Scalability Patterns

Best Practices for Designing High Volume Websites

—

Design your website to handle a million hits a day

—

Chander Dhall

Apress®

Scalability Patterns

Best Practices for Designing High Volume Websites

Chander Dhall

Apress®

Scalability Patterns: Best Practices for Designing High Volume Websites

Chander Dhall
Austin, Texas, USA

ISBN-13 (pbk): 978-1-4842-1074-1 ISBN-13 (electronic): 978-1-4842-1073-4
https://doi.org/10.1007/978-1-4842-1073-4

Library of Congress Control Number: 2018951608

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Todd Green
Development Editor: Laura Berendson
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484210741. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

*In loving memory of my parents to whom
I owe endless gratitude.*

Table of Contents

About the Author

Chander Dhall, CEO of Cazton, is an internationally acclaimed software architect and consultant. He is an eight-time awarded Microsoft Most Valuable Professional, Google Developer Expert, Azure Advisor, Cosmos DB Insider, ASP.NET Insider, Web API Advisor, and the Dev Chair for DevConnections. He speaks at top tech conferences around the world like Microsoft Tech Ed, Microsoft Visual Studio Launch Event, MVP Mix, DevConnections, NDC, Techorama, Build Stuff, and many others. He's a professional software architect, trainer, open source contributor, community leader, and organizer with years of experience in enterprise software development.

Chander Dhall is a world-renowned technology leader in architecting and implementing solutions. He has not only rescued software development teams, but also implemented successful projects under tight deadlines and difficult business constraints. His company has a proven track record of not just saving clients millions of dollars, but also providing an expedited delivery time. Chander's team of experts speak in top technical conferences in the world.

As an advisor and insider to top technologies and frameworks, Chander has insight into new releases on technologies used by millions of developers. His critical advice, impeccable vision, futuristic strategy backed with creating and establishing best practices in the industry are some of the reasons for his unparalleled success.

At the same time, he's a voracious and highly respected speaker. Chander is known to elucidate critical and complex concepts while making them easy to understand. He's one man who can talk to anyone from a junior developer, a senior architect, all the way to an executive and knows how to speak their language. He has utilized his amazing business sense having started companies with zero funding and turned them into multimillion-dollar companies in less than two years.

About the Technical Reviewer

Fabio Claudio Ferracchiati is a senior consultant and a senior analyst/developer using Microsoft technologies. He works for BluArancio (`www.bluarancio.com`). He is a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microsoft Certified Professional, and a prolific author and technical reviewer. Over the past 10 years, he's written articles for Italian and international magazines and coauthored more than ten books on a variety of computer topics.

Preface

Over the course of my professional career, my team and I have rescued countless projects that were either running out of budget or were behind schedule. At the same time, we have created projects or augmented teams to help clients start projects from scratch and take them all the way to completion. The success of a project does not solely depend on hiring a couple of rock star architects or developers and assuming everything will fall into place. Success needs to be in the DNA of the team. Good software leaders inculcate the successful qualities in each and every member of the team.

This book is the result of a presentation I did for Microsoft Tech Ed in Barcelona, Spain, in 2014. While the concepts are not new, I believe when it comes to the scalability of a web-based system and scaling your database system, these principles are timeless.

When we talk about scaling our systems, microservices is considered as a panacea to all ills; however, there are many systems that may not need a complete microservices model. Whereas microservices brings agility to our systems, we may not have the need to deploy four times a day. We may not have systems that are written in different languages and frameworks that need to be part of the system we are building.

Even though this book talks about microservices, it's just one of the patterns that we will explore. Understanding scalability patterns and what's good, bad, and ugly about all of those is the real aim of the book. Having worked with start-ups, mid-size corporations, and Fortune 500 companies with global presence, I been fortunate to see what works and what fails. Patterns change all the time, but principles are fundamental to the system that runs on them. If you were to use an RDBMS like Sql Server that is better for scaling up than scaling out, normalization would take precedence over duplication of the data. However, if you are using a No-Sql database like MongoDB that scales out easily, then duplication might take precedence over normalizing in certain cases where availability is important.

With the ever-increasing technologies and frameworks, enterprise architects and developers struggle with getting tied into legacy tech, which makes it harder for them to take advantage of better technologies that could have solved many of the problems they are facing. However, it's not just the technologies or the tech stack. More importantly, it's also the patterns and principles that are being used to architect the system. It's very easy to lose focus on the big picture, especially in big organizations where it's easy for teams to be in siloes and become myopic about they are doing. While working with different companies, my team and I keep an eye on similar mistakes in approaches and strategies and try to disseminate the learnings to other people on the team and eventually to teams working at other clients. It's reasonable to make mistakes; however, the goal is to learn from these mistakes.

This book focuses on different patterns of scalability. The goal of the book is for you to understand the strengths and limitations of these patterns. Understanding of the underlying principles is the key to architecting, developing, deploying, and then maintaining a great system that is futuristic, easy, and less costly to maintain and has the best ROI. The book is not opinionated, and the goal is not to force opinions. It's a critical analysis of all the current practices I have witnessed in the last decade. The real goal is to expose and encourage a scientific way of thinking that is principle based. The principles change per technology. Functional programming languages can't be expected to be written with the same principles as object-oriented languages. Patterns are problem specific. We cannot use a Strategy Pattern to solve the same problem as a Factory Pattern.

Software development is an ever-evolving field. Change is the only constant we know. The key takeaway from the book is to understand the mistakes that have already been made and to learn from them. No solutions suggested in this book should be considered absolute or final. I'm sure we will continue to see newer problems with each passing day, and the great tech community will continue to solve them with continued dedication. Knowledge, experience, agility, and continuous improvement are key components to making projects, teams, and companies successful.

I wish I could put all of that in a book, but as you know a book is just a way to express some of your opinions. With this book, I wish to spread ideas and hope to establish a relationship with the readers. In the social media world, it's easy to connect and I hope you will reach out. I value your opinions, criticism, and suggestions. I had a great time writing this book. I hope you enjoy reading it.

CHAPTER 1

Introduction

In this world of ever-increasing data usage, scalability has become more of a necessity than a luxury. This can very well be attributed to the tremendous growth in the size of smartphones and tablet users. According to Internet Live Stats (`www.InternetLiveStats.com`), we have about 3,915,062,366 users (as of May 9, 2018). With the world population being roughly 7.6 billion users, this number will only continue to grow. With the advent of mobile devices and the constant dependency on these, there is a big need of a "**Scalability First**" approach or "**Cloud Oriented Architecture**."

Scaling Up has traditionally been a much simpler architecture than Scaling Out. Previously, a high-powered server with Relational Databases was enough to power Enterprise Applications and websites. Data transfer and interaction between the user and the websites were fairly limited. Nevertheless, social networks are more of a game changer in that the data generated is enormous. Just for the sake of comparison, buying a plane ticket online versus commenting incessantly on Facebook are two entirely different business cases. Whereas the former requires the bare minimum input from the user, the latter has hardly any limits. Whereas the former requires robustness, security, and is not heavy on data or interaction, a typical RDBMS might be a good solution for it. Accuracy and validity of the transaction end up being more important for such a transaction over speed. Whereas in the case of social networks, performance and scale of the website are far more important than being able to successfully save a post created three years ago. In comparison, losing the order history of plane tickets sold three years ago might be an unforgettable nightmare for an airline.

In recent years, businesses have been trying their best to drive their customers to mobile apps. There are apps for almost anything a consumer would want to do: ranging from buying a cup of coffee to monitoring his heart rate. The boom in Web followed by the boom in mobile has been instrumental in making it difficult for a typical Relational Database to be able to scale to the limits needed. Again, scaling is also very relative. If you own a coffee shop, you only need to be able to scale up to the amount of real orders you can handle in a day. If the infrastructure of your coffee shop only allows you to sell 1,000 cups of coffee a day, there is no point in creating a website that can scale to receiving 100 million orders in a day.

Note Scalability is a science as well as an art. Technical architects need to work in tandem with the business in order to better understand its scalability needs.

Scaling – An Art and a Science

Scaling is an art and a science. During the last recession, I was brought onto a project that had been going on for four years and had more than 300 bugs. The two consulting companies that had been working on it had blamed it on the clarity of requirements and practicality of timelines. Not even one major module out of four was production ready. And guess what? This was a perfect project to work on and prove the importance of scalability.

Background

This was a company that was highly profitable and had a manager who did extremely well with Excel sheets and a high volume of low-cost, low-skilled people. They could do everything from tracking a package, to invoicing, and making sure the customers were happy. However, if the CEO of the company wanted to find out where a $2MM machine was located, and the only person who knew had taken the day off, there was no real way of finding out. Automation was the need of the hour.

It turns out that both the companies had brought in a bunch of smart developers and architects who somehow either didn't understand the business or chose to ignore it. A quick glance at the code base proved that the code was not uniform. Some of the code was very well written; however, there were islands of repetitive code base, unit testing barely existed and even worse, integration testing was not fully automated.

Now let's look at this database. The code base had 2,500 stored procedures. Most of these were 100–1,000 lines of code, and one stored procedure was copied about six times on average and the last couple of lines were changed. Just imagine changing the common code in 10 such stored procedures. It would mean changing 60 files at the bare minimum and the testing effort behind it would be massive. The code had a data layer, but it was not consistent. The absence of a good Object Relational Mapper was one of the major problems. An ORM is not an end-all and be-all for a data layer. Nevertheless, it makes sense to have an ORM as part of your data layer. Most of the standard queries run equally well with an ORM. There are surely cases where ORMs may decrease performance, and so using a Stored Procedure in tandem is an approach that cannot fail. ORMs make it easy to catch errors at compile time, reduce development effort, and decrease the chances of bugs in production. A Stored Procedure could be great for performance and should be used in such situations where performance affects the business. At the same time, altering a production database is usually not a good idea, but with Stored Procedures that's sometimes the only way to go.

The UI was pretty standard. It was Web and Windows at the time and later, I assume, other UIs (User Interfaces) would be added. However, there was no use of an API. Dynamic Link Libraries were the only way to share reusable code. This was not considered that bad in those ages, but it could have been better. Windows UI code is pretty standard and there is not much room to maneuver for a developer, so we won't go much into that here. The Web UI code was overly complicated and nonstandard. Simple things like the presence of a few different kinds of grids in the same application were simply not needed. For the consistency of the website, ensuring speed of development and reusability, a partial control on a page needed to be standardized across the entire enterprise. It may not have been a good idea to have multiple CSS for a standard button with the same size. Let's say there is a basic grid that is used multiple times in the same app, like a basic grid with just some regular sorting on column names. Assume there are three such grids and all of them are built using three different libraries. Now for some reason, the business wants to add filtering on each grid. This will increase the work threefold. What if two different kind of grids are used on the same page? That means additional JavaScript and CSS files on the page. Whereas, by standardizing a grid across the application, and adding custom CSS and JavaScript where needed, we could have alleviated both of these problems.

Without going into too much detail about the actual project needs, I would like to talk about only the requirements that are relevant to this course. Looking at the solution, it was quite clear that the developers were brought in using Agile, which was a great starting point at that time, but the solution showed signs of missing the complete picture. The code was broken into modules and the typical mistake of assuming all modules are mutually exclusive was committed. That was apparent given the pockets of mock code and data that were present. This was required to fill in pieces of puzzle that were not accounted for when the current module was designed. For example, the Shipment Module was designed and implemented without even defining the API of how the system was to handle payments. So, when the Payments Team came up with an API, it broke everything in Shipment. All in all, the design was exceptionally good in some areas, but the complete system was quite dysfunctional. That is the basis of my previous comment about smart developers. There was surely a lack of a good hands-on architect on the team, which would explain why the project was in such a terrible state. There were five major modules and only one of them had actually been done with more than 300 known bugs. So, after four years of development, nothing was actually working.

A quick look at the architecture suggested that the system configuration, topology, servers that were asked for in production environment, and the amount of resources requested were enough to run a company that gets more than 100 million unique hits a day. Later in a meeting with the CEO, I was amazed to find out that the company had only 140 part-time employees who would ever use this system. It was like creating a rocket ship to go to the grocery store. The recruiter who had hired me was confident in my abilities to complete this project and had gotten approval to have me there for an extended period of time. I was told by internal employees of the company that it would take four more years for this project to be delivered and they didn't expect it to work well. The morale was surely low.

After a few hours of my assessment, my next meeting was with the CEO. It's important to remember one thing about executives: they all want to know when the project will be done and how much it is going to cost. You basically have only 10 minutes to prove your worth. At least, that's what I had heard.… The CEO was very impressive and extremely good at what he did, although, he had not been involved directly in managing the technical projects. His disdain for technical people and lack of trust in them was being hidden by his great leadership skills. In his situation, this was not surprising at all. Ten minutes into the meeting, the big question came up. Before answering, I had to be fully informed of the business's needs. After an enlightening conversation of the CEO's vision, I was confident that I could finish the entire project by myself in 9–12 months. This number also had the added buffer I needed in case something external jeopardized my plan. If you have ever bid for projects with multiple vendors, you surely know this: the lowest and the highest bids usually disqualify themselves. I almost disqualified myself here. He called my recruiter right after the meeting and was dismayed and also bewildered by my estimate. After a bit of back and forth, we realized he didn't have trust in my estimate because it was so much lower than expected. The only way to establish trust now was to have my recruiter guarantee a smaller, more seemingly realistic result. So, the promise was made to deliver one major module in three months. Fortunately, it was delivered in nine weeks and the entire project was finished in eight months with no other resource added to the team.

Bottom line: Software Estimation is hard, and it becomes even harder when the scope is not defined or understood well. Usually, a lot of time is spent worrying about technical aspects, but not in understanding what the business really wants. Yes, scalability is truly a science. Let's say there are 100 million users on a website. Assuming the website doesn't make simultaneous ajax requests at any given instant, there cannot be more than a 100 million concurrent requests. Assuming we have a machine that can handle five million concurrent requests (and the system resources are utilized less than 70%), we know that we need 20 such machines to scale. But how realistic is this scenario? Statistics are mostly misleading. Daily, monthly and yearly analysis of the data can result in predictive analysis and the nature of business, clientele, location of users, time of the year, etc., can play a big part in understanding when to really scale to the highest level. Most of the time, it might be sufficient to handle the traffic with just two machines. Scaling with infinite resources is easy, but scaling with the most optimum use of resources and creating a sustainable solution that doesn't foreclose future options of growth is what we should be aiming for. This is where scalability really becomes an art.

Response Times: What Are We Aiming For?

Amazon claims that just an extra 1/10th of a second on its response times will cost 1% in sales. Google claims that half a second increase in latency caused its traffic to drop by one-fifth. Have you worked in real-time systems with internal SLAs (Service Level Agreements) of eight seconds? Quite surprisingly, there are web applications, APIs, and back-end processes with that kind of response time. Whereas, an offline system that doesn't need to have a real-time response can have extended SLAs. The longer the SLA of a real-time system, the more adversely it affects the business. The recent trend in the industry has been to decrease the response times and it's a welcome move. Note, if any organization hasn't tackled this until recently, it's high time that efforts are made to alleviate this problem.

So, what is the magic number for competitive advantage? How much faster does a website need to be in order to have it? It turns out that the magic number is 250 milliseconds. According to Harry Shum, a computer scientist and speed specialist at Microsoft, "Two hundred fifty milliseconds, either slower or faster, is close to the magic number now for competitive advantage on the Web."

REST Principles

When it comes to scalability, the best principles to follow are the REST principles. REST stands for REpresentational State Transfer, which is a term coined by Roy Fielding. REST promotes an architecture that is over HTTP, which happens to be a stateless protocol for web-based applications. The six principles to be discussed are the following:

1. Client–server: The principle states, *"Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered."* Even though this principle sounds pretty straightforward in today's world, keep in mind that this dissertation was written around the time when they were a bunch of monolithic apps that had high coupling. It was extremely hard for the client and server to not be dependent on each other.

In my honest opinion, Roy deserves credit for suggesting this principle way before smartphones became prevalent. This makes a lot of sense when we have multiple kinds of UIs, for example, thick clients apps on Windows, iOS, or Android platform along with web apps.

2. Stateless: The principle states, "The client–server communication is further constrained by no client context being stored on the server between requests." Since HTTP is stateless, it makes perfect sense for all client server interactions to be stateless. Being stateless means that the server will not store anything about the last HTTP request a client has made and will treat each and every request as a completely new request. What happens if the application needs to be stateful? There are many ways to handle that but the most scalable way would be to use a second-level cache like Redis, Memcached, App Fabric, etc.

3. Caching: The principle states, "Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance." Caching data that doesn't change very often like product catalogs and their description is always a good idea. It not only increases the response times of the application, it also reduces the number of calls to the database. In my personal experience at different clients, they either have no caching or a partial caching strategy. Caching strategies need to be at the core of any application architecture. One thing to stress though

is having a strategy about "what not to cache." Quite interestingly, this is as important as having a list of items to cache. Certain items should not be allowed to be cached at all. These include critical items that, if cached, or cached incorrectly might display a wrong version of the item to the user and may cause business errors that might result in loss of business and client confidence.

4. Layered system: The principle states, "A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches." Imagine building monolithic apps to serve your users over HTTP, be it an iPhone app, Android app, Windows app, or even a gaming console like Xbox or Samsung TV app. What if all of them had to write all the server logic separately? It makes sense to use an API. Once we have an API the client has no clue about the underlying back-end servers and can just conform to an authenticated and authorized series of request-response-based HTTP calls via a Web API. On a separate note, a lot of these apps could be built with the same technology like Xamarin or Cordova, a similar code base for most of the UI functionality, which is another improvement and works well for most of the applications.

5. Code on demand: The principle states, “Servers are able temporarily to extend or customize the functionality of a client by the transfer of executable code.” This makes perfect sense in situations where we need to exploit a different kind of capability that could be specific to an operating system and might not be that easy to develop on a particular UI, let’s say web browsers. Flash or Silverlight are good examples of code on demand. For DRM-related functionality, Silverlight had a huge advantage over HTML5. Both these technologies used an executable to add rich functionality to web applications.

6. Uniform Interface: The principle states, “The uniform interface between clients and servers, discussed below, simplifies and decouples the architecture, which enables each part to evolve independently.” Anytime we create an API, it should follow simple, uniform interface for accessing and manipulating resources. API consumers should be able to create a resource, when possible, and be able to access relative URIs to access related resources. HATEOAS (Hypermedia as the Engine of Application State) is a constraint of the REST application architecture. With the use of HATEOAS we create a hypermedia-driven site. This site simplifies navigation of the REST interfaces dynamically by including hypermedia links with the responses.

If you want more details on REST, please read Roy’s dissertation.

Concepts

There are certain concepts that are important to understand scalability well. They might be defined a little differently than expected, but there is a reason behind it. An attempt has been made to explain the reasoning behind the definitions below.

Scalability: It's usually described as the potential of a system to handle a certain amount of work and its ability to grow seamlessly as needed. Let's pay attention to some key details now. How about the potential of a network or even a process to scale or is it limited to just a system? What is meant by handling work? Is it just user traffic? Would a system be considered scalable if a sudden growth in traffic causes the system to be down momentarily? It's important to understand the significance of the word "seamless" in the definition. Of course, no system needs to be defined to be able to handle unlimited traffic as even the most scalable system in the world will be hit by a finite population; the population of the world plus the number of automated processes that are making requests to it. Predetermining the system's strength and its ability to scale seamlessly help plan the system well and go a long way in the success of the business.

In technical terms, scalability is defined as the potential of a system, network, or even a process to handle a certain number of simultaneous users, sessions, transactions, or operations and grow seamlessly as needed.

1. Scaling Up: Adding more system resources to the current system. This involves either upgrading parts of the system like processors, memory, hard disks, etc., or simply replacing the hardware with a more superior one. This is also referred to as Vertical Scaling.

2. Scaling Out: Scaling up is usually the easiest resolution, but can become a bottleneck with increased availability needs. The law of diminishing returns applies here. Adding more resources beyond

a certain threshold could cost a huge cost overhead for minimal gain in performance or even reliability. Also, a single point of failure may lead to a huge downtime impact when it's not backed up well and the Mean Time to Repair (see below) could be high. Apparently, a more recent trend involves addition of commodity-level hardware to an existing system. Adding additional nodes to a cluster in a back-end system is referred to Scaling Out. Each server is independent of the other server and the failure of one server, ideally, should not impact the availability of the entire system.

Reliability: A system that performs as expected for a specific period of time is known as a reliable system. During this specified period of time, reliability is defined as the concerned system's resistance to failure. A partial failure of a system should not lead to the complete failure of a reliable system. Below are the different measures to calculate reliability.

1. Mean Time Between Failures (MTBF): This is defined as the difference of Total Time Elapsed and Total Downtime divided by the number of failures. MTBF = (Total Time Elapsed - Total Downtime)/(Number of failures)
2. Mean Time to Repair (MTTR): This is defined as the average time taken to repair a failed component.

Performance: Performance is sometimes incorrectly defined as time taken per operation; however, performance really boils down to amount of work accomplished compared to the time and resources used. Good performance, hence, is nothing but the optimum utilization of all resources involved. Time taken per operation will be automatically reduced if the computation of the result involves the most optimum utilization of resources for that particular task. Nevertheless, a high-performing system may involve one or more of the following: good responsiveness, low resource utilization, high throughput and/or availability, short data transmission time, etc.

Responsiveness: Time taken per operation is known as responsiveness. In real life, it gets used as a measure of performance, but it's equally important to consider resource use while calculating performance.

Availability: A direct effect of scalability is availability. Availability generally refers to the ability of a user to access a system during the window of time in which the system is supposed to be accessible. In the web world, most systems are supposedly accessible 24/7, so the window of time may be a little redundant here. One big thing to keep in mind is responsiveness of the system. A system won't be considered available if the response time is overly delayed. For example, if an average response from a website takes less than one second, the system will be considered largely unavailable if the system takes, let's say, one minute to respond.

The following formula is used to calculate Availability:

Availability (%) = (Total Time Elapsed - Total Downtime)/(Total Time Elapsed)

Single points of failure: Imagine a website running on just one web server. For example, IIS, Apache, or Nginx. Now, imagine it has just one database. If the web server fails, but the database is working, the system will still be down. If the database fails and the web server is up, the system will be partially down. It will be able to serve static HTML files; however, it won't be able to get any dynamic data from the database. Assuming the system relies heavily on the data from the database for every single call, the system will be considered a failure. So, in the example above, both the Web Server and the Database are Single Points of Failures (SPOFs). SPOF is the part of the system that if it stops working, it will lead to the collapse of the entire system.

Fault Tolerance: Systems that don't collapse even after failures of certain individual components of a system are fault tolerant systems. Imagine a situation where there 10 web servers behind a load balancer.

Even if few of them are down, the traffic could still be balanced between the rest of them. A typical user might not even feel any difference. It could very well result in the loss of some temporary information though (like session information). There are techniques that will be discussed later to minimize even those situations.

Downtime Impact: It is the impact of the downtime of a server, service, or a resource. The impact could be measured in terms of the users affected, perceived business loss, loss of reputation, etc. For example, if an e-commerce website does a business of roughly $10 million on Thanksgiving and Christmas between 10 a.m. to 8 p.m., a downtime of two hours could be a loss of $2 million or more. Of course, the bad experience could also mean permanent loss of customers especially if the website was hacked and the customers now feel insecure trusting the platform. According to Dunn & Bradstreet, 59% of Fortune 500 companies experience a minimum of 1.6 hours of downtime per week (Reference: `http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html#sthash.JBOjDOVi.dpuf`).

Theory

There's a well-known joke about theorists: A theorist is "Someone who doesn't believe in anything that is working well in practice until and unless he could prove it out in theory." A similar joke exists about academicians: "An academician is a person who is willing to assume anything, but responsibility." Of course, I'm in no mood to put myself down or my community down; however, jokes or stereotypes may have some truth to them. As a problem solver, absolute satisfaction comes only when the problem is solved both in theory as well as practice. I must admit though, if the knowledge is sans practical experience, it could be a humbling experience trying to scale a system to more than a billion hits a day, especially a transactional system that is e-commerce based.

At the same time, the practical experience of making things work without fully understanding the theory behind it could be self-defeating, too. The goal is to create a sustainable solution that scales itself. Hence, the underlying theory backed with practical experience is the best combination.

Here’s a small example that explains the intent behind the discussion.

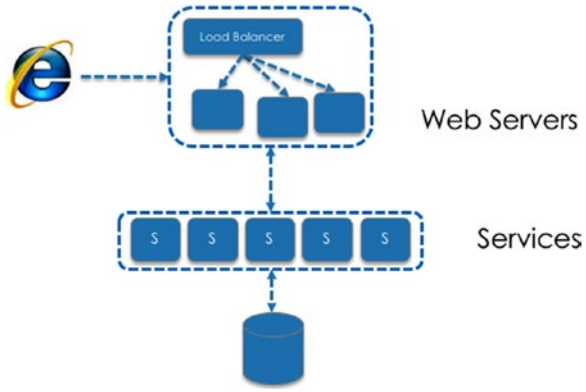


***Figure 1-1.** Scaling a web application using services (old approach)*

Ages ago, books were written that evangelized services to scale businesses’ applications. The solution was either promoted or interpreted as elucidated in Figure 1-1. When I read about it, I was easily convinced that this would be a good solution for some projects in some organizations, but it can’t be the end-all be-all of scaling almost any solution. Anyone with sustained success in architecting Enterprise Grade solutions could have seen the pitfalls of blindly following the approach on any projects under the sun. To my utter surprise, this approach was implemented on critical projects at almost every client I worked with for the next 8–10 years.

Well, it was a great opportunity to help clients as well as get the practical experience of scaling brownfield applications. Theoretically, it sounds really good. Here's the argument in favor of using the approach in Figure-1:

1. **Speed of development**: Since there are multiple services for all major business divisions, it makes sense to divide and conquer. Let's assume a company has multiple teams responsible for some kind of business functionality to handle payments, orders, inventory, products, etc. With this approach, they can have different services. They can define the APIs for these services, and any team looking to build on top of another team would just need the contract of the service to continue their development.

2. **Testability**: Every team can test their services independently. Assuming the team has the required domain knowledge, testing the service independently is an easier task. Any bugs found by the consumer of the services can be centrally reported to the team responsible for it.

3. **Performance and Scalability**: Each team is responsible for the performance and scalability of the service. They are expected to maintain a certain SLA.

4. **Fault Tolerance**: It's very easy to spawn multiple services doing the same task to make the solution fault tolerant. If the master service is down, replica services will still be handling requests and the system will be fault tolerant.

5. **Deployment**: The approach makes the deployment modular and easier. Redeploying a service does not mean redeploying the entire app.

6. **Access Management**: Sometimes, there are reasons to expose one business functionality quite differently than others. For example, if all the services need to remain internal to a company's network but a service like payments needs to be exposed to other B2B customers who want to leverage your payments engine behind the scenes, then payments, being a service, could now be exposed to the rest of the world (of course, with the right credentials). Any service related to e-commerce transactions that is responsible for sensitive customer data could very well be kept in a Tier 2 server (Tier 1 servers in this case refer to servers exposed via HTTP and Tier 2 is an internal server that can't be accessible unless via a Tier 1 server). Trust me, the list is actually much longer and compelling arguments could be made as to why this is a great architecture.

