EPISODE 1137

[INTRODUCTION]

[00:00:00] JM: Twitter is a social media platform with billions of objects; people, tweets, words, events and other entities. The high-volume of information that gets created on Twitter every day leads to a complex engineering problem for the developers building the Twitter search index.

Nico Tonozzi is an engineer at Twitter and he joins the show to talk through the problem space of search at Twitter as well as some recent challenges that he had to tackle in the continuously changing Twitter product.

I want to mention that we're looking for writers and podcasters for Software Engineering Daily. You can send an email to erica@softwareengineeringdaily.com if you want to learn more.

[INTERVIEW]

[00:00:46] JM: Nico, welcome to the show.

[00:00:49] NT: Thanks, Jeff.

[00:00:50] JM: You work at Twitter, and you work on search. What makes search at Twitter unique?

[00:00:57] NT: I'd say the biggest thing for us is being able to do everything in real-time. It's been a problem that every day the retrieval system has to grapple with like what is the indexing latency, how quickly do you need results? How quickly do you need updates? And for us, it's been a concern for a really long time since we started doing search at Twitter, which is like 2008, 2009. So we've been dealing with this problem for a long time. There're a lot of solutions we've come up around it and a lot of teams have come to depend on that.

[00:01:28] JM: And that makes it such that I'm sure off-the-shelf solutions like Elasticsearch are not really going to satisfy what you need.

[00:01:36] NT: Yeah. For the most part, a lot of the search engines out there won't scale quite to the needs that we have. They won't be quite as performant for the issues we've had. There are solutions out there that we have been investigating and we've been pushing forward. I think today, the Vespa search engine and the Elasticsearch engine can comply with a lot of our requirements. We're seeing what things we can push to those systems. But they can't handle a lot of the indexing latency requirements and throughput requirements that we have today at Twitter.

[00:02:10] JM: The key metric that you have in the Twitter search engine, at least one of the key metrics is indexing latency. Can you explain what indexing latency is?

[00:02:21] NT: Yeah. By indexing latency, we mean the time between somebody hits post on their tweet or send on their tweet and the time it's able to show up in the search engine. And that's important metric for us, because we drive a lot of the home timeline traffic. So whenever somebody pulls up in their Twitter app, they'll see the home timeline. And if it took two minutes for a tweet to show up in their home timeline, then that would be totally break that expectations. They'd be confused. They say, "I know my friend tweeted a minute ago, or 15 seconds ago, or I expected my own tweet to show up in my timeline." So it's important for us to drive down that indexing latency metric.

[00:03:02] JM: And why is it so important to be real-time?

[00:03:07] NT: That's one of Twitter's kind of driving focuses. We really believe that the future of social media and news is being instant, being able to pull open Twitter and say, "Hey, this is what's going on in your world now today. This is what the experts think of it. This is what your friends think of it." And without real-time, you can't really have a conversation. It kind of turns into email, it kind of turns into a blog post. And while those are nice and they have the benefits of like being long-form, you don't truly have a conversation. You don't have the interactivity.

[00:03:48] JM: So what was the naïve solution to making Twitter real-time, and why did you have to improve upon that naïve solution?

[00:03:57] NT: Well, originally, we started out with like a MySQL-based index. And that was even before my time. And that kind of worked well for a time, but then back in like 2011, it turned out that it wouldn't scale to the new tweet volumes that we were seeing. And so we expanded it to the Lucene-based solution. And Lucene at the time didn't have any capability for a real-time search or new real-time search, and they didn't have any capability for early termination. And early termination is the method that we use to end a search request early. We say, "Hey, if we get the top 10 hits, then we've searched enough and we don't need to search further into the index."

So at the time, we didn't have any of those capabilities. So we built them ourselves kind of as a backing store for Lucene. And so these backing store for Lucene, it worked well, it was pretty effective and we got it to the point where we only had 15 seconds of indexing latency. That worked really well for like 5 to 7 years. But about a year ago, we decided that we wanted to drive indexing latency lower so that we could serve more of the home timeline and so that we could serve profile timelines from the search index.

