Graph-Based Identity Resolution at Scale

Chinmay Nerurkar
Michael Berry
Subha Narasimhan
Yana Volkovich
Who is Xandr?
Our Goal: Deliver Unique Value Across the Advertising Ecosystem

Build and scale a premium advertising marketplace across all forms of consumer engagement through the innovative use of technology, data and premium content.

**Consumers**
MAKE advertising more welcomed, relevant, and less interruptive for Consumers

**Buyers**
OPTIMIZE return on investment against desired outcomes utilizing converged and brand-safe solutions

**Sellers**
MAXIMIZE the value of inventory for Publishers and protect the viewer experience
The Value of Xandr’s Technology Platform

Brands

Agencies

Publishers

Consumers

Xandr’s Data Solutions
- Identity
- Audience
- Attribution + Insights

InvestTV

Monetize

CTV/OTT

Digital Video

Display/Native

Data-Driven Linear TV

Addressable Video
Unique Ad Formats: Pause Ads
Identity Graph
What is an identity graph?

An Identity Graph stitches together different identifiers into a unified view of **people**, the **households** they belong to and **devices** they use.

Why is Identity important?

**People use multiple devices and screens daily**

Identity enables cross-device & converged addressable advertising
- **More efficiency**...Household/Consumer Frequency Capping
- **More reach**...Audience Extension to Linked Devices
- **More lift**...Conversion Attribution across Devices

**People expect relevancy and personalization**

**People demand privacy**
*Identity allows consent elections across device & affiliated brands.*

**People will be harder to reach & target in a 3rd party cookieless future**
*Deterministic 1st party ID consortiums of publishers & brands help.*
The Big Picture

1st Party Foundation
Customers, Devices, Cookies

3rd Party Enrichment
Expand to full U.S.

Normalize & Curate
Graph Identity Resolution
Consent Filtering

Identity Graph
Graph Model

A Broad Collection From Multiple Sources

<table>
<thead>
<tr>
<th>Subscription ID</th>
<th>Household ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Aaa-001</td>
</tr>
<tr>
<td>001</td>
<td>Aaa-002</td>
</tr>
<tr>
<td>002</td>
<td>Bbb-001</td>
</tr>
<tr>
<td>003</td>
<td>Ccc-001</td>
</tr>
<tr>
<td>003</td>
<td>Ccc-002</td>
</tr>
<tr>
<td>003</td>
<td>Ccc-003</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Household ID</th>
<th>Device ID</th>
<th>Subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aaa-001</td>
<td>1002001</td>
<td>Cookie</td>
</tr>
<tr>
<td>Aaa-001</td>
<td>1002002</td>
<td>IDFA</td>
</tr>
<tr>
<td>Aaa-001</td>
<td>1002003</td>
<td>AAID</td>
</tr>
<tr>
<td>Aaa-001</td>
<td>1002004</td>
<td>Cookie</td>
</tr>
<tr>
<td>Bbb-001</td>
<td>1003001</td>
<td>Cookie</td>
</tr>
<tr>
<td>Bbb-001</td>
<td>1003002</td>
<td>AAID</td>
</tr>
</tbody>
</table>
Architecture

Where does TigerGraph Fit in?
Graph Identity Resolution
Household Labeling Algorithms

**HashMin algorithm** is a **label propagation** algorithm:
- We assign a unique ID to every vertex in the database.
- Each node's label is updated with minimum of its connected neighbors' labels.
- Repeat until no change in any label.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comms</th>
<th>Memory</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashMin</td>
<td>low</td>
<td>low</td>
<td>slow</td>
</tr>
<tr>
<td>HashToAll</td>
<td>very high</td>
<td>very high</td>
<td>very fast</td>
</tr>
<tr>
<td>HashToMin</td>
<td>medium</td>
<td>high</td>
<td>fast</td>
</tr>
<tr>
<td>HashGreaterToMin</td>
<td>medium</td>
<td>medium</td>
<td>very fast</td>
</tr>
</tbody>
</table>
Label Persistence Algorithms

For households with approximately the same membership in current and prior builds, we want to have the same label, so that we can track it over time.

We used a modified version of Gale Shapley (Stable Marriage Problem):

- Synthetic Household Vertex is assigned a provisional label = unique vertex ID.
- Each run Household may add or lose objects, split or merge with other household(s).
- Match and Rank by the number of shared objects.
- When there is a match, we propagate prior label to current, otherwise the current label is used.