Whereas, this architecture makes sense for quite a few business cases, no architecture is such that it will be the best architecture for all the problems in the world. The success of a project really depends on the ability of an architect to understand the unique challenges of an organization, vast experience on all sorts of projects (small-scale, mid-scale, and large-scale), experience on multiple operating systems, environments, etc. (thick client apps, thin client apps, linux, windows, iOS, IOT-based systems), ability to understand and develop (hands-on coding ability) the entire framework (if needed), visionary capability, and the strength of the personal network of both technical and business people.

Nevertheless, I believe in failing fast. Every architecture has certain strengths, weaknesses, and limitations. It's always good to understand those before implementation. Let's brainstorm some potential problems with the architecture to see where it may fail.

1. **Increased out-of-process calls**: You might think this solution will provide separation of concerns and hence high cohesion and low coupling. That is correct; however, imagine a scenario where the call comes to the Order Service that needs to contact the Inventory Service to make sure that the product exists, then might need to make a call to the Catalog Service to get the current pricing and then contact the Cart Service to see if any kind of promotions can be applied to the particular order. This might mean the services having to call other services on which they are dependent.

   Of course, there are better ways of fixing this problem. In fact, the major reason behind the evolution of a services-oriented architecture to evolve in this way is the lack of understanding of the final model. Most cases, individual teams end up creating these services, hoping some day they'll magically fit together across the organization.

In Figure 1-2, we have the following dependencies:

1. S1 -> S2
2. S2 -> S1, S2 -> S3, S2 -> S5
3. S3 -> S4, S3 -> S5
4. S4 -> S5

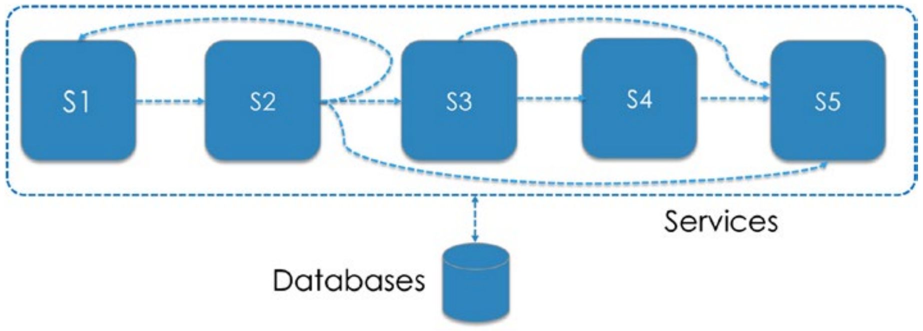


***Figure 1-2.** Services dependent on each other*

We can easily identify circular dependencies, and in a complex application, unless strict standards are defined, there is a lot of potential for failure especially in circumstances in which the need to scale is unexpected. Dependency between S1 and S2 is circular, and from a high-level diagram like this, is hard to identify recursive dependencies. Circular dependencies slow us down, but a recursive dependency leads to partial or complete failure and must be avoided in any instance.

2. **Performance**: Going back to our example of the checkout page. In order to check out an online shopping cart, we need a lot of information. In a monolithic web application (assuming there is no common API) with more of a Model View Controller architecture, an easy way to go about this request would be to make a call to the controller and then the controller makes a call to the database. Assuming that the entities in question are the Payment, User, Product, Order, and Inventory objects, we would really need to make one call to the Relational Database Management System and join the corresponding tables to get the result back to the web server.

Now, imagine doing this with what we have above? Imagine we have five different services, separately built and deployed by different teams, all corresponding to the tables. We shall be making five different out-of-process calls, which is surely more expensive than the one call to the database. What about the database join? For such an involved query I'm sure we will need to do some database joins. We may need customerId from the *Customer* table to reference the credit card information from the *Payments* table. In order to check the cart out, we will need a join between *Customer* and *Payments* table and may be more joins eventually. If that is the case, there is no other option than to do an in-memory join. Of course, an in-memory join is fast, but using the web server to do data joins is not at all a good strategy. A web server should really be concerned with I/O more than computation. A web server should be the least utilized in any computation behavior and should be able to respond as soon as possible to every incoming request. The more the usage of resources, the worse the scalability of the system. Another side effect of this approach might also mean bringing more data than is needed from the services, doing a join to get the common data we need for the request, and then sending back what was asked for. Now since the join is not happening at the level of the RDBMS, but at the level of the web server, we are forced to grab more data from the database than needed. That will also reduce performance.

3. **Single Points of Failure**: An application architecture cannot be considered to be fully scalable if it does not adhere to a simple principle of having no Single Points of Failure (SPFs). Testing and Debugging is really a process to deliberately try to fail the application in test environment so as to avoid any potential failure in production. When Microsoft was testing Azure, it was said that it tested for every possible scenario imaginable, even a meteor strike. It surely made sense with what they were trying to do, which was to create datacenters all over the world with geo-replication between continents. Any natural- or human-based cause for failure cannot be ignored.

   Imagine the first scenario where we had an MVC (Model-View-Controller) app interacting directly with the database. So there are only two points of failure that we are concerned about: one is the web server and the other one is the database. It's very easy to back up the database timely. We can also have a Master-Slave configuration with regular updates. At the same time, it's quite easy to horizontally scale the web server.

   In the second scenario, however, we have the web server, the database, and the services and hence, the number of SPFs is a lot higher. Does that mean that it is a bad idea? No, but it is something to consider. Based on the size of your app and its usage, it might not be a great idea to have increased SPFs. But in a full-blown enterprise app, it might be worthwhile to explore this route by making sure we have respective slaves for each and every service or SPF.

4. **Cost**: With the increase in services, the cost will surely increase. With every service added, it's quintessential to make sure that there is a secondary service that is either available all the time in a Master-Slave setting or is a backup service that starts functioning the moment the first service is down. The Master-Slave arrangement is a much more reliable solution and is more real time, while in the latter solution, turning on the secondary service and switching the app to it could take some time and the system could have a downtime. Downtime impact is different for different apps and different businesses and should always be avoided.

   Comments: Does every app need to scale? Consider a phone book app for your entire organization of 100 employees. Even if you grow to 10,000 users in two years (quite unusual), chances are it could be run using just one web server. Does it make sense to use SOA for it? Surely not. Is the data from the phone app needed in other apps? If yes, maybe just an API-based architecture might be sufficient. Very rarely there will be a case to create separate services from independent business logic within an app with limited business functionality. At the same time, in certain situations multiple services are the way to go and the costs will be reduced in the long term.

   On a separate note, the moment someone is following the industry best practices, most likely, they are already behind in today's world. It takes a while for something to become a best practice and it takes even longer for the best practice to reach the masses.

Of course, it's a great thing to follow best practices, but it's also important to understand that with the pace with which something becomes an established best practice, we might have already been behind the industry experts who have moved to a much better solution. For example, how much of SQL has changed in the last 10 years? Which principles of SQL have changed in the last 30 years? Not much has changed when it comes to the principles. So, when the DBAs feel like they have mastered the best practices of data, quite honestly, they've only mastered the best practices of SQL. And maybe the best solution for their application might be a No-SQL or some kind of polyglot persistence. Polyglot Persistence really is an industry-wide accepted term for a solution that comprises multiple kind of data stores, including but not limited to SQL databases, Document databases, Key-Value pairs, Big Table, etc.

Remember the industry-wide move to the Services-Oriented Architecture? and then later to Microservices? which some people are calling Nano-Services. Moving to this type of architecture because it fits the organization is a good move; however, incorporating the architecture just because it's a best practice may not always work. Additionally, a lack of understanding of the architecture may result in a faulty implementation or a variation of implementations may cause failure. It's not hard to have a general architecture that works for most business problems in a very optimum fashion, but it's very easy to overlook the intricacies of that architecture and end up with a not-so-great implementation.

CHAPTER 2

Scaling – An Art and a Science

No matter how well thought out, for an architecture that is created on a slide deck but not backed by a proof of concept or real world implementation failure, chances of failure are high. The bottom line is that a theoretical architecture created without much real-world experience can lead us to failure. A possible exception is an architecture that works well for a specific app or a particular company. But that does not mean we can use the same architecture elsewhere and it's guaranteed to work. Even that can fail for a different app and a different organization. Just because it has worked well in practice doesn't mean it will always work. A good example of this is an app created with some Single Point of Failure (SPFs). It's very possible that the organization never had an SPF that actually failed. This doesn't mean it's an architecture that will be successful all the time. Even theory doesn't guarantee everything 100%, which is the same in the case of practical implementations. They don't guarantee 100% success either. It is in our interest to use the best of both worlds. Both theory and practical knowledge are needed to create a world-class architecture. When Dhall Architecture (to be defined later) had to be implemented for the first time, failure was not an option. Quite fortunately, it succeeded. Later, I tried this on bigger environments and bigger applications and it was successful. There was one thing consistent in my approach, and it has been since I started architecting software; and that's failing the architecture on the slide deck first. If an architecture that looks great on a slide deck, fails after the brainstorming session, it's bound to fail. So, my secret of success lies in deliberate attempts to fail the architecture iteratively. I utilized sessions of brainstorming followed by deliberate attempts to fail the architecture. This is an on-going process. The process is simple:

1. Understand all the business requirements.
2. Make a conscious effort to understand the future vision of the company as well as theproject.
3. List all technical, functional and non-functional requirements.
4. Create an architecture that encompasses everything from 1-3.
5. Brainstorm on how to fail it by identifying:
    a. Failures,
    b. Bottlenecks,
    c. Downtime.
6. Come up with an architectural solution to the problems identified in 5.
7. Add it to the architecture and create a better pattern.
8. Repeat the process (steps 3-7) until we come up with an architectural pattern that works.
9. Use these principles to create a modern architecture.

Note Even though the techniques used in Dhall Architecture have not been invented by me, combining these techniques and innovating on them has been a part of my professional work for a large number of years. I do claim to be the only person using these techniques. Dhall Architecture to me is a thought process and a living philosophy that changes over time with failures, successes and experience. Since, there is no name out there for such an architecture I've given it a name. Feel free to call it what you like.

Having worked in different business domains including, but not limited to airlines, gaming, entertainment, health care, cosmetics, e-commerce, finance, insurance, technology, social networks, and many others, I have been fortunate to learn a lot about these domains. It's worth appreciating the individuals who could do both technology and business well. It's very rare to find such individuals. For the most part, in any software development effort, there are very few individuals who understand the business down to the minutest detail. In the same vein, there are very few individuals who understand a great breadth as well as depth of technology. So, finding a rare gem who has mastered both is extremely difficult. In my experience in the industry, that one person can make the entire project successful even if backed by an average team.

What's important to keep in mind is that the chances are that the domain expert has been with the company for at least five years. That, in itself, means that he has worked at no other company for those five years. While it's great for domain knowledge, his or her technical knowledge is limited to some blog posts and books that are in the market. Trust me, not all experts end up writing books or even blog posts. Especially folks like me who have a big team of employees working for them and are involved in doing projects for multinational corporations, don't really have time to pen down all what we know. The domain expert has limited technical knowledge unless he has been really open to inviting experts time and again and they have been open to sharing their knowledge.

Similarly, tech experts like me have to keep up with the domain knowledge regardless of the extent of knowledge we have. It will be an unfair statement to say that we are as good as the domain experts working for our clients. Fortunately, domain-based knowledge doesn't change as often as technology, so it's not hard to get ramped up on it. Most projects fail because decisions were made based on the partial understanding of few blog post. Disinformation circulated in the tech world are the cause of failure of a large proportion of software development projects. Every now and then, we hear news of a website failure, enterprise application, native mobile, or tablet app and how it could not scale as per the load. Some examples of this are Instagram, Pokémon now, Twitter, etc.

We have gotten a little sidetracked here, but the approach that has been described needs to be used with the domain expert in room and only after understanding the actual business requirements. It's important to have a realistic understanding of the kind of scale that is needed. Scaling is a science and it's deceptively simple when we talk about it in theory. For example, if your current system can handle 10,000 requests per second and uses only one web server, then in order to go to a million requests per second, you should need 100 servers. We only wish it was that simple! What if the database is an RDBMS? Since it was part of the web server, can we scale the web server the way we described above? Surely not without significant changes in the architecture. Can we scale it if we have 100 web servers and 100 database servers? It's not as easy as it sounds. Horizontal scaling of web servers might be easy, but scaling the RDBMS is a different animal altogether.

What if we plan for a huge scale and then our domain expert tells us that we don't have more than 1,000 users a day on average, but the website gets a billion hits a day on both Thanksgiving and Christmas? In layman's terms, we might be able to run the website with just one server throughout the year, but for just two days we may require 100 servers. Is it worth it to pay for those servers all year long or lease them as you go via a cloud provider? I can't stress enough the need to work with the domain/business expert on creating a successful scalable solution. This is just a cursory description of scalability. We haven't even touched the tip of the iceberg yet. Even though I plan to deep dive into certain concepts, the actual code for implementation will be outside the purview of the book. It's important to understand that the architect leading scalability efforts needs to be hands-on. I wouldn't have been able to scale applications if I didn't write code myself. Hands-on architects can strive to achieve good performance and use the resources optimally.Good performance per request ensures good scalability. I have been involved in projects where we improved scalability by improving performance and reducing the number of servers. My team and I learned our most important lesson that in order to scale, we need to take performance seriously. We follow the simple process of getting an independent module completed and then we make sure it's tested well before we tackle performance if that module qualifies for a performance check. A performance check, early in the game, is an interesting deal altogether. Sometimes a lot of effort up front may not be the wisest approach, but the mentality of fixing performance issues only once we have built it will most likely lead to failure.

Let's look at a real-life example. In one of my projects, an architect was retrieving all the rows from a table in the database and then doing a join in memory. When I showed curiosity regarding the rationale behind it, he mentioned that there will be no case when the table will return more than 300 rows. He didn't see any value in going back to the database again and again. The logic was to cache the records in the web server to speed up paging retrievals. My assertion was that this is a generic functionality that also involved paging and could be used for any other entity. An entity (in the application) was a real-world object that got data from a corresponding table in the database (using a repository pattern). The logic sounded good on paper; however, my question was valid. How do we prevent a junior developer from not copying the pattern? The architect didn't seem to think it was a problem. In fact, he didn't think anyone in his right mind would even do that. During further analysis, we found out that it had already been referenced in three different places. The good part was that he had encapsulated the logic well and used dependency injection. Refactoring here would be a cake walk. My further discussions with him reiterated why it was a bad idea to cache using the web server cache. It is sufficient for a small application, but it is a scalability nightmare. Long story short, we used Redis to add a second-level cache in it and fixed the logic to query Redis for only 20 records. Every page displayed only 20 records, and the data barely changed. Even if it did, we had an offline process to keep Redis in sync.

The architect is and was a good architect. Good architects think about encapsulating logic, using dependency injection, use generics, and make it reusable. What makes him even better is his ability to take criticism, be open to conversation, learn, and apply the right solution. However, there is one lesson that is learned hard in software development. That is preventing the lowest common denominator on the team from misusing the code base or not implementing the architectural principles set forth by the team. On a separate note, it is quite obvious to question the use of Redis in this case. You are right in assuming that there is no need to use a second-level cache for just one function call.

Trust me, there were a lot of other reasons to use Redis in the application and a lot of these will be unveiled in subsequent chapters.

CAP Theorem

No discussion of scalability is complete without the CAP (Consistency, Availability and Partition Tolerance) Theorem. It is also known as Brewers' Theorem after the computer scientist, Eric Brewer. In layman's terms, CAP theorem states that in any distributed system, it's only possible to get two out of the three guarantees viz. Consistency, Availability and Partition Tolerance. Keep in mind, a distributed system is one that is made up of individual machines or nodes. These nodes communicate effectively with each other via messages. A failure of a particular node may not mean the failure of the system. We all know about Master-Slave configurations. In a distributed system, we have multiple nodes and in order to make our system resistant to failure, we may need to back up a Master node into one or more slave nodes. Different systems can use different algorithms to achieve the same results.

However, for our understanding, let's take an example of a distributed system where a Master node is the only node that the system can write to. Assuming we get a request to add an order to the system. The order gets written to the Master node. Once the order is added to the Master node, let's assume the system makes the Master node send a message to all slave nodes to add the new order. If the request is made to a slave that has not been updated yet, the order won't exist and the system will be deemed to be inconsistent. But, if the system makes sure that a subsequent read request to the system will be able to guarantee the retrieval of the latest request, it's considered to be a consistent system. Relational Database Management Systems are consistent. Every read request returns the most current data.

1. Consistency: A distributed system that returns the most current data no matter which node the request was made to is considered to guarantee consistency. In layman's terms, if a write or update request to any node in the system is replicated to other nodes in the system, before the read request, it's a consistent system. So, the bottom line is that every read will return the most recent write. The system will not return stale data but the most recently updated data. In order to achieve consistency, the system has to update all the relevant nodes at each request, before allowing any reads from the system on that particular resource.

2. Availability: It is the ability of a node to respond to requests if the node hasn't failed. Availability allows for failed nodes. However, if the node hasn't failed and doesn't respond to a legitimate request, it is considered to not be available. In order to achieve availability, the system needs to replicate data between different nodes.

3. Partition Tolerance: It is the guarantee of a system to respond to requests even when the system is partially down. No failure less than a complete failure of the system should allow the system to respond incorrectly. So, if the connections between some nodes in the system are lost, the system is partition tolerant, if and only if the system as a whole is still consistent and available.

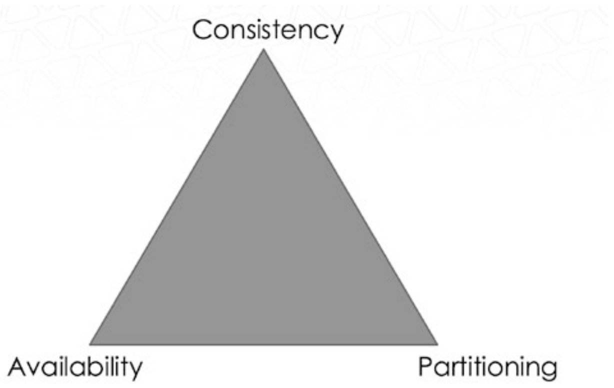


Figure 2-1. *The CAP Theorem*

Fallacies of Distributed Computing

In order to understand distributed computing, this white paper is highly recommended: https://pages.cs.wisc.edu/~zuyu/files/fallacies.pdf. It elucidates common mistakes an engineer makes with architecture distributed systems. These fallacies are the following:

1. Network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. Network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. Network is homogenous.

How to Achieve Scaling

Step 1: Getting Started

The simplest system to start deploying an app to is one that has the entire app on one server. For now, let's assume we have a website running on a web server like IIS or Apache and has an RDBMS like SQL Server, MySQL, PostGre, or Oracle. So, in this system we will have both the web server and the database server on the same physical server.One of the common

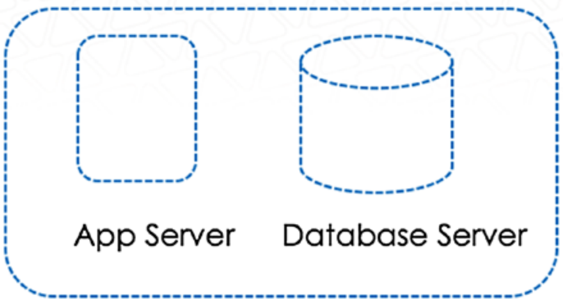


Figure 2-2. *A server that hosts a web server with a database server*

mistakes in designing architectures is the faulty imaging of the server. This architecture fails to see the difference between the database and app server. App servers need a better CPU and RDBMSs thrive on more RAM. With more RAM, the more frequent queries could be now retrieved from it. The larger the RAM, the better the subsequent retrievals. The web server can always use more cores if there is a computation-intensive job. For the most part, it can be argued that web servers should never do a computation-intensive job. If there is one, we can always make an API call to a process that finishes the job. This is surely a good approach. However, in a real-world scenario, this could become an overkill in certain applications.

One of the big improvements in the Node.js community has been to make it a multi-threaded system. A single thread with multiple load balancers can get the job done. However, there is no point not using the additional cores we may have access to. By no means there is an implication that a web server doesn’t need RAM and a database server doesn’t need a better CPU. The point is that there are different configurations for both these servers, and they need to be configured separately in order to be optimized to perform their jobs well. One of the big mistakes is to optimize both the database and the web server in the same way. That’s why the approach of having both the web app and the database in the same server doesn’t work that well.

Let's evaluate the current architecture? If the database fails, the system fails. Similarly, if the web server fails, the entire system fails. It could be argued that in case of failure of the database, the system can still serve static files. Yes, that's true. However, we will keep the focus on enterprise systems of scale, and it will be correct to assume that they just won't be serving static files. So, we have two single points of failure in the system, and we have already seen that the system doesn't scale.

Step 2: Vertical Scaling

So, how does it relate to scalability? The only feasible way to scale the architecture in Step 1 would be to add more RAM and CPU to the existing server. This is also called Vertical Scaling or increases the hardware resources without altering the number of servers or nodes. Vertical Scaling is also referred to as scaling up. Scaling up requires adding more resources to the same server. It's more like upgrading the server to a better configuration of hardware assets. On the contrary, scaling out means adding another server to the system with a similar configuration. This works up to a certain amount of traffic. However, after that the law of diminishing returns kicks in and the system is impossible to scale. After a certain tip-off point, any addition of resources doesn't increase the amount of traffic the system can handle. In layman's terms, there is a tipping point, beyond which the system would just not scale.

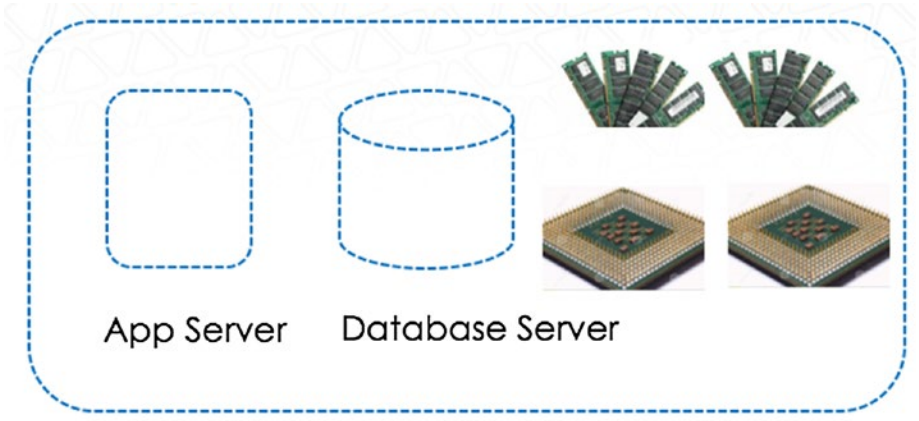


Figure 2-3. *Adding more RAM and Hard Drives to the server in Figure 2-2*

One of the other, often overlooked, downsides of vertical scaling is that it requires downtime. Imagine replacing the RAM of the system, you will need your system to stop and be unresponsive at least for some time. If the system is just one server, downtime is unavoidable. Vertical scaling could also mean tremendously high costs beyond a certain configuration. It's always cheaper to add another node with commodity-level hardware than upgrading to an elite configuration. Beyond a certain configuration of hardware resources, costs increase exponentially.

Let's evaluate the current architecture. If the database fails, the system fails. Similarly, if the web server fails the entire system fails. Similar to Step 1, Step 2 has single points of failure in the system and we have already seen that the system doesn't scale.

Step 3: Vertical Partitioning

As we can see our major problem is that the system is still one cohesive system and it won't be available to serve any request the moment any of the nodes (database and the app server) fails. This is where Vertical Partitioning can be used, which means deploying each service, be it a database, a web API, or an app server, on a separate node. It is when each node or cluster performs different kinds of tasks. For example, notification server, mail server, payments service, RDBMS, web server, etc., perform different tasks and are different from each other.

There are certain benefits to this approach over Step 2. In Step 2, if for some reasons, the database ended up maxing either the CPU or the RAM, the app server would barely be functional and vice versa. Vertical partitioning avoids the situation of sharing resources and avoids the conflicts arising therefore. It also reduces context switching. Context switching is the process of storing the execution context for state of a process or thread so that execution can be resumed later from the same point.

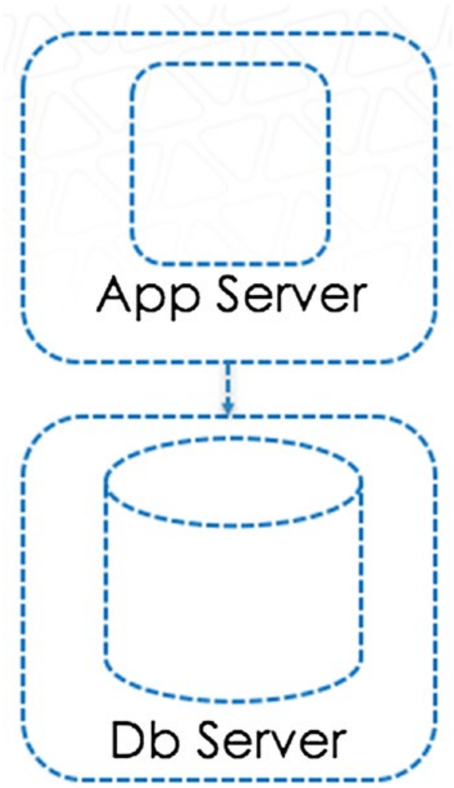


Figure 2-4. *Vertical Partitioning*

Another important factor to consider is that the web server needs to be tuned differently than the database server. In case of a web server, we may need to upgrade the server kernel to the latest version of TCP, increase the initial congestion window size, and enable window scaling to increase the receive window size to a higher value. In certain cases, we may need to disable slow start after Idle. Other optimizations may include enabling TCP fast open, investigating the possibility eliminating redundant data transfers, compressing the transferred data, and reusing TCP connections whenever possible. Most of these optimizations are great for the web server. However, they do not mean much in case of the database server. The database server for an RDBMS may need to be tuned entirely different than No-SQL database like Redis, Elastic search, MongoDB, or Cassandra. A typical mistake in the enterprise has been using the same image (and optimizations) for different kinds of servers. With vertical scaling, we have the advantage of optimizing and tuning according to our needs. We can tune the web server as well as the database differently as they are now independent of each other.

Let's evaluate the current architecture. If the database fails, the entire system fails, except that it will still be able to serve static files. Similarly, if the web server fails the entire system fails. Similar to Steps 1& 2, Step 3 has two single points of failure in the system and we have already seen that the system doesn't scale.

Step 4: Horizontal Scaling

Horizontal Scaling, commonly referred to as Scaling Out, is the ability to add any amount of hardware and software entities in a fault tolerant system, thereby increasing the actual capacity of the unit as a whole. Fault tolerance is the property of a system to continue to operate without complete failure even though parts of the system might fail. So, the idea is simple. Rather than having to pay for high-powered machines and scale up, it's better to pay for commodity-level machines and scale them out. Any failure of a machine won't bring the entire system down as we don't have a single point of failure. And it is surely a cost-effective way. Scaling up will reach to a maximum at some point of time and or another, and it's not feasible for scaling the system indefinitely. However, scaling out is a very promising solution for scalability when compared to scaling up.

