Uber Graph Benchmark

Statistically representative graph generation and benchmarking

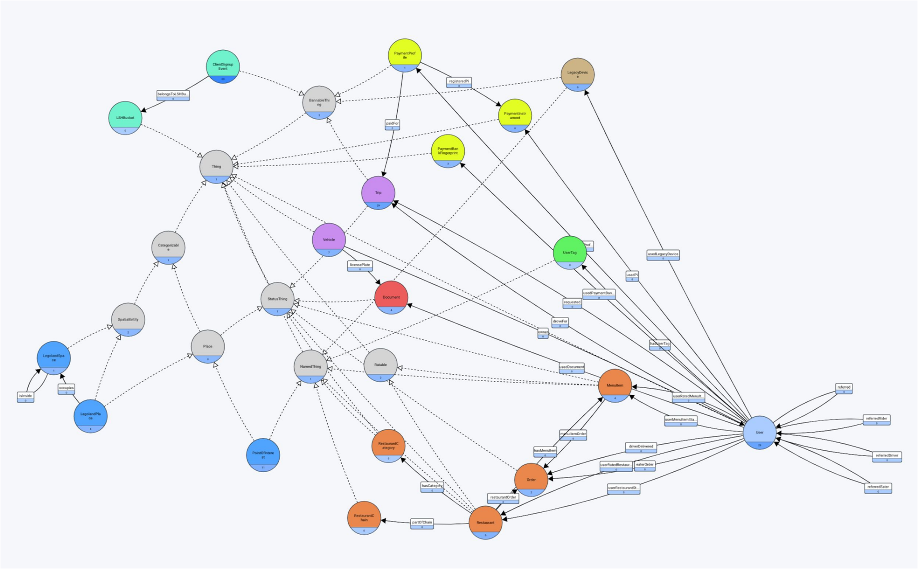
Chris Lu
Joshua Shinavier

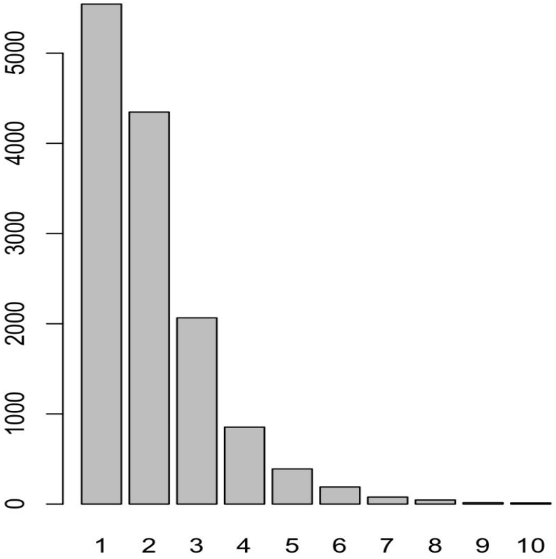
UBER

Agenda

- Why we need a new benchmarking framework
- How it works
- Invitation to participate

Let’s put our data in a graph database!





Just pick one graph database?

Pick one graph database!

- Various graph database vendors.
- It’s possible to build a graph database on key-value stores.
    - Various key-value stores.
    - Graph database on MySQL database
- We already have X key-value store, how does it compare to Y graph database?

It’s complicated to choose one

- What is the read performance?
  - Data size: vertex/edge count, properties
  - Edge distribution
  - Hardware
  - Query pattern:
    - Concurrency
    - # hops
    - Fanout limit
    - Edge filtering
- What is the write performance?
- ...

The answer? It’s complicated.

- What is the read performance?
- What is the write performance?
  - Graph Storage is usually denormalized
  - Denormalize data to optimize for performance
    - Storing adjacency list twice.
    - Build vertex-centric index.
  - Write amplification, transaction for consistency
- ...

More questions

- What is the read performance?
- What is the write performance?
- What is the read performance during heavy writes?
- What is the storage cost? Capacity planning? SKU?

Read the marketing materials

- Count keywords: "scalable", "fast", "in memory", "distributed"
- This is what Artificial Intelligence does.
- Hope your CTO does not do this...

Read other benchmarks, but

- Data is different
  - Schema
  - Edge distribution
  - Data size
- Hardware
  - Network
  - CPU
  - Memory
  - Disk
- Access patterns
  - ...

