EPISODE 329

[INTRODUCTION]

[0:00:00.5] JM: Search is a common building block for applications. Whether we are searching Wikipedia or our log files, the behavior is similar. A query is entered and the most relevant documents are returned. The core data structure for search is an inverted index.

Elasticsearch is scalable, resilient search tool that shards and replicates a search index. Philipp Krenn from Elasticsearch joins the show today to discuss how search works and how Elasticsearch scales. We use Wikipedia as a running example for how a query is processed and how documents are stored. If you've ever wondered how search works, or if your company uses Elasticsearch and you just want to know more about it, this is a great episode for you.

[SPONSOR MESSAGE]

[00:00:54.0] JM: Spring is a season of growth and change. Have you been thinking you'd be happier at a new job? If you're dreaming about a new job and have been waiting for the right time to make a move, go to hire.com/sedaily today. Hired makes finding work enjoyable. Hired uses an algorithmic job-matching tool in combination with a talent advocate who will walk you through the process of finding a better job.

Maybe you want more flexible hours, or more money, or remote work. Maybe you work at Zillow, or Squarespace, or Postmates, or some of the other top technology companies that are desperately looking for engineers on Hired. You and your skills are in high demand. You listen to a software engineering podcast in your spare time, so you're clearly passionate about technology.

Check out hired.com/sedaily to get a special offer for Software Engineering Daily listeners. A $600 signing bonus from Hired when you find that great job that gives you the respect and the salary that you deserve as a talented engineer. I love Hired because it puts you in charge.

Go to hired.com/sedaily, and thanks to Hired for being a continued long-running sponsor of Software Engineering Daily.

[INTERVIEW]

[0:02:25.4] JM: Philipp Krenn works at works at Elasticsearch on infrastructure and developer advocacy. He has a degree in software and information engineering and has worked for many years as a web developer prior to joining elastic. Philipp, welcome to Software Engineering Daily.

[0:02:38.8] PK: Hi Jeffrey, it's a pleasure to be here.

[0:02:41.3] JM: It's a pleasure to have you. Most listener know what a search engine is. We all use Google to search webpages, but there are many other things that can be searched through. What are some applications that search is useful for?

[0:02:56.6] PK: Pretty much everywhere where you have lots of information, you want to search for something. Elasticsearch, for example, is widely used in lots of websites and applications. One of them would be Wikipedia. If you search anything in the search box in Wikipedia, your search will actually go through Elasticsearch in the background. If you search on GitHub, or Stack Overflow, all of those will actually go through Elasticsearch in the backend. If you don't want to rely on your search going through Google, but have it on your own servers or within your own site, then you would need some extra software to actually do that for you.

[0:03:33.1] JM: If we're talking from an architectural point of view, not from a specific type of search engine, can you describe the high-level components that fit together to allow users to search through a collection of data?

[0:03:48.5] PK: Yes. Normally, I compare it to relational databases or database in general, where I would say that database is a very much black and white; you store something and then you can retrieve it. Whereas full-text search normally operates more in shades of gray. You're not only interested in exact matches, but you want to search for something like a concept, so it's more like shades of gray. You don't really care about if it's single or plural. Sometimes you don't even care if it's a noun or an adjective, you just want to search for that concept.

Full-text search normally adds some overhead during the store part, what we call index. You would just index some document you have and do some additional work for that before actually storing it so you can retrieve it afterwards easily. What that indexing would do is, normally, you would remove any formatting, like HTML texts for example. Then, you could remove something like stop words, which are very common words, which appear in nearly every document and would not add much value to your documents.

Then, you can do something like stemming, which basically removes the word endings and uses a word down to its root so you don't care about singular or plural or any specific flexion. Then, you could also add stuff like synonyms. If you have multiple terms for the same meaning and you don't know what the users want to search for, you could just add those synonyms to allow them to search for anyone of them and find the content you have.

[0:05:25.1] JM: Right. You're talking about the process of building an index to be searched, and the index that we're searching through is a way to lookup documents. You often have a corpus of documents. The Wikipedia example is perfect. You've got the entirety of Wikipedia, it's all of these articles about biology, and physics, and history, and you want to be able to map search terms to documents. In order to do that, you build what is called an inverted index.

I think you described it pretty well, where you're building a way to map search terms to those different documents. If I have an article about swimming, then the inverted index will look through the article about swimming and it will find things like water, and chlorine, and laps, and then it maps those as the keys in the index to that article. All of those keys might map with some degree of accuracy to that article. How closely they map to that article is part of the fuzziness that you're talking about where it's not like a relational database where you're always going to get swimming if you look up the word water, but you might get it sometimes.