[00:05:17] JM: So is the system the use today still based on Lucene?

[00:05:20] NT: Yeah. We rely on Lucene to execute the query. And Lucene is kind of the API against which we code. But the underlying data structures are not based on Lucene. They're these skipless-based data structures, these in-memory data structures that satisfy our performance requirements and are fast enough and scalable enough to meet Twitter's needs.

[00:05:45] JM: How do you alter the underlying data structures of a prepackaged piece of software like Lucene? You have to going into the open source code and modify the data structures?

[00:05:58] NT: No, luckily. That could be a route we could go down, but it would lead to – It'd be really hard, right? Because you would end up forking the code and then we wouldn't get the benefits of the open source Lucene. And every few months, they release an upgrade and they usually package a lot of great improvements in those upgrades, a lot of new features, and we always want to get those. We always try to stay on the latest staple Lucene version.

But Lucene is really well-designed and that they have this extensible codex, these extensible readers, extensible searchers. So you can say, “Hey, when I grab an index searcher from my segment, I actually want to return like my special custom Twitter index searcher.” And that index searcher can, in turn, lookup using any term dictionary at once. It can use any index implementation at once. But then that still lets us use the Lucene implementation of say like a phrase query or a conjunction query. So I’d say we get the both of worlds that way.

[00:07:04] JM: Tell me about the data structures that you needed to implement and what those data structures look like.

[00:07:09] NT: So the most fundamental data structure in a search engine is a posting list. And a posting list is just a list of document IDs. And so as a document in the search world, we call whatever you’re indexing a document. They call it a row in the SQL world or something. But as it comes in, you tokenize it into words and the split up each word. So if you say the white cat, you split each of those into words; and we call those our terms. And each of those terms goes into a posting list.

And so if the white cat had the document ID of 32, then we would add 32 to the posting list for the cat, white. And so then it's really efficient for a searcher to go and look up and say, “Hey, find me all the documents that have the term cat in them.” And so you can just scan through the list and return all those documents quickly.

The other components that you need to implement to do that are like the term dictionary. So that means being able to, given a word, find the posting list that it's associated with. And for that, you can use like a finite state transducer. Almost an automaton complex data structure, but at its base, it's really just a lookup from a term to an ID. And then using that ID, you can find the posting list, or you can use something as simple as like a hash map from a word to a pointer.

[00:08:40] JM: So when I tweet on Twitter, what is happening on the underlying search engine?

[00:08:47] NT: There're a few different steps. Of course, when you tweet, it will go to Twitter’s API endpoints, and it'll be validated by the tweet service. And if the tweet service decides, “Hey, this is a tweet that is valid and we want to create it,” that's kind of what decides what happens.

Once that's complete, they'll write it to a queue. I think we call it the Tweet events queue. And then a bunch of different services read from that queue and say, "Hey, some services might record metrics and they might say, "This is how many tweets we created today." But as far as the search engine goes, we read that tweet from the tweet events queue and ingest it. And the first thing you do with the search engine is you tokenize the terms. And so tokenizing means splitting up the text of a document into the various parts that you might want to index on, you might want to look up by later.

So for a lot of things, that might mean getting rid of capitalization. Like a lot of the time, lower case v is equivalent to capitalized V, and you'd want to be able to search for both. You might want to get rid of plurals. And one of the things you have to do here is you have to be really considerate of the different languages. Like if you're tweeting in English, it's pretty simple to split on spaces and to use spaces as your word separator. For a lot of Chinese, Korean and Japanese languages, or for the CJK languages, you don't have that luxury. They don't always have spaces, or there's many different ways that you could possibly break up a given sequence of characters. So you have to analyze it more carefully and invest a little more in libraries that can do that. And there're some great open-source libraries out there now that do that.

