EPISODE 934

[INTRODUCTION]

[00:00:00] JM: Redis is an in-memory database that persists to disk. Redis is commonly used as an object cache for web applications. Applications are composed of caches and databases. A cache typically stores the data in memory and a database typically stores the data on disk. Memory has significantly faster access time but is more expensive and is volatile, meaning that if the computer that is holding that piece of data in memory goes offline, the data will be lost.

When a user makes a request to load their personal information, the server will try to load that data from a cache. If the cache does not contain the user's information, the server will go to the database to find that information.

Alvin Richards is chief product officer with Redis Labs and he joins the show to discuss how Redis works. We explored different design patterns for making Redis high availability or for using it as a volatile cache on a single node, and we talked through the read and write path for Redis data.

Full disclosure; Redis Labs is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[00:01:19] JM: Monday.com is a team management platform that brings all of your work, external tools and communications into one place making cross-team collaboration easy. You can try Monday.com and get a 14-day trial by going to Monday.com/sedaily. If you decide to become a customer, you will get 10% off by coupon code SEDAILY.

What I love most about Monday.com is how fast it is. Many project management tools are hard to use because they take so long to respond, and when you're engaging with project management and communication software, you need it to be fast. You need it to be responsive and you need the UI to be intuitive.

Monday.com has a modern interface that's beautiful to look at. There are lots of ways to use Monday, but it doesn't feel overly opinionated. It's flexible. It can adapt to whatever application you need, dashboards, communication, Kanban boards, issue tracking.

If you're ready to change the way that you work online, give Monday.com a try by going to Monday.com/sedaily an get a free 14-day trial, and you will also get 10% off if you use the discount code SEDAILY.

Monday.com received a Webby award for productivity app of the year, and that's because many teams have used Monday.com to become productive. Companies like WeWork, and Philips and Wix.com. Try out Monday.com today by going to Monday.com/sedaily.

Thank you to Monday.com for being a sponsor of Software Engineering Daily

[INTERVIEW]

[00:03:10] JM: Alvin Richards, welcome to Software Engineering Daily.

[00:03:12] AR: Thank you for inviting me along today.

[00:03:15] JM: We're going to talk about Redis today, and in order to talk about Redis, we need to talk about caching, because Redis is most commonly used at least among the people I talk to for caching. Why do applications need caching?

[00:03:32] AR: That's a great question. You're absolutely correct. Most people think of Redis as a cache, and that's how they first come across it. The reason they're looking for a cache is the round trip cost of accessing their data and they're materializing that data back into their service or into their web application. Typically there's a big round trip to go to your traditional database. That data may not be in memory, so it has to go to disk. Therefore, you have a very long latency in order to be able to service the information that is required for your code to do what it's doing.

[00:04:12] JM: Web development is often built around objects. So if you think about object-oriented programming, you have users, you have comments, you have shopping cart data. If

we're building around objects, we want our cache to think in terms of objects as well. Describe how an object cache is used in modern application development.

[00:04:35] AR: It kind of comes down to how you think about the objects in your application code. If it's in Java, you probably got a hash, or in Python, you've got a dict. You've got some notion of some organization of data around the business object or the domain object you're manipulating, whether that is a shopping cart, or comments, or something else.

Typically, if you get in this from a relational database, you may have to do a query that joins lots of data together, or if you're going to a JSON store, like Mongo, you may be able to pull this out of a single JSON document. You go through this process of getting into database into a form that your code is now manipulating.

The way that you'd use a cache is to take that data in the way that you're manipulating it and either store that in a cache-like Redis as a binary blob or you could use any of the other constructs that Redis offers, for example, hashes, lists, sets, sorted sets in order to be able to manipulate that data inside the cache rather than having the round trip to the database.

[00:05:48] JM: An object cache is in-memory, which is faster than disk, but it's often more expensive or it's volatile. That's why we don't just do everything in memory. So how do we decide what we want in memory in our cache versus the things that we want on disk?

[00:06:10] AR: It's a good question. I mean, if I take a step back, the bigger question is what is memory? Some of the advances that have occurred recently, and if you look at, for example, Intel's Optane Persistent DC Memory, essentially you've got a form factor that looks like DRAM. It's got the speed and access time of DRAM, but if you turn off the power and turn the power back on, the data is still there.

The nature of memory being dynamic is going to be changing and evolving over the next 5, 10 years. We could be moving away from a paradigm that a data in DRAM is dynamic and ephemeral to data in memory is now persistent. Now, that doesn't necessarily mean it's reliable, because if you lose the whole computer, then you still lost everything. So you need to have

multiple copies or replicas of the data. But what it does mean is the way that you think of memory is going to change.

