GUIDELINES
ON
BUILDING
SCALABLE WEB APPLICATIONS

Government of India
National Informatics Centre
Ministry of Electronics and Information Technology
A-Block, CGO Complex, Lodhi Road, New Delhi - 110003

Guidelines
on
Building
Scalable Web Applications

(NIC-SQG-SA-1.0)

Version 1.0

October 2018

Abstract

Modern Web Application requirements are driven by the need for a highly scalable and performance centric platform. The non-functional requirements like performance, availability and scalability are as important as functional requirements of the application. This document captures guidelines and best practices for building web scale applications. The document describe the challenges of building scalable applications and recommend architectural designs and techniques that can be employed at both the application and infrastructure levels to meet these challenges.

The document will help e-Governance application architects in

- Understanding challenges of building web scale applications
- Apply best practices of building scalable architectures
- Understand unique characteristics and scalability considerations for various tiers of multi-tier architecture

Key words: Guidelines, Scalability, Performance, Application Architecture, Deployment Architecture, Content Delivery Network, Web Tier, Database tier, Caching, Performance Testing, Sharding, data partitioning, vertical scalability, horizontal scalability

Government of India
Ministry of Electronics & Information Technology
National Informatics Centre
New Delhi

October, 2018

Contributors

Prepared By	Software Quality Group
Reviewed By	
Approved By	

Amendment Log

Version	Date	Brief Description	Section Change
1.0	12 Oct 2018		

CONTENTS

Abstract ii
Table of Figures vii
1. Introduction 1
1.1 Purpose of the Document 2
1.2 Definitions, Acronyms and Abbreviations 2
1.3 Overview 2
2. Key Concepts 4
2.1 Background 4
2.2 Key Concepts 4
2.2.1 Performance 4
2.2.2 Scalability 4
2.2.3 Load and Load Balancing 7
2.2.4 Caching 9
2.2.5 Content Delivery Network 9
2.2.6 Functional Partitioning 10
2.2.7 Replication 10
2.2.8 Data Partitioning 11
2.2.9 High Availability 12
3. Scalability in Web Applications 13
3.1 Introduction 13
3.2 Scalability Layers 14
3.3 Dimensions of Scalability 15
3.3.1 Load Scalability 15
3.3.2 Functional Scalability 15
3.4 Issues and Challenges in Scalability 15
3.4.1 Scalability challenges at the software level 16
3.4.2 Scalability challenges at the hardware level 18
4. Approach for Scalable Architecture 19
4.1 Introduction 19
4.2 AKF Scale Cube 19
4.2.1 The AKF Scale Cube for Applications 23
4.2.2 The AKF Scale Cube for Databases 27
5. Scaling strategy for Front-end Layer 32
5.1 Managing State 33

5.1.1 HTTP Sessions 34
5.2 Components of the Scalable Front End 37
5.2.1 Load Balancers 37
5.2.2 Web Servers 40
5.2.3 Caching 41
6. Scaling strategy for Database Server 42
6.1 Database Scaling Strategies 42
6.1.1 Architectural approaches for Horizontal Database Scaling 43
6.1.2 Replication 45
6.1.3 Data Partitioning 48
7. Key Principles of Software Architecture 51
7.1 Key Design Principles 51
7.1.1 Keep design Light weight 52
7.1.2 Use appropriate Enterprise Integration methodologies 52
7.1.3 Strive for Statelessness 52
7.1.4 On-Demand Data Loading 53
7.1.5 Resource Pooling 54
7.1.6 Scalability by Design 54
7.1.7 Latency and Throughput optimization 55
7.1.8 Minimize Coordination 55
7.1.9 Enforce high cohesion and loose coupling 56
7.1.10 Avoid Blocked Waits 57
8. Caching for Web Applications 58
8.1 Main scenarios for the use of caching 58
8.2 Impact on scalability, availability, and performance 60
8.3 Cache concepts 61
8.4 Cache design 62
8.5 Caching strategy 62
8.5.1 Caching at the end user 63
8.5.2 Presentation layer 63
8.5.3 Application layer 63
8.5.4 Database layer 63
8.6 Cache metrics and administration 64
8.7 Cache Products 64
9. Resource requirement Analysis for Capacity Planning 66
9.1 Main steps in Capacity Planning 66
9.2 Capacity Planning Parameters 67

9.2.1 Throughput and TPS 68
9.2.2 Work done per transaction 68
9.2.3 Think time 68
9.2.4 Active users and concurrency 68
9.2.5 Message size 69
9.2.6 Latency 69
9.3 Number of Hits 69
9.4 Determining the processor size 69
9.5 Sizing the memory 69
9.6 Performance Metrics 70
9.7 Application Level Measurement 71
9.8 Storage Capacity 72
9.9 Storage I/O Patterns 72
9.10 Measuring Load on Web Servers 73
9.11 Database Capacity 73
9.12 Caching 73
9.13 Application Design and Optimization 74
10. Reference Architecture for Scalable Web Applications 75
10.1 Reference Architecture 75
11. Scalability Testing 81
11.1 Scalability testing parameters 81
11.2 Sub-categories of Scalability testing 81
11.2.1 Performance Testing 81
11.2.2 Stress Testing 82
11.3 Open Source Scalability Testing Tools 83
12. Key Takeaways for Building Scalable Web Applications 85
References 87

Table of Figures

Figure 1: Vertical scaling 5
Figure 2: Horizontal Scaling 6
Figure 3: Load Balancing 7
Figure 4 : Content Delivery Network 9
Figure 5 : Functional Partitioning 10
Figure 6 : Replication 11
Figure 7 : Data Partitioning 11
Figure 8: Scalability Layers 14
Figure 9 : Issues and challenges in scalability 16
Figure 10 : AKF Cube 20
Figure 11 : AKF Cube - X-Axis Example 21
Figure 12 : AKF Cube - Y-Axis Example 22
Figure 13 : AKF Scale Cube for Application: X-Axis Example 24
Figure 14 : AKF Scale Cube for Application: Y Axis Example 25
Figure 15: AKF Scale Cube for Application: Z Axis Example 26
Figure 16: AKF Scale Cube for Database 27
Figure 17: AKF Scale Cube for Database: X-Axis Example 28
Figure 18: AKF Scale Cube for Database : Y-Axis Example 29
Figure 19: AKF Scale Cube for Database: Z-Axis Example 30
Figure 20: Stateful Server 33
Figure 21: Stateless server 33
Figure 22 : Clustered Session Management 35
Figure 23 : Session Data in External Store 35
Figure 24 : Sticky session using Load Balancer 36
Figure 25: Detailed front-end architecture 37
Figure 26: Load Balancer 38
Figure 27: Shared Disk Storage 44
Figure 28: Shared Nothing Architecture 44
Figure 29 : Data Replication 46
Figure 30: Vertical Partitioning 48
Figure 31: Horizontal Partitioning 49
Figure 32: Key Design Principles 51
Figure 33: Request processing with Cache 58
Figure 34: Database Object Cache 61
Figure 35: Caching Layer 62
Figure 36 : Single server deployment 75
Figure 37 : Web and Database Server Deployment 76
Figure 38: Load balancing and Application server scaling 76
Figure 39: Master/Slave Database replication 77
Figure 40: Database Sharding 78
Figure 41: Caching with Object cache 79
Figure 42: Multiple instances 80

1. Introduction

Government of India has undertaken multiple initiatives using Information and Communication Technology, as a strategic tool to provide improved services to citizens under Digital India Programme. The vision of the Government is to transform India into a digitally empowered society and knowledge economy with a mantra of **"Minimum Government, Maximum Governance".** The objectives of Digital India can only be achieved by automating the government processes and providing the extending services online.

There was a time when government applications mostly had limited scope and were implemented mainly for automation of backend office processes. The applications were mostly intra-department and accessible to limited number of users within ministry/department. The advent of digital technology has fundamentally changed this pattern. Today, the citizen want services that are available online anytime, anywhere and from any type of device. To satisfy these new expectations, the government must transform the way it provides services. The applications now have to be available 24x7x365 and should support large number of users including citizens. As the services are now paperless and core processes depend on them, government processes should also meet desired performance levels. It is also much more difficult now, to estimate the expected number of users.

In view of this, it is very important that architecture of e-governance portals/ applications are reliable, extensible and scalable to adapt to the changing environments and load.

National Informatics Centre is in the forefront of implementing large scale e-governance solutions and providing technical consultancy to various departments/ministries at both State and Central government level. It is important to note that the applications designed and developed by NIC are both resilient and scalable. The applications should be able to scale seamlessly as demand increases and decreases, and be resilient enough to withstand the loss of one or more compute resources.

Scalability cannot be an afterthought. It is very important that proper consideration is given to the scalability from the design stage itself. Proper design and planning to implement component-wise scalability can help develop horizontally scalable web applications.

In this document, we will describe the challenges of building scalable web applications and recommend architectural designs and techniques that can be employed at both the application and infrastructure levels to meet these challenges. We will also focus on the individual tiers of a multi-tiered architecture and pay particular attention to the unique characteristics and scalability considerations for each tier of the architecture.

1.1 Purpose of the Document

All components of the system need to be scalable and hence, some strategies, which can be considered to make Web and database layer scalable are discussed. For read-intensive applications, use of caching can provide large performance gains because it reduces application processing time and database access, sometimes dramatically. Caching becomes an important strategy to build web scale systems. This document provides details about using caching for building web scale applications.

The document will help e-Governance application architects in:

- Understanding the challenges of building web scale applications
- Apply best practices and design patterns for building scalable architectures
- Understand unique characteristics and scalability considerations for various tiers of multi-tier architecture

1.2 Definitions, Acronyms and Abbreviations

.

Item	Description
API	Application Processing Interface
CDN	Content Delivery Network
CMS	Content Management System
CSS	Cascading Style Sheets
DNS	Domain Name System
ESB	Enterprise Service Bus
RAC	Real Application Cluster
SOA	Service Oriented Architecture
TPS	Transactions Per Second
VIP	Virtual IP
VM	Virtual Machine

1.3 Overview

To start with, the document introduces the important key concepts in the context of performance, scalability, Load Balancing, Caching, Partitioning, CDN etc. in the context of building highly available and scalable applications.

There are various stages and layers like Internet, Web Server, Application Server, Database server and application itself are involved in the whole process. Bottleneck at any level can have impact on the performance of the application. To achieve scalability, it is important that all of these layers are scalable. The impact of various layers on scalability, issues and challenges of scalability are discussed in detail to enable the readers in understanding the approaches to be taken for overcoming those issues and challenges.

An approach called ***"Scalable Cube Model"*** also called "AKF Scale Cube Model" which can be applied while architecting highly scalable applications is discussed in detail. The Scale Cube model can be applied broadly to any software solution for building scalable architectures. How the approach can be applied at both application servers and database servers is discussed.

The scalability strategies which can be employed both at web layer and database layers are deliberated upon. At the database level architectural approaches - shared disk approach and shared nothing approach, data replication and data partitioning techniques are provided.

The key fundamental design principles to be followed from software architecture point of view are provided.

Caching is one of the important technique, which can be used at various layers from the browser to the application backend database so as to achieve scalability and improved performance in our application. The document provides details on using caching on various layers to achieve scalability.

Capacity planning is of deployment infrastructure is very import to avoid performance issues. The steps to plan and estimate resource requirement have been discussed. The load testing is an important step in capacity planning and accordingly a section is included on Scalability testing of Web application.

An attempt is being made to provide a reference architecture that can scale horizontally at Application and Database level. The architecture can be used as a starting point which can be tweaked as per project requirement.

2. Key Concepts

2.1 Background

As mentioned above, the scalability of the application has to be considered from the design stage itself. Before we dive into the design and architectural aspects, it is important to have clear understanding of core concepts like performance, scalability, high availability, content delivery networks, load balancing etc.

This section discusses the various key concepts for design of scalable web applications.

2.2 Key Concepts

The following are the details on key concepts.

2.2.1 Performance

Performance refers to the capability of a system to provide a certain response time while serving a defined number of users or process a certain amount of data. Performance is about speed by which the process is being executed. Therefore, performance is a software quality metric and is measured in numbers.

Two of the most important metrics of performance are “Response Time” and “Throughput”. Response Time is how long it takes to process a request. Throughput is number of requests that the application can process within defined time interval. In web application context, it is number of requests per second handled by the application.

If our application performance require changes i.e. the application is required to serve more users or process more data, the performance i.e. response time/throughput will be affected by resource constraints. Performance and scalability are not same.

2.2.2 Scalability

In simplest term, Scalability is the ability of the system to overcome performance limits by adding more resources. The performance increase is in proportion to the resources added. Here, increasing in performance means serving more number of users or processing more amount of data. A system is scalable, if the performance does not deteriorate if load increases and performance can be maintained by adding additional resources.

A scalable system has three characteristics:

- The system can accommodate increased usage,
- The system can accommodate an increased data set.
- The system is maintainable and works with reasonable performance.

Scalability is not just about speed. While Performance measures how fast and efficiently a system can complete certain computing tasks, scalability measures the trend of performance on increase of load.

If the application performance deteriorates with increase load (i.e. number of users or volume of transactions) before reaching the desired load level, then it is not a scalable application. Scalability is one of the most valuable quality attributes of a system.

Scalability can be achieved in two ways:

1. Vertical Scalability
2. Horizontal Scalability

Vertical Scalability

Vertical scaling or Scaling up refers to resource maximization of a system to expand its ability to handle increasing load. In hardware terms, this includes adding more processing power and memory to the physical/virtual machine running the application. In software terms, scaling up may include optimizing queries and application code. The Optimization of hardware resources like parallelizing or optimizing number of running processes also help in scaling up.

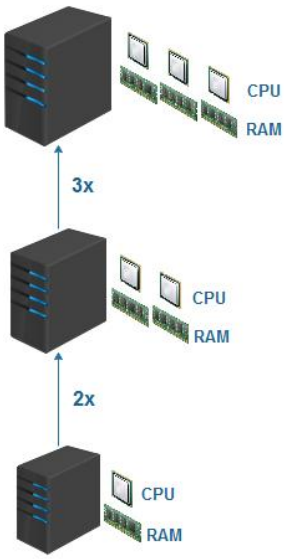

Figure 1: Vertical scaling

As depicted in the figure, the application is running on a single machine. To scale application, the processing power and memory is increased to handle more load.

The Vertical scaling looks straightforward but it suffers from many disadvantages. Firstly, the addition of hardware resources results in diminishing returns instead of super-linear scale. The cost for expansion also increases exponentially. Further, there is also downtime requirement of application for scaling up. If all of the web application services and data reside on a single unit, vertical scale on this unit does not guarantee the application's availability.

Horizontal Scalability

Horizontal Scaling or Scaling out refers to resource increment by the addition of server/node to the system. This means adding more servers/nodes of smaller capacity instead of adding a single server/node of larger capacity. The requests for resources are then spread across multiple servers thus reducing the excess load on a single machine. Horizontal scaling means enhancing the performance of server /node by adding more instances to our pool of servers so that load can be spread.

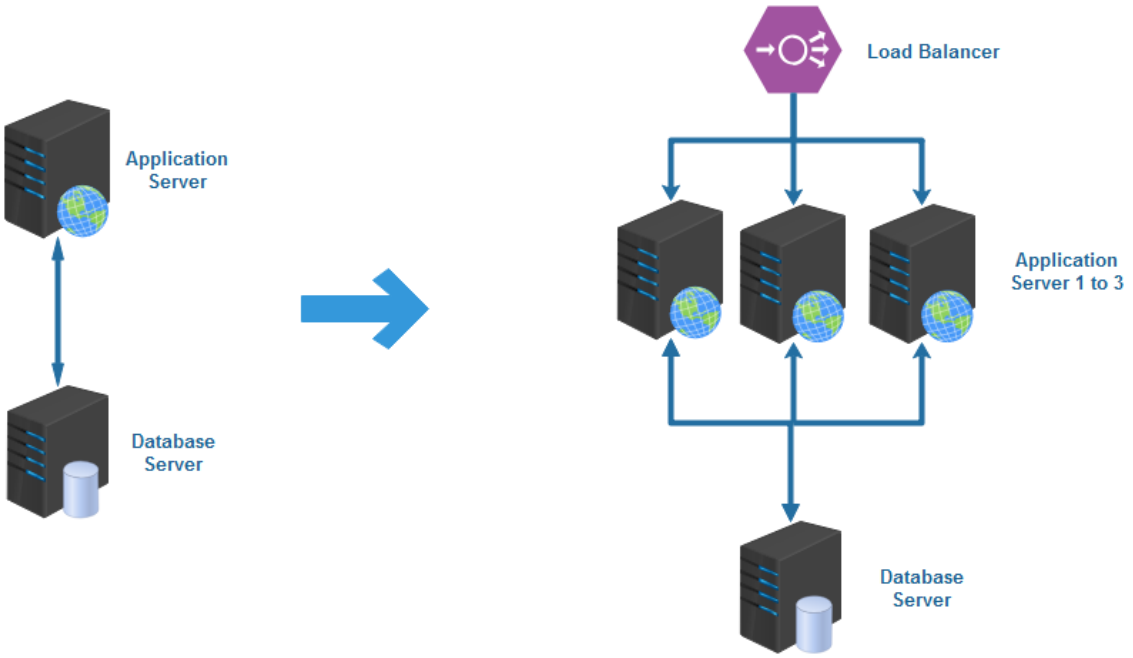

Figure 2: Horizontal Scaling

The above figure depicts horizontal scaling. The application load is now distributed on multiple application servers by using load balancers.

Having multiple servers/nodes allows the possibility of keeping the system up even if some of the server/nodes go down, thus, avoiding the "single point of failure" problem and increasing the availability of the system.

However, there are some disadvantages of horizontal scaling as well. Increasing the number of servers/nodes means that more resources need to be deployed in their maintenance. The code of application also needs changes to allow parallelism and distribution of work among multiple servers/nodes.

In the next section, we look at the technique of load balancing that is often used to support horizontally scaled web application architecture.

2.2.3 Load and Load Balancing

As mentioned above, horizontal scalability means to distribute load on multiple servers/nodes. We need some management system to distribute user request /loads to these servers efficiently. We have multiple requests coming in to the same IP, which now have to be serviced by these multiple servers/nodes. The problem is to decide which server would respond to which request. The solution come from a technique called load balancing.

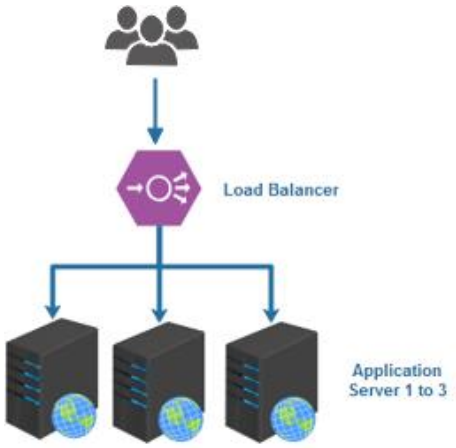

Figure 3: Load Balancing

A load balancer is a specialized hardware or software component that distributes requests/traffic coming to a single IP address over multiple servers that hide behind the load balancer. Load balancers are used to share the load evenly among multiple servers and allow dynamic addition and removal of machines.

Load sharing is effective distribution of load among available servers. Load balancers at various layers normally do this. In a large enterprise level application, all layers such as the web layer, application layer, database layer, and service layer may have load balancers to distribute the load optimally among the servers.

Load balancers employ various policies and work distribution algorithms to optimally distribute the load. Some of the strategies are:

- ***Round Robin:*** In this each server receive requests in sequential order. This is the simplest strategy, similar in spirit to First In First Out.
- ***Least number of connections***: the server with the lowest number of connections will be directed the request.
- ***Fastest response time:*** the server that has the fastest response time (either recently or frequently) will be directed the request.
- ***Weighted:*** this strategy can be highly customized since the weightage can be configured. This strategy takes into account the scenario where servers in the cluster may not have the same capacity (processing power, storage, etc.). Thus, the more powerful servers will receive more requests than the weaker ones under weighted strategy.

Load distribution is vital for large applications to scale and respond during heavy loads. Requests need to be distributed optimally among web servers, application servers, database servers etc. This not only ensures optimal resource utilization but also eases the load on each of the available servers, improving response time.

Load-balanced servers are most scalable and efficient if they are stateless i.e. they do not have to track and store information between each client request. If they must track state (stateful), then it is required to apply session management techniques.

Hardware Load Balancing

The most straightforward way to balance requests between multiple servers is to use a hardware appliance. The network traffic is directed to a shared IP called virtual IP (VIP), or listening IP. This VIP has an IP address that is attached to the load balancer. Once the load balancer receives a request on this VIP, it will need to make a decision on where to send it, based on its load balancing strategy.

Some of the advantages of using the hardware load balancer are:

- Adding and removing real servers from the VIP happens instantly.
- Load balancing can be done as desired e.g. traffic can be directed to a web server that has extra capacity (more RAM or a bigger processor) to make use of all the resources, instead of allowing extra capacity to go unused.

Software Load Balancing

Software load balancing is a cheap alternative to hardware load balancers. HAProxy and NGINX are two popular open source load balancing software.

Software load balancers are broadly classified into two categories which are based on the network layer information they use for load balancing.

1. ***Layer 4 load balancers*** make use of the information provided by TCP at the network layer. The load balancer captures the request at this layer and use source IP, destination IP and port available in TCP stream for routing the request. The load balancer usually selects a server without looking at the content of the request.
2. ***Layer 7 load balancers***, on the other hand, inspect the message right up to the application layer by examining the HTTP request itself. They look at the request along with header information and use that for load balancing. The requests thus can be balanced based on information in the query string, cookies or any header we choose as well as the regular layer information including source and destination addresses.

2.2.4 Caching

In simplest terms, Caching is about storing the result of common operations/queries temporarily to handle these operations faster.

Caching is a technique that enables applications to store data (like web pages, data from database etc.) that is likely to be reused in memory, so that the data may be served faster during subsequent requests. By adding caches to our servers, we can avoid reading the webpage or data directly from the server, thus reducing both the response time and the load on our server. This helps in making our application more scalable.

Caching can be applied at many layers such as the database layer, web server layer, and network layer. We will further discuss caching and caching strategies at various levels in a separate chapter.

2.2.5 Content Delivery Network

A Content Delivery Network (CDN) is a collection of third party servers deployed in different geographical locations. Akamai is one such service. The CDN servers keep cached copies of content (such as images, web pages, etc.) and serve them from nearest location. The use of CDN improves page load time for user as the data is retrieved at a location closest to it. This also helps in increasing the availability of content, since it is stored at multiple locations.

