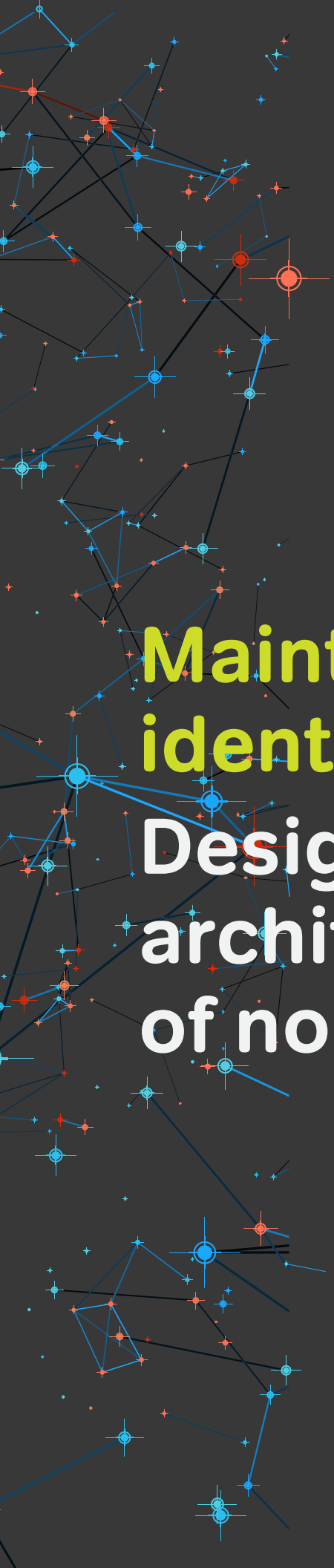


Lineate

Maintaining an identity graph at scale: Designing a database architecture for billions of nodes

Contents

Lineate

Building an Identity

In the ad tech world, the user ID-matching process is key to producing high-quality data for ad requests, reporting, and audience targeting. Big players in the industry receive ID-matching requests from many sources such as publishers, DSPs and SSPs, and other data enrichers. These IDs are used for cookie matching, first-party data, email tracking, SDK tracking, ad requests, browser-level tracking, and data exchanges with partners. Capturing and linking all this data at scale is a significant technical challenge.

Ad tech companies need not only capture user and device data but also make it queryable and actionable for clients to use for reporting or other data-related applications. It is also crucial for ad tech companies to combine all the ID pairs they've collected through ID matching in an identity graph. An identity graph helps advertisers get a full picture of a user's journey through a marketing funnel across multiple browsers, apps, and devices. For example, if a user looks for a product on their desktop computer, the advertiser is able to target that same user on their mobile phone and reach them with an email campaign or in social networks. So this set of user IDs makes up an identity graph that stores the relationship between a user's personal devices, browsers, IP addresses, or household devices.

As a result, ad tech companies and publishers need to store and join many different data sources. When dealing with these data sources, the main challenge is that tracking signals may come from the same user multiple times per minute, which produces terabytes of data daily. The majority of this data is duplicate ID pairs. Companies need to first deduplicate data, to keep only unique connections between IDs, and then join multiple data sources into a queryable data structure. Without deduplication, the data volume is usually so big that it is not possible to run a query against even one month's worth of data. The problem seems straightforward; however, the solution is more complicated than it first appears to be. Over the years, Lineate has tried various approaches before settling on a paradigm that has proven itself for large-scale ad traffic, both on bare metal and on the cloud.

The Technical Challenge

A user-ID tracking system receives only pairs of IDs, not an entire set of IDs at once. For example, when a user logs into a website, the tracking system may receive a cookie-to-email-hash pair, but the general cookie-matching process limits cookie-to-cookie pairs between multiple partners, and so on. The tracking system's task is to build out the user/device graph, starting with a single pair of IDs and creating more complex relationships from there. One of the key requirements for our ideal system was not only to store and query data but also to handle hundreds of thousands of insertions per second, almost 24/7, all while other applications are using the system uninterrupted (doing full-scan data jobs or single key-value or range-key queries). Besides handling such a large volume of insertions and read load, the system must be able to scale horizontally as the business grows.

Our testing approach involved providing clean and prepared-for-insertion data of approximately 500 thousand ID pairs per second. More than 50% of that data was duplicates that needed to be cleaned within a few minutes of insertion. To illustrate the scale of this requirement, it is worth mentioning that this produced more than one terabyte of tracking event data per day. The cookie-based tracker alone produced more than ten thousand files per hour to analyze, and one of our partners supplied about 400 thousand files per hour to deduplicate and insert. Different types of IDs have different life cycles, so some IDs in the database needed to be stored for just a few weeks, but other types of IDs may need to be stored for multiple months.

