSIVENESS

ZipG interface is rich enough to implement functionalities from several industrial graph stores. We have implemented Facebook TAO [23], LinkBench [19] and Graph Search [5] on top of ZipG. We discuss this implementation next.

Table 2 shows how to implement TAO and LinkBench queries using ZipG interface. Note that TAO and LinkBench have the same set of queries, but significantly different query distribution (more write-heavy). These queries can be broadly classified into two categories based on whether or not timestamps are involved in the queries. Queries that do not operate on timestamps (obj_get and assoc_count in Table 2) are easily mapped to ZipG API — get_node_property(id, *), and the `EdgeCount` metadata using get_edge_record, respectively. Queries that do use timestamps essentially translate to `assoc_get` query in TAO. ZipG is particularly efficient for these kind of queries since, as discussed in Section 3, ZipG can efficiently perform binary search on timestamps and return corresponding edges and their properties. Algorithms 1 and 2 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 [5]. Implementing graph search queries is even simpler in ZipG since most queries directly map to ZipG API, as shown in Table 3.

5. EVALUATION

We now 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 [9] 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 [14] with Cassandra 2.2.1 [38] 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. 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 8 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. As such, we use TAO and LinkBench workloads (original query distributions, Table 2) and a synthetic Graph Search workload (Table 3) for our evaluation. 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. We also considered other benchmarks [6, 20, 24, 28] but had to discard them either because

---

**Table 2: Queries in TAO [23] and LinkBench [19] workloads.**

<table>
<thead>
<tr>
<th>Query</th>
<th>Execution in ZipG</th>
<th>Execution in TAO %</th>
<th>Execution in LinkBench %</th>
</tr>
</thead>
<tbody>
<tr>
<td>assoc_range</td>
<td>Algorithm 1</td>
<td>40.8</td>
<td>50.6</td>
</tr>
<tr>
<td>obj_get</td>
<td>get_node_property</td>
<td>28.8</td>
<td>12.9</td>
</tr>
<tr>
<td>assoc_get</td>
<td>Algorithm 2</td>
<td>15.7</td>
<td>0.52</td>
</tr>
<tr>
<td>assoc_count</td>
<td>get_edge_record</td>
<td>11.7</td>
<td>4.9</td>
</tr>
<tr>
<td>assoc_time_range</td>
<td>Algorithm 2</td>
<td>2.8</td>
<td>0.15</td>
</tr>
<tr>
<td>assoc_add</td>
<td>append</td>
<td>0.1</td>
<td>9.0</td>
</tr>
<tr>
<td>obj_update</td>
<td>delete, append</td>
<td>0.04</td>
<td>7.4</td>
</tr>
<tr>
<td>obj_add</td>
<td>append</td>
<td>0.03</td>
<td>2.6</td>
</tr>
<tr>
<td>assoc_del</td>
<td>delete</td>
<td>0.02</td>
<td>3.0</td>
</tr>
<tr>
<td>obj_del</td>
<td>delete</td>
<td>&lt; 0.01</td>
<td>1.0</td>
</tr>
<tr>
<td>assoc_update</td>
<td>delete, append</td>
<td>&lt; 0.01</td>
<td>8.0</td>
</tr>
</tbody>
</table>

**Table 3: The Graph Search Workload and implementation using ZipG API.**

<table>
<thead>
<tr>
<th>QID</th>
<th>Example</th>
<th>Execution in ZipG</th>
</tr>
</thead>
<tbody>
<tr>
<td>GS1</td>
<td>All friends of Alice</td>
<td>get_neighbor_ids(id, *, *)</td>
</tr>
<tr>
<td>GS2</td>
<td>Alice’s friends in Berkeley</td>
<td>get_neighbor_ids(id, *, {p1})</td>
</tr>
<tr>
<td>GS3</td>
<td>Musicians in Berkeley</td>
<td>get_node_ids({p1, p2})</td>
</tr>
<tr>
<td>GS4</td>
<td>Close friends of Alice</td>
<td>get_neighbor_ids(id, type, *)</td>
</tr>
<tr>
<td>GS5</td>
<td>All data on Alice’s friends</td>
<td>assoc_range(id, type, &amp;, *)</td>
</tr>
</tbody>
</table>

**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 ← Ø
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

**Algorithm 2** assoc_time_op(id1, atype, id2set, hi, lo, limit)

Obtain at most limit edges with source node id1, edge type atype, timestamp in the range [hi, lo), and destination in id2set.

