ZipG: A Memory-efficient Graph Store for Interactive Queries

3/7/19

Authors: Anurag Khandelwal, Zongheng Yang,
Evan Ye, Rachit Agarwal*, Ion Stoica

Presented by: Tuhin Tiwari

CS 848: Big Data Management Platforms

UNIVERSITY OF WATERLOO

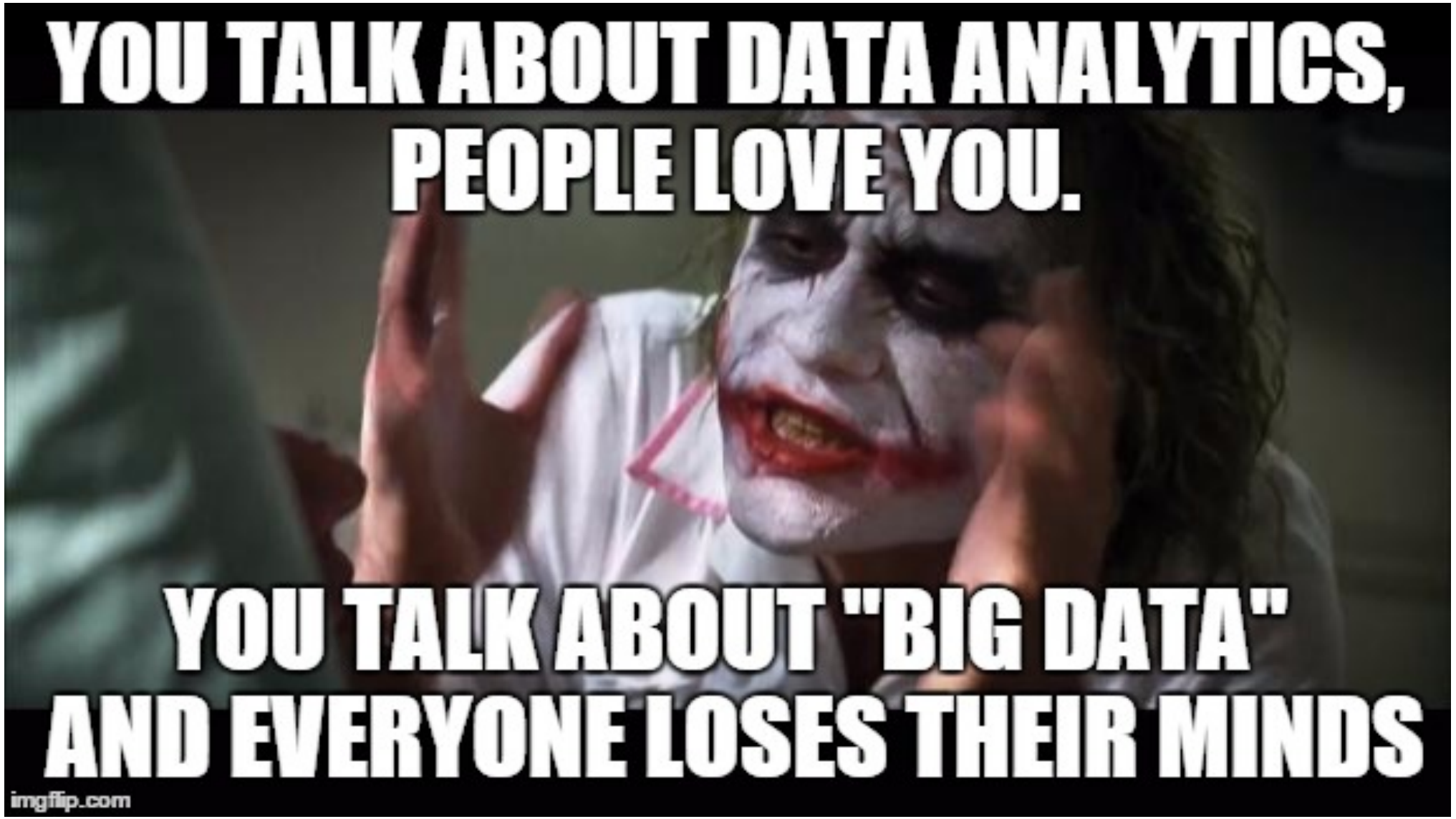
Content

- Introduction
- ZipG API
- ZipG Implementation
- Evaluation
- ZipG Pros & Cons
- Conclusion
- Discussion

UNIVERSITY OF WATERLOO

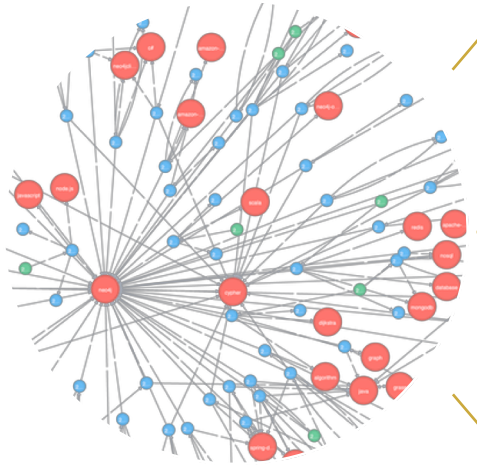
INTRODUCTION

UNIVERSITY OF
WATERLOO



UNIVERSITY OF WATERLOO

What's the problem?