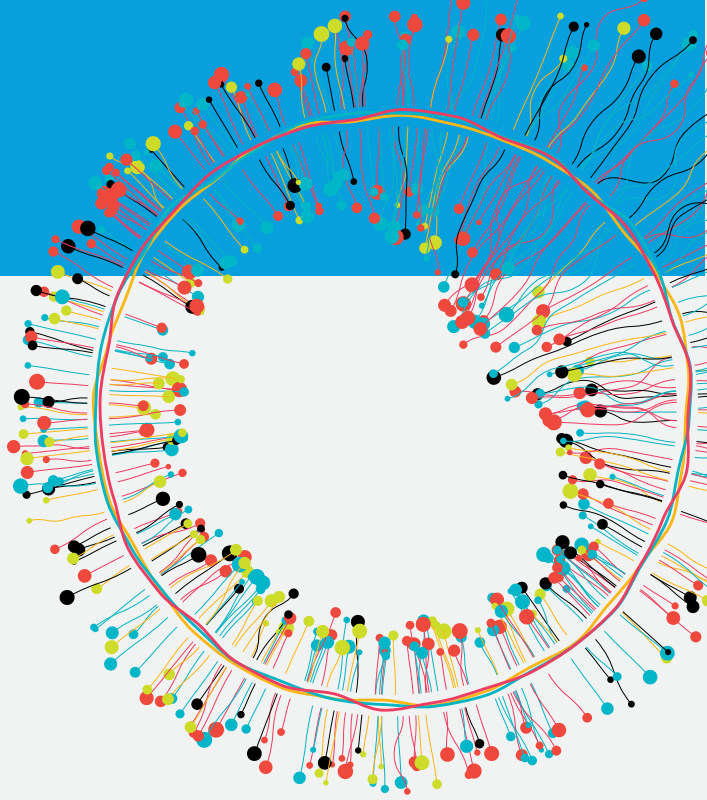
Our Search for a Solution

The graph data model may tempt ad tech companies to try storing user IDs in a graph database. The built-in graph traversal features of a graph database seem like an easy way to store IDs and their relationships, and to quickly do multi-hop queries. For example, such functionality makes reaching an email hash through a browser cookie, or getting a mobile advertisement ID through an email hash, relatively simple to implement. However, we found the convenience of graph databases was outweighed by the challenges of scaling to the data sizes we needed.

At Lineate, we were thrilled to try graph database solutions for our use cases. We considered many graph solutions, such as Neo4j, Amazon Neptune, NebulaGraph, TigerGraph, and others. As with most Lineate projects, we preferred to use a serverless architecture so we could be more cloud-native and spend less effort on database maintenance.

In the end, we chose a combination of technologies and rolled our own graph query capabilities, rather than look for everything in a single graph database. We found that different graph solutions could store large amounts of data and make pretty quick multi-hop ID queries. However, in ad tech there is often a need to implement full table scans, for example, when a partner needs to join a company's user or device data with its own and output a user intersection between the two systems. These wide queries proved problematic with all the graph databases we tried.

The Results



Our experiments with the various database solutions demonstrated that Amazon Neptune can't handle a large insertion load and was far from hitting our required numbers. Other databases also struggled with large loads. We found that commercial solutions such as TigerGraph were able to insert the required data, but under such a substantial load it was impossible for the system to query data with expected performance. We concluded that graph databases are better suited for a smaller volume of data mutations and for queries other than full table scans.

We also tried a few NoSQL solutions, such as ScyllaDB and Hbase, and we found that the final system states that were able to scale and handle the read and write load were HBase-based (AWS EMR) solutions, in which we manually implemented graph-like indexes. Yes, we lost the ability to do multi-hop lookups in arbitrary directions, like from hash to cookie or back, but our indexes stored a predefined graph path. So it is possible to query device IDs through a cookie or email hashes. In addition, HBase scaled well, could handle the write load, and allowed us to run full-scan Spark queries to join the data. HBase also enabled us to avoid manual deduplication because it has unique keys by design. Aging data eviction was also achieved with the HBase TTL feature.

Alternative solutions that also worked well for us were parquet files on AWS S3 and multi-layer deduplication; for example, we first deduplicated minute data, then hour data. Then from the hourly data we constructed a unique set of IDs for each day, then for each week, and so forth.

Conclusion

Like so much in ad tech, there is no single silver bullet for maintaining an identify graph, but it can be done cost effectively using the right technologies for each function.

Dedicated graph databases sounded like an attractive option to build identity graphs because of the natural way they can build and search relationships across data. The challenge was to achieve this in the face of an enormous amount of incoming data every second. We had hoped we would be able to find a way to overcome this challenge, using other technologies to build and maintain the data graph, while using graph database capabilities to query it. In the end, however, we found the constant need for full table scans and ranges made this approach unsuccessful.

We did achieve great success using HBASE to build the graph, leveraging its built-in TTL and deduplication features. This solved building the graph under load, while we loaded the data into Spark to allow high-speed querying over large datasets. The combination of HBASE and Spark performed well. We also have had success pre-aggregating time series data using Parquet on AWS S3.

The same approach can be built with other similar technologies, but this is now our preferred stack. It fits into our standard Luna architecture, and has proven successful in real-world usage.

Thank you.

Can we help you with your ambitious goals?

Talk to us today at lineate.com/contact

Lineate