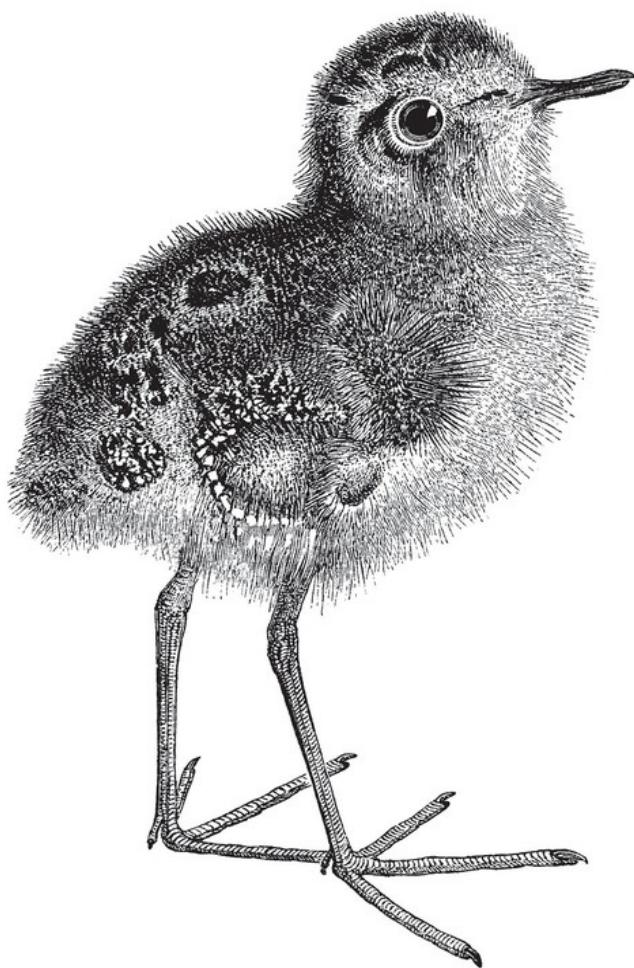


O'REILLY®

# System Design on AWS

Building and Scaling Enterprise Solutions



Early  
Release  
RAW &  
UNEDITED

Jayanth Kumar &  
Mandeep Singh

# **System Design on AWS**

Building and Scaling Enterprise Solutions

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

**Jayanth Kumar and Mandeep Singh**

# **System Design on AWS**

by Jayanth Kumar and Mandeep Singh

Copyright © 2024 Jayanth Kumar and Mandeep Singh. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

- Acquisitions Editor: Megan Laddusaw
- Development Editor: Melissa Potter
- Production Editor: Katherine Tozer
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- October 2024: First Edition

## **Revision History for the Early Release**

- 2023-06-27: First Release
- 2023-08-31: Second Release
- 2023-10-09: Third Release

- 2023-12-07: Fourth Release
- 2024-02-02: Fifth Release
- 2024-03-26: Sixth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098146894> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *System Design on AWS*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-14683-2

[FILL IN]

# Brief Table of Contents (*Not Yet Final*)

---

## Part I System Design

Chapter 1 System Design Trade-Offs and Prescriptions (available)

Chapter 2 Storage Types and Relational Stores (available)

Chapter 3 Non-relational Stores (available)

Chapter 4 Caching Policies and Strategies (available)

*Chapter 5 Scaling Approaches and Mechanisms* (unavailable)

Chapter 6 Communication Networks and Protocols (available)

*Chapter 7 Containerization Images and Deployments* (unavailable)

*Chapter 8 Orchestration Designs and Patterns* (unavailable)

## Part II Diving Deep into AWS Services

Chapter 9 AWS Network Services (available)

Chapter 10 AWS Storage Services (available)

Chapter 11 AWS Compute Services (available)

Chapter 12 AWS Orchestration Services (available)

Chapter 13 AWS Big Data, Analytics, and ML Services (available)

## Part III System Design Use Cases

Chapter 14 Designing URL Shortener (available)

Chapter 15 Designing Web Crawler and Search Engine (available)

*Chapter 16 Designing Social Network and News Feed System*  
(unavailable)

*Chapter 17 Designing Online Game Leadership Board* (unavailable)

*Chapter 18 Designing Payment Gateway and System* (unavailable)

*Chapter 19 Designing Chat Application* (unavailable)

*Chapter 20 Designing Video Processing Pipeline for Streaming Service* (unavailable)

*Chapter 21 Designing Stock Exchange* (unavailable)

# Part I. System Design Basics

---

This unit provides you with a solid foundation in system design, offering insights into the fundamental principles that underpin all types of software systems, ranging from trade-offs to the criteria for picking the right tools and technologies and thus, the right architecture patterns.

By the end of this unit, you will understand:

- The system trade-offs that naturally arise in large-scale distributed systems.
- How to make the right choices for datastore, ranging from file, block, object store to databases, both relational and non-relational databases.
- How to support large throughput and low latency for both the storage and the compute using caches and CDNs.
- How to horizontally scale the system using load balancers, API Gateways and reverse proxy.
- The choices between different communication and network protocols at different layers of OSI and TCP/IP models and how to choose when to use what.
- The common system design architecture patterns, which are commonly used in the industry and how to adapt them to their technical architecture.
- The different AWS services that map to different system design paradigms, which we will explore further in later chapters.

The first eight chapters as part of this unit will cover:

1. Chapter 1 will delve you into the world of trade-offs in large-scale distributed systems. We'll explore essential concepts like reliability, scalability, and maintainability, shedding light on why these trade-offs emerge. We'll tackle common misconceptions through the fallacies of distributed computing and address pivotal choices such as space vs. time, latency vs. throughput, performance vs. scalability, and consistency vs. availability. Additionally, we'll offer practical guidelines and strategies that have emerged from years of system design wisdom, providing tangible approaches to enhance system performance and efficiency.
2. Chapter 2 will introduce you to different types of data storage types, ranging from file, block, and object stores. We will cover the relational databases, their concepts and architecture in this chapter. You'll gain insights into the intricacies of scaling relational databases, including techniques like partitioning, indexing, replication, federation, sharding, and denormalization.
3. Chapter 3 will unveil the architecture of non-relational databases, such as key-value, wide column, document, and graph databases, and how leaderless architectures enable scalability through mechanisms like quorum and consistent hashing.
4. Chapter 4 will introduce you to caching, where we explore diverse caching strategies spanning different tiers, from client and server to application and database levels. We'll dissect core caching approaches like write-through, read-through, refresh-ahead, write-back, and cache-aside. Moreover, we'll unravel the potential of Content Delivery Networks (CDNs) – both Push and Pull CDNs – and demonstrate how they leverage edge locations to reduce latency and optimize content delivery.

5. Chapter 5 will explore vertical and horizontal scaling, elucidating various approaches to scaling compute resources. Load balancers, API gateways, and reverse proxies will take center stage, along with discussions on L4 network scaling and L7 application layer scaling. Concepts like failover strategies, sticky sessions, and reverse proxies will come to life as we delve into the nuances of creating scalable and resilient architectures.
6. Chapter 6 demystifies network protocols, unraveling the intricacies of TCP/IP, UDP, HTTP, RMTP, XMPP, and their suitable applications. You'll gain a comprehensive understanding of synchronous and asynchronous service design and delve into standards like SOAP, REST, GRPC, and GraphQL. By mastering these communication concepts, you'll be better equipped to make informed choices for different use-cases.
7. Chapter 7 is all about containerization – the deployment of images and the orchestration of Docker containers. The concept of serverless architecture will also make its appearance, along with discussions on microservices and their design principles. By the end of this chapter, you'll be well-versed in the intricate world of containers, microservices, and serverless architecture.
8. Chapter 8 discusses orchestration in detail, where we explore asynchronous architectures, message brokers, message queues, and publisher-subscriber patterns. As you delve deeper into large-scale software engineering design, you'll compare microservices with monolithic designs, serverless architecture, saga pattern, event sourcing pattern, lambda architecture, kappa architecture, and domain-driven design. The intricate dance between orchestration and choreography will be unveiled, guiding you through the complexities of designing robust and adaptable systems.

With this comprehensive journey in this unit through the nuances of system design, you'll be well-prepared to wield these principles and

technologies in creating effective and efficient large-scale systems.

Later, Part II will cover the AWS services in detail, bringing these concepts to light in their use and trade-offs.

# Chapter 1. System Design Trade-offs and Guidelines

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

Today's modern technological revolution is happening because of large scale software systems. Big enterprise companies like Google, Amazon, Oracle, and SAP have all built large scale software systems to run their (and their customer's) businesses. Building and operating such large scale software systems requires first principles thinking to design and develop the technical architecture before actually putting the system into code. This is because we don't want to be in a state where these systems will not work/scale after writing 10k lines of code. If the design is right in the first place, the rest of the implementation journey becomes smooth. This requires looking at the business requirements, understanding the needs and objectives of the customer, evaluating different trade-offs, thinking about error handling and edge cases, contemplating futuristic changes and robustness while worrying about basic details like algorithms and data structures. Enterprises can avoid the mistake of wasted software development effort by carefully thinking about systems and investing time in understanding bottlenecks, system requirements, users being targeted, user access patterns and many such decisions, which in short, is System Design.

This chapter covers the basics of system design, with the goal of helping you to understand the concepts around system design itself, the system trade-offs that naturally arise in such large-scale software systems, the fallacies to avoid in building such large scale systems and the guidelines—those wisdoms which were learnt after building such large scale software systems over the years. This is simply meant to introduce you to the basics—we'll dig into details in later chapters, but we want you to have a good foundation to start with. Let's begin by digging into the basic system design concepts.

## System Design Concepts

To understand the building blocks of system design, we should understand the fundamental concepts around systems. We can leverage *abstraction* here, the concept in computer science of obfuscating the inner details to create a model of these system design concepts, which can help us to understand the bigger picture. The concepts in system design, be it any software system, revolve around communication, consistency, availability, reliability, scalability, fault tolerance and system maintainability. We will go over each of these concepts in detail, creating a mental model while also understanding how their nuances are applied in large scale system design.

## Communication

A large scale software system is composed of small sub-systems, known as servers, which communicate with each other, i.e. exchange information or data over the network to solve a business problem, provide business logic, and compose functionality. Communication can take place in either a synchronous or asynchronous fashion, depending on the needs and requirements of the system.

**Figure 1-1** shows the difference in the action sequence of both synchronous and asynchronous communication.

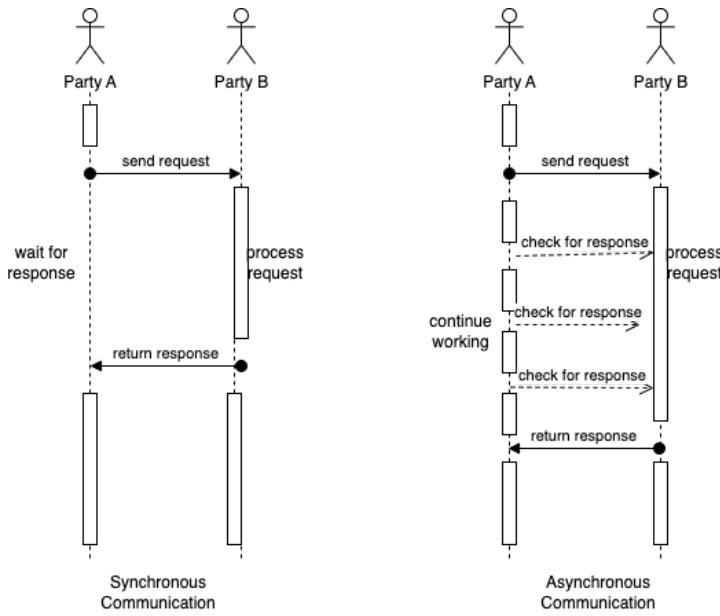


Figure 1-1. Sequence Diagram for Synchronous vs Asynchronous Communication

Let's go over the details of both communication mechanisms in the following sections.

### NOTE

We will cover the communication protocols as well as mechanisms for asynchronous communication in detail in Chapter 6: Communication Networks and Protocols.

## Synchronous Communication

Consider a phone call conversation with your friend, you hear and speak with them at the same time and also use pauses in between to allow for conversation to complete. This is an example of synchronous communication, a type of communication in which two or more parties communicate with each other in real-time, with low latency. This type of communication is more immediate and allows for quicker resolution of issues or questions.

In system design, a communication mechanism is synchronous when the receiver will block (or wait) for the call or execution to return before continuing. This means that until a response is returned by the sender, the application will not execute any further, which could be perceived by the user as latency or performance lag in the application.

## Asynchronous Communication

To give you an example of asynchronous communication, consider that instead of a phone call conversation, you switch to email. As you communicate over email with your friend, you send the

message and wait for the reply at a later time (but within an acceptable time limit). You also follow a practice to follow up again if there is no response after this time limit has passed. This is an example of asynchronous communication, a type of communication in which two or more parties do not communicate with each other in real-time. Asynchronous communication can also take place through messaging platforms, forums, and social media, where users can post messages and responses may not be immediate.

In system design, a communication mechanism is asynchronous when the sender does not block (or wait) for the call or execution to return from the receiver. Execution continues on in your program or system, and when the call returns from the receiving server, a “callback” function is executed. In system design, asynchronous communication is often used when immediate response is not required, or when the system needs to be more flexible and tolerant of delays or failures.

In general, the choice of synchronous or asynchronous communication in system design depends on the specific requirements and constraints of the system. Synchronous communication is often preferred when real-time response is needed (such as the communication between the frontend UI and the backend), while asynchronous communication is often preferred when flexibility and robustness are more important (such as the communication to check the status of a long running job).

## **Consistency**

Consistency, i.e. the requirement of being consistent, or in accordance with a set of rules or standards, is an important issue when it comes to communication between servers in a software system. Consistency can refer to a variety of concepts and contexts in system design.

In the context of distributed systems, consistency can be the property of all *replica* nodes (more on this in a moment) or servers having the same view of data at a given point in time. This means that all replica nodes have the same data, and any updates to the data are immediately reflected on all replica nodes.

In the context of data storage and retrieval, consistency refers to the property of each read request returning the value of the most recent write. This means that if a write operation updates the value of a piece of data, any subsequent read requests for that data should return the updated value. Let's discuss each of these in more detail.

### **Consistency in Distributed Systems**

Distributed systems are software systems, which are separated physically but connected over the network to achieve common goals using shared computing resources over the network.

Ensuring consistency, i.e. providing the same view of the data to each server in distributed systems can be challenging, as multiple replica servers may be located in different physical locations and may be subject to different failures or delays.

To address these challenges, we can use various techniques in distributed systems to ensure data consistency, such as:

#### *Data replication*

In this approach, multiple copies of the data are maintained on different replica nodes, and updates to the data are made on all replica nodes simultaneously through blocking synchronous communication. This ensures that all replica nodes have the same view of the data at any given time.

#### *Consensus protocols*

Consensus protocols are used to ensure that all replica nodes agree on the updates to be made to the data. They can use a variety of mechanisms, such as voting or leader election, to ensure that all replica nodes are in agreement before updating the data.

#### *Conflict resolution*

In the event that two or more replica nodes try to update the same data simultaneously, conflict resolution algorithms are used to determine which update should be applied. These algorithms can use various strategies, such as last writer wins or merge algorithms, to resolve conflicts.

Overall, ensuring consistency in a distributed system is essential for maintaining the accuracy and integrity of the data, and various techniques are used to achieve this goal.

### **Consistency in Data Storage and Retrieval**

Large scale software systems produce and consume a large amount of data and thus, ensuring consistency in such data storage and retrieval is important for maintaining the accuracy and integrity of the data in these systems. For example, consider a database that stores the balance of a bank account. If we withdraw money from the account, the database should reflect the updated balance immediately. If the database does not ensure consistency, it is possible for a read request to return an old balance, which could lead to incorrect financial decisions or even, financial loss for us or our banks.

To address these challenges, we can use various techniques in data storage systems to ensure read consistency, such as:

#### *Write-ahead logging*

In this technique, writes to the data are first recorded in a log before they are applied to the actual data. This ensures that if the system crashes or fails, the data can be restored to a consistent state by replaying the log.

#### *Locking*

Locking mechanisms are used to ensure that only one write operation can be performed at a time. This ensures that multiple writes do not interfere with each other and that reads always return the value of the most recent write.

#### *Data versioning*

In this technique, each write operation is assigned a version number, and reads always return the value of the most recent version. This allows for multiple writes to be performed concurrently, while still ensuring that reads return the value of the most recent write.

Overall, ensuring consistency in data storage and retrieval is essential for maintaining the accuracy and integrity of the data, and various techniques are used to achieve this goal.

#### **NOTE**

We will discuss some of the above techniques for ensuring Consistency in detail in Chapter 2: Storage Types and Relational Stores and Chapter 3: Non-Relational Stores.

## Consistency Spectrum Model

Since consistency can mean different things, the consistency spectrum model helps us reason about whether a distributed system is working correctly, when it's doing multiple concurrent things at the same time like reading, writing, and updating data.

The consistency spectrum model represents the various consistency guarantees that a distributed system can offer, ranging from Eventual Consistency to Strong Consistency. The specific consistency guarantee chosen depends on the specific requirements and constraints of the system. Let's walk through the consistency levels in the consistency spectrum model.

### *Strong Consistency*

At one end of the spectrum, strong consistency guarantees that all replica nodes have the same view of the data at all times, and that any updates to the data are immediately reflected on all replica nodes. This ensures that the data is always accurate and up-to-date, but can be difficult to achieve in practice, as it requires all replica nodes to be in constant communication with each other.

#### **NOTE**

We will cover the strong consistency requirements of relational databases as part of ACID property in Chapter 2: Storage Types and Relational Stores.

### *Monotonic read consistency*

Monotonic read consistency guarantees that once a client has read a value from a replica node, all subsequent reads from that client will return the same value or a more recent value. This means that a client will not see "stale" data that has been updated by another client. This provides a stronger consistency guarantee than eventual consistency, as it ensures that a client will not see outdated data.

### *Monotonic write consistency*

Monotonic write consistency guarantees that once a write operation has been acknowledged by a replica node, all subsequent reads from that replica node will return the updated value. This means that a replica node will not return outdated data to clients after a write operation has been acknowledged. This provides a stronger consistency guarantee than eventual consistency, as it ensures that a replica node will not return outdated data to clients.

### *Causal consistency*

Causal consistency works by categorizing operations into dependent and independent operations. Dependent operations are also called causally-related operations. Causal consistency preserves the order of the causally-related operations. It guarantees that if two operations are causally related into dependent and independent operations, then they will be seen in the same order by all processes in the system. This means that if operation A must happen before operation B, then all processes in the system will see A before they see B. This provides a stronger consistency guarantee than eventual consistency, as it ensures that the order of related operations is preserved.

### *Eventual Consistency*

At the other end of the spectrum, eventual consistency guarantees that, given enough time, all replica nodes will eventually have the same view of the data. This allows for more flexibility and tolerance of delays or failures, but can result in temporary inconsistencies in the data.

#### NOTE

We will cover the tunable consistency feature of some of the non-relational columnar databases like Cassandra in Chapter 3: Non-relational Stores.

**Figure 1-2** shows the difference in the result of performing action sequence under strong consistency and eventual consistency. As you can see in the figure on the left for strong consistency, when x is read from a replica node after updating it from 0 to 2, it will block the request until replication happens and then, return 2 as result. On the other side in the figure on the right for eventual consistency, on querying the replica node, it will give stale result of x as 0 before replication completes.

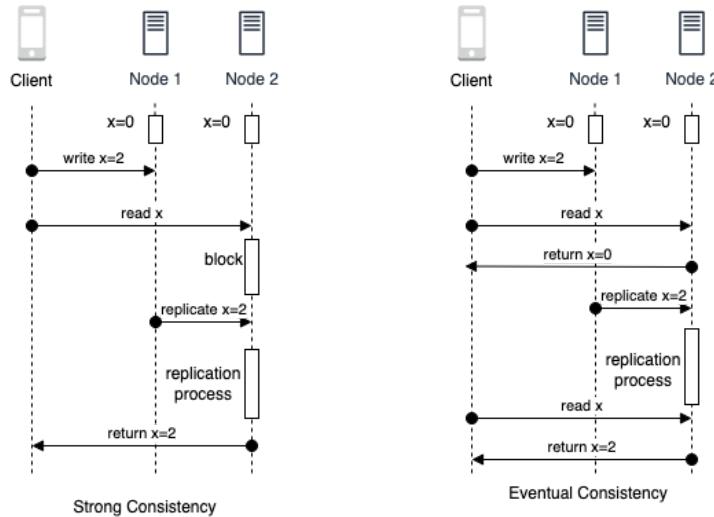


Figure 1-2. Sequence Diagram for Strong Consistency and Eventual Consistency

In general, the consistency spectrum model provides a framework for understanding the trade-offs between consistency and availability in distributed systems, and helps system designers choose the appropriate consistency guarantee for their specific needs.

## Availability

In a large-scale software system, subsystems or servers can go down and may not be fully available to respond to the client's requests — this is referred to as system's availability. A system that is highly available is able to process requests and return responses in a timely manner, even under heavy load or in the face of failures or errors. Let's try to quantify the measurement of availability of the system.

### Measuring Availability

Availability can be measured mathematically as the percentage of the time the system was up (total time - time system was down) over the total time the system should have been running.

$$\text{Availability\%} = \frac{\text{(Total Time} - \text{Sum total of time system was down)}}{\text{Total Time}} \times 100$$

Availability percentages are represented in 9s, based on the above formula over a period of time. You can see the breakdown of what these numbers really work out to in Table 1-1.

T  
a  
b  
l  
e  
1  
-  
1  
.A  
v  
a  
il  
a  
b  
il  
it  
y  
P  
e  
r  
c  
e  
n  
t  
a  
g  
e  
s  
R  
e  
p  
r  
e  
s  
e  
n  
t  
e  
d  
i  
n  
9  
s

---

Availability %   Downtime per Year   Downtime per Month   Downtime per Week

90% (1 nine) 36.5 days 72 hours 16.8 hours

99% (2 nines) 3.65 days 7.2 hours 1.68 hours

99.5% (2 nines) 1.83 days 3.60 hours 50.4 minutes

99.9% (3 nines) 8.76 hours 43.8 minutes 10.1 minutes

99.99% (4 nines) 52.56 minutes 4.32 minutes 1.01 minutes

99.999% (5 nines) 5.26 minutes 25.9 seconds 6.05 seconds

99.9999% (6 nines) 31.5 seconds 2.59 seconds 0.605 seconds

99.99999% (7 nines) 3.15 seconds 0.259 seconds 0.0605 seconds

The goal for availability is usually to achieve the highest level possible, such as "five nines" (99.999%) or even "six nines" (99.9999%). However, the level of availability that is considered realistic or achievable depends on several factors, including the complexity of the system, the resources available for maintenance and redundancy, and the specific requirements of the application or service.

Achieving higher levels of availability becomes progressively more challenging and resource-intensive. Each additional nine requires an exponential increase in redundancy, fault-tolerant architecture, and rigorous maintenance practices. It often involves implementing redundant components, backup systems, load balancing, failover mechanisms, and continuous monitoring to minimize downtime and ensure rapid recovery in case of failures.

While some critical systems, such as financial trading platforms or emergency services, may strive for the highest levels of availability, achieving and maintaining them can be extremely difficult and costly. In contrast, for less critical applications or services, a lower level of availability, such as 99% or 99.9%, may be more realistic and achievable within reasonable resource constraints.

Ultimately, the determination of what level of availability is realistic and achievable depends on a careful evaluation of the specific requirements, resources, costs, and trade-offs involved in each

particular case.

### Availability in parallel vs in sequence

The availability of a system that consists of multiple sub-systems depends on whether the components are arranged in sequence or in parallel with respect to serving the request.

Figure 1-3 shows the arrangement of components in sequential system on the left, where the request needs to be served from each component in sequence vs the parallel system on the right, where the request can be served from either component in parallel.

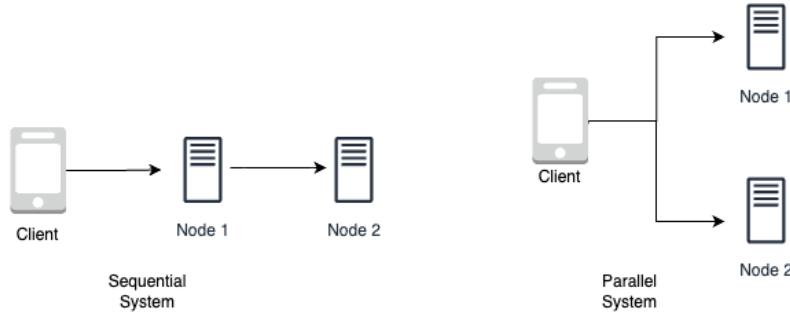


Figure 1-3. Sequential system vs Parallel system

If the components are in sequence, the overall availability of the service will be the product of the availability of each component. For example, if two components with 99.9% availability are in sequence, their total availability will be 99.8%.

$$\begin{aligned} \text{Availability of two components in sequence} &= \text{Availability of component 1} * \text{Availability of component 2} \\ &= .999 * .999 = .998001 \sim 99.8\% \end{aligned}$$

On the other hand, if the components are in parallel, the overall availability of the service will be the sum of the availability of each component minus the product of their unavailability. For example, if two components with 99.9% availability are in parallel, their total availability will be 99.9999%. This can lead to significantly higher availability compared to the same components arranged in sequence (6 9s from 3 9s in the above example).

$$\begin{aligned} \text{Availability of 2 components in parallel} &= 1 - \text{Unavailability of both components} \\ &= 1 - P(\text{component 1 going down}) * P(\text{component 2 going down}) = 1 - (.001 * .001) \\ &= 1 - .000001 = .999999 \sim 99.9999\% \end{aligned}$$

Overall, the arrangement of components in a service can have a significant impact on its overall availability, and it is important to consider this when designing a system for high availability.

### Ensuring Availability

Ensuring availability in a system is important for maintaining the performance and reliability of the system. There are several ways to increase the availability of a system, including:

#### Redundancy

By having multiple copies of critical components or subsystems, a system can continue to function even if one component fails. This can be achieved through the use of redundant load balancers, failover systems, or replicated data stores.

#### Fault tolerance

By designing systems to be resistant to failures or errors, the system can continue to function even in the face of unexpected events. This can be achieved through the use of error-handling mechanisms, redundant hardware, or self-healing systems.

### *Load balancing*

By distributing incoming requests among multiple servers or components, a system can more effectively handle heavy load and maintain high availability. This can be achieved through the use of multiple load balancers or distributed systems.

#### **NOTE**

We will cover load balancing in detail in Chapter 5: Scaling Approaches and Mechanisms and the different types of AWS load balancers in Chapter 9: AWS Network Services.

Overall, ensuring availability in a system is important for maintaining the performance and reliability of the system, and various techniques can be used to increase availability.

## **Availability Patterns**

To ensure availability, there are two major complementary patterns to support high availability: fail-over and replication pattern.

### *Failover Patterns*

Failover refers to the process of switching to a redundant or backup system in the event of a failure or error in the primary system. The failover pattern chosen depends on the specific requirements and constraints of the system, including the desired level of availability and the cost of implementing the failover solution. There are two main types of failover patterns: active-active and active-passive.

#### *Active-active failover*

In an active-active failover pattern as shown on the left in [Figure 1-4](#), multiple systems are used in parallel, and all systems are actively processing requests. If one system fails, the remaining systems can continue to process requests and maintain high availability. This approach allows for more flexibility and better utilization of resources, but can be more complex to implement and maintain.

#### *Active-passive failover*

In an active-passive failover pattern as shown on the right in [Figure 1-4](#), one system is designated as the primary system and actively processes requests, while one or more backup systems are maintained in a passive state. If the primary system fails, the backup system is activated to take over processing of requests. This approach is simpler to implement and maintain, but can result in reduced availability if the primary system fails, as there is a delay in switching to the backup system.

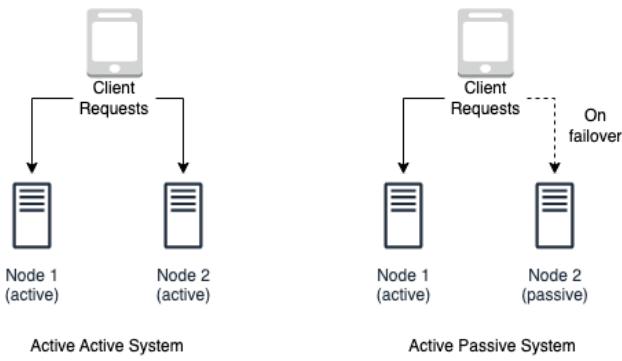


Figure 1-4. Active Active Failover system setup vs Active Passive Failover system setup

The failover pattern can involve the use of additional hardware and can add complexity to the system. There is also the potential for data loss if the active system fails before newly written data can be replicated to the passive system. Overall, the choice of failover pattern depends on the specific requirements and constraints of the system, including the desired level of availability and the cost of implementing the failover solution.

#### NOTE

These failover patterns are employed in relational datastores, non-relational data stores and caches, and load balancers, which we will cover in detail in Chapter 2: Storage Types and Relational Stores, Chapter 3: Non-relational Stores, Chapter 4: Caching Policies and Strategies and Chapter 5: Scaling Approaches and Mechanisms respectively.

### *Replication Patterns*

Replication is the process of maintaining multiple copies of data or other resources in order to improve availability and fault tolerance. The replication pattern chosen depends on the specific requirements and constraints of the system, including the desired level of availability and the cost of implementing the replication solution.

There are two main types of replication patterns: Multi leader and Single leader.

#### *Multi leader replication*

In a multi leader replication pattern as shown on the left in [Figure 1-5](#), multiple systems are used in parallel and all systems are able to both read and write data. This allows for more flexibility and better utilization of resources, as all systems can process requests and updates to the data simultaneously. A load balancer is required or application logic changes need to be made to support multiple leaders and identify on which leader node to write. Most multi leader systems are either loosely consistent or have increased write latency due to synchronization to remain consistent. Conflict resolution comes more into play as more write nodes are added, leading to increase in latency. However, this approach can become more complex to implement and maintain, as it requires careful management of conflicts and errors.

#### *Single leader replication*

In a single leader replication pattern as shown on the right in [Figure 1-5](#), one system is designated as the leader system and is responsible for both reading and writing data, while one or more follower systems are used to replicate the data. The follower systems can only be used for reading data, and updates to the data must be made on the leader system. Additional logic is required to be implemented to promote a follower to the leader. This approach is simpler to implement and maintain, but can result in reduced availability if the leader system fails, as

updates to the data can only be made on the leader system and there is a risk of losing the data updates.

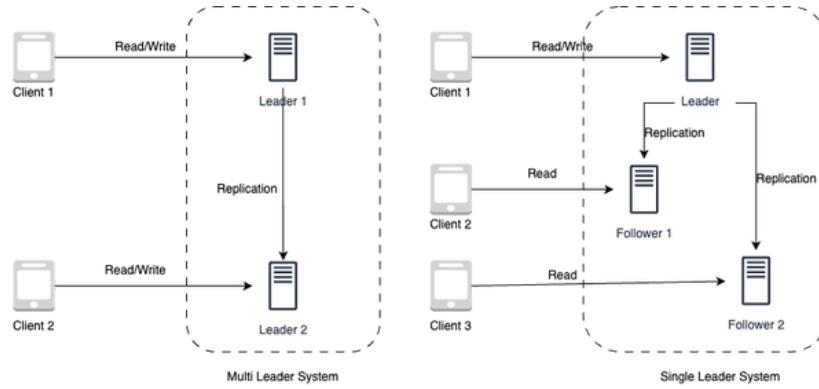


Figure 1-5. Multi-leader replication system setup vs Single leader replication system setup

There is a risk of data loss if the leader system fails before newly written data can be replicated to other nodes. And thus, the more read replicas that are used, the more writes need to be replicated, which can lead to greater replication lag. In addition, the use of read replicas can impact the performance of the system, as they may be bogged down with replaying writes and unable to process as many reads. Furthermore, replication can involve the use of additional hardware and can add complexity to the system. Finally, some systems may have more efficient write performance on the leader system, as it can spawn multiple threads to write in parallel, while read replicas may only support writing sequentially with a single thread. Overall, the choice of replication pattern depends on the specific requirements and constraints of the system, including the desired level of availability and the cost of implementing the replication solution.

#### NOTE

We will cover how relational and non-relational datastores ensure availability using single leader and multi-leader replication in Chapter 2: Storage Types and Relational Stores and Chapter 3: Non-relational Stores. Do look out for leaderless replication using consistent hashing to ensure availability in non-relational stores like key-value stores and columnar stores in detail in Chapter 3: Non-relational Stores.

## Reliability

In system design, reliability refers to the ability of a system or component to perform its intended function consistently and without failure over a given period of time. It is a measure of the dependability or trustworthiness of the system. Reliability is typically expressed as a probability or percentage of time that the system will operate without failure. For example, a system with a reliability of 99% will fail only 1% of the time. Let's try to quantify the measurement of reliability of the system.

### Measuring Reliability

One way to measure the reliability of a system is through the use of mean time between failures(MTBF) and mean time to repair (MTTR).

#### Mean time between failures

Mean time between failures (MTBF) is a measure of the average amount of time that a system can operate without experiencing a failure. It is typically expressed in hours or other units of

time. The higher the MTBF, the more reliable the system is considered to be.

$$\text{Mean time between failures (MTBF)} = \frac{\text{(Total Elapsed Time - Sum total of time system was down)}}{\text{Total Number of Failures}}$$

#### Mean time to repair

Mean time to repair (MTTR) is a measure of the average amount of time it takes to repair a failure in the system. It is also typically expressed in hours or other units of time. The lower the MTTR, the more quickly the system can be restored to operation after a failure.

$$\text{Mean time to repair (MTTR)} = \frac{\text{(Total Maintenance Time)}}{\text{Total Number of Repairs}}$$

Together, MTBF and MTTR can be used to understand the overall reliability of a system. For example, a system with a high MTBF and a low MTTR is considered to be more reliable than a system with a low MTBF and a high MTTR, as it is less likely to experience failures and can be restored to operation more quickly when failures do occur.

### Reliability and Availability

It is important to note that reliability and availability are not mutually exclusive. A system can be both reliable and available, or it can be neither. A system that is reliable but not available is not particularly useful, as it may be reliable but not able to perform its intended function when needed.

On the other hand, a system that is available but not reliable is also not useful, as it may be able to perform its intended function when needed, but it may not do so consistently or without failure. In order to achieve high reliability and availability to meet agreed service level objectives (SLO), it is important to design and maintain systems with redundant components and robust failover mechanisms. It is also important to regularly perform maintenance and testing to ensure that the system is operating at its optimal level of performance.

In general, the reliability of a system is an important consideration in system design, as it can impact the performance and availability of the system over time.

#### NOTE

Service level objectives and goals including Change Management, Problem Management and Service Request Management of AWS Managed Services, which will be introduced in section 2 - Diving Deep into AWS Services are provided in [AWS documentation here](#).

### Scalability

In system design, we need to ensure that the performance of the system increases with the resources added based on the increasing workload, which can either be request workload or data storage workload. This is referred to as scalability in system design, which requires the system to respond to increased demand and load. For example, a social network needs to scale with the increasing number of users as well as the content feed on the platform, which it indexes and serves.

#### Scalability Patterns

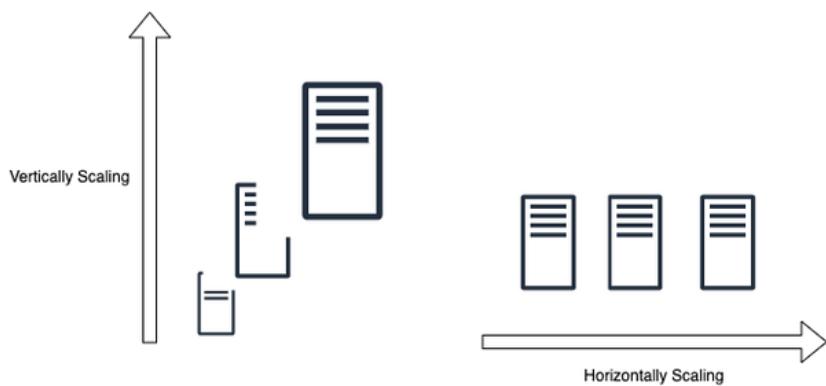
To ensure scalability, there are two major complementary patterns to scale the system: vertical scaling and horizontal scaling.

### *Vertical Scaling*

Vertical scaling involves meeting the load requirements of the system by increasing the capacity of a single server by upgrading it with more resources (CPU, RAM, GPU, storage etc) as shown on the left in [Figure 1-6](#). Vertical scaling is useful when dealing with predictable traffic, as it allows for more resources to be used to handle the existing demand. However, there are limitations to how much a single server can scale up based on its current configuration and also, the cost of scaling up is generally high as adding more higher end resources to the existing server will require more dollars for high end configurations.

### *Horizontal Scaling*

Horizontal scaling involves meeting the load requirements of the system by increasing the number of the servers by adding more commodity servers to serve the requests as shown on the right in [Figure 1-6](#). Horizontal scaling is useful when dealing with unpredictable traffic, as adding more servers increases the servers capacity to handle more requests and if demand arises, more servers can further be added to the pool cost-effectively. However, though horizontal scaling provides a better dollar cost proposition for scaling, the complexity of managing multiple servers and ensuring they work collectively as an abstracted single server to handle the workload is the catch here.



*Figure 1-6. Vertical Scaling vs Horizontal Scaling*

#### **NOTE**

We will cover both scaling approaches and mechanisms in detail in Chapter 5: Scaling Approaches and Mechanisms.

In early stage systems, you can start scaling up by vertically scaling the system and adding better configuration to it and later, when you hit the limitation in further scaling up, you can move to horizontally scaling the system.

### **Maintainability**

In system design, maintainability is the ability of the system to be modified, adapted, or extended to meet the changing needs of its users while ensuring smooth system operations. In order for a software system to be maintainable, it must be designed to be flexible and easy to modify or extend.

The maintainability of a system requires covering these three underlying aspects of the system:

#### *Operability*

This requires the system to operate smoothly under normal conditions and even return back to normal operations within stipulated time after a fault. When a system is maintainable in terms of

operability, it reduces the time and effort required to keep the system running smoothly. This is important because efficient operations and management contribute to overall system stability, reliability, and availability.

#### *Lucidity*

This requires the system to be simple and lucid to understand, extend to add features and even, fix bugs. When a system is lucid, it enables efficient collaboration among team members, simplifies debugging and maintenance tasks, and facilitates knowledge transfer. It also reduces the risk of introducing errors during modifications or updates.

#### *Modifiability*

This requires the system to be built in a modular way to allow it to be modified and extended easily, without disrupting the functionality of other subsystems. Modifiability is vital because software systems need to evolve over time to adapt to new business needs, technological advancements, or user feedback. A system that lacks modifiability can become stagnant, resistant to change, and difficult to enhance or adapt to future demands.

By prioritizing maintainability, organizations can reduce downtime, lower maintenance costs, enhance productivity, and increase the longevity and value of their software systems.

## **Fault Tolerance**

Large scale systems generally employ a large number of servers and storage devices to handle and respond to the user requests and store data. Fault tolerance requires the system to recover from any failure (either hardware or software failure) and continue to serve the requests. This requires avoiding single points of failure in the large system and the ability to reroute requests to the functioning sub-systems to complete the workload.

Fault tolerance needs to be supported at hardware as well as software levels, while ensuring the data safety, i.e. making sure we don't lose the data. There are two major mechanisms to ensure data safety: replication and checkpointing.

### **Replication**

Replication based fault tolerance ensures data safety as well as serving the request by replicating both the service through multiple replica servers and also, replicating the data through multiple copies of data across multiple storage servers. During a failure, the failed node gets swapped with a fully functioning replica node. Similarly data is also served again from a replica store, in case the data store has failed. The replication patterns were already covered in the previous section in availability.

### **Checkpointing**

Checkpointing based fault tolerance ensures that data is reliably stored and backed up, even after the initial processing is completed. It allows for a system to recover from any potential data loss, as it can restore a previous system state and prevent data loss. Checkpointing is commonly used to ensure system and data integrity, especially when dealing with large datasets. It can also be used to verify that data is not corrupted or missing, as it can quickly detect any changes or issues in the data and then take corrective measures. Checkpointing is an important tool for data integrity, reliability, and security, as it ensures that all data is stored properly and securely.

#### NOTE

Recovery Manager of databases use checkpointing to ensure the durability and reliability of the database in the event of failures or crashes. This will be covered in detail in Chapter 2: Storage Types and Relational Stores.

There are two checkpointing patterns — it can be done in either using synchronous or asynchronous mechanisms.

#### *Synchronous checkpointing*

Synchronous checkpointing in a system is achieved by stopping all the data mutation requests and allowing only read requests, while waiting for all the checkpointing process to complete for the current data mutation to ensure its integrity across all nodes. This always ensures consistent data state across all the nodes.

#### *Asynchronous checkpointing*

Asynchronous checkpointing in a system is done by checkpointing asynchronously on all the nodes, while continuing to serve all the requests (including data mutation requests) without waiting for the acknowledgement of the checkpointing process to complete. This mechanism suffers from the possibility of having inconsistent data state across the servers.

So now we have covered the basic concepts and requirements of a large scale system and strive towards building a performant and scalable system—one that is also highly available, reliable, maintainable and is fault-tolerant. Before diving deep into how to build such systems, let's go through the inherent fallacies as well as trade-offs in designing such systems.

## **Fallacies of Distributed Computing**

As a large-scale software system involves multiple distributed systems, it is often subject to certain fallacies that can lead to incorrect assumptions and incorrect implementations. **These fallacies** were first introduced by [L. Peter Deutsch](#) and they cover the common false assumptions that software developers make while implementing distributed systems. These eight fallacies are:

#### *Reliable Network*

The first fallacy is assuming that “The network is reliable”. Networks are complex, dynamic and often, unpredictable. Small issues like switch or power failures can even bring the entire network of a data-center down, making the network unreliable. Thus, it is important to account for the potential of an unreliable network while designing large scale systems, ensuring network fault tolerance from the start. Given that networks are inherently unreliable, to build reliable services on top we must rely on protocols that can cope with network outages and packet loss.

#### *Zero Latency*

The second fallacy is assuming that “Latency is zero”. Latency is an inherent limitation of networks, constrained by the speed of light, i.e. even in perfect theoretical systems, the data can't reach faster than the speed of light between the nodes. Hence, to account for latency, the system should be designed to bring the clients close to data through edge-computing and even choosing the servers in the right geographic data centers closer to the clients and routing the traffic wisely.

#### *Infinite Bandwidth*

The third fallacy is assuming that "Bandwidth is infinite". When a high volume of data is flowing through the network, there is always network resource contention leading to queueing delays, bottlenecks, packet drops and network congestion. Hence, to account for finite bandwidth, build the system using lightweight data formats for the data in transit to preserve the network bandwidth and avoid network congestion. Or use multiplexing, a technique that improves bandwidth utilization by combining data from several sources and send it over the same communication channel/medium.

#### *Secure Network*

The fourth fallacy is assuming that "The network is secure". Assuming a network is always secure, when there are multiple ways a network can be compromised (ranging from software bugs, OS vulnerabilities, viruses and malwares, cross-site scripting, unencrypted communication, malicious middle actors etc) can lead to system compromise and failure. Hence to account for insecure networks, build systems with a security first mindset and perform defense testing and threat modelling of the built system.

#### *Fixed Topology*

The fifth fallacy is assuming that "Topology doesn't change". In distributed systems, the topology changes continuously, because of node failures or node additions. Building a system that assumes fixed topology will lead to system issues and failures due to latency and bandwidth constraints. Hence, the underlying topology must be abstracted out and the system must be built oblivious to the underlying topology and tolerant to its changes.

#### *Single Administrator*

The sixth fallacy is assuming that "There is one administrator". This can be a fair assumption in a very small system like a personal project, but this assumption breaks down in large scale distributed computing, where multiple systems have separate OS, separate teams working on it and hence, multiple administrators. To account for this, the system should be built in a decoupled manner, ensuring repair and troubleshooting becomes easy and distributed too.

#### *Zero Transport cost*

The seventh fallacy is assuming that "Transport cost is zero". Network infrastructure has costs, including the cost of network servers, switches, routers, other hardwares, the operating software of these hardware, and the team cost to keep it running smoothly. Thus, the assumption that transporting data from one node to another is negligible is false and must consequently be noted in budgets to avoid vast shortfalls.

#### *Homogenous Network*

The eight fallacy is assuming that "The network is homogeneous". A network is built with multiple devices with different configurations and using multiple protocols at different layers and therefore we can't assume a network is homogenous. Taking into consideration the heterogeneity of the network as well as focusing on the interoperability of the system,( i.e. ensuring subsystems can communicate and work together despite having such differences) will help to avoid this pitfall.

## NOTE

The AWS Well-Architected Framework consists of six core pillars that provide guidance and best practices for designing and building systems on the AWS cloud, avoiding these fallacies and pitfalls.

*Operational Excellence* pillar avoids the fallacy of Single Administrator and Homogenous Network. *Security* pillar avoids the fallacy of Secure Network. *Reliability* pillar avoids the fallacy of Reliable Network and Fixed Topology. *Performance Efficiency* pillar avoids the fallacy of Zero Latency and Infinite Bandwidth. *Cost Optimization* pillar as well as *Sustainability* pillar avoids the fallacy of Zero Transport Cost.

The book will not cover the AWS Well-Architected Framework in detail and it is left as an excursion for the readers.

These fallacies cover the basic assumptions we should avoid while building large scale systems. Overall, neglecting the fallacies of distributed computing can lead to a range of issues, including system failures, performance bottlenecks, data inconsistencies, security vulnerabilities, scalability challenges, and increased system administration complexity. It is important to acknowledge and account for these fallacies during the design and implementation of distributed systems to ensure their robustness, reliability, and effective operation in real-world environments. Lets also, go through the trade-offs in the next section, which are generally encountered in designing large scale software systems.

## System Design Trade-offs

System design involves making a number of trade-offs that can have a significant impact on the performance and usability of a system. When designing a system, you must consider factors like cost, scalability, reliability, maintainability, and robustness. These factors must be balanced to create a system that is optimized for the particular needs of the user. Ultimately, the goal is to create a system that meets the needs of the user without sacrificing any of these important factors.

For example, if a system needs to be highly reliable but also have scalability, then you need to consider the trade-offs between cost and robustness. A system with a high level of reliability may require more expensive components, but these components may also be robust and allow for scalability in the future. On the other hand, if cost is a priority, then you may have to sacrifice robustness or scalability in order to keep the system within budget.

In addition to cost and scalability, other trade-offs must be taken into account when designing a system. Performance, security, maintainability, and usability are all important considerations that must be weighed when designing a system. There are some theoretical tradeoffs that arise in system design, which we will discuss in this section.

### Time vs Space

Space time trade-offs or time memory trade-offs arise inherently in implementation of the algorithms in computer science for the workload, even in distributed systems. This trade-off is necessary because system designers need to consider the time limitations of the algorithms and sometimes use extra memory or storage to make sure everything works optimally. One example of such a trade-off is in using look-up tables in memory or data storage instead of performing recalculation and thus, serving more requests by just looking up pre-calculated values.

### Latency vs Throughput

Another trade-off that arises inherently in system design is latency vs throughput. Before diving into the trade-off, let's make sure you understand these concepts thoroughly.

### *Latency, Processing time and Response time*

Latency is the time that a request waits to be handled. Until the request is picked up to be handled, it is latent, inactive, queued or dormant.

Processing time, on the other hand, is the time taken by the system to process the request, once it is picked up.

Hence, the overall response time is the duration between the request that was sent and the corresponding response that was received, accounting for network and server latencies.

Mathematically, it can be represented by the following formula:

$$\text{Response Time} = \text{Latency} + \text{Processing Time}$$

### *Throughput and Bandwidth*

Throughput and bandwidth are metrics of network data capacity, and are used to account for network scalability and load. Bandwidth refers to the maximum amount of data that could, theoretically, travel from one point in the network to another in a given time. Throughput refers to the actual amount of data transmitted and processed throughout the network. Thus, bandwidth describes the theoretical limit, throughput provides the empirical metric. The throughput is always lower than the bandwidth unless the network is operating at its maximum efficiency.

Bandwidth is a limited resource as each network device can only handle and process limited capacity of data before passing it to the next network device, and some devices consume more bandwidth than others. Insufficient bandwidth can lead to network congestion, which slows connectivity.

Since latency measures how long the packets take to reach the destination in a network while throughput measures how many packets are processed within a specified period of time, they have an inverse relationship. The more the latency the more they would get queued up in the network, reducing the number of packets that are being processed, leading to lower throughput.

Since the system is being gauged for lower latency under high throughput or load, the metric to capture latency is through percentiles like p50, p90, p99 and so on. For example, the p90 latency is the highest latency value (slowest response) of the fastest 90 percent of requests. In other words, 90 percent of requests have responses that are equal to or faster than the p90 latency value. Note that average latency of a workload is not used as a metric, as averages as point estimates are susceptible to outliers. Because of the latency vs throughput trade-off, the latency metric will go down as the load is increased on the system for higher throughput. Hence, systems should be designed with an aim for maximal throughput within acceptable latency.

## **Performance vs Scalability**

As discussed earlier in the chapter, scalability is the ability of the system to respond to increased demand and load. On the other hand, performance is how fast the system responds to a single request. A service is scalable if it results in increased performance in a manner proportional to resources added. When a system has performance problems, it is slow for a single user (p50 latency = 100ms) while when the system has scalability problems, the system may be fast for some users (p50 latency = 1ms for 100 requests) but slow under heavy load for the users (p50 latency = 100ms under 100k requests).

## Consistency vs Availability

As discussed earlier in the chapter, strong consistency in data storage and retrieval is the guarantee that every read receives the most recent write, while high availability is the requirement of the system to always provide a non-error response to the request. In a distributed system where the network fails (i.e. packets get dropped or delayed due to the fallacies of distributed computing leading to partitions) there emerges an inherent trade-off between strong consistency and high availability. This trade-off is called the CAP theorem, also known as Brewer's theorem.

### CAP Theorem

The CAP theorem, as shown in Venn Diagram [Figure 1-7](#), states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees: consistency (C), availability (A), and partition tolerance (P). According to the theorem, a distributed system can provide at most two of these guarantees at any given time. Systems need to be designed to handle network partitions as networks aren't reliable and hence, partition tolerance needs to be built in. So, in particular, the CAP theorem implies that in the presence of a network partition, one has to choose between consistency and availability.

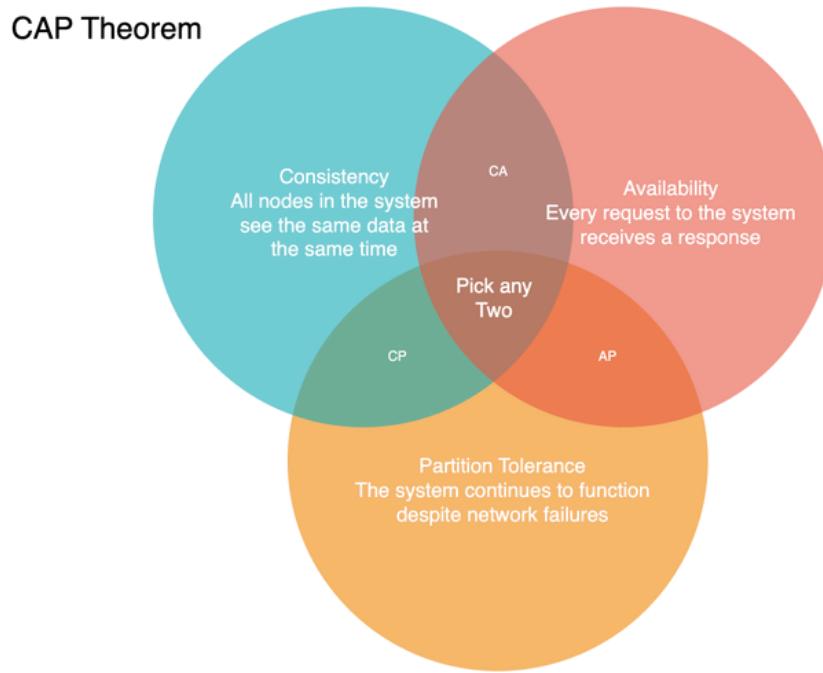


Figure 1-7. CAP Theorem Venn Diagram

However, CAP is frequently misunderstood as if one has to choose to abandon one of the three guarantees at all times. In fact, the choice is really between consistency and availability only when a network partition or failure happens; at all other times, the trade-off has to be made based on the PACELC theorem.

### PACELC Theorem

The PACELC theorem, as shown in Decision Flowchart [Figure 1-8](#), is more nuanced version of CAP theorem, which states that in the case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C) (as per the CAP theorem), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C). This trade-off arises naturally because to handle network

partitions, data and services are replicated in large scale systems, leading to the choice between the consistency spectrum and the corresponding latency.

If the system tries to provide for strong consistency at the one end of the consistency spectrum model, it has to do replication with synchronous communication and blocking to ensure all the read replicas receive the most recent write, waiting on the acknowledgement from all the replica nodes, adding to high latency. On the other hand, if the system does asynchronous replication without waiting for acknowledgment from all nodes, it will end up providing eventual consistency (i.e. the read request will eventually reflect the last recent write) when the replica node has acknowledged the data mutation change for serving the requests.

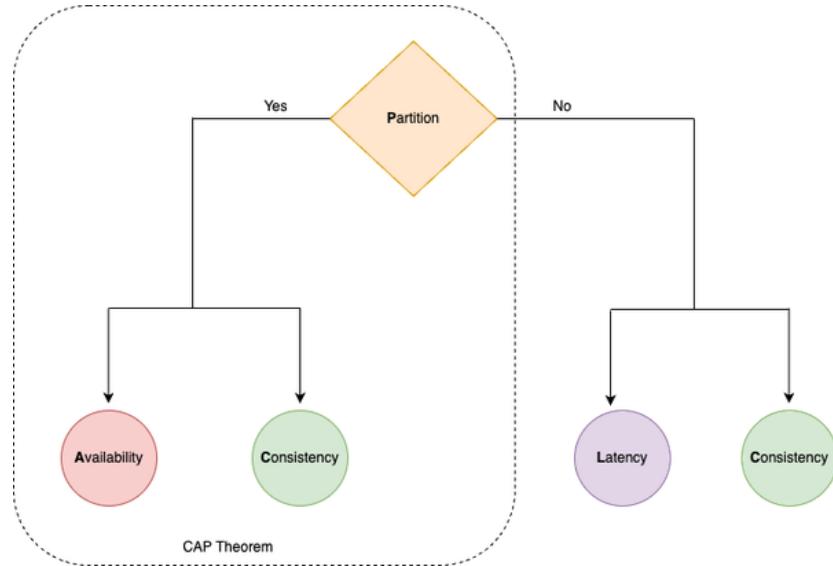


Figure 1-8. PACELC Theorem Decision Flowchart

In summary, the CAP and PACELC theorems are important concepts in distributed systems design that provide a framework for understanding the trade-offs involved in designing highly available and strongly consistent systems.

#### NOTE

We will cover how non-relational stores navigate CAP theorem trade-off by providing BASE property in detail in Chapter 3: Non-relational Stores.

Given such requirements, fallacies and trade-offs in system design, in order to avoid repeating mistakes of the past we should prescribe to a set of guidelines learnt by the previous generation of software design practitioners. Let's dig into those now.

## System Design Guidelines

System design will always present the most interesting and challenging trade-offs, and a system designer should be aware of the hidden costs and be well-equipped to get it right — though not perfect! These guidelines, which have emerged over years of practicing system design, guide us to avoid pitfalls and handle trade-offs while designing large scale-systems. These guidelines aren't just vague generalities but virtues that help reflect on why the system was designed the way it was, why it was implemented the way it is and why that was the right thing to do.

## **Guideline of Isolation: Build It Modularly**

*Controlling complexity is the essence of computer programming.*

—Brian Kernighan

The first guideline is to build the system modularly, i.e. break down a complex system into smaller, independent components or modules that can function independently, yet also work together to form the larger system. Building it modularly helps in improving all the requirements of the large scale system:

### *Maintainability*

Modules can be updated or replaced individually without affecting the rest of the system.

### *Reusability*

Modules can be reused in different systems or projects, reducing the amount of new development required.

### *Scalability*

Modules can be added or removed and even scaled independently as needed to accommodate changes in requirements or to support growth.

### *Reliability*

Modules can be tested and validated independently, reducing the risk of system-wide failures.

Modular systems can be implemented in a variety of ways, including through the use of microservices architecture, component-based development, and modular programming, which we will cover in more detail in chapter 7. However, designing modular systems can be challenging, as it requires careful consideration of the interfaces between modules, data sharing and flow, and dependencies.

## **Guideline of Simplicity: Keep it Simple, Silly**

*Everything should be made as simple as possible, but no simpler*

—Eric S. Raymond

The second guideline is to keep the design simple by avoiding complex and unnecessary features and avoiding over-engineering. To build simple systems using the KISS (Keep it Simple, Silly) guideline, designers can follow these steps:

1. 1. Identify the core requirements: Determine the essential features and functions the system must have, and prioritize them.
2. 2. Minimize the number of components: Reduce the number of components or modules in the system, making sure each component serves a specific purpose.
3. 3. Avoid over-engineering: Don't add unnecessary complexity to the system, such as adding features that are not necessary for its functioning.
4. 4. Make the system easy to use: Ensure the system is intuitive and straightforward for users to use and understand.
5. 5. Test and refine: Test the system to ensure it works as intended and make changes to simplify the system if necessary.

By following the KISS guideline, you as a system designer can build simple, efficient, and effective systems that are easy to maintain and less prone to failure.

## **Guideline of Performance: Metrics Don't Lie**

*Performance problems cannot be solved only through the use of Zen meditation.*

—Jeffrey C. Mogul

The third guideline is to measure then build, and rely on the metrics as you can't cheat the performance and scalability. Metrics and observability are crucial for the operation and management of large scale systems. These concepts are important for understanding the behavior and performance of large-scale systems and for identifying potential issues before they become problems.

### *Metrics*

Metrics are quantitative measures that are used to assess the performance of a system. They provide a way to track key performance indicators, such as resource utilization, response times, and error rates, and to identify trends and patterns in system behavior. By monitoring metrics, engineers can detect performance bottlenecks and anomalies, and take corrective actions to improve the overall performance and reliability of the system.

### *Observability*

Observability refers to the degree to which the state of a system can be inferred from its externally visible outputs. This includes being able to monitor system health and diagnose problems in real-time. Observability is important in large scale systems because it provides a way to monitor the behavior of complex systems and detect issues that may be impacting their performance.

Together, metrics and observability provide the information needed to make informed decisions about the operation and management of large scale systems. By ensuring that systems are properly instrumented with metrics and that observability is designed into the system, you can detect and resolve issues more quickly, prevent outages, and improve the overall performance and reliability of the system.

## **Guideline of Tradeoffs: There Is No Such Thing As A Free Lunch**

*Get it right. Neither abstraction nor simplicity is a substitute for getting it right.*

—Butler Lampson

The fourth guideline is “there is no such thing as a free lunch” (TINSTAAFL), pointing to the realization that all decisions come with trade-offs and that optimizing for one aspect of a system often comes at the expense of others. In system design, there are always trade-offs and compromises that must be made when designing a system. For example, choosing a highly optimized solution for a specific problem might result in reduced maintainability or increased complexity. Conversely, opting for a simple solution might result in lower performance or increased latency.

This guideline TINSTAAFL highlights the need for careful consideration and balancing of competing factors in system design, such as performance, scalability, reliability, maintainability, and cost. Designers must weigh the trade-offs between these factors and make informed decisions that meet the specific requirements and constraints of each project.

Ultimately, you need to realize that there is no single solution that is optimal in all situations and that as system designers, you must carefully consider the trade-offs and implications of their decisions to build the right system.

## Guideline of Use Cases: It Always Depends

*Not everything worth doing is worth doing well.*

—Tom West

The fifth guideline is that design always depends, as system design is a complex and multifaceted process that is influenced by a variety of factors, including requirements, user needs, technological constraints, cost, scalability, maintenance and even, regulations. By considering these and other factors, you can develop systems that meet the needs of the users, are feasible to implement, and are sustainable over time. Since there are many ways to design a system to solve a common problem, it indicates a stronger underlying truth: there is no one "best" way to design the system, i.e. there is no silver bullet. Thus, we settle for something reasonable and hope it is good enough.

## Conclusion

This chapter has given you a solid introduction to how we design software systems. We've talked about the main concepts behind system design, the trade-offs we have to make when dealing with big software systems, the fallacies to avoid in building such large scale systems and the smart guidelines to avoid such fallacies.

Think of system design like balancing on a seesaw – you have to find the right balance between different trade-offs. As an overall guideline, system design is always a trade-off between competing factors, and you as a system designer must carefully balance these factors to create systems that are effective and efficient.

Now, in the next set of chapters (Part I), we're going to explore the basic building blocks of systems. We'll talk about important topics, like where we store data, how we speed things up with caches, how we distribute work with load balancers, and how different parts of a system talk to each other through networking and orchestration.

Once you've got a handle on those basics, we'll dive into the world of AWS systems in Part II. This will help you understand how to make big and powerful systems using Amazon Web Services. And all this knowledge will come together in Part III, where we'll put everything into practice and learn how to design specific use-cases and build large-scale systems on AWS.

But before we get there, our next stop is exploring different ways to store data and introduce you to relational databases in the next chapter.

# Chapter 2. Storage Types and Relation Stores

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

In modern computing systems, data storage is an essential component that plays a vital role in system design. As businesses and organizations continue to scale and grow, they generate and store vast amounts of data, which makes it important to have a scalable and reliable storage infrastructure that can keep up with the increasing demands. Efficient and reliable storage of data is necessary to ensure that applications and services can function effectively and provide optimal performance to end-users. There are various types of data storage solutions available, ranging from traditional file-based systems to more modern block and object stores. Additionally, databases play a significant role in storing and managing structured data, making them a critical aspect of system design.

In this chapter, we will explore the different types of data storage solutions available in the context of system design. We will start by discussing the traditional file-based storage systems and their limitations, before moving on to block and object stores, which have become increasingly popular in recent years due to their ability to handle large amounts of unstructured data. We will also explore the different types of databases available in the relational world in this chapter followed by the non-relational stores in the next chapter, as well as the advantages and disadvantages of each.

As we examine the challenges of scaling relational databases, we'll explore various techniques such as partitioning, indexes, replication, federation, sharding, and denormalization that can be used in relational stores to improve performance and meet the demands of large-scale applications.

By the end of this chapter, you will have developed a solid understanding of the various data storage solutions available, the strengths and weaknesses of different relational database types and how they can be used to design systems that are efficient, reliable, and scalable. This chapter will serve the foundation of the concepts to compare and choose between different AWS relational database offerings including AWS RDS flavors and AWS Aurora covered in Chapter 10.

## **Data Storage Format**

The evolution of data storage hardware over the years posed a challenging problem—how to store the data in a particular format, which hides out the underlying storage hardware and can be read, written and modified across different hardwares and their evolutions. This lead the computer scientists and engineers to come up with device drivers, particularly storage drivers, which were installed on different operating systems to allow working with the data on that

particular storage hardware, stored in a particular “format”, agnostic to underlying hardware.

Three particular formats as shown in **Figure 2-1**, emerged based on how the data is logically arranged on the storage hardware: file-based, block-based, and object-based storage, each with their own capabilities and limitations. File storage organizes and represents the data as a hierarchy of files and folders; Block storage chunks data into arbitrarily organized fixed-size blocks; and Object store organizes them as whole objects linked with the associated metadata. Let's go through each of these storage formats in detail.

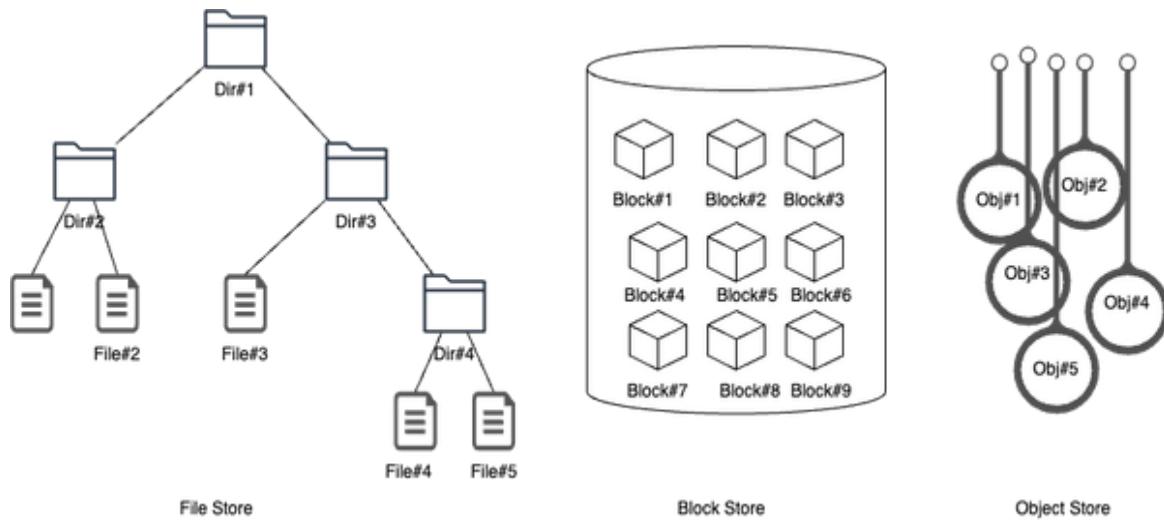


Figure 2-1. Storage abstraction in File store, Block store and Object store

## File Storage

File storage refers to the storage of data in a folder, just like how paper documents are kept in a filing cabinet. When the data is needed, the computer must know the path to find it, which can even be a long arduous path string.

Data in files is organized and retrieved using metadata, like a library card catalog. Consider a closet full of file cabinets. Every document is arranged in some form of logical hierarchy - by cabinet, by drawer, by folder, by file and then, by paper. This is where the concept of hierarchical storage comes from and the file system provides similar

hierarchical structure to organize files, and data is stored in blocks or pages on disk.

It is the oldest and most widely used data storage system for direct and network-attached systems, great for storing complex files and easier to navigate compared to other formats due to the logical hierarchy. Any time you access files on your personal computer, you leverage the file storage.

The caveat with file storage is, just like with your filing cabinet, that the virtual drawer can only open so far. File-based storage systems must scale out by adding more systems, rather than scale up by adding more capacity.

File-based storage is commonly used for storing structured data, such as documents, images, videos, and audio files.

### NOTE

AWS Elastic File Store (EFS) is a scalable, fully managed file storage service offered by AWS that provides shared file storage for EC2 instances, enabling multiple instances to access the same data concurrently, making it suitable for applications requiring shared access to files. We will cover it in more detail in Chapter 10 - AWS Storage Services.

## Block Storage

Block-based storage organizes data into fixed-size blocks or pages, which are stored on disk or flash memory. Each block is assigned a unique address, and data can be read or written to individual blocks. Block storage is designed to separate the data from the user's environment and distribute it across multiple environments that are better suited to serve the data. This means, some block data can be on a Windows environment, some on a Linux environment, and so on. When data is requested, the storage software reassembles the data blocks from these environments and delivers them back to the

user. Typically used in storage-area network (SAN) environments, block storage requires a functioning server to operate.

Since, unlike file storage, block storage doesn't rely on a single path to data, it can be retrieved quickly. Each block lives on its own and can be partitioned so it can be accessed in a different operating system, which gives the user complete freedom to configure their data. It's an efficient and reliable way to store data and is easy to use and manage. It works well with enterprises performing big transactions and those that deploy huge databases, meaning the more data you need to store, the better off you'll be with block storage.

The caveat with Block storage is that it can be expensive. It has limited capability to handle metadata, which means it needs to be dealt with at the application or database level, adding another complexity for a developer or systems administrator to worry about.

Block-based storage is commonly used in enterprise storage systems, such as SAN and NAS, and it provides better performance, reliability, and scalability than file-based storage.

#### NOTE

AWS Elastic Block Storage (EBS) offers scalable block storage volumes on AWS cloud that can be attached to EC2 instances, providing durable, high-performance storage for applications that require low-latency access to data, such as databases and applications that need block-level storage. We will cover it in more detail in Chapter 10 - AWS Storage Services.

## Object Storage

Object-based storage, or object storage, is a type of storage architecture where data is broken down into discrete units called objects and kept in a single repository, rather than in traditional file and folder structures or server blocks. These objects are stored across distributed hardware and are accessed using a unique

identifier and metadata that describes the data, including information like age, security and access contingencies, and even details like the location equipment used to create a video file.

Object storage volumes function as modular units, with each one being a self-contained repository of data. Retrieval of this data is done using the unique identifier and metadata, which distributes the workload and enables administrators to apply policies for more efficient searches. Object storage is also known for its scalability, cost efficiency, and suitability for static data and unstructured data. It has a simple HTTP API that is used by most clients in all programming languages.

However, there are limitations to object storage. Objects cannot be modified, meaning that data must be written entirely at once. Also, object storage does not work well with traditional databases because writing objects is a slow process and programming an app to use object storage API is not as straightforward as using file storage.

### NOTE

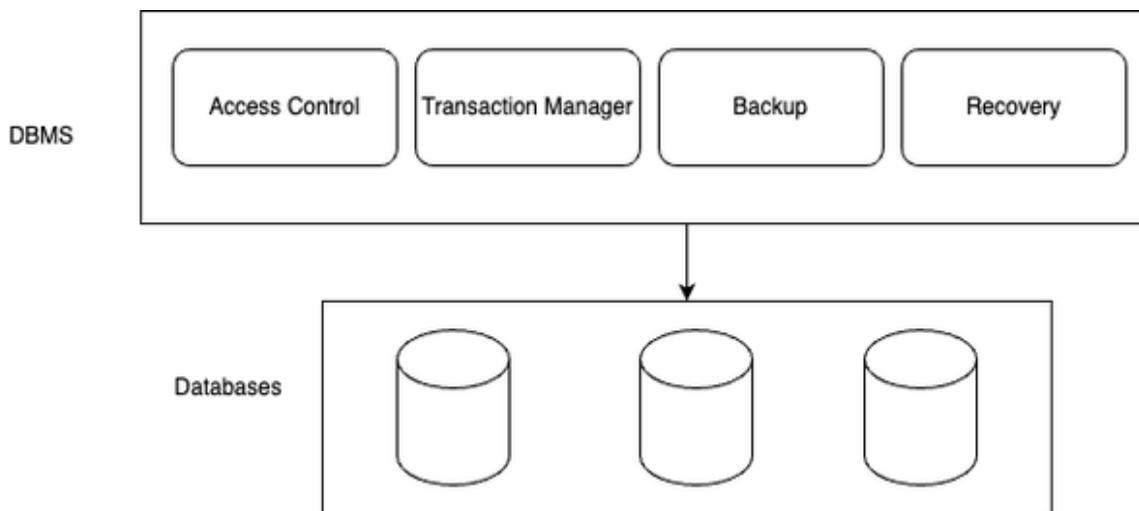
AWS S3 (Simple Storage Service) is an object storage service offering by AWS that provides highly scalable and durable storage for a wide range of data types, accessible via APIs. It is suitable for storing and retrieving large amounts of data, backups, static website content, and as a data lake for analytics. We will cover it in more detail in Chapter 10 - AWS Storage Services.

In summary, the choice of data storage format depends on the type of data being stored, as well as the performance, reliability, and scalability requirements of the system. File-based storage is best suited for structured data, while block-based and object-based storage are more appropriate for unstructured data. Block-based storage provides better performance and reliability than file-based storage, while object-based storage offers better scalability and cost-effectiveness.

We covered different data storage formats and their comparison in terms of usage, cost, performance and scalability. Next let's discuss how data is stored and structured in databases for easy retrieval and processing.

A database is a structured collection of data that is organized and stored in a computer system for easy access, retrieval, and management. It is designed to efficiently store, retrieve, and manipulate large volumes of data. A database typically consists of one or more tables, which are organized into rows and columns. Each row represents a unique record in the database, while each column represents a specific attribute or piece of information about the record.

Database management systems (DBMS), on the other hand as shown in [Figure 2-2](#) are a collection of software systems that sit on top of a database, acting as a bridge between the database and users. DBMS offers multiple interfaces or APIs that enable users to store, retrieve, and manipulate data. Additionally, DBMS provides a range of features related to transactions, recovery, backups, concurrency, authentication and authorization, metadata catalog, and other capabilities.



*Figure 2-2. Databases vs Database Management System*

Databases can be classified into two major types, based on their structure, usage, and functionality: relational and non-relational databases. Let's go through relational databases in detail in this chapter. We'll be covering non-relational databases in Chapter 3.

## Relational Databases

Relational databases use a set of tables with relationships between them to organize data.

Relational databases are the most commonly used type of database, designed to store and manage large amounts of structured data. They are based on the [relational model](#), which was first introduced by Edgar F. Codd in the 1970s. This model organizes data into tables, which can be related to each other using keys.

Tables in a relational database are organized into rows and columns, similar to a spreadsheet. Each row represents a record, and each column represents an attribute or a piece of data about the record. The columns are defined by a data type, which specifies the type of data that can be stored in that column.

[Figure 2-3](#) below shows a sample implementation of logical schema design of an Order database.



Figure 2-3. Logical schema design of Order Database

The list below discusses these concepts, which form the logical components of a database schema design, in detail.

### Tables

Tables are the fundamental units of a relational database. They are used to store and organize data in rows and columns. Each table represents a specific entity or concept, and each row in the table represents an individual instance of that entity, while each column represents a specific attribute or property.

### Rows

Rows, also known as records or tuples, represent individual instances of data stored in a table. Each row contains data values that correspond to the attributes defined by the table's columns.

Rows are unique within a table, typically identified by a primary key.

### *Columns*

Columns, also known as fields or attributes, represent the specific properties or characteristics of the data stored in a table. Each column has a defined data type that determines the kind of data it can store, such as integers, strings, dates, or binary data.

### *Relationships*

One of the key features of a relational database is the ability to create relationships i.e. associations between tables. This is achieved using keys, which are columns that uniquely identify each record in a table. The most common types of relationships are one-to-one, one-to-many, and many-to-many. These relationships help maintain data consistency and enable efficient retrieval of related information.

### *Keys*

Keys are used to establish relationships between tables and ensure data integrity. The primary key uniquely identifies each row in a table, and it must have a unique value for each record. Foreign keys are used to establish relationships between tables by referencing the primary key of another table. By establishing these relationships between tables using keys, data can be easily retrieved and updated across multiple tables. We will cover them in more detail in the next section.

### *Indexes*

Indexes are data structures that enhance the performance of queries by providing quick access to specific data within a table. They are created on one or more columns of a table, allowing

faster search and retrieval operations based on the indexed values.

### *Constraints*

Constraints enforce rules and conditions on the data stored in a database. Common constraints include primary key constraints, foreign key constraints, unique constraints, and check constraints. They ensure data integrity, enforce referential integrity, and prevent the insertion of invalid or inconsistent data.

### *Views*

Views are virtual tables derived from the data stored in one or more tables. They are created based on predefined queries and provide a way to present data in a customized or simplified manner without altering the underlying tables. Views can be used for security purposes, data abstraction, or to simplify complex queries.

### *Transactions*

Relational databases use *transactions* to keep their state consistent. In the context of database management systems, a transaction is a logical unit of work that represents a series of operations performed on a database as a single indivisible unit. These operations may include inserting, updating, or deleting data from one or more tables in the database. The transactions ensure data consistency and integrity by allowing multiple operations to be executed together. The ACID (Atomicity, Consistency, Isolation, Durability) properties govern the behavior of transactions, which we will cover in more detail in the upcoming section.

These logical core components work together to form the foundation of a relational database schema, providing a flexible and efficient

way to store, organize, and retrieve structured data.

## **Relational Database Concepts**

Let's cover basic concepts around relational databases including SQL, ACID, ER Model etc, which form the foundation of modern relational databases.

### **SQL**

Relational databases use a structured query language (SQL) to manipulate and retrieve data. SQL allows users to create, modify, and query databases using a set of commands. Some common SQL commands include SELECT, INSERT, UPDATE, and DELETE.

The query language serves as a programming language that establishes the syntax, structure, and semantics governing interactions with the database. It provides a standardized format for storing, accessing, and manipulating data within the database.

The SQL can be categorized in these four types, each serving a distinct purpose:

#### *Data Definition Language (DDL)*

The Data Definition Language is query language utilized to create and modify the structure and framework of database objects. It encompasses operations such as creating tables, defining indexes, dropping tables, and removing indexes.

#### *Data Manipulation Language (DML)*

The Data Manipulation Language is query language employed to create, update, delete, and retrieve information from the database. It involves operations such as inserting rows into tables, updating existing rows, deleting rows, and querying data.

#### *Data Control Language (DCL)*

The Data Control Language is a query language designed to grant or revoke access to entities and operations within the database for clients. It includes commands such as granting privileges to users, revoking privileges, and managing security permissions.

### *Transaction Control Language (TCL)*

The Transaction Control Language is query language dedicated to managing and ensuring the consistent execution of a group of database operations as a single unit. It encompasses commands such as committing transactions to make changes permanent or rolling back transactions to discard changes.

Collectively, these query languages form the set of SQL query type, providing the necessary tools to define, manipulate, control access, and manage transactions within a database system.

## **ACID**

The ACID model is a set of properties that ensure that database transactions are processed reliably and accurately. ACID stands for:

### *Atomicity*

A transaction must be all or nothing. Atomicity ensures that a sequence of operations is treated as a single logical unit of work. Either all the operations within the unit of work are successfully completed, or none of them are applied at all. This guarantee prevents partial updates to the database. When all operations are successfully executed, it is known as a commit. In case of failure, all the operations are rolled back, reverting the database to its original state. Atomicity provides application developers with the confidence that the database will always be in a consistent state, even in the event of failures. It also enables safe retrying of operations without concerns about creating duplicate data.

## *Consistency*

A transaction must maintain the consistency of the database. Consistency guarantees ensure that the database transitions from one valid state to another. During the transition, the database enforces rules and constraints defined by the application to maintain data integrity. Consistency is a user-controlled property, meaning that the application must define the valid rules and constraints that lead to a consistent final state. For example, an application may define a rule that the balance of a bank account should always remain positive. The database itself does not enforce these rules but ensures that any changes made to the data adhere to the defined consistency constraints.

## *Isolation*

Transactions must be isolated from each other to ensure that they do not interfere with each other. Isolation ensures that concurrent transactions do not interfere with each other, maintaining data integrity and preventing conflicts. Multiple transactions can run simultaneously, and the isolation guarantee dictates how they should interact. If the result of concurrently executing transactions is the same as if they were executed sequentially, then the database supports isolation. Isolation levels, such as Read Uncommitted, Read Committed, Repeatable Read, and Serializable, define the degree of isolation provided by the database. Each level offers different trade-offs between concurrency and data integrity, allowing applications to choose the appropriate level based on their requirements.

## *Durability*

Once a transaction is committed, its changes must be permanent and survive any subsequent failures. Durability guarantees ensure that once a transaction is committed, its changes are permanently stored and will survive system failures such as

crashes or power outages. Typically, this involves persisting the data to nonvolatile storage like a disk. Durability ensures that critical data remains safe and accessible in the long term. Even in the face of catastrophic events, the database will retain the committed changes and recover them when the system is restored.

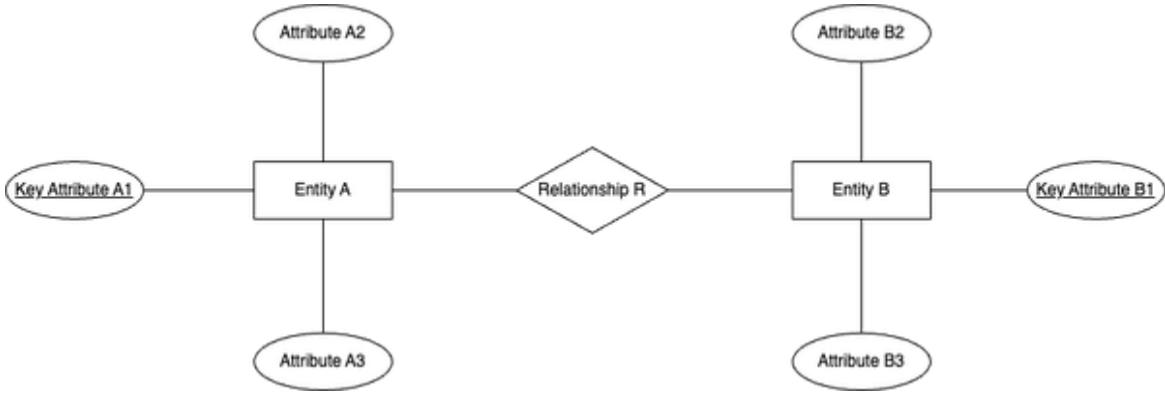
The ACID properties ensure that database transactions are processed in a reliable and consistent manner, even in the presence of failures or concurrent operations. By enforcing these properties, database management systems can maintain the integrity and reliability of the data stored in the database.

In summary, ACID properties provide a set of guidelines for ensuring that database transactions are processed in a reliable and consistent manner, even in the presence of failures or concurrent operations. By adhering to these properties, database management systems can ensure that the data stored in the database remains accurate, reliable, and consistent.

## ER Model

The ER (Entity-Relationship) model is a conceptual model used in relational database design to describe the relationships between entities (objects or concepts) and their attributes. It provides a graphical representation of the database schema, which can be used to design, communicate, and understand the structure of the database.

The ER model, as shown in [Figure 2-4](#) represents entities as rectangles, attributes as ovals, and relationships between entities as diamonds. Relationships can be one-to-one, one-to-many, or many-to-many. The ER model is a useful tool for designing and communicating the structure of a database, as it allows designers to visualize the relationships between entities and the attributes that describe them.