[0:06:44.1] PK: Yeah. What normally happens, for example, if we stick to the swimming page on Wikipedia, we would just take that entire document, then extract the — We call them tokens, which is basically all the words you have there. Then, we run them through this indexing pipeline we have. For example, swimming would probably be reduced down to swim. If you have swim, swimming, that wouldn't make a difference. We would just store the word swim.

Then, you don't even need fuzziness right away. As soon as somebody searches for swim, you would have a direct match. You could add fuzziness on top of that. If somebody would misspell swim, even though that's bit hard on that specific word, you could optionally add something like fuzziness there. Of course, you could add synonyms for swim as well if people want to search for something else.

[0:07:38.6] JM: You're describing these different ways that you can add richness to what is the core abstraction of the inverted index, where search terms are mapped to specific documents that we're looking up, and I'm glad that we've been able to give people an explanation for this inverted index in case they needed a reminder or an introduction to it.

With those concepts in mind, with that being the core data structure that we're building here, let's talk about Elasticsearch. What is Elasticsearch?

[0:08:10.9] PK: Elasticsearch is one of these full-text search engines. It's based on Apache Lucene. That is actually what is doing all the hard work we have just discussed. Lucene is the thing storing the data on this and making it searchable. It's doing the index process; so stop words, stemming, tokenization, all of that is Lucene's work. Kind of around that, a shell, we have Elasticsearch. That shell or that outer layer, the Query DSL, it provides rest interface and does the replication and distribution of your data over multiple nodes. The actual search work is done by Apache Lucene, and then a nice way to interact that, that is provided by Elasticsearch.

[0:08:58.1] JM: What does the term elastic refer to?

[0:09:01.3] PK: I'm not even really sure. We started off as Elasticsearch. Actually, Elasticsearch was kind of the third implementation. The first two were called Compass One and Compass Two. Then, Shay, our CEO by now, renamed the third one Compass Three, but he called it Elasticsearch. After we started adding more and more products, it was not just Elasticsearch anymore. I guess it was just shortened down to Elastic. Yeah, that's what we have been sticking to. Since elasticity and scaling and stuff like that is a common concern, I guess elastic is very fitting.

[0:09:39.3] JM: Yeah, that was what I was hoping it indicated, because I've got a lot of questions about scaling, because, of course, if we're building a search engine for something as big as Wikipedia, or something as big as all of the logs that we're storing for application, we're going to need some elasticity. Can you give an introduction to some of the elastic properties of this search index that we're building?

[0:10:07.6] PK: There are generally lots of moving parts. Let's start with kind of the simple stuff. In the background, your store your data to one index, the index is pretty much like a database in the database world. That one index can be split into multiple so-called shards. By default, those would be five shards. Each of these shards can either live on server, or if you have multiple servers, the data would be distributed over your multiple servers automatically. That is how we split up data. If you have more than five servers and want to use more servers for storing your one index, you would need to change the number of shards when you create that index.

In addition to that sharding, we also replicate the data. By default, we have a replica of one, basically, meaning you have your five shards. Then, for each one of these shards, you would have one copy on another node. If one node dies and a few shards go missing with that node, you will still have those replicated on another instance, and that is kind of the basic concept how to distribute data and also be resilient to failures. By using that approach, it's very easy to scale horizontally, so you just add more nodes to your cluster and your shards will be automatically distributed and replicated and that's basically all you need to care for.

[0:11:38.2] JM: I think this is a good opportunity to talk about sharding and replication more generally, because this is a technique for building a resilient system that is true whether we're building a search index or a relational database. I know you just gave a pretty good explanation for what sharding and replication do, but can you go more generally into what sharding and replication are, just maybe give an alternative explanation for the explanation you just gave.

[0:12:11.3] PK: I think the term sharding actually started in a computer game. Was it Ultima online? I cannot recall. There was a game and there was some crystal, and that crystal was split up into multiple shards, and I think every shard was one of the game worlds, something like that. That is where the term sharding originally comes from.

You have one big thing and you want to divide it into multiple smaller, more manageable pieces. In the example of the crystal — Yeah, the crystal was split up and each one of them contained some part of that world, where anything that stores data, the shards will contain some part of that data you want to store in its entirety.

[0:12:53.3] JM: The number of shards that we have a reflection of how diverse, I guess, the volume of the data — The volume of unique pieces of data in our database and we break up those shards and then we replicate them some number of times, because, first of all, the entire sharded database is too big to fit on any one machine. Second of all, if one of your machines with a unique piece of data fails, then you want to have a backup. You want to have your shards replicated on to different machines.

We're dealing with these compute environments where machines are failing all the time. For the average user, they're working on AWS and maybe don't notice any failure. If you work at a company of any reasonable size, you start to notice failures when your machines — The machines that you're spinning up, they'll fail occasionally.