Now, you asked the question how do you choose what data should be in cache and what should be on disk. Well, ultimately, everything needs to be on a persistence store. We use disk as the term for persistence store, but as we just said, memory can now be persistent. What you really want from an object cache is the ability to evict out the least frequently used data under some policy. The data that you're accessing most frequently is in the cache, and in the case where the cache is not there, you still have the persistent copy sitting in your traditional database or storage system.

[00:07:52] JM: If I'm writing a web application, there's this problem of needing to access my data in the fastest fashion possible, but I often don't know what is the fastest route to my data, because sometimes it's in a cache. Sometimes it's on disk. Sometimes it might be in a different cache. We have the notion of the cache hierarchy, where you have different speed of access times with different caches.

If my data is not in a cache, then I'm going to want to disk and I'm going to want to get it from a database. As a programmer, do I have to write logic to defer to disk in the cases where I am not going to hit in the cache or does the cache take care of going to the database and figuring out whether to go to disk?

[00:08:52] AR: I think the simple answer is ultimately as the application developer, you don't want to and you shouldn't need to care. What you want is the best access to that data with the least amount of latency. There's a class developers where there are frameworks like Spring that will encapsulate the logic of I'll go to the cache, and if it's not in the cache, I'll go to the database and materialize the data that way.

There's a class of frameworks that alleviate the application developer from that responsibility. Now, with great power comes great responsibility. You kind of have to know some of the fundamentals of what's going on. At the other end of the spectrum, there are many classes of applications where you absolutely care about the minutia of the millisecond in order to execute the business case. For example, I am bidding on a Google AdWord, I've got about 75

milliseconds in order to place my bid. Therefore, the direct control of the cache and what you get from where is a very key design points in those use cases.

I think the case is that the developer needs to be aware of the tradeoffs they're making in terms of simplicity of their code versus the ability to directly control an outcome. But ultimately at the end of the day, the simplest form of using a cache is it's transparent to you.

[00:10:29] JM: Many developers who are working with JavaScript base systems are using MongoDB as their database layer. How would you contrast the usage and the design of Redis with a database like MongoDB?

[00:10:50] AR: MongoDB is very well-suited to languages where you want to ultimately materialize that data in a JSON form, and you can see from Java or Python how you can take some of the native data structures and represent that in JSON. So MongoDB provides a great way to store JSON data in a reliable way. It has certain scalability features and it has a powerful query language. These are all great attributes of a modern database.

Ultimately, the architecture of MongoDB is it has the notion that data is stored on disk. If you request that data, it first has to be shuffled into memory. Then it can be manipulated in memory. If that data has been changed, it then has to be shuffled back to disk. Ultimately, that all takes time.

What we have seen in terms of the industry, we've gone from sort of user's dictating behavior and scale to machines dictating speed and scale. As a consequence, it means that even with a database with MongoDB with its scaling capabilities, it's still not able to surface those request for data in those very high-volumes or low latency that's required for certain classes of applications. That's why cache CacheLight, Redis is very prominent and still used in those applications even if you're using modern classes of databases because of the speed, the scale and the performance that Redis can augment those systems.

[00:12:39] JM: Redis is probably the most widely used object cache, at least in the conversations that I have with web developers. But it was not the first object cache. It's not the only object cache. How does the design of Redis differ from other object caches?

[00:13:02] AR: I mean, I think fundamentally, the design of any system has to take into account the people who will use it. One of the reasons for the popularity of Redis is it provides a series of data structures that are familiar to any developer; hashes, lists, sets, sorted sets. The developers are already in that mindset. So when they come to use Redis, they're go, "Look, I've already got this object in a hash. Therefore, I can transpose it into Redis in a very, very simple way and I'm now manipulating that data structure on the Redis side using commands that are already familiar to me because I'm already using those constructs."

I think the popularity is that it fits the developer's operational model of how data should work and how data can be manipulated. So it's a natural fit. If I wind back to the days of relational databases and object relational mapping tools, how you manipulated your data in the code was orders of magnitude different from how it was stored and manipulated within a database, which caused this sort of impedance mismatch.

The advantage for developers using Redis is it's very natural for them. That's true I think for all of the popular modern technologies, is that they've taken the developers as the key persona and tried to surface the developer's needs in order to help them accelerate, solve the problems that they're trying to solve.