In order to scale the web server layer, we will need a load balancer. A load balancer is quintessential to balance the traffic across all nodes (in this case, web servers) in order to prevent any individual node from getting overloaded with requests. Once the machine gets more requests than it can handle, the system can become unresponsive. At the same time, in the absence of a load balancer, the additional nodes provide no value. When it comes to a load balancer, a case could be made for either a software or a hardware load balancer. Software load balancers are known to implement a combination of algorithms such as Weighted Scheduling Algorithm, round robin scheduling, least connection first scheduling. Some of the examples are HAProxy, NGINX, Varnish, LVS, mod_athena etc. And hardware load balancers consist of specialized routers or switches that balance the load between the server and the client. Some of the examples are Cisco system catalyst, Barracuda, Coytepoint, F5 Big-IP, etc. It's really not that one is better than the other, it's more about what features are more important to the business. And to be more specific, the discussion is really not about hardware or software load balancers per se, it's more about buying a proven appliance versus building our own or using an open source solution. With a high-end commercial offering, it's easy to get a richer feature set that has less management effort, which could also mean less management costs. There's also an advantage of getting support from the vendor and, of course, you have the luxury of getting data and statistics, which help in troubleshooting. Additionally, using an open source software load balancer may mean an increased effort on installing it the first time, especially if that means adding other features that are not available with the default solution. That said, open source software load balancers work great too. It's all about what we are comfortable with.

Assuming that we are dealing with mostly HTTP traffic when it comes to the web world, it's a common practice to include HTTP accelerators with load balancing. As the name suggests it accelerates the traffic by acting as a transparent forward proxy for HTTP traffic that saves static objects in a cache (memory-based or disk-based) and serves them from it whenever possible. The logic is pretty simple. All web servers indicate, as part of their response, whether an object is static. This is done using HTTP headers, for example, **Cache-Control** and **Pragma**. It is also possible to exclude certain specific objects from caching, if needed.

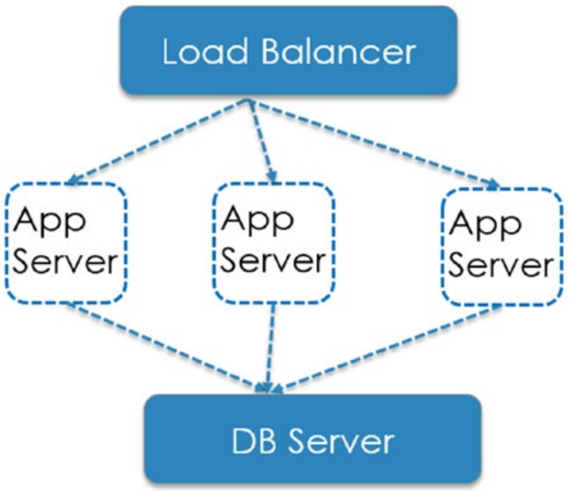


Figure 2-5. *Horizontal Scaling the App Servers*

In this step, we are horizontally scaling up the web server only. We are assuming that our database is an RDBMS, and it doesn't require us to scale beyond that one server. As you can see in the diagram, the load balancer equally divides the incoming calls among the app servers. If the traffic increases, we can add any amount of web servers and the system should continue to operate well. At the least, we have scaled the web server layer now and any failure of a hardware like hard disk or RAM in a web server will only affect that particular web server. It won't affect the health of the complete system. It will still affect the concurrent sessions that were part of that particular server. However, the subsequent request from the user will be redirected to a healthy server. The downtime is pretty minimal and the recovery is almost seamless from the users' point of view.

In case of horizontal scaling, the system increases the overall capacity by increasing the number of nodes (which in this case are the web servers). Each and every node is expected to perform the same exact tasks and are supposed to be identical in nature. It's a common best practice to have the same operating system image, as well as the same version of software and tools installed on the nodes. Each and every node should use the same configuration for security as well as performance tuning of the server. If it's a web server, it's better to tune it for high I/O performance, and hence a better upgraded CPU is sometimes preferred over memory and the configuration should open it up to perform better for scaling requests. Whereas if it's a database server, generally speaking, memory may be preferred over CPU, and it needs to be tuned to perform better as a database. This configuration needs to be consistent across all the nodes. Especially when it comes to security, most of the times, hackers exploit the vulnerability that comes in a part of sample code and applications that comes in with the operating system or a particular software that was installed. So, it's very important to keep a consistent image that has undergone load testing, security testing, etc., and is configured for the specific type of application.

Cost Analysis: It's not true that horizontal scaling is always cheaper. It truly depends on your app. If your app has a small number of users, it might be just better to stick with vertical scaling, at least as long as it's a cheaper and more viable option. For an app that doesn't require a lot of scale, scaling out might mean additional licensing costs on not just the operating system (unless you're using a free one) but also other tools that are being used on that server. And in certain cases the additional hardware might be more expensive than just upgrading the hardware at times. More servers also might mean more management headache. So, if the system doesn't require the kind of scale we are talking about, vertical scaling is not a bad option. Again, this perspective is not true for systems that need high scale. But it's important to mention this.

By now, you've probably noticed that we still have two points of failure. One is the database and one is the load balancer itself. There is really no point in scaling the app server if we are going to be dependent on the load balancer. Any failure or malfunction in the load balancer would mean failure of the entire system. Even if the database layer is built to scale indefinitely along with the web server, the point of entry to the system is going to be the load balancer, and apparently, it doesn't conform to our standard practice of not having any single point of failure.

Let's evaluate the current architecture. If the database fails, the entire system fails, except that it will still be able to serve static files. Similarly, if the load balancer fails, the entire system fails. Similar to previous steps, we still have single points of failure in the system, and that is not going to create a scalable system.

We shall talk about dealing with the problem of database as a bottleneck in the next step. However, can we horizontally scale the load balancers to start with? Generally speaking, this is very easy. There are two ways we can do this.

1. Active-Active
2. Active-Passive

As the name suggests, Active-Active is a configuration where we have at least two load balancers independently capable of handling traffic and balancing it between the respective web servers. Both of them run on primary and secondary nodes. In case of a failover the other load balancer now services the failed-over clients, as well as any new clients. When the failed load balancer is backup, it takes its client connections back and resumes its services. This process is called Failback.

Active-Passive is a configuration where the primary load balancer takes care of all client connections and balancing traffic between the web servers. However, the secondary load balancer doesn't not play an active role in balancing traffic. Rather, it stays in a listening mode. It monitors the primary load balancer regularly. In case of a failover, the secondary load balancer takes over and handles the entire traffic and balances it across the web servers.

Either approach will solve the problem of removing the load balancer as a single point of failure and is scalable out of the box. Depending on the load, it makes sense to increase the load balancers. The Active-Active configuration could be considered to be more expensive generally than Active-Passive. However, we need to do a good job analyzing the cost. Now, we need to think of the following:

1. What's the cost of downtime of the app if the load balancers are not scaled at the required rate (assuming a sustained increase in traffic)?

2. Active-Active configuration means that the load balancers operate in a way that all of them are used, almost equally, at any given point of time. What happens if one of them goes down, and what's the cost of that slowness for the end user?

3. How are we maintaining user sessions? In case of sticky sessions, a particular user may be sent back to a particular web server by a particular load balancer. This is called a sticky session. What happens to this user's session when that load balancer fails over?

If we think through these questions, we realize that some approaches including sticky sessions may not be the best approaches for scalability. In fact, for anything to scale we should create a system where a failure in any node does not disrupt the entire system, and no failure other than the failure of the entire system brings business to a halt. Even though you might be familiar with the concept of session management, it's worth revisiting this in the context of scalability. As we know, the HTTP protocol is a stateless protocol. That means that every request (and response) is independent of other interactions to the same Web server. In layman's terms, the protocol does not allow a place to capture state. However, in real-life scenarios, there are times when we need to associate some information about the status of the user such as access rights and localization settings that should be applied during a particular session to each and every interaction the user has with our web server. Traditionally, in applications that ran out of just one web server, programmers used to capture state in the same web server. That is surely not something that can scale. In order to make a solution horizontally scalable, we need to make sure that every single web request from the same user can be easily handled by any web server. There are three different kinds of solutions that exist when it comes to session management. They are:

1. Sticky Sessions: As we discussed earlier, this is a scenario where if a particular user was served by, let's say, Server X, every subsequent request by the same user must go back to server X. This definitely has a few downsides.

    a. The major one being the possibility of unequal load in certain servers compared to others.

    b. In case server X goes down, we lose all the sessions on that server. Imagine a user having his entire shopping cart in that session; it would not be a good experience.

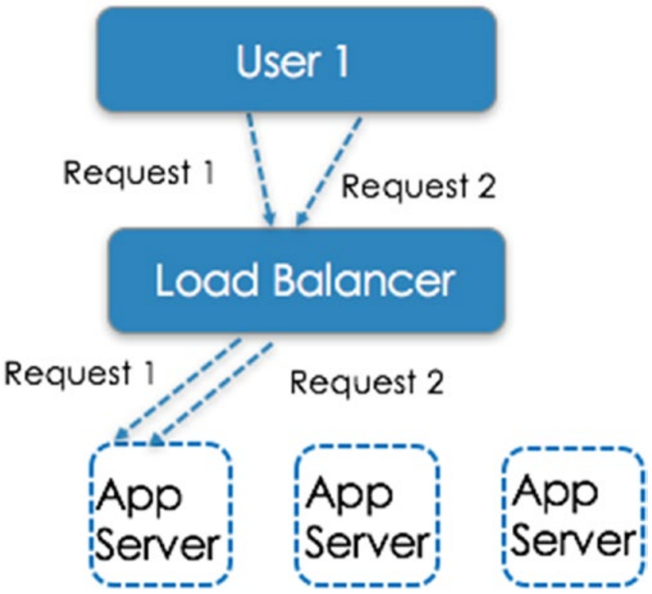


Figure 2-6. *Sticky Sessions*

2. Web session clustering: This is an approach in which the session is created in one server, then requests are made from that server to replicate the session to other servers. Usually the session is stored in memory for speed. This solves the problem of scalability as the subsequent request of the same user can be directed to any web server. It does create the additional network I/O overhead though especially with the increase in the number of web servers. At the same time, the increased number of web servers is usually because of the increased number of users. That means a big chunk of the web server memory is going to be used for replicating the sessions. Imagine having about 100 servers and each server serving a million users each. Don't you think it's redundant to have 100 million sessions

in each server? Imagine the cost of serializing and deserializing the entire session object and sending it over the wire and replicating it. And there is a slight possibility that the new request from the same user arrives a server that may not have the latest session information yet. What if the request to replicate the session information fails? And even if we can handle some of these cases by retries and other algorithms, why add this additional overhead?

So, in my opinion, it's a solution that could be used for a smaller number of web servers and users, or the application doesn't have a lot of session writes. However, there are better solutions for a higher load otherwise.

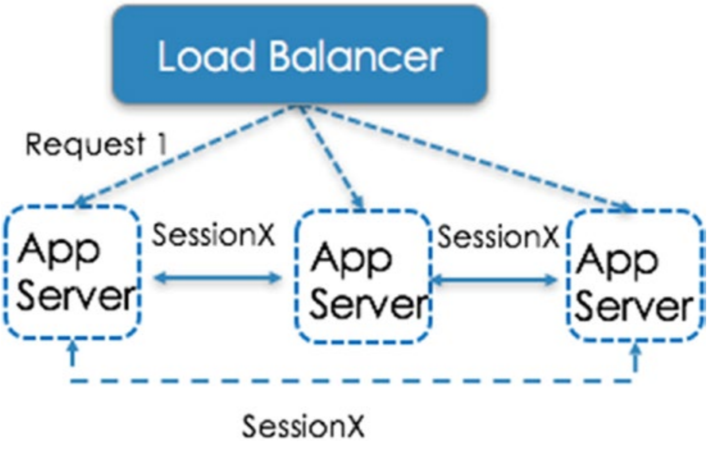


Figure 2-7. *Web session clustering*

3. Centralized session store: Another way to scale the session is using a centralized session store. In this solution, we do not want the web servers to have any session information whatsoever. Instead, we want to have a centralized session store where all

the session information is kept. For the purpose of scale, this works if the session store is a scalable store. Imagine something like Redis or Memcached, or any other in-memory store that scales out of the box. For an application that requires a lot of session writes and has an ever-increasing number of users, this is a very scalable solution. Just a quick note: there are cases when you may need to persist the session. Nevertheless, the choice of the data store totally depends on the type of business we have. The session could be persisted on the disc in case of a No-SQL session store like Redis. Or it could also be persisted in a Relational Database Management System (RDMS). The trade-off in that case would be performance. However, I like to reiterate that in real life it's really the business that defines the choice of technology and we have to be cognizant of the fact what our priorities are while designing an architecture. At the same time, we have to make sure that the architecture is scalable and futuristic. Traditionally, a lot of people in the enterprise have preferred the (RDBMS) database over No-SQL, but there are in-memory databases like Redis that are fast but also have the ability to persist data on disk. These databases are by far the the best in performance as well as scalable out of the box. And with the right configuration and right planning, most of the concerns regarding losing session data can be easily mitigated.

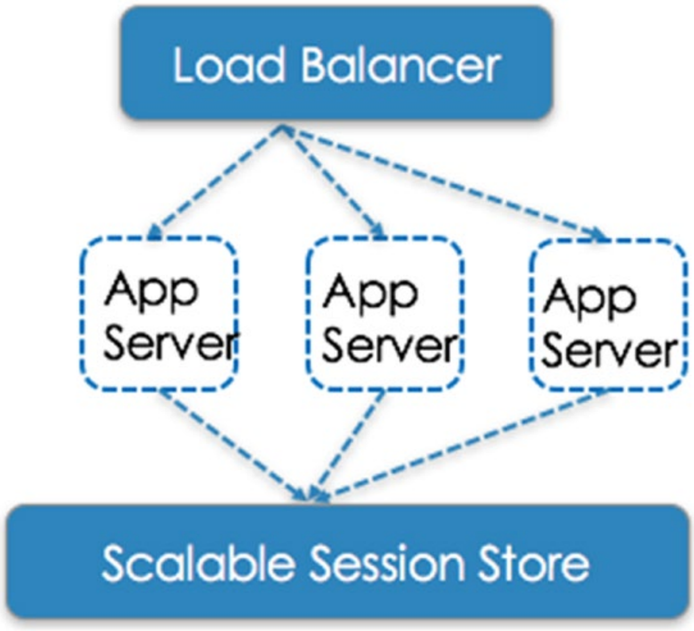


Figure 2-8. *Centralized session store*

Note An app server could be an API or a traditional web architecture like MVC. For scalability of the system, it's preferable to have an API in the app server. Given that we have many different devices nowadays, API is preferred over a traditional web architecture. An API could support both thick-client (native apps) or thin-client (web-based) architecture and any kind of device (mobile, tablet, desktop, smart TV, etc.).

Conclusion

One of the benefits of speaking at internationally acclaimed conferences is meeting smart architects, developers, executives all around the world, and receiving information about architecture choices that can be blockers for scalability. Until recently, there were many new projects that started with less accurate choices in designing applications of scale. Even today there are frameworks out there that lead you toward creating sticky sessions. In the absence of a good architect, developers, for the most part, may get misled into using the framework API and end up making wrong decisions. For example, while creating a cache for a web application, it's quite normal for you to use the caching API that is part of the web framework that is being used. More often than not, that would mean using an in-memory cache. However, for an app to scale, it's beneficial to use a second-level caching engine like Redis or Memcached.

It's not very surprising that many start-ups actually start with both the web server and the database server on the same server. Later, when they end up scaling, they incorrectly use the same configuration for both the servers. I hope by now it's very clear that both the web server and the database server need to have completely different configurations. They need to have configurations that pertain to their specific needs. Tuning a web server is entirely different than tuning a database server.

So far, we have seen some very basic scaling and partitioning strategies that end up in varied server configurations. However, these are also building blocks toward a more robust architecture down the line. All scalable architectures will have a combination of vertical and horizontal partitioning as well as scaling. There are no either-or strategies. We will now discuss some advanced concepts.

CHAPTER 3

Scaling – Advanced Concepts

In this Chapter we build on the SPOFs discovered previously and make our architecture better. Concepts like caching, partitioning and scaling are discussed in detail. If you remember the figure from the previous chapter (Figure 3-1), you might recall that we have two single points of failure (SPOF). In this case, we had the database and the load balancer as the SPOFs.

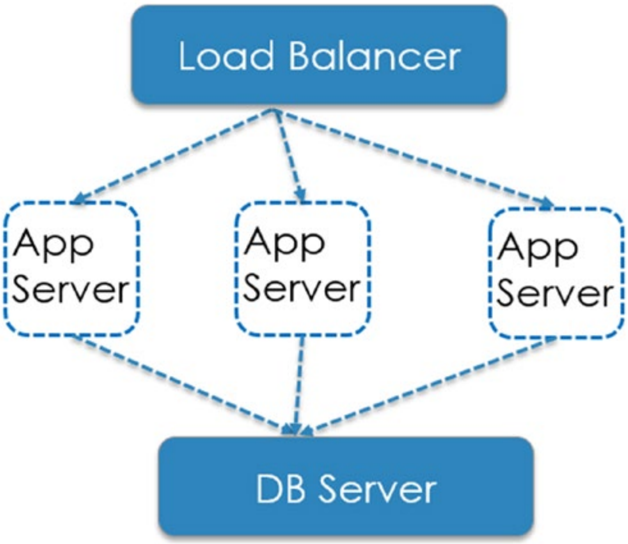


Figure 3-1. *Horizontally scaled app servers with a load balancer and a relational database*

However, we later looked at eliminating one of these by using an Active-Active or Active-Passive solution for the load balancers. In both of the solutions, we have minimized the dependence of the application on the load balancer. Unless and until all the load balancers decide to die at the same time, the system will continue to run smoothly. Load balancer solutions can vary from hardware or software load balancers to a hybrid combination of those. The details of such solutions are outside the purview of the book.

Note We talk about App servers throughout the book. This can mean having a regular web app, an MVC app, a Web API or another Web application. This encompasses pretty much anything that runs on a Web server. Most applications of scale need to have an API in order to support multiple clients like a desktop web application, mobile web application, native mobile or tablets app, Windows app, Mac app etc. We assume that the readers will use the right architecture standards according to their needs. Since the focus of this book is entirely on scalability, we don't get into the details of security, API gateways, throttling strategies and other essential components of architecture.

Caching

In layman terms, caching is just a simple mechanism to speed up a system by storing commonly used data that doesn't change that often and storing it close to where it's used. It is also, inadvertently, a mechanism to decrease the load on a system component like a database. As we discussed in Chapter 1, caching is one of the REST principles. When we work with cache we need to understand that there is always a possibility of having stale data and the application should be fine working with it. We can use various caching patterns that display fresh data from the system of record by hydrating the cache timely and accurately.

Primarily caching starts with mostly an in-memory cache that lives in the server memory. The data that is supposed to be cached lives in the server RAM and if the application asks for it, it's sent back to the user. Let's walk through an example.

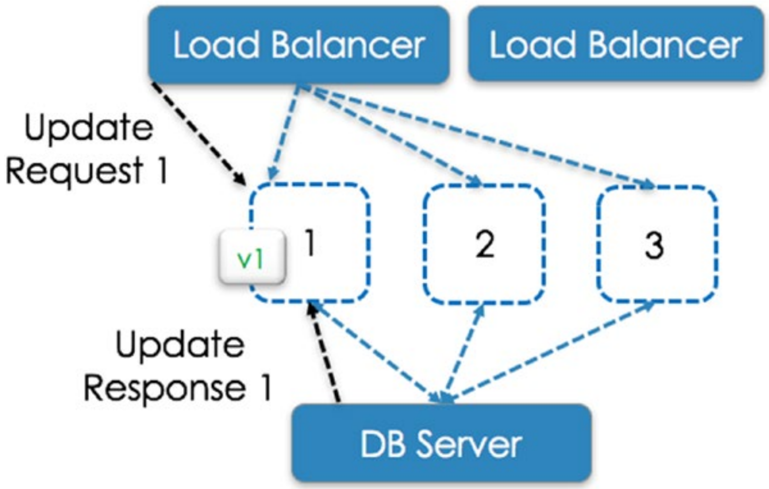


Figure 3-2. *Diagram showing one request updating the value to version 1 (v1) and storing it in Server 1*

In Figure 3-2 above, we can see that one of the load balancers makes a request to the database server to update a resource. This request is illustrated as *Update Request 1*. The database responds with the updated value. That's called the response *Update Response 1*. We can assume that the updated value is called *v1*. If we work to use some kind of in-memory cache, we will now have the value v1 in server number 1 and there will be no cache in server numbers 2 and 3.

It's quite possible that there is another update request that goes to server number 3. That request is illustrated as *Update Request 1* in the following diagram (Figure 3-3). If all goes well the database will respond back with the updated value v2. Since the cache is supposed to be in memory and there is no way for the servers to share updated values among each other, it turns out that server number 1 will still have the outdated value v1. Server number 2 won't have that problem just because it will have no other option but to go to the actual source of record, which is the DB server.

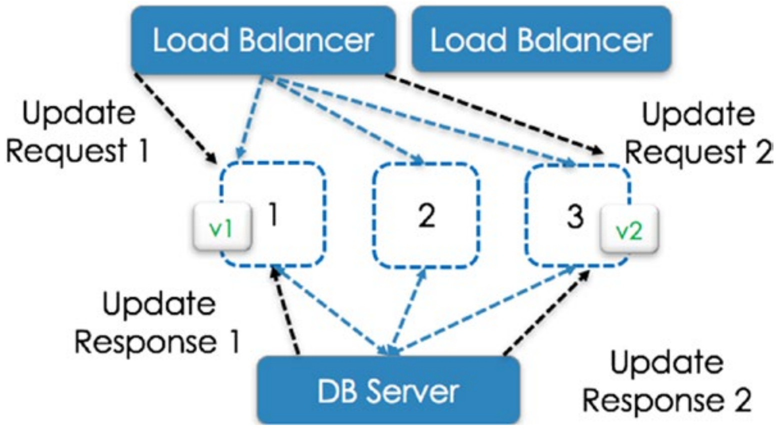


Figure 3-3. *Diagram showing second request updating the value to version 1 (v2) and storing it in server 3*

Even though there are ways to solve such problems, it always makes sense to use a second-level cache like Redis or Memcached. Second-level caching servers are scalable out of the box. Since the cache is shared, all the updates from the web servers will go to one place. All the web servers can now make a call to that one proxy or agent and it will return the last updated value. With increasing traffic, we can horizontally scale the web servers, and in return, horizontally scale the caching servers as needed. This system not only is highly scalable, it also has a better overall performance and system utilization. One of the major benefits of having a second-level caching server is that it reduces the number of calls to the DB server drastically.

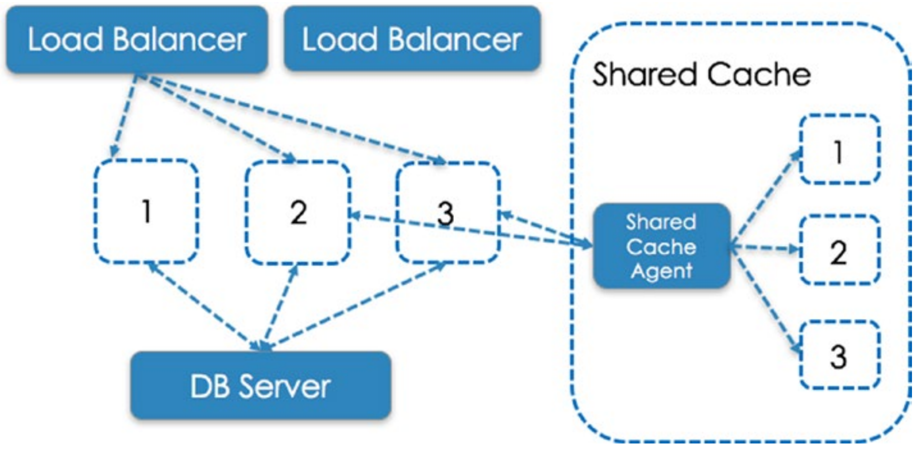


Figure 3-4. *Diagram showing use of Shared Cache*

Figure 3-4 above shows web server numbers 2 and 3 making calls to a shared cache agent that works as a proxy to the shared cash servers behind the scenes. This agent could be as simple as another load balancer or a more involved proxy depending on the actual caching system.

Step five, we were able to successfully reduce the number of calls to the database server. It's a welcome step, with the increase in traffic database being a possible point of contention. At the same time, we still have the database as a single point of failure.

Understanding Partitioning

In order to improve the performance of a database we can use different database partitioning techniques. These are hardware, vertical, and horizontal partitioning. When a table grows in size due to the number of rows and columns, it leads to an increase in the scan time. Let's assume we have a table with 10 columns and 1,000 rows. If we search via a primary key, the data set is small enough to have a very quick retrieval. If the key is also a clustered index, the retrieval could be even faster. However, over a period of time, this data could be millions of rows. That will surely increase the scan time. It's also normal to assume that this table itself could now have more than 25 columns itself. A partition of this table can be helpful in improving the speed of querying quite a bit. One way to partition would be to reduce the table into smaller tables. Once that is done, an average query will be quick to scan because it will execute for a small portion of that data. This helps in speeding up maintenance tasks that include, but are not limited to, rebuilding the indices for backing up the table. We can also achieve partitioning without splitting tables by simply putting these tables on individual disk drives. We can partition the database in three different ways:

1. Vertical Partitioning
2. Horizontal Partitioning
3. Hardware Partitioning

1. Vertical Partitioning

Now that the load balancer is taken care of, we do not have any SPOF except at the Data Layer. It's really our RDBMS that is running on a single server. Most of the time, scaling of the RDBMS can start with replicationand denormalization. However, vertical partitioning a database might not be a bad option. It's almost the first logical step to scaling the database.

There are two types of vertical partitioning. They are:

1. Normalization
2. Row Splitting

Normalization is the process of removing redundancy in a database by finding the right linkages between tables and linking them using primary key and foreign key relationships. Imagine having a Cart table that has all the product, customer, order, payments, and cart info. It can be a nightmare to keep adding redundant values like customer name, customer address, customer phone number, and other values from the customer table to the Cart table. It's the same way we don't want the product model, product price, and other information to be duplicated in the Cart table. Traditionally, Relational databases (RDBMs) have been known to use the technique of normalization and create different tables for different entities. In this case, it would be Customer, Product, Payment, and Cart. All these tables will be linked by primary keys and foreign keys. For example, the Cart table will have foreign keys like ProductId, CustomerId, OrderId, and PaymentId. Based on the business requirements these entities, may have a one to one, one to many, or many to many relationships among each other.

Row splitting: Even though the normalization is a very good technique to remove redundant data and establish relationships between data tables, there are times when row splitting must be used to reduce scan times. It works by dividing the original table into multiple tables with the same number of rows but fewer columns each. All the split tables continue to use the same unique key. In order to retrieve the original table with all the columns, all we need to do is query on the split tables with the same ID.

A good example of using vertical partitioning would be a report where searches are made on only some specific columns. For example, searches that made using the city, ZIP Code, State, and ID. However, the report may have some kind of verbose columns like blog post, a picture of a blob, etc. It may make sense to move the blog posts and the picture blob to a completely different table. This will surely increase the overall performance. However, one thing to keep in mind is that vertical partitioning may adversely affect the performance if the partitions are very large. At the same time, we have to make sure that we have a sound join strategy as the data is in multiple tables.

2. Horizontal Partitioning

In contrast to vertical partitioning, horizontal partitioning divides the table into multiple tables that have the same number of columns but fewer rows. Imagine running an e-commerce business for 20 years. Imagine getting hundreds of thousands of orders in a month. Even though the Order table can grow to a huge size, it might make sense to start partitioning that table horizontally per year. Partitioning data according to the age of the data is very common. Depending on the use case, it may also make sense to partition the data per quarter or even per month. So even though the table will have the same number of columns, the number of rows in that table will decrease drastically.