Distributed

- How do you achieve execution of large fraction of queries (possibly in a distributed setting)

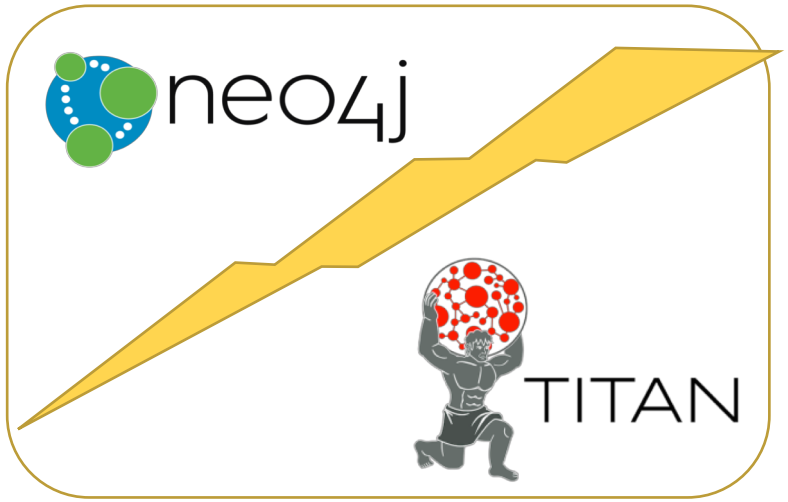
Memory-Efficient

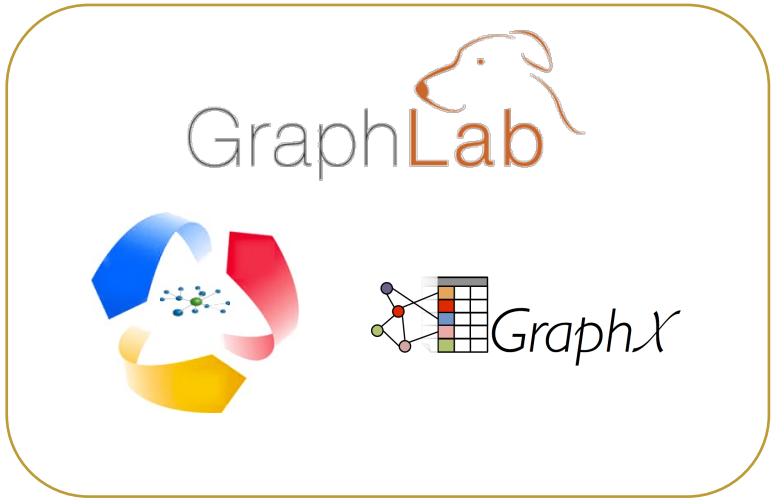
- Which is stored in-memory

Interactive Graph Queries

- to achieve query interactivity

Existing Graph Systems



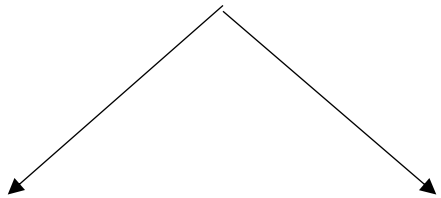


Flexibility Vs Scalibility

Offline Graph Analytics

Query Interactivity

- High performance for interactive user-facing queries
- Millisecond-latency & High throughput
- Query : Find friends of Alice who live in Ithaca.

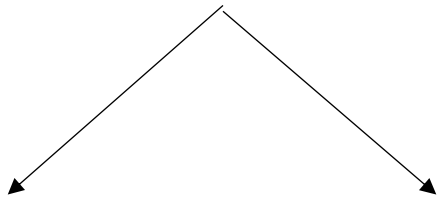


2 sub-queries. Compute final result using join or intersection.

Find friends of Alice. Then for each friend, check whether or not friend lives in Ithaca

Query Interactivity

- High performance for interactive user-facing queries
- Millisecond-latency & High throughput
- Query : Find friends of Alice who live in Ithaca.