But once you've split it up into terms, you also want to enrich the tweet. You want to say, "Hey, maybe there are different data sources that we can pull from. Maybe we can figure out is this author like a reputable person? Is this tweet likely to get a lot of engagement on it?" And you can store all of that into the search index. So it's easy to look up later, easy to retrieve, and you can score it and determine how popular that tweet could be.

[00:11:05] JM: Do you want to open up the search faculties of Twitter to the tendrils of Google? Or just Google completely do their own search indexing off of the tweet fire hose?

[00:11:21] NT: Google doesn't hook directly into the Twitter API for search results, I'm pretty sure. I think they index Twitter the same way they'd index any other webpage. They can index the text of the page and they can say, "Oh! These are the things on the – This is the text on a page. This is the user. They can analyze that the same way they analyze any other page" and put it into their search index. So that's the way that they analyze it. And that's one way we consider ourselves differentiated from Google, because it will take Google days to crawl a

particular tweet. And they don't have insight into what is growing at a wild rate. Often times, people will tweet and they are people who nobody has ever heard of. They have 10 followers. And then that tweet goes viral. Maybe it's particularly hilarious or particularly apt for the moment. And so that tweet will absolutely blow up.

And because Google doesn't really have insight into what is trending at that time, they don't really have insight into why a particular tweet might be popular or they might not have even crawled that tweet ever. So we can really do that quickly. We can do it accurately based on the information we have inside our systems.

[00:12:36] JM: Of all that information you described about indexing a particular tweet, how much of that is async and how much of it is you have to do it all before the search engine can even index it at all? Can you sort of do a naïve indexing of the tweet and then kind of do all the crazy stuff and then re-index it again? Or what do you do there?

[00:12:57] NT: We try to like get as much as we can in the initial indexing event, but there're a lot of information we don't have at that point. So when like we get the tweet from the tweet service, obviously we know the text of the tweet. We know the user of the tweet. But there's a lot of different systems at Twitter that we want to pull information from that haven't evaluated that information yet. And then there are things like how many times has this tweet been like? How many times has this tweet been replied to?" That of course is changing over time, and we won't have access to that for a long amount of time.

We try to index as much as we can, but then – And so that'll be like the core data structures. That will be the text. And then over time, we get to – We will periodically check with other services and see if information is available for a tweet. And some of that information is like we have a URL crawler service that will patch whether a given article shared is a news article, or if it's like a social media article. And we'll index that. It'll index the URL, and often that'll take like 10 or 15 seconds to be available.

We have a couple different things to try to evaluate different metrics of the tweet, like, "Oh! Is this tweet talking about a particular sport?" If it's talking about a particular sport, then we can tag it with the sport and try to return it inquiries for that sport. And that's another thing that the

machine learning models take a little while to run. It's run by a separate team. And so we can't depend on it being in the core event. So it will be available a little later.

And then, of course, people can hit the heart button on a tweet, and we call that a favorite. That can be up to – It could be years later. And we always try to record that and track that, and so people can – And we try to index, "Has this tweet been liked by this person? How many times has this tweet been liked?" Because that's really useful when you're trying to find – It's really common to say, "Hey, I want to find all the tweets that are the most popular. I want to find the top 10 most faves tweets in the last week."

[00:15:00] JM: Right. So there you're mentioning something that is essentially a ranking function. You have to rank the tweets in order of relevancy. And I'd love to know what other signals go into ranking, because obviously search is not just a matter of looking at what the actual terms are, but you have to do ranking on the return results. So what are the other signals that come into a ranking function?

[00:15:24] NT: Yeah. There're a few different things. The most basic signals we can get are, obviously, like does the text match the query, and that we get from Lucene? And how closely does it match? And for that, we use what we call like a TFIDF-like ranking. So if the document is really short and it matches all of the – Matches the entire query, then that would be a really high-ranking text score. Maybe if it's a really common word, like the query has the term the and the document also is the term the. Then that would not vary much to the score. So just because a query and a document both have the term the, you wouldn't necessarily have a high-impact on the score. Whereas if you had the terms like the zoo, you're searching for the phase the zoo. And a query had zoo, but not the, you would want to rank the document with the word zoo more highly.