1: Initialize id2set and limit based on query.
2: rec ← get_edge_record(id1, atype)
3: (beg, end) ← get_time_range(rec, hi, lo)
4: results ← Ø
5: for i ← beg to min(beg+limit, end) do
6:   edgeEntry ← get_edge_data(rec, i)
7:   Add edgeEntry to results if destination ∈ id2set
8: end for
9: return results
Table 4: Datasets used in our evaluation.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#nodes &amp; #edges</th>
<th>Type</th>
<th>On-disk Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>orkut [35]</td>
<td>~3M &amp; ~117M</td>
<td>social</td>
<td>20 GB</td>
</tr>
<tr>
<td>twitter [22]</td>
<td>~41M &amp; ~1.5B</td>
<td>social</td>
<td>250 GB</td>
</tr>
<tr>
<td>uk [22]</td>
<td>~105M &amp; ~3.7B</td>
<td>web</td>
<td>636 GB</td>
</tr>
<tr>
<td>small</td>
<td>~32.3M &amp; ~141.7M</td>
<td>social</td>
<td>20 GB</td>
</tr>
<tr>
<td>medium</td>
<td>~403.6M &amp; ~1.76B</td>
<td>social</td>
<td>250 GB</td>
</tr>
<tr>
<td>large</td>
<td>~1.02B &amp; ~4.48B</td>
<td>social</td>
<td>636 GB</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Neo4j</th>
<th>Titan-C</th>
<th>Titan</th>
<th>ZipG</th>
</tr>
</thead>
<tbody>
<tr>
<td>orkut/linkbench-small</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>twitter/linkbench-medium</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>uk/linkbench-large</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 5: ZipG’s storage footprint (Section 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. Note that the y-scales differ for the two plots.

They focused on graph batch processing [24], single-server settings [6, 28], or only a few specific queries [20].

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 [23]. 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 [7] 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 the 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). The main reason is the secondary indexes stored by Neo4j and Titan to support various

8 Intuitively, Titan uses delta encoding for edge destinationNodeIDs, and variable length encoding for node and edge attributes [15] which leads to high CPU overhead during query execution. Moreover, enabling LZ4 compression for Cassandra’s SSTables reduces the storage footprint for Titan, but requires data decompression for query execution.

queries efficiently; ZipG, on the other hand, executes queries efficiently directly on a memory-efficient representation of the input graph. Table 5 shows which of the graphs fit completely in memory for which system.

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 than Titan compressed (for reasons discussed in Footnote 8). 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. This leads to ZipG achieving significantly high throughput for the obj_get query (Figure 6(c)). 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 (Figure 6(d)). 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 Figure 6(b) and 6(f).

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 (Figure 6). 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.
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 [19] — as a result, the average number of edges accessed per query is much larger than in the TAO workload, leading to lower query throughput.

We also observe that Neo4j and Titan observe very low throughput for all three datasets. Neo4j’s execution of read-only queries is relatively fast, e.g., assoc_range and obj_get in Figure 7(a) and 7(b). However, the mutation of graph elements becomes a bottleneck for Neo4j. Intuitively, each write (e.g., assoc_add in Figure 7(c)) or update operation (e.g., assoc_update and obj_update in Figure 7(d) and 7(e)) requires modifications at multiple random locations due to Neo4j’s pointer-based approach, which leads to significantly reduced throughput. 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 like assoc_range (Figure 7(a)) is significantly lower because Cassandra is not optimized for range queries.

ZipG avoids both of the above issues and achieves significantly higher throughput for the small and medium datasets. In particular, all graph writes and updates are isolated to a write-optimized LogStore through Fanned Updates (Section 3.5), while edge-based operations do not need to scan the entire neighborhood to filter the required edges (Section 3.4). However, ZipG’s throughput drops significantly for the large dataset; this is due to the relatively lower compressibility of LinkBench generated graphs, which prevents a crucial data-structure in the underlying Succinct representation for the NodeFile from fitting in memory. As a consequence, ZipG observes a significantly reduced throughput for the obj_get query as seen in Figure 7(b).

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 (Figure 8) follow a very similar pattern as for TAO workload (Figure 6), 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 $1.23 \times$ 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 $3 \times$ higher throughput than Neo4j-Tuned.

