Distributed Systems & Big Data

NoSQL databases are increasingly used in real-time web applications and big data. Learn about distributed systems, issues, and models, as well as consistent hashing, big data, and analytics.

Table of Contents

Distributed Systems

In this video, we're going to take a look at distributed data systems. What are they and what are the various models that allow us to implement a distributed database system? Distributed systems. This is a term you hear all the time these days, particularly in conversations about databases. Obviously, this means that the data is distributed. But let's look a little more deeply at exactly what that means and, more importantly, how do we accomplish it? First of all, any system where your data is distributed in any fashion…perhaps you have a couple of servers that are redundant across your network. Well then, you have a distributed system. Now, more often, when we're discussing distributed systems, we're thinking about something on a little grander scale – perhaps a network cluster, or even a cloud. So, essentially, if I take my data and I distribute it across two or more servers, I have a distributed system. But this begs the question of how do I distribute that data? Then how do I access it? There're primarily four approaches to doing this. There is sharding. There's the master slave. There's peer to peer, and then there's some combination of these approaches. In this video, we're going to take a brief look at each of these, so you'll have a general understanding of how they work.
Heading: Basic Types.

The basic types of distributed systems are sharding, master slave, peer to peer, and combined.

Let's start with sharding. Another term for sharding is horizontal partitioning. A database shard is literally a horizontal partition of data. So, let's assume that I have a terabyte of data and I chop it into four 250 GB subsections. Each of those sections is considered a shard. Now, with sharding, I'll put each shard on a separate node or separate nodes, and each of those shards does its own reads and writes. Now, you should keep in mind that for the issue of availability – making sure that one node going down doesn't lose data – normally, what happens is a shard is duplicated on a couple of nodes. So, I have these four 250 GB shards. Instead of simply putting them on four nodes, I might put them on eight nodes or 12 nodes.

And, in that case, I have some duplication of the data. And I would have maybe a grouping of nodes that contain a particular shard or duplicates of a particular shard. Now, one problem with this is, there can be some latency when you're doing your query. Now, if my query only needs to access one of those shards, then it's going to go very, very fast. However, if I have a complicated query that has to look at more than one shard, then I'm going to have to go to multiple nodes to get that data. The query will still work, but things are going to be a little slower, and my response time will be a little slower.
Heading: Sharding.

Sharding is the use of a horizontal partition with separate read and write nodes. It results in some latency.

The master slave paradigm is another way of doing distributed data. The data's replicated across multiple nodes – as many as you feel are necessary – and each of those nodes is a duplicate. So, let's go back to our fictitious terabyte of data. I might have five nodes, each of which has a complete copy of the terabyte of data. However, I will have one node that is designated as the primary – the master node. This is where all the others get their copies of the data. So, normally, if you're going to do any sort of update – you're going to insert, you're going to alter a record, you're going to delete a record – that has to be done on the master or primary. And then the master or primary will take that data and replicate it to the other subordinate nodes. Now, any user can execute as many queries as they want on any subordinate node. I think, as you can clearly see, a master slave situation is very scalable for reads. I have a bunch of queries happening. I can just keep adding on subordinate nodes and spread that workload out across as many as I need to. However, it doesn't help much with the scalability of writes. No matter how many subordinate nodes I have – whether it be five or 500 – I have a single master that's responsible for all the updates, all the writes, all the deletes, and that can be an issue. So, normally, master slave is a pretty good approach if you know you're going to have a whole lot more queries than you are modifications.
Heading: Master Slave.

In a master-slave system, data replication happens across multiple nodes. The master node is used to update data while other nodes are used for queries. This type of system provides high scalability for reads but less scalability for writes.

Another option is simply to do peer to peer. Let's go back to our hypothetical terabyte of data. I simply copy that entire terabyte to as many nodes as I wish – let's say five. Each and every one of these nodes could accept a write, an update, a delete, or a read. So, usually, when a user is interacting with the system, they will automatically be directed to the nearest node, wherein they will do any modification, update, query, whatever they want to do. Now, this is fantastic for load balancing, and it equally load balances the writes as well as the reads. But consistency becomes an issue and that should be fairly obvious. If I make an update to one of those nodes and, just a second or so later, you're doing a query for the same data on a different node, it's very likely you're not going to see my changes. That can lead to stale data. So, peer to peer is a very good solution when you're okay with a little bit of stale data.
Heading: Peer to Peer.

On a peer-to-peer system, data replication involves multiple nodes and no master. The nodes accept reads and writes. The system supports good load balancing but not consistency.