But in terms of other signals, obviously, like we take like the user engagement signals are very impactful in terms of scoring. So if a tweet has a bunch of likes, or a bunch of re-tweets of like how toxic a user could be. And so if a user is banned or if they're on suspension, then we would probably rank their tweets a little lower in the search results, and a couple different things like that.

[00:16:52] JM: The amount of work that goes into indexing a tweet, I think we've illustrated at this point. And you wrote this article or co-wrote this article about reducing search indexing latency. Tell me a little bit more about how latency was penalizing the success of search.

[00:17:10] NT: We kind of think of search as to – Search is a loaded term at Twitter. Search means both like the user-facing search where you type into the search bar and you get your search results page. You can see the trending on there, that kind of stuff. But inside Twitter, we also have like the search infrastructure team. And the search infrastructure team provides the backing store for tech search. And it also provides the backing store for like a lot of the home timeline, the backing store for trends. And so a lot of different teams will – Or our customers of the search infrastructure teams, and those teams can call out to the search infrastructure using like our flexible query language. And it was really for those use cases that we most strongly needed the indexing latency improvements.

For the search results page, it's common. Most of the time, like the top tweet will have existed for at least a few minutes, because those are the tweets that tend to have the most favorites, or the most likes, the most engagement. So a tweet wouldn't instantly be the most popular tweet. But for other use cases, like if we want to create a user's timeline based on the search index, it's required that that happens instantly in the perception of the user, so within a second or so. And that's also important if we want to use search to drive the user's profile timeline. Like if you clicked on a user and you'd just seen their tweet in your timeline, you would expect it to show up on their profile page. And so that was a requirement for other teams who are wanting to adapt search for those use cases.

[00:18:50] JM: Tell me more about how changing the underlying data structures has improved latency.

[00:18:57] NT: So, yeah. As I was saying, like we had those data structures that worked pretty well for a long time, but they required us to buffer the tweets and store them for a long time and wait until all of the results were available and to sort them. And so we were able to swap it out for this more flexible data structure where posting lists are internally changeable. You can insert into the middle of them and you still get all of the – Or like early termination properties that we had of her previous posting lists.

And so at this point now we're able to basically ingest tweets as quickly as they come. There're a few milliseconds or a few hundred milliseconds of lag from the service that creates the tweets and writes it to a persistent storage. And then there're a few hundred milliseconds of lag from writing it to Kafka. But we were able to drive it down from the very perceptible 15 or 22nd indexing latency to the one-second of latency, which is nearly imperceptible for the use cases that we care about. We call it near real-time search. And that's been sufficient to onboard a lot more of the home timeline to search. Give them those flexible ranking signals and flexible query language to build the home timeline. And we're working now to onboard more use cases.

[00:20:22] JM: All of this search infrastructure is built in Java, right?

[00:20:25] NT: Yeah, that's right.

[00:20:27] JM: Why is Java a good platform for writing search indexes?

[00:20:31] NT: That's a good question, because Java isn't typically the language that you think of for really high-performing services, right? I often think, "Hey, Java –" They create a lot of garbage, and like the JVM is pretty heavyweight. But it turns out that Java kind of flies when everything gets optimized. When the JIT recorder, or the JIT profiler, the just-in-time optimizers starts going. So the hot loops that you have, like the most core part of your search engine really quickly gets turned into a really optimized code.