Since many of the biggest companies are hosting their databases on a cloud service provider, like Amazon Web Services, and these computers are failing all the time, it is important to have a replication factor, which is how many times you're replicating each piece of data that can account for how often these machines in the datacenter, somewhere, are failing.

What are some strategies around deciding how often you're going to replicate each shard of data?

[0:14:26.9] PK: Maybe we should take a step back to talk about why would you shard? That is one of the most common questions. How many shards do you need and what is your replication factor?

Think about replication, is they help with two things. Firstly; they make your data more resilient to failures. Depending on how many copies of your data you have, you can take more numbers of service that fail. Secondly, in the case of Elasticsearch, the replicas can also be used to increase your read capabilities. You can either read on the primary shard, or any of its replicas.

If you have some high-traffic website with — For example, you have a shop, and Black Friday comes along and lots of people want to read more data from your shop, you could just dynamically increase the number of replicas to scale up your read since you have more sources to get your data from.

On the other hand, to scale up right, you will need more shads so that more service can spread that writing load evenly over all of them. What further kind of dictates the number of shards you want to have is either how much data can one node store. How many service will you need to store the data. B; what is the right throughput you want to achieve overall your nodes. That will kind of dictate your number of replicas and shards. There are no fixed numbers. Often, people ask, “What should those numbers be?” Our answer is always, “It depends.”

One thing you should definitely avoid is thinking, “Well, I don’t know many shards we’ll use in the future. I will just add a hundred,” which is totally a random value, but you should not do that, because every shard has a specific overhead in terms of file handles and memory. If you have a rather small cluster and have lots and lots of shards, we always call that oversharding and you will just lose a lot of resources just for sharding your data.

Also, then, if you search over all your shards, you will need to combine those results from all the shards. That adds quite some of your overhead. It allows you to scale up very highly, but if you never have that problem to scale that much, you’re just wasting lots of resources on the number of shards. There is no final answer what is kind of the right shard or replica number. It will always depend on your use case. Also, what is your write load, what is your read load, and what growth do you anticipate for the future.

[SPONSOR MESSAGE]

[00:17:12.0] JM: Software engineers know that saving time means saving money. Save time on your accounting solution. Use FreshBooks Cloud Accounting Software. FreshBooks makes easy accounting software with a friendly UI that transforms how entrepreneurs and small business owners deal with a day-to-day paperwork. Get ready for the simplest way to be more productive and organized. Most importantly, get paid quickly.

FreshBooks is not only easy to use, it's also packed full of powerful features. From the visually appealing dashboard, you can see outstanding revenue, spending, reports, a notification center, and action items at a glance. Create and send invoices in less than 30 seconds. Set up online payments with just a couple of clicks. Your clients can pay by credit card straight from their invoice. If you send an invoice, you can see when the client has received and opened that invoice.

FreshBooks is also known for award-winning customer service and a real live person usually answers the phone in three rings or less. FreshBooks is offering a 30-day unrestricted free trial to Software Engineering Daily listeners. To claim it, just go to freshbooks.com/sed and enter Software Engineering Daily in the How Did You Hear About Us section. Again, that's freshbooks.com/sed.

Thanks to FreshBooks for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:18:54.0] JM: To give, maybe an idea of at least some relative sharding and replication factors — Wikipedia is a giant application and there're tons of users that are using it and it's also got a lot of updates. It's getting dynamically updated throughout the day. Then, there's also the log management system for Wikipedia, and I imagine that — Let's just assume that Wikipedia uses Elasticsearch to manage its logs. I think we'll get into log management with Elasticsearch in a little more detail later on.

Just as two examples, how would you contrast the scalability, the sharding and the replication factor of a site like Wikipedia with the log management system of Wikipedia?

[0:19:41.4] PK: Generally, your excess patterns for search and logging are a bit different. For search, normally, you have — I would not say small, it can be in the gigabytes or low-figures of terabytes data, but your search use case normally is kind of limited in the size of data. You always want to allow very quick searches on that.

There, you will probably have more service for less data. Whereas, for the logging use case, you often have rather old data and you do not need queries to return that quickly. There, you often have, like the limiting factor for the log use cases often cannot ingest all the data I'm producing. If you have a thousand to ten thousand updates or requests per second to store, that will dictate how many nodes you need to have and how much storage do you need depending on how long you store your data.

Whereas, for the search use case, you will have, often, less data, but it needs to be searchable really quickly and probably you want to have lots of memory for that. Whereas, for the storing of log data, often, you will have larger discs since it's not that critically searched, it takes two or three seconds, which, on the other hand, would be totally unacceptable for the search use case. It's just finding kind of the right tradeoff for the use cases, and that will then dictate what kind of service you need and, also, how many service you need.