[00:14:44] JM: I'd like to talk through the architecture of Redis because it's quite a good example of modern distributed systems. Redis can run in a variety of distributed architectures. So you can run Redis as a single node. It's not like one of these database systems where you necessarily need to be doing some kind of three-phase commit, or two-phase commit because it's a cache. You are planning on – In many setups, you're planning on having on cache misses on occasion. You're planning on needing to go to the backup database layer to access your data under some conditions.

It's interesting, because you can have Redis setups where you have a single node. You can have setups where you have two nodes. You could have setups where you have a main Redis layer and then a backup layer of redundancy for high-availability. I'd like to talk through some of these different options. Let's start with just the single node option.

What are the use cases where I would just want a single node of Redis?

[00:15:57] AR: That's a good question. Ultimately, Redis is a very, very simple architecture, and this has allowed it to be used in very small caching use cases, like a single node as well as cases where you've got tens or hundreds of nodes. Ultimately, architecture of Redis is a single processing and control thread. So people say Redis is single-threaded. Well, it's easy to think of it as single-threaded. There are other threads doing other activities, but from the consumption of requests coming in, each one is processed in a single atomic fashion.

Now, the key design point of Redis is anyone of those operations has got a well-defined time complexity. So it's known beforehand how many essentially CPU cycles it's going to take to execute. So this allows for a very sophisticated system to be built with a very dependable and reliable latency. As people will know, if your operation is going to be O1 in time complexity, it means it's a very predictable operation. When you've got ON or OLogin, then it's going to be a function of your data.

In a very simple form, Redis provides a very simple way to have a cache that allows you to support the reads and write operations that you need in a very predictable way. So that allows people to have a great deal of certainty about how that system will behave not only when they run it in development, but when they run it in production as well.

[SPONSOR MESSAGE]

[00:17:53] JM: As a programmer, you think an object. With MongoDB, so does your database. MongoDB is the most popular document-based database built for modern application developers and the cloud area. Millions of developers use MongoDB to power the world's most innovative products and services, from crypto currency, to online gaming, IoT and more. Try Mongo DB today with Atlas, the global cloud database service that runs on AWS, Azure and Google Cloud. Configure, deploy and connect to your database in just a few minutes. Check it out at mongodb.com/atlas. That's mongodb.com/atlas.

Thank you to MongoDB for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:18:49] JM: When I think about an object storage system where I'm requesting single objects, I kind of think of just a hash map with just simple put and get operations. But I can imagine there are plenty of scenarios where I would want to get multiple objects out of a cache. Tell me about the read and write paths for doing a single entity read or write and for multiple entity reads and writes and if we're just talking about the single node of Redis.

[00:19:29] AR: Yeah. So it will come down to a number of factors, and those factors will include things like what is the data structure you want to manipulate? What is the size of the data? For example, reading a 10-byte string is going to be a fairly simple operation to execute. Essentially, you take the key, you look up that key's location, you go read the hundred bytes. You could do multiple key reads in a single command to get all of those together. So there's going to be a cumulative effect of getting all of those key values together before it's returned to the client.

But then each one of the data structures that Redis supports is going to have a different complexity associated with it. For example, in a hash, if I go and get all of the field values, that's going to be the different cost than, say, just getting one or two values out of that hash. That's why a lot of the time we focus on the optimization of each one of those commands and each one of those data structures to ensure that we have the minimum amount of CPU used in order to execute that operation.

The story doesn't change too much when we get to multiple nodes and we can talk about that a little later in the conversation, because essentially the way that a Redis cluster looks at the world is just a slice of that data and you're manipulating a key and you're changing the values of that key. As a consequence, you are going to one of the ten nodes in order to do that. You don't have to go to all of the nodes in order to find out where that key exists.

[00:21:13] JM: Why would I want to build a Redis system with multiple nodes? Why wouldn't I just want to use a single node for everything?

[00:21:19] AR: I mean, multiple nodes are a way in any distributed system to scale out. At some point you're going to hit some system constraint. Either you are utilizing all the CPU available for

that particular process or you are out of addressable DRAM. The reason for using multiple Redis processes is to explore modern day computing architectures. As we know, Intel is slowly increasing the performance of each core, but what they're really doing is giving you lots and lots more cores per processor.

As a consequence, you need a software architecture that can truly exploit all of the cores. Anybody who's built a multithreaded system will know that there are all sorts of comprises that you end up having to make with schedulers and so on and so forth and it's very difficult to fully utilize 24, 48, 96 cores.