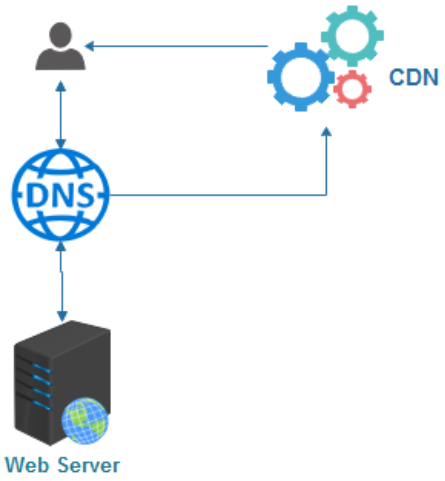

Figure 4 : Content Delivery Network

The CDN servers make requests to our Web server to validate the content being cached, and update them if required. The content being cached is usually static such as HTML pages, images, JavaScript files, CSS files, etc.

2.2.6 Functional Partitioning

The main idea behind the functional partitioning technique is to look for parts of the application focused on a specific functionality and create independent subsystems out of them.

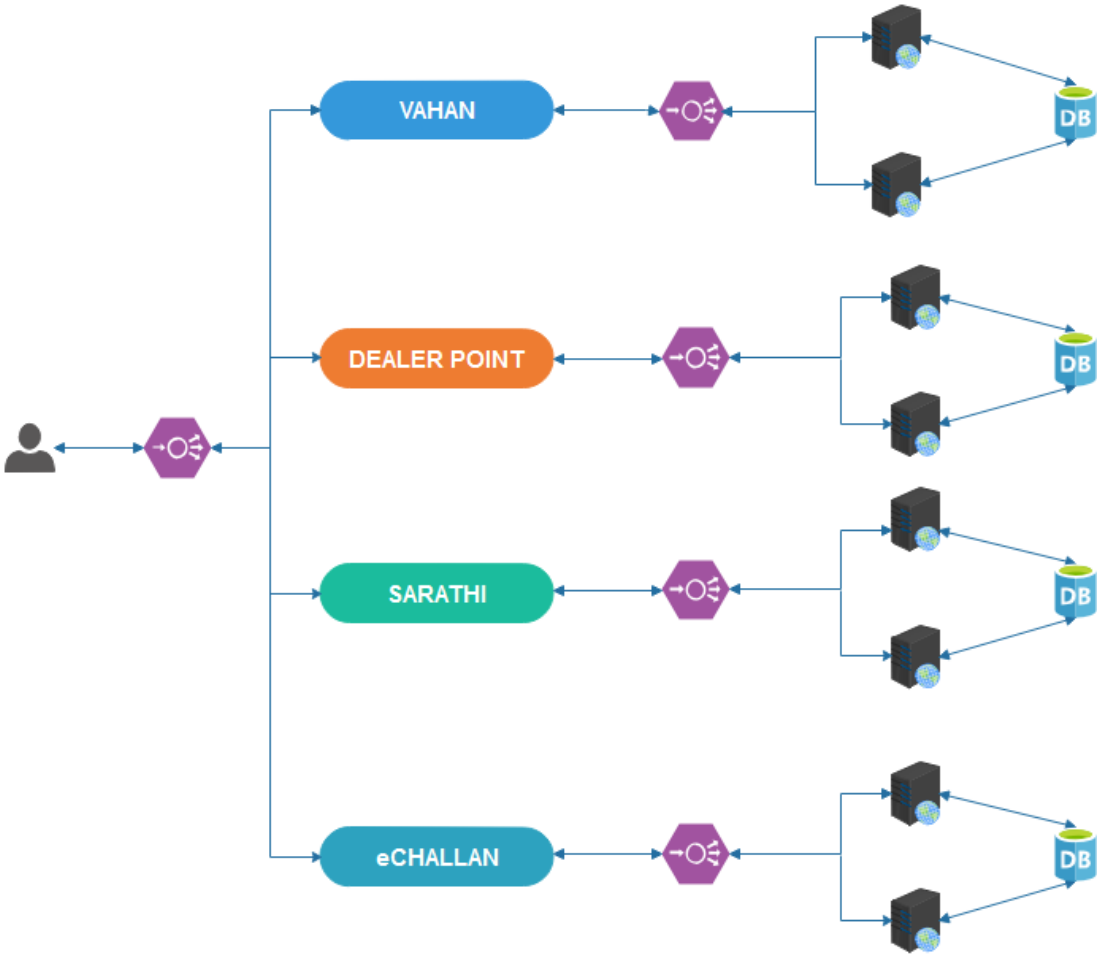

Figure 5 : Functional Partitioning

In the context of infrastructure, functional partitioning is the isolation of different server roles. We have Web Servers, Database Servers, cache servers, API servers, message queue servers, etc. Each of these components can be deployed separately and can scale separately. Similarly, the applications can also be partitioned depending on the functionalities like Submission of applications by citizens, Approval workflow and integration services. In a more advanced form, functional partitioning is dividing a system into self-sufficient applications.

2.2.7 Replication

Database is the most common bottleneck in web applications, since a lot of reads and writes occur at the database level. It is very important to consider strategies for Database scalability. Replication is one of the most important strategy to achieve horizontal scalability

in database. In database replication, copies (replicas) of data are stored on multiple machines. Clients may read from any of these replicas. Whenever a write occurs, the updated data is replicated to all machines to ensure consistency. The database server that does this replication is called the 'Master', while those at the receiving end of the replication are called 'Slaves'.

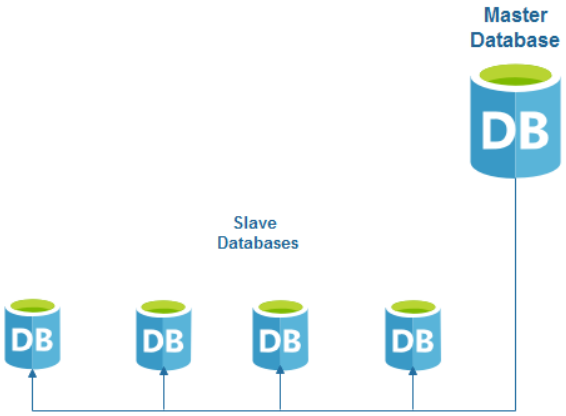

Figure 6 : Replication

2.2.8 Data Partitioning

The Data partitioning is another strategy for scaling databases. The data partitioning involves partitioning the data in a logical way and distribute subsets of it on different machines.

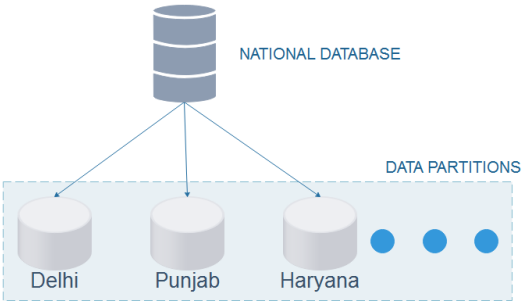

Figure 7 : Data Partitioning

The entire data is not cloned on each machine as done in the case of replication. Instead of cloning the entire data set onto each machine, each of the server has its own subset of data, which it can control independently and is called '*Share Nothing*'. '*Share Nothing*' is an architectural principle where each node is fully autonomous.

As each node can make its own decisions about its state without the requirement to propagate state changes to other servers as done in replication, there is no need for data synchronization, locking, and that failures can be isolated because nodes do not depend on one another. For example, we can partition our data based on Regions or States. We can have separate database server for each region or state.

2.2.9 High Availability

High availability is the ability of a system to be continuously available to users without any loss of service. High availability is critical for operations and business continuity as well as for achieving 24x7 service. High availability is a quality of a system or component that assures a high level of operational performance for a given period of time, where 100% means that the system never fails whereas in 99% availability, we can have max of 3.65 days of downtown in a year. Due to these factors, high availability is now a critical and "must-have" requirement for large web applications.

One of the important goals of high availability is to eliminate single point of failure in our infrastructure. Any component which does not have redundancy becomes single point of failure.

Each layer of our stack must have redundancy to eliminate single points of failure. For example, if we have two identical, redundant web servers behind a load balancer, the traffic coming from clients will be equally distributed between the web servers, but if one of the servers goes down, the load balancer will redirect all traffic to the remaining online server.

3. Scalability in Web Applications

3.1 Introduction

As we know, a web application is a computer program that utilizes various web technologies and web browser to perform tasks over Intranet/Internet.

Web applications make use of combination of server-side scripting languages like PHP, ASP.NET, JSP etc. to handle processing, retrieval and storage of information, and client-side scripting languages like HTML and JavaScript to present information to users. The combination of these technologies allow users to interact with application using online forms, content management systems etc.

Here is what a typical web application flow looks like:

- **User** triggers a request to the **web server** over the **Internet**, either through a web browser or the application's user interface
- **Web server** forwards this request to the appropriate **application server**
- **Web application server** performs the requested task – such as querying the **database** or processing the data – then generates the results of the requested data
- **Web application server** sends results to the **web server** with the requested information or processed data
- **Web server** responds back to the client
- The requested information is then displayed to the user on his browser

There are various stages and layers like Internet, Web Server, Application Server, Database server and application itself are involved in the whole process. Bottleneck at any level can have impact on the performance of the application. To achieve scalability, it is important that all of these layers are scalable. This section discusses about impact of various layers on scalability, issues and challenges of scalability which will later help us in understanding the approaches to be taken for overcoming those issues and challenges.

3.2 Scalability Layers

Understanding various layers involved is the first step in understanding scalability. The following diagram depict the layers in web application.

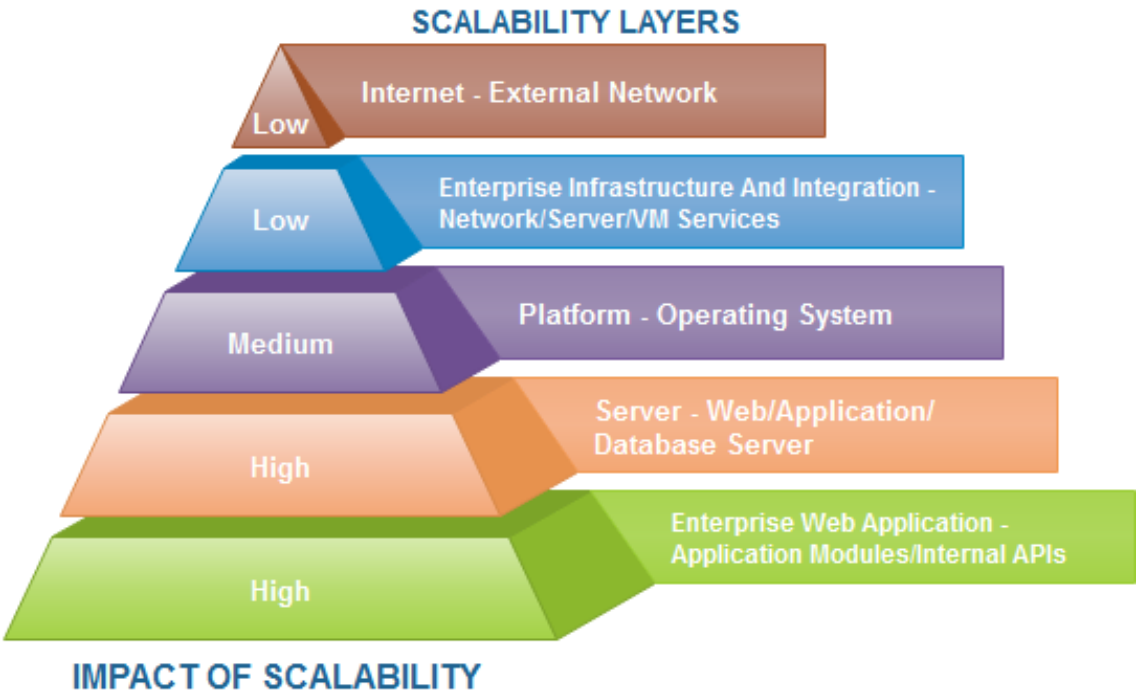

Figure 8: Scalability Layers

The depicted layers are based on their sequence and order in which they contribute to scalability of web application in the request processing chain. For example, when the user makes a request in an enterprise web application, the request will originate from the user's browser/device and then it reaches enterprise infrastructure components such as the firewall, load balancer, and enterprise network. The hosting platforms such as operating systems will then receive the request. The request will then be routed to the corresponding enterprise web server and application server, which then sends the request to the application.

As can be seen from the diagram, the maximum impact on scalability comes from application we design, then from the Web Servers, Application Servers and Database Servers on which they are hosted and then to the platform where they are hosted. The most important point to be remembered is that the Scalability cannot be an afterthought. The application has to be designed to be scalable. The following section discusses about dimensions of scalable web application.

3.3 Dimensions of Scalability

As scalability applies to multiple layers and multiple components, the meaning of scalability varies based on context. The two of the most important dimensions of scalability are:

3.3.1 Load Scalability

The Load Scalability indicates the ability of the application to handle an increase in workload without significant degradation of application performance. The workload can increase in number of users accessing the application, data volume, number of batch jobs, number of service requests etc. If the performance of the application remains within an acceptable range with an increase in workload, it can be called as load scalable.

3.3.2 Functional Scalability

The functional scalability indicates the ability of an application to add additional functionalities without significant degradation in performance. It is measured by the effort and cost with which new functionality can be added to the existing application with minimal impact to the application's performance.

3.4 Issues and Challenges in Scalability

The ability to scale is measured in different dimensions. Some of the main scalability issues are:

Handling more data
This is one of the most common challenge. Processing more data puts load on the system, as data needs to be sorted, searched through, read / write from / to disks, written and travel over the network.

Handling higher concurrency levels
Concurrency measures the number of users the system can serve at the same time. Concurrency is difficult to achieve as servers have a limited amount of central processing units (CPUs) and execution threads.

Handling higher interaction rates
The third dimension of scalability is the rate of interactions between the system and the users. The rate of interactions measures how often the user exchange information with the servers. The rate of interactions can be higher or lower independent of the amount of concurrent users. The main challenge related to the interaction rate is latency.

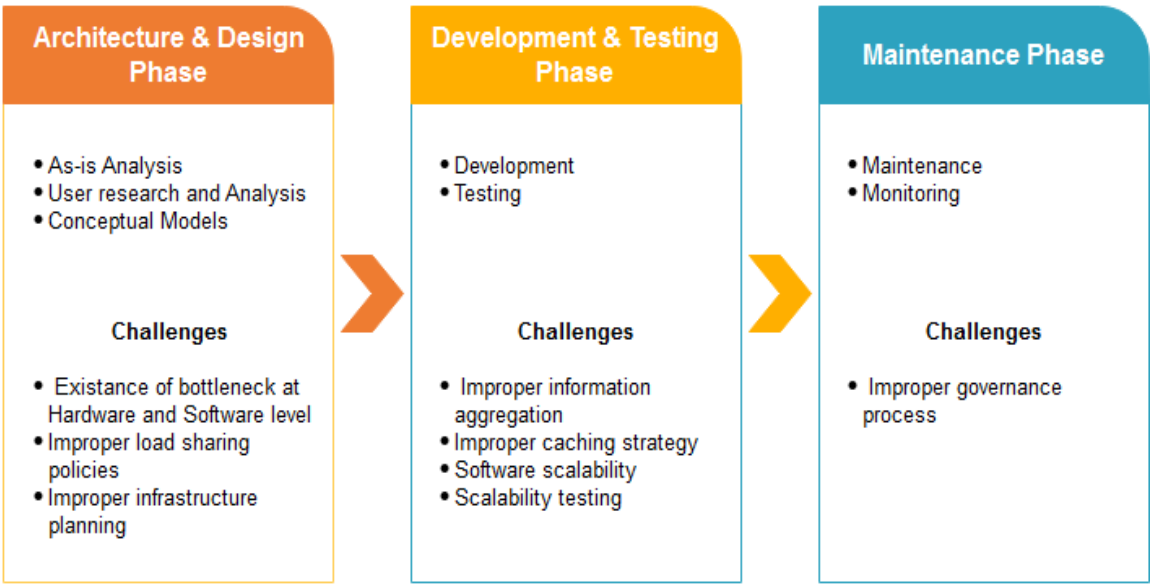

Figure 9 : Issues and challenges in scalability

As depicted above, most of the challenges are related to the application architecture and deployment architecture/hardware.

3.4.1 Scalability challenges at the software level

The following are the important issues which can have impact on the scalability of the application:

Hardware and software bottlenecks in web application

A bottleneck is essentially a software or hardware component that can cause congestion for normal processing and that impact the overall performance, throughput, and other performance parameters. For example, a single-instance web server with limited CPU and memory can become a bottleneck during heavy loads; the web server would consume lot of system resources and the response will be very slow. Similarly, the database connections can also become a bottleneck with impact on response.

Bottleneck affects multiple quality attributes such as performance and availability, as well as scalability, the processing chain is impacted and the system fails to handle a heavy workload. The overall impact of scalability also depends on the layer in which the bottleneck has occurred. For example, if the bottleneck occurs at the database server layer and if the application server has a caching mechanism to cache the data from the database server, then the impact will not be visible immediately, because the requests do not go to the origin database. Hence, scalability will not be affected until the cache expires.

Improper caching strategy

In simple terms, a cache is a temporary storage of critical data that is normally located locally or closest to areas where the data is used. Caching is done to reduce the resource calls and increase the performance and response time.

Caching has a multifold impact on scalability:

- Caching reduces the load and requests on other upstream systems, thereby helping all constituent systems to scale better.
- Caching also reduces the impact of a bottleneck.

A badly designed caching strategy or the absence of a caching strategy will make the load go directly to all involved origin systems, and their scalability limits will be impacted.

Improper data/information aggregation

Information or data aggregation is one of the main features of enterprise web applications, wherein the data are retrieved from multiple data sources through service calls, mash-ups, and other aggregation techniques. In most of the cases, a single web page would end up aggregating data from 2-3 distinct data sources.

Aggregation of data from multiple systems of records can impact scalability if they are not fully tested for peak loads. Services-based information aggregation is the most commonly adopted integration practice. However, if the source systems are not designed to handle the expected load, then the service calls would quickly pile up, leading to memory or thread exhaustion. If the data sources and services used in data aggregation have scalability limits, this directly impacts the end-to-end scalability. There are some techniques such as caching, asynchronous loading, on-demand loading, and partial-page rendering that can minimize the impact of this on overall scalability.

Scalability of application software components

Application code is one of the primary influencers of scalability. This includes software components at all layers such as presentation components, business components, and interfacing components. From a scalability standpoint, enterprise integration components that interact with external interfaces and the components that require high memory or CPU (such as factory classes that create objects, file parsers) play a vital role.

Non-scalable software components affect the scalability of the entire system. If any of the software components are not scalable, this acts as a bottleneck during peak loads.

Absence of scalability testing

It involves identifying and simulating all scalability scenarios of the real world on all applicable components, layers, and systems. Testing has to be done in isolation as well as in combination of these components. A thorough understanding of the user base, key processes, and participating systems is key to design the testing scenarios, which closely mimic their real-world counterparts. A comprehensive scalability testing would uncover components which cause memory leaks, continuous CPU spikes, resource utilization, ability of infrastructure components to handle peak load. Minimal or incomplete scalability testing makes the application as well as the system vulnerable to scalability issues.

3.4.2 Scalability challenges at the hardware level

Listed below are the scalability challenges at the hardware and process level

Improper infrastructure planning

Infrastructure planning involves designing optimal infrastructure elements such as server hardware, network capacity, storage capacity to satisfy the scalability, availability and performance requirements. This is done through sizing and capacity planning activities.

One of the main infrastructure-related issues is to have non-clustered configuration wherein servers work in 'stand-alone' fashion without any standby or backup mechanisms. When the application is under heavy stress, a single server would become the bottleneck and its hardware configurations become the limiting factor. Without a cluster of server nodes it would not be possible to distribute the load and hence there would be no backup option. Similarly using network interfaces, memory and hard disk capacity, and CPU cores without adequate sizing and capacity planning would pose serious challenges to scalability.

Improper or nonexistent load-sharing policies

Load sharing is effective distribution of load among available servers. In a huge enterprise application, all layers such as the web layer, application layer, data base layer, and service layer needs load balancing strategy to distribute the load optimally among the servers. Load balancers employ various policies and work distribution algorithms such as round-robin, hash based, or response-time-based algorithms for optimal load distribution.

Load distribution is vital for systems to scale and respond during heavy loads. Requests should be optimally distributed among web servers, application servers, database servers, and other upstream systems. This will not only ensure optimal resource utilization but will also ease the load on each of the available servers, improving response time. If suboptimal load-sharing algorithms are used, a single server would become a bottleneck.

4. Approach for Scalable Architecture

4.1 Introduction

As we discussed earlier, building a scalable Web Application does not happen by accident and the system is not automatically scalable. The basic strategy for building a scalable system is to design it with scalability in mind from the start. The initial design must be engineered to scale to meet the requirements of the system. Further, it should also include features that provides options to meet future growth requirements.

In the previous section, we have discussed about various issues and challenges which have impact on scalability of an application. Since the maximum impact on scalability is from the application we design and the infrastructure they are hosted on (i.e. Web Servers, Application Servers, Database Servers etc.), it is important that proper planning is being done and best practices are followed during design, development and hosting of application.

In this section, we will discuss an approach called ***"Scalable Cube Model"*** which can be very effectively applied to build scalable web applications. *Martin Abbott & Michael Fisher* in their book *The Art of Scalability* has provided the model called "AKF Scale Cube Model" which can be applied while architecting highly scalable applications. The Scale Cube model can be applied broadly to any software solution for building scalable architectures. This section explain how the approach can be applied at both application servers and database Servers levels.

4.2 AKF Scale Cube

The AKF scale provides three options using which the scaling of web application is achieved:

- Replicate the entire system (horizontal duplication & Cloning);
- Split the system into individual functions, services, or resources (functional or service splits); and
- Split the system into individual chunks (Data Partitioning/Sharding).

The AKF Scale Cube conceptualizes these three categories as X-axis, Y-axis, and Z-axis of a Cube. The following diagram represents the AKF cube:

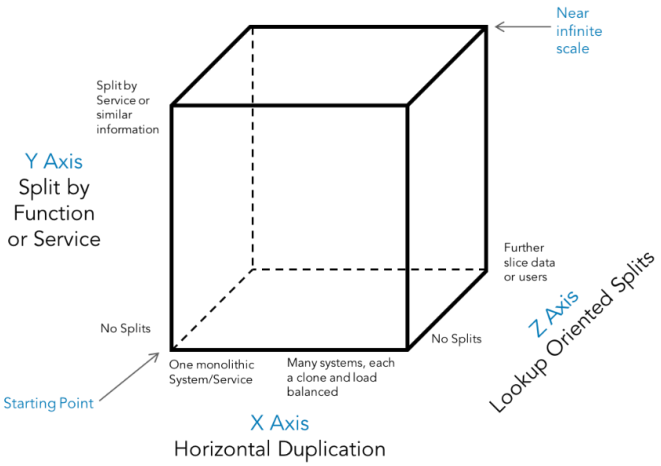

Figure 10 : AKF Cube

The starting point for application is with coordinates (0,0,0) which is the point with least scalability. It is where the application is usually deployed on a single physical server.