[0:21:14.7] JM: We jumped to scalability. Let's scale back a little bit and talk about just the core search index again. I probably should have gone over this more in the beginning, but I think it's okay. I think it still fits here. We could talk about types and mappings that you can setup in Elasticsearch, but I think people can look this information up.

Basically, every type consist of a name and a mapping. A blog post, for example, would have its own mapping, which defines the properties of that type that we want to index. If you're indexing your blog with an Elasticsearch search engine, you will give every blog post a mapping that sets up — It gives the search index some information about how you want results to be returned. This is sort of like setting up the schema of a database. This doesn't answer as much about the fuzziness.

Let's talk a little bit more about the fuzziness, because as we discussed in the beginning, there is this culmination in a search engine like Elasticsearch where you've got part relational database type of behavior where the searching has a clear definition what kinds of results it's going to return, then you have more of a fuzziness also. Can you describe how the relational type properties and the fuzziness, affect how a way search query might return results?

[0:22:50.7] PK: Okay. Before we jump into the searching part, one point about the types and the mapping, firstly; mappings can either be created when you create the index, which you always do for production use cases, but it would be automatically generated from the first document you insert into an index. If you have some attributes, Elasticsearch would try to guess the right data types from those feeds you provide. Which is often working reasonably well for the prototyping phase, but for production, you should not really rely on that.

Secondly; the types, I always say they're like an enumeration type or something like that. On one index, you can store different types of information in one index. The only problem is Lucene, the underlying storage engine does not really know about these types. We have just decided one or two weeks ago that types will, in the future, go away in Elasticsearch. Do not rely on multiple types in one index too much anymore, because they will be gone in the foreseeable future.

Coming back to the original question about search. Basically, what you can do search-wise is we have — On the one hand, we have something like bullying queries, where you can define exact things to match. You could just say, "This must have a specific feed," or "this must be in a specific range." Then, we have the full-text search capabilities which can work on these index terms. For example, you have the stemming from swimming to swim, and it will search on everything, anything that resembles this swim regardless if it was swim, swimming, whatever.

On top of that, you can put more advanced search concepts. You could use n-grams, which would be use search for specific groups of letters in terms if you don't have an exact math on something. That would be, for example, if you have blueberry, or let's say we have blueberries in the plural. Elasticsearch for stemming would reduce that to blueberri with an I at the end, but that would never match on anything that is blue. To search for something like that, you would need to extract groups of letters to actually be able to match that blue, since we do not do any partial matches otherwise. Otherwise we would just compare the entire tokens we have.

Another advance search context would be we can add these fuzziness. We can say a few letters are either missing different or too much between what we have stored and what you're searching. Basically, this is Levenshtein distance. With a Levenshtein distance of one, for example, you could have a word in your index, and what you search for if that has one letter to little, one letter different, or one letter too much, that would still match. In the beginning, we implemented it quite naively with a brute force approach, but it has been then re-implemented with full automaton to actually be able to generate kind of a network of all the possibilities to not rely on the brute force approach of fuzziness, so that should make it much quicker to search for things like that.

[0:26:21.7] JM: Okay. Now that we've talked about the setup of our indexing, and we've talked about the elasticity component, let's talk about a read from top to bottom. Let's say I opened up Wikipedia, I do a search for — I don't know. How to learn to swim? Let's say that. What's going to happen on the server side when that query hits the Elasticsearch cluster?

[0:26:52.9] PK: We have declines for different languages. We have the bindings for whatever your language be; Python, Ruby, Java, .NET, and lots of others. You would probably use that binding, or you could just fire a rest request against any of the nodes you have in your cluster.

Normally, we'll round-robin the nodes you have configured in your connection. You pick one of these nodes by randomly, since you round-robin, and that node will be your client node. Previously, we called that coordinating node as well, and that will coordinate your query. You send the request to that one node, that node then figures out, "Okay. This is this specific index. The index has these shards and replicas," and it will make sure that your query hits every single shard either primary or replica and hit the right nodes where these shards reside.

Then, each node having one of these shards would actually do your search on that node and get the partial result of that node. Then, it would send that partial result back to the coordinating node. That node would then put together the entire result.

Actually, it's even a bit more complicated than that. In the first step, all the shards would only return kind of their matches. If you search for the 10 best hits, each of these shards would return its own 10 best matches, but only the I.D. and the so-called score, which is basically a quality attribute. The coordinating node would then take the 10 best results globally from all the sub-results it has and would request documents, the nodes, having the data they have, just requesting them through the I.D. it has received.

It's a two-step approach. It's like gather, where you just get the I.D's first and then collect the final results. Then, the coordinating node will send you back that result that you have just requested.

[0:29:06.1] JM: Okay. In that example, you talked about the client or coordinator node. Is this any node in the cluster? Are all of the nodes potentially client nodes?