The second overhead of ZipG is highlighted in Figure 8(c) for queries like “Find musicians in Berkeley”. 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.

### 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 $2 \times$ 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 $2.5 \times$ higher throughput in distributed settings compared to single server setting. Note that our distributed servers have $10 \times 8$ 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.

**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.

### 6. RELATED WORK

**Graph Stores.** In contrast to graph batch processing systems [30, 31, 37, 41, 50], graph stores [2, 8–10, 12, 14, 23, 27, 44, 49] focus on queries that are often user-facing and require millisecond-level latency. We already compared the performance of Neo4j and Titan against ZipG in Section 5. There are other systems, however. For instance, Virtuoso [16], GraphView [8] and Sparksee [12] use secondary indexes to support efficient graph traversals; these systems suffer from storage overhead problems similar to Neo4j (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 [21, 22, 25, 29, 32, 33, 42, 45, 51]. 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 [1, 5, 23, 54] involving node and edge attributes. ZipG achieves compression without compromising expressiveness, and is able to execute all queries from Facebook TAO [23], LinkBench [19] and graph search [5].

7. 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 execute tens of thousands of queries from TAO, LinkBench and Graph Search workloads for a graph with over half a TB of data on a single 244GB server. This leads to an order of magnitude (sometimes as much as $23 \times$) higher throughput than Neo4j and Titan. We get similar gains in distributed settings compared to Titan, where ZipG achieves as much as $20 \times$ higher throughput.

8. REFERENCES

[8] Microsoft GraphView. 
    tinkerpop/blueprints/wiki/Property-Graph-Model.
    docs/titan/current/data-model.html.
[17] A. Ahmed, N. Shervashidze, S. Narayanamurthy, 
    R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: 
    Virtuoso Universal Server. 
[18] Bharat, Krishna and Broder, Andrei and Henzinger, 
    N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, 
    P. Boldi and S. Vigna. The Webgraph Framework I: 
    Suffix Array. 
[19] Jure Leskovec and Andrej Krevl. SNAP Datasets: 
    http://snap.stanford.edu/data.
    Benchmarking graph databases on the problem of 
    compressing social networks. In *New Trends in Database and 
[21] 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.
[22] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, 
    O. Erling, and P. Boncz. Graphalytics: A big data 
    benchmark for graph-processing platforms. In *ACM 
    GRADES*, 2015.
[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)*, 2009.
    a Workload-Driven Approach to Database Replication 
    and Partitioning. *Proceedings of the VLDB Endowment*, 
    2010.
[25] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, 
    Weaver: A High-Performance, Transactional Graph 
    Store Based on Refinable Timestamps. *CoRR*, 
    abs/1509.08443, 2015.
[26] 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)*, 2015.
[27] W. Fan, J. Li, X. Wang, and Y. Wu. Query Preserving 
    Graph Compression. In *ACM International Conference 
    on Management of Data (SIGMOD)*, 2012.
[28] 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)*, 2012.
[29] 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.
    Weaver: A High-Performance, Transactional Graph 
    Store Based on Refinable Timestamps. *CoRR*, 
    abs/1509.08443, 2015.
[31] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, 
    Weaver: A High-Performance, Transactional Graph 
    Store Based on Refinable Timestamps. *CoRR*, 
    abs/1509.08443, 2015.
    Franklin, and I. Stoica. GraphX: Graph Processing in a 
    Distributed Dataflow Framework. In *USENIX 
    Symposium on Operating Systems Design and 
    Implementation (OSDI)*, 2014.
[33] S. Suresh. The connectivity server: Fast access to linkage 
    information on the web. In *USENIX Technical 
    Workshop on Large-Scale Compression Techniques*. 
[34] S. Suresh. The connectivity server: Fast access to linkage 
    information on the web. In *USENIX Technical 
    Workshop on Large-Scale Compression Techniques*. 
[35] 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)*, 2012.
    Franklin, and I. Stoica. GraphX: Graph Processing in a 
    Distributed Dataflow Framework. In *USENIX 
    Symposium on Operating Systems Design and 
    Implementation (OSDI)*, 2014.
[37] 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.
[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.
    Franklin, and I. Stoica. GraphX: Graph Processing in a 
    Distributed Dataflow Framework. In *USENIX 
    Symposium on Operating Systems Design and 
    Implementation (OSDI)*, 2014.
[40] 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.
[41] 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.
[42] 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.