Which test to trust?

Test with your own data!

How about a DIY benchmark?

```
* learn how to load data into the system

* learn how to query data

* build benchmark system {

        * benchmark loading data

        * benchmark querying data

        * benchmark loading and querying at the same time

}
```

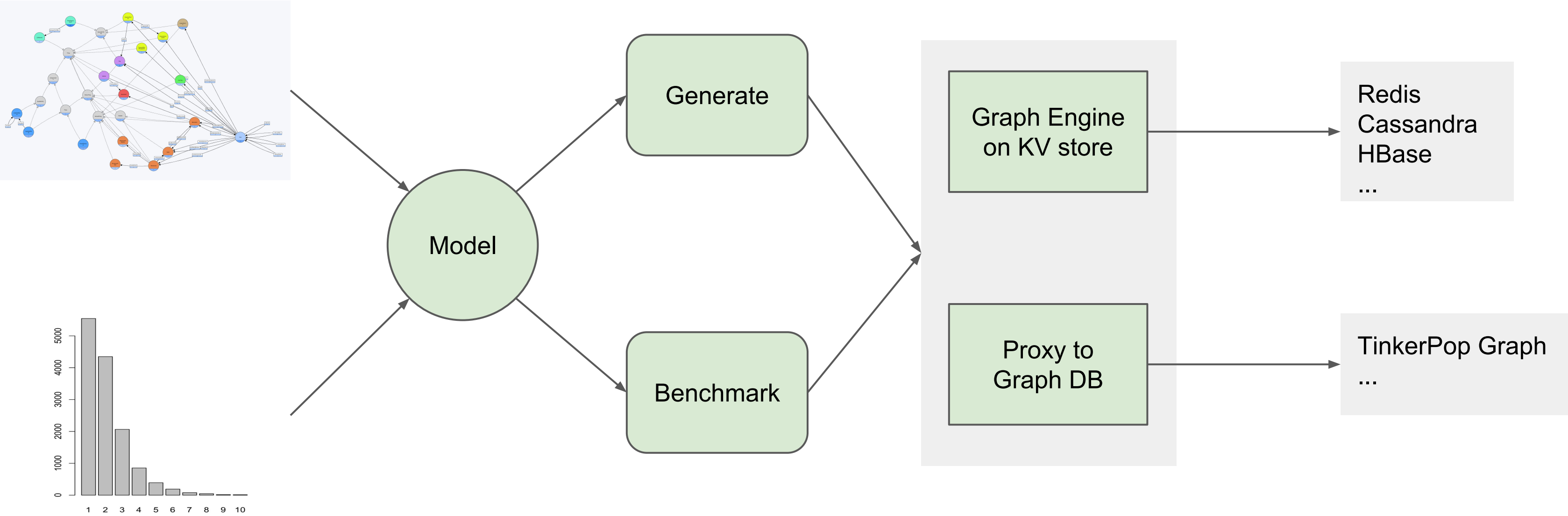
How about a DIY benchmark?

```
for each system {
    * learn how to load data into the system
    * learn how to query data
    * build benchmark system {
        * benchmark loading data
        * benchmark querying data
        * benchmark loading and querying at the same time
    }
}
```

We need a graph benchmark framework to

- Generate a graph representative of your data
  - Any size (as large as you need)
  - Any schema
  - Any distribution
  - Repeatable
- Store data into
  - Any graph store
  - Any key-value store
- Run k-hop subgraph on generated graph, starting from one random vertex

Uber Graph Benchmark Architecture



Model

Generate

Benchmark

Graph Engine on KV store

Proxy to Graph DB

Redis
Cassandra
HBase
...

TinkerPop Graph
...