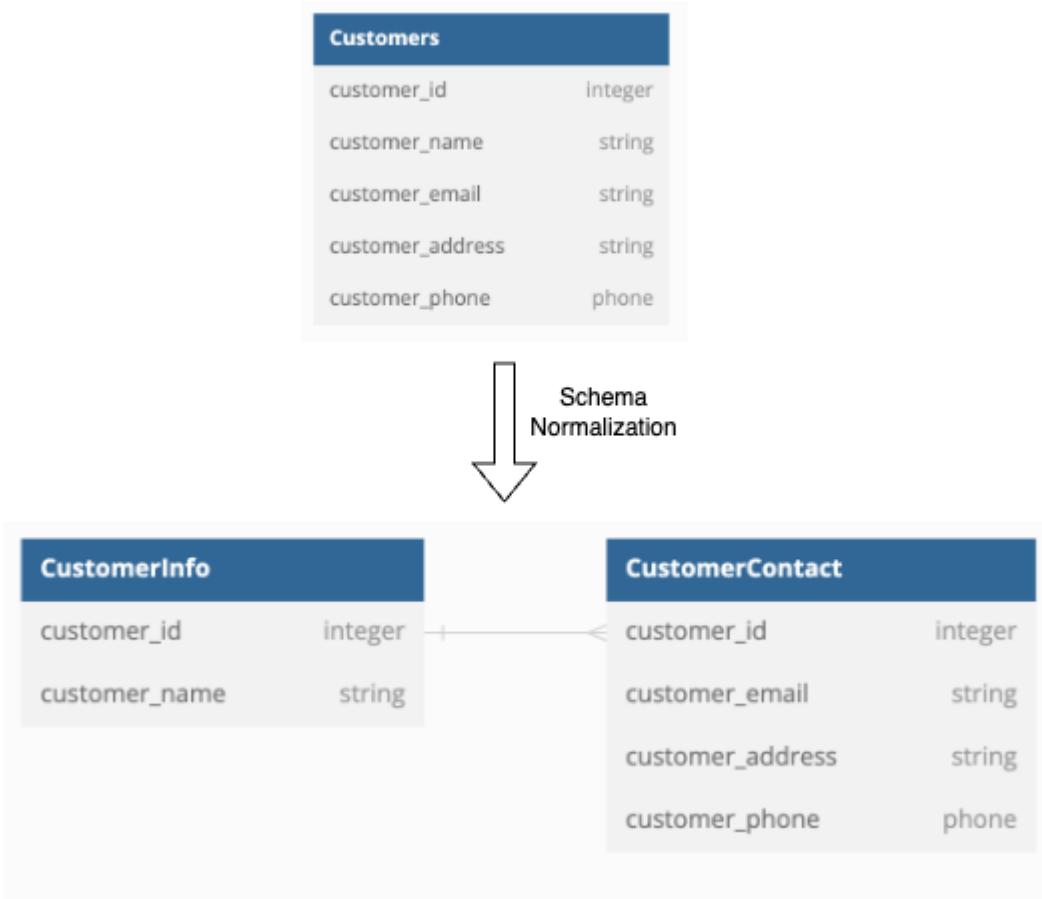
*Figure 2-4. ER Model Representation*

## Schema Normalization

Schema normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves breaking down a larger table into smaller, more manageable tables, each with its own unique purpose and set of attributes. The goal of schema normalization is to minimize data duplication and ensure that each piece of data is stored in only one place in the database.

Let's take an example of a table called "Customers" in our Orders database that has the following columns: customer\_id, customer\_name, customer\_email, customer\_address, and customer\_phone. In this table, the customer\_name, customer\_email, customer\_address, and customer\_phone are repeated for each customer.

To normalize this schema as shown in [Figure 2-5](#), we can decompose the "Customers" table into two tables: "CustomerInfo" and "CustomerContact". The "CustomerInfo" table will contain customer\_id and customer\_name, while the "CustomerContact" table will contain customer\_id, customer\_email, customer\_address, and customer\_phone.



*Figure 2-5. Schema Normalization of Customers Table*

By doing this, we have eliminated the redundancy of customer information in the “Customers” table and stored it only once in the “CustomerInfo” table. This normalization reduces the storage space required for the database, eliminates data inconsistencies that may arise from redundant data, and makes it easier to update customer information without having to change it in multiple places.

## Keys

Keys are used in relational databases to establish relationships between tables. A key is a column or set of columns that uniquely identifies each row in a table. There are several types of keys:

- Candidate key: A candidate key is a column or set of columns in a table that can uniquely identify each record in that table. It is called a “candidate” key because it is a potential primary key

“candidate” for the table. For example, in a table of customers, the customer\_email could be a candidate key, as each customer would have a unique email address.

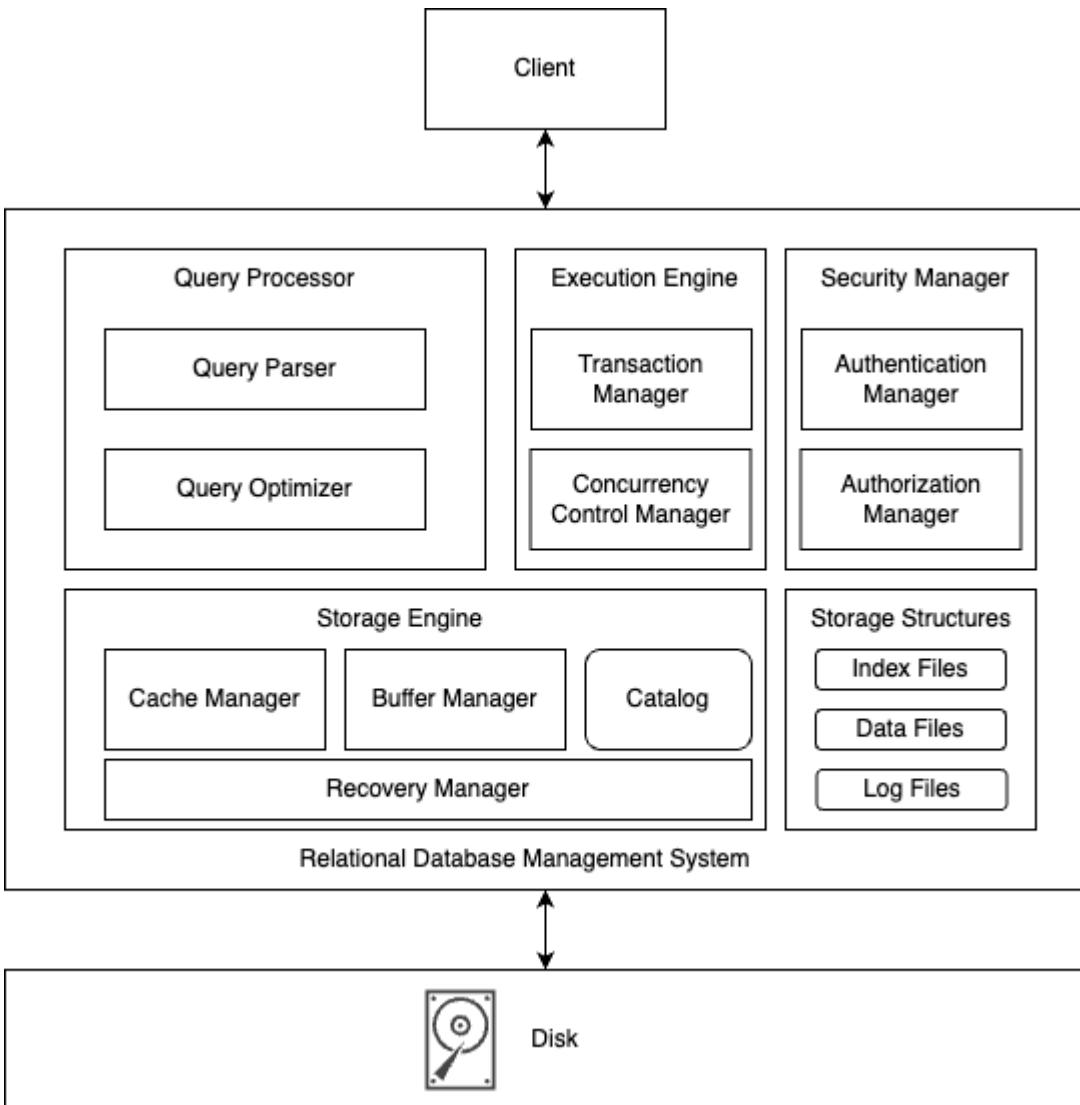
- Primary key: A primary key is a candidate key that has been chosen as the main key for the table. It uniquely identifies each record in the table and is used as a reference by other tables that have relationships with this table. For example, in a table of orders, the Order ID could be the primary key.
- Foreign key: A foreign key is a column or set of columns in one table that refers to the primary key of another table. It is used to establish relationships between tables and ensure data integrity. For example, in a table of orders, there could be a foreign key that references the customer ID in the customers table, to link each order to the customer who ordered it.

In summary, the ER model is a conceptual model used in database design to describe the relationships between entities and their attributes. Schema normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. Keys are used in relational databases to establish relationships between tables, and include candidate keys, primary keys, and foreign keys.

Lets discuss the core components required for a Database Management System architecture. The exact architecture may vary from one implementation to another but generally, the core components remain the same.

## **Relational Database Management System Architecture**

This section discusses Relational Database Management System (RDBMS) architecture, which is shown in [Figure 2-6](#).



*Figure 2-6. Relational Database Management System Architecture Block Diagram*

Let's go over the components in detail one by one.

### *Query Processor*

The query processor takes the user query and translates it into an execution format suitable for underlying execution engine. It has two main submodules - Query Parser and Query Optimizer.

### *Query Parser*

The query processor takes the query given by the user and parses it into Abstract Syntax Tree (AST), which serves as an

intermediate representation of the query for execution. It performs parsing, tokenization, syntax validation, semantic analysis, and tree construction in the process of generating AST.

### *Query Optimizer*

The query optimizer utilizes the Abstract Syntax Tree (AST) generated by the query parser to generate an optimized plan for executing the user's query. By considering internal statistics like data cardinality, placement, and costs associated with local and remote execution, the optimizer evaluates multiple execution plans.

### *Execution Plan*

An execution plan represents a series of steps organized in a directed dependency graph, which must be executed in a specific order to fulfill the user's query. Leveraging the internal statistics, the query optimizer selects the most cost-effective execution plan among the alternatives. Subsequently, it forwards this chosen plan to the execution engine for processing.

### *Execution Engine*

The execution engine is responsible for executing the query plan generated by the query processor. It interacts with the storage engine to retrieve data, performs any necessary joins, filtering, and sorting operations, and returns the result set to the user. The execution engine is also responsible for orchestrating the execution plan across distributed nodes in a distributed relational database.

### *Storage Engine*

The storage engine manages the physical storage and retrieval of data including access and manipulation of data in the database. It handles tasks such as data page management, file allocation,

data compression, and indexing. The storage engine interacts with the execution engine to fetch and store data efficiently.

### *Buffer Manager*

The buffer manager handles the management of data buffers in memory. It controls the movement of data between disk and memory, optimizes disk I/O operations, and ensures that frequently accessed data is kept in memory for faster access. The buffer manager minimizes disk access by caching data pages in memory.

### *Cache Manager*

The cache manager handles the management and optimization of data caching. It stores frequently accessed data in memory to improve query performance by reducing disk I/O operations. The cache manager ensures that the most frequently used data is readily available in memory for faster retrieval.

### *Transaction Manager*

The transaction manager plays a vital role in coordinating and overseeing operations on the data structures within the storage structures module. Its primary responsibility is to ensure that a sequence of operations either executes successfully as a whole or gets rolled back entirely, leaving no partial updates behind.

This crucial guarantee provides end users with the confidence that the database will consistently maintain its integrity before and after executing any database operations. To achieve this, the transaction manager collaborates closely with the concurrency control manager and the recovery manager. By leveraging these components, the transaction manager ensures that the visible data in the database remains consistent and aligns with the expectations of end users.

## *Concurrency Control Manager*

The concurrency control manager handles concurrent access to the database by multiple users or transactions. It ensures that transactions are executed in an isolated and consistent manner, preventing conflicts and maintaining data integrity. It manages locking, transaction isolation levels, and conflict resolution mechanisms.

## *Recovery Manager*

The recovery manager ensures the durability and reliability of the database in the event of failures or crashes. It manages transaction logging, checkpointing, and crash recovery mechanisms to maintain data integrity and consistency. The recovery manager ensures that the database can be restored to a consistent state after a failure.

The recovery manager ensures durability by maintaining an immutable data structure known as a log file within the DBMS. This log file diligently records every write operation applied to the database, ensuring its ability to facilitate recovery processes. Essentially, the recovery manager acts as a reliable and persistent intermediate storage for all write requests.

Each written page is meticulously preserved in the primary memory, designated as a “dirty page.” To ensure durability, the recovery manager batches all the dirty pages and asynchronously synchronizes them with the disk. Once the dirty pages have been successfully flushed to disk, they are considered “clean pages.” In addition, before a write request is acknowledged as completed by the client, it is also appended to a disk-resident append-only data structure. This additional step provides a safeguard against data loss of dirty pages in the event of a crash or system restart.

The log file serves as a valuable checkpoint, allowing the recovery process to restore data for the dirty pages. On system

restart, the operating system takes responsibility for flushing all the remaining dirty pages to disk and discarding any uncommitted transactions. This ensures that the system begins in a clean state, ready to process subsequent writes and reads, without the risk of incomplete or inconsistent data.

### *Security Manager*

The security manager is responsible for enforcing data security and access controls. It authenticates users, manages user permissions and privileges, and ensures data confidentiality, integrity, and availability. The security manager protects the database from unauthorized access and maintains data privacy.

### *Catalog*

The catalog, also known as the data dictionary or metadata repository, stores the metadata about the database schema, tables, columns, indexes, constraints, and other database objects. It provides information about the structure and organization of the database, allowing the RDBMS to interpret and manipulate the data accurately.

These core components work together to provide the necessary functionality for managing and manipulating data in a RDBMS. They ensure efficient query processing, data storage, concurrency control, data recovery, and data security.

In the next section, we will explore the different ways of optimizing query performance in RDBMS using index types, including primary and secondary indexes, SQL tuning, Denormalization and Query Federation.

## **Optimizing Relational Databases**

Optimizing relational databases is an essential task for any database administrator or developer to improve the performance of the

queries that run on top of it. Let's cover various ways to optimize databases in detail.

## **Indexes**

One way to optimize SQL queries is by using indexes. An index is a data structure that improves the speed of data retrieval operations on a database table. By creating indexes on frequently used columns, the optimizer can quickly find the data needed to execute the query, resulting in faster query execution times instead of performing full table scan. Columns that you are querying (SELECT, GROUP BY, ORDER BY, JOIN) could be faster with indices. Indices are usually represented as self-balancing B-tree that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. Following are the two types of indexes, which can be created in a RDBMS:

### *Primary Indexes*

A primary index is an index that is created on a table's primary key. The primary key is a column or set of columns that uniquely identify each row in the table. By creating a primary index on the primary key, the database can quickly find the location of a specific row in the table, resulting in faster data retrieval times.

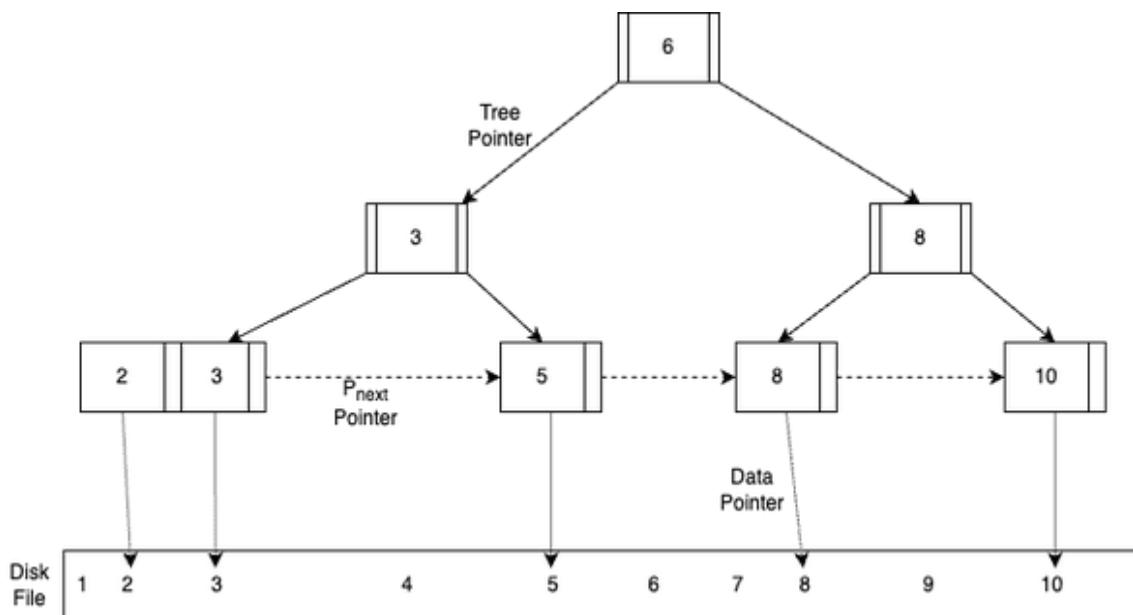
### *Secondary Indexes*

A secondary index is an index that is created on a non-primary key column or set of columns. A secondary index can be used to improve the performance of queries that involve filtering or sorting data based on a specific column or set of columns. For example, if a query frequently filters data based on a customer's phone number, creating a secondary index on the phone number column can significantly improve the query's performance.

B+ trees are widely used in RDBMS as an efficient indexing structure for facilitating fast data retrieval and efficient query processing. The

B+ tree data structure provides efficient key-based searching and range queries on large amounts of data.

A B+ tree, illustrated in [Figure 2-7](#) is a self-balancing tree data structure that stores data in a sorted manner. It consists of internal nodes and leaf nodes, where leaf nodes contain the actual data records or pointers to the data records. Each node in the B+ tree has a fixed number of keys and pointers. The keys in the internal nodes act as separators, guiding the search process, while the leaf nodes store the data records or pointers to the data records in sorted order.



*Figure 2-7. B+ tree data structure representation*

When an index is created on a column, a B+ tree index is constructed, where each node in the tree corresponds to a range of values from the indexed column. The leaf nodes of the B+ tree contain the values of the indexed column along with the corresponding pointers to the actual data records. B+ trees can also be used to create multi-column indexes in RDBMS. In this case, each node in the B+ tree contains multiple keys, allowing for efficient searching and querying based on multiple columns. Multi-column

indexes are beneficial for queries that involve complex conditions or join operations on multiple columns.

Let's understand the effectiveness of B+ tree in different types of queries, scans and operations:

### *Efficient Searching and Range Queries*

B+ trees provide efficient key-based searching and range queries, which are crucial for improving query performance in RDBMS. When a query involves searching for a specific value or a range of values, the B+ tree index allows the database system to quickly navigate the tree to find the desired data records. The balanced nature of the B+ tree ensures that the height of the tree remains relatively small, resulting in faster search operations with logarithmic time complexity.

### *Sequential and Range Scans*

B+ trees also enable efficient sequential and range scans in RDBMS. The leaf nodes of the B+ tree are linked together, allowing for efficient scanning of the entire index or a specific range of values. Sequential or range scans are often performed for queries involving sorting, aggregations, or when a subset of data needs to be retrieved based on specific conditions.

### *Update and Insert Operations*

B+ trees efficiently handle update and insert operations in RDBMS. When a new record is inserted, the B+ tree index is updated in a way that ensures the tree's balance and sorted order are maintained. Similarly, when a record is updated, the corresponding entry in the B+ tree index is adjusted accordingly. This ability to handle dynamic updates and inserts without significant performance degradation makes B+ trees suitable for transactional systems.

However, using a lot of indices comes with its own caveats as placing an index can keep the data in memory, requiring more space. Writes could become slower since the index also needs to be updated. When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

## SQL Tuning

As prescribed under “Prescription of Performance : Metrics Don’t Lie” in Chapter 1, to improve the performance of SQL queries, it is first important to benchmark and profile your queries to uncover bottlenecks. Benchmarking will involve simulating high-load and high-throughput situations, while profiling will require measuring the p9s of the query and analyzing slow query log to identify performance issues. Based on these metrics, SQL tuning involves improving SQL queries to improve their performance by removing the bottlenecks and optimizing query plans.

One way to optimize SQL query is by minimizing large write operations as performing operations such as writing, modifying, deleting, or importing extensive amounts of data can have a significant impact on the performance of queries. These operations may even result in table blocking when tasks involve updating and manipulating data, adding indexes or check constraints to queries, processing triggers, and similar actions. Moreover, the act of writing a substantial volume of data will inevitably lead to an increase in the size of log files.

To optimize SQL performance, another technique is to schedule query execution during off-peak hours. This approach is particularly beneficial when dealing with multiple SELECT queries involving large tables or executing complex queries with nested subqueries, looping queries, and the like. When executing resource-intensive queries in a database, RDBMS applies locks to the tables involved, preventing simultaneous access by different transactions. Consequently, other users are unable to work with those tables, leading to limited access

to specific data. Running heavy queries during peak times not only strains the server but also restricts other users' data access.

Adding multiple tables to a query and performing joins can potentially burden the query and lead to performance issues. Moreover, when dealing with a large number of tables to retrieve data from, it may result in an inefficient execution plan. To achieve efficient query plans, JOIN elimination is one of the many techniques employed. By dividing a single query into several separate queries that can be joined later, unnecessary joins, subqueries, and tables can be eliminated. This approach helps streamline the query and enhance its performance by removing redundant and extraneous elements.

Enhancing RDBMS performance and optimizing SQL queries are critical aspects for both database developers and administrators. They must meticulously evaluate various factors such as the selection of specific operators, the number of tables involved in a query, the query's size, its execution plan, statistics, resource allocation, and other performance metrics. These considerations play a pivotal role in determining whether query performance improves or deteriorates. By carefully analyzing and addressing these factors, developers and administrators can effectively tune and improve query performance.

## **Denormalization**

In most systems, read operations significantly outnumber write operations by a ratio of 100:1 or even 1000:1. Performing a complex database join during a read can be resource-intensive, particularly due to disk operations.

Denormalization is the technique aimed at improving read performance, although it may come at the cost of reduced write performance. It involves duplicating data across multiple tables to avoid costly joins. Some RDBMS offer materialized views, which

handle the task of storing redundant information and maintaining consistency among redundant copies.

When data is distributed through methods such as federation and sharding, managing joins across different data centers adds further complexity. Denormalization can help alleviate the need for complex joins in such scenarios.

However, denormalization also has its disadvantages due to data duplication, leading to redundancy. Maintaining consistency among redundant copies requires the use of constraints, which further, adds complexity to the database design. A denormalized database under heavy write load may perform worse compared to its normalized counterpart.

Overall, denormalization can provide performance benefits for read-heavy workloads, but it introduces trade-offs in terms of data redundancy and increased complexity in database management.

## **Query Federation**

Query federation is a technique that involves splitting a large query into smaller queries that can be executed independently on different database servers. This technique requires schema federation, i.e. functional partitioning of the database across multiple database servers. By executing smaller queries on different servers, the overall query execution time can be reduced, resulting in faster query results. Query federation is useful for optimizing queries that involve large amounts of data or complex joins.

In conclusion, optimizing relational databases using indexes is a critical task for any database administrator or developer. By using these above techniques, database administrators can improve the performance of their databases and provide faster and more efficient access to data. Next, we cover the techniques used to scale relational databases in practice including partitioning, sharding and replication.

## Scaling Relational Databases

As businesses grow and their data needs increase, it becomes necessary to scale their databases to handle the increased workload. Scaling refers to the process of increasing the capacity of a database to accommodate more data, users, and transactions. In this section, we will explore scaling relational databases using partitioning, sharding, and replication.

### Partitioning

Partitioning is the process of dividing a large database table into smaller, more manageable parts called partitions. Each record within the database is assigned to a specific partition, ensuring that every record belongs to one and only one partition. Each partition operates as an independent database, capable of executing read and write operations autonomously. As a result, database queries can be directed towards a single partition to focus on specific data or distributed across multiple partitions for broader processing. The client can either direct the query to a specific partition or to a coordinator node, which then forwards the query to the right set of partitions, thus orchestrating the partitions. There are two types of partitioning as shown in [Figure 2-8](#): vertical and horizontal, with two possible approaches to horizontal partitioning:

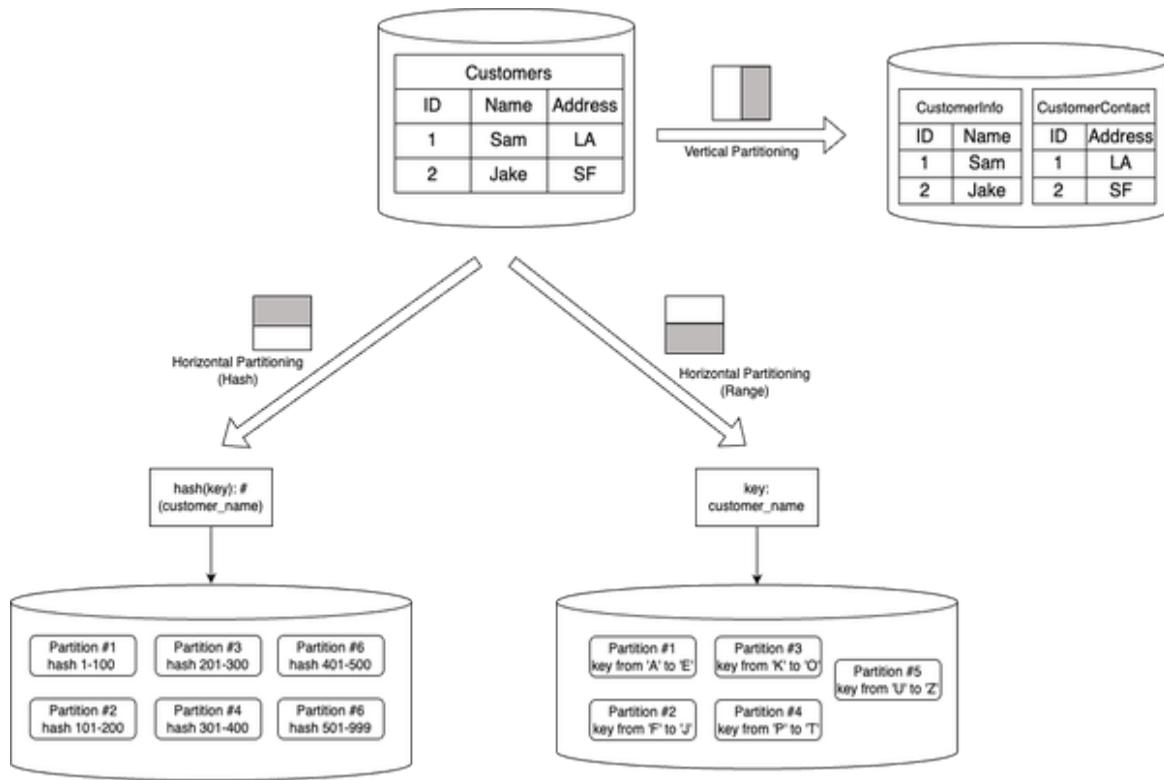


Figure 2-8. Partitioning approaches in database

### *Vertical partitioning*

Vertical partitioning involves splitting a table by columns. For example, a customer table might be split into two tables, one containing customer information and another containing customer contact.

### *Horizontal partitioning*

Horizontal partitioning involves splitting a table by rows. For example, a large customer table might be split into smaller tables, each containing a specific range of customer records based on their last name or zip code. Horizontal partitioning can again have two approaches: Hash Partitioning and Range Partitioning.

### *Hash Partitioning*

The partition by key hash strategy involves generating a hash of the key and evenly distributing it among the partitions. This approach helps avoid data skew and eliminates hot spots within the system.

In this strategy, a hash function (such as MD5 or SHA-256) is applied to the input key of the table. The hash ranges are then divided into buckets, with each bucket assigned to a specific partition. It's important to note that a single host instance can accommodate multiple partitions.

For this strategy to work effectively, the hash function must be deterministic, meaning that the same key should always produce the same hash value and be directed to the same partition. Additionally, each key should resolve to a unique hash value, ensuring a balanced distribution of data across the partitions.

The advantage of hash partitioning is that it eliminates the hot spots and skew partition problem by unique deterministic hashing strategy. However, since the partitioning is on the key's hash, this strategy can't do range queries. We need to query all the partitions, if we want to fetch a set of keys, making the DBMS do a *scatter and gather pattern*.

### *Range Partitioning*

In range partitioning, we partition a continuous range of keys into separate buckets, which are then assigned to specific partitions. It is important to note that a single host instance has the capability to accommodate multiple partitions. The key range assigned to each bucket may or may not be continuous, allowing flexibility in the distribution of data.

Within each partition, the keys are stored in sorted order. This organization of data enables efficient range scan queries, as it simplifies the process of retrieving data within specific key ranges. By maintaining sorted order within each partition, the

system can swiftly access and retrieve data that falls within a given range.

One of the key benefits of range partitioning is that individual partitions store keys in a sorted order, facilitating efficient range scans. For instance, in the figure, it becomes easy to retrieve all customers having email starting with 'P' in lexicographic order. Additionally, the range partitioning strategy is relatively straightforward to implement. However, if the access patterns on partitions are uneven or unfair, certain partitions may end up with more data or queries than others, leading to skewed partitions and placing a heavier load on specific partitions, potentially causing congestion. Such imbalanced partitions, experiencing a disproportionately high workload, are referred to as hot spots. To illustrate, in [Figure 2-8](#), if a significant majority of customers have email beginning with 'A', it can result in the choking of Partition 1 due to an overwhelming amount of data and queries concentrated in that partition.

Partitioning a database offers advantages for both large datasets as well as enables high throughput capability. Partitioning enables the distribution of data across multiple machines via sharding, allowing for the handling of datasets that exceed the capacity of a single machine. By spreading the data across multiple partitions, the database can accommodate and manage large-scale data requirements.

With data distributed across multiple partitions, read and write queries can be processed independently by each partition. This parallel processing capability allows the database to handle a higher overall throughput compared to what a single machine can handle. By leveraging the collective power of multiple machines, partitioning enhances the system's ability to handle concurrent queries and transactions efficiently.

Thus, partitioning can improve query performance by reducing the amount of data that needs to be scanned to execute a query. This results in faster query response times and better scalability.

## Sharding

Sharding is the process of distributing a large database across multiple servers. Each server contains a subset of the data, and queries are distributed across all servers. Sharding is useful for scaling databases that have become too large to be managed on a single server. For example, in a customer database, as the number of customers grows, more shards are added to the cluster to accommodate the increasing load.

There are two types of sharding as shown in [Figure 2-9](#): vertical and horizontal, with different possible approaches to horizontal partitioning, including hash-based sharding, range-based sharding, and round-robin sharding.

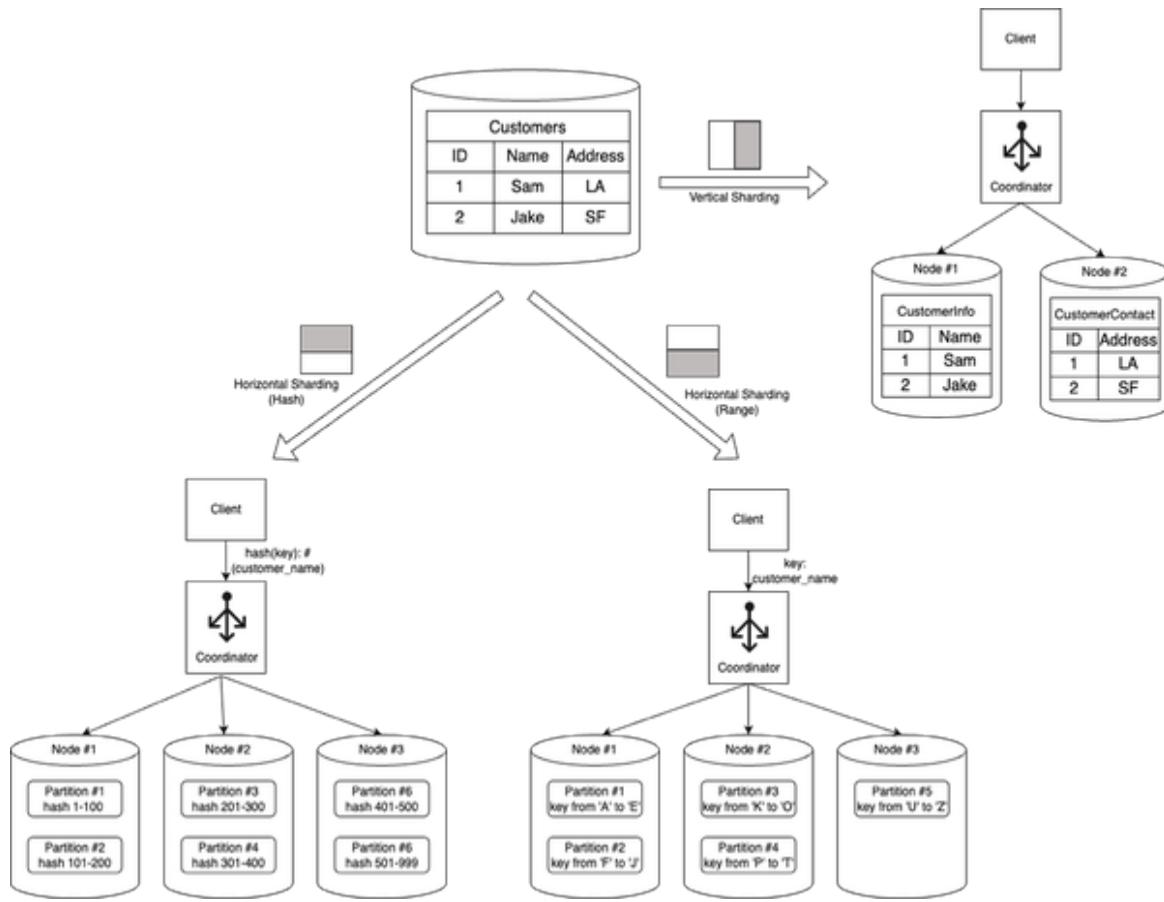


Figure 2-9. Sharding approaches in database

Sharding offers similar advantages to federation, including reduced read and write traffic, decreased replication, and improved cache utilization. It also helps reduce the size of indexes, leading to faster query performance. In the event of a shard failure, the other shards remain operational, although implementing replication is crucial to prevent data loss. Sharding eliminates the need for a central master for write serialization, enabling parallel writes and increasing overall throughput.

Common approaches to shard a customer table include using the customer's last name, initial, or geographic location. However, sharding comes with its disadvantages. Application logic needs to be updated to handle shard-specific operations, potentially resulting in complex SQL queries. Data distribution can become unbalanced, particularly if a shard has a subset of power customers, leading to

increased load on that particular shard. Rebalancing shards adds complexity, although using a consistent hashing-based sharding function can minimize data transfer. Joining data from multiple shards becomes more challenging, and overall, sharding introduces additional hardware requirements and increased system complexity.

## **Replication**

Replication is the process of copying data from one database server to another. Every node that stores a copy of the data is called a replica. Replication is a key feature in distributed databases that offers several advantages, enhancing availability, load distribution, and reducing latency. By maintaining multiple copies of data across different host machines, replication provides the following benefits:

### *High Availability*

Replication ensures high availability by storing data copies on multiple host machines. In the event of a failure or downtime of one host machine, the database can seamlessly redirect read and write operations to other live machines that hold replicated data. This fault tolerance mechanism prevents service disruptions and ensures continuous access to data, enhancing the overall availability of the system.

### *Load Distribution*

With replication, the database can distribute read and write queries across multiple host machines. This load distribution strategy prevents overburdening of individual machines, thereby improving the overall system performance and scalability. By spreading the workload, replication enables efficient utilization of computing resources and better handling of concurrent user requests, resulting in enhanced throughput and responsiveness.

### *Reduced Latency*

Replicating data across geographically distributed host machines allows for placing the replicated copies closer to end users. This proximity reduces the network latency experienced by users when accessing the database. By minimizing the distance between the data and the user, replication improves response times and enhances the user experience, especially in scenarios where low latency is critical, such as real-time applications or distributed systems with users located in different regions.

### *Disaster Recovery and Data Resilience*

Replication serves as a foundation for disaster recovery strategies. By maintaining multiple copies of data, distributed databases can withstand catastrophic events, hardware failures, or natural disasters. In case of data loss or unavailability on one host machine, the replicated copies can be used for data recovery and system restoration. Replication ensures data resilience and minimizes the risk of data loss, contributing to the overall reliability and robustness of the database.

### *Scalability and Performance*

Replication plays a crucial role in scaling distributed databases. As the data size or user load increases, additional host machines can be added to the system, each hosting a replicated copy of the data. This horizontal scaling approach allows the database to handle larger workloads and accommodate growing user demands. By distributing data and queries, replication contributes to improved system performance, enabling efficient parallel processing and reducing response times.

There are two types of replication for RDBMS: Single-leader and Multi-leader, both of which are illustrated in [Figure 2-10](#).

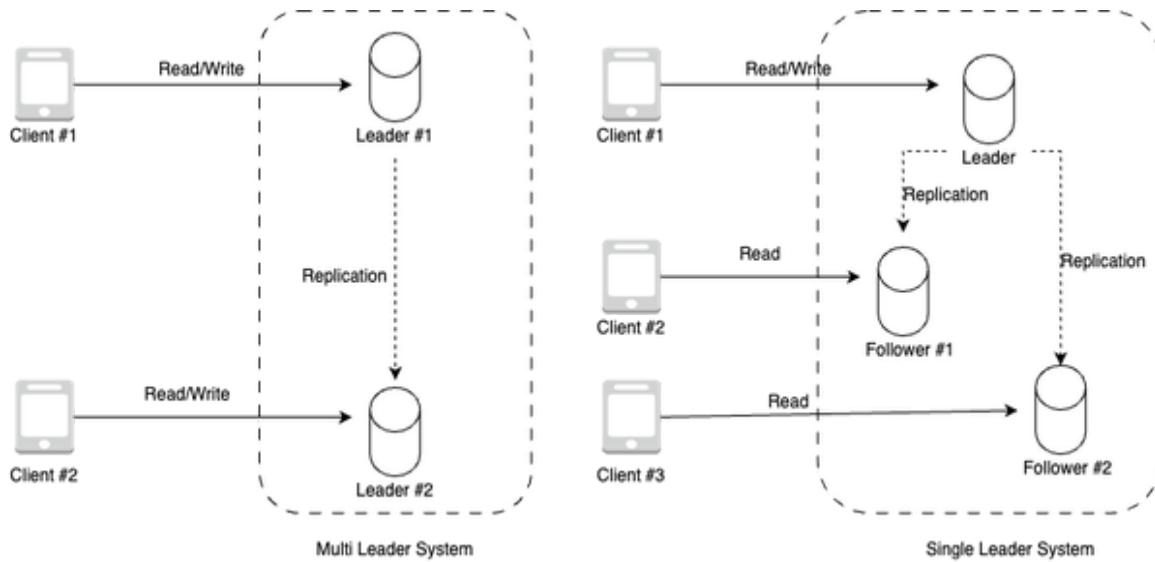


Figure 2-10. Replication approaches in database

### *Single-leader replication*

In single leader replication, one database server serves as the leader, and the other server(s) serve as followers. The leader server handles all write operations, while the follower servers handle read operations. Changes made to the reader server are replicated to the follower servers, ensuring that all servers contain the same data either synchronously or asynchronously. This type of replication is useful for scaling read-heavy databases.

### *Multi-leader replication*

In multi-leader replication, each database server can both read and write data. Changes made to one server are replicated to the other server either synchronously or asynchronously, ensuring that all servers contain the same data. This type of replication is useful for scaling databases that require high availability.

For the above replication types, replication can be done on follower nodes either as full replication, snapshot based replication, transactional replication (i.e. replicating the transactional updates) or

key based incremental replication (i.e. scanning the database for modified keys and only replicating data for those keys). The replication can be done either synchronously or asynchronously, each having its own caveats.

In distributed databases, synchronous replication provides a mechanism for replicating data from a leader replica to follower replicas using synchronous communication, keeping data always up-to-date across all replicas. By maintaining synchronous replicas that are always in sync with the leader replica, this replication mechanism offers several advantages:

### *Consistent Writes*

With synchronous replication, the client considers a write operation successful only when it is acknowledged by both the leader replica and the synchronous follower replica. This strict synchronization ensures that the data remains consistent across replicas. By waiting for confirmation from both replicas, the system guarantees that the write has been durably committed and is available for subsequent read operations. This consistency is vital for applications that require strict data integrity and accuracy.

### *Immediate Failover*

One significant advantage of synchronous replication is its ability to facilitate immediate failover in the event of a leader replica crash. Since the synchronous replicas are always up-to-date with the leader, any of these replicas can be immediately promoted as the new leader without any data loss. This seamless transition ensures continuous availability of the system even in the face of failures. By eliminating downtime and minimizing data loss, synchronous replication enhances the overall resilience and reliability of the distributed system.

### *Data Durability*

Synchronous replicas guarantee that data is durably stored across multiple replicas. When a write operation is confirmed by both the leader and synchronous follower replicas, it ensures that the data is safely persisted on multiple machines. This redundancy provides data durability, protecting against data loss in the event of a replica failure or system crash. By maintaining multiple synchronized copies, synchronous replication ensures that critical data remains intact and recoverable, enhancing the overall data resilience of the system.

### *Consistency in Read Operations*

Synchronous replication not only ensures consistency in write operations but also in read operations. As the synchronous replicas are always up-to-date with the leader, read operations can be performed on any of the replicas with the guarantee of accessing the most recent and consistent data. This feature enables load balancing and improves the system's ability to handle read-intensive workloads.

In summary, synchronous replication in distributed databases provides the benefits of consistent writes, immediate failover, data durability, and consistency in read operations. By maintaining synchronous replicas that are always in sync with the leader, this replication mechanism ensures data integrity, continuous availability, and reliable access to up-to-date information. Synchronous replication is particularly valuable in scenarios where strict consistency and high availability are paramount, such as in financial systems, real-time applications, and mission-critical environments.

In distributed databases, asynchronous replication provides a mechanism for replicating data from a leader replica to asynchronous replicas. This replication approach offers specific characteristics and considerations:

### *Near Real-Time Updates*

Asynchronous replicas apply changes from the leader replica in near real-time, albeit with a potential delay. This means that the asynchronous replicas may not be immediately up-to-date with the latest changes made on the leader. The time lag between the leader and asynchronous replicas results in a temporary inconsistency in data across the replicas. However, over time, the asynchronous replicas converge as they catch up with the leader's updates.

### *Data Lag and Potential Staleness*

Due to the asynchronous nature of replication, the asynchronous replicas can lag behind the leader replica. The extent of this lag depends on various factors such as network conditions, system load, and the volume of changes being replicated. As a result, the asynchronous replicas may not reflect the most recent state of the data. This potential staleness can impact applications or use cases that require access to the most up-to-date information. It's essential to consider this trade-off between near real-time updates and potential data lag when employing asynchronous replication.

### *Risk of Data Loss*

Because asynchronous replicas can be lagging or stale, promoting an asynchronous replica as the new leader in the event of a leader replica crash can introduce the risk of data loss. Since the asynchronous replica may not have received or applied all the changes made on the previous leader, promoting it prematurely can result in missing or inconsistent data. To mitigate this risk, careful considerations and measures, such as monitoring the replication lag and ensuring that the asynchronous replica has caught up sufficiently, should be taken before promoting it as the new leader.

### *Scalability and Performance*

Asynchronous replication is often favored in distributed systems that prioritize scalability and performance over strict consistency. By allowing some flexibility in the replication process and tolerating temporary data inconsistencies, asynchronous replication can handle high write throughput and accommodate systems with large-scale deployments. This approach enables the system to distribute the workload and scale horizontally while maintaining acceptable response times.

In summary, asynchronous replication in distributed databases provides near real-time updates but introduces the possibility of data lag and potential staleness. While it offers scalability and performance advantages, the risk of data loss exists when promoting an asynchronous replica as a new leader. Careful planning, monitoring, and appropriate fallback mechanisms are necessary to ensure data integrity and minimize the impact of data inconsistencies when employing asynchronous replication.

Replication in distributed databases offers numerous benefits, including high availability, load distribution, reduced latency, disaster recovery capabilities, scalability, and enhanced performance. By leveraging replication strategies, organizations can ensure robustness, fault tolerance, and optimized access to data in distributed environments.

Scaling distributed relational databases is an important task for businesses that need to handle increasing amounts of data and traffic. Partitioning, sharding, and replication are all useful techniques for scaling databases. Partitioning can improve query performance by reducing the amount of data that needs to be scanned to execute a query. Sharding is useful for scaling databases that have become too large to be managed on a single server. Replication is useful for scaling databases that require high availability or have heavy read traffic.

To conclude the chapter, let's go over the most popular open-source RDBMS choices in the next section.

## **Open-source Relational Database Systems**

Open-source RDBMS are popular choices for businesses and developers due to their affordability, flexibility, and community support. Two widely used open-source RDBMS are MySQL and PostgreSQL. In this section, we will compare and contrast MySQL and PostgreSQL and explore their similarities and differences.

### *MySQL*

**MySQL** is an open-source RDBMS that was first released in 1995. It is one of the most popular RDBMS in the world, with a large user community and a broad range of features. MySQL is known for its speed and scalability, making it an ideal choice for businesses that need to handle high volumes of data and traffic.

Features of MySQL include:

- Supports SQL and non-SQL queries
- Scalable architecture
- High performance
- Simple and easy to use
- Good for read-heavy workloads
- Strong support for clustering and replication

### *PostgreSQL*

**PostgreSQL** is an open-source RDBMS that was first released in 1996. It is known for its robustness, reliability, and feature-richness. PostgreSQL is a popular choice for businesses that require advanced data management capabilities and powerful query processing.

Features of PostgreSQL include:

- Supports SQL and non-SQL queries
- ACID-compliant transactions
- Strong support for stored procedures and triggers
- Supports JSON and XML data types
- Good for write-heavy workloads
- Offers advanced data management features

Both MySQL and PostgreSQL, as compared in Table 2-1, are open-source RDBMS, are ACID-compliant and support SQL and non-SQL queries. Both MySQL and PostgreSQL offer strong support for replication and clustering. However, MySQL has a simpler and more straightforward syntax, while PostgreSQL offers more advanced features and capabilities. MySQL is better suited for read-heavy workloads, while PostgreSQL is better suited for write-heavy workloads.

*T  
a  
b  
/e  
2  
-  
1  
.M  
y  
S  
Q  
L  
v  
s  
P  
o  
s  
t  
g  
r  
e  
S  
Q  
L*

Property

MySQL

PostgreSQL

---

Introduction	MySQL is open-source RDBMS.	PostgreSQL is an object-relational database management system (ORDBMS). It provides all the facilities of RDBMS with additional support of object oriented concepts like classes, objects and inheritance.
Datatype Support	Supports standard SQL types.	Supports the standard SQL types along with many advanced types such as array, jsonb, and user-defined type.

JSON Support	<p>Supports JSON documents to be stored in a column by converting to an internal format that permits quick read access to document elements.</p> <p>JSON columns cannot be indexed directly. To create an index that references such a column indirectly, you can define a generated column that extracts</p>	<p>Support two data types related to json.</p> <p><b>Json:</b> This stores an exact copy of the input text which processing functions must reparse on each execution.</p> <p><b>Jsonb:</b> This is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster</p>
--------------	---	---

the information that should be indexed, then create an index on the generated column. to process, since no reparsing is needed. It also supports indexing which can be a big advantage.

## Indexes

Types of indexes include primary key, foreign key, unique Index, single column, multi column index(upto 16 columns). spatial indexes.

Types of indexes include primary key, foreign key unique index, single column, multi column index(upto 32 columns) Besides that, it also support following two types.

Expression indexes: can be created with an index of the result of an expression or function, instead of simply the value of a column.

Partial indexes: index only a part of a table.

## Replication

MySQL replication is one-way asynchronous

PostgreSQL has synchronous replication

replication where one server acts as a master and others as slaves. You can replicate all databases, selected databases or even selected tables within a database.

(called 2-safe replication), that utilizes two database instances running simultaneously where the master database is synchronized with a slave database. Unless both databases crash simultaneously, data won't be lost.

Performance	MySQL implements concurrent connections by spawning a thread-per-connection. This is relatively low overhead.	Postgres, however, uses a process-per-connection design. This is significantly more expensive than a thread-per-connection design. Postgres also seems to have poor support for handling large connection counts even when there is sufficient memory available.
-------------	---	--

Speed	By not including certain SQL features, MySQL stays light to prioritize speed and reliability. MySQL's speed is specially apparent, when it	Postgres supports query plans that can leverage multiple CPUs in order to answer queries with greater speed. This, coupled with strong support for multiple
-------	--	---

comes to highly concurrent read-only operations.

concurrent writes, makes it a great choice for complex operations like data warehousing and online transaction processing (OLTP).

In conclusion, both MySQL and PostgreSQL are powerful open-source RDBMS that offer a broad range of features and capabilities. MySQL is known for its speed and scalability, while PostgreSQL is known for its robustness and advanced data management capabilities. When choosing between MySQL and PostgreSQL, it's important to consider the specific needs of your business and the requirements of your application.

#### NOTE

AWS Relational Database Services (RDS) offers multiple options by providing various database engine versions (commonly referred to as "flavors"), instance classes, and storage types. These flavors are essentially different configurations of managed database engines, each optimized for specific use cases and workloads. AWS RDS supports several popular relational database engines, including MySQL, PostgreSQL, Oracle, SQL Server, and MariaDB. We will cover AWS RDS in more detail in Chapter 10 - AWS Storage Services.

## Conclusion

This chapter has provided a comprehensive exploration of relational databases and their underlying concepts, architecture, and strategies for scalability. We began by delving into various types of storage mechanisms, including file, block, and object stores, laying the foundation for understanding how data is managed and accessed.

From there, we transitioned into a thorough examination of relational databases, elucidating the key principles that govern their design and operation.

The focal point of the chapter revolved around addressing the challenges of database scalability. We uncovered a range of advanced techniques, such as partitioning, indexing, replication, federation, sharding, and denormalization, each offering unique solutions to accommodate growing datasets and increasing user demands. By exploring these strategies, you have gained valuable insights into how to effectively optimize performance and maintain the integrity of relational databases as their usage expands.

To further enrich your understanding, we introduced two prominent open source databases, MySQL and PostgreSQL, allowing you to acquaint yourselves with practical implementations of the concepts covered. These databases serve as powerful tools, showcasing the real-world application of the theoretical knowledge presented in the chapter.

As we conclude this chapter on relational databases, we open the door to the exciting realm of non-relational databases. The next chapter will embark you on an exploration of alternative database paradigms that have gained prominence in recent years. Non-relational databases, with their diverse models and unique features, present a compelling alternative to traditional relational systems. By delving into this subject, you will broaden your horizons and deepen your understanding of the evolving landscape of database technologies.

# Chapter 3. Non-relational Stores

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

In the ever-evolving landscape of modern data management, traditional relational databases have long been the stalwarts of structured data storage and retrieval. However, the rise of new technologies, diverse data formats, and the need for high-performance, scalable solutions has given birth to a new class of databases known as non-relational or NoSQL databases. These databases have gained popularity for their ability to handle the challenges posed by today's data-intensive applications and their capacity to adapt to various data models, while offering unprecedented scalability and performance.

Traditionally, relational databases have been the cornerstone of data management, providing a structured and standardized approach to organizing and retrieving information. However, as the digital

universe continues to expand exponentially, traditional relational models face inherent limitations when confronted with the demands of modern web applications, real-time analytics, and large-scale distributed systems.

Non-relational databases break away from the rigid constraints of the relational paradigm and introduce novel data models and storage mechanisms that challenge the status quo. These databases prioritize scalability, fault tolerance, and low-latency access, enabling organizations to effectively handle massive volumes of data and support dynamic, rapidly evolving data requirements.

In this chapter, we dive deep into the realm of non-relational databases. We will unravel the fundamental principles, key concepts, and various types of NoSQL databases, shedding light on their strengths, weaknesses, and ideal use cases. Our exploration will encompass a range of popular NoSQL database models, including document stores, key-value stores, columnar databases, and graph databases, each tailored to address specific data management challenges. We'll also discuss their architecture and how leaderless architecture supports scaling using quorum, optimistic replication, consistent hashing, and hinted hand-off.

Throughout this chapter, we will delve into the intricacies of data modeling, query languages, and consistency models associated with non-relational databases. We will examine the core architectural principles that underpin their design, such as horizontal scalability, distributed data storage, and decentralized control. Additionally, we will explore the trade-offs that come with embracing non-relational databases, including data consistency, data integrity, and the impact on application development.

By the end of this chapter, you will have gained a comprehensive understanding of non-relational databases, empowering you to make informed decisions when choosing the right data storage solution for your organization's unique needs. Whether you are a seasoned data

professional or a curious enthusiast, prepare to embrace the world of non-relational databases and discover the remarkable possibilities they offer in the era of big data and beyond.

### NOTE

AWS offers a range of non-relational database services, such as Amazon DynamoDB, Amazon DocumentDB, Amazon Neptune, Amazon ElasticCache, Amazon OpenSearch, Amazon Keyspaces and more, to meet customer requirements for different business use-cases, which we will cover in more detail in Chapter 10 - AWS Storage Services.

Let's cover basic concepts in the next section around non-relational databases including BASE, schema-less design, horizontal scalability, high availability etc, which form the foundation of modern non-relational databases.

## Non-relational Database Concepts

Some of the fundamental concepts that characterize non-relational databases include:

### *Schema Flexibility*

Unlike relational databases that enforce a fixed schema, non-relational databases allow for dynamic and flexible schema design. This means that data can be stored without the need to pre-define a strict structure, making it easier to accommodate varying data formats and evolving application requirements.

### *Data Models*

Non-relational databases support various data models, each optimized for specific use cases. The most common data models include:

## *Document Stores*

These databases store and manage data in flexible, semi-structured documents, often in formats like JSON or BSON. This approach is ideal for applications dealing with complex or dynamic data.

## *Key-Value Stores*

These databases store data in a simple key-value format, making them highly efficient for read and write operations. They are commonly used for caching and high-speed data retrieval.

## *Column-Family Stores*

These databases organize data into column families, allowing efficient storage and retrieval of large volumes of data, especially in analytical and data warehousing scenarios.

## *Graph Databases*

Graph databases focus on relationships between data entities, making them well-suited for applications that require complex querying and analysis of interconnected data.

## *Scalability*

Non-relational databases are designed to scale horizontally, distributing data across multiple nodes or servers. This enables them to handle massive datasets and high levels of concurrent traffic. Scaling is achieved through techniques like sharding and replication, allowing applications to grow seamlessly as demand increases.

## *High Availability and Fault Tolerance*

Many non-relational databases prioritize high availability and fault tolerance. They are often designed to handle hardware failures, network partitions, and other disruptions without compromising data integrity or accessibility.

## *BASE*

In the realm of non-relational databases, the acronym BASE stands for Basically Available, Soft state, Eventually consistent. BASE is an alternative approach to data consistency and availability as per CAP theorem, compared to the ACID (Atomicity, Consistency, Isolation, Durability) properties traditionally associated with relational databases. BASE is particularly relevant in distributed and large-scale systems where high availability and scalability are key requirements, often at the expense of strict strong consistency.

### **NOTE**

Please refer to Chapter 1 - System Design Tradeoffs and Guidelines to read about CAP theorem, a trade-off which naturally arises in distributed systems. Also, refer to Chapter 2 - Storage Types and Relational Stores to read about ACID in more detail, which is fundamental property in relational databases.

## *Basically Available*

The “Basically Available” aspect of BASE emphasizes that a non-relational database should always ensure some level of availability, even in the face of network partitions, hardware failures, or other issues. This means that the database system is designed to respond to client requests even if it means sacrificing some level of consistency in the data. Availability is a critical requirement for modern applications that cannot afford downtime or unresponsiveness.

## *Soft State*

In a non-relational database system following the BASE principles, the concept of “Soft State” is adopted. Soft state implies that the state of the system can change over time due to various factors, such as node failures, network delays, or concurrent updates. This contrasts with the traditional ACID principle of maintaining a hard or rigid state consistency at all times. In a BASE system, it’s acknowledged that the system’s state might be fluid, and applications need to be designed to handle such variability gracefully.

## *Eventually Consistent*

Perhaps the most distinct characteristic of BASE is the principle of “Eventually Consistent.” Unlike ACID, which enforces strict consistency at all times, BASE allows for a temporary lack of consistency between replicas in a distributed system. In other words, it recognizes that updates to the database may take some time to propagate across all nodes, and there might be a brief period during which different nodes may have slightly different views of the data. However, over time, as the system resolves conflicts and synchronizes updates, the data will eventually converge to a consistent state.

BASE principles are particularly suitable for scenarios where rapid scalability and high availability are paramount, such as in modern web applications, real-time analytics, and large-scale data processing. It’s important to note that while ACID properties provide strong guarantees of data integrity and consistency, they can potentially hinder performance and scalability in distributed environments. BASE, on the other hand, offers a trade-off between consistency and availability, allowing applications to maintain responsiveness and continue functioning in the face of network partitions or hardware failures.

It's worth mentioning that the choice between ACID and BASE depends on the specific requirements of an application. Some applications, like financial systems or traditional relational databases, may prioritize strict consistency provided by ACID, hence the need to use relational databases. Meanwhile, others, like social media platforms or content delivery networks, may benefit more from the availability and scalability offered by BASE principles.

Let's go over the different types of non-relational databases based on data models discussed above in detail, starting with key value databases.

## **Key Value Databases**

Key-value databases are a type of non-relational database that offer a simple yet powerful data model, enabling efficient storage and retrieval of data based on unique keys using distributed hash tables. This design principle, which revolves around mapping keys to their corresponding values, provides a highly scalable and flexible approach to data management. In this section, we will delve into the architecture and design considerations of key-value stores, uncovering the inner workings of these streamlined databases.

### **Data Model**

The fundamental concept of a key-value store revolves around a basic data structure - the key-value pair. The data is stored in tables, similar to RDBMS but the table abstraction is built over the key value store. Each key is associated with a corresponding value, forming a unit of data storage, called item. The keys are typically unique within the database and are used to access and retrieve their associated values. The values in a key-value store can be of various types, including strings, numbers, binary data, or even complex structures like JSON objects. Let's go over the data model characteristics of

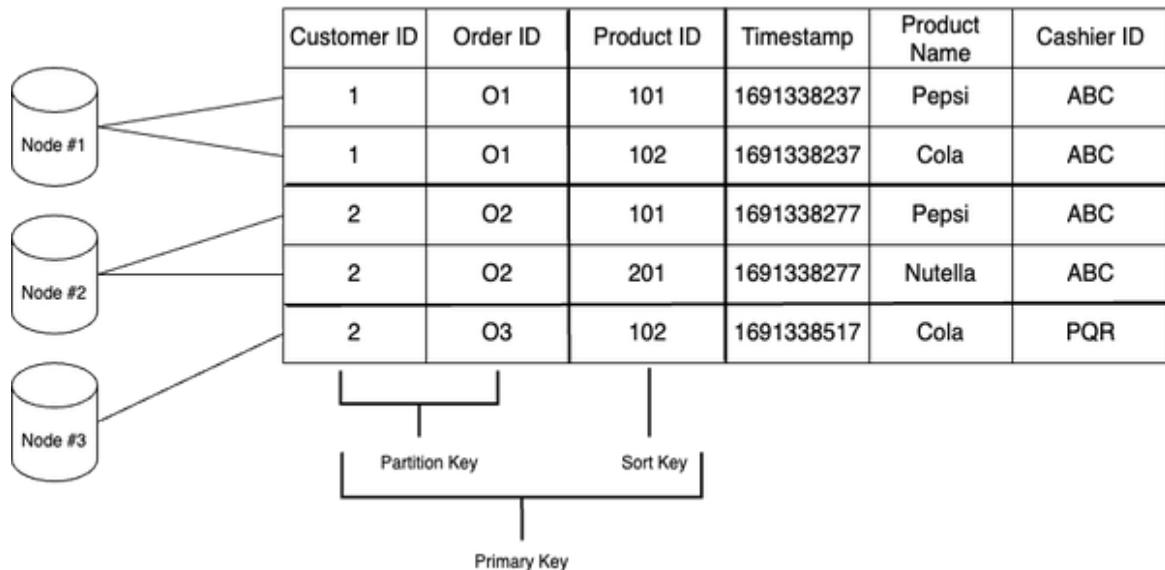
key-value stores with respect to schemaless design and key implementation.

## No Fixed Schema or Indexes

Unlike traditional relational databases, key-value stores do not enforce a fixed schema. This means that different key-value pairs can have varying structures, allowing for dynamic data models. So, some item records can have attributes that other records don't have. Additionally, key-value stores typically lack complex indexing mechanisms, as they rely heavily on the efficiency of key-based lookups for data retrieval. Indexing, if present, is often limited to the keys themselves, optimizing the process of locating the desired values.

## Keys

In a key-value store, the primary key, partition key, and sort key are fundamental concepts that help organize and retrieve data efficiently, as shown in [Figure 3-1](#). Let's explore each of these concepts in more detail:



*Figure 3-1. Keys in Key Value Store*

## Primary Key

The primary key is a unique identifier associated with each key-value pair in the key-value store. It serves as the primary means of accessing and retrieving data. The primary key provides a direct mapping to the corresponding value, allowing for fast retrieval operations. Typically, the primary key is a simple data element such as a string, integer, or a combination of multiple fields. By ensuring uniqueness, the primary key guarantees that each key-value pair is uniquely identifiable within the store.

### *Partition Key*

The partition key is a subset of the primary key and is responsible for data distribution across multiple storage partitions in a distributed key-value store. It determines the storage location of the data within the store. The partition key is used to partition the data set into smaller, manageable subsets that can be distributed across different physical or virtual storage nodes. Each partition operates independently, allowing for horizontal scaling and improved performance. By selecting an appropriate partition key, data can be evenly distributed and evenly balanced across the partitions, avoiding hotspots and ensuring efficient storage and retrieval operations.

### *Sort Key*

The sort key, also known as a range key, is an optional attribute used to order or sort the data within a partition. While the primary key uniquely identifies each key-value pair, the sort key allows for efficient range queries and data retrieval in a specific order. The sort key is particularly useful when you want to query a range of data based on a specific criterion, such as retrieving all records with timestamps within a certain range or retrieving data in alphabetical or numerical order. By organizing data based on the sort key, the key-value store can perform efficient range-based queries and optimize retrieval performance.

In summary, the primary key uniquely identifies each key-value pair in a key-value store. The partition key determines the data distribution across multiple storage partitions, enabling horizontal scalability and load balancing. The sort key, while optional, facilitates efficient sorting and range-based queries within each partition. Understanding and appropriately utilizing these key concepts in a key-value store helps optimize data storage, retrieval, and query performance.

## **Data Access and Retrieval Operations**

In a key-value store, several operations are commonly used to manipulate data associated with specific keys. These operations include:

### *GetItem*

The GetItem operation retrieves the attributes of an item associated with a given primary key. By specifying the primary key, the GetItem operation returns the set of attributes that describe the item. If no item exists for the provided key, the GetItem operation returns an empty result.

### *PutItem*

The PutItem operation is used to insert an item into the store. If there is no existing item with the specified key, the operation creates a new item and associates it with the key. If an item already exists for the given key, the PutItem operation replaces the existing item with the new item.

### *UpdateItem*

The UpdateItem operation is employed to modify an existing item. If an item with the specified key exists, the operation updates the attributes of that item. It allows adding new attributes, modifying existing ones, or removing attributes from

the item. However, if no item exists for the given key, the UpdateItem operation adds a new item instead.

### *DeleteItem*

The DeleteItem operation is used to remove an item from the key-value store. By providing the primary key of the item, the DeleteItem operation deletes the corresponding item from the store. If no item exists for the specified key, no action is taken.

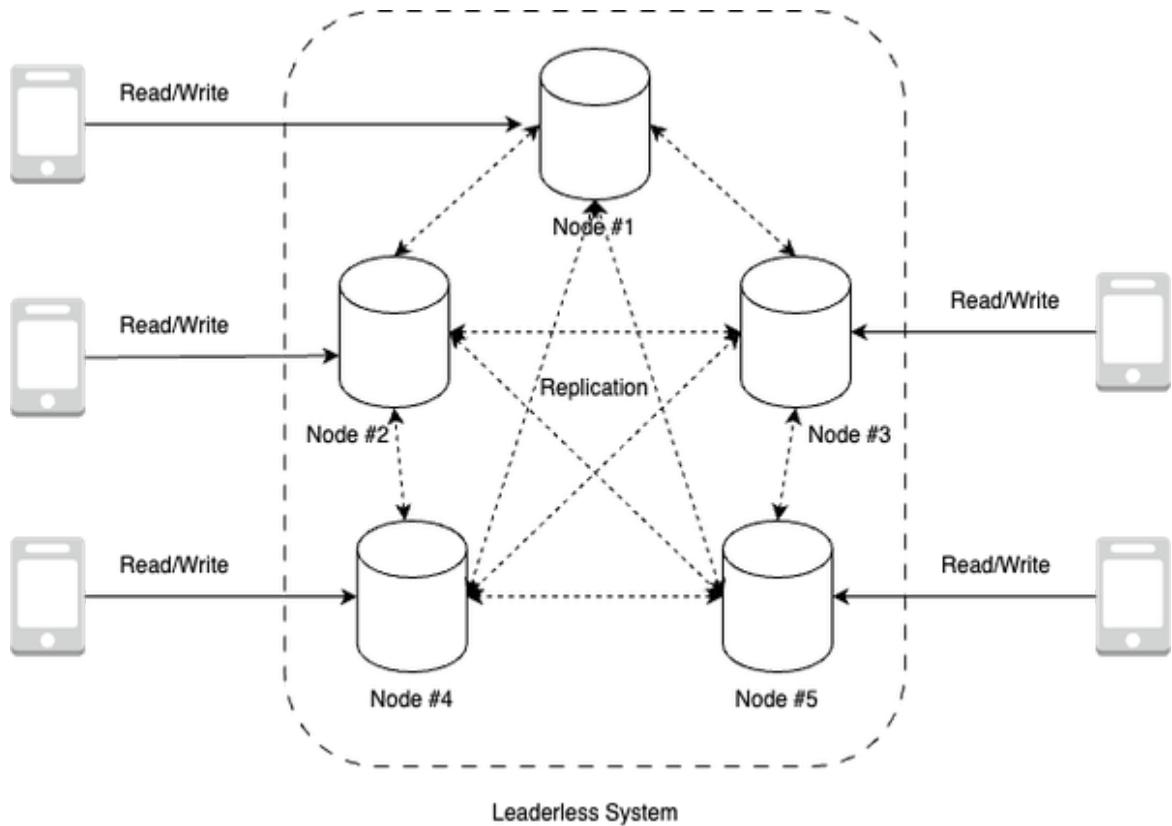
These operations form the basic set of functions used to interact with a key-value store, allowing for item insertion, update, deletion, and retrieval based on their associated keys. By utilizing these operations, developers can effectively manage and manipulate data within a key-value store.

## **Scaling Key Value Stores**

Key-value stores are designed to handle massive volumes of data and support horizontal scalability. To achieve this, they often adopt a distributed architecture, where data is partitioned across multiple nodes or servers. Each node is responsible for storing a subset of the data, allowing the system to handle high loads and provide fault tolerance. Distributed key-value stores employ techniques such as leaderless replication using consistent hashing or partitioning algorithms to evenly distribute the data across the nodes. Let's go through the leaderless replication technique in detail.

## **Leaderless Replication**

Leaderless replication is a technique used in distributed systems, including key-value stores, to achieve high availability and fault tolerance without relying on a single designated leader node. Instead of having a dedicated leader responsible for coordinating read and write operations, all nodes in the system, as shown in [Figure 3-2](#) are equal and can accept client requests independently.



*Figure 3-2. Leaderless System Setup*

In leaderless replication, data is typically divided into smaller partitions or shards, and each shard is replicated across multiple nodes in the system using peer to peer replication. This replication ensures data redundancy and fault tolerance. When a client wants to read or write data, it can send the request to any node in the system without needing to know which node is the leader. Each node has the capability to handle read and write operations, eliminating the bottleneck and single point of failure associated with a dedicated leader.

To ensure consistency, leaderless replication often employs techniques such as quorums and conflict resolution mechanisms. Quorums define the minimum number of successful responses required for an operation to be considered successful. For example, a quorum of  $N/2+1$  nodes may be required to acknowledge a write operation to ensure that a majority agrees on the state change.

Conflict resolution mechanisms such as vector clocks and Merkle trees help resolve conflicts that may arise when different nodes receive conflicting updates for the same data. Merkle trees assist in conflict resolution by efficiently identifying the specific differences between two versions of a dataset. By comparing the root hashes of these trees, one can pinpoint the exact branches where discrepancies occur, simplifying the process of identifying and reconciling conflicting data.

Leaderless replication provides high availability and fault tolerance because the system can continue operating even if some nodes become unavailable. Clients can send requests to any available node, and the system handles the replication and coordination transparently. However, ensuring strong consistency across the replicas can be more challenging in a leaderless replication model compared to systems with a designated leader. Hence, Amazon DynamoDB is based on leader-follower architecture, which we discussed in Chapter 1.

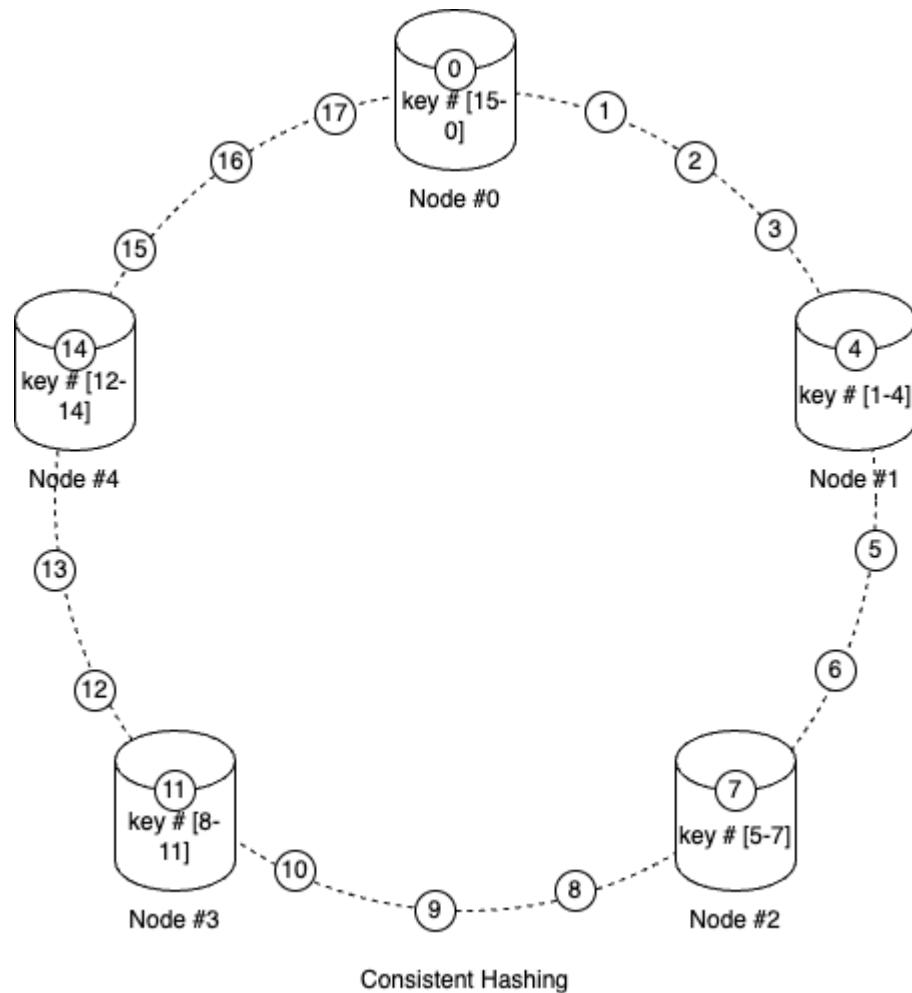
## **Consistent Hashing**

Consistent hashing is a hashing technique commonly used in distributed systems, including key-value stores, to distribute data across a set of nodes while minimizing the impact of node additions or removals on the overall data distribution.

In traditional hashing, data is hashed to determine the node responsible for storing or serving it. However, this approach becomes problematic when nodes are added or removed from the system. In such cases, the distribution of data needs to be recalculated, resulting in significant data movement and potential disruptions.

Consistent hashing, as shown in [Figure 3-3](#) addresses this issue by introducing a ring-like structure that represents the set of nodes in the system. Each node is assigned a position on the ring using a hash function. Data is also hashed to a position on the ring. The

node whose position is the closest clockwise to the data's position becomes responsible for storing or serving that data.



*Figure 3-3. Consistent Hashing Implementation*

The key advantage of consistent hashing is that when a node is added or removed, only a fraction of the data needs to be remapped to new nodes. Most of the data remains assigned to the same nodes as before, minimizing the data movement and disruption in the system. This makes consistent hashing highly scalable and efficient in distributed environments.

Furthermore, consistent hashing provides load balancing among nodes since data is evenly distributed across the ring, and nodes tend to have a similar number of data partitions assigned to them. It

also allows for easy scaling by adding or removing nodes, as the redistribution of data is limited to the affected partitions.

In summary, consistent hashing is a technique that enables efficient and scalable data distribution in distributed systems. It minimizes the impact of node additions or removals by requiring only a fraction of the data to be remapped, ensuring high availability, load balancing, and easy scalability in key-value stores and other distributed systems.

## **Availability in Key-Value Stores**

In a distributed key-value store, ensuring high availability is crucial to maintaining system responsiveness and reliability. Several mechanisms are commonly employed to achieve availability in such environments. Let's explore some of these mechanisms:

### *Optimistic Replication*

Optimistic replication is a technique used to provide availability in the presence of network partitions or temporary failures. In this approach, multiple replicas of data are maintained across different nodes in the system. When a write operation occurs, the changes are propagated asynchronously to the replicas. Instead of waiting for acknowledgments from all replicas before considering the write operation successful, the system assumes success and continues processing. This optimistic approach reduces latency and allows for continued availability even if some replicas are temporarily unavailable. However, it introduces the possibility of temporary inconsistencies between replicas, which are eventually resolved through background synchronization processes.

### *Sloppy Quorum and Last Write Wins*

Sloppy quorum and Last Write Wins (LWW) are techniques used to achieve availability and eventual consistency in the face of

network partitions or replica unavailability. In sloppy quorum, a subset of replicas, rather than the full set, is required to acknowledge a read or write operation. By allowing a relaxed quorum requirement, the system can continue operating even if some replicas are unavailable. However, this relaxation can result in temporary inconsistencies between replicas. To resolve conflicts in write operations, the Last Write Wins (LWW) strategy is often employed. In LWW, if multiple replicas receive conflicting updates for the same key, the update with the latest timestamp (according to a predefined ordering mechanism) takes precedence. This approach sacrifices strong consistency for availability, ensuring that the most recent write is eventually propagated to all replicas.

### *Hinted Handoff*

Hinted Handoff is a mechanism used to handle temporary unavailability or network partitions in a distributed key-value store. When a write operation is performed on a replica that cannot immediately communicate with the primary replica, the write is temporarily stored in the form of a "hint." These hints are later delivered to the appropriate replica once it becomes available again. This approach allows for continued availability of write operations, even in the presence of temporary replica failures. By leveraging hinted handoff, the system ensures that no data is lost and that all updates are eventually applied to the appropriate replicas.

These availability mechanisms, including optimistic replication, sloppy quorum, LWW, and hinted handoff, enable key-value stores to provide high availability in distributed environments. By balancing the trade-offs between availability and consistency, these mechanisms ensure that the system remains responsive and resilient, even in the face of network partitions, temporary failures, or replica unavailability.

## **Advantages, Trade-offs and Considerations**

One of the primary advantages of key-value stores is their ability to deliver high performance and low-latency access to data. Since the data is accessed directly using a key, the retrieval process is highly efficient and typically involves minimal processing overhead. Key-value stores excel in use cases where rapid read and write operations are essential, such as caching, session management, and real-time data processing.

While key-value stores offer remarkable scalability and performance benefits, they do come with certain trade-offs. One of the major considerations is the limited querying capabilities. Key-value stores are optimized for simple key-based lookups and lack the advanced querying capabilities provided by relational databases. Moreover, transactions and complex operations involving multiple keys can be challenging to implement in a distributed key-value store.

Key-value stores find extensive use in various domains, including web applications, distributed systems, and caching layers. They excel in scenarios where fast and direct access to data is crucial, such as user session management, user profiles, product catalogs, and real-time analytics.

## **Open Source Key Value Databases**

Open-source key-value databases provide developers with flexible and scalable solutions for managing data in a distributed and highly available manner. One popular open-source key-value store, Dynamo, has gained significant attention and adoption.

Dynamo is a highly available and scalable key-value store developed by Amazon. It was designed to handle the demanding requirements of Amazon's shopping cart service. While Dynamo itself is not open-source, its principles have influenced the development of various open-source implementations such as Riak, Voldemort, and Dynomite.

Features of Dynamo include:

#### *Data Model*

Dynamo follows a simple key-value data model, where each item is uniquely identified by a key. The value associated with the key can be any binary data.

#### *Consistency Model*

Dynamo uses a tunable eventual consistency model, which means that updates may not be immediately reflected across all replicas.

#### *Query Language*

Dynamo does not provide a built-in query language. Instead, it relies on simple key-based operations for data access.

#### *Highly Available*

Dynamo prioritizes high availability, ensuring that data remains accessible even in the face of failures or network partitions.

#### *Horizontal Scalability*

Dynamo supports horizontal scaling by distributing data across multiple commodity hardware servers, allowing for increased storage capacity and improved performance with reduced compute cost.

#### **NOTE**

AWS offers Amazon DynamoDB, which is inspired by [the paper on Dynamo](#) and its initial version, which we will cover in more detail in Chapter 10 - AWS Storage Services.

In conclusion, the architecture and design of key-value stores prioritize simplicity, scalability, and high-performance data access. By leveraging a straightforward key-value data model and distributed storage, these databases empower organizations to efficiently handle large-scale data management challenges while providing rapid and reliable data retrieval capabilities. Understanding the principles behind key-value stores equips data professionals with a valuable tool in their arsenal to tackle modern data-intensive applications.

Let's go over the next type of non-relational database based on the document data model, storing and managing data in the form of documents.

## Document Databases

Document databases are a type of non-relational database that are specifically designed for storing, retrieving, and managing semi-structured data in the form of documents. They provide a flexible and schema-less approach to data storage, making them ideal for applications with dynamic and evolving data structures. In this section, we will explore the architecture and design considerations of document stores, shedding light on their key features and benefits.

### Data Model

At the core of a document store lies the document data model. Instead of organizing data into tables with predefined schemas, document stores store data as self-contained documents analogous to rows in relational databases, typically represented in formats such as JSON, BSON, or XML. These documents can have varying structures and can contain nested fields, arrays, and key-value pairs, allowing for hierarchical representation of data. The documents are organized into collections, which are analogous to tables in a relational database. The document model provides the ability to

store and retrieve complex data structures as a single unit, making it well-suited for handling unstructured or semi-structured data. Let's explore the key components of a document store schema and its architecture.

### *Collections*

Collections in a document store are containers that hold related documents. They provide logical grouping of data, allowing you to organize and manage data based on its characteristics or purpose. For example, in an e-commerce application, you might have collections for products, orders, and customers. Each collection can have its own set of documents, each representing an individual entity or record.

### *Documents*

Documents are the fundamental unit of data in a document store. They are analogous to rows in a relational database. A document is a JSON-like structure that stores data as key-value pairs. It represents a single entity or record and can contain nested structures and arrays. Documents within a collection can have different sets of fields, allowing for flexible data structures. This dynamic nature of document stores is particularly advantageous in scenarios where data evolves rapidly or when dealing with diverse and unpredictable data sources. For example, within a "products" collection, each document could have fields such as "name," "price," "description," and so on.

### *Operators*

Document stores provide a rich set of operators to perform various operations on documents. These operators allow you to query, update, and manipulate data within documents. Common operators include:

*Insert:*

Inserts a new document into a collection.

*Update:*

Modifies an existing document by updating or adding fields.

*Delete:*

Removes a document from a collection.

*Query:*

Retrieves documents from a collection based on specified criteria.

*Aggregation:*

Performs complex data aggregations and transformations on documents, such as grouping, sorting, and aggregating data.

*Projection*

Projection is a powerful feature in document stores that allows you to retrieve only the desired fields from a document. With projection, you can specify which fields to include or exclude in the result set. Indexing mechanisms, such as secondary indexes, are commonly used to optimize query performance, enabling fast access to the desired documents based on specific fields or criteria. This can significantly reduce network traffic and improve query performance, as only the required data is transferred from the database to the client.

In summary, document stores offer a flexible and schema-less approach to data storage and retrieval. Collections group related documents, while documents represent individual entities or records. Operators provide functionality for data manipulation, while projection allows for selective retrieval of fields. The distributed architecture of document stores enables high availability, fault

tolerance, and scalability. With their rich features and flexible data model, document stores are well-suited for a wide range of use cases, from content management systems to real-time analytics applications.

## **Availability in Document Stores**

The architecture of a document store typically involves a distributed system with multiple nodes. These nodes work together to provide high availability, fault tolerance, and scalability. Each node can handle read and write operations independently, allowing for parallel processing and improved performance. Data within a collection is often partitioned or sharded across multiple nodes to distribute the workload and enable horizontal scaling.

Document stores employ various mechanisms to achieve availability. Let's explore three key components: replica sets, primary-secondary node clusters, and the heartbeat mechanism.