[0:29:19.8] PK: By default, it could be any node, but you can split up the roles of the Elasticsearch nodes. We have data nodes. These keep data and do the actual search work. We have muscle nodes. These keep track of the data soft the entire cluster .We have client nodes. These don't keep any data themselves. They are just coordinating these queries and requesting the results.

Quite freshly, in the latest five release, we have so-called ingest nodes, which can do some parsing and enrichment of the a data when you store it. We also have tribe nodes, which can connect to multiple clusters. By default, a node would be a master node, and a data node, and would just do all the work. Once you have a bigger and bigger cluster, you might want to split up these different roles of nodes to specify them a bit more and to keep your cluster in a more reliable state.

You would start out by splitting out the master nodes, so that they just keep the state of the cluster and don't do any other hard work to keep your cluster state quite reliable. Then, you could also split out these client nodes. For example, if you do a very big aggregates, that takes up a lot of memory. You do not want your data nodes to swap out all the data they have just fetched from disc to memory, because they need to put together some big aggregate.

You wouldn't want to pay the price to always re-fetch data from the disc just because some other process just took up all your memory and expired the cache you have. You would split it up so your data nodes just have as much data loaded into memory as possible, whereas the client nodes, they're good for coordinating your queries and then combining results together to the final set you want to return to your application.

[0:31:19.8] JM: As I mentioned earlier, these elastic search clusters are often in a datacenter somewhere with dubious hardware quality on some of the machines and nodes are failing all the time. You may have a failure in the middle of a query. You may have a failure in the middle of a write. Actually, before we talk about those failures, we just discussed a read. Why don't you discuss a write?

[0:31:46.8] PK: Okay. A write, it will depend, do you provide the I.D. yourself, or do you let Elasticsearch generate the I.D.? In any case, any document you have will have an I.D. either assigned by you or generated by Elasticsearch.

Then, the next step is whatever the client node is, it will hash that I.D., so it will be evenly distributed overall the shards. Then, after hashing the I.D., it knows which the right primary shard for that data is. That is part of the cluster state information. It knows these area of the hash value is for that shard. That shard resides on that specific node.

It will forward the data to the primary shard where that I.D. landed on after being hashed. Then, the primary shard will check, "Is this a meaningful document? Can in insert it? Does that not conflict with the mapping I have?" For example, if you have a field and that is of the type integer and you try to insert a string there, the node will immediately say, "I cannot process that," and would give you back a client error.

Assume everything works and you cannot find a document to that specific index and type, it would write that data down in — We call it trans log. In Phosphorus, it would be your write ahead log. That data would be written down. Then, it would be forwarded to the replicas, by default, one. You could set it to zero or more than one. These would then accept the write, write it down as well. Only then would the primary shard acknowledge write back to the client node, and would then let it know, "Okay. I have applied the write operation al down." So it would return at 201 if it had created the documents, or 200 if it had just updated the document.

One thing that is might be a bit surprising is that data, if you do a full-text on it, is not immediately available by default. By default, we have something called a refresh of one second. Every second, all the documents you have added would be added to this inverted index. Actually, Lucene would create a new segment for each of these one second intervals you have, and that is what is then stored on disc and what is actually searchable.

You can immediately retrieve a document by its I.D., but it's only available through the full-text search part after this refresh interval has parsed. That, again, is a tradeoff of performance. You could either set this much higher than you would write fewer segments, but it takes long or until your documents are searchable, or you can keep a low-level value, like one second, or you could even disable it entirely, but then you would have lots of very small documents, which will probably not perform that well. Your documents are very quickly searchable then. Again, it depends on how quickly you want to be able to search something.

[SPONSOR MESSAGE]

Encapsula has a global network of over 30 data centers that optimize routing and cacher content. The same network of data centers that is filtering your content for attackers is operating as a CDN and speeding up your application.

To try Encapsula today, go to encapsula.com/sedaily and check it out. Thanks again Encapsula.

[INTERVIEW CONTINUED]

[0:36:53.9] JM: We've talked through both the read and the write processes now, both of them involve a lot of coordination. What happens during a machine failure in one of these — Maybe you could describe some failure cases of read and failure cases of write. You don't have to go — I guess that's a big question, but take it for what you will.

[0:37:13.3] PK: Depending on what kind of machine phase, different things will happen. If it was just a node keeping data, at some point, the other nodes will recognize that. There is a specific time out, how long they will wait until they assume the node is really that. Once that the node is marked up as that in a cluster state, the master node will make sure that the data is replicated in the right fashion against.

The node that had died, if it had any primary shards and you have any replica shards of those, those replica shards would be marked as primary shards then on another node. Those shards would then be replicated to another node to, again, get to your replica factor you have configured. If some replica shards are on that server and are gone, the primary shard would make sure that it replicates its data to another node.