Graph Gen: Schema-Aware

- Create a statistical model of the dataset capturing:
    - Schema (vertex and edge labels, properties)
    - Label/key probabilities and frequency distributions
- Use the model to generate graphs of any size
- Generated graph should be “close enough” to real data to be used as a substitute

Graph Gen: Define Graph Schema

- Composable from different teams
- Example

```
entities:
- label: User
  relations:
  - label: isDriver
    description: Whether a user is a driver
    to: core.Boolean
  - label: driverStatus
    description: Current status of the driver (Active, Rejected, Waitlisted, etc.)
    to: core.String
```

```
relations:
 - label: requested
   description: The relationship between a user and a trip he or she requested
   extends:
    - core.relatedTo
   from: users.User
   to: Trip
   cardinality: OneToMany
```

Graph Gen: Define Graph Data Distribution

- Vertex

```yaml
vertices:
 - type: users.User
   weight: 13
 - type: trips.Trip
   weight: 8
 - type: documents.Document
   weight: 13
```

Graph Gen: Define Graph Data Distribution

- Vertex

```
vertices:
 - type: users.User
   weight: 13
 - type: trips.Trip
   weight: 8
 - type: documents.Document
   weight: 13
```

- Property

```
properties:
 - type: users.driverStatus
   values:
     - value: "Active"
       weight: 90
     - value: "Rejected"
       weight: 5
     - value: "Waitlisted"
       weight: 5
```

Generate Vertices

- Vertex id generation
  - Customizable for each graph db.
  - Id = F(String vertexLabel, long sequencedNumber)
- Property types
  - Basic: Boolean, Date, Decimal, Double, Float, Long, String
  - Structured: Email, PhoneNumber, UnixTime, Year
  - List of values with weights.
- Vertex properties
  - F(String vertexLabel, long sequencedNumber, String propertyName, PropertyType type)
  - Deterministic property value for filtered traversals.

Graph Gen: Define Graph Data Distribution

- Vertex

```
vertices:
 - type: users.User
   weight: 13
 - type: trips.Trip
   weight: 8
 - type: documents.Document
   weight: 13
```

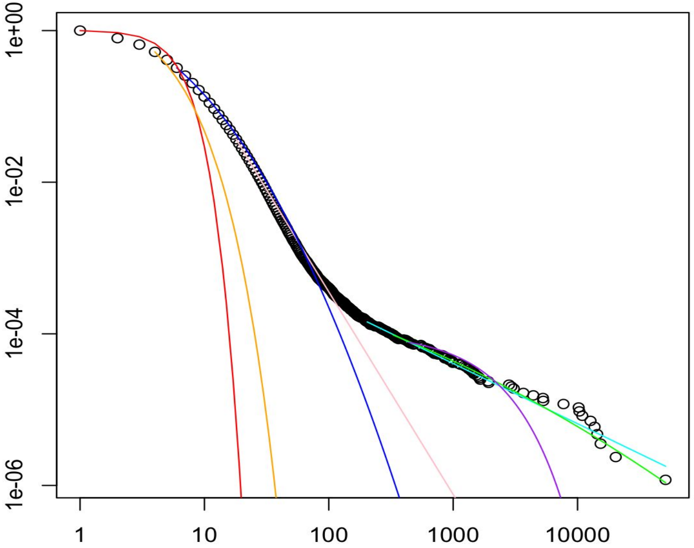
- Property

```
properties:
 - type: users.driverStatus
   values:
     - value: "Active"
       weight: 90
     - value: "Rejected"
       weight: 5
     - value: "Waitlisted"
       weight: 5
```

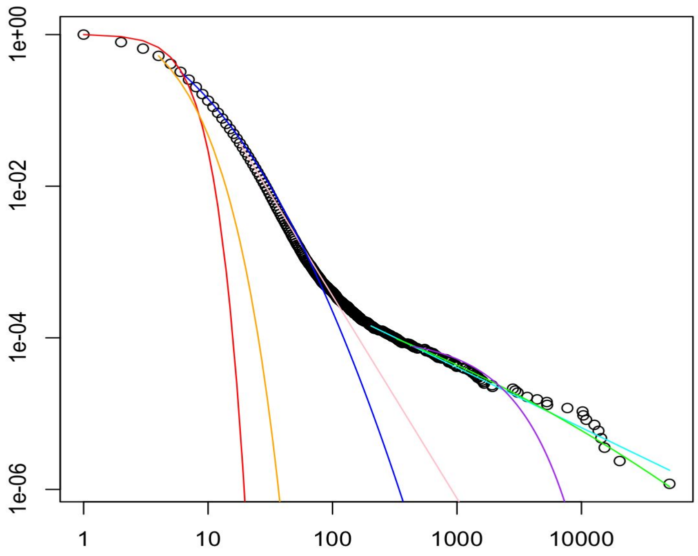
- Edge

```
edges:
 - type: trips.droveFor
   out:
     existenceProbability: 0.1
     logMean: 2.0
     logSD: 0.3
   in:
     existenceProbability: 0.9
```

How to deal with complicated distributions?