Horizontal partitioning definitely can reduce the size of the table and speed up queries. However, like vertical partitioning, this also needs to be done carefully. We should take the appropriate time to analyze the data as well as user behavior. Otherwise, we may end up in scenarios that quite frequently query multiple tables. Data from these tables will have to be merged using UNION operators, which are not the most appropriate things for performance.

3. Hardware Partitioning

With a tremendous improvement in database hardware in recent times, it makes sense to take advantage of it. Nowadays, the hardware is so good that sometimes we forget to even optimize it. Even small changes that will improve by 10 milliseconds can be very helpful for the overall performance. It depends entirely on the frequency with the which the users execute that particular query.

With servers with multiple CPU cores, it’s quite common to use multiple threads so that multiple queries can run at the same time. One of the common scenarios is using one thread per table to speed up queries. Another way to partition the hardware is to use RAID (redundant array of independent disks) devices. These devices enable data to be striped across multiple disk drives. For example, if we are storing tables on different drives, this can drastically improve the query performance when it comes to joining those tables. At the same time, the table that has been stored on a single drive will be slower to the same table stripped over multiple drives.

Hardware Partitioning

As we have discussed earlier, an RDBMS is easy to scale up but not so good for scale-out. Scaling up of the database could have been done without a SAN too, couldn't it? Could we not just have added more CPU and more RAM? Yes, we could have. However, with increased load the disk I/O would be the one that becomes the bottleneck. It won't be able to keep up with the CPU. At that juncture, one of the ways to scale up an RDBMS is using a Storage Area Network. A SAN is basically a network of switches that connect servers with storage arrays such as disk arrays, tape libraries, and optical jukeboxes. They are accessible to servers so that the devices appear to the operating system as locally attached devices. In order for traffic from the storage network to not appear on a LAN (Local Area Network), the SAN has its own network of devices. These devices are generally not accessible to the LAN.

With a topology similar to Ethernet switches, a SAN's physical layer comprises a network of either Fibre Channel or Ethernet switches. Being a combination of hardware and software not only allows automatic backup of data, it also allows monitoring of the storage and backup process. It's composed of three layers:

- Host layer
- Fabric layer
- Storage layer

The host layer is composed of servers that allow access to the SAN. The operating system of the host uses Host Bus Adapters, which are hardware cards used to communicate with the storage devices in the SAN.

The fabric layer is composed of the networking devices used by the SAN to move data from the initiator to the target. It's comprised of a varied number of devices that include SAN routers, switches, bridges, gateway devices, and cables.

A storage layer comprises all sorts of storage devices in a SAN. A multitude of hard disks and magnetic tape devices may constitute the storage layer because they are joined through a RAID (redundant array of independent disks). RAID is a disk system that contains multiple disk drives, called an array, to provide greater performance, reliability, storage capacity, and lower cost.

Nowadays the storage arrays are a lot more sophisticated and cost effective. They come with features like data snapshots and have better availability and performance. They are capable of data mirroring not just within the storage array but also across storage arrays.

They also have the ability to allocate storage to a server outside the physical disk boundaries that support the storage.

SAN brings along with it a multitude of benefits:

1. Increased performance (by reducing disk contention): While architecting we need to look for contenting objects as well as figure out which objects are accessed frequently and which are not. It might make sense to colocate frequently accessed objects with objects that are barely accessed together. However, we may choose to separate contending objects on different disks. If needed, we may also consider separating them across SCSI adapters.

2. Improved availability: Typically, the database gets improved availability by using RAID 1 or RAID 5, which provides redundancy in return. However, allocation of storage for each server can be a tedious task and overestimating as well as underestimating storage needs are a common problem. SAN comes in handy to overcome the issues of capacity and availability.

3. Backups: Unlike traditional backups that need to be small backups in order to reduce restore times, SANs could be counted upon to have faster backups. SANs are capable of continuously capturing database snapshots. They do it smartly by not copying actual data but duplicating pointers to original data.

4. Database updates: SANs can help reduce risks while updating databases. Depending on the SAN configuration, the storage arrays can help quickly clone the database. A read-only clone, after testing, could be converted to a writeable clone. This method is far superior to restoration of the database and reduces the risks caused by upgrade, outages, corruption of the data, etc.

Even though hardware partitioning with SAN has a lot of advantages, at a higher level, it's nothing more than vertical partitioning. The reason is because we still have one RDBMS and that scales it up but really doesn't scale it out. Nevertheless, it's a great option to have, especially when we want to make sure that we get all the benefits provided by SAN without having to lose the benefits provided by the RDBMS. Scaling out an RDBMS could be a lot more challenging than scaling it up. However, we can serve millions of users without having to scale out the database. So, sometimes, it's not a bad idea to use a SAN before even considering scaling out (Figure 3-5). Given the benefits that it brings to the table, it's more than just an idea worth evaluation.

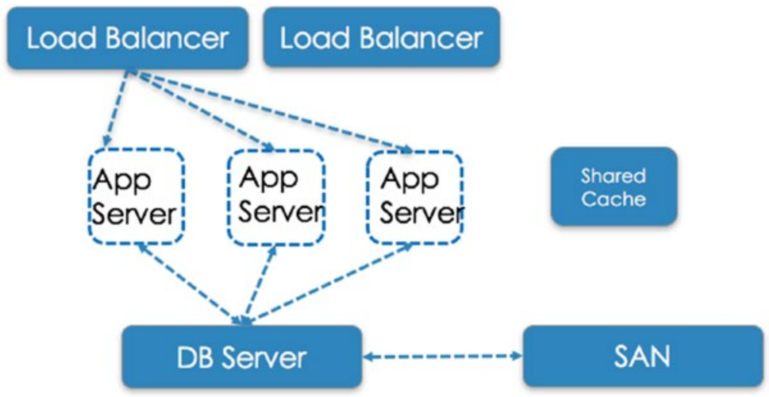


***Figure 3-5.** Diagram showing DB Server extended via SAN*

Well, it's time to reevaluate our architecture. After adding the SAN, we can say that even though the load balancers and the app servers are horizontally scaled and are no more a single point of failure, the database server still is. Not only is the database server a SPOF, it's also a point of contention. We can scale out all components of the architecture so far except the database server. SANs are easy to scale but the details on how to scale a SAN are outside the focus of this book. So even though it's a great solution for a significant amount of load, we still recognize some opportunities for improvement.

RDBMS Horizontal Scaling

At this point of time in our application architecture, the obvious and the only choice we are left with is to horizontally scale the database server. That means we should have multiple replicas of the database, and they should be able to take on similar tasks without having to depend on each other. When we replicate a database one time, it's a very simple process. However, since the replicas are horizontally scaled, any of the app servers should be able to write to an agent or a proxy that talks to the replicated databases without having to worry about data inconsistency. In order for that to be successful, there are different ways this can be accomplished.

In a transactional system that upholds ACID (Atomicity, Consistency, Isolation, Durability) properties, database transactions need to guarantee validity even in the event of errors, power failures, etc. In order for that to occur, we need to make sure that until and unless the change (update or a write request) is propagated to all the replicas, it should not send out an acknowledgment to the app servers.

For example, we can have a configuration where we allow reads from any server but writes to occur in only one server. Let's assume that someone makes a write request to complete an order. We would want that order to be replicated to the other instances. If, for some reasons, the user makes a request to retrieve the order before it was replicated to the other databases, he might not be able to see the order he just placed. If we guarantee the replication, the retrieval can slow down depending on the number of replicas. This is just one kind of replication. Replication could be of different types. Most major databases support replication and, if needed, we can also use agents that are capable of replicating not just a major RDBMS but also replicating data among different kinds of RDBMSs. For example, we can use an agent that will replicate data between SQL Server, Oracle, Postgres, and MySQL. Different kinds of replications are the following:

1. Transactional replication: This is based on transactional consistency. That means even with multiple servers (with one publisher and multiple subscribers), the system would continue to behave the same way as it would with one RDBMS server. As we all know that an RDBMS is a transactional data base that upholds ACID guarantees, transactional consistency means that these servers would do the same.

2. Merge replication: In case of Merge replication, as the name suggests, the publishers as well as different subscribers merge their data. Consider a situation where the publisher takes a snapshot of its data and shares it with the subscribers initially. Once that is done the subscribers may go offline, update their data, go back online and not only receive but also merge their changes with the current publishers and subscribers.

Transactional replication is used in the following circumstances:

1. When we want the changes to be propagated from the publisher to the subscriber as they occur in real time. This is usually the case in applications that require latency between the publisher and the subscribers.
2. When changes to the data are very frequent.
3. When the data is very critical and consistency of the data is more important than performance of the overall app.
4. When the application triggers certain functionality and needs to access intermediate states rather than just the net change to that particular row.

Transactional replication is usually the first step. When in an RDBMS environment, it's the least unobtrusive way to go from one server to multiple servers. If we are building an app with critical data, for example, an e-commerce application, a health care application, a payments API, or any other critical enterprise data, it makes sense to use transactional replication for that particular piece of data. In traditional computer science terminology, sometimes the publisher is also known as the master and subscribers are known as the slaves. In this kind of replication, the publisher is invoked for all the writes and the subscribers are updated by the publisher in real time.

One of the major disadvantages of this approach is performance. Imagine having one publisher and many different subscribers. In that situation, it will be very hard to scale. In fact, after a certain number of subscribers depending on the app load, it might not be a viable option. Imagine being an e-commerce company that does thousands of transactions per second on a peak day like Thanksgiving. Even though it's a good step toward adding redundancy and data location and helps back up your data in real time, in may not be the most suitable way to deal with applications that require tremendous scale. That said, it still takes us very far and lets us build a system that now does not have a single point of failure and can be scaled more than anything we've seen so far.

Merge replication is used in the following cases:

1. When the application cares about the net change rather than incremental state changes.
2. When the application cares about availability and performance more than consistency of the data. The data will eventually be consistent, but that may mean a certain read from a subscriber that hasn't been updated may not have the most current copy of the data.

Merge replication has some disadvantages too. One of the major disadvantages is making sure that there are no conflicts during merging and, if there are, resolving them efficiently and accurately. Another big disadvantage is the presence of inconsistent data in real time. So even though it makes sense for a lot of applications, applications requiring real-time consistency are not suitable for merge replication.

Snapshot replication: Both merge and transactional replication use snapshot replication for initial data, which simply means taking a snapshot from the publisher initially and copying it over to the subscriber. Later, the changes are propagated according to the type of replication. Some RDBMSs classify snapshot replication as an altogether different category.

The architecture diagram looks like the following as shown in Figure 3-6.

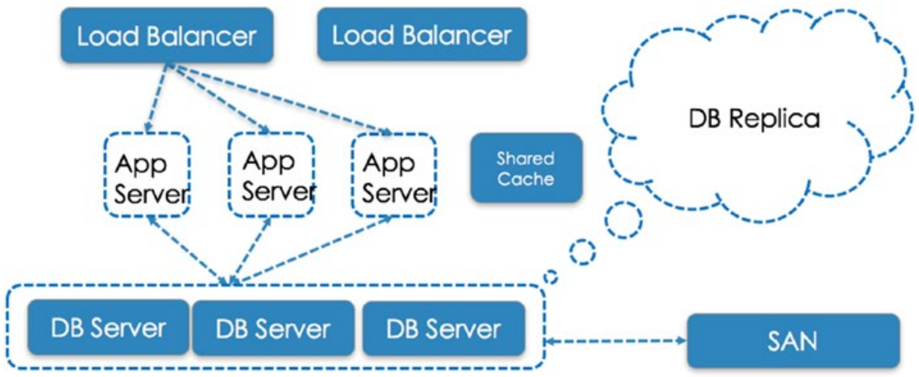


Figure 3-6. *Diagram showing the replication of relational database*

Now let's evaluate the architecture in terms of scalability. These are database replicas and they have a very decent advantage when it comes to scaling the database. Failure of one instance, be it the publisher or one of the subscribers, will not bring the application down as the database has been replicated in near real time. This, definitely, is a step forward in making sure that our system is scalable.

However, we still haven't fixed the problem of scaling beyond the capacity of an individual server. Beyond a certain limit, no increase in hardware will help increase the performance much. The disk I/O won't be able to keep up with the increase in CPU and eventually get to a point that we will need to scale beyond that one database. Keep in mind these are just replicas and will have the same set of data. This is not the same as scaling out web servers. When we scaled out web servers it was very clear that they would service similar but different requests. In this case, that is simply being replicated to all the servers. It would be a completely different thing if all the servers had different data. Ideally, that would mean a real scale-out situation. This is not referred to as a scale-out.

What about the addition of the SAN? It's surely helpful, however, it's just vertically scaling the hardware as we have seen previously. Any amount of vertical scaling or scaling up has a limit and as the application grows, we are bound to reach a point of contention, beyond which we won't be able to serve any more users effectively. An ideal solution would be to have the ability to add any number of databases (that can be replicated separately for redundancy) that keep data that is different than other databases without any dependency whatsoever on other databases so that addition of new databases can cope with the increase in load. Quite clearly as of now, our architecture is not capable of doing that.

RDBMS Horizontal Partitioning

As discussed earlier, horizontal partitioning is when we have the same number of columns in a table, but we choose to move some of the rows to a completely different server altogether. Traditionally, many companies employ this model for line of business applications. For example, clients that have a lot of data could have a stand-alone server whereas certain other clients could be so small that they may end up sharing a database server. In case of customers that have a dedicated server, over a period of years the data might grow beyond a single server. In that case, as mentioned earlier, we can start breaking the data down based on years or quarters or even months. The decision of choosing the breakdown criteria solely relies on the business application and the current and past usage of data.

For example, there could be a case where customers do not want to retrieve data that is 10 calendar years or older. At the same time, they may want to create static reports from data that is 2 to 8 years old. But they may have a serious need of working very actively with data that is less than 2 years old. In such a case it makes perfect sense to divide the data in three different tiers. These are 10 years or older, 2–10 years, and less than 2 years old. Every single tier could be in a completely separate database cluster. Since these are calendar years, the classification may not change as frequently. Application code can have a lookup table that lets them know where exactly to query. In some cases, it might make sense to have an agent or even a second-level cache like Redis or Memcached that has the lookup table in memory to speed up retrieval. When the categories are small enough it might make sense to just have this as part of the configuration file in the application or the API.

We can use multiple kinds of techniques or strategies to create horizontal partitioning solutions. The example above elucidates ***age-based or balanced (least-used) strategy*** where we segregate the clusters according to the usage pattern. If the users are prone to accessing new or older data in the same priority, we can also use a ***round-robin strategy***. In this strategy, data gets added on a round-robin basis in different clusters. Another well-known strategy is ***first-come first-served***, and as the name suggests, it segregates data based on a first-come first-served basis. In our example, if we have multiple clients and we choose to add the first 100 clients to one cluster and keep adding every set of 100 clients to subsequent clusters, we shall be following the first-come first-served strategy. The first-come first-served strategy is very similar to value-based strategy except that the ***value-based strategy*** is a little bit more flexible and can work on any particular value. In a value-based strategy, assuming that every client has an ID and clients with ID 1 to 100 are in one cluster, 101 to 200 are in another cluster. However, as mentioned, value-based strategy is a little bit more flexible and can work on any available value. In this case, we can take any column and use that to generate a value classification. For example, we may want to segregate data by the name of the state. In that case, we will be creating a cluster per state or a cluster for multiple states depending on the business needs. Lastly, we can also use a ***hash-based strategy*** that uses a custom hash function that decides which cluster the user data should be written to.

Below in Figure 3-7 you can find a simplified diagram that illustrates horizontal partitioning of the database.

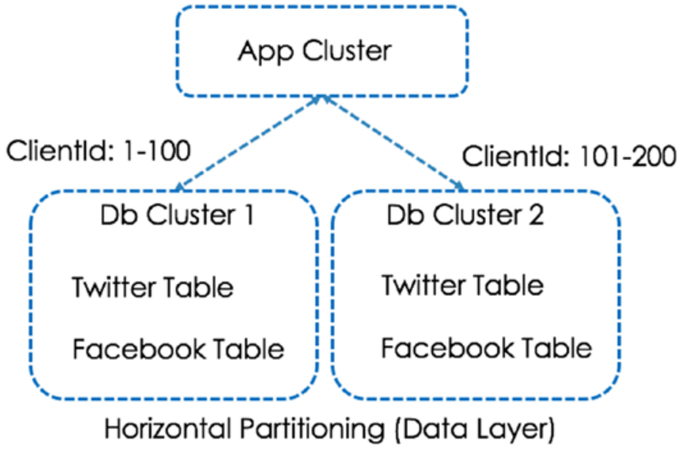


Figure 3-7. *Diagram showing two different clusters each having data for 100 clients*

At first glance, horizontal partitioning might seem to be selectively simple and attractive. However, we live in a world where business changes more often than we think it does. Constant changes redefine our architecture and even though we may have used the best possible horizontal partitioning mechanism, we cannot guarantee that our solution will be the best solution for all future purposes. Let's assume that client No. 10 acquires client No. 200. The transition is not that hard but may still require some manual work like making sure that there is no conflicting data. Since after merging the data, the client No. 10 would be the only client appropriate tests need to be done to make sure we don't end up overwriting some referential data. What if there is a business case to add shared data among different clients based on a new business requirement. Where do we keep the overlapping data? You will be surprised I have seen production systems where clients have chosen to keep two copies of overlapping data (in both the client tables) and have decided to update both the copies with every single write. Even in a simple scenario like the one above, every single write call that affects overlapping data in one cluster would require another call to the other cluster and will slow the system down. What if we have multiple such clusters and multiple clients with overlapping data? That will mean not just slowing the system down but potential for data inconsistencies. The more the clients to be updated, the more we run into distributed system related problems too. What if the data is in different servers geographically? Anything is possible. For the most part, if horizontal partitioning of the data layer is not done very smartly or in a futuristic manner, it's nothing more than a hack and is bound to create problems.

RDBMS Vertical Partitioning

Whereas horizontal partitioning keeps the same table structures but moves rows to a completely different server or cluster, vertical partitioning is a strategy in which we keep different tables in different clusters. Even though it sounds very simple, it requires a lot of domain knowledge and usage patterns of the application to create a sound vertical partitioning strategy. Depending on the business needs, we can easily find tables that do not interact much with other tables and place them in one cluster. As we discussed above, in row splitting we create new tables by separating out certain columns from the original table. One of the good ways to partition vertically is by gathering enough data to find tables that have slow scan time. Once we have figured out which tables need to be scaled vertically, we need to look at the data and find which columns are the ones that are accessed more frequently than others.

Let's take an example of the table *BlogPost*. Most of the searches might only search either the *postID* or the column *Post*. What if we have blog posts that are quite lengthy? Most of the times searches will be about certain keywords from within the blog posts. In an ideal scenario, it's wise to use a search engine like Lucene, solar, elastic search, etc., for search functionality. However, for now, let's assume that we are not allowed to use any external search engine. Well, in that case, we will need to work with what we have. We can create an index column that has the indices and a link to the blog post and move the blog posts out in a completely different table.

What about the pictures in the BlogPost table? In an ideal scenario, we will use some kind of blob storage outside the database. If it is public, it might make sense to use CDN. However, assuming that we are not allowed to use any external tool or CDN, we can use vertical partitioning and move the pictures to a completely different table and reference them via relationships. One thing to keep in mind is that in case of vertical partitioning, the new table will have a completely different name altogether. Later, when we end up doing some kind of hardware partitioning, it's important to categorize tables per disk in order to get optimum performance. One of the common practices is to colocate tables in a cluster that are related. For example, old e-commerce data could be in a completely different cluster as compared to verbose data like blog posts, and comments, etc., could be a completely different cluster.

Even though we can solve a lot of these problems in a much better way with hybrid persistence models, commonly known as polyglot persistence, we're trying to work under a constraint (which is scaling an RDBMS). Figure 3-8 illustrates horizontal partitioning of the database.

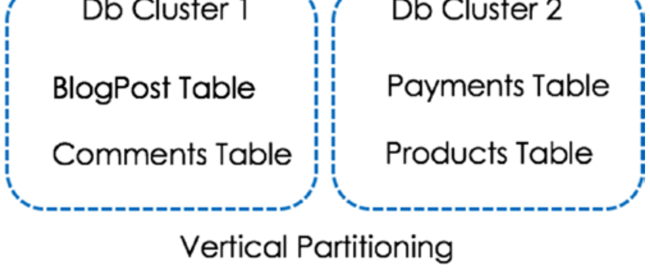


Figure 3-8. *Diagram showing vertical partitioning of data by adding different tables in different clusters*

Even though the structure looks very simple and has its own benefits, including helping with scalability, it has certain downsides. One of the major disadvantages of this approach is having to do cross-cluster joins in situations where we have relationships that exist in separate clusters. For example, if we have a User table in database cluster 2, that means we will need the AuthorID in the blog posts table as well as the user ID in the comments table to be referenced to the users table in order to get the identity. Well, if that's the case why can't we move the user table to cluster 1? We surely can. However, that will create another problem for users in the payments table as well as other tables like order, customer, etc., that are part of e-commerce business for the company. Now they will need to do a cross join to cluster 1. Both of these could be very inefficient. Of course, these are just simple examples of the kind of things to look for before making a decision on how to segregate the data. In many cases, vertical partitioning is very fruitful for scalability purposes. However, if the strategy is not fully vetted and sorted out, it can lead to problems like the ones discussed above.

In complex domains, both horizontal and vertical partitioning strategies can be of limited help just because of the complexity of the data involved. They can still take us along way before we run into problematic situations.

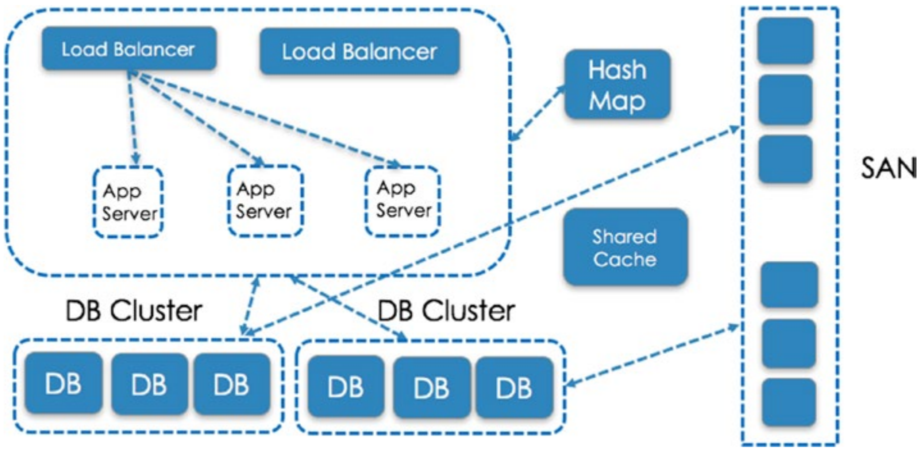


Figure 3-9. *Diagram showing DB clusters extended by SANs*

At this point of time our architecture looks something like Figure 3-9 above. Just as a clarification, we need the global hash map so that the app servers know which Db clusters to call for a particular kind of data.

Set – A Unit of Scale

With a solution we have come up with so far, we can easily say that it's a very scalable solution. What we want to keep in mind is that the data bases are not scalable beyond a certain point because we're using relational databases. However, they clearly are not a single point of failure. In my consulting experience, I see this kind of architecture all the time and it's just a matter of time that the diagram above becomes a unit of scale and companies create a maintenance nightmare by multiplying that unit of scale over and over again. In Figure 3-10 below you can see that we have taken whatever we created in step No. 9 and made it a unit of scale, which is called a set. With the kind of users we have, this set could serve millions of users. When we get to the point that we need to expand beyond that set, we can use another set for a different set of users. This architecture can definitely keep it going for a while but it's not the best architecture. It's not that this architecture does not work; in fact it works quite well for many cases. However, as an architect we have a responsibility to create something that is very flexible, scalable, performs well, is easy to understand, has less problems, and lasts a lot longer than an architecture like this.

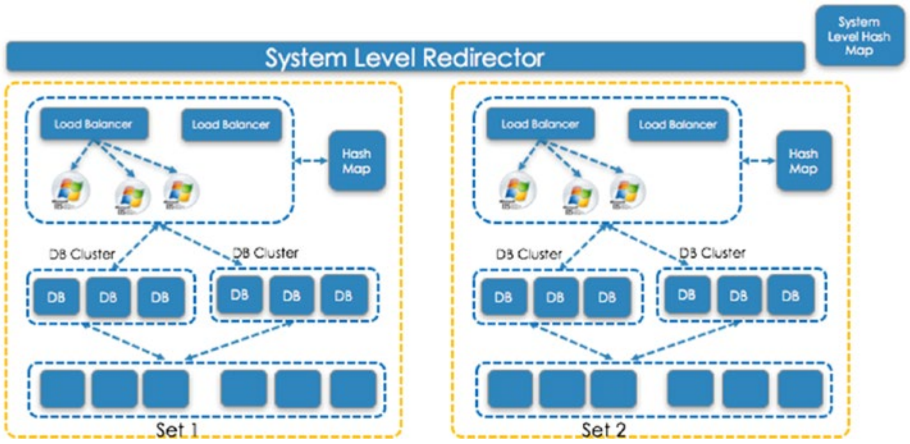


Figure 3-10. *System-level redirector balancing traffic between sets*

Let us evaluate our current architecture.

1. Load balancers are fully scalable and are not a single point of failure.
2. App servers are fully scalable and are not a single point of failure.
3. Shared cache is fully scalable and is not a single point of failure.
4. Databases are definitely not a single point of failure. However, it will be wrong to assume that they are fully scalable.

Let me make an attempt to explain point No. 4. One of the biggest constraints we've had so far is that we are only using one kind of database: relational databases. The problem with replicating and partitioning relational databases is that you can only go so far until other problems like performance start occurring. We have already discussed the disadvantages of vertical and horizontal partitioning, especially when it comes to overlapping data between two or more tables that could be placed in separate clusters. It's not just a performance hit, it's also a maintenance nightmare. Imagine having to deal with all that logic in your application. Imagine having to duplicate data and storing it in multiple locations. Imagine having to have offline processes that check for the validity and congruency of the data. The list can go on and on.

Conclusion

We have come to the point that we have an architecture that has the potential to scale for most Enterprise applications out there. The architecture has successfully gotten rid of all single points of failures and has the potential to scale to millions of users a day and be able to store an incredible amount of data. For a lot of business-to-business applications, this could be considered a close-to-perfect architecture. However, the architecture has a constraint that the system of record happens to be an RDBMS and there's a reason that it has been highlighted quite often in this chapter. In the recent years, scalability needs have increased drastically but most traditional architects have have continued to just focus on scaling an RDBMS. At the same time, the industry has quite successfully moved to polyglot persistence. In my personal experience, any successful system of scale has been built on polyglot persistence. Relational databases are simply not a panacea to all ills anymore. They have their place but there's a lot of room for adding different types of NoSQL databases. We will now pursue our journey toward creating an architecture that is overall better, more scalable, has higher performance, is easy to use and understand, and quite surprisingly is overall much cheaper.

CHAPTER 4

Concepts We Tend to Ignore

Before we get to the next level and create the best architecture, it's advisable to look in to some concepts that we tend to ignore. By no means it's a complete list of concepts that are needed to create a scalable architecture. The idea behind this list is to create awareness. I would also like to stress the need for looking out of the box for similar concepts and ideas that keep coming in with new technologies, frameworks, hardware, and technological paradigm shifts that occur along the time. For example, we won't be talking about security at all. This does not mean security is not important. In fact, it is the most important part of any application. However, since the focus of the book is on scalability concepts we assume that readers will take care of important, but unrelated concepts like security.

