**Chapter 6. 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*.

While Chapter 2 and Chapter 3 of this book illustrate multiple storage types and solutions, this chapter explores AWS-specific storage [services](https://aws.amazon.com/products/storage/), 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](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html) 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](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html) 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](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html) 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](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html) 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. Theirmajor 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. Theirmajor 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](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volumes-multi.html) volumes feature allowing users to attach a single Provisioned IOPS SSD volume to max of sixteen [Nitro-based EC2](https://aws.amazon.com/ec2/nitro/) 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](https://docs.aws.amazon.com/efs/latest/ug/performance.html) 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](https://docs.aws.amazon.com/efs/latest/ug/storage-classes.html) 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](https://www.amazonaws.cn/en/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*](https://www.amazonaws.cn/en/fsx/windows/)

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](https://www.amazonaws.cn/en/fsx/lustre/" \t "_blank)*

This is built on a popular high-performance file system, [Lustre](https://www.lustre.org/" \t "_blank). 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](https://docs.aws.amazon.com/AmazonS3/latest/userguide/storage-class-intro.html) 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](https://docs.aws.amazon.com/kms/latest/developerguide/overview.html) (KMS) referred to as [SSE-KMS](https://docs.aws.amazon.com/AmazonS3/latest/userguide/specifying-kms-encryption.html). Additionally, you can configure client-side encryption as well for more security.

*Object Lock*

[Object Locks](https://docs.aws.amazon.com/AmazonS3/latest/userguide/object-lock-overview.html) 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](https://docs.aws.amazon.com/AmazonS3/latest/userguide/access-analyzer.html) for S3 to identify buckets that grant public access.

*IAM Policy*

[IAM Policy](https://aws.amazon.com/blogs/security/writing-iam-policies-how-to-grant-access-to-an-amazon-s3-bucket/) is helpful in granting users or groups read-write access to S3 resources.

*Amazon Macie*

Amazon [Macie](https://aws.amazon.com/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](https://docs.aws.amazon.com/AmazonS3/latest/userguide/acl-overview.html) 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](https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-iam-policies.html) 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](https://docs.aws.amazon.com/AmazonS3/latest/userguide/example-bucket-policies.html) 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](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_CreateDBInstance.html) 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](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.DBInstanceClass.html) 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](https://aws.amazon.com/rds/aurora/getting-started/) 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](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/data-api.html) 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](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.Encryption.html) and [Aurora](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Overview.Encryption.html) 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.

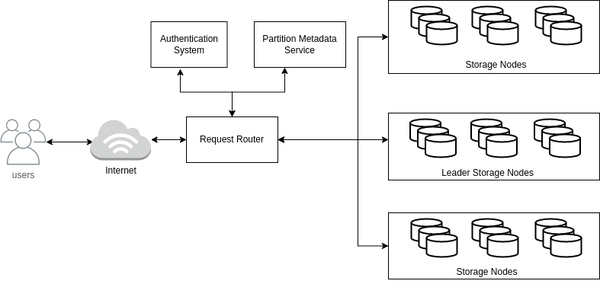
Below are the keys responsible for data storage on DDB:

*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 6-1](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch06.html#fig_1_dynamodb_internal_architecture). 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 6-1. 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](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html) 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](https://aws.amazon.com/dynamodb/pricing/provisioned/) 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](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html) 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.

|  |  |  |
| --- | --- | --- |
| 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. |
| Table 6-1. Differences in LSI and GSI | | |

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](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/transaction-apis.html) 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](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html) 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](https://docs.aws.amazon.com/documentdb/latest/developerguide/document-database-documents-understanding.html) 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](https://docs.aws.amazon.com/documentdb/latest/developerguide/document-database-use-cases.html), 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.

|  |  |
| --- | --- |
| SQL | MongoDB/DocumentDB |
| Table | Collection |
| Row | Document |
| Column | Field |
| Primary Key | Object Id |
| Table 6-2. SQL vs DocumentDB terminology | |

Amazon DocumentDB is [MongoDB](https://www.mongodb.com/) compatible with support for powerful ad-hoc queries and comes with [transaction](https://docs.aws.amazon.com/documentdb/latest/developerguide/transactions.html) 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](https://docs.aws.amazon.com/documentdb/latest/developerguide/functional-differences.html). Unlike with DDB, we need to specify the [instance class](https://docs.aws.amazon.com/documentdb/latest/developerguide/db-instance-classes.html) 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.