How to deal with complicated distributions?



Use edge distribution CSV file

degree	count
1	10000000
2	1000000
3	100000
4	30000
5	10000
6	3000
7	1000
8	600
9	200
...	
1000	1
1300	1
1800	1
20025	1
60123	1

Generate Edges

- Create edges:
  - Pick a source vertex
  - Pick a target vertex
  - Connect them
- Function of (random seed, edge distribution)

Graph Gen: Consistent Partitioned

- Need to support massive graphs.
- Support Spark to partition the generated graph.
- Each partition has its own seed.
    - partition seed = base_seed + partition_index
- Generate a graph of 1 billion vertex and edges in one hour.

Implemented Storages

- Graph DB with Gremlin support
- Key-value store and prefix-key-value store
  - Vertex:
    - Row key = (VertexLabel, VertexId)
    - Value: properties

Edge Storage

Key-value

- AdjacencyList
- Row Key = (EdgeLabel, sourceVertexId)
- Value : List of (targetVertexId, edge properties)

Prefix-key-value

- Multi-rows of edges
- Row key = {(**EdgeLabel, sourceVertexId**), targetVertexId}
- Value: edge properties
- To query a source vertex edges, scan rows with prefix (**EdgeLabel, sourceVertexId**).

Support Anchored Queries

- Starting from a randomly chosen vertex.
- Implemented K-hop query for key value stores.
- Or any anchored path query

```
startVertexLabel: users.User
queryType: gremlin
queryText: >
  g.V(x).outE('trips.requested').subgraph('s').outV()
  .outE('trips.droveFor').subgraph('s').outV()
  .outE('documents.usedDocument').subgraph('s').outV()
  .cap('s').next()
```

Example Output

```
{
  "write.vertex": {
    "Operations": 999998,
    "Average(us)": 513.58802002204,
    "Variance(us)": 1343.5852131123756,
    "Min(us)": 52.967,
    "Max(us)": 157801.328,
    "95thPercentile(us)": 1347,
    "99thPercentile(ms)": 2
  },
  "write.edge": {
    "Operations": 628233,
    "Average(us)": 1066.3619543274551,
    "Variance(us)": 792.03133339263,
    "Min(us)": 129.731,
    "Max(us)": 33993.35,
    "95thPercentile(ms)": 2,
    "99thPercentile(ms)": 4
  },
  "read.vertex": {
    "Operations": 38349,
    "Average(us)": 1734.14784575869,
    "Variance(us)": 2459.3334678627807,
    "Min(us)": 70.626,
    "Max(us)": 58464.943,
    "95thPercentile(ms)": 4,
    "99thPercentile(ms)": 11
  },
  "read.edge": {
    "Operations": 138349,
    "Average(us)": 1573.6572573997644,
    "Variance(us)": 2037.7534999789132,
    "Min(us)": 41.721,
    "Max(us)": 56861.816,
    "95thPercentile(ms)": 4,
    "99thPercentile(ms)": 10
  },
   "subgraph": {
   "Operations": 100000,
   "Average(us)": 12439.45511393,
   "Variance(us)": 5797.011929088904,
   "Min(us)": 668.139,
   "Max(us)": 105834.563,
   "95thPercentile(ms)": 32
  },
  "subgraph.vertex.count": 67236,
  "subgraph.edge.count": 60742,
  "non.empty.subgraph.count": 6494
}
```

Open source

https://github.com/uber/uber-graph-benchmark

Uber Graph Benchmark

- Graph Generation
  - Schema-aware
  - Flexible distribution
  - Deterministic
  - Highly scalable thanks to Spark support
- Plugins for graph storage backend
  - Extensible for different storage systems
- Graph query
  - K-hop subgraph queries

Contributions welcome

- Test your graph database solutions
- Share your data schema, distributions and queries
- Add adapters for key-value stores
- Add adapters for graph databases
- Other:
  - Different edge generation algorithms
  - More vertex property types

Link

https://github.com/uber/uber-graph-benchmark

Email

- Chris Lu <chris.lu@uber.com>
- Joshua Shinavier <joshsh@uber.com>