The beauty of the Redis architecture is that for the most part, you're using a core and a core and a half and you can fully exploit that. So running multiple Redis processes allows you to fully exploit that computing architecture. Therefore, getting the most value out of it.

What you're trying to do is ensure that you can use all the addressable RAM that's available to you and use all the addressable CPU that's available to you. The way you do that in a Redis architecture is by running many, many, many Redis processes.

[00:22:55] JM: Redis will need to be both sharded and replicated in its most distributed fashion. Could you explain those two terms, sharding and replication and describe how they fit into a distributed Redis architecture.

[00:23:13] AR: Absolutely. I mean, sharding and replication solve two different related problems. Sharding is the ability to essentially take a dataset. Think of an address book where you're going from A to Z. Sharding is just taking a horizontal partition of that data and then splitting it between processes. A to M may be on one processor, N to P on another, Q to Z on a third. So you can take the logical horizontal partition of that data and separate it out.

The architecture of Redis means that you can keep on subdividing those partitions or those sharding ranges so that you get finer, finer granularity of ranges. Therefore, more processes that are running, and therefore the ability to exploit more cores. Sharding is a way to scale out to utilize more computing power. That could be CPUs. It could be addressable DRAM.

Replication on the other hand is a technique to ensure that multiple copies of the data exist. Why do you need multiple copies of that data? Well, imagine it's Black Friday and you got your shopping carts and you're about to check out and that computing node where your cached data is being stored has an outage. Either because it's lost its network connectivity, or the process died, or the machine rebooted, or the machine just burned and died because of the workload of Black Friday. Replication means that multiple copies of that data exist, which means that if one copy disappears, then a second copy is still available.

Going back to our Black Friday example, while if that first node fails where the data is present, then the second copy of the data exists. To the user, they get redirected to that second copy of the data. For their user experience, they may see sort of a slight slowdown, but they hadn't lost the contents of their shopping cart. For that user experience, that's absolutely key, because there's nothing worse than trying to check out your shopping basket goes empty. I'm sure we've all lived through that at various times of our lives where your lovely shopping basket gets emptied out for you.

So you can take that analogy and put it to any use case where even for a cache, you need reliability of that data. That data needs to be in multiple locations within a single data center or even in multiple data centers so that you can survive various business failures and still be able to surface that data through the application or that service correctly and maintain that business continuity.

[00:26:13] JM: Now, many people who are listening to this podcast episode probably have worked at a company where there was significant caching infrastructure. These people have probably also worked on their own side project where there may or may not have been caching in place. I think it's fairly common to experience two sides of the caching coin, which is I've got a very simple caching setup, and the other side of the coin where I've got a fairly complex caching setup.

I think what most people actually do not see is the process of building out and scaling up that caching infrastructure. The actual process of going through the sharding and replication and dealing with the pain points of having caching infrastructure that is present, but not sufficient because a company is scaling up.

Tell me about the process of scaling out a Redis cluster? What are the demands that you're going to be seeing where you know I'm going to need to start scaling this thing? What is the process by which I actually technically scale out my Redis cluster?

[00:27:41] AR: That's an interesting question and the answer is a bit complicated, because how you do this in open source Redis is different from how you do it in Redis Enterprise. Now, Redis open source has got all the ability to do the sharding and the scale out. it requires a considerable amount of operator effort in order to do that.

Inside the architecture of Redis Enterprise, the architecture is fundamentally different to allow the seamless scaling out without needing any interruption from the application of the client. The way we do this architecturally is at the outside rim of a Redis Enterprise cluster, something that we call the proxy. That's essentially what the client or the application code connects to.

Inside that cluster, the proxy understands the current sharding topology where the data lies. It acts as like a router to route that requests that came through the client to the right part of the cluster to execute. What this means is as more shards are added or more compute capacity is added, behind-the-scenes, the data can be resharded and redistributed around that cluster. But the client itself doesn't actually know the location of that date. So the data can be moved transparently around as you scale out.

This is not the case in open source. In open source, you actually do need to know the locality of that data in order to be addressing the right part of the cluster. There are differences between what you can get in pure open source as supposed to what you get in Redis Enterprise.

Typically, for a larger organization, even if you start off with a single process but you know you have a requirement to scale out eventually, then it's simpler to start off with a solution like Redis Enterprise, because as a said, you can add in these additional notes without requiring any downtime or any changes in your application code. That is possible with open source. It just requires a little more planning in advance and being very rigorous with your code and how you talk about the data and its location.