### *Replica Sets*

A replica set is a group of nodes in a document store that contains multiple copies of the data. Each replica set consists of a primary node and one or more secondary nodes. The primary node is responsible for handling write operations and acting as the primary source of data. Secondary nodes replicate data from the primary node and serve as backups.

Replica sets provide fault tolerance by allowing automatic failover in the event of a primary node failure.

### *Primary-Secondary Node Clusters*

In a primary-secondary node cluster architecture, the primary node accepts write operations and maintains the authoritative copy of the data. Secondary nodes replicate the data from the primary node and serve as read replicas. Clients can read from

any of the secondary nodes, distributing the read workload and improving read performance.

The primary-secondary node cluster architecture ensures that read operations are scalable and can be handled by multiple nodes. In the event of a primary node failure diagnosed by heartbeat mechanism, one of the secondary nodes is promoted as the new primary, and the cluster continues to operate without data loss.

### *Heartbeat Mechanism*

The heartbeat mechanism is responsible for monitoring the health and status of nodes, detecting failures, initiating failover processes, and promoting secondary nodes as new primaries when needed. It involves regular communication between nodes to detect failures and trigger appropriate actions. Nodes in a document store periodically exchange heartbeat messages to signal their availability. If a node fails to respond within a specified time period, it is considered unavailable, and the replica set or cluster takes necessary action to maintain availability.

Overall, these mechanisms provide fault tolerance, automatic failover, data redundancy, and efficient read scaling. By leveraging these features, document stores can maintain continuous access to data, withstand failures, and deliver consistent and reliable performance to applications and users.

## **Advantages, Trade-offs and Considerations**

Document stores find extensive use in various application domains, including content management systems, e-commerce platforms, real-time analytics, and mobile app development. They excel in scenarios where flexibility in data modeling, dynamic schema evolution, and efficient querying of complex data structures are paramount. Document stores also facilitate the storage and retrieval

of unstructured or semi-structured data, making them suitable for scenarios involving user-generated content, sensor data, log files, and social media feeds.

While document stores provide flexibility and scalability, there are trade-offs to consider. The flexibility of the schema-less design can sometimes lead to data consistency challenges, as the enforcement of data integrity constraints may be delegated to the application layer. Additionally, the lack of strong schema enforcement and complex joins can impact certain types of analytical or reporting queries that rely heavily on relationships and aggregations.

In summary, the architecture and design of document stores revolve around the document data model, flexible schema-less storage, distributed scalability, and powerful querying capabilities. By embracing the document-oriented approach, these databases empower organizations to handle semi-structured and evolving data with ease, enabling efficient storage, retrieval, and manipulation of complex data structures. Understanding the principles behind document stores equips data professionals with a versatile tool for managing diverse and dynamic data.

## **Open-source Document Databases**

**MongoDB** is a widely adopted open-source document database known for its scalability, performance, and developer-friendly features. It uses a flexible JSON-like document model, allowing developers to store and retrieve data in a schema-less manner. MongoDB supports dynamic schemas, which means that each document in a collection can have its own unique structure.

Features of MongoDB include:

### *Flexibility*

MongoDB's schema-less design enables flexible data models and easy adaptation to evolving application requirements. It allows

developers to handle varying and evolving data structures within the same collection.

### *Rich Query Language*

It provides a powerful query language with support for complex queries, indexing, and aggregation. It allows developers to express a wide range of query operations and transformations.

### *Multi-document Transactions*

Transactions in MongoDB adhere to the ACID properties, discussed in detail in Chapter 2. MongoDB supports multi-document transactions, enabling developers to work with multiple documents in a single transaction. This is particularly useful in scenarios where data in multiple collections needs to be updated together while maintaining data integrity.

### *Data Consistency Models*

It supports various consistency models, allowing developers to choose the level of consistency needed for their applications.

### *Horizontal Scalability*

It supports horizontal scaling through sharding, allowing data to be distributed across multiple nodes. This enables high performance and the ability to handle large data volumes and high traffic loads.

### *Replication and High Availability*

It supports replica sets, which are self-healing clusters that provide data redundancy and automatic failover. Replica sets ensure high availability and fault tolerance by maintaining multiple copies of data.

In conclusion, MongoDB is a popular open-source document store with unique features and capabilities. It has a large and active community, providing extensive support, documentation, and a wide range of third-party integrations. MongoDB's flexibility and community support make it suitable for various applications and deployment scenarios, including both on-premises and cloud.

#### NOTE

AWS offers Amazon DocumentDB, which is MongoDB compatible with support for powerful ad-hoc queries and comes with transaction support similar to Amazon DynamoDB, which we will cover in more detail in Chapter 10 - AWS Storage Services.

Let's go over the next type of non-relational database based on columnar data model, organizing data into column families, allowing efficient storage and retrieval of large volumes of data.

## Columnar Databases

Columnar databases or column-family databases, also known as Wide column stores, are a type of non-relational database that offer a unique architecture optimized for handling vast amounts of structured and semi-structured data. These databases excel at managing large-scale distributed systems, analytics, and use cases requiring fast read and write performance. In this section, we will explore the architecture and design considerations of wide column stores, unveiling their key features and advantages.

### Data Model

Unlike traditional relational databases that organize data into rows, wide column stores employ a column-oriented data model with columns grouped into column families or column groups. In this

model, data is stored and retrieved by columns rather than by rows. Each column represents a particular attribute or field, and data values belonging to that attribute are stored contiguously on disk. This design offers significant advantages in terms of query performance, as it allows for efficient compression, data skipping, and column-level operations like filtering and aggregation.

Columnar databases employ various compression techniques to improve storage efficiency and query performance. Since each column is stored separately, column-oriented storage allows for better compression ratios. Compression can be applied individually to each column based on its data characteristics, such as data type or redundancy. This not only reduces storage requirements but also improves data access speed by reducing disk I/O and memory footprint.

## **Flexible Schema Design**

The schema design in a wide column store is flexible, enabling the addition or removal of columns without altering the entire dataset. This flexibility allows for easy adaptation to changing business requirements and evolving data models. Schema changes can be performed independently for each column family, providing greater agility in managing data structures.

## **Keys**

Wide column stores typically use two types of keys: the partition key and the clustering key, which constitute the composite primary key.

### *Partition Key*

The partition key is used to distribute data across the nodes in a cluster. It determines the physical location where data is stored. Data is partitioned based on the partition key, and each partition is stored on a separate node. Efficient selection of the partition key is crucial for even data distribution and optimal query performance.

## *Clustering Key*

The clustering key is used to define the order of data within a partition. It allows for efficient sorting and range-based queries within a partition. The clustering key can consist of multiple columns, defining a hierarchical sorting order for the data.

Choosing the appropriate partition key and clustering key is a crucial aspect of designing a wide column store database. These keys determine how data is distributed, stored, and accessed within the database. Partition key should be chosen on a column with high cardinality, aiming for even data distribution, while clustering key should be chosen based on query access pattern of the application depending on how data is retrieved in specific order or pattern.

## **Consistency Levels**

Wide column stores provide different levels of consistency to balance performance and data integrity. Wide column stores often offer tunable consistency, allowing developers to configure consistency levels on a per-operation basis. This enables the fine-tuning of consistency requirements for specific read and write operations.

Consistency levels in tunable consistency within a wide column store are based on the concept of quorum, which refers to the minimum number of replicas that need to participate in a read or write operation. The number of replicas required to form a quorum can vary depending on the desired consistency level. Let's explore the consistency levels and their corresponding quorum requirements:

### *Eventual Consistency*

Eventual consistency is the weakest level of consistency, allowing for the highest degree of availability and low latency. In wide column stores, eventual consistency means that replicas are asynchronously updated over time, resulting in the potential for

temporary data inconsistencies. For eventual consistency, the quorum requirements can be defined as:

- Any: In this model, the operation is considered successful as soon as it is applied to at least one replica, without the requirement for acknowledgment or synchronization across all replicas. Over time, the updates are propagated and eventually converge to a consistent state across the system.

### *Weak Consistency*

Weak consistency is a less common consistency level and provides a lower level of data integrity compared to strong consistency. In wide column stores, weak consistency allows for more relaxed requirements, prioritizing availability and low latency over data consistency. For weak consistency, the quorum requirements can be defined as:

- Quorum: In this model with a quorum-based approach, the consistency level is achieved when a majority of replicas (more than half) acknowledge the operation. This means that for a successful read or write operation, the data is considered consistent if it is read or written to at least one replica within the quorum.
- Local Quorum: In this model, a local quorum refers to a majority of replicas within the local data center or region. The operation is considered consistent if it is completed within the local quorum.

### *Strong Consistency*

Strong consistency provides the highest level of data consistency but may come with increased latency and reduced availability. In wide column stores, strong consistency ensures that all replicas are synchronously updated and consistent before a response is

returned. For strong consistency, the quorum requirements can be defined as:

- All: In this model, the operation must be acknowledged and applied to all replicas in the cluster before it is considered consistent. This means that for a read or write operation to be successful, it must be propagated to and acknowledged by every replica in the system.
- Each Quorum: In this model over multiple data centers or regions, each quorum refers to a majority of replicas within each data center or region. The operation is considered consistent when it is completed within each quorum.

It's important to note that the specific consistency levels and their corresponding quorum requirements may vary between different wide column store databases. The above examples provide a general understanding of how consistency levels are defined based on quorum in a tunable consistency model within a wide column store. When selecting a consistency level, it's crucial to consider the trade-offs between consistency, availability, and performance based on the specific requirements of your application and use case.

In conclusion, wide column stores offer a flexible data model, allowing for dynamic schema evolution and efficient storage of structured and semi-structured data. The schema design is adaptable, and keys (partition and clustering) play a crucial role in data distribution and query optimization. Consistency levels provide options to balance performance and data integrity. Understanding the wide column store data model, schema design, keys, and consistency levels is essential for effectively utilizing wide column stores in various applications and use cases.

## Columnar Store Architecture

This section discusses columnar store architecture, which may vary from one implementation to other but generally, the core components remain the same. Let's explore the key components of data storage in columnar storage and how they contribute to efficient data management.

### *Commit Log*

The commit log is a write-ahead log that records all write operations performed on the database. It serves as a durable and sequential log of changes made to the data. Whenever a write operation occurs, it is first written to the commit log for durability and fault tolerance. The commit log ensures that no data is lost in the event of a system failure or crash.

### *Memtable*

The memtable is an in-memory data structure that stores recent write operations before they are flushed to disk. It acts as a write buffer, temporarily holding the data in memory before persisting it to disk in an efficient manner. The memtable is typically structured as an ordered hash table or skip list, allowing for fast write operations. As the memtable fills up, it is periodically flushed to disk, creating SSTables.

### *SSTables (Sorted String Tables)*

SSTables are the on-disk data structure in columnar storage. They are immutable and sorted by key, enabling efficient range queries and compression. Each SSTable contains a set of sorted key-value pairs, where the keys correspond to the row keys and the values represent the columnar data for each key. SSTables are optimized for fast read operations and can be compacted to improve performance and storage efficiency. Columnar storage leverage Bloom filters to ascertain if an SSTable contains data

related to a specific partition. Bloom filters are used for index scans but not for range scans. Bloom filters function as probabilistic sets to quickly test whether an element is a member of the set, offering a memory-accuracy trade-off. It's particularly efficient in cases where false positives are acceptable but false negatives are not desired.

### *Compaction Strategies*

Compaction is the process of merging and compacting multiple SSTables to improve read performance and manage disk space. Various compaction strategies are employed to balance read and write performance, as well as disk space utilization:

#### *Size Tiered Compaction:*

This strategy groups SSTables into levels based on their size. As SSTables in a level reach a certain threshold, they are merged into a new SSTable in the next level. It reduces the number of SSTables and simplifies data access at the cost of some write amplification.

#### *Leveled Compaction:*

In this strategy, SSTables are organized into multiple levels, with each level containing SSTables of roughly equal size. SSTables are compacted within each level, and as they progress to higher levels, they are merged with larger SSTables. Leveled compaction offers more balanced read and write performance at the expense of higher disk space overhead.

#### *TimeWindow Compaction:*

This strategy is specifically designed for time-series data, where data is organized based on a time window. It allows for

efficient expiration of old data by dropping or merging SSTables based on time-based criteria.

### *Tombstones for Soft Delete*

Tombstones are special markers used for soft deletes in columnar storage. When a row or column is deleted, a tombstone is created to indicate the deletion. Tombstones ensure that deleted data is properly handled during compaction and read operations, maintaining data consistency across SSTables. During compaction, tombstones are used to identify and remove deleted or expired data.

Excessive tombstones in columnar storage can lead to significant increase in garbage collection (GC) pauses during compaction, as columnar stores need to process and eliminate these tombstones, slowing down data access and increasing latency. Also, large numbers of tombstones consume storage space and querying data in the presence of numerous tombstones can result in slower query performance as the columnar database has to shift through these markers, potentially causing unnecessary overhead and delays in retrieving relevant data. Effective tombstone management is crucial to maintaining columnar store's performance and responsiveness.

In summary, data storage in columnar storage involves various components such as the commit log, memtable, SSTables, compaction strategies, and tombstones. These components work together to ensure durability, efficient write buffering, disk storage optimization, and data consistency. Understanding these elements is crucial for effectively managing and leveraging the benefits of columnar storage in data-intensive applications.

## **Advantages, Trade-offs and Considerations**

Wide column stores find widespread use in various domains, particularly in analytics, big data processing, and time-series data. They are well-suited for applications that involve complex queries, ad hoc analysis, data warehousing, and high-speed data ingestion. Use cases include log analysis, financial analytics, real-time reporting, IoT data processing, and customer behavior analysis.

While wide column stores offer significant advantages in scalability and query performance, they come with trade-offs. Due to their distributed nature and complex data organization, they often require more advanced data modeling and query optimization compared to traditional relational databases. Wide column stores may not be as suitable for scenarios that heavily rely on complex joins across multiple tables.

In conclusion, the architecture and design of wide column stores revolve around the column-oriented data model, distributed scalability, and efficient analytics capabilities. By leveraging the benefits of columnar storage and flexible schema design, these databases enable organizations to efficiently store, retrieve, and analyze large volumes of structured as well as semi-structured data.

## **Open Source Columnar Databases**

**Apache Cassandra** is a highly scalable and distributed open-source columnar database known for its ability to handle massive amounts of structured and semi-structured data across multiple commodity servers. It offers a robust architecture and a range of features that make it suitable for high-performance and fault-tolerant applications. Let's delve into Apache Cassandra and explore its key features.

Features of Cassandra include:

*Distributed Query Language (CQL)*

Cassandra utilizes CQL, a query language similar to SQL, to interact with the database. CQL provides a familiar syntax for developers and offers features like data definition, data manipulation, and querying capabilities. It simplifies data access and supports complex querying, filtering, and sorting operations.

### *Distributed and Decentralized Architecture*

Cassandra follows a distributed architecture model, employing a peer-to-peer approach. It organizes data across a cluster of nodes, allowing for easy scalability and fault tolerance. Each node in the leaderless cluster can perform read and write operations independently, resulting in a distributed and highly available database system.

### *Linear Scalability*

Cassandra's architecture enables linear scalability, meaning it can handle increasing workloads by simply adding more nodes to the cluster. This scalability allows for seamless growth as data volumes and user demand increase, making it a suitable choice for applications with rapidly growing datasets.

### *High Availability and Fault Tolerance*

Cassandra provides built-in high availability and fault tolerance mechanisms. It uses peer-to-peer replication across nodes, ensuring data redundancy and preventing data loss in the event of node failures. Cassandra automatically replicates data across multiple nodes based on the replication factor, guaranteeing data availability and durability.

### *Tunable Consistency*

Cassandra offers tunable consistency, allowing developers to define the level of consistency required for each read and write operation. This flexibility enables balancing performance and data

consistency based on application-specific needs. Developers can configure consistency levels from strong consistency to eventual consistency.

### *Flexible Data Replication and Data Centers*

Cassandra allows the replication of data across multiple data centers or geographic regions, providing geographical redundancy and disaster recovery capabilities. It supports multiple replication strategies, including network topology-aware data replication, ensuring data availability and low-latency access in distributed environments.

In summary, Apache Cassandra is an open-source columnar database that offers scalability, high availability, fault tolerance, tunable consistency, and support for distributed and decentralized architectures. With its columnar data model, distributed query language (CQL), time-series data capabilities, and active community support, Cassandra empowers developers to build robust and scalable applications that can handle massive amounts of data across distributed environments.

#### **NOTE**

AWS offers Amazon Keyspaces, which is Apache Cassandra compatible, highly scalable, available, and managed wide-column database service offered as a serverless solution, which we will cover in more detail in Chapter 10 - AWS Storage Services.

Let's go over the next type of non-relational database based on a graph data model, well-suited for applications that require complex querying and analysis of interconnected data..

# Graph Databases

Graph databases are a specialized type of non-relational database designed to handle highly interconnected data and complex relationships. They excel at storing, querying, and traversing graph-like structures, making them ideal for scenarios involving social networks, recommendation systems, fraud detection, and knowledge graphs. In this section, we will explore the architecture and design considerations of graph stores, uncovering their key features and advantages.

## Data Model

At the heart of a graph store lies the graph data model, which represents data as a collection of interconnected nodes (vertices) and relationships (edges). Each node typically corresponds to an entity or an object, while the relationships represent the connections between nodes. Graph databases store these entities, relationships, and associated properties, allowing for efficient traversal and querying of the graph structure. The graph data model offers a natural representation of complex and interconnected data, enabling rich and expressive data modeling.

Unlike traditional databases that focus on data entities and their attributes, graph stores prioritize the relationships between entities. Relationships in graph databases can have properties and can be directed or undirected, allowing for various types of connections. The relationship-centric design of graph stores enables efficient navigation of the graph, facilitating queries that traverse nodes and relationships to uncover patterns, paths, and insights within the data.

## Data Access and Retrieval

Graph stores provide powerful query languages or APIs specifically designed for traversing and querying graph structures. These

languages, such as Cypher (used in Neo4j) or Gremlin (used in Apache TinkerPop), allow users to express complex graph patterns, perform graph traversals, and filter and aggregate data based on relationships and properties. The query languages provide a declarative and expressive syntax that simplifies the process of querying and exploring the graph data.

Graph stores employ various indexing techniques to optimize graph traversal and query performance. They typically use indexes on nodes, relationships, or properties to speed up the lookup of specific elements within the graph. Additionally, graph databases often utilize caching mechanisms to store frequently accessed graph elements in memory, further improving query response times. These indexing and caching strategies enable graph stores to efficiently handle complex graph traversals and pattern matching queries.

## **Advantages, Trade-offs and Considerations**

Graph stores are particularly well-suited for applications that involve complex relationships, network analysis, and recommendation systems. They find extensive use in social networking platforms, fraud detection systems, recommendation engines, knowledge graphs, and data lineage analysis. Graph databases excel in scenarios where understanding and analyzing the relationships between entities are of primary importance.

While graph stores offer powerful graph traversal and querying capabilities, they may face challenges when dealing with massive datasets and complex graph patterns. The performance of graph queries heavily depends on the size and structure of the graph, and certain complex queries may require additional optimization techniques. Additionally, graph stores may not be the best choice for scenarios that primarily involve simple tabular data or scenarios that require strong transactional capabilities.

In summary, the architecture and design of graph stores revolve around the graph data model, relationship-centric design, efficient graph traversal and querying, and scalable distributed architectures. By leveraging the power of graph structures and relationships, these databases enable organizations to store, navigate, and uncover valuable insights within highly interconnected datasets.

## **Open-source Graph Databases**

**Neo4j** is a leading open-source graph database known for its powerful graph processing capabilities and intuitive query language. Neo4j is designed to efficiently store, manage, and traverse highly connected data, making it ideal for applications that heavily rely on complex relationships and interconnections.

Features of Neo4j include:

### *Graph Data Model*

Neo4j is built around the property graph data model, which consists of nodes, relationships, and properties. Nodes represent entities or objects, relationships define connections between nodes, and properties store key-value pairs associated with nodes and relationships. This flexible and expressive data model allows developers to represent and explore complex relationships easily.

### *Graph Query Language*

Neo4j uses Cypher, a powerful and intuitive query language specifically designed for graph databases. Cypher allows developers to express complex graph patterns and perform advanced graph traversals using a familiar and human-readable syntax. It provides rich querying capabilities, including filtering, aggregations, and pattern matching, making it easier to extract meaningful insights from highly connected data.

## *Scalability and Performance*

Neo4j offers excellent scalability and performance for graph data processing. It supports horizontal scaling by distributing the graph across multiple machines in a cluster, enabling high throughput and accommodating large-scale deployments. Neo4j's query optimizer efficiently executes queries, leveraging index structures and caching mechanisms to ensure optimal performance even on massive graphs.

## *ACID Compliance*

Neo4j provides strong ACID (Atomicity, Consistency, Isolation, Durability) guarantees to ensure data integrity and reliability. It maintains strict transactional consistency, allowing multiple operations to be grouped into atomic units of work. This ensures that changes made to the graph are durable and consistent, even in the presence of concurrent operations.

## *Native Graph Processing*

Neo4j is designed from the ground up as a native graph database, allowing it to leverage the power of graph processing algorithms. It provides a wide range of graph-specific operations, such as shortest path calculations, centrality measurements, community detection, and graph analytics. These built-in capabilities enable developers to perform complex graph analysis and traverse relationships efficiently.

## *Data Visualization*

Neo4j provides visualization tools that allow developers to explore and visualize graph data. These tools help understand the structure of the graph, identify patterns, and gain insights into complex relationships. Visualization enhances the intuitive understanding of the data model and aids in decision-making processes.

With its graph data model, powerful Cypher query language, scalability, performance, ACID compliance, native graph processing capabilities, vibrant community, and visualization tools, Neo4j empowers developers to build sophisticated applications that leverage the power of relationships and graph data. Whether it's social networks, recommendation systems, fraud detection, or knowledge graphs, Neo4j provides a robust foundation for graph-based applications.

#### NOTE

AWS offers fully managed graph database service, Amazon Neptune, which we will cover in more detail in Chapter 10 - AWS Storage Services.

## Conclusion

As we conclude our exploration of non-relational databases, it's evident that the landscape is rich with diversity. From key-value stores optimized for rapid operations, to columnar stores tailored for analytical prowess, and document and graph databases catering to flexible data structures and intricate relationships, each type offers a unique set of capabilities. We did not cover other database types like timeseries (ex. InfluxDB), in-memory (ex. Redis), text search (ex. ElasticSearch), vector (Milvus), geospatial, ledger databases, but the choices are immense. The choice of database type, as shown in [Figure 3-4](#) should be guided by a deep understanding of your application's requirements, workload characteristics, and growth expectations.

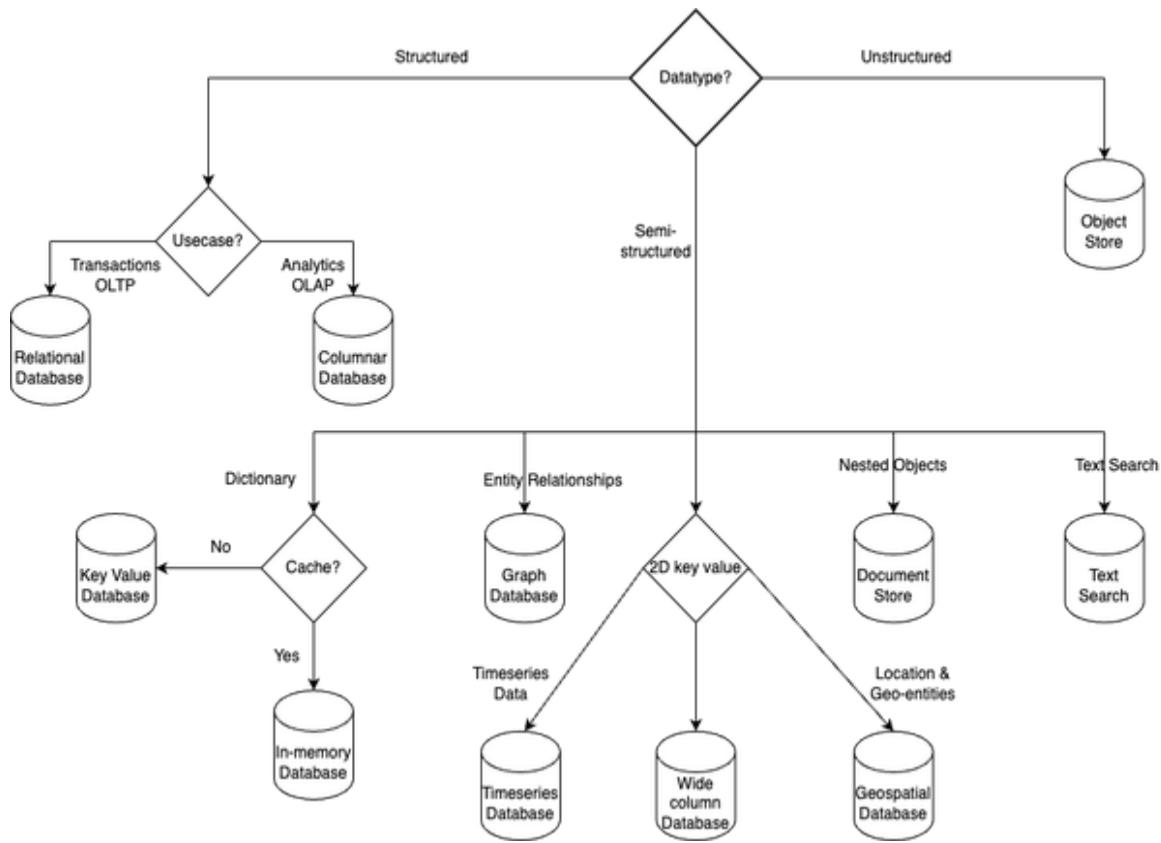


Figure 3-4. Database selection decision flowchart

We've taken a journey through different kinds of databases in Chapter 2 and 3, each with its own special abilities. An application may require a relational database or few non-relational database types along with it to support the specific use case. Think of them like different tools in a toolbox, ready to help us manage data in exciting ways.

Suppose, you have to put a nail in the wall and you have a hammer, wrench, screw-driver and other tools. Sure, using multiple tools means you have to learn how to handle them differently, but that's okay. Each tool has its job, and it's important to use the right one (the hammer in this case) for the task at hand. So, remember, always pick the right tool for the job, instead of trying to make one tool do everything or use all the tools – because it just doesn't work that way!

Let's relook at the comparison between relational and non-relational databases in Table 3-1 to refresh your knowledge.

T  
a  
b  
/  
e  
3  
-  
1  
.R  
e  
/  
a  
ti  
o  
n  
a  
/  
v  
s  
N  
o  
n  
-  
r  
e  
/  
a  
ti  
o  
n  
a  
/  
l

S  
t  
o  
r  
e

Property	Relational Store	Non-relational Store
Data Model	<p>Follows a strict schema.</p> <p>Data is stored in tables with well defined structure and adding columns to a defined table can be difficult.</p>	<p>Are schema-less.</p> <p>Data are stored in tables with well defined structure</p> <p>Data can be structured as key-value pairs, JSON and adding columns to a defined table can be like documents, or any of the other various other nosql designs like graph, wide column etc.</p>
Hierarchical Storage	<p>These databases are not suited for hierarchical data storage.</p>	<p>These databases are best suited for hierarchical or interconnected data storage.</p>
Scalability	<p>SQL databases are better suited for vertical scaling.</p>	<p>NoSQL databases are horizontally scalable. And as such can employ techniques like sharding etc to facilitate</p>

handling of huge amounts of data.

Joins	Suitable for queries that require a lot of complex joins across multiple tables.	Joins are possible but involve a lot of computational overhead to perform especially on a distributed database.
-------	--	---

CAP Theorem	SQL transactions always comply with ACID properties (Strong Consistency).	NoSQL databases generally follow BASE properties (Eventual Consistency).
-------------	---	--

As we conclude this chapter on non-relational databases, we hope you feel equipped with the knowledge and know-hows of databases, along with a basic understanding of when to choose which database. The next chapter will embark you on an exploration of caching policies and strategies, reducing the time required to access frequently accessed data, even from datastores.

# Chapter 4. Caching Policies and Strategies

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

In computing, a cache is a component or mechanism used to store frequently accessed data or instructions closer to the processor, reducing the latency of retrieving the information from slower storage or external resources. Caches are typically implemented as high-speed, small-capacity memories located closer to the CPU than the main memory. The goal is to improve overall system performance by reducing the time required to access data or instructions.

The concept of cache revolves around the principle of locality, which suggests that programs tend to access a relatively small portion of their data or instructions repeatedly. By storing this frequently accessed information in a cache, subsequent access to the same data can be served faster, resulting in improved performance.

When data is accessed from a cache, there are two possible outcomes: cache hit and cache miss. A cache hit occurs when the requested data is found in the cache, allowing for fast retrieval without accessing the slower main memory or external resources. On the other hand, a cache miss happens when the requested data is not present in the cache, requiring the system to fetch the data from the main memory or external storage. Cache hit rates measure the effectiveness of the cache in serving requests without needing to access slower external storage, while cache miss rates indicate how often the cache fails to serve requested data.

This chapter will cover important information to help you understand how to use data caching effectively. We'll talk about cache eviction policies, which are rules for deciding when to remove data from the cache to make retrieval of important data faster. We'll also cover cache invalidation, which ensures the cached data is always correct and matches the real underlying data source. The chapter will also discuss caching strategies for both read and write intensive applications. We'll also cover how to actually put caching into action, including where to put the caches to get the best results. You'll also learn about different ways caching works and why Content Delivery Networks (CDNs) are important. And finally, you'll learn about two popular open-source caching solutions. So, let's get started with the benefits of caching.

## Caching Benefits

Caches play a crucial role in improving system performance and reducing latency for several reasons:

### *Faster Access*

Caches offer faster access times compared to main memory or external storage. By keeping frequently accessed data closer to the CPU, cache access times can be significantly lower, reducing the time required to fetch data.

### *Reduced Latency*

Caches help reduce latency by reducing the need to access slower storage resources. By serving data from a cache hit, the system avoids the delay associated with fetching data from main memory or external sources, thereby reducing overall latency.

### *Bandwidth Optimization*

Caches help optimize bandwidth usage by reducing the number of requests sent to slower storage. When data is frequently accessed from the cache, it reduces the demand on the memory bus or external interfaces, freeing up resources for other operations.

### *Improved Throughput*

Caches improve overall system throughput by allowing the CPU to access frequently needed data quickly, without waiting for slower storage access. This enables the CPU to perform more computations in a given amount of time, increasing overall system throughput.

Amdahl's Law and the Pareto distribution provide further insights into the benefits of caching:

### *Amdahl's Law*

Amdahl's Law states that the overall speedup achieved by optimizing a particular component of a system is limited by the fraction of time that component is utilized. Caches, being a critical optimization component, can have a significant impact on overall system performance, especially when the fraction of cache hits is high. Amdahl's Law emphasizes the importance of efficient caching to maximize the benefits of performance optimization.

### *Pareto Distribution*

The Pareto distribution, also known as the 80/20 rule, states that a significant portion of the system's workload is driven by a small fraction of the data. Caching aligns well with this distribution by allowing the frequently accessed data to reside in a fast cache, serving the most critical operations efficiently. By focusing caching efforts on the most accessed data, the Pareto distribution can be leveraged to optimize performance for the most important workloads.

In summary, caches provide faster access to frequently accessed data, reducing latency and improving overall system performance. They help optimize bandwidth, increase throughput, and align with principles such as Amdahl's Law and the Pareto distribution to maximize performance benefits.

The next section will cover different policies to perform cache eviction, like techniques such as least recently used (LRU) and least frequently used (LFU), which can help you choose the best caching policy for different situations.

## **Cache Eviction Policies**

Caching plays a crucial role in improving the performance and efficiency of data retrieval systems by storing frequently accessed data closer to the consumers. Caching policies determine how the cache handles data eviction and replacement, when its capacity is reached. Cache eviction policies try to maximize the cache hit ratio—the percentage of time the requested item was found in the cache and served. Higher cache hit ratio reduces the necessity to retrieve data from external storage, resulting in better system performance. In this section, we will explore various caching policies, including Belady's algorithm, queue-based policies (FIFO, LIFO), recency-

based policies (LRU, TLRU, MRU, SLRU), and frequency-based policies (LFU, LFRU).

## **Belady's Algorithm**

Belady's algorithm is an optimal caching algorithm that evicts the data item that will be used furthest in the future. It requires knowledge of the future access pattern, which is usually impractical to obtain. Belady's algorithm serves as a theoretical benchmark for evaluating the performance of other caching policies.

### *Queue-Based Policies*

Queue-based cache eviction policies involve managing the cache by treating it like a queue. When the cache reaches its capacity, one of the queue-based policies is used to remove data to make space for new data.

#### *FIFO (First-In-First-Out)*

FIFO is a simple caching policy that evicts the oldest data item from the cache. It follows the principle that the first data item inserted into the cache is the first one to be evicted when the cache is full. FIFO is easy to implement but may suffer from the “aging” problem, where recently accessed items are evicted prematurely.

#### *LIFO (Last-In-First-Out)*

LIFO is the opposite of FIFO, where the most recently inserted data item is the first one to be evicted. LIFO does not consider the access pattern and can result in poor cache utilization and eviction decisions.

### *Recency-Based Policies*

Recency-based cache eviction policies focus on the time aspect of data access. These policies prioritize keeping the most recently accessed items in the cache.

### *LRU (Least Recently Used)*

LRU is a popular caching policy that evicts the least recently accessed data item from the cache. It assumes that recently accessed items are more likely to be accessed in the near future. LRU requires tracking access timestamps for each item, making it slightly more complex to implement.

### *MRU (Most Recently Used)*

MRU evicts the most recently accessed data item from the cache. It assumes that the most recently accessed item is likely to be accessed again soon. MRU can be useful in scenarios where a small subset of items is accessed frequently.

## *Frequency-Based Policies*

Frequency-based cache eviction policies prioritize retaining items in the cache based on how often they are accessed. The cache replaces items that have been accessed the least frequently, assuming that rarely accessed data may not be as critical for performance optimization.

### *LFU (Least Frequently Used)*

LFU evicts the least frequently accessed data item from the cache. It assumes that items with lower access frequency are less likely to be accessed in the future. LFU requires maintaining access frequency counts for each item, which can be memory-intensive.

### *LFRU (Least Frequently Recently Used)*

LFRU combines the concepts of LFU and LRU by considering both the frequency of access and recency of access. It evicts the item with the lowest frequency count among the least recently used items.

## **Allowlist Policy**

An allowlist policy for cache replacement is a mechanism that defines a set of prioritized items eligible for retention in a cache when space is limited. Instead of using a traditional cache eviction policy that removes the least recently used or least frequently accessed items, an allowlist policy focuses on explicitly specifying which items should be preserved in the cache. This policy ensures that important or high-priority data remains available in the cache, even during periods of cache pressure. By allowing specific items to remain in the cache while evicting others, the allowlist policy optimizes cache utilization and improves performance for critical data access scenarios.

Caching policies serve different purposes and exhibit varying performance characteristics based on the access patterns and workload of the system. Choosing the right caching policy depends on the specific requirements and characteristics of the application.

By understanding and implementing these caching policies effectively, system designers and developers can optimize cache utilization, improve data retrieval performance, and enhance the overall user experience. Let's discuss different cache invalidation strategies, which are applied post identifying which data to evict based on the above cache eviction policies.

## **Cache Invalidation**

Cache invalidation is a crucial aspect of cache management that ensures the cached data remains consistent with the underlying data

source. Effective cache invalidation strategies help maintain data integrity and prevent stale or outdated data from being served. Here are three common cache invalidation techniques: active invalidation, invalidating on modification, invalidating on read, and time-to-live (TTL).

### *Active Invalidation*

Active invalidation involves explicitly removing or invalidating cached data when changes occur in the underlying data source. This approach requires the application or the system to actively notify or trigger cache invalidation operations. For example, when data is modified or deleted in the data source, the cache is immediately updated or cleared to ensure that subsequent requests fetch the latest data. Active invalidation provides precise control over cache consistency but requires additional overhead to manage the invalidation process effectively.

### *Invalidating on Modification*

With invalidating on modification, the cache is invalidated when data in the underlying data source is modified. When a modification operation occurs, such as an update or deletion, the cache is notified or flagged to invalidate the corresponding cached data. The next access to the invalidated data triggers a cache miss, and the data is fetched from the data source, ensuring the cache contains the most up-to-date information. This approach minimizes the chances of serving stale data but introduces a slight delay for cache misses during the invalidation process.

### *Invalidating on Read*

In invalidating on read, the cache is invalidated when the cached data is accessed or read. Upon receiving a read request, the cache checks if the data is still valid or has expired. If the data is expired or flagged as invalid, the cache fetches the latest data

from the data source and updates the cache before serving the request. This approach guarantees that fresh data is always served, but it adds overhead to each read operation since the cache must validate the data's freshness before responding.

### *Time-to-Live (TTL)*

Time-to-Live is a cache invalidation technique that associates a time duration with each cached item. When an item is stored in the cache, it is marked with a TTL value indicating how long the item is considered valid. After the TTL period elapses, the cache treats the item as expired, and subsequent requests for the expired item trigger cache misses, prompting the cache to fetch the latest data from the data source. TTL-based cache invalidation provides a simple and automatic way to manage cache freshness, but it may result in serving slightly stale data until the TTL expires.

The choice of cache invalidation strategy depends on factors such as the nature of the data, the frequency of updates, the performance requirements, and the desired consistency guarantees. Active invalidation offers precise control but requires active management, invalidating on modification ensures immediate data freshness, invalidating on read guarantees fresh data on every read operation, and TTL-based invalidation provides a time-based expiration mechanism. Understanding the characteristics of the data and the system's requirements helps in selecting the appropriate cache invalidation strategy to maintain data consistency and improve overall performance.

The next section covers different caching strategies to ensure that data is properly consistent between the cache and the underlying data source.

# Caching Strategies

Caching strategies define how data is managed and synchronized between the cache and the underlying data source. In this section, we will explore several caching strategies as shown in **Figure 4-1**, including cache-aside, read-through, refresh-ahead, write-through, write-around, and write-back.

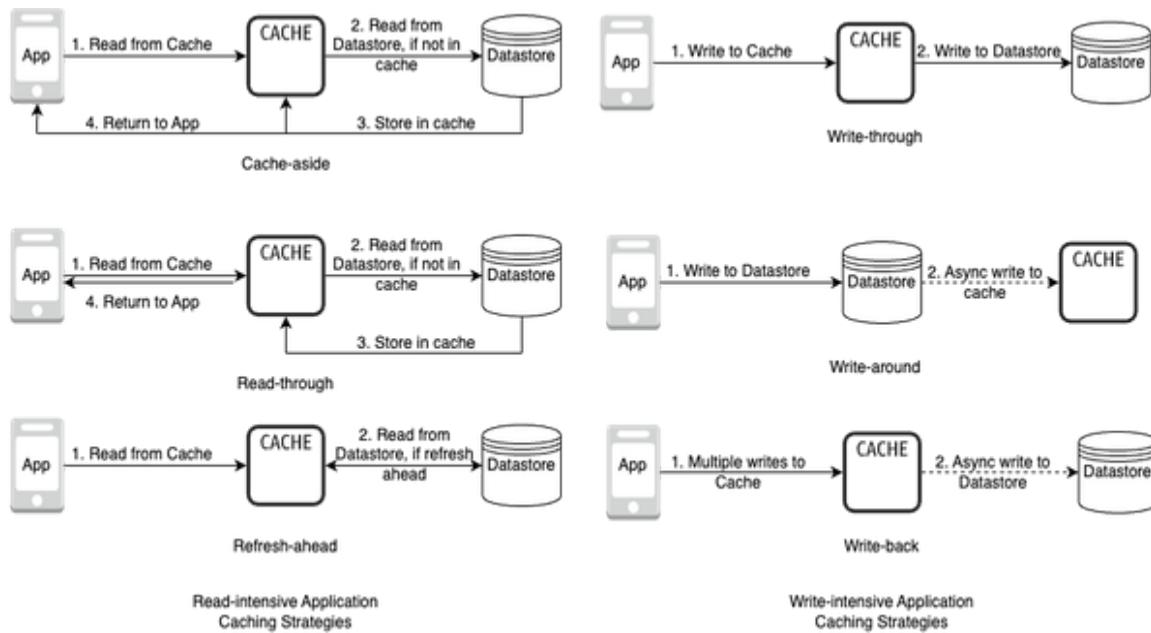


Figure 4-1. Caching Strategies

The left-hand side of the diagram displays read-intensive caching strategies, focusing on optimizing the retrieval of data that is frequently read or accessed. The goal of a read-intensive caching strategy is to minimize the latency and improve the overall performance of read operations by serving the cached data directly from memory, which is much faster than fetching it from a slower and more distant data source. This strategy is particularly beneficial for applications where the majority of operations involve reading data rather than updating or writing data.

Let's take a look at those in more detail:

*Cache-Aside*

Cache-aside caching strategy, also known as lazy loading, delegates the responsibility of managing the cache to the application code. When data is requested, the application first checks the cache. If the data is found, it is returned from the cache. If the data is not in the cache, the application retrieves it from the data source, stores it in the cache, and then returns it to the caller. Cache-aside caching offers flexibility as the application has full control over caching decisions but requires additional logic to manage the cache.

### *Read-Through*

Read-through caching strategy retrieves data from the cache if available; otherwise, it fetches the data from the underlying data source. When a cache miss occurs for a read operation, the cache retrieves the data from the data source, stores it in the cache for future use, and returns the data to the caller. Subsequent read requests for the same data can be served directly from the cache, improving the overall read performance. This strategy offloads the responsibility of managing cache lookups from the application unlike Cache-aside strategy, providing a simplified data retrieval process.

### *Refresh-Ahead*

Refresh-ahead caching strategy, also known as prefetching, proactively retrieves data from the data source into the cache before it is explicitly requested. The cache anticipates the future need for specific data items and fetches them in advance. By prefetching data, the cache reduces latency for subsequent read requests and improves the overall data retrieval performance.

The right-hand side of the diagram displays the write-intensive strategies, focussing around optimizing the storage and management of data that is frequently updated or written. Unlike

read-intensive caching, where the focus is on optimizing data retrieval, a write-intensive caching strategy aims to enhance the efficiency of data updates and writes, while still maintaining acceptable performance levels. In a write-intensive caching strategy, the cache is designed to handle frequent write operations, ensuring that updated data is stored temporarily in the cache before being eventually synchronized with the underlying data source, such as a database or a remote server. This approach can help reduce the load on the primary data store and improve the application's responsiveness by acknowledging write operations more quickly.

Let's take a look at those in more detail:

#### *Write-Through*

Write-through caching strategy involves writing data to both the cache and the underlying data source simultaneously. When a write operation occurs, the data is first written to the cache and then immediately propagated to the persistent storage synchronously before the write operation is considered complete. This strategy ensures that the data remains consistent between the cache and the data source. However, it may introduce additional latency due to the synchronous write operations.

#### *Write-Around*

Write-around caching strategy involves bypassing the cache for write operations. When the application wants to update data, it writes directly to the underlying data source, bypassing the cache. As a result, the written data does not reside in the cache, reducing cache pollution with infrequently accessed data. However, subsequent read operations for the updated data might experience cache misses until the data is fetched again from the data source and cached.

#### *Write-Back*

Write-back caching strategy allows write operations to be performed directly on the cache, deferring the update to the underlying data source until a later time. When data is modified in the cache, the change is recorded in the cache itself, and the update is eventually propagated to the data source asynchronously on schedule or when specific conditions are met (e.g., cache eviction, time intervals). Write-back caching provides faster write operations by reducing the number of immediate disk writes. However, it introduces a potential risk of data loss in the event of a system failure before the changes are flushed to the data source.

Each caching strategy has its own advantages and considerations, and the selection of an appropriate strategy depends on the specific requirements and characteristics of the system.

By understanding these caching strategies, system designers and developers can make informed decisions to optimize data access and improve the overall performance of their applications. Let's cover different deployment options for a cache in the overall system and how it affects the performance and data sharing.

## Caching Deployment

When deploying a cache, various deployment options are available depending on the specific requirements and architecture of the system. Here are three common cache deployment approaches: in-process caching, inter-process caching, and remote caching.

### *In-Process Caching*

In in-process caching, the cache resides within the same process or application as the requesting component. The cache is typically implemented as an in-memory data store and is directly accessible by the application or service. In-process caching

provides fast data access and low latency since the cache is located within the same process, enabling direct access to the cached data. This deployment approach is suitable for scenarios where data sharing and caching requirements are limited to a single application or process.

### *Inter-Process Caching*

Inter-process caching involves deploying the cache as a separate process or service that runs alongside the applications or services. The cache acts as a dedicated caching layer that can be accessed by multiple applications or processes. Applications communicate with the cache using inter-process communication mechanisms such as shared memory, pipes, sockets, or remote procedure calls (RPC). Inter-process caching allows multiple applications to share and access the cached data, enabling better resource utilization and data consistency across different components. It is well-suited for scenarios where data needs to be shared and cached across multiple applications or processes within a single machine.

### *Remote Caching*

Remote caching involves deploying the cache as a separate service or cluster that runs on a different machine or location than the requesting components. The cache service is accessed remotely over a network using protocols such as HTTP, TCP/IP, or custom communication protocols. Remote caching enables distributed caching and can be used to share and cache data across multiple machines or even geographically distributed locations. It provides scalability, fault-tolerance, and the ability to share cached data among different applications or services running on separate machines. Remote caching is suitable for scenarios that require caching data across a distributed system or when the cache needs to be accessed by components running on different machines.

The choice of cache deployment depends on factors such as the scale of the system, performance requirements, data sharing needs, and architectural considerations. In-process caching offers low latency and direct access to data within a single process, inter-process caching enables sharing and caching data across multiple applications or processes, and remote caching provides distributed caching capabilities across multiple machines or locations.

Understanding the specific requirements and characteristics of the system helps in selecting the appropriate cache deployment strategy to optimize performance and resource utilization. Let's cover different caching mechanisms to improve application performance in the next section.

## Caching Mechanisms

In this section, we will explore different caching mechanisms, including client-side caching, CDN caching, web server caching, application caching, database caching, query-level caching, and object-level caching.

### *Client-side Caching*

Client-side caching involves storing cached data on the client device, typically in the browser's memory or local storage. This mechanism allows web applications to store and retrieve static resources, such as HTML, CSS, JavaScript, and images, directly from the client's device. Client-side caching reduces the need to fetch resources from the server on subsequent requests, leading to faster page load times and improved user experience.

### *CDN Caching*

Content Delivery Network (CDN) caching is a mechanism that involves caching static content on distributed servers strategically located across different geographic regions. CDNs serve cached content to users based on their proximity to the CDN server,

reducing the latency and load on the origin server. CDN caching is commonly used to cache static files, media assets, and other frequently accessed content, improving the overall performance and scalability of web applications.

### *Web Server Caching*

Web server caching refers to caching mechanisms implemented at the server-side to store frequently accessed content. When a request is made to the server, it first checks if the requested content is already cached. If found, the server serves the cached content directly, avoiding the need to regenerate the content. Web server caching is effective for static web pages, dynamic content with a long expiration time, and content that is expensive to generate.

### *Application Caching*

Application caching involves caching data within the application's memory or in-memory databases. It is typically used to store frequently accessed data or computation results that are costly to generate or retrieve from other data sources. Application caching improves response times by reducing the need for repeated data retrieval and computation, enhancing the overall performance of the application.

### *Database Caching*

Database caching focuses on improving the performance of database operations by caching frequently accessed data or query results. This caching mechanism can be implemented at different levels: query-level caching and object-level caching.

### *Query-Level Caching*

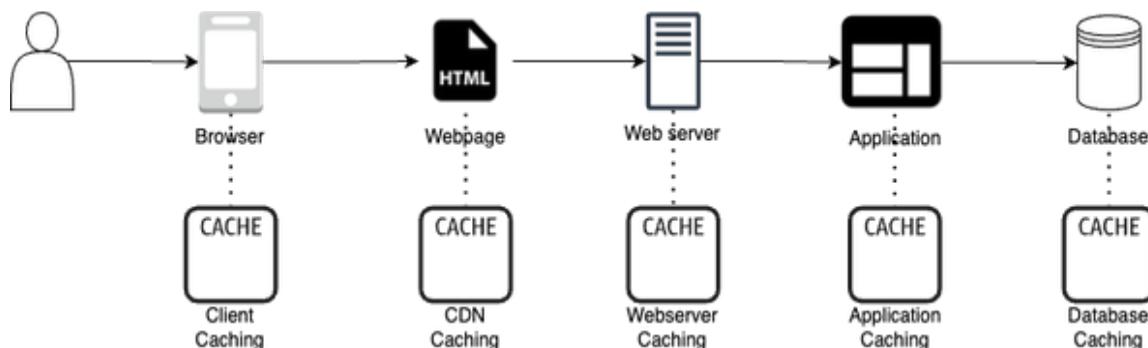
Query-level caching involves storing the results of frequently executed queries in memory. When the same query is

executed again, the cached result is served instead of querying the database again, reducing the database load and improving response times.

### *Object-Level Caching*

Object-level caching caches individual data objects or records retrieved from the database. This mechanism is useful when accessing specific objects frequently or when the database is relatively static. Object-level caching reduces the need for frequent database queries, improving overall application performance.

By employing these caching mechanisms as shown in [Figure 4-3](#), organizations and developers can optimize data retrieval, reduce latency, and improve the scalability and responsiveness of their systems. However, it is essential to carefully consider cache invalidation, cache coherence, and cache management strategies to ensure the consistency and integrity of the cached data.



*Figure 4-2. Caching mechanisms employed at different stages*

Out of the above mechanisms, Content Delivery Networks (CDNs) play a crucial role in improving the performance and availability of web content to end-users by reducing latency and enhancing scalability by caching at edge locations. Let's cover CDNs in detail in the next section.

# Content Delivery Networks

CDNs employ various strategies and architectural models to efficiently distribute and cache content across geographically distributed servers. This section explores different types of CDNs, including push and pull CDNs, optimization techniques for CDNs, and methods for ensuring content consistency within CDNs.

CDNs can be categorized into two main types: push and pull CDNs.

## *Push CDN*

In a push CDN, content is pre-cached and distributed to edge servers in advance. The CDN provider proactively pushes content to edge locations based on predicted demand or predetermined rules. With push CDNs, content is only uploaded when it is new or changed, reducing traffic while maximizing storage efficiency. This approach ensures faster content delivery as the content is readily available at the edge servers when requested by end-users.

Push CDNs are suitable for websites with low traffic or content that doesn't require frequent updates. Instead of regularly pulling content from the server, it is uploaded to the CDNs once and remains there until changes occur.

## *Pull CDN*

In a pull CDN, content is cached on-demand. The CDN servers pull the content from the origin server when the first user requests it. The content is then cached at the edge servers for subsequent requests, optimizing delivery for future users. The duration for which content is cached is determined by a time-to-live (TTL) setting. Pull CDNs minimize storage space on the CDN, but there can be redundant traffic if files are pulled before their expiration, resulting in unnecessary data transfer.

Pull CDNs are well-suited for websites with high traffic since recently-requested content remains on the CDN, evenly distributing the load.

CDNs employ different optimization techniques to improve the performance of caching at the edge server. Let's cover some of these optimization techniques in detail.

### *Dynamic Content Caching Optimization*

CDNs face challenges when caching dynamic content that frequently changes based on user interactions or real-time data. To optimize dynamic content caching, CDNs employ various techniques such as:

#### *Content Fragmentation*

Breaking down dynamic content into smaller fragments to enable partial caching and efficient updates.

#### *Edge-Side Includes (ESI)*

Implementing ESI tags to separate dynamic and static content, allowing dynamic portions to be processed on-the-fly while caching the static fragments.

#### *Content Personalization*

Leveraging user profiling and segmentation techniques to cache personalized or user-specific content at the edge servers.

### *Multi-tier CDN architecture*

Multi-tier CDN architecture involves the distribution of content across multiple layers or tiers of edge servers. This approach allows for better scalability, fault tolerance, and improved content

delivery to geographically diverse regions. It enables efficient content replication and reduces latency by bringing content closer to end-users.

### *DNS Redirection*

DNS redirection is a mechanism employed by CDNs to direct user requests to the nearest or most suitable edge server. By resolving DNS queries to the most appropriate server based on factors like geographic proximity, network conditions, and server availability, CDNs optimize content delivery and reduce latency.

### *Client Multiplexing*

Client multiplexing refers to the technique of combining multiple HTTP requests and responses into a single connection between the client and the edge server. This reduces the overhead of establishing multiple connections and improves efficiency, especially for small object requests, resulting in faster content delivery. Content Consistency in CDNs

Ensuring content consistency across multiple edge servers within a CDN is crucial to deliver the most up-to-date and accurate content. CDNs employ various methods to maintain content consistency, including:

### *Periodic Polling*

CDNs periodically poll the origin server to check for updates or changes in content. This ensures that cached content is refreshed to reflect the latest version.

### *Time-to-Live (TTL)*

CDNs utilize Time-to-Live values, specified in HTTP headers or DNS records, to determine how long cached content remains valid. Once the TTL expires, the CDN fetches updated content from the origin server.

## *Leases*

CDNs use lease-based mechanisms to control the duration of content caching at the edge servers. Leases define a specific time window during which the content remains valid before requiring renewal or revalidation.

### **NOTE**

AWS offers Amazon Cloudfront, a pull CDN offering built for high performance, security, and developer convenience, which we will cover in more detail in Chapter 9 - AWS Network Services.

Before ending the section, let's also understand that using a CDN can come with certain drawbacks also due to cost, stale content and frequent URL changes. CDNs may involve significant costs depending on the amount of traffic. However, it's important to consider these costs in comparison to the expenses you would incur without utilizing a CDN. If updates are made before the TTL expires, there is a possibility of content being outdated until it is refreshed on the CDN. CDNs require modifying URLs for static content to point to the CDN, which can be an additional task to manage.

Overall, CDNs offer benefits in terms of performance and scalability but require careful consideration of these factors and the specific needs of your website. At the end of this chapter, let's dive deeper into two popular open-source caching solutions to understand their architecture and how they implement the caching concepts discussed in the chapter.

## **Open Source Caching Solutions**

Open source caching solutions, such as Redis and Memcached, have gained popularity due to their efficiency, scalability, and ease of use.

Let's take a closer look at Memcached and Redis, two widely adopted open-source caching solutions.

## **Memcached**

Memcached is an open-source, high-performance caching solution widely used in web applications. It operates as a distributed memory object caching system, storing data in memory across multiple servers. Here are some key features and benefits of Memcached:

### *Simple and Lightweight*

Memcached is designed to be simple, lightweight, and easy to deploy. It focuses solely on caching and provides a straightforward key-value interface for data storage and retrieval.

### *Horizontal Scalability*

Memcached follows a distributed architecture, allowing it to scale horizontally by adding more servers to the cache cluster. This distributed approach ensures high availability, fault tolerance, and improved performance for growing workloads.

### *Protocol Compatibility*

Memcached adheres to a simple protocol that is compatible with various programming languages. This compatibility makes it easy to integrate Memcached into applications developed in different languages.

### *Transparent Caching Layer*

Memcached operates as a transparent caching layer, sitting between the application and the data source. It helps alleviate database or API load by caching frequently accessed data, reducing the need for repetitive queries.

Let's take a look at Memcached's architecture.

## Memcached Architecture

Memcached's architecture consists of a centralized server that coordinates the storage and retrieval of cached data. When a client sends a request to store or retrieve data, the server handles the request and interacts with the underlying memory allocation strategy.

Memcached follows a multi-threaded architecture that enables it to efficiently handle concurrent requests and scale across multiple CPU cores. In this architecture, Memcached utilizes a pool of worker threads that can simultaneously process client requests. Each worker thread is responsible for handling a subset of incoming requests, allowing for parallel execution and improved throughput. This multi-threaded approach ensures that Memcached can effectively handle high traffic loads and distribute the processing workload across available CPU resources. By leveraging multiple threads, Memcached can achieve better performance and responsiveness, making it suitable for demanding caching scenarios where high concurrency is a requirement.

In terms of memory allocation, Memcached employs a slab allocation strategy. It divides the allocated memory into fixed-size chunks called slabs. Each slab is further divided into smaller units known as pages. These pages are then allocated to store individual cache items. The slab allocation strategy allows Memcached to efficiently manage memory by grouping items of similar sizes together. It reduces memory fragmentation and improves memory utilization.

When a new item is added to the cache, Memcached determines the appropriate slab size for the item based on its size. If an existing slab with enough free space is available, the item is stored in that slab. Otherwise, Memcached allocates a new slab from the available memory pool and adds the item to that slab. The slab allocation strategy enables efficient memory utilization and allows Memcached to store a large number of items in memory while maintaining optimal performance.

Overall, Memcached's architecture and memory allocation strategy work together to provide a lightweight and efficient caching solution that can handle high traffic loads and deliver fast data access times. By leveraging memory effectively and employing a scalable architecture, Memcached enables applications to significantly improve performance by caching frequently accessed data in memory.

## **Redis**

Redis, short for **R**emote **D**ictionary **S**erver, is a server-based in-memory data structure store that can serve as a high-performance cache. Unlike traditional databases that rely on iterating, sorting, and ordering rows, Redis organizes data in customizable data structures from the ground up, supporting a wide range of data types, including strings, bitmaps, bitfields, lists, sets, hashes, geospatial, hyperlog and more, making it versatile for various caching use cases. Here are some key features and benefits of Redis:

### *High Performance*

Redis is designed for speed, leveraging an in-memory storage model that allows for extremely fast data retrieval and updates. It can handle a massive number of operations per second, making it suitable for high-demand applications.

### *Persistence Options*

Redis provides persistence options that allow data to be stored on disk, ensuring durability even in the event of system restarts. This feature makes Redis suitable for use cases where data needs to be retained beyond system restarts or cache invalidations.

### *Advanced Caching Features*

Redis offers advanced caching features, such as expiration times, eviction policies, and automatic cache invalidation based on time-to-live (TTL) values. It also supports data partitioning and replication for scalability and fault tolerance.

### *Pub/Sub and Messaging*

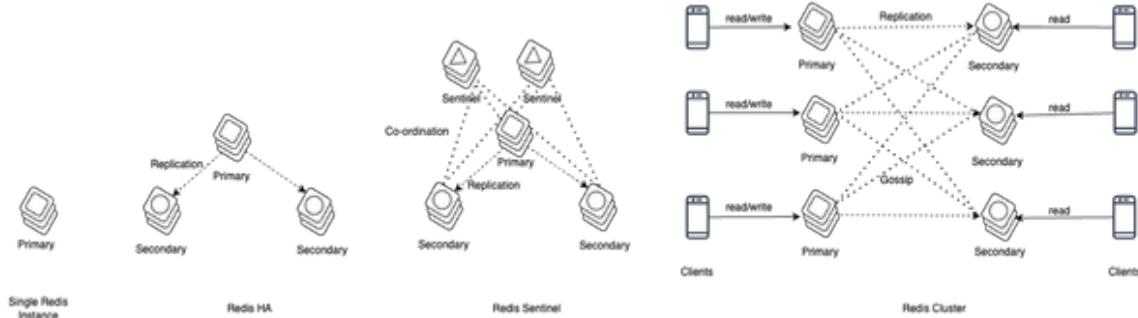
Redis includes publish/subscribe (pub/sub) messaging capabilities, enabling real-time messaging and event-driven architectures. This makes it suitable for scenarios involving real-time data updates and notifications.

Redis serves as an in-memory database primarily used as a cache in front of other databases like MySQL or PostgreSQL. By leveraging the speed of memory, Redis enhances application performance and reduces the load on the main database. It is particularly useful for storing data that changes infrequently but is frequently requested, as well as data that is less critical but undergoes frequent updates. Examples of such data include session or data caches, leaderboard information, and roll-up analytics for dashboards.

Redis architecture is designed for high performance, low latency, and simplicity. It provides a range of deployment options for ensuring high availability based on the requirements and cost constraints. Let's go over the availability in Redis deployments in detail, followed by persistence models for redis durability and memory management in Redis.

## **Availability in Redis Deployments**

Redis supports different deployment architectures as shown in Figure 4-3, including a single Redis instance, Redis HA (High Availability), Redis Sentinel, and Redis Cluster. Each architecture has its trade-offs and is suitable for different use cases and scalability needs.



*Figure 4-3. Redis Deployment Setups*

### *Single Redis Instance*

In a single Redis instance setup, Redis is deployed as a standalone server. While it is straightforward and suitable for small instances, it lacks fault tolerance. If the instance fails or becomes unavailable, all client calls to Redis will fail, affecting overall system performance.

### *Redis HA (High Availability)*

Redis HA involves deploying a main Redis instance with one or more secondary instances that synchronize with replication. The secondary instances can help scale read operations or provide failover in case the main instance is lost. Replication ID and offset play a crucial role in the synchronization process, allowing secondary instances to catch up with the main instance's data.

### *Redis Sentinel*

Redis Sentinel is a distributed system that ensures high availability for Redis. Sentinel processes coordinate the state and monitor the availability of main and secondary instances. They also serve as a point of discovery for clients, informing them of the current main instance. Sentinel processes can start a failover process if the primary instance becomes unavailable.

### *Redis Cluster*

Redis Cluster enables horizontal scaling by distributing data across multiple machines or shards. Algorithmic sharding is used to determine which Redis instance (shard) holds a specific key. Redis Cluster employs a hashslot mechanism to map data to shards and allows for seamless resharding when adding new instances to the cluster. Gossip Protocol is used in Redis Cluster to maintain cluster health. Nodes constantly communicate to determine the availability of shards and can promote secondary instances to primary if needed.

## Durability in Redis Deployment

Redis provides two persistence models for data durability: RDB files (Redis Database Files) and AOF (Append-Only File). These persistence mechanisms ensure that data is not lost in case of system restarts or crashes. Let's explore both models in more detail:

### *RDB Files (Redis Database Files)*

RDB is the default persistence model in Redis. It periodically creates snapshots of the dataset and saves them as binary RDB files. These files capture the state of the Redis database at a specific point in time. Here are key features and considerations of RDB persistence:

#### *Snapshot-based Persistence*

RDB persistence works by periodically taking snapshots of the entire dataset and storing it in a file. The frequency of snapshots can be configured based on requirements.

#### *Efficiency and Speed*

RDB files are highly efficient in terms of disk space usage and data loading speed. They are compact and can be loaded back into Redis quickly, making it suitable for scenarios where fast recovery is essential.

#### *Full Data Recovery*

RDB files provide full data recovery as they contain the entire dataset. In case of system failures, Redis can restore the data by loading the most recent RDB file available.

However, it's worth noting that RDB files have some limitations. Since they are snapshots, they do not provide real-time durability and may result in data loss if a crash occurs between two snapshot points. Additionally, restoring large RDB files can take time and impact the system's performance during the recovery process.

### *AOF (Append-Only File)*

AOF persistence is an alternative persistence model in Redis that logs every write operation to an append-only file. AOF captures a sequential log of write operations, enabling Redis to reconstruct the dataset by replaying the log. Here are key features and considerations of AOF persistence:

#### *Write-ahead Log*

AOF persists every write operation to the append-only file as a series of commands or raw data. This log can be used to rebuild the dataset from scratch.

#### *Durability and Flexibility*

AOF offers more durability than RDB files since it captures every write operation. It provides the ability to recover data up to the last executed command. Moreover, AOF offers different persistence options (such as every write, every second, or both) to balance durability and performance.

#### *Append-only Nature*

AOF appends new write operations to the end of the file, ensuring that the original dataset is never modified. This approach protects against data corruption caused by crashes or power failures.

However, AOF persistence comes with its own considerations. The append-only file can grow larger over time, potentially occupying significant disk space. Redis offers options for AOF file rewriting to compact the log and reduce its size. Additionally, AOF persistence typically has a slightly higher performance overhead compared to RDB files due to the need to write every command to disk.

In practice, Redis users often employ a combination of RDB and AOF persistence based on their specific requirements and trade-offs between performance, durability, and recovery time objectives.

It's important to note that Redis also provides an option to use no persistence (volatile mode) if durability is not a primary concern or if data can be regenerated from an external source in the event of a restart or crash.

## **Memory Management in Redis**

Redis leverages forking and copy-on-write (COW) techniques to facilitate data persistence efficiently within its single-threaded architecture. When Redis performs a snapshot (RDB) or background saving operation, it follows these steps:

1. 1. Forking: Redis uses the fork() system call to create a child process, which is an identical copy of the parent process. Forking is a lightweight operation as it creates a copy-on-write clone of the parent's memory.
2. 2. Copy-on-Write (COW): Initially, the child process shares the same memory pages with the parent process. However, when either the parent or child process modifies a memory page, COW comes into play. Instead of immediately duplicating the modified page, the operating system creates a new copy only when necessary.

By employing COW, Redis achieves the following benefits:

*Memory Efficiency*

When the child process is initially created, it shares the same memory pages with the parent process. This shared memory approach consumes minimal additional memory. Only the modified pages are copied when necessary, saving memory resources.

### *Performance*

Since only the modified pages are duplicated, Redis can take advantage of the COW mechanism to perform persistence operations without incurring a significant performance overhead. This is particularly beneficial for large datasets where copying the entire dataset for persistence would be time-consuming.

### *Fork Safety*

Redis uses fork-based persistence to avoid blocking the main event loop during the snapshot process. By forking a child process, the parent process can continue serving client requests while the child process performs the persistence operation independently. This ensures high responsiveness and uninterrupted service.

It's important to note that while forking and COW provide memory efficiency and performance benefits, they also have considerations. Forking can result in increased memory usage during the copy-on-write process if many modified pages need to be duplicated. Additionally, the fork operation may be slower on systems with large memory footprints.

Overall, Redis effectively utilizes forking and copy-on-write mechanisms within its single-threaded architecture to achieve efficient data persistence. By employing these techniques, Redis can perform snapshots and background saving operations without significantly impacting its performance or memory usage.

Overall, Redis offers developers a powerful and flexible data storage solution with various deployment options and capabilities.

Both Redis and Memcached are excellent open-source caching solutions with their unique strengths. The choice between them depends on specific requirements and use cases. Redis is suitable for scenarios requiring versatile data structures, persistence, pub/sub messaging, and advanced caching features. On the other hand, Memcached shines in simple, lightweight caching use cases that prioritize scalability and ease of integration.

### NOTE

AWS offers Amazon ElastiCache, compatible with both Redis and Memcached for real-time use cases like caching, session stores, gaming, geo-spatial services, real-time analytics, and queuing, which we will cover in more detail in Chapter 10 - AWS Storage Services.

## Conclusion

In concluding this chapter on caching, we have journeyed through a comprehensive exploration of the fundamental concepts and strategies that empower efficient data caching. We've covered cache eviction policies, cache invalidation mechanisms, and a plethora of caching strategies, equipping you with the knowledge to optimize data access and storage. We've delved into caching deployment, understanding how strategic placement can maximize impact, and explored the diverse caching mechanisms available. Additionally, we've touched upon Content Delivery Networks (CDNs) and open-source caching solutions including Redis and Memcached, that offer robust options for enhancing performance. By incorporating Redis or Memcached into your architecture, you can significantly improve application performance, reduce response times, and enhance the

overall user experience by leveraging the power of in-memory caching.

As we move forward in our exploration of enhancing system performance, the next chapter will embark on an exploration of scaling and load balancing strategies. Scaling is a pivotal aspect of modern computing, allowing systems to handle increased loads gracefully. We will also delve into strategies for load balancing in distributing incoming traffic efficiently. Together, these topics will empower you to design and maintain high-performing systems that can handle the demands of today's dynamic digital landscape.

# Chapter 5. Communication Networks and Protocols

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

How do you know what is going on in someone's mind? You ask them. You might simply say, "I believe something is troubling you and sharing it might help to take the burden off your mind." This is a simple exchange of words and we call it communication. There can be different ways to communicate with one another. For example, in this scenario this conversation might have happened in-person, or via a messaging application or a phone call in a language you both understand. We introduced you to the concept of Communication in Chapter 1 in very brief and we'll extend the idea of communication in this chapter in much more detail.

The Internet evolved over time and a lot of solutions are built to solve new problems we encounter while communicating over the internet. It is important that there are guidelines or mechanisms

ensuring people and machines can work with each other. We'll start our discussion with communication models to strengthen the understanding of communication over the internet following certain rules referred to as protocols. You might have seen a lot of keywords floating around such as TCP, XMPP, WebSockets, WebRTC, HTTP, GraphQL, REST, SOAP, etc. and it can be very confusing trying to determine what these terms mean and in which scenarios they should be used. This chapter will help you gain understanding of communication related terms and we'll try to identify the right scenarios for using a particular technology.

## **Communication Models and Protocols**

The first thing that you need in order to communicate with another person is a common language, which you both can understand and use to convey your messages. This common language has a set of rules, generally referred to as grammar and these rules define how you speak and how you write. A similar analogy applies in machines—for machines to communicate over the internet, they follow certain rules referred to as communication protocol. There are different communication protocols operating at different layers in the communication model. The layers in a communication model are representation of how data travels across the internet from one location to another in a reliable way. The layers also ensure data transformation from one form to another so it can be understood by both humans and the machines. We'll talk about two such models, named Open Systems Interconnection (OSI) and Transmission Control Protocol/ Internet Protocol (TCP/IP) and then discuss the protocols.

### **OSI Model**

OSI Model divides the communication network into seven layers with each layer responsible to perform a specific job(s). The OSI model is

a reference model where the top layer is an application layer (Layer 7) and the bottom most layer is a Physical Layer (Layer 1). Layer 7 corresponds to data on how humans perceive such as plain English language and Layer 1 corresponds to data on how machines perceive such as binary format. For any communication to happen, the data travels through OSI model layers from top to bottom and bottom to top as shown in Table 6-1.

**NOTE**

We'll dive deep into the example protocols in Table 6-1 in follow up sections.

T  
a  
b  
/e  
5  
-  
1  
.O  
S  
I  
M  
o  
d  
el

---

<b>Layer Number</b>	<b>Layer Name</b>	<b>Layer Function</b>
---------------------	-------------------	-----------------------

---

Layer 7	Application Layer	<p>Layer 7 has direct access to the user data and the software application relies on it via protocols such as HTTP, SMTP, SSH, etc.</p> <p>This layer is also helpful in service advertisement, such as which services are available over the network to connect.</p>
---------	-------------------	---

---

Layer 6	Presentation Layer	<p>This layer is responsible for formatting, encryption, decryption and compression of data.</p> <ul style="list-style-type: none"><li>• Formatting ensures that Layer 6 is able to format the data as expected by Layer 7.</li><li>• If the data is in encrypted format, Layer 6 decrypts the data before passing on to Layer 7 or encrypts the data before passing on to Layer 5 (if encryption is a requirement).</li><li>• To reduce the data footprint, Layer 6 compresses the data before passing on to Layer 5.</li></ul>
Layer 5	Session Layer	<p>This layer is responsible for session lifecycle— session initiation,</p>

session maintenance and session termination.

---

Layer 4	Transportation Layer	<p>This layer is responsible for data transport between the devices via protocols such as TCP and UDP. It additionally helps with data buffering, error control and windowing.</p> <ul style="list-style-type: none"><li>• It figures out an optimal speed for data communication between sender and receiver so that a receiver with slow connection receives the data at required speed.</li><li>• The extra data can be buffered in a buffer queue so it can be delivered at speed needed. In case the buffer queue is full, the new data segments are dropped, referred to as tail drop.</li><li>• The layer can request data retransmission if any data</li></ul>
---------	----------------------	--

segments are lost to ensure reliable connection in case of TCP protocol.

- The Windowing feature identifies the amount of data that can be transferred before expecting an acknowledgement from the receiver.

Layer 3	Network Layer	This layer is responsible for data transfer between the networks with the help of logical addressing such as IP addresses. Layer 4 breaks down the data into smaller chunks before passing on to the Layer 3 referred to as data packets.
---------	---------------	---

Layer 2	Data Link Layer	This layer helps in data transfer between the devices from incoming network interface to the outgoing network interface in the form of frames (packets are further broken into data frames). This layer has two parts, Media Access Control (MAC) and Logical
---------	-----------------	---

## Link Control (LLC).

- MAC helps with physical addressing of devices involved in the network communication. The MAC address of any device is globally unique and is helpful in device identification for data delivery.
- The LLC helps with flow control and error control. The flow control and error control in this layer is inside the same network as compared to Layer 3 where it was across the networks.

Layer 1	Physical Layer	The data frames from Layer 2 are converted to bit streams (in the form of 1s and 0s) in Layer 1. This layer involves the physical devices such as cables and switches.
---------	----------------	--

In modern age internet applications, we don't see an OSI model being practically present and it only serves as a reference communication model. The internet communications happen via TCP/IP model in general. The TCP/IP model layers don't match one-to-one with OSI model layers but there are quite a few similarities.

## **TCP/IP Model**

The TCP/IP model is also referred to as the Internet Protocol suite which mainly includes TCP, IP and UDP (User Datagram Protocol, to be discussed shortly). In comparison to the OSI model, the TCP/IP model combines multiple layers into one layer as shown in Table 6-2 to execute similar functionality as you learned in Table 6-1.

*T  
a  
bl  
e 5  
- 2  
. D  
if  
fe  
r  
e  
n  
t  
la  
y  
e  
rs  
o  
f  
O  
S  
I  
a  
n  
d  
T  
C  
P  
/I  
P  
m*

*o  
d  
e/*

---

**OSI model  
layer****TCP/IP model  
layer****TCP/IP protocol  
examples**

---

Application, Presentation, Session      Application      SMTP, HTTP

---

Transport

Transport

TCP, UDP

---

Network

Internet

IP, ICMP

---

Data Link

Data Link

IEEE 802.2

---

Physical

Physical Network      **Ethernet, Token Ring**

---

## NOTE

We recommend going through [RFC documentation](#) for more dive deep into each of the protocols discussed in this chapter. You can search in your favorite search engine with keywords something like “TCP protocol RFC documentation”.

We've linked the [TCP documentation](#) for reference.

There are different protocols which operate at different layers in the communication models. We'll discuss the most general ones to help you gain an understanding of how communication over the internet works, starting with the Network layer protocols such as IP, and also discuss how different protocols are used for specific communication purposes.

## Network Layer Protocols

The most commonly used protocol in the Network/Internet layer is Internet Protocol (IP). It is a set of rules responsible for the delivery of data packets from a source IP address to a destination IP address. We'll discuss more about IP addresses and their types in Chapter 9. The IP address uniquely identifies the source and destination of data packets across the network and helps routers to route this information from one place to another. Data across the internet is often broken down into chunks (referred to as fragmentation) to avoid the burden of delivering large packets along with reduced latency. Each packet in IP protocol contains IP header and payload. The IP header contains source IP address, destination IP address along with any additional metadata (such as IP version, packet size, time to live) required to route the packet. The payload consists of actual data (referred to as IP datagrams) that is to be transferred over the network.

## NOTE

Chapter 9 of this book, AWS Network Services, will give you a basic understanding of IP addresses and how they help with unique identification of devices over the internet.

Another protocol which operates on this layer is Internet Control Message Protocol (ICMP). This protocol is generally used for network diagnostics and implementing error mechanisms. For example, to check if Google servers are responding to the traffic, we can use the command—`ping google.com` (command output included below as reference). Let's say the `google.com` URL is not resolved to a server, the router can respond back with an ICMP message to the sender.

```
PING google.com (142.250.74.206) 56(84) bytes of data.  
64 bytes from fra24s02-in-f14.1e100.net (142.250.74.206): icmp_seq=1 ttl=116  
time=9.97 ms  
64 bytes from fra24s02-in-f14.1e100.net (142.250.74.206): icmp_seq=2 ttl=116  
time=16.5 ms  
64 bytes from fra24s02-in-f14.1e100.net (142.250.74.206): icmp_seq=3 ttl=116  
time=16.9 ms
```

The network layer protocols in general work in conjunction with transport layer protocols such as TCP and UDP for data delivery, which we'll focus on in the following sections.

## Transmission Control Protocol

The Transmission Control Protocol (TCP) is a transport layer protocol and offers reliable communication between sender and receiver with capability to retransmit data in case of packet loss. The communication via TCP is initiated between sender and receiver via a 3-way handshake process as shown in [Figure 5-1](#). The 3-way handshake process is an agreement between the two parties to send and accept the data. The TCP header includes information such as source & destination port number, sequence number (identifies how much data is sent during a session), acknowledgement number

(useful at receiver end to send acknowledgement and request for next segment), window (number of bytes receiver can accept from sender), etc.

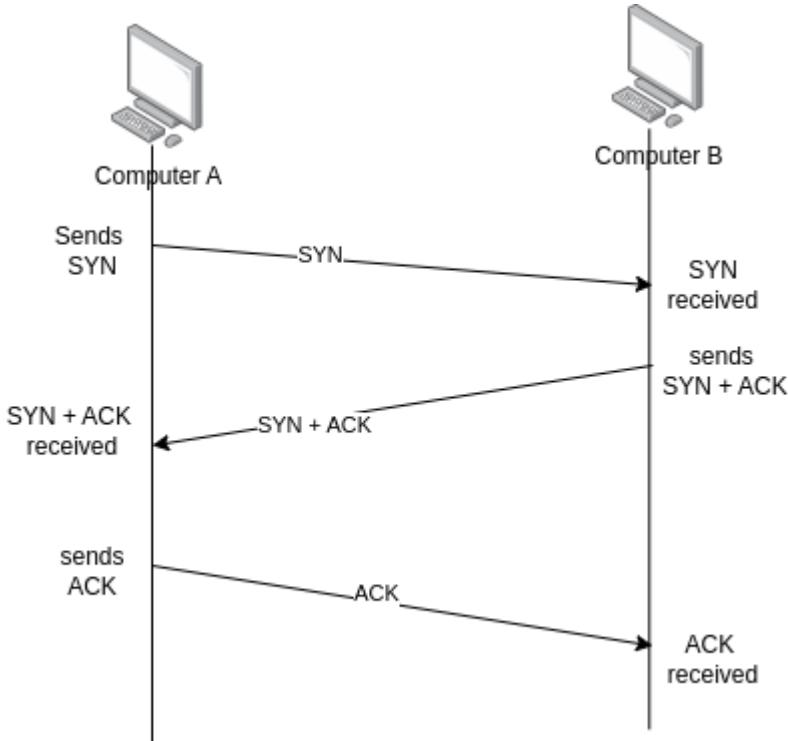


Figure 5-1. TCP 3-way handshake process

### NOTE

A port is a virtual point managed by an Operating System which defines an entry or exit to a software application (numbered from 0 to 65535). You can compare them to your personal computer ports, such as USB ports. Each port is responsible for certain types of network communication; for example, HTTP operates over port 80.

The additional work, such as establishing communication or retransmitting lost packets, comes with extra cost of latency. If there are no such requirements, we can go with another transport layer protocol, referred to as User Datagram Protocol (UDP).

## User Datagram Protocol

The UDP, another transport protocol, is less reliable, but faster, in comparison to the TCP. It is faster because it doesn't involve a 3-way handshake process for connection establishment and provides no guarantee of data delivery, so there is no overhead of retries. The UDP header includes information such as source & destination port, length, UDP checksum and data. The UDP header is not as heavy as TCP as it holds minimum data for segment delivery and further it doesn't provide assurance of order of segments (data received at receiver might not be in the same order as sent by sender). UDP is an ideal candidate for use cases such as voice or video calls over the internet because it's fine if some of the UDP segments are dropped in between, but we expect the communication in real time so latency factor is critical. However, not having a 3-way handshake between sender and receiver to start communication can also become a security threat. To illustrate how this works, let's discuss how the communication via UDP works:

- A sender sends a UDP segment to a destination on a specific port.
- The receiver checks if the request is expected by any application on that port. If yes, then the request is consumed by application for further processing.
- If not, the receiver responds with an ICMP packet mentioning the destination is not reachable.

Now, assuming the above processing happens at a large scale, some bad actors intentionally bombard UDP requests to the server and server resources are entirely consumed in responding back to the requests, causing Denial-of-service to actual traffic as well. There are mechanisms we need to place in our system to mitigate these kinds of issues. One way could be ignoring these requests after a configured threshold by not responding with ICMP messages. The

downside of this approach could be dropping of actual customer requests, but at-least we'll be able to keep our system healthy. Another way could be leveraging third party applications offered by cloud providers such as [AWS Shield](#).

Both TCP and UDP serve their purposes and you should choose based on your own business requirements. We prefer TCP for reliability and full-duplex (bi-directional communication) and UDP for faster response time and unidirectional communications.

The layer operating on top of the network layer is the application layer in the TCP/IP model. Let's dig deeper into a few application layer protocols starting with HyperText Transfer Protocol (HTTP).

## HyperText Transfer Protocol

The world is heavily dependent on the internet for day to day work and most of this work happens on the web browsers or any software applications running on mobile devices. Communications over the internet generally happen in client-server model where client requests some information (HTTP request) and the server responds with data (HTTP response) corresponding to the request on port 80 (and port 443 for HTTPS, a secure version of HTTP). Let's take an example HTTP request responsible for retrieving customer orders based on order Id to gain more familiarity with how the protocol works.

```
GET /api/orders?id=23234555&customerId=dhfsd348e48ddd HTTP/1.1
Host: api.myFoodApp.com
Authorization: Bearer MyAccessToken
User-Agent: MyOnlineFoodOrderApplication/1.0
Accept: application/json
HTTP/1.1 200 OK
Content-Type: application/json
{
  "orderId": "23234555",
  "customerId": "dhfsd348e48ddd",
  "amount": 199,
  "currency": "INR",
  "items": [
```

```
{  
    "product": "Paneer Tikka Masala",  
    "quantity": 1,  
    "price": 119  
},  
{  
    "product": "Tandoori Roti Butter",  
    "quantity": 3,  
    "price": 80  
}  
],  
"address": "House 740, 1st Block, Koramangala, Bengaluru",  
"status": "Delivered"  
}
```

### NOTE

The request responses included in this chapter are for illustration purposes only and were not tested in the production environment.