Now, as you can see, each one of these three solutions has some very good strong points and a few issues…a few weak points. So, what people often do, is some sort of combination. Maybe I have sharding, but I designate a single master to handle all the updates. Or perhaps I have peer to peer, but these are peer to peer clusters and each cluster handles a shard. Any sort of combination where I take the best of both worlds and put it together is a combined approach. And, particularly in the world of NoSQL, you're going to see each of the three major ones we discussed – sharding, master slave, peer to peer. And you're going to see some relatively customized combinations, where people have mixed and matched the pieces they felt were most appropriate to solve their problem.

Distributed Issues

In this video, we're going to discuss issues with distributed systems. First and foremost, we need to consider the characteristics of the data. Obviously, a distributed system is meant to be used when you have large datasets. But the typical usage pattern can even include huge files – sometimes, although not commonly, even hundreds of gigabytes for a single file or a single dataset. The data is rarely updated in-place. Now, what that means is that, regardless of the model you're using – perhaps you're using sharding, or peer to peer, or master slave…it doesn't matter – data's updated on one node, and then propagated to the other nodes. So, the data is not necessarily consistent across all nodes. This leads to an issue with any distributed system – the issue being consistency – and you have to work with it. In any distributed system, reads – read operations, retrievals or queries – are very, very common. And so are appending. This leaves the characteristics of the servers – often called chunk servers. Any given file is split into contiguous chunks. Some references indicate that a typical chunk could be 16-64 MB but, to be honest, in modern times the chunk could be almost any size.
Heading: Data Characteristics.

A distributed system is meant to be used with large files and data sets. Data updates are rare and occurs through replication and propagation. Common actions include reads and appends.

Heading: Server Characteristics.

Data is split into contiguous data chunks. These can be any size. They're replicated multiple times and are physically separate.

The main issue to keep in mind is, regardless of the size of the chunk, you're going to have it replicated at least two times, but more would be preferable. Again, think about the various models of master slave and peer to peer and sharding. Well, we don't want to ever have a situation where a single node is the only repository for a chunk of data. So we're going to replicate it and, usually, that's at least two times, but even more is better. Because

of that replication, we'd like those data chunks to be physically separate on different servers, and those servers in different geographical areas. Now the purpose for setting up a distributed system, regardless of the issues that you may have to overcome, is really twofold – the first being availability. It's an unfortunate fact that bad things happen. First of all on a technical level, servers fail, hard drives crash. On a more human or physical level, disasters happen – fires occur, floods, tornadoes, earthquakes, hurricanes. Any of these things can render a particular server inaccessible.
Heading: Purposes.

The purposes of using a distributed system are availability and load balancing.

Now, in our global economy, where most organizations span multiple geographical areas, it's really unacceptable to have a disaster – no matter how large or how small, whether it be a simple hard-drive crash or something terrible like a hurricane – but a disaster in one region to render the users in another region unable to work, unable to interact with the data. Well, that's just unacceptable today. So, one of the purposes…one of the primary purposes of any distributed system is to provide availability. We have the data replicated into multiple geographical areas that are remote from each other. That way – whether it be a hard-drive crash, a natural disaster, or any other sort of interruption to availability – it's almost impossible for all of the nodes to be unavailable. Therefore, the users in one area can still access data even if one particular region is offline.

The second issue is load balancing. Load balancing simply means that we've distributed the load across multiple servers, so no one single server bears the brunt of dealing with all the different users. This allows us more robust responses. If we have a 100,000 users' mail all hitting a single server – no matter how powerful that server is, no matter how robust and how much bandwidth the connection has – that's a serious burden. But, if we distribute those users across say, 20 servers, then they're going to get a much more responsive environment. So, remember, regardless of the issues to overcome, the two main purposes of distributed systems are availability and load balancing.

Distribution Models

Let's discuss the various models of distribution the distributed systems can use. Obviously, one model for data is a single server. A single server is clearly not a distributed system, but that's the baseline to start talking about. Then we have replication. I can simply replicate a single server and have two, three, four other servers. This would be a traditional database cluster. However, when I'm replicating, I have to consider how the data is replicated. And that includes the various models of sharding, master slave, and peer to peer. In case you don't recall, sharding is when I take the data and divide it into chunks and different chunks are replicated out. Master slave is when individual node oversees subordinate nodes and all writes go to the master – all reads come from the slaves. And peer to peer is a situation where reach node is totally equivalent. Now, the obvious problem with a single server is there's simply no redundancy and no availability. I say there's weak scalability, and you may think there's no scalability at all. You can always add more RAM, more hard drive, beef up the server, and scale up a little – but there's some significant limits to what you can

do. So, that's why we say it has weak scalability; not necessarily no scalability.
Heading: Models.