[00:30:11] JM: With cache technology, you often hear the term cache eviction, and an eviction is where you would check if there is – Well, let's say you make a read and there's something that's not in the cache. So you have to go to the database. You go to the database and you find what you are looking for, and then oftentimes you're going to add that thing that you just looked up in the database to the cash. When you do this, you sometimes find that, "Oh!" Your cache is actually full, and you decide to evict data from the cache in order to make room for this piece of data that you just looked up, because many times you look up a piece of a data, you decide, "Okay. Probably this data is going to be looked up again soon. I should add it to the cache and I'm getting evict some data from my cash in order to make room for this new piece of data."

Why wouldn't we just always allocate enough memory for the caching that we need and then just evict data from the cache as we run out of space in the cache? Why do we need to scale up the amount of memory that is available to the cache?

[00:31:33] AR: I mean, the reason for wanting to scale out is essentially to increase the size of your cache data. So let's take a simple example that you're running in the cloud and you've provisioned a particular node from your private cloud vendor in this 14 gig of the DRAM. Well, let's say your applications is growing over the time and you now need 28 gig or data. Well, if your dataset size is now 28 gig, then with your one node you can only have 14 gig. So you will be constantly evicting data. We'll talk about kind of eviction policy in a second. The ability to scale out is now more of that data is is now in memory.

What are potential eviction policies? Well, in a product like Redis, there are several built-in eviction policies that will just manage data out on essentially at least frequently used basis. But also from a developer's perspective – So let's bear in mind that the developers have a responsibility here, is they'll have some understanding of their use case of how long the data needs to be retained in memory.

The developer can decide how long that key should live in memory. It could be 10 seconds. It could be a minute. It could be an hour. They get to define for that particular key that they're manipulating what that policy should be.

Thirdly, in Redis Enterprise, there is technology called Redis on Flash. Essentially what this does is it combines DRAM with flash, or SSD, or NVME tries to keep the most frequently used data in DRAM and the least frequently data and the flash drive or SSDs so that you can essentially reduce the cost overhead of storing everything in DRAM with the knowledge that if you are going to SSD, then you're going to have a slower latency for that first access to that object.

There are several strategies you can use, but ultimately if you want to scale out and have more data with that low latency access, then you need some way to aggregate computing resources together. That's where the technics of sharding come into place, and that's the motivation for running multiple Redis processes across multiple compute nodes.

[00:34:06] JM: So let's say my awesome website is suddenly slammed with traffic, and in order to accommodate that growing traffic, I decide to scale up my caching infrastructure on-the-fly. I want to grow my Redis cluster. When Redis scales up and it has to do a re-sharding process, is that going to degrade performance? What does Redis do in a middle of a resharding and a scale up operation?

[00:34:41] AR: That's a really good question. The answer is kind of involved, but let me try and talk you through that. So let's say you've run out of compute capacity. What you do is you a provision a new piece. It's got more memory. It's got some cores. The first thing is that's introduced into the cluster. What Redis Enterprise will do, it will say, "Great! There is more compute resources available."

Now, when I start resharding, I can take a key range and then split that into two. That means I can now decide which parts of those key ranges can then be migrated to other compute servers in order to spread that workload out. As that is happening, obviously what we'll need to do is you'll need to copy that data from node A to node B. As you're doing that, there's going to be some network consumed doing that and there's going to be CPU cycles consumed doing that. It's very low impactful because each Redis process inside Redis Enterprise is limited to 25 gig of data. There's a very small amount of data that needs to be moved.

As we've talked about earlier with the proxy that's at the front of the cluster, at the point that data has been moved and both sides agree that they have a coherent copy of that data, it then requires just a simple change in the proxy to now reroute those requests to the new location of the data. It's a very low impact operation for the cluster to automatically reshuffle the data to take advantage of the new compute capacity that's been made available.

[00:36:33] JM: When we talk about distributed systems and we talk about scalability in a modern context, it's common to discuss Kubernetes. Kubernetes is pretty useful for managing distributed systems infrastructure. Does Kubernetes make it easier to – For those who don't know, Kubernetes is a container orchestration system for managing your different containers. Essentially, managing your different servers. Does Kubernetes make it easier to manage a distributed Redis cluster?

[00:37:13] AR: I think Kubernetes is one of the many ways to orchestrate. If I take a step back, we've got customers who will take our software raw, deploy it raw either from VMs or on bare metal and will manage that cluster on their own behalf. That means that they take on the responsibility for provisioning an orchestration all the failing modes themselves. You could take on an orchestration framework like Pivotal Cloud Foundry, or OpenShift, or Kubernetes, then those frameworks supplement not just the provisioning but also the maintenance of those clusters.