Going deeper into the HTTP request, it contains details such as HTTP method (GET in this case), version (1.1 in this case), host specifying where the request is being routed to with required authentication mechanism. We can specify different HTTP methods based on the type of call to the server—below are the widely used methods:

#### *GET method*

is used to retrieve data without any state modification inside the system (ensures idempotent nature). For example, retrieve order details based on order Id and customer Id.

#### *The POST method*

is used to create a new resource on the server or send any data for processing. It is non-idempotent in nature. For example, place a food order and complete the transaction.

#### *The PUT method*

is also idempotent in nature and used to update any resource on the server. For example, update the address after placing an order.

### *The DELETE method*

is idempotent in nature and used to remove any resource from the servers. For example, delete a saved address entry.

Next to the HTTP method is the request target specifying the location on the host (server) that the client is trying to access. HTTP has evolved over time and the version represents which particular specification the application is using. The older versions are deprecated and the most used version these days are HTTP/1.1, HTTP/2 and HTTP/3.

### *The HTTP/1.1*

is the most widely used version and offers better support to previous versions such as reuse of TCP connection to make multiple requests.

### *The HTTP/2*

version optimizes the header to be available in compressed format (lesser bandwidth as compared to HTTP/1.1), single TCP connection for a domain, and server push capability to avoid any polling from clients for asynchronous calls (we will elaborate more in upcoming sections).

### *HTTP/3*

improves on the speed as compared to HTTP/2 by using UDP instead of TCP/IP protocols, also referred to as **QUIC** (Quick UDP Internet Connections) and therefore avoids occasional TCP/IP congestion problems by implementing a congestion control algorithm.

There can be some additional metadata as part of request, referred to as HTTP header. It can include information such as User-Agent (where the request is being made from, such as software application, web browser name), Accept (the type of content format expected from the server), etc. Now, taking a look at the HTTP response, it includes the response to the request (order details for order Id and customer Id). Alongside the response object, it also has the HTTP status code. The status code represents what happened with the request, for example, 200 OK means the request was successfully processed by the server. The status codes are represented from 100 series to 500 series as shown in Table 6-3.

T  
a  
b  
/  
e  
5  
-  
3

.  
H  
T  
T  
P  
r  
e  
s  
p  
o  
n  
s  
e  
s  
t  
a  
t  
u  
s  
c  
o  
d

---

## **Status Code Series   Status Code Meaning & Examples**

---

100	<p>For informational purposes such as noting that the server has received the request and it is in processing.</p> <p style="text-align: center;">100 – Continue 101 – Switching Protocols</p> <hr/>
200	<p>Note that the request is successfully processed by the server.</p> <p style="text-align: center;">200 – OK with the response associated with the corresponding request. 201 – The request is processed by the server and a new resource is created.</p> <hr/>
300	<p>This means that the server has redirected the request to another resource for fulfilling it.</p> <p style="text-align: center;">301 – This means the resource has moved permanently to a new location and the client should update the references on its end. 302 – This means the resource has temporarily moved to a new location.</p> <hr/>

400	<p>The 400 series of codes means there was an issue in the client request and it should be rectified before sending it to the server.</p> <p>400 – The request is not as expected for the server request target.</p> <p>401 – The request doesn't contain valid credentials for authentication.</p> <p>403 – The client is not authorized for the request.</p> <p>404 – The requested resource is not available on the server.</p>
-----	--

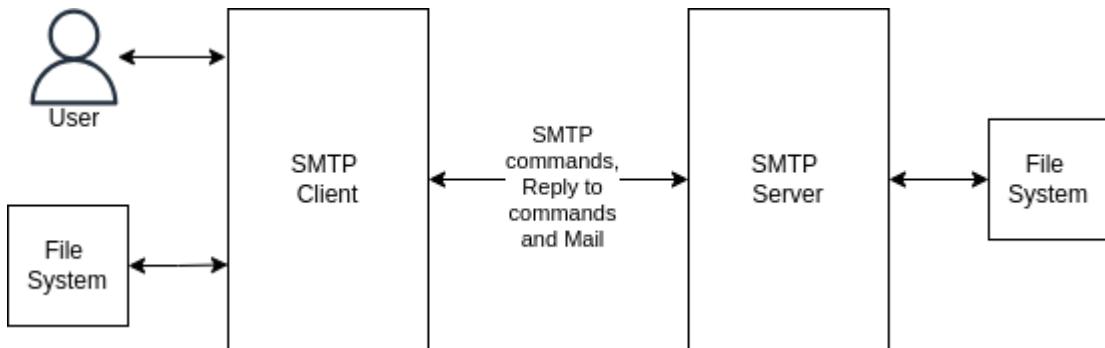
500	<p>This series of codes means the server encountered an error while the request was being processed.</p> <p>500 – The server encountered some unexpected exception while processing the request.</p> <p>503 – The server is temporarily unavailable to take the client requests.</p>
-----	--

While HTTP is a general protocol for application communications over the internet, there are some protocols designed to solve specific use cases, such as Simple Mail Transfer Protocol (SMTP) for email communications. Let's dig into that now.

## Simple Mail Transfer Protocol

The software applications interact via application layer protocols such as Simple Mail Transfer Protocol (SMTP) for email services and further the communication flow through other layers in the TCP/IP model. The SMTP protocol is useful for email communications

between two or more parties. Let's take a look at an example (as shown in [Figure 5-2](#)) of how a person can send an email to you via different mail agents along the way.



*Figure 5-2. Email communication via SMTP*

The SMTP client identifies the mail domain (such as gmail.com, hotmail.com, etc) and then establishes a two way stateful transmission channel to the corresponding SMTP server(s) to transfer mail. The connection is initiated via SMTP commands and the server replies to corresponding commands with success or failure. The SMTP server can be the final destination of mail delivery or it can be an intermediate server to forward the mail further. In case the sender and receiver belong to different domains (for example, email from gmail.com to hotmail.com), the intermediate server is referred to as a Relay server.

Let's consider an example of how physical mail delivery works to help you understand it better; you've reached the local post office in town and posted two letters—one to be delivered in your own town and other to be delivered to a friend in a nearby town. The first letter can be delivered without involving any other post office but in the case of the second letter, the letter is transferred to the post office of a nearby town and then finally delivered to your friend. We are required to involve another post office because our delivery agents can't possibly directly go to other towns to deliver letters and hence we require another post office in between, or, in the case of SMTP, the relay servers to bear this responsibility.

The SMTP server can also behave as a gateway, meaning the further transport of mail is carried out with help of other protocols and not via SMTP. The SMTP clients and servers are also referred to as Mail Transfer Agents (MTAs) as they offer the mail transfer service. The SMTP **commands** are used for communication between the parties and replies are the success or a failure acknowledgement corresponding to the commands sent.

We often add additional media to emails as attachments, which is not inherently supported by SMTP. For these purposes, we use something called Multipurpose Internet Mail Extensions (MIME). The MIME protocol makes SMTP extensible to support media attachments and conversion of non-ASCII characters to 7-bit ASCII characters (English language alphabets and numbers). Recall that we described ports very briefly in the Transmission Control Protocol section—note that the SMTP operates on port 25 for plaintext communications and port 587 for encrypted mail communications.

The SMTP essentially operates on a push mechanism, transferring email contents from clients to the servers. There are two other protocols, Post Office Protocol (POP) and Internet Message Access Protocol (IMAP) which can specifically work on the pull mechanism, retrieving the email from servers to send to the user devices. POP connects with the email servers, downloads the content to the local machine and deletes the emails from the server, while IMAP directly reads from servers without downloading content to the local machine or deleting the mails after reading. Both of these protocols have their own set of benefits: POP is fast because the message is pre-downloaded for reading, but since the messages are deleted from the servers, we can't read the same emails on different devices. IMAP allows us to read emails on any number of devices, but it is not as fast as POP.

We've discussed protocols designed specifically for email communications, but email communications are not near real time.

Next we'll move to Extensible Messaging and Presence Protocol (XMPP) offering near real time communication over the internet.

## Extensible Messaging and Presence Protocol

The XMPP is an XML (Extensible Markup Language) based instant messaging protocol with additional features such as presence information (noting when a user is online or offline) and contact list (roster). XMPP helps keep communication near real time between users by streaming XML stanzas (a fundamental unit of communication in XMPP with message and recipient information) over a persistent TCP or HTTP connection. The XMPP can be operated in a more efficient manner by using websockets as a sub protocol (we'll discuss this shortly) instead of using HTTP binding, as websockets are inherently built for bi-directional communications. This protocol can operate in asynchronous fashion as well which essentially means the recipient user doesn't have to be online. This can be helpful in use cases such as notification broadcasting.

The lifecycle of an XMPP connection starts with the client initiating a TCP connection with a server on port 5222 for plain-text and 5223 for encrypted communications. The next steps are:

- Send stream header to start the negotiation process for stream features such authentication mechanisms, encryption methods, etc.
- After negotiation, the client shares credentials for authentication depending on whether it was agreed upon during the negotiation process.
- The next step is client binding to a resource. This helps in unique identification with use of a Jabber Id, which we'll explain in a moment.
- After these steps, a session can be established (if needed) by sending a session request to a server by the client.

- Once a session is established, the client shares the presence information with the server. Now the client can request a roster and then continue with sending or receiving messages via XML stanzas.
- Similar to any other protocol, the connection is terminated at the end of the session once requested by the client and acknowledged by the server.

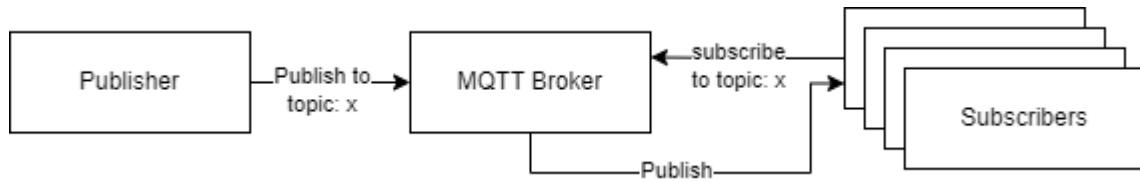
XMPP is a decentralized protocol—meaning anyone can host XMPP servers—and works similarly to SMTP. Every user has a unique Id referred to as Jabber Id (JId—note that XMPP was previously known as Jabber) similar to an email address. The JId is denoted by a unique Id along with the domain name—for example, myUserId@example.com. The user may log in from multiple devices, which can be represented by specifying the resource identifier, such as myUserId@example.com/mobile. The XMPP is a widely used protocol for chat applications—for example, you might have used or heard about the **WhatsApp** chat application. WhatsApp is built on top of **ejabberd**, an open source software written in **Erlang** programming language that operates over XMPP.

**MQTT** (Message Queuing Telemetry Transport) is another real time protocol which we'll discuss next.

## Message Queuing Telemetry Transport

As the name suggests, MQTT was originally designed for applications to send telemetry data to and from space probes using minimum resources. MQTT protocol is heavily used for IOT (Internet Of Things) devices communication and it can be used to support messaging applications as well. MQTT is based on the publish/subscribe model, users can send messages to a topic (publish) and receive them by subscribing to particular topics. The publishers and subscribers are completely independent of each other

and the publish-subscribe actions are facilitated by MQTT brokers as shown in [Figure 5-3](#).



*Figure 5-3. MQTT architecture*

MQTT supports bi-directional communications between the devices and the cloud and offers multiple [QoS](#) (Quality of Service) levels for ensuring message delivery guarantees. The QoS levels are defined separately by the publisher (to publish a message to a topic) and the subscriber (to consume a message from a topic). Below are the three QoS levels supported by MQTT:

#### *At most once— QoS Level 0*

QoS 0 is based on the fire and forget principle. The sender publishes a message only once (no retransmission) and the consumer doesn't acknowledge receiving a message for confirmation.

#### *At least once— QoS Level 1*

QoS 1 ensures the message is delivered to the subscriber at-least once so there is a possibility of duplicate message and the subscriber should appropriately handle it. The publisher waits for an acknowledgement of message delivery and retransmits the message if not received in a certain timeframe.

#### *Exactly once— QoS Level 2*

QoS 2 guarantees exactly once message delivery to the recipients. This is achieved by four-part verification between publisher and subscriber for every message (thus reducing the efficiency).

We can select the QoS levels based on the requirement of efficiency and reliability. The QoS 0 is most efficient but least reliable while QoS 2 is most reliable but least efficient.

Protocols are essentially a set of rules to establish communication between the involved parties. In the next section, we'll discuss the different communication types available over the internet and how we can use them as necessary for our business needs.

## Communication Types

We briefly touched upon the general idea of synchronous (sync) and asynchronous (async) communication in Chapter 1. In synchronous communication between two parties, the first party requests some data and then waits for the response from the second party. In the case of asynchronous communication, the first party requests some data but doesn't wait for the response. The second party can respond in the near future via events or any other mechanism. Both sync and async modes of communications have their own set of challenges and benefits and it's very important that we identify when to use which for our business use cases.

Sync communications are generally preferred for low latency calls—we're aware that the server will return the response for the corresponding request in the expected timestamp and won't take too long. The whole premise is that if we already know that the request is going to take time for processing, then we're blocking a system resource at the client's end and not performing any actual work. Please note that using async mode doesn't mean the system is latency heavy—the request can be processed as soon as it is consumed by the service here as well. The fundamental difference lies in the method of communication.

With async calls, the response is not immediately returned for a request nor is it returned over the same connection. There are two different ways to receive a response in the near future—either the

client sends a request to the server again after some time, requesting a response, or the server delivers the response without the client needing to request it. The first mechanism is referred to as a pull based mechanism and second is called a push based mechanism.

There can be multiple methods of communication in the pull and the push based mechanisms and we'll discuss them one by one starting with the pull based approaches, namely HTTP regular polling and HTTP long polling.

## **Pull Mechanism: HTTP Polling**

We'll take an example of an online food ordering platform to understand different kinds of pull mechanisms. Once the order is placed, we want to know the status of the order, such as order preparation ongoing, delivery agent assigned, delivery agent has picked up food, location of delivery agent and finally food being delivered at our doorstep. Table 6-4 shows different approaches the client application can take to the server application to know the latest status of order.

T  
a  
bl  
e  
5  
-  
4.  
P  
u/  
l  
M  
e  
c  
h  
a  
n/  
s  
m  
a  
p  
p  
r  
o  
a  
c  
h  
e  
s

---

<b>Approach</b>	<b>Approach Considerations</b>
-----------------	--------------------------------

---

## HTTP Regular Polling

1. The client asks the server for recent status at regular intervals, such as 5 seconds.
  2. The server checks the status and responds back.
  1. Clients regularly invoke an API and might not get a response, wasting network bandwidth.
  2. We don't recommend this approach for production workloads as it will consume unnecessary resources.
- 

## HTTP Long Polling

1. Clients ask the server for recent status at regular intervals, such as 1 minute.
1. It requires less server calls compared to HTTP regular polling reducing overall resource consumption, hence the approach is an improvement over HTTP regular polling.

2. The server only responds if there is a change in status, otherwise it holds the connection for a fixed interval, such as 1 minute.
2. We recommend implementing exponential strategy in case a response is not received in the first call. For example, you might have seen how Gmail responds when there's no internet connection. First it tries to connect within 1 second, then 5 seconds, then 30 seconds and so on.
3. Client reestablishes the connection after this interval to check for the most recent status.

The next set of approaches that we're going to discuss are push based mechanisms, starting with the [websockets](#). It essentially means clients don't ask for a response, it's the server's responsibility to send a response once the results are ready.

## **Push Mechanism: WebSockets**

In HTTP long polling, the connection is terminated after a certain time and then the client re-initiates a connection, but what if we require a persistent connection? WebSockets represent a bi-directional persistent connection (full-duplex communication on a single TCP connection) over HTTP where clients and servers communicate with each other. Once the connection is established between client and server, the messages can flow to and fro. A

popular example for these kinds of requirements is a chat application. Two people (or more in case of user groups) interact with each other in real time, so a persistent connection makes more sense here as compared to client applications asking for a response again and again. The WebSocket connection follows a similar flow to the TCP. Any connection is started with a handshake process and then the actual data transfer happens.

The handshake process before the actual data transfer ensures the agreement between the client and the server for bi-directional communication. The first step is that the client initiates a HTTP GET call (as shown in code snippet below) to the host server with Connection type set as Upgrade and Sec-WebSocket-Key (random **base64**-encoded key) for ensuring security.

```
GET ws://myChatApp.com/chat HTTP/1.1
Host: myChatApp.com
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: q4xkc032u266gldTuKaS0w==
```

Once the server receives this request, it responds with HTTP status code 101, meaning the protocol is switched from HTTP to websocket. This is shown in the code snippet below. It also responds with Sec-WebSocket-Accept (created by server by appending predefined GUID (258EAFA5-E914-47DA-95CA-C5AB0DC85B11 as defined in [RFC 6455](#)) to Sec-WebSocket-Key, hashing it to **SHA-1** algorithm and encoding it to base64).

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: skjfdPPEKjdksejsl+sdr=
```

On receiving the server acknowledgment, the websocket connection is established for the client and server to communicate with each

other. The communication messages can be textual or binary format following a frame structure. A frame should always be masked before sending to the server—if not, the connection is closed by the server upon receiving an unmasked frame. The frame consists of payload data and additional metadata such as whether the frame is masked or not (denoted by 1 or 0), payload length, and masking key. The payload data is a combination of extension data and application data. The application data consists of the original message and extension data is an optional field which can help achieve additional capabilities such as encryption, compression, etc. Finally, the connection between client and server is torn down at the end of the communication.

Now, we might not always have a requirement for bi-directional connection. We may just require a communication channel from server to clients. We'll discuss this idea in the next section where we elaborate about Server Sent Events (SSE).

## **Push Mechanism: Server Sent Events**

Let's go back to our example of how we can check the status of an order we placed on an online food ordering application. The SSE can be a feasible approach in this case as the server is responsible for updating the order status and there is no as such input required from the client's end. The server is not dependent on any message delivery acknowledgements from clients—if there is such a requirement, SSE might not be the best fit for that use case. The SSE is implemented over a long lived HTTP connection to consume any updates or notifications from the server in text format. We define an EventSource in a client application which includes the URL of the SSE event stream on the server and any new event is delivered to this eventsource via the event stream.

We discussed in Chapter 1 that the network is not reliable and due to this unreliability, a long lived HTTP connection can always break. In these kinds of scenarios, we need the capability to reestablish the

connection from where we left off so as to continue consuming the events from the server. SSE has this capability built-in—it can reestablish dropped connections and recover any messages that might have been missed by the clients during the connection issue.

We mentioned that responses in case of async architectures can be delivered via websockets or SSE. Yet another way could be publishing a response to a topic which clients have subscribed to. We'll elaborate on this idea in Chapter 8 while discussing publisher subscriber architectures and in Chapter 12 while discussing AWS services, SQS and SNS.

Up to this point, we've discussed different networking protocols and mechanisms for communications over the internet. With networking protocols, there are common communication standards (protocols or architectural styles) that are generally followed in the client-server architecture. Let's dig deeper into this.

## **Common Communication Protocol Standards**

In the client-server architecture, the clients interact with servers to retrieve some information or perform an action on the server. Consider a scenario where each client and server define their own paradigms for interaction over the internet, it becomes very difficult for software applications to adapt and implement a vast set of paradigms. This can be easy if the client-server calls are commonly defined and everyone uses them. In this section, we'll explain the concept of Application Programming Interface (API) and its multiple types to identify how they suit particular business use case requirements. The API enables two software applications to communicate over the internet using a set of definitions and protocols. We'll start off our discussion with Remote Procedure Call (RPC) and then explore other paradigms.

## Remote Procedure Call

Typically a single machine can't hold all the necessary implementation logic and part of the code is hosted somewhere else, which may or may not be owned by your company. Looking at an example of fulfilling a food order on an online food ordering application, the frontend application calls backend service to place an order, the backend service talks with the payment service to process the payment in the desired payment mode, the payment service talks with external providers such as a bank or a payment gateway (PG) to process a payment ,and finally the order is placed. Now, consider the interaction of our own payment service with a PG —the PG is not owned by us but we invoke the payment processing method similarly to how we call any local method in our code base. This is referred to as Remote Procedure Call (RPC).

A procedure is a block of code responsible to execute a specific set of task(s)—it can be as simple as a program of adding two numbers or in this case, processing a payment via PG. RPC means executing this piece of code, aka procedure, on a remote machine as if it is executed on your local personal computer or laptop. Now, let's understand why we need such a mechanism and we'll discuss how this remote code execution works.

If the code is hosted on a remote machine, the one definite benefit we get is code reusability and scalability. The code can be shared among multiple clients offering abstraction to the clients on the code execution and it is scalable in a way that the code can be distributed among multiple machines. It additionally abstracts out the network communication on how the local machine interacts with the remote machine to execute a procedure, for a developer it's similar to any local procedure call. There can be multiple teams in an organization and they might be owning up hundreds of microservices written in different programming languages. We define the interfaces and RPC helps in cross language and platform support by taking the

headache of data marshaling (encoding) and unmarshalling (decoding) on its plate.

Moving on to how RPC works, recall that while discussing the benefits of RPC, we mentioned that the different microservices might be written in different programming languages so it is not possible to directly invoke the remote function. We need an intermediary which can handle this conversion logic —this is referred to as stub. A Stub function offers a similar interface as a remote method which clients can invoke and it internally handles the implementation details, such as network communication and marshaling of method arguments to a message over the network. On the server side, the stub function will unmarshal the message to the arguments as expected by remote method, and once the method processing is complete, the method response is again marshaled and sent back over the network where the client side stub function unmarshal to object readable by client machine.

The common interface or a contract to specify the remote procedures and data types are defined in the Interface Definition Language (IDL). This interface is language agnostic, meaning it will work with any desired programming language. The IDL is used to generate the stubs used for client communication. Some examples include Web Services Description Language (WSDL) used with Simple Object Access Protocol (SOAP), Apache Thrift IDL, etc.

While there are benefits, there are also downsides, since we're operating in a distributed environment. Some of the issues we can face here, which we don't have to consider with local method invocation, are network latency, network failures, handling retry mechanisms for network calls, ensuring the interfaces are compatible as systems evolve, etc. Some examples of RPC include **SOAP**, Representational State Transfer (REST), **gRPC**, and **Thrift**. Let's explore SOAP in the next section.

## Simple Object Access Protocol

As the name dictates, SOAP is a protocol that offers mechanisms to clients and servers to communicate with each other using XML messages. SOAP is not tied to any specific protocol and can be used alongside other protocols such as HTTP, SMTP, TCP, etc. though it is mostly used along with HTTP. We'll take an example of order status retrieval from an online food ordering application via SOAP over HTTP to understand the terminology associated with this protocol.

```
POST /order/status HTTP/1.1
Host: api.myFoodApp.com
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <soap:Header>
        <AuthorizationHeader xmlns="http://api.myFoodApp.com/headers">
            <Token>MyAuthToken23456</Token>
        </AuthorizationHeader>
    </soap:Header>
    <soap:Body>
        <GetFoodOrderStatusRequest xmlns="http://api.myFoodApp.com/foodservice">
            <OrderId>23234555</OrderId>
            <CustomerId>dhsd348e48ddd</CustomerId>
        </GetFoodOrderStatusRequest>
    </soap:Body>
</soap:Envelope>
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <soap:Body>
        <GetFoodOrderStatusResponse xmlns="http://api.myFoodApp.com/foodservice">
            <OrderId>23234555</OrderId>
            <CustomerId>dhsd348e48ddd</CustomerId>
            <Status>DELIVERED</Status>
        </GetFoodOrderStatusResponse>
    </soap:Body>
</soap:Envelope>
```

The SOAP envelope provides information around the client-server interaction, such as the operation being performed or how the message is being processed. The envelope consists of header and body—the header is an optional attribute and it can constitute data such as a security token for authentication. The body consists of the request payload, as in the data required for processing specific web service operation. The encoding style included in payload is an optional parameter and can be added per our implementation requirements—it defines how the content in the SOAP envelope is encoded.

Consider an example of an online food ordering application. There can be multiple clients of this application—for example, there may be one client responsible for placing the order, another for payment processing and so on. The food ordering application can expose different APIs to support all these use cases. We need a mechanism to publish different operations to all the clients with expected input and output so it's easy to figure out the integration details. We solve this by using Web Service Description Language (WSDL). WSDL is an XML based language helpful in defining the web service operations along with the expected input and output.

SOAP is relatively complex with its XML based structure and has its own learning curve. You won't find many new use cases being developed via this protocol. Developers tend to lean towards simpler paradigms such as Representational State Transfer (REST) over SOAP for application development and interaction over the internet.

## **Representational State Transfer**

REST is widely used alongside HTTP protocol for client-server interactions and uses HTTP methods to implement different CRUD (Create, Read, Update and Delete) operations. REST is a software architectural style and not a protocol like SOAP and is used extensively for building applications in today's era of the internet. Let's discuss the guiding principles of the REST design pattern. Note

that the APIs following these standards are referred to as RESTful APIs.

- The communication over REST APIs include client, server and resources. The resource is a fundamental entity which can be uniquely identified by an URL. The clients can interact with the server to perform any operation on these resources via HTTP methods.
- REST operates over a stateless paradigm, meaning the server is not responsible for maintaining the state of older requests. The state maintenance (if required) should be handled by the client and all relevant information should be part of the HTTP request in order for the server to process it.
- It is possible that responses to particular REST APIs don't change too frequently. In these kinds of scenarios, the responses can be cached at client's end, at the browser level, utilizing a Content Delivery Network (CDN) or any other caching mechanism.
- There is a uniform interface between the components involved in information transfer following standardized ways for communication. This ensures that it's easy to evolve the capability of the software application in the future. The key factors include unique resource identification, resource update via representations (client and server communicate to update a resource in formats such as plain text, JSON, XML or HTML), self-descriptive messages (the server should share relevant information in response in order for it to be processed by the client) and HyperMedia as the engine of application state (clients should be able to navigate through the resources via hyperlinks present in server response).
- There can be multiple systems involved in gathering information. The client is abstracted out from these multiple

layer calls inside the system and it is only responsible for sending requests to the first layer (such as Load Balancer, API Gateway or any other service endpoint) and gathering a response from it.

- The server can deliver code on demand in the form of scripts in response to client requests and clients can execute it as per requirement.

### NOTE

We'll dive deep into Load Balancer, API Gateway, Network Address Translation (NAT), IP addresses and related networking concepts in Chapter 9 – AWS Network Services.

As REST can operate with JSON, it's much simpler for developers to work with. SOAP is known for its support for advanced security features, typed data and procedures (SOAP allows remote procedure calls similar to local procedure) but it comes with higher complexity. We can definitely introduce advanced security aspects in REST as well, though it requires extra design considerations which SOAP is inherently built to support. As always, it varies from use case to use case and there is no perfect choice. The REST architectural pattern operates on a fixed schema defined by the server but in some cases we might look for flexibility in a way that the client is able to define data needs. Let's explore this idea in more detail in the next section where we talk about [GraphQL](#).

## GraphQL

GraphQL is an API query language which allows clients to write a query specific to data retrieved on a single API endpoint (unlike REST where we define endpoints per API). Let's look at an example to compare GraphQL with REST. Let's say we want to retrieve a food item name and description to show food items available for a

particular restaurant on an online food ordering application. With REST, we'll require an API which can return these exact details or utilize an API that returns complete food item information and shows only the required information on the application (in similar fashion, there can be cases where we need to combine data from multiple APIs). There can be scenarios of over-fetching or under-fetching of data with REST, which can be resolved by the query flexibility offered by GraphQL—in addition to a few other features which we'll discuss shortly. Querying only the required data also helps in consuming only the needed network bandwidth which can make applications faster on slower networks.

Now that you have a fair idea about GraphQL's advantages over REST, let's dig a little deeper into how it works. GraphQL has a strong type system to define the entities supported via client's query and are written in GraphQL Schema Definition Language (SDL). Let's extend our previous example to show how it can be supported via GraphQL.

```
type Restaurant {  
    id: String!  
    name: String!  
    rating: Float  
    foodItems: [FoodItem!]!  
}  
type FoodItem {  
    id: String!  
    name: String!  
    rating: Float  
    isVeg: Boolean!  
}  
query {  
    Restaurant (id: "lsoa34444lsoa") {  
        name  
        foodItems {  
            name  
            description  
        }  
    }  
}
```

We defined two objects, Restaurant and FoodItem (exclamation mark (!) denotes a mandatory attribute) in SDL which can support our example. To gather data on food items, the query specifies only the required attributes by the client.

### NOTE

Please visit the GraphQL [documentation](#) for full insights on GraphQL.

Moving forward, querying (data read) is not only the capability we need with the APIs—we also need support for operations such as creating, updating, and deleting data (data modification on server datastores). GraphQL supports these operations with a feature called Mutations. The mutations start with a mutation keyword with syntax similar to GraphQL queries. Let's look at an example mutation of creating a new food item for a restaurant.

```
mutation {
  createFoodItem(name: "Paneer Tikka Masala", isVeg: true) {
    id
    name
  }
}
```

Mutations can also send back a response similar to a GraphQL query, along with performing its main job of data modification, like in the above example, the mutation will return id and name of newly created food item as part of response. We might also be interested in knowing about any new changes happening on the server side. For this, we can leverage technologies such as SSE or WebSockets, which we discussed earlier in the chapter. We can also use GraphQL subscriptions (defined by Subscription keyword) which inherently establishes a bi-directional connection to inform about any new events in near real-time. For example, if a new food item is created.

The GraphQL query, mutation and subscriptions are defined in GraphQL schema and offer clients insights on the API capabilities.

One of the benefits of GraphQL over REST is its integration with a single endpoint and the fact that it serves all the API needs with the same API endpoint, instead of multiple. As our system evolves, it's possible that frontend applications will need to interact with multiple backend applications to gather the necessary data and each backend application exposes their own GraphQL endpoint. This can be optimized by choosing a GraphQL client such as [Apollo](#) or [AWS AppSync](#) (which we'll discuss in Chapter 12) to help applications scale with much more ease.

## Web Real-Time Communication

People around the world interact with each other over text messages, voice, or video calls. Text messages are simple to implement and operate, compared to voice and video calls. Think of a scenario where you're interacting with your friend over a video call, what do you expect experience-wise from the video calling application? The primary expectations are that you're able to see your friend clearly and able to hear their voice without any lag. The Web Real-Time Communication (webRTC) offers a mechanism to support audio and video communications in web browsers or mobile applications over the internet by APIs without a requirement to install any specific plugin. Additionally, it offers support for file sharing and data exchange between the web services in near real time. Up to this point, all the protocols or the architectural patterns we've discussed have included a server and we focused our discussion around client-server communication. WebRTC allows direct peer-to-peer connection without any requirement of a server in between.

Note that though peers can communicate directly, a server can be introduced as well if the need arises. A key consideration to note is that the latency can increase substantially if the number of peers

communicating with each other increases to double digits—for example, a voice channel on **discord servers** with 1000 members.

Note that WebRTC is not itself a protocol—it's a tool working on top of multiple protocols to serve our use-case. Some of the APIs offered by WebRTC include RTCPeerConnection for establishing connections between the peers, getUserMedia to access audio and video from a user's device, and RTCDATAChannel for bi-directional data exchange between the peers. Let's dig a little deeper into how the peer-to-peer connection is established.

- For communication over the internet and connecting with another device, we require the IP address and the port number. It becomes difficult if the devices are behind a Network Address Translation (NAT—more details on this in Chapter 9). In these kinds of scenarios, webRTC can utilize Interactive Connectivity Establishment (ICE), which helps with figuring out the optimal communication route. We don't require ICE if it is not a direct peer-to-peer connection and our system operates on client-server architecture.
- To communicate with any device, we need a unique address. This might not be possible if you're operating inside a private network behind a NAT (all devices in this network will have the same public IP address and are distinguished by assigned private IP address). **STUN** (Session Traversal Utilities for NAT) protocol is used if the devices are behind a NAT. Both the peers connect with the STUN server to figure out the IP address, port (allocated by NAT corresponding to private IP address and port) and type of NAT the device is behind. Once this information is obtained, it will be used as part of the Session Description Protocol (**SDP**) as shown in [Figure 5-3](#).
- The two peer devices exchange ICE candidates (network address and port) with each other via a signaling mechanism such as SDP. SDP includes information such as media type

(audio/video), transport protocol (UDP, IP, etc), media format, remote address and port for the media.

- There may be some scenarios where direct peer-to-peer connection is not feasible due to restrictive firewalls and the STUN server fails to do its job. In these scenarios, we can use something called **TURN** (Traversal Using Relays around NAT) servers. A TURN server acts as an intermediary between the peers to communicate, where essentially one device sends a message to the TURN server and then the TURN server relays this to another device. Please note that a TURN server can be used single handedly to serve the functionality of STUN and relay traffic, but the STUN server is always used as the first point of action. This is because using the TURN server to stream high quality data is relatively expensive, adding an extra hop in the architecture shown in [Figure 5-4](#).

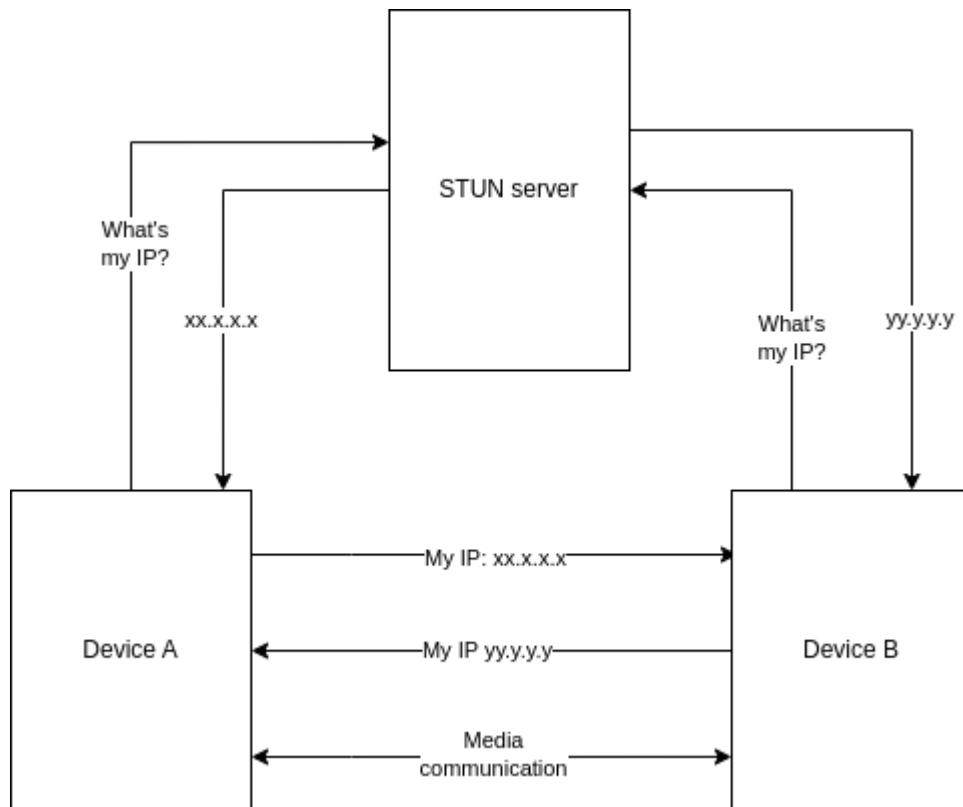


Figure 5-4. WebRTC communication with STUN server

Once the peer-to-peer connection is established, media data can be exchanged in both directions to carry forward the communication. WebRTC is used across the IT industry to support voice and video communications. While we tried to cover the basic and most widely used communication tools, there are multiple other protocols or technologies built and designed to solve specific use cases that we simply didn't have the room to cover here.

## Conclusion

The OSI model is a good starting point to understand data communication over the internet. The device communications we see around us mostly follow the TCP/IP model with TCP, UDP and IP being the base protocols for data transport. There are multiple protocols built around them such as HTTP, SMTP, XMPP, etc. to serve specific use cases. It can be confusing to try to determine which protocol makes more sense to implement for your specific use case, and it's important that you identify the most critical requirements in order to make your decision. You may also find that combining multiple technologies to meet customer demands is your best option. For example, XMPP is extensively used for text messages and WebRTC is a preferred choice for audio/video communications. Direct peer-to-peer communication can become complex, particularly when you involve multi-user experiences such as group video/audio calls. Combining these tools can reduce this complexity, bringing the benefits of the text, audio and video functionality of WebRTC and XMPP together.

We also discussed communication types and which mechanism might make more sense for our own implementation. We suggest having you use case requirements in place before trying to compare your options—there may never be a clear winner when designing large scale distributed systems. We ended our discussion with

different communication standards and architectures for how client-server information sharing works across the internet.

In the next chapter we'll dig into the concept of containers. Docker and Kubernetes are hot keywords in the industry—it's likely you've already heard of these concepts, which are used to deploy and run your software applications.

# **Part II. Diving Deep into AWS Services**

---

At the end of this unit, the readers will understand:

- The Network services offered by AWS and how to utilize them to build a resilient cloud infrastructure.
- The Storage services offered by AWS and how to pick the right storage service for different use-cases.
- The Compute services offered by AWS and how to identify the right size and type of computing resource, while keeping the variable cost in check using containers and other serverless offerings for different types of workloads.
- The Orchestration services offered by AWS and how to properly decouple and scale the application on AWS using these offerings.
- The Big Data & Analytics Services offered by AWS and how to use the services to run large scale jobs and answer queries on large datasets to derive value out of their Big Data.
- The ML services offered by AWS and how to solve ML business use-cases and set up ML pipelines and operations for their business using the managed offerings.

The next five chapters as part of this unit, will cover

1. Chapter 9 will dive into the fully-managed Network services that AWS offers. We'll learn how to use them to create a strong and

resilient cloud infrastructure. We'll cover things like VPC (Virtual Private Cloud), Subnets, NAT Gateway (a way for your cloud systems to communicate securely with the outside world), Route53 (for managing domain names), Cloudfront (to make your web services load faster), and different types of load balancers.

2. Chapter 10 will explore the managed Storage services provided by AWS and help you understand how to choose the right storage service for different needs. The chapter will start with Elastic Block store, Elastic File Store and S3 object store and its different classes. The chapter will then cover databases including RDS, Aurora, DynamoDB, DocumentDB, InmemoryDB for Redis, AWS Keyspaces, AWS Timestream and AWS Neptune, ElastiCache including ElastiCache for Redis and ElastiCache for Memcached.
3. Chapter 11 will introduce you to the Compute services AWS offers. We'll figure out how to choose the right kind and size of computer resources for different jobs. We'll look at things like EC2 (Elastic Compute Cloud) – it's like renting virtual computers in the cloud. Then we'll talk about Lambda (a special kind of on-demand computing that happens in response to events), ECR (Elastic Container Registry), ECS Fargate (which helps manage and deploy containers), and ECS EKS cluster (for running applications in containers).
4. Chapter 12 will cover the managed Orchestration services from AWS. These services help you manage and coordinate different tasks in your software system. We'll talk about SQS (Simple Queue Service) – a way for different parts of your system to communicate indirectly, SNS (Simple Notification Service) for sending messages, and MQ for ActiveMQ and RabbitMQ – they help different parts of your system talk to each other. We'll also explore Step Functions, which help you piece together and manage workflows and AWS Managed Workflows to help you

manage and monitor your workflows. Lastly, we will cover Cloudwatch to monitor and debug your cloud applications, AWS IAM for authentication and authorization over AWS resources, Cognito for managing identity pools and AppSync to orchestrate all AWS services under one GraphQL endpoint.

5. Chapter 13 will explore AWS's Big Data, Analytics & ML Services. These services help you work with huge amounts of data and find valuable insights. We'll check out Kinesis – it's like a stream of data flowing in; Redshift – a powerful tool for analyzing data; EMR – for processing big data; Athena – for asking questions about your data; Data Pipeline – for moving data around; Glue – for preparing and transforming data; and Quicksight – for creating visual reports. This chapter will also introduce you to the ML services offered by AWS in a nutshell and how to solve ML business use-cases and set up ML data pipelines and operations for your business using the managed offerings.

By the end of this unit, you'll have a clear understanding of all these AWS services and how to use them effectively. You'll know how to build a strong and secure cloud infrastructure that can handle different tasks and workloads.

Later, Part III is all about real-world examples of how to design systems using AWS Cloud. We'll break down different use cases step by step in an easy-to-understand way using the concepts from Part I and AWS services from Part II.

# Chapter 6. AWS Network Services

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

People interact with multiple applications (or with each other) via the internet. We touched upon concepts around 'Communication Networks and Protocols' in Chapter 6—this chapter is an extension of Chapter 6 that will introduce you to AWS networking services such as Amazon VPC, Amazon Route 53, Amazon Elastic Load Balancer (ELB), Amazon API Gateway. This is intended to help you set up networking infrastructure on AWS cloud and explore the networking concepts related to different services and how to establish connectivity between them.

To make systems more reliable, safe, and to ensure they operate as per their own business requirements, companies used to prefer setting up their own infrastructure in on-premise data centers, but that comes with a high infrastructure management cost. If you are just launching your start-up idea, it will likely be too costly to set up a personal data center. Cloud provides cost benefits (analyzing the cost of resources and optimizing them to full potential) along with a lot of flexibility. For example, It is much harder to sell back a physical server that you bought for setting up your personal data center than it is to turn off an Amazon EC2 machine (server) in AWS cloud. The decision of whether to go with a Cloud provider based solution (and if yes, then which Cloud provider?) or with an on-premise data center will heavily depend on the business requirements.

AWS operates on a **shared responsibility model**, meaning the customer and AWS both work together to make best use of services, in a secure and cost effective way where AWS is responsible for "security of the cloud" and customers are responsible for "security in the cloud". Before we deep dive into multiple AWS Networking Components, let's dig a little into where these AWS services are located and how they are segregated per customer.

## Getting Started With AWS

To start with AWS Cloud, customers need to [create an AWS account](#). An AWS Account is a fundamental entity in Amazon Web Services that provides access to a wide range of cloud services. This account will hold all the information related to your AWS resource (such as compute instances, storage, networking etc) creation, management, operations, support,

billing etc. With an AWS account, you can provision and configure resources, monitor usage and costs, set security and access controls, and interact with various AWS services. It will be very important to identify how many AWS accounts should be set up. This will vary from use case to use case and scale the company operates on—below are some general suggestions to consider when deciding what number of AWS accounts one should configure:

- Complete separation of applications, meaning every application is launched in a new AWS account. This option presents benefits of full separation and effective cost management of applications, but presents challenges in the way of too much operational cost.
- Separation of AWS accounts based on business type—for example, a cab booking company operates in domains like cab availability, payments, analytics etc. All applications related to a single business will operate in a single account, which also provides the advantages of correlated applications located near to each other optimizing latency and offloading requirements of resource setup for service connection between different accounts. These applications can be segregated in Amazon VPCs or AZs depending on the use-case.
- Separation of AWS accounts based on software domains—for example, having separate accounts for networking, monitoring, storage, and security and auditing. This kind of setup provides the benefits of easier operations management. For example, all the networking configurations are present in a single account that can be managed by the Network engineering team.

AWS recommends creating a multi account set up to ensure there is clear division of responsibility and keeping future scaling of systems in mind. For example, separate networking accounts will help to keep all networking configurations at a single place and with access only to the networking team. AWS offers a service called [Landing Zone](#), which provides a baseline to get started with multi-account architecture, identity and access management, governance, data security, network design, and logging. AWS Landing Zone can be orchestrated by customers themselves or as a managed service. One such service, [AWS Control Tower](#), can be used to set up initial prescriptive configuration and further customizations can be made as organizations scale.

## AWS Regions

Once the AWS account is created, you're ready to launch the first server in the cloud. A very intriguing question though, where is this cloud? AWS Cloud operates from a physical location which might be near or far from you. AWS defines these geographically distributed locations around the world where AWS operates data centers as [AWS regions](#) - for example the US North Virginia region is called us-east-1. Each region is a separate and independent geographic area, isolated from other regions. You can choose the AWS region to launch resources per your business requirements, such as latency constraints or compliance regulations, or for AWS service availability in the region. For example Netflix architecture has presence in three AWS regions, ensuring high availability even in cases of a regional failure. Keeping resources at multiple locations helps to build resilient systems, which we touched upon in Part-I of this book. Further, there are some global services such as [Amazon S3](#) or

AWS Identity and Access Management (AWS IAM) service where region selection is not a requirement. You can check the complete list of regions available across the globe [here](#).

#### NOTE

There is a possibility that not all AWS services are present in your AWS region.

## AWS Availability Zones

Customers choose to deploy resources in multiple regions to improve availability, latency, and many other business use-cases. A key point to note is not everyone can afford to replicate resources at multiple AWS regions. Does that mean they compromise application availability then? Cloud providers work on a shared responsibility model which means that in order to ensure a good run on cloud, customers and cloud providers have to work together. A simple example could be deploying your servers in **multiple availability zones** instead of just one. That way, even if one AZ goes down, there are servers operational in other AZs to serve the traffic. You can think of a single region as a cluster of data centers and each individual (or combination of) data center is an availability zone. An **availability zone** is physically isolated from other availability zones by meaningful distance with independent power, cooling and fast private fiber-optic low-latency network connectivity. The purpose of AZs is to provide fault tolerance, resilience, and high availability by allowing you to distribute your applications and data across multiple AZs within a region. AZ are classified with suffix to region names—for example, us-east-1a or us-east-1b are AZs within the us-east-1 region.

## AWS Local Zones

Consider the following scenario: you are all set to launch your startup for the people of New Delhi, India. The application setup requires single digit p99.99 latency for API operations, but the nearest AWS region is Mumbai, India. The operations are optimized well and your use-case doesn't allow you to use pre-cached data such as utilizing Content Delivery Network or retrieving data from servers located at specific AZ. How can you overcome such an issue? This can potentially be solved by placement of database instances near to customers and this can be achieved via AWS Local Zones. Local Zones help to set up infrastructure near to customers, which is connected to AWS regions via fast paced network.

AWS Local Zones are an extension of an AWS Region and are designed to bring AWS services closer to specific geographic areas with low latency requirements. Local Zones are geographically separate from their parent region and are located in metropolitan areas. They provide a subset of AWS services and are primarily intended for latency-sensitive workloads that require proximity to end-users or specific on-premises resources.

You can check all available locations for Local Zones [here](#) and list of AWS services supported [here](#).

## AWS Edge Locations

Consider that HBO is planning to stream a new season of Game of Thrones and the first episode will be out on upcoming Sunday morning 8 AM. GOT is popular worldwide and it is

expected to be watched in multiple countries at the same time. How can HBO ensure the best customer experience via full HD video quality with no video buffering? The AWS Content Delivery Network Service called **Amazon CloudFront** helps to place the content near to users at locations referred as Edge Locations so they can be served in minimum time possible.

AWS Edge Locations are points of presence (PoPs) distributed globally to bring AWS services closer to end-users. Edge Locations help improve the performance of content delivery by acting as caching and content delivery endpoints for Amazon Cloudfront, thus reducing latency and improving data transfer speeds.

You can think of AWS Edge locations as data centers which are connected with AWS regions to support fast upload and download of data. Some other services like Amazon Route 53, an Amazon DNS service use the same setup for faster resolution of DNS queries. **Figure 6-1** shows connectivity of users via AWS edge locations to AWS regions. We'll explore more about Amazon Route 53 and Amazon CloudFront towards the end of this chapter.

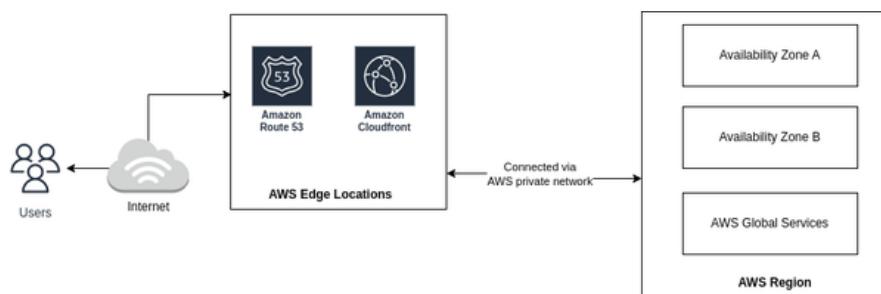


Figure 6-1. Connectivity to AWS region via Edge Location

We gathered a very high level of view of how AWS Cloud is set up and made accessible to customers across the globe. Let's dig right into how connectivity can be established with AWS Cloud and look at different networking components within it.

## Introduction to AWS Networking Services

Figure 6-2 from the last section described how users are establishing connections to AWS services located within the AWS region. It's important to understand how this accessibility is maintained and how AWS Cloud provides a similar abstraction to one's private data center. This section will introduce you to multiple offerings provided by AWS in the context of networking and connectivity.

### Amazon VPC

Setting up your own data center could be costly both in terms of monetary costs and operations management. Amazon VPC, short for Virtual Private Cloud, provides a similar level of infrastructure separation as you'd have in a personal private data center. We can think of Amazon VPC as your personal data center located inside AWS Cloud.

Amazon Virtual Private Cloud (VPC) is a service provided by AWS that allows you to create a virtual network in the cloud. It provides you with control over your network environment, including IP address range selection, creation of subnets, configuration of route tables, and network gateways. Amazon VPC enables you to securely launch resources like Amazon EC2

instances, Amazon RDS, and Amazon Elastic Load Balancers within a logically isolated section of the AWS cloud.

### NOTE

This book will not outline the infrastructure setup steps but will focus more on a deep dive into AWS core concepts and will provide links to AWS documentation for the most updated guidance on setup.

AWS Cloud is utilized by a lot of customers and Amazon VPC plays a vital role in setting up boundaries between the customer resources. AWS accounts come with one default VPC to launch resources and further more VPCs can be created as per business use-case requirements. There is a default limit of 5 VPCs per account which can be increased by raising a query with AWS Support Center. Let's dig into Amazon VPC in more details, starting with some basic networking knowledge helpful in making better decisions on Amazon VPC setup.

## IP Addresses

Let's consider a real-world example. You're very excited to meet your new friend in person and both of you agreed to meet at Cafe Delhi Heights, 3rd Floor, 301 & 302, Ambience Mall, Gurugram, India. However, you need a specific location, more specifically a table number in this restaurant and not just the restaurant location. This pin-pointed location in the networking world is called an IP address—a unique string of characters that identifies each computer. Every device should have an IP address to connect with another device on the internet. IP addresses can be of type IPv4 or IPv6. Let's discuss each of these in a bit more detail.

### IPv4

IPv4 is a 32 bit or 4 byte address space where each byte is represented via decimal numbers (binary octet) and separated by dot (.), for example 192.168.1.0. How can we identify to which destination the traffic should be routed to? To establish a similar analogy, how do you think the Postal Service delivers mail at your doorstep or you know to meet your new friend at Table Number 21 at Cafe Delhi Heights? There are two components required in general for unique identification. For physical mail, it would be your zip code/postalcode/area PIN code (depending on your country) and your house number, and for meeting your new friend, it would be the restaurant location in Ambience Mall, Gurugram and then the table number in this restaurant.

In similar fashion, to deliver network packets to your personal computer, there are two components involved—a network component (network ID) and a host component (host ID). For example, your office network will map to a network ID and your personal computer will map to a host ID.

The division of the number of bits that should be allocated to network Id and host Id is defined via IP classes as shown in [Figure 6-2](#). There are in total 5 IP classes from A-E, out of which Class D is reserved for multi-tasking and Class E is reserved for research purposes.

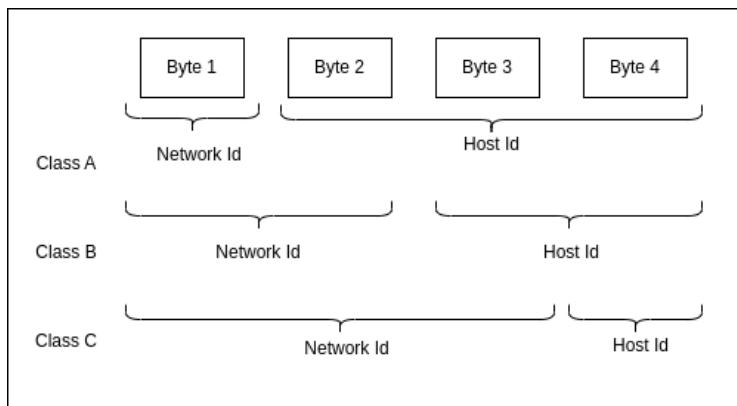


Figure 6-2. IP address classes

One difference between Class A, B, and C is the number of bits assigned to each network Id and host Id. Another is the range of IP addresses that are allowed in each class, as represented in Table 9-1.

T  
a  
b  
l  
e  
6  
-  
1  
.C  
/  
a  
s  
s  
-  
w  
is  
e  
I  
P  
v  
4  
A  
d  
d  
r  
e  
s  
s  
r  
a  
n  
g  
e

---

IP Address Class	From Range	To Range
------------------	------------	----------

---

Class A	1.0.0.0	127.255.255.255
---------	---------	-----------------

---

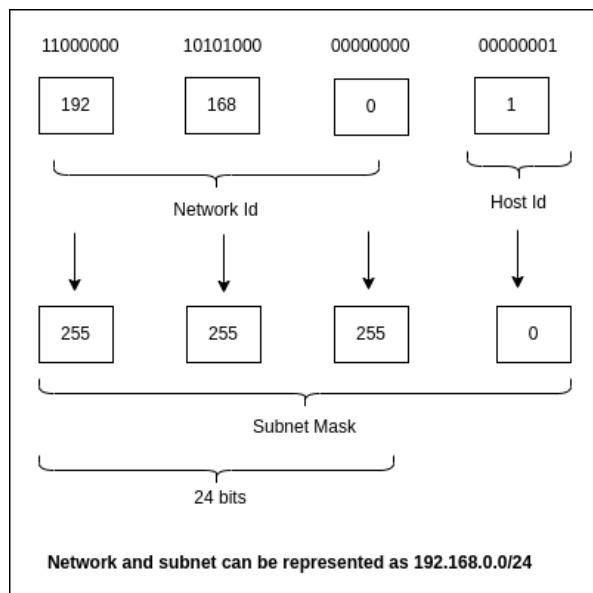
Class B	128.0.0.0	191.255.255.255
---------	-----------	-----------------

---

Class C	192.0.0.0	223.255.255.255
---------	-----------	-----------------

Deciding which class suits your specific business use-case depends on the requirement of the number of networks and the number of hosts in a network. For example, Class A provides 126 network IDs and 16,777,214 host IDs, whereas Class C provides 2,097,152 network IDs and 254 host IDs. The division of IP address or network space into network and host address is achieved via Subnet Masks.

As the name suggests, subnets help in dividing the parent network into sub-networks. Subnet mask is the division of IP address into network and host address. It is a 32 bit number where host bits are set to 0 and network bits are set to 1. Amazon VPC logically isolates the resources at regional level and further division into AZ can be achieved via subnets—we'll be exploring this while creating our first Amazon VPC in follow up sections. [Figure 6-3](#) shows an example IPv4 address along with subnet mask representation.



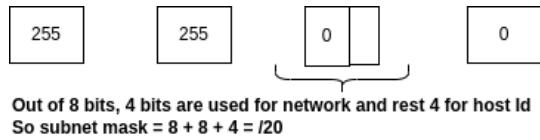
*Figure 6-3. Representation of IPv4 address and subnet mask*

**NOTE**

Keep in mind that 0 and 255 are reserved for special purposes and therefore can't be assigned to hosts. Hence the number of host IDs in Class A is 126 and not 128.

A specific problem that arises with IPv4 addresses is the very limited number of addresses available in comparison to the number of devices and networks across the world. The intermittent solution to slow down exhaustion of IPv4 addresses is via Classless Inter-Domain Routing (or CIDR).

CIDR is a way of representing an IP address and its subnet mask. The classless concept was introduced in 1993 to slow down the exhaustion of IPv4 addresses. CIDR helps in varying subnet mask length, skipping standard division via classes. For example, you can create either /16 or /24 subnet masks but not /20. This kind of division helps in optimizing the class space of IP addresses. You can see this represented in [Figure 6-4](#).

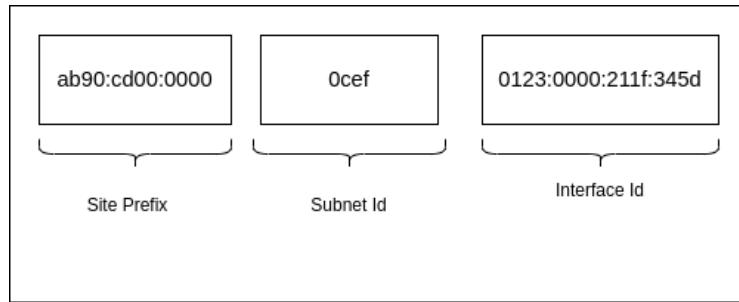


*Figure 6-4. Variable length Subnet mask*

However, the long term solution is migration to IPv6 addresses.

## IPv6

IPv6 is 128 bits which helps in much larger network space as compared to 32 bits IPv4 address. IPv6 is represented by 8 groups of 4 hexadecimal digits and separated by colons. For example, ab90:cd00:0000:0cef:0123:0000:211f:345d. [Figure 6-5](#) describes simple IPv6 notation where site prefix represents public topology allocated by an Internet Service Provider(ISP) or Regional Internet Registry, subnet Id represents private topology which is internal to specific network, and interface Id represents a unique device identifier which is configured via interface's MAC address by using IEEE's Extended Unique Identifier (EUI-64) format.



*Figure 6-5. Simple IPv6 address representation*

Both IPv4 and IPv6 addresses can be further divided into public and private IP addresses based on their visibility. The concept of public-private is more used for IPv4 addresses considering the limited availability of addresses. We'll be using examples of IPv4 addresses to walk through it.

## Private and Public IP Addresses

Let's refer back to our earlier real-life example. You agreed to meet your friend at a reserved table number 21 at Cafe Delhi Heights and enjoyed the positive vibe and food of the restaurant. You got curious and decided to visit the kitchen where this delicious food was prepared, but the restaurant owner denied access to the kitchen and mentioned only specific individuals can go to the kitchen. Here the kitchen is private space and the general sitting area is public space. Certainly there are ways to get to the private space (aka the kitchen in this scenario) and we'll explore this in follow up sections.

Consider the example of your work from home setup—you've a router for Wi-Fi connection and all the devices (such as mobile phone, office laptop, personal laptop, etc.) connect with this router to access the internet. If you run a Google search on all the devices for, "What's my IP?", you'll get the same IP address. This IP address is called a public IP address and is assigned by the ISP. Restarting the router might lead to a different IP address on all the devices, but it will be the same across the devices.

All these devices can connect with each other without going over the internet via Private IP address. A Private IP address is assigned by the router to each device connected to it. Table 9-2 shows the range of addresses allowed for Private IP—the rest all can be used for public network space.

T  
a  
b  
l  
e  
6  
-  
2

.  
P  
r  
i  
v  
a  
t  
e  
I  
P  
v  
4  
a  
d  
d  
r  
e  
s  
s  
s  
p  
a  
c  
e  
p  
e  
r  
c  
l  
a  
s  
s

Class

IP Range

---

Class A      10.0.0.0 – 10.255.255.255

Class B      172.16.0.0 – 172.31.255.255

Class C      192.168.0.0 – 192.168.255.255

The IP addresses are assigned dynamically from the provided IP address range on AWS resource creation. In case there is a requirement to associate specific IP addresses to a resource, we can utilize Elastic IP addresses.

### Elastic IP Address

Elastic IP address is a static and public IPv4 address associated with an AWS account which can be assigned to an Amazon EC2 instance or network interface without changing on any state change. This can be considered to be used for scenarios with requirements of static IP address without being changed over time. For example, avoid Amazon EC2 instance IP address change if a new instance is spawned up to replace an unhealthy instance.

Please note that there is an additional cost associated with Elastic IP addresses. It is chargeable and bought for a specific region, IP addresses of one region are not accessible in other regions. Now, let's move forward on key considerations for the creation of Amazon VPC.

## Considerations for Amazon VPC Creation

The networking ideas we gathered in previous sections will help us to set up our first VPC. The VPC wizard asks for IPv4 CIDR block, IPv6 CIDR block, and tenancy. The IPv4 CIDR block is a required input that must be added to proceed further. The VPC can work in dual mode, operating with both IPv4 and IPv6, and the allowed CIDR block for IPv4 is between /16 to /28 and for IPv6, it is fixed to /64.

You'll need to consider the following points about CIDR block:

- VPC creation requires an initial IPv4 CIDR block, although you might have a use-case to just use IPv6 address space. AWS VPC supports both IPv4 and IPv6 address space in a single VPC and it can be specified during creation.
- The CIDR block should be specified from the RFC 1918 range, and it is recommended to use a private IP address range, though public can also be chosen.
- The size of the CIDR block can't be changed once created. Customers should carefully plan the size of CIDR considering future needs to avoid hurdles or rework in the future. In short, bigger CIDR blocks can be chosen if you're not able to gather concrete details to have more flexibility in the future.

- The CIDR block should not overlap with existing CIDR blocks associated with VPCs. This is also essential for utilizing services such as [VPC Peering](#) and [AWS Direct Connect](#), the CIDR blocks should not overlap across the VPCs.
- You can associate up to 5 additional CIDR blocks( this is a soft limit and can be adjustable up to 50) to a VPC, the additional CIDR block's range should be strictly smaller than the primary CIDR block.

#### **NOTE**

Soft Limit means the limit is adjustable and can be increased by following AWS Service Quotas or with help of AWS Support team.

Next we need to consider tenancy. Let's go back to our real-world example. There are two ways of booking Cafe Delhi Heights for meeting your new friend. The first option is to book the entire restaurant, as this is a top secret meeting and you don't want any other people to be available in your booked time. This is called dedicated tenancy.

The second option is to reserve a table for two—other available tables can be booked by other people which is called shared tenancy. It's logical to go with option two considering the cost unless there are specific reasons such as a secret meeting. The same concept is applicable when we request servers from AWS. The instances can either run on shared hardware (other people's instances can run on the same hardware via virtualization) or on dedicated hardware where your instances are separated from other customer's resources, depending on your business needs. This option can be selected at the time of Amazon VPC creation and all the instances launched in VPC are created with the same option by default, unless overridden.

#### **NOTE**

AWS provides multiple tools for creation of resources such as AWS CLI, AWS CDK or via AWS Console. As a beginner, you can start experimenting via AWS Console but as systems scale, we recommend maintaining a code repository to provision any AWS resources, popularly known as Infrastructure as Code(IaC). This could help on multiple fronts like replicating the same resources in another region or maintaining infrastructure audit.

The next step in VPC configuration is setting up subnets. Subnets help in separation of resources across multiple availability zones. Let's dig into those.

## **Subnets**

As mentioned earlier in the chapter, subnet means a sub network, logical subdivision of IP Address range inside the VPC CIDR block. Why should one divide network space into multiple sub-networks? The general idea of separation of resources per availability zone is to ensure high application availability. Each subnet is associated with an availability zone (AZ) within an AWS region.and by associating resources to the specific subnets, we ensure resources are launched into that specific AZ. One interesting fact around subnet mapping to AZ is it could vary from customer to customer—the us-east-1a for you can be us-east-1b for your friend.

AWS abstracts out these details from customers to ensure uniform allocation (or as per availability) of resources.

Each subnet should be assigned a CIDR block similar to VPC—the CIDR block to be assigned can be the same as the VPC CIDR block or subset of it. The key point is that the CIDR block across the subnets inside your VPC should not overlap. The first four and the last IP address in each subnet CIDR block are reserved and can't be assigned to resources. The subnet can be a public subnet or private subnet depending on its connectivity to the internet. Let's dig a little deeper into that.

## Public Subnet vs Private Subnet

Public and private subnets look similar, the difference lies in their ability to connect to the internet. For example, a web application's servers can be placed in a public subnet so that it is accessible to the general public, and the database servers can be placed in a private subnet to limit accessibility from the general public, see [Figure 6-6](#) for reference. You were denied access to take a look at Cafe Delhi Heights kitchen as it's located in a private space (or private subnet) though you can freely roam in a general sitting area, as this is public space or public subnet.

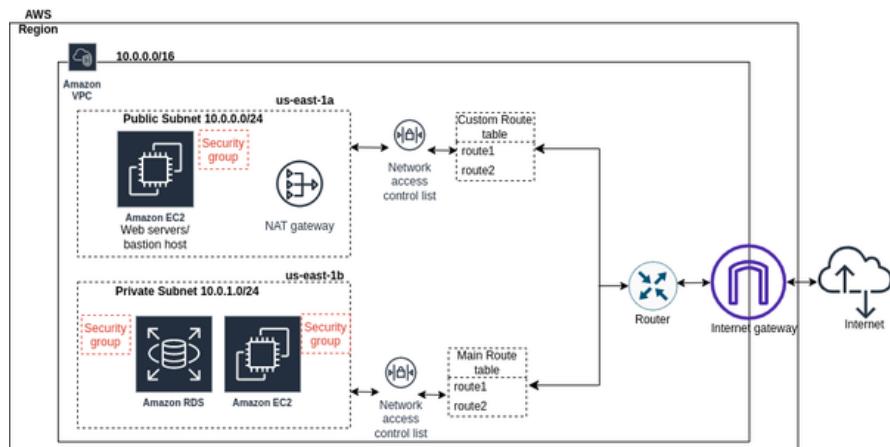


Figure 6-6. Overview of different networking components

The resources in the public subnet can access the internet due to the presence of a direct route to an [Internet Gateway](#), whereas resources in the private subnet require a Network Address Translation(NAT) service or a NAT instance to access the internet—more details are captured in the [NAT gateway](#) and Internet gateway section.

The key point is that the direct route to the internet gateway is the only differentiating factor between public and private subnets. A subnet with resources having a public IPv4 address but no direct route to the internet gateway is referred to as a private subnet.

When setting up your VPCs and subnets inside your AWS account, consider these following best practices:

- Use multiple subnets: Create multiple subnets within different availability zones to achieve fault tolerance and high availability.
- Isolate resources: Use separate subnets for different types of resources to improve security and network segmentation.

- Public and private subnets: Place resources with public access in public subnets and sensitive resources in private subnets.

## Internet Connectivity

As we mentioned earlier, VPC provides a logically isolated network in AWS Cloud, but how does this logically isolated section connect with the internet? Or how can internet traffic access the resources located in the subnets? The routing of traffic and securing of resources is achieved via components such as route tables, internet gateway, security groups, etc which we'll be focusing on in the following sections.

### Route Tables

In the AWS Cloud, you just need to configure required routes in the route table and don't have to worry about setting up a router. A route table is a collection of rules, referred to as routes, that determine where network traffic from your subnet or gateway is directed. We can assume AWS internally maintains a router to facilitate this routing.

Every subnet should be associated with a route table, and the main route table is created implicitly to provide private access among the subnets. You should create more custom route tables as required and assign them to subnets—note that the same route table can be associated with multiple subnets. The key consideration here should be to create a route table for clear division of routing responsibility—this helps in better maintainability and readability of routes by not overcrowding a single route table.

Let's consider the two kinds of route tables:

#### *Main Route Table*

The Main Route Table, which automatically comes with a VPC, by default contains the first entry for local routing in the VPC, which helps resources in different subnets to establish connectivity with each other. Every subnet creation leads to implicit association to the main route table. If required, customers can create a custom route table and explicitly associate it to a subnet.

There are a couple of things you need to consider about the Main Route Table:

- It can't be deleted and the custom route table can't be set as the main route table, though it can be replaced with a custom route table for subnet association.
- You can add additional routes to the main table—our recommendation would be to avoid this and create custom route tables for adding custom routes.

#### *Custom Route Table*

Custom Route Tables don't contain any routes by default and should be updated as per network traffic routing requirements. Custom route tables can be deleted unlike the main route table though there should be no subnet associations for this operation to be successful.

Now that you understand the two types of route tables and their purpose, let's look at how these routes are added and what they look like. There are two important inputs in the route table - Destination and Target.

- Destination is added as CIDR, basically a range of IP addresses that specifies where the network packets should go, for example, 10.0.0.0/16.
- Target specifies how the network packets will reach the destination, such as gateway, network interface or a connection—for example internet gateway(igw) for routing traffic to internet or VPC peering connection. An example Route Table is shown in Table 9-3.

T  
a  
b  
l  
e  
6  
-  
3

.  
R  
o  
u  
t  
e  
T  
a  
b  
l  
e  
w

it  
h  
i  
n  
t  
e  
r  
n  
e  
t

c  
o  
n  
n  
e

c  
ti  
vi  
t

y  
vi

a  
I  
n  
t  
e

r  
n  
e  
t  
G  
a  
t  
e  
w  
a  
y

Destination	Target
VPC CIDR	Local
0.0.0.0/0	igw-id

Route Tables direct network traffic in and out of a subnet but it doesn't apply any security filters on this traffic. AWS provides software firewalls, **Security Groups**(SGs) and **Network Access Control Lists**(NACLs) to implement traffic filters which are useful in controlling the network traffic permissions. Both of these components help to control the traffic that can flow in and out of VPC.

## Security Groups

Let's understand how SGs can help to control the traffic we want to essentially filter out or disallow any unwanted traffic. SGs are created at the VPC level and assigned at an instance level, controlling inbound and outbound traffic at an instance level based on protocols, ports and IP addresses. Your EC2 instance can have one or more SGs. There will always be one SG associated to an instance, and, if not created, a default SG will be associated, which is created at time of VPC creation.

To secure and limit access for incoming and outgoing traffic to instances, you need to add inbound and outbound rules to your SG. Inbound rules define traffic that is allowed to the instance and outbound defines traffic that is allowed from the instance.

Consider the following facts about SGs:

- SGs are stateful. For example, you fire one request from your personal laptop to an EC2 instance to get some data and your IP address is added as part of Inbound rules. You'll

get a response back even if there are no outbound rules included for your personal laptop. In short, rules allowed in one direction will automatically be allowed in the opposite direction, there is no requirement for explicitly adding them.

- You can't delete the default SG. Our recommendation would be to create a custom SG as needed. Multiple SGs can be associated to an instance, but you should create SGs keeping future scale in mind. For example, you should avoid duplicate rules SG and reuse wherever it's possible.
- You should only add required access and not overexpose the resources. For example, for SSH connection, only allow Port 22 for a set of IP addresses and not the entire internet.
- SG doesn't provide an option to define explicit deny rules—all the rules which don't match the allowed rules condition are implicitly denied access to resources.

For each rule added as part of inbound or outbound rules, you'll need to specify the following parameters as shown in Table 9-4:

#### *Type*

It represents the type of traffic. On the basis of chosen 'type' value, AWS determines Protocol and Port range automatically. There is also a custom type to add custom values, for example for port range.

#### *Source or Destination*

Source attribute is added for inbound rules and Destination attribute is added for the outbound rules. Here, you can add specific IP addresses, complete CIDR blocks or other security groups as well.

#### *Description*

It is helpful to identify why a certain rule is added or what purpose it solves. This is an optional field, but we recommend adding a description for easy future references.

T  
a  
bl  
e  
6  
-  
4.  
E  
x  
a  
m  
pl  
e  
S  
G  
O  
u  
t  
b  
o  
u  
n  
d  
R  
ul  
e

Type	Protocol	Port Range	Destination	Description
SSH	TCP	22	117.212.92.68	Test SSH rule

## Network Access Control Lists

NACLs are stateless packet filters which are attached at the subnet level, unlike SGs, thus controlling inbound and outbound traffic at the subnet level. NACLs provide the ability to add, allow, or deny rules for outbound or inbound traffic at the subnet level. You may think of NACLs as an additional layer of security on top of SGs which ensure to block the traffic if SGs are too flexible.

If you think back to our example of the Cafe Delhi Heights kitchen, consider that there are individual chefs in the kitchen—you may be allowed to enter the kitchen, but still the chef may refuse to talk to you.

There are a few key things that you need to remember about NACLs:

- NACLs are stateless. Customers need to add explicit rules for both inbound and outbound traffic to allow or deny actions.
- VPC comes with a default NACL, which will be attached to all the subnets inside VPC and it allows all inbound/outbound traffic. For fine grain traffic control, you can create a custom NACL or modify rules in an existing one—our recommendation would be to create custom NACLs as needed.
- NACLs define inbound or outbound rules to allow or deny actions. The rules will be evaluated in sequence and once a particular rule succeeds, all the rules in sequence below will be skipped. If none of the rule succeeds, then final rule marked as '\*' will evaluate it as a deny action.

For each inbound or outbound rule, you'll need to specific the following parameters as shown in Table 9-5:

#### *Rule Number*

Rule numbers a sequence of numbers in which rules are evaluated which can be numbered from 1 to 32766.

#### *Type*

Type represents type of traffic. Based on chosen type, AWS pre-populates protocol and port range. Custom protocol and port range can also be added as per selected type.

#### *Source*

Source represents inbound rules added as a CIDR block.

#### *Destination*

Destination represents outbound rules added as a CIDR block.

#### *Allow or Deny*

For every rule, an explicit action should be added as 'Allow' or 'Deny'. This action determines if traffic is allowed or denied.

T  
a  
bl  
e  
6  
-  
5  
.E  
x  
a  
m  
pl  
e  
N  
A  
C  
L  
I  
n  
b  
o  
u  
n  
d  
R  
ul  
e

Rule Number	Type	Protocol	Port Range	Source	Allow/Deny
100	All traffic	All	All	0.0.0.0/0	Allow
*	All traffic	All	All	0.0.0.0/0	Deny

Route Tables, SGs and NACLs help to configure the routes and configure network security. To enable the connectivity as per the configured routes, AWS offers Internet Gateway or NAT Gateway which we'll be exploring in the next section.

## **Amazon VPC To Internet Connectivity**

We described VPC as a personal data center in AWS Cloud, but how will the users establish connectivity with this personal data center and how will this personal data center connect to the internet? In this section, we'll dig into different AWS components which helps to resolve these connectivity hurdles.

### **Internet Gateway**

In the section 'Route Tables' Table 9-3, we showed you a route as Destination '0.0.0.0/0' and Target as 'igw-id' for allowing internet access to resources. The destination for this route refers to the entire internet and the target is an internet gateway identifier.

Internet Gateway (IGW) is a horizontally scalable and highly available AWS managed VPC software component that provides connection between your VPC and the internet. IGW is managed by AWS and is a highly available, redundant and horizontally scalable application.

Internet Gateway is attached to VPC and can span across the subnets in different AZs.

In the context of our Cafe Delhi Heights restaurant, IGW represents the front door to get in or get out of the restaurant. Much like you can only go to the public sitting area in the restaurant and can't go inside the kitchens, the same applies to the IGW—it only helps to connect to resources in public subnets.

Here are a few key points about IGW:

- IGW helps in establishing connectivity in both the directions, from internet to VPC and vice versa by using public IP addresses.
- IGW provides NAT support for instances with public IPv4 addresses in the subnet. For traffic leaving for the internet from instance, IGW makes sure the reply to request is sent back to public IPv4 address, and for traffic destined for resource public IPv4 address, IGW ensures translation to instance's private IP address before the traffic is delivered to VPC.
- Internet Gateways are used in conjunction with route tables to determine the path of network traffic. A route table associated with the VPC directs traffic destined for the internet to the Internet Gateway. It acts as the default gateway for outbound traffic and routes it to the appropriate destination.
- Internet Gateways enable VPC resources to communicate with other AWS services, such as Amazon S3 or Amazon DynamoDB, over the internet.
- Internet Gateways are stateless, which means they don't maintain any information about the state of network connections. Each packet is evaluated independently based on routing rules and security settings.
- Internet Gateways are designed to be highly available and scalable. They are automatically replicated across multiple availability zones within a region, providing redundancy and fault tolerance.

IGW helps to establish internet connectivity from public subnets but how about private subnets? There can be scenarios of private subnet resources requiring internet access, one

such example could be downloading the latest software update. The private subnet to public internet connectivity is achieved via a NAT Gateway.

## NAT Gateway

The cooks working in the kitchen asked the head chef if she can collect feedback from customers about the food served—only the head chef is allowed to interact with customers directly in the main restaurant space and cooks only work in the kitchen, just to avoid overwhelming the customers or increasing the crowd in the main area.

The role of head chef is served by the NAT gateway for a subnet—it helps instances in a private subnet to connect with the internet via the Internet Gateway. You may think of NAT gateway as a bridge between internet gateway and private instances. Internet gateway requires public IP for interaction with the internet and NAT gateway facilitates that support. This support is available for TCP, UDP and ICMP protocols.

NAT Gateways (Network Address Translation Gateways) are AWS-managed network devices that allow resources within private subnets in a VPC to initiate outbound internet connections while preventing direct inbound access from the internet by hiding their private IP addresses.

To go a layer deeper, NAT means Network Address Translation, in simple terms it converts private IP address to NAT device public IP address and is mapped back to private IP address on return of response from internet. To establish a NAT Gateway, you need to allocate an Elastic IP address (EIP) and associate it with the NAT Gateway. The EIP serves as a static, public IP address that represents the NAT Gateway and is used for communication with the internet.

Here are the key things you need to consider about NAT Gateway:

- For high availability, NAT Gateway should be set up at the AZ level so that if an AZ goes down, it doesn't impact traffic serving capability from other AZs.
- Another reason for NAT Gateway division at AZ level could be avoiding packet drops for traffic greater than 10 million packets per second.
- NAT gateway can't be used by external internet to initiate connection with instances in private subnet.
- NAT gateway provides public(default) and private connectivity. Public will be useful for connectivity with outside internet while private will be useful for connection with other VPCs or on-premise networks.
- NAT Gateway comes with additional infrastructure cost unlike Internet Gateway, whose pricing only depends on traffic flow via it.

As we discussed connectivity to personal data centers(meaning VPC), there could also be scenarios where we own multiple such data centers. The follow up section covers different mechanisms available in AWS to create cross VPC connectivity,

## Amazon VPC to Amazon VPC Connectivity

You may have use cases such as AWS resources integration across VPCs, security, presence in multiple regions, etc. where different components need to connect with each other residing in

different VPCs, which could be in the same or different AWS accounts while maintaining isolation. To return to our cafe example, Cafe Delhi Heights is becoming more popular day by day and more people are coming in, so the restaurant head opened a new place for customers to sit—however, the food is still prepared in the old kitchen. Now here, there is a requirement to establish good connectivity for faster delivery of food to the newly opened location, in a secure way so as no one finds out. In similar fashion, there can be a need to operate two microservices in different VPCs(or different AWS accounts altogether) and for these microservices to communicate with each other in a secure way, AWS provides different connectivity options.

These connectivity options mainly fall under two kinds of relationships: many-to-many and hub-and-spoke. Traffic is managed individually between each VPCs in many-to-many VPC relationships, whereas a central resource manages traffic routing between the VPCs in hub-and-spoke model. The follow up section explores both of these models.

### Amazon VPC Peering

**VPC Peering** is based on a many-to-many approach where one VPC peers with another VPC to enable full bidirectional private network connectivity. It enables resources in different VPCs to communicate with each other using private IP addresses as if they were in the same network. VPC Peering doesn't support transitive dependency and can be a cost effective interconnectivity method if the number of VPC is less than 10. As the number of VPC increases, the mesh can become really complex to manage and operate.

You can manage the connections via route tables, SGs and NACLs to allow specific resources or subnets to utilize the VPC peering connection. The network packets between the VPCs flow via AWS private network with no bandwidth constraints—there is no physical hardware required for this setup, you only pay for the amount of data transfer.

Think of a scenario where all the Cafe Delhi Heights restaurants have their own bar, but food is prepared at a central location and supplied to all the smaller setups, extending this example to a system architecture consisting of web frontend and backend. The connectivity between the backend and frontend tier can be established via VPC Peering assuming both are hosted in separate VPC. [Figure 6-7](#) shows connectivity between servers running in different VPCs via VPC Peering connection.

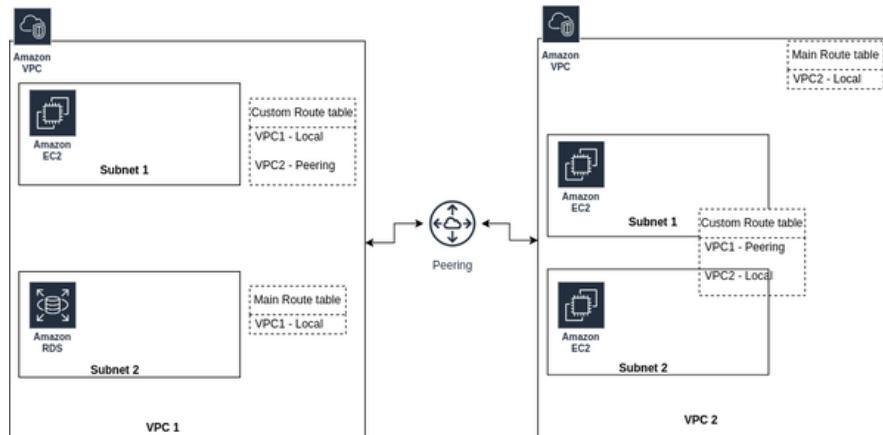


Figure 6-7. VPC Peering

The downside of establishing connection with VPCs at large scale via VPC Peering is resolved with another AWS Service called AWS **Transit Gateways**.

### AWS Transit Gateways

Transit Gateway is a scalable solution to establish connectivity between multiple VPCs, on-premise networks and other AWS services. Transit Gateway is a regional resource based on hub and spoke model which acts as intermediary to set up all the network routing at a single place via routing tables, be it VPCs or hybrid connectivity methods such as Virtual Private Network (VPN) or AWS Direct Connect.

For better control of networking routes, Transit Gateway can be set up in a separate networking AWS account where network engineers can manage at centralized location.

**Figure 6-8** shows Transit Gateway acting as a central resource for providing connectivity between VPCs, AWS Direct Connect and VPN Connection.

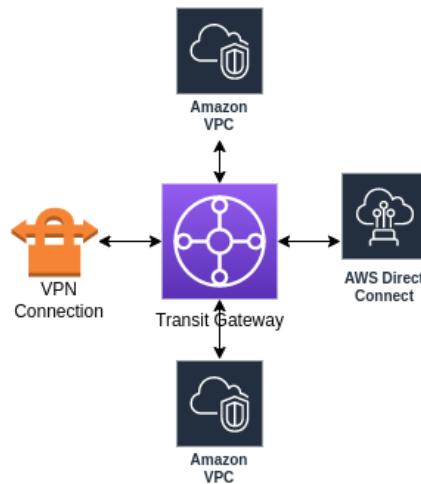


Figure 6-8. AWS Transit Gateway

The key difference between Transit Gateway and VPC Peering is scale—transit gateway is scalable for connectivity across thousands of VPCs. Other parameters are favorable to VPC peering such as lower cost, no bandwidth constraints and reduced latency.

**Figure 6-8** depicted frontend and backend tier connectivity via VPC Peering which are present in different VPCs—another solution for this kind of setup is utilizing AWS **PrivateLink**.

### AWS PrivateLink

AWS PrivateLink helps to privately expose an application to consumers in another VPC and ensures traffic flows in AWS backbone network without going over the internet. The key difference is traffic flow direction—VPC Peering provides bidirectional connectivity, but if clients need to server requests only using private IP addresses, AWS PrivateLink will facilitate this kind of connectivity.

#### *Connectivity via AWS PrivateLink*

AWS PrivateLink is established between two parties: the one which allows access to its specific service can be referred as the service provider and others consuming this service are considered the consumers.

### *Service Provider*

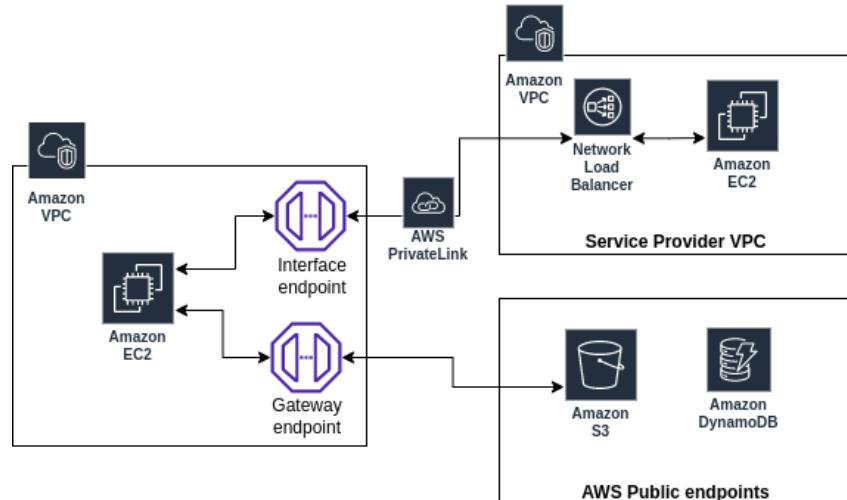
To make service available in a region, service providers will create an endpoint service which is mapped with a network load balancer. Application Load balancer can't be directly attached with endpoint service.

### *Service Consumer*

Service consumers create a VPC endpoint to connect their VPC to endpoint services by specifying the service name which is being created by the service provider.

As represented in [Figure 6-9](#), the endpoint could be an interface endpoint or gateway endpoint. An Interface endpoint helps to establish connection to an endpoint service in another VPC, where Gateway helps to connect with Amazon S3 or DynamoDB using private IP addresses.

One more type is gateway load balancer which can be used to manage third party virtual appliances, for example compliance. (More on load balancer shortly.)



*Figure 6-9. Connectivity via Endpoint Services*

Apart from establishing connectivity between multiple VPCs, there are certain businesses which still operate on on-premise data centers. In the process of migrating to Cloud, there can always be an intermediary state where some operations are served by on-premise data centers and rest via AWS Cloud. Let's explore different solutions offered by AWS to establish connectivity between these two separate data centers.

## **Hybrid Connectivity**

Current infrastructure of Cafe Delhi Heights is that food is prepared at a central location and a bar is located at all the locations. To make our systems more efficient, we're planning to move most of our food preparation to all the locations as well, but still some of the key dishes will be prepared at the central location. In this scenario, the operations happen at two places, this is what we mean by hybrid connectivity.

It could be possible that you started with your personal data center but over time you decided to move to AWS Cloud. Now, the service infrastructure is maintained by AWS but the databases are still managed on an on-premise data center. To support this, there needs to be some mechanism to establish connectivity between on-premise data centers and AWS data centers—that's a Virtual Private Network.

## AWS Virtual Private Network

What mechanism can be used to securely connect on-premise data centers to AWS VPC? The service framework which ensures this data security is Virtual Private Network, aka VPN.

**Figure 6-10** describes an example connection setup using AWS VPN from customer data center to VPC located in an AWS region.

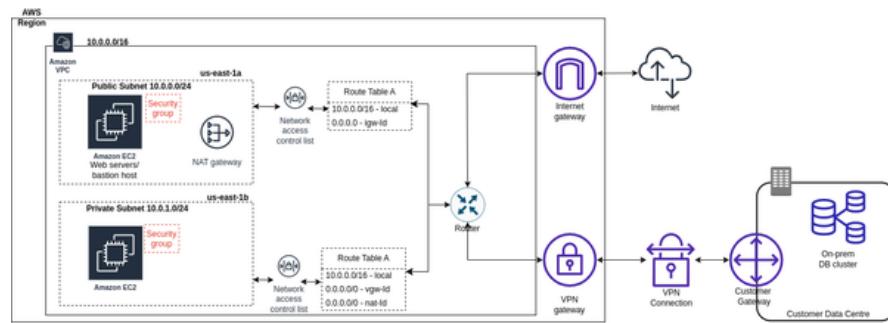


Figure 6-10. VPN Connection

The components for VPN setup are:

### Customer Gateway

A software or hardware component which is setup at the customer's end. You can read more about device requirements [here](#).

### VPN Connection

A secure and encrypted connection channel between the customer gateway and VPN gateway. There are two tunnels being setup so as to avoid any availability issue due to failure or scheduled maintenance.

### VPN Gateway

This component is present at AWS side to ensure communication between VPC and VPN connection.

VPN transfers data packets over the internet anonymously. AWS provides another service to configure dedicated networks for connectivity, referred to as AWS Direct Connect.

## AWS Direct Connect

AWS Direct Connect provides the capability to configure a dedicated network connection (Ethernet fiber-optic cable) for data transfer from the customer data center to an AWS direct connect location without the use of the internet. AWS direct connect location is configured with a router to route the traffic and connect to the AWS backbone network. It provides consistent and high-bandwidth connectivity, suitable for large-scale or latency-sensitive

workloads. You can use [AWS Direct Connect Resiliency toolkit](#) to ensure maximum availability of connections set up from personal data center to AWS Direct Connect Location. [Figure 6-11](#) is an extension of [Figure 6-6](#) and describes an example connection setup from customer data center to Amazon VPC via Direct Connect location. The main components are:

#### *Customer Router*

The customer router is installed at an on-premises data center holding all networking rules and helps routing the traffic from data center to AWS Direct Connect Location. Customer Router connects with router at Direct Connect Location via 802.1Q VLAN ethernet cable.

#### *Direct Connect Location*

Direct Connect is available worldwide at multiple [locations](#). We can select a location closest to on-premise data center to minimize cost and latency. You can check for all networking requirements on [AWS doc](#).

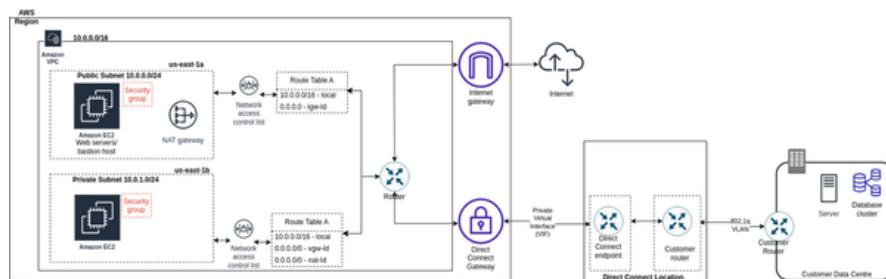
#### *Direct Connect Gateway*

Direct Connect Gateway helps in establishing connection between VPC and Direct Connect Location via private virtual interface.

#### *Virtual Interface(VIF)*

Virtual Interfaces are helpful in setting up private secure connections to required resources such as S3 without going over the internet. You can select either Public VIF, Private VIF or transit VIF depending on use-case.

- Public VIF is helpful in connecting to AWS resources over public IP addresses such as Amazon S3.
- Private VIF is helpful in connecting to AWS resources using their private IP addresses hosted in AWS VPC.
- Transit VIF is helpful in connecting to AWS resources using their private IP addresses hosted in AWS VPC through the transit gateway.



*Figure 6-11. AWS VPC connectivity with On-prem data center via Direct Connect*

We explored multiple concepts and AWS services revolving around how network packets flow inside the Amazon network, as well as multiple connectivity options. The rest of the chapter focuses on the entry point for these network packets as applications scale such as [Route 53](#), [Load Balancers](#), [API Gateway](#). We'll conclude the chapter by discussing AWS provided Content

Delivery Network referred as **AWS CloudFront** for placing data content near to users to enable faster retrieval.

## Amazon Route 53

Route 53 is a scalable and highly available Domain Name System(DNS) available in the AWS ecosystem which helps in domain registration, DNS routing and health checking. We started our 'IP Addresses' section by stating that every device on the internet requires an IP address to establish connection with other devices, but what if a human being is operating one side of a connection? It can be very difficult to remember all the IP addresses, well unless you've photographic memory.

Human beings are good at remembering names as compared to numbers—it's easier to remember [www.google.com](http://www.google.com) instead of 192.168.1.0 or to remember Cafe Delhi Heights, Ambience Mall instead of latitude and longitude as 28.525446566084423, 77.09008858115097. Domain Name System makes our lives easy by facilitating the conversion from domain name to IP addresses. **Figure 6-12** shows how a domain such as [www.google.com](http://www.google.com) is resolved to an IP address and finally accessible to end users. After the DNS resolution in Step 1 to 5, the browser establishes TCP connection with google web servers and then sends a HTTP request to the servers. The server handles the request and sends back an HTTP response as final step 7. The browser renders this response in the browser tab for further user interactions. We should have this general understanding of what happens when we type in [www.google.com](http://www.google.com) in the web browser to understand the data flow on the internet.

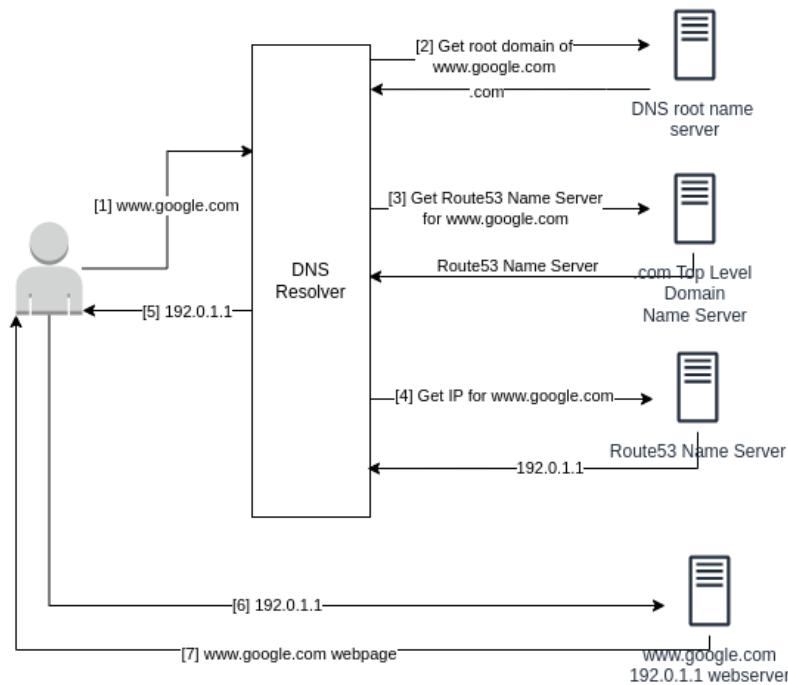


Figure 6-12. DNS resolution via Route 53

These are the key considerations about Route 53:

- Route 53 can be used to register new domain names and additionally you can transfer existing DNS to be managed by it.
- You can create, update and manage your public DNS records via AWS Route 53 along with health checks to monitor the health of applications, web servers and related resources.
- You can configure **routing policies** for traffic management on DNS records, specifying how Route53 responds to queries.
- Route53 supports various DNS record types such as A, AAAA, CNAME, MX, TXT, and more, enabling flexible DNS configuration.
- DNS records of your domain are collectively stored in Hosted Zones to answer the domain queries.
- Route 53 utilizes Anycast as networking and routing technology, which helps in reduced latency via routing requests through the nearest data center and higher availability via presence of multiple servers to respond to traffic instead of just one origin server.

Next we'll turn our attention to the AWS Elastic Load Balancer.

## Amazon Elastic Load Balancer

As discussed in Chapter 5 on Scaling Approaches and Mechanisms, Load Balancers help improve the availability, scalability, and fault tolerance of applications by distributing traffic across healthy targets. Load balancers can automatically scale based on traffic patterns and health checks, ensuring optimal performance.

The AWS Elastic Load Balancer (ELB) is a managed service which scales for the customer's traffic on the go and provides the following capabilities:

- Distribute the incoming traffic among a pool of resources.
- Capability to serve requests without disruption as new resources are added or old ones removed.
- Monitoring resource health via ELB provided health checks.

AWS ELB is available as Application Load Balancer (ALB), Network Load Balancer (NLB), Classic Load Balancer(CLB) and Gateway Load Balancer(GWLB). As the name suggests, ALB operates at application layer(Layer 7) for HTTP, HTTPS and gRPC protocols, NLB operates at network layer(Layer 4) for TCP, UDP and TLS protocols and CLB is a legacy version of load balancer which supports both layer4/layer7 traffic. GWLB is used as a Layer 3 Gateway and Layer 4 load balancer for IP protocol.

You can read a detailed product comparison on [AWS](#).

Our recommendation is to take a bottoms up approach by thinking about the base feature requirements for a given workload. These requirements along with pricing will decide which load balancer type is best suitable to your workload—there can be additional features offered by ELB that will be cherry on the top. [Figure 6-13](#) shows traffic distribution by ELB to different types of targets. The main components of ELB are:

## *Load Balancer*

LB is a single point of contact from the client's perspective, it can then further forward the request to configured listeners.

## *Listeners*

Listener is a process that checks for the client's request using the configured protocol and port number. The request is forwarded to target groups based on the rules associated with a listener.

## *Target Groups*

Target Groups directs traffic to configured targets using the configured port and protocol, for example EC2 instances, IP addresses, etc.

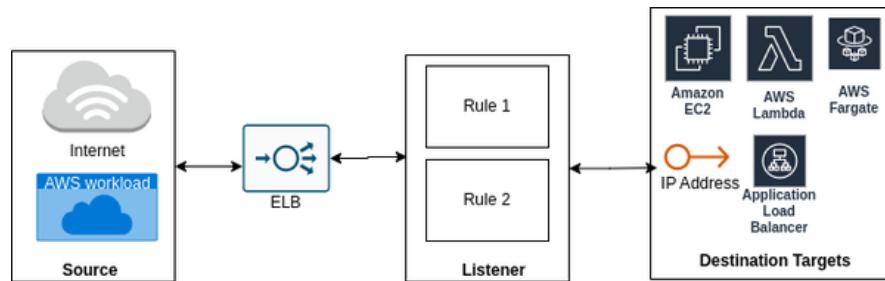


Figure 6-13. Elastic Load Balancer

A few key considerations about ELB are:

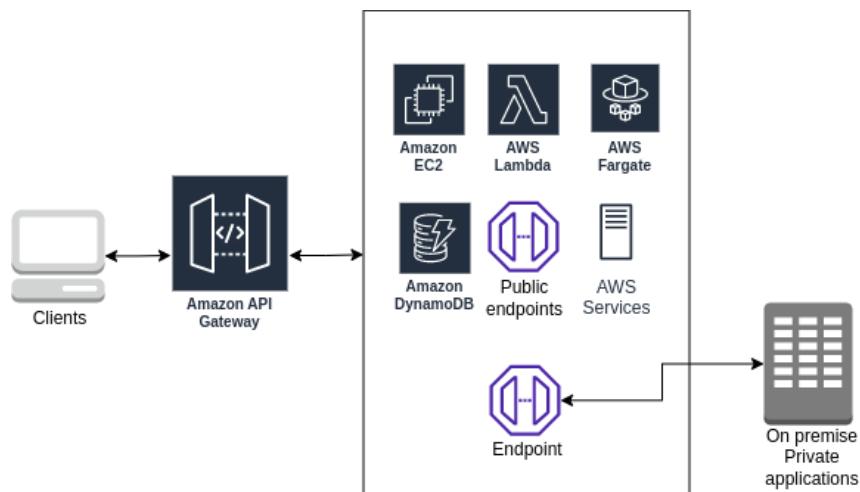
- NLB can have targets such as ALB, containers, instances and IP. ALB can have targets as containers, instances, IP and lambda, and GWLB can have targets as IP and instance.
- Gateway Load Balancers are useful in systems such as firewalls, intrusion detection and prevention systems, deep packet inspection systems, etc. It enables customers to deploy, scale and manage virtual appliances. One example scenario for inspection systems is placement of GWLB in between source and destination for packet analysis and monitoring.
- ALB doesn't provide support for private link and static IP addresses for inbound traffic—NLB is a suitable choice for this use-case. In scenarios where additional features of ALB are value-add to NLB or vice-versa, ALB can be used as a target to NLB to gain benefits from both worlds.
- As ALB has request header data, it can support request routing in multiple ways, such as path based, host based, query string parameter based and source IP address based routing. In contrast to this, NLB doesn't have capability to inspect the HTTP request which makes it a little lighter in processing as compared to ALB resulting in reduced latency.
- NLB is optimized to handle sudden spiky traffic patterns; for ALB it's a best practice to inform AWS Support in advance if a traffic spike is expected to pre-allocate the capacity.

- NLB supports long lived TCP connections unlike ALB. This is helpful in scenarios with requirements such as a huge number of persistent connections, such as WebSocket connection for an online gaming application.
- ALB provides support for AWS Web Application Firewall(WAF) and authentication mechanisms such as Amazon Cognito, OpenID Connect, etc. This helps in offloading this responsibility from the application and making it lighter.
- The load balancer will reside in a public subnet and you can host the backend resources in a private subnet, which are not directly accessible to outside traffic.
- Idle Timeout configuration for ALB can be set between 1 to 4000 seconds (the default is 60 seconds) where for NLB this configuration is 350 seconds.

Now let's turn our attention to the API gateway.

## Amazon API Gateway

API Gateway is a fully managed AWS service that helps in creating, publishing, maintaining, monitoring and securing REST, HTTP and WebSocket APIs. Consider a scenario that instead of human beings serving your order at Cafe Delhi Heights, there are robots deployed who take and serve the food orders. You order food by selecting the food items from the kiosk installed at the table and once the food is prepared, it is served by robots to you—well in this scenario you don't know if in the kitchen the food is prepared by robots or human beings. That's the beauty of resource abstraction, customers use the API Gateway's published API to perform specific functions at scale and API Gateway can internally connect with any AWS service as shown in [Figure 6-14](#).



*Figure 6-14. API Gateway connectivity with AWS Services*

There are a few key considerations with API Gateway:

- API Gateway supports both stateless and stateful APIs.
- REST and HTTP APIs are type of stateless APIs. These both support the same basic functionality and are HTTP based to support standard methods like GET, PUT, POST,

etc. REST APIs support some additional features such as API keys, per-client rate/usage throttling, request validation etc.

- WebSocket APIs are stateful which operate on the basis of WebSocket protocol with full duplex client-server communication.
- API Gateway can support authentication via AWS IAM(Identity and Access Management) policies, Lambda authorizer functions and Amazon Cognito user pools.
- API Gateway provides monitoring via CloudWatch and CloudTrail services.
- API Gateway can avoid web exploits such as SQL injection via AWS Web Application Firewall(WAF) integration.
- API Gateway can directly connect with AWS Services such as DynamoDB(DDB) via service APIs reducing intermediary infrastructure cost. Consider a scenario of retrieving DDB record basis partition key. The general implementation would be API Gateway invokes a Lambda which then connects with DDB to get data. API Gateway removes the need for AWS Lambda in between and can directly connect with DDB and serve responses.
- API Gateway doesn't provide health checks for backend resources the way it is supported via ELB.
- API Gateway REST APIs timeout configuration lies between 50 milliseconds to 29 seconds, 30 seconds for HTTP APIs and 2 hrs connection duration for WebSocket API with idle timeout as 10 minutes.
- API Gateway can be used for cross account/region integration—for example, AWS Lambda owned by different teams in different AWS accounts as per business requirements but served by central AWS account API Gateway.
- API Gateway supports caching of endpoint's responses which helps in reducing traffic to endpoint and improvement in latency. Time to Live(TTL) for caching can vary from 0(caching is disabled) to 3600 seconds with default as 300 seconds.

API Gateway and Load Balancers help to abstract out backend infrastructure from customers. Consider a scenario where a customer's requests hit API Gateway and further one of the microservice fetches data from S3, this operation is definitely latency intensive if the file size is large. The latency can be reduced by placing the content near to the customer's location via the AWS Content Delivery Network service referred as CloudFront.

## Amazon CloudFront

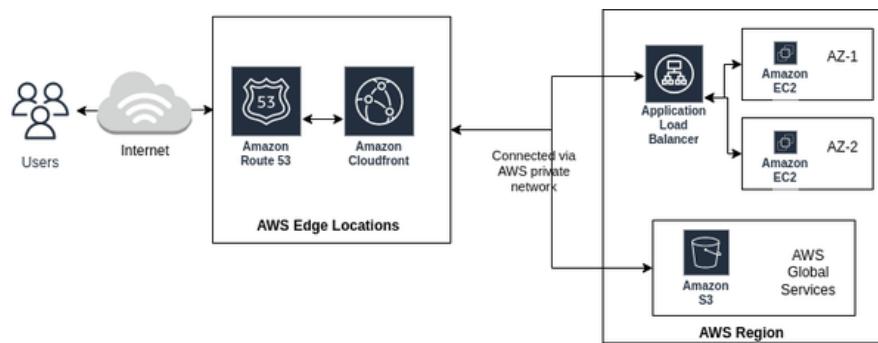
We briefly touched on two ways to manage food preparation and delivery operations for Cafe Delhi Heights, first one was food is prepared at a central location and delivered to smaller storefronts as per demand and second was to set up kitchens at all the locations.

There is definitely an extra cost to set up kitchens at every location—another option is analyzing the food item demands and preparing the food in advance at a central location and storing it at smaller outlets ahead of time. This way the food is prepared at only a single place and the customers are served at all the outlets in the minimum time possible.

In software systems, one key factor to improve customer's experience is via serving the request with minimum latency. One potential solution is to replicate infrastructure in multiple AWS regions and utilize AWS local zones. This solution should only be accessed as a last resort for your business architecture and latency should not be the key factor to finalize it. Another cost effective solution is to cache the content near the customer's location via a Content Delivery Network (CDN), as discussed in Chapter 4. Amazon CloudFront is CDN, a world-wide network of data centers called edge locations which helps to achieve low latency for serving both static and dynamic content in a secure way via AWS Shield, IAM, WAF and TLS certificates.

There are a few things you need to keep in mind about CloudFront:

- CloudFront can be used to serve both static and dynamic content over HTTP or WebSocket protocols. For example, static content placed in S3 buckets or dynamic content generated via any web service such as running on EC2 servers can be directly served via CloudFront as described in [Figure 6-15](#).
- CloudFront by default integrates with AWS Shield to help avoid DDoS attacks and additionally can be integrated with AWS WAF for application layer security.
- CloudFront provides both encryption at rest and in transit. Data at edge locations is always stored in encrypted format and Amazon Certificate Manager or custom certificates can be used for in-transit traffic.
- For ensuring user level access to content, CloudFront provides options for signed cookies, signed URLs and geo-restriction.



*Figure 6-15. Content Distribution via Amazon CloudFront*

In short, Amazon CloudFront allows us to cache the content near our application users and serve the user queries faster.

## Conclusion

We started off this chapter with discussion around how you can get started with AWS Cloud and then we dove deep into different networking concepts and services offered by AWS to set up networking infrastructure on the AWS Cloud. AWS Cloud comes with a plethora of services and it's really important to understand how a specific service will solve a problem statement. It's not easy and feasible for everyone to set up their own personal data center and AWS solves this problem by offering Amazon VPC as a solution.

You can assume Amazon VPC as your personal data center. There may be requirements for multiple data centers to connect and AWS provides different connectivity mechanisms to enable this support. It is also important that only intended users access the resources hosted in AWS Cloud and this can be ensured by utilizing security mechanisms such as security groups, NACLs and adding appropriate rules in the route tables.

Finally we discussed how the internal resources can be abstracted out from external users by using solutions such as Amazon ELB and Amazon API Gateway. AWS also offers a highly scalable DNS service, Amazon Route 53 for managing domain names. Consider a simple example to illustrate the usage of Amazon Route 53—let's say we started off with using NLB and the users hit this endpoint. At a later point, we realized ALB was a more appropriate choice instead of NLB. In this situation, there are three options—we could share the new ALB endpoint with customers, keep NLB as front-facing to ALB with ALB redirecting traffic to the application, and use Amazon Route 53 domain so the same domain name is accessible to customers with no impact on how internal resources change over time.

Data storage solutions are an important aspect to consider while storing and maintaining data in the cloud. In the next chapter, we'll explore different types of AWS Storage services such as DynamoDB, S3, Relational Databases and further discuss each of these services with respect to the use cases they best serve.

# Chapter 7. AWS Storage Services

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

While Chapter 2 and Chapter 3 of this book illustrate multiple storage types and solutions, this chapter explores AWS-specific storage **services**, which map to different kinds of storage solutions present in the market. We'll start our deep dive with traditional storage services like block storage and then move towards various kinds of databases used to store data identifying business and technical requirements.

Referring back to our real-life example of Cafe Delhi Heights from Chapter 9, Cafe Delhi Heights is going online where customers can look for available food items and place an order from the chain of restaurants. To provide the best customer experience and food delivery services online, Cafe Delhi Heights will store data such as restaurant images, food menu, marketing campaigns information,

etc. on AWS storage services. Cafe Delhi Heights has given us the job to make this happen. Let's dig into identifying their storage requirements and choosing the best storage solution from a pool of services offered by AWS. We need to be able to:

- Store a customer's profile and access details like username and password.
- Store food information such as available menu items.
- Store different kinds of media, such as images of restaurant food items, customer uploaded reviews, etc.
- Perform big data analytics to improve customer experience, like separation of good and bad reviews.
- Create food communities and allow people to interact in social circles.
- Search for food items based on multiple identifiers such as food name, restaurant location, ratings, etc.
- Archive application logs and metrics after ninety days and have them persist for one year.

#### NOTE

Please read through Chapter 2 and Chapter 3 of this book to understand the key storage concepts first. This chapter is built on top of those chapters and dives deep into AWS services mapping to concepts we discussed in there.

AWS Storage services enable customers to store, protect, and analyze data without the worry of any operational overhead. We'll discuss storage services in two major sections—cloud storage and databases. We'll start with cloud storage and dive deep into block storage, file storage, and object storage services.

# Cloud Storage on AWS

AWS offers multiple options for cloud storage: Amazon Elastic Block Storage (EBS) as block storage service, Amazon Elastic File Storage (EFS) as file storage service and Amazon Simple Storage Service (S3) as object storage service. We discussed the general concepts about these storage options in Chapter 2 and in this section, we'll explore the AWS services offered to support these solutions. Each of these services can cater to specific storage requirements and we'll dig deeper into the benefits each service can provide in the following sections.

## Amazon Elastic Block Storage

**Amazon EBS** is a block storage solution which acts similar to any physical hard drive attached to a personal computer. Similarly Amazon EBS can be attached to an Amazon **EC2** instance (more on EC2 in Chapter 11) whose lifecycle is independent of Amazon EC2 instance and is flexible in nature, meaning, it allows configuration modification on live production workloads such as:

- Dynamic increase in size.
- Change provisioned IOPS capacity.
- Change volume type.

EBS volumes provide access to data stored on disk with minimum latency and is preferred for use-cases requiring frequent disk access such as:

- Creating and maintaining your own database instead of using any services offered by AWS Cloud.
- Operating System boot volumes.

For use-cases which require temporary block storage and better I/O performance than EBS (such as caching, device buffers, etc), AWS

offers another storage solution called **Instance Store** which is directly attached to an instance, unlike EBS which is network-attached storage. The data in the instance store is wiped out in scenarios of disk failures, EC2 instance state update to stop/hibernate/terminate. Instance Store doesn't come with the flexibility similar to EBS volumes, such as attaching to any EC2 instance, dynamic increase in size, etc.

AWS offers a wide range of **EBS volumes** to choose from based on multiple factors, such as cost and performance. Let's explore the scenarios in which we can use specific volume types:

### *Solid State Drive(SSD)*

These volume types include General Purpose SSD and Provisioned IOPS SSD which are optimized for transactional workload. Their major performance attribute is small I/O size.

### *Hard Disk Drive(HDD)*

These volume types include Throughput Optimized SSD and Cold HDD which are optimized for large streaming workloads. Their major performance attribute is throughput.

### *Magnetic Disks*

These volume types are mostly suitable for smaller workloads where performance is not a key factor. For example, infrequent data access from storage disk.

Here are few key points of consideration about EBS volumes:

- Multiple EBS volumes can be attached to an EC2 instance.
- The scope of EBS volume is at Availability Zone level. An EBS volume in us-east-1a can't be attached to an EC2 instance in us-east-1b AZ. We can create a point-in-time snapshot(backup) of

EBS volumes to enable replication and ensure high availability across AZs or regions.

- EBS volumes were first launched as a non-sharing block storage, meaning an EBS volume can't be shared among multiple EC2 instances. AWS launched the Multi-Attach **EBS** volumes feature allowing users to attach a single Provisioned IOPS SSD volume to max of sixteen **Nitro-based EC2** in the same AZ for Linux applications.
- EBS volumes are automatically replicated within AZ to prevent any data loss.

There are certain limitations of EBS volumes such as scope at AZ level and data sharing across EC2 instances. AWS offers another storage solution, referred to as AWS EFS to overcome these limitations.

## **Amazon Elastic File Storage**

**Amazon EFS** is a shared file system which allows storage sharing across multiple servers that are placed in a region or at an on-premise data center. Amazon EFS is a fully managed serverless solution with no management overhead and supports all AWS Compute platforms such as EC2, ECS or Lambda. There is no need to pre-configure storage space and pricing is determined based on the storage space we utilize for our applications.

Amazon EFS is generally used for business use-cases such as big data analytics, machine learning workload, content management, etc. Both EFS and EBS storage solutions offer low latency for data access, though there are multiple contributing factors in overall performance, such as latency, throughput, and Input Output per second(IOPS).

EFS offers multiple **storage classes** and we can select a storage class that best fits our business needs:

### *EFS Standard and Standard-Infrequent Access(Standard-IA)*

Both of these storage classes provide the highest level of availability across the AZs in an AWS region. EFS Standard is optimal for frequent data access use-cases whereas Standard-IA is the better choice for infrequent data access and a cost optimal solution. The latency of first byte read or write is higher for IA storage class type.

### *EFS One Zone and One Zone-Infrequent Access(EFS One Zone-IA)*

The parameter on which One Zone storage classes differ from Standard storage classes is availability and the cost you pay. One Zone ensures high availability within a single AZ whereas Standard spans across the AZs within a region. Due to this factor, One Zone is relatively cheaper. AWS Backup can be used further for better durability which helps in replicating the data across three AZs.

Cloud is still relatively new technology to host your business applications, and there are still scenarios where you'll rely on other file systems such as Windows File Server. For such business use-cases, AWS offers a managed service—**Amazon FSx** which scales automatically without any operational overhead.

### **Amazon FSx**

We can leverage Amazon FSx to run popular open source and licenced file systems without the worry of maintenance or hardware setup. There are two FSx variants and we can select either of them as needed per our use-case:

#### *Amazon FSx for Windows File Server*

This is built on Windows Server and is accessible over Server Message Block(SMB) protocol from Windows, Linux, or MacOS to

execute business use-cases such as data deduplication, end-user file store, and Microsoft Active Directory(AD) integration.

### *Amazon FSx for Lustre*

This is built on a popular high-performance file system, [Lustre](#). This works well for compute intensive workloads such as machine learning, high-performance computing, video rendering, etc. Additionally it can be linked with Amazon S3 allowing access and process of data concurrently from file system as well as S3 API.

A server is required for accessing data from EBS and EFS storage services. Amazon S3 is another unlimited storage solution offered by AWS which allows users to directly access the files over the public internet or private AWS network without the requirement of a server in between.

## **Amazon Simple Storage Service**

AWS Cloud provides Amazon Simple Storage Service(S3) as an unlimited object storage solution. We can use S3 to store and protect data for a wide range of use-cases, such as websites, media storage, backups, big data analytics, etc. S3 allows storage of objects in containers referred to as buckets. Let's cover some key concepts in regard to S3 that will help deepen your understanding:

### *Bucket*

A bucket is a container, identified by a unique bucket name. You need to specify the bucket name and AWS region during bucket creation. There are also other options to specify, such as versioning support.

### *Object*

An object is a file that is uploaded to a bucket, which is identified by a key name that is unique across the bucket. An object in the

bucket has exactly one key as an identifier.

### *Versioning*

S3 also allows you to store multiple versions of the same object. This allows users to access any object version and is helpful in scenarios such as application failure. A unique version ID is associated with each version of the object.

The data storage pattern can vary from use-case to use-case therefore S3 offers multiple storage classes. We can select a storage class based on business requirements such as: Do we need to access the data frequently? Or do we require a data archival store? Let's dig into the different storage classes here.

## **Amazon S3 Storage Classes**

It is important to identify your business use-case and then choose the **S3 storage class** that best fits in order to make the most out of S3 with minimum cost. Below are the storage classes offered by S3 based on the object access type and redundancy offered:

### *Frequently Accessed Objects*

This storage class is preferred for latency-sensitive(at the millisecond level) business use-cases. There are two subclasses based on required redundancy support –

- S3 Standard – This is the default storage class on bucket creation.
- Reduced Redundancy – This storage class can be chosen for noncritical business use-cases with reduced redundancy as compared to S3 Standard. AWS recommends the S3 storage class over Reduced Redundancy as it is also more cost effective.

### *Infrequently Accessed Objects*

This storage class is preferred for long lived and infrequently accessed data such as backups. There are two subclasses based on required redundancy support –

- Standard-IA – Data is stored redundantly in multiple AZs so objects are resilient to a loss of AZ.
- One Zone-IA – Data is stored in single AZ so it is less resilient as compared to Standard-IA, which also makes it relatively cheaper. This storage class is preferred if data loss is not critical for business or data can be re-created in some way.

### *Archived Objects*

This storage class provides low-cost data archiving support with resiliency similar to S3 Standard storage class. There are three subclasses based on data retrieval needs –

- S3 Glacier Instant Retrieval – This is useful for rarely accessed data that requires millisecond retrieval. The storage cost is less and data access cost is at the higher end if we compare this storage class with Standard-IA.
- S3 Glacier Flexible Retrieval – This is useful for data accessibility in minutes (1-5 minutes). It requires a minimum storage duration period of 90 days, so even if data is removed or transitioned before 90 days, the cost will be calculated for 90 days.
- S3 Glacier Deep Dive – This storage class is the least costly, requires minimum storage duration of 180 days, and has a default retrieval time of 12 hours. You can also use the bulk retrieval option with data being retrieved within 48 hours.

Consider a scenario with a requirement of frequent data access for 30 days and infrequent access after it. Such use-cases can be satisfied via S3 Lifecycle configurations—these configurations are the

rules that apply to bucket objects. There are two type of actions that you need to define as part of these configurations:

#### *Transition Actions*

Define object transition to another storage class, such as move object from Standard to Standard-IA storage class after 30 days.

#### *Expiration Actions*

Define object expiration. S3 will delete objects automatically once they are expired as per configured rule.

Optionally, S3 provides intelligent-tiering support so data is automatically moved to different storage classes without any operational overhead. Here we've outlined the lifecycle followed by objects to be moved to different storage class:

#### *Frequent Access*

Objects uploaded or transitioned to S3 Intelligent-Tiering are stored in the Frequent Access tier.

#### *Infrequent Access*

Objects are moved to the Infrequent Access tier if they're not accessed for consecutive 30 days.

#### *Archive Instant Access*

Objects are moved to the Archive Instant Access tier if they're not accessed for 90 consecutive days. S3 provides two optional archive access tiers based on time period of data access. If activated, data is moved to Archive Access if not accessed for a minimum of 90 consecutive days and moved to Deep Archive Access if not accessed for a minimum of 180 consecutive days.

Storage classes help to store data to S3 based on access patterns or required redundancy support. Another factor to consider is data security while storing objects to S3. The next section explores different options S3 offers to secure your data.

## **Amazon S3 Data Security**

Objects in S3 bucket are by default accessible to the owner of the bucket. S3 provides different mechanisms to secure data and configure access permissions to S3 bucket and objects.

### *Encryption*

S3 encrypts all objects by default with server-side encryption (SSE-S3) unless a different encryption option is selected, such as using [AWS Key Management Service \(KMS\)](#) referred to as [SSE-KMS](#). Additionally, you can configure client-side encryption as well for more security.

### *Object Lock*

[Object Locks](#) can be enabled on versioned buckets. Once enabled, objects are stored using a write-once-read-many (WORM) model which could be a regulatory requirement for an organization. Object Locks provide an extra protection layer on object changes and helps in preventing object deletion or overwriting for a fixed retention period or indefinitely (referred to as Legal hold).

### *Block Off Public Access*

This option is useful for managing and setting up centralized control on public access to the bucket. This configuration can also be applied at the account level so that it is reflected for all S3 buckets in the AWS account. In general, it is not a good idea to make a bucket public unless there is a specific business requirement such as a bucket with publicly accessible media files.

We can use **Access Analyzer** for S3 to identify buckets that grant public access.

### *IAM Policy*

**IAM Policy** is helpful in granting users or groups read-write access to S3 resources.

### *Amazon Macie*

Amazon **Macie** is helpful in identifying and securing sensitive data stored in S3 buckets, such as credit card numbers. It also helps in evaluating bucket-level access controls (similar to S3 Access Analyzer) such as publicly accessible buckets.

### *ACLs*

**S3 ACLs** are attached to an S3 bucket or object as a subresource and helps in managing access to other users. The ACLs are disabled by default, and it is recommended to keep them that way unless there is a requirement to control access of each object individually. Instead of ACLs, we can rely on **policies** for access control.

### *Bucket Policy*

Bucket policy helps in managing access permissions to an S3 bucket and objects present in it. For example, it can allow account A to read objects from an S3 bucket owned by account B. **Here** are a few examples for S3 bucket policies. Bucket policies are limited to 20 KB in size.

We covered different services that allow you to store data with cloud storage and process it similarly to using storage on a personal computer. Next we'll discuss the database services offered by AWS. You should already be familiar with how data is stored in databases

from our coverage in Chapter 2 for relational databases and Chapter 3 for non-relational databases, so now we'll dig into AWS specifics.

## AWS Databases

AWS offers a range of database services, such as Amazon Relational Database Service (RDS), Amazon DynamoDB, Amazon DocumentDB, Amazon ElasticCache, and more, to meet customer requirements for different business use-cases. These services minimize or completely remove any operational overhead of software and hardware management and enable us to gain the needed scale for our organization.

The following sections will go into AWS's database services in more detail. We'll start with their relational database offering, which falls under the umbrella service RDS with support for Oracle, Microsoft SQL Server, MySQL, PostgreSQL, MariaDB and Amazon Aurora as database engines.

### Amazon RDS

Without Amazon RDS, you have a couple of options when you want to run relational database engines like MySQL or PostgreSQL, which were covered in Chapter 2. Should you want to use, for example, a MySQL database engine on your local machine, you'd need to download the installation package and install it on your machine. If you're using AWS Cloud, you could install the package on an EC2 machine—but once it's up and running you'll also have to take care of maintenance, attaching EBS volumes, etc.

Amazon RDS helps us remove all this overhead so we can just focus on creating a DB instance with a few clicks and start using it. You can follow the steps from the AWS [documentation](#) for creating RDS DB instances, but there are a few details that are important to keep in mind during database creation.

## *Engine Type*

We can select the database engine that meets our needs, and once we select a specific engine, we are required to specify some additional configurations such as Edition and Engine Version.

## *Templates*

Templates help in pre-selecting the setting options while setting up a database—the available templates are Production, Dev/Test, and Free tier.

## *Settings*

Settings include details like DB instance identifier (which is unique across AWS account in the region) and Credentials settings (username and password).

## *DB Instance Class*

DB instance class determines computation and memory capacity of an instance. We can select from a list of **standard, memory-optimized, and burstable classes** depending upon our workload's processing and memory requirements.

## *Storage*

Storage configurations allow you to select storage type from SSD, HDD or Magnetic disks, and storage allocation number with auto scaling enabled or disabled.

## *Availability & Durability*

To ensure high availability of production databases, RDS allows multi-AZ deployment options to create a standby instance in different AZ. This helps to overcome any database or AZ failure.

## *Connectivity*

Connectivity configurations allow you to select a VPC DB instance and additional configurations for establishing connection with DB instances such as subnet group, security group, whether or not the instance is publicly accessible, AZ and database port.

In addition to above configurations, you can also configure settings such as backups, monitoring, maintenance windows, and more. In addition to supporting open source and licensed database engines, Amazon provides its own proprietary database engine, referred to as Amazon Aurora which is compatible with MySQL and PostgreSQL.

## **Amazon Aurora**

**Amazon Aurora** is up to five times faster than MySQL and up to three times faster than PostgreSQL running on the same hardware and it scales automatically as per application needs. Aurora creates six copies of your data distributed across 3 AZs and continuously backs it up to S3. Aurora also provides capability to replicate data across multiple regions for faster global access. Additionally, we can use Aurora Serverless for unpredictable workloads or for development purposes when we don't have a requirement to keep DB instances running all the time.

Here are few key considerations about Aurora –

- There is no requirement to configure storage requirements like RDS—it is handled internally by Aurora.
- Aurora Serverless operates based on minimum and maximum Aurora Capacity Units and not on instance types like RDS.
- RDS works at the VPC level. Classic RDS can be opened to the public internet or restricted to allow specific IPs to access it from outside. With Aurora Serverless, you can't provide public access but you can use the **Data API** or leverage an EC2 machine in the same VPC as Aurora to facilitate the connectivity.

- For data security or meeting compliance requirements for data encryption at rest, Amazon **RDS** and **Aurora** both provide functionality to encrypt DB instances.

Amazon RDS is a beneficial service when your business requirements require support for relational structured data with complete ACID properties compliance. Many modern applications don't have a requirement for strong ACID compliance, full relational support or fixed schema. Amazon NoSQL databases are viable solutions to address these kinds of use-cases while also providing scalability, high performance, and no operational overhead. We described multiple types of NoSQL databases in Chapter 3 and in this chapter, we will cover the equivalent AWS offered services to different types of NoSQL databases, starting with Amazon DynamoDB as a key-value database solution.

## **Amazon DynamoDB**

AWS provides Amazon DynamoDB (DDB) as a key-value database offering which is designed to provide single-digit millisecond latency for any workload scale. DDB stores data in tables and follows schemaless design. We discussed the primary key, partition key and sort key in Chapter 3 in context of using key value data stores to uniquely identify items. In DDB architecture, the primary key is represented by a partition key or a combination of partition key and sort key. Any queries on data can be performed via AWS Command Line Interface(CLI), AWS Console or AWS Software Development Kit(SDK) in your preferred language. DDB is based on the leader-follower nodes model and internally stores the data on storage nodes referred to as partitions.

Below are the keys responsible for data storage on DDB:

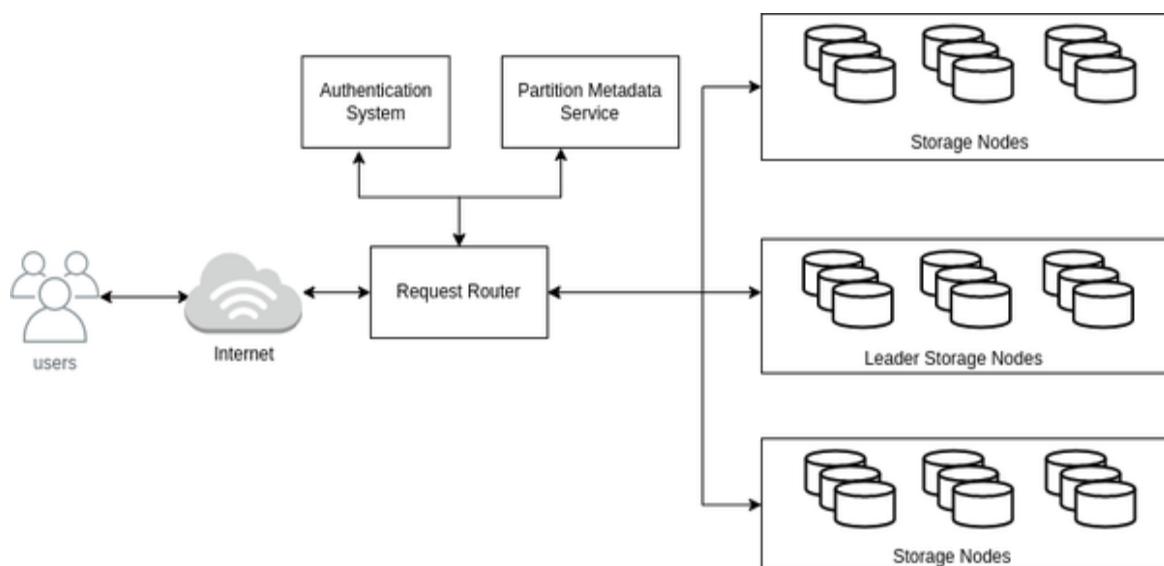
### *Partition Key*

Partition Key is mandatorily specified at time of table creation and it is used as an input to internal hash function to determine a

physical partition where data should be stored. A Request Router (RR) component is responsible for routing the request to a specific partition as shown in [Figure 7-1](#). RR is a stateless service which consults with Partition Metadata Service for determining the partition and then the request is forwarded to a specific partition. The PUT request is sent to the leader node while GET request can be served via the leader or follower node depending on the required consistency support.

### Sort Key

Sort Key is an optional attribute on table creation. If it is specified, then the primary key is a combination of partition key and sort key, otherwise it's just the partition key. The combination of partition key and sort key should be unique for data storage. All items with the same partition key are stored together on the same physical partition in a sorted order by sort key value. We recommend creating a DDB table with both partition key and sort key as per the read/write query patterns and it will serve the user requests without need of Global Secondary Indexes(GSIs).



*Figure 7-1. DynamoDB internal architecture*

DDB is a fully managed service and we don't have to configure any servers or storage space for using it, instead we can select from different **capacity modes** for allowed reads and writes per specific tables. DDB defines reads and writes as Read Capacity Units(RCU) and Write Capacity Units(WCU) respectively and **pricing** is based on consumed RCU and WCU, along with required consistency or transaction support. Let's understand available DDB capacity modes:

### *Provisioned*

Provisioned means customers can configure specific RCU and WCU values for tables that are to be supported and DDB internally manages the resources to support this scale. This is the preferred mode for applications with predictable traffic, when you can easily forecast RCU and WCU requirements.

### *AutoScaling*

For varying workloads with undeterminable fixed Provisioned capacity, we can use AutoScaling mode with lower and upper RCU and WCU limits and DDB automatically scales up or down based on the traffic.

### *On-demand*

DDB On-demand is a serverless kind of support where we don't need to configure RCU and WCU values. DDB manages the scale in the backend and we as customers don't have to worry about it. This is preferred mode for use-cases such as tables with unknown workloads or unpredictable application traffic.

The DDB table might be responsible for serving multiple use-cases and not all of these use-cases will be performant just with primary key design. DDB provides **index support** to make queries faster and this can be created on top of the table as necessary. We can think of a DDB index as a child table that is created by selecting some of the attributes from the main table. DDB supports two types of

indexes, Local Secondary Index(LSI) and Global Secondary Index(GSI) as described in Table 10-1.

*T  
a  
b  
l  
e  
7  
-  
1  
.D  
if  
f  
e  
r  
e  
n  
c  
e  
s  
i  
n  
L  
S  
I  
a  
n  
d  
G  
S  
I*

Parameter	LSI	GSI
Lifecycle	The LSI lifecycle is the same as that of the table. It can only be created/deleted along with table creation/deletion.	The GSI lifecycle is independent of table and can be created/deleted as required.
Primary Key Schema	The partition key is the same as the base table with a different sort key.	The partition key and sort key can be different from the base table.
Querying Capability	Scoped to the partition as of the base table, hence the term “local” with indexed data size limitation of 10 GB.	Queries can span across the partitions on base table data, hence the term “global” with no size limitations.
Provisioned Capacity	LSI shared read/write throughput capacity with the base table.	GSI has independent settings for read/write throughput from the base table.

Read Consistency	Offers both strong and eventual read consistency.	Offers only eventual read consistency.
------------------	---	--

As we move from traditional relational databases to a wide variety of modern NoSQL databases, one key difference is transaction support. Transaction is inherently available in SQL like databases but not supported in all of the NoSQL databases. DDB provides transaction support and we can choose the specific **read and write APIs** if our use-case has a requirement for transactions. Another difference is strong consistency support—DDB provides both eventual and strong consistency support (though GSI only supports eventual consistency) and we can specify the required **support type** as we access the data.

We discussed identification of physical partitions based on the partition key and different capacity modes which helps us to configure read and write throughput for DDB tables. The throughput values apply on a table as a whole and are equally divided among the partitions being created. The partitions have hard limits for allowed 3000 RCU and 1000 WCU and requests are throttled once this limit is breached. To avoid this request throttling, also referred to as hot key, we should carefully design the schema understanding the query patterns as well as keeping DDB architecture in mind. Let's move to a document database offering called Amazon DocumentDB.

## Amazon DocumentDB

Amazon DocumentDB is a document database offered by AWS which is designed to store and query data as JSON-like documents. A **document** is structured as a collection of nested key-value pairs and can be useful in scenarios such as storing food menu information, like for Cafe Delhi Heights's online store. We've added a sample

document for a customer order below. We can add or remove attributes to the food menu items with ease without the worry for fixed schema like relational databases. Some other business use-cases include content **management**, real-time big data, and maintaining user profiles.