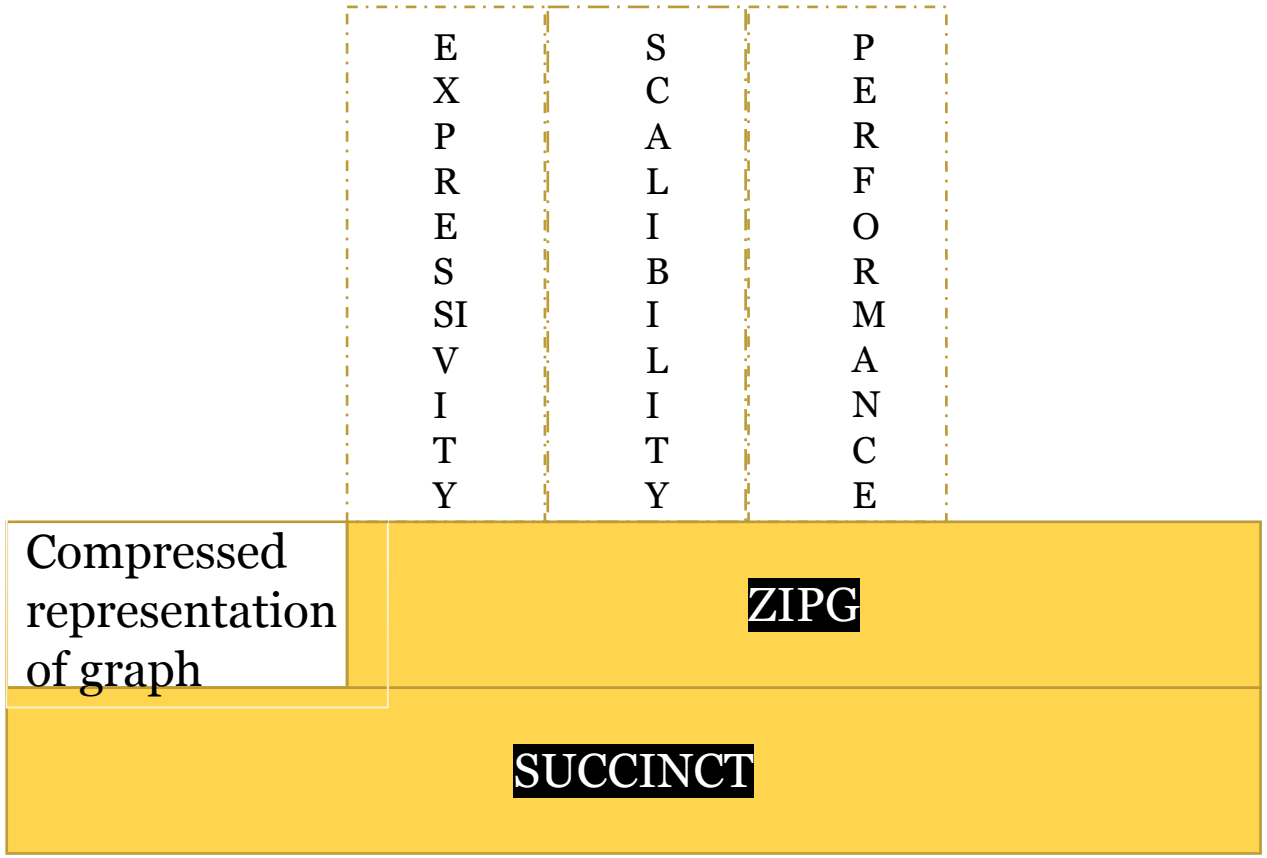
2 sub-queries. Compute final result using join or intersection

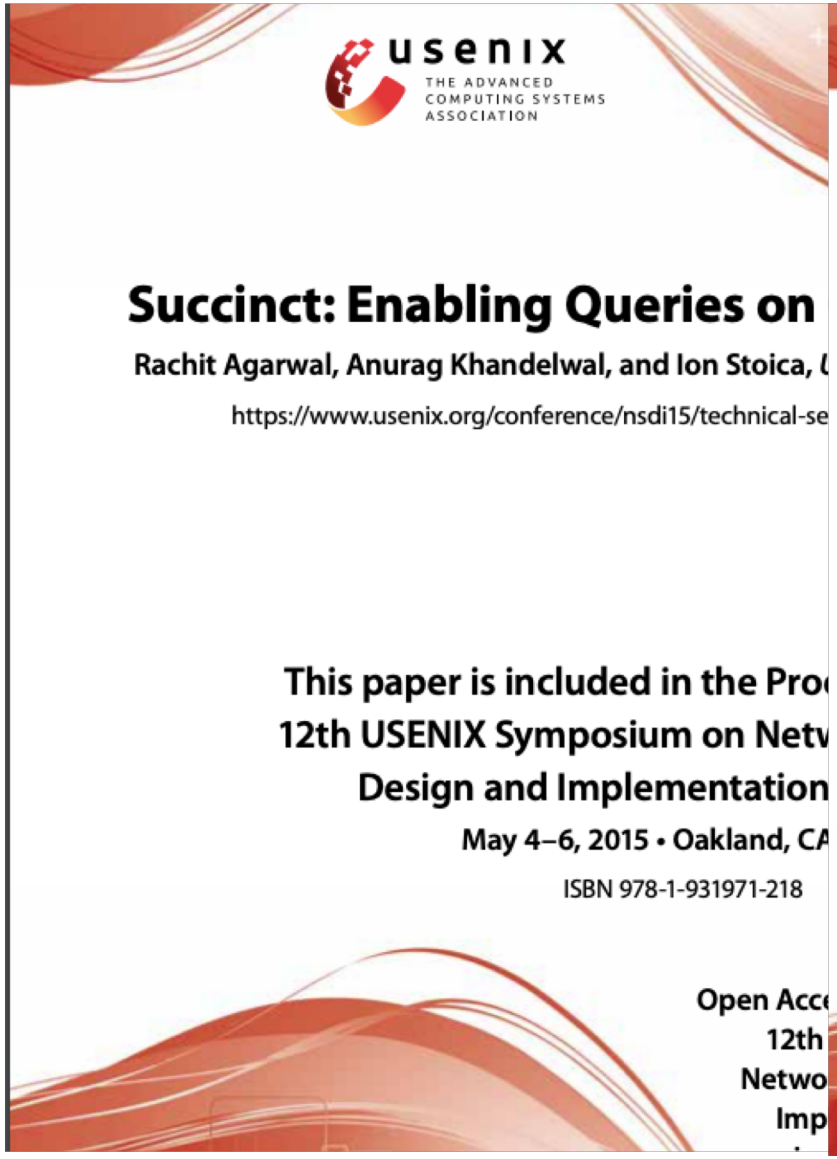
Join Bomb

Find friends of Alice. Then for each friend, check whether or not friend lives in Ithaca