It might be possible that most people in our team are already aware of most, or even all, of these concepts. That still may not mean that all of these concepts get incorporated before making decisions about scaling our applications.

Async Non-Blocking I/O

When Node.js was launched it became popular overnight. It was based on one strong fundamental that any time we make an out-of-process call, we should not be wasting the hardware resources and instead free up the thread to do other things. It was interesting that even though the operating systems supported that for a while, there were not many web frameworks taking advantage of it. The ones that were incorporating this aspect were different than Node.js because they were not async-first.

Even though it was asynchronous in nature and was specifically non-blocking when it came to I/O (Input/Output) calls, there was some misinformation out there where people mis-understood the framework to be non-blocking, even for computation. That was simply disinformation. The website specifically said ***non-blocking I/O*** but apparently some bloggers misunderstood the concept. Node.js used one thread and that will make sure the thread will block if it's doing a computation intensive job. So let's understand what async non-blocking I/O means.

I/O is simply your first interaction with any out-of-process call, be it an API call, a call to the system's disk such as file system access, or call to a database. Node.js, since the beginning provided async methods that would not block the event loop when an out-of-process call used to be made. This had the advantage of the thread being available for other operations and not having to wait for the process calls. This was incredible for making applications that are scalable without much effort as there were a lot of async operations in the standard library itself. When the process call completed, the thread would pick it up and send a response back to the user.

If we were to make an attempt to compare Node.js to contemporary web frameworks, it was easy to say that other web frameworks may have support for asynchronous programming, but they were not async-first. Most of the contemporary web frameworks would work with multiple trends at the same time, taking advantage of multicore CPU servers. However, whenever a call was made, by default, it would be a synchronous call. That means the thread would block until the call was completed and then be available to do something else. Unless someone took advantage of concurrency-enabling libraries in those frameworks, the calls would be blocking by default. A very good analogy to describe this is someone cooking food in a kitchen. Imagine the chef is the thread in question.

Traditional frameworks blocks are such that while you are boiling food in the kitchen, you are not allowed to do anything else. Once the food is boiled, you're allowed to cook vegetables and then eat them. However, in Node.js, the chef starts to boil the water and goes on to read the recipe while he chooses to cut the vegetables or even open the door for a visitor to come in. We all do not want to be the chef in traditional frameworks for obvious reasons.

Even though Node.js was async-first, it did not mean that the code never blocked. In fact, the code would block if someone used the synchronous version of the method instead of an asynchronous one. It could also block if someone ended up writing a computation-intensive method. For example, the calculation of Fibonacci series or any other algorithm (that is computation intensive) would completely block the thread.

Note It's a misconception that blocking a thread means blocking the CPU. Usually the threading logic gets delegated to the operating system. At the same time, developers need to be careful while altering the thread priority.

So, if someone is using Node.js but introduces blocking code, it may slow the website quite a bit. It's not that hard to mess up the code to a level that it also affects scalability. Let's take an example where we have 10 different API calls that take less than 100ms response times. Right after the thread makes these calls, it will be freed up to do something else. At this point of time the thread has to do a computation-intensive job that takes, let's say, 2 seconds. The API calls were made asynchronously and we should be getting the results back in about 100ms (assumption). What would that mean for us? That would simply mean that even though we have the results back in about 100–110 ms (approximately), we will have to wait another 2 seconds just because the thread was blocked owing to the execution of the operation. If we were to ignore the latency from the server to the user for now, it still means that the calls (that could have gotten back to the user in roughly 100ms) would take 2.1 seconds minimum. To make matters worse, what happens if that process takes 10 seconds? An even worse situation would be the thread dying in the middle of a critical operation. All it takes is one developer adding synchronous operations here and there in the code, and the entire web server we'll be delaying the calls.

It's not quite uncommon to see these kinds of practices in real production code. Another real question is why are we discussing all this? Given the real-world experiences I've had with multiple clients, choosing the right technology and the right framework is just the beginning. There's a lot more to creating a full-blown scalable application. Whereas an async-first framework is very helpful to help scale applications, not adhering to the right practices may mean not getting the expected results. It's highly critical to make the right decisions and then make sure they're implemented under the right guidance.

Even though being single threaded could have been looked upon as a huge disadvantage of Node.js, just by using load balancers we could fix the problem. Rather than using multicore servers, we can now use commodity-level hardware and scale the application. What happens to competition-intensive Jobs? There are different ways to handle that problem, but one simple way is to use an API that handles those jobs independently. If needed, we can also use a language like F# that is specifically designed for high computation and can be accessed via an API that talks to the Node.js server.

Here's a simple illustration (Figure 4-1) that just focuses on scaling a Node.js application and delegating computation-intensive jobs to that F# server.

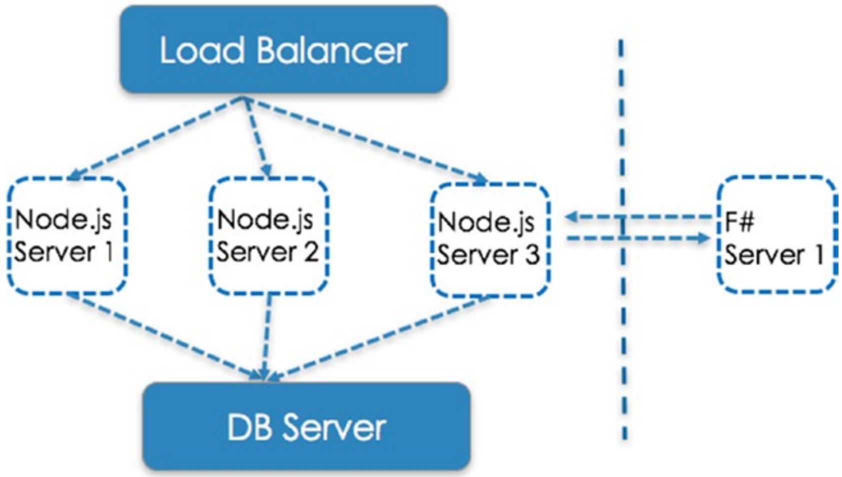


Figure 4-1. *Node.js application delegating computation-intensive jobs to F#*

Recently with the introduction of .Net core, an open source framework, Microsoft did a great job integrating the benefits of a popular framework like .Net and an asynchronous framework like Node.js. .NET Core took inspiration from Node.js and is a very lightweight server. Like Node.js, it adds all the functionality using packages. This is a much better approach than the traditional monolithic .NET framework, which was bigger in size to begin with. It came preinstalled with a bunch of functionality that may not be needed for every app out there. With .NET Core, users can add lean libraries as they go and make very lightweight applications. They also created asynchronous functions for out-of-process calls for every synchronous function in the framework. This framework is faster than Node.js in certain cases and slower in others. But for a lot of developers who like to code in the enterprise as well as on critical applications, C# ends up being a much better language for the server compared to JavaScript (Node.js code is written in JavaScript). C# allows compile time checking and is a full-blown object-oriented language like Java. That said, both these frameworks are very popular and have their own benefits as well as downsides. At the same time, both these languages are extremely popular and have their own following.

Microsoft's dedication toward async-first and creating a lightweight framework is in line with the architectural principles Node.js was built upon. That, in itself, is a validation that the industry is moving toward such principles. The major takeaway here is that our job is incomplete with just the selection of the framework. The bulk of the work is in creating guidelines to make sure every single developer on the team writes code in the right manner. More important than enforcing guidelines to write code correctly is to enforce guidelines to not write code incorrectly. More often than not, it's the incorrect way of writing code that fails projects. It's the technical debt that is accrued over a period of time that is the biggest threat to the scalability of the app as well as the future of the project or the business.

Caching

Our conversation about scaling the persistence layer has been restricted to scaling the relational databases so far. Even though caching might seem to have a benefit of just speeding up the system, one of the major benefits of caching, in practicality, happens to be decreasing the load on the major database. In our case, it happens to be a relational database. Since it's very clear by now that we can't scale out relational databases beyond a certain limit, it makes sense to reduce the load on it. One of the other major benefits of caching is storing data close to where it's called from. This helps speed up retrievals and enhances the user experience. To recap, the major benefits are the following:

1. Speeding up the system.
2. Storing data close to the client.
3. Decreasing load on the actual system of record.
4. Auto-scalabilty benefits of the second-level caching engine.
5. Reuse of the caching engine by multiple applications, APIs, or web services.

Although caching has some significant advantages, one of the caveats of caching is the fact that the application needs to have a way of managing stale data and refreshing the cache. In certain cases, the application might end up displaying stale data, too. That means caching cannot be used for all sorts of use cases. However, it makes perfect sense for certain use cases. Imagine loading all the pictures from a CDN and then caching them on the browser. This will be an example of a UI cache. Imagine loading all the product specifications from a second-level cache server. This will be a perfect example of API cache. Why product specifications? When was the last time the product specification for a particular model of a product changed after the product was released? Almost never. For example, a 24-inch monitor released for production in 2017, when sold in 2018, will still have the same specifications for that particular model number. The bottom line is that it makes sense to take such data and cache it. However, there are examples of data that could make sense for a certain period of time and would not often change but might still have the potential to change. That data can be cached as well. Lastly, we also have database cache that could mean going as far as querying some, most, or even all database queries in memory. Again, this totally depends on the business case.

Different kinds of caches are listed below.

1. Query cache
2. Object cache
3. Session cache
4. API cache
5. Page cache

Query cache, as discussed earlier, is a common technique of caching the query results. As we know, databases are persisted on the disk. Caching some queries in the RAM of the database server makes the retrieval faster. The application can also query the database and choose to cache the query results on the application server. This would mean using the memory of the API or the application server to store these results.

Object cache is mainly a cache for application objects that are stored in memory. The retrieval is faster that a cached query. If there is a clear separation of concerns in the application architecture, for example, having an API that has a business logic layer, data access layer and the caching layer, it makes more sense to cache objects than a query sometimes. In case of a query we may retrieve data and then manipulate the data before we create what is known as a data transfer object (or DTO). It's not the queried columns but usually a subset of it that makes it to the DTO and then is consumed by the User Interface. If we cache the query, we will still need to manipulate the objects before we have the final object. However, caching exactly the object that is needed by the UI can speed up retrieval.

Session cache is when a user's session information is stored. A second-level cache engine is a very good place to store a user's session data. Making a call to the database on every request increases database load, reduces performance, and eventually, reduces the quality of the user experience. If it's stored in a cache the retrieval is faster. For scalability purposes it's a great thing to introduce as now we have reduced not just the additional data that gets added to the database but also the number of calls to the database. Why cache the user session? As we all know HTTP is stateless, but we need the state to improve the user experience. So, we use the session with a configurable timeout to make sure that the user has a seamless experience. Session information is stored in order to make sure that the user is identified and served correctly by the system regardless of the actual server serving the request behind the scenes.

API cache is when we cache responses at the API level. GET requests can be cached but POST, PUT, and DELETE cannot be cached. It might be valuable to cache resources that don't change that often. Anything from text, pdf, images, and even videos can be cached at the API level. There are times when for reasons, especially security, that some companies prohibit the use of GET. In such cases, the retrieval of resources is also done via POST, which cannot be cached unfortunately. API cache is usually implemented using a gateway cache or a reverse proxy. In order to invalidate the cache, there is a purge request to the proxy that notifies it to remove the resource. Note: The proxy must be configured to handle this method and actually implement the logic to remove and update the actual resource in question.

Page Cache is when the entire page could be cached and served as if there is no change. A good use case of something like that would be a documentation website like Microsoft MSDN. It has documentation for a lot of frameworks as well as technologies. Very rarely the documentation would change. So, it makes perfect sense to cache the entire page. It's mostly suited for a write-once, read-frequently kind of a paradigm. If the page is write-heavy or prone to updates quite frequently, this might not be a great approach.

Data Categorization

Now that we have walked through some of the different kinds of caches, we need to take a look at data categorization. Data that could be cached is categorized under three different types:

1. Reference Data
2. Resource Data
3. Activity Data

Reference Data happens to be heavy on reads and with almost little or zero updates to it once it's part of the system of record. It's also data that allows concurrent access for the most part. Good examples of this kind of data are product descriptions, static content like blog posts, questions in a survey, and similar data.

Activity Data happens to be both read and write data. However, this data is the result of user activity for the most part and is specific to the user. Some examples of activity data are shopping cart content, comments added by the users, reviews made by the user, and even responses to questions or surveys.

Resource Data happens to be both read and write data, but unlike activity data, this data is shared between different users. One of the good examples of resource data could be the number of units in stock. Another good example is the final rating for a product based on data from multiple users.

Caching Guidelines

It's important to understand caching guidelines well before designing an architecture. Very minimal errors like categorizing data wrongly, not caching data appropriately, failing to purge the cache appropriately, and sometimes caching data beyond a certain time limit could prove to be very costly for business.

1. Key is the key: When caching data, it's simply a bunch of key-value pairs. For a second-level cache like Memcached, which is super fast and incredibly scalable, it accepts data that is a bunch of string key-value pairs. For something more involved like Redis, it supports a lot of other data structures including hashes, lists, sets, sorted sets, etc. When retrieving data, it's advisable to use the minimal but complete set of unique identifiers we need to retrieve that data. The reason it needs to be minimal is because keys consume space in the cache, and if we don't craft them smartly it will lead to reduced performance over time.

2. Caching the right data: This could be understood better by the use of a real-world example. For an e-commerce website, we may have a service that provides the recommendations for a particular user. Recommendations might be further divided based on a particular category, for example, electronics, books, videos, etc. This could be further divided into subcategories like laptops, tablets, and phones with the Electronics category. For the first time when the user makes a call, we might get the generic user preferences and cache them. Let's call it UP1 which stands for User Preferences for user No. 1. Once he selects his category (Electronics), we may be able to retrieve U1-Electronics preferences and now cache them. We can now choose to cache UP1 and U1-Electronics separately as well as merge them together in a separate category called UP1:: U1-Electronics.

   What's the benefit of this approach? Let's say the user now picks a completely new category like Video. In this case, we will only go to the database for the U1-Videos but not for UP1 as we have it in cache already. After some time, we will have the entire set of preferences in cache. The fundamental concept is to cache the right data or the data that is of some value to us rather than just caching an output of a web request made by the user. User patterns and behaviors change all the time so it's always advisable to consider a strategy that is flexible to work with as well as easy to change.

3. Categorize the data appropriately: First and foremost, it's very important to distinguish and categorize data that will be cached and not cached. Often what's overlooked is data that must never be cached. This can vary per business. Resources that are critical, change frequently, and are considered data of utmost importance usually don't make a good candidate to be cached. For example, account balances in a bank should not be cached. The customer can make a decision based on a cached but stale version of the data, and that could result in a not-so- great customer experience. Second, it's important to categorize the data as reference data, activity data, and resource data and come up with a customized policy on how to handle this. Invalidating the cache is extremely important and technologies like Redis provide a good publisher-subscriber model that can be used to refresh the caches relatively quickly. There are times when additional application logic is written to be more proactive in looking for changes between the system of record and the cache. This doesn't mean we need to retrieve the entire table. It could be something as simple as querying for a timestamp like LastUpdatedDateTime and accordingly flushing the cache.

4. Fall back to the system of record: What happens when, for some reasons, the cache is down or doesn't return a single record. No matter what the case, we cannot fully rely on the cache. There should always be logic in the application or the API to make a call to the actual source of record in case the cache fails. At the end of the data, the cache doesn't guarantee data retention and should never be relied upon as the system of record.

5. Be careful with the type of data stored in the cache: The general guidelines are not to store important data in the system of record and refresh the cache from it. However, there are times when it might make sense to add some data in the cache temporarily, especially data that might not need to be stored in the database permanently but might make sense for some user interaction temporarily. For example, temporary settings based on the session information of an authenticated but not authorized user, could be directly stored in the cache. Since, the data is linked to an anonymous user for whom there is no actual record in the database, the temporary settings could be written into the cache and removed automatically after some time. That said, data that is important, costly to reproduce, has legal ramifications if lost, as well as needed for the accuracy of the business transactions should be first saved in the system of record and then in the cache.

6. Cache the smallest objects: The example in point No. 1 shows that we started with smaller settings and then clubbed them together. Whenever we need data from the cache we should be smart about how to retrieve it in a way that we get the least amount of data needed for our use. This improves the overall performance of the application. Even though caches are fast reducing the number of round trips to the cache, a cluster will significantly improve the performance of the application.

7. Do not alter data coherency to reduce the data size: Caching the smallest available objects is a good idea. However, this doesn’t mean breaking an object down to multiple small ones and then reconstructing it on the fly. There are some challenges with that approach. The actual process of combining various small objects might be flawed. Sometimes, it could be more expensive in terms of performance. One of the common examples to compare Memcached vs. Redis performance is regarding strings. Memcached doesn’t natively support different data structures like Redis does. It stores everything as a string. The problem with that is that in the real world we may need to store an entire data structure in cache. Even though Memcached may be mostly faster than Redis on retrieving the same strings as it is, it may actually be slower in circumstances where Redis might be able to retrieve the entire object as it is, compared to Memcached where we might have to run a function on the fly to convert it to the format we need it in.

Cached Item Removal

Deleting an item from a cache is as important as adding an item to a cache. There are three different ways an object can be removed from the cache.

1. Expiration
2. Explicit removal
3. Eviction

Expiration is configurable at different levels. We can provide expiration time frames at the cache cluster level. We can also choose to enforce expiration time frames per resource. So, for example, any time an item is added to the cache it can have an expiration period associated with it. Every time the item is updated it will restart the expiration period.

Explicit removal is when we explicitly remove an item from the cache. This happens when the item has been changed in the system of record and we need to remove the item and add the latest version of that item in our caching engine.

Eviction is when an object (mostly the least recently used object) gets removed due to lack of space. This could be a situation where we may lose objects that may not have actually expired. With second-level caches being automatically scalable on commodity-level hardware, these kinds of situations are rare but may still occur depending on the architecture and the amount of data in the cache. However, this is definitely a challenge with write-aside cache strategy or a caching strategy that calls for using a data store nearby for quick access. For example, caching in memory on a web server would be an example of write-aside cache and anytime the cache is full, it will start evicting objects that may not be due for expiration.

Write-aside cache could still be done using a second-level cache in the same data center as well as on local cache (caching locally at the client machine). Local cache could also mean caching locally at the web server level. However, while scaling we need to keep in mind that this will not make copies of these objects and will increase the chance of staleness. (Refer: Caching in Chapter 3). It's only good for immutable client-side objects that change quite infrequently.

Caching is in itself a very involved topic. While working with clients on critical projects we have policies that include, but are not limited to, the following:

1. Data protection especially for important cached data like user sessions, shopping cart, mainframe data.

2. Persisting cached data very frequently without losing performance benefits.
3. Publisher-subscriber model to keep the data as fresh as possible.
4. High availability and replication while keeping accurate data in the replicated caches.
5. Prevention from node failures and strategies to combat it.
6. Alternate strategies to identify, resolve, as well withstand data loss without degradation of user experience.
7. Reducing costs of high availability.
8. Graceful degradation while planned upgrades and patches to the operating system as well as the caching engine and the system.

Content Delivery Networks

In order to understand the importance of Content Delivery Networks (or CDNs) we need to understand why latency is important. A congestion window is the limit of data the sender can send into the TCP network before receiving an ACK (or acknowledgment) from the receiver. The received window (rwnd) is the limit of data that could be accepted by the receiver. TCP uses both congestion and a receive window in order to prevent congestion in the network. When the packets from the sender or the source exceed what the destination (or receiver) can handle, it can lead to network congestion. That's why the protocol introduces the limit.

This limit increases with every single pair of requests (from sender) and acknowledgments (from receiver). In order to reach the congestion window (cwnd) size of size N, the time taken is calculated using the following formula:

$$\text{Time} = \text{RTT} \times \left[log_2 \left(\frac{\text{N}}{\text{initial cwnd}} \right) \right]$$

Here, RTT means Round Trip Time.

Let's assume we are sending data between San Francisco and New York. The round-trip time between the two cities is 42ms. One of the ways TCP congestion control happens is by using the TCP Slow-start strategy. Slow-start can begin with an initial cwnd of 1, 2, 4, or 10 Maximum Segment Size (or MSS). The MSS happens to be the largest amount of data that a device on the TCP network can receive. With every acknowledgment (or ACK) the window size will be effectively doubled per round-trip time (assuming that the ACKs are not delayed). The process in the diagram is a three-way TCP handshake in which the SYN (or Synchronize) message is sent from the sender to the receiver, which sends the SYN-ACK (Synchronize-Acknowledgment) back to the sender. This is how the TCP socket connection is established. Next, if we assume that the actual request is made to the server and the server takes about 50ms to process the request. We will assume the MSS of 10 (which is the maximum allowed size). This means the server can send 14.6KB to the receiver in 113 ms. With every round trip time this number should effectively double, so you can see the data that can come in with the subsequent requests is 29.2KB and 58.4KB. If the total size of the response from the server was just 116.8KB we will need approximately 239ms. This looks like a small number but is really high when we consider the RTT time between the two cities, which is just 42ms and the size of the response is pretty minimal.

Latency is a constant and is a fraction of the speed of the light. If we assume that we have the best transmission media (out of fiber optics, coaxial cables, etc.) and there is no loss of packet due to external reasons, latency itself is a big factor in performance of the application. In this case, we only assumed that the server takes just 50ms to process the request and prepare a response. However, it's the congestion limits and the way TCP works that slowed us down. So, the key learning here is that in order to speed up responses for our users, we must make use of servers that are near the user's location. This is where CDNs come to the rescue. See Figure 4-2.

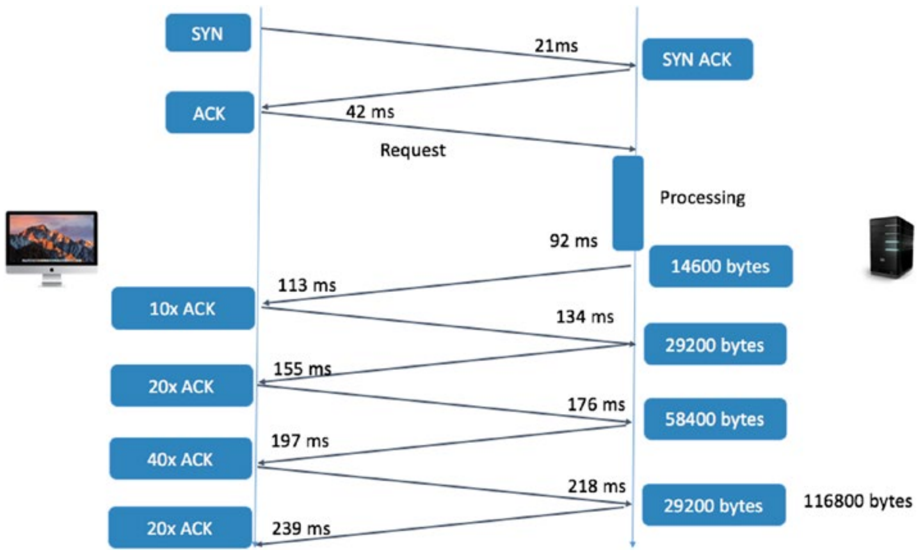


Figure 4-2. *TCP slow start between San Francisco and New York*

CDNs stores a cached version of the content in multiple geographical locations. These are also known as points of presence (or PoPs). There are quite a few CDN services that are available out there and there is no real need to buy hardware all over the world if the user base is out there. However, static content like static html, pdf files, images, and thumbnails as well as rich media could be added in a CDN and allow it to serve the user from the closest PoP.

TCP, HTTP/1.1, and HTTP 2

One of the often-overlooked strategies is of optimizing TCP. Here are some tips:

1. Upgrade server kernel to the latest version: Even though the MSS or the Maximum Segment Size had been increased from 4 to 10 a few years ago, a lot of servers (especially the ones that run legacy applications) have not been upgraded to take the advantage. In order to often take advantage of upgrades, we should upgrade the server kernel to the latest version. Usually with legacy apps upgrades are not that easy. However, if it could be done without disrupting any business, it's completely worth it.
2. Increase the initial congestion size.
3. Enable window scaling. This increases the receive congestion window size. The previous limit used to be 65,534 bytes. The current limit is one GB.
4. Disable slow start after idle. This is important because we do not want TCP two have to go through the same slow start scenario described above after it's been idle for quite some time.
5. Use TCP fast open. It allows data to be carried in the SYN and SYN-ACK packets and consumed by the receiving end during the initial connection handshake. This saves up to one full round-trip time (RTT) compared to the standard TCP.

In case of applications over HTTP 1.1, please consider making some other performance improvements discussed below:

1. Investigate DNS lookups and try to reduce them.
2. Redirects must be avoided at all costs.
3. Reduce HTTP requests by using bundling and minification.
4. Critical resources might need to be inline in case of a web application. Bundling and minification are great for most of the scripts and stylesheets; however, sometimes it's worth sending inline scripts and stylesheets for rendering the critical areas of the page without which the website may not function at all.
5. Compress text using GZip.
6. Optimize images per device. Using the same size of images for an iPhone and desktop could unnecessary reduce performance.
7. Add ETags to avoid fetching duplicate content.
8. Make sure to add an expires header on resources.
9. Investigate using local storage for stylesheets and scripts in case of a web application.

HTTP/2 provides many features that are a huge improvement over HTTP/1.1. It solves a lot of problems that were created by the previous protocol, and most of the performance hacks suggested above are not required for HTTP/2-based applications. Some of the features of HTTP/2 are new binary framing, parallel requests, header compression, server push, and stream prioritization. Binary framing is allowed such that translation from text to binary is not needed, thereby increasing performance. The server has the ability to push resources to the client.

That means as long as the client doesn't specifically bar the server from sending data toward it, the server can send additional resources to a client for future use. One of the best features of HTTP/2 is single connection, which remains open as long as the website is open. This reduces the number of round trips. It also allows multiple requests on the same connection. This feature is called multiplexing. Prioritization is another great feature in which the server can assign dependency levels to resources. This way the resources with higher priority will download first. It allows interleaving of requests and response messages. HTTP/2 supports header compression, which reduces overhead. It's a matter of time and all browsers will support HTTP/2. Currently all major browsers support it over HTTPs (not on HTTP though).

Reverse Proxy

Now that we have talked about caching, TCP, and HTTP, it might be worthwhile to discuss Reverse proxy or HTTP accelerators. HTTP Accelerators can be used to improve the response time to remote users. There are commercially available products that can serve as good reverse proxies.

The accelerators act as a forward proxy for multiple HTTP clients. By examining the headers of the resources, the accelerator can understand which objects can be cached. So, it saves static objects in memory-based or disk-based object cache. Whenever the user requests those objects, they had retrieved from the cache itself. Even though the accelerator is smart enough to cache only the objects that have the required headers (Cache-Control and pragma), it could be configured to exclude specific objects from caching.

It makes use of the HTTP header (Last-Modified) to detect if the static object is still fresh. If the object has been modified, it will retrieve the latest copy of the object and send it to the user. The configuration allows us to disable specific objects from being cached. We can configure disabling caching enforcement by modifying the conditional request from the browser. Sometimes for some kinds of static objects, including stylesheets and scripts, the browser may send a conditional request to the server. If this object is not available in this HTTP accelerator, we can modify the request and move the condition so that the server is forced to respond with the object that would later be saved in the proxy. We can also configure it to enable web server compression. Just FYI, mostly the proxy has a better compression than the web server. However, this is just one configuration option which is a good one to test before we use it. Other configuration options include flushing the cache for troubleshooting purposes. A major benefit of a commercial accelerator is the ability to access analytics on acceleration statistics.