If your master node fails, there can always be one master node in your Elasticsearch clusters since that is the one instance that keeps track of your cluster state. The other master eligible nodes would do a vote on who can become the next master and who keeps then track of that master state. That is kind of what happens when a node dies. Then, if the read or write operation goes out and it's just failure is in flight, the operation would then just time out. There is time out for every query when there is an error just happening at that specific moment. As soon as we have something reachable again, the coordinating node would direct the traffic to the right node.

[0:38:55.8] JM: Okay. I have heard about these issues where if you're somebody who is managing an Elasticsearch cluster, there are these statuses, I guess like a red, yellow, and a green status that indicate how live a node on a cluster is. If you have things that are in a yellow state, this is equivalent to the canonical partial failure, and partial failures in computer science are typically harder to deal with than complete failures.

Can you talk about what causes partial failures and why those are so hard to manage in an Elasticsearch cluster?

[0:39:41.6] PK: The colors you mentioned, green state is you have all the primary shards you have area available, and all the replicas you have configured are also available in your cluster. The yellow state means you have all your primary shards, so there is at least one copy of your data, but your replication factor is not honored at the moment.

If you have a replication factor of one and no dies and your replica, or your primary is gone, you still have another copy. You have one copy, all your data is readable and writable at that moment, but you do not honor your replication factor. While the data is being replicated to another node, you will be stuck on that yellow state.

Also, if you have just a single node and you have the replication factor of one, it cannot replicate to that node. It doesn't make sense to replicate data on the same node. We will never do that. The data always needs to be replicated to another node. If you have a replication factor one with a single node, that cluster will always be in the yellow state. All your queries will work, but you do not have the capabilities to have all the copies you want, so you're not as resilient as you could be.

Only the red state means that some data is not available. For example, if you have had a replication factor of zero and a node with that one copy of the data fail, your cluster would be in the red state. If you have a replication factor of one and two nodes fail and one of them has the primary shard and the other one has the replica shard for that specific cluster, then your cluster would be in the right state. Now all your data is available anymore. That is basically what these colors mean.

[0:41:27.5] JM: Okay. We've explored many of the scalability aspects of Elasticsearch. We've explored the basic data structure that we are scaling, which is the inverted index. Let's talk about some application level, architectural level questions. The typical application these days uses multiple different databases to store its data and this is the polyglot persistence model.

Even in something like Wikipedia, where you're been discussing Elasticsearch is the main system of record, I imagine if I click on a link for a page, sometimes things with that page are going to have to load data from a database and it's probably pulling from a place that is not Elasticsearchs, because it wouldn't make sense if the query to get the page, the page assets, is very well-defined. You wouldn't want to go to an Elasticsearch cluster, I assume. At least tell me if I'm wrong — Go ahead.

[0:42:33.0] PK: I remember correctly, Wikipedia is using MariaDB for the storage of its data, and then for full-text search, would add Elasticsearch. Something like that is a very common scenario.

[0:42:46.6] JM: Yeah. Okay. Can you talk more generally about polyglot persistence and how Elasticsearch fits into a multi-database model?

[0:42:56.9] PK: I think polyglot persistence was very appealing in the beginning, or specifically for developers. For the ops people, polyglot often means quite a headache, because when you develop on some feature for two weeks, that is fine for you. If the ops people have then to maintain it for the next three years, they're probably less happy if you just add random data stores.

While I think that using multiple data stores makes perfect sense, I think you should be kind of careful not add too many just to experiment with stuff since your ops people will not be too happy with you in long-term.

There are these examples where you have five data stores for one thing, and I'm never sure if that is really a good idea. Of course, you can do that. One common aspect — Or Elasticsearch would come into the picture is if you need some good and scalable full-text search capabilities for your product, which is often an important point for shop systems, big informational sites.

Another point where Elasticsearch comes in handy is if you have any logs you want to store and then be able to search them, or if you have anything around metrics, analytics. Those are use cases where Elasticsearch often fits well. It can cover quite a lot of ground for that. There's always these elusive question, "Is Elasticsearch a good primary data store?" Normally, our answer is it's not really a primary data store. Some people use it as such, but we are not sure it's always the right fit for it. It always depends.

Sometimes it can work, but for many use cases, you don't want to put all your eggs in one basket and have it in the full-text search engine, especially for resiliency reasons. While we're improving a lot, there are always smaller issues you can run into. It's always a tradeoff between ease of use, powerfulness, resiliency. For relational databases, for example, they had , I would say, 30 years to bake. Often, NoSQL solutions has had any time close to that to evolve and mature.