The JVM is great at optimizing the code to like reduce the number of allocations. And it gets within like a factor of two or even less of C, or optimized C++, or Rust or something like that. But it has a lot of benefits that those languages don't have. Specifically, at Twitter, there is a ton of JVM infrastructure around tooling, debugging, profiling. There's a really mature ecosystem of like RPC libraries. Like Finagle, for example, is open source. And so everybody at Twitter uses Finagle. They all use thrift. And so any time you want to say, "Hey, I want to see what this team's service does. Or I want to see how this works. I want to call this team." There is no trying to figure out HTTP endpoints and trying to figure out the exact way that it works. You just instantiate your thrift client and it all works instantly. And you get type checking for free. You get deserialization. You get observability. So Twitter has like – And Finagle even has this built in. It

automatically has tracing. So you get a huge number of benefits from being in that JVM ecosystem, especially if your company is bought into that.

[00:22:29] JM: Coming back to the data structures, the movement towards unrolled link lists, or I guess away from unrolled link lists into skip lists. Could you articulate why that was important?

[00:22:45] NT: Yeah. The unrolled link list we had – Unrolled link lists are kind of a fun data structures, because you get some of the benefits of link list, but also some of the benefits of a vector. You never need to reallocate the link list. You just allocate on to the end. But the way that we implemented them and the way that's easiest to do if you're having concurrency – Yeah, let me step back and talk a little bit about concurrency. It's really important for our use cases that the searcher is able to execute at the same time as the writer, because we get thousands of tweets every second. And every treat, we have to write into the index. And so if you paused all the searchers while the writer was working, then it would take a really long time before you are able to search again. Or you might never be able to complete your search request.

So the way we deal with that is we have the searcher and the writer accessing the data structures concurrently. And the way that we do that for the link list is you never expose an element that the writer is working on until the writer has finished with it. So then once the writer is finished working on the data structure, you let the searcher see it. You publish a pointer to the next point in the data structure list.

With the unrolled link list, the only way to easily do that without – Or the only way to feasibly do that in Java and to use these memory efficient data structures that we've come up with is they have to be append-only, or prepend-only. So you can only add elements at the start of the list, otherwise the searcher would be able to see an element that hasn't been fully initialized. Or you'd have to copy your data structures and it would become a lot more expensive to do the reading and writing. That didn't satisfy our requirements, our use case.

So the skip lists satisfy a lot of the same requirements, and that they're really fast for reading, really fast for writing. They support concurrent access. But they have the additional benefit that you can easily add elements into the middle of them. So skip list is kind of like a link list, except it has skip pointers into various points in it. And those make it so that you're able to access any

element of the list and login time, and you're able to rewrite internal elements without invalidating the previous one. You can also do that in login time. And so it ends up being a really efficient way to mutate those posting lists.

[00:25:16] NT: And for people who have lost the lead of what the heck we're talking about, the skip list is an underlying data structure that is used for what?

[00:25:26] NT: Yeah. Sorry if I lost anybody there. The skip list is used as the data structure for the posting list. The posting list is the document IDs that have a given word in them. So it might be the posting list for a cat. It's my favorite example. You might have the posting list with all the documents with the word cat in it, and that's used for the quick retrieval of those documents. And the skip list is another kind of – It's an implementation that we used to satisfy the requirements of the posting list. So it can tell you what are all the documents that have this term. And it has a little bit of extra information there, like the position. So you could do a phrase query. So the skip list is the backing structure for the posting list. Does that make a little more sense?

[00:26:14] NT: It does. And the skip list is essentially trading off space and time, right? Like the look up is faster, and you tradeoff for that, because each record or pointers to the records are duplicated in the different layers.

[00:26:29] NT: Yeah, exactly. Each element will have a pointer associated with it. And that's how you're able to rewrite things internally. That's how you're able to mutate it in a concurrency safe way. And then the overhead then is, yeah, that you have to track different points into skip list so you're able to efficiently access each element. So there's a little bit of overhead given with that. But it's pretty tightly bounded, and it's also pretty tunable. So you're able to say, "Hey, I want half of the elements to have a parent pointer." And so then your overhead would be bounded to about 2X, I think. Or you can make it much lower. You can say, "Oh, I just want 1 in 20 elements to have a parent pointer."