Most of the time load balancers and HTTP accelerators would be part of the same commercial package. This could be either hardware or software based, or a combination of it. One of things to note is the use of the reverse proxy with an Async non-blocking I/O back end. Unless it's a legacy app, all new applications should use Async non-blocking I/O as it reduces the overall overhead. ASP.NET Core has recently picked up a lot on performance and the major reason behind that is it's async-first approach like Node.js.

Sometimes, the reverse proxy may also work with a lightweight server like Lighttpd to get cached static content. This way it will reduce the number of calls to the back-end server. If you're using CDNs for all the static content, you may not need something like a Lighttpd, but for certain applications and business use cases it might make a lot of sense.

IP Anycasting

When we go to the address bar in a browser and type the name of a website, let' say `http://cazton.com`, the Border Gateway Protocol (BGP) will ensure that the request goes to the Cazton.com server via the best route through the Internet. Using IP routing tables, IP Anycasting is smart enough to serve the user's request from the closest server topographically (not necessarily geographically). Typically, anycast is when a group of servers use the same IP address. Anycasting uses an IP address range that is advertised in the BGP messages and routers are made aware of the topographical path. This way the routers know of which neighbors provide the shortest path to the advertised IP address.

The major benefit of this approach is being able to add new servers and making the routers aware of the new neighbors that can serve the request faster to a certain set of users. This cuts down on latency and bandwidth costs and improves user experience drastically. The main benefits of this approach are seen when you have regions of high traffic and the servers are far apart, including across the major oceans. Apart from performance improvement and service reliability, it also balances the load and reduces the impact of DoS (or Denial of Service attack). It reduces the likeliness of the DoS attack impacting the entire service and reduces it to the local server.

Microservices

Transitioning from monolith apps to services was a logical progression. In order to have services or APIs that could communicate with external as well as internal systems, it made sense to take a particular component of the system and convert it into services. For example, in an e-commerce website it makes sense to take the payments functionality and create a completely different service that only has one responsibility, which is taking care of payments. Similarly, we can create the following services: pricing, customer, product, inventory, etc.

As we can see, this allows us to have more flexibility in having the services contract with other components within and outside our system. This also helps in scaling development and testing efforts. In large corporations, some of the services could be the responsibility of different teams. This has the potential to scale as well. However, if we do not scale the System of record, this architecture would not scale beyond the point.

Note Please refer to Figure 1-2 for more details on SOA with a non-scalable back end.

In recent years, microservices architecture has become very popular. Quite interestingly, the SOA was implemented quite differently across the industry. Micro services architecture is the natural progression from SOA toward a more well-defined architecture that is meant for scale and is lightweight. Traditional monolithic apps were hard to scale. At the same time, deployment could be a nightmare depending on the size of the system. My team and I have been invited to work on monolithic systems that took days of preparation for the deployment teams and would take hours for successful deployment. Quite unfortunately, this would mean deploying new features in a product every few weeks a very huge cost. Sometimes, this would mean being able to deploy only three to four times a year. With the microservices architecture, the same system would eventually be deployed faster and partial updates to the system could happen even a few times a day.

To be fair to SOA, it wasn't that the architecture was insufficient, it was really the way it was implemented that some monolithic projects became such. For the most part, the industry had to learn from the experience of implementing monolithic projects before progressing to a more elegant architecture. The reason I call microservices the natural progression to SOA is because it is almost like SOA done right. With the standardization of the microservices architecture across the industry, the DevOps (Development and Operations) tooling has become so much better in the last few years: DevOps is a practice that aims to unify the software development and operations in order to achieve automation and monitoring at all steps of software construction. This would include testing, integration, and deployment with infrastructure as code. The direct benefits of DevOPS include shorter development cycles, increase in deployment frequency, partial releases of the system, and a more controlled and dependable deployment cycle.

Why Microservices?

In the last decade, we have witnessed our growth of Internet-enabled devices all across the world. Be it cell phones, tablets, desktops, IOT-enabled devices, the amount of activity on the Web has skyrocketed. This means every business that has the potential to grow quickly needs to be able to scale according to the usage patterns no matter how unpredictable they are. Let's take an example of an on-demand online library that has videos and pictures. In a monolithic architecture, it might be very normal to assume that all the videos and pictures would be served from the same service. Let's call the service media service. After some time with the increase in number of users, we should be able to add multiple instances of the same service in different servers. As you can see, even if we grow the business drastically we are able to scale the service using a simple SOA architecture for services. See Figure 4-3.

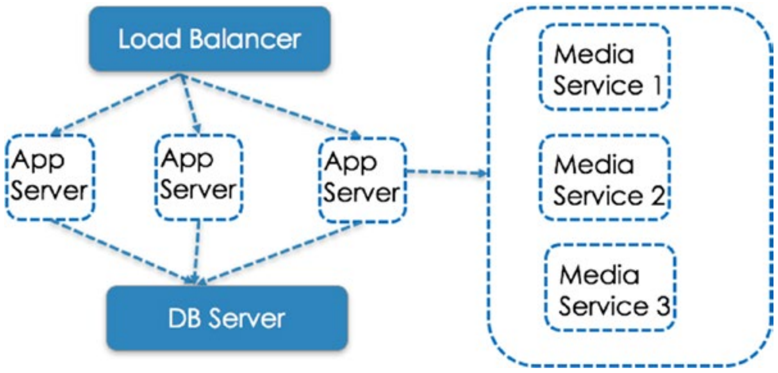


Figure 4-3. *Scaling the Media service that has video, picture, and thumbnail functionality*

Next, the business has grown, and we are asked to display thumbnails on all the videos. That will mean changing the media service to add certain functionality for the thumbnails. This means that even though the video and picture functionality was working correctly, we will still need to re-deploy the media service to add the new functionality. Let's assume that we have really long videos. That could mean that scaling the video portion is as easy as creating copies of videos close to the users. However, imagine the number of thumbnails needed for videos that are about an hour long. Depending on the operating system, if the thumbnail size grows beyond a certain number, it might mean crossing the limit of files allowed in a folder. Scaling the thumbnail is a completely different problem then scaling the video.

What about the situation where certain videos are extremely popular? In that case, we might need to look into moving such videos on faster hard drives. We might need to cache some of these videos as close to the user as possible. In the current case we have no other option but to deploy the media service as it is everywhere. What if we wrote the Media service in a language or framework that gets discontinued or wasn't async-first. The only way to take advantage of a better, faster, async-first language would mean rewriting the whole service in a different language. These are just some problems with this architecture. To summarize, it's not just a cost problem. It's a scalability issue. It's a deployment problem. It's a management problem. It's a resource allocation problem.

In the micro services model, we develop software applications that have independently deployable and modular services that serve a business goal. If, we work to solve the same problem using microservices, we would have created two different services to begin with: the Video service and the Photo service. Let's assume, that the Video service is written in .NET and the Photo service is written in Java. After some time, we could have added the Thumbnail service, which could have been written in a completely different language like Node.js.

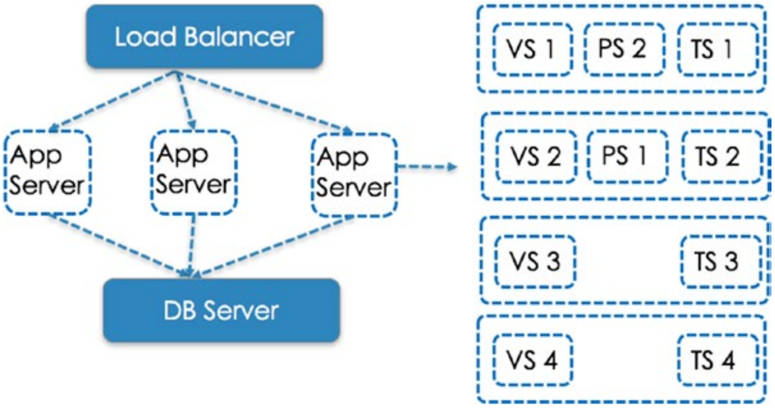


Figure 4-4. *Scaling independent microservices for video, photo and thumbnail*

In the diagram above (Figure 4-4), we have four versions of Video Service (VS1-4) and Thumbnail Service (TS1-4) each and two versions of Photo Service (PS1-2).

You can see that the benefit of microservices architecture is that we cannot only deploy but also scale the services independently. We can also use different languages and frameworks to create these services. System architecture is modular and independently deployable; we can now fix the problem we had with the videos by caching the videos close to the users. The major benefit of that approach is that it would require no change whatsoever to the photo service or the thumbnail service. Below you can find a very simplified version of the architecture where we can have multiple different clients accessing the services (Figure 4-5).

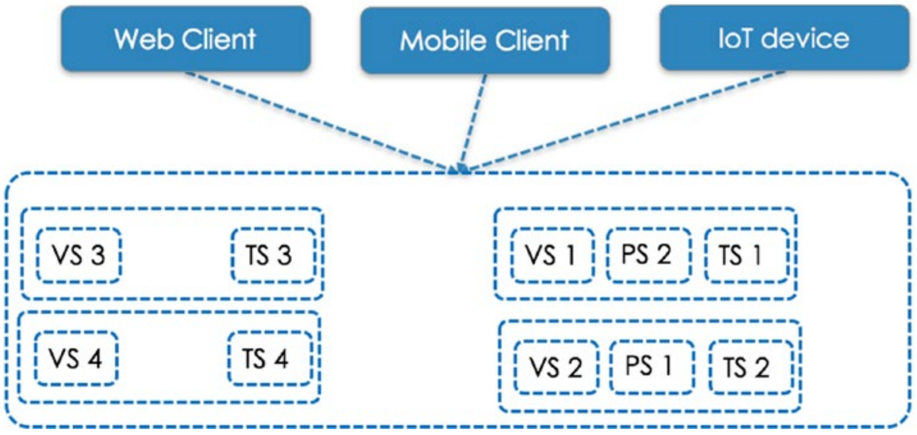


Figure 4-5. *Independently scaled microservices being consumed by different types of clients*

Another huge problem that comes in the way of continuous delivery for monolithic services (not microservices) is versioning. Let us assume that after some time, we have users wanting us to parse the data that comes from the video and display subtitles. They would also like to listen to just an audio file. In the monolithic architecture, we will need to add functionality to create an audio file and then consume it via some kind of text-parsing functionality that will also have to be a part of the media service now. In most of the cases the text parser would be a completely different package. What if the video service wants to upgrade to the latest text parser version and for some reasons the audio service cannot? The media service is still a monolithic service. It's written in one language and it uses shared packages. Even though video and audio are different sets of the same service, they are bound to use the same versions of the packages. Independent upgrade of package is simply not feasible. The diagram below (Figure 4-6) is an illustration of all the different functionalities sharing shared packages. The video and audio services are sharing the same version of the text parser.

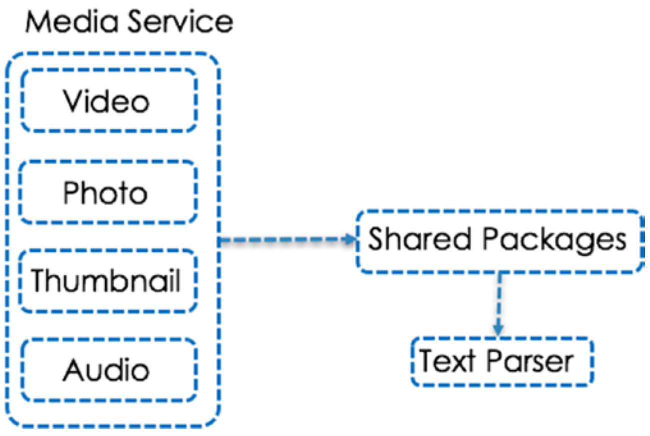


Figure 4-6. *Media service using the same version of shared packages (including the text parser) for the video, photo, thumbnail, and audio functionality*

In a microservices-based architecture, we can surely use shared packages. However, we also have the flexibility to use different versions of the same packages with different services. So in this case we can have the video service using Text Parser version 1 and the audio service using Text Parser version 2 or the other way around. The great news here is that upgrading the package for audio service will, in no way, affect the video service. This way we do not have conflicting dependencies, we can do independent deployments, we can use multiple technology stacks and scale our development efforts. The sole benefit of being able to use multiple technology stacks in itself goes a long way in being able to retain talent as well as be able to use the latest and greatest technologies to get better results. Below you can find a diagram illustrating the same (Figure 4-7).

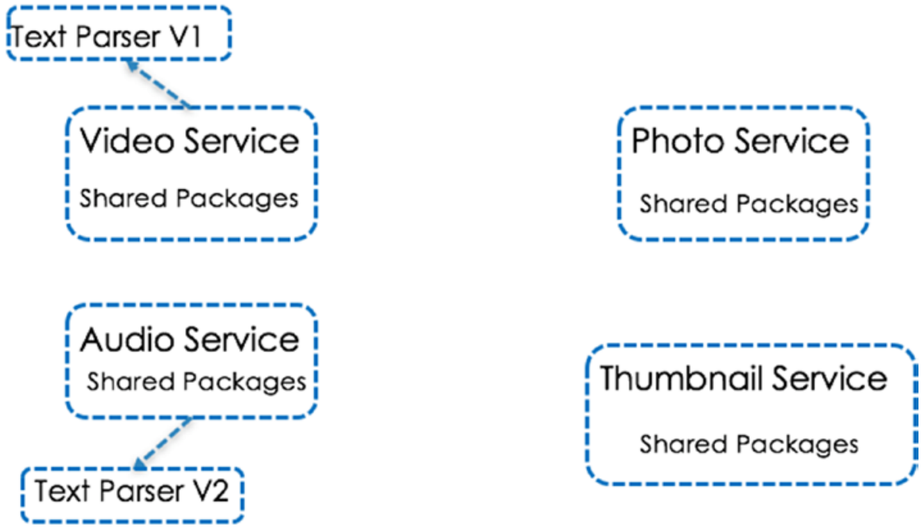


Figure 4-7. *Microservices architecture showing use of different versions of the Text Parser in Audio and Video service*

HTTP is stateless but the application does have needs for statefulness. Microservices could be stateful or stateless depending on the application architecture. Most systems of scale would most probably have both stateless as well as stateful services. In case of a cloud offering like Microsoft Azure, we can use Azure Service Fabric that allows us to create Stateful as well as Stateless service. Both the services are created similarly and have code, config, and data separated out. Based on the need of the application as well as the state of it, we can scale all the three components accordingly. Stateful microservices have no need for queues and caches and have low latency. In case of stateless microservices we need queues, caches, and partitioned storage so we can store the application state in a scalable fashion. It's strongly recommended to start with stateless microservices and introduce state as described earlier in the book rather than looking for an out-of-the-box stateful architecture. This will go a long way in making sure that the system is scalable, deployable, and will be flexible enough to newer changes in business needs as well as architecture. When in doubt, refer back to the REST principles for creating a scalable solution.

Summary

In this chapter, we discussed some really important concepts including Async non-blocking I/O, caching, reverse proxy, CDNs, and microservices. This, in no way, is an exhaustive list that you would need for performance and scalabilty but are essential concepts setting us for success. Microservices architecture has a lot of great benefits for systems of scale. Introduction of a combination of reverse proxy, CDNs, and Async non-blocking I/O reduces the system load, moves static data close to the user, improves performance, and makes the system more scalable. Caching strategy is very critical for the success of the system. It's worth spending time and devising a cohesive strategy for caching before the start of the project and then before every release cycle. This should be done for at least the first few sprints. The architect as well as the domain expert needs to be involved with developers so they can understand the reasoning behind some of the decisions.

CHAPTER 5

Relational vs. No-Sql

Quite contrary to what it sounds, ***No-Sql*** means ***Not Only Sql***. So far in the book we have been assuming that the persistence layer is only relational database or a scaled-out version of it. Before we actually talk about the architecture using No-Sql databases, it's worth understanding the differences and appreciating the different kinds of No-Sql databases out there and understanding the different strengths they bring to the table. No-Sql databases are of many different kinds. The name given to the entire category of databases is kind of unfair. In my personal experience, some very good architects have made the mistake of thinking of Relational and No-Sql databases as an ***either-or***, which is not the best way to approach architecting systems of scale.

No-Sql Databases

If you remember the discussion of CAP theorem in Chapter 2, relational databases are consistent as well as highly available but lack at partitioning (CA from the CAP theorem). In fact, when we start partitioning relational databases, we encounter many problems. It's highly disruptive to shard or partition an RDBMS. Depending on the nature of partitioning, we may even lose the benefits of a relational model such as schema consistency while partitioning. We may need to create and maintain the schema on every server. RDBMSs are also hard to scale out and scaling up is in the only option without the core properties of ACID (Atomicity, Consistency, Isolation, and Durability).

On the flip side, No-Sql databases, for the most part, are good at Availability and Partitioning (AP from the CAP theorem). No-Sql databases are easy to scale out even with commodity-level hardware. However, there is a cost to it and that is consistency. In case of a No-Sql database, copies of data are stored in different servers. So that means any time we update a particular copy for data, the No-Sql database lets the application know that the data has been updated. Let's assume we are updating a user's last name. This will mean that the last name will be updated from version 1 to version 2. At this point in time, the No-Sql database sends one request each to the other servers where the data has been duplicated so as to update those copies of data. This would mean that all the copies of the users of records will be updated to version 2. However, before these copies are updated, if the application makes a request do any of the servers sharing duplicated data, there's a chance of retrieving version 1 of the last name. This is a very simplistic example of eventual consistency. Eventual consistency is different than transactional consistency. In case of eventual consistency, there is an informal guarantee that unless any new updates are made to an item, all the reads to that item will eventually return the last updated value. In case of transactional consistency, the application will only receive an acknowledgment if, and only if, all the copies of the item (in question) have been successfully updated to the latest version.

Apart from being able to scale automatically, No-Sql databases are very good at holding structured, semi-structured, and unstructured data. Structured data is data that has defined length and format. This could include phone numbers, email addresses, and mailing addresses, etc. Semi-structured data includes XML, JSON, and similar formats. Unstructured data includes text and multimedia content like email messages, documents (or PDFs), photos, audio and video files, etc. Most No-Sql databases are schema-free or schema-agnostic. The reason is that they are built for scaling out. The idea is to be able to take schema-agnostic documents of data (that can scale out easily) that may be related to each other, but the relationships are not enforced by the database schema.

Relational databases use Normalization, which is a technique used to provide the following benefits.

1. To prevent duplication of data.
2. Enforcing real-life relationships so data is organized into logical groupings.
3. So that changes can be made in one place thereby making updates to the data easier and faster.
4. Preserving the integrity of the data.

   Why is all this important for a relational database? One reason is because relational databases cannot afford to duplicate data as scaling out is hard.

   Rather than duplicating the data in individual tables, they use the concepts of joins. Using joins, the data can now be retrieved from multiple tables that are associated via some kind of relationship. However, as mentioned earlier, since No-Sql databases can scale out easily with commodity-level machines, they can afford duplicating data. Highly structured data in case of relational databases makes sense but also adds some problems that get alleviated in case of No-Sql databases.

Highly structured data is usually the result of normalization. There is nothing wrong in that. However, it's not the best for all use cases. Highly structured normalized data can make retrieval slow. It's true that normalization makes writes or updates to the database really fast. That's because the updates are happening in the minimum number of tables if normalization is done currently. However, imagine cases that are read heavy. For example, these include product descriptions, product metadata, blog posts, and many such data items. Normalization slows retrieval with the increased number of tables involved. It's not quite uncommon to see more than five tables joined in queries. On the contrary, in case of No-Sql databases designed properly, it may just be as simple as making one query to a document. It's a very well-known thing that in critical systems, the DBAs (Database Administrators) spend a lot of time fixing performance problems on queries. Most of it is because of normalization. We are not saying that normalization is bad. In fact, it's something that is definitely required, especially in case of highly structured data in relational databases. However, there are cases where we really don't need to have highly structured data and denormalization might be the key to speed up retrievals.

Why not just use the relational database and denormalize the tables for retrieval? Quite interestingly, this happens more often than not. In certain systems, there are databases that have normalized tables and considered the source of truth. However, there is also a denormalized version of those tables for speeding up queries. This denormalized version could be in a completely different database and it could be really fast for retrieval. Yes, the solution works. However, I'd like for you to consider the fact we have created a denormalized database. More clearly, we have created a denormalized RDBMS. Bear with me and spell it out. We have created a denormalized Relational (Normalized) database management system. How does that sound to you? Denormalized Normalized database is just a hack. That's exactly the point. In order to make this work, we will still need an offline process that keeps the databases in sync. If that's the case, we could have considered a No-Sql database for this. A better solution would be using the relational database for data that needs structure and using a No-Sql database for semi-structured as well as unstructured data.

Types of No-Sql Databases

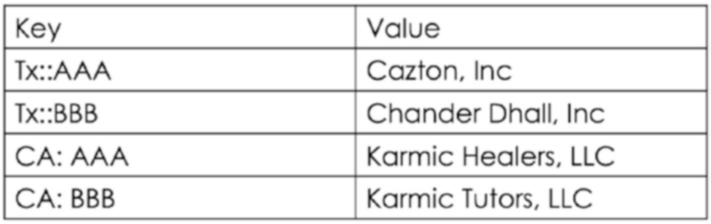
As stated earlier, there are many different kinds of No-Sql databases. They are the following:

1. Key-Value
2. Document
3. Column
4. Graph

Key-Value Store

These are schema-less, high-performance data stores with high availability. Partitioning, replication and auto-recovery are a given with such data stores. As the name suggests, they are simply a hashtable of key-value pairs, and the only possible retrieval mechanism is making a query using the actual key. In many cases, these are used as a second-level cache for the most part. These can vary from using just a string like MemcachedDB, to supporting data structures in strings or JSON, as well as BLOBs in case of Redis and Riak KV. For the most part these stores have a very simple API for the Key-Value pairs. It is getting one key using a simple Get-query or multiple keys using a Multi-get, Put-query to associate the value with the key as well as Delete-query to remove the entry of the key-value pair.

Figure 5-1 shows a simple Key-Value pair. The key is just a combination of the state and the Tax Id. The value is the name of the company. So, if someone has to create a web application for searches to be made, it would require a drop-down for state and then a textbox for the Tax Id. Once the application has both the values, there will be a function to concatenate the input after sanitizing it. Once we have the Key (let's say, TX::AAA), which is a combination of the State (e.g., TX) and Tax Id (e.g., AAA), we can get the value of my company, which is "Cazton, Inc."



Key	Value
Tx::AAA	Cazton, Inc
Tx::BBB	Chander Dhall, Inc
CA: AAA	Karmic Healers, LLC
CA: BBB	Karmic Tutors, LLC

Figure 5-1. *A Simple Key-Value Store*

Key-Value stores are capable of handling a constant stream of operations, both read as well as write, with low latency. Key-Value stores can also be configured to persist data to the disk if a certain number of write operations occur per second. So, for example, if you'd like to persist data every second if there are more than 100 writes, in case of Redis, it will just be a simple configuration. Redis would allow you to set multiple such criteria for persisting data.

Values are stored as a blob. This, thereby, removes the need to index the data to improve performance. However, the value is opaque so filtering is pretty much not an option. That's because the value is usually opaque. They can be used for saving user profiles, user preferences, session information, etc. They can also be used to retrieve product recommendations. One of the often-overlooked use case is use of a customers' profile, preferences, and shopping habits to generate a wide array of customized ads and coupons.

Note Key-Value store are like caches but are more powerful. Usually, a cache is just a read-heavy store that works in conjunction with a database. However, a Key-Value store is fully capable of persisting data to the disk. At the same time, a Key-Value store can be used for writes as well as updates. One other distinguishing factor is a transactional guarantee. Since caches are stored in RAM, unlike Key-Value stores, they are not resilient to server failure and won't be able to provide any sort of transactional guarantees.

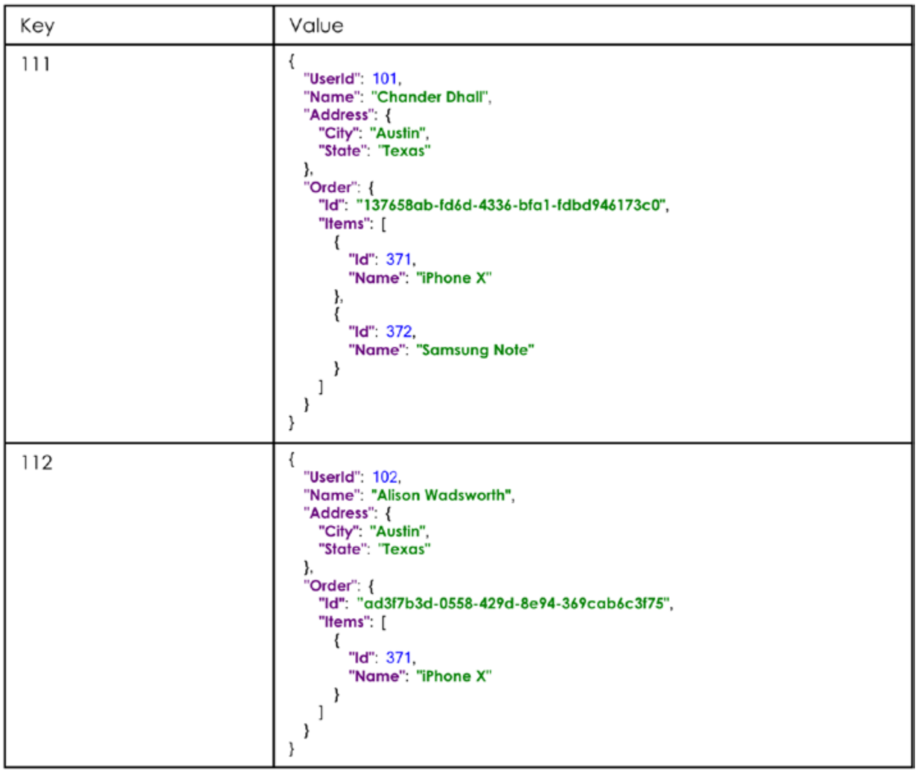
When it comes to scalabilty and performance they are probably the best. However, a big problem with Key-Value stores is that they lack most of the complex functionality compared to any other data stores. One good example is of foreign keys. It will have to be introduced by the user manually and enforced by the application too. However, it's a good idea to enforce foreign keys via the application, and it's good to have the database enforcing it too. Of course, lack of transaction capability compared to other data stores is another issue. But again, the use of Key-Value stores is for some specific scenarios and it makes sense to use them considering the low overhead.

The popular Key-Value stores may differ in the way they are implemented. For example, MemcachedDB supports simply Key-Value pairs and support ordering of keys. Redis is known to maintain data in RAM while also persisting it on disk. Couchbase Server not only stores data in RAM but is capable of supporting rotating disks. Aerospike has the capability to support RAM as well as SSDs (Solid State Drives). The latter is used to implement secondary indices.

Document Databases

These are by far the most popular of No-Sql databases. The document databases the values in documents, which are of a particular format like XML, JSON, BSON (binary encoding of JSON objects), etc. One of the major distinguishing features of a document Database is that it embeds metadata for the stored values. That makes it be able to retrieve results not just based on the query but also on the contents of the values.