Yes, often, a conservative approach for that is probably still a good approach. Even though I work for Elastic, I would not say just use Elasticsearch for everything. Be careful. Use it for the right approaches, or problems to solve, and then it will make you very happy. It's not the hammer and nail thing you just use one tool for everything. That's probably still not a great idea.

Yeah, polyglot definitely has a place, but just be careful not to have an explosion of data storage technologies. Probably, for many people, something like relational database will work well. Maybe a key value store for caching and keeping session information. Then, Elasticsearch for full-text search logging metrics, but it always depends. Another thing is there is so much technology around that you can just explore, but it also makes it much harder to find the right combination of tools, I guess.

[0:46:05.3] JM: If you're Wikipedia and you're using Elasticsearch and you're also using MariaDB, is MariaDB a full representation of the same Elasticsearch world, or does the Elasticsearch cluster — Does the single point of truth have to be replicated elsewhere? I guess my question is; is MariaDB here, in this architecture that we're discussing — I know we're not talking the exact architecture of Wikipedia, but maybe a hypothetical architecture that Wikipedia might resemble. Are we keeping MariaDB, which is a relational database — Are we keeping that because the query shapes that will hit that database are different than Elasticsearch, or are we keeping it because it's a backup, a reliable backup that we can use in place of Elasticsearch?

[0:46:52.1] PK: While I'm not sure of the specific architecture of Wikipedia, I don't think it's backup. I'm not sure that is the general use case people want to do. While SQL can be a bit hard and daunting at some points, it's still very powerful and widely used. Many developers also want to use that, so I don't think you would add a relational database just as a backup, but you will often interact with that directly. Whereas you would then just offload the search capabilities to Elasticsearch and probably in logging metrics and other use cases.

[0:47:28.1] JM: Okay. Let's talk a little bit more about the logging example. We touched on it a little bit earlier, but Elasticsearch is widely used a system of record for logging, and I think there are number of reasons for this. One I can think of is that the log data, there's so much log data. A lot of it is well-formed. You have your log messages that are well-formed, and so you can make a decent schema out of it using the Elasticsearch models for doing schema, the mappings, and the types.

Also, you can afford to lose some log data, I think, so it's not catastrophic if you end up having this type of failure that you're talking about. Why wouldn't you use Elasticsearch as the core system of record? Maybe it's okay to lose some data for logging. I shouldn't talk so much about this. Maybe you could just speak a little bit about the typical logging infrastructure that somebody is using when they're using Elasticsearch.

[0:48:30.1] PK: Yeah. First off, I'm just careful not to give the wrong impression here, but it's not like if you use Elasticsearch, that you will just lose documents every day. This is not what is happening. Even if you're testing stuff on the lab conditions and you just crashing nodes all the time, losing data is actually quite hard and it takes some effort to do. As long as you have configured your cluster and you're using an up-to-date versions, since we're improving quite a lot, and resiliency s well, data loss is not something you will experience commonly.

Most people who lose data are — Pretty much everybody, or all the cases I've seen for the last half year or so, or even longer where mostly bad configurations. There is probably nothing to do against that. Data loss is not something that happens very commonly.

Coming back to your original question. I think what was also the appeal of Elasticsearch is that it has more of this entire stack around it, because, at first, when you think about it, full-text search and log management don't really go together that well, or it just sounds like two different problems to solve.

The nice thing about Elasticsearch and the so-called elastic stack is that is kind of entire ecosystem around that. We have Logstash to parse and enrich your data. In the beginning, it would also collect your logs. Now, we have the beats, which are just like lightweight agents or shippers who can run on all of your servers. They will just forward data and can either insert the data directly into Elasticsearch or have the parse and enriched by Logstash.

Enriching would be something like you have an IP address and you want to get the geo-location of that IP address. You would just store the city, the country, and the region of that IP address so you can easily pinpoint that afterwards. Once you have everything in Elasticsearch, you can then use Kibana to visualize and explore your data.

With these four open-source products, you have a total stack to monitor your metrics and logs. I guess that is one of the big appeals. On the other hand, since it's open-source, it's very easy and cheap to get started. Thanks to the horizontal scalability of Elasticsearch, it's also very easy to scale it up. We have customers using dozens of nodes in one cluster, or even three digit nodes in one cluster just to store lots and lots of these logs you have.

[0:51:11.6] JM: Okay. We're running low on time, but just one more question as an application perspective, because I think we've touched on a lot of the lower level details. What if I was building a ridesharing platform and I wanted to search a geo-space to find the nearest cars to a user? My understanding is that Elasticsearch will be useful for this. Can you explain why that is?

[0:51:37.8] PK: Oh, yeah. We call him Geo-Nick. In earlier versions of Apache Lucene, you could do geo queries, so you have a longitude and latitude. You have a point. Then, you can either just calculate the distance, or do a specific geometrical figure around it and find stuff within a region. There were lots of capabilities, but performance was not that great.