Random access & locality

ZipG





usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

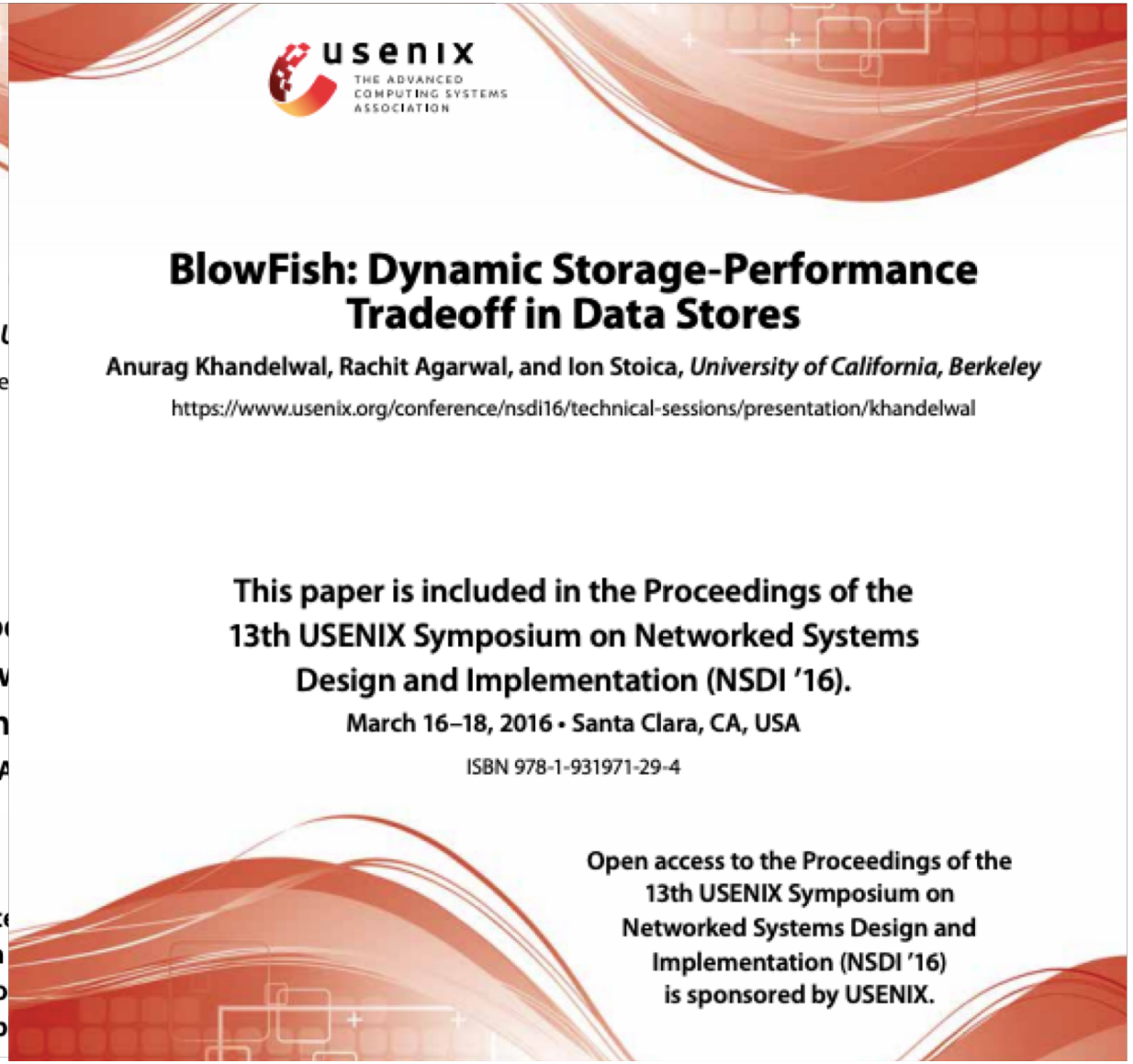
Succinct: Enabling Queries on

Rachit Agarwal, Anurag Khandelwal, and Ion Stoica, U

https://www.usenix.org/conference/nsdi15/technical-se

This paper is included in the Proc
12th USENIX Symposium on Netw
Design and Implementation
May 4–6, 2015 • Oakland, CA
ISBN 978-1-931971-218

Open Acc
12th
Netwo
Imp



usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores

Anurag Khandelwal, Rachit Agarwal, and Ion Stoica, *University of California, Berkeley*