- Overlap counts with each prior household are stored in each current and prior household vertex. This is expensive in terms of memory when there are large connected components.
Large Clusters Splitting Algorithms

**Centrality** is a measure of a vertex’s importance. We use several known centrality measures as well as some modifications of the traditional ones.

**Degree Centrality** is the outdegree of a vertex.
- We exclude vertices with degree centrality higher than a threshold.
- This eliminates bad vertices, e.g. a corporate IP that can connect hundreds of devices.

**Betweenness Centrality** of a vertex is the number of shortest paths that flow through it. Peripheral nodes have low BC score.
- The central node in the figure has a very high BC score.
- Identifying such nodes and removing them (with edges) can split a GCC into its constituent households.
- Very expensive in terms of resources (memory and CPU)
Implementation
Implementing algorithms on TigerGraph

First Thought: Build One Query to do it all!

- Perform all steps - Label Propagation, Creating Synthetic Groupings, Label Persistence - one after another in a single GSQL query
- No need to write results of intermediate steps to disk

But…

- The query becomes long, complex, harder to extend
- Will very likely run out of memory and fail for any significant amounts of data
- Harder to introspect the output of intermediate steps and debug

Better Implementation: Break it down to a series of successive steps

1. Label Propagation using HashMin algorithm
2. Break down unrealistically large groupings
3. Create Synthetic Grouping vertices
4. Connect Synthetic Groupings to other vertices in the group
5. Run Label Persistence using Gale Shapley algorithm

Benefits…

- Each query performs one simple step and is easier to maintain
- Each query commits changes to the graph, simplifying introspection and debugging
- Easier to extend query logic and test changes
- Easier to stay within memory envelope and handle more data
Lambda Processing Architecture

- Data is loaded into the graph using the /ddl RESTPP endpoint at the start of the day
- Automated Job Scheduling system kicks off GSQL queries to perform identity resolution and generate synthetic groupings
- Synthetic Groupings are extracted from TigerGraph to local disk & moved out to consumer’s S3 storage buckets
- The automated process is repeated daily - loading new data and graph processing - to deliver an updated version of persistent Synthetic Groupings
Does it scale?

- Distributed graph with 5+ billion vertices and 7+ billion edges
- Up to 1 billion daily graph updates from input
- 300 million vertices and 1+ billion edges created by the algorithms
- We built a 10 node TigerGraph cluster. Each node has 48 cores, 400GB RAM, 3GBps NVMe storage.
- Running BFS-style algorithms, like Label Persistence, spanning over a large distributed graph is extremely memory intensive
- We can add more RAM to the cluster nodes but vertical scaling has limits. We need to scale horizontally

**Clean the Input Data**
- Filter out and exclude vertices with large outdegrees before applying logic. They cause unrealistically large connected components, are rarely useful from the business sense and cause memory issues.
- Create better pre-processing pipeline for filtering out bad actors before they are added to the graph

**Divide and Conquer – Process part of the graph at a time**
- Shard the graph into N sub-graphs using a scheme appropriate for the use case, then process each part independently to scale horizontally
- For our use case of Label Persistence
  - Run label propagation to split the graph into N different subgraphs of connected components
  - Run Algorithm on each sub-group independently
Optimization Results

• More than 35% reduction in peak memory utilization per node with 3 shards
• Total runtime increased by less than 10%
• Enables horizontal scaling for graph processing
• Increased stability and reliability of the cluster

Next Steps…

• Differential data loading and graph processing
• Testing new data models and more advanced graph algorithms
### Lesson #1 – Exercise ‘Graph Thinking’ to solve graph problems

- Avoid creating your data model that encourages a user to use attributes as indices.
- Avoid framing queries with complex joins and where clauses on ‘index’ fields. This will cost performance.
- Take advantage of index free adjacency and the graph native nature of TigerGraph to get the best bang for your buck.

### Lesson #2 – Solve problems in a MapReduce fashion

- TigerGraph is built ground up for distributed graph processing. Use Distributed queries as much as possible.
- Map your larger problem down into smaller sub-problems that can be solved individually.
- Reduce solutions to these sub-problems to build your result.
Credits

Yana Volkovich
Subha Narasimhan
Abraham Greenstein
Swapnil Pandit
Avinash Katika
Ted Will
Elijah Hall
Aekta Amin
Jian-Ning Wong
Jash Lee

Michael Berry
Chinmay Nerurkar
Sal Rinchiera
Noah Stebbins
Bill Landers
Guru Raghavendra Reddy Batchu
Ron Lissack
Alison Su
Charlie Song
Matthew Drobnak

Thanks to the TigerGraph Team for Support
Thank You