```
{  
  "Name": "Mandeep Singh",  
  "orderId": "1234-1234-4567",  
  "FoodItems": [  
    {  
      "itemName": "Biryani",  
      "qty": 1  
    },  
    {  
      "itemName": "Nuggets",  
      "qty": 2  
    }  
  ]  
}
```

For reference, Table 10-2 spells out how the terminology associated with DocumentDB compares to that of relational databases.

*T  
a  
bl  
e  
7  
-  
2  
.S  
Q  
L  
v  
s  
D  
o  
c  
u  
m  
e  
n  
t  
D  
B  
t  
e  
r  
m  
in  
ol  
o*

*g  
y*

SQL                    MongoDB/DocumentDB

---

Table                  Collection

---

Row                    Document

---

Column                Field

---

Primary Key      Object Id

---

Amazon DocumentDB is **MongoDB** compatible with support for powerful ad-hoc queries and comes with **transaction** support similar to DDB. There are also few functional benefits of DocumentDB over MongoDB, such as transaction support for all CRUD statements including operations on multiple documents. For all functional differences between MongoDB and DocumentDB, please refer to the **AWS documentation**. Unlike with DDB, we need to specify the **instance class** and number of instances for DocumentDB cluster setup. For production systems, we recommend choosing at least three instances for higher availability. The final step will be to set up

username and password for authentication and establish connection with DocumentDB cluster.

AWS also provides **migration** tools such as AWS Database Migration Service (DMS) to facilitate easier migration of MongoDB workloads from on-premise or EC2 servers to DocumentDB. The storage business use cases can vary and for use cases with requirements for graph databases, we can leverage Amazon Neptune.

## Amazon Neptune

Going back to our Cafe Delhi Heights example, one of the requirements for launching an online food restaurant is creating food communities among the people, allowing for tracking things such as people's likes and dislikes of food items. Consider a scenario where you want to figure out a common food item among your community that is liked by everyone.

These kinds of use-cases with highly connected data can be stored and queried at scale with AWS fully managed graph database service, **Amazon Neptune**. Neptune is optimized to store and map billions of relationships and enable real time navigation of connections with millisecond query response time via **Apache TinkerPop Gremlin** or **OpenCypher** for property graph databases and **SPARQL** for Resource Description Framework (RDF) format graph databases. AWS recommends consulting the GitHub repository "**AWS Reference Architectures for Using Graph Databases**" to look into graph data models & use-cases, query languages and sample examples of deployment architectures.

Another example use-case for Neptune is to get a clear picture of our cloud infrastructure around how different services or entities are connected to each other. For example, you may want to know which entities use a particular IAM role as shown in **Figure 7-2** or figuring out over-permissive IAM policies with "\*" marked on resource

permissions. For example—We might not want all the users in the organization to modify network security groups.

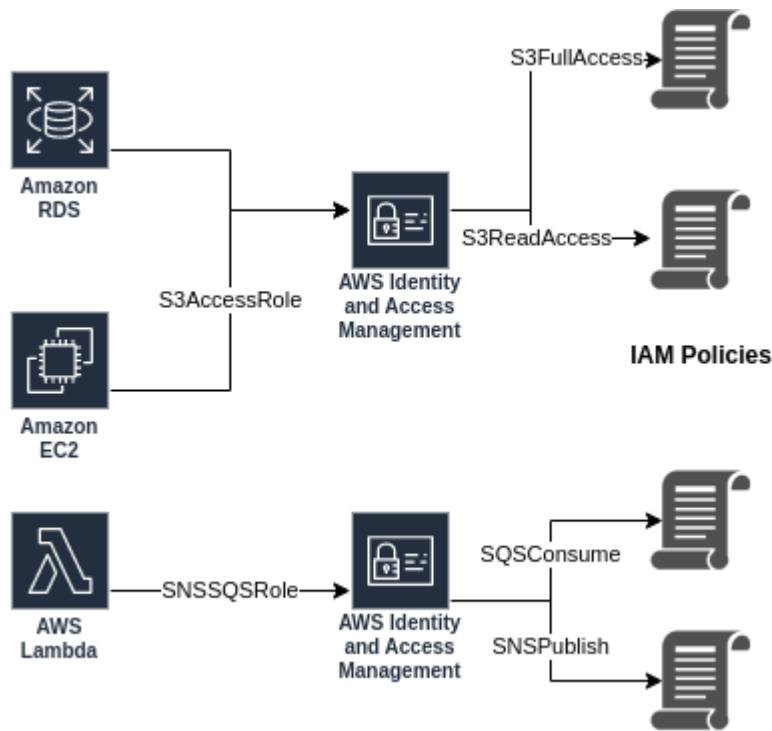


Figure 7-2. Graph representation of IAM role-entities association

For a step-by-step guide to cluster setup, you can refer to the [AWS documentation](#). Here's we'll discuss some points that you should be aware of during your graph database cluster setup:

### Compute

The Neptune cluster is launched with a specific instance type that should be selected at cluster creation time. The cluster can have a maximum of 1 writer and up to 15 read replicas, and note that writer instances scale vertically whereas read instances can scale vertically or horizontally. We recommend configuring at least 1 read replica for higher application availability and better read performance. If a writer instance goes down, the read replica is promoted to writer and the writer is restarted as a read replica. In case of only a single writer instance in a cluster, the cluster might be down for a few minutes as it restarts and comes back online.

Neptune also offers serverless configuration which can be useful for variable workloads. We can specify an upper capacity limit and it is only used if needed.

### *Storage*

Neptune stores data six times across three AZs. Neptune storage layer is independent of compute layers and scales independently and automatically as necessary. It starts with 10 GB and grows up to 128 TB as data increases.

### *Traffic Distribution*

Neptune provides separate writer and reader endpoints. Writer endpoint (referred to as cluster endpoint) points to the primary instance and should be used for write operations and optionally read queries. Reader endpoint distributes the requests across the read replicas in round-robin fashion and should be used for all read queries. Neptune additionally provides an instance endpoint (an endpoint that connects to a specific DB instance) and a custom endpoint that can be used to represent a set of DB instances. Neptune doesn't offer load balancing functionality on instances. For any such requirement, it should be built in application code.

### *Caching*

Neptune offers three types of caching support to improve performance.

- Buffer Cache – Buffer cache is in-memory cache to improve query performance and Neptune allocates  $\frac{2}{3}$  of memory of instance to cache.
- **Lookup Cache** – Lookup cache is an instance level cache and is helpful in improving read performance for queries with repetitive lookup of property values or RDF literals. It is

enabled by default for R5d instances and uses the instance's NVMe-based SSD to store these values for quick access.

- **Query Results Cache** – We can cache Gremlin read-only query results to get faster responses on query re-run. For clearing the cache, we can specify time-to-live(TTL) for queries, clear cache at query level or clear entire cache.

We discussed in-memory databases and their benefits in Chapter 4. We'll discuss Amazon ElasticCache as a managed offering by AWS for in-memory data stores.

## Amazon ElasticCache

In-memory databases are extensively used as caching solutions to improve system performance. Amazon ElasticCache is a managed service offered by AWS as a distributed cache environment and works with both Redis and Memcached engines, both covered in detail in Chapter 4. Both Redis and Memcached are in-memory key-value caching solutions but there are **associated differences** and it is important to figure out which caching engine meets your business requirements. You can look at Memcached as a simple key-value caching solution to offload reads from primary data source with multi-threading support whereas Redis comes with functionalities such as persistence for long-lived data, advanced data types such as lists and sets, sorting and ranking datasets in memory, geo-spatial support, message broker, etc.

Here are a few key considerations around each of the caching engines available as part of Amazon ElasticCache. For a step-by-step guide for cluster creation, please follow the AWS page for **Redis** and **Memcached**.

### *Subnet*

As noted in Chapter 8, the number of IP addresses available for use are based on subnet CIDR. As a first step in cluster creation,

we should create a subnet associated with AWS VPC with support available for the required number of IP addresses so as they are assigned to nodes in the cluster.

### *Data Tiering*

Data can be tiered between memory and NVMe SSD storage for optimizing cost and is ideal for applications that access up to 20% data regularly. This support is available with ElasticCache for Redis for the R6gd instance family and infrequent data is moved to SSD asynchronously once memory (DRAM) is fully utilized, as shown in [Figure 7-3](#). The storage can scale up to 1 PB in a single cluster.

### *Auto Scaling*

Redis and Memcached both can be scaled horizontally and vertically. Redis works on the concept of primary node and read replicas and can be launched with two kinds of configurations – cluster mode disabled and cluster mode enabled.

- For cluster mode disabled, there can be a maximum of one primary node and up to five replica nodes. This configuration scales horizontally for reads and vertically for writes.
- For cluster mode enabled, there can be a maximum of 500 nodes and up to 5 replicas associated with each node.

### *Redundancy & Replication*

ElasticCache for Redis has [multi-AZ deployment](#) support similar to Amazon RDS. Whenever a primary node is down, read replica (1-5 read replicas placed in different AZs) is promoted to primary and it ensures high availability even if the entire AZ is down. The replication of data on read replicas also ensures separation of read and write workload.

### *Multi-Threading*

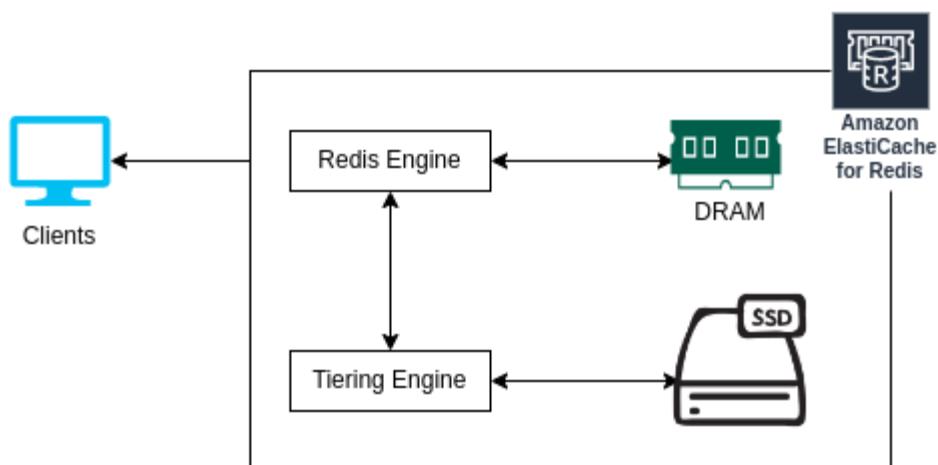
Redis is a single-threaded process when incoming requests are handled sequentially, while Memcached is multi-threaded, meaning it makes good use of larger Amazon EC2 instance sizes with multiple cores.

### *Persistence*

Redis can be used as a standalone database for application as it supports persistence for long lived data, while Memcached is a pure caching solution that should be used in front of any database such as RDS to improve read performance.

### *Encryption and Compliance*

Redis supports Payment Card Industry Data Security Standard([PCI DSS](#)), Health Insurance Portability and Accountability Act([HIPAA](#)) and Federal Risk and Authorization Management Program([FedRAMP](#)) and encryption capabilities. Memcached doesn't have strong support for authentication and encryption, so the recommendation is to launch Memcached nodes in private subnets with no public connectivity to ensure higher security.



*Figure 7-3. Redis Data Tiering Support*

As stated earlier, we can use Memcached or Redis to offload our database read queries such as RDS or DDB. DDB offers a custom in-

memory cache offering as well called Amazon **DynamoDB Accelerator** (DAX) to improve read performance.

## Amazon DynamoDB Accelerator

DAX is a cluster of primary node and read replicas which run inside VPC as shown in **Figure 7-4**. For accessing DAX, a DAX client is installed alongside an application on the server and it directs the application's DDB API requests to the DAX cluster.

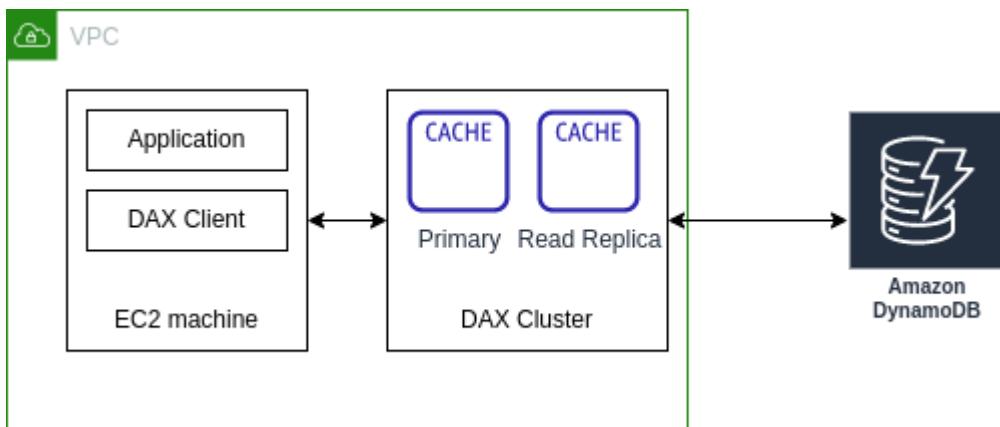


Figure 7-4. DAX Cluster setup

The below list describes how read and write APIs execute in the presence of both DAX cluster and DDB.

### *Read APIs*

The eventually consistent read API calls (GetItem, BatchGetItem, Query and Scan) are served from DAX. In the scenario of cache miss, the request is passed to DDB to retrieve the response. As DDB returns the result, it also writes the result to the DAX cache on the primary node.

### *Write APIs*

For the write API calls (BatchWriteItem, UpdateItem, DeleteItem and PutItem), data is first written to the DDB table and then to the DAX cluster. The API returns success only if write is successful on both DDB and DAX.

You may want to display search results to users based on the user's search query—in the example of Cafe Delhi Heights, these search results could be food name or restaurant location or any supported filters. A search with a food name should reflect all the restaurant locations serving the food item, and search with the restaurant location should display all dishes served by that location. How can such data be modeled in a database for faster retrieval? For these kinds of business use-cases, AWS offers a search database, referred to as Amazon OpenSearch.

## Amazon OpenSearch

**OpenSearch** is an open-source project derived from Elastic Search (ES), unlike most of the AWS services which run on proprietary software. OpenSearch is popular for use-cases such as full text search, logs & analytics, ingestion pipelines, and machine learning. On top of the search engine, AWS provides OpenSearch dashboards which are helpful in visualizing the data. Here are few key considerations about OpenSearch architecture and how we as customers can make best out of it:

### *Cluster Setup*

OpenSearch can work as a single-node or multi-node cluster. As this is an open-source project, you can also set up your own cluster following the steps [here](#). OpenSearch is a managed service which helps you to avoid the operational overhead of set up and works on instance based (master and data node(s)) or Serverless mode. AWS recommends setting up an odd number of master nodes for a production OpenSearch cluster with a minimum of three nodes. The type of master node can be decided based on the **number of data nodes** in our domain such as setting up m5.large.search or m6g.large.search for 1-10 data nodes count.

You can skip setting up dedicated master node(s) for development use-cases(one of the data nodes hold master node responsibilities) to save cost, though it is recommended to configure them for production scenarios for better cluster stability and dedicated nodes for cluster management tasks.

### *Storage*

We can select either Amazon EBS or instance store volumes to be associated with the instances—the volume types available will depend on the chosen instance type.

### *Cluster Access*

For cluster access and networking, we can choose either VPC access or Public access. VPC access allows you to operate the cluster inside your VPC and additionally allows you to configure subnets, security groups and IAM role.

### *Deployment*

OpenSearch allows configuration updates such as an increase in EBS storage space or upgrading instance type on a live-running cluster. The configurations are deployed to new cluster via **blue/green deployment** to avoid any impact. We recommend performing any such updates during off-peak hours.

### *Dashboard Authentication*

OpenSearch provides the option to configure authentication for dashboards via **SAML authentication** or **Amazon Cognito authentication**.

In the next section, we'll explore the Amazon Timestream database useful for time-series data use cases.

## **Amazon Timestream**

Amazon Timestream is a fully managed serverless database service offered for use-cases that require time-series data operations, such as DevOps analysis data. Amazon Timestream automatically scales to handle trillions of events per day with 1/10 cost and is up to 1000 times faster when compared to relational databases.

Here are few points of consideration for Amazon Timestream database:

- As TimeSeries DB is serverless, it is easy to use with no requirement for analyzing number or type of instances.
- TimeSeries DB provides built-in analytics support with interpolation and smoothing functions for identifying trends, patterns and anomalies using standard SQL.
- Data is encrypted by default with standard key or customer owned KMS key.
- TimeSeries provides support for data retention policies based on configured time and moving data to different storage tier as per policy. It supports two storage tiers – In-memory tier (preferred for latency sensitive queries) and Magnetic disk tier (preferred for analytical queries).
- Records require timestamp as a mandatory dimension and data stored to DB cannot be deleted or updated. Removal of data should be handled via configuration of retention policies.

We discussed the idea of wide-column databases in Chapter 3 and then discussed an open-source variant referred to as Apache Cassandra. Let's discuss AWS managed solution, Amazon Keyspaces to meet the similar requirements.

## Amazon Keyspaces

Amazon Keyspaces is a highly scalable, available, and managed wide-column database service (which we covered in Chapter 3) with compatibility with open-source Apache **Cassandra** database. Amazon Keyspaces is offered as a serverless service, meaning we don't have to worry about resource provisioning and can simply focus on building our applications. A few use-cases where Amazon Keyspaces can be considered are:

- Migration of on-premise Cassandra clusters or clusters running on EC2 instances to fully managed AWS service. This helps in reducing large cluster management overhead such as managing deployment, figuring out best configurations for your workload such as JVM tuning for garbage collection, understanding Cassandra internals, provisioning capacity for expected workloads, newer version upgrades, patching & maintaining cluster infrastructure, etc.
- You have a business requirement to be compatible with open source Cassandra.
- Amazon Keyspaces has better integration with AWS services such as Cloudwatch for monitoring purposes or IAM for authentication as compared to customers managing on their own.

Now, since Amazon Keyspaces abstracts out infrastructure management, we don't have access to low level cluster management control plane APIs. Amazon Keyspaces offers two types of capacity modes for table's read and write workloads similar to DDB: provisioned capacity with auto-scaling and on-demand. All the data in Keyspaces tables is by default encrypted and replicated in multiple AZs for high availability and durability. Please refer to the **AWS guide** for functional differences between Cassandra and Amazon Keyspaces.

## Conclusion

We discussed a wide range of storage services offered by AWS that can be used to fulfill our business needs. The Cloud Storage section focused on primitive storage options such as block storage, file storage, and object storage, while the databases section covered AWS services offered for relational and non-relational databases. It is quite important to understand the business use-case for choosing the best cost optimal storage service. To that end, let's take another look at our Cafe Delhi Heights storage requirements. Table 10-2 shows which AWS storage services can be used to fulfill these requirements.

T  
a  
b  
l  
e  
7  
-  
3  
.S  
t  
o  
r  
a  
g  
e  
S  
e  
r  
v  
i  
c  
e  
i  
d  
e  
n  
ti  
fi  
c  
a  
ti  
o

*n  
f  
o  
r  
C  
a  
f  
e  
D  
e  
l  
h  
i  
H  
e  
i  
g  
h  
t  
s*

---

Requirement      Storage Service

---

Store customer's profile and access details,  
such as username and password      RDS, Aurora,  
ElasticCache

---

Store food information, such as  
available menu items      DDB, DocumentDB,  
Keyspaces, OpenSearch

---

Store different kinds of media such as images of restaurant food items, customer uploaded reviews, etc.

S3

Big Data Analytics to improve the customer experience, such as separation of good and bad reviews.

S3, EFS, DDB

Create food communities and allow people to interact in social circles.

Neptune

Search for food items based on multiple identifiers such as food name, restaurant location, ratings, etc.

OpenSearch

Application logs and metrics archival after ninety days and persist for one year.

S3,  
OpenSearch

### NOTE

Please refer Chapter 2 and 3 for general guidelines and a flowchart around choosing a storage solution based on your business use-case.

For handling the client requests or processing on data stored on AWS storage service, we require servers or compute platforms. In the next chapter, we'll discuss and deep dive into different kinds of compute platforms offered by AWS and how we can identify the best option for our business use-case.

# Chapter 8. AWS Compute Services

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

You love programming and spend your free time designing and creating new software applications on your personal computer. You recently created an application that converts input images to cartoon images and demoed this to your friend who happens to be a software engineer at a big tech company. The friend really liked the idea and suggested sharing the application with other people—plus it can also help you to earn some extra dollars. But how can you achieve this task? How can you make a program running on a personal computer accessible to other people?

For this specific example, you need some mechanism that allows people to send an image to a personal computer and the personal computer returns output as a cartoon image via the internet. Here, in this scenario, the personal computer is acting as a server. A server

is nothing but a computer that is running most of the time (i.e. a highly available application) responsible for serving any kind of request. This server can be accessible just to you or any other computer or other users who all are referred to as clients.

Servers are an essential part of any application and AWS offers multiple compute platforms to run applications. A compute platform in AWS Cloud is a virtual server hosted in AWS data center which can be accessed over the internet. We'll start off our discussion with a compute platform similar to our personal computer, called the Amazon Elastic Compute Cloud (EC2) machine and then move off towards AWS Lambda and containerization services present in AWS Cloud.

## Amazon Elastic Compute Cloud

Amazon EC2 is a scalable virtual server hosting service provided by AWS. We can leverage **Amazon EC2** service to create virtual machines in AWS Cloud with a required set of configurations such as CPU, memory, storage, networking, etc. These virtual machines or virtual computing environments are referred to as instances. AWS provides the option to select bare metal servers or go with a virtualized environment maintained via hypervisor, which should be determined by identifying your business and compliance requirements. Here is a general description about hypervisor and bare metal servers:

### *Hypervisor*

A virtualization component helpful in running multiple virtual machines on a single physical server as shown in **Figure 8-1**. Hypervisor helps in allocating computing resources such as CPU and memory to virtual machines running their own operating system and applications. AWS supports two hypervisors based on available instance types – **Xen** and **Nitro**. **Nitro** system is a new generation hypervisor with enhanced security and performance.

## Bare Metal Server

Bare metal servers are physical servers dedicated to a single tenant. They're helpful for workloads which require direct access to Intel Xeon processor infrastructure such as [Intel VT-x](#) or a strict compliance requirement or a need to run a custom hypervisor of your own. For example, run [Oracle hypervisor](#) instead of Nitro..

### NOTE

One Amazon EC2 instance doesn't necessarily map to one physical server in AWS Cloud.

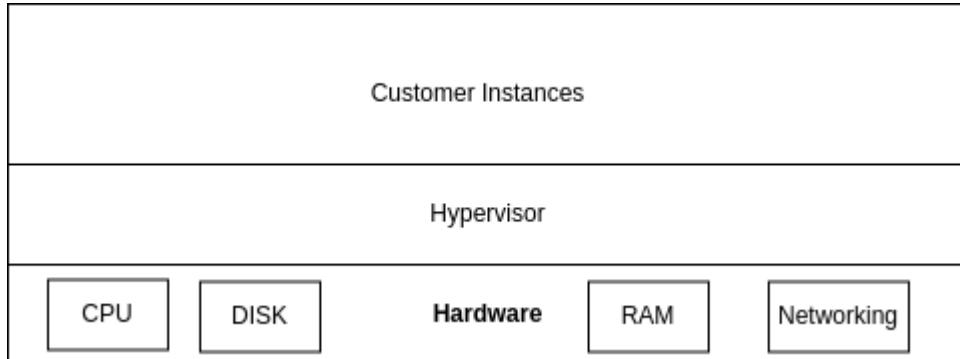


Figure 8-1. Xen Hypervisor

Launching and running instances in AWS Cloud is easy and removes the headache of maintaining physical hardware. Amazon EC2 instances can be launched in just a few clicks on the AWS console following the steps on the [AWS page](#). The instances are launched with a specific set of configurations determined by Amazon Machine Image (AMI).

## Amazon Machine Image

We launch an Amazon EC2 instance with the different configurations, such as: the Operating System the instance is launched with;

storage volumes attached to an instance; install dependencies before instance boot up, custom security configurations, etc. These configurations can be defined via a pre-configured template, referred to as Amazon Machine Image (AMI), which enables easy and efficient provisioning of EC2 instances with desired software stack. Consider a scenario where you as an engineer have responsibility to install security dependencies to all the EC2 instances launched in organization. One way is to run scripts on these EC2 instances after they are launched but a more optimal way could be to create a base AMI with all required dependencies installed. This essentially saves a lot of time and ensures 100% confidence that none of EC2 instances is missed for required dependency.

We, as customers, can launch EC2 instances using freely available AMIs from the AWS community, purchase AMIs from third parties such as Red Hat, or create our own custom AMIs depending on our needs. For example, Amazon Linux and Amazon Linux 2 AMIs are available free of cost and are fully supported and maintained by AWS. AWS provides a marketplace, where users can choose from a wide variety of publicly available AMIs. Here are few considerations for choosing the best AMI for your business use-case:

### *Region*

AMIs are created specific to the region. If an AMI is not available in a specific region, you can **copy the AMI** to the required region and further use it to launch EC2 instances.

### *Operating System & Architecture*

EC2 instance runs on Operating System and you can choose from multiple Operating System flavors(32-bit or 64-bit architectures) **supported by AWS**.

### *Launch Permissions*

An AMI can be owned by AWS, businesses or individuals. As per the ownership, an AMI can be available to the general public, a set of users or just a single individual.

### *Root Device Storage*

The AMIs are either backed by Amazon EBS or by Instance Store (refer Chapter 9–Storage Services for more details) as root device storage. The root device storage contains the image used to boot the EC2 instance. There are significant **differences** in both of these storage choices, such as size limit (64 TB for io2 EBS Block Express and 10 GB for instance store), data persistence, boot time (usually less than 1 minute for EBS and less than 5 minute for Instance Store), etc.

#### **NOTE**

You can check complete details on Amazon EC2 instance root device storage [here](#).

As we finalize AMI for an EC2 instance, the next decision we need to make is the instance type. AWS offers a wide range of instance types based on different capabilities and we can choose which best suits our requirements.

## **Instance Type**

The hardware requirements can vary from customer to customer and with this in mind, EC2 service provides different instance types. Instance types in AWS EC2 define the hardware and performance characteristics of an EC2 instance. We might not always have a concrete answer to which instance type is most suitable and in these scenarios, we can start by installing our application on any general purpose instance and then perform load testing. This direction will help determine the limits where this specific instance breaks and it

can eventually help in figuring out instance type and number of instances needed for the workload.

Different instance types offer varying combinations of CPU, memory, storage, and networking capacities. Here are different instance type families available in AWS Cloud:

#### *General Purpose Instances*

General purpose instances are suitable for most of the workloads in the industry and they keep a fine balance between compute, memory and network resources.

#### *Compute Optimized Instances*

Compute Optimized Instances are suitable for CPU intensive workloads such as high performance computing, big data analytics, etc.

#### *Memory Optimized*

Memory Optimized Instances are suitable for workloads with requirements for large data sets processing in RAM such as in-memory caches.

#### *Storage Optimized*

Storage Optimized Instances are suitable for workloads with requirements for high, sequential read and write access to large data sets on local storage such as databases.

#### *Accelerated Computing Instances*

Accelerated Computing Instances are GPU-based instances and suitable for workloads with requirements to high performance computing, computational finance, machine learning, etc.

Choosing the appropriate instance type is crucial for achieving optimal performance and cost-efficiency for specific workloads. For

updated guidance on instance types and instance families, refer to the [AWS page](#) on Amazon EC2 instance types. There are some additional key considerations as we launch an EC2 instance:

- To establish connection with an instance after it has launched, you should select a key pair to enable secure shell(SSH) access to the instance.
- EC2 instances are launched at the AZ level within a specific region. EC2 allows you to modify default network configurations, such as the VPC instance it should be launched in, subnet in VPC, security group, etc. These configurations help in determining how the connection to launched instances will be established.
- We can choose to assign public IP to an instance. Public IP assignment helps in connecting to an instance directly from the public internet given that we've appropriate permissions either via security groups or NACLs.

### NOTE

For production environments, we recommend using AWS CloudFormation or any other tool to launch infrastructure via code instead of AWS Console. This will help in better maintainability and replicating the infrastructure in any other required region or AWS account.

For running the application at scale, we need a good balance of vertical scaling and horizontal scaling for a number of instances in the entire infrastructure. By looking at usage metrics, we can figure out the minimum number of instances that will be required throughout a longer period of time. For such scenarios, cost can be reduced by using [Reserved](#) EC2 instance capacity to get up to 72% discount compared to On-Demand instance pricing based on the usage commitment— one-year or three-year commitment. Note that

On-Demand instances are instances that we launch as needed from AWS Console, while we pre-commit to use EC2 instances for a period of time in case of Reserved instances.

In contrast to Reserved Instances, there may be use-cases where we're not much worried about losing an instance while the workload is running or the workload can be resumed later on once instances are available in the EC2 pool. For such scenarios, we can leverage something called Spot **instances**. Spot instances are less expensive than On-Demand instances. Let's take an example, assume us-east-1a AZ has a total capacity of 100 m4.large instances. Out of these 100, half of them are already in use by customers. Further, based on AWS historic analysis, 30 instances will be requested by customers shortly as dedicated capacity. In this EC2 pool of 100 instances, there are 20 such instances left that no-one is getting benefit from—neither AWS nor customers. So these instances are made available to customers as spot instances at a lesser cost. For scenarios such as AWS customers requiring more than the 30 as predicted earlier, any of the spot instances can be turned off and assigned to requesting customers.

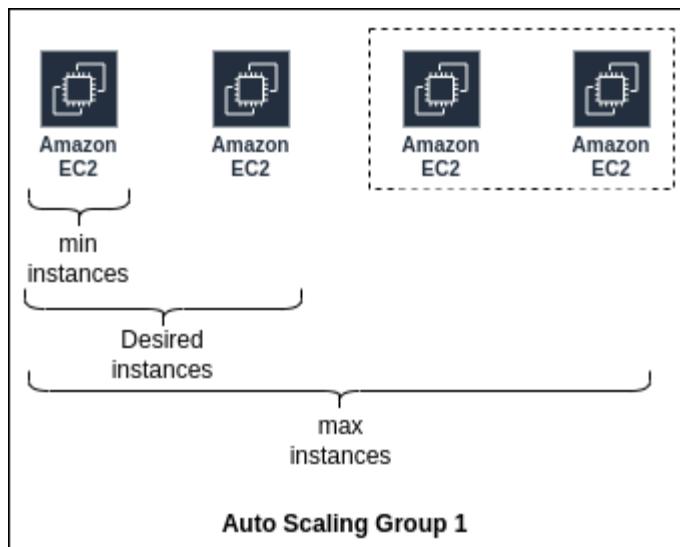
Traffic on applications running on top of EC2 instances can vary across the day and we might not always require a fixed number of running EC2 instances. AWS offers an Auto Scaling feature which helps in automatically adding and removing instances for the application. For example, the application requires 10 instances at noon but only two at midnight. Let's dive into that next.

## **Auto Scaling**

Auto scaling is an AWS feature that automatically adjusts the number of EC2 instances in the collection in response to changing workload demands. We can create a collection of EC2 instances based on any factor, such as AZ, instance type, etc., and these collections are referred to as Auto Scaling groups(ASG) when auto scaling is enabled on them. In these groups, we can specify

minimum and maximum number of instances and apply an auto scaling policy for automatically adding and removing the instances within this limit. It helps maintain application performance and availability by dynamically scaling the capacity up or down based on predefined scaling policies. Autoscaling can be configured to scale based on metrics like CPU utilization, network traffic, or custom metrics defined by the user. It ensures that the application scales seamlessly to handle increased traffic and reduces costs during periods of low demand. For example, refer to **Figure 8-2** for how a new instance is added as cumulative CPU Utilization limit is breached for ASG:

- For Auto Scaling Group 1, the minimum number of instances configured is one and the maximum is four.
- Assume that the auto scaling policy states to add a new instance to the group if maximum CPU Utilization exceeds 75% for continuous 5 minutes.
- Now as the limit is breached, auto scaling kicks in with the desired number of instances as two from one and it launches a new EC2 instance.



*Figure 8-2. Auto Scaling*

Large applications in AWS Cloud might be running thousands of EC2 machines to serve the customer traffic or internal system processing and failures are bound to happen. The EC2 instances might go down due to unforeseen hardware failures and we as customers should have proper mechanisms in place to overcome these failures. There are multiple ways to handle these issues such as:

- Auto Scaling policies to always maintain the desired number of instances within a group.
- EC2 instances launched in multiple AZs to withstand AZ failure.
- EC2 Auto Recovery is one mechanism to relaunch the instance in case of failure. The instance is relaunched with the same configurations, such as instance Id, metadata, attached EBS volumes, private IP, and elastic IP addresses.

EC2 is the oldest and most widely used service for running applications in AWS Cloud. EC2 provides control to maintain the software stack and maximum visibility of hardware operations. However, it might not always be a requirement to know how application software is deployed or you might want to avoid any operational maintenance of systems such as OS patching. For these scenarios, AWS offers serverless services where customers don't need to worry about instance provisioning or capacity planning. Serverless, as the name dictates, abstracts out server details and directly allows us to run our workloads. AWS Lambda operates on a serverless model allowing code execution with required scale. Let's dig deeper into AWS Lambda next.

## AWS Lambda

AWS Lambda is a fully managed serverless compute service that lets users run code without overhead of server maintenance. The only thing we need in order to run our applications on AWS Lambda is

code in our preferred language, referred to as Lambda function. It supports multiple programming languages and enables rapid development and deployment of event-driven applications. The backend architecture of Lambda is provisioned as per function runtime and executed on customer invocation events. Lambda automatically scales the underlying infrastructure to match the workload, providing high availability and reliability. Here are the main concepts in relation to the AWS Lambda service that you'll need to understand:

### *Function*

Function is an application's codebase in your preferred programming language supported by AWS Lambda and is invoked with the help of a trigger to execute specific logic.

### *Trigger*

Trigger is any resource or configuration which holds the responsibility of invoking Lambda function—for example, **invocation** via AWS console or via any AWS services such as **Simple Queue Service(SQS)**, **DDB streams**, **Kinesis**, etc.

### *Event*

Event is a custom JSON-formatted document or any other structure specific to AWS service, such as SNS notification, which is used to pass on information to Lambda function for processing.

### *Lambda Execution Environment*

Lambda backend infrastructure internally creates a secure and isolated runtime environment for function execution. This runtime environment is referred to as the execution environment and is set up with help of user inputs such as function language runtime, available memory and maximum execution time (maximum supported configuration is 15 minutes) for function. The lambda execution shuts down as the execution time elapses.

The execution environment is launched with specific instruction set architecture and Lambda offers two type of architectures to determine computer processor type:

- arm64 with 64-bit ARM architecture for AWS **Graviton2** processor.
- x86\_64 with 64-bit x86 architecture for x86-based processors. We recommend using arm64 architecture for cost and performance efficiency over x86.

### *Deployment Package*

The Lambda function code can be deployed via deployment package in below two ways:

- A .zip file with code and its dependencies stored in Amazon S3 with maximum allowed size of 250MB. Lambda holds responsibility for providing operating system and runtime for function.
- A container image stored in **Amazon Elastic Container Registry** (ECR) with code and its dependencies with maximum allowed size of 10GB. The image should be compatible with **Open Container Initiative** and should have the operating system and runtime included in itself.

### *Layer*

Common code such as utilities or libraries can be zipped as layers instead of adding them to each Lambda function. Layers can be attached to Lambda and help in code sharing, reducing overall Lambda function code bundle size and the startup time. Layers additionally support versioning to maintain multiple versions of code libraries and ensuring backward compatibility.

### *Destination*

Destination is another AWS service that is configured to receive invocation records on completion (success/failure) of asynchronous Lambda execution. AWS supports standard SQS queue, standard SNS topic, Lambda function and EventBridge event bus as destination.

### NOTE

AWS continually adds new language runtimes—check for updated guidance on supported language runtimes [here](#).

The way Lambda function should be invoked for processing depends on your use-case—some options include an **object addition to S3 bucket** or scheduled weekly via **Cloudwatch events**. You may choose a specific trigger for lambda invocation depending on your use-case. Lambda supports three types of **invocation modes**—synchronous, asynchronous or poll-based invocation:

#### *Synchronous Invocation*

Synchronous invocation can be achieved in multiple ways—via API Gateway, Application Load Balancer, AWS CLI, etc. In this invocation mode, the customers wait for a response until Lambda completes its execution. It is preferred for latency sensitive workloads with a maximum of 15 minutes execution timeout. In case Lambda is triggered via any AWS service, the execution timeout could be less as well, such as 29 seconds for API gateway.

#### *Asynchronous Invocation*

Asynchronous invocation can be achieved via S3, Simple Notification Service(SNS), CloudWatch events, etc. In this invocation mode, response is returned immediately and request is queued by Lambda for processing. It can additionally be

configured to handle retries and direct invocation response to configured **destination** unlike synchronous mode of invocation.

### *Polling Invocation*

This is the preferred mode for stream or queue based services such as **DDB streams**, **Kinesis**, **SQS queues** or **Kafka**. In this invocation mode, Lambda holds responsibility to poll the AWS service and synchronously invoke Lambda function for processing. The good part is AWS doesn't charge customers for message polling and we only pay for Lambda function invocation. The retry mechanism is dependent on the data expiration in AWS service, such as from one minute to 14 days for SQS queue.

We mentioned Lambda provision resources on invocation, which could result in more request processing time due to cold start problems. Cold start refers to the time Lambda takes to provision resources and prepare an execution environment involving function code download from S3 or ECR, creating an environment with memory and runtime configurations and executing any initialization code out of main function code. After this complete set up, Lambda starts serving customer requests. Cold start time could vary from 100 ms to 1 second and typically occurs for 1% of total invocations as analyzed by AWS on production workloads. AWS recommends multiple ways to reduce cold start problems as described below:

### *Warm Start*

The execution environment is not deleted as soon as the function completes its execution, it is retained for some time so as to serve continuous requests invoking the same Lambda function. This ensures skipping the set up of execution time on each Lambda invocation, referred to as warm start. The cold start is not in the customer's hands and Lambda can create an execution environment on a need basis. For example, Lambda creates a

fresh execution environment for concurrent invocation to handle increasing traffic.

We can use warm start for our own advantage by intentionally invoking the Lambda function with dummy input before it serves actual requests. This approach is only recommended for scheduled or low-traffic workloads and not for production environments operating at large scale.

### *Provisioned Concurrency*

Provisioned Concurrency ensures the execution environment is set up in advance ready to serve requests. For example, Lambda function with Provisioned Concurrency of four will create four execution environments in parallel and it can be extremely helpful for functions with large initialization code-base.

### *Memory Configurations*

AWS provides an option to configure memory that is available for Lambda function varying from 128 MB to 10,240 MB. The compute power allocated to Lambda function is proportional to configured memory so it is advisable to tune memory settings to improve performance. You can come up with the best memory configuration for your workload via running test runs on Lambda function. Additionally you can explore [AWS Lambda Power Tuning](#) tool for this purpose.

### *Static Initialization Optimization*

As lambda prepares the execution environment, it configures any required connections or download code dependencies. A large number of dependencies can definitely add to the cold start time, so it is recommended to import only required dependencies. Additionally if the function code base becomes too large where such a scenario is not feasible, other avenues can be explored such as breaking a function into multiple child functions.

## *SnapStart*

For Java11 runtime environments, snapstart improves function start-up time by taking an encrypted snapshot of memory and disk state of the initialized execution environment. The new function invocation resumes its execution from the cached snapshot, therefore improving on cold start time.

As we mentioned, Lambda supports a variety of invocation modes and scales automatically for required traffic. The key deciding factor when deciding whether to choose Lambda as a compute platform is deciding whether you're fine with too much abstraction of infrastructure. Consider: Is there a requirement to fine tune system efficiency at the infrastructure level? Is it ok if there is a latency spike due to cold start issues? These are some of the questions you should try to answer in order to make your decision.

If you'll recall, we took a deep dive into different containerization concepts in Chapter 6. Here we'll explore how we can use containers as a compute platform and launch our systems on AWS Cloud.

## **Containerization Services**

AWS offers container orchestration service for running Docker containers via two services, referred to as Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). ECS is a fully managed container orchestration service offered by AWS while EKS is a managed service for running open-source **Kubernetes** on AWS cloud.

### **NOTE**

This section is built on top of concepts covered in Chapter 7—please make sure you've read through that chapter first.

# Amazon Elastic Container Service

Amazon ECS is a fully managed highly available service that helps with deployment, operational management and scaling containerized applications for the required traffic load. ECS integrates with other AWS services, such as EC2, ECR (Elastic Container Registry), and Elastic Load Balancer, to provide a complete container management solution. Let's cover some important terminology related to ECS:

## *Task*

Task is the basic unit of deployment in ECS responsible for running one or more containers as shown in [Figure 8-3](#).

## *ECS Service*

ECS service groups identical tasks to scale and monitor in a single place.

## *ECS Cluster*

The infrastructure is registered to an ECS cluster, which acts as logical grouping of ECS tasks or services running via ECS Fargate or ECS EC2 launch types, or both types in the single cluster.

## *Task Definitions*

Task Definition defines various [configurations](#) for a task, such as runtime platform, task size, launch type, data volumes, environment variables, container definition for configuration specific to container launched within a task, etc. Task definitions enable you to define complex multi-container applications and specify how they should be orchestrated. Here is an example with Fargate as the launch option:

```
{  
  "family": "fargate-task-definition",  
  "containerDefinitions": [  
    {  
      "image": "aws_account_id.dkr.ecr.us-west-
```

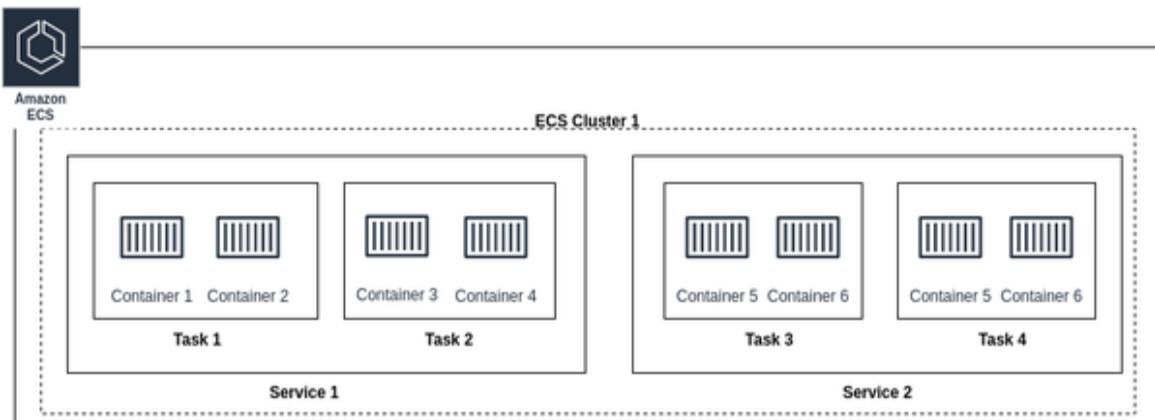
```

    "image": "2.amazonaws.com/repository:tag",
      "name": "my-fargate-application",
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80,
        }
      ]
    },
    "essential": true,
    "cpu": "256",
    "memory": "512",
    "executionRoleArn":
    "arn:aws:iam::aws_account_id:role/ecsTaskExecutionRole",
    "networkMode": "awsvpc",
    "runtimePlatform": {
      "operatingSystemFamily": "LINUX"
    },
    "requiresCompatibilities": [
      "FARGATE"
    ]
  }
}

```

### *Amazon Elastic Container Registry (ECR)*

**ECR** is an AWS managed container image registry which can be used to push, pull and manage docker or Open Container Initiative (OCI) images.



*Figure 8-3. ECS task, service, and cluster representation*

There are two types of launch options available with ECS, the first being ECS EC2 where we manage our containers on a fleet of EC2 machines, and the second being ECS Fargate, a serverless option where we hand over complete operational management responsibility to AWS. The key differences between these two deployment options are captured in Table 11-1.

T  
a  
bl  
e  
8  
-  
1  
.A  
m  
a  
z  
o  
n  
E  
C  
S  
L  
a  
u  
n  
c  
h  
T  
y  
p  
e  
s

Comparison Factor   Amazon ECS   EC2   Amazon ECS Fargate

---

Operational Management	Compute layer is managed by customers—such as instance type and number of EC2 instances, application scaling, OS level patching, etc. ECS EC2 supports auto scaling feature to adjust number of instances based on resource utilization or application demand.	Serverless, meaning customers don't have to worry about server management tasks such as applying OS security patches. We can specify OS, <b>vCPU</b> , <b>memory</b> and auto scaling configurations.
------------------------	--	---

Pricing	There is no separate cost for ECS, customers are <b>charged</b> for EC2 instances running time, plus any storage selection, such as EBS volumes.	<b>Cost</b> is based on vCPU, memory, CPU architecture, and storage selection for ECS tasks or EKS pods, and customers are charged only when container workloads are active.
---------	--	--

Business use-cases	AWS recommends ECS EC2 for workloads with high CPU or memory requirements, cost optimization requirements, application requirements for persistent storage	AWS recommends ECS Fargate for large workloads to be optimized for low overheads, small workloads with occasional bursts,
--------------------	--	---

access or compliance and organization requirements to manage infrastructure on your own.

and batch workloads.

Limitations	Extra operational overhead at the customer's end for selection of EC2 instances, maintaining fleets, OS patching, etc.	No support for GPU and EBS volumes for persistent storage. You can use <b>EFS volumes</b> for persistent storage and bind mounts for ephemeral storage. In short, there are less customizations available for customers as compared to ECS EC2.
-------------	--	---

ECS seamlessly integrates with other AWS services such as CloudWatch for observability, IAM for security and access control, **AWS Cloud Map** and Application Load Balancer (ALB) to provide service discovery and load balancing capabilities. Cloud Map allows users to register and discover services dynamically, making it easier for containers to communicate with each other. ALB provides load balancing across containers, distributing traffic based on configured rules, helping to achieve high availability and scalability. Here are some additional key benefits of AWS ECS:

### *Scalability*

ECS allows you to scale your containerized applications seamlessly based on demand. It automatically adjusts resources to handle traffic spikes efficiently via auto scaling or serverless compute options.

### *High Availability*

ECS ensures high availability of your containers by spreading them across multiple Availability Zones within a region. It automatically recovers failed containers and keeps your applications running smoothly.

### *Cost-Effective*

With ECS, you only pay for the resources you use. It optimizes resource allocation and scales based on actual demand, helping to reduce costs.

### *Security and Compliance*

ECS integrates with AWS Identity and Access Management (IAM), allowing you to control access to your containers and resources. It also provides encryption options for data in transit and at rest.

As mentioned earlier, EKS is another service offered by AWS to launch applications on containers. Let's explore how it differs from ECS and in which use-cases customers generally prefer to go with EKS instead of ECS.

## **Amazon Elastic Kubernetes Service**

EKS is a fully managed Kubernetes service that simplifies the deployment, management, and scaling of containerized applications using Kubernetes on AWS Cloud. AWS EKS eliminates the need for manual setup and configuration of Kubernetes clusters, allowing developers to focus on their applications rather than the underlying infrastructure. EKS is available with EKS EC2 and Fargate launch types, similar to ECS. Now you might wonder, why are there two different containerized services and how do I decide which is best for my use-case? Some of the key benefits of AWS EKS include:

### *Kubernetes Managed Control Plane*

Amazon EKS manages the Kubernetes cluster control plane which includes the API server and **etcd persistence database**. This eliminates the administrative overhead of managing and scaling the control plane from the customer's plate, ensuring high availability and reliability guarantee from AWS.

### *Multi-AZ Support*

Amazon EKS supports running Kubernetes clusters across multiple availability zones, providing enhanced availability and fault tolerance. By distributing cluster resources across different AZs, applications running on EKS can tolerate failures and continue to operate seamlessly. The EKS control plane runs in three AZs with support for automatic detection and replacement of unhealthy nodes.

### *Automated Kubernetes Version Upgrades*

EKS automatically handles Kubernetes version upgrades for the control plane, simplifying the process of staying up-to-date with the latest features and security patches. This helps ensure that clusters are running on stable and secure Kubernetes versions without requiring manual intervention.

### *Hybrid Deployments*

For highly latency sensitive workloads, EKS on **AWS Outposts** can be used to operate applications at on-premise data centers. AWS Outposts extends AWS infrastructure, services and tools to customer's personal data center so essentially making it feasible to run AWS EKS workloads there.

### *Integration with AWS Services*

EKS seamlessly integrates with various AWS services, enabling developers to leverage the extensive ecosystem of AWS resources and capabilities. It integrates with services like Elastic

Load Balancing, Amazon VPC, AWS Identity and Access Management (IAM), AWS CloudTrail, and more, providing a unified experience for managing and securing applications.

### *Scalability and Elasticity*

With Amazon EKS, scaling Kubernetes clusters to accommodate increased workloads is effortless. It leverages Amazon EC2 instances and Auto Scaling Groups to dynamically scale worker nodes based on demand or we can use Fargate launch as a serverless option. This allows applications to handle traffic spikes and changing resource requirements seamlessly. We can also leverage EC2 spot instances to reduce our cost of EKS clusters or use GPU optimized instances for high performance computing.

### *Security and Compliance*

EKS incorporates several security features to protect applications and data. It integrates with AWS Identity and Access Management (IAM) to control access to cluster resources, supports fine-grained network policies using Amazon VPC, and allows encryption of data at rest using AWS Key Management Service (KMS). EKS is also compliant with various industry regulations, such as [SOC](#), [PCI](#), [ISO](#), [FedRAMP-Moderate](#), [IRAP](#), [C5](#), [K-ISMS](#), [ENS High](#), [OSPAR](#), [HITRUST CSF](#), and [HIPAA](#).

### *Monitoring and Logging*

EKS provides integrations with popular monitoring and logging services, including Amazon CloudWatch and AWS CloudTrail. These services enable developers to gain insights into cluster performance, monitor resource utilization, and track API calls and events for auditing and troubleshooting purposes.

### *Ecosystem and Community*

As an open-source platform, Kubernetes has a vibrant ecosystem and community. EKS benefits from this ecosystem by providing compatibility with Kubernetes-native tools, frameworks, and operators. Developers can leverage these tools to enhance their application deployment, management, and observability within the EKS environment.

Your love for open-source could be just another reason to choose AWS EKS, where you have complete visibility about the ongoing development and Kubernetes internal architecture. EKS on AWS offers the benefits of both worlds, open-source support, as well as the offloading of management tasks to AWS, along with easier integration with other AWS services.

## Conclusion

Selecting a compute platform for launching applications could be a difficult choice, so you should clearly lay out all the application requirements or expectations in order to help you to make a wise decision. There is a great philosophy of a ‘two way door’ decision-making process at Amazon which states, ‘you can always revert back your decision if things are not working out as expected’. In a similar way, the compute platform can always be reiterated as application scope or traffic pattern changes for your application. Here are few recommendations we think will be helpful in making your decision about your compute platform:

### *Flexibility*

EC2 offers maximum flexibility to users with customizations at the hardware level, in addition to how software applications are deployed. In contrast, Lambda abstracts the resources completely and customers only have to focus on the code with restrictions such as a maximum 15 minutes invocation time. The high flexibility brings large operational overhead of system

maintenance such as OS patching for security issues, figuring out the most efficient instance type for application, configuring auto scaling policies, deploying applications, etc.

### *Learning Curve*

As we start out building and launching applications, familiarity with the platform is another factor to consider. More familiarity means less unknowns and quicker application launches. For example, if you're completely new to any compute platform, it's very easy to start off with Lambda and launch applications in the minimum time possible as you just have to focus on writing code and skip all headache of application deployment and maintenance for required traffic.

### *Traffic Patterns*

All compute platforms offered by AWS can operate at high scale. You should understand requirements in hand and how a specific platform is best for serving the business use-case along with cost efficiency. For example, EC2 or ECS EC2 provides more hardware control so could be a better choice for high latency sensitive operations or GPU accelerated computing.

### *Cost*

It's important to consider the cost we pay for resources in AWS Cloud. The pricing model for each of the compute platforms varies and there is no direct head to head comparison. Cost could be dependent on traffic patterns, size of workload, resource selection such as EC2 instance type or ECS Fargate task size or Lambda memory configuration, etc. We recommend considering operational overhead as well, because eventually there will be people maintaining the operations.

While you might not know the best solution right off the bat, don't forget, it's always a two-way door decision and can always be reiterated. In the next chapter, we'll be exploring orchestration services offered by AWS Cloud, such as Simple Queue Service (SQS) and AWS Step Functions, along with how application health can be monitored via metrics and logging on AWS Cloudwatch.

# Chapter 9. AWS Orchestration Services

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

For running systems at large scale, multiple components interact and coordinate with each other to perform any task referred to as orchestration. This communication and coordination between application components can be managed by different AWS services and we can pick and choose our services based on our use-case. In this chapter, we'll deep dive into core orchestration services such as **Simple Notification Service (SNS)** which can be used to broadcast a message to multiple subscribers and **AWS Step Functions** which can be used for coordination and execution of different AWS or custom services in specific order.

We'll start off with discussion around AWS services which are helpful in achieving publisher-subscriber design pattern in the AWS Cloud environment such as **Simple Queue Service (SQS)**, **SNS** & **Amazon**

Managed Streaming for Apache Kafka (MSK) and then later on move on to monitoring, authorization and authentication services such as CloudWatch, Amazon Cognito and AWS IAM.

## Amazon Managed Streaming for Apache Kafka

We discussed message brokers and their architecture in Chapter 8. Apache Kafka is an open-source message broker and event streaming platform used extensively for designing event driven architectures. AWS offers managed service for Apache Kafka to reduce operational overhead on the customer's end, allowing them to focus more on developing software rather than infrastructure deployment and maintenance of Apache Kafka clusters. Amazon MSK exposes control-plane operations such as create, update & deletion of clusters and data-plane operations such as production and consumption of events to customers.

When **setting up** the Amazon MSK cluster, we specify the number and type of broker nodes along with storage capacity in each AZ. A minimum of two AZ must be selected to ensure high cluster availability. Amazon MSK automatically detects broker failures and replaces the node with a healthy node with the same IP address. Similar to multiple other services in the AWS environment, Amazon MSK is also offered in two capacity modes, Serverless and Provisioned. We can choose the Serverless option to let AWS worry about infrastructure configuration while Provisioned allows us to select nodes and storage configurations. Additionally, the coordination and synchronization between the broker nodes is managed via **Zookeeper** (discussed in Chapter 5) nodes and Amazon MSK creates these nodes for us.

The next set of configurations in cluster setup are related to networking, security and monitoring. You should select AZs and subnets in which brokers should be launched and then define

security measures. First let's discuss on how clients can access Amazon MSK cluster:

### *Unauthenticated access*

As the name suggests, we can allow clients to directly access our cluster without any authentication, though this is not a recommended way to configure access control.

### *IAM role based access*

The access to Amazon MSK cluster is managed by IAM roles and associated IAM policies.

### *SASL/SCRAM authentication*

**SASL/SCRAM** means Simple Authentication and Security Layer/Salted Challenge Response Mechanism. We essentially use sign-in credentials (username and password) for clients and store them to **Amazon Secrets Manager** associated with a secret resource. This provides a secure method of cluster access while Amazon Secrets Manager takes full responsibility for audit, update and rotation of credentials. Additionally, we can share the same credentials across the clusters as necessary.

### *TLS Authentication*

We can use TLS authentication for client access and store these certificates in Amazon Certificate Manager (**ACM**).

For data security, we can encrypt our data both at rest and in-transit. Encryption at rest can be enabled by AWS managed or customer managed key. For in-transit encryption, we can use TLS encryption for communications within the cluster and use PlainText or TLS Encryption or both for communication between broker and clients.

**Amazon Kinesis** is another service which provides data streaming capabilities similar to Amazon MSK and we'll deep dive into it in the next chapter. AWS also offers SNS and SQS (which we'll discuss shortly) which can be used for creation of topics for publishing events and the same can be consumed by multiple consumers such as SQS. In Chapter 11, we discussed why customers might prefer EKS instead of ECS for launching their applications. The similar analogy applies here, Kafka is an open source message broker service which can be used in place of SNS-SQS. The reasons you might choose it are similar as well—you could be migrating your workloads to AWS Cloud or running existing workloads on AWS Cloud, or you just love open-source software with full visibility on development features. Let's move our attention to SQS and SNS now which can be used in place of Amazon MSK for management of topics and queues.

## **Amazon Simple Queue Service**

Chapter 1 introduced you to the concept of asynchronous communication and Chapter 7 to Message Queues. SQS is a fully managed AWS queue service which automatically scales as per customer traffic requirements. It is a highly available service which doesn't require any maintenance and deployment work from the customer's end. Some key use-cases where SQS stands out are:

### *Application decoupling*

Two microservices don't interact with each other directly and communicate with SQS inbetween. This avoids any service dependency such that one service being down doesn't impact the other.

### *Back pressure control*

The client consuming messages from SQS can consume messages at its own rate. The messages in SQS are kept until a

maximum of 14 days or as per customer configurations.

Let's understand the important concepts and key considerations related to SQS:

### *Visibility Timeout*

The message from the queue is not deleted after it is received by one consumer, instead it is kept in SQS owned temporary storage until a timeout setting, referred to as visibility timeout. Once the consumer processes the message, it should inform SQS to delete the message within the visibility timeout. In scenarios set up for no communication till visibility timeout, the message is made available again for other consumers to consume. It can have values from 0 seconds to 12 hours.

### *Retention Period*

This is the amount of time the message is kept in the queue if not received by any consumers. It can have values from one minute to 14 days.

### *Dead Letter Queue*

There may be scenarios where a consumer failed to process a message but we might need the message for use-cases such as retrying after sometime or for debugging purposes. This is where Dead Letter Queue(DLQ) can help—the consumer can push failed messages to DLQ for any further action at a later point of time.

### *Maximum Message Size*

The maximum allowed message size is 256KB. This allowed size should be good enough for all standard use-cases but for scenarios with requirements for more than it, we recommend you consider S3 or DDB. The message can be stored as an S3 or DDB object and the same can be referenced in a SQS message.

## *Message Delivery*

Two consumers can't compete for a single message at the same time, only a single thread can process a message at once. This ensures reliability as multiple producers can send messages and multiple consumers can receive messages.

## *SQS Types*

SQS offers two types of queues—Standard and FIFO. Standard queues don't guarantee message ordering while FIFO guarantees message ordering, refer to Table 12-1 for a detailed comparison.

T  
a  
b  
l  
e  
9  
-  
1  
.S  
t  
a  
n  
d  
a  
r  
d  
v  
s  
F  
I  
F  
O  
q  
u  
e  
u  
e  
s

Comparison Parameter Standard Queue FIFO Queue

---

Message ordering	Message ordering works on the best effort basis. It is not guaranteed that messages are delivered in the same order they are received by SQS.	Message ordering is guaranteed.
Throughput	Unlimited throughput, scales as per customer needs.	Maximum allowed is 300 TPS. Additionally, requests can be batched in a batch of 10 so via this, essentially we can achieve a maximum of 3000 TPS.
Message Delivery	Messages might be delivered more than once to clients, standard queue ensures at least-once delivery. The application logic should be idempotent to avoid unexpected behavior.	Messages are delivered exactly once.
Cost	Relatively cheaper than FIFO.	<b>~25% costlier</b> than standard queues.

SQS is widely used with Simple Notification Service(SNS) to meet message broker requirements without actually maintaining any infrastructure such as support use-case of **messaging fanout**. For a single message to be delivered to multiple recipients, it can be published to an SNS topic to which multiple SQS are subscribed to. The copy of the message is sent to each SQS from which it can be independently processed by clients. Let's dig more into SNS—we'll discuss what use-cases it solves and how it differs from SQS.

## Amazon Simple Notification Service

SNS is an intermediary service which enables communication between producers and consumers. A producer essentially publishes a message to a SNS topic and then SNS holds the responsibility to forward this message to all the subscriptions (aka consumers) via push mechanism. The supported consumers are Amazon Kinesis Data Firehose streams, SQS, Lambda, HTTP(S) endpoints, email, mobile push notifications and mobile text messages. Going back to our Cafe Delhi Heights restaurant example from previous chapters, let's think a little bit about different offers and coupons we get on our phones. As your phone number is saved to the restaurant system and you agree to receive text messages, the restaurant sends messages if there are any new offers or discounts. Here, the restaurant is a message producer which publishes messages to discount channels and we as subscribers get these messages.

Before moving onto important concepts related to SNS, let's understand the main differences between SQS and SNS via Table 12-2.

*T  
a  
bl  
e  
9  
-  
2  
.S  
Q  
S  
a  
n  
d  
S  
N  
S  
c  
o  
m  
p  
a  
ri  
s  
o  
n*

Message Delivery mechanism	Application should poll messages from SQS.	SNS pushes the messages to its subscribers.
Parallelism	Only a single consumer can access a message at any time and it's the consumer's responsibility to delete messages from the queue once processed.	SNS pushes messages to all its consumers in parallel and each consumer can process the message independently.
Message Delivery Latency	As it's the application's responsibility to poll messages from SQS, it can introduce some latency.	Messages are delivered to consumers as soon as they are published (near real-time) to SNS topics.

From the above differences, it's clear that both SQS and SNS solve different use-cases and one service should not be seen as a replacement to another service. Now, let's explore more on SNS by looking into important concepts and key considerations related to SNS:

### *Message Attributes*

Message attributes are optional metadata items that can be sent along with the message body of messages, represented by key, value and type with supported types being String, String.Array,

Binary and Number. These attributes can be used to take appropriate action on a message without processing the message body. Message attributes can optionally be used to filter messages at subscriber level. For example, a SNS topic publishes weather condition messages for the entire Asia continent and it has the country code included in the message attributes. So if the application only wants to process India-specific messages, it can directly filter out other messages without even reading the message.

### *Message Filtering*

You saw one example above of how message attributes can be helpful in message filtering. This is achieved via assigning a filter policy to topic subscription, this way only the intended messages are delivered to subscribers. Filter policy can also be applied on the message body, though the message body should be a well-formed JSON object.

### *Message Durability*

The message received from the publisher is stored across multiple AZs before acknowledging success. This ensures message availability even during AZ downtimes.

### *Message Delivery Failure Handling*

There can be scenarios when a subscriber is not available to receive messages, so SNS has a retry policy based on the **delivery protocol** to overcome unavailability issues. In case the message is not delivered even after retries, we can configure a dead-letter queue so messages are pushed there and can be polled when our system comes back online.

### *Message Security*

To ensure data security or to handle sensitive data, we can enable **Server Side Encryption** (SSE) on SNS topics by specifying an AWS KMS key. Once enabled, SNS encrypts the received message and stores it in encrypted form. The message is decrypted when it is being forwarded to topic subscribers. SSE only encrypts the message body so we recommend avoiding storing any sensitive information in topic and message metadata.

### *Message Ordering*

To cater strict message ordering needs and prevent message duplication, SNS supports **FIFO topics** similar to what we discussed in SQS. We can use standard topics by default if there are no such requirements.

A combination of SQS, Lambda and SNS is widely used for multiple use-cases to support event-driven asynchronous architecture and orchestrate simple workflows. For use-cases with requirements to orchestrate complex workflows in event-driven steps, let's explore AWS's offered workflow orchestration services.

## **Workflow Orchestration**

AWS offers two workflow orchestration services, Step Functions and Amazon Managed Workflow for Apache Airflow (MWAA). Let's take a simple example of online food ordering at our favorite restaurant—Cafe Delhi Heights. The sequence of steps to place an order after food selection are: process payment, send food order requests to restaurants, choose delivery partners, and track delivery, as shown in **Figure 9-1**. Both the offered services are fully managed by AWS so customers can focus on building their applications without any worry of infrastructure maintenance. Increased application reliability is the main benefit of executing application logic in steps as it ensures

independent execution (issues in one component don't impact others) and proper failure handling.



Figure 9-1. AWS Step Function Food Ordering Workflow

AWS Step Functions is an Amazon proprietary service, while Amazon MWAA is built on top of open-source [Apache Airflow](#). Let's explore more on both of these options.

## AWS Step Functions

AWS Step Functions (SFn) is an AWS fully managed serverless and visual workflow orchestration service which helps customers to create and run **state machines** with ease to coordinate among the distributed applications components. Here are some basic terms you need to know in order to fully understand SFn:

### *State Machine*

A state machine represents the orchestration workflow as a sequence of steps, relationship among them and their input & output. These steps coordinate among themselves and interact with different components such as an [Activity](#), Lambda Function, SQS, DynamoDB or any [other service](#) to complete a specific operation.

### *Task*

A single state or step in state machine is referred to as a task, which determines action to be executed, such as calling Lambda to fetch an object from a S3 bucket and parse it. In short, a single state's responsibility is to take input from the previous state, execute specified work and pass on the output to the next state.

## *Activity*

A task holds responsibility to perform some work—the work can be executed by workers on an EC2 machine, ECS, Lambda function or any application with the ability to make HTTP connections hosted anywhere. This is enabled by SFn feature called activity. A SFn workflow step awaits for activity workers to poll SFn for work via [GetActivityTask](#) API, complete the work and send execution callback to SFn via [SendTaskSuccess](#) or [SendTaskFailure](#) API. The timeout value to wait for callback can be configured in task definition as per the nature of application and your use-case.

We use [Amazon States Language \(ASL\)](#) or SFn console to create our workflows. ASL is JSON based structured language to define workflow states declaratively as described in the code snippet:

```
{  
    "Comment": "Amazon States language Example",  
    "StartAt": "Hello World",  
    "States": {  
        "Hello World": {  
            "Type": "Task",  
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloWorld",  
            "End": true  
        }  
    }  
}
```

SFn always encrypts the data at rest using transparent server-side encryption and all the data being passed from SFn to any integrated services is encrypted using Transport Layer Security (TLS). This ensures protecting our sensitive data without any operational overhead. The action being taken by the state is determined by the type of state. Here are different state types that we can choose from while designing our state machine:

*Pass*

The **Pass state** doesn't perform any work and just passes its input as its output. This state is mostly useful in state machine debugging and can also be used in **transforming input** so as transformed data is passed to the following state.

### *Task*

As we discussed in previous sections, task essentially performs the work in a state machine. The **Task state** represents a single unit of work via a Lambda function, activity or API actions of different AWS services.

### *Choice*

The **Choice state** is similar to the if-else statement of any programming language. It is a conditional statement that evaluates to true or false to determine further actions to be taken in a state machine.

### *Wait*

The **Wait state** helps in taking a pause action during state machine workflow execution. The timeout value can be a relative value in seconds from state start time or an absolute time as exact timestamp.

### *Succeed*

The **Succeed state** is terminal state to stop workflow execution successfully.

### *Fail*

The **Fail state** is the opposite of Succeed state and stops the workflow execution with failure. We can also explicitly catch the failure to take any specific action or just ignore failure to move onto the following state.

## *Parallel*

The **Parallel state** is helpful in adding multiple branches in a state machine. This can be helpful in executing independent states in parallel to make workflow processing faster.

## *Map*

The **Map state** is helpful to run a set of workflow steps in parallel for each item in the dataset as a JSON array or CSV file. Map state supports **inline** and distributed mode of operation and we can select one of them as per our requirements.

- Inline mode supports input as a JSON array from the previous step and can run up to 40 concurrent executions and the execution history should not exceed 25,000 entries.
- Distributed mode can support JSON or CSV files stored in Amazon S3 or a JSON array from the previous step. This mode offers high concurrency as compared to inline mode and supports up to 10,000 parallel executions without any hard limit on execution history.

### **NOTE**

The difference between parallel and map state is support for dynamic parallelism. We define multiple branches to be executed in parallel in parallel state while map state can create parallel child branches dynamically based on input.

We should always be prepared for service disruption and to overcome failure scenarios in SFn, we recommend adding **error handling, retry mechanisms** and alerting mechanisms as applicable for better success of the state machine.

The business requirements vary from use-case to use-case, such as event processing volume, system durability, workflow execution

time, idempotency, etc. To address these different requirements, SFn is offered in two types—Standard and Express workflows. The detailed comparison of these two types is captured in Table 12-3.

T  
a  
bl  
e  
9  
-  
3  
.S  
t  
a  
n  
d  
a  
r  
d  
a  
n  
d  
E  
x  
p  
r  
e  
s  
s  
w  
o  
r  
kf  
lo  
w  
c

*o  
m  
p  
a  
ri  
s  
o  
n*

Comparison Parameter	Standard Workflows	Express Workflows
Primary use-case	Used for long running workloads(up to 1 year), durable and auditable workflows such as ETL workloads.	Used for short lived (up to 5 years) high-volume event-processing workloads such as streaming data processing.
Execution model	Workflow is executed exactly once unless explicit retry behavior is in place.	Support is available for both synchronous and asynchronous execution models. The execution is at least once for async and at most once for sync workflows.  Sync workflows can be invoked from API Gateway, Lambda function or by using <a href="#">StartSyncExecution API call</a> .

Support for SFn activities	Supported with a maximum of 1000 pollers allowed on the same activity arn.	Not supported.
Execution history	Maximum 25,000 entries in execution event history. To avoid this, we can explore using the Distributed mode Map state or starting a new state machine from Task state of running state machine execution.	No hard limit on the number of events.

#### **NOTE**

The type selection should be an informed choice as it can't be changed after workflow creation.

We can use the AWS SFn console to debug our workflows for failures on any other analysis, such as execution time, input, output, failure reason, retry mechanism, etc. which helps in diagnosing production issues with ease.

## **Amazon Managed Workflow for Apache Airflow**

**Apache Airflow** is an open-source tool to programmatically create, schedule and monitor workflows. Amazon MWAA uses message queues to orchestrate an arbitrary number of workers to desired scale, but it also comes with setup and operational management

cost. AWS takes this workload off the customer's plate and offers Amazon MWAA as a managed service to scale as necessary. Here are some features which distinguishes it from open-source tools:

### *Scalability*

We can define minimum and maximum of workers for the Amazon MWAA environment and AWS handles the capacity management.

### *Security*

We can use IAM roles and policies for different AWS services access management and additionally, all the workers and schedulers run in [Amazon MWAA's VPC](#) ensuring private fleet management. All data is encrypted by default using AWS KMS as added security. We can enable private or public access modes as necessary by using [AWS IAM](#) policies and [AWS IAM Identity Center](#). Public access mode ensures accessibility of Apache Airflow server over the internet via VPC endpoint while private access mode is only accessible in your VPC.

### *Operations Management*

The setup is very easy with a few clicks in the AWS console and we don't have to worry about version upgrades or applying any security patches.

### *Monitoring*

To monitor and analyze Amazon MWAA tasks and workflows, we can use Apache Airflow logs and Apache Airflow metrics in CloudWatch.

We require logs and metrics capabilities to monitor our own application health as well as the health of AWS resources we're

using to create our infrastructure. Let's turn our attention to CloudWatch service which offers these features.

## Amazon CloudWatch

Launching an application on AWS Cloud infrastructure is the first step, but now we have to ensure this application keeps on running, serving customer traffic without any issues. To monitor application health and debug any issues that occur, AWS offers CloudWatch. CloudWatch helps with storage & search capability on application logs, visualizing application metrics & dashboards and setting up alerts on any unexpected application behavior.

## Application Logs

As applications run to serve customer requests, there are logs generated—these can be custom logs or language runtime specific logs. These logs are helpful in debugging any issue that has occurred in the system. We can publish logs to CloudWatch in real time for any compute platform we're using and look for specific error logs via CloudWatch search functionality or CloudWatch Insights feature. Here are some key features and terminology associated to CloudWatch logs:

### *Logs Storage*

CloudWatch can act as a central repository of all of our application or other AWS services logs. The logs are part of log streams, which are a sequence of log events that share the same source. Further all the log streams with similar properties such as source, retention, monitoring and access control settings are part of a log group. For example, software applications have been set up to create hourly log streams which means every hour a log stream can be created similar to log\_stream\_2023\_05\_15\_07 (log\_stream\_name\_year\_month\_day\_hour).

## *Personal Information Identification Data*

We should avoid logging any personal identification information such as social security numbers, medical history, credit card details, etc. This might be pushed to production by mistake and to detect such logs, CloudWatch provides a capability to **identify and mask** the log statement by leveraging pattern matching and machine learning.

## *Logs Search*

We definitely need a search capability on the top of logs to help with debugging. CloudWatch supports multiple filters by following pattern matching syntax. We can additionally use these filters to identify specific patterns in logs and publish a metric for it such as counting ERROR 400 for a particular API.

CloudWatch offers Log Insights as an additional feature for interactive search, analysis and visualization of logs data. We can search in up to 50 log groups in a single request via supported **query language**. For example, total number of InternalServerException across five microservices log groups.

For general visualization such as CPU utilization or total invocations of an API where we don't want to run queries to plot graphs on top of data, the effective way is to utilize CloudWatch metrics and CloudWatch alarms can be used for raising alerts for any unexpected behavior.

## **Metrics and Alarms**

CloudWatch metrics helps in application monitoring and enables users to search & graph metrics and create alarms on top of them. For example, it can plot service custom metrics such as number of 500 HTTP error codes and raise an alarm if, for a continuous 15

minutes, the total count of 500 HTTP errors per minute is greater than 10. Let's dig into metrics and alarms in a little more detail.

## Metrics

A metric is essentially a time-ordered set of data points being sent to CloudWatch for visualization. As mentioned in the previous section, the metrics can be AWS resource metrics or custom application metrics. Most of the AWS services publish some default metrics without charging customers, and we can definitely further tune them as necessary—for example, with minute level granularity. Similar to Log Insights, [CloudWatch Metrics Insights](#) feature can be used to query and find patterns on different published metrics and further create alarms on them. Let's understand some terminology associated with metrics:

### *Namespace*

Namespace is a root level identifier to distinguish metrics from one another. For example, EC2 service metrics are captured in the EC2 namespace. Similarly we can create our custom namespaces, for example, multiple microservices in a single AWS account can be separated by namespaces.

### *Dimensions*

Dimension is a metric identification property represented by name-value pairs. A single metric can have up to 30 dimensions associated with it. For example, for representing API latency of specific operation of an application, the namespace is application name with two dimensions as API name and metric type.

### *Metric Resolution*

You can choose between standard resolution and high resolution, depending on the granularity required for your use-case. Metrics are available at a minimum of 1-minute granularity in standard resolution or 1-second in the case of high resolution.

## *Statistics*

Statistics can be used for data aggregation over a period of time —a couple of examples are maximum latency or sum of API invocations in the last hour. We can choose from a [list of supported statistics](#) according to our business needs, such as Maximum, Minimum, SampleCount, Sum, Average, Percentile, etc. Further, each statistic has a [unit associated](#) with it, such as count, percent, seconds, etc.. For example, the average CPU Utilization in the last hour would be shown as 75 percent.

The CloudWatch metrics can also be streamed for further analysis to the destination of our choice in near real-time in JSON or [OpenTelemetry 0.7.0](#) format. A single metric stream can include up to 1000 filters which are helpful in streaming selected metrics, giving us control on the metrics we want to stream.

Some use-cases where this could be helpful are:

- Deliver metrics to S3 via Amazon Kinesis Data Firehose Delivery Stream. This can be useful (along with billing data) to look for cost optimization and resource performance.
- Third-party service providers can be chosen as a destination for monitoring and troubleshooting applications.

Metrics are good for visualization and debugging purposes but for raising an alarm or sending a notification for unexpected behavior, we should set up alarms.

## **Alarms**

Application monitoring is an important aspect for maintaining application health and alarms help to monitor application metrics and notify users or other applications to take appropriate action when the configured thresholds are breached. For example, you can take auto scaling action by adding one instance when an

application's EC2 CPU utilization is greater than 75 percent and remove one instance when it is less than 30 percent. Let's discuss the important things to consider when setting up alarms:

### *Metric*

An alarm is set up on top of a single metric or a math expression on a combination of metrics. To set up an alarm, it should include all metric information such as metric namespace, metric name and all dimensions.

### *Alarm Conditions*

An alarm is invoked when specific conditions are met and these conditions are defined by parameters such as threshold type, alarm condition, data points to alarm and missing data treatment.

- There are two supported types of threshold configurations: static and anomaly detection.
- Static configuration can be used to configure a specific threshold value on greater than, greater/equal to, lower than and lower/equal to alarm conditions.
- Anomaly detection allows the use of a band of values as a threshold with supported conditions as outside of the band(greater than or less than), lower than and greater than the band.
- We might not always want to raise an alarm with just one metric data point. This is a configurable property for an alarm and can be set up as per alarm requirement.
- There can be scenarios when metric is not pushed to CloudWatch—this can be due to application issues or if there was no actual metric to be published for a specific time period. In both of these cases, we can define how to treat the missing data. The supported behaviors are missing data being

treated as good (within the threshold), bad (breaching the threshold), ignored (maintaining current alarm state) and missing (transition alarm to insufficient data for alarm condition evaluation).

### *Configurable Actions*

The alarm can be in three states: ok state, in-alarm state or insufficient data to evaluate alarm. The alarm actions can be configured on state transition from one to another. The supported alarm actions are notification, auto scaling action, EC2 action, and systems manager action.

- Notification action allows us to notify people via sending a message to the Simple Notification Service (SNS) topic.
- Auto Scaling action allows us to add or remove instances from an auto scaling group as per alarm threshold.
- EC2 action allows us actions such as rebooting, stopping, terminating or recovering of an EC2 instance.
- We can also set up **Systems Manager** action by creating an **OpsItem** which the operations team can look into and take appropriate action on an alarm.

We can also create an alarm on top of multiple alarms, referred to as *composite alarm*. This can be helpful to reduce operation noise by configuring actions on composite alarms and skipping the same on child alarms. The composite alarm is invoked based on the configured conditions or **rule expressions** such as reach to alarm state when all child alarms are in alarm state or any one of them in alarm state. For example, you could set up a composite alarm on application resource utilization such as CPU utilization, memory usage and storage space.

The alarms are activated on state change in metric based on a certain threshold. Similar to this, any state in AWS resources is

captured via CloudWatch Events.

## CloudWatch Events

AWS services or resources transition from one state to another state. For example, as we launch an EC2 machine, it transitions from PENDING to RUNNING state after its entire setup and configurations. These state changes can be streamed in near real-time via CloudWatch Events to a desired destination to take any appropriate action. CloudWatch Events can additionally be used to schedule automated actions using cron or rate expressions. For example, we could run a Lambda function everyday at 7 AM to aggregate the previous day's earnings for our online food delivery restaurant. Some other **target destinations** include EC2 instances, Amazon Kinesis Data Streams, ECS tasks, **System Manager Run Command** and **Automation**, SQS, SNS topics, etc.

AWS offers another service similar to CloudWatch Events called Amazon EventBridge. Both of these services have the same underlying architecture but Eventbridge provides more features to customers. Amazon EventBridge provides backward compatibility to CloudWatch Events APIs and works on a similar concept to CloudWatch Events—it receives an **event** for state change and applies an associated **rule** to route this event to up to five **targets**.

AWS console automatically redirects to EventBridge page on selection of rules from CloudWatch console. We can also use the CloudWatch Events page, we recommend using EventBridge for more features, such as event ingestion from Software as a Service(SaaS) applications. Let's discuss on few of these added benefits in more detail:

### *Integration with Saas Providers*

Looking at an example of a **PagerDuty** application, it can be used to collect data from multiple platforms to provide an unified view to inform team members about any issue occurrence. Let's say,

based on a PagerDuty alert, there is a requirement to restart an EC2 machine. This action can be automated by sending an event to EventBridge and it can deliver that event action to a specific target. If we remove EventBridge from the picture here, polling and custom web hooks are mostly used to fetch data from third party providers to our software application. Polling can be compute-intensive—you might receive empty responses multiple times, whereas web-hooks require communication over the public internet. To address this, Amazon Eventbridge allows integration with **SaaS providers** over private AWS networks without exposing it to the public internet.

### *Custom Event Buses*

Event bus provides a way to communicate from event senders to receivers. All events in CloudWatch Events are routed via the default event bus present in the AWS account, whereas Amazon EventBridge allows creation of **custom event buses** specific to customer workloads and controlled access.

### *Enhanced Rules*

Amazon EventBridge allows **content based filtering** to filter values in events. This allows us to apply filtering at the EventBridge level, which helps our application to consume only required event traffic. Additionally we don't need this filtering logic in our application.

### *Schema Registry*

Amazon EventBridge stores schemas for AWS services, integrated SaaS providers and our custom events, helping with inference during our development process.

CloudWatch is an extremely important AWS service ensuring health monitoring of other AWS services and our own custom applications

deployed in the AWS Cloud environment. Security of AWS services and resources is another important aspect to consider to avoid any misuse. The next section of this chapter explores AWS IAM service which we can use for access control mechanisms in our AWS account.

## AWS Identity and Access Management

We touched upon using IAM at multiple places in this book to enable access control and enforce security measures in terms of authentication and authorization for different AWS resources or our overall AWS account. As you create your AWS account for the first time, you log in via root user credentials (with full administrative access) but when accessing different resources or allowing multiple people to login and access an AWS account, we recommend you provide only required access to avoid any mis-use. This permissions granularity is defined via IAM users, groups, roles and policies. For example, a support engineer might only have read-only access for CloudWatch logs but a DevOps engineer might have access to create new log groups or delete existing log groups.

Before we dive deep into more features and benefits of IAM, let's discuss the basic terminology:

### *Root User*

Root user is the user generated as you create your AWS account with email and password credentials. The root user has essentially all the permissions to perform any action in an AWS account and it can't be denied access by any explicit IAM policies. AWS Organizations **Service Control Policy** (SCP) can be used to limit permissions of root user of a member account.

We recommend creating a separate administrative user to perform day-to-day tasks and only use the root credentials for

**tasks** which can only be performed by root user, such as account deletion, sign up for AWS GovCloud(US), register as seller, etc.

### *IAM User*

IAM users can represent a person or service with a specific set of permissions to perform a task. IAM users have long term credentials and don't expire over time. The credentials can be a username & password for logging to AWS console or combination of AWS secret key ID and AWS secret access key for logging programmatically. We recommend avoiding creation of IAM users and use IAM roles with temporary credentials wherever possible. If there is a requirement to create IAM users, we recommend to regularly update passwords and **rotate** aws secret keys.

### *IAM Group*

IAM Groups group users with the same set of permissions. We can't login to AWS accounts by using group names, but groups make operations easy to manage a similar set of users. This can be helpful in managing users with the same permissions and reduce our burden of assigning the same permissions again and again to users as compared to adding a new user to the group.

### *IAM Role*

IAM Role is similar to IAM User but it is not associated with a specific person or service. It is temporarily assumed by a person or service or another AWS account to perform any task. For example, applications running on an EC2 machine will assume a role to make API calls to other resources, such as downloading a file from S3. This ensures temporary access only for the needed time and is automatically revoked after that. IAM Roles can be created with restricted access and this also helps in preventing any unintended changes to the AWS resources environment.

We recommend IAM roles over IAM users with temporary access for a limited period of time. There may be some use-cases where IAM Users will be explicitly required, such as third-party workloads with inability to assume IAM roles, [AWS CodeCommit](#) access, Amazon Keyspaces access or any IAM users created for any emergency issues (we should ensure [Multi-Factor authentication](#) in these cases).

### *IAM Policy*

[IAM Policy](#) is a JSON object (apart from [access control lists](#) policy type which we discussed in Chapter 10 and is based on [XML structure](#)) representing a set of permissions which is attached to an IAM identity (user, group or role) or an AWS resource.

Consider the example above regarding how EC2 can be provided access to download objects from S3. Now for that role to have S3 download permissions, we need to attach an IAM policy similar to the code below. The [JSON document of IAM policy](#) essentially contains Effect as Allow or Deny, with Action specifying all the resource operations and Resource specifying determined resources with additional elements such as Condition, Principal and Sid.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3>ListBucket"],
      "Resource": ["arn:aws:s3:::samplebucket"]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": ["arn:aws:s3:::samplebucket/*"]
    }
  ]
}
```

```
]  
}
```

In Chapter 9 we discussed bringing Cafe Delhi Heights online in relation to different AWS storage services. Now you've created an AWS account to host and operate the restaurant online and hired many people to help you out. Everyone contributing in development and testing of software applications will require access to the AWS account. There are multiple ways you can provide access:

- Share the root credentials with everyone in the team. We don't recommend this option, as it provides complete access to everyone. This is not required and might be misused—either knowingly or unknowingly.
- Create IAM users and associate them to groups such as developer-group, devops-group, support-group, etc. This can still be considered a preferred option, but there are some downsides as we discussed in the IAM Role section above.
- You might already be managing the team via some Identity provider (IdP) such as Google, Facebook, Amazon, etc. We can create an IAM Identity Provider entity to establish trust between the IdP and AWS account. This helps all users to assume an IAM role for a limited time and a limited set of permissions without exposing AWS access keys.

As different teams work together, there will always be use cases where one team requires access to an AWS resource in another team's AWS account, or it could be possible that multiple teams own resources in the same account. For these scenarios, the access can be managed via trust and permissions policies also referred to as *Delegation*. The trust policy specifies trusted account members that are allowed to assume a role, and the permission policy specifies what permissions a trusted user has to perform a task.

AWS IAM helps with securing and managing different AWS resources access. AWS offers another service for user pool management working as a Identity Provider(IdP) and many other features for user authentications. Let's dig into this service, Amazon Cognito, in the next section.

## Amazon Cognito

Amazon Cognito is a fully managed highly scalable Customer Identity and Access Management (CIAM) service which helps customers set up and manage their identity pools for authentication (AuthN) and authorization (AuthZ). Amazon Cognito takes the complete ownership of managing compute and storage for supporting this. We can use Amazon Cognito for new user registrations, existing user logins, maintaining guest users and defining access controls. Let's take a look at a simple example: we want customers of our online restaurant platform to be able to upload images along with food reviews. To allow customers to upload an image directly to our AWS account's S3 bucket, we can provide temporary write access via Amazon Cognito.

### NOTE

AuthN establishes the identity of an entity and AuthZ determines what this entity has access to.

In AWS account access terms, AuthN establishes if you can login to an AWS account and AuthZ establishes if you can access a specific AWS resource.

There are certain concepts we should dive deep into for understanding working of Amazon Cognito:

*Identity Providers*

An Identity Provider (IdP) holds responsibility for storage and management of user's digital identities. Amazon Cognito can itself act as an IdP and we can also integrate it with third-party IdPs (supported protocols are **OAuth 2.0**, **SAML** and **OIDC**) such as Google, Amazon, Facebook, etc. It is also referred to as Federated Identity since it can be used across the platforms for user authentication. The creation of a trust relationship between AWS and an external identity provider is referred to as Federation. For example, you likely see Google as a sign-in choice on many applications you use day-to-day.

### *User Pools*

A User Pool is essentially a user directory which acts as an IdP. The user records are stored in the user pool directory both for users signed in via Cognito or via third-party integrated IdP. It additionally supports all the features such as sign up, forgot password, login, password lengths and policy, enabling MFA, etc.

### *Identity Pools*

Amazon Cognito Identity Pools help with temporary credentials for application access both for authenticated and non-authenticated users by taking valid tokens such as JWT token or SAML token as input. The temporary role can be assigned to users based on different tags we specify for users. If we look at an example of online food ordering for Cafe Delhi Heights restaurant, the users are allowed to look for food and select food items without logging into the system, but to finally place an order, users must be logged into the system. This can be managed by redirecting to Amazon Cognito Hosted UI once the user clicks on "place order" and then again redirects to payments to confirm order.

### *User Pool Hosted UI*

We need some kind of user interface for users to sign up or login to our application. Amazon Cognito offers a URL hosted by AWS for interaction between users and our application. This interface can include email, password for login, or any integrated third-party IdPs as shown in **Figure 9-2**. Once users are authenticated, they are directed to a specific application page specified as a redirect URL. We can also **customize** the UI page as necessary, for example with a custom logo or CSS customizations.

### Triggers

We can consume **Cognito Events** such as user pre-authentication or post-authentication and run Lambda functions based on them via Amazon Cognito Triggers. The Lambda triggers can help with running extra logic for AuthN and AuthZ of users for underlying applications. For example, we can run a custom set of rules of validation, such as noting that the user location for accessing a particular web page should be India before a user can sign up to the application.

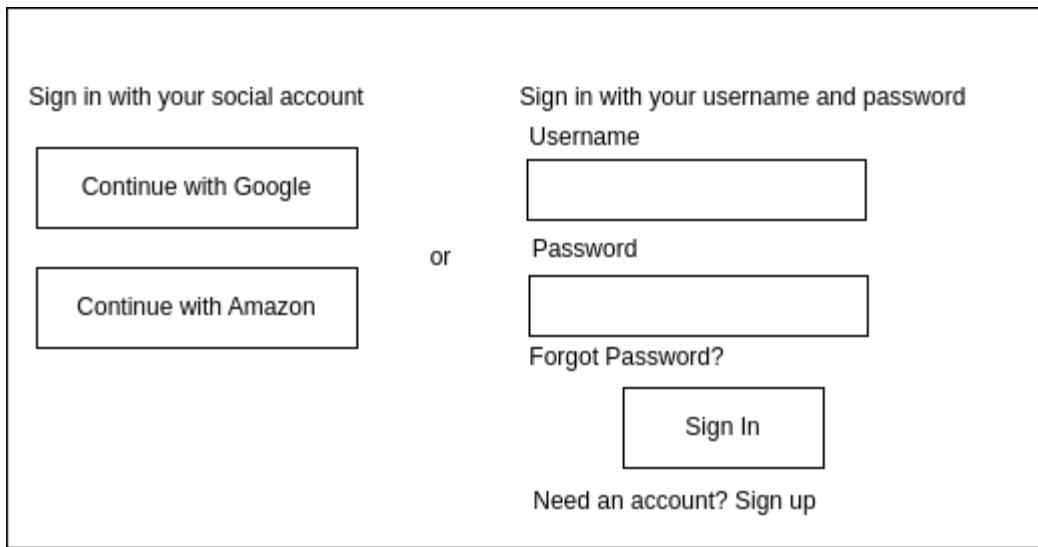


Figure 9-2. Amazon Cognito Hosted UI

There are different ways software applications can interact with Amazon Cognito for AuthN and AuthZ:

- Users interact with Amazon Cognito via the frontend application to get a token and use the same token in subsequent API calls. The backend application can interact with Amazon Cognito via AWS SDK Amazon Cognito APIs to validate those tokens against the users.
- Amazon Cognito provides native integration with API gateway without any customer efforts. In case our application APIs are offered via API gateway, we don't require manual token verification and it can be handled automatically as per the configurations.
- We can also use AWS Amplify to reduce the manual setup at our end and it automatically creates required infrastructure. We can use AWS SDK to interact with Amazon Cognito.

Let's get back to our example of using Amazon Cognito to assign temporary credentials for uploading an object to Amazon S3. There are essentially two ways of doing this: with Amazon Cognito or by using **S3 pre-signed URLs**. Let's discuss both of these options in more detail:

### *S3 pre-signed URL*

A pre-signed URL can be created with specific permissions on an S3 bucket or a specific object—note that the creator of the pre-signed URL should have the permissions to create it. The one downside of this approach could be pre-defining the object key name, so in order to create pre-signed URLs at runtime, we need some compute layer in between. For example, an AWS Lambda captures file name from a client, then creates a S3 pre-signed URL and responds back to clients to further upload an object.

### *Amazon Cognito*

We can use Amazon Cognito's Identity Pools to assign temporary credentials to users and allow them to directly upload files to S3.

The use of this definitely depends on the kind of use-case we're working on, as this option requires Amazon Cognito setup. It could make our work easy if there is already a setup in place for users AuthN and AuthZ.

The above options can also be used in conjunction as Amazon Cognito authenticates users and generation of pre-signed URL is handled by Lambda. We talked about temporary role assignment based on different tags in Identity Pools—the role can also be assigned based on token and IAM policy, such as you get the role related information as part of the token and you can use the same to assign the role. For requirements with dynamic role creation, **attributes for access control** can be helpful in IAM policy.

In the fast pace of development, AWS really helps by providing services such as IAM or Cognito to help with AuthN and AuthZ and we don't need to build solutions from ground up. Yet another service is AWS AppSync to help frontend and backend teams scale independently in development of APIs at large scale. Let's look into what AWS AppSync has to offer.

## AWS AppSync

To fully understand how AWS AppSync works, let's look at how food items are being displayed to customers on an online food ordering platform. The frontend application essentially displays food items along with their prices, restaurant coupons, food reviews, etc. and this data is maintained by different applications in backend microservices architecture. Here, the frontend application has two options, the first being to invoke APIs of each service and gather data, and the second being to have a single API which collects data from all these different backend services and returns to the frontend.

Recall that we discussed [GraphQL](#) in Chapter 6–Communications Networks and Protocols. AWS AppSync is essentially a single GraphQL endpoint that can be used to query multiple databases, microservices and APIs in a single network call. In the above example as well, we can use AWS AppSync to expose a single endpoint to the frontend which can gather data from different microservices. AWS AppSync can also be used for creation of Pub/Sub APIs to publish notifications to subscribed clients such as real-time sports updates via fully managed serverless websocket connections.

Let's move forward to understand the base components you should be aware of to start with AWS AppSync:

### *GraphQL Schema*

GraphQL Schema is essentially a data model specifying the query pattern for data retrieval.

### *Resolvers*

Resolvers are the components that provide a link between schema and different data sources. They are responsible for the conversion of GraphQL payload to underlying system protocol and execute if there are associated IAM permissions.

### *Data Sources*

Data Source is the data destination from which Resolvers can fetch the data using the access credentials. For example, DynamoDB, RDS, lambda, HTTP endpoints. The data sources shown in [Figure 9-3](#) can be present in the same AWS account or different AWS accounts.

### *Merged API*

A single AppSync API constructed from merge operation of multiple source APIs GraphQL schemas, resolvers and data

sources. The source APIs can be managed by independent teams and they can collaborate via Merged API.

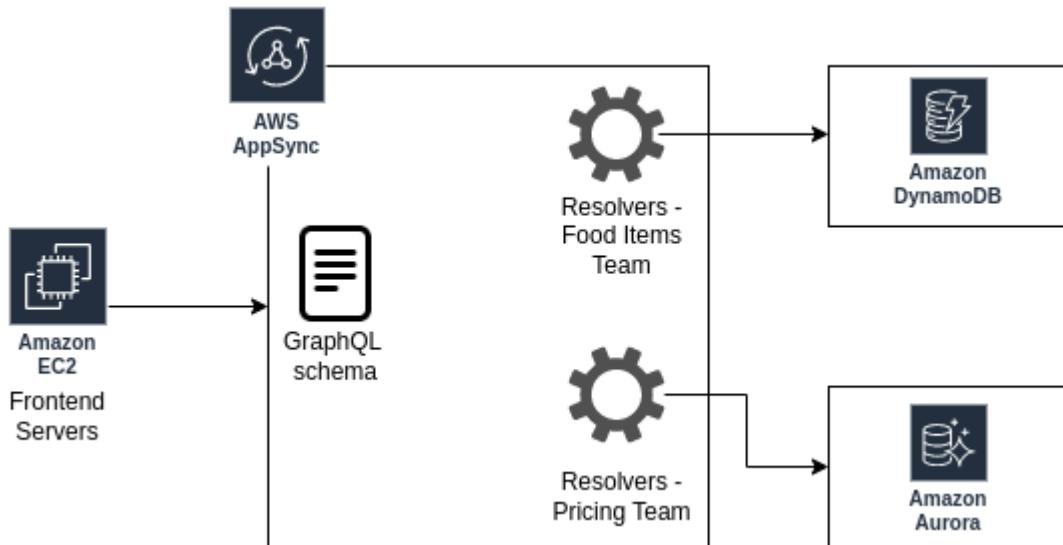


Figure 9-3. AppSync integration with multiple data sources

Now you might argue, why use AWS AppSync instead of deploying our own GraphQL infrastructure? The reason is fairly simple—using AWS AppSync removes all the operational overhead from the customer's plate, along with development effort for AuthN & AuthZ, caching, encryption and serving advanced use-cases such as building chat functionality, location aware notifications, live gaming scoreboards, **offline support along with AWS Amplify**, etc. To offer these benefits, AWS AppSync is a fully serverless application which offers **build-time composition of merged APIs** and easy integration with AWS services such as AWS WAF, Cloudwatch, Cognito and TLS encryption support.

## Conclusion

As you know by this point, AWS offers a wide variety of services and it is very important to understand your business requirements to know how to utilize the best services for your goals. For example, a fully managed serverless application such as AWS Step Functions will

not give us full visibility around how workflows are orchestrated behind the scenes, but the important question is do we really need to know that? Because AWS SFn is saving us a lot of management and operations costs with additional debugging capabilities via visual workflows.

We've also stressed upon application monitoring as an important aspect for ensuring high availability and this can also be achieved by placing metrics, logs and alarms in CloudWatch. We later deep dived into how authentication and authorization services can help in securing our applications and how different components in the AWS infrastructure environment can communicate with each other in a secure and restricted way. We recommend followig the principle of Least Privilege, essentially providing only the minimum set of permissions to application, user or another AWS account that is required to perform an operation.

In the next chapter, we'll explore how we can launch big data and machine learning workloads on AWS cloud and operate at large scale to serve our customers.

# Chapter 10. Big Data, Analytics and Machine Learning Services

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 13th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

In the world of information technology, data is generated at a huge volume. This data can be just the user information of all registered users on online food ordering applications or capturing real-time user actions on the application. The data generated at large volume is referred to as Big Data. If you have a use case to store this data, you can utilize the storage solutions we discussed in Chapter 10 based on your requirements. This chapter focuses on how we can process the data at high volume. How can we generate insights out of data already present in storage or live streaming data by running data analytics or machine learning models on top of it? For example, determining the most ordered food item based on location or restaurants with highest rating in particular locality.

The first part of this chapter introduces you to AWS big data, live streaming and analytics services such as Amazon Elastic MapReduce ([EMR](#)), [AWS Glue](#), [Amazon Athena](#), [Amazon Kinesis](#), [Amazon Redshift](#) and [Amazon Quicksight](#). The second section explores how you can run Machine Learning ([ML](#)) workloads on AWS Cloud and different services supporting it.

## AWS Big Data and Analytics

Information is vital to make business decisions or serve our customers better, but the volume of data is increasing, ranging from terabytes to petabytes (and more) along with the variety of data—the data can be in any form. We require specific tools for storage and processing of big data. The traditional tools can become a bottleneck as we can no longer operate on a single machine for data processing. Another challenge is the velocity at which this data is produced—we also require tools to consume the data and produce insights from it in near real time to gain maximum output. In this section, we'll talk about data processing tools such as Amazon EMR and AWS Glue; analytics tools such as Amazon Athena and Amazon Redshift; live streaming ingestion and analytics via Amazon Kinesis offerings; and business intelligence service called Amazon Quicksight.

### Amazon Elastic MapReduce

We introduced you to Hadoop in Chapter 8, an open-source software popular for big data processing. Amazon EMR is a managed service useful for execution of data processing frameworks and tools such as [Map Reduce](#), [Apache Spark](#), [Apache Hive](#), [Apache HBase](#), and [Presto](#). The set up and management of large clusters requires time and expertise but Amazon EMR makes it easy by offering a managed service where we can launch clusters in minutes and run large data processing workloads.

Open source Hadoop cluster defaults to **Hadoop Distributed File System** (HDFS) for data storage, and we can also explore using different tools available in the community to extend the storage to Amazon S3. HDFS is tied to the local disks which disappears once the cluster terminates. Now, as our use-case is processing big data, our storage will grow over time and since HDFS is attached to compute instances, the number of instances will also increase. There is a possibility that our use-case has high storage requirements but the compute requirement is relatively less, so we're paying for the compute but not using it at full capacity. This is where the **EMR File System** (EMRFS) appears as a better solution.

## EMRFS

EMRFS uses S3 as a file system for our data processing instead of local HDFS, so essentially it's a connector that links EMR clusters to S3. EMRFS ensures streaming of data directly to S3 and uses HDFS as intermediate storage.

Before we dive deep into EMRFS, please understand that EMRFS is not a solution to every problem and you should not see it as a replacement to HDFS. HDFS can still be a better option for jobs with iterative reads on the same dataset or I/O intensive workloads (meaning workloads that require frequent disk access and retrieving data over network, as these scenarios could prove to be latency intensive operations). You can use the combination of Amazon S3 and HDFS to address the temporary data storage (data is lost on cluster termination) limitation of HDFS by leveraging tools such as **s3-dist-cp** tool to copy data from S3 to HDFS or from HDFS to S3.

Let's look into some of the benefits that EMRFS offers over traditional HDFS:

- In scenarios when you require the same data to be accessed by multiple clusters, we need to copy data from one cluster to another since HDFS is associated with a single cluster and not shared among all clusters. We can save this cost by using the

EMRFS as a storage option. EMRFS allows multiple clusters to access the data from the same place, decoupling compute and storage completely, and also offers flexibility to scale them independently.

- In addition to the above point, we might not always require our EMR cluster to be running. So we can terminate the cluster and storage is maintained in S3 saving compute costs.
- To ensure data availability, HDFS replicates the data at multiple nodes as per the replicator factor. For example, with a replication factor as 3, we'll be paying 3x the storage cost. EMRFS is backed by S3 so we don't have to worry about **data durability** as it is already handled by S3.

The above data points give a clear picture of how EMRFS is a useful utility to consider for storing data. We can make best out of any tool if we follow certain best practices associated with it—let's discuss some tips around EMRFS:

- The data should be partitioned so that the EMR cluster should fetch only the data that is required for processing. This ensures faster data retrieval along with reduced costs.
- The file size should be optimized in a way that we don't have too many files or too few files. As we start off EMRFS usage, try to avoid files with size less than 128 MB because it will ensure less calls to S3 along with HDFS requests.
- We also recommend considering data compression, ensuring less storage space usage on S3. This helps in reducing storage costs at two fronts:storage cost and network costs for data retrieval.
- The data access patterns can vary based on business use-case. It could be possible that you're only interested in a subset of columns across the columns or a subset of rows across the

rows. **Apache Parquet** and **Apache ORC** are columnar file formats that give increased read performance for prior use-case and **Apache Avro**, a row optimized file format is useful for latter use-case.

Now you should have a fair grasp around the concept of EMRFS as a storage option. Now let's explore more on other EMR features and specifications.

## EMR Cluster Considerations

The data processing jobs run on a set of servers, and instance types can be chosen based on workload requirements. As a compute option, we can choose to run an EMR cluster on **EC2 instances**, **EKS** or go completely **Serverless**.

The EMR cluster architecture can have three types of nodes, namely master node, core node and task node as shown in **Figure 10-1**.

### *Master Node*

Master node is the primary node in the EMR cluster. It uses YARN resource manager service for application resource management and runs HDFS NameNode service to track status of jobs submitted on cluster along with instance groups health monitoring.

### *Core Node*

The core nodes are responsible for running the Data daemon for HDFS data storage and Task Tracker daemon to perform computation tasks on data. The EMR cluster can have a maximum of one core **instance group or instance fleet**.

### *Task Node*

Task nodes don't provide data storage—they should be used as extra computation power to the EMR cluster. EMR by default ensures that the application master of any job runs on the core

node to ensure the job doesn't terminate if any of the task nodes is terminated (as in case of EC2 spot instances). We can configure up to 48 task instance groups depending on the job requirements.

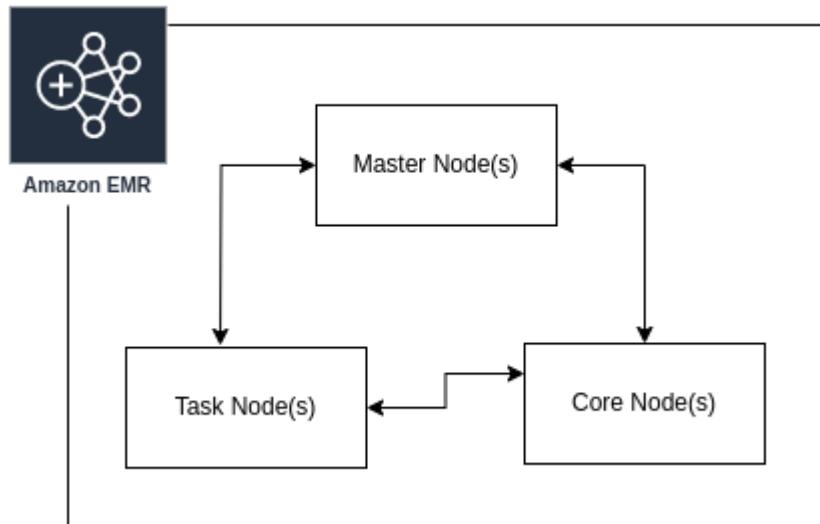


Figure 10-1. EMR Cluster nodes

EMR cluster allows customers to choose from the available applications and it will auto-install all the software without any effort from our end. There may also be a requirement to install any custom software or modify a configuration for nodes on EMR nodes. To support this, we can use **EMR bootstrap actions** which executes the action after the cluster instances are up.

The EMR cluster can consist of a large number of nodes depending on the requirement and this large number will definitely add to the resources cost we incur from AWS. We can introduce some cost saving mechanisms as described below:

- Amazon EMR provides configuration to auto terminate clusters. This ensures the cluster is auto terminated once the submitted jobs are completed.
- You may also have requirements for long running clusters as well. If you're sure of capacity requirements, you can explore

[reserved instances](#) and [AWS saving plans](#).

- If the EMR workloads are non-critical, you can use EC2 spot instances (discussed in Chapter 11). The spot instances are offered at up to 90% discount as compared to on-demand instances and can help substantially reduce the costs. You can also set up clusters with a mix of spot and on-demand instances to ensure the cluster is always up and running but also cluster pricing in mind.
- For the variable workloads on long running clusters, you can use automatic scaling policies on instance groups to add or remove instances.

Before we close our discussion on Amazon EMR, let's ponder upon few of the considerations to ensure high reliability of workloads running on EMR clusters:

- The instances in clusters should be spread across AZs to avoid any AZ downtimes. By default, the [Hive](#) metastore is stored on the primary node's file system. We recommend storing the metastore [outside of the EMR cluster](#) such as inside multi-AZ RDS cluster or Amazon Aurora to avoid data loss on cluster termination.
- We recommend using EMR multi-master instead of a single master node. The master node is a single point of failure and if it goes down, the entire cluster goes down. Multi-master node setup ensures to provide this safety net.
- The critical data should be kept in S3 instead of local HDFS to avoid any data dependency issues.
- We recommend EC2 spot instances only for task nodes and not for core nodes. As HDFS is managed by core nodes for data storage, termination of these nodes might lead to data loss.

To address the requirement of spawning EMR clusters at regular intervals, we can leverage Amazon EventBridge and automate the workflow via AWS Step Function like so: create an EMR cluster with specific configuration, submit jobs to the cluster and once the jobs are complete, terminate the cluster. In some scenarios, we just care about executing the ETL jobs without any worry of instance or storage management. AWS Glue is a completely serverless data integration application offered by AWS to perform analytical tasks on big data.

## AWS Glue

AWS Glue is a serverless data integration service helping to make sense of data with different features. Before we dig deeper into these features, let's quickly understand some of the terminology associated with AWS Glue:

### *Classifier*

A **Classifier** validates whether it can handle the data available in specific format and if it can, then classifies into a **StructType** object. We can use Classifiers offered by AWS Glue or define a custom Classifier as **needed** such as data file is not in format being supported by AWS Glue.

### *Metadata*

The metadata is the inferred schema by the Classifier from the available data in any of our datastores.

### *Database*

A database is a place where you keep metadata tables. A single table can be associated to a single database only and in case the database is not specified, AWS Glue uses the default database.

### *Data Catalog*

A Data Catalog maintains databases which then consist of one or more metadata tables. These tables can be useful as source and target

### *Data Crawler*

The responsibility of Data Crawler is to crawl the data from data sources such as Amazon S3, figure out the schema via Classifier and then create the required tables or partitions in Glue Data Catalog. For use cases where the data schema frequently changes, we can run **Crawler on schedule** and it automatically figures out modifications since the last run and creates new tables or partitions as per requirement.

### *Table Partitioning*

Partitioning is a way to improve the query performance on data. Let's understand it with the help of an example as shown in **Figure 10-2**. Our application processes sales data and is **stored in Amazon S3** divided into folders with parent folder as year and child folder as month and days. It can have further sub folders such as hours or minutes depending on type and amount of data. Now if the data is partitioned in such a fashion, it becomes very easy to query for it. Extracting data for February 2022 means going to the folder for Year 2022 and then to sub folder Month 02 making the query much faster. There can be multiple partition keys for a table and we can create a **partition index**, a subset of partitions to avoid loading all the partitions.

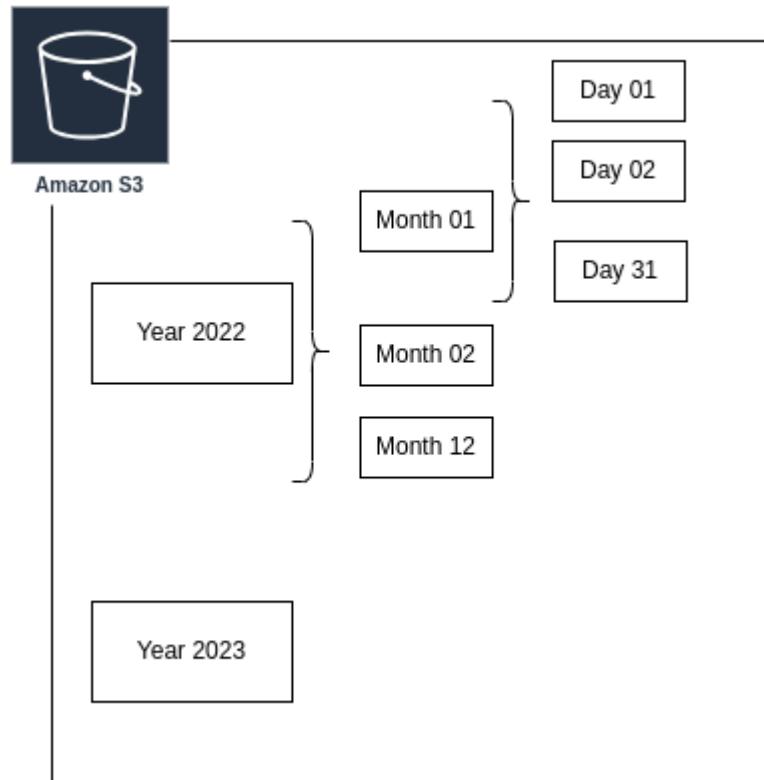
### *Data Engine*

Data Engine is helpful in running data processing jobs on top of the data. AWS Glue supports three data engines at the time of this writing, namely **AWS Glue for Apache Spark**, **AWS Glue for Ray** and **AWS Glue for Python Shell** and we can choose one based on our workload requirements.

- With AWS Glue for Apache Spark, we can write ETL code in Python and Scala languages and the code executes in a distributed environment. The Spark environment on AWS Glue can be assumed as an Serverless EMR cluster being operated by AWS. The Spark engine is supported for both batch processing and live streaming data. AWS Glue also addresses a very general problem of disk spilling due to data shuffling (data redistribution within the cluster) by providing **Cloud Shuffle plugin**. This plugin allows the use of Amazon S3 or any other object storage in data spilling situations. Further, this plugin is available as open source software so can be used in custom Spark jobs as well.
- With AWS Glue for Python, a single node Python engine is offered to run workloads written in Python. It offers default integration with open source libraries such as numpy, pandas, etc and we can add our custom libraries as well. The one problem that customers might face is workload scaling as data grows, which can be resolved by using **AWS Glue for Ray**.
- AWS Glue for Ray allows you to execute Python code in a distributed environment set up and is scalable to hundreds of nodes. This processing engine is based on open source compute framework **Ray** which is extensively used to run Python workloads at large scale such as deep learning models.

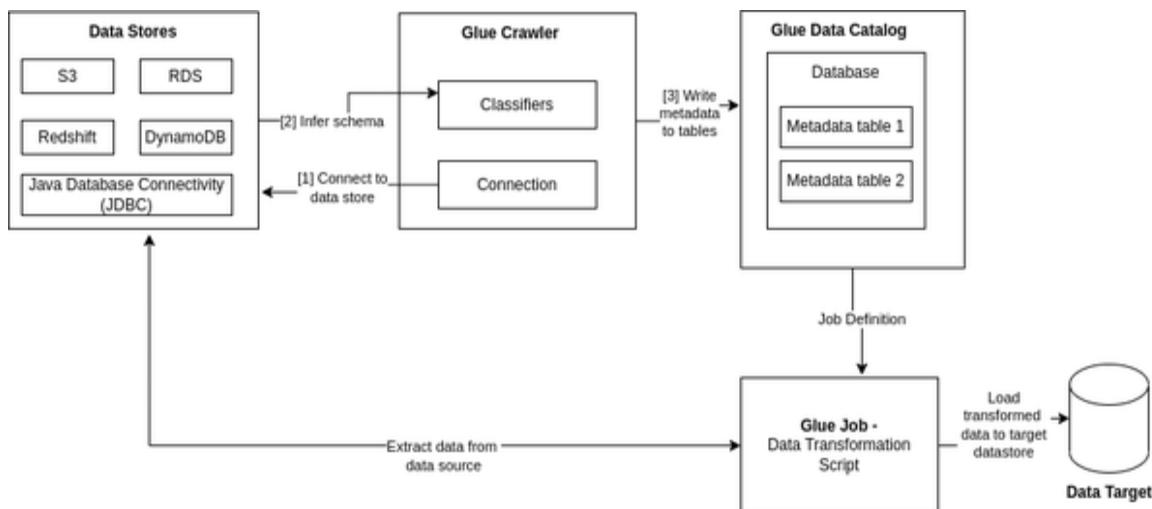
### *Data Processing Units*

AWS Glue allows customers to configure types of workers to run the data processing jobs. These workers are referred to as Data Processing Units (DPUs). We can **analyze our workload requirements** for best suited configuration of DPUs.