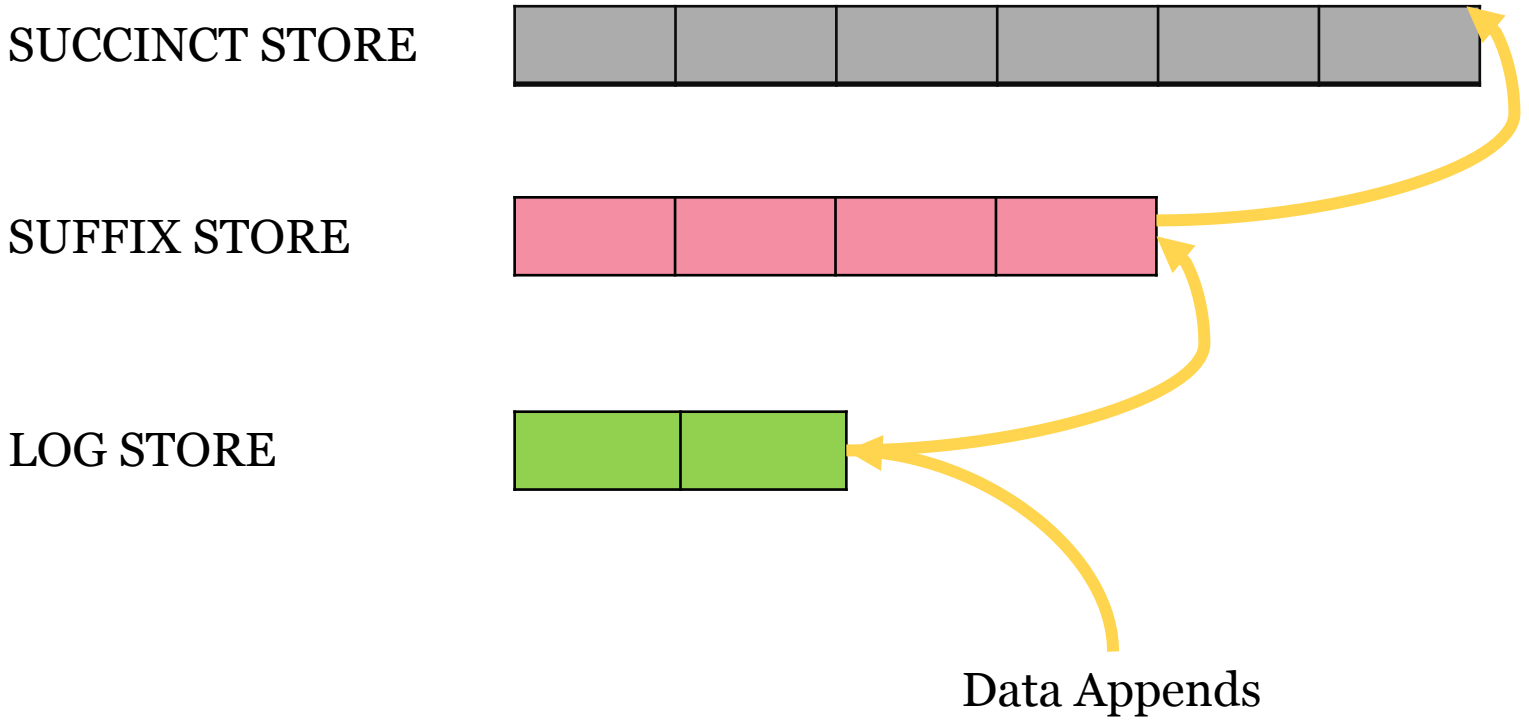
https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khandelwal

This paper is included in the Proceedings of the
13th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '16).
March 16–18, 2016 • Santa Clara, CA, USA
ISBN 978-1-931971-29-4

Open access to the Proceedings of the
13th USENIX Symposium on
Networked Systems Design and
Implementation (NSDI '16)
is sponsored by USENIX.

