Amplitude has grown significantly both as a product and in data volume since our last architecture revamp, Wave. When we realized that our Wave Architecture would not be effective long-term at powering the types of queries our customers want to run, we started planning the next iteration.

We had two main goals for the new system:

1. The ability to perform complex behavioral analyses (e.g. to power features like Compass and Pathfinder)

2. Cost-effective scalability

After extensive research, we decided to build an in-house column store that is designed specifically for behavioral analytics. We call the resulting system Nova, and it has allowed us to serve queries that often scan tens of billions of events, with 95% of queries taking less than 3 seconds. We’re excited to share the thought process around how we got here and some of the key design decisions we made.

This white paper covers:

- Limitations of pre-aggregation in our previous architecture
- Why we decided to build an in-house column store
- Nova’s user-centric computational model
- The storage engine underlying our system
Limitations of Pre-Aggregation

One of the core principles of our existing architecture was the concept of pre-aggregation. Pre-aggregation is based on breaking high-dimensional queries down into simple, low-dimensional components, and then aggregating the components prior to the query. Then when the query is actually run, significantly less computation needs to be done, which makes pre-aggregated queries very fast. This is effective for a lot of different queries in behavioral analytics, ranging from segmentation to simpler forms of retention. The trade-off, of course, is that you have limited flexibility in the types of queries that can be run efficiently. If a query cannot leverage pre-aggregated results, then it is forced to do a full computation that is very expensive. Some of our new features like Compass and Pathfinder are significantly more complex than previous ones and cannot take full advantage of pre-aggregation. The upside of pre-aggregation is cost and speed, but the breadth of queries led to a fragmented and cost-inefficient system that was slow for the more complex queries.

It became clear that we needed an overhaul of the query architecture that puts flexibility as one of the top priorities, thus enabling features that give our customers a deeper understanding of user behavior. For anyone working on analytics databases today, the natural choice is a column-oriented database (column store).
Overview of column stores

The idea of storing data in a column-oriented format in databases has been around for a long time (30+ years). This article provides a great summary of the history of column stores and the techniques that make them so effective. In contrast to a traditional relational database, which is optimized for reading, updating, and inserting single records, a column store is optimized for answering queries that scan a large dataset, which is the exact use case of analytics. In a column store, the data is structured in a way that allows for high performance despite the fact that queries have to scan considerable amounts of data to produce results. The main advantage of a column store over any sort of pre-aggregated system is that it offers great flexibility for queries, since all of the raw data is available.

There are quite a number of column store technologies out there today (many of which happen have blue logos), and it’s useful to understand the differences between them. They can be broadly placed into one of three categories: services, commercial, and open-source.

In the services bucket, we find the offerings of two of the big infrastructure players, namely Amazon Redshift and Google BigQuery. These are by far the simplest solutions to get running, as they handle the management of the underlying machines for you. Both have SQL-like interfaces and can be used by anyone who has SQL experience. We are heavy users of Redshift here at Amplitude, since it allows our customers to build reports using SQL. The main commercial offering is HP Vertica, which is the evolution of one of the first column stores, C-Store. It is one of the older products out there, and has a strong enterprise foothold in the space.

Lastly, there are a couple of recent open-source column stores that we looked into. The first is Citus Data, which is an extension that turns PostgreSQL into a distributed database that can store data in a columnar format. This is cool because PostgreSQL is a widely-used and mature database (we run a few instances of it ourselves) that has many integrations already. Druid, on the other hand, is an analytics data store built from the ground up for high performance and has its own query API that supports many different types of aggregations.

For the vast majority of use cases, choosing one of these is the right decision, as they have done all the hard work already of building a highly-scalable system for analytical queries.
Advantages of an in-house column store

Given all of the existing work that has been done around analytical column stores, you would need compelling reasons to build a new one. We spent quite a bit of time understanding the intricacies of our use case and their implications on our technology: the end result was a handful of key advantages to an in-house column store:

1. **Query flexibility is one of the most important aspects of our database system** (it’s one of the motivating factors for the architecture revamp in the first place!). We’re constantly innovating on the analytics we provide, so it’s important that we can handle everything from simple aggregations to machine learning algorithms.

2. **Our event data is immutable, and we can use that assumption to improve performance.** There are a few special instances of mutability that we’ll go into later, but not having to support generic updates removes a lot of the complexity.