*Figure 10-2. Data partitioning folder wise in Amazon S3*

Let's combine our understanding of different concepts into a simple architecture around how AWS Glue operates in [Figure 10-3](#). Once the structure of data is created, we can create and run AWS Glue ETL jobs with any of the data engines of our choice for further visualization and analysis.



*Figure 10-3. AWS Glue architecture overview*

The above diagram illustrates a high level picture of how AWS Glue manages and runs ETL jobs. We'll now discuss some additional features and offerings of AWS Glue that you can utilize on need basis:

- AWS Glue offers Glue Studio with support for visual workflows, built-in transformations, and Glue Studio notebook. This helps to run tests on the data and once we're happy with testing, the code from the notebook can be imported to an ETL job.
- AWS Glue offers a **Data Quality** check feature that automatically figures out conditions or logics on which data quality is validated. We can further override them as required. Consider an example of sales data: A sales company started off its operations in the year 2021 but there is some data with the year marked as 2019. Here, we can be sure that this is invalid data and should be scraped out or require alerts put on top of it. This feature is available for both data at-rest and in-transit.
- There is another validation feature of **Personal Identifiable Information** (PII) data supported by AWS Glue. This can be used to detect sensitive data and take any appropriate action such as masking it.
- To reduce compute cost and configure an optimal number of DPU's to run AWS Glue jobs, we can leverage the **Auto Scaling** feature. This helps to automatically add or remove workers as per workload requirement. We can also configure a maximum worker number to ensure AWS Glue doesn't configure workers above a particular limit.
- In Chapter 11, we discussed EC2 Spot instances and how they can be helpful in reducing overall compute costs. AWS Glue offers similar functionality with **Flex job type** offering up to 34% cost savings. We can leverage this in scenarios of non-critical workloads.

- There may always be use cases where it would be useful to **schedule** the crawlers or jobs to run at a particular time or on regular intervals. AWS Glue inherently provides support for this. We can additionally trigger jobs and crawlers on external events via AWS Eventbridge.
- AWS Glue offers easy integration with **GitHub and AWS CodeCommit** to manage source version control for our AWS Glue jobs.
- AWS Glue provides **AWS Glue Schema Registry** as another useful feature. As the name dictates, it's a registry where we can publish our schemas and can also enforce on data streaming service integrations such as Amazon MSK, Apache Kafka, Kinesis Data Streams, Kinesis Data Analytics and AWS Lambda.

From the above long list of features, it is evident that AWS Glue is a powerful ETL service and helps a lot to reduce operational overhead and can operate at large scale. After navigating through Amazon EMR and AWS Glue, you might be wondering how the Spark application running on Amazon EMR serverless option differs from AWS Glue Spark. Let's throw some light on this in Table 13-1 as both of these options look quite similar.

T  
a  
bl  
e  
1  
0  
-  
1.  
A  
m  
a  
z  
o  
n  
E  
M  
R  
S  
er  
v  
er  
le  
s  
s  
a  
n  
d  
A  
W  
S  
G  
lu  
e

*c  
o  
m  
p  
ar  
is  
o  
n*

Parameter	Amazon EMR Serverless	AWS Glue
Feature functionality	Data processing and analytics tool which leverages open-source software.	An end-to-end ETL solution with abstraction over software and it offers AWS customized solutions with focus on data integration, data catalog and running transformations on data sources.
Operational overhead	Fully managed service	Customers specify the number and type of DPUs.
Supported big data applications	Apache Spark and Apache Hive.	Apache Spark and Python jobs.

AWS Glue and Amazon EMR vary on their base functionality and we as customers should clearly lay out our requirements to choose one for our workloads. In the next section, we'll discuss **Amazon Athena**, a service helpful to query and analyze data stored in Amazon S3.

## Amazon Athena

Amazon Athena is a fully managed serverless big data analysis tool which offers SQL query support on top of data stored in Amazon S3. The data analysis doesn't require any transfer of data to any other data storage, the data can be directly read from S3 objects and can be queried upon. We can compare Amazon Athena with **Presto** running on an Amazon EMR cluster which provides similar functionality. The main advantage of such solutions is that data analysis can be performed on raw data without any transformation at the customer's end.

Amazon Athena requires that we as customers define the databases and tables with schema. We can do this task manually or leverage AWS Glue crawler/**Apache Hive metastore** to perform this task for us. Once the database and table is defined, we can query on this data with simple SQL. The key consideration here is as we execute the SQL query, Amazon Athena internally submits a job that operates in asynchronous fashion and the response to the query is saved to Amazon S3. Additionally, we can integrate Amazon Athena with various **BI tools** for interactive analysis such as Amazon Quicksight which we'll discuss later in this chapter.

Like multiple other services, Amazon Athena's main advantage is that you don't have to worry about infrastructure management. Let's discuss a few other features and considerations for using this service for our data analytics use cases:

- We mentioned that Amazon Athena is used to query on data stored in Amazon S3 but what about data present in other AWS data stores such as DynamoDB or on-premise data sources? For

these kinds of use cases, Amazon Athena offers a feature called **Federated Query**. This feature allows us to run SQL query on a variety of data stores, be it AWS Cloud or on-premise. Amazon Athena achieves this via a Lambda based data source connector—the connector essentially helps to establish a connection and retrieve data.

- In addition to the above point, Amazon Athena can have multiple Lambda invocations in parallel to fastening data processing tasks. It is also possible that data don't fit into Lambda memory and to handle this, the data is spilled to Amazon S3 and avoids any data loss situations.
- **User Defined Functions** (UDFs) is another feature of Amazon Athena supported via Federated query SDK. It allows you to write custom Java code on Lambda and invoke directly from SQL queries to perform any pre-processing or post-processing tasks on data. Consider a scenario of sensitive data handling; to avoid storage of sensitive information to S3 as part of Amazon Athena query response, it is masked via logic executed in UDF.
- Amazon Athena also allows us to use deployed Machine Learning (ML) models on **Amazon Sagemaker** (which we'll discuss later in the chapter). We can directly invoke these **models in SQL query** for required data analysis—one such example could be detecting negative reviews for our online food ordering platform.
- To help with regular scheduling of workflows or invocation based on certain events, we can leverage Amazon EventBridge. For use cases of workflow orchestration and integration with other services, we can utilize AWS Step Functions.
- The limitation of using Amazon Athena instead of self managed Amazon EMR clusters or Amazon Redshift (which we'll discuss shortly) could be time taken for query execution for large

datasets, so it might not prove to be the best solution for latency sensitive operations.

Amazon EMR, AWS Glue and Amazon Athena are powerful services for big data analytics tasks. The discussion we had up until now should help in evaluation of business use cases and how a particular service can be best suited for specific use cases. In the next section, we'll focus on event streams handling in near real time and run analytics on top of it via Amazon Kinesis.

## **Amazon Kinesis**

We started our chapter introduction by taking an example of capturing real time user interactions of users on our online food ordering application. Amazon Kinesis offers capabilities to serve similar kinds of use-cases along with analytics and ETL. There are four sub-services available under the belt of Amazon Kinesis, namely **Kinesis Data Streams** (KDS), **Kinesis Data Firehose** (KDF), **Kinesis Data Analytics** (KDA) and **Kinesis Video Streams** (KVS). Let's start our discussion with Kinesis Data Streams.

### **Kinesis Data Streams**

Extending the example of capturing real time user interactions, we've multiple devices operated by users referred to as source. This interaction stream is then ingested and stored by Kinesis Data Streams. As the stream is now available in Kinesis, it can be processed based on requirements such as Spark on EMR clusters and finally the results can be dumped to a destination such as DynamoDB or you can analyze the results with any analytical tool as shown in [Figure 10-4](#). In short, data streaming on Kinesis enables us to ingest, process and analyze huge volumes of data at large scale without any management overhead on our plate.

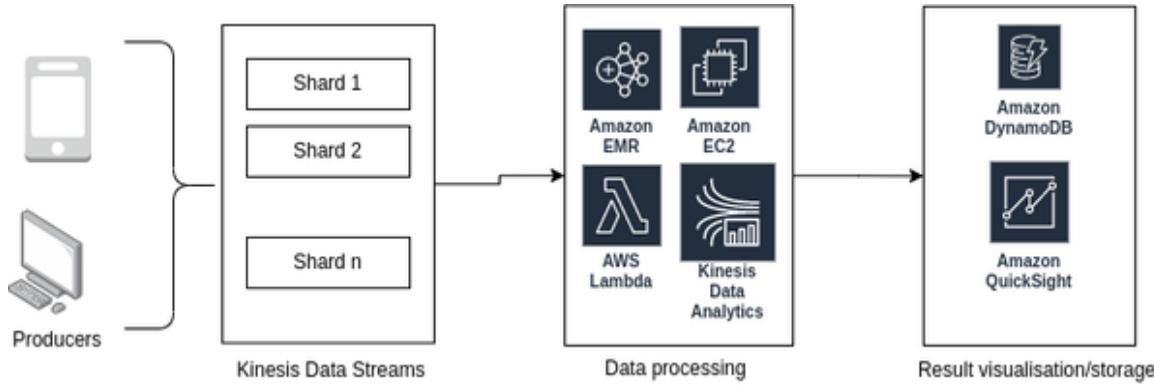


Figure 10-4. Kinesis Data Streams

A shard is the base throughput unit in KDS and is a uniquely identified sequence of data records in a stream. A shard can ingest up to 1 MB/second supporting up to 1000 TPS and can emit up to 2 MB/second to consumers. We can **configure a number of shards** by looking at the supported TPS by a shard and our business use-case production and consumption rate. This configuration is required for Provisioned mode of KDS. We can also explore On-Demand mode where this overhead is handled by AWS itself and it automatically scales as per requirement.

The producers publish data records to data streams which consists of a sequence number, partition key and data blob:

- The sequence number is auto-assigned by Kinesis Data Streams and it is unique per partition key within its shard.
- The partition key is helpful in grouping data at the shard level, meaning to which shard a data record belongs to. The key should be chosen such that it resolves to uniform data distribution across the available shards.
- Data blob is an immutable sequence of bytes which consists of the content or information with a maximum allowed size as 1 MB.

Let's move forward to discuss key considerations while using KDS:

- To address the limitation of 2 MB/seconds fan-out at a shard level, KDS offers an additional feature referred to as **Enhanced Fan-Out** (EFO). This can be used to have a dedicated pipe at each consumer level of shard to have 2 MB/second read throughput.
- KDS temporarily stores the data for a time period of 24 hours to 7 days and in the same time, it can also be replayed for any retry strategies.
- We can use **AWS KMS** to encrypt sensitive data as it enters Kinesis Data Streams.
- Kinesis Data Streams provides multiple options on shards such as **update** number of shards, **split a shard** or **merge shards** into one. This helps to ensure full capacity utilization of a shard. The update shard operation can internally merge the split shards to configure the updated number of shards config. Splitting or merging of a shard can be useful if a shard is too “hot” or “cold”, meaning it is receiving too high traffic or too little traffic. If the traffic is too high, it makes sense to split it to balance out the traffic while if the traffic is too low, it makes sense to merge the shards to optimize the number of shards.

We discussed Amazon MSK in Chapter 12 which also provides streaming capabilities similar to Amazon Kinesis Data Streams. As we discussed earlier in the book, the finalization of particular technology depends on the business use-case and no technology is actually better than the other—every tool has their pros and cons. With this in mind, let’s lay out how these two technologies differ from one another in Table 13-2.

*T  
a  
bl  
e  
1  
0  
-  
2  
.A  
m  
a  
z  
o  
n  
M  
S  
K  
a  
n  
d  
Ki  
n  
e  
si  
s  
D  
a  
t  
a  
S  
tr  
e*

*a  
m  
s  
c  
o  
m  
p  
a  
ri  
s  
o  
n*

Parameter	Amazon MSK	Kinesis Data Streams
Operations	<p>The base logical entity is partition and we're required to configure a number of brokers for the cluster launch.</p> <p>The Serverless compute option doesn't require customers to configure the number and type of brokers and it is fully managed by AWS.</p>	<p>The base logical entity is shard and we're required to specify the number of shards as per our workload.</p> <p>There is also an option of on-demand capacity which doesn't require customers to specify shard configuration and is managed by AWS.</p>
Latency	<p>The messages are available immediately</p>	<p>It offers low latency—data to consumers is</p>

	after it is written to the topic. The latency can be relatively at the lower end as compared to Kinesis Data Streams.	available to consume within <b>200 milliseconds</b> . With enhanced fan-out the latency is relatively lower than 200 milliseconds.
--	---	--

Message Ordering	Messages within a topic partition maintain an order.	Events within a single shard maintain order.
------------------	--	--

Message Delivery	Amazon MSK has support for exactly-once message delivery semantics. This could be helpful for streaming financial transactions with no requirement for explicit handling of duplication in application code.	KDS offers at least-once message delivery semantics. The application should have a <b>duplicate handling mechanism</b> in case the system doesn't expect duplicate records.
------------------	--	---

Pricing	<b>Cost</b> is based on number and type of broker instances per hour and the amount of storage allocated to the broker. For a serverless option, the cost is	Customers are <b>charged</b> by the shard and the number of PUT operations to write data into the stream. No charge for reading data which is less than 7 days old. For on-demand option,
---------	---	--

	<p>based on hourly rate of clusters and each partition being created along with storage required for write and read from the topics.</p>	<p>the cost is based on per GB data written/read from data streams along with the number of streams. There is additional charge for features like data retention for more than 24 hours and enhanced fan-out</p>
--	--	--

Data Retention	<p>Data retention can be configured up to the amount of storage available on the brokers.</p> <p>In case the requirement is to keep data for more than 1 year, Amazon MSK could be the preferred choice though we recommend looking into AWS archival storage solutions if it could be helpful for the business use-case.</p>	<p>Data is accessible by default up to 24 hours and can be increased up to a year.</p>
----------------	---	--

Migration Overhead	<p>Amazon MSK could be a great choice if you're already managing the Kafka clusters on your own on Amazon EC2 instances or on an on-premise data center. This helps in reduction of any migration workload to a</p>	<p>People with no familiarity with any of the services should look for other parameters to decide on a service to be used to meet business requirements.</p>
--------------------	---	--

different setup for streaming.

Integration with AWS services	Both KDS and Amazon MSK offer easy integration with other AWS services such as EC2, Kinesis Data analytics, EMR etc.
-------------------------------	--

The above table should be helpful in devising some insights around which solution could be better for any future use-cases. In the last chapter, we also discussed SNS and SQS in relation to the publisher-subscriber design model. If you think about it, KDS offers a similar kind of model where data is published to streams and multiple consumers can consume the data from a stream. The big difference is near real time processing being offered by KDS or Amazon MSK streaming solution. In the case of SNS and SQS model, the data should be ingested first to any specific system for analysis but KDS allows for data analysis as we receive the data. Please refer to Chapter 12 for complete understanding of additional features offered by SQS and SNS. Now let's move to another Kinesis offering, Kinesis Data Analytics (KDA) which is useful for analytics purposes in near real time as data is processed via Kinesis Data Streams or Amazon MSK.

## Kinesis Data Analytics

Kinesis Data Analytics is a fully managed service for [Apache Flink](#) capabilities that puts no overhead on the customer's plate and scales automatically for the desired throughput of incoming data. Kinesis Data Analytics helps in processing, querying and analyzing streaming data in near real time and sends it to the configured data

destinations such as Kinesis Data Streams, Amazon MSK, S3, and Amazon Opensearch. Let's try to get some understanding of Apache Flink before we discuss more specific to KDA.

- Apache Flink is an open-source distributed processing engine framework helpful in processing real time data streams.
- It supports both bounded and unbounded streams processing. Bounded streams refers to batch processing while unbounded refers to processing of streaming data in real time.
- It performs stateful computations in-memory ensuring low latency. The stateful nature ensures exactly-once processing of streaming data events. Apache Flink ensures the state is maintained even if the system goes down by asynchronous checkpointing the state to durable storage.
- It supports **SQL**, **Table API**, **DataStream APIs** and **stateful functions** for easy access of data. The SQL and Table APIs have the most abstraction while datastreams APIs and stateful functions give more granular control and visibility to customers. It has language support for Java, Scala, Python and SQL.

The deployments and infrastructure management can be a huge pain point for customers and also requires expertise around Apache Flink core concepts. Amazon KDA is a serverless offering, allowing customers to focus on business logic while AWS manages everything end to end. Here are details around some of the features specific to KDA:

- As KDA is available in the AWS environment, it offers easy integration with other AWS services such as Kinesis Data Streams, Amazon MSK, Amazon S3, Amazon Opensearch, Amazon Cloudwatch, Amazon DynamoDB, Kinesis Data Firehose and AWS Glue Schema Registry.

- KDA can also be easily integrated with open source tools via [Apache Flink connectors](#) and also offers support to build custom connectors for use-cases which are not supported as of now.
- The integration with AWS Glue Data catalog helps in storing and sharing of metadata across multiple applications.
- KDA offers [KDA Studio](#) which is basically a wrapper around Apache Flink SQL and Table APIs to analyze the streaming data in interactive way via [Apache Zeppelin](#) serverless notebooks. KDA Studio supports SQL, Scala and Python language.

### **NOTE**

A notebook is a web-based developer interactive environment running in the browser of your choice. The developers can write queries or code in supported languages for different use-cases such as data transformation, visualization and analysis.

Primarily, Kinesis Data Analytics is a fully managed version of Apache Flink and since it is part of AWS Cloud offerings, we can easily integrate with other AWS services. Now let's discuss one more use case where we just have a requirement to load live streaming data to destinations. This requirement is met via Kinesis Data Firehose so let's divert our attention to this service now.

## **Kinesis Data Firehose**

Kinesis Data Firehose (KDF) is data delivery service for live streaming data to different destinations such as Amazon S3, Amazon OpenSearch, [Amazon Redshift](#), [Splunk](#) or any custom HTTP/HTTPS endpoints to easily integrate with third-party data storage providers. KDF is a fully managed service and customers don't have to worry about infrastructure and service maintenance. KDF doesn't offer any storage of its own so we don't have the feature of replaying messages like we do with Kinesis Data Streams. We can additionally

configure KDF to transform data before delivering it. There are some built-in transformations provided by KDF, such as conversion to Apache Parquet format. We can also add our custom transformations using AWS Lambda functions.

Here are few considerations related to KDF:

- The data record sent by the producer to the KDF delivery stream should not exceed 1000 KB.
- To enhance security and reduce storage space at destination data storage solutions, we can batch, compress and encrypt the data before loading.
- KDF buffers incoming streaming data to a certain size (buffer size) or certain period of time (buffer interval) before being delivered to data destinations. The buffer size could range from 1 to 128 MB for Amazon S3, 1 to 100 MB for Amazon OpenSearch and 0.2 MB to 3 MB for AWS Lambda. The buffer interval for all these service integrations can range from 60 to 900 seconds.
- When Amazon Redshift is configured as a data destination, KDF first delivers data to the Amazon S3 bucket and then issues **COPY command** to load data from S3 bucket to Redshift cluster.
- When Amazon S3 is selected as a delivery destination, we can also enable dynamic partitioning of data. This helps at two fronts: it provides easy access of data in S3 for querying purposes and removes the need of partitioning at source or after the data is stored. We can specify a specific key or an expression evaluated at runtime to identify keys to be used for partitioning.
- There is a limitation on KDF when it comes to data delivery to multiple destinations. A single delivery stream can deliver data only to a single Amazon Redshift cluster or table, single Amazon

S3 bucket, and single Amazon OpenSearch cluster or index currently. If there is a requirement to deliver at multiple destinations, a separate delivery stream should be created.

- KDF supports at-least once data delivery semantics. There is a possibility that data is duplicated at the delivery destination due to retry mechanisms put in place for failure handling.

In short, KDF is a streaming ETL service fully managed by AWS for the customers. Now let's move to another Kinesis offering, Kinesis Video Streams service.

## Kinesis Video Streams

Kinesis Video Streams (KVS) are optimized for delivering live streaming video data to AWS in near real time from millions of sources for any processing, such as running Machine Learning algorithms or applying any custom video processing. KVS can be used for offline batch analytics as well, and supports other data streams such as audio data, images, RADAR data, etc.

KVS can be useful in a lot of **scenarios**, let's take an example of how KVS can be useful in building a household security system. People are not always at home and it could be really helpful if they can get a real time continuous feed of what's happening at their homes. To build this, a security company installs the cameras at home in desired locations. These cameras act as a source of information and have KVS Producer libraries installed to connect with KVS on AWS Cloud securely. The connection helps in constantly streaming the video feed to KVS where we can optionally store it or redirect the data to consumers. The consumers are applications that consume this data such as you accessing the feed on your mobile device. This could really be helpful in cases such as robbery or alarming you if there are unannounced people at home, and all this can be achieved in real time.

A few key considerations about KVS include:

- KVS provides durable data storage for configurable retention periods. It automatically encrypts the data at rest & in-transit and creates an index over this data based on producer-generated or service-side timestamps for easy & quick access.
- KVS is a fully managed Serverless offering so we as customers don't have to worry about infrastructure management or any expertise to tune any kind of configurations.
- KVS provides a parser **library** that helps consumer applications to get data from video streams in very low latency. It can be used in Java applications for MKV video streams.

We recommend analyzing your business requirements and then choosing the specific Kinesis offering which best suits your needs. Continuing on our journey to analytics tools at huge scale, now let's discuss the AWS powered business intelligence tool, Amazon Quicksight.

## **Amazon QuickSight**

Looking back at our online food ordering application, it stores data in multiple data stores such as total orders in a particular region, top rated restaurants, most ordered food items, etc. How can we create a single unified dashboard connecting to different datastores without providing direct access to datastores to everyone? This can easily be achieved by using Amazon Quicksight. Amazon Quicksight is a fully managed serverless business intelligence service offering analytics, visualization and reporting. It can connect to different kinds of datastores and include it in a single dashboard, the data can include AWS data, third-party data, spreadsheet data, SaaS data, and more.

Here are key features and considerations about Amazon Quicksight:

- Amazon Quicksight essentially enables every user to perform analytics and visualize the data for your use case with very less expertise. With Amazon Quicksight, you don't need to be

dependent on business intelligence teams for your data requests.

- Amazon Quicksight is offered in two variants, **Standard and Enterprise edition**. The Standard edition contains all the features described above. The Enterprise edition includes more advanced capabilities such as automated and customizable data insights powered by Machine Learning, security features including federated user identities, single sign-on, encryption at rest, etc. The Enterprise edition also helps to monetize the dashboards via pay-per-session pricing model. The consumers of the dashboard you've created pay for it per their usage.
- As part of Enterprise edition, there is an additional feature available referred to as **Amazon Quicksight Q**. It is a natural language processing tool where instead of writing queries, you can directly ask questions such as "top 5 restaurants in Mumbai".
- Amazon Quicksight supports fast advanced calculations to serve data via SPICE (Super-fast, Parallel, In-memory Calculation Engine). To make use of this in-memory data store, it should be enabled as part of the database creation or editing process and customers need to configure required **SPICE capacity**.

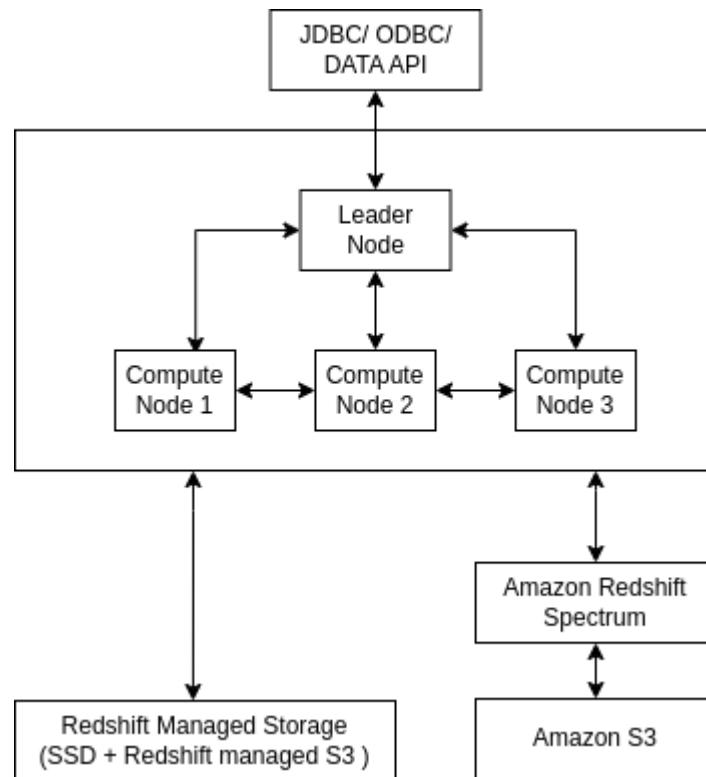
We mentioned that Amazon Quicksight is helpful to draw insights and it can do it for any data source. One such data source is Amazon Redshift which we'll discuss in the following section.

## Amazon Redshift

In simple terms, Amazon Redshift is a data warehousing tool which can act as a data store for data from multiple sources and allows us to run SQL queries at a single place for data analytics. Amazon Redshift is an AWS managed service and can scale to petabytes of data with an **elastic scaling** option. There is also a **Serverless** option,

which helps to avoid any burden of resource provisioning and workload management.

We'll start with discussing the architecture of Amazon Redshift and then move towards key features offered by this service. Amazon Redshift is columnar PSQL based storage and is based on leader follower cluster architecture where leader acts as a query coordinator who parses and complies the query and forwards the query to follower (compute) nodes, which works in parallel to gather results of the query as shown in [Figure 10-5](#). There can be 2 to 128 nodes in an Amazon Redshift cluster and in the case of the Serverless option, this entire architecture is abstracted out from customers. The query can be invoked from SQL clients via [JDBC/ODBC](#) connection or use Data API. The [Data API](#) offers query execution in both synchronous and asynchronous fashion and can query for results later within 24 hours with a query Id.



*Figure 10-5. Amazon Redshift architecture*

Here are some additional details to consider about the Amazon Redshift architecture:

- The compute nodes are partitioned into slices and each slice has its own allocated disk and memory to perform parallel processing.
- The data is stored into immutable **blocks**; A block consists of column data spanning across the multiple rows and a full block can contain millions of values with size of 1 MB.
- To improve query performance over the data blocks, Amazon Redshift maintains in-memory metadata information referred to as **Zone Maps**. It stores the minimum and maximum values for a block and helps in effectively pruning the data blocks that don't contain data for specific query. The Zone Maps can be optimized by using a **sort key** which defines how data is sorted on the physical disk. The sort key works well except for very few scenarios such as: when there is only one block per column per slice, the values within blocks have the same prefix (for strings longer than 8 bytes, Amazon Redshift takes the first 8 bytes as prefix so sorting won't matter if it's same), and when the column contains single distinct value (sorting won't matter as minimum and maximum values are consistent).
- To optimize the query performance, we can optionally choose to provide a **distribution style** for data. This ensures data is evenly distributed across the cluster and we can make best out of parallel processing. There are four distribution styles supported by Amazon Redshift; AUTO (default option), EVEN, KEY and ALL.
  - For the AUTO option, Amazon Redshift chooses the distribution style based on size of the table data and Amazon Redshift internally switches between the different distribution styles based on the size.

- For the EVEN option, data is distributed across the compute node slices in round robin fashion by the leader node. This is a recommended option when there are no join operations on the table.
- For the KEY option, the leader node checks the column value and places it on the compute node slice matching its value.
- For the ALL option, the entire table data is available on each node. This is only recommended if there are very few insert or update operations on the table.
- Amazon Redshift offers Redshift Spectrum which can use Amazon S3 as storage and the compute nodes are only responsible for gathering the results. This helps customers to directly query the data from Amazon S3 without loading it to Amazon Redshift clusters. One key consideration here is repeated reads against Amazon S3 as storage doesn't adhere to transactional guarantees.
- Amazon Redshift also offers Redshift Managed Storage (RMS) allowing customers to scale compute and storage independently. It uses S3 as persistent storage and offers high speed SSD backed Tier-1 cache support. RMS allows scaling up to 128 TB per instance and up to 16 PB per cluster storage capacity. For scaling further, we can always utilize Amazon S3 as storage.
- We can choose from **different instance types** available for our Amazon Redshift cluster. The feature is supported by the new Amazon Redshift instance type—**RA3**. The other instance types are DC2 (Dense Compute) with SSD as storage and DS2 (Dense Storage) with magnetic disks as storage. The DS2 instance types are legacy and we don't recommend using them for your workloads.

Amazon Redshift Spectrum is kind of similar to Amazon Athena where we query on top of the data that sits in Amazon S3. The one big difference is we can configure compute, allowing the queries to be much faster. There are definitely additional feature sets of both of these services which we should evaluate before choosing one for our workloads:

- Amazon Redshift allows us to create **Materialized views** from one or more tables to make the queries faster. The Materialized views contain pre-compute result sets based on table joins with all or subset of columns, aggregations and filters. We can also create Materialized views on top of already created Materialized views similar to the base tables.
- Amazon Redshift offers **workload management tools** (WLM) for separation of different query workloads and assigning priority so that important queries are executed and less important queries can be throttled or aborted.
- We can use **COPY** command to load data to Amazon Redshift from Amazon S3, Amazon DynamoDB, Amazon EMR or any remote host accessible via Secure Shell Connection (SSH). There is also support available for **auto-copy** which automatically ingests data from Amazon S3 to Amazon Redshift clusters.
- We can integrate Amazon Redshift with Amazon MSK or Kinesis Data Streams for ingesting live streaming data. We can use **Informatica Data Loader for Amazon Redshift** to access data from on-premise or third party applications.
- We can use Amazon Redshift **Federated Query** feature to access live data from external data stores without loading it to Amazon Redshift clusters such as Amazon RDS PostgreSQL & MySQL and Amazon Aurora PostgreSQL & MySQL.

- Amazon Redshift also allows **data sharing** across the clusters. This helps in isolating the read workloads without copying data on multiple clusters.
- Amazon Redshift provides **direct integration** with Amazon Aurora so as to avoid any pipeline set up by customers and the data is directly available in Amazon Redshift once pushed to Amazon Aurora within seconds.
- **Amazon Redshift Advisor** is a feature that provides recommendations for cluster optimization around query performance and cost savings. For example, query tuning recommendations, deletion of unused clusters, and table data compression.

In summary, Amazon Redshift is a great tool to use for data warehousing and analytics use cases scaling to petabytes of data. In the next section, we'll look into different Machine Learning services offered by AWS and how they can fit into different business use cases.

## Machine Learning on AWS

To support Machine Learning (ML) and Artificial Intelligence (AI) use cases, AWS offers a range of **services** from building and running our own ML models to leveraging fully customized applications for specific purposes. For example; **Amazon Polly** helps in converting text to speech, **Amazon Comprehend** helps in figuring out insights and relationships in unstructured data, **Amazon CodeGuru** offers intelligent recommendations to improve code quality, etc. The services can be divided into three broad categories: application services, platform services and frameworks & hardware solutions.

Application services are customized solutions for solving a specific use case. Platform services allow us to create our own services such as **Amazon Sagemaker** and **AWS DeepLens** by training and deploying

ML models. AWS offers different frameworks such as [PyTorch](#), [Keras](#), [Tensorflow](#), etc and special hardware with customized [CPU](#) and [GPU](#) to run ML workloads. We'll start off our discussion with Amazon Sagemaker and then briefly touch upon other offerings.

### NOTE

Intelligence means information—our ability to ask questions and provide reasons for the answers associated with these questions. We call it AI when this intelligence is shown by machines. ML is a branch of AI representing this intelligence based on previous data and algorithms. This book will not introduce you to ML concepts but only a few of the AWS services offering ML capabilities.

## Amazon SageMaker

The default benefit we achieve by choosing AWS Cloud is running any service without the worry of infrastructure management. Amazon Sagemaker is an AWS managed service which allows customers to prepare data, build, train and deploy ML models quickly without the worry of managing any infrastructure. Amazon Sagemaker offers features for each of these steps in finally deploying your models with ease and getting the best out of it with minimal effort:

### *Data Preparation*

Amazon Sagemaker offers tools for initial data preparation that allow you to build on top of it, such as [Data Wrangler](#). Data Wrangler can be added in ML workflows in [Amazon Sagemaker Studio](#) to import, prepare, transform, identify features and analyze data.

### *Build*

Customers can use Jupyter notebooks inside Amazon Sagemaker Studio to build their ML models. AWS offers its own custom built algorithms that run efficiently on AWS infrastructure as well as

popular open-source frameworks such as TensorFlow, PyTorch, etc. We can also write our custom code if the already available solutions don't fit our use case.

### *Train*

As Amazon Sagemaker manages all the infrastructure, we don't have to worry about training. We can use all the managed infrastructure from storage to compute.

### *Deploy*

Deployment of ML models is also taken care of by Amazon Sagemaker itself. There are multiple ways to deploy a model and a particular way can be selected based on the use case. We recommend using **real-time inference** for workloads with low latency requirements, **serverless inference** as a fully managed solution and for workloads that can tolerate cold start problems, **asynchronous inference** for workloads with large payload sizes (up to 1GB) and near real time processing requirements and **batch transforms** for processing entire datasets.

Amazon Sagemaker also offers support for A/B testing by enabling multiple ML models behind a single endpoint, A/B testing helps in figuring out how different models are performing and if they are working as per expectations.

Depending on our use case, we can decide to use Amazon Sagemaker for some or all of the above features in building and deploying our entire pipelines. For example, we can bring our already trained model on on-premise infrastructure and deploy it on Amazon Sagemaker.

Next we'll cover some other key features offered by Amazon Sagemaker:

- Amazon Sagemaker includes multiple features which help improve our ML models, such as [Amazon Sagemaker Clarify](#). As the name dictates, it detects any potential bias and helps clarify the predictions that ML models make using a feature attribution approach.
- Metrics are an important tool for visualizing performance. To visualize a ML model's performance, we can use [Amazon Sagemaker Debugger](#). It additionally helps to identify system bottlenecks for EC2 instance jobs with metrics such as CPU, GPU, GPU memory, network and Data I/O.
- We mentioned that Amazon Sagemaker helps with the entire cycle from data preparation to model deployments. As we scale our systems, it's very important that it is automatically managed to reduce operational overhead on our plate. [Amazon Sagemaker Pipelines](#) helps with this automation and building end-to-end CI/CD pipelines.
- The ML model performance can vary based on hyperparameter values. Amazon Sagemaker Automatic Model Tuning ([AMT](#)) runs the training job multiple times and figures out the best version of the model.
- It's important to train ML models on good quality datasets to get the most out of them. [Amazon Sagemaker Ground Truth](#) helps with automation of this process by creating high quality labeled datasets. For data labeling tasks, we can choose the workforce of our choice: from independent contractors, to our own private workforce, to vendor companies on AWS marketplace.

Amazon Sagemaker is a great tool for building ML solutions on AWS Cloud. We recommend going through the [AWS documentation](#) for an even deeper dive into its capabilities, as we couldn't possibly fit everything into this chapter.

There may be use cases where we don't have expertise to build our own solutions, in these scenarios we can leverage different services offered by AWS built to solve specific use cases. Let's explore some of these services in the next section.

## AWS ML Application Services

Building ML solutions on our own takes time as well as expertise in a field where we might not want to invest. If you have a general use case for ML, you can use fully managed services by AWS without the worry of building, deploying and maintaining ML models. Some of these services include:

### *Amazon CodeWhisperer*

You might be familiar with the concept of pair programming where two people have a discussion and then code together.

**Amazon CodeWhisperer** is your coding companion and helps you with code by generating code suggestions from small code snippets to entire functions. It also helps to flag the security issues in code and offers suggestions to remediate the raised concerns and can be easily integrated in your favorite code editor.

### *Amazon Comprehend*

**Amazon Comprehend** helps in gathering insights from a document or a collection of documents. Taking our example of an online food ordering application, customers write reviews on the ordered food and we can use Amazon Comprehend for figuring review behavior (positive/negative) or identifying food items being talked about. Amazon Comprehend can identify insights such as entities (person names, places, items, etc), PII information, document language, sentiment (positive, negative, neutral or mixed), and more.

### *Amazon Kendra*

**Amazon Kendra** is a ML powered search engine on top of structured or unstructured data repositories. It uses Natural Language Processing to determine a document most relevant to user's search queries. We use Amazon Kendra along with **Amazon Lex** to build **AI chatbots** for use cases such as customer support to resolve user queries on our online food ordering application.

### *Amazon Forecast*

**Amazon Forecast** is helpful for accurate time-series forecasts. Consider an example of an online business: how do you predict traffic for an upcoming sale based on historical traffic or what growth looks like if you launch in a new region. Amazon Forecast automatically figures out combinations of ML algorithms suitable for your dataset and helps you with the forecasts.

### *Amazon Rekognition*

**Amazon Rekognition** is a recognition service helpful in image and video analysis. This service can power use cases such as search over image and video content, face identification and verification, adult content detection, text extraction from images, etc.

### *Amazon Transcribe*

**Amazon Transcribe** is a speech to text conversion service. Consider an example where you offer a service where customers can submit their feedback via telephone for food items ordered on an online food ordering application. Amazon Transcribe can help derive valuable **insights** from such calls. Some other use cases where it can help is taking voice input and converting it to text for processing in your systems, converting audio and video content to be searchable by text, subtitles for any videos, etc.

These customized offerings help in direct integrations within our applications with minimal effort. To take an example, assume that

you own a blog hosting service where people can manage their blogs and share material with their followers. Now to add a functionality of text to audio, you can either build your own solution or you can use Amazon Polly for a much faster start. As we've mentioned before, we can always improve on the solutions in the future if our use case changes or we see any bottlenecks in existing solutions. The key point is time to market—if you start with a custom solution it might take much more time to launch the service as compared to direct integration with Amazon Polly. In the next section, we'll discuss special infrastructure support provided by AWS to build our ML workloads.

## AWS ML Infrastructure

AWS offers EC2 instances specialized to handle ML workloads both for training and inference. These instances include hardware based accelerators, also referred to as co-processors to enhance the computing power and perform tasks such as graphics processing, floating point number calculations, and data pattern matching in a much more efficient manner as compared to software running on general purpose CPUs. Some of the examples of accelerators are Graphical Processing Units ([GPUs](#)), Field Programmable Gate Arrays ([FPGAs](#)), AWS [Inferentia](#), and AWS [Trainium](#). Let's talk about the custom solutions offered by AWS to accelerate our ML workloads:

### *AWS Trainium*

The Amazon EC2 [Trainium](#) instances offer up to 50% cost-to-train savings over comparable Amazon EC2 instances. These instances are optimized to run [Deep Learning](#) (DL) training workloads with native support of different data types such as FP32, TF32, BF16, FP16, UINT8, and configurable FP8. It supports AWS Neuron SDK which is natively integrated with [PyTorch](#) and [Tensorflow](#) so as existing framework applications can be used with minimal code changes.

## *AWS Inferentia*

The Amazon EC2 **Inferentia** instances are designed to run ML inference applications with high throughput and low latency while saving costs as compared to general purpose Amazon EC2 instances. These types of instances offer high speed connectivity between the accelerators, enabling us to deploy billions of parameters across multiple accelerators on Amazon Inferentia EC2 instances.

### **NOTE**

Inference means reaching a conclusion based on logical reasoning and evidence, so ML inference essentially means running ML models on live data to get a final answer (or make a prediction). The training is a pre-phase to inference where we train ML applications to learn from existing data and then use this intelligence at time of inference.

The EC2 instances are purpose built to optimize compute heavy tasks such as running large scale models, natural language processing, speech recognition, and computer vision. You might consider the cost there—if the accelerator in these EC2 instances makes processing so fast then it would definitely be costly and you should consider if you can afford such an expenditure. As these instances are specifically designed for ML workloads, they are cost effective if you compare them to EC2 instances with similar capabilities. For example, **Amazon EC2 Inf1** instances provide 2.3x higher throughput and up to 70% lower cost per inference than comparable EC2 instances. The **Amazon EC2 Inf2 instances** are cheaper than Inf1, 4x higher throughput and 10x lower latency. A very good thing about using AWS Cloud is that AWS innovates on the behalf of customers. We as customers might be happy with the performance and cost benefits of Amazon EC2 Inf1 instances, but

AWS launched a new version with more improvement to existing options.

## Conclusion

This chapter introduced you to AWS services that are helpful in building and running Big Data analytics and Machine Learning workloads at any scale of your business use case. You'll notice that some services have similar features—such as when we compare Amazon MSK to SQS-SNS or to Kinesis Data Streams. Make sure you always have your requirements in hand when comparing services to make sure you're making the right selection. Your use case requirements are of utmost importance when selecting a service.

Many times you may want to use a combination of multiple services to serve your use case best, such as Amazon EMR to process the live streaming data published to Amazon MSK or have Amazon Redshift data plotted via Amazon Quicksight. As AWS provides easy integration with other services, it is a seamless experience to use any service and fit it into other service components to meet our requirements.

In the second part of this chapter, we looked into AWS ML offerings which essentially help to run our ML workloads with zero or minimal operational overhead. There are multiple services offered by AWS to solve specific use cases, and these offerings help us to utilize the ML capabilities without having vast knowledge of machine learning concepts. For example, think of building a language translation tool on your own vs directly using [Amazon Translate](#). Amazon Translate will reduce the time-to-market to launch your own application with translation capability.

We concluded our chapter with discussion around multiple infrastructure options that are specialized to run ML workloads at scale with higher throughput and lower cost as compared to general purpose comparable hardware. This is the final chapter in this section

of the book and we're confident that now we can use our understanding of the material covered in Parts 1 and 2 to build real time large scale systems. In Part 3, we'll walk through use cases for building systems on AWS Cloud, starting with a URL shortening service.

# **Part III. System Design Use Cases**

---

This unit covers some common system design use cases and examples, which we will work together to build and scale on top of AWS Cloud. For all the use cases, we'll start with base architecture (say, for a startup with 1000 customers) and try to modify and scale it to millions of customers, discussing the bottlenecks that might occur in design, and comparing AWS services along the way. Each use case will follow the format outlined below:

1. Understanding the problem
  - Background
  - Business use-case
2. Discussing the requirements
  - Technical Requirements
  - Scale Considerations
    - Storage
    - Bandwidth and Throughput
    - Latency
3. Building with the first principles
  - Concepts and Principles
  - A Rough System Design

#### 4. Starting with Day 0 architecture on AWS

- System Design
  - Diagram
  - Discussion
- Issues and Limitations

#### 5. Scaling to millions of users for Day 1 and beyond

- System Design
  - Diagram
  - Discussion
- Best Practices for the architecture

At the end of this unit, the readers will understand:

- The business use case around these system design problems and how to ask the right questions to evaluate the requirements.
- The technical non-functional requirements for the use cases and the scale requirements for storage, throughput and latency, which percolates to capacity and cost estimations.
- How to build the solution with the first principles using system design basics and how to handle the edge-cases and nuances around the design.
- How to start with a Day 0 architecture on AWS with an AWS architecture design diagram, which can start as small as a Minimum Viable Product and can scale up to thousands of customers.
- How to scale the architecture to millions of users for Day 1 and beyond on AWS, making it secure, high-performing, resilient,

and efficient.

The next ten chapters as part of this unit, will cover

1. Imagine a tool that takes long web addresses and turns them into short ones.

Chapter 14 will dive into the world of URL shorteners (a tool that takes long web addresses and turns them into short ones). We'll learn why they're important, what they need to do, and how to build them using AWS. We'll start with a simple setup for a startup, and then see how to make it work for millions of users.

2. Ever wondered how search engines like Google find information on the web?

Chapter 15 is all about web crawlers and search engines like Google and Bing. We'll explore how these systems work, what they need to do, and how to create them using AWS. We'll start small and then figure out how to handle a massive amount of data.

3. Ever wondered how social networks like Facebook connect people from all over the world?

Chapter 16 will cover how to design a social network like Facebook or Instagram and create a newsfeed system to connect people from all over the world. We'll cover how to make sure people can share and see updates, and how to build it all using AWS.

4. Online games are a blast, but have you ever thought about how the leaderboards work?

Chapter 17 will dive deep into designing a system for tracking online game scores and ranking players. We'll use AWS to make sure the leaderboards can handle lots of players and updates.

5. Making online payments safe and easy is crucial but how do you build such a safe transactional system?

Chapter 18 explores designing a payment system that securely handles transactions. We'll learn how to use AWS to build a reliable and secure payment gateway.

6. Ever wondered how chat apps like WhatsApp keep us connected no matter where we are?

Chapter 19 delves into designing a chat application that lets people send messages in real time. We'll see how AWS can help us create a seamless and responsive chat experience.

7. Streaming videos online on Youtube and Twitch is a big deal nowadays but how can you build such a system?

Chapter 20 will teach you how to design a system that processes and streams videos smoothly. We'll explore how to make sure viewers can watch videos without interruptions using AWS.

8. Ever wondered how stock exchanges keep the financial world moving?

Chapter 21 looks into designing a system for buying and selling stocks. You'll learn how to create a reliable and efficient stock exchange using AWS.

9. Imagine how travel experiences would be without ride hailing services like Uber and Lyft.

Chapter 22 will explore designing a system that matches riders with drivers. You'll see how AWS can help create a fast and reliable ride-hailing experience.

10. Ever wondered how websites like Amazon and Pinterest recommend things you might like?

Chapter 23 dives into designing a recommendation system. You'll learn how to use AWS to analyze data and suggest content that users might enjoy.

By the end of this unit, you'll have a clear understanding of how to tackle different real-world challenges using AWS Cloud. You'll know how to design systems that are efficient, reliable, and ready to handle lots of users. So, let's dive in and start designing such awesome systems!

# Chapter 11. Designing a URL Shortener Service

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 13th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

We explored the concept of DNS in Chapter 9, noting that it's easier to remember a website's URL than their IP addresses—but what about the long URLs? It's easy to remember the root part of the URL (for example, learning.oreilly.com in <https://learning.oreilly.com/library/view/learning-system-design/9781098146887/>) but you can easily forget the long URL. We often tend to embed **links** to text because that increases the readability. Another way of sharing long URLs is by shortening them to shorter URLs. For example, LinkedIn automatically shortens any URL that is part of a post because it helps in increasing post readability and user interaction. In microblogging applications such as Twitter (now known as X), there is a limit on characters for a single post , so to reduce the length of your text, you can shorten the URL via some URL shortener service such as [tinyurl](#) or [bitly](#) and

attach it to the post. This chapter explores the design of an URL shortener service and discusses how to deploy the system on AWS Cloud. We'll start our discussion with requirement gathering and expectations from the system and later jump to the fine details of the system.

## System Requirements

You should have a clear goal in mind of what problem you need to solve. From there, the next step to design any such system, big or small, is to gather the requirements. These requirements include:

- Why is this system needed? What is the business use case this system is solving?
- Who are the users of this system? How many users?
- Is there already a system that you can leverage instead of designing a new system from scratch? Systems like bitly expose URL shortening APIs to business customers so why should one build their own system instead of using bitly system APIs.
- Is the system's latency critical?

The requirements list can be huge and time consuming but it's a very important step. We had multiple discussions in Part I and Part II regarding how to compare two technologies and pick the best solution depending on your understanding of the business use case. It is not possible to weigh the pros and cons of your options if your requirements are not clearly defined. The requirements of any system are widely specified in two categories— functional requirements (FR) and non-functional requirements (NFR).

## **Functional and Non-Functional Requirements**

Functional requirements mean functionalities or features offered by the system to the end users. The main expectations from a URL shortener service are simple:

- Systems takes input as a long URL and returns a shortened URL.
- The short URL should redirect to the long URL when accessed by any user.

Along with the most critical requirements described above, you can also consider value add requirements which can be nice to have from a customer's perspective or helpful in deriving business value. Some of these include:

- Custom URL creation support.
- Analytics on the URL access patterns such as most popular URLs.
- Expiry for a URL so that the URL is auto expired and is no longer accessible after a fixed period of time.
- The application should be developed as plugin based architecture ensuring extensibility of the architecture. The system has capability to expose APIs to the third party clients to integrate their applications with our system to generate short URLs.

We should always gather as many requirements as possible about what might come in the future, but we shouldn't get so caught up thinking about future expectations that we lose track of current expectations. The additional requirements will help to design a system in an extensible way so that new features can be added without needing to re-architect the system.

Another type of requirements are non-functional requirements, also called NFR. NFR determines constraints the system operates on and don't directly impact the user feature wise, but are important as they ensure the quality of system operations. To take a few examples:

- Security, to ensure the system is not exploited by bad actors.
- High availability of the system, to ensure a high uptime percentage in a year.
- Observability, to ensure appropriate metrics/alerts are in place for constant monitoring of system health.
- Low latency for short URL creation and redirection.
- Datastores should be durable and ensure correctness of the data for the expiry time configured for an URL. The data should reside in the system until the expiry time or explicitly removed by a user.
- Fault tolerance in the system via mechanisms such as retry handling.
- Interoperability in the system architecture, as the system operates at high scale and can be broken down into multiple subsystems. How will different subsystems interact with each other?

These requirements will help in making better decisions while designing the system. Another aspect of the requirement gathering step is identifying the scale the system will operate on.

## **System Scale**

Scale refers to the number of users or the traffic we're expecting on the system. We should be able to answer questions such as:

- What is the potential number of user requests per second to generate a short URL?
- How many user requests for short URL to long URL redirection?
- Do we've any idea around the amount of storage we require to store the data?, etc.

Coming up with scale numbers can be tricky if you're starting out fresh with any system and you might not have 100% accurate data to come up with these numbers. In these scenarios, it's ok to make a fair assumption with a balanced thinking of not too little and not too much.

We recommend following the **Make it Work, Make it Right, Make it Fast** principle and all the chapters in Part III follow the same philosophy. It's good to be optimistic that one day our system will serve one billion active users but you don't have to build the system from day one to support this scale. The goal of huge scale should never hold you back from launching the product—just make sure that the system is extensible enough, so if there is a need in future, it can be evolved as new users are on-boarded and new features are introduced. For example, [Figure 11-1](#) shows the initial launch of the amazon.com website—there was no point back then thinking of making it work for a billion customers.



# Welcome to Amazon.com Books!

*One million titles,*  
*consistently low prices.*

(If you explore just one thing, make it our personal notification service. We think it's very cool!)

## **SPOTLIGHT! -- AUGUST 16TH**

These are the books we love, offered at Amazon.com low prices. The spotlight moves **EVERY** day so please come often.

## **ONE MILLION TITLES**

Search Amazon.com's [million title catalog](#) by author, subject, title, keyword, and more... Or take a look at the [books we recommend](#) in over 20 categories... Check out our [customer reviews](#) and the [award winners](#) from the Hugo and Nebula to the Pulitzer and Nobel... and [bestsellers](#) are 30% off the publishers list...

## **EYES & EDITORS, A PERSONAL NOTIFICATION SERVICE**

Like to know when that book you want comes out in paperback or when your favorite author releases a new title? Eyes, our tireless, automated search agent, will send you mail. Meanwhile, our human editors are busy previewing galleys and reading advance reviews. They can let you know when especially wonderful works are published in particular genres or subject areas. Come in, [meet Eyes](#), and have it all explained.

## **YOUR ACCOUNT**

Check the status of your orders or change the email address and password you have on file with us. Please note that you **do not** need an account to use the store. The first time you place an order, you will be given the opportunity to create an account.

*Figure 11-1. Amazon.com initial website launch from 1995*

### **NOTE**

Some other reasons for redesigning any system could be the cost and operational maintenance such as when Uber started with DynamoDB to build a ledger store but later on moved to their own custom build storage solution, [DocStore](#).

The same principle applies while we figure out components in any system, the goal should always be to get the system out in the

hands of the public and gather feedback. There is no win in over-optimizing the system if we never reach the goal of solving the customer problems and finally generating revenue from it; after all, everyone is in business to earn money from the designed system or help other people to do so (as with open source projects).

Coming back to the scale numbers our system should support, let's move ahead with the following assumptions about our requirements:

Generate short URL from long URL - 1k requests/second

Short URL to long URL redirection - 20k requests/second

Average duration of URL persistence in system - 1 year

There are a lot of database solutions present in the market and in order to decide which one is best for our use case, there's another important parameter we need to consider: total storage space required. Knowing this can help with answers to the questions such as

- What is the expected cost we'll bear if we use Amazon DynamoDB or Amazon Aurora?
- Do we require data partitioning from the start?

Other things important for scale could be figuring out an instance type and number of instances for applications deployed on Amazon EC2 instances or memory requirements for AWS Lambda. As we pointed out earlier, it's really hard to get the real numbers before putting the system in production and serving the actual consumer traffic. We recommend making fair assumptions and then improving further based on learnings. One key consideration is system load testing; Load testing helps in gaining confidence that the system is able to perform in the expected fashion (system is reliable) with the increased traffic demands. Let's discuss the storage space requirements next.

## Storage Space

The actual storage required will vary depending on the type of data storage solution we use, because the data storage and access patterns are different for each database. For an approximate calculation, the storage space will depend on the number of URLs, length of short URL & long URL, user metadata and URL metadata such as expiry time, creation time, etc. The number of URLs generated in a year can be derived from assumed traffic of 1000 requests/second.

Total URLs in 1 year = (1000 requests/second) \* 60 (per minute) \* 60 (per hour) \* 24 (per day) \* 365 (per year) = ~31.53 Billion

Before moving further along on storage calculation, we need to figure out the ideal length of a short URL. The length should be such that the system doesn't run out of unique short URLs and should be able to support the required scale of 31.5 Billion unique URLs per year. In Chapter 9, we discussed the IPv6 addresses as a replacement for IPv4 addresses because the world is running out of unique IP addresses. We definitely don't want this kind of problem to occur in our URL shortener system, so the length of the URL should be calculated with proper considerations from the beginning. You might suggest a length of 12 or 15 to be on the safe side, but that defeats the purpose of shortening the URL.

We can think about generating a short URL from the numbers 0-9, as well as a-z and A-Z alphanumeric characters. What is the maximum number of unique URLs generated for a given length with these 62 unique characters?

Length(1) =  $62^1 = 62$   
Length(6) =  $62^6 = 56.8$  Billion  
Length(7) =  $62^7 = 3.5$  Trillion  
Length(8) =  $62^8 = 218.3$  Trillion

With reference to the calculations above, the URL system can support the required scale with URL length greater than or equal to 6. Considering the future scale, we can finalize URL length as 7, now

let's go back to our original question: what is the required storage space? Considering a simple database schema of storing short URLs, long URLs, expiry time and metadata, we can calculate our storage needs as:

Short URL(7 characters) = 7 bytes  
Average long URL(100 characters) = 100 bytes  
Expiration date(long) = 8 bytes  
Average Metadata(userIP, userPreferences, etc.) = 1 KB  
Total = ~1150 Bytes.

For simplicity, let's take 1 KB as storage requirement for a single URL.

Total storage for 1 year = 31.53 Billion \* 1 KB = 29.37 TB

In addition to the above, the system has other storage requirements as well such as:

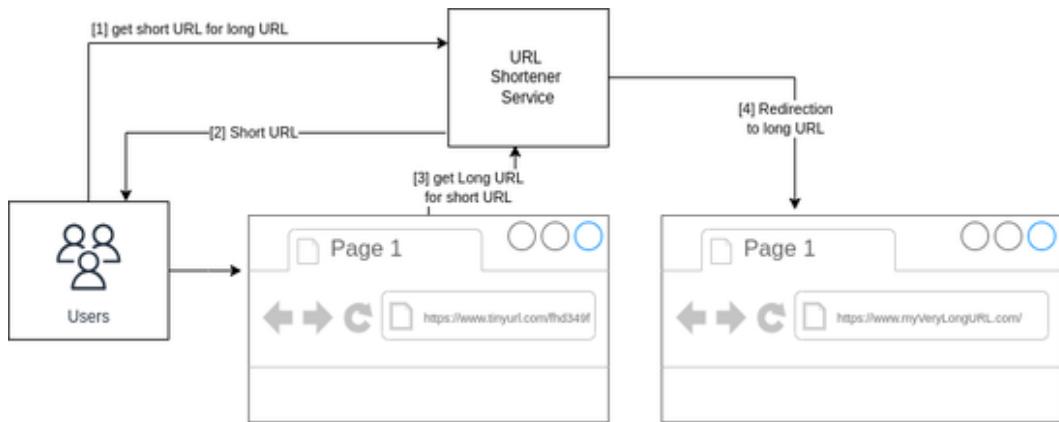
- Maintain cache to improve query performance.
- Analytics data.
- User authentication database. This can be required if the system offers additional capabilities to logged in users such as custom URL creation, view analytics, etc.

With all these things in our mind, let's move forward with the design of URL shortener service.

## Starting with the Design

We mentioned in the requirement gathering section about not thinking too far ahead because we don't know what our system will look like in the future or what new features we might introduce to evolve with the market. We should try to keep our systems open ended so that we can introduce new features with ease when they are actually needed. [Figure 11-2](#) shows the most basic architecture

and user interactions with the URL shortener service to meet the defined functional requirements. The users create a short URL corresponding to a long URL utilizing the URL shortener service and then use the same short URL in the web browser (or terminal) to access the URL. The URL shortener service figures out the long URL corresponding to the short URL and redirects the request.



*Figure 11-2. User interaction with URL Shortener Service*

The first thing in the design we need to figure out is how a short URL can be generated from a long URL so as it can be optimally stored supporting faster user lookup queries.

## URL Shortening Algorithm

Hashing seems to be a viable solution to solve this problem, using a hash function such as **MD5** which takes a long URL as input and returns a hash of this long URL as output. The length of the generated hash can be greater than 7, so the system should trim the generated hash to a reduced length. This solution is easy to implement but it has one problem— collision. Two different long URLs have a possibility of generating the same hash and even if the hash is different, the truncated length could be the same. At large scale, this can become a frequent problem, making short URL generation operation a bottleneck.

There can be different methods of collision handling—one possible solution is to take some other set of characters instead of truncating to the first 7 characters. The hash generation approach also requires a data store look up (multiple lookups in case of collisions) which adds to extra latency of the operation. In short, the hashing solution is not so perfect for our system requirements. Let's think about an option that would avoid the collisions and perhaps also avoid the database lookups in order to make operation faster.

A key point to consider here is the long URL and short URL don't have to be related—this would essentially be a unique Id generation system which can map the generated Id to a long URL. Let's discuss a few approaches for generation of unique Id. The Id can be as simple as a counter incremented every time a new request is received from the users or an **AUTO\_INCREMENT** like feature from a database. In case the system itself (without involvement of the database) handles the id generation, we're sure of solving both the problems of collision handling and avoiding database lookups. Think of it as a server initializing a counter value with "1" and then increment it every time a new request arrives. The mapping can be stored to a database and then the short URL is returned to the users as shown in **Figure 11-3**. The sequence of alphanumeric characters from 0-9, a-z and A-Z when used for encoding is referred to as **Base62** encoding.

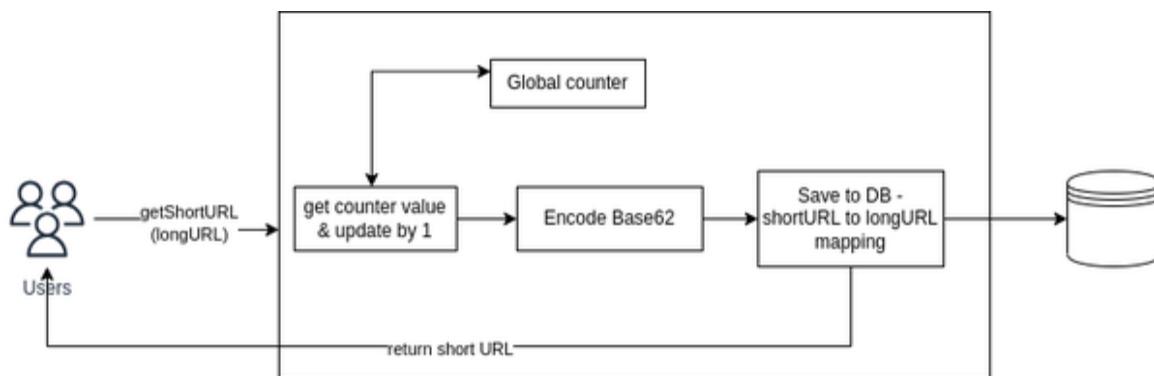


Figure 11-3. Short URL generation by maintaining a counter

Please note that by using Base62 encoding on the counter values (decimal numbers), the short URL will be unique but not always of length 7. For example:

```
Base62 (1) = 1  
Base62 (10) = A  
Base62 (61) = z  
Base62 (62) = 10  
Base62 (61) = 11  
Base62 (10000001) = 4C93
```

To ensure the length is always 7, we can append random characters to the generated Base62 value from the decimal counter value. We haven't decided on the maintenance of counter value. The simple solution is to maintain a global counter variable in our application which is updated after every new short URL generation. But there can be multiple machines on which the application is running which can lead to a duplicate counter value, and even if there is a single machine, it poses a risk of going down, which would make the counter value again start from zero.

To solve this, we can consider maintaining this global counter in a database. This way it solves the problem of a single machine going down (counter is not initialized to zero every time) or multiple machines being responsible for URL generation (duplicate counter values across the machines). However, it can also cause issues because we can't allow multiple application threads to access the counter at the same time in order to avoid race conditions and if we apply a lock (a single machine reads and updates the value), it will increase operation latency. Let's simplify the entire architecture further to ensure optimal application performance. We can assign additional responsibility to the system of pre-generating the short URLs since they are not dependent on the long URL. This way the multiple machines can utilize the pre-generated Ids (short URLs) solving all the bottlenecks.

Our URL shortener service consists of an additional component responsible for pre-generation of short URLs and keeping them in

memory for faster access. An important decision to make here is whether we give this additional responsibility to our URL shortener service or host it as an independent service. We discussed monolith and microservices architecture in Chapter 7 and the benefits associated with them. A single component will ensure comparatively lesser hardware cost (at least in the beginning), as there's no requirement of network calls between the services. What you need to consider in order to make the decision here is what value you're getting out if this is an independent component instead of hosted as part of the main application. A separate id generation or key generation service (KGS) can be beneficial for other use cases as well and this is independent responsibility in itself doesn't have to be tied to URL shortener service. Given this, we'll go with the idea to host a separate service with the responsibility to generate unique keys and pass those onto our URL shortener service when requested. Let's dive into that next.

## **Key Generation Service**

KGS has a simple responsibility of providing unique Ids when requested by our URL shortener service. To ensure traffic control on KGS as well as make the URL shortener system more efficient, the operation can offer a list of Ids, say 1000 in single operation, and the URL shortener service can re-request once these Ids are exhausted. Now let's dig a little deeper into the working of KGS—how should these unique Ids be generated?

We can consider the database auto increment feature such as MySQL [auto\\_increment](#) or PostgreSQL [serial/sequence](#). This functionality is applicable for a leader instance (if multiple database instances) and it can become a single point of failure for maintenance of Ids, hampering the overall system availability. Further, implementing this feature becomes very hard to do in the case of multi-instance database setup such as horizontally scalable NoSQL databases. We can take inspiration from [Flickr](#) architecture

which uses MySQL auto-increment feature with REPLACE INTO query to get a globally unique new id on every new query as shown in code snippet below.

```
CREATE TABLE `Tickets64` (
  `id` bigint(20) unsigned NOT NULL auto_increment,
  `stub` char(1) NOT NULL default '',
  PRIMARY KEY (`id`),
  UNIQUE KEY `stub` (`stub`)
) ENGINE=InnoDB
```

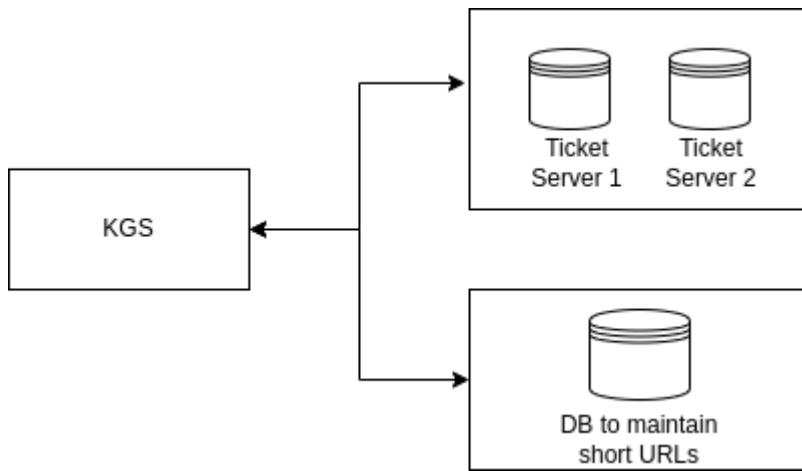
```
REPLACE INTO Tickets64 (stub) VALUES ('a');
SELECT LAST_INSERT_ID();
```

To increase system availability and avoid a single point of failure, two database servers are used starting with an even and odd number and offset as two to avoid collision as reflected in code snippet below.

```
TicketServer1:
auto-increment-increment = 2
auto-increment-offset = 1
```

```
TicketServer2:
auto-increment-increment = 2
auto-increment-offset = 2
```

Further regular snapshots should be taken of these two databases as well to ensure backup is always kept in case of any fatal scenarios. As only a single row is maintained within a table, we can be sure that the server will not run out of storage space. The Id generation process speed will be dependent on database operation, so KGS should pre-generate the short URLs and maintain used and unused short URLs in another database, as shown in [Figure 11-4](#).



*Figure 11-4. Unique Id generation via Ticket Servers*

The two ticket servers generate even and odd Ids respectively but this management overhead will increase if the ticket servers are deployed in multiple regions instead of just one. One potential solution to ensure uniqueness across regions could be adding a datacenter prefix to the short URL such as a 1 for us-east-1, 2 for eu-west-1 and so on, making the URL make it 8 characters instead of 7 such as a 1 for us-east-1, 2 for eu-west-1 and so on.

### NOTE

In 2010, Twitter proposed another approach for generation of 64-bit unique Ids, referred to as **Snowflake**. **Snowflake** can generate 4096 unique Ids per millisecond with a combination of timestamp (41 bits for epoch milliseconds timestamp), machine identifier number (10 bits giving us up to 1024 machines) and sequence number (12 bits for local counter per machine). The remaining one bit is a signed-bit always set to 0. The Snowflake approach will not directly work for our URL Shortener because the system always requires a 7 characters length string but implementation can be built using a similar concept. The main reason for Twitter to use this approach was the requirement for sorting and ensuring ordering guarantees. The URL shortener system has no such requirement of creating Ids based on timestamp so we're free to choose the approach which is easy to implement and maintain the uniqueness.

Let's quickly talk about database selection for KGS. KGS needs to maintain unique Ids which can be sent as part of response to the

URL shortener service when requested. The KGS database should also ensure that it returns unused Ids on every new call and this information should be maintained in the database. Now the system has three options:

- Maintain two tables for unused and used keys. The server takes a list of keys from the unused table, updates the used table and returns a response. The maintenance of two tables for the same data offers additional challenges of data synchronization; It can be solved by using transactions across the tables via default database support or ensuring it via application code.
- Maintain a single table for both unused and used keys but maintain a distinction by keeping a boolean value.
- The table only keeps unused keys and deletes the keys from the database once they are sent to URL shortener service. This option has the least complexity, we don't have any use case to lookup used keys but we recommend the system to still maintain the used keys for any use case that might come in future to support. You can consider this solution if the used keys are stored in some archival store and it can be better than the first approach in cost terms.

Additionally, KGS should keep some Ids in the buffer to serve the queries faster—but how many Ids should be in the buffer? A simple option is to generate all the short URLs in advance before we even launch the application, but we don't recommend this. We suggest maintaining a balance—create unique Ids if they are needed, keeping some Ids in advance depending on application traffic. Let's now discuss the APIs that need to be supported by the URL Shortener service and KGS.

## APIs

APIs help clients integrate with the system to make use of supported functionalities. The URL shortener service should support two main APIs: creating a short URL and retrieving a long URL back from a short URL. The below code snippet shows an API signature for creation of short URLs.

```
POST /v1/createShortUrl
{
    longUrl,
    customUrl,
    expiry,
    userMetadata
}
```

The longUrl is a mandatory parameter in API requests for generating a short URL. The additional attributes are added to support other functionalities of the application. These include:

- customUrl parameter is helpful if the user wishes to generate a custom URL instead of a random 7 characters length string. For handling this parameter, the URL shortener service will directly check in the database if the custom URL is already created and if not, the mapping of custom URL to long URL will be stored in the database.
- expiry enables users to specify custom expiry on the URL instead of what is being defined by the system.
- userMetadata helps to gather extra information about the user such as IP address, geographical location, browser, etc helping in driving analytics.

## NOTE

It's possible that the same custom URL will be created by multiple users. Amazon DynamoDB's default PutItem behavior is to update the item if the primary key already exists, but this is not a valid option for the URL shortener system. The PutItem request should contain **ConditionExpression** to fail the request for the same custom URL request.

The response of the API can be either success or failure. You might remember our discussion of HTTP status codes from Chapter 6—the status codes are helpful in determining the kind of HTTP response the client has received from the server. We should define proper mechanisms for both of them so that the clients can understand the response in failure scenarios and take any further actions such as retry. The failures can be due to invalid parameters from the client side or some issue at the backend application. Below are two example response scenarios:

```
HTTP/1.1 200 OK
{
    "shortUrl": "https://www.tinyurl.com/dfjdf47"
}
```

```
HTTP/1.1 500 Internal Server Error
{
    "error": "Please try again after some time"
}
```

As a user creates a short URL and shares with other people or accesses it themselves, the API request and response could look something like below:

```
GET v1/getLongUrl
{
    shortUrl,
    userMetadata
}
HTTP/1.1 302 Found
Location: https://www.example.com/my/long/url

HTTP/1.1 404 Not Found
```

```
{  
    "error": "short url doesn't exist"  
}
```

Let's discuss any additional considerations before moving to the AWS components being used to deploy the system in production.

## System Considerations

By this point we've proposed two services in the architecture: the URL shortener service and KGS for supporting end to end functionality. The URL shortener service is responsible for short URL generation as well as redirection. The APIs are simple enough to be supported by a single system but the traffic patterns are quite different for both of these APIs; the traffic for URL redirection will be way higher as compared to URL generation.

Think of a celebrity generating a short URL and sharing the URL with followers on social media platforms—all the users will try to access the URL, increasing the overall system traffic. In these kinds of scenarios, we recommend starting with a single system and then evolving if traffic handling becomes a problem in a single system. Separating a system into two components can come with extra overhead, such as a database being accessed from two separate services. The URL shortener service will have both read and write use cases whereas the URL redirect service will have read use cases on the database.

It is not generally a good idea for two services to directly interact with a single database; the most robust solution is to expose the database operations via another service, responsible for handling all database operations along with cache maintenance.

We'll also need to finalize database choice, given a wide variety of database solutions available in the market. As per the requirements, the system should essentially maintain a mapping between short URL and long URL to serve the URL redirect queries. This use case

can be perfectly solved by using a key value database such as Amazon DynamoDB. There are multiple factors involved in the selection of a perfect database for a specific use case and we explored this idea in Chapters 2, 3 and 10. There are many benefits of DDB that we get out of the box, such as horizontal scaling, infrastructure management with no worry of scaling up read/write replicas, time-to-live configuration (TTL), and DDB streams for analytics..

Similarly, we can use DDB for KGS as well to maintain the generated Ids. DDB is a fully managed service by AWS and we as customers don't have to worry about maintaining the storage infrastructure. You should list down the read and write query patterns to finalize on the DDB schema.

We've discussed the individual components and how the system will operate to serve the requirements. However, the business aspects of the application are important as well, and eventually we want to earn money from our systems. Some examples for revenue generation could be offering URL analytics or a custom URL domain for the paid users. Analytics in itself is a big portion of the URL shortener system and the entire end-to-end analytics pipeline is not in the scope of this book. However, in the next section we'll shed some light on how custom domain support can be built in the system.

## **Custom Domain Support**

One of the extended requirements of the system was to build an extensible system which is able to expose the APIs to third party clients. Assume that Google Drive leverages bitly services to shorten the URL but Google doesn't want to share bitly short URLs to the users but rather a custom Google short URL.

Enabling custom domain support in the system is somewhat similar to maintaining multiple URL shortener systems, each with different

domain names. Now the important point to consider here is whether the system has virtual separation or a physical separation. Physical separation means deploying multiple systems in parallel, each serving a specific domain, and virtual separation means the same system catering to all the clients with internal code separation or configurations.

Let's try to solve this via a single DDB table which maintains the short URL to long URL mappings. DDB requires partition key and sort key as part of primary key, so the system defines:

```
Partition Key = short URL  
Sort Key = tenant Id
```

The tenant Id is an identification useful in figuring out different tenants (systems onboarded to URL shortener system).

- The short URL is created via an API call by the onboarded system so they are expected to pass the tenant Id as part of the request. The tenant Id can be created when the system is onboarded or this can also be figured out based on credentials used by the system in the API call.
- On get call as well, tenant Id can be derived based on the domain being accessed.
- The key in the cache store should be created with a combination of short URL and tenant Id for easy retrievals.

We can definitely create different tables per tenant but a careful call should be taken based on the number of onboarded systems and their scale of operations. We can take a hybrid approach as well, often referred to as cell based architecture. Each cell is responsible to serve a set of onboarded tenants and the system maintains the mapping on which cell is responsible to serve a particular tenant and they can operate independently because they don't share the data between them. There are also other things to be taken care of in this architecture like load balancing between the cells, one cell might

just onboard a single tenant but another cell might include 10 tenants depending on the scale requirements.

We discussed multiple components and considerations for the URL shortener system, now let's combine the components to work together and discuss the launch of the product on AWS Cloud.

## Launching the System on AWS

The entire system design is shown in [Figure 11-5](#). One additional component in the design which we haven't yet discussed is the analytics pipeline. The URL shortener service publishes the event of user activity to the queue and the analytics pipeline takes care of analyzing and offering insights on data. For creation of URLs, users will not create short URLs directly—rather there will be a frontend application gathering the customer request and then forwarding to the URL shortener service for URL generation.

### NOTE

To simplify the diagrams, the services introduced in our system design diagrams may not include a load balancer or API gateway to front the application. You can always assume that the service includes a load balancer or any similar component to distribute load into multiple instances of the application.

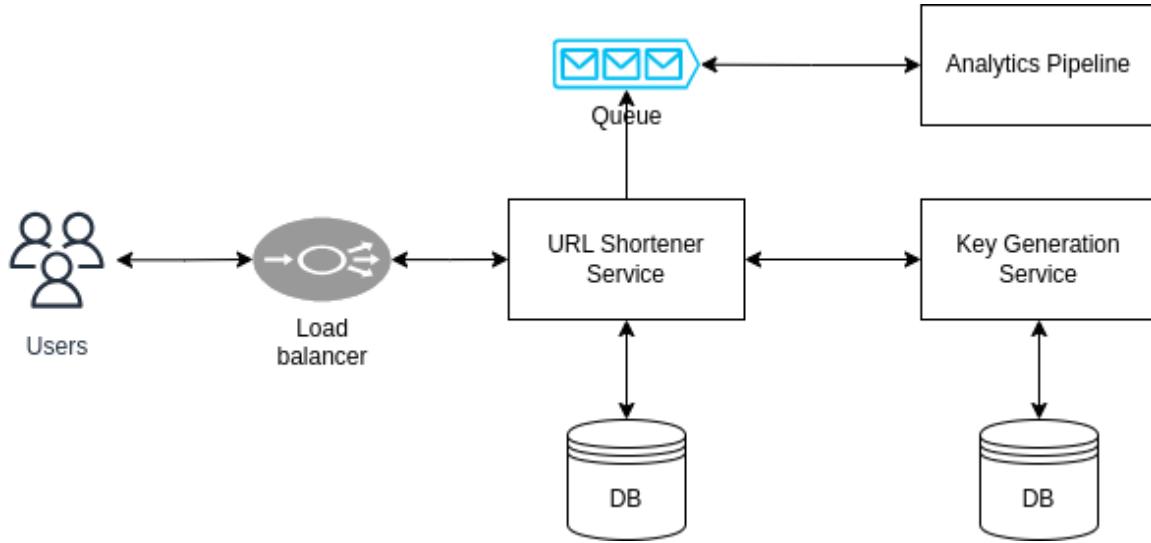


Figure 11-5. URL shortener system architecture

Now let's deploy this architecture on AWS cloud starting from the initial launch.

## Day Zero Architecture

Keeping the **Make it Work, Make it Right, Make it Fast** principle in mind, the day zero architecture could look something like [Figure 11-6](#).

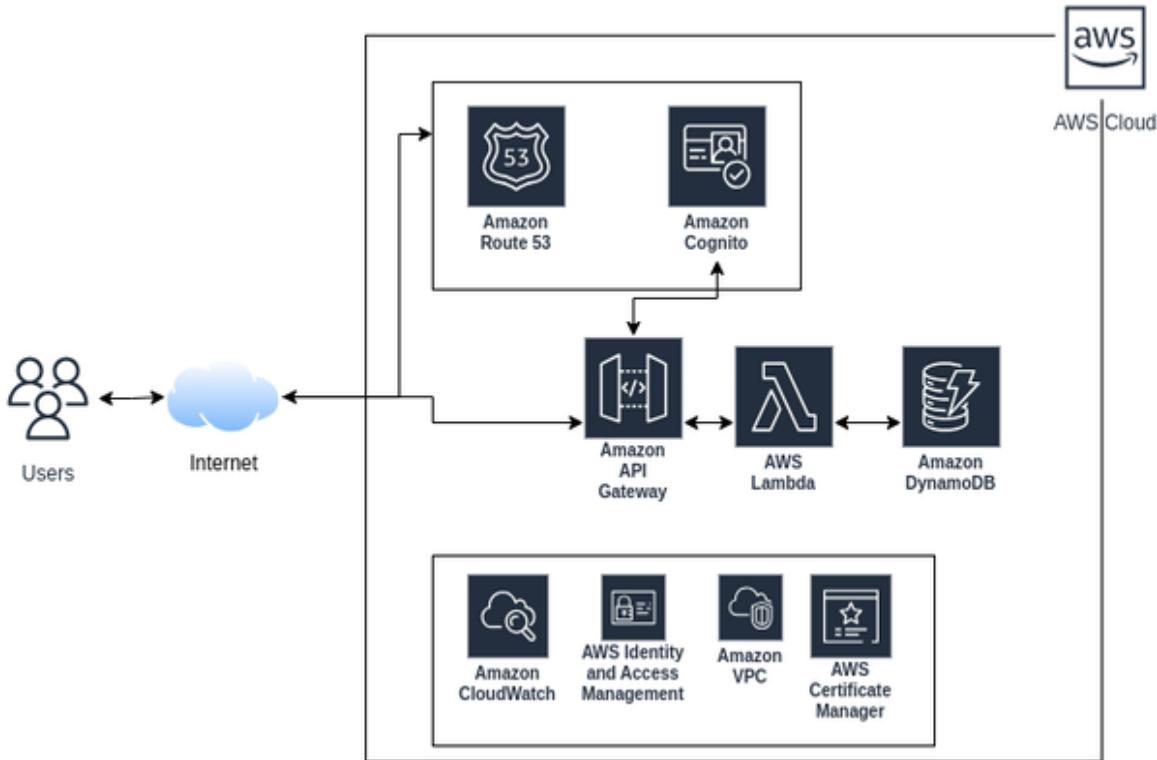


Figure 11-6. URL shortener service on day zero

As we're proposing to use AWS Lambda for operations on the URL, we can choose to deploy separate lambdas based on APIs or a single lambda handling all the operations. The diagram includes some additional services such as Amazon CloudWatch for monitoring, Amazon Cognito and AWS IAM for authentication and authorization, Amazon VPC to launch the resources and AWS Certificate Manager to SSL/TLS certificate management. Another approach for application deployment can be usage of **AWS App Runner** service to fully manage deployment of containerised web applications or backend API services as shown in [Figure 11-7](#). AWS App Runner launches all the resources in the VPC managed by AWS itself.