As we move along any axis and make changes to our architecture, the scalability of the system increases. Each axis has different kind of impact on scale characteristics of an application. For example, split along the x-axis (Replication / cloning) will allow handling of growth of transactions very well but will not have much impact on system's ability to store, retrieve, and search through data.

In the ideal case, we can design a system for infinite scalability along all three axis, if cost is not the consideration. But, in real life implementations, we should choose the most suitable and cost effective solution based on our scalability requirements.

The X-Axis of the Cube

In X-Axis, we replicate the service on multiple servers. The throughput is increased by this horizontal duplication as the service gets replicated and more servers are now available for processing requests. It is also known as horizontal scaling or scaling out. For example, we can use replicas of Web server behind load balancer where the load is distributed by load balancer to all the web servers. We can also run multiple instances of web Server on single server with good configuration. For example, in Vahan and Sarathi applications under transport projects three tomcat instances are configured per server with total 12 Tomcat instances on four physical servers to handle load.

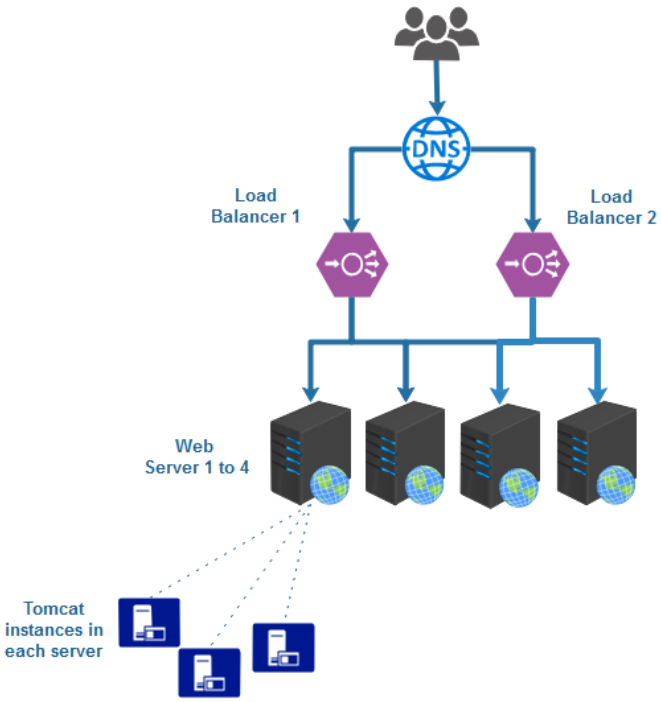

Figure 11 : AKF Cube - X-Axis Example

The x-axis does not scale well with increase in data or with complex transactions that require special handling. If each transaction can be completed independently by all replicas, then the performance improvement is proportional to the number of replicas.

If the processing of transactions require these replicas to communicate with each other, the scaling gets impacted. For example, transactions that write new data, which needs to be communicated to all replicas, may enforce these replicas to hold off update transactions on the data until all replicas have received the change.

The Y-Axis of the Cube

The 'Y' axis of the cube represents splitting the system on service or functionality. This helps in scaling out the system as separate dedicated resources can be allocated to each service/Function. In our cloud deployment, we use this technique by having separate VMs for Web Server and Database Server. If both are deployed in single machine, they will compete for CPU, RAM, disk buffer cache etc. By moving these two major functions (Web Server, Database Server) on two separate VMs, both are able to perform better with available dedicated resources. We can have another server as application server to handle the middleware.

Similarly, we can split our application functionally, with each function deployed on separate sever or group of servers. We split the application into multiple, different services with each service responsible for one or more closely related functions. For example, in Vahan

application, Homologation function is implemented as separate service which interacts with Vahan Service.

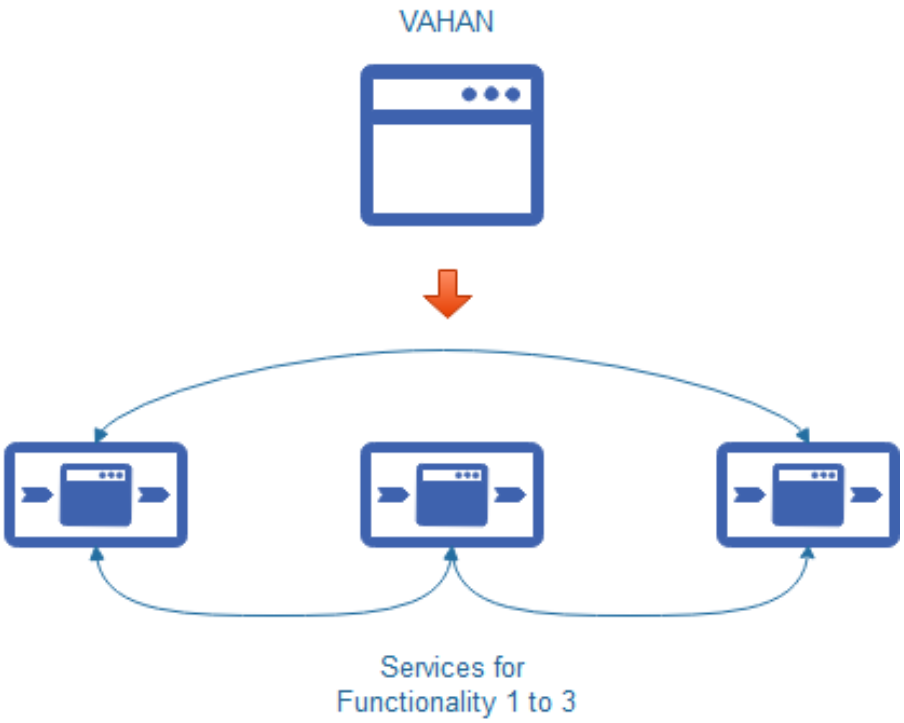

Figure 12 : AKF Cube - Y-Axis Example

To separate the functions, the same has to be loosely coupled to each other. This helps them to scale independently. Micro services support such a scaling model, as they are independent, smaller, fine-grained services with their own database.

Further, we can apply 'X' axis split to each function/service. Combining both X and Y axis scaling, we can get better scalability because each service can be deployed with load balancer. Scaling individual parts is less complicated to replicate and less expensive as we can add capacity to the function/service which needs it.

In addition to splitting along subsystem boundaries, y-axis scaling can also be used by splitting along workflow or transaction type. For example, the citizen interface of an application with functionalities like entering application, upload document and fees payment etc. can be deployed as a separate instance. After the application is submitted, the same can be moved to another instance for approval and verification workflow. This will help in better management of the application and its underlying infrastructure.

Similarly, we can give specific user categories special treatment by sending the traffic from these categories of users to dedicated servers.

So, the 'Y' axis split can be applied by:

- Splitting by Service with each service on its own set of servers
- Splitting by Function with each subsystem related to function on its own set of servers
- Splitting by Type of Users
- Splitting by Type of Transactions

The Z-Axis of the Cube

In 'Z' Axis of the cube, we scale the system by splitting the data into identifiable segments, each of which are allocated dedicated resources. In 'Z' Axis splitting, we divide the data instead of application. For example, we can divide our database by period (5 years each). We can also divide on the basis of States/Region. We can have separate database for each state.

Queries related to a particular state are forwarded to the server related to the state. Queries that are related to multiple states are sent to all the appropriate database servers and the responses are combined.

Implementation of 'Z' axis splitting requires considerable changes to the application. Thus scaling on the Z-axis is considered only when scaling using 'X' axis and 'Y axis are not able to meet our requirement.

Some of the other methods used for Data partitioning are:

- By Hash Prefix
- By Organizational Division: For example, Administration, Ports, Inland waterways
- By Type of User
- Arbitrary Grouping : If the system can scale for 30000 users, then we put another server for next group of users/

Now we will briefly discuss about using AKF scale cube for scaling at the application level and at database level.

4.2.1 The AKF Scale Cube for Applications

The X-Axis of the AKF Application Scale Cube

As explained earlier, the x-axis of the AKF Application Scale Cube represents cloning/replication of services. For example, if we deploy application that is scaled using 'X' axis, we deploy the application on multiple systems behind a load balancer and each of the system can respond to any request coming with the same result.

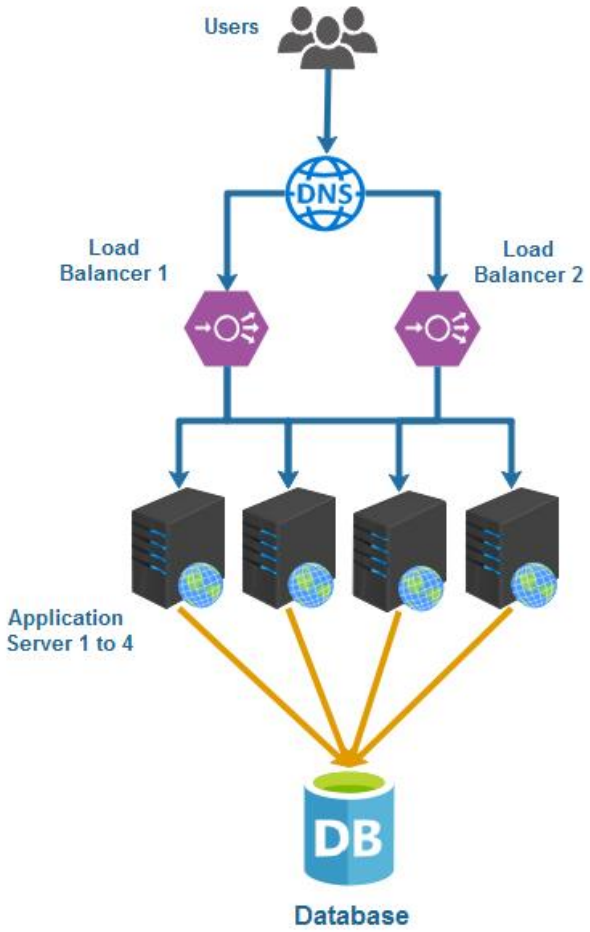

Figure 13 : AKF Scale Cube for Application: X-Axis Example

In the case of stateless application, the request can be forwarded to any of the system for processing. In the cases where persistency or state needs to be maintained, transactions from a specific user is simply linked to one of the instance using session cookies from load balancer.

Some of the major benefits of this approach are:

- The 'X' axis split approach is simple to implement
- It allows near infinite scale from the point of view of inbound number of requests/transactions
- It does not increase the complexity of deployment environment

For example, if there is increase in number of transactions from 1000 requests per second to 10000 requests per second, we only need to add 10 times the number of systems or cloned services to handle the increase in requests.

The Y-Axis of the AKF Application Scale Cube

As explained earlier, the y-axis represents separation of work responsibility within our application. In Y Axis Split of application, we separate/split our application based on functionality in separate modules or micro-services. The services/modules are independently deployed. This helps in routing the specific requests to specific modules. For example, in Transport MMP, separate modules are deployed on separate set of servers. The data is shared through Database connections or APIs. Each module/service is now scalable applying 'X' axis of the cube.

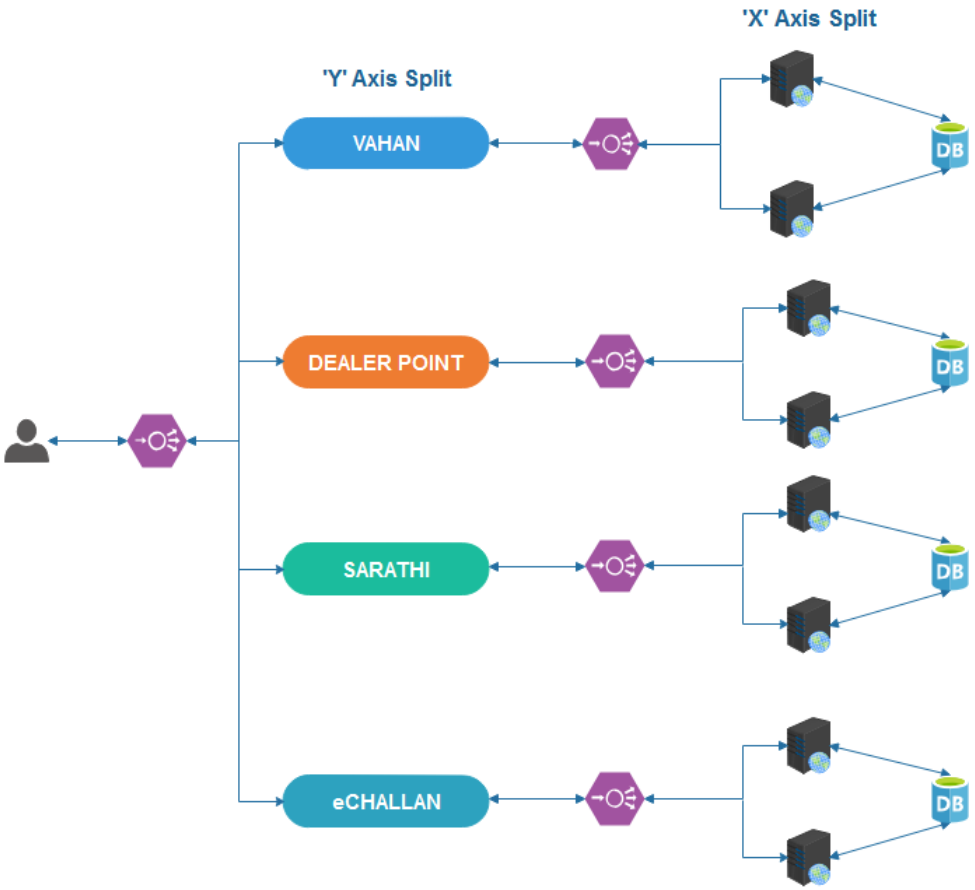

Figure 14 : AKF Scale Cube for Application: Y Axis Example

This independent management of functions allows for right sizing of servers on the X-Axis based upon unique load characteristics of each module. If planned properly, it also allows us to select most suitable technology stack for our modules to meet its specific requirements. Even more significantly, it also allows for right-choice selection of the application stack technology to match the specific need.

Some of the benefits associated with 'Y' axis split are:

- Y-axis split helps in addressing the issues related to complex and large data sets and code base.
- They help in scaling the transaction volume as well with each independent set. The code also gets split in modules and services helping in better maintainability.
- Y-axis splits also helps in reducing the processing time of a transaction as the data and instruction sets that are being executed or searched are smaller.

Some of the drawbacks associated with this split are:

- The implementation of 'Y' Axis split is more complicated than the 'X' Axis split as it requires re-architecting of the application in services.
- The operations and infrastructure team need to manage more number of servers thereby adding more complexity.
- The URL/URI structures grow resulting in referencing of services which requires complete understanding of the structure and deployment architecture.

The Z-Axis of the AKF Application Scale Cube

The Z-axis of the Application Scale Cube is a split based on a particular value like user type, region etc. which is determined at the time of transaction. For example, all requests from a particular user type are forwarded to a particular set of servers. Similarly, all requests of a particular region are forwarded to a particular set of servers.

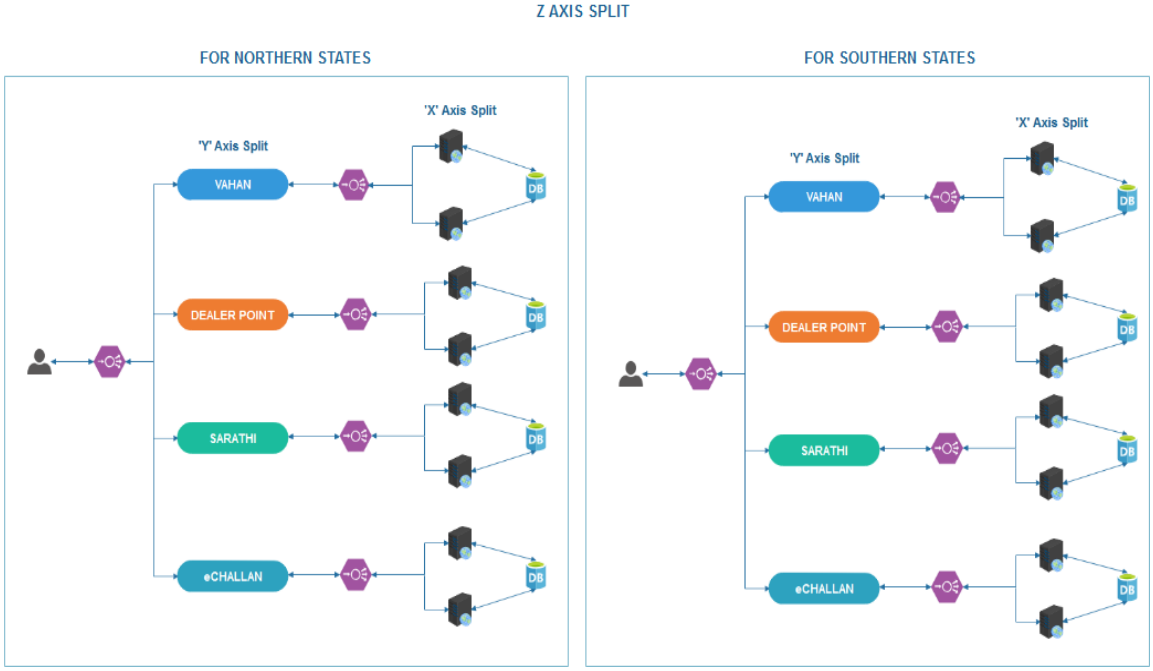

Figure 15: AKF Scale Cube for Application: Z Axis Example

In z-axis split, the partition is being carried out not only of the transactions but the data necessary for processing of those transactions. To perform a z-axis split, we try to look for similarities among groups of transactions across several services. If a z-axis split is performed along with 'X' and 'Y axis split, there will be number of similar instances of servers for each set.

Some of the benefits of this split are:

- Increases transactional scalability,
- Increases fault isolation,
- Increases the cache-ability of objects necessary to complete our transactions.

Drawbacks:

- Increases the code complexity
- Increases operational complexity as we are now required to monitor more number of servers. Managing same code across similar instances becomes difficult.

4.2.2 The AKF Scale Cube for Databases

As we know, the database is the most common point of congestion in our applications and most complex to scale.
The new AKF Scale Cube for databases now looks like the figure given below:

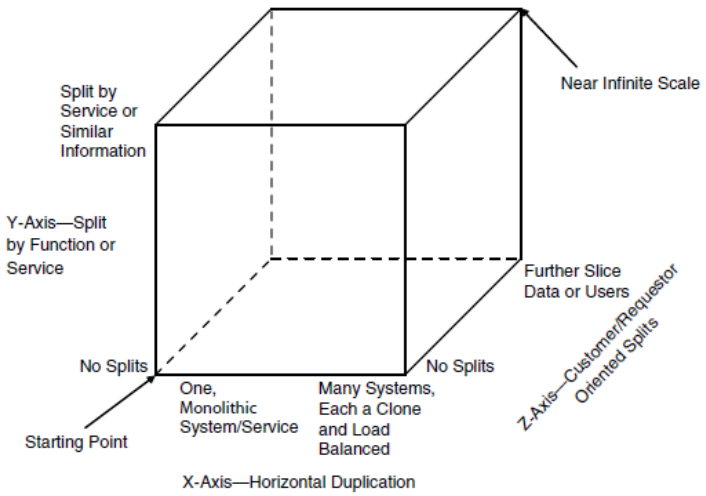

Figure 16: AKF Scale Cube for Database

The lowest left point of the cube with coordinates (0,0,0) represents single database and all access is made to this system. This become a single point failure and is only scalable vertically by adding more memory and computers. Now we will see how we can apply the scale cube splits at database level.

The X-Axis of the AKF Database Scale Cube

In x-axis of the AKF Database Scale Cube, we clone the data in multiple servers. Cloning in x-axis is achieved by using the replication feature available in the database. When we apply 'X' axis scaling on our data tier, all the databases in multiple servers will have exactly the same data except some minor differences resulting due to replication delay.

A request for data can be served from any of the databases. Usually, in this type of implementation, writes are made to a single node within the replicated copies to reduce read/write contention on the nodes. The reads are implemented on other replicated nodes. While implementing this split, we should be careful that if there are any sensitive read operations with time sensitivity, the same should be implemented through the node/server where Write operation is implemented.

The X-Axis split has been implemented in large number of projects where number of transactions are very high to achieve scalability.

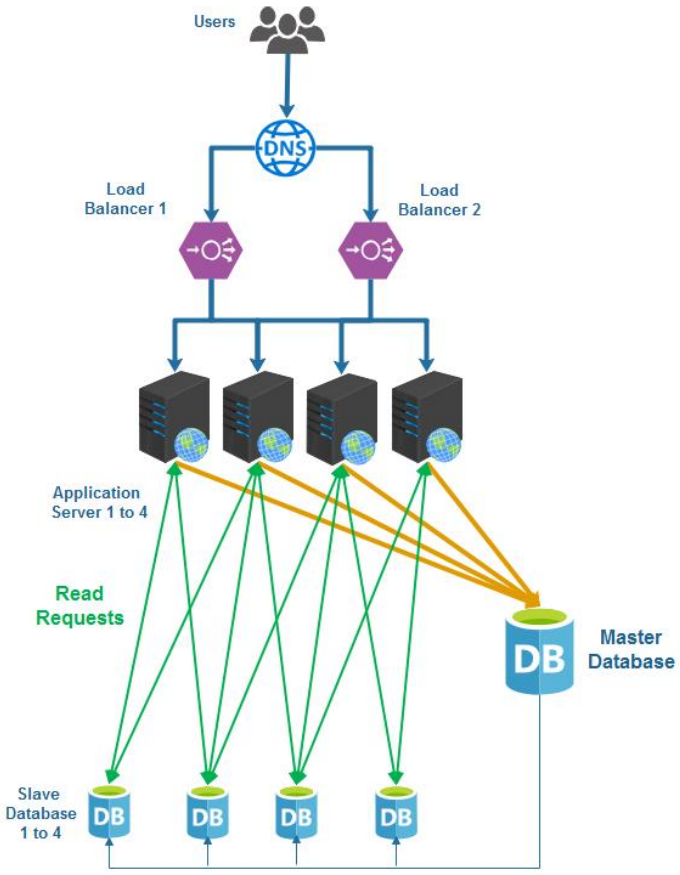

Figure 17: AKF Scale Cube for Database: X-Axis Example

Some of the benefits of implementing this split are:

- X-axis splits allow us to easily scale if there is increase in number of inbound requests/transactions, as we only need to add more read nodes.
- Most of the RDBMS systems have native implementation of replication technologies which allows for maintaining multiple read nodes/servers. The server with "Write" permission is called "Master"" while the other nodes with read only copies are called "Slaves".
- As similar hardware/software is used to achieve this split, capacity planning for the required infrastructure is easy.