Models include single-server models and models that use replication, including sharding, master slave, and peer-to-peer models.

Heading: Problems with Single Server.

The problems with a single-server mode include issues associated with availability, redundancy, and scalability.

Problems with distributing data. The first and most important is consistency. It doesn't matter which of the models I use – whether it be sharding, master slave, or peer to peer – it's impossible that all loads would be consistent at all points in time. There's going to be some point in time where some update, some change, some modification has not yet been distributed to all the relevant nodes. That leads to a consistency issue and it leads to what's called stale data. You're looking at data that may not reflect the most current changes. This is acceptable…it's workable, but you have to be aware of it. Then, when you're creating your data-store solution, you have to keep consistency in mind and take some steps to deal with that issue. What specific steps you take are going to depend on your specific situation, your needs, and the particular solution you're implementing. Two other issues are read and write issues. How are you going to handle reads? Which node will handle the read? Will all nodes be able to handle reads? Same thing happens with the write. Are you going to have a single node handle the writes – such as in a master slave, or would you prefer to have all nodes handle the writes? Each of these issues and each of these possible solutions has strong and weak points. You have to pick the one that's most appropriate for your situation.
Heading: Problems with Distributing.

Problems with distributed systems include issues associated with consistency and with reads and writes.

Consistent Hashing

Consistent hashing – it's a very important topic in the world of distributed databases. It basically gives us a method, a reliable method, of being able to find where the data is in all those various nodes. Let's begin with, what is consistent hashing? Well, frankly speaking, it's the answer to a number of questions. The first and most obvious question is how do we map data to nodes? We know that, regardless of the model you're using – master slave or sharding or peer to peer, that you have data distributed across a number of nodes. But we have to be able to find that data, so we have to know which node has what data. If the data is distributed geographically – in other words, it's not just on several modes, but those nodes are dispersed in disparate geographic areas – well, how's that going to affect mapping? What happens if a node's offline? Now, one of the reasons for using distributed solutions is, just in case a node is offline, data is still accessible from other nodes. So, we're accounting for this possibility. We know nodes go offline. How does that affect mapping? If you simply statically mapped a particular user to a particular node, that wouldn't work

well if that node went offline.
Heading: What is it?

Problems to consider include mapping data to nodes, the fact that data is distributed, and offline nodes. Hashing uses the equation p = hash (k) mod n.

The usual answer to these three questions is to use a process called hashing. Now, if you've taken a basic course in discrete mathematics, you're very familiar with the hash. If you haven't, the first thing to point out is this should not be confused with the concept of cryptographic hashes like MD5 and SHA-1. It's a totally different thing. In this case, a hash is a way of basically mapping values and keys. You map a database object – we'll call it o – to some partition, p, on a network node. Now, let's stop right there. By database object, we're being kind of generic. It could be a table, it could be a record, an aggregate, a document in a document store… any number of things. I just simply want to know: where is it? Not just what node is it on, but on what partition?

So, I hash that object's primary key, which we'll call k, to the set of n available nodes. Well, what do I mean by primary key? Obviously, in the relational database world, that has a specific meaning. It's an identifier that uniquely identifies a record in a table. But here, we mean it more generally – any sort of unique identifier that would let us know which database object we're talking about. Is simple hashing sufficient? Well, it can be and it has been used for a long time, but consistent hashing is basically…think of it as hashing plus. It's an improvement on hashing that was designed specifically for data partitioning. For example, simple hashing might not work in a setting where you have nodes frequently joining and leaving the cluster at runtime. The hash is constantly changing. You have a node that's here; now it's gone. You have a node that wasn't here; now it's here. That constant change can be a problem with simple hashing.
Heading: Is there a better way?

Consistent hashing uses the data partitioning technique. K divided by N keys need to be remapped.

So, consistent hashing is a special kind of hashing where only K/N keys need to be remapped. In other words, K, being the number of keys, divided by N, a certain number of nodes, needs to be remapped. Put another way – without going into the deep mathematics, which aren't really necessary if you understand NoSQL – consistent hashing basically says, if a node joins or leaves the cluster, I only have to remap a couple of the keys…some of the keys, not all of them. So, if we associate each node with one or more hash value intervals, where the interval boundaries are determined by calculating the outer-reach node and the identifier. Now, think about that for a second. Essentially, instead of hashing a node to a specific key – a node identifier, I've got an interval which shows me a range of nodes and I'm going to hash that value to that range. In other words, my key isn't tied to a node – it's tied to a range of nodes. And, if one particular node is removed, the interval is taken over by a node in the adjacent interval.
Heading: What does this accomplish?