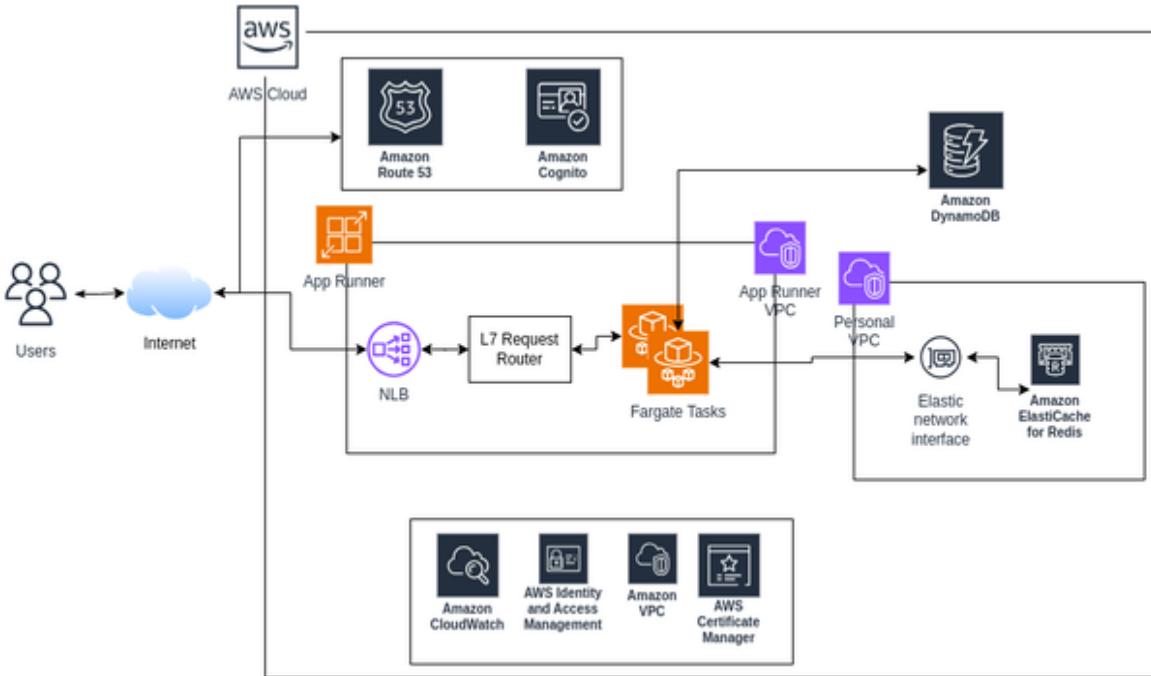


Figure 11-7. Day zero architecture with AWS App Runner

AWS App Runner abstracts all the components and customers just need to focus on application development. It is built with a combination of ECS Fargate, auto scaling, ELB and ECR and supports Python, Node.js, Java, Go, Rails, PHP and .Net as language runtime. Customers are required to specify a GitHub source code repository or an ECR image and App Runner takes all the responsibility of application deployment as ECS Fargate tasks. App Runner [CreateService API](#) returns a secure URL which can be used to access the service APIs.

Looking at the overall architecture, the user request is routed to NLB after domain resolution via Route 53 and then forwarded to Application Layer L7 request router which further redirects to ECS Fargate tasks. We additionally added Amazon ElastiCache in the diagram to showcase the **connectivity** between AWS App Runner VPC and customer managed VPC. AWS App Runner can also be configured to accept traffic via customer managed personal VPC by using AWS Private Link endpoint.

There is always a possibility that the day zero architecture system faces some bottlenecks as it scales to serve more user traffic and features, so let's try to dig deeper into the architecture.

## Scaling to Millions and Beyond

With our day zero architecture, we try to launch the product without planning too far ahead and choose the technologies we're most familiar with instead of trying out something very new which will include a learning curve. In short, time to market should be minimum. Let's consider an example of a compute platform. There are multiple options such as AWS Lambda, Amazon EC2, Amazon ECS, Amazon EKS. If you're most comfortable with Amazon EKS and have prior experience with it, choose that option as a compute platform to start with.

We recommend setting proper metrics and alarms to identify any bottlenecks via tools such as Amazon CloudWatch and [AWS X-Ray](#). AWS X-Ray service helps in request tracing in distributed system architecture and figuring out which specific component is taking more time in processing. As we identify specific components causing issues in the overall system, we can dig deeper to figure out a resolution. We'll dig deeper into the architectures in [Figure 11-6](#) and [Figure 11-7](#) to identify any issues and address them to ensure the system's high availability.

## Storage Layer

The system is using Amazon DynamoDB as a data store to maintain the URL mappings. We don't have to worry about the scaling capabilities of DDB as it is managed completely by AWS. To begin with, we can utilize on-demand capacity mode and as we figure out traffic patterns, provisioned mode with auto scaling can be used to reduce costs. The application can be launched in multiple AWS regions and DDB global tables can be used to ensure data is replicated across the regions. A key consideration on global tables is

approximately **one second** of replication lag between two or more regions, so it might happen that a short URL is not available in another region as soon as it is created.

For URLs generated by popular users, the read traffic can be really huge so there is no point in serving all read queries directly from the database. DDB guarantees single digit millisecond latency on direct key operations but to make it even more seamless and bring down latency to microseconds, we can add a caching layer on top of DDB such as Amazon ElastiCache or Amazon DynamoDB Accelerator (DAX) to improve read performance.

An important key consideration while you're designing any system with DDB as your storage option is justifying the requirement for caching with DAX or ElastiCache—which one should you use? The ideal choice may be DAX as it is specifically built for DDB to improve query read performance, but Amazon ElastiCache might be a better choice if the queries are already being served by it for any other use case or you're more familiar with the technology. This saves integration effort into newer technology, saving implementation bandwidth. Further it is always a good idea to perform benchmarking tests to compare two technologies to support a technical decision because in the end, we need a combination of low latency and a cost efficient solution.

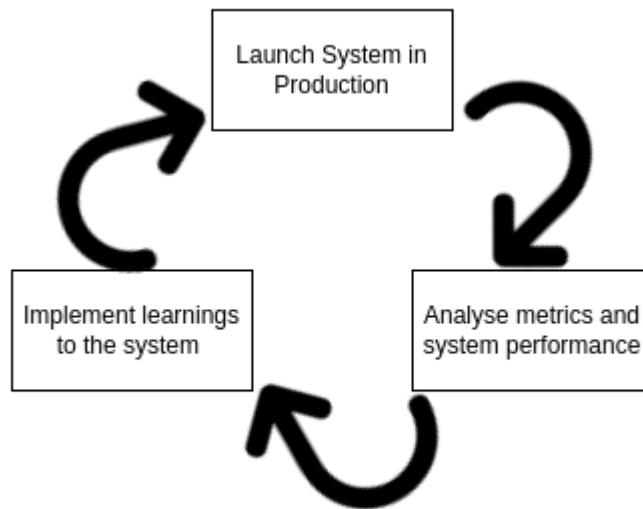
We'll focus next on the compute layer from the day zero architecture.

## Compute Layer

We preferred AWS Lambda in [Figure 11-6](#) and AWS App Runner in [Figure 11-7](#) to avoid the setup of Amazon EC2 and avoid operational burden, but can Lambda or App Runner scale to millions of customers? AWS Lambda automatically scales as per traffic requirements without any customer intervention. However, the problem which we might face is p100 latency which can increase from time to time due to cold start issues. We discussed a couple of

solutions to overcome the cold start problem in Chapter 11, such as provisioned concurrency which can help to address the latency bottleneck.

AWS App Runner allows maximum 25 instances per service and maximum 200 concurrent requests per instance, leading to 5000 concurrent requests per service. We recommend following the [documentation](#) on AWS limits for updated information on service quotas. The number of requests limit can become a bottleneck as per our scale considerations in the System Scale section. A point of consideration here is the limits can be increased in the future by AWS and this should limit us in launching the first version of the product. AWS Cloud offerings or open source solutions evolve and cloud providers innovate on the behalf of their customers, so in the future it is possible that AWS App Runner will be able to support much higher scale with lower cost. Remember that no architecture is considered the final architecture and we can always reevaluate and reiterate in case of any bottlenecks as per the software lifecycle in [Figure 11-8](#).



*Figure 11-8. Software lifecycle*

Here are few things to consider to scale the compute part of your system:

- Consider migrating to unmanaged versions of services, such as moving to Amazon ECS EC2 from Amazon ECS Fargate to have more control over hardware configurations.
- The system will have auto scaling in place to handle any increased load during peak times. The ASGs should be in different AZs to withstand AZ downtimes. In the case of managed systems like Amazon ECS Fargate, AWS takes care of deploying the system into multiple AZs to ensure availability. We recommend taking this point into consideration if you're managing the systems.

Let's incorporate all these concepts and present the final architecture on AWS.

## **Day N Architecture**

We're using a single AWS account to host all the services in the architecture—this can be evolved further as necessary in the future to host services in different AWS accounts or to take any other approach of account separation as we discussed in the 'Getting Started with AWS' section of Chapter 9. Please note that using multiple AWS accounts can increase data flow cost across the services and shared databases can become difficult to manage, though it can make the responsibility of each account simpler with clear boundaries.

Expanding the discussion from Chapter 9, another degree of isolation is at the VPC level. If all the applications are created in the same AWS account, are they present in the same VPC or different VPC? Further, if in the same VPC, are they hosted in the same subnet or different subnets? The resources should definitely be launched in different subnets (even for a single application) to

ensure high availability across the AZs. Launching all applications in a single VPC helps in reducing network costs, but needs extra care with managing the security of inter application communications. The rules can be managed by NACLs, and we recommend maintaining a limited number of rules. The rules are evaluated one by one and a large number of rules can compromise network efficiency by taking more time in evaluation (as compared to all rules being evaluated in parallel).

Our URL shortener system architecture can include five microservices for serving simplistic functional requirements:

- FrontendService (FES) responsible for receiving the customer traffic and redirecting to specific service VPC as per the operation.
- URLCreatorService (UCS) responsible for creation of URLs.
- URLReaderService (URS) responsible for URL redirection from short URL to long URL.
- KeyGenerationService (KGS) responsible for generation of unique keys to be used as short URLs.
- DataManagementService (DMS) is a thin layer of CRUD APIs between databases and services in the architecture. DMS is introduced in the architecture because the DDB and ElastiCache are accessible to two systems.

We are looking at a huge scale so we recommend deploying the applications in separate VPCs for proper isolation. Another architecture pattern followed by organizations is deploying multiple microservices on the same EKS cluster (in the same VPC). This makes sense at limited scale but again in context of day N architecture, we recommend using **separate EKS clusters** for service deployments for complete resource isolation (please revisit chapter 7 for kubernetes architecture).

## NOTE

It is not always true that the AWS cloud components can introduce bottlenecks with increasing scale—it could also be the application code introducing bugs and bottlenecks in the system. For example, the DDB key schema worked well on day zero but is not performing very well with increased traffic. We can use tools such as [Amazon CodeGuru Reviewer](#) and [Amazon DevOps Guru](#) to identify potential bottlenecks and then work towards resolution.

Figure 11-9 shows the final picture of the architecture of the URL shortener system.

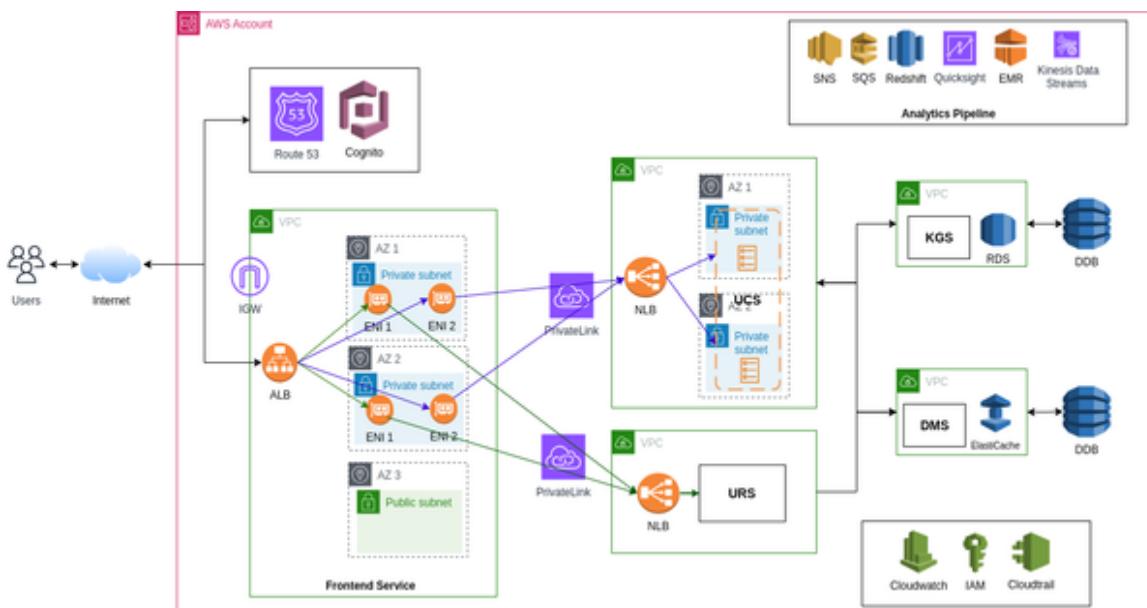


Figure 11-9. URL Shortener System

Here are some additional points to keep in mind, which we left out of the diagram to make it cleaner:

- The services residing in different VPCs require a mechanism to connect them to each other. AWS Transit gateways (TGW) provide bidirectional communication between the VPCs while AWS PrivateLinks provide unidirectional communication. Further, the CIDR blocks of VPC should not overlap in case of TGW but PrivateLink doesn't take this into consideration.

- The microservices also interact with AWS services which are present in our owned VPC. These services should be accessed via [AWS PrivateLink](#) instead of NAT gateways. This ensures the traffic doesn't go over the internet which adds to application security and minimizes operation latency. NAT gateway should only be used in case an explicit connection to the internet is needed.
- Other ways for microservices running on Amazon ECS to establish communication with each other can be via [Amazon ECS Service Discovery](#), [AWS App Mesh](#) and [Amazon ECS Service Connect](#).
- Amazon ElastiCache is used as a caching layer on top of DDB to serve read queries faster. ElastiCache will remove the least used data as per the cache eviction policy—we can introduce an additional AWS Lambda function (invoked on DDB data deletion event) to ensure data removal from the cache if removed from DDB. This can happen in the case of any custom expiry on the short URL by the users. Instead of the AWS Lambda function, we can also implement a consumer in DMS service itself which can handle this responsibility.
- The architecture presents deployment in a single AWS region. We recommend creating all the infrastructure via code for easy replication to other regions if required. The system resources are deployed in multiple AZs to ensure increased redundancy in case of AZ downtimes.
- The main reason for separating UCS and URS is the huge difference in traffic patterns. A separate service ensures independent scaling and clear separation of concern.
- The system architecture should ensure the increased traffic is from actual users and throttle any unwanted traffic at API Gateway or ELB level by utilizing AWS services such as [AWS](#)

**WAF** and **AWS Shield** or deploying custom solutions. This can be handled in the Frontend service.

The points above are not an exhaustive list of issues to take care of while serving customers. We will cover additional topics in more detail in the remaining use case chapters of Part III and you should take a holistic view of all chapter learnings to deploy systems in production to ensure high standards.

## Conclusion

We started off Part III with the URL shortener system which is a popular interview problem and a very common system we use in our daily lives. The gist of any new system design is to start with requirement clarifications and system boundaries. You should clearly know what a system is expected to do and what a system will never do—this helps in making better design decisions. Once the requirements and scope of the system is finalized, we move on to identifying the high level components. The scale a system is supposed to serve is an important factor to identify these components along with the business use case. A system operational with 10k users might not work the same way with 100k users.

We moved through multiple possible approaches, discussing their pros and cons, along with potential AWS Cloud components. We brought in many of the different AWS services we discussed in Part II of this book and we acknowledge that recognizing which one to choose for your specific use case could be difficult. We recommend choosing a technology that will take minimum time to understand and deploy—down the line, it can be modified or replaced by another technology if needed.

System design is an iterative process, so start small and enhance your systems on the go as needed. We recommend not thinking too far ahead, because your priorities may shift after five or ten years.

In the next chapter, we'll design a web crawler and search engine system such as Google search, using AWS components.

# Chapter 12. Designing a Web Crawler and Search Engine

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 15th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

You planned a get together with your loved ones in the holiday season. You love cooking and decided to cook all the food by yourself, but you don't have the recipes for the dishes you wish to prepare. What is the best possible resolution here? You could ask your friends if they have the recipes, or go looking through cookbooks, but a very simple yet effective solution is using Google search. Google looks across the internet and finds you the best results on how to prepare a specific meal. How does Google go through such a vast sea of information and find the perfect answer? In this chapter, we'll try to figure out this answer by digging into the architecture of such search systems.

At a high level, the entire system consists of two subsystems, a web crawler and a search engine. A web crawler is essentially a software

responsible for crawling the web. The content on the internet is growing exponentially and web crawlers need to ensure the content is regularly crawled to maintain most updated content. The Search Engine sits on top of content accumulated by web crawlers and stores it in such a way that it can look for user searched keywords from the content and present the most useful results, as shown in **Figure 12-1**.

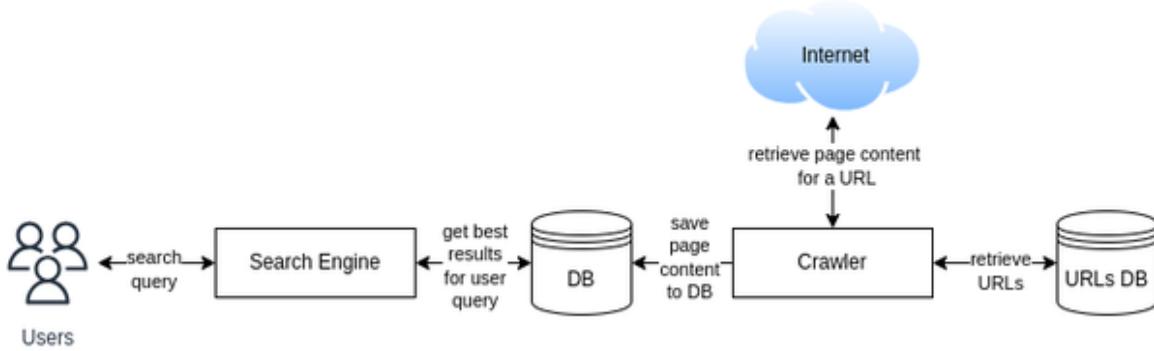


Figure 12-1. Web Crawler and Search Engine 10000 ft. architecture

With this basic understanding, let's start by gathering functional and non-functional requirements of the proposed system.

## System Requirements

Web crawlers can be used for multiple purposes and focusing on one use case makes it easy to move in the right direction with our design. For this chapter, content retrieval via search engine is the main problem statement we're trying to solve, but you could also look into other issues such as web monitoring tasks like copy infringement issues. A web crawler is needed in the system to gather the data but from the user's perspective, and the product is a search engine. The functional and non-functional requirements are captured in the next section.

## Functional and Non-Functional Requirements

The functional requirements of the system include:

- Get top results (title, subtitle, page URL) for user search query.
- The results are frequently updated and the system should be able to identify the content freshness rate from the web pages on the internet.

And the non-functional requirements include:

- The system should be highly available (able to serve user queries most of the time) and highly reliable (the queries should be served correctly).
- Low latency for search queries. Considering the huge number of pages mapping to user query, the system should return the most relevant results within milliseconds of response time.
- Data freshness—the underlying storage to serve search queries should be regularly updated.
- Data consistency is not super critical for this use case. A lag in content available on websites and searchable via the search engine is acceptable.

We should also document the system boundaries or what a system is not expected to do. For this use case, that includes:

- The system will not include recommendation and content relevance algorithms for gathering the search results.
- The search doesn't support near real time search results. For example, the live score of an ongoing cricket match.

Next, let's jump to the expected scale for this system.

## **System Scale**

The system has two components: web crawler and search engine. Both of these components can be independent systems in themselves and we'll structure the chapter content in a similar

fashion to discuss them independently. This makes sense, as the requirements of both of these systems vary and they are expected to solve different use cases, with one system's (web crawler) output being used as input to another system (search engine). We'll consider the below assumptions for expected scale on the system.

## Web Crawler

The important points of consideration with the web crawler are the number of pages on the web to be crawled and how often these pages need to be crawled. As per a quick Google search, there are around 1.13 Billion websites created as of June 2023 and around 200 Million are active. The web crawler system is only concerned about the active websites. Inside a single website, there can be internal redirects or external redirects to a different page. The external redirects are already counted as part of the 200 million websites, but we need to consider the internal redirects. We'll take assumptions for the average numbers and can use the calculation below to determine how much storage we'll need:

```
Active websites = 200 Million  
Average pages/website = 50  
Total pages = 200M * 50 = 10 Billion  
Average page size = 2 MB  
Total storage = 2MB * 10B = 20 PB
```

The web pages can include different forms of content such as text, images, videos. The web crawler should only crawl and keep the data that will be used by the search engine. We'll limit the scope of the system to only support text queries in the English language. This will also potentially reduce the storage space—let's say 500 KB per page accounting to 5 PB of total storage. Another consideration here is the requirement to store the entire page.

## NOTE

The storage requirement calculated in this book considers content is stored in the same format as received, but this can certainly vary based on data store selection, data compression algorithms wherever applicable, separate caching data store, etc.

The content on the internet keeps adding up in the form of new websites and web pages, and the content is constantly updated on the existing web pages. The system needs to ensure the content freshness so as it stores the most relevant data from the internet to offer the ideal search experience. Let's look into the scale for the search engine system next.

## Search Engine

What do we estimate the number of requests per second to the search engine system will be? Users open the browser and search with some keywords and the system should be able to respond with search results that match the search query. Let's consider Google's scale, since it is basically the most visited website in the world.

Google serves roughly 8.5 Billion searches per day (99k requests/second). To support the search scale, data should be indexed which requires storage space apart from what we discussed in the web crawler requirements.

We'll work to build a scalable system and talk about key concepts involved in the system, but keep in mind that Google has evolved a lot over time to support this scale and it will not be possible to capture all such details in a single chapter. Some of the factors that should be considered are:

- The search results can vary based on the geographical region. For example, the search query with keywords "latest news" might generate different results in India as compared to the USA.

- The system can receive similar search queries from different users and it should include good caching capabilities to make queries faster.

With the above requirements and scale considerations, let's now dig deeper into the design of the system.

## Starting with the Design

As we did in the 'System Requirements' section, we'll discuss the system design analysis for the web crawler and the search engine system separately. Before we dig deeper into the design, we recommend reading through the Google [architecture](#) paper published back in 1997. This paper nicely captures the high level architecture of web crawler and search engine systems and serves as the starting point to begin the system design analysis.

Here is a crisp summary of the architecture presented in the technical paper mentioned above:

- To start the crawling process, an URL server provides a list of URLs to the crawler system to fetch web pages from the internet. Each crawler maintains its own DNS cache for faster lookups.
- Store server system receives the web pages gathered by the crawler and its responsibility is to compress (using [ZLIB](#) compression) and store the HTML content of every web page to a repository. The repository data structure is an independent component and can be used to build all other data structures in the design.
- The Indexer reads the web pages from the repository, uncompress them and parses the content. The functions performed by Indexer include:

Parsing of content to ensure removal of any errored or unwanted content such as non-ASCII characters.

Any new URL (web links) found by parsing the web page is assigned an identification, called docId. All the important information about links such as where the link is pointing from and to, and text of the link is stored into data structure referred to as Anchors.

Every word is converted into a wordId by using an in-memory hash table called lexicon.

The occurrences of the word Ids in the document are stored in a set of barrels in the form of partially sorted forward index called hits. The hits include the actual word, position of word in the document, approximate font size and capitalization. These characteristics can help in determining relevance of words as compared to other words in the document, such as large font size in the beginning of the document can represent the heading of the document).

- URL resolver reads anchor files and converts relative URLs into absolute URLs which map to the docIds with the additional responsibilities below:

Puts the anchor text into the forward index barrel, associated with the docId anchor is pointed to.

Generates a database of links which are pairs of doc Ids. The link database is further used to compute page rank for all the documents.

- Sorter takes forward barrels (sorted by doc Id) and sorts them by wordId in place to generate the inverted index for title, anchor hits and full text. The sorter further divides a barrel into baskets as short or full barrels based on the fact that they can fit into main memory or not.

- The searcher is run by a web server to serve the search queries. To process any search query, it parses the query, converts words into word Ids, seeks to the beginning of docList for every word in the search query in the short barrel, finds the document matching all the words, computes the rank of document and perform the same set of operations on full barrel. Finally, sort the matching rank documents and return top k as response.

### NOTE

Forward and inverted indices are terms often used in search use cases. Forward index is a mapping of document Id to a set of words or keywords. Inverted index is inverse mapping of forward index containing the mapping from keywords to document Id.

Let's dig more into the design of web crawlers.

## **Designing Web Crawler**

We'll expand on the architecture from the previous section to design a scalable web crawler system. The first thing we'll need to consider is the list of URLs that a system should start with (seed URLs) for crawling the web. How should this list be maintained and how should it be updated on a regular basis? There is no unified source that can be used to retrieve all the URLs to be used in the seed URL list. The only way to build this list is by starting out with most known domains and then constantly updating the list on web crawls. The steps to building this list include:

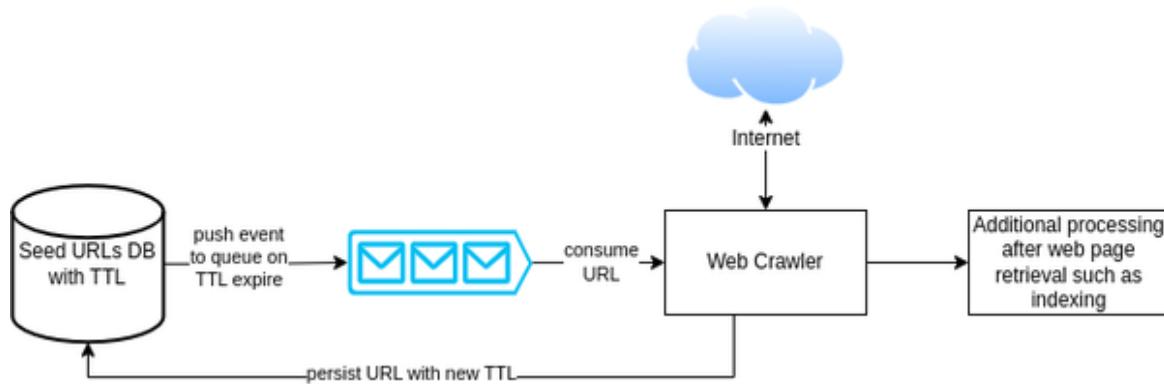
- Make a platform so that new website owners submit their site details available for crawling.
- Retrieve the list of website links/references in the website being crawled.

- When a user queries for a new website, the crawler checks this website and adds the website to the list.

The crawling frequency of the web crawler system is not fixed and can vary from website to website. Historical analysis of the website is one of the ways to determine the crawling frequency. This helps in determining if the website is frequently or rarely updated. News websites can be frequently updated with fresh content but personal websites are in general rarely updated.

Let's discuss one potential architecture based on web page recrawling at a fixed regular time interval as shown in [Figure 12-2](#). The architecture describes one potential implementation idea and can work with the steps below:

1. Seed URLs DB stores the URL with zero TTL or an appropriate TTL so that not all URLs are pushed to the queue at the same time. The implementation of change data capture on TTL expiration can vary based on DB choice—for example, DDB streams can be leveraged in the case of Amazon DynamoDB.
2. On TTL expiration, URLs are pushed to the queue and the web crawler consumes the same.
3. Web crawler retrieves the content of a web page from the internet and sends it for additional processing such as building indices.
4. Web crawler saves the URL again to DB with new TTL. As a web crawler processes the web page content, there are also extra web links included, which are also pushed to the queue for processing.



*Figure 12-2. Recrawling of web pages based on configured TTL*

On the same note, a web crawler system should have a change detection algorithm deployed to efficiently figure out if the content is updated without crawling it entirely. Here are few ways to accomplish that:

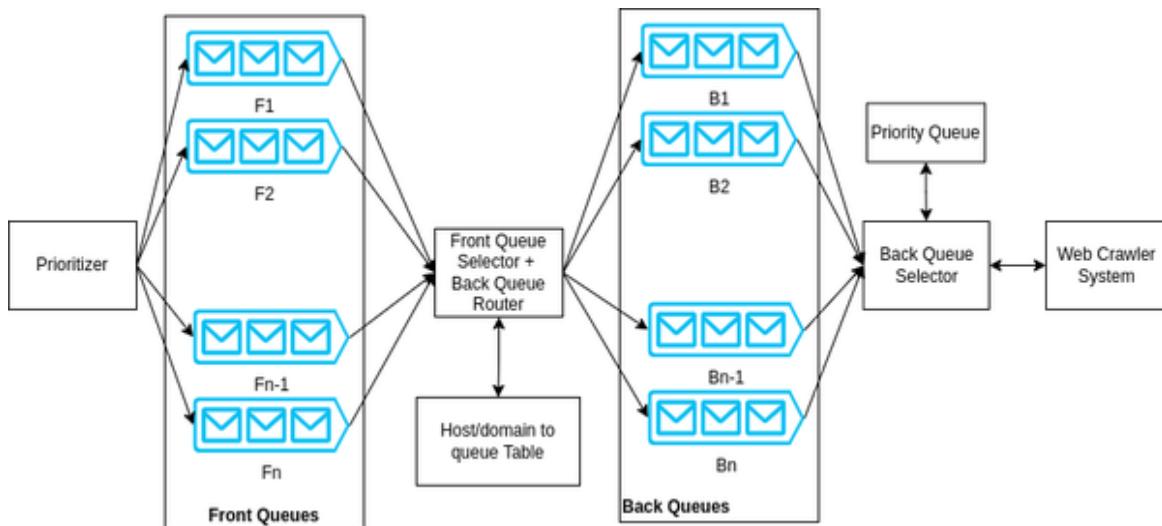
- **ETag** (entity tag) is an HTTP response header which uniquely represents a requested source and it is typically a hash of the content. The ETag value can be used to determine if the requested web page has changed or not. The **Last-Modified** HTTP response header containing date and time of last modification is another way of knowing when the resource was last updated.
- Content comparison can be another way of validating content modification. However, it is definitely not an efficient process to compare word by word—this is essentially the same as reading the entire web page. A more efficient solution is figuring out specific portions of the page where the content is updated. One such solution is Merkle Tree (discussed in Chapter 3) which helps in efficiently evaluating the updated section; it is generally used by databases such as **dynamo** to resolve anti-entropy among multiple replicas. Google crawler uses the **Simhash** algorithm to help in figuring out content duplication of the web pages.

- Occasionally a website will become popular due to the fact that a celebrity shared it over social media. The web crawler can detect such traffic surges, then recrawl the website for any content update.

Earlier we mentioned a queue which can be used to push the URL for a web crawler to consume. The TTL approach based on time is not generally used directly in the large scale web crawlers because the page to be crawled next can depend on multiple other factors apart from TTL. The TTL factor in web crawler terms is referred to as politeness, meaning how frequently a crawler fetches content from the web page server. This can happen too often if the URLs referenced in the web page belong to the same domain. Another important factor is figuring out the high quality pages that need to be crawled on priority. The system responsible for determining the next URL to be crawled in web crawler architecture is generally referred to as URL Frontier.

## **Improving URL Frontier**

The previous architecture can be extended further to address the limitations and takes priority and politeness both into consideration. The architecture is referenced from **Mercator** web crawler architecture and shown in **Figure 12-3**.



*Figure 12-3. URL Frontier architecture*

The functioning of different components of URL frontier system are described below:

#### *Prioritizer and Front Queues*

Prioritizer and Front Queues (FIFO) are responsible for figuring out the priority in which the URL should be crawled by the web crawler system. Prioritizer assigns the priority to the URL from 1 to n and based on this number, the URL is pushed to a specific front queue. The priority assigning algorithm can take a number of factors into consideration such as percentage change in web page between two crawls, newness of the webpage, user traffic on the webpage, and more.

#### *Front Queue Selector and Back Queue Router*

This component is responsible for taking the URLs based on the priority and passing them to the back queues. It maintains an additional database to keep domain to back queue numbers to ensure the URLs from the same host are served via a single back queue. This is done to make sure a single domain is called once at most (or as per the domain configurations configured at the **search engine** level) by the web crawler at a time.

## *Back Queues*

Back Queues ensure politeness of the URL to ensure the web server is not bombarded too frequently (single request at a time with some lag between the requests). The number of back queues should be more than the number of crawler threads available in the system to ensure every thread is busy at any given time. This can be configured as three times the number of threads to start with and later tune with proper performance testing of the system.

## *Priority Queue*

Priority Queue is a heap implementation which stores one entry for each back queue and provides the time at which a particular host (or domain) can be contacted again. The heap is created on min-heap implementation and the root element provides the URL which can be contacted first.

## *Back Queue Selector*

The Back Queue Selector interacts with the heap to take out the root containing the domain with minimum time and then take out the URL from the particular back queue. Once the URL is crawled by the crawler, the heap is updated with the new time to configure a new politeness value.

The FIFO queues in the architectures keep a fixed number of URLs in the main memory and store most of the URLs on the disk to support high crawl rate on commodity hardware. Even with the best hardware machines, disk support is required considering the scale of the internet. The disk storage algorithms can become a bottleneck for processing a huge number of URLs due to very frequent reads and writes to the disk, as pointed out in the **IRLBot** research paper, which tested results with 184K new URLs generated per second from crawled web pages.

## NOTE

The current design and implementation of large scale web crawlers such as Google and Bing is proprietary and not available to the general public. We explored multiple public research papers along with our own knowledge base to ensure the web crawler system's scalability to support today's internet scale.

IRLBot suggests the Disk Repository and Update Management (DRUM) algorithm as an algorithm for ensuring URL uniqueness. DRUM maintains k memory arrays each of size M and k disk buckets ( $Q_i$ ) and spreads key-value pairs between them based on the key. The data is moved to disk buckets as memory arrays get filled up. For pages with a lot of sub URLs (in the millions), the politeness rate-limiting can become a bottleneck. IRLBot solves this via Span Tracking and Avoidance through Reputation (STAR) algorithm. The reputation of a page is calculated based on in-degree links from other domain web pages. There is always a possibility that some web pages don't show much relevance due to being new or because they have fewer in-degree links, but they should also be crawled to ensure search engines can support any user queries. This is solved by Budget Enforcement with the Anti-Spam Tactics (BEAST) algorithm. BEAST maintains an additional queue to all such URLs and keeps on reevaluating their relevance to figure out when the URLs should be crawled by the web crawler system.

The web crawler system also ensures avoiding infinite loop scenarios while crawling the web. It can happen that an URL is referenced at multiple places in a single web page (or multiple web pages during a single crawl of the system) or a URL contains reference to itself.

## URL Uniqueness and Duplicate Detection

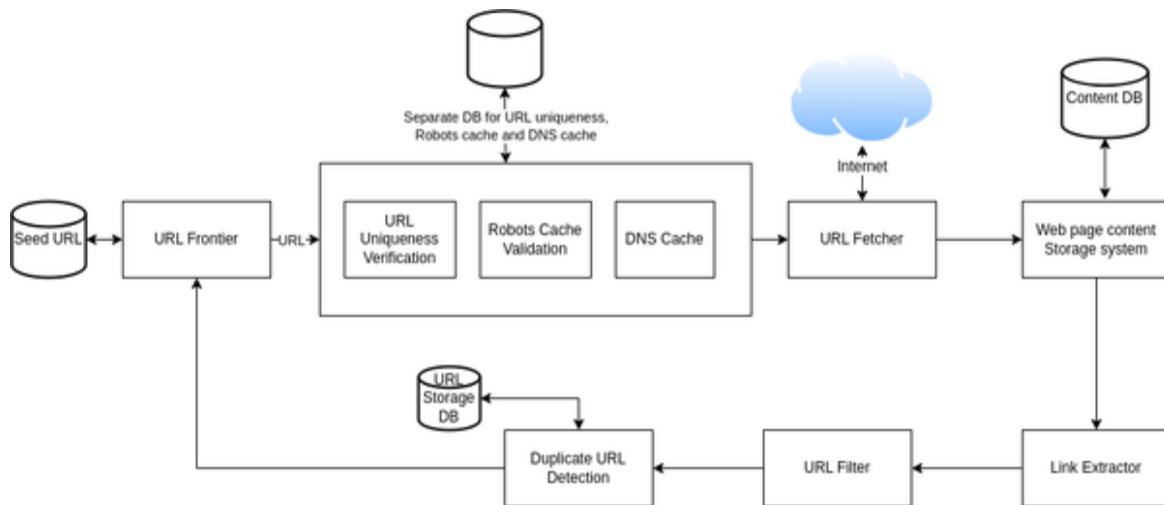
The system needs a mechanism for duplicate URL detection. This can be done by maintaining a separate global database across the crawler machines maintaining the list of visited URLs in a single crawl. As the query to determine whether the URL is duplicate needs

to be fast and doesn't need to be persisted across the crawls, we can use Redis and store the URLs as keys. The Redis cluster can contain TTL policies associated with an URL (which vary from URL to URL) so the URL is automatically removed from the cache.

Bloom filters is another technique which can be used to determine if the URL was visited or not—the key consideration here is the probabilistic nature of bloom filters. It is possible that bloom filter returns yes for an already crawled URL, causing recrawl of the URL.

The crawler will also be sure to respect any exclusions on the web server configured in robots.txt file. Robots.txt file determines whether a web page is allowed for crawling or any specific resource is not to be indexed by the search engines. On top of exclusions by the web server, there are also exclusions maintained by the web crawler to avoid visiting and indexing content from any blocked websites.

Let's combine all the discussed components to present the final picture of the web crawler system as shown in [Figure 12-4](#).



*Figure 12-4. Web Crawler architecture*

There are a few additional items to consider about the system:

- The crawler is a multi-region deployment and it is better that a crawler crawls from the same location a website is located for

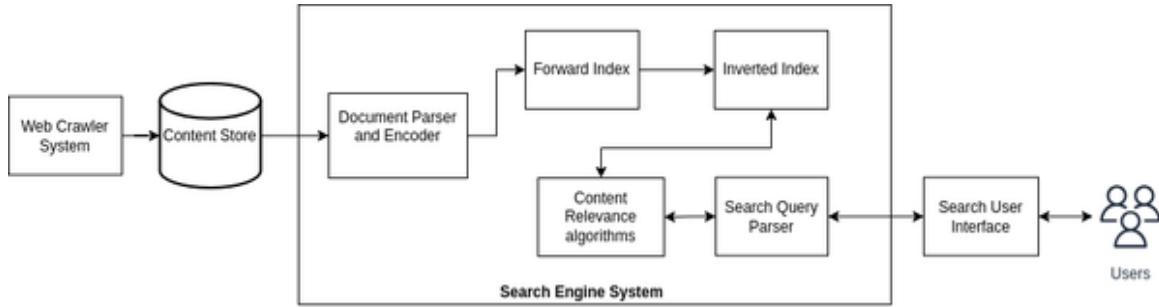
improved performance. Another reason for this is that some sites, such as government websites, are blocked from being accessed from other countries, so it becomes necessary for a crawler to run in the same region the site is located.

- The crawler mostly uses HTTP/1.1 protocol version to retrieve the web pages. It can use more recent HTTP versions such as HTTP/2 depending on whether or not it's supported by the website. The newer version saves computing resources both for the crawler as well as the website.
- The content on some of the websites can be huge so the web crawler always caps on the bandwidth consumption. For example, it will only download 15 MB of content and skip the rest of it.

The web crawler's job finishes as soon as the web page is downloaded. The search engine system then takes this content as input and indexes it efficiently to serve user search queries. Let's explore the architecture of the search engine system in the next section.

## **Designing Search Engine**

The Search Engine system takes a user query as input and provides relevant web pages in response to the query. The web pages are downloaded and stored to a repository (or some kind of database) by the web crawler system which we discussed in the last section. The search engine should be able to figure out the best matching results for the user search query from this content repository. We'll define a high level architecture of the search engine system as shown in [Figure 12-5](#) and then dive deep into each of the components.



*Figure 12-5. High level architecture of search engine system*

With the view of the entire architecture, the crawler downloads the web pages and the Document Parser and Encoder is responsible for tasks like parsing the document and removing unuseful data, and data is passed to the forward index. The forward index stores the data and then it is passed to the inverted index for reverse mapping of data. For any user search query, the query is parsed to search keywords, the content relevance system (such as PageRank) computes the most relevant results and returns them to the user.

You might wonder why we need a forward index in the architecture, as the inverted index should be sufficient enough to serve user queries efficiently, and it's a valid question. The creation of forward index allows storing the words as they are parsed from the web page document, allowing the inverted index to be a completely independent process that can happen asynchronously. There are different open source solutions already built to support inverted index data structure creation. Let's discuss our options for the search engine system further in the next section.

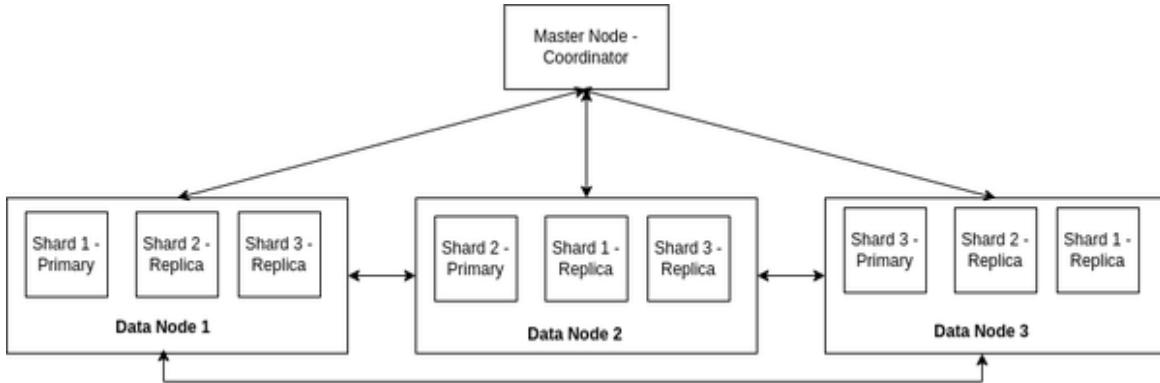
## Indexing Strategies

We'll start the discussion by diving deep into the architecture of open source search solutions such as Elasticsearch (ES) and Amazon OpenSearch service (we discussed this very briefly in Chapter 10). Many organizations such as **Netflix** use Elasticsearch to build inverted indices and build search functionality on top of it. ES is a non-relational datastore service which can be used to store JSON

documents and ES constructs inverted indices to support the search queries on the stored data.

An index is quite similar to an Amazon RDS table. The indices are further divided into sub-indices referred to as shards (the shards internally map to the **Lucene** index—ES is built on top of Lucene). The documents are stored on the shards, and a document is similar to a row in the Amazon RDS table. Each shard in itself consists of multiple segments which are kept in memory (and a transaction log on disk to overcome failures) and are synced to disk in fixed time intervals (referred to as refresh interval). A segment is an immutable data structure and is available for search once saved to disk. If a use case requires data to be available for search as soon as it is written, the refresh interval can be configured to a lower value. The smaller segments are merged into a larger segment from time to time improving the overall search performance (search needs to be performed on a smaller set of segments). We'll discuss more about the shard configurations in Chapter 18.

To support high availability and read scale, the shards are replicated on multiple data nodes as shown in [Figure 12-6](#). The data is written to the primary shard on one of the data nodes and then replicated to replica shards asynchronously. The system tries to ensure primary and replica shards are not co-located on the same data node (given the ES cluster has those number of nodes as per shards configuration). Now take a look at Data Node 1 in [Figure 12-6](#), it contains primary shard 1 and replicas for shard 2 and 3. The replicas are useful to scale the read operations without impacting the write performance, but the key consideration here is all the three shards are sharing the same CPU and RAM resources of the data node, where one shard is responsible for writes and the other two for reads. ES uses disk to store the indices and offers multiple caching [techniques](#) on top of it to make queries faster.



*Figure 12-6. Data indexing and replication between ES shards*

Further, the reads can be scaled by adding more replicas but what about the primary shard? If there are issues with a data node consisting of a primary shard, it can impact the indexing performance until the data node is recovered or replaced by some other node, becoming a bottleneck in the overall architecture. In ES architecture, the search engine is responsible for indexing the data and uses the same index to support user search queries. This can potentially introduce a bottleneck of operation impacting the performance of another; meaning reads and writes are happening on the index at the same time but there can be sudden traffic bursts in any one of them impacting the other—for example, a viral search keyword queried by a huge number of users in the same timestamp. In Part II we covered the architecture of multiple AWS services and the key thing was decoupling compute and storage in services like Amazon Redshift, Amazon EMR, Amazon Aurora, etc. Along similar lines, is it possible to architect the system in such a way that both write and read operations can scale independently and are completely decoupled?

To resolve this, we can leverage cloud object storage as intermediary storage where indexing engines upload the data and search engines can download the data, as shown in [Figure 12-7](#). The multipart upload and download ensures uploading the large files in chunks to ensure better reliability and ease in re-uploading or re-downloading a smaller chunk if the operation fails. It is recommended to keep

shards size relatively small (not too small, to a few GBs only) which helps in improving indexing latency. This also makes scaling (adding or removing nodes) search and indexing clusters faster due to reduction in data rebalancing overhead.

### NOTE

We'll discuss more about optimal shard size for ElasticSearch cluster in Chapter 18.

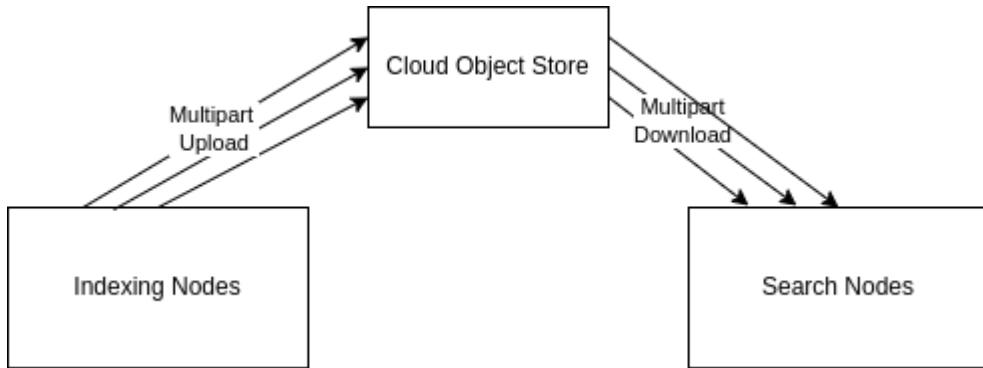


Figure 12-7. Decoupled indexing and search systems

The other benefits of cloud storage options such as Amazon S3 is that it includes no management overhead of persistent backups, cross AZ and region replication. In short, we're getting all the cloud storage availability benefits in the search engine system. And again, it is always a point of discussion, deciding whether to use any cloud provider and fully manage a custom solution on your own at such a huge scale. There is no foolproof answer to this and we recommend evaluating as your system requirements, cloud pricing considerations, and control needed over the service evolve.

We ensured the indexing and search process can scale independently. Now, to address indexing system bottleneck on write operations, the system can ensure the following mechanisms:

- The write operations should be done in batch or bulk instead of single write operation for each and every query. This will help to

reduce load on the system.

- Instead of leader-follower (primary-secondary shards) architecture, the system can be designed to support leader-leader architecture where data is being written to multiple primary shards at the same time with a consensus and quorum algorithms in place. If there would be a single leader, there is no requirement of consensus and quorum so there are tradeoffs in both of the approaches.

We're clear on the fact that we're not using the ElasticSearch or Amazon OpenSearch for building the search engine system and we need a custom solution to build the inverted indices. An inverted index can be really huge given the amount of data and the system will include a mechanism to divide the data into multiple machines and clusters. A single huge index cluster is tough to manage as compared to multiple small clusters (well, not too small) and can become a bottleneck in architecture. How should an inverted index be divided into a set of machines? We can represent an inverted index as a mapping from a keyword to the list of document Id and frequency the word appeared in the document.

```
Term (key) - {docId, frequency} (value)
Hello - [{doc1, 5}, {doc2, 3}, {doc4, 1}]
World - [{doc1, 5}, {doc3, 4}]
```

There are two approaches here to divide the data:

#### *Document Based*

All the terms from a document are stored on a single machine, offering the benefit of calculating page relevance easily (if all the words in a search query are present in a single document, the system is able to retrieve the result on a single machine only). To gather search results, the master instance forwards the query to all the worker instances and then combines the results to respond to the user queries.

## *Term Based*

To simply illustrate this, assume the system has 100 terms extracted out of the crawled web pages. These 100 terms are divided into a set of machines, say 10 machines with 10 terms each. To respond to search queries, the master instance can figure out which worker instances to go to and it doesn't need all the workers like in the case of document based partitioning. The downside of this approach is creating intersection of search results because the terms corresponding to the same document can reside on different machines that require data transfer over the network.

Based on the above trade-offs, we'll choose document based partitioning to maintain the inverted indices. Let's discuss scaling of search nodes facilitating the user read operations next.

## **Improving Search Performance**

The latency is supercritical and it is important the user queries are answered in sub-milliseconds. The ideal suggestion is the search nodes should keep all the data primarily in RAM, meaning all the inverted indices are cached but it comes with a cost. Disk is  $\sim 100x$  cheaper than RAM and considering the scale requirements, the cost differentiation between disk and RAM varies by huge factor.

In the case when all the data is kept in main memory, is there a requirement for persistent storage? For any node failures, the data can always be downloaded from cloud storage. It's a tough call to make and we recommend keeping the inverted indices as backup in disk storage as well. Yes, they can always be downloaded from cloud storage but that can take more time as compared to local disk seeks. There are pros and cons associated with any approach we take; whether indices are completely in memory or disk. We can take a middle ground to keep the majority of indices in the memory with disk as persistence option. The indexing algorithm will ensure that

the data is stored in a manner requiring minimum disk seek operations (ideally one). The compression of data becomes important in case the maximum amount of data resides in the main memory.

## **Data Compression**

We discussed the storage space requirements for the web crawler based on a few assumptions in the System Requirements section. The general idea of compression is that the operation of compressing and then decompressing will add to the latency of both indexing and search operations. This is because the system is spending some time first in compressing and then at a later stage, decompressing the data to support user search queries. One thing we're 100 percent sure on is that compression can save storage space. Given that decompression can severely impact the search latency, we should think about latency a bit more. The search latency is critical in the system—indexing latency is also critical but we can survive with little lag because it is saving us the huge storage cost and we're fine with an eventual consistent system. We'll discuss this in two parts: when the inverted index resides in disk vs when it resides completely in memory.

### *Inverted Index in Memory*

To process an inverted index list without compression requires two steps; moving data from memory to CPU cache and then processing it. With compression, it adds an extra step of decompressing the data as well. The operation of data movement from memory to CPU cache becomes really fast on compressed data and decompression takes relatively lesser time as compared to data moved. So essentially, the search query latency is improved with compression on the data.

### *Inverted Index on Disk*

As per the discussion in the above point, this adds one additional step of the movement of data from disk to main memory, but this also performs well because we're moving less data. This increases the search query evaluation speed, giving us a benefit of low storage cost plus the improved performance with the compression in place in the system.

With all of that taken into consideration, the search queries are quite faster in responding to user requests with compression in place. Now think of a scenario when a particular search query becomes popular and the system is receiving a lot of requests with the same search query—this may become a bottleneck in the search system. To overcome this, the system ensures the auto scaling is in place to replicate the data on more nodes to serve queries. Additionally, it maintains a cache layer regional to users to serve the same search queries without even hitting the search system's inverted indices. This ensures the search system is not impacted and user queries are served reliably and with much less latency. There is definitely an implementation in place to invalidate the regional cache from time to time and maintain the data freshness. As the search engine system creates inverted indices, it also assigns the score to each page to make sure relevant content is served to the users.

## **Relevance of Search Results**

For any search query, there can be hundreds of search results and the search engine system should ensure the results are in sorted format with high score web pages on the top. Here are few of the factors that contribute in deciding the content relevancy of a web page:

- A web page is assigned a higher score if a lot of other web pages point to it. In these web pages as well, if some top domain adds the reference to a page, it becomes more relevant. For example, a research paper used as a reference on Stanford

University might hold more value as compared to a research paper quoted in 10 lesser known universities.

- The words in the web page title, heading, larger font or any other special characteristics (e.g. embedded links, bold/italic/underlined font, etc.) can be given higher weightage.
- The location of the web page adds to a score in a sense that a web page could be very relevant in the USA but not so relevant in India.
- The search engine prioritizes positive content instead of negative content in top search results unless the search query explicitly evaluates to the negative content.

There are many more attributes to decide a page rank but we'll close on the search engine discussion and move on to launching the system on AWS Cloud in the next section.

## Launching the System on AWS

As we did in chapter 14, we'll follow the policy of quickly setting up the application and choose AWS services that require no to minimum expertise, and then further tackle the problems as we understand more about how the overall system is functioning and the expectations from the users.

### Day Zero Architecture

The day zero architecture is based on this AWS [blog post](#) about scaling up a serverless web crawler and search engine. The entire process of the web crawler system can be broken down into a series of steps and it essentially keeps on repeating these same steps until the entire list of URLs is crawled. To break down the workflow into a series of steps, AWS Step Functions (discussed in Chapter 12) is a great choice of service on AWS cloud. [Figure 12-8](#) shows the state

machine representing the working of the web crawler for downloading web pages.

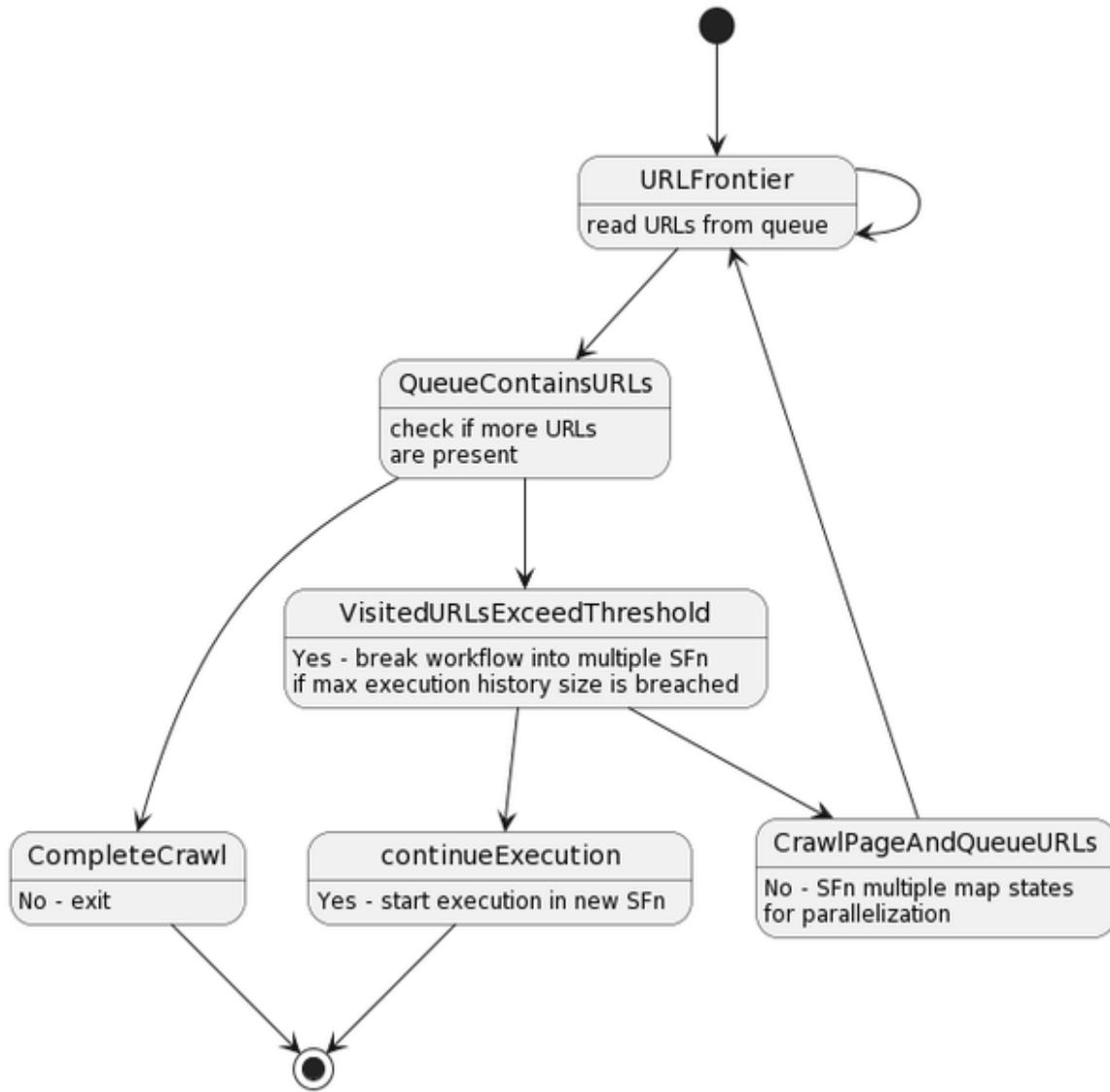


Figure 12-8. State machine for web crawler implementation

All the states in the above state machine can be executed for separate lambda functions and also utilize AWS SFn map states for parallelization. There is also a requirement to store the URLs which can be passed along between the states for evaluation, such as if the URL is already crawled or if there are more URLs to be crawled. Since the list of URLs is huge, it's not possible to pass between the states using AWS SFn only—the system requires persistent storage.

For database choices, the system doesn't require any relations or strong consistency across the data so non-relational services such as Amazon DynamoDB or Amazon Keyspaces can be an ideal choice here. The system stores the available URLs and any related metadata and requires direct key operations on these URLs. We recommend using a database based on previous expertise here as none of these two choices stands out clearly. For illustration purposes in this chapter, we'll use Amazon DDB. The system essentially requires two tables, one as a building block of the URL Frontier system to store URLs and used across the crawls. Another table is created at the start of the state machine and deleted at termination of the state machine; this table stores the URLs crawled during the ongoing crawl to ensure a single URL is not crawled multiple times.

Once the web page is crawled by the crawler system, it needs a repository database to dump all the web page content. For such use cases, Amazon S3 is the preferred choice to store the data and the same can be used in the second step of architecture for a search engine to consume this data and index it. Building a custom search engine is a big task and does require an end-to-end thought process but to deploy the system quickly, we can use Amazon Kendra. We introduced you to Amazon Kendra very briefly in Chapter 13.

Amazon Kendra is a fully managed service offering intelligent search capabilities. Amazon Kendra is widely used as an enterprise search solution often targeted to build a custom search engine on top of internal documentations present at multiple places such as slack, google drive, relational databases, confluence doc, etc. It offers **connectors** to connect with these different data sources and will automatically build a search solution on top of these. As the crawled web page data is stored into Amazon S3, the **S3 connector** can be used for configuring Amazon S3 as a data source to Amazon Kendra. The entire architecture of the web crawler and search engine system after combining all these components is shown in [Figure 12-9](#).

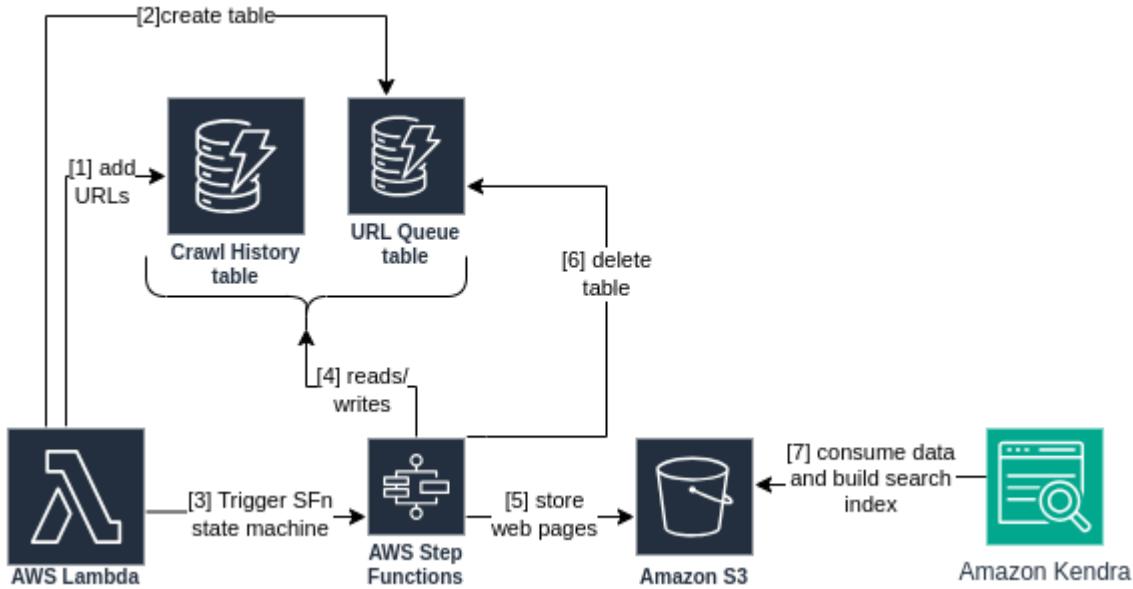


Figure 12-9. Web crawler and search engine architecture

Interestingly, Amazon Kendra also offers web crawler as a [connector](#) and we wouldn't even require the web crawler part of the architecture. All we need to do is provide the seed URLs and Amazon Kendra can take the responsibility of crawling the websites and indexing it to support the search on top of it. Let's move on to dive deep into the architecture for any limitations the system can face as it scales to serve more and more traffic.

## Scaling to Millions and Beyond

If we ignore infrastructure cost on AWS Cloud for a while, can you then think of any limitations in the architecture explained in the previous section? Ideally, the architecture uses AWS serverless components so the components should automatically scale up and down as per the nature of traffic without any user interference. Some of these limitations include:

- AWS SFn can face higher latency and quota limitations. This was addressed by suggesting multiple SFn and usage of map states.

- AWS Lambda as a compute option scales automatically as per the provided memory configurations. The cold start is not much of an issue because the system can survive a little lag, but we can still leverage a few of the options such as provisioned capacity to make the process of downloading web pages a little faster.
- Amazon DDB is a fully managed service which scales indefinitely to support the storage requirements. One point of consideration here is the hard limit for RCU and WCU consumption on a single partition—it should exceed the limit of 1000 WCU and 3000 RCU or else the requests are throttled. The Amazon DDB table is created with partition key as URL. Now considering one RCU is consumed for one request, the system can breach this limit if the following situation occurs: a URL is referenced in a lot of web pages and the web crawler is working concurrently for more than 3000 web URLs at the same time. The positive side of this could be that even if the requests are throttled, we can assume the URL is already present in the DDB table and safely skip it (the request was throttled because the partition is present in the DDB). The problem that can surface is retrieval of additional attributes associated with a URL.
- We can consider latency as one parameter on URL retrieval from DDB as there is no caching introduced in the system. Further, the system doesn't store robots cache and DNS cache which adds to the overall latency of calls to web servers.
- The web page content is stored into Amazon S3 which offers unlimited storage and automatically replicates the data for high availability, so there's no bottleneck here.
- Amazon Kendra is again a fully managed service and due to this nature, we don't control how data is stored or what capacity is offered to the customers. The service offers fixed **capacity** for storage and queries to the customers and it can further be

incremented with the help of AWS support. This control is often preferred at the customer's end if the system needs to support a huge scale, because it makes it easy to custom build the software as per each user's needs.

If your system is operating at Google's scale, or even 1/10th of this scale, it makes sense to manage the implementation on your own instead of relying on the managed services such as Amazon Kendra. This saves cost along with offering the fine tuned control on low level configurations of the system.

#### NOTE

We're not discouraging you from using AWS managed services—they are quite great and save a lot of time building software without any operational burden and expertise on the user's end. We recommend evaluating your system architecture from time to time and deciding on what works out best for you.

In day zero architecture, we used Amazon Kendra to help out with the search engine functionalities. Now, let's figure out which microservices can be used to replace Amazon Kendra (referencing functionality from [Figure 12-6](#)) and build the architecture from the ground up.

#### *Text Preprocessing Service*

This service takes care of text filtering and removes unwanted stop words, and then creates a forward index with a relevant set of words. This is not a straightforward implementation of direct removal of words and can include multiple options to consider. For example, you can perform a Google search on entire sentences by entering it in double quotes (and to support these search queries, the system needs to index the entire sentence).

#### *Inverted Index Creation Service*

This service is responsible for taking input as a forward index and creating an inverted index for the content. The created inverted indices are uploaded to Amazon S3 and downloaded by Inverted Index Reader Service.

#### *Inverted Index Reader Service*

This service is responsible for downloading the inverted indices and handling the read queries from the Search Keyword Parser Service.

#### *Content Relevance Service*

This service is responsible for hosting algorithms to get the most relevant content for given search keywords. There can be multiple services responsible for content relevance and each can work on different sets of criterias, such as geographical proximity or personal user interests.

#### *Search Keyword Parser Service*

This service is responsible for taking the user input and parsing into the set of keywords for further use by content relevance service and Inverted Index Reader Service. The results are combined and returned to the Frontend Service.

#### *Frontend Service*

This service is responsible for taking user requests and taking actions such as apply rate limiting, discard unwanted traffic, etc. This is essentially a gatekeeper service which only forwards actual traffic to backend services.

There are a lot of things that can happen in the system and the system should be resilient to handle the fault scenarios gracefully. You can build confidence by intentionally breaking the production environment on known parameters and seeing if the system is able

to handle the faults. This practice is referred to as chaos engineering; it is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production. The experiment can be as simple as an EC2 instance being down; is it replaced by another one? In the AWS cloud environment, you could use [AWS Fault Injection Simulator](#) (FIS) service. There are also a lot of open source alternatives available to run these experiments.

Let's incorporate all the components and complete the final architecture on AWS in the next section.

## Day N Architecture

You can definitely start with a single AWS account but for day N architecture, we recommend breaking the architecture into sub components with proper resource isolation—as we already did in the previous sections. To start with, let's assume there are two teams in the organization, owning a web crawler and search engine system independently (there can be more; for example, teams owning machine learning initiatives, infrastructure management or subteams in web crawler and search engine itself). Both the teams deploy their architecture into separate AWS accounts managed under the parent AWS account.

There are no strict guidelines on how many AWS accounts you should have. Different organizations take different approaches based on their requirements. For example, it is very common for many organizations to have separate accounts for each microservice (and even different AWS accounts at the regional level for multi region deployment of critical services) while there are organizations with a single AWS account managing all the resources across the microservices.

In chapter 14, we mentioned having multiple AWS accounts can increase data flow costs, but it also simplifies certain other things

such as billing, complete resource isolation, development flexibility for teams, etc. Similar to previous sections, we'll present the final architecture broken down into two subsystems, starting with web crawler.

## **Web Crawler**

Let's revisit [Figure 12-5](#) and figure out AWS services and resources that can be used to deploy the suggested architecture on AWS cloud, starting with multiple databases used. The web crawler research papers suggest different data structure implementations to store and reference the data, though we'll evaluate if existing storage options on AWS can be used to suffice the use case as described in Table 15-1.

T  
a  
bl  
e  
1  
2  
-  
1.  
W  
e  
b  
cr  
a  
w  
le  
r  
d  
at  
a  
st  
o  
re  
s

---

<b>Use case</b>	<b>AWS Service</b>	<b>Comments</b>
Seed URL	Amazon DynamoDB	The URL Frontier system takes the URLs from this system to begin the crawling process. Amazon

DDB is a preferred choice due the fact the system maintains a direct mapping (key value pair from the URL to additional metadata) and no requirements of data relationships and strong consistency over the database indices.

---

URL Storage and URL uniqueness	This stores all the list of the URLs system figures out during the crawling of web pages. The URLs are added to this database after checking if it's already not present.
--------------------------------	---

URL Filter	This DB maintains the blocked URLs the crawler system doesn't wish to crawl.
------------	--

Robots Cache	Amazon DynamoDB and/or Amazon ElastiCache	Both these caching solutions help to save time during the crawls. Amazon ElastiCache can be used single handedly as well to improve the latency with disk persistence (to recover data in case of node failures).
--------------	---	---

- Robots cache helps the system figure out any

exclusions on subsequent crawls even before crawling.

- DNS cache maintains the IP address of the web server without making the round trip of the internet.

## DNS Cache

---

Webpage Content

Amazon S3 and Amazon DynamoDB

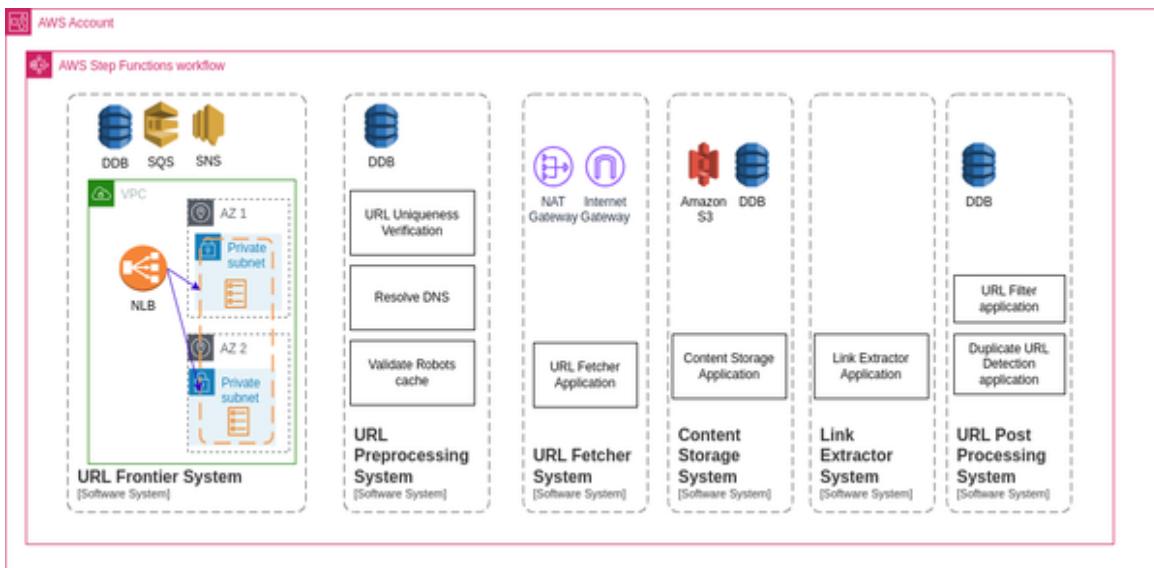
Amazon S3 is an object storage solution which is a preferred choice for storing content like text files, images, videos, etc. so we can utilize the same to store the web pages. Amazon DynamoDB can be helpful to store any metadata such as URL to S3 file path for web page content (helpful across the web page crawls).

The URL Frontier system implementation requires queues for URL processing so it should be able to push URLs to the queue(s) and it

can be consumed by the web crawler system for further processing. We'll leverage a combination of Amazon SQS and Amazon SNS for this purpose; Amazon MSK can also be used for the same purpose as well. The URL Frontier publishes URLs to a single SNS topic and it can route subscribed SQS FIFO queues based on priority attribute in the message. The URL Fetcher system requires additional NAT gateway and internet gateway to communicate with the internet to retrieve the web pages from the internet.

Talking about the overall system architecture, the entire crawler processing is essentially a sequence of steps and it makes sense to use a state machine such as AWS SFn or Amazon MWAA (discussed in chapter 12) at large scale as well. We can definitely build our own state machines but using these AWS services help us to leverage the inbuilt debugging features along with error handling mechanisms. The web crawler system architecture can well fit as an event driven system. A system completes its processing and announces the completion status; the other system takes this output as input and begins its processing and so on. We can also achieve this via SNS-SQS architecture instead of AWS SFn. AWS SFn has an edge over the SQS-SNS because of in-built feature capabilities such as error handling and ease of debugging.

The selection of compute options is another important factor for the application deployment, and we recommend taking the learnings from chapter 11 for different types of compute. In short, there is no perfect answer and we try to take a middle ground by choosing the Amazon EKS EC2 deployment option. We consider this a middle ground based on our previous experience with the technologies and then traffic analysis or quota limits enforced by AWS. Similar to chapter 14, for better resource isolation, we'll deploy the microservices in different VPCs and use a separate Amazon EKS cluster for each of the microservices, as shown in [Figure 12-10](#).



*Figure 12-10. Web Crawler architecture*

## NOTE

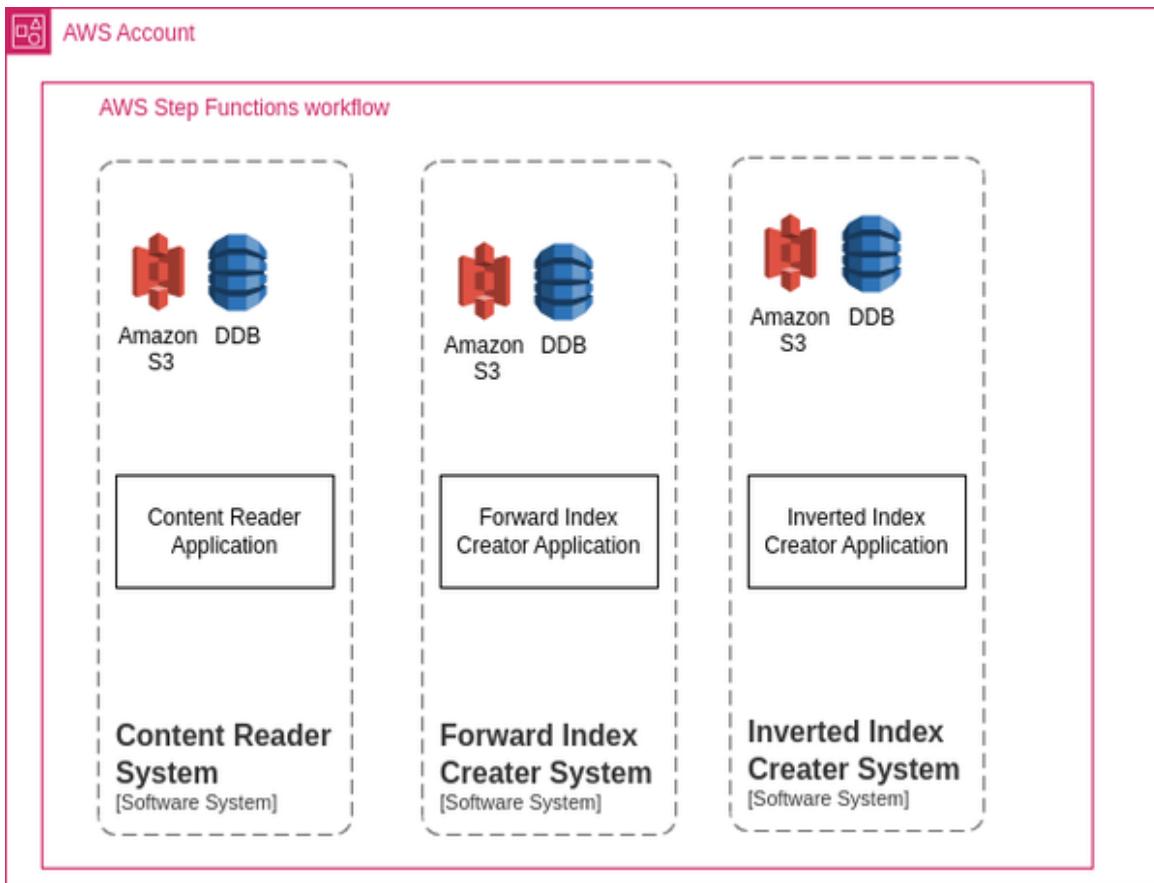
The AWS services for monitoring, access management, and logging are not included in the architecture in order to keep the diagram less cluttered. Please assume them as part of any architecture for overall observability.

As the web pages are ingested into the content storage system, the web crawler system notifies the search engine system about the new content. On receiving this notification, the search engine system begins its processing of creation of indices. We'll discuss the AWS architecture of the search engine system next.

## Search Engine

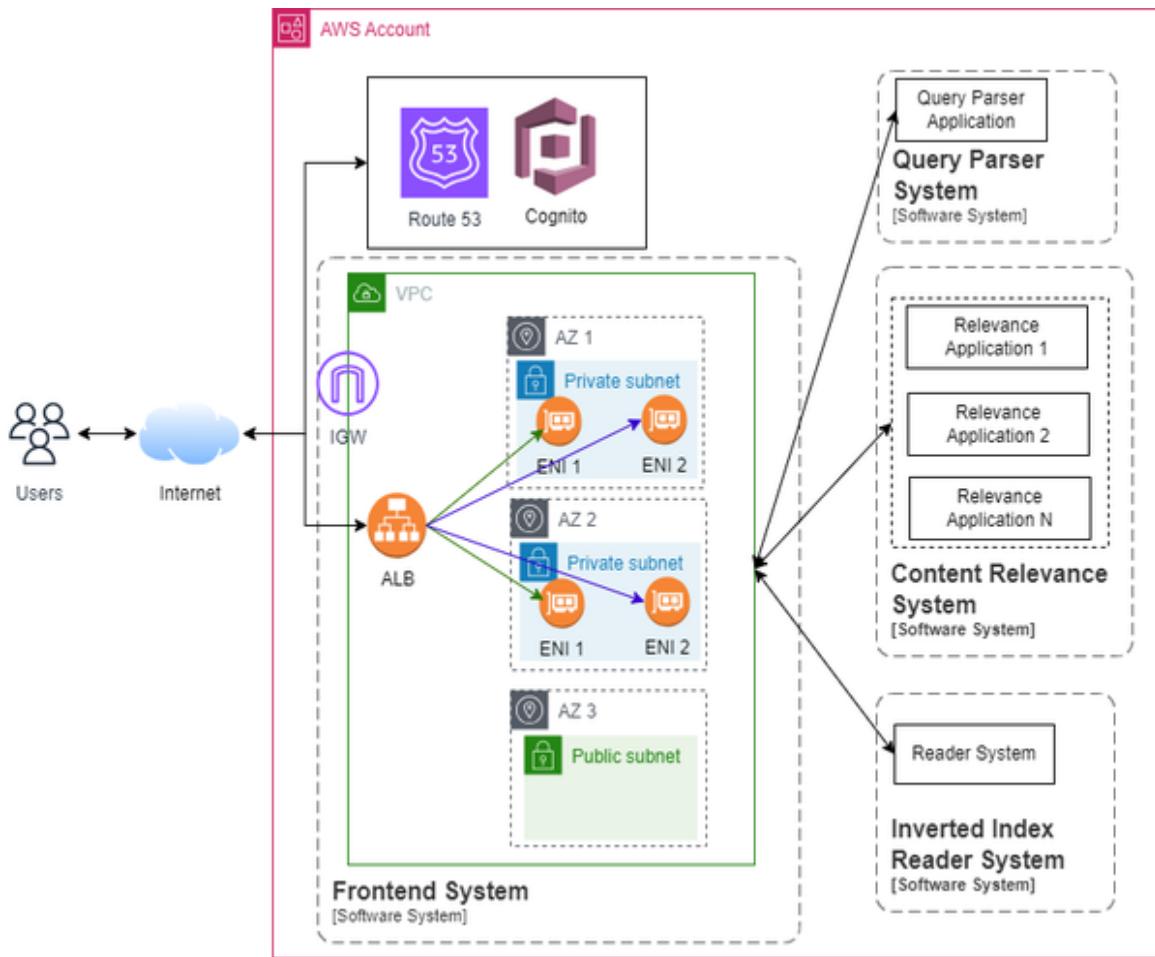
We described the high level architecture of the search engine system in [Figure 12-6](#) and microservices in the 'Scaling to Millions and more' section. We'll go with a framework similar to the web crawler system for application deployment. For creating forward and inverted indices, we'll not be relying on any of the AWS services such as Amazon OpenSearch for search functionality and instead will create custom solutions following the guidelines laid out in the 'Designing

Search Engine' section. [Figure 12-11](#) and [Figure 12-12](#) describes the search engine system's architecture on AWS cloud.



*Figure 12-11. Search Engine architecture for Inverted Index Creation*

The Content Reader Application is responsible for reading the content from a repository stored by the web crawler system and performing preprocessing actions and making it ready for forward index creation. We'll use Amazon S3 to store large files such as web pages, forward and inverted indices and use them across the system. Amazon DDB is used for any metadata storage requirements in the architecture. The Forward Index Creator System constructs forward indices using the output from the previous step and this is being used as input to Inverted Index Creator System to construct inverted indices.



*Figure 12-12. Search Engine System for serving user queries*

The Inverted Index Reader Application is responsible for downloading the inverted indices created by Inverted Index Creator System and allowing reads on top of it. Content Relevance System consists of multiple applications to figure out relevance of search results and can utilize services such as Amazon Sagemaker for requirements of running any ML algorithms. The Frontend System is responsible to take in user queries, preprocess them with the help of Query Parser System and then return results with the help of relevance and reader system.

Let's conclude the chapter with key takeaways and quick summary.

## Conclusion

Web crawler and search engine systems are complex architectures and require a lot of iteration to crawl the web efficiently and present the most relevant search results to the users. The content relevance algorithms hold a very important place in search engines because users care about the good search results only and nothing else. This is true for any system design—as long as we’re able to serve users in the best way possible, nothing else matters. The design decisions should include user inputs as concrete functional requirements.

Over the years, there have been multiple papers published on distributed web crawlers and search engines and we recommend reading them before implementing one of your own. The reading helps in avoiding reinventing the wheel and taking inspiration from already done work. We had a couple of critical comparisons while choosing a technology for the system architecture such as in-memory inverted indices vs on-disk inverted indices, using Amazon OpenSearch vs building custom solutions, etc. We recommend creating a list of pros and cons mapping to your system requirements to finalize on a solution from multiple available choices.

In the next chapter, we’ll take design decisions to architect a social networking system such as Facebook, LinkedIn, etc.

## About the Authors

**Jayanth Kumar** is a Software Development Manager at Amazon, where he is currently building large-scale software systems. Kumar is a millennial polymath, published poet, certified AWS Solutions Architect Professional, entrepreneur, engineering leader and an assistant professor. He earned his bachelorâ€™s degree from IIT Bombay and his masterâ€™s degree from UCLA, where he studied Multi-Objective Deep Learning. He formerly held software engineering positions at SAP Germany and Silicon Valley. Later, as an entrepreneur, he held the positions of Head of Engineering at Goodhealth and Engineering Manager at Delhivery, an Indian unicorn company. He is always seeking a challenge and new opportunities for learning, and focuses on building robust mechanisms and systems that will stand the test of time.

**Mandeep Singh** is a Software Engineer at Jupiter. He fell in love with Cloud, AWS Technologies and Distributed Systems in the role of Big Data Cloud Support Associate at AWS and as a Software Engineer at Amazon. He supports the learning and development of others by sharing lessons on Cloud and Distributed Systems on his YouTube channel. He enjoys morning runs, weight lifting, cooking and spending time with his family in the serene ambience of his hometown.