Some of the drawbacks associated with this type of split are:

- Cost of replicating large amounts of data is more as x-axis implementations are complete copies of primary database.
- Further, X-axis split on database can handle more number of transactions/read requests, it is not able to address the issues related to increase in database size.

The Y-Axis of the AKF Database Scale Cube

In Y-axis of the AKF Database Scale Cube, we separate our data in separate schemas by aligning with the Y-axis split of application. The 'Y-Axis' split helps in splitting the large monolithic database by separating the data into schemas that are related to applications which are operating on that data.

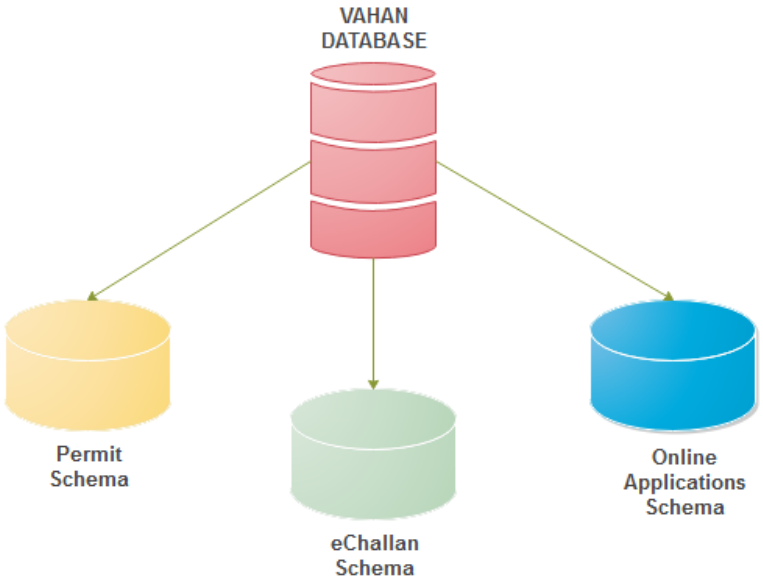

Figure 18: AKF Scale Cube for Database : Y-Axis Example

In a database, this split is implemented by moving tables and data to a different schema or even to another database instance within the same physical hardware. This split requires

changes to the application code to address to the changed schemas. Y-axis splits are most commonly implemented to address the issues associated with a dataset that has grown significantly in complexity or size and which is likely to continue to grow.

Some of the benefits associated with this split are:

Benefits:

- The Y-axis split help in scaling the transaction volumes because the requests are moved to separate logical systems in turn decreasing logical contention for the data.
- At the operational level, this splits help reduce the processing time of transactions as the data being searched and retrieved is smaller and related to the service performing the transaction.

The Z-Axis of the AKF Database Scale Cube

As with the Application Scale Cube, the z-axis of the AKF Database Scale Cube consists of splitting the database based on values that are determined at the time of the transaction. As explained earlier, this split is performed by looking up or determining the location of data based on a reference to the requestor.

The easiest way to think of the difference between y-axis and z-axis splits is to differentiate between things that we know before a request happens and things we must look up or determine at the time of the transaction.

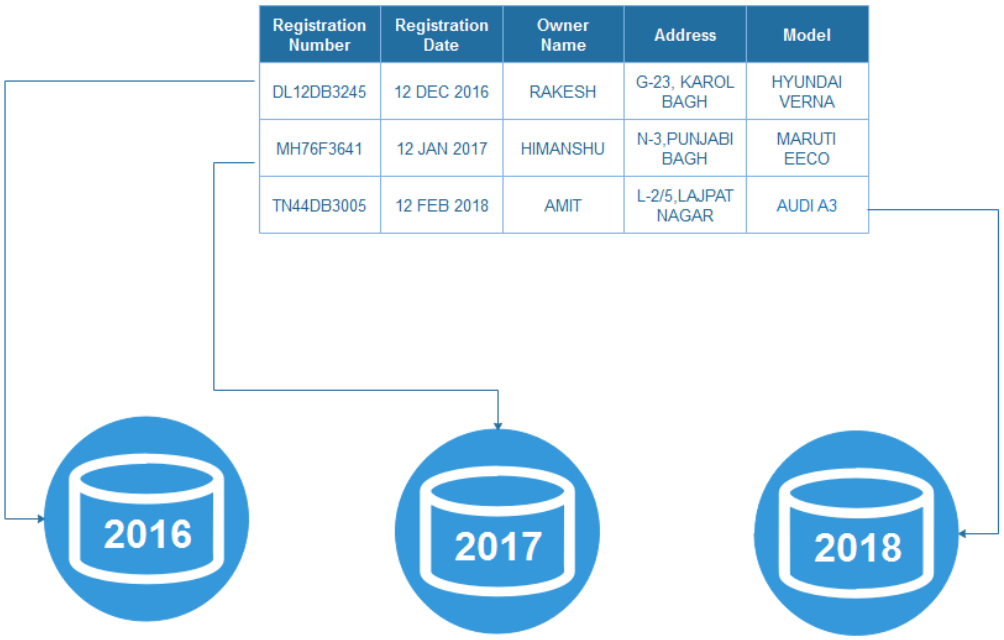

Figure 19: AKF Scale Cube for Database: Z-Axis Example

The split or partitioning is value-based row-level separation of data to allow for storage on separate servers for distributed load. The separation is typically based on the value of key

fields such as State, User Type etc. Scale is achieved by intelligently routing requests for specific contexts (shards) that are independently managed.

Some of the benefits associated with this split are:

Benefits:

- This increases transactional and data scalability of our solution. The load is evenly distributed which makes the system fault tolerant.

Drawbacks:

- The z-axis split is more costly as we need to make changes to our application. Further, the operational management also becomes complex as we need to manage number of servers performing similar functions.
- Joining the data from these multiple systems also becomes difficult for handling queries/searches from these multiple systems.

The Scale Cube helps us keep the critical dimensions of system scale in mind and provides for various options while architecting our solution. All the three splits can be combined to achieve near infinite scalability for our solution.

In the next few chapters, we will discuss about strategies we can apply at various layers like presentation layer, database layer, integration layer etc. of our web application.

5. Scaling strategy for Front-end Layer

As we discussed in third chapter, the web application has multiple layers like network layer, presentation layer, database layer, integration layer etc. The bottleneck at any layer can have impact on the performance of the application. It is important that we plan our architecture to ensure that each layer is scalable to meet increase in demand.

As we know, the frontend layer components allow users to interact with the application. The various components involved in front-end layer of a web application are:

- Client (usually a web browser)
- Network components between the client and the data center
- Data Centre components like Routers, Firewalls etc.
- Load Balancers
- Web Servers

As every request and response has to pass through the frontend layer, it receives most of the traffic. Because of this, it has the maximum concurrency and throughput requirements. The design of application and its deployment architecture has maximum impact of scalability of front-end. In this chapter, we will discuss about some of the strategies which can be used to make front-end of our application scalable.

Our front-end application to be scalable:

1) Has to be mostly stateless
2) Should use caching effectively
3) Should allow horizontal scalability by simply adding more servers

The user's request from browser is sent to the Web Server for further processing and response has to go back to the user. If each request is completely independent, the requests can be processed by any server. This make it possible for us to just add more servers to scale without bothering about any other aspect. But, as we know in our applications, the user makes a request, wait for the response and based on the response makes another request. The server needs to remember the previous interaction (i.e. the state) of the user.

So, carefully managing state is one of the important aspect of scaling the front end of our web application. If we can design our application by removing the user state from our front-end servers, we can scale our front-end layer by just adding more server.

In the following section, we will discuss about Stateful and Stateless servers and how we should manage state to achieve scalability.

5.1 Managing State

As discussed, Stateful server is where the server maintains the state of the user between requests. The following diagram represents the stateful server

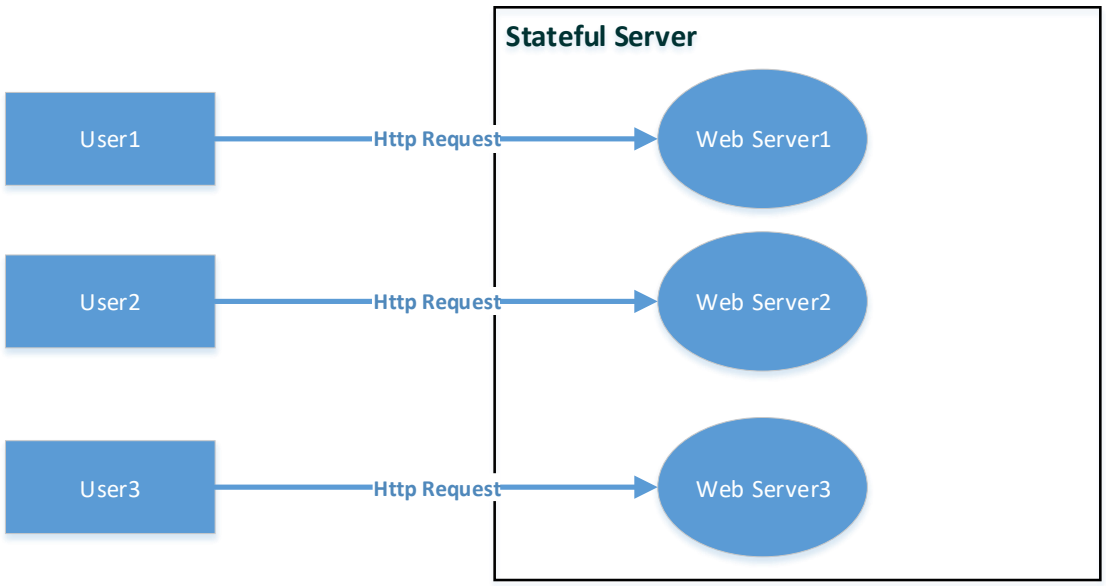

Figure 20: Stateful Server

Here each user session is bound with a particular server instance as the server holds session information about the user which is not accessible to other web servers. One of the drawback with Stateful server is that in case of system crash, the session data is lost.

The Stateless server is where the web Servers are stateless and user is not bound to a particular Web server for a session. This is achieved by making the Web Server Stateless. One way to achieve this is by making session information of all users accessible to all web servers from a centralized session storage. State information is available to all the web servers

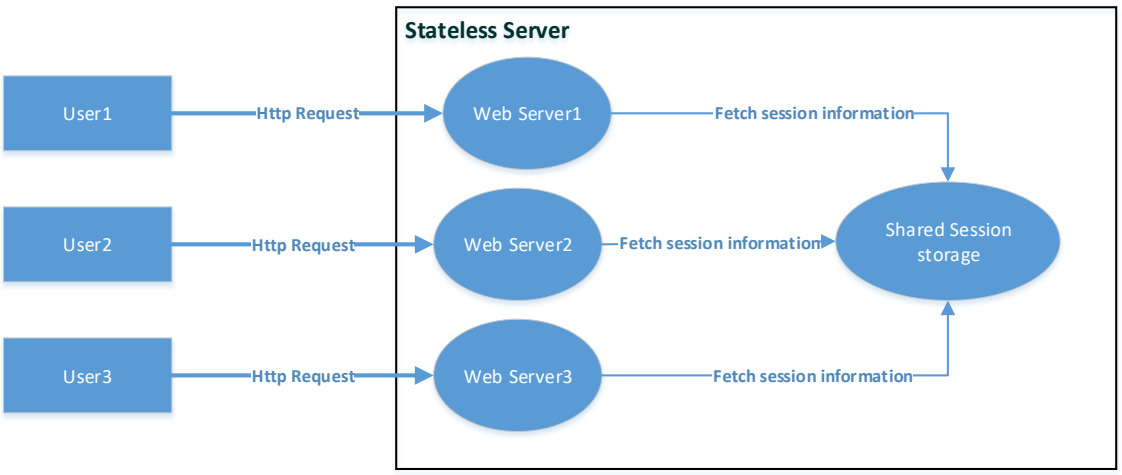

Figure 21: Stateless server

Now we will discuss about the most common types of state information stored in the front-end layer and the way to handle them.

5.1.1 HTTP Sessions

Hypertext Transfer Protocol (HTTP) is the protocol used over the Web. When we visit a website, our requests are typically part of an HTTP session. As we know, HTTP protocol is stateless itself and all requests and responses are independent. However, sometimes we need to keep track of user's activity across multiple requests. For example, when a User logs into our application, no matter on which web page/module he/she visits after logging in, the credentials needs be with the server, until he/she logs out. Therefore, this is managed by creating a session. The web applications developed techniques to create a concept of a session on top of HTTP so that servers could recognize multiple requests from the same user during the user session.

The basic concept behind session is that whenever a user starts using our application, we can save a unique identification information about him, in an object which is available throughout the application, until it is destroyed. Whenever a user wants to exit from our application, we destroy the object with the information.

The sessions are implemented using cookies. When a user sends a request to the web server without a session cookie, the server starts a new session by sending a response with a new session cookie header.

By using cookies, the server can now recognize which requests are part of the same sequence of events. Even if multiple browsers connected to the web server from the same IP address, cookies allow the web server to figure out which requests belong to a particular user.

When the user login, the application stores user identifier and additional data in the web session scope. The data stored in the web session scope is then available to the application on each user request. In the case of Java, a web session scope is usually stored in the memory of the web application container.

To make our web servers scalable, we need to store session information outside the web server so that it is accessible to all the web servers.

There are three approaches to handle this:

- **Store session state in cookies**
  We can store session data in cookies. If the session data is small, there is no problem but if the session data is large, it makes requests slower as the data transmitted in

cookies is to be transmitted in full. Further, it has security implication because of which it needs to be stored in encrypted format.

- **Clustered session management**
  In a clustered environment, the session data is replicated to all the clustered web servers so that it is available to all the web servers. There is no single point of failure in this case.

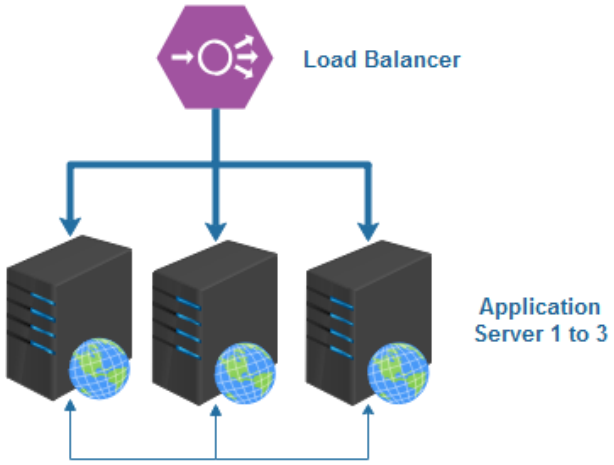

Figure 22 : Clustered Session Management

- **Store session data in external data store**
  In this approach, the web server does not store session data between web requests, which makes it stateless. The session data is stored in some external data storage like database or fie system. Some application servers like JBoss or Tomcat have built-in mechanisms to handle cross-server replication of session data. We also have some other options like Memcached, Redis, DynmoDB etc. The following diagram represents this implementation.

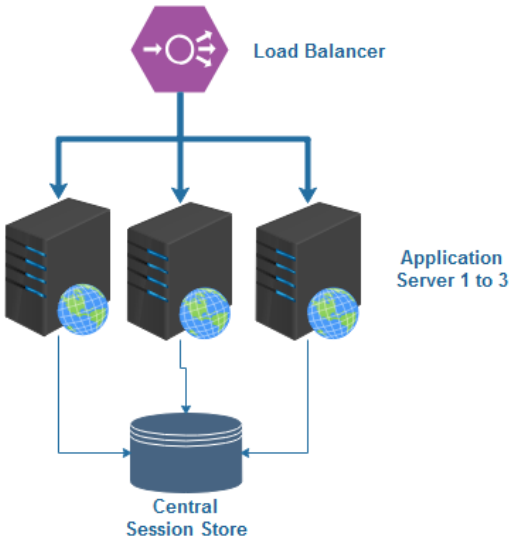

Figure 23 : Session Data in External Store

- **Sticky sessions with Load Balancer**
  As we know, Load Balancer routes user requests to web servers. We can use the feature called ”Sticky Session”, which enables the load balancer to bind a user's session to a specific web server instance. This ensures that all requests from the user during the session are sent to the same web server instance.

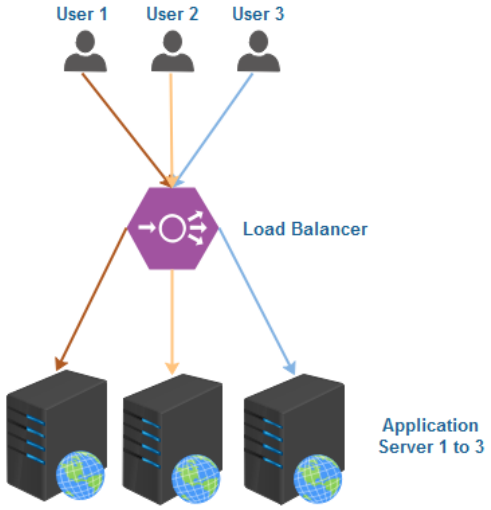

Figure 24 : Sticky session using Load Balancer

This option is easy to implement and no changes are required in the application. Number of large project like e-Transport MMP has implemented sticky session feature.

Following are some of the drawbacks which are observed:

- If the server goes down, session is lost. This is sometime very frustrating for the user as he needs to start session fresh. This is same as storing session data into local web server.
- Depending on "sticky" implementation in load balancer, Load Balancer may direct unequal load to some servers.

Though “Sticky Session” looks like a good solution, it breaks the fundamental principle of statelessness.

Once we allow our web servers to be unique, by storing any local state, we lose flexibility as we will not be able to restart, decommission, or safely auto-scale web servers without braking users’ sessions because the session data is bound with a particular web server.

Instead, better option is to keep all session scope data in cookies or store session data in a shared object store accessible from all web server instances.

5.2 Components of the Scalable Front End

As we know, Front-end layer includes components like web servers, load balancers, Domain Name System (DNS), reverse proxies, and CDN. Now we will look at the scalability impact of each component on the front-end infrastructure and see what technologies can be used in each area. The following figure shows a high-level overview of the key components most commonly found at the front-end layer.

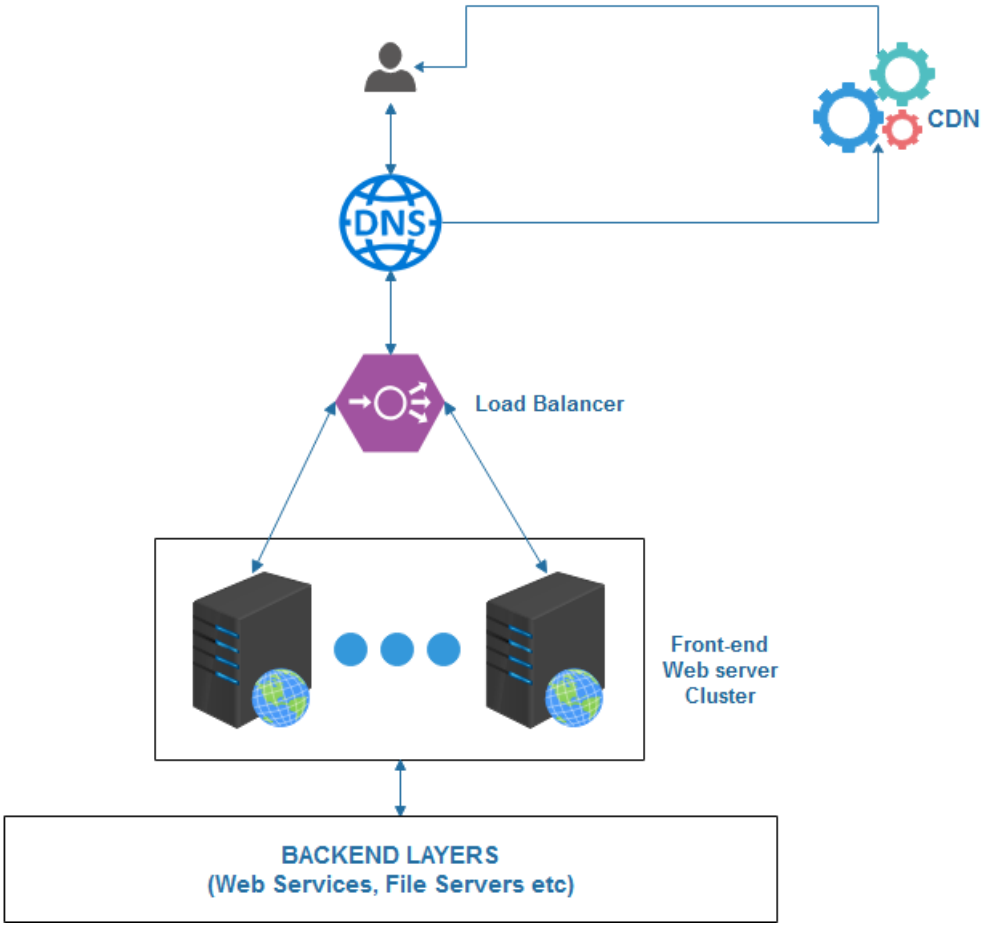

Figure 25: Detailed front-end architecture

In the following sections, we will discuss about each component in more detail.

5.2.1 Load Balancers

As we discussed earlier, A load balancer sits in front of web servers and route client requests to all the available servers and ensuring that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts to send requests to it.

A load balancer performs the following functions:

- Distributes client requests or network load efficiently across multiple servers.
- Ensures high availability and reliability by sending requests only to servers that are available.
- Provides flexibility to add or remove servers as per demand.

By using a load balancer between the web servers and the users, the traffic is routed through the load balancer which helps in hiding the details about underline deployment architecture including server details from outside world.

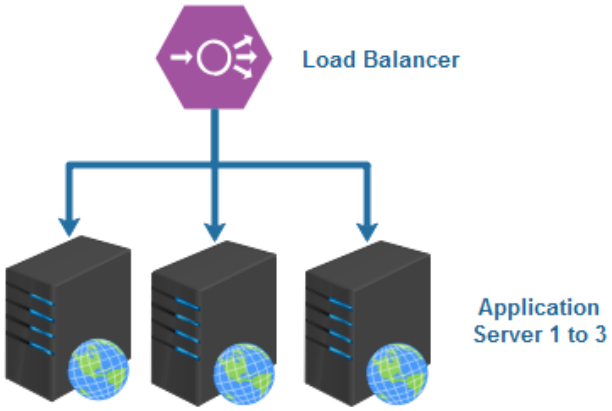

Figure 26: Load Balancer

Benefits of using a load balancer:

- **Hides server maintenance:** Web server can be taken out of the load balancer pool, then wait for all active connections to 'drain' and then safely shut down the web server without affecting even a single client. We can use this method to perform 'rolling updates' and deploy new software across the cluster without any downtime.