Elastic people figured out that that was not — Or that was an issue at some point, and then they hired one of my, now, colleagues, Nick Nice. We call him Geo-Nick, and he re-implemented the geo features on Lucece. Once they were in Lucene, we could add them to Elasticsearch. Now, geo-locations are very useful in Elasticsearch and also very performant, especially since version five, since they were pretty much total rewritten there.

Geo-locations, just having a point and then finding another close point, or finding all the points in one region, that is totally doable. In the Query DSL, we have quite powerful geo query capabilities.

[0:52:45.3] JM: What are some of the features that are being worked on with Elasticsearch right now?

[0:52:50.0] PK: It's so many, that it's kind of hard to keep track at the moment. Big stuff coming in the next version is, again, on the resiliency side. We're adding sequence numbers, which should help with speeding up the replications in case of failures. Also, at some point, like cross-datacenter application. You cannot only have a cluster in one location, like one AWS region, but could probably replicate the data over to another cluster in another location. We will get rid of types.

We are always trying to improve the upgrade process. Right now, the upgrade process of Elasticsearch between major versions is a full cluster restart, which is, for many people, very painful. We're also working on lots of performance improvements both on the Lucene side and in Elasticsearch itself. The entire stack is kind of moving.

In all of the open-source and also our commercial plugins, there are lots of new stuff coming up. We've just had Elastic{ON}, our yearly conference in San Francisco, and there we already kind of got previews of some of the cool new features coming up. Kind of my personal favorites re for Filebeat, the thing that collects log files. It will automatically support specific protocols now, which makes it much easier to get started.

You can just collect something like syslog, engine X-logs, Apache logs. Those will be automatically parsed in the right format, so it will extract something like HTTP headers, or return codes, IP addresses, geo locations and store that in Elasticsearch, and you don't need to configure anything. Also, Kibana is getting lots of new visualizations and a way to visually build a metrics, or to visualize your metrics easier. Elasticsearch is improving on that upgrade process. Also, adding more advance features I don't want to dive too deep into right.

Logstach is getting its persistent queue mechanism, so that you probably don't need a queue in front of it anymore, which is a common use case. These president queues are in the works. We're also improving the monitoring aspect of all the components. Right now, we monitor Elasticsearch, Kibana, and Logstash. Beats will be added soon. In one of the upcoming versions, we will also collect logs. Elasticsearch, for example, produces in Elasticsearch itself again. You can search those easily. You probably want to throw that into a second cluster.

Just to grow the stack, right now, it's pretty much blocks of Legos, so you have all these elements, but there's some assembly required to put them together. I think that the next steps are to make this starting experience nicer. There are more features or more solutions coming right out of the box and less assembly for our user is required.

Even though you will still be able to configure anything you want. You're free to do whatever your use case is. It can be logging, search, analytics, metrics, the Elasticsearch is not a one-trick pony, just focused in one of them, but we're always trying to expand to newer horizons. For example, we've recently acquired a machine learning company. Machine learning, basically, anomaly detection. That is one of the big things that will come out very soon.

[0:56:19.0] JM: Philipp, where can people find you online, either social media, or otherwise?

[0:56:24.6] PK: Yes. I should be very easy to find. Once you know my online nickname, which Xeraa, which sounds very weird and many people it must be something like Zena, and I'm a girl, but no it's not. It's actually a Rot13 of my last name, which is Krenn. When you rotate these letters by 13, it's Xeraa. That is both my Twitter handle. You can find me at xeraa.net on the interwebs, and I'm pretty much wherever I am, I use that nickname, because nobody else does.

Once you have that, I should be easy to find. I'm doing lots and lots of conferences sand meet ups mainly in Europe. You will have a good chance to find me — I think, until summer, I have already 20 conferences scheduled now. Yeah, I will be around. If you're in Europe, it should be easy to find me and I'm always happy to hand out stickers and answer questions there.

[0:57:23.7] JM: All right. I hope you come to the states for an American tour sometime soon as well.

[0:57:30.3] PK: Yeah. I'm currently discussing with our marketing team if we could do that. Let's see.

[0:57:37.0] JM: Okay. Philipp, thanks a lot for coming on Software Engineering Daily. It's been a real pleasure talking to you.

[0:57:40.8] PK: Oh, it was all my pleasure. Thank you.

[END OF INTERVIEW]

[0:57:46.9] JM: Thanks to Symphono for sponsoring Software Engineering Daily. Symphono is a custom engineering shop where senior engineers tackle big tech challenges while learning from each other. Check it out at symphono.com/sedaily. That's symphono.com/sedaily. Thanks again Symphono.

[END]