UNIVERSITY OF WATERLOO

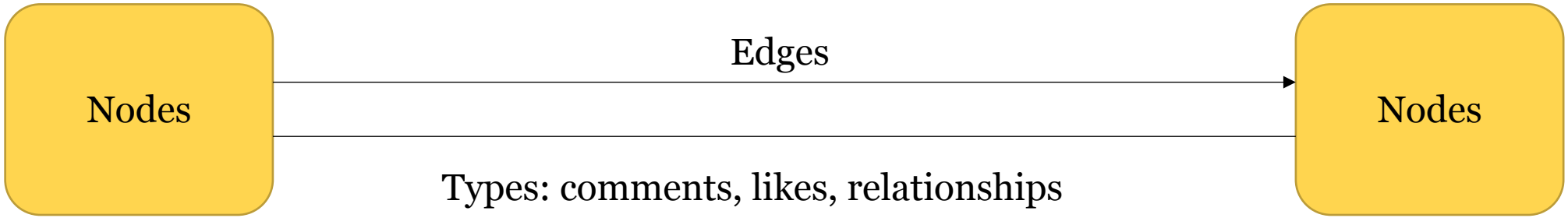
Succinct's Data Compression



ZIPG API

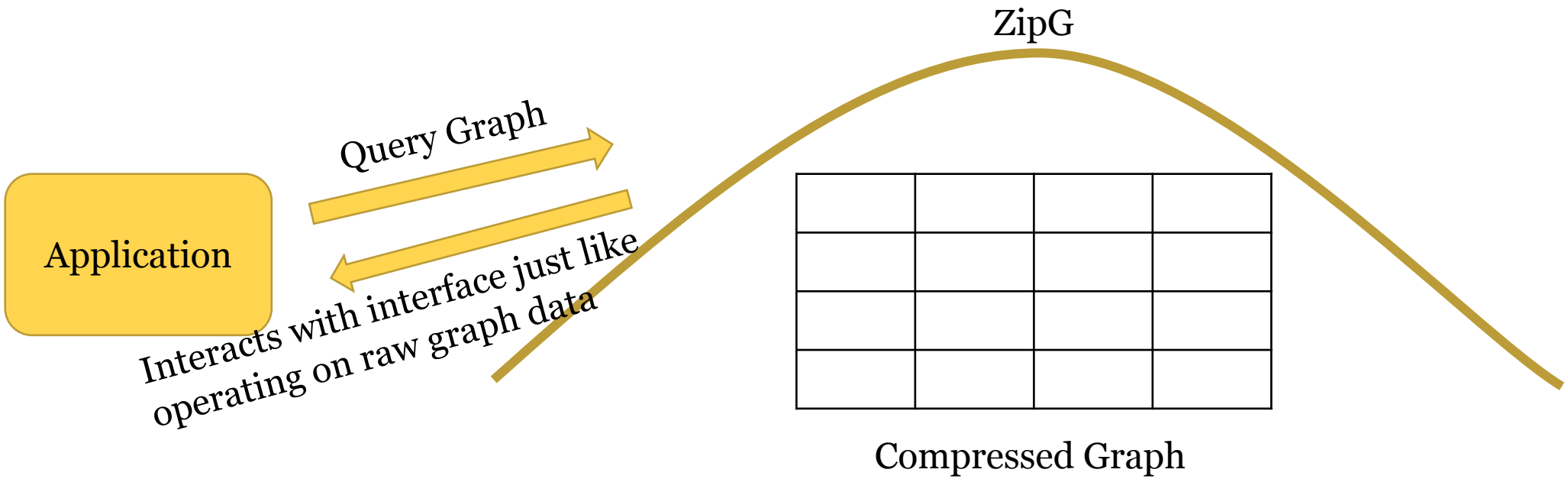
UNIVERSITY OF WATERLOO

ZipG's Data Model

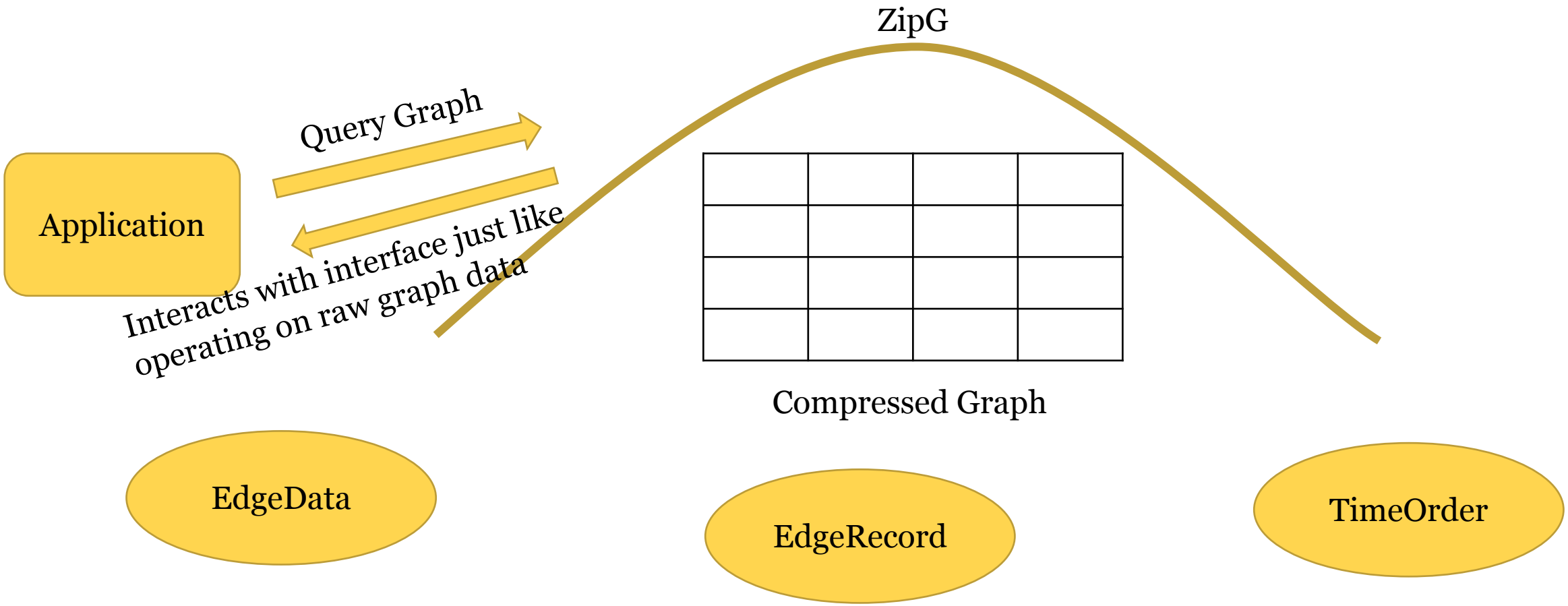


PropertyList
Tuple : {*PropertyID, PropertyValue*}

ZipG's Interface



ZipG's Interface



ZipG API

- Exposes a functionally rich API to implement all queries from TAO, LinkBench, and GraphSearch.
- Node-based

`List<NodeID> g.get_neighbor_ids(nodeID, edgeType, propertyList)`	Find Alice's friends who live in Boston.

- Edge-based

`EdgeRecord g.get_edge_record(nodeID, edgeType)`	Get all information on Alice's friends.