- **Seamlessly increase capacity**: We can add more web servers at any time without our clients ever realizing it. As soon as we add a new server, it can start receiving connections; there is no propagation delay as when DNS-based load balancing is used.

- **Efficient failure management**: Web server failures can be handled quickly by simply taking a faulty web server out of the load balancer pool. Even if provisioning a new web server takes time, we can quickly remove a broken instance out of the load balancer pool, so that new connections are not forwarded to that faulty machine.

- **Effective resource management**: SSL offloading, sometimes also called SSL termination, is a load balancer feature allowing us to handle all SSL encryption/decryption work on the load balancer and uses unencrypted connections internally.

Types of Load Balancers

As discussed earlier, there are two categories of load balancers namely, hardware- and software-based load balancers. The primary goal of load balancers is to distribute the workload and increase the reliability and the availability of resources. Both types of load balancers use different routing mechanisms and scheduling algorithms. Some of the major differences between the two kinds of load balancer are capacity, feature set, and the application architecture.

Software-Based Load Balancers

Software-based load balancers are the software component which works as load balancer.

The following are the algorithms implemented by software based load balancers.

- ***Weighted***: are used when we have servers of different configurations. For example, if we have server with different compute and memory, it will be ideal to direct the traffic more towards the servers having higher amount of compute and memory. The other servers will get less traffic

- ***Round-robin*** are useful when we have servers with same amount of compute and memory. These balancers follow the Round Robin algorithm approach and requests are sent sequentially to each server one by one.

- ***Least Connections*** work on the basis of least connections first algorithm. According to this algorithm, the request will be sent to the servers having least number of connections. This algorithm is useful with sticky session.

Two of the popular software based load balancers are Nginx and HAproxy. The main advantage of Nginx is that it is also a reverse HTTP proxy, so it can cache HTTP responses from our servers. Not only can we scale out the web service layer by adding more servers to the Nginx pool, but we can also benefit greatly from its caching capabilities, reducing the resources needed on the web services layer.

HAProxy, on the other hand, is simpler in design than Nginx, as it is just a load balancer. HAProxy has built-in high-availability support (HAProxy stands for High Availability Proxy), which makes it more resilient to failures and simplifies failure recovery.

In both cases, whether we use Nginx or HAProxy, we will need to scale the load balancer ourselves. The capacity limit is most likely reached by having too many concurrent connections or by having too many requests per second being sent to the load balancer.

Both Nginx and HAProxy can forward thousands of requests per second for thousands of concurrent clients before reaching the capacity limit. This should be enough for most

applications, so we should be able to run our web application on a single load balancer (with a hot standby) for a long time.

When the limits of load balancer capacity are reached, we can scale out by deploying multiple load balancers under distinct public IP addresses and distributing traffic among them. As long as the load balancers are interchangeable and the web servers are stateless, we can keep adding more load balancers to scale horizontally.

Hardware Load Balancer

A dedicated hardware load balancer should be considered if a high-traffic website is being hosted in a physical data center. By having hardware load balancers, we mainly benefit from high throughput, extremely low latencies, and consistent performance. Hardware load balancers are highly optimized to do their job, and having them installed over a low-latency network makes a big difference. They can often handle hundreds of thousands or even millions of concurrent clients, making it much easier to scale vertically. The main downside of using Hardware Load Balancer is its cost and the experienced technical resources required to manage it.

5.2.2 Web Servers

As we know, the primary function of a web server is to store, process and deliver web pages to clients. The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP). The pages delivered by web servers are mostly HTML documents, which may include images, style sheets and scripts in addition to the text

The front-end servers should not contain business logic. They should be treated as a presentation layer and web service results aggregation layer only. As the front end is mainly about handling user interactions, rendering views, and processing user input, it makes sense to use technologies that are good at these tasks. It is recommended to use dynamic languages like JavaScript, PHP, Python, Ruby etc. for the front-end web application development, rather than using core java, C, CGI etc.

It is advisable to use same technology stack across layers, as they allow code reuse and teams to master fewer technologies

Once language and framework are selected, we need to select the actual web server on which to run the application. From the scalability point of view, it does not matter much which language we choose for our application and on which web server it is running. As long as the front-end web servers are stateless, we can always scale out horizontally by simply adding more servers.

No matter what web server we choose, the most important thing for your scalability is to keep your front-end machines stateless.

5.2.3 Caching

Caching is one of the most important techniques for scaling the front-end of our web applications. When it comes to scaling the front end of the web application. In place of trying to add more servers to make them respond faster to clients' requests, we can use caching to avoid having the web servers to serve these requests. Caching is so important to the scalability that we have a dedicated chapter on caching in this document. We will have detailed discussion on the subject in the chapter on caching.

The following are the ways in which caching can be used to scale front-end layer.

Integrate a CDN

To scale our front-end, we should consider to integrate a CDN(Content Delivery Network) which can be used as proxy for all of the web requests coming to the web servers. It can also be used solely for static files like images, CSS, and JavaScript files. If we decide to serve all of the traffic via the CDN, we may be able to leverage it to cache entire pages and even AJAX responses. For some web application types, we can serve most of the traffic from the CDN cache, resulting in less load on our servers and better response times.

Use a reverse proxy

Unfortunately, not all web applications can use CDN to effectively cache entire pages. The more personalized our content is and the more dynamic the nature of our web application, the harder it becomes to cache entire HTTP responses. In such cases, we may be better off deploying own reverse proxy servers to gain more control over what is cached and for how long. Most common alternatives for it are reverse proxies like Nginx.

Store data directly in the browser

Another way to use caching in the front-end layer is to store data directly in the browser. Modern browsers supporting the web storage specification lets us store significant amounts of data. This is especially useful when developing web applications for mobile clients as we want to minimize the number of web requests necessary to update the user interface. By using local browser storage from the JavaScript code, we can provide a much smoother user experience, reducing the load on our web servers at the same time.

Using Shared Object Cache

We can also use Shared Object Cache like Redis or Memcached using which we can cache responses in an object cache.

6. Scaling strategy for Database Server

Database scalability refers to the ability of a database server to scale with the increase of workload.

Scalability at the database layer has always been a challenging task for architects which also happens to be one of the areas that is typically underestimated and later leading to major issues. Most of the databases perform efficiently until that threshold of data is reached which is when we start seeing low database performance hampering the overall application performance.

Therefore, it is very important from an architecture point of view to design a scalable database solution that can address not just the current load but also can take care of future needs.

Primary bottleneck for databases are

1) Input/Output(I/O) operations,
2) CPU
3) Memory

Therefore, it is essential to isolate these bottlenecks by distributing the data and resources required for performing these operations.

In the following sections, we will discuss about various database-scaling strategies.

6.1 Database Scaling Strategies

There are two broad strategies for scaling database systems:

Vertical Scaling or Scaling Up

Vertical scaling or scaling up is the process of adding resources, such as memory or more powerful CPUs to an existing server. Adding resources to a system results in performance gains, but this requires reconfiguration and downtime. Furthermore, there are limitations to the amount of additional resources that can be applied to a single system.

Vertical scaling has been a standard method of scaling for traditional RDBMSs. However, every piece of hardware has limitations that, when met, cause further vertical scaling to be impossible. For example, if our system only supports 256 GB of memory, when we need more memory we must migrate to a bigger server, which is a costly and risky procedure requiring database and application downtime and migration process.

Horizontal scaling or scaling out

Horizontal scaling or scaling out is the process of adding more nodes (new servers) to an existing system. However, Horizontal scaling depends on the capability of software to use these networked computer resources and other technology constraints.

There are various reasons why we would like to distribute a database across multiple servers i.e. horizontally scale them:

- **Scalability**
  If data volume, read load, or write load grows bigger than what a single server can handle, we can potentially spread the load across multiple machines.

- **Fault tolerance/high availability**
  If the application needs to continue working even if one server (or several servers, or the network, or an entire datacenter) goes down, we can use multiple servers to give redundancy. When one fails, another one can take over.

- **Latency**
  If we have users distributed geographically, it would be a good choice to deploy servers at various locations worldwide so that each user can be served from a datacenter that is geographically close to them. This reduces the network latency.

6.1.1 Architectural approaches for Horizontal Database Scaling

Databases scalability is often implemented by clustering. With clustering, multiple servers are used to serve database requests. There are two predominant architectures for implementing database clustering: shared-disk and shared-nothing.

There are two architectural approaches related to horizontal scaling of database:

Shared-disk architecture

It uses several servers with independent CPUs and RAM, but stores data on an array of disks that is shared between the servers, which are connected via a fast network. With a shared-disk environment, all of the connected systems share the same disk devices. Each processor still has its own private memory, but all the processors can directly address all the disks. This means that there is no need to break apart data into separate partitions because all of the data is shared in shared-disk implementations.

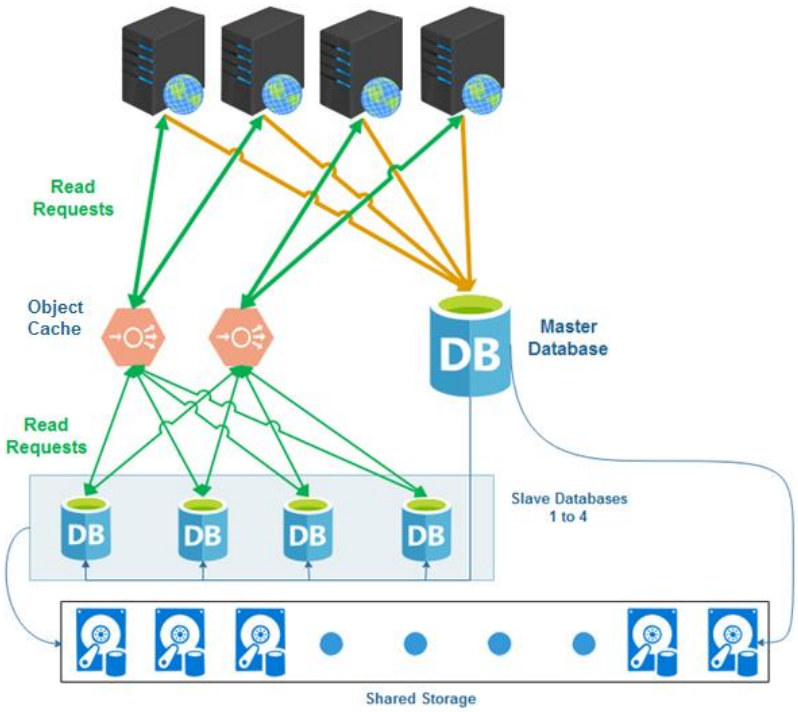

Figure 27: Shared Disk Storage

It is important to understand that only the disks are shared. Main memory is not shared; each processor has exclusive access to its memory. Because any processor can cache the same data from disk, a cache coherency mechanism is necessary to ensure consistency when multiple nodes modify the data. A lock management capability is also required in the system to manage the consistency of the data as it is being modified by multiple nodes.

Shared-disk is usually viable for applications and services requiring modest shared access to data, as well as applications or workloads that are very difficult to partition.

Oracle RAC is a good example of database systems that implement a shared-disk approach for clustering.

Shared-Nothing Architectures

In this approach, each server or virtual server running the database software is called a node. Each node uses its CPUs, RAM, and disks independently. Any coordination between nodes is done at the software level, using a conventional network.

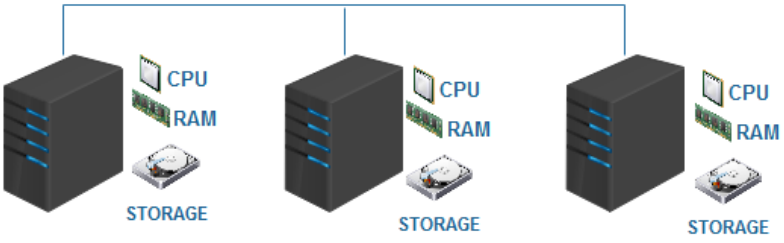

Figure 28: Shared Nothing Architecture

Shared-nothing Architecture can scale up to thousands of servers because they do not interfere with one another i.e. nothing is shared hence it improves the overall scalability. The scalability of shared-nothing clustering makes it ideal for read-intensive applications.

A disadvantage of shared-nothing is that a partitioning scheme must be designed to apportion the data across the nodes of the database. We will discuss various approaches to partitioning later in this chapter.

6.1.2 Replication

Data Replication is the technique used for storing data in more than one site or node. It is useful in improving the availability of data. In simple terms, Replication is copying data of a database from one server to another server so that all the users can share the same data without any inconsistency. The result is a distributed database in which users can access data relevant to their tasks without interfering with the work of others.

In data replication, duplication of transactions are carried out on ongoing basis to ensure that the source and replica are in synchronization.

Following are some of the advantages of using replication:

- Reduce latency by keeping data geographically close to the users
- Increase availability by allowing the system to continue working even if some of its nodes have failed
- Increase read throughput by scaling out the number of servers that can serve read queries

There are two main approaches to replication:

Master-Slave replication
In Master-Slave Replication, Clients send all writes to a single node (the Master), which sends a stream of data change events to the other replicas (Slaves). Reads can be performed on any replica, but reads from Slaves might be stale.

Multi-Master replication
Clients send each write to one of several Master nodes, any of which can accept writes. The Masters send streams of data change events to each other and to any follower nodes.

Each node that stores a copy of the database is called a replica. To ensure that all the data ends up on all the replicas, every write to the database needs to be processed by every replica; otherwise, the replicas would no longer contain the same data.

The most common approach used in various projects is master–slave replication (also known as active -passive or leader-follower based replication).

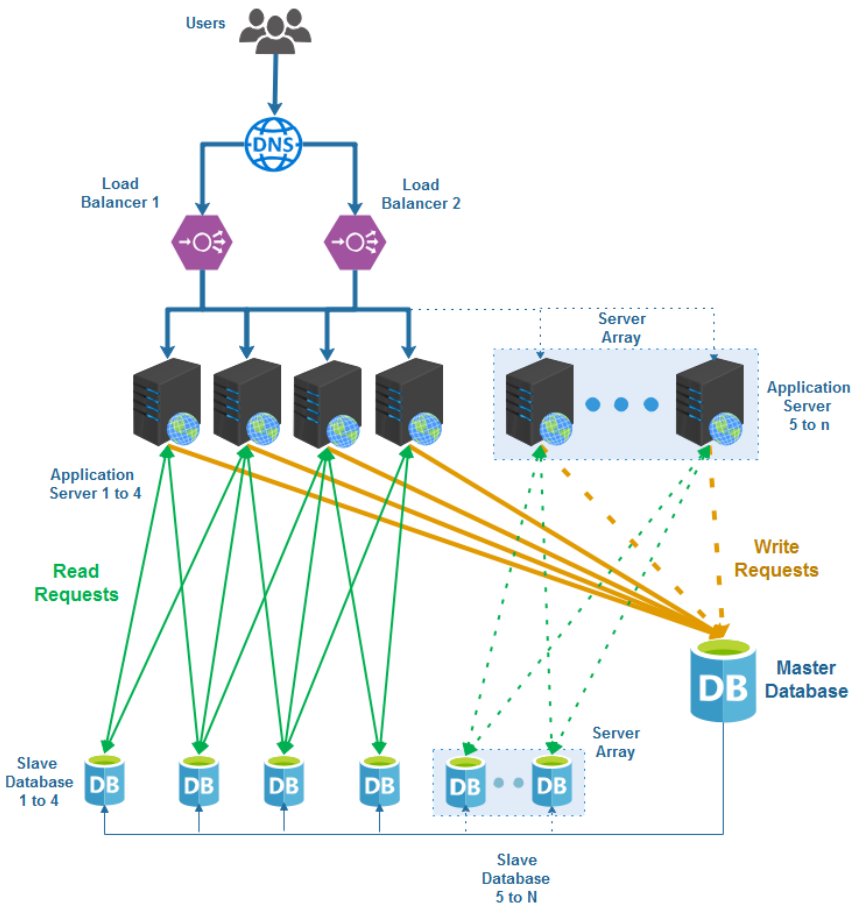

Figure 29 : Data Replication

It works as follows:

1) One of the replicas is designated the master (also known as leader or active or primary). When clients want to write to the database, they must send their requests to the master, which first writes the new data to its local storage.

2) The other replicas are known as slaves (read replicas or followers or hot standbys). The data change is sent by the master to all of its slaves as part of a replication log or change stream. Each slave takes the log from the master and updates its local copy of the data-base by applying all writes in the same order as they were processed on the master.

3) When a client wants to read from the database, it can query either the master or any of the slaves. However, writes are only accepted on the master as the slaves are read-only for the client.

This mode of replication is a built-in feature of many relational databases, such as PostgreSQL.

Synchronous Versus Asynchronous Replication

The primary difference between synchronous replication and asynchronous replication is the way in which data is written to the replica nodes. In synchronous replication data is written to the master and slaves simultaneous. An important detail of a replicated system is whether the replication between nodes happens synchronously or asynchronously.

Synchronous Replication

The advantage of synchronous replication is that the slave is guaranteed to have an up-to-date copy of the data that is consistent with the master. If the master suddenly fails, we can be sure that the data is still available on the slave nodes.

The disadvantage is that if the synchronous slave doesn't respond due to a crash or network failure, the write cannot be processed. The master blocks all writes and waits until the synchronous replica is available again. For that reason, it is impractical for all slaves to be synchronous as any one node outage would cause the whole system to halt.

In practice, if we enable synchronous replication on a database, it usually means that one of the slaves is synchronous, and the others are asynchronous. If the synchronous slave becomes unavailable or slow, one of the asynchronous slaves is made synchronous. This guarantees that we have an up-to-date copy of the data on at least two nodes: the master and one synchronous slave. This configuration is sometimes also called semi-synchronous.

Asynchronous Replication

Often, master-slave based replication is configured to be asynchronous. In this case, if the master fails and is not recoverable, any writes that have not yet been replicated to slaves are lost. This means that a write is not guaranteed to be durable, even if it has been confirmed to the client.

A fully asynchronous configuration has the advantage that the master can continue processing writes, even if all of its slaves have fallen behind.

Each approach to replication has advantages and disadvantages:

- Single-master replication is popular because it is fairly easy to implement and there is no conflict resolution to worry about.
- Multi-leader and leaderless replication can be more robust in the presence of faulty nodes, network interruptions, and latency spikes—at the cost of being harder to reason about and providing only very weak consistency guarantees.

6.1.3 Data Partitioning

Data partitioning is another scalability techniques next to functional partitioning and scaling by adding clones/nodes. The core motivation behind data partitioning is to divide the data set into smaller pieces so that it could be distributed across multiple servers. Partitioning a database improves performance and simplifies maintenance. By splitting a large database table into smaller, individual tables, queries that access only a fraction of the data can run faster because there is less data to scan. Maintenance tasks, such as rebuilding indexes or backing up a table, can run more quickly. Data Partitioning is of two types i.e. Vertical Partitioning and Horizontal Partitioning.

Vertical Partitioning

Vertical partitioning is achieved by dividing a table into multiple tables that contain fewer columns. The two types of vertical partitioning are normalization and row splitting:

- Normalization involves removal of redundant columns from a table and putting them in secondary tables that are linked with primary key and foreign key relationships.
- Row splitting divides the original table vertically into tables with fewer columns. Each logical row in each of partitioned table are identified by a UNIQUE KEY column that is identical in all of the partitioned tables.

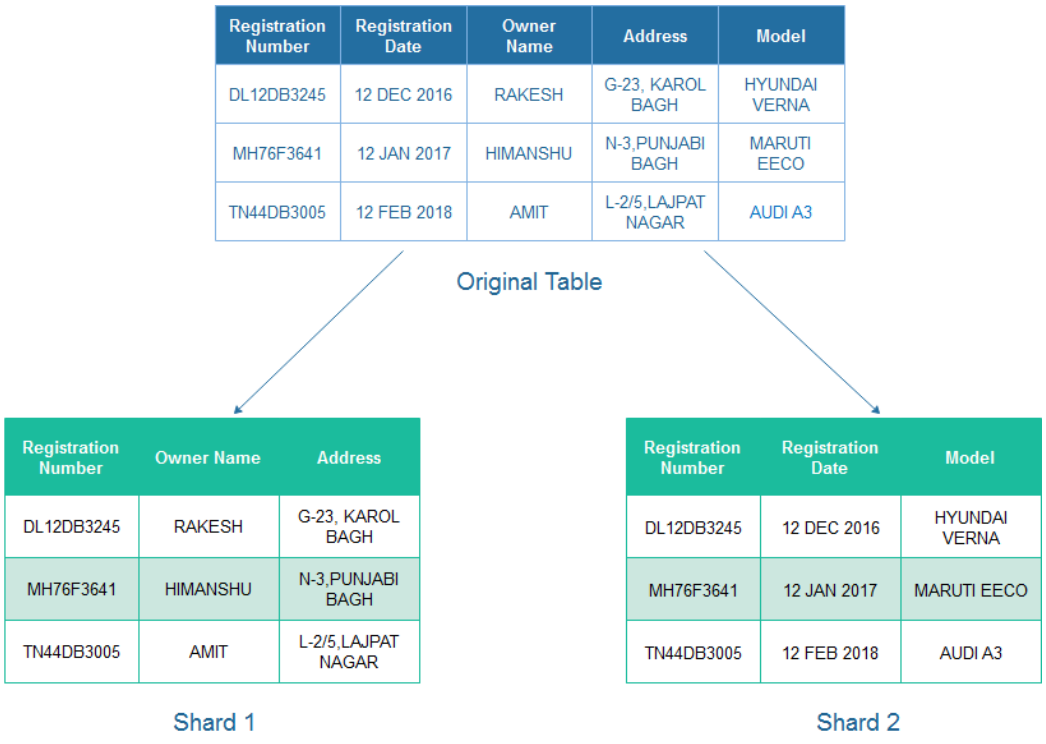

Figure 30: Vertical Partitioning

Vertical partitioning makes queries to scan less data. This increases query performance. For example, in the table above the columns referenced more frequently are split into a separate table.

Vertical partitioning should be considered carefully, because analyzing data from multiple partitions requires queries that join the tables. Vertical partitioning also could affect performance if partitions are very large.

Horizontal Partitioning

Horizontal Partitioning is splitting our data into smaller chunks which are spread across distinct separate buckets which can be a table, a schema, or a different database. Whenever we need to scale, we are able to move our shards to new nodes thus improving performance.