And then if you say just wanted 20 elements has a parent pointer, then you'll probably have to search over 20 elements before you get to the hit that you care about, versus if you say, "Oh, I want half the elements to have a parent pointer," then you'll probably only have to search over two hits before you get to the one you care about.

[00:27:32] JM: And so can a skip list be searched in a concurrent or a parallel fashion? Do you have multiple threads that are searching the different layers of a skip list?

[00:27:41] NT: Yeah. It can all happen concurrently. So the servers that Twitter uses can have tens or maybe even 100 cores running at once, and all of those can access the same data structure at the same time. And that's really important to us, because data structures take up a lot of memory. They could take up hundreds of gigabytes on the machine. And if every thread had a copy of the data structure, then we would run out of memory really quickly. And it would never work. And then each writer would also have to write to every one of those data structures. So it'd be really tough for our use case.

So the way we do it is all of the readers or the searchers are executing a query, and all of the underlying data structures are safe to access at the same time, and we make sure that we follow the Java memory model closely and make sure that we don't ever expose like unsafe behavior where maybe a list is partially initialized, but partially not initialized. As long as you follow the spec carefully and you make sure that you kind of have a system of concurrency rules, then you can recode that evaluates against those concurrency rules and within those rules. And you can do the searching at the same time as you do the reading. And you can have all that running in parallel and avoid all the overhead of duplicating everything. And you can have the writer working at the same time as long as it's following the specification of the Java memory model.

[00:29:19] JM: And tell me what happens in an update to the search index data structure.

[00:29:27] NT: So when we get an update – As I said, we like index everything, or we tokenize everything into the constituent parts. And eventually it gets to our indexing server. And the indexing server says, "Okay, I have this document." And the first thing we'll do is figure out what document ID it should have. And that's a whole another can of worms. But the document ID is typically a 32-bit in Lucene. It's always positive. And so we say, "Okay, this document got document ID 321," and then take that document ID, 321, and put it into the posting list for all the terms that this Tweet has. We have the posting list for cat. And let's say the tweet has the word cat in it. So we'll look up the posting list for cat and then try to traverse that and see if the

document ID is already in that posting list or in that skip list there. Assuming that it's not, you'll have to add a new element on to the end of the allocation array.

So you determine that it's not in the posting list, and you're trying to add the document ID 321 to the posting list for cat. And so the way that you do that is you allocate a new element on to the end of our allocation pool, and that will have the document ID in there. It'll have the position, maybe some scoring information or some extra data. And then once you've allocated that on to there, as you were – You search through the skip list and you try to find the position where the new allocation would go. So as you're traversing down you say, "Oh, is this element greater than 321? No. We'll keep going. Is this one greater than 321? No. Keep going." And then finally you hit that element that's greater than 321, or the end of the list. You say, "Ah! This is where I put the element."

And so you'll grab that pointer that was pointing to the element that was greater than 321, for example, is going to element 400. And you say, "Okay, I'm going to change this. So instead of going to element 400, I'll have the pointer go to element 321." And then in element 321, you'll make sure that points to element 400 so you still have the contiguous list. So they're all allocated smoothly and you don't break up the list, right? You still want searchers to be able to be searching through the list. And either they see element 321 or they don't, but they still get to see element 400 and all the elements that you had previously allocated in there.

[00:32:17] JM: Tell me a little bit more about how you've changed the tweet ingestion pipeline to improve latency.

[00:32:25] NT: So when we had this 15-second limit that was kind of imposed by our system and our design, we had a lot of different – We could kind of say, "Hey, it doesn't really matter if we wait around for this. We can wait for that service to finish running its analysis on the tweet and then do it a few seconds later." We can say, "Oh! Basically, it doesn't matter how fast anything goes," because 15 seconds is a lot of time, but when we were trying to drive it down to one-second, or basically user imperceptible indexing latency. So we had to remove all of those delays to be able to do that.