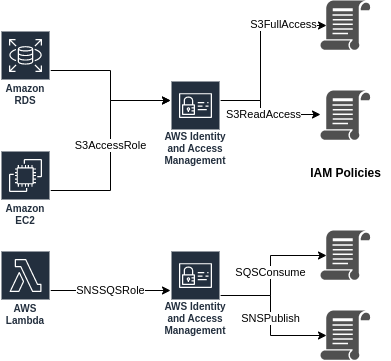
AWS also provides [migration](https://docs.aws.amazon.com/documentdb/latest/developerguide/docdb-migration.html) tools such as AWS Database Migration Service (DMS) to facilitate easier migration of MongoDB workloads from on-premise or EC2 servers to DocumentDB. The storage business use cases can vary 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](https://aws.amazon.com/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](https://tinkerpop.apache.org/) or [OpenCypher](http://opencypher.org/" \t "_blank) for property graph databases and [SPARQL](https://www.w3.org/TR/rdf-sparql-query/) for Resource Description Framework (RDF) format graph databases. AWS recommends consulting the GitHub repository “[AWS Reference Architectures for Using Graph Databases](https://github.com/aws-samples/aws-dbs-refarch-graph)” 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 6-2](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch06.html#fig_2_graph_representation_of_iam_role_entities_associat) 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 6-2. Graph representation of IAM role-entities association**

For a step-by-step guide to cluster setup, you can refer to the [AWS documentation](https://docs.aws.amazon.com/neptune/latest/userguide/manage-console-launch-console.html). 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 ⅔ of memory of instance to cache.
* [Lookup](https://docs.aws.amazon.com/neptune/latest/userguide/feature-overview-lookup-cache.html) 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](https://docs.aws.amazon.com/neptune/latest/userguide/gremlin-results-cache.html) 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](https://d0.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf) 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](https://docs.amazonaws.cn/en_us/AmazonElastiCache/latest/red-ug/GettingStarted.CreateCluster.html" \t "_blank) and [Memcached](https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/GettingStarted.CreateCluster.html" \t "_blank).

*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 6-3](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch06.html#fig_3_redis_data_tiering_support). 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](https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/AutoFailover.html) 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*

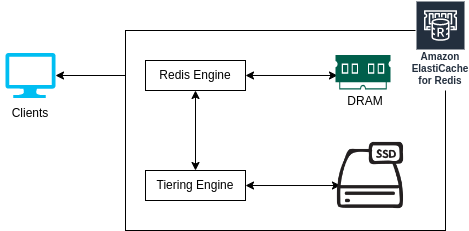
Redis is a single-threaded process when incoming requests are handled sequentially, while Memcached is multi-threaded, meaning it makes good use of larger Amazon EC2 instance sizes with multiple cores.

*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](https://en.wikipedia.org/wiki/Payment_Card_Industry_Data_Security_Standard)), Health Insurance Portability and Accountability Act([HIPAA](https://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act)) and Federal Risk and Authorization Management Program([FedRAMP](https://en.wikipedia.org/wiki/FedRAMP" \t "_blank)) 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.

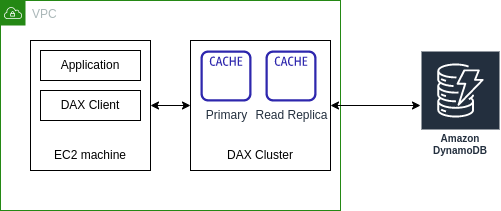


**Figure 6-3. Redis Data Tiering Support**

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](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DAX.concepts.html" \t "_blank) (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 6-4](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch06.html#fig_4_dax_cluster_setup). 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 6-4. 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](https://github.com/opensearch-project) 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](https://opensearch.org/docs/latest/tuning-your-cluster/cluster/). 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](https://docs.aws.amazon.com/opensearch-service/latest/developerguide/managedomains-dedicatedmasternodes.html) 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](https://docs.aws.amazon.com/opensearch-service/latest/developerguide/managedomains-configuration-changes.html) 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](https://docs.aws.amazon.com/opensearch-service/latest/developerguide/saml.html) or [Amazon Cognito authentication](https://docs.aws.amazon.com/opensearch-service/latest/developerguide/cognito-auth.html).

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](https://cassandra.apache.org/_/index.html) 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](https://docs.aws.amazon.com/keyspaces/latest/devguide/functional-differences.html) 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.

|  |  |
| --- | --- |
| 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 |
| Table 6-3. Storage Service identification for Cafe Delhi Heights | |

**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.