Consistent hashing affects hash value intervals. If a node is removed, an adjacent node assumes the value for the interval.

Big Data

Big data, this is a very important concept in the NoSQL world. NoSQL was essentially designed to handle big data. It's a term that gets used all over in the IT industry but most people are a little bit vague on exactly what it means. So what is big data? Well it's sort of a generic term for any collection of data sets that gets so big it becomes difficult to process that data using traditional data processing methods. Now, some sources will say larger than a single terabyte, some will say larger than 10 terabytes. And it's certainly a given that as computing power improves and our traditional data processing applications improve is that threshold for what we consider big data is going to get larger. The bottom line is if you have very, very large collections of data in your normal data processing applications, those tried-and-true applications that have worked for years simply aren't effective, then you probably have big data. Now why is this important for a conversation about NoSQL? Well simply speaking, NoSQL was designed to handle big data. You might wonder how common is this or put another way, how big is data getting? As early as 2008, Google was processing 20,000 terabytes each and every day. I want you to let that sink in because that was several years ago and it's undoubtable that they have increased their volume since then.
Heading: Basic Concepts.

Big data refers to large data sets or collections of data sets, which may be greater than 10 terabytes. Big data is difficult to process.

Heading: Examples.

Google processes 20,000 terabytes a day. Wayback Machine contains 30,000 terabytes of data. Facebook has 30,000 terabytes of user data. Other examples of institutions that handle big data are eBay, Amazon, and the US government.

The Wayback Machine or archive.org had 30,000 terabytes of data and Facebook has over 30,000 terabytes of user data. Let's stop and think about that. We had indicated previously that to a lot of people over 10 terabytes or over a single terabyte is considered big data. We're talking about situations where there's 30,000 terabytes of data or even 20,000 terabytes per day. I think under meaning…any meaningful definition of big data, these method meet the definition. There's other places where you see big data, if you think about it, eBay and Amazon, various government agencies, IRS and others, huge, huge amounts of data. Okay, is it simply the fact that we have a lot of data? Well, to some extent, there's actually four characteristics we look at in big data. The first is volume, how much data are we talking about? But the next is variety, the data can be very, very variable. You could have lots of different types of data in a big data situation. Think about eBay as an example or Google. Google's an even better example, all sorts of searches, all sorts of queries, all sorts of ads, very different parameters for each one of these, that's definitely some significant variety of the data.
Heading: Characteristics.

The characteristics to consider are volume, variety, velocity, and complexity.

Velocity's another issue, how fast are we getting new data? Well you couldn't reasonably get to situations of 20 and 30,000 terabytes of data, unless that data was being generated very quickly. If you think about…maybe you want to track every single sale on Amazon, well that's going to go up by tens of thousands per day. What if you wanted to track every single interaction that every taxpayer had with the IRS, well we have well over a 100 million taxpayers in this country, each one interacting on some level with the IRS on a fairly regular basis. That's a lot of data being generated very, very fast. Last, we look at complexity. The sheer size, variety, and philosophy of data is going to mean it's probably… at least has the potential to get very complex, there's all kinds of nuances. When I mentioned the IRS as an example, obviously each individual tax return in and of itself can be quite complex. Imagine a 100 or 150 million of those tax returns and then throw in corporate tax returns and nonprofit, all sorts of other types, that is some pretty complicated data.

So when we're looking at big data, we're asking is there a lot of data, is the data varied, is new generated…data being generated quite quickly, and is the data possibly very complex? Why is this important? Why do we care about big data? But first of all, E-commerce has grown astronomically since its introduction in 1995. Nowadays, the volume of transactions going on in E-commerce situation are just staggering. There are probably millions of transactions occurring every single day and they're all varied, the data is increasing rapidly, kind of complex data too. This cries out for a big data solution. What about research? The most obvious scientific study that comes to my mind would be the Human Genome Project, trying to map all those different genes and all those chromosomes using samples from lots and lots of different people, that's a lot of data. And of course data analysis, when we're looking for patterns in the data, we're trying to understand the data. Not just have an amorphous blob of information but have meaningful information coming out of that data, information we can act on. Well we need data analysis techniques for that. This is the essence of what big data is.
Heading: Why is it important.