And there were two really big ones for that. The first one was getting rid of the sorting. So as I was saying before, all of our tweets were sorted by the time or by tweet ID. And tweet IDs have this interesting property where they're ordered by time pretty much. So all of the – The most significant bits are ordered by time. And so if you sort everything by tweet ID, they'll also be roughly sorted by time.

And so we relied on that property. And so we would gather up all the tweets over the past 5 or 10 seconds and sort them and then write them out. And once those got written to the indexer where they'd be in sorted order. The indexing server could always rely on that, and they would be able to consistently write out the tweets in the correct order. And that was required for a previous posting list implementation.

But for our new posting list, we added those changes so that we could make the mutations in real-time. And so we said, "Hey, you know what? We can get rid of these delays and these sorting buffers so that you can write into the index as soon as they come in." And sometimes they'll be sorted by time. Sometimes they won't. And it doesn't matter. You can just write it as they come and put it into the correct spot in the posting list. So that was one big change we made.

The other big change we made was we decided to split the indexing into the real-time indexing and the asynchronous indexing. And we talked about that little bit. We said, "Okay, we'll take the core data structure, index that. As soon as it's available, write it all into the index." And then the extra data, maybe the other services take 10 seconds to finish doing whatever they do." And then once all that's finished, we can write the update. So it will get like a more enriched document with different features, different signals. And we'll write that update the index a little bit later.

[00:35:11] JM: Is cost management an issue when you're talking about how to scale up this search engine pipeline?

[00:35:18] NT: Yeah. Cost is definitely one of the things that we think about on our team and at Twitter. Twitter has their own computational resources, and those have a mechanism where they tell us how much they cost basically. So you can say, "Oh! Hey, these servers cost

$100,000 this month," or maybe a lot more, or maybe a lot less. As that member gets bigger and bigger, it's really easy to add servers. But at some point you say, "Holy cow, that's a lot of money. That's enough to pay the salary of a few engineers."

And so once you start getting to the point where you're paying hundreds of thousands of dollars a year for servers, it becomes worth it to invest that time into optimizing the computation. Or maybe if it's computationally inefficient and that's affecting like a user's query, like it's increasing the time it takes for them to load their phone, load their hometown line on their phone. And so at that point, you would want to start optimizing for cost and trying to see, "Hey, is there a way we can make these queries faster? Or is there a way that we can allocate our servers in a way that costs less?"

And so for us, what we typically do is we try to – Basically, we try to release it in and computationally-efficient way. And then once it goes out, we'll measure latency. We'll measure the averages. We'll measure the high-percentiles too. So sometimes we'll make a change where average latency looks fine. Basically, no different at all. But then maybe one in a thousand queries takes three times as long. And those are the kind of bugs that are really hard to track down but are important to do for cost saving and for user benefits.

And so we'll try to keep tight metrics on that and then try to make sure that cost doesn't grow out of proportion with the amount of traffic that we're taking.

[00:37:21] JM: What's the processor rolling out code changes to the search index?

[00:37:27] NT: That's a pretty – At Twitter, we've got some pretty developed tooling, and like lots of companies also have the same CICD thing. You could rely on TravisCI. It automatically push your build into production. And we do something kind of similar. We have a periodic deployment schedule periodically, daily or weekly. The code will get deployed by our system. And the way we do that is each data partition, basically, will update independently on. And so we split our data into partitions so that we're able to improve the speed of searching and we're able to reduce the size of the index in a particular server, because if we had all the entire index of all the tweets ever on one server, it would be petabytes. And so it wouldn't fit on to single server and be able to search it in an efficient way. So we break it into many smaller pieces. And each

partition will be – You'll roll the code on to one. That code – Hopefully, there is an index that it can load. And if you can love the index, it's pretty quick to start up. Sometimes there will be a change where we have to change the index format. So we won't be able to load an index.