For example, on Kubernetes, we build in to our operator the ability to do auto recovery. So if you lose containers as part of that deployment, then we can automatically do the restart and re-sync and reestablish the state in the quorum of the cluster automatically. I think those frameworks greatly reduce the administrative burden of dealing with a distributed system and allow for much more autonomous deployment.

That doesn't mean that you have eliminated that role of the operator. The operator is still key in terms of capacity planning and understanding the hotspots and optimizing the deployment, but the day-to-day keeping the lights on can be automated to a greater extent than was possible in other orchestration frameworks. I think the simplification of these big distributed systems is the advantage that these orchestration frameworks provide.

[00:39:07] JM: The company that you work at is Redis Labs, and you make products and support around Redis. Redis is open source. What do people buy from you?

[00:39:22] AR: Redis is a great open source project that's been going since about 2010. The original motivation of the founders of Redis Labs was to build a managed cloud service that was enterprise grade for Redis. So that included things like failover, automatic scale out through sharding, GO replication so you can replicate across data centers.

As a company, we started off as a managed cloud service. As our business have evolved, we carry on selling a managed cloud service on all the major cloud vendors, but we also sell the raw Redis Enterprise software for our large customers who may be have record tree or compliance needs to be able to run this in-house rather than on a public cloud.

But our experience of running Redis as a cloud service has allowed us to bake in and optimize a whole bunch of things into that codebase to allow any organization to run this at scale successfully through our Redis Enterprise product. What are people buying? They're either buying a managed service for Redis or they're buying Redis Enterprise that they can deploy and deploy on-premise within their own data centers.

[SPONSOR MESSAGE]

[00:40:53] JM: Modern applications are built on top of cloud platforms, and open source software, and APIs. As our applications become higher level, we can manage bigger systems with fewer people. We can easily create different environments for AB tests and continuous delivery pipelines. We can manage our software with configuration files rather than imperative logic. But as we have more environments and we have more configuration, we have application sprawl. Our configuration files can drift out of date with changes in dependencies, and licenses, and standards.

Atomist is a platform for better, safer software delivery. Atomist helps you understand what is going on across your application giving you a single pane of glass that helps you get a complete picture of the state of your different environments and configuration systems. Atomist can help you with the inevitable application drift, and you can find out more about how to avoid

application drift by going to softwareengineeringdaily.com/atomist. You can get a free drift report and figure out how Atomist can help you solve the problems of application drift.

Can you easily identify which ports your Docker containers expose? Do you have an accurate account of how many versions of core technologies like TypeScript, or Spring Boot you are using? Do you know how many of your applications still use Java 6? Go to softwareengineeringdaily.com/atomist to learn about Atomist and how to avoid application drift.

Atomist can also help with compliance, CICD, dependency management and many more parts of your application. Go to softwareengineeringdaily.com/atomist to learn more. Thanks to Atomist for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:42:55] JM: When people come to you, when they come to Redis Labs and they say, “Okay, my Redis is slow or it's having lots of cache misses. I don't understand why.” What's the most common thing that they're doing wrong?

[00:43:12] AR: Typically, I mean for most people, the deployment of Redis is very simple. It works out of the box as is. The challenge will become of like any tool is how are you using it? This goes back to some of the things we talked about earlier, which is time complexity.

For example, if I'm navigating a list of a billion elements, that's an expensive operation. So if I'm trying to navigate a billion element lists thousands of times a second, that's going to be expensive for any technology to perform. Typically, the combination of issues that we most normally see is obviously there’s not enough DRAM to support the dataset. As a consequence, data is being, and therefore data is being evicted and constantly reloaded. That's a fairly simple thing to solve.

The more complex situations to solve are how the system is being used to solve the business case. Fortunately, Redis has been out long enough that there are lots of good well-defined patents and blogs and documentation that says, “Look, if you want to do a leaderboard, here’s a

couple of strategies for doing a leaderboard. Here are a couple of strategies for doing rate limiting," and so on and so forth.

For the larger development community out there, there are blueprints that they can use and be successful with out-of-the-box. But there's also a set of tooling. We recently launched a product called Redis Insight, which is a free tool to use for developers and DevOps. What it will do is it will show you pictorially inside your Redis cluster and what data is there and how it's organized, but it will also show you which keys are hot, which keys are being accessed the most? Which operations are the slowest? It'll start giving you the ammunition to actually understand and how to optimize the system yourself by looking at some of these key indicators.