3. **An area where we have some leeway is eventual consistency.** When we ingest events, it’s not essential that the data be made immediately available after accepting the write, only that it is queryable after a short period of time (typically a minute or so). In the same vein as immutability, dropping strong consistency allows for optimizations.

4. **Finally, we have more control over where and how the data is stored, which reduces costs.** While all column-oriented databases are designed to be horizontally scalable, it’s necessary for us to operate cost-efficiently at high volumes since sampling is bad.

Furthermore, we have worked a lot with Amazon Redshift and have also explored Google BigQuery in the past. Ultimately, we always found use cases that were not supported, especially given the unique behavioral analyses we do, so having control over the underlying system has a certain appeal.

There are significant downsides as well that we weighed against the advantages, namely the engineering effort required as well as the fact that we might have to reinvent a lot of the basic reliability and scaling mechanisms. Fortunately, we have some experience mitigating the latter by leveraging the AWS ecosystem, so it mostly came down to whether the time and resources spent developing our own system would pay off. Given that analytics and distributed systems are our core competencies, developing a competitive advantage in this space is a worthwhile investment. Thus we embarked on a multi-month journey to build Nova, an in-house column store that would scale with the company.
We think of Nova as a distributed columnar storage engine with a real-time MapReduce computational model centered around the concept of users. We’ll first describe the model before getting into the details of how data is stored. As a reminder, one of our main goals was query flexibility, and MapReduce is a great fit as a generic framework for expressing many types of computations.

One of the fundamental principles that we follow is moving code to data rather than the other way around. Behavioral analytics is highly dependent on looking at all the events for a user at once. In a distributed system, this amounts to performing a distributed join on user, which means that data needs to be partitioned by user to avoid moving it, hence the user-centric model. Once you’ve done this, a query can be highly parallelized by running independent workers on disjoint subsets of users. In our computational model, the reduce step assumes that it has the full view of a user’s events.

For example, suppose you are computing the conversion for a funnel in an e-commerce application with steps Register ➔ View Item ➔ Add Item ➔ Purchase. Given all of the events that a user has performed, it’s easy to determine how far they made it in the funnel. Now, assume that you compute the funnel conversion independently for users in disjoint sets A, B, and C that make up the complete user base. The important property of behavioral queries that we leverage is that the results of the query on subsets of users are small (not dependent on the number of users) and can be aggregated together. In this case, we see that the total funnel conversion is simply obtained by summing the conversion at each step across A, B, and C.
If you’re following closely, you may have noticed that, if the data is partitioned by user, can’t the shuffle step be omitted, thus collapsing map and reduce? There is one major caveat to this, which is the problem of merged users or aliasing. It is possible that, at any point in the future, we may gain information that two users within Amplitude are actually the same. We choose not to retroactively modify the event data since that violates immutability, so we instead apply a query-time mapping that tells us the true user for each event. This means that at the data layer, we do not have a fully-partitioned dataset, necessitating the shuffle; fortunately, since merging and aliasing only account for a small percentage of users, we incur minimal cost for the step that is often the bottleneck of a MapReduce.

Finally, after the reduce computes the query result on subsets of users, there is an aggregation layer that follows. While these results are small relative to the event data, they can still be non-trivial in size (e.g. when doing a large cardinality grouping), so we use hierarchical aggregation to eventually produce a single result to return to the client.

MapReduce is a popular model of computation because it is both simple and flexible; nearly any type of analytical query can be represented as a MapReduce. In combination with the user-centric focus, we are able to efficiently analyze behavior across huge user bases. To reiterate the first advantage of an in-house column store, we are able to express behavioral queries such as Compass and Pathfinder within this model (albeit through writing code) that are impractical to do in SQL.

The underlying storage engine

While there are a lot of changes from our existing system, one aspect that we decided to keep is the use of lambda architecture. To recap, lambda architecture is a design pattern that separates real-time ingestion from batch-computed data in order to localize the complexity and cost of real-time updates. Instead of these layers producing pre-aggregated results, they now convert the raw data into an optimized columnar format, but the benefits are the same. One helpful abstraction is separating the source of the data (real-time vs batch layer) from the query logic, so that all queries look like they operate on a single data format.