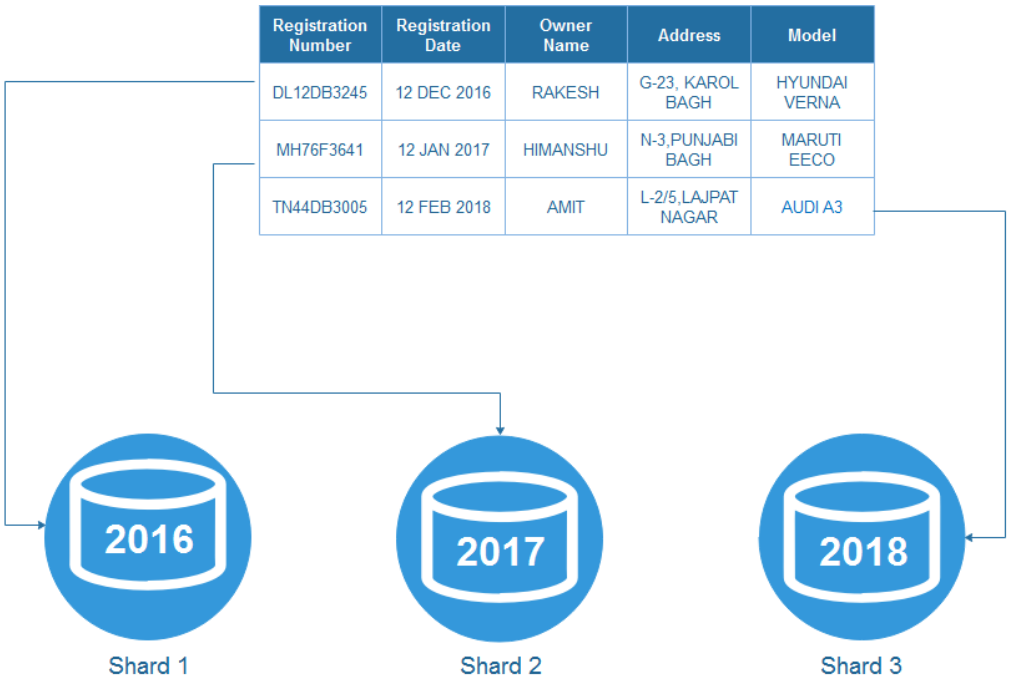

Figure 31: Horizontal Partitioning

Here the Table is split on row. Each table then contains the same number of columns, but fewer rows. For example, a table that contains one lakh rows could be partitioned horizontally into 4 tables, with each smaller table representing one quarter of data for a specific year. Any queries requiring data for a specific quarter only reference the appropriate table.

Determining how to partition the tables horizontally depends on how we want to analyse our data. We should partition the tables so that queries reference as few tables as possible. Otherwise, excessive UNION queries, used to merge the tables logically at query time, can affect performance.

Sharding is often used with a shared-nothing approach to automate partitioning and management.

Advantages of Sharding

Following are some of the advantages of sharding:

- The most important advantage of sharding is that when applied correctly, it allows us to scale the database servers horizontally to almost any size.

- By using application-level sharding, none of the servers needs to have all of the data. This allows us to have multiple database servers, each with a reasonable amount of RAM, hard drives, and central processing units (CPUs) and each of them being responsible for a small subset of the overall data, queries, and read/write throughput. By having multiple servers, we can scale the overall capacity by adding more servers rather than by making each of our servers stronger.

- Since sharding splits data into disjointed subsets, we end up with a shared-nothing architecture. There is no overhead of communication between servers, and there is no need for cluster-wide synchronization or blocking.

7. Key Principles of Software Architecture

In chapter 3, we discussed some issues / challenges of scalability at software level like concurrent connection, improper use of hardware, piling of service calls leading to memory exhaustion etc.

To address the scalability challenges faced at the software level, the focus must be kept on the web application architecture from the beginning of the architecture design process.

Following are some of the important guiding principles that can be incorporated to overcome scalability challenges at the software level.

7.1 Key Design Principles

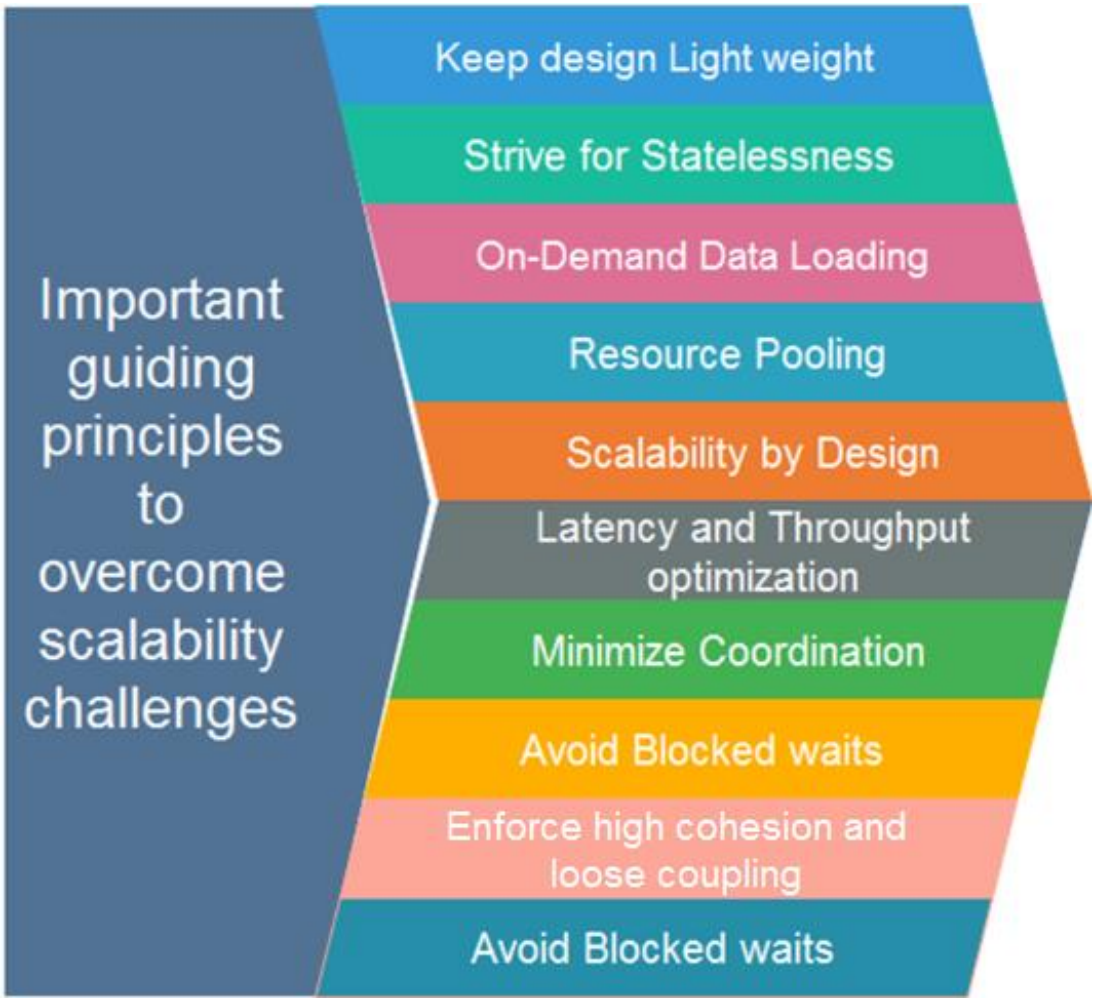

Figure 32: Key Design Principles

7.1.1 Keep design Light weight

For a web application, the performance of the component is judged by its size (in bytes) and the load time (in seconds). Hence, it is important to keep the component lightweight. The key best practices followed in this regard are as follows:

- Minimizing the number of static assets (such as images, JavaScript, Cascading Style Sheets (CSS)) required by the component. We can achieve this by compressing and then merging them to form a minimal set. The compression will help in reducing the size while merging component will reduce the number of HTTP requests required.
- We should use AJAX-based asynchronous calls for server invocations to have partial page refresh.
- For data transfer across requests, JSON is a lightweight alternative as compared to XML.
- Adoption of REST-based integrations rather than other heavy weight alternatives such as Simple Object Access Protocol (SOAP)

Scalability is achieved as:

- A lightweight web page or a presentation component requires minimal data transfer over the network and takes less time to load. The faster and lighter the component is, the easier it is for it to handle more requests and hence enhance its scalability.
- The design puts minimal load on server resources such as CPU, memory, and network.

7.1.2 Use appropriate Enterprise Integration methodologies

We should use appropriate enterprise integration methodologies to achieve scalability:

- *Service Oriented Architecture* : We should use REST based services for flexibility and scalability
- *Asynchronous integration*: The asynchronous communication should be preferred over synchronous API calls to achieve better performance.
- *Lightweight and on-demand data transfer:* We should use lightweight alternatives like JSON in place of XML and service invocations should be done only when required.

Scalability is achieved as SOA and asynchronous integration reduces the load on the CPU and other resources of the source system.

7.1.3 Strive for Statelessness

The request details are not persisted by a stateless session once the request is serviced. The stateless nature of transactions and requests makes the application more scalable.

Following are some of the techniques which can be used to make the application stateless:

- The transactions and web pages must be in a stateless state. The web page can be made stateless by leveraging cookies and storing encrypted state information within them.
- We can use techniques such as URL parameters and single-web-request transactions to eliminate or minimize sessions.
- We should use REST architecture, which supports stateless service invocation. The REST-based services can be employed for integration with other systems.
- Reducing session stickiness helps load balancers to distribute the load more efficiently and improves seamless horizontal scaling.
- We should minimize the state information stored in the session, if session creation is inevitable. This helps the application server in optimal session state replication across cluster nodes.

Scalability is achieved as:

- If the application is stateless, load balancers can easily distribute the load without bothering with session stickiness.
- Stateless architecture enables us to design caching in a more efficient fashion. A cache can be used and shared among all web requests. Caching further increases scalability.
- The stateless nature also reduces the overhead of session state synchronization for application servers.

7.1.4 On-Demand Data Loading

This guiding principle dictates that the loading of the data should be done only when it is needed or only when it is requested.

Following are some of the practical instances of on-demand loading in a web based scenario:

- The data to the second page should be fetched only when it is required and requested by the user.
- Similarly, populate a drop-down with large data only when the user accesses that component. Once the data is retrieved, it is better to cache them so that subsequent access to the same data set becomes faster.
- Images and page content of the bottom section of the page can be loaded only when the user scrolls to that portion

Scalability is achieved as this technique reduces the amount of CPU processing and amount of data transferred over the Network for each request.

7.1.5 Resource Pooling

We can create a managed pool of resources using resource pooling which otherwise would be costly to establish and maintain in real time. A database connection pool, thread pool, and service pool provide great flexibility during peak load to achieve scalability. The pool infrastructure offers many features such as maximum/minimum pool connections, initial connections, max time-out, and idle time-out values.

Resource pools such as database connection pools also maintain multiple logical connections over fewer physical connections, and facilitates reuse of the connections, which brings in more scalable efficiency. Establishing and maintaining connection with external resources such as databases and service endpoints are costly operations as they consume lot of memory and CPU.

Scalability is achieved as:

- Using managed resource pools allows to optimally maintain the connections with minimal overhead on system resources. The decreased overhead on system resource such as CPU and memory increases capability.
- Higher number of resource requests can be handled efficiently by managing their pools.

7.1.6 Scalability by Design

The Static analysis should be used to identify potential scalability issues in the code. We can use tools to analyze the heap size, CPU cycles and transaction time taken by the program for proactive code analysis. We can determine the actual scalability factor of our code can carrying out load and stress testing.

- **Static analysis using code metrics:** During static analysis, the following key metrics can be captured using the analyzer tool:

    - Number of Calls: The number of calls made to a method in the given use case.
    - Cumulative time: Time spent at each method with the transaction flow.
    - Method Time: Total time spent in execution of that particular method in the use case.
    - Average Method Time: It's the ratio of total time spent at that method to total calls made to that method.

We can perform the test repeatedly under simulated load conditions to observe the trend and understand the behavior of our software.

The following are some of the common insights we can draw from these metrics:

- If the average method time stays the same across various load scenarios, then that method can be said to be scalable.
- If the average method time increases with an increase in user load, then it points to inherent scalability issues with that method.

It is very important that we identify and analysis the scalability issues at an early stage of our development as this will help to address the issues at source and minimize the effort and cost of changes later.

7.1.7 Latency and Throughput optimization

Latency is the time taken for the first response from the server, and throughput is the total number of transactions the system can process in a given time interval. Low latency with high throughput is a desirable quality of a scalable system.

The System latency is a good indicator of the scalability of our system. A highly scalable system will not have any change in latency even when there is increase in the load.

The following are some of the best practices that help to achieve low latency :

- Minimizing the amount of data transferred over Network using lightweight formats like JSON for data exchange on the web.
- Implement On-demand data loading in place of preloading of entire data.
- Asynchronous data loading can be used to reduce the initial page load time.
- The Response HTML should be compressed to minimize the data transfer. .
- Minimal I/O operations using caching are helpful to reduce the latency for an operation.

Hence, scalability is achieved by maintaining highly scalable architecture with minimum latency and maximum throughput.

7.1.8 Minimize Coordination

Coordination between various application services should be minimized to achieve scalability.

Most cloud applications consist of multiple application services — web front ends, databases, business processes, reporting and analysis, and so on. As we discussed earlier, we can run these services on multiple instances to achieve scalability and reliability,

The coordination in these services limits the benefits of horizontal scale and creates bottlenecks. As we scale out the application and add more instances, we will see increased lock contention. In the worst case, the front-end instances will spend most of their time waiting on locks.

We should consider patterns such as Command and Query Responsibility Segregation (CQRS) and Event Sourcing. These two patterns can help to reduce contention between read workloads and write workloads.

The CQRS pattern separates read operations from write operations. In some implementations, the read data is physically separated from the write data.

In the Event Sourcing pattern, state changes are recorded as a series of events to an append-only data store. Appending an event to the stream is an atomic operation, requiring minimal locking.

Use asynchronous parallel processing: For operation that requires multiple steps that are performed asynchronously (such as remote service calls), we can call them in parallel, and then aggregate the results.

Try to use optimistic concurrency whenever possible: Pessimistic concurrency control uses database locks to prevent conflicts. This can cause poor performance and reduce availability. With optimistic concurrency control, each transaction modifies a copy or snapshot of the data. When the transaction is committed, the database engine validates the transaction and rejects any transactions that would affect database consistency.

Partition data: We should avoid putting all of our data into one data schema that is shared across many application services. A micro services architecture enforces this principle by making each service responsible for its own database. Within a single database, partitioning the data into shards can improve concurrency, because a writing to one shard does not affect other shards.

7.1.9 Enforce high cohesion and loose coupling

A service is cohesive if it provides functionality that logically belongs together. Services are loosely coupled if you can change one service without changing the other. High cohesion generally means that changes in one function will require changes in other related functions.

All successful applications change over time, whether to fix bugs, add new features, bring in new technologies, or make existing systems more scalable and resilient. If all the parts of an application are tightly coupled, it becomes very hard to introduce changes into the

system. A change in one part of the application may break another part, or cause changes to ripple through the entire code.

Micro-services are becoming a popular way to achieve an evolutionary design. A service is cohesive if it provides functionality that logically belongs together. Services are loosely coupled if we can change one service without changing the other. High cohesion means that changes in one function will require changes in other related functions.

7.1.10 Avoid Blocked Waits

During event handling and for service/resource calls, blocked waiting for response or acknowledgment should be avoided as it blocks the entire processing.

Two of common strategies to achieve this are as follows:

- By using asynchronous invocation wherein the request is sent and further processing is continued. Once the response is ready, the caller is notified via callback methods.
- By using messaging queue wherein the caller sends a message to the message queue and continues further processing. The caller, being a subscriber to the queue, is notified upon message processing.

Scalability is achieved, as avoiding synchronous waits would reduce the overall processing time, and it and also reduces the load on resources.

8. Caching for Web Applications

In caching reusable responses are stored in order to make subsequent requests faster. Caching works by caching the HTTP responses for requests according to certain rules. Subsequent requests for cached content can then be fulfilled from a cache closer to the user instead of sending the request all the way back to the web server.

Instead of trying to add more servers or make them respond faster to clients' requests, caching is used in order to avoid having to serve these requests directly from the primary server. This makes caching one of the most important technique, which can be used at layers from Browser to Database to achieve scalability and performance in our application.

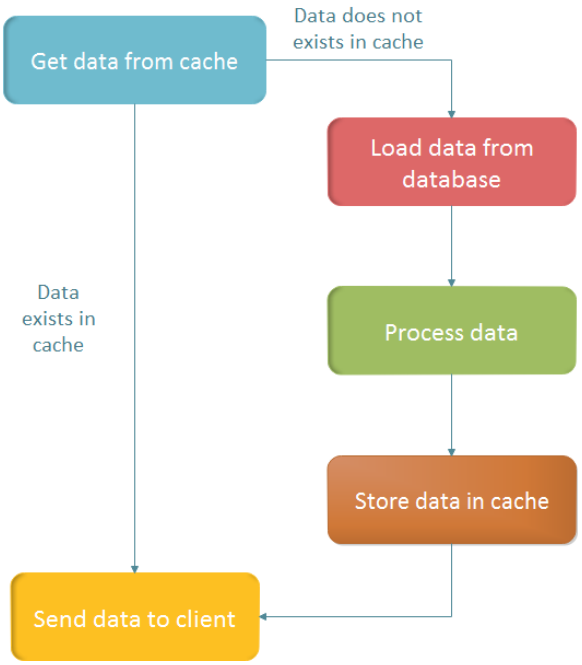

Figure 33: Request processing with Cache

In this chapter, we will discuss about how we can leverage Caching to achieve scalability.

8.1 Main scenarios for the use of caching

Most of the infrastructure systems such as the web server, application server, and database server have various built-in features for caching.

The main scenarios where caching can be used are given below:

Costly Resource Consumption: Costly operations in terms of resources used and memory/CPU/bandwidth consumed can be assessed to use caching.

In an n-tier architecture, there are various systems integrated with the application layer such as the database, services, reporting systems, legacy systems, and so forth. Call invocations for these systems normally tends to be costly. Some scenarios where caching can be used are

- Data fetched from the database through database reads
- Reports retrieved from reporting system.

Resource Bottlenecks: Some software and hardware components can become bottlenecks during heavy load. In such scenarios, data from those systems can be cached to reduce the load and calls for every request.

Costly service calls: Service calls should be analyzed for caching possibilities and tolerance for data staleness. Some sample cache candidates that fall into this category include:

- Internal and external web service call invocations
- Data services available within the three-tier architecture
- Service invocations to system of record.

Frequently executed repeated operations, calls and frequently used data: All frequently used operations and application data should be analyzed for caching opportunities. Some sample cache candidates that fall into this category include:

- Application look-up data such as configuration data and application start-up parameters
- Application-specific master list of values such as country, state, city, village, language and other domain specific masters
- Database lookup values
- Frequently used data results from complex calculations.

Static data and least frequently updated data: Any data or content that is not dependent on the user context or other dynamic parameters should be cached for optimal performance. Content present on a FAQ page, Contact Us page, and database values for country list are usually static.

Costly calculations and file parsing: In some instances, an end-result value or data computation would not vary for a given user session, then it makes sense to pre-compute the values and cache it to save the time required to do the calculation in real time. Similarly, file parsing is another kind of computation overhead, especially in the case of large files. Hence, cache alternative for this would be to store the parsed file into a more quickly accessible data structure in the memory.

8.2 Impact on scalability, availability, and performance

This section describes the impact of caching on performance, scalability, and availability.

Performance optimization through caching:

It is possible to optimize the performance of the web page, business process, and transaction with a well-designed cache.

Following are the main ways in which the cache plays a principal role in improving the performance of any enterprise application:

- Caching eliminates the time consumed in making costly resource calls such as database calls, web service invocations, remote API calls, content reads from CMS.
- A cache of pre-computed results and data avoids the complex calculations involved in real time.
- By caching a file content, the enterprise application can avoid the file reads over a network, expensive file parsing operations, and object creation in real time.

Scalability advantage from caching:

Caching can address the following scalability challenges:

- Scalability issues caused due to resource bottlenecks and choking points can be minimized by using caching.
- Non-scalable integrated systems such as enterprise interfaces or service endpoints pose challenges to the overall scalability of the system. Caching minimizes this impact by serving the requested data directly from cache and helps the system and application to handle heavy load during peak traffic.
- Caching at various levels such as web server caching, server side caching, database level caching, and service caching would address the scalability issues in those layers. It minimizes the load on hardware and software resources and hence reduces latency related issues.

High availability through caching

In a typical n-tier architecture, the availability of the overall system and application is dependent on all participating systems in the delivery chain. A distributed caching configuration enables high availability of the application by supporting features such as session switchover and failover scenarios.

8.3 Cache concepts

Caching can be implemented at various layers for different types of data objects.

Following are the broad categories of caches:

- **Object caching**: In this category, frequently used database lookup, data requiring costly resource and service calls, controlled list values, user profile attributes, search results, page fragments, user roles, and permissions are ideal candidates for object caching. This caching strategy aims at reducing costly server or resource round trips and costly computations. Object caching can be implemented using custom caching components or built-in or open-source caching frameworks.

- **Application cache**: This is achieved through cache proxies and reverse cache proxies to improve performance and reduce the load on servers. Proxy caches act as forward proxies that cache the network lookups and content, speeding up response time and minimizing resource utilization. Web server uses reverse proxy cache for optimizing DNS and network lookups.

- **Configuration-based caching**: The application server provides cluster-wide caching and cache replication parameters that can be configured while using built-in caching APIs.

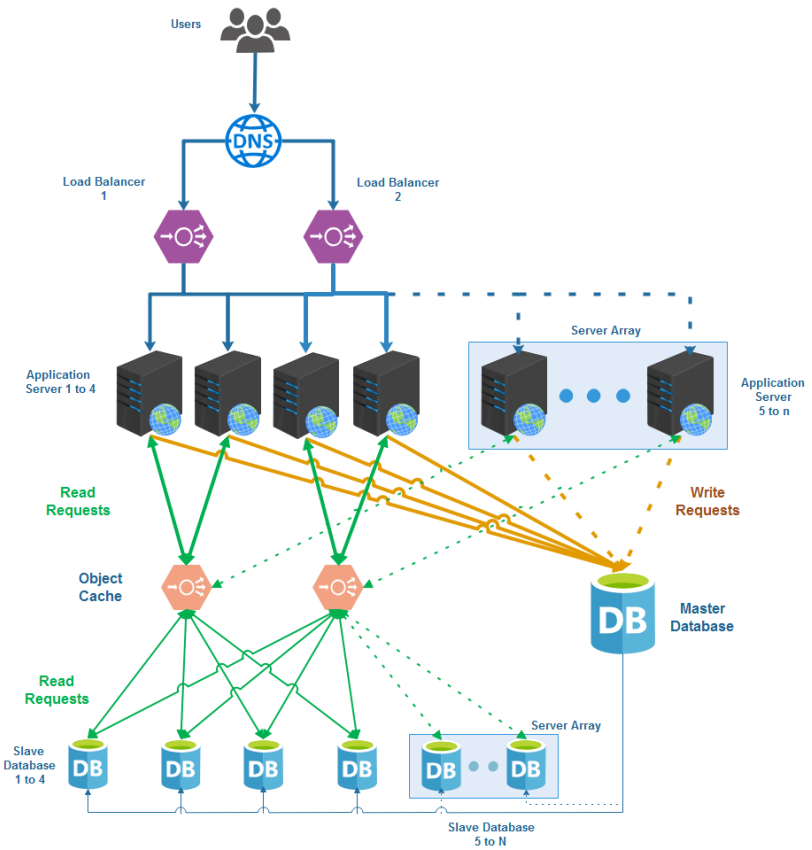

Figure 34: Database Object Cache

User session-based cache categories:

Another dimension of caching is the applicability of cache values to the user session. Using these criteria, there are two broad categories of caching:

- **Static cache:** A static cache stores user session independent values such as a list of countries, languages, and so forth which can be used by all user sessions. These static values can be cached in a global cache area and hence can be shared across multiple applications and user sessions.