[00:45:23] JM: If I was looking at the indicators from the Insights, the Redis Insights, how could I translate that information into something actionable? What would be a common thing to look for that I could say, "Okay. Now it's time to – I don't know, scale up or like have fewer write operations."

[00:45:45] AR: Yeah. I mean, how I would approach it is typically where I start is looking at the slow log. The slow log will tell you all of the operations that are taking a long time. Now, bear in mind as we said earlier, every Redis process is essentially single-threaded, in that a request will come in Redis will execute that from beginning to end, return the results before it now accepts the next command to process. If you've got a command that's taking 400 milliseconds, that means anything that's queued up behind it has got to wait 400 milliseconds before it can execute.

The first question is should that command be taking 400 milliseconds? Is it the right command to be executed? If the answer to that is yes, then one strategy is to scale out so that you've got the data distributed across more Redis processes, and so therefore you got fewer bottlenecks.

For the most part, when you start looking at the slow log, people realize that there is a process or a threat that's running and executing a command that's either inefficient or doesn't need to be executed at the frequency it's been run at. That helps people to then look at their architecture and their implementation to make sure that they're running the right commands on the right data structures with the right time complexity. That's one way of looking at it.

The other way of looking at it is hot keys, right? You may have a particular counter or a list that lots of process are constantly either reading or making changes to. You then have a bottleneck around a particular key or a small set of keys out of the billions you may be storing. Again, this may come back to an implementation or an architectural choice that you've made in that development process.

Both the slow logs and the hot keys are a great first indicator of the next questions you should be asking yourself and the operators and the development team associated with that. How you then solve that may be a combination of scaling out. It may be a combination of making some data structure changes or some implementation changes in order to better improve the parallelization of your application.

[00:48:17] JM: Are there any newer patterns or use cases for Redis the you've seen recently?

[00:48:25] AR: Yeah, I think that the primary shift has been – I mean, traditionally, if you ask anybody what is Redis, they'll say it's an in-memory cache. The shift that we started to see is people using Redis as the primary database. That's not just use cases. That's a sort of paradigm shift of I use this class of technology to solve problem X, to now solving a very different class of problems. Some of that is also what we've been doing to extend Redis. Traditionally, Redis has had a simple set of data structures like lists, strings, sorted sets.

What we've been doing at Rdis Labs is to extend those data structures to incorporate things at full text search, time series data, graph data, JSON data, and not further extends the set of data structures and therefore the use cases where Redis gets used. So you'll see Redis being used in gaming for things like leader boards, for people who are doing APIs to do rate limiting. In finance that can be things like credit scoring and risk analysis. In telcos, that could be knowing user provisioning limits when services are being requested.

The general theme is Redis gets used when you're either dealing with a high-volume of data that's changing quickly or you need to have very low levels of latency that are very, very consistent to be able to meet the operating demands of those use cases.

[00:50:06] JM: It's quite interesting. I can imagine, if you need to model Redis as something like a graph database, a graph database has edges. So if you would need to model Redis as a graph database, you're going to need to store additional information that gives you the necessary metadata to make graph database traversal. I would say the same thing for a full text search. What kinds of systems do you need to build to augment a Redis system with something like full text search.

[00:50:46] AR: Good question. What we found overtime is people were using native data structures, the list sets and hashes in order to solve some of these domain problems. So what we did research and time series in graph is to essentially encapsulate those needs into fundamental structures within inside Redis so that you didn't have to combine multiple data structures in order to achieve your goal. The Redis data structure, that should do that on your behalf.

Full-text search is a great one where there are certain applications where you need to traverse a lot of data very quickly with all of the features that you need in a full text search query. You need start words, you need antonyms, you need all of this other kind of great stuff, but you also need to look at a lot of data quickly in a very low latency.

What we've done is combined the needs of each one of those data structures and optimize it in the Redis way to get low latency. For example, graph is a great example of using the graph BLAST library in order to store sparse matrices and then use a form of matrix manipulation in order to be able to do those traversals of those nodes and edges.

We've combined the kind of needs for those data structures along with the characteristics of Redis in order to, I think, provide a very unique way to process that information in a way that makes sense to those use cases.

[00:52:30] JM: I want to begin to wrap up. You mentioned hardware earlier on. One thing that I think is interesting about hardware is if we could reduce the cost of memory or if we could improve the reliability of in-memory systems, we could have much faster infrastructure. I mean, in some sense, this is an inevitability, because I mean it continues to happen. It's one of these Moore's law-like trends where you just see improvements in memory infrastructure.