For examples, if like the data structures that we're using change, it's often easier to just rebuild the index. And so you can do that. And the way we do that is by reading the raw data once again and indexing it all. And then once that server comes online, we say, "Okay, this data partition has enough replicas in it." And so we can restart the next replica. And that replica will be restarted with the new code and possibly the new index format.

[00:39:16] JM: Are you still working on changes to the search index? Or is this like a completed project?

[00:39:24] NT: We finished with the indexing latency project. And like the indexing latency is now low enough to satisfy all the needs, or all the product requirements that we're aware of. And so we haven't been doing too many changes to the core indexing data structures to produce the indexing latency, but it's really common. We're making changes in the search engine all the time. It's very common for customers to come to us and say, "Hey, I want to add new field to the search index. Or I want to be able to – I want to change the tokenization of a particular language."

And so that'll require us to go in and maybe we'll need to add new query operator. Maybe we'll need to slightly modify how an existing data structure works. Maybe we'll need to modify some of the internal caching mechanisms that we have going to support the new requirements, to support our new customers. So the core is pretty solid with the new changes. But we're always changing things around the core, always changing the data of trying out new experiments.

[00:40:36] JM: What changes do you anticipate making in the near future?

[00:40:40] NT: As far as data goes, we're always trying to optimize for basically our product teams. So some of the things that we're trying out are basically a more flexible way to index new signals. So the way that product teams retrieve a search query, they can specify, "Hey, I want to score these tweets by the tech score and the number of favorites, for example. And so one of

the investments that we've been making recently is allowing customers to send us whatever signals they want.

So we've created like a service that can accept new incoming updates, and maybe a customer can say, "Hey, I want to add the number of quote tweets that this tweet has got the index, and so that I can score based on that feature later." And so one of the things we've been working on is making it configurable to add new scores and make it really robust so that any customer can send us any signal that they want. And so we're basically trying to free up as much resources as we can for those customers and try to make it more flexible. So we're not in their way when they want to add features to the index.

[00:42:05] JM: It's worth just asking a little bit about Twitter engineering as a whole. In 2020, Twitter has become a basic component of the internet. It extends into so many other applications and it's such a fun part of our world. What is it like being in the eye of the storm?

[00:42:16] NT: I feel like in the eye of the storm. It is calmer than it might appear. And I think the biggest reason for that is we have – As a company, we've settled on our values and our vision. I think that's something that doesn't really change over time. I think that we've said we want to be the place where the public conversation happens. We want to be the place where people break the news, where they analyze the news. You can come to see how you should think about the news. And I think that really gives us a little bit of stability and clarity. When you believe those things really strongly, you can be a little more disconnected from the day-to-day tumultuousness of what's happening out there. Yeah. Basically, I think having a mission-driven company is really important when times are wild.

[00:43:15] JM: Any final reflections on this search indexing latency problem? Do you think anything – I don't know. Maybe there's somebody out there who's having search problems. Some piece of wisdom you can ascribe to them.

[00:43:27] NT: I think a lot of the problems feel like they're hard technical problems. So when you're looking at this, you say, "Oh man! I really want the drive down the indexing latency, or I really want to improve the speed at which we can search over this, or reduce the memory usage. And I think that's kind of our inclination as software engineers, because those problems

are tractable. They're something you can name and you can analyze you can figure out a solution for. But I would say keep the end goal in site. And for some, I'd say – For twitter, a lot of the important parts is like being quick to iterate on consumer features or being able to adapt if there is a gigantic spike in the rate of tweets or the number of people who are using Twitter.

So think about the direction you're driving and how flexible you will be to future requirements, because in the end, it's all about the value that you deliver for your customers. Whether that's an end user on our phone, or it's an internal team at your company where they might want to change how the search works. So try to keep that in mind as you're building.

[00:44:44] JM: Well, Nico, thanks for coming on the show. It's been a real pleasure talking to you.

[00:44:47] NT: Thanks for having me, Jeff. It was fun.

[END]