Let's take an example of a dataset for an e-commerce website. Regardless of what format it uses, the data may have some entities like ***User, Order, and Product***. In turn, ***Product*** may have some other fields like userId, model, description, and cost. ***User*** may have fields like name, email, shipping address, and mailing address. ***Shipping address*** in turn could have fields like street name, city, state, and zip. If we know the ***userId,*** we can it to filter the values. However, that is the same in case of a Key-Value store. As we discussed above, we should be able to filter data based on the contents of the values. So, in case of a user, we can use the zip code, (which is part of the content of the value for the key ***userId***) as a filter. The resultant data set would be users in a particular zip code. How about finding details of all products in the zip code, city, or state? If we have the right association, the document Database is fully capable of retrieving the resultant dataset. See Figure 5-2.



Key	Value
111	{ "UserId": 101, "Name": "Chander Dhall", "Address": { "City": "Austin", "State": "Texas" }, "Order": { "Id": "137658ab-fd6d-4336-bfa1-fdbd946173c0", "Items": [{ "Id": 371, "Name": "iPhone X" }, { "Id": 372, "Name": "Samsung Note" }] } }
112	{ "UserId": 102, "Name": "Alison Wadsworth", "Address": { "City": "Austin", "State": "Texas" }, "Order": { "Id": "ad3f7b3d-0558-429d-8e94-369cab6c3f75", "Items": [{ "Id": 371, "Name": "iPhone X" }] } }

Figure 5-2. *A simple Document Database*

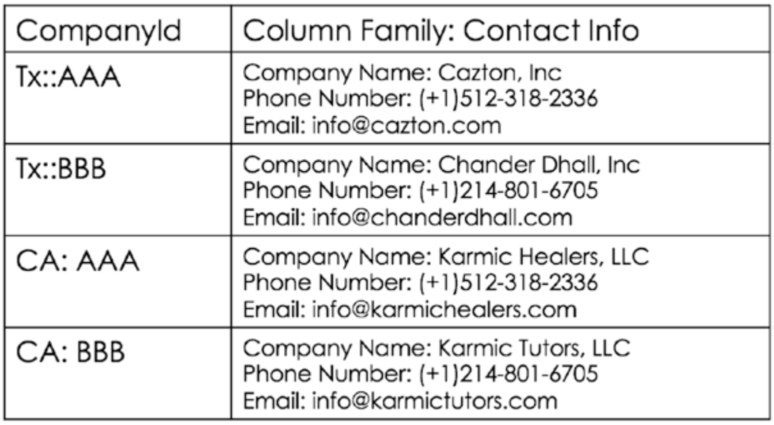
Partitioning, replication, and auto-recovery are provided out of the box with document databases. Since replication is supported, data could live in multiple servers easily. In such stores it makes sense to use techniques like Map-Reduce (to retrieve data from multiple servers and merge it before sending it back to the user). Map-Reduce is a specialization of the generic strategy split-apply-combine, which is used for data analysis. In a distributed environment ***MapReduce*** is used in parallel execution and is used for processing and generating big data sets. The ***Map*** procedure usually is responsible for filtering and sorting the data, and the ***Reduce*** procedure performs a summary operation. Behind the scenes it's more like a three-step process.

First, the ***Map*** procedure is applied by each worker node to local data. The result is stored in some form of temporary storage. The master node ensures that only one copy of redundant data is actually processed. Map procedure creates data based on output keys. Second is the ***shuffle*** phase. The worker nodes redistribute data based on output keys that were generated by the map function. Shuffling is done in a way that all data belonging to one key must be located on the same worker node. Third, In the ***Reduce*** phase, worker nodes finally process each group of output data (per key) in parallel.

Some of the more popular document databases include MongoDB, Couchbase, and CosmosDB. I had to add CosmosDB here even though it's hard to categorize CosmosDB. Given the fact that it's the only database with five levels of consistency support that I know of, this is one database that could be a replacement for an RDBMS as well as a No-Sql database. We do have one caveat with CosmosDB. Even though we can access it using .NET, Node.js, Java, and even MongoDB API, it only runs on Azure. Couchbase and MongoDB are very widely used. Recently, Couchbase made news as LinkedIn was looking at different solutions and they ended up moving to Couchbase where they end up doing around 10M+ queries a second.

Column-Family Databases

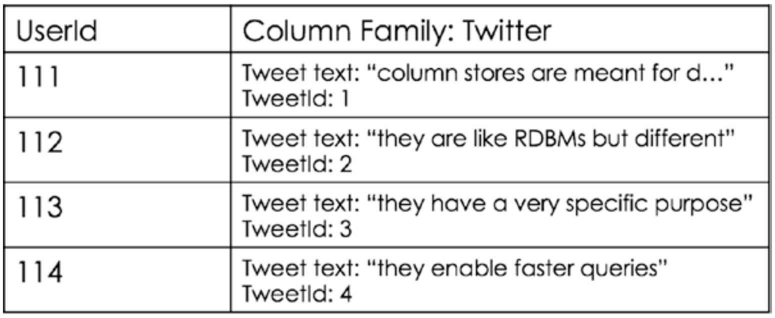
Like relational databases, data is stored in rows and columns in a column-family Database. However, data is stored in cells that are grouped in columns of data. The logical grouping of all these columns is called a column family. Column families can create any number of columns either statically (via the predefined schema) or at runtime. These columns are logically related together and are typically retrieved as a unit that correspondence to a real-world entity. The real power of the column-family Database is in denormalizing data into column families, thereby increasing retrieval.



CompanyId	Column Family: Contact Info
Tx::AAA	Company Name: Cazton, Inc Phone Number: (+1)512-318-2336 Email: info@cazton.com
Tx::BBB	Company Name: Chander Dhall, Inc Phone Number: (+1)214-801-6705 Email: info@chanderdhall.com
CA: AAA	Company Name: Karmic Healers, LLC Phone Number: (+1)512-318-2336 Email: info@karmichealers.com
CA: BBB	Company Name: Karmic Tutors, LLC Phone Number: (+1)214-801-6705 Email: info@karmictutors.com

Figure 5-3. *A simple Column-Family Database with Column Family: Contact Info*

In Figure 5-3, there is a simple column Database with a key that is Company Id and the Column Family is Contact Info. As you can see we are using the same key as before. However, the data structure of the value (in this case Column Family) is a little bit more involved. We have a Company Name, Phone Number, as well as an Email. As long as we have the key, we can get the exact detail of the family we need. We can also do the same for a different column family. Let’s say we have company employees using Twitter and we have their Twitter usernames. Since all the tweet data is public, if we want to create a column family to store the Twitter data, we can. Figure 5-4 shows how that data will be stored. A great benefit of this approach is scalability. Column family data can live separately in different machines altogether. Querying the key serves as a path to which machine the data lives in. For example, if we know the key 112 we can get the corresponding Tweet text and TweetId in this case. Keep in mind that column families can have lot more fields. We've only displayed two fields to keep it simple (Figure 5-4).



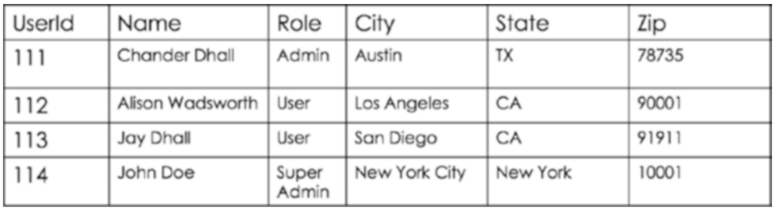
UserId	Column Family: Twitter
111	Tweet text: "column stores are meant for d..." TweetId: 1
112	Tweet text: "they are like RDBMs but different" TweetId: 2
113	Tweet text: "they have a very specific purpose" TweetId: 3
114	Tweet text: "they enable faster queries" TweetId: 4

Figure 5-4. *A simple Column-family Database with Column Family: Twitter*

A layman's explanation of Column stores is that it's very similar to RDBMS but the data is stored in columns instead of rows. This could be very misleading for people coming from a relational database as the principles are quite different altogether. Column-Databases have been inspired by the Google big table and are meant specifically to work in a distributed environment whereas relational databases are not. The benefit of having the column families is to be able to retrieve data based on it. In a relational database, that will mean a bunch of joins to get the same data. However, in this case since the data is already aggregated smartly, we just need the key to get the right column within a column family. We can see that we won't need a join to retrieve this data in case of Column-Database. Joins require indexing of the entire dataset. Rather than running the query on multiple machines separately, we are better off using the key that will point us to the exact machine where the data is stored.

In a column Database, data is stored in a column family, which is pretty much the same way it's stored on the disk. Column-Databases are definitely different from document stores. In case of document stores, we have the ability to search through the content. This requires indexing the entire dataset. Some documents store like CosmosDB index all the data by default. CosmosDB also allows the ability to turn off indexing for a particular entity.

Let's take a more involved example that will probably clarify how the data is stored in a Column data Database. Assume we have an RDBMS with the following data (Figure 5-5):



UserId	Name	Role	City	State	Zip
111	Chander Dhall	Admin	Austin	TX	78735
112	Alison Wadsworth	User	Los Angeles	CA	90001
113	Jay Dhall	User	San Diego	CA	91911
114	John Doe	Super Admin	New York City	New York	10001

Figure 5-5. *A relational database showing a User table*

As we can see there are two column families here: one is specific to address and the other to authorization.

```
{
  "chander-austin-111": {
    "name": "Chander Dhall"
    "role": "Admin"
  },
  "address": {
  "city": "Austin"
  "state": "TX"
    "zip": 78735
  }
}
{
  "alison-losAngeles-112": {
    "name": "Alison Wadsworth"
    "role": "User"
  },
  "address": {
    "city": "Los Angeles"
    "state": "CA"
    "zip": 90001
  }
}
{
  "jay-sanDiego-113": {
    "name": "Jay Dhall"
    "role": "User"
  },
  "address": {
    "city": "San Diego"
    "state": "CA"
    "zip": 91911
  }
}
{
  "john-newYorkCity-114": {
    "name": "John Doe"
    "role": "Super Admin"
  },
  "address": {
    "city": "New York City"
    "state": "New York"
    "zip": 10001
  }
}
```

Figure 5-6. *A Column-family database representation of the relational data*

As you can see above in Figure 5-6, we are using a special key here that is a combination of firstName, city, and the userId. The userId alone would have been sufficient. However, this makes our data readable for the example. If you pay close attention to the data above, it's actually not correctly formatted JSON. In fact, the absence of a comma between different properties within a family proves that it's not JSON. As you can see both the column families are flattened in the column-family database as they are. This makes the query retrieval very straightforward as well as super fast. The same data would look like the table (Figure 5-5) in case of a relational database and in case of a document database this data would as follows (Figure 5-7):

```
 {
   "chander-austin-111": {
     "name": "Chander Dhall",
     "role": "Admin",
     "address": {
       "city": "Austin",
       "state": "TX",
       "zip": 78735
     }
   },
   "alison-losAngeles-112": {
     "name": "Alison Wadsworth",
     "role": "User",
     "address": {
       "city": "Los Angeles",
       "state": "CA",
       "zip": 90001
     }
   },
   "jay-sanDiego-113": {
     "name": "Jay Dhall",
     "role": "User",
     "address": {
       "city": "San Diego",
       "state": "CA",
       "zip": 91911
     }
   },
   "john-newYorkCity-114": {
     "name": "John Doe",
     "role": "Super Admin",
     "address": {
     "city": "New York City",
     "state": "New York",
     "zip": 10001
   }
 }
}
```

Figure 5-7. *JSON representation of the Column-family database data*

This should clarify the major differences in the representation of data when it comes to Column-family database, Document, and relational databases. Even though Column-family databases are extremely scalable and have fast retrieval, thanks to how they are structured and indexing of column families, they are surely not the best data stores for supporting connections and relationships natively. Even if you were to create your own connections, chances are that they may perform inferior to a graph database, document database, and relational data Database.

Google big table, HBase, and Cassandra are examples of column Database. Most of the No-Sql databases are actually hybrids of different types of No-Sql databases. For example, Cassandra is a hybrid between a key-value and a column-Database.

Graph Databases

Often No-Sql databases are critiqued for not having joins. Do we really need joins in a flattened dataset though? Absolutely, not. However, since most of us come from a relational background, we are always looking for similar paradigms in No-Sql databases. It's really not that we want joins. What we are actually missing is the need for relationships. Even though it's possible, it may not be practical to have relationships in other No-Sql databases, a graph-based No-Sql database is the best for it. Graph databases use a graphical representation instead of rows and columns used by relational databases. Another major benefit is that the schema is not enforced rigidly either.

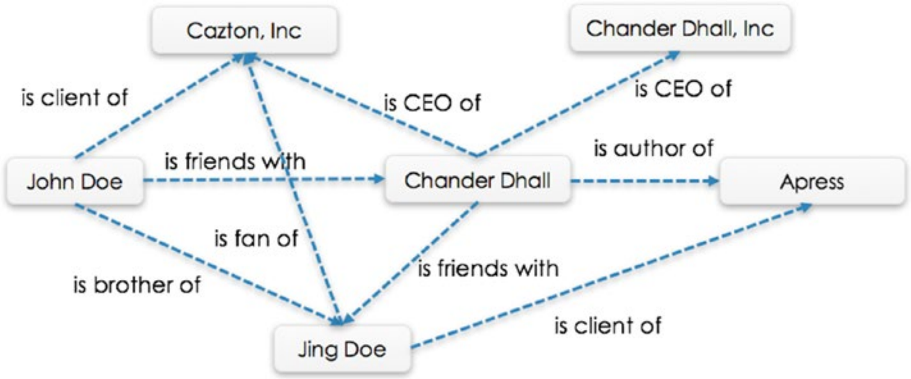


Figure 5-8. *Graph Database showing multiple entities and relationships*

All the nodes of the graph correspond to real-world entities. Edges specify the relationships between these nodes. Each node knows the adjacent nodes. Even with the increase in the number of nodes, the cost of a hop (or taking a step from one node to another) remains the same. The idea of the graph databases is to specify real world relationships as it is and be able to speed up queries including hierarchies and relationships. Queries are sped up because there are indices that speed up lookups.

In Figure 5-8, multiple labels have been depicted with some properties like rel, since, and id, which show the relationship, start date of that relationship, as well as id of that particular relationship. Note, the id of the node is the id of the entity and the id of the edge is the id of the relationship.

If you were to pay attention to the edges in Figures 5-8 as well as 5–9, you will notice that it's very easy to get the edge data as well as the labels. However, in a relational database the only way to get that data would be to have a join between at least two different tables, which is surely a costly operation. In the real world, this can involve many different kinds of tables. As you can see, the relationships in Figure 5-7 are simple relationships but the ones in Figure 5-8 depict multiple attributes. Please note that the relationship between the nodes is depicted by directional edges. However, Figure 5-9 shows an example of two-way relationships. These are just very simple examples. But the value of graph databases in recommending things based on interests, behavior on an application, as well as purchase history is unprecedented. In such cases, there are a lot more relationships between entities like customer to entities like products. Not only that the relationships are bidirectional and have a lot more attributes. Hence, graph databases make a good candidate for social networks and recommendation engines.

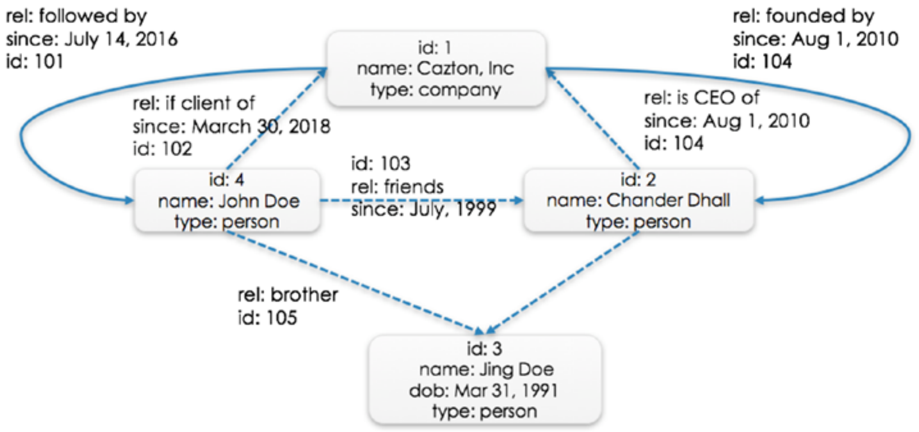


Figure 5-9. *Graph Database showing multiple entities, relationships, and multi-labeled properties*

As compared to a Key-Value store that is optimized for lookups, the graph data store is optimized for traversing data between nodes. Generally speaking, lookup would be slower in case of graph database compared to a Key-Value store. Note, we are able to create a tree kind of structure in a Key-Value pair if we have to. This is usually done by having subsets of Key-Value pairs within the immediate parent key, all the way to the root or main parent key. However, a graph database is not just capable of having a simple tree structure, but it also has actual relationships that are quite intertwined, more like a graph. So, if we were to stick to the tree analogy, it's like having a tree with branches and a trunk, at the same time, roots that connect to other trees in a forest. Some graph databases also end up using Lucene behind the scenes that helps them index their data better and in certain cases they could end up being as fast as a Key-Value store.

Even though both relational databases and graph data stores are capable of handling relationships well, the former are optimized for aggregation, but the latter are optimized for connections. This makes the queries faster for handling all sorts of relationships because they are indexed. However, this doesn't make Graph databases the end-all and be-all database for Master Data Management (or MDM). They are not very useful for processing high volumes of databases and since they are not optimized for aggregation of data, they are not very good at handling queries that span a vast majority of the database. They do have their place in a polyglot persistence scenario where they are one kind of database, but the primary database may not be a graph database. One thing to remember is that a well-designed and optimized RDBMS might be almost as good as a graph data store on retrieving relationships, but it's also good at a lot of other use cases that the graph database might not be.

Full-Text Search Engine Databases

Even though Full-Text Search Engine databases are a specialized form of document databases, the way they are designed and the fact that they have completely different use cases, makes a case for having a separate discussion. As you know that in the document database, it's really the key that's important. If we know the key, we can get the data we need. If we know the value and the subsequent keys within the value, we can also get the contents. In a Search Engine though, it's all about the value. The value gets indexed after the Search Engine processes it, using analyzers (examines text and generates a token stream), tokenizers (break data into lexical units called tokens), and filters (examines tokens and then keeps them, transforms them, or discards them). Once the value is indexed we retrieve the data we need using the indices.

Full-text search engine databases are optimized to handle textual data, and in many different languages. They have a very granular indexing structure and have flexible query operations. Every document has a weight (or score) attached to it by default. This score can be altered by boosting (discussed below) the document higher or lower so it could be displayed accordingly.

A typical indexing process means analyzing the content, creating a document, analyzing it using some kind of analyzer, indexing that particular document, and adding the index where all indices are stored. These are the stages in a search engine:

1. Acquire content: We get data from different sources. The idea of this stage is to make sure we remove redundant pieces of data we don't need. For example, if we got data from a website that is in HTML, we might want to remove the HTML markup.

2. Build documents: A document is a unit of search and has fields with values. For example, if we are searching for, let's say Laptops, on an e-commerce website. We will be able to retrieve N (where N is the number) laptops provided they exist in the search engine. That means we were able to retrieve N documents.

3. Boosting: While retrieving the N documents in No. 2 above, the search engine would rank them using some sort of ranking algorithm. However, during the build process we can also rank a document higher than others by increasing the boost value. This process is called Boosting. For example, if someone is searching for "24-inch monitor" and we have a 27-inch monitor on sale, it makes sense to boost the latter so the user can see it in search results.

Boosting can be done during build as well as during retrieval except it's a little bit more expensive to do it during retrieval process.

4. Analyze document: A text needs to be broken into tokens that make sense. For example, we may not need to keep the stop words like "a," "an," and "the." We can use the built-in Stop Word Analyzer for this. We may also want to make sure we tokenize phrases together: "passed away"' and '"passed out"' are different than storing tokens '"passed," "away," and "out." We can use a custom phrase analyzer for it.

5. Index document: Finally, we can index this document and store it in the search engine's source of record, which could be files, databases, and eventually a copy of this might also live in the RAM just to speed up retrieval.

A good search engine is one that has high recall without sacrificing precision. Precision happens to be the fraction of retrieved instances that are relevant. Recall is the fraction of relevant instances that are retrieved. So, if we were to search for my company name "Cazton, Inc" and we get, let's say 15 results and only 12 out of 15 were relevant to my search then the precision is 12/15. However, what if there are total 25 results in the search engine that pertain to 'Cazton, Inc'? That means the recall is just 15/25.

Another important concept is one of an inverted index. It is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents. If you've used an index at the end of the book, you know what an inverted index is. In a book, we can look for a keyword and it shows a list of all pages that keyword is in. In case of search engines, we can use the same logic and make our applications easily searchable. Let's assume we want to find out which pages on Cazton.com refer to trainings. In that case, we can use the inverted index "training." A more involved example is show below. Here are some of the trainings on Cazton.com

1. Mastering ASP.NET core with Angular and EF Core.
2. Mastering Microservices – (Node.js, ASP.NET Core, or Java).
3. Data Science Made Easy.
4. Scalability patterns training.

Assuming the analyzer gets rid of unwanted characters like '-', '.' and '(', the index for above data would look like this:

ASP.NET core: {(1,1), (2,3)}

Mastering: {(1,0), (2,0)}

To simplify things, we used numbering. This could have very well been the url of the page where the training is listed. Nevertheless, search engines are not only important but are the only kind of databases that make perfect sense for adding search capabilities in the website. They can be used for speeding up searches, creating ranking algorithms, and also some good recommendation engines. With the addition of artificial intelligence in search engines lately, their scope has become broader. Some of the popular search engines are based on Lucene. They are Solr and Elasticsearch. Azure search is a search engine in Azure that provides abstractions that make it easier for developers without a background in information retrieval to be able to add search capabilities to their applications.

Summary

In this chapter we discussed different types of No-Sql databases. We can see that all these No-Sql databases are built for different purposes and have different strengths. The problem occurs when we try to use one kind of database – be it No-Sql or relational – to take care of all our needs. For applications that require a high-scale, it's important to understand that very rarely we can find a use that for which only one kind of database would be sufficient. We need to agree that it's best to find the right database for the right problem and be smart about managing data consistency across the board. Often times in the enterprise, we see one kind of database solving problems it's not meant to solve. We also find primarily relational databases as well as document databases advertised as an end-all and be-all solutions for almost everything. Sometimes, even the so-called experts indulge in this kind of favoritism (or cult following) and even though they may be an expert in that particular database, that doesn't make them a data expert or a scalability and performance expert. The rule of the thumb is that if there is an expert out there, that guarantees that one of these databases will work better than all other databases for all scenarios possible, the only thing we can infer from this is that they are not experts.

That said, CosmosDB is a database that provides five different consistency levels. Microsoft also supports a .NET, Node.js, Java, and a python SDK for Cosmos DB. It runs in Azure and is fully managed. It automatically replicates all the data to any number of regions worldwide. It provides a wide variety of consistency levels.

Strong consistency: This is an RDBMS-like consistency. With every request, the client is always guaranteed to read the latest acknowledge write. However, this is slow, and in order to use this the Cosmos DB account cannot be associated with more than one region.

Bounded staleness: This level guarantees that the reads may lag behind by at most x versions of the document or a certain time interval by the client. For example, if the client sets x=2, the user will be guaranteed to get a document no later than the last two versions. It is the same with time. If the time is set to five seconds, every five seconds the resource will be guaranteed to have been written to all replicas to make sure that subsequent requests can see the latest version.

Session: This is the most popular of all, and as the name suggests, is scoped to a client session. Imagine someone added a comment on a product on an e-commerce website. The user who commented should be able to see it; however, it will take some time before other users on the website can see it too.

Eventual: As the name suggests, the replicas will eventually converge in absence of any additional writes. This happens to be the one with the weakest read consistency but the fastest of all options.

Consistent Prefix: By selecting this option, you ensure that sequence of writes is always reflected during subsequent reads.

The fact that CosmosDB is in Azure and provides guarantees of 15ms writes and 10ms reads, it's worthwhile to use it for most applications of scale. We've had great experience implementing systems of scale at our customers. However, this is for a specific customer base that has either moved fully to Azure or is on its way.

CHAPTER 6

Polyglot Persistence

So far, we have seen that no single database, be it Sql or No-Sql, has been able to satisfy all the business needs. Prior to mass flooding of the world with mobile devices (cell phones and tablets) and even IoT devices, it might have been fine to use a relational database for most applications. However, the advent in social media and the ever-increasing number of users have led to polyglot persistence being the panacea for scalability. In Figure 3-10, we had arrived at an architecture that could scale most of the business applications. We had created a scalable set, and we displayed two sets that were being fed traffic by a system-level redirector.

The architecture successfully got rid of all single points of failures and has the potential to scale to millions of users a day and be able to store an incredible amount of data. However, the architecture still has a constraint that the system of record happens to be an RDBMS. We will now continue our journey toward creating an architecture that is overall better; more scalable; has higher performance; is easy to use and understand; and quite surprisingly, overall much cheaper.

Offline Processing

One of the often-ignored techniques in architecture is ***offline processing***. Imagine going on an e-commerce website, getting your cart ready, and adding the payment information. Let's assume you are a trusted customer. Imagine having issues with the payment service. This could be resolved in

© Chander Dhall 2018
C. Dhall, *Scalability Patterns*, https://doi.org/10.1007/978-1-4842-1073-4_6

real time and may take forever. What if the real-world issue takes 20 minutes? Would you want the customer to keep retrying for 20 minutes? Chances of losing the customer are high. A different workflow instead could be storing the customer's order as well as the payment information (assuming your system is legally compliant to do that) and providing him with a customer order. Once the payment problem is resolved, there could be two options. Depending on the legislation of the place your business runs in, it might allow you to accept the payment or it might mean we might have to send an email to the customer to verify something like a security code of the credit card before the actual order is processed.

This is just one example of taking a workflow offline and making the customer's problem ours. In simple terms, whatever we can do to make sure that the customer has a phenomenal experience on the application or the website, that's the key to success. So, one of the rules of thumb my team has is to gauge if anything could be made offline, without sacrificing any user experience. It's important to consider user experience though. We should never make something offline just for the sake of it. Let's take an example of a unit of work. A unit of work is a very commonly known pattern. It's used to group one or more operations in a single transaction. A good example would be transferring money from a checking to savings account. If some money is withdrawn from the checking account but never makes it to a savings account, that will be a big problem. So, it makes sense to add them to the same unit of work. This way we can we rest assured that the transaction will either succeed or fail as a whole but won't bring inconsistency in data.

Sometimes, architects and developers go overboard with the idea of using unit of work pattern and that might mean using it incorrectly. I like to ask this question in my presentations worldwide and, not surprisingly, 90% of the time I get the wrong answer. The question is simple. If there is a login functionality on a website that has two distinct calls to the API, do they fall under the same unit of work? Step 1 is creating the login via the API. Step 2 is sending out the email. It's human to think of this as a unit of work but it’s impractical. The reason even smart developers get this wrong sometimes is because they think of the scenario where if the email didn’t go out, the user should not be allowed to log in. This is definitely a good way to look at it. However, if this is part of the same unit of work we have a problem. Let’s assume the email is down for 10 minutes. That means the user will try to keep creating the same username over and over again for at least 10 minutes. Chances of him leaving the website are high.