- Delete, Append and Update

Graph Representation - NodeFile

- NodeIDs and NodeProperties.
- Small metadata - length of PropertyValues
- 3 Data structures : PropertyID, Flat file, 2D array

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‡

Graph Representation - NodeFile

- NodeIDs and NodeProperties.
- Small metadata - length of PropertyValues
- 3 Data structures : PropertyID, Flat file, 2D array

Latency b/n Storage and Random Access

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‡

Graph Representation - EdgeFile

- Edges and EdgeProperties.
- 3 tuples – {sourceNodeID, destinationNodeID, EdgeType}
- Small metadata - EdgeCount
- Edge record contains(from left to right) :

$NodeID#EdgeType

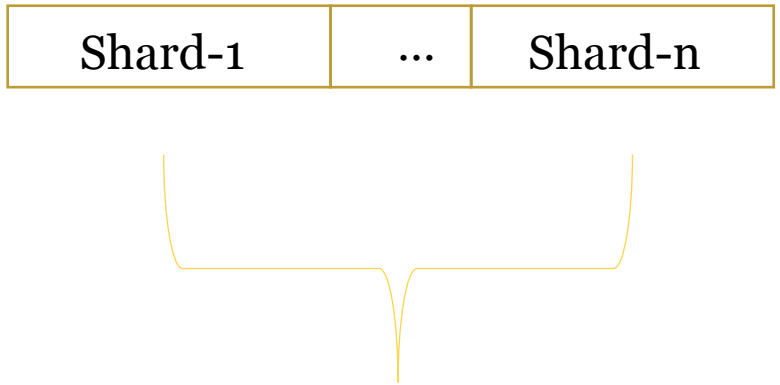
  - Timestamps, DestinationIDs, PropertyList
  - Doesn't support search on Edge PropertyList.

ZIPG IMPLEMENTATION

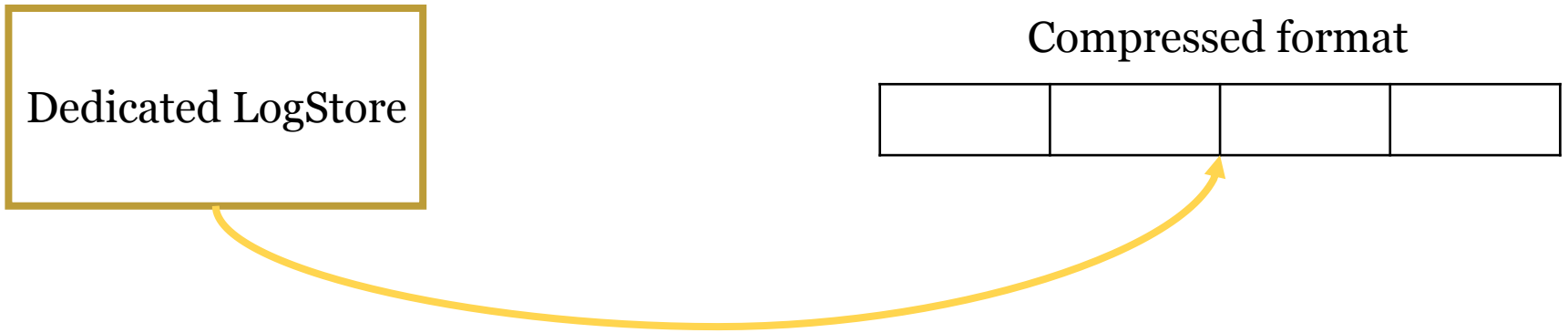
UNIVERSITY OF WATERLOO

Graph Partitioning (Sharding)

- Hash-partitioning scheme – NodeID maps to shards
- Default : 1 per core
- Data corresponding to NodeID (PropertyList and EdgeInformation) is stored in each shard.
- All node and edge-data associated with a node is co-located on the same shard.