Big data is important for e-commerce, scientific research, data analysis, and social media.

Big Data Analytics

In this video we're going to talk about big data analytics. Big data is a term that gets thrown around all the time and depending on what source you read, it's anything from over a terabyte, over 10 terabytes in data. Suffice it to say, that it's a lot of data. Data usually requires analysis so there's analytic techniques specifically for big data analytics. Two major categories are ROLAP and MOLAP. ROLAP, or Relational On-Line Analytical Processing is a data processing or data analysis processing technique really meant for relational databases. Multi-Dimensional On-Line Analytical Processing is designed for databases with multiple dimensions, in other words, with maybe a little different sort of structure. If you think about it for just a moment, that's a very good description of a NoSQL

database. One of the purposes for data analytics is data mining. Now data mining can be discussed at great length. There are entire books written on the topic, graduate courses over the topic, but at its essence, data mining is the attempt to derive answers you did not expressly ask for. In other words, normal database queries, whether you're querying a relational database or a NoSQL database, normal queries ask a specific question and get a specific answer.

Heading: ROLAP versus MOLAP.

ROLAP stands for Relational On-Line Analytical Processing. MOLAP stands for Multi-dimensional On-Line Analytical Processing.

Heading: Data Mining.

Data mining reveals useful, unexpected patterns in data. It involves a non-trivial extraction of implicit information. Data mining is exploration and analysis of large quantities of data by automatic or semi-automatic means.

Data mining is about looking for patterns and deriving answers to questions you haven't thought to ask. Discovery of useful and often unexpected patterns in the data is really the cornerstone of data mining and that issue of possibly unexpected relates back to what I was just saying about answering questions you might not have thought to ask. It also involves the non-trivial extraction – let's stop right there, non-trivial. Data's only important if there's something we can do with it. Trivial data elements that aren't actionable are of no use in data mining. So we want non-trivial extraction of implicit information, that's the information that's in the data but might not be standing out explicitly and obvious. It might be implied via patterns of the data or search algorithms. Data mining is about exploring and analyzing the data using some automated process to look through tons of data, and explore and analyze and find out what's going on, what's there.

At its heart, you could also say data mining is simply pattern discovery. One of my favorite examples of data mining involves a record producing company that decided to execute a data mining program versus their data warehouse. Much to their surprise, they discovered a large number of senior citizens purchasing rap music. It took a little while to get to the bottom of this. At first, they thought the data mining was flawed but it turns out these were grandparents purchasing this music for grandchildren for holidays. This was unexpected data, it was a pattern that no one would've thought to ask for. No one would've queried the database and said, how many people over 60 purchase rap music? Data mining provided that information and that's just one of many examples. Now data mining is split into some tasks, some different types of data mining. It's not the case that you do all of these. Each of these is an aspect of data mining.

Heading: Data Mining Tasks.

Data mining tasks can be predictive or descriptive. Predictive tasks include classification, regression, deviation detection, and collaborative filtering. Descriptive tasks include clustering, association rule discovery, and sequential pattern discovery.

First, let's talk about classification. Can we classify the data into various groupings? This

gives us predictive value because once I've classified data, then I can put new data into a particular class and make some predictions based on the class it's in. Now classification and clustering are usually closely related because clustering is all about bringing together related data into a cluster, which then facilitates classification. Association rule discovery is a descriptive technique where we're trying to find out what rules govern the association of one piece of data to another. We can see some data appears in the cluster, like senior citizens purchasing rap music. What's the rule that associates those two data points: rap music purchase and senior citizen? Well the rule would be grandchildren of a particular age, probably between the ages of 12 and 20. Sequential pattern discovery is another way to do descriptive analysis and data mining. I'm trying to look through a sequence of data and see what pattern emerges.

Now regression analysis and deviation detection, these are more highly advanced statistical tools and it requires a fairly deep knowledge of statistics to get in depth on those but essentially we're applying statistical analysis to a large body of data, trying to get information. And a similar thing could be said about the collaborative filter, trying to see how various things collaborate to produce a particular piece of data. Then NoSQL is ideally suited for big data analytics. NoSQL, at its heart, is all about queries with inserts and updates being the less common. Well because of that, it's perfect for doing analysis where you're just going to be pulling data, you're not going to be putting more data in there. Plus, big data in analytics implies really big data. Well NoSQL scales really well to large data sources so it's a really good solution for big data analytics.
Heading: NoSQL and Big Data Analytics.

NoSQL is better structured for big data analytics because its emphasis is on queries rather than updates and it scales well to large data sources.