<table>
<thead>
<tr>
<th>USERS</th>
<th># REGISTER</th>
<th># VIEW ITEMS</th>
<th># ADD ITEM</th>
<th># PURCHASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5,000</td>
<td>2,000</td>
<td>300</td>
<td>200</td>
</tr>
<tr>
<td>B</td>
<td>7,000</td>
<td>4,000</td>
<td>200</td>
<td>150</td>
</tr>
<tr>
<td>C</td>
<td>4,000</td>
<td>2,500</td>
<td>150</td>
<td>100</td>
</tr>
<tr>
<td>TOTAL</td>
<td>16,000</td>
<td>8,500</td>
<td>650</td>
<td>450</td>
</tr>
</tbody>
</table>

In our model, funnel conversion is first computed independently for 3 disjoint sets of users: A, B, and C. The total funnel conversion can then be computed by simply summing the conversion at each step.
Another similarity to our previous architecture is the heavy usage of AWS services, especially S3. We’ve written about the advantages of S3 in the past, but they’re worth mentioning again because of how much they simplify building systems. The key features of S3 that make it invaluable are its durability and cost; by using S3 as the canonical data store, we ensure that data is never lost even if we don’t implement any durability mechanisms ourself. Once you have that guarantee, everything else in the system can act as a cache on top of S3 and doesn’t have to worry about replication. We heavily leverage this insight to build services that are simple and stateless yet have high durability guarantees (and reasonable costs!).

Given all of this, the final piece is what the storage layer of Nova actually looks like and the optimizations we have made to get the best performance. Data is broken out by users, time, and ingestion batch in order to have immutable chunks that can be managed easily; every day we produce a new set of chunks that are saved to S3. Given that the system functions as a cache on top of S3, you would typically expect to run into complex cache invalidation problems, but here immutability saves us. Since the chunks can never change, they do not get invalidated, and we only need to observe when new data is available. Once again, the caveat to this is information about merged users, but we built a separate mechanism for that as the scale is significantly smaller (the size of the merged user information is less than 1% of the raw data).

Within a single immutable chunk of data, we apply many of the standard columnar optimization techniques to get high throughput on queries. This includes column-specific encoding and compression, such as delta encoding timestamps and dictionary encoding just about everything else, from User IDs to device types. Dictionary encoding turns out to be one of the most
powerful forms of compression not only from a storage standpoint but also for computing “group by” queries since you have already bucketed the events by value; this can turn a hashmap lookup into an array lookup, which is a huge difference for CPU-bound computations. The columnar data format is typical of any such database, and we apply the generic LZ4 compression algorithm on top of individual columns when the data is large enough.

<table>
<thead>
<tr>
<th>METADATA HEADER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELTA-ENCODED TIMESTAMP COLUMN</td>
</tr>
<tr>
<td>DICTIONARY-ENCODED USER ID COLUMN</td>
</tr>
<tr>
<td>DICTIONARY-ENCODED PROPERTY COLUMN</td>
</tr>
</tbody>
</table>

Building on existing work

One of the main challenges with building an in-house column store is developing it in a way that has both the reliability and performance of other solutions. We are fortunate to build upon the services of AWS and the work of many before us who have figured out best practices. There are a handful of systems that we drew inspiration from during the design and development of Nova:

- **PowerDrill** is an analytical column store built by Google that uses a highly-optimized storage format to process billions of rows interactively;
- **Facebook** built a performance monitoring system called SCUBA that has a hierarchical aggregation model on top of an in-memory storage layer;
- **Trill** is a columnar database designed by Microsoft to allow flexible queries that still compile to performant data processing code; and
- **Metamarkets’ Druid**, the open-source column store we described earlier, uses lambda architecture to ensure long-term data integrity.

While we did go down the path of building our own system, we were able to leverage and combine several of the ideas behind these technologies. The resources provided were invaluable to us when making difficult design decisions (of which there were many!).
The Future of Scalable Analytics

Analytics is all about the performance versus cost trade-off, and Nova is designed from the ground up to achieve the best of both worlds. Today, we process over 70 billion events per month and regularly serve queries that scan 10-20 billion events; overall, 95% of queries take less than 3 seconds, which allows for powerful interactive analysis. In the future we will be processing and querying orders of magnitude more than we do now, and we expect Nova to take us there. Many systems exist that offer the capability of handling data at large volumes by throwing money at the problem and adding more machines, but for us what matters is being able to do this cost-efficiently so that our customers can get the most value.

 Scalability aside, a column store is a very flexible foundation on which to build new features, and owning the architecture empowers us to develop advanced functionality that a SQL-based system can never achieve. We are excited to use the technology we have developed to push the boundary of behavioral analytics.

Interested in learning how Nova can power your analytics and help you get deeper insights, faster?

GET IN TOUCH