- **Session-based cache:** It involves caching user-specific values such as user preferences, user details, and personal preferences accessed across a user session multiple times. The scope is mostly restricted to the user session in this scenario.

8.4 Cache design

Designing an optimal caching strategy involves the following steps:

1. ***Choosing caching scenarios at all layers:*** All the scenarios that make expensive resource calls, database queries, frequently used values, service invocations, and scenarios that involve costly computation can be explored for caching opportunities.

2. ***Choosing an appropriate cache eviction algorithm and other cache configuration parameters:*** Appropriate cache eviction algorithm can be selected based on content refresh rate, refresh timings, and potential usage pattern of cached objects. The key design goal for caching is to maximize the cache hit ratio.

8.5 Caching strategy

An effective caching strategy is Layer-wise caching strategy. An enterprise application built using n-tier architecture typically involves multiple hardware and software systems in the request processing pipeline. In such a scenario, building a caching layer for each tier is the most effective way to optimize the performance. The following diagram represents where cache which can be applied at each layer.

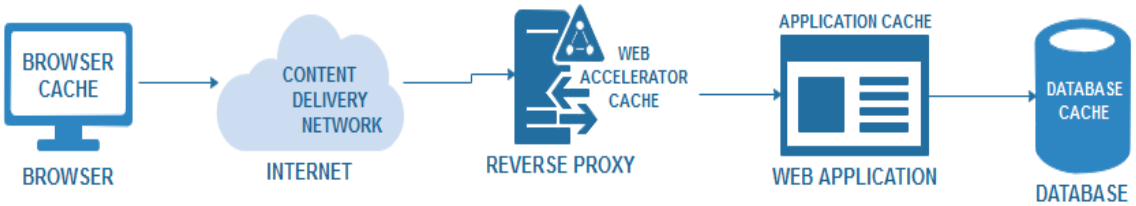

Figure 35: Caching Layer

Layer-wise caching helps making the systems scalable by getting the "data of interest" from the nearest possible location for that layer, thereby reducing further dependencies on upstream systems and the network.

The following section discusses about the caching strategies which can be used at various layers.

8.5.1 Caching at the end user

Browser caching: It can be leveraged to cache assets and other content. HTTP cache headers "expires," "cache-control," and "last- modified" can be set to control the cache timings for various static assets such as images, binary files, JavaScript files, style sheets etc.

Edge-side caching using content delivery networks (CDNs): Widely distributed CDN servers across diverse geographies use intelligent routing algorithms to serve the static global assets from the optimal location so as to minimize load times and minimize requests to the origin servers.

8.5.2 Presentation layer

Web server caching: Most of the web servers provide configurable values for caching static resources.

Caching proxies: Reverse proxies that are in front of the web servers can cache static assets on their own and reduce the load on the origin web server. They can also cache static and dynamic content to speed up a response. HAProxy can be used as reverse proxy for caching the content

8.5.3 Application layer

Object caching: Various application objects that are fetched from remote layers and data sources can be stored in this cache. Cached objects include search results, query results, page fragments, lookup values, and such.

Configuration caching: Many application servers provide features to cache key configuration values such as pool size, cluster settings, and cache configuration values.

8.5.4 Database layer

Database cache: Many database servers provide configurations to cache the query results.

Snapshot tables: These tables contain the view of tables from remote database instances

or as result of complex joins.

Lookup table: This table contain the pre-computed results from many other tables and database objects. A stored procedure can perform complex computations on a regular basis and update the lookup tables with the end result. The application can query the lookup table in real time to minimize the computation overhead. The lookup table can also be used to store the static list of values such as a country list, language list, product list, and other lists.

8.6 Cache metrics and administration

Key caching metrics are listed below that should be monitored to check the optimal usage of the cache and validate the cache configuration settings. Built-in or custom cache monitor interfaces often provide the real-time monitoring feature to check these metric values:

Cache hit ratio*:* This indicates the total number of requests satisfied through cache hits.

Cache hit ratio = (Total number of cache hits)/(Total number of requests).

A higher value validates the cache strategy adopted.

Cache miss ratio: This indicates the total number of requests that missed the cache and hence went to the origin server.

Cache miss ratio = (1 - cache hit ratio)

If there is a significant number of cache misses, it is time to recheck the cached objects and cache eviction algorithms.

Cache size: This is the total memory consumed by the cache.

8.7 Cache Products

Some of the popularly used open source caching products are listed below :

Memcached*:* is a distributed memory object caching system that alleviates database load to speed up dynamic Web applications. The system caches data and objects in memory to minimize the frequency with which an external database or API (application program interface) must be accessed.

Redis*:* is another fast in-memory key-value data store for use as a database, cache, message broker, and queue. Redis is a popular choice for caching, session management, real-time analytics etc.

Ehcache: is another cache solution that boosts performance, offloads our database, and simplifies scalability. Ehcache scales from in-process caching, all the way to mixed in-process/out-of-process deployments with terabyte-sized caches.

Varnish : Varnish Cache is a web application accelerator also known as a caching HTTP reverse proxy. This can be installed in front of web server and configure it to cache the contents. Its configuration language VCL enables writing policies on how incoming requests should be handled. In such a policy, we can decide what content we want to serve, from and how the request or response should be altered.

9. Resource requirement Analysis for Capacity Planning

As discussed earlier, Scalability cannot be after thought. Proper planning is required to achieve the desired performance. It is important to identify the servers capacity needed to meet the requirements related to performance and availability. Capacity planning is the process of determining the production capacity needed to meet changing demands for our web application.

Without proper capacity planning for our deployment infrastructure, we may end-up either grossly under-prepared or over-prepared. A discrepancy between the capacities we have planned for and the actual demand can result in failures and disruption or under-utilized resources.

9.1 Main steps in Capacity Planning

The main steps in capacity planning are:

1) **Demand analysis**
   In this step, we gather all information about the current demand, workload, trends, and essentially all aspects of usages of infrastructure elements such as CPU, memory, and network. Collecting the infrastructure usage statistics using tools, and preparing a questionnaire for each of the involved infrastructure elements, and compiling the responses from all stakeholders can do this. Given below is a sample questionnaire about two infrastructure components.

   Capacity Planning Questionnaire

   Some of the parameters, which can be considered are

   - What is the current traffic between the web server, application server and database server?
   - Which application is using the bulk of the bandwidth?
   - Are there any seasonal or behavior trends related to peak traffic?
   - What is the traffic between the application server and upstream and downstream systems?
   - What is the average response time?
   - What is the peak transaction per hour?
   - What is the maximum number of database users?
   - What is the size of maximum concurrent users?

2) Current capacity analysis

Once we get all aspects of current and future demands of the workload on systems, we can analyze and determine if the existing capacity of the systems can meet those demands. We will establish the threshold and benchmark values for identifying if the resource is over utilized or underutilized. Following table contains a sample threshold value for resources on Linux server.

We must look at the current resource utilization for an extended duration and identify the following values:

- Resources that are underutilized
- Resources that are heavily utilized
- A comparison of average utilization against the threshold and benchmark values
- An end-to-end analysis to identify bottleneck.

The inputs from the above values can be used to optimize the capacity planning.

3) Future Capacity Planning

Using historical data analysis, trend analysis, and anticipated workload, we can use prediction models to arrive at the required infrastructure capacity. In addition to a prediction model, we should also consider the vendor recommendations for capacity. Most of the software and hardware vendors provide recommended configuration and specifications for optimal usage of their products. This information should be used as main input in coming up with final capacity numbers. Hence, a combination of predicted capacity and vendor recommendations can be used to arrive at final capacity numbers.

It is always recommended to take the capacity planning as a two-step process: perform the initial assessment based on obtained/anticipated resource usage and then test the new capacity using simulation models during the integration testing phase to ensure that the planned capacity adequately meets the anticipated demands.

During this stage, the capacity numbers may need to be fine-tuned based on system performance.

9.2 Capacity Planning Parameters

There are multiple methodologies of carrying out capacity planning. We should understand the impact of factors like concurrency, transactions per second (TPS), work done per transaction, etc. Other factors, which can determine the capacity of a system, are complexity of transactions, latency and external service calls, memory allocation, memory utilization etc.

It is advisable to test the performance of the system to see whether it will perform to the expected capacity once the system is set up as per the capacity planning requirements.

9.2.1 Throughput and TPS

Throughput is defined as the number of messages processed over a given interval of time. Throughput is a measure of the number of actions per unit time, where time can be in seconds, minutes, hours, etc.

9.2.2 Work done per transaction

For each incoming 'transaction' request, server performs set of operations. This would mean a number of CPU instructions would be triggered to process the said message. These instructions might include application processing as well as system operations like database access, external system access, etc. If the transaction is a simple 'pass through' that would mean relatively lesser processing requirements than a transaction that triggers a set of further operations.

9.2.3 Think time

In a web application, users submit requests, which are then processed at the server side before returned to a user. The user then often waits on the response, and 'processes' it, before submitting again. This delay is the user think time that falls between requests, and can be taken into account when calculating optimum system load. We should remember that In case of machine to machine integration, the think time parameter value is relatively lower. For capacity planning, the average think time is useful in arriving at an accurate throughput number.

9.2.4 Active users and concurrency

A system would have a total number of users - this might not affect the server capacity directly, but is an important metric in database sizing for instance. Of these total set of users, a subset of them will be active users - users who use the system at a given time.

Concurrent active users are the number of distinct users concurrently accessing the system at any given time. In an application server with a stateful application that handles sessions, the number of concurrent users paly a large role than in the case of ESB, which handles stateless access.

Conventionally, system throughput increases with the number of concurrent users until it reaches peak capacity before it starts degrading. Thus, it is important to calculate the maximum concurrency a system can handle.

9.2.5 Message size

The size of the message passed across the network is also an important factor in calculation of system capacity. For larger messages more processing power and memory is required.

9.2.6 Latency

As we know, “latency” is the amount of time it takes for the web server to receive and process a request. To meet desired response time, the latency needs to be considered at all levels including network/overhead bandwidth.

9.3 Number of Hits

Accurate sizing requires a reasonable estimate of the average number of hits per day. If the system is already in place and we have web traffic information available. The day is broken into four-hour window and traffic is measured. The highest number of hits in the four hour window (peak period) is multiplied by 6 to plan for sizing our infrastructure.

If we are provisioning our infrastructure from scratch, we need to consider and analyse the architecture of the our application, including number of pages, objects per page, and the target users etc to calculate estimated number of hits.

9.4 Determining the processor size

We need to ensure that the CPU utilization of less than 80 percent is maintained. This is crucial to maintaining efficient response times as higher CPU utilization can result in longer queue which will have impact on response time. In the case of Server-generated dynamic content it is difficult to size the processor. Benchmarking data for the dynamic content generation processing is essential. This data is used to estimate the number of CPUs and CPU speed necessary to serve the dynamic content. Calculating the average computational effort per hit requires an understanding of the proportion of hits that will cause dynamic content to be served.

9.5 Sizing the memory

Knowing the resources to be used can help to determine the amount of memory required. For Web servers, the byte size of all the software resources running on the processor must be considered, beginning with the operating system. This estimation can then serve as a guide to identify the amount of unused portion of memory. Ten percent of memory needs to be free; this is also known as memory free space.

The availability of sufficient memory prevents the server from accessing the disk frequently as the frequently accessed information can be retrieved directly from memory rather than

accessing the disk. It also enhances the end-user experience while resulting in less work on the server.

The following consumers must be considered in order to calculate the memory requirements for RAM:

OS and Web server memory usage. The number of concurrent connections directly affects memory. Memory usage for the applicable operating system and Web server can be determined through the documentation.

Generating dynamic content. The amount of memory required to generate dynamic pages along with the number of concurrent connections must be considered to estimate how much memory is required.

Calculating dynamic and static content ratios. Since some content that appears dynamic is actually served as static content make sure that the overestimation is not done for the percentage of dynamic material.

Operating system and Web server caching. Two key types of caching are key to Web server performance: operating system file system caching and Web server caching. It is necessary to allocate and account for memory for these processes.

Once the memory requirement is determined, it should be considered as an absolute lower limit beyond which the server can fail.

9.6 Performance Metrics

An increase in Database queries per second and web server requests per second effect the below parameters:

- Disk utilization
- I/O Wait
- RAM usage
- CPU usage

Understanding how each of the above parameters are affected can help in deciding how much and when capacity needs to be added across database servers or application servers.

Most operating systems come with some basic built-in utilities that can measure and record various performance and consumption metrics.

For capacity planning, measurement tools should provide, at minimum, an easy way to:

- Record and store data over time
- Build custom metrics
- Compare metrics from various sources
- Import and export metrics

Metric collection system architecture consists of an *agent* that runs on each of the physical machines being monitored, and a single *server* that aggregates and displays the metrics. As the number of nodes in the infrastructure grows, more than a single server may be required for performing aggregation, especially in the case of multiple data centre operations. The agent's job is to periodically collect data from the machine on which it is running and send a summary to the metric aggregation server. The metric aggregation server stores the metrics for each of the machines.

Capacity planning is not complete without the measurement and history of the system and application-level metrics.

Finding the ceilings of each part of the architecture involves the same process:

- Measure and record the server's primary functions, for e.g. hits, database queries etc.

- Measure and record the server's fundamental hardware resources, i.e. CPU, memory, disk, network usage

- Determine how the server's primary functions relates to its hardware resources, i.e. "n" database queries result in "m" percent CPU usage.

- Find the maximum acceptable resource usage (or ceiling) based on both the server's primary function and hardware resources by one of the following:

    - Artificially (and carefully) increasing real production load on the server through manipulated load balancing or application techniques.

    - Simulating as close as possible a real-world production load.

9.7 Application Level Measurement

Server statistics give only a partial understanding of the capacity of the system hence; there is a need to also measure and record higher-level metrics specific to the application.

Some of the metrics that can be tracked are listed below:

- User registrations (daily, cumulative)
- No. of applications filed
- Documents uploaded (daily, cumulative)
- Documents uploaded per hour
- Average document size
- API traffic (API keys in use, requests made per second, per key)

9.8 Storage Capacity

Choice of storage media can be governed by the below two factors :

- The maximum capacity of the storage media
- The rate at which the data can be accessed

Sizing the disk drives and number of drives is as important as determining memory and processor speed. It is important to never exceed 85 percent usage of disk drive space. Base the selection on 85 percent of used space and 15 percent free space.

Optimizing disk performance. Sizing and performance are not the same. Sizing is based on a conservative percentage of performance capabilities to allow for peaks and spikes in usage. Choose the size of the disk based on the site size, considering the 80/20 ratio i.e. using 80 percent of disk capacity and maintaining 20 percent of disk space for other network requirements.

Consumption rates

When planning the storage needs for an application, the first and foremost consideration should be the consumption rate. This is the growth in data volume measured against a specific length of time. For sites that consume, process, and store rich media files, such as images, video, and audio, keeping an eye on storage consumption rates can be critical to the business.

9.9 Storage I/O Patterns

Accessing the storage an important consideration. Disk utilization metrics can vary, depending on the storage architecture being measured. They depend on the below factors:

- Amount of data read from the disk.
- Amount of data written to the disk.
- CPU waiting-time for reading/writing to finish.

Disk utilization and throughput can be measured by monitoring disk consumption and disk I/O consumption irrespective of the hardware solution used.

9.10 Measuring Load on Web Servers

Web server capacity is application-specific. Understanding of both system and application-level metrics can help with a long view of the usage metrics, which can serve as the foundation for the capacity plan.
Capacity planning for web servers (static or dynamic) is peak-driven, and therefore elastic, unlike storage consumption. The servers consume a wide range of hardware resources over a daily period, and have a breaking point somewhere near the saturation of those resources. The goal is to discover the periodic peaks and use them to drive our capacity trajectory.

9.11 Database Capacity

Outside of the basic server statistics, below are a number of database-specific metrics that need to be tracked:

- Queries-per-second (SELECTs, INSERTs, UPDATEs, and DELETEs)
- Connections currently open
- Lag time between master and slave during replication
- Cache hit rates

Finding database ceilings
A more focused approach to finding database ceilings is to increase the load on a live production server. This exercise becomes easier if database load balancing (via hardware appliance or within the application layer) is employed on database servers. Increasing load to a database can reveal the effects load has on resources, and hence expose the point at which load will begin to affect replication lag.

9.12 Caching

Disks are the slowest pieces of infrastructure, which makes accessing them expensive in terms of time. Most large-scale sites alleviate the need for making these expensive operations by caching data in various locations. In web architectures, caches are often used to store database results or actual files. Both approaches call for the same considerations with respect to capacity planning. They are examples of reverse proxies, which are specialized systems that cache data sent from the web server to the client.

The two main factors that affect cache capacity are

- Size of working set
- The extent to which data is dynamic or changing. Dynamicity of data dictates the choice for caching.

9.13 Application Design and Optimization

In capacity planning, the design of the application or software plays a big role. If a session is created for each concurrent user, this means some level of memory consumption per session. The amount of memory and processing capacity required for each operation is determined by factors such as open database connections, the number of application 'objects' stored in memory, and the amount of processing that takes place. Well-designed applications share the resources effectively and keep these numbers low. Profiling and load testing an application with tools (e.g. for Java apps - JProfiler, JConsole, JMeter, etc.) would help determine the bottlenecks of an application.

10. Reference Architecture for Scalable Web Applications

In the earlier chapters, we have discussed about various components of a web application and strategies which we can apply at various layers like Web Layer, Application Layer and Database layer to make our application scalable.

In this chapter, we will use the strategies discussed, to build a generalized reference architecture for our web applications which can be implemented for projects. The architecture needs to be tweaked as per specific application type and its requirements. Further, the deployment on cloud provides ideal environment for scalable applications because it allows for rapid resource allocation in times of peak demand as well as de-allocation of resources if demand declines.

10.1 Reference Architecture

A basic web application can function using a single server which acts as Web Server, Application as well as Database server. We can install all the components in the single server and deploy our application.

Figure 36 : Single server deployment

In production environment, we do not use this deployment architecture as this becomes a single point of failure and the system is not scalable.

As an advancement to the basic architecture shown above, the application and the database server can be separated as the number of users and concurrent sessions increase & further hardware can be added to each i.e. vertical scaling can be done as required at application and database server level. This is also required to move database server in Militarized zone due to security considerations.

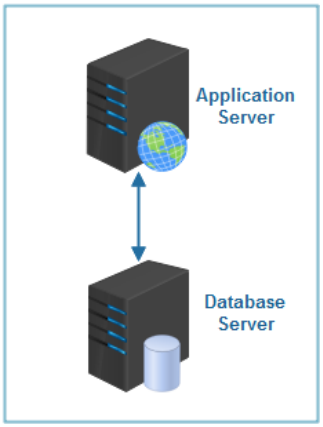

Figure 37 : Web and Database Server Deployment

With the increase of number of requests, vertical scaling cannot deal with the growing number of concurrent sessions and HTTP requests, introducing a load balancer and horizontal scaling at the application server level can help in processing more number of requests. A load balancer helps to distribute the traffic over multiple application servers to ensure faster processing of request and efficient utilization of the multiple application servers.

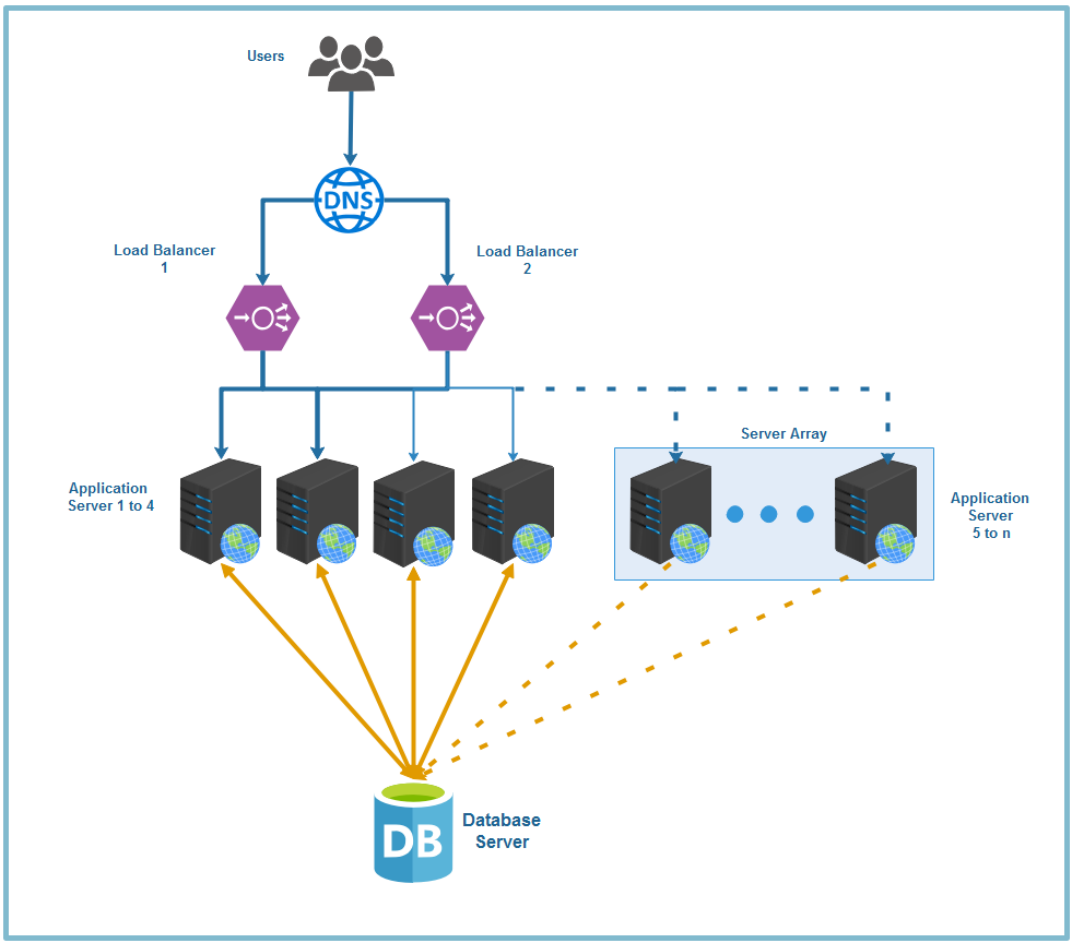

Figure 38: Load balancing and Application server scaling

The above shown approach has a disadvantage as it uses a single Database server which can act as a single point of failure and bottleneck if number of connections increases. In order to avoid this issue, scaling can be implemented at database server level through the master-slave approach in which all the critical writes and reads are done with a single database server and the data is replicated to the slaves which process the read only requests as shown below.