However, if they are both different transactions, it works perfectly fine. The user creates an account on the website and can go do other things. The email service resumes and he gets an email. There needs to be an offline queue that is tracking the requests in case of the email service or an offline process checking the database for new registrations and making sure emails have been sent out. In my personal experience, offline checks are mostly just considered good to have, but in reality they are very essential for systems to work well. In distributed systems, we can always expect things to not work. We should expect service, databases, third-party APIs, and every single component to fail. That’s why we need to have a background process that makes sure the system functions properly and the data is coherent. Depending on the programming language we choose, there are many different asynchronous messaging frameworks that are available in today’s world. Choice of the framework would depend a lot on the actual needs of the project as well as familiarity of the language in which the framework has been written in. Most commonly used ones are Advanced Message Queuing Protocol, Message Queuing Telemetry Transport, Java Messaging Service, and .NET Messaging Service. LinkedIn’s Kafka (open source software) takes message queuing to the next level. It has a reactive pub-sub architecture that allows various different consumers to process the logic based on the same messages (or events) because it publishes these messages to topics and persists them. It scales out very easily and has configuration options for reliable messaging. To know more about Kakfa, feel free to read my article on it. You can find it at `https://www.cazton.com/consulting/kafka`

Regardless of what technology we use, it's important to have a policy of offline-first. This means, if parts of a process could be made to go offline, we need to investigate the possibility and feasibility of it as a team. By reducing the online interaction with the client, we are reducing the risk of customer displeasure in terms of failure. That said, critical functionality with real-time needs should be made (highly) available to the customer and attempts should be made to reduce the response time on it.

Polyglot Persistence

Polyglot persistence means using multiple kinds of database technologies. So far, we have made a case for using more than one kind of database technologies. A good way to understand this would be to take an example of an e-commerce application. Let's assume we have an e-commerce application, Cazton.com. For now, we can simplify the business by saying that it only sells electronic items. A typical user goes to the website and searches an electronic item of choice be it a laptop, desktop, a cell phone, a hard drive, a RAM, or anything similar.

However, as we know, the home page is also a pretty loaded page. With the exception of websites like Google.com, most businesses like to display a variety of products on the home page. There is nothing wrong in this approach. In fact, it's also a way to let the customer know what all products and services the website offers. However, the problem occurs when we wait for an entire page to load before any functionality works. What if the user only wants to search for a 24-inch monitor, buy it if the price is fair, and leave? He will still have to wait for the entire page to load. Depending on the number of images, the actual markup, the stylesheets, and JavaScript files, it could take forever. It can even get worse with peak load times on the website. So how do we fix this problem?

First, we need to make sure that the critical styles (CSS files) and the critical script (JavaScript files) are made in line with HTML. What's the reason behind this? If you recall, we just discussed a typical user wanting to just search for the product he wants to buy and leave, rather than browsing through the home page. If we wait for the entire home page to load, we will be reducing the customer's user-experience. That's why we want to make sure that the critical styles and scripts are downloaded with the HTML file before we download the rest of the scripts and styles. Think of this as having a search button on the home page, ready to go while the rest of the page is loading. Second, since our home page usually gets most of the traffic, it's also a great idea to use a CDN as well as cache the static content.

Third, we need to make sure that we use a search engine database for the search functionality. What about the dynamic text on the home page that comes from a database? We need to make sure that if it is text that does not change that often, it's stored as well as served from a No-Sql database. Why is that? We do not want data in a relational database that does not belong there, simply because relational database is hard to scale out. So, we need to make sure that that the relational database is home to data that is critical to the company. Our goal should be to allocate the data smartly (where it belongs best) so we delay the relational database from maxing out the capacity.

Isn't that a simple fix? Can we not just get a server with more RAM and hard disk space? Yes, we can. However, beyond a certain number there is no real advantage to adding more capacity or memory, and it also gets extremely costly. What do we do with the static text on the home page? If the text is something that does not change very often, we can use a caching database server like Redis. Depending on your app, caching could be done on the browser as well as using the second-level cache.

What happens to the images and thumbnails if this is a mobile device? Images must be optimized per device. A lot of times we can see that it's the same image and a different stylesheet is used per device. This can make pictures look smaller but the mobile device still has to load the same image that was created for the desktop. This must be mitigated across the board. Another important thing that needs to be done is to optimize the stylesheets such as layout and ensure that paint operations are not triggered more often than necessary in the browser. This could be done by using simple techniques as hiding the element, making all the changes in the DOM (not the render tree), and then making the element visible again. This way layout will only be triggered once. If you need to read more about this I would investigate *Virtual Dom* approaches as well as frameworks. Since this is not a book about web development, we will not get into more details on web performance.

Note If you're interested in learning about web performance, feel free to check out some web performance techniques on Cazton.com.

Now that our user has made a search for a 24-inch monitor, the Search engine will quickly display the results. Here are certain things to note:

1. The number of items returned by the search engine should be set by default for device. For example, if you are displaying ten results in the auto-complete box for a desktop, you may only want to show six results on a tablet and maybe just four for a mobile phone user.

2. Why not just use an RDBMS? There are a few major reasons. First, the search engine will scale out easily. Second, a Search engine will be faster to respond. Third, we want to reduce the load on the relational database.

3. What if we have a 27-inch monitor on the sale and it might be priced as much as the 24-inch monitor? We would definitely want to boost it up. This means it will show up higher in the search results based on the

boosting score we provide. This is easier to implement in a search engine than a relational database as boosting is natively supported in a search engine.

Now that the user has made the selection, we assume that he clicks on the 27-inch monitor. This may mean going to a completely new page from the home page. Let's assume the home page is `http://Cazton.com` and the new page is `http://cazton.com/monitors/88` where 88 is the id for the monitor in question. This page will have some sections. They could be product description, comparative products, and comments regarding the product. Of course, it may have images as well. Again, the images could be served from a CDN. What about the data? There could be hundreds of comments. Product descriptions could be really large and may talk about a lot of other things including warranties, shipping information, and return policy, etc. Our job is to identify what changes often and what doesn't. This will help us decide whether we use a No-Sql or a relational database. At the same time, it's important for us to identify which out of these are consistent across the board. For example, the return policy could be easily the same across the entire system simply because as a business it makes sense to do that. Things that are consistent throughout the system need to live in the second-level cache and could be cached on the browser. One thing most often overlooked is the aspect of cache failing. Of course, in that case we should be pointing to the source of truth. In this case, that could be a document database.

We still haven't talked about comments. Comments are not like product description. Product description is a write-once and read-only kind of scenario. Very rarely should a product description change. However, comments could change any time. Even though the actual comment that has been uploaded may not change, the list of comments is an ever-changing list with new comments coming in any time. A product can have zero comments, all the way to hundreds of thousands of comments. This is where a lot of architects would add comments in a relational database. Bad idea. Why so? Simply, because it' verbose data. The logic behind adding the data in a relational database is that it's transactional so we will able to make sure that the last comment added gets retrieved in the next call to the database. In case of a No-Sql database that's not usually the case. It may have session consistency. That means the user who created the comment might be able to retrieve it, however, the change may take time to get through to all redundant (or replicated) servers. This is exactly what's called eventual consistency. That means some users may not see the new comment. This is possible in a high traffic website. Let's ponder this for a moment. How important is comment No. 439 if the user can see 438 comments before it? Hardly important. This is where the business of the company should be looked at before making a technical decision. It's completely fine to make the comments live in a No-Sql (preferably document database) rather than a relational database. That eventual consistency overhead for a very small time affecting a miniscule amount of user experience for a very small number of users is nothing compared to the elephant in the room that is scaling the RDBMS.

To play devil's advocate, what if it's comment No. 2? Assuming comment No. 1 was a 1-star comment placed by a company competitor and comment No. 2 was going to be a 1-star rating of the product by an actual user, this comment could have changed the mind of the buyer. Even though it's highly unlikely, it's possible that we can get into a situation like this. This can be resolved by a No-Sql database still. There are a couple of ways. One is that you can make the No-Sql guarantee transactional consistent. Some databases provide that option. CosmosDB provides that option along with five other consistencies as discussed earlier. This will slow down the writes because it will make sure that all replicas have the new comment. So, there is a tradeoff but it's possible. Another way is to choose eventual consistency (or even session consistency) but then write API and UI logic to either poll the new comment or use websockets on the API and observables on the UI to make sure the value is always updated. This will provide a very similar experience. But the bottom line is that there is no good reason to add the comments to a relational database. The cost of doing that is way too high. Remember our goal. We don't want to delay maxing out the relational database for as long as we can.

So now let's assume the user has read everything about the product, compared it with competitive products and read the reviews on page no. 2. Now, she wants to buy a product of her choice. When she selects the Add to Cart option we can now add the data to a relational database. Why now? Simply because order-related data is very important for any business. This is data we cannot afford to lose. Remember, the business pays taxes based on this data. And everything from adding to the cart all the way to adding customer information as well as payment and shipping information needs to be in the relational database. What about speed in this case? Security of data takes precedence over user experience in this case. Additionally, the relational database will be much faster now simply because of the reduced, thanks to the caching engine, search engine, and the document database taking most of the load so far. Not only do the relational databases provides transactional guarantees, they also provide a wide array of functionality (more than almost all No-Sql database) including reporting, encryption, etc.

Note We could have also used transactional consistency in CosmosDB for the Add to Cart option, used session consistency for the comments, and eventual consistency for the product description and related data. This way Cosmos DB could have been a panacea for all ills. However, since Cosmos DB only works on Azure, we cannot assume that everyone reading the book is going to use Azure. So, we will continue to talk in terms of relational and No-Sql databases separately. Once we understand the concepts, actual implementation is just trivial compared to decision making that's needed ahead of implementation.

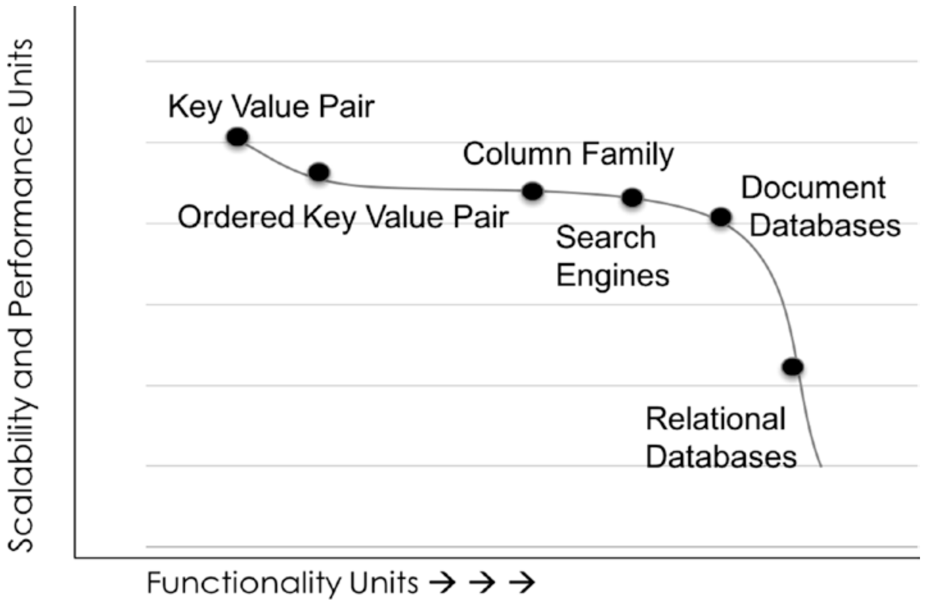


Figure 6-1. *Comparing different databases in terms of scalability and performance vs depth of functionality offered by the datababase*

In Figure 6-1 above, an attempt has been made to compare the different databases on the criteria of scalability and performance as well as on the depth of the functionality offered by them. It's very obvious that a simple Key-Value pair store will be the fastest in terms of retrieval and hence responsiveness will be high. And since we are just scaling a hashtable or a dictionary for that matter, as long as we know which server the value is in, we can get there with just one hop. Ordered Key-Value pairs or for that matter Key-Value stores like Redis may be just a bit slower but are almost as good as the Key-Value stores.

Note We have previously discussed how Redis could be faster than Memcached in certain cases. This graph can change based on the actual implementation and the business use case in question. The graph assumes we are using these technologies correctly and with accurate implementation.

If column families are done right, they could be almost as scalable as a Key-Value pair store for the same amount of data. Since column-family databases are used for limited business cases, they do a great job scaling, better than most databases for the right fit of the business cases.

Search engines are similar in scaling and performance but also limited in terms of business use cases. They usually focus on the value whereas the document databases focus on the key portions. If you follow releases of document databases carefully, you will notice some of them add complex search functionality in a few major releases. Why is that? It's because they can add structured (or key-related) searches easily just because of the way document databases are constructed but they are missing indices (which are created on the values not keys) that could support unstructured or semi-structured searches.

Note In my personal experience, architecting and delivering a search solution for a Fortune 100 client, they were using a search engine directly from their application. This is when the search engine didn't have a default security add-on that distinguished between the role of a user. In short, that meant one rest call to the search engine could have deleted all the indices. Even though those indices could be re-created it could have taken anywhere from 15–30 minutes based on their data, and that would have meant slow user experience as the searches would have to fall back on the system of record, which was a relational database. In such cases, the solution is to go through an API that directly makes calls to the search engine, which is the second tier of servers only accessible to the internal network. Not just that, we should make sure that there is only one port open through the firewall and that the web server checks for the correct rights before making the calls. Lastly, we should also make sure we change the default port. Most search engines that get hacked are the ones running on default ports (with default admin passwords wherever applicable).

Document databases offer the most competition to relational databases in terms of functionality. They are almost as good as the relational databases. However, they do score really high on performance and scalability. It's truly hard to compare a traditional document database to the modern ones any more. Nowadays, modern document databases like Cosmos DB support all sorts of consistencies. In fact, when Cosmos DB was launched, it was actually called DocumentDB. Since then, it was very clear that Cosmos DB literally does everything (in the Cosmos) that a Sql or No-Sql database can do and at the same time is available throughout the world on Microsoft's cloud, Azure, and so it made sense to change the name to Cosmos DB.

Relational databases still have their value and will continue to do so. Because of the performance of Document databases, sometimes architects build systems that may not even have a relational database. That works for some business cases where there is a lot of social data that may not be as important as e-commerce data. If a company is hosting tax-related, payroll-specific, health care or e-commerce data, it's hard to envision a system with complete absence of relational database (with the exception of Cosmos DB). It makes sense to incorporate a strategy of minimally using relational databases. This means we use relational databases only for data that belongs there. This should not be misconstrued to mean that all systems will have a minimum amount of data in the relational database. There are certain systems that may only make sense to be in the relational database. Due to some security policies, data-related restrictions, client contracts, and geographical restrictions, there might be cases where the entire data may have to be served from a certain quality of data centers that may be centrally located and each customer may get its instance of a relational database. This might be costly but could scale the way we described the relational databases scale. However, the discussion assumes there is a valid case for scaling the system and it tries to summarize the different patterns that are possible.

In short, for a system to scale well we need to have the following components. In Figure 6-2, as you can see there is a system-wide hash table that maps the calls to right place. We have load balancers that direct the traffic to the underlying web servers, which are easily horizontally scalable. Offline processing is a big component of it and since it has been explained before, it has been structured in the diagram in the way caching has. Relational databases have been shown to be scaled in the same fashion we explained earlier using SANs and database clusters. However, if we have the search databases and document databases that are auto-scalable, along with caching data stores that are auto-scalable too, we will rarely reach a point that the relational databases will need to scale beyond a certain limited number.

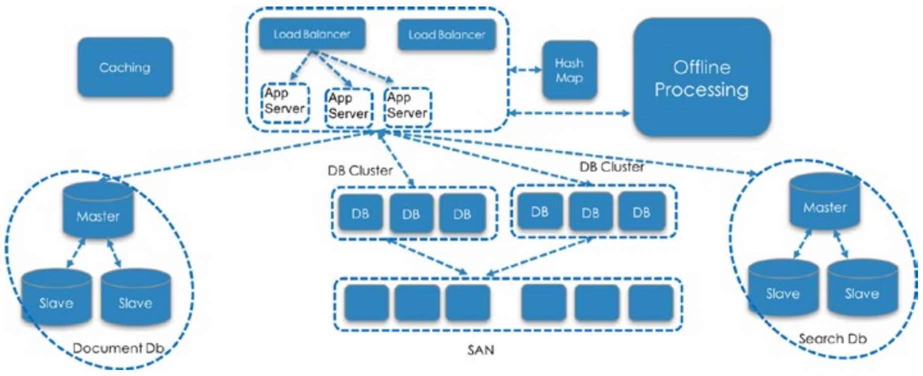


***Figure 6-2.** Dhall Architecture: Finalizing the scalable architecture*

This is a very simplified representation of Dhall Architecture, an architecture that can scale most of the systems in the contemporary world. It's more conceptual than a full-blown architecture diagram of an actual system. For example, security is a concept assumed to be understood and for that reason it hasn't been mentioned here. That does not mean we don't use security, but the focus is only on scalability patterns. Imagine something like Netflix. Can it scale using the following principles? Of course it can. However, the actual diagram would be quite different. It may have more than 700 different microservices. The beauty of the conceptual representation above is that if you remember all the concepts discussed, you can now create a microservices-based representation of the above diagram, tailor it per your needs, and get the benefits of a scalable architecture that has high responsiveness. The diagram does not get into every single detail that has been discussed earlier but assumes you will use the concepts discussed in the book, for example CDNs, wherever necessary.

Note Dhall Architecture is less of an architecture, but more of a thought process. It encompasses all the major concepts and techniques discussed in the book. In order to create a truly scalable system we need to use principles that are based on optimal utilization of resources. First, we need to create units that are independent and use the least amount of resources. Next, all we need to do is scale them.

The book is not about microservices but in certain business cases microservices make a lot of sense. It's a good trend in the right direction. Except certain instances where architects go a little overboard with it (especially on systems that would not need any scale) for the most part it's a trending architecture that pays off both in the short and the long run. In fact, because of microservices-based architecture, different technology stacks can now be combined to create a scalable architecture. Many of my clients have moved to a microservices model after my recommendation just to be able to take advantage of the latest tech that is out there while the monolith app is slowly broken down. In certain cases, the applications have combinations of Classic ASP pages routing to ASP.NET Web forms that may route to Java websites. Rather than rewriting the legacy applications overnight, it makes sense to move them to a microservices model (if it makes sense for the business) and then slowly rewrite certain components. In case of Single page applications on the Web too, we are witnessing the rise of web components that have the same principle of interoperability (as that of microservices on the API layer) on the UI. That means if a web component is written in a SPA framework, for example, Aurelia, Ember, or Vue, it should be able to work in an Angular or a React application. The concept of interoperability and being able to scale multiple technologies together is what makes microservices very appealing to almost all modern systems.

For microservices to be successful, we need to make sure we have a highly successful DevOps practice. We need to make sure we automate the build and unit tests from the version control itself. Continuous integration and delivery needs to be a must. We need to also automate acceptance tests. After we run iterations of all these tests, only then should we submit the latest changes for user acceptance testing before the release. User telemetry as well as usage monitoring of the system are a must. We need to have code metrics and peer code reviews along with architectural validation. If possible, we need to have a developer sandbox in a cloud-based infrastructure with a cloud-based developer and test labs to test for failovers as well as scale of the system. Auto-scaling configuration must be a part of such a system. Deliberate attempts should be made to fail the individual components of a system in dev, test as well as staging environment, and none of these attempts should bring the system to a halt. A system is only supposed to be unavailable if and only if all services of the system go down at the same time. For that matter, geo replication of the entire system is also an option and should be tested via automation.

Microservices architecture requires a lot of discipline and dedication in teams. They should be self-managing teams that own as well as are accountable for lapses and failures and ready to team up to take the right steps in the short as well as the long run. Infrastructure as code or a way to provision the entire IT infrastructure via code is very essential to create an auto-scale environment. All major cloud offerings have automated infrastructure via code and it makes perfect sense to use infrastructure configuration that manages the environment automatically. For microservices to work, there needs to be a strong core that takes care of major and most important functionalities for the company. It needs to be part of the company's DNA to have cross-team collaboration especially in terms of DevOps and automation. Sharing ideas across the board becomes extremely important so efforts are not being duplicated. There needs to be a team of stakeholder architects from every team that communicates in a broader form across the teams and work toward mitigating any hurdles in the way. Teams have to be truly agile. For someone who's been working Agile and Lean for more than a decade, I feel sorry to say that even now, many places just either use Agile as a religion and don't get the best benefit out of it or they just pay lip service to Agile. Over and over again, my team has been part of teams where we had to create Agile metrics before we doubled, tripled, and sometimes even quadrupled the velocity in the teams we worked with. It's not just about the velocity, it's about the value too. These principles work but it's important to see them as a whole. When we start picking à la carte principles based on our convenience, there is a grave threat to the sustainability of what we are delivering. That's why at Cazton, Inc., we do regular training every fortnight for at least couple of hours along with software development by embedding our team members with the client teams as well as recruiting the top talent for the client.

Microservices models are different for every client and based on a lot of different factors. These are your legacy stacks, legacy and current software architecture, amount of current as well as anticipated scale, business direction and needs, nature of the business, and so on. There is no one size fits all and there never will be. There have been times when we have helped customers move from monolithic to microservices architecture and are able to successful achieve write times of less than 15 ms and read times of less than 10 ms using Cosmos DB on planet scale for critical services. In Figure 6-3, an attempt has been to made to depict a typical microservices-based architecture from a company that has kiosks, call centers, a website as well as stores that bring in customers to buy goods. For now, we call the company Cazton.com.

Note Cazton, Inc is my company and this is a fictitious representation. However, the concepts are fully valid and derived from actual architectures that are successful in production.

In the Figure 6-3 we have certain core functionality. In this case the company has certain physical stores, kiosks, and a huge web presence at Cazton.com. We also assume that the company has customer care centers that also lead to sales as well as exchanges. Assuming every company has a marketing and finance department, all the needed functionality for those two departments is present in the core. The core of the company could be on-premise or on cloud but is still on some kind of infrastructure as code mechanism that guarantees auto-scale as well failover protection. The circle around the company's core is of cloud-enabled services that interface between the company's core as well as third-party companies. For examples, Shipping data service interacts with all the third-party shipping companies to make sure shipment happens on time as promised. In the diagram you can only see one shipping company. This is by design to reduce redundancy. Similarly, if the pricing changes it needs to change in all third-party outlets. These could be websites that display deals and subscribe to the pricing service.

Note Here's a link to a case study of a successful microservices implementation involving Cazton, Inc on Microsoft.com. `https://customers.microsoft.com/en-us/story/elkjop-retailers-azure`. For more curious readers this link has more details: `https://www.cazton.com/blogs/executive/microservices-success-story`

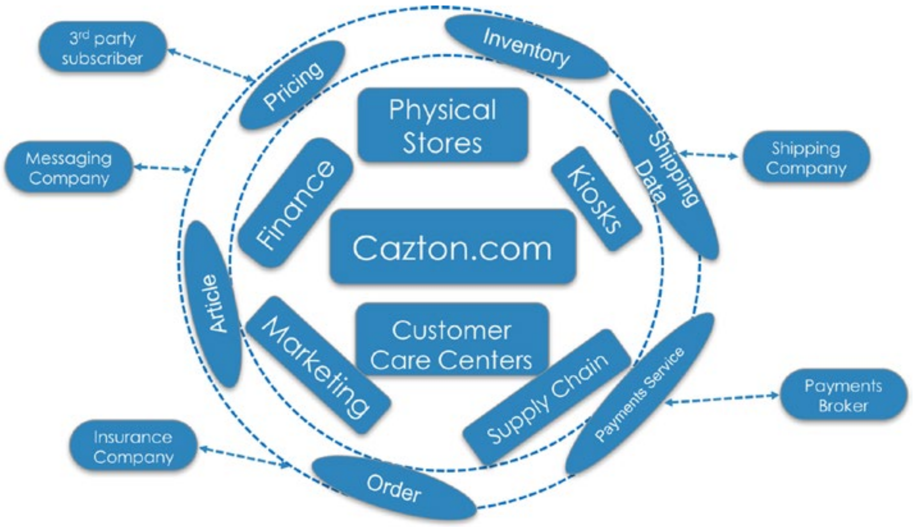


Figure 6-3. *A microservices architecture with a strong core on-premise or on cloud, surrounded by core auto-scalable services that are cloud enabled, and third-party services*

You might be wondering that this is a completely different depiction than the one in Figure 6-2. Even though they might look very different, they are actually very complementary and work together very well. In Figure 6-2 we have the app servers. They could be used either for the application server or as the web server. They could also be used for an API server that hosts a Web API that is a REST-based HTTP service. Cazton.com, the physical stores, kiosks, as well as the customer care centers may be very well talking to the same web API that is on auto-scale. Those APIs could be redirected to a search database, caching engine, document database, or a relational backend as per the need. We can also create the cloud-enabled APIs and scale them in the same fashion as Figure 6-2 architecture. Finally, it’s all about interactions and making sure we don’t end up creating circular references and duplicating the work. Any service that gets hammered more than others could be scaled out separately and easily with the microservices model. We can always scale back the service when the need is over.

One of the best ways to auto-scale is by configuring a certain number of CPU and RAM usage on the server. For example, when 60% of the system resources are being utilized due to an increase in load, it might be time to spawn off another server. Any time the overall systems resources are being used less than 30% it might be time to now scale back the additional server that was just spawned off. This technique assumes that we already have a highly redundant scaled out topology of servers that is auto-scalable. This is just a simplified version of auto-scale and does not in any way represent the entire strategy.

Summary

Even though we have reached the end of the book, we have just began understanding how to scale systems to perfection. Scaling a system is surely a science but science could be misleading. If someone creates code that can handle one hundred thousand users per server, in theory, he would need a million servers to handle a billion users. That may not be true at all. If the developer writes code that doesn’t scale out of one server, it may never be able to serve users correctly. At the same time, if someone can optimize the code, use different kinds of databases discussed in the book, be mindful of which parts to scale, and use the microservices model effectively, he may be able to scale to a billion users with even one hundred servers. Scaling is an art as well as a science. Usage patterns along with artificial intelligence are the keys to creating an auto-scalable system. However, before getting there, there is a lot of meticulous and disciplined work that needs to be done at all levels.

One of the big reason behind my success as an architect has been the continued hands-on approach I have lived with. Architects who lose the hands-on touch may be prone to making error-prone architecture decisions. No matter how much time I spent learning on a daily basis both at work and outside work, having the right team members as well as growing the team have also been big reasons behind my success. Scaling systems is almost impossible if the entire team is not educated timely about the pros and cons of the decisions. Building a truly scalable system comes at a huge learning curve and failures could prove very costly. Training the resources in the fundamental of scaling is just a beginning. There is a lot more guidance, supervision, as well as constant mentoring that is needed to make a team truly successful in a creating highly responsive scalable systems. The patterns discussed in this book are some of the most important patterns in the contemporary world. There are wide arrays of other architectures that haven't been discussed intentionally as they are the right fit for certain use cases. However, I hope, by now, you understand that more than the patterns, it's the principles used to arrive at those patterns that are important.

We all have unique problem solvers in us. The great thing about technology is that we amaze ourselves with newer ways of solving the same problems. Even though education, experience, and intellect help us solve a lot of problems, it's really the drive and passion that differentiates our solution from an average one. Individual problems are easy to solve but systems of scale require tremendous thought process and timely action. With this book an attempt has been made to consolidate major solutions that already exist in the industry so that it can help executives and architects make sound decisions for their projects. At the same time, for beginners and intermediate level developers, this could be the start to know what to do and even more importantly, what not to do. Hope this book helps make you and your team successful.

Index