Can you give me a status check on impending hardware changes that could push us to rethink how we do caching and other in-memory operations?

[00:53:21] AR: I mean, let's do a thought experiment. How does the program change if you have infinite CPU calls or infinite amounts of memory that's persistent? I think that's kind of the path that we're going down. We're going to be seeing a single computing node with a thousand 24 cores. You're going to see with in tens of terabytes of DRAM. With persistent memory, you already can get 12 terabytes of persistent memory in a server. I kind of think the way that we're going is everything is ultimately going to be in memory and that memory for the most part is going to appear to be persistent.

Now, you need much more than just persistence. You need replications to be able to deal with ultimately the durability of date. You have to survive various modes of failure from single compute servers, or racks, or data centers so that you got many survivable copies of that data.

Ultimately, you need a software architecture that can exploit that hardware footprint. I think that's kind of why Redis is uniquely situated, which is it's got an architecture that's already optimized just to deal with memory. The fact that that memory is now persistent is an advantage, because it doesn't have to shuffle data between disk and memory in order to take advantage of it. You can already take advantage of cores.

By running lots and lots of Redis processes, you can exploit those computing architectures. The fact that memory is now persistent means that you don't have to shuffle data between these different tiers in order to actually effectively manipulate it. I think, ultimately, that's where the compute landscape is taking us.

I think if you've got an architecture that can't exploit those inherent changes in a computing landscape, then you'll always end up with something that's inefficient or wasteful of resources or simply can't exploit those hardware that's available to it.

There are many great examples of how we have done that in the past. If you think about virtual machines, it's all about how do I drive better utilization? I think Redis is a great solution for

driving that utilization for not just your caching needs for data, but also feel persistent needs of data.

[00:55:58] JM: Last question. What can the field of computer science learn from the field of photography?

[00:56:04] AR: The great thing about photography has been around considerably longer than computer science. It has evolved in bursts. If you think about where we started with plate cameras, to modern day hundred-megapixel cameras, it shows that the rate of change can accelerate rapidly over time. I think what we can assume from computing is there's going to be orders of magnitude improvement in performance and capacity and the ability to represent the world around us in a compute architecture as we have seen in plate cameras to digital photography.

I think the real question is have we built the right frameworks and the processes and the technology that we can build and manipulate that not just in a reliable and robust way, but also in an ethical way and in a robust and reasonable way that we don't penalize particular communities or particular geo locations based on our ability to look at data more efficiently and therefore calls fragmentation in each one of those communities and those geographies.

I think, ultimately, we have great tools and great capacities at our hands that are being orders of magnitude better down the road. Are we responsible enough to be able to use that in a reasonable and ethically viable way?

[00:57:48] JM: Alvin Richards, thank you for coming on Software Engineering Daily. It's been a pleasure.

[00:57:51] AR: Thank you so much for the invite.

[END OF INTERVIEW]

[00:58:02] JM: I never liked searching for a job. It's painful. Engineers don't want to make a sacrifice of their time to do phone screens, and whiteboard problems and take-home projects.

Everyone knows that software hiring is not perfect, but what's the alternative? Triplebyte is the alternative.

Triplebyte is a platform for finding a great software job faster. Triplebyte works with more than 400 tech companies including Dropbox, Adobe, Coursera and Cruise. If you've been hearing about Triplebyte for a while, you will be happy to know that Triplebyte just launched a brand-new machine learning track and they'll now be helping machine learning engineers find jobs in the same way that they've already helped generalist, and frontend, and mobile engineers. It's amazing seeing Triplebyte expand into these specific verticals, because they're so efficient about matching high-quality engineers to great jobs.

Go to triplebyte.com/sedaily to find out more about how Triplebyte works. You can take a quiz to get started, and if you end up taking a job with Triplebyte, you get an additional $1,000 signing bonus because you'll use the link triplebyte.com/sedaily.

If you make it through that quiz, you get interviewed by Ttriplebyte and you get to go straight to multiple onsite interviews. Its economies of scale for software engineering interviews is pretty sweet to see that centralized in a place that gives you those economies of scale. I'm a fan of Triplebyte. I hope it gets bigger and bigger and creates more and more economies of scale in the miserable hiring process of getting a job as a software engineer. Make that painful process a little bit better with triplebyte.com/sedaily.

Thank you Triplebyte for being a sponsor.

[END]