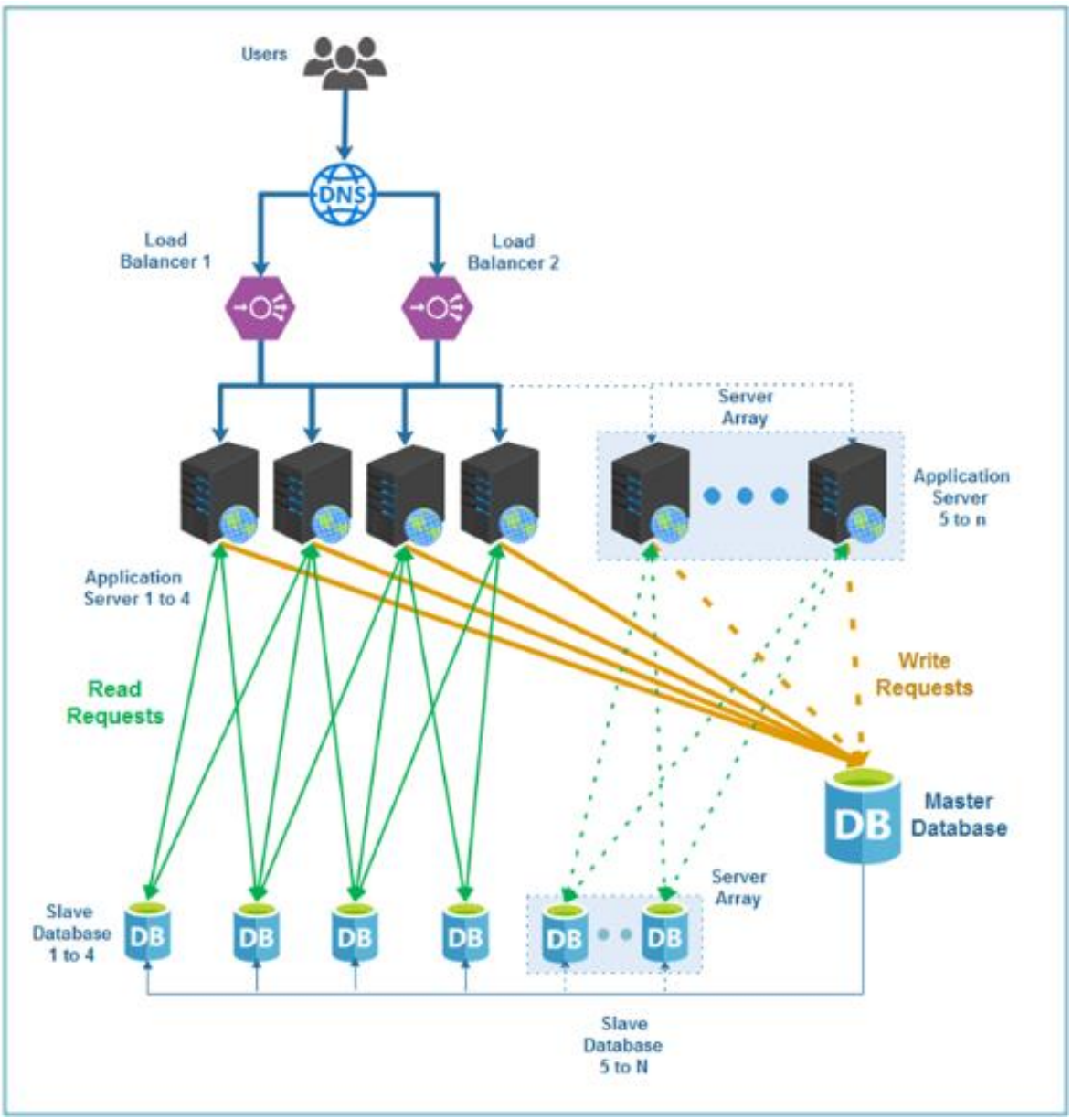

Figure 39: Master/Slave Database replication

In case the database servers are not able to keep up with the load and ever increasing data size, database sharding can be implemented as shown below:

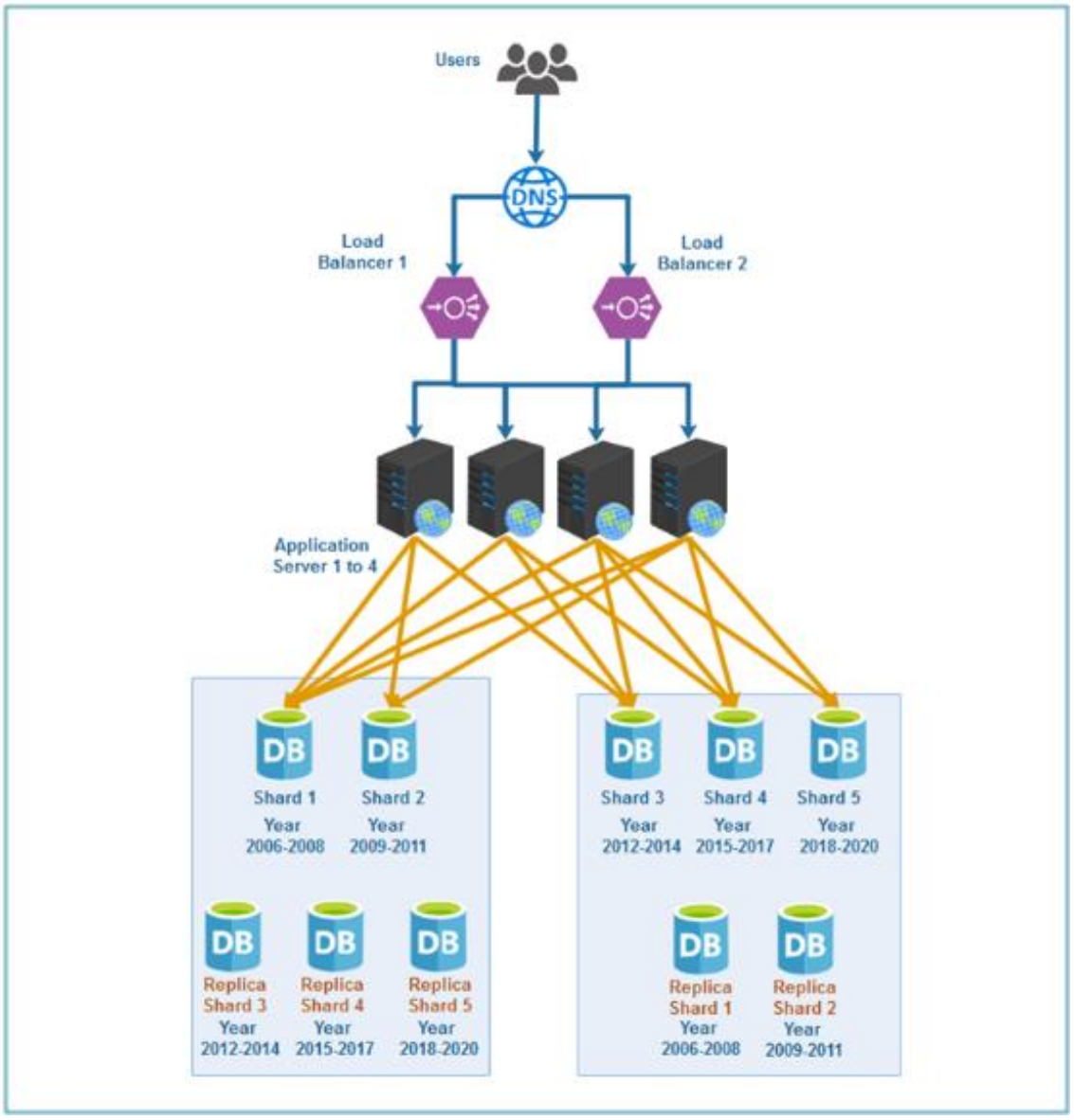

Figure 40: Database Sharding

Another approach to improve the performance of web applications is implementing caching at different levels.

The figure given below shows the addition of an object cache between the application server and database servers. We can use Cache server like Memcached or Redis for meeting the caching requirements at the application level.

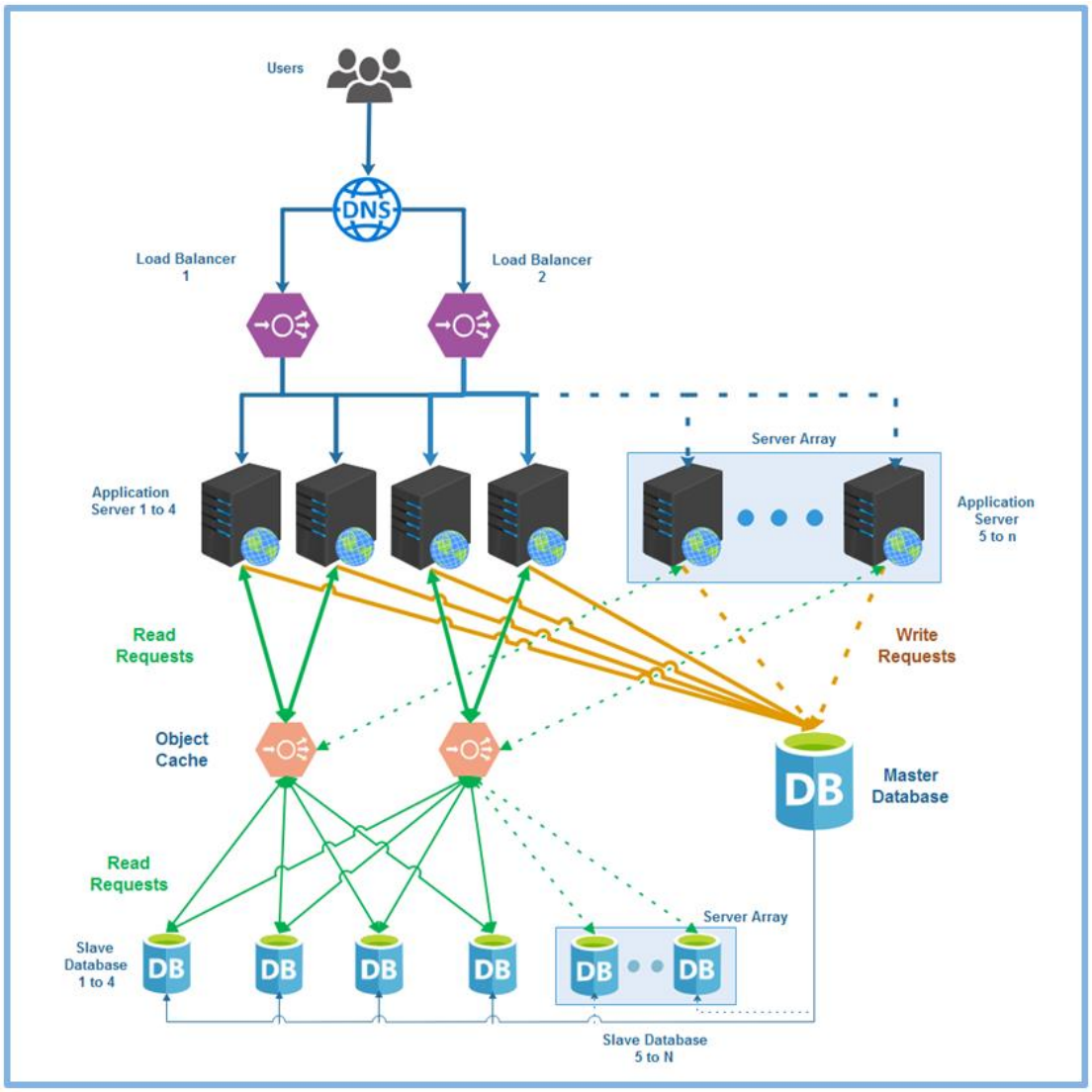

Figure 41: Caching with Object cache

The Load Balancer should also be redundant to avoid single point of failure. For most of our applications, the above architecture is sufficient to meet our requirements.

To further add scalability, we can divide our deployment on region with separate set implemented for each region. Each deployment can then act as a single set serving a collection of users. As the number of users increase further more sets can be added as required.

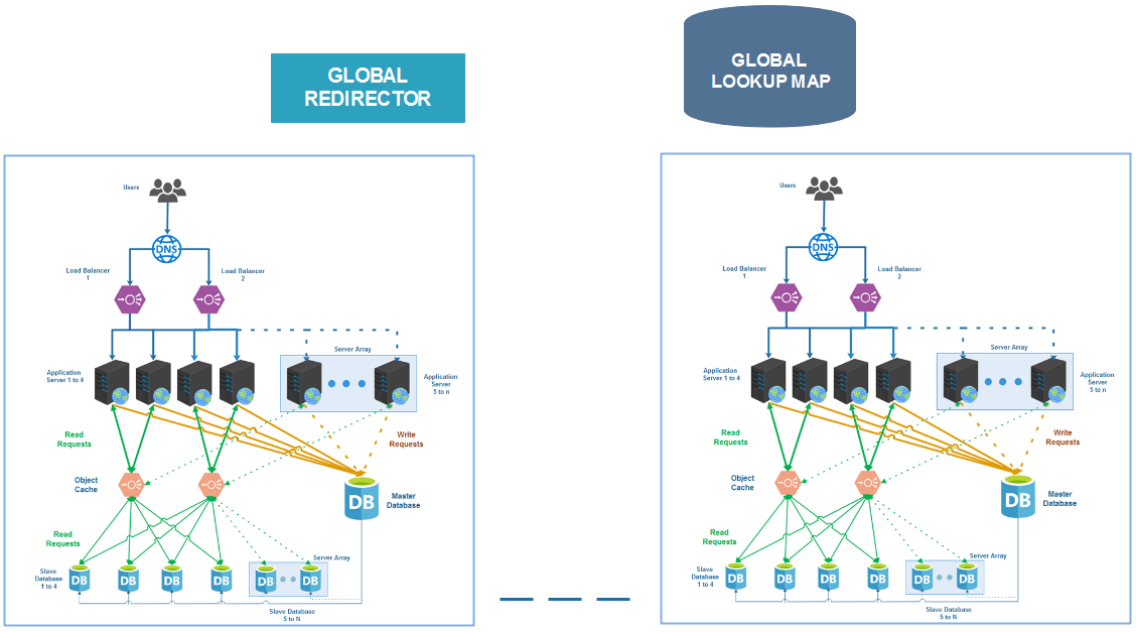

Figure 42: Multiple instances

11. Scalability Testing

Web applications must be able to handle large increases in concurrent users, data volume and other workloads hence they must be thoroughly tested for the same after development. Scalability tests provide a way to simulate different scenarios to ensure that our application is ready for any situation.

The basic goal of a scalability test is to determine at which point the application stops scaling, and then figure out how to fix it. It helps us learn the application's user limit by assessing client-side degradation and end user experience as well as server-side robustness and degradation under heavy loads.

11.1 Scalability testing parameters

During the scalability testing scenarios, the following parameters are required to be closely monitored:

- Response time and application performance for requests
- Throughput: work done per unit of time (transactions/s or bytes/s)
- CPU utilization during normal and peak loads
- Memory utilization: during normal and peak loads
- Load size: number of concurrent users.
- I/O operations
- Database pool size
- % Network utilization
- Resource pooling
- Session usage
- Thread pool size
- Locks

11.2 Sub-categories of Scalability testing

There are two sub categories of Scalability testing i.e. Performance Testing and Stress Testing

11.2.1 Performance Testing

Performance testing covers a broad range of engineering evaluations where the emphasis is on the final measurable performance characteristic. The goal of performance testing is to identify, document, and where possible eliminate bottlenecks in the system.

Load testing is a process used in performance testing, it involves putting load or user demand on a system in order to measure its response and stability. The purpose of load testing is to verify that the application can meet a desired performance objective. It must be considered that load and performance testing are not substitutes for proper architecture.

Load tests should determine the following:

- Maximum user load that the system can handle per time period and concurrently
- Average response time during average and peak load
- Resource utilization

When conducting performance testing, the following steps are the critical steps to completing it properly:

1. **Criteria**: Establish what criteria are expected from the application, component, device, or system that is being tested.
2. **Environment**: Make sure that the testing environment is as close to production as possible to ensure that your test results are accurate.
3. **Define tests**: There are many different categories of tests that one should consider for inclusion in the performance test. These include stress test, spike test, endurance test, Load testing and scalability testing.
4. **Execute tests**: This step is where the tests are actually being executed in the environment established in Step 2.
5. **Analyze data**. The data is analyzed to take necessary action like fixing bugs, creating indexes etc .
6. **Repeat tests & analysis**. As necessary to validate bug fixes or as time and resources permit, continue testing and analyzing the data.

11.2.2 Stress Testing

Stress testing is a process that is used to determine an application's stability when subjected to above normal loads. Stress testing, as opposed to load testing, goes well beyond the normal traffic, often to the breaking point of the application, in order to observe the behaviours.

Stress tests should determine the following:

- Maximum requests that can be handled by the application
- Breakdown points.

When performing stress testing, the following steps are the critical steps to completing it properly:

1. **Identify objectives**: Identify why we are performing the test. These goals usually fall into one of four categories: establish a baseline, identify behaviour during failure and recovery, identify behaviour during loss of resources, and determine how the failure of one service will affect the entire system.
2. **Identify key services**: Time and resources are limited so you must select only the most important services to test.
3. **Determine load**: Calculate or estimate the amount of load that will be required to stress the application to the breaking point.
4. **Environment**: The environment should mimic production as much as possible to ensure the validity of the tests.
5. **Identify monitors**: Plan ahead by using the objectives identified in Step 1 as criteria for what must be monitored.
6. **Create load:** Create the actual load data, preferably from user data.
7. **Execute tests**: This step is where the tests are actually being executed in the environment established previously.
8. **Analyze data:** The last step is to analyze the data.

11.3 Open Source Scalability Testing Tools

- **Apache JMeter**: It is a Java platform application. It is mainly considered as a performance testing tool and it can also be integrated with the test plan. In addition to the load test plan, a functional test plan can also be created. This tool has the capacity to be loaded into a server or network so as to check on its performance and analyze its working under different conditions.

- **OpenSTA:** Open STA stands for Open System Testing Architecture. This is a GUI-based performance tool used by application developers for load testing and analyzing. The current toolset is capable of performing the heavy load test and analysis for the scripted HTTP and HTTPS.

- **Locust:** It is a simple-to-use, distributed, user load testing tool. It is used to help load test web sites or other applications. Locust can also help us figure out how many concurrent users a system can handle.

- **nGrinder:** It was designed to be an enterprise-level performance testing solution by making stress testing easy and to provide a platform that allows us to create, execute and monitor tests.

- **Apache ab:** It is a tool for benchmarking Apache Hypertext Transfer Protocol (HTTP) server. It is designed to give an impression of how our current Apache installation performs and also shows us how many requests per second our Apache installation is capable of serving.

12. Key Takeaways for Building Scalable Web Applications

For building scalable web application here are a few key takeaways to keep in mind:

- **Over-engineering is one of the many enemies of scale**: Systems that are overly complex limit the scalability of the application whereas simple systems are more easily and cost effectively maintained and scaled.

- **Scalability cannot be after thought**: Designing and thinking about scale comes relatively cheaply and thus should happen frequently. Given this relatively low cost we can discuss and sketch out a design for how to scale our platform well in advance of the need.

- **Design for Horizontal Scalability or Scaling out**: Scaling up is failing up as eventually there will be a point that either the cost becomes uneconomical or there is no bigger hardware made so, instead of keeping up expectations to use bigger/expensive systems for scaling up, design for scaling out by using the AKF Scale cube to determine the correct split in terms of
    a) X-Axis: Duplication of services or databases to spread transaction load
    b) Y-Axis: Scaling data sets, transactions, and engineering teams.
    c) Z-Axis: Split by some unique aspect of the data such as ID, name, geography etc.

- **Leverage Caching**:
    a) For each object type (IMAGE, HTML, CSS etc.) consider how long the object can be cached for and implement the appropriate header for that timeframe.
    b) Leverage Ajax and cache Ajax calls as much as possible to increase user satisfaction and increase scalability.
    c) Decrease load on Web servers by using a reverse proxy caching server and delivering previously generated dynamic requests and quickly answering calls for static objects by using page caching.
    d) Implement object caches to help the system scale i.e. consider implementing an object cache anywhere computations are performed repeatedly, this is done primarily between the database and application tiers.
    e) Use CDNs (content delivery networks) to offload traffic

- **Strive for Statelessness**: The implementation of state limits scalability and increases cost. If we must keep great amounts of data associated with a user's interactions at any given time, then we can house fewer users on any given system at any given time. State and session cost us both in terms of memory and processing power, and as a result is an enemy to our cost effective scale goals.

- **Loosely coupled components**: Use asynchronous communication techniques to ensure that each service and tier is as independent as possible. This allows the system to scale much farther than if all components are closely coupled together.

- **Actively Use Log Files**: The application's log files must be regularly monitored using monitoring tools, export the errors and assign resources for identifying and solving the issue. Keep a process in place for this as log files are excellent sources of information about how the application is performing.

References

Abbott, Martin L., and Michael T. Fisher. Scalability Rules: Principles for Scaling Web Sites. Addison-Wesley, 2017.

Shivakumar, Shailesh Kumar. *Architecting High Performing, Scalable and Available Enterprise Web Applications*. Morgan Kaufmann Publishers, 2015.

Abbott, Martin L., and Michael T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley, 2015.

Ejsmont, Artur. *Web Scalability for Startup Engineers: Tips & Techniques for Scaling Your Web Application*. McGraw-Hill Education, 2015.

Allspaw, John. *The Art of Capacity Planning*. O'Reilly, 2008.

Rasband, Matt, and Eugene Ciurana. "Scalability & High Availability - Dzone Refcardz." Dzone.com, 29 June 2017, dzone.com/refcardz/scalability.

High Scalability - www.highscalability.com

"Scalability." *Cloud Computing Trends: 2018 State of the Cloud Survey*, www.rightscale.com/blog/tag/scalability.