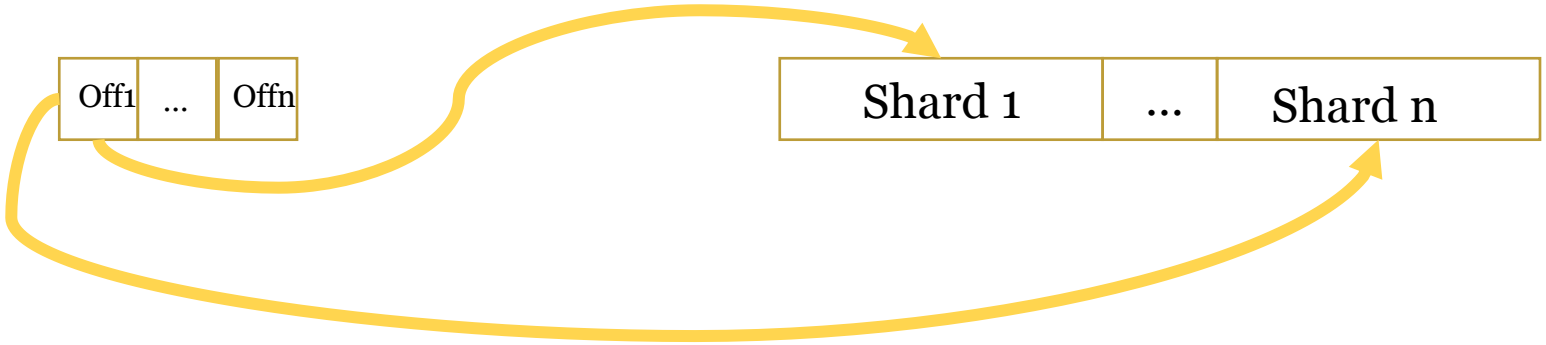
Log Store Server



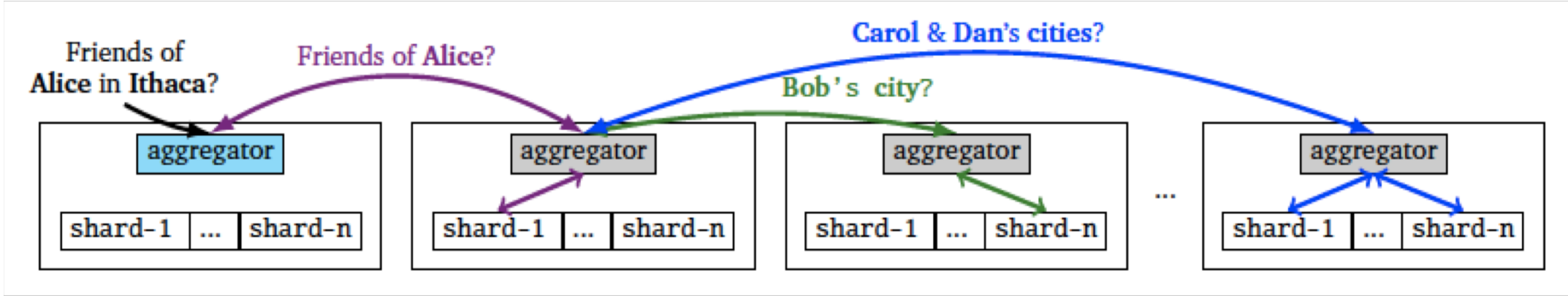
- Challenge : High write rates over compressed graphs
- Minimal interference on ongoing query-execution.
- Memory efficiency – over per-server LogStores
- Avoids complicated data structure for concurrency

Fanned Updates

- Challenge : Fragmentation, new edges added to a node.
- Avoid touching all shards
- Update pointers logically chain data corresponding to same node or edge.
- Pointers - uncompressed



Function Shipping

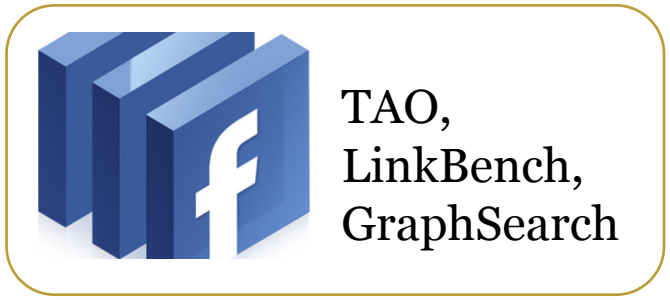


- Computation pushed closer to data via function shipping
- Supports multi-level i.e., a subquery may be further decomposed into sub-subqueries & forwarded to respective servers.

EVALUATION

UNIVERSITY OF WATERLOO

Evaluation



Regular path queries and graph traversals

- Storage Footprint : ZipG is 2x smaller than Neo4j but comparable to Titan - compressed
- TAO :
  - ZipG sees 2.5x increase in throughput proportional to the number of cores – in distributed setting.
- LinkBench :
  - Write-based queries outperform in ZipG. No change in performance in distributed setting.
- Graph Search :
  - Neo4j-Tuned achieves better performance when the uncompressed graph fits entirely in memory- ZipG overheads. No change in performance in distributed setting.
- ZipG outperforms Neo4j for branched or long linear path traversals. Graph traversal – Neo4j outperforms because of aggregation overheads in ZipG.

ZIPG PROS & CONS

UNIVERSITY OF WATERLOO

Advantages	Disadvantages
1. Dedicated LogStore help avoid decompression of data as well as achieve higher memory efficiency.	1. Fault-tolerance is traditional. But not described in detail for LogStore since that is the central system.
2. Data Persistence is achieved by performing all write operations on LogStore and update pointers in sec. store.	2. Overhead if the uncompressed data fits entirely in memory.
3. Concurrency control is also achieved through LogStore. Locks only required for uncompressed update pointers and deletion bitmaps. 4. ZipG is rich enough to implement almost all complex queries	3. Search-based queries like “Find Musicians in Ithaca” are an overhead as ZipG touches all partitions. 4. No guarantees of consistency and transactions

CONCLUSION

UNIVERSITY OF WATERLOO

KEY TAKEAWAY

- ZipG, a memory-efficient graph store
- Executes queries directly on compressed graph representation (no decompression & scans)
- Rich functionality
  - Queries from industrial workloads: Regular path queries & graph traversals
- New log-structured graph storage
  - Efficient read & write queries

DISCUSSION

UNIVERSITY OF WATERLOO

DISCUSSION

- Could the Authors have chosen a better system than Succinct as the underlying layer of ZipG?
- Will it be a good idea to include bitmap representation(from previous paper) for all operations instead of using suffix and inverse-suffix data structures?
- What can the alternative of LogStore be to avoid overhead for data that completely fits in-memory?
- Deletion is mentioned using bitmap but what about their Garbage Collection Policy?
- Do the pointers really not create an overhead, even when the number of edges increase at a high rate?
- Update queries are implemented as deletes followed by an append. With no proper fault tolerance in place, the system might lose some edge properties if the system shuts down.
- Conflict between input data and delimiters chosen for PropertyList might be